UNIVERSITY OF PISA

COMPUTER ENGINEERING MASTER DEGREE

Large-Scale and Multi-Structured DataBases

WeFood

Professors:

Pietro Ducange

Alessio Schiavo

Group Members:

Giovanni Ligato

Cleto Pellegrino

Giuseppe Soriano

ACADEMIC YEAR 2023/2024

Index

1. Introduction

WeFood is a Social Network where users can share their recipes and provide feedbacks about other users' recipes through comments and star rankings. It manages in a completely automatic way the calories of the recipes, so the users do not have to worry about that when they post a new recipe. Using the search engine, users can discover new top rated recipes to amaze their friends, filtering by ingredients or calories. Furthermore, users can follow other users and get suggestions about new users to follow.

2. Requirements

Describing the requirements it is important to distinguish between functional and non-functional requirements.

2.1. Functional Requirements

The main functional requirements for *WeFood* can be organized by the actor that is involved in the use case.

1. **Unregistered User**:
   - 1.1. Browse *recent*[1] recipes;
   - 1.2. Sign Up.
2. **Registered User**:
   - 2.1. Log In;
   - 2.2. Log Out;
   - 2.3. Upload a Post (Recipe);
   - 2.4. Modify his/her Posts;
   - 2.5. Delete his/her Posts;
   - 2.6. Comment a Post;
   - 2.7. Modify his/her own comments;
   - 2.8. Delete his/her own comments or comments on his/her Posts;
   - 2.9. Evaluate by a star ranking a Post;
   - 2.10. Delete his/her own star rankings;
   - 2.11. View the Recipe of a Post;
   - 2.12. View the Total Calories of a Recipe;
   - 2.13. View the Steps of a Recipe;

[1] Time interval can be manipulated by the actor using a slider.

2.14. View the Ingredients of a Recipe;

2.15. View the Calories of an Ingredient;

2.16. Browse most *recent* Posts;

2.17. Browse most *recent* top rated Posts;

2.18. Browse most *recent* Posts by ingredients;

2.19. Browse most *recent* Posts by calories (minCalories and maxCalories);

2.20. View his/her own personal profile;

2.21. Modify his/her own personal profile (e.g. change Name, Surname, Password);

2.22. Delete his/her own personal profile;

2.23. Find a User by username;

2.24. View other Users' profiles;

2.25. Follow a User;

2.26. Unfollow a User;

2.27. View his/her Friends (i.e. the Users he/she follows and that follow him/her);

2.28. View his/her Followers;

2.29. View his/her Followed Users.

3. **Admin**:

3.1 Log In;

3.2 Log Out;

3.3. Browse all the Users;

3.4. Browse all the Posts;

3.5. Ban a User;

3.6. Unban a banned User;

3.7. Delete a Post;

3.8. Delete a Comment;

3.9. See statistics about the usage of *WeFood*;

3.10. Add a new Ingredient.

2.2. Non-Functional Requirements

The non-functional requirements for *WeFood* are as follows.

1. **Performance**: the overall system must be able to handle a request in less than `1.5` seconds, because the user experience would be negatively affected by a longer response time. Being a social network, it is necessary to have a good performance in order to provide a good user experience.
2. **Availability**: the system must be available 24/7 for allowing users to use it at any time.
3. **Security**: the system must be secure and protect users' data even from possible attacks. In particular, the information transmitted between client and server must be over HTTPS. Furthermore, the system must protect users' passwords by hashing them before storing in the database.
4. **Reliability**: the system must be reliable and must not lose the information uploaded by the users. It must be caple of recovering from a crash and restore the data in a consistent state, exploiting the replicas of the database.
5. **Usability**: the GUI offered to the users must be easy to use and intuitive. Each user should be able to use the application without any training and in about 15 minutes.
6. The **Back-End** must be written in Java.

3. Design

After having defined the requirements, it is possible to proceed with the design of the system.

3.1. Use Case Diagram

Translating the requirements into a graphical representation we obtain the *UML Use Case Diagram* shown in Figure 1.

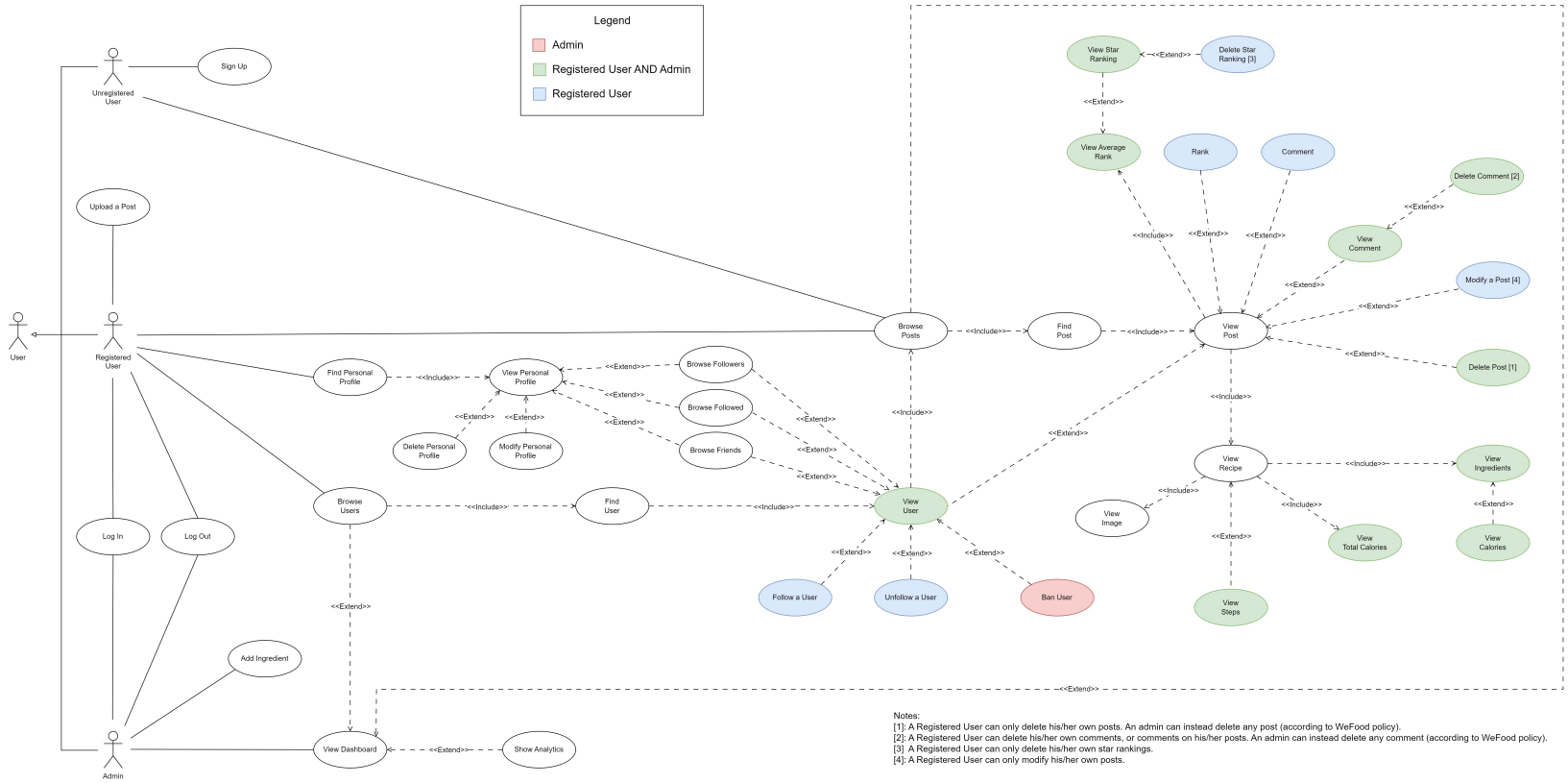

Figure 1: UML Use Case Diagram.

3.2. Class Diagram

The *UML Class Diagram* shown in Figure 2 represents the main entities of the system and their relationships.

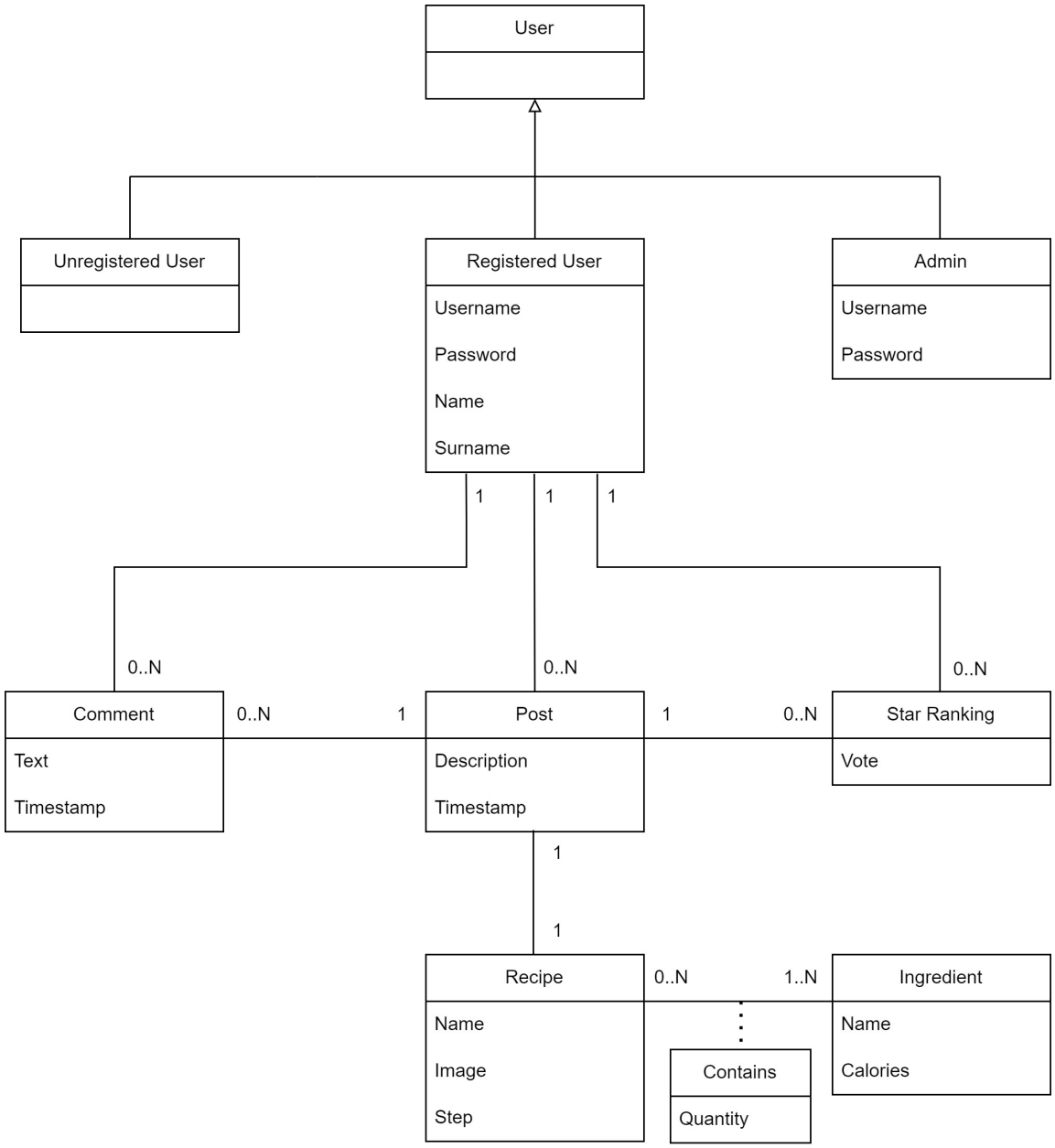

Figure 2: UML Class Diagram.

4. DataBases

Before cleaning and preparing the dataset needed to populate the databases, it is necessary to define the structure of the latter. In particular, two different databases will be used: a document DB and a graph DB.

4.1. Document DB

The entities managed by the document DB are the following.

- User (Unregistered User, Registered User, Admin);
- Post;
- Recipe;
- Comment;
- StarRanking;
- Ingredient.

4.1.1. Collections

The collections designed for storing the information inside the document DB are three: User, Post and Ingredient.

The structure of the User collection is as follows.

```
[User]:
{
    _id: ObjectId('...'),
    type: "Admin", # Applicable only for Admin
    username: String, [UNIQUE]
    password: String,
    name: String, # Not Applicable for Admin
    surname: String, # Not Applicable for Admin
    posts: [{ [REDUNDANCY(1)]
                idPost: ObjectId('...'),
                name: String,
                image: String
    }, ...] # Not Applicable for Admin
}
```

This collection is used both to store information about the Registered Users and the Admins. The field `type` is used to distinguish between the two types of Users, and it is set only for the Admins because they will be in minority compared to the Registered Users. The field `username` is required for the authentication of the Users and it is unique. So there cannot be two Users with the same username. The `password` is used for the authentication as well, and it contains the password provided by the User at the moment of the registration. For security reasons, the password is hashed before being stored in the database. The fields `name` and `surname` are used to store the name and the surname of the Registered Users and hence they are not applicable for the Admins for which it is not necessary to know their names and surnames. Lastly, the field `posts` is used to link (i.e. document linking) the Registered Users with their Posts. In particular, it contains a list of

objects, each one containing the id of a Post and the `name` and the `image` of the Recipe of the Post. The fields `name` and `image` are redundant because they are already stored in the Post collection, but they are useful for the queries that involve the User collection because they avoid the need of joining the Post collection.

The structure of the Post collection is a little bit more complex because it manages the information about the Posts, the Recipes, the Comments and the StarRankings. The decision of storing a Post in his own collection instead of embedding it in the document of the User that created it (i.e. document embedding) is due to several reasons:

- a User can publish an unlimited number of Posts, and a Post can have an many Comments and StarRankings. So the size of the document of a User would have grown indefinitely and very quickly, and this would have lead to a degradation of the performance of the system;
- the Posts are the main entities of the system and they are used in many queries. So having them in a separate collection ensures better performance by limiting the size of the individual document and taking advantage of tailored indexes.

```
[Post]:
{
    _id: ObjectId('...'),
    idUser: ObjectId('...'),
    username: String, [REDUNDANCY(2)]
    description: String,
    timestamp: Long,
    recipe: {
                name: String,
                image: String,
                steps: [String, ...],
                totalCalories: Double, [REDUNDANCY(3)]
                ingredients: [{
                                name: String,
                                quantity: Double
                }, ...]
    },
    starRankings: [{
                        idUser: ObjectId('...'),
                        username: String, [REDUNDANCY(4)]
                        vote: Double
    }, ...],
    avgStarRanking: Double, [REDUNDANCY(5)]
    comments: [{
                idUser: ObjectId('...'),
                username: String, [REDUNDANCY(6)]
                text: String,
                timestamp: Long
    }, ...]
}
```

Here the field `idUser` is used to link the Post with the Registered User that created it and `username` contains his/her username. The field `description` is used to store the description of the Post provided by the User. The field `timestamp` is used to store the timestamp of when the Post was uploaded. The field `recipe` is used to store the information about the Recipe contained in the Post. In particular, it contains the `name` and the `image` of the Recipe, the `steps` of the Recipe, the `totalCalories` of the Recipe and the list of `ingredients` of the Recipe together with the respective quantities in grams. It is important to notice that `image` does not contain the whole image, but just the URL of the image. The latter is stored or online or locally in the server. The field `starRankings` is used to store the information about the star rankings of the Post. In particular, it contains a list of objects, each one containing the id and the `username` of the Registered User that provided the star ranking and the `vote`, that represents the vote expressed by the Registerd User. The field `avgStarRanking` is used to store the average of the star rankings of the Post. The field `comments` is used to store the information about the comments of the Post. It contains a list of objects, each one containing the id and the `username` of the RegisteUser that provided the comment, the `text` and the `timestamp` of the comment.

The last collection is the Ingredient collection, that is used to store the information about the Ingredients. The structure of the Ingredient collection is very simple because it contains only the `name` and the `calories` per 100 grams of the Ingredient. The `name` is unique because does not make sense to have two Ingredients with the same name.

```
[Ingredient]:
{
    _id: ObjectId('...'),
    name: String, [UNIQUE]
    calories: Double
}
```

4.2. Graph DB

The entities that are managed by the graph DB are just: User (Registered User), Recipe and Ingredient.

In the following two sections, a comprehensive analysis of the nodes and relationships within the Graph DB schema, as illustrated in Figure 3, will be presented.

4.2.1. Nodes

The nodes designed for storing the information inside the graph DB are three, one for each entity.

The User node is used to store the information about the Registered Users. Each node contains the `_id` of the Registered User that is stored in the User collection of the document DB and his/her `username`.

```
(User):
    - _id: String
    - username: String [REDUNDANCY(7)]
```

The Recipe node is used to store the information about the Recipes. Each node contains the `_id` of the Post that contains the Recipe, which is stored withing the Post collection of the document DB.

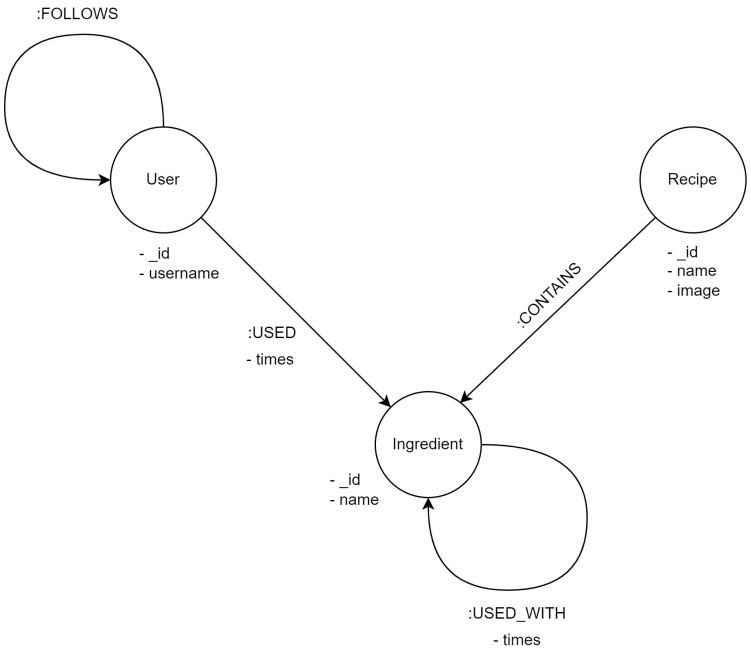

Figure 3: Graph DB schema.

Additionally, it includes the Recipe's `name` along with its corresponding `image`.

```
(Recipe):
    - _id: String
    - name: String [REDUNDANCY(8)]
    - image: String [REDUNDANCY(9)]
```

The Ingredient node is used to store the information about the Ingredients. Each node contains the `_id` of the Ingredient that is stored in the Ingredient collection of the document DB and the `name` of the Ingredient.

```
(Ingredient):
    - _id: String
    - name: String [REDUNDANCY(10)]
```

4.2.2. Relationships

Between the described nodes are possible several relationships that are formally defined as follows.

`(User)-[:FOLLOWS]->(User)`

This relationship allows Users to follow other Users. Two Users become friends when they follow each other.

`(User)-[:USED]->(Ingredient)`

```
(times: int) [REDUNDANCY(11)]
```

This relationship, instead, allows to quickly retrieve the Ingredients that have been used by the Users in their Recipes. The `times` attribute, in addition to being used for counting the number of times that an Ingredient has been used by a User, is used to keep track of the fact that the relationship with the Ingredient still can exist in other Recipes after the deletion of a Recipe by a User. Only when `times` becomes 0 the relationship can be deleted.

```
(Ingredient)-[:USED_WITH]->(Ingredient)
             (times: int) [REDUNDANCY(12)]
            [BIDIRECTIONAL]
```

This relationship allows to quickly retrieve the Ingredients that have been used together in the Users' Recipes. Here the `times` attribute is used for counting the number of times that two Ingredients have been used together. This relationship is bidirectional because if an Ingredient `A` has been used with an Ingredient `B`, then also the Ingredient `B` has been used with the Ingredient `A`.

```
(Recipe)-[:CONTAINS]->(Ingredient)
```

This last relationship allows to retrieve the Ingredients that are contained in a Recipe.

4.3. Redundancies

The proposed database models have redundancies, denoted as `[REDUNDANCY(n)]`. This means that the information they contain can be obtained through alternative means, often involving more intricate operations than a straightforward retrieval of the redundant value. These redundancies were cautiously introduced to enhance the system's reading performance and subsequently reduce response times. However, to maintain *data consistency*, writing operations are necessary to keep the redundancies updated. While reading operations are more frequent for the redundancies, the writing operations are generally less frequent. This approach is deemed more convenient for optimizing overall system performance. Redundancies also allow to avoid the need for joins, particularly when dealing with inter-database connections. A detailed explanation of the reasons justifying the introduction of the redundancies is provided in Table 1.

Table 1: Redundancies introduced into the database models.

(1) `DocumentDB:User:posts` **Reason**: To avoid joins. **Original/Raw Value**: `DocumentDB:Post`
(2) `DocumentDB:Post:username` **Reason**: To avoid joins. **Original/Raw Value**: `DocumentDB:User:username`
(3) `DocumentDB:Post:recipe:totalCalories` **Reason**: To avoid joins and to avoid computing the total calories of a Recipe every time a Post is shown.

Original/Raw Value: It is possible to compute the total calories of a Recipe by summing the calories of the Ingredients contained in the Recipe In particular the precise formula is the following: $\sum_i \left(quantity_i \cdot \frac{calories100g_i}{100} \right)$ where $quantity_i$ is the quantity of the i-th Ingredient contained in the Recipe and $calories100g_i$ is the amount of calories contained in 100 grams of the i-th Ingredient that can be retrieved from the `Ingredient` collection.

---

(4) `DocumentDB:Post:starRankings:username`
Reason: To avoid joins.
Original/Raw Value: `DocumentDB:User:username`

---

(5) `DocumentDB:Post:avgStarRanking`
Reason: To avoid computing the average star ranking of a Post every time is shown.
Original/Raw Value: It is possible to compute the average star ranking of a Post by averaging the values contained in `DocumentDB:Post:starRankings:vote`

---

(6) `DocumentDB:Post:comments:username`
Reason: To avoid joins.
Original/Raw Value: `DocumentDB:User:username`

---

(7) `GraphDB:(User):username`
Reason: To avoid joins with the DocumentDB.
Original/Raw Value: `DocumentDB:User:username`

---

(8) `GraphDB:(Recipe):name`
Reason: To avoid joins with the DocumentDB.
Original/Raw Value: `DocumentDB:Post:recipe:name`

---

(9) `GraphDB:(Recipe):image`
Reason: To avoid joins with the DocumentDB.
Original/Raw Value: `DocumentDB:Post:recipe:image`

---

(10) `GraphDB:(Ingredient):name`
Reason: To avoid joins with the DocumentDB.
Original/Raw Value: `DocumentDB:Ingredient:name`

---

(11) `GraphDB:(User)-[:USED]->(Ingredient):times`
Reason: To avoid computing the total number of times that a User used an Ingredient.
Original/Raw Value: It is possible to compute the total number of times that a User used an Ingredient by counting the number of times that the User used that Ingredient in his/her Recipes (information that can be retrieved from the DocumentDB).

---

(12) `GraphDB:(Ingredient)-[:USED_WITH]->(Ingredient):times`
Reason: To avoid computing the number of times that an Ingredient is used with another one.
Original/Raw Value: It is possible to compute the number of times that an Ingredient is used with another one by counting the number of times that all the Users used these two Ingredients together in their Recipes (information that can be retrieved from the DocumentDB).

---

5. Dataset

To populate the databases with a substantial volume of realistic data, datasets sourced from Kaggle were employed.

5.1. Raw Dataset

The initial raw datasets are related to the main functionalities of *WeFood*. In particular, datasets about recipes and ingredients were found.

- Calories per 100 grams in Food Items `[1]`
- Recipes and Interactions `[2]`
- Recipes and Reviews `[3]`

Contained in these datasets there are *almost* all the information needed to populate the databases. Indeed, in around 1 GB of raw data it is possible to find 2225 food items, over 500,000 recipes and 1,400,000 reviews. After a careful analysis, however, it was found that something was missing.

1. Personal Information about the Users: only the username of the Users was available (`AuthorName` in `recipes.csv` of `[3]`).
2. The quantity of each ingredient in the recipes: `RecipeIngredientQuantities` in `recipes.csv` of `[3]` contains some numbers that could be useful for this purpose, but they are not clear and there is no documentation about them. Indeed there is no way to understand if they are the quantities of the ingredients in grams or in other units of measure (e.g. just to have an idea there numbers like the following: 1, 1/2, 3, 5, etc). Furthermore, there are a lot of `NA` values in this column.

Everything else is in the datasets, and need only to be cleaned and appropriately merged to obtain the structure needed for the population of the databases.

5.2. Cleaning Process

There is the need to clean the datasets because in them there are plenty of information that are not useful for the purposes of *WeFood* and would result only in a waste of space. For achieving this goal, the datasets were analyzed in detail and the information that were not useful were discarded. The cleaning process was performed using Python.

Below a separate description of the cleaning process for each entity identified in the design phase is provided.

Ingredient: The starting point was: `ingr_map.pickle` in `[2]`. Here there are lots of fields useful for machine learning related tasks, but not for the *WeFood* purposes. Only fields strictly needed for linking recipes with `[1]` have been kept. The final result is the following. To facilitate referencing in the subsequent merging process, each intermediate product generated during the cleaning process will be assigned a distinct name, starting with the following.

```
Ingredient_A: {
    "raw_ingr":"pretzels",
    "replaced":"pretzel",
```

```
    "id":5711
}
```

The first field `raw_ingr` contains the original text of the ingredient, the one inserted by the user in the recipe. The second field `replaced` contains the simplified representation of the ingredient and the last field `id` contains the unique identifier of the ingredient.

At this point, the file `calories.csv` in `[1]` was analyzed. In this file in addition to the calories per 100 grams of each ingredient there are also Food Categories associated to them. These categories were really useful because they allowed to devise a plan for dealing with the lack of the quantity of each ingredient in the recipes in a simple but effective way. The idea was the following:

1. to associate to each `FoodCategory` two quantities, `quantity_min` and `quantity_max`, that are a realistic representation of the quantities used in real life for that specific `FoodCategory` (this labour intensive work was done with the support of *ChatGPT*);
2. to generate a random quantity for each ingredient in each recipe in the range `[quantity_min, quantity_max]`.

This solution does not provide a precise quantity for each ingredient in each recipe, but it is a good approximation that won't produce unrealistic results. This means that there won't be a recipe where there are 500 grams of `salt`, because the maximum quantity of `salt` that can be used in a recipe is 10 grams (i.e. `Herbs&Spices`, the `FoodCategory` of `salt`, has `quantity_max` equal to 10 grams).

After having removed some duplicates from `calories.csv`, based on the `FoodItem` field, the fields that were not useful were discarded. An example can be of help for understanding the final structure of the file.

```
Ingredient_B: {
    "FoodCategory":"Pastries,Breads&Rolls",
    "FoodItem":"Pretzel",
    "Cals_per100grams":"338 cal",
    "quantity_min":100,
    "quantity_max":500
}
```

Recipe: After the conversion of `RAW_recipes.csv` in `json` to have a more readable format, the file was analyzed in detail. Here there were lots of fields that could be discarded. The structure after the cleaning is, as before, better described by an example object.

```
Recipe_A: {
    "name":"pretzel crust",
    "id":194491,
    "contributor_id":356062,
    "submitted":"2006-11-07",
    "steps":"['preheat oven to 350', 'crush pretzels in a blender', 'add sugar
    ↪  and butter and mix well', 'press into a 9 inch pie plate', 'bake at 350
    ↪  for 8 minutes and cool', 'add desired filling']",
    "description":"recipe for a basic pretzel crust i found in a magazine. i
    ↪  don't think i saw it posted here yet.",
```

```
    "ingredients":"['pretzels', 'sugar', 'butter']"
}
```

Here the `name` is the name of the Recipe, `id` is the unique identifier of the Recipe and will be useful for linking the recipe with the interactions (i.e. comments and star rankings) of the dataset [2]. The `contributor_id` is the unique identifier of the User that created the Recipe. The `submitted` field is the timestamp of when the Recipe was uploaded. The `steps` field contains the steps of the Recipe, the `description` field contains the description that will be used for the Post that contains the Recipe and the `ingredients` field contains the list of the ingredients of the Recipe. Observing carefully the `steps` and the `ingredients` it is clear that they must be transformed into an array of strings because at the moment they are just strings.

From `recipes.csv` of [3], instead, it is possible to retrieve the URLs of the images of the Recipes. Thus only the fields `RecipeId` and `Images` are retained. Note that not all the Recipes have an image, and some of them have more than one image. Where no image is available, a default image will be applied. Viceversa, if multiple images are present, only the first image will be utilized.

```
Recipe_B: {
    "RecipeId":194491,
    "Image":"https:\/\/img.sndimg.com\/food\/image\/upload\/
    ↪  w_555,h_416,c_fit,fl_progressive,q_95\/v1\/img\/recipes
    ↪  \/19\/44\/91\/3xjO4aXTeiZpOajAsBRX_0S9A6246.jpg"
}
```

Comment: In `RAW_interactions.csv` of [2] the reviews (i.e. comments of *WeFood*) and the ratings (i.e. star rankings of *WeFood*) are stored together, because to each review there is associated a rating. In *WeFood* it is not the same, because a Registered User can leave a comment without providing a star ranking and viceversa. So the first step was to separate the two types of interactions.

```
Comment_A: {
    "user_id":430471,
    "recipe_id":194491,
    "timestamp":"2007-04-09"",
    "text":"This tasted great, however, I couldn't get it to hold together good.
    ↪  Either I didn't crush the pretzels small enough or I should have used a
    ↪  little more butter. But either way it was easy to make and tasted great.
    ↪  I used it as a base for a cheesecake pudding with fresh strawberries on
    ↪  the top."
}
```

Star Ranking: As previously mentioned, also the star rankings were stored in `RAW_interactions.csv` of [2]. For this reason the cleaning process was similar as the one used for the comments.

```
StarRanking_A: {
    "user_id":254614,
    "recipe_id":194491,
    "vote":4
}
```

Post: Posts are an abstraction of the Recipes that was introduced in *WeFood*. In the datasets there

is no information about them, so they will be created from scratch in the next merging process.

User: As previously noted, another lack of the datasets was the absence of personal information about the Users. Indeed, only the username of the Users was available. For not having an inconsistent situation where the Users have a name and a surname that are not coherent with their username, the username was dropped as well. In this way, it was possible to generate all the information needed for the Users using the Python library `Faker`. In particular, the name and the surname of the Users were generated randomly, and the username was computed accordingly using the following structure:

```
username = f"{name.lower()}_{surname.lower()}_{num}"
```

Where `num` is a random number in the range `[1, 99]`. All this is made paying attention to the fact that the username must be unique. By employing this approach, it was possible to create a User for each `contributor_id` of the Recipes (i.e. the Users who uploaded at least one Recipe). Users who did not upload any Recipe (i.e. who only appear in interactions) were excluded as a sufficient number of users was already available. The passwords (currently in plain) were generated randomly as well.

```
User_A: {
    "contributor_id":356062,
    "name":"Justin",
    "surname":"Alexander",
    "username":"justin_alexander_34",
    "password":"e6FMX30hGu"
}
```

5.3. Merging Process

After having cleaned the datasets, and having generated the missing information, it was possible to proceed with the merging process. This step was necessary to recreate the structure needed for the population of the databases. Also the merging process was performed using Python and Jupyter Notebook.

Because the merging process aims to produce the final structure of the documents that will be stored in the document DB, the latter will follow a different partitioning compared to the one used in the cleaning process. Here the partitioning will be based on the collections of the document DB.

Ingredient: Obtaining the final structure for the `Ingredient` collection is straightforward if `Ingredient_B` is considered. Indeed, it is sufficient to:

- rename `FoodItem` to `name`;
- rename `Cals_per100grams` to `calories`;
- take the float value of `calories`;
- discard `FoodCategory`, `quantity_min` and `quantity_max`.

```
{
    name: "Pretzel",
    calories: 338.0
}
```

The `_id` field will be automatically generated at the moment of the insertion in the database.

Post: Being the main collection of *WeFood*, the `Post` collection was also the one that takes more time to be merged. For this reason it is necessary to describe a step at a time for not complicating too much the explanation.

1. Merge of `Recipe_A` and `Recipe_B` on `id` and `RecipeId` respectively for including the URLs of the images inside the Recipes. In this way `Recipe_AB` is obtained.
2. The subsequent task involves converting the `ingredients` array of strings within `Recipe_A` into an array of objects. Each of these objects will include in the final structure the ingredient's `name` and its corresponding `quantity` expressed in grams. It is crucial to emphasize that the `name` of the ingredient must match an entry in the Ingredient collection. For achieving this:
    a. Firstly, `Ingredient_A` and `Ingredient_B` are matched on `replaced` and `FoodItem` respectively. For maximizing the number of matches, the matching was performed with the support of a Python Library called `fuzzywuzzy` which merges strings by likelyhood using *Levenshtein Distance*, which is a metric used in information theory, linguistics, and computer science for measuring the difference between two sequences. Thanks to this approach, no `Ingredient_A` was left unmatched with an `Ingredient_B`. An example of the matching is the following.

```
Ingredient_AB: {
    "raw_ingr":"pretzels",
    "replaced":"pretzel",
    "id":5711,
    "FoodCategory":"Pastries,Breads&Rolls",
    "FoodItem":"Pretzel",
    "Cals_per100grams":"338 cal",
    "quantity_min":100,
    "quantity_max":500
}
```

    b. By noticing that the `raw_ingr` field of `Ingredient_AB` contains the name of the ingredients as they are inserted by the users in the recipes, it possible to do a further match (this time using exact equality) between the latter and the strings contained in `ingredients` of `Recipe_AB`.

    c. At this point, `ingredients` will be an array of objects of the type of `Ingredient_AB`. For each of this object, it is possible to add a field `quantity` that is a random number in the range `[quantity_min, quantity_max]`. This is the quantity of the ingredient in grams that will be used in the recipe. After removing the no longer needed fields and making other minor modifications, here's the new structure.

```
Recipe_C: {
    "name": "pretzel crust",
    "id": 194491,
    "contributor_id": 356062,
    "submitted": "2006-11-07",
    "steps": [
        "preheat oven to 350",
        "crush pretzels in a blender",
```

```
            "add sugar and butter and mix well",
            "press into a 9 inch pie plate",
            "bake at 350 for 8 minutes and cool",
            "add desired filling"
        ],
        "description": "recipe for a basic pretzel crust i found in a
        ↪  magazine. i don't think i saw it posted here yet.",
        "ingredients": [
            {
                "foodItem": "Pretzel",
                "Cals_per100grams": 338.0,
                "quantity": 176
            },
            {
                "foodItem": "Sugar",
                "Cals_per100grams": 405.0,
                "quantity": 102
            },
            {
                "foodItem": "Butter",
                "Cals_per100grams": 720.0,
                "quantity": 43
            }
        ],
        "Image": "https://img.sndimg.com/food/image/upload/
        ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
        ↪  19/44/91/3xjO4aXTeiZpOajAsBRX_0S9A6246.jpg"
    }
```

d. Now, exploiting the field `Cals_per100grams`, which has not been discarded yet, it is possible to compute the `totalCalories` of the recipe. This is done by executing the following operations for each recipe.

```
totalCalories = 0
for ingredient in ingredients:
    totalCalories += ingredient["quantity"] *
↪  (ingredient["Cals_per100grams"]/100)
```

e. In the end, the desired structure for `ingredients` was obtained.

```
Recipe_D: {
    "name": "pretzel crust",
    "id": 194491,
    "contributor_id": 356062,
    "submitted": "2006-11-07",
    "steps": [
        "preheat oven to 350",
        "crush pretzels in a blender",
```

```
            "add sugar and butter and mix well",
            "press into a 9 inch pie plate",
            "bake at 350 for 8 minutes and cool",
            "add desired filling"
        ],
        "description": "recipe for a basic pretzel crust i found in a
        ↪  magazine. i don't think i saw it posted here yet.",
        "ingredients": [
            {
                "foodItem": "Pretzel",
                "quantity": 176
            },
            {
                "foodItem": "Sugar",
                "quantity": 102
            },
            {
                "foodItem": "Butter",
                "quantity": 43
            }
        ],
        "totalCalories": 1317.58,
        "Image": "https://img.sndimg.com/food/image/upload/
        ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
        ↪  19/44/91/3xjO4aXTeiZpOajAsBRX_0S9A6246.jpg"
    }
```


3. Creation of the Posts involves relocating fields unrelated to the Recipes. Furthermore, the Posts themselves contain the Recipes.


```
Post_A: {
    "id": 194491,
    "idUser": 356062,
    "description": "recipe for a basic pretzel crust i found in a magazine. i
    ↪  don't think i saw it posted here yet.",
    "timestamp": "2006-11-07",
    "recipe": {
        "name": "pretzel crust",
        "image": "https://img.sndimg.com/food/image/upload/
        ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
        ↪  19/44/91/3xjO4aXTeiZpOajAsBRX_0S9A6246.jpg",
        "steps": [
            "preheat oven to 350",
            "crush pretzels in a blender",
            "add sugar and butter and mix well",
            "press into a 9 inch pie plate",
            "bake at 350 for 8 minutes and cool",
```

```
            "add desired filling"
        ],
        "ingredients": [
            {
                "foodItem": "Pretzel",
                "quantity": 176
            },
            {
                "foodItem": "Sugar",
                "quantity": 102
            },
            {
                "foodItem": "Butter",
                "quantity": 43
            }
        ],
        "totalCalories": 1317.58
    }
}
```


4. By considering `Comment_A` and `StarRanking_A`, it is possible to create the `comments` and `starRankings` arrays of the Posts. Specifically, each Comment or StarRanking is added to the corresponding Post based on the `recipe_id` field.


```
Post_B: {
    "id": 194491,
    "idUser": 356062,
    "description": "recipe for a basic pretzel crust i found in a magazine. i
    ↪  don't think i saw it posted here yet.",
    "timestamp": "2006-11-07",
    "recipe": {
        "name": "pretzel crust",
        "image": "https://img.sndimg.com/food/image/upload/
        ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
        ↪  19/44/91/3xjO4aXTeiZpOajAsBRX_OS9A6246.jpg",
        "steps": [
            "preheat oven to 350",
            "crush pretzels in a blender",
            "add sugar and butter and mix well",
            "press into a 9 inch pie plate",
            "bake at 350 for 8 minutes and cool",
            "add desired filling"
        ],
        "ingredients": [
            {
                "foodItem": "Pretzel",
                "quantity": 176
```

```
            },
            {
                "foodItem": "Sugar",
                "quantity": 102
            },
            {
                "foodItem": "Butter",
                "quantity": 43
            }
        ],
        "totalCalories": 1317.58
    },
    "comments": [
        {
            "user_id": 430471,
            "timestamp": 1176076800000,
            "text": "This tasted great, however, I couldn't get it to hold
            ↪  together good.  Either I didn't crush the pretzels small
            ↪  enough or I should have used a little more butter. But either
            ↪  way it was easy to make and tasted great.  I used it as a
            ↪  base for a cheesecake pudding with fresh strawberries on the
            ↪  top."
        },
        {
            "user_id": 254614,
            "timestamp": 1182902400000,
            "text": "You have to crush the pretzels fine. There is a definite
            ↪  taste of salt, sugar and butter in the crust. It was great
            ↪  with a pudding pie filling but I would not make it with a
            ↪  fruit filling.You want the crust flavor to be a part of the
            ↪  dessert. Add waxed paper or non stick foil to press into pie
            ↪  pan, works very well. Thanks for posting."
        }
    ],
    "starRankings": [
        {
            "user_id": 430471,
            "vote": 3
        },
        {
            "user_id": 254614,
            "vote": 4
        }
    ]
}
```


5. To reconstruct the collection's structure there isn't much left to do. Indeed, it is only necessary

to convert the `timestamp` of creation of the Post in `Long`, to add the `avgStarRanking` field and to include the `username` of the Users, available in `User_A`, whenever their `id` appears.

```
Post_C: {
    "id": 194491,
    "idUser": 356062,
    "username": "justin_alexander_34",
    "description": "recipe for a basic pretzel crust i found in a magazine. i
    ↪  don't think i saw it posted here yet.",
    "timestamp": 1162857600000,
    "recipe": {
        "name": "pretzel crust",
        "image": "https://img.sndimg.com/food/image/upload/
        ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
        ↪  19/44/91/3xjO4aXTeiZpOajAsBRX_OS9A6246.jpg",
        "steps": [
            "preheat oven to 350",
            "crush pretzels in a blender",
            "add sugar and butter and mix well",
            "press into a 9 inch pie plate",
            "bake at 350 for 8 minutes and cool",
            "add desired filling"
        ],
        "ingredients": [
            {
                "quantity": 176,
                "name": "Pretzel"
            },
            {
                "quantity": 102,
                "name": "Sugar"
            },
            {
                "quantity": 43,
                "name": "Butter"
            }
        ],
        "totalCalories": 1317.58
    },
    "comments": [
        {
            "timestamp": 1176076800000,
```

```
                "text": "This tasted great, however, I couldn't get it to hold
                ↪  together good.  Either I didn't crush the pretzels small
                ↪  enough or I should have used a little more butter. But either
                ↪  way it was easy to make and tasted great.  I used it as a
                ↪  base for a cheesecake pudding with fresh strawberries on the
                ↪  top.",
                "idUser": 430471,
                "username": "bonnie_vincent_27"
            },
            {
                "timestamp": 1182902400000,
                "text": "You have to crush the pretzels fine. There is a definite
                ↪  taste of salt, sugar and butter in the crust. It was great
                ↪  with a pudding pie filling but I would not make it with a
                ↪  fruit filling.You want the crust flavor to be a part of the
                ↪  dessert. Add waxed paper or non stick foil to press into pie
                ↪  pan, works very well. Thanks for posting.",
                "idUser": 254614,
                "username": "christopher_bass_52"
            }
        ],
        "avgStarRanking": 3.5,
        "starRankings": [
            {
                "vote": 3,
                "idUser": 430471,
                "username": "bonnie_vincent_27"
            },
            {
                "vote": 4,
                "idUser": 254614,
                "username": "christopher_bass_52"
            }
        ]
    }
```


User: In `User_A`, the only thing missing for having the User collection is the array field `posts` which contains a simplified representation of the Posts uploaded by the User. By employing the `idUser` in `Post_C`, all the user's posts can be retrieved, and the necessary fields can be selected.


```
User_B: {
    "contributor_id": 356062,
    "name": "Justin",
    "surname": "Alexander",
    "username": "justin_alexander_34",
    "password": "e6FMX30hGu",
    "posts": [
```

```
        {
            "idPost": 234229,
            "name": "layered ice cream candy cake",
            "image": "https://img.sndimg.com/food/image/upload/
            ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
            ↪  23/42/29/picztrpPR.jpg"
        },
        {
            "idPost": 194491,
            "name": "pretzel crust",
            "image": "https://img.sndimg.com/food/image/upload/
            ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
            ↪  19/44/91/3xjO4aXTeiZpOajAsBRX_0S9A6246.jpg"
        },
        {
            "idPost": 226341,
            "name": "quesadilla combos",
            "image": "https://img.sndimg.com/food/image/upload/
            ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
            ↪  22/63/41/Xc9MqI3jSBmO7sx9cUFN_quesadilla-combos_0511.jpg"
        },
        {
            "idPost": 282971,
            "name": "raspberry lime rugalach",
            "image": "https://img.sndimg.com/food/image/upload/
            ↪  w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/
            ↪  28/29/71/picYumCGj.jpg"
        },
        ...
    ]
}
```


5.4. Population

After completing the preceding steps, the datasets have been appropriately cleansed, merged, and are now prepared for being imported into the databases. The file dimensions are as follows:

- Post collection: approximately 700MB;
- User: 69MB;
- Ingredient: 266KB.

However, specific procedures are needed to populate the two types of databases.

5.4.1. Document DB

It's true that all the necessary files for populating the documentDB have been generated, but there are some points that may need further clarifications.

- Firstly, in cases where a field is null or empty (e.g. a Post with no associated image, comment, or star ranking), within the context of a NoSQL database, there is no necessity for these fields to be assigned the null value to reduce unnecessary memory usage. Consequently, if there is missing information in a field, that field will simply be absent from the structure.
- Secondly, the `_id` fields are not yet incorporated into the aforementioned structures, while conversely, the old *ids* from the datasets still persist. This is due to the intention to utilize the `_id` values provided by MongoDB (i.e. the documentDB employed in the implementation). To preserve the connection between documents and facilitate the substitution, a straightforward yet effective procedure was implemented. Documents were imported into their respective MongoDB collections, and subsequently, a *JSON export* was performed, leading to the insertion of MongoDB's `_id` within all the documents. Utilizing the old *ids*, the linkage between documents was established, and a substitution was executed by assigning the new MongoDB `_id` whenever the old *id* was encountered. Subsequently, all collections were re-imported. In this way, the linkage was now based on the MongoDB `_id`, and no problems were encountered at all.

5.4.2. Graph DB

Populating Neo4j (i.e. the adopted graphDB) proved to be a bit more challenging. Unlike MongoDB, the import process from JSON is not as straightforward. Moreover, the structure defined earlier was specific to the MongoDB collections. To address this, Python scripts were developed. After establishing a connection with the Neo4j DBMS using the Neo4j driver for Python, these scripts initiate the creation of all the nodes before establishing the relationships between them. This process relies on both *Cypher* queries and the information found in the JSON documents to construct the entire graph database from scratch.

6. Queries

Here are all the queries required to access the databases and implement *WeFood* functionalities. They are grouped into basic CRUD operations and more intricate aggregations or query suggestions.

6.1. CRUD operations

The set of fundamental operations includes creating, reading, updating, and deleting data within the databases.

6.1.1. Create

Creation operations are:

1. `Create a new User`: a new User signs up to *WeFood*;
2. `Create a new Post / Recipe`: a User uploads a new Recipe;
3. `Create a new Comment`: a User comments a Post;
4. `Create a new StarRanking`: a User rates a Post;
5. `Create a new Ingredient`: an Admin adds a new ingredient;
6. `Create a new following relationship`: a User follows another User;
7. `Create a new Recipe-Ingredient relationship`: a new recipe is created and contains an ingredient;
8. `Create a new User-Ingredient relationship`: a User uses an ingredient;
9. `Create a new Ingredient-Ingredient relationship`: an ingredient is used with another ingredient.

MongoDB

1. `Create a new user`:

```
db.User.insertOne({
    username: String,
    password: [HASHEDSTRING],
    name: String,
    surname: String
})
```

2. `Create a new Post`:

```
db.Post.insertOne({
    idUser: ObjectId("..."),
    username: String,
    description: String,
    timestamp: Long,
    recipe: {
        name: String,
        image: String,
        steps: [String, ...],
        totalCalories: Double,
        ingredients: [{
```

```
            name: String,
            quantity: Double,
        }, ...]
    }
})
```

3. Create a new Comment:

```
db.Post.updateOne({
    _id: ObjectId("...")
}, {
    $push: {
        comments: {
            idUser: ObjectId("..."),
            username: String,
            text: String,
            timestamp: Long
        }
    }
})
```

4. Create a new StarRanking:

```
db.Post.updateOne({
    _id: ObjectId("...")
}, {
    $push: {
        starRankings: {
            idUser: ObjectId("..."),
            username: String,
            vote: Double
        }
    }
})
```

5. Create a new Ingredient:

```
db.Ingredient.insertOne({
    name: String,
    calories: Double
})
```

Neo4j

1. Create a new User:

```
CREATE (u:User {
    _id: String,
    username: String
})
```

2. Create a new Recipe:

```
CREATE (r:Recipe {
    _id: String,
    name: String,
    image: String
})
```

5. Create a new Ingredient:

```
CREATE (i:Ingredient {
    _id: String,
    name: String
})
```

6. Create a new following relationship:

```
MATCH (u1:User {username: String}), (u2:User {username: String})
MERGE (u1)-[:FOLLOWS]->(u2)
```

7. Create a new Recipe-Ingredient relationship:

```
MATCH (r:Recipe {_id: String}), (i:Ingredient {name: String})
CREATE (r)-[:CONTAINS]->(i)
```

8. Create a new User-Ingredient relationship:

```
MATCH (u:User {username: String}), (i:Ingredient {name: String})
MERGE (u)-[r:USED]->(i) ON CREATE SET r.times = 1 ON MATCH SET r.times =
↪  r.times + 1
```

9. Create a new Ingredient-Ingredient relationship:

```
MATCH (i1:Ingredient {name: String}), (i2:Ingredient {name: String})
MERGE (i1)-[r:USED_WITH]->(i2) ON CREATE SET r.times = 1 ON MATCH SET r.times
↪  = r.times + 1
```

6.1.2. Read

Reading operations are:

1. Find User by username;
2. Find User Page by username;
3. Get all the Ingredients;
4. Find Ingredient by name;
5. Find Most Recent Top Rated Posts;
6. Find Most Recent Top Rated Posts by set of ingredients;
7. Find Most Recent Posts by minCalories and maxCalories;
8. Find Post by Recipe name;
9. Find Post by _id;
10. Find Banned Users;
11. Find Users Followed by a User;

12. `Find Followers of a User`;
13. `Find Friends of a User`: a User's friends are the Users that follow him/her and that he/she follows;
14. `Find Recipes by set of ingredients`;

MongoDB

1. `Find User by username`:

```
db.User.find({
    username: String
}, {
    posts: 0
})
```

2. `Find User Page by username`:

```
db.User.find({
    username: String
}, {
    username: 1,
    posts: 1,
    deleted: 1
})
```

3. `Get all the Ingredients`:

```
db.Ingredient.find({}, {
    _id: 0
})
```

4. `Find Ingredient by name`:

```
db.Ingredient.find({
    name: String
}, {
    _id: 0
})
```

5. `Find Most Recent Top Rated Posts`:

```
db.Post.find({
    timestamp: {
        $gte: Long
    }
}, {
    "recipe.name": 1,
    "recipe.image": 1
}).sort({
    avgStarRanking: -1
}).limit(limit)
```

6. Find Most Recent Top Rated Posts by set of ingredients:

```
db.Post.find({
    timestamp: {
        $gte: Long
    },
    "recipe.ingredients.name": {
        $all: [String, ...]
    }
}, {
    "recipe.name": 1,
    "recipe.image": 1
}).sort({
    avgStarRanking: -1
}).limit(limit)
```

7. Find Most Recent Posts by minCalories and maxCalories:

```
db.Post.find({
    timestamp: {
        $gte: Long
    },
    "recipe.totalCalories": {
        $gte: minCalories,
        $lte: maxCalories
    }
}, {
    "recipe.name": 1,
    "recipe.image": 1
}).sort({
    timestamp: -1
}).limit(limit)
```

8. Find Posts by Recipe name:

```
db.Post.find({
    "recipe.name": {
        $regex: String,
        $options: "i"
    }
}, {
    "recipe.name": 1,
    "recipe.image": 1
}).limit(10)
```

9. Find Post by _id:

```
db.Post.find({
    _id: ObjectId("..."),
}, {
```

```
    _id: 0,
    idUser: 0,
    "starRankings.idUser": 0,
    "comments.idUser": 0
})
```

10. Find Banned Users:

```
db.User.find({
    deleted: true,
    name: { $exists: true }
})
```

Neo4j

11. Find Users Followed by a User:

```
MATCH (u1:User {username: String})-[:FOLLOWS]->(u2:User)
RETURN u2
```

12. Find Followers of a User:

```
MATCH (u1:User)-[:FOLLOWS]->(u2:User {username: String})
RETURN u1
```

13. Find Friends of a User:

```
MATCH (u1:User {username: String})-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(u1)
RETURN u2
```

14. Find Recipes by set of ingredients:

```
MATCH (r:Recipe)-[:CONTAINS]->(i:Ingredient)
WHERE i.name IN [String, ...]
WITH r, COLLECT(i.name) AS ingredients
WHERE ALL(ingredient IN [String, ...]
          WHERE ingredient IN ingredients)
RETURN r
LIMIT 15
```

6.1.3. Update

Update operations are:

1. Update User's information: A User can update his/her password, name or surname;
2. Update Post: A User can update the description of a Post;
3. Update Comment: A User can update the text of a Comment;

MongoDB

1. Update User's information:

```
db.User.updateOne({
    _id: ObjectId("...")
```

```
}, {
    $set: {
        password: [HASHEDSTRING],
        name: String,
        surname: String
    }
})
```

2. Update Post:

```
db.Post.updateOne({
    _id: ObjectId("...")
}, {
    $set: {
        description: String
    }
})
```

3. Update Comment:

```
db.Post.updateOne({
    _id: ObjectId("..."),
    comments: {
        $elemMatch: {
            idUser: ObjectId("..."),
            timestamp: Long
        }
    }
}, {
    $set: {
        "comments.$.text": String
    }
})
```

6.1.4. Delete

Deletion operations are:

1. `Delete User`: Users have the option to delete their own profiles, bearing in mind that all *non-personal information* will be retained for statistical purposes. Once a profile is deleted, re-registration using the previous username is not permitted;
2. `Delete Post`;
3. `Delete Post from User`;
4. `Delete Comment`;
5. `Delete StarRanking`;
6. `Delete Recipe`;
7. `Delete following relationship`;
8. `Delete / Decrement User-Ingredient relationship`;

Deletions not allowed:

- `Delete Ingredient`: it is not possible to delete an Ingredient because otherwise all the Recipes that contain it would be inconsistent;
- `Delete Ingredient-Ingredient relationship`: this relationship is neither removed nor decremented when a Recipe is deleted, for statistical purposes.

MongoDB

1. `Delete User`: It is important to first mark the User as deleted before proceeding to erase his/her personal information.

```
db.User.updateOne({
    _id: ObjectId("...")
}, {
    $unset: {
        password: "",
        name: "",
        surname: "",
        posts: ""
    },
    $set: {
        deleted: true
    }
})
```

2. `Delete Post`:

```
db.Post.deleteOne({
    _id: ObjectId("...")
})
```

3. `Delete Post from User`:

```
db.User.updateOne({
    _id: ObjectId("...")
}, {
    $pull: {
        posts: {
            idPost: ObjectId("...")
        }
    }
})
```

4. `Delete Comment`:

```
db.Post.updateOne({
    _id: ObjectId("...")
}, {
    $pull: {
        comments: {
```

```
            username: String,
            timestamp: Timestamp
        }
    }
})
```

5. Delete StarRanking:

```
db.Post.updateOne({
    _id: ObjectId("...")
}, {
    $pull: {
        starRankings: {
            idUser: ObjectId("...")
        }
    }
})
```

Neo4j

1. Delete User:

```
MATCH (u:User {username: String})
DETACH DELETE u
```

6. Delete Recipe:

```
MATCH (r:Recipe {_id: String})
DETACH DELETE r
```

7. Delete following relationship:

```
MATCH (u1:User {username: String})-[r:FOLLOWS]->(u2:User {username: String})
DELETE r
```

8. Delete / Decrement User-Ingredient relationship:

```
MATCH (u:User {username: String})-[r:USED]->(i:Ingredient {name: String})
SET r.times = r.times - 1
WITH r
WHERE r.times = 0
DELETE r
```

6.2. Suggestions and Aggregations

In this section, more relevant queries are presented, categorized into two sub-sections: suggestions and aggregations. Suggestions queries propose new information to users, based on their preferences and the preferences of their friends. Aggregations queries, instead, offer statistical insights into the stored data.

6.2.1. Suggestions

1. Show Most / Least used Ingredients;
2. Show Most used Ingredients by a User;
3. Suggest users to follow: a User is suggested to follow the friends of his/her friends;
4. Suggest most popular combination of ingredients;
5. Suggest new ingredients based on friends' usage;
6. Suggest most followed users;
7. Find Users by Ingredient usage: find Users who have employed a particular ingredient most frequently.

Neo4j

1. Show Most / Least used Ingredients:

```
MATCH (u:User)-[r:USED]->(i:Ingredient)
RETURN i, SUM(r.times) AS times
ORDER BY times DESC
LIMIT 5

MATCH (u:User)-[r:USED]->(i:Ingredient)
RETURN i, SUM(r.times) AS times
ORDER BY times ASC
LIMIT 5
```

2. Show Most used Ingredients by a User:

```
MATCH (u:User {username: String})-[r:USED]->(i:Ingredient)
RETURN i, r.times AS times
ORDER BY times DESC
LIMIT 5
```

3. Suggest users to follow:

```
MATCH (u1:User {username:
↪  String})-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(u3:User)
WHERE (u2)-[:FOLLOWS]->(u1)
AND NOT (u1)-[:FOLLOWS]->(u3)
RETURN u3
LIMIT 10
```

4. Suggest most popular combination of ingredients:

```
MATCH (i1:Ingredient {name: String})-[r:USED_WITH]->(i2:Ingredient)
RETURN i1, i2, r.times AS times
ORDER BY times DESC
LIMIT 5
```

5. Suggest new ingredients based on friends' usage:

```
MATCH (u1:User {username:
↪  String})-[:FOLLOWS]->(u2:User)-[r:USED]->(i:Ingredient)
```

```
WHERE (u2)-[:FOLLOWS]->(u1)
AND NOT (u1)-[:USED]->(i)
RETURN i, r.times AS times
ORDER BY times DESC
LIMIT 5
```

6. Suggest most followed users:

```
MATCH (u1:User)-[:FOLLOWS]->(u2:User)
RETURN u2, COUNT(u1) AS followers
ORDER BY followers DESC
LIMIT 5
```

7. Find Users by Ingredient usage:

```
MATCH (u:User)-[r:USED]->(i:Ingredient {name: String})
RETURN u, i, r.times AS times
ORDER BY times DESC
LIMIT 10
```

6.2.2. Aggregations

(#1): Compute the *ratio of interactions* and the *average* avgStarRanking by distinguishing between posts with and without images (i.e. no field `image` inside `recipe`). For the Posts with images, the ratio of interactions are computed as follows:

$$ratioOfComments = \frac{TotNumberOfComments}{TotNumberOfPosts}$$

$$ratioOfStarRankings = \frac{TotNumberOfStarRankings}{TotNumberOfPosts}$$

here $TotNumberOfComments$, $TotNumberOfStarRankings$ and $TotNumberOfPosts$ are computed by considering only the Posts with images. Similarly, the same calculations are performed for Posts without images.

```
db.Post.aggregate([
    {
        $project: {
            _id: 1,
            hasImage: {
                $cond: {
                    if: {
                         $eq: [{ $type: "$recipe.image" }, "missing"]
                    },
                    then: false,
                    else: true
                }
            },
            comments: {
```

```
                $size: {
                    $ifNull: ["$comments", []]
                }
            },
            starRankings: {
                $size: {
                    $ifNull: ["$starRankings", []]
                }
            },
            avgStarRanking: {
                $ifNull: ["$avgStarRanking", 0]
            }
        }
    },
    {
        $group: {
            _id: "$hasImage",
            numberOfPosts: {
                $sum: 1
            },
            totalComments: {
                $sum: "$comments"
            },
            totalStarRankings: {
                $sum: "$starRankings"
            },
            avgOfAvgStarRanking: {
                $avg: "$avgStarRanking"
            }
        }
    },
    {
        $project: {
            _id: 0,
            hasImage: "$_id",
            ratioOfComments: {
                $divide: ["$totalComments", "$numberOfPosts"]
            },
            ratioOfStarRankings: {
                $divide: ["$totalStarRankings", "$numberOfPosts"]
            },
            avgOfAvgStarRanking: 1
        }
    }
])
```

(#2): Given a User, show the *number* of Comments and StarRankings he/she has done and the

average of this StarRankings.

```
db.Post.aggregate([
    {
        $match: {
            $or: [
                {"comments.username": String},
                {"starRankings.username": String}
            ]
        }
    },
    {
        $project: {
            filteredComments: {
                $filter: {
                    input: "$comments",
                    as: "comment",
                    cond: {$eq: ["$$comment.username", String]}
                }
            },
            filteredStarRankings: {
                $filter: {
                    input: "$starRankings",
                    as: "starRanking",
                    cond: {$eq: ["$$starRanking.username", String]}
                }
            }
        }
    },
    {
        $group: {
            _id: null,
            avgOfStarRankings: {
                $avg: {$sum: "$filteredStarRankings.vote"}
            },
            numberOfStarRankings: {
                $sum: {$size: "$filteredStarRankings"}
            },
            numberOfComments: {
                $sum: {$size: "$filteredComments"}
            }
        }
    },
    {
        $project: {
            _id: 0,
            numberOfComments: 1,
```

```
            numberOfStarRankings: 1,
            avgOfStarRankings: 1
        }
    }
])
```

(#3): After filtering the Recipes by name, retrieve the *average* amount of calories of the first 10 Recipes ordered by descending avgStarRanking.

```
db.Post.aggregate([
    {
        $match: {
            "recipe.name": { $regex: String, $options: "i" }
        }
    },
    {
        $sort: {
            avgStarRanking: -1
        }
    },
    {
        $limit: 10
    },
    {
        $group: {
            _id: null,
            avgOfTotalCalories: {
                $avg: "$recipe.totalCalories"
            }
        }
    },
    {
        $project: {
            _id: 0,
            avgOfTotalCalories: 1
        }
    }
])
```

(#4): Given a User, show the *average* `totalCalories` of the Recipes published by him/her.

```
db.Post.aggregate([
    {
        $match: {
            username: String
        }
    },
    {   $project: {
```

```
            recipeCalories: "$recipe.totalCalories"
        }
    },
    {   $group: {
            _id: null,
            avgCalories: {
                $avg: "$recipeCalories"
            }
        }
    }
])
```

7. DataBases Deployment

As previously mentioned, MongoDB and Neo4j are the chosen databases for the deployment phase, serving as the documentDB and graphDB, respectively. The subsequent discussion will focus on the precautions taken when deploying these databases in a real-world setting.

7.1. MongoDB

MongoDB is the first database to be deployed, and it has been set up in a cluster with *three* machines to improve fault tolerance.

7.1.1. ReplicaSet

To deploy MongoDB across different machines, a *ReplicaSet* was necessary. A ReplicaSet is a collection of `mongod` instances organized hierarchically to ensure *redundancy and high availability*. The ReplicaSet, named `lsmdb`, comprises a Primary node with a priority of `2` and two Secondary nodes with priorities of `1.5` and `1`. The Primary node is the sole member capable of receiving write operations. Once a write operation is executed on the Primary node, the Secondary nodes replicate the same operation in their datasets. Configuring the ReplicaSet settings allows specifying the desired level of acknowledgment for a write operation, known as the *write concern*. In the proposed scenario of *WeFood*, having a social network that handles non-sensitive data, waiting for all nodes to acknowledge the operation is unnecessary. In the event of a primary node that crash immediately after a write operation, resulting in the failure to replicate to secondary nodes, the data loss is acceptable. Users can simply redo the operation (after the secondary nodes have elected a new primary node) without encountering dangerous or critical consequences. Therefore, a write concern of `w: 1` is adopted, allowing the ReplicaSet to return control once *one* node has acknowledged the write operation.

Additional configurable options include `j` and `wtimeout`.

- The `j` option determines when a node acknowledges writes, either after applying the write operation in memory or after writing to the *on-disk journal*. The default, when `w: 1`, is as if it were specified as false, acknowledging writes after the write in memory.
- The `wtimeout` specifies the time limit for an operation, and the default is `0`, meaning the write operation will block indefinitely if the level of write concern is unachievable. This situation will be handled in the code by setting a *client-level* `wtimeout` of `5000ms` (5 seconds), causing an exception to be thrown if the write operation is not completed within this time frame.

Regarding reading operations, all members of the replica set *can* accept read operations, although by *default*, applications direct reads to the primary member. In the case of a social network like *WeFood*, read operations can be directed to secondary nodes, even if they are not as updated as the primary node, as MongoDB asynchronously updates data to the secondary nodes. Thus, to ensure the lowest response time for read operations, the *read concern* is set to `nearest` at the *client-level*, meaning read operations will be performed on the nearest node (i.e. the node with the lowest latency).

7.1.2. Sharding

When it comes to *Sharding*, it is crucial to assess before the potential benefits it can bring. Sharding is the horizontally partitioning of data across multiple servers, that can help in enhancing scalability and performance. However, in certain situations, opting for sharding may prove impractical or undesirable. For instance, in the current implementation of *WeFood*, sharding the Post collection based on a specific field, such as `timestamp`, could result in latency issues when querying with unrelated filters (e.g., by `totalCalories` or by `avgStarRanking`), leading to an inefficient process.

To elaborate a little further, if the application was designed with predefined *Categories* for Recipes, sharding the Post collection based on the `category` field might achieve balanced load distribution among shards. However, this approach was *intentionally avoided* to offer users the flexibility to explore diverse recipes without constraints. In the current implementation, indeed, Users can search for Recipes using *various filters*, discovering Recipes *beyond* fixed categories.

Similarly, the decision not to implement the Sharding for the User collection is justified by the fact that it is realistic to expect that the User collection will not grow as much as the Post collection. For this reason, the complexity of implementing and managing the Sharding for the User collection is deemed unnecessary.

In summary, while sharding can significantly enhance database performance, its appropriateness is strictly tied to the specific demands of the application. In this case, considerations regarding uncorrelated filters, diverse exploration, and distinct growth rates between Users and Posts *influence* the decision not to adopt a sharding approach.

7.1.3. Indexes and Constraints

Considering the different collections stored in MongoDB, the following indexes and constraints have been implemented:

- `Ingredient`:
  - `name`: basic index (ascending order) *and* unique constraint.
- `Post`:
  - `timestamp`: basic index (ascending order);
  - `recipe.totalCalories`: basic index (ascending order).
- `User`:
  - `username`: basic index (ascending order) *and* unique constraint.

Specifically:

```
--> Ingredient:name
```

```
db.Ingredient.createIndex( { "name": 1 }, { unique: true } )
```

Index	nReturned	executionTimeMillis	totalKeysExamined	totalDocsExamined
No	1	4	0	1911
Yes	1	0	1	1

Operation: Find Ingredient by `name`.

Reason: It becomes evident that the inclusion of an index on the `name` field can speed up the find operation. Furthermore, the drawbacks of adding this index are negligible, given the infrequency of writing operations on the Ingredient collection: only the admin has the authority to introduce new ingredients, and the expectation is that such additions will occur rarely. On the other hand, the reading operations like the one that has been analyzed are expected to be frequent among the Users because all the social network is based on Recipes and so on Ingredients. So, adding an index on the `name` field will improve the User experience.

```
--> Post:timestamp
```

```
db.Post.createIndex( { "timestamp": 1 } )
```

Index	nReturned	executionTimeMillis	totalKeysExamined	totalDocsExamined
No	100	109	0	231323
Yes	100	3	100	100

Operation: Find the 100 most recent Posts.

```
--> Post:recipe.totalCalories
```

```
db.Post.createIndex( { "recipe.totalCalories": 1 } )
```

Index	nReturned	executionTimeMillis	totalKeysExamined	totalDocsExamined
No	100	144	0	231323
Yes	100	7	100	100

Operation: Find 100 Posts with totalCalories between `minCalories` and `maxCalories`.

Reasons: Given the substantial volume of the Post collection, it is necessary to define specific indexes, as outlined above. This approach ensures that operations frequently executed by users of *WeFood* yield significant benefits. Since a larger number of users browse posts compared to those creating new posts, the presence of these indexes is justified, despite the potential slowdown in write operations required to maintain them.

It's worth noting a subtle aspect: the decision not to define an index on the `avgStarRanking` field. This choice is thoughtful because this field is updated every time a user rates a post. Introducing an index on `avgStarRanking` would necessitate frequent updates with every rating, resulting in more write operations than the potential benefits conferred by the index.

In contrast, the fields `totalCalories` and `timestamp` remain the same after the initial Post creation. As a result, the indexes defined on them experience updates only once. This distinction is crucial for optimizing the overall performance of the system.

```
--> User:username
```

```
db.User.createIndex( { "username": 1 }, { unique: true } )
```

Index	nReturned	executionTimeMillis	totalKeysExamined	totalDocsExamined
No	1	78	0	27901
Yes	1	2	1	1

Operation: Find User by `username`.

Reason: The inclusion of an index on the `username` field for the Users, as illustrated in the table, proves highly beneficial in significantly reducing the execution time of queries involving `username` lookup. This optimization is particularly valuable during the login phase and when Users or administrators need to access a User Profile. Examining potential drawbacks in this scenario, it becomes evident that the index does not require frequent updates. Specifically, updates are unnecessary after the creation (i.e., sign up) of a new user, as users cannot change their usernames. Similarly, updates are not required when a user decides to delete their account from the platform. This is because the `username` information is retained even after account deletion, preventing other users from signing up with the same `username`.

7.2. Neo4j

The deployment of Neo4j was more straightforward compared to the previous setup. This ease was attributed to the deployment on a *single machine*, whereas setting up a cluster on multiple machines with replicas would have required the enterprise edition of Neo4j.

7.2.1. Indexes

The indexes implemented in Neo4j are:

- `Ingredient`:
  - `name`: text index.
- `Recipe`:
  - `_id`: text index.
- `User`:
  - `username`: text index.

More in detail:

`--> Ingredient:name`

```
CREATE TEXT INDEX ingredient_index FOR (i:Ingredient) ON (i.name);
```

Index	Total DB hits	ms
No	3826	74
Yes	5	5

Operation: Find Ingredient by `name`.

Reason: As for MongoDB, introducing an index on the `name` field of the Ingredients helps to improve the User experience. Indeed, all the suggestions regarding the Ingredients will receive a boost after the introduction of this index.

--> Recipe:_id

```
CREATE TEXT INDEX recipe_index FOR (r:Recipe) ON (r._id);
```

Index	Total DB hits	ms
No	462651	202
Yes	6	14

Operation: Find Recipe by `_id`.

Reason: The introduction of such an index is justified by the fact that this `_id` corresponds to the `_id` of the Post that contains the Recipe that is stored in MongoDB. By doing this, the process of finding a Recipe by `_id` is optimized as it is in MongoDB, where `_id` is by default indexed, being the primary key of the collection. Improvement in the performance of the system will be evident both in the creation phase and in the deletion phase of the Recipes, during which a particular Recipe is searched by `_id` among all the Recipes stored in the graph DB.

--> User:username

```
CREATE TEXT INDEX user_index FOR (u:User) ON (u.username);
```

Index	Total DB hits	ms
No	55808	65
Yes	5	12

Operation: Find User by `username`.

Reason: locating a specific User by `username` is a prerequisite for various features in *WeFood*. These operations, along with those involving *suggestions*, are also connected to the *friendship system* provided to Users. Updating this index is not a relevant issue since Users cannot modify their usernames. The index only needs updates in the rare event of Users deleting their profiles. Considering the infrequency of such occurrences, the introduction of this index is expected to bring about more benefits than drawbacks.

7.3. Consistency, Availability and Partition Tolerance

The trio of properties highlighted in the title represents the fundamental characteristics of a distributed system. However, as the *CAP theorem* states, achieving all three simultaneously in a functional system is deemed impossible. Aligned with the predetermined Non-Functional Requirements (Section **2.2.**), it becomes necessary to prioritize the *Availability* and *Partition Tolerance* of the system, allowing for a measured relaxation in *Consistency constraints*. This strategic approach allows to the primary system actors, the Users, to persist in utilizing the application even if certain displayed information is *not entirely up-to-date*. In practical terms, this might translate to scenarios such as not immediately viewing the latest Recipes (e.g., during a network partition) or encountering non-updated suggestions after the upload of a new Recipe (e.g., if Neo4j is temporarily down). Nevertheless, Users are assured of *eventually* accessing the most recent information. The timeline for this "eventually" remains uncertain, as consistency updates may

employ diverse approaches, and in the event of significant faults, human operator intervention may be necessary.

This is the main idea behind the design of the system. Consequently, the intentional alignment of the design with the intersection of Availability and Partition Tolerance in the CAP theorem places considerable emphasis on achieving eventual consistency. The subsequent paragraphs will delve into a detailed discussion of the system's consistency management, focusing particularly on the *Intra-Database* and *Inter-Database* consistency.

7.4. Intra-Database Consistency

Ensuring consistency within a database is crucial for maintaining the integrity of stored data, particularly in cases of *redundancies*, as evident in the *WeFood* Database Model. MongoDB demands particular attention for the management of redundancy updates, while Neo4j, lacking redundancies among nodes, requires careful consideration only for *inter-database consistency*, that will be discussed later on.

Turning attention to MongoDB collections, a deliberate decision has been made to prioritize updating the Post collection *first*, in scenarios involving multiple operations for redundancy updates. This choice is justified by the need to *consistently update* the Post collection to avoid compromising the User experience, especially if subsequent operations encounter issues. For instance:

- Creation or Deletion of a Post: the Post collection is updated first, followed by the update of redundancy in the User collection;
- Deletion of a User Account: after removing all Posts associated with the User in the Post collection, the User collection is updated by eliminating all the non-personal information.

When ranking a Post, the entire process is contained within the Post collection. Specifically, when a User adds a new vote to a Post, a *practical approximation* is employed to update the `avgStarRanking` field. Instead of re-evaluating all elements in the `starRankings` array, the server follows this sequence:

1. when the User views the post, the server has already transmitted all star rankings;
2. upon receiving the new vote and the Post with all star rankings, the server first adds the new star ranking to the `starRankings` array field in the Post collection;
3. subsequently, the server calculates the updated average star ranking by considering the User's provided copy of the starRankings array, along with the new vote;
4. the `avgStarRanking` field is then directly updated on the Post collection.

While recognizing that this approach may result in a temporarily stale `avgStarRanking` due to the local client copy *and* potential votes occurring after the User viewed the Post, it is understood that this discrepancy is more likely during the initial wave of votes but diminishes over time. As time progresses, the `avgStarRanking` is expected to *converge* to the correct value, ensuring accuracy as the upload time recedes into the past. This same approach applies to the deletion of a vote, with the understanding that the observations regarding the staleness of `avgStarRanking` remain pertinent.

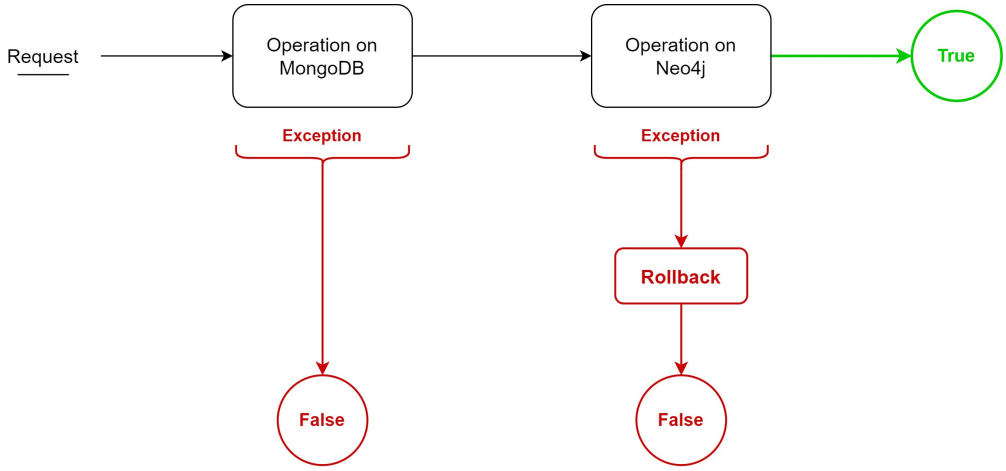

Figure 4: Strict Consistency scheme.

7.5. Inter-Database Consistency

A more complex issue than the previous one arises when it comes to maintaining consistency across different databases. The type of consistency guaranteed depends on the nature of the operation being executed.

1. For *operations necessitating inter-database consistency and performed by actors other than Registered Users* (i.e. the primary actors of the system), **strict consistency** is implemented. This includes:

   - *Registration of an Unregistered User*: prioritizing the verification of the setup's accuracy precedes the User's ability to utilize the application;
   - *Creation of an Ingredient by the Admin*: the emphasis lies on confirming the success of the process rather than providing a swift response.

   In cases of strict consistency, the databases are rolled back to their previous state if an operation fails. The depicted scheme (Figure 4) involves executing the operation first in MongoDB. If unsuccessful, no modifications are made, and a `False` response is returned. Upon success, the same operation is performed in Neo4j, introducing a minor delay for the User. If issues arise, MongoDB is rolled back to maintain consistency between the two databases, resulting in a `False` return. Successful execution yields a `True` return, signifying consistency between the databases.

2. Conversely, when dealing with Registered Users who require high availability and quick responses, even delicate operations demanding inter-database consistency are managed with **eventual consistency**. These operations include:

   - *Uploading or Deleting a Post*;
   - *Deleting a Registered User Account*.

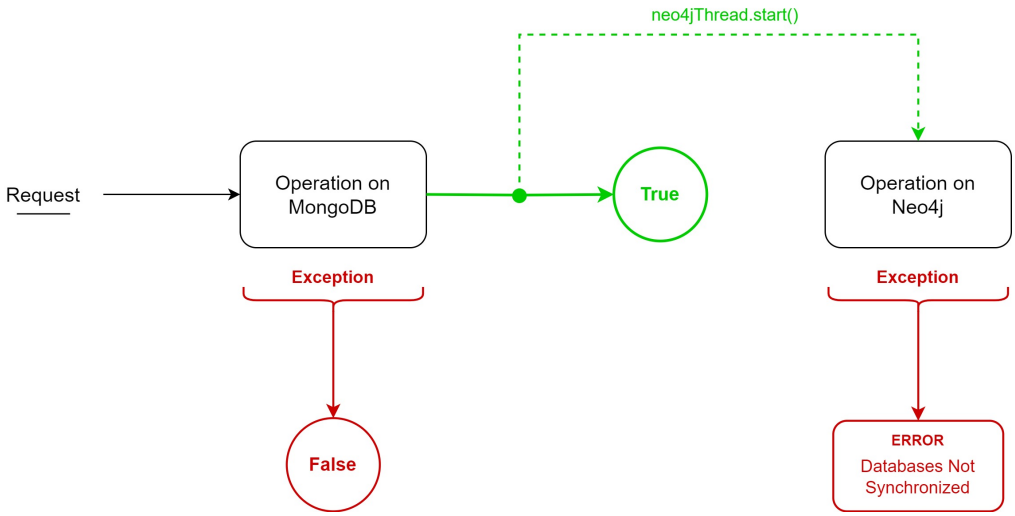

Figure 5: Eventual Consistency scheme.

Eventual consistency (Figure 5) is then employed in these scenarios. The operation begins in MongoDB, and if unsuccessful, no further action is taken before returning `False`. If MongoDB operation succeeds, an immediate `True` response is provided, eliminating the need for waiting. Subsequently, a Thread is initiated to handle consistency asynchronously with Neo4j. Since no critical information is stored in Neo4j, Users can continue using the application even if the two databases are temporarily inconsistent. If the Thread encounters issues, no action is taken, prioritizing system availability over consistency. Log messages or automatic triggers can be scheduled for future consistency recovery. Users can seamlessly use the application, and if Neo4j is down, they can still access features unrelated to Neo4j. Human operators, notified by log messages, are responsible for rectifying the situation if necessary. If both MongoDB and Neo4j operations succeed, no further intervention is required.

---

aggiungere link github e rendere pubblico il repository + inserire credenziali utente e admin in manuale utente

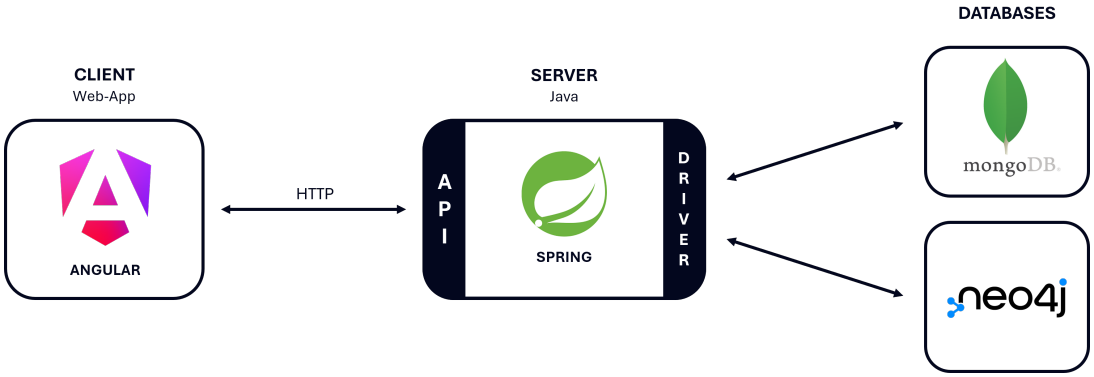

Figure 6: System Architecture.

8. Implementation

The implementation phase represents the last step of the project, as it allows to transform conceptual ideas into a functional software system. This section will begin with a brief overview of the system architecture, followed by a more detailed description of the client and server components.

8.1. System Architecture

The system architecture, depicted in Figure 6, encompasses three main components: *client*, *server*, and *databases*. Together, these components enable the complete functionality of *WeFood*.

The chosen frameworks for application development involve *Angular* [4] for the client and *Spring Boot* [5] for the server. The client, responsible for user interaction, triggers communication with the server through *HTTP* requests utilizing the server's available *APIs*. In response, the server interacts with MongoDB and Neo4j databases using their respective *drivers*.

A fundamental aspect of the adopted communication model is its adherence to *RESTful principles*. REST, or Representational State Transfer, is an architectural style that emphasizes a *stateless* and *standardized* approach to system communication. Additionally, the communication involves encapsulating all necessary information within the body of the HTTP requests, formatted in *JSON*. This RESTful communication approach provides advantages such as scalability, flexibility, and simplicity.

8.2. Client

The client component, constructed using *HTML*, *CSS*, and *TypeScript*, is a *Web Application* and serves as the user interface for interacting with the system. In this context, Angular enables the enhancement of both the development and testing phases. Moreover, it organizes the application into *components*, the building blocks of the user interface, encapsulating specific functionalities. *Services* instead are reusable and injectable objects and facilitate shared functionality (i.e. retrieval or business logic) across the application.

8.3. Server

The server, employing a Model-View-Controller (MVC) architecture, processes client requests and manages data. This structured design ensures modularity, scalability, and maintainability and aids in maintaining a clear *separation of concerns*, enhancing code organization.

Specifically, the server consists of the following packages:

- `repository`: contains the MongoDB and Neo4j drivers (i.e. `base` sub-package), along with the interfaces and classes whose task is to directly communicate with the databases by defining the low-level queries (i.e. `mongodb` and `neo4j` sub-packages);
- `model`: contains the classes that define the structure of the data stored in the databases;
- `dao`: allows to define the high-level queries that will be used by the services hiding the details of the low-level queries;
- `dto`: defines the classes that are used to transfer data between the different layers of the server;
- `service`: contains the classes that define the business logic of the server, including the management of the *intra-database* and *inter-database* consistency;
- `controller`: contains the classes that define the RESTful APIs;
- `apidto`: defines the classes that are used to transfer data between the server and the client.

It is worth examining further the implemented *models* and the role of the *BaseMongoDB* class, which acts as the driver for managing MongoDB queries. The following paragraphs will elaborate on these aspects.

8.3.1. Models

Starting with the conceptual UML Class Diagram (Section **3.2**), adjustments have been made to realize the actual implementation in Java. Outlined below are the details specific to each model.

Admin:

- username: `String`
- password: `String` (hashed)

RegisteredUser:

- _id: `ObjectId`
- username: `String`
- password: `String` (hashed)
- name: `String`
- surname: `String`

Post:

- username: `String`
- description: `String`
- timestamp: `Date`
- comments: `List<Comment>`

- starRankings: `List<StarRanking>`
- recipe: `Recipe`
- avgStarRanking: `Double`

Comment:

- username: `String`
- text: `String`
- timestamp: `Date`

StarRanking:

- username: `String`
- vote: `Double`

Recipe:

- name: `String`
- image: `String`
- steps: `List<String>`
- ingredients: `Map<String, Double>`
- totalCalories: `Double`

Ingredient:

- name: `String`
- calories: `Double`

8.3.2. BaseMongoDB

The `BaseMongoDB` class, located within the `base` sub-package of the `repository` package, serves as the *MongoDB query driver*. This class offers *extra* features beyond the standard MongoDB Java driver. Specifically, this abstract Java class delivers a comprehensive set of functionalities for interacting with MongoDB. It establishes connections and executes queries through a unified interface providing a method (i.e. `executeQuery`) that accepts a query in the *MongoShell format* as its input parameter.

8.3.2.1. Connection Handling

At the core, the class manages a MongoDB connection using the `MongoClient` instance. It employs the *singleton pattern* to ensure that only one instance of `MongoClient` exists. This approach prevents unnecessary multiple connections to the database. The connection details, such as host address and database name, are configured as *static final strings*. In detail, the parameters have been set in the following way:

- `private static final String MONGODB_DATABASE = "WeFood"`: the name of the database;
- `private static final String WRITE_CONCERN = "1"`: the *write concern* is set to `"1"`, indicating that the write operation will be considered successful as soon as the data is written to the primary node in the MongoDB cluster;
- `private static final String WTIMEOUT = "5000"`: the *write timeout* is configured to `5000` milliseconds (i.e. 5 seconds). This setting specifies the maximum amount of time the server

will wait for a write operation to be acknowledged. If the operation is not acknowledged within this timeframe, it will result in a *timeout exception*. This timeout value helps in maintaining a balance between application responsiveness and waiting for database operations, ensuring that the application does not hang indefinitely on slow write operations;

- `private static final String READ_PREFERENCE = "nearest"`: the *read preference* has been set to `"nearest"`, which directs the server to read from the nearest member (either primary or secondary) of the MongoDB cluster based on network latency. This setting is crucial for achieving *low-latency* read operations, as it ensures that the application reads data from the geographically closest node, thereby reducing network latency and improving the overall read performance.

8.3.2.2. Query Execution

The central component for query execution is the method `executeQuery`. It accepts a single input, `mongosh_string`, which holds all the information needed for executing the MongoDB query. Notably, `mongosh_string` encapsulates an executable query directly within the MongoDB shell (i.e. `mongosh`), encompassing details such as the collection name, the intended operation, and associated parameters. The driver's responsibility lies in parsing this string and transforming it into a format compatible with the MongoDB Java driver. This approach ensures a high degree of *flexibility*, allowing the driver to execute a wide range of MongoDB queries, including intricate aggregation pipelines, *without requiring manual translation* of the query into the Java driver format. The string's structure follows this pattern:

```
db.collection.operation({param1: value1, ...})
```

The method initiates by dissecting the `mongosh_string` to extract essential components: the collection name and operation details. The operation name (e.g., `find`, `insertOne`) is isolated, guiding the method to invoke the corresponding *operation-specific* method. The parameters for the operation, which are in JSON format, are extracted from the remaining part of the string. Depending on the operation, additional processing of these parameters may be necessary, achieved through parsing methods, regular expressions, or JSON manipulations.

- `find`: for the `find` operation, the `executeQuery` method isolates the `operationDoc` and proceeds with a tailored execution process. The query parameters are converted into *BSON* format compatible with the MongoDB Java driver. *Criteria*, *projection details*, *sorting instructions*, and *query limits* are extracted and formatted. The `find` method then executes the query against the specified MongoDB collection, collecting and returning the results.
- `aggregate`: handling the `aggregate` operation follows a similar precision pattern but caters to the complexity of MongoDB aggregation. The `operationDoc` for an aggregate operation contains a sequence of MongoDB aggregation pipeline stages. These stages, represented as an array of JSON objects, are parsed into *BSON* objects. Each stage, such as `$match`, `$group`, and `$sort`, is accurately converted from its string representation into *BSON* objects. At this point, the aggregate method executes this pipeline against the specified MongoDB collection, resulting in a list of Document objects representing the final aggregated output.
- `insertOne`: for the `insertOne` operation, the `operationDoc` contains data for the document to be inserted in a JSON-like format. Parsing this data into a Document object, the `insertOne`

method is invoked, producing an `InsertOneResult` indicating the success of the operation, including the `_id` of the inserted document.

- `updateOne`: the `updateOne` operation involves parsing the `operationDoc` into *filter criteria* and *update details*. The filter criteria, responsible for determining the document *target* to be updated, is converted into a Document object, matching the BSON format expected by the MongoDB Java driver. The update portion, instead, specify *how* the document should be updated and potentially can contain various operators such as `$set`, `$unset`, `$push` and `$pull`. This one needs to be parsed into a BSON format. Then, the `updateOne` method executes the update operation, producing an `UpdateResult` indicating the success of the operation and the number of documents updated.
- `deleteOne`: for the deleteOne operation, the `executeQuery` method focuses on the `operationDoc`, containing criteria for selecting the document to be deleted. Parsing this into a Document object, the `deleteOne` method executes the delete operation using the MongoDB Java driver, yielding a `DeleteResult` object with information about the outcome, including the number of documents deleted.

8.3.2.3. Error Handling

The class adheres to a consistent approach for error handling throughout its implementation. Various methods within the class have the potential to throw exceptions, including:

- `MongoException`;
- `IllegalArgumentException`;
- `IllegalStateException`.

These exceptions play a crucial role in signaling failures during database operations, contributing to a robust system of error reporting and handling. The approach relies on calling methods to take charge of error management, ensuring a clear and effective mechanism for handling exceptions.

8.4. Future Enhancements

Upon completing the implementation, specific areas may warrant further exploration. The following points outline potential *future enhancements* to elevate the functionality of *WeFood*:

- *API Security*: the APIs provided by the server are not yet prepared for a real deployment. They lack proper authentication and authorization mechanisms. For a genuine deployment, it's essential to secure sensitive APIs using a JWT (JSON Web Tokens) authentication mechanism. This involves exchanging a token between the client and the server, containing user information. The server verifies the token's validity and checks if the user is authorized to access the requested API. Additionally, *roles* and *authorities* must be defined for different Users, ensuring the right access to specific APIs based on their roles.
- *HTTPS Implementation*: in the current testing deployment, HTTPS has not been implemented for communication between the client and server. This is a critical aspect that must be addressed in a real-world deployment.
- *Neo4j Replicas*: exploring additional replicas of Neo4j could enhance the scalability and fault tolerance of the application. Licensing limitations may be a constraint, but for a fully operational social network, overcoming this obstacle would be essential.

- *Distributed Server*: the current server implementation is not distributed, limiting scalability. In a real-world deployment, distributing the server across multiple nodes could improve scalability and fault tolerance.
- *Feature Expansion*: to broaden the feature set of the social network, introducing additional functionalities may be advisable. This could involve enhancing user interactions, introducing new content types, or refining existing features.
- *Optimizing Response Time*: efforts should be made to further reduce the response time of the web application. This can be achieved by optimizing queries, implementing caching mechanisms, and developing more efficient algorithms for all the functions within the application.
- *Mobile App*: introducing a *mobile application* for *WeFood* would serve as a *valuable addition*, enabling users to access the social network seamlessly from their mobile devices. This transition doesn't necessitate a comprehensive redesign of the client component. Leveraging the current implementation of the web application with Angular provides the flexibility to utilize specific frameworks, like *Ionic*, to *effortlessly* transform the web application into a mobile-friendly format.

In conclusion, the evolutionary process within the deployment of an *authentic social network* extends well beyond the mere implementation phase. The *real-world context* introduces a myriad of aspects that require careful attention and cannot be overlooked.

9. User Manual

Anyone without an account attempting to access *WeFood* will be automatically re-directed to the main feed, displaying the most recent and top-rated posts (Figure X). On this page, two sliders on the left side allow users to customize the time range and the total number of displayed posts. Additionally, a login button is situated in the top right corner. Upon clicking, a pop-up for registration/login will appear. If the user opts for "Sign Up Now," the pop-up will display all the necessary details that an Unregistered User must input to complete the registration process. At the bottom of the pop-up, in the login section(Figure Y), there is a specific button for the admin to enter their login credentials, providing access to their personalized screens. As the admin is already registered, there's no need for them to create a username or password; these details will be communicated to them by the developers through pre-established channels[2].

Upon registration or login[3], users gain access to their personal page and all the functionalities of WeFOod. The initial view for a registered user is a page where they can browse posts, accompanied by left-side buttons to personalize searches using sliders or ingredients (Figure Z). The interface also provides ingredient suggestions based on friends' usage.

In the center of the screen, the user can view all the posts found by their searches. Hovering over posts reveals recipe names, and clicking on them displays a recipe pop-up (Figure P), including the average star ranking at the top right corner. Hovering over the pop-up image reveals the calories of the recipe, and clicking on it allows the user to enter recipe details such as ingredients and steps (Figure A). The comments and votes sections enable user interaction with the post. Users can click on the name of the recipe creator to enter their user page.

On the right side of the screen, a button to upload a post is available, allowing users to add an image, recipe name, ingredients, and more (Figure V).

Additionally, in the top right corner, users have a profile button providing access to the personal profile page. On this page, users find suggestions and statistics (if available) on the left/right side of the screen, along with buttons to find friends/followers/followed and the feed button to return to the feed section.

By clicking on the username in the top left corner, users can modify personal information such as name and surname or delete the personal profile. A confirmation prompt allows users to reconsider before permanently deleting the profile.

When viewing another registered user's page, the current user sees the other user's post in the center of the screen (Figure M). The display includes general statistics about the user, their friends/followers/followed, and a button to follow the user (which turns to "unfollow" after clicking). Users can return to the feed from this screen.

In the admin's personal dashboard, there are statistics presented in either a histogram format or rankings, accompanied by buttons to access the global feed section or find users.

The admin also has the capability to create new ingredients. It's crucial to note that once ingredients are created, there is no option to delete them. Therefore, all steps involved in the ingredient creation

[2]In the current testing environment, the admin credentials are: username: `admin`, password: `password`.

[3]To try the application as a Registered User, instead, it is possible to use the following credentials: username: `user`, password: `user`.

process must be executed with care.

The dashboard additionally provides visibility into all banned users. To unban a registered user, the admin must navigate to the banned user page and click the button to lift the ban.

Clicking on the feed, the admin gains access to posts and buttons to personalize searches, mirroring the functionality available to registered users. Morover the admin can interact with posts, by deleting comments or the post itself if the policies of WeFood are not respected.

10. References

[1] Calories in Food Items (per 100 grams) - https://www.kaggle.com/datasets/kkhandekar/calories-in-food-items-per-100-grams - Accessed: December 2023.

[2] Food.com Recipes and Interactions - https://www.kaggle.com/datasets/shuyangli94/food-com-recipes-and-user-interactions?select=PP_recipes.csv - Accessed: December 2023.

[3] Food.com - Recipes and Reviews - https://www.kaggle.com/datasets/irkaal/foodcom-recipes-and-reviews?select=reviews.csv - Accessed: December 2023.

[4] Angular - https://angular.io/ - Accessed: January 2024.

[5] Spring - https://spring.io/ - Accessed: January 2024.