Assignment 2

Due date: 24/12/2017

Submission is in pairs. Please contact 236363cs@gmail.com for any question you might have.

1. Introduction

You are about to take a lead part in the development of "Tech-flix", a (fictional) Technion's content and movies streaming application. Similar to other media streaming applications, in Tech-flix users can view, rate and get recommendations about movies.

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 in order for them 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.

Viewer

- Attributes:

Description	Type	Comments
Viewer ID	Integer	
Name	String	

- Constraints:
  1. IDs are unique across all viewers.
  2. IDs are positive (>0) integers
  3. Name is not optional (not null)

- Notes:
  1. In the class Viewer you will find the static function badViewer() that returns an invalid viewer.

Movie

- Attributes:

Description	Type	Comments
Movie ID	Integer	
Name	String	The name (title) of the movie
Description	String	Short description of the movie.

- Constraints:
  1. IDs are unique across all movies.
  2. IDs are positive (>0) integers.
  3. Name and Description are not optional.

  Notes:

  1. In the class Post you will find the static function badMovie() that returns an invalid movie.

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*

The is no hierarchy between the values and you should return the value that corresponds with the state of your database.

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 createViewer(Viewer viewer)
Adds a viewer to the database.
input: viewer to be added
output: ReturnValue with the following conditions:
** OK in case of success*
** BAD_PARAMS in case of illegal parameters*
** ALREADY_EXISTS if student already exists*
** ERROR in case of database error*

Viewer getViewer(Integer viewerId)
Returns the viewer profile by the given id
input: viewer id
output: The viewer profile in case the viewer exists. BadViewer otherwise

ReturnValue deleteViewer(Viewer viewer)
Deletes a viewer from the database.
Deleting a viewer will delete their views and likes history
input: viewer to be deleted
output: ReturnValue with the following conditions:

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

ReturnValue updateViewer(Viewer viewer)
Updates a viewer name to the new given value.
input: updated viewer
output: ReturnValue with the following conditions:
** OK in case of success*
** NOT_EXISTS if viewer does not exist (does not matter if name is null or not)*
** BAD_PARAMS in case of illegal parameters (null name)*
** ERROR in case of database error*

ReturnValue addMovie(Movie movie)
Adds a movie to the database
input: movie to be added
output: ReturnValue with the following conditions:
** OK in case of success*
** BAD_PARAMS in case of illegal parameters*
** ALREADY_EXISTS if movie already exists*
** ERROR in case of database error*

Post getMovie(Integer movieId)
returns the movie by given id
input: movie id
output: Movie if the post exists. BadMovie otherwise

ReturnValue deleteMovie(Movie movie)
Deletes a movie from the database
input: movie to be deleted
output: ReturnValue with the following conditions:
** OK in case of success*
** NOT_EXISTS if movie does not exist*
** ERROR in case of database error*

ReturnValue updateMovie(Movie movie)
Updates a movie's description
input: updated movie
output: ReturnValue with the following conditions:
** OK in case of success*

** NOT_EXISTS if movie does not exist (does not matter if description is null)*
** BAD_PARAMS in case of illegal parameters (null description)*
** ERROR in case of database error*

3.3 Basic API

ReturnValue addView(Integer viewerId, Integer movieId)
Marks a movie as watched by the viewer
Input: viewer id, movie id
Output: ReturnValue with the following conditions:
** OK in case of success*
** NOT_EXISTS if viewer or movie do not exist*
**ALREADY_EXISTS if the viewer already watched the movie*
** ERROR in case of database error*

ReturnValue removeView(Integer viewerId, Integer movieId)
Marks a movie as unwatched by the viewer
Input: viewer id, movie id
Output: ReturnValue with the following conditions:
** OK in case of success*
** NOT_EXISTS if viewer or movie do not exist, or the viewer hasn't watched the movie yet*
** ERROR in case of database error*

Integer getMovieViewCount(Integer movieId)
Return the number of viewers who watched the movie
Input: movie id
Output:
** the amount of viewers who watched the movie, given a valid movie id*
** 0 in any other case*

ReturnValue addMovieRating(Integer viewerId, Integer movieId, MovieRating rating)
Gives the movie a rating by the viewer
Input: viewer id, movie id, movie rating (enum {LIKE, DISLIKE})
Note: A movie can rating can be updated , to both different rating, or the same rating (giving a movie the same rating will NOT increase this rating count over 1)
Output: ReturnValue with the following conditions:
** OK in case of success*
** NOT_EXISTS if viewer or movie do not exist, or the viewer hasn't watched the movie*
** ERROR in case of database error*

Hint: you might want to use UPSERT (insert or update) in this function. Please read :
https://www.postgresql.org/docs/9.6/static/sql-insert.html
https://wiki.postgresql.org/wiki/UPSERT

ReturnValue removeMovieRating(Integer viewerId, Integer movieId)
Removes the rating of the movie, given by the viewer
Input: viewer id, movie id
Output: ReturnValue with the following conditions:
** OK in case of success*
** NOT_EXISTS if viewer or movie do not exist, or the viewer hasn't rated the movie yet*
** ERROR in case of database error*

Integer getMovieLikesCount(Integer movieId)
Return the number of viewers who liked the movie
Input: movie id
Output:
** the amount of viewers who liked the movie, given a valid movie id*
** 0 in any other case*

Integer getMovieDisikesCount(Integer movieId)
Return the number of viewers who disliked the movie
Input: movie id
Output:
** the amount of viewers who disliked the movie, given a valid movie id*
** 0 in any other case*

3.4 Advanced API

ArrayList<Integer> getSimilarViewers(Integer viewerId)
Return a list of viewers id who watched at least (>=) 75% of the movies the given viewer watched. Ordered by id in ascending order.
Input: viewer id
Output:
**ArrayList with the relevant viewers id*
**Empty ArrayList in any other case*

ArrayList<Integer> mostInfluencingViewers()
Return the top 10 viewers id with the highest views and rating count ordered by views count and then rating count (both in descending order). In case of equality order by id in ascending order
input: none
Output:
**ArrayList with the relevant viewers id*
**Empty ArrayList in any other case*

ArrayList<Integer> getMoviesRecommendations(Integer viewerId)
Returns the top 10 liked movies (in descending order) id by similar viewers, and the viewer did not watch yet. In case of equality order by id in ascending order. In case of 0 similar viewers return empty list.
Note: the likes count is based only on the similar viewers likes
input: viewer id
Output:
**ArrayList with the relevant movies id*
**Empty ArrayList in any other case*

ArrayList<Integer> getConditionalRecommendations(Integer viewerId, Integer MovieId)
Returns the top 10 liked movies (in descending order) by similar rankers that the current viewer did not watch. In case of equality order by id in ascending order. In case of 0 similar rankers return empty list.
Similar rankers - A subset of similar viewers to the current viewer who ranked the current movie with the same rank as the current viewer.
Note: the likes count is based only on the similar rankers likes
input:current viewer id,current movie id
Output:
**ArrayList with the relevant movies id*
**Empty ArrayList in any other case*

4. Database

4.1 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.

4.2 Connecting to the Database using JDBC

Each of you should download, install and run a local PosgtreSQL 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 as 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: cd236363), 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.

4.3 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:

NOT_NULL_VIOLATION (23502),
FOREIGN_KEY_VIOLATION(23503),
UNIQUE_VIOLATION(23505),
CHECK_VIOLIATION (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_VIOLIATION* has been returned)

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

Tips

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

2. Use the enum type PostgreSQLErrorCodes to evalute the server response on your queries. It is HIGHLY recommended to use the exceptions mechanism to validate input, rather than use Java's "if else".

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

4. 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.
5. Use views whenever possible to allow your queries to be readable and maintainable.
6. Remember - you are also graded on your database design (tables, views etc....).
7. 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. You don't have to explain obvious actions but do expand on the advanced API and your overall design and non trivial actions.

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

Note the unit test should not be submitted.

Good Luck!