Software documentation

Part 1: DATA:

Create and normalize the dataset:

The dataset is taken from Kaggle. It was 8 csv files, each one for each year of 2015-2022. Each csv file had all the column, creating duplicates and mess. At first, we add column of "season" to each csv (for the csv of 2015, we add to all the records '2015' in the c 'season' column), then we joined all the csv files together, and we could differentiate similar records from different csv files by the column season we added.

After creating the joined table that unites all the tables from all seasons of the dataset, we normalize the dataset. At first, we removed the unnecessary column, and then, splitting the table by NF discipline.

For example, player can have 8 records (2015-2022) in the dataset, and in all records, his id points on his name, but in each record, some of his details may change (like his overall increases or decreases), so we need to split 2 tables of players, one with the constant data (his name, his nationality and etc.) and his dynamic data (his overall, his salary…).

We did the same thing for the teams, and the leagues.

After all of that, we had 5 tables of:

1) Players
2) Players_season
3) Teams
4) Teams_season
5) Leagues

We will explain about each table in the next part.

Create the tables for the users:

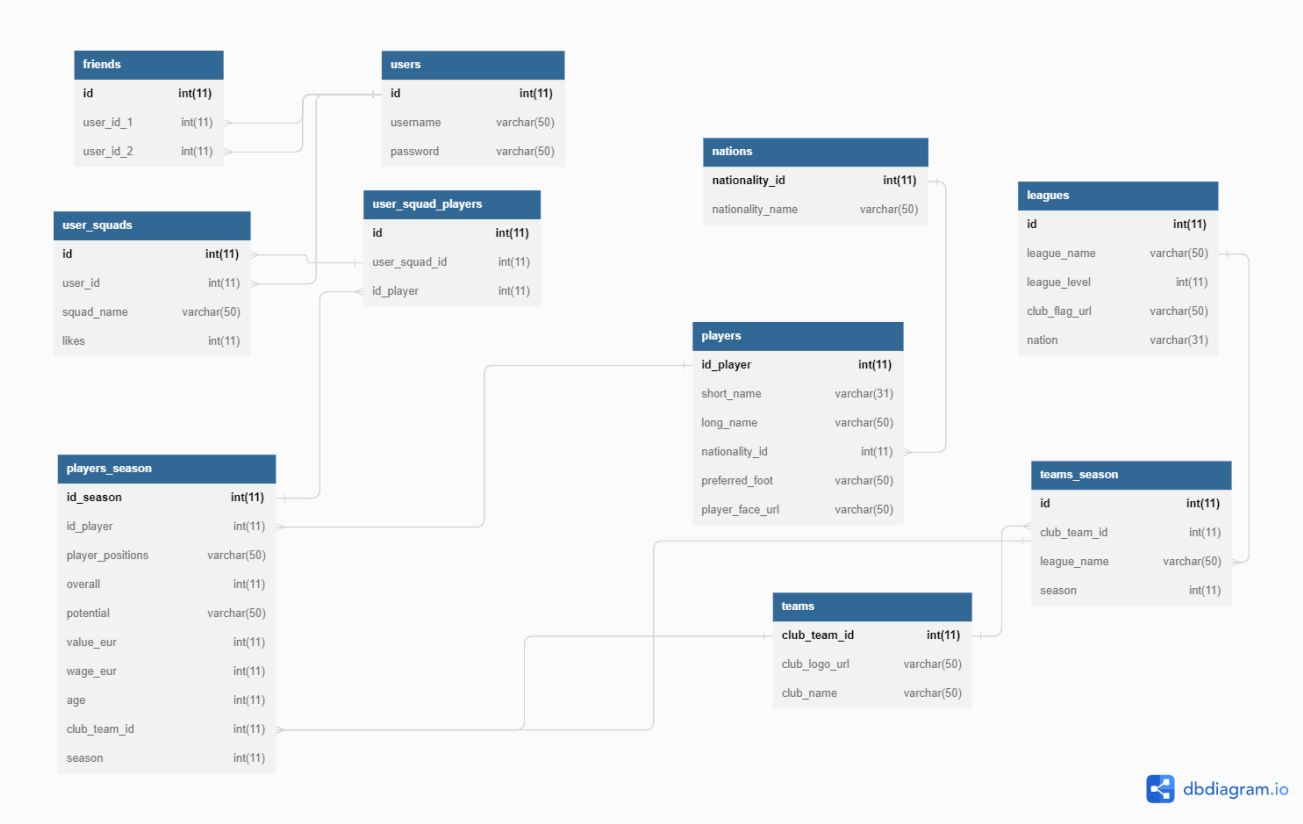
After we had our normalized dataset on our database in MySQL, we created the table – "users", with all the users we have in our database. To make the app more active, we created a script with the library "faker" that makes up 100 users with passwords:

```
from faker import Faker

import csv


fake = Faker()

# Create an empty list to store the data

data = []

# Append a header row to the list

data.append(['username', 'password'])

# Create an empty set to store the usernames

usernames = set()

# Generate 100 rows of fake data

for _ in range(100):

    # Use the Faker library to generate a fake username

    username = fake.user_name()

    # Check if the username already exists in the set

    while username in usernames:

        username = fake.user_name()

    # Add the username to the set

    usernames.add(username)

    # Use the Faker library to generate a fake password

    password = fake.password()

    # Append the data to the list

    data.append([username, password])

# Write the data to a CSV file

with open('users.csv', 'w', newline='') as file:

    writer = csv.writer(file)

    writer.writerows(data)
```

We added them to the database, and then created table – 'friends', to record friendship between 2 users in each line. Then we created

the tables: user_squads – where we record the squads user creates, and user_squad_players – where we record the players and the squads they are part of.

Part 2: schema:



As described earlier, we have "player"-"players_season" and "teams"-"teams_season", as the constant data presented in the "players/teams", and the dynamic data in the "players/teams_season". "Players" and "players_season" are connected by "id_player" (as "players_season.id_season" is for recording the "same" player but from different seasons), and "teams" and "teams_season" are connected by "club_team_id" (as "teams_season.id" is for recording the "same" team but from different seasons). "players_season" is connected to "teams" (as well to "teams_season", since the name of a team is constant) by "club_team_id". Leagues represented in one table and they are connected to "teams_season" by "league_name" (names are unique and the ids in Leagues are auto increment for internal use).

In this tables, the keys are:

Leagues: 1 – id. 2 – league_name.

Teams: 1 – club_team_id.

Teams_season: 1 – id, 2 – (club_team_id AND season).

Players: 1 – id_player.

Players_season: 1 – id_season. 2 – (id_player AND season).

As the tables "players/teams_season" record the same players more than once but from different season, the "constant" id (players.id_player or teams.club_team_id) with season refers to a specific record of player/team in a specific season.

At the users section, "friends" describe relationship between 2 users, as a record in this table contains 2 "user_id"s connected to the user_id in users. In "friends", the key Is "id" or both "user_id_1/2".

"user_squads" describes squads of users. Each record contains "id", "user_id" (that refers to "users" by "user_squads.user_id" ←→ "users.id"), and "squad_name". "user_squad_players" connect players to squads, as it has "user_squad_id" (that refers to "user_squads" by "user_squad_players.user_squad_id" ←→ "user_squads.id"), and "id_player" (that refers to "players_season" by "players_season.id_season" ← → "user_squad_players.id_player. As the squads can include the "same" player more than once, if the records are from different seasons). Both have "id" as their main key.

Part 3: queries:

- Query that returns the list of friend to a user with id=user_id:

```
SELECT * FROM friends WHERE user_id_1 = {user_id}
OR user_id_2 = {user_id}
```

- Query that increases the number of squad Likes by one:

```
- UPDATE user_squads
        SET likes = likes + 1
        WHERE id = {squad_id}
```

- Query that return the number of squad's Likes:

```
SELECT likes FROM user_squads WHERE id = {squad_id}
```

- Query that gives the players of a squad:

```
- SELECT short_name, overall, player_positions
  FROM players_season ' \
  JOIN user_squad_players ON
  players_season.id_season =
  user_squad_players.id_player
- JOIN players ON players_season.id_player =
  players.id_player ' WHERE
- user_squad_players.user_squad_id ={squad_id}
```

Query that gives the list of squads of a user:

```
SELECT id, squad_name FROM user_squads WHERE
user_id = {user_id}
```

- query to check if a name of a friend exist in the DB:

```
- SELECT * FROM users WHERE username =
  \'{friend_name}\'
```

- Query to add a new record of 2 users to the friends table:

```
INSERT INTO friends(user_id_1, user_id_2)
VALUES(%s, %s)
```

- Query to add a new squad:

```
- INSERT INTO user_squads(user_id, squad_name,
  likes) VALUES(%s, %s, %s)
```

- Query to find the latest squad (in order to add it players):

```
SELECT MAX(id) FROM user_squads;
```

- Query to find id_season of a player by his second key (id_player and season):

```
- SELECT id_season FROM players_season
  JOIN players ON players_season.id_player =
  players.id_player
  WHERE short_name = {player[0]}\' AND season =
  {player[-1]};
```

- Query to insert player to a squad (after we have the squad_id of the latest squad):

```
INSERT INTO
user_squad_players(user_squad_id, id_player)
VALUES(%s, %s)
```

- Query that returns player details:

```
- SELECT short_name, overall, nationality_name,
  value_eur, wage_eur, age, player_positions,
  potential,season FROM
      (SELECT players.short_name,
  players_season.overall,
- nations.nationality_name,
  players_season.value_eur,
  players_season.wage_eur, players_season.age
  players_season.player_positions,
  players_season.potential,
  players_season.season FROM players
  JOIN players_season ON players.id_player =
  players_season.id_player
  JOIN nations ON players.nationality_id =
  nations.nationality_id WHERE short_name =
  '{player_details[0]}' AND season =
  {player_details[-1]})
- as subquery;
```

- query that returns list of searched players by "player search":

```
SELECT short_name, overall, nationality_name,
value_eur, wage_eur, age, player_positions,
potential,season
FROM
         (SELECT players.short_name,
players_season.overall, nations.nationality_name,
players_season.value_eur, players_season.wage_eur,
players_season.age
, players_season.player_positions,
players_season.potential,
players_season.season FROM players
JOIN players_season ON players.id_player =
players_season.id_player
JOIN nations ON players.nationality_id =
nations.nationality_id
```

After that, comes "where" with the settings the user sets, and ending by "as subquery".

- query that returns list of searched players by "team search":

```
-  SELECT DISTINCT short_name, club_name,
   overall, nationality_name, value_eur,
   wage_eur, age,
   player_positions, potential, season
-  FROM
       (SELECT players.id_player,
   players.short_name, players_season.overall,
   nations.nationality_name, teams.club_name,
   players_season.value_eur,
   players_season.wage_eur, players_season.age,
   players_season.player_positions,
   players_season.potential,
   players_season.season FROM players
   JOIN players_season ON players.id_player =
   players_season.id_player
   JOIN nations ON players.nationality_id =
   nations.nationality_id
   JOIN teams ON players_season.club_team_id =
   teams.club_team_id JOIN teams_season ON
   players_season.club_team_id =
   teams_season.club_team_id
```

After that, comes "where" with the settings the user sets, and ending by "as subquery".

- query that returns list of searched players by "team search":

```
-  SELECT DISTINCT short_name, club_name,
   league_name, overall, nation, value_eur,
   wage_eur, age, player_positions, potential,
   season FROM
               (SELECT players.id_player,
   players.short_name, players_season.overall,
   leagues.nation, teams.club_name,
   players_season.value_eur,
   players_season.wage_eur,
-  teams_season.league_name, players_season.age,
   players_season.player_positions,
   players_season.potential,
   players_season.season FROM players JOIN
   players_season ON players.id_player =
   players_season.id_player JOIN teams ON
   players_season.club_team_id =
   teams.club_team_id JOIN teams_season ON
   players_season.club_team_id =
   teams_season.club_team_id AND
   players_season.season = teams_season.season
   JOIN leagues ON teams_season.league_name =
   leagues.league_name
```

After that, comes "where" with the settings the user sets, and ending by "as subquery".

- query that returns list of players of the best national team:

```
- SELECT players.short_name AS player_name,
  players_season.player_positions,
  players_season.overall, players_season.season
  FROM players JOIN players_season ON
  players.id_player = players_season.id_player
  JOIN nations ON nations.nationality_id =
  players.nationality_id
  JOIN (SELECT id_player, MAX(overall) as
  max_overall FROM players_season
  GROUP BY id_player) as max_ratings
  ON max_ratings.id_player =
  players_season.id_player AND
  max_ratings.max_overall =
  players_season.overall
  WHERE nations.nationality_name = \'{name}\'
  ORDER BY players_season.overall DESC LIMIT 11
```

- Query that return list of the most popular players:

```
SELECT short_name, overall, player_positions,
SUM(likes) as total_likes, season
FROM players_season
JOIN players ON players_season.id_player =
players.id_player
JOIN user_squad_players ON players_season.id_season
= user_squad_players.id_player
JOIN user_squads ON
user_squad_players.user_squad_id = user_squads.id
GROUP BY id_season
ORDER BY total_likes DESC LIMIT 11;
```

- Query that gives username by id:

```
- SELECT username FROM users WHERE id= {user_id}
```

- Query to sign in:

```
SELECT * FROM users WHERE username = \'{username}\'
AND password = \'{password}\'
```

- Query to register:

```
- INSERT INTO users(username, password)
  VALUES(%s, %s)
```

- Query to get the user with the highest number of Likes:

```
SELECT users.username, SUM(user_squads.likes) as
total_likes FROM user_squads JOIN users ON
user_squads.user_id = users.id GROUP BY
user_squads.user_id ORDER BY total_likes DESC
LIMIT 1;
```

- Query to get the most popular squad:

```
- SELECT user_squads.squad_name, users.username,
  SUM(user_squads.likes) as total_likes
  FROM user_squads
  JOIN users ON user_squads.user_id = users.id
  GROUP BY user_squads.id ORDER BY total_likes
  DESC LIMIT 1;
```

Part 4: code:

The code was written in python, and divided into 2 files: gui.py, myApi.py.

gui.py – in this file, we implemented the UI that is visible for the users. All the pages, texts, inputs and buttons. In order to do that, we used the tkinter library, which is for desktop apps. We learned the API of this library through sources in the internet such as Python-tutorial, Udemy, Youtube and etc. a challenging part was "rendering" the pages and show updated data after events. We did

that by creating "update" function that are called after every change in the app.

myApi.py – in this file, we implemented the "behind the scenes" part. All the connections to the database, the queries and the logic. One of the challenges in this part was to add players to the squad that was just created, since we can't know it id. The solution was to find the MAX(id), as it is the last squad that was created. The goal was to hide this part of the code, so we can use it for a future use of this API, when we change the UI, but still need the same logic and queries, and also if we want to use the current UI, but with different database, logic and API.