

UNIVERSITY OF PISA

Large Scale and Multi-Structured Databases

*Year 2022/2023*

SMART FRIDGE

ANTONINO NIGRO

GABRIELE MARCUCCETTI

MIRCO CONCU

github link: https://github.com/tonioskr/SmartFridge.git

**INDEX**

1. INTRODUCTION pag. 3

2. REQUIREMENTS ANALYSIS pag. 3

* 2.1. Application Actors
* 2.2. Functional and Non-Functional Requirements

3. UML DIAGRAMS pag. 4

* 3.1. Use-Case Diagram pag. 4
* 3.2. Class Diagram pag. 5

4. DATABASE ORGANIZATION pag. 6

* 4.1. Data Model pag. 7
* 4.2 Graph DB pag. 9
* 4.3 Data among DBs pag.10

5. SOFTWARE ARCHITECTURE pag.11

* 5.1. Repository Structure pag.12

6. INSTRUCTION MANUAL pag.15

* 6.1. user side pag.15
* 6.2. admin side pag.24

7. AGGREGATIONS AND INDEXES pag.26

* 7.1 Database consistency management pag.26
* 7.2 Aggregation pag.27
* 7.3 Test and Statistical Analysis (indexes) pag.31
* 7.4 Typical “on-graph” query pag.33
* 7.5 CRUD operation for MongoDB pag.35

8. SHARDING AND REPLICAS pag.43

9. CONCLUSIONS pag.44

1 **INTRODUCTION**

Smart fridge is an application developed to allow a user to effectively manage his fridge. Each user has his own fridge, represented by a list of ingredients which are specified the quantities and the expiring dates. The user can also modify his own fridge by removing ingredient, and modify the quantity. The user can also create and publish his recipes and see other recipes shared by other users. Can also follow other uses; this is useful to see the recipes published by them. Based on the ingredients that he has on his fridge the application offers him a list of possible recipe that he can prepare with them.

**2. REQUIREMENTS ANALYSIS**

Actor: user - registered user - administrator

. **Functional Requirement - User**

1. The system must allow unregistered user to register
2. The system must allow unregistered user to login

. **Functional Requirement - Registered User**

1. The system must allow registered user to browse product
2. The system must allow registered user to find product by name
3. The system must allow registered user to insert ingredient into the fridge
4. The system must allow registered user to remove ingredient from the fridge
5. The system must allow registered user to modify the quantity of a ingredient in the fridge
6. The system must allow registered user to see detail of the ingredient
7. The system must allow registered user to see his/her account information
8. The system must allow registered user to create a new recipe
9. The system must allow registered user to browse recipes
10. The system must allow registered user to add a comment to a recipe
11. The system must allow registered user to like a recipe
12. The system must allow registered user to follow/unfollow other users
13. The system must allow registered user to see suggested users
14. The system must allow registered user to see recipe published by the followed user
15. The system must allow registered user to see recipe which ingredient match with the one that he/she has in the fridge

. **Functional requirement - administrator**

1. The system must allow registered administrator to update ingredient information
2. The system must allow registered administrator to remove an ingredient
3. The system must allow registered administrator to remove comments of a recipe
4. The system must allow registered administrator to remove recipe
5. The system must allow registered administrator to modify a recipe

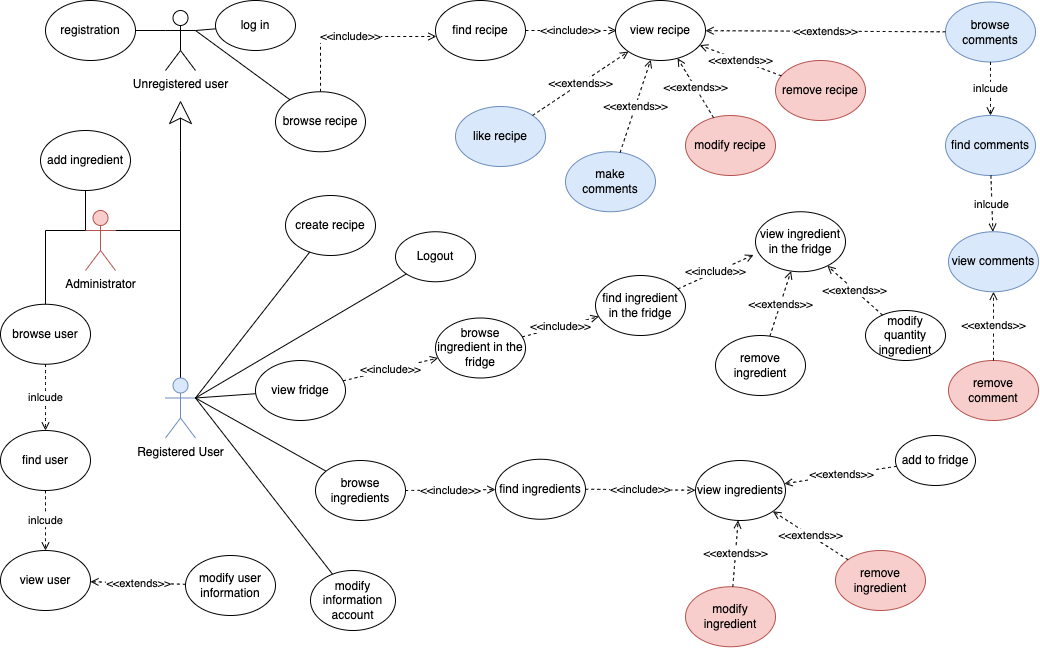
**Non-functional requirement**

* The system must be tolerant to data loss
* Code has to written in a way that can be easily readable and easily maintainable
* The application must be user-friendly
* The system must be available 24/7
* The system must be connected to internet

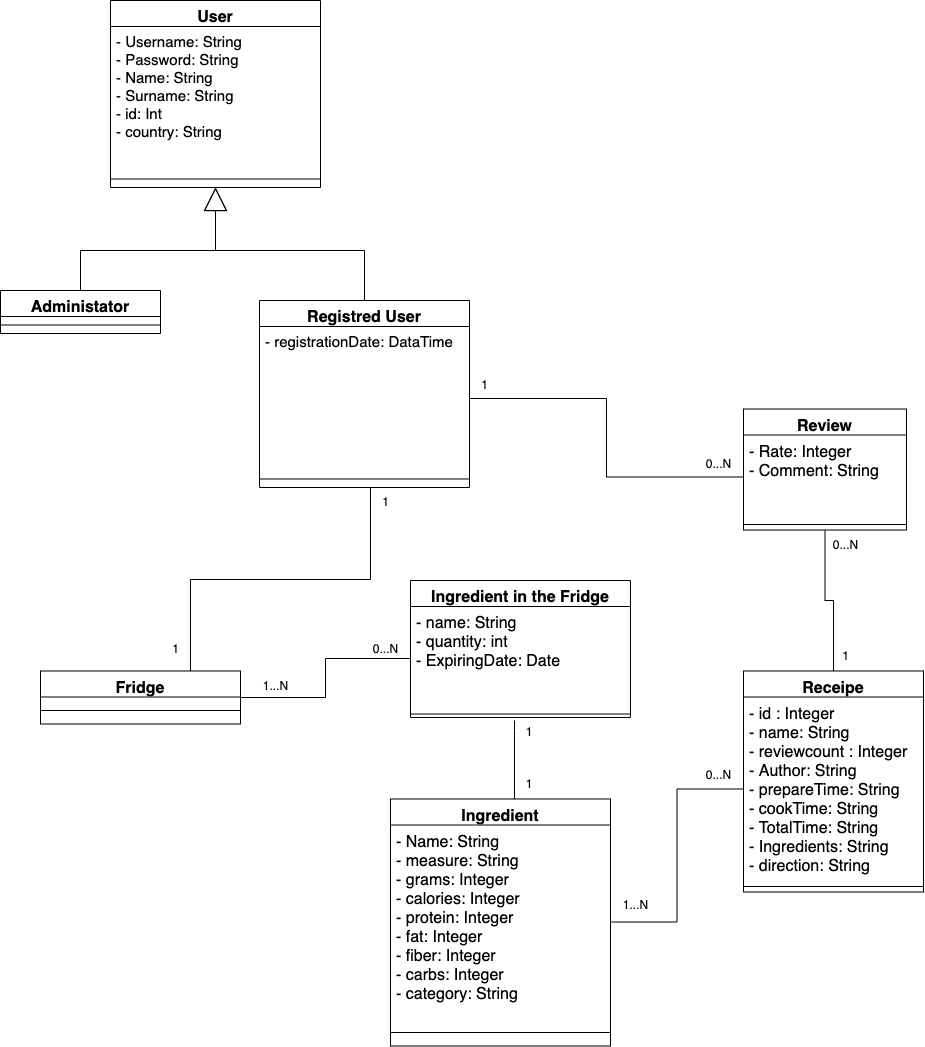
**3. UML DIAGRAMS**

**3.1. Use-Case Diagram**

**I**n the diagram we can see three main actors: User, Unregistered User and the Administrator. The user, meaning a unregistered User can only register. The Admin and the registered user can use different services offered by the application as shown above.

****

**3.2. Class Analysis Diagram**



User is a generalization of the two types of users: administrator and registered user. A registered user owns a fridge, regardless of whether it is empty or contains any products, and a fridge belongs to only one registered user.

A fridge can contain 1 or more ingredient, just as it could happen that it is empty, while a ingredient can be placed in one or more fridges.

An ingredient in the fridge is characterized by a name, a quantity, which represents how many units of that ingredient are contained in the fridge, and an expiry date, declared when the ingredient is placed in the fridge.

One or more ingredients can form a recipe. On the other hand, a recipe contains at least one ingredient, just as it could contain N.

A recipe may have 0 or more reviews. A given review, on the other hand, belongs to only one recipe.

A review can only be made by a registered user. A given review will have been made by a given user, who however may have made n reviews. At the same time, it is not said that a user has made at least one review.

**4. DATABASE ORGANIZATION**

We mainly used two datasets: one for ingredients and one for recipes.

For the ingredients the dataset comes from kaggle: <https://www.kaggle.com/datasets/pes12017000148/food-ingredients-and-recipe-dataset-with-images>

For the recipe with the author: <https://www.kaggle.com/datasets/kanaryayi/recipe-ingredients-and-reviews>

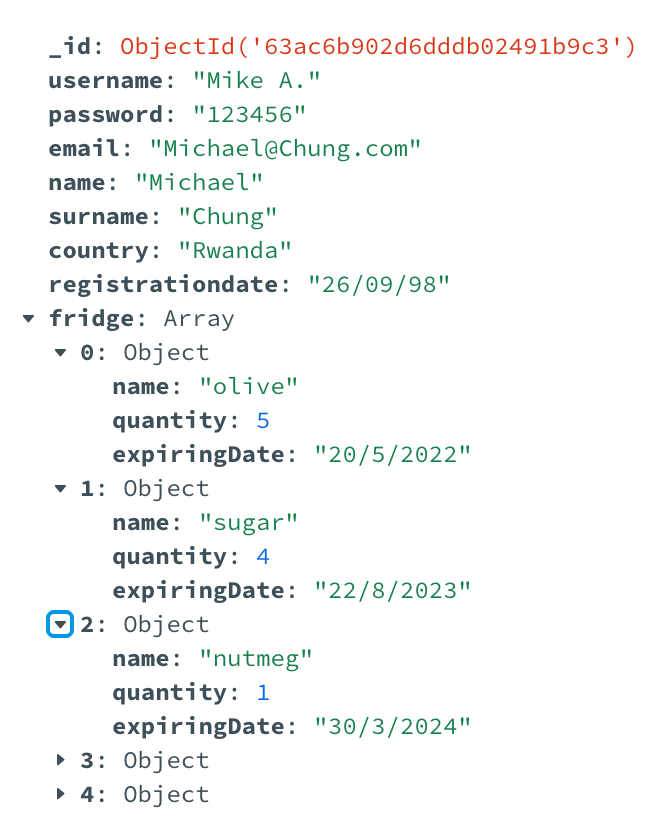
From the link we can see different file with extension csv.   
For building the final collection recipe we did the following:

1. join recipes.csv and clean\_review.csv. This two files has the same field except for the ingredient one where in one his a String where each ingredient is associated with the quantity to be used in the recipes. In the other, the field ‘ingredient’, is an array of ingredient. We need both to offer different functionality explained in the manual. Call the resulting final recipe.csv
2. we changed extension of the file recipe.csv and reviews.csv in .json
3. Then, thanks to a code written in java by us, from the file .json, we created the class recipe and review.
4. associate the id of the author of the review with his username
5. embedded the review into the corresponding recipes
6. convert the class recipes in json

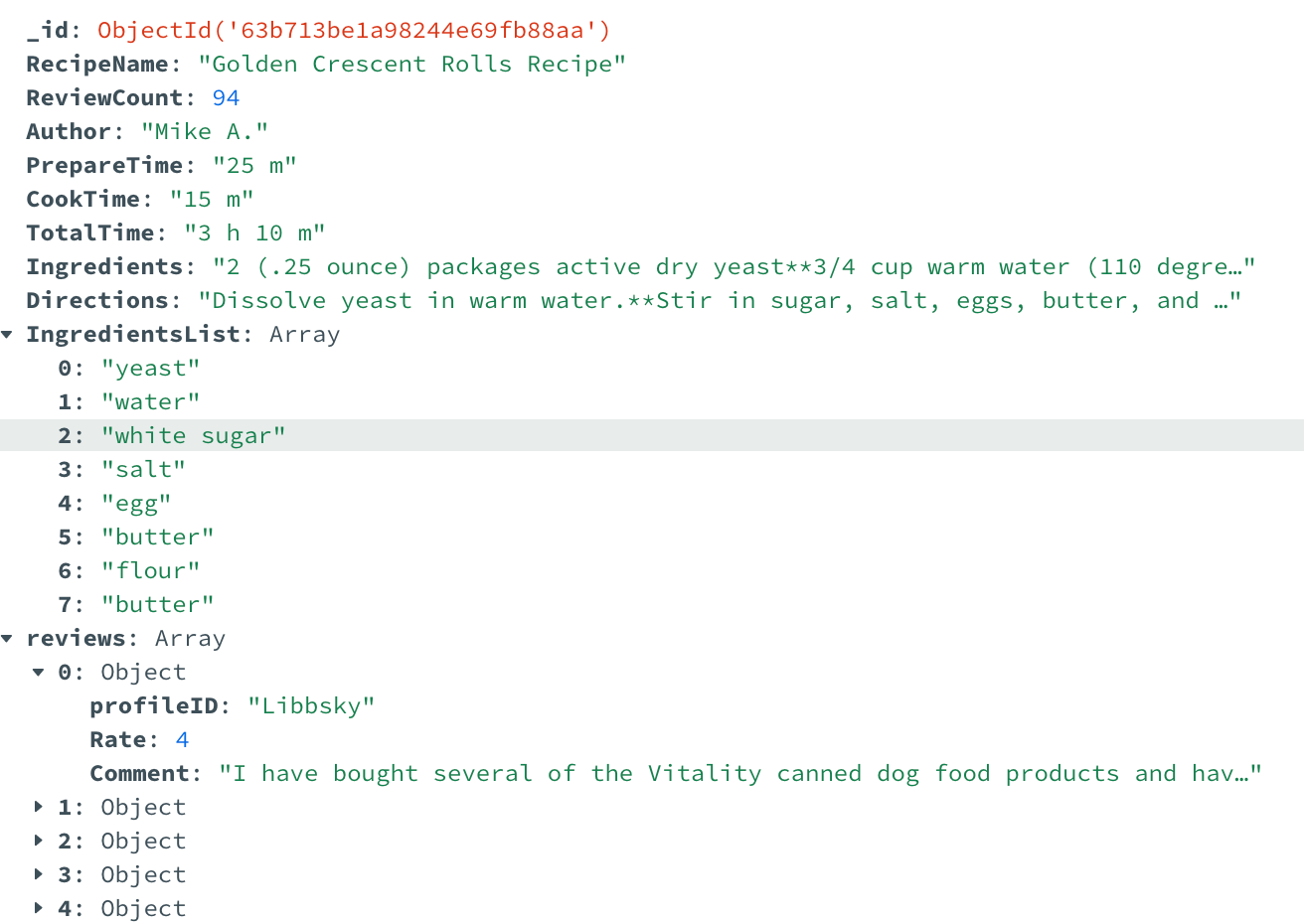
For the User collection we did not used any predefined dataset but we built it, downloading files containing a list of name, surname, country and randomly associate the three. Then for each user we created the email as ‘[firstname@lastname.com](mailto:firstname@lastname.com)’. The fridge is composed by a set fo triple composed by selecting ingredients, numbers which symbolizes the quantity present in the fridge, a date with symbolizes the expiring date of the ingredient.

The final size of the datasets used are: about 21mb for users, about 62mb for recipes and 0.2mb for ingredients.

**4.1. Data Model**There are 3 collection: User, Ingredient and Recipe.



In the image above there is an example of a json document representing a user. In addition to the username, password, email and other information characterizing the user, there is an embeded document regarding the products in the user's fridge in question.



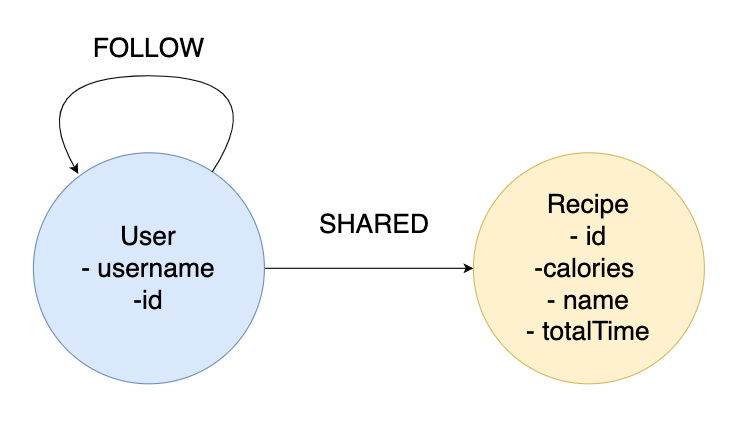
In the image above there is an example of a json document representing a recipe. Preparation time is when you'll be busy in the kitchen. You'll be mixing, mashing, stirring, and doing whatever else the recipe's instructions say to do. Cooking time is when the food is actually in the oven or on the stove top. Total time is the total amount of time that passes from when you start preparing to when you serve it on the table. The list of ingredients is also present, both with the quantities and as an embedded document, referring to the ingredients in general contained in it. Finally, there is an embedded document referring to all the comments in the recipe.



Finally, we see the collection relating to the ingredients, in which the various food characteristics are present, such as calories, measure, protein, fat, fiber, carbs and category.

**4.2 GRAPH DB**

The graph databases is used to avoid multiple joins that may be too computationally intensive.

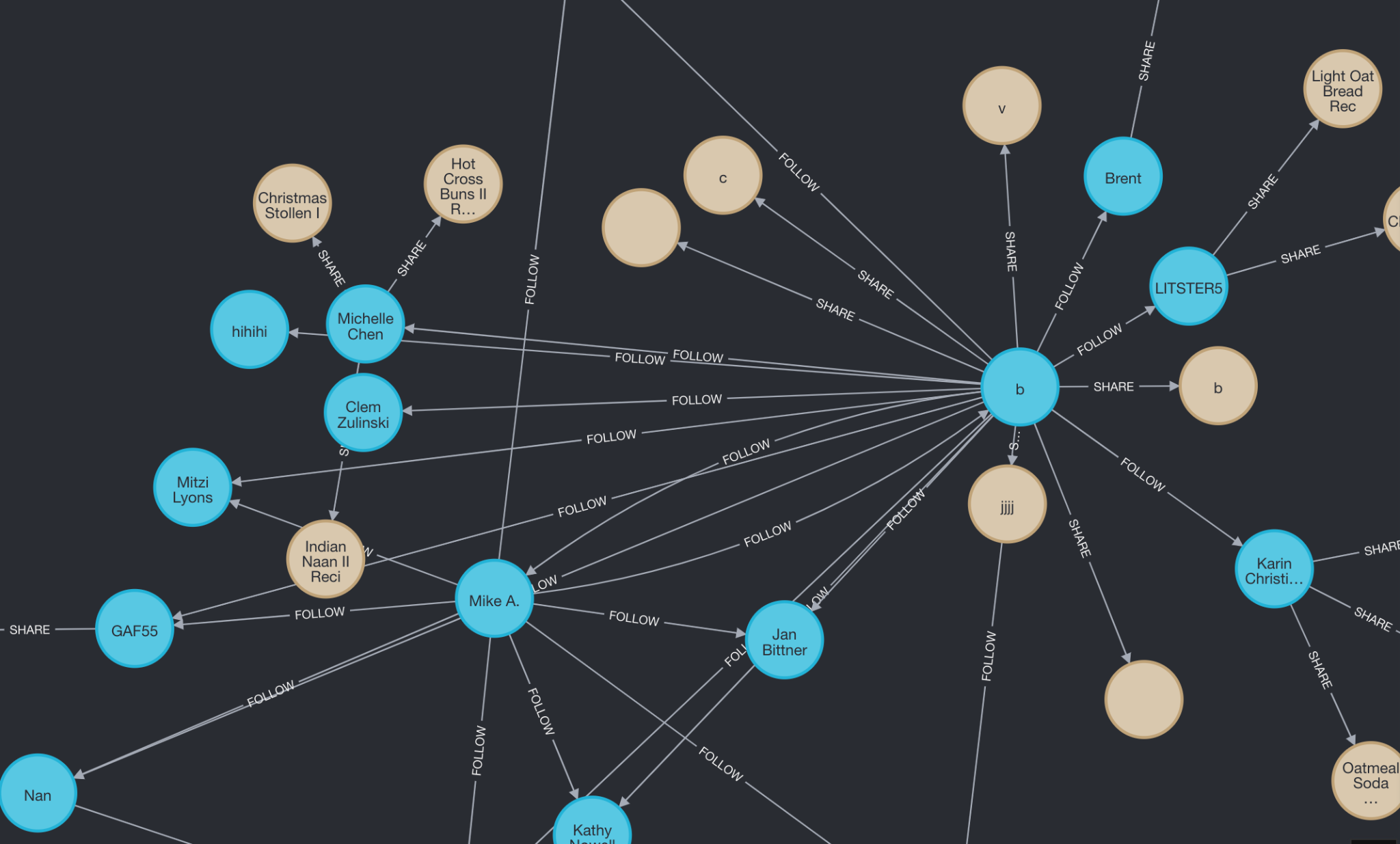


In the database there are 2 types of nodes:

* User, that represents a user inside the graph. The only attributes is username
* Recipe, that represents a recipe inside the graph. Its attributes are the number

The purpose of the graph db is to speed up certain operations. We opted for the graph db to manage the more "social" part of the smart fridge precisely because of its speed in certain operations. For example, when showing followed users or recipes posted by a user.

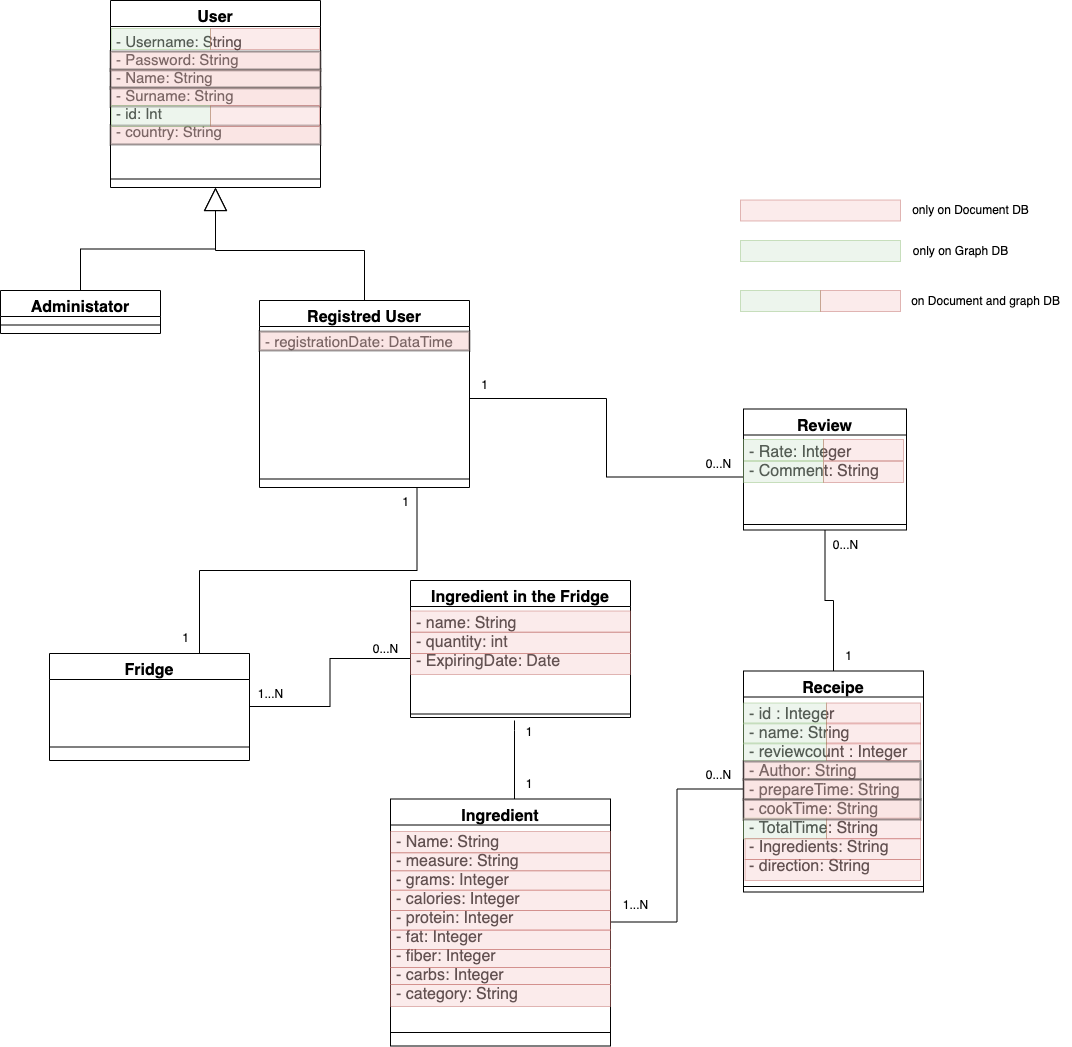
If a user follows another user, a relationship is created within the graph db between those two "follow" users. If a user publishes a recipe, a relationship is created in the graph db between that user and that "shared" recipe.

In the figure below we can see a preliminary snapshot of the graph db****

**4.3 DATA AMONG DBs**

For the sake of improving responsiveness and also being able to handle, in the most efficient way ,big sets of data we retrieve information from due databases.

As you can see from the diagram all the fields are stored in mondoDB and only some fields are stored also in the graph db. This does not mean that the graph is a surplus, because not using it won’t allow us to handle the network which a user can follow other and see the others’ recipes. In the graph alone, it’s something that cannot be seen from the diagram, are stored the recipe published by a user.



**5. SOFTWARE ARCHITECTURE**

The application is developed in Java using Intellij as IDE. The software used are:

* MongoDB and Neo4J ad NoSQL DBMS
* java: as object program oriented
* java fxml: as frontend tool used to help us building the GUI

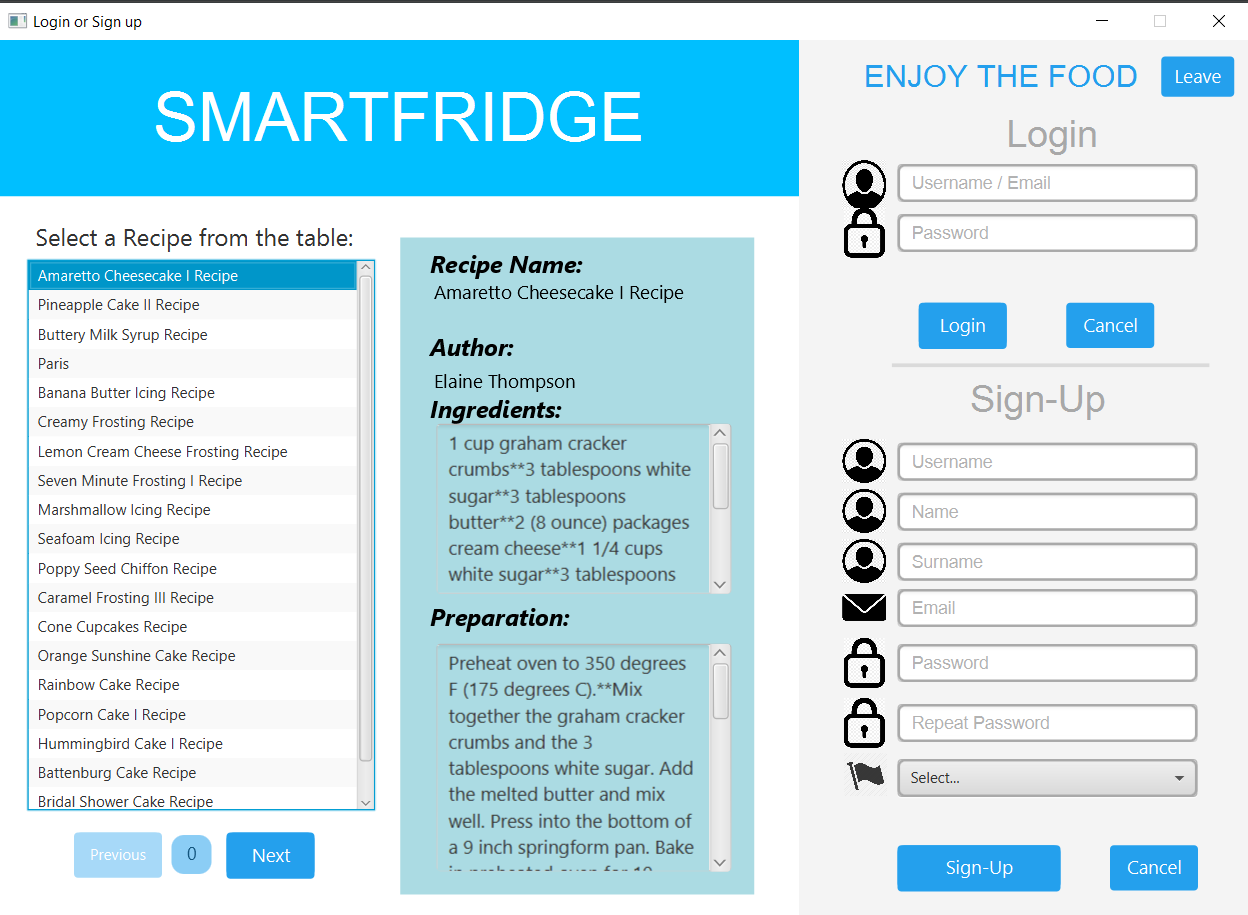
**5.1. Repository Structure**

riportare la struttura del codice specificando cosa contengono le varie cartelle

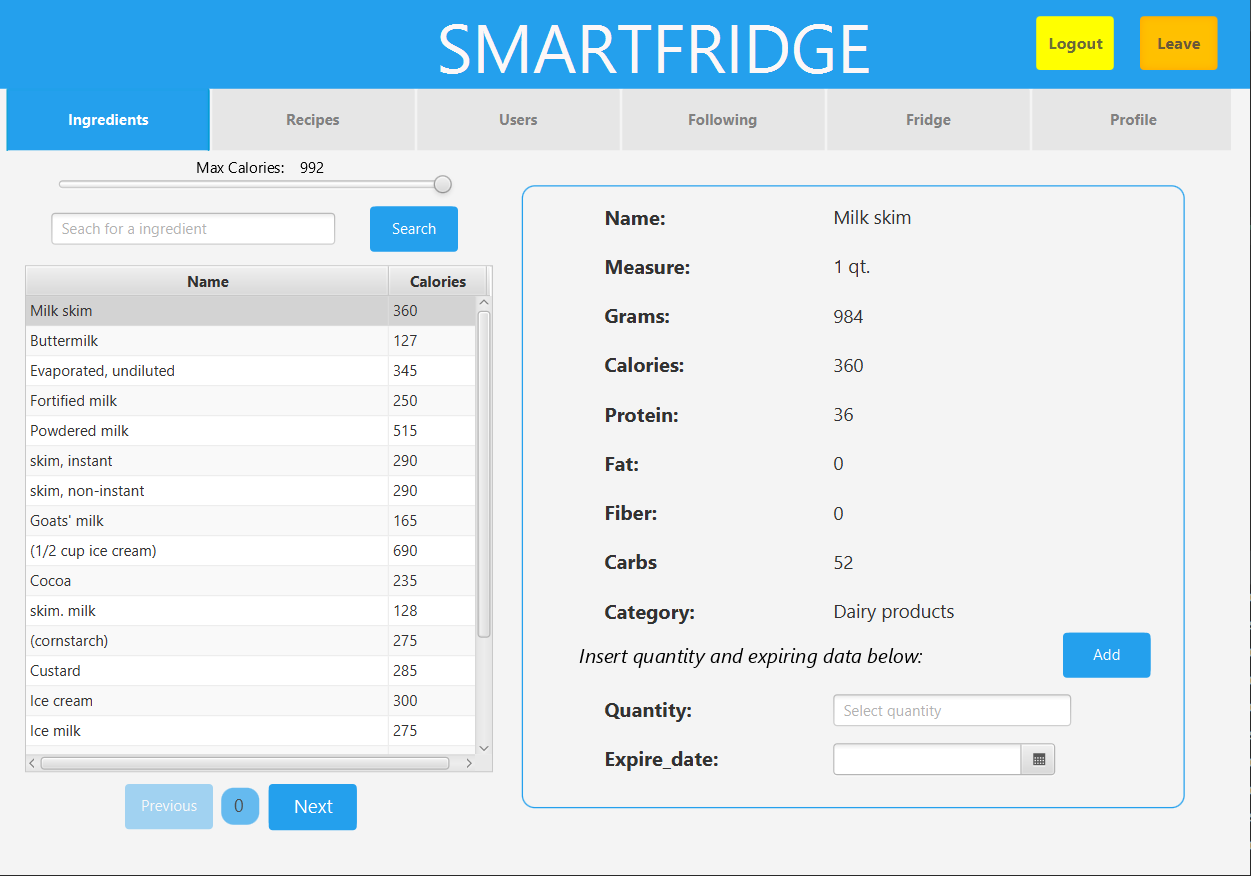
* **com.example.SmartFrige**
  + **controller:** contains java file that handle the various functionalities of the function
    - **AddIngredientToFridgeController:** script that handle the add of an ingredient, specifying the quantity and the expiring date, to the fridge
    - **AddRecipeController:** script that allows us to specify all the fields of a recipe and send the request to save it on the db
    - **AllCommentsController:** script that allows us to view all the comment of a given recipe
    - **AllIngredientsController:** script which shows all the information about the ingredient. We can search for ingredients by specifying in the specific input field the name. The ingredient are showed 20 by 20 by pushing the special button.
    - **AllRecipesController:** script which shows all the information about the recipe. We can search for recipes by specifying in the specific input field the name. The recipes are showed 20 by 20 by pushing the special button.
    - **AllUsersController:** script which shows all the information about the users. We can search for user by specifying in the specific input field the name. The user are showed 20 by 20 by pushing the special button.
    - **FollowedUserController:** script that allows to see all the users followed by the user that logged in. We can also unfollow a user
    - **FridgePageController:** Script that allows us to see all the ingredient saved in fridge. We can also modify the quantity and/or remove them
    - **HomePageAdminController:** script that handle the home page of the admin
    - **HomePageController:** script that handle the home page of the admin
    - **LoginController:** script that handles the login page
    - **ModifyRecipeController:** Script that handles a page which we can modify recipe information
    - **ProfilePageController:** Script that handle a page which are showed all the information of the user logged in
    - **ViewRecipeController:** Script that allows us to view the specifics of the clicked recipe
  + **DAO:** Scripts that handles the dialog with the db
    - **aggregationsMongo:** script in which are implemented all the aggregation in mongo
    - **FridgeDAO:** script in which are specified all the function to add/update/delete/retrieve ingredient in fridge (the collections accessed is the “User” one)
    - **IngredientDAO:** Script in which are specified all the function to add/update/delete/retrieve ingredients (the collections accessed is the “Ingredient” one)
    - **IngredientInTheFridgeDAO:** script in which are specified all the function to add/update/delete/retrieve ingredient in fridge, act on the single entity. The fridgeDAO acts on all the ingredient in the fridge
    - **RecipeDao:** script in which are specified all the function to add/update/delete/retrieve recipe in fridge (the collections accessed is the “Recipe” one. We also access to the graphDB)
    - **UserDAO:** script in which are specified all the function to add/update/delete/retrieve user in fridge (the collections accessed is the “User” one. We also access to the graphDB)
  + **DbMaintanance:** contains classes that handles connections with the dbs
    - **MongoDbDriver:** Script that the document db connection
    - **Neo4jDriver:** Script that the graph db connection
  + **DTO:** contains all the dto class
    - **fridgeDTO:** this class stores all the information of a fridge that matches with the field of the ones present in MongoDB
    - **IngredientDTO:** this class stores all the information of a ingredient that matches with the field of the ones present in MongoDB
    - **IngredientInTheFridgeDTO:** this class stores all the information of a ingredient, present in the fridge, that matches with the field of the ones present in MongoDB. This class has specified the name of the ingredient, the related quantity and expiring date
    - **RecipeDTO:** this class stores all the information of a recipe that matches with the field of the ones present in MongoDB
    - **ReviewDTO:** this class stores all the information of a review that matches with the field of the ones present in MongoDB
    - **userDTO:** this class stores all the information of a user that matches with the field of the ones present in MongoDB
  + **model**
    - **Fridge:** this class stores all the information of a ingredient, present in the fridge
    - **Ingredient:** this class stores all the information of a ingredient
    - **IngredientInFridge:** this class stores all the information of a ingredient, present in the fridge
    - **Recipe:** this class stores all the information of a recipe
    - **RegisteredUser:** this class stores all the information of a user that is registrated
    - **Review:** his class stores all the information of a review
    - **SimpleLocalDateProperty:** class that allows us to give the SimpleProperty characteristics to the LocalDate class cause to be used in a table view we need it
    - **User:** his class stores all the information of a user
  + **utils**
    - **utils:** class that contains utils functions
  + **Application.java:** java class with main method
* **com.example.SmartFridge:** contains all the fxml file of the application. It’s the GUI directory
  + **AddIngredient.fxml:** specifies the structure of the page where to add an ingredient
  + **AddRecipe.fxml:** the structure of the page in which you can insert new recipe
  + **AllComments.fxml:** specifies how to render comments when showed
  + **AllIngredient.fxml:** specifies the page that render the table that shows all the ingredients
  + **AllRecipes.fxml:** specifies the page that render the table that shows all recipes and the button that trigger main action
  + **AllUsers.fxml: s**pecifies the page that render the table that shows all the users and a textfield used to search a user
  + **FollowedUser.fxml:** specifies the page that render the table that shows the followed users
  + **FridgePage.fxml:** specifies the page that render the table that shows all the product currently available in the fridge and buttons to operate it
  + **HomePage.fxml:** specifies the main page after the login
  + **HomePageAdmin.fxml:** specifies the main page after the login for the admin
  + **LoginPage.fxml:** specifies the first page where user can signup/login
  + **ModifyRecipe.fxml:** specifies the page where user can modify a recipe
  + **ProfilePage.fxml:** specifies the page where user can see his information and modify it
  + **ViewProduct.fxml:** specifies the page where user can see a product information and add it to his own fridge or remove it
  + **ViewRecipe.fxml:**specifies the page where user can add a new recipe

**6. INSTRUCTION MANUAL**

6.1. How to use SmartFridge - user side

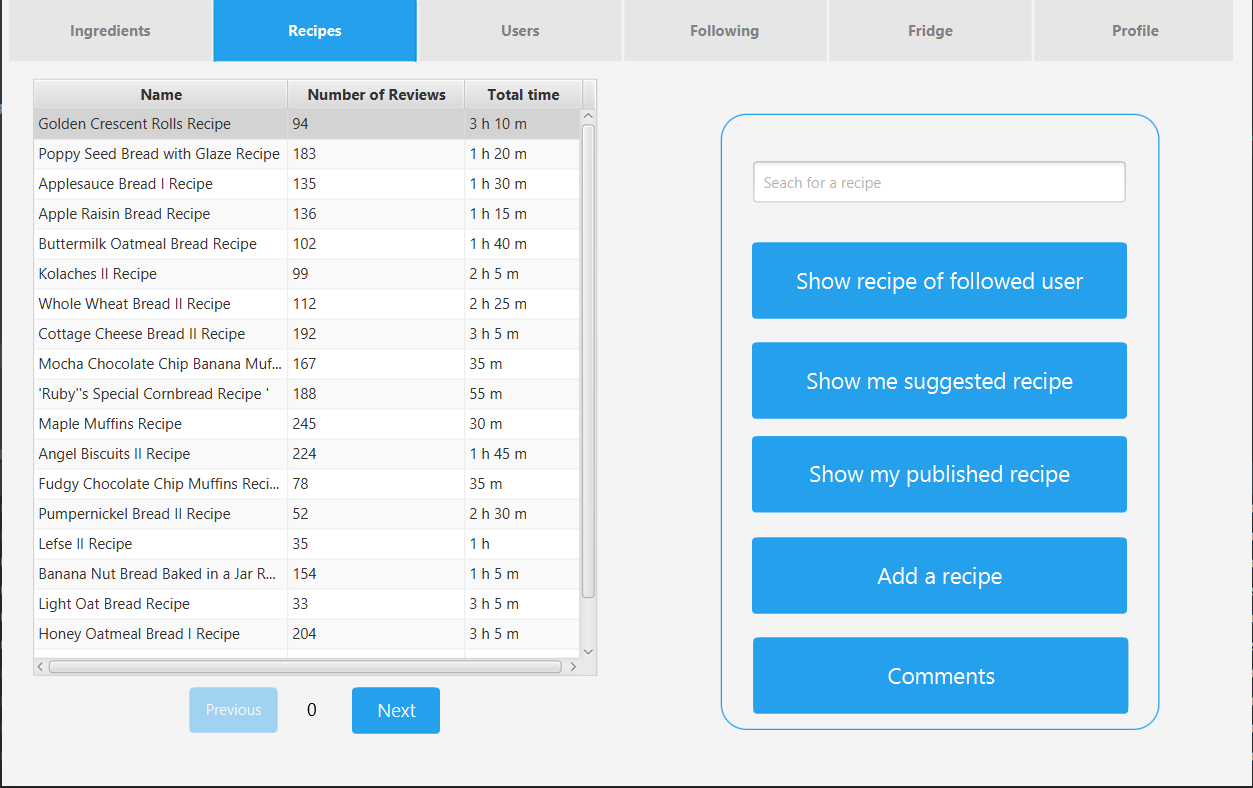


This is the initial page of the application, where you can register (if you don't have your own profile yet) or where you can log in or view recipes. To register, you must enter your name, surname, your own username, email, password and nationality.

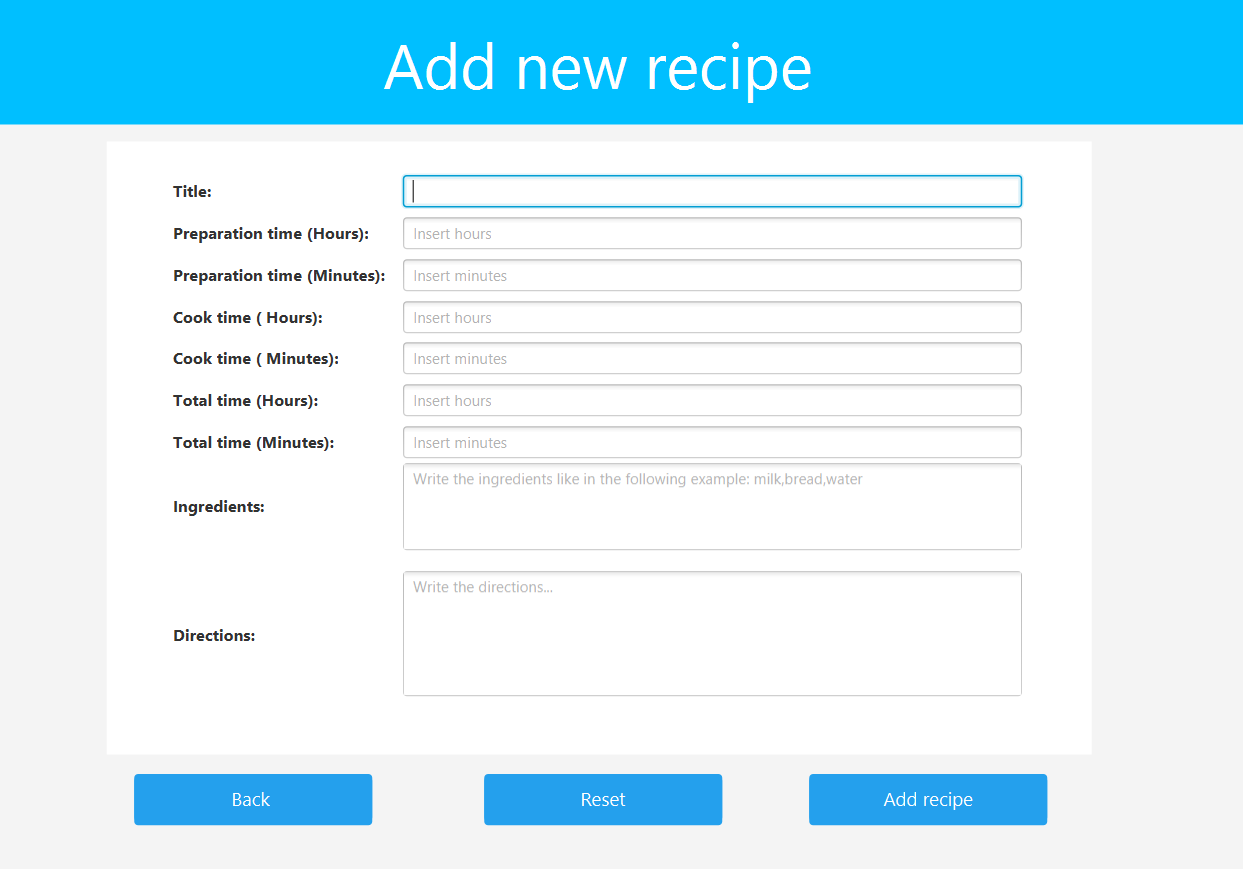


The home page of the application looks like in the photo above. It is possible to log out, to return to the login page. Then there are six buttons through which you can access: the list of all products, the list of all recipes, the list of all users, your fridge, the users you follow, your profile information.

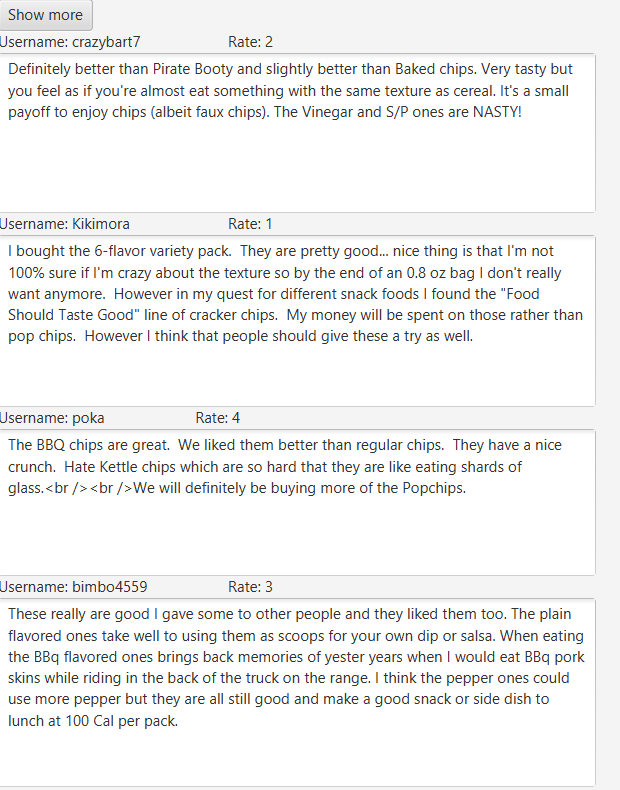
The list of all products is visible. The products are displayed 20 at a time, and pressing "show more" displays the next 20 and so on. By pressing 2 times on a product the list of its details is displayed, and by adding quantity and expiry date it is possible to add the product to your fridge.



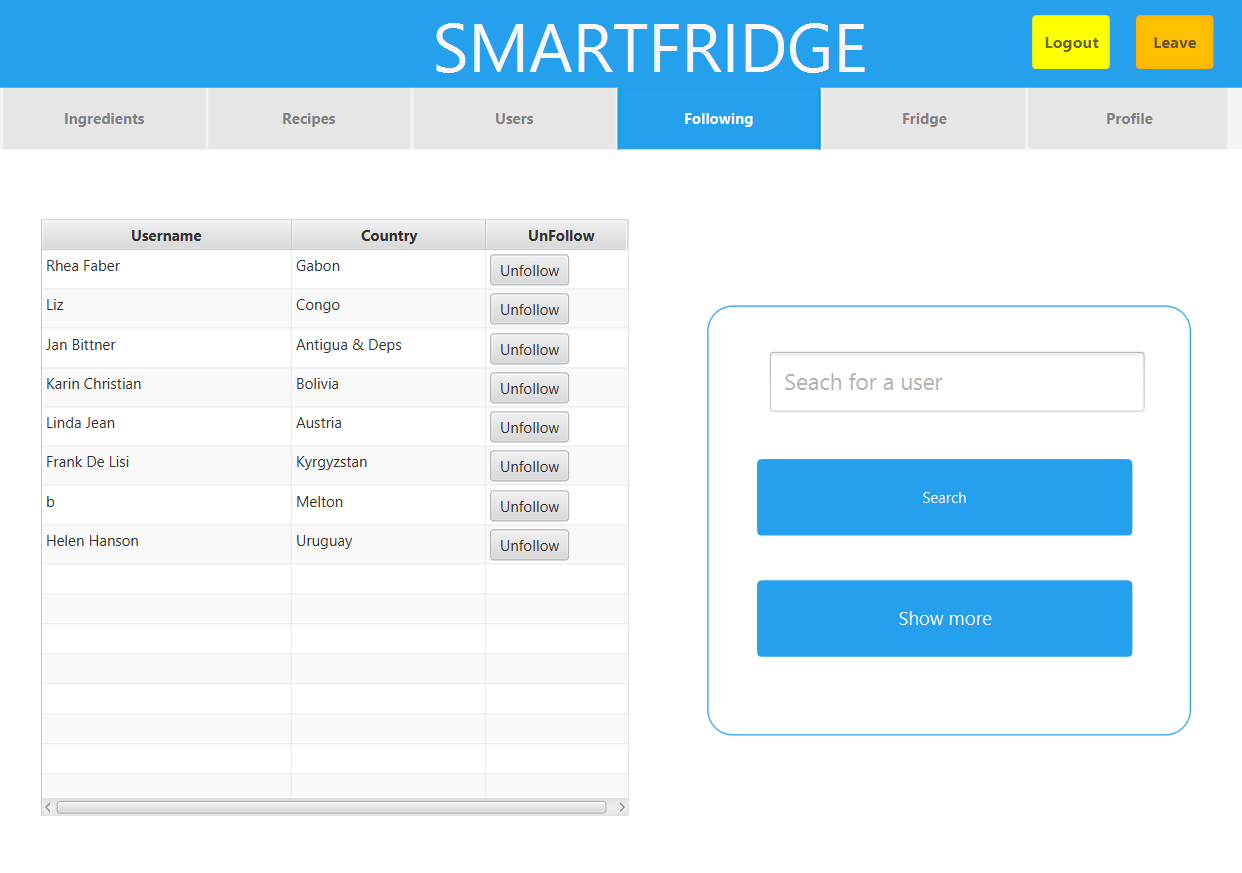
There is a list of all recipes. You can choose whether to view published recipes, suggested recipes or view recipe details. Also this time 20 are shown at a time and by pressing on "show more" the next 20 are displayed



Pressing on "add recipe" the following page is displayed, in which to fill in the fields with the details of the recipe that you intend to insert in the db.

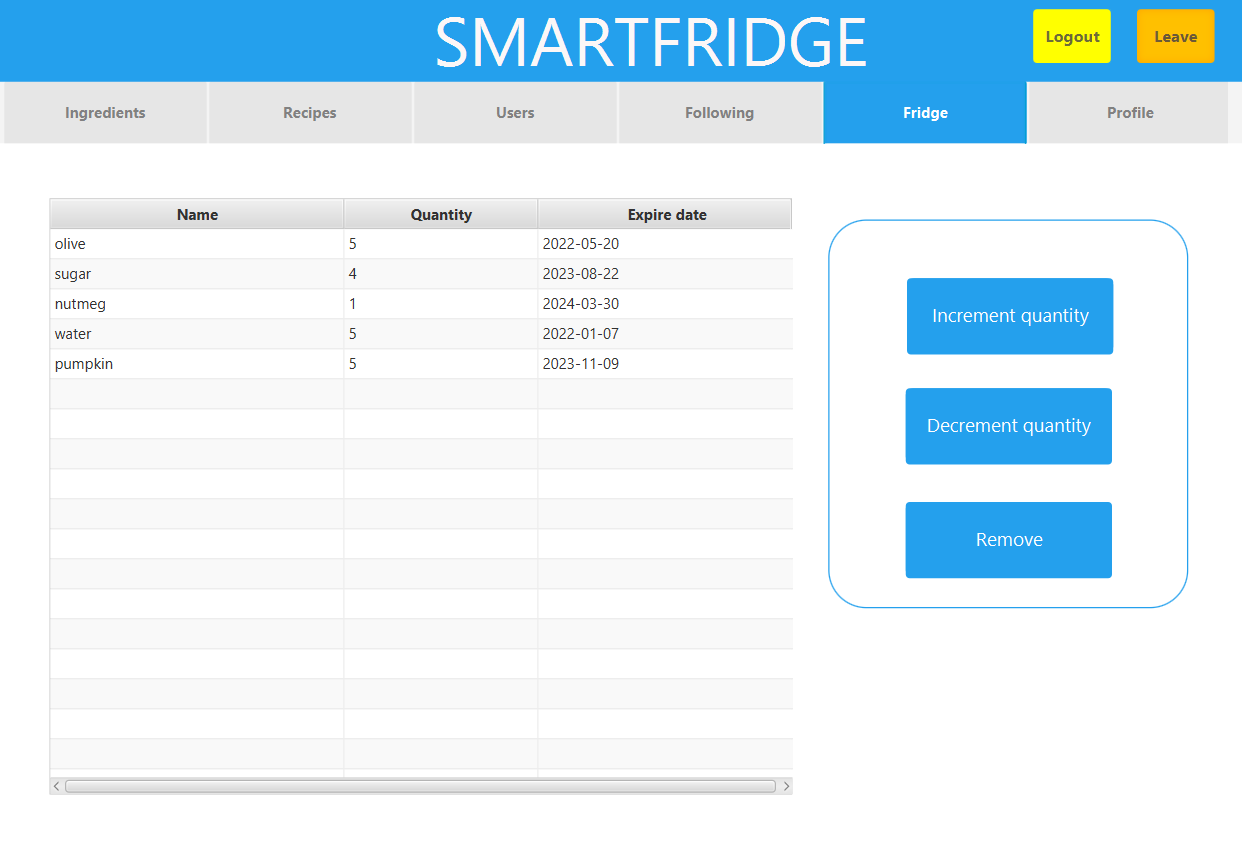


Otherwise you can go to view the details of the recipe and you can add a comment and an rate relating to that recipe, as well as view the comments made by other users to that recipe.

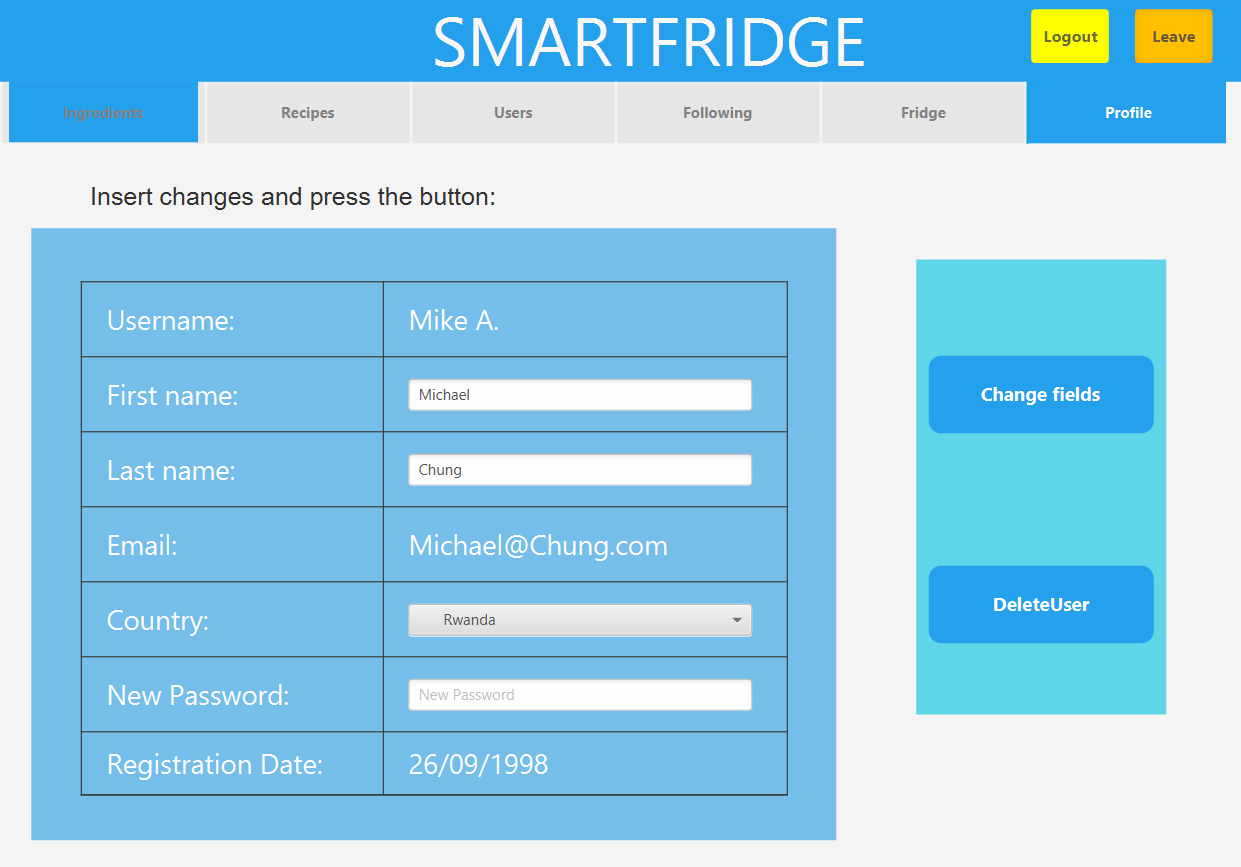


it is then possible to view the list of all users. It is possible to follow a selected user or view suggested users. Again, 20 users are displayed at a time, scroll forward in the list by pressing "show more".

It is also possible to view the list of followed users, and for each one choose to unfollow them.



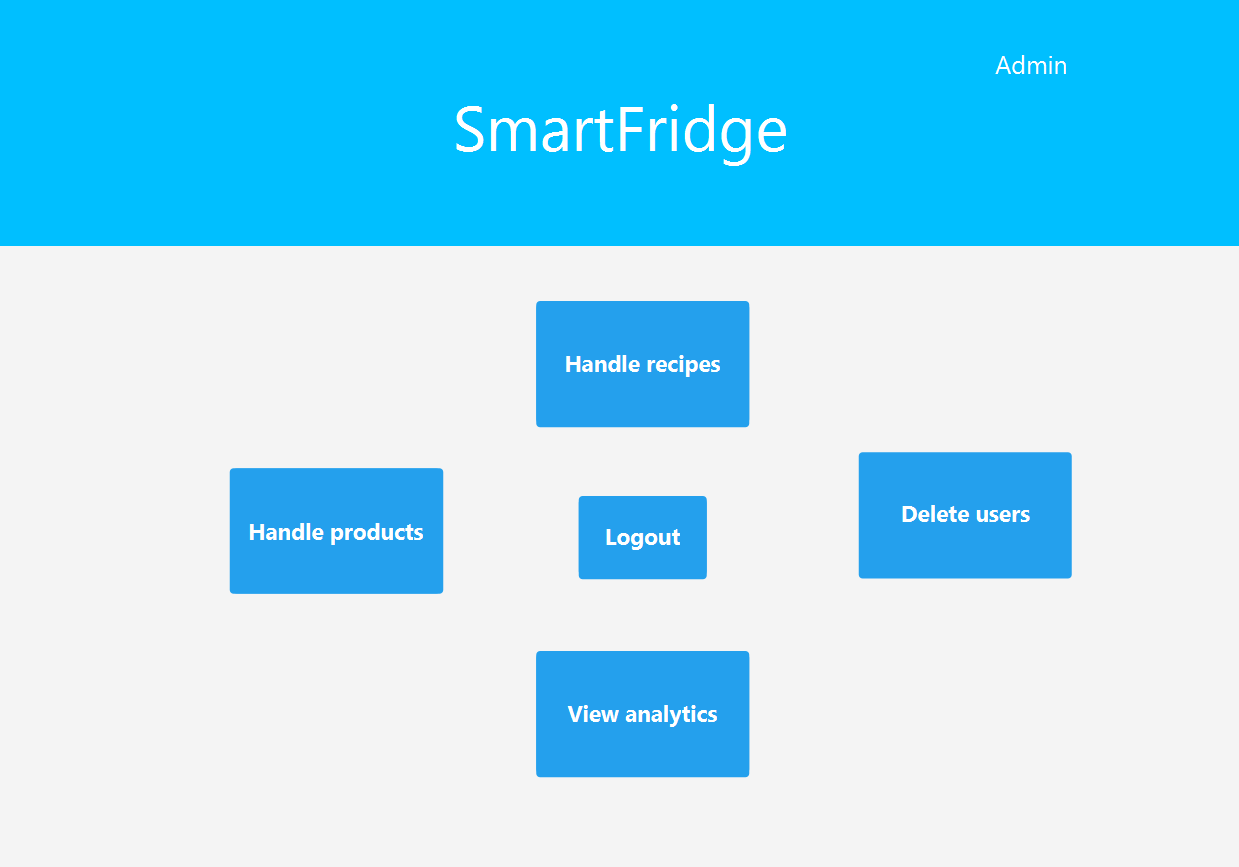
You can then view the status of your fridge: for each product in the fridge, you can view the name, quantity and expiry date. By pressing on a product it is possible to increase or decrease its quantity, or remove it.



Finally, you can view the details of your profile, change your password or delete your profile.

6.2. How to use SmartFridge - admin side

If, on the login page, you log in with the credentials of "admin" as username and "admin" as password, the admin home page will appear, and the superimposed name will be "admin". In this case, we are logged in as an administrator and have more privileges than a normal user.

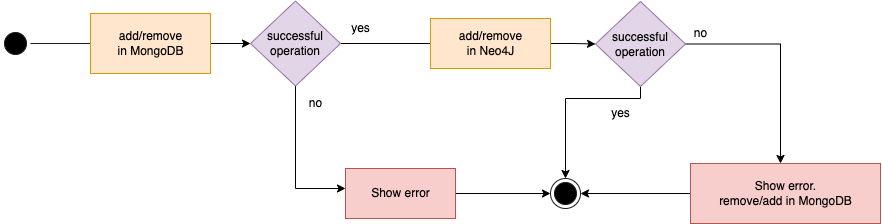


The admin will be able to remove a user, or delete/edit/add ingredients and delete/modify recipes. In recipes, the admin can also delete comments.

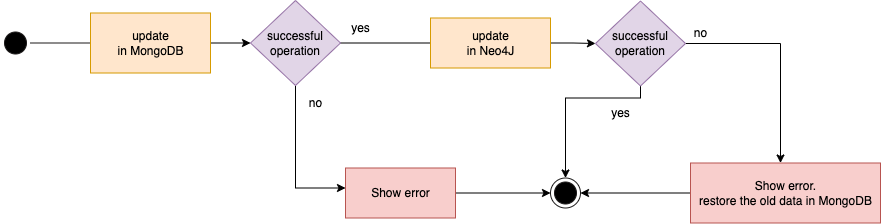
**7. Database consistency management, Aggregation, Indexes   
 7.1 Database consistency management**

The use of to different databases imposes to us to define database consistency management strategy.

The ones adopted for handling problems due to removing/adding user/recipe is explained in the following uml activity diagram.

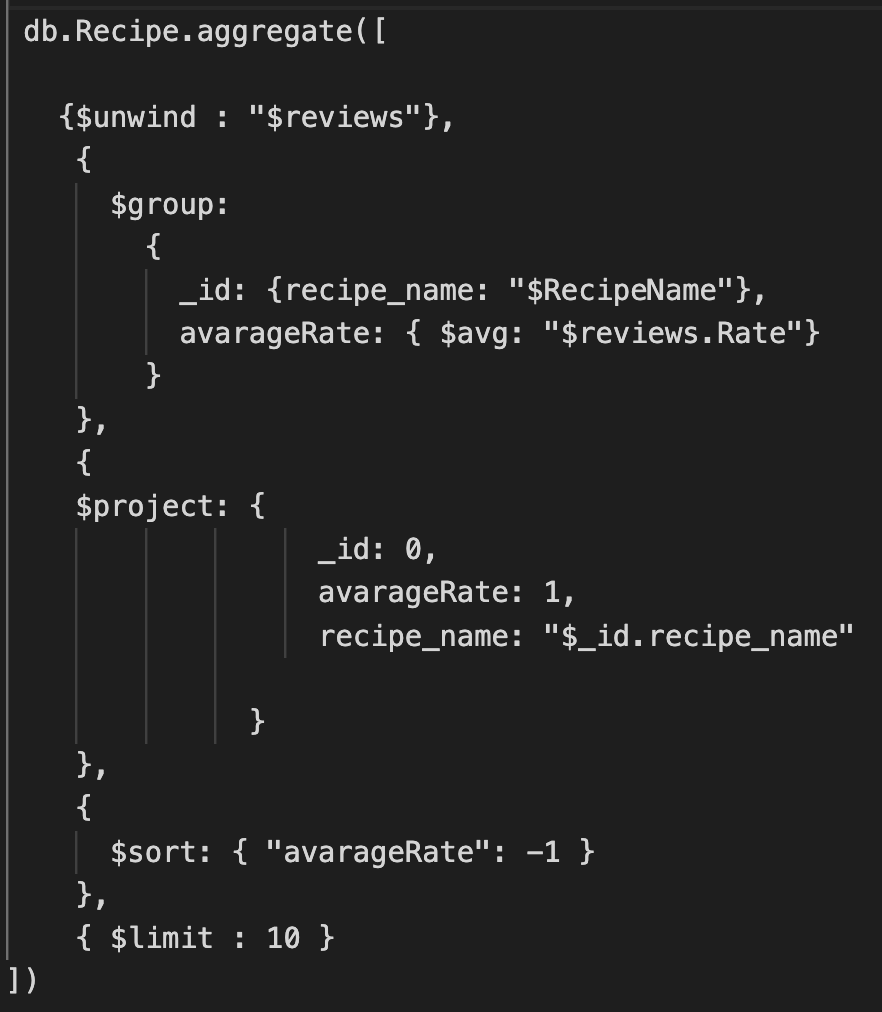


For the update operation the following uml activity diagram explains the strategy adopted

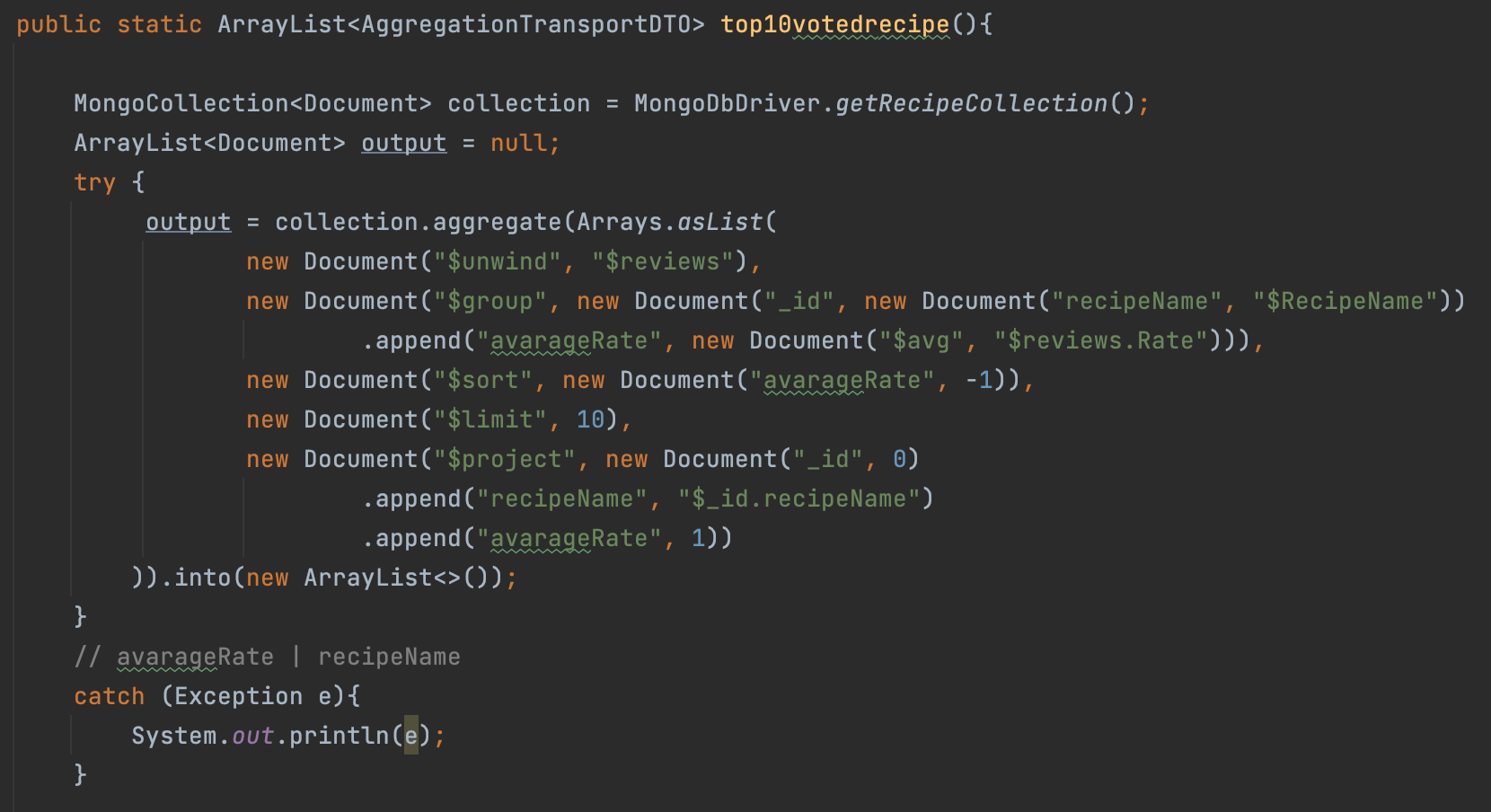
****

**7.2. Aggregations**

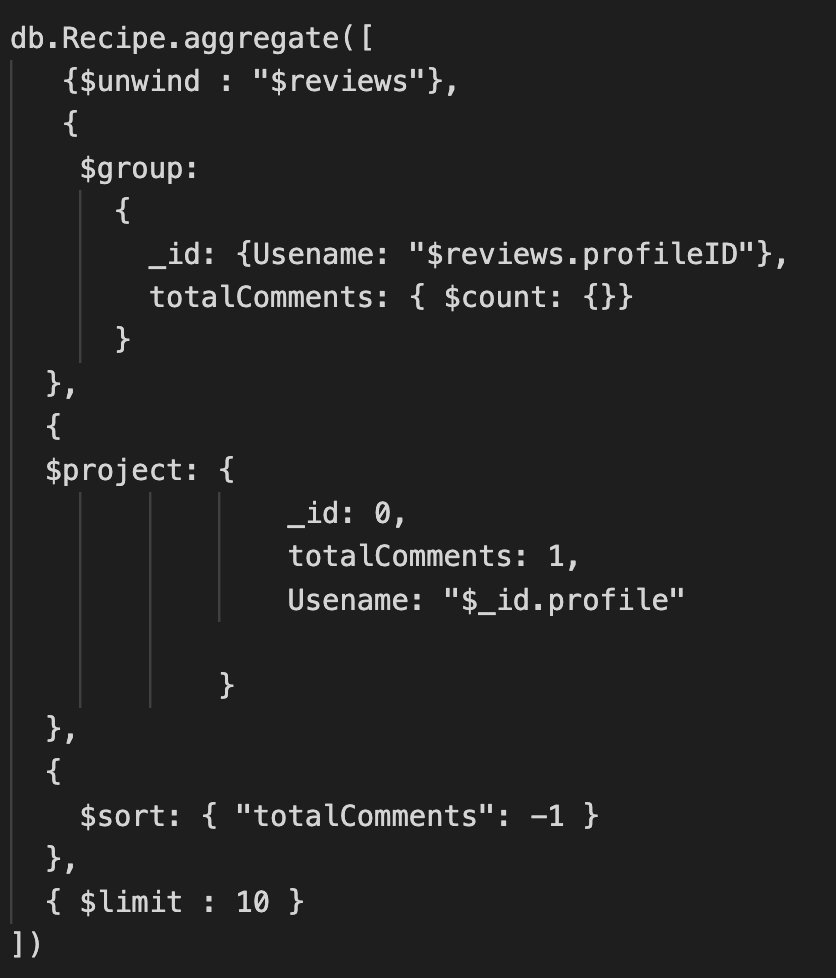
**7.2.1 (top 10 voted recipe)**

****

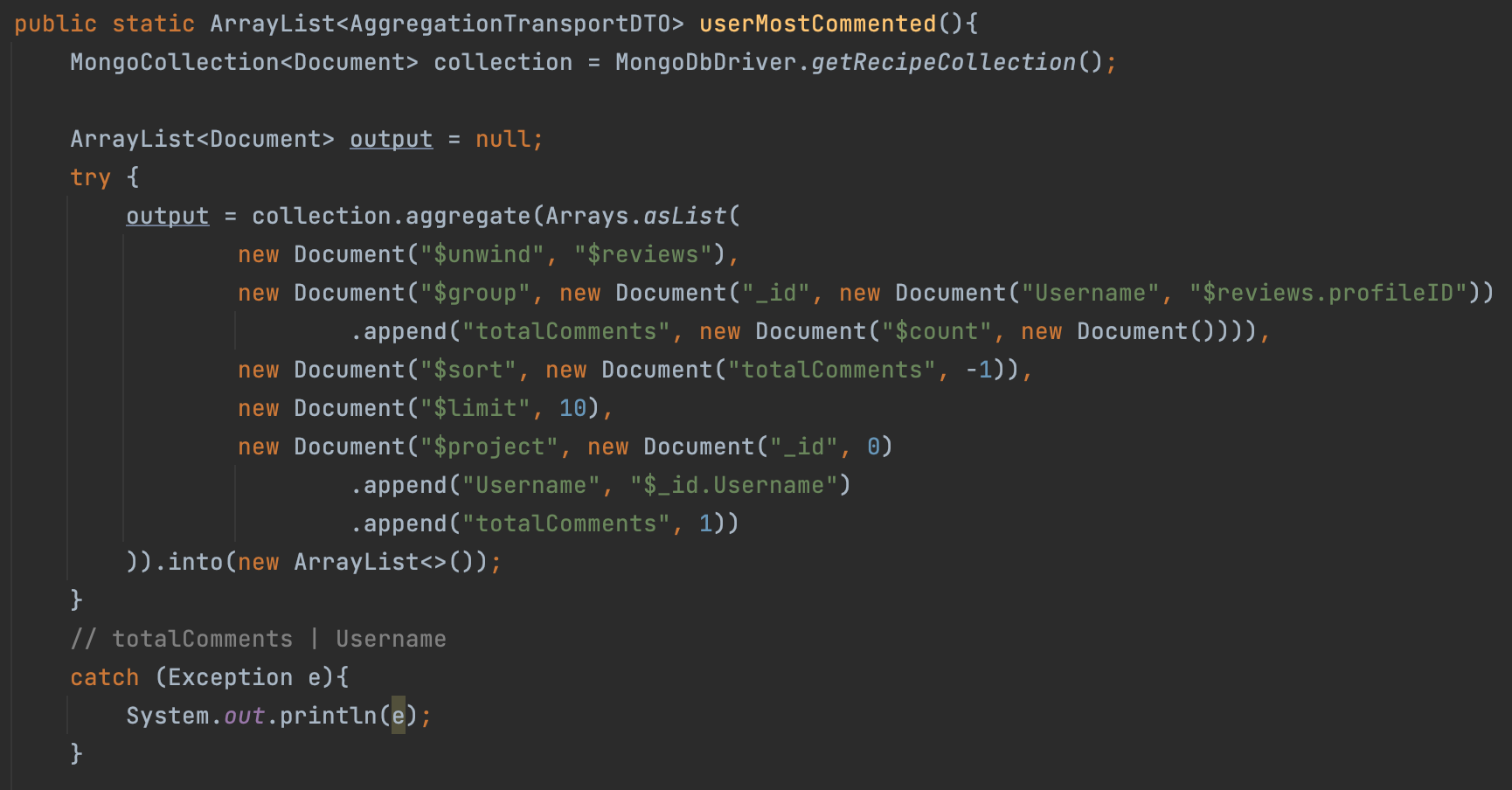
The result in java is:

****

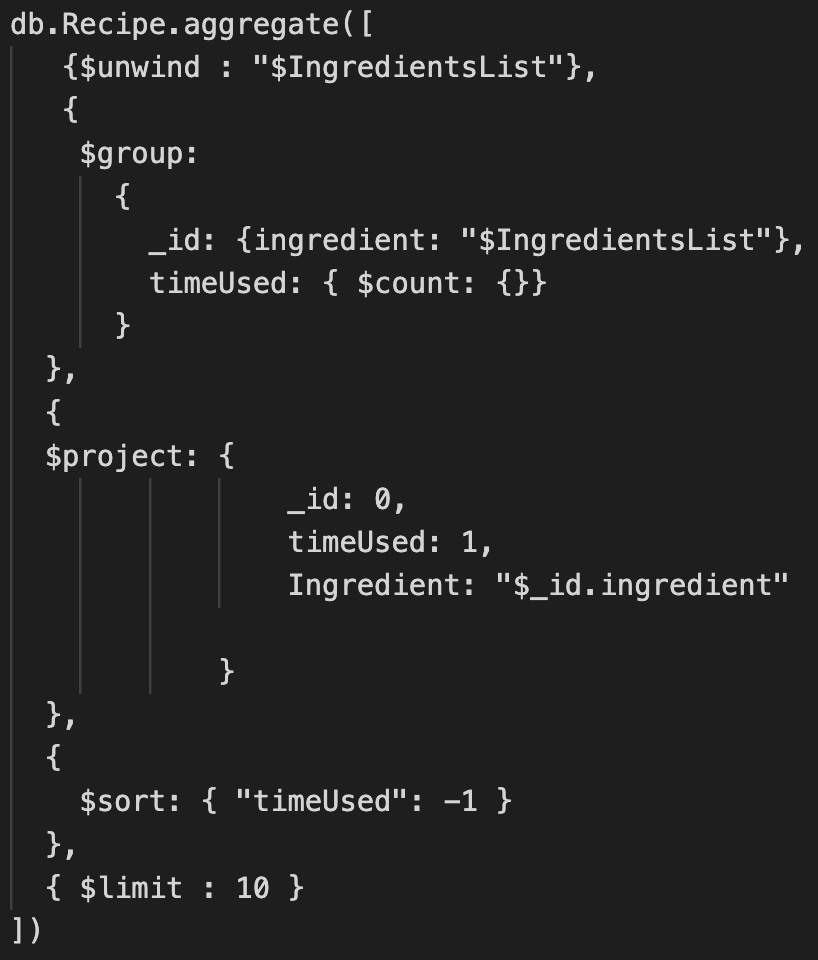
**7.2.2 (users who commented the most)**

****

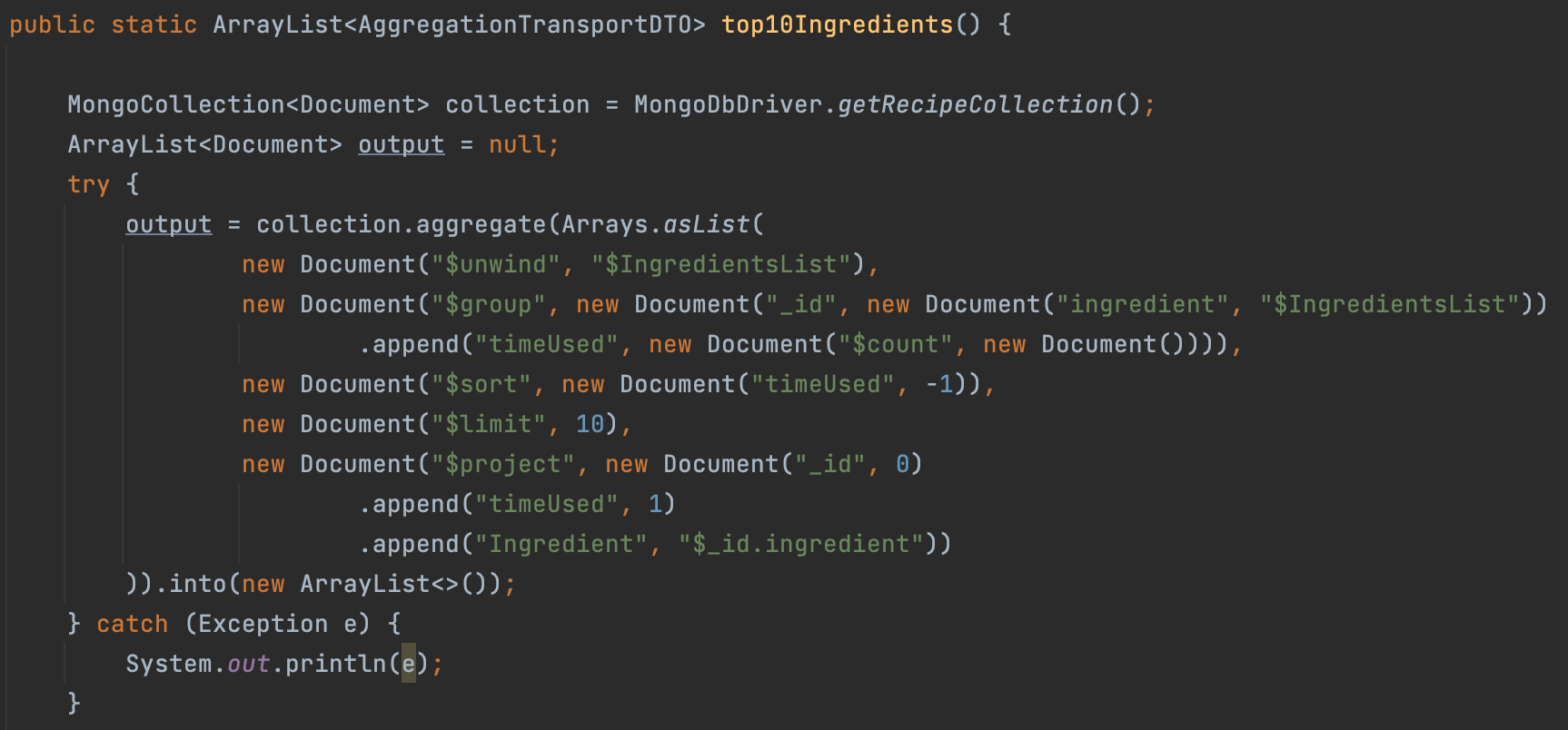
The result in java is:

****

**7.2.3 (top 10 most used ingredients in recipes)**

****

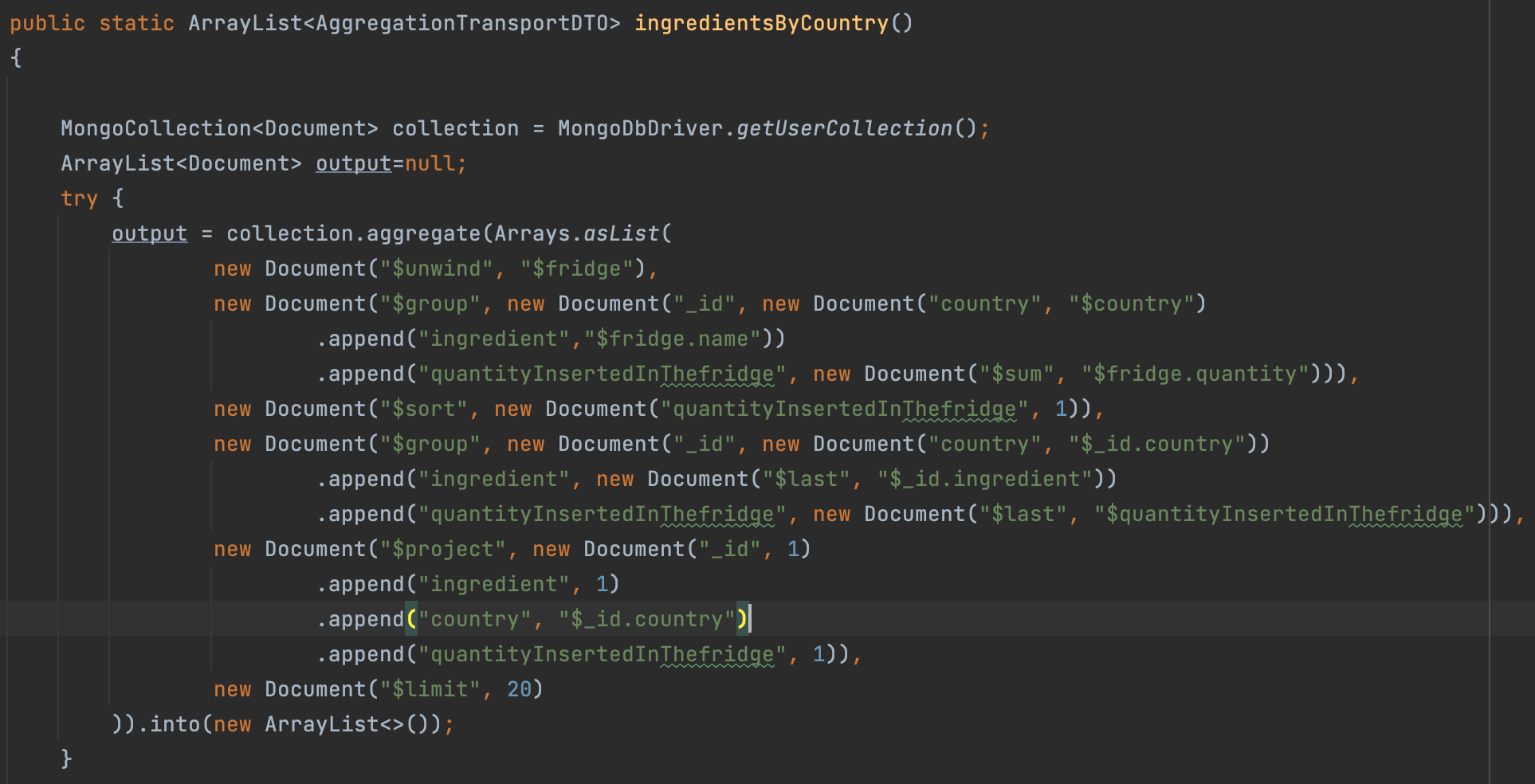
The result in java is:



**7.2.4 (ingredient most put in the fridge by country)**

****

The result in java is:



**7.3. Test and Statistical Analysis (indexes)**

we are gonna report some of the main query issued and the corresponding usage frequency:

| Query | Frequency |
| --- | --- |
| search ingredient by name | MEDIUM |
| search recipe by name | HIGH |
| search user by name | HIGH |
| modify quantity of an ingredient in the fridge | MEDIUM |
| visualize comment | LOW |

Our application is a read heavy one, and we implemented, in order to speed up the execution of some query, 4 indexes argued below.

Indexes are a tool to be used to helps the execution of query, in our case, of MongoDB.

We specified four different indexes

* for the User collection we specified an index on the field Username. Statistics of performing the query

*“db.User.find({username: “user\_name” })”*

| use of index | Docs examined | keys examined | exectionTimeMillis |
| --- | --- | --- | --- |
| no | 43836 | 0 | 41 |
| yes | 1 | 1 | 1 |

* for the Recipe collection we specified an index on the field RecipeName. Statistics of performing the query

“*db.Recipe.find({RecipeName: “recipe\_name” })*”

| use of index | Docs examined | keys examined | exectionTimeMillis |
| --- | --- | --- | --- |
| no | 12086 | 0 | 15 |
| yes | 1 | 1 | 0 |

* for the Ingredient collection we specified an index on the field food. Statistics of performing the query

“db.Ingredient.find({calories : {$gt : 400 ”}})”

(this index is not used in the application but it can be useful for additional implementation such as search ingredient with calories which surpasses a value)

| use of index | Docs examined | keys examined | exectionTimeMillis |
| --- | --- | --- | --- |
| no | 334 | 0 | 3 |
| yes | 27 | 27 | 0 |

* for the User collection we specified an index on the field “fridge.name”. Statistics of performing the query

“ *db.User.find({"fridge.name":"olive"})*”

| use of index | Docs examined | keys examined | exectionTimeMillis |
| --- | --- | --- | --- |
| no | 43837 | 0 | 241 |
| yes | 3721 | 3721 | 17 |

Commands to create indexes in mongo shell

db.User.createIndex({username : 1})

db.Recipe.createIndex({RecipeName : 1})

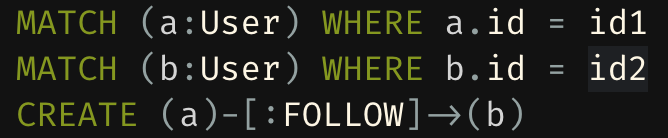
db.Ingredient.createIndex({calories: 1})

db.User.createIndex({"fridge.name":1})

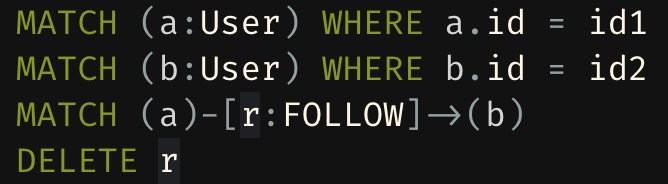
**7.4 typical “on-graph” query** (the query for each one are specified below the table)

| # | domain-specific queries | graph-centric queries | details |
| --- | --- | --- | --- |
| 1 | follow other user | create the relationship between two entity user |  |
| 2 | unfollow other user | delete the relationship between two entity user |  |
| 3 | create user | create node User |  |
| 4 | delete user | delete node User and all of his relationships |  |
| 5 | create a recipe | create a node and a relationship between that node and the author node | the author node is the the node corresponding to the user that shared that recipe |
| 6 | delete recipe | delete a node and a relationship between that node and the author node |  |
| 7 | show suggested user | find the user followed by the ones followed by the user who issues the request | if Mike follows Carla that follows Giorgio, if Mike issues a query, the result will be Giorgio |
| 8 | show suggested recipe | find the recipe node that has a relationship with the user followed | recipe that are shared by the followed user |
| 9 | show top 3 influencer | find the User entity with the highest degree |  |

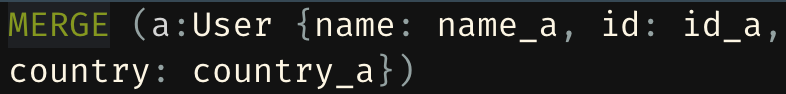
1



2



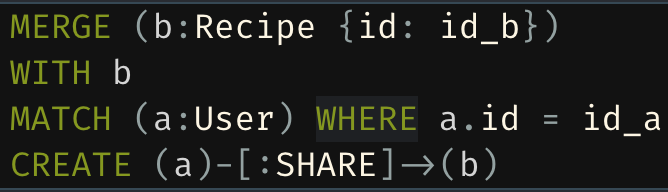
3



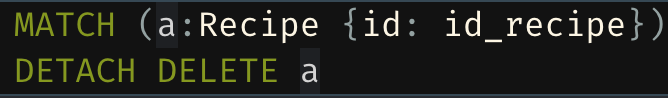
4



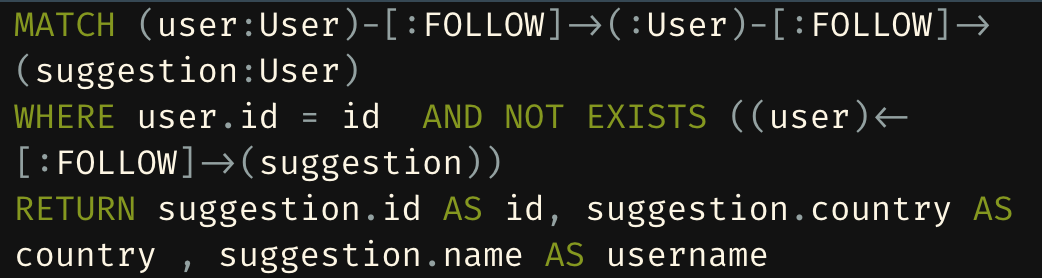
5



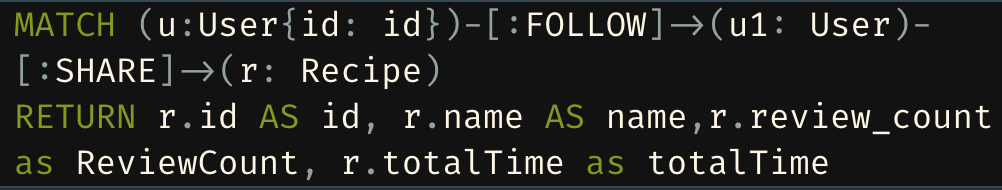
6



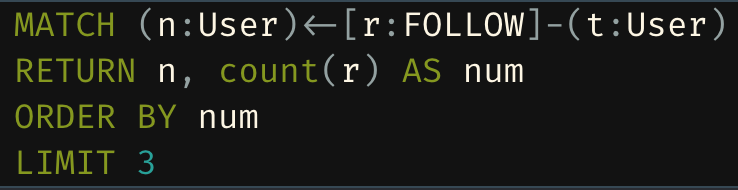
7



8

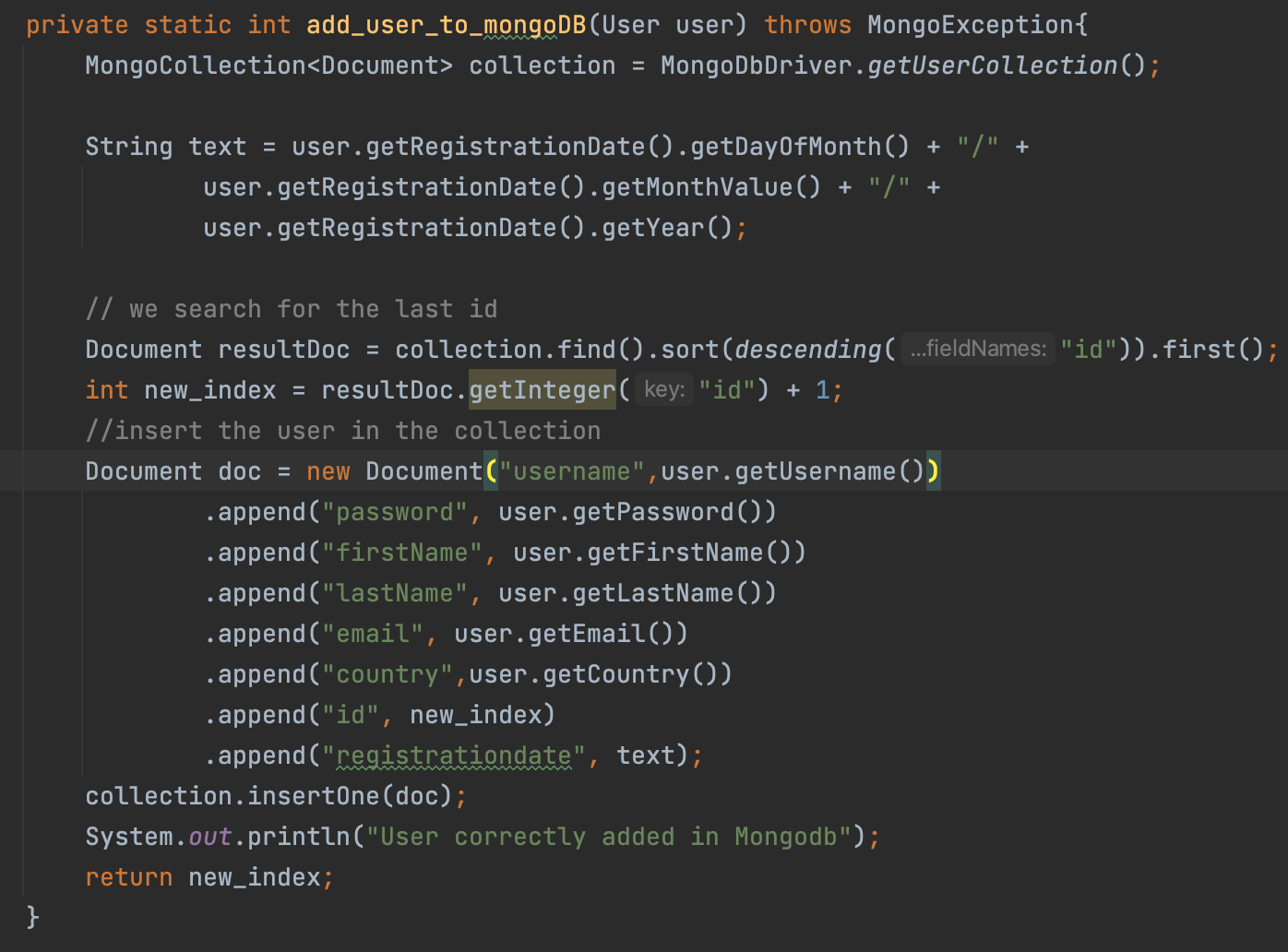


9



**7.5 CRUD operation for MongoDB**

* **create user**

****

db.User.insertOne( {

username: "username",

password: "password",

firstName: "firstname",

lastName: "lastName",

email: "email",

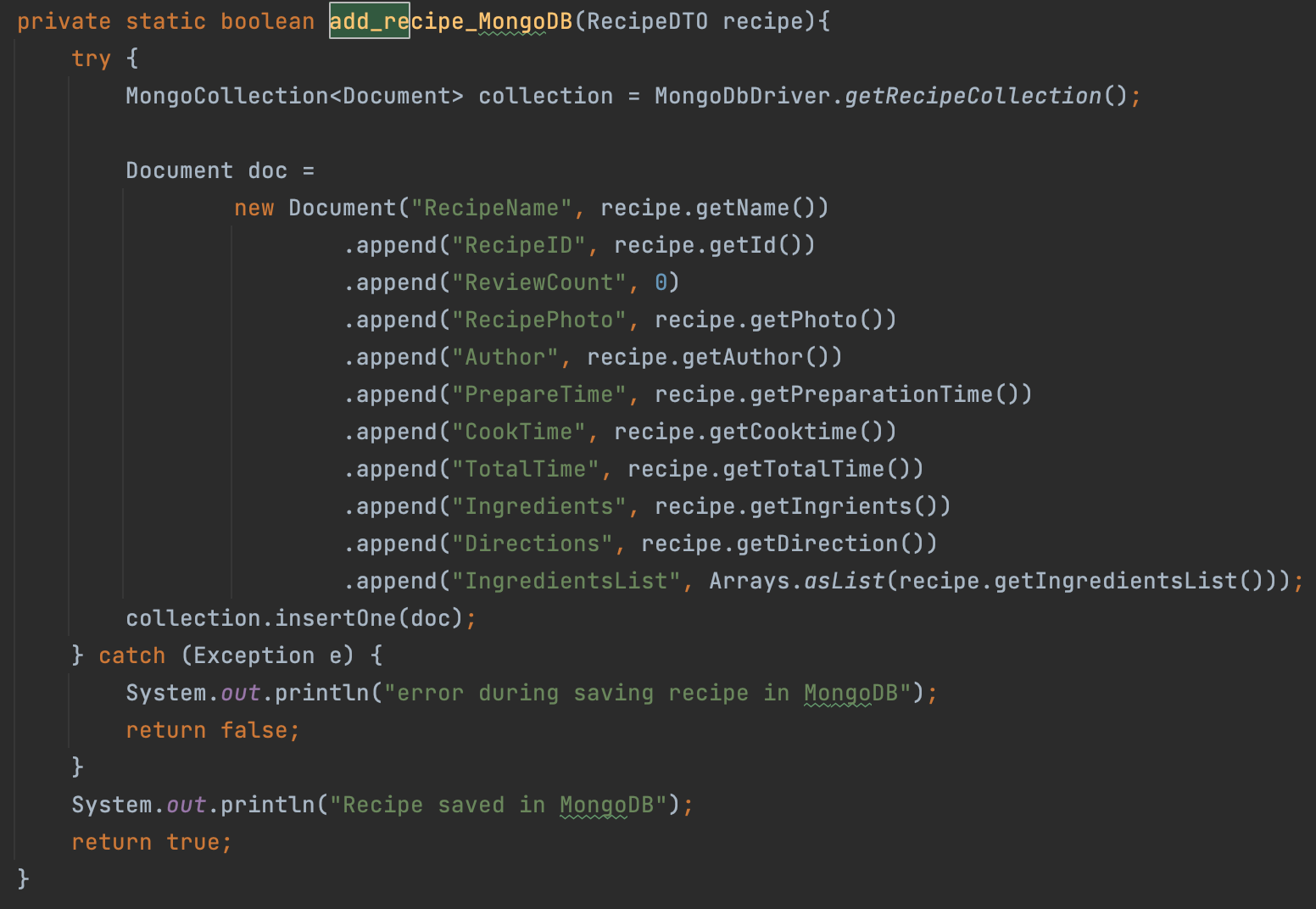
country: "country",

id: "id",

registrationdate: "registrationdate"

} )

* **create recipe**

****

db.Recipe.insertOne( {

RecipeName: "RecipeName",

RecipeID: "RecipeID",

ReviewCount: "ReviewCount",

Author: "Author",

PrepareTime: "PrepareTime",

CookTime: "CookTime",

TotalTime: "TotalTime",

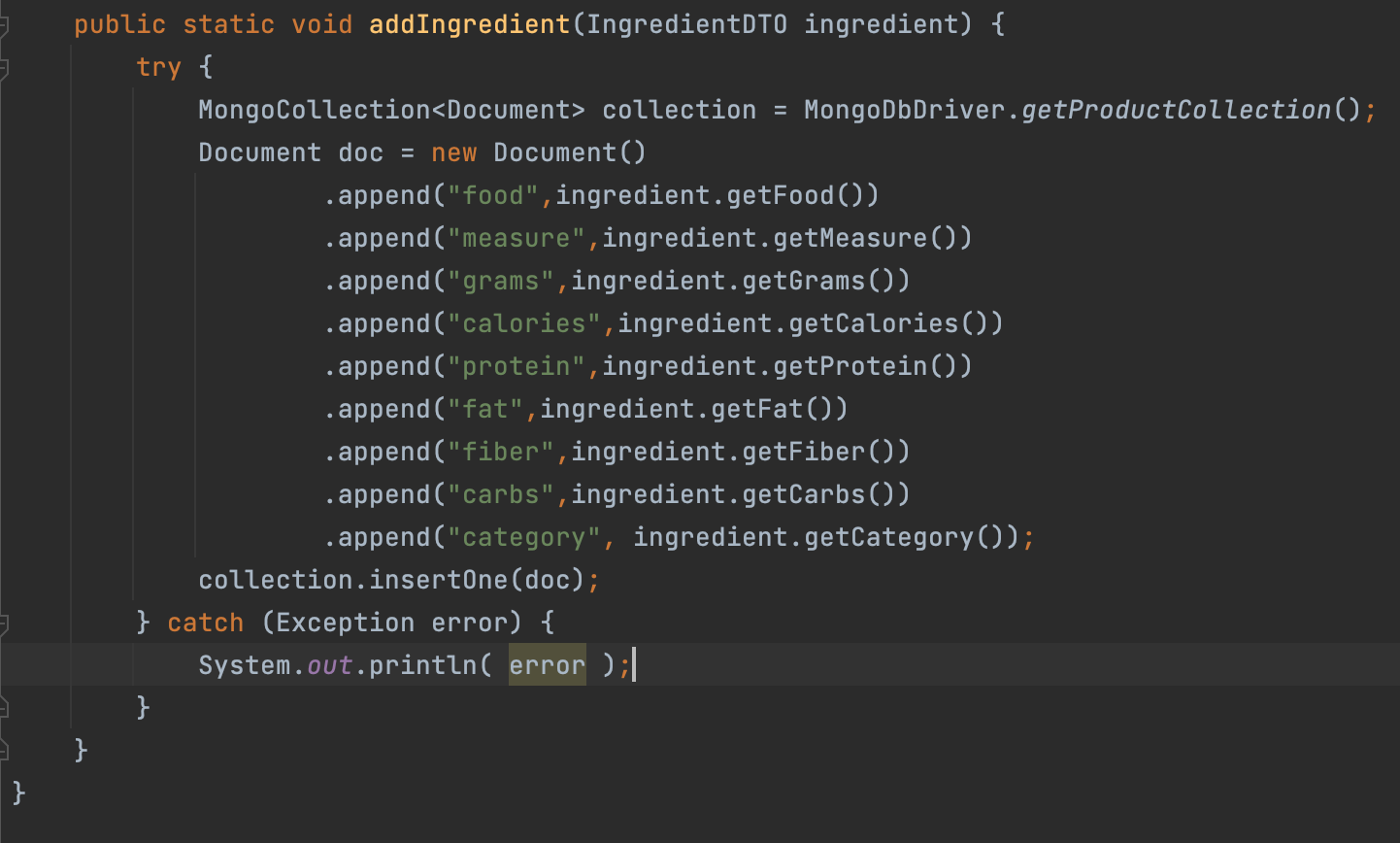
Ingredients: "Ingredients",

Directions: "Directions",

IngredientsList: "IngredientsList"

} )

* **create ingredient**

****

db.Ingredient.insertOne( {

food: "food",

measure: "measure",

grams: "grams",

calories: "calories",

protein: "protein",

fat: "fat",

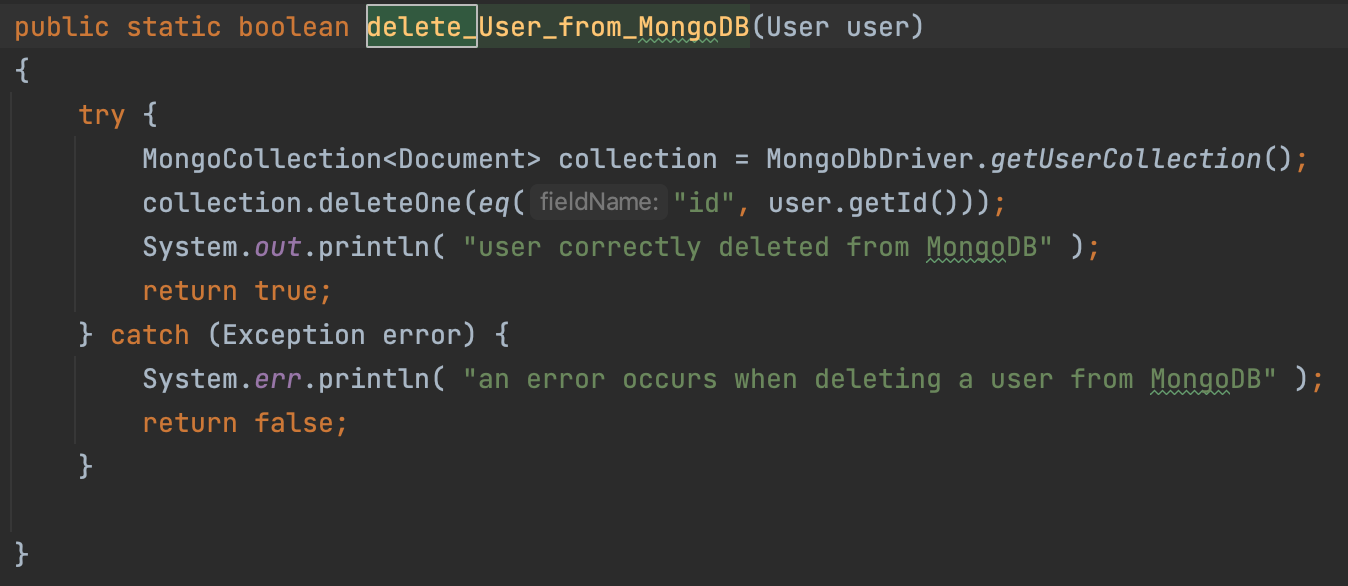
fiber: "fiber",

carbs: "carbs",

category: "category"

} )

* **delete user**

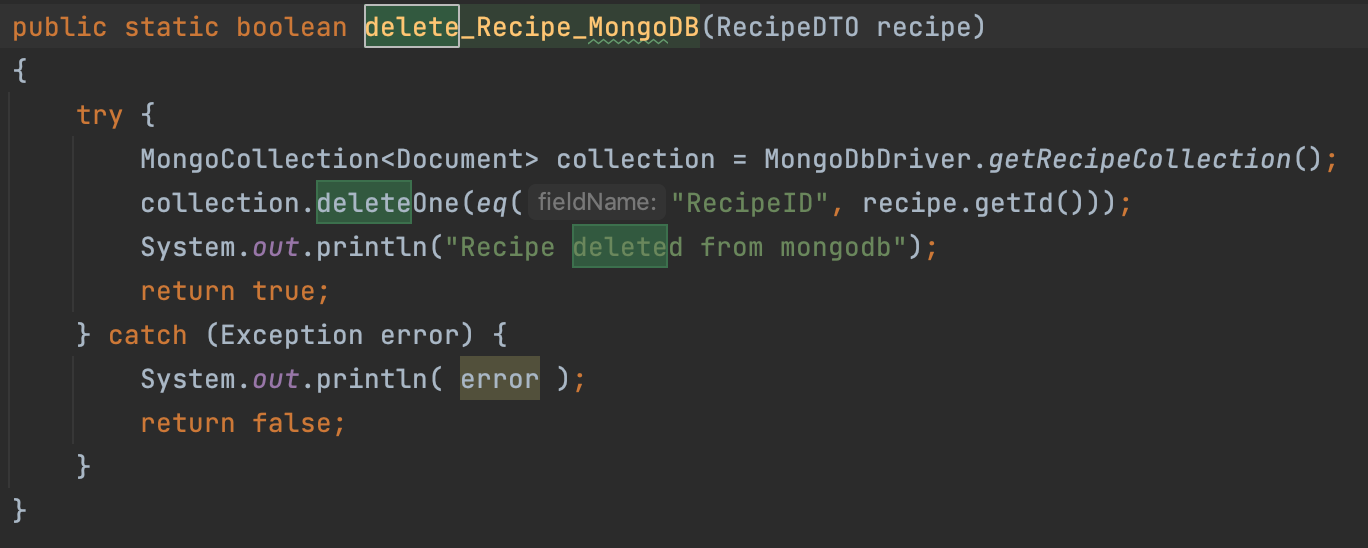
****

db.User.deleteOne( {

{ id : "id" }

)

* **delete recipe**

****

db.Recipe.deleteOne( {

RecipeID : "RecipeID"

})

* **delete ingredient**

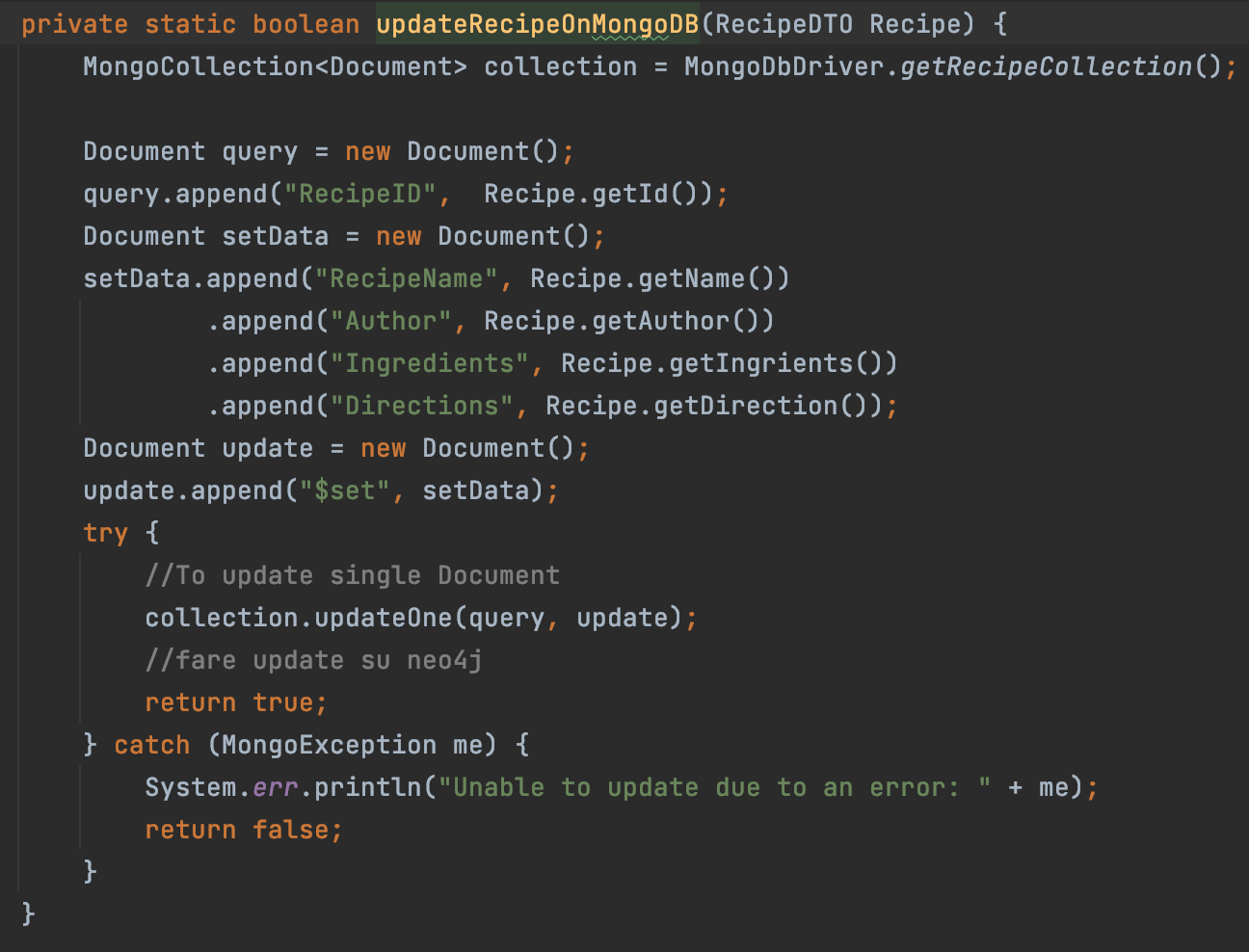
****

db.Ingredient.deleteOne( {

{ food : "food" }

)

* **update recipe**

****

db.Recipe.updateOne( {

{ RecipeID: "RecipeID" },

{ $set:

{

RecipeName: "RecipeName",

ReviewCount: "ReviewCount",

Author: "Author",

PrepareTime: "PrepareTime",

CookTime: "CookTime",

TotalTime: "TotalTime",

Ingredients: "Ingredients",

Directions: "Directions",

IngredientsList: "IngredientsList"

}

}

)

* **update user**

****

db.User.updateOne( {

{ id: "id" },

{ $set:

{

username: "username",

password: "password",

firstName: "firstname",

lastName: "lastName",

email: "email",

country: "country",

registrationdate: "registrationdate"

}

}

)

* **update ingredient**

db.Ingredient.updateOne( {

{ food: "food" },

{ $set:

{

measure: "measure",

grams: "grams",

calories: "calories",

protein: "protein",

fat: "fat",

fiber: "fiber",

carbs: "carbs",

category: "category"

}

}

)

**8. SHARDING AND REPLICAS**

“SmartFridge” is a read heavy application because most of the operation are searching for user/recipes/ingredient.  
As specified in the Non-Functional Requirements, our system has to offer fast response time, high availability and be tolerant to network partition. Due to the requirement above we chose to orient out application on the AP edge of the CAP triangle.

With this choice we ensure availability and partition protection despite of consistency, because it’s not necessary , in our case, to have strict consistency among replica cause this would lower the performance of system by lowering the response of a read request. A problem that can arise is that if the user asks for a replica that is stored into a server that is down, the data returned won’t be updated.

**Replicas**

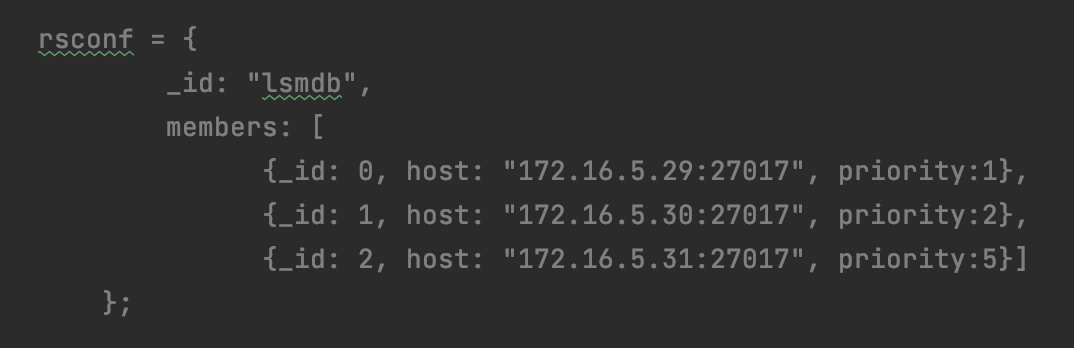
Regarding handling write operations, to ensure fast response time, we chose that when a write request occurs, only on one server that will be issued before returning back the control to the user. Related to this we adopted the eventual consistency paradigm to have at a certain point all the replicas will be updated.

The usage of replicas helps us also with the tolerance of network partitioning because if a server goes down that request can be redirected to other servers that contain a replica of the data requested.

The university of Pisa provided us three different VM where we can deploy three replicas. We have three different replicas for MongoDB, and one for Neo4J cause for it replicas are reserved to the one that a premium account

The command which was used to set cluster configuration in MongoDB on the virtual machines is show in the figure below. To the virtual machine in which runs MongoDB and Neo4J we have assigned the lowest priority for balancing the load.

We tried to balance the load by giving a lower priority to the VM running both Neo4J and MongoDB.



**SHARDING**

The use of the sharding technique can help us managing the load. For using sharding we must define the keys and the partitioning algorithm.

As partitioning algorithm we opted for consistent hashing which can help us in relocating data when a node is added / removed.

As shard key we selected the following:

* for User collection: key: ”id” which is a unique field
* for Ingredient collection: key: “\_id” automatically generated by mongoDB
* for Recipe collection: key: ”RecipeID” is a unique field

**CONCLUSIONS**

SmartFridge is application with a simple idea but with a huge goal: helping us organizing the ingredient in the fridge without let them expire and to suggest us recipe to be done with all or with some of our ingredient