Assignment 2

SQL Programming

Due date: 20/12/2018

Submission is in pairs only.
T.A. in charge – Moshe Sebag.
Any question about the assignment should be sent to maman.23636.hw2@gmail.com.

1. Introduction

You are about to take a lead part in the development of “Technify”, a (fictional) Technion's digital music application. In Technify users can listen to the music they like, follow playlists and more. Technify is a smart service that also recommends users new music according to the songs they have listened to.

Your mission is to design the database and implement the data access layer of the system. Typically, the data access layer facilitates the interaction of other components of the system with the database by providing a simplified API that carries out a predefined desired set of operations. A function in the API may receive as input arguments business objects. These are regular Java classes that hold a special semantic meaning in the context of the application (typically, all other system components are familiar with them). The ZIP file that accompanies this document contains the set of business objects to be considered in the assignment, as well as the full (unimplemented) API. Your job is to write code into these functions to fulfil their purpose as described below.

Please notice:

1. The database design is your responsibility. You can create and modify it as you see fit. You will be given grade for your database design, so bad and inefficient design will suffer from points reduction.

2. Every calculation involving the data, like filtering and sorting, must be done by querying the database. You are prohibited from performing any calculations on the data using Java. Additionally, when writing your queries, you should only use the material learned in class.

3. It is recommended to go over the relevant classes Java files and understand their usage.

4. All provided business classes are implemented with default constructor and getter\setter to each field.
2. Business Objects

In this section we describe the business objects to be considered in the assignment.

**User**

Attributes:

<table>
<thead>
<tr>
<th>Description</th>
<th>Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>User ID</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>User name</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>Country</td>
<td>String</td>
<td>The user's country</td>
</tr>
<tr>
<td>Premium</td>
<td>Boolean</td>
<td>A Boolean attribute which indicates whether the user is premium user (=true) or not (=false)</td>
</tr>
</tbody>
</table>

Constraints:

1. IDs are unique across all users.
2. IDs are positive (>0) integers.
3. Name, Country and Premium are not optional (not null).

Notes:

1. In the class User you will find the static function badUser() that returns an invalid user.

**Song**

Attributes:

<table>
<thead>
<tr>
<th>Description</th>
<th>Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Song ID</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>Song name</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>genre</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>Country</td>
<td>String</td>
<td>The country the song was published in</td>
</tr>
<tr>
<td>Play count</td>
<td>Integer</td>
<td>The count of times users have listened to the song</td>
</tr>
</tbody>
</table>

Constraints:

1. IDs are unique across all songs.
2. IDs are positive (>0) integers.
3. Song name, and genre are not optional (not null).
4. Every new song play count is initialized with 0.
5. Play count can't be negative.

Notes:

1. In the class Song you will find the static function badSong() that returns an invalid song.
Playlist

Attributes:

<table>
<thead>
<tr>
<th>Description</th>
<th>Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Playlist ID</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>genre</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>A short text that gives some information about the playlist.</td>
</tr>
</tbody>
</table>

Constraints:

1. IDs are unique across all playlists.
2. IDs are positive (>0) integers.
3. Genre and description are not optional (not null).

Notes:

1. In the class Playlist you will find the static function badPlaylist() that returns an invalid playlist.

3. API

3.1 Return Type

For the return value of the API functions, we have defined the following enum type:

ReturnValue (enum):

- OK
- NOT_EXISTS
- ALREADY_EXISTS
- ERROR
- BAD_PARAMS

3.2 CRUD API

This part handles on the CRUD - Create, Read, Update, Delete operations of the business objects in the database. Implementing this part correctly will cause easier implementations of the more advances APIs

ReturnValue addUser(User user)

Adds a user to the database.

input: user to be added.

output: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters
* ALREADY_EXISTS if user already exists
* ERROR in case of database error

**User getUserProfile(Integer userId)**

Returns the user profile by the given id.
input: user id.
output: The user profile (a User object) in case the user exists. BadUser otherwise.

**ReturnValue deleteUser(User user)**

Deletes a user from the database.
Deleting a user will cause him/her to stop follow any playlist.
input: user to be deleted.
output: ReturnValue with the following conditions:
* OK in case of success
* NOT_EXISTS if user does not exist
* ERROR in case of database error

**ReturnValue updateUserToPremium(Integer userId)**

Updates a user premium status to be premium.
input: user id. Assume userId is not null.
output: ReturnValue with the following conditions:
* OK in case of success
* ALREADY_EXISTS if user is already premium (use getUserProfile for this)
* NOT_EXISTS if user does not exist
* ERROR in case of database error

**ReturnValue updateUserToNotPremium(Integer userId)**

Updates a user premium status to be not premium.
input: user id. Assume userId is not null.
output: ReturnValue with the following conditions:
* OK in case of success
* ALREADY_EXISTS if user is already not premium (use getUserProfile for this)
* NOT_EXISTS if user does not exist
* ERROR in case of database error

**ReturnValue addSong(Song song)**

Adds a song to the database.
input: song to be added.
output: ReturnValue with the following conditions:
* OK in case of success
* BAD_PARAMS in case of illegal parameters
* ALREADY_EXISTS if user already exists
* ERROR in case of database error

Notice that a new song has 0 plays so its playCount initial value is 0.

**Song getSong(Integer songId)**

Returns the song which the given id belongs to
input: song id
output: The song in case the song exists. BadSong otherwise.

**ReturnValue deleteSong(Song song)**

Deletes a song from the database.
Deleting a song will delete it from any playlist.
input: song to be deleted
output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if song does not exist
* ERROR in case of database error

**ReturnValue updateSongName(Song song)**

Updates a song name.
input: updated song
output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if user does not exist
* BAD_PARAMS in case of illegal parameters (null name)
* ERROR in case of database error

**ReturnValue addPlaylist(Playlist playlist)**

Adds a new playlist to the database.
input: playlist to be added.
output: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters
* ALREADY_EXISTS if user already exists
* ERROR in case of database error
Playlist getPlaylist(Integer playlistId)

Returns the playlist which the given id belongs to
input: playlist id.
output: The playlist in case the playlist exists. BadPlaylist otherwise.

ReturnValue deletePlaylist(Playlist playlist)

Deletes a playlist from the database.
notice: this function DOES NOT delete songs from the database.
input: playlist to be deleted.
output: ReturnValue with the following conditions:
* OK in case of success
* NOT.Exists if playlist does not exist
* ERROR in case of database error

ReturnValue updatePlaylist(Song song)

Updates the playlist's description.
input: updated playlist.
output: ReturnValue with the following conditions:
* OK in case of success
* NOT.Exists if playlist does not exist
* BAD_PARAMS in case of illegal parameters (null description)
* ERROR in case of database error

3.3 Basic API

ReturnValue songPlay(Integer songId, Integer times)

The song which has songId had value of times more plays on Technify.
The query should add "times" to the song current play count.

Notice: times might be negative. (i.e. the play count can decrease).

input: The song that was listened id and the number of times it was played.
output: ReturnValue with the following conditions:
* OK in case of success
* NOT.Exists if song does not exist
* ERROR in case of database error
* BED_PARAMS if the new play count would be negative.
ReturnValue addSongToPlaylist(Integer songid, Integer playlistId)

Adds the song of the given song id to the playlist of the given playlist id. This action is possible only if the song and the playlist belong to the same genre.
Input: song id and playlist id.
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if the song or the playlist do not exist
* BAD_PARAMS in case the song and the playlist doesn't have the same genre OR if the song or the playlist do not exist
* ALREADY_EXISTS if the song already belongs to the given playlist
* ERROR in case of database error

ReturnValue removeSongFromPlaylist(Integer songid, Integer playlistId)

Remove the song of the given song id from the playlist of the given playlist id. Notice: This action DOES NOT remove the song from the database.
Input: song id and playlist id.
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if the song or the playlist do not exist or if the song isn’t in the playlist
* ERROR in case of database error

ReturnValue followPlaylist(Integer userId, Integer playlistId)

Marks that the user follows the playlist.
Input: id of following user, the playlist id the user started to follow
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if the user or the playlist do not exist
* ALREADY_EXISTS if the user already follows the playlist
* ERROR in case of database error

ReturnValue stopFollowPlaylist(Integer userId, Integer playlistId)

Marks that the user does not follow the playlist.
Input: id of following user, the playlist id the user stopped to follow.
Output: ReturnValue with the following conditions:
* OK in case of success
* NOT_EXISTS if the user or the playlist do not exist, or the user doesn't follow it.
* ERROR in case of database error

**Integer getPlaylistTotalPlayCount(Integer playlistId)**
Return the sum of play counts for all the songs in the playlist.
Input: playlist id
Output:
* the sum of play counts for all the songs in the playlist, given a valid playlist id
* 0 in any other case

**Integer getPlaylistFollowersCount(Integer playlistId)**
Return the number of users that follow the playlist.
Input: playlist id
Output:
* the amount of users that follow the playlist, given a valid playlist id
* 0 in any other case

**String getMostPopularSong()**
Returns the name of the song that belongs to the highest number of playlists. In case of equality return the one with the highest id (between those which are most popular).
Input: None
Output:
*String with the name of the most popular song
*The string "No songs" if all playlists are empty
*null in any other case

**Integer getMostPopularPlaylist()**
Returns the playlist that has the highest sum of play counts for all its songs ids. In case of equality return the one with highest id (between those which are most popular).
Input: None
Output:
*Integer with the id of the most popular playlist.
*0 in any other case
3.4 Advanced API

Notice: In any of the following functions, if you are required to return list in size X but there are less than X results, return shorter list which contains the relevant results.

**ArrayList<Integer> hottestPlaylistsOnTechnify()**

Return a list of 10 playlists' ids with the highest rating on Technify (in descending order).
Where rating is calculated as follows:

\[
\text{rating} = \frac{\text{SumPlayCounts}}{\text{NumOfSongs}}
\]

SumPlayCounts = sum of all play counts of songs in the playlist.
NumOfSongs = the amount of songs that belongs to the playlist.

In case of equality order by id in ascending order.

Notice: to avoid dividing by 0, Empty playlists has no rating and therefore can't be returned by this function.

Input: none

Output:

*ArrayList with the relevant playlists id (if there are less than 10 playlists the ArrayList will be smaller than 10)
*Empty ArrayList in any other case

**ArrayList<Integer> getSimilarUsers(Integer userId)**

Return a list of 10 "similar users" of the user with the given id.
Where similar users" defined as users who follows at least (>=) 75% of the playlists the given user follows. Notice that one cannot be a similar user of himself.
Ordered by id in ascending order.

Input: user id

Output:

*ArrayList with the relevant users ids
*Empty ArrayList in any other case

**ArrayList<Integer> getPlaylistRecommendation (Integer userId)**

Returns the up to 5 most followed playlists by similar users (as defined in previous function), that the user does not follow (in descending order) ids. In case of equality order by id in ascending order. In case of 0 similar users return empty list.
Note: the playlist's followers count is based only on the similar users.
input: user id
Output:
*ArrayList with the relevant playlists id if the user is premium
*Empty ArrayList in any other case

ArrayList<Integer> getTopCountryPlaylists(Integer userId)

Returns the ArrayList of playlists ids that have the highest sum of songs' play counts (in descending order) with a demand that each of those playlists has at least one song from the user's country that belongs to them. In case of equality order by id in ascending order. Returns the top 10 of them. This function should work only for premium user.

input: user id
Output:
*ArrayList with the relevant playlists id if the user is premium
*Empty ArrayList in any other case

ArrayList<Integer> getSongsRecommendationByGenre(Integer userId, String genre)

Returns the up to 10 most played songs (in descending order) ids that belong to the genre (from the input) and do not belong to any playlist the user follows. In case of equality order by id in ascending order.

input: user id
Output:
*ArrayList with the relevant playlists songs id if the user is premium
*Empty ArrayList in any other case

4. Database

5.1 Basic Database functions

In addition to the above, you should also implement the following functions:

void createTables()
Creates the tables and views for the solution.
void clearTables()
Clears the tables for the solution (leaves tables in place but without any data).

void dropTables()
Drops the tables and views from DB.

5.2 Connecting to the Database using JDBC

Each of you should download, install and run a local PostgreSQL server from https://www.postgresql.org. You may find this guide helpful.

To connect to that server, we have implement for you the DBConnector class that creates a Connection instance that you should work with in order to interact with the database.

For establishing successfully a connection with the database, you should provide a proper configuration file to be located under the folder `src/main/resources` of the project. A default configuration file has already been provided to you under the name Config.properties. Its content is the following:

database=jdbc:postgresql://localhost:5432/cs236363
user=java
password=12345678

Make sure that port (default: 5432), database name (default: cs236363), username (default: java), and password (default: 12345678) are those you specified when setting up the database.

In order to get the Connection instance, you should invoke the static function DBConnector.getConnection(). To submit a query to database, do the following:

1. Prepare your query by invoking connection.prepareStatement(<your query>). This function returns a PreparedStatement instance.

2. Invoke the function execute() or executeQuery() from the PreparedStatement instance.

The DBConnector class also implements the following functions which you may find helpful:

1. printTableSchemas() – prints the schemas of the tables in the database.

2. printSchema(ResultSet) - prints the schema of the given ResultSet.

3. printResults(ResultSet) - prints the underlying data of the given ResultSet.
5.3 SQL Exceptions

When preparing or executing a query, an SQL Exception might be thrown. It is thus needed to use the try/catch mechanism in order to handle the exception. For your convenience, the PostgreSQLErrorCodes enum type has been provided to you. It captures the error codes that can be returned by the database due to error or inappropriate use. The codes are listed here:

\[
\text{INTEGRITY\_CONSTRAINT\_VIOLATION (23000),}
\text{RESTRICT\_VIOLATION (23001),}
\text{NOT\_NULL\_VIOLATION (23502),}
\text{FOREIGN\_KEY\_VIOLATION (23503),}
\text{UNIQUE\_VIOLATION (23505),}
\text{CHECK\_VIOLATION (23514)};
\]

To check the returned error code, the following code should be used inside the catch block: (here we check whether the error code CHECK\_VIOLATION has been returned)

```java
if(Integer.valueOf(e.getSQLState()) == PostgreSQLErrorCodes.CHECK_VIOLATION.getValue())
{
    //do something
}
```

Notice you can print more details about your errors using:

```java
catch (SQLException e) {
    e.printStackTrace();
}
```

**Tips**

1. Create auxiliary functions that convert a record of ResultSet to an instance of the corresponding business object.

2. Use the enum type PostgreSQLErrorCodes. It is HIGHLY recommended to use the exceptions mechanism to validate input, rather than use Java’s “if else”.

3. Devise a convenient database design for you to work with.

4. Before you start programming, think which Views you should define to avoid code duplication and make your queries readable and maintainable.

5. (Think which sub-queries appear in multiple queries).

6. Use the constraints mechanisms taught in class in order to maintain a consistent database.
Use the enum type PostgreSQLErrorCodes in case of violation of the given constraints.

7. Remember - you are also graded on your database design (tables, views).

8. Please review and run example.java for additional information (ArrayList, String) and implementation methods.

**Submission**

Please submit the following:

1. The file solution.java where all of your code should be written in.

2. The file `<id1>_<id2>.pdf` in which you explain in detail your database design and the implantation of the API.

3. The file submitters.txt that consists of the following two lines:
   `<id1><email1>`
   `<id2><email2>

   For example,
   `<123456789>@lib@campus.technion.ac.il`
   `<987654321>@sha@campus.technion.ac.il`

Note that you can use the unit tests framework (JUnit) as explained in details in the PDF about installing IDE, but no unit test should be submitted.

**Good Luck!**