Large-Scale and Multi-Structured Databases

Project Design

Diet Manager

Bensi Simone, Malloggi Fabio, Nocchi Tommaso

Introduction

DietManager is an application that gives to registered standard user the opportunity to follow diets provided by nutritionists. Basically, standard users can look up for diets, foods, nutritionists, other users information using filters and view which is the most relevant diet or nutritionist accordingly to some rankings. Standard users can also follow a specific diet and check diet progress: this is made as the match between selected diet and foods that user has eaten. Moreover, DietManager allows registered nutritionists to add and remove diets. The administrators of the application are users who can do the same researches of other users as well as being able to view some statistics about users.

Link to the project on GitHub:
https://github.com/FabioMalloggi/LSMDB_Project

Design

Function and Non-Functional Requirements

This section describes the requirements that the application must provide.

Main Actor

- Standard User
- Nutritionist
- Administrator

Functional Requirements

A standard user is allowed to:

- Register
- Sign in
- Logout
- Look up for a diet
    - By name
    - By nutritionist
- Look up for a food
    - By name
- View most followed diet
- View most popular diet
- View most succeeded diet
- View recommended diets
- View k most followed diets of a given nutritionist
- View most popular nutritionist
- View most eaten food given a category
- Look up for a standard user or nutritionist
    - By username
    - By country
- Follow a diet

- Unfollow a diet
- View current diet
- Stop a diet
- Check diet progress
- Add food to eaten food list
- View their own eaten foods
- Remove their own eaten food

A nutritionist is allowed to:

- Register
- Sign in
- Logout
- View most popular nutritionist
- View most followed diet
- View most popular diet
- View most succeeded diet
- Look up for a diet
    - By name
    - By nutritionist
- Look up for a standard user or nutritionist
    - By id/username
    - By country
- Add Diet
- Remove Diet

An administrator is allowed to:

- Sign in
- Logout
- Look up for a diet
    - By name
    - By nutritionist

- Look up for a food
    - By name
- View most followed diet
- View most popular diet
- View most succeeded diet
- View recommended diets
- View most followed diets of a given nutritionist
- View most popular nutritionist
- View most eaten food given a category
- Look up for a standard user or nutritionist
    - By id/username
    - By country
- View most suggested nutrient for each nutritionists
- Look up for a standard user by username
- Remove a standard user
- Add Food
- Remove Food

Non-Functional Requirements

- The content of the application should be highly available in any moment.
- The code must be readable and easy to maintain.
- The system needs to be tolerant to data lost.
- The system needs to have fast response times, in order to make the application enjoyable for users.

Dataset

Data was collected from following sources:

- kaggle.com, from which we taken 2 datasets
- data.world, from which we taken 1 dataset

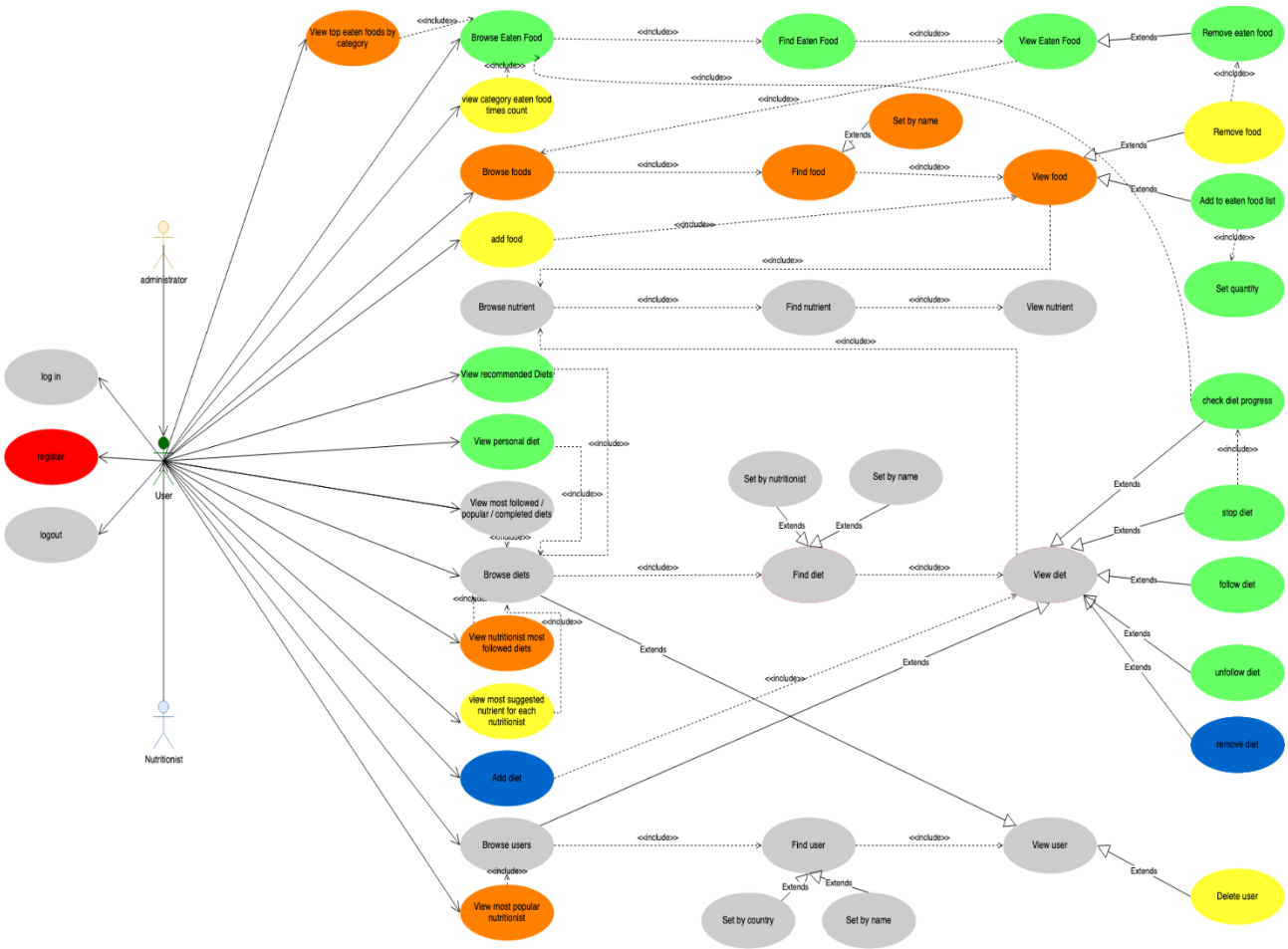
Collected data was divided into the following collections:

- Standard users and nutritionists data are taken from one of the selected datasets in Kaggle.com, where we found about 270000 entries related to generic users. We needed to filter this dataset in order to maintain only useful attributes, such as users name, country, sex, age. We populate the standard users collection with the 90% of this dataset entries and the remaining part was used for populate nutritionists collection. Other information such as username and password were generated based on extracted names.
- Foods data are taken both from data.world and from one of the selected datasets in kaggle.com, where we found altogether about 150000 entries related to generic foods. We needed to filter the overall dataset in order to maintain only useful attributes, such as food name, category and information related to nutrients contained in food, such as nutrient name, unit of measure, quantity per 100 grams of food.

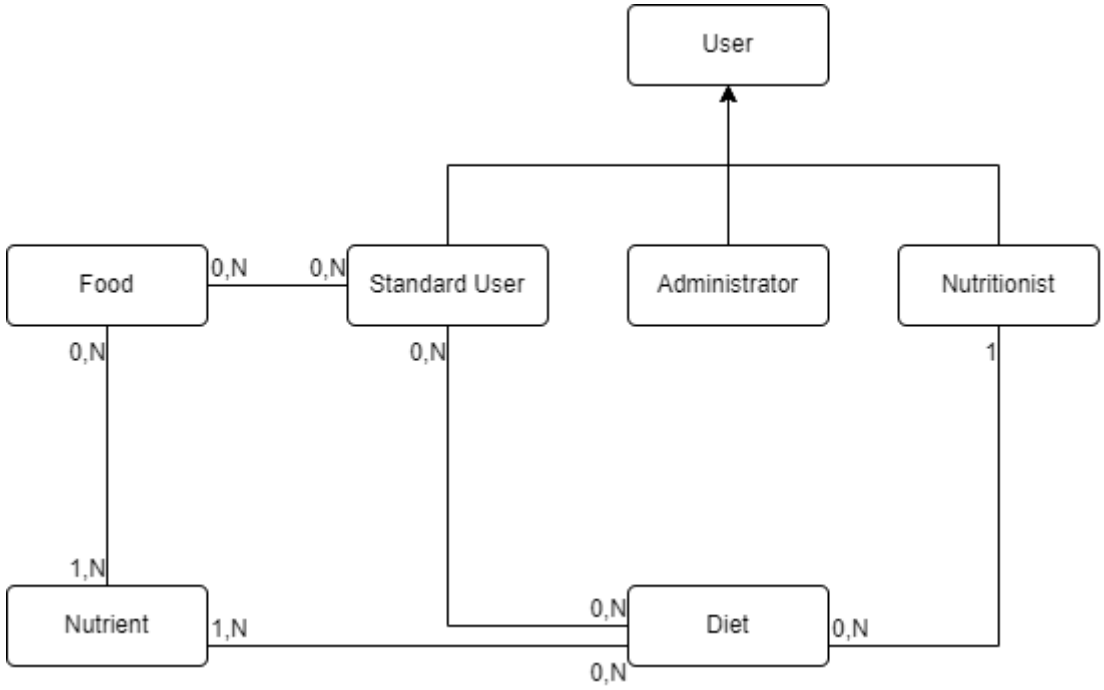
UML Use case diagram

Actions in the use case diagram can be performed by subset of users. Each action ellipse is filled with one of the following colors accordingly to which subset of users can perform that action.

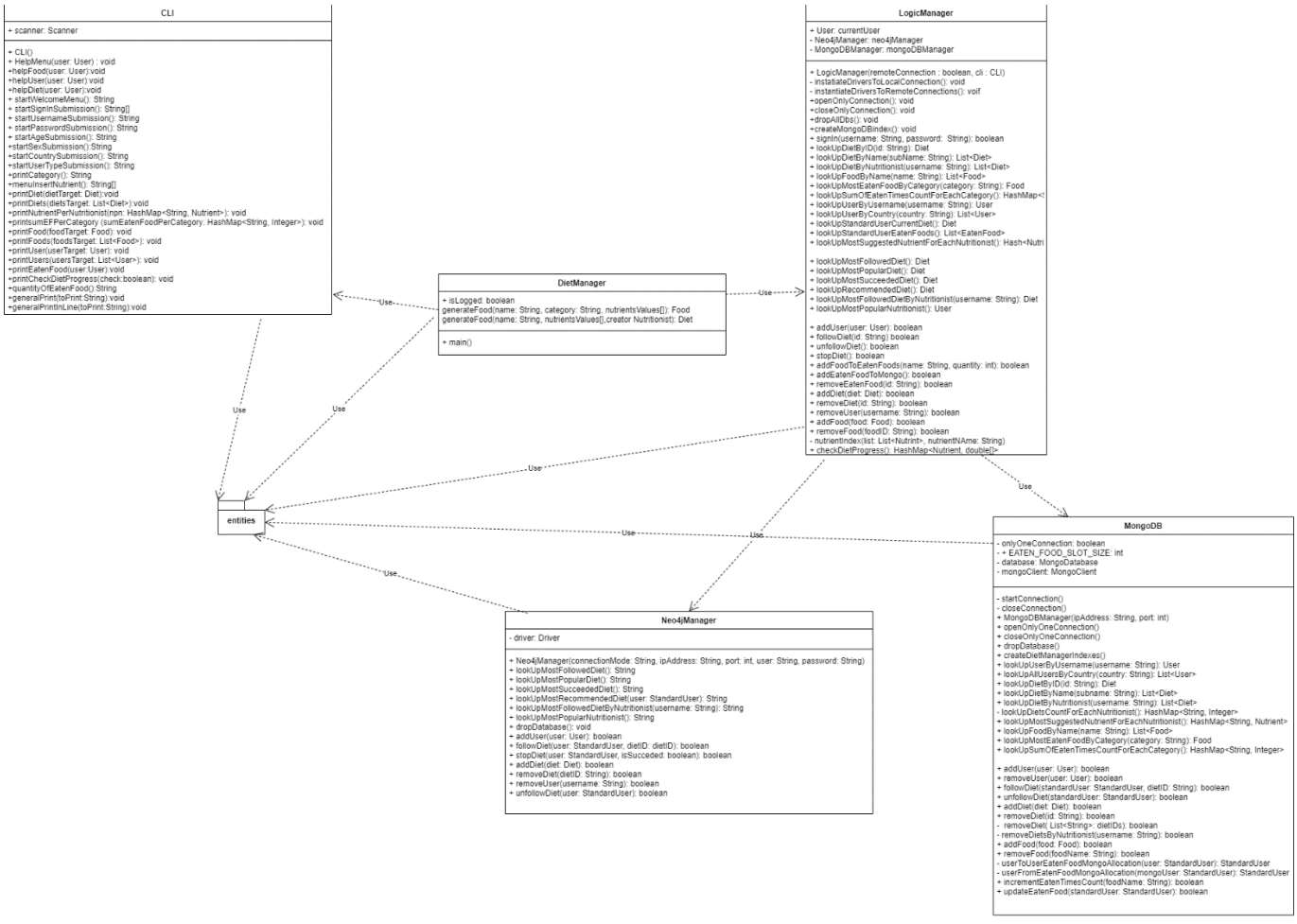
Only standard users
Only nutritionists
Only administrators
Standard users and nutritionists
Standard users and administrators
Standard users, administrators and nutritionists

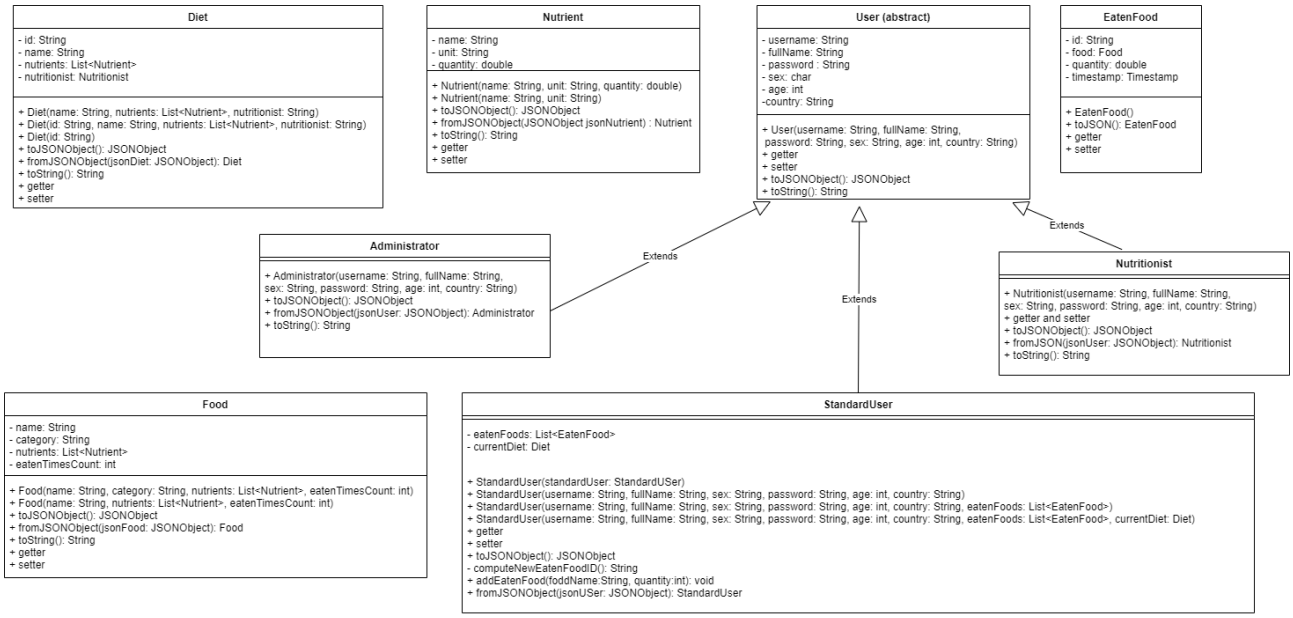


UML Class Analysis



Class Diagram





User (abstract)

Attribute Name	Type	Description
username	String	User's unique identifier requested for logging in the system
fullName	String	User's full name
password	String	User's password requested for logging in the system
country	String	User's country
sex	char	User's sex
age	int	User's age

Standard User (extends User)

Attribute Name	Type	Description
currentDiet	Diet	User's currently followed diet

Administrator (extends User)

Attribute Name	Type	Description
(same as the superclass)		

Nutritionist (extends User)

Attribute Name	Type	Description
(same as the superclass)		

Food

Attribute Name	Type	Description
name	String	Food's unique identifier
category	String	Food's category

Nutrient

Attribute Name	Type	Description
name	String	Nutrient's unique identifier
unit	String	Nutrient's unit of measure
quantity	double	Nutrient's quantity per 100 grams of related food

Diet

Attribute Name	Type	Description
id	String	Diet's unique identifier
name	String	Diet's name

EatenFood

Attribute Name	Type	Description
id	String	Eaten food's unique identifier
foodID	String	Related food's identifier
quantity	double	Eaten food's quantity per 100 grams of related food
timestamp	Timestamp	Instant in which eaten food has been added

Data Model

MongoDB – Document Organization

Collections:

- Foods Collection: about 7300 foods
- Users Collection: about 52000 users (both standard users and nutritionists)
- Diets Collection: about 2000 diets

Foods collection schema:


```
[
    {
      "_id": "PRINGLES CRISPS CHEDDAR CHEESE 5.96OZ",
      "category": "CHIPS/CRISPS/SNACK MIXES - NATURAL/EXTRUDED (SHELF STABLE)",
      "eatenTimesCount": 7,
      "nutrients":
      [
        {
          "unit": "G",
          "quantity": 4.4,
          "name": "Protein"
        },
        {
          "unit": "MG",
          "quantity": 43,
          "name": "Calcium Ca"
        },
        {
          "unit": "MG",
          "quantity": 24,
          "name": "Magnesium Mg"
        }
      ]
    },
    {
      "_id": "BANQUET BASIC CHEESY MAC AND BEEF MEAL, 7 OZ",
      "eatenTimesCount": 34,
      "nutrients":
      [
        {
          "unit": "G",
          "quantity": 4.55,
          "name": "Protein"
        },
        {
          "unit": "UG",
          "quantity": 3181.7999999999997,
          "name": "Vitamin A IU"
        }
      ]
    }
]
```


Each food document in the collection contains some food information and a list of nutrients. For each nutrient belonging to that list are stored some nutrient information for each 100 grams of the related food. Is also present a counter field that is a redundancy made in order to optimize

the query “view most eaten food given a category”, avoiding not comfortable join operations between two collections: indeed, without this field the query would require to scan both the whole users collection and the foods collection, and it would be quite heavy for the application because we estimated that the query would be often issued.

Users collection schema:

```
[
    {
        _id: "AlessandroAndrei",
        country: "Italy",
        password: "AlessandroAndrei",
        sex: "M",
        fullName: "Alessandro Andrei",
        userType: "standardUser",
        age: 25
        currentDiet: "61d71499cc42d5580397a41d"
        eatenFoods:
        [
            {
                foodName: "ENGLISH MUFFINS",
                quantity: 161,
                id: "1",
                timestamp: "2022-01-10 16:16:18.688"
            },
            {
                foodName: "DRUMSTICK LEAVES,RAW",
                quantity: 161,
                id: "1",
                timestamp: "2022-01-10 16:16:18.688"
            },
            ...
        ]
    },
    {
        _id: "IlyasAbbadi",
        country: "Algeria",
        password: "IlyasAbbadi",
        sex: "M",
        fullName: "Ilyas Abbadi",
        userType: "nutritionist",
        age: 19
    },
    {
        _id: "admin0",
        country: "Italy",
        password: "admin0",
        sex: "M",
        fullName: "Administrator0",
        userType: "administrator",
        age: 22
    }
]
```

Each user document in the collection contains some user information and a list of eaten foods. For each eaten food belonging to that list are stored the id of the food in the foods collection, the quantity in grams of the

eaten food, and the timestamp when the eaten food was added to the collection. Is also present a field which maintains the currently followed diet of the user. This is a redundancy made in order to optimize the query “view current diet”, avoiding to access to graph DB to get user current diet. It would be very heavy for the application because we estimated that the query would be very frequently issued.

Diets collection schema:


```
[
    {
        "_id": {"$oid": "61d71499cc42d5580397a41d"},
        "nutritionist": "LeeSang-Eun",
        "name": "Fat Burning Diet",
        "nutrients":
        [
            {
              "unit": "KCAL",
              "quantity": 239.16279376206984,
              "name": "Energy"
            },
            {
              "unit": "G",
              "quantity": 565.0446922119274,
              "name": "Protein"
            },
            ...
        ]
    }
    {
        "_id": {"$oid": "61d7180519262d40d14846c4"},
        "nutritionist": "LeeSang-Eun",
        "name": "Fat Burning Diet",
        "nutrients":
        [
            {
              "unit": "KCAL",
              "quantity": 239.16279376206984,
              "name": "Energy"
            },
            {
              "unit": "G",
              "quantity": 565.0446922119274,
              "name": "Protein"
            },
            ...
        ]
    }
]
```


Each diet document in the collection contains some diet information and a list of nutrients. For each nutrient belonging to that list are stored some nutrient information for each 100 grams of the related food. Is also

present a field which maintains the nutritionist who provided the diet. This is a redundancy made in order to optimize the query “view most suggested nutrient for each nutritionist”, avoiding not comfortable join operations between two collections: indeed, without this field the query would require to scan both the whole users collection and the diets collection, and it would be quite heavy for the application because that query is computationally expensive.

Neo4j – Nodes Organization

Nodes:

- User: about 50000 standard users
- Nutritionist: about 1300 nutritionists
- Diet: about 2000 diets

Relationships:

- Nutritionist → PROVIDES → Diet: this relationship is created when a Nutritionist creates a Diet, and it’s removed when a Nutritionist removes a Diet.
- User → FOLLOWS → Diet: this relationship is created when a User follows for the first time a Diet, and it’s removed when a User unfollows a Diet. The relationship has an attribute “status”: it’s set to “current” when a User follows a Diet; it’s set to “completed” when a User stops a Diet and he had successfully completed it; it’s set to “failed” when a User stops a Diet and he had not successfully completed it.

Implementation

The application is composed by three java modules in order to guarantee high readability. Packages are organized in modules by layer as follows:

- dietmanager/client
- dietmanager/services
    - dietmanager/services/entities
    - dietmanager/services/datageneration
- dietmanager/persistence

Java Entities

- User
- StandardUser
- Nutritionist
- Administrator
- Nutrient
- Food
- EatenFood
- Diet

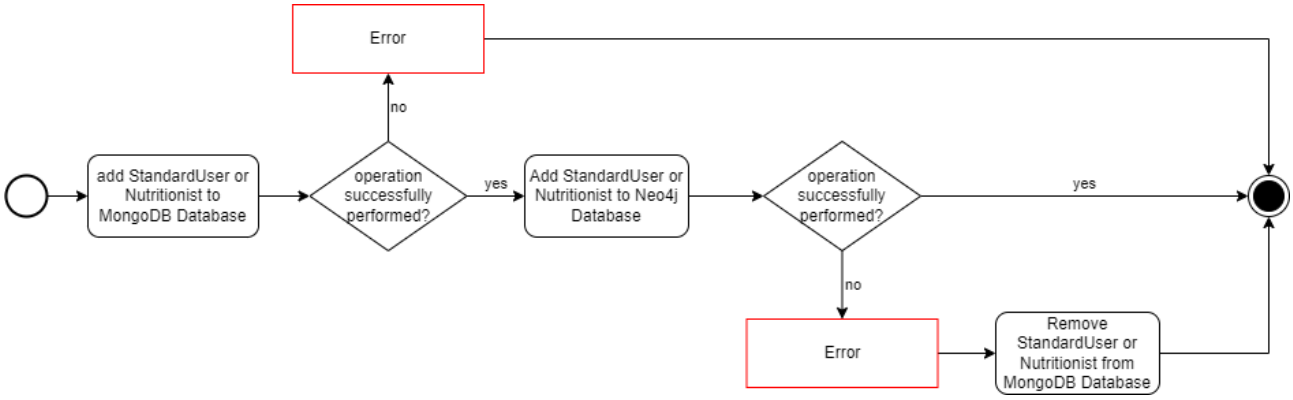
```
public abstract class  User {

    private String username;
    private String fullName;
    private String password;
    private String sex;
    private int age;
    private String country;

}


public class StandardUser extends User {

    private List<EatenFood> eatenFoods;
```

```
    private Diet currentDiet;
}
public class Administrator extends User {}
public class Nutritionist extends User {}
public class Nutrient{
    private String name;
    private String unit;
    private double quantity;
}
public class Food{
    private String name;
    private String category;
    private List<Nutrient> nutrients;
    private int eatenTimesCount;
}
public class EatenFood {
    private String id;
    private Food food;
    private int quantity;
    private Timestamp timestamp;
}
public class Diet {
    private String id;
    private String name;
    private List<Nutrient> nutrients;
    private String nutritionist;
}
```

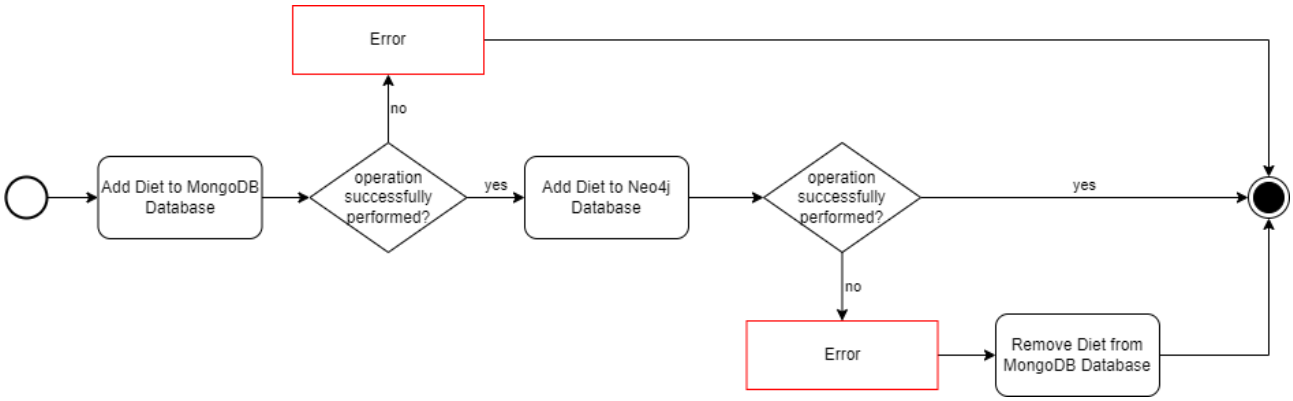
Database Consistency Management

Using two different database architectures for data management, we needed to manage possible states of inconsistency between entities in Neo4j and documents stored on MongoDB. Situations in which this problem occurs are listed in the following diagrams, that also explain how we have implemented the error recovery.

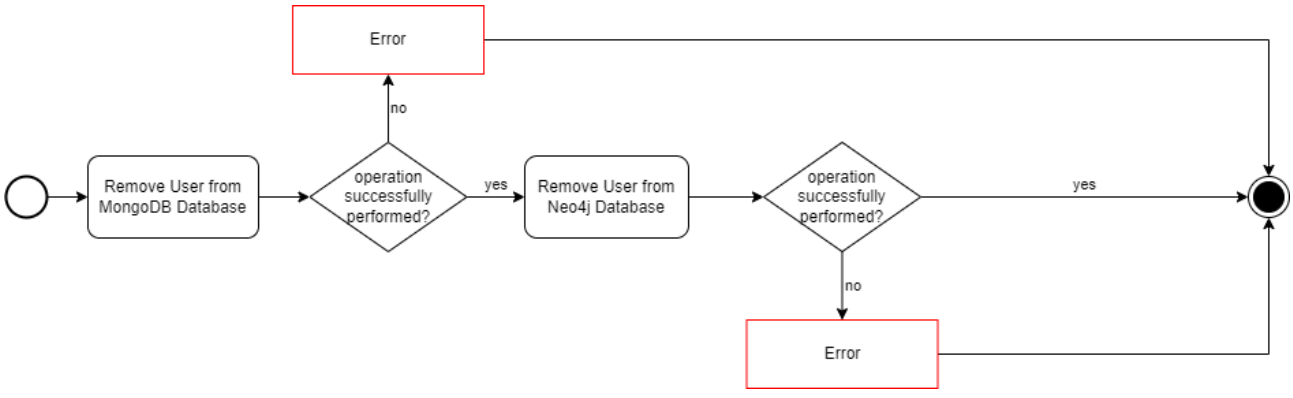
Register standard user or nutritionist



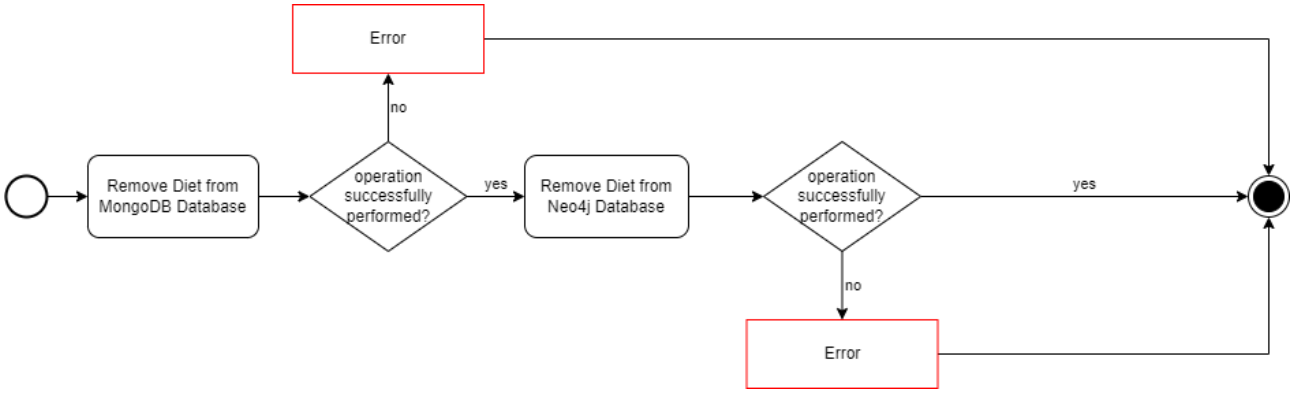
Add diet



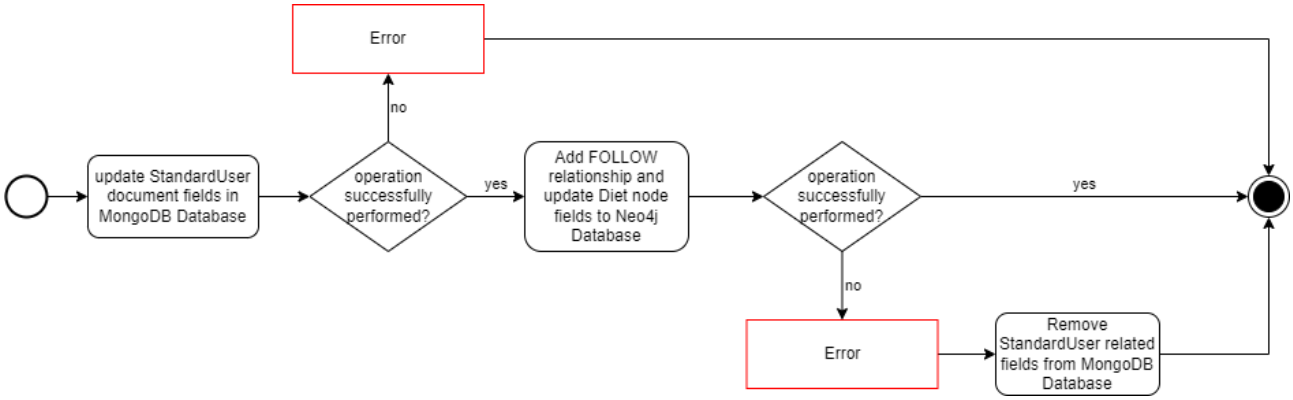
Remove user



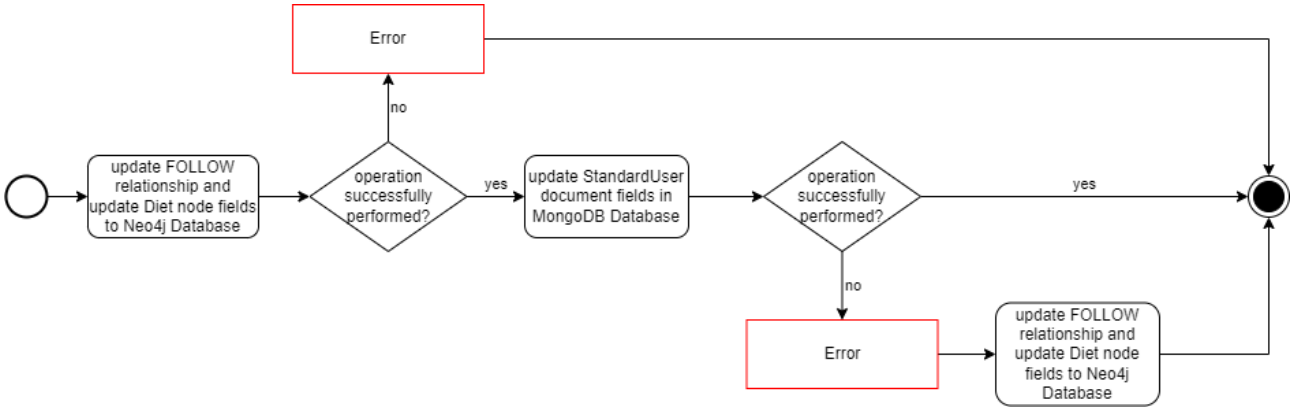
Remove diet



Follow diet



Stop diet



These diagrams highlight that writings are made first on one database, then, if there aren't any errors, them are made on the other. In case of error an error message will be displayed. In case of error on the second writing the previous operation done on the other database will be rollback.

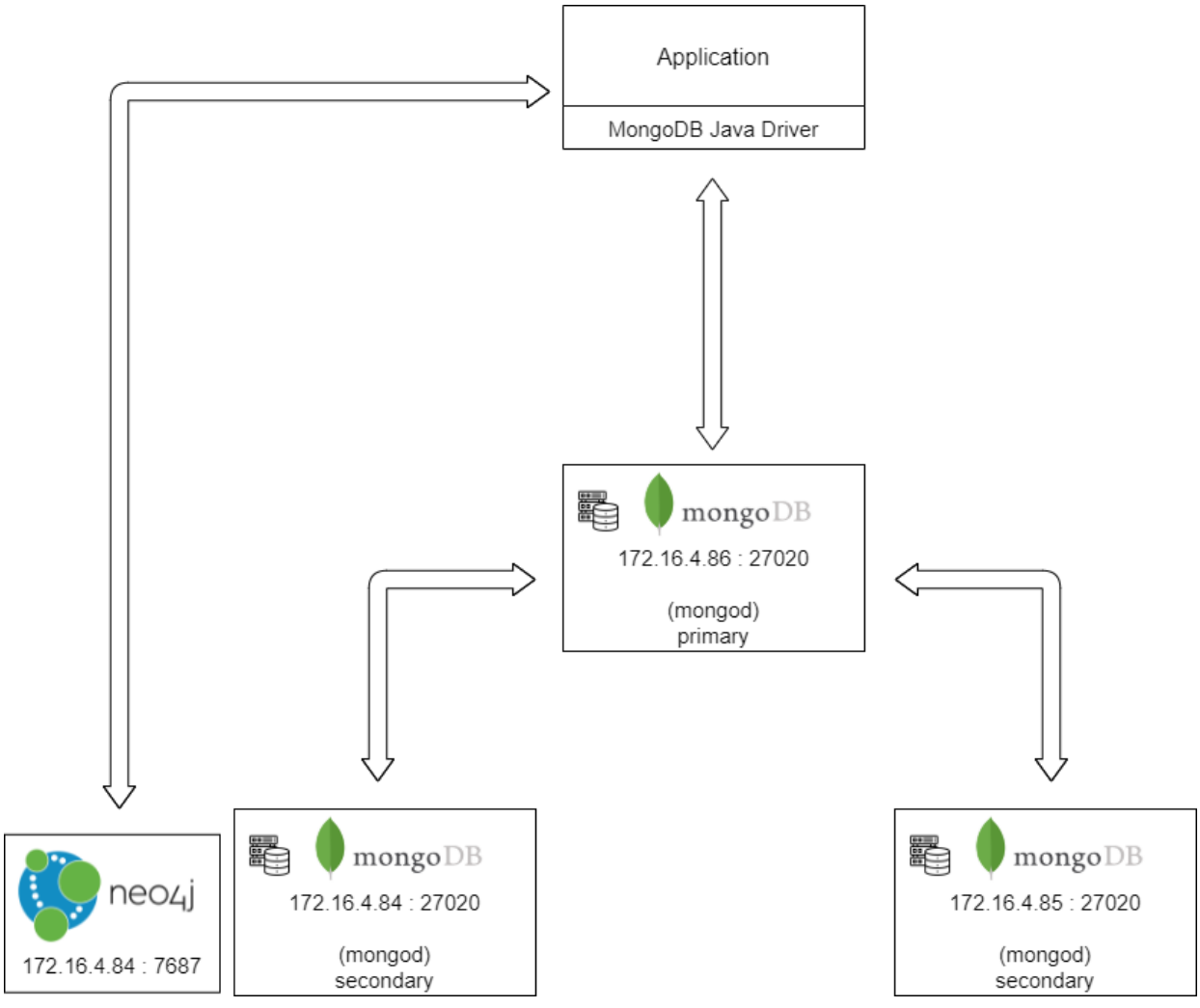
MongoDB – Replica Set

In order to guarantee high service availability we need to set up a cluster.

Three mongod instances are hosted on three different virtual machines to create a replica set on the shared cluster, instead neo4j instance is hosted on a single VM. Servers are able to communicate with others in order to ensure eventual consistency among replicas.

The replica features are listed below:

VM	IP address	Port	OS
Replica-0	172.16.4.84	27020	Ubuntu
Replica-1	172.16.4.85	27020	Ubuntu
Replica-2	172.16.4.86	27020	Ubuntu



Replica configuration

The configuration is shown below:

```
rsconfig = {
_id: "lsmdb",
members: [
{_id: 0, host: "172.16.4.84:27020", priority:1},
{_id: 1, host: "172.16.4.85:27020", priority:2},
{_id: 2, host: "172.16.4.86:27020", priority:5}] };
```

The replica on the VM that has address 172.16.4.86 will be elected as the primary because it has the highest priority. If the primary falls, the VM that has address 172.16.4.85 will be elected as the new primary because it has higher priority than the VM that has address 172.16.4.86.

The application is read oriented: readings are generally more frequent and more expensive than writings, as you will understand from the query analysis performed in the next chapter. However, the application has to manage several writings, due to the fact that one of the users main activity is to add foods to personal eaten food list after have read them.

Moreover, the system has high availability and time response requirements. For these reasons it is tuned as follow:

- Write concern: 1.
    - The system retrieve the control to the user immediately after the first write operation on the server which is communicating with the client. Therefore, it allows the system to achieve the minimum response time during writes operation at the cost of introducing some kind of eventual consistency paradigm.
- Read preferences: nearest.
    - Read operations are performed on the node with the lowest network latency to have the fastest response.

To ensure availability and fast response time we decided to adopt eventual consistency because it's not fundamental to have all data recently updated.

Some of the main elements that lead to those choices regards the application purpose itself: the activity of following a diet required at least a period of some weeks, while the insertion of eaten foods is performed at least 3 times a day. Furthermore, all the activities of the users concern either readings over diets and foods, that once inserted cannot be changed, or writing operations that concerns the users himself, like inserting an eaten food, without any importance for other users and that can even be inserted again by the users if he find out it was lost.

To set those options, the following connection methods are used in MongoDB Java Driver:

```
String uri = "mongodb://" + ipAddress + ":" + port;
connectionUri = new ConnectionString(uri);
MongoClientSettings mcs = MongoClientSettings.builder()
        .applyConnectionString(connectionUri)
        .readPreference(ReadPreference.nearest())
        .retryWrites(true)
        .writeConcern(WriteConcern.ACKNOWLEDGED)
        .build();
mongoClient = MongoClients.create(mcs);
```

Analysis

Query Analysis - Mongo DB

Read operations		
Operation	Expected Frequency	Cost
Sign in	High	Low (1 read)
Look up for a diet by id	Average	Low (1 read)
Look up for a diet by name	Average	Average (multiple reads)
Look up for a diet by nutritionist	Average	Low (few reads)
Look up for a food by name	Very High	Low (1 read)
Look up most eaten food given a category	Average	Low(1 read)
Look up for a user by username	Low	Low (1 read)
Look up for a nutritionist by username	Average	Low (1 read)
Look up for a user by country	Low	Low (few reads)
Look up for a nutritionist by country	Average	Low (few reads)
Look up for standardUser personal current diet	High	Low (2 reads)
Look up for StandardUser personal eaten foods	Average	Low (1 read)
Check diet progress	Average	Low (2 reads)
View most suggested nutrient for each nutritionists	Low	Very High (Complex Aggregations)

Write operations		
Operation	Expected Frequency	Cost
Register a new User (Nutritionist, StandardUser)	Low	Low (create new document)
Follow a Diet - Only with redundancy	Average	Low (read to check if a diet is already currently followed and then update document fields)
Stop a Diet - Only with redundancy	Average	Low (update document)
Add Food to EatenFoods by name	Very High	Low (add embedded document to tail in an array field)
Remove EatenFood by id from EatenFoods	Low	Average (remove embedded document)
Add a Diet	Average	Low (create new document)
Remove a Diet by id	Low	Low (remove document)
Remove User by username	Low	Low (remove document)
Add a Food	Low	Low (create new document)
Remove Food by id	Low	Low (remove document)

Crud operations – MongoDB

Create

- User: the following method is called every time a user registers himself into the application.

```
public boolean addUser(User user){
    openConnection();
    MongoCollection<Document> usersCollection = database.getCollection(COLLECTION_USERS);
    InsertOneResult insertOneResult = usersCollection.insertOne(userToDocument(user));
    closeConnection();
    return insertOneResult.wasAcknowledged();
}
```

- Diet: the following method is called every time a diet is added by a nutritionist.

```
public boolean addDiet(Diet diet){
    if(diet.getId() != null)
        return false;
    openConnection();
    MongoCollection<Document> dietCollection = database.getCollection(COLLECTION_DIETS);
    Document dietDocument = dietToDocument(diet);
    InsertOneResult insertOneResult = dietCollection.insertOne(dietDocument);
    // adding '_id' to diet object passed as argument
    diet.setId(dietDocument.getObjectId("_id").toString());
    closeConnection();
    return insertOneResult.wasAcknowledged();
}
```

- Food: the following method is called every time a food is added by an administrator.

```
public boolean addFood(Food food){
    openConnection();
    MongoCollection<Document> foodCollection = database.getCollection(COLLECTION_FOODS);
    InsertOneResult insertOneResult = foodCollection.insertOne(foodToDocument(food));
    closeConnection();
    return insertOneResult.wasAcknowledged();
}
```

Read

- User: the following method is called when users want to look up for another user.

```
public User lookUpUserByUsername(String username){
    openConnection();
    MongoCollection<Document> usersCollection = database.getCollection(COLLECTION_USERS);
    Document userDocument = usersCollection.find(eq(User.USERNAME, username)).first();
    closeConnection();
    return userFromDocument(userDocument);
}
```

- Diet: the following method is called when users want to look up for a diet.

```
public Diet lookUpDietByID(String id){
    openConnection();
    Document dietDocument = null;
    try {
        MongoCollection<Document> dietCollection = database.getCollection(COLLECTION_DIETS);
        dietDocument = dietCollection.find(eq(Diet.ID, new ObjectId(id))).first(); // there can be at least only 1 match.
    }catch(Exception e){}
    closeConnection();
    return dietFromDocument(dietDocument);
}
```

- Food: the following method is called when a standard user or an administrator want to look up for a food.

```
public List<Food> lookUpFoodsByName(String subname){
    openConnection();
    String regex = "^.*"+subname+".*$";

    List<Food> foods = new ArrayList<>();
    MongoCollection<Document> foodCollection = database.getCollection(COLLECTION_FOODS);
    try(MongoCursor<Document> cursor = foodCollection.find(
            eq(Food.NAME, subname)).iterator()){
        while(cursor.hasNext()){
            foods.add(foodFromDocument(cursor.next()));
        }
    }
    closeConnection();
    return foods;
}
```

Update

- The following method is called when a standard user wants to follow a diet.

```
public boolean followDiet(StandardUser standardUser, String dietID){
    MongoCollection<Document> userCollection = database.getCollection(COLLECTION_USERS);
    Bson userFilter = Filters.eq( User.USERNAME, standardUser.getUsername() );
    Bson insertCurrentDietField = Updates.set(StandardUser.CURRENT_DIET, dietID);
    UpdateResult updateResult = userCollection.updateOne(userFilter, insertCurrentDietField);
    return updateResult.wasAcknowledged();
}
```

- The following method is called when a standard user wants to stop a diet.

```
// used for both 'unfollowDiet' and 'stopDiet'
public boolean unfollowDiet(StandardUser standardUser){
    MongoCollection<Document> userCollection = database.getCollection(COLLECTION_USERS);

    Bson userFilter = Filters.eq( User.USERNAME, standardUser.getUsername() );
    Bson deleteCurrentDietField = Updates.unset(StandardUser.CURRENT_DIET);
    Bson deleteEatenFoodsField = Updates.unset(StandardUser.EATENFOODS);

    UpdateOneModel<Document> updateRemoveCurrentDietDocument = new UpdateOneModel<>(
            userFilter, deleteCurrentDietField);
    UpdateOneModel<Document> updateRemoveEatenFoodsDocument = new UpdateOneModel<>(
            userFilter, deleteEatenFoodsField);

    List<WriteModel<Document>> bulkOperations = new ArrayList<>();
    bulkOperations.addAll(Arrays.asList(updateRemoveCurrentDietDocument,updateRemoveEatenFoodsDocument));
    BulkWriteResult bulkWriteResult = userCollection.bulkWrite(bulkOperations);
    return bulkWriteResult.wasAcknowledged();
}
```

Delete

- User: the following method is called when an administrator wants to remove an user.

```
public boolean removeUser(User user){
    openConnection();
    boolean isSuccessful = true;
    if(user != null) {
        // if the user to be removed is a Nutritionist, all his diets must be deleted altogether.
        if(user instanceof Nutritionist){
            isSuccessful = removeDietsByNutritionist(user.getUsername());
        }

        // deleting user given his username
        if(isSuccessful) {
            MongoCollection<Document> usersCollection = database.getCollection(COLLECTION_USERS);
            DeleteResult deleteResult = usersCollection.deleteOne(eq(User.USERNAME, user.getUsername()));
            isSuccessful = deleteResult.wasAcknowledged();
        }
    } else {
        isSuccessful = false;
    }
    closeConnection();
    return isSuccessful;
}
```

- Diet: the following method is called when a nutritionist want to remove a diet.

```
public boolean removeDiet(String dietID) { return removeDiet(Arrays.asList(dietID)); }

public boolean removeDiet(List<String> dietIDs){
    openConnection();
    // deleting 'diet' document from 'diets' collection
    boolean isSuccessful;

    List<ObjectId> dietObjectIds = new ArrayList<>();
    for (String dietId: dietIDs){
        dietObjectIds.add(new ObjectId(dietId));
    }

    // deleting 'currentDiet' field and consequently 'EatenFoods' field from each 'user' document whose current diet value
    // correspond to one of the diets to be removed
    MongoCollection<Document> userCollection = database.getCollection(COLLECTION_USERS);

    Bson userFilter = Filters.in( StandardUser.CURRENT_DIET, dietIDs );
    Bson deleteCurrentDietField = Updates.unset(StandardUser.CURRENT_DIET);
    Bson deleteEatenFoodsField = Updates.unset(StandardUser.EATENFOODS);

    UpdateManyModel<Document> updateRemoveCurrentDietDocument = new UpdateManyModel<>(
            userFilter, deleteCurrentDietField);
    UpdateManyModel<Document> updateRemoveEatenFoodsDocument = new UpdateManyModel<>(
            userFilter, deleteEatenFoodsField);

    List<WriteModel<Document>> bulkOperations = new ArrayList<>();
    bulkOperations.addAll(Arrays.asList(updateRemoveCurrentDietDocument,updateRemoveEatenFoodsDocument));
    BulkWriteResult bulkWriteResult = userCollection.bulkWrite(bulkOperations);
    isSuccessful = bulkWriteResult.wasAcknowledged();

    // if the previous operation completed successfully, all diets are removed from 'diets' collection
    if(isSuccessful){
        MongoCollection<Document> dietCollection = database.getCollection(COLLECTION_DIETS);
        DeleteResult deleteDietsResult = dietCollection.deleteMany(in(Diet.ID, dietObjectIds));
        isSuccessful = deleteDietsResult.wasAcknowledged();
    }
    closeConnection();
    return isSuccessful;
}
```

- Food: the following method is called when an administrator wants to remove a food.

```
public boolean removeFood(String foodName){
    openConnection();
    boolean isSuccessful = false;

    EatenFood eatenFood = new EatenFood();

    // Replace all instances of EatenFoods referring to the food to be removed with empty slot of eatenfoods
    String elementIdentifier = "elem";
    MongoCollection<Document> userCollection = database.getCollection(COLLECTION_USERS);

    UpdateResult updateResult = userCollection.updateMany(
            new Document(StandardUser.EATENFOODS+"."+EatenFood.FOOD_NAME, foodName),
            new Document("$set",
                    new Document(StandardUser.EATENFOODS+".$["+elementIdentifier+"]",
                            new Document(EatenFood.ID, eatenFood.getId())
                                    .append(EatenFood.FOOD_NAME, eatenFood.getFoodName())
                                    .append(EatenFood.QUANTITY, eatenFood.getQuantity())
                                    .append(EatenFood.TIMESTAMP, eatenFood.getTimestamp().toString())
                    )
            ),
            new UpdateOptions().arrayFilters(
                    Arrays.asList(new Document(elementIdentifier+"."+EatenFood.FOOD_NAME, foodName))
            )
    );
    closeConnection();

    // Consistency issue: only if the precedent operation completed successfully then we can delete the food
    if(updateResult.wasAcknowledged()){
        openConnection();
        MongoCollection<Document> foodCollection = database.getCollection(COLLECTION_FOODS);
        DeleteResult deleteResult = foodCollection.deleteOne(eq(Food.NAME, foodName));
        isSuccessful = deleteResult.wasAcknowledged();
        closeConnection();
    }
    return isSuccessful;
}
```

Analytics implementation – MongoDB

The analytics involving the MongoDB database are represented below

Nutritionist Analytic:

This analytics is based on Diets collection and needs to know the total number of Diets for each Nutritionist. The output is composed by a number of lines equals to the number of Nutritionists that generated at least a diet, and each line is composed by the username of the Nutritionist and the total number of diets

```
[{$group: {
 _id: '$nutritionist',
 numDiets: {
  $sum: 1
 }
}}]
```

This aggregation is exploited in conjunction with a second and bigger aggregation, for counting the most used Nutrient for each Nutritionist.

This first aggregation is necessary to consider the real total number of diets for each nutritionist when we compute the average values of each nutrient. Otherwise the average would be computed considering only the number of diets in which that nutrient is used by nutritionist target.

In the following the Mongo Java Driver code is shown:

```
private HashMap<String, Integer> lookUpDietsCountForEachNutritionist(){
    openConnection();
    MongoCollection<Document> dietCollection = database.getCollection(COLLECTION_DIETS);
    HashMap<String, Integer> nutritionistDietsCountMap = new HashMap<>();

    Document currentDocument;
    try(MongoCursor<Document> cursor = dietCollection.aggregate(Arrays.asList(
            new Document("$group",
                new Document("_id", "$"+Diet.NUTRITIONIST)
                        .append("numDiets", new Document("$sum", 1L)))) ).iterator()){

        while(cursor.hasNext()){
            currentDocument = cursor.next();
            nutritionistDietsCountMap.put(currentDocument.getString(Nutritionist.USERNAME), (int)(long)currentDocument.getLong("numDiets"));
        }
    }
    closeConnection();
    return nutritionistDietsCountMap;
}
```

Nutrient For Each Nutritionist Analytic

This analytic is based on the Diets collection and needs to know the most used nutrient for each Nutritionist. The output is composed by a line with the total quantity for each Nutrient and each Nutritionist

First step: For each diet, generating a number of documents equals to the number of nutrient present in the diet target

```
[
{$unwind: {
 path: '$nutrients',
 preserveNullAndEmptyArrays: false
}},
```

Second step: matching the document where the nutrient is not 'Energy'

```
{$match: {
 'nutrients.name': {
  $ne: 'Energy'
 }
}},
```

Third Step: group the result of the previous steps by nutritionist, name of the nutrient, and the corresponding unit of the nutrient and finally calculate the sum of the nutrient quantity

```
{$group: {
 _id: {
  nutritionist: '$nutritionist',
  nutrient: '$nutrients.name',
  unit: '$nutrients.unit'
 },
 totalQuantity: {
  $sum: '$nutrients.quantity'
 }
}},
```

Fourth step: This is the formatting step; we order the final result to display it correctly. Moreover, to display each total quantity with the same ‘unit’, namely Grams, we test if the previous unit is equal to UG and in this case we divided for 1000000 the previous total quantity, else if the previous unit is equal to MG we divided for 1000 the previous total quantity, otherwise we preserve the real value.

```
{$project: {
 _id: 0,
 nutritionist: '$_id.nutritionist',
 nutrient: '$_id.nutrient',
 totalQuantity: {
    $cond: { 'if': { $eq: ['$_id.unit', 'UG' ]}, then: { $divide: ['$totalQuantity',1000000] },
        'else': { $cond: { 'if': { $eq: [ '$_id.unit', 'MG' ]}, then: { $divide: [ '$totalQuantity',1000]}, 'else': '$totalQuantity'} } }
 }
}},
```

In the following the Mongo Java Driver code is shown:

```
    Bson unwindDocument = new Document("$unwind",
            new Document("path", "$"+Diet.NUTRIENTS)
                    .append("preserveNullAndEmptyArrays", false));
    Bson matchDocument = new Document("$match",
            new Document(Diet.NUTRIENTS+"."+Nutrient.NAME,
                    new Document("$ne", "Energy")));
    Bson group1Document = new Document("$group",
            new Document("_id",
                    new Document(Diet.NUTRITIONIST, "$"+Diet.NUTRITIONIST)
                            .append("nutrient", "$"+Diet.NUTRIENTS+"."+Nutrient.NAME)
                            .append(Nutrient.UNIT, "$"+Diet.NUTRIENTS+"."+Nutrient.UNIT))
                    .append("totalQuantity",
                            new Document("$sum", "$"+Diet.NUTRIENTS+"."+Nutrient.QUANTITY)));
    Bson project1Document = new Document("$project",
            new Document("_id", 0L)
                    .append(Diet.NUTRITIONIST, "$_id."+Diet.NUTRITIONIST)
                    .append("nutrient", "$_id.nutrient")
                    .append("totalQuantity",
                            new Document("$cond",
                                new Document("if",
                                        new Document("$eq", Arrays.asList("$_id."+Nutrient.UNIT, "UG")))
                                        .append("then",
                                                new Document("$divide", Arrays.asList("$totalQuantity", 1000000L)))
                                        .append("else",
                                                new Document("$cond",
                                                        new Document("if",
                                                                new Document("$eq", Arrays.asList("$_id"+Nutrient.UNIT, "MG")))
                                                                .append("then",
                                                                        new Document("$divide", Arrays.asList("$totalQuantity", 1000L)))
                                                                .append("else", "$totalQuantity"))))));
    try(MongoCursor<Document> cursor = dietCollection.aggregate(Arrays.asList(
            unwindDocument,
            matchDocument,
            group1Document,
            project1Document )).iterator()){

        Document currentDocument;
        while(cursor.hasNext()){
            currentDocument = cursor.next();

            nutritionists.add(
                    currentDocument.getString(Diet.NUTRITIONIST));
            nutrients.add(new Nutrient(
                    currentDocument.getString("nutrient"),
                    "G",
                    currentDocument.getDouble("totalQuantity")
            ));
```

Count how many foods have been eaten For Each Category Analytic

This analytic is based on the foods collection and query the database to compute the sum of times all foods belonging to each category have been eaten. The output is composed by a list of documents which one containing for a category of foods the corresponding count.

First Step: group by category name computing the sum of all the 'eatenTimesCount' fields of the corresponding foods.

```
[{$group: {
 _id: '$category',
 sumOfEatenTimesCount: {
  $sum: '$eatenTimesCount'
 }
}}]
```

In the following the Mongo Java Driver code is shown:

```
public HashMap<String, Integer> lookUpSumOfEatenTimesCountForEachCategory(){
    if( ! onlyOneConnection) openConnection();
    HashMap<String, Integer> categoryEatenTimesCountMap = new HashMap<>();
    MongoCollection<Document> foodCollection = database.getCollection(COLLECTION_FOODS);

    Bson groupDocument = new Document("$group",
            new Document("_id", "$"+Food.CATEGORY)
                    .append("SumOfEatenFoodsCount",
                            new Document("$sum", "$"+ Food.EATEN_TIMES_COUNT)));

    try(MongoCursor<Document> cursor = foodCollection.aggregate(Arrays.asList(
            groupDocument )).iterator()){

        Document currentDocument;
        while(cursor.hasNext()){
            currentDocument = cursor.next();
            categoryEatenTimesCountMap.put(
                    currentDocument.getString(Food.NAME),
                    currentDocument.getInteger( key: "SumOfEatenFoodsCount")
            );
        }
    }
    if( ! onlyOneConnection) closeConnection();
    return categoryEatenTimesCountMap;
}
```

Index Analysis – MongoDB

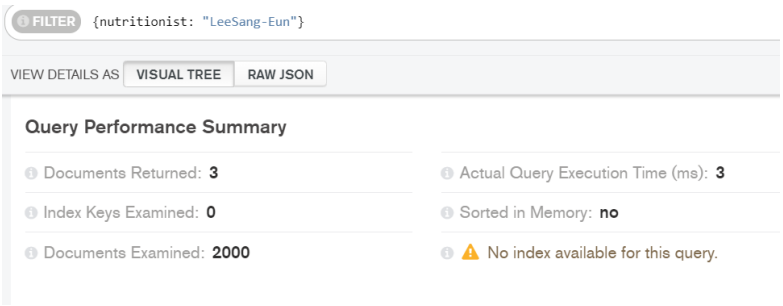
In order to speed up the application, we decided to use the following indexes:

Index name	Index type	Collection	Attributes
Query “retrieve diets by nutritionist”	Simple	Diet	nutritionist
Query “look up user/nutritionist by country”	Simple	User	country
Query “look up most eaten food given a category”	Compound	Food	category, eatenTimesCount

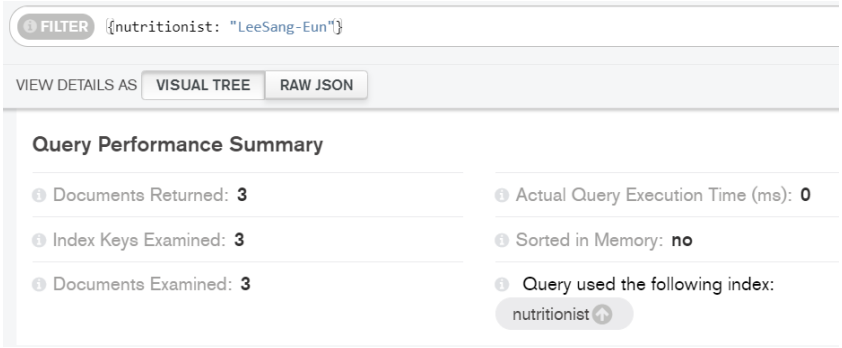
1. Query “retrieve diets by nutritionist”

```
db.diet.createIndex(
        {nutritionist: 1} ,
        {name: “query for retrive diets by nutritionist” }
)
```

Without index:



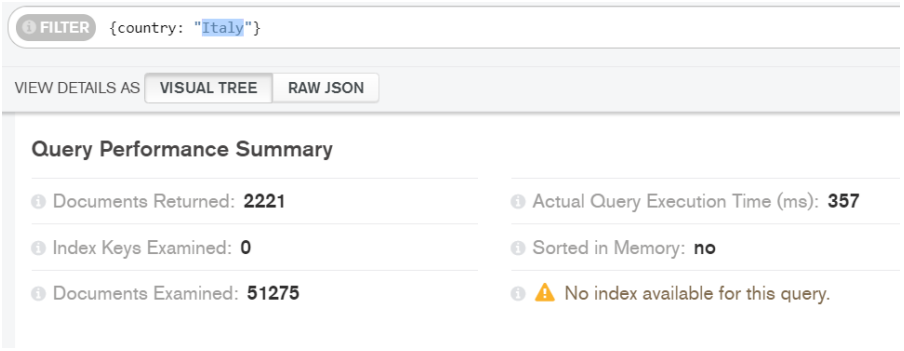
With index:



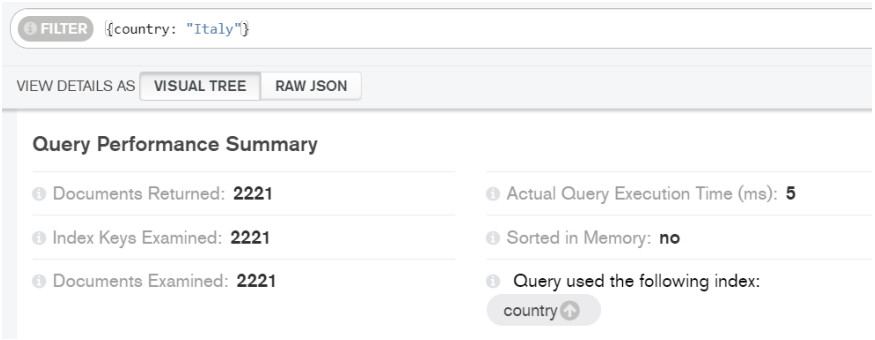
2. Query “look up user/nutritionist by country”

```
db.user.createIndex(
{country: 1}
)
```

Without index:



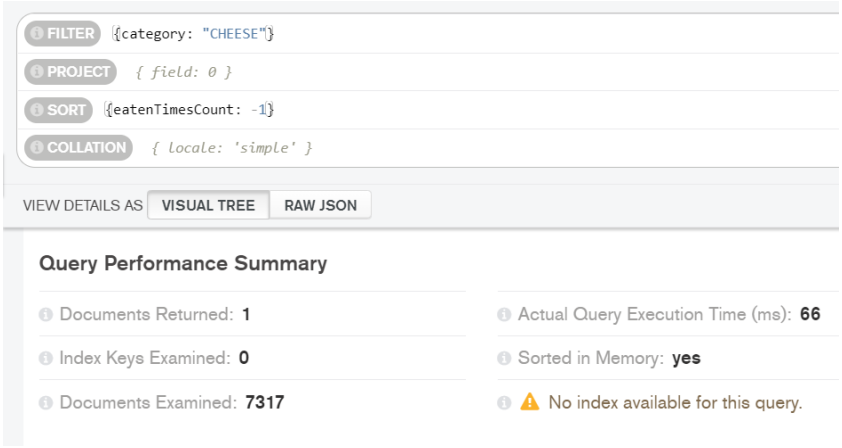
With index:



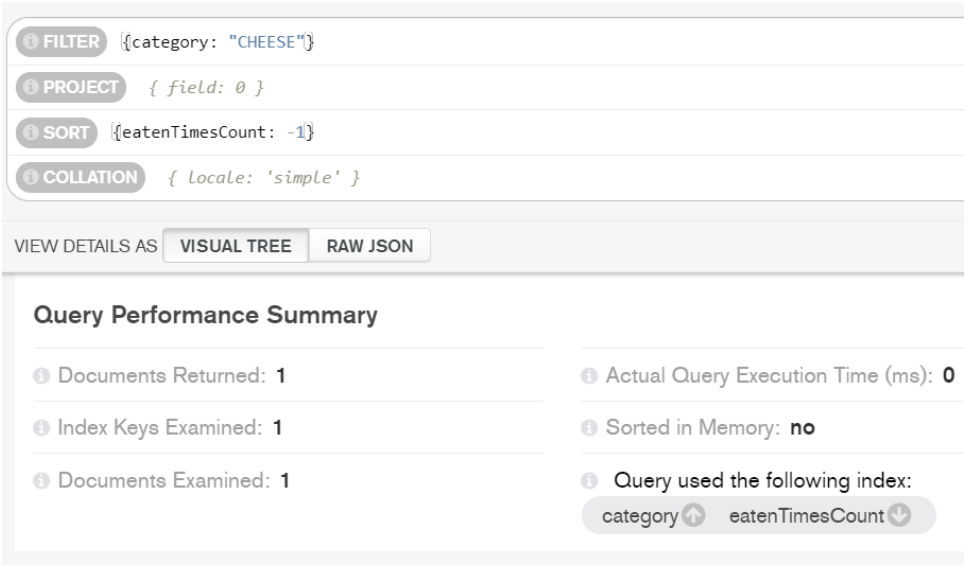
3. Query “look up most eaten food given a category”

```
db.food.createIndex(
        {category:1, eatenTimesCount: -1}
        {name: “query for retrive the most eatenFood”}
)
```

Without index:



With index:



Query Analysis - Neo4j DB

Read operations		
Operation	Expected Frequency	Cost
Look Up for the most followed diet	Average	Average (multiple reads)
Look Up for the most popular diet	Average	Average (multiple reads)
Look Up for the most completed diet	Average	Average (multiple reads)
Look Up for the recommended diet	Average	Very High (complex aggregation)
Look Up for the most followed diet of a given nutritionist	Average	Low (few reads)
Look Up for the most popular nutritionist	Average	Average (multiple reads)

Write operations		
Operation	Expected Frequency	Cost
Register StandardUser or Nutritionist	Low	Low (create a new node)
Follow a Diet	Average	Average (create a new relationship and update node field followersCount)
Stop a Diet	Average	Average (update a relationship and update node fields followersCount, succeededCount, failedCount)
Add Diet	Average	Average (create a new node and create relationship)

Remove Diet	Low	High (delete node and delete multiple relationships)
Remove User	Low	High (delete node and delete multiple relationships)

Queries Implementations - Neo4j

A selection of Queries regarding the Neo4j database is showed below:

Create “Follow” relationship between Standard User and Diet

```
"MATCH (user:User), (diet:Diet) WHERE user.username = $username " +
"AND diet.id = $id CREATE (user)-[:FOLLOWS { status: \"current\"}]->(diet) " +
"SET diet.followersCount = diet.followersCount + 1",
```

Update “Follow” relationship between Standard User and Diet

```
"MATCH (user:User)-[follows:FOLLOWS]->(diet:Diet) WHERE user.username = $username " +
        "AND diet.id = $id SET follows.status = $status, diet.followersCount = diet.followersCount - 1, " +
        "diet.failedCount = diet.failedCount + $failedCountIncrement, " +
        "diet.succeededCount = diet.succeededCount + $succeededCountIncrement",
```

Delete “Follow” relationship between a Diet and Standard User

```
"MATCH (user:User)-[follows:FOLLOWS]->(diet:Diet) WHERE user.username = $username " +
"AND diet.id = $id SET diet.followersCount = diet.followersCount - 1 " +
"DELETE follows",
```

Delete Diet and all its relationships

```
"MATCH (diet: Diet{id: $id}) DETACH DELETE diet"
```

Create a Diet and its relationship with the corresponding Nutritionist

```
"MERGE (:Diet {id: $id, name: $name, followersCount: 0, succeededCount: 0, failedCount: 0})"
```

Analytics Implementations – Neo4j

Find the most followed diet

```
"MATCH (diet: Diet) RETURN diet.id AS ID ORDER BY diet.followersCount DESC LIMIT 1"
```

Find the most popular diet

```
"MATCH (diet: Diet) RETURN diet.id AS ID " +
"ORDER BY diet.succeededCount+diet.followersCount DESC LIMIT 1"
```

Find the most succeeded diet

```
"MATCH (diet: Diet) RETURN diet.id AS ID ORDER BY diet.succeededCount DESC LIMIT 1"
```

Find the most recommended diet

```
"MATCH (user1: User)-[f1:FOLLOWS]->(diet1: Diet)<-[f2:FOLLOWS]-(user2: User)-[f3:FOLLOWS]->(diet2: Diet) " +
    "WHERE f1.status = f2.status AND user1.username = $username " +
    "AND f3.status = \"succeeded\" WITH diet2, count(*) AS validArrowCount " +
    "RETURN diet2.id AS ID ORDER BY validArrowCount DESC LIMIT 1",
```

Find the most popular nutritionist

```
"MATCH (nutritionist: Nutritionist)-[:PROVIDES]->(diet: Diet) " +
    "RETURN nutritionist.username AS username " +
    "ORDER BY diet.succeededCount+diet.followersCount DESC LIMIT 1"
```

Find the most followed diet by nutritionist

```
"MATCH (nutritionist: Nutritionist)-[:PROVIDES]->(diet: Diet) "+
"WHERE nutritionist.username = $username " +
"RETURN diet.id AS ID ORDER BY diet.followersCount DESC LIMIT 1"
```

Sharding proposal

Accordingly to Non-functional requirements, the application ensures AP edge of the CAP triangle, namely availability and partition tolerance.

The possibility to implement a sharding for the database would allow to further reduce the latency/response time for server interactions, improving user experience.

In that regard, we thought to partition the document database in shards hosted each one on a different server. Such sharding, together with the already implemented data replication, would allow for load-balancing purposes, achieving better performances. To implement such sharding, we would select three different sharding keys, one for each collection of our document database:

- Foods collection: key “category”
- Users collection: key “country”
- Diets collection: key “nutritionist”

We selected the hashing partitioning method, in order to map the fields chosen as sharding key through a hash function.