UNIVERSITÀ DI PISA

Msc. in Artificial Intelligence and Data Engineering

Social Restaurant

Large Scale and Multi Structured Databases Project

Leonardo Turchetti

Academic Year 2022/2023

Contents

1 Introduction

Social Restaurant is a Java application which allows users to search, comment and booking tables on restaurants retrieved from different sources.
The application has two main purposes:

- Merge in a unique database, restaurants published in various open access archives providing fast and efficient ways to search restaurants.
- Realizing a social network, allowing users to interact and express their opinions about restaurants. Users can also create Restaurants favourite list in which they can add restaurants.

Project repository: https://github.com/Leonardo-Turchetti/Progetto-Large-Scale

1.1 Dataset

The first component of the application database are the restaurants. In order to respect the variety constraint we downloaded restaurants information from two different sources: the first source is a dataset from Kaggle that restaurants from European Cities from TripAdisor; the second source is another dataset from Kaggle that includes Bangalore's restaurants from Zomato.
The other components of the dataset are the Users and the Bookings and in this case, we populated manually the database about this two type of data.

2 Requirements

2.1 Functional Requirements

This section describes the requirements that the application provides.

Unregistered User:

- Unregistered User can register a Registered User account on the platform

Registered User:

- Registered User can search for Restaurants
  - by title
  - by city
  - by cuisine style
- Registered User can add and remove Restaurants from their Restaurants Favourite List
- Registered User can search other Users by username
- Registered User can follow and unfollow Users
- Registered User can comment Restaurants
- Registered User can like Restaurants
- Registered User can book a table on a restaurants
- Registered User can delete a booking of a restaurants
- Registered User can log out from the platform

Admin:

- Admin can do all the operations that a Users can do
- Admin can compute restaurants, booking and users analytics
- Admin can search all the booking by a specific period of time

2.2 Non Functional Requirements

- The application needs to be highly available and always online.
- The system needs to be tolerant to data lost and to single point of failure.
- The application needs to provide fast response search results to improve the user experience.
- The application needs to be user-friendly so a GUI must be provided.

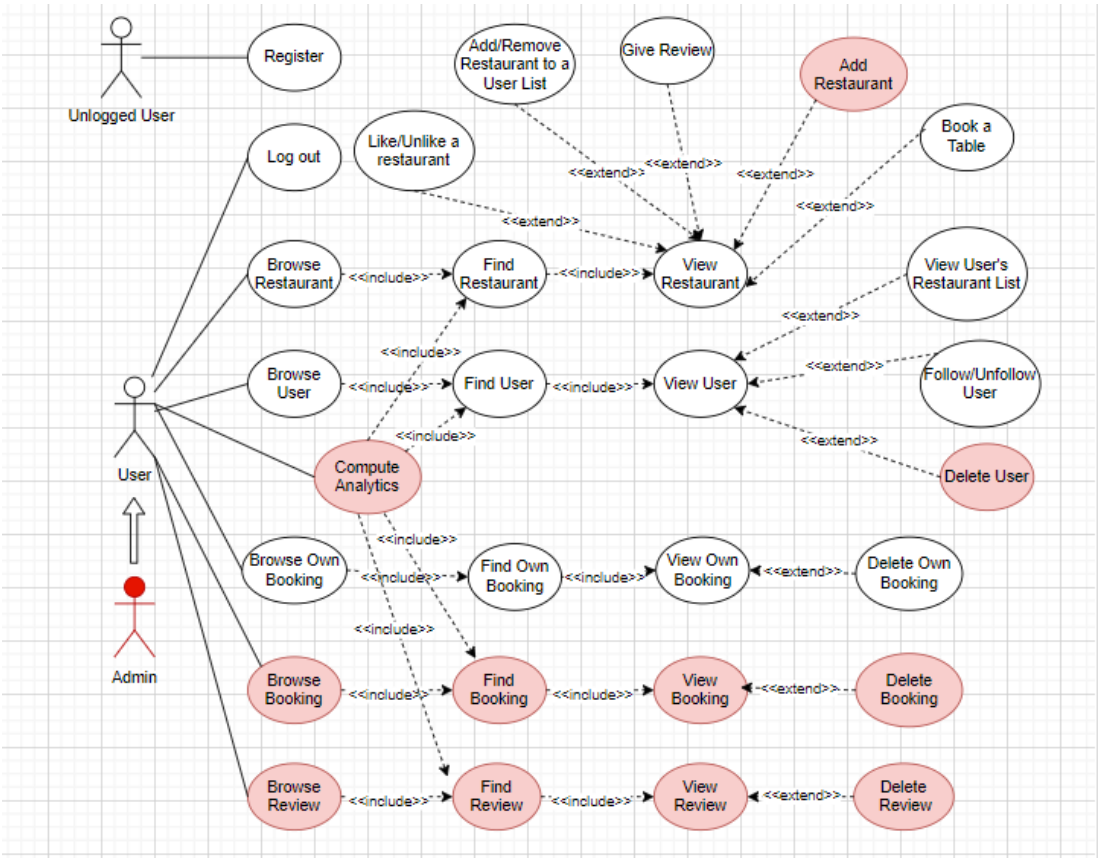
2.3 Main Actors

The main actors of the application are three:

- **Unregistered User**: User that is not registered on the application, in order to access he must sign-up.
- **Registered User**: User that is registered, he can access application by logging-in.
- **Admin**: User with highest level of privilege, he can elect or dismiss moderators, he can delete users.

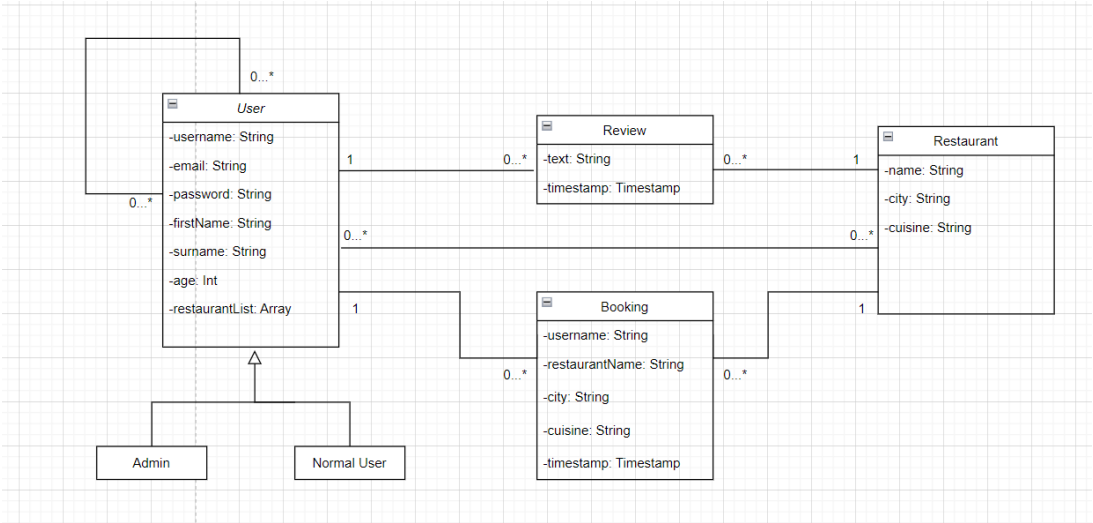
3 Use Case and UML Class Diagram Design

3.1 Use Case

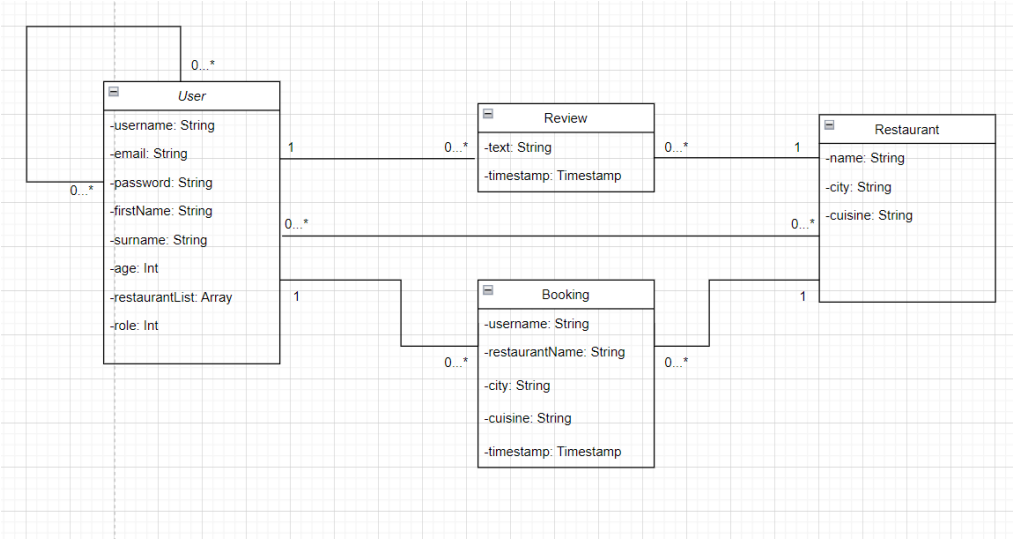


3.2 UML Class Diagram

There are four main entities: Restaurant, User, Booking, Review. In the diagram, User is a generalization of the two main actors (Registered User and Administrator) of the use case diagram, each actor can perform in addition to its own action, the same action of User.



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



4 Architectural Design

4.1 Software Architecture and Application Package Structure

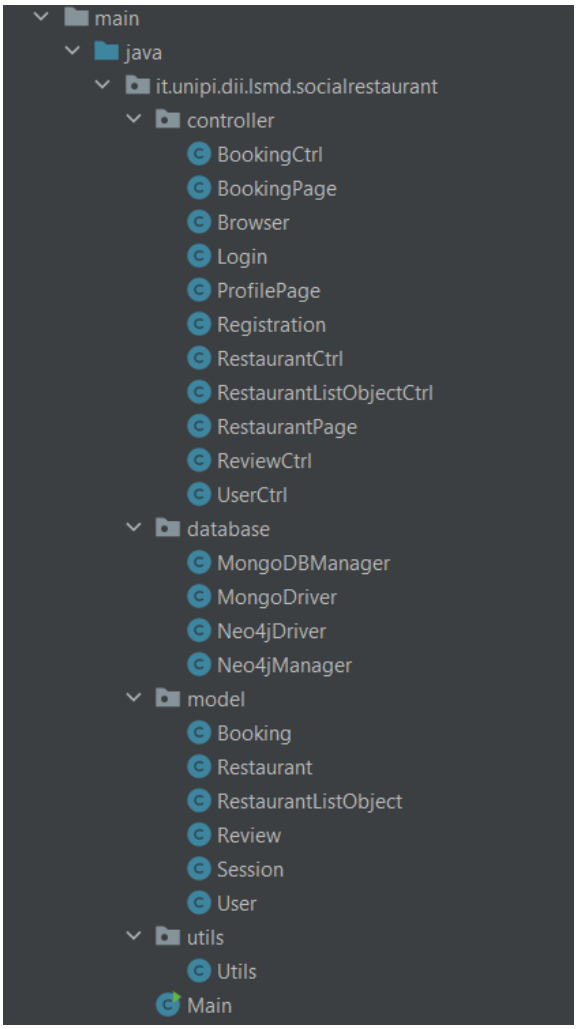
The application was implemented as client-server architecture, with middleware implemented on the client side. Client's features can be divided in three parts:

- **Front-end**: this part consists of a graphical interface developed in JavaFX and FXML files. User can use the GUI to interact with the application and managed by controllers used to handle events and views of the informations.
- **Middleware**: the duty of this part is to handle the connections and communications with the servers, both MongoDB and Neo4j.
- **Back-End**: this part is represented by the two servers MongoDB and Neo4j

The SocialRestaurant application is composed of the following packages and classes.

it.unipi.dii.lsmd.socialresturant.model
This package contains the classes required for the model. These classes are the java bean for our application. Classes:

- Restaurant: The class contains paper information.
- User: The class contains user information
- RestaurantListObject: The class contains information of a user's restaurant list.
- Review: The class contains review information.
- Booking: The class contains booking information.
- Session: The page contains information about the logged user.



it.unipi.dii.lsmd.socialresturant.database
This package contains the classes to interface with databases. Classes:

- MongoDBDriver: Implement the methods for to manage the connection with MongoDB.
- MongoDBManager: In this class are implemented all the queries to interact with MongoDB.
- Neo4jDriver: Implement the methods for to manage the connection with Neo4j.
- Neo4jManager: In this class are implemented all the queries to interact with Neo4j.

it.unipi.dii.lsmd.socialresturant.controller
This package contains the classes to interface with databases. Classes:

- Browser: this class manage the homepage and allows the user to navigate to the other pages of application and to carry out search operation and browser.
- ReviewCtrl: this class manage the view of review card.
- Login: this class manage the login page of application.
- RestaurantCtrl: this class manage the view of restaurant card.
- RestaurantPage: this class manage the page of a generic restaurant and allows the user to review, like or add restaurant to a restaurant list.
- ProfilePage: this class manage the generic user page of the application and allows the user to view/modify his data or view data of another user.
- RestaurantListObjectCtrl: this class manage the view of resturant list card.
- BookingPage: this class manage the booking page, where information about booking is displayed
- Register: this class manage the register page of application.
- UserCtrl: this class manage the view of user card.
- BookingCtrl: this class manage the view of booking card.

it.unipi.dii.lsmd.socialresturant.utils
This package contains utility functions. Classes:

- Utils: this class contains function for manage the configuration parameters necessary for the application and the scene change for pages.

4.2 Document DB and Graph DB Model

We chose to use a document database in our application to manage most of the data regarding users, restaurants and booking. This choice allows us to respect the flexibility and high performance requirements. We decided to use three collections: Resturants, Users and Booking. Regarding the performance, using a document DB allows us to embed objects that are commonly used together to avoid expensive join operations. We chose to embed restaurant list inside the document of the user that owns it because inside a User Page are shown also all the restaurants that he added. In each restaurant is stored only a subset of information, the ones that are used to display a preview of the restaurants, while the complete information about restaurants can be retrieved in the corresponding restaurant document. We also decided to embed reviews inside the resturant document, for similar reasons: in the Resturant Page we show all the reviews related to the resturant. The schema less approach offered by document dbs also allows us to handle in a flexible way users without restaurant list or restaurant without reviews.

In the following figures are shown an example of a document of Restaurant, User and Booking collection.

```
_id: ObjectId('63dec2b1494d1ec44b0f2e5e')
username: "bill"
restaurantname: "Audrion"
city: "Athens"
cuisine: "French"
timestamp: "2022-08-15"
```


We decided to use a graph database to manage the social network features of the application. Our choice was to keep the graph database as "light" as possible and to use it only for handling social network relationships and social network-based analytics.
The GraphDB nodes with their attributes are the following:

- **User**: username
- **Restaurant**: name,city,cuisine

The relationships present in our graph database are the following:

- **Follows**: If a user follows another user (:User)-[:FOLLOWS]→(:User)
- **Like**: If a user likes a restaurant (:User)-[:LIKE]→(:Restaurant)
- **Add**: If a user add a restaurant to his restaurant list (:User)-[:ADD]→(:Restaurant)

4.3 Distributed Database Design

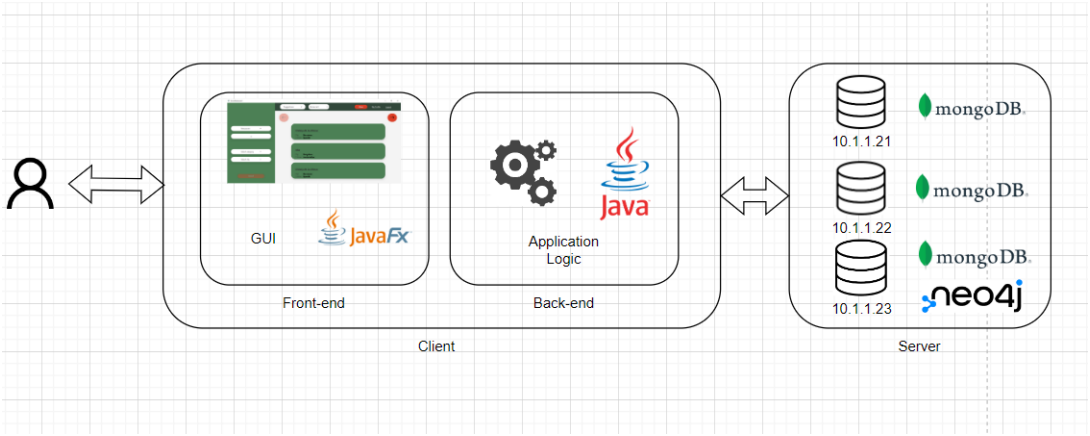
According to the Non-Functional Requirements, our system must provide high availability, fast response times and to be tolerant to data lost and single point of failure. To achieve such results, we orient our application on the A (Availability), P (Partition Protection) edge of the CAP triangle. In our application we want to offer a high availability of the content, even if an error occurs on the network layer, at the cost of returning, to the users, data which is not always accurate. For this reason, we adopt the Eventually Consistency paradigm on our dataset.

- **High availability** of the service due to the way we handle the writes operation. In fact, after receiving a write operation we will update one server, and the replicas will be updated in a second moment. In this way writes operation don't keep the server busy for too much time.
- **Partition Protection** of the service is guarantee by the presence of replicas in our cluster, if one server is down, we can continue to offer our service by searching the content in the replicas.

The possibility to implement a sharding would permit to evenly balance the workload among the nodes, improving users' experience.

4.3.1 Replicas

We decide to use replicas due to guarantee Partition protection and Availability of the content. When the user does a write operation just one server will be updated immediately, in this way the write operation will not take so much time and other users that are going to do read operations, which we suppose will be the most frequent ones, don't have to wait too much. The presence of replicas helps us also if we have network problems and we can't reach one server, in this case the query will be redirected to another server which contains the replica. Unfortunately, to guarantee these two services we can't ensure that the user will retrieve the most updated content. When the "user update" process runs, we estimate once a day, we want to guarantee that all the replicas will have the same information about new users, so only in this case we need a consistency constraint, and the write operations will take more time. In our project we have only 3 replicas are present in our systems: one for each machine of the provided cluster. However, in our implementation we have replicas only for **MongoDB**, because **Neo4J** replicas is a premium feature.



4.3.2 Sharding

To implement the sharding we select two different sharding keys, one for each collection of the document database (User, Restaurant and Booking Collection), for each collections we use as partitioning method the Consistent hashing, because if the number of nodes of the cluster will change, it will be easier to relocate data. The sharding keys are:

- For the User Collection we choose the attribute "username", which is unique among the users, as sharding key.
- For the Restaurant Collection we choose the attribute "city" as sharding key.
- For the Booking Collection we choose the attribute "timestamp" as sharding key.

5 MongoDB Design and Implementation

The MongoDB database contains the following collections: Restaurant, Booking and Users. We decided to store in each document all the information that we need to build the information pages. Sometimes we use mongo also to display advanced information retrieved by analytic query (basically when we do not need to use the relationships between entities).

5.1 Queries implementation

5.1.1 CRUD Operation

Create

```
db.Users.insertOne(
        {
        username: "username",
        email: "email",
        password: "password",
        firstName: "firstname",
        lastName: "lastName",
        age: "age"
        }
    )
```

Update

```
db.Users.updateOne(
            {
         username: "username",
            email: "email",
         password: "password",
        firstName: "firstname",
        lastName: "lastName",
        age: "age",
        type: "type"
            }
    )
```

Delete

```
db.Users.deleteOne(
            {
            username: "username"
            }
        )
```

Get User By Username

```
db.Users.findOne(
        {
    username: "username"
        }
    )
```

5.1.2 Analytics

Get the most commented resturants
Select restaurants with the highest number of reviews in a specific period.

```
public List<Pair<Restaurant, Integer>> getMostCommentedRestaurants(String period, int skipDoc, int limi
    LocalDateTime localDateTime = LocalDateTime.now();
    LocalDateTime startOfDay;
    switch (period) {
        case "all" -> startOfDay = LocalDateTime.MIN;
        case "month" -> startOfDay = localDateTime.toLocalDate().atStartOfDay().minusMonths(1);
        case "week" -> startOfDay = localDateTime.toLocalDate().atStartOfDay().minusWeeks(1);
        default -> {
            System.err.println("ERROR: Wrong period.");
            return null;
        }
    }
    String filterDate = startOfDay.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

    List<Pair<Restaurant, Integer>> results = new ArrayList<>();
    Gson gson = new GsonBuilder().serializeSpecialFloatingPointValues().create();
    Consumer<Document> convertInRestaurant = doc -> {
        Restaurant restaurant = gson.fromJson(gson.toJson(doc), Restaurant.class);
        results.add(new Pair(restaurant, doc.getInteger( key: "totalComments")));
    };

    Bson unwind = unwind( fieldName: "$reviewList");
    Bson filter = match(gte( fieldName: "reviewList.timestamp", filterDate));
    Bson group = new Document("$group",
            new Document("_id",
                    new Document("name", "$name")
                            .append("address", "$address")
                            .append("city", "$city")
                            .append("cuisine", "$cuisine"))
                    .append("totalComments",
                            new Document("$sum", 1)));
    Bson project = project(fields(excludeId(),
            computed( fieldName: "name",  expression: "$_id.name"),
            computed( fieldName: "address",  expression: "$_id.address"),
            computed( fieldName: "city",  expression: "$_id.city"),
            computed( fieldName: "cuisine",  expression: "$_id.cuisine"),
            include( ...fieldNames: "totalComments")));
    Bson sort = sort(Indexes.descending( ...fieldNames: "totalComments"));
    Bson skip = skip(skipDoc);
    Bson limit = limit(limitDoc);
    restaurantCollection.aggregate(Arrays.asList(unwind, filter, group, project,
            sort, skip, limit)).forEach(convertInRestaurant);

    return results;
}
```

Mongo Shell:

```
> db.restaurant.aggregate([
  {$unwind: "$reviewList"},
  {$match: {"reviewList.timestamp": {$gte: "2021-12-20 00:00:00"}}},
  {$group: {_id: {name: "$name", city: "$city", cuisine:"$cuisine"}, totalComments: {$sum: 1}}},
  {$project: {_id: 0, name:"$_id.name", city:"$_id.city", cuisine:"$_id.cuisine",totalComments: 1}},
  {$sort: {totalComments: -1}},
  {$skip: 3},
  {$limit: 3}
]);
```

Get categories summary by comments

This function returns the categories of cuisine ordered by the number of reviews in a specific period.

```
public List<Pair<String, Integer>> getCategoriesSummaryByComments(String period) {
    LocalDateTime localDateTime = LocalDateTime.now();
    LocalDateTime startOfDay;
    switch (period) {
        case "all" -> startOfDay = LocalDateTime.MIN;
        case "month" -> startOfDay = localDateTime.toLocalDate().atStartOfDay().minusMonths(1);
        case "week" -> startOfDay = localDateTime.toLocalDate().atStartOfDay().minusWeeks(1);
        default -> {
            System.err.println("ERROR: Wrong period.");
            return null;
        }
    }
    String filterDate = startOfDay.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

    List<Pair<String, Integer>> results = new ArrayList<>();
    Consumer<Document> rankCategories = doc ->
            results.add(new Pair<>((String) doc.get("_id"), (Integer) doc.get("tots")));

    Bson unwind = unwind( fieldName: "$reviewList");
    Bson filter = match(gte( fieldName: "reviewList.timestamp", filterDate));
    Bson group = group( id: "$cuisine", sum( fieldName: "tots",  expression: 1));
    Bson sort = sort(Indexes.descending( ...fieldNames: "tots"));
    restaurantCollection.aggregate(Arrays.asList(unwind, filter, group, sort)).forEach(rankCategories);

    return results;
}
```

Mongo Shell:

```
db.restaurant.aggregate([
  {$unwind: "$reviewList"},
  {$match: {"reviewList.timestamp": {$gte: "2021-12-20 00:00:00"}}},
  {$group: {_id: "$cuisine", tots: {$sum: 1}}},
  {$sort: {tots: -1}}
]);
```

Get cities summary by booking

This function returns the cities of ordered by the number of booking in a specific period.

```java
public List<Pair<String, Integer>> getCitiesSummaryByBooking(String period) {
    LocalDateTime localDateTime = LocalDateTime.now();
    LocalDateTime startOfDay;
    switch (period) {
        case "all" -> startOfDay = LocalDateTime.MIN;
        case "month" -> startOfDay = localDateTime.toLocalDate().atStartOfDay().minusMonths(1);
        case "week" -> startOfDay = localDateTime.toLocalDate().atStartOfDay().minusWeeks(1);
        default -> {
            System.err.println("ERROR: Wrong period.");
            return null;
        }
    }
    String filterDate = startOfDay.format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

    List<Pair<String, Integer>> results = new ArrayList<>();
    Consumer<Document> rankCategories = doc ->
            results.add(new Pair<>((String) doc.get("_id"), (Integer) doc.get("tots")));

    Bson unwind = unwind( fieldName: "$timestamp");
    Bson filter = match(gte( fieldName: "timestamp", filterDate));
    Bson group = group( id: "$city", sum( fieldName: "tots",  expression: 1));
    Bson sort = sort(Indexes.descending( ...fieldNames: "tots"));
    bookingCollection.aggregate(Arrays.asList(unwind, filter, group, sort)).forEach(rankCategories);

    return results;
}
```

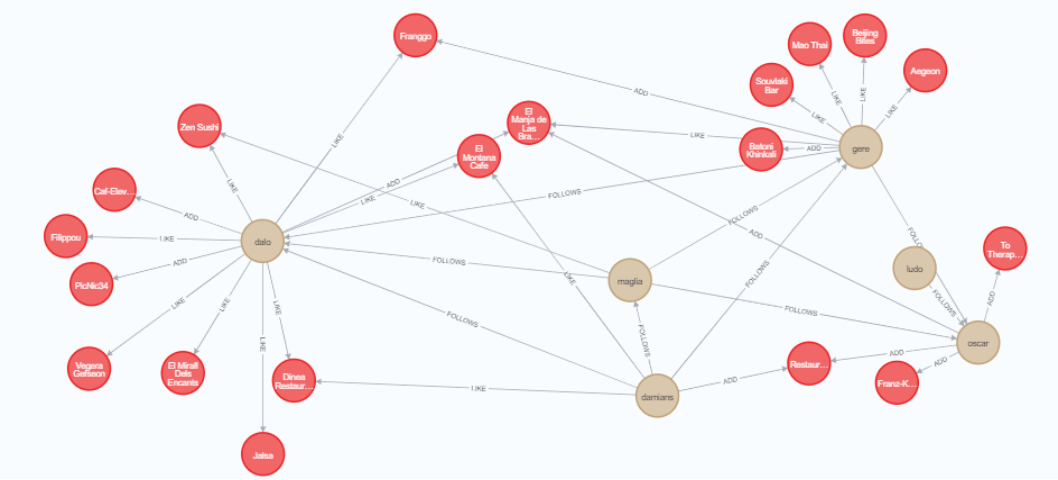
Mongo Shell:

```
> db.booking.aggregate([
    {$unwind: "$timestamp"},
    {$match: {"timestamp": {$gte: "2021-12-20 00:00:00"}}},
    {$group: {_id: "$city", tots: {$sum: 1}}},
    {$sort: {tots: -1}}
  ]);
```

6 Neo4j Design and Implementation

In Neo4j database there are the following entities: Restaurant and User, that contain only the basic information needed to show a preview of the entity. There are also the following relationships between entities:

- USER – LIKE → RESTAURANT: Each user can like one or more restaurants; it is useful for showing suggestions based on restaurant's like.
- USER – ADD → RESTAURANT: Each user can add one or more restaurants to his restaurant list; it is useful for showing suggestions based on restaurant's add.
- USER – FOLLOWS → USER: Each user can follow one or more users; it is useful for showing suggestions based on users.



6.1 Queries Implementation

6.1.1 CRUD Operation

Create

```
//Create a User node
        CREATE (u:User {username: $username})


//Create a Restaurant node
        CREATE (r:Restaurant{name: $name, city: $city, cuisine: ...
            $cuisine})


//Create a User-LIKE->Restaurant relation
        MATCH (a:User), (b:Restaurant)
        WHERE a.username = $username AND (b.name = $name)
        MERGE (a)-[r:LIKE]->(b)
        ON CREATE SET r.date = $timestamp


//Create a User-ADD->Restaurant relation
        MATCH (a:User), (b:Restaurant)
        WHERE a.username = $username AND (b.name = $name)
        MERGE (a)-[r:ADD]->(b)


//Create a User-FOLLOWS->User relation
        MATCH (u:User {username: $username}), (t:User {username: ...
            $target})
        MERGE (u)-[p:FOLLOWS]->(t)
        ON CREATE SET p.date = $timestamp
```

Read

```
\\Retrieve NumFollower of a User
      MATCH (:User {username: $username})<-[r:FOLLOWS]-()
      RETURN COUNT(r) AS numFollowers


\\Check a FOLLOWS relationship
      MATCH (a:User{username:$userA})-[r:FOLLOWS]->(b:User ...
          {username:$userB})
      RETURN COUNT(*)


\\Check an ADD relationship
      MATCH (:User{username:$user})-[r:ADD]->(p:Restaurant)
      WHERE (p.name = $name)
      RETURN COUNT(*)


\\Check an ADD relationship
     MATCH (:User{username:$user})-[r:LIKE]->(p:Restaurant)
     WHERE (p.name = $name)
     RETURN COUNT(*)
```

```
\\Retrieve NumLike of a Restaurant
      MATCH (p:Restaurant)<-[r:LIKE]-()
      WHERE p.name = $name
      RETURN COUNT(r) AS numLikes
```

Update

```
\\Update User Username
        MATCH (u:User {username: $oldUsername})
        SET u.username = $username
```

Delete

```
//Delete a User
        MATCH (u:User) WHERE u.username = $username DETACH DELETE u


//Delete a Restaurant
        MATCH (r:Restaurant) WHERE r.name = $name DETACH DELETE r


//Delete a FOLLOWS relationship
        MATCH (:User {username: $username})-[r:FOLLOWS]->(:User ...
           {username: $target})
        DELETE r


//Delete a LIKE relationship
        MATCH (u:User{username:$username})-[r:LIKE]->(b:Restaurant)
        WHERE b.name = $name
        DELETE r


//Delete an ADD relationship
        MATCH (u:User{username:$username})-[r:ADD]->(b:Restaurant)
        WHERE b.name = $name
        DELETE r
```

6.1.2 Analytics

Most followed users
The query returns the list of the most followed users.

```
MATCH (target:User)<-[r:FOLLOWS]-(:User)
RETURN DISTINCT target.username AS Username,
COUNT(DISTINCT r) as numFollower
ORDER BY numFollower DESC, Username
SKIP $skip
LIMIT $num
```

Most liked restaurants
The query returns the list of the most liked restaurants in a specific period.

```
MATCH (:User)-[l:LIKE]->(p:Restaurant)
WHERE l.date >= $start_date
RETURN p.name AS Name, p.city AS City, p.cuisine AS Cuisine,
COUNT(l) AS like_count
ORDER BY like_count DESC, Name
SKIP $skip
LIMIT $limit
```

Category summary by likes
The query returns a list of the most liked categories of cuisine in a specific period.

```
MATCH (p:Restaurant)<-[l:LIKE]-(:User)
WHERE l.date >= $start_date
RETURN COUNT(l) AS nLikes, p.cuisine AS Cuisine
ORDER BY nLikes DESC
```

Suggested Users
The query returns a list of suggested users for the logged user. Suggestions are based on most followed users who are 2 FOLLOWS hops far from the logged user (first level), while the second level of suggestion returns most followed users that have likes in common with the logged user.

```
MATCH (me:User {username: $username})-[:FOLLOWS*2..2]->(target:User), (target)<-[r:FOLLOWS]-()
WHERE NOT EXISTS((me)-[:FOLLOWS]->(target))
RETURN DISTINCT target.username AS Username,
COUNT(DISTINCT r) as numFollower
ORDER BY numFollower DESC, Username
SKIP $skipFirstLevel
LIMIT $firstLevel
UNION
MATCH (me:User {username: $username})-[:LIKE]->()<-[:LIKE]-(target:User), (target)<-[r:FOLLOWS]-()
WHERE NOT EXISTS((me)-[:FOLLOWS]->(target))
RETURN target.username AS Username,
COUNT(DISTINCT r) as numFollower
ORDER BY numFollower DESC, Username
SKIP $skipSecondLevel
LIMIT $secondLevel
```

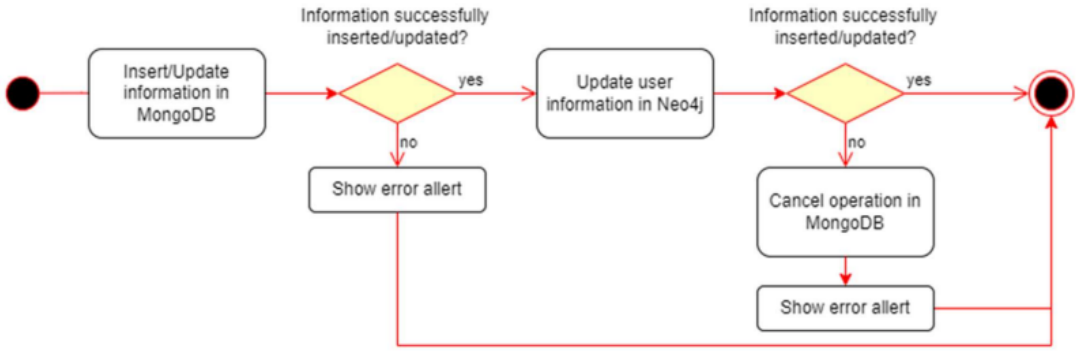
Suggested Restaurants

The query returns a list of suggested restaurants for the logged user. Suggestions are based on restaurants liked by followed users (first level) and restaurants liked by user that are 2 FOLLOWS hops far from the logged user (second level). Restaurants returned are ordered by the number of times they appeared in the results, so restaurants that appear more are most likely to be like the interests of the logged user.

```
MATCH (target:Restaurant)<-[r:LIKE]-(u:User)<-[:FOLLOWS]-(me:User{username:$username})
WHERE NOT EXISTS((me)-[:LIKE]->(target))
RETURN target.name AS Name, target.city AS City, target.cuisine as Cuisine,
COUNT(*) AS nOccurences
ORDER BY nOccurences DESC, Name
SKIP $skipFirstLevel
LIMIT $firstlevel
UNION
MATCH (target:Restaurant)<-[r:LIKE]-(u:User)<-[:FOLLOWS*2..2]-(me:User{username:$username})
RETURN target.name AS Name, target.city AS City, target.cuisine as Cuisine,
COUNT(*) AS nOccurences
ORDER BY nOccurences DESC, Name
SKIP $skipSecondLevel
LIMIT $secondLevel
```

7 Cross-Database Consistency Management

The operations which require cross-database consistency management are Add/Remove/Update a User Add/Remove Restaurant. For the update of the users, we have only to keep consistency on the field "username", because it is the only information updatable in Neo4J entity.
In general, if an error occurs with the first write operation we show an error to the user, while if an error occurs with the second write operation, we also show an error, but we must undo the first write operation.



8 Index analysis

As the query analysis has revealed, our application is read heavy, so we introduced indexes to tune queries that are executed more frequently.

8.1 Restaurant's Search Index

One of the main purposes of the application is to provide fast search results regarding restaurants, so we will focus mostly on indexes related to restaurants.
We will perform statistical tests to understand whether the insertion of an index is beneficial or not in terms of query performance.
Index "Name"

```
\\Query SearchRestaurantbyName("a")
Result without index:
        executionTimeMillis: 0
        totalKeysExamined: 100
        totalDocsExamined: 100

Result with index "name":
        executionTimeMillis: 1
        totalKeysExamined: 100
        totalDocsExamined: 78
```

Index "Cuisine"

```
\\Query SearchRestaurantbyCuisine("Italian")
Result without index:
        executionTimeMillis: 1
        totalKeysExamined: 100
        totalDocsExamined: 100

Result with index "cuisine":
        executionTimeMillis: 0
        totalKeysExamined: 100
        totalDocsExamined: 13
```

8.2 Booking's Search Index

Another main purpose of the application is to provide fast search results regarding booking, so we will focus mostly on indexes related to booking.
We will perform statistical tests to understand whether the insertion of an index is beneficial or not in terms of query performance.

Index "Username"

```
\\Query SearchBookingbyUsername("a")
Result without index:
        executionTimeMillis: 1
        totalKeysExamined: 51
        totalDocsExamined: 51

Result with index "username":
        executionTimeMillis: 1
        totalKeysExamined: 51
        totalDocsExamined: 19
```

Index "RestaurantName"

```
\\Query SearchBookingbyRestaurant("Audrion")
Result without index:
        executionTimeMillis: 0
        totalKeysExamined: 51
        totalDocsExamined: 51

Result with index "restaurantname":
        executionTimeMillis: 0
        totalKeysExamined: 51
        totalDocsExamined: 11
```

8.3 User's Search Index

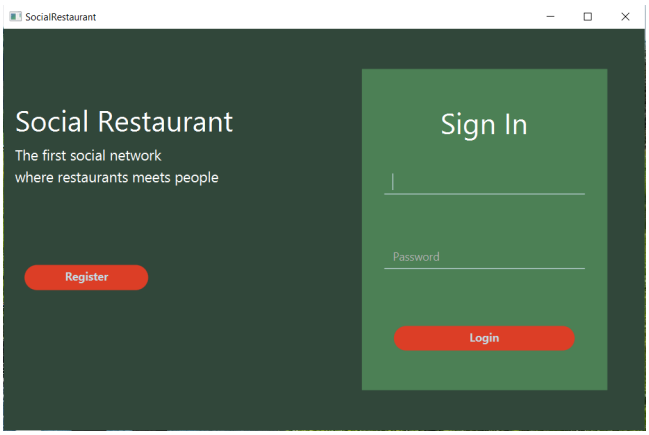
The last main purpose of the application is to provide fast search results regarding user, so we will focus mostly on indexes related to user.
We will perform statistical tests to understand whether the insertion of an index is beneficial or not in terms of query performance.

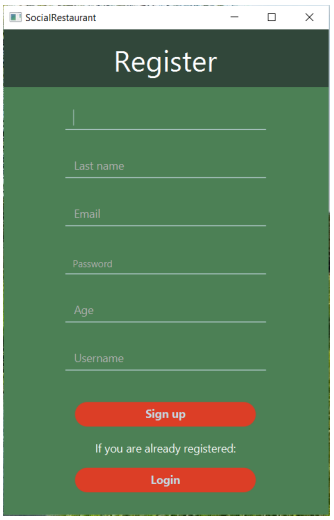
Index "Username"

```
\\Query SearchUserbyUsername("a")
Result without index:
        executionTimeMillis: 0
        totalKeysExamined: 20
        totalDocsExamined: 20

Result with index "username":
        executionTimeMillis: 0
        totalKeysExamined: 20
        totalDocsExamined: 12

\\Query SearchUserbyUsername("co")
Result without index:
        executionTimeMillis: 0
        totalKeysExamined: 51
        totalDocsExamined: 51

Result with index "username":
        executionTimeMillis: 0
        totalKeysExamined: 20
        totalDocsExamined: 1
```

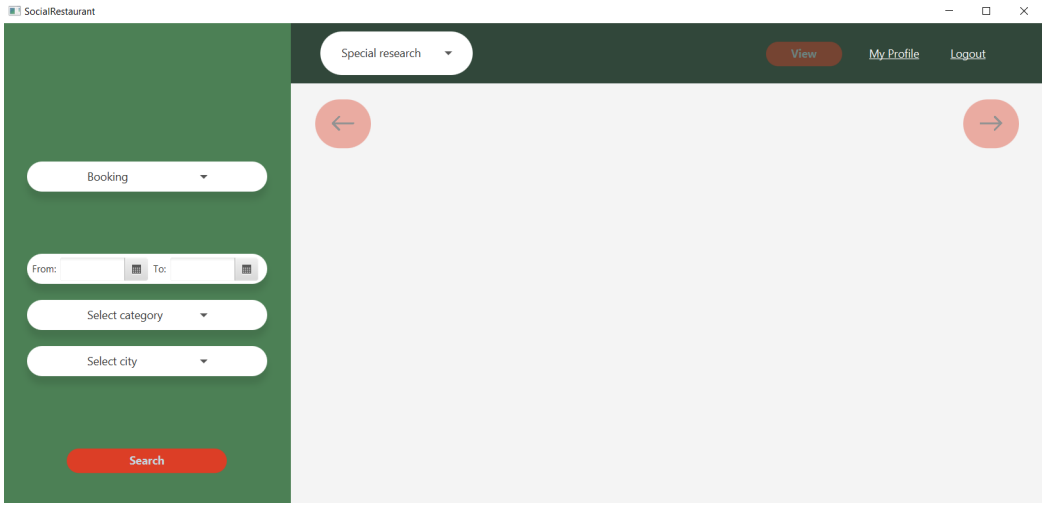
9 User Manual

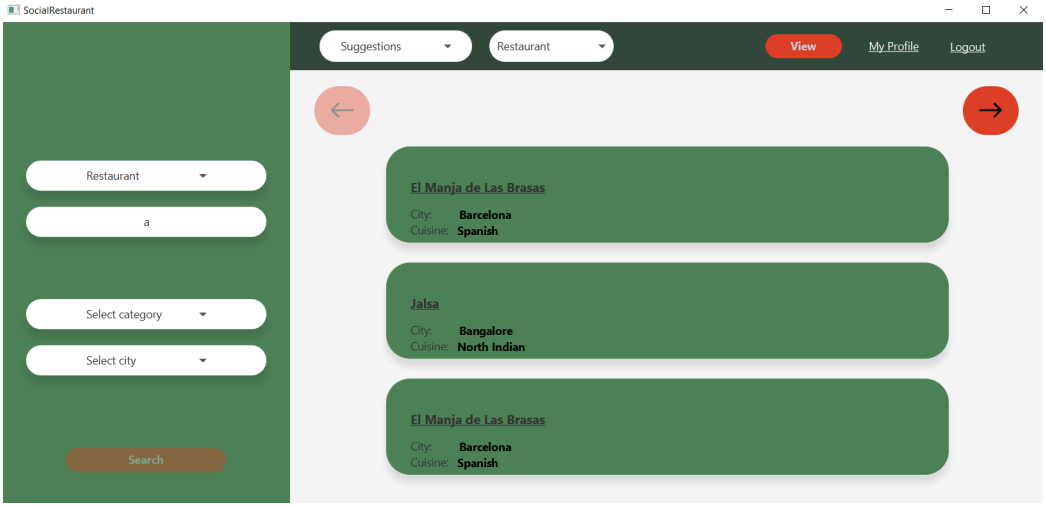
"SocialRestaurant" is a user-friendly application, where users can search restaurant for booking a table and interact with other users as a social network. The first page which a user will see is the **Login** page, where a user can insert his own credentials or go to the **Register** page to create a new account.



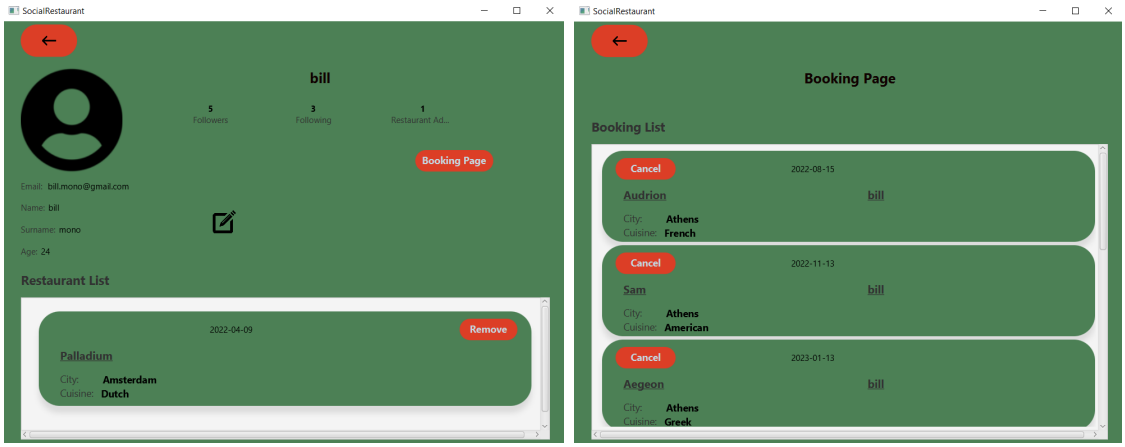


After the login we are redirected to the **Browser**, in this page users can search by parameters other users, restaurants or booking(only if the user is the admin). There is also the possibility to select special view of the data inside the application, in this way is possible to see some suggestions, analytics or summaries(some of this operations are possible only in the admin mode).

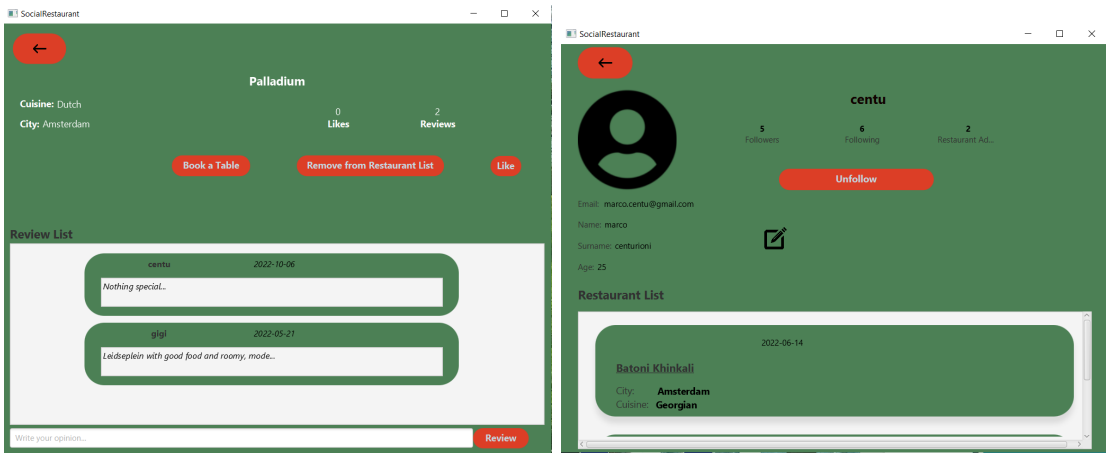




If the user moves to his **Profile** page, he will see his own information, the number of users that he follows and the number of his follower. He can also edit his profile or go this his **Booking** page. If the user is the admin on his **Profile** page it can also add a new Restaurant to the application, while in the **Profile** page of the other user it can also see his booking page and delete the user.



Clicking on a restaurant a user will be redirected to the **Restaurant** page. Here a user can see the information about the restaurant, add or remove the restaurant from his restaurant list, like the restaurant and booking a table.
Clicking on a username a user will be redirected to the **Profile** page of another user. Here a user can see the information about the user, follow/unfollow the user to see his restaurant favourite list.



Project repository: https://github.com/Leonardo-Turchetti/Progetto-Large-Scale