Universities of Burgos, León and Valladolid

Master's degree

Business Intelligence and Big Data in Cyber-Secure Environments

Thesis of the Master's degree in Business Intelligence and Big Data in Cyber-Secure Environments

Extraction, transformation, loading and visualization of combined Twitter and Spotify data in a scalable architecture

Presented by Adrián Riesco Valbuena
in University of Burgos — July 7, 2022
Supervisor: Álvar Arnaiz González

Universities of Burgos, León and Valladolid

Master's degree in Business Intelligence and Big Data in Cyber-Secure Environments

D. Álvar Arnaiz González, professor of the department named Computer Engineering, area named Computer Languages and Systems.

Exposes:

That the student Mr. Adrián Riesco Valbuena, with DNI 71462231N, has completed the Thesis of the Master in Business Intelligence and Big Data in Cyber-Secure Environments titled "Extraction, transformation, loading and visualization of combined Twitter and Spotify data in a scalable architecture".

And that thesis has been carried out by the student under the direction of the undersigned, by virtue of which its presentation and defense is authorized.

In Burgos, July 7, 2022

Approval of the Supervisor:

D. Álvar Arnaiz González

Resumen

Este proyecto consiste en el desarrollo de un proceso de Extracción, Transformación y Carga para capturar datos de las APIs de **Twitter** y **Spotify** APIs utilizando herramientas de software del ámbito del Big Data y con un ciclo de vida gestionado mediante metodologías ágiles. El proceso ETL consiste en la recolección de los tuits con el hashtag `#NowPlaying` de la API de Twitter, la limpieza del texto y el aislamiento de los nombres de las canciones y artistas (eliminando caracteres fuera del alfabeto latino, hashtags, urls y menciones), la consulta a la API de Spotify para recoger los datos de las canciones, la combinación de ambos conjuntos de datos, su envío al Data Warehouse, su posterior consulta desde el back-end de la aplicación web y, finalmente, su presentación al usuario final en un front-end personalizado. Para servir los datos en el front-end se han diseñado dos visualizaciones, una consistente en una **tabla** que permite filtrar, ordenar y ocultar/mostrar columnas, y la otra consistente en una combinación de un formato de **barras** y con uno de **línea**, con un selector para cada uno de ellos, permitiendo al usuario ordenar por cualquier columna y cambiar la cantidad de datos mostrados. Las herramientas utilizadas son **Apache Airflow** como orquestador del flujo de datos, **Apache Spark** como ejecutor del proceso ETL, **Apache Cassandra** como almacén de datos, y una combinación de **Flask** y **Bootstrap**, junto con **Chart.js** y **Datables**, para la creación de la aplicación web personalizada. Todos los servicios se han encapsulado en contenedores dentro de la misma red utilizando **Docker Compose** como orquestador.

Descriptores

Airflow, Apache, API, Big Data, Bootstrap, Cassandra, Chart.js, Datatables, Docker, Docker Compose, ETL, Flask, Spark, Spotify, Twitter.

Abstract

This project consists of the development of an Extraction, Transformation and Loading process to capture data from **Twitter** and **Spotify** APIs using software tools from the Big Data domain and with a life cycle driven by agile methodologies. The ETL process consists of collecting from the Twitter API the tweets with the hashtag `#NowPlaying`, cleaning the text and isolating the track and artist names (removing characters outside the Latin alphabet, hashtags, urls and mentions), querying the Spotify API to collect the track data, combining both data, sending it to the Data Warehouse, querying it from the back-end of the web application and, finally, serving it to the end user in a custom front-end. Two visualizations have been designed to serve the data in the front-end, one consisting of a **table** that allows filtering, sorting and hiding/showing columns, and the other consisting of a combination of a **bar** chart and a **line** chart, with a selector for each format and allowing the user to sort by any column and change the amount of data displayed. The tools used are **Apache Airflow** as flow orchestrator, **Apache Spark** as ETL process executor, **Apache Cassandra** as Data Warehouse, and a combination of **Flask** and **Bootstrap**, together with **Chart.js** and **Datatables**, to create the custom web application. All services have been encapsulated in containers within the same network using **Docker Compose** as the orchestrator.

Keywords

Airflow, Apache, API, Big Data, Bootstrap, Cassandra, Chart.js, Datatables, Docker, Docker Compose, ETL, Flask, Spark, Spotify, Twitter.

Contents

List of Figures

List of Tables

Report

Introduction

Social networks are currently a fundamental aspect of society. People usually use social networks to share experiences, opinions, and aspects of their lives and interact with other people. Using social networks as a data source we can access a huge amount of information and be able to build accurate analyses on practically any topic.

Another aspect that has gradually permeated our society is the concept of subscriptions to services, be it music, movies, games, or almost any concept that we can think of. Not many years ago, the concept of paying for subscriptions to services, where you do not actually own the content you pay for and instead get temporary access whose duration is defined by how long you continue to pay for the subscription, was relegated to very specific services and was not nearly as globalized as it is today.

The global acceptance of subscription as a service is reflected in social networks, where users can comment on the different music, movie, and game platforms, turning each new release into a social phenomenon. In order to take advantage of both worlds, this project uses the social network Twitter to obtain information about the latest music listened to by users (by searching for a particular hashtag) and then consult the data of the song and the artist involved that Spotify, a platform based on music as a service, has.

For the development of the project, a series of scripts have been designed for the execution of the ETL (Extract, Transform and Load) process aimed at consuming the data from both APIs, processing and loading them into the data warehouse for subsequent delivery to a customized front end, all based on recognized tools in the Big Data field, such as Apache Airflow, Apache Spark, Apache Cassandra, Docker Compose and Flask, among others.

Project objectives

The initial objectives through which the use case was built consisted in the following points:

- Ability to obtain data in real time.
- Combination of at least two different data sources.
- Potential to scale in both technology and data volume.
- Involvement of various technologies in the Big Data field.
- Use of open source tools.

After a research, the author designed the use case and built the project objectives:

- Build a pipeline to gather information about last songs listened from the **Twitter API**.
- Find information about the songs (name, artist and audio features) through the **Spotify API**.
- Execute all the ETL[1] process in **Apache Spark**.
- Store all the data in a Data Warehouse under a known technology, **Apache Cassandra**.

[1]ETL is the acronym for Extract, Transform and Load, the three phases for data processing

- Visualize the information in a custom front-end and back-end created with **Flask** and **Bootstrap**.
- Orchestrate all the data flow with **Apache Airflow**.
- Develop the project with **Docker** and **Docker Compose** to ensure deployment through heterogeneous environments.

Through these global objectives, the low-level functional and technical requirements were specified, as shown in the appendix B. The detailed use case is described in the section 5.1.

Theoretical concepts

In this section are covered the theoretical concepts in which the project has been based. All concepts are described in a detailed and simple way since this master's thesis can be aimed at technical and non-technical students.

3.1 Big Data

The term **Big Data** is used to refer to data sets whose size exceeds the capabilities of the software systems used to capture, preserve and process the data within an acceptable time window.

The definition of Big Data is usually linked to the three dimensions or Vs defined by Doug Laney in 2001 [1]:

- **Volume.** The amount of data to handle.
- **Velocity.** The rate at which new data is created and consumed.
- **Variety.** The diversity or form of the captured data.

After the definition of the three Vs, more dimensions have been added: Veracity, Value, Validity, Vagueness, Volatility... The author even found an article where 42 different dimensions of Big Data were defined [1].

From the moment Big Data is discovered until it can be used and generate value, it goes through a life cycle made up of the following phases: discovery, raw data ingestion, raw data processing, storage, integration with other data, analysis, and presentation of results.

Due to the complexity of big data, it usually requires a complex and varied technological ecosystem to be managed.

3.2 ETL

An **ETL** process refers to the stages that every data processing exercise usually goes through:

1. **Extract.** Raw data is collected from the source and fed into our Data Lake, which preserves it in its original form.

2. **Transform.** The raw data is cleaned and transformed to be used in our environment.

3. **Load.** Once structured and filtered, the data is entered into our Data Warehouse.

3.3 API

An **Application Programming Interface** or **API** is an interface that defines the interactions that can be made with a software system. The APIs generally define the data that can be requested and sent to the system, the way to authenticate to it and the format of the returned data [2].

In relation to web development, most of the APIs work according to Hypertext Transfer Protocol (HTTP), a communication protocol that allows information transfers through files on the World Wide Web. Additionally and not exclusive, a large number of APIs are developed according to the REST architectural style, defined by Roy Fielding in the year 2000 and which is based on a series of principles that seek to facilitate development:

1. **Uniform interface** for all resources, forcing all queries made to the same resource (each with a specific Uniform Resource Identifier or URI) to have the same form regardless of the origin of the request.

2. **Decoupling between the client and the server**, making the only information that the client must know about the server is its identifier (URI) and that the only action to be carried out by the server is to return the data required in the request.

3. **Stateless queries**, meaning that each request must contain all the information necessary to be processed without requiring an additional request or storing any type of state.

4. **Allow**, whenever possible, both **client-side and server-side caching** to reduce the load of the former and increase the scalability of the latter.

5. **Layer system**, allowing multiple intermediaries between the client and the server and preventing them from knowing in any case if they are communicating with the other party or with an intermediary.

6. Although the resources exchanged are usually static, a REST architecture can optionally have **responses that contain snippets of executable code**.

In general terms, an API based on a REST architecture serves to make it easier for developers to develop applications that interact with the resources published by it.

3.4 Orchestrator

The great variety of applications and services that exist in technological environments, where there are workflows with various actors with interdependencies between them, make their management and automation enormously complex [3]. The more complex a system is, the more difficult it is to manage the intervening factors.

System automation usually improves efficiency, simplifies management and reduces associated costs, both in terms of time spent and personnel required to control it. Related to automation is orchestration, which, while the scope of the former is limited to a single task, the latter comprises multi-step processes and workflows.

Orchestrators are software systems that focus precisely on the orchestration of a multitude of processes, workflows and tasks. Two main orchestrators have been used in this project: **Docker Compose**, which acts as an orchestrator for the work environment containers, and **Apache Airflow**, which orchestrates the project's workflow.

3.5 NoSQL Databases

A **database** is a set of data belonging to the same context and stored for later use, and can be updated periodically. The best known type of databases are relational databases. In a relational database, the data attributes are

stored in the form of columns, previously defined, and the values are stored in the rows of the table for all its columns or attributes. These databases have an associated query language called SQL (Structured Query Language).

Relational database properties are summarized in **ACID** properties:

- **A**tomicity. The process is done completely or it is not done.
- **C**onsistency. Only valid data is written.
- **I**solation. The operations are performed one at a time.
- **D**urability. When an operation is performed, it persists and is not undone even if the system crashes.

However, in the face of the massive volumes of data that are associated with the concept of Big Data, relational databases have a series of limitations:

- Reading the data is **costly**, as data is represented in tables, queries involve joining large data sets and filtering the results.
- The stored information usually has **similar structures**, a concept that does not agree well with Big Data, where the variety of data structure is greater.
- **Scalability** is not their strongest factor, since they were initially designed with a single server in mind or, at most, with replicas and load balancing.

Distributed databases are limited by the CAP theorem:

- **C**onsistency. The information remains coherent and consistent after any operation, with all copies having the same data at all times.
- **A**vailability. The system continues to function even if any of its nodes or parts of the software or hardware fail, and all reads and writes complete successfully.
- **P**artition tolerance. The system nodes will continue to function even if the connection between them fails or messages are lost, maintaining their properties.

According to CAP's theorem, any distributed database with shared data among its nodes can have at most two of the three properties at the same time. This theorem resulted in databases with relaxed ACID properties, that is, with BASE properties:

- **B**asically **A**vailable: The store works most of the time, even if failures occur.
- **S**oft-State: Stores or their replicas do not have to be consistent at all times.
- **E**ventually Consistent: consistency happens eventually, as it is something that is taken for granted at some point in the future. All copies will gradually become consistent if no further updates are run.

Non-relational databases or **NoSQL** (Not Only SQL) is a term introduced by Carl Strozzi in 1998 that describes all those databases that do not follow the same design patterns as relational databases. Non-relational databases follow the **BASE** properties and have advantages such as:

- They do not require a fixed data schema.
- The data is replicated on multiple similar nodes and can be partitioned.
- They are horizontally scalable, that is, by adding new nodes.
- They are relatively inexpensive and simple to implement, with a host of open source alternatives.
- They provide fast read and write speeds, with fast key-value access.

However, the main disadvantages of non-relational databases is that they do not support certain features of relational databases (join, group by, order by...) except within their partitions, they do not have a query language standard such as SQL and its relaxed ACID or BASE properties give lesser guarantees.

Non-relational databases are mainly divided into four groups:

- **Key/value**. Their data model is very simple, since they only store keys and values. They are very similar to a hash table, they are fast, they have great ease of scaling, eventual consistency and fault tolerance, although they cannot support complex data structures.

- **Column oriented**. Data is stored in columns instead of rows. The data is semi-structured, easily distributable, provides high reading speed, calculations on attributes are faster (especially aggregations such as averages) and are perfect when you want to do many operations on large data sets, but they are not the most efficient for writing or when you want to retrieve all records. The data model has columns, super columns, column families, and super column families.

- **Document oriented**. These are key-value stores in which the value is stored as a document with a defined format, so the final data model is collections of documents with a key-value structure (JSON, PDF, XML...). They are schema-less, highly scalable, programmer-friendly, and support rapid development.

- **Graph oriented**. They represent information as the nodes of a graph and their relationships as edges, using graph theory to traverse it. Their strength is the analysis of the relationships between their objects and they represent hierarchical information very well, but they are not particularly good for scaling and tend to have a higher learning curve.

3.6 Containers

Containerization is the packaging of code together with its dependencies, configurations and libraries to form a lightweight executable that can be executed in any infrastructure regardless of its system [4]. In this way, developers can focus on developing applications safely and quickly without worrying about subsequent execution, since the code they develop will be compiled into a package with all its dependencies, abstracting it from the operating system, isolating it and making it portable.

The main advantages of containerization are summarized in:

- **Portability**. The container's abstraction from the host operating system allows it to run consistently on any platform.

- **Agility**. The emergence of open source container engines like Docker has made it easier to integrate with DevOps elements and to run on different operating systems.

- **Speed**. Containers are light and fast to run due to their lack of an operating system and their limited content.

- **Fault isolation**. Each container is isolated and runs independently of the rest, so failures do not propagate between them.

- **Efficiency**. The container software shares the kernel of the operating system of the machine and the application layer can be shared between containers, making better use of system resources.

- **Ease of management**. Orchestrators make it extremely easy to install, manage, scale, and maintain containers, and the simplicity of containers also works in its favor.

- **Security**. Isolating containers acts as a security barrier against the spread of malware throughout the container environment.

A container is considerably lighter than a virtual machine, since it contains only an application and the elements necessary for its execution, while virtualization includes the entire operating system. The container has an engine to be executed and an orchestrator is usually used to manage several containers and their interconnections.

3.7 Continuous Integration / Continuous Delivery

Continuous integration / continuous development or **CI/CD** is a software development and delivery method based on the introduction of automation in the stages of the development process, allowing work on iterables of the project subjected to testing phases. Continuous integration refers to the automation of development processes and building iterations, while continuous development refers to the continuous delivery of software and its deployment to the production environment [5].

A well-constructed CI/CD cycle helps developers merge new changes with the original project, as well as validate changes to ensure no new deficiencies or bugs are introduced into the product. Each functionality added to the main repository is tested both unitarily and functionally in an automated way, including these tests and allowing a quick analysis of possible conflicts before launching the new iteration of the product.

3.8 Template engines

A **template engine** is software aimed at combining templates and data models to produce formatted output by rendering the data on the server side. The template engines transform the variables in the templates with the actual values and send them to the client [6].

Techniques and tools

In this section are presented the methodological techniques and development tools used to carry out the project.

4.1 GitHub

GitHub[2] is a collaborative development platform created in 2008 and based on the Git version control system. Github has a freemium model and provides the ability to host both public and private projects, focusing primarily on code development [7].

GitHub is the repository in which the project has been managed and hosted, and Git is the version control through which the commits have been made from the local environment. For the agile methodology, the Milestones have been used as sprints and the Issues as the tasks to be carried out in each of them.

4.2 Postman

Postman[3] is a tool that allows the user to build and use APIs in a simple way. Some of its characteristics are the API repository (easier storage, cataloging and collaboration), the availability of tools to help in the API design, testing and documentation, the workspaces to organize the work, and built-in integrations with tools such as GitHub, Azure DevOps, Jenkins, Splunk, Slack and Microsoft Teams. In addition, Postman is based on open source technologies, which provides the ability to be easily extended [8].

[2] https://github.com/
[3] https://www.postman.com/

Twitter API

Twitter is an American social network founded in 2006 that allows users to share short posts (280 characters since 2017), known as tweets, and interact with those of other users through replies, likes, retweets or quotes [9]. Although it has recently incorporated additional payment functions, this social network is free to use and is accessible on multiple platforms. Currently, the social network has 217 million active users daily [10].

On the other hand, Twitter is also known for giving certain facilities to developers to make their products interact with the platform. The company has a Twitter Developer portal where a multitude of resources and useful documentation are posted [11]. This portal contains a description of the API that Twitter offers, how to authenticate, the different endpoints to which queries can be launched, and the associated usage limits.

The **Twitter API**, currently in its second version, allows the user to request and receive a wide variety of data. Depending on the query launched, the user can receive a series of different objects, each with its own fields and parameters:

- **Tweets**. It represents the basic block of communication between Twitter users.
- **Users**. It represents a user account and its metadata.
- **Spaces**. It represents a space (virtual places in Twitter where users can interact in live conversations) and its metadata.
- **Lists**. It represents a Twitter list (used to configure information visualized in the timeline) and its metadata.
- **Media**. It represents any image, video or GIF attached to a tweet and can be obtained by expanding the Tweet object.
- **Polls**. It represents a poll (choices, duration, end-time and results) and can be obtained by expanding the Tweet object.
- **Places**. It represents a place identified in a tweet and can be obtained by expanding the Tweet object.

Of all the endpoints available in the API (manage tweets, user lookup, search spaces, full-archive tweet search...), in this thesis only the Recent Search endpoint has been used, which returns a list of the most recent tweets

based on the rules entered in the query. Both the number of tweets to receive and the id or date of the oldest tweet to be returned can be specified, and this endpoint allows receiving up to one hundred tweets per query and includes a pagination token to handle larger results [12].

Spotify API

Spotify is a company of Swedish origin founded in 2006 that provides audio streaming services, currently being one of the companies with the largest number of users among all those that have this type of service. Spotify has a catalog made up of music and podcasts, including more than 82 million songs, distributed through a free service (limited control and periodic announcements) with the option of a premium subscription. Its business model is based on advertising and paying users, and it pays royalties to artists based on the proportion of streaming of their songs compared to the total played [13].

Spotify has a developer portal that provides a wealth of documentation to help design and implement various use cases. Spotify has an API based on a REST architecture with different published endpoints that return metadata of artists, albums and songs from its own catalog, as well as information on users, lists and music saved by them, in JSON format [14].

The **Spotify API** has several endpoints to which queries can be sent to collect or modify information: Albums, Artists, Shows, Episodes, Search [15], Tracks [16], Users, Playlists, Categories, Genres, Player and Market. During this thesis the following have been used:

- **Search**. Search for Item allows to obtain information about the Spotify catalog of artists, songs, albums, playlists, shows or episodes. You can specify the type or types of objects to return, in this case being the type "tracks". Only one search can be performed per query.
- **Tracks**. Get Tracks' Audio Features allows to obtain the characteristics of a set of songs specified by their id. The characteristics returned are as follows:
  - **Acousticness**. Confidence measure from 0.0 to 1.0 about whether the track is acoustic, with 1.0 representing high confidence that it is acoustic.
  - **Danceability**. It describes with a value between 0.0 and 1.0 how suitable a track is for dancing based on a combination of its musical elements, with 1.0 being the greatest danceability.

- **Duration_ms**. It represents the duration of the track in milliseconds.
- **Energy**. It represents with a value between 0.0 and 1.0 the conception of the energy level of the track, being 1.0 the maximum energy value.
- **Instrumentalness**. It predicts with a value between 0.0 and 1.0 whether or not the track contains vocals. Values above 0.5 usually represent tracks without vocals, and the closer to 1.0 the more likely they are.
- **Key**. It indicates the key (in a musical context, the dominant scale) the track is in, with each key having an assigned integer starting with 0. If no key is detected, the value is -1.
- **Liveness**. It represents audience presence with a value between 0.0 and 1.0. Values greater than 0.8 indicate a high probability that the track was recorded live.
- **Loudness**. Indicates the average volume of a track in decibels, with values generally contained between -60dB and 0dB.
- **Mode**. Indicates the modality of the track, being the value 1 greater and 0 less.
- **Speechiness**. Detects the presence of spoken words in a track. Values less than 0.33 typically indicate instrumental tracks without vocals, values between 0.33 and 0.66 songs with music and vocals, and values greater than 0.66 podcasts, audiobooks, and similar formats.
- **Tempo**. Indicates the tempo or rhythm of a track in beats per minute.
- **Time_signature**. Represents the estimated time signature value, with values between 3 and 7 indicating 3/4 and 7/4 time signatures, respectively.
- **Valence**. Indicates with values between 0.0 and 1.0 the musical positivity transmitted by the track, where high values indicate greater positivity, while low values indicate greater negativity.

4.3 Apache Airflow

Apache Airflow[4] is a service orchestrator that allows you to plan, manage, and monitor workflows [17]. It was created in 2014 by Airbnb with the aim

[4] https://airflow.apache.org/

of handling the company's huge data flows, and published in 2015 under an open source license. In March 2016 the project joined the Apache Software Foundations incubator and was published as a top level project in 2019.

Airflow is used to automate jobs by breaking them down into smaller tasks. For example, this project uses Airflow to automate the ETL process that consumes data from Twitter and Spotify, processes it, and serves it to the user. Among the main features of Airflow are scalability and ease of integration with other tools.

The main element used by Airflow are the **Directed Acyclic Graphs** or **DAGs**, which are groups of tasks connected to each other through dependencies like the nodes of a graph. The word *direct* indicates that the existing relationships in the graph must only have one direction (bidirectional relationships between nodes or tasks are not allowed), while the word *acyclic* means that cycles cannot exist in the graph (nodes or tasks cannot be executed more than once). Tasks are defined by means of an operator and there is a very extensive library with operators that allow defining a wide variety of services such as BashOperator, to execute Bash commands, or SparkSubmitOperator, to submit a task to Spark. Regarding the programming language, Python is the one in which DAGs are developed.

Airflow allows you to have control of the tasks executed through a record of their executions, the time, the current or final status and the generated logs. In addition, it allows certain parameters to be associated with each task, such as, for example, the maximum execution time allowed.

4.4 Apache Spark

Apache Spark[5] is a multi-language engine that emerged in 2009 at the University of California used for data processing and machine learning designed for both simple single-node and distributed architectures [18]. Apache Spark supports the Python, Scala, SQL, Java and R programming languages and allows to design pipelines for large volumes of data for batch and streaming processing. Currently, Spark is one of the most widely used large-scale data processing frameworks.

Key features of Apache Spark include its ability to scale, its speed, and the multitude of free resources made available by the large Apache community. In addition, Apache Spark has been adapted to the main cloud

[5] https://spark.apache.org/

solutions, such as Databricks, Amazon Web Services, Google Cloud Platform and Microsoft Azure.

The Apache Spark architecture is mainly composed of:

- **Driver.** Turn code into tasks to distribute to worker nodes. This code goes on to form DAGs, which indicate the order of the tasks and the node in which they are going to be executed.
- **Executors.** They run on worker nodes and execute assigned tasks.

Finally, one of the most characteristic elements of Spark is the Resilient Distributed Dataset or RDD. RDDs are abstractions that represent a collection of immutable objects that can be divided among the different nodes of a cluster and executed in a distributed way.

4.5 Cassandra

Apache Cassandra[6] is a widely used open source non-relational database that emerged in 2008 as a Google code open source project and in March 2010 as an Apache top level project. It is based on a system that acts as a mix between key-value and column-oriented and introduces Cassandra Query Language or CQL, an alternative query language to SQL with a similar syntax [19, 20].

Among its main features are:

- **Scalability**. Its performance increases linearly by adding new nodes, which can be added in real time without affecting system performance.
- **Fault tolerance**. Due to its distributed system architecture, data is automatically replicated across multiple nodes and enables replication and redundancy to be provided across multiple datacenters. Cassandra works with partitions, where each node owns a set of tokens whose ranges determine what data to send to each node within the cluster.
- **Decentralization**. All nodes in the cluster are assigned the same role, with no master node that can act as a single point of failure.

[6] https://cassandra.apache.org/_/index.html

- **Consistency**. Considering the CAP theorem (Consistency-Availability-Partition tolerance), Cassandra is built to be AP, that is, to sacrifice consistency to ensure continuous operation without failures. Cassandra allows you to adjust this level of consistency by selecting the minimum number of nodes that must validate a read or write operation before it is considered successful.

Cassandra was used in the project as the main Data Warehouse to store Twitter and Spotify data.

4.6 Flask

Flask[7] is a Python-based open-source web framework created in 2004 and used to build web applications [21]. It is considered a micro web framework because it does not lay on other tools or libraries to work, building a solid core and providing the capability to extend its operation. Flask uses Jinja as its template engine.

During development, Flask v2.1.2 has been used to build the backend of the web application, gathering the data from the database and routing the user to the three existing views.

4.7 Bootstrap

Bootstrap[8] is an open-source CSS (Cascading Style Sheets) framework whose objective is to facilitate the design of responsive web interfaces [22]. Bootstrap bases its operation on a grid system with divisions of up to twelve columns, six responsive levels and a multitude of classes and configurations.

During development, Bootstrap v5 has been used to organize the visual appearance of the three views of the web application and control the placement and responsiveness of its elements to screen size variations.

4.8 Chart.js

Chart.js[9] is an open source Javascript library used to make charts using HTML code [23]. The use of this library is very simple, it has a large user

[7] https://flask.palletsprojects.com/en/2.1.x/
[8] https://getbootstrap.com/
[9] https://www.chartjs.org/

community and, although it does not have a wide variety of graph types, they are enough to satisfy basic visualization needs.

During development, a Chart.js v2.5.0 mixed graph with data in bar and line formats has been used to build the final visualizations.

4.9 Datatables

Datatables[10] is an open source plugin for the jQuery javascript library that provides an easy and flexible way to design HTML-based data tables [24]. Among the features that can be added are the selection of the number of rows to display, pagination, multi-column sorting, search and customization of colors and formats.

During development, Datatables v1.12.1 was used to create the final display in table format, complete with the ability to hide and show table columns.

4.10 Docker

Docker[11] is an open-source tool that emerged in 2013 that allows you to add an abstraction layer to the deployment of applications by automating them in a container from software [25]. Through Docker, software development can be simplified and applications can be executed in different work environments more easily.

The containers raised by Docker are associated with an image. To build an image in Docker, a file called Dockerfile with a specific syntax is used. Once built, the image will contain the Dockerfile along with the libraries and code specified, and any desired containers can be launched from it.

Docker allows integration with tools such as Ansible and Jenkins, and with cloud providers such as Microsoft Azure, Google Cloud Platform and Amazon Web Services, among others.

[10] https://datatables.net/
[11] https://www.docker.com/

4.11 Docker Compose

Docker Compose[12] is a solution that allows you to manage multiple Docker containers on a single host machine [26].

Docker Compose uses a yaml file to indicate the services you want to build along with the images each one should be based on. In this file you can indicate environment variables, dependencies, ports and even commands that must be launched at startup.

While Docker Compose is the Docker container management solution on a single machine, Docker Swarm is the orchestration tool for managing containers on multi-machine distributed architectures.

[12] https://docs.docker.com/compose/

Relevant aspects of the project

5.1 Analysis

The first step of the project was to assess the **feasibility** and **viability** analysis of the concept devised. The author was looking to use two data sources with:

- Actual and updated data, preferably related to the social interest.
- The possibility of getting a stream data flow, avoiding static datasets.
- The potential to combine both sources to get an added value.

Taking into account the previous aspects, the author found **Twitter** and **Spotify** as interesting options. Both provide robust APIs for smooth development and have the necessary features to combine the collected data. Consequently, the author designed the following use case:

1. The **Twitter API** is consulted to gather the *tweets* with the hashtag `#NowPlaying`. The name of the endpoint queried is *recent search*.
2. The **tweet** is cleaned up, stopwords and other hashtags are removed, and the remaining text (which usually corresponds to the track name and artist) is isolated. The names of the endpoints queried are *search for item* and *get tracks' audio features*.
3. The **Spotify API** is queried to collect the identified track information.
4. The data is formatted and stored in a **.csv** file.

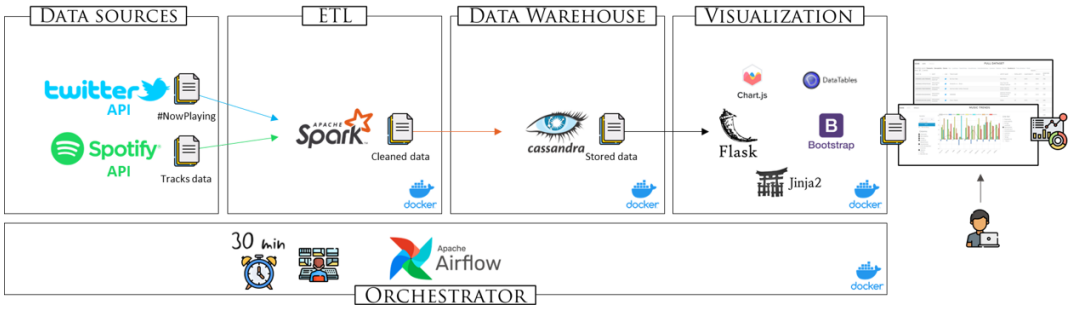

Figure 5.1: Project flowchart

5. The data is uploaded to the **Data Warehouse** and the .csv is stored as a history file.

6. The data served from the Data Warehouse is requested by the **back-end** and served on the **front-end**.

7. The data is displayed to the **user** on the front-end.

The final set of tools and data flow are shown in the flowchart 5.1.

5.2 Design

During the design phase, the author performed the following tasks:

- **Analyze the output of both APIs** using Postman to create the script to extract the data.

- **Identify the most appropriate software tools** to meet the project requirements. At this point, Apache Airflow was determined for flow orchestration, Apache Spark for data processing, Apache Cassandra for data storage, Flask and Bootstrap for data visualization (along with Chart.js and Datatables), and Docker and Docker Compose for service container management.

- **Organize at a high level the Sprints** that must be dedicated to each desired functionality and generate an overview of when each of them must be achieved so as not to affect the project timeline.

5.3 Implementation

During the development phase, the following tasks were performed:

- Containers for each service were defined in Docker Compose and custom images were created in Docker (if needed):
  - **Apache Airflow** containers configured for flow orchestration: webserver, scheduler, worker, init, triggerer, redis, postgres, client, and flower. Airflow setup required a custom image with the following packages installed via PyPI as additional requirements: "apache-airflow-providers-apache-spark", "requests", "pandas", "cqlsh". In addition, in the Airflow Dockerfile, the Java SDK 11 was installed and the `JAVA_HOME` variable set. The image used as base image was the official Airflow image (version 2.3.0) found on DockerHub[13], as well as the Docker Compose base file[14].
  - **Apache Spark** containers were configured for data processing: master and three workers. Spark setup required a custom image with the following packages installed via PyPI as additional requirements: "requests", "pandas", "cqlsh". The image used as base image was the Bitnami Spark image (version 3.1.2) obtained from DocherHub[15].
  - An **Apache Cassandra** container was configured for data storage, using an additional container to set up the database configuration. The image used was the official Cassandra (version 4.0) image found on DockerHub[16] and no additional requirements were needed.
  - A **Linux** container was configured for the web application. The container required a custom image with the following packages installed via PyPI as additional requirements: "flask" (version 2.1.2), "cassandra-driver" (version 3.25.0), "flask-cqlalchemy" (version 2.0.0), redis", "Cmake", "cryptography". The image used as base image was the official Python image (version "3.8-slim-buster") found on DockerHub[17].

[13] https://hub.docker.com/r/apache/airflow
[14] https://airflow.apache.org/docs/apache-airflow/stable/start/docker.html#docker-compose-yaml
[15] https://hub.docker.com/r/bitnami/spark
[16] https://hub.docker.com/_/cassandra
[17] https://hub.docker.com/_/python

- A **Python script** (PySpark, Python API for Apache Spark) was created to collect, join, transform and store the data from the Twitter and Spotify APIs to a .csv file.
- The **DAG** in Apache Airflow was configured to automate data extraction, transformation, and loading. The process is triggered every 30 minutes and collects 100 tweets in every instance. Within the DAG, there are three tasks (collect data, send data to Cassandra, and create historic file) which involved the operators *SparkSubmitOperator* (installed via PyPI with the package name "apache-airflow-providers-apache-spark") and *BashOperator*.
- **Apache Spark** was configured to be able to receive the script sent by Apache Airflow and communicate with Apache Cassandra.
- **Apache Cassandra** was configured with the keyspace structure required.
- **Flask and Bootstrap** were used to build the back and front-end of the web application, along with the **Chart.js** and **Datatables** resources.
- The **data were normalized** to avoid excessively different ranges between metrics. The normalization was made by adding a property (`@property`) to the data object that returns a list with the normalized data 5.2. The following fields were changed:
  - **Text modifications**: "artists_name" was modified to replace the comma with a line break to distinguish between artists collaborating on the same track, and "created_at" was parsed to a MM/DD/YYYY HH:MM:SS format.
  - **Numerical modifications**: "danceability", "energy", "speechiness", "acousticness", "instrumentalness", "liveness" and "valence" were moved from a 0-1 range to a 0-100 range and rounded to three decimal places, "loudness" and "tempo" were also rounded, and "duration_ms" was changed from milliseconds to seconds.

Once the project is deployed, three visual interfaces can be accessed that can help the user to better understand the process:

- **Apache Airflow user interface.** Referenced in Figure 5.3, it is accessible through port 8080 (`http://localhost:8080`) and allows

```python
class Tweetsandtracks(db.Model):
    keyspace = app.config['CASSANDRA_KEYSPACE']
    id_tweet = db.columns.Integer(primary_key=True)
    text_tweet = db.columns.Text()
    created_at = db.columns.DateTime()
    url_tweet = db.columns.Text()
    id_track = db.columns.Text()
    name = db.columns.Text()
    popularity = db.columns.Integer()
    artists_id = db.columns.Text()
    artists_name = db.columns.Text()
    danceability = db.columns.Float()
    energy = db.columns.Float()
    key = db.columns.Integer()
    loudness = db.columns.Float()
    speechiness = db.columns.Float()
    acousticness = db.columns.Float()
    instrumentalness = db.columns.Float()
    liveness = db.columns.Float()
    valence = db.columns.Float()
    tempo = db.columns.Float()
    duration_ms = db.columns.Integer()
    time_signature = db.columns.Integer()
    mode = db.columns.Float()

    @property
    def list(self):
        return [self.id_tweet, self.created_at.strftime("%m/%d/%Y %H:%M:%S"), self.url_tweet, self.name, self.artists_name.replace("{", "").replace("}", ""),
        self.popularity, round(self.danceability*100, 3), round(self.energy*100, 3), self.key, round(self.loudness, 3), round(self.speechiness*100, 3),
        round(self.acousticness*100, 3), round(self.instrumentalness*100, 3), round(self.liveness*100, 3), round(self.valence*100, 3),
        round(self.tempo, 3), round(self.duration_ms/1000, 0), self.time_signature, round(self.mode, 0)]
```

Figure 5.2: Object representing the Cassandra table

access, among other things, to the Airflow configuration and the list of configured DAGs, being able to observe their instances and obtain metrics such as execution times.

- **Apache Spark user interface.** Referenced in Figure 5.4, it is accessible through port 8181 (`http://localhost:8181`) and allows to observe the master node and the three workers, as well as their last tasks performed.

- **Front-end user interface.** It is accessible through port 8000 (`http://localhost:8000`) and contains three views:

  - **Home.** Referenced in Figure 5.5, it shows a brief introduction of the project and an animated gif to illustrate the implemented data flow.

  - **Data.** Referenced in Figure 5.6, it displays a table with all the data extracted. The table has been made with Datatables and allows the user to search and sort the data, as well as hide and show columns at will.

  - **Visuals.** Referenced in Figure 5.7, it displays a chart with two data selectors, a multi-select to add data as a bar and a single-select to add data as a line. A script was added that does not allow both selectors to have the same column in their data, so when a column that is present in one of them is selected in the other, it is removed from the previous one. It also has a selector to choose the amount of data to display (values of 5, 10, 15 and 20, to avoid saturating the graph), as well as an order selector

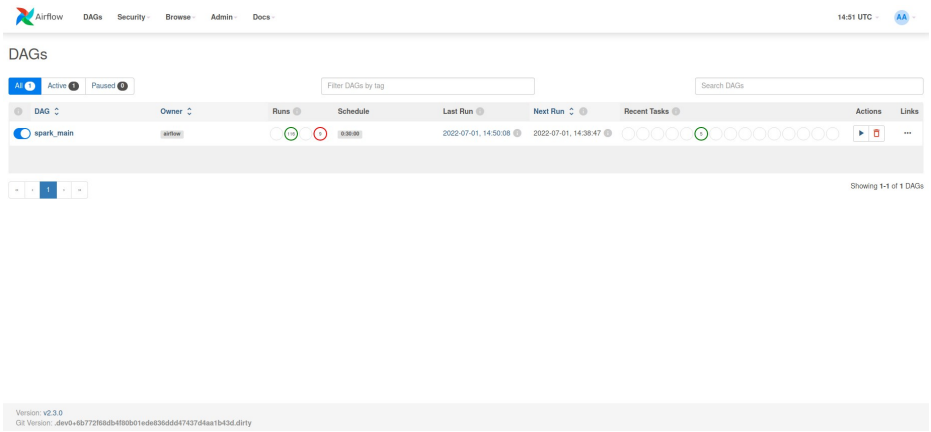

Figure 5.3: Airflow User Interface

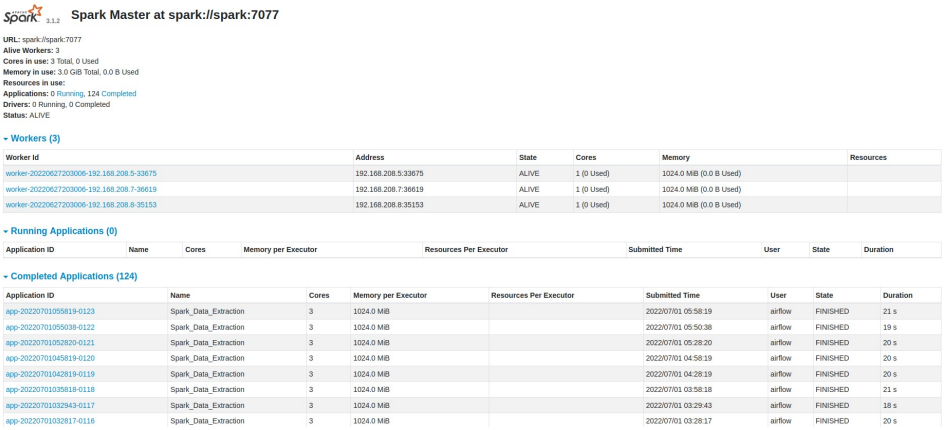

Figure 5.4: Spark User Interface

that allows the user to sort by any of the columns in ascending or descending order.

More details on user interfaces are provided in section E.4.

The most relevant aspects and issues faced by the author during the development are:

- **Developer keys.** Both Twitter and Spotify required to obtain developer keys in order to use their APIs. For obvious reasons, the *.env* file provided in GitHub does not have the developers keys added.

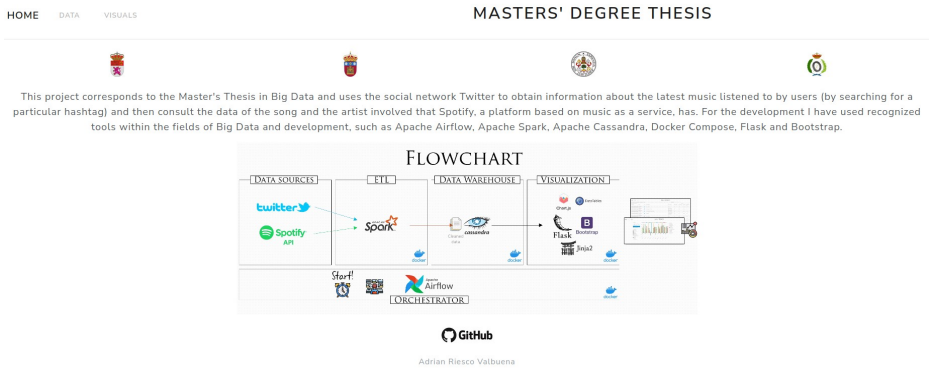

Figure 5.5: Front-end home view

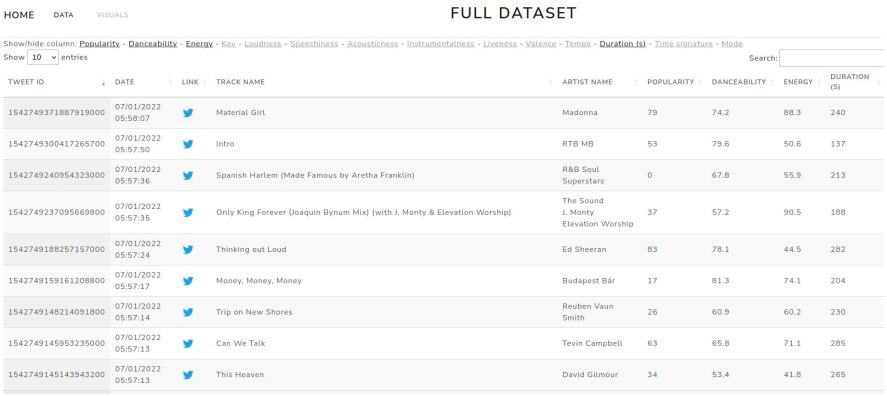

Figure 5.6: Front-end data view

- **Rate limits of the APIs.** The Twitter API allows the developer to collect up to 500 000 tweets per month, what forced the author to set up a limit for both the development and the production phases.

- **Airflow operators.** Airflow does not have an operator for Cassandra that manage data loading from a .csv to the database. In addition, the command `COPY FROM`, usually used to send data from a file to Cassandra, was not designed to be used inside a Python script. The author had to use the *`BashOperator`* to connect to Cassandra from the Airflow Webserver container and run the `COPY FROM` command, instead of the *`PythonOperator`* or any external Cassandra operator.

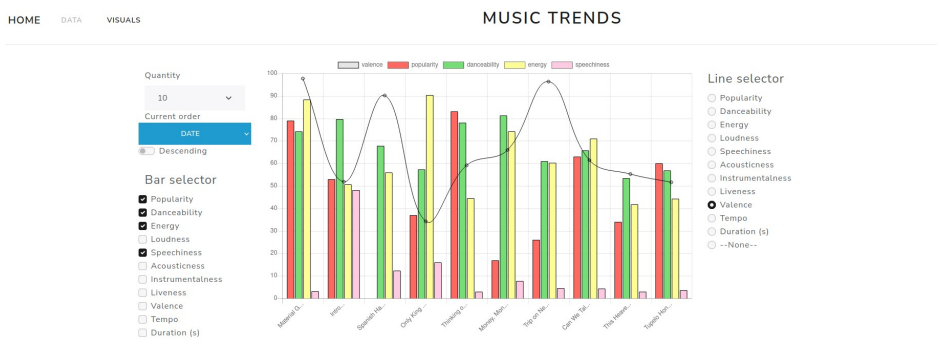

Figure 5.7: Front-end graph view

- **Airflow and Spark connection.** To avoid the user having to manually create the connection between Airflow and Spark using the Airflow UI, the author used an environment variable to define the connection so that it can be created during the Docker Compose deployment. As a quirk, according to the Airflow documentation, "Connections defined in environment variables will not show up in the Airflow UI or using airflow connections list"[18].

- **Database schema configuration at launch.** Apache Cassandra required some external resources to build the database schema. Rather than using a script and build a custom image, the author found an interesting solution consisting on the deployment of a temporary container ("cassandra-init" service in the Docker Compose file) that waits for the Cassandra container to be healthy and then creates the schema via cqlsh command. The solution was found on Stackoverflow[19] and it referenced one of the Netflix's repositories[20], where a service named "cassandra-load-keyspace" was performing the same task.

- **Data representation.** Before deciding to use Datatables, the author tried to build a custom implementation for the sort and search capabilities. After spending a large part of a Sprint trying to configure the

[18] https://airflow.apache.org/docs/apache-airflow/stable/howto/connection.html#storing-a-connection-in-environment-variables

[19] https://stackoverflow.com/questions/40443617/init-script-for-cassandra-with-docker-compose

[20] https://github.com/Netflix/osstracker/blob/master/docker-compose.yml

solution using other frameworks and libraries, the author discovered Datatables and used it to build the final view of the table.

- **Mismatched tracks.** Occasionally the text extracted from the tweets is linked to a different tracks than the one that can be read. This is mainly due to two factors:

  – Although the hashtag used to capture them is an English language expression (NowPlaying), the captured tweets are in several languages. During data cleaning, the author has seen fit to remove characters outside the basic Latin alphabet, i.e., keeping only Unicode characters located in the 0-127 range. This means that when a track contains characters in another alphabet, as it happens in a Korean track, but the user who wrote the tweet enters larger letters in the Latin alphabet (for example, to express his mood about it), the system retains only the latter and uses them to identify the track.

  – To simplify and speed up the process, the result collected as valid from Spotify is the first one returned by the endpoint to which the query is launched, which causes deviations when the text resulting from cleaning up the tweet is similar for several different tracks.

  However, the author carried out an analysis on a random sample of tracks and, given the project objectives, considered that the erroneous percentage (10%) was small enough to be able to continue with the development of the project and build a fairly reliable database.

The project development was undertaken following an Agile methodology, with 9 Sprints of 2 weeks of duration being represented as *Milestones* in Github and the tasks as *Issues*.

Related works

Both the software tools used in the project, widely recognized within the Big Data industry, and the data sources, from large cap and renowned companies, have been applied to countless projects throughout their lifetime.

During the planning and study phase of the APIs and tools, the author discovered several projects that have combined both elements in different ways. Although no project has been found that combines both APIs and involves a data flow similar to that of the present project, the author has decided to mention those considered most interesting, either because of their data flow, their usefulness or the originality of the idea:

- **Tweetpy** [21]. Created in 2009 by Joshua Roesslein, this project contributes a Python library to interact with the Twitter API, simplifying interactions during development. Despite considering it as an alternative, the author decided not to make use of it in order to learn directly how to interact with the API. However, in the case of having to interact with user accesses, it would probably have been chosen.
- **Spotipy** [22]. Created in 2014 by Paul Lamere, this project provides a Python library to interact with the Spotify API. As with the previous library, it was decided not to use it to learn from direct interaction with the API, since the data flow of the project does not involve complex access permissions.
- **SpotaTweet** [23]. Although its data flow is not analogous to that of the project, its use case bears some similarity, with a collection of

[21] https://www.tweepy.org/
[22] https://spotipy.readthedocs.io/
[23] https://github.com/twitterdev/spotatweet

tweets with the hashtag `#NowPlaying` and a subsequent search for the song listened to on Spotify with the goal of displaying a preview to the user.

- **Divide for Spotify** [24]. This project makes use of the Spotify API to divide the songs that have been liked among the playlists of the user in question. This project has been chosen for its originality, as it takes advantage of the API to add a very interesting functionality to the platform, automatically dividing the songs of a playlist to choose from a selection of playlists chosen by the user.
- **Spotify to Twitter** [25]. This project combines both APIs to automate a Twitter account to publish tweets with songs from a Spotify playlist. The project is based on `Node.js`, an open source, cross-platform execution environment for the JavaScript-based server layer.

The projects mentioned above are only a fragment of those encountered during the initial analysis. Due to the breadth of Big Data domains, the use cases are innumerable and specialized tools can be of use in a number of different environments.

[24] https://divideforspotify.com/
[25] https://github.com/transitive-bullshit/spotify-to-twitter

Conclusions and future work lines

Given the requirements established in the planning phase, the final result is considered a success. The author has learned how to handle the tools included in the scope and how to interconnect them to achieve a complete ETL flow with a simple but effective visualization layer.

In case of continuing with the presented work, the author has performed an exercise of analysis and identification of the next steps and improvements that could be implemented:

- **Improve the percentage of correctly identified tracks.** This action could be addressed in several ways, including: discarding tweets with characters outside the Latin alphabet (not only specific characters), removing stop words in several languages so as not to have to discard all characters not included in the Latin alphabet, or increasing the number of tracks returned by Spotify and performing a comparative method (bag of words, TI-IDF, etc.) between their titles and the full text of the tweet. This would improve the reliability of the database and give more confidence when building the global metrics.

- **Ensure that we capture as much data as possible.** In the case of wanting to develop the project on a larger scale, both providers can be contacted to try to obtain an increase in the consumption limit of their APIs. In this way, depending on the new limitation, it could try to divide its catch at a certain time. In the best case, it would be possible to capture all tweets, even if our system captures them periodically, since the identifier of the last tweet captured in the previous iteration can be used as a reference for the endpoint to start the new capture from that point.

- **Add visualizations in the front-end layer.** Additional visualizations could be introduced that would allow the user, for example, to group by artist and to obtain global metrics that answer questions such as: "How popular are this artist's tracks?" or "How danceable are this artist's tracks compared to this other artist's tracks?". Another possible view to develop would be one that allows tracks to be observed by metric ranges and answers questions such as what percentage of the total tracks are between 80-100 of the energy metric.

- **Upload from history functionality.** Another improvement that would facilitate user interaction with the tool would be the ability to load the file history from the tool itself. In this way, the user could stop the process, select his historical files and be able to load only the desired data.

- **Replace Docker Compose with a more suitable tool.** At the design level, although Docker Compose fits the requirements for this project, in the case that a large scalability is required, it would be convenient to replace the tool with another one that allows container management across multiple hosts, such as Docker Swarm or Kubernetes.

- **Refactor the code.** In the event that the process increases in the amount of data captured, it is advisable to perform an analysis of the capture process and a refactoring of the necessary parts to speed up the process, if necessary. With the current data volume, the execution speed of the whole process triggered by Apache Airflow takes an average of 30-35 seconds.

- **Partial or total migration to the cloud.** Finally, if deemed consistent with new requirements that may arise in the future, parts or even the project could be migrated to a cloud infrastructure. For example, the Data Warehouse could be replaced by one of the native offerings of the major Cloud providers.

Most of these steps were considered for inclusion at some point in the development phase, but were discarded to ensure that the project, which for time reasons was conceived from the beginning as a proof-of-concept rather than an end-user oriented application, remained on schedule.

Appendixes

Appendix A

Project Plan

A.1 Introduction

The project planning was decided in an initial meeting between the author and his tutor. It was based in an Agile methodology, with two-weeks *sprints* and meetings between the author and his tutor conditioned to their availability.

The project repository was stored in GitHub under the url https://github.com/AdrianRiesco/Data-Engineer-project. Each *sprint* was created as a *milestone*, with the *issues* contained there being the tasks assigned. The *issues* were created to reflect tasks at most eight hours, allowing the author segregate his work and manage each *sprint* better. The author closed an *issue* when the task was finished and a *milestone* when the *sprint* was over, regardless of its state. If a task remained in an open state when a *sprint* reached its planned end date, the *issue* was transfered to the next *milestone*.

A meeting was held by the author and his tutor at the end of each sprint. During these meetings, both of them reviewed the state and development of the tasks of the corresponding sprint and planned the tasks of the next sprint. All the *milestones* and *issues* can be consulted in the project repository.

A.2 Temporary planning

The sprints carried out for the development of the project are described below with their corresponding dates:

Initial meeting. Held on Monday January 31st, it was the start point for the first sprint. During this meeting, the objective of the project, the data source and the tools to be used were validated by both the author and his tutor. The author previously made a research and came with an idea and the tutor exposed his point of view to create the final goal.

Sprint 1. Weeks of January 31st and February 7th. This Sprint had the following tasks assigned:

- Configure the work environment.
- Configure the project memory template.
- Write a draft of the objectives and main goals.
- Write a brief description of the tools selected.
- Write a brief explanation of the selected tools and the work methodology.
- Inspect Twitter API.
- Inspect Spotify API.

The end-of-sprint meeting was held on Wednesday February 16th. Analysis: Most of the activities were realized by the author, excepting the Inspection of the Spotify API. Regarding the Twitter API, the author inspected the output and he concluded that it had the characteristics needed to be used to launch queries to the Spotify API (the tweet could be cleaned to get the song name and artist).

Sprint 2. Weeks of February 14th and February 21st. This Sprint had the following tasks assigned:

- Write the code to gather information from Spotify.
- Write the code to gather information from Twitter.
- Write a description of Spotify and Twitter APIs.
- Write the API inspection process in the "Programmer guide" section.
- Write the project introduction.
- Write the Twitter and Spotify data description.

The end-of-sprint meeting was held on Wednesday February 2nd. Analysis: All the activities were accomplished on time. The author

identified some potential problems when extracting information from the Spotify API, such as the name of the song must be quite accurate to be able to get the search results.

Sprint 3. Weeks of February 28$^{\text{th}}$ and March 7$^{\text{th}}$. This Sprint had the following tasks assigned:

- Write a description of Twitter API.
- Write a description of Spotify API.
- Write the description of the Twitter data.
- Write the description of the Spotify data.
- Improve and unify the code written to collect data from both APIs.
- Set up the Spark environment using Docker.

The end-of-sprint meeting was held on Wednesday March 16$^{\text{th}}$. Analysis: During this sprint, the author had difficulties configuring the Docker environment, which derived in a delay of the other tasks. These tasks were transfered to the next sprint.

Sprint 4. Weeks of March 14$^{\text{th}}$ and March 21$^{\text{st}}$. This Sprint had the following tasks assigned:

- Write a description of NoSQL Databases.
- Set up the Spark environment using Docker.
- Set up the Airflow environment using Docker.

The end-of-sprint meeting was held on Wednesday May 4$^{\text{th}}$. Analysis: Due to personal reasons, the author could not continue with the project during the month of April, so there was a temporary pause in the planning.

Sprint 5. Weeks of May 2$^{\text{nd}}$ and May 9$^{\text{th}}$. This Sprint had the following tasks assigned:

- Configure the DAG in Airflow.
- Learn the fundamentals of Flask.
- Redesign the project plan excluding the month of April

The end-of-sprint meeting was held on Wednesday May 18$^{\text{th}}$. Analysis: All tasks were accomplished on time.

Sprint 6. Weeks of May 16$^{\text{th}}$ and May 23$^{\text{rd}}$. This Sprint had the following tasks assigned:

- Deploy Cassandra environment.
- Write the description of the Twitter data.
- Write the description of the Spotify data.
- Write a description of an orchestrator.
- Write a description of CI/CD.

The end-of-sprint meeting was held on Monday May 30$^{\text{th}}$. Analysis: All tasks were accomplished on time. The Cassandra deployment took most of Sprint's time, generating a few prototypes before the final version.

Sprint 7. Weeks of May 30$^{\text{th}}$ and June 6$^{\text{nd}}$. This Sprint had the following tasks assigned:

- Integrate Cassandra with Airflow and Spark.
- Create the ETL workflow to load the data to Cassandra.
- Write a description of Apache Spark.
- Write a description of Docker and Docker Compose.
- Write a description of Flask, Jinja and Bootstrap.

The end-of-sprint meeting was held on Tuesday June 14$^{\text{th}}$. Analysis: All tasks were accomplished on time. The integration of Cassandra with Airflow and Spark took most of Sprint's time.

Sprint 8. Weeks of June 13$^{\text{th}}$ and June 20$^{\text{th}}$. This Sprint had the following tasks assigned:

- Create the front-end with Flask and Bootstrap.
- Integrate the ETL workflow with the front-end.
- Write section 5 "Relevant aspects of the project".
- Write section 7 "Conclusions of the project".

The end-of-sprint meeting was held on Monday June 27$^{\text{th}}$. An additional meeting was held in the middle of the sprint to review and propose changes to the first version of the front-end. Analysis: Although the front-end development was very time-consuming, all tasks were accomplished on time. At this point, all development work was finished.

Sprint 9. Weeks of June 27rd and July 4th. This Sprint had the following tasks assigned:

- Write section 6 "Related works".
- Complete Appendix A "Project plan".
- Complete Appendix B "Requirements".
- Complete Appendix C "Design".
- Complete Appendix D "Programmer's guide".
- Complete Appendix E "User manual".
- Review the entire project report.

The end-of-sprint meeting was held on M— July –th. An additional mid-sprint meeting was held to review the report and plan the final tasks to finalize the project for defense.Analysis: The report was finished and the project was delivered.

A.3 Feasibility study

The architecture of the project and the use case were designed to ensure its feasibility.

Economic feasibility

The project is based on open-source platforms to ensure its economic and legal feasibility. The APIs where the information was gathered are free to use if the developer keeps his queries under specific limit rates.

Legal feasibility

The project is based on open-source platforms to ensure its economic and legal feasibility. The licenses in which the technology and tools used in the project are based are:

- **Twitter License.** Free with usage limitations and requires developer account [27].
- **Spotify License.** Free with usage limitations and requires developer account [28].
- **Apache License.** Free with limitations [29].

Tools	Twitter	Spotify	Apache	BSD-3-Clause	MIT	GPL
Twitter API	X					
Spotify API		X				
Apache Airflow			X			
Apache Spark			X			
Apache Cassandra			X			
Flask				X		
Bootstrap					X	
Datatables					X	
Chart.js					X	
TEXMaker						X

Table A.1: Tools and technologies' licenses

- **Flask License.** Free with limitations, uses BSD-3-Clause license [30].
- **Bootstrap License.** Free with limitations, uses Massachusetts Institute of Technology (MIT) license [31].
- **TEXmaker License.** Free with limitations, uses GNU General Public License (GPL) license [32].

Appendix B

Requirements

B.1 Introduction

This section lists the general objectives and requirements identified during the initial planning of the project and on whose fulfillment the development of the project has focused.

B.2 General objectives

The requirements through which the use case was built were the following:

- Ability to obtain data in real time.
- Combination of at least two different data sources.
- Potential to scale in both technology and data volume.
- Involvement of various technologies in the Big Data field.
- Use of open source tools.

B.3 Catalog of requirements

Functional requirements

The functional requirements (FR) that the project had to meet were:

- **FR1**. The data must be obtained from the Twitter hashtag *#NowPlaying* every 30 minutes, taking care of API rate limits.
  - **Status**: Met.
  - **Rationale**: The Apache Airflow DAG was configured with a run frequency of 30 minutes. The script used to collect the data has a maximum of 100 tweets collected per run to comply with API rate limits.
- **FR2**. There must be at least two different visualizations and one of them must provide the ability to view all of the stored data.
  - **Status**: Met.
  - **Rationale**: There are two visualizations in the web application, "Data" and "Visuals", and the first one displays all the stored data.
- **FR3**. At least one of the visualizations must show last songs name, artist and audio features.
  - **Status**: Met.
  - **Rationale**: The "Data" view of the web application displays all required fields.
- **FR4**. At least one of the visualizations must have a link to the source tweet.
  - **Status**: Met.
  - **Rationale**: The "Data" view of the web application contains a link to the source tweet.
- **FR5**. At least one of the visualizations must have the ability to compare different metrics.
  - **Status**: Met.
  - **Rationale**: The "Visuals" view of the web application provides the ability to visually compare metrics.
- **FR6**. At least one of the visualizations must combine two different types of visualizations.
  - **Status**: Met.

  - **Rationale**: The "Visuals" view of the web application combines bar and line formats in the same chart.

- **FR7**. Both visualizations must provide sorting capabilities.
  - **Status**: Met.
  - **Rationale**: Both "Data" and "Visuals" views of the web application provides provides the ability to sort by metric.

- **FR8**. Both visualizations must be responsive to different screen sizes.
  - **Status**: Met.
  - **Rationale**: The web application was developed using Bootstrap as the CSS framework to facilitate scaling and resizing on different screens.

Technical requirements

The technical requirements (TR) that the project had to meet were:

- **TR1**. The development must have the ability to be deployed in different environments with minimum effort.
  - **Status**: Met.
  - **Rationale**: The services were developed using multiple containers managed via Docker Compose.

- **TR2**. The data flow must be automated, with the entire process orchestrated by a single tool.
  - **Status**: Met.
  - **Rationale**: All data flow is orchestrated by Apache Airflow.

- **TR3**. The execution of the ETL process must be done with a tool that can scale and run in distributed environments.
  - **Status**: Met.
  - **Rationale**: Data extraction, transformation and loading (into a .csv file) is processed by Apache Spark, and data loading to Cassandra is performed by a Cassandra Query Language shell (cqlsh) command.

- **TR4**. The data warehouse must have the ability to escalate in terms of a Big Data problem.
  - **Status**: Met.
  - **Rationale**: Apache Cassandra is the Data Warehouse used.
- **TR5**. The web application must be designed with widely recognized tools.
  - **Status**: Met.
  - **Rationale**: The web application was designed with Flask and Bootstrap, and the visualizations with Chart.js and Datatables.
- **TR6**. All the tools used must be open source.
  - **Status**: Met.
  - **Rationale**: All the tools used to develop the project are open source.

Appendix C

Design specification

C.1 Introduction

This section shows the structure of the data used and the flow and architecture diagrams.

C.2 Data design

Twitter data structure

The data received from Twitter queries has the following structure (the data has been obtained from a real query and its output has been reduced by trimming certain elements due to their length):


```
{"data":[
      {
         "id":"1533311938209382403",
         "entities":{
            "annotations":[
               {
                  "start":43,
                  "end":62,
                  "probability":0.6061,
                  "type":"Other",
                  "normalized_text":"Breakfast In America"
               }
            ],
            "urls":[
               {
                  "start":84,
                  "end":107,
                  "url":"https://t.co/YiXxSepm8x",
                  "expanded_url":"https://rideshare.airtime.pro",
                  "display_url":"rideshare.airtime.pro",
```

```
                "images":[
                    {
                        "url":"https://pbs.twimg.com/news_img/
                            1532561096858640397/3mUiSDDN?format=jpg&
                            name=orig",
                        "width":1920,
                        "height":1200
                    },
                    {
                        "url":"https://pbs.twimg.com/news_img/
                            1532561096858640397/3mUiSDDN?format=jpg&
                            name=150x150",
                        "width":150,
                        "height":150
                    }
                ],
                "status":200,
                "title":"Rideshare Radio",
                "description":"Hits from the 70's 80s 90s 00s 10s
                     20s, No Talking just back to back Music
                    totally commercials free 24/7",
                "unwound_url":"https://rideshare.airtime.pro"
            }
        ],
        "hashtags":[
            {
                "start":0,
                "end":11,
                "tag":"Nowplaying"
            },
            {
                "start":129,
                "end":139,
                "tag":"Rideshare"
            }
        ]
    },
    "created_at":"2022-06-05T04:57:08.000Z",
    "text":"#Nowplaying 2010 Remastered - Supertramp -
        Breakfast In America  - Stream here-&gt; https://t.co/
        YiXxSepm8x - Non Stop Hits 24/7 #Rideshare #Radio #Hits
         #Uber #RideshareRadio #Petrol #Parcoursup #
        PlatinumJubilee #ENGvNZ #PrideMonth"
  },
  {
    Second tweet.
  }
],
"meta":{
  "newest_id":"1533311938209382403",
  "oldest_id":"1533311921256124416",
  "result_count":10,
  "next_token":"b26v89c19zqg8o3fpyzltxkhapj3hf1q96mc01w4yl3el"
}
}
```


The fields required for the project are:

- **id.** Tweet id, useful uniquely identify the tweet.
- **text.** Tweet text, useful to identify the song played.
- **entities.** useful to clean the text and remove hashtasg, cashtags, mentions and urls.
- **created_at.** Tweet creation date.

Spotify data structure

The data received from Spotify queries to Search endpoint ("Search for Item") has the following structure (the data has been obtained from a real query and its output has been reduced by trimming certain elements due to their length):


```
{"tracks":{
      "href":"https://api.spotify.com/v1/search?query=Savoy+Brown+
          Wang+Dang+Doodle+&type=track&offset=0&limit=1",
      "items":[
         {
            "album":{
               "album_type":"album",
               "artists":[
                  {
                     "external_urls":{
                        "spotify":"https://open.spotify.com/artist/
                            17obwOahRWI121iMUZznh2"
                     },
                     "href":"https://api.spotify.com/v1/artists/
                         17obwOahRWI121iMUZznh2",
                     "id":"17obwOahRWI121iMUZznh2",
                     "name":"Savoy Brown",
                     "type":"artist",
                     "uri":"spotify:artist:17obwOahRWI121iMUZznh2"
                  }
               ],
               "available_markets":[
                  "MX",
                  "US"
               ],
               "external_urls":{
                  "spotify":"https://open.spotify.com/album/
                      5oq20r8iNOO9fpw8R2h3vE"
               },
               "href":"https://api.spotify.com/v1/albums/
                   5oq20r8iNOO9fpw8R2h3vE",
               "id":"5oq20r8iNOO9fpw8R2h3vE",
               "images":[
                  {
                     "height":640,
                     "url":"https://i.scdn.co/image/
                         ab67616d0000b2735dd44bf0a252e30d4bb2e7c8",
                     "width":640
```

```
                },
                {
                    "height":300,
                    "url":"https://i.scdn.co/image/
                        ab67616d00001e025dd44bf0a252e30d4bb2e7c8",
                    "width":300
                }
            ],
            "name":"Street Corner Talking",
            "release_date":"1971-01-01",
            "release_date_precision":"day",
            "total_tracks":8,
            "type":"album",
            "uri":"spotify:album:5oq20r8iNOO9fpw8R2h3vE"
        },
        "artists":[
            {
                "external_urls":{
                    "spotify":"https://open.spotify.com/artist/
                        17obwOahRWI121iMUZznh2"
                },
                "href":"https://api.spotify.com/v1/artists/
                    17obwOahRWI121iMUZznh2",
                "id":"17obwOahRWI121iMUZznh2",
                "name":"Savoy Brown",
                "type":"artist",
                "uri":"spotify:artist:17obwOahRWI121iMUZznh2"
            }
        ],
        "available_markets":[
            "MX",
            "US"
        ],
        "disc_number":1,
        "duration_ms":440733,
        "explicit":false,
        "external_ids":{
            "isrc":"GBF077120720"
        },
        "external_urls":{
            "spotify":"https://open.spotify.com/track/
                7p99XDR7dKaIMTYV3zia0V"
        },
        "href":"https://api.spotify.com/v1/tracks/
            7p99XDR7dKaIMTYV3zia0V",
        "id":"7p99XDR7dKaIMTYV3zia0V",
        "is_local":false,
        "name":"Wang Dang Doodle",
        "popularity":19,
        "preview_url":"None",
        "track_number":7,
        "type":"track",
        "uri":"spotify:track:7p99XDR7dKaIMTYV3zia0V"
    }
],
"limit":1,
"next":"https://api.spotify.com/v1/search?query=Savoy+Brown+
    Wang+Dang+Doodle+&type=track&offset=1&limit=1",
"offset":0,
```

```
        "previous":"None",
        "total":11
    }
}
```


The fields required for the project are:

- **id.** Track id, useful uniquely identify the track.
- **name.** Track name, useful to identify the track.
- **popularity.** Popularity of the track.
- **artists' id.** ID of the artists.
- **artists' name.** Name of the artists.

The data received from Spotify queries to the Tracks endpoint ("Get Tracks' Audio Features") has the following structure (the data has been obtained from a real query and its output has been reduced by trimming certain elements due to their length):


```
{"audio_features":[
        {
            "danceability":0.516,
            "energy":0.36,
            "key":7,
            "loudness":-11.264,
            "mode":1,
            "speechiness":0.03,
            "acousticness":0.83,
            "instrumentalness":0.885,
            "liveness":0.116,
            "valence":0.144,
            "tempo":127.176,
            "type":"audio_features",
            "id":"7dg3XqARw7qOrkt9pZZNRF",
            "uri":"spotify:track:7dg3XqARw7qOrkt9pZZNRF",
            "track_href":"https://api.spotify.com/v1/tracks/
                7dg3XqARw7qOrkt9pZZNRF",
            "analysis_url":"https://api.spotify.com/v1/audio-analysis/
                7dg3XqARw7qOrkt9pZZNRF",
            "duration_ms":233812,
            "time_signature":4
        },
        {
            Group of features of the second track.
        }...]
}
```


The fields required for the project are:

- id.
- danceability.
- energy.
- key.
- loudness.
- mode.
- speechiness.
- acousticness.
- instrumentalness.
- liveness.
- valence.
- tempo.
- duration_ms.
- time_signature.

Cleaned data

After the cleaning process, the resulting data structure is:

- id_tweet.
- text.
- created_at.
- url_tweet.
- id_track.
- name.
- popularity.
- artists_id.

- artists_name.
- danceability
- energy
- key
- loudness
- mode
- speechiness
- acousticness
- instrumentalness
- liveness
- valence
- tempo
- duration_ms
- time_signature

C.3 Architectural design

The component diagram is shown in the figure C.1. Each block represents a service deployed in one or more containers, except for data sources, which are external APIs. The UML format is referenced in the figure C.2.

C.4 Procedural design

The flow diagram is shown in the figure C.3.

1. Get tweets with the hashtag `#NowPlaying`.
2. Search the track information and audio features.
3. Get the track information and audio features.
4. Load the data into the Data Warehouse.

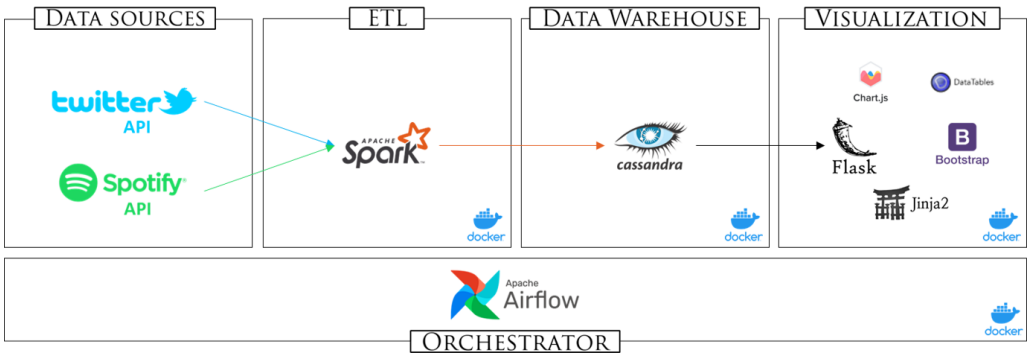

Figure C.1: Component diagram

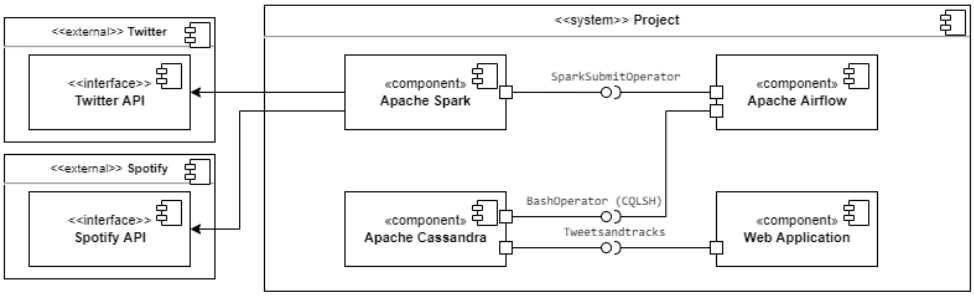

Figure C.2: UML component diagram

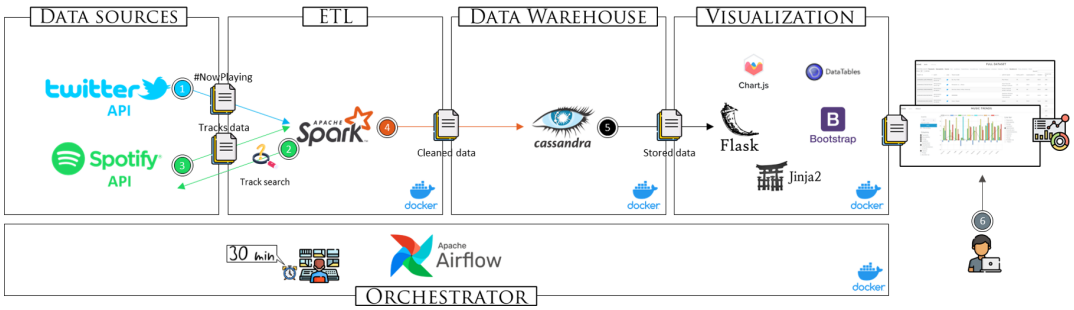

Figure C.3: Flow diagram

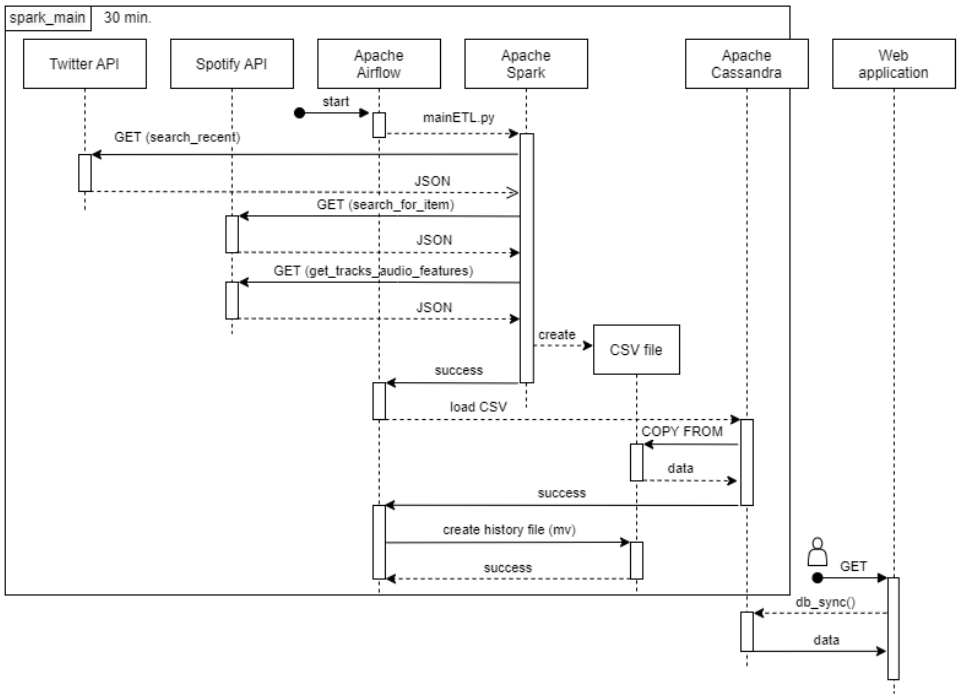

Figure C.4: UML sequence diagram

5. Request the data from the web application.

The sequence diagram is depicted in figure C.4. The loop represents the "spark_main" DAG, executed every 30 minutes and composed of three API calls, an Apache Cassandra load and the creation of the history file. The section outside the loop represents the user accessing the web application.

Appendix D

Programming technical documentation

D.1 Introduction

This section contains the directory structure used and the main technical details that a user wishing to reproduce or execute the project should be aware of.

D.2 Directory structure

The project repository, hosted on GitHub, has the following directory structure:

- `airflow`. It contains the created DAG and the folders that Airflow needs to function properly.
- `cassandra`. I contains the database schema required.
- `doc`. It contains the project report and the LaTeX files used to generate it.
- `docker`. It contains the Docker Compose file required to deploy the project and the Airflow, Spark and Flask folders used for building custom images.
- `spark`. It contains the script used to collect the API data, the necessary libraries and the history files.

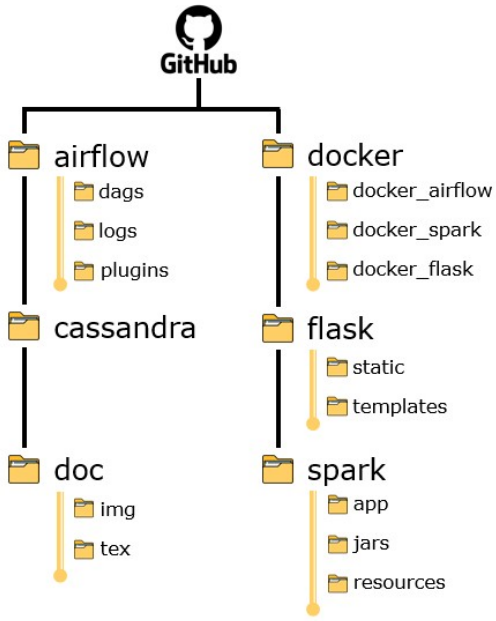

Figure D.1: Directory structure

The directory structure is shown in the figure D.1.

D.3 Programmer's guide

Analysis

During the analysis phase, the author inspected the output of Twitter and Spotify APIs using Postman. In the first place, relying on the Twitter documentation, the author examined the Twitter API by following the next steps:

1. Get access to the Twitter Developer Portal.
2. Get the credentials needed to consult the different endpoints of the API.
3. Import the *Twitter API v2* collection on Postman.
4. Create a fork of the automatically created environment (*Twitter API v2*) and collection *Twitter API v2* to be able to edit the values.

5. Modify the environment to include the following developer keys and tokens:
   - Consumer key (`consumer_key`).
   - Consumer secret (`consumer_secret`).
   - Access token (`access_token`).
   - Token secret (`token_secret`).
   - Bearer token (`bearer_token`).
6. In the collection tab, select the endpoint *Search Tweets* $\longrightarrow$ *Recent search* for the initial exploration. Configure the following parameters:
   - `query` = `#NowPlaying`
   - `tweet.fields` = `created_at,entities`
   - `max_results` = `10`
7. Now, the query can be sent `https://api.twitter.com/2/tweets/search/recent?query=%23NowPlaying&max_results=10&tweet.fields=created_at,entities` to get the 10 most recent tweets with the hashtag ***#NowPlaying*** and receive their basic information (id, text) as well as the entities (hashtags, urls, annotations...) and the creation time stamps.

After analyze the data gathered from the Twitter API, the author inspected the Spotify API (more specifically the endpoint "Search for Item"), following the next steps:

1. Get access to the Spotify Developer Portal.
2. Enter in the developers console and select the "Search for Item" endpoint.
3. Specify the parameters of the search. The type "track" can be specified and a limit of one can be added to receive only the first song found.
4. After clicking get carrier token, that token can be used by clicking Try Me or by copying the resulting query into a Linux console to test the output.
5. The result of this query is the first search result containing artist, song and album information. With the artist and song ids, other endpoints can be queried to obtain audio analysis, audio characteristics and artist information, among others.

Development

To run the project, the following items must be installed in the system:

- Docker[1]. The author used the version `20.10`.
- Docker Compose[2]. The author used the version `1.29`.
- Python[3]. The author used the version `3.8`.

The above packaged were installed in a "`Ubuntu 20.04.4 LTS`" virtual machine. In case of using a different operating system, please refer to the package documentation.

It is important to note that to run or play the project, the user has to modify the `.env` file located within the `docker` folder and include the respective developer keys for both Twitter and Spotify.

> *The commands described for the installation and usage of the project are oriented to Linux environments.*

D.4 Compilation, installation and execution of the project

The steps needed to run the project are:

1. Ensure that the folders `spark/resources/history`, `airflow/logs` and `airflow/plugins` have the necessary permissions by executing the command `sudo chmod -R 777 folder_to_set_permissions`:
   - `sudo chmod -R 777 spark/resources/history`
   - `sudo chmod -R 777 airflow/logs`
   - `sudo chmod -R 777 airflow/plugins`
2. Open a `command prompt` and move to the `docker` folder.
3. Launch the environment with "`sudo docker-compose up -build`" (`-build` ensures that all the images are built during the deployment). It can be launched in the background by adding the flag `-d`: "`sudo docker-compose up -build -d`".

[1] https://docs.docker.com/engine/install/ubuntu/
[2] https://docs.docker.com/compose/install/
[3] https://docs.python-guide.org/starting/install3/linux/

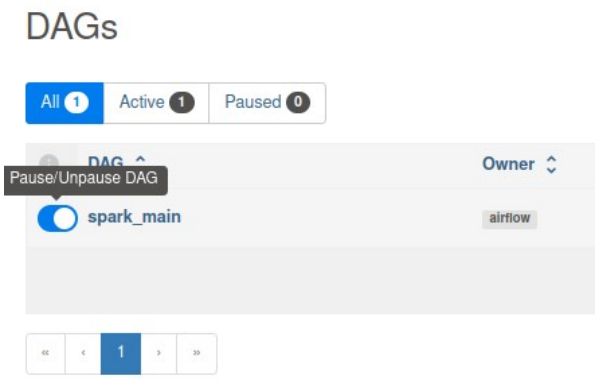

Figure D.2: Pause/unpause DAG

4. Wait until all the services have started and are in a healthy state. It usually is achieved when the `airflow_webserver_container` is continuously displaying a status message. If the flag `-d` was included, the logs can be accessed with the command "`sudo docker-compose logs`". A list of the services can be displayed with the command "`sudo docker ps`".

5. Access to the Airflow UI (`http://localhost:8080`, user "`airflow`" and password "`airflow`") and start the DAG "`spark_main`" as shown in the figure D.2.

6. Access to the web application (`http://localhost:8000`) and check that the "`Data`" and "`Visuals`" views contain the captured data.

7. To stop the containers, the command "`sudo docker-compose down`" can be used. To remove the networks and docker volumes, "`sudo docker-compose down -v`" must be used. The process can also be stopped by pausing the DAG as explained in step 5.

The author faced some problems during the development phase that are easily solved:

- If the build fails for an error in the Airflow container related with the `/opt/airflow/logs` folder, it will probably be due to the permissions in the `airflow/logs` folder. To solve it, open a command prompt in the `airflow` folder and execute the commands "`sudo chmod -R 777 logs`" and "`sudo chmod -R 777 plugins`".

- If the DAG fails, the logs can be consulted in the `airflow/logs/dag_id=spark_main` folder. If the "`cassandra_load`" task is failing, it can be due to the permissions of the `spark/resources/history` folder. To solve that, open a command prompt in the `spark/resources` folder and execute the command "`sudo chmod -R 777 history`". Then, remove the "`data.csv`" file located in the `spark/resources` folder (or manually create the history file) and execute the DAG again.

D.5 System tests

To test that the project is working in a proper way, there are a few tests that can be performed:

1. Access to the Airflow UI and review the status of the last iterations of the "`spark_main`" DAG.
2. Access to the Spark UI (`http://localhost:8181`) and check that the worker nodes are executing or have executed the tasks sent from Airflow.
3. Access to the `spark/resources/history` folder and check that the historic file has been created with the format "`YYYYMMDDhhmmss`" (year, month, day, hour, minute, second).
4. Access to the "`Visuals`" view in the web application and visualize the metrics sorting the data by different columns. Then, access to the "`Data`" view and check that the data displayed is the same. E.g., sort by `energy` in descendant order in both views and check that the lists match.

Appendix E

User documentation

E.1 Introduction

This section summarizes the elements that the user must have and the steps that must be followed to correctly execute the project.

E.2 User requirements

The technical user requirements to run the project are defined in the section D.3. As a summary:

1. Docker.
2. Docker Compose.
3. Python 3.

In addition to the technical requirements, the user must have at least a basic/intermediate knowledge level of the operating system that will be used to run the project.

> *The commands described for the installation and usage of the project are oriented to Linux environments.*

E.3 Installation

The steps that need to be followed to install the project are referenced in the section D.4. A simpler summary is shown below:

1. Open a `command prompt` and execute the following commands to set up the permissions of Airflow required folders:
   - `sudo chmod -R 777 spark/resources/history`
   - `sudo chmod -R 777 airflow/logs`
   - `sudo chmod -R 777 airflow/plugins`
2. Move to the `docker` folder and launch the environment with "`sudo docker-compose up -build -d`".
3. Wait until all the services have started and are in a healthy state.
4. Access to the Airflow UI (http://localhost:8080, user "`airflow`" and password "`airflow`") and start the DAG "`spark_main`" as shown in the figure D.2.
5. Use the command "`sudo docker-compose down`" to stop the process.

E.4 User's manual

The usage of the project is simple. After installing and running the project, including the activation of the Airflow DAG, all data results are presented to the user in the web application (http://localhost:8000), containing three views.

- **Home.** It shows a brief introduction of the project and an animated gif to illustrate the implemented data flow, along with a link to the repository on GitHub. The "Home" view and its components are referenced in figure E.1.
- **Data.** It displays a table with all the data extracted. The table has been made with Datatables and allows the user to:
  - **Column selector**. Allows to select the columns to be hidden or shown. The default view is shown with all columns hidden except "Popularity", "Danceability", "Energy" and "Duration(s)". The hidden columns are grayed out.
  - **Size selector**. Allows to select the amount of data to be displayed, limited to 10 (default), 25, 50 and 100.
  - **Search bar**. Allows to search the entire dataset, regardless of page or hidden columns.

  - **Page selector**. Allows to select the page to be displayed (first by default).
  - **Number of entries**. Shows the range of rows displayed on the page and the total number of entries in the dataset.
  - **Data displayed**. The displayed data can be sorted by clicking on the column headers. The default sorting is by date from most recent to least recent.

  The "Data" view and its components are referenced in figure E.2.

- **Visuals.** It displays a chart composed by with two data selectors, a multi-select to add data as a bar and a single-select to add data as a line. A script was added that does not allow both selectors to have the same column in their data, so when a column that is present in one of them is selected in the other, it is removed from the previous one. It also has a selector to choose the amount of data to display (values of 5, 10, 15 and 20, to avoid saturating the graph), as well as an order selector that allows the user to sort by any of the columns in ascending or descending order.

  - **Quantity selector**. Allows to select the amount of data to be displayed, limited to 5, 10 (default), 15 and 20.
  - **Order selector**. Allows to select the data sorting, with a drop-down to select the column and a switch to select the direction (ascending or descending). The default sorting is by date from most recent to least recent.
  - **Column selector for the bars**. Allows to select the columns to hide or show in the bar format. The default view displays only the "Popularity" column.
  - **Column selector for the line**. Allows to select the columns to hide or show in the line format. The default view does not display any columns (selected).
  - **Data displayed**. The displayed data can be hidden by clicking on the labels shown at the top.

  The "Visuals" view and its components are referenced in figure E.3.

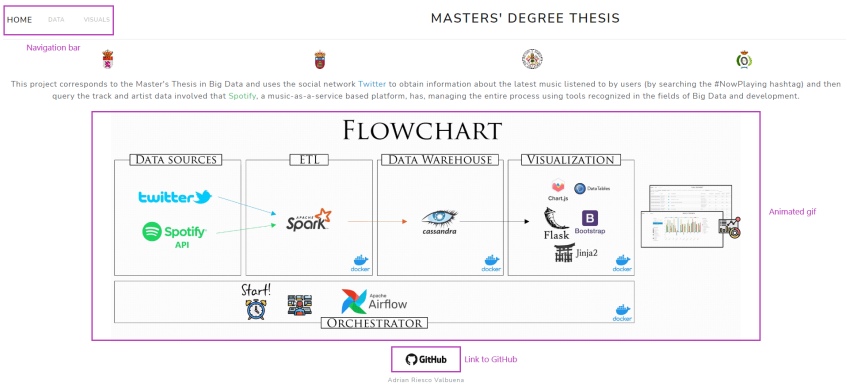

Figure E.1: Front-end home view

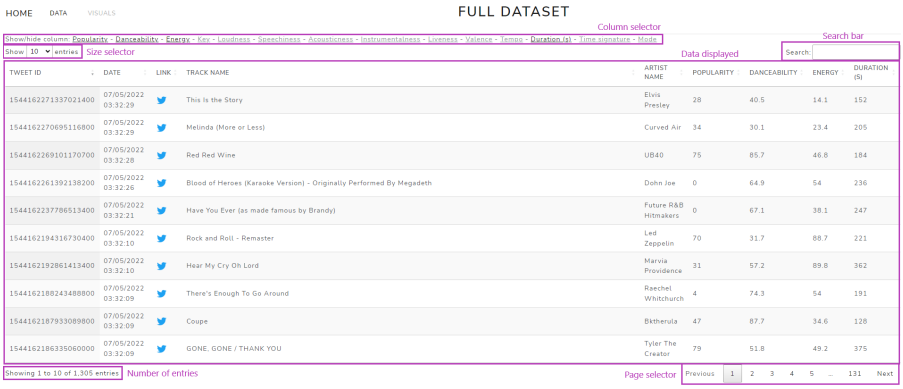

Figure E.2: Front-end data view

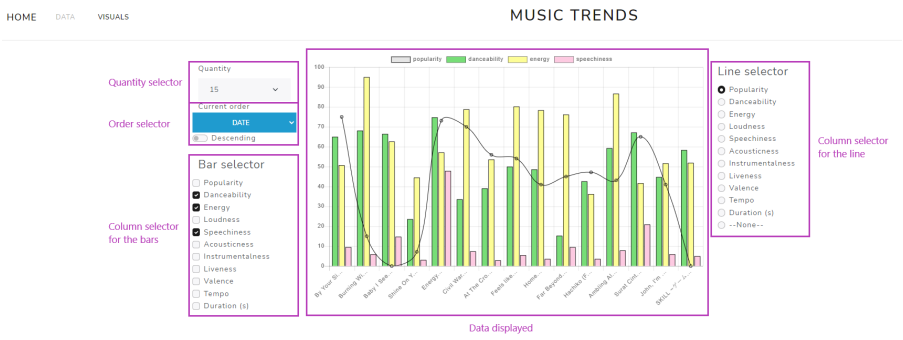

Figure E.3: Front-end graph view

Bibliography

[1] Ali Shah. Big data in healthcare, 2022. https://www.researchgate.net/publication/328859276_Big_Data_in_Healthcare_A_Survey (February 5th 2022).

[2] International Business Machines Corporation (IBM). Rest apis, 2022. https://www.ibm.com/cloud/learn/rest-apis#:~:text=An%20API%2C%20or%20application%20programming,representational%20state%20transfer%20architectural%20style. (February 12th 2022).

[3] Red Hat. Orchestrator, 2022. https://www.redhat.com/es/topics/automation/what-is-orchestration (March 19th 2022).

[4] International Business Machines Corporation (IBM). Containerization, 2022. https://www.ibm.com/cloud/learn/containerization (March 19th 2022).

[5] Red Hat. Orchestrator, 2022. https://www.redhat.com/en/topics/devops/what-is-ci-cd (May 17th 2022).

[6] Educative. What are template engines?, 2022. https://www.educative.io/answers/what-are-template-engines (June 11th 2022).

[7] Wikipedia. Github, 2022. https://es.wikipedia.org/w/index.php?title=GitHub&oldid=143733693 (May 27th 2022).

[8] Inc. Postman. Postman, 2022. https://www.postman.com (February 6th 2022).

[9] Wikipedia. Twitter, 2022. https://en.wikipedia.org/w/index.php?title=Twitter&oldid=1075330090 (February 19th 2022).

[10] Variety. Twitter daily users 2021, 2021. https://variety.com/2022/digital/news/twitter-q4-2021-earnings-users-growth-1235176882 (February 26th 2022).

[11] Inc. Twitter. Twitter developer, 2022. https://developer.twitter.com (February 5th 2022).

[12] Inc. Twitter. Recent search, 2022. https://developer.twitter.com/en/docs/twitter-api/tweets/search/introduction (February 19th 2022).

[13] Wikipedia. Spotify, 2022. https://en.wikipedia.org/w/index.php?title=Spotify&oldid=1077879878 (February 19th 2022).

[14] Spotify Technology S.A. Spotify for developers, 2022. https://developer.spotify.com (February 5th 2022).

[15] Spotify Technology S.A. Spotify for developers - search for item, 2022. https://developer.spotify.com/documentation/web-api/reference/#/operations/search (February 12th 2022).

[16] Spotify Technology S.A. Spotify for developers - get tracks' audio features, 2022. https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-audio-features (February 12th 2022).

[17] Apache. Apache airflow, 2022. https://airflow.apache.org (March 19th 2022).

[18] InfoWorld. Apache spark, 2022. https://www.infoworld.com/article/3236869/what-is-apache-spark-the-big-data-platform-that-crushed-hadoop.html (March 19th 2022).

[19] Apache. Apache cassandra, 2022. https://cassandra.apache.org/_/index.html (May 20th 2022).

[20] Wikipedia. Apache cassandra, 2022. https://es.wikipedia.org/w/index.php?title=Apache_Cassandra&oldid=143581116 (May 20th 2022).

[21] Pallets Projects. Flask, 2022. https://flask.palletsprojects.com/en/2.1.x/ (June 11th 2022).

[22] Bootstrap. Bootstrap, 2022. https://getbootstrap.com/ (June 11th 2022).

[23] W3Schools. Chart.js, 2022. https://www.w3schools.com/ai/ai_chartjs.asp (June 11th 2022).

[24] SpryMedia Ltd. Datatables, 2022. https://datatables.net/ (June 11th 2022).

[25] Inc. Docker. Docker, 2022. https://www.docker.com (February 19th 2022).

[26] Inc. Docker. Docker compose, 2022. https://docs.docker.com/compose/ (February 19th 2022).

[27] Inc. Twitter. Twitter developer agreement and policy, 2022. https://developer.twitter.com/es/developer-terms/agreement-and-policy (February 5th 2022).

[28] Spotify Technology S.A. Spotify developer terms, 2022. https://developer.spotify.com/terms/ (February 5th 2022).

[29] Apache. Apache license version 2.0, 2022. https://www.apache.org/licenses/LICENSE-2.0 (March 5th 2022).

[30] Pallets Projects. Flask license 2.1, 2022. https://flask.palletsprojects.com/en/2.1.x/license/ (March 26th 2022).

[31] Bootstrap Team. Bootstrap license, 2022. https://github.com/twbs/bootstrap/blob/main/LICENSE (March 26th 2022).

[32] Pascal Brachet. Texmaker, 2022. https://www.xm1math.net/texmaker/ (February 5th 2022).