CSCC01 - Software Engineering
Assignment 3

University of Toronto Mississauga
Winter 2020

A3 - Microservices

DESCRIPTION

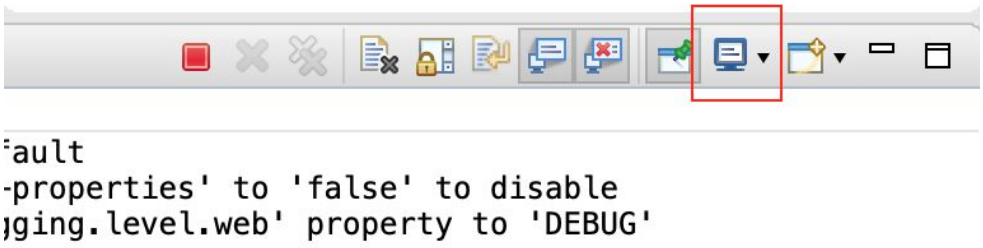
You will be creating backend Java API using 2 microservices that will communicate with each other to accomplish specific use-cases. You will be using MongoDB and Neo4J databases for the two services, and will be implementing this assignment using the Java Spring framework.

OBJECTIVE

1. To understand general design and purpose of microservice architecture
2. To understand and establish communication between multiple microservices
3. To create Node REST APIs that are supported by Neo4j graph and Mongo databases
4. To utilize Git and code style correctly

ENVIRONMENT SETUP

1. Install MongoDb and Neo4J databases (username: neo4j, password: 1234), and the Eclipse IDE and Maven (you sadly **must** use Eclipse for this assignment)
2. Clone the starter code repo
3. Import both services (Song and Profile) in Eclipse as individual Maven projects (File ➔ Import ➔ Maven ➔ Existing Maven Project) from the clone repo under
4. Right click each project and click “Run As” ➔ “Java Application” to run service. Note: You will not need to restart each service on a code change, the server will auto-refresh (there is not need to do any maven compilation, Eclipse will handle all that)
5. Click on the blue monitor to switch between the console output of both microservices



HOW TO SUBMIT YOUR WORK AND WHAT TO SUBMIT

Your work will be submitted through GitHub classroom on the master branch. Please ensure that your final submissions are merged into the master branch prior to the submission deadline.
Please ensure that your code runs on the Linux lab machines correctly prior to submitting it

Please submit the two Eclipse projects under profile-microservice and song-microservice. The project directory must look like the following (you can ignore the target folders in the song and profile microservices):

```
.
├── profile-microservice
│   ├── HELP.md
│   ├── mvnw
│   ├── mvnw.cmd
│   ├── pom.xml
│   ├── src
│   │   ├── main
│   │   └── test
│   └── target
│       ├── classes
│       └── test-classes
└── song-microservice
    ├── HELP.md
    ├── mvnw
    ├── mvnw.cmd
    ├── pom.xml
    ├── src
    │   ├── main
    │   └── test
    └── target
        ├── classes
        └── test-classes
```

STARTER FILES

- **starter_code/profile-microservice** - Contains the Eclipse project that is responsible for the Profile microservice
- **starter_code/song-microservice** - Contains the Eclipse project that is responsible for the Song microservice
- ***Controller Classes** - Microservice routes are placed here. This is the class that will be receiving and returning response to Clients
- ***Impl Classes** - Microservice database operations and functions are placed here
- **DbQueryStatus.java** - The object that needs to be returned from every database operation which includes the result and possible data that the route may require
- **DbQueryExecResult.java** - The status of DB function calls
- The following files need to be implemented
    - ProfileDriverImpl.java
    - PlaylistDriverImpl.java
    - ProfileController.java
    - SongController.java
    - SongDalImpl.java

REQUIREMENTS

Please implement the following REST API

Each REST API must return the OK status if the operation was successful, otherwise the appropriate not-"OK" status must be returned (i.e. "INTERNAL_SERVER_ERROR" or "NOT_FOUND"). Furthermore, all of the microservice routes will also return the path that was called via the "path" property (you don't have to modify this part of the code) for your reference (we will not be checking this when marking).

Song Microservice

Contains all Song information and routes. ***This service must run on port 3001***

- GET /getSongTitleById/{songId}
    - **Description:** Retrieves the song title given the song
    - **URL Parameters:** songId - a String that represents the _id of a specific song
    - **Expected Response:**
        - "status"
            - "OK", if the title was retrieved successfully
            - <string>, any other status if the song was not retrieved successfully

        - “data”: <string>, the title of the song that was retrieved
    - **Example Response:**

```
{
"status": "OK",
"data": "Never going to give you up"
}
```

- DELETE /deleteSongById/{songId}
    - **Description:** Deletes the song from the MongoDB database and from all Profiles that have the songId added in their “favourites” list
    - **URL Parameters:** songId - a String that represents the _id of a specific song
    - **Expected Response:**
        - “status”
            - “OK”, if the song was deleted successfully from all locations
            - <string>, any other status if the song was unable to be deleted
- POST /addSong
    - **Description:** Adds a Song to the Song database
    - **Body Parameters:**
        - songName - The name of the song
        - songArtistFullName - The name of the song artist
        - songAlbum - The album the song is a part of
    - **Expected Response:**
        - “status”
            - “OK”, if the song was deleted successfully from all locations
            - <string>, any other status if the song unable to be added
        - “data” - The songName, songArtistFullName, songAlbum, songAmountFaourites, and the id of the Song should all be returned if the Song is added successfully
    - **Example Response:**

```
{
    "data": {
        "songName": "testDeleteSong",
        "songArtistFullName": "111",
        "songAlbum": "testDeleteSongArtist",
        "songAmountFavourites": "0",
        "id": "5d65df689a2efe318808f4cc"
    }
    "status": "OK"
}
```

- PUT /updateSongFavouritesCount/{songId}?shouldDecrement=
    - **Description:** Updates the song’s favourites count

    - **URL Parameters:**
        - songId - a String that represents the _id of a specific song
    - **Query Parameters:**
        - shouldDecrement - States whether the song count should be incremented or decremented by 1. Accepts string values of “true” or “false”
    - **Expected Response:**
        - “status”
            - “OK”, if the song was updated successfully from all locations
            - <string>, any other status if the song’s favourite count was unable to be updated

Profile Microservice

Contains all Profile and Playlist information and routes. ***This service must run on port 3002***

- POST /profile
    - **Description:** Adds a profile to the Profile database, creates userName-favorites playlist and creates relation (nProfile:profile)-[:created]->(nPlaylist:playlist)
    - **Body Parameters:**
        - userName - The name of the Profile
        - fullName - The full name of the User of the profile
        - password- The password associated with the Profile
    - **Expected Response:**
        - “status”
            - “OK”, if Profile is created and added correctly to the database
            - <string>, any other status if the Profile was unable to be created
- PUT /followFriend/{userName}/{friendUserName}
    - **Description:** Allows a Profile to follow another Profile and become a friend
    - **URL Parameters:**
        - userName - The userName of the Profile that will follow another Profile
        - friendUserName - The userName of the Profile that will be followed
    - **Expected Response:**
        - “status”
            - “OK”, if the user is able to follow the specified User
            - <string>, any other status if the Profile was unable to follow the specified Profile
- PUT /unfollowFriend/{userName}/{friendUserName}
    - **Description:** Allows a Profile to unfollow another Profile and no longer be “friends” with them

    - **URL Parameters:**
        - userName - The userName of the Profile that will unfollow another Profile
        - friendUserName - The userName of the Profile that is being unfollowed
    - **Expected Response:**
        - “status”
            - “OK”, if the user is able to unfollow the specified User
            - <string>, any other status if the Profile was not able to unfollow the specified Profile
- PUT /likeSong/{userName}/{songId}
    - **Description:** Allows a Profile to like a song and add it to their favourites. You can like the same song twice.
    - **URL Parameters:**
        - userName - The userName of the Profile that is going to like the song (i.e. add the song to their favourites)
        - songId - a String that represents the _id of the song that needs to be liked
    - **Expected Response:**
        - “status”
            - “OK”, if the User is able to like the Song
            - <string>, any other status if the Profile was not able to like the specified Song
- PUT /unlikeSong/{userName}/{songId}
    - **Description:** Allows a Profile to unlike a song and remove it from their favourites. You cannot unlike the same song twice.
        - **URL Parameters:**
            - userName - The userName of the Profile that is going to unlike the song
            - songId - a String that represents the _id of the song that needs to be unliked
        - **Expected Response:**
            - “status”
                - “OK”, if the User is able to unlike the Song
                - <string>, any other status if the Profile was not able to unlike the specified Song (including going below 0)
- GET /getAllFriendFavouriteSongTitles/{userName}
    - **Description:** Returns the Song names of all of the Songs that the User’s friends have liked
        - **URL Parameters:**
            - userName - The userName of the Profile whose friends songs we will retrieve

        - songId - a String that represents the _id of the song that needs to be unliked
    - **Expected Response:**
        - “status”
            - “OK”, if the User is able to unlike the Song
            - <string>, any other status if the Profile was not able to unlike the specified Song
    - **Example Response:**

```
{
    "data": {
        "tahmid": [
            "The Lego Movie",
            "Henry IV",
            "Land of Milk and Honey"
        ],
        "shabaz": [
            "Sliver",
            "Off the Black",
            "The Lego Movie"
        ]
    },
    "status": "OK"
}
```

Database

You also have to implement the following DB methods in each respective service to use within the routes. Below is a description of what each DB method should do

Song Microservice

- The following requirements are implemented in the starter code for you, but please ensure that the following requirements are satisfied
    - You database **must** be called “csc301-test”
    - Your collection **must** be called “songs”
    - Your database **must** be connected to port 27017
- Song Schema
    - **_id** - The ObjectId of the document (i.e. a unique identifier for each document). The id is auto-generated by Mongo when a document is inserted into the database
    - **songName** - The name of the song
    - **songArtistFullName** - The full name of the artist of the song

    - **songAlbum** - The album the song belongs to
    - **songAmountFavourites** - The amount of Profiles that have favourited the Song
- DB Methods to Implement
    - **DbQueryStatus addSong(Song songToAdd)** - Adds the specified song to the Mongo DB
    - **DbQueryStatus findSongById(String songId)** - Returns the specified Song that matches the songId
    - **DbQueryStatus getSongTitleById(String songId)** - Returns the song title which matches the songId
    - **DbQueryStatus deleteSongById(String songId)** - Deletes the specified song from the Mongo DB
    - **DbQueryStatus updateSongFavouritesCount(String songId, boolean shouldDecrement)** - Increments/decrements a Song's like count. You should not be able to decrement a song's likes below 0.

Profile Microservice

- The following requirements are implemented in the starter code for you, but please ensure that the following requirements are satisfied
    - Your database username and password **must** be "neo4j" and "1234" respectively
    - Your database **must** be connected to 7687
- Neo4j database schema **MUST** follow the schema below:
    - The nodes MUST be exactly:
        - "profile"
        - "song"
        - "playlist"
    - The relationships **MUST** be exactly
        - (:profile)-[:created]->(:playlist)
        - (profile)-[:follows]->(:profile)
        - (:playlist)-[:includes]->(:song)
    - Each "song" node must have a "songId" attribute
- DB Methods to Implement
    - **DbQueryStatus likeSong(String userName, String songId)** - Adds the songId to the Profile's favourites list
    - **DbQueryStatus unlikeSong(String userName, String songId)** - Removes the songId from the Profile's favourites list
    - **DbQueryStatus deleteSongFromDb(String songId)** - Deletes the songId from all favourites playlist
    - **DbQueryStatus createUserProfile(String userName, String fullName, String password)** - Creates the Profile and adds it to the Neo4J DB

- **DbQueryStatus followFriend(String userName, String frndUserName)** - Allows the userName Profile to follow the frndUserName Profile
- **DbQueryStatus unfollowFriend(String userName, String frndUserName)** - Allows the userName Profile to unfollow the frndUserName Profile
- **DbQueryStatus getAllSongFriendsLike(String userName)** - Returns a list of all the songIds for all of userName Profile's friends

Git Usage

You are required to use the Git flow for this assignment. You will be evaluated on proper usage of Git flow and commit messages

Code Style

You will be required to appropriately use variable naming conventions, file naming conventions in both microservices. You will also be required to comment your routes and function signatures appropriately

TESTING YOUR CODE

Run your services. See *Environment Setup* for how to run each service

CLI

From the command line the best way to test your endpoints is using curl.

```
curl -X POST http://localhost:PORT/routeNameHere/ --data \
     '{ "key": "value", "other": "thing" }'
```

- The -X flag specifies the REST action you wish to use
- The --data flag specifies the body of the request if one is required

Postman

Postman is a very nice tool to use to test your endpoints

Sending a Request and Reading a Response

- Open up Postman and select the type of request you would like to send (1)
- Enter your request URL (2)
- Open up the request body tab (3) and select x-www-form-urlencoded (4) if you are sending body parameters

- Your response data will be shown in the body area (5) when you click the Send button