# Assignment 2

## SQL Programming

**Due date: 22.12.21**

Submission is in pairs.
Please use hw2's piazza forum for any question you may have.

## 1. Introduction

You are about to take a lead part in the development of the **"FStats"** database, a website that holds information about football matches, players, and more.

In **FStats**, users with admin privileges (you), can add a match, player, stadium and so on.

**FStats** is a smart service that gives you statistics about football in general.

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 business objects as Input arguments. These are regular Python classes that hold 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 implement these functions so that they fulfill their purpose as described below.

**Please note:**

1. The database design is your responsibility. You may create and modify it as you see fit. You will be graded 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 Python. Furthermore, you cannot define your own classes, your code must be contained in the functions given, except for the case of defining basic functions to avoid code duplication. Additionally, when writing your queries, you may only use the material learned in class.

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

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

5. You may not use more than **one** SQL query in each function implementation, not including views. Create/Drop/Clear functions are not included!

## 2. Business Objects

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

### Team

Referred as a simple positive unique integer.

### Match

Attributes:

| Description | Type | Comments |
|---|---|---|
| Match's ID | Int | The match id. |
| Competition | String | International or domestic. |
| Home team ID | Int | Home team's id. |
| Away team ID | Int | Away team's id. |

Constraints:

1. Matches' IDs are unique across all matches ~~and teams~~ (and positive).
2. Competition values are either International or Domestic and home and away can not be the same id.
3. All attributes are not optional (not null).

Notes:

1. In the class Match you will find the static function badMatch() that returns an invalid Match.

**Player**

Attributes:

| Description | Type | Comments |
|---|---|---|
| Player ID | Int | The ID of the player. |
| Team ID | Int | The ID of the team the player plays in. |
| Age | Int | The player's age. |
| Height | Int | The player's height in cm. |
| Preferred foot | String | Left or right. |

Constraints:

1. Players' IDs are unique across all players ~~and teams~~.
2. IDs, Age and Height are positive (>0) integers.
3. Preferred foot values are either 'Left' or 'Right'.
4. All attributes are not optional (not null).

Notes:

1. In the class Player you will find the static function bad Player() that returns an invalid Player.

**Stadium**

Attributes:

| Description | Type | Comments |
|---|---|---|
| Stadium ID | Int | The ID of the stadium. |
| Capacity | Int | The capacity of the stadium. |
| Belong to | Int | The team's ID this stadium belongs to. |

Constraints:

1. IDs are unique across all Stadiums.
2. IDs and Capacity are positive (>0) integers.
3. All attributes except Belong to are not optional (not null).
4. There can not be 2 rows with the same 'Belong to' beside NULL.

Notes:

1. In the class Stadium you will find the static function bad Stadium() that returns an invalid Stadium.

# 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

In case of conflicting return values, return the one that appears first on each section.

## 3.2 CRUD API

This part handles the CRUD - Create, Read, Update and Delete operations of the business objects in the database. Implementing this part correctly will lead to easier implementations of the more advanced APIs.

Python's equivalent to NULL is None.
You can assume the arguments to the function will not be None, the inner attributes of the argument might consist of None.

**ReturnValue addTeam(Int teamID)**

Adds a **team** to the database.
**Input**: teamID to be added.
**Output**: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters.
* ALREADY_EXISTS if a team with the same ID already exists.
* ERROR in case of a database error

**ReturnValue addMatch(Match match)**

Adds a **match** to the database.
**Input**: match to be added.
**Output**: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters.
* ALREADY_EXISTS if a match with the same ID already exists.
* ERROR in case of a database error

Note: you can assume we will not test an addition of a match with ID that already exists with teamID that does not exist.

**Match getMatchProfile(Int matchID)**

Returns the match profile of
**matchID**.

**Input**: Match ID.
**Output**: The match profile (a match object) in case the match exists. BadMatch() otherwise.

**ReturnValue deleteMatch (Match match)**

Deletes a **match** from the database.
Deleting a **match** will delete it from everywhere as if it never existed.
**Input**: match to be deleted.
**Output**: ReturnValue with the following conditions:

* OK in case of success ~~or if match does not exist (ID wise).~~
* NOT_EXISTS if match does not exist.
* ERROR in case of a database error

**ReturnValue addPlayer (Player player)**

Adds a player to the database.
**Input**: player to be added.
**Output**: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters
* ALREADY_EXISTS if a player with the same ID already exists.
* ERROR in case of a database error

Note: you can assume we will not test an addition of a player with ID that already exists with teamID that does not exist.

**Player getPlayerProfile (Int playerID)**

Returns the player with playerID as its id.

**Input**: player id.
**Output**: The player with playerID if exists. BadPlayer() otherwise.

**ReturnValue deletePlayer(Player player)**

Deletes a player from the database.
Deleting a player will delete it from everywhere as if the player never existed.
**Input**: player ~~ID~~ to be deleted.
**Output**: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if player does not exist.
* ERROR in case of a database error

**ReturnValue addStadium (Stadium stadium)**

Adds a Stadium to the database.
**Input**: Stadium to be added.
**Output**: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters
* ALREADY_EXISTS if a Stadium with the same ID already exists or the team already owns a stadium.
* ERROR in case of a database error

Note: you can assume we will not test an addition of a stadium with ID that already exists with teamID that does not exist.

**Stadium getStadiumProfile (Int StadiumID)**

Returns the Stadium with StadiumID as its id.

**Input**: Stadium id.
**Output**: The Stadium with StadiumID if exists. BadStadium() otherwise.

**ReturnValue deleteStadium(Stadium stadium)**

Deletes a Stadium from the database.
Deleting a Stadium will delete it from <u>everywhere</u> as if it never existed.
**Input**: Stadium~~ID~~ to be deleted.
**Output**: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if Stadium does not exist.
* ERROR in case of a database error

You may not use getProfile() functions in your implementation, all must be done via SQL.

## 3.3 Basic API

**ReturnValue playerScoredInMatch(Match match, Player player, int Amount)**

The **player** has scored **Amount** goals in **match**.
**Input**: The player, match and number of goals.
**Output**: ReturnValue with the following conditions:

* OK in case of success.
* BAD_PARAMS in case amount is not positive.
* NOT_EXISTS if player/match does not exist.
* ALREADY_EXISTS if the player already scored in this match.
* ERROR in case of a database error.

Note: you can assume player is a part of a team playing in match.

**ReturnValue playerDidntScoreInMatch(Match match, Player player)**

The **player** did not score in **match**, if the player did, delete its record.
**Input**: The player and match.
**Output**: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if match/player does not exist or player did not already score in match.
* ERROR in case of a database error.

**ReturnValue matchInStadium(Match match, Stadium stadium, Int attendance)**

The **match** is taking place in **stadium** with **attendance** spectators.
**Input**: The match, stadium and number of spectators.
**Output**: ReturnValue with the following conditions:

* OK in case of success.
* BAD_PARAMS in case attendance is negative, assume attendance<stadium's capacity.
* ALREADY_EXISTS if the match already taking place in any stadium.
* NOT_EXISTS if match/stadium does not exist.
* ERROR in case of a database error.

**ReturnValue matchNotInStadium(Match match, Stadium stadium)**

The **match** is no longer taking place in **stadium**, even if it already has a record.
**Input**: The match and stadium.
**Output**: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if match/stadium does not exist or match does not take place in stadium.
* ERROR in case of a database error.

**Float averageAttendanceInStadium(Int stadiumID)**

Returns the average attendance in stadium with **stadiumID**.
**Input**: stadium's ID.
**Output**:

* The average size in case of success.
* 0 in case of division by 0 or stadiumID does not exist, -1 in case of other error.

**Int stadiumTotalGoals**(**Int stadiumID)**

Returns the total amount of goals scored in stadium with **stadiumID**.
**Input**: stadiumID of the requested stadium.
**Output**:

* The sum in case of success.
* 0 if the stadium does not exist,-1 in case of an error.

**Bool playerIsWinner(Int playerID, Int matchID)**

Returns true if the player with **playerID** scored at least half of the goals in the match with **matchID.**

**For example:**

The score was 4-2 for the player's team and the player scored 3 – Winner.

**Input**: player's ID and match's ID.
**Output**:

* True if **playerID** is a winner, False otherwise.
* False if the **playerID** or **mathcID** does not exist or no goals were scored in this match or in case of an error.

**List<Int> getActiveTallTeams()**
Returns a List (up to size 5) of active teams' IDs that have at least 2 players over the height of 190cm. Active team is a team who played at least 1 match at home or away (a record in Matches).
**The list should be ordered by IDs in descending order.**
**Input**: None.
**Output**:

* List with the teams' IDs.
* Empty List in any other case.

**List<Int> getActiveTallRichTeams()**
Returns a List (up to size 5) of active, rich teams' IDs that have at least 2 players over the height of 190cm. Active team is a team who played at least 1 match at home or away (a record in Matches). Rich team is a team who owns a stadium of size >55,000.
**The list should be ordered by IDs in ascending order.**
**Input**: None.
**Output**:

* List with the teams' IDs.
* Empty List in any other case.

**List<Int> popularTeams()**
Returns a List (up to size 10) of teams' IDs that in every single game they played as 'home team' they had more than 40,000 attendance.
**The list should be ordered by IDs in descending order.**
Note: if a match did not take place in a specific stadium, treat it as if it had less than 40,000 attendance.

**Input**: None.
**Output:**

* List with the teams' IDs.
* Empty List in any other case.

## 3.4 Advanced API

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

**List<Int> getMostAttractiveStadiums()**

Returns a list containing attractive stadiums' IDs .

The most attractive stadium is the stadium in which the most goals were scored in its matches, and so on.

**The list should be ordered by attractiveness in descending order, in case of equality order by ID in ascending order.**

**Input**: None

**Output**:

*List with the stadiums' IDs.
*Empty List in any other case.

**List<Int> mostGoalsForTeam(Int teamID)**

Returns a list of up to 5 players' IDs who scored the most goals for the team with teamID.

**The list should be ordered by:**

- **Main sort by the number of goals in descending order.**
- **Secondary sort by players' IDs in descending order.**

**Input**: The teamID of the team in question.

**Output**:

*List with the players' IDs.
*Empty List in any other case.

Note: only player who play for this team can be in this list.

**List<Int> getClosePlayers (Int playerID)**

Returns a list of the 10 "close players" to the player with **playerID**.
Close players are defined as players who scored in at least (>=) 50% of the matches the player with **playerID** did. Note that a player cannot be a close player of itself.
**The list should be ordered by IDs in ascending order.**

**Input**: The ID of a player.

**Output**:

*List with the players' IDs that meet the conditions described above.
*Empty List in any other case.

Note: players can be close in an empty way (player in question did not score in any match).

# 4. Database

## 6.1 Basic Database functions

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

**void createTables()**
Creates the tables and views for your solution.

**void clearTables()**
Clears the tables for your solution (leaves tables in place but without any data).

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

Make sure to implement them correctly.

## 6.2 Connecting to the Database using Python

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

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

For establishing successful connection with the database, you should provide a proper configuration file to be located under the folder Utility of the project. A default configuration file has already been provided to you under the name database.ini. Its content is the following:

```
[postgresql]
host=localhost
database=cs236363
user=username
password=password
port=5432
```

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

To get the Connection instance, you should create an object using
conn = Connector.DBConnector() (after importing "import Utility.DBConnector as Connector" as in Example.py). To submit a query to your database, simply perform conn.execute("query here"). This will return a tuple of (number of rows affected, results in case of SELECT).
Make sure to close your session using .close().

### 6.3 SQL Exceptions

When preparing or executing a query, an SQL Exception might be thrown. It is thus needed to use the try/catch (try/except in python) mechanism to handle the exception. For your convenience, the DatabaseException 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 except block: (here we check whether the error code *CHECK_VIOLIATION* has been returned)

```
except DatabaseException.CHECK_VIOLATION as e:
    #  Do stuff
```

Notice you can print more details about your errors using print(e).

## Tips

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

2. Use the enum type DatabaseException. It is highly recommended to use the exceptions mechanism to validate Input, rather than use Python'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.

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

5. Use the constraints mechanisms taught in class to maintain a consistent database. Use the enum type DatabaseException in case of violation of the given constraints.

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

7. Please review and run Example.py for additional information and implementation methods.

8. AGAIN, USE VIEWS!

## Submission

Please submit the following:

A zip file named <id1>-<id2>.zip (for example 123456789-987654321.zip) that contains the following files:

1. The file Solution.py where all your code should be written in (your code will also go through dry exam).

2. The file <id1>_<id2>.pdf in which you explain in detail your database design and the implantation of the API (each function and view). Is it **NOT** required to draw a formal ERD but it is indeed important to explain every design decision and it is highly recommended to include a draw of the design (again, it is **NOT** required to draw a formal ERD).

3. The file <id1>_<id2>.txt with nothing inside.

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

Make sure that is the exact content of the zip with no extra files/directories and no typos by using:

'python check_submission.py  <id1>-<id2>.zip'

(script and zip in the same directory).

Any other type of submission will fail the automated tests and result in **0** on this assignment.

**You will not have an option to resubmit in that case!**

**Good Luck!**