



# UNIVERSITÀ DI PISA

## Large-scale and Multi-structured Databases

### Project SafeBite



Samay A. S. Ruiz Muenala  
Basma Adawy  
Aways Sarkhot

A.Y. 2023-2024

# Index

Introduction.....	4
Dataset .....	5
Products .....	5
Reviews.....	5
Users .....	6
Main Actors .....	7
Functional Requirements.....	7
Non-Functional Requirements .....	8
Use case diagram.....	9
UML class diagram.....	9
Database organization.....	10
MongoDB .....	10
Users .....	10
Products .....	11
Reviews .....	12
Neo4j .....	13
Software architecture .....	15
Package structure .....	15
Recommendations implementation .....	16
Products suggestions .....	16
Users suggestions.....	17
Data analytics and statistics .....	18
MongoDB queries.....	18
Neo4j queries .....	21
Database management .....	22
Consistency .....	22
Replication .....	22
Configuration parameters.....	23

Sharding.....	23
Indexes .....	24
Improvements .....	25

# Introduction

*SafeBite* is a specialized web application designed to facilitate the discovery of gluten-free products based on individual dietary restrictions.

Users can create a personalized diet profile, indicating diet type (halal, vegan, vegetarian, pescatarian, or normal) and allergens. This information forms the basis for personalized product recommendations, aligning with unique dietary restrictions.

Additionally, *SafeBite* fosters community engagement through integrated social media features, allowing users to connect with others who share similar dietary preferences.

The platform offers a curated catalogue of gluten-free products for ensuring efficient experiences for those with gluten intolerance or those opting for a gluten-free lifestyle.

Application code is accessible through the following GitHub repository URL:

<https://github.com/sisarui/SafeBiteProject>

# Dataset

## Products

All data on products has been retrieved from *Open Food Facts*. *Open Food Facts* is a collaborative, global database of more than 3 million food products with ingredients, allergens, nutrition facts and all information we can find on product labels.

As it is maintained through the contribution of more than 100.000 volunteers, the dataset includes inconsistencies, missing values, and non-standardized data formats (use of different languages such as English, French, Spanish, ... for instance).

Although we have performed a certain level of data cleansing on our dataset and given that the focus of this project is on non-relational databases design rather than on data accuracy and consistency, we do not guarantee absolute validity of our data. Therefore, users may still get recommendations for products that are not compatible with their dietary restrictions due to limitations within the initial dataset.

Much effort has been made to filter out gluten-containing products, as well as products without any ingredient information from our collection.

The raw dataset has been downloaded from Kaggle:  
[https://www.kaggle.com/datasets/konradb/open-food-facts?resource=download&select=en.openfoodfacts.org.products\\_out.csv](https://www.kaggle.com/datasets/konradb/open-food-facts?resource=download&select=en.openfoodfacts.org.products_out.csv)

Its original size was 8.04 GB. After undergoing cleansing and filtering stages, it went down to 120.58 MB.

## Reviews

The reviews collection has been obtained through scraping of two online groceries shopping websites. *Morrisons*<sup>1</sup>, fifth largest supermarket chain in the United Kingdom, and *Ocado*<sup>2</sup>, a British business which licenses grocery technology.

Coherently with the scope of our project we decided to scrape only reviews from the “*Free from Gluten*” sections.

---

<sup>1</sup> <https://groceries.morrisons.com/webshop/startWebshop.do>

<sup>2</sup> <https://www.ocado.com/webshop/startWebshop.do>

From the data at our disposal, we decided to extract the following fields: rating score (integer from 1 to 5), header, date, text. We subsequently implemented a random matching process to associate each review to a user and product existing in our database.

The dataset has observed a modest increase in size. It eventually amounted to 282.62 kB.

## Users

With regards to the user's data, we had no choice but to create a synthetic dataset ourselves. This was done by executing a Python script<sup>3</sup> with import of the *Faker* package.

For each user we have generated username, email, password, country, date of birth, gender.

Additionally, since users needed to have a dietary profile, we have implemented a random pairing system to ensure that each user is assigned a specific diet type along with a list of allergens.

The user's data therefore consists of 306 users, 2 admins included. Its size is 65.54 kB.

---

<sup>3</sup> *usersInfo.py*

# Main Actors

*SafeBite* has 3 main actors:

- **Admin**

He is the administrator of the application. He has special privileges that allow him, for example, to delete users' accounts or to do actions that the other users cannot do.

- **Unregistered Users**

Users that use the application for the first time. They need to create an account to access *SafeBite* services.

- **Registered Users**

Users that already have accounts on the application need to sign in using their credentials to access their personal area and *SafeBite* services.

# Functional Requirements

- **An unregistered user can**
  - Sign up to create his own account to start accessing the application services.
- **A registered user can**
  - Log in to the application.
  - Log out from the application.
  - View/Update his profile.
  - Search for products and view their pages with reviews.
  - Like/Unlike products.
  - Rate and write comments on products.
  - View his liked products.
  - View the distribution of ratings he gave to products over time.
  - Get products recommendations.
  - Search for users and view their profiles.

- View his connections.
- Follow/Unfollow other users.
- Get users recommendations.
  
- **The admin can**
  - Log-in/Log-out the application.
  - Search for a product.
  - Add/Delete/Update products.
  - Remove reviews.
  - Search for a user.
  - View/Delete a user profile.
  - View users and products analytics.

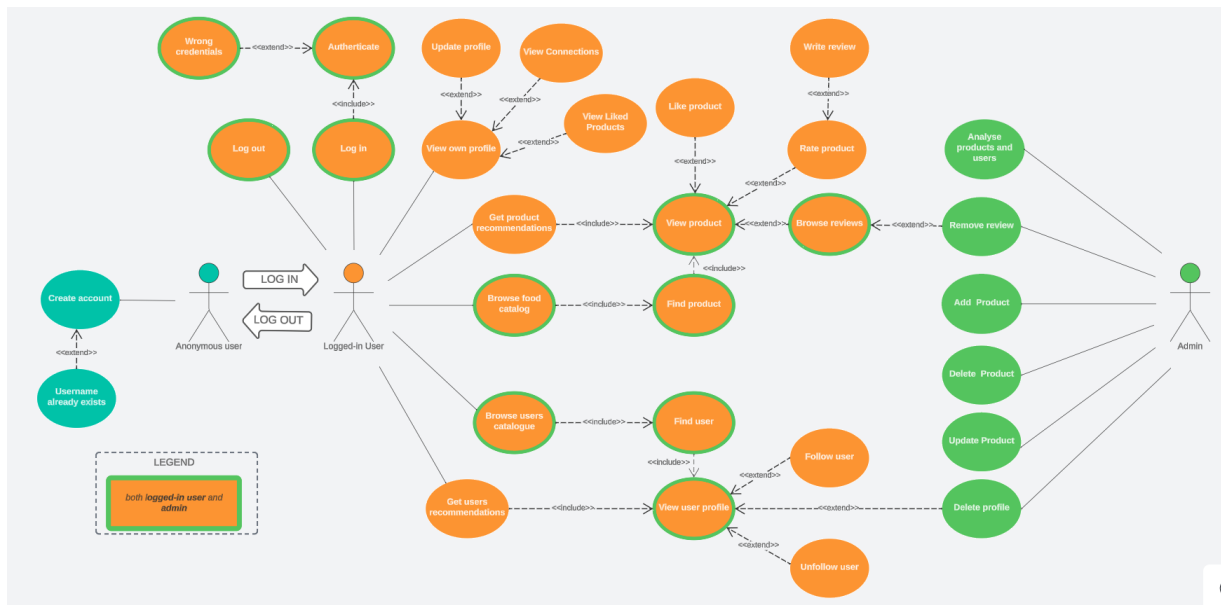
## Non-Functional Requirements

Non-functional requirements of our application are:

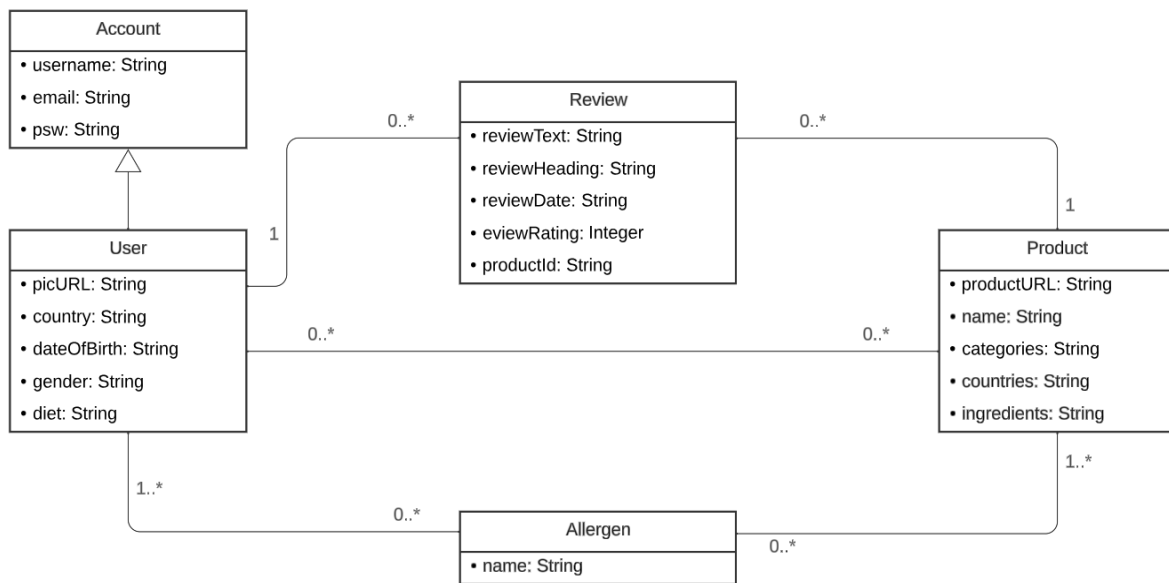
- **Usability:** To meet the requirement of being easy to use and understand for a common user, we implemented a graphical user-friendly interface.
  
- **Performance:** Addressing the need for minimal response time, we optimized performance by applying indexes and configuring MongoDB with a write concern of W1 and a read preference set to nearest, ensuring a swift and responsive user experience.
  
- **Availability and Partition Tolerance:** In response to the requirement for continuous accessibility, even during system failures and network partitions, we adopted a robust approach by using two databases – MongoDB with 3 replicas and Neo4J with 1 replica. While acknowledging potential inconsistencies, this design always prioritizes application availability for users.



## Use case diagram



## UML class diagram



# Database organization

We made use of two non-relational databases: MongoDB (as Document DB) and Neo4j (as Graph DB).

## MongoDB

The MongoDB database (*SafeBite*) comprises three collections: *Users*, *Products*, *Reviews*.

### Users

Each document in the *Users* collection represents a single service user entity. As such it consists of the following fields:

- *\_id*: id automatically generated by Mongo (ObjectId)
- *user\_name*: username (String). Unique for each user.
- *email*: e-mail address of the user (String)
- *password*: user's password (String)
- *country*: user's country location (String)
- *date\_of\_birth*: user's date of birth (String)
- *gender*: user's gender (String)
- *diet\_type*: diet type followed by user (String). Any between "vegan", "vegetarian", "pescatarian", "halal", "none".
- *allergy*: embedded document with one field only called *allergens*. *Allergens* is an array of Strings. The elements of this array can be any of the 30 allergens displayed by the application in the signup process<sup>4</sup>.
- *admin*: if true indicates that the account is of an administrator. (Boolean)

---

<sup>4</sup> These allergens are: "celery", "wheat", "rye", "barley", "oats", "spelt", "kamut", "cabbage", "broccoli", "cauliflower", "kale", "brussel sprouts", "collard greens", "crustaceans", "eggs", "fish", "milk", "lupin", "molluscs", "mustard", "nuts", "peanuts", "sesame", "soy", "tomato", "apple", "peaches".

```

lsmdb [direct: primary] SafeBite> db.Users.find().limit(1)
[
  {
    _id: ObjectId('65a278b9131583f9a215c660'),
    user_name: 'briandavis',
    email: 'mullinschristopher@example.net',
    password: '4261eca71d7b763f',
    country: 'Pitcairn Islands',
    date_of_birth: '1952-07-03',
    gender: 'Male',
    diet_type: 'vegan',
    allergy: { allergens: [ 'celery', 'cabbage', 'crustaceans', 'lupin' ] },
    admin: false
  }
]

```

## Products

Each document in *Products* represents a single food product entity. Fields may vary by document. The complete schema of such documents comprises the following fields:

- *\_id*: id automatically generated by Mongo (ObjectId)
- *created\_datetime*: timestamp of the document insertion (Date)
- *last\_modified\_datetime*: timestamp of the last update on the document (Date)
- *last\_modified\_by*: username of the administrator who last modified the document (String)
- *product\_name*: name of the food product (String)
- *quantity*: quantity and unit of measurement of food package (String)
- *brands*: brands of the product (String)
- *categories\_tags*: food categories to which the product belongs (String)
- *labels\_tags*: labels associated to the product (String)
- *countries\_en*: countries where the product is available (String)
- *ingredients\_tags*: product ingredients (String)
- *ingredients\_analysis\_tags*: tags based on ingredients (String)
- *allergens*: allergens contained in food product
- *food\_groups\_en*: broader food groups which the product belongs to (String)
- *main\_category*: main category which the product belongs to (String)
- *image\_url*: URL to the product image (String)

```
{
  _id: ObjectId('65aa4a8f043c65daf29a2b21'),
  created_datetime: ISODate('2015-10-11T14:09:21.000Z'),
  last_modified_datetime: ISODate('2015-10-12T14:13:32.000Z'),
  last_modified_by: 'sam',
  product_name: 'moutarde au goût de raisin ',
  quantity: '100g',
  brands: 'courte paille',
  categories_tags: 'en:condiments,en:saucen:mustards,en:groceries',
  labels_tags: 'fr:delois-france',
  countries_en: 'France',
  ingredients_tags: 'fr:eau-graines-de-teguments-de-moutarde-vinaigre-de-vin-rouge-sel-vin-rouge-sucrer-mout-de-raisin,fr:oignons-colorants-extraits-de-carotte-et-extrait-de-paprika-huile-de-tournesol-son-de-moutarde-sel,fr:cette-moutarde-uniquement-disponible-chez-courte-paille',
  ingredients_analysis_tags: 'en:palm-oil-content-unknown,en:vegan-status-unknown,en:vegetarian-status-unknown',
  allergens: 'en:mustard',
  food_groups_en: 'Fats and sauces,Dressings and sauces',
  main_category: 'en:groceries',
  image_url: 'https://images.openfoodfacts.org/images/products/000/000/000/0100/front.3.400.jpg'
},
```

## Reviews

Each *Reviews* document is made up of the following fields:

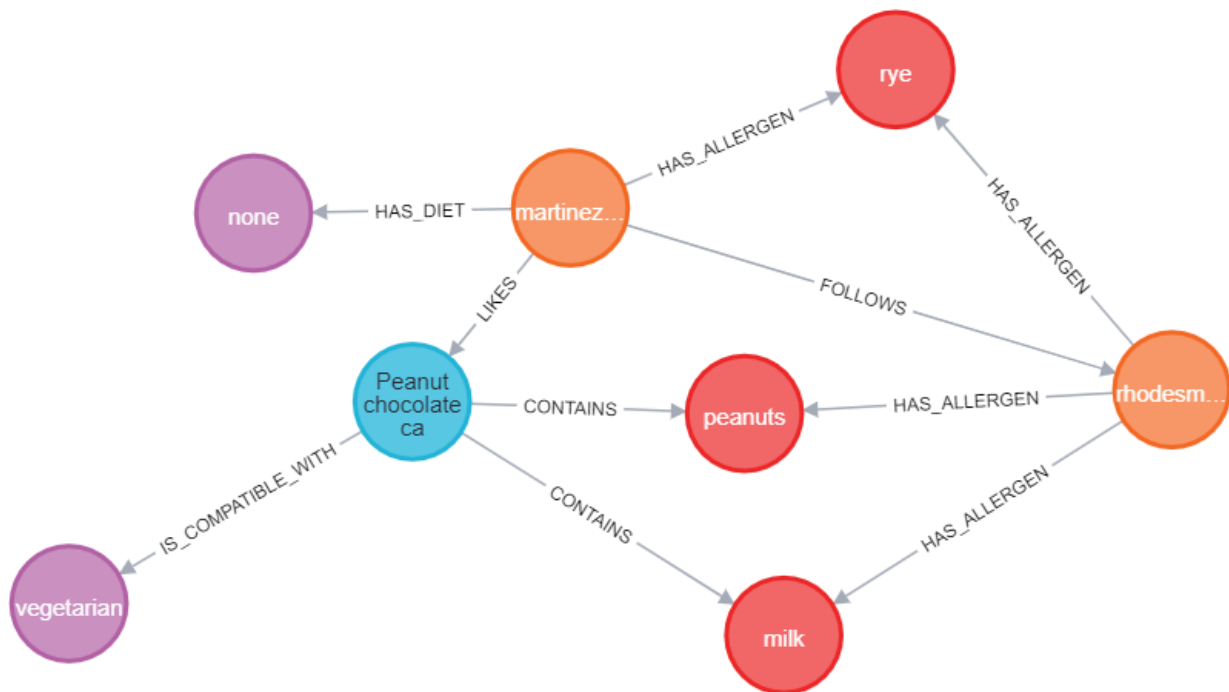
- *\_id*: id automatically generated by Mongo (ObjectId)
- *Review Rating*: rating score. Any integer between 1 and 5, included (Integer)
- *Review Heading*: heading of the review (String)
- *Review Text*: text of the review (String)
- *Review Date*: date when the review was written (String)
- *Product Name*: name of the reviewed product (String)
- *User*: author of the review (String)
- *Product ID*: id of the product (ObjectId)

```
lsmdb [direct: primary] SafeBite> db.Reviews.find().limit(1)
[
  {
    _id: ObjectId('65aa96583f1831edaeb7e1b3'),
    'Review Rating': 1,
    'Review Heading': 'Vile',
    'Review Text': 'Texture and taste were really unpleasant. Sorry but there was nothing positive to say at all. They also needed longer to cook than on the pack',
    'Review Date': '25 December 2022',
    'Product Name': 'Maine maple syrup',
    User: 'chelseaneal',
    'Product ID': ObjectId('65aa4af4043c65daf2a2faf7')
  }
]
```

As shown in the picture we decided to include Product ID in Reviews fields. This way we solved the many-to-one relationship between Review and Product. The Product name has been stored as well so to connect to one collection only for displaying reviews.

## Neo4j

Our Neo4j database is mainly used for storing data on relations between different entities.



Therefore, entity nodes only contain the necessary data for the implementation of our graph queries.

Our graph database stores three different entity nodes with different labels:

- **User:** only property is *user\_name*
- **Product:** its properties are *id* and *name*
- **Allergen:** only property is *name*
- **Diet:** only property is *type*


























As for relations, we have the following 6:

- (Product)--**CONTAINS**-->(Allergen): this relation signifies that a product contains a specific allergen.
- (Product)--**IS\_COMPATIBLE\_WITH**-->(Diet): this relation implies that the product aligns with a specific diet.
- (User)--**HAS\_DIET**-->(Diet): this relation indicates that a user has a specific diet.
- (User)--**HAS\_ALLERGEN**-->(Allergen): this relation denotes that the user is allergic to a particular substance.
- (User)--**FOLLOWS**-->(User): this relation represents a social connection where one user follows another user.

# Software architecture

The software architecture of our web application is designed with a layered approach, following a *layered architecture* style. This structured organization separates the application into distinct layers, each responsible for specific functionalities. On the client side, Jakarta Server Pages (JSP) files, cascading style sheets (CSS), and JavaScript collectively shape the user interface. Meanwhile, server-side logic is implemented through Servlets and Data Access Object (DAO) files, including the Neo4jManager class for seamless interaction with the Neo4j database. The deployment architecture involves the utilization of virtual machines (VMs) for hosting our databases.

## Package structure

- ✓  > SafeBiteProject [SafeBiteProject ma
  - >  Deployment Descriptor: SafeBitePi
  - >  JAX-WS Web Services
  - >  Java Resources
  - >  Deployed Resources
  - >  build
  - ✓  src
    - ✓  main
      - ✓  java
        - >  dao
        - >  follow
        - >  login
        - >  model
        - >  mongo
        - >  product
        - >  review
        - >  search
        - >  signup
        - >  update
        - >  user
        - >  utilities
      - >  webapp
    - >  target
      -  pom.xml
      - >  Servers

- *src/main/java*: this directory contains the source code for the Java classes of the application. It includes packages for data access objects (DAOs), models, and utilities.

- *src/main/webapp*: this directory contains the web application resources, including JSP, CSS, Javascript files. The webapp directory also contains a WEB-INF directory, which stores deployment configuration files such as web.xml.

- *target*: this directory is generated by Maven during the build process. It contains compiled class files, JAR files, and other artifacts that are ready to be deployed to a web server.

- *pom.xml*: this file is the Maven project object model (POM) file.

- *src/main/java*: this directory contains the source code for the Java classes of the application. It includes the following packages:

  - > *dao*: this package contains *UserDAO.java*, *ProductDAO.java*, *ReviewDAO.java* and *Neo4jManager.java*. Each DAO corresponds to a specific data collection in MongoDB (Users, Products, Reviews)

and offers CRUD operations on entities within. *Neo4jManager.java* manages connections and interactions with the Neo4j graph database by means of the neo4j java driver. Moreover, it provides methods for executing analytical queries against Neo4j.

> *model*: this package defines the data model of the application through Java classes. *User.java*, *Product.java*, *Review.java* represent the structure of data, defining attributes and behaviours of users, products, and reviews.

> *mongo*: this package houses one class file only, *MongoAggregations.java*. This file contains utility methods for constructing complex aggregation queries against MongoDB collections, enabling data analysis.

All other packages contain servlet files. Each servlet handles incoming HTTP requests, interacts with DAOs and models, and generates responses for the web application.

## Recommendations implementation

We have implemented two methods for providing recommendations to users. Both leveraging the Neo4j database. The functions are defined within the *Neo4jManager* class.

### Products suggestions

The first one, *findRecommendedProducts()*, aims to provide products compatible with the user's dietary profile.

```
public List<String> findRecommendedProducts(String userName) {
    List<String> recommendedProducts = new ArrayList<>();

    try {
        Result result = neo4jSession.run(
            "MATCH (user:User {user_name: $userName})-[:HAS_DIET]->(diet:Diet) " +
            "OPTIONAL MATCH (user)-[:HAS_ALLERGEN]->(allergen:Allergen) " +
            "MATCH (product:Product) " +
            "WHERE " +
            "(diet.type = 'none' AND NOT (allergen IS NOT NULL AND (product)-[:CONTAINS]->(allergen)) AND NOT (user)-[:LIKES]->(product)) " +
            "OR " +
            "(diet.type <> 'none' AND (product)-[:IS_COMPATIBLE_WITH]->(diet) AND NOT (allergen IS NOT NULL AND (product)-[:CONTAINS]->(allergen)) " +
            "AND NOT (user)-[:LIKES]->(product))) " +
            "RETURN DISTINCT product.id AS productId, product.name AS productName " +
            "LIMIT 10",
            Values.parameters("userName", userName)
        );

        while (result.hasNext()) {
            Record record = result.next();
            String productId = record.get("productId").asString();
            String productName = record.get("productName").asString();
            String combined = productId + "-" + productName;
            recommendedProducts.add(combined);
        }

        return recommendedProducts;
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("Error finding recommended products: " + e.getMessage());
        return null;
    }
}
```

This is done following a graphic-centric approach as we first examine the relationships stemming from the User node (retrieved via the *userName* argument). More specifically the



user's diet type and eventual allergens are collected by leveraging *HAS\_DIET* and *HAS\_ALLERGEN* relations. Afterwards, the *type* field of the Diet node is analysed. If it differs from "none" then Product nodes are first checked for diet compatibility by means of the *IS\_COMPATIBLE\_WITH* relation. Otherwise, products are immediately filtered based on the allergen nodes they are associated with through the *CONTAINS* relation. If there are no such relationships between the Product node and the User's allergens, then the product node is still considered. The last filter is made on the *LIKE* relation. The query checks for the lack of this relationship for providing suggestions not already familiar to the user.

## Users suggestions

Users are recommended using the *findPotentialFriends()* method. This is done by looking into the already existing connections of the User.

```
public List<String> findPotentialFriends(String userName) {
    List<String> potentialFriends = new ArrayList<>();

    try {
        Result result = neo4jSession.run(
            "MATCH (user:User {user_name: $userName})-[:FOLLOWS]->(friend)-[:FOLLOWS]->(potentialFriend) " +
            "WHERE NOT (user)-[:FOLLOWS]->(potentialFriend) " +
            "RETURN DISTINCT potentialFriend.user_name AS potentialFriendName",
            Values.parameters("userName", userName)
        );

        while (result.hasNext()) {
            Record record = result.next();
            String potentialFriendName = record.get("potentialFriendName").asString();
            potentialFriends.add(potentialFriendName);
        }

        return potentialFriends.isEmpty() ? null : potentialFriends;
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("Error finding potential friends: " + e.getMessage());
        return null;
    }
}
```

More specifically, the cypher query retrieves nodes connected to the user through the *FOLLOWS* relationship, creating a path that includes a mutual friend who, in turn, follows a potential friend. To ensure that the potential friend is not already directly connected to the user, a filtering condition is applied to exclude existing *FOLLOWS* relationships between the user and the potential friend. After processing all results, the method returns the list of potential friend usernames.

# Data analytics and statistics

In this section we describe the application's analytics aspect. We briefly go through each functionality and provide aggregation pipeline and cypher queries involved in the implementation.

## MongoDB queries

1. **Gender distribution by diet:** implemented by the *calculateGenderPercentagebyDietType()* method. For each diet type it returns the percentages of the users' gender following such diet.

```
[
  {
    "$group": {
      "_id": {
        "diet_type": "$diet_type",
        "gender": "$gender"
      },
      "count": { "$sum": 1 }
    }
  },
  {
    "$group": {
      "_id": "$_id.diet_type",
      "totalUsers": { "$sum": "$count" },
      "genderCounts": {
        "$push": {
          "gender": "$_id.gender",
          "count": "$count"
        }
      }
    }
  },
  {
    "$project": {
      "_id": 0,
      "diet_type": "$_id",
      "percentages": {
        "$map": {
          "input": "$genderCounts",
          "as": "gc",
          "in": {
            "gender": "$$gc.gender",
            "percentage": {
              "$multiply": [
                {
                  "$divide": [ "$$gc.count", "$totalUsers" ]
                },
                100
              ]
            }
          }
        }
      }
    }
  }
]
```

2. **Top product category tally:** implemented by the *getProductCategoryCounts()* method. It returns the number the top 10 categories by products count.

```
[
  { $group: { _id: "$main_category", count: { $sum: 1 } } },
  { $sort: { count: -1 } },
  { $limit: 10 },
  { $group: { _id: "$_id", count: { $sum: "$count" } } },
  { $sort: { count: -1 } }
]
```

3. **Top product counts by brand and country:** implemented by the *getProductCountByBrandAndCountry()* method. It returns the top 20 product counts, categorized by brand and country.

```
[
  {
    "$match": {
      "brands": { "$exists": true },
      "countries_en": { "$exists": true }
    }
  },
  {
    "$group": {
      "_id": {
        "brands": "$brands",
        "countries_en": "$countries_en"
      },
      "productCount": { "$sum": 1 }
    }
  },
  {
    "$project": {
      "brandOwner": "$_id.brands",
      "country": "$_id.countries_en",
      "productCount": 1,
      "_id": 0
    }
  },
  {
    "$sort": {
      "productCount": -1
    }
  },
  {
    "$limit": 20
  }
]
```

4. **Top 10 reviewers and average rating:** implemented by the *getTopReviewersAndAverageRating()* method. It returns a list of the users who reviewed the most products along with their average rating.

```
[
  {
    "$group": {
      "_id": "$User",
      "totalProductsReviewed": {
        "$sum": 1
      },
      "averageRating": {
        "$avg": "$Review Rating"
      }
    }
  },
  {
    "$sort": {
      "totalProductsReviewed": -1
    }
  },
  {
    "$limit": <limit>
  },
  {
    "$project": {
      "_id": 0,
      "User": "$_id",
      "totalProductsReviewed": 1,
      "averageRating": 1
    }
  }
]
```

5. **User rating distribution:** implemented by the *getUserRatingDistributionOverTime()* method. It returns information about a user's rating distribution over time, including the review dates, corresponding ratings, average rating for each date, and the total count of reviews on those dates.

```
[
  {
    "$match": {
      "User": "username"
    }
  },
  {
    "$group": {
      "_id": "$Review Date",
      "ratings": { "$push": "$Review Rating" },
      "count": { "$sum": 1 }
    }
  },
  {
    "$project": {
      "_id": 0,
      "Review Date": { "$toDate": "$_id" },
      "ratings": 1,
      "count": 1
    }
  },
  {
    "$addFields": {
      "averageRating": {
        "$cond": {
          "if": { "$eq": [ "$ratings", [ null ] ] },
          "then": null,
          "else": { "$arrayElemAt": [ "$ratings", 0 ] }
        }
      }
    }
  },
  {
    "$sort": {
      "Review Date": 1
    }
  }
]
```

## Neo4j queries

1. **Most popular users with number of followers:** implemented by the *getMostPopularUsersWithFollowersCount()* method. It returns the top 5 users with the most followers.

```
MATCH (u:User)-[:FOLLOWS]->(follower:User)
RETURN u.user_name AS username, COUNT(follower) AS followers
ORDER BY followers DESC LIMIT 5
```

2. **Number of followers per diet:** implemented by the *getMostFollowedDietsWithUserCount()* method. It returns the number of users following each diet.

```
MATCH (u:User)-[:HAS_DIET]->(diet:Diet)
RETURN diet.type AS dietType, COUNT(u) AS userCount
ORDER BY userCount DESC
```

3. **Number of users allergic to a specific allergen:** implemented by the *getAllergensWithUserCount()* method. For each allergen we get the number of users allergic to it.

```
MATCH (u:User)-[:HAS_ALLERGEN]->(allergen:Allergen)
RETURN allergen.name AS allergenName, COUNT(u) AS userCount
ORDER BY userCount DESC LIMIT 10
```

# Database management

## Consistency

In adherence to the principles outlined in the CAP theorem, our project implementation follows an Availability-Partition tolerance (AP) strategy. Consequently, we do not

guarantee perfect consistency among our databases, and discrepancies may occur in the event of unexpected issues arising during CRUD operations.

Specifically, during the creation, update, or deletion of User and Product documents/nodes, our methodology addresses potential inconsistencies by incorporating a safeguard mechanism. Our methods initiate the process with MongoDB. In case of a failure during the corresponding operation in Neo4j, a rollback mechanism is triggered, undoing the previous operation in MongoDB.

Furthermore, concerning CRUD operations involving Neo4j, our approach centers around the use of transactions to ensure data integrity. Notably, minimal intervention was required for MongoDB CRUD operations, as MongoDB inherently guarantees atomicity for single-document operations.

## Replication

Our MongoDB database is strategically configured with a three-replica set, each hosted on a separate virtual machine, featuring one primary node and two secondary nodes. This deployment enhances fault tolerance and high availability. The primary node manages write operations, while the secondary nodes replicate data, allowing for automatic failover and minimizing downtime in the event of a primary node failure.

```
rsconf = {_id: "lsmdb",
  members: [{_id: 0, host: "10.1.1.20:27017", priority:1},
            {_id: 1, host: "10.1.1.21:27017", priority:2},
            {_id: 2, host: "10.1.1.22:27017", priority:5}
  ]
};
```

In contrast, our Neo4j database is deployed as a single instance within our virtual cluster.

## Configuration parameters

In configuring our web application's interaction with MongoDB, we set the following parameters:

- **Write Concern:** the write concern is set to 1, indicating that write operations are considered successful once written to the primary node. This configuration optimizes availability.
- **Read Preference:** the read preference is set to "nearest," directing read operations to the nearest MongoDB replica set member. This optimizes read performance by minimizing network latency.
- **Timeout:** the timeout parameter for MongoDB operations is set to 5000 milliseconds (5 seconds). This setting defines the maximum duration allowed for the completion of a database operation, contributing to overall system responsiveness.

```
MongoClients.create("mongodb://10.1.1.20:27017,10.1.1.21:27017,10.1.1.22:27017/" +  
                    "?w=1&readPreferences=nearest&timeout=5000");
```

These configurations aim to maximize availability, following our chosen AP approach in system architecture.

## Sharding

In this paragraph, we conduct an analysis on the possibility of using sharding for our project, considering advantages and disadvantages this implementation would bring.

Since our application primarily serves as a catalog for browsing products and considering that all app users must have an account to use the service, the main data challenges may arise from the expanding size of the Products and Users collections. Therefore, sharding techniques may be implemented on these collections. Specifically, a sharding key on the Users collection may be implemented using the "user\_name" as the key, which would enhance user searching functions. Similarly, the Products collection can be sharded using "\_id" as the key.

Given the nature of our dataset, sharding based on diet compatibility and allergens cannot be implemented, as these relationships are managed by the Neo4j database. We acknowledge that it would be very convenient, especially for implementing a filter browsing function.

Regarding the sharding approach, and considering our desire to ensure maximum availability, a consistent sharding strategy would be the best option for our application.

## Indexes

One of the non-functional requirements for SafeBite application is the performance by addressing the need for the fast response to the user.

MongoDB, indexes play a crucial role in improving query performance by allowing the database to quickly locate and retrieve documents. For optimizing query performance, we focused on addressing delays in searching for products by name or substrings.

Before implementing indexes, product searches, such as looking for items with the substring "cookies," took an extended 1777 milliseconds to yield results.

```
Received searchTerm: cookies  
Constructed MongoDB Query: {"explainVersion": "1", "q  
MongoDB query took 1777 milliseconds.
```

In response, we strategically introduced a *product\_name\_text* index of type text within our Products collection, specifically designed for full-text searches on product names.

As a result of this the outcome was remarkable, the query response time significantly decreased to 280 milliseconds, enhancing the user experience and minimizing delays in providing relevant information.

```
DB connections closed successfully.  
Received searchTerm: cookies  
Query Execution Plan: {"explainVersion": "1", "q  
MongoDB query took 280 milliseconds.
```

Fortunately, our user searches by username did not encounter similar performance issues, likely due to the manageable number of users. However, we remain proactive in our approach. If faced with a scenario involving a substantial user base, we would consider creating additional indexes in the Users collection to maintain optimal performance.



## Improvements

We acknowledge that our application can undergo many improvements. For instance, as our application presents itself as a product catalog it would be convenient to implement a filtering option based on food category or country availability for instance. Another thing missing is the option to create more administrator accounts. Registration form allows only for normal users to sign up to the website.