****

*Large Scale and Multi-structured Databases*

*PokeMongo: Project Documentation*

Edoardo Fazzari, Mirco Ramo, Olgerti Xhanej

Summary

[1. Introduction 3](#_Toc59209992)

[2. Analysis stage 4](#_Toc59209993)

[2.1 Functional requirements and use cases 4](#_Toc59209994)

[2.1.1 Use Cases List 4](#_Toc59209995)

[2.1.2 UML Use Case Diagram 7](#_Toc59209996)

[2.2 Non-functional requirements 8](#_Toc59209997)

[2.3 Sources, velocity properties and volume of data 8](#_Toc59209998)

[2.5 Main application queries 10](#_Toc59209999)

[2.6 Feasibility study and load estimation 11](#_Toc59210000)

[3 Project Stage 12](#_Toc59210001)

[3.1 Adopted Databases 12](#_Toc59210002)

[3.2 Document Database 13](#_Toc59210003)

[3.2.1 Queries handled 13](#_Toc59210004)

[3.2.2 Entities handled 13](#_Toc59210005)

[3.2.3 Collections structure 14](#_Toc59210006)

[3.2.4 Indexes 15](#_Toc59210007)

[3.3 Graph Database 18](#_Toc59210008)

[3.3.1 Queries handled 18](#_Toc59210009)

[3.3.2 Entities handled 18](#_Toc59210010)

[3.3.3 Graph Structure 19](#_Toc59210011)

[3.3.4 Indexes 20](#_Toc59210012)

[3.4 Redundancies and consistency management 21](#_Toc59210013)

[3.4.1 Team Handling 21](#_Toc59210014)

[3.4.2 User’s redundancies 22](#_Toc59210015)

[3.4.3 Pokémon’s redundancies 22](#_Toc59210016)

[3.4.4 The Analytic collection 22](#_Toc59210017)

[3.5 Db properties 23](#_Toc59210018)

[3.5.1 Availability 23](#_Toc59210019)

[3.5.2 Replicas 23](#_Toc59210020)

[3.5.3 Eventual consistency 24](#_Toc59210021)

[3.5.4 Sharding 25](#_Toc59210022)

[3.5.5 Pros and drawbacks 26](#_Toc59210023)

[3.6 Clients, servers, daemon threads 26](#_Toc59210024)

[3.7 Technologies and frameworks 28](#_Toc59210025)

[4 Implementation Stage 29](#_Toc59210026)

[4.1 Package structure 29](#_Toc59210027)

[4.1.1 Package analysis 29](#_Toc59210028)

[4.1.2 Package analysis: cache 30](#_Toc59210029)

[4.1.3 Package analysis: dataanalysis 32](#_Toc59210030)

[4.1.4 Package analysis: exceptions 33](#_Toc59210031)

[4.1.5 Package analysis: javafxextensions 33](#_Toc59210032)

[4.1.7 Package analysis: security 42](#_Toc59210033)

[4.1.8 Package analysis: userInterface 43](#_Toc59210034)

[4.1.9 Package analysis: utils 44](#_Toc59210035)

[5 Test stage 45](#_Toc59210036)

# Introduction

PokeMongo is a gaming application in which users compete each other to build up the best Team choosing between the set of Pokémon available in the environment; they can also follow other users in order to make new friends basing on common friends or common interests. Moreover users can express sentiments on Pokémon, choosing their favorite ones and posting/commenting on them.

Every trainer (normal user) can build up his own team. Every Team is composed by up to 6 distinct Pokémon and is assigned to a numerical value (points) based on features and properties of the chosen Pokémon, for ranking purposes.

Users can also navigate through the ranking in order to visualize the best teams (according to the values cited before) and the most used/caught Pokémon, both among their friends, grouped by country and among worldwide players.

User can browse for a specific Pokémon using the Pokédex tool, in which he/she can lookup for Pokémon according to search filters like Pokémon name, Type or Points.

Moreover, as a “real” Pokémon Trainer, the user is invited to *Catch ‘em‘ all*, i.e. to try to get a new Pokémon in order to create/update his/her own team. Thus, it is provided to the user a prefix number of daily Pokéball to be used to try to capture them. At each Pokémon is associated a probability to catch it, the higher the Pokémon’s value, the lower the probability.

Furthermore, the user can exploit the social network structure of the application to make new friends and discover new Pokémon. Indeed, he/she can search for new friends by username or choosing them among the provided *recommended friends* list.

The user can choose his/her favorite Pokémon, obtaining in this way a shortcut to catch it faster, and can *post* or *answer* to *posts* in order to express his/her opinion on that Pokémon.

In addition, to extend the dynamic behavior of the application, the *catch rate* (i.e. the probability to get a Pokémon using a Pokéball) changes in time depending on the number of users who have that Pokémon: the more it is popular, the harder will be to catch it. Since the rankings’ points are computed based on the *catch rate*, the winning strategy could be on predicting which Pokémon will become popular in the near future and try to get it early! Every user has access to the visualization of the temporal drift of the *catch rate*.

The safeguard and the improvement of the application is in charge of admin users. They are able to ban mischievous users, delete inappropriate posts or comments, add/remove Pokémon to the collection, consult geo-temporal usage statistics which are useful to make new business plans.

# Analysis stage

## 2.1 Functional requirements and use cases

### 2.1.1 Use Cases List

* An unregistered user can
  + - * + Register
* A registered user
  + - * + Sign in
        + Consult Pokédex:

Search by name

Search by type(s)

Search by Pokédex ID

Search by catch rate

Search by points

Search by Pokémon characteristics like height or weight

* + - * + Consult rankings:

Most popular Pokémon among all users

Most popular Pokémon in each country

Best world team

Best friends’ team

Best team by country

* + - * + Find users:

See recommended users based on common friends

See recommended users based on common Pokémon interests

Find users by username

Follow them

Unfollow them

* + - * + Interact with Pokémon network:

Insert a Pokémon in his/her own favorite Pokémon list

Remove a Pokémon from the favorite ones

Create a post on a Pokémon to share opinions

Add answers to posts

The post owner can also remove the post at his/her will

* + - * + Team handling:

View team

Remove Pokémon from the team

Change name to the team

Save modified team

View the value of the team

* + - * + Catching:

Browse a Pokémon you want to catch searching it by name

Select a Pokémon you want to catch from the list of favorites

Try to catch the Pokemon to add to your team

* + - * + Settings:

Change email

Change password

Change country

* + - * + Logout:

Exit from the account

Return to the sign in window

* + - * + At each time can:

See the remaining daily Pokéballs

Start/Mute music

See his/her username

* + - * + At each time a Pokémon name is visible

By clicking on a Pokémon name, visualize all the information about it

* + - * + At each time a username is visible

By clicking on the username, visualize all the information about the user

* An admin can

Sign in

Add Pokémon to the Pokédex

Remove Pokémon from the Pokédex

See number of registered users in time

See number of logins per day

See number of logins per day in every country

Remove a user from the system

Remove posts from the network

Remove answers from the network

Consult rankings

Logout

* The system should

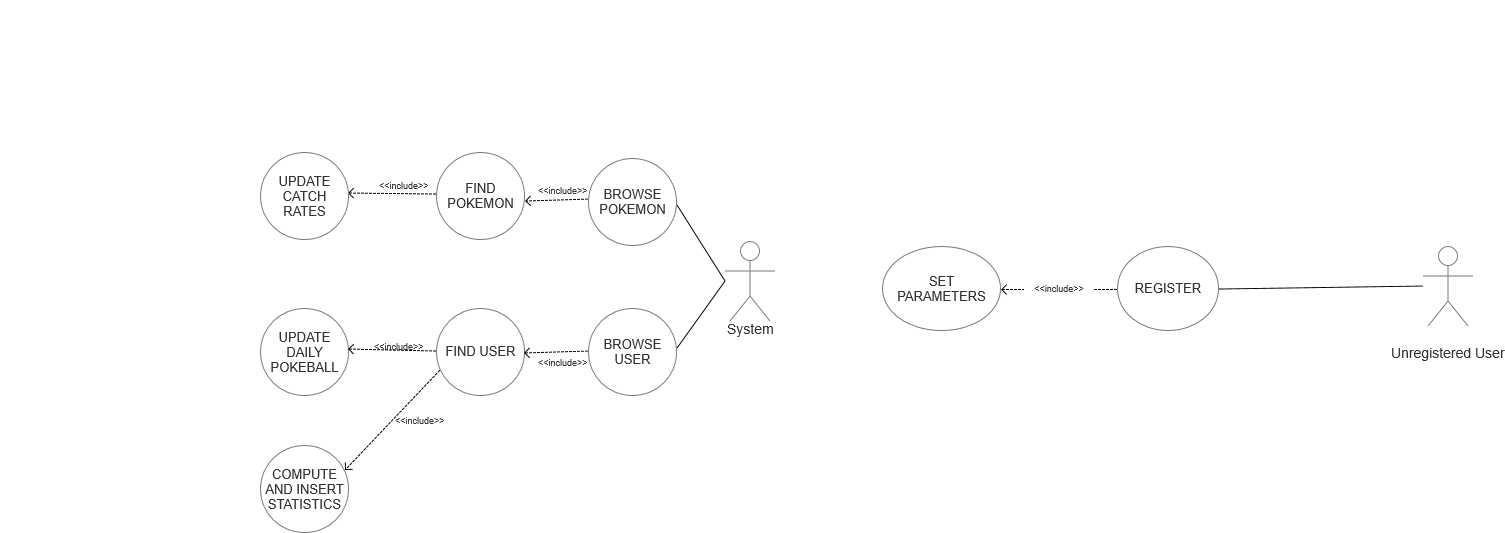
Daily update Pokéball number of each user

Periodically update Pokémon’ catch rates based on the number of users that own that Pokémon

Update team points if the user has 6 Pokémon of different types

Periodically compute usage statistics to be consulted by the administrators

### 2.1.2 UML Use Case Diagram



(\*) Only for the user who created the post

Browse-find-view answers had not been reported

## 2.2 Non-functional requirements

* The application should guarantee a high availability
* It should be easy to use, especially for children and youngsters, and enjoyable
* It should have a read-your-own-writes consistency on each user’s own team, so he/she can always be sure that Pokémon have been correctly caught/freed up
* The application should always provide to each user the most recent version of the rankings in order to permit him/her to immediately verify his/her progresses
* The statistics regarding usage and catch rate evolution are not needed to be real-time, they can be updated periodically and be eventually consistent
* Posts, comments and answers must follow a causal-consistency
* Response time is an important issue: redundancies and larger memory consumptions are preferred over high latencies
* Passwords are crypted for security reasons
* A graphical interface and the usage of multimedia are crucial for an involving game experience

## 2.3 Sources, velocity properties and volume of data

Data stored in the application backend has been downloaded and imported from the following sources:

*[https://pokeapi.co](https://pokeapi.co/)*, *[https://bulbapedia.bulbagarden.net/wiki](https://bulbapedia.bulbagarden.net/wiki/Main_Page)* 🡪 Pokémon data

*<https://gist.github.com/kalinchernev/486393efcca01623b18d>* 🡪 Countries data

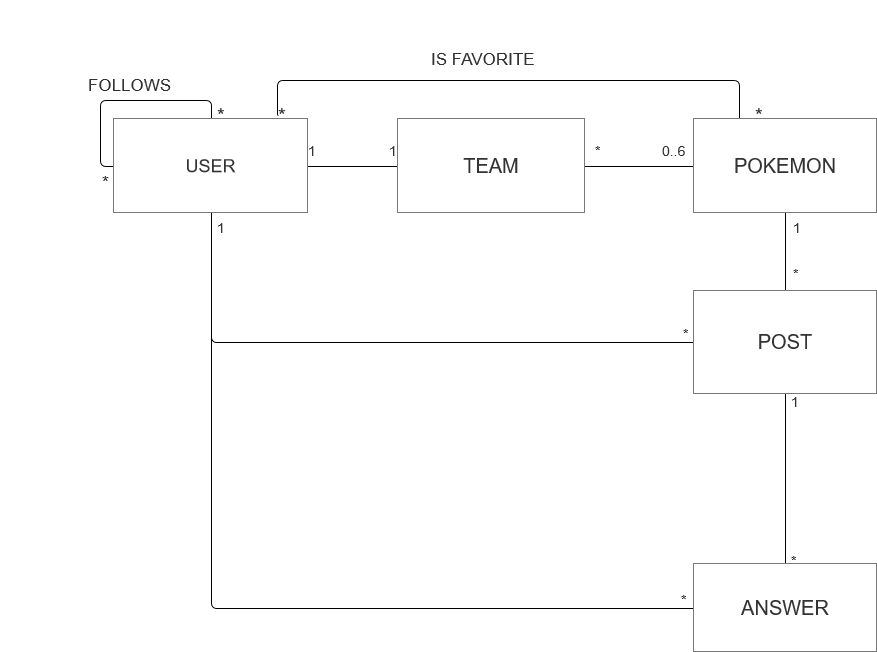
*<https://github.com/smashew/NameDatabases/blob/master/NamesDatabases/surnames/all.txt>* 🡪 Data for the generation of realistic users

All the imported data has been modified, updated and preprocessed in order to satisfy the application needs.

Users added have the only purpose of showing the application functionalities, privacy issues they are not real people; anyway they have been created using realistic criteria.

Velocity is guaranteed by the dynamic *catch rate* mechanism: the popularity of a Pokémon influences both its *catch rate* and the amount of points that it will provide. As a consequence, users are continuously stimulated by catching new Pokémon, in order to try to raise their amount of points: in this way old teams’ data becomes quickly out-of-date.

Volume of data, considering 250K users, almost 1K Pokémon and about 500K posts is no lower than 100Mb

2.4 UML Entities Diagram

A User can build up only 1 Team: of course, each Team has just one owner.

A Team is composed of a maximum of 6 Pokémon, every Pokémon can be caught by anyone, so can belong to many Teams.

A User can follow many Users, in the meanwhile he/she can have many followers.

A User can have many favorites Pokémon. A Pokémon can be favorite of many Users.

A Post is created just by one User on one Pokémon. A User can create many posts and a Pokémon can have many Posts talking about it.

An answer is written by one User and it refers to one Post. Users can submit many Answers and there can be many Answers behind a Post.

## 2.5 Main application queries

* Insert a user into the system at registration time
* Create a new Pokémon (admin only)
* Insert a Pokémon into a team
* Create a new Post
* Create a new Answer
* Create a follow relationship
* Add a Pokémon to the favorites
* Retrieve user information at login time
* Retrieve a user by username when looking for a new friend
* Retrieve team information based on user
* Retrieve Pokémon information using several filters
* Retrieve recommended users
* Retrieve list of a user’s friends
* Retrieve a Pokémon by name when trying to catch it
* Retrieve all the posts relative to a Pokémon
* Retrieve all the answers to a post
* Retrieve user’s favorite Pokémon
* Modify user settings (email, password, country)
* Update team’s name
* Update team’s points
* Update Pokémon’ *catch rates* 🡪Analytics: find % of users that own that Pokémon
* Remove a user (admin only)
* Remove a Pokémon (admin only)
* Remove a post (only admin and post’s owner)
* Remove a follow relationship
* Remove a Pokémon from the favorite ones
* Analytics: ranking of most popular Pokémon in world/each country
* Analytics: ranking of best teams in the world/each country/among friends
* Analytics: evolution on time of a Pokémon *catch rate*
* Analytics: evolution on time of number of logins per day/total users/logins per day by country (admin only)

## 2.6 Feasibility study and load estimation

PokéMongo is an application designed to be spread worldwide and played by plenty of users. In this paragraph we will try to estimate a realistic computational and memory load, this valuation will be taken into account in the project stage and will be at the foundation of the choices presented in the next chapters

* Since the globality of the app and the Social Network structure, we can estimate 5-10M of registered users. This means about 1M of logins-per-day.
* Registered Pokémon are 893. Even though there is the possibility for an admin to add new Pokémon, we think that they will be no more than 1K at every time.
* Expert users will probably generate a higher amount of posts/comments rather than new users. On average, there will be about 4-5M of posts/comments per day.
* Beginners are likely to generate an higher load of follow/unfollow requests respect to expert users. On average, it’s reasonable to count about 5 follow/unfollow requests per login.
* Pokéballs and Pokémon capture is the catchiest feature of the game. Very likely almost the totality of the users that logs into the app will spend all his/her available daily Pokéballs. Anyway it’s also probable that the most intriguing Pokémon will be the ones with low *catch rate.* Since there are 10 Pokéballs available each day, but the weighted average probability of catching a Pokémon can be estimated as near 10%, there will be about 1M of team updates per day
* As said in the previous point, we can count about 10 catch tries per day. It’s likely that the chosen Pokémon was taken from the provided favorite shortcut. Moreover, likes are integrating part of this Social Network, not only a practical tool for catching Pokémon. So we can say that there will be about 2M of likes per day.
* We can estimate that on the average a user will consult ranking twice per day. Indeed immediately after log in and at the catching of a new Pokémon are possible occasions in which the user could be interested in seeing his/her progresses. For this reason we can consider 2M of ranking consulting per day
* Very few users will change his/her settings or password, since they are long term fields: this kind of updates will be no more than 30-40K per day.

# 3 Project Stage

## 3.1 Adopted Databases

According to concept presented in the previous chapter we can make the following considerations:

1. Because of the performance constraint, a fast backend is required. Moreover, since the aim is to spread the application worldwide, the database infrastructure should be easy to distribute.
2. Pokémon must store heterogeneous data like URLs, different kinds of bios, float arrays and so on.
3. Users are divided into normal users and admins. Although the second ones are few, a denormalized approach could be better to handle the fact that these 2 categories have very different attributes.
4. Rankings are real-time OLAP queries: they need fast aggregation strategies.
5. Favorite Pokémon, friends, posts and answers together form a real Social Network.
6. A Team, in a normalized relational model, could be seen as a relationship table between Users and Pokémon. Anyway a huge table with a lot of duplicated PokémonID is not scalable for the requirements of this application. We need to find the best way to perform quickly both the retrieving of a user’s team and the ranking of the most used Pokémon, optimizing if possible memory consumption.

The points 1 to 4 guided the choice of a Document Database for handling User and Pokémon data. The flexibility, denormalization and performance of this kind of the database make it the most appropriate one.

The point 5 is best handled by a Graph DB, optimized for networks and different kinds of relationships. Moreover, we realized that the best way to handle a team is to decompose it in a set of Graph Relationships (USER – OWNS 🡪 POKEMON). Indeed, in this way queries mentioned at point 6 are very fast (just counting incoming/outcoming edges, see paragraph 3.3.1), and there are no useless, waste-memory repetitions of User IDs/Pokémon IDs.

Since each user can have only a team, team name and points are stored in the user collection.

## 3.2 Document Database

### 3.2.1 Queries handled

* Insert a user into the system at registration time
* Create a new Pokémon (admin only)
* Retrieve user information at login time
* Retrieve a user by username
* Retrieve Pokémon information using several filters
* Retrieve a Pokémon by name when trying to catch it
* Modify user settings (email, password, country)
* Update team’s name
* Update team’s points
* Update Pokémon’ *catch rates*
* Remove a user (admin only)
* Remove a Pokémon (admin only)
* Analytics: ranking of best teams in the world/each country/among friends
* Analytics: evolution on time of a Pokémon *catch rate*
* Analytics: evolution on time of number of logins per day/total users/logins per day by country (admin only)

### 3.2.2 Entities handled

Document Database stores information about USERS and POKEMON.

In particular it remembers user’s anagraphics and login data, last login, remaining Pokéballs, team name and earned points; a Boolean field distinguish admin from normal users. Admins have no points nor team or Pokéballs.

In a separate collection are stored data about Pokémon: PokédexId (*src: PokeAPI*), characteristics, one or more types, bio, images URLs, current *capture\_rate* and its last 30 *catch\_rates* stored into an array of Floats.

The details of the collections are reported in the following paragraph.

### 3.2.3 Collections structure

Immagine che contiene testo

Descrizione generata automaticamente

User Collection

Relevant attributes:

* *Admin*: TRUE🡪admin, FALSE🡪normal user
* *Username:* unique mnemonic ID of the user
* *Email:* must respect typical e-mail format
* *Password:* encrypted version of the user-chosen password
* *Last Login:* timestamp of the last time the user logged into the application
* *dailyPokeball:* number of daily Pokéballs left. They are up to 10 per day
* *points:* worth of his/her team

Immagine che contiene testo

Descrizione generata automaticamente

Pokémon Collection

Relevant attributes:

* *Id*: Pokédex ID (unique)
* *Name:* unique mnemonic ID of the Pokémon
* *Capture\_Rate*: current index of probability to catch the Pokémon
* *Portrait/Sprite:* URLs of the graphical representations of this Pokémon
* *Capture\_Rates:* array of the last 30 values of the *capture\_rate*, one for each of the last 30 days.

### 3.2.4 Indexes

* USERNAME

The first field in which we study the possibility of indexing is the *username* one in the *user* collection. A username is a REQUIRED and UNIQUE field of each user, and it is his/her mnemonic id inside the application.

The field username is involved in the following queries:

W1-)Insert a new username at registration time of an arbitrary user

W2-)Remove a username when an admin delete’s a user from the system

R1-)Check uniqueness of a username at registration time

R2-)Check user’s credential at login time

R3-)Find a user by username before submitting a new follow request

Assuming that a registered user will play the game for about 100 days before “getting bored”, we can state that the number of logins-per-day will be 100 times the number of registrations-per-day: this means that the queries R1+R2 are submitted 101 times more than query W1.

Moreover, we can assert that query W2 will be very rare, while R3 is a popular query among the network structure of the application, say 30 times the number of registered users: we find out that read operations on this field are about 130 times the number of write operations.

Now consider MongoDb performances with and without using an index on the *username* field, in a Database populated by 250k users.

db.user.find({username:”eee”}, {username:1}).explain(“executionStats”)

Immagine che contiene testo, monitor, screenshot, elettronico

Descrizione generata automaticamente

In the picture on the right is reported the output of the query when we do not use an index. Execution time is huge due to the very high number of docs examined.

Immagine che contiene testo, screenshot, elettronico, computer

Descrizione generata automaticamenteOn the contrary, with an index, the same query need an execution time almost 100 times lower, and of course thanks to the index, DBMS only need to examinate one document. Moreover the unique property permits to eliminate the need of submitting query R1 at each registration

Considering the very high speed-up ratio of the indexing and the high frequency of this kind of queries w.r.t. the write operations (as explained before), a UNIQUE INDEX on *username* has been created.

COUNTRY

As seen before, starting from the application queries we demonstrate the benefits of an index in the field *country.*

W1-)Insert the country data at registration time

W2-)Remove all the user’s data if a user is banned by an admin

W3-)Changing of settings after a user changes residence’s country

R1-)Rank all users by country

R2-)Rank countries with the highest logins-per-day ratios

Let x be the number of registrations-per-day (W1), w.r.t this number W2 and W3 are very rare operations. Indeed, even though we can expect mischievous behaviors from some user, the number of country changes will never be comparable with x.

On the other hand, in order to guarantee a read-your-own-write eventual consistency on ranking R1, this query is recomputed every time a user asks to see the ranking itself. Thus, since the gameplay is highly based on rankings, we can estimate that R1 frequency will be about 400x.

Furthermore we have to consider R2. Despite the fact that this query is executed just once per day (so frequency(R2)<<x), it is an asynchronous procedure sensitive to execution time since it needs to lock the entire collection, make it unavailable to users for a while.

As seen before, let us compare DBMS performances with and without a *country* index.

db.user.find({country:"Italy"}).explain("executionStats")

Immagine che contiene testo, monitor, screenshot, elettronico

Descrizione generata automaticamenteConsidering again about 250k users, without an index we need to scan the whole database, which means a medium-high execution time for each request.

Immagine che contiene testo, screenshot, monitor, elettronico

Descrizione generata automaticamente

On the contrary, we have a very high increase of performances introducing and index on *country*: execution time is about 58 times lower and the only documents examined are the ones that must be returned.

To summarize, considering the difference in frequency between reads and writes and the high decrease of execution time, an index on *country* has been introduced.

Pokémon’s NAME

Queries on Pokémon’s name:

W1-) Insert a new Pokémon into the Database

W2-) Delete a Pokémon from the Database

R1-) Search a Pokémon by name in the Pokédex

R2-) Browse a Pokémon by name in *Catch’Em’All* in order to try to catch it

R3-) Check name’s uniqueness of each Pokémon when added to the database

Again, W1 and W2 are rare and admin-related operations: this means that this queries will not require a frequent update of the index. On the contrary R1 and especially R2 are very frequent gameplay queries inside the application: we can estimate that R1+R2 frequency will be several orders of magnitude higher than W1+W2 one.

R3 instead is a query always required before W1, but it can be managed by DBMS adding a unique property to the index, thus reducing computational cost of the operation itself

In terms of execution time, the final report is the following:

Immagine che contiene testo, elettronico, screenshot, schermo

Descrizione generata automaticamenteImmagine che contiene testo, screenshot, monitor, elettronico

Descrizione generata automaticamente

Find with index

Find without index

Even if we have little changes on execution time due to the limited number of Pokémon, we can see how the index permits to decrease very much the number of examined documents.

For the reasons explained before and because of the very high ratio between reads and writes, we consider this little improvement enough relevant for the application purposes.

## 3.3 Graph Database

### 3.3.1 Queries handled

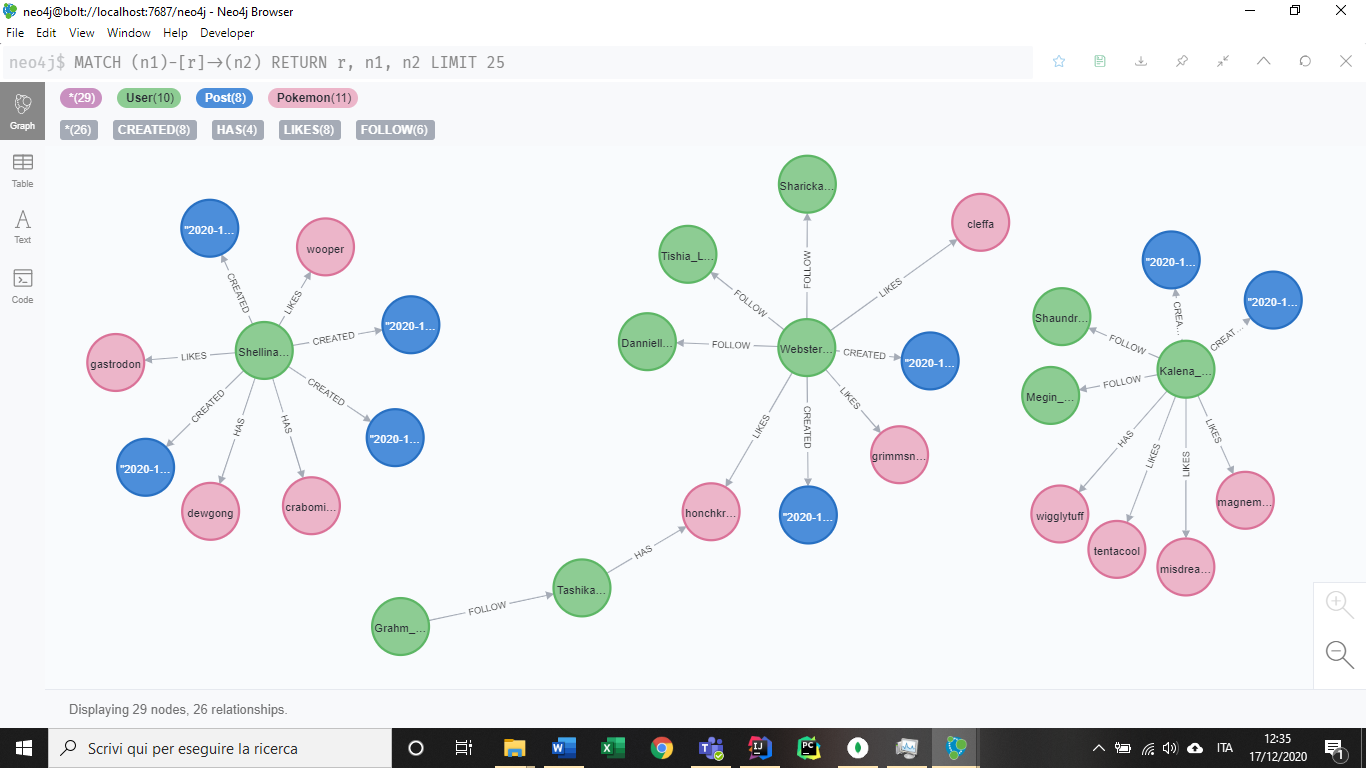
|  |  |
| --- | --- |
| APPLICATION QUERIES | GRAPH QUERIES |
| Insert a user into the system at registration time | Insert a new USER node into the graph |
| Create a new Pokémon (admin only) | Insert a new POKEMON node into the graph |
| Insert a Pokémon into a team | Add a OWNS relationship between a USER node and a POKEMON node |
| Create a new Post | Add a new POST node into the graph |
| Create a new Answer | Add a new POST node into the graph |
| Create a Follow relationship | Add a new FOLLOW edge |
| Add a Pokémon to the favorites | Add a new IS FAVORITE edge |
| Retrieve team information based on user | Given USER u, retrieve all the POKEMON nodes connected to u through a HAS edge |
| Retrieve recommended users | Match all the USER nodes at distance 2 from the given USER node U according to one of these patterns(the first one has precedence)  (U)—FOLLOWS🡪(USER f)—FOLLOWS🡪(USER recomm.)  (U)—LIKES🡪(POKEMON p)🡨LIKES—(USER recomm.) |
| Retrieve list of a user’s friends | Match all the USER nodes linked to U by an outcoming FOLLOWS edge |
| Retrieve all the posts relative to a Pokémon | Match all the POST nodes which are linked to the POKEMON node P through a TOPIC edge |
| Retrieve all the answers to a post | Match all the POST nodes which are linked to the POSTnode P through a TOPIC edge |
| Retrieve user’s favorite Pokémon | Match all the POKEMON nodes which are linked to the USER node U through a LIKES edge |
| Analytics: find % of users that owns a Pokémon | Count outcoming HAS edges from the POKEMON node p.  Divide the result by the total number of USER nodes |
| Remove a user (admin only) | Delete a USER node |
| Remove a Pokémon (admin only) | Delete a POKEMON node |
| Remove a post (only admin and post’s owner) | Delete a POST node |
| Remove a follow relationship | Delete a FOLLOWS edge |
| Remove a Pokémon from the favorite ones | Delete a LIKES edge |
| Analytics: ranking of most popular Pokémon in world/each country | Count ni= number of HAS incoming edges of POKEMON node pi, for each POKEMON node.  Sort k highest n1…nk and return relative p1…pk |

### 3.3.2 Entities handled

The Graph Database stores all the information needed to build the NETWORK INFRASTRUCTURE of the application:

* User’s usernames and country
* Pokémon’s name (+other attributes discussed in the paragraph 3.4)
* Post’s creation date and content
* HAS relationships for team handling, storing also the chosen slot for consistency checking
* LIKES relationships between a User and a Pokémon, for favorites handling
* FOLLOWS relationships between users
* TOPIC relationships between a Post and a Pokémon, in order to see the posts written about a specific Pokémon
* TOPIC relationships also between a Post and another Post, in order to visualize the comments to a Post
* CREATED relationships between a User and a Post to map the owner of each post/comment

### 3.3.3 Graph Structure



In the previous image a portion of the graph structure is reported.

Pokémon nodes are pink, user ones are green, blue nodes represent posts.

The property stored are the following:

**USER (node):** country, username

**POKEMON (node):** name, *capture rate*, sprite, type

**POST (node):** content, creation date

**FOLLOW [relationship]:** *no properties needed*

**LIKES [rel.]:** *no properties needed*

**HAS [rel.]:** slot

**CREATED [rel.]:** *no properties needed*

**TOPIC [rel.]:** *no properties needed*

### 3.3.4 Indexes

## 3.4 Redundancies and consistency management

As said in the paragraph relative to non-functional requirements (2.2), performance is an issue for the presented application. Thus we decided, whenever we had to choose from fast queries and reduced memory consumption, to give more importance to the first one, introducing redundancies to minimize pseudo-join operations.

Anyway, this has been done respecting a sort of “common sense”, so if we had to choice between spending a lot of memory for a minimum performance improvement or turning down the maniacal hunting of performance to the advantage of a relevant memory saving, we did the second one.

In the following paragraph are presented the main introduced redundancies and denormalizations, explaining also the implemented consistency mechanism to handle them.

### 3.4.1 Team Handling

In order to maximize response velocity, the Team Entity has been fully denormalized and decomposed. Indeed, as we explained before, a Team is nothing more than a name, and a collection of Pokémon owned by a user. To each team is associated an amount of points, computable starting from the Pokémon composing the Team and their *catch rate.*

Since every user can have only a Team, the Team Name property can be directly be stored into the Document Db’s user collection.

The amount of points is not recomputed each time the team is retrieved but it stored as a redundancy in the user collection until it is changed.

The collection of Pokémon is maintained as up to 6 edges between a User node and Pokémon nodes in the Graph Database. This choice is due the fact that:

* An array of Pokémon in the user collection was not so good for ranking most used Pokémon
* An array of Owner users in the Pokémon collection was bad for retrieving a user’s team.
* Considering both the previous arrays was terribly memory-expensive and costly for write accesses. Since the team is a central game-play write feature, this solution is not suitable.
* Considering two arrays in the same fashion as before, but storing IDs instead of plain documents was the worst idea in terms of performance: it would determine a pseudo-join operation for each r/w access.
* A Team collection would mean not overcoming the problems given by the relational model.
* Storing everything in a graph, thus repeating Team Name and points in each relationship was extremely memory-consuming. In the implemented way the retrieving of all the information is still fast since it can be parallelized.

### 3.4.2 User’s redundancies

The Document Database’s user collection already stores all the information about each user. Anyway, we decided to replicate some of these attributes in the Graph Database for performance purposes. In particular they are:

* **Username.** Despite the fact that DBMSs always provide an identification mechanism not related to the one made by the programmer, we chose to repeat the username to quickly retrieve friends’ and post/comment owners’ name.

This additional field is not so memory-intensive but can speed-up very much these queries

* **Country.** As considered in the paragraphs 2.6 and 3.2.4, there will be very few users that will update their settings compared to the ones that will consult rankings. Since the *Most used Pokémon by Country* is a Graph Database query, we decided to introduce this redundancy

### 3.4.3 Pokémon’s redundancies

Like for the users, a Document collection already stores Pokémon information. For similar causes we introduced these redundancies:

* **Name*, capture rate, sprite, type*:** everyone for the same reason that is speeding-up the retrieving of the information needed to capture a Pokémon and it to the team. In this way adding/removing/finding Pokémon in/from/of a team is totally handled by the Graph Database, delegating to the Document Database the only task of storing the team name.

If these speeded-up queries are very frequent (see paragraph 2.6), we can also assert that write accesses to the considered attributes are rare: name, sprite and type are constant values of a Pokémon, and as we will see in the paragraph 3.6, *capture rate* is update only once-per-day.

Eventually, since Pokémon nodes are very few w.r.t. other nodes, these redundancies are not very memory-expensive

### 3.4.4 The Analytic collection

As said at paragraph 2.1, admin can consult usage statistics in order to evaluate business plans and other possible optimizations.

To do that, there are two possible approaches:

* Computing analytics each time they need them
* Storing computed analytics in a separated collection and retrieve them every time they are needed

Referring again to our non-functional requirements (par. 2.2), the mechanism that suits best the performance constraint is the second one.

Immagine che contiene testo

Descrizione generata automaticamenteFor this reason, the Document Database hosts also an *Analytic* collection, structured as follows.

This structure is very suitable for the queries presented in the par. 2.1.

As we will discuss in the par. 3.6, the *Analytic* collection is updated daily and using a bunch of strategies to minimize the database stress.

## 3.5 Db properties

### 3.5.1 Availability

A very important non-functional requirement of the application is *availability*.

Indeed, users expect to always have access to the game.

To ensure availability, a cluster of virtual machines has been used, as described in the next paragraph, moreover as reported in paragraphs 3.5.3 and 3.5.4 further mechanism has been projected/implemented in order to guarantee a high level of availability.

To ensure it, not only architectural solutions have been exploited, but also software design ones: as described more in deep in the paragraph 3.6, we decided to defer analytics operations, computing aggregate results only once a day (midnight US) instead of every time data is required.

This means faster response time of the application during the normal usage, reducing load on the servers. On the other hand, a little overhead in the database is generated at that time, and we have to consider that even if we can estimate a low usage rate in the US at midnight, due to the time zone variation somewhere users might experience some delay. This point is better discussed at the paragraph 3.6

### 3.5.2 Replicas

As anticipated in the previous paragraph, the application is provided with replicas support.

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.

The Replicas Mechanism allows us to ensure a high level of availability of the cluster, overcoming possible server’s crashes/breakdowns. Fault-tolerance and also scalability are guaranteed, together with an accurate design of the eventual consistency, presented in the next paragraph.

Even though we achieve a single-server breakdown fault-tolerance, our network is still exposed to a possible down of the link with the cluster.

### 3.5.3 Eventual consistency

As described before, we preferred guaranteeing low latencies and high availability rather than a strict consistency.

To be precise, we wanted to achieve a *read-your-writes* consistency for game players, since they expect that, at the catching/freeing up of a Pokémon, their team and their ranking position is immediately updated taking into account this modifies.

At the same time, we implemented a *causal-consistency* for the Social Network management: a user usually sees immediately his own posts, but in any way it’s ensured that Posts referring to the same Pokémon will be always in order.

Eventually, for what concerns admins’ analytic data, there is *no real consistency warranties* that data will be updated: the relative deferred write operation ensures that data has been correctly memorized and journaled, but when a read operation arrive we don’t know if it will pick up new or old data.

* The read-your-write consistency is been implemented according to the official documentation of Mongo at <https://docs.mongodb.com/manual/reference/read-concern/#read-your-own-writes>.

So write-concern is *majority,* read-concern is also *majority* and read-preference is on the *primary* when possible (*preferred*).

* Social Network’s *causal consistency* is ensured by the Neo4J framework, that is the specific Graph Database we decided to use(see par. 3.7 for further details)
* The *eventual consistency* for usage analytic data is structured this way:

On write operations, the control is returned to the application if the *majority* of the servers in the cluster have acknowledged it and only after the operation has been correctly journaled. As described by MongoDb documentation, this is enough to guarantee that the operation cannot be lost. Other servers’ data will be eventually consistent, anyway reads are performed in just one of the *secondaries*, immediately after taking a *snapshot* of the database, so that the aggregation operations will not deteriorate server’s performance. Since we have no control on which secondary server will be in charge of each read operation, we cannot ensure any kind of consistency, in fact:

* + Read and writes are made from different actors, so is meaningless talking about *read-your-writes* consistency
  + Since there are no sessions, it’s pointless considering a sort of *session-consistency*
  + We cannot ensure *monotonic-read-consistency*: since we don’t know the secondary server that will be in charge of each read operation, it can happen that, given two reads ri—HB🡪ri+1, ri reads from an updated server while ri+1 doesn’t.
  + Theoretically we have no *monotonic-write-consistency* since the connection with the servers crosses Internet, so we cannot know if an older write request will be received after a newer one.

From a practical point of view we are sure that this kind of consistency will be always guaranteed, due the fact that 2 consecutive writes come 24 hours one from the other

* + There are no cause-effect relationship in this kind of data, so no *eventual consistency*

### 3.5.4 Sharding

**N.B.** Despite the fact that we projected here a Sharding mechanism, this is not been implemented in the database cluster.

Sharding consists on partitioning data according to a specific policy. Data can be retrieved in the right server/cluster/partition using a Sharding Key.

A good Sharding algorithm is the one that permits to retrieve quickly data, but also to partition it homogeneously.

In our application we chose to design a **Geographical Sharding**: each partition is in charge of data generated by users coming from a specific bunch of countries. This kind of division is able to guarantee both the properties discussed before.

* It’s easy-to-retrieve because the Sharding Key is a pure key, thus it does not need any computation. As we will see in the chapter 4, it’s convenient to keep in memory (caching) data of the user currently logged: this means that the calculation of the Sharding Key is no more than a simple read access in the main memory
* It’s also homogenous if we plan accurately how to partition countries among servers/clusters. Indeed it’s not realistic to assert equal usage distribution on every country, but could either not being enough partitioning according to continents/areas.

Against this problem the most useful tool provided by the application is the *usage analytic* collection: in this way admins are always updated on the amount of load generated by each country, and so they can plan wisely how and where to dispose servers/clusters.

In this way a further optimization could be applied to the application: whereas a cluster is in charge of a particular area, analytic aggregations could be performed at local midnight instead of at 00:00:00 U.S.

(eg. Let us suppose that, starting from the analytic data provided, admins decided to put a new server on the north of Italy to serve Italy, Switzerland and Austria. We could change the application so that the daemon thread that computes analytic data will go in execution at 00:00:00 Central Europe instead of 00:00:00 U.S.)

### 3.5.5 Pros and drawbacks

Starting from the characteristics of the database infrastructure described in this paragraph, we can state the following considerations:

1. The followed project approach prefers in general performance over storage saving. Thus, to guarantee a good game experience with over 5M of users a discrete storage capability is required.
2. Availability is ensured by always-on servers and use of replicas. However there could be little delays when analytics are written.
3. There is no strict consistency among different servers, for no service. For our purposes, eventual consistencies are been finely designed depending on the specific task. They can be consulted at the paragraph 3.5.3.
4. The architecture is the master-slave one: this means that master can be a single point of failure. Although, DBs adopted are capable of performing an election in case of primary’s fault, in this case the system will be still capable of working but every NON-JOURNALED write will be lost.
5. In spite the fact that this is a performance-centered project, we decide to repeat rankings computing each time a read request arrives. Surely this is not the fastest approach, but we thought that providing a user each time the most updated ranking possible was the best way to encourage him/her to play and try to climb the charts.

## 3.6 Clients, servers, daemon threads

The architecture of our application involves the presence of *clients*, that will be the users’ devices, and of some *servers* that will run in one ore more clusters of machines and that will handle all the data according to the instructions given by clients.

The applicative code runs entirely on the clients: every interaction with the GUI is handled locally and may trigger a send of a request to the database.

In order to minimize servers’ computational load, they do not execute applicative code, but they only are in charge of providing answers to queries.

This means that the application has not been designed as independent from the database infrastructure, but as we will discuss in chapter 4 an accurate information hiding mechanism prevents strong dependencies on the backend implementation.

Apart from the usage analytics (and image download/caching), the applicative code is synchronized with query responses: it stops waiting for an answer from the database.

As already cited before, an analytic function is performed in a deferred fashion every day at 00:00:00, and it is in charge of computing usage statistics rather than updating all Pokémon’s catch rates.

Thus, a *daemon thread* is required: it wakes up at 00:00:00, aggregates data as described in par. 2.5, updates the *Analytics* collection on the Document Database and eventually sleeps for 24 hours.

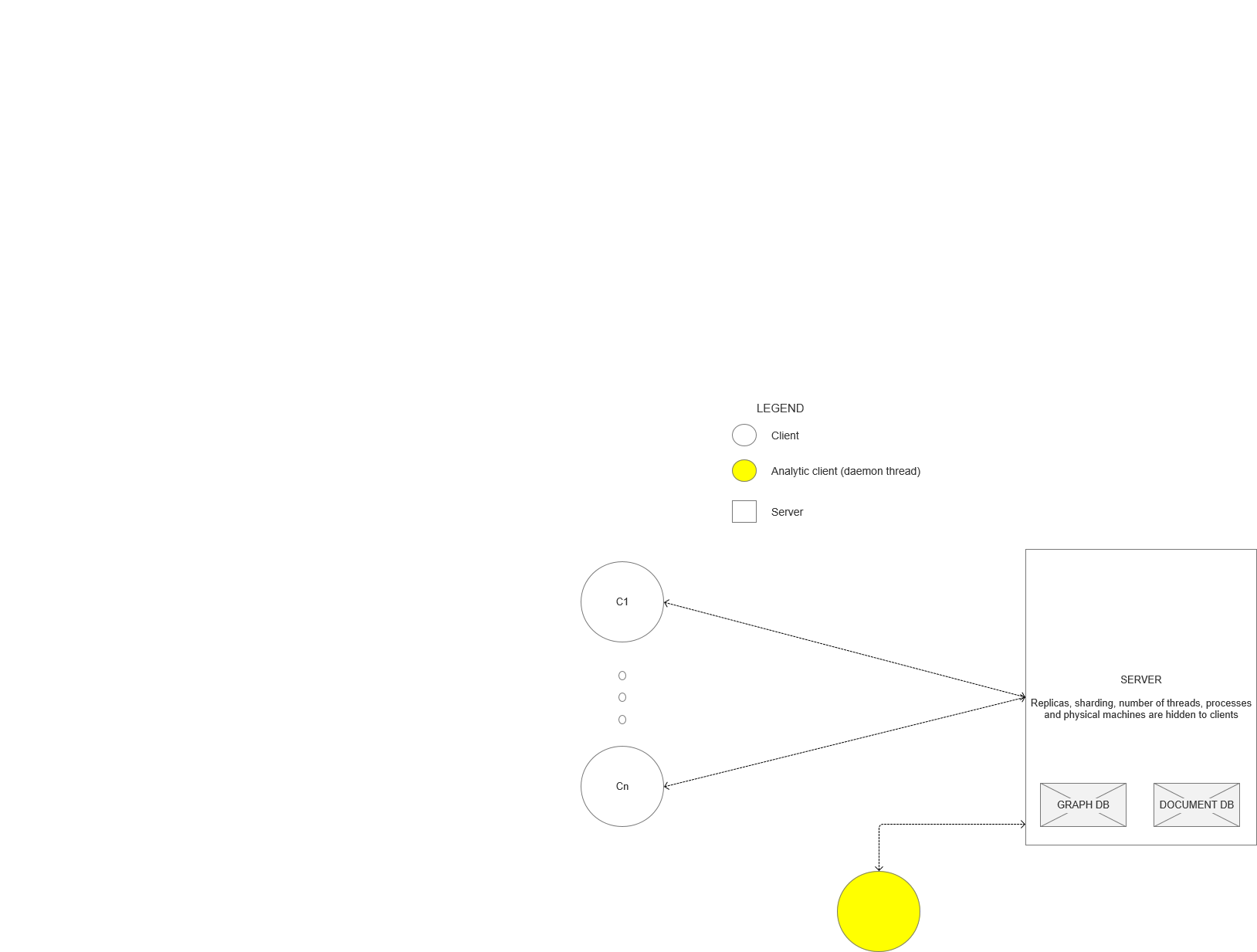
This operation cannot be performed by normal clients, since it can be computed only once for everyone and it would require a user to have his own device always on; anyway we wanted not to make it be performed by servers, both to reduce their computational efforts and because we cannot take for granted physical access to the server’s file system.

For those reasons we decide to introduce an additional “special” client in which to host the *daemon thread*. This client must be always-on, does not need high computational power nor a large amount of memory and runs his own piece of applicative code.

To simulate its presence, we implemented its methods/classes in a separated module of the application, so that we have no coupling at all between normal clients and this one.

**N.B**. : admin’s devices are normal clients!

In the following picture a schema of the application architecture is reported.



## 3.7 Technologies and frameworks

The most important technologies and framework we chose to build up our project are discussed here. For a more accurate report of these refer to chapter 4.

* As Document Database we chose is MongoDB: this framework is very easy-to-use, is suitable for large data collections and supports indexing, clustering, replica sets and Sharding. DMBS is optimized for analytics, and the query language is very well integrated with the most famous programming languages.
* The Graph Database selected is Neo4J, an intuitive and powerful framework, supports indexing, clustering and replicas. Neo4j provides a very expressive query language and some fundamental ensured properties, like Safety (fault-tolerance), Scaling (through Replicas) and Causal Consistency.
* The main programming language used is Java 8. It provides JavaFX support for GUI design, drivers for the communication with databases and it is compatible with all the main available frameworks/plugins
* Maven as build and dependency manager, it is been used to organize the code, support pre-build tests, import quickly dependencies and dividing the project into the two modules as explained in the previous chapter
* Mockito for building up very simple mock-ups in order to conduct tests on the interaction between classes
* Junit for testing classes/methods and for fast bug discovering/recovering

# 4 Implementation Stage

## 4.1 Package structure

Package structure decision was as important task in PokeMongo, we wanted to ensure an high level of readability and maintainability.

Although the classical “root package” which specifies the “domain.company.projet”, in our case “it.unipi.dii.lsmsd.pokemongo”, all the packages are structured by layers. In this way, we decided to name the packages according to they function architecturally rather than their identity according to the business domain. Here the structure:

Immagine che contiene tavolo

Descrizione generata automaticamente

We tried to maintain the name of the packages as simple as possible, and in a way they are all easy to read and to understand.

We also followed the convention of having the first character in the package names in lower case, in order to avoid conflicts with class or interface names.

### 4.1.1 Package analysis

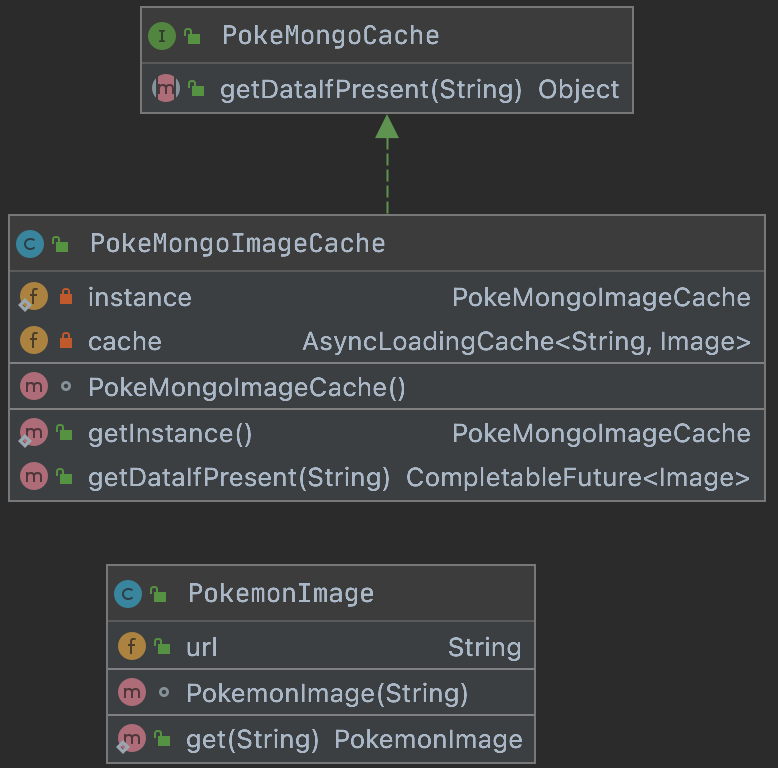
The “bean” package contains few classes that are used as beans while the application runs.

Immagine che contiene testo

Descrizione generata automaticamente

|  |  |
| --- | --- |
| Class Name | Short Description |
| User | The User class is used for instantiating object that refers to a specific user |
| Pokemon | The Pokemon class is used for instantiating object that refers to a specific Pokemon |
| Post | The Post class is used for instantiating object that refers to a specific Post. Responses (aka subPosts) are considered post also. |
| Analytic | This class is used for containing the information regarding a particular day. |
| CountryData | Used in the Analytic bean, it contains the information regarding a single country and the analytic strictly associated to it. |

### 4.1.2 Package analysis: cache

The cache package contains classes that are helpful for caching images, we will talk about that in chapter 4.3.2. Despite what written above, this is one of the few packages that has a feature logic structure inside. We maintain in this package not only the classes/interface that handle the caching functionality, but also a javafx class extension which is PokemonImage. This class is strictly connected to the caching systems, because it contains the image we want to cache. We decided to use this approach to have a cleaner look and an easier maintainability for the caching systems. 

|  |  |
| --- | --- |
| Class Name | Short Description |
| PokeMongoCache | Simply an interface. |
| PokeMongoImageCache | The implementation of the interface described. |
| PokemonImage | An Image (javaFX) extension that will contains the image we want to show to the user in the GUI |

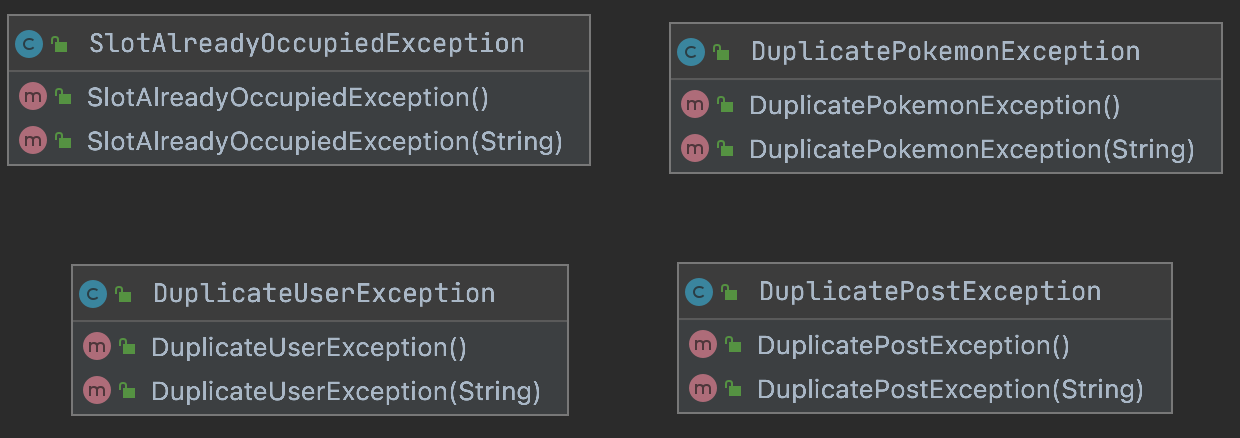
### 4.1.3 Package analysis: dataanalysis

This package is used for instantiating factory structures about the data analysis we made in the project. Every factory is dependent of an interface. Immagine che contiene testo, screenshot, nero, targa

Descrizione generata automaticamente

|  |  |
| --- | --- |
| Class Name | Short Description |
| AdminAnalytics | Simply an interface for the analytics related to the admin user |
| AdminAnalysisFactory | Has a static method that returns a specific implementation of the interface AdminAnalytics |
| UserRanker | Simply an interface for the analytics for user ranking |
| UserRankerFactory | Has a static method that returns a specific implementation of the interface UserRanker |
| PokemonRanker | Simply an interface for the analytics for pokemon ranking |
| PokemonRankerFactory | Has a static method that returns a specific implementation of the interface PokemonRanker |

### 4.1.4 Package analysis: exceptions

This package contains classes that extend the class Exception of Java. 

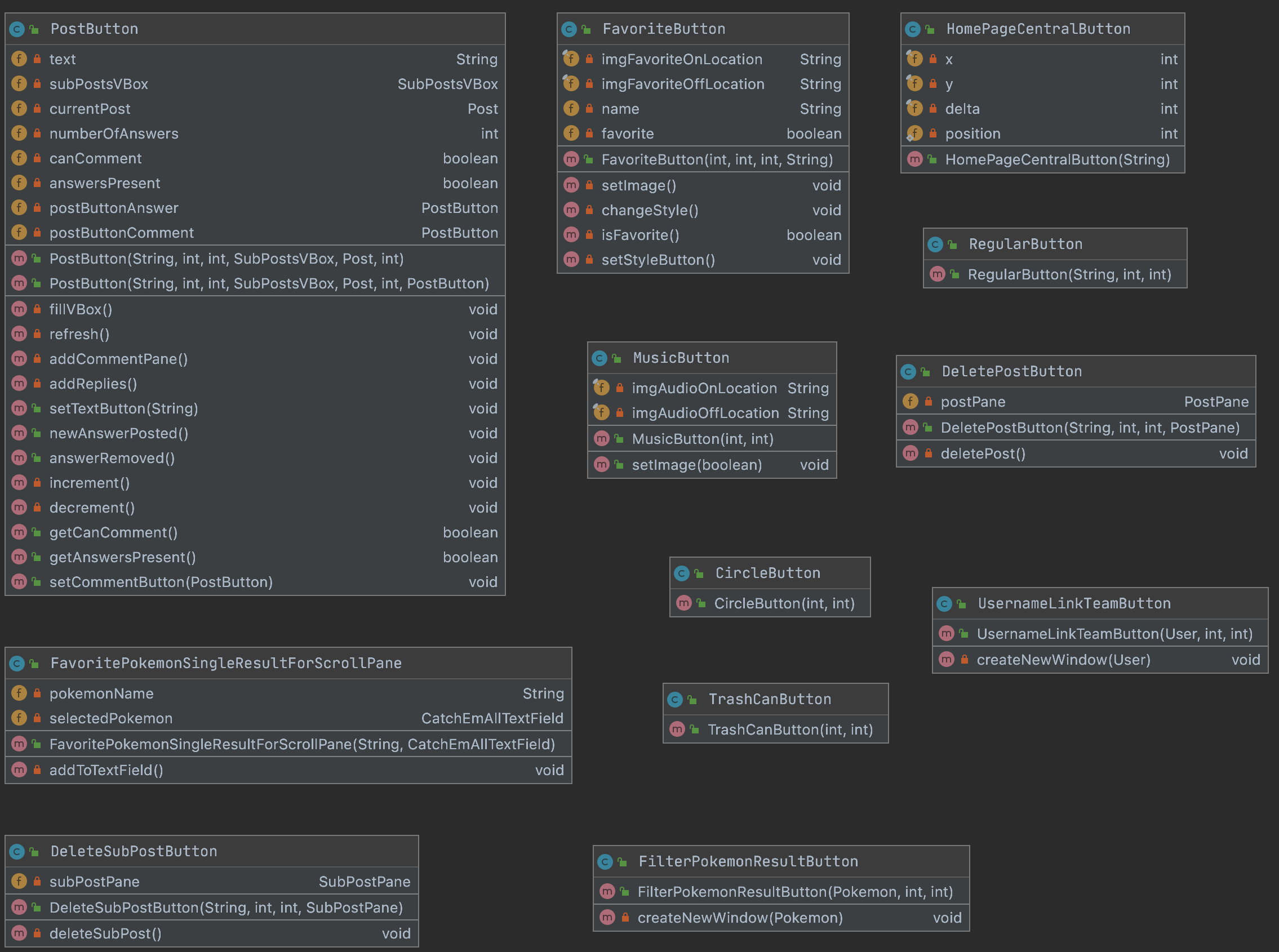
|  |  |
| --- | --- |
| Class Name | Short Description |
| SlotAlreadyOccupiedException | Exception thrown when a user try to catch a Pokemon and he has the slot he want to use already occupied by one other Pokemon |
| DuplicatePokemonException | Exception thrown when an admin try to insert a Pokemon that is already present |
| DuplicateUserException | Exception thrown when an anonymous user try to create a register user, but the username he writes is already taken. |
| DuplicatePostException | Exception thrown if an identical Post is created. |

### 4.1.5 Package analysis: javafxextensions

In this package are present 11 sub-packages, any of them related to a specific extension of a JavaFX Node.

#### 4.1.4.1 javafxextensions: buttons

Here are present all the classes that extend Button from JavaFX



|  |  |
| --- | --- |
| Class Name | Short Description |
| HomePageCentralButton | Specific button for the HomaPage. |
| MusicButton | Button for turning the music on/off |
| RegularButton | For creating buttons like “BACK”, “SUBMIT”, etc… |
| TrashButton | Button for eliminating a Pokemon in the Team |
| CircleButton | Helpful for creating button with a circular shape |
| PostButton | Specific button for submitting a comment in the post section of a Pokemon |
| DeletePostButton | Button for deleting a Post |
| DeleteSubPostButton | Button for deleting a SubPost (aka response) |
| FilterPokemonResultButton | Specific button for displaying the name of a Pokemon in a query result. At the click it creates a new Stage with the information about the Pokemon (check PokemonWindowGroup). |
| FavoritePokemonSingleResultForScrollPane | This button is used for showing the name of the Pokemon than are Favorite. Clicking on it will be a shortcut for capturing the Pokemon the button says about. |
| UsernameLinkTeamButton | Specific button for displaying the username of a User in a query result. At the click it creates a new Stage with the team of the User (check TeamUserWindowGroup). |

#### 4.1.5.2 javafxextensions: charts

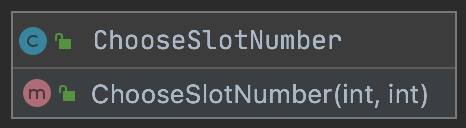
It contains a class that extends LineChart from JavaFX.

Immagine che contiene testo

Descrizione generata automaticamente

|  |  |
| --- | --- |
| Class Name | Short Description |
| LineChartThirtyDaysFacotry | The class helps for the creation of different Line Charts, which can have different meanings (e.g. number of logins, number of users, …)  This is used for every plot in the application. |

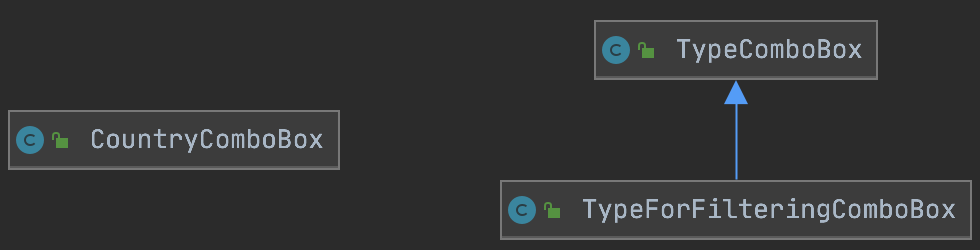
#### 4.1.5.3 javafxextensions: choicebox



|  |  |
| --- | --- |
| Class Name | Short Description |
| ChooseSlotNumber | Choice box that lets the user to select the slot for saving the Pokemon in captured |

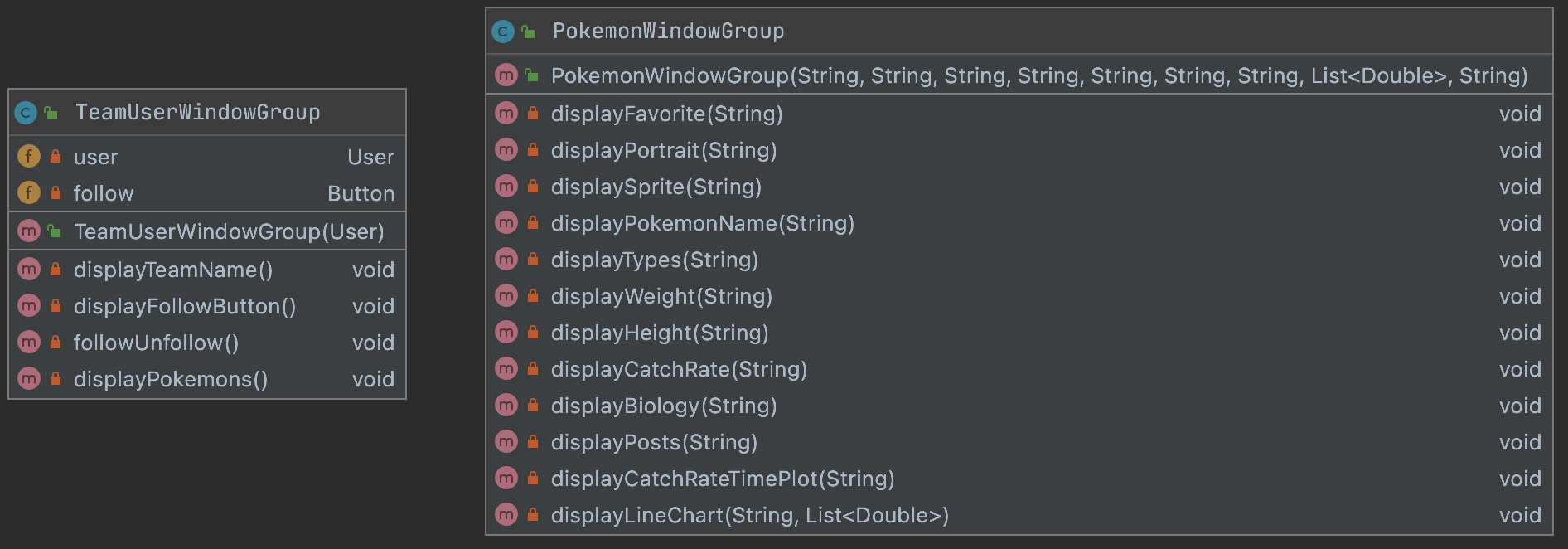
#### 4.1.5.4 javafxextensions: combobox

A ComboBox can be seen as a ChoiceBox, the user select the elements in it in the same way.



|  |  |
| --- | --- |
| Class Name | Short Description |
| CountryComboBox | Let the user to select the country |
| TypeComboBox | General ComboBox for choosing the type of a pokemon |
| TypeForFilteringComboBox | Specific TypeComboBox for the filtering Pane. |

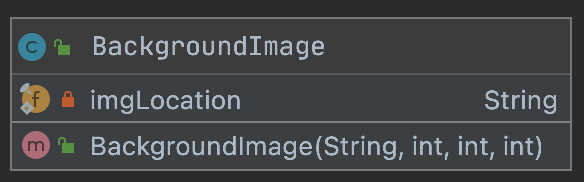
#### 4.1.4.5 javafxextensions: group

The group extensions are used for creating new windows with particular information regarding something. 

|  |  |
| --- | --- |
| Class Name | Short Description |
| TeamUserWindowGroup | Instantiates all the Node that are needed for creating the window which display the team of a specific user |
| PokemonWindowGroup | Instantiates all the Node that are needed for creating the window which display the information of a specific Pokemon along with the posts related to it |

#### 4.1.5.6 javafxextensions: imageviews

Extensions of ImageView



|  |  |
| --- | --- |
| Class Name | Short Description |
| BackgroundImage | Helpful for adding image in the background. |

#### 4.1.5.7 javafxextensions: labels

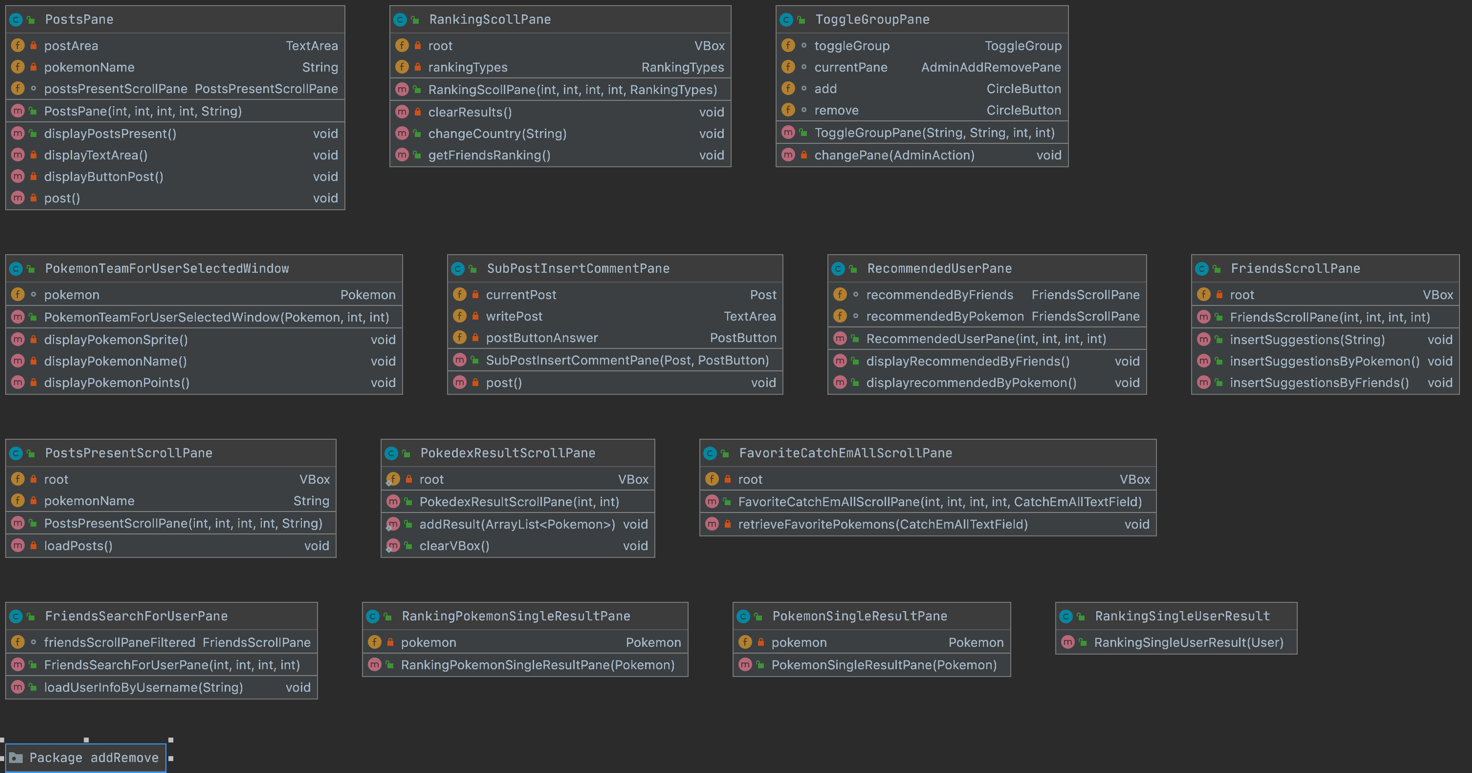
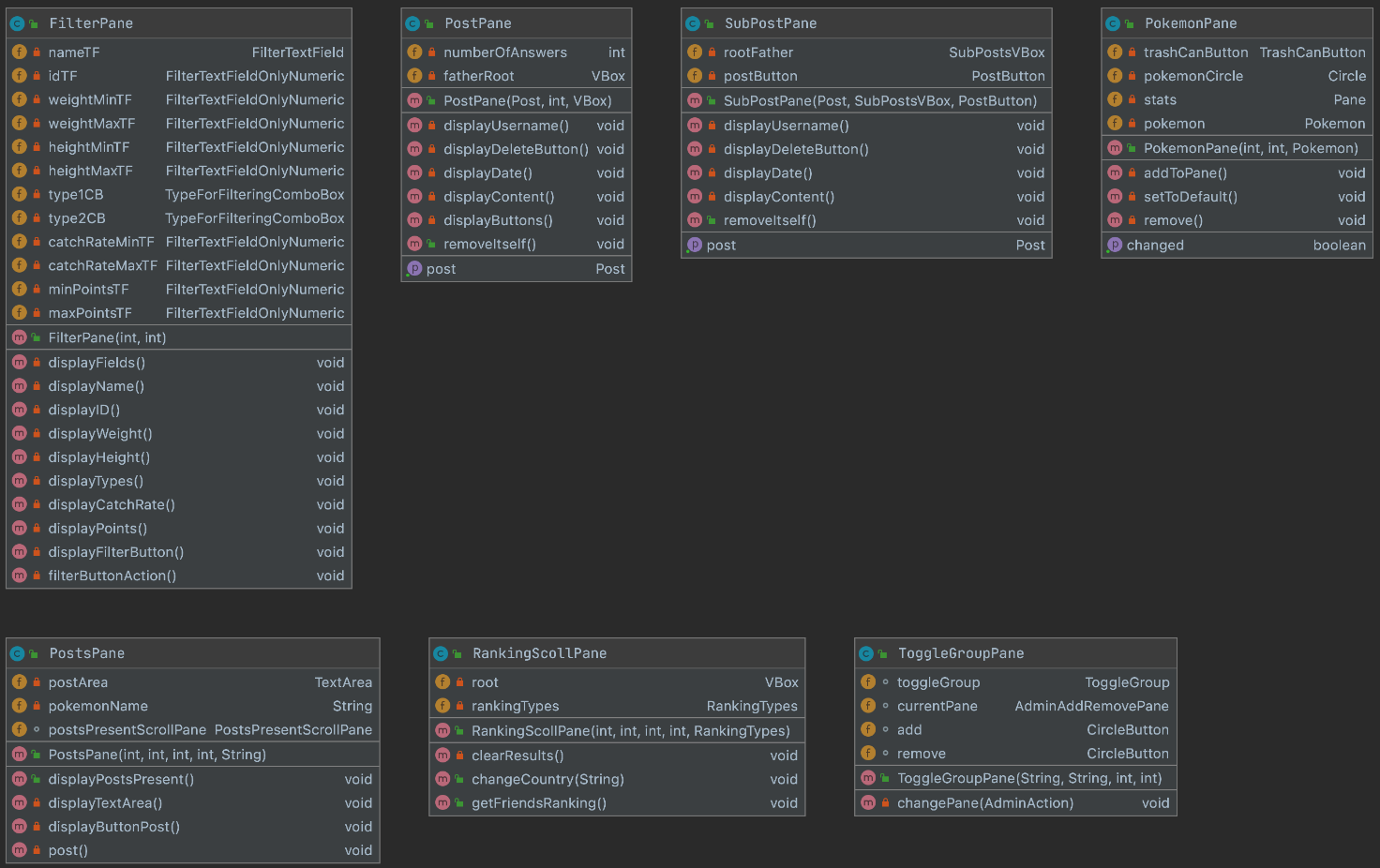
This package contains different types of labels useful for different situations.



|  |  |
| --- | --- |
| Class Name | Short Description |
| InvalidFormEntryLabel | Used when an error occurs at the filling of an entry in a form. |
| PokemonWindowLabel | A specific Label that is used in the Stage created with the information of the a specific Pokemon |
| TitleLabel | Used for creating title in a prefix position. |
| FieldRelatedLabel | Used to indicate what a TextField is related to |
| FieldLabel | Used for the labels in the filter Pane |

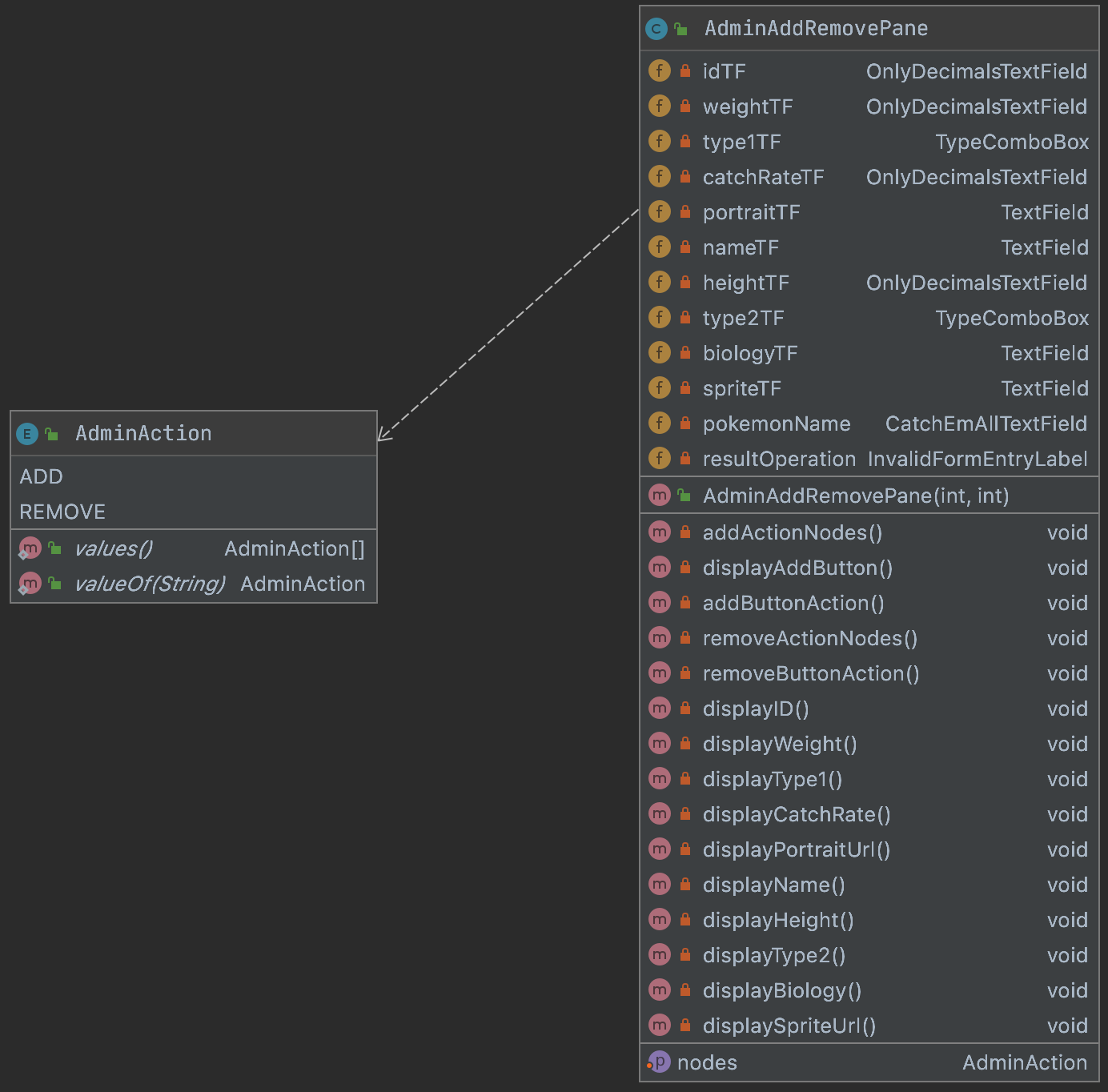
#### 4.1.5.8 javafxextensions: panes

The Panes are the most important JavaFX extension we made in the project. The Panes help the system to be more modular. Modularity by the Panes is archived by dividing every complex components of the GUI in sub components that can be used and modified as stand alone (this gives us also an high level of maintainability). Only one type of Pane is standing separated by the others, inside the addPane package contained in the pane package, this because this pane is strictly connected to an enum that is present in that same folder (we just want to divide this particular enum, to the rest of the panes that, in fact, do not interact with it).



|  |  |
| --- | --- |
| Class Name | Short Description |
| FilterPane | Specific pane for the filtering |
| PostPane | Specific pane for a post (not a response) |
| SubPostPane | Specific pane for a sub-post (aka response) |
| PokemonPane | Specific pane for showing the Pokemon in the TeamScene |
| PostsPane | Pane that contains all the post part of a Pokemon Window |
| RankingScrollPane | Pane that can be scrolled. It contains other panes that are specific for something (e.g. a user, a Pokemon). |
| ToggleGroupPane | Specific pane for creating a toggle group. |
| PokemonTeamForUserSelectedWindow | Specific pane for showing a single Pokemon in the other user window. |
| SubPostInsertCommentPane | Specific pane that is used to create the Nodes for a response to a Post. The need of that comes by the fact the TextArea and the button in it should be horizontal to each other (impossible in the VBox this Pane it’s used). |
| RecommendedUserPane | Specific Pane for the recommended section in the Friends page. |
| FriendsScrollPane | Specific ScrollPane to visualize friends users (an even the one recommended). |
| PostsPresentScrollPane | Specific ScrollPane to visualize a limited number of Posts. |
| PokedexResultScrollPane | Specific ScrollPane to visualize the result of a filtering operation. |
| FavoriteCatchEmAllScrollPane | Specific ScrollPane to visualize the Pokemon set as favorite |
| FriendsSearchForUserPane | Specific pane for searching an user (Friends scene) |
| RankingPokemonSingleResultPane | Specific pane to be inserted in a ScrollPane extension. It gives some information about the Pokemon (used in the Ranking) |
| PokemonSingleResultPane | Specific pane to be inserted in a ScrollPane extension. It gives some information about the Pokemon (used in the Pokedex) |
| RankingSingleUserResult | Specific pane to be inserted in a ScrollPane extension. It gives some information about the Pokemon (used in the Ranking) |

The addRemove package is characterized of these classes:



|  |  |
| --- | --- |
| Class Name | Short Description |
| AdminAddRemovePane | Specific Pane for the ADD/REMOVE scene. |
| AdminAction | Contains the name of the action that an admin can do regarding the Pokemon management. |

#### 4.1.6 Package analysis: persistence

The persistence package contains all the classes related to the communication with the databases. In the image below you can see how it is structured. The Factories classes are used as said before about the Ranking. Immagine che contiene testo

Descrizione generata automaticamente

|  |  |
| --- | --- |
| Class Name | Short Description |
| Database | Shared interface among Databases, defines remote connections and structures of basic CRUD operations |
| UserManager | Shared interface for the managing of users, defines the fundamental operations |
| PokemonManager | Shared interface for the managing of Pokemon, defines the fundamental operations |
| UserNetworkManager | Shared interface for the managing of Pokemon, defines the fundamental operations |
| PostManager | Shared interface for the managing of Post, defines the fundamental operations |
| MongoDbDatabase | Implementation of Database specific for MongoDB, to be extended with other classes. |
| UserManagerOnMongoDb | Extension of MongoDBDatabase, handles the user related queries in MongoDb |
| AdminAnalysisOnMongoDb | Extension of MongoDBDatabase, handles the admin related queries in MongoDb |
| PokemonManagerOnMongoDb | Extension of MongoDBDatabase, handles the Pokemon related queries in MongoDb |
| Neo4jDbDatabase | Implementation of Database specific for Neo4j, to be extended with other classes. |
| UserNetworkManagerOnNeo4j | Extension of Neo4jDbDatabase, handles the user related queries in Neo4j |
| PostManagerOnNeo4j | Extension of Neo4jDbDatabase, handles the post related queries in Neo4j |
| TeamManagerOnNeo4j | Extension of Neo4jDbDatabase, handles the Team related queries in Neo4j |
| Filter | Enum that contains the names of the filters used in the filter pane. |
| UserNetworkManagerFactory | Has a static method that returns a specific implementation of the interface UserNetworkManager |
| PokemonManagerFactory | Has a static method that returns a specific implementation of the interface PokemonManager |
| TeamManagerFactory | Has a static method that returns a specific implementation of the interface TeamManager |
| UserManagerFactory | Has a static method that returns a specific implementation of the interface UserManager |
| PostManagerFactory | Has a static method that returns a specific implementation of the interface PostManager |

### 4.1.7 Package analysis: security

It contains the PasswordEncryptor class, we will discuss it in chapter 4.3.3

### 4.1.8 Package analysis: userInterface

The userInterface package contains all the classes that are related to the creation of the GUI. The approach taken is a hierarchical one, in order to increase the modularity of the code. Immagine che contiene testo

Descrizione generata automaticamente

|  |  |
| --- | --- |
| Class Name | Short Description |
| PokeScene | General scene that contains the elements shared by every scene |
| PokeSceneWithHeader | General scene with only the Header in it (the header contains the username of the user logged and the number of pokemon) |
| PokeSceneWithTitle | General scene with only the title |
| SignUp | Sign up page |
| PokeSceneWithBlastoiseCharizard | General scene that extends PokeSceneWithTitle and adds to the scene the image of Charizard and Blastoice |
| LogIn | The first scene the user will see at the opening of the application. As the name suggests the class displays the Nodes regarding the LogIn |
| PokeSceneWithHeaderAndAggreateBlastoiceCharizard | General scene that combines the PokeSceneWithHeader and the PokeSceneWithBlastoiceCharizard |
| HomePage | As the name suggests the class displays the Nodes regarding the HomePage |
| PokeSceneWithHeaderAndBackButton | General scene that contains the Header and the Back Button |
| RankingScene | As the name suggests the class displays the Nodes regarding the Ranking |
| Pokedex | As the name suggests the class displays the Nodes regarding the Pokedex |
| TeamScene | As the name suggests the class displays the Nodes regarding the Team |
| CatchEmAll | As the name suggests the class displays the Nodes regarding the CatchEmAll page |
| Settings | As the name suggests the class displays the Nodes regarding the settings |
| AnalyticsScene | As the name suggests the class displays the Nodes regarding the admin analytics scene |
| Friends | As the name suggests the class displays the Nodes regarding the friend scene |
| RemoveUserScene | As the name suggests the class displays the Nodes regarding the remove user scene |
| AddRemovePokemon | As the name suggests the class displays the Nodes regarding the scene where the admin can add or remove a pokemon |
| SceneNames | Enum containing the different types of scene. Helps for the managing the changing in the scenes. |
| RankingTypes | Enum containing the different types of ranking |
| MusicPlayer | Handles the music. |
| CurrentUI | Handles the current UI. This is the bone of the entire package. |

### 4.1.9 Package analysis: utils

This package contains utility classes. Immagine che contiene testo

Descrizione generata automaticamente

|  |  |
| --- | --- |
| Class Name | Short Description |
| FormValidatorPokeMongo | Used for check if a field is well filled |
| LoggerThread | A thread that writes information about all the action taken by the code. |

4.1.10 Obfuscation

Our package structure organization gave us the possibility to exploit code obfuscation. We use code obfuscation in the way to hide how the connection of the database is done. To do that we limited some classes to have only a package scope and, to interact with them, we use the Manager classes presented before.

4.2 APIs and SPIs

4.3 Main tools

For enhancing our performances and for giving to the user a better application, some tool are used. We focus in this chapter about: GSON, caching system, password encrypt and the logger.

4.3.1 GSON

Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code of.

We use Gson mainly for communicating with MongoDb and for setting the configuration information.

4.3.2 Caching mechanism and multimedia management

The Pokemon game series became famous for the creature a player can capture and use, so we thought to include the Pokemon images also in our project.

The images about the Pokemons are not stored locally, because this will increase our project size too much. To avoid this huge increment we take all the images, only when they are needed, from a GitHub repository. Here a problem came: loading a lot of images would slow down our system and that is against our no-functional requirements. The caching comes then in handy.

public class PokeMongoImageCache implements PokeMongoCache {

//Singleton

private static PokeMongoImageCache *instance*;

private AsyncLoadingCache<String, Image> cache;

public static PokeMongoImageCache getInstance() {

if (*instance* == null) {

*instance* = new PokeMongoImageCache();

}

return *instance*;

}

PokeMongoImageCache(){

cache = Caffeine.*newBuilder*()

.expireAfterAccess(10, TimeUnit.*MINUTES*) //After this time without read/write the resource is deallocated

.maximumSize(1000) //The number of images

.buildAsync(k -> PokemonImage.*get*(k));

}

public CompletableFuture<Image> getDataIfPresent(String url){

Logger.*vlog*("Attemp to get image at: " + url);

return cache.get(url);

}

}

What the cache does is simply store asynchronously an image. Why asynchronous? If the operation would be done in a synchronous way the user has to wait that the image is properly store for seeing and using the UI. Using an asynchronous way the user can interact with the UI even if the images are already loaded, this create a better application usage.

For creating all this functionality we used the Caffeine API.

4.3.3 Password Encryptor

4.3.4 Logger

4.4 Analytics queries

4.4.1 User Rankings

4.4.2 Pokémon Rankings

4.4.3 Usage Statistics

4.4.4 Dynamic Catch Rate

4.5 Business logic

4.5.1 Points computing

4.5.2 Dynamic Catch Rate computing

# 5 Test stage

## 5.1 Privacy and security

### 5.1.1 Privacy issues

The only data about a user stored and managed by the application is just the one submitted by the user himself at login time, plus a timestamp of the last login.

The only data about a user visible by other users are: *username, country, team name, points*.

The timestamp of the last login is used for applicative purposes only, and it’s aggregated in evaluating the Analytics. For no reason admins or other users can access to the last login timestamp of a user

### 5.1.2 Password Encryption

As discussed in the paragraph 4.3.3, all the users’ passwords are strongly encrypted and they are never compared using their plain text version.

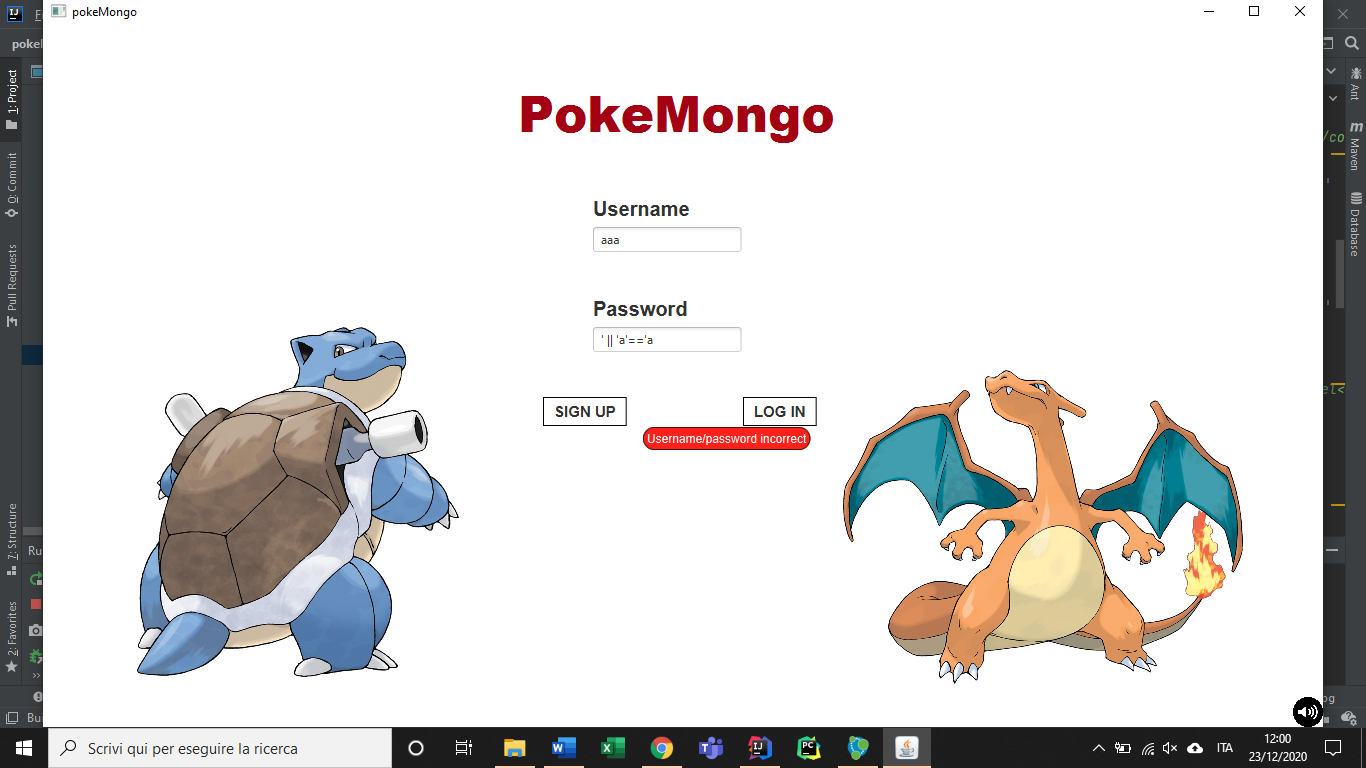
The encryption is performed immediately after the click on the *submit* button, this means that no module/class/driver/DB of the application can have access to the plain text password, but the Password Encryptor itself.

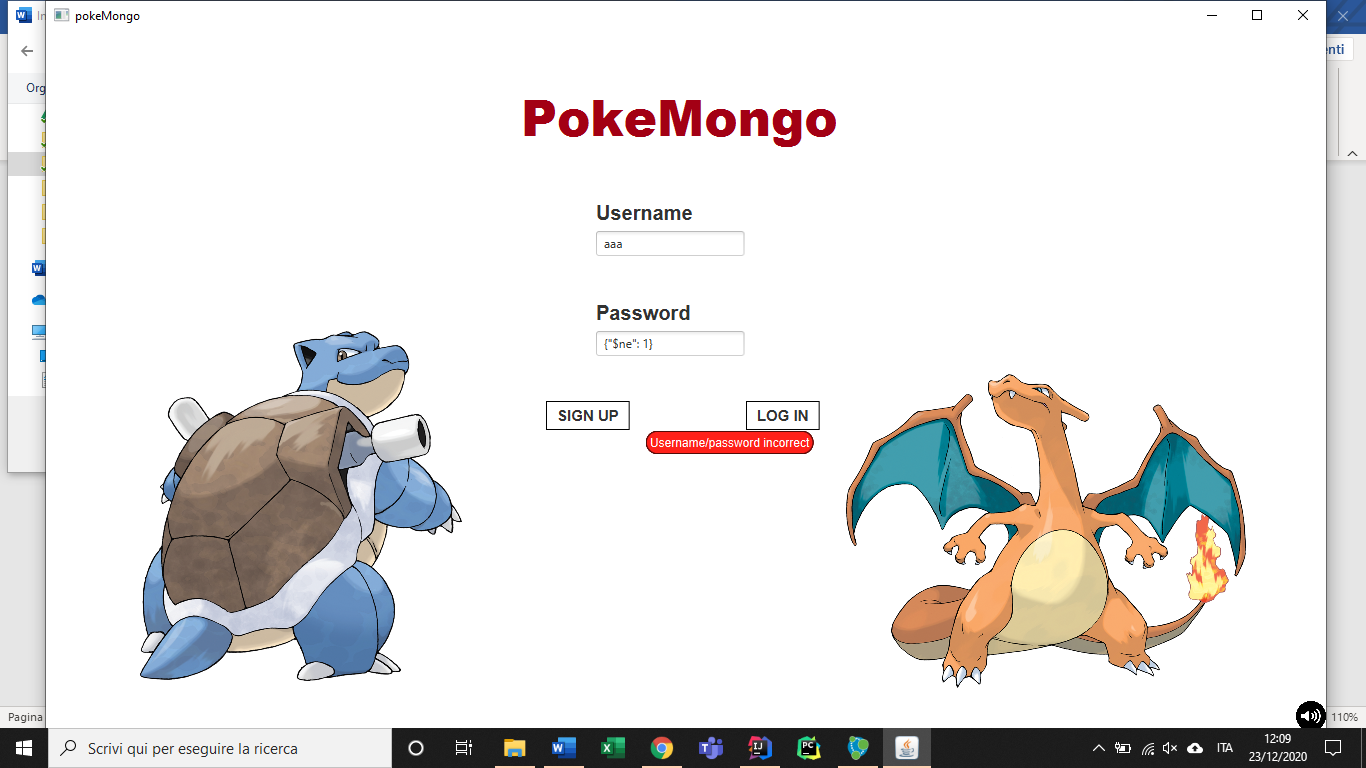
Moreover, given the encrypted password, there is no way to retrieve back the plain text one, nor to correctly access to the application using the hashed one.

The application does not deal with Record&Playback attacks, anyway having no access to the source Java code is very difficult to use it to successfully steal an account

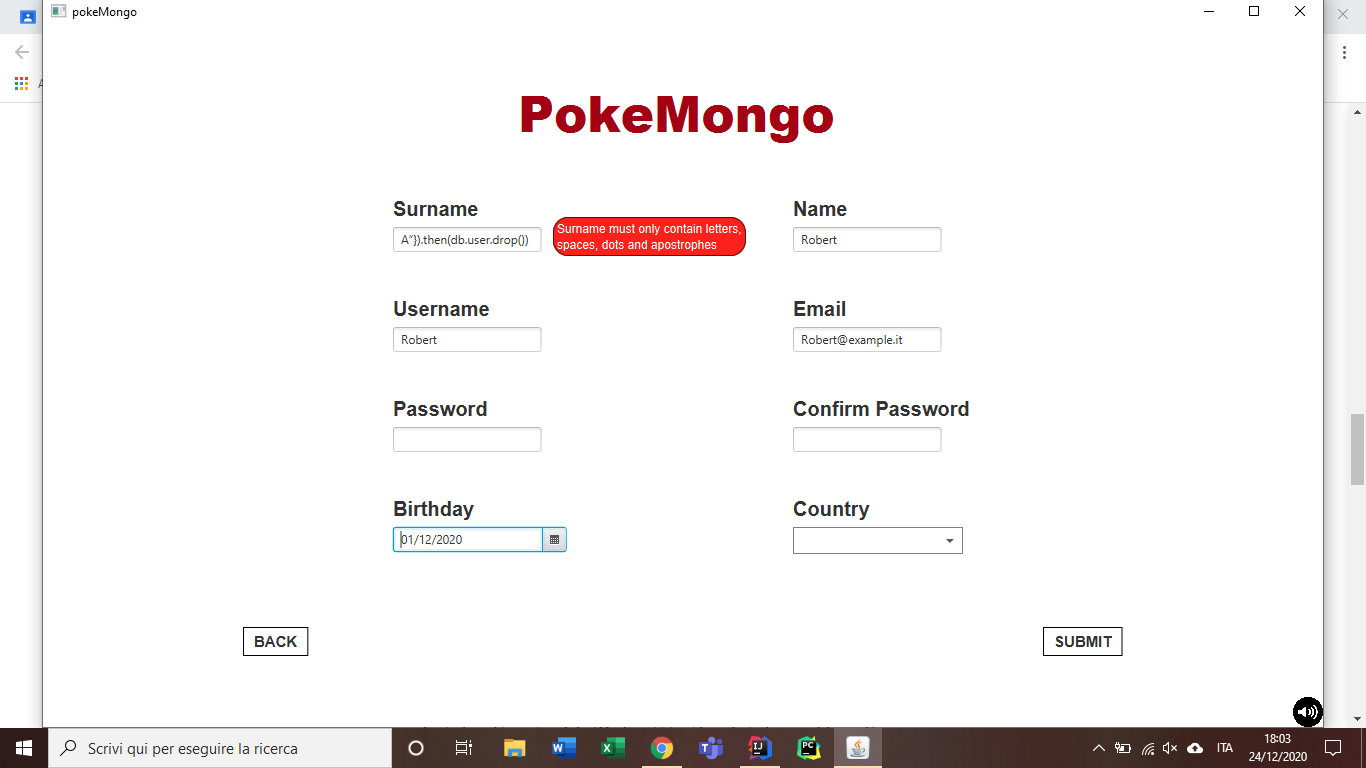
### 5.1.3 Input injections

**N.B.** In the real implementation we used an input obfuscation to hide password input. In the following example it has been removed for demonstrative purposes only.

Example 1: usernames are real registered username, passwords are famous MongoDB injection strings.



Example 2: input validation on the signup form blocks code injection attempts (it would have removed all the collection)



5.2 Unit Tests

5.3 Robustness

5.4 Performance

5.3 Robustness

5.4 Performance