

Instituto Tecnológico de Costa Rica

IC4302 - Bases de Datos II

Documentación Proyecto 2

Profesor: Nereo Campos Araya

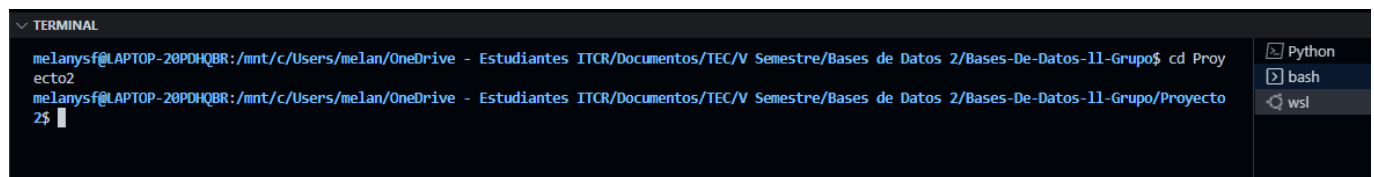
Estudiantes:

- Fiorella Zelaya Coto - 2021453615
- Isaac Araya Solano - 2018151703
- Melany Salas Fernández - 2021121147
- Moisés Solano Espinoza - 2021144322
- Pablo Arias Navarro - 2021024635

Instrucciones para ejecutar su proyecto

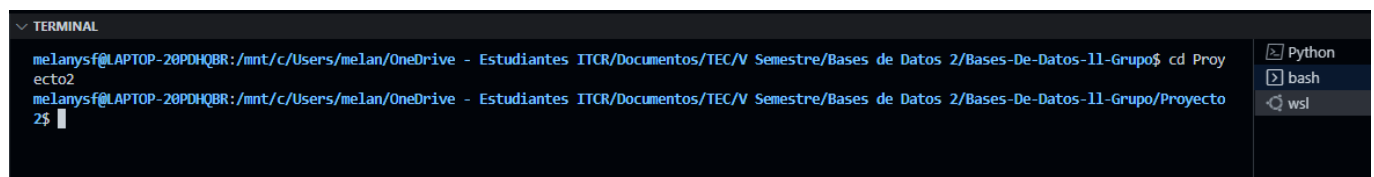
1- Ejecución del loader

1.1. Para ejecutar la imagen debe abrir una consola wsl en la carpeta del proyecto.



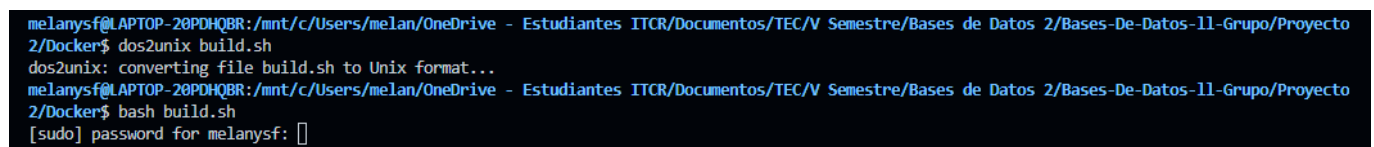
```
melanysf@LAPTOP-20PDHQB: /mnt/c/Users/melan/OneDrive - Estudiantes ITCR/Documentos/TEC/V Semestre/Bases de Datos 2/Bases-De-Datos-II-Grupo$ cd Proyecto
melanysf@LAPTOP-20PDHQB: /mnt/c/Users/melan/OneDrive - Estudiantes ITCR/Documentos/TEC/V Semestre/Bases de Datos 2/Bases-De-Datos-II-Grupo/Proyecto$
```

1.2. Dirigirse a la carpeta **Docker** con "cd Docker"



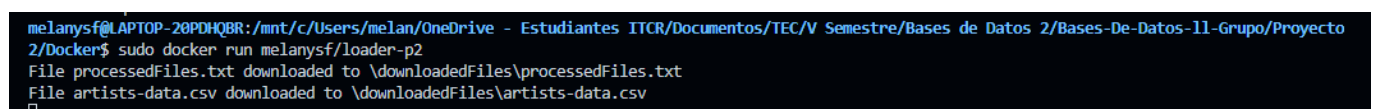
```
melanysf@LAPTOP-20PDHQB: /mnt/c/Users/melan/OneDrive - Estudiantes ITCR/Documentos/TEC/V Semestre/Bases de Datos 2/Bases-De-Datos-II-Grupo$ cd Proyecto
melanysf@LAPTOP-20PDHQB: /mnt/c/Users/melan/OneDrive - Estudiantes ITCR/Documentos/TEC/V Semestre/Bases de Datos 2/Bases-De-Datos-II-Grupo/Proyecto$
```

1.3. Ejecutar "bash build.sh", en caso de problemas, ejecutar "dos2unix build.sh" y luego de nuevo el "bash build.sh".



```
melanysf@LAPTOP-20PDHQB: /mnt/c/Users/melan/OneDrive - Estudiantes ITCR/Documentos/TEC/V Semestre/Bases de Datos 2/Bases-De-Datos-II-Grupo/Proyecto
2/Docker$ dos2unix build.sh
dos2unix: converting file build.sh to Unix format...
melanysf@LAPTOP-20PDHQB: /mnt/c/Users/melan/OneDrive - Estudiantes ITCR/Documentos/TEC/V Semestre/Bases de Datos 2/Bases-De-Datos-II-Grupo/Proyecto
2/Docker$ bash build.sh
[sudo] password for melanysf:
```

1.4. Para ejecutar la imagen, usar "sudo docker run melanysf/loader-p2".

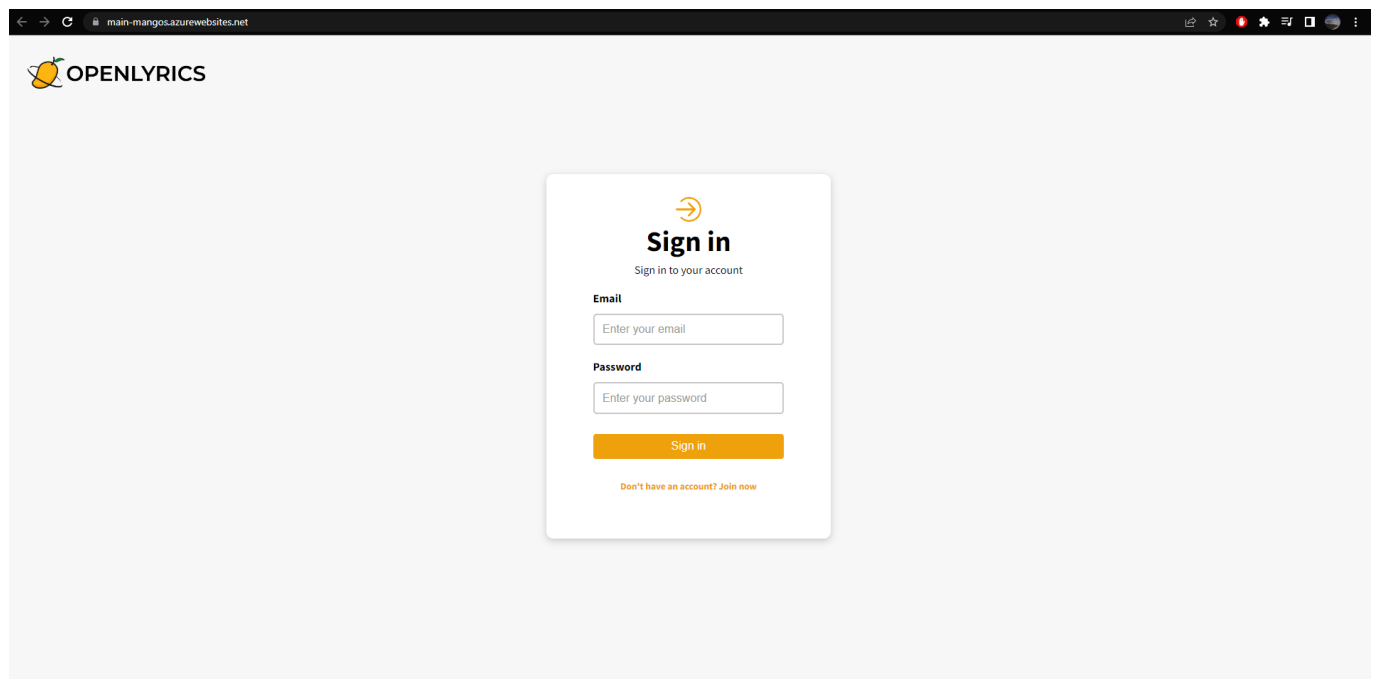


```
melanysf@LAPTOP-20PDHQB: /mnt/c/Users/melan/OneDrive - Estudiantes ITCR/Documentos/TEC/V Semestre/Bases de Datos 2/Bases-De-Datos-II-Grupo/Proyecto
2/Docker$ sudo docker run melanysf/loader-p2
File processedFiles.txt downloaded to \downloadedFiles\processedFiles.txt
File artists-data.csv downloaded to \downloadedFiles\artists-data.csv
```

Nota: Es importante, antes de hacer el build, haber abierto docker desktop.

2- Ejecución de la aplicación en React

1. Para utilizar la aplicación, debemos de ingresar al siguiente link: <https://main-mangos.azurewebsites.net/>



Nota: La aplicación de React fue subida a Azure Services, por lo cual no necesita que se ingresen comandos de manera manual para que la aplicación corra de localmente en el dispositivo, pues la app está alojada en Azure, es decir, los archivos y recursos necesarios para que la aplicación funcione se encuentran en los servidores de Azure.

Componentes

Loader

Parse Artists.csv

Se define la función `parseArtists` para hacer la lectura y el parseo de los artistas de los archivos de artistas.

```
def parseArtists(artistDownloaded_var):  
    try:  
        client = MongoClient(uri) #Mongo Client  
        client = MongoClient(uri, server_api=ServerApi('1'))  
        #Prueba conexión con Mongo DB  
        client.admin.command('ping')  
        print("Successfully connected to MongoDB")
```

Lo primero que se hace en la función es abrir la conexión de Mongo DB, se envía un ping para comprobar que la conexión es correcta.

```
db = client[str(DatabaseName)] #Database to be use  
collection = db[str(ArtistsCollection)] #Database to be use
```

Después, definimos la base de datos y la collection que se va a usar para cargar/bajar datos a Mongo Atlas.

```
#csv_reader to parse the csv file
csv_reader = csv.reader(artistDownloaded_var, delimiter=',')
header = next(csv_reader) #skip the header
```

También, se define un csv reader para hacer la lectura y parseo del csv de artistas, además, se define el delimitador por el cual se separan los campos y se hace un skip de la fila del header.

```
documents = [] #list of documents to be inserted
doc = {} #document to be inserted in the list of documents

artistsNames = collection.distinct("artist") #To verify if the artist is already in the database by name
max = 0
```

Posteriormente se define una lista para almacenar los documentos que serán insertados en la collection de Mongo y un documento que almacena la información del artista que se está leyendo actualmente. También existe la variable **artistsNames** para obtener los nombres de los artistas que existen actualmente en Mongo, esto se usa para hacer la verificación de los artistas que ya han sido agregados a la collection. Se define un max en caso de que se desee limitar la cantidad de artistas que se van a subir a la collection.

```
for row in csv_reader:
    if max == 100:
        break
    if not row[0] in artistsNames:
        #Parse of the csv file
        doc['artist'] = row[0]
        #parsing genres
        doc['genres'] = row[1].split(';')
        doc['songs'] = row[2]
        doc['popularity'] = row[3]
        doc['link'] = row[4]
        #Add the document to the list of documents
        documents.append(doc)
        #Add the artist name to the list of artists names in the database
        artistsNames.append(row[0])
        doc = {} #reset the document
    else:
        print(row[0] + " is already on the collection")
    max = max + 1
```

En el ciclo para recorrer las filas del csv se verifica si se llegó al límite de artistas subidos a Mongo, si aun no se ha alcanzado, se verifica si el nombre del artista esta en la lista de artistsNames, si no esta, debe ser agregado a la lista de documentos, para esto se hace el parse y se le asigna los valores correspondientes a cada parse del documento, para genres se hace un split con el ";" para almacenar los genres como una array.

```
#Insert the list of documents into the database
collection.insert_many(documents)
client.close() #close the connection
```

Se usa la función insert_many para insertar todos los documentos a Mongo y se cierra el cliente.

```

except Exception as e:
    print("Unexpected error:", e)
    return 0
return 1

```

Finalmente, si hay un error se despliega el error en la consola.

Parse Lyrics.csv

Se define la función parseLyrics para hacer la lectura y el parseo de las canciones de los archivos de letras de canciones.

```

def parseLyrics(lyricsDownloaded_var):
    try:
        client = MongoClient(uri) #Mongo Client
        client = MongoClient(uri, server_api=ServerApi('1'))
        #Prueba conexión con Mongo DB
        client.admin.command('ping')
        print("Successfully connected to MongoDB")

```

Lo primero que se hace en la función es abrir la conexión de Mongo DB, se envía un ping para comprobar que la conexión es correcta.

```

db = client[str('OpenLyricsSearch')] #Database to be use
collection = db[str('lyricsCollection')] #Collection to be use

```

Después, definimos la base de datos y la collection que se va a usar para cargar/bajar datos a Mongo Atlas.

```

#csv_reader to parse the csv file
csv_reader = csv.reader(artistDownloaded_var, delimiter=',')
header = next(csv_reader) #skip the header

```

También, se define un csv reader para hacer la lectura y parseo del csv de artistas.

```

documents = [] #list of documents to be inserted
doc = {} #document to be inserted in the database

artistCollection = db[str('artistsCollection')] #Collection of artists
artistDocuments = list(artistCollection.find()) #list of artists documents
songLinks = collection.distinct("songLink") #list of song names in the database

```

Por otro lado, se definen:

- "artistCollection": colección de artistas existentes en la base de datos.

- "artistDocuments": todos los documentos de la colección de artistas.
- "songLinks": lista de todos los nombres de canciones en la base de datos. Se define el delimitador por el cual se separan los campos.
- "documents": lista para los documentos que van a ser insertados en la base de datos.
- "doc": documento que se creará y almacenará la información del documentos "actual" dentro del for para insertarlo en documents.

También se define un max en caso de que se desee limitar la cantidad de artistas que se van a subir a la collection.

```
for row in csv_reader:
    if max == 100:
        break
    #Obtain the artist document from the list of artists documents
    matching_dict = list((d for d in artistDocuments if row[0] == d['link']))
    #Case 1: The artist is not in the database
    if (matching_dict.__len__() == 0):
        print("The artist " + row[0] + " is not in the database")
        continue

    #Case 2: The song is not in the database
    elif(row[2] not in songLinks):
        #Parse of the csv file
        doc['artist'] = matching_dict[0]["artist"]
        doc['genres'] = matching_dict[0]["genres"]
        doc['popularity'] = matching_dict[0]["popularity"]
        doc['songs'] = matching_dict[0]["songs"]
        doc['artistLink'] = matching_dict[0]["link"]
        doc['songName'] = row[1]
        doc['songLink'] = row[2]
        doc['lyric'] = row[3]
        doc['language'] = row[4]

        #Add the song name and the artist name to the list of song names in the database
        songLinks.append(row[2])

        #Add the document to the list of documents
        documents.append(doc)
        doc = {} #reset the document

        max = max + 1
    else:
        print("The song " + row[1] + " by " + matching_dict[0]["artist"] + " is already on the collection")
```

Se define un ciclo para ir por cada fila del csv. Primeramente, se obtiene el documento del artista que hace match con el link del autor del lyric que estamos recorriendo actualmente.

Para insertar datos se verifica lo siguiente:

1. La existencia del artista que se obtuvo mediante el link de la canción, comprobando que "matchingDict" tenga un len superior a 0.. En este caso, se imprime el mensaje en consola y se continua con la siguiente fila.
2. Verifica que la canción que se va a insertar no exista para evitar datos duplicados. Si la canción no existe, entonces se almacenan los datos en "doc" y luego se inserta en "documents". 2.1. Si la canción ya existe, se imprime el mensaje en consola.

Ademas, se utiliza el link de la canción para verificar la unicidad del documento a insertar.

Este link se inserta a "songLinks" (localmente), lo que permite llevar el registro de las canciones que ya existen y las que estamos agregando para verificar que no se inserten datos duplicados en las siguientes iteraciones.

Luego de esto, se vacia el documento actual.

```
#Insert the list of documents into the database
collection.insert_many(documents)
client.close() #close the connection
```

Se usa la función insert_many para insertar todos los documentos a Mongo y se cierra el cliente.

```
except Exception as e:
    print("Unexpected error:", e)
    return 0
return 1
```

Finalmente, si hay un error se despliega el error en la consola.

Download File

Se define la funcion DownloadFile para descargar los archivos desde Azure. Esta funcion recibe como parámetros el nombre del archivo a descargar y el path del archivo.

```
def downloadFile(filename, filePath):
    try:
        # Connection with blob storage
        blob_service_client = BlobServiceClient.from_connection_string(connectionString)
        container_client = blob_service_client.get_container_client(containerName)
        blob_client = container_client.get_blob_client(filename)
```

Primeramente, se realiza la conexión con el Blob Storage.

```
# Opens and reads the archive
with open(filePath, "wb") as my_blob:
    download_stream = blob_client.download_blob()
    my_blob.write(download_stream.readall())
```

Luego, se crea el archivo en modo de escritura binaria utilizando el path recibido por parámetros. Se descarga el archivo desde el Blob Storage y se escribe en el archivo recién creado.

```
currentFile = open(filePath, 'r', encoding='utf-8')
```

Posteriormente, se abre el archivo recién creado en modo lectura con encoding UTF-8 y se almacena en la variable "currentFile".

```
print(f"File {filename} downloaded to {filePath}")
return currentFile
```


Para finalizar con el proceso de descargado, se imprime un mensaje de confirmación en consola y se retorna el archivo recién decargado desde Blob Storage.

```
except Exception as e:
    print(e)
```

Finalmente, si hay un error se despliega el error en la consola.

getAllBlobFiles

Esta función se usa para obtener los archivos que estan en el blob storage y hacer el parseo de los archivos que aún no han sido procesados (Es decir, los que no están en el txt de archivos procesados que se encuentra en el Blob Storage).

 processedFiles.txt

Lo primero que se hace es hacer la conexión con el blob storage.

```
def getAllBlobFiles():
    try:
        # Connection with blob storage
        blob_service_client = BlobServiceClient.from_connection_string(connectionString)
        container_client = blob_service_client.get_container_client(containerName)
```

Despues, se optiene la lista de los blob en el container y se almacenan en la variable **blob_list**. Posteriormente, se descarga el txt que contiene los nombres de los archivos que ya han sido subidos utilizando la función **downloadFile()**.

```
# List all blobs in the container
blob_list = container_client.list_blobs()

# Download txt that contains the processed files
processedFileTxt = downloadFile(ProcessedFiles, path_File + "\\\" + ProcessedFiles)
processedFiles = processedFileTxt.readlines()
```

Despues, se obtienen los archivos que han sido procesados y se agregan a la lista de "files".

```
files = []
for blob in blob_list:
    files.append(blob.name)
```

Se crea un nuevo archivo que va a ser usado para hacer el update del archivo con los nombres de los que ya han sido procesados. También se define la variable **content** y se inicializa con un string vacío. Esta variable almacenará el contenido del archivo actualizado.

Luego, se recorren los **filenames** que estan en la lista de **files** y se verifica si este nombre ya esta en la lista de archivos procesados, si no esta, se verifica si el nombre del archivo tiene artists o lyrics en el filename. Además, se ignora el archivo que tiene los nombres de los archivos procesados. Finalmente, tenemos la variable **content**, en esta se van a agregar los nombres de los archivos que han sido procesados para actualizar el txt de archivos procesados en el Blob Storage.

```
newFile = open(path_File + "\\\" + 'newFile.txt', 'wb')
content = ""
for fileName in files:
    if fileName not in processedFiles:
        currentFile = downloadFile(fileName, path_File + "\\\" + fileName)

        # Verify if the file is an Artist or Lyrics file
        if "artists" in fileName:
            parseArtists(currentFile)
            pass

        elif "lyrics" in fileName:
            parseLyrics(currentFile)
            pass

        elif "processedFiles.txt" == fileName:
            print("skip")
            continue

    content = content + fileName + "\n"
```

Finalmente, se escribe en el file lo que esta en la variable **content** y se cierra este archivo. Se llama a la función para hacer un el update del archivo en el blob.

```
# Write the content in the new processed files txt
newFile.write(content.encode('utf-8'))
newFile.close()

# Update the processed files txt in Blob Storage
updateBlobFile(path_File + "\\\" + 'newFile.txt')

return files
except Exception as e:
    print(e)
```

updateBlobFile

Esta función hace un update de un archivo que se encuentra en el blob storage, en el cual se encuentran los nombres de los archivos que ya han sido procesados.

Primero, se establece la conexión al Blob Storage y se obtiene el archivo buscado (**processedFiles.txt**).

Luego, se abre un archivo en modo de lectura binaria, indicándole el path que recibe la función por parámetros. Este path es el path del archivo txt en el BlobStorage. Una vez abierto el archivo, se guarda el contenido (tipo lista) de este en la variable **content**.

Se intenta hacer un decode a la variable content. Si esto falla, entonces tira la excepción. Si no falla, continúa el proceso para convertir el contenido de esta lista en un string. Para hacer esto, se recorre la lista y se agrega cada línea a **newContent**, la cual es la variable que construye el string con el nuevo contenido.

Por último, se actualiza el archivo en el BlobStorage y se retorna un string de confirmación.

Finalmente, si hay un error se despliega el error en la consola.

```
def updateBlobFile(filepath):
    try:
        # Connection with blob storage
        blob_service_client = BlobServiceClient.from_connection_string(connectionString)
        container_client = blob_service_client.get_container_client(containerName)
        blob_client = container_client.get_blob_client(ProcessedFiles)

        # Uploads the new file to the blob
        with open(filepath, "rb") as data:
            content = data.readlines()
            # Convert the content to a string
            try:
                decoded_list = [element.decode('utf-8') for element in content]
                print(str(decoded_list))
            except Exception as e:
                print(e)

        # Convert the content to a string
        newContent = ""
        for line in decoded_list:
            newContent = newContent + line
        newContent = newContent.encode('utf-8') # Convert the new string to bytes

        blob_client.upload_blob(newContent, overwrite=True) # Overwrites the existing blob

        print(f"File {ProcessedFiles} updated in Blob Storage with {filepath}")
    except Exception as e:
        print(e)
```

selectRandomGenre

Esta función selecciona un genero random para los lyrics.

```
def selectRandomGenre(genres):
    genreIndex = random.randint(0, len(genres)-1)
    selectedGenre = genres[genreIndex]
    return selectedGenre
```

MongoDB

```

-
OpenLyricsSearch

artistsCollection

lyricsCollection

```

Se definen la base de datos OpenLyricsSearch con las collections artist y Lyrics, además, se define un índice con los facets para hacer consultas sobre la información de lyrics.

```

1 {
2   "mappings": {
3     "dynamic": true,
4     "fields": {
5       "artist": {
6         "type": "stringFacet"
7       },
8       "genres": {
9         "type": "stringFacet"
10      },
11      "language": {
12        "type": "stringFacet"
13      },
14      "popularity": {
15        "type": "numberFacet"
16      }
17    }
18  }
19 }

```

API

El API es utilizado para habilitar los distintos endpoints http para las distintas funcionalidades de la aplicación. Existen 3 endpoints distintos, cada uno con su respectiva funcionalidad. A continuación, se listan cada uno de los endpoints y se explica su utilidad:

Facets Endpoint

Método HTTP: GET

<https://main-app.politebush-c6efad18.eastus.azurecontainerapps.io/facets/list/string:phrase>

Este endpoint recibe la frase de la letra de la canción que se está buscando y devuelve la lista de filtros por los que se podrán filtrar los resultados para la aplicación.

```

@app.route('/facets/list/<string:phrase>', methods=['GET'])
def facets(phrase):
    try:
        client = MongoClient(str(uri))

        db = client.get_database(str(DatabaseName))

```

```
collection = db.get_collection(str(LyricsCollection))

pipeline = [
    {
        '$search': {
            'index': 'default',
            'text': {
                'query': phrase,
                'path': 'lyric'
            }
        }
    },
    {
        '$facet': {
            'languageFacet': [
                {
                    '$group': {
                        '_id': '$language',
                    }
                }
            ],
            'genresFacet': [
                {
                    '$group': {
                        '_id': '$genres',
                    }
                }
            ],
            'artistFacet': [
                {
                    '$group': {
                        '_id': '$artist'
                    }
                }
            ]
        }
    }
]

results = collection.aggregate(pipeline)

languages = []
artists = []
genres = []

for document in results:
    for language in document['languageFacet']:
        tmpLanguage = {"name": language['_id']}
        languages.append(tmpLanguage)
```

```

        for artist in document['artistFacet']:
            tmpArtist = {"name": artist['_id']}
            artists.append(tmpArtist)

        for genre in document['genresFacet']:
            tmp = genre['_id']
            if tmp[0] == " ":
                tmp = tmp[1:]
            tmpGenre = {"name": tmp}
            genres.append(tmpGenre)

    return {"languages": languages, "artists": artists, "genres":
removeRepeatedGenres(genres)}
except pymongo.errors.PyMongoError as e:
    return str(e)

```

Search Endpoint

Método HTTP: GET

<https://main-app.politebush-c6efad18.eastus.azurecontainerapps.io/search/string:phrase/string:artist/string:language/string:genre/string:minPop/string:maxPop/string:amountOfSongs>

Este endpoint recibe la frase de la letra de la canción que se está buscando, el artista, el lenguaje, el género, el mínimo y el máximo de la popularidad y la cantidad de canciones y devuelve la lista de resultados compatibles con la búsqueda y los respectivos filtros.

```

@app.route('/search/<string:phrase>/<string:artist>/<string:language>/<string:genre>/<string:minPop>/<string:maxPop>/<string:amountOfSongs>', methods=['GET'])
def search(phrase, artist, language, genre, minPop, maxPop, amountOfSongs):
    try:
        client = MongoClient(str(uri))

        db = client.get_database(str(DatabaseName))

        collection = db.get_collection(str(LyricsCollection))

        searchPipeline = [
            {
                '$search': {
                    'index': 'default',
                    'text': {
                        'query': phrase,
                        'path': 'lyric'
                    }
                }
            }
        ]
    
```

```
    highlightsPipeline = [
        {
            '$search': {
                'index': 'default',
                'text': {
                    'query': phrase,
                    'path': 'lyric'
                },
                'highlight': {
                    'path': 'lyric'
                }
            }
        },
        {
            '$project': {
                'highlights': {'$meta': 'searchHighlights'}
            }
        }
    ]

    if artist != "null":
        artistFilter(searchPipeline, artist)

    if language != "null":
        languageFilter(searchPipeline, language)

    if genre != "null":
        genreFilter(searchPipeline, genre)

    if minPop != "-1" and maxPop != "-1":
        popularityFilter(searchPipeline, minPop, maxPop)

    if amountOfSongs != "-1":
        amountOfSongsFilter(searchPipeline, int(amountOfSongs))

    results = collection.aggregate(searchPipeline)
    highlights = collection.aggregate(highlightsPipeline)

    data = []

    for document in results:
        tempId = str(document['_id'])
        tempDoc = {'_id': tempId, 'artist': document['artist'], 'songLink':
document['songLink'],
                    'songName': document['songName'], 'popularity':
document['popularity']}
        tempHighlights = []
        for highlight in highlights:
            if str(highlight['_id']) == tempId:
                tempHighlights = highlight['highlights']
```

```

        break

    highestScore = 0
    highestHighlight = None
    for highlightedPhrase in tempHighlights:
        if highlightedPhrase['score'] > highestScore:
            highestScore = highlightedPhrase['score']
            highestHighlight = highlightedPhrase

    try:
        processedHighlights = mainPhrase(highestHighlight['texts'])
        tempDoc['highlights'] = processedHighlights[1]
        tempDoc['lyric'] = shortLyric(document['lyric'],
processedHighlights[0])
    except:
        pass
    data.append(tempDoc)

    return {"data": data}
except pymongo.errors.PyMongoError as e:
    return str(e)

```

Details Endpoint

Método HTTP: GET

<https://main-app.politebush-c6efad18.eastus.azurecontainerapps.io/details/string:artist/string:songName>

Este endpoint recibe el link de la canción de la que se desea conocer la información y devuelve toda la información correspondiente para la aplicación.

```

@app.route('/details/<string:artist>/<string:songName>', methods=['GET'])
def details(artist, songName):
    try:
        client = MongoClient(str(uri))

        db = client.get_database(str(DatabaseName))

        collection = db.get_collection(str(LyricsCollection))

        songLink = "/" + artist + "/" + songName

        pipeline = [
            {
                '$search': {
                    'index': 'default',
                    'text': {
                        'query': songLink,
                        'path': 'songLink'
                    }
                }
            }
        ]
    except:
        pass

```

```

        }
    },
    {
        '$limit': 1
    }
]

results = collection.aggregate(pipeline)

data = []
for document in results:
    tempDoc = {'artist': document['artist'], 'genres': document['genres'],
'popularity': document['popularity'],
                'songs': document['songs'], 'songLink': document['songLink'],
'songName': document['songName'], 'lyric': document['lyric']}
    data.append(tempDoc)

return {"data": data}
except pymongo.errors.PyMongoError as e:
    return str(e)

```

Funciones Auxiliares del API

Método mainPhrase

Método que recibe la lista de highlights y devuelve la frase principal de la canción que se relaciona a la búsqueda.

```

def mainPhrase(highlights):
    phrase = ""
    highlightList = []
    hitFlag = False
    for highlight in highlights:
        if highlight['type'] == 'hit':
            phrase += highlight['value']
            hitFlag = True
            highlightList.append(highlight)
        elif highlight['type'] == 'text' and hitFlag == True:
            if len(phrase) <= 80:
                if '\n' in highlight['value']:
                    phrase += highlight['value'].split('\n')[0]
                    tempHighlight = {'type': 'text', 'value':
highlight['value'].split('\n')[0]}
                    highlightList.append(tempHighlight)
                    break
            else:
                phrase += highlight['value']
                highlightList.append(highlight)

```

```
        else:
            break
    return [phrase, highlightList]
```

Método shortLyric

Método que recibe la letra de la canción y la frase principal de la búsqueda y retorna las 4 líneas más cercanas a la canción relacionada con la búsqueda.

```
def shortLyric(lyric, substring):
    jumps = 0

    lyricsArray = lyric.split('\n')
    endIndex = len(lyricsArray) - 1

    newString = ''

    jumps = 0
    find = False
    for i, line in enumerate(lyricsArray):

        if jumps <= 4 and find == False:
            newString += line + '\n'

        if (substring in line or find):
            if find == False:
                find = True
                newString = ''
                jumps = 0
                if endIndex == i:
                    for j in range(4):
                        newString += lyricsArray[i - (4 - j)] + '\n'
                    break

            if line == '':
                jumps -= 1
            if jumps >= 4:
                break

        newString += line + '\n'

    jumps += 1

    return newString
```

Método artistFilter

Método que agrega el filtro por artista a la consulta si el artista se especifica.


```
def artistFilter(pipeline, artist):
    match = {
        '$match': {
            'artist': artist
        }
    }
    pipeline.append(match)
```

Método genreFilter

Método que agrega el filtro por género a la consulta si el género se especifica.

```
def genreFilter(pipeline, genre):
    match = {
        '$match': {
            'genres': genre
        }
    }
    pipeline.append(match)
```

Método languageFilter

Método que agrega el filtro por lenguaje a la consulta si el lenguaje se especifica.

```
def languageFilter(pipeline, language):
    match = {
        '$match': {
            'language': language
        }
    }
    pipeline.append(match)
```

Método popularityFilter

Método que agrega el filtro por popularidad a la consulta si se especifica los rangos.

```
def popularityFilter(pipeline, minPop, maxPop):
    match = {
        '$match': {
            'popularity': {
                '$gte': float(minPop),
                '$lte': float(maxPop)
            }
        }
    }
```

```
}  
pipeline.append(match)
```

Método amountOfSongsFilter

Método que agrega el filtro por cantidad de canciones a la consulta cuando la cantidad de canciones se especifica.

```
def amountOfSongsFilter(pipeline, amountOfSongs):  
    limit = {  
        '$limit': amountOfSongs  
    }  
    pipeline.append(limit)
```

Método removeRepeatedGenres

Método que quita los generos repetidos de una lista de generos para los facets.

```
def removeRepeatedGenres(genres):  
    genresList = []  
    for genre in genres:  
        if genre not in genresList:  
            genresList.append(genre)  
    return genresList
```

App de React

Organización del proyecto

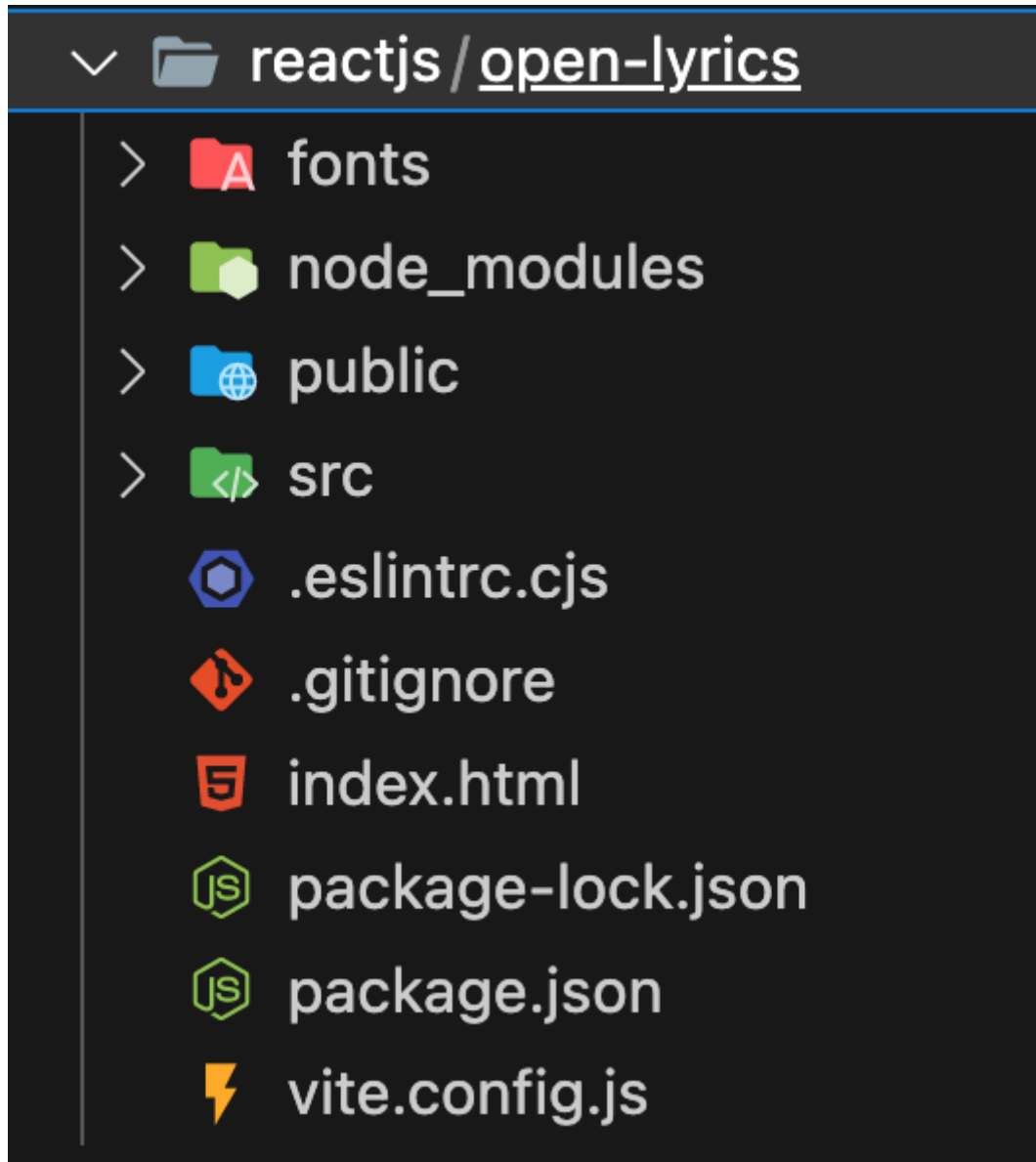
La aplicación web de este proyecto fue realizada utilizando React JS. Para crearlo se utilizó vite. Se le puso de nombre open lyrics.

Como estandarización para desarrollar el proyecto se determinó:

- Empezar el nombre de las rutas y componentes con mayúscula y luego usar camelcase.
- Crear un file.module.css diferente para cada .jsx que contenga html.
- Utilizar las imágenes en ./public y las fuentes en ./fonts.
- Si se utilizan valores 'quemados', estos se deberán transferir al archivo de constantes para tener un mejor control de estos y poder reutilizarlos en la aplicación.
- Si hay elementos visuales que se comparten entre rutas, entonces se utilizará una ruta madre y luego se tendrán las otras rutas adentro de esta.
- Si hay partes del código que se ocupan reutilizar en otro lado, se convertirán en componentes .jsx.
- Manejar una buena documentación interna.
- No mantener imports que no se utilicen en los archivos .jsx.

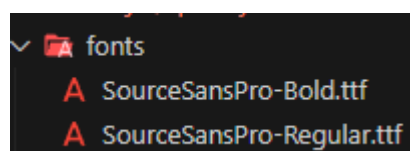
A continuación se muestra el directorio de la aplicación. Se manejan las siguientes carpetas:

- **./fonts** para guardar las tipografías utilizadas en la interfaz.
- **./node_modules** contiene la instalación de los módulos.
- **./public** contiene las imágenes de la interfaz y el favicon.
- **./src** es la carpeta en la que está toda la lógica de la app, esta se explicará en detalle más adelante.
- **./** es la ruta principal y aquí se encuentra el index.html, los json de configuración y el .gitignore para poder utilizar el proyecto en Github.



Se van a ir describiendo cada una de estas carpetas para comprender por completo la estructura de la aplicación web.

1) **./fonts**



Sourcer Sans Pro es la fuente elegida para utilizar en la interfaz. Como no es una fuente que está por defecto en html se descargaron los archivos .ttf y se colocaron aquí para posteriormente ser importadas como las

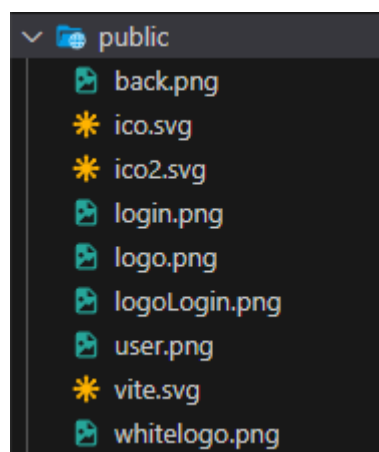
fuentes por defecto de la aplicación.

2) ./node_modules

```
{
  "name": "open-lyrics",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  > Debug
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint src --ext js,jsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview"
  },
  "dependencies": {
    "@babel/runtime": "^7.21.5",
    "firebase": "^9.22.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-icons": "^4.8.0",
    "react-router-dom": "^6.11.1",
    "react-slider": "^2.0.4"
  },
  "devDependencies": {
    "@types/node": "^20.2.3",
    "@types/react": "^18.0.28",
    "@types/react-dom": "^18.0.11",
    "@vitejs/plugin-react": "^4.0.0",
    "eslint": "^8.38.0",
    "eslint-plugin-react": "^7.32.2",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.3.4",
    "vite": "^4.3.2"
  }
}
```

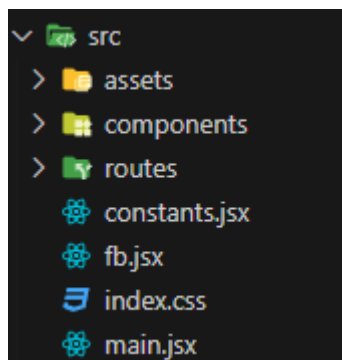
En esta carpeta están instalados todos los módulos necesarios para que la aplicación funcione. En la imagen se muestra el nombre de las dependencias utilizadas. Para cada una de estas se tuvo que ejecutar el `npm install {name}`.

3) ./public

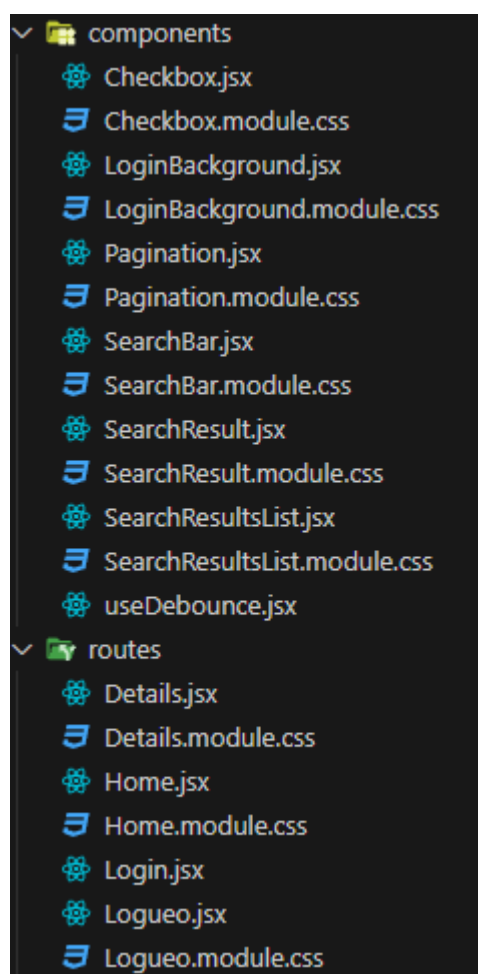


En esta carpeta guardamos las imágenes. Como es la carpeta public no se ocupa poner la ruta en el momento de llamar a los recursos. Se guardan los logos, íconos y favicon.

4) ./src



Esta es la carpeta que contiene todo el código de la aplicación. Como es un proyecto de Vite todos los archivos son .jsx. Esto se divide en components y routes. El archivo main.jsx es el archivo principal de la aplicación, en el que se configura el router y las rutas a utilizar. Se tiene el archivo constants.jsx que va a tener los valores que se utilizan en todo lugar de la app, fb.jsx contiene la configuración de firebase.



En components están los componentes que se reutilizan en react. Se utiliza de esta forma para hacerlo más modular y comprender mejor cómo funciona el código. Para cada componente se tiene su propio archivo de estilo. Lo utilizamos de una forma diferente, en vez de llamar los archivos {name}.css los nombramos {name}.module.css. Esto se hace para poder especificar los estilos específicos para cada elemento del html.

En routes se guardan las rutas de la aplicación. En la especificación se determina que se ocupan cuatro rutas: Login, Create User, Home y Details. Para la ruta de Login y Create User se utilizan los archivos Login.jsx y Logueo.jsx. Para el Home se utiliza Home.jsx y para Details Details.jsx.

Explicación del código

constants.jsx

```
// firebase
export const apiKey = "AIzaSyBMEKpHHG1KGXKYXAZGP74Sc3y9_wAvv1s";
export const authDomain = "proyecto1bd-678f1.firebaseio.com";
export const detailsLink =
  "https://main-app.politebush-c6efad18.eastus.azurecontainerapps.io/details/";
export const projectId = "proyecto1bd-678f1";
export const storageBucket = "proyecto1bd-678f1.appspot.com";
export const messagingSenderId = "38043828434";
export const appId = "1:38043828434:web:7b146c27db19c60f4aad0b";
export const measurementId = "G-WNS4LENGWE";

// routes
export const loginRoute = "/";
export const createUserRoute = "/";
export const homeRoute = "/home";
export const detailsRoute = "/details";

// login route
export const userImg = "/user.png";
export const loginImg = "/login.png";

// loginBackground
export const loginLogo = "/logo.png";

// home route
export const facetsApiLink =
  "https://main-app.politebush-c6efad18.eastus.azurecontainerapps.io/facets/list/";
export const homeLogo = "/logo.png";

// searchBar
export const searchBarLink =
  "https://main-app.politebush-c6efad18.eastus.azurecontainerapps.io/search/phrase/";

// details route
export const backImg = "/back.png";
export const detailsLogo = "/whitelogo.png";
```

En este archivo se tienen todos los valores que se utilizan en la aplicación. Con comentarios se organiza a qué corresponden cada constante. Se exportan para poder ser utilizadas e importadas desde otros archivos.

main.jsx

```
import React from "react";
import ReactDOM from "react-dom/client";
import { RouterProvider, createBrowserRouter } from "react-router-dom";

import * as Constants from "../constants";

import Login from "../routes/Login.jsx";
import LoginBg from "../components/LoginBackground.jsx";
import Home from "../routes/Home.jsx";
import Details from "../routes/Details.jsx";
import "../index.css";

// web page routes
// routes of the web page, there are 4. Login, create user, home and details
const router = createBrowserRouter([
  {
    path: Constants.loginRoute,
    element: <LoginBg />,
    children: [
      {
        path: Constants.createUserRoute,
        element: <Login />,
      },
    ],
  },
  {
    path: Constants.homeRoute,
    element: <Home />,
  },
  {
    path: Constants.detailsRoute,
    element: <Details />,
  },
]);

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

Se crea el router de rutas. Se crea LoginBG que es el background del login y create user. Como es el mismo fondo entonces es la ruta madre y esta tendrá de hija a Login. Las otras rutas son Home y Details.

Ya en la parte inferior se renderizan las rutas con el RouterProvider. Esto hace que la interfaz comience a funcionar. Todos los nombres de las rutas se importan desde el archivo de constantes.

firebase.jsx

```
// Import the functions you need from the SDKs you need
import firebase from "firebase/compat/app";
import "firebase/compat/auth";

import * as Constants from "../constants";

export const firebaseProperties = firebase.initializeApp({
  apiKey: Constants.apiKey,
  authDomain: Constants.authDomain,
  databaseURL: Constants.databaseURL,
  projectId: Constants.projectId,
  storageBucket: Constants.storageBucket,
  messagingSenderId: Constants.messagingSenderId,
  appId: Constants.appId,
  measurementId: Constants.measurementId,
});
```

Contiene la configuración de firebase desde el SDK para poder usar la autenticación. Se exporta esta configuración.

index.css


```
* {
  box-sizing: border-box;
}

@font-face {
  font-family: "Source Sans";
  src: url("../fonts/SourceSansPro-Regular.ttf");
}

@font-face {
  font-family: "Source Sans Bold";
  src: url("../fonts/SourceSansPro-Bold.ttf");
}

:root {
  min-height: 100vh;
  font-family: Source Sans, Inter, Avenir, Helvetica, Arial, sans-serif;
  font-size: 16px;
  line-height: 24px;
  font-weight: 400;
  /* background-image: radial-gradient(ellipse at top left, #7f50e4, #4c05d0); */

  /* Background color for the login and create user pages */
  background-color: #f7f7f7;
  font-synthesis: none;
  text-rendering: optimizeLegibility;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  -webkit-text-size-adjust: 100%;
}

body {
  margin: 0;
  /* text-align: center; */
  color: #000000;
}
```

En este css está la configuración general de la aplicación. Se importan las fuentes y se les coloca el nombre a utilizar. También se coloca el color del fondo y otros valores por defecto para que sean heredados a todos los archivos de la página.

login.jsx

```
import { Outlet, useNavigate } from "react-router-dom";
import React, { useEffect } from "react";
import { firebaseProperties } from "../fb";
import Logueo from "../Logueo";

// style
import * as Constants from "../constants";

function Login() {
  const navigate = useNavigate();
  const [usuario, setUsuario] = React.useState(null);

  // handles the login action
  useEffect(() => {
    firebaseProperties.auth().onAuthStateChanged((usuarioFirebase) => {
      console.log("you are already logged in with:", usuarioFirebase);
      setUsuario(usuarioFirebase);
    });
  }, []);

  // handles logout
  const logOut = () => {
    firebaseProperties.auth().signOut();
  };

  // handles the navigation to the home page
  useEffect(() => {
    if (usuario) {
      navigate(Constants.homeRoute);
    }
  }, [usuario, navigate]);

  return (
    <
      <Outlet />
      {logOut()}
      {!usuario && <Logueo setUsuario={setUsuario} />}
    </>
  );
}

export default Login;
```

Se importan los datos de firebase para manejar el inicio de sesión y la creación de los usuarios. En este jsx no se maneja el html, en este está la lógica de autenticación. Si se logra iniciar sesión, el usuario será redirigido a la ruta Home para que pueda empezar a hacer las búsquedas.

logueo.jsx

```
return (
  <div>
    { /* <h1> {isRegistering ? "Regístrate" : "Inicia sesión"}</h1> */ }
    <form onSubmit={handleSubmit} className={classes.loginForm}>
      {isRegistering ? (
        <img src={Constants.userImg} alt="user icon" />
      ) : (
        <img src={Constants.loginImg} alt="login icon" />
      )}

      <h1 className={classes.title}>
        { " " }
        {isRegistering ? "Create account" : "Sign in"}
      </h1>
      {isRegistering ? (
        <p>Personal information</p>
      ) : (
        <p>Sign in to your account</p>
      )}

      <div className={classes.box}>
        <label className={classes.subTitle} htmlFor="email">
          Email
        </label>
        <div className={classes.inputContainer}>
          <input
            className={classes.input}
            type="email"
            value={emailText}
            onChange={handleChangeEmail}
            required={true}
            placeholder="Enter your email"
            id="emailField"
          />
          <div className={classes.highlight}></div>
        </div>

        <label className={classes.subTitle} htmlFor="passwordField">
          Password
        </label>
        <div className={classes.inputContainer}>
          <input
            className={classes.input}
            type="password"
            required={true}
            value={passwordText}
          />
        </div>
      </div>
    </form>
  </div>
)
```

Este solo es un fragmento de este archivo. En este también se utiliza firebase para poder hacer la creación o inicio de sesión. Para la interfaz se usa un form con los campos de email y contraseña. Estos dos inputs deben ser llenados para poder ingresar. Si ocurre un error la aplicación lo notificará. Cuando se crea un usuario la sesión se iniciará automáticamente y se redigirá a Home.

Aquí se evidencia cómo es que se le dan los estilos a los elementos del html utilizando el elemnto importado llamado `classes`.

checkbox.jsx

```
// receives the list of the facet, the list of selected
// elements of the facet, its set and the prefix to differentiate the ids
const Checkbox = ({ list, setList, selected, setSelected, prefix }) => {
  const handleChange = (e, index) => {
    const activeData = document.getElementById(index).checked;
    // console.log(document.getElementById(index));

    if (activeData) {
      setSelected((oldData) => {
        if (oldData.includes(e.target.value)) {
          return [];
        } else {
          return [e.target.value];
        }
      });
    } else {
      setSelected(selected.filter((value) => value !== e.target.value));
    }

    // if (activeData) {
    //   setSelected((oldData) => [...oldData, e.target.value]);
    // } else {
    //   setSelected(selected.filter((values) => values !== e.target.value));
    // }
  };

  return (
    <div className={classes.checkbox}>
      {list.map((item, i) => (
        <div className={classes.individualCheck} key={i}>
          <input
            className={classes.checkboxbutton}
            id={item.name + prefix}
            type="checkbox"
            value={item.name}
            checked={selected.includes(item.name)}
            onChange={(e) => handleChange(e, item.name + prefix)}
          />
          <span>{item.name}</span>
        </div>
      ))}
    </div>
  );
}
```

Este .jsx contiene el componente de checkbox. Es un área que almacena varios checkbox que se cargan desde una lista y aplicándoles el método `map()` se genera un cuadrado para cada elemento. Como cada checkbox debe de tener un id diferente para poder trabajar correctamente, el componente recibe un prefijo que se utilizará para diferenciar los identificadores únicos. El texto que se muestra en cada checkbox es el .nombre de cada objeto de la lista del facet.

Cuando un checkbox se presiona, se activa la función `handleChange()` que recibe el evento y el índice del elemento. Aquí se comprueba si el checkbox ya estaba activado o no, si no estaba activado entonces se procede a activar, pero si estaba activado se desactivará.

Para los facets de artistas, géneros e idiomas solo se puede seleccionar un elemento a la vez, por esta razón es que si un checkbox está activo y luego se presiona otro, se desactivará el anterior para activar el nuevo.

Nota: Abajo de esta lógica está la versión anterior, en la que se pueden seleccionar más de un checkbox a la vez.

LoginBackground.jsx

```
import { Outlet } from "react-router-dom";

import classes from "../LoginBackground.module.css";
import * as Constants from "../constants";

function LoginBackground() {
  return (
    <>
      <div className={classes.body}>
        <img className={classes.image} src={Constants.loginLogo} alt="logo" />
      </div>
      <Outlet />
    </>
  );
}

export default LoginBackground;
```

En este jsx se crea el fondo del apartado o vista de inicio de sesión o registro de usuario. Además, se carga y define el logo correspondiente a la aplicación.

Pagination.jsx

```
// Se encarga de gestionar la paginación de los resultados dividiéndolos en páginas separadas
const Pagination = ({
  totalPosts,
  postsPerPage,
  setCurrentPage,
  currentPage,
}) => {
  let pages = [];

  const totalPages = Math.ceil(totalPosts / postsPerPage);

  // Número de páginas a mostrar antes y después de los puntos suspensivos
  const visiblePages = 2;

  if (totalPages <= visiblePages + 2) {
    // Mostrar todas las páginas si no hay suficientes páginas para mostrar con puntos suspensivos
    for (let i = 1; i <= totalPages; i++) {
      pages.push(i);
    }
  } else {
    const currentPageIndex = currentPage;

    // Agregar página inicial
    pages.push(1);

    // Agregar páginas antes de los puntos suspensivos
    let startPage = Math.max(2, currentPageIndex - visiblePages);
    let endPage = Math.min(startPage + visiblePages * 2, totalPages - 1);

    if (startPage > 2) {
      pages.push("...");
    }

    for (let i = startPage; i <= endPage; i++) {}
  }
}
```

Este es solo una parte del archivo. Este jsx se encarga de generar la paginación de los resultados que se van a mostrar una vez que se ha realizado la búsqueda. En este componente se calcula el número total de páginas en relación del número de elementos que se quieren mostrar por página y el total de elementos para posteriormente generar una lista de botones que representan las páginas y permiten el desplazamiento entre las páginas.

SearchBar.jsx

```

// search bar in which the call to the api
// to request the lyrics of songs is handled
export const SearchBar = ({
  setInput,
  artists,
  languages,
  genres,
  setResults,
  valuesPopularity,
  valuesSongs,
  setCurrentPage,
}) => {
  const [inputSearch, setInputSearch] = useState("");
  const debounceValue = useDebounce(inputSearch, 300);

  // this useEffect is responsible for calling the api to request
  // the results every time something is written in the search bar
  // or when an element of the facets is marked or modified
  useEffect(() => {
    const artistsParam = artists.length > 0 ? artists : null;
    const languagesParam = languages.length > 0 ? languages : null;
    const genresParam = genres.length > 0 ? genres : null;

    setInput(inputSearch);
    setCurrentPage(1);

    const getData = async () => {
      if (inputSearch === "") {
        setResults([]);
        return;
      } else {
        fetch(
          Constants.searchBarLink +
            inputSearch +
            "/" +
            artistsParam +

```

Este es solo una parte del archivo. En este jsx se encuentra el componente SearchBar, este componente maneja la barra de búsqueda para buscar las letras de las canciones. En este componente al escribir en la barra de búsqueda, luego de un tiempo gracias al "useDebounce.jsx" que se explicará después, se hace una llamada al API con los parámetros de búsqueda, como la letra escrita en la barra, artistas, idiomas y géneros, para posteriormente mostrar los resultados de la búsqueda.

SearchResult.jsx

```

export const SearchResult = ({ result }) => {
  const [lyric, setLyric] = useState("");
  const navigate = useNavigate();

  useEffect(() => {
    if (result) {
      const regex = /\n|\r\n|\n\r/g;
      let formattedLyric = result.fragment.replace(regex, "<br>");

      const hitValues = result.highlights
        .filter((item) => item.type === "hit")
        .map((item) => item.value);

      const replaceFn = (match) => `<strong>${match}</strong>`;

      formattedLyric = formattedLyric.replace(
        new RegExp(`\\b(${hitValues.join("|")})\\b`, "gi"),
        replaceFn
      );

      setLyric(formattedLyric);
    }
  }, [result]);

  return (
    <div className={classes.searchResult}>
      <div className={classes.name}>{result.name}</div>
      <div className={classes.artist}>{result.artist}</div>
      <div className={classes.songFragment}>
        <div
          className={classes.lyric}
          dangerouslySetInnerHTML={{ __html: lyric }}
        />
      </div>
      <Link
        to="/details"
        state={{ info: JSON.stringify(result.link) }}
        className={classes.button}
      >
        Show details
      </Link>
    </div>
  );
}

```

Se encarga de mostrar los resultados de la búsqueda de letras de las canciones. En este se reciben los resultados de la búsqueda del API y se colocan uno debajo del otro, se muestra el nombre de la canción, el artista y una fracción de la letra de la canción en donde se vea la coincidencia que esta tiene con lo ingresado en la barra de búsqueda. (Las palabras coincidentes mostradas en el pequeño fragmento de la canción son resaltadas para visualizar la relación entre lo buscado y los resultados)

SearchResultsList.jsx


```
import classes from "../SearchResultsList.module.css";
import { SearchResult } from "../SearchResult";

// list of all results, for each one a SearchResult is created
export const SearchResultsList = ({ results }) => {
  return (
    <div className={classes.resultsList}>
      {results.map((result, id) => {
        return <SearchResult result={result} key={id} />;
      })}
    </div>
  );
};

export default SearchResultsList;
```

Este jsx contiene un componente llamado SearchResultsList, el cual se encarga de mostrar una lista de los resultados de búsqueda. En este se itera sobre cada elemento del arreglo que contiene los resultados de la búsqueda y se crea un componente SearchResult para cada uno de ellos. Como se explicó anteriormente en el apartado de SearchResult.jsx, cada uno de estos componentes se crean utilizando los datos del resultado correspondiente a cada uno de ellos.

useDebounce.jsx

```
import { useEffect, useState } from "react";

export const useDebounce = (value, delay) => {
  const [debounceValue, setDebounceValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebounceValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debounceValue;
};
```

Se crea un hook llamado useDebounce, el cual retrasa la ejecución de una acción hasta que un valor se mantenga constante durante un cierto tiempo. Este es utilizado para que las búsquedas no se realicen cada vez que se cambia una letra en la barra de búsqueda, sino que se realice cuando ya se deje de modificar el texto durante un tiempo definido. (Disminuye la cantidad de consultas que se le deben de hacer al API y mejora el rendimiento de la aplicación)

Home.jsx

```
// load all the data from the api to the facets
useEffect(() => {
  console.log("input:" + Constants.facetsApiLink + input);
  if (input === "") {
    handleCheckAllChange();
    eraseFacets();
  }
  if (input !== "") {
    fetch(Constants.facetsApiLink + input)
      .then((response) => response.json())
      .then((data) => {
        if (
          Array.isArray(data.artists) &&
          Array.isArray(data.languages) &&
          Array.isArray(data.genres)
        ) {
          setArtistsFacet(
            data.artists.sort((a, b) => a.name.localeCompare(b.name))
          );
          console.log("artistas facet: " + artistsFacet);
          setLanguagesFacet(
            data.languages.sort((a, b) => a.name.localeCompare(b.name))
          );
          setGenresFacet(
            data.genres.sort((a, b) => a.name.localeCompare(b.name))
          );
        }
      })
      .catch((error) => {
        console.log("Error getting the data:", error);
      });
  }
}, [input, setInput]);

//update song slider range
useEffect(() => {
  setMaxSongs(results.length);
  setValuesSongs(results.length);
}, [results, maxSongs]);
```

En este jsx es en donde se juntan todas las otras partes, pues se podría decir que esta es la pagina principal en donde se implementan la mayoría de funcionalidades de esta aplicación. En este se administran los filtros, las búsquedas y los resultados en la interfaz de búsqueda. Se utilizan hooks de estado para controlar los valores de los filtros y la paginación. También se utilizan hooks de useEffect para realizar la llamadas a una API y actualizar datos. En esta se muestra el logo de la aplicación, la barra de búsqueda, los filtros, los deslizadores y una lista de resultados con paginación.

Docker

Como se está utilizando vite junto a React, para poder encapsular la app en una imagen de Docker se tienen que hacer unos cambios y agregar archivos.

En el archivo vite.config.js se agregaron las configuraciones del server. Aquí se especifica el watch, host, strictPort y port. Con todas estas configuraciones ya se puede crear la imagen de Docker.

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  server: {
    watch: {
      usePolling: true,
    },
    host: true, // needed for the Docker Container port mapping to work
    strictPort: true,
    port: 5173, // you can replace this port with any port
  },
});
```

En el directorio principal de la app se agregó .dockerignore y Dockerfile. En el .dockerignore se agregan las carpetas y files que no se tienen que agregar a la imagen. Estos son node_modules, npm-debug.log, build, .git, *.md y .gitignore. En el Dockerfile está la configuración de la imagen. Se importa node, se copia package.json porque ahí se encuentran los módulos que se tienen que instalar, se instalan, se expone el puerto 5173 que es el que utiliza vite y se ejecuta el comando `npm run dev`.

.dockerignore

```
node_modules
npm-debug.log
build
.git
*.md
.gitignore
```

Dockerfile

```
FROM node

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

EXPOSE 5173

CMD ["npm", "run", "dev"]
```

Teniendo todo esto listo, solo hace falta ejecutar los comandos para hacer la imagen y subirla a Dockerhub. Son los siguientes:

- `docker build -f Dockerfile -t moisose/open-lyrics .`
- `sudo docker push moisose/open-lyrics`

Ahora bien, si se quiere ejecutar la imagen en local, se tiene que ejecutar:

- `docker run -p 5173:5173 -d moisose/open-lyrics`

En el navegador se accede al localhost en el puerto 5173 y la app funciona correctamente.

Para el caso de este proyecto, se solicita que la app funcione en Azure services. Para poder configurar esto se tiene que ir a la carpeta original de este proyecto. Una vez ahí se accede a: `./infraestructure-p2/container_services.tf`. Ahí en la parte de resource, luego en site_config y en application_stack, se encuentran los valores para docker_image y docker_image_tag. En esta sección solamente es poner el nombre de la imagen que se subió a Dockerhub que en este caso es `docker.io/moisose/open_lyrics` y en el docker_image_tag se coloca `latest`.

Con esto configurado ya está todo listo para utilizar la aplicación. Si por alguna razón se tiene que hacer alguna modificación al código, entonces se tiene hacer de nuevo la imagen y subirla a Dockerhub. Azure se encargará automáticamente de actualizar la app en un corto tiempo.

Pruebas realizadas

Resultados de las pruebas unitarias

API

Las pruebas unitarias fueron realizadas mediante este código.

```
baseUrl = 'https://main-app.politebush-c6efad18.eastus.azurecontainerapps.io'
phrase = 'another'
artist = 'beyonce'
language = 'es'
genre = 'Pop'
minPop = '0'
maxPop = '100'
amountOfSongs = '10'
songName = 'robot.html'

You, 1 minute ago | 1 author (You)
class test_api(unittest.TestCase):

    def testFacets(self):
        url = baseUrl + '/facets/list/' + phrase

        response = requests.get(url)
        self.assertEqual(response.status_code, 200)

        condition = ('Error' in response.json())
        self.assertEqual(condition, False)

    def testSearch(self):
        url = baseUrl + '/search/' + phrase + '/' + artist + '/' + language + '/' + genre + '/' + minPop + '/' + maxPop + '/' + amountOfSongs

        response = requests.get(url)
        self.assertEqual(response.status_code, 200)

        condition = ('Error' in response.json())
        self.assertEqual(condition, False)

    def testDetails(self):
        url = baseUrl + "/details/" + artist + "/" + songName

        response = requests.get(url)
        self.assertEqual(response.status_code, 200)

        condition = ('Error' in response.json())
        self.assertEqual(condition, False)

if __name__ == "__main__":
    unittest.main()
```

Los resultados fueron correctos en las 3:

```
...
-----
Ran 3 tests in 1.494s

OK
```

Conclusiones

- 1- La comunicación entre el los miembros de grupo de trabajo es fundamental para un buen desarrollo del proyecto.
- 2- Se debe mantener una buena organización para poder realizar el trabajo.
- 3- Es de gran importancia entender los conceptos básicos vistos en clase para realizar el proyecto.
- 4- El tener un buen control de versiones y la correcta utilización de github facilita el trabajo en equipo.
- 5- Se deben aplicar buenas prácticas de programación para mantener el orden.
- 6- Mantener la estructura definida del proyecto es esencial para evitar el desorden.
- 7- Se debe desarrollar un código legible y entendible.

- 8-** Se debe organizar el equipo de trabajo desde el día 1.
- 9-** Se debe tener una estructura clara y ordenada del proyecto y lo que requiere.
- 10-** Es importante la división de trabajo para poder desarrollar todos los componentes.

Recomendaciones

- 1-** Hacer reuniones periódicas para discutir los avances del proyecto y mejorar la comunicación.
- 2-** Mantener la organización de la tarea, siguiendo la infraestructura y recomendaciones dadas por el profesor.
- 3-** Repasar los conceptos vistos en clase y complementar con investigación mejorar el entendimiento y aumentar la eficacia con la que se trabajará.
- 4-** Hacer uso de github para el control de versiones y trabajo en conjunto.
- 5-** Seguir un estándar de código.
- 6-** Seguir aprendiendo y enriqueciendo el conocimiento después de finalizar el trabajo.
- 7-** Investigar sobre las diferentes herramientas esenciales para desarrollar la solución e ir tomando apuntes sobre los aspectos importantes de cada uno de estas. Esto facilitará el desarrollo de la solución.
- 8-** Tener una buena estructura del proyecto y dividir el proyecto de forma funcional para avanzar progresivamente.
- 9-** Repartir y asignar tareas a cada integrante del equipo.
- 10-** Definir roles en el equipo de trabajo para mantener el orden y procurar buena dinámica de trabajo.