UNIVERSITÀ DI PISA

Computer Engineering, Artificial Intelligence and Data Engineering

Large-Scale and Multi-Structured Database

PokèMongo

Project Documentation

TEAM MEMBERS:
Edoardo Fazzari
Mirco Ramo
Olgerti Xhanej

Academic Year: 2020/2021

Contents

1 — Introduction

PokeMongo is a gaming application in which **Users** compete each other to build up the best **Team** choosing between the set of **Pokémons** available.

1.1 Description

Every **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.

A **User** 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 or commenting on them.

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 *Point*s.

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.

2 — Analysis

2.1 Functional Requirements and Use Cases

2.1.1 Use Cases List

- An *unregistered user* can
  - Register
- A *registered user* can
  - Login
  - Consult *Pokèdex*
    - Search by *Name*
    - Search by *Type(s)*
    - Search by *Pokédex ID*
    - Search by *Catch Rate*
    - Search by *Points*
    - Search by **Pokemon** characteristics like *Height* or *Weight*
  - Consult *Ranking*:
    - Most popular **Pokèmon** among all **Users**
    - Most popular **Pokèmon** in each *Country*
    - Best World **Teams**
    - Best Teams among Friends
    - Best Teams by *Country*
  - Find **Users**:
    - See recommended **Users** based on common friends
    - See recommended **Users** based on common Pokémon interests
    - Find **Users** by *username*
    - Follow/Unfollow them
  - Interact with **Pokèmon** network:
    - Insert/Remove a **Pokémon** in his/her own favorite Pokémon list
    - Create a **Post** on a **Pokémon** to share opinions
    - Add answers to **Posts**

    - Follow/Unfollow them
    - The post owner can also remove the **Post** at his/her will
  - **Team** handling:
    - Remove **Pokemon** from the **Team**
    - View **Team**
    - Change name of 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 favourites
    - Try to catch a **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 Pokèballs*
    - Mute/Unmute Music
  - At each time a *pokèmon name* is visible:
    - By clicking on a *Pokémon name*, visualize all the information about it

- An *admin* can
  - Sign In
  - Add **Pokèmon** to the *Pokédex*
  - Remove **Pokèmon** from *Pokédex*
  - See the number of registered **Users** in time
  - See the numbers of login per day
  - See the numbers of login per day in every *Country*

- Remove a **User** from the system
- Remove **Posts/Answers** from the system
- Consult *Rankings*
- Logout

• The *system* should

- Daily update Pokeball number of each **User**
- Periodically update **Pokèmon** *catch rates* based on the number of users that own that **Pokèmon**
- Update **Team** points if the **User** has six **Pokémons** of different *types*
- Periodically compute usage statistics to be consulted by the administrators

2.1.2 UML Use Cases Diagram

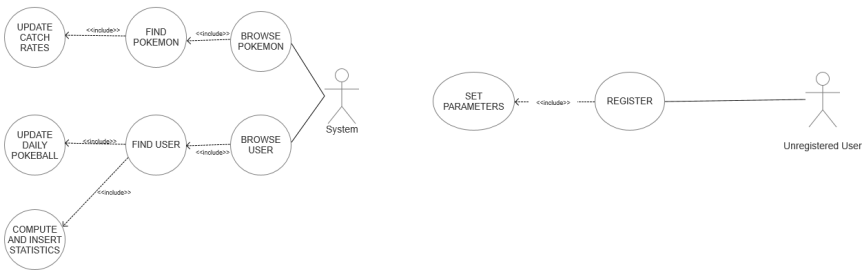


Figure 1: Use Case Diagram 1

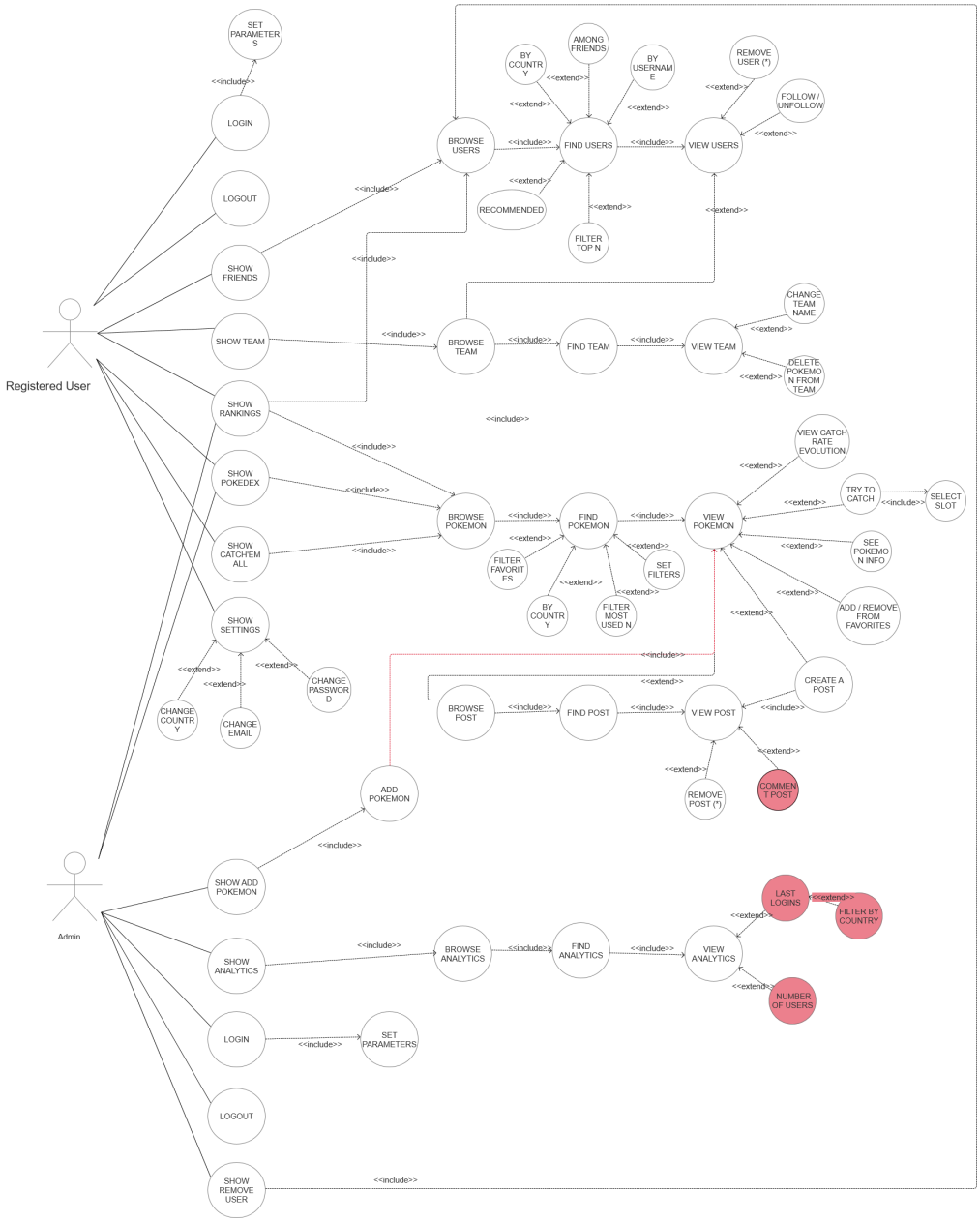


Figure 2: Use Case Diagram 2
(*) only for the User who created the Post and Admins, in Red Browse-find-view comments and browse-find-view answers had not been reported)

2.2 Non-Functional Requirements

- The application should guarantee a high availability. 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** 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 User 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:

1. **Pokèmon Data** → `https://pokeapi.co`, `https://bulbapedia.bulbagarden.net/wiki`
2. **Countries data** → `https://gist.github.com/kalinchernev/486393efcca01623b18d`
3. **Data for the generation of realistic users** → `https://github.com/smashew/NameDatabases/blob/master/NamesDatabases/surnames/all.txt`

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, **for 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

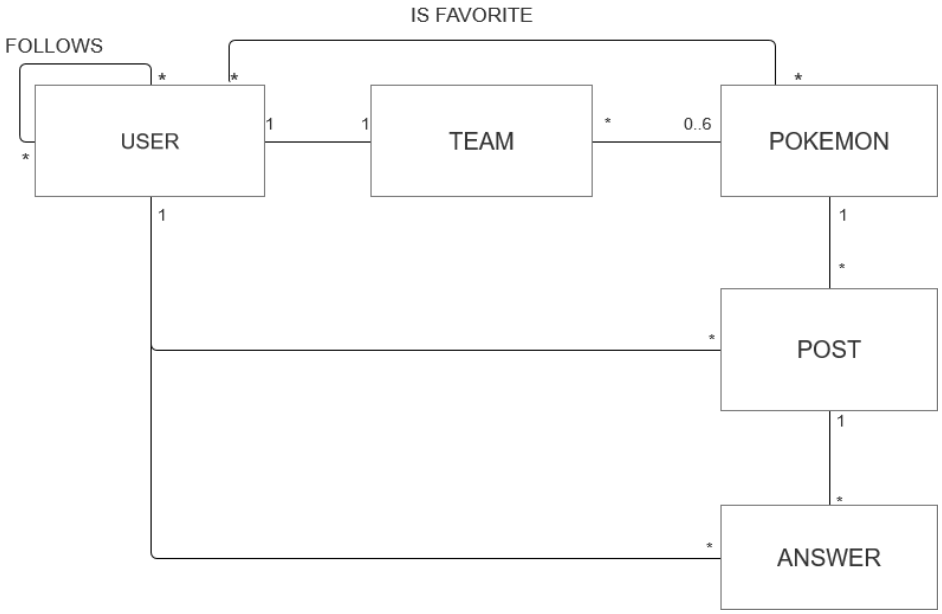


Figure 3: UML Entity Diagram

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

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

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

4. A **User** can have many favourites **Pokémon**. A **Pokémon** can be favourite of many **Users**.

5. 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.

6. 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 favourites
- 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 favourites **Pokémons**
- Modify **User** settings (*email, password, country*)
- Update **Team**'s *name*
- Update **Team**'s *points*
- Update **Pokémon**'s *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 favourite 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/Answers** 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 favourite 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

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 two 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 due to the requirements of this application. There is a 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

NOTE: the queries implementation will be presented in the Chapter 4

- 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**'s *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 **Pokèmons**.

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 (source: 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

Figure 4: 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**

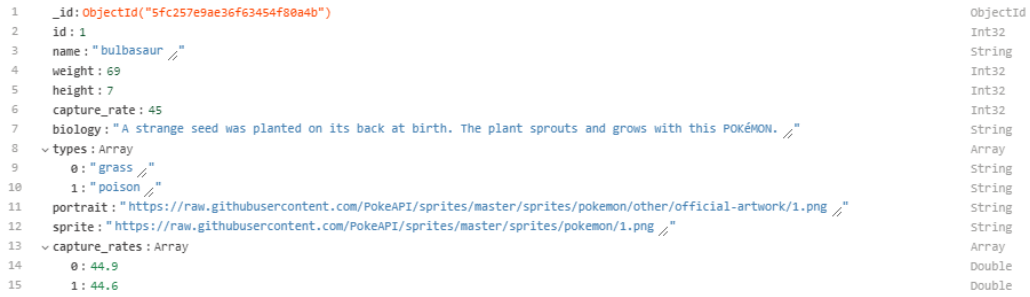


Figure 5: Pokèmon Collection

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:

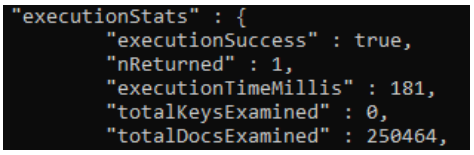
Type	Query
W1	Insert a new *username* at registration time of an arbitrary **User**
W2	Remove a *username* when an admin delete's a **User** from the system
R1	Check uniqueness of a *username* at registration time
R2	Check **User**'s credential at login time
R3	Find a **user** by *username* when a new follow request is submitted

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

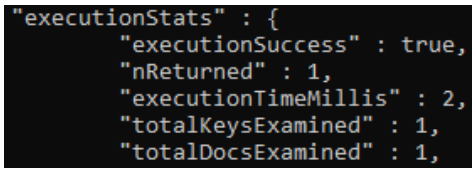
popular query among the network structure of the application, say 30 times the number of registered **Users**: we find out that read operations on this field are about 130 times the number of write operations. Now consider MongoDb performances with and without using an index on the username field, in a Database populated by 250k users.

```
> db.user.find({username:"eee"},{username:1}).explain("executionStats")
```

After submitting the previous command the following results are obtained.



(a) Results without index



(b) Results with index

In the picture on the left 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. On 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 examine 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**.

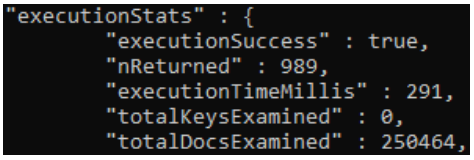
Country As seen before, starting from the application queries we demonstrate the benefits of an index in the field *country*.

Type	Query
W1	Insert the *country* data at registration timer
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

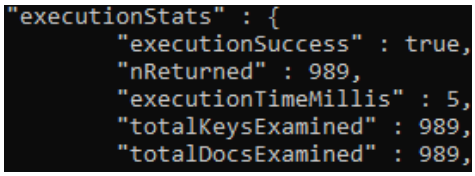
Let x be the number of registrations-per-day (W1), w.r.t this number W2 and W3 are very rare operations. Indeed, even though we can expect mischievous behaviours from some user, the number of country changes will never be comparable with x.

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 $frequency(R2) \ll x$), 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")
```



(a) Results without index



(b) Results with index

Considering again about 250k **Users**, without an index we need to scan the whole database, which means a medium-high execution time for each request.
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**.

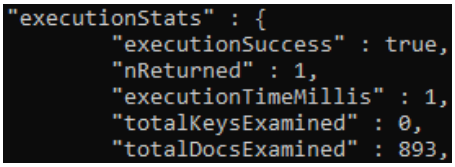
Pokemon Name Queries on **Pokémon**'s *name*:

Type	Query
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

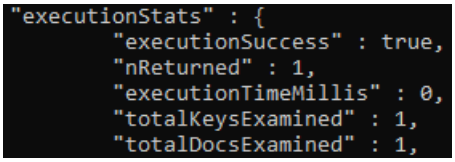
Again, W1 and W2 are rare and admin-related operations: this means that this queries will not require a frequent update of the index. On the

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:

```
> db.user.pokemon({name: "pikachu"}).explain("executionStats")
```



(a) Results without index



(b) Results with 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. All things considered, **an index on *pokemon name* has been introduced**.

3.3 Graph Database

3.3.1 Queries Handled

Users related

Application Queries	Graph Queries
Insert a new **User** into the system at registration time	Insert a new USER node into the graph
Create a *follow* relationship between the current **User** and the selected **User**	Add a new FOLLOW edge from the current **User** Node to the selected **User** Node

Retrieve *recommended* **Users** (1)	Match all the USER nodes at distance 2 from the given USER node U according to one of these patterns: (U)—FOLLOWS→(USER f)—FOLLOWS →(USER recomm.)
Retrieve *recommended* **Users** (2)	Match all the USER nodes at distance 2 from the given USER node U according to one of these patterns: (U)—LIKES→(POKEMON p)←LIKES—(USER recomm.)
Retrieve friend list of a **User** U	Match all the USER nodes linked to the related **User** Node of U by an outcoming FOLLOWS edge
Retrieve **User**'s favourite **Pokémon**	Match all the POKEMON nodes which are linked to the USER node U through a LIKES edge
Remove a **Pokémon** from the favourite ones	Delete a LIKES edge
Modify **User** settings (*country*)	Modify USER node by changing the country property
Remove a **User** (*admin* only)	Delete a USER node
Remove a *follow* relationship	Delete a FOLLOWS 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

Team related

Application Queries	**Graph Queries**

Retrieve **Team** composition based on **User**	Given USER u, retrieve all the POKEMON nodes connected to u through a HAS edge
Insert a **Pokémon** into a **Team**	Add a OWNS relationship between a USER node and a POKEMON node

Pokemon related

Application Queries	**Graph Queries**
Create a new **Pokémon** (*admin* only)	Insert a new POKEMON node into the graph
Update **Pokémon** catch rate	Modify USER node by changing the *catch rate* property
Remove a **Pokémon** (*admin* only)	Delete a POKEMON node
Analytics: ranking of most popular *Pokémon* in world/each country	Count n_i = number of HAS incoming edges of POKEMON node p_i, for each POKEMON node. Sort k highest $n_1 \dots n_k$ and return relative $p_1 \dots p_k$

Post/Answer related

Application Queries	**Graph Queries**
Create a new **Post**	Add a new POST node into the graph
Create a new **Answer**	Add a new POST node into the graph

Retrieve all the **Posts** related 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 POST node P through a TOPIC edge
Remove a **Post** (only *admin* and *post's creator*)	Delete a POST node

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
- **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 favourites 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 **Answers** to a **Post**
- **CREATED relationships** between a **User** and a **Post** to map the owner of each **Post**/**Answers**

3.3.3 Graph structure

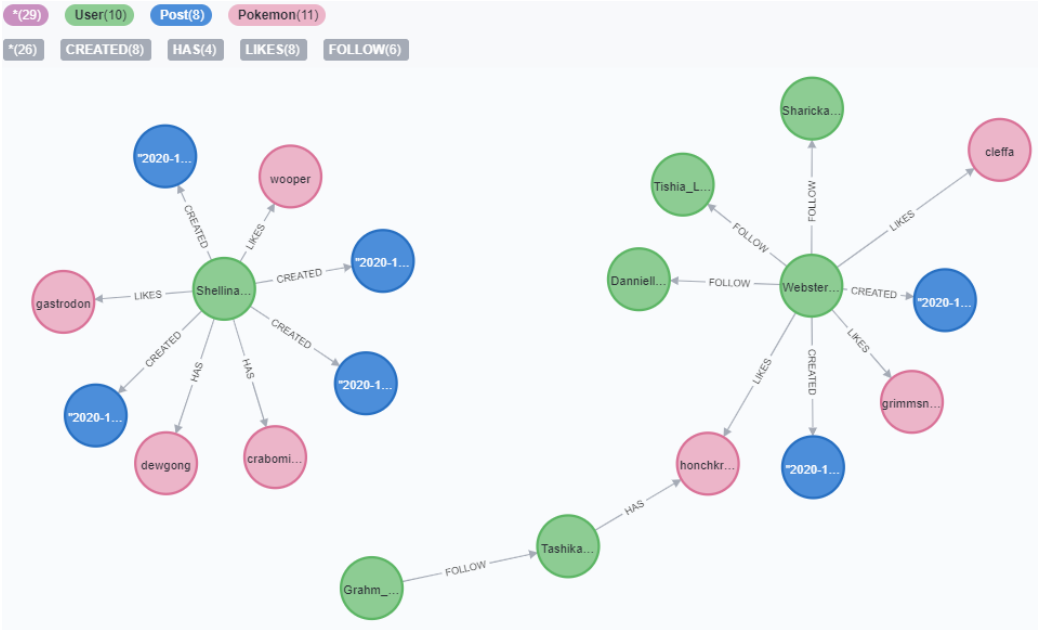


Figure 9: Graph Structure

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 properties 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 (relationship): no properties needed
- HAS (relationship): *slot* (integer)
- CREATED (relationship): no properties needed
- TOPIC (relationship): no properties needed

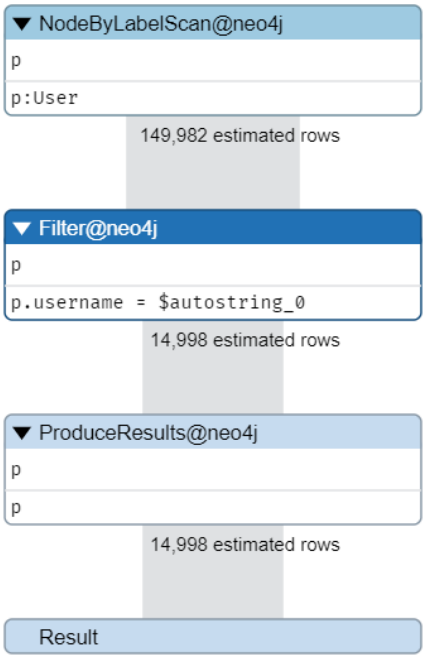
3.3.4 Indexes

Username As seen in the Indexes Paragraph for MongoDB, the queries that involve the *username* field are the following:

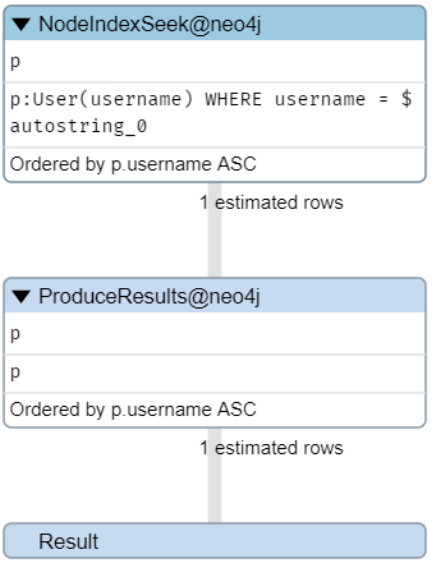
Type	Query
W1	Insertion of a new **User** in the GraphDB
W2	Deletion of an existing **User** in the GraphDB
R1	Search of a **User** u by *username* when an answer to a u's **Post** is written
R2	Search of a **User** by *username* when a new *follow* request is submitted

Assuming that the number of new registration is far more higher that the number of **Users** deletion, we can state that $|W1| \gg |W2|$. Furthermore, is likely that the numbers of login per day are far more than the new registrations at high User number loads, as stated in the Paragraph 2.6.
So, let x be the number of new logins per day, we can say that the number of each read operation will probably be a multiple of x, so, at the end, we can state that $|Ri| \gg |W1| \gg |W2|, i = 1, 2$. Hence, the application, at the GraphDB side, is **read intensive** and this statement leads to prove that an usage of an index for this field is good.
All things considered, from Neo4j Desktop we can compute the following command, which is a simple find by username, in order to get some performance statistics before and after the index addition:

```
neo4j$ explain match (p:User) where p.username = "
  Grahm_Gschwendtner1989" return p
```



(a) Results without index



(b) Results with index

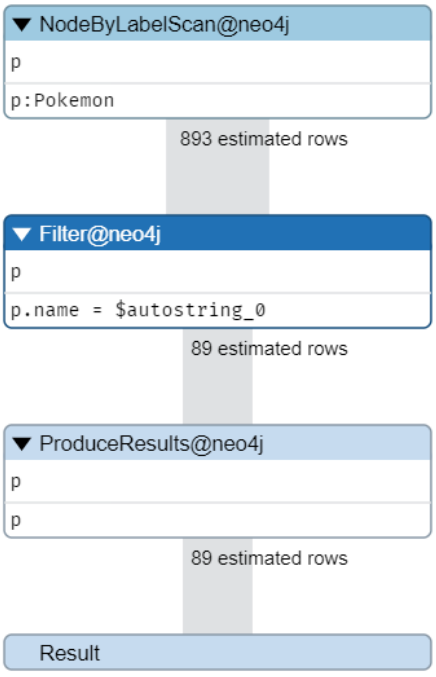
In this case we can notice a huge improvement in the number of rows handled with several order of magnitude and a slightly improvement on the timing. Even if the timing improvement is not huge we have to take into consideration the extreme simplicity of the "find by username" query, which does not show properly the benefits of handling only one row at the beginning of the query computation. All things considered, **an index on the field *username* has been created.**

Pokèmon Name The queries that involve the *name* field are the following

Type	Query
W1	Insertion of a new **Pokèmon** in the GraphDB
W2	Deletion of an existing **Pokèmon** in the GraphDB
R1	Search of a **Pokèmon** by name in order to catch it
R2	Search of a **Pokèmon** by name in order to create a post on it
R3	Search of a **Pokèmon** by name in order to save it as a favourite **Pokèmon**

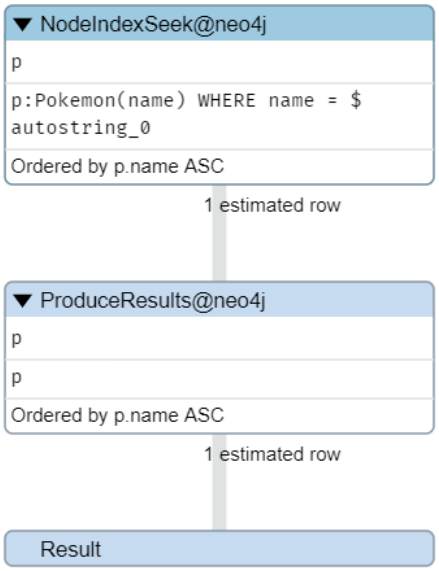
As said before, the number of addition or deletion of a **Pokèmon** are extremely rare due to the fact that only the admin could do that. So the writing operation of **Pokèmon** are done only by the admins where the read operation are done by the normal **Users**, which number is surely bigger. This consideration is enough to consider the operations on this field as **read intensive** and is enough to justify the thought of adding an index. In the following figure are presented the query and the relative performance results.

```
neo4j$ explain match (p:Pokemon) where p.name = "pikachu"
 return p
```



Completed after 68 ms.

(a) Results without index



Completed after 57 ms.

(b) Results with index

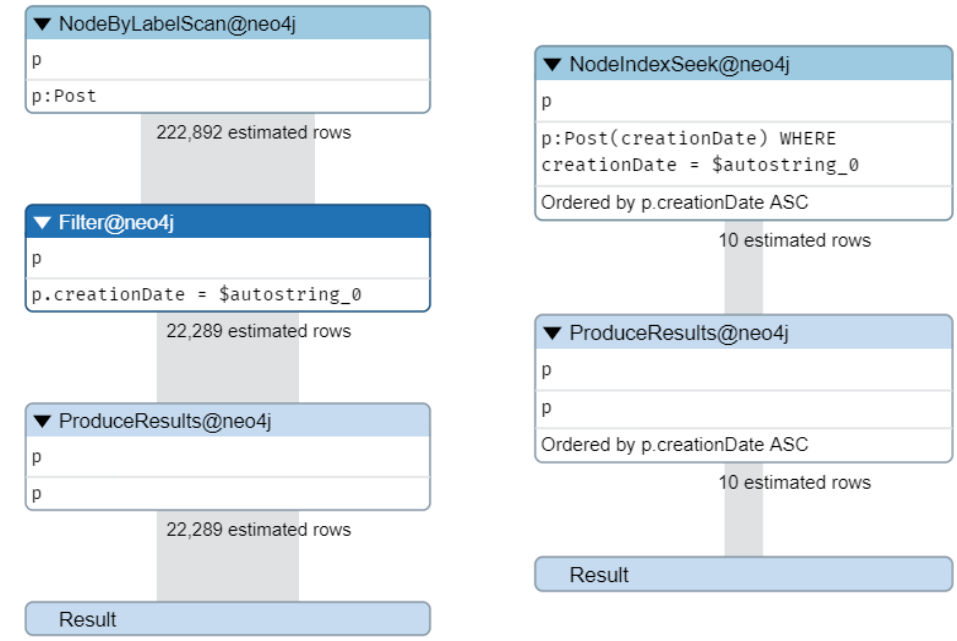
Even in this case we can see a slightly improvement on the timing performance and an improvement on the rows considered at the beginning of the query computation. The total number of Pokèmon is small with respect to other entities but surely the 3 grade of magnitude of difference for the estimated row should grant a big advantage with bigger and more complicated queries on Neo4j. All things considered, **an index on the field *pokèmon name* has been added**.

Post Creation Date The queries that involve the *creation date* field are the following

Type	Query
W1	Insertion of a new **Post** in the GraphDB
W2	Deletion of an existing **Post** in the GraphDB
R1	Search of a **Post**] in order to be addressed as a topic of an Answer
R2	Search of a **Post** in order to show every post related to a Pokemon

In this case we can say $|W1| \gg |W2|$ because only admins and the **User** who created the it can delete a **Post**. Then, in general, we can state even that if a **User** writes, say 1 **Post** per day, we can surely expect that he isn't the only one who has written a **Post**. So, in order to write a **Post** the **User** will view other **Posts** of a specific **Pokemon** (see UML Use Case Diagram for details). We can make the same reasoning even to the replies to a **Post**, because in order to make a reply the viewing action will come first. All things considered, we can assume that all the operation on the *creationDate* of a **Post** are **read intensive**. In the following figure are presented the query and the relative performance results.

```
neo4j$ explain match (p:Post) where p.creationDate = "
  2020-12-15T22:05:32.382000000" return p
```



(a) Results without index

(b) Results with index

We can make the same reasoning made for the *username* field. We can also see that the *creationDate* is not a UNIQUE among all **Posts** but we can think that filters better than the *content* field. Therefore, **this field has been chosen specifically for creating an index for the Post entity**.

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 choose 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 **Pokémons** 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 even through the addition of an index, as seen in the Paragraph (3.3.4)

- *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 Pokemon'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. For this reason, the Document Database hosts also an Analytic collection, structured as follows.

Figure 16: Analytic Collection

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 Database 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 $r_i | HB \rightarrow r_{i+1}, r_i$ reads from an updated server while r_{i+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 Client, Server and Daemon Thread

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.

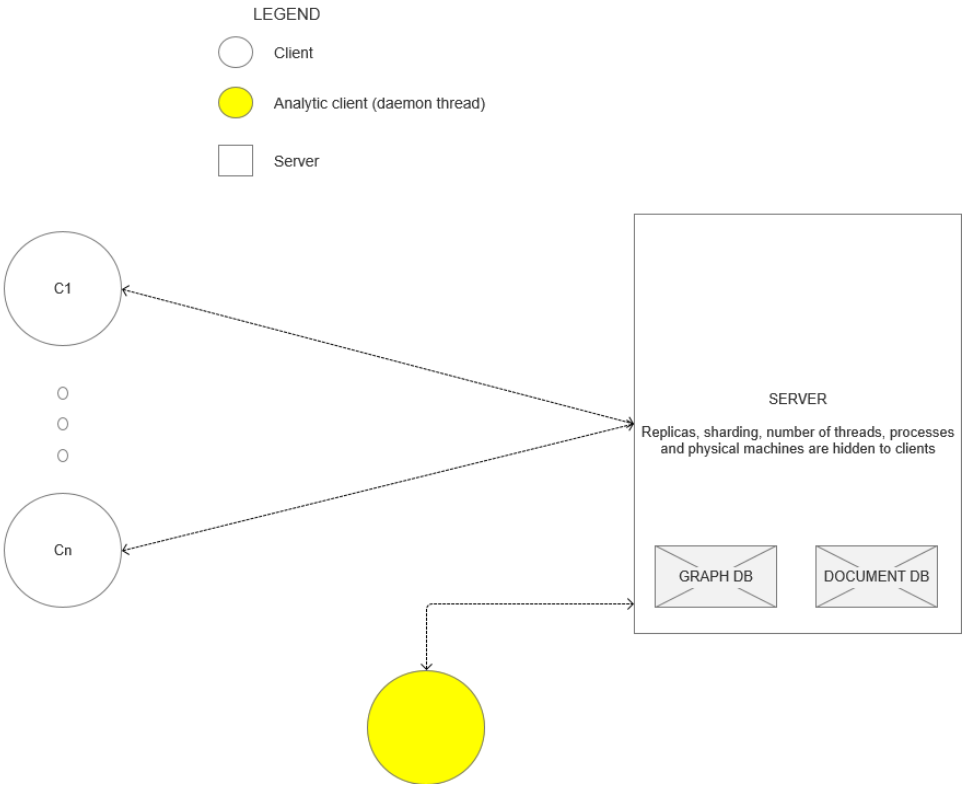


Figure 17: Service Architecture

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
- **Junit** for testing classes/methods and for fast bug discovering/recovering

4 — Implementation

4.1 Package structure

Package structure decision was as important task in ***PokèMongo***, 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:

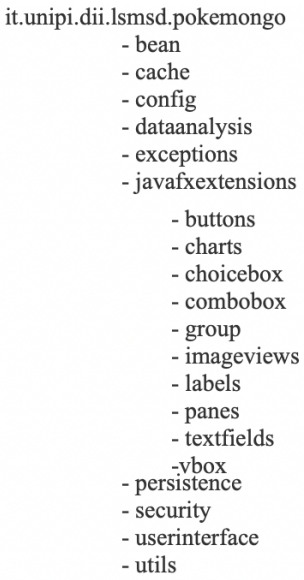


Figure 18: Package Structure

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: Bean

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

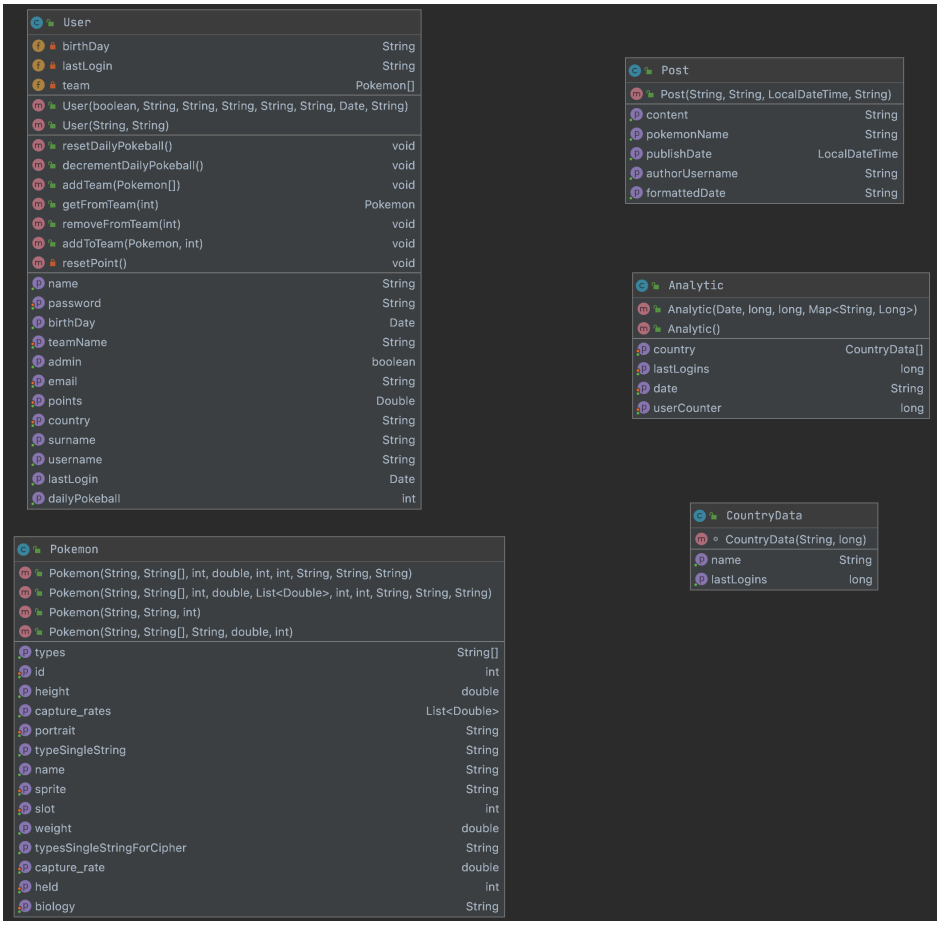


Figure 19: Bean Package Class Structure

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.

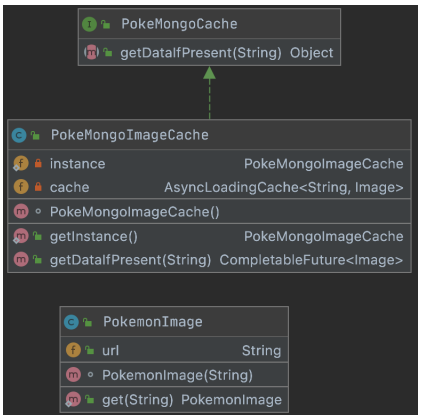


Figure 20: Cache Package Class Structure

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.

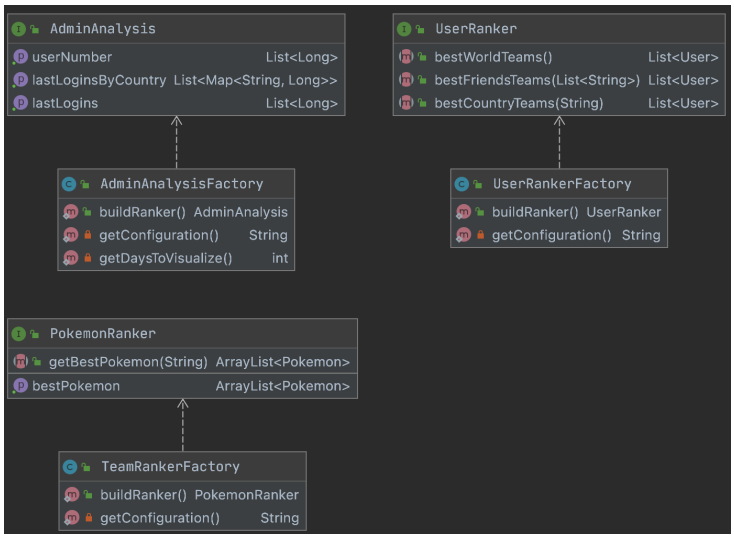


Figure 21: dataAnalysis Package Class Structure

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 **Pokèmon** 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.

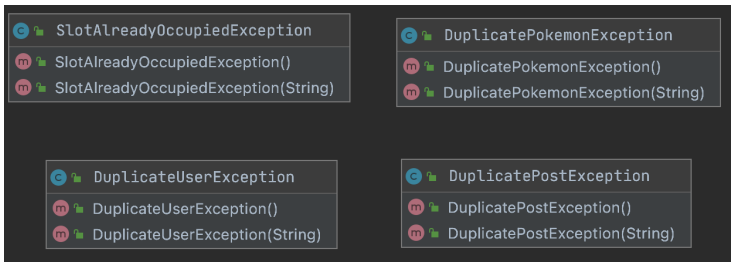


Figure 22: Exceptions Package Class Structure

Class Name	**Short Description**
SlotAlreadyOccupiedException	Exception thrown when a user try to catch a **Pokèmon** and he has the *slot* he want to use already occupied by one other **Pokèmon**
DuplicatePokemonException	Exception thrown when an admin try to insert a **Pokèmon** 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.

javafxextensions: buttons Here are present all the classes that extend *Button* from JavaFX

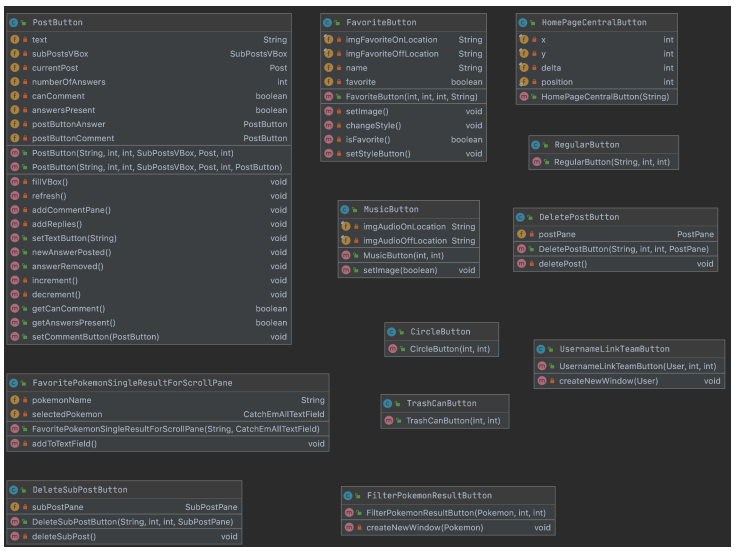


Figure 23: javafxExtension/buttons Package Class Structure

Class Name	**Short Description**
HomePageCentralButton	Specific button for the HomePage.
MusicButton	Button for turning the music on/off
RegularButton	For creating buttons like “BACK”, “SUBMIT”, etc...
TrashButton	Button for eliminating a **Pokèmon** 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 **Pokèmon**

DeletePostButton	Button for deleting a **SubPost** (aka response)
FilterPokemonResultButton	Specific button for displaying the name of a **Pokèmon** in a query result. At the click it creates a new Stage with the information about the **Pokèmon** (check PokemonWindowGroup).
FavoritePokemonSingleResultForScrollPane	This button is used for showing the name of the **Pokèmon** than are Favourite. Clicking on it will be a shortcut for capturing the **Pokèmon** 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).

javafxextensions: charts It contains a class that extends *LineChart* from JavaFX.

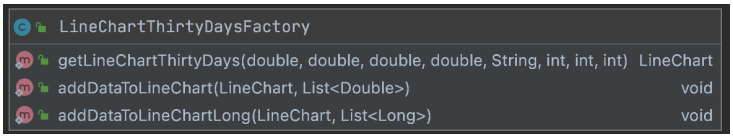


Figure 24: javafxExtensions/charts Package Class Structure

Class Name	**Short Description**
LineChartThirtyDaysFactory	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

javafxextensions: choicebox It contains a class that extends LineChart from JavaFX.

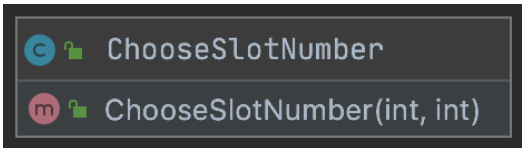


Figure 25: javafxExtensions/choicebox Package Class Structure

Class Name	**Short Description**
ChooseSlotNumber	Choice box that lets the user to select the slot for saving the **Pokèmon** in captured

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

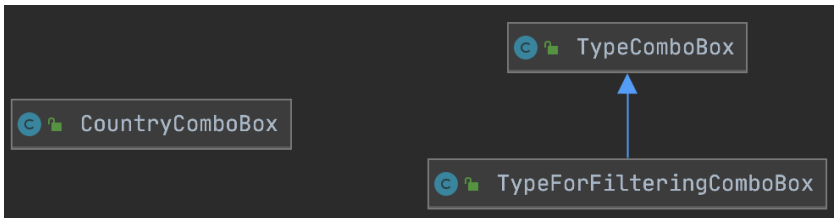


Figure 26: javafxExtensions/combobox Package Class Structure

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

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

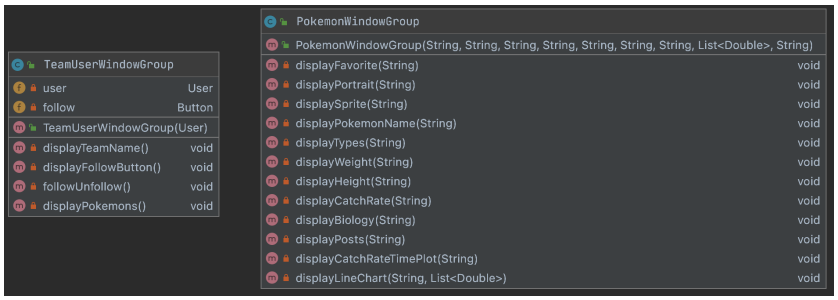


Figure 27: javafxExtensions/group Package Class Structure

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 **Pokèmon** along with the **Posts** related to it

javafxextensions: imageviews Extensions of ImageView

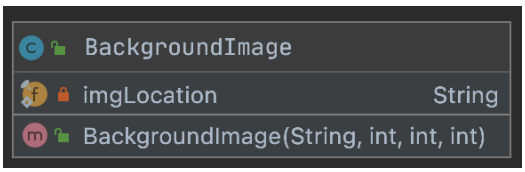


Figure 28: javafxExtensions/imageviews Package Class Structure

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

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

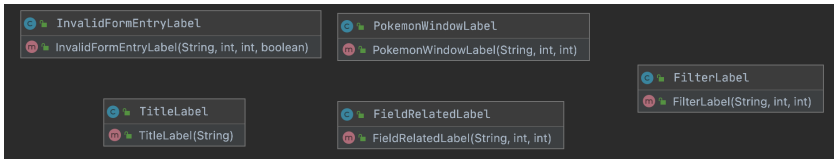


Figure 29: javafxExtensions/labels Package Class Structure

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 a specific **Pokèmon**
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

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).

Figure 30: javafxExtensions/panes Package Class Structure

Class Name	Short Description
RankingScrollPane	Pane that can be scrolled. It contains other panes that are specific for something (e.g. a **User**, a **Pokèmon**).

ToggleGroupPane	Specific pane for creating a toggle group
PokemonTeamForUserSelected Window	Specific pane for showing a single **Pokèmon** 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 **Pokèmon** set as favourite
FriendsSearchForUserPane	Specific pane for searching an user (Friends scene)
RankingPokemonSingleResult PaneSpecific	Specific pane to be inserted in a ScrollPane extension. It gives some information about the **Pokèmon** (used in the *Ranking*)
PokemonSingleResultPane	Specific pane to be inserted in a ScrollPane extension. It gives some information about the **Pokèmon** (used in the *Pokedex*)
RankingSingleUserResult	Specific pane to be inserted in a ScrollPane extension. It gives some information about the **Pokèmon** (used in the *Ranking*)

The addRemove package is characterized of these classes:

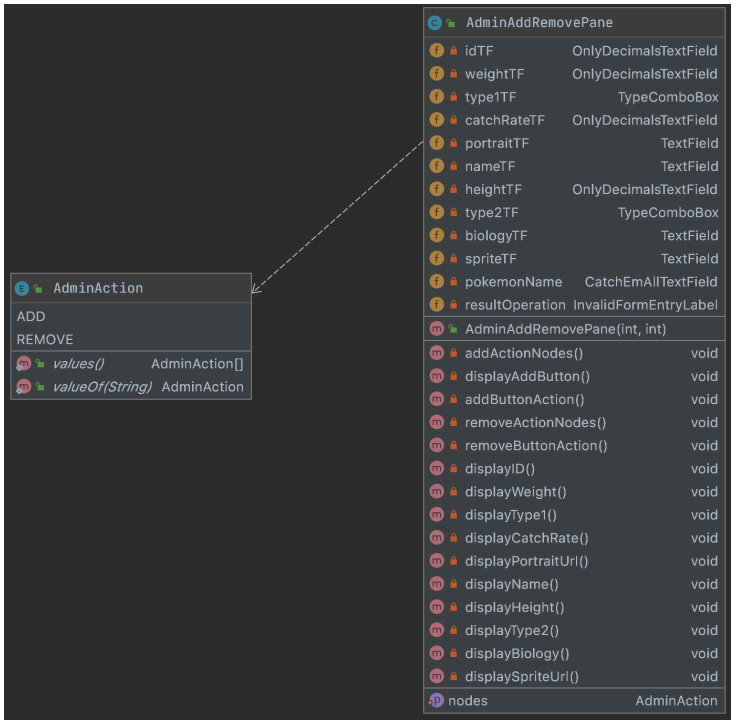


Figure 31: javafxExtensions/panes/addRemove Package Class Structure

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 **Pokèmon** 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.

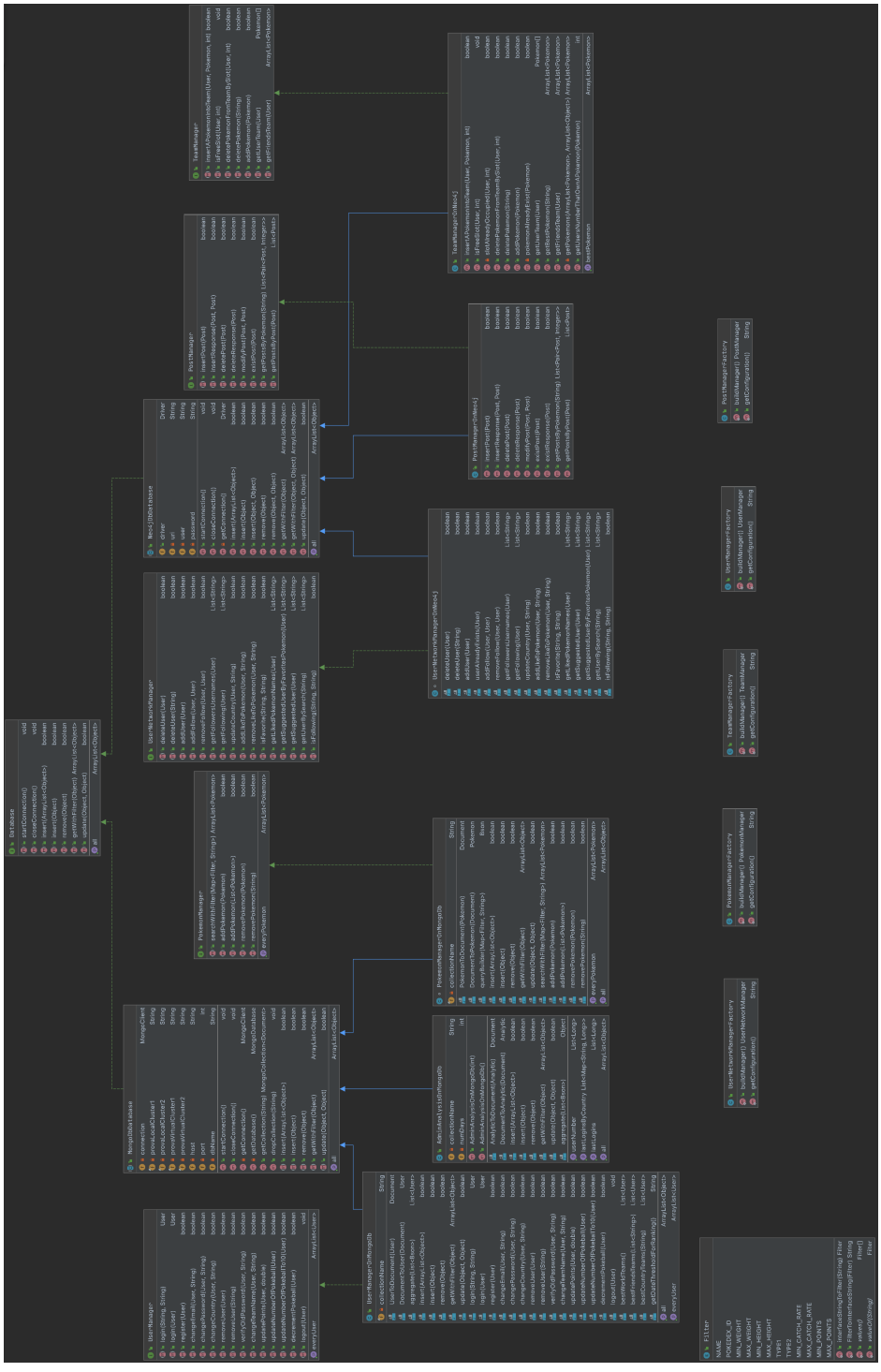


Figure 32: persistence Package Class Structure

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 **Pokèmon**, defines the fundamental operations
UserNetworkManager	Shared interface for the managing of **Pokèmon**, 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 **Pokèmon** 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.

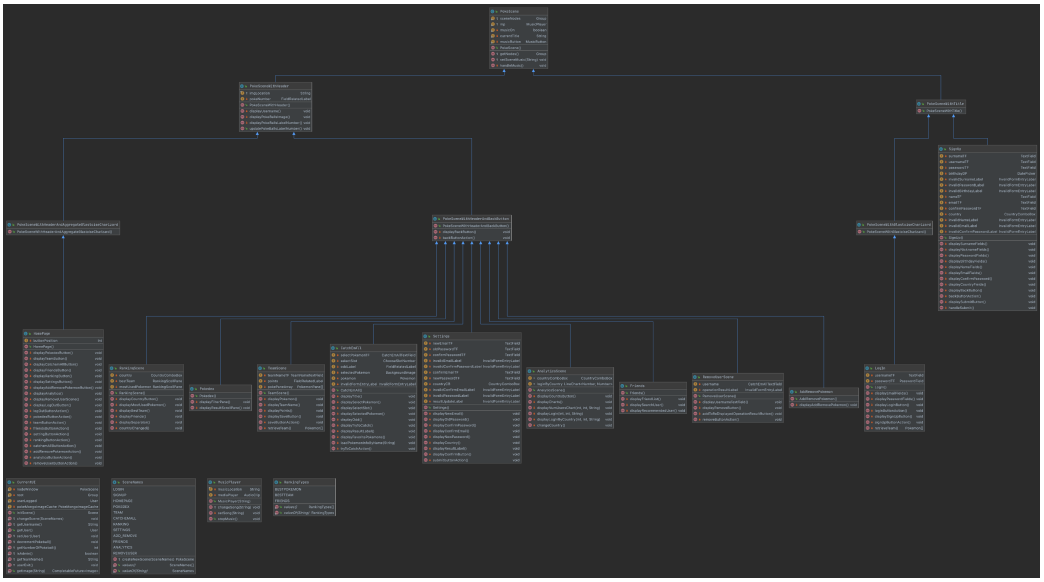


Figure 33: userInterface Package Class Structure

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 **Pokèmon**)
PokeSceneWithTitle	General scene with only the title
SignUp	Sign up page
PokeSceneWithBlastoise Charizard	General scene that extends PokeSceneWith-Title and adds to the scene the image of Charizard and Blastoise
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
PokeSceneWithHeaderAnd AggregateBlastoiseCharizard	General scene that combines the PokeSceneWithHeader and the PokeSceneWith-BlastoiseCharizard

HomePage	As the name suggests the class displays the Nodes regarding the HomePage
PokeSceneWithHeaderAndBack Button	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 *Friends* 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 **Pokèmon**
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.

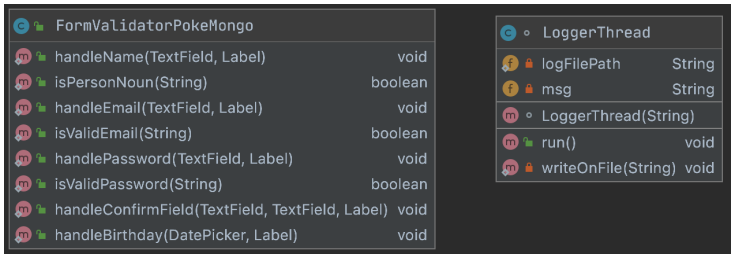


Figure 34: utils Package Class Structure

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 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.

4.2.1 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.

4.2.2 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 **Pokèmon** images also in our project.
The images about the **Pokèmons** 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.

```
1  public class PokeMongoImageCache implements PokeMongoCache
    {
2    //Singleton
3    private static PokeMongoImageCache instance;
4    private AsyncLoadingCache<String, Image> cache;
5
6    public static PokeMongoImageCache getInstance() {
7      if (instance == null) {
8        instance = new PokeMongoImageCache();
9      }
10     return instance;
11   }
12
13   PokeMongoImageCache(){
14     cache = Caffeine.newBuilder()
15     .expireAfterAccess(10, TimeUnit.MINUTES)
16     .maximumSize(1000)
17     .buildAsync(k -> PokemonImage.get(k));
18   }
19
20   public CompletableFuture<Image> getDataIfPresent(String
    url){
21     Logger.vlog("Attemp to get image at: " + url);
22     return cache.get(url);
23   }
24 }
```

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.

4.2.3 Password Encryptor

Encrypting a password is typically used to protect it from eavesdropping. We encrypt it and then we sent it to the database, in this way no eavesdropper can snatch the password in transit. How does the encryption is done? We decided to use the class org.apache.commons.codec.digest.DigestUtils, that has operations to simplify common MessageDigest tasks. Thus, we use DigestUtils.sha256Hex() calculate the SHA-256 digest of a string composed by the password of the user plus one other string.

```
1  public class PasswordEncryptor {
2
3    @VisibleForTesting
4    public static String encryptPassword(String plainPassword
     ){
5      String s = "randomSalt";
6      String encryptedPassword = cipher(plainPassword, s);
7      return encryptedPassword;
8    }
9
10   public static String cipher(String pwd, String salt) {
11     return DigestUtils.sha256Hex(pwd+salt);
12   }
13 }
```

4.2.4 Logger

We thought that useful logs would provide us (especially when someone has to debug/maintain someone else's code) with help when trying to understand what the code actually does. The Logger takes care of recording the events that happen during runtime.

```
1  public class Logger {
2
3    public static void warning(String text){
4      if(ConfigDataHandler.getInstance().configData.
     verbosityLevel >= 1){
5        LoggerThread lt = new LoggerThread("[WARNING] " +
     text);
6        lt.start();
```

```
7       }
8
9     }
10
11    public static void error(String text){
12      if(ConfigDataHandler.getInstance().configData.
    verbosityLevel >= 1){
13        LoggerThread lt = new LoggerThread("[ERROR] " + text)
    ;
14        lt.start();
15      }
16    }
17
18    public static void log(String text){
19      if(ConfigDataHandler.getInstance().configData.
    verbosityLevel >= 1){
20        LoggerThread lt = new LoggerThread("[LOG] " + text);
21        lt.start();
22      }
23    }
24
25    public static void vlog(String text){
26      if(ConfigDataHandler.getInstance().configData.
    verbosityLevel >= 2){
27        LoggerThread lt = new LoggerThread("[VLOG] " + text);
28        lt.start();
29      }
30    }
31
32    public static void vvlog(String text){
33      if(ConfigDataHandler.getInstance().configData.
    verbosityLevel >= 3){
34        LoggerThread lt = new LoggerThread("[VVLOG] " + text)
    ;
35        lt.start();
36      }
37    }
38  }
```

Every function create and start a new LoggerThread, it will simply print in a file the log infos.

```
1   class LoggerThread extends Thread{
2     private static String logFilePath="log/logFile.txt";
3     private String msg;
4
5     LoggerThread(String msg){
6       this.msg=msg;
7     }
8
```

```
9     public void run(){
10      String newMsg = Instant.now().toString() + " " + msg +
    "\n";
11      writeOnFile(newMsg);
12    }
13
14    private static synchronized void writeOnFile(String msg){
15      try {
16        Files.write(Paths.get(logFilePath), msg.getBytes(),
    StandardOpenOption.APPEND);
17      }
18      catch (IOException i){
19        i.printStackTrace();
20      }
21    }
22
23  }
```

4.2.5 Form Validator

The FormValidatorpokMongo checks if the **User** fills the textfields in the proper way, if he doesn't that it shows up an error as a label. The validator functions makes use of Regular Expression, in this way me ensure that the field is well written.

```
1   public class FormValidatorPokeMongo {
2
3     /**
4     * In this section are present the event handler for the '
    setOnKeyReleased' event in the form.
5     */
6     public static void handleName(TextField nameTF, Label
    invalidNameLabel){
7       if(FormValidatorPokeMongo.isPersonNoun(nameTF.getText()
    ))
8       invalidNameLabel.setVisible(false);
9       else
10      invalidNameLabel.setVisible(true);
11    }
12
13    /**
14    * Check if the string contains only letters, spaces, dots
    and apostrophes.
15    */
16    @VisibleForTesting
17    public static boolean isPersonNoun(String possibleNoun){
18      Pattern pattern = Pattern.compile("^[a-zA-Z '.]+$");
19      Matcher matcher = pattern.matcher(possibleNoun);
```

```
20      return matcher.find();
21    }
22
23    public static void handleEmail(TextField emailTF, Label
    invalidEmailLabel){
24      if(FormValidatorPokeMongo.isValidEmail(emailTF.getText
    ()))
25      invalidEmailLabel.setVisible(false);
26      else
27      invalidEmailLabel.setVisible(true);
28    }
29
30    /**
31    * Check if the email follows the format example@domain.
    tld
32    */
33    @VisibleForTesting
34    public static boolean isValidEmail(String possibleEmail){
35      Pattern pattern = Pattern.compile("^[\\w-\\.]+@([\\w
    -]+\\.)+[\\w-]{2,4}$");
36      Matcher matcher = pattern.matcher(possibleEmail);
37      return matcher.find();
38    }
39
40    public static void handlePassword(TextField passwordTF,
    Label invalidPasswordLabel){
41      if(FormValidatorPokeMongo.isValidPassword(passwordTF.
    getText()))
42      invalidPasswordLabel.setVisible(false);
43      else
44      invalidPasswordLabel.setVisible(true);
45    }
46
47    /**
48    * Checks if the password contains minimum eight
    characters, at least one letter and one number.
49    */
50    @VisibleForTesting
51    public static boolean isValidPassword(String
    possiblePassword){
52      Pattern pattern = Pattern.compile("^(?=.*[A-Za-z])
    (?=.*\\d)[A-Za-z\\d]{8,}$");
53      Matcher matcher = pattern.matcher(possiblePassword);
54      return matcher.find();
55    }
56
57    public static void handleConfirmField(TextField fieldTF,
    TextField confirmFieldTF, Label invalidConfirmFieldLabel){
```

```
58        String password = fieldTF.getText(), confirmPassword =
      confirmFieldTF.getText();
59
60        if(password.equals(confirmPassword))
61        invalidConfirmFieldLabel.setVisible(false);
62        else
63        invalidConfirmFieldLabel.setVisible(true);
64      }
65
66      /**
67      * Checks if the birthday date selected is valid: future
      dates cannot be picked
68      */
69      public static void handleBirthday(DatePicker birthdayDP,
      Label invalidBirthdayLabel){
70        LocalDate localDate = birthdayDP.getValue();
71        LocalDate today = LocalDate.now();
72        System.out.println(today);
73
74        if(localDate.isAfter(today)){
75          invalidBirthdayLabel.setVisible(true);
76        } else {
77          invalidBirthdayLabel.setVisible(false);
78        }
79      }
80    }
```

4.3 Business Logic

The business logic used in ***PokeMongo*** is the following. To ensure a high variability between **Teams** we decided to cope the *catch rate* of each **Pokemon** in a way that if it is held by a lot of **Users** it will decrease, thus will let **Users** to try to catch other **Pokemon**, with the same rarity but held by less people. Although, the *catch rate* is related also to the *points*, if a **Pokemon** has a lower *catch rate* the *points* are higher, so catching other **Pokemon** (not the famous ones) may look like illogical at first, but keep in mind that the probability that having a rare **Pokemon** is very low as the "CATCH'EM ALL" page, so it is quite impossible that every user has the same rare **Pokemons**.

4.3.1 Points computing

The *points* related to a **Pokemon** are computed in a very simple way and they are strictly linked to the catch rate of a **Pokèmon** (a value between 3

and 255).

$$points_{single\ pokemon} = 255 - dynamic\ catch\ rate \tag{1}$$

The total *points* held by a **User** is computed as follow:

$$Total\ points = \left(\sum_{Team} points_{single\ pokemon} \right) \times multiplier \tag{2}$$

The multiplier is a value of 1.5 that is applied when in the **Team are presented all Pokemon that has different types** from each other, otherwise the multiplier will be 1. The total points is computed only when a **User** login or when he catches/remove a **Pokèmon**.

4.3.2 Dynamic Catch Rate computing

The **dynamic catch rate** is expatiate as follows:

$$dyn.\ catch\ rate = catch\ rate \times \left(1 - \frac{\#users\ that\ has\ the\ Pokemon}{total\ number\ of\ users} \right) \tag{3}$$

4.4 Analytics queries

In this chapter are present how the analytic are really compute in our system.

4.4.1 User Rankings

We have three different types of ranking that regards the **Users**: the *World Best Team*, the *Friend Best Team* and the *"Country" Ranking* (where *country* is a specific country, e.g. Italy).

World Best Team The *World Best Team* consider all the **User** in the world and retrieve a limited number of **User** who has the highest value of *points*. This is done querying MongoDb as follow (function present in UserManagerOnMongoDB):

```
1  public List<User> bestWorldTeams() {
2    Bson match = match(and(eq("admin", false), lte("lastLogin
   ", getDateThresholdForRanking())));
3    Bson sort = sort(descending("points", "birthDay"));
4    Bson limit = limit(ConfigDataHandler.getInstance().
   configData.numRowsRanking);
5    Bson project = project(fields(excludeId(), include("
   username", "teamName", "points", "birthDay", "country")));
```

```
6       return aggregate(Arrays.asList(match, sort, limit,
        project));
7   }
```

What is done is matching **Users** that aren't admin and that have the *lastLogin* field within a certain range of days. Then we order the result in a descending way, considering the *points* and the birth day, this one just to order **Users** that have the same value for *points* in order to greeting younger players. As said before we limit the number of result (this can be modified in the configuration file) and we project only the feature we are interested into.
We have limited the ranking based on *lastLogin*, because the **User** *points* are only updated when the **User** use the application, in order to have a less computational effort in the server databases. This, also, makes the **User** *points* something that **lose meaning after a certain period of time**.

Friends Best Team This kind of ranking is made thanks to the combination of MongoDb and Neo4j, because friends are retrieve from Neo4j and all information about them are, then, retrieve from MongoDb. To do that in our code we use two function, one from UserNetworkManagerOnNeo4j

```
1   public List<String> getFollowersUsernames(User target){
2     List<String> followersUsernames = new ArrayList<String>()
      ;
3     String query = "MATCH (to:User)-[h:FOLLOW]->(from:User)
      WHERE from.username = $username RETURN to.username";
4     ArrayList<Object> res = getWithFilter(query, parameters("
      username", target.getUsername()));
5     for(Object o: res){
6       Record r =(Record)o;
7       String username = r.get("to.username").asString();
8       followersUsernames.add(username);
9     }
10    return followersUsernames;
11  }
```

And the other from UserManagerOnMongoDb

```
1   public List<User> bestFriendsTeams(List<String>
      friendsUsername) {
2     Logger.vlog("COMPUTING BEST FRIENDS TEAMS");
3     Bson sort = sort(descending("points", "birthDay"));
4     Bson limit = limit(ConfigDataHandler.getInstance().
      configData.numRowsRanking);
5     Bson match = match(and(eq("admin", false), in("username",
      friendsUsername), lte("lastLogin",
      getDateThresholdForRanking())));
```

```
6     Bson project = project(fields(excludeId(), include("
      username", "teamName", "points", "birthDay", "country")));
7     return aggregate(Arrays.asList(match, sort, limit,
      project));
8   }
```

Country Best Team This is similar to the *World Best Team*, but we use an additional match for the country.

```
1 public List<User> bestCountryTeams(String country) {
2   Bson match = match(and(eq("country", country), eq("admin",
      false), lte("lastLogin", getDateThresholdForRanking())));
3   Bson sort = sort(descending("points", "birthDay"));
4   Bson project = project(fields(excludeId(), include("
      username", "teamName", "points", "birthDay", "country")));
5   Bson limit = limit(ConfigDataHandler.getInstance().
      configData.numRowsRanking);
6   return aggregate(Arrays.asList(match, sort, limit, project)
      );
7 }
```

4.4.2 Pokemon Rankings

We only perform two types of ranking regarding the **Pokemons**, the ***World* Best Pokemon** and ***Country* Best Pokemon**.

World Best Pokemon Most used **Pokemons** in the whole world. We compute them using Neo4j (TeamManagerOnNeo4j):

```
1   public ArrayList<Pokemon> getBestPokemon() {
2     ArrayList<Pokemon> pokemonArrayList = new ArrayList<>();
3     String query = "MATCH ()-[h:HAS]->(p:Pokemon) return p.
      name, count(h) AS held, p.sprite ORDER BY held DESC LIMIT
      " +  + ConfigDataHandler.getInstance().configData.
      numRowsRanking;
4     ArrayList<Object> res = getWithFilter(query);
5     return getPokemons(pokemonArrayList, res);
6   }
```

Country Best Pokemon Most used **Pokemons** in a specific country. We compute them using Neo4j (TeamManagerOnNeo4j)

```
1 public ArrayList<Pokemon> getBestPokemon(String country) {
2   ArrayList<Pokemon> pokemonArrayList = new ArrayList<>();
```

```
3   String query = "MATCH (u:User)-[h:HAS]->(p:Pokemon) WHERE u
      .country = $country return p.name, count(h) AS held, p.
      sprite ORDER BY held DESC LIMIT " + ConfigDataHandler.
      getInstance().configData.numRowsRanking;
4   ArrayList<Object> res = getWithFilter(query, parameters("
      country", country));
5   return getPokemons(pokemonArrayList, res);
6 }
```

4.4.3 Usage Statistics

We created few function for handling some simple statistics that can be useful for the **admin User**, in order to see how many the people in the world use the application and how often. We decided to include three types of statistic: *total number of users*, *users that logged during a single day*, *users that logged during a single day in a specific country* (we limit the number of country to the top 15). Because all the information needed is stored in MongoDb we query the database in order to create a new document with all the information specific for the analytics of a single day.

Total Users :

```
1   public long getUserNumber() {
2     Bson match = match(ne("admin", true));
3     Bson count = group("$admin", sum("userNumber", 1));
4     Bson project = project(fields(include("userNumber")));
5     Document result = aggregate(Arrays.asList(match, count,
      project)).get(0);
6     return result.getInteger("userNumber").longValue();
7   }
```

Today Login :

```
1   public long getTodayLogin(){
2     Calendar lastDay = Calendar.getInstance();
3     lastDay.setTime(new Date());
4     lastDay.add(Calendar.DATE, -1);
5     String yesterday = new SimpleDateFormat ("yyyy-MM-dd'T'HH
      :mm:ss.SSS'Z'", Locale.US).format(lastDay.getTime());
6     Bson match = match(and(gte("lastLogin", yesterday), ne("
      admin", true)));
7     Bson count = group("$admin", sum("loginNumber", 1));
8     Bson project = project(fields(include("loginNumber")));
9     Document result = aggregate(Arrays.asList(match, count,
      project)).get(0);
10    return result.getInteger("loginNumber").longValue();
```

```
}
```

Country Today Login :

```
public Map<String, Long> getLastLoginsByCountry() {
  Calendar lastDay = Calendar.getInstance();
  lastDay.setTime(new Date());
  lastDay.add(Calendar.DATE, -1);
  String yesterday = new SimpleDateFormat ("yyyy-MM-dd'T'HH
  :mm:ss.SSS'Z'", Locale.US).format(lastDay.getTime());
  Bson match = match(and(gte("lastLogin", yesterday), ne("
  admin", true)));
  Bson count = group("$country", sum("lastLogin", 1));
  Bson sort = sort(descending("lastLogin"));
  Bson limit = limit(15);
  Bson project = project(fields( include( "_id", "lastLogin
  ")));
  List<Document> result = aggregate(Arrays.asList(match,
  count, sort, limit, project));
  Map<String, Long> map = new HashMap<>();
  for(Document d: result)
  map.put(d.getString("_id"), d.getInteger("lastLogin").
  longValue());
  return map;
}
```

4.4.4 Dynamic Catch Rate

The *dynamic catch rate* is saved in the **Pokèmon** json as an array of 30 values, in which the first one indicates the current catch rate of the **Pokèmon** (the other 29 values are using for plotting the catch rate chart). Let's analyze it step by step. The first thing we have to do is to retrieve from the databases all the **Pokèmon**.

```
PokemonManager pokemonManager = PokemonManagerFactory.
 buildManager();
TeamManager teamManager = TeamManagerFactory.buildManager()
 ;

ArrayList<Pokemon> pokemons = pokemonManager.
 getEveryPokemon();
```

Then for each of them we retrieve how many trainers has it.

```
List<Pair<String, Integer>> trainersPerPokemon =
 teamManager.getUsersNumberThatOwnsAPokemonNotFiltered();
```

Next step is to compute the catch rate as describe in chapter 4.3.2 for every **Pokèmon**.

```
int index = 0;
int numTrainers;
double new_catch_rate;
Pokemon oldPokemon;
List<Double> capture_rates;
for(Pokemon p: pokemons){
  oldPokemon = new Pokemon(p.getName(), p.getTypes(), p.getId
   (), p.getCapture_rate(), p.getCapture_rates(), (int)p.
   getHeight(), (int)p.getWeight(), p.getBiology(), p.
   getPortrait(), p.getSprite());
  Pair<String, Integer> currentTrainers = trainersPerPokemon.
   get(index);
  if(trainersPerPokemon.get(index).getKey().equals(p.getName
   ())){
    numTrainers = currentTrainers.getValue();
    index++;
  } else {
    numTrainers = 0;
  }

  new_catch_rate = p.getCapture_rate()*(1 - (numTrainers*1.0)
   /(userNumber));
```

Then we want to ensure that in the array are only present 29 values, not one more, because we want at the end of the operation to have precisely 30 entries.

```
  capture_rates = p.getCapture_rates();

  if(capture_rates.size() >= 30){
    while(capture_rates.size() < 30)
    capture_rates.remove(0);
  }
```

Then we add the new value computed to a list that will be used to update the *catch rate* of every single **Pokèmon** using the operation called at last.

```
  capture_rates.add(new_catch_rate);
  long count = pokemonManager.updatePokemon(oldPokemon, p);
  catchRatesOfPokemons.add(new PokemonAndCatchRate(p.getName
   (), new_catch_rate));
}
teamManager.updateCatchRateOfPokemon(catchRatesOfPokemons);
```

5 — Test

5.1 Privacy and Security

5.2 Unit Test

5.3 Robustness

5.4 Performance