

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*](https://github.com/Fochi1999/FantaManager)

Summary:

[Introduction 4](#_Toc124871202)

[Requirements 5](#_Toc124871203)

[Main actors 5](#_Toc124871204)

[Functional Requirements 5](#_Toc124871205)

[Non-Functional Requirements 6](#_Toc124871206)

[Use Cases 6](#_Toc124871207)

[UML Class Diagram 6](#_Toc124871208)

[Data Model 6](#_Toc124871209)

[Distributed Database Design 6](#_Toc124871210)

[Architectural Design 8](#_Toc124871211)

[Software architecture 8](#_Toc124871212)

[Client Side 8](#_Toc124871213)

[Server side 8](#_Toc124871214)

[Database Organization 8](#_Toc124871215)

[MongoDB 8](#_Toc124871216)

[Redis 12](#_Toc124871217)

[Inter-Databases Consistency 13](#_Toc124871218)

[Implementation 13](#_Toc124871219)

[Scraping 13](#_Toc124871220)

[Main Packages and Classes 14](#_Toc124871221)

[it.unipi.dii.ingin.lsmsd.fantamanager 14](#_Toc124871222)

[it.unipi.dii.ingin.lsmsd.fantamanager.admin 14](#_Toc124871223)

[it.unipi.dii.ingin.lsmsd.fantamanager.collection 14](#_Toc124871224)

[it.unipi.dii.ingin.lsmsd.fantamanager.formation 14](#_Toc124871225)

[it.unipi.dii.ingin.lsmsd.fantamanager.page\_controllers 14](#_Toc124871226)

[it.unipi.dii.ingin.lsmsd.fantamanager.player\_classes 15](#_Toc124871227)

[it.unipi.dii.ingin.lsmsd.fantamanager.trades 15](#_Toc124871228)

[it.unipi.dii.ingin.lsmsd.fantamanager.user.user 15](#_Toc124871229)

[it.unipi.dii.ingin.lsmsd.fantamanager.user.userMongoDriver 16](#_Toc124871230)

[it.unipi.dii.ingin.lsmsd.fantamanager.user.util 16](#_Toc124871231)

[src/main/resources 16](#_Toc124871232)

[Constraints 16](#_Toc124871233)

[Most relevant queries 17](#_Toc124871234)

[Indexes 17](#_Toc124871235)

[Test 17](#_Toc124871236)

# 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 card to play in order to obtain points.

A user can possess an unlimited number of cards resembling his/her favorite footballers, view his statistics and decide to trade or discard the owned cards. Trades are also possible in the game; a user can view all trade offers created by other users, accept them or search for a particular card.

The application permits the user to view the global or a region ranking and view other users’ profile and his/her information’s. A store page is also implemented to allow users to buy cards or pack of cards.

When the user register to the application is granted to him/her some credits to start and five packs of cards for free, each of them containing random cards.

# 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 in order 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 player. (Find by name)
* A user can view the collection of players 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 collection of players.
* A user can decide the player’s formation for the next week.
* A user can view the global 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 player his collection with another user for another player (duplicate of players are allowed to be received). There are no limits to the player’s collection size.
* A user can view all trade request.
* A user can search a trade request by player 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 buy one or more packs of players.
* A user can buy a specific player.
* A user can discard a player 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 the matchday.

## Non-Functional Requirements

* The application needs to be **consistent** to provide correct information to all the 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.

## Use Cases

//TODO

## UML Class Diagram

//TODO

## Data Model

//TODO

## Distributed Database Design

//TODO (sharding)

# 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. (Describe how are divided)

//TODO ricontrollare

## Database Organization

### 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:

Immagine che contiene testo

Descrizione generata automaticamente

Every attribute 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.

Immagine che contiene testo

Descrizione generata automaticamenteImmagine che contiene testo

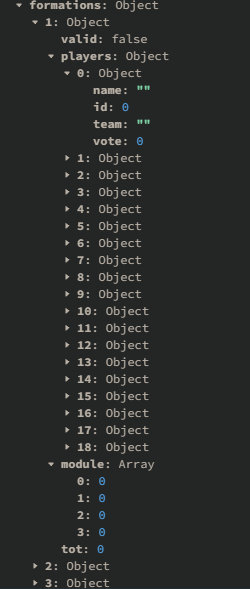
Descrizione generata automaticamenteLet’s give a special look to general\_statistics and statistics objects:

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.

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 attribute 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 the current matchday has been played.

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

**//TODO inserire immagine user dopo cambio email**

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 vote received in the current matchday.

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

**Immagine che contiene testo

Descrizione generata automaticamente**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 the cards that the user that create the trade offers and the cards that the same user wants to receive. 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 creates 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 see the trade with the point of view of the user logged in.

At last, the status value indicates the status of the trade: 0 for pending, 1 for completed and accepted by someone.

### 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): //TODO dare occhiata ai nomi usati per indicare le key

* **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 because there are a lot of writes instead of reads.

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 the value with the specific button.
* **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.

## 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.

When a trade is created, accepted or deleted, 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:

* 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.

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 is involved.

# Implementation

## Scraping

A scraping script was written with Python in order to retrieve informations about footballers and his statistics. Those informations were 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.

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 informations 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 the *calculate\_matchday.java* file; it contains the functions that involves the calculation of a matchday. This is how the scraping is simulated into the application.

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

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

* *card\_collection.java*:
* *collection.java*:
* *LineTable.java*:

### 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*:
* *player\_formation.java*:

### it.unipi.dii.ingin.lsmsd.fantamanager.page\_controllers

This package contains all the files that permits to perform the action when a user interacts with the graphical interface of the application.

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

* *AdminPageController.java*
* *ChoisePlayerFormationController.java* (where a user can choose a card during the formation set up)
* *CollectionController.java*
* *FormationController.java*
* *HomeController.java*
* *login\_registrationController.java*
* *NewTradeController.java* (where a user can create a new trade)
* *OptionController.java*
* *RankingController.java*
* *SeeCardController.java* (where a user sees card’s info)
* *SeeUserController.java* (where a user sees another user’s info)
* *SeeUserFormationController.java* (where a user sees his formation)
* *ShopController.java*
* *ShotsStatsController.java*
* *TradesController.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.
* *general\_statistics\_class.java*:
* *matchday\_class.java:*
* *player\_class.java:*
* *see\_card.java:* includes all functions used in the *SeeCard* page and not involves changes in the GUI. Also contains some little calls on the database.
* *shots\_info\_class.java:*
* *statistics\_class.java:*

### 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.

### 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*:
* *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.

### 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. //TODO ricontrollare questa parte (indexes?)

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 when retrieving informations 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

//TODO inserire screenshot di aggregations

## Indexes

//TODO

# Test

//TODO

# User Manual

//TODO