UNIVERSITÀ DI PISA

MSc in Artificial Intelligence and Data Engineering

Large-Scale and Multi-Structured Database Project

L. Cocchella, A. Petrillo, G. Piacentini

https://github.com/LudovicaCi/ForkTalk.git

Academic year 2022 – 2023

Index

1. Introduction

ForkTalk is a Java-based web application that allows users to search, rate and book restaurants based on their preferences. The system features a user-friendly interface that enables users to search for restaurants based on parameters such as location, cuisine, rating, and price range.

Users can create restaurant lists, like them, and follow other users to get recommendations for new restaurants. They can also write reviews for restaurants they have visited and read reviews from other users to help them make informed decisions.

The system includes a secure user authentication mechanism that ensures that only registered users can access the application's features. An administrator account is also available to manage the content of the application, such as removing restaurants, restaurant lists and suspend users.

Overall, the ForkTalk System provides a convenient and efficient way for users to discover new restaurants, connect with other users, and book reservations with ease.

2. Analysis

2.1 Actors

There are three types of actors that can interact with the application: unregistered users, registered users (User and Admin) and Restaurant.

2.2 Functional Requirements

An **unregistered** user can:

- Create an account on the application.

A **registered user** can:

- login/logout
- manage its profile:
  - view, modify or delete its profile.
  - view or delete the reviews made by himself.
  - view, modify or delete its Restaurants Lists.
  - view or delete its reservations.
- search for Restaurants by:
  - Location.
  - Cuisine.
  - Price.
  - Ratings.
- search another User by username, name, surname and/or email.
- search Restaurants Lists by title.

- manage Restaurants Lists:
  - create a new Restaurants Lists.
  - add or remove Restaurants from their Restaurants Lists.
  - delete their Restaurants Lists.
  - follow Restaurants Lists made by other Users.
- follow another User.
- write reviews about Restaurants.
- make/delete a reservation in a Restaurant.
- View a list of the top-rated restaurants by city.
- View the list of restaurants with the highest lifespan.
- View the most active Users.
- view a list of the most followed Restaurants Lists.
- view a list of k suggested restaurants.
- view a list of k suggested users.
- view a list of k suggested restaurant List.

A **Restaurant** can:

- login/logout
- manage its profile:
  - view, modify, or delete its profile.
  - view or delete Reservations.
  - view reviews related to its own Restaurant.
  - view statistics regarding reviews.

An **Administrator** can:

- do all the operations that a User can do.
- delete Restaurants.
- delete Restaurant Lists.
- suspend/unsuspend a user.
- make/remove a user into an Administrator.

2.3 Non-functional requirements

The non-functional requirements of the application are described in the following list:

- **Performance**:
  - The application must respond quickly to user requests, with a low response time to improve the user experience.
  - The platform should support a high number of concurrent users, ensuring a good user experience.
- **Usability**:
  - The application needs to be user-friendly, easy to use and intuitive.

- **Reliability**:
    - The application needs to be highly available and always online.
    - Failures of individual nodes or connections between nodes should not affect the system. The application should be designed to handle such failures and maintain overall system functionality.
    - The operation of booking or deleting a reservation must be atomic, ensuring that it is either fully completed or not executed at all.
- **Scalability**:
    - The application should be designed to handle an increase in the number of users and data without compromising performance or usability.
- **Consistency**:
    - Eventual Consistency should be maintained to handle data consistency across multiple data sources.

2.4 Use-Case Diagram

In the following picture we can see the **Use-Case Diagram** of the application:

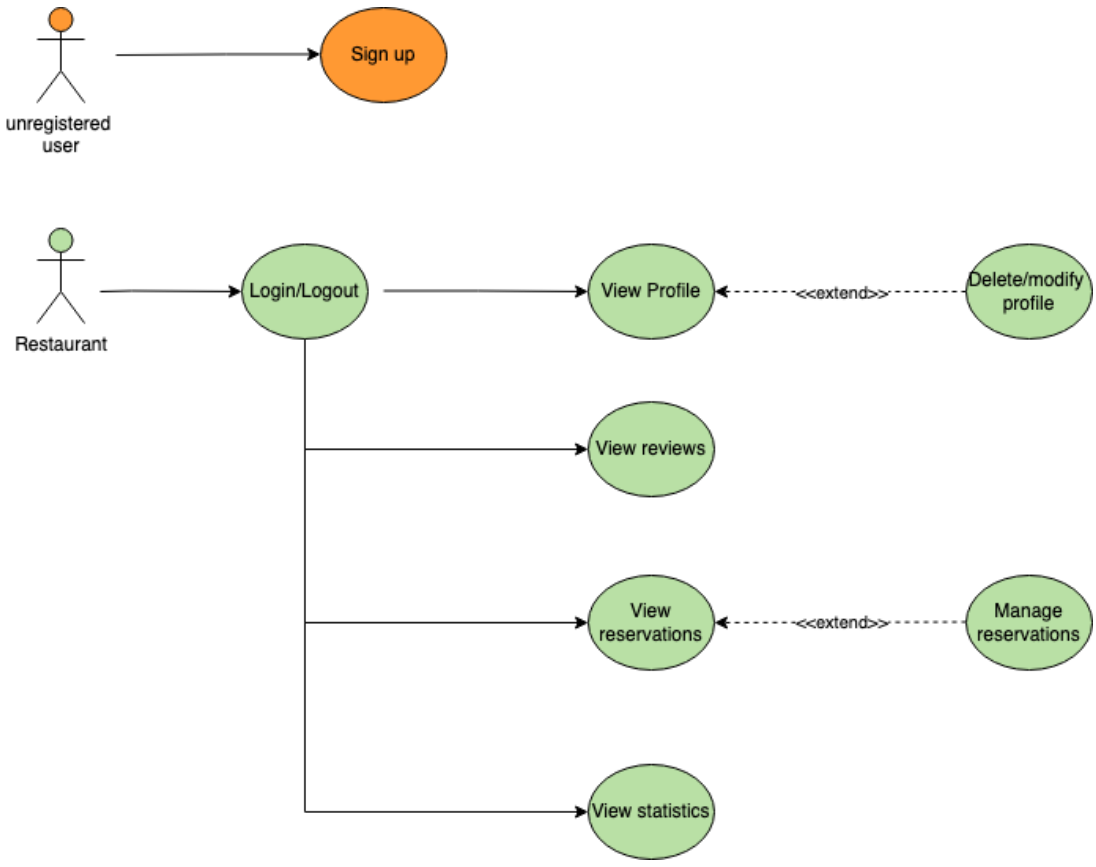


Figure 1: Use-Case Digram

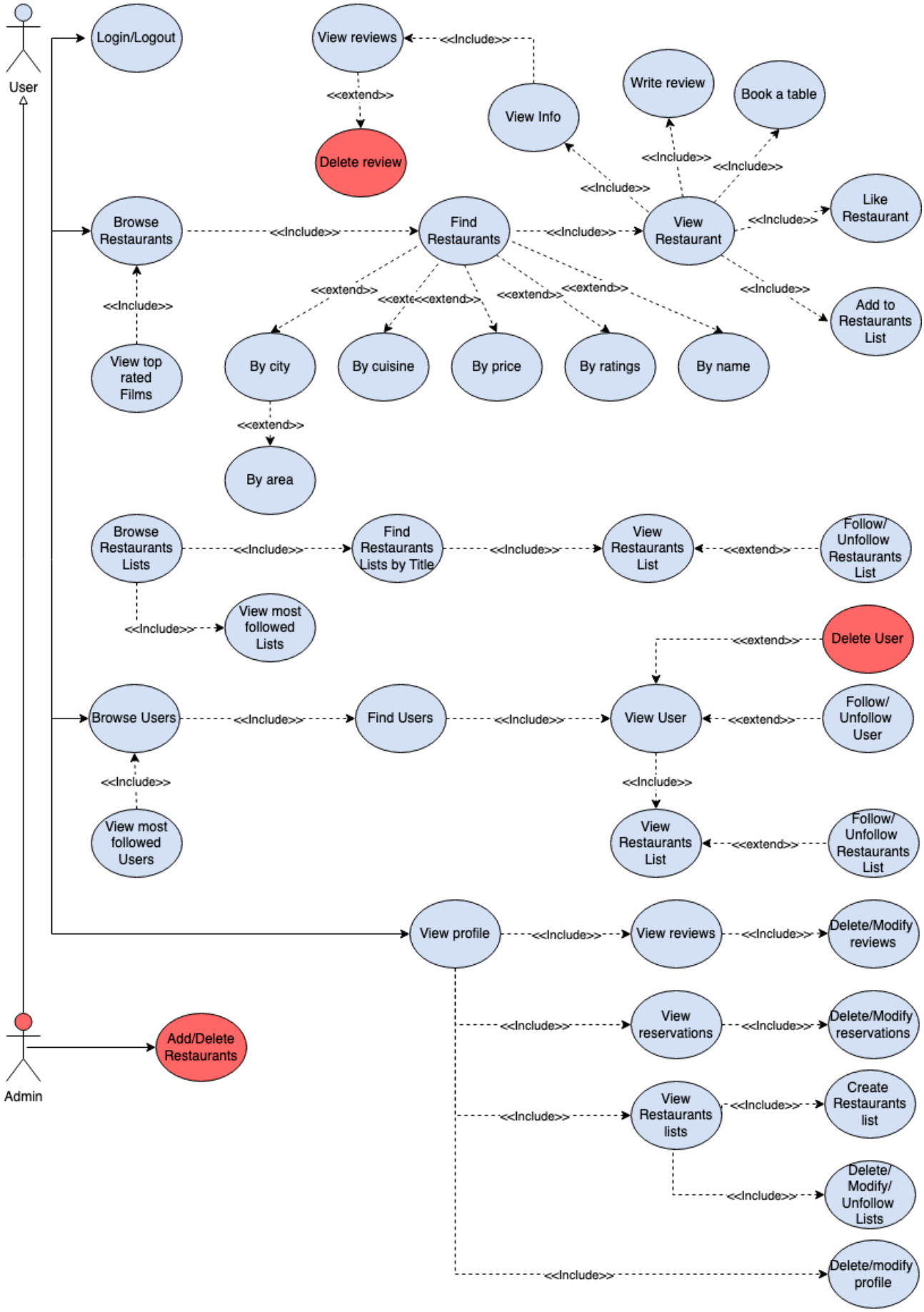


Figure 2: Use-Case Digram

2.5 Class Analysis

There are five main entities: User, Restaurant, Restaurant List, Review, Reservation. In the diagram below, User is a generalization of the two main actors: Registered User and Administrator.

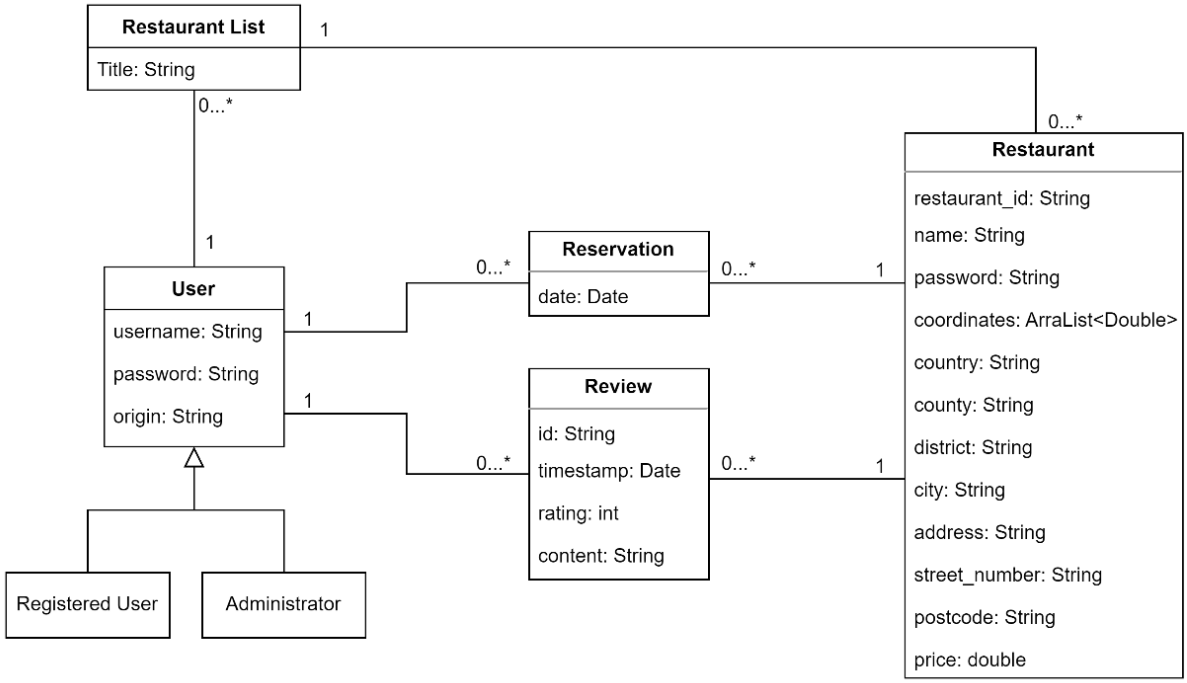


Figure 3: UML Class Diagram (with generalization)

We solve the generalization by adding one attribute to the class user for specify the role of the generic user.

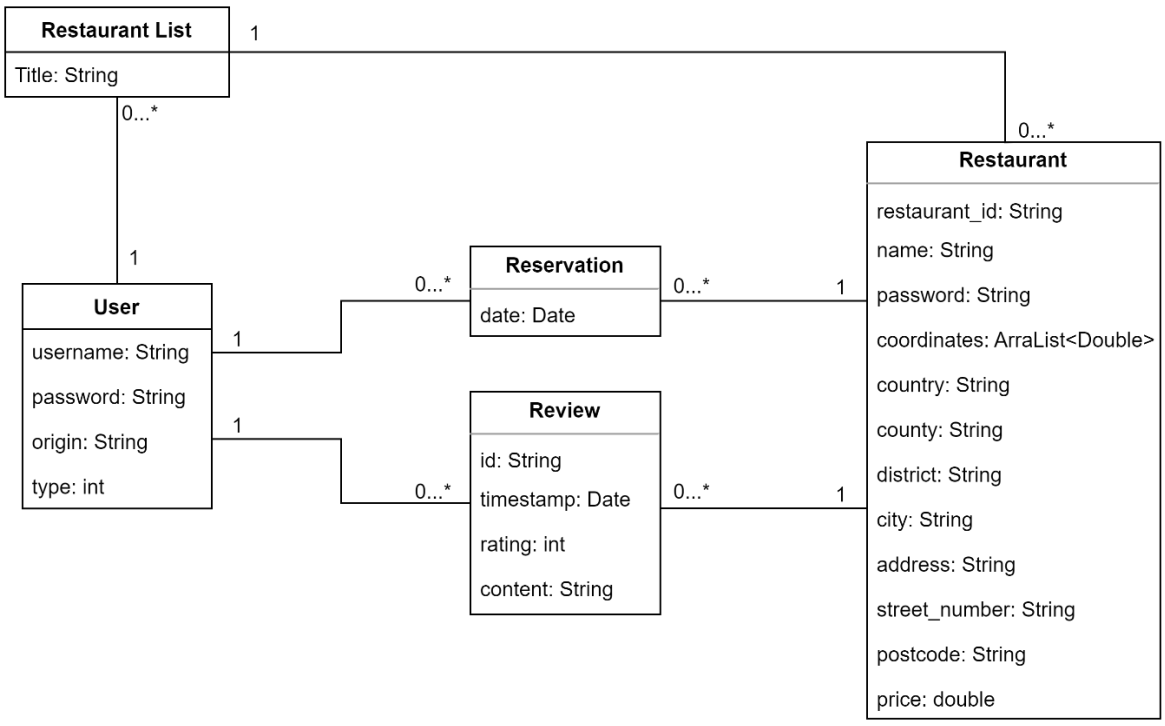


Figure 4: UML Class Diagram

The table below displays a list of all the attributes and their corresponding descriptions for each class:

USER CLASS		
FIELD	**TYPE**	**DESCRIPTION**
Username	String	Username of User
Password	String	Password
Origin	String	Nation of origin
Role	Int	IF 0 -> User IF 1 -> Administrator

RESTAURANT CLASS		
FIELD	**TYPE**	**DESCRIPTION**
restaurant_id	string	id of the restaurant
name	string	name of the restaurant
password	string	password
coordinates	ArrayList <double>	[latitude,longitude]
country	string	country
county	string	county
district	string	district
city	string	city
address	string	address
street_numer	string	street number
postcode	string	post code
price	double	average price p.p.
features	ArrayList<String>	Tags
reservations	ArrayList<Reservation>	Reservations
rating	Double	Rating
reviews	ArrayList<Reviews>	reviews

RESTAURANT LIST CLASS		
FIELD	**TYPE**	**DESCRIPTION**
title	string	title of the list
restaurants	ArrayList<restaurants>	list of restaurants

REVIEW CLASS		
FIELD	**TYPE**	**DESCRIPTION**
id	string	id
timestamp	date	date and time
rating	int	score (1-5)
content	string	Text
reviewer	String	Username of reviewer

RESERVATION CLASS		
FIELD	**TYPE**	**DESCRIPTION**
clientUsername	String	Username of client
clientName	String	Client name
clientSurname	String	Client surname
restaurantId	String	Id of the restaurant
restaurantName	String	Name of the restaurant
restaurantCity	String	City of restaurant

restaurantAddress	String	Address of restaurant
date	String	Date of the reservation
people	Int	Number of people

3. Design

3.1. Dataset

To achieve a realistic large-scale database, various sources were integrated and standardized, ensuring consistent data across the system. All information were obtained from the following sources:

1. https://www.kaggle.com/datasets/stefanoleone992/tripadvisor-european-restaurants
2. https://www.kaggle.com/datasets/arthurdfr/trip-advisor-review

The first link is a dataset containing all the information about European Restaurants, while the second one contains all the reviews corresponding to only the British Restaurants of the first dataset. Some preprocessing was performed to integrate the two datasets, standardize the data, and handle missing values.

For what concerns volume we have that the final dimension, obtained after the preprocessing phase, is about 167 MB.

The variability is obtained by adding reviews or make reservation to restaurants and deleting them by the administrator or by deleting them because they are too old.

3.2. Data organization

We decided to use two different databases for storing our data which are MongoDB as document database and Neo4j as a graph database.

MongoDB is used for storing data regarding the users and restaurants, while using Neo4j we are going to manage all the relationships between the entities and to implement the social part of our application as follow/unfollow users, like restaurants, follow/unfollow restaurant lists created by other users.

3.2.1. MongoDB – Documents organization

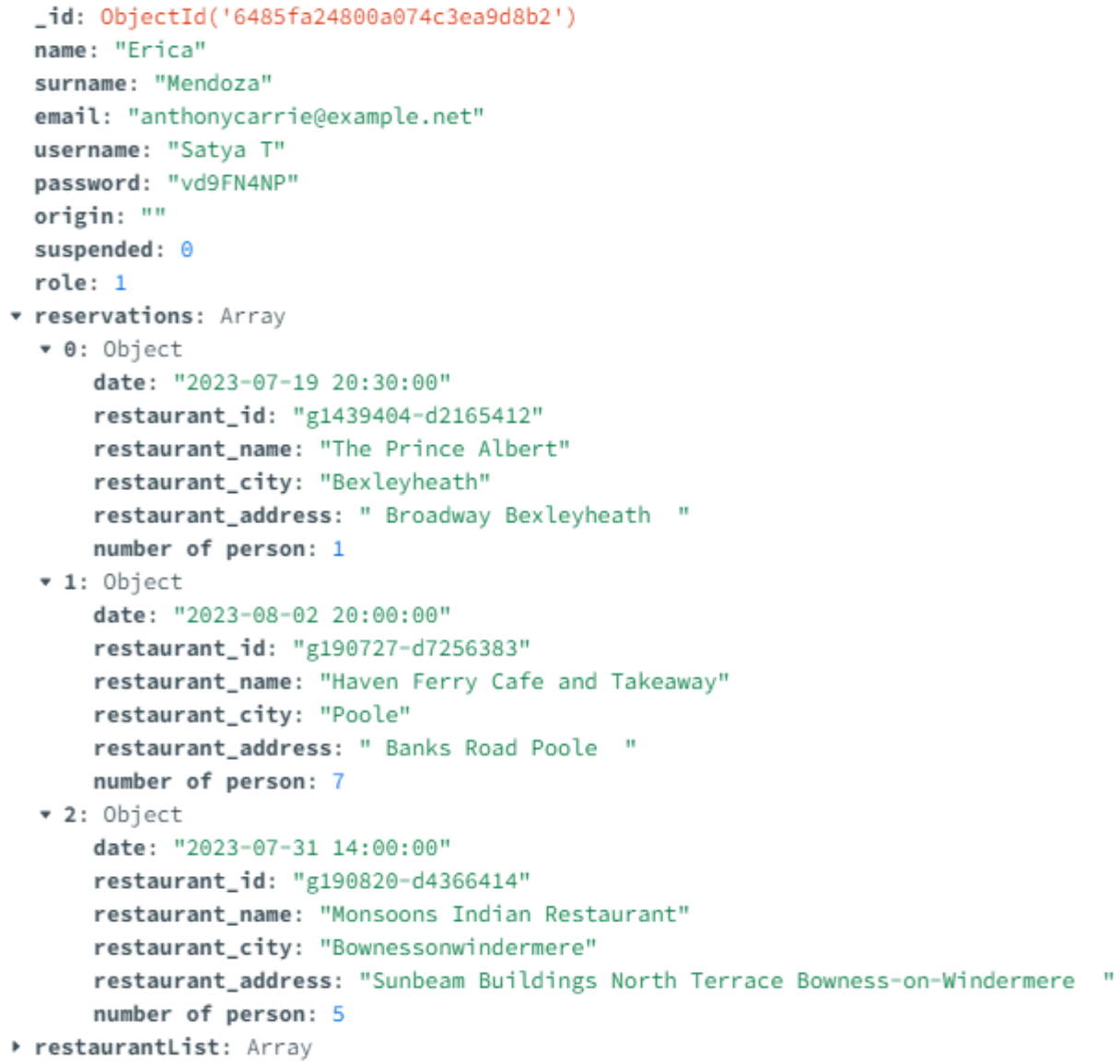
The entities handled by the document DB are:

- User
- Restaurant
- RestaurantList
- Reservation
- Review

In the document DB are stored two different collections:

- **Users**
- **Restaurants**

The collection “Users” is organized in this way:



The document of the collection “Users” contains all the personal information regarding the user, the password for the login, an array of documents where each document contains the information regarding the user's table reservations, and an array of documents containing all the restaurant lists made by the user for example a restaurant list named “favorite sushi restaurant in London” and the corresponding sushi restaurant selected by the user.

The collection “Restaurants” is organized in this way:

Figure 5: organization of Restaurant document

The document of the collection “Restaurants” contains all the general information regarding the restaurant, an array of strings which contains all the features of the restaurant as for example cuisine information, an array of documents containing the reviews written by the users about the restaurant and an array of documents containing the reservations.

3.2.2. Neo4j – Nodes organization

We decided to use a graph database to manage the social network features of the application. In the graph database there are about 219.961 nodes and 86.774 relationships. The nodes are the following:

1. The **User** node, representing the user registered within the application, which has as properties the username, the name and the surname.
2. The **Restaurant** node, representing the restaurant inside the application, having as properties the id.
3. The **RestaurantList** node, representing a list of restaurants created by a user, having as properties the title and the owner of the list.

Below there are the relationships between nodes which are:

1. **User – [:Follows] -> User**, which represents a user that follows another user.
2. **User – [:Follows] -> RestaurantList**, which represents a user that follows a RestaurantList created by a user.
3. **User – [:Likes] -> Restaurant**, which represents a user that put a like to a Restaurant.

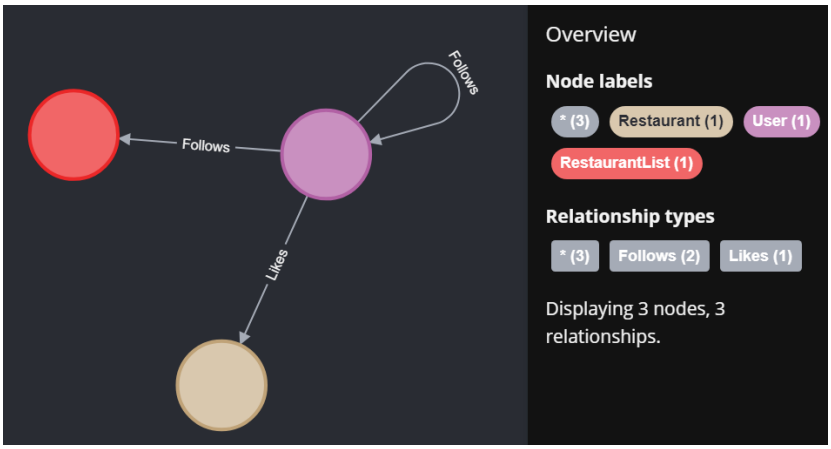


Figure 6: Neo4j Schema Visualization

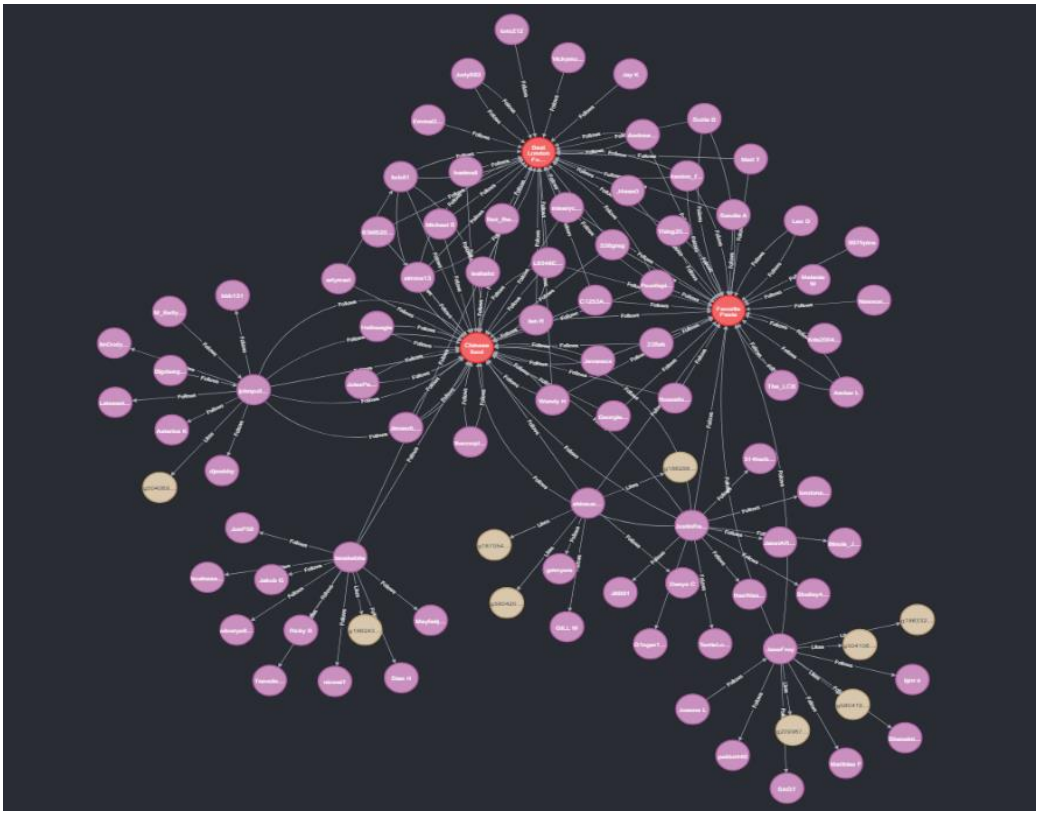
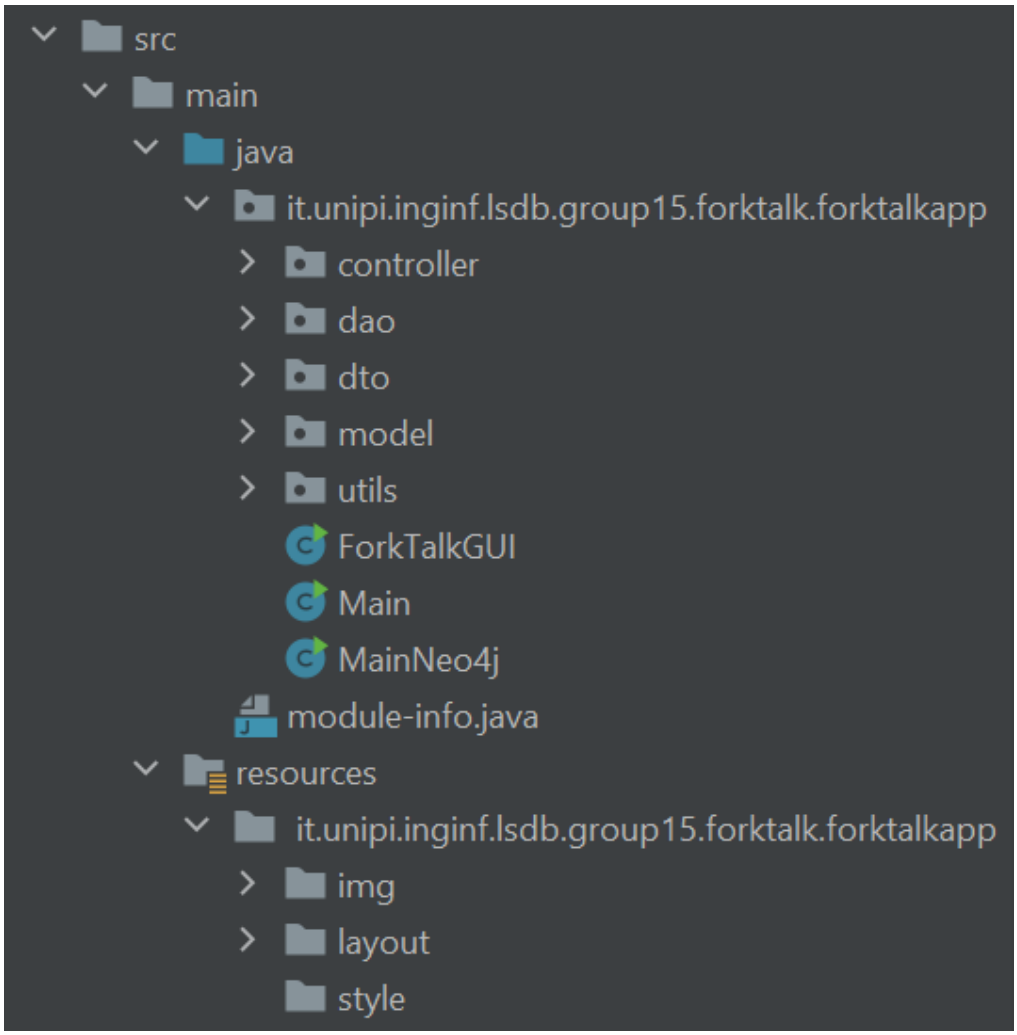


Figure 7: Partial View of the Neo4j Graph

4. Implementation

4.1. Package Structure

In the figure below we show how the implementation code is organized.



As we can see our application is composed of the following packages:

- **it.unipi.inginf.lsdb.group15.forktalk.forktalkapp.controller**
  This package contains the classes required for the controller part of the Model-view-controller of the application and for each different page to be shown to the user, a controller is implemented, which manages all the action taken by the user and updates the views.
- **it.unipi.inginf.lsdb.group15.forktalk.forktalkapp.dao.mongoDB**
  This package contains the classes required for implementing the connection and the access to MongoDB database.
- **it.unipi.inginf.lsdb.group15.forktalk.forktalkapp.dao.neo4j**
  This package contains the classes required for implementing the connection and the access to Neo4j database.

- **it.unipi.inginf.lsdb.group15.forktalk.forktalkapp.dto**
  This package contains the classes used to transfer data from the database to the application.

- **it.unipi.inginf.lsdb.group15.forktalk.forktalkapp.model**
  This package contains the classes which represent the entities of the database.

- **it.unipi.inginf.lsdb.group15.forktalk.forktalkapp.utils**
  This package contains the class which contains all the utility methods that are used by other classes in the application.

4.2. Most relevant queries

In the following section the most relevant queries performed with MongoDB and Neo4j will be presented.

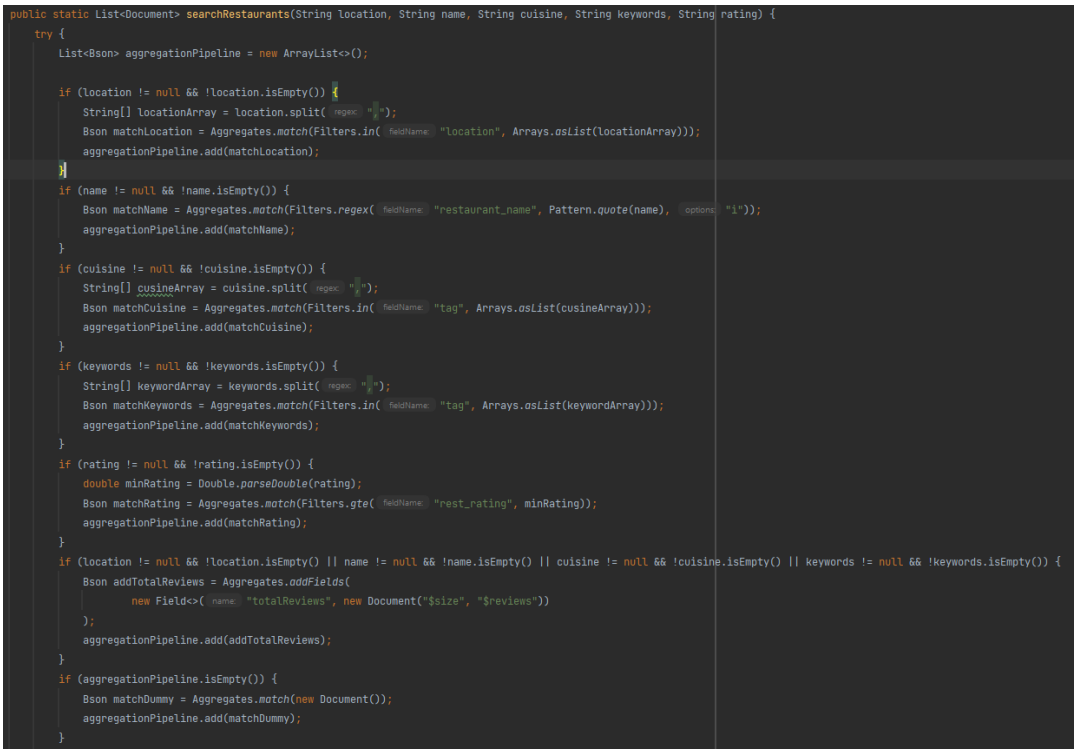
4.2.1. MongoDB – queries and analytics

These are the main CRUD queries that have been implemented:

- Create a new User.
- Create a new Restaurant.
- Create a new Restaurant List
- Create a new Reservation.
- Create a new Review.
- Retrieve User given username.
- Retrieve Restaurant List given User or Title.
- Retrieve Restaurant given the name or the Cuisine.
- Retrieve Restaurants given Restaurant List title.
- Retrieve Reviews given a Restaurant.
- Retrieve Reviews given a User.
- Retrieve Reservations given User.
- Retrieve Reservations given Restaurant.
- Update User.
- Update Restaurant.
- Update Restaurant List.
- Delete User.
- Delete Restaurant.
- Delete Review from Restaurant.
- Delete Restaurant List.
- Delete Reservation.
- Delete Restaurant from Restaurant List.

These are the search queries that have been implemented:

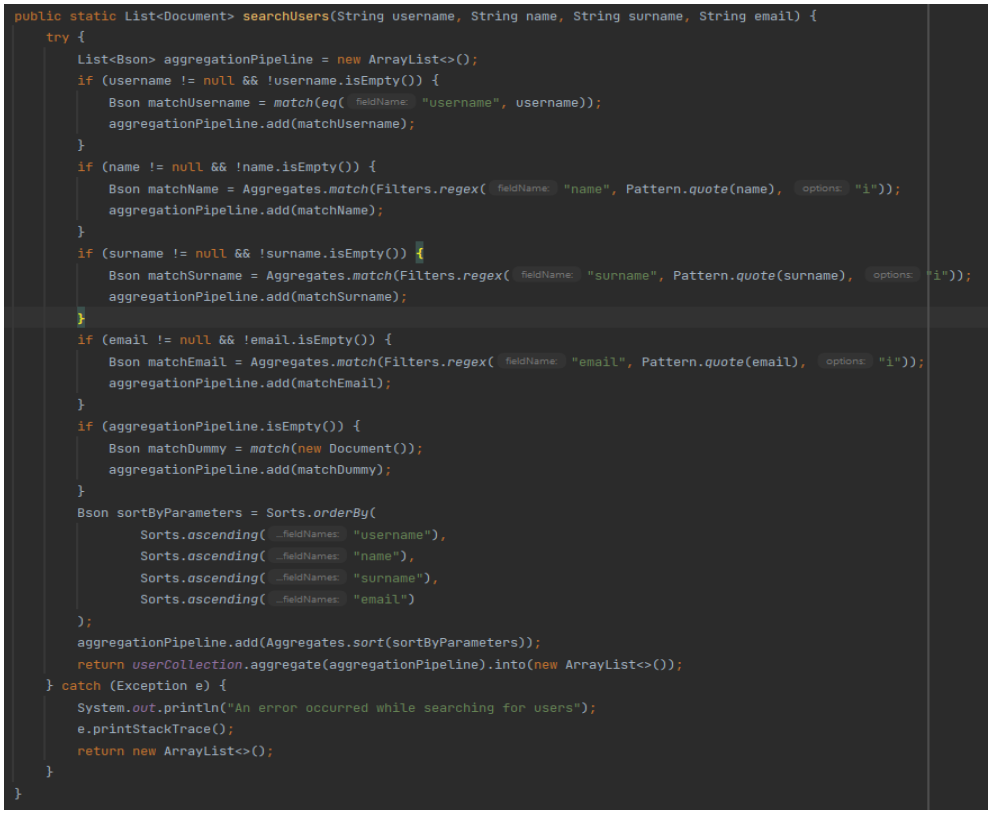
- Search for Restaurants given parameters.



```
public static List<Document> searchRestaurants(String location, String name, String cuisine, String keywords, String rating) {
    try {
        List<Bson> aggregationPipeline = new ArrayList<>();

        if (location != null && !location.isEmpty()) {
            String[] locationArray = location.split( regex: ",");
            Bson matchLocation = Aggregates.match(Filters.in( fieldName: "location", Arrays.asList(locationArray)));
            aggregationPipeline.add(matchLocation);
        }
        if (name != null && !name.isEmpty()) {
            Bson matchName = Aggregates.match(Filters.regex( fieldName: "restaurant_name", Pattern.quote(name),  options: "i"));
            aggregationPipeline.add(matchName);
        }
        if (cuisine != null && !cuisine.isEmpty()) {
            String[] cusineArray = cuisine.split( regex: ",");
            Bson matchCuisine = Aggregates.match(Filters.in( fieldName: "tag", Arrays.asList(cusineArray)));
            aggregationPipeline.add(matchCuisine);
        }
        if (keywords != null && !keywords.isEmpty()) {
            String[] keywordArray = keywords.split( regex: ",");
            Bson matchKeywords = Aggregates.match(Filters.in( fieldName: "tag", Arrays.asList(keywordArray)));
            aggregationPipeline.add(matchKeywords);
        }
        if (rating != null && !rating.isEmpty()) {
            double minRating = Double.parseDouble(rating);
            Bson matchRating = Aggregates.match(Filters.gte( fieldName: "rest_rating", minRating));
            aggregationPipeline.add(matchRating);
        }
        if (location != null && !location.isEmpty() || name != null && !name.isEmpty() || cuisine != null && !cuisine.isEmpty() || keywords != null && !keywords.isEmpty()) {
            Bson addTotalReviews = Aggregates.addFields(
                    new Field<>( name: "totalReviews", new Document("$size", "$reviews"))
            );
            aggregationPipeline.add(addTotalReviews);
        }
        if (aggregationPipeline.isEmpty()) {
            Bson matchDummy = Aggregates.match(new Document());
            aggregationPipeline.add(matchDummy);
        }
```

- Search for Users given parameters.



```
public static List<Document> searchUsers(String username, String name, String surname, String email) {
    try {
        List<Bson> aggregationPipeline = new ArrayList<>();
        if (username != null && !username.isEmpty()) {
            Bson matchUsername = match(eq( fieldName: "username", username));
            aggregationPipeline.add(matchUsername);
        }
        if (name != null && !name.isEmpty()) {
            Bson matchName = Aggregates.match(Filters.regex( fieldName: "name", Pattern.quote(name),  options: "i"));
            aggregationPipeline.add(matchName);
        }
        if (surname != null && !surname.isEmpty()) {
            Bson matchSurname = Aggregates.match(Filters.regex( fieldName: "surname", Pattern.quote(surname),  options: "i"));
            aggregationPipeline.add(matchSurname);
        }
        if (email != null && !email.isEmpty()) {
            Bson matchEmail = Aggregates.match(Filters.regex( fieldName: "email", Pattern.quote(email),  options: "i"));
            aggregationPipeline.add(matchEmail);
        }
        if (aggregationPipeline.isEmpty()) {
            Bson matchDummy = match(new Document());
            aggregationPipeline.add(matchDummy);
        }
        Bson sortByParameters = Sorts.orderBy(
                Sorts.ascending( ...fieldNames: "username"),
                Sorts.ascending( ...fieldNames: "name"),
                Sorts.ascending( ...fieldNames: "surname"),
                Sorts.ascending( ...fieldNames: "email")
        );
        aggregationPipeline.add(Aggregates.sort(sortByParameters));
        return userCollection.aggregate(aggregationPipeline).into(new ArrayList<>());
    } catch (Exception e) {
        System.out.println("An error occurred while searching for users");
        e.printStackTrace();
        return new ArrayList<>();
    }
}
```

These are the analytics queries that have been implemented:

- Analytics: See k most active Users.

```
public static List<Document> getUsersWithMostReviews(int k) {
    List<Document> userList = new ArrayList<>();

    AggregateIterable<Document> result = restaurantCollection.aggregate(Arrays.asList(
            new Document("$unwind", "$reviews"),
            new Document("$group", new Document("_id", "$reviews.reviewer_pseudo")
                    .append("totalReviews", new Document("$sum", 1L))),
            new Document("$sort", new Document("totalReviews", -1L)),
            new Document("$limit", k)
    ));

    for (Document document : result) {
        userList.add(document);
    }

    return userList;
}
```

- Analytics: See k Restaurants with highest lifespan (given by the milliseconds passed between the first and the last Reviews).

```
public static List<Document> getHighestLifespanRestaurants(int k){
    List<Document> restaurantList = new ArrayList<>();

    AggregateIterable<Document> result = restaurantCollection.aggregate(Arrays.asList(new Document("$unwind", "$reviews"),
            new Document("$group",
                    new Document("_id", "$_id")
                            .append("rest_id",
                                    new Document("$first", "$rest_id"))
                            .append("minDate",
                                    new Document("$min",
                                            new Document("$toDate", "$reviews.review_date")))
                            .append("maxDate",
                                    new Document("$max",
                                            new Document("$toDate", "$reviews.review_date")))),
            new Document("$project",
                    new Document("_id", 1L)
                            .append("rest_id", 1L)
                            .append("minDate",
                                    new Document("$dateToString",
                                            new Document("format", "%Y-%m-%d")
                                                    .append("date", "$minDate")))
                            .append("maxDate",
                                    new Document("$dateToString",
                                            new Document("format", "%Y-%m-%d")
                                                    .append("date", "$maxDate")))
                            .append("lifespan",
                                    new Document("$divide", Arrays.asList(new Document("$subtract", Arrays.asList("$maxDate", "$minDate")), 24L * 60L * 60L * 1000L)))),
            new Document("$sort",
                    new Document("lifespan", -1L)),
            new Document("$limit", k)
    ));

    for (Document document : result) {
        restaurantList.add(document);
    }

    return restaurantList;
```

- Analytics: See k top Restaurants with the highest number of 5 stars Reviews given a cuisine

```java
public static List<Document> getTopKRatedRestaurantsByCuisine(int k, String cuisine) {
    try {
        List<Bson> aggregationPipeline = new ArrayList<>();

        if (cuisine != null && !cuisine.isEmpty()) {
            Bson matchCuisine = Aggregates.match(Filters.in( fieldName: "tag", cuisine));
            aggregationPipeline.add(matchCuisine);
        }

        Bson matchRating = Aggregates.match(Filters.elemMatch( fieldName: "reviews", Filters.eq( fieldName: "review_rating",  value: 5)));
        aggregationPipeline.add(matchRating);

        Bson addTotalReviews = Aggregates.addFields(
                new Field<>( name: "totalReviews", new Document("$size", "$reviews"))
        );
        aggregationPipeline.add(addTotalReviews);

        Bson sortTotalReviews = Aggregates.sort(Sorts.descending( ...fieldNames: "totalReviews"));
        aggregationPipeline.add(sortTotalReviews);

        Bson limitResults = Aggregates.limit(k);
        aggregationPipeline.add(limitResults);

        return restaurantCollection.aggregate(aggregationPipeline).into(new ArrayList<>());
    } catch (Exception e) {
        System.out.println("An error occurred while retrieving top-rated Italian restaurants");
        e.printStackTrace();
        return new ArrayList<>();
    }
}
```

- Analytics: Get the total number of reviews and the average rating for a given period

```java
public static Document getReviewsStatsByDateRange(String restId, String startDate, String endDate){

    AggregateIterable<Document> result = restaurantCollection.aggregate(Arrays.asList(new Document("$match",
                    new Document("rest_id", restId)),
            new Document("$unwind", "$reviews"),
            new Document("$match",
                    new Document("$expr",
                            new Document("$and", Arrays.asList(new Document("$gte", Arrays.asList(new Document("$dateFromString",
                                            new Document("dateString", endDate)
                                                    .append("format", "%Y-%m-%d")),
                                    new Document("$dateFromString",
                                            new Document("dateString", "$reviews.review_date")
                                                    .append("format", "%Y-%m-%d")))),
                                    new Document("$lt", Arrays.asList(new Document("$dateFromString",
                                            new Document("dateString", startDate)
                                                    .append("format", "%Y-%m-%d")),
                                    new Document("$dateFromString",
                                            new Document("dateString", "$reviews.review_date")
                                                    .append("format", "%Y-%m-%d"))))))),
            new Document("$group",
                    new Document("_id",
                            new BsonNull())
                            .append("total_reviews",
                                    new Document("$sum", 1L))
                            .append("average_rating",
                                    new Document("$avg", "$reviews.review_rating"))),
            new Document("$project",
                    new Document("total_reviews", 1L)
                            .append("average_rating",
                                    new Document("$round", Arrays.asList("$average_rating", 1L))))
    ));

    return result.first();
}
```

4.2.2. Neo4j – queries and analytics

These are the main CRUD queries that have been implemented:

- Create User
- Create RestaurantList
- Create Restaurant
- Delete User
- Delete RestaurantList
- Delete Restaurant

These are queries that have been implemented exploiting the relationships between nodes:

- Get the number of likes of a given Restaurant:

```
"MATCH (:Restaurant{rest_id: $rest_id})<-[r:Likes]-()" +
 "RETURN COUNT(r) AS numLikes"
```

- Get the number of followers of a given User:

```
"MATCH (:User {username: $username})<-[r:Follows]-() " +
 "RETURN COUNT(r) AS numFollowers"
```

- Get the number of followers of a given Restaurant List:

```
"MATCH (:RestaurantList {title: $title, owner: $owner})<-[r:Follows]-() " +
 "RETURN COUNT(r) AS numFollowers"
```

- Get the number of users that a given User follows:

```
"MATCH (:User {username: $username})-[r:Follows]->()

RETURN COUNT(r) AS numFollowers"
```

- Check if a given User likes a given Restaurant:

```
"MATCH (:User{username:$username})-[r:Likes]->(p:Restaurant)

WHERE p.rest_id = $rest_id " +
"RETURN COUNT(*)"
```

- Check if a given User likes a given Restaurant List:

```
"MATCH (a:User{username:$user})-[r:Follows]->(b:RestaurantList{title:$title, owner:$owner }) " +
 "RETURN COUNT(*)"
```

- Check if a given User A follows a given User B

```
"MATCH (a:User{username:$usernameA})-[r:Follows]->(b:User{username:$usernameB}) " +
 "RETURN COUNT(*)"
```

- Check if a given User have put a like to a given Restaurant

```
"MATCH (a:User), (b:Restaurant) " +
 "WHERE a.username = $username AND b.rest_id = $rest_id " +
 "MERGE (a)-[r:Likes]->(b)"
```

- A given User removes a like to a given Restaurant

```
"MATCH (a:User{username:$username})-[r:Likes]->(b:Restaurant) " +
 "WHERE b.rest_id = $rest_id DELETE r"
```

- A given User A start to follow a given User B

```
"MATCH (u1:User {username: $usernameA}), (u2:User {username: $usernameB}) " +
 "CREATE (u1)-[:Follows]->(u2)"
```

- A given User A stop to follow a given User B

```
"MATCH (:User {username: $usernameA})-[r:Follows]->(:User {username: $usernameB}) " +
 "DELETE r"
```

- A given User start to follow a given Restaurant List

```
"MATCH (u1:User {username: $username}), (rl:RestaurantList {title: $title, owner: $owner}) " +
 "CREATE (u1)-[:Follows]->(rl)"
```

- A given User stop to follow a given Restaurant List

```
MATCH (:User {username: $username})-[r:Follows]->(:RestaurantList {title: $title, owner: $owner}) " +
 "DELETE r"
```

- Search a Restaurant List given parameters

```
MATCH (rl:RestaurantList)
WHERE toLower(rl.title) CONTAINS toLower($searchTitle)
OPTIONAL MATCH (rl)<-[r:Follows]-()
RETURN rl, COUNT(r) AS numFollowers
ORDER BY numFollowers DESC"""
```

These are the analytics queries that have been implemented:

- Get the Most Liked Restaurants:

```
MATCH (r:Restaurant) <- [l:Likes]-()
RETURN r, COUNT(l) AS numLikes
ORDER BY numLikes DESC
LIMIT $limit"""
```

- Get the Most Followed Restaurant Lists:

```
MATCH (r:RestaurantList) <- [l:Follows]-()
RETURN r, COUNT(l) AS numFollowers
ORDER BY numFollowers DESC
LIMIT $limit"""
```

- Get the Most Followed Users

```
MATCH (u:User) <- [l:Follows]-()
RETURN u, COUNT(l) AS numFollowers
ORDER BY numFollowers DESC
LIMIT $limit"""
```

- Get Suggested Restaurants: the final results are the union of the results of three different queries. The first one returns the restaurants liked by users who liked the same restaurants of the logged user which are not already liked by him. The third query returns the most liked restaurants. The third query is useful especially when a user doesn't have any connections yet for example a just registered User.

```
MATCH (me:User {username: $username})-[:Likes]->(r1:Restaurant)<-[:Likes]-(other:User)-[:Likes]-
>(similarRestaurant:Restaurant)
WHERE NOT EXISTS((me)-[:Likes]->(similarRestaurant))
AND other <> me
WITH DISTINCT similarRestaurant, COUNT(DISTINCT other) AS similarityScore
RETURN similarRestaurant AS r, similarityScore AS recommendationScore
ORDER BY recommendationScore DESC
LIMIT $limit
UNION
MATCH (me:User {username: $username})-[:Follows]->(other:User)-[:Likes]->(r2:Restaurant)
WHERE NOT EXISTS{
                        MATCH (me:User {username: $username})-[:Likes]->(r1:Restaurant)<-[:Likes]-
                        (other:User)-[:Likes]->(similarRestaurant:Restaurant)
                        where r2 = similarRestaurant
                }
RETURN DISTINCT r2 as r, 0 AS recommendationScore
LIMIT $limit
UNION
MATCH (r:Restaurant) <- [l:Likes]-()
RETURN r, COUNT(l) AS recommendationScore
ORDER BY recommendationScore DESC
LIMIT $limit"""
```

- Get Suggested Restaurant Lists: the final results are the union of the results of two different queries. The first query returns the restaurant list followed by followed users of the logged user which are not already followed by him. The second query returns the most followed restaurant list. The second query is useful especially when a user doesn't have any connections yet for example a just registered User.

```
MATCH (me:User {username: $username})-[:Follows]->(:User)-[:Follows]->(rl:RestaurantList)
WHERE NOT EXISTS {
 MATCH (me)-[:Follows]->(rl)}
WITH rl, COUNT(DISTINCT me) AS followCount
RETURN rl, followCount AS recommendationScore
ORDER BY recommendationScore DESC
LIMIT $limit
UNION
MATCH (rl:RestaurantList) <- [l:Follows]-()
```

```
WITH rl, COUNT(l) as numFollowers
RETURN rl, numFollowers AS recommendationScore
ORDER BY numFollowers DESC
LIMIT $limit"""
```

- Get Suggested Users: the final results are the union of the results of two different queries. The first query returns the Users followed by followed users of the logged user which are not already followed by him. The second query returns the most followed Users. The second query is useful especially when a user doesn't have any connections yet for example a just registered User.

```
MATCH (me:User {username: $username})-[:Follows]->(:User)-[:Follows]->(u:User)
WHERE NOT EXISTS ((me)-[:Follows]->(u))
AND u<>me
WITH u, COUNT(DISTINCT me) AS followCount
RETURN u, followCount AS recommendationScore
ORDER BY recommendationScore DESC
LIMIT $limit
UNION
MATCH (u:User) <- [l:Follows]-()
WITH u, COUNT(l) as numFollowers
RETURN u, numFollowers AS recommendationScore
ORDER BY numFollowers DESC
LIMIT $limit"""
```

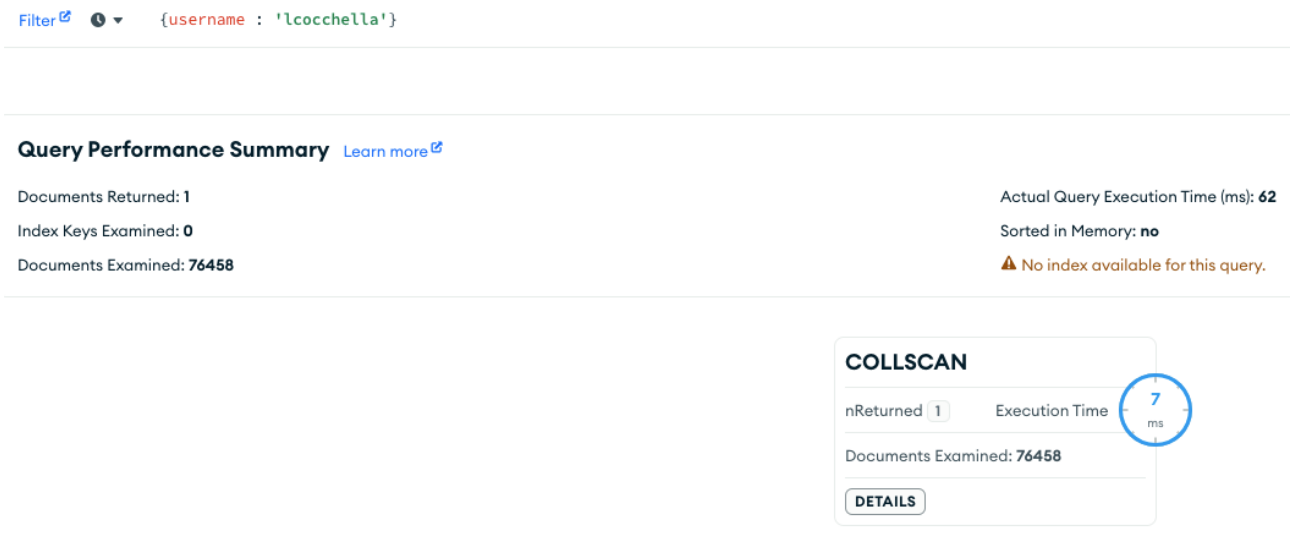
4.2.3. MongoDB Indexes

Not using indexes on a document database can lead to scan the entire database when performing queries. This means that every document needs to be inspected to find the desired results, implying potentially slow and inefficient queries. However, by creating simple indexes on relevant parameters, the database can narrow down the search to only the relevant documents, significantly improving query performance and speeding things up.

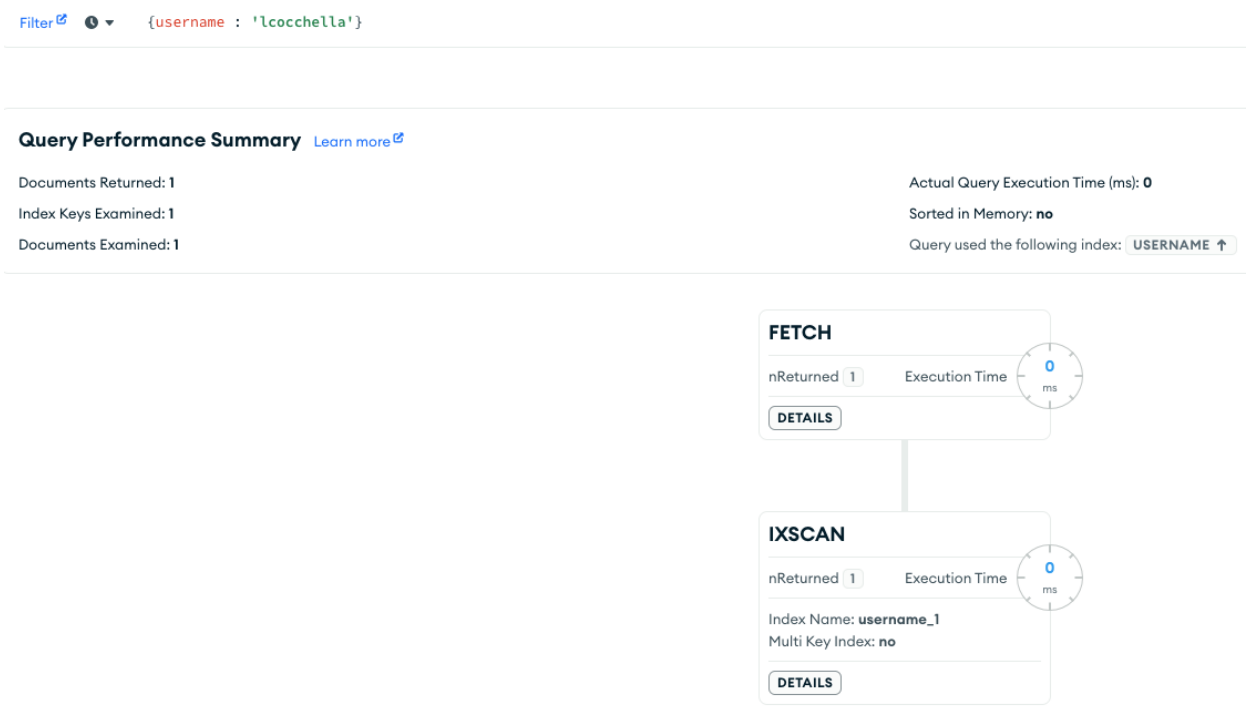
4.2.3.1. Index on the user

Username

Searching the username 'lcocchella' without the index imply searching in the whole database and it takes 7 ms to complete the operation.



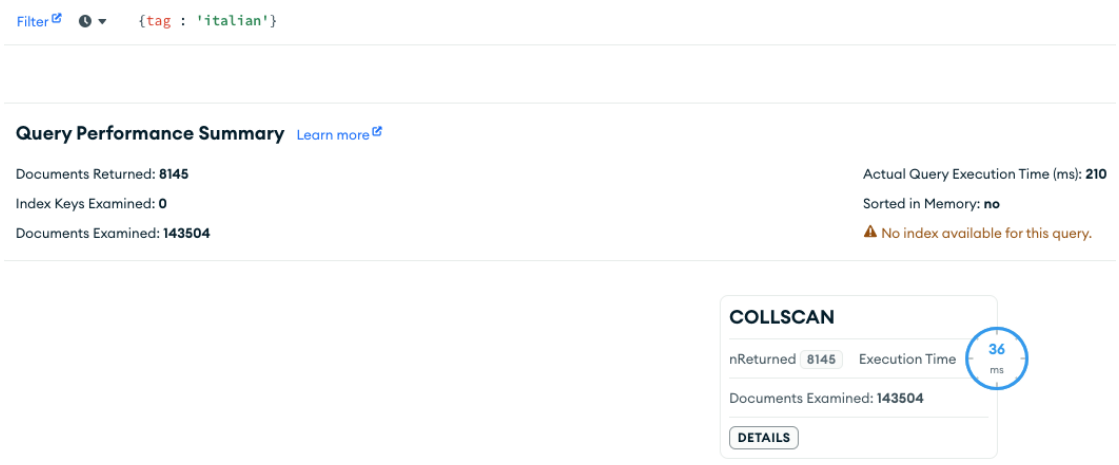
By adding the index on the username, we can see that it improves the performance searching directly that document. In this way we reach the results in 0ms.



4.2.3.2. Index on the Restaurant

Searching for a restaurant in the whole database takes a bit of time, we can improve this search by creating indexes to achieve better performance.
Here some examples of index usage on Tag, Reservation date, Location and Restaurant Name:

Tag

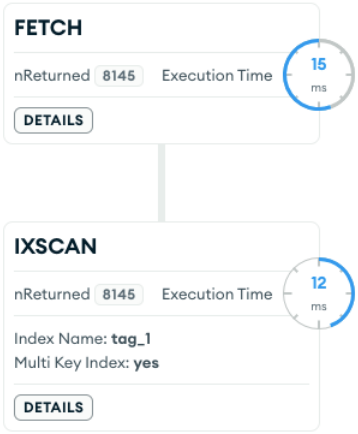


Filter {tag : "italian"}

Query Performance Summary Learn more

Documents Returned: **8145**
Index Keys Examined: **8145**
Documents Examined: **8145**

Actual Query Execution Time (ms): **57**
Sorted in Memory: **no**
Query used the following index: TAG



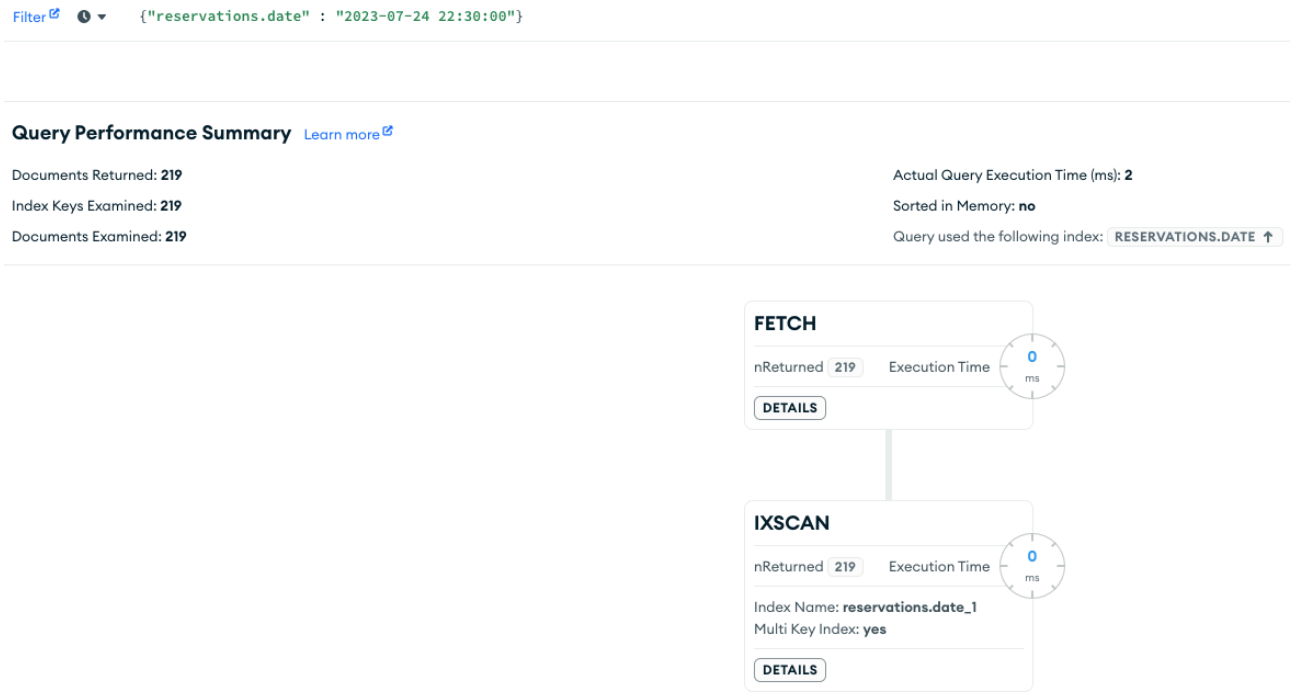
Reservation date

Filter {"reservations.date" : "2023-07-24 22:30:00"}

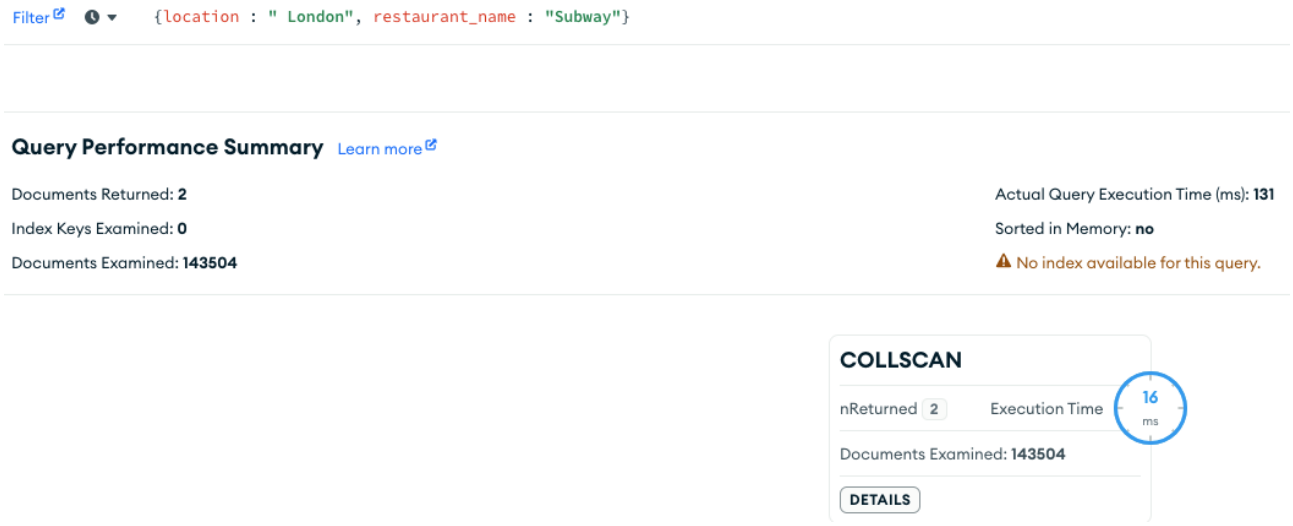
Query Performance Summary Learn more

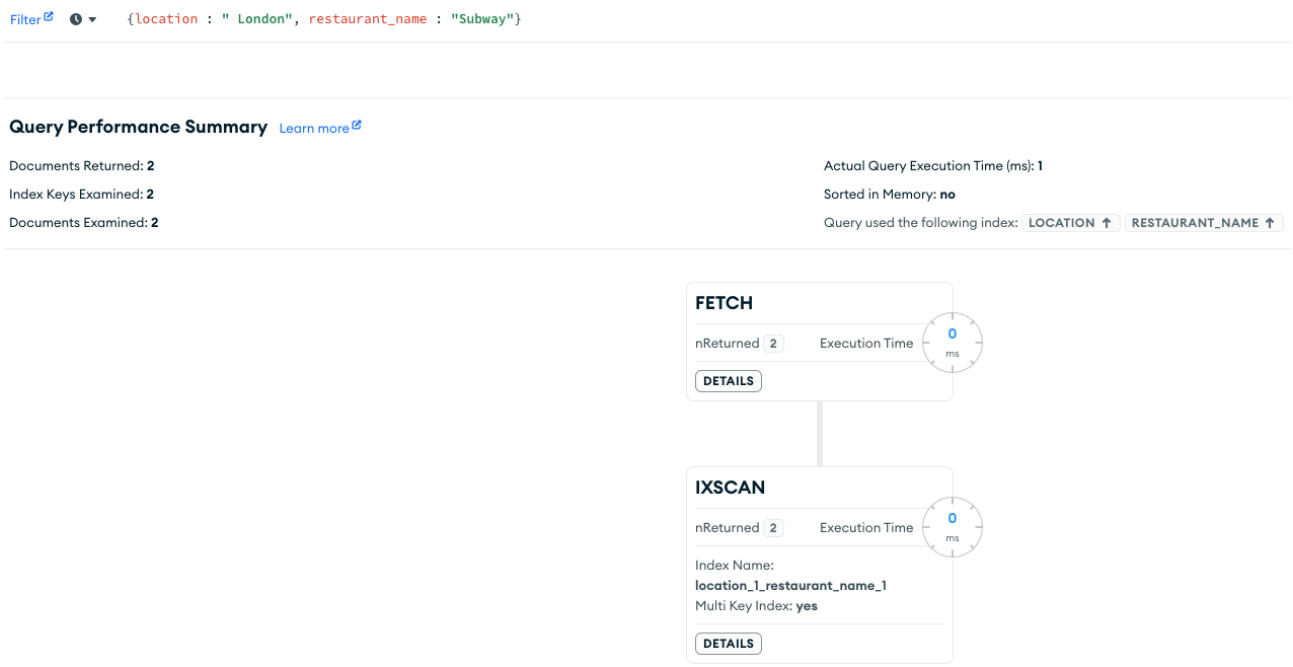
Documents Returned: **219**
Index Keys Examined: **0**
Documents Examined: **143501**

Actual Query Execution Time (ms): **280**
Sorted in Memory: **no**
⚠ No index available for this query.



Index compound on location and name





Here are some examples of queries in which we use the indexes:

1. Username of User

QUERY	DESCRIPTION
loginUser	The login is made with username and password
searchUser	The search can be made with username, name, surname, and email
getRestaurantListByUser	It retrieves the restaurantLists associated with the username
getReservations	Retrieves the reservation made by a specific username

2. Tags

QUERY	DESCRIPTION
searchRestaurants	The search includes cuisine and keyworks (they are both found within the tags)

getTopKRatedRestaurantsByCuisine	The cuisines are inside the tags

3. Reservation Date

QUERY	DESCRIPTION
makeLocalReservation	Make a reservation in a specific restaurant on a specific date (yyyy-MM-dd hh:mm:ss)
getEmptySeatsByDate	Returns the number of empty seats given a restaurant and a date
deleteFreeSlots	Remove slots of a given date in a given restaurant

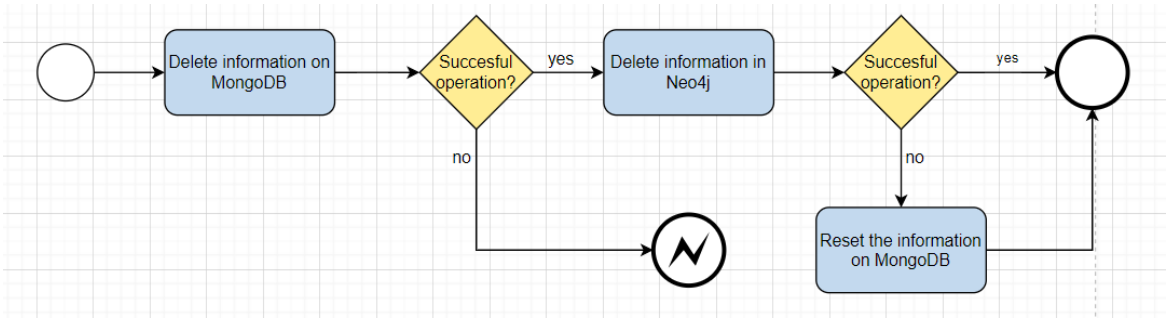
4. Location and name

QUERY	DESCRIPTION
serachRestaurants	The search includes both name and location of a restauant

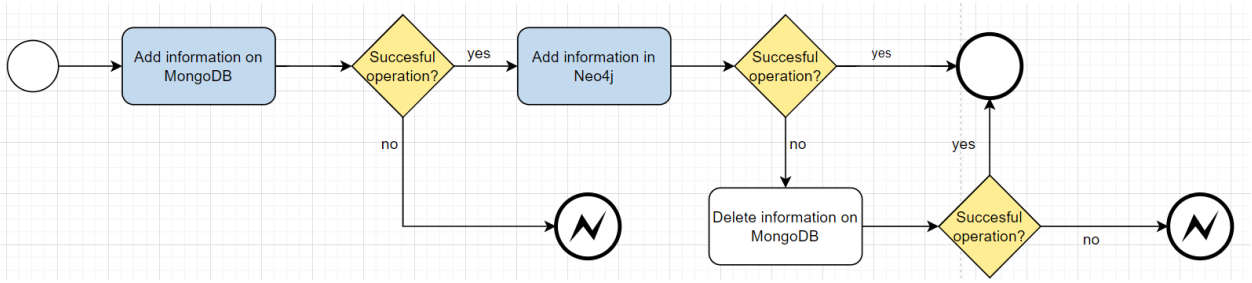
4.2.4. Cross-Database Consistency Management

We have opted to maintain data in two distinct databases, resulting in the duplication of certain information. As a result, it has become imperative to effectively manage any potential discrepancies in the data. Specifically, this circumstance arises exclusively during operations involving data present in both MongoDB and Neo4j. This occurs in situations such as:

- Delete Restaurant or User



- Add Restaurant or User



5. Database Properties

5.1. Redundancies and reservation handling

The redundancy issue we encounter primarily revolves around reservations. As previously mentioned, while discussing the collections, reservation documents are present in both the Restaurants and the Users collections. Instead of opting for a separate collection exclusively for reservations, we carefully considered the application's query requirements before deciding.

Creating a dedicated collection for reservations would inevitably lead to significant growth as the number of registered restaurants increases. This is mainly because restaurants can add numerous empty slots, for future reservations. Consequently, retrieving and examining all reservations for a particular restaurant —a frequent operation—would involve scanning through a substantial number of documents.

To address this concern and optimize the efficiency of retrieving a restaurant's reservation, we decided to embed the reservations within the corresponding restaurant's document. Similarly, we included reservations within the user document to enable users to retrieve all their reservations without having to search through the reservations of every restaurant.

While this approach necessitates a double write operation for each booking or deletion, we anticipate that individual users will typically have a limited number of reservations. Hence, the impact of the double write operation is expected to be minor compared to the number of times users read their own reservations. Additionally, given the regularity with which restaurant owners check their reservation lists, this nesting approach significantly enhances overall performance.

In addition to duplicating certain information about restaurants and users in our system, we have also stored them in a Graph Database. However, it's important to note that we selectively include only the attributes that are essential for executing the queries effectively.

Specifically, when it comes to users, we store their name, surname, and username in the Graph Database as it is crucial for establishing relationships and performing relevant queries. As for restaurants, we store their ids. These attributes are carefully chosen based on the specific needs of our query operations, ensuring optimal query execution and retrieval of the necessary information.

By storing these key attributes in the Graph Database, we can leverage the power of graph querying to efficiently navigate relationships and retrieve relevant data. This selective inclusion of

attributes helps us streamline the querying process, enhancing the overall performance and effectiveness of our system.

5.2. Consistency, Availability and Partition Tolerance

Ensuring high availability is an essential non-functional requirement of the application. To achieve this, a cluster of replica sets is employed, hosted across three virtual machines.

The primary objective of this cluster is to guarantee partition tolerance, which means that even if a single node fails or experiences issues, the overall system should not collapse. The cluster is designed to sustain such failures and continue functioning seamlessly, maintaining the application's availability at all times.

The replicas for neo4j Database were not deployed on the virtual cluster, but only a single instance is present in the virtual machine at the address 10.1.1.18:7687.

By implementing sharding in the database, we can significantly decrease the latency and response time for server interactions. To achieve this goal, we have decided to partition the document database into shards, with each shard hosted on a separate server. This approach, combined with the existing data replication, enables effective load balancing, ensuring the Non-Functional Requirements.

To implement the sharding mechanism, we have decided to employ distinct sharding keys and partitioning methods for each collection within our document database.

Specifically, we will select two different sharding keys, one for the Users Collection and the other for the Restaurants Collection. This partitioning strategy will enable efficient distribution and management of data across the shards, optimizing query performance and resource utilization.

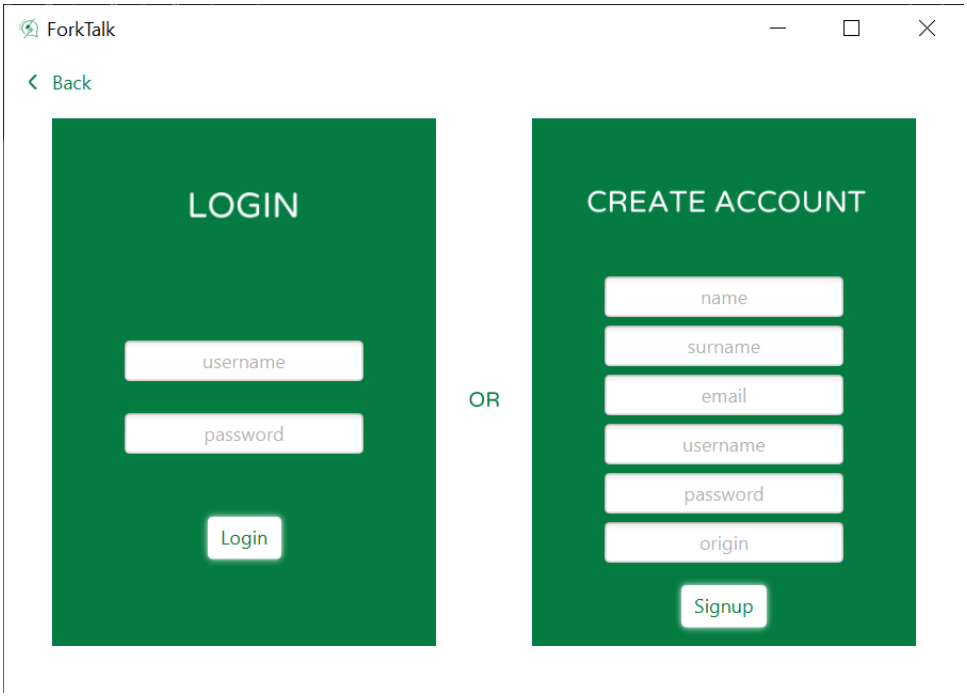
- *Users collection*: To effectively shard the Users Collection, we have selected the "username" attribute as the sharding key. To achieve optimal load balancing and minimize response time, we have chosen a List as the partitioning method. With the List partitioning method, we will map each username to a specific server within the cluster. This mapping process will be designed in a manner that evenly distributes the Users data across the available servers, ensuring a balanced load distribution.

- *Restaurant collection*: To shard the Restaurant Collection, we have determined that the "rest_id" attribute will serve as the sharding key. For the partitioning method, we have opted for the Hashing. This approach involves applying a hash function to the "rest_id" values to determine in which shard each restaurant's data will go. Hashing provides a balanced distribution of data across different servers, helping to evenly distribute the load and improve

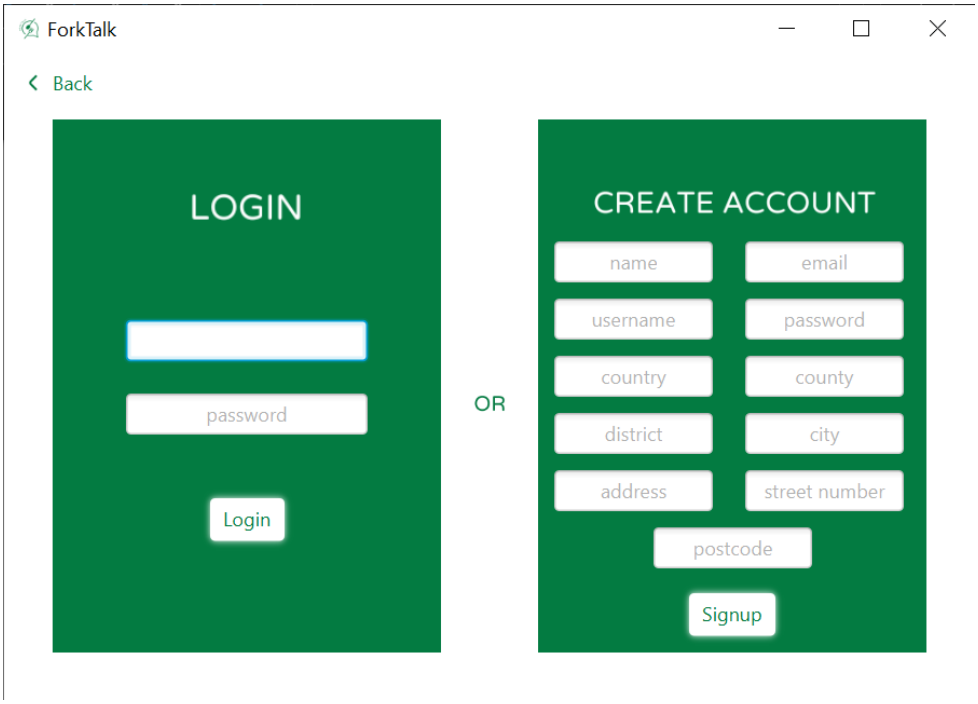
performance. Additionally, to handle scenarios where a server becomes unavailable or new servers are added to the cluster, we plan to incorporate Consistent Hashing.

6. User Manual

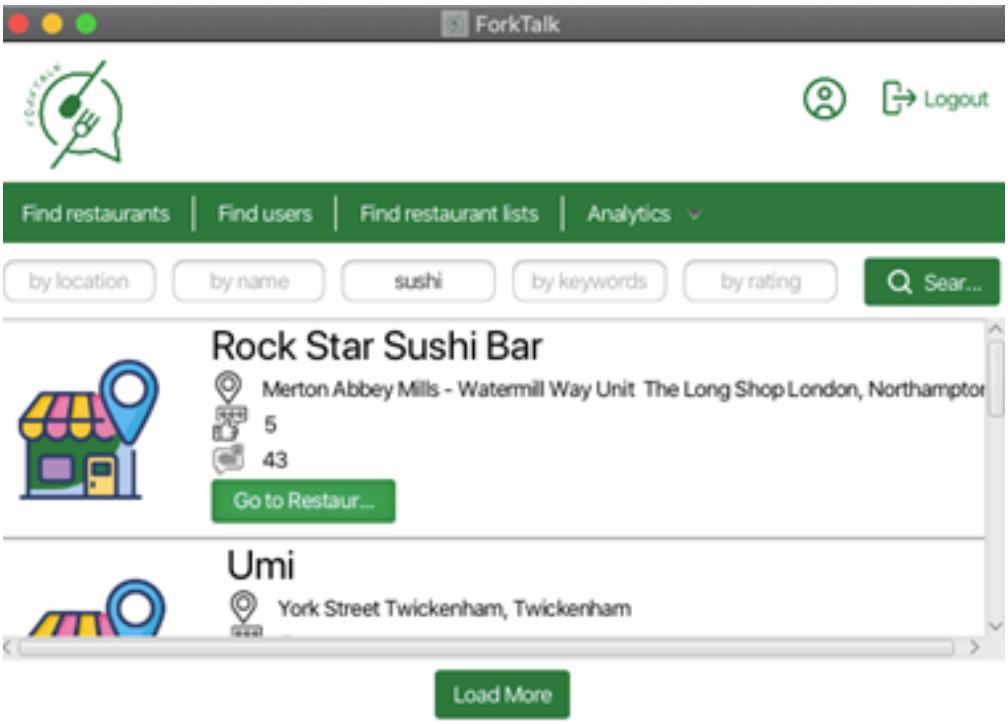
- Login/Register use page



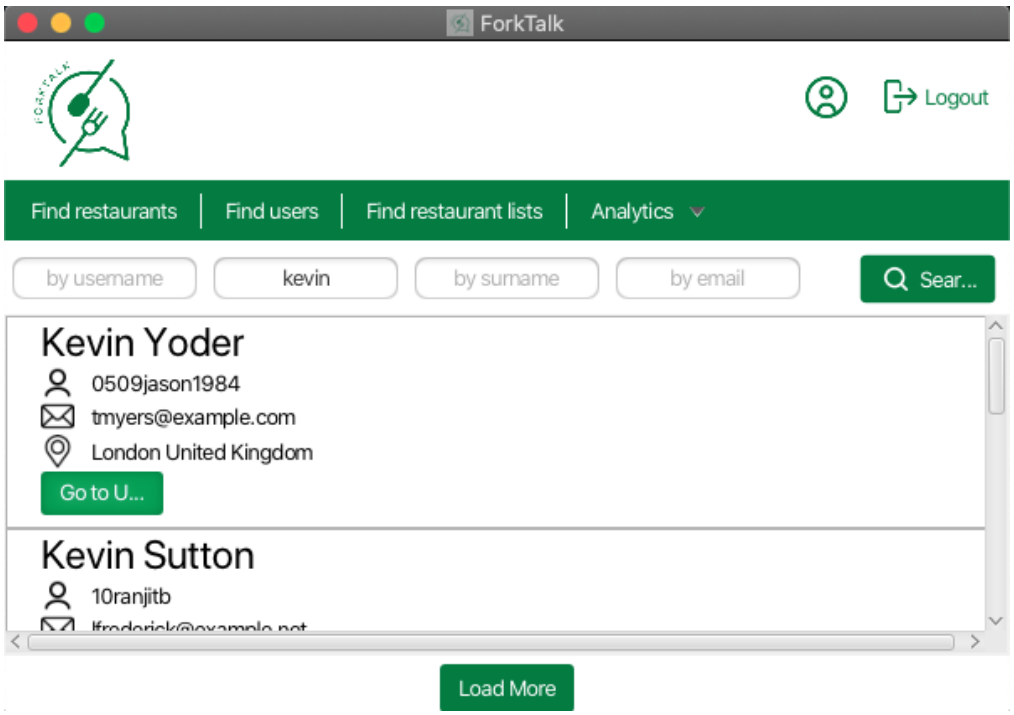
- Login/Register restaurant page



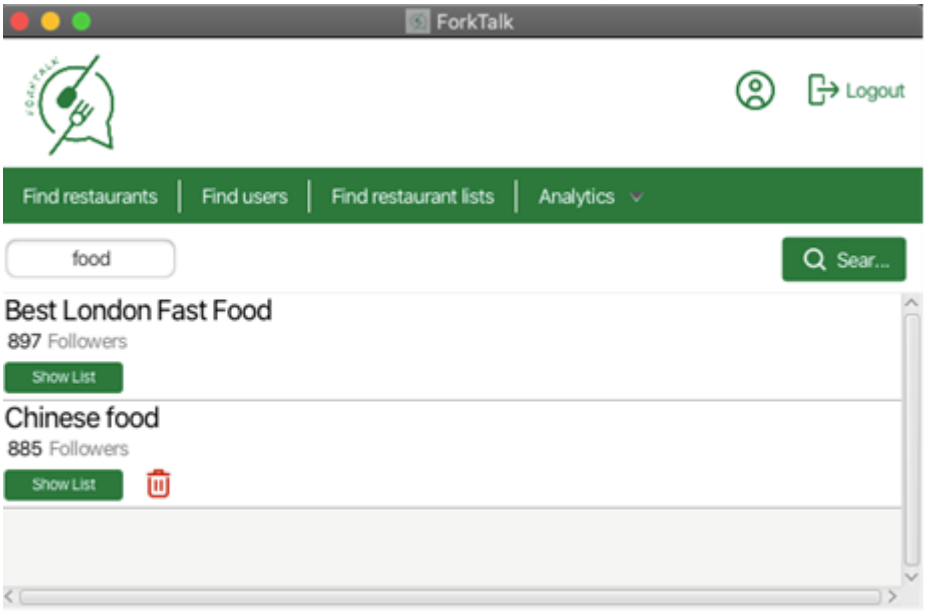
- Logged user -> find restaurant -> by cusine



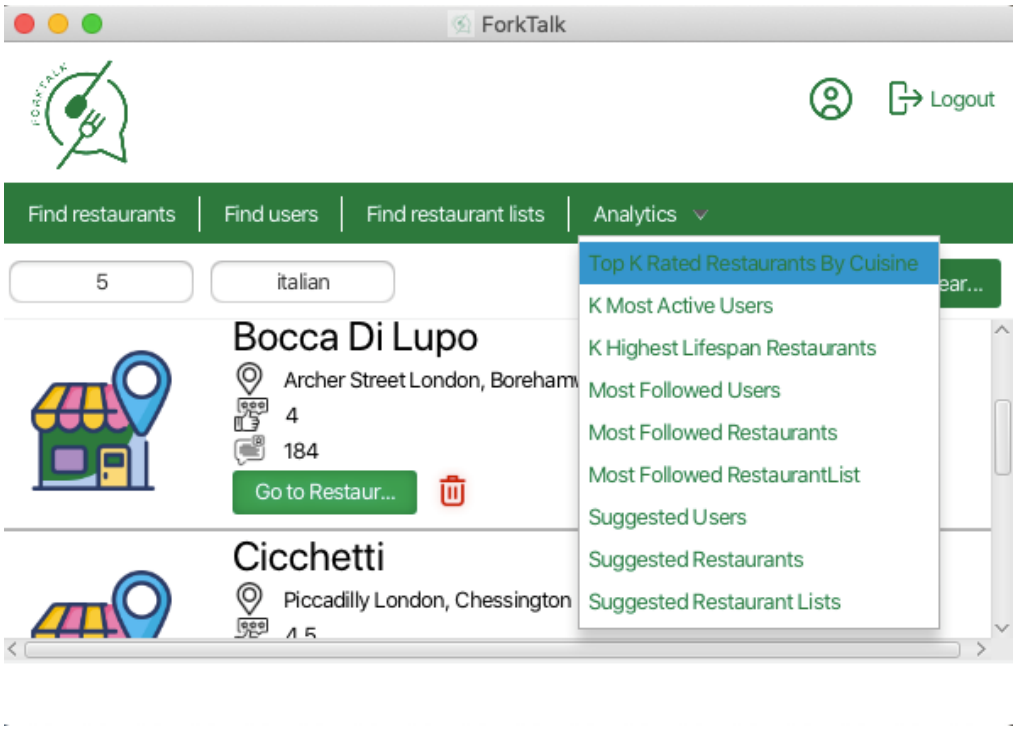
- Logged user -> find user -> by name



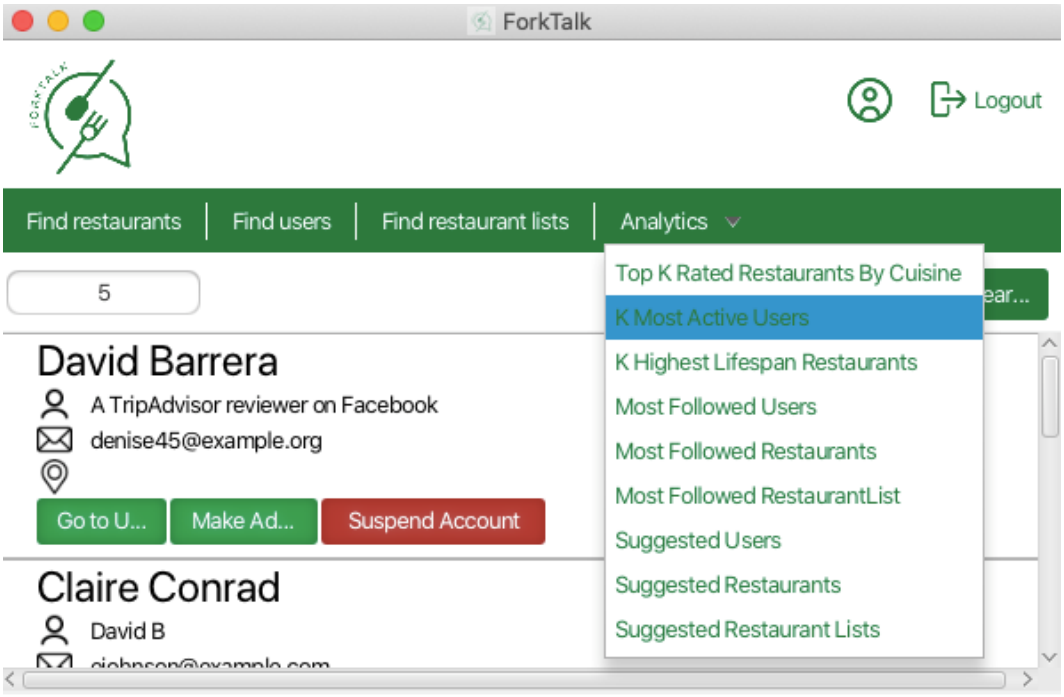
- Logged user -> find restaurantList -> by title



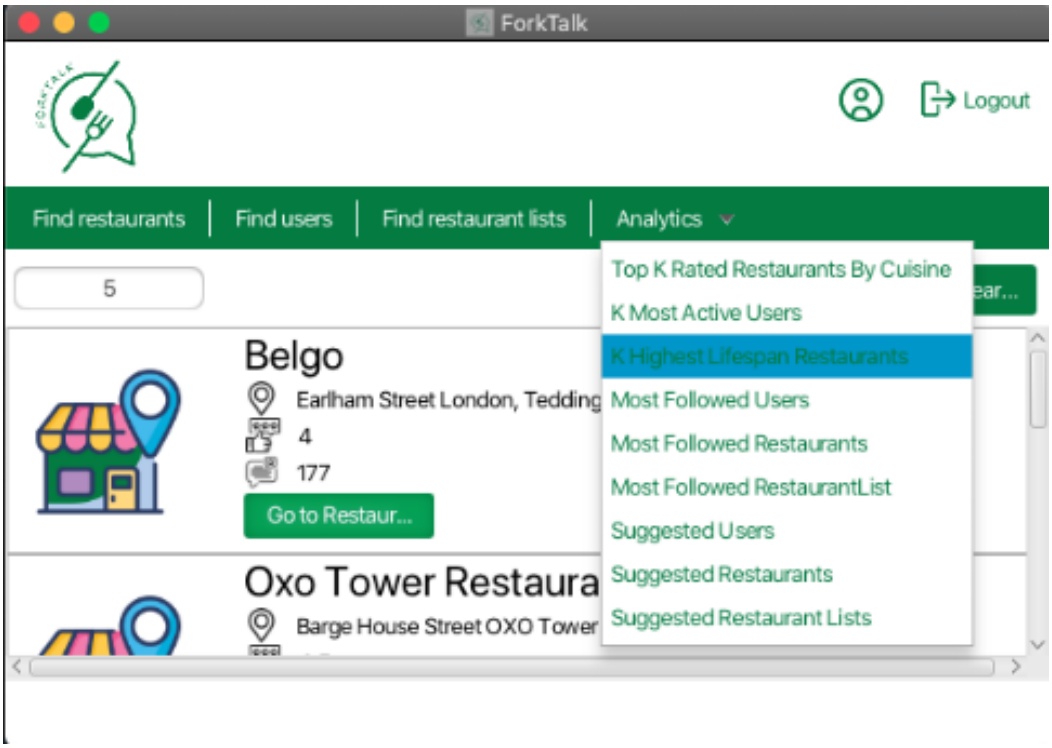
- ANALYTICS:
  - Find the top K rated Restaurants by cuisine



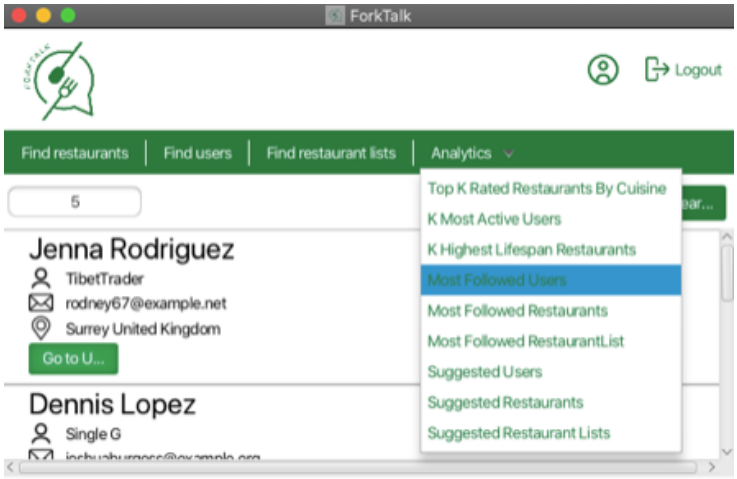
- Find the K Most ActiveUsers



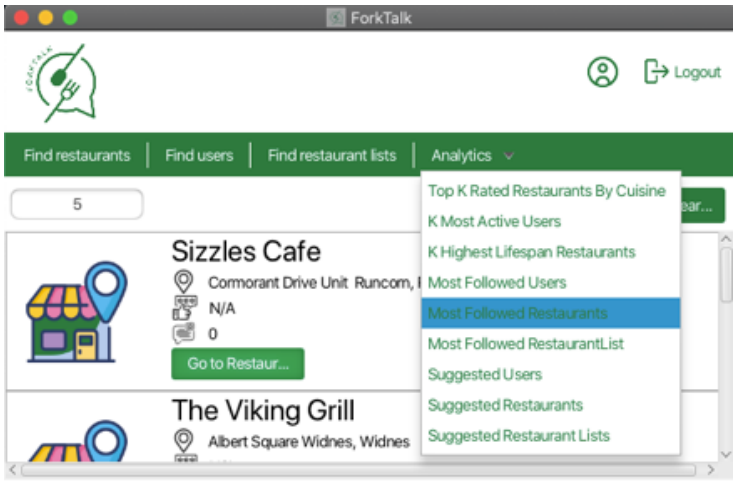
- Find the k highest Lifespan Restaurants



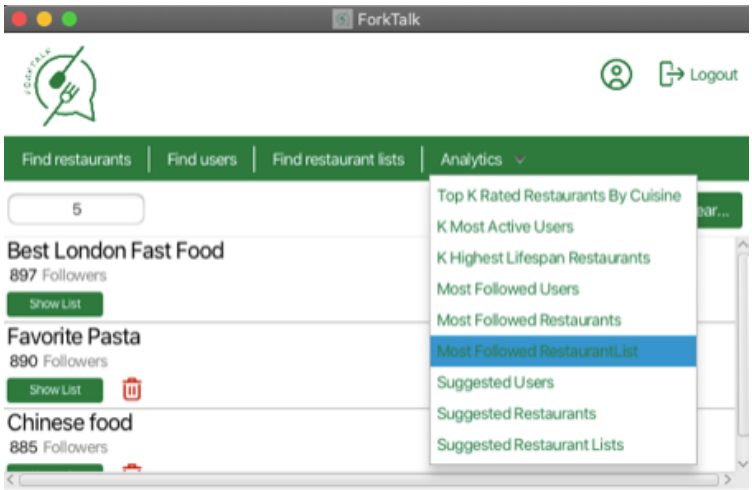
- Find the Most Followed Users



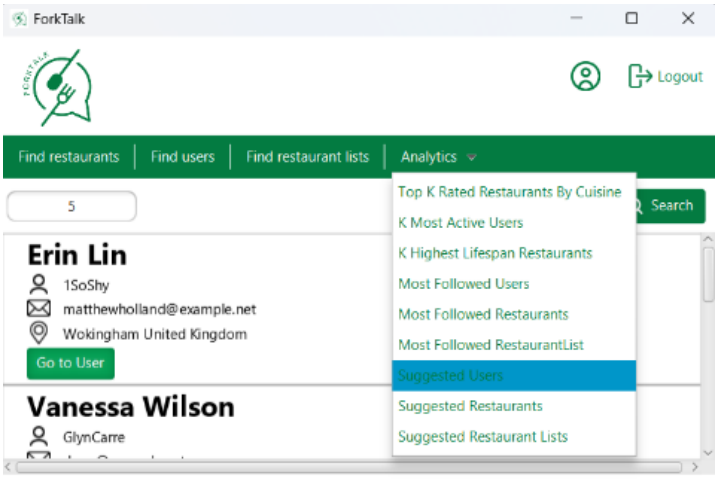
- Find the most followed Restaurants



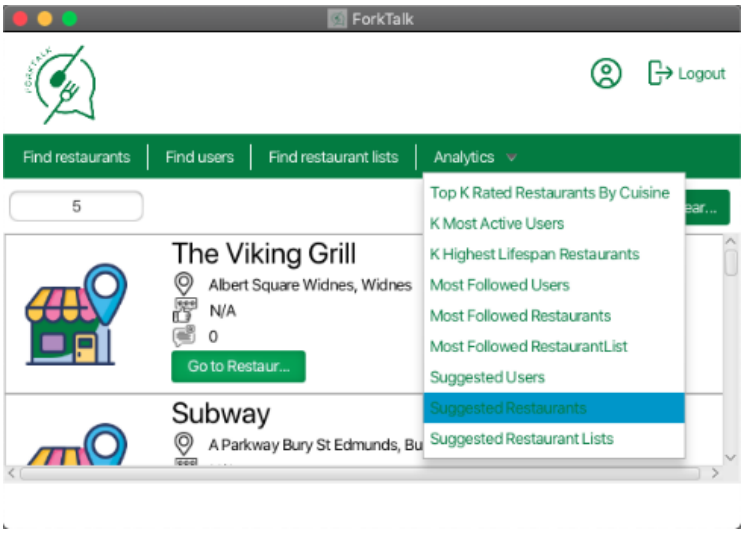
- Find the most followed RestaurantList



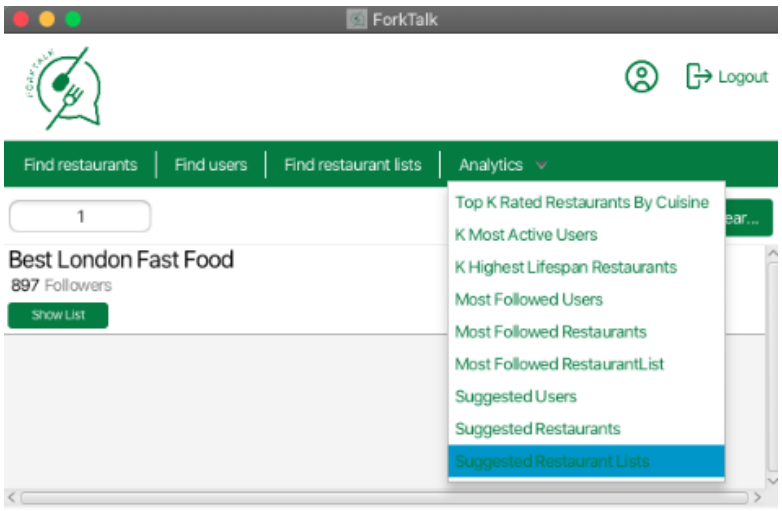
- Find the suggested Users



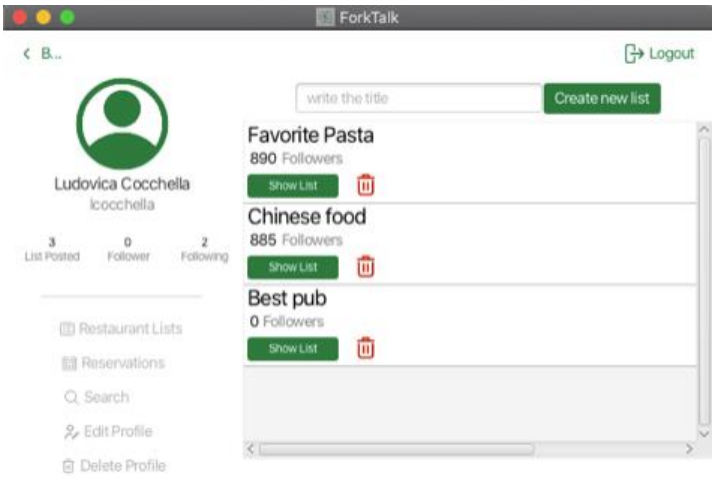
- Find the suggested Restaurants



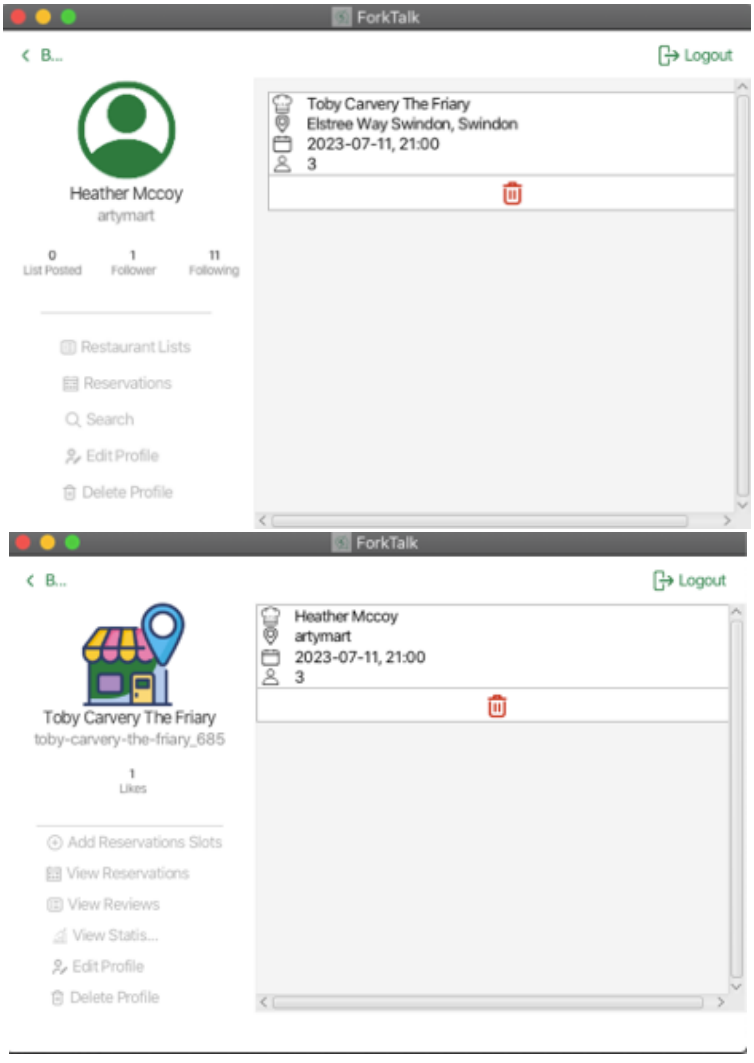
- Find the suggested RestaurantLists



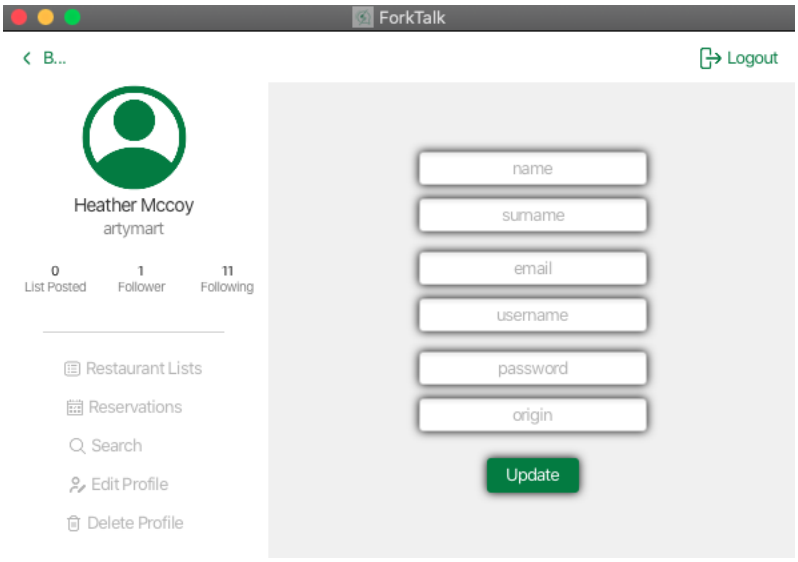
- USER
  - User -> Restaurant Lists



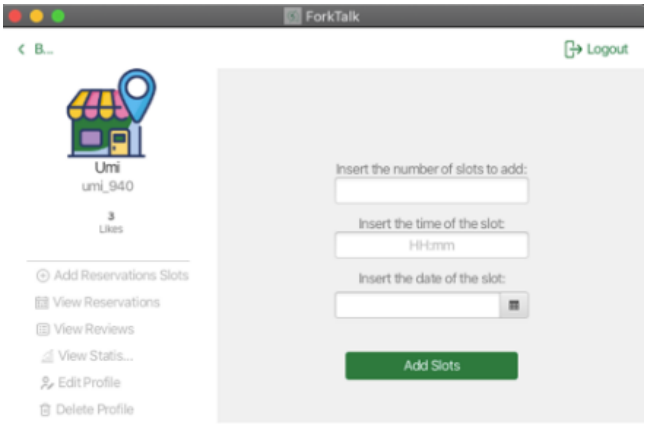
  - User -> Reservations



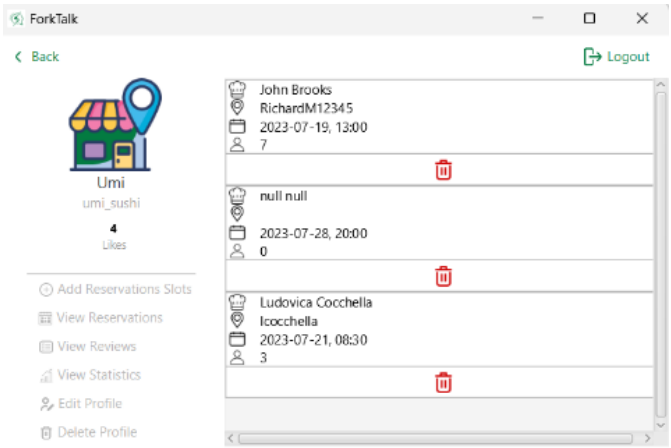
- User -> edit profile



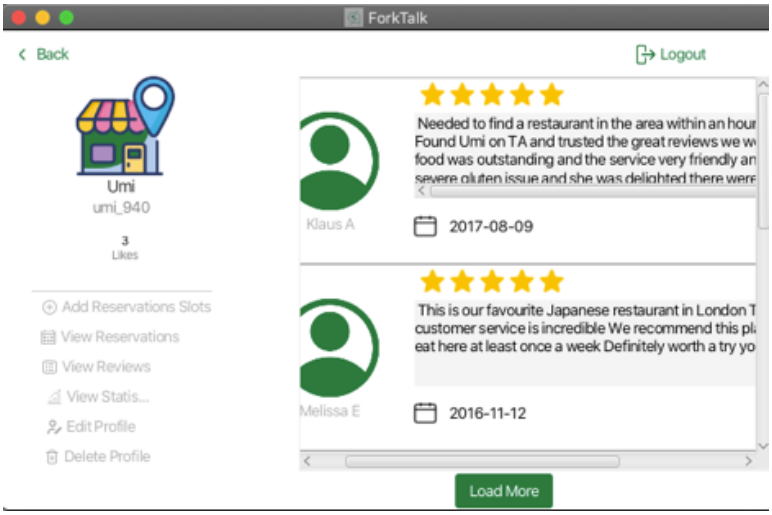
- RESTAURANT
  - Add reservations slots



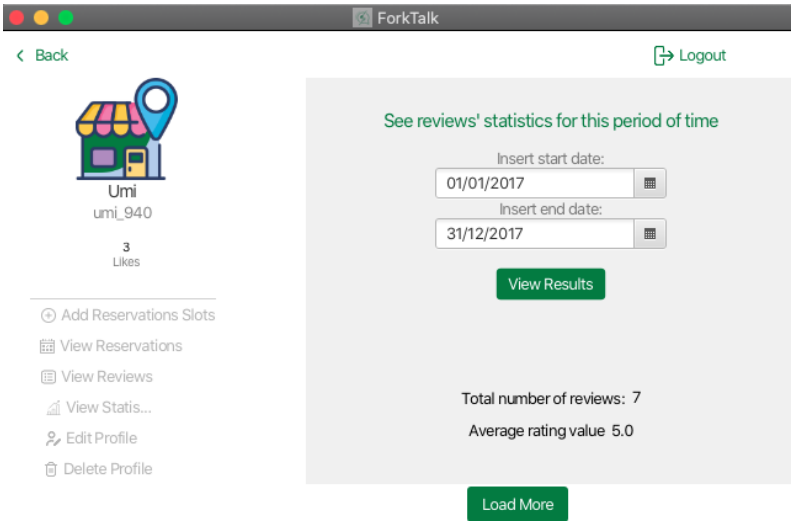
  - View reservations



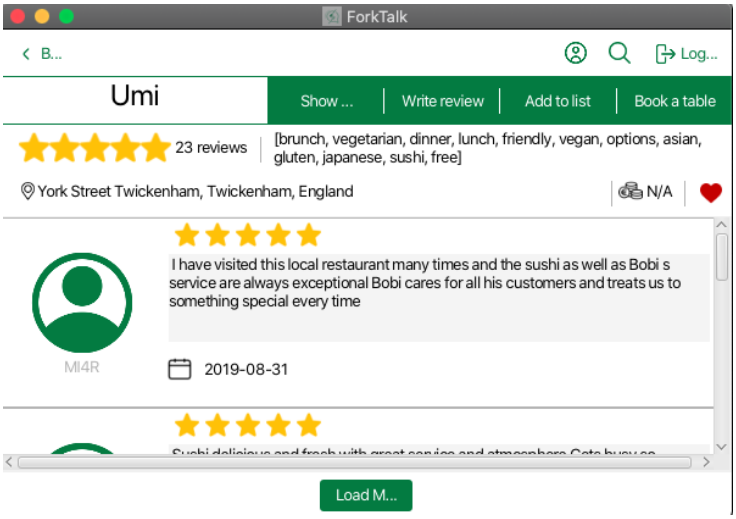
- View Reviews



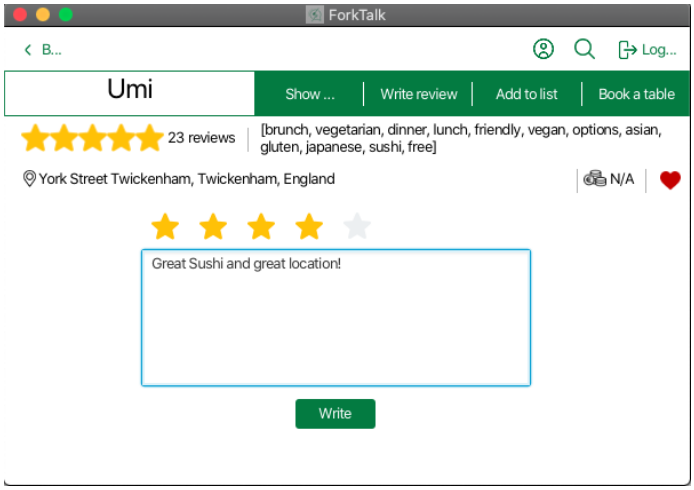
- View Statistics



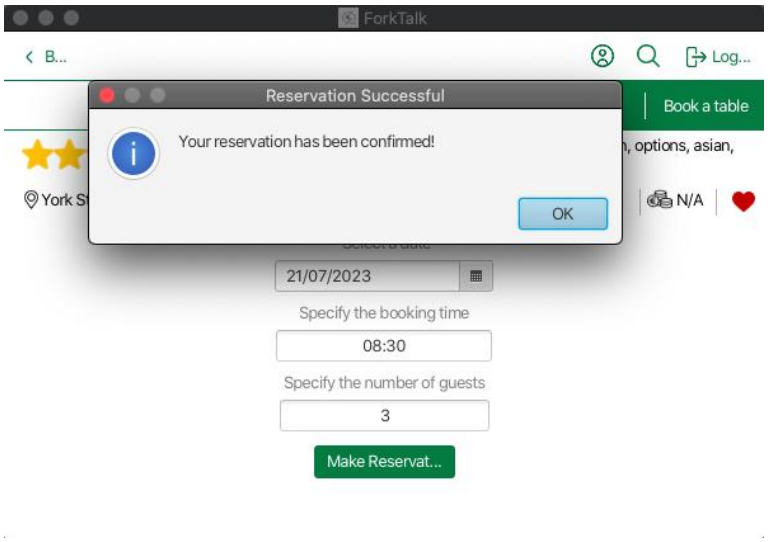
- User -> Go to restaurants -> show restaurants (like botton clicked)



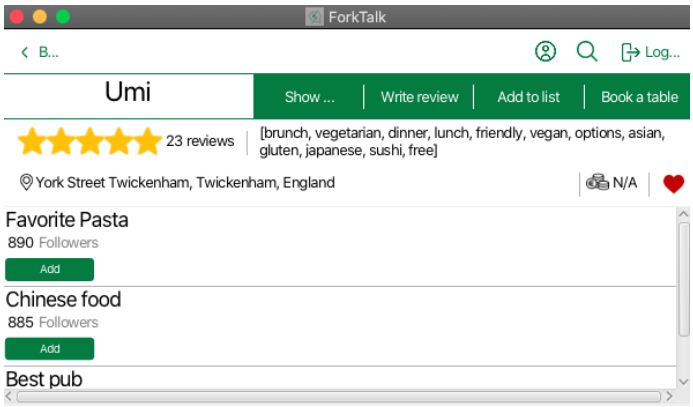
- User -> Go to restaurant-> write review



- User -> Go to restaurant -> Book a table



- User -> Go to restaurant -> Add to List



- User -> Find RestaurantLists -> show List