SISTEMA DE RECOMENDACIÓN DE CANCIONES DE SPOTIFY

TRABAJO DE FIN DE GRADO DE INGENIERÍA INFORMÁTICA

Descripción breve

Aplicación realizada utilizando la API de Spotify con el objetivo de mostrar canciones relacionadas a una dada según una serie de parámetros.

Sergio Rivas Delgado

s.rivasd@alumnos.urjc.es

ÍNDICE

1. INTRODUCCIÓN

A) MOTIVACIÓN

Hoy en día disponemos de una gran cantidad de datos que generan millones de dispositivos electrónicos que tenemos a nuestro alrededor, desde relojes inteligentes con los que podemos llevar el control de nuestra ruta al correr hasta poder mandar un correo electrónico a nuestros compañeros de trabajo. De la misma manera que es interesante toda esta recolección de datos, cobra una gran importancia el poder compartirlo con los demás para darle un mayor valor o importancia a los objetivos conseguidos.

El crecimiento de la cantidad de información de la que disponemos está creciendo en los últimos años de manera exponencial. De esta manera nuestro conocimiento tiene que avanzar para equipararse con la enorme cantidad de datos de los que disponemos.

Aunque disponemos de gran cantidad de dispositivos y redes sociales para poder interactuar con todos estos datos, nos faltan herramientas apropiadas para poder procesarlos correctamente. En la actualidad vivimos en la época del *"Big Data"* en la que se suben una enorme cantidad de vídeos a *Youtube* o se publican millones de *Tweets* acerca de un suceso de actualidad en sólo minutos.

"Big Data" es el término que usamos para referirnos a una gran cantidad de datos, de manera que estos no se pueden procesar con las herramientas tradicionales. Sin embargo, hay que tener precaución a la hora de procesar estos datos y para conseguir un buen resultado debemos tener en cuenta la regla de las 4 V's. La primera de ellas y la más evidente es el **Volumen**, debemos tener en cuenta la capacidad de cómputo ya que se prevee que en el año 2020 los datos se multipliquen por 44 con respecto de los que disponíamos en 2009. La segunda de ellas es la **Variedad**, porque a mayor cantidad de datos, más tipos de los mismos puede haber, por lo que debemos de tenerlo en cuenta a la hora de procesarlos. La tercera V es la **Velocidad** ya que es importante ofrecer un buen servicio al usuario y proporcionarle la información en un tiempo aceptable. La última es la Veracidad, porque tenemos que saber discriminar entre los datos que nos van a aportar valor y los que no.

Ante esta falta de herramientas surgió mi motivación para llevar a cabo este trabajo de fin de grado, ya que me gusta que sea algo que aporte valor y que sea de utilidad para que en un futuro algún desarrollador más lo pueda usar o mejorar para proyectos muy interesantes.

A todo el mundo le gusta la música, ya sea *Pop*, *Rock*, *Electrónica*… En este sentido la aplicación más conocida mundialmente para poder escuchar, compartir, crear listas con tus canciones favoritas y muchas cosas más es Spotify.

Ilustración 1. Logo actual de la compañía Spotify.

Spotify se trata de una aplicación multiplataforma que es usada para la reproducción de música vía *streaming.* Cuenta con dos versiones, una gratuita y otra premium mediante la cual tenemos algunas ventajas como evitar publicidad y anuncios en la aplicación, mejorar la calidad del audio, poder escuchar tu música sin necesidad de una conexión a internet, poder tener más libertad a la hora de escuchar una canción, artista o lista de reproducción completa, o tener disponibles modos adicionales como el modo radio.

Se implantó el 7 de octubre de 2008 en el mercado europeo, pero no fue implantado en el resto del mundo hasta el año 2009. Podemos tener acceso a *Spotify* desde prácticamente todos los sistemas operativos del mercado (lo que es bastante poco común entre la mayoría de las aplicaciones en la actualidad) entre los que se encuentran *Microsoft Windows, Mac OS X, Linux, Windows Phone, Symbian, iOS, Android* y *BlackBerry*.

En cuanto a su origen cabe mencionar que es una empresa de origen sueco, concretamente de Estocolmo, aunque a lo largo de su desarrollo a firmado convenios con importantes discográficas como pueden ser *Universal Music, Sony BMG, EMI Music, Hollywood Records, Interscope Records* y *Warner Music.*

Se trata de una aplicación con una enorme cantidad de usuarios por lo que de esta manera cobra más sentido nuestro proyecto como ayuda al procesado de esta enorme cantidad de información. Según datos de junio de 2015, un total de 75 millones de usuarios están activos en la plataforma (de los cuales 20 millones son usuarios *premium*).

Por último, es interesante mencionar cómo funciona la transferencia de archivos de audio en *Spotify*. Se trata de una combinación de servidores dedicados al *streaming* y en la transferencia de red de pares (P2P) que serían los propios usuarios entre sí. No se necesita nada más que una conexión de 256 kbit/s y utiliza el códec de audio Vorbis (Ogg). Los usuarios premium, como ventaja tienen una calidad de audio superior a los usuarios básicos (q9).

B) DESCRIPCIÓN DEL TRABAJO DE FIN DE GRADO

El proyecto consiste en la implementación de una aplicación que actúe como recomendador de canciones de Spotify de una manera visual e intuitiva. Basta con que el usuario, una vez arrancada la aplicación, introduzca la canción con la que quiere comenzar su búsqueda (también se le da la posibilidad de introducir el nombre del artista o grupo para hacer la búsqueda más rápida y concisa). Esta zona está situada en la parte superior de la aplicación de manera que es fácilmente identificable en cuanto el usuario entra en la aplicación, aunque sea la primera vez que la está usando.

Una vez el usuario ha realizado la primera búsqueda, la aplicación se pone a trabajar con el API de Spotify para proporcionar un grafo que visualmente le proporciona una gran experiencia al usuario. Siguiendo los colores de la aplicación original de Spotify, los artistas se representarán mediante nodos grises, y las canciones del color verde característico de Spotify. En cuanto a la identificación, simplemente con pasar el cursor por encima de cada nodo podemos saber fácilmente de que artista o canción se trata. Mientras que la aplicación está trabajando con el API de Spotify nos aparecerá un icono de carga en la parte superior del grafo. Una de las dudas que surgió en esta parte fue cómo diferenciar canciones o artistas que habían sido incluidas en el grafo en una llamada anterior o posterior a la API. Esto fue resuelto de una manera muy fácil e intuitiva para el usuario, a medida que aumentamos las llamadas a la API, las canciones y artistas más recientes tendrán un mayor tamaño en la aplicación, quedando las canciones y artistas más antiguos en la aplicación de un tamaño inferior.

En relación con el grafo, se ha implementado una funcionalidad de gran utilidad para que el usuario se sienta cerca de su aplicación original de Spotify y nuestra herramienta le resulte realmente útil. Si realizamos *click* en un artista del grafo una vez se ha cargado totalmente en nuestra página, se nos abre una nueva pestaña en el navegador, que nos indica en la versión web de Spotify la página del grupo que hemos seleccionado (y por lo tanto el resto de sus discos, canciones o incluso próximas fechas y lugares de giras). Sin embargo, si hacemos *click* en una canción del grafo, además de continuar con la expansión del mismo buscando nuevas canciones relacionadas, se nos abre otra pestaña en el navegador con la reproducción de la canción en la versión web de Spotify desde donde podemos acceder a las demás canciones del disco. Hay que indicar que, para acceder a esta funcionalidad, el usuario tiene que iniciar sesión en Spotify para que el comportamiento y la experiencia sea la más adecuada.

Además de la zona de representación del grafo, contamos con una zona que nos muestra los detalles de la última canción seleccionada. Estos datos son el título de la canción, el artista de la misma, la duración de la canción, la popularidad dentro de la aplicación de Spotify y la fecha de lanzamiento del álbum, así como la foto del mismo de manera que sea una experiencia enriquecedora por parte del usuario a la hora de utilizar la aplicación.

Como mencionaré más adelante, durante el desarrollo del trabajo, he implementado varias mejoras en la experiencia de interacción del usuario con la aplicación. Con estas mejoras, tanto la zona de representación del grafo como la de los detalles de la canción seleccionada se actualizan en un corto periodo de tiempo para evitar esperas innecesarias.

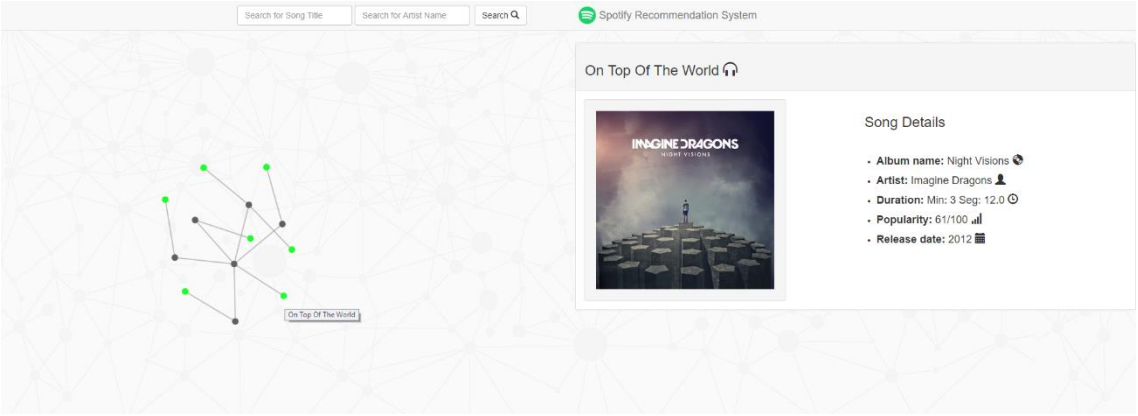


Ilustración 2. Captura de pantalla de un ejemplo de ejecución de la aplicación.

C) ESTADO DEL ARTE

Una vez te dispones a comenzar un proyecto que pueda ser útil para que otros usuarios puedan aprovechar tu trabajo para tener una buena experiencia, una parte importante del tiempo de preparación debe ir a visualizar otros trabajos anteriores que puedan estar relacionados con el tuyo.

Ilustración 3. Logo de la herramienta Google Académico.

Una herramienta muy útil para poder investigar de un solo vistazo información relevante para tu propósito es Google Académico. Esta versión del famoso buscador Google nos permite extraer información relacionada con el ámbito de estudio e investigación proporcionándonos PDFs, libros, *e-books* y todo tipo de archivos de interés.

En cuanto comencé a buscar me di cuenta de la falta que hacía una aplicación como la de mi proyecto ya que no abundan este tipo de trabajos de investigación acerca de cómo poder ofrecer al usuario una buena recomendación de canciones siguiendo sus gustos. No obstante, mostraré tres trabajos bastante relacionados y que me han parecido de gran interés:

Combining Spotify and Twitter Data for Generating a Recent and Public Dataset for Music Recommendation (PDF)

En primer lugar, tenemos esta aplicación realizado por *Martin Pichl, Eva Zangerle* y *Günther Specht.* Básicamente, se basa en sacar datos de la red social *Twitter* para interpretarlos y poder utilizarlos como huella digital para saber qué tipo de música les puede resultar afín a determinados usuarios con el fin de tener un recomendador de *Spotify*. Como comentan los autores en sus conclusiones se trata de un sistema limitado ante un gran número de recomendaciones además de tener que usar la relación entre dos redes sociales (lo que obliga a que el usuario de la aplicación esté dado de alta en ambas).

Deep content-based music recommendation (PDF)

Por otro lado, tenemos el proyecto de *Aaron Vandenoord, Sander Dieleman* y *Benjamin Schrauwen.* En este caso, nos hablan de la dificultad concreta de recomendar una canción y no tanto en la forma. Tienen en cuenta la gran cantidad de plataformas disponibles en la actualidad y la disparidad de criterios en cuanto a recomendar una u otra canción al usuario final. En su memoria nos intentan aportar soluciones acerca de la base de datos a utilizar o como predecir los factores de latencia a través del audio de la canción determinada. También es de gran importancia el aporte sobre la factorización de matriz ponderada, ya que como nos comentan los autores, muchas veces los usuarios no votan o valoran expresamente todas las canciones que escuchan. Todos estos aspectos a tener en cuenta nos ayudan bastante a entender el problema de la complejidad de recomendar canciones y como poder afrontar las posibles soluciones.

A Survey of Music Recommendation Aids (PDF)

Por último, me gustaría destacar la entrevista realizada por *Pirkka Åman* y *Lassi A. Liikkanen.* En esta encuesta nos informan acerca de la importancia que tienen los anuncios que nos recomiendan los sistemas de *streaming* online como pueden ser *Spotify* o *Last.fm.* Centrándonos en el caso de *Spotify*, destacan que se trata de una aplicación con interfaz muy intuitiva, facilita el acceso a nuestra música favorita al ser multiplataforma y tiene una gran calidad de audio. Pero por otro lado comentan que posee un sistema de recomendación bastante limitado. Lo que nos muestra una vez más la gran importancia y utilidad de nuestro proyecto.

2. OBJETIVOS

Una vez asignado el trabajo lo primero fue aprender las tecnologías que iban a ser empleadas para interactuar con el API de Spotify.

Por un lado, consistiría en aprender el lenguaje de programación *Python* que es el utilizado para hacer las llamadas a la API. Al haber ya desarrollado ya con lenguajes de todo tipo en la carrera se me hizo bastante fácil aprender la sintaxis de *Python*.

Por otro lado, debía aprender cómo funcionaba la base de datos *Neo4j*. La base de datos *Neo4j*, se trata básicamente de un grafo formado por nodos y aristas que representan la información y relaciones guardadas en la misma.

Comencé por aprender *Python* poco a poco mediante problemas y ejercicios simples como realizar funciones, bucles y algunas estructuras de control de flujo para irme familiarizándome con el lenguaje. En esta parte me resultó un poco complejo la manera de llamar a mis programas con los argumentos por consola, pero una vez solventados estos problemas se volvió mucho más rápido y sencillo el desarrollo.

Una vez ya con el lenguaje interiorizado me dispuse a realizar mis primeras pruebas con el API de *Spotify*. Me resultó fácil el registro para sacar los identificadores propios de mi cuenta de desarrollador, pero un poco más complejo el tema de realizar peticiones al API una vez que me autenticase ya que no estaba muy bien indicado el tema de las variables de entorno en *Windows*. El poder realizar muchos de estos ajustes en *PyCharm* ha hecho que sea un IDE con el que me he sentido muy cómodo a lo largo del desarrollo.

Posteriormente comencé a realizar diferentes peticiones basándome en ejemplos de la API y propios míos para entender y comprender de primera mano cómo funciona dicha API. Estos ejemplos fueron tales como los álbumes publicados por un artista, la búsqueda de canciones dentro de un álbum, la búsqueda de los propios artistas mediante la URI o simplemente su nombre, la búsqueda de artistas relacionados con uno dado y la búsqueda de las propiedades de una canción. Éste último ejemplo nos acerca bastante a lo que buscamos para poder implementar nuestra aplicación sobre Spotify.

Ya entrando un poco más en materia, implementé un par de ejercicios más complejos. El primero consistía en buscar con una profundidad 3 los artistas relacionados con uno dado, para saber de esta manera un conjunto de artistas relacionados entre sí. El segundo consistía en sacar todas las canciones de todos los álbumes de un artista determinado y de cada una de estas canciones sus principales propiedades como la bailabilidad, ritmo, positivismo...

Una vez llegados a este punto, comencé a familiarizarme con la base de datos *Neo4j* y su lenguaje para realizar consultas y demás operaciones con la base de datos, *Cypher*. El aprendizaje me resultó muy sencillo ya que es similar a *SQL* y ya estaba

acostumbrado a trabajar con bases de datos relacionales. Realicé algunos ejemplos de grafos sencillos con libros y películas relacionándolos entre sí.

Una vez aprendida las tecnologías a emplear, me pareció buena idea conectarlas en forma de ejercicio realizado anteriormente sobre las propiedades de las canciones de un artista dado. De forma que toda esta información se muestre en forma de grafo en la base de datos *Neo4j*.

En este paso del aprendizaje resultó problemático el tema de las relaciones entre nodos para ciertos ejemplos ya que estas relaciones son más apropiadas para asociar distintos tipos de entidades como pueden ser álbumes y canciones o canciones y artistas.

Una vez que hemos conseguido almacenar la información en la base de datos *Neo4j*, el próximo paso es representarla correctamente. Debido a la poca información y documentación en internet con respecto al grafo de *Neo4j*, me he visto obligado a utilizar la herramienta *Bokeh* para representar dicha información almacenada.

Debido al desconocimiento de dicha librería de Python he procedido a instalarla, que gracias al IDE *Pycharm* ha sido muy sencillo de realizar. Antes de ponernos con el problema real y cohesionarlo todo, he ido poco a poco implementando aplicaciones más sencillas para familiarizarme con la librería.

Teniendo en cuenta las exigencias del problema, decidí utilizar el *framework* de Python llamado *Flask* mediante el cual se pueden crear fácilmente interfaces gráficas utilizando la herramienta de D3JS en lugar de *Bokeh* como había planteado en un principio. Una vez encontrada una aplicación de ejemplo el siguiente paso es adaptarla a nuestro problema concreto. Cuando haya completado esta tarea, el siguiente paso será documentarse acerca de la maquetación con D3JS.

Tras unas semanas familiarizándome con *Flask*, conseguí hacer una aplicación de prueba que guardaba la información de la API en *Neo4j* y posteriormente mostraba el grafo en la pestaña del navegador. Esta aplicación relacionaba un álbum dado con las canciones de este. El siguiente paso es profundizar más con la API y con *Flask* para representar relaciones más complejas.

Flask me planteó varios problemas al principio ya que los métodos tenían que devolver un objeto *JSON* para que *D3JS* pudiese interactuar con los datos de manera correcta. Una vez solventados estos problemas solo planteó ciertas dudas el tema del algoritmo de la aplicación en sí. Una vez en este punto, la primera aproximación de la aplicación sólo permitía mostrar las canciones y artistas relacionados a uno dado sin poder profundizar más. Otro de los problemas que me dio *Flask* fue a la hora de transmitir información desde la interfaz y cuantas URLs iba a necesitar de cara a realizar búsquedas de canciones o artistas. Sin embargo, se solucionó de una manera muy simple, a través de parámetros. De esta manera, si el usuario sólo introduce el nombre de la canción y no el del artista, también se realiza la búsqueda. La cuestión determinante de esta aplicación es que también se puede utilizar ese método al

realizar las siguientes búsquedas haciendo *click* en una canción del grafo, ya que se obtiene el nombre de la misma y se llama a esa URL.

El próximo paso fue poder mostrar en la herramienta web de Neo4j todas las relaciones correctamente antes de pasar a *D3JS* en el navegador web. Lo más difícil en este apartado fue distinguir entre todos los niveles de llamada a la API para poder formar correctamente la estructura. Además, como suele pasar con el tema de las APIs siempre suele haber ciertas excepciones con algunos datos como puede ser que una canción esté compuesta o representada por varios artistas, y eso nos puede llevar a error a la hora de poder guardar la información en la base de datos. Esto fue solucionado escogiendo al artista con más relevancia o importancia en el álbum correspondiente a la hora de conformar las relaciones.

Una vez finalizada esta aproximación había que transmitir la información de la base de datos *Neo4j* a la representación en el navegador web mediante *D3JS.* La información en internet acerca de esta librería es bastante extensa y diversa, ya que se usa para multitud de diferentes proyectos de diferente índole. Sin embargo, gracias a un aporte de mi tutor a modo de diapositivas de ciertos ejercicios me pude familiarizar más rápidamente con la librería.

Como ya comentaré más adelante en detalle, en este punto del trabajo he implementado dos aproximaciones diferentes. Una mediante *Flask* devolviendo información a través de *JSON* y otra accediendo a la base de datos desde código *JavaScript*.

La primera aproximación podemos decir que es la más modularizada y estructuralmente correcta. Aunque el principal problema venía en la pobre experiencia del usuario ante grandes volúmenes de datos. La segunda, solventa estos problemas de esperas no deseadas, pero de cara a la estructura sea la menos habitual.

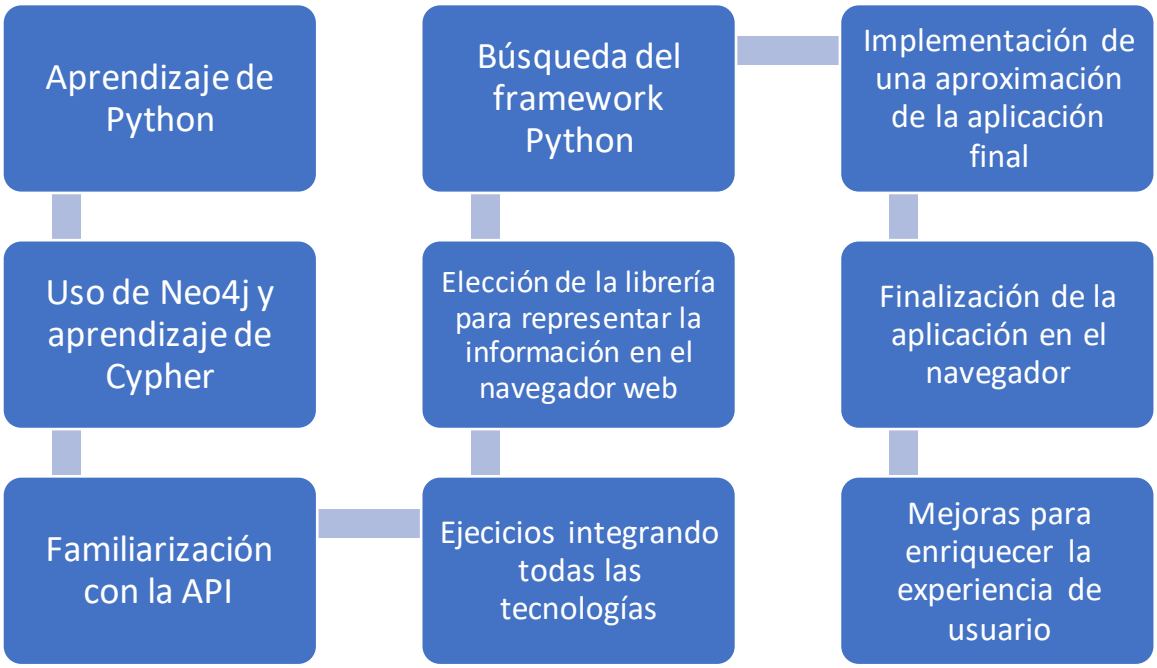
Con la versión de *Flask* use *jQuery* para traer la información de la API para el módulo de detalles de la canción en la interfaz web. Me resultó bastante fácil porque ya había tratado con ello en el grado, más concretamente en la asignatura de desarrollo de aplicaciones web.

Una vez implementada la funcionalidad básica de la aplicación, pensé que sería de gran ayuda para el usuario mejorar la relación de nuestra herramienta con *Spotify* en sí (más allá de su obvia conexión con la API). En este aspecto se me ocurrió el tema de abrir una pestaña nueva en el navegador cada vez que el usuario quisiera interactuar con un artista o una canción del grafo. Tenía dos formas de hacerlo, mediante el nombre de uno u otro, o a través del código URI de Spotify. Comencé a implementar la primera alternativa, pero la descarté por lo siguiente. La pestaña que se habría nos llevaba a una página que realizaba automáticamente la búsqueda en la base de datos de *Spotify* por nosotros. El problema viene cuando es una canción con un título bastante común como puede ser *Tatoo.* Si nos encontramos en este caso puede ser que nuestra canción se encuentre de las últimas en el listado y se nos haga bastante tedioso encontrarla.

Por las anteriores razones opté por la segunda opción. La única dificultad añadida era traer la información del URI en cuestión desde la base de datos. Una vez hecho esto sólo teníamos que asignar la URL en función de si se trataba de un artista o de una canción teniendo en cuenta el tipo de nodo seleccionado. De esta forma la pestaña que se nos abre es bastante más directa y relacionada con lo que podemos esperar desde un principio. En el caso de que hagamos *click* en un artista, se nos abre directamente la página del artista en Spotify con toda su información de relevancia. En el caso de una canción (además de realizar la búsqueda de las canciones relacionadas a través de la API) se nos abre una pestaña con la canción implicada reproduciéndose y el disco al que pertenece con las demás canciones del mismo.

Para finalizar y darle un toque para que el usuario tenga más control de la aplicación, he añadido un par de barras de carga para que el usuario sepa en todo momento cuando la aplicación está trabajando para traerle toda la información. Esto es debido a que la velocidad de la API es limitada y para un proyecto que busca tanta información es bastante trabajoso.

A continuación, muestro un esquema para indicar más visualmente los pasos u objetivos que he ido cumpliendo en el desarrollo del proyecto.



3. DESCRIPCIÓN ALGORÍTMICA

La estructura que sigue el grafo es la siguiente. En primer lugar, busca la canción indicada por el usuario y la introduce en el grafo. Posteriormente encuentra el artista relacionado con dicha canción buscada. Este artista puede ser introducido por el usuario a mano, aunque si no lo hace, la herramienta lo busca en la API, aunque dificulta el rendimiento.

Una vez tenemos en nuestra base de datos la canción y el artista iniciales, el siguiente paso es buscar todos los artistas relacionados con el que poseemos actualmente según el criterio de *Spotify*. A priori no se introducen todos ya que puede ser que no sea necesario y esto empeoraría el funcionamiento de nuestra aplicación. Según vamos recorriendo los artistas relacionados, vamos mirando en los discos que tienen publicados y en sus correspondientes canciones para ver si estas canciones están relacionadas con la que hemos comenzado nuestra búsqueda.

El criterio que hemos seguido para indicar que una canción es parecida a otra es el siguiente. Si observamos detenidamente la API de *Spotify* tenemos que cada canción dispone de una serie de atributos que la caracteriza y que podemos utilizar a nuestro propio criterio para diseñar nuestro algoritmo. Los atributos en los que nos hemos basado son los siguientes:

- *Danceability:* Esta propiedad nos indica como de apropiada es una canción para ser bailada de acorde a elementos como el tempo, la estabilidad del ritmo, la fuerza o la regularidad total. El valor de 0 indica la canción con menos capacidad para ser bailada y un valor de 1 la que más capacidad tendría.
- *Energy:* Nos representa una medida de la intensidad y la actividad. Por lo general, las canciones con más energía se sienten más rápidas, altas y ruidosas.
- *Liveness:* Detecta la presencia de público en la grabación. Un alto nivel aumenta la probabilidad de que la canción haya sido grabada en vivo (sobre todo si supera el valor de 0.8).
- *Mode:* Indica la modalidad (menor o mayor) de una canción. El tipo de escala a partir del cual se deriva su contenido melódico. El modo mayor se representa con un 1 y el menor con un 0.
- *Speechiness:* Detecta la presencia de palabras habladas en una canción. Cuanto más discurso tiene el audio (como puede ser el caso de un audiolibro) más cercano a 1 es el atributo.
- *Acousticness:* Es una medida acerca de cómo de acústica es la melodía. El valor de 1 representa una gran firmeza de que la canción es acústica.
- *Instrumentalness:* Predice si la canción no tiene contenido vocal. Cuanto más cercano a 1 es el valor, más probabilidad tiene esa ausencia de contenido vocal.

- *Valence:* Por último, este atributo mide cuánta positividad conlleva una determinada canción. Las canciones con un nivel elevado son más felices o

eufóricas mientras que las que tienen un valor bajo son más tristes o depresivas.

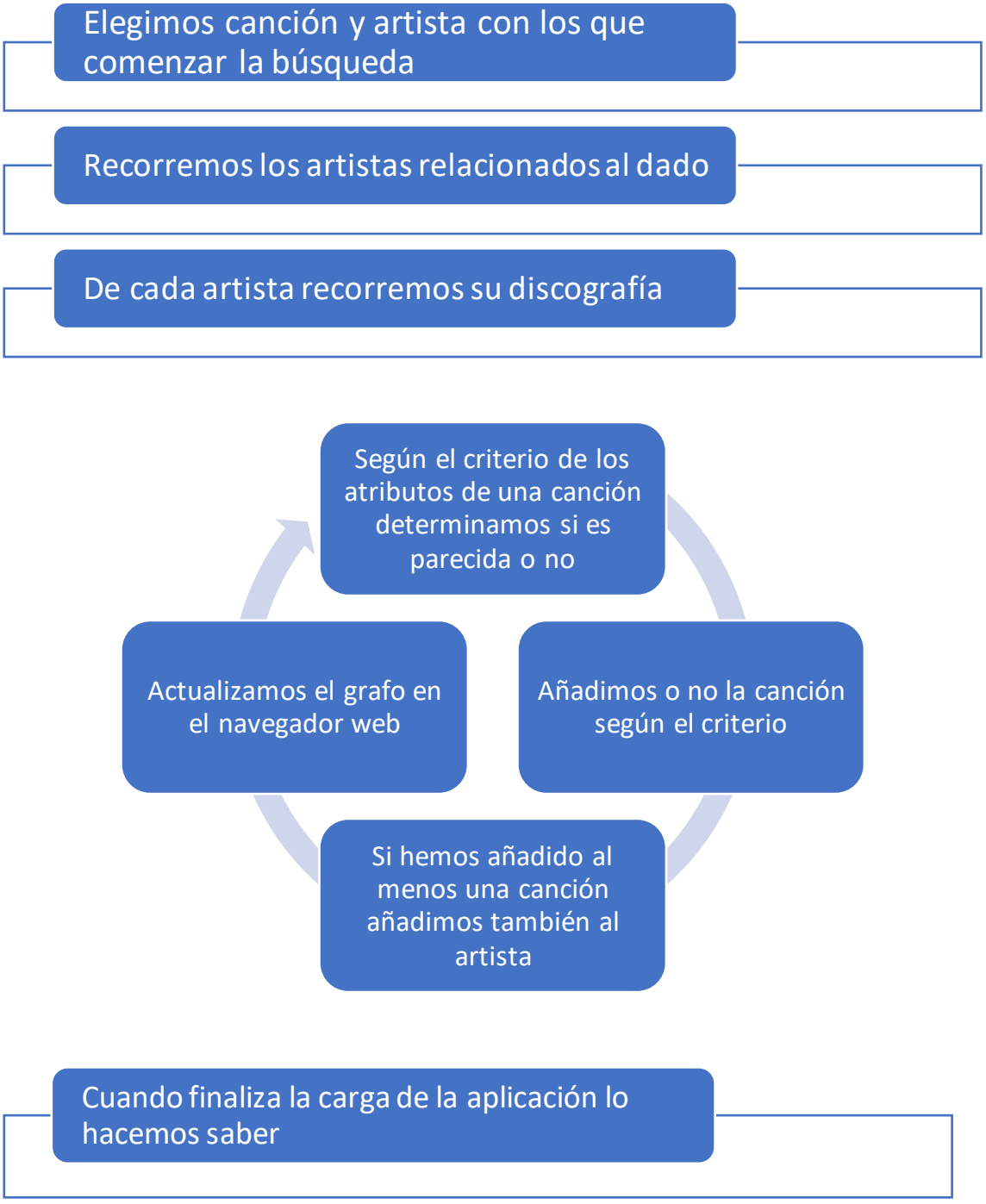
Una vez que almacenamos todos estos valores de nuestra canción original, procedemos a compararlos con los que vamos obteniendo del resto de canciones que vamos recorriendo. Si la diferencia de valores entre todos los atributos de ambas canciones comparadas es inferior a 0.4 deducimos que se trata de una canción parecida. Este valor ha sido extraído a partir de un gran número de experimentos que han definido este valor cómo el más apropiado ya que no muestra demasiadas canciones, pero sí las necesarias.

Procedemos a recorrer las canciones del artista y una vez que hemos llegado al final es cuando tomamos la decisión acerca de incluirle o no en nuestro grafo. En el caso de que se haya añadido al menos una canción sí que debemos de incluir a dicho artista en nuestra representación. Por el contrario, si no se han encontrado canciones relacionadas a la dada en dicho artista tenemos que prescindir de él por el momento.

De esta forma vamos iterando por todos los artistas relacionados y su correspondiente discografía. Cuando hemos finalizado dicho proceso decimos que hemos llegado al final de la iteración y el grado tiene otro nivel más añadido. Para mostrarlo más gráficamente, una iteración o ciclo se representaría de la siguiente manera:

Esta estructura cambia de cierta manera con la segunda aproximación que realicé en el algoritmo. Como ya comenté anteriormente, el tema de devolver *JSON* con *Flask* y representarlos mediante *D3JS* producía un cuello de botella que era conveniente mejorar. De esta manera, el acceso desde *JavaScript* a *Neo4j* para ir actualizando

periódicamente el grafo mejora notablemente la experiencia de usuario. La manera de actualizar el ciclo de trabajo de la aplicación con este cambio sería de la siguiente manera.



Paralelamente a la carga y actualización del grafo, también se realiza la descarga de los detalles de la canción seleccionada. Se reciben los datos de la API para mostrarlos paralelamente a la carga del grafo para mejorar notablemente la experiencia de usuario.

4. DESCRIPCIÓN INFORMÁTICA

En este apartado vamos a entrar más en detalle acerca de por qué he escogido estos lenguajes y herramientas para sacar adelante este proyecto y los detalles y curiosidades de todos y cada uno de ellos. También volveré a hacer hincapié en las dos implementaciones realizadas (tanto la que incluye código *JavaScript* y *Flask* como la que sólo contiene *Flask* en relación con la base de datos *Neo4j*).

A) LENGUAJES, IDE, FRAMEWORKS Y LIBRERÍAS

- ***PYTHON***

Ilustración 4. Logo del lenguaje de programación Python.

Se trata de un lenguaje de programación interpretado cuya filosofía es la realización de un código legible para cualquier programador que se enfrente a él. Cabe destacar la gran virtud de que se trata de un lenguaje multiparadigma, lo que significa que soporta orientación a objetos, programación imperativa y programación funcional. Además de todo esto es un lenguaje interpretado de tipado dinámico y multiplataforma.

Fue diseñado en 1991 por parte del holandés *Guido van Rossum,* es administrado por la *Python Software Foundation* y posee una licencia de código abierto denominada *Python Software Foundation License* que es compatible con la licencia pública general de GNU (a partir de la versión 2.1.1 del lenguaje).

Una de las características que podemos reseñar es la resolución dinámica de nombres. Esto quiere decir que relaciona un determinado método y el nombre de una variable durante la propia ejecución del programa. Otra característica bastante reseñable es la facilidad que nos proporciona este lenguaje para su extensión o modificación. Incluso

es bastante sencillo de transcribir a otros lenguajes de programación en caso de necesitarlo como puede ser C o C++.

Una buena manera de medir la repercusión de un lenguaje de programación es a través del índice *TIOBE*. Este índice mide la popularidad e importancia de todos los lenguajes de programación utilizados. En mayo de 2018, se encuentra cuarto, solamente por detrás de lenguajes tan importantes como *Java, C* o *C++*. Sin embargo, es un lenguaje que se encuentra al alza como podemos observar en la siguiente gráfica que nos proporciona el índice *TIOBE*:

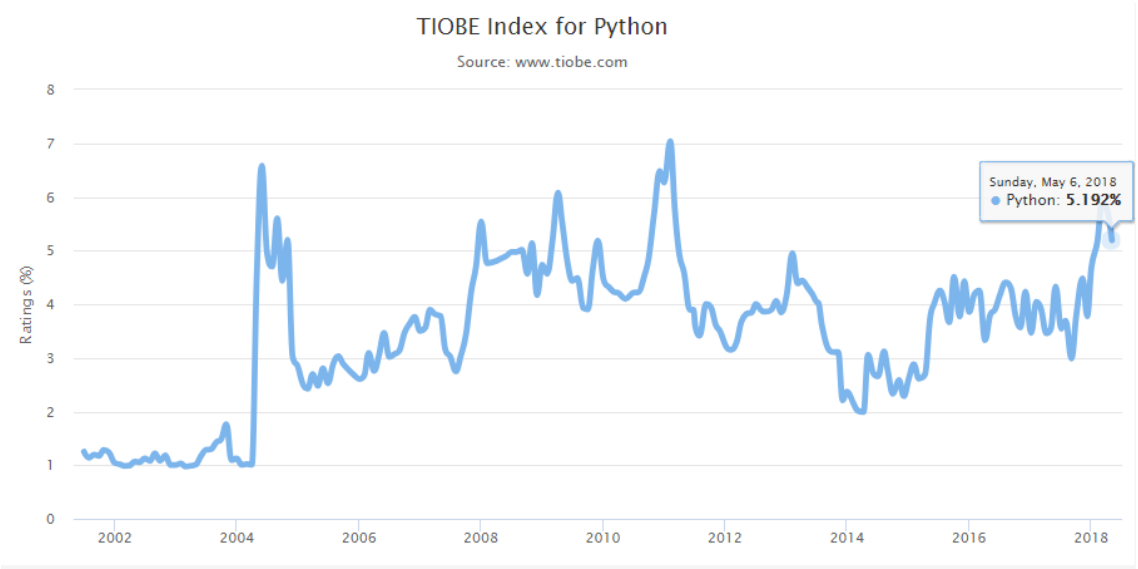


Ilustración 5. Gráfica de la importancia del lenguaje Python en el índice TIOBE.

Un aspecto más que relevante que hay que mencionar sobre Python es que muchos expertos en la materia lo han bautizado como el lenguaje del futuro. Esto es debido a su gran capacidad de utilización en ramas de la tecnología tan punteras como pueden ser el *Big Data*, la *Inteligencia Artificial*, el *Data Science* o la *Ciberseguridad* (como lenguaje de Scripting). Todo esto unido a que la comunidad del lenguaje es cada vez más grande, hay más consultas en *Stack Overflow* y hay más documentación disponible hace que los desarrolladores tengamos que seguir muy de cerca a este lenguaje en los próximos años.

En el aspecto personal, era un lenguaje de programación con el que no había trabajado y, una vez desarrollado el proyecto, sólo tengo buenas palabras para él. Sinceramente, una vez que has aprendido los primeros lenguajes de programación, el resto ya resultan más sencillos debido a la experiencia adquirida. No obstante, *Python* proporciona tal cantidad de funcionalidades intuitivas que hacen de la implementación a través de él una buenísima experiencia que repetiré en un futuro cercano sin duda.

- ***JAVASCRIPT***

Ilustración 6. Logo del lenguaje de programación JavaScript.

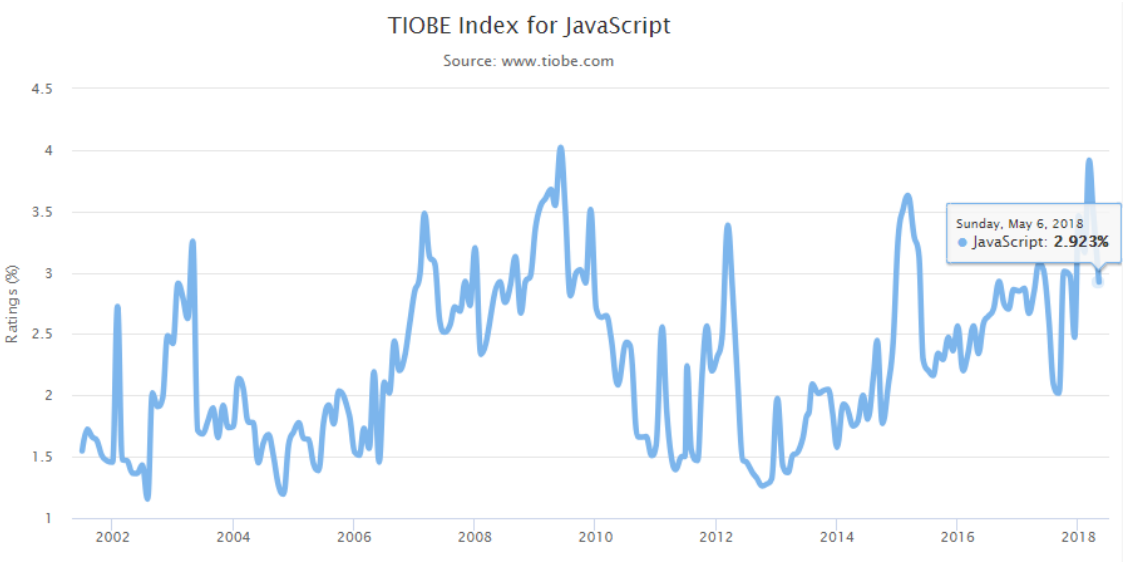
Es un lenguaje de programación interpretado proveniente del estándar *ECMAScript*. Sus principales características son la orientación a objetos, que está basado en prototipos, es impertativo y tiene un tipado débil y dinámico lo que le proporcionan una gran flexibilidad.

Apareció en 1995 siendo diseñado por *Netscape Communications Corp* y *Mozilla Fundation.* Su propósito original era proporcionar servicio en el lado del cliente de manera que aportase ciertas funcionalidades extra a la interfaz de usuario mediante uno o varios *Scripts*. No obstante, hoy en día podemos tener código *JavaScript* también en el código del servidor o en cualquier otro lugar, como es el caso de *Nodejs*. También es ampliamente utilizado hoy en día para realizar llamadas *AJAX* y compartir así información entre cliente y servidor de manera dinámica.

Un tema de bastante controversia relacionado con este lenguaje es el tema de sus versiones. Desde 2012, todos los navegadores modernos soportan *ECMAScript 5.1* aunque la sexta versión está liberada desde julio de 2015.

Desde mi punto de vista, es un lenguaje con bastante utilidad. En este sentido, también tenemos que tener en cuenta que tiene que ser utilizado en proyectos en los que aporte valor y no de cualquier manera. Desde mi experiencia personal, puede ser un lenguaje frustrante para un principiante ante la pobre gestión de errores. Una vez que solventas los primeros pasos, es un lenguaje que puede ser de gran importancia en la carrera profesional de cualquier programador.

Por realizar una comparativa con *Python, JavaScript* se encuentra en la séptima posición del índice *TIOBE*. Se trata de un lenguaje con bastantes altibajos debido a su particular estilo que no es del gusto de todos los programadores. A continuación, la gráfica de la importancia de JavaScript según el índice *TIOBE*:



- ***HTML***

Ilustración 7. Logo del lenguaje de marcado HTML.

HTML no se trata de un lenguaje de programación en sí. Como sus siglas indican, se trata en un lenguaje de marcas de hipertexto que se usa la elaboración de páginas web.

Su lanzamiento oficial data de 1997 y es un estándar a cargo de la *World Wide Web.* Sirve de referencia del software que conecta con la elaboración de páginas web en diferentes versiones y proporciona una estructura y forma al código.

Su filosofía de desarrollo se basa en la diferenciación. Cuando queremos añadir un elemento externo a la página, basta con hacer una referencia a él sin necesidad alguna de incrustarlo directamente. No obstante, al haber una gran cantidad de plataformas diferentes y también una variedad de versiones del lenguaje, se han suprimido y añadido diferentes funcionalidades a lo largo del desarrollo por lo que hay que tener esto en cuenta a la hora de utilizarlo en nuestras aplicaciones web.

En mi caso ya tenía experiencia con este lenguaje de marcado debido a que he realizado varias aplicaciones tanto individualmente como con más compañeros. Debido a ello, apenas me ha costado trabajo realizar una simple interfaz para alojar esta herramienta de recomendación.

- ***CSS***

Ilustración 8. Logo del lenguaje de estilo CSS.

Al igual que lo comentado anteriormente con *HTML*, el lenguaje *CSS* tampoco se trata de un lenguaje de programación al uso, pero ha sido de vital necesidad para el desarrollo de nuestra aplicación. Es definido como un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado previamente implementado con un lenguaje de marcado como puede ser el propio *HTML*.

Se suele usar habitualmente la combinación entre HTML, CSS y JS para dotar a las aplicaciones web de un atractivo visual de cara a crear un buen vínculo con el usuario.

Aunque también se permite incrustar código de estilo en el propio lenguaje de marcado, la correcta estructuración del lenguaje de estilo es aparte, en archivos .css destinados a tal fin. Por medio de este conjunto de sentencias se aplican capas, layouts, colores, tamaños y diferentes tipos de fuentes a un determinado elemento previamente seleccionado con un selector.

A mi modo de ver, resulta un lenguaje bastante intuitivo a la hora de dar los primeros pasos. Pero a medida que queremos realizar acciones más complejas, resulta que cuenta con una complejidad que aumenta de manera exponencial.

Para romper una lanza a su favor, hay que mencionar qué si se encuentra estructurado correctamente, con cambiar una sola línea se puede cambiar el estilo de una página completa.

Por último, existen SASS Y SCSS que amplían dicho lenguaje CSS con el fin de poder realizar acciones más complejas de una manera mucho más sencilla, sobre todo en grandes proyectos.

- ***PyCharm***

Ilustración 9. Logo del IDE PyCharm.

Para desarrollar este proyecto el IDE utilizado ha sido *PyCharm*. Se trata de un entorno de desarrollo integrado usado en programación, especialmente para el lenguaje *Python*.

Es desarrollado por *JetBrains* (su primera versión data de julio de 2010) al igual que otro IDE bastante conocido para programar en *Java* como es *Intellij.*

También cabe destacar que es multiplataforma ya que puede usarse tanto en *Linux*, como en *macOS* y *Windows*. La versión de comunidad está publicada bajo licencia Apache, aunque nosotros hemos tenido acceso a la edición profesional por el hecho de utilizarlo como estudiantes.

En particular, en mi caso de uso me gustaría destacar la facilidad para establecer variables del sistema, la gran utilidad del terminal integrado en el entorno, la fácil sincronización con *Git* de cara a tener mi propio control de versiones e incluso la integración con todo tipo de *frameworks* de *Python*.

- ***Flask***

Ilustración 10. Logo del framework de Python Flask.

Se trata de un *framework* minimalista escrito en Python que permite la creación rápida de aplicaciones web en el lenguaje de programación anteriormente mencionado.

Está basado en la especificación *WSGI* de *Werkzeug* y el motor de templates *Jinja2* y tiene una licencia *BSD*. También podemos hablar de que se trata de un *framework* multiplataforma.

Es perfecto para nuestra aplicación ya que apenas necesitamos una página para representar toda la información de manera útil. Sin embargo, si quisiéramos ampliarla en un futuro sería recomendable tener en cuenta otros *frameworks* como podría ser *Django*.

- ***D3js***

Ilustración 11. Logo de la librería D3js.

A la hora de representar el grafo en nuestra aplicación web necesitamos la ayuda que nos proporciona *D3js*. Es una librería de JavaScript que sirve para representar, a partir de unos datos dados, infogramas dinámicos e interactivos en el entorno web.

Esta librería hace uso de *SVG, HTML* o *CSS*. Es sucesora de la anterior librería empleada en estos casos conocida como *Protovis*. Su primera versión es del 18 de febrero de 2011, desarrollada por los autores *Mike Bostock* y *Jeffrey Heer.*

En cuanto a los diferentes pasos que sigue para realizar la representación podemos mencionar las selecciones, las transiciones, la asociación de datos y la gestión de nodos basada en los datos.

Como herramientas o bibliotecas alternativas a D3js podemos mencionar *AnyChart, Datacopia, gnuplot* o *Matlab*.

Mi experiencia con la librería ha sido positiva, aunque me gustaría mencionar que hay un gran salto entre la sintaxis para principiantes y los ejemplos más sofisticados, lo que puede despistar bastante a la hora de aprender a usar de manera correcta esta librería gráfica.

- ***JQuery***

Ilustración 12. Logo de la librería jQuery.

Para finalizar este apartado, para realizar la primera versión de la aplicación he contado con *jQuery*. Es conocida como una librería multiplataforma de JavaScript creada y desarrollada por *John Resig*, siendo lanzada el 26 de agosto 2006.

Su principal funcionalidad es el acceso al DOM del HTML, manejar eventos, desarrollar animaciones o realizar ciertas peticiones AJAX (como es nuestro caso). Actualmente está perdiendo bastante importancia debido a la aparición de *frameworks* como *Angular* o *React* pero para ciertas acciones puntuales continúa siendo de gran ayuda.

Es software libre y código abierto y cuenta con una gran cantidad de documentación y dudas resultas a lo largo de toda la web.

A modo de curiosidad, empresas como *Microsoft* o *Nokia* incluyen esta tecnología en sus plataformas como por ejemplo en el conocido *IDE* de *Microsoft Visual Studio*.

B) BBDD

- ***Neo4j***

Ilustración 13. Logo de la base de datos Neo4j.

Con respecto a la elección de la base de datos, había que tener en cuenta que fuese una tecnología empleada en la actualidad y con bastante potencial de futuro. Además, que tuviese una herramienta visual para facilitar al desarrollo y tuviese la suficiente flexibilidad para los datos que íbamos a obtener directamente desde el API de *Spotify*.

Técnicamente, se trata de un software libre de base de datos orientada a grafos, que se encuentra implementada en *Java*. Los desarrolladores definen a Neo4j como un motor de persistencia embebido, basado en disco, transaccional y que almacena los datos en grafos en lugar de tablas como realizan las bases de datos relacionales.

Está desarrollada por *Neo Technology* y su lanzamiento oficial data de 2007. Es una *startup* proveniente de Suecia que tiene base en *Malmö* y en *San Francisco Bay Area* en Estados Unidos.

Otra característica interesante que tiene esta base de datos es que cuenta con un lenguaje de *queries* propio como es *Cypher*. Se trata de un lenguaje que sirve para realizar consultas. Es un lenguaje intuitivo y bastante fácil de aprender, con sentencias simples y cortas con respecto a *SQL*, por ejemplo.

Las relaciones, siguiendo la sintaxis de *Cypher*, se representan mediante flechas y los nodos se indican entre paréntesis. Dichos nodos y relaciones se pueden etiquetar dando aún más modularidad a esta base de datos.

Para finalizar, el tiempo de respuesta de una *query* en *Neo4j* depende de los nodos y relaciones que debe recorrer y no de la base de datos en total. Esta circunstancia hace que la eficiencia sea una de las prioridades para los desarrolladores de esta novedosa base de datos orientada a grafos.

C) OTRAS HERRAMIENTAS

- ***GitHub***

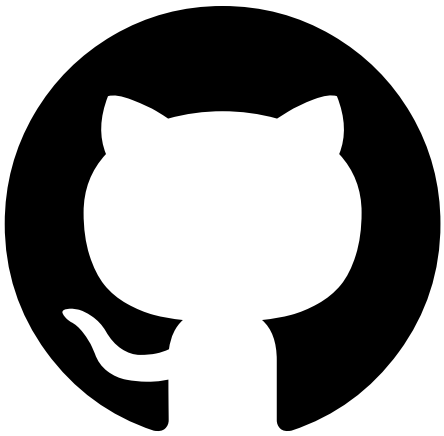


Ilustración 14. Logo de la plataforma GitHub.

Con el fin de tener un buen control de versiones y no perder ningún tipo de información a medida que avanzaba el desarrollo, me ha sido de gran utilidad la creación de un repositorio en *GitHub*.

Estamos hablando de una famosa plataforma de desarrollo colaborativo que sirve para alojar proyectos utilizando el sistema de control de versiones *Git*. Típicamente se usa para compartir desarrollo de programación de aplicaciones, aunque puede ser usado para compartir cualquier tipo de archivos y documentos.

El software que actúa por debajo de *GitHub* está implementado en el lenguaje de programación *Ruby*. Desde enero de 2010 está disponible bajo el nombre de GitHub, aunque anteriormente la plataforma era conocida como *Logical Awesome.*

El código se almacena en los repositorios, aunque estos tienen dos modos. Por defecto, los usuarios solamente pueden tenerlos en modo público. No obstante, con licencia de estudiante o a través de previo pago podemos tener el privilegio de alojar nuestro código en un repositorio. Esto no ha afectado para nada a nuestro proyecto ya que se trata de software libre y no requerimos ningún tipo de privacidad.

- ***Trello***

Ilustración 15. Logo de la aplicación Trello.

De cara a tener las tareas de un modo más ordenado y estructurado, me ha resultado de gran ayuda incorporar *Trello* al proyecto.

Es un software de administración de proyectos con interfaz web, cliente para iOS y Android para la organización de proyectos.

Fue desarrollado por la compañía Fog Creek Software y su lanzamiento oficial fue el 13 de septiembre de 2011. Está desarrollada principalmente en JavaScript y es multilenguaje.

Básicamente se trata de un tablón virtual en el que se puede añadir tarjetas para organizar las actividades de tu proyecto o cualquier idea de carácter personal. Sigue una filosofía Kanban y tiene funcionalidades tales como añadir comentarios, adjuntar archivos, etiquetar eventos o compartir el tablero con más usuarios.

- ***Google Chrome***

Ilustración 16. Logo del navegador Google Chrome.

Para las pruebas de interfaz y revisión de ciertas partes puntuales de código (especialmente JavaScript) he usado el navegador *Google Chrome*. La elección se ha debido principalmente a que su interfaz de las herramientas para desarrolladores es de gran calidad.

Se trata de un navegador web de código cerrado desarrollado por *Google*, aunque está derivado de otros proyectos de código abierto. Está disponible para descargar gratuitamente en cualquier plataforma o sistema operativo. Su lanzamiento oficial fue el 2 de septiembre de 2008.

Es ampliamente conocido y elegido por muchos como el mejor navegador web. Cuenta con un total de 750 millones de usuarios.

Como funcionalidades avanzadas posee la posibilidad de agregar y cambiar temas visuales en la interfaz de *Chrome*, un gran soporte de extensiones, traducción automática y sugerida de sitios web, y visor nativo de *PDFs* entre otras muchas.

Por último, cuenta con opciones extras con respecto a otros navegadores del mercado como puede ser la búsqueda por voz, las páginas ocultas, las opciones experimentales, los atajos de teclado y ratón y el ahorro de datos.

- ***API Spotify***

Ilustración 17. Logo de la API de Spotify.

Para dar fin a este apartado queda por mencionar una de las herramientas usadas más imprescindible del proyecto, el API de *Spotify*.

La interfaz de programación de aplicaciones API es una interfaz de línea de comandos que permite a los desarrolladores para realizar la codificación y el material relacionado para asegurarse de que la aplicación se desarrolla de la mejor manera.

Como es lógico, para poder acceder a la API de Spotify tienes que tener una cuenta en la propia plataforma aportando tus datos. Hay que destacar también la importancia de que una API de un servicio tan importante en la red sea gratuita, permitiendo así que miles de desarrolladores en todo el mundo puedan crear una comunidad de gran importancia.

Una vez realizado nuestro acceso, disponemos de un token y un nombre de usuario con el que se nos permite hacer un número limitado de consultas por unidad de tiempo.

En cuanto a mi experiencia personal, me ha resultado bastante fácil la integración con el entorno *PyCharm* ya que aporta bastante información acerca de cómo utilizar los métodos del API.

D) PRIMERA VERSIÓN DE LA APLICACIÓN

En este apartado ya vamos a entrar con más detalle en cómo está implementada la aplicación viendo la estructura del código. Esta primera versión de la aplicación consta de un proyecto del framework de *Python Flask* que implementa los siguientes métodos con su correspondiente funcionalidad:

- ***get_db():*** Este primer método nos permite establecer la conexión con la base de datos Neo4j para poder manejar la información que guardamos en ella correctamente.
- ***close_db():*** Este segundo método es el opuesto del anterior. De esta manera, se encarga de cerrar la conexión con la base de datos Neo4j cuando la aplicación ya ha finalizado su ejecución. Se encuentra bajo la etiqueta *@app.teardown_appcontext.* Esta etiqueta nos indica que cuando la aplicación se detiene por cualquier circunstancia, este método se encarga de cerrar la sesión de la base de datos para evitar problemas posteriormente.
- ***favicon():*** Una de las peculiaridades del framework Flask que me asaltó durante el desarrollo es la extraña manera que tiene de asignar un favicon (icono que aparece en la parte de la pestaña del navegador) al proyecto. Tenemos que invocar al método *send_from_directory()* con la ruta en la que hemos guardado el archivo con nuestro icono. Añadido a esto, hay que implementarlo bajo la etiqueta *@app.route('/favicon.ico')*.

- ***get_details():*** Este método es exclusivo de la primera versión de la aplicación. Se encarga de ejecutar una sentencia *Cypher* que nos permite recuperar los detalles de la canción que acabamos de seleccionar, con el fin de mostrarnos los detalles de la misma en la parte derecha de la aplicación. Los detalles que recuperamos de la base de datos son:

    - El nombre de la canción.
    - El nivel de llamada en el que nos encontramos.
    - El artista que canta el tema seleccionado.
    - La popularidad de la canción en *Spotify.*
    - La duración de la canción.
    - El nombre del álbum al que pertenece dicha canción.
    - La fecha de publicación del disco o canción.
    - La imagen del disco correspondiente.

  Una vez que hemos extraído todos estos datos desde Neo4j, es el momento de representarlos en el navegador. Posteriormente indicaremos como se encarga *jQuery* de ello.

- ***get_index():*** Podríamos decir que este es el método más importante de toda la aplicación. El que se encarga en primer lugar de cargar la plantilla de la aplicación. Una vez que comenzamos a interactuar con ella, es el que se encarga de ir haciendo llamadas al API de Spotify y guardando toda la información en la base de datos.
  Entrando en detalle consta de una serie de *ifs* o condiciones que diferencian entre los distintos estados en que se puede encontrar la aplicación. El primer estado claro es que el usuario no haya introducido aún ningún dato en la aplicación. En ese caso, lo único que hace este método es cargar la platilla HTML en el navegador web.
  El segundo estado es que el usuario haya comenzado a usar nuestra herramienta, pero sólo ha introducido el nombre de la canción (dejando en blanco el nombre del artista). Llegados a este punto, la aplicación tiene que buscar en la API las canciones que poseen el título que el usuario ha indicado. Como decisión de diseño se ha establecido que nuestra aplicación seleccione la canción más popular (en el caso de que haya varias con el mismo nombre).
  En el tercer estado, el usuario ha introducido tanto el nombre de la canción como el nombre del artista y, por lo tanto, su identificación es mucho más rápida y precisa.
  Una vez tenemos los primeros datos los almacenamos en nuestra base de datos Neo4j (de momento no hace falta controlar duplicaciones ya que se trata de la primera llamada a la API), almacenamos los atributos de la canción con la que comenzamos la búsqueda para poder compararlos en un futuro con el resto de posibles canciones recomendables. Una vez realizado esto, buscamos entre los artistas relacionados con el que hemos introducido en la búsqueda. A su vez,

dentro de cada artista buscamos sus discos y, dentro de estos, las canciones que tiene registradas en la plataforma *Spotify*.
Aquí comenzamos a aplicar nuestro filtro. Por cada una de las canciones extraemos las propiedades mencionadas anteriormente como son:

- *Danceability*
- *Energy*
- *Liveness*
- *Mode*
- *Seechiness*
- *Acousticness*
- *Instrumentalness*
- *Valence*

Una vez extraemos los valores, los comparamos con los que hemos guardado anteriormente de la canción con la que estamos realizando la búsqueda. Si la suma de las diferencias de todas las propiedades es menor o igual que 0.4 seleccionamos esa canción como recomendable y la guardamos en la base de datos. Por el contrario, si la diferencia es mayor, dicha canción sería descartada.

Vamos recorriendo de esta manera todas las canciones de cada disco y todos los discos de cada artista para llegar a conformar el grafo inicial. Visualmente quedaría el proceso de la siguiente forma:

Cuando estamos hablando de una llamada cuando ya tenemos información introducida en la base de datos la situación se vuelve más compleja. El primer cambio relevante es que ya no introducimos el nombre de la canción, sino que hacemos *click* en ella. Para ello disponemos de un método en *JavaScript* que nos permite extraer el dato directamente desde el nodo (hablaremos de él más adelante).

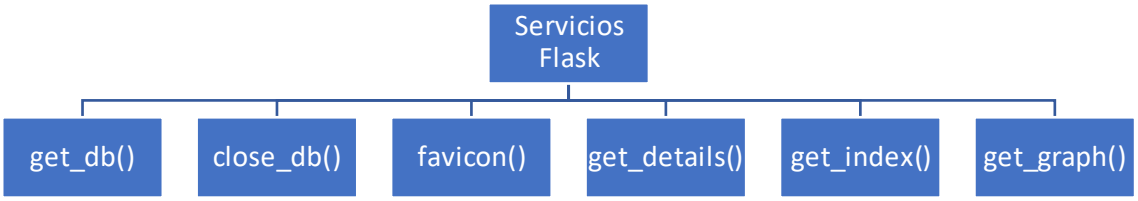
Una vez tenemos localizado el nombre y el artista de dicha canción lo primero que tenemos que hacer es evitar que sus nodos se repliquen en la base de datos ya que si has hecho *click* en ellos es porque anteriormente ya habían sido introducidos en el grafo. El siguiente paso es buscar los artistas relacionados, con sus correspondientes discos y canciones (como en el caso inicial). Sin embargo, tenemos que tener en cuenta que un artista puede estar relacionado con varios y no se debe introducir nada más que una vez. No obstante, si las canciones no estaban en el grafo sí que deben ser añadidas.

Al finalizar todas las iteraciones, debido a que puede haber canciones que se han introducido por error o por datos no verificados por la API debemos eliminarlos para evitar confusiones.

Por último, mencionar que este método se encuentra bajo la etiqueta *@app.route("/"),* lo que nos indica que es la ruta raíz de la aplicación.

- ***get_graph():*** Este método también se encuentra solamente en la primera versión de la aplicación. Una vez que el método *get_index()* ha finalizado su ejecución. El proyecto *Flask* ejecuta este método que se encuentra bajo la etiqueta *@app.route("/graph").* Este fragmento de código cuenta con que la información en la base de datos está correctamente actualizada y comienza a crear un objeto JSON que al finalizar su ejecución devolverá para que D3js se encargue desde la parte del cliente de pintar dicha información en el grafo. Mediante las variables globales *nodes* y *links*, se van almacenando todos los elementos del grafo. Según el nivel en el que se encuentre la aplicación, dicho método se encarga de extraer el artista seleccionado e ir insertando a su alrededor todos los artistas y canciones relacionados a él. De cada nodo almacenamos su id, su nombre, su *label* (que servirá para asociarle un color y un tamaño dependiendo del tipo de nodo que sea) y su *URI* de *Spotify*. Por su parte, para insertar las relaciones, establecemos su origen y si destino en función de la posición o índice que ocupan dichos nodos en el array *nodes*. Por último, hay que hacer una nueva pasada por la base de datos en el caso en el que, a ciertos artistas que ya se encontraban en la base de datos con anterioridad, se les inserten nuevas canciones que tengan un nivel más actual.

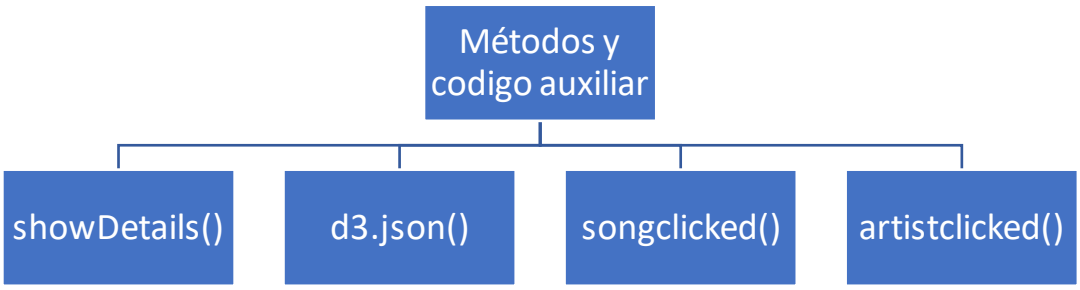
Para explicar gráficamente y de una mejor manera este proyecto *Flask*, se estructuraría de la siguiente manera:



Para recopilar toda la información que nos proporcionan los métodos del proyecto *Flask* nos ayudamos de los siguientes métodos en JavaScript (utilizando las librerías *jQuery* y *D3js*):

- ***showDetails():*** En primer lugar, como su propio nombre indica, este método se encarga de mostrarnos en la parte derecha de la interfaz web el conjunto de detalles que nos proporciona el API de Spotify acerca de la canción recientemente seleccionada. Tiene acceso al *JSON* que nos devuelve el método *getDetails()* del proyecto *Flask* de manera que pueda acceder a la canción principal del nivel actual en el que nos encontramos. Una vez que le llega la información, se encarga de reproducirla de manera gráfica en la interfaz mediante selectores de la librería *jQuery*, que nos permite manipular el DOM.
- ***d3.json("/graph", function…):*** La librería gráfica *D3js* nos proporciona un método de bastante utilidad. El mencionado método nos permite, a partir de un archivo o elemento *JSON* dado, realizar una función a modo de *callback* que se encargue de procesar la información en un grafo. Teniendo esto en cuenta, una vez que el método *getGraph()* del proyecto *Flask* nos devuelve la información, nos encargamos de pintar el grafo en pantalla. Para ello nos ayudamos de variables como *SVG, Force* o *Width/Height* (estas últimas para determinar el tamaño del grafo final).
- ***songclicked():*** Ahora entramos a comentar otro tipo de métodos. Estos se encargan de responder a una acción directa del usuario como puede ser un botón o un nodo del propio grafo. En este caso particular, el código se ejecuta cuando el usuario de la aplicación hace *click* sobre el nodo que representa una canción. Cuando detectamos dicha selección, lo primero que hacemos es marcar el nodo en rojo para que el usuario reconozca con un simple vistazo cual ha sido el efecto de su interacción con nuestra herramienta. Seguido a esto realizamos la acción más importante como es la puesta en marcha de la búsqueda en la API accediendo al nombre de la canción de manera directa gracias a *D3js*. Además, para que el usuario tenga una primera respuesta satisfactoria de su selección, abrimos directamente en el browser una nueva pestaña con la canción seleccionada siendo reproducida y con todos los detalles del disco a su alcance. Por último, también ponemos en marcha el icono de carga para que el usuario sepa que la aplicación está trabajando en segundo plano con el API.

- ***artistclicked():*** De manera análoga, para el caso en el que el usuario pueda llegar a hacer *click* en el nodo que representa a un artista realizamos las siguientes acciones. En primer lugar, marcamos como seleccionado (pintamos de color rojo) dicho nodo para que el usuario tenga constancia de su acción. Seguidamente, abrimos una nueva pestaña en el navegador web para que el usuario pueda acceder a la página del artista en la versión web de Spotify. Desde allí tendrá acceso a sus giras, su discografía y muchos más detalles de interés. Como no es el objetivo de nuestro recomendador la búsqueda por artistas directamente, este método no realiza ningún tipo de interacción con el API.
- ***Otros métodos:*** De manera adicional, también disponemos de un par de sentencias a reseñar. Para que comience la sensación de carga en la página, *jQuery* se encarga de mostrar dichos iconos de carga en el caso de que el usuario haya hecho *click* en el botón de búsqueda o aún no se haya cargado completamente la página.
- ***Maquetación (HTML+CSS):*** En este aspecto me siento particularmente satisfecho. Ya que se trata de una aplicación bastante intuitiva, la idea era que no mostrase una interfaz muy recargada y liosa de utilizar. Además, era de gran interés hacerla ampliable por si la aplicación experimenta mejoras en un futuro. Se ha logrado de manera sencilla y eficaz, mezclando la temática de Neo4j y Spotify (los colores del gráfico son los característicos). Básicamente consta de tres componentes principales. En la parte superior se encuentra la barra de búsqueda mediante la que podemos introducir nuestra canción y artista iniciales. En la zona principal de la aplicación distinguimos entre la zona izquierda y derecha. En la zona izquierda tenemos la parte principal de nuestra herramienta como es el grafo, mientras que a mano derecha encontramos los detalles de la última canción seleccionada.



E) SEGUNDA VERSIÓN DE LA APLICACIÓN

Como ya he mencionado con anterioridad, una vez finalizado el desarrollo de toda la funcionalidad de la aplicación, se percibía que la velocidad del API es limitada. Añadido a esto, ya que la cantidad de datos que manejamos es bastante grande, se hacía una experiencia de usuario bastante tediosa.

Para solucionar esto, además de realizar la acción de abrir las pestañas de *Spotify Web* para que el usuario realizase más acciones durante ese tiempo, se ha tomado la decisión de ir actualizando tanto la información del grafo como la de los detalles de la canción desde la conexión de *JavaScript* con la base de datos de nuestra propia aplicación, como es *Neo4j*.

Esto conlleva un importante cambio en el código ya que parte de los servicios *Flask* desaparecen para dejar paso a un aumento de código en la parte de *JavaScript*. Voy a proceder ahora a mencionar los cambios en los servicios *Flask*.

Entre los métodos que continúan en nuestra aplicación se encuentran por supuesto los que conllevan una interacción con la base de datos como son *get_db()* y *close_db().* Como nuestra aplicación continúa siendo un proyecto *Flask*, también conservamos el método que nos proporciona el *Favicon*, como es *favicon().*

De la misma manera, como la funcionalidad y el algoritmo es el mismo, el método que obtiene los datos del API y los almacena en la base de datos Neo4j se mantiene intacto. Se trata del método *get_index()* situado bajo la etiqueta *@app.route("/").*

Una vez mencionados los anteriores métodos que quedan sin cambios, procedemos a comentar los cambios producidos. Como se sobreentiende de los anteriores párrafos, el resto de los servicios *Flask* se eliminan del proyecto. Los más significativos eran *get_graph()* y *get_details().* En esta segunda versión de la aplicación, la funcionalidad de estos métodos es aportada por las nuevas funciones JavaScript en el lado del cliente. La supresión de ambos métodos se debe principalmente a temas de eficiencia. Ya que el API de *Spotify* cuenta con una velocidad limitada, no podíamos dejar de lado la experiencia del usuario al utilizar nuestra aplicación. No obstante, hay que destacar que, aunque hayan sido sustituidos, realizaban perfectamente su función.

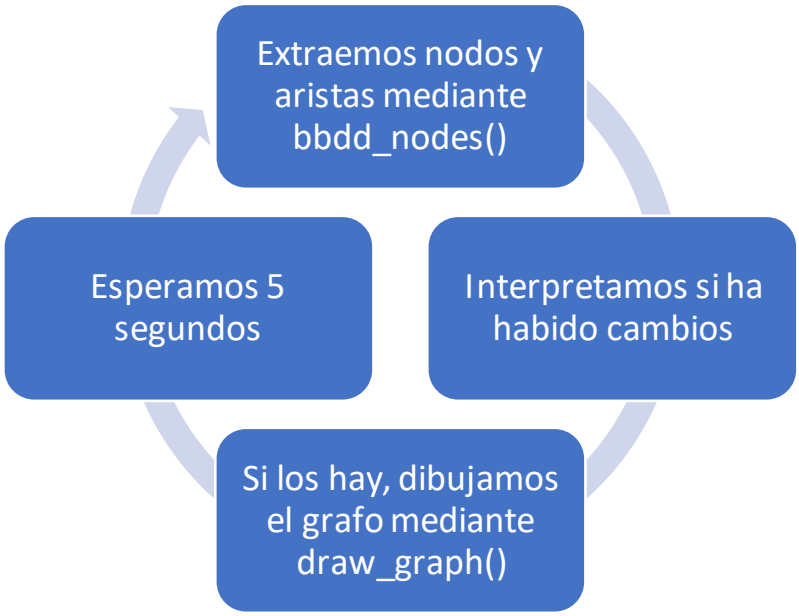
Ya que el método *get_details()* sigue sirviéndonos nuestra aplicación *Flask* y realiza toda la interacción con el API, vamos a comentar y analizar la manera actual de procesar dichos datos.

Las funciones de Spotify que realizan las funciones sustitutivas son las siguientes:

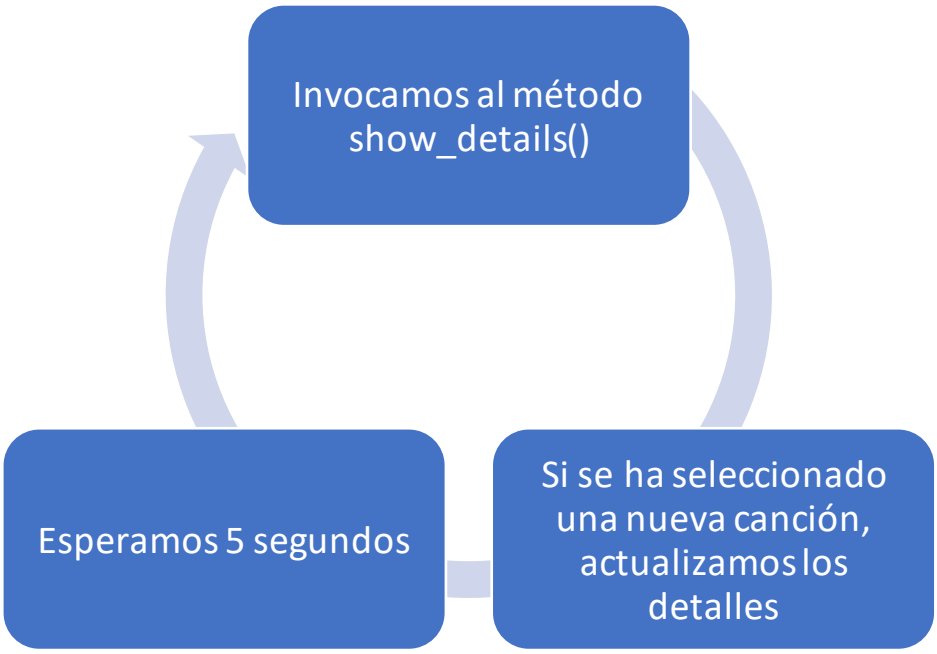
- ***connect():*** En la anterior versión, la aplicación *Flask* era la única parte del código que interactuaba con la base de datos directamente. Ya que mientras la API nos proporciona datos, nosotros tenemos que seguir procesándolos al mismo tiempo, es necesario establecer la conexión con *Neo4j* desde el método *connect().*

- ***bbdd_nodes(session):*** Una vez se ha establecido con éxito la conexión con la base de datos, procedemos a traernos la información que necesitamos. Por un lado, necesitaremos los nodos que se encuentran actualmente procesados, y por otro las relaciones que se han establecido entre ellos hasta el momento. Dicha información es solicitada periódicamente. Es decir, cada 5 segundos desde el inicio de la aplicación, este método es llamado. Si tiene alguna novedad, ya sea relacionado con los nodos o con las relaciones, el grafo será actualizado en consecuencia. Si no hay novedades, la aplicación permanecerá intacta hasta la próxima llamada, y así no entorpecerá la experiencia de usuario.
- ***draw_graph(graph):*** El tercer paso es evidente. Si ya hemos establecido la conexión con la base de datos y hemos rescatado la información directamente desde Neo4j, la próxima acción será pintar adecuadamente el grafo en nuestra aplicación para que el usuario pueda ir apreciando el proceso poco a poco. Para ello, al igual que en la anterior versión de la aplicación, nos ayudamos de *SVG, FORCE* y *WIDTH/HIGH* que nos permite establecer las dimensiones en las que se mueve nuestro grado. Este método es llamado cada vez que nuestro método *bbdd_nodes(session)* detecta cambios en la base de datos. Una de las dificultades que tuve al re-implementar este método fue la siguiente. En la versión anterior de la aplicación, esta parte del código sólo era llamada cuando la página se recargaba por acciones de la herramienta. Ahora tenía que ser llamado un mayor número de veces para crear la sensación de un progreso continuado. Al ocurrir esto, se producían comportamientos anómalos. Por ejemplo, ciertos nodos se quedaban inmóviles u otros se superponían encima de otros. Esto era debido a que el método creaba un componente SVG en el HTML de la página cada vez que el método era invocado y de esta manera se iban superponiendo. Mediante la sentencia *d3.select("svg").remove();* antes de realizar una nueva llamada, el problema fue solucionado de manera simple.
- ***showDetails():*** Este método pertenecía bajo la misma nomenclatura a la implementación anterior pero esta vez su contenido varía de manera sustancial. Anteriormente, utilizábamos *jQuery* para interpretar la información proporcionada por la ruta */details* que se encontraba en nuestro proyecto *Flask*. Ahora tenemos que interpretar la información directamente con llamadas a la base de datos. De la misma manera, este código se ejecuta cada 5 segundos desde el inicio de la aplicación para que el usuario tenga que esperar el menor tiempo posible. Entrando en detalles de implementación más concretos, comenzamos realizando una *query* en *Cypher* que nos trae todas las canciones que hemos ido seleccionando hasta el momento. Como el objetivo es quedarnos con la más actual, debemos recorrer dicha colección y nos quedamos con la canción que estamos buscando. Una vez que tenemos la canción deseada, extraemos sus detalles concretos para representarlos en nuestra herramienta.

Para que sea más visible la mejora en la aplicación voy a mostrar unos gráficos en forma de ciclo que nos indiquen el funcionamiento actual de la aplicación. Estos ciclos se realizan de forma simultánea mientras que la aplicación se encuentra en funcionamiento. Teniendo en cuenta el ciclo de actualización del grafo tendríamos el siguiente grafo (después de producirse la conexión con la base de datos mediante el método *connection()*):



Paralelamente a este ciclo tenemos el de la actualización de los datos de la actual canción seleccionada. Se iría actualizando de la siguiente forma:



5. RESULTADOS

Una vez hemos explicado y analizado el cómo, el qué y el por qué, es momento de analizar lo que en realidad hemos estado persiguiendo durante todo el desarrollo de la aplicación, los resultados. Lo primero que hay que tener en cuenta es la cantidad ingente de datos que nos aporta el API de Spotify y por lo tanto la dificultad reseñable para procesarlos. Por ello he implementado la segunda versión de cara a intentar representar los resultados de una manera más clara, concisa y atractiva.

Aun así, como todas las aplicaciones tiene limitaciones y las vamos a ir indicando cuando sea el momento oportuno. Como es lógico, a medida que el usuario pasa más tiempo interactuando con nuestra aplicación, aumenta el tamaño del grafo y es más difícil representar los datos de manera ordenada en la página. Pese a esto, se ha solucionado de una buena manera creando una interfaz web completamente dedicada a la muestra de datos sin elementos sobrantes que harían que la aplicación fuese muy recargada. Para visualizar mejor todos y cada uno de los casos que se pueden dar, vamos a mostrar más detalles en el siguiente apartado.

A) CASOS DE EJEMPLO

La mejor manera de mostrar cómo funciona nuestra aplicación es mostrando lo que podríamos llamar el camino que seguiría un usuario de nuestra aplicación a medida que avanza por ella. Cabe mencionar llegados a este punto que los resultados dependen totalmente de la canción seleccionada en cada caso. Si seleccionamos una canción o artista muy popular y con muchos artistas relacionados, nuestro grafo crecerá de una manera mucho más rápida que si se da el caso opuesto. Sea como sea el camino seguido por el usuario vamos a comenzar con el ejemplo a continuación.

I. GRAFO PEQUEÑO

Una vez el usuario entra en nuestra aplicación, lo primero que tiene que hacer es introducir los datos de la primera canción que quiere buscar mediante el buscador de la parte superior. Una vez hecho esto, nuestra aplicación se pone a trabajar con el API de Spotify. Por lo general, una vez realizamos esta búsqueda, nuestra base de datos suele acabar con entre 4 y 20 nodos dependiendo de la popularidad de la canción escogida.

Es la única vez que utilizamos los cuadros de texto de la parte superior ya que, si continuamos ampliando nuestro grafo en más niveles, deberemos interactuar directamente con los nodos del grafo.

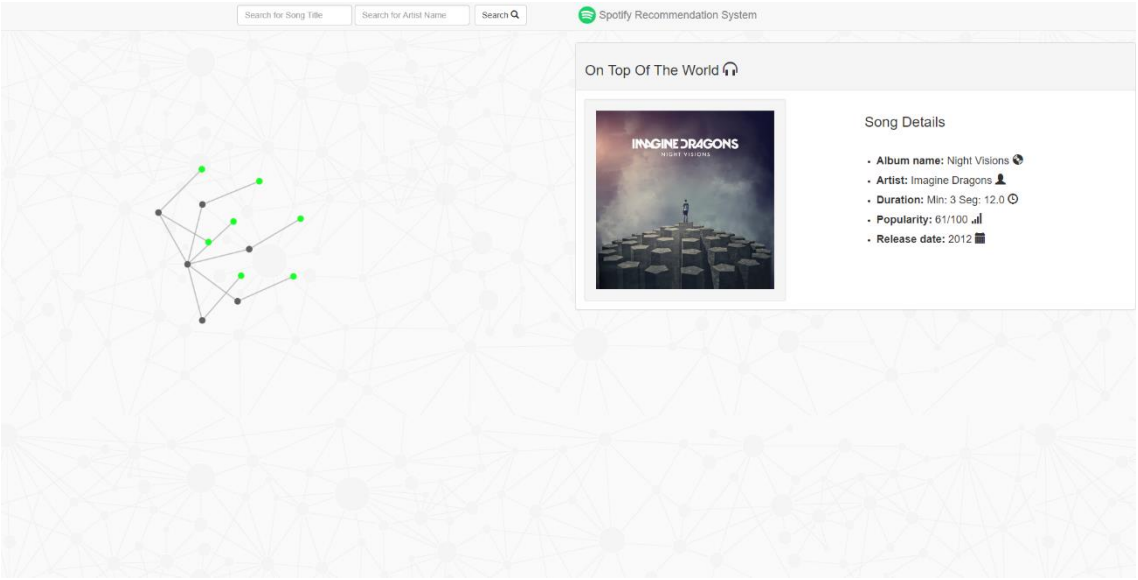


Ilustración 18. Captura de pantalla de un caso con un grafo pequeño.

En esta captura de pantalla de nuestra aplicación se muestra el caso en el que el usuario haya introducido la canción *On Top Of The World* de *Imagine Dragons*. En este caso contaríamos con un total de 12 nodos en nuestra aplicación.

II. GRAFO MEDIANO

En este apartado nos vamos a centrar en el caso en el que el usuario haya realizado entre 2 y 3 llamadas al API de Spotify. A esa altura es más complicado generar una información precisa acerca de los nodos que pueden formar parte del grafo en ese momento. No obstante, debemos contar con entre 20 y 70 nodos más o menos. Llegados a este punto, además de haber observado las periódicas actualizaciones de nuestro grafo también hemos debido observar ya varias veces la actualización en la parte de los detalles de la canción. Es muy interesante y habitual el caso en el que selecciones alguna canción que jamás hayas escuchado y te sorprenda gratamente.

Siguiendo el ejemplo que hemos mostrado en el grafo anterior, hemos decidido seleccionar la canción *Feather* del grupo *X Ambassadors.* Si en la situación anterior contábamos con un total de 12 nodos, en el caso actual contamos con un total de 46. De esta manera hemos ampliado en un total de 34 nodos. Como hemos mencionado anteriormente, a medida que aumentamos el grafo la probabilidad de encontrar canciones similares es mayor y, de la misma manera, las canciones son menos parecidas a la canción de origen, difuminándose así las propiedades de esta.

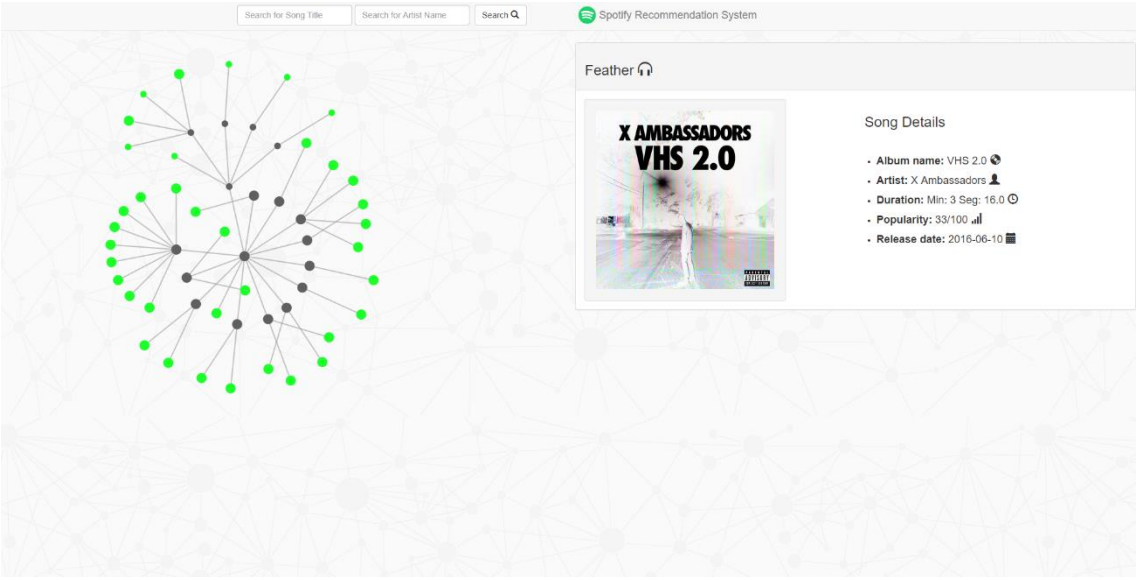
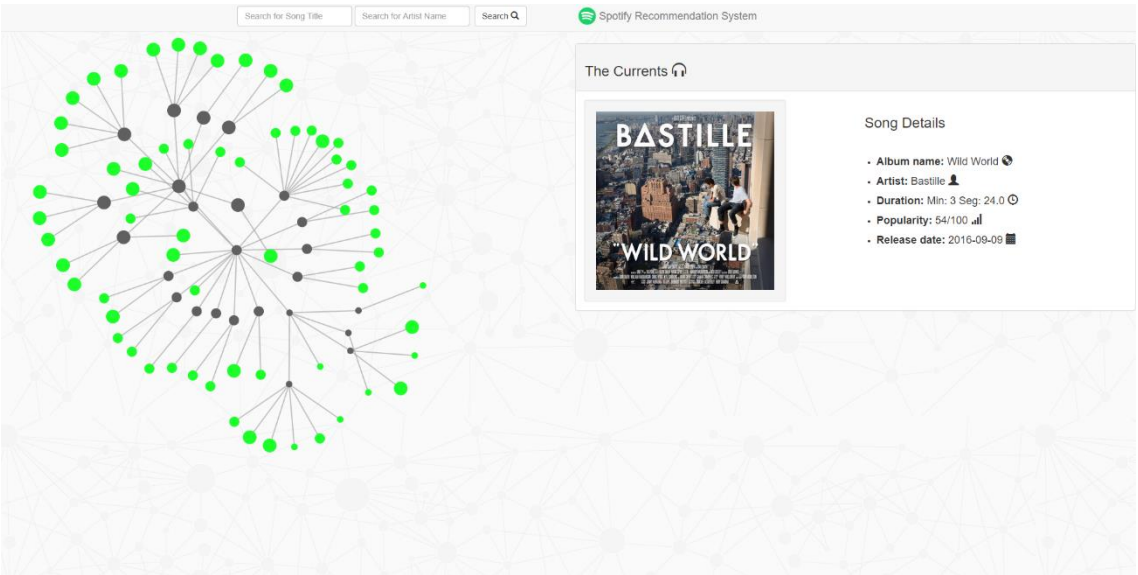


Ilustración 19. Captura de pantalla de un caso con un grafo mediano.

III. GRAFO GRANDE

Para los expertos y usuarios que hayan sido impresionados por nuestra herramienta llegamos a este punto. Sin duda alguna ya habremos interiorizado el fácil proceder de nuestra aplicación y habremos descubierto alguna canción nueva que nos haya gustado. En este punto habremos realizado ya entre 4 y 5 llamadas y nuestro grafo contará con entre 40 y 120 nodos con total facilidad.



Para finalizar esta ruta por nuestra aplicación, vamos a terminar con un grafo de tamaño bastante considerable. Esta vez realizamos *click* sobre la canción *The Currents*

del grupo *Bastille*. Como podemos observar a simple vista, el grafo ya es bastante grande ya que llega a los 91 nodos, desde los 46 del último nivel.

Como mencionaba en niveles anteriores, también se puede apreciar el aumento del tamaño de los nodos de los niveles actuales, permitiendo así diferenciar de manera sencilla qué nodos han sido añadidos recientemente y cuales llevan más tiempo en nuestro ciclo de la aplicación.

Dicho esto, hay que tener en cuenta que nuestra aplicación puede seguir funcionando con muchos más nodos. Sin embargo, y como es lógico, cuando sobrepasamos el nivel 5 es mucho más difícil procesar la información, aunque sí que sigue realizando las funciones que le exigimos, de lo cual me siento muy orgulloso.

B) TABLA RESUMEN (CON TIEMPOS)

Llegamos ya a uno de los apartados clave del trabajo. Debido a la gran cantidad de datos que estamos manejando en nuestra herramienta, tenemos que tener en cuenta que, de cara a una buena de usuario, tenemos que manejarlos de la mejor manera posible. Este es el principal motivo por el cual se han ido realizando evoluciones en la aplicación para que esta sea lo más atractiva posible. Para entrar más en detalle vamos a realizar una serie de pruebas comparando los tiempos que tarda nuestra aplicación y cada cuanto se van realizando actualizaciones en nuestra interfaz web.

En primer lugar, vamos a reproducir el anterior camino que hemos mostrado en la sección de muestra de ejemplos de grafos concretos. Es decir, primero vamos a realizar la búsqueda de la canción *On Top Of The World* de *Imagine Dragons*, posteriormente vamos a hacer *click* sobre la canción *Feather* de *X Ambassadors* y para finalizar seleccionaremos la canción *The Currents* del grupo *Bastille*.

A continuación, vamos a mostrar una tabla resumen de todos los tiempos que transcurren en la aplicación y cada cuanto se actualiza nuestro grafo con nuevos nodos.

Tabla 1. Tabla de tiempos del primer ejemplo.

Nivel del grafo	Canción	Artista	Tiempo empleado	Nodos finales
Nivel 1	On Top Of The World	Imagine Dragons	2 min 2 segundos	12
Nivel 2	Feather	X Ambassadors	3 min 54 segundos	46
Nivel 3	The Currents	Bastille	4 min 15 segundos	91

Para aumentar un poco más la perspectiva de la aplicación, no nos limitamos a este primer ejemplo y realizamos otro de manera que podamos analizar mejor el comportamiento de nuestra aplicación. En el segundo ejemplo comenzaremos realizando la búsqueda de la canción *El secreto de las tortugas* de *Maldita Nerea*. Seguidamente haremos *click* en *Kamikaze* de *Lagarto Amarillo* y finalizaremos con la selección de *Carita de buena* de *Efecto Pasillo*. A continuación, mostramos detalladamente los datos de este segundo ejemplo:

Tabla 2. Tabla de tiempos del segundo ejemplo.

Nivel del grafo	Canción	Artista	Tiempo empleado	Nodos finales
Nivel 1	El Secreto de las Tortugas	Maldita Nerea	3 min 1 segundo	32
Nivel 2	Kamikaze	Lagarto Amarillo	4 min 6 segundos	65
Nivel 3	Carita de buena	Efecto Pasillo	3 min 5 segundos	84

Una vez mostrados los detalles de ambos ejemplos vamos a proceder a interpretarlos. En primer lugar, hay que mencionar que dichos datos se han obtenido con la intención de ser lo más exactos posibles y de esta manera hemos comparado con todas las canciones de todos los artistas disponibles en el API de Spotify (se podrían aplicar ciertas restricciones como ya comentaremos en la sección siguiente de conclusiones y trabajo futuro).

Dicho esto, como punto negativo tenemos que señalar el tiempo de respuesta del API de Spotify ya que es bastante lento cuando tratamos de trabajar con un volumen de datos considerable, ya que he realizado ejemplos simples y en ese caso sí que funciona razonablemente bien.

Por otro lado, podemos apreciar un dato que da bastante valor a nuestra implementación de la solución y es la siguiente. Por ejemplo, en el caso del primer ejemplo, en el primer nivel tenemos un total de 12 nodos procesados en 122 segundos por lo que nuestra aplicación procesa un nodo cada 10 segundo aproximadamente. Si avanzamos al segundo nivel, obtenemos un total de 46 nodos (34 más que en el nivel anterior) y los procesamos en un total de 234 segundos lo que sale a aproximadamente 8 segundos cada nodo introducido al grafo. Por último, tenemos un total de 91 nodos (45 nodos más que en el segundo nivel) procesados en 225 segundos que sale a una media de 6 segundos el nodo.

Esto nos indica que a medida que trabajamos con más datos, el algoritmo es igual e incluso más eficiente que cuando comienza (ya que posee cierta información ya en la base de datos). Se podría esperar que a medida que el algoritmo avanzase, le costase más procesar la información debido a la cantidad tan grande de datos con la que está trabajando, pero no es así.

Hay que recordar que estos son los tiempos totales de cada nivel. Es decir, desde que el usuario realiza *click* o introduce la canción en el buscador. Como ya hemos comentado anteriormente, nuestra segunda versión de la aplicación va realizando actualizaciones periódicas de la interfaz de usuario para mejorar la experiencia de nuestra herramienta. De esta forma, a los 5 segundos de seleccionar una canción, ya contamos con los detalles de esta en el componente destinado a ello. Por otro lado, cada 5 segundos se va comprobando si hay nuevos nodos para introducir en el grafo (y aproximadamente cada 10 segundos se actualiza con nueva información).

6. CONCLUSIONES Y TRABAJO FUTURO

Como conclusión a esta memoria descriptiva del trabajo de fin de grado que he realizado me gustaría reseñar un gran número de cosas.

Lo primero me gustaría dar valor a la idea de la aplicación, ya que a medida que he ido buscando información en internet me he dado cuenta de que es muy poco común este tipo de aplicaciones que manejan gran cantidad de datos en una interfaz de usuario y en una base de datos.

En cuanto a la experiencia con el API de Spotify me gustaría reseñar que ha sido positiva. A pesar de ser un tanto lenta para ciertos algoritmos concretos, ha sido bastante fácil e intuitivo el aprendizaje de los métodos y estructuras de datos que maneja la misma.

Relativo al tiempo que emplea el API en devolvernos la información, como he comentado en el apartado anterior se podrían hacer unos ajustes. Bien es cierto que he intentado realizar una aplicación que represente fielmente la realidad en ciertos casos puede ser demasiada complicación. Aunque se puede considerar reducir la cantidad de muestra a buscar, esto nos va a producir una pérdida de información y solo lo haremos en el caso en el que esta sea tolerable. Respecto a esto, es cierto que podríamos reducir la búsqueda de canciones parecidas de alguna de las siguientes maneras:

- Podríamos evitar buscar entre tantos artistas relacionados de cada uno de ellos ya que a medida que aparezcan en el grafo van a ir apareciendo.
- De la misma manera, si un artista lleva muchos años en activo, se podría llegar a reducir la cantidad de discos entre los que vamos a buscar canciones.
- Dentro de los discos hay muchas canciones en la mayoría de los casos poco populares y se podrían obviar desde ese punto de vista.

De esta manera la aplicación tardaría bastante menos en ejecutar y obtendríamos resultados satisfactorios. Pese a ello, no aprovecharíamos de la mejor manera la buena fuente de datos que nos proporciona el API de Spotify.

Con respecto lenguaje de programación sólo puedo decir que me ha encantado. Uno de los objetivos que tenía en mente cuando comencé a pensar en posibles trabajos de fin de grado era poder aprender y utilizar ciertas tecnologías que no había manejado mucho durante el transcurso de la carrera. De esta manera y con este pensamiento en mente comencé a aprender *Python* (del que ya tenía muy buenas experiencias de proyectos que había visto por internet anteriormente). Su sintaxis es muy intuitiva y fácil de aprender. Además, al ser un lenguaje de muy alto nivel, podemos hacer algoritmos de mucha complejidad en muy pocas líneas, lo que por otra parte aumenta su legibilidad. Gran parte de los proyectos que van a revolucionar el mundo de la programación y la tecnología en general están escritos en *Python*.

Más concretamente, si hay algo que ayuda a tu buena experiencia con un lenguaje de programación es el entorno con el que trabajas. *PyCharm* en este sentido sólo te da ventajas. Desde consejos a la hora de la sintaxis del lenguaje, hasta su propio terminal integrado en la aplicación o la posibilidad de manejar las variables de entorno en sólo unos *clicks*.

En cuanto al *framework* de *Python* que he utilizado (*Flask*) podríamos decir que le viene genial a la aplicación. No hemos necesitado muchos servicios ni tenemos muchas rutas diferentes, por lo que es lógica la elección de este *framework*. Sin embargo, si en un futuro deseamos ampliar la aplicación de manera considerable, deberíamos tener en cuenta la sustitución del framework por otro como puede ser *Django*.

Comentando un poco la parte de la interfaz de usuario, *D3js* ha sido de gran ayuda y la mejor opción que podíamos haber escogido. Es cierto que en un principio tuvimos en cuenta otras librerías como puede ser *Bokeh* pero decantarse por *D3js* nos favoreció en el desarrollo de la aplicación. A la hora de elegir entre una librería u otra que nos ofrecen una funcionalidad similar siempre debemos quedarnos con la que más documentación y comunidad tenga sobre todo de cara a la posibilidad de que nos surjan ciertas dudas de más complejidad. Es cierto que requiere una gran curva de aprendizaje en un principio. Sin embargo, una vez que te acostumbras a su sintaxis y métodos básicos puedes realizar verdaderas obras de arte.

Otra conclusión importante acerca del proyecto tiene que ver con las dos versiones que he implementado de la aplicación. En el primer caso, teníamos una estructura más clásica de lo que sería una aplicación, pero tenía una pobre experiencia de usuario. En cuanto a la segunda versión, tenemos una experiencia de usuario bastante mejorada, pero se aprecia una estructura un tanto más novedosa y alternativa.

Para combinar ambas se podría implementar una mejora en la estructura. He estado investigando acerca de cómo poder hacer que, directamente desde *Flask*, podamos actualizar el grafo y la zona de los detalles de la canción de manera dinámica.

La principal solución que he podido encontrar sería la siguiente:

```python
import threading
import time
from flask import Flask
app = Flask(__name__)

@app.before_first_request
def activate_job():
    def run_job():
        while True:
            print("Run recurring task")
            time.sleep(3)

    thread = threading.Thread(target=run_job)
    thread.start()

@app.route("/")
def hello():
    return "Hello World!"


if __name__ == "__main__":
    app.run()
```

Ilustración 20. Captura de un ejemplo de muestra acerca del uso de threads en Flask.

Comentando este ejemplo que he encontrado investigando en internet, podríamos decir que habría que crear varios hilos alternativos al principal. El hilo principal se encargaría de la recolección de la información del API y su introducción en nuestra base de datos. Uno de los hilos se estaría encargando de recoger la información de la base de datos, y el último se encargaría de ir actualizando la información de los detalles de la canción seleccionada.

Como es entendible, el anterior ejemplo es muy simple y funciona sin problemas. Sin embargo, al tener una aplicación más compleja como la nuestra, tenemos que tener en cuenta que cuando iniciamos el *thread* principal y se encuentra trabajando con el guardado de la información en la base de datos, ninguno de los otros *threads* acceden a la información.

La clave sobre este asunto debe estar en aplicar ciertos mecanismos de sincronización, de manera que cuando un *thread* solicite entrar a manejar cierta información, pueda hacerlo.

Esta mejora nos ayudaría bastante, ya que la experiencia de usuario sería muy similar a la que obtenemos con nuestra segunda versión de la aplicación, pero manteniendo una estructura mucho más organizada.

A continuación, y para finalizar este apartado de posibles mejoras de la aplicación en un futuro vamos a comentar las más principales que se me pueden llegar a ocurrir. No obstante, hay que tener en cuenta que como cualquier otra aplicación tiene infinidad de posibles aspectos a mejorar y se podrían incluir cualquiera que se le ocurra al desarrollador (gracias también en parte a que hemos hecho la aplicación con una estructura modular).

En primer lugar, una de las funcionalidades que se pueden implementar para mejorar la experiencia de usuario sería la posibilidad de escuchar la canción en nuestra propia herramienta o al menos una parte de ella. Es cierto que esta funcionalidad se intentó implementar, pero acarrea varios problemas. El primero de ellos y más importante es que, realizando una búsqueda por varias canciones en el *API* nos hemos dado cuenta de que no todas las canciones tienen entre sus atributos una escucha, lo que puede llevarnos a una inconsistencia de la aplicación y a la confusión del usuario final. Otro de los problemas, por ejemplo, sería que se solaparía con la función ya implementada de abrir una nueva pestaña en el navegador con la canción seleccionada ya reproduciéndose desde la versión web de *Spotify*.

Otra funcionalidad de gran utilidad sería la inclusión de ciertas listas de canciones que más has escogido en tu uso de la aplicación o ciertas estadísticas como puede ser tu grupo favorito, tu estilo de música más buscado o tu canción favorita. Para ello habría que realizar una especie de inicio de sesión para identificar al usuario que está utilizando la aplicación y poder mostrarle sus propios datos personalizados.

Ya hemos comentado en apartados anteriores que en cuanto una de las canciones que estamos buscando se diferencia en menos de 0.4 entre la suma todos sus atributos con respecto a la canción que tenemos seleccionada, la canción se considera parecida. Sin embargo, si la diferencia es mayor de 0.4 no se considera parecida y se descarta. Pues bien, como posible mejora a la aplicación se podría dejar al usuario que utilice nuestra herramienta la posibilidad de escoger como de parecida quiere que sean las canciones que nuestra aplicación va a buscar. Eso simplemente se podría hacer añadiendo más opciones a la hora de realizar la primera búsqueda en la barra superior de nuestra aplicación. Por ejemplo, se puede dejar tres opciones entre: canciones muy similares, similares o poco similares. Sin embargo, esto puede llevar a errores, ya que si elegimos valores muy bajos se puede dar el caso de que no encontremos ninguna canción parecida. Por el contrario, si el número elegido es demasiado alto, nos puede llevar a una masificación de información en nuestro grafo. Teniendo en cuenta esto hay que realizar bastantes pruebas para optimizar la aplicación.

Además de la posibilidad de añadir el filtro comentado anteriormente se pueden implementar muchas más búsquedas personalizadas. Por ejemplo, la posibilidad de elegir el idioma en que estén cantadas las canciones que vamos a introducir en el grafo o el estilo de música al que queremos restringir nuestro algoritmo.

De cara a una representación aún mayor de datos en nuestra herramienta para que todo esté realmente conectado se pueden poner más datos de cada uno de los artistas que poseemos en nuestra base de datos. Por ejemplo, se podrían mostrar las redes sociales de cada uno de ellos como pueden ser Facebook, Twitter, Instagram o muchas otras más. No obstante, con el tema de la GDPR tenemos que tener cuidado con los datos que damos a conocer de cada uno de los artistas en nuestra aplicación. Puede que para datos concretos necesitemos la autorización expresa del artista o de alguna organización.

Con respecto al tema de las redes sociales, además de encontrarnos como en nuestro caso con el *API* gratuito de Spotify, podemos tener acceso al *API* de alguna red social como puede ser la de Twitter. Una buena evolución de nuestra aplicación sería integrar los tweets con impresiones sobre nuestro artista en tiempo real o de su cuenta propiamente dicha. Un problema que le veo yo a esta ampliación podría ser que necesitemos una relación 1:1, es decir, que todos los artistas de *Spotify* tengan Twitter y estén activos. Más allá de limitaciones como esta podríamos realizar una combinación de ambas *APIs* que sea muy beneficiosa para nuestro usuario final de la aplicación.

Más relacionado con el algoritmo en sí, se me ocurre la posibilidad de incorporar una nueva clase en nuestro grafo que fuesen los discos de un determinado artista. El beneficio de esta modificación sería la mejora en la estructuración de la información, pero puede llevar a una masificación de nodos y habría que estudiar la manera perfecta de implementar todo esto.

Para finalizar, me gustaría comentar una curiosa alternativa que tuve al principio del trabajo compartida con mi tutor.

La idea implementada es la siguiente. El usuario realiza una búsqueda de una canción en el API de Spotify y espera que el sistema busque entre todas las canciones de los artistas relacionados con el inicial y le muestre todas las canciones parecidas a la dada. A medida que avanza por la aplicación va seleccionando nodos en el grafo que permiten ir ampliando la búsqueda a lo largo y ancho del API.

Pues bien, esta era una de las ideas. La otra es algo diferente pero también igual de interesante. Se trata básicamente de buscar un camino entre dos canciones. Trataré de explicarlo con la mayor simplicidad posible. Inicialmente deberemos seleccionar dos canciones que a priori son totalmente distintas.

Una vez tenemos estas dos canciones, debemos ir avanzando por el API de Spotify recorriendo canciones cada vez menos parecidas a una de ellas y más parecidas a la otra para acabar dando con la canción del extremo opuesto. En primer lugar, hay que decir que es bastante difícil de averiguar dicho camino porque para nosotros desde fuera, el API de Spotify se comporta como una caja negra, ya que no sabemos con total certeza lo que contiene.

Otro de los problemas principales serían la imposibilidad de saber cómo me estoy acercando a la otra canción y no me estoy desviando por cualquiera de los infinitos caminos alternativos. En este sentido, si podemos saber que las canciones se van pareciendo menos, pero es muy complicado saber en qué medida debemos de dejar de parecernos a una canción y comenzar a parecernos a la del extremo opuesto.

Como ampliación de la aplicación se podría crear una nueva pestaña e implementar esta nueva funcionalidad que sería bastante curiosa y atractiva para el usuario que actúe con nuestra herramienta.

7. BIBLIOGRAFÍA

- ***Referencias de las herramientas y lenguajes de programación utilizados.***

  Lenguaje de programación Python
  https://www.python.org/

  Framework de Python Flask
  http://flask.pocoo.org/

  Lenguaje de programación JavaScript
  https://www.javascript.com/

  Librería de JavaScript jQuery
  https://jquery.com/

  Librería de JavaScript D3js
  https://d3js.org/

  Entorno de programación para Python PyCharm
  https://www.jetbrains.com/pycharm/

  API de Spotify
  https://beta.developer.spotify.com/documentation/web-api/

  Documentación acerca del API de Spotify
  http://spotipy.readthedocs.io/en/latest/

  Aplicación de gestión de tareas
  https://trello.com/

  Sistema de control de versiones
  https://github.com/

  Base de datos Neo4j
  https://neo4j.com

  Lenguaje de marcado HTML
  https://www.w3schools.com/html/

  Lenguaje de estilo CSS
  https://www.w3schools.com/css/

- ***Referencias a los proyectos mencionados en el apartado de estado del arte***

  Combining Spotify and Twitter Data for Generating a Recent and Public Dataset for Music Recommendation. (Martin Pichl, Eva Zangerle, Günther Specht).
  https://dbis.uibk.ac.at/sites/default/files/2017-03/gvdb14.pdf

  Deep content-based music recommendation. (Aaron van den Oord, Sander Dieleman, Benjamin Schrauwen).
  http://papers.nips.cc/paper/5004-deep-content-based-music-recommendation.pdf

  A Survey of Music Recommendation Aids. (Pirkka Åman, Lassi A. Liikkanen).
  https://pdfs.semanticscholar.org/d904/0ff891c0afe9cb275419bcf4a6d6f12c5fc0.pdf

- ***Referencias a la información del apartado de conclusiones y trabajos futuros***

  Página oficial de Twitter

  https://twitter.com/

  Página oficial de Facebook

  https://es-es.facebook.com/

  Página oficial de Instagram

  https://www.instagram.com/?hl=es

  Página del API de Twitter

  https://developer.twitter.com/en/docs.html

  Página del ejemplo de trabajo con Threads en Flask

  https://networklore.com/start-task-with-flask/