Index

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.

Requirements

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

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* 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;
   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.

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 the 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.

Design

After having defined the requirements, we can proceed with the design of the system.

Use Case Diagram

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

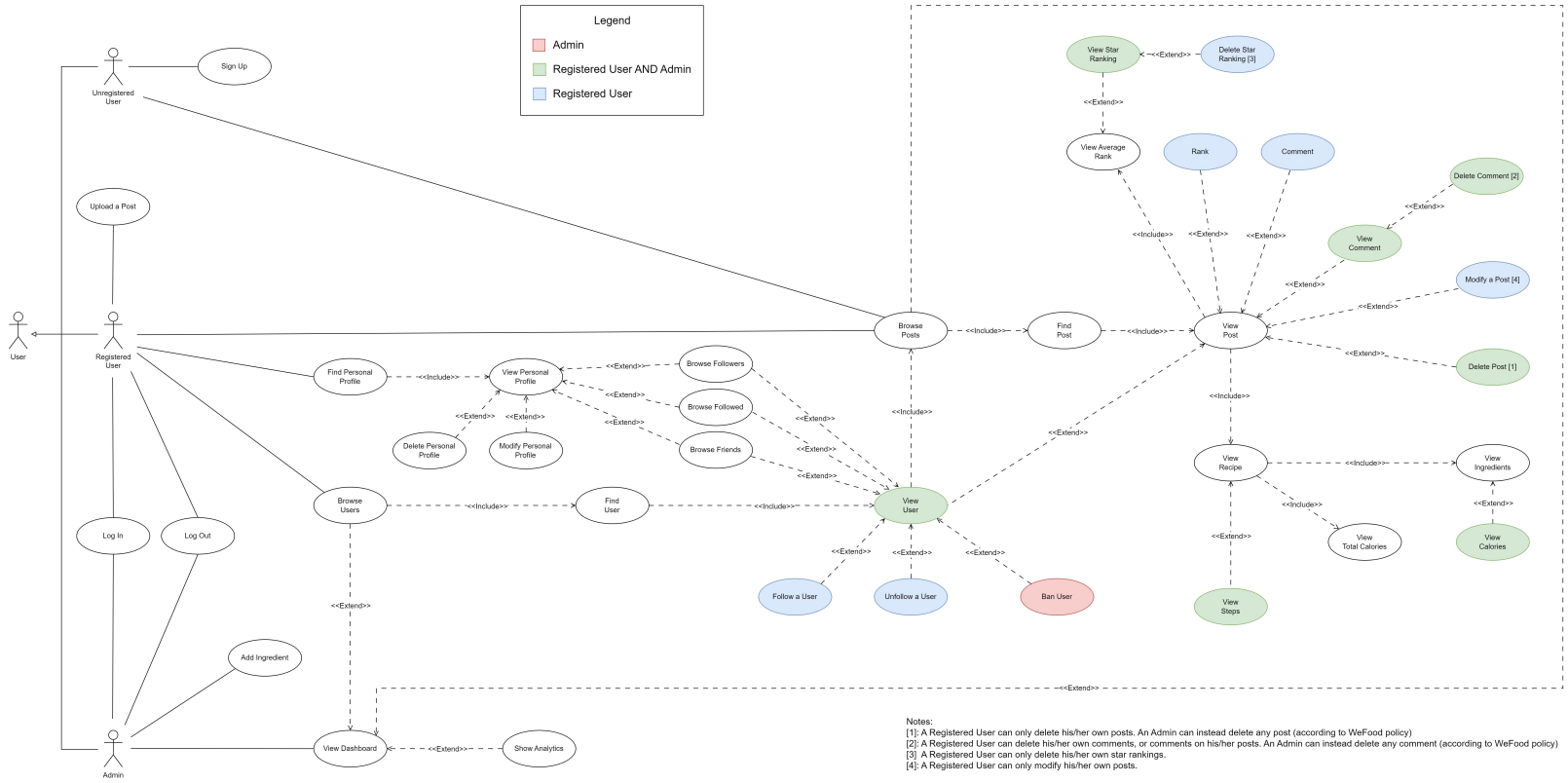

Figure 1: UML Use Case Diagram.

Class Diagram

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

More details on the classes and their attributes are as follows.

Admin:

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

RegisteredUser:

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

Post:

- user: `RegisteredUser`
- description: `String`
- timestamp: `Date`
- comments: `List<Comment>`
- starRankings: `List<StarRanking>`
- recipe: `Recipe`

Comment:

- user: `RegisteredUser`
- text: `String`
- timestamp: `Date`

StarRanking:

- user: `RegisteredUser`
- vote: `Double`

Recipe:

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

Ingredient:

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

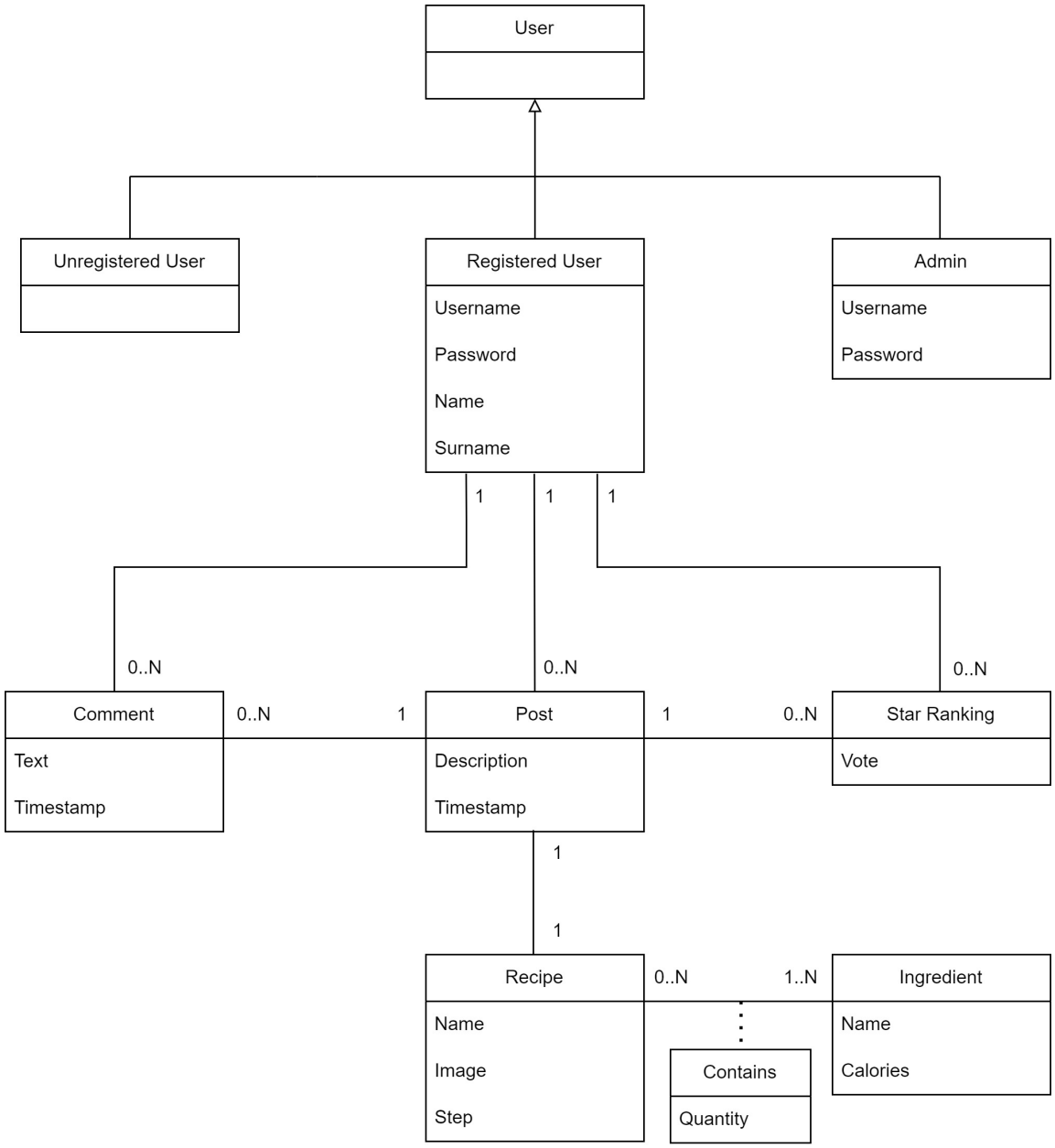

Figure 2: UML Class Diagram.

Dataset

For the population of the databases with a large amount of realistic data, we used datasets from Kaggle.

Raw Dataset

The raw datasets from which we started are related to the main functionalities of *WeFood*. In particular we found datasets about recipes and ingredients.

Source: Calories.info - Calories in Food Items (per 100 grams) https://www.kaggle.com/datasets/kkhandekar/calories-in-food-items-per-100-grams Food.com - Recipes and Interactions https://www.kaggle.com/datasets/shuyangli94/food-com-recipes-and-user-interactions?select=PP_recipes.csv Food.com - Recipes and Reviews https://www.kaggle.com/datasets/irkaal/foodcom-recipes-and-reviews?select=reviews.csv

Description: A group of datasets which contains recipes posted by users with their respective reviews and interactions done by other users. There is also a list of food with some info about calories.

Volume: Around 1 GB. Specifically: 2225 food items; 180K+ recipes and 700K+ recipe reviews; over 500,000 recipes and 1,400,000 reviews.

Variety: Calories.info and Food.com

Cleaning Process

The cleaning process was done using Python and Jupyter Notebook. After having analyzed the datasets in detail we decided to drop the information that was not useful for our purposes.

Merging Process

After having cleand the datasets we started to merge them in order to obtain the information needed for populating the databases.

Here it is worth noticing that the data about the Users, because incomplete in the datasets we found (i.e. it was only provided the username of the users) was generated randomly using the Python library Faker. In this way the name and surname of the users are coherent with their username. The passwords are generated randomly too.

TO BE COMPLETED

Step #1: - Inside the the file called calories.csv we have the following structure: FoodCategory,FoodItem,per100grams,Cals_per100grams,KJ_per100grams - Since quantities are missing in the file RAW_recipes.csv (we saw that in the second dataset, in the file recipes.csv there were numbers related to quantities but these one were really difficutl to understand, because there were no unit of measures and only little numebrs as 1, 2/3 etc. Furthermore no documentation was provided about the meaning of these numbers), we decided to generate meaningful ranges of values for each FoodCategory of the file calories.csv (with the support of ChatGPT). In this way we can associate to each ingriedient of a recipe a random number extracted from a uniform distribution in the range of values of the FoodCategory of that ingredient. In this way we can generate a recipe

with random quantities of ingredients, that are meaningful and realistic (i.e. we can't have 500g of salt in a recipe).

```
Canned Fruit: 100-500
Fruits: 50-300
Tropical & Exotic Fruits: 50-300
Potato Products: 100-500
Vegetables: 50-400
Fast Food: 50-500
Pizza: 100-600
Cheese: 50-200
Cream Cheese: 50-200
Milk & Dairy Products: 100-500
Sliced Cheese: 50-200
Yogurt: 100-250
Beef & Veal: 100-500
Cold Cuts & Lunch Meat: 50-200
Meat: 100-500
Offal & Giblets: 50-200
Pork: 100-500
Poultry & Fowl: 100-500
Sausage: 100-300
Venison & Game: 100-500
Candy & Sweets: 30-100
Ice Cream: 50-200
(Fruit) Juices: 100-300
Alcoholic Drinks & Beverages: 30-100
Beer: 100-300
Non-Alcoholic Drinks & Beverages: 100-300
Soda & Soft Drinks: 100-300
Wine: 50-200
Cereal Products: 50-200
Oatmeal, Muesli & Cereals: 50-200
Pasta & Noodles: 100-400
Dishes & Meals: 100-500
Soups: 100-500
Legumes: 50-300
Nuts & Seeds: 30-100
Oils & Fats: 10-100
Vegetable Oils: 10-100
Baking Ingredients: 5-200
Fish & Seafood: 100-400
Herbs & Spices: 1-10
Pastries, Breads & Rolls: 100-500
Sauces & Dressings: 10-100
Spreads: 10-50
```

- Then we updated the file calories.csv by adding two new columns with the min and max values of the range of values for each FoodCategory. Here's the new structure of the file: FoodCategory,FoodItem,per100grams,Cals_per100grams,KJ_per100grams,MinQuantity,MaxQuantity

```
# First, let's read the contents of the 'quantities.txt' file to understand its
↪  structure
file_path_quantities = 'quantities.txt'

with open(file_path_quantities, 'r') as file:
    quantities_content = file.read()

# Parsing the quantities.txt content to create a dictionary mapping each food
↪  category to its quantity range
quantities_dict = {}

# Splitting the content into lines and then processing each line
for line in quantities_content.split('\n'):
    if line:
        category, quantities = line.split(': ')
        min_quantity, max_quantity = quantities.split('-')
        quantities_dict[category] = (int(min_quantity), int(max_quantity))
import pandas as pd

# Next, let's read the contents of the 'calories.csv' file to understand its
↪  structure
file_path_calories = 'calories.csv'

# Reading the CSV file
calories_df = pd.read_csv(file_path_calories)
# Now, we will add the 'quantity_min' and 'quantity_max' columns to the calories
↪  dataframe based on the food category from the quantities dictionary
# Function to extract quantity range from the dictionary
def get_quantity_range(food_category, quantities_dict):
    return quantities_dict.get(food_category, (None, None))

# Applying the function to each row in the dataframe
calories_df['quantity_min'], calories_df['quantity_max'] =
↪  zip(*calories_df['FoodCategory'].apply(lambda x: get_quantity_range(x,
↪  quantities_dict)))

# The 'FoodCategory' values in the CSV file seem to have no spaces (e.g.,
↪  'CannedFruit' instead of 'Canned Fruit')
# We need to adjust the key matching in the quantities dictionary accordingly
# Adjusting the dictionary keys to match the format in the CSV
adjusted_quantities_dict = {''.join(key.split()): value for key, value in
↪  quantities_dict.items()}
```

```
# Reapplying the function with the adjusted dictionary
calories_df['quantity_min'], calories_df['quantity_max'] =
↪  zip(*calories_df['FoodCategory'].apply(lambda x: get_quantity_range(x,
↪  adjusted_quantities_dict)))

# Specifying the path for the new CSV file
new_csv_file_path = 'updated_calories.csv'

# Saving the dataframe to a CSV file
calories_df.to_csv(new_csv_file_path, index=False)
```

Fare unique su updated_calories.csv su FoodItem (controlla esempio Bacon) Rieseguire il fuzzywuzzy script e vedere se ci sono duplicati su raw_ingr

See Data... (# Load Estimation)

DataBase

Document DB

Entities: - User - Post - Comment - StarRanking - Recipe

Collections: - User - Post

Collections

Structure of the collections:

User { _id: #, type: "Admin", # Applicable only for Admin username: String, [:unique] password: String, name: String, # Not applicable for Admin surname: String, # Not applicable for Admin posts: [{ _idPost: #, name: String, [R] image: String_URL [R] }, ...] }

Post { _id: #, _idUser: #, username: String, [R] description: String, timestamp: Timestamp, recipe: { name: String, image: String_URL, steps: [String, ...], totalCalories: Double, [R] ingredients: [{ name: String, quantity: Double }, ...] }, avgStarRanking: Double, [R] starRankings: [{ _idUser: #, username: String, [R] vote: Double }, ...], comments: [{ _idUser: #, username: String, [R] text: String, timestamp: Timestamp }, ...] }

Ingredient { _id: #, name: String, [:unique] calories: Double }

Graph DB

Entities: - User: - _id (PK of User in Document DB) - username [R]

- Recipe:
  - _id (PK of Post in Document DB)
  - name [R]
- Ingredient:
  - _id (PK of Ingredient in Document DB)
  - name [R]

Relationships: - User –> :FOLLOWS –> User

- User –> :USED –> Ingredient (times: int) [R] Here times keeps track of the fact that the relationship with the ingredient still exists in other recipes after the deletion of a recipe. If times = 0, even then the relationship can be deleted.
- Ingredient –> :USED_WITH –> Ingredient
  (times: int) [R] [This relationship is bidirectional]
- Recipe –> :CONTAINS –> Ingredient

Statistics and Queries

CRUD operations

Create

- Create a new user
- Create a new post / recipe
- Create a new comment
- Create a new star ranking
- Create a new ingredient
- Create a new following relationship
- Create a new recipe-ingredient relationship
- Create a new user-ingredient relationship
- Create a new ingredient-ingredient relationship

MongoDB

- Create a new user

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

- Create a new post

```
db.Post.insertOne({
    idUser: #,
    username: String,
    description: String,
    timestamp: Timestamp,
    recipe: {
                name: String,
                image: String_URL,
                steps: [String, ...],
                totalCalories: Double,
```

```
            ingredients: [{
                            name: String,
                            quantity: Double,
            }, ...]
    }
})
```

- Create a new comment

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

- Create a new star ranking

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

- Create a new ingredient

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

Neo4j

- Create a new user

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

- Create a new recipe

```
CREATE (r:Recipe {
    name: String
})
```

- Create a new ingredient

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

- Create a new following relationship

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

- Create a new recipe-ingredient relationship

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

- Create a new user-ingredient relationship
  - Check if the relationship already exists
  - If it exists, increment the times attribute
  - If it doesn't exist, create the 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
  ```

(Done after the creation of a Recipe) - Create a new ingredient-ingredient relationship - Check if the relationship already exists - If it exists, increment the times attribute - If it doesn't exist, create the 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
```

Dobbiamo farlo in entrambi i versi!

Read

- Show users
- Shows posts
- Show comments of a Post
- Show star ranking of a Post
- Show ingredients
- Show friends
- Show followers
- Show followings
- Show calories of an ingredient
- Show steps of a Recipe

- Show recipe of a Post
- Show ingredients of a Recipe
- Show recipes filtering by the ingredients
- Show recipes filtering by the calories (lower bound and upper bound) (if the totalCalories is the same, we show the ones with the highest star ranking)
- Show the most/least recent posts (timestamp)

MongoDB

- Show users

```
db.User.find()
```

- Show posts

```
db.Post.find()
```

- Show comments

```
db.Post.find({
    _id: #,
}, {
    comments: 1
})
```

- Show star ranking of a Post

```
db.Post.find({
    _id: #,
}, {
    starRankings: 1
})
```

- Show ingredients

```
db.Ingredient.find()
```

- Show calories of an ingredient

```
db.Ingredient.find({
    name: String,
}, {
    calories: 1
})
```

- Show steps of a Recipe

```
db.Post.find({
    _id: #,
}, {
    recipe: {
        steps: 1
```

```
    }
})
```

- Show recipe of a Post

```
db.Post.find({
    _id: #,
}, {
    recipe: {
        name: 1,
        image: 1,
        steps: 1,
        totalCalories: 1,
        ingredients: 1
    }
})
```

- Show ingredients of a Recipe

```
db.Post.find({
    _id: #,
}, {
    recipe: {
        ingredients: 1
    }
})
```

- Show recipes filtering by the calories (lower bound and upper bound) (if the totalCalories is the same, we show the ones with the highest star ranking)

```
db.Post.find({
    "recipe.totalCalories": {
        $gte: lowerBound,
        $lte: upperBound
    }
}).sort({
    avgStarRanking: -1
})
```

- Show the most recent posts (timestamp)

```
db.Post.find().sort({
    timestamp: -1
})
```

- Show the least recent posts (timestamp)

```
db.Post.find().sort({
    timestamp: 1
})
```

Neo4j

- Show followings

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

- Show followers

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

- Show friends

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

- Show recipes filtering by the ingredients

```
MATCH (r:Recipe)-[:CONTAINS]->(i:Ingredient)
WHERE i.name IN [String, ...]   # List of ingredients
RETURN r
```

Update

- Update user's information
  - [username] : we have to mantain the consistency
  - password
  - name
  - surname
- Update post
  - description
- [Update recipe] : we have to mantain the consistency
- Update comment
- [Update user-ingredient relationship] (it is done in the create operation)
- [Update ingredient-ingredient relationship] (it is done in the create operation)

MongoDB

- Update user's information
- Update password

```
db.User.updateOne({
    _id: #,
}, {
    $set: {
        password: hashedString,
```

```
    }
})
```

- Update name

```
db.User.updateOne({
    _id: #,
}, {
    $set: {
        name: String,
    }
})
```

- Update surname

```
db.User.updateOne({
    _id: #,
}, {
    $set: {
        surname: String,
    }
})
```

- Update post
- description

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

- Update comment

```
db.Post.updateOne({
    _id: #,
    comments: {
        $elemMatch: {
            _idUser: #,
            timestamp: Timestamp,
        }
    }
}, {
    $set: {
        "comments.$.text": String,
    }
})
```

Delete

- [Delete user] : we give the possibility to the user to delete his/her own profile.
- Delete post
- Delete comment
- Delete star ranking
- Delete following relationship
- [Delete ingredient] : no because otherwise we would lose all the information about the ingredient (e.g. recipes, etc.)
- Delete user-ingredient relationship (see delete a post)
- Delete ingredient-ingredient relationship (see delete a post)

MongoDB

- Delete user Before deleting a user we have to call delete post for each post of the user But before we can have to delete the recipes inside the posts of the user from Neo4j (calling the delete recipe operation) After this operation we can delete the user from Neo4j

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

Now we can delete evry post of the user from the Post collection (delete post operation)

At this point we can delete all the fields from the user collection, leaving only the username

```
db.User.updateOne({
    _id: #,
}, {
    $unset: {
        password: "",
        name: "",
        surname: "",
        posts: "",
    }
})
```

Now we have to add a field called deleted to the user collection

```
db.User.updateOne({
    _id: #,
}, {
    $set: {
        deleted: true,
    }
})
```

- Delete post Before deleting a post we have to mantain the consistency of the DBs

We need to delete the post from the User collection

```
db.User.updateOne({
    _id: #,
}, {
    $pull: {
        posts: {
            _idPost: #,
        }
    }
})
```

We need to delete all the relationships of the recipe contained in the post from Neo4j

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

We need to update the times attribute of the relationships of the ingredients used together in the recipe contained in the post from Neo4j

```
MATCH (i1:Ingredient {name: String})-[r:USED_WITH]->(i2:Ingredient {name:
↪  String})
SET r.times = r.times - 1
IF r.times = 0 THEN
    DELETE r
END IF
``` [DA PROVARE]

We need to update the relationships among the user and the ingredients used in
↪ the recipe contained in the post from Neo4j
```javascript
MATCH (u:User {username: String})-[r:USED]->(i:Ingredient {name: String})
SET r.times = r.times - 1
IF r.times = 0 THEN
    DELETE r
END IF
``` [DA PROVARE]

Now we can delete the post from the Post collection
```javascript
db.Post.deleteOne({
    _id: #,
})
```

- Delete comment

```
db.Post.updateOne({
    _id: #,
}, {
    $pull: {
```

```
        comments: {
            _idUser: #,
            timestamp: Timestamp,
        }
    }
})
```

- Delete star ranking

```
db.Post.updateOne({
    _id: #,
}, {
    $pull: {
        starRankings: {
            _idUser: #,
        }
    }
})
```

Neo4j

- Delete following relationship

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

Query

Analytics

- Show most active users (da vedere) (DA ELIMINARE)
- Show most followed users
- [Show post with most comments] (Ci serve veramente?)
- Show posts with the highest/lowest star ranking
- Show most/least used ingredients
- Show most/least used ingredients by a user
- Show total amount of calories of a recipe
- Show the average of the avgStarRanking of a User's posts
- Average amount of grams of ingredients used in equal set of ingredients.
- Find recipes filtering the name
- To add others...

MongoDB

- Show posts with the highest star ranking (if the avgStarRanking is the same, we show the most recent one)

```
db.Post.find().sort({
    avgStarRanking: -1,
```

```
    timestamp: -1
})
```

- Show total amount of calories of a recipe

```
db.Post.find({
    _id: #,
}, {
    recipe: {
        totalCalories: 1
    }
})
```

- Show the average of the avgStarRanking of a User's posts

```
db.Post.aggregate([
    {
        $match: {
            _idUser: #,
        }
    },
    {
        $group: {
            _id: null,
            avgOfAvgStarRanking: {
                $avg: "$avgStarRanking"
            }
        }
    }
])
```

- Average amount of grams of ingredients used in equal set of ingredients. (Da implementare)
- Find recipes filtering the name

```
db.Post.find( { "recipe.name": { $regex: "pork", $options: "i" } } )
```

Neo4j

- Show most followed users

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

- Show most used ingredients

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

```
ORDER BY times DESC
LIMIT 5
```

- Show least used ingredients

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

- 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
```

- Show least used ingredients by a user

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

Suggestions

- Suggest users to follow (based on the user's friends)
- Suggest users to follow (based on common ingredients)
- Suggest most popular combination of ingredients
- Suggest new ingredients based on friends' usage
- Suggest most followed users

Neo4j

- Suggest users to follow (based on the user's friends)

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

- Suggest users to follow (based on common ingredients)

```
MATCH (u1:User {username: String})-[r1:USED]->(i:Ingredient)<-[r2:USED]-(u2:User)
WHERE NOT (u1)-[:FOLLOWS]->(u2)
RETURN u2
```

- Suggest most popular combination of ingredients

```
MATCH (i1:Ingredient)-[r:USED_WITH]->(i2:Ingredient)
RETURN i1, i2, r.times AS times
```

```
ORDER BY times DESC
LIMIT 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
```

- Suggest most followed users

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

Aggregations

- (#1)Show the ratio of interactions (number of comments / number of Posts and number of star rankings / number of Posts) and the average of avgStarRanking distinguishing among posts with and without images (i.e. no field image inside recipe)

```
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 he/she has done, the number of star rankings he/she has done and the average of this star rankings

```
db.Post.aggregate([
    {
        $match: {
            $or: [
                {"comments.username": "cody_cisneros_28"},
                {"starRankings.username": "cody_cisneros_28"}
            ]
        }
```

```
    },
    {
        $project: {
            filteredComments: {
                $filter: {
                    input: "$comments",
                    as: "comment",
                    cond: {$eq: ["$$comment.username", "cody_cisneros_28"]}
                }
            },
            filteredStarRankings: {
                $filter: {
                    input: "$starRankings",
                    as: "starRanking",
                    cond: {$eq: ["$$starRanking.username", "cody_cisneros_28"]}
                }
            }
        }
    },
    {
        $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 recipes by name, retrieve the average amount of calories of first 10 recipes ordered by descending avgStarRanking

```
db.Post.aggregate([
```

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

We can use this to show when a post is shown, the average amount of calories of the recipes with similar name

Others...

- Dato utente vediamo calorie medie per ricette pubblicate da lui

```
db.Post.aggregate([ { $match: { username: "cody_cisneros_28" } }, { $project: {
↪  recipeCalories: "$recipe.totalCalories" } }, { $group: { _id: null,
↪  avgCalories: { $avg: "$recipeCalories" } } }] )
```

- Given a User, show among the posts he/she has done, the one with the highest avgStarRanking

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

```
        $sort: {
            avgStarRanking: -1
        }
    },
    {
        $limit: 1
    },
    {
        $project: {
            _id: 0,
            post: {
                _id: "$_id",
                description: "$description",
                timestamp: "$timestamp",
                recipe: "$recipe",
                avgStarRanking: "$avgStarRanking",
                starRankings: "$starRankings",
                comments: "$comments"
            }
        }
    }
])
```

- Average number of TotalCalories for recipes that contains a specific set of ingredients that have an average Star ranking greater than a given value

```
db.Post.aggregate([
    {
        $match: {
            recipe: {
                ingredients: {
                    $elemMatch: {
                        name: {
                            $in: [String, ...]   # List of ingredients
                        }
                    }
                }
            },
            avgStarRanking: {
                $gte: Double
            }
        }
    },
    {
        $group: {
            _id: null,
            avgOfTotalCalories: {
```

```
                $avg: "$recipe.totalCalories"
            }
        }
    },
    {
        $project: {
            _id: 0,
            avgOfTotalCalories: 1
        }
    }
])
```

Redundancies

Table 1: Redundancies introduced into the model.

(1) `DocumentDB:User:posts:name` **Reason**: To avoid joins. **Original/Raw Value**: `DocumentDB:Post:recipe:name`
(2) `DocumentDB:User:posts:image` **Reason**: To avoid joins. **Original/Raw Value**: `DocumentDB:Post:recipe:image`
(3) `DocumentDB:Post:username` **Reason**: To avoid joins. **Original/Raw Value**: `DocumentDB:User:username`
(4) `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.
(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:starRankings:username` **Reason**: To avoid joins. **Original/Raw Value**: `DocumentDB:User:username`
(7) `DocumentDB:Post:comments:username` **Reason**: To avoid joins.

Original/Raw Value: `DocumentDB:User:username`

---

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

---

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

---

(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).

CONSISTENCY, AVAILABILITY AND PARTITION TOLERANCE [X]

Distributed Database Design

According to the non functional requirements expressed before, we should guarantee Availability and Partition tolerance, while consistency constraints can be relaxed. Indeed the application that we are designing is a social network, where the users are the main actors. We orient the design to the AP intersection of the CAP theorem ensuring eventual consistency. Indeed is important to always show some data to the user, even if it is not updated. For example, if a user is not able to see the latest posts of his friends, he will be a little disappointed but at the end it won't be the such a big problem because eventually it will be able to see them.

Replicas

We deployed mongoDB and neo4j with the following configuration: - MongoDB: 3 nodes (1 primary and 2 replicas DA VEDERE SE FARE 3 repliche con stessi poteri) - Neo4j: 1 node (1 primary) (we didn't implement the replicas because we would have needed the enterprise edition)

Read Operations: In WeFood, as in every Social Network, read operations are the most frequent and critical operations. For this reason we have to guarantee the lowest response time possible even if data is not updated to the latest version. So we decided to provide a response from the first available replica.

Write Operations: To ensure the low latency that we discussed before, write operations are considered successful when just a replica node (da sistemare dopo che si è scelta configurazione) wrote the data.

Handling inter database consistency

Using two different databases implemented redundancy of data, so for this reason any fail in insertion/up-date/deletion of data can cause inconsistencies between these to DBs, for this reason, in case of exceptions during write operations on one of the databases causes a rollback. If the operation succeeds on MongoDB, a success response is sent to the user, and the graph db becomes eventually consistent: if an exception occurs after this phase, a rollback operation starts bringing back the DBs in a state of consistency. Check write operations in the Replicas!

Sharding

In our application as it is implemented it is not useful to design the sharding approach. The main reason behind this decision is that we give the users the possibility to find the posts using completely uncorrelated filters that are not linked by a particular relationship (i.e. such as a common field). Indeed if for example we decided to shard the post collection by the timestamp field, we would have latency issues when we have to find the posts using other filters (e.g. by totalCalories). Indeed we would have to query all the shards and then merge the results. This would be a very inefficient approach. For this reason we decided to not implement the sharding approach.

For example if WeFood were implemented by considering a category for each recipe, where the category could be chosen from a fixed set of categories (e.g. geographic area, culture, ..), we could have decided to shard the post collection by the category field of the recipe. In this way we would have reached a good balancing of the load among the shards. But we decided not to proceed in this direction because we wanted to give the users the possibility to explore different recipes without any constraint.

We do not consider the sharding approach for the User collection because we do not expect the users to grow as much as the posts. Indeed we expect that the number of users will be much lower than the number of posts. For this reason we decided to not implement the sharding approach at all.

Configuration of MongoDB (ReplicaSet)

j = false (we do not handle sensitive data and there is no need to wait for the journal to be written to disk) w = 1 (write concern) wtimeout = 0 (handled in Server Java code)

Read Preferences

// Read Preferences at client level MongoClient mongoClient = MongoClients.create(“mongodb: //localhost: 27018, localhost: 27019, localhost: 27020/” + “?readPreference=nearest”);

NEAREST because in this way we can access the node with the lowest latency (Read from any member node from the set of nodes which respond the fastest. (Responsiveness measured in pings))

// Read Preferences at DB Level MongoDatabase db = mongoClient.getDatabase(s: “LSMDB”) .withReadPreference(ReadPreference. secondary ());

// Read Preferences at collection level MongoCollection myColl = db.getCollection(s: “students”) .withReadPreference(ReadPreference. secondary ());

Indexes and Constraints

MongoDB

Possible Indexes:

- Ingredient: name

db.Ingredient.createIndex({ “name”: 1 }, { unique: true })

In this way we also ensure the unique constraint on the name field

Find Ingredient by name

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

We can clearly see that by adding an index on the name field we can speed up the find operation. We do not have any disadvantage in adding this index because the writing operations are not frequent on the Ingredient collection. Because the admin is the only one that can add new ingredients and we do not expect that the admin will add new ingredients very frequently.

- Post: timestamp

Find the 100 most recent posts

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

Reason: we expect to have more read operations than write operations. And furthermore the writing operations are already ordered because posts are published in a chronological order.

- Post: recipe: totalCalories

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

- User: username

db.User.createIndex({ “username”: 1 }, { unique: true })

In this way we also ensure the unique constraint on the username field

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

Reason: since the user is the actor of the social network that performs the most number of operations and most of these operations are find operations (e.g. showing posts with different filters) we decided to implement indexes on the above fields. In this way we can speed up the find operations. We estimate that the number of find operations done by the admin will be negligible compared to the number of find operations done by the users.

try { // Prova ad inserire il documento collection.insertOne(nuovoDocumento); } catch (MongoWriteException e) { // Controlla se è un errore di duplicazione chiave if (e.getError().getCategory() == ErrorCategory.DUPLICATE_KEY) { System.out.println(“Impossibile inserire il documento: il valore è già presente.”); } else { System.out.println(“Errore durante l’inserimento del documento:” + e.getMessage()); } }

Indexes of Neo4j

Da discutere dopo aver caricato tutti i dati ed eventualmente prevedere indice per ricette che sono davvero tante

System Architecture

MVC

Backend

General description

Frameworks and components

Implementation

PERFORMANCE TEST

USER MANUAL

BIBLIOGRAPHY