



# UNIVERSITÀ DI PISA

Large-Scale and Multi-Structure Database

## COOGETHER

Application developed by Cantini Irene, Perrone Federica,  
Tempesti Pietro



# Sommario

<b>Abstract .....</b>	<b>5</b>
<b>Introduction .....</b>	<b>6</b>
<b>Design .....</b>	<b>7</b>
<b>Functional and Non-Functional requirements.....</b>	<b>7</b>
Main Actors .....	7
Functional Requirements .....	7
Non-Functional Requirements .....	7
Product Requirements.....	7
Organizational Requirements .....	8
<b>Dataset .....</b>	<b>8</b>
<b>Use Cases .....</b>	<b>8</b>
<b>Analysis Classes.....</b>	<b>9</b>
<b>Data Model .....</b>	<b>13</b>
MongoDB – Document Organization .....	16
Neo4j – Nodes Organization.....	17
Data among databases .....	17
<b>Implementation .....</b>	<b>19</b>
<b>Main Modules.....</b>	<b>19</b>
<b>Packages Description .....</b>	<b>19</b>
it.unipi.lmmsdb.coogether.coogetherapp.bean .....	19
it.unipi.lmmsdb.coogether.coogetherapp.config.....	19
it.unipi.lmmsdb.coogether.coogetherapp.controller.....	19
it.unipi.lmmsdb.coogether.coogetherapp.persistence .....	20
it.unipi.lmmsdb.coogether.coogetherapp.pojo .....	20
it.unipi.lmmsdb.coogether.coogetherapp.utils.....	20
<b>Constraints.....</b>	<b>20</b>
<b>Most relevant Queries .....</b>	<b>21</b>
CRUD Operations.....	21
MongoDB.....	21
Neo4j .....	26
Analytics Implementations.....	33
MongoDB.....	33
Neo4J .....	39
<b>Cross-Database Consistency Management .....</b>	<b>42</b>
<b>MongoDB – Replica Set .....</b>	<b>44</b>
Replica Configuration .....	44
Replica crash.....	46
<b>Analysis.....</b>	<b>47</b>
<b>MongoDB Queries Analysis .....</b>	<b>47</b>
Index Analysis .....	47
<b>Neo4J Queries Analysis .....</b>	<b>48</b>
Index Analysis .....	49
<b>Sharding proposal .....</b>	<b>50</b>

<b>User Manual .....</b>	<b>51</b>
<b>Home page.....</b>	<b>51</b>
Filters.....	52
Analytics .....	52
<b>Login .....</b>	<b>52</b>
<b>Registration .....</b>	<b>53</b>
<b>Show users.....</b>	<b>53</b>
Filters.....	54
Analytics .....	54
<b>User page.....</b>	<b>54</b>
<b>Add or update recipe .....</b>	<b>55</b>
<b>View recipe .....</b>	<b>56</b>

The code of the application is available at:  
<https://github.com/Fedeperrone98/CooGether>

## Abstract

**CooGether** is an application where users can share their recipes and interact with recipes proposed by other users. In this app, a user can leave a comment and add a review under every recipe: comments should be used as tips/tricks and variations of the original recipe and the owner of the recipe, if a modification proposed is particularly interesting, he can also update the original recipe.

A user can create his network of recipes' uploaders by following other users, in order to see their uploads with a higher priority in the application.

The app has also a ranking system, which allow any user to visualize the users whose recipes have received the best review.

# Introduction



**FIGURE 1 - LOGO**

**CooGether** is the recipe application which makes everyone cook!

We want to become the reference point on the web for those who want to learn how to cook, for those who love to bring traditional dishes to the table or want to amaze with always new ideas.

Behind each recipe there is a user who works with care and passion to share his idea and experiments.

All recipes are designed and tested in your kitchens; they are described clearly and completely with photo and instructions step by step.

With one click you are sure to find the recipe you were looking for: the most authentic version of the great regional classics and ethnic and international dishes, the latest trends on the net, the most original and tasty combinations.

All you must do is get in front of the stove and cook with the millions of users who choose every day to prepare something special for the community!

But it is not all!! We look forward to hearing your opinion about all the dishes you cook!

Welcome to CooGether!

# Design

## Functional and Non-Functional requirements

This section describes the requirements that the application must provide.

### Main Actors

The application interacts with three entities: unregistered users, registered users and administrator.

### Functional Requirements

An **unregistered user** can:

- Browse the recipes.
- Browse comments of a specified recipe.
- Browse the registered users.
- Browse the recipes of a specific user.
- Signup.

A **registered user** can do anything that an unregistered user can do, and:

- Login.
- Logout.
- Add a recipe.
- Add a comment on a specified.
- Update his own recipes.
- Delete his own recipes.
- Follow/unfollow a registered user.
- Unregister himself from the application.

The **administrator** can do anything that a registered user can do, and:

- Delete everyone's recipes.
- Delete users.
- Change the role of another user in Admin/ Normal User.

### Non-Functional Requirements

The **non-functional requirements** of the applications are described in the following list.

#### Product Requirements

- **Usability:** the application must be easy to use and intuitive.
- **Efficiency:** the application must have low response time in navigation.
- **Reliability:** the application must handle data inconsistency and wrong user inputs.
- **Data Consistency:** the application must handle the data consistency across multiple data sources.
- **Availability:** the application must be able to response to any query.
- **Partition tolerance:** failures of individual nodes or connections between nodes do not affect the system as a whole.

## Organizational Requirements

- If an user delete his own account, also his recipes are deleted. For the comments, instead, we decide to not delete them.

## Dataset

In order to simulate a large-scale database, we collect lots of information thanks to open source resources.

The most important information that we need are the recipes and their comments, on <https://www.kaggle.com/irkaal/foodcom-recipes-and-reviews> we found an almost complete dataset, scraped from <https://www.food.com/>. In order to ensure the variety property, we found a second dataset on

[https://www.kaggle.com/hugodarwood/epirecipes?select=full\\_format\\_recipes.json](https://www.kaggle.com/hugodarwood/epirecipes?select=full_format_recipes.json), scraped from [www.epicurious.com](http://www.epicurious.com).

We perform some modifications on these datasets, like remove duplicates and useless information, in order to obtain all right information necessary for a correct usage of the application.

We randomly generate users on <https://randomuser.me>.

At the beginning, we started with a dataset containing: 20.000 recipes (50 MB), 170.000 comments (70 MB), 65.000 users (10 MB).

## Use Cases

The **use case diagram** of the application is the following picture (Figure 2):





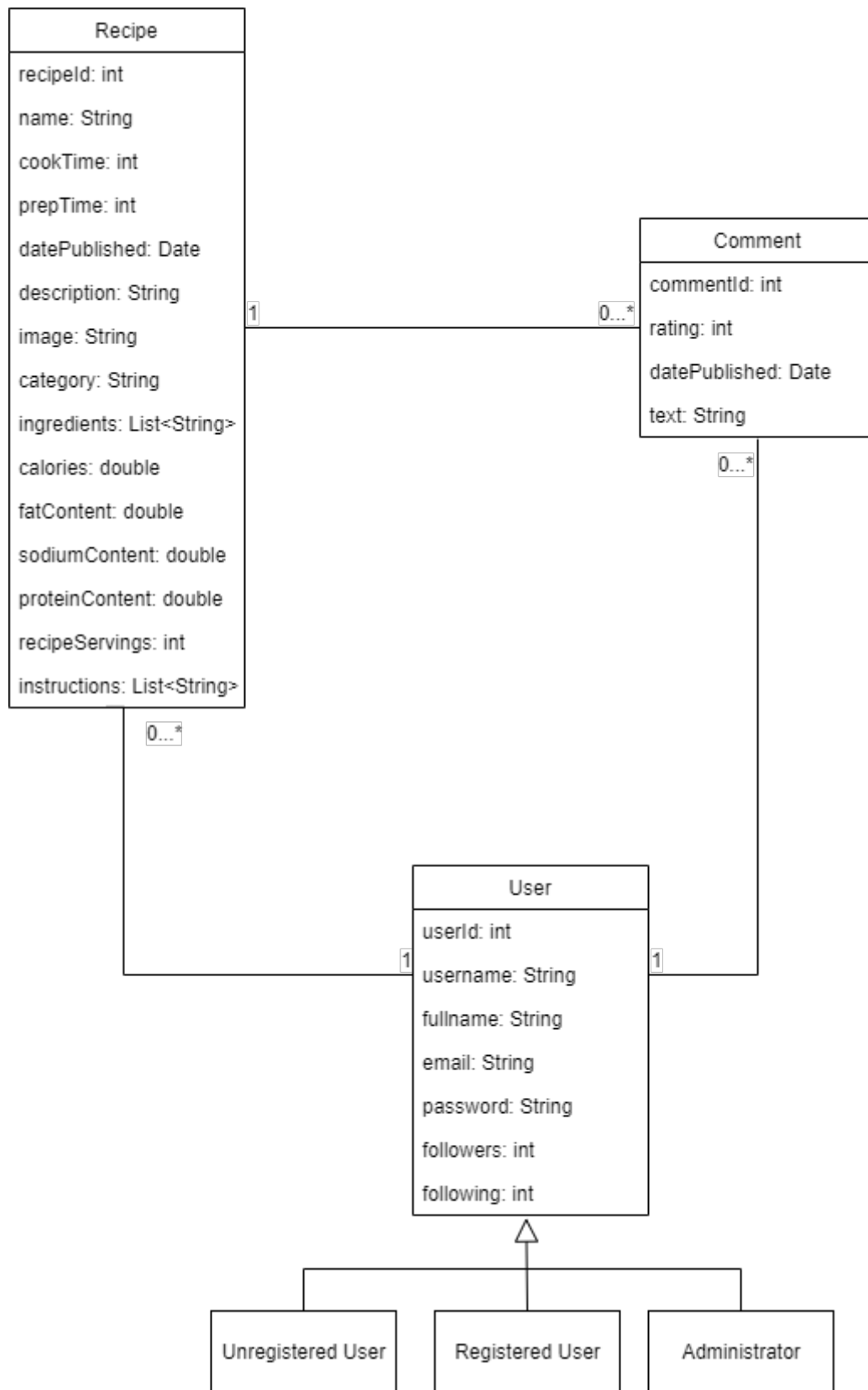


FIGURE 3 - UML CLASS DIAGRAM (WITH GENERALIZATION)

We solve the generalization putting an attribute in the entity User (Figure 4). It is an integer and we call it **role**: if the user is a registered one, role is 0; if, instead, it is the administrator, role is 1. If the user is an unregistered one, we do not save any information about it.

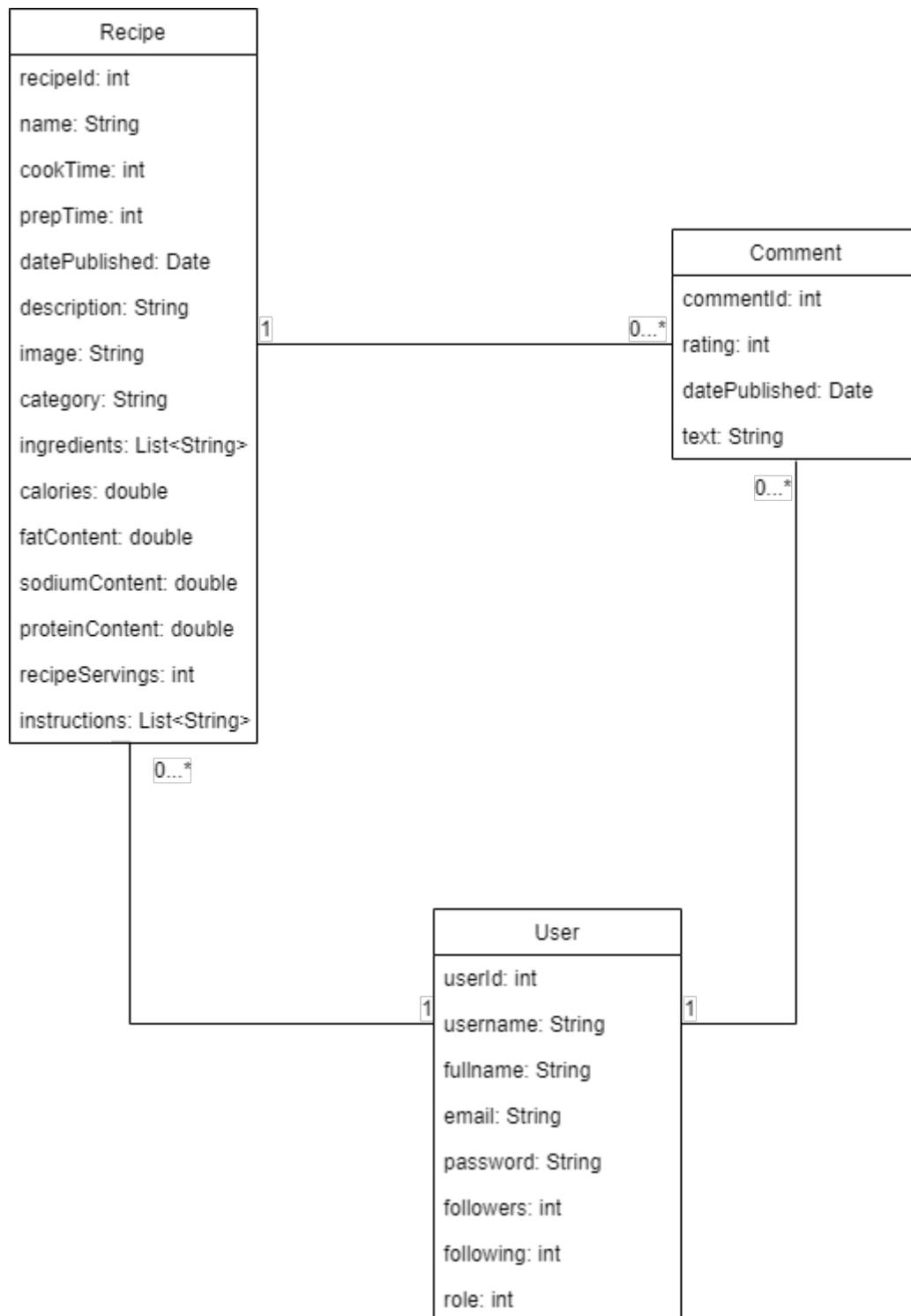


FIGURE 4 - UML CLASS DIAGRAM

- Each user can post zero or more recipes.
- Each user can write zero or more comments related to a recipe.
- Each recipe can have zero or more comments.
- Each recipe is posted by one user.

- Each comment is referred to a recipe.
- Each comment is written by a user.

Let analyse the attributes for each class in the following tables.

Recipe class		
Attribute name	Type	Description
recipeld	int	Unique Id of the recipe
name	String	Title of the recipe
authorId	int	Unique Id of the user that posts the recipe
authorName	String	Username of the user that posts the recipe
cookTime	int	Cooking time of the recipe
prepTime	int	Preparation time of the recipe
datePublished	Date	Publication date of the recipe
description	String	Description of the recipe
image	String	URL of the recipe picture
category	String	Category of the recipe
ingredients	List<String>	Ingredients to be used in the recipe
comments	List<Comments>	Comments that refers to the recipe
calories	double	Calories contained in the recipe
fatContent	double	Fat contained in the recipe
sodiumContent	double	Sodium contained in the recipe
proteinContent	double	Protein contained in the recipe
recipeServings	int	Portions of the recipe
instructions	List<String>	Instruction to prepare the recipe

Comment class		
Attribute name	Type	Description
commentId	int	Unique Id of the comment
authorId	int	Unique Id of the user that writes the comment
authorName	String	Username of the user that writes the comment
rating	int	Number of stars (from 1 to 5) that the user give to the recipe
datePublished	Date	Publication date of the comment
text	String	Text of the comment

User class		
Attribute name	Type	Description
userId	int	Unique Id of the user
username	String	Username of the user
fullName	String	Full name of the user
email	String	Email of the user
password	String	Password chosen by the user, used for the login phase
followers	int	Number of users followed by the user
following	int	Number of users following the user
role	int	Role of the user (0: Registered User, 1: Administrator )

## Data Model

In this section is present a description of the documents and nodes that are store in the database.

In order to find a perfect solution for our application, we perform a preliminary analysis to decide how to store the comments for each recipe.

We compared two possible solutions:

In the first one, we stored the comments in both MongoDB and Neo4j.

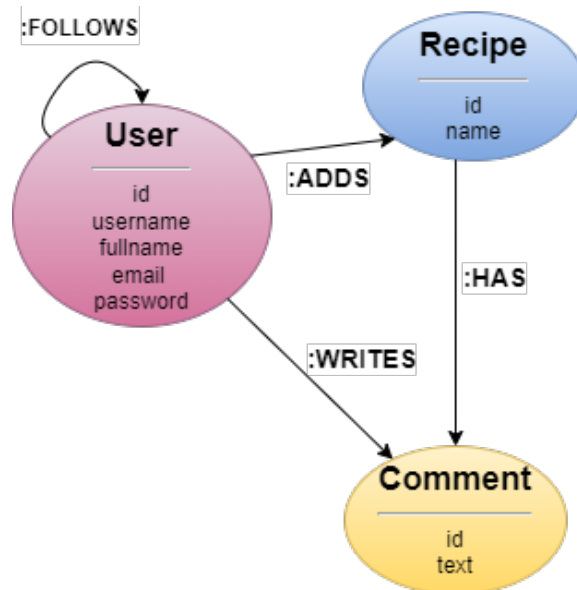
- In MongoDB, we decided to store an array of comments' documents for each recipe, containing only a few fields, so as not to exceed the maximum size allowed for each document (16MB). The fields stored in the array of comments' documents for each recipe were the comment id, the author id, the author name, the rating, the publication date and the modification date.
- In Neo4j, we had the node type Comment, which had as its property: the comment id and the text.

This is an example of recipe document (Figure 5):

```
1 {
2   "recipeId":42,
3   "name":"Cabbage Soup",
4   "authorId":1538,
5   "authorName":"Duckie067",
6   "cookTime":1800,
7   "prepTime":1200,
8   "datePublished":937714740000,
9   "description":"Make and share this Cabbage Soup recipe from Food.com.",
10  "image":"https://img.sndimg.com/food/image/upload/w_555,h_416,c_fit,fl_progressive,q_95/v1/img/recipes/42/picVEMxk8.jpg",
11  "recipeCategory":"Vegetable",
12  "ingredients":[{"name: plain tomato juice, qty: 46",
13                "name: cabbage, qty: 4",
14                "name: onion, qty: 1",
15                "name: carrots, qty: 2",
16                "name: celery, qty: 1"}],
17  "comments":[{"reviewId":46368,"authorId":71084,"rating":5,"dateModified":1048588311000,"authorName":"smurfy57","datePublished":1048588311000},
18              {"reviewId":118530,"authorId":158091,"rating":0,"dateModified":1092745950000,"authorName":"Sassyface ohagan ", "datePublished":1092745950000},
19              {"reviewId":165407,"authorId":208121,"rating":3,"dateModified":1112874043000,"authorName":"drhousespcatcher", "datePublished":1112874043000},
20              {"reviewId":177583,"authorId":151591,"rating":2,"dateModified":1117400309000,"authorName":"Annie M.", "datePublished":1117400309000},
21              {"reviewId":185238,"authorId":226656,"rating":0,"dateModified":1120316367000,"authorName":"lilblondie10169113", "datePublished":1120316367000},
22              {"reviewId":186959,"authorId":215395,"rating":4,"dateModified":1120985178000,"authorName":"skitchen kitchen", "datePublished":1120985178000},
23              {"reviewId":804233,"authorId":1060485,"rating":5,"dateModified":1233862142000,"authorName":"allyop135", "datePublished":1233862142000},
24              {"reviewId":853747,"authorId":465080,"rating":0,"dateModified":1240241812000,"authorName":"dogsandwoods", "datePublished":1240241812000},
25              {"reviewId":863050,"authorId":99678,"rating":5,"dateModified":1241561464000,"authorName":"Pollywog", "datePublished":1241561464000}],
26  "calories":103.6,
27  "fatContent":0.4,
28  "sodiumContent":959.3,
29  "proteinContent":4.3,
30  "recipeServings":4,
31  "recipeInstructions":["Mix everything together and bring to a boil.",
32                        "Reduce heat and simmer for 30 minutes (longer if you prefer your veggies to be soft).",
33                        "Refrigerate until cool.", "Serve chilled with sour cream."]
```

FIGURE 5 - ORGANIZATION OF RECIPE DOCUMENT

The GraphDB's architecture is shown below (Figure 6):



**FIGURE 6 - GRAPHDB ARCHITECTURE**

In the second solution, instead, we stored the comments only in MongoDB, so we eliminated the Comment nodes from Neo4j and we integrated the text of the comments in a field of the array of document.

The analysis was done in two directions:

1. **Document Size:** find the comment with the largest size, and the recipe document with the maximum number of comments; then calculate the maximum size of a document (assuming that it has the maximum number of comments and that each of them has the maximum size).
  1. The recipe with the maximum number of comments has 2182 comments.
  2. The maximum comment's size is 4438 bytes.
  3. The maximum recipe's size is:
 
$$9577\text{bytes} + 4438\text{bytes} * 2182\text{comments} = 9693293\text{bytes} \approx 10\text{MB}$$
 Where 9577 is the maximum recipe size without comments.

We found that the maximum size of a recipe document in the worst case is smaller than 16MB with a good margin (6MB).

2. **Efficiency:** we evaluated the time needed to perform some CRUD operations, in particular those concerning access to both databases, such as getting the comments of a specific recipe. We analysed the recipe that has the maximum number of comments (2182).

In the first solution, to get all comments of this specific recipe, we need access to both MongoDB and Neo4j database, because the comments' information is distributed among both databases. The total execution time was (Figure 7, Figure 8):

$$90ms + 28ms = 118ms.$$



FIGURE 7 - QUERY EXECUTION TIME IN NEO4J

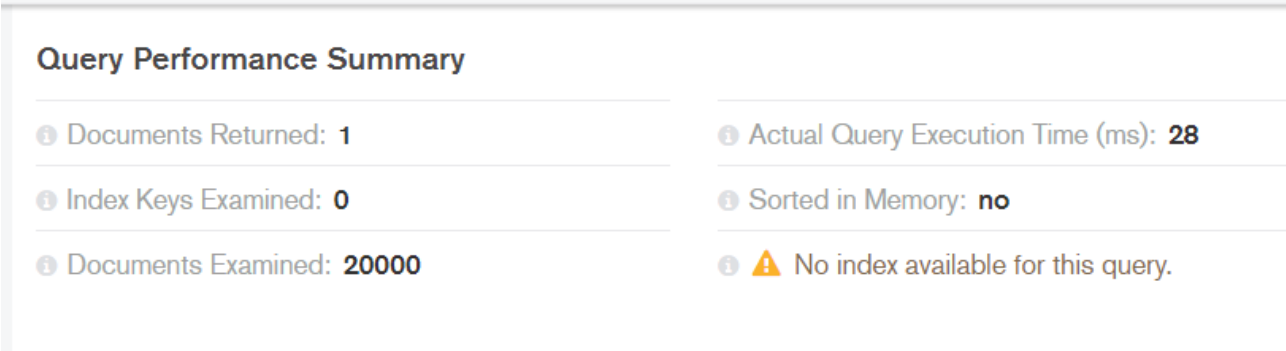


FIGURE 8 - QUERY EXECUTION TIME IN MONGODB

- In the second solution, we need to access only in MongoDB to get all comments of a specific recipe. The execution time was 35ms (Figure 9):

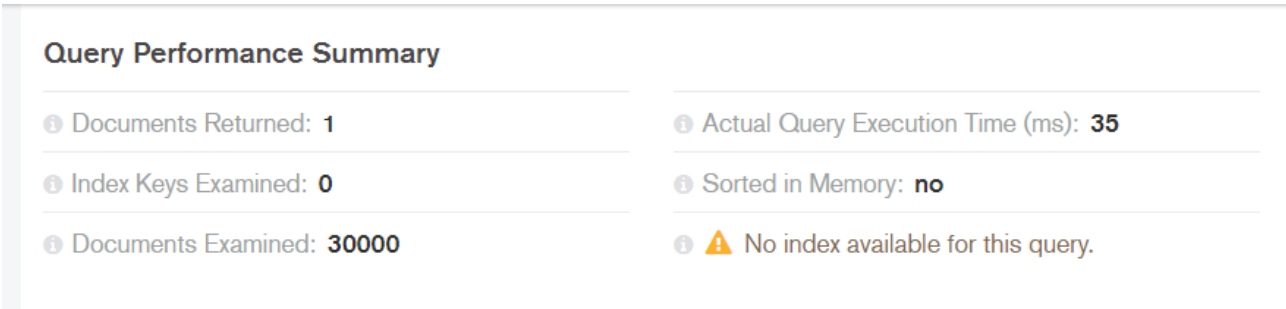


FIGURE 9 - QUERY EXECUTION TIME IN MONGODB

With this analysis we decided to maintain the second solution because the maximum document's size allowed by MongoDB is not a problem and is better in term of performance.

In the following paragraphs, we present the solution adopted.

## MongoDB – Document Organization

In MongoDB we are stored two collections:

1. **Recipe Collection:** 30.000 recipes.
2. **Utility Collection:** 1 document.

The Recipe Collection is organized in this way (Figure 10):

```
_id: ObjectId("61d56ce2de2165adfe69e970")
recipeId: 52
name: "Cafe Cappuccino"
authorId: 2178
authorName: "troyh"
cookTime: 0
prepTime: 300
datePublished: 936126300000
description: "Make and share this Cafe Cappuccino recipe from Food.com."
image: "https://img.sndimg.com/food/image/upload/w_555,h_416,c_fit,fl_progress..."
recipeCategory: "Beverages"
✓ ingredients: Array
  0: "1/2 instant coffee"
  1: " 3/4 sugar"
  2: " 1 nonfat dry milk solid"
✓ comments: Array
  ✓ 0: Object
    reviewId: 116178
    authorId: 133174
    rating: 5
    authorName: "PaulaG"
    comment: "I haven't made this in years. When the kids were small, I would make ..."
    datePublished: 1091466808000
  calories: 62.2
  fatContent: 0.1
  sodiumContent: 36.6
  proteinContent: 2.7
✓ recipeInstructions: Array
  0: "Stir ingredients together."
  1: "Process in a blender until powdered."
  2: "Use 2 Tbsp. of mixture for each cup of hot water."
```

FIGURE 10 - RECIPE COLLECTION

The Utility Collection is organized in this way (Figure 11):

```
_id: ObjectId("61ed9e399337900b1ea673dc")
name: "maxID"
recipe: 86410
comment: 2090356
user: 2002901635
```

FIGURE 11 - UTILITY COLLECTION



In this collection we saved the information concerning the maximum id of some fields to not insert fields with duplicate id.

### Neo4j – Nodes Organization

The structure of the nodes with their relations is described below (Figure 12):

1. The **Users** node, representing the users registered within the application, having as properties the id, the username, the first name, the last name, the email and the password.
2. The **Recipes** node, representing the recipes stored within the application with a small subset of properties: id, name, category and publication date.

Relationships:

1. **User → FOLLOWS → User**, which represents a user following another in the application, and it is created when a User starts to follow another User. This relationship has no properties.
2. **User → ADDS → Recipe**, which represents a recipe added by a user within the application, and it is created when a User posts a recipe. This relationship has no properties.

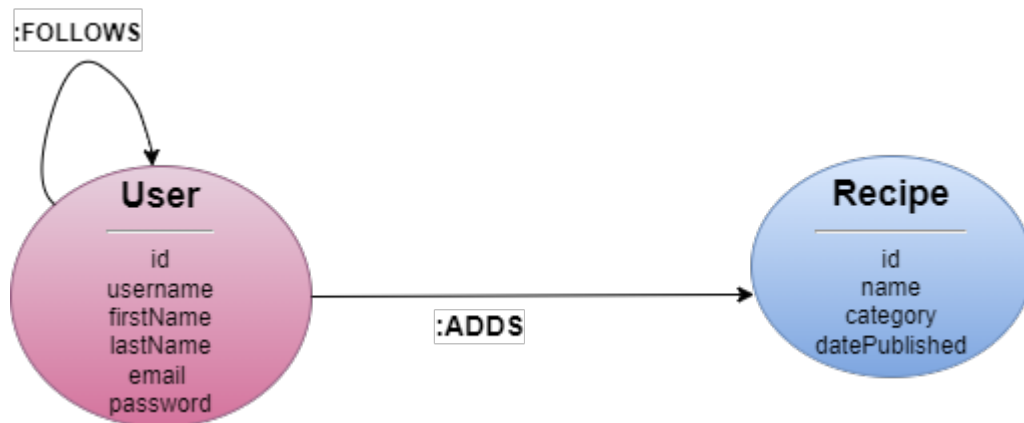


FIGURE 12 - GRAPHDB ORGANIZATION

### Data among databases

Due to the large amount of information to deal with, it is important to split up them in a way that allow us to handle them in the fastest and easiest way.

Summing up the concepts exploited so far, we have the following storing strategy:

- User: stored only on GraphDB.
- Recipe: stored on DocumentDB and partially on GraphDB.
- Comment: stored only on DocumentDB.

The reason why we store recipes on both databases is because we want to retrieve social network information from it (like suggested recipes). Hence, we store on Graph DB a part of the recipe information, and the whole information about recipes on DocumentDB. When a recipe list is loaded, only a partial view of them is shown and it is retrieved from GraphDB. When the user clicks on this partial view, the all information of the recipe is loaded from the DocumentDB.

We can see the storing strategy also in the following picture (Figure 13):

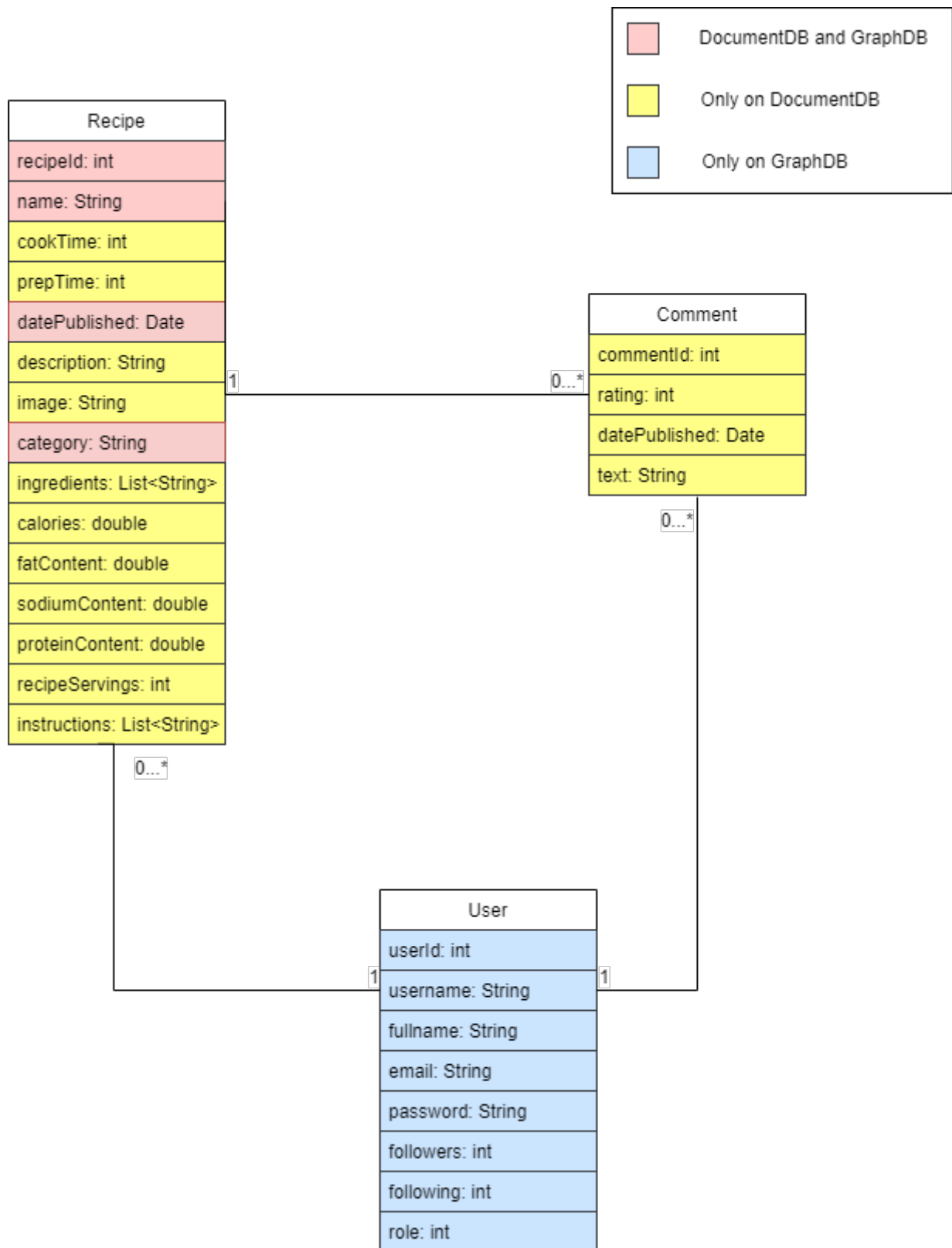


FIGURE 13 - STORING STRATEGY

# Implementation

## Main Modules

The implementation code is divided into two main modules: CoogetherApp and JsonLoader.

- JsonLoader: is a program we wrote to get our initial dataset, extracting the information that interests us from the two initial datasets (see chapter 1) and to load our dataset into the remote virtual machines.
- CoogetherApp: is the actual application, whose implementation will be analysed in more details in the next sections.

## Packages Description

In this section will be presented the main packages of CooGether module and the respective classes.

### [it.unipi.lmmsdb.coogether.coogetherapp.bean](#)

This package contains the classes required for the java bean.

Classes:

- Comment: This class stores all the information about a comment, like the text, the username of the author, the grade given to the recipe and the timestamp of creation.
- Recipe: This class stores all the information about a recipe, like the title, the ingredients, and so on.
- User: This class stores all the information about a user, like the username, the password, and so on.

### [it.unipi.lmmsdb.coogether.coogetherapp.config](#)

This package is used to handle the configuration parameters, stored in config.xml. The schema for the validation is in the file config.xsd. The validation is very important to be sure of the correctness of the file config.xml.

Classes:

- ConfigurationParameters: this class stores all the configuration parameters needed by the application. For example the IP for the Neo4j database. These values do not need to be changed, so only get methods are provided.
- SessionUtils: This class is used to maintain the information of the session, like the logged user.

### [it.unipi.lmmsdb.coogether.coogetherapp.controller](#)

This package contains the classes required for the controller part of the MVC pattern. For each different page to be shown to the user, a special controller has been implemented, which manages the events resulting from the actions taken by the user and updates the model and the view.

Classes:

- AddRecipeController: this class manages the page of the application used for insert a new recipe.
- FollowersViewController: this class manages the snapshot of the follower users, and all the operations that can be done on him.
- FollowingViewController: this class manages the snapshot of the following users, and all the operations that can be done on him.

- HelloController: this class handles the homepage section of the application (shows the recipes order by publication date and handles the event, like the click on a recipe snapshot).
- LoginViewController: this class manages the login page of the application.
- RecipeViewController: this class handles the page in which we show all the information about a recipe. In this page it is possible also to comment a recipe and see the comments already done.
- RegistrationViewController: this class manages the register page of the application.
- UserDetailsViewController: this class manages the profile section of the application. Thanks to this class it is possible to manage some events such as deleting my profile, change password, and so on.
- UsersViewController: this class manages the snapshot of the users, and all the operations that can be done on him.

### [it.unipi.lmmsdb.coogether.coogetherapp.persistence](#)

This package deals with managing the persistence of data, in fact it contains the classes used to interface with databases.

Classes:

- MongoDBDriver: this class implements DatabaseDriver and is responsible for implementing all the queries that must be run on MongoDB.
- Neo4jDriver: this class implements DatabaseDriver and is responsible for implementing all the queries that must be run on Neo4j.

### [it.unipi.lmmsdb.coogether.coogetherapp.pojo](#)

This package contains the classes used to do a perfect mapping of the json file.

Classes:

- CommentPojo: class used to map the comments in a recipe.
- DatePojo: class used to map the different date formats of the two data sources.
- RecipePojo: class used to map the recipes stored in mongoDB.

### [it.unipi.lmmsdb.coogether.coogetherapp.utils](#)

This package contains a class used to store all the utility functions that we use in the application.

Classes:

- Utils: this class is used for containing some utility functions used inside the application (to avoid code replication).

## Constraints

Constraints have been added in the two databases, for the recipe Id, for the user Id, for the user's username, and user's email, which must be unique and always present.

- Constraint on the recipe id (on MongoDB):

```
1. db.recipe.createIndex (
2.   {
3.     recipeId :1
4.   },
5.   {
6.     unique : true ,
7.     name : " recipeId_constraint "
```

```
8.     }
9. )
```

- Constraint on the recipe id (on Neo4j):

```
1. CREATE CONSTRAINT recId_constraint IF NOT EXISTS
2. FOR (r:Recipe)
3. REQUIRE r.id IS UNIQUE
```

- Constraint on the user id (on Neo4j):

```
1. CREATE INDEX uid_index FOR (u:User) ON u.id
```

- Constraint on the username (on Neo4j):

```
1. CREATE INDEX username_index FOR (u:User) ON u.username
```

- Constraint on the email (on Neo4j):

```
1. CREATE CONSTRAINT email_constraint IF NOT EXISTS
2. FOR (u:User)
3. REQUIRE u.email IS UNIQUE
```

## Most relevant Queries

In the following section will be presented the most relevant queries performed with MongoDB and Neo4j.

### CRUD Operations

#### MongoDB

##### ADD RECIPE

This query gives the opportunity to insert a new recipe in the application with all the required information.

- Input: an object with all the recipe's information.
- Output: a new document.

#### JAVA LANGUAGE

```
1. public static boolean addRecipe(Recipe r){
2.     openConnection();
3.
4.     try{
5.
6.         Document doc= new Document();
7.         doc.append("recipeId", r.getRecipeId());
8.         doc.append("name", r.getName());
9.         doc.append("authorId", r.getAuthorId());
10.        doc.append("authorName", r.getAuthorName());
11.        if(r.getCookTime() != -1)
12.            doc.append("cookTime", r.getCookTime());
13.        if(r.getPrepTime() != -1)
```

```

14.         doc.append("prepTime", r.getPrepTime());
15.         doc.append("datePublished", r.getDatePublished());
16.         doc.append("description", r.getDescription());
17.         doc.append("image", r.getImage());
18.         doc.append("recipeCategory", r.getCategory());
19.         doc.append("ingredients", r.getIngredients());
20.         ArrayList<Document> commentsToAdd = new ArrayList<>();
21.         for(Comment c: r.getComments()){
22.             Document d = new Document("commentId", c.getCommentId())
23.                 .append("authorId", c.getAuthorId())
24.                 .append("rating", c.getRating())
25.                 .append("authorName", c.getAuthorName())
26.                 .append("comment", c.getText())
27.                 .append("dateSubmitted", c.getDatePublished());
28.             commentsToAdd.add(d);
29.         }
30.         doc.append("comments", commentsToAdd);
31.         if(r.getCalories()!=-1)
32.             doc.append("calories", r.getCalories());
33.         if(r.getFatContent()!=-1)
34.             doc.append("fatContent", r.getFatContent());
35.         if(r.getSodiumContent()!=-1)
36.             doc.append("sodiumContent", r.getSodiumContent());
37.         if(r.getProteinContent()!=-1)
38.             doc.append("proteinContent", r.getProteinContent());
39.         if(r.getRecipeServings()!=-1)
40.             doc.append("recipeServings", r.getRecipeServings());
41.         doc.append("recipeInstructions", r.getRecipeInstructions());
42.
43.         collection.insertOne(doc);
44.
45.     }catch(Exception ex){
46.         closeConnection();
47.         return false;
48.     }
49.     closeConnection();
50.     return true;
51. }

```

## UPDATE RECIPE

This query gives the opportunity to change some information about a recipe.

- Input: an object with all the recipe's information (old and new information).
- Output: a new document with the modified information.

## JAVA LANGUAGE

```

1.  public static boolean updateRecipe(Recipe r){
2.      openConnection();
3.      try{
4.          boolean res=deleteRecipe(r);
5.          if(!res)
6.          {
7.              System.out.println("A problem has occurred in modify recipe");
8.              return false;
9.          }
10.
11.          res= addRecipe(r);
12.          if(!res)
13.          {
14.              System.out.println("A problem has occurred in modify recipe");
15.              return false;
16.          }
17.
18.      }catch(Exception ex){

```

```

19.         closeConnection();
20.         return false;
21.     }
22.     closeConnection();
23.     return true;
24. }

```

## DELETE RECIPE

This query gives the opportunity to eliminate a recipe from the application.

- Input: an object with all the recipe's information.

## JAVA LANGUAGE

```

1.  public static boolean deleteRecipe(Recipe r){
2.      openConnection();
3.      try{
4.          collection.deleteOne(Filters.eq("recipeId", r.getRecipeId()));
5.      }catch(Exception ex){
6.          closeConnection();
7.          return false;
8.      }
9.      closeConnection();
10.     return true;
11. }

```

## ADD COMMENT

This query gives the opportunity to insert a new comment in the application.

- Input: an object containing the recipe to insert and an object containing the comment information.

## JAVA LANGUAGE

```

1.  public static boolean addComment(Recipe r, Comment c){
2.      openConnection();
3.      System.out.println(r.getRecipeId());
4.      try{
5.          Document com = new Document("commentId", c.getCommentId())
6.              .append("authorId", c.getAuthorId())
7.              .append("rating", c.getRating())
8.              .append("authorName", c.getAuthorName())
9.              .append("comment", c.getText())
10.             .append("dateSubmitted", c.getDatePublished());
11.
12.             Bson filter = Filters.eq( "recipeId", r.getRecipeId() ); //get the parent-
document
13.             Bson setUpdate;
14.             if(r.getComments() != null && r.getComments().size() > 0)
15.                 setUpdate = Updates.push("comments", com);
16.             else {
17.                 ArrayList<Document> comList = new ArrayList<>();
18.                 comList.add(com);
19.                 setUpdate = Updates.set("comments", comList);
20.             }
21.
22.             collection.updateOne(filter, setUpdate);
23.
24.         }catch(Exception ex){
25.             closeConnection();
26.             return false;
27.         }
28.         closeConnection();

```

```
29.     return true;
30. }
```

#### GET RECIPE FROM ID

This query gives the opportunity to retrieve an object containing a specific recipe information.

- Input: a recipe id.
- Output: an object with the recipe's information.

#### JAVA LANGUAGE

```
1.  public static Recipe getRecipesFromId( int id){
2.      openConnection();
3.      Recipe recipe;
4.      ObjectMapper objectMapper = new ObjectMapper();
5.      objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
6.      ArrayList<Document> myDoc = new ArrayList<>();
7.      try{
8.          for(Document doc: collection.find(eq("recipeId", id))){
9.              System.out.println(doc.toString());
10.             myDoc.add(doc);
11.         }
12.         RecipePojo pojo = objectMapper.readValue(myDoc.get(0).toJson(),
RecipePojo.class);
13.         recipe = Utils.mapRecipe(pojo);
14.         closeConnection();
15.         return recipe;
16.     } catch (Exception e) {
17.         e.printStackTrace();
18.         closeConnection();
19.         return null;
20.     }
21. }
```

#### FIND RECIPE GIVEN CERTAIN PARAMETERS

These queries give the opportunity to search a list of recipes which match with specific parameters.

First query based on author's name:

- Input: a string who represent the author's name.
- Output: a list of recipes ordered by the publication data.

#### MONGODB QUERY LANGUAGE

```
1. db.recipe.find
2. (
3.     {"authorname":<authorName>},
4.     {_id:0, recipeId:1, name:1, authorName:1, datePublished:1}
5. ).sort(
6.     {datePublished:1}
7. )
```

#### JAVA LANGUAGE

```
1.  public static ArrayList<Recipe> getRecipesFromAuthorName(String username){
2.      ArrayList<Document> results;
3.
4.      openConnection();
```



```

5.         Bson myMatch = Aggregates.match(Filters.eq("authorName", username));
6.         Bson mySort = sort(Sorts.descending("datePublished"));
7.         Bson projection = Aggregates.project( fields(excludeId(),
include("recipeId", "name", "authorName", "recipeCategory", "datePublished")));
8.
9.         results = collection.aggregate(Arrays.asList(myMatch, mySort, projection))
10.            .into(new ArrayList<>());
11.
12.         closeConnection();
13.         return getRecipesFromDocuments(results);
14.     }

```

Second query based on recipe category:

- Input: a string who represent a recipe category.
- Output: a list of recipes ordered by the publication data.

#### MONGODB QUERY LANGUAGE

```

1. db.recipe.find(
2.   {"recipeCategory": <recipeCategory> },
3.   {_id:0, recipeId:1, name:1, authorName:1, recipeCategory:1,
datePublished:1}
4. ).sort({datePublished:1})

```

#### JAVA LANGUAGE

```

1. public static ArrayList<Recipe> getRecipesFromCategory(String category){
2.     ArrayList<Document> results;
3.
4.     openConnection();
5.     Bson myMatch = Aggregates.match(Filters.eq("recipeCategory", category));
6.     Bson mySort = sort(Sorts.descending("datePublished"));
7.     Bson projection= Aggregates.project(fields(excludeId(), include("recipeId",
"name", "authorName", "recipeCategory", "datePublished")));
8.
9.     results = collection.aggregate(Arrays.asList(myMatch,mySort, projection)).into(new
ArrayList<>());
10.
11.     closeConnection();
12.     return getRecipesFromDocuments(results);
13. }

```

Third query based on two ingredients:

- Input: two strings representing two different ingredients that the recipe must have.
- Output: a list of recipes.

#### MONGODB QUERY LANGUAGE

```

1. db.recipe.find(
2.   {$and:[
3.     {"ingredients": {$regex:<ingredient_2>/}},
4.     {"ingredients": {$regex:<ingredient_2>/}}
5.   ]},
6.   {_id:0, recipeId:1, name:1, authorName:1, ingredients:1}
7. )

```

```

1. public static ArrayList<Recipe> getRecipesFromTwoIngredients(String ing1, String ing2){
2.     ArrayList<Document> results;
3.
4.     openConnection();
5.     String pattern1=".*" + ing1 +".*";
6.     String pattern2=".*" + ing2 +".*";
7.     Bson myMatch_1= Aggregates.match(Filters.regex("ingredients", pattern1));
8.     Bson myMatch_2= Aggregates.match(Filters.regex("ingredients", pattern2));
9.     Bson projection= Aggregates.project(fields(excludeId(), include("recipeId",
    "name", "authorName", "recipeCategory", "datePublished"))));
10.
11.     results = collection.aggregate(Arrays.asList(myMatch_1,myMatch_2,
    projection)).into(new ArrayList<>());
12.
13.     closeConnection();
14.     return getRecipesFromDocuments(results);
15. }

```

## Neo4j

## ADD RECIPE

This query gives the opportunity to insert a new recipe in the application with all the required information.

- Input: an object with all the recipe's information.
- Output: A new node.

```

1. public boolean addRecipe(Recipe r){
2.
3.     try(Session session= driver.session()){
4.
5.         session.writeTransaction((TransactionWork<Void>) tx ->{
6.             tx.run("match (u:User) where u.id=$usId CREATE (r:Recipe {id:$recId,
    name:$title, category:$recCat, datePublished:$datePublished}), (u)-[:ADDS]->(r)",
7.                 Values.parameters(
8.                     "usId", r.getAuthorId(),
9.                     "recId", r.getRecipeId(),
10.                    "title", r.getName(),
11.                    "recCat", r.getCategory(),
12.                    "datePublished",
    r.getDatePublished().toInstant().atZone(ZoneId.systemDefault()).toLocalDate()
13.                );
14.
15.            return null;
16.        });
17.    }catch(Exception ex){
18.        ex.printStackTrace();
19.
20.        return false;
21.    }
22.    return true;
23. }

```

## UPDATE RECIPE

This query gives the opportunity to change some information about a recipe.

- Input: an object with all the recipe's information (old and new information).
- Output: a new node with the modified information.

## JAVA LANGUAGE

```
1. public boolean updateRecipe(Recipe r){
2.     Recipe old = getRecipeFromId(r.getRecipeId());
3.     if(deleteRecipe(r)){
4.         if(addRecipe(r))
5.             return true;
6.         else{
7.             addRecipe(old);
8.             return false;
9.         }
10.    }
11.    else
12.        return false;
13.
14.
15. }
```

## DELETE RECIPE

This query gives the opportunity to eliminate a recipe from the application.

- Input: an object with all the recipe's information.

## JAVA LANGUAGE

```
1. public boolean deleteRecipe(Recipe r){
2.     try(Session session= driver.session()){
3.
4.         session.writeTransaction((TransactionWork<Void>) tx -> {
5.             tx.run( "MATCH (r:Recipe) " +
6.                     "WHERE r.id=$id " +
7.                     "DETACH DELETE r",
8.                     Values.parameters( "id", r.getRecipeId()) );
9.             return null;
10.        });
11.        return true;
12.
13.    }catch(Exception ex){
14.        ex.printStackTrace();
15.        return false;
16.    }
17. }
```

## ADD USER

This query gives the opportunity to insert a new user in the application.

- Input: an object containing all the user's information.
- Output: a new node.

## JAVA LANGUAGE

```
1. public boolean addUser(User u){
2.     try(Session session= driver.session()){
3.         session.writeTransaction((TransactionWork<Void>) tx -> {
4.             String[] names = u.getFullName().split(" ");
5.             tx.run ("CREATE (u:User {id: $id, username: $username, firstName:
6. $firstName, lastName: $lastName, email: $email, password: $password})",
7.                     Values.parameters("id", u.getUserId(), "username", u.getUsername(),
8. "firstName", names[0], "lastName", names[1], "email", u.getEmail(), "password",
9. u.getPassword()));
10.             return null;
11.        });
12.    }
13. }
```

```

9.
10.     }catch(Exception ex){
11.         ex.printStackTrace();
12.         return false;
13.     }
14.     return true;
15. }

```

## UPDATE USER

This query gives the opportunity to update specific user's information.

- Input: an object containing all the user's information.

## JAVA LANGUAGE

```

1. public boolean updateUser(User u){
2.     try(Session session= driver.session()){
3.
4.         session.writeTransaction((TransactionWork<Void>) tx -> {
5.             tx.run ( "match (u:User {id:$id}) " +
6.                 "set u.email=$email, u.fullname=$fullName,
7.                 u.password=$pass, u.username=$userName",
8.                 Values.parameters("email", u.getEmail(), "fullName",
9.                 u.getFullName(), "pass",
10.                 u.getPassword(), "userName", u.getUsername(), "id",
11.                 u.getUserId()));
12.             return null;
13.         } );
14.     }catch(Exception ex){
15.         ex.printStackTrace();
16.         return false;
17.     }
18. }

```

## DELETE USER

This query gives the opportunity to eliminate a user from the application.

- Input: an object containing all the user's information.

## JAVA LANGUAGE

```

1. public boolean deleteUser(User u){
2.     try(Session session= driver.session()){
3.
4.         session.writeTransaction((TransactionWork<Void>) tx -> {
5.             tx.run( "MATCH (u:User) WHERE u.id=$id DETACH DELETE u",
6.                 Values.parameters( "id", u.getUserId()) );
7.             return null;
8.         });
9.         return true;
10.
11.     }catch(Exception ex){
12.         ex.printStackTrace();
13.         return false;
14.     }
15. }

```

## FOLLOW

This query gives the opportunity to create a new relationship between two existing users.

- Input: the id of the people who follow and the id of followed people.

- Output: a new relationship.

#### JAVA LANGUAGE

```

1. public boolean follow(int following, int follower){
2.     try(Session session= driver.session()){
3.
4.         session.writeTransaction((TransactionWork<Void>) tx -> {
5.             tx.run ("match (a:User) where a.id= $usera " +
6.                 "match (b:User) where b.id=$userb " +
7.                 "merge (a)-[:FOLLOWS]->(b)",
8.                 Values.parameters("usera", following, "userb", follower));
9.             return null;
10.        } );
11.
12.    }catch(Exception ex){
13.        ex.printStackTrace();
14.        return false;
15.    }
16.    return true;
17. }

```

#### UNFOLLOW

This query gives the opportunity to eliminate a relationship between two existing users.

- Input: the id of the people who stop follow and the id of followed people.

#### JAVA LANGUAGE

```

1. public boolean unfollow(int following, int follower){
2.     try(Session session= driver.session()){
3.
4.         session.writeTransaction((TransactionWork<Void>) tx -> {
5.             tx.run ("match (a:User {id: $usera}) -[f:FOLLOWS]-> (b:User {id:$userb}) "
6.             +
7.                 "delete f",
8.                 Values.parameters("usera", following, "userb", follower));
9.             return null;
10.        } );
11.
12.    }catch(Exception ex){
13.        ex.printStackTrace();
14.        return false;
15.    }
16.    return true;
17. }

```

#### GET USERS

This query gives the opportunity to retrieve all the user present in the application.

- Output: users list.

#### JAVA LANGUAGE

```

1. public ArrayList<User> getUsers( int skip, int limit){
2.     ArrayList<User> users= new ArrayList<>();
3.
4.     try(Session session= driver.session()){
5.
6.         session.readTransaction(tx->{
7.             Result result = tx.run("match (u:User) " +

```

```

8.         "return u.id, u.username, u.email, u.firstName, u.lastName
9.         " +
10.         "order by u.username asc " +
11.         "skip $toSkip " +
12.         "limit $toLimit "
13.         , Values.parameters("toSkip", skip, "toLimit",limit));
14.         while(result.hasNext()){
15.             Record r= result.next();
16.             int id = r.get("u.id").asInt();
17.             String username = r.get("u.username").asString();
18.             String email = r.get("u.email").asString();
19.             String fullName = r.get("u.firstName").asString() + " " +
r.get("u.lastName").asString();
20.             User user= new User(id, username, fullName, email);
21.             users.add(user);
22.         }
23.         return users;
24.     });
25.
26.     }catch(Exception ex){
27.         ex.printStackTrace();
28.         return null;
29.     }
30.
31.     return users;
32. }

```

## GET RECIPES

This query gives the opportunity to retrieve all the recipes present in the application.

- Output: users list.

## JAVA LANGUAGE

```

1. public ArrayList<Recipe> getRecipes(int skip, int limit){
2.     ArrayList<Recipe> recipes= new ArrayList<>();
3.
4.     try(Session session= driver.session()){
5.
6.         session.readTransaction(tx->{
7.             Result result = tx.run("match (r:Recipe) where r.id IS NOT NULL and
(r.name is not null or r.name <> 'null') " +
8.             "return r.id, r.name, r.datePublished,
r.category order by r.datePublished desc " +
9.             "skip $toSkip " +
10.            "limit $toLimit"
11.            , Values.parameters("toLimit",limit, "toSkip",
skip));
12.
13.            while(result.hasNext()){
14.                Record r= result.next();
15.                int id = r.get("r.id").asInt();
16.                String name = r.get("r.name").asString();
17.                Date date;
18.                if(!r.get("r.datePublished").isNull()){
19.                    date = java.util.Date.from(r.get("r.datePublished").asLocalDate()
20.                    .atStartOfDay().atZone(ZoneId.systemDefault()).toInstant());
21.                }
22.                else
23.                    date = new Date();
24.                String category = r.get("r.category").asString();
25.                Recipe recipe= new Recipe(id, name, date, category);
26.                recipes.add(recipe);
27.            }

```

```

28.         return recipes;
29.     });
30.
31.     }catch(Exception ex){
32.         ex.printStackTrace();
33.         return null;
34.     }
35.     return recipes;
36. }

```

#### MAKE ADMIN

This query gives the opportunity to set a user as admin.

- Input: user's information.

#### JAVA LANGUAGE

```

1. public Boolean makeAdmin( User u){
2.     try(Session session= driver.session()){
3.         session.writeTransaction((TransactionWork<Void>) tx -> {
4.             tx.run ("match (u:User {id:$id}) " +
5.                 "set u.role=1 " +
6.                 "return u.role", Values.parameters("id", u.getUserId()));
7.             return null;
8.         } );
9.
10.    }catch(Exception ex){
11.        ex.printStackTrace();
12.        return false;
13.    }
14.    return true;
15. }

```

#### MAKE NOT ADMIN

This query gives the opportunity to set a user as normal user.

- Input: user's information.

#### JAVA LANGUAGE

```

1. public boolean makeNotAdmin( User u){
2.     try(Session session= driver.session()){
3.         session.writeTransaction((TransactionWork<Void>) tx -> {
4.             tx.run ("match (u:User {id: $id}) " +
5.                 "set u.role=0 " +
6.                 "return u.role", Values.parameters("id", u.getUserId()));
7.             return null;
8.         } );
9.
10.    }catch(Exception ex){
11.        ex.printStackTrace();
12.        return false;
13.    }
14.    return true;
15. }

```

#### FIND USERS GIVEN CERTAIN PARAMETERS

These queries give the opportunity to search a user which match with specific parameters.

First query based on user's full name:

- Input: a string which represents the user's full name.
- Output: an users list.

```

1. public ArrayList<User> getUsersFromFullname(String name){
2.     ArrayList<User> users= new ArrayList<>();
3.
4.     try(Session session= driver.session()){
5.         String[] sName= name.split(" ");
6.         session.readTransaction(tx->{
7.             Result result = tx.run("match (u:User) " +
8.                                     "where u.firstName = $fName and
9.                                     "return u.id, u.email, u.username, u.firstName,
10.                                     u.lastName"
11.                                     , Values.parameters("fName", sName[0], "lName", sName[1]));
12.             while(result.hasNext()){
13.                 Record r= result.next();
14.                 int id = r.get("u.id").asInt();
15.                 String username = r.get("u.username").asString();
16.                 String email = r.get("u.email").asString();
17.                 String fullName = r.get("u.firstName").asString() + " " +
18.                 r.get("u.lastName").asString();
19.                 User user= new User(id, username, fullName, email);
20.                 users.add(user);
21.             }
22.             return users;
23.         });
24.     }catch(Exception ex){
25.         ex.printStackTrace();
26.         return null;
27.     }
28.
29.     return users;
30. }

```

Second query based on user's id:

- Input: an integer which represents the user's id.
- Output: a single user.

```

1. public User getUsersFromId(int id){
2.
3.     try(Session session= driver.session()){
4.         User user;
5.         user = session.readTransaction(tx->{
6.             Result result = tx.run("match (u:User) where u.id = $id " +
7.                                     "return u.id, u.password, u.email, u.username,
8.                                     u.firstName, u.lastName, u.role",
9.                                     Values.parameters("id", id));
10.
11.             if (result.hasNext()){
12.                 Record r= result.next();
13.                 int uid = r.get("u.id").asInt();
14.                 String username = r.get("u.username").asString();
15.                 String password = r.get("u.password").asString();
16.                 String email = r.get("u.email").asString();
17.                 String fullName = r.get("u.firstName").asString() + " " +
18.                 r.get("u.lastName").asString();
19.                 int role;
20.                 if(!r.get("u.role").isNull()) {
21.                     System.out.println("role defined");
22.                     role = r.get("u.role").asInt();

```



```

21.         }
22.         else
23.             role = 0;
24.         return new User(uid, username, fullName, password, email, role);
25.     }
26.     return null;
27. });
28. return user;
29. }catch(Exception ex){
30.     ex.printStackTrace();
31.     return null;
32. }
33. }

```

## Analytics Implementations

### MongoDB

#### TOP-K HEALTHIEST RECIPES

This query gives the opportunity to search the top-k healthiest recipes for a given recipe category, where the “healthiest recipes” means the recipe with fewer calories, less fat, less sodium, and with more proteins.

- Input: a string who represent a recipe category and how many recipes to show.
- Output: a list of recipes.

#### MONGODB QUERY LANGUAGE

```

1. db.recipe.aggregate(
2. [{
3.     $match: {
4.         recipeCategory: <recipeCategory>
5.     }
6. }, {
7.     $match: {
8.         calories: {
9.             $exists: true
10.        },
11.        fatContent: {
12.            $exists: true
13.        },
14.        sodiumContent: {
15.            $exists: true
16.        },
17.        proteinContent: {
18.            $exists: true
19.        }
20.    }
21. }, {
22.     $sort: {
23.         calories: 1,
24.         fatContent: 1,
25.         sodiumContent: 1,
26.         proteinContent: -1
27.     }

```

```

28. }, {
29.     $limit: <howManyToShow>
30. }]
31. )

```

## JAVA LANGUAGE

```

1. public static ArrayList<Recipe> searchTopKHealthiestRecipes(String category, int k){
2.     ArrayList<Recipe> recipes;
3.     ArrayList<Document> results;
4.     Gson gson = new Gson();
5.
6.     openConnection();
7.     Bson m = Aggregates.match(Filters.eq("recipeCategory", category));
8.     Bson m1 = Aggregates.match(Filters.exists("calories", true));
9.     Bson m2 = Aggregates.match(Filters.exists("fatContent", true));
10.    Bson m3 = Aggregates.match(Filters.exists("sodiumContent", true));
11.    Bson m4 = Aggregates.match(Filters.exists("proteinContent", true));
12.    Bson s = sort(Sorts.ascending("calories"));
13.    Bson s1= sort(Sorts.ascending("fatContent"));
14.    Bson s2= sort(Sorts.ascending("sodiumContent"));
15.    Bson s3= sort(Sorts.descending("proteinContent"));
16.    Bson l= limit(k);
17.
18.    results = collection.aggregate(Arrays.asList(m, m1, m2, m3, m4, s, s1, s2, s3,
19.    1)).into(new ArrayList<>());
20.
21.    closeConnection();
22.    return getRecipesFromDocuments(results);
23. }

```

## TOP-K RECIPES WITH THE HIGHEST NUMBER OF 5-STAR REVIEWS

This query gives the opportunity to search, given a certain category, the recipes which received, most time, comments with 5 star which is the maximum.

- Input: a string who represent a recipe category and how many recipes to show.
- Output: a list of recipes.

## MONGODB QUERY LANGUAGE

```

1. db.recipe.aggregate([
2.     {$match:{"recipeCategory": <recipeCategory> }},
3.     {$unwind:"$comments"},
4.     {$match:{"comments.rating":5}},
5.     {$group: {_id:"$recipeId", count:{$sum:1}}},
6.     {$sort: {count:-1}},
7.     {$limit: <howManyToShow> }
8. ])

```

## JAVA LANGUAGE

```

1. public static ArrayList<Recipe> searchTopKReviewedRecipes(String category, int k){
2.     ArrayList<Document> results;
3.
4.     openConnection();
5.     Bson myMatch_1 = Aggregates.match(Filters.eq("recipeCategory", category));
6.     Bson myUnwind = unwind("$comments");

```

```

7.         Bson myMatch_2 = Aggregates.match(Filters.eq("comments.rating", 5));
8.         Bson myGroup = new Document("$group", new Document("_id", new
Document("recipeId", "$recipeId")
9.             .append("name", "$name")
10.            .append("recipeCategory", "$recipeCategory")
11.            .append("datePublished", "$datePublished")
12.            .append("authorName", "$authorName"))
13.            .append("count", new Document("$sum", 1)));
14.         Bson mySort = sort(Sorts.descending("count"));
15.         Bson myLimit = limit(k);
16.
17.         results =
collection.aggregate(Arrays.asList(myMatch_1, myUnwind, myMatch_2, myGroup, mySort, myLimit))
18.             .into(new ArrayList<>());
19.
20.         closeConnection();
21.         ArrayList<Document> recipeValues = new ArrayList<>();
22.         for(Document doc: results){
23.             recipeValues.add((Document) doc.get("_id"));
24.         }
25.
26.         return getRecipesFromDocuments(recipeValues);
27.     }

```

#### FASTEST RECIPES TO PREPARE

This query gives the opportunity to sort the recipes by cook time and preparation time, for a given recipe category.

- Input: a string who represent a recipe category and how many recipes to show.
- Output: a list of recipes ordered by cook and preparation time.

#### MONGODB QUERY LANGUAGE

```

1. db.recipe.aggregate(
2.   [{
3.     $match: {
4.       recipeCategory: <recipeCategory>
5.     }
6.   }, {
7.     $match: {
8.       cookTime: {
9.         $exists: true
10.      },
11.       prepTime: {
12.         $exists: true
13.      }
14.    }
15.  }, {
16.    $sort: {
17.      cookTime: 1,
18.      prepTime: 1
19.    }
20.  }, {$limit: <howManyToShow>}]
21. )

```

```

1. public static ArrayList<Recipe> searchFastestRecipes(String category, int k){
2.     ArrayList<Document> results;
3.
4.     openConnection();
5.     Bson m1 = Aggregates.match(Filters.eq("recipeCategory", category));
6.     Bson m2 = Aggregates.match(Filters.exists("cookTime", true));
7.     Bson m3 = Aggregates.match(Filters.exists("prepTime", true));
8.     Bson s1 = sort(Sorts.ascending("cookTime"));
9.     Bson s2 = sort(Sorts.ascending("prepTime"));
10.    Bson myLimit = limit(k);
11.
12.    results = collection.aggregate(Arrays.asList(m1, m2, m3, s1, s2,
    myLimit)).into(new ArrayList<>());
13.
14.    closeConnection();
15.    return getRecipesFromDocuments(results);
16. }

```

### RECIPE WITH THE FEWEST INGREDIENTS

This query gives the opportunity to sort the recipes by the number of ingredients, for a given recipe category.

- Input: a string who represent a recipe category and how many recipes to show.
- Output: a list of recipes ordered by the number of ingredients.

### MONGODB QUERY LANGUAGE

```

1. db.recipe.aggregate(
2.  [{
3.    $match: {
4.      recipeCategory: <recipeCategory>
5.    }
6.  }, {
7.    $unwind: {
8.      path: '$ingredients'
9.    }
10.  }, {
11.    $group: {
12.      _id: '$recipeId',
13.      number_of_ingredients: {
14.        $count: {}
15.      }
16.    }
17.  }, {
18.    $sort: {
19.      number_of_ingredients: 1
20.    }
21.  }, {$limit: <howManyToShow>}]
22. )

```

```

1. public static ArrayList<Recipe> searchFewestIngredientsRecipes(String category, int k){
2.     ArrayList<Document> results;
3.
4.     openConnection();
5.     Bson m1 = Aggregates.match(Filters.eq("recipeCategory", category));
6.     Bson u= unwind("$ingredients");
7.     Bson g= new Document("$group", new Document("_id", new
Document("recipeId", "$recipeId")
8.         .append("name", "$name")
9.         .append("recipeCategory", "$recipeCategory")
10.        .append("datePublished", "$datePublished")
11.        .append("authorName", "$authorName"))
12.        .append("numberOfIngredients", new Document("$sum", 1)));
13.     Bson s= sort(Sorts.ascending("numberOfIngredients"));
14.     Bson myLimit = limit(k);
15.
16.     results = collection.aggregate(Arrays.asList(m1, u, g, s, myLimit)).into(new
ArrayList<>());
17.     closeConnection();
18.
19.     ArrayList<Document> recipeValues = new ArrayList<>();
20.     for(Document doc: results){
21.         recipeValues.add((Document) doc.get("_id"));
22.     }
23.
24.     return getRecipesFromDocuments(recipeValues);
25. }

```

### USERS' RANKING SYSTEM

This query gives the opportunity to retrieve the top-k users, sorted by the average mark assigned to their recipes.

- Input: Number of users to retrieve and how many users to show.
- Output: a user list.

### Mongodb query language

```

1. db.recipe.aggregate([
2.   { $unwind: "$comments" },
3.   { $group: { _id: { recipe: "$recipeId", author: "$authorId" },
4.   avgRating: { $avg: "$comments.rating" } } },
5.   { $group: { _id: "$_id.author", avgRating: { $avg: "$avgRating" }
6.   } },
7.   {$sort: {avgRating: -1}},
8.   {$limit: <numberOfUsers>}
9. ]);

```

```

1. public static ArrayList<User> userRankingSystem(int k){
2.     ArrayList<User> users;
3.     ArrayList<Document> results;
4.
5.     openConnection();
6.     Bson unwind_comments = unwind("$comments");
7.     Bson group1 = new Document("$group", new Document("_id",

```

```

8.                                     new Document("recipe",
"$recipeId")
9.                                     .append("author",
"$authorId"))
10.                                    .append("avgRating", new
Document("$avg", "$comments.rating"));
11.        Bson group2 = Aggregates.group("$_id.author", Accumulators.avg("avgRating",
"$avgRating"));
12.        Bson sort_by_rating = sort(Sorts.descending("avgRating"));
13.        Bson limitResults = limit(k);
14.
15.        results= collection.aggregate(Arrays.asList(unwind_comments, group1, group2,
sort_by_rating, limitResults))
16.        .into(new ArrayList<>());
17.        closeConnection();
18.        Neo4jDriver neo4j = Neo4jDriver.getInstance();
19.        if(results.size() == 0)
20.            return null;
21.        users = new ArrayList<>();
22.        for(Document d: results){
23.            User user = neo4j.getUsersFromId(d.getInteger("_id"));
24.            users.add(user);
25.        }
26.        return users;
27.    }

```

#### RECIPES WITH THE HIGHEST LIFESPAN

This query allows to the admin to find which recipes received comments for the longest timespan (the difference between the timestamp of the latest comment and the first comment on a recipe).

- Input: K, number of recipes to show.
- Output: Top-K durable recipes.

#### MONGODB QUERY LANGUAGE

```

1. db.recipe.aggregate([
2.   {$unwind: "$comments"},
3.   {$group: { _id: "$recipeId", mostRecentComment: { $max:
4.     "$comments.dateModified" }, leastRecentComment: { $min:
5.     "$comments.dateModified" }}}},      {$project: { _id: 0, recipe:
6.     "$_id", lifespan: { $subtract: ["$mostRecentComment",
7.     "$leastRecentComment"]}}},
8.   {$sort: {lifespan: -1}},
9.   {$limit: 10}
10.  ]);

```

#### JAVA LANGUAGE

```

1. public static ArrayList<Recipe> searchHighestLifespanRecipes(int k){
2.     ArrayList<Recipe> recipes;
3.     ArrayList<Document> results;
4.     Gson gson = new Gson();
5.
6.     openConnection();
7.     Bson unwind_comments = unwind("$comments");
8.     Bson group1 = new Document("$group", new Document("_id", new
Document("recipeId", "$recipeId")

```

```

9.         .append("name", "$name")
10.        .append("recipeCategory", "$recipeCategory")
11.        .append("datePublished", "$datePublished")
12.        .append("authorName", "$authorName"))
13.        .append("mostRecentComment", new Document("$max",
"$comments.dateModified"))
14.        .append("leastRecentComment", new Document("$min",
"$comments.dateModified"))));
15.        BsonArray operands = new BsonArray();
16.        operands.add(new BsonString("$mostRecentComment"));
17.        operands.add(new BsonString("$leastRecentComment"));
18.        Bson project_lifespan = Aggregates.project(fields(excludeId(), include("_id"),
19.        new Document("lifespan", new Document("$subtract",
operands))));
20.        Bson sort_by_lifespan = sort(Sorts.descending("lifespan"));
21.        Bson limitResults = limit(k);
22.
23.        results= collection.aggregate(Arrays.asList(unwind_comments, group1,
project_lifespan, sort_by_lifespan,
24.        limitResults)).into(new ArrayList<>());
25.        closeConnection();
26.        ArrayList<Document> recipeValues = new ArrayList<>();
27.        for(Document doc: results){
28.            recipeValues.add((Document) doc.get("_id"));
29.        }
30.
31.        return getRecipesFromDocuments(recipeValues);
32.    }

```

Neo4J

#### SUGGESTED RECIPES

This is the query to get the suggested recipes, means the recipes added by an user that is followed by a specific user.

- Input: the id of a specific user, how many recipe to skip and how many recipe to retrieve.
- Output: a list of recipes added by the followers of a specific user.

Domain-specific	Graph-centric
What are the suggested recipes for a specific user? (The recipes posted by an user that is followed by the specific user)	What are the nodes (Recipe) that have an exact distance of two ingoing hops (the first of type FOLLOWS and the second of type ADDS) from the node take into consideration (User)?

CYPHER

```

1. match (u:User)-[f:FOLLOWS]->(u2:User)
2. match (r:Recipe)<-[a:ADDS]-(u2)
3. where u.id= <idUser>
4. return r, u, u2, f, a

```

JAVA LANGUAGE

```

1. public ArrayList<Recipe> searchSuggestedRecipes(int skip, int howMany, String userId){
2.     ArrayList<Recipe> recipes= new ArrayList<>();
3.
4.     try(Session session= driver.session()){

```

```

5.
6.         session.readTransaction(tx->{
7.             Result result = tx.run("match (u:User)-[f:FOLLOWS]->(u2:User) " +
8.                                     "match (r:Recipe)-[a:ADDS]-(u2) " +
9.                                     "where u.id=$userId " +
10.                                    "return r.id, r.name, r.category, r.datePublished order by
11. r.datePublished desc " +
12.                                     "skip $skip limit $limit",
13.                                     Values.parameters("userId", userId, "skip", skip, "limit",
14. howMany));
15.
16.             while(result.hasNext()){
17.                 Record r= result.next();
18.                 int id= r.get("r.id").asInt();
19.                 String name = r.get("r.name").asString();
20.                 Recipe rec= new Recipe(id, name);
21.                 recipes.add(rec);
22.             }
23.             return recipes;
24.         });
25.     }catch(Exception ex){
26.         ex.printStackTrace();
27.         return null;
28.     }
29.     return recipes;
30. }

```

## MOST FOLLOWED USER

This query returns a list of the user who have the highest number of followers.

- Input: the number of users to returned.
- Output: a list of users.

Domain-specific	Graph-centric
Who are the most followed users?	Which are the nodes with the greatest number of entering edge (follows).

## CYPHER

```

1. match (u:User)<-[f:FOLLOWS]-(:User)
2. return u.id, count(DISTINCT f) as follower
3. order by follower desc
4. limit 5

```

## JAVA LANGUAGE

```

1. public ArrayList<User> mostFollowedUsers(int limit){
2.     ArrayList<User> users = new ArrayList<>();
3.
4.     try ( Session session = driver.session() ) {
5.         session.readTransaction( tx -> {Result result = tx.run(
6.             "match (u:User)<-[f:FOLLOWS]-(:User) " +
7.             "return u.id, u.username, count(DISTINCT f) as follower " +
8.             "order by follower desc " +
9.             "limit $1",
10.             Values.parameters( "1", limit) );

```



```

11.
12.         while(result.hasNext()){
13.             Record r = result.next();
14.             int id= r.get("u.id").asInt();
15.             String name= r.get("u.username").asString();
16.             int followers= r.get("follower").asInt();
17.             User u= new User(id, name, followers);
18.
19.             users.add(u);
20.         }
21.
22.         return users;
23.     });
24. } catch (Exception ex){
25.     ex.printStackTrace();
26.     return null;
27. }
28. return users;
29. }

```

### TOP-K MOST ACTIVE USERS

This query allows users to find who have published the highest number of recipes.

- Input: K number of users to return.
- Output: Users list.

Domain-specific	Graph-centric
Who are the most active users?	Which are the user nodes with the greatest number of exiting edges (adds)?

### CYPHER LANGUAGE

```

1. MATCH (user: User)-->(x:Recipe)
2. RETURN user, count(x)
3. ORDER BY count(x) DESC
4. LIMIT 10

```

### JAVA LANGUAGE

```

1. public ArrayList<User> getMostActiveUsers(int k){
2.     ArrayList<User> users = new ArrayList<>();
3.
4.     try(Session session = driver.session()){
5.         session.readTransaction( tx -> {Result result = tx.run(
6.             "match (user:User) --> (x:Recipe) " +
7.             "return user.id, user.username, count(x) " +
8.             "order by count(x) " +
9.             "limit $k", Values.parameters("k", k) );
10.
11.         while(result.hasNext()){
12.             Record r = result.next();
13.             int id = r.get("user.id").asInt();
14.             String username = r.get("user.username").asString();
15.             User u = new User(id, username);
16.             users.add(u);
17.         }
18.         return users;
19.     });
20. }catch(Exception ex){

```

```

20.         ex.printStackTrace();
21.         return null;
22.     }
23.     return users;
24. }

```

## Cross-Database Consistency Management

Given how the data and the application queries have been distributed between the two databases, the operations which require a cross-database consistency management are the following:

1. Add new recipe (Figure 14)

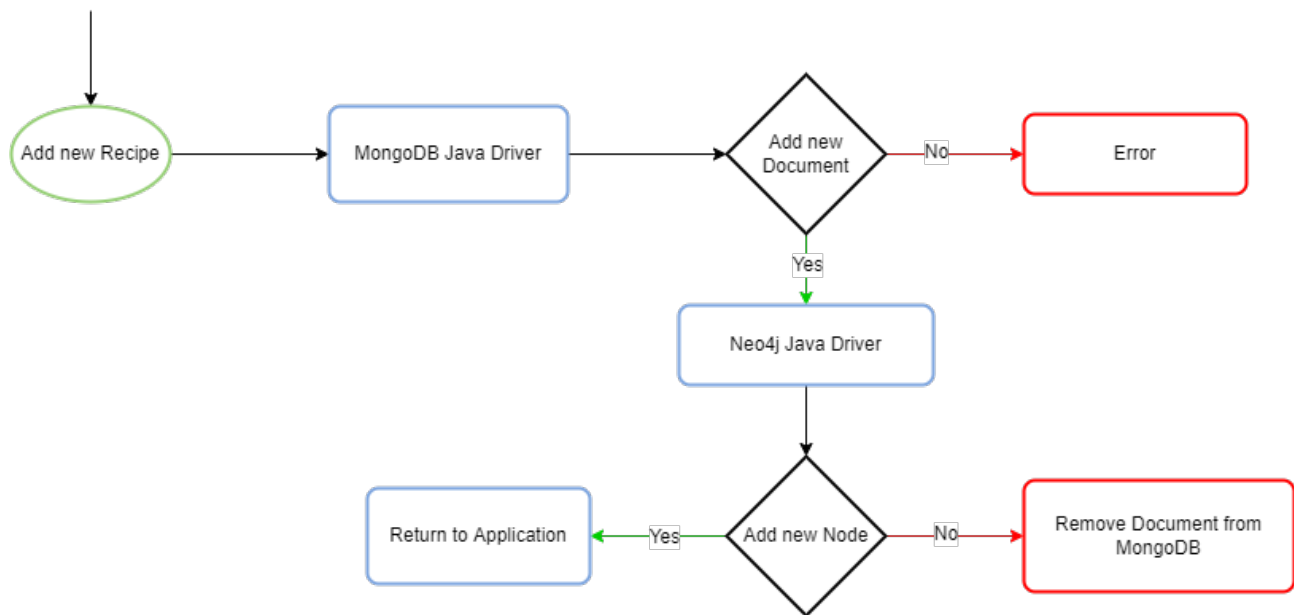


FIGURE 14 - ADD NEW RECIPE

2. Update a recipe (Figure 15)

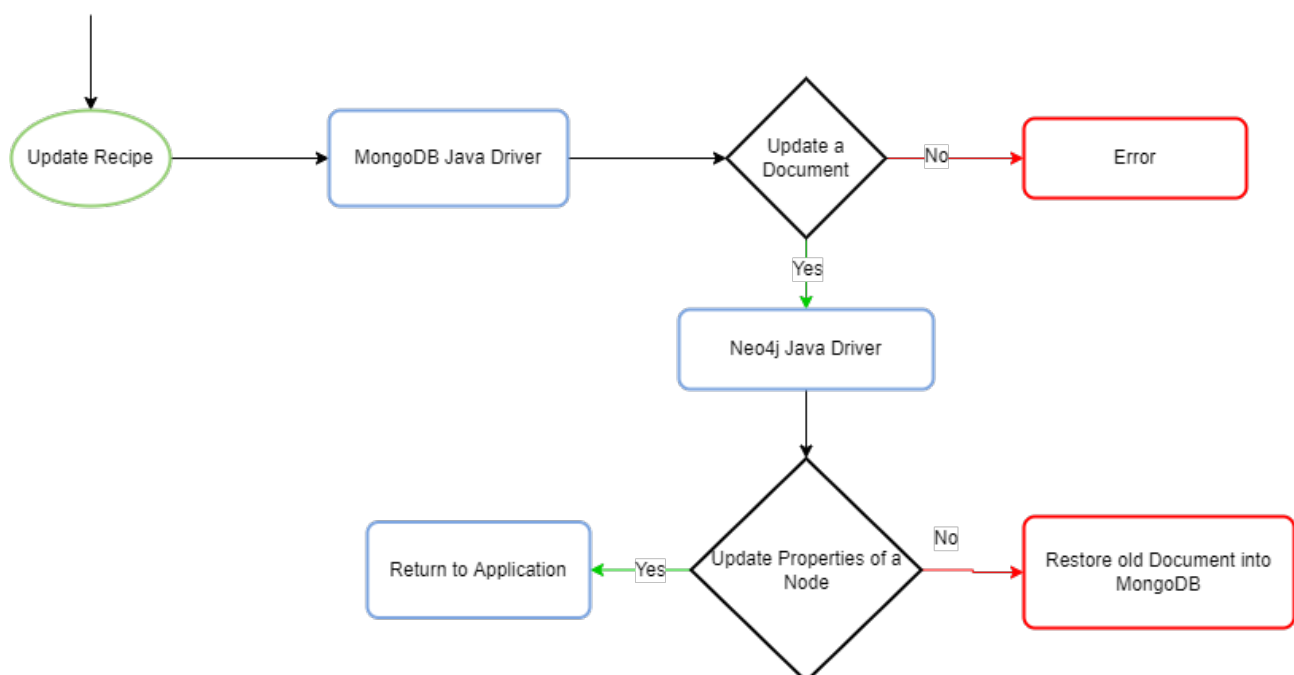


FIGURE 15 - UPDATE RECIPE

### 3. Delete a recipe (Figure 16)

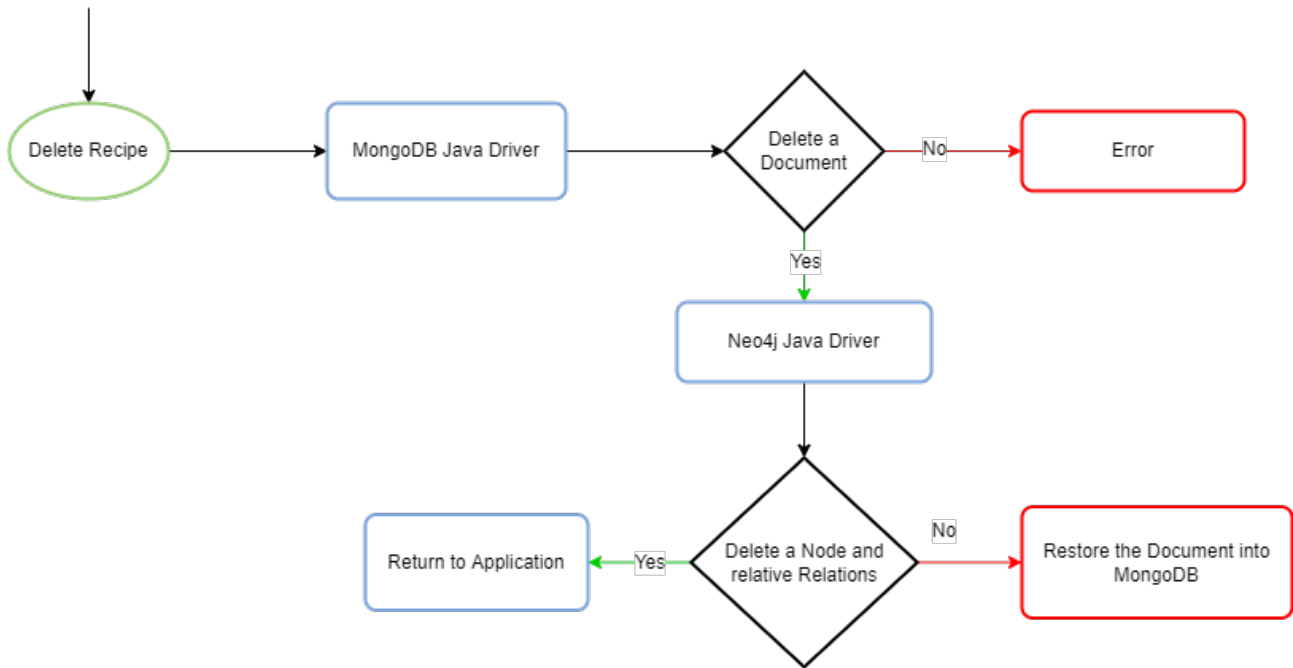


FIGURE 16 - DELETE RECIPE

### 4. Delete a user (Figure 17)

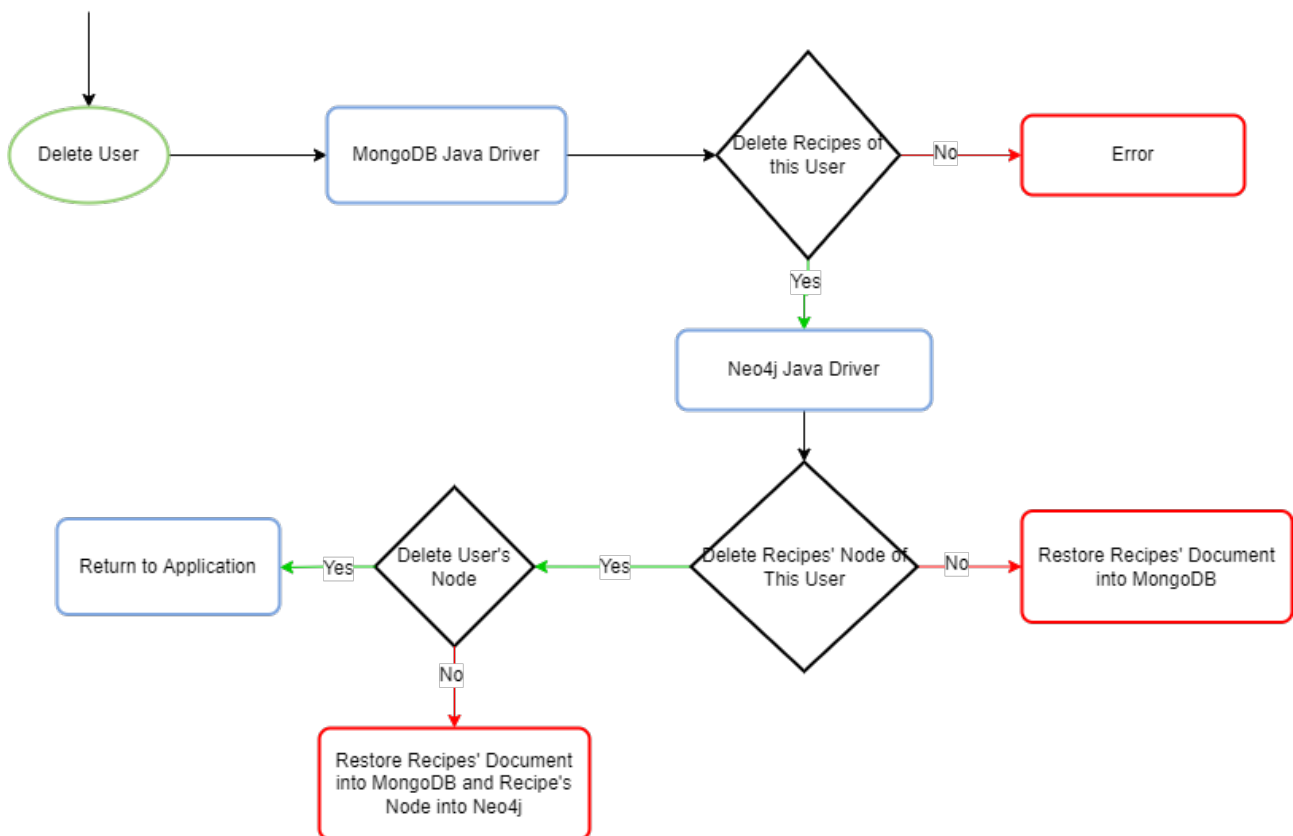


FIGURE 17 - DELETE USER

## MongoDB – Replica Set

In our application, some of the most important requirements are the Availability and the Partition Tolerance, thus is very important to have a distributed system in order to ensure:

- An high availability of the service, so as we are able to response to any query.
- A partition protection, so as failures of individual nodes or connections between nodes do not impact the system as a whole.

These two objectives can be reached using replicas (copy of the same data on different servers).

In the following section, we discuss about this technique explaining the design choices performed in order to obtain the best performance achievable and in order to respect the given requirements.

### Replica Configuration

A replica set is available and it is composed by a primary, the server that takes client requests, and two secondaries, the servers that keep copies of the primary's data.

The mongod instances are hosted on three different virtual machines, provided by the University of Pisa, in order to create a shared cluster, and one of them hosts an instance of neo4j (Figure ).

Each member of the set is able to communicate with the other members and they have the same port in which they are listened:

Virtual machine	IP address	Port	OS
Replica-0	172.16.4.51	27017	Ubuntu
Replica-1	172.16.4.52	27017	Ubuntu
Replica-2	172.16.4.53	27017	Ubuntu

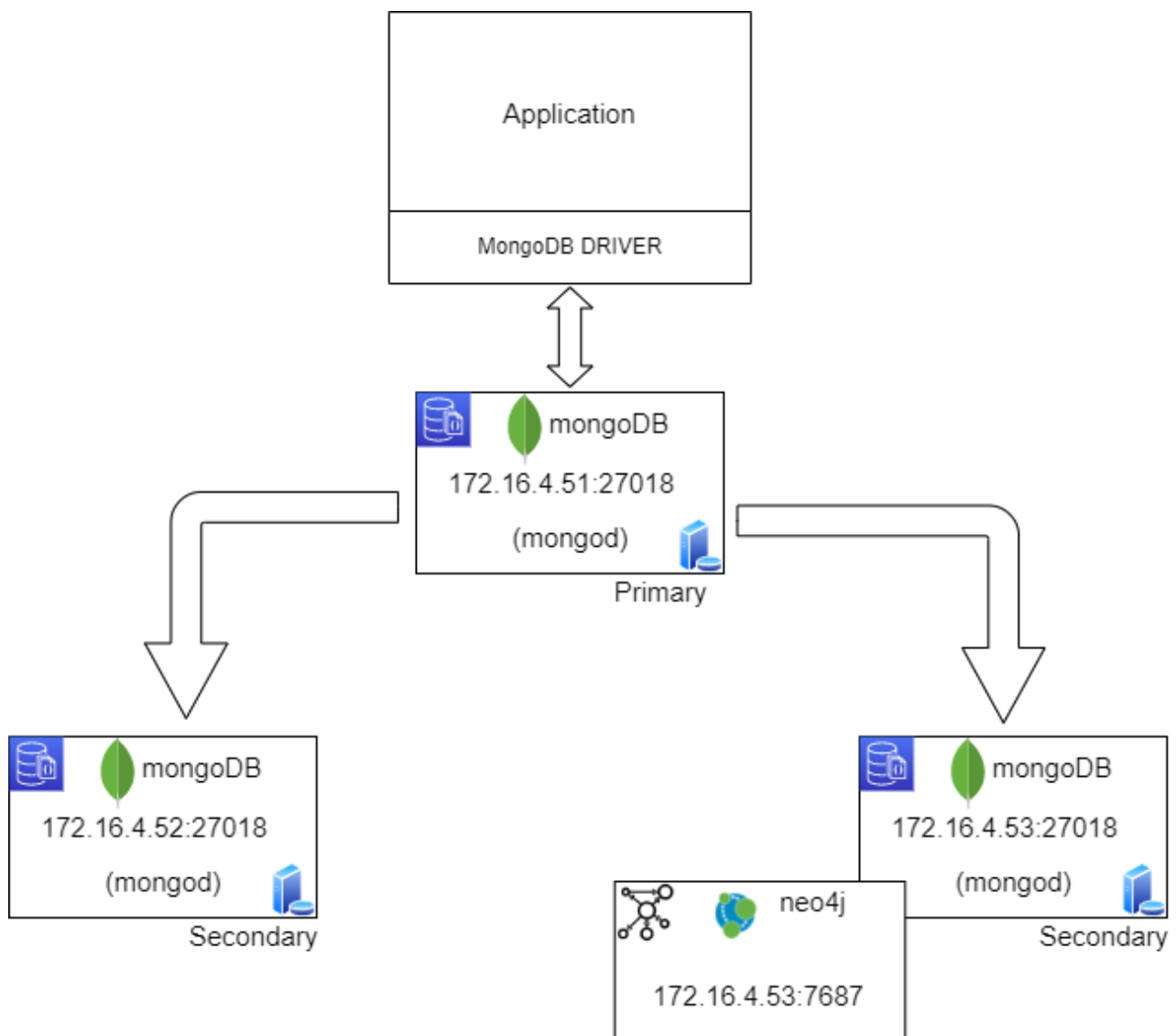


FIGURE 18 - DISTRIBUTED REPLICASET

The replica configuration is shown below:

```
1. rsconf={
2.   _id: "coogether",
3.   members: [
4.     {_id:0,host:"172.16.4.51:27018",priority:3},
5.     {_id:1,host:"172.16.4.52:27018",priority:2},
6.     {_id:2,host:"172.16.4.53:27018",priority:1},
7.   ]
8. };
```

The VM 172.16.4.51 has the highest priority, so unless problem arises, the replica on this machine will be elected like primary.

The application is not characterized by a huge amount of writings, at most one document is added or updated. Instead, the readings are very frequent and often involve multiple documents, which makes the application a read oriented one, as you will understand from the next chapters.

Replicas ensure system availability. Analysis of the queries, described below, shows that reads predominate over writes. It is therefore essential that the user always read the updated version of the data. So, every time a write takes place it expects all replicas to be updated, in this way the reads can be performed on the replica with less network latency in order to have a faster response.

We have decided to introduce both Write Concern and Read Preferences constraints and set this configuration:

```
1. "mongodb://172.16.4.51:27018,172.16.4.52:27018,172.16.4.53:27018/?retryWrites=true&w=3&wtimeoutMS=5000&readPreference=nearest"
```

1. Write concern → w=3: wait for all the replicas to be updated.
2. Read preference → readPreference=nearest: read from the member of the replica set with the least network latency to have the fastest response.

### Replica crash

If the primary fails, one of the secondary will be elected as the new primary. Since we have assigned a priority on each replicas, we can control the behaviour of the election algorithm to predict which of the two secondaries will be promoted. In our case, when the primary is not available, the VM 172.16.4.52 will be marked as the new primary.

When we set up the replica set the cluster will have this configuration:

	Primary	Secondary	Secondary
IP	172.16.4.51	172.16.4.52	172.14.4.53
Port	27018	27018	27018

If the primary crashes or is unavailable for some reason, the new configuration will be as follows:

	Primary	New Primary	Secondary
IP	<del>172.16.4.51</del>	172.16.4.52	172.14.4.53
Port	<del>27018</del>	27018	27018

## Analysis

### MongoDB Queries Analysis

Write Operation		
Operation	Expected Frequency	Cost
Add Recipe	Low	Average (add a document)
Update Recipe	Low	Average (update a document)
Delete Recipe	Low	Average (delete a document)
Add Comment	Medium	Low (add a field)
Delete Recipes of a User	Low	Very High (delete multiple documents)

Read Operation		
Operation	Expected Frequency	Cost
Get Recipes From Id	High	Low (1 read)
Get Recipes From Author Name	High	Average (multiple reads)
Get Recipes From Category	High	Average (multiple reads)
Get Recipes From Two Ingredients	High	Average (multiple reads)
Search Top K Healthiest Recipes	High	Average (multiple reads)
Search Top K Reviewed Recipes	High	Very High (complex aggregation)
Search Fastest Recipe	High	Average (multiple reads)
Search Fewest Ingredients Recipe	High	High (complex aggregation)
User Ranking System	High	Very High (very complex aggregation)
Search Highest Lifespan Recipes	Low	High (complex aggregation)

### Index Analysis

We decided to use indexes in order to speed up the application. We performed some test to measure the speed improvement obtained by using indexes. This test is carried out using the `explain()` function offered by MongoDB.

The index that we added on MongoDB are:

1. Index on Recipe Id.
2. Index on Author name.
3. Index on Recipe Category.

Query	Results without Index	Results with Index
Get Recipes From Id	executionTimeMillis: 20ms, totalKeysExamined: 0,	executionTimeMillis: 0ms, totalKeysExamined: 1,

	totalDocsExamined: 30000	totalDocsExamined: 1
Get Recipes From Author Name	executionTimeMillis: 23ms, totalKeysExamined: 0, totalDocsExamined: 30000	executionTimeMillis: 1ms, totalKeysExamined: 692, totalDocsExamined: 692
Get Recipes From Category	executionTimeMillis: 28ms, totalKeysExamined: 0, totalDocsExamined: 30000	executionTimeMillis: 4ms, totalKeysExamined: 2481, totalDocsExamined: 2481
Search Top K Healthiest Recipes	executionTimeMillis: 84ms, totalKeysExamined: 0, totalDocsExamined: 30000	executionTimeMillis: 15ms, totalKeysExamined: 2480, totalDocsExamined: 2480
Search Top K Reviewed Recipes	executionTimeMillis: 153ms, totalKeysExamined: 0, totalDocsExamined: 30000	executionTimeMillis: 60ms, totalKeysExamined: 2480, totalDocsExamined: 2480
Search Fastest Recipe	executionTimeMillis: 67ms, totalKeysExamined: 0, totalDocsExamined: 30000	executionTimeMillis: 37ms, totalKeysExamined: 2480, totalDocsExamined: 2480
Search Fewest Ingredients Recipe	executionTimeMillis: 95ms, totalKeysExamined: 0, totalDocsExamined: 30000	executionTimeMillis: 33ms, totalKeysExamined: 2480, totalDocsExamined: 2480

## Neo4J Queries Analysis

Read Operation		
Operation	Expected Frequency	Cost
Get Users	High	Average (multiple reads)
Get Users From Unique	Low	Low (1 read)
Get Users From Full Name	High	Average (multiple reads)
Get Users From Id	High	Low (1 read)
Get Number of Following Users	Average	Average (multiple reads)
Get Number of Follower Users	Average	Average (multiple reads)
Get Following Users	Average	High (multiple reads)
Get Follower Users	Average	High (multiple reads)
Get Recipes	High	Average (multiple reads)
Get all Categories	High	Average (multiple reads)
Search Suggested Recipes	High	High (multiple reads)
Most Followed User	High	High (multiple reads)
Get Most Active User	High	High (multiple reads)

Write Operation		
Operation	Expected Frequency	Cost
Add Recipe	Low	Average (create a node and a relation)
Update Recipe	Low	Low (update properties of a node)
Delete Recipe	Low	Average (delete a node and a relation)



Delete Recipes of a User	Low	Very High (delete multiple nodes and multiple relations)
Add User	Low	Low (create a node)
Update User	Low	Low (update properties of a node)
Delete User	Low	High (delete multiple nodes and multiple recipes)
Follow	High	Low (create a relation)
Unfollow	High	Low (delete a relation)
Make Admin	Low	Low (update properties of a node)
Make Not Admin	Low	Low (update properties of a node)

## Index Analysis

The index that we added on Neo4j are:

- Index on Recipe Category.
- Index on Recipe Publication Date.
- Index on User Id.
- Index on Username.

Query	Results without Index	Results with Index
Get Users	resultConsumedAfter: 49ms resultAvailableAfter: 1ms	resultConsumedAfter: 42ms resultAvailableAfter: 1ms
Get Users From Unique	resultConsumedAfter: 129ms resultAvailableAfter: 1ms	resultConsumedAfter: 54ms resultAvailableAfter: 22ms
Get Users From Id	resultConsumedAfter: 42ms resultAvailableAfter: 1ms	resultConsumedAfter: 10ms resultAvailableAfter: 35ms
Get Number of Following Users	resultConsumedAfter: 40ms resultAvailableAfter: 1ms	resultConsumedAfter: 1ms resultAvailableAfter: 217ms
Get Number of Follower Users	resultConsumedAfter: 54ms resultAvailableAfter: 1ms	resultConsumedAfter: 1ms resultAvailableAfter: 19ms
Get Following Users	resultConsumedAfter: 38ms resultAvailableAfter: 1ms	resultConsumedAfter: 1ms resultAvailableAfter: 19ms
Get Follower Users	resultConsumedAfter: 40ms resultAvailableAfter: 1ms	resultConsumedAfter: 0ms resultAvailableAfter: 21ms
Get Recipes	resultConsumedAfter: 62ms resultAvailableAfter: 1ms	resultConsumedAfter: 59ms resultAvailableAfter: 1ms
Get all Categories	resultConsumedAfter: 21ms resultAvailableAfter: 0ms	resultConsumedAfter: 22ms resultAvailableAfter: 10ms
Search Suggested Recipes	resultConsumedAfter: 800ms resultAvailableAfter: 0ms	resultConsumedAfter: 1ms resultAvailableAfter: 152ms

## Sharding proposal

We think to split the whole dataset in three different chunks starting from the replica set that we have. We also think to use three replica set for each shard in order to avoid failures and maintain high availability.

We decided to use as **sharding keys** of recipe collection the field **category**, because the queries return, most of the time, recipes of a specific category.

Finally, according to the decision taken, we propose the **Hashed Strategy** as a partition algorithm. In this way, the field chosen as the sharding key will be mapped through a hash function.

## User Manual

In this chapter we discuss how to use the application from the user's point of view.

In order to configure the application an XML file is provided; this file contains configuration parameters that can affect the proper work of the application. The file contains the following code:

```
• <?xml version="1.0" encoding="UTF-8"?>
  <it.unipi.dii.inginf.lsdب.cogether.config.ConfigurationParameters>
    <mongoFirstIp>172.16.4.51</mongoFirstIp>
    <mongoFirstPort>27018</mongoFirstPort>
    <mongoSecondIp>172.16.4.52</mongoSecondIp>
    <mongoSecondPort>27018</mongoSecondPort>
    <mongoThirdIp>172.16.4.53</mongoThirdIp>
    <mongoThirdPort>27018</mongoThirdPort>
    <mongoUsername></mongoUsername>
    <mongoPassword></mongoPassword>
    <mongoDbName>cogether</mongoDbName>
    <neo4jIp>172.16.4.53</neo4jIp>
    <neo4jPort>7687</neo4jPort>
    <neo4jUsername>neo4j</neo4jUsername>
    <neo4jPassword>root</neo4jPassword>
  </it.unipi.dii.inginf.lsdب.cogether.config.ConfigurationParameters>
```

## Home page

At the first launch of the application the user visualizes the list of recipes the application offers and a set of fields to define some recipe's filter.

The user can also login or register itself clicking on the user icon and visualise all the registered user clicking on the appropriate bottom (Figure 19).

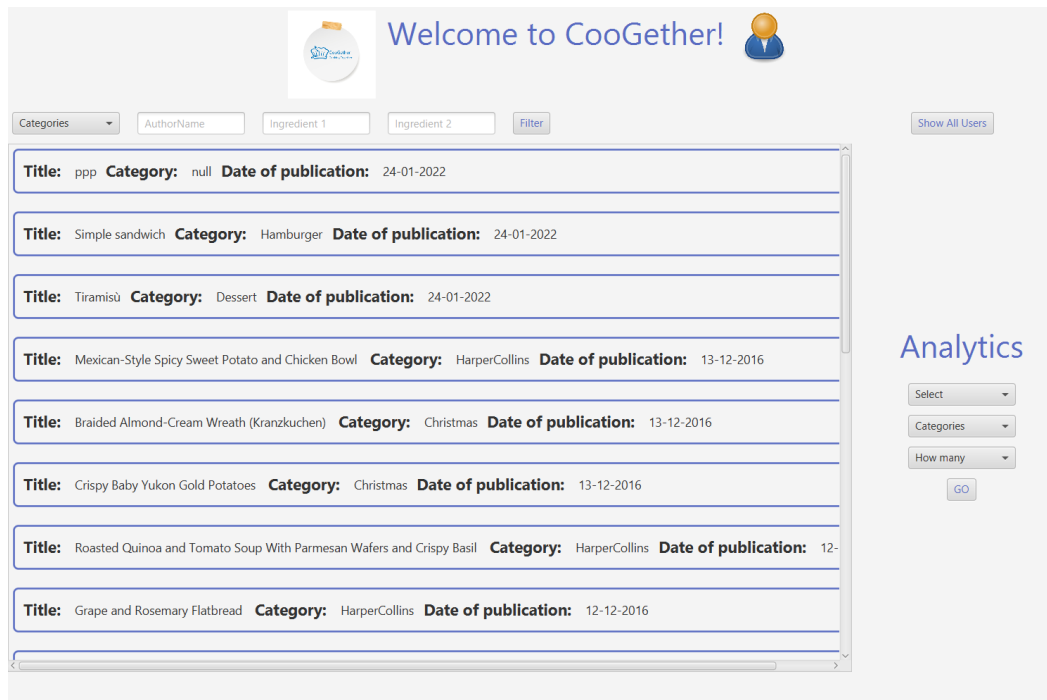


FIGURE 19 - HOME PAGE

## Filters

The possible filters are (Figure 19):

1. Category: composed by a choice box containing all the present category.
2. Author Name: the user must insert the desired username
3. Ingredient1 and Ingredient2: they must be selected together to visualise only the recipes containing these two ingredients.

Only one of these three types of filters must be selected.

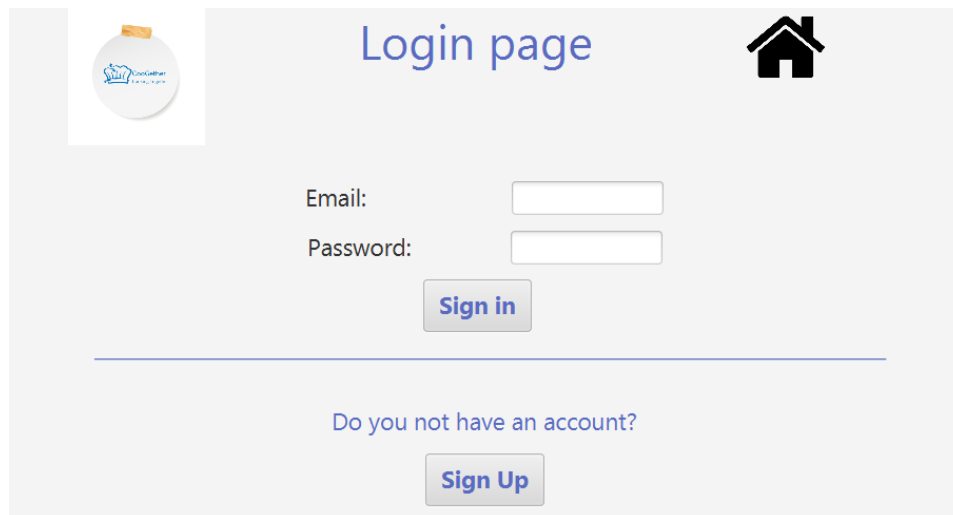
## Analytics

On the right side of the page (Figure 19) the user can define the analytics that the application offers. For all of them three information are mandatory:

- Analytics' type: choose among top healthiest, plus votes equal to 5, top fastest and with few ingredients. If the user is an admin user it can find an additional analytics to retrieve recipes with top highest lifespan.
- Recipes' category: choose a specific category.
- How many: choose how many recipes visualise.

## Login

In this page the user can insert its email and its password to log in or click on Sign Up to register itself (Figure 20).

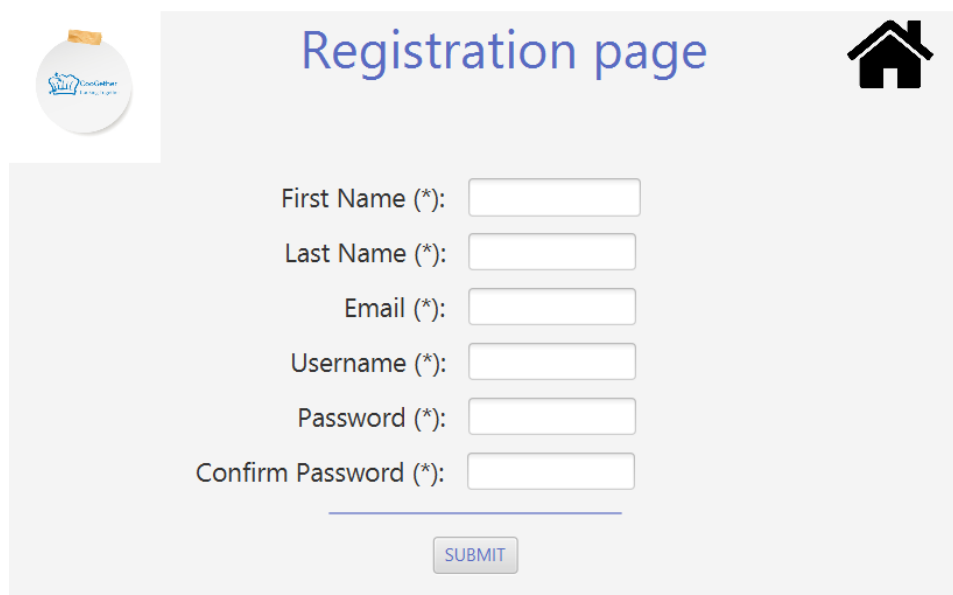


The login page features a light gray background. In the top left corner, there is a circular logo with a blue and white design. In the top right corner, there is a black house icon. The title "Login page" is centered at the top in a blue font. Below the title, there are two input fields: "Email:" and "Password:". Below the "Password:" field is a "Sign in" button. A horizontal line separates the login section from the registration section. Below the line, the text "Do you not have an account?" is centered, followed by a "Sign Up" button.

FIGURE 20 - LOGIN PAGE

## Registration

In this page (Figure 21) the user can insert all the required information to register itself in the application.



The registration page features a light gray background. In the top left corner, there is a circular logo with a blue and white design. In the top right corner, there is a black house icon. The title "Registration page" is centered at the top in a blue font. Below the title, there are six input fields: "First Name (\*)", "Last Name (\*)", "Email (\*)", "Username (\*)", "Password (\*)", and "Confirm Password (\*)". Below the "Confirm Password (\*)" field is a "SUBMIT" button.

FIGURE 21 - REGISTRATION PAGE

## Show users

In this page (Figure 22) the user finds all the registered users. It can decide to follow or unfollow one or more of them clicking on the appropriate bottom. If the user is admin it find an additional button to make admin another user.

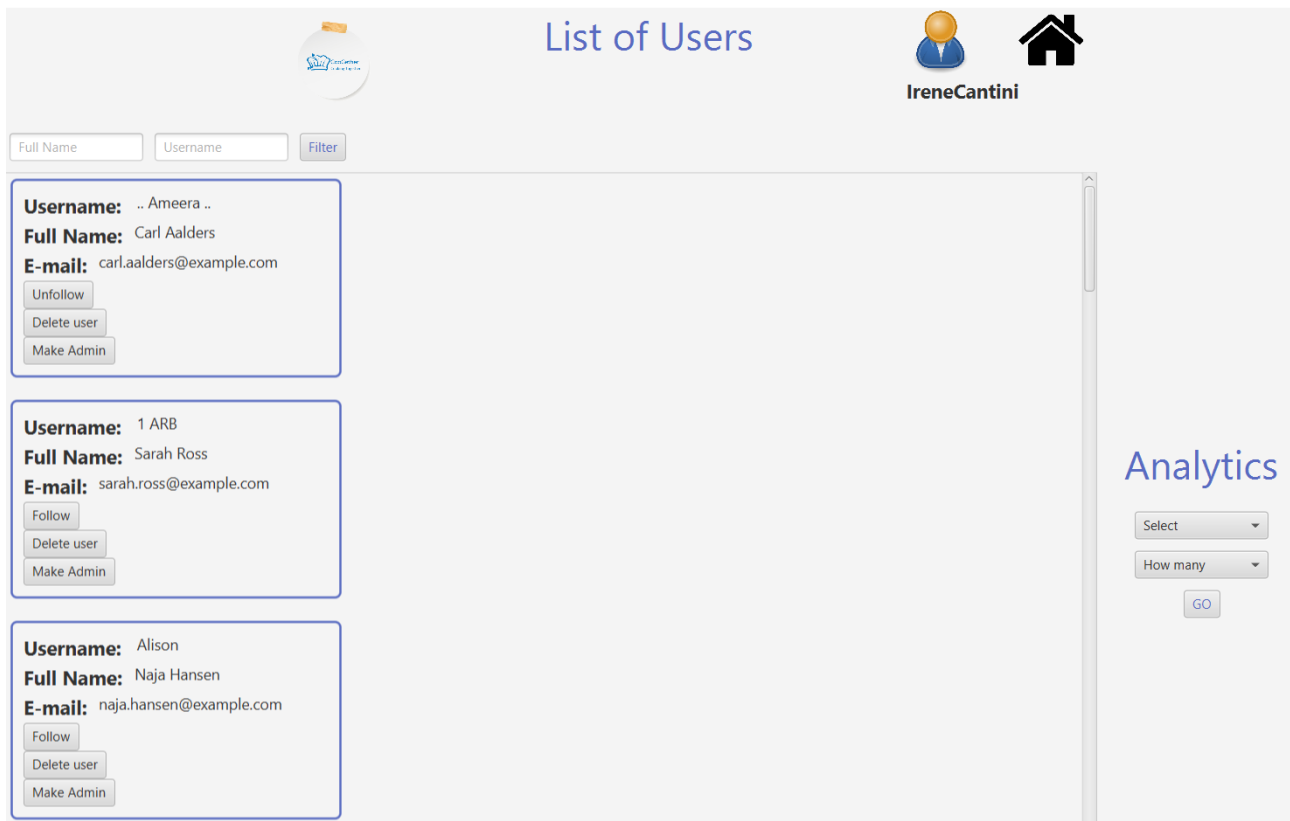


FIGURE 22 - LIST OF USERS PAGE

## Filters

The possible filters are (Figure 22):

1. User Full name: the user must insert the desired full name
2. User username: the user must insert the desired username

Only one of these three types of filters must be selected.

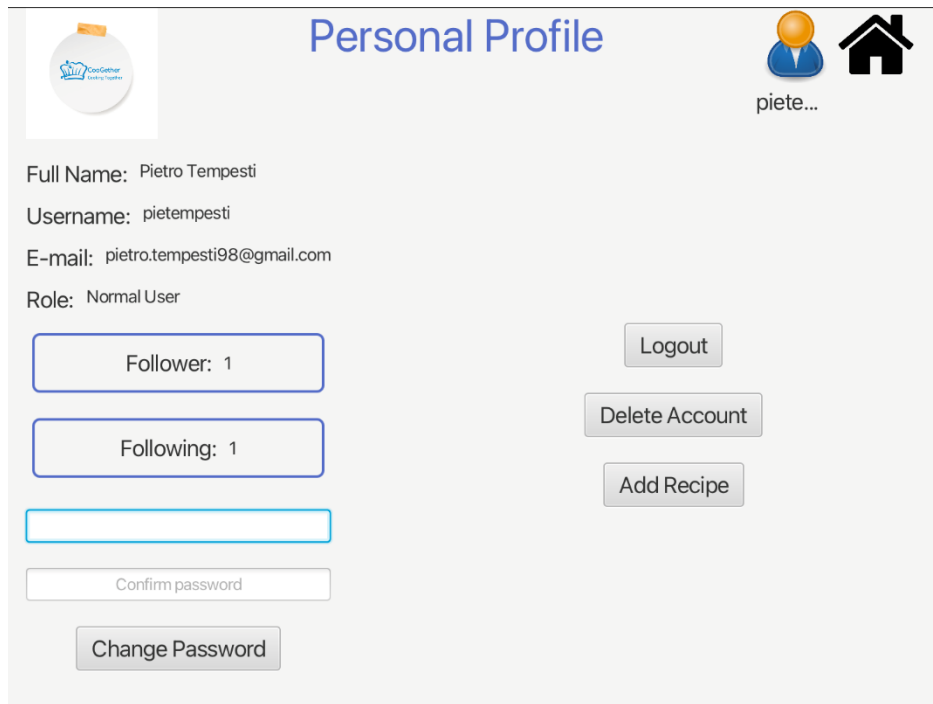
## Analytics

On the right side of the page (Figure 22) the user can define the analytics that the application offers. For all of them three information are mandatory:

- Analytics' type: choose among top active users, top followed users and user ranking system
- How many: choose how many users visualise.

## User page

In this page (Figure 23) all the user information is shown. The user can click on Follower or Following to visualize the respective users list, can logout itself, delete its account and add a new recipe.



**Personal Profile**

Full Name: Pietro Tempesti  
 Username: pietempesti  
 E-mail: pietro.tempesti98@gmail.com  
 Role: Normal User

Logout  
 Delete Account  
 Add Recipe

Follower: 1  
 Following: 1

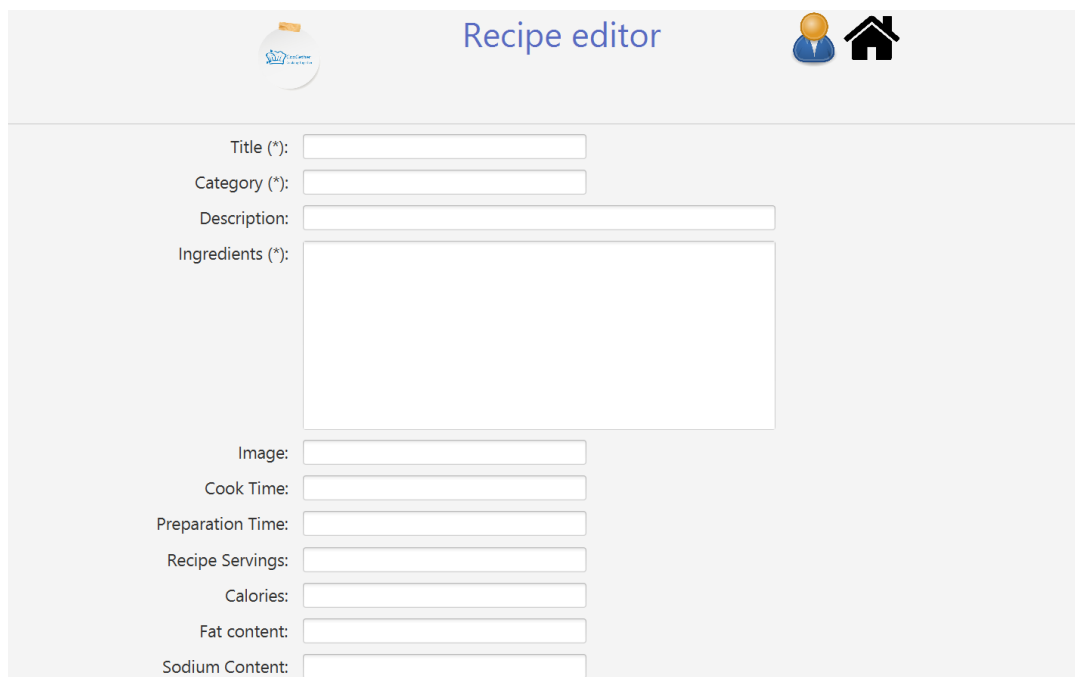
Confirm password  
 Change Password

**FIGURE 23 - PROFILE PAGE**

## Add or update recipe

In this page (Figure 24) the user can insert all the required information about a new recipe in the application.

The same page is shown when the user want update some field of an own recipe.



**Recipe editor**

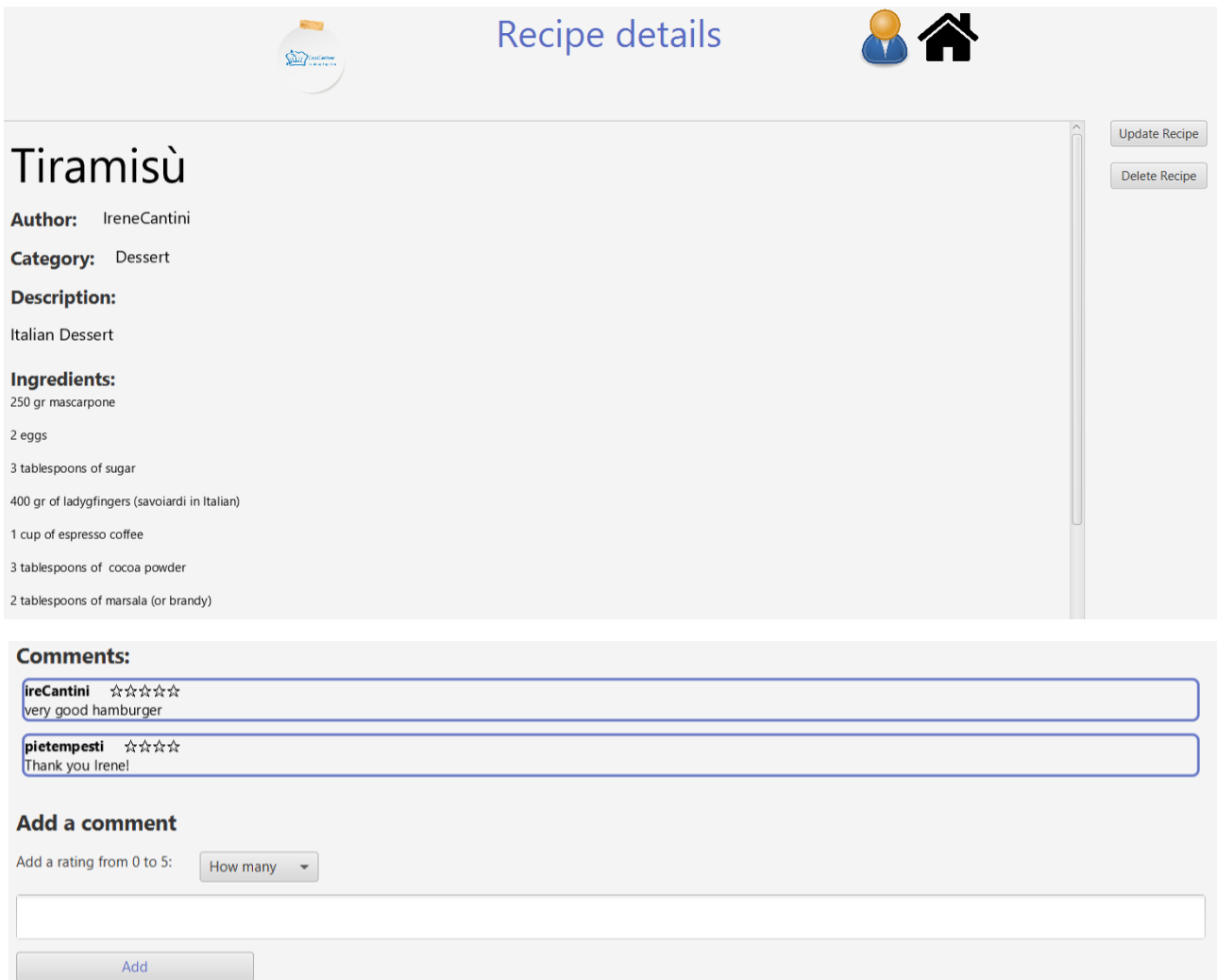
Title (\*):   
 Category (\*):   
 Description:   
 Ingredients (\*):   
 Image:   
 Cook Time:   
 Preparation Time:   
 Recipe Servings:   
 Calories:   
 Fat content:   
 Sodium Content:




**FIGURE 24 - RECIPE EDITOR PAGE**

## View recipe

In this page (Figure 25) the user finds all the information about a recipe and it can leave a new comment.

If the user is also the recipe's owner, it can decide to update or delete it clicking on special buttons.



 **Recipe details**  

# Tiramisù

**Author:** IreneCantini

**Category:** Dessert

**Description:**  
Italian Dessert

**Ingredients:**  
250 gr mascarpone  
2 eggs  
3 tablespoons of sugar  
400 gr of ladyfingers (savoiardi in Italian)  
1 cup of espresso coffee  
3 tablespoons of cocoa powder  
2 tablespoons of marsala (or brandy)

**Comments:**

**IreneCantini** ☆☆☆☆  
very good hamburger

**pietempesi** ☆☆☆  
Thank you Irene!

**Add a comment**

Add a rating from 0 to 5:

FIGURE 25 - RECIPE DETAILS PAGE