UNIVERSITÀ DI PISA

Master Degree in Artificial Intelligence and Data Engineering

Large-Scale And Multi-Structured Database

Rechype

Tommaso Amarante, Edoardo Morucci, Niko Salamini

Github repository: https://github.com/TommyTheHuman/Rechype

Contents

Chapter 1

Introduction and Requirements

Rechype is a social networking application based on sharing recipe's ideas. Users that are registered to the service can create new recipes or take one from the set offered by the system that is continuously updated by other users. The system provides recipes coming from cocktailDB, Spoonacular API and PunkAPI, the first containing cocktail's recipes, the second containing mainly food recipes and the last one providing beer recipes. A set of ingredients, extracted from Spoonacular API, is accessible to the user to build their own fridge and create new recipes or drinks. The interface provides a way to create meals, those are composition of different recipes and drinks that can be labeled with different information like the name and the type (breakfast, brunch etc.). What kind of interactions can happen among users?

- Users can view recipes or drinks published by other users and can like them or add to favorites. The recipe/drink page provides a lot of details regarding the method used, the nutritional information (in the case of food recipes), the ingredients that have been used.
- The suggestion that are proposed to the user are based on his follows and likes. Those are called personal suggestion, there are also global suggestions involving global number of likes, follows and other things.
- A user is characterized by a level, that represents his experience in terms of number of published recipes and drinks. This fact leads to the introduction of competition among the users to become the better one in terms of experience.

1.1 Functional requirements

In the application we can observe three types of users: *Unlogged*, *logged* and *Admin*. The description is shown below along with possible actions they can perform.

Unlogged Users

He is the no-identity user browsing the welcome page.

- They can register to the service creating a new account that will be associated with a Logged User.
- They can login to the service inserting the credentials of an existing User account.

Logged Users

He is the user that has already logged in into the service.

- They can log out.
- They can manage their fridge adding or removing ingredients.
- They can manage their meals.
  - They can view them.
  - They can create a new one defining the recipes and drinks that composes it.
  - They can delete it.
- They can manage their recipes.
  - They can search for them using filters or not.
  - They can add to favorite other user’s recipes.
  - They can add a new one defining its ingredients along with other details.
- They can manage their drinks.
  - They can search for them using filters or not.
  - They can add to favorite other user’s drinks.
  - They can add a new one defining its ingredients along with other details.
- They can search for Users by filters or not and follow or unfollow them.
- They can view Users profiles along with their recipes and drinks.
- They can browse personal suggestions about recipes/drinks/users.
- They can browse global suggestions about best recipes/drinks/users.

Admin

A special user that can perform privileged actions on the application.

- They can ban users from the system.
- They have access to the statistics of the application.

1.2 Non-Functional requirements

The application is based on the concept of social networking, to provide the best user experience, we must achieve some important goals:

- User-Friendliness.
- Fast responses.
- High Availability of Contents.
- Fault-Tolerance, in terms of single-point of failure and data lost.

  To satisfy those requirements we have focused on the availability and partition tolerance features of the CAP triangle.

Chapter 2

Specifications

2.1 Actors and Use Case Diagram

There are 3 actors involved in this application, based on the previous requirements:

- **Unlogged User** can login with his credentials or can register to our application providing all the necessary informations.
- **Logged user** can access all the functionalities of the application. They can find other users and follow them, they can create different meals and they can also create drinks or recipes for the community. They can add ingredients to their own fridge and view suggestion about recipes, drinks, ingredients and users. They can add likes to recipes and drinks.
- **Admin** can view all analytics such as the user rank and they can ban a user if necessary.

The next two images describe the use case of the entire application and its class diagram.

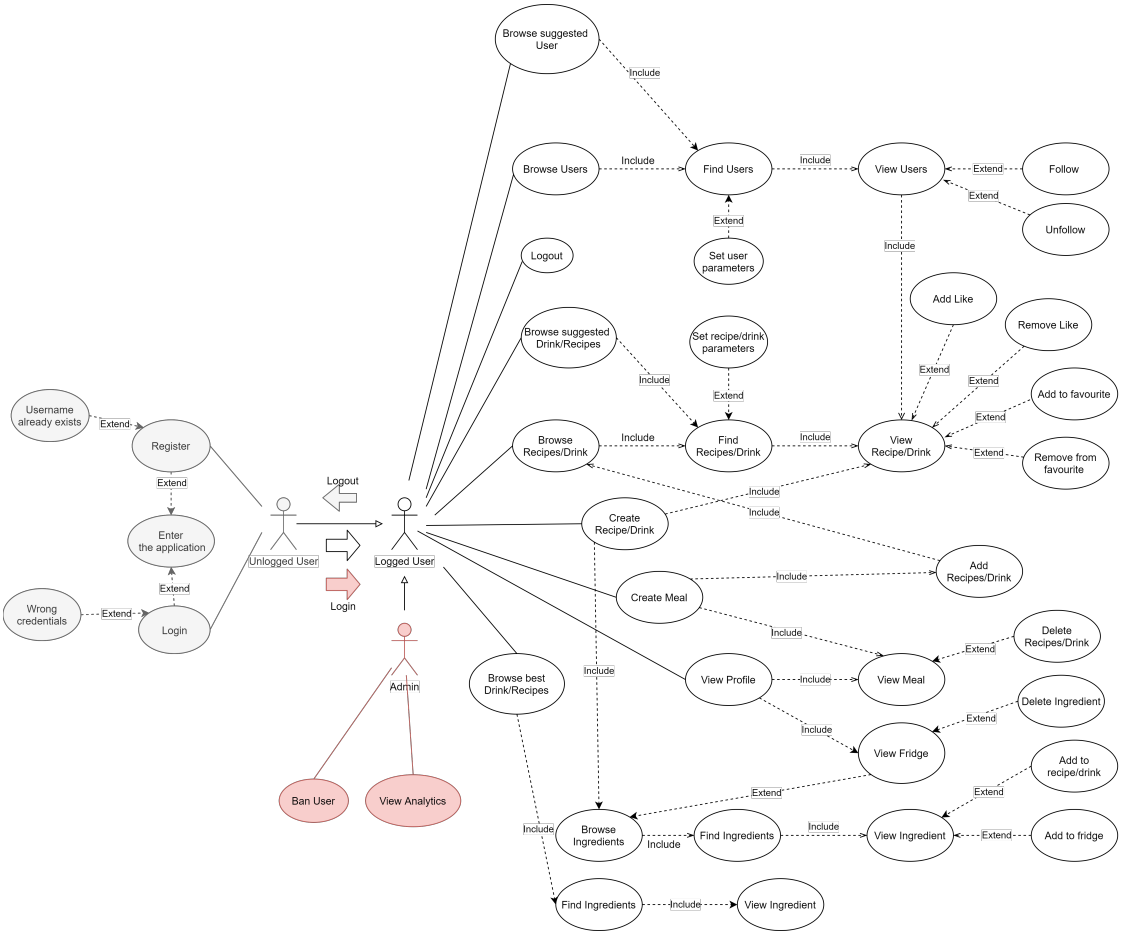

Figure 2.1: Use case diagram

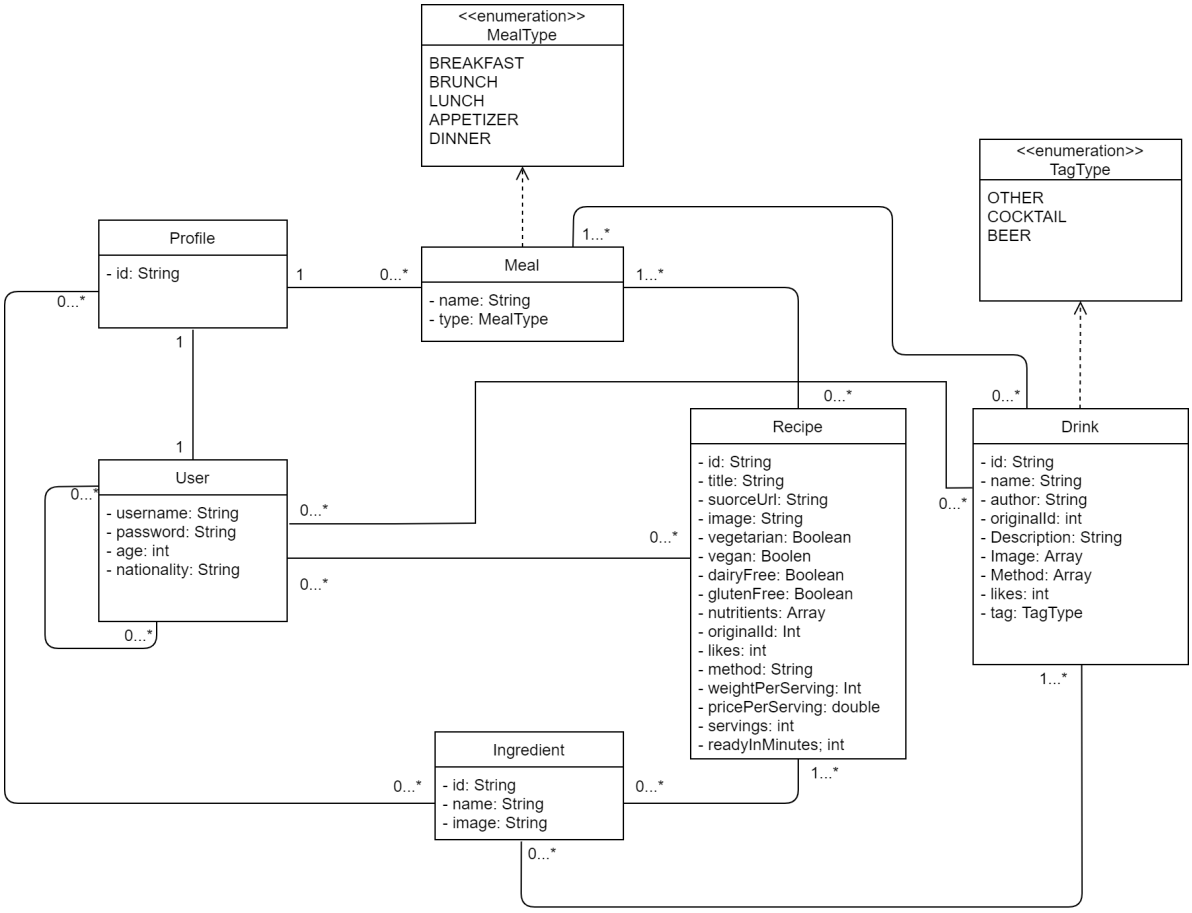

Figure 2.2: Class diagram

Chapter 3

Architectural Design

3.1 Software Architecture

Rechype is based on the *client-server* paradigm. We will explain the two part of this application.

3.1.1 Client side

The client code is organized in different packages:

GUI Package

The classes of this package exploits all the others in order to build the graphic interface and define the events involving the user's interaction with it.

Entity packages

(Profile, User, Drink, Recipe, Ingredient): Those packages are all organized in the same way:

- An *interface* to provide functionalities to the other packages.
- An *implementation class* of the interface where high-level functionalities, regarding the entity, are implemented.
- A *factory* to provide the functionalities of this entity package to external packages.
- A class representing the *entity*.
- A *DAO class* used to access the databases.

Persistence Package

It contains static classes used to **access the Databases**, in addition there is a configuration file to allow database tuning.

3.1.2 Server side

The server side is made up of *3 virtual machines* which host a distributed MongoDB and a single instance of Neo4j.

3.1.3 General Architecture

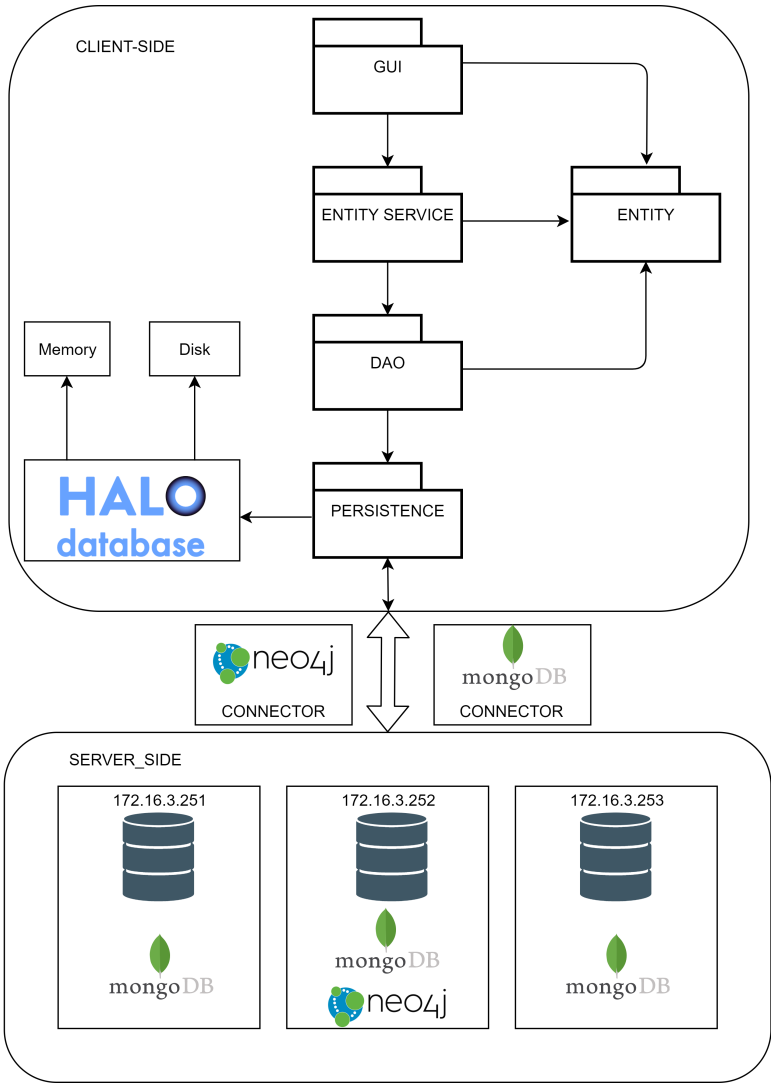

Figure 3.1: General Architecture Scheme

3.1.4 Dataset Composition

The central entity of the application are the recipes, we have referred to 2 different types: drink and food. In order to reinforce the variety of the dataset we have considered 3 different sources:

- **The PunkAPI** for beer's recipes (Drinks).
- **Spoonacular API** for both kind of recipes.
- **CocktailsDB** for cocktail's recipes (Drinks).

Beers, Cocktails and part of the Spoonacular API have been used to compose the drink's recipes. To compose food's recipes only the Spoonacular API has been used. All the recipe's ingredients extracted from the APIs are merged together and provided to the user to allow the creation of new recipes and drinks.

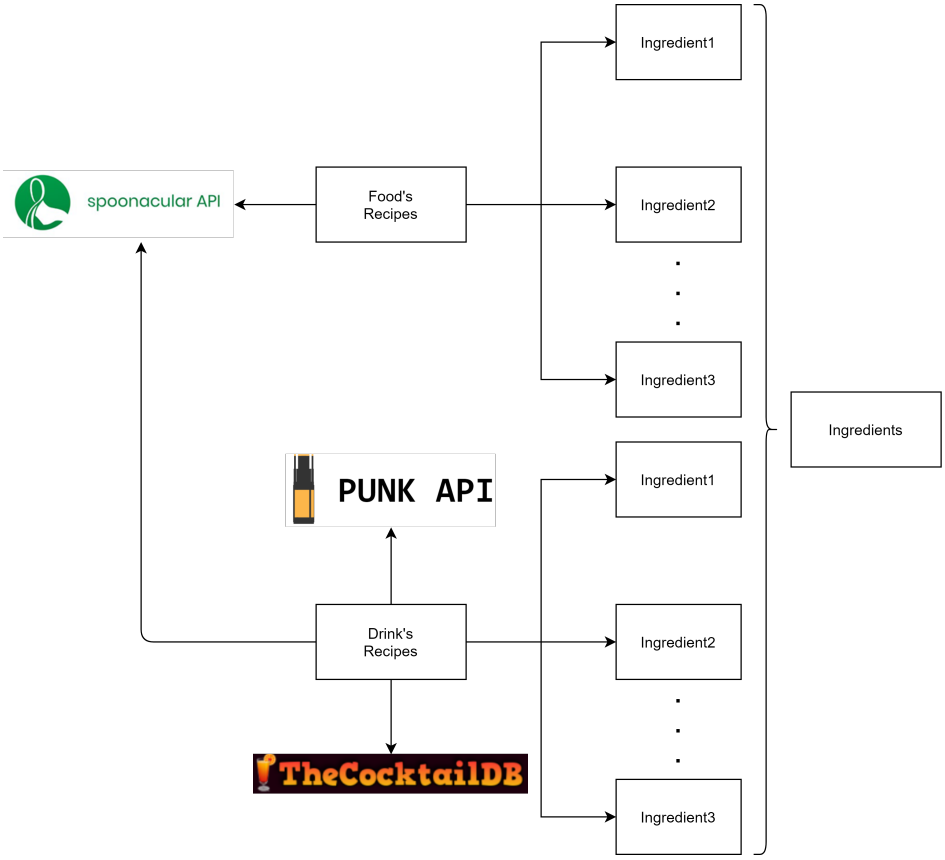

Figure 3.2: Dataset Composition Scheme

We decided to maintain the original Id from the API in order to avoid adding duplicates if we want to expand our collections in the future. An example could be an ETL Daemon that periodically queries all the APIs and expands the entire database with the new objects.

3.1.5 Database choices and motivation

We have exploited the features of 3 different types of databases:

- **MongoDB**: We have chosen a documentDB to exploit the support that provides in terms of storage and document embedding. Another important aspect is its suitability to indexing and complex queries elaboration, this last point is fundamental for the analytic part of the application.
- **Neo4j**: We have chosen a graphDB mainly for the support to suggestion operations, it is fundamental to retrieve information, traversing from an entity of the database to another, without affecting too much the performances. Graph databases are very suitable for this task.
- **HaloDB**: We have chosen a key-value db to cache some data on the client-side during the usage of the application, in particular for the search operation. The contribute that is given by this component is very important, it allows the application to avoid useless accesses on server-side.

Chapter 4

Design and Implementation of HaloDB

HaloDB is a fast and simple embedded key-value store written in Java. It comprises on 2 main components: an *index in memory* which stores all the keys, and append-only *log files* on the persistent layer which stores all the data. To reduce Java garbage collection pressure the index is allocated in native memory, outside the Java heap.

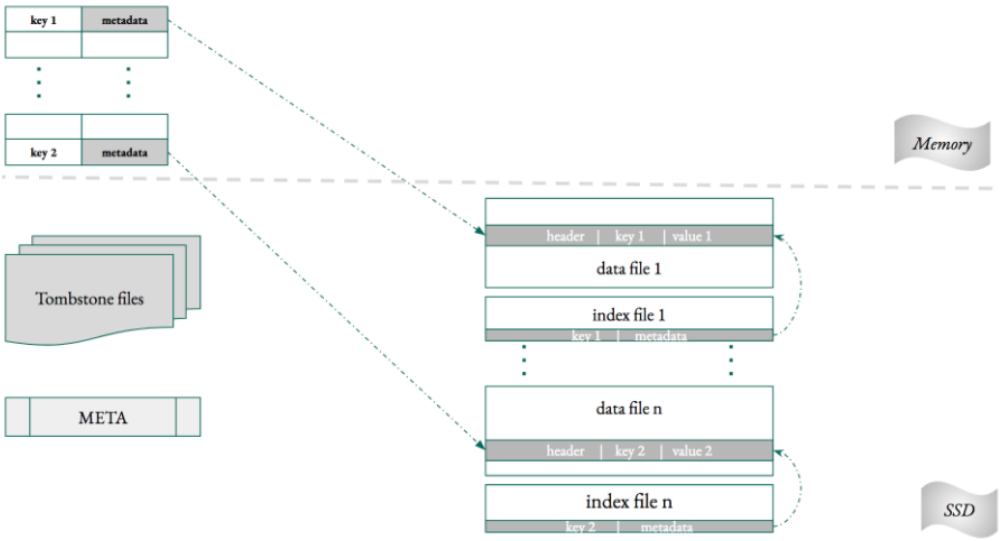

Figure 4.1: HaloDB Scheme

We have exploit HaloDB feature to cache data during the search operation in the application, in this way with only **one access** to the server-side (on MongoDB) we can get the collection's data and reuse them in a second moment during a particular event. For example in the recipes search we first get data from MongoDB with a request on server-side, after that the retrieved data are cached and if a click event occurs the recipe's page GUI is built using cached data on client-side.

4.1 Key Structure

The keys have a very simple structure, the data that we want to cache are the ones that could be searched in the application, so: users, recipes, drinks and ingredients. We have store all data in a single bucket, the general key structure is: **"EntityName:MongoId:JSONObject"**. In this way we guarantee the key uniqueness, all the entities are stored in a MongoDB collection with a different id so all the keys will be unique. For example in case of an ingredient: "ingredient:hass avocados:JSONObject".

4.2 Data Flushing

HaloDB has a background compaction thread which removes stale data from the DB, so the flushing is automatically handled during the run of the application. In addition, when the application is closed, we flush manually all the data on the append file on disk, removing all key-value pairs. We have also set 2 option for the automatic handling, one for deleting tombstone records (created during the deletion of a value) when the database starts and one for removing all the key-index pairs stored in memory when the database is closed.

Chapter 5

Design and Implementation of MongoDB

This database contains five collections to store information about recipes, drinks, ingredients, users and private profiles.

5.1 Collection

5.1.1 User Collection

Each user has his informations stored in a document in the Users Collection. We have introduced a redundancy of recipes and drinks in two arrays of documents that contains the ones created or favored by the user. In this way when the user page is displayed with a single access to the database we easily obtain the recipes and drinks (with a reduced set of fields) of the user. Once a recipe or drink are clicked, an access to the full document on the corresponding collection is performed obtaining all the remaining information.


```
[{
  "_id": "ed",
  "password": "ed",
  "country": "Algeria",
  "age": 25,
  "level": 4,
  "recipes": [
    {
      "_id": "60946899bdf7630236e87fc4",
      "name": "The Ultimate Dry Rub",
      "author": "Spoonacular",
      "pricePerServing": 462.81,
      "image":
```

```
        "https://spoonacular.com/recipeImages/1077221-312x231.jpg",
    "vegetarian": true,
    "vegan": true,
    "dairyFree": true,
    "glutenFree": true
   },
   {...}
  ],
  "drinks": [
   {
    "_id": "60945966bdf7630236e6f7a2",
    "name": "Drink & Dish: Pumpkin Pie Martini",
    "author": "Spoonacular",
    "tag": "other",
    "image":
       "https://spoonacular.com/recipeImages/596648-312x231.jpg"
   },
   {...}
  ]
}
```

5.1.2 Profile Collection

Each user has also a profile's document, user's meals and fridge are stored in 2 different arrays of documents. Each meal document contains 2 arrays of recipes and drinks with a set of reduced fields. Inside the fridge arrays of ingredients are stored. Since the user's infromation (profile infos and other user infos) are not always accessed together we have decided to split them in two separate collections: the users collection for the public information and profiles collection for the private information. The structure of the collections are shown in the pictures.

```
{
  "_id": "ed",
  "meals": [
   {
    "title": "mealprova",
    "type": "Dinner",
    "recipes": [
     {
      "_id": "60946899bdf7630236e87fc4",
      "name": "The Ultimate Dry Rub",
      "author": "Spoonacular",
```

```
        "pricePerServing": 462.81,
        "image":
           "https://spoonacular.com/recipeImages/1077221-312x231.jpg",
        "vegetarian": true,
        "vegan": true,
        "dairyFree": true,
        "glutenFree": true
      }
    ],
    "drinks": [
      {
        "_id": "60945966bdf7630236e6fa1c",
        "name": "Strawberry Daiquiri",
        "author": "Spoonacular",
        "tag": "other",
        "image":
           "https://spoonacular.com/recipeImages/1120177-312x231.jpg"
      }
    ]
   },
   {...}
   ],
  "fridge": [
   {
     "name": "bread crumbs",
     "quantity": "343",
     "image": "https://spoonacular.com/cdn/ingredients_100x100/bread
        crumbs"
   },
   {...}
  ]
}
```

5.1.3 Recipes Collection

The document that refers to a recipe contains all information about it. There are two arrays of documents: one for the ingredients and one for the nutrients. To improve read performances the information about the ingredients is composed by a subset of the totality stored in the ingredients collection.

```
{
  "_id": {
```

```
    "$oid": "60946899bdf7630236e87f81"
  },
  "name": "15 Minute Paleo Tacos & Taco Seasoning",
  "author": "Spoonacular",
  "vegetarian": false,
  "vegan": false,
  "glutenFree": true,
  "dairyFree": true,
  "pricePerServing": 507.21,
  "servings": 4,
  "image": "https://spoonacular.com/recipeImages/666825-312x231.jpg",
  "description": "The recipe 15Minute Paleo Tacos ...",
  "readyInMinutes": 15,
  "originalId": 666825,
  "method": "The recipe follow those steps:...",
  "likes": 0,
  "weightPerServing": 538,
  "ingredients": [
    {
      "ingredient": "avocados",
      "amount": "0.5 "
    },
    {...}
  ],
  "nutrients": [
    {
      "name": "Calories",
      "amount": 415.03,
      "unit": "kcal"
    },
    {...}
  ]
}
```

5.1.4 Drinks Collection

This collection contains document about drinks. Recipes and drinks collections are separated because they aren't accessed together during the usage of the application. The document contains an array of documents specifying the ingredients used along with their amounts. The tag field specifies the drink's category: beer, drink, other.

```
{
  "_id": {
    "$oid": "609455aebdf7630236e6ecbb"
  },
  "name": "A1",
  "author": "CocktailsDB",
  "originalId": "17222",
  "method": "Pour all ingredients into a cocktail shaker, mix and
     serve over ice into a chilled glass.",
  "description": "Alcoholic Cocktail",
  "image":
     "https://www.thecocktaildb.com/images/media/drink/2x8thr1504816928.jpg",
  "ingredients": [
    {
      "ingredient": "Gin",
      "amount": "1 3/4 shot "
    },
    {...}
  ],
  "likes": 0,
  "tag": "cocktail"
}
```


5.1.5 Ingredients Collection

The ingredients are stored independently because they can be accessed on the fridge page or during the creation of a drink/recipe. We decided to put together drinks and recipes ingredients to avoid duplicates, this is possible using the name of the ingredient as the _id field in the collection.


```
{
  "_id": "hass avocados",
  "image": "avocado.jpg",
  "nutrients": [
    {
      "name": "Fiber",
      "amount": 6.7,
      "unit": "g"
    },
    {
      "name": "Carbohydrates",
      "amount": 8.53,
```

```
    "unit": "g"
  },
  {...}
  ]
}
```

5.2 CRUD Operations

5.2.1 Create

The function used to create documents are:

1. `UserDao.checkRegistration`: create a user document.
2. `ProfileDao.insertProfile`: create a profile document.
3. `RecipeDao.addRecipe`: create a recipe document.
4. `DrinkDao.addDrink`: create a drink document.

5.2.2 Read

The function used to read documents are:

1. `UserDao.checkLogin`: check the username and the password specified.
2. `UserDao.getUserById`: get the user specifying id.
3. `UserDao.getUserByText`: get the users specifying a part of username.
4. `UserDao.checkSavedRecipe`: check if user has saved the recipe/drink with the specified id.
5. `UserDao.getUserRecipeAndDrinks`: get nested recipes and drinks.
6. `RecipeDao.getRecipesByText`: get recipes specifying a part of name.
7. `RecipeDao.getRecipeById`: get recipe specifying id.
8. `DrinkDao.getDrinksByText`: get drinks specifying a part of name.
9. `DrinkDao.getDrinkByKey`: get drink specifying id.
10. `ProfileDao.getProfileByUsername`: get profile specifying username.

5.2.3 Update

The function used to update documents are:

1. `UserDao.addNestedRecipe`: update user document adding new nested recipe/-drink to user.
2. `UserDao.removeNestedRecipe`: update user document removing nested/drink recipe from user.
3. `UserDao.banUser`: update recipes and/or drinks document decreasing the number of likes.
4. `RecipeDao.addLike`: update recipe document increasing the number of likes.
5. `RecipeDao.removeLike`: update recipe document decreasing the number of likes.
6. `DrinkDao.addLike`: update recipe document increasing the number of likes.
7. `DrinkDao.removeLike`: update recipe document decreasing the number of likes.
8. `ProfileDao.addMealToProfile`: update profile document adding a new nested meal.
9. `ProfileDao.deleteMealFromProfile`: update profile document removing the meal specified.
10. `ProfileDao.addIngredientToFridge`: update profile document adding a new nested ingredient.
11. `ProfileDao.deleteIngredientFromProfile`: update profile document removing the ingredient specified.

5.2.4 Delete

The function used to delete documents are:

1. `UserDao.banUser`: delete user document.
2. `ProfileDao.deleteProfile`: delete profile document.

5.3 Queries Analysis

The write operations specified above are less frequent then the read operations, the latter are more frequently and are performed on higher number of documents. In addition the requirements of the system are **availability** and **fast response time**, for these reasons we set:

- **Write concern: 1**
- **Read preference: nearest**

The first one is to allow the system to perform the write operation as quickly as possible, the writes affect only the primary node and replicas will be updated next in time (eventual consistency paradigm). The nearest read preferences specifies that the read operations are performed on the node with the fastest response, that is, the one with the least network latency.

5.4 Analytics

To extract interesting and useful information from data we chose some pipelines aggregation.

5.4.1 Best user by like

Description: Select users which have the higher number of likes.
Mandatory parameter: recipes or drinks.
Optional parameters: age range and nation of users.

Mongo java driver: `RecipeDao.getUserRankingByLikeNumber`
Mongo in the following we consider the case in which admin selects all possible parameters. For the drinks the query is the same but starts with `db.drinks`.

```
db.recipes.aggregate([
  { $match: {"author" :
    { $nin :["Spoonacular", "PunkAPI", "CocktailsDB"] }
  }},
  { $lookup:{
      from:"users",
      localField:"author",
      foreignField:"_id",
      as:"user"
    }
  },
```

```
  { $match:
    { "user.age": {$lt:maxAge}}
  },
  { $match:
    { "user.age": {$gt:minAge}}
  },
  { $match:
    { "user.country":"specified country" }
  },
  {
    $unwind: "$user"
  },
  { $group:
    {
       _id:"$user._id",
       count: {$sum:"$likes" }
    }
  },
  { $sort:{count:-1} },
  { $limit: 20}
])
```

5.4.2 Best user by like with category

Description: Select users which have the higher number of likes.

Mandatory parameter: recipes or drinks.

Optional parameters: category of recipe (vegan, vegetarian, ecc...) or drinks (beer, cocktail and others).

Mongo java driver: `RecipeDao.getRankingUserByLikeAndCategory`

Mongo in the following we consider the case in which admin selects vegetarian as category. For the drinks the query is the same but starts with `db.drinks` and has different option for categories.

```
db.recipes.aggregate([
  { $match:
    { "vegetarian": true }
  },
  { $group:
    {
       _id:"$author",
       count: {$sum:"$likes" }
```

```
    }
  },
  { $sort:{count:-1} },
  { $limit: 20}
])
```

5.4.3 Best user by health recipes

Description: Select users which have the higher number of healthy recipes (weightPerServing/calories < 0.8).

Optional parameters: level of the users.

Mongo java driver: `UserDao.getHealthRankByLevel`

Mongo In the following query the admin selects "silver" as level, so the level must be greater that 5 and less than 10.

```

db.users.aggregate([
  { $match:
    { "level": {$gt: 5}
  },
  { $match:
    { "level": {$lte: 10}
  },
  { $lookup:{
    from:"recipes",
    localField:"_id",
    foreignField:"author",
    as:"recipe"
  }},
  { $unwind:"$recipe" },
  { $unwind:"$recipe.nutrients" },
  { $match:
    { "recipe.nutrients.name":"Calories" }
  },
  { $match:
    { "recipe.nutrients.amount": {$gt:0} }
  },
  { $project:
    {
       _id:1,
       healthIndex: {
```

```
        $divide:["$recipe.weightPerServing",
           "$recipe.nutrients.amount"]}
    }
  },
  { $match:
    { healthIndex: {$lte:0.8 }}
  },
  { $group:
    {
       _id:"$_id",
       count:{$sum:1}
    }
  },
  { $sort:{ count:-1 }},
  { $limit: 20}
])
```

5.4.4 Popular ingredient

Description: Select ingredients which are most used in the recipes, counting them and ranking them.

Optional parameters: consider recipe with max values of "ready in minutes", selection of a recipe features (low calories, high protein and/or low fat).

Mongo java driver: `RecipeDao.getIngredientRanking`

Mongo in the following we consider the case in which admin selects 30 minutes as max ready in minute and low calories recipes. For the drinks the query is the same but start with `db.drinks`.

```
db.recipes.aggregate([
  { $match:
    { "readyInMinutes":{ $lte:30 }}
  },
  { $unwind: "$ingredients" },
  { $unwind: "$nutrients" },
  { $match:
    { "nutrients.name":"Calories" }
  },
  { $match:
    { "nutrients.amount":{ $lte:600 }}
  },
  { $group:
```

```
    {
       _id:"$ingredients.ingredient",
       count:{$sum:1}
    }
  },
  { $sort: {count:-1} },
  { $limit:60 }
])
```

5.4.5 Most saved recipes

Description: Select the most saved recipes or drinks.
Mandatory parameter: recipes or drinks.

Mongo java driver: `UserDao.mostSavedRecipes`
Mongo For the drinks the query is the same but start with `db.drinks`.

```
db.users.aggregate([
  { $unwind: "$recipes" },
  { $group:
    {
       _id:"$recipes._id",
       name:{ $first: "$recipes.name" },
       count:{$sum:1}
    }
  },
  { $sort:{count:-1} },
  { $project:
    {
       _id:0,
       name:1,
       count:1
    }
  }
])
```

5.5 Indexes Structure

We have evaluated different indexes to improve read performances on the database. The analysis of each case is explained below.

5.5.1 User's Search Index

The read operation on the user's collection involves filters on level and age field. Considering that the level field is updated every time a new recipe is created, an index on it would be too expensive. For this reason we have analyzed an index only on the age field. To speed-up the read operation on the user's collection we can consider 2 possibilities:

base case:

db.users.find({$and:[{_id:{$regex:/.*a.*/}}, {age:{$gte:25}}, {age:{$lt:50}}]}).hint({_id:1}) .explain("executionStats")

```
{
executionTimeMillis: 11
totalKeysExamined: 1999
totalDocsExamined: 1248
}
```

age and _id:

db.users.find({$and:[{_id:{$regex:/.*a.*/}}, {age:{$gte:25}}, {age:{$lt:50}}]}) .hint({age:1,_id:1}).explain("executionStats")

```
{
executionTimeMillis: 2
totalKeysExamined: 577
totalDocsExamined: 369
}
```

age:

db.users.find({$and:[{_id:{$regex:/.*a.*/}}, {age:{$gte:25}}, {age:{$lt:50}}]}) .hint({age:1}).explain("executionStats")

```
{
executionTimeMillis: 9
totalKeysExamined: 1999
totalDocsExamined: 1248
}
```

From the results we can affirm that the best index, in terms of performance, is age:1, _id:1.

5.5.2 Recipe's Search Index

In the case of the recipe's search we have to consider 2 types of filters: the booleans (vegetarian, vegan, dairy free, gluten free) and price per serving. The indexing of booleans is not a good idea, the reason is that the values are too similar and performance improvement would be really poor. Regarding the price per serving we can evaluate a compound index with the title field. Finally we have the likes sort that is not worth it cause the update of the index is too expensive for the system.

base case:

```
db.recipes.find({$and:[{name:{$regex:/.*a.*/}},
{pricePerServing:{$gte:500}},{pricePerServing:{$lt:1000}}]})
    .hint({_id:1}).explain("executionStats")
```

```
{
executionTimeMillis: 128
totalKeysExamined: 26353
totalDocsExamined: 26353
}
```

pricePerServing and name:

```
db.recipes.find({$and:[{name:{$regex:/.*a.*/}},
{pricePerServing:{$gte:500}},{pricePerServing:{$lt:1000}}]})
    .hint({pricePerServing:1, name:1}).explain("executionStats")
```

```
{
executionTimeMillis: 26
totalKeysExamined: 2640
totalDocsExamined: 1978
}
```

pricePerServing:

```
db.recipes.find({$and:[{name:{$regex:/.*a.*/}},
{pricePerServing:{$gte:500}},{pricePerServing:{$lt:1000}}]})
    .hint({pricePerServing:1}).explain("executionStats")
```

```
{
executionTimeMillis: 35
totalKeysExamined: 2640
totalDocsExamined: 2640
```

```
}
```

From the results we can affirm that the best index, in terms of performance, is pricePerServing:1, name:1.

5.5.3 Drink's Search Index

In the drink's search the filtering is only on tags, but possible tags are only three so performance improvement would be poor. By the fact that likes field is critical for updates, the only index to evaluate is on title.

base case:

db.drinks.find({name:{$regex:/.*a.*/}}).hint({_id:1}).explain("executionStats")

```
{
executionTimeMillis: 27
totalKeysExamined: 5250
totalDocsExamined: 5250
}
```

title:

db.drinks.find({name:{$regex:/.*a.*/}}).hint({name:1}).explain("executionStats")

```
{
executionTimeMillis: 28
totalKeysExamined: 5250
totalDocsExamined: 3521
}
```

The conclusion is that title index does not improve the performance, so no indexes are used for the drinks collection. This is probably caused by the eccessive title's length of drinks, too much for an index.

5.6 Dimension of nested document

Since we have collections with arrays of embedded documents, like recipes and drinks in the User collection, if we need to update several times by adding nested documents, we have the problem that if the expected space for that document is exceeded the dbms must reallocate it. This can lead to relevant database inefficiencies because relocation can be more than one depending on the number of updates.

Solution: Allocate the document with an already defined size, putting at the creation of a document a predefined number of nested objects but with fields to set (therefore initially empty). We can set a limit on the number of nested object that a user can add, if we can't set this limit (for the application logic) we have to evaluate a trade-off between the limit of nested objects and the performance of the total system. We can also add more empty nested objects every time a document is relocated in order to limit the number of relocation in the application.

Chapter 6

Neo4j Design and Implementation

6.1 Nodes

Each node represent a main entity of our application, to be more specific there are 4 main entities:

- **User**: contains *username*, *level*, *country*, *age*.
- **Recipe**: contains *author*, *MongoDB id* as string, *imageUrl*, *name*, *price per serving* and the attributed for *vegetarian*, *vegan*, *dairyfree* and *glutenfree*.
- **Drink**: contains *author*, *MongoDB id* as string, *name*, *imageUrl* and the *tag*.
- **Ingredient**: contains *id* that represnet the ingredient name and some ingredients also have the *imageUrl*.

All the information on the nodes are necessary for the suggestion queries.

6.2 Relations

- **USER -FOLLOWS-> USER**: this relation is added when a user follows another users, this relation also has an attribute called *since* that represents the date in which the relation was created.
- **USER -LIKES- RECIPE**: this relation is added when a user clicks the like button on a recipe page. Also this relation has the *since* attribute. This relation is equivalent on a *Drink* node. The relation will be USER -LIKE-> DRINK.
- **RECIPE -CONTAINS-> INGREDIENT**: this relation is useful in order to retrieve the best ingredient. Also in this case there is an equivalent relation for a Drink node, DRINK -CONTAINS-> INGREDIENT. This relation is only present in user's recipes.

- **USER -OWNS-> RECIPE**: this relation is added when a user creates a recipe and also in this case there is an equivalent relation for the Drink node, USER -OWNS-> DRINK.

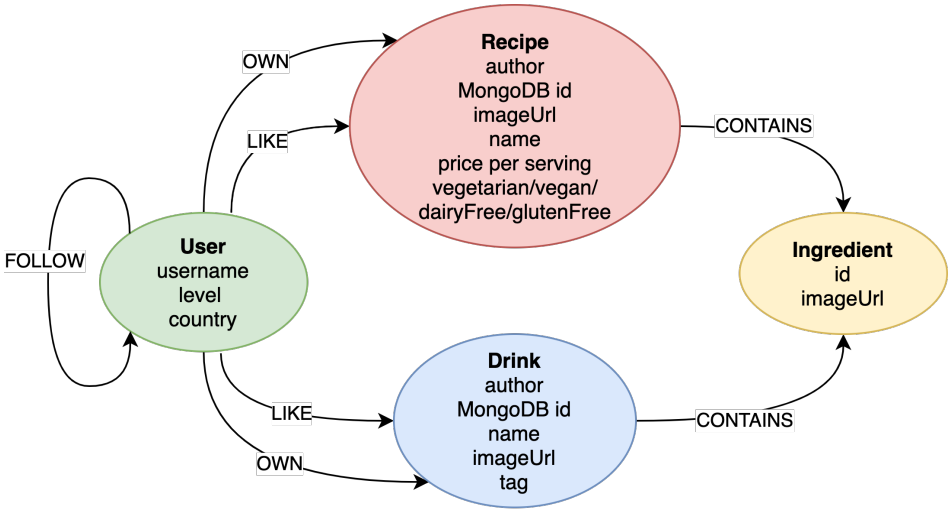

Figure 6.1: Nodes and relations.

6.3 Queries Implementation

6.3.1 CRUD Operations

Create

Operation	Cypher Implementation
Create a user node	CREATE (ee:User { username: $username, country: $country, level: $level , age:$age})
Create a recipe node with OWN relation and CONTAIN relation	MATCH (u:User) WHERE u.username=$owner MATCH(i1:Ingredient) WHERE i1.id="hass avocados" CREATE (ee:Recipe { id:$id, name: $name, author: $author, pricePerServing: $pricePerServing, imageUrl: $imageUrl, vegetarian: $vegetarian, vegan: $vegan, dairyFree: $dairyFree, glutenFree: $glutenFree}) CREATE (u)-[rel:OWNS {since:date($date)}]->(ee) CREATE (ee)-[:CONTAINS]->(i1)
Create a drink node with OWN relation and CONTAIN relation	MATCH (u:User) WHERE u.username=$owner MATCH(i1:Ingredient) WHERE i1.id="Lemon" CREATE (d:Drink { id:$id, name: $name, author: $author, imageUrl: $imageUrl, tag: $tag}) CREATE (u)-[rel:OWNS {since:date($date)}]->(d) CREATE (d)-[:CONTAINS]->(i1)
Creatre FOLLOW relation	MATCH (uu:User) WHERE uu.username = $myName MATCH (uu2:User) WHERE uu2.username = $userName CREATE (uu)-[rel:FOLLOWS {since:date($date)}]->(uu2)
Create LIKE relation	MATCH (uu:User) WHERE uu.username = $username MATCH (rr:Recipe) WHERE rr.id = $_id CREATE (uu)-[rel:LIKES {since:date($date)}]->(rr)

Read

Operation	Cypher Implementation
Check if a FOLLOW relation is present	MATCH (u:User {username: $myName })-[rel:FOLLOWS]->(u2:User {username: $userName}) return rel
Chech if a LIKE relation is present on a Recipe Node	MATCH (u:User {username: $username })-[rel:LIKES]->(r:Recipe {id:$_id}) return rel
Check if a LIKE relation is present on a Drink Node	MATCH (u:User {username: $username })-[rel:LIKES]->(d:Drink {id:$_id}) return rel

Update

Operation	Cypher Implementation
Update the user level	MATCH (u:User) WHERE u.username=$username SET u.level=u.level+1

Delete

Operation	Cypher Implementation
Delete a User node and detach all its relations	MATCH (u:User) WHERE u.username=$username OPTIONAL MATCH (u)-[rel1:LIKES]->(r:Recipe) OPTIONAL MATCH (u)-[rel2:LIKES]->(d:Drink) WITH collect(r.id) AS RecipeIds, collect(d.id) AS DrinkIds, u DETACH DELETE u return RecipeIds, DrinkIds
Delete FOLLOW relation (DB consistency)	MATCH (uu:User {username:$myName})-[rel:FOLLOWS]->(uu2:User {username:$userName}) delete rel
Delete User node (Cross-DB consistency)	MATCH (u:User { username: $username }) delete u
Delete LIKE relation on recipes (Cross-DB consistency)	MATCH (uu:User {username:$username})-[rel:LIKES]->(r:Recipe {id:$_id}) delete rel
Delete LIKE relation on drinks (Cross-DB consistency)	MATCH (uu:User {username:$username})-[rel:LIKES]->(d:Drink {id:$_id}) delete rel

6.3.2 On-Graph Queries

User

Graph-centric query	Domain query
Considering all the User vertices that have at least 1 outgoing "OWNS" edge to a Drink Vertex and to a Recipe Vertex. Select all the vertices using the "OWNS" edge with the "date" attribute in the range of the last 7 days and count the "LIKES" edges on the matching vertices (recipes and drinks).	Who are the users, considering the ones that have created at least one drink and one recipe globally, that have the highest number of likes on recipes and drinks created in the last week?
Considering the Vertex A, select all the vertices that takes 1 hop from Vertex A with "FOLLOWS" edge. Select all the vertices that takes 1 hop with "FOLLOWS" edge from the vertices selected before, ignore the vertices that have incoming "FOLLOWS" edge from Vertex A. Rank the vertices by the number of outgoing "FOLLOWS" edges.	Who are the users with the highest number of followers that are followed by the users that a certain "User A" follows and that they are not already followed by User A?

First query

```
MATCH (u:User)-[:OWNS]->(r:Recipe)
OPTIONAL MATCH (:User)-[owns1:OWNS]->(r)<-[likes1:LIKES]-(:User)
WHERE date($date)-duration({days:7})<owns1.since<=date($date)+duration({days:7})
WITH u, count(likes1) as RecipesLikes
MATCH (u)-[:OWNS]->(d:Drink)
OPTIONAL MATCH (:User)-[owns2:OWNS]->(d)<-[likes2:LIKES]-(:User)
WHERE date($date)-duration({days:7})<owns2.since<=date($date)+duration({days:7})
WITH u, count(likes2)+RecipesLikes as totalLikes
RETURN u AS User, totalLikes
ORDER BY totalLikes DESC, User.username ASC
LIMIT 10
```

Second query

```
MATCH (u1:User)-[:FOLLOWS]->(:User)-[:FOLLOWS]->(newUser:User)
WHERE u1.username=$username AND newUser.username<>$username AND (NOT
((u1)-[:FOLLOWS]->(newUser)))
MATCH (:User)-[rel:FOLLOWS]->(newUser)
RETURN newUser, count(rel) AS totalFollowers
ORDER BY totalFollowers DESC, newUser.username ASC
LIMIT 10
```

Drinks

Graph-centric query	Domain query
Select all the drink vertex with incoming "LIKES" edge that has the attribute "date" in the range of the last 7 days.	Which are the drinks that have received the highest number of like in the last week?
Considering the Vertex A, select alle the vertices that takes 1 hop from Vertex A with "FOLLOWS" edges. Select the Drink vertices with the "OWNS" edges coming from the vertices selected before excluding Vertex A. Count the number of incoming "LIKES" edges with the "date" attribute in range of last 7 days.	Which are the drinks that are created by the users that "User A" follows that have the highest number of likes in the last week?

First query

```
MATCH (:User)-[likes:LIKES]->(d:Drink)
WHERE date($date)-duration({days:7})<likes.since<=date($date)+duration({days:7})
RETURN d AS DrinkNode, count(likes) AS totalLikes
ORDER BY totalLikes DESC, DrinkNode.name ASC
LIMIT 10
```

Second query

```
MATCH (u:User {username: $username })-[rel:FOLLOWS]->(u2:User)
MATCH (u2)-[relLikes:LIKES]->(d:Drink)
WHERE date($date)-duration({days:7})<relLikes.since<=date($date)+duration({days:7})
AND d.author<>$username
WITH d AS DrinkNode, count(relLikes) AS likesNumber
RETURN DrinkNode, sum(likesNumber) as totalLikes
ORDER BY totalLikes DESC, DrinkNode.name ASC
LIMIT 10
```

Recipes

Graph-centric query	Domain query
Select all the recipes vertex with incoming "LIKES" edge that has the attribute "date" in the range of the last 7 days.	Which are the recipes that have received the highest number of like in the last week?
Considering the Vertex A, select alle the verteces that takes 1 hop from Vertex A with "FOLLOWS" edge. Select the Recipes verteces with the "OWNS" edges with the "date" attribute contained in the last 7 days.	Which are the recipes that are created by the users that "User A" follows that have the highest number of likes in the last week?

First query

```
MATCH (:User)-[likes:LIKES]->(r:Recipe)
WHERE date($date)-duration({days:7})<likes.since<=date($date)+duration({days:7})
RETURN r AS RecipeNode, count(likes) AS totalLikes
ORDER BY totalLikes DESC, RecipeNode.name ASC LIMIT 10
```

Second query

```
MATCH (u:User username: $username )-[rel:FOLLOWS]->(u2:User)
MATCH (u2)-[relLikes:LIKES]->(r:Recipe)
WHERE date($date)-duration(days:7)<relLikes.since<=date($date)+duration(days:7)
AND r.author<>$username
WITH r AS RecipeNode, count(relLikes) AS likesNumber
RETURN RecipeNode, sum(likesNumber) as totalLikes
ORDER BY totalLikes DESC, RecipeNode.name ASC
LIMIT 10
```

Ingredients

Graph-centric query	Domain query
Select all the Ingredient vertices. Match all the incoming "CONTAINS" edges from recipes and drinks vertices. Select the Recipe and Drink vertices that has the attribute "date" in the incoming "OWNS" edge in range of the last 7 days. Count the number of "CONTAINS" edges grouped by the ingredient "id".	Which are the ingredients that are most used during the last week?

Query

```
MATCH (i:Ingredient)
WHERE date($date)-duration(days:7)<owns.since<=date($date)+duration(days:7)
WITH i, count(con) as RecipesAdding
OPTIONAL MATCH (:User)-[owns2:OWNS]->(d:Drink)-[con2:CONTAINS]->(i)
WHERE date($date)-duration(days:7)<owns2.since<=date($date)+duration(days:7)
WITH i, count(con2)+RecipesAdding as totalAdding
RETURN i AS Ingredient, totalAdding
ORDER BY totalAdding DESC, i.id ASC
LIMIT 10
```

6.4 Neo4j Indexes Structure

In order to enhance the performances of read requests on the Neo4j database we have considered an index on the user's entity involving the username field.

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

This index could help queries where the starting point of the graph traversal is username dependent, like the suggestions ones.

Chapter 7

Additional implementations details

7.1 Sharding: A possible implementation

To satisfy the non-functional requirements, in particular to improve the availability feature, we have considered, only in theory, a possible sharding method.

- For the users collection we can consider the **country** field as a shard key and a partitioning method based on **list**. In this way user's information are segmented by country on the cluster and each server, locally deployed, would serve the incoming requests for the country's users. For countries that produce a critical amount of requests we should consider an higher number servers and add some more replicas.
- For the other collections (recipes, ingredients and drinks) the only way to get an equal distribution is to use a shard key based on the _id field and a partitioning method using an hash table.

What about adding or removing nodes from the cluster? We should consider a way to relocate shard keys on the cluster following the principles of consistent hashing.

Sharding, along with data replication, allows the balance of the work on the cluster without overloading a single node. In this way data are well-distributed and the response latency to the client-side is reduced, giving a better experience to the user.

7.2 Cross-Database Consistency

In the application is important to take care of cross-database consistency.

- When we need to **add an entity** (Fig 7.1) or to **add or remove a like on a recipe** (Fig 7.2) we start by adding the entity or the relation on MongoDB or

Neo4j. If an error occurs an error is written in a log file and the operation fails. We move to the second operation, if an error occurs there, the error is written in the log file and we try to remove the entity or the relation from the other DataBase, if an error occurs during this deletion a **Parse error** is written into the log file.

- When we need to **delete a user** (Fig 7.3) or to **update the user level** (Fig 7.4) we start the first operation from MongoDB, if an error occurs we log it as a **Parse error**, if there is no error we can delete or update in Neo4j, if an error occurs here it has to be logged into the file as a **Parse error**. Note that in the case of user deletion the function is used to guarantee the consistency between the profiles and the users collection.

- When we need to **ban a user** (Fig 7.5) we try to delete from MongoDB, if an error occurs we log it and the operation fails directly. Then we can try to delete the entity from Neo4j, if the deletion fails a **Parse error** is logged. If there is no error we must update the recipes' and drinks' likes in MongoDB, if an error occurs during the update a **Parse error** is logged.

If the consistency can't be guaranteed we can use the **Parse error**. An external software periodically reads the log file and solve automatically the inconsistencies involving the "[PARSE]" key that contains the actions that have to be redone and the objects affected by the failed actions. The external software can analyze every line and solve every log with the "[PARSE]" key.

7.2.1 Add an entity (User, Recipe or Drink)

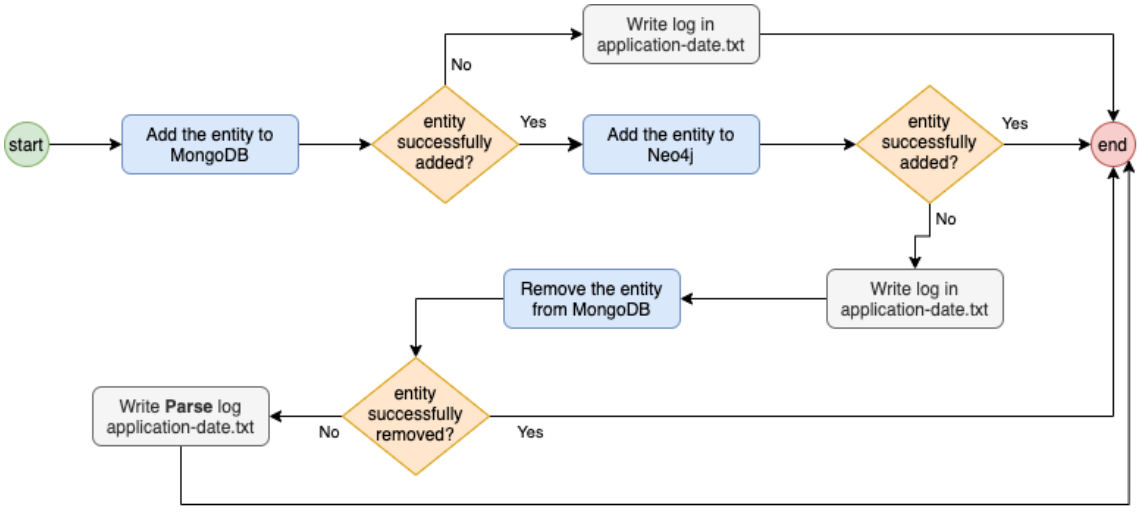

Figure 7.1

7.2.2 Add/remove like on Recipe or Drink

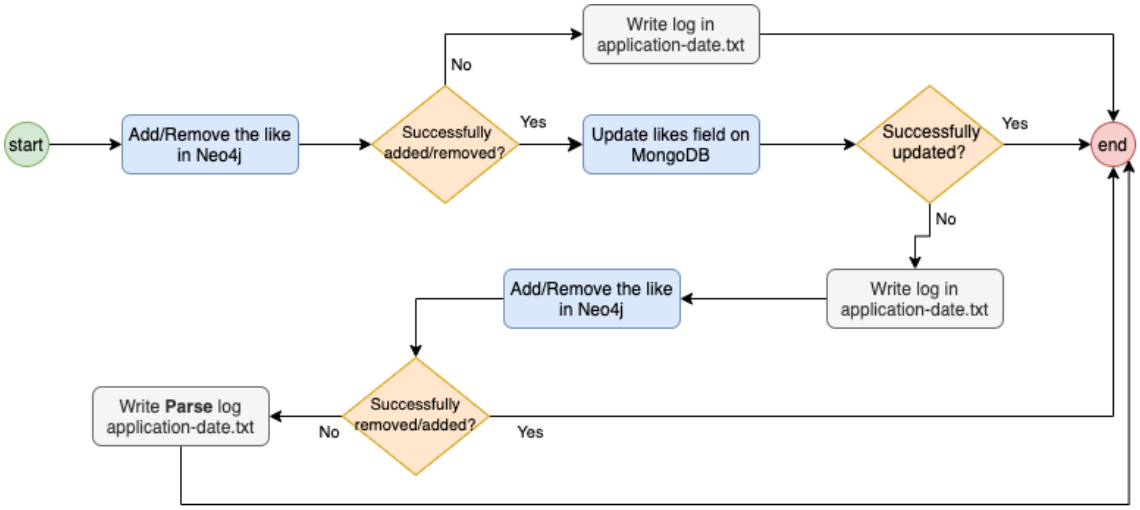

Figure 7.2

7.2.3 Delete User

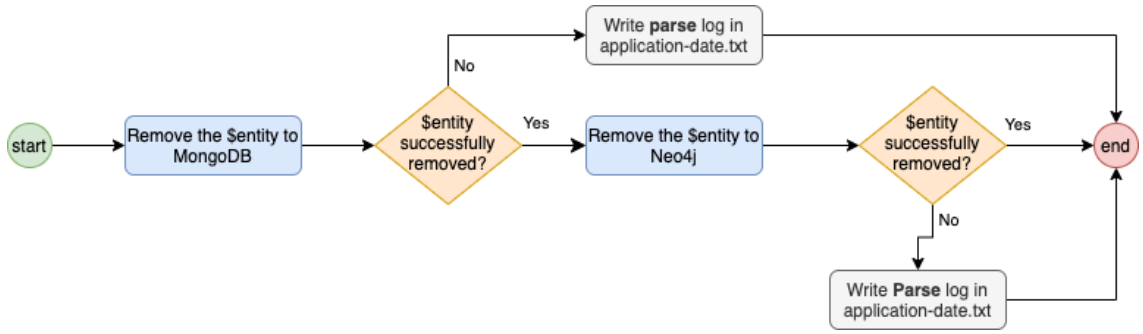

Figure 7.3

7.2.4 Update level

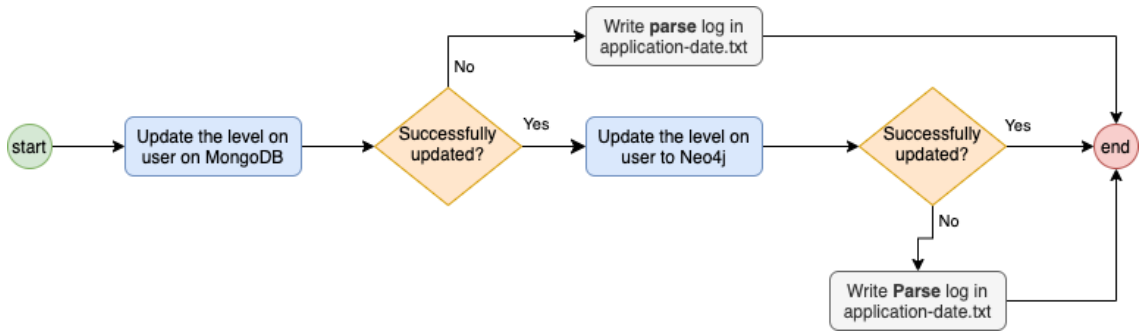

Figure 7.4

7.2.5 Ban User

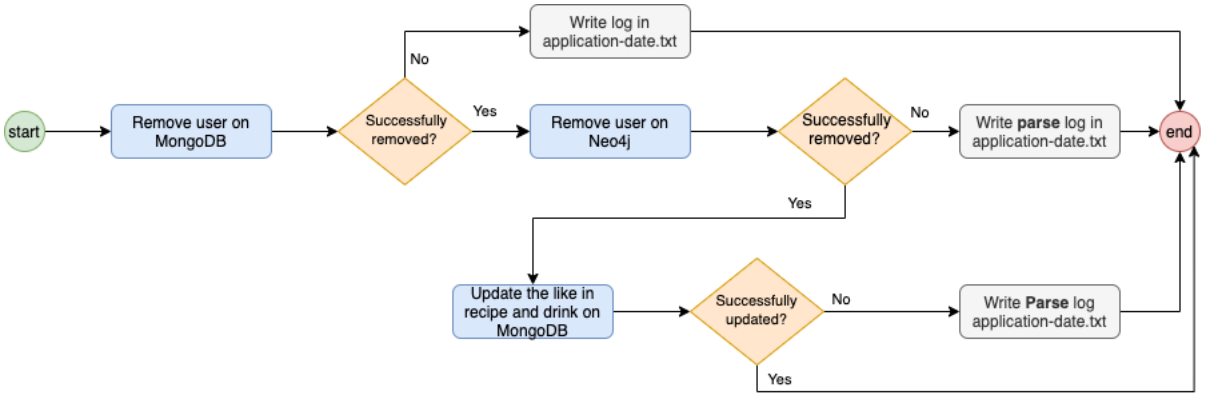

Figure 7.5

Chapter 8

User Manual

The first page shown to the user is called *Landing Page* and it contains a form to login and a form to register a new account as shown in figure 8.1.

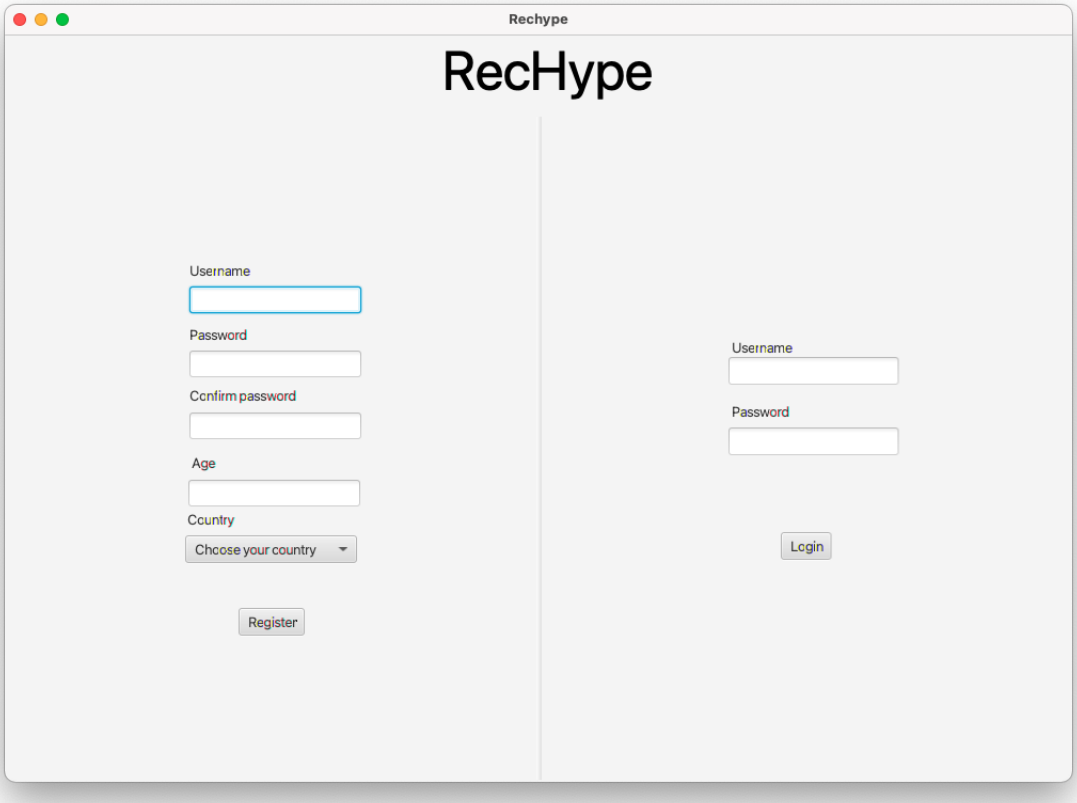

Figure 8.1: landing Page

Once the user is logged the *HomePage* is shown and it contains different useful suggestion, such as:

- **Suggested Recipes**: it shows recipes with the highest number of likes in the week in the follower set of the user.
- **Suggested Drinks**: it shows the drinks with the highest number of likes in the week in the follower set of the user.
- **Suggested Users**: it retrieves the users followed by the user's followed and returning the users with the highest number of followers.
- **Best Users**: it shows the users that have created the highest number of recipes or drink in the week and that have received the highest number of likes on those recipes in the week.
- **Best Drinks**: it shows the drinks with the highest number of likes obtained in the week.
- **Best Ingredients**: it shows the most used ingredients in the week.
- **Best Recipes**: it shows the recipes that have obtained more likes in the week.

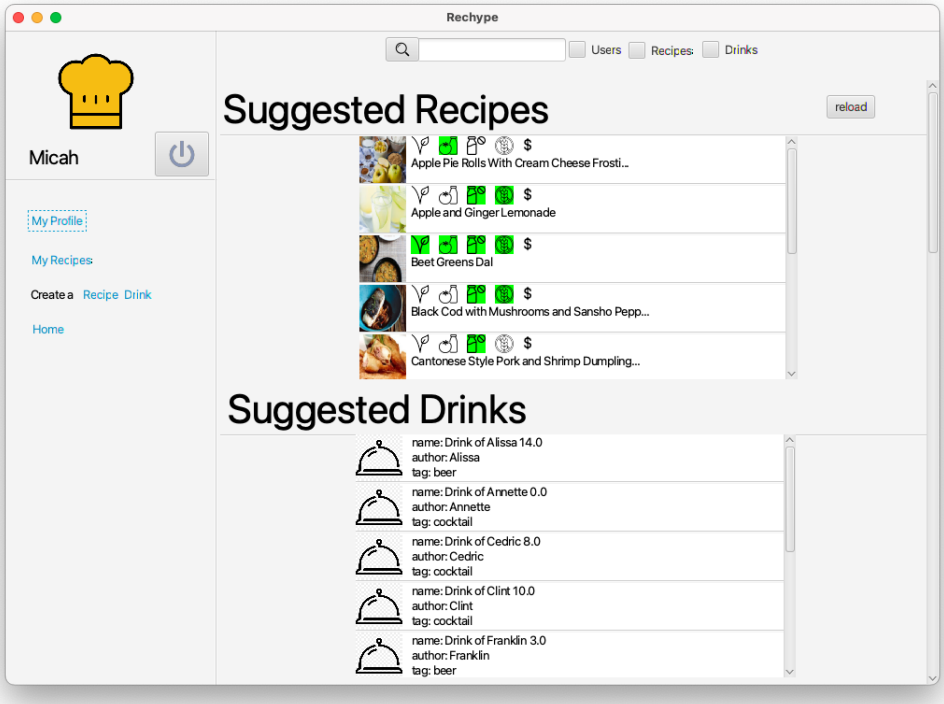

Figure 8.2: Home Page

As shown in figure 8.2 there is also a sideBar that is equal in every page and permits the user to move through the application. Also the searchbar remains the same in every page and, as shown in figure 8.3, permits to search specifying some filters.

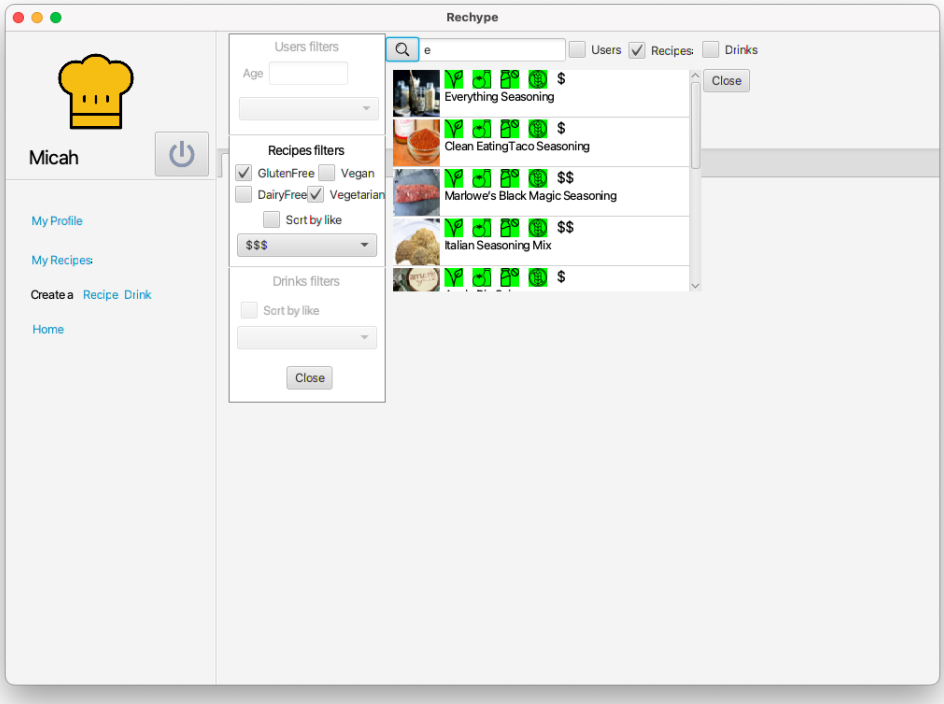

Figure 8.3: Search bar

You can click on a result from the search to open the *recipe* or *drink* page as shown in figure 8.4 and in figure 8.5. In that pages you can view more information about the recipe such as a short description and the steps to create it on your own. From this page you can also add the recipe to your favorites or like it.

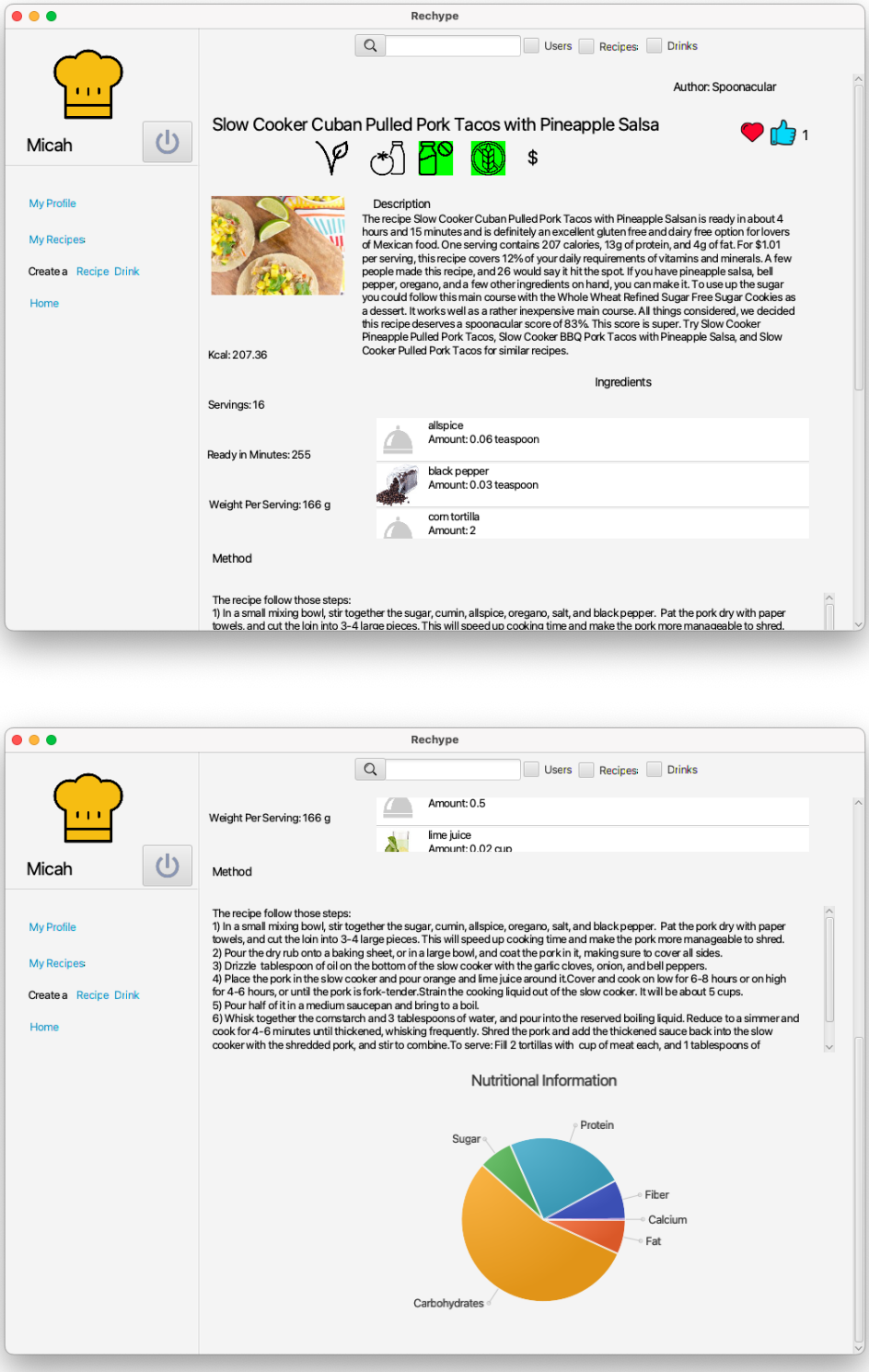

Figure 8.4: Recipe page

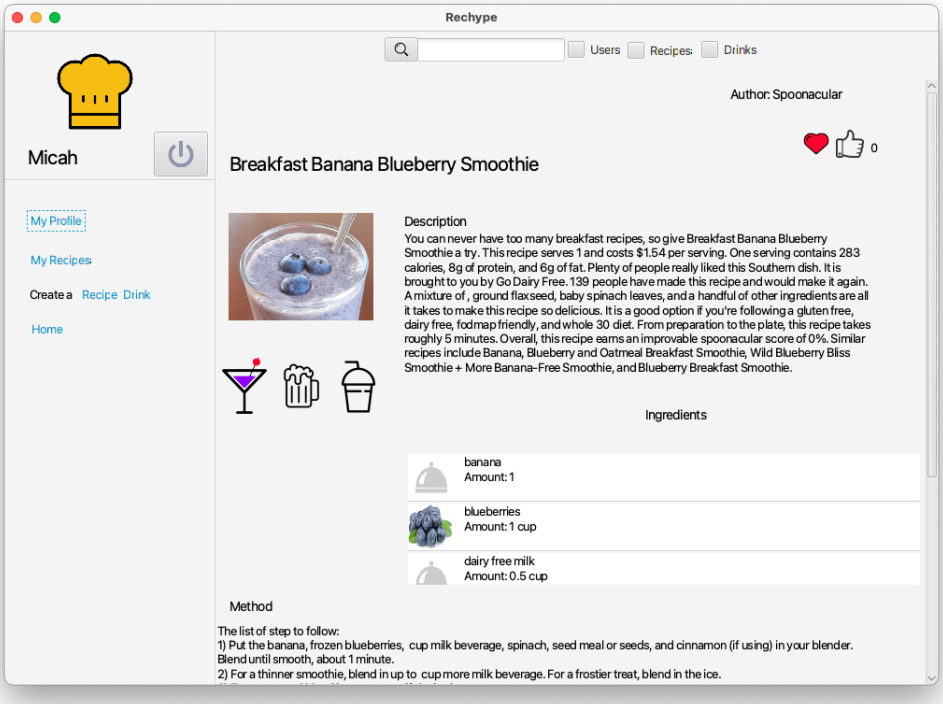

Figure 8.5: Drink page

From the side bar you can go to the page in which you can create your personal recipe by completing the form shown in figure 8.6. You can choose all the ingredients from the dedicated page 8.8. Once you have added your recipe you will be redirected to its page. You can also create a drink in the same way with its dedicated page shown in figure 8.7

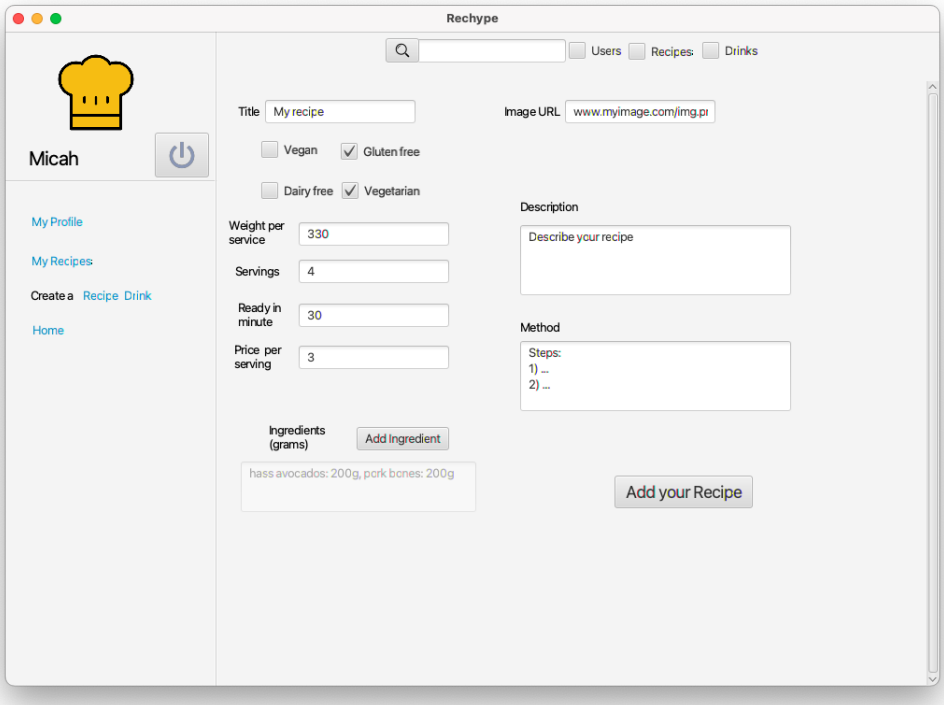

Figure 8.6: Add recipe page

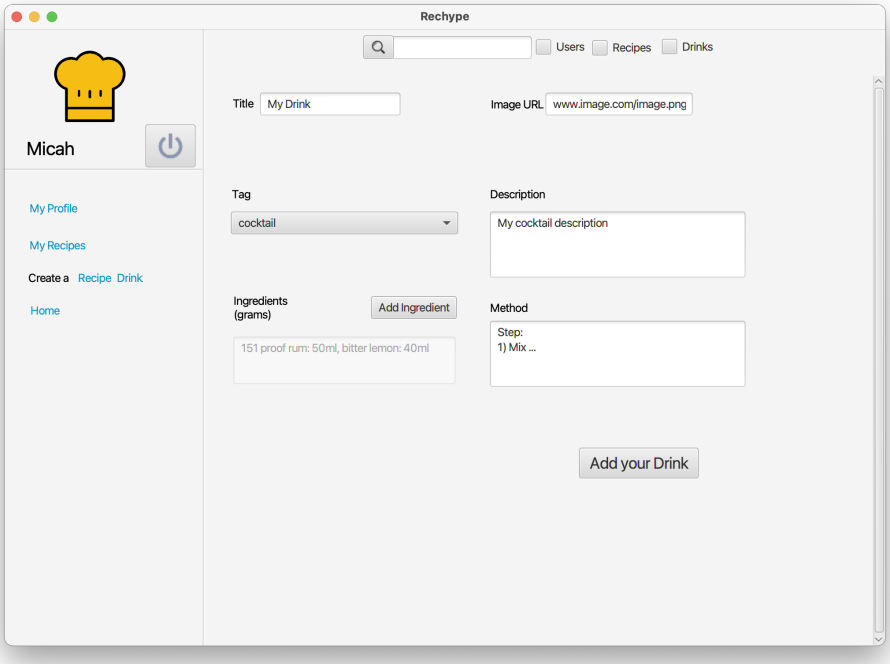

Figure 8.7: Add drink page

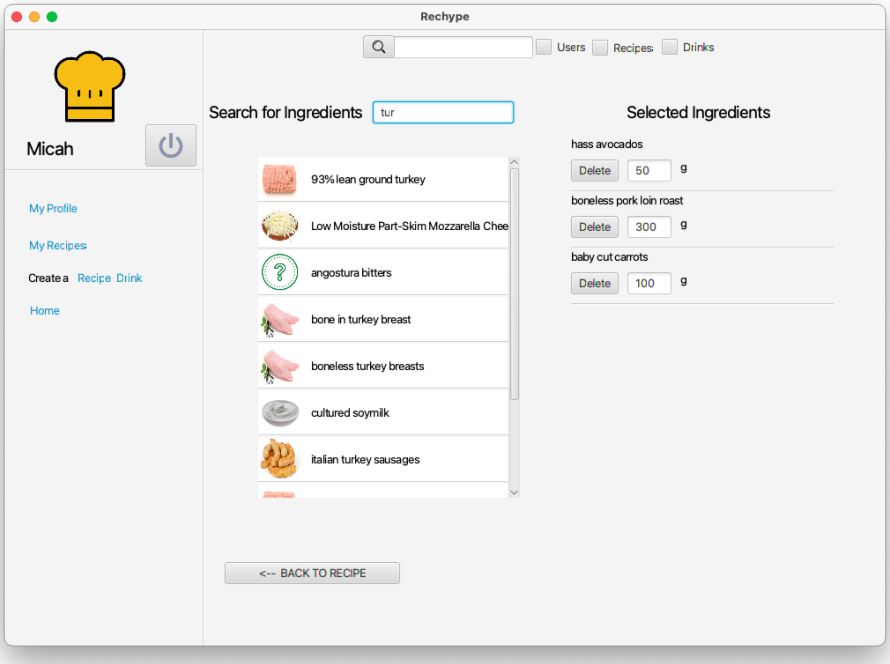

Figure 8.8: Add ingredient page

From *MyRecipes* page you can scroll through all the recipes or drinks that you have created or that you have marked as favourite like it is shown in figure 8.9.

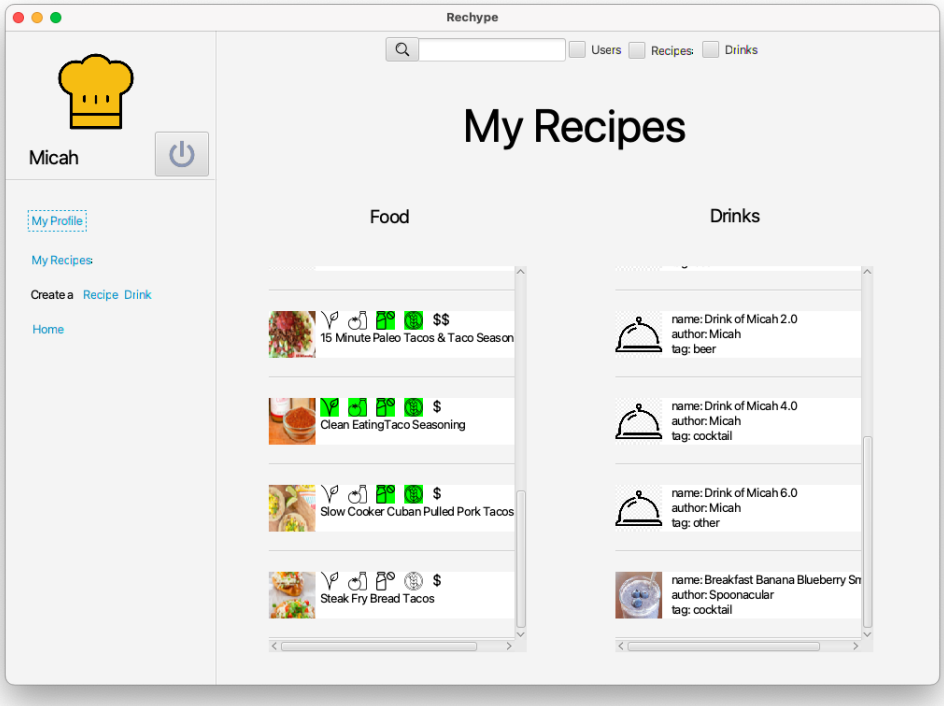

Figure 8.9: My recipe Page

From *MyProfile* you can scroll trough the meal you have created as shown in figure 8.10 or you can access the list of ingredients you have on your fridge as shown in figure 8.12, you can add the ingredients from a page similar to the one shown in figure 8.8. By clicking the *Create Meal button* you will be redirected to a dedicated page in which you can create your personal meal, it must be composed at least from 2 recipes in order to be valid, you can see the page in figure 8.11.

Figure 8.10: Profile Page (Meal)

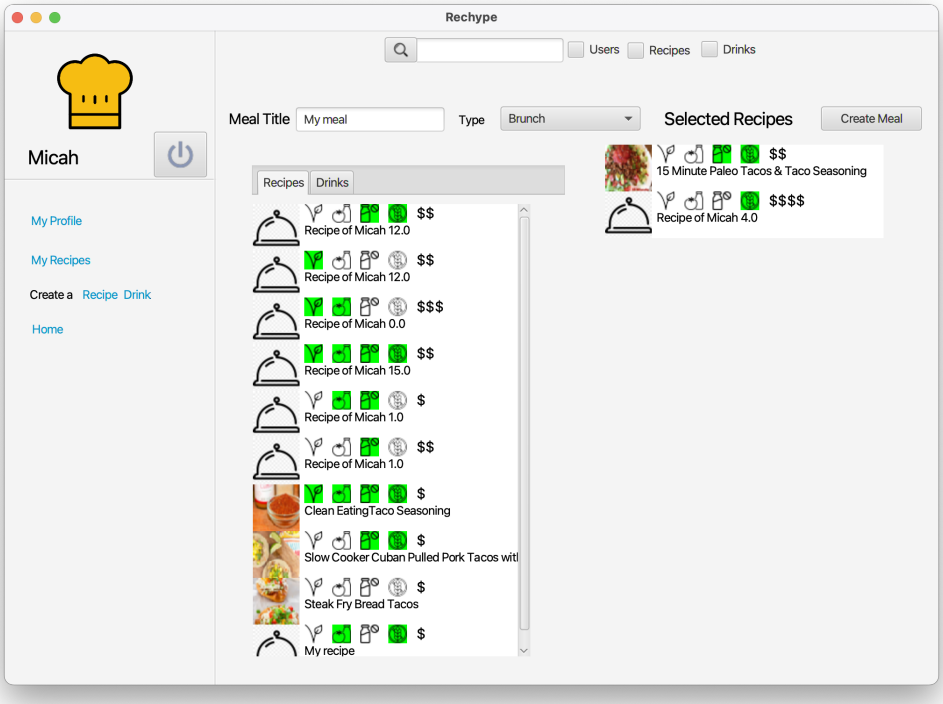

Figure 8.11: Add meal

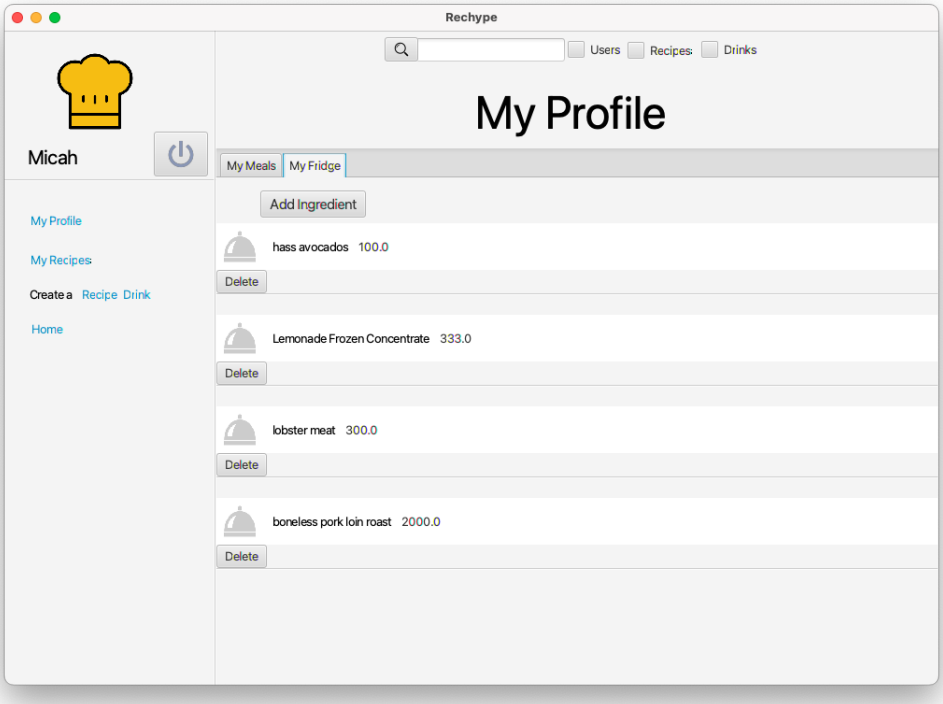

Figure 8.12: Add Profile Page (Fridge)