Documentación escrita

Evaluación: Proyecto II
Curso: Bases de datos II
Profesor: Gerardo Nereo Campos Araya
Estudiantes:
Kendall Fabián Guzmán Ramírez - 2019076561
Kenneth Palacios Molina - 2020035407
Iván Solís Ávila - 2018209698
Jose Pablo Quesada R. - 2020211670
Carlo Eduardo Leiva Medaglia - 2021032973

Explicación Loader

A continuación se proporciona una explicación paso a paso del código:

1. Importación de bibliotecas: Se importan las bibliotecas necesarias para el funcionamiento del código, incluyendo `os`, `BlobServiceClient` y `DefaultAzureCredential` de Azure Blob Storage, `pandas`, `MongoClient` y `ConfigurationError`, `ServerSelectionTimeoutError` de MongoDB, e `io`.
2. Configuración de credenciales: Se establece el nombre del contenedor de Blob Storage en la variable `blob_container_name`, el nombre del archivo de letras en `lyrics_file_name` y el nombre del archivo de artistas en `artists_file_name`. Estas variables se utilizarán más adelante en el código.
3. Definición de la función `merge_data_frames`: Esta función toma dos dataframes como entrada y realiza una combinación interna basada en la columna "ALink". Renombra la columna "Link" del segundo dataframe a "ALink" y luego realiza la combinación. El resultado es el dataframe combinado.
4. Bloque `try-except-finally`: Este bloque se utiliza para controlar los posibles errores y realizar operaciones específicas en caso de éxito o fallo.

a. Carga de los datos de Blob Storage: Se crea una instancia del cliente de Blob Storage utilizando la cadena de conexión almacenada en la variable de entorno `ACCOUNT_URL`. Luego, se obtiene el cliente del contenedor y se descarga el archivo CSV de letras mediante `blob_client`. El archivo se guarda localmente con el mismo nombre. A continuación, se muestra un mensaje indicando que el archivo ha sido descargado. Se repite el mismo proceso para el archivo de artistas.

b. Lectura de los datos en dataframes: Se utilizan las funciones `pd.read_csv` para leer los archivos CSV descargados y almacenar los datos en los dataframes `data_frame_Lyrics` y `data_frame_Artists`.

c. Manipulación de datos: La columna "Genres" del dataframe `data_frame_Artists` se procesa mediante la función `apply` y se divide en una lista de géneros separados por "; " si la columna es una cadena. Esto se hace para normalizar los datos antes de combinarlos con el dataframe `data_frame_Lyrics`.

d. Combinación de dataframes: Se llama a la función `merge_data_frames` pasando `data_frame_Lyrics` y `data_frame_Artists` como argumentos. El resultado se almacena en el dataframe `final_data_frame`, que contiene los datos combinados de letras y artistas.

e. Procesamiento por chunks y carga en MongoDB Atlas: El dataframe `final_data_frame` se divide en chunks de tamaño `chunk_size` utilizando el método `groupby` y se itera sobre cada chunk. Dentro del bucle,

se crea una instancia del cliente de MongoDB Atlas utilizando la cadena de conexión almacenada en la variable de entorno `MONGO_CONNECTION_STRING`. Luego, se accede a la base de datos y a la colección especificada. El chunk se convierte en una lista de diccionarios mediante el método `to_dict('records')` y se inserta en la colección de MongoDB Atlas utilizando `insert_many`. Una vez completada la inserción, se cierra la conexión con MongoDB Atlas. Se imprime un mensaje indicando el número del chunk insertado.

f. Manejo de errores: Si se produce alguna excepción durante el proceso, se captura y se imprime un mensaje de error. La excepción se vuelve a lanzar para que pueda ser manejada en un nivel superior si es necesario.

g. Bloque `finally`: En este bloque, se realizan algunas acciones finales. Si se creó una instancia del cliente de MongoDB (`mongo_client`) y no es `None`, se cierra la conexión. Además, se comprueba si los archivos CSV descargados existen en el sistema y, de ser así, se eliminan.

Creación de índices

La funcionalidad de full-text search de MongoDB nos da la posibilidad de buscar y obtener documentos utilizando text fields. Este se basa en crear índices en campos de texto definidos dentro de una colección. Full-text utiliza el estándar de texto de Lucene. Para habilitar full-text search en una colección, se puede seguir el tutorial en la documentación oficial En este proyecto se utilizan dos métodos:

`mongo_search`

```
mongo_search(query: str, path: str, limit: int, query_type: str)
```

En este método se crea un pipeline de la siguiente manera:

```
pipeline = [
            {
                "$search": {
                    "index": index_name,
                    query_type: {
                        "path": path,
                        "query": query,
                    },
                }
            },
            {"$limit": limit},
        ]
```

Esto genera el json correspondiente a la consulta de full-text que vamos a ejecutar.

- `index_name`: el nombre del índice al que le vamos a aplicar la consulta.
- `query_type`: puede ser "text" o "phrase"
    - `text`: realiza la búsqueda utilizando cada palabra por separado. Ej: si se escribe "You're gonna go far kid", la query va a buscar "You're" "gonna" "go" "far" "kid", es decir, cada una de las palabras en el path que se escoja.

    - `phrase`: realiza una búsqueda utilizando la frase completa. Ej: si se escribe "You're gonna go far kid", la query va a buscar toda la frase en el path que se escoja.
- `path`: esta es la ruta o rutas en las que se va a aplicar la query.
- `query`: esta es la frase que se va a buscar, tomando en cuenta todo lo anterior.
- `limit`: esto dice la cantidad de resultados que se van a obtener.

`mongo_filter_search`

```
mongo_filter_search(paths: list, queries: list, limit: int, query_type: str)
```

En este método se crea un pipeline de la siguiente manera:

```
pipeline = [
            {
                "$search": {
                    "index": index_name,
                    "compound": {
                        "must": [
                            {query_type: {"query": query, "path": path}}
                            for path, query in zip(paths, queries)
                        ]
                    },
                }
            },
            {"$limit": limit},
        ]
```

Como se puede ver, este pipeline tiene varias diferencias importantes. La primera a notar es el uso de la keyword `"compound"`. Este permite hacer búsquedas complejas utilizando `"must"`, `"mustNot"` y `"should"`. En este caso específico no es necesario utilizar compound, pero se hizo de esta manera para ser escalable a futuro.

Explicación API

En primer lugar, se importan las bibliotecas necesarias, como `os` para acceder a variables de entorno, `flask` para crear la aplicación web, `pymongo` para interactuar con MongoDB, `json` para trabajar con datos en formato JSON, `bson` para manejar los ObjectId de MongoDB y `flask_cors` para habilitar CORS (Cross-Origin Resource Sharing) en la aplicación.

A continuación, se define una clase `JSONEncoder` que extiende `json.JSONEncoder` y se utiliza para serializar objetos de tipo `ObjectId` de MongoDB como cadenas.

La función `verify_connection_to_mongo` se encarga de verificar la conexión con MongoDB. Obtiene la cadena de conexión de la variable de entorno `CONNECTION_STRING`, crea un cliente `MongoClient` utilizando la API estable "1" y envía un ping al servidor para confirmar una conexión exitosa.

La función `mongo_search` realiza una búsqueda en la base de datos MongoDB. Recibe como parámetros una consulta `query`, una ruta `path` en la estructura de datos de MongoDB, un límite `limit` para el número de resultados y un tipo de consulta `query_type`. Utiliza el cliente `MongoClient` para conectarse a la base de datos, construye una tubería de agregación para realizar la búsqueda utilizando el operador `$search` de MongoDB, y limita la cantidad de resultados utilizando el operador `$limit`. Luego, serializa los resultados en formato JSON y cierra la conexión con la base de datos.

La función `mongo_filter_search` realiza una búsqueda filtrada en la base de datos MongoDB. Recibe como parámetros una lista de rutas `paths`, una lista de consultas `queries`, un límite `limit` y un tipo de consulta `query_type`. Verifica que las listas de rutas y consultas tengan la misma longitud y luego realiza una búsqueda utilizando el operador `$search` de MongoDB junto con el operador `$must` para combinar las consultas en una búsqueda compuesta. Al igual que en `mongo_search`, se limita la cantidad de resultados y se serializan en formato JSON.

Las rutas de la aplicación Flask están definidas para manejar las solicitudes HTTP. La ruta "/mongo/connection" responde a una solicitud GET y verifica la conexión con MongoDB. La ruta "/mongo/search" responde a una solicitud POST y realiza una búsqueda simple en la base de datos utilizando los parámetros proporcionados en formato JSON. La ruta "/mongo/search/filters" responde a una solicitud POST y realiza una búsqueda filtrada utilizando los parámetros proporcionados en formato JSON.

Finalmente, se configura la aplicación Flask y se inicia el servidor si el script se ejecuta directamente.

Node JS App(Vue)

Para el desarrollo de la página web solicitada se optó por el Framework Vue ya que este es sumamente versátil y posee muchas funcionalidades prehechas, además de ser más sencillo de utilizar que frameworks grandes como Angular. Ya que Vue está diseñado para que las aplicaciones que se desarrollen acá sean modularizadas y con componentes reutilizables, optamos por crear un conjunto de componentes y vistas que son las que utilizarán numerosas veces en el sistema. En cuanto a componentes tenemos LogIn, SignUp, SongInfo y SongPreview.

- **LogIn:** este componente simplemente pide los datos de inicio de sesión al usuario y posee un botón para realizar dicha acción. Esta acción va a Firebase y verifica los datos y si son correctos redirige a la página principal del sistema, si no lo son envía un mensaje de error al usuario.
- **SignUp:** este componente también es muy simple. Pide los datos para registrar un usuario y una vez se indica que el usuario está listo, los verifica y los envía a Firebase para registrar un usuario.
- **SongInfo:** este componente muestra todos los datos de una canción, desde su nombre y autor, hasta su letra. Ya que estos datos son variantes, están guardados en el documento dentro de variables que pueden ser cambiadas según sea necesario enseñar distinta información.
- **SongPreview** este componente se utiliza para mostrar una previsualización de las canciones y que, así, el usuario pueda elegir sobre qué canción ver más detalles. Igualmente recibe por parámetro los datos de la canción a mostrar.

Para el desarrollo de sign up y login se dio uso a la plataforma Firebase, esta permite una capa de seguridad al encriptar el id de usuarios registrados en la misma y poder acceder a esta información mediante el correo y la contraseña ingresada, La conexión se realiza a traves de un enlace con el proyecto creado y con metodos disponibles de la librería para vue llamada igual que la plataforma.

Para efectos de la conexión con el API se utilizó la librería Axios la cual permite la ejecución de HTTP,con lo que se permitió la ejecución de los metodos POST construidos en el API y requeridos para la comunicación con la base de datos MongoDB, la información obtenida por el response de los metodos fue almacenada en una variable e implementada luego para los usos que se necesitara darle a la información.

Pruebas Unitarias

La clase `TestAPI` hereda de `unittest.TestCase`, lo que indica que es una clase de prueba que contiene varios métodos de prueba.

El método `setUp` se ejecuta antes de cada prueba y se encarga de establecer la URL base de la API que se va a probar.

El método `test_mongo_connection` verifica si la conexión con la base de datos MongoDB es exitosa. Envía una solicitud GET a la ruta `/mongo/connection` de la API y verifica que la respuesta tenga un código de estado 200 y un texto que dice "Succesful Connection!".

El método `test_mongo_search` realiza una búsqueda en la base de datos MongoDB utilizando una solicitud POST a la ruta `/mongo/search`. Envía datos de búsqueda específicos y luego verifica que la respuesta tenga un código de estado 200 y el ID del primer resultado coincida con el valor esperado.

El método `test_mongo_search_filters` realiza una búsqueda con filtros en la base de datos MongoDB utilizando una solicitud POST a la ruta `/mongo/search/filters`. Envía datos de búsqueda con varios filtros y luego verifica que la respuesta tenga un código de estado 200 y el ID del primer resultado coincida con el valor esperado.

Cada método de prueba utiliza el método `assertEqual` para verificar si los valores esperados coinciden con los valores reales obtenidos de la API.

Al final del código, se utiliza `unittest.main()` para ejecutar todas las pruebas definidas en la clase `TestAPI`.

Conclusiones y Recomendaciones

Recomendaciones

- Revisar y asegurarse de que todas las bibliotecas y dependencias necesarias estén instaladas correctamente antes de ejecutar el código.
- Ejecutar el código de las pruebas unitarias para verificar que el API este en correcto funcionamiento.

Conclusiones

- Desde el inicio del proyecto nos planteamos dividirnos las cargas de trabajo, de esta manera cada uno de los integrantes tendría una tarea que realizar y de esta manera podríamos realizar el trabajo de manera mucho mas sencilla y mas rápida.
- Docker al igual que en el desarrollo del proyecto pasado, fue igual una herramienta bastante importante para poder tener una imagen dentro de un contenedor en Azure y de esta manera correr nuestras aplicaciones en la nube.

- Gracias a que ya habíamos trabajado previamente en el proyecto pasado con Azure como nuestro servicio en la nube, fue mucho mas fácil saber manipular este, y gracias al trabajo pasado no tuvimos contratiempos en cuanto al uso de este servicio.

- El desarrollo del API fue una tarea bastante sencilla, ya que previamente utilizamos flask para el desarrollo de esta, esta librería de python la tenemos muy familiarizada. También, se implemento pymongo para interactuar con MongoDB, lo cual no es un modulo complejo de usar, gracias a toda la información que hay sobre este.

- El uso de GitHub para el desarrollo del proyecto fue una parte importante para este, decidimos tener diferentes ramas, dependiendo del componente que se iba a trabajar, asi cada integrante podría trabajar en una rama correspondiente, cuando todos los componentes estaban listos, simplemente se hacia merge a nuestra rama principal.

- implementar los UnitTest fue sencillo, gracias a la librería de python que nos lo permite, simplemente se verificaba el funcionamiento de cada ruta de nuestra API obteniendo la respuesta que se esperaba para dicha ruta.

- Desarrollar uin indice de búsqueda fue bastante sencillo y de hecho fue algo sumamente importante para el desarrollo del API, ya que a la hora de buscar la cancion, se utiliza el indice de búsqueda creado el cual con la información proporcionada, nos encontrara las canciones que coincidan.

- El desarrollo de este proyecto fue mucho mas rápido y mas eficiente que el del proyecto pasado, esto gracias a que ya teníamos la experiencia pasada trabajando en grupo, asi como el conocimiento de herramientas previamente utilizadas. El trabajo en grupo de este proyecto fue colaborativo y coordinado, cada integrante contribuyo con su parte para realizar el mejor trabajo posible.

- La documentación del proyecto fue elaborada de la mejor manera, se incluyeron instrucciones claras sobre cómo configurar el entorno de desarrollo, ejecutar la aplicación y realizar pruebas.

- La actitud de cada miembro del equipo fue positiva y colaborativa y esto fue de suma importancia para el éxito del proyecto. Todos estuvieron dispuestos a asumir responsabilidades y a ayudarse mutuamente.