UNIVERSITÀ DI PISA

Master's degree in Artificial Intelligence and Data Engineering

Academic Year 2022/2023

Large-Scale and Multi-Structured Databases

FantaManager

Emmanuel Piazza, Edoardo Focacci, Matteo Razzai

GitHub repository: *https://github.com/Fochi1999/FantaManager*

Summary:

Introduction

FantaManager is a game application that mix the ideas of collecting cards and play them in a *Fantasy Football* (*Fanta Calcio* in Italian) style in order to compete with other people online.

The core function of the application is to collect cards and build up a team for the matchday, at the end of it, points will be assigned to the user based on the real footballer performance. The user must set his line-up and decide which cards to play in order to obtain points and credits.

A user can possess an unlimited number of cards resembling his/her favorite footballers and decide to trade or discard the owned cards.

Trades are possible in the game; a user can view all trade offers created by other users, accept them or search for a particular card. A user can also create his/her own trade offer by selecting offered and wanted cards, credits can also be exchanged.

The application permits the user to view the global or regional ranking, view other users' profile and his/her information. A store page is also implemented to allow users to buy cards or pack of cards. A user can also view a card's statistics during a specific matchday.

Requirements

Main actors

The application implements only three types of users:

- **Unregistered user**: the user that opens the application. Can only login or register.
- **User**: the normal user of the application that can do the basic interaction.
- **Admin**: the administrator of the application that has complete interaction with it.

Functional Requirements

- An unregistered user can only register and login to the application. It is mandatory to do the login operation to use the application.
- A user can view the list of cards (footballers) available in the game.
- A user can view a card's page and his statistics during the week. (Rating, number of goals, number of assists, etc.)
- A user can search for a specific card by filters or not.
- A user can view the collection of cards obtained in the game. (In Italian: *Rosa*)
- A user can edit or delete his account.
- A user can search a user's profile. (Find by name)
- A user can view another user's formation for a specific matchday.
- A user can decide the card's formation for the next week.
- A user can view the global or regional ranking for the current season.
- A user can do the logout operation.

- A user can make a public trade request in order to trade a card in his collection with another user for another card (duplicate of cards are allowed to be received). There are no limits to the user's cards collection size.
- A user can view all trade requests.
- A user can search a trade request by card offered or requested.
- A user can cancel a trade offer previously made.
- A user can accept a pending trade request.

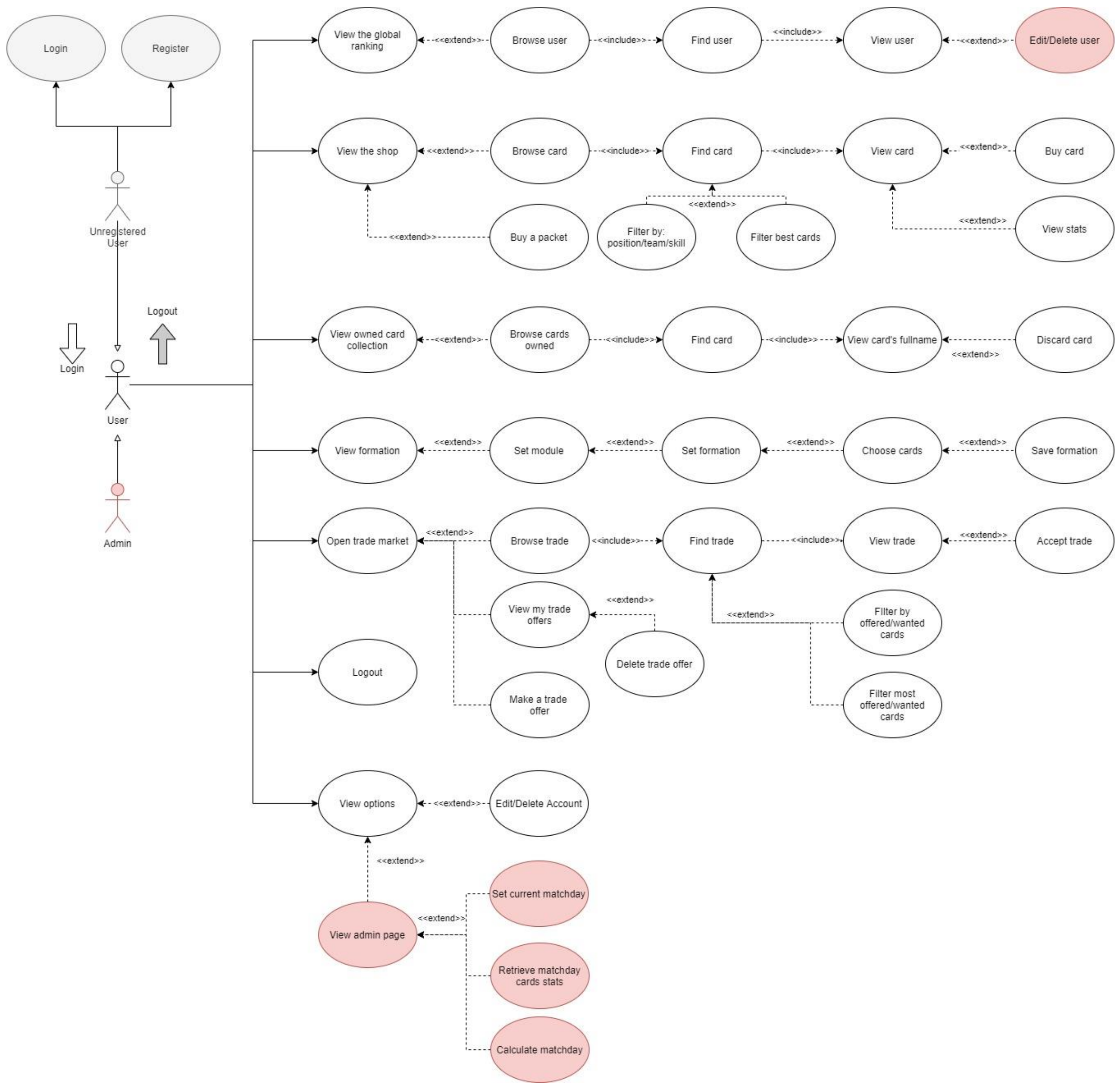
- A user can view the store page.
- A user can view a card’s statistics.
- A user can buy one or more packs of cards.
- A user can buy a specific card.
- A user can discard a card and receive credits in exchange.

- A user can login as an **Admin**. An Admin can do all the operations above, plus:
    - An admin can edit or delete another user’s account.
    - An admin can calculate the results of a matchday.

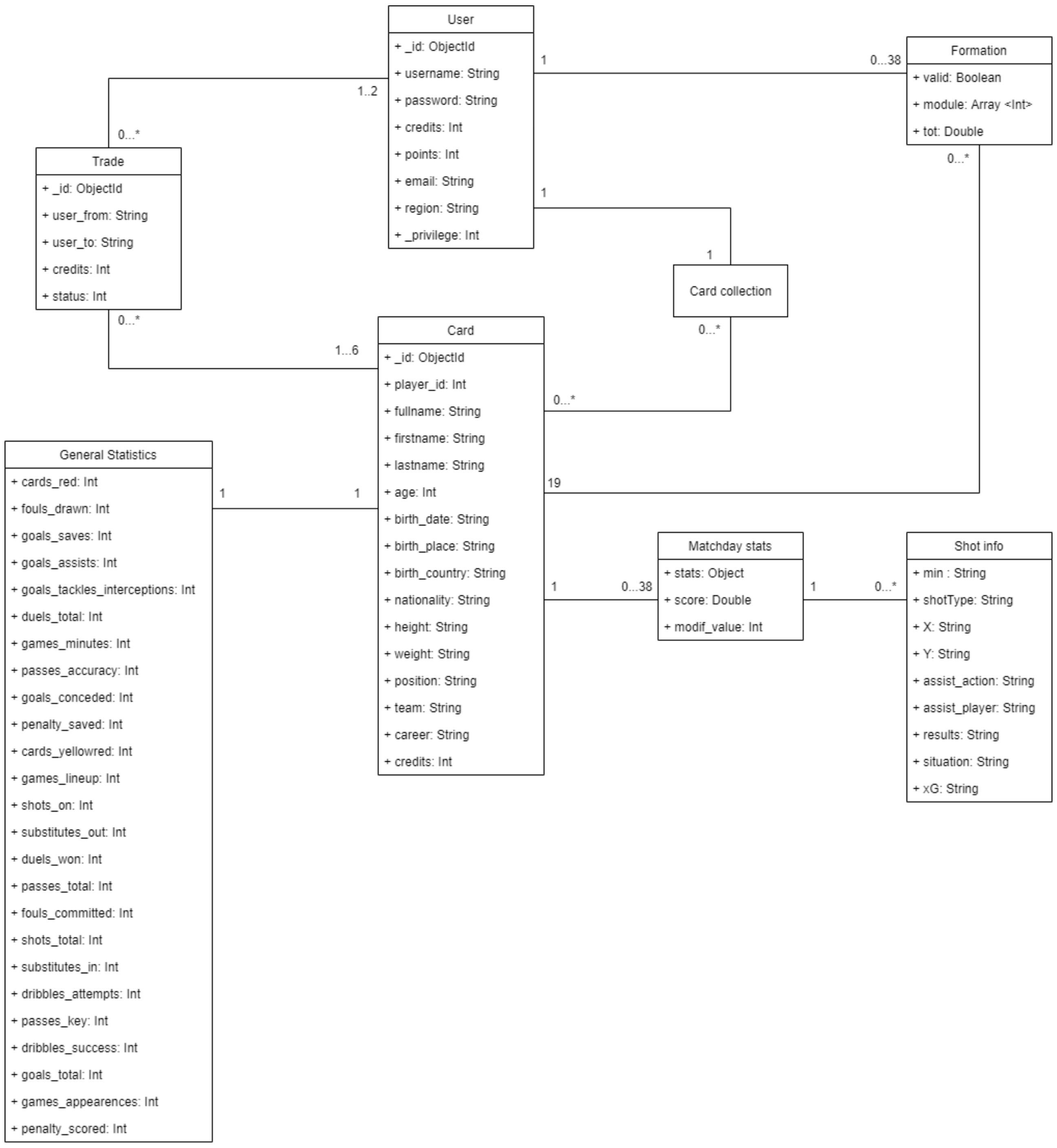
Non-Functional Requirements

- The application needs to be **consistent** regarding the application’s economy (trade and credits) to provide fair information among users.
- The transactions must be **monotonic**: every user must see the last version of the data and every trade request must be managed in the same order they are accepted.
- The password must be **protected** and stored **encrypted** for privacy issues.
- The system will be available 24 * 7.
- A Graphical User Interface is crucial for an involving game experience.
- The statistics regarding cards are not needed to be real-time, they are updated periodically, and they will be eventually consistent.

UML Use Cases



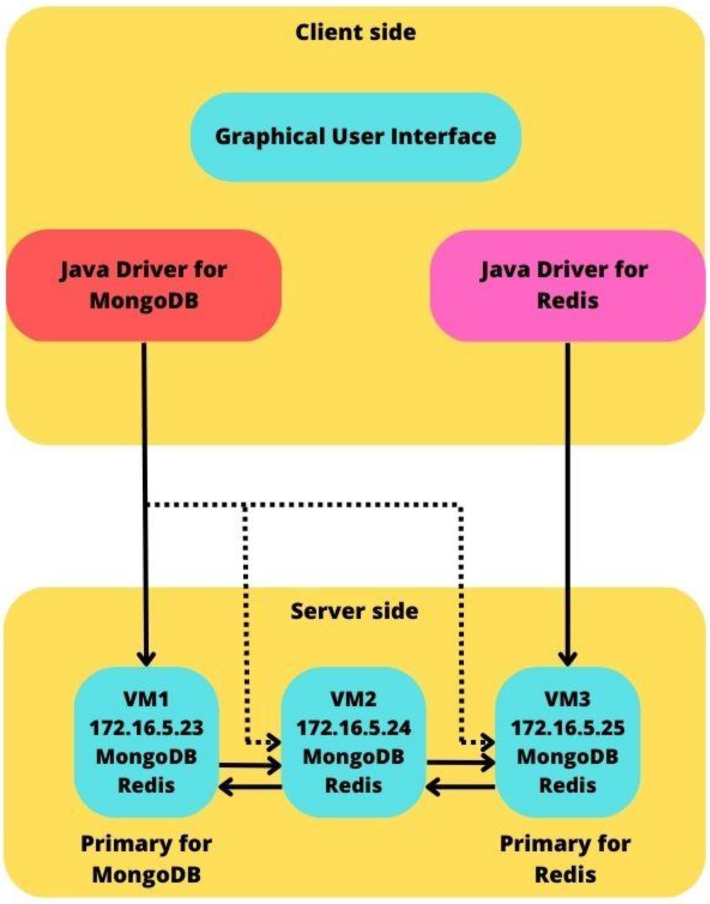
UML Class Diagram



Architectural Design

Software architecture

The application was implemented as a client-server architecture, with a middleware implemented on the client side.



Client Side

The client side can be divided into two main modules:

- The front-end module, which consists of a graphical user interface based on JavaFX. This allows the user to interact with the application in a simple and intuitive way.
- A middleware module, needed to interface the client to the server. Drivers were implemented to interface with MongoDB and with Redis.

The code is divided into package each represent a main feature of the application (ex: Controller, User, Trades, etc.)

Server side

The server side, as already mentioned, consists of three virtual machines, on which MongoDB and Redis are executed. There are some replicas of these servers, replicas are copies of the main server, updated in a deferred way, and used not only for backup purposes, but also for taking in charge of some read operations (if configured properly) so that to reduce load on the primary server. The Replicas Architecture used is a Master-Slave fashion, in which only the master can be in charge of write operations.

MongoDB

MongoDB server consists of 3 instances, where one act as primary and two as secondary server. Only the primary can accept writes, whereas the secondary in some cases can make the workload of the primary lighter accepting some read, the following paragraph explains in which circumstances this is acceptable.

3 different collections are uploaded on MongoDB: Users, Trades and Cards.

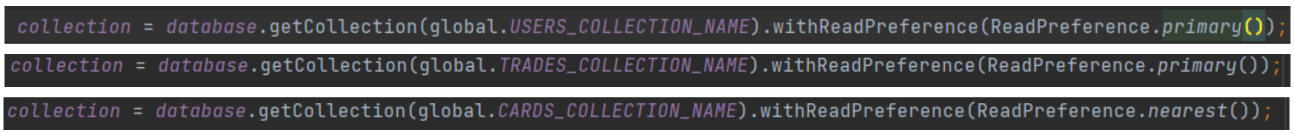
Users and Trades present in their collection, as explained in detail later, some fields where the economy of the game is involved, thus is necessary that every action that involves these two collections is consistent. Indeed, Users expect their trades to not be rollbacked for some consistency issue. To ensure that is needed to set the replicas of the collection Trades and Users so that a strong consistency is guaranteed.

The reads on these two collections are so done only on the primary server, so that every user can see the last updated state of the collection, and no one can see an earlier state. Also, given that the writes are executed only on the primary, for example, two users can't accept a trade at the same time.

Regarding the Cards collection, a more relaxed consistency is acceptable, indeed the writes on this collection are done only by the admin approximately one time every week, and an eventual consistency after this update is guaranteed by MongoDB.

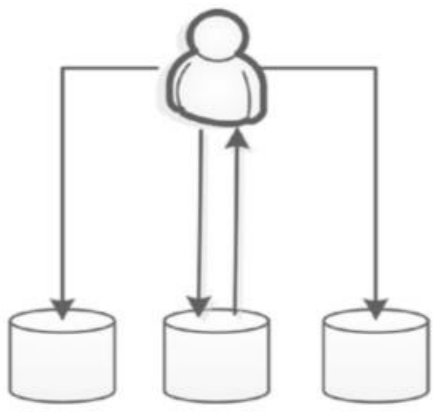
So, the reads can be performed on other server (*nearest()* option), making the queries to see some stats regarding the cards faster and more available.

For the writes we decided to utilize the *majority()* option, in case of a primary server shutdown a secondary should have the updated data.


```
collection = database.getCollection(global.USERS_COLLECTION_NAME).withReadPreference(ReadPreference.primary());
collection = database.getCollection(global.TRADES_COLLECTION_NAME).withReadPreference(ReadPreference.primary());
collection = database.getCollection(global.CARDS_COLLECTION_NAME).withReadPreference(ReadPreference.nearest());
```

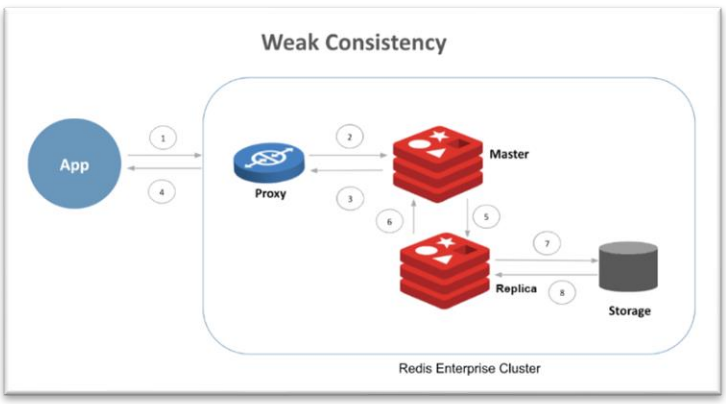
Redis

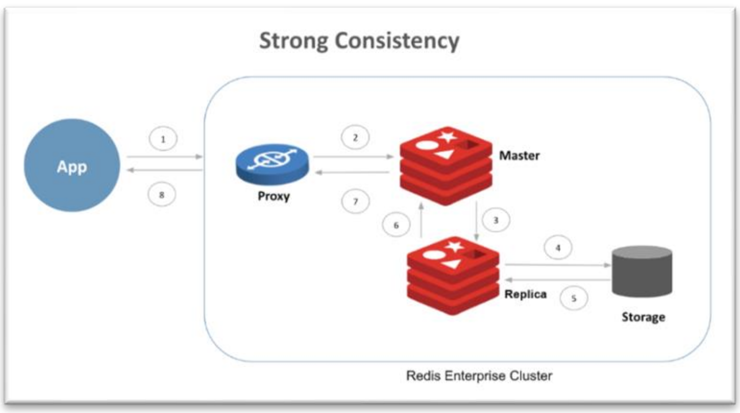
Redis server consist of 3 separate instances, where one act as the primary and two as secondaries. Same as MongoDB, only the primary can accept writes, but in this case the secondaries cannot accept reads, given that consistency is one of the main factors that we should prioritize on the values saved on Redis.
The card's collection of each user is saved into Redis; this choice is due to the fact that writes are faster on a Key-Value DB. Consistency is one of the values that we must prioritize because when a user acquires a new card, he/she expects to have that card in his squad and the ability to choose him for the next matchday.
Even if Redis use asynchronous replication (as explained in its documentation) we can obtain that. We decided to utilize the *WAIT* option of Redis, this option, after each write, checks that a minimum number of replicas are connected to the main server, to ensure that each write is propagated to each replica. In this way we can decide to only make one read. The data read must be updated to the last value.



N=3 W=3 R=1
Slow writes, fast reads, consistent
There will be 3 copies of the data.
A write request only returns when all 3 have written to disk.
A read request only needs to read one version.

The difference between the possible settings of Redis:





Database Organization

Database choices

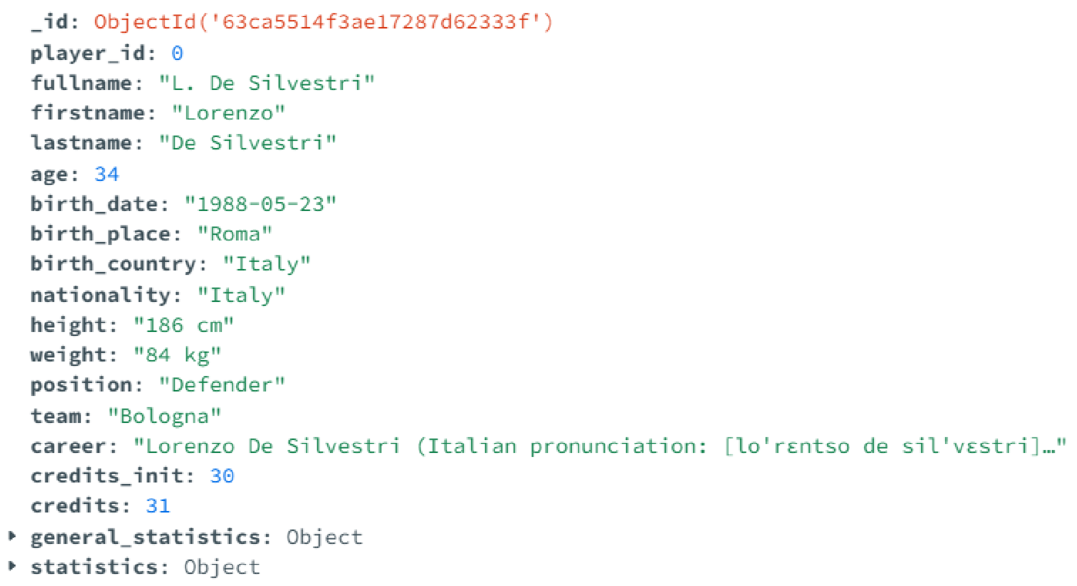
- **MongoDB:** We decided to use a DocumentDB for its schema-less properties (in Trades we can have from 1 to 3 offered cards, and from 0 to 3 wanted cards) and document embedding. Another important aspect is the possibility of indexing and for the application's analytics is very important the possibility of make complex queries.

- **Redis:** We chose Redis for the part of User's cards collection, we thought about a Key-value DB for that specific part because we enable trades and the possibility to buy cards and packets (which contains random cards) for all the season, so we expect a lot of changes in the cards collection of a User; adding cards (for example after a completed Trade or after buy a card in the Shop) or delete cards (after a Trade or if a User decides to discard one of his/her own card). More details on this choice are available on Redis design chapter.

MongoDB

We have three entities stored in the Document DB:

- **Cards**: in this document is stored a card that can be collected from users and resembles a footballer. The structure is the following:

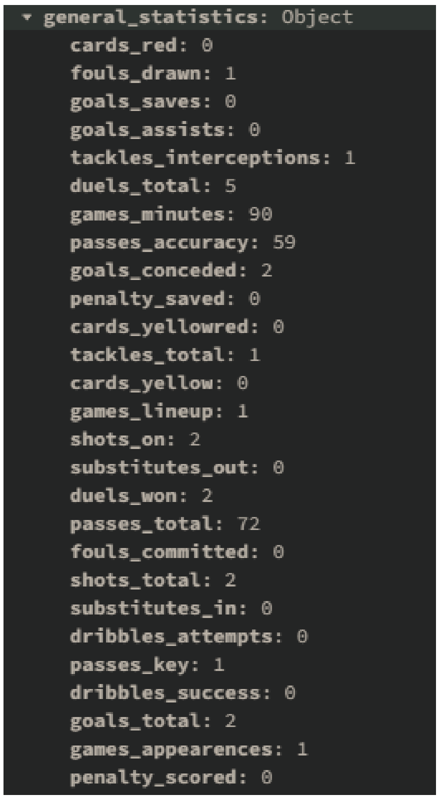


```
  _id: ObjectId('63ca5514f3ae17287d62333f')
  player_id: 0
  fullname: "L. De Silvestri"
  firstname: "Lorenzo"
  lastname: "De Silvestri"
  age: 34
  birth_date: "1988-05-23"
  birth_place: "Roma"
  birth_country: "Italy"
  nationality: "Italy"
  height: "186 cm"
  weight: "84 kg"
  position: "Defender"
  team: "Bologna"
  career: "Lorenzo De Silvestri (Italian pronunciation: [lo'rɛntso de sil'vɛstri]…"
  credits_init: 30
  credits: 31
▸ general_statistics: Object
▸ statistics: Object
```

Every field resembles an attribute of the real footballer and some of his information. The *career* attribute is optional, because it was not possible to retrieve a summary of the footballer's career from Wikipedia for every footballer.

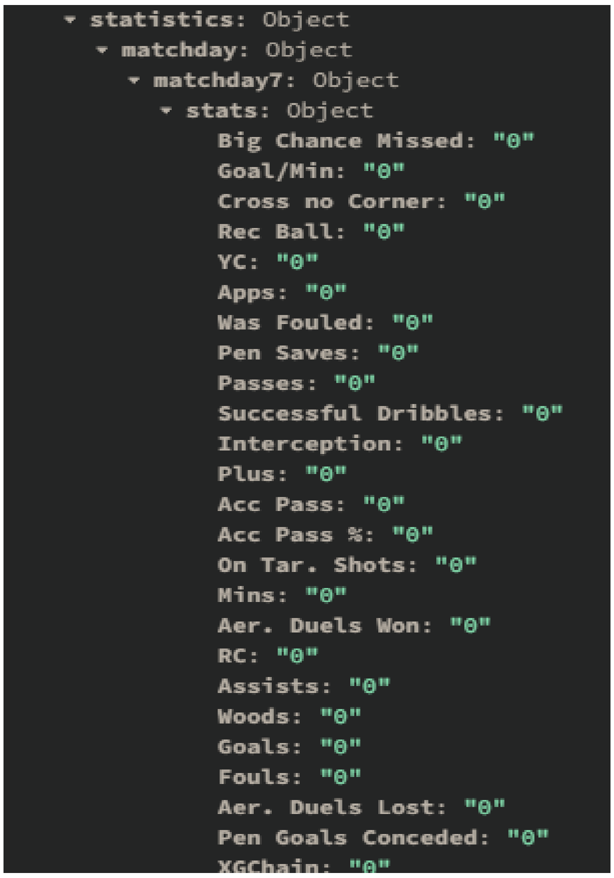
Let's give a special look to *general_statistics* and *statistics* objects.

Inside the general_statistics object is inserted attributes resembling all actions done by the footballer during the whole season. Obviously, at the start of the season all values are settled to zero and are modified only after a matchday has been calculated.

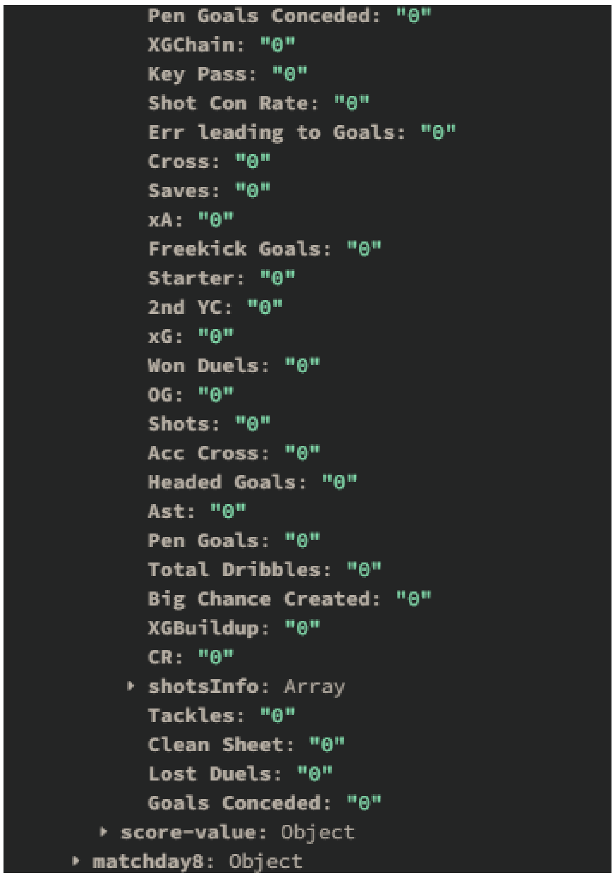


```
▾ general_statistics: Object
    cards_red: 0
    fouls_drawn: 1
    goals_saves: 0
    goals_assists: 0
    tackles_interceptions: 1
    duels_total: 5
    games_minutes: 90
    passes_accuracy: 59
    goals_conceded: 2
    penalty_saved: 0
    cards_yellowred: 0
    tackles_total: 1
    cards_yellow: 0
    games_lineup: 1
    shots_on: 2
    substitutes_out: 0
    duels_won: 2
    passes_total: 72
    fouls_committed: 0
    shots_total: 2
    substitutes_in: 0
    dribbles_attempts: 0
    passes_key: 1
    dribbles_success: 0
    goals_total: 2
    games_appearences: 1
    penalty_scored: 0
```

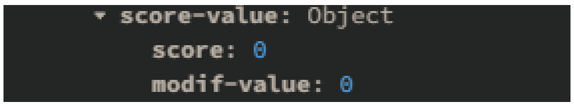
Meanwhile, inside the statistics object is inserted an array of 38 elements, each representing a matchday. Inside a matchday object there are a list of attributes resembling what the footballer did during the matchday, his score and some shots info (if he made some). The matchday array is already initialized for every card and his values are update only after a matchday has been played.



```
▾ statistics: Object
  ▾ matchday: Object
    ▾ matchday7: Object
      ▾ stats: Object
          Big Chance Missed: "0"
          Goal/Min: "0"
          Cross no Corner: "0"
          Rec Ball: "0"
          YC: "0"
          Apps: "0"
          Was Fouled: "0"
          Pen Saves: "0"
          Passes: "0"
          Successful Dribbles: "0"
          Interception: "0"
          Plus: "0"
          Acc Pass: "0"
          Acc Pass %: "0"
          On Tar. Shots: "0"
          Mins: "0"
          Aer. Duels Won: "0"
          RC: "0"
          Assists: "0"
          Woods: "0"
          Goals: "0"
          Fouls: "0"
          Aer. Duels Lost: "0"
          Pen Goals Conceded: "0"
          XGChain: "0"
```

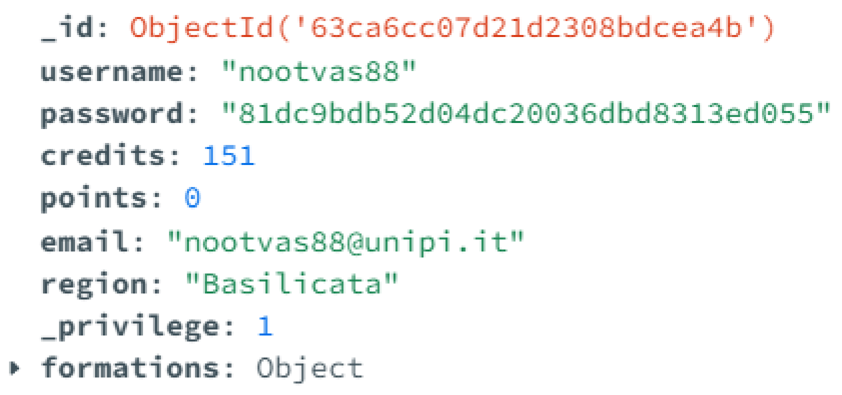


```
          Pen Goals Conceded: "0"
          XGChain: "0"
          Key Pass: "0"
          Shot Con Rate: "0"
          Err leading to Goals: "0"
          Cross: "0"
          Saves: "0"
          xA: "0"
          Freekick Goals: "0"
          Starter: "0"
          2nd YC: "0"
          xG: "0"
          Won Duels: "0"
          OG: "0"
          Shots: "0"
          Acc Cross: "0"
          Headed Goals: "0"
          Ast: "0"
          Pen Goals: "0"
          Total Dribbles: "0"
          Big Chance Created: "0"
          XGBuildup: "0"
          CR: "0"
        ▸ shotsInfo: Array
          Tackles: "0"
          Clean Sheet: "0"
          Lost Duels: "0"
          Goals Conceded: "0"
      ▸ score-value: Object
    ▸ matchday8: Object
```



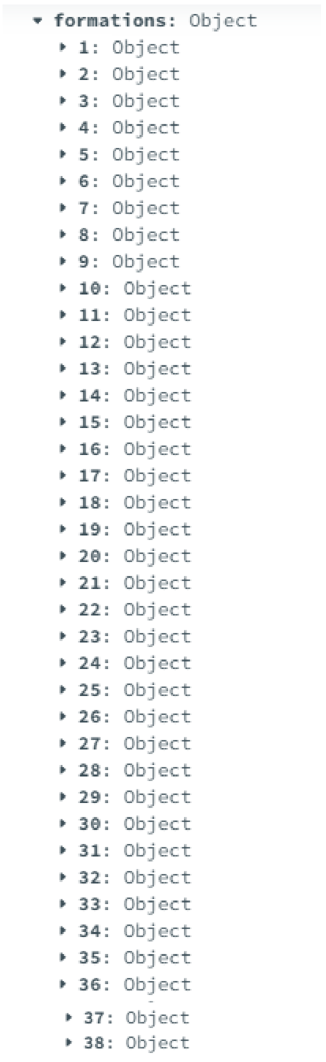
```
▾ score-value: Object
    score: 0
    modif-value: 0
```

- **Users:** in this collection are stored all the users and the information about them. The structure is the following:

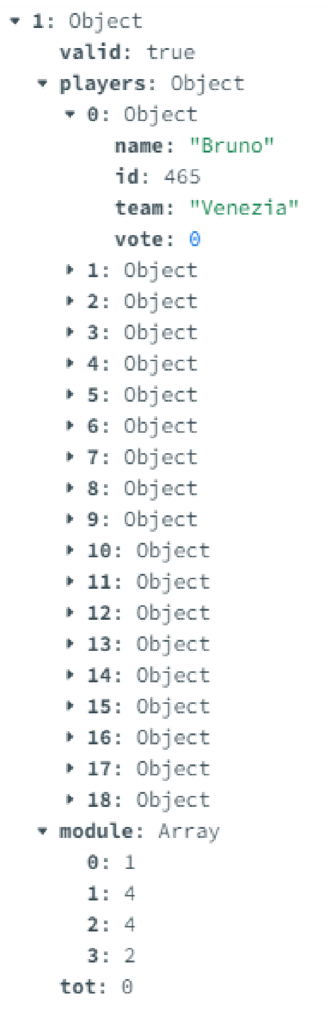


```
  _id: ObjectId('63ca6cc07d21d2308bdcea4b')
  username: "nootvas88"
  password: "81dc9bdb52d04dc20036dbd8313ed055"
  credits: 151
  points: 0
  email: "nootvas88@unipi.it"
  region: "Basilicata"
  _privilege: 1
▸ formations: Object
```

The formation object is used to store the formation settled from the user for each matchday. Inside that object there are 38 other objects (one for each matchday) and inside each of them are listed the cards played, the module used, and the total score received after the matchday has been calculated. For each card is stored the name, the id, the team, and the score received in the current matchday.

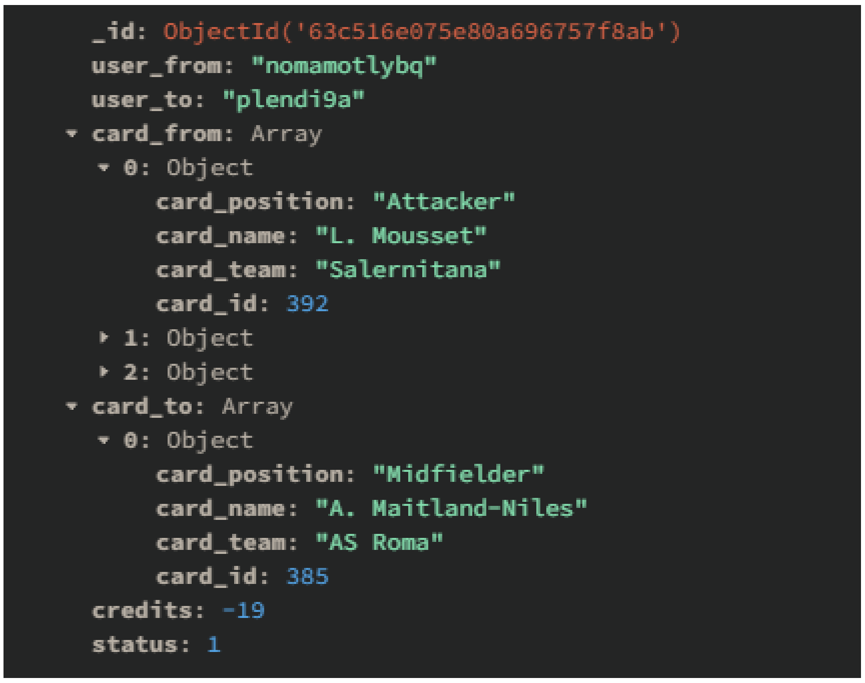


```
▾ formations: Object
  ▸ 1: Object
  ▸ 2: Object
  ▸ 3: Object
  ▸ 4: Object
  ▸ 5: Object
  ▸ 6: Object
  ▸ 7: Object
  ▸ 8: Object
  ▸ 9: Object
  ▸ 10: Object
  ▸ 11: Object
  ▸ 12: Object
  ▸ 13: Object
  ▸ 14: Object
  ▸ 15: Object
  ▸ 16: Object
  ▸ 17: Object
  ▸ 18: Object
  ▸ 19: Object
  ▸ 20: Object
  ▸ 21: Object
  ▸ 22: Object
  ▸ 23: Object
  ▸ 24: Object
  ▸ 25: Object
  ▸ 26: Object
  ▸ 27: Object
  ▸ 28: Object
  ▸ 29: Object
  ▸ 30: Object
  ▸ 31: Object
  ▸ 32: Object
  ▸ 33: Object
  ▸ 34: Object
  ▸ 35: Object
  ▸ 36: Object
  ▸ 37: Object
  ▸ 38: Object
```



```
▾ 1: Object
    valid: true
  ▾ players: Object
    ▾ 0: Object
        name: "Bruno"
        id: 465
        team: "Venezia"
        vote: 0
    ▸ 1: Object
    ▸ 2: Object
    ▸ 3: Object
    ▸ 4: Object
    ▸ 5: Object
    ▸ 6: Object
    ▸ 7: Object
    ▸ 8: Object
    ▸ 9: Object
    ▸ 10: Object
    ▸ 11: Object
    ▸ 12: Object
    ▸ 13: Object
    ▸ 14: Object
    ▸ 15: Object
    ▸ 16: Object
    ▸ 17: Object
    ▸ 18: Object
  ▾ module: Array
      0: 1
      1: 4
      2: 4
      3: 2
    tot: 0
```

- **Trades:** in this document is stored a trade that a user created to exchange cards with another user. The structure is the following:



```
  _id: ObjectId('63c516e075e80a696757f8ab')
  user_from: "nomamotlybq"
  user_to: "plendi9a"
▾ card_from: Array
  ▾ 0: Object
      card_position: "Attacker"
      card_name: "L. Mousset"
      card_team: "Salernitana"
      card_id: 392
  ▸ 1: Object
  ▸ 2: Object
▾ card_to: Array
  ▾ 0: Object
      card_position: "Midfielder"
      card_name: "A. Maitland-Niles"
      card_team: "AS Roma"
      card_id: 385
  credits: -19
  status: 1
```

The fields user_from and user_to resemble respectively the user that created the trade and the user who accepted it. At the creation of the trade the string *user_to* is empty and the write is done when a user accepts the trade.
The fields card_from and card_to resemble respectively:
  - Card_from – the cards offered by the user that created the trade.
  - Card_to – the cards wanted by the user that created the trade.

The maximum number of cards that can be traded is 3 (for each side) and a user can decide to offer a card even without receiving some in exchange. For each card is saved the position, the name, the team, and the id.
The credits value resembles the total number of credits exchanged, if the value is negative that means that the user that created the trade wants to receive that number of credits, if the number is positive this means that the credits are offered by the trade creator. This is implemented for seeing the trade with the point of view of the logged-in user.
At last, the status value indicates the status of the trade: 0 for pending, 1 for completed and accepted by someone.

Redundancies

In order to achieve fast queries, we introduced some redundancies, to avoid join-operation with other collections. The principal redundancy and denormalization regards the Cards collection.

We replicate some attributes about this collection to avoid join operations.

In **Redis** we saved *player_id, name, team* and *position* attributes in order to avoid join operations when a User retrieve his/her owned cards collection (to view it or when he wants to upload the formation for the next matchday). In case of formation, when he chooses a player, he needs to know the position, the team and obviously the name, to make the right choice on the starting lineup.

In **Trades** collection we inserted some embedded documents regarding the offered/wanted cards, with *player_id, name, team* and *position* attributes. We did this because after a successful trade the specified cards has to be added to or deleted from the card's collection of the users involved in the Trade; so, we need all the four fields in case of addition to the collection, because we need to put this info in the user's card collection on Redis. In case of deleting a card from the user's card collection, we need at least the player_id field, but we added the other three parameters to make understand, to the logged user, which player is involved in the trade (name and team are necessary to understand who is the footballer, but also the position can be useful). So, we decide to add the embedded document to the *card_from/card_to* fields.

In **Users** collection we made the same choice in the *players* field inside the *formation* field for that specific match. We need the quoted 4 fields for retrieve the inserted formation, if the User makes a login after he inserted the formation but before a match is played, he can see which card he put on the starting lineup, and maybe change his idea. Then we gave the possibility to all the users to see the previous formation of the other users.

Problem of nested documents

We have collections with arrays of embedded documents, like the field statistics.matchday in Cards and the field formation for the Users. We have the problem of the growing of the document, it can cause database inefficiencies because if a collection exceeded the expected space for that document, the DBMS would have to relocate it, creating inefficiencies.

The solution to the problem is the following. We know the number of matches of Serie A, so we decided to put 38 embedded documents for each document of Cards collection and Users collection, each of them with empty fields, in case of string, and with zero, in case of Integer or Double.

Redis

This database contains all elements that resemble the card collection of every user. The keys inserted are the following (the *xxx* indicate the numerical value of the attribute):

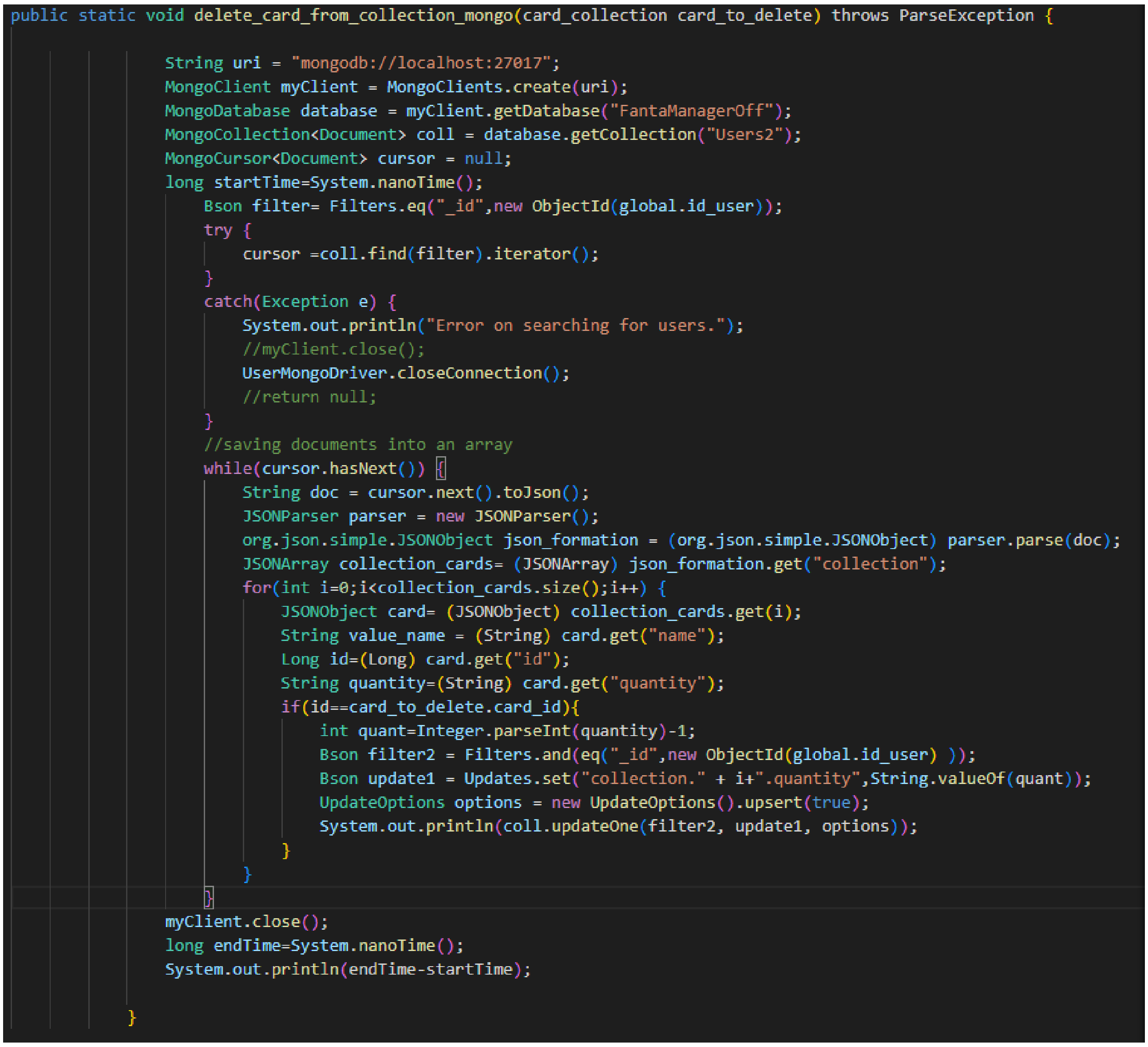
- **user_id:*xxxx*:card_id:*xxx*:name** – contains the name of the card owned by the user.
- **user_id:*xxxx*:card_id:*xxx*:position** – contains the position of the card owned by the user.
- **user_id:*xxxx*:card_id:*xxx*:team** – contains the name of the team of the card owned by the user.
- **user_id:*xxxx*:card_id:*xxx*:quantity** – contains the number of copies of the same card owned by the user.

Those elements were added inside the Key-Value DB instead of the Document DB because, being *FantaManager* an application focused on trade and collect cards with other users, this approach is faster given that there are a lot of writes instead of reads.

The alternative could be put a ‘collection’ field in the User collection on MongoDB, with documents embedded with fields which represent the same formation saved on Redis.

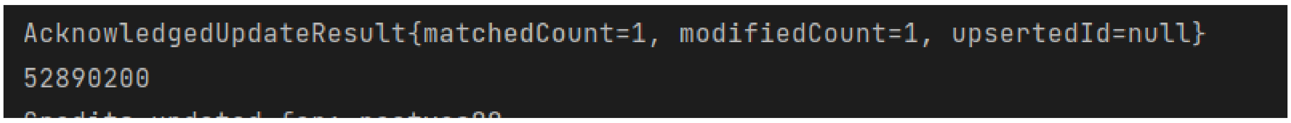
To understand how the performance changes after an edit (for example, the quantity of a specific card collected by a user) we tried two difference functions with the objective to decrease the quantity of a cards.

In the following function we can see the quoted operation on MongoDB:



```java
public static void delete_card_from_collection_mongo(card_collection card_to_delete) throws ParseException {

                String uri = "mongodb://localhost:27017";
                MongoClient myClient = MongoClients.create(uri);
                MongoDatabase database = myClient.getDatabase("FantaManagerOff");
                MongoCollection<Document> coll = database.getCollection("Users2");
                MongoCursor<Document> cursor = null;
                long startTime=System.nanoTime();
                    Bson filter= Filters.eq("_id",new ObjectId(global.id_user));
                    try {
                        cursor =coll.find(filter).iterator();
                    }
                    catch(Exception e) {
                        System.out.println("Error on searching for users.");
                        //myClient.close();
                        UserMongoDriver.closeConnection();
                        //return null;
                    }
                    //saving documents into an array
                    while(cursor.hasNext()) {
                        String doc = cursor.next().toJson();
                        JSONParser parser = new JSONParser();
                        org.json.simple.JSONObject json_formation = (org.json.simple.JSONObject) parser.parse(doc);
                        JSONArray collection_cards= (JSONArray) json_formation.get("collection");
                        for(int i=0;i<collection_cards.size();i++) {
                            JSONObject card= (JSONObject) collection_cards.get(i);
                            String value_name = (String) card.get("name");
                            Long id=(Long) card.get("id");
                            String quantity=(String) card.get("quantity");
                            if(id==card_to_delete.card_id){
                                int quant=Integer.parseInt(quantity)-1;
                                Bson filter2 = Filters.and(eq("_id",new ObjectId(global.id_user) ));
                                Bson update1 = Updates.set("collection." + i+".quantity",String.valueOf(quant));
                                UpdateOptions options = new UpdateOptions().upsert(true);
                                System.out.println(coll.updateOne(filter2, update1, options));
                            }
                        }
                    }
                myClient.close();
                long endTime=System.nanoTime();
                System.out.println(endTime-startTime);

            }
```

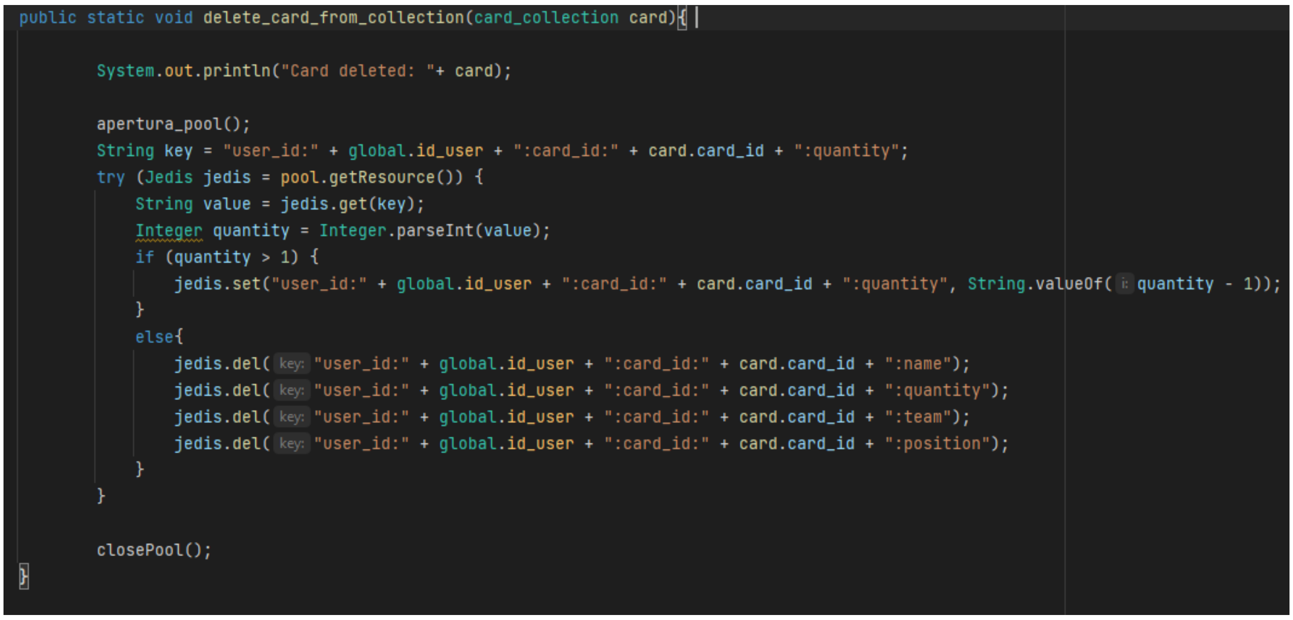
The performance of this action was the following:



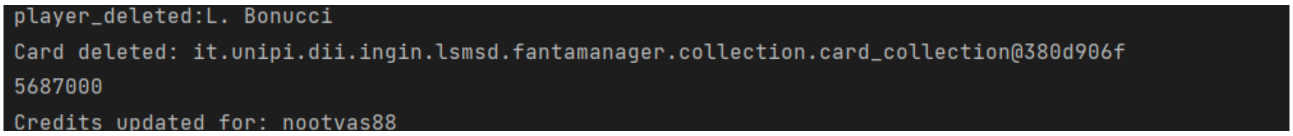
```
AcknowledgedUpdateResult{matchedCount=1, modifiedCount=1, upsertedId=null}
52890200
```

So, more or less **53 ms**.

Using the function used in our application:


```
public static void delete_card_from_collection(card_collection card){

        System.out.println("Card deleted: "+ card);

        apertura_pool();
        String key = "user_id:" + global.id_user + ":card_id:" + card.card_id + ":quantity";
        try (Jedis jedis = pool.getResource()) {
            String value = jedis.get(key);
            Integer quantity = Integer.parseInt(value);
            if (quantity > 1) {
                jedis.set("user_id:" + global.id_user + ":card_id:" + card.card_id + ":quantity", String.valueOf( i: quantity - 1));
            }
            else{
                jedis.del( key: "user_id:" + global.id_user + ":card_id:" + card.card_id + ":name");
                jedis.del( key: "user_id:" + global.id_user + ":card_id:" + card.card_id + ":quantity");
                jedis.del( key: "user_id:" + global.id_user + ":card_id:" + card.card_id + ":team");
                jedis.del( key: "user_id:" + global.id_user + ":card_id:" + card.card_id + ":position");
            }
        }

        closePool();
}
```

The performance was:


```
player_deleted:L. Bonucci
Card deleted: it.unipi.dii.ingin.lsmsd.fantamanager.collection.card_collection@380d906f
5687000
Credits updated for: nootyas88
```

So, **5ms**. Ten times less than the same operation on MongoDB.

Thinking of a Trade operation, in which we can have 6 changes of quantity (maximum 3 cards offered e 3 accepted), the total times for a Trade operation using Redis will be much lower than in the Mongo scenario.

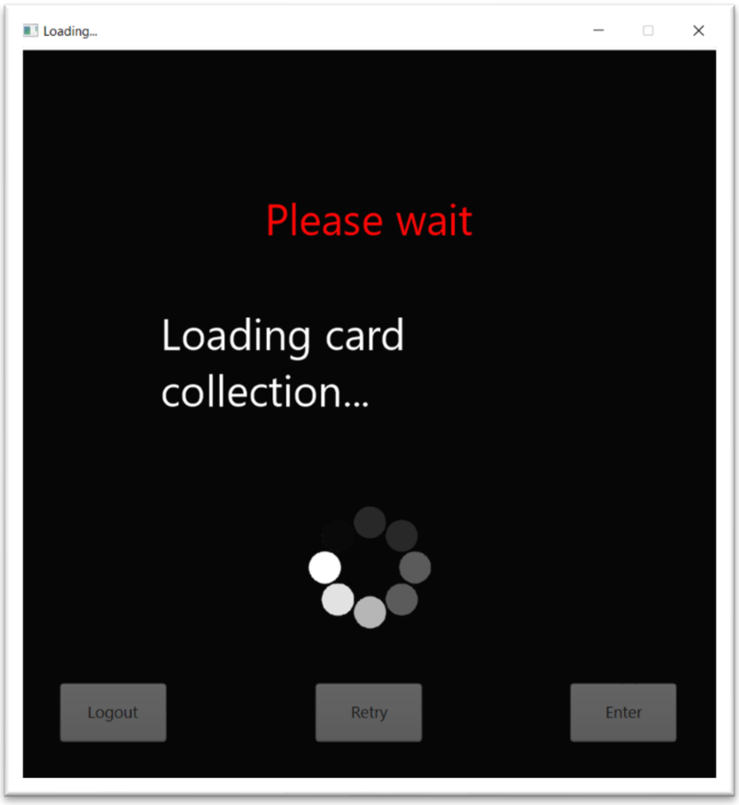
Other admin-only keys were added:

- **admin:next_matchday –** is read by every user after the login and indicates the number of the matchday that is currently playing. Only the admin can update this value.
- **admin:updated_matchdays –** is an array that contains a list of the updated matchday, only the admin can read and write the value of the key. It is used to know which matchday has been updated and which not. The importance of this key is for situation in which the calculation of a matchday, for some reason, has been postponed (for example: a lot of matches has been postponed for some reason, like we saw during the pandemic)

Caching

It is implemented a sort of caching mechanic inside the application. After a user is logged-in inside the application, a several read operations were made.

The list of all user’s card collection and the full cards list is retrieved at the start of the application and is saved locally in order to guarantee better performance during the usage of all functionalities (ex: view the *Shop Page* or the *Collection Page*).



Please wait

Loading card collection...

If the initial loading goes through an error the user can choose to retry the load or logout from the application. If the loading is completed without errors the user can decide to enter the application or logout from it.

The locally saved list of user’s owned cards is also updated after a trade is created, accepted, or deleted; this update is always made after the write operation on Redis is successfully completed. When a user performs the logout operation the local list of owned cards is cleared.

The full cards list, instead, is retrieved only once at the start of the application and even if a user logs out and another user logs in, the full cards list will not be deleted from the memory in order to have a faster loading screen for the next user.

Inter-Databases Consistency

Considering how the data was distributed between the two databases, we must consider the problem of consistency on the information that involves the trades system and the delete of a user:

1. When a trade is created, accepted, or deleted, some functions are used in the process that update both the Document DB and Key-Value DB. It is important to consider the fact that one of the two updates can go through an error, so rollback-like operations were implemented to maintain consistency:
    - The Document DB is the first thing that is updated, if this operation occurred in an error, no changes will be made on the Key-Value DB.
    - The Key-Value DB update is the operation that follow next; if that action goes through an error, the documents involved in the last step are reverted in his previous state or restored if they were deleted.

2. When a user is deleted from the database, instead, the first thing done is delete all his collection of cards from the Key-Value DB and then all documents involved with this user are removed from the Document DB. This method is used because in case of error is simpler retrieve all deleted cards instead of all trades on which the user was involved.

Implementation

Scraping

A scraping script was written with Python in order to retrieve information about footballers and his statistics. That information is retrieved all in once and saved into files. This approach was implemented to simulate a real scraping system; a real weekly running scraping algorithm could have been too heavy to process and maintenance.

In the application, the admin decides when stats are retrieved from the files and inserted into cards' documents, even though this action must be made once every week, if possible.

The source of those information were popular football stats sites such as: Understats, Kickest and APIFootball. Wikipedia was used to retrieve a brief description of footballers, when possible.

Main Packages and Classes

In this section will be presented the main packages of the application and the respective classes.

it.unipi.dii.ingin.lsmsd.fantamanager

This package just contains the *app.java* file that permits to launch the application.

it.unipi.dii.ingin.lsmsd.fantamanager.admin

This package contains:

- *calculate_matchday.java*: it contains the functions that involves the calculation of a matchday.
- *retrieve_matchday.java*: This is how the scraping is simulated into the application. We simulate to take stats of each match from the outside and write them on Cards collection.
- *general_statistics.java:* This class is used to manage changes and the calculation of the card's general statistics about the current season.

it.unipi.dii.ingin.lsmsd.fantamanager.collection

This package contains all the files involved in the card collection of a user:

- *card_collection.java*: class used for managing the *card* object within the User's collection of cards.
- *collectionRedisDriver.java*: includes all functions that interact with Redis.
- *LineTable.java*: only used for printing the collection of card in a javafx table.

it.unipi.dii.ingin.lsmsd.fantamanager.formation

This package contains all the files involved in the formation that a user can set up:

- *formation.java*: class used for managing the *formation* object.
- *player_formation.java*: class used for managing a single card inserted on a formation.

it.unipi.dii.ingin.lsmsd.fantamanager.page_controllers

This package contains all the files that permits to perform an action when a user interacts with the graphical interface of the application. Apart from the *Home Page* controller and the *Loading Screen*, all other controllers are inserted into another package that represents the entity on which is used primarily in.

The name of the file is self-explanatory on which page are used in:

- *HomeController.java*
- *LoadingScreen.java*
- **users**
  - *AdminPageController.java*
  - *login_registrationController.java*
  - *OptionController.java*
  - *RankingController.java*
  - *SeeUserController.java*
- **trades**
  - *TradesController.java*
  - *NewTradeController.java*
- **cards**
  - *CollectionController.java*
  - *SeeCardController.java*
  - *ShopController.java*

    - *ShotsStatsController.java*

- **formation**
    - *FormationController.java*
    - *ChoisePlayerFormationController.java*
    - *SeeUserFormationController.java*

it.unipi.dii.ingin.lsmsd.fantamanager.player_classes

This package contains all the files involved with the card entity:

- *CardMongoDriver.java*: includes all functions that interacts with the database.
- *see_card.java:* includes all functions used in the *SeeCard* page and not involving changes on the GUI. Also contains some little calls on the database.

it.unipi.dii.ingin.lsmsd.fantamanager.trades

This package contains all the files involved with the trade entity:

- *Trade.java*: this class is used to interact with a trade entity and to save it locally.
- *TradeMongoDriver.java*: includes all functions that interacts with the database about the *trades* collection.

it.unipi.dii.ingin.lsmsd.fantamanager.user.user

This package contains some files involved with the user entity and the login operation:

- *login.java*: includes all functions that permits the login/registration operations.
- *user.java*: this class is used to interact with a user entity and to save it locally.

it.unipi.dii.ingin.lsmsd.fantamanager.user.userMongoDriver

This package contains other files involved with the user entity and the connection with the databases:

- *formationMongoDriver.java*: includes all functions that interacts with the database on the formation page.
- *OptionsMongoDriver.java:* includes all functions that interacts with the database on the options page.

- *RankingMongoDriver.java:* includes all functions that interacts with the database on the ranking page.
- *SeeUserMongoDriver.java:* includes all functions that interacts with the database on the *SeeUser* page.
- *UserMongoDriver.java:* includes functions that opens and close the connection with the database about the *users* collection.

it.unipi.dii.ingin.lsmsd.fantamanager.user.util

This package contains utilities file and global variables called many times in the application:

- *global.java*: contains all the global variable used.
- *hash.java:* contains a hash function used for secure passwords.
- *util_controller.java*: contains some function used often on controllers.
- *utilities.java:* contains some general function used in the application but not associated to any package or entity previously mentioned.

src/main/resources

In this path are contained all files involved with the GUI which is formed of *.fxml* files, *.css* files and some images.

Constraints

Some constraints were added in two entities:

- A user's username must be unique. This is implemented because when interacting with the Document DB the username is used as primary key. During the registration of a new user, it is checked that the username doesn't already exists on the database.
- A card's id must be unique. This is implemented because when interacting with the Key-Value DB this attribute is used for retrieving information of a user's card collection. Card's ids are automatically generated when creating the card entity and his value are determined on his insertion number on the database.
- In the same trade, a user cannot insert in the offer (or wanted) fields the same card more than once. This constraint is used for easily search for cards into the database, it is regulated with some simple fields check into the page controller before the trade creation.

Most relevant queries

CRUD Operations

It is a list of simple CRUD operations applied on MongoDB, the pipeline aggregations are shown in the next chapter.

Users

- Retrieve a **User** attribute (*UserMongoDriver*)
- Delete a **User** and his **collection of cards** (*SeeUserMongoDriver*) → A **Normal User** can only delete his/her own account, and the **Admin** can delete every User
- Retrieve a **User** searching by his username (*SeeUserMongoDriver)*
- Search the first 100 **Users** by points for a specific region (*RankingMongoDriver*)
- **Best_for_region (aggregation)** (*RankingMongoDriver*)
- Search **User** inside the ranking table by his/her username or part of it (*RankingMongoDriver*)
- Update **User** credits (after a match they take the points that they have done like credits to spend, or if they buy a packet or a player or make a trade with credits involved) (*OptionsMongoDriver*)
- Find duplicate regarding mail or username, for example, at the time of registration a **User** can't use a username of another User
- Change a **User** attribute
- A User can insert his/her own formation for that matchday

Trades (*TradeMongoDriver*)

- Delete a specific **Trade** using its id
- **Retrieve_most_present (aggregation)**
- A **User** can search all the **Trades** made by himself/herself
- Search a **Trade** using the fullname of a Card or part of it (offered or wanted)
- Search all pending **Trade** (with "status" field equal to 0)
- Search **Trade** by its id
- Update **Trade** status
- Insert a new **Trade**
- Delete all **Trades** involving a specific **User**
- If a **User** decides to change his/her own username, the username has to be changed also from all the **Trades** involving that **User**

Cards (*CardMongoDriver*)

- Retrieve all the **Cards**
- Search a specific Card using the player’s card fullname or part of it
- **Search_card_by (aggregation)**
- Retrieve **Cards** by *team*
- Retrieve **Cards** by *position*
- Retrieve **Cards** by *team* and *position*
- Retrieve **Card**’s *credits* (its value)
- Update **Card’**s *general statistics* and *credits* after a matchday (only by admin)
- **Best_cards (aggregation)**

Operations on Redis

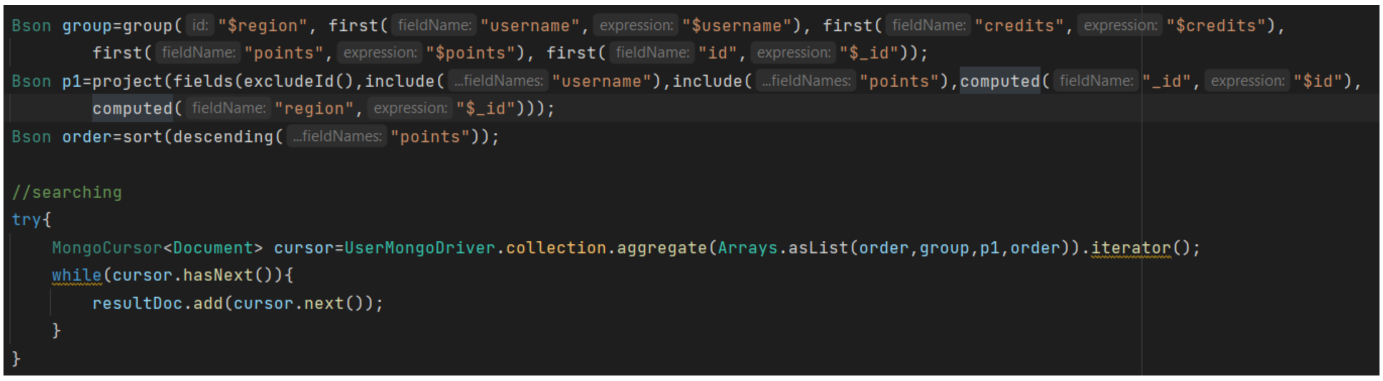
- Retrieve user’s cards collection
- Delete a card from user’s cards collection
- Add a card from user’s cards collection
- Change quantity of a card in user’s cards collection
- Delete all the cards owned by a specific User

Analytics on MongoDB

For show some interesting information we use pipelines aggregation.

1) **BEST FOR REGION** *(RankingMongoDriver)*

This aggregation shows the **User** with the highest number of *points* for each *region.*

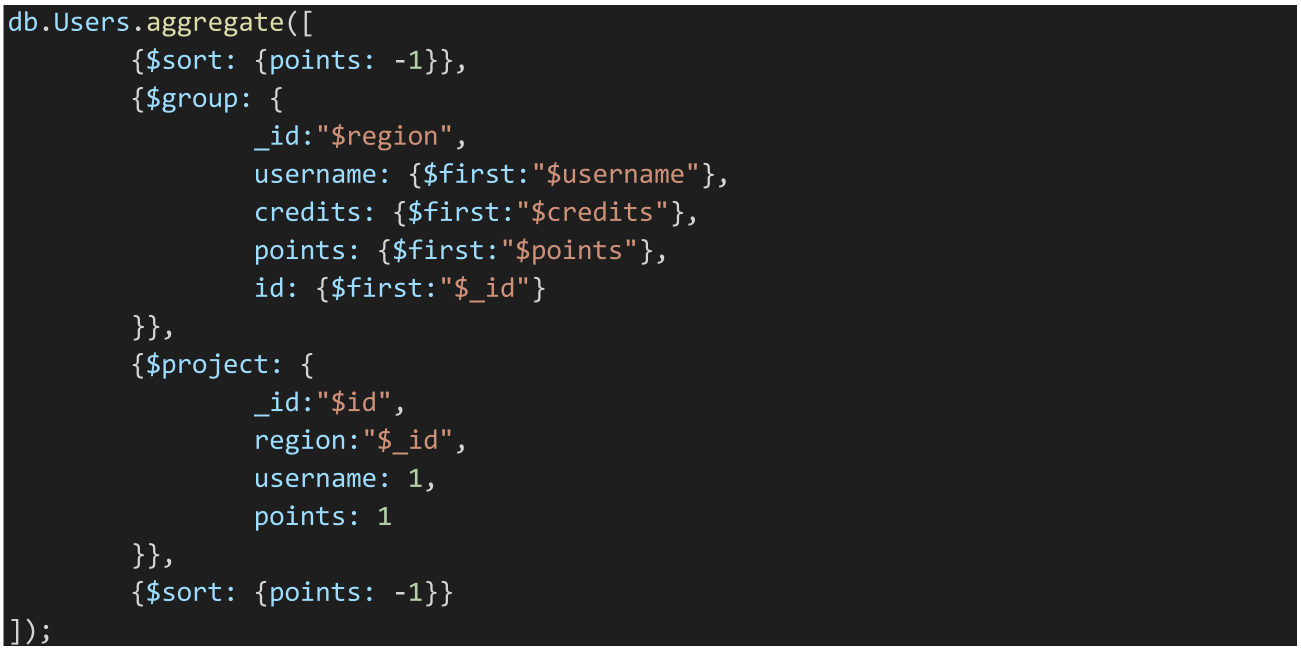


```
Bson group=group( id: "$region", first( fieldName: "username", expression: "$username"), first( fieldName: "credits", expression: "$credits"),
        first( fieldName: "points", expression: "$points"), first( fieldName: "id", expression: "$_id"));
Bson p1=project(fields(excludeId(),include( ...fieldNames: "username"),include( ...fieldNames: "points"),computed( fieldName: "_id", expression: "$id"),
        computed( fieldName: "region", expression: "$_id")));
Bson order=sort(descending( ...fieldNames: "points"));

//searching
try{
    MongoCursor<Document> cursor=UserMongoDriver.collection.aggregate(Arrays.asList(order,group,p1,order)).iterator();
    while(cursor.hasNext()){
        resultDoc.add(cursor.next());
    }
}
```

<u>Stages:</u>

a) **Sort** the document in descending order of the *points* field
b) **Group** all the documents by region and take the specified field from the first document regarding each group
c) **Project** stage for include only the fields that we need in the ranking table and for change the name to the id field after grouping
d) **Sort** the result documents to have the list of the best Users for each region sorted by points

The following image represents the pipeline aggregation on MongoDB.

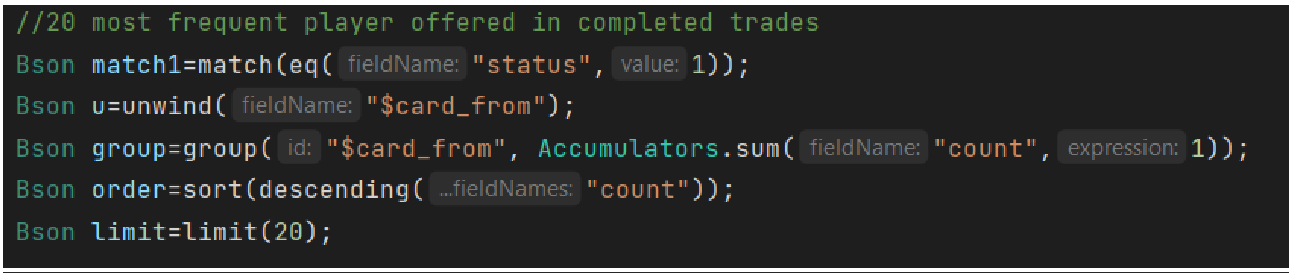


```
db.Users.aggregate([
        {$sort: {points: -1}},
        {$group: {
                _id:"$region",
                username: {$first:"$username"},
                credits: {$first:"$credits"},
                points: {$first:"$points"},
                id: {$first:"$_id"}
        }},
        {$project: {
                _id:"$id",
                region:"$_id",
                username: 1,
                points: 1
        }},
        {$sort: {points: -1}}
]);
```

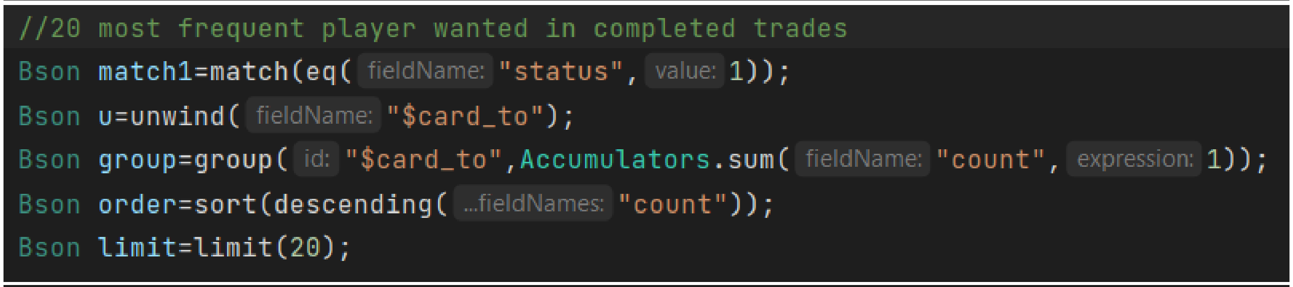
2) RETRIEVE MOST PRESENT *(TradeMongoDriver)*

A User can choose between offered and wanted cards, and this aggregation retrieve the 20 most frequently offered/wanted cards in completed Trades (completed Trades means with status equal to 1).

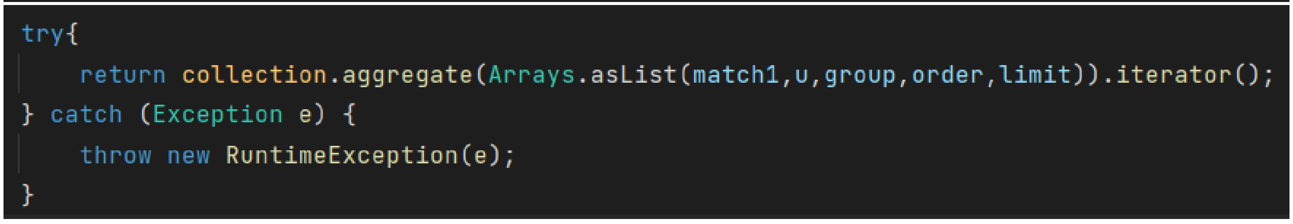
These statistics could have been managed using a specific field in Card, but it was more difficult to manage because of the consistency of this information between Trades e Cards. And inside every vector there are at maximum 3 element, so the total number of documents will not increase so much after the unwind stage.



```
//20 most frequent player offered in completed trades
Bson match1=match(eq( fieldName: "status", value: 1));
Bson u=unwind( fieldName: "$card_from");
Bson group=group( id: "$card_from", Accumulators.sum( fieldName: "count", expression: 1));
Bson order=sort(descending( ...fieldNames: "count"));
Bson limit=limit(20);
```



```
//20 most frequent player wanted in completed trades
Bson match1=match(eq( fieldName: "status", value: 1));
Bson u=unwind( fieldName: "$card_to");
Bson group=group( id: "$card_to",Accumulators.sum( fieldName: "count", expression: 1));
Bson order=sort(descending( ...fieldNames: "count"));
Bson limit=limit(20);
```



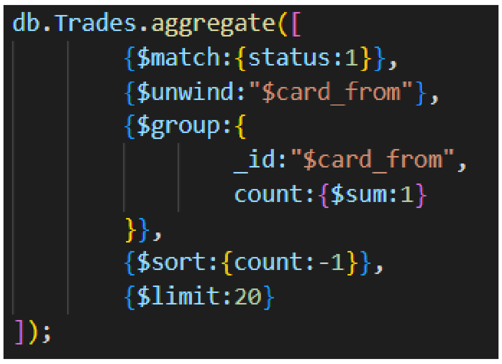
```
try{
    return collection.aggregate(Arrays.asList(match1,u,group,order,limit)).iterator();
} catch (Exception e) {
    throw new RuntimeException(e);
}
```

Stages:

a) **Match** the document with *status* equal to 1
b) **Unwind** the card in *card_from / card_to* vector to have a document for each card involved in one Trade
c) **Group** by *card_from / card_to* field that after the unwind include only one Card and a new field *count* has created for count the number of time that a specific Card is present in a Trade
d) **Sort** in descending order by *count* field
e) **Limit** to the first 20 document to find the 20 most present Card

The following image represents the pipeline aggregation on MongoDB, where the User has chosen to see the statistics regarding the offered Card.

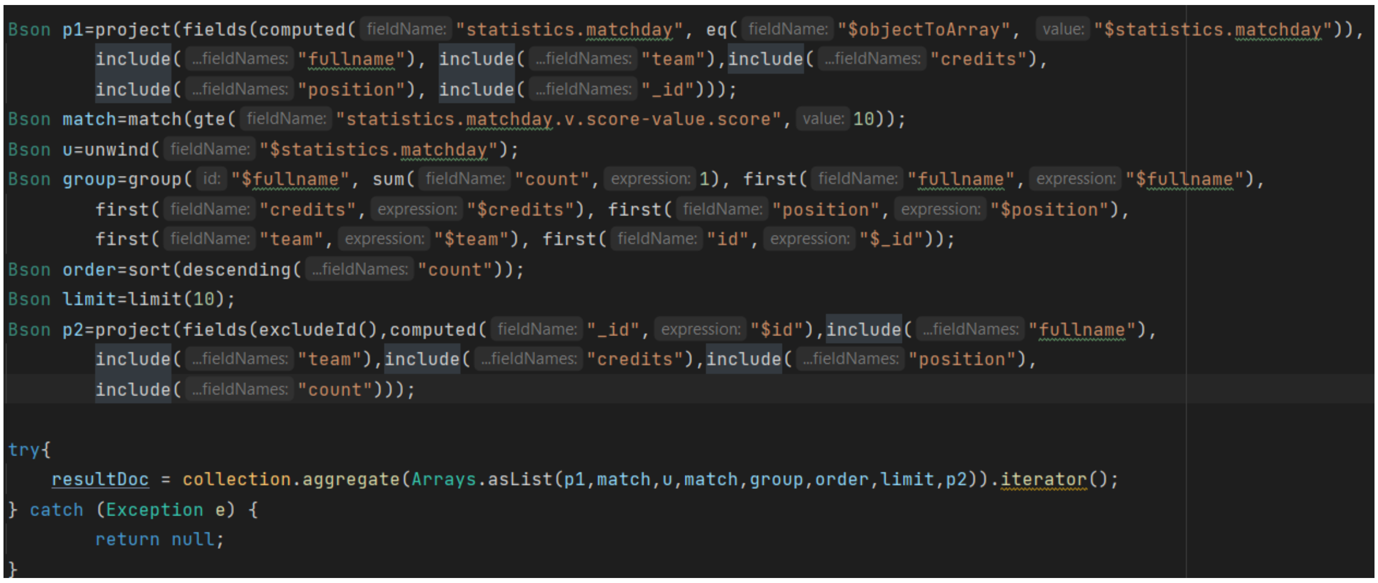
For wanted Card is the same, but with $card_to instead of $card_from.



```
db.Trades.aggregate([
        {$match:{status:1}},
        {$unwind:"$card_from"},
        {$group:{
                _id:"$card_from",
                count:{$sum:1}
        }},
        {$sort:{count:-1}},
        {$limit:20}
]);
```

3) **BEST CARDS FOR SCORE** *(CardMongoDriver)*

This aggregation shows the 10 Cards, representing a player, who take more time a matchday score greater than 10.



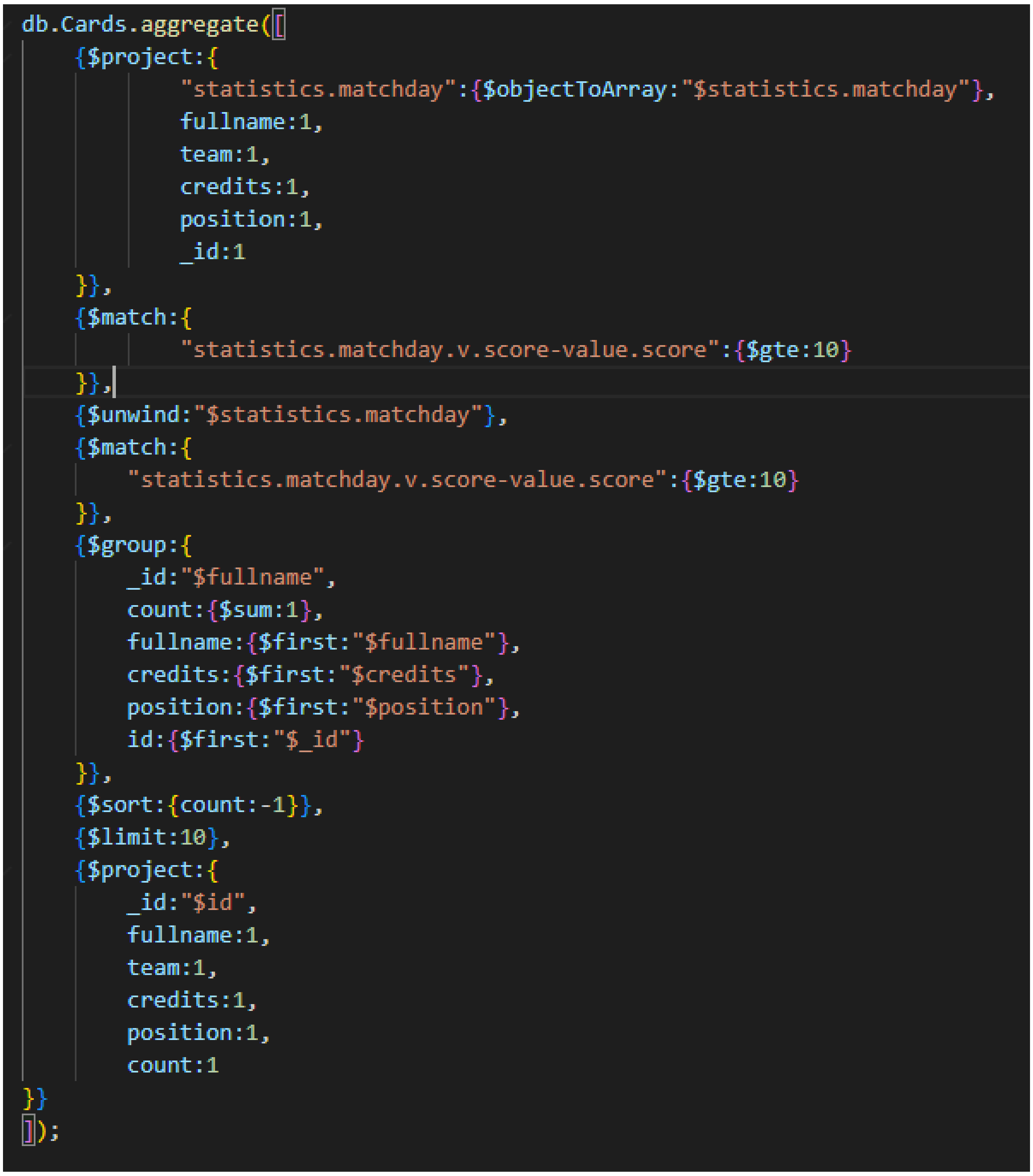
```
Bson p1=project(fields(computed( fieldName: "statistics.matchday", eq( fieldName: "$objectToArray",  value: "$statistics.matchday")),
        include( ...fieldNames: "fullname"), include( ...fieldNames: "team"),include( ...fieldNames: "credits"),
        include( ...fieldNames: "position"), include( ...fieldNames: "_id")));
Bson match=match(gte( fieldName: "statistics.matchday.v.score-value.score", value: 10));
Bson u=unwind( fieldName: "$statistics.matchday");
Bson group=group( id: "$fullname", sum( fieldName: "count", expression: 1), first( fieldName: "fullname", expression: "$fullname"),
        first( fieldName: "credits", expression: "$credits"), first( fieldName: "position", expression: "$position"),
        first( fieldName: "team", expression: "$team"), first( fieldName: "id", expression: "$_id"));
Bson order=sort(descending( ...fieldNames: "count"));
Bson limit=limit(10);
Bson p2=project(fields(excludeId(),computed( fieldName: "_id", expression: "$id"),include( ...fieldNames: "fullname"),
        include( ...fieldNames: "team"),include( ...fieldNames: "credits"),include( ...fieldNames: "position"),
        include( ...fieldNames: "count")));

try{
    resultDoc = collection.aggregate(Arrays.asList(p1,match,u,match,group,order,limit,p2)).iterator();
} catch (Exception e) {
        return null;
}
```

Stages:

a) **Project** the document to transform the object *statistics.matchday* in an array, visible for the unwind.
b) **Match** all the document which has at least one matchday in which the score is greater than 10.
c) **Unwind** the vector *statistics.matchday* to obtain a document for each match of each football player.
d) **Match** only the document obtained after the unwind with score greater than 10
e) **Group** all the documents by *fullname*, we are grouping by each football player to count how many documents with score greater than 10 he has.

f) **Sort** by the new field *count*.
g) **Limit** at the first 10 documents.
h) **Project** to obtain the necessary field with the right name for print the result on the Shop page (*ShopController*)

The following image represents the pipeline aggregation on MongoDB.



```
db.Cards.aggregate([
    {$project:{
            "statistics.matchday":{$objectToArray:"$statistics.matchday"},
            fullname:1,
            team:1,
            credits:1,
            position:1,
            _id:1
    }},
    {$match:{
            "statistics.matchday.v.score-value.score":{$gte:10}
    }},
    {$unwind:"$statistics.matchday"},
    {$match:{
        "statistics.matchday.v.score-value.score":{$gte:10}
    }},
    {$group:{
        _id:"$fullname",
        count:{$sum:1},
        fullname:{$first:"$fullname"},
        credits:{$first:"$credits"},
        position:{$first:"$position"},
        id:{$first:"$_id"}
    }},
    {$sort:{count:-1}},
    {$limit:10},
    {$project:{
        _id:"$id",
        fullname:1,
        team:1,
        credits:1,
        position:1,
        count:1
}}
]);
```

In the previous pipeline aggregation we have to use $objectToArray to converts the document statistics.matchday, in which we have all the matchday statistics with key “matchdayK” and for value, all the statistics for that matchday, in a vector, usable by the unwind.

In order to avoid this last operation, we should have put this score values in a specific array field (an array with 38 values, one for each match), but we preferred to keep them after the statistics inside the matchdayK document.

Also, if we implemented this change we will have used the unwind anyway, to retrieve a document for each score associated to his own player’s card.

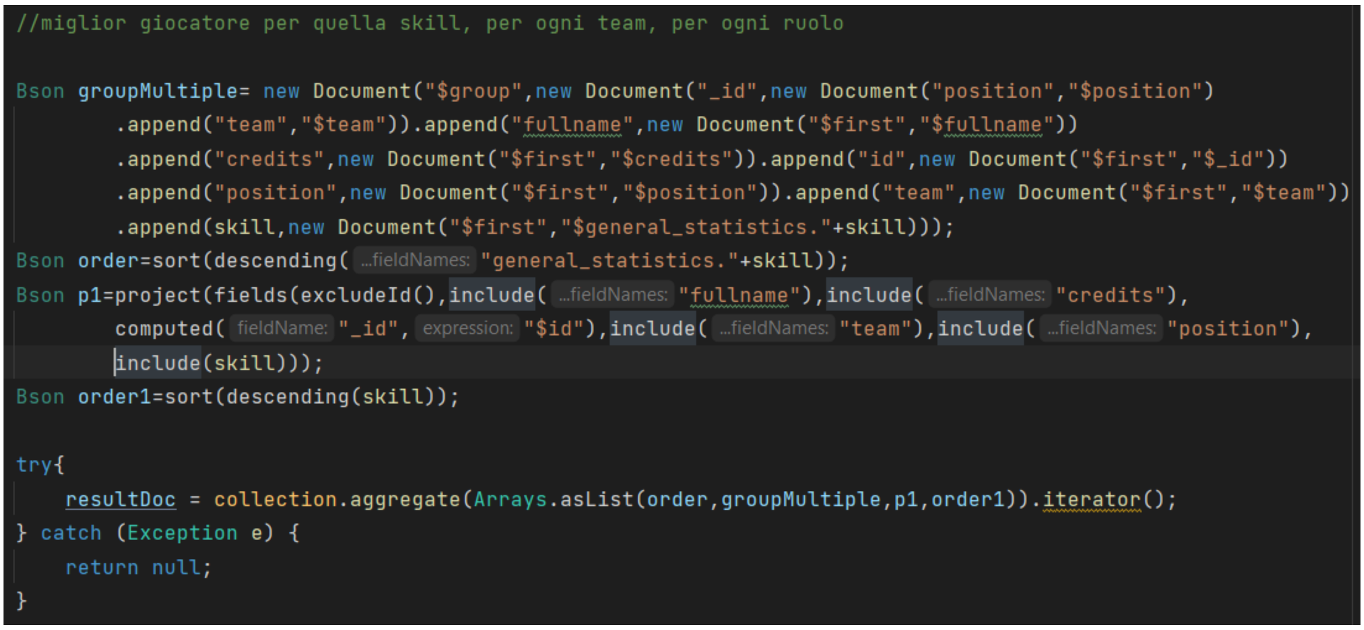
4) **BEST CARDS FOR SKILL** *(CardMongoDriver)*

For skill we mean a specific card’s general statistics.

At the bottom of *Shop Page*, we have three choice boxes:

1. For choose the skill (general statistics)
2. For choose the role of the player
3. For choose the team of the player

4a) When a User sets only the skill choice box, the application will show to him the results of this aggregation pipeline:



```
//miglior giocatore per quella skill, per ogni team, per ogni ruolo

Bson groupMultiple= new Document("$group",new Document("_id",new Document("position","$position")
        .append("team","$team")).append("fullname",new Document("$first","$fullname"))
        .append("credits",new Document("$first","$credits")).append("id",new Document("$first","$_id"))
        .append("position",new Document("$first","$position")).append("team",new Document("$first","$team"))
        .append(skill,new Document("$first","$general_statistics."+skill)));
Bson order=sort(descending( ...fieldNames: "general_statistics."+skill));
Bson p1=project(fields(excludeId(),include( ...fieldNames: "fullname"),include( ...fieldNames: "credits"),
        computed( fieldName: "_id", expression: "$id"),include( ...fieldNames: "team"),include( ...fieldNames: "position"),
        include(skill)));
Bson order1=sort(descending(skill));

try{
    resultDoc = collection.aggregate(Arrays.asList(order,groupMultiple,p1,order1)).iterator();
} catch (Exception e) {
    return null;
}
```

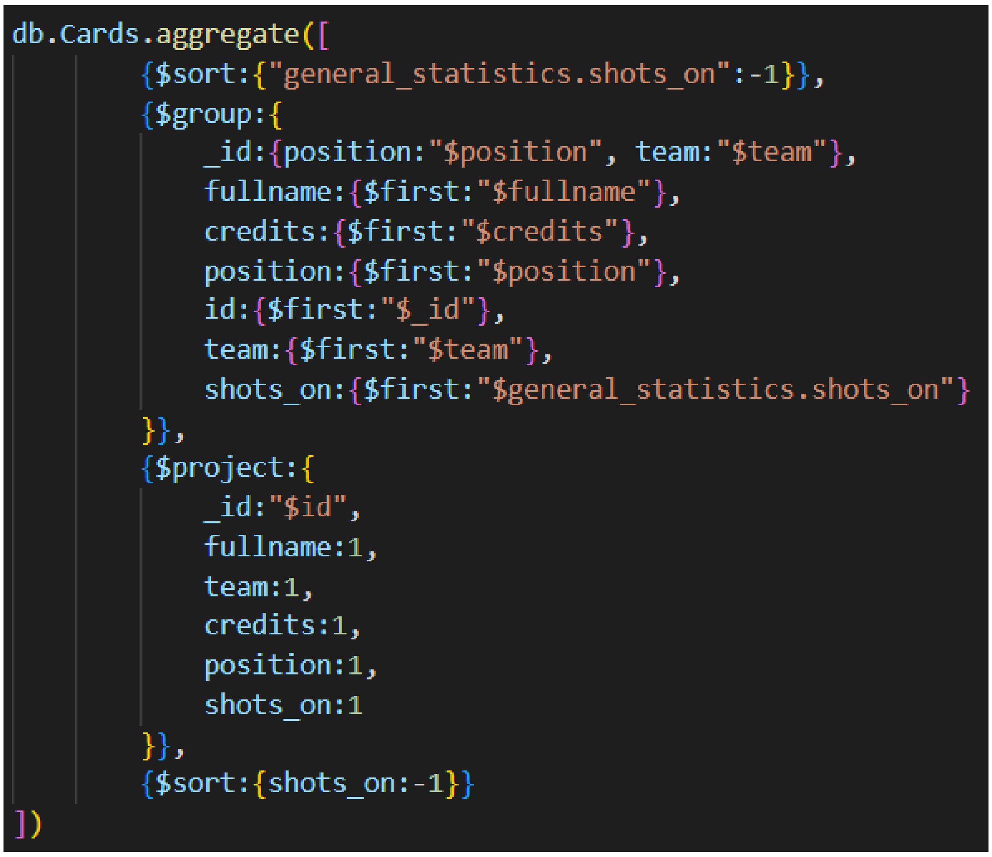
This aggregation shows the best player’s card for that specific skill, for each team, for each role.

So, in the results we will have 80 documents, one for each role (Goalkeeper, Defender, Midfielder, Attacker), for each team (20 Serie A teams).

<u>Stage:</u>

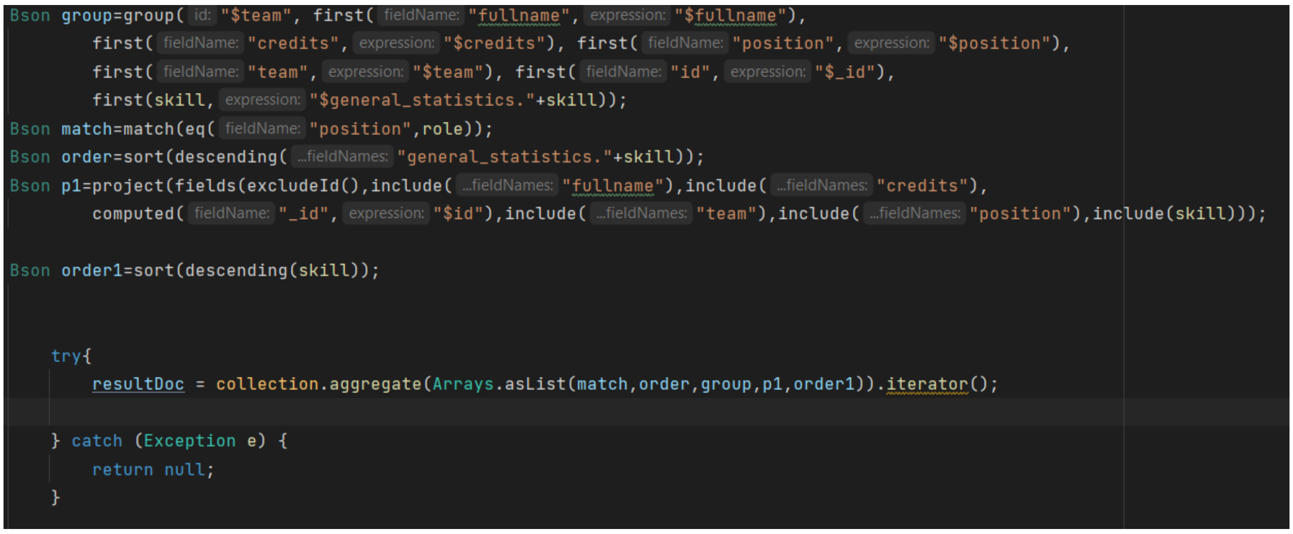
a) **Sort** all the documents by descending order of that skill.
b) **Group** by team and position and take the attributes for the first document for each group.
c) **Project** stage for computed field about the cards' id in "_id" and include all the other specified fields.
d) **Sort** all the documents by descending order of that skill, to have the best player's card for that specific skill, for each team and role, in descending order of that skill.

The following image represents the pipeline aggregation on MongoDB:



```
db.Cards.aggregate([
        {$sort:{"general_statistics.shots_on":-1}},
        {$group:{
            _id:{position:"$position", team:"$team"},
            fullname:{$first:"$fullname"},
            credits:{$first:"$credits"},
            position:{$first:"$position"},
            id:{$first:"$_id"},
            team:{$first:"$team"},
            shots_on:{$first:"$general_statistics.shots_on"}
        }},
        {$project:{
            _id:"$id",
            fullname:1,
            team:1,
            credits:1,
            position:1,
            shots_on:1
        }},
        {$sort:{shots_on:-1}}
])
```

4b) When a User sets a skill and the role of the player, the application will show to him the results of this aggregation:

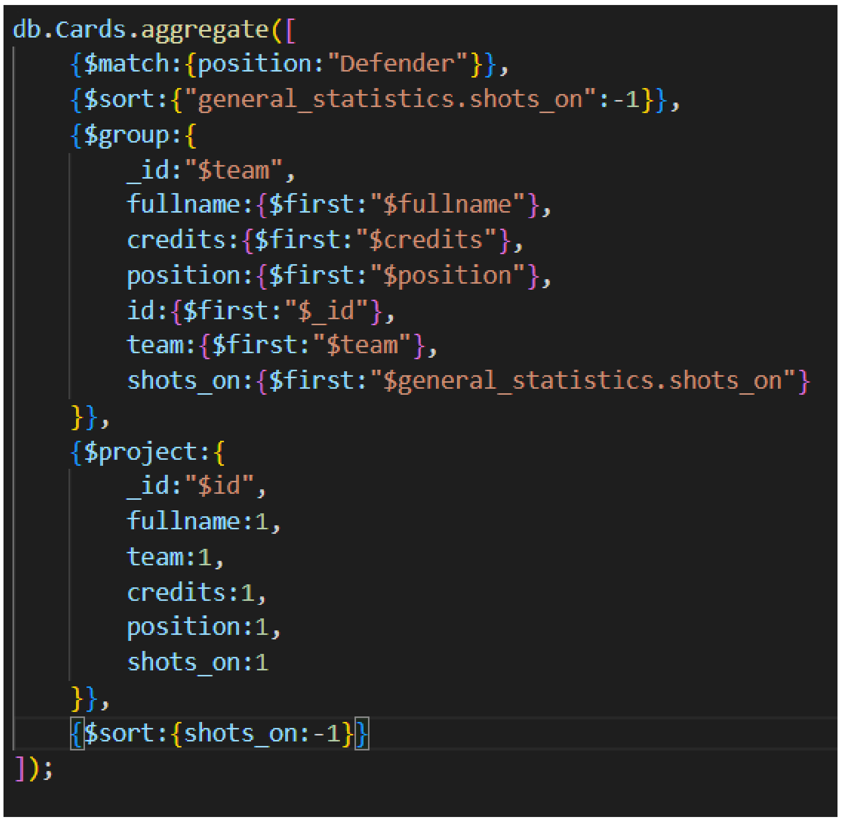


```
Bson group=group( id: "$team", first( fieldName: "fullname", expression: "$fullname"),
        first( fieldName: "credits", expression: "$credits"), first( fieldName: "position", expression: "$position"),
        first( fieldName: "team", expression: "$team"), first( fieldName: "id", expression: "$_id"),
        first(skill, expression: "$general_statistics."+skill));
Bson match=match(eq( fieldName: "position",role));
Bson order=sort(descending( ...fieldNames: "general_statistics."+skill));
Bson p1=project(fields(excludeId(),include( ...fieldNames: "fullname"),include( ...fieldNames: "credits"),
        computed( fieldName: "_id", expression: "$id"),include( ...fieldNames: "team"),include( ...fieldNames: "position"),include(skill)));

Bson order1=sort(descending(skill));


    try{
        resultDoc = collection.aggregate(Arrays.asList(match,order,group,p1,order1)).iterator();

    } catch (Exception e) {
        return null;
    }
```

This aggregation is very similar to the previous one (4a), with the addition of a **match** stage before the group stage.

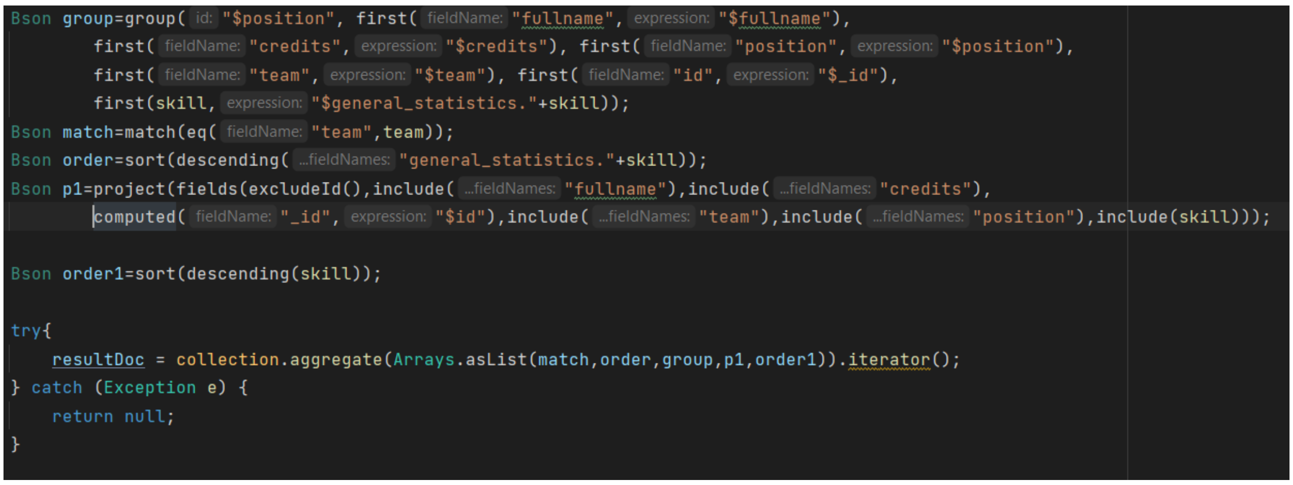
The other difference is that the **group** stage is made only by the team of a Card, and not by team and role like in the previous one, because now the role is specified.

The following image represents the pipeline aggregation on MongoDB (case of skill=shots_on and role=Defender):



```
db.Cards.aggregate([
    {$match:{position:"Defender"}},
    {$sort:{"general_statistics.shots_on":-1}},
    {$group:{
        _id:"$team",
        fullname:{$first:"$fullname"},
        credits:{$first:"$credits"},
        position:{$first:"$position"},
        id:{$first:"$_id"},
        team:{$first:"$team"},
        shots_on:{$first:"$general_statistics.shots_on"}
    }},
    {$project:{
        _id:"$id",
        fullname:1,
        team:1,
        credits:1,
        position:1,
        shots_on:1
    }},
    {$sort:{shots_on:-1}}
]);
```

4c) When a User sets a skill and the team of the player, the application will show to him the results of this aggregation:

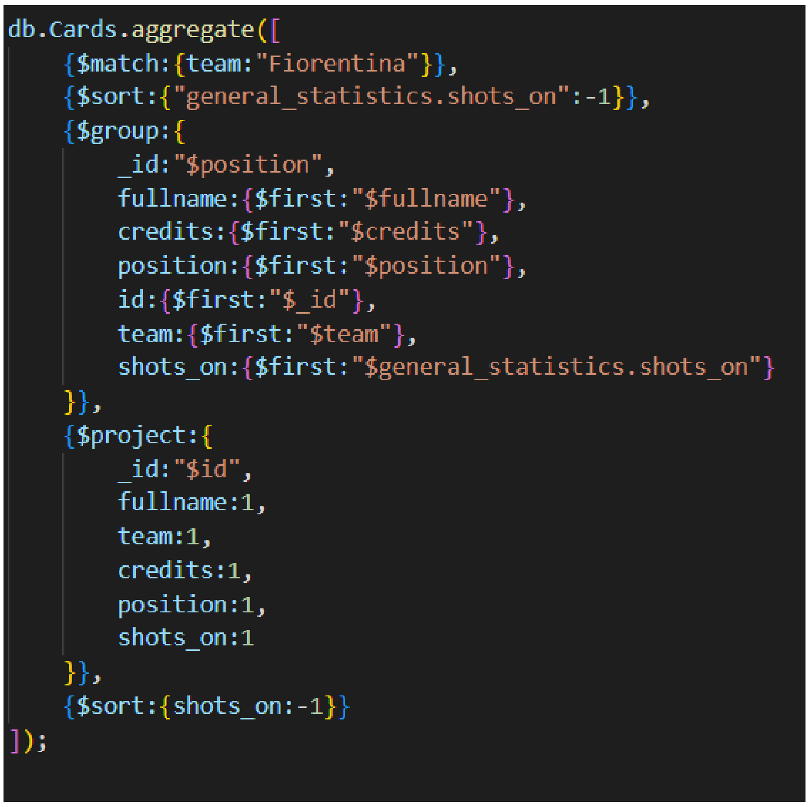


```
Bson group=group( id: "$position", first( fieldName: "fullname", expression: "$fullname"),
        first( fieldName: "credits", expression: "$credits"), first( fieldName: "position", expression: "$position"),
        first( fieldName: "team", expression: "$team"), first( fieldName: "id", expression: "$_id"),
        first(skill, expression: "$general_statistics."+skill));
Bson match=match(eq( fieldName: "team",team));
Bson order=sort(descending( ...fieldNames: "general_statistics."+skill));
Bson p1=project(fields(excludeId(),include( ...fieldNames: "fullname"),include( ...fieldNames: "credits"),
        computed( fieldName: "_id", expression: "$id"),include( ...fieldNames: "team"),include( ...fieldNames: "position"),include(skill)));

Bson order1=sort(descending(skill));

try{
    resultDoc = collection.aggregate(Arrays.asList(match,order,group,p1,order1)).iterator();
} catch (Exception e) {
    return null;
}
```

This aggregation is very similar to the previous one (4b), with the difference of the **match** stage, that now regards the team instead of the role

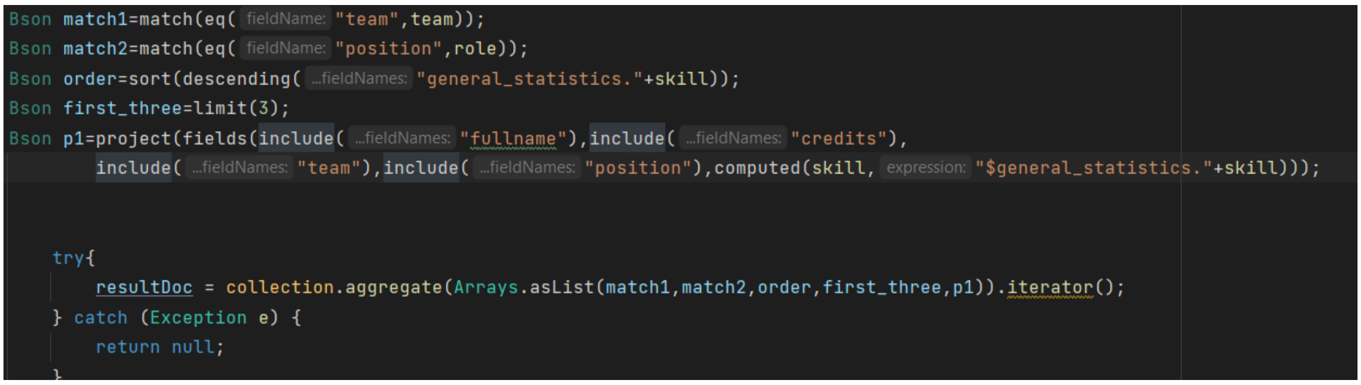
The other difference is that the **group** stage is made by the position of a Card, because now the team is specified.

The following image represents the pipeline aggregation on MongoDB (case of skill=shots_on and team=Fiorentina):



```
db.Cards.aggregate([
    {$match:{team:"Fiorentina"}},
    {$sort:{"general_statistics.shots_on":-1}},
    {$group:{
        _id:"$position",
        fullname:{$first:"$fullname"},
        credits:{$first:"$credits"},
        position:{$first:"$position"},
        id:{$first:"$_id"},
        team:{$first:"$team"},
        shots_on:{$first:"$general_statistics.shots_on"}
    }},
    {$project:{
        _id:"$id",
        fullname:1,
        team:1,
        credits:1,
        position:1,
        shots_on:1
    }},
    {$sort:{shots_on:-1}}
]);
```

4d) When a User sets a skill, the team and the role of the player, the application will show to him the results of this query:



```
Bson match1=match(eq( fieldName: "team",team));
Bson match2=match(eq( fieldName: "position",role));
Bson order=sort(descending( ...fieldNames: "general_statistics."+skill));
Bson first_three=limit(3);
Bson p1=project(fields(include( ...fieldNames: "fullname"),include( ...fieldNames: "credits"),
        include( ...fieldNames: "team"),include( ...fieldNames: "position"),computed(skill, expression: "$general_statistics."+skill)));


    try{
        resultDoc = collection.aggregate(Arrays.asList(match1,match2,order,first_three,p1)).iterator();
    } catch (Exception e) {
        return null;
    }
```

4e) If a User sets only the team and the position of a player, there will be only two stages of **match** regarding the two specified fields.

4f) If a User sets only the team, or the position, of a player, there will be a simple query who retrieve all the cards with that position or team specified.

Indexes

We analyze the performance of read query with and without Indexes, and we decide to add some Indexes to improve the reading on the MongoDB collections.

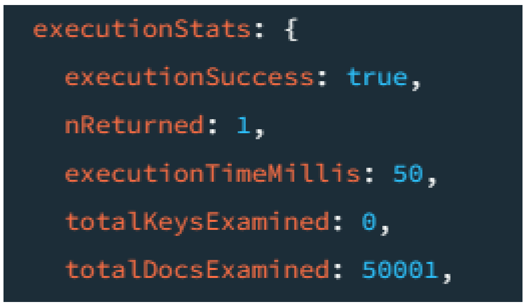
All tests were made in local.

Users **collection**

We need very often find operation on the username field of Users, so we analyzed the query on that field regarding a specific User.

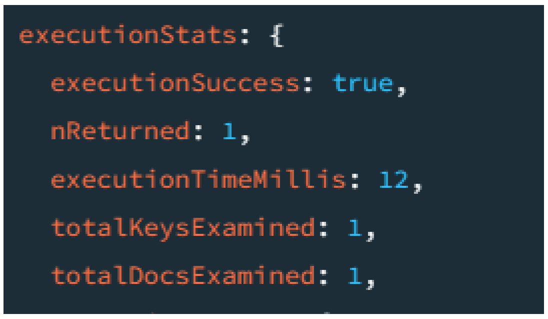
Before Indexing

db.Users.find({username:"nootvas88"}).explain("executionStats")



```
executionStats: {
  executionSuccess: true,
  nReturned: 1,
  executionTimeMillis: 50,
  totalKeysExamined: 0,
  totalDocsExamined: 50001,
```

After Indexing → db.Users.createIndex({username:1})

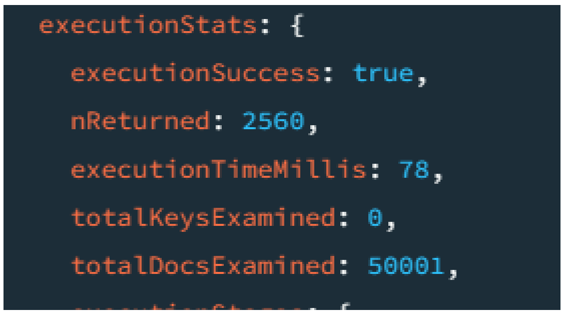


```
executionStats: {
  executionSuccess: true,
  nReturned: 1,
  executionTimeMillis: 12,
  totalKeysExamined: 1,
  totalDocsExamined: 1,
```

We make also some query searching on the User’s region, so we analyzed the possibility to put an Indexes also on the region field.

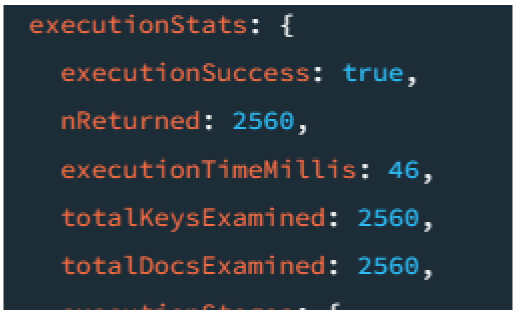
Before Indexing

db.Users.find({region:”Lombardy”}).explain(“executionStats”)



```
executionStats: {
  executionSuccess: true,
  nReturned: 2560,
  executionTimeMillis: 78,
  totalKeysExamined: 0,
  totalDocsExamined: 50001,
```

After Indexing → db.Users.createIndex({region:1})



```
executionStats: {
  executionSuccess: true,
  nReturned: 2560,
  executionTimeMillis: 46,
  totalKeysExamined: 2560,
  totalDocsExamined: 2560,
```

The improvement will be greater when the number of Users will increase, so we decide to **set these two Indexes**.

Trades collection

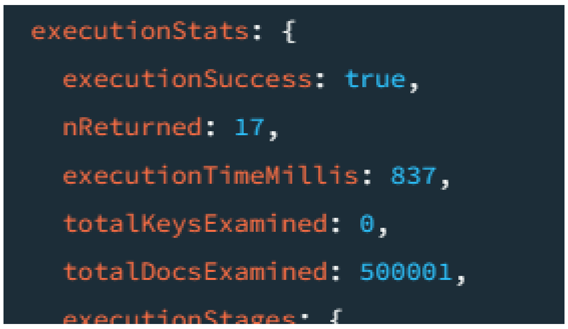
We expect a lot of Trades, so we need to improve the performance of reading trade collection.

A User can see his/her own Trades for handle them, for example deleting a Trade made by himself/herself.

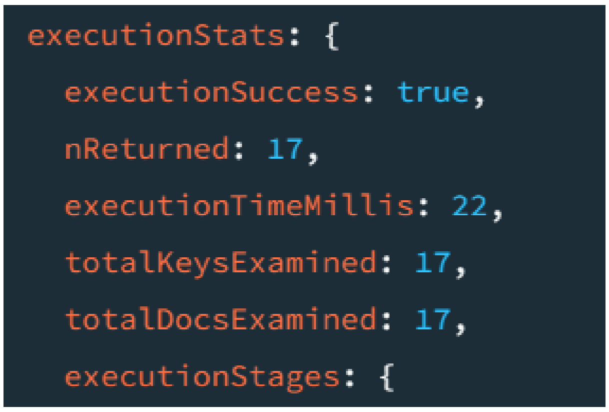
So, we analyze the search by the field user_from, made with and without Index.

Before Indexing

db.Trades.find({user_from:"nootvas88"}).explain("executionStats")


```
executionStats: {
  executionSuccess: true,
  nReturned: 17,
  executionTimeMillis: 837,
  totalKeysExamined: 0,
  totalDocsExamined: 500001,
  executionStages: {
```

After Indexing →db.Trades.createIndex({user_from:1})

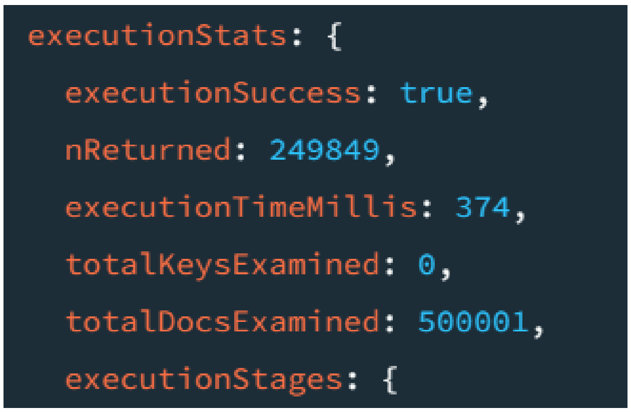

```
executionStats: {
  executionSuccess: true,
  nReturned: 17,
  executionTimeMillis: 22,
  totalKeysExamined: 17,
  totalDocsExamined: 17,
  executionStages: {
```

We decide **to introduce an Index on user_from** in Trades collection.

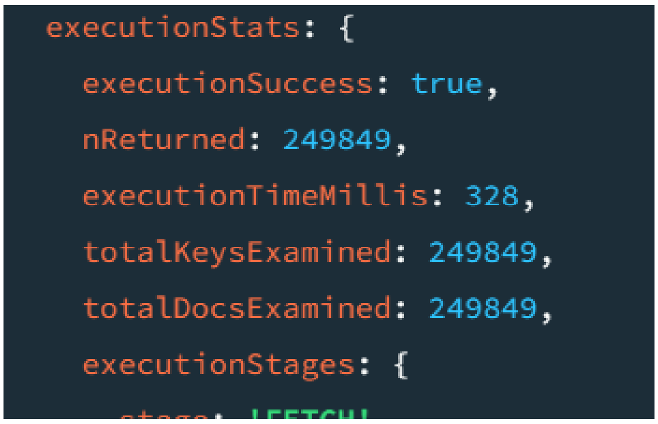
The other frequent find operation is made on the status field (1: accepted trade, 0: pending trade), we want to show to the User only the pending Trades, and we keep all the accepted Trades in the database only for analytics and statistics.

Before Indexing

db.Trades.find({status:0}).explain("executionStats")


```
executionStats: {
  executionSuccess: true,
  nReturned: 249849,
  executionTimeMillis: 374,
  totalKeysExamined: 0,
  totalDocsExamined: 500001,
  executionStages: {
```

After Indexing → db.Trades.createIndex({status:1})



```
executionStats: {
  executionSuccess: true,
  nReturned: 249849,
  executionTimeMillis: 328,
  totalKeysExamined: 249849,
  totalDocsExamined: 249849,
  executionStages: {
    stage: 'FETCH'
```

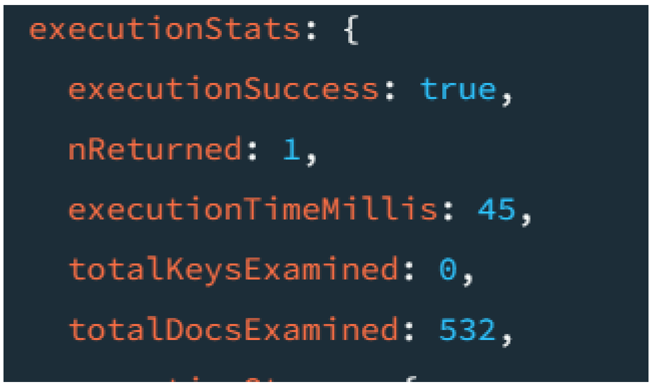
Even if we have little changes on execution time, we can see that in this way we have decrease the number of *totalDocsExamined* and seen the high number of Trades for this type of application we decided to **add an Index on field status**.

Cards collection

On cards collection we analyzed the possibility to add Indexes on fullname, role or team.

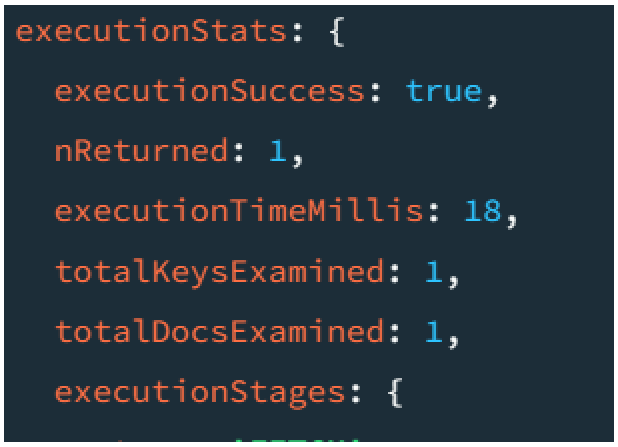
Before Indexing

db.Cards.find({fullname: “L.Torreira”}).explain(“executionStats”)



```
executionStats: {
  executionSuccess: true,
  nReturned: 1,
  executionTimeMillis: 45,
  totalKeysExamined: 0,
  totalDocsExamined: 532,
```

<u>After Indexing</u> → db.Cards.createIndex({fullname:1})



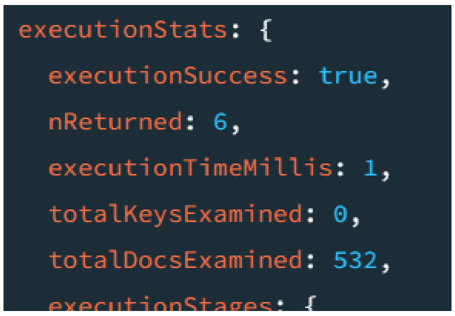
```
executionStats: {
  executionSuccess: true,
  nReturned: 1,
  executionTimeMillis: 18,
  totalKeysExamined: 1,
  totalDocsExamined: 1,
  executionStages: {
```

Even if we have a limited number of Cards, we decide to **introduce an Index on fullname** field see the result before and after the Index.

Then we try the query which a User can find a Card by team and position.

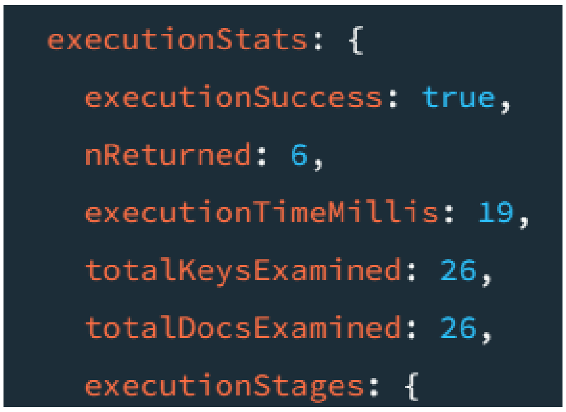
<u>Before Indexing</u>

db.Cards.find({position:"Attacker", team:"Fiorentina"}).explain("executionStats")



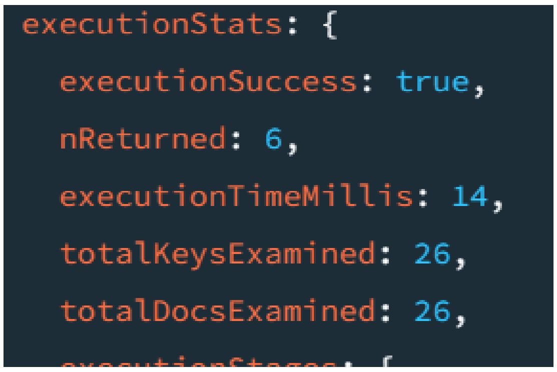
```
executionStats: {
  executionSuccess: true,
  nReturned: 6,
  executionTimeMillis: 1,
  totalKeysExamined: 0,
  totalDocsExamined: 532,
  executionStages: {
```

<u>After Indexing</u> → db.Cards.createIndex({team:1})



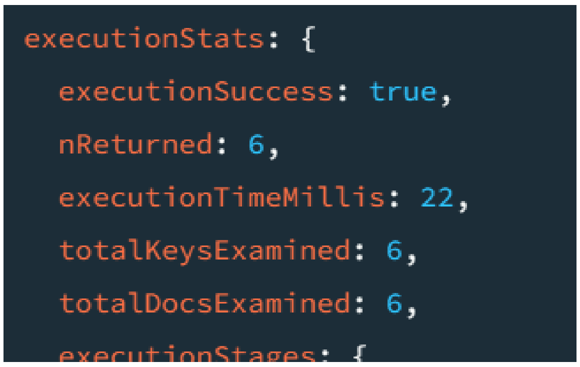
```
executionStats: {
  executionSuccess: true,
  nReturned: 6,
  executionTimeMillis: 19,
  totalKeysExamined: 26,
  totalDocsExamined: 26,
  executionStages: {
```

<u>After Indexing</u> → db.Cards.createIndex({position:1})



```
executionStats: {
  executionSuccess: true,
  nReturned: 6,
  executionTimeMillis: 14,
  totalKeysExamined: 26,
  totalDocsExamined: 26,
```

<u>After Indexing</u> → db.Cards.createIndex({position:1, team:1})



```
executionStats: {
  executionSuccess: true,
  nReturned: 6,
  executionTimeMillis: 22,
  totalKeysExamined: 6,
  totalDocsExamined: 6,
  executionStages: {
```

None of these Indexes would improve our performance, so we decided to **not** insert an index for team and position.

Sharding, a possible implementation

For the *users* collection we can consider the region field as a shard key and a partitioning method based on list. In this way user’s information are segmented by country on the cluster and each server, locally deployed, would serve the incoming requests for the users’ region. For regions that produce a critical amount of requests we should consider a higher number of servers and add some more replicas.

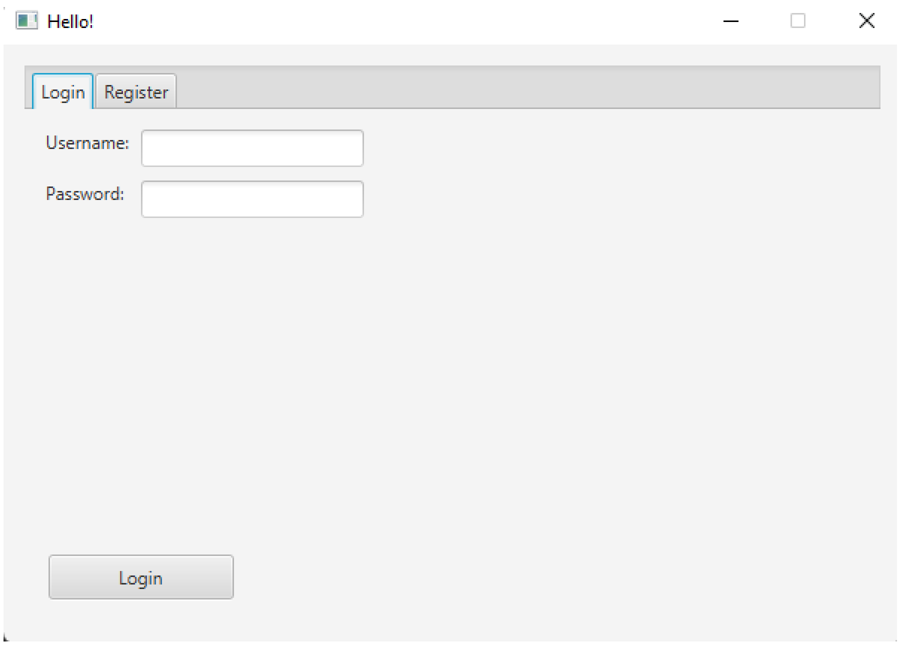
For the *trades* collection we can consider the *_id* as a shard key and a partitioning method based on Hashing. Trades information are not easily divisible by other means, but a Hashing algorithm on the *_id* field makes it possible.

For the cards collection we thought that a Sharding method is not that useful, as its size is limited and known. However, considering for example the role as a shard key, a partitioning method based on list is possible.

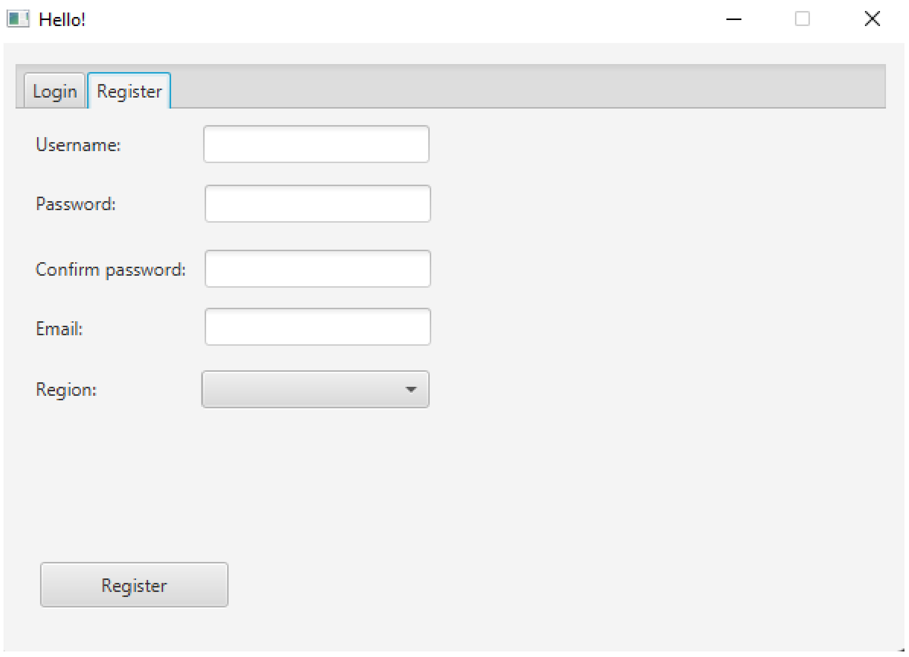
User Manual

Login/Registration and Home Page

The first page shown to the user is the login/registration page (or *Hello! Page*) where he can login into his account or create a new one.

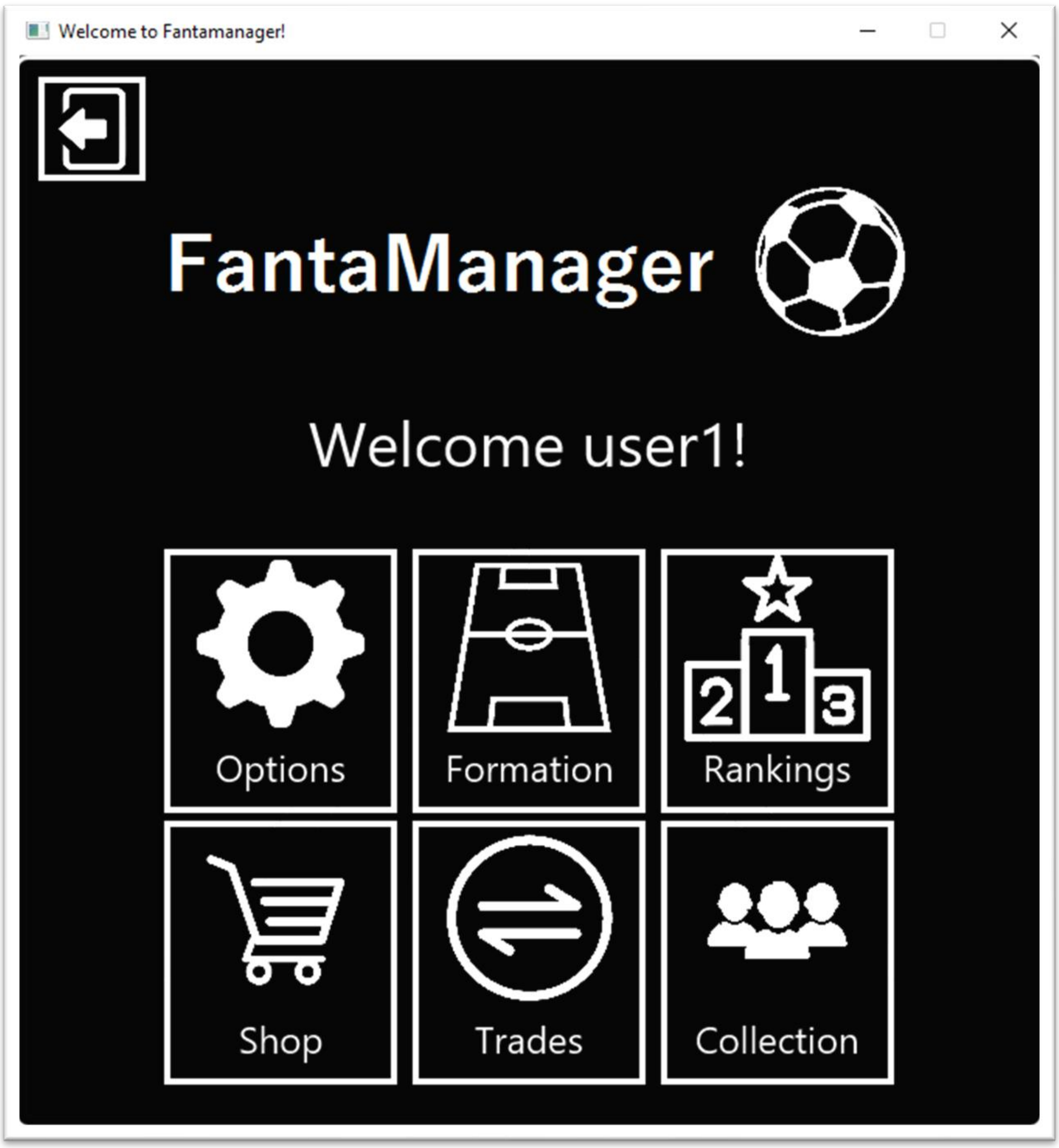


Some checks are also added to registration fields, such as a confirm password checks or an email input check.



After the login/registration phase is completed, a loading page will show up.

After all information is retrieved, the user enters the *Home Page* where he can choose what function of the application to use:



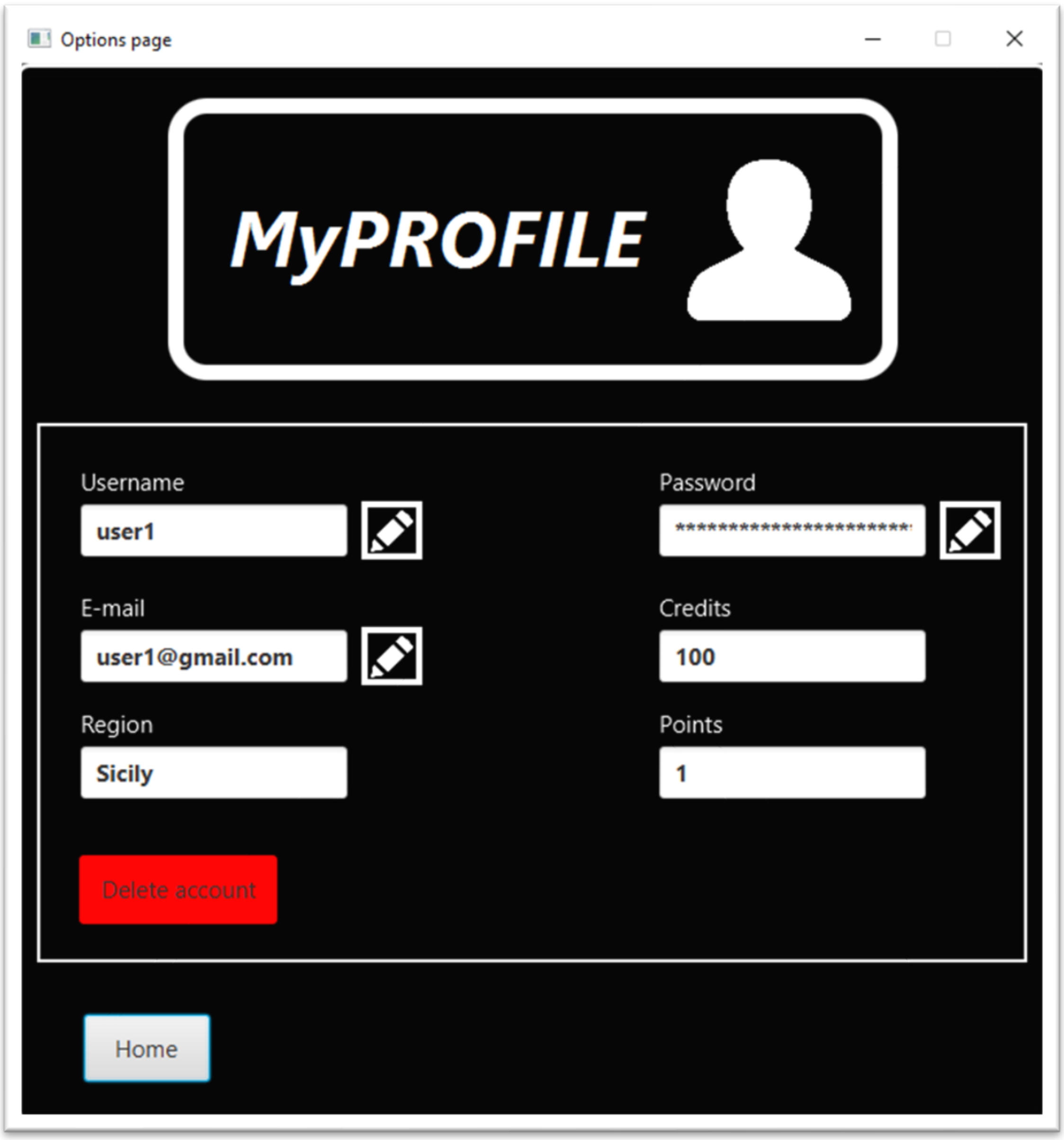
Every button represents a section of the application, such as Options, Ranking and so on...

The button in the upper left corner of the page permits the *logout* operation and go back to the login/registration phase.

Options Page

This is the page where a user can view and edit his information, such as username, password, and email. All fields with a pencil icon next to it can be edited. Some restraints must be respected when editing an attribute, such as: the new username must be unique, and the email input must be a real email string.

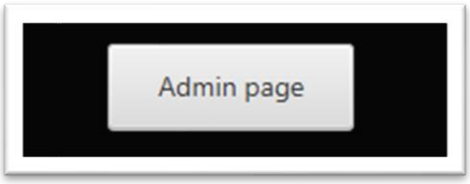
A user can also delete his account by clicking on the red button. After that, a confirm option appears and, if the user decides to continue, all his information will be deleted from the databases and he/she will be logged out from the application, returning to the login/registration phase.

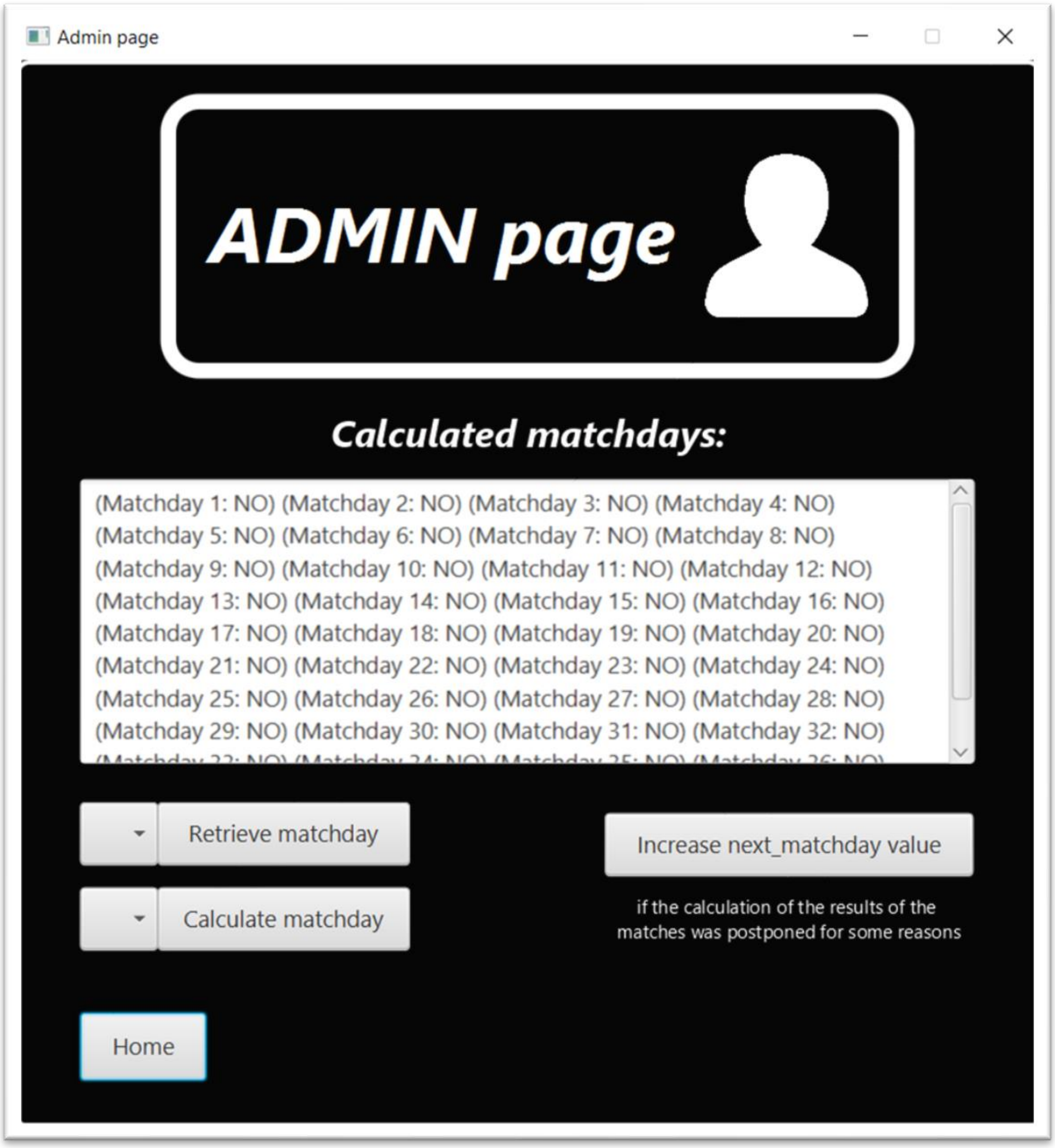


Username: user1
Password: ***********************
E-mail: user1@gmail.com
Credits: 100
Region: Sicily
Points: 1

Admin Page

If the admin is logged in, starting from the *Options Page* he can access a special page called *Admin page* by clicking the button on the bottom right of the page. This button is visible only by the admin.





(Matchday 1: NO) (Matchday 2: NO) (Matchday 3: NO) (Matchday 4: NO)
(Matchday 5: NO) (Matchday 6: NO) (Matchday 7: NO) (Matchday 8: NO)
(Matchday 9: NO) (Matchday 10: NO) (Matchday 11: NO) (Matchday 12: NO)
(Matchday 13: NO) (Matchday 14: NO) (Matchday 15: NO) (Matchday 16: NO)
(Matchday 17: NO) (Matchday 18: NO) (Matchday 19: NO) (Matchday 20: NO)
(Matchday 21: NO) (Matchday 22: NO) (Matchday 23: NO) (Matchday 24: NO)
(Matchday 25: NO) (Matchday 26: NO) (Matchday 27: NO) (Matchday 28: NO)
(Matchday 29: NO) (Matchday 30: NO) (Matchday 31: NO) (Matchday 32: NO)

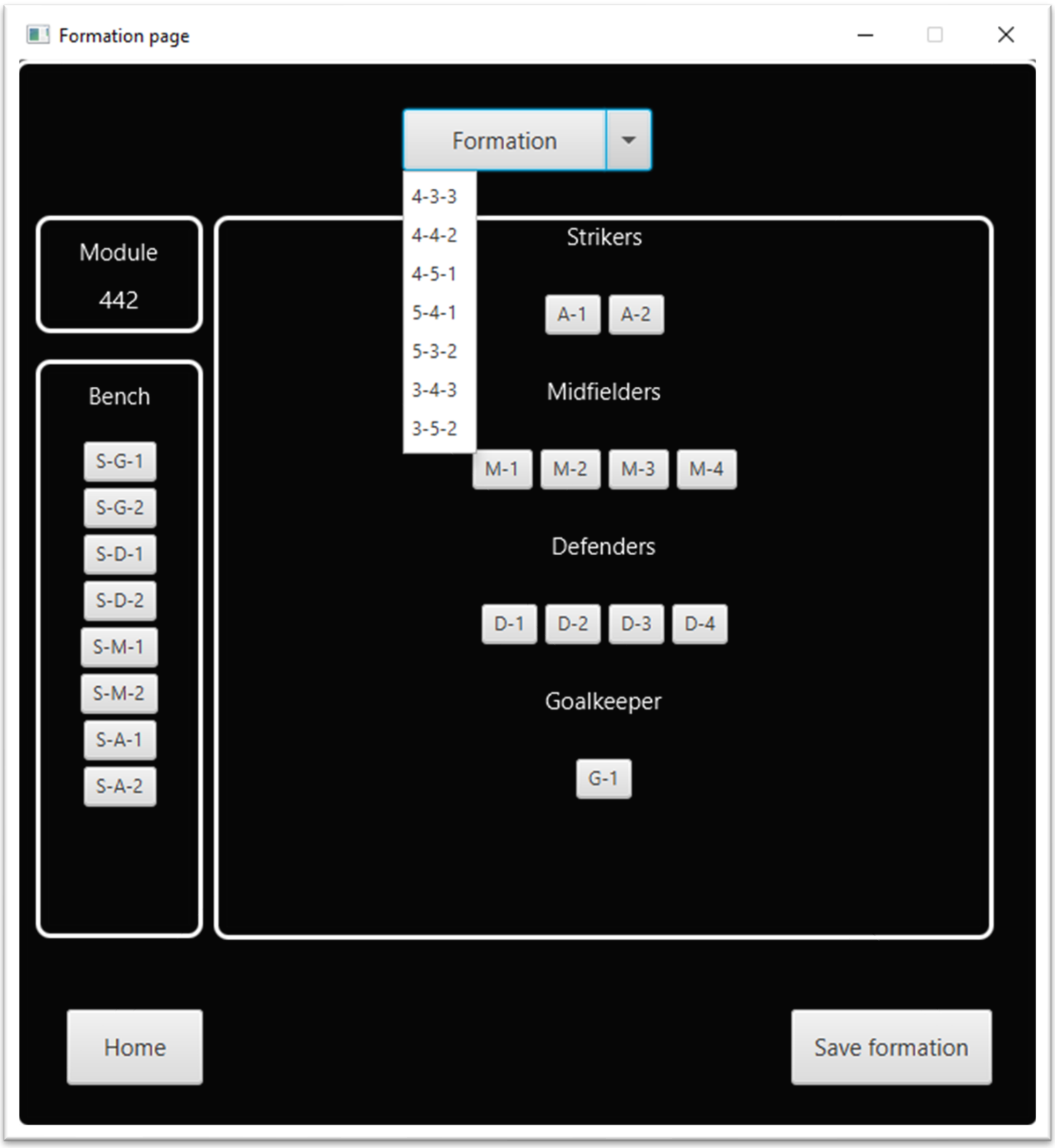
if the calculation of the results of the matches was postponed for some reasons

From this page the admin can easily manage all information about the matchday and his calculation. The admin can view a list of all matchday and if they are calculated or not. The admin can increase the *next_matchday* variable's value and retrieve and calculate a specific matchday.

Formation Page

In this page a user can set up his formation for the current matchday. He must choose a module from the list and then select 19 different cards from his collection to set up his squad. After that, he can save his formation on the database. The formation is also saved locally so if the user exits and re-enters the page the application remembers the last settled formation, even if is unfinished.

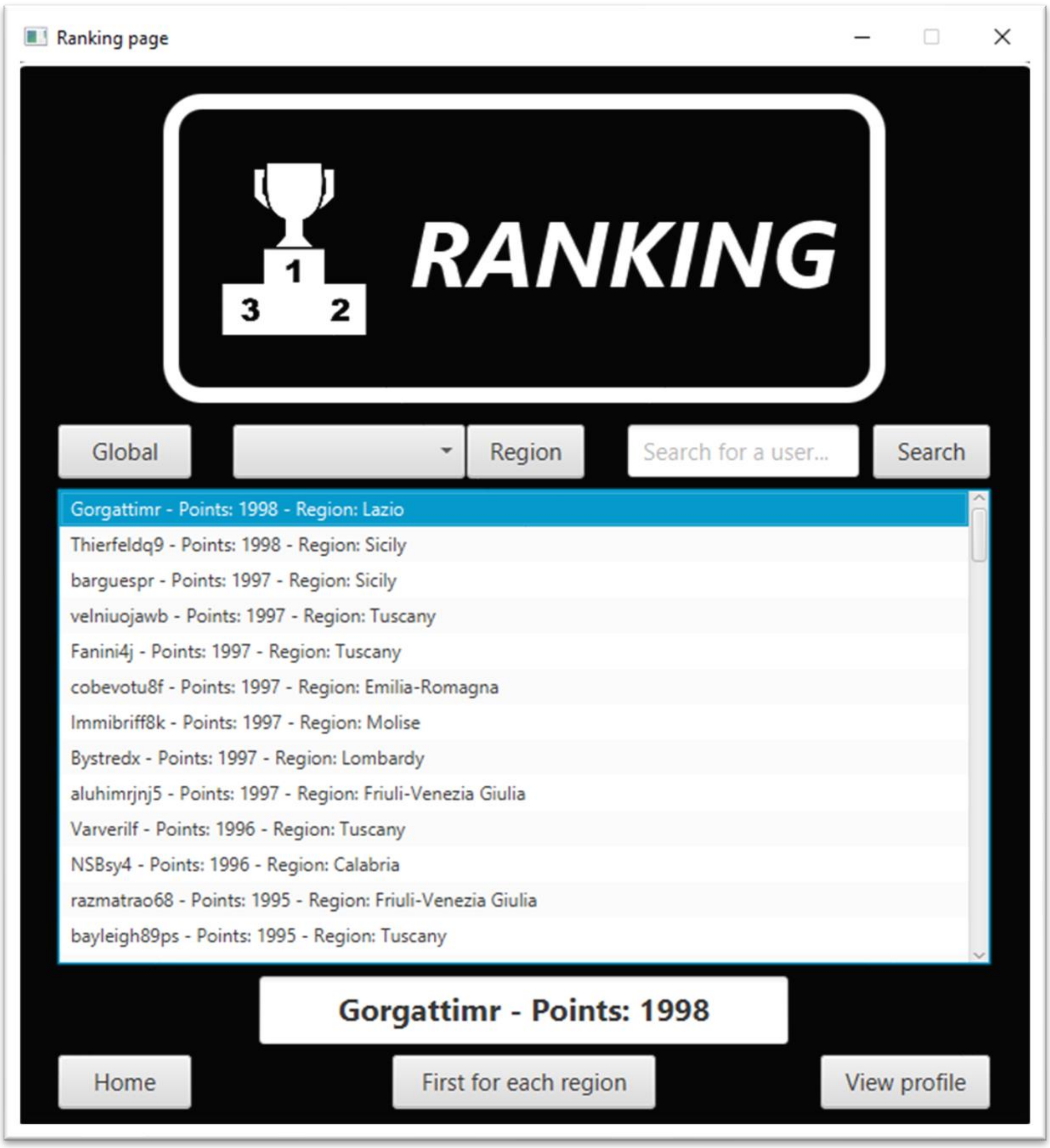
It is important that every card is inserted in his specific role and there are no duplicates, otherwise the formation cannot be saved on the database.



Ranking Page

In this page the user can view the global ranking of the application. A different regional ranking can be shown if the user inserts a specific region and then clicks the *Region* button. The application can also show which user is the first of each region with the appropriate button positioned on the bottom of the page.

A user can also search for another user by typing the username (or part of it) in the appropriate field and then by the clicking on the *Search* button. If a user clicks on a user from the list, that user will be highlighted on a bigger field positioned at the center of the page to have a better view on his ranking info.



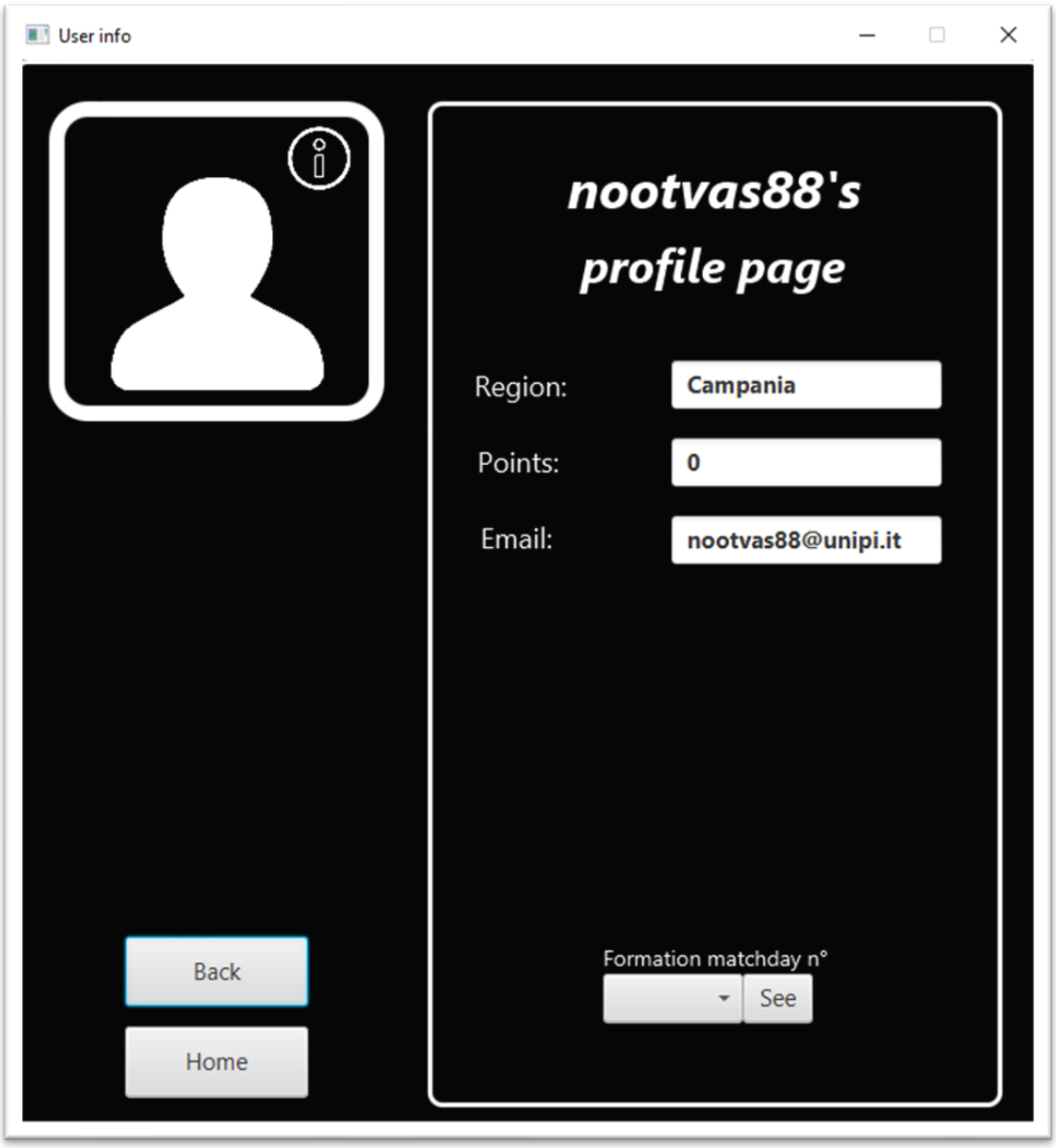
Gorgattimr - Points: 1998 - Region: Lazio
Thierfeldq9 - Points: 1998 - Region: Sicily
barguespr - Points: 1997 - Region: Sicily
velniuojawb - Points: 1997 - Region: Tuscany
Fanini4j - Points: 1997 - Region: Tuscany
cobevotu8f - Points: 1997 - Region: Emilia-Romagna
Immibriff8k - Points: 1997 - Region: Molise
Bystredx - Points: 1997 - Region: Lombardy
aluhimrjnj5 - Points: 1997 - Region: Friuli-Venezia Giulia
Varverilf - Points: 1996 - Region: Tuscany
NSBsy4 - Points: 1996 - Region: Calabria
razmatrao68 - Points: 1995 - Region: Friuli-Venezia Giulia
bayleigh89ps - Points: 1995 - Region: Tuscany

Gorgattimr - Points: 1998

See User’s Information Page

From the *Ranking Page*, if the user selects another user and then clicks on the *View profile* button, this page will show up:

From here, a user can view another user’s information and the formations of the current or past matchdays by clicking on the appropriate buttons.



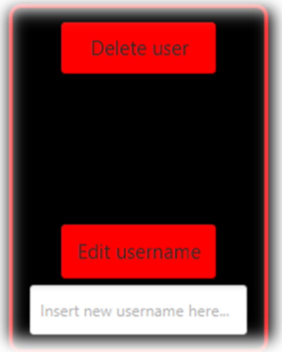
nootvas88's profile page

Region: Campania

Points: 0

Email: nootvas88@unipi.it

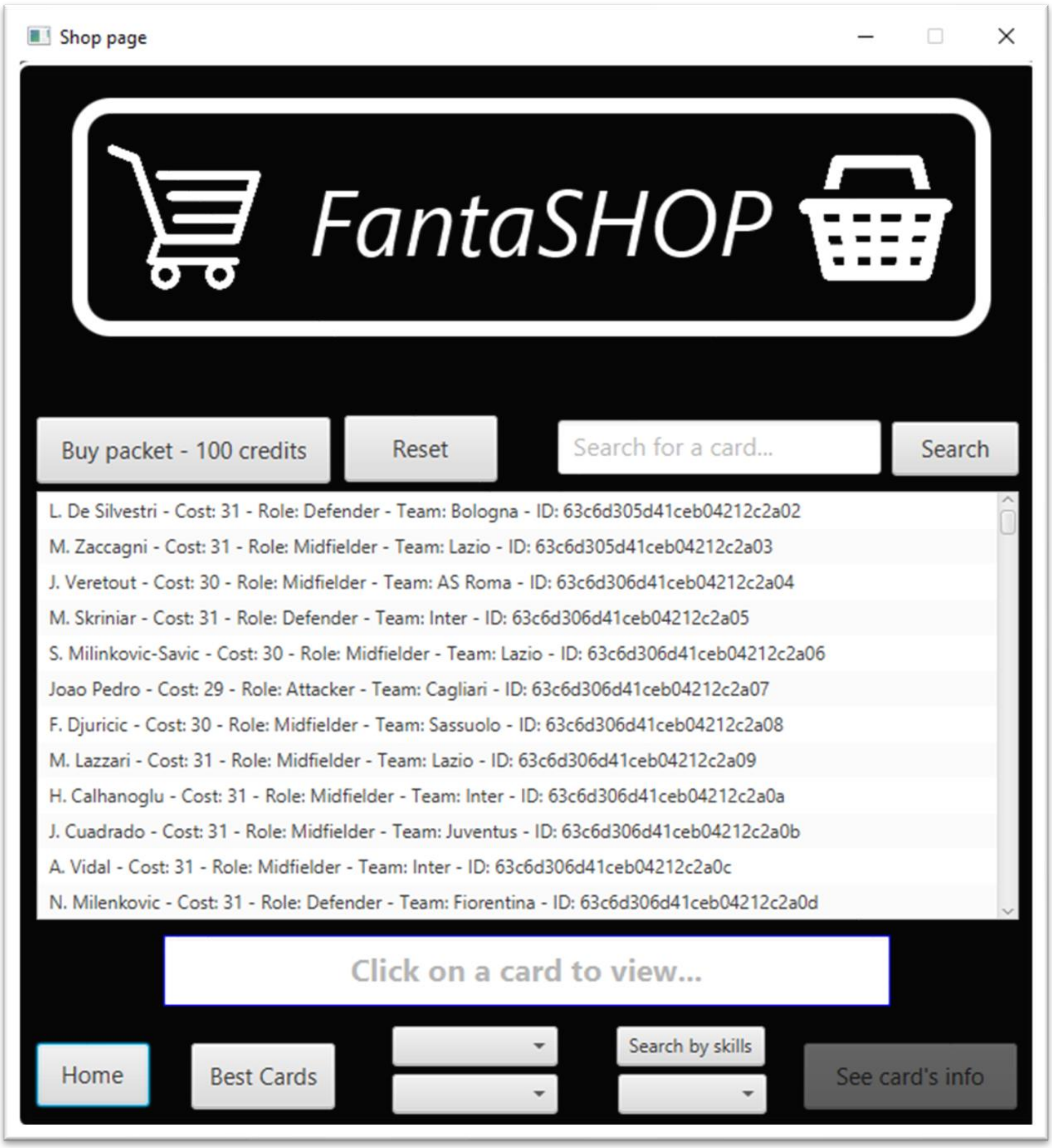
If an admin opens this page, it can also delete this user or change his/her username by using the purpose-built buttons that appears on the left side of the page.



Shop Page

In this page the user can view the list of all cards featured in the application and search for a particular card by name typing on the appropriate field.

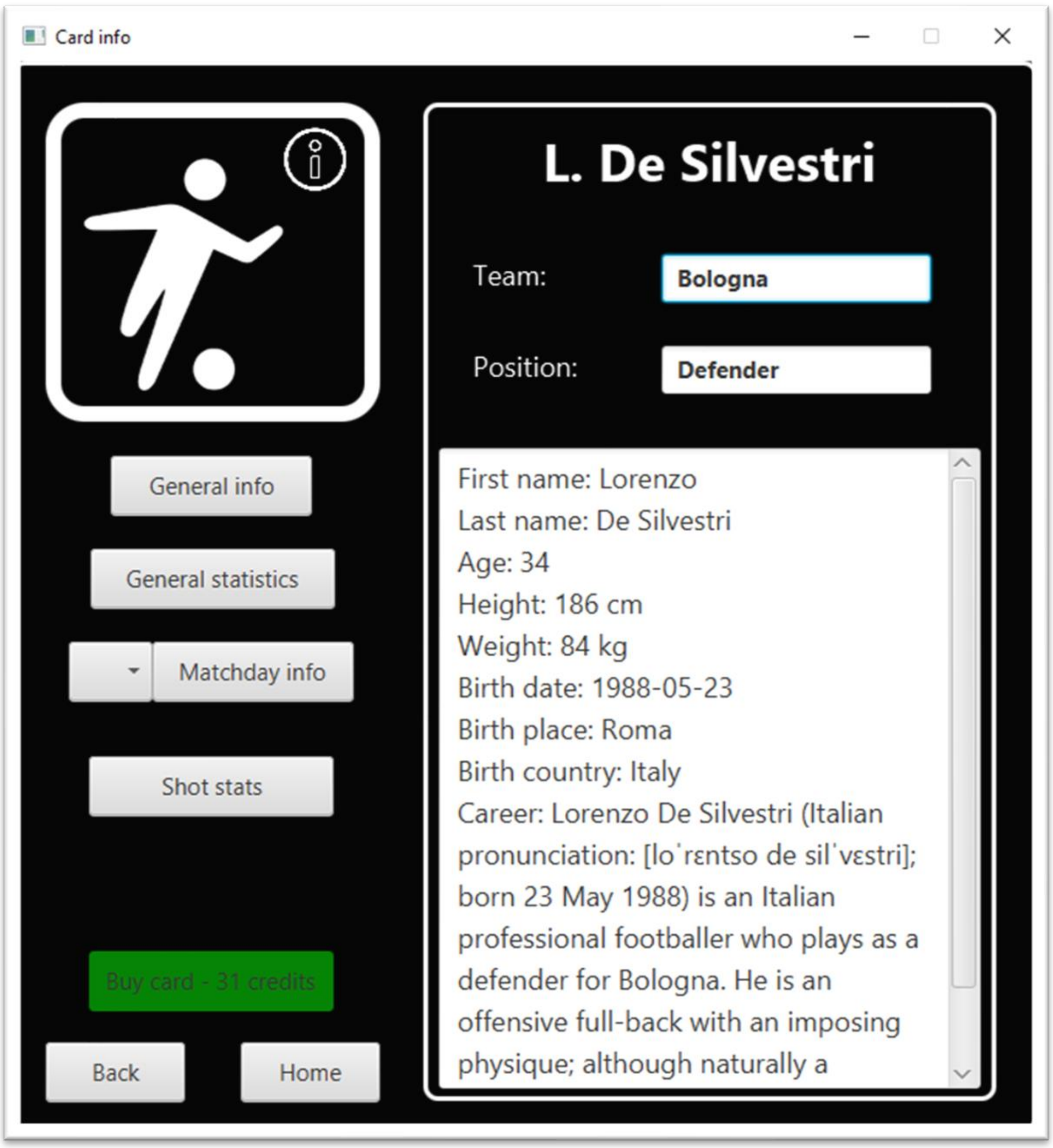
Other search options are implemented in this page, such as: the *Best Cards* filter that search the top 10 cards which got the better performing during the whole season; a user can also filter a card by team name, by skill or by position and view the top results. The *Reset* button resets all previously mentioned fields and show the full list of cards. The user can also buy a packet of cards spending 100 credits and receiving back 5 random cards.



```
L. De Silvestri - Cost: 31 - Role: Defender - Team: Bologna - ID: 63c6d305d41ceb04212c2a02
M. Zaccagni - Cost: 31 - Role: Midfielder - Team: Lazio - ID: 63c6d305d41ceb04212c2a03
J. Veretout - Cost: 30 - Role: Midfielder - Team: AS Roma - ID: 63c6d306d41ceb04212c2a04
M. Skriniar - Cost: 31 - Role: Defender - Team: Inter - ID: 63c6d306d41ceb04212c2a05
S. Milinkovic-Savic - Cost: 30 - Role: Midfielder - Team: Lazio - ID: 63c6d306d41ceb04212c2a06
Joao Pedro - Cost: 29 - Role: Attacker - Team: Cagliari - ID: 63c6d306d41ceb04212c2a07
F. Djuricic - Cost: 30 - Role: Midfielder - Team: Sassuolo - ID: 63c6d306d41ceb04212c2a08
M. Lazzari - Cost: 31 - Role: Midfielder - Team: Lazio - ID: 63c6d306d41ceb04212c2a09
H. Calhanoglu - Cost: 31 - Role: Midfielder - Team: Inter - ID: 63c6d306d41ceb04212c2a0a
J. Cuadrado - Cost: 31 - Role: Midfielder - Team: Juventus - ID: 63c6d306d41ceb04212c2a0b
A. Vidal - Cost: 31 - Role: Midfielder - Team: Inter - ID: 63c6d306d41ceb04212c2a0c
N. Milenkovic - Cost: 31 - Role: Defender - Team: Fiorentina - ID: 63c6d306d41ceb04212c2a0d
```

See Card’s Information Page

From the *Shop Page*, if the user selects a card and clicks on the *See card’s info* button, this page will show up:



L. De Silvestri

Team: Bologna

Position: Defender

First name: Lorenzo
Last name: De Silvestri
Age: 34
Height: 186 cm
Weight: 84 kg
Birth date: 1988-05-23
Birth place: Roma
Birth country: Italy
Career: Lorenzo De Silvestri (Italian pronunciation: [loˈrɛntso de silˈvɛstri]; born 23 May 1988) is an Italian professional footballer who plays as a defender for Bologna. He is an offensive full-back with an imposing physique; although naturally a

From there the user can view all card’s statistics and information, all actions done on each matchday and all footballer’s shots stats. The user can also buy the card by clicking the green button and by spending the shown credits.

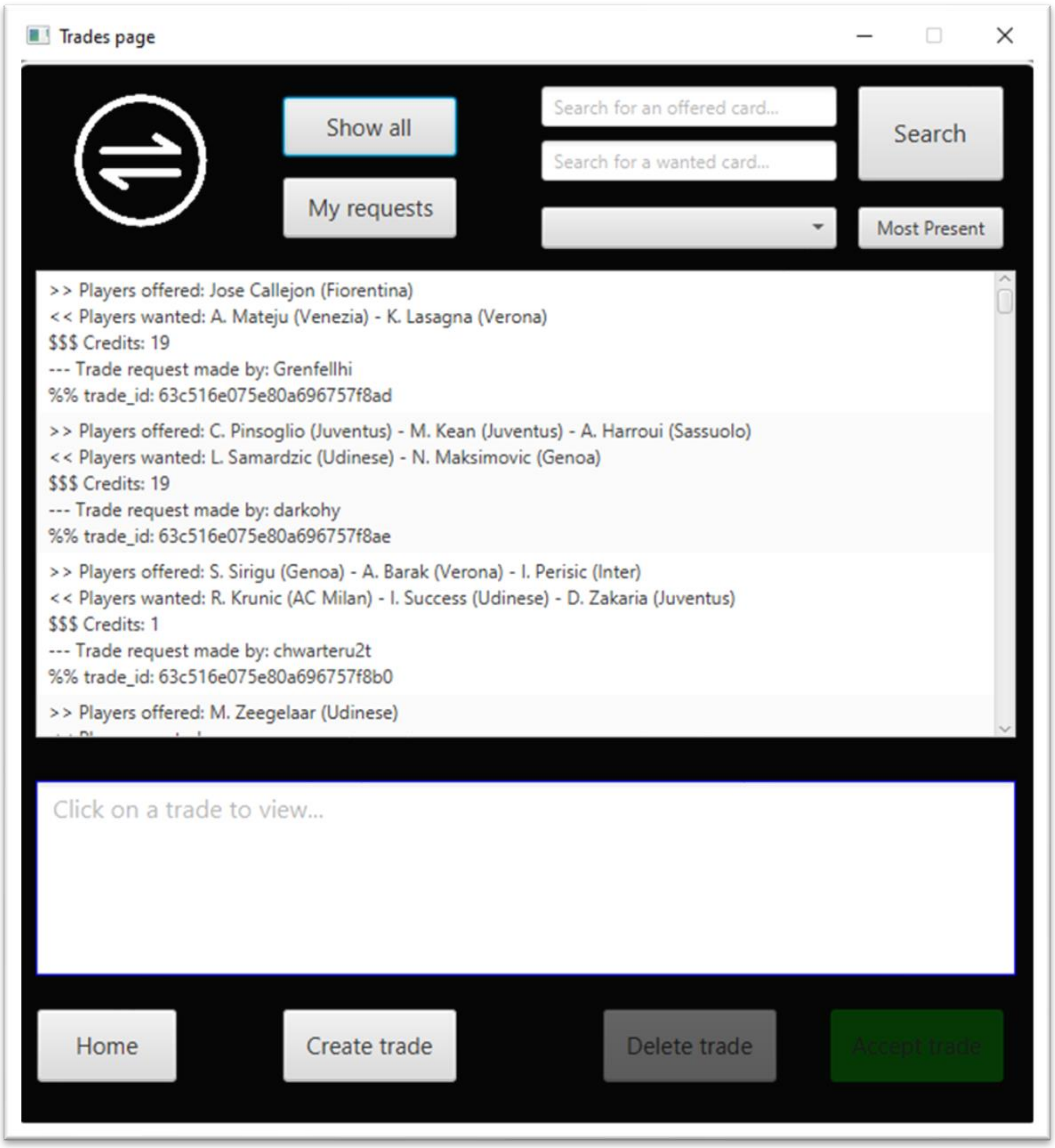
Trades Page

In this page the user can view all the trades done by the other users.

When opening the page, a list of pending trades can be seen; this list can be seen again by clicking the *Show all* button.

The user can also search for a particular trade by typing the name of a card in the appropriate field and search for an offered or a wanted card. When the user clicks on a trade and meet the requirements to accept it, the green *Accept trade* button will be clickable and the user can decide to accept the trade or not.

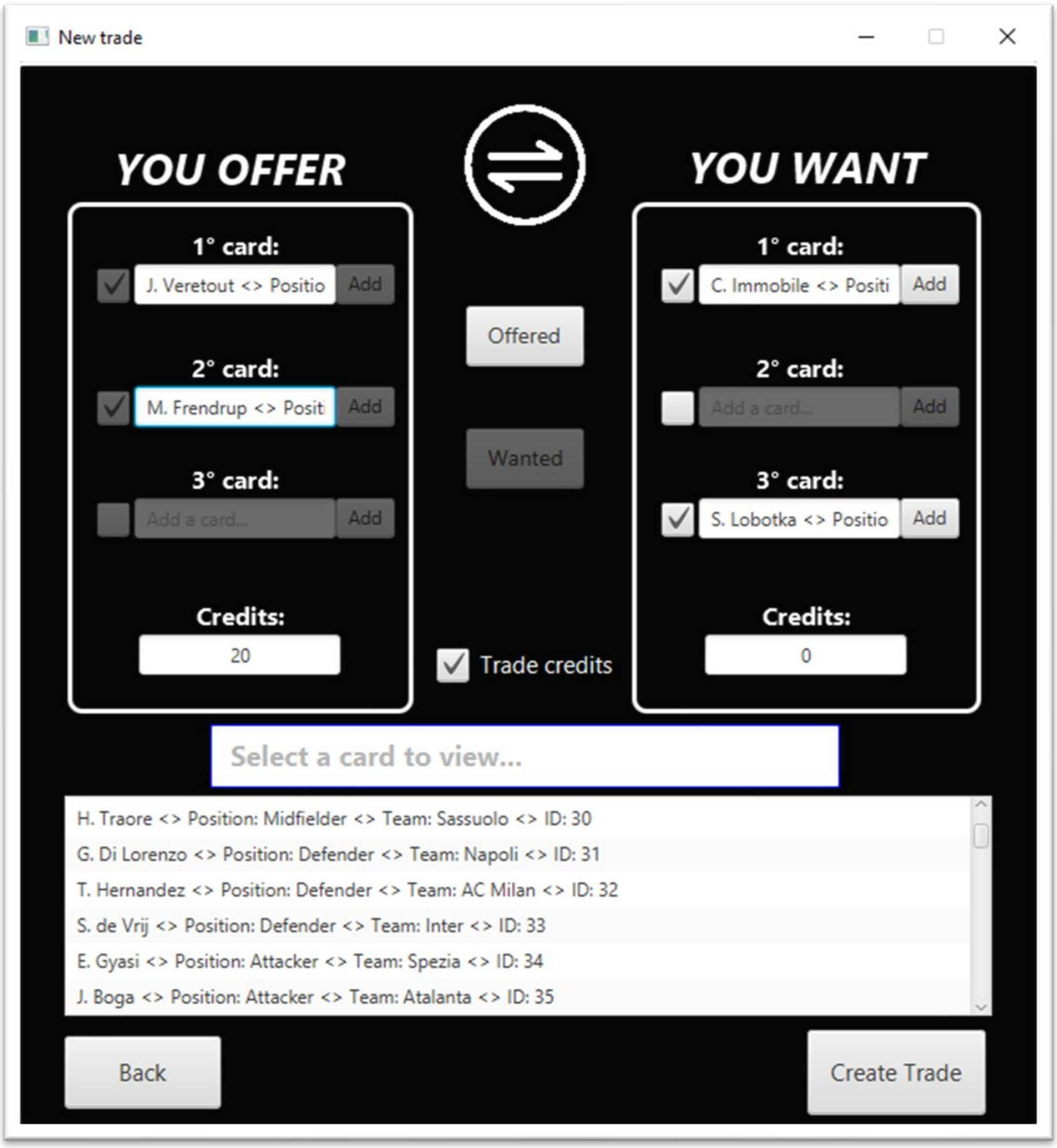
The user can also view which cards are the most offered and the most wanted by using the appropriate button. By clicking the *My requests* button, the user can view all his trades (pending and completed) and can decide to delete one of his pending trades by clicking the *Delete trade* button. A user can only delete the trades made by him/herself.



```
>> Players offered: Jose Callejon (Fiorentina)
<< Players wanted: A. Mateju (Venezia) - K. Lasagna (Verona)
$$$ Credits: 19
--- Trade request made by: Grenfellhi
%% trade_id: 63c516e075e80a696757f8ad
>> Players offered: C. Pinsoglio (Juventus) - M. Kean (Juventus) - A. Harroui (Sassuolo)
<< Players wanted: L. Samardzic (Udinese) - N. Maksimovic (Genoa)
$$$ Credits: 19
--- Trade request made by: darkohy
%% trade_id: 63c516e075e80a696757f8ae
>> Players offered: S. Sirigu (Genoa) - A. Barak (Verona) - I. Perisic (Inter)
<< Players wanted: R. Krunic (AC Milan) - I. Success (Udinese) - D. Zakaria (Juventus)
$$$ Credits: 1
--- Trade request made by: chwarteru2t
%% trade_id: 63c516e075e80a696757f8b0
>> Players offered: M. Zeegelaar (Udinese)
```

New Trade Page

Starting from the *Trade Page* if the user clicks on the *Create trade* button, this page will show up:



H. Traore <> Position: Midfielder <> Team: Sassuolo <> ID: 30
G. Di Lorenzo <> Position: Defender <> Team: Napoli <> ID: 31
T. Hernandez <> Position: Defender <> Team: AC Milan <> ID: 32
S. de Vrij <> Position: Defender <> Team: Inter <> ID: 33
E. Gyasi <> Position: Attacker <> Team: Spezia <> ID: 34
J. Boga <> Position: Attacker <> Team: Atalanta <> ID: 35

To create a new trade offer, the user must select the cards that wants to offer and to receive.

At the start, all checkboxes will be unchecked. If a user wants to insert a card in a location, he must check the appropriate field, by doing this the *Add* buttons will became clickable. After that, the user must click on a card from the list on the lower part of the page and then add the card in the field by clicking on the *Add* button.

If the user wants to navigate through the owned cards list and the total cards list, he must click on the *Offered* and *Wanted* button. Only one copy of the same card can be offered or received. It is possible to leave the *wanted cards* fields empty to accept the trade.

The user can also add credits to his trade; he/she must first check *Trade credits* checkbox and then add an integer value to the appropriate field.

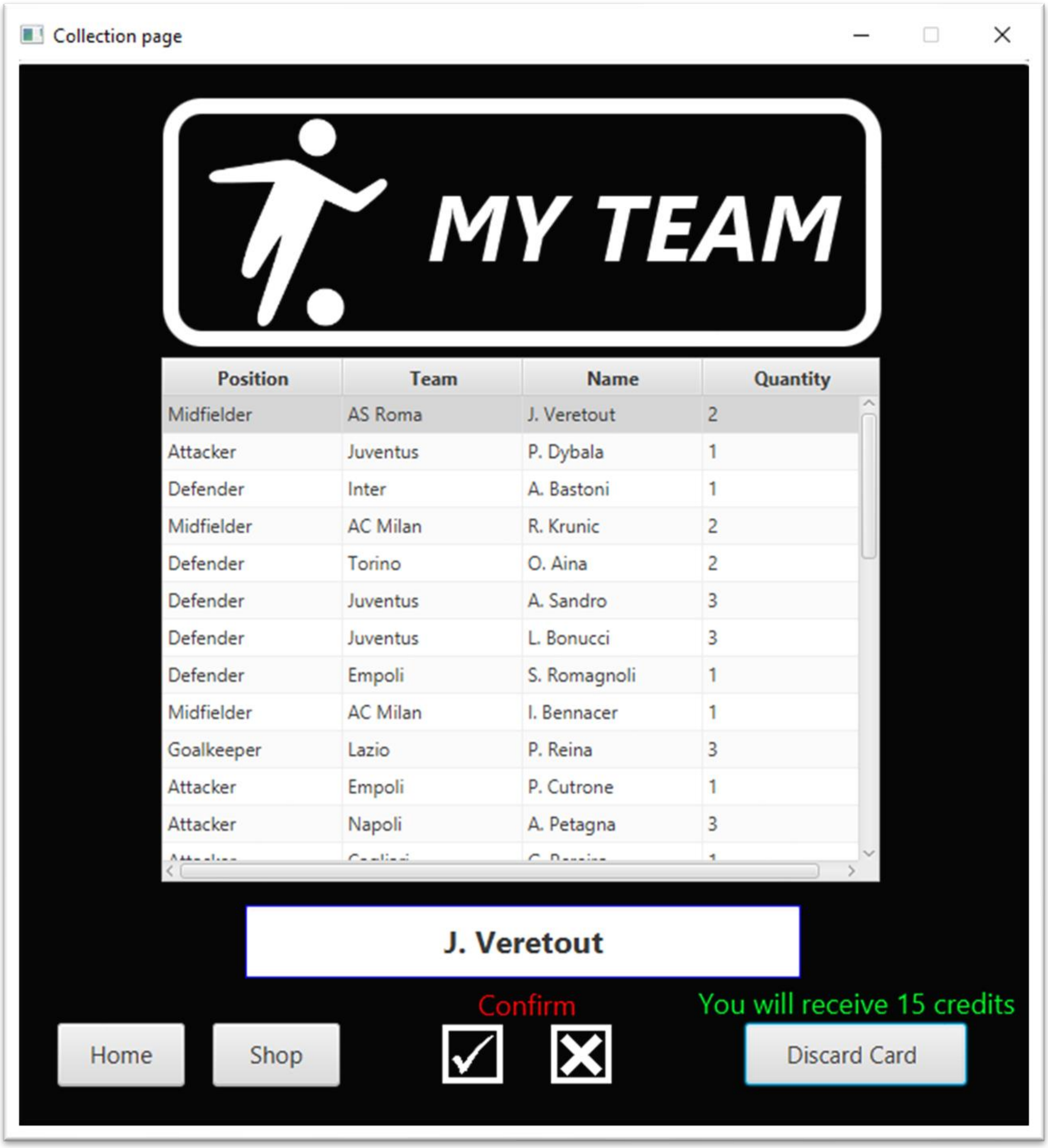
If all requirements are satisfied (credit check, no doubles, at least one card offered) the user can create the trade and it will be inserted into the database.

Collection Page

In this page the user can view its entire card collection.

When a card is selected, it will be highlighted on the lower blue bordered field. After that, the user can decide to discard the card and receive credits in exchange by clicking on the *Discard card* button and confirming the choice.

The credits received from discarding a card is equal to its current value divided by 2.



Position	Team	Name	Quantity
Midfielder	AS Roma	J. Veretout	2
Attacker	Juventus	P. Dybala	1
Defender	Inter	A. Bastoni	1
Midfielder	AC Milan	R. Krunic	2
Defender	Torino	O. Aina	2
Defender	Juventus	A. Sandro	3
Defender	Juventus	L. Bonucci	3
Defender	Empoli	S. Romagnoli	1
Midfielder	AC Milan	I. Bennacer	1
Goalkeeper	Lazio	P. Reina	3
Attacker	Empoli	P. Cutrone	1
Attacker	Napoli	A. Petagna	3

J. Veretout

Confirm

You will receive 15 credits