JustRecipe

Group Project for Large Scale and Multi-Structured Databases

Francesco Campilongo
Daniele Cioffo
Francesco Iemma

Academic Year 2020/21

Contents

Introduction

In the social network era the large scale databases topic is very relevant. The handling of big data such as informations of users and moreover is a critical asset of our society. In fact, from the viewpoint of the security is very important to handle in the correct way this very huge amount of data because, otherwise, it is possible to have leak of critical information that cause critical issues about users privacy.
Another problem caused from large amount of data created from the applications used every day from all of us, is the following: *How we can manage this data?*.
Nowadays we have a lot of tools to do this, the most famous, and maybe the most used, is for sure MongoDB. It allows us to handle a huge amount of data without critical issues and maintaining good performance. Another well-known tool is Neo4J (and so Graph DB) that is in charge of handle the social part of the application, in fact the graph is indeed a network, and so this is the best way to represents a social network where the interactions between users are fundamental and very widespread.
Our aim is to design and implement a modern application which can handle a huge amount of data maintaining good performance and implementing a social network side in order to exploit the desire for social relations of our society.

Chapter 1

Dataset

In this first chapter of this document we will talk about searching for the initial dataset.
As specified in the project documentation, the dataset had to be at least 50MB large, and this quantity could not be generated directly within the application. So we did an initial search, finding two datasets, which were generated by their authors by performing the scraping on the sites *www.FoodNetwork.com* and *www.Epicurios.com*. The second dataset was more complete (more nutritional values), so it was used as the main dataset. The other dataset was used to complement the other, reaching a total of 67.8 MB, with 45349 recipes.
To correctly extract the data present in the two datasets we wrote a program in Java, called *RecipeReade*r, thanks to which we adapted the two different formats and removed the duplicates (recipes with the same title that were present in both datasets). To implement this program we used the GSon library and the Jackson library.
The variety property is ensured by using two different sources. The velocity / variability properties are ensured because comments and recipes are eliminated and added inside the application, indeed this data may lose importance after a certain time interval since new data quickly arrives.

Chapter 2

Design

2.1 Introduction To The Application

The topic of cuisine is extensively widespread in our society. In fact we can think at the success achieved by tv shows related to cooking in the last years and also at the fact that a lot of chefs are becoming superstars. Then there is another important factor: the coronavirus outbreak.
With the coronavirus outbreak a lot of people became cuisine lovers, in fact at the first moments of the pandemia several ingredients as flour and yeast were very hard to find, because people were confined in their home and so they had more free time.
But this topic is not a recente one. The first recipe book dates back to eigth century B.C. and it is the so-called *Eraclio* (by the name of the city in which he was found). Then also an important latin writer, Apicio, wrote one of the most important recipe books of the roman era: *De Re Coquinaria* which dates back to the first century B.C..
So the topic of cuisine is inherent to human nature, because the necessity of eating is a basic need. Furthermore, everyone has experimented the infamous question: "What will I eat this evening?". JustRecipe has the aim of answer to this question, it has the aim of helping university student or workers to retrieve and to do fast and simple recipes.
So this application is basically a recipe book but it is also more than this.
JustRecipe is also a social network which allow people to enjoy, to ex-change ideas about cooking, to feel less lonely in this hard period.

2.2 Requirements

2.2.1 Main Actors

The main actors of the application are four:

- Unregistered User

  He is the user which open the application for the first time, in order to access he must sign-up.

- User

  He is the normal user (the registered one).

- Moderator

  He is in charge of controlling the comments and eventually delete the ones which contain abuses.

- Administrator

  He is the most powerful actor, he can delete users and recipes and he is also in charge of elect moderators

Each actor can do all the features of the previous ones in the list.

2.2.2 Functional Requirements

Features offered to the Unregistered User

- Registration

  In order to access the application an user must sign-up. Otherwise he is not allowed to access and to use all the functionalities.

Features offered to the Registered User

- Login/Logout

  The only way to access the application, as we said previously, is to sign-up and login. At the end the user can logout and close the session.

- Search a recipe

  It's possible to search a recipe searching for the title and for categories.

- Browse suggested recipes

  The suggestions will be offered in a proper section, they are done considering the relationships between the user logged, the users followed by the user logged and so on so forth.

- Browse recipes of following users

  In a proper section (i.e. the Homepage) the user can browse the recipes of the following user. Indeed he can see only a snapshot of the recipes. If he wants a more in-depth view, he can click on it and see the recipe page in which he is able to see all the recipe details.

- Add a recipe

  The user can insert a new recipe.

- Edit own recipes

  The user can edit the recipes previously added by himself.

- Comment recipes

  Every user can make a comment about recipes

- Follow another user

  The most important feature of each social network: the users can follow each others.

- Like a recipe

  In order to evaluate a recipe each user can like its.

Features offered to the Moderator

- Delete comments

  The moderator is in charge of delete comments which contain racist abuse, crude terms and so on so forth.

Features offered to the Administrator

- Delete users

  The admin can delete the users which don't respect the application guidelines.

- Delete recipes

  The admin can delete recipes not correctly inserted

- Elect moderators

  In order to handle better the application, the admin can elect some users as moderators.

2.2.3 Non-Functional Requirements

The non-functional requirements of the applications are described in the following lists:

- *Usability*: The application must be user-friendly so a GUI is adopted. A low response time is necessary in order to avoid too long waits for the user.
- *Data Availability*: For modern shared-data systems and for a social network application, the most crucial requirement is that the data must be always available as the service too.
- *Tolerance to single point of failure*: If a server crashes another one is available, this is ensured by the use of replicas.
- *Data Consistency*: The operations must be monotonic and so all the users must see the last version of the data and the update operations must be performed in the same order in which they are issued.
- *Reliability*: The application must work without crashes and so it must handle exceptions if they occur.
- *Flexibility*: Due to the fact that the recipes attribute are not all mandatory, the data must be handled in a flexible way.

2.2.4 Actors and Use Cases

The use case diagram of the application is described in the figure 2.1

Figure 2.1: Use Case Diagram

Some observations on the diagram are necessary:

- The circles in blue are the ones which described actions available only for the owner of the object on which the actions are applied.

  So, in detail, this means that a **User** can edit/delete a profile if and only if he owns this profile. Then he can edit a recipe and/or a comment if and only if he adds that recipe or that comment.

- When we are seeing the recipe detail we can go on the user that have been added that recipe. So the extend relation between *View Recipe* and *View User* means this.

- The action *Browse Recipes of following users* is available only if the **User** follows at least one user. Otherwise he can start to follow users and only after this he can see suggested recipes (*Browse Suggested Recipes*). In this case, due to the fact that the user follows nobody, he will see the most famous recipes in general because it's impossible to suggest specific recipes due to the fact that he has no following and no likes or comments.

- The actions in red are the ones that can be performed only by the **Administrator**
- The actions in orange are the ones that can be performed only by the **Moderators**

2.3 UML Class Diagram

Let analyze the UML Class Diagram. There are three main entities: User, Comment, Recipe. It's important to point out that the **User** of the Use Case Diagram is here the so-called *Registered User* and the *User* of this diagram is the generic user. Then we underline the fact that, in order to represent the three actors of the use case diagram, a generalization is needed.

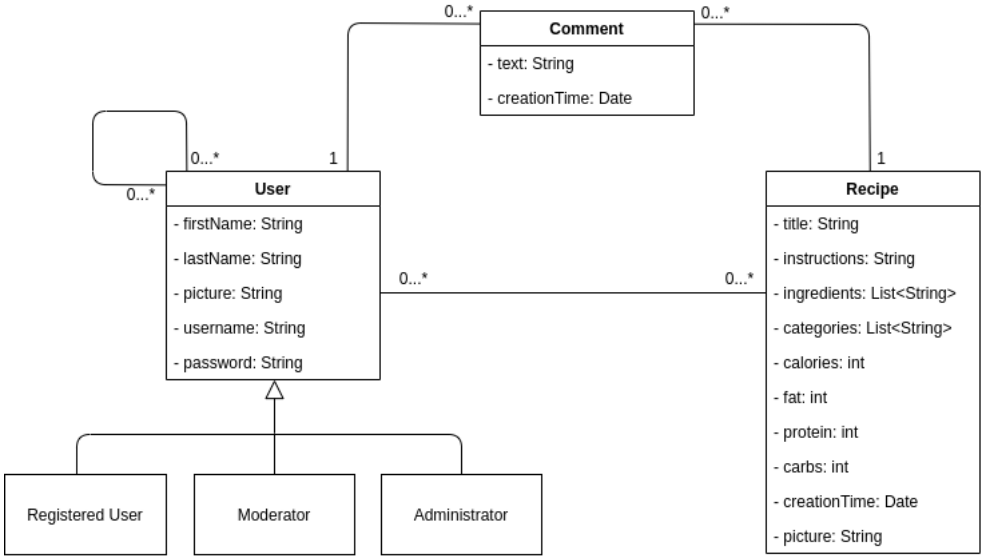

Figure 2.2: UML Analysis Classes Diagram with generalization unsolved

Observing the figure 2.2 it's possible to understand that we can resolve the generalization putting an attribute in the entity *User*. It is an integer and we call it *role*: if it's a *Normal User* role is 0; if *Moderator* then 1; if *Administrator* then 2.

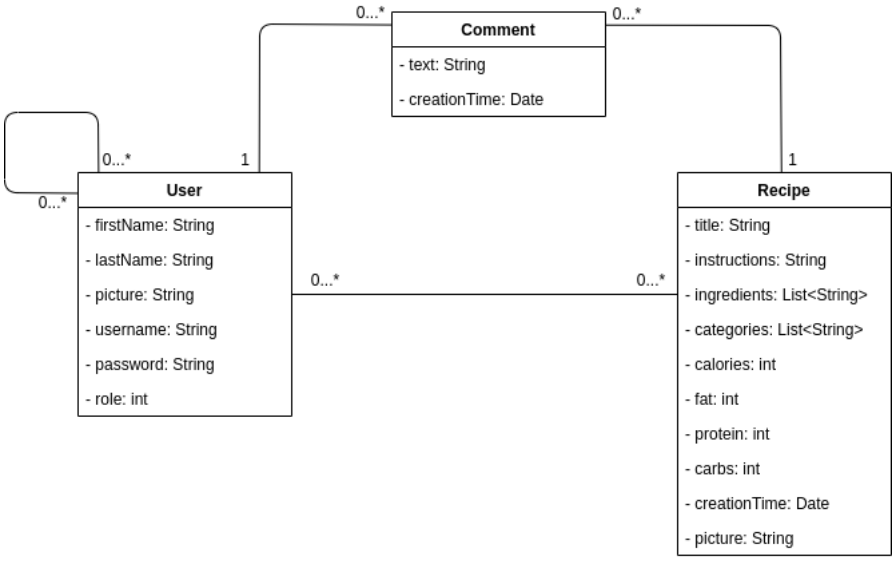

Figure 2.3: UML Analysis Classes Diagram

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

Table 2.1: Classes definitions

Class	Description
Registered User	A standard user (registered one) who can only perform basic operations
Moderator	User who can also check comments and decide to delete them
Administrator	Most powerful user, he can also delete users and recipes, and elect moderator
Comment	Comment posted by a user
Recipe	Recipe added by one user

Table 2.2: Classes attributes - User

Attribute	Type	Description
firstName	String	First name of the user
lastName	String	Last name of the user
picture	String	URL of the picture
username	String	Username of the user (identifier)
password	String	Password chosen by the user, used for the login phase
role	int	Role of the user (0: Normal User, 1: Moderator, 2: Administrator)

Table 2.3: Classes attributes - Comment

Attribute	Type	Description
text	String	Plain text of the comment
creationTime	Date	Creation timestamp of the comment

Table 2.4: Classes attributes - Recipe

Attribute	Type	Description
title	String	Title of the recipe (identifier)
instructions	String	Operation to be performed to make the recipe
ingredients	List<String>	Ingredients to be used in the recipe
categories	List<String>	Categories to which the recipe belongs
calories	int	Calories contained in the recipe
fat	int	Fat contained in the recipe
protein	int	Protein contained in the recipe
carbs	int	Carbs contained in the recipe
creationTime	Date	Creation timestamp of the recipe
picture	String	URL of the recipe picture

2.4 Data Model

In this section we will discuss about the design choises performed in terms of data model in order to handle in the better way our dataset.

2.4.1 DocumentDB

The document database is used in order to deal with a very large amount of data in an easy and fast way. So due to the fact that our dataset is very large this is the best design choice to achieve the *usability* requirement, in particular for what is related to the latency. Moreover, we use a DocumentDB in order to respect the *flexibility* requirement, in fact it is schema-less and this allow us to maintain heterogeneous recipes in our database. For instance the user can insert a recipe without some attributes (carbs and fat for example) and we have no issues due to the schema-less property of DocumentDB. Furthermore this is very important for what concern the comments: in fact comments are embedded in the recipe document and the schema-less approach allow us to add comment in a flexible way and to have recipes without comments. So, to summarize, we use only one collection, *recipes*, in which every document is a recipe that contains also the comments. So this is how we decided to implements the *one-to-many* relationship between *Recipe* and *Comment*.
This is an example of one Recipe that contains all the value (this is not mandatory, like we said before):

```
{
  "_id":
      {"$oid": "5fdb5fd86796ee4e73ef5b84"},
  "title":
      "Lentil, Apple, and Turkey Wrap ",
  "instructions":
      "1. Place the stock, lentils, celery, carrot, thyme, and
      salt in a medium saucepan and bring to a boil. Reduce heat
      to low and simmer until the lentils are tender, about 30
      minutes, depending on the lentils. (If they begin to
      dry out, add water as needed)..." ,
  "ingredients":
      ["4 cups low-sodium vegetable or chicken stock", "1 cup
         dried brown lentils", ...],
  "categories":
      ["Sandwich", "Bean", "Fruit", "Tomato", "turkey", ...],
  "calories":
      426,
  "fat":
      7,
  "protein":
      30,
  "carbs":
      20,
  "creationTime":
      {
        "$date": "2020-12-17T13:40:40.658Z"
      },
  "authorUsername":
      "oscar.evans",
  "picture":
      "https://assets.epicurious.com/photos/551
         b0595e7851a541a30b23f/master/pass/239173_lentil-apple-
         and-turkey-wrap_6x4.jpg",
  "comments":
      [
        {
```

```
            "authorUsername": "oliver.smith",
            "text": "Very good!!!",
            "creationTime":
            {
              "$date": "2020-12-17T13:50:40.658Z"
            }
          },
          {
            "authorUsername": "jessica.evans",
            "text": "Fantastic",
            "creationTime":
            {
              "$date": "2020-12-17T13:52:40.658Z"
            }
          },....
        ]
}
```

2.4.2 GraphDB

The Graph database is the one that handles the social part of the application. Thanks to it we are able to analyze the relationships among users and their recipes.
We have two type of nodes within the database:

- **User**: it is the node that represents the user inside the graph. Its attribute are: *firstName*, *lastName*, *username*, *password*, *picture*[1], *role*[2].
- **Recipe**: it is the node that represents the recipe within the graph. Its attribute are: *title*, *calories*, *fat*, *carbs*, *protein*, *picture*[1]

The information about the user are present only on the GraphDB because we handle the social part only with it. Instead some information about recipes are present in both databases, in particular on GraphDB we have info that are already present also in the DocumentDB, but the opposite is not true.

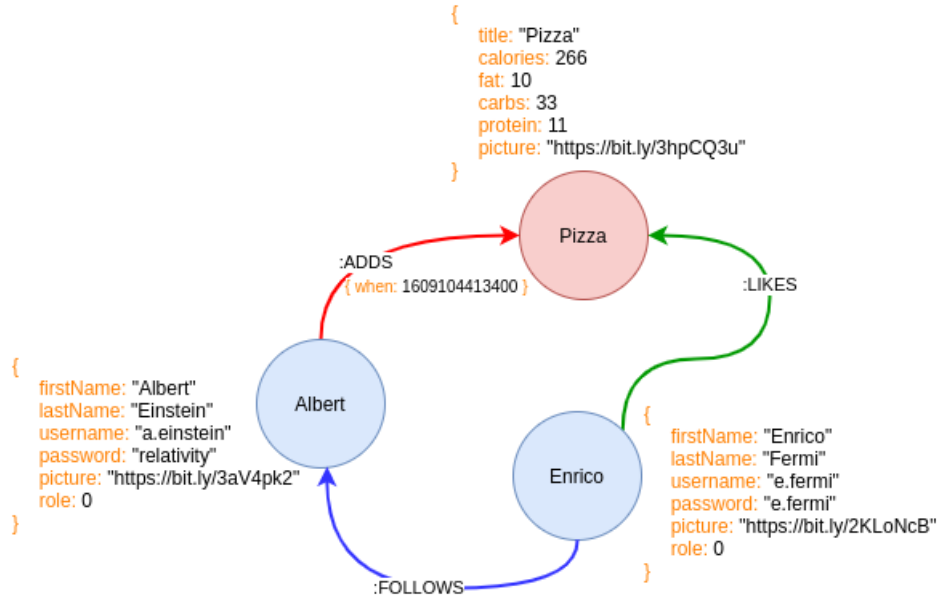

Figure 2.4: Example of GraphDB nodes and relationships

[1]Only the URL of the picture is saved in this field, otherwise the size of this attribute would be too large
[2]0 for the normal user, 1 for moderator and 2 for the administrator

The GraphDB relationships are:

- **Adds**: if the user A has added the recipe R, then we have a relationship *:ADDS* from A to R ($A \rightarrow R$).

  We have as property, the one called *when* that is the timestamp which indicates when the recipe has been added.

- **Follows**: if the user A has followed the user B, then we have a relationship *:FOLLOWS* from A to B ($A \rightarrow B$).

- **Likes**: if the user A has liked the recipe R, then we have a relationship *:LIKES* from A to R ($A \rightarrow R$).

All the relationships that starts from a node X are deleted if X has deleted, furthermore if X is a user, also the recipes that he has added will be eliminated. In figure 2.4 we can see an example of the nodes, the relationships and the attributes.

2.4.3 Data Among Databases

Because of 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 faster and easiest way, in order to have a good *usability*. 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

The reason why we store recipes on both databases is because we want to retrieve social network information from it (likes, suggested recipes based on who added it, and so on). Hence we store on Graph DB a part of the recipe information, and the whole information about recipes on DocumentDB. This partial view of the recipe is the so-called *recipe snapshot*. Also for the user exist a snapshot but it exists for implementation reason, in fact in this case all the information about the user are on GraphDB but only a part is in the snapshot. Generally speaking a snapshot is a partial view of an entity (User or Recipe). When the user clicks on the snapshot then he will see all the information (so for what concerns the recipes, the snapshot is done with GraphDB and the detailed page with DocumentDB).
We can see the storing strategy also in figure 2.5.

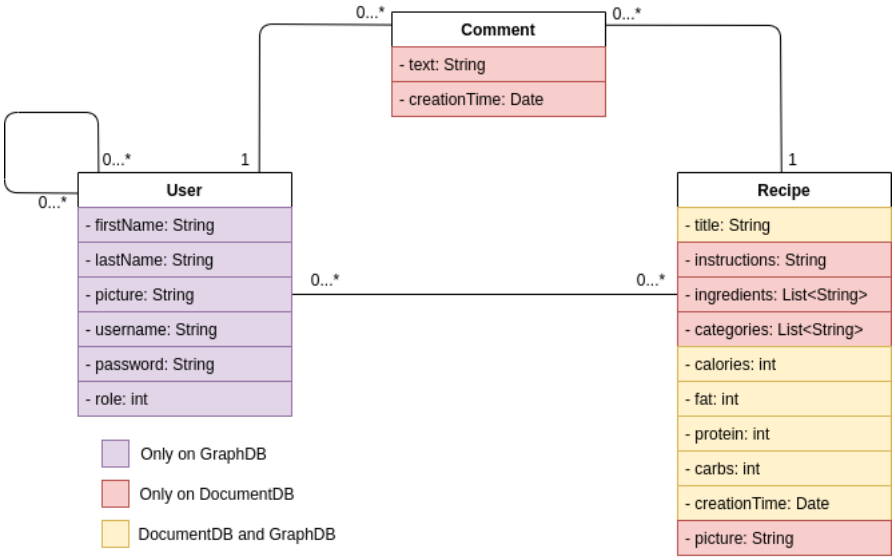

Figure 2.5: Storing Strategy

Another important detail on the Data Model is that we have chosen MongoDB as DBMS for the DocumentDB and Neo4j as DBMS for the GraphDB.

2.5 Distributed Database Design

In our application some of the most important requirements are the *Availability* and the *Consistency*. Thus is very important to us to have a distributed system in order to ensure:

- An high availability, due to the fact that we have more servers (in our case we have 3 virtual machines provided by University of Pisa[3]) and so we haven't the single point of failure issue.
- An high consistency, because, even if the data on a server would be corrupted (for a server malfunction for instance), we have an update replica that maintains the correct data (see also 2.5.2).
- A low latency, in fact thanks to sharding and replicas the load is balanced among the cluster servers. We have to remark that this is not always true because, in order to maintain consistency, it is possible to have a longer latency with respect to the one that we would achieve without consistency (replicas must be updated and this requires time).

This two objectives can be reached using replicas (copy of the same data on different servers) and sharding (the dataset is split up into different parts, each part is on a different server of the cluster or on more than one server depending from the design choises performed).
In the following sections we discuss about these two techniques explaining the design choices performed in order to obtain the best performance achievable and in order to respect the given requirements.

2.5.1 Replicas

We decide to use replicas of the same data in order to ensure *availability* and *consistency*. Nevertheless we have to point out that replicas alone don't ensure consistency, because it's possible to have one or more replicas that are not updated, this doesn't happen in our case because *in order to commit a write operation all replicas must be updated*. This because the application is mainly read-heavy and so we need fast reads operation and, in order to achieve this, we can accept slow write operations. Moreover, this configuration ensures the *Consistency* requirement.
In particular 3 replicas are present in our system: one for each machine of the provided cluster. However in our implementation only for MongoDB we have three replicas, for Neo4J we have only one instance running on server 172.16.3.107: theoretically speaking the Neo4J replicas must be three but it's a premium feature.
So, in order to sum up, let's point out in the following list, the most important points about replicas that must be taken into account.

- 3 replicas for MongoDB, one for each server of the cluster
- 1 instance for Neo4J on 172.16.3.107
- All replicas must be updated in order to commit a write operation

As it was expressly requested in the project documentation, a local cluster was also implemented, thanks to which it was possible to test the use of the application even with replicas on Neo4j, in addition of course to those for MongoDB.

2.5.2 Sharding

It's possible to use the following fields as shard key (because they are present in all documents of the collection):

- Object ID (provided by MongoDB)
- Title
- Creation Time

[3] 172.16.3.157 - 172.16.3.107 - 172.16.3.108

- Instructions
- Ingredients

Using *Instructions* and *Ingredients* has no sense. For what concerns the creation time we search for the most time the most recent recipes, so if we would use it as shard key, we will access always to the shard that contains the last added recipes and the sharding strategy will be unbalanced and so it is useless. At the end we choose the *Object ID* provided by Mongo because it is an automatically generated value which is better than the *Title* that is inserted by the user.

2.6 Software Architecture

In this section the aspects related to the software architecture will be considered, mainly by analyzing the division of tasks between the various modules of the application. The main language used in developing the application is Java.

2.6.1 Client - Server Architecture

The application was implemented as a client-server architecture, with middleware implemented on the client side.

Client Side

The client features can be divided into two modules:

- The front-end module, which consists of a graphical interface based on JavaFX, which allows the users to use the application in the simplest way possible. This is critical to the usability requirement.
- A middleware module, needed to interface the client with the server. More precisely, a driver has been implemented to interface with MongoDB and one to interface with Neo4j.

The application was developed following the MVC (Model, View, Controller) pattern. The View displays the data contained in the model and was mainly developed using .fxml files, which allow you to write all the graphic components and their properties in separate files. The Model provides methods for accessing data useful for the application. The Controller receives the commands from the user and implements them by changing the status of the other two components. This division allowed us to completely separate the three components, for a more readable and maintainable code.
Maven was used to support the implementation of the application.

Server side

The server side, as already mentioned, consists of three virtual machines, on which MongoDB and Neo4J are executed.

2.6.2 Inter-Databases Consistency

Considering how the data was distributed between the two databases, we must consider the problem of consistency on the information that forms the snapshots of the recipes, which are the only information shared by the two databases. The general idea is that if I have to update the data of both databases I have to consider the case where the first update is successful and the second is not (I have to undo the first operation), and the case where the first update is not successful (and therefore I don't even have to do the second). The cases that can lead to these inconsistencies are the following:

- Deleting a recipe
- Adding a recipe
- Update a recipe
- Deleting a user (in this case his recipes must also be deleted)

MANCANO I DISEGNI

Chapter 3

Implementation and Test

3.1 Main Modules

The implementation code is divided into two main modules: RecipeReader and JustRecipe.

- *RecipeReader*: is a program we wrote to get our initial dataset, extracting the information that interests us from the two initial datasets (see chapter 1).
- *JustRecipe*: is the actual application, whose implementation will be analyzed in more details in the next sections.

3.2 Main Packages and Classes

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

3.2.1 it.unipi.dii.inginf.lsdb.justrecipe.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.

3.2.2 it.unipi.dii.inginf.lsdb.justrecipe.main

This package contains the Main class, that starts the application.
Classes:

- *Main*: this class extends *Application* and implements the *start* method.

3.2.3 it.unipi.dii.inginf.lsdb.justrecipe.model

This package contains the classes required for the model. These classes are the java bean for our application.
Classes:

- *Comment*: This class stores all the information about a comment, like the text, the username of the author 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.
- *Session*: This class is used to maintain the information of the session, like the logged user. We used the singleton design pattern for this class.

3.2.4 it.unipi.dii.inginf.lsdb.justrecipe.persistence

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

- *DatabaseDriver*: this *interface* declares all the methods that has to be implemented in a database driver. This methods are *initConnection()* and *closeConnection()*.
- *MongoDBDriver*: this class implements DatabaseDriver and is responsible for implementing all the queries that have to be run on MongoDB. We used the Singleton design pattern, because a single instance of this driver must be shared by all application classes.
- *Neo4jDriver*: this class implements DatabaseDriver and is responsible for implementing all the queries that have to be run on Neo4j. We used the Singleton design pattern, because a single instance of this driver must be shared by all application classes.

3.2.5 it.unipi.dii.inginf.lsdb.justrecipe.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:

- *WelcomePageController*: this class manages the login/register page of the application.
- *HomePageController*: this class handles the homepage section of the application (shows the recipes of following users and handles the event, like the click on a recipe snapshot).
- *DiscoveryPageController*: this class manages the discovery section of the application. So it is in charge of showing the results of search made by the user.
- *ProfilePageController*: this class manages the profile section of the application. The profile could be either mine or someone else's. Thanks to this class it is possible to manage some events such as following a user, deleting my profile, seeing his recipes, and so on.
- *AdministrationPageController*: this class handles the Administration section of the application. Of course, not all the users can access to this page, only the moderator (for seeing the last comments) and the administrators (for doing all the possible operations) can access.
- *EditProfilePageController*: this class manages the page for editing the personal profile. The user can insert a new profile image, change the personal information, and so on. The administrator can change the role of the user in this page.
- *AddRecipePageController*: this class manages the page of the application used for insert a new recipe.
- *RecipeSnapshotController*: this class manages the single recipe snapshot, like the click on him.
- *RecipePageController*: 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.

- *CommentController*: this class manages the single comment and all the operations that can be done on him.
- *UserSnapshotController*: this class manages the single snapshot of one user, and all the operations that can be done on him.

3.2.6 it.unipi.dii.inginf.lsdb.justrecipe.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).

3.3 Constraints

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

- Constraint on the *title* of the recipe (on MongoDB):

```
db.recipes.createIndex (
  {
    title:1
  },
  {
    unique: true,
    name: "title_constraint"
  }
)
```

- Constraint on the *title* of the recipe (on Neo4j):

```
CREATE CONSTRAINT title_constraint
ON (r:Recipe)
ASSERT r.title IS UNIQUE
```

- Constraint on the *username* of the user:

```
CREATE CONSTRAINT username_constraint
ON (u:User)
ASSERT u.username IS UNIQUE
```

3.4 Most Relevant Queries

In the following section will be presented the most relevant query performed with MongoDB and Neo4J. Some important point must be underlined:

- The operation of *skip* and *limit*, use in the most of the following queries, are necessary due to implementation reasons. In fact the results are not shown all in the same page (because of the size of the database the output can be huge) but only a subset of the result are shown (for instance, the first X) and in order to go on and see the other result, the query will be performed with different value of *<howManyToSkip>* (the first time is 0, the second is X, the third 2X and so on) and *<howManyToGet>* (it's a fixed number, must be equal to the previous X)
- In Neo4J part, due to performance reasons, some queries compute also followers, followings and recipes added by each user. This is necessary because when the user is given as result this information must be showed and if we don't compute it here we have to do another query for each user.

3.4.1 MongoDB

Search Recipes given a username

This query gives the opportunity to collect all the recipes written by a specific User through checking the author username field. This search is case sensitive cause we are using the username.

- Input: a string who represent an username (unique through the users), how many recipe to skip, and how many to get.
- Output: a list of recipes, all added by the user erlier selected, ordered by the .

```
db.recipes.aggregate (
  [
    {
      $match :
      {
        authorUsername: <authorUsername>
      }
    },
    {
        $sort :
      {
        creationTime : -1
      }
    },
    {
      $skip: <howManyToSkip>
    },
    {
      $limit: <howManyToGet>
    }
    ]
)
```

Search for recipe title

Given a portion of the title, this query is capable of returning as a result a set of recipes whose titles contain it. Research must be case insensitive (see *options:"i"* in the match step of the aggregation)

- Input: portion of the title, how many recipes to skip and how many recipes to get.
- Output: set of recipes.

```
db.recipes.aggregate (
  [
    {
      $match :
      {
        title:
        {
          $regex: /^.*<portionOfTheTitle>.*$/,
          $options: "i"
        }
      }
    },
    {
      $sort :
```

```
      {
        creationTime : -1
      }
    },
    {
      $skip: <howManyToSkip>
    },
    {
      $limit: <howManyToGet>
    }
  ]
)
```

Get the username of the most versatile user

This query let select the most versatile user, the one who had covered more recipe categories (if he/she added at least 5 different recipe in a *category*, than this *category* is covered).
The users are sorted by the number of distinct categories. If users have the same number of distinct categories the lexicographic order is taken into account.

- Input: *no input*
- Output: For this early implementation of this application, we need only the best most versatile user (the first one, his/her username)

```
db.recipes.aggregate
(
  [
    {
      $unwind : "$categories"
    },
    {
      $group :
      {
        _id:
        {
          "author":"$authorUsername",
          "category":"$categories"
        },
        numRecipe:
        {
          $sum:1
        }
      }
    },
    {
      $match:
      {
        numRecipe:
        {
          $gte: 5
        }
      }
    },
    {
      $group :
      {
        _id : "$_id.author",
        distinctCategories:
```

```
        {
          $sum: 1
        }
      }
    },
    {
      $project:
      {
        'username': '$_id',
        distinctCategories: 1,
        _id: 0
      }
    },
    {
      $sort :
      {
        distinctCategories: -1,
        username: 1

      }
    },
    {
      $limit: 1
    }
  ]
)
```

Search for recipe category

Given the category, or a part of this, the query returns recipes that belong to it. The search is case insensitive.
It's important to point out that a recipe belongs to more than one categories, so a recipe can be part of the result of the search of two different categories (this is normal and it is allowed).

- Input: portion of the category or the entire one *<portionOfTheCategory>*, how many recipes to skip *<howManyToSkip>* and how many to get *<howManyToGet>*.
- Output: set of recipes belonging to the given category

```
db.recipes.aggregate (
  [
    {
      $match :
      {
        categories:
        {
          $regex: /^.*<portionOfTheCategory>.*$/,
          $options: "i"
        }
      }
    },
  {
    $sort :
    {
      creationTime : -1
    }
  },
  {
    $skip: <howManyToSkip>
```

```
  },
  {
    $limit: <howManyToGet>
  }
  ]
)
```

Search for recipe ingredients

This query give the possibility to find a specific number of recipes given some of the recipe ingredients (or a portion of them). The search is case insensitive for a more easy to use experience.

- Input: a list of ingredients, how many recipes to skip and how many recipe to show.
- Output: a list of recipes, all of them have at least all the ingredients specified before. The results are ordered from the newer to the older recipe.

It is important to say that there will be a *match* step for each ingredients specified by the user.

```
  db.recipes.aggregate (
  [
  {
    $match :
    {
      ingredients:
      {
        $regex: /^.*<ingredients[i]>.*$/,
        $options: "i"
      }
    }
  },
  {
    $sort :
    {
      creationTime : -1
    }
  },
  {
    $skip: <howManyToSkip>
  },
  {
    $limit: <howManyToGet>
  }
  ]
 )
```

Most common recipe categories

This query allows you to get a ranking of the categories most used by users for their recipes.

- Input: how many categories to skip, how many categories to get.
- Output: list with the categories ordered by the number of use.

```
db.recipes.aggregate (
  [
    {
      $unwind : "$categories"
    },
```

```
    {
      $group :
      {
        _id : "$categories",
        numberOfRecipes:
        {
          $sum: 1
         }
      }
    },
    {
      $project:
      {
        'category': '$_id',
        numberOfRecipes: 1,
        _id: 0
      }
    },
    {
      $sort :
      {
        numberOfRecipes : -1
      }
    },
    {
      $skip: <howManyToSkip>
    },
    {
      $limit: <howManyToGet>
    }
  ]
)
```

Last Comments

The query returns the last comments sorted for creation time, in order to have the most recent as first and the most old as last.

- Input: *howManyToGet* and *howManyToSkip*
- Output: set of the last comments sorted by creation time in descendant order.

```
db.recipes.aggregate (
  [
    {
      $unwind : "$comments"
    },
    {
      $sort :
      {
        "comments.creationTime" : -1
      }
    },
    {
      $skip: <howManyToSkip>
    },
    {
      $limit: <howManyToGet>
    }
```

```
  ]
)
```

Weekly Recipes per User

A query who returns all the recipes between two dates, first day of the week and first day of the next week.

- Input: *Username* who represent the user.
- Output: set of recipes which the user has added in the current week.

```
db.recipes.aggregate {
  $match:
    {
      $and:
        [
          {
            creationTime
            {
              $gte: <firstDateOfWeek>,
              $lt: <firstDateOfNextWeek>
            }
          },
          {
            authorUsername: <username>
          }
        ]
    }
  }
```

3.4.2 Neo4J

Now we will talk about the queries on Neo4j, but first we need to consider their domain-specific to graph-centric translation.

Table 3.1: Graph-Centric Query and Domain-Specific Query

Graph-Centric Query	Domain-Specific Query
How many incoming edges are incident to vertex A?	How many users follow one user? How many likes has a recipe received?
How many hops does it take to get from vertex A to vertex B?	How many follows relations are between User A and User B? (used within the search for suggested recipes)
What is the centrality measure of vertex B? (How well connected is?)	Most followed and active users, Best recipes, Most liked users

A partial example of a graph (screenshot made during the execution of the program) is shown in the figure 3.1 , in order to have more clear the concepts that will be expressed later in the implementation of the queries.

Figure 3.1: Partial example of the graph in Neo4j

Suggested Recipes

We have two levels of suggestions with different relevance.

- First Level

  Recipe R is a first level suggestion for the user X, if R has been added by the user Y where Y is followed by W who is followed by X ($X \rightarrow W \rightarrow Y$). Or R has been added by Z where Z is followed by Y ($X \rightarrow W \rightarrow Y \rightarrow Z$).

- Second Level

  Recipe R is a second level suggestion for the user X if R has been added by the user Y where Y is the owner of at least N recipes liked by X.

The two level of suggestions are in a single query, the first recipes are the ones related to the first level, then at the end we have the ones related to the second level. The code is show below:

```
MATCH path =
(recipe:Recipe)<-[a:ADDS]-(owner:User)<-[:FOLLOWS*2..3]
-(me:User{username:$u})
WHERE owner.username <> $u
RETURN recipe.title, recipe.calories, recipe.carbs,
recipe.protein, recipe.fat, recipe.picture,
a.when, owner.username
ORDER BY length(path) ASC, a.when DESC
SKIP $howManyToSkipFirstLv
LIMIT $howManyToGetFirstLv
UNION
MATCH (:User{username:$u})-[l:LIKES]->(:Recipe)<-[:ADDS]-
(owner:User)
```

```
14  WITH DISTINCT(owner) AS owner, COUNT(DISTINCT l) AS numLikes
15  WHERE numLikes > $threshold AND owner.username <> $u
16  MATCH (owner)-[a:ADDS]->(recipe:Recipe)
17  RETURN recipe.title, recipe.calories, recipe.carbs,
18  recipe.protein, recipe.fat, recipe.picture, a.when,
19  owner.username
20  ORDER BY a.when DESC
21  SKIP $howManyToSkipSecondLv
22  LIMIT $howManyToGetSecondLv
```

Recipes of following users

This query returns all the recipes of the following users of a specific user.

- Input: How many recipe name to skip, how many recipe name to show and a string who represents a specific username.
- Output: A set of recipe name of the specific user's following users have added.

```
1  MATCH (u1:User{username:$username})-[:FOLLOWS]->(u2:User)-[a:
       ADDS]->(r:Recipe)
2  RETURN r.title AS title, r.calories AS calories,
3     r.fat AS fat, r.protein AS protein, r.carbs AS carbs,
4     r.picture AS picture, u2.username AS authorUsername
5  ORDER BY a.when DESC
6  SKIP $skip LIMIT $limit
```

Most followed and active users

This query returns the list of the most followed and active users, namely the influencers. Most followed means that the list is ordered by the number of followers. Active means that the list is ordered by the number of recipes added by the user.

- Input: How many users to skip, how many users to show.
- Output: A list of the most followed and active users.

```
1  MATCH (u:User)
2  OPTIONAL MATCH (u)<-[f1:FOLLOWS]-(:User)
3  OPTIONAL MATCH (u)-[f2:FOLLOWS]->(:User)
4  OPTIONAL MATCH (u)-[a:ADDS]->(:Recipe)
5  RETURN u.firstName, u.lastName, u.username,
6    u.firstName AS firstName, u.lastName AS lastName,
7    u.picture AS picture, u.username AS username,
8    u.password AS password, u.role AS role,
9  COUNT(DISTINCT f1) AS follower,
10 COUNT(DISTINCT f2) AS following,
11 COUNT(DISTINCT a) AS numRecipes
12 ORDER BY follower DESC, numRecipes DESC
13 SKIP $howManySkip LIMIT $howMany
```

Most liked users

This query returns the list of the most liked users, namely the users who received more like to their recipes.

- Input: How many users to skip, how many users to show.
- Output: A list of the most liked users.

```
MATCH (u:User)
OPTIONAL MATCH (u)-[:ADDS]->(:Recipe)<-[l:LIKES]-(:User)
OPTIONAL MATCH (u)<-[f1:FOLLOWS]-(:User)
OPTIONAL MATCH (u)-[f2:FOLLOWS]->(:User)
OPTIONAL MATCH (u)-[a:ADDS]->(:Recipe)
RETURN u.firstName, u.lastName, u.username,
  u.firstName AS firstName, u.lastName AS lastName,
  u.picture AS picture, u.username AS username,
  u.password AS password, u.role AS role,
COUNT(DISTINCT f1) AS follower,
COUNT(DISTINCT f2) AS following,
COUNT(DISTINCT a) AS numRecipes,
COUNT(DISTINCT l) AS totLikes
ORDER BY totLikes DESC
SKIP $howManySkip LIMIT $howMany
```

Best Recipes

This query returns the list of the best recipes, namely the most liked ones.

- Input: How many recipes to skip, how many recipes to show.
- Output: A list of the best recipes.

```
MATCH (:User)-[l:LIKES]->(r:Recipe)
MATCH (u:User)-[:ADDS]->(r)
RETURN r.title AS title, r.calories AS calories,
  r.fat AS fat, r.protein AS protein, r.carbs AS carbs,
  r.picture AS picture, u.username AS authorUsername,
COUNT(DISTINCT l) AS likes
ORDER BY likes DESC
SKIP $skip LIMIT $limit
```

To clarify, the first match is used to avoid to consider the recipes which have not at least one like, and the second match is used to find the user that adds the recipe.

Search for username

Given a portion of the username, this query is able to return all users whose usernames contain it. The search must be case insensitive, so the toLower function is used.

- Input: portion of the username, how many users to skip and how many users to get.
- Output: set of users.

```
MATCH (u:User)
WHERE toLower(u.username) CONTAINS toLower($username)
OPTIONAL MATCH (u)<-[f1:FOLLOWS]-(:User)
OPTIONAL MATCH (u)-[f2:FOLLOWS]->(:User)
OPTIONAL MATCH (u)-[a:ADDS]->(:Recipe)
RETURN u.firstName AS firstName, u.lastName AS lastName,
  u.picture AS picture, u.username AS username,
  u.password AS password, u.role AS role,
COUNT(DISTINCT f1) AS follower,
COUNT(DISTINCT f2) AS following,
COUNT(DISTINCT a) AS numRecipes
SKIP $skip LIMIT $limit
```

Search for user's full-name

Given the full-name or a part of it, the query returns the users that contains in their full-name the given input. The query is case-insensitive.

- Input: The full-name (*$fullName*), how many users to skip (*$skip*) and how many users to get (*$limit*).
- Output: The users that contains in their full-name the given input *$fullName*.

```
MATCH (u:User)
WHERE
toLower(u.firstName+' '+u.lastName)
CONTAINS toLower($fullName)
OR
toLower(u.lastName + ' ' + u.firstName)
CONTAINS toLower($fullName)
OPTIONAL MATCH (u)<-[f1:FOLLOWS]-(:User)
OPTIONAL MATCH (u)-[f2:FOLLOWS]->(:User)
OPTIONAL MATCH (u)-[a:ADDS]->(:Recipe)
RETURN u.firstName AS firstName, u.lastName AS lastName,
 u.picture AS picture, u.username AS username,
 u.password AS password, u.role AS role,
COUNT (DISTINCT f1) AS follower,
COUNT (DISTINCT f2) AS following,
COUNT (DISTINCT a) AS numRecipes
SKIP $skip LIMIT $limit
```

3.5 Tests and Statistical Analysis

3.5.1 Queries Analysis

The most queries consist in read operations and so we can say that the application is read-heavy. In the table 3.2 we can see the frequency of each query.

Table 3.2: Queries Frequency

Query	Frequency
Search Recipes given a username	High
Search for Recipe Title	Medium
Get the username of the most versatile user	Low
Search for Recipe Category	Low
Search for Recipe Ingredients	Low
Most Common Recipe Categories	Low
Last Comments	Medium
Weekly Recipes per User	High
Suggested Recipes	High
Recipes of Following Users	High
Most Followed an Active Users	Low
Most Liked Users	Low
Best Recipes	Low
Search for Username	Medium
Search for User's Full Name	Low

3.5.2 Index Analysis

In this subsection the indices for both MongoDB and Neo4J will be considered, and statistical tests will be performed to understand their benefit. For each index will be considered the queries

that are in some way affected by its presence. For performance measures of Neo4j queries, the keyword *PROFILE* was used at the beginning of the query, which allows you to see the steps behind the execution of the query. We have to underline that in using *PROFILE* we have to neglect the first value of execution time, because this is affected by the time that Neo4J spends in executing the statistical information and he graphical output given by the command, from the second execution the time execution is reliable (the most of the previous information are already computed). For MongoDB, we used the *explain()*.
The following index will be analyzed:

- Recipe Title (MongoDB and Neo4j)
- User Username (Neo4J)
- Recipe Creation Time (MongoDB)
- Comment Creation Time (MongoDB)
- On attribute *when* on relation ADDS (Neo4J)

Index on Recipe Title

Since a constraint on the recipe title was needed, and this was implemented with an index on it, then the index on the recipe title was not taken into account in these tests, because it is mandatory.

Index on User Username

The constraint on the username is also inserted through an index, so the index on the username is also mandatory.

Index on Recipe Creation Time

A study of performance has been completed, the results are available in table 3.2. The table displays that the index on the creation time field is a good index, which improve the performance of the application, by reducing the number of document and keys examined and reducing the execution time, for all the query considered. All the following (the ones in table 3.2) query has been tried as they could appear during the natural execution of the application. All of this query are sorted by the creation time (descending) but the first one, this consider the recipes added by a user in a date interval. This results are so good that the index tested has been implemented on MongoDB.

Table 3.3: Test on creationTime Index

Query	Results without Index	Results with Index
Weekly Recipes per User	executionTimeMillis : 30, totalKeysExamined : 0, totalDocsExamined : 45348	executionTimeMillis : 2, totalKeysExamined : 0, totalDocsExamined : 0
Search Recipes given a username	executionTimeMillis : 37, totalKeysExamined : 0, totalDocsExamined : 45348	executionTimeMillis : 7, totalKeysExamined : 1280, totalDocsExamined : 1280
Search for Recipe Title	executionTimeMillis : 57, totalKeysExamined : 45348, totalDocsExamined : 389	executionTimeMillis : 29, totalKeysExamined : 6126, totalDocsExamined : 6126
Search for recipe category	executionTimeMillis : 75, totalKeysExamined : 0, totalDocsExamined : 45348	executionTimeMillis : 2, totalKeysExamined : 129, totalDocsExamined : 129
Search for recipe ingredients	executionTimeMillis : 226, totalKeysExamined : 0, totalDocsExamined : 45348	executionTimeMillis : 32, totalKeysExamined : 4591, totalDocsExamined : 4591

Index on Comment Creation Time

A possible index is the one on the creation time of the comment, this index affect the query *Last Comments*. In order to see if the index must be added or not, let analyze the result of the index analysis for this index on that query (see table 3.3)

Table 3.4: Statistics about comment creation time index

Query	Results without index	Results with index
Suggested Recipes	Examined: 125843 Execution Time: 179ms	Examined: 125843 Execution Time: 113ms
Recipes of Following Users	Examined: 82836 Execution Time: 39ms	Examined: 82836 Execution Time: 38ms

From the result obtained we can see that the addition of that index do not imply a improve in the performance but instead a downgrade of performance. In fact the query with the index require more time with respect to the one without index.
For this reason we don't create an index on the creation time of the comment.
The reason for the downgrade of the performance is because MongoDB creates a multi index, so it create an index for recipe and another index for comments. Thus if the index is present it examines a number of keys that is larger than the number of recipes, because it takes into account recipes and comments.

Index on Recipe Addition Time

A possible index is the one on the attribute *when* of the relations *ADDS* in Neo4J database. In order to evaluate the performance of this index let analyze the test results in table 3.4.

Table 3.5: Statistics about index on recipe addition time

Query	Results without index	Results with index
Suggested Recipes	Examined: 125843 Execution Time: 179ms	Examined: 125843 Execution Time: 113ms
Recipes of Following Users	Examined: 82836 Execution Time: 39ms	Examined: 82836 Execution Time: 38ms

As we can see from the table only for the first query we have a good margin of benefit, but this margin is not good enough to justify the addition of this index, because even if it improves the performance on the *Suggested Recipes* query, it deteriorates the writing performance because we have to update the index for each insertion.

Chapter 4

User Manual