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 Pokémon 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 favorites
    * 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
    * 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 the 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 Pokemon catch rates based on the number of users that own that pokemon
  - Update team points if the user has 6 Pokémon of different types
  - Periodically compute usage statistics to be consulted by the administrators

2.1.2 UML Use Cases Diagram

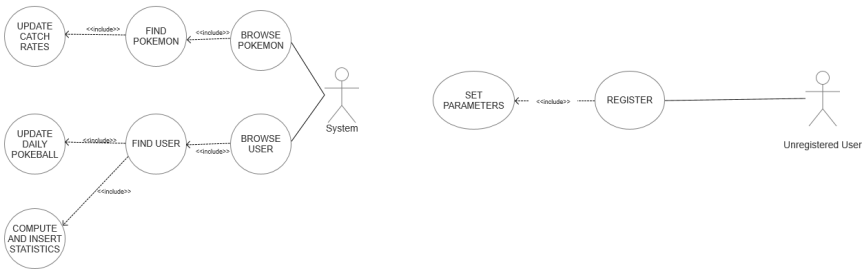


Figure 1: Use Case Diagram 1

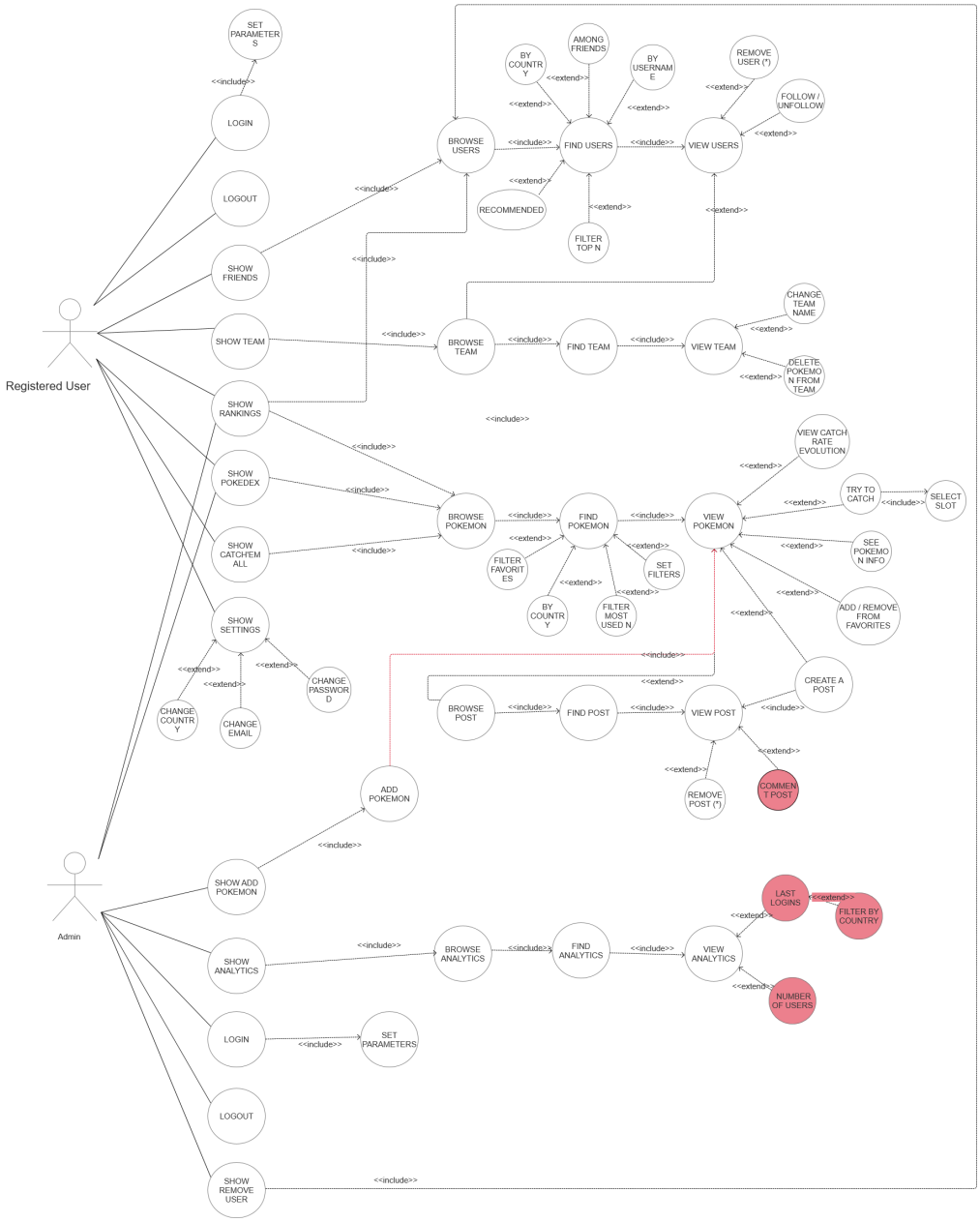


Figure 2: Use Case Diagram 2
(*) only for the User who created the Post and Admins, in Red Browse-find-view comments and browse-find-view answers had not been reported)

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, comments and answers must follow a **causal-consistency**
- **Response time** is an important issue: redundancies and larger memory consumptions are preferred over high latencies
- **Passwords are crypted** for security reasons
- A graphical interface and the usage of multimedia are crucial for an involving game experience

2.3 Sources, Velocity properties and Volume of data

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

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

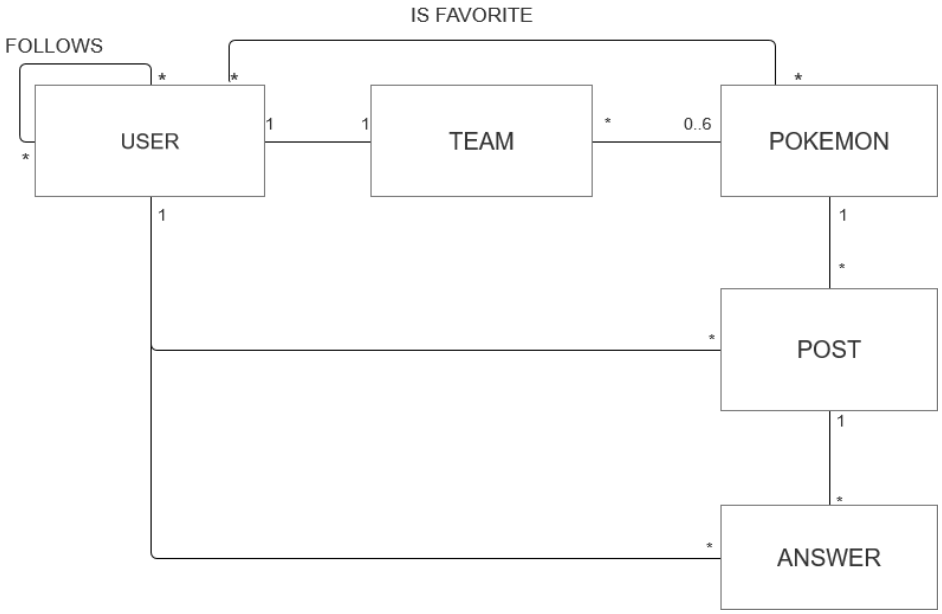


Figure 3: UML Entity Diagram

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 **Pokémon**, 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 favorites **Pokémon**. A **Pokémon** can be favorite 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 favorites
- Retrieve **User** information at login time
- Retrieve a **User** by username when looking for a new friend
- Retrieve **Team** information based on user
- Retrieve **Pokémon** information using several filters
- Retrieve recommended **Users**
- Retrieve list of a **User**'s friends
- Retrieve a **Pokémon** by name when trying to catch it
- Retrieve all the **Posts** relative to a Pokémon
- Retrieve all the **Answers** to a **Post**

- Retrieve **User**'s favorite **Pokémon**
- Modify **User** settings (email, password, country)
- Update **Team**'s name
- Update **Team**'s points
- Update **Pokémon**'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 favorite ones
- Analytics: ranking of most popular **Pokémon** in world/each country
- Analytics: ranking of best **Teams** in the world/each country/among friends
- Analytics: evolution on time of a **Pokémon** catch rate
- Analytics: evolution on time of number of logins per day/total **Users**/logins per day by country (admin only)

2.6 Feasibility Study and Load Estimation

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

- Since the globality of the app and the Social Network structure, we can estimate 5-10M of registered users. This means about 1M of logins-per-day.
- Registered Pokémon are 893. Even though there is the possibility for an admin to add new Pokémon, we think that they will be no more than 1K at every time.

- Expert users will probably generate a higher amount of posts/comments rather than new users. On average, there will be about 4-5M of posts/comments per day.

- Beginners are likely to generate an higher load of follow/unfollow requests respect to expert users. On average, it's reasonable to count about 5 follow/unfollow requests per login.

- Pokéballs and Pokémon capture is the catchiest feature of the game. Very likely almost the totality of the users that logs into the app will spend all his/her available daily Pokéballs. Anyway it's also probable that the most intriguing Pokémon will be the ones with low catch rate. Since there are 10 Pokéballs available each day, but the weighted average probability of catching a Pokémon can be estimated as near 10%, there will be about 1M of team updates per day

- As said in the previous point, we can count about 10 catch tries per day. It's likely that the chosen Pokémon was taken from the provided favorite shortcut. Moreover, likes are integrating part of this Social Network, not only a practical tool for catching Pokémon. So we can say that there will be about 2M of likes per day.

- We can estimate that on the average a user will consult ranking twice per day. Indeed immediately after log in and at the catching of a new Pokémon are possible occasions in which the user could be interested in seeing his/her progresses. For this reason we can consider 2M of ranking consulting per day

- Very few users will change his/her settings or password, since they are long term fields: this kind of updates will be no more than 30-40K per day.

3 — Project

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

- 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

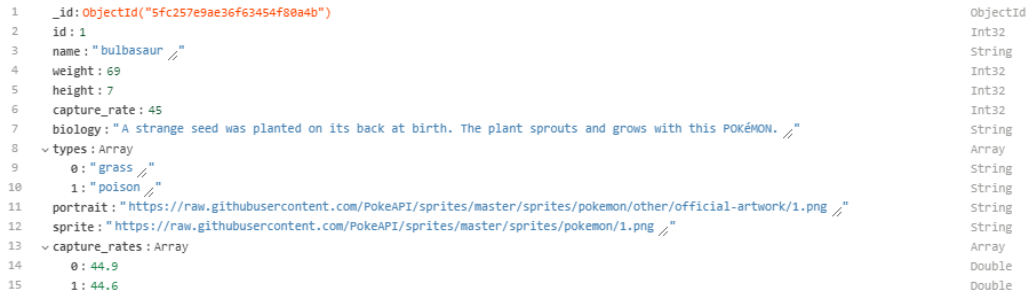


Figure 5: Pokèmon Collection

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:

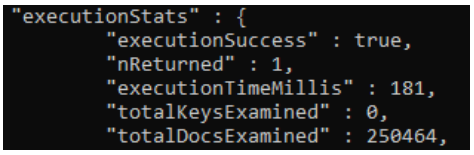
Type	Query
W1	Insert a new username at registration time of an arbitrary user
W2	Remove a username when an admin delete's a user from the system
R1	Check uniqueness of a username at registration time
R2	Check user's credential at login time
R3	Find a user by username when a new follow request is submitted

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

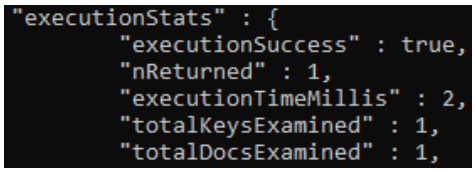
popular query among the network structure of the application, say 30 times the number of registered users: we find out that read operations on this field are about 130 times the number of write operations. Now consider MongoDb performances with and without using an index on the username field, in a Database 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 examinate one document. Moreover the unique property permits to eliminate the need of submitting query R1 at each registration. Considering the very high speed-up ratio of the indexing and the high frequency of this kind of queries w.r.t. the write operations (as explained before), a UNIQUE INDEX on username has been created.

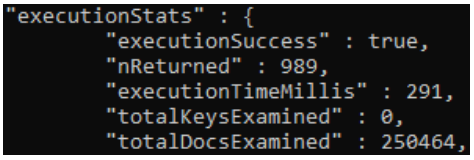
Country As seen before, starting from the application queries we demonstrate the benefits of an index in the field *country*.

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

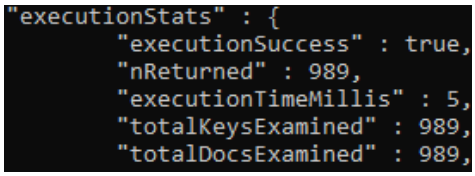
Let x be the number of registrations-per-day (W1), w.r.t this number W2 and W3 are very rare operations. Indeed, even though we can expect mischievous behaviors from some user, the number of country changes will never be comparable with x.

On the other hand, in order to guarantee a read-your-own-write eventual consistency on ranking R1, this query is recomputed every time a user asks to see the ranking itself. Thus, since the gameplay is highly based on rankings, we can estimate that R1 frequency will be about $400x$.
Furthermore we have to consider R2. Despite the fact that this query is executed just once per day (so $frequency(R2) \ll x$), 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.

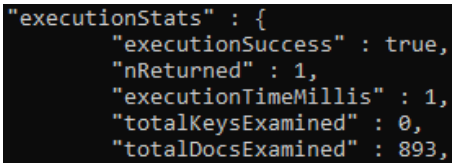
Pokemon Name Queries on Pokémon's name:

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

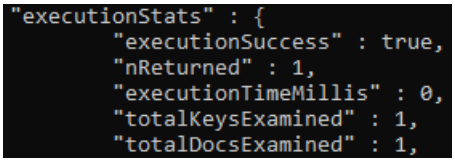
Again, W1 and W2 are rare and admin-related operations: this means that this queries will not require a frequent update of the index. On the

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.

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	Match all the USER nodes at distance 2 from the given USER node U according to one of these patterns(the first one has precedence) (U)—FOLLOWS→(USER f)—FOLLOWS →(USER recomm.) (U)—LIKES→(POKEMON p)←LIKES—(USER recomm.)
Retrieve 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 favorite Pokémon	Match all the POKEMON nodes which are linked to the USER node U through a LIKES edge
Remove a Pokémon from the favorite 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 owner)	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 comments to a **Post**
- **CREATED relationships** between a **User** and a **Post** to map the owner of each post/comment

3.3.3 Graph structure

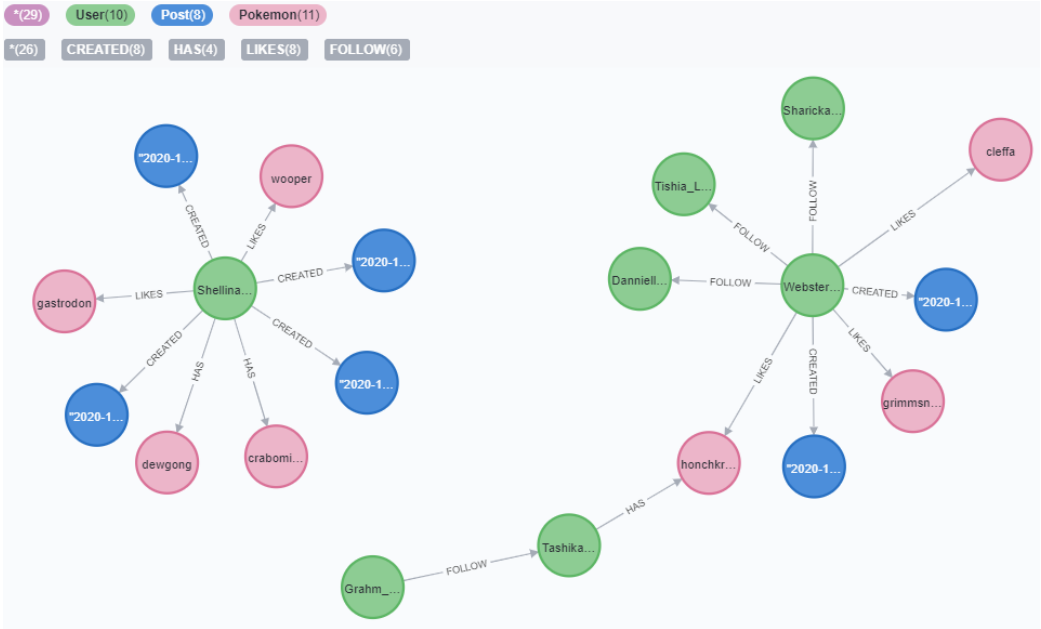


Figure 9: Graph Structure

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 property stored are the following:

- USER (node): *country*, *username*
- POKEMON (node): *name*, *capture rate*, *sprite*, *type*

- POST (node): *content*, *creation date*
- FOLLOW (relationship): no properties needed
- LIKES (relationship): no properties needed
- HAS (relationship): slot
- CREATED (relationship): no properties needed
- TOPIC (relationship): no properties needed

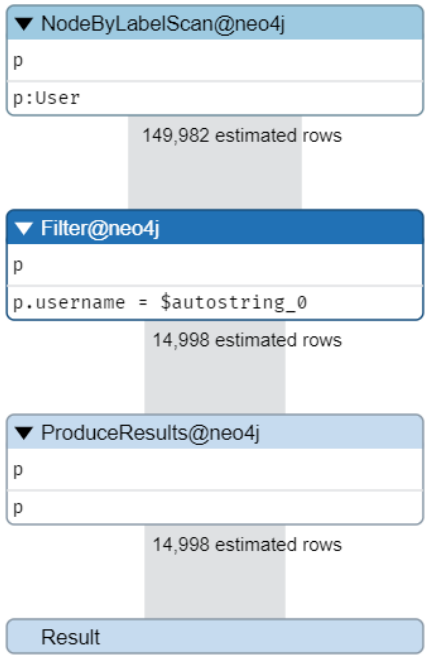
3.3.4 Indexes

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

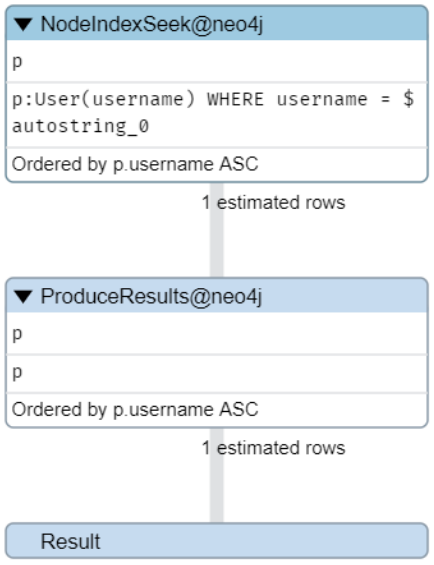
Type	Query
W1	Insertion of a new **User** in the GraphDB
W2	Deletion of an existing **User** in the GraphDB
R1	Search of a **User** u by username when an answer to a u's Post is written
R2	Search of a user by username when a new follow request is submitted

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

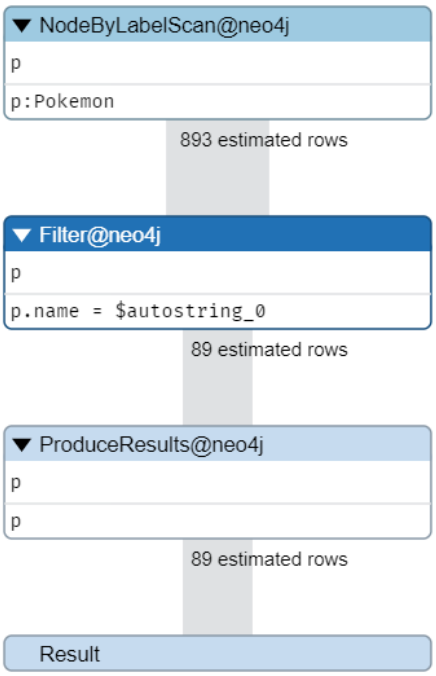
In this case we can notice a huge improvement in the number of rows handled with several order of magnitude and a slightly improvement on the timing. Even if the timing improvement is not incredible we have to take into consideration the extreme simplicity of the "find by username" query, which does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

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

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

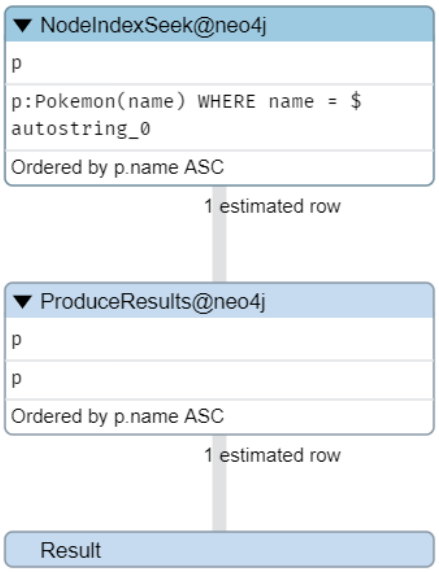
As said before, the number of addition or deletion of a Pokemon are extremely rare due to the fact that only the admin could do that. So the writing operation of pokemon are done only by the admins where the read operation are done by the normal users, which number is surely bigger. This consideration is enough to consider the operations on this field as **read intensive** and is enough to justify the presence of 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

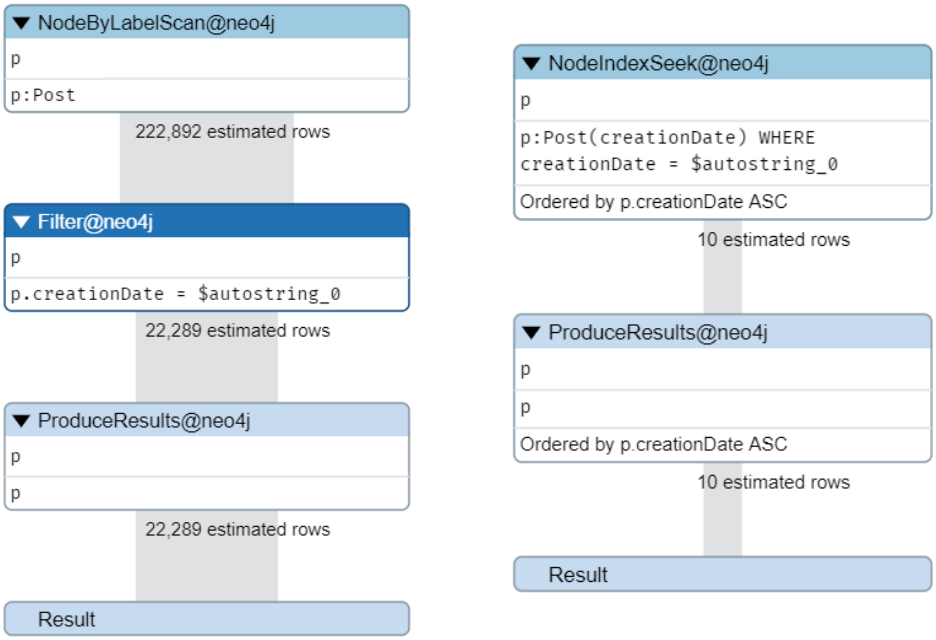
Even in this case we can see a slightly improvement on the timing performance and an improvement on the rows considered at the beginning of the query computation. The total number of pokemon is small with respect to other entities but surely the 3 grade of magnitude of difference for the estimated row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

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 down, say 1 post per day, we can surely expect that he isn't the only one who has wrote 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 Pokémon composing the **Team** and their catch rate.

Since every **User** can have only a **Team**, the *Team Name* property can be directly be stored into the Document Db's user collection. The amount of *points* is not recomputed each time the **Team** is retrieved but it stored as a redundancy in the user collection until it is changed. The collection of **Pokémon** is maintained as up to 6 edges between a **User** node and **Pokémon** nodes in the Graph Database. This choice is due the fact that:

- An array of **Pokémon** in the user collection was not so good for ranking most used **Pokémon**

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

3.4.2 User's Redundancies

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

- *Username*: Despite the fact that DBMSs always provide an identification mechanism not related to the one made by the programmer, we chose to repeat the username to quickly retrieve friends' and post/comment owners' name. This additional field is not so memory-intensive but can speed-up very much these queries 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

3.5.2 Replicas

3.5.3 Eventual Consistency

3.5.4 Sharding

3.5.5 Pros and Drawbacks

3.6 Client, Server and Daemon Thread

3.7 Technologies and Frameworks

4 — Implementation

4.1 Package structure

Package structure decision was as important task in PokeMongo, we wanted to ensure an high level of readability and maintainability. Although the classical "root package" which specifies the "domain.company.projet", in our case "it.unipi.dii.lsmsd.pokemongo", all the packages are structured *by layers*. In this way, we decided to name the packages according to they function architecturally rather than their identity according to the business domain. Here the structure:

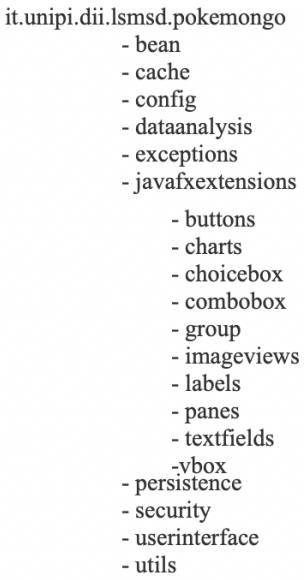


Figure 17: Package Structure

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.

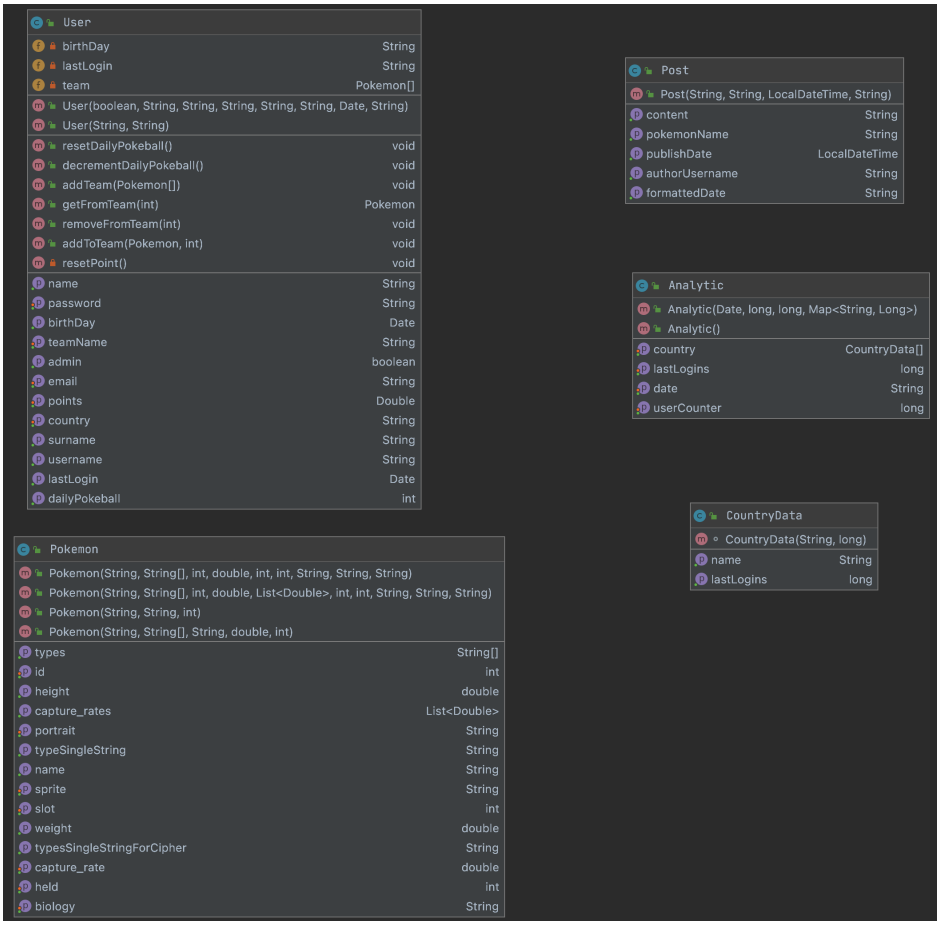


Figure 18: Bean Package Class Structure

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.

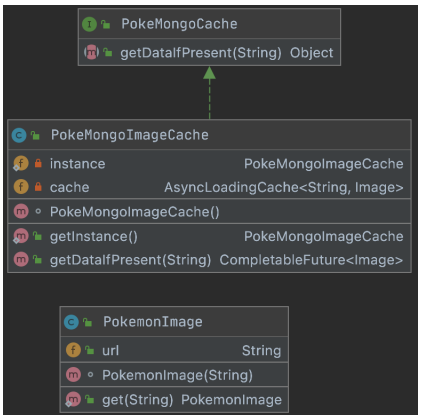


Figure 19: Cache Package Class Structure

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.

5 — Test

5.1 Privacy and Security

5.2 Unit Test

5.3 Robustness

5.4 Performance