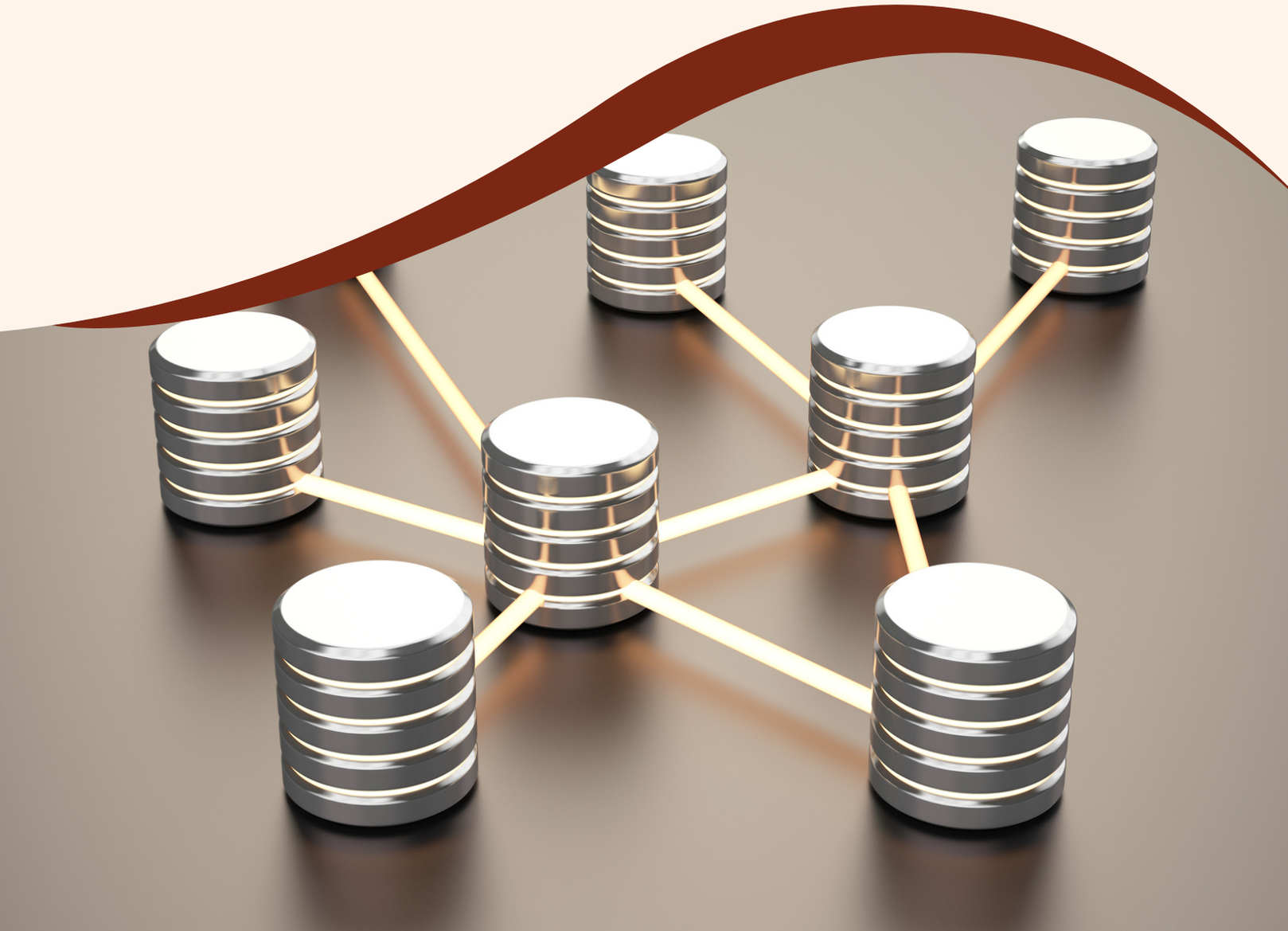
UNIVERSITÀ DI PISA

LARGE-SCALE AND MULTISTRUCTURE DATABASE PROJECT

Gabriele Marino
Matteo Pasqualetti
Francesco De Vita

NEO4FOOD
DELIVERY



CONTENTS

1. Introduction

Neo4Food is a service website where people can find Restaurants and request for food deliveries. Customers can also use the built-in social-network system to get recommendations based on following people and rated Restaurants.
The application is also meant to be used by restaurants themself to expand their distribution chain.

1.1. Dataset

The dataset we are using for the application holds basic information about Restaurants from major cities in the USA along with their available offered dishes. We have also integrated scraped data about Italian Restaurants and related offered dishes. We also scraped comments around different websites.

Customers and Orders on the other hand, have been dynamically generated.

To let different databases, work together, we had to adapt them a little bit by normalizing fields.

2. Design

2.1. Main actors

We have 4 different types of main actors:

Unregistered users
Users that use the application for the first time. They can browse Restaurants and look for available dishes, but to use any other service they must sign up.

Customers
Users that have an account. They can browse Restaurants, place orders, look for past orders and follow other people to get recommendations.

Restaurants
Special type of users. They can confirm orders placed to them and manage their available dishes.

Administrator
Special type of users. They have detailed information about general trends in the application.

2.1.1. Functional Requirements

Unregistered Users
Login – To access most of the services, an unregistered user must login.
Signup – Whoever doesn't have an account, can register any new one.
Find a restaurant – Even if not registered, any User can still search for Restaurants by zip code.

Customer
Modify personal data – Any User can change their personal information at any time.
Find a Restaurant – Customers can search Restaurants by zip code.

Filter – While searching for Restaurants, filters may be used to match only given categories of Restaurants. There is a special filter to get recommendation based on followed people.
Place orders – Once a Customer selects a Restaurant, he can choose what to order and then confirm.
Confirm order – Once considered dishes have been selected, a Customer must insert a payment method and payment number to place the order.
Past orders – Customers can see past orders along with recently made orders to see their status.
Followers – Customers can search for other people and follow them.
Rate – Customers can see and leave ratings on Restaurants.

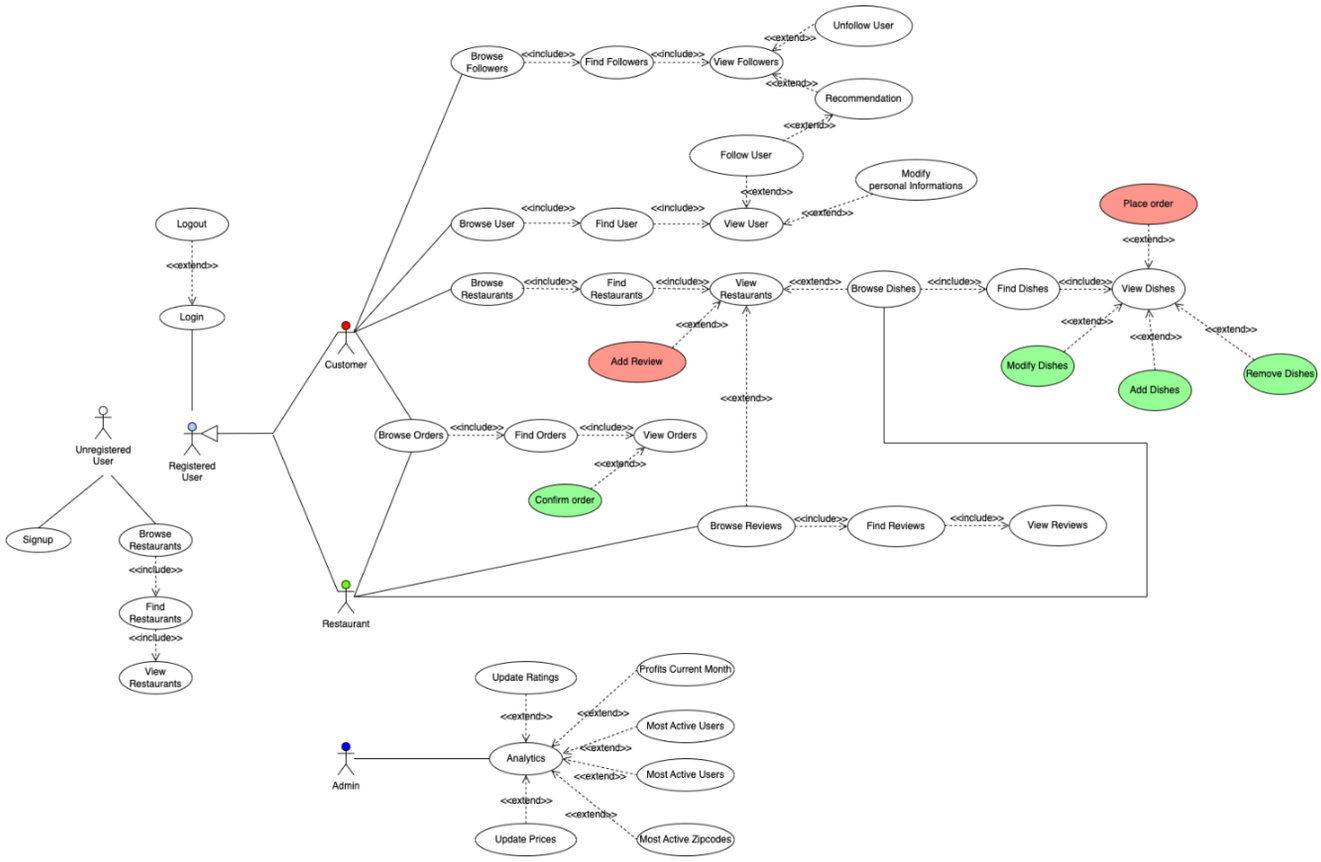
Restaurants
Login – To access their page, any Restaurant must login.
Pending orders – Restaurants can see not yet confirmed orders to see what they have to deliver.
Confirm orders – Once any order is ready for delivery, they can confirm the order to warn the user.
Modify dishes – Restaurants can modify, add, or remove available dishes from their personal page.
Past orders – Restaurants can consult the list of confirmed past orders.
Ratings – Restaurants can see ratings made to them by Customers.

Administrators
Access – To access the Admin page, a special link must be used.
Analytics – Admin can consult a special page with major information about the worldwide trend of the application.
Update ratings – They can use a special button to update Restaurants' ratings.
Update prices – They can use a special button to update Restaurants' price range.

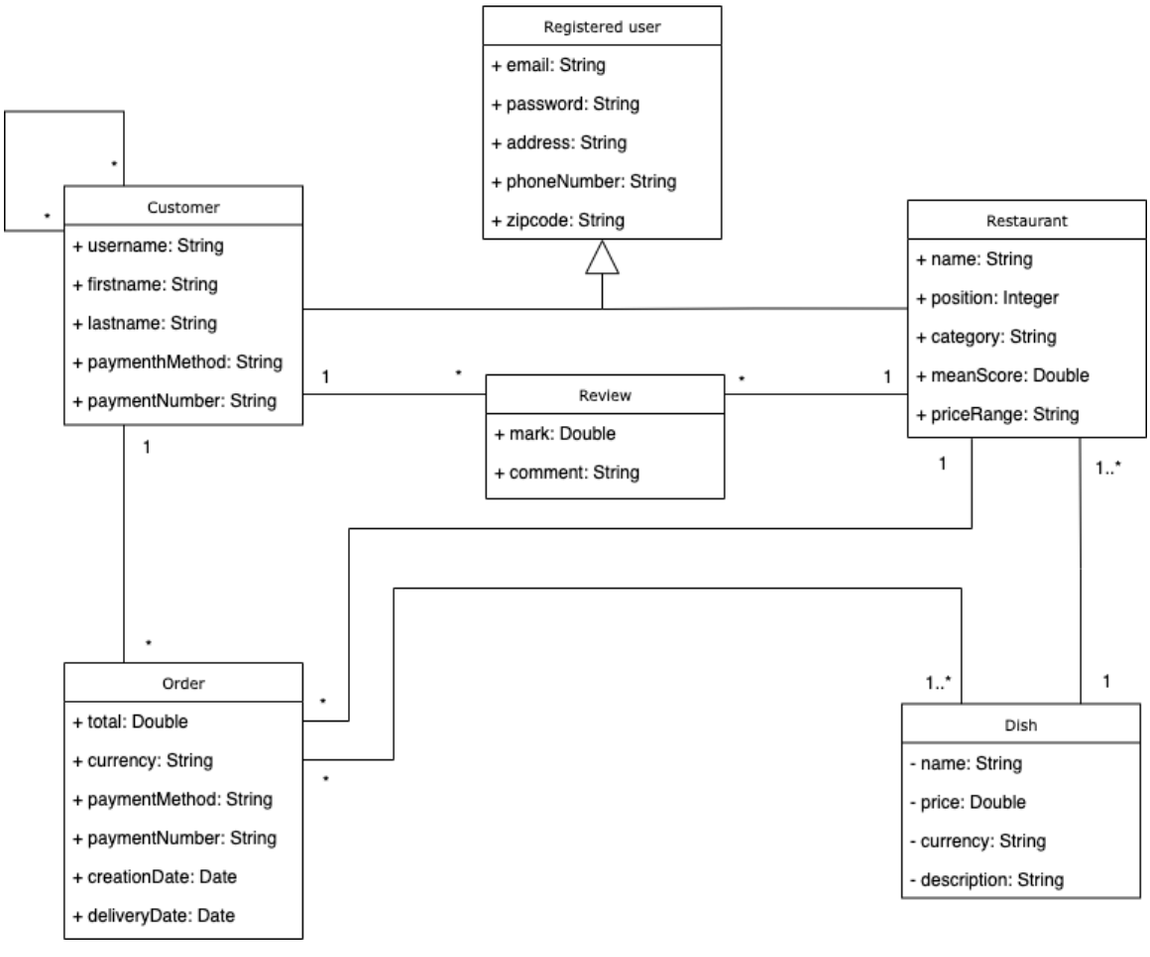
2.1.2. Non-Functional Requirements

The application must be available 24/7.
The application must respond to any user input within seconds.
The application must satisfy all requests even at high loads.
The application must be user-friendly with high usability.

2.1.3. Use Case Diagram



2.2. UML Class Diagram



2.2.1. Class Definition

Registered User

This is the base class for both Customer and Restaurant classes. It consists of all attributes in common between the two classes.

Customer

Derived class of Registered User. It represents the User that will use the app to make orders and all the other actions.

Restaurant

Derived class of Registered User. It represents the Restaurant that offers dishes to User and deliver Order to them.

Dish

This class represents what a Restaurant has to offer. They are also included in any Order.

Order

This class represents the list of Dish ordered by a Customer to a Restaurant. Holds all the useful information about an Order.

Review

This class holds the mark and the optional comment a User made towards a Restaurant.

2.3. Data Model

2.3.1. Document DB

The Document DB was chosen to manage a large amount of data with different fields in order to guarantee a flexible solution for different data gathered from different sources, in different layouts.

Restaurant

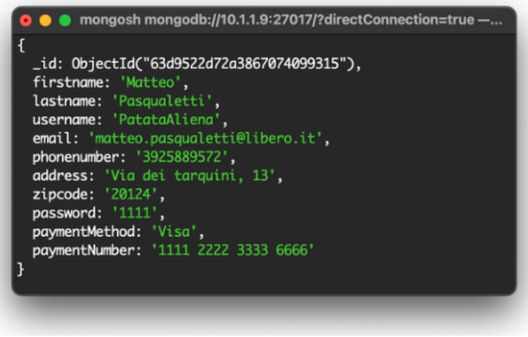
Since dishes available are always showed in combination with Restaurant's information, they have been placed embedded in the Restaurant owner document. Also, embedding them in Orders document, we solved two "many-to-one" relationships at once.

We chose to put only pending orders embedded in the Restaurant because the Restaurant itself is more likely to see them the most instead of both pending and confirmed orders together.
Because of the presence of pending orders both embedded in Restaurant and inside the orders collection, we had to pay attention while confirming the order, because this could cause inconsistency in data. In particular, when a new order is created or a pending order is confirmed, both order and Restaurant collections must be updated at the same time guaranteeing ACID transactions. This was solved by creating a transaction.

Without using available tools, if the second write doesn't commit, we had to remove also the first one and send a warn failure to the User.

On the other hand, a customer does not have any order embedded because we assumed that they will consult this list only a few times, not so much to justify another redundancy.

User



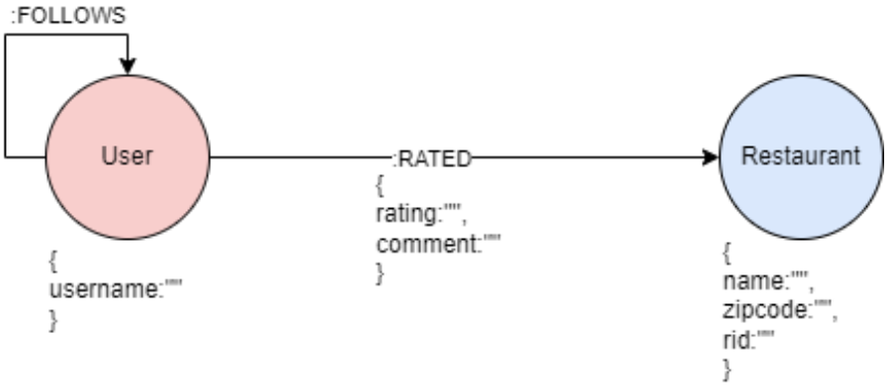
In the end, to avoid an explosion in order size if embedding orders, we decided to create a separate collection.

Order

Any order has embedded only the ordered dishes along with price, and quantity.

2.3.2. Graph DB

Graph DB has been chosen to have fast and flexible response to relationships between Users and Restaurants.



Nodes in the graph

User – Represent a User, only contains their nickname.
Restaurant – Represent a Restaurant, holds Id, name, and zip code. The redundancy on the zip code, and Id are used to make recommendations and retrieve more information from Document DB only if the user wants to.

Relationships

FOLLOW – Any User can follow another. One way relation. Doesn't contain any attributes.
RATED – Any User can rate a Restaurant. One way relation. It holds the given mark and the text if present.

2.4. Database Design

According to non-functional requirements, we should guarantee high-availability and partition protection. Indeed, we oriented our application on the AP edge of the CAP theorem at the cost of consistency.

To guarantee AP features, we configured our 3 virtual machines to have one primary and 2 replicas to ensure partition protection. To balance the load and guarantee high-availability, we configured our cluster to allow secondary replicas to satisfy requests if the primary is buys, and to commit any write operation after one successful write.

2.4.1. Replicas

3 replicas for Document DB.
1 replica for Graph DB.
Only 1 write is needed to commit any write operation.
Read preference on Nearest.

Write concern

To guarantee a low-latency response and high availability of the application, we chose that any write operation may be considered completed when one write is complete. Our global write concern is set as W1.

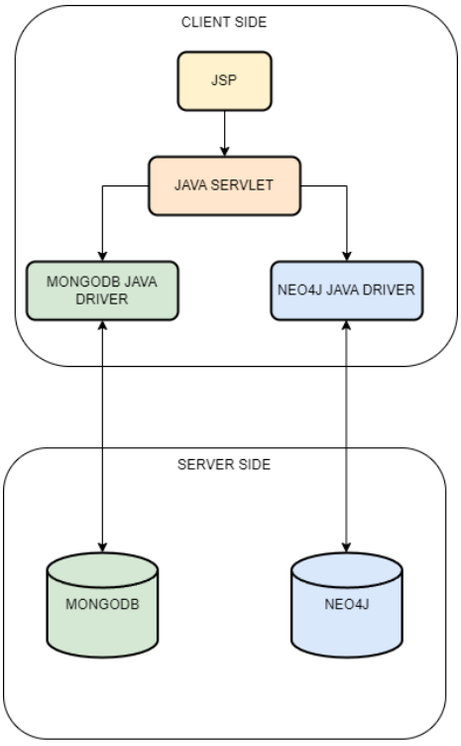
Read concern

We decided to set the read preference on Nearest replica, because we wanted to guarantee low latency, as we considered Neo4Food more oriented on reads than on writes.

2.4.2. Sharding

On our case, the most proper way to shard our database is to partition on zip codes. Most of the operations are made by zip code, so any related read-write operation or any aggregation can be made directly on the same server without wasting time searching in which server the information the user needs is.

2.5. Architecture



2.5.1. Client-Server

Client

The client only consists in 1 module:

Front-end: web pages requested and offered by the website to make proper requests.

Server

The server consists in 2 modules:

Middleware: Web servlet that makes possible the communication between client and back-end. It's also responsible for managing requests from a User.

Back-end: this module receives requests from the middleware, manages databases and offers responses for the user.

2.5.2. Inter-Database Consistency

Because we used two different types of databases, there is some data that can cause inconsistencies if not well handled between the two databases. In our case, since we tried to keep redundancies to the minimum, the only operations that may cause inconsistencies are:

- *Add user*

- *Remove user*
- *Add restaurant*
- *Modify restaurant*
- *Remove restaurant*

As discussed in paragraph 2.3.1, we have some redundancies on each database. Regarding Inter-Database ones, we must ensure that when creating, removing, or modifying any entity on MongoDB, we do the same correctly on Neo4J.

If any operation fails in the second step, we have to perform a Rollback.

3. Implementation

In this section we describe the classes created for the application and main relevant queries.

3.1. Main packages and classes

It.unipi.lsmsdb.neo4food.constants
Constants: class that contains constants used in the project.

It.unipi.lsmsdb.neo4food.dao.mongo
BaseMongo: abstract class that implements connection and access to MongoDB.
AdminMongoDAO: class that contains all "Admin-class" queries to Mongo. Extends BaseMongo.
AggregationMongoDAO: class that contains all aggregations made on Mongo. Extends BaseMongo.
OrderMongoDAO: class that contains all operations that involves Orders. Extends BaseMongo.
RestaurantMongoDAO: class that contains all "Restaurant-class" queries. Extends BaseMongo.
UserMongoDAO: class that contains all "Customer-class" queries. Extends BaseMongo.

It.unipi.lsmsdb.neo4food.dao.neo4j
BaseNeo4J: abstract class that implements connection and access to Neo4J.
SocialNeoDAO: class that holds all queries based on following and comments. Extends BaseNeo4J.
SupportNeoDAO: class that contains queries to keep consistency between Mongo and Neo4J. Extends BaseNeo4J.
UtilityNeoDAO: class that contains all special queries made by admin. Extends BaseNeo4J.

It.unipi.lsmsdb.neo4food.dto
AnalyticsDTO: class used to transfer data about analytics from middleware to front-end.
CommentDTO: class used to transfer data about comments from middleware to front-end.
DishDTO: class used to transfer data about dishes from middleware to front-end.
ListDTO: class used to transfer list of DTO objects from middleware to front-end.
OrderDTO: class used to transfer data about orders from middleware to front-end.
RestaurantDTO: class used to transfer data about restaurants from middleware to front-end.
UserDTO: class used to transfer data about user from middleware to front-end.

It.unipi.lsmsdb.neo4food.model
Comment: class that holds comment's information.
Dish: class that holds dish's information.
Order: class that holds order's information.
RegisteredUser: class that holds base users' information.
Restaurant: class that holds restaurant's information

User: class that holds user's information

It.unipi.lsmsdb.neo4food.service

ServiceProvider: static class that holds instances of DAO classes to use the same connection for different queries.

It.unipi.lsmsdb.neo4food.servlet

Admin: servlet class that satisfies all requests from Admin.
Checkout: servlet that satisfies all requests regarding order placement and confirmation.
Login: servlet that satisfies both user and restaurant login request and user signup.
Logout: servlet that satisfies both user and restaurant logout.
PersonalPage: servlet that satisfies all user's requests regarding their personal page.
RestaurantPage: servlet that satisfy all restaurant's requests regarding their personal page.
SearchRestaurants: servlet that satisfies all requests regarding searching restaurants.
Social: servlet that satisfies all "Social-like" requests.

It.unipi.lsmsdb.neo4food.utility

Utilities: class that contains static methods used in the whole project.

3.2. Most relevant queries

In this section we report some of the most important queries for both MongoDB and Neo4J.

3.2.1. MongoDB

There we will discuss about the two most used queries in the whole project:

Index Analysys	Execution Time	Keys Examined	Docs Examined
_id	739	602701	602701
user	2	19	19

The first one retrieves the order history of a customer. Because of the huge number of orders in this collection, creating an Index on the username improved the performance of retrieving orders dramatically. We used an Index on the restaurant ID too, to also improve performance for restaurants.

Index Analysys	Execution Time	Keys Examined	Docs Examined
_id	79	38581	38581
zip_code	0	7	7

The second most used is searching for a restaurant by zip code. There was not a significant improvement in Execution time, but the System had to search between way less documents and keys.

3.2.2. Neo4J

```
MATCH (u:User)-[s:RATED]→(r:Restaurant)
WITH u1.username as username
COUNT(s.rating) as nrating
AVG(s.rating) as avg_rating
ORDER BY nrating DESC
RETURN username, avg_rating as avg, nrating
LIMIT 10
```

This query is used by the Admin. It retrieves the top 10 Customers which write reviews the most, along with respective average score.

```
MATCH   (u1:User)-[:FOLLOWS]→
        (u2:User)-[s:RATED]→(r:Restaurant)
WHERE   u1.username = $user AND r.zipcode = $zipcode
AND NOT (u1)-[:RATED]→(r)
WITH r, COUNT(s.rating) AS nrating,
        AVG(s.rating) AS avg_rating
ORDER BY nrating DESC, avg_rating DESC
RETURN  r.rid AS rid, r.name AS name, avg_rating AS avg
LIMIT 5
```

The following is used by the system to recommend to a Customer a Restaurant inside given zip code, which were not rated by him, and was rated by his following users. The number of following people and average score gave by them condition the result.

```
MATCH   (:User)← [f:FOLLOWS]-(:User)
RETURN  u.username AS username,
        COUNT(f) AS nfollowers
ORDER BY nfollowers DESC
LIMIT 10
```

This simple but particular query is used by Customers and the Admin to find out who are the influencers. In other words, it returns the top 10 most followed users.

```
MATCH (r:Restaurant)←[a:RATED]-(:User)
WITH r, avg(a.rating) as score
RETURN r.rid as rid, score as avgScore;
```

The next query is useful for made-up a recommendation system based on the following user of the current user's fiends, i.e., return all the users that my friends follow but not by the current user.

```
MATCH (u1:User)-[:FOLLOWS]→
      (u2:User)-[:FOLLOWS]→
      (u3:User)←[f:FOLLOWS]-(:User)
WHERE NOT (u1)-[:FOLLOWS]→(u3)
AND u1.username = $user
WITH COUNT(f) AS nfollowes, u3
ORDER BY nfollowers DESC
RETURN nfollowers, u3.username AS username
LIMIT 10
```

3.3. Test and statistics

3.3.1. Aggregations Analysis

Faster Restaurant by Zipcode

The previous aggregation, return the faster restaurant in a given zip code, i.e., restaurants that take less time for delivery food.

Dish Mode

Considering all dishes from a particular restaurant, this query returns the most ordered dished ever

Best Dish of the Day

Considering all dishes from a particular restaurant, the previous query returns the most ordered dishes of the day.

Most Busy Hours

Taking into account all the orders of the last month from a given restaurant, we want to know what the busiest hours of the day are, counting the order by hour.

Month's Revenue

The previous query was used by the Administrator to find out the top restaurant of the month, based on their revenue.

Top 10 Zipcodes

The previous query was used by the Administrator to know the top zip codes, based on the activity of the users in terms of made orders.

The Usual

This query is particular, because it tries to emulate a restaurateur who answers the question: “I’ll take the usual, please”. We retrieve the most ordered pool of dishes for a given restaurant and user.

Most Expensive Dishes

The previous query it’s used by the Administrator to find out the most expensive dishes.

Average Price

The previous query was used by the Administrator for update, inside the collection restaurant, the attribute "*price_range*" that indicate how much it's expensive a restaurant.

3.3.2. Indexes Analysis

In this section we will show the differences for aggregations between not and using a custom index.

Restaurant Collection

Aggregation	Custom Index	Execution Time	Keys Examined	Docs Examined
Average price	price_range: 1	4464 (3829)	38581 (38581)	38581 (38581)
Most expansive dishes	price_range: 1	4685 (4205)	38581 (38581)	38581 (38581)

NB: The number between brackets indicates the result obtained using the index described in the second column.

Those 2 aggregations had no benefits by using any index, because they were aggregating the whole data in the collection.

In the end, we created two indexes in the Restaurant collection, which are "zipcode: 1" for searching and "email: 1" for Restaurant login. In total they weigh 0.5% of the total collection size (uncompressed).

We also created an index for similar purposes on Users collection, for similar performance improvements as shown in paragraph 3.2.1.

In that case, the index consists of 1% of total collection size (uncompressed).

Orders Collection

Aggregation	Custom Index	Execution Time	Keys Examined	Docs Examined
Faster restaurants by zip code	zipcode: 1	1158 (27)	602701 (1729)	602701 (1729)
Dish mode	RID: 1	767 (415)	602701 (752)	602701 (752)
Best dish of the day	RID: 1	879 (13)	602701 (52)	602701 (52)
Most busy hours	RID: 1, cDATE: 1	946 (0)	602701 (20)	602701 (20)
Month's revenue	RID: 1, cDATE: 1	791 (10)	602701 (415)	602701 (415)
Top 10 zip codes	zipcode: 1	951 (951)	602701 (602701)	602701 (0)
The Usual	user: 1, RID: 1	896 (6)	602701 (5)	602701 (5)

NB: The number between brackets indicates the result obtained using the index described in the second column.

As this collection is the most used one, we created a lot of Indexes for different purposes.
As said in paragraph 3.2.1, we have two indexes on "user: 1" and "restaurantId: 1" to have extremely fast Orders retrieval.

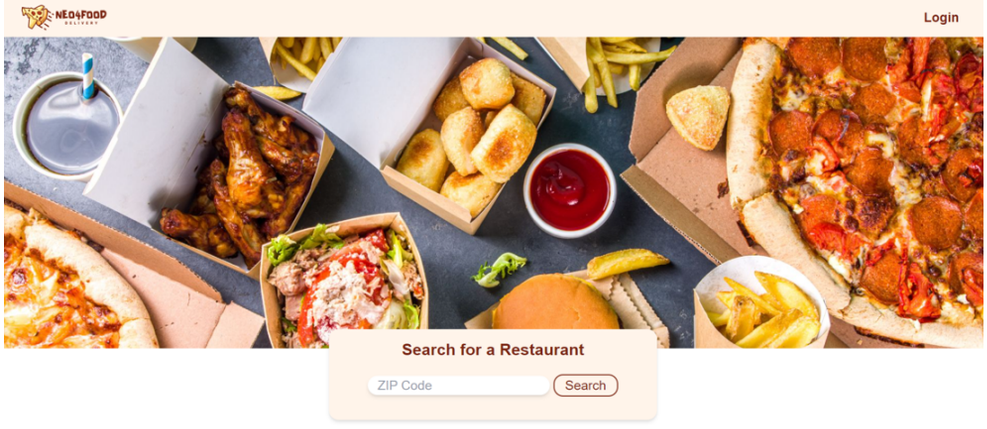
Excluding the Top 10 Zipcodes aggregation which has no significant performance improvements in using a custom index, the others are speed up by a lot by using them.
The main drawback in this case is that they weigh 15% of the total collection size (uncompressed).

4. Manual

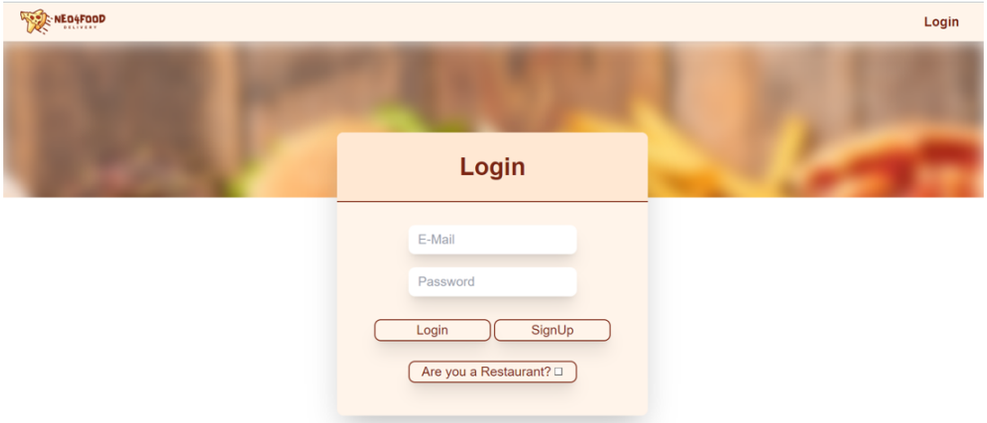
4.1. Customer

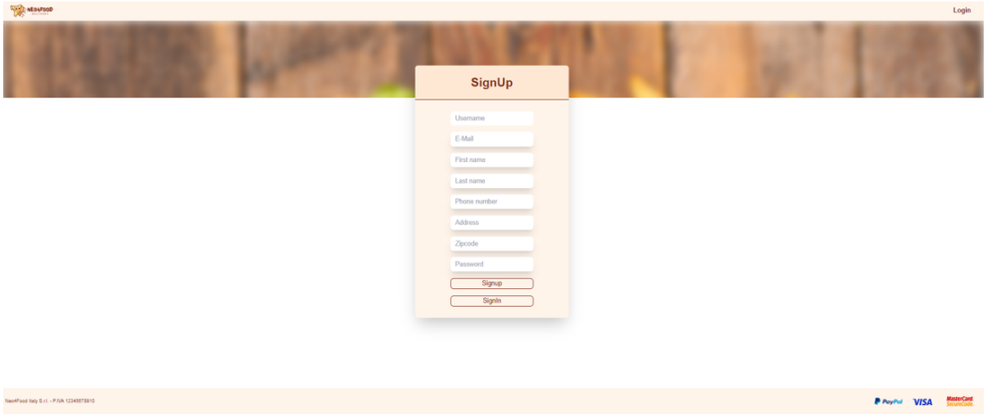
4.1.1. Registration and Login

To have access to the majority of services offered by the website, any user must login or sign up using the top-right button shown in the first displayed page.

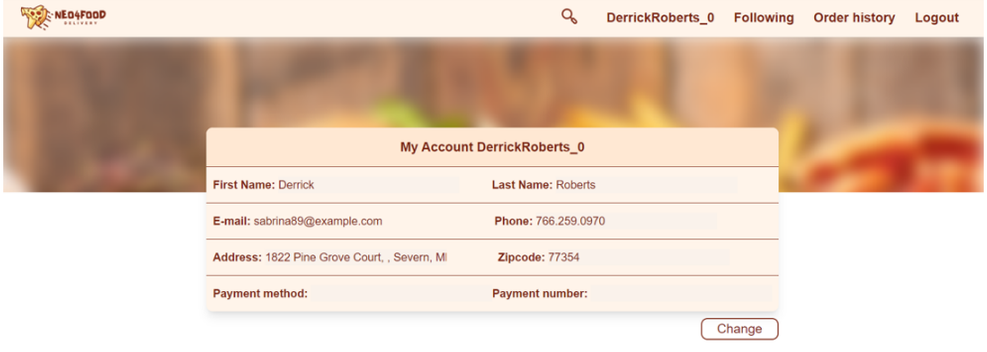


Once there, the user can insert the email and password and press login or can press the signup button to access the register form to create a new user. All fields are mandatory to register. Some information may be changed later.





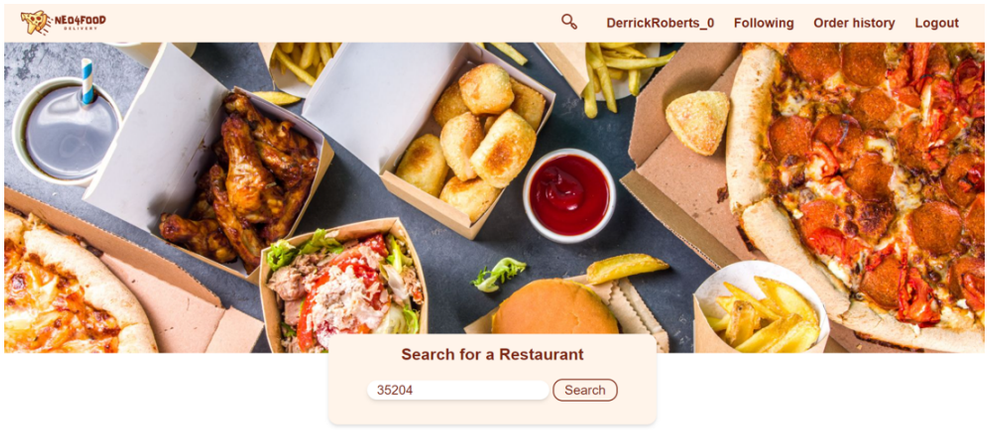
Once registered or logged in, the user is automatically redirected again to the main page. By clicking on their username on the top of the screen, they will be redirected to their personal page, where they can change personal information.



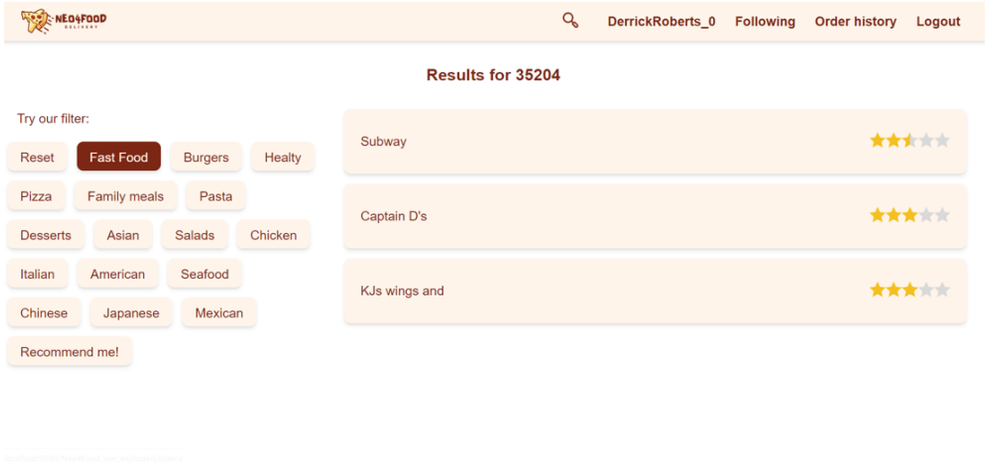
There, if the user adds a payment method and payment number, they will be later automatically placed during the checkout of an order.

4.1.2. Restaurants searching

In the main page the user can insert the desired zip code to find all available restaurants in that zip code, if any is present. They will be displayed as names and average ratings given by the community.



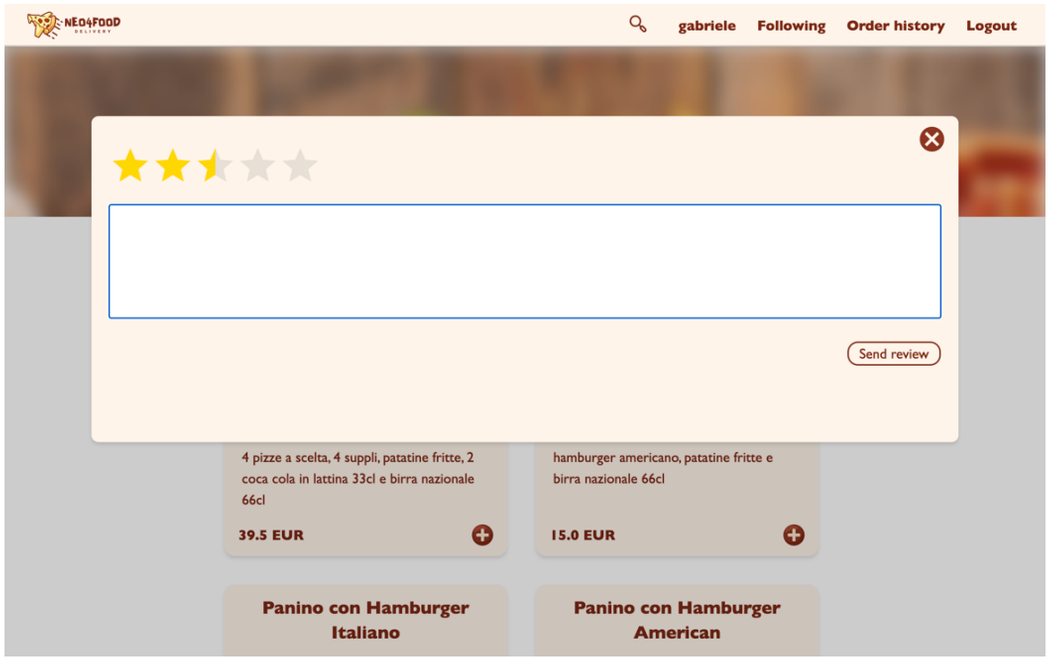
Once there, the user can browse and choose between restaurants. In addition, the user can filter by choosing one of the filters offered on the left of the screen.



Using the special filter “Recommend me!”, the system will show, if present, restaurants rated by your followers along with average score given by the latter.

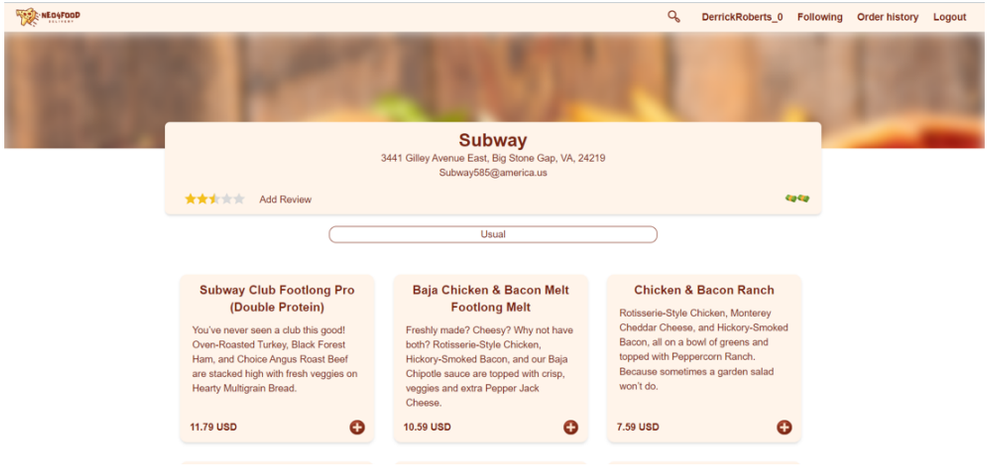
4.1.3. Reviews

A registered user can leave a review or a comment to the restaurant. He has to click on the stars of the restaurant page for see all the reviews, or he can click on the button “add review” for leave a rate and, if he wants, a comment.

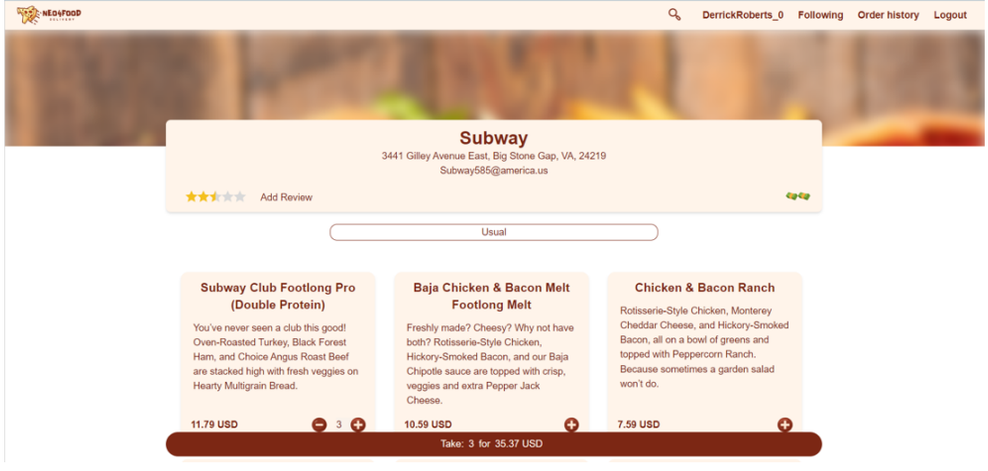


4.1.4. Order placement

Once the user selects a restaurant, the page with more info and available dishes will be shown. There the user can click on + to add dishes to the order. In addition, by clicking on the stars, he can view reviews given by other users. By clicking on the proper label, instead, they can write a review themself.

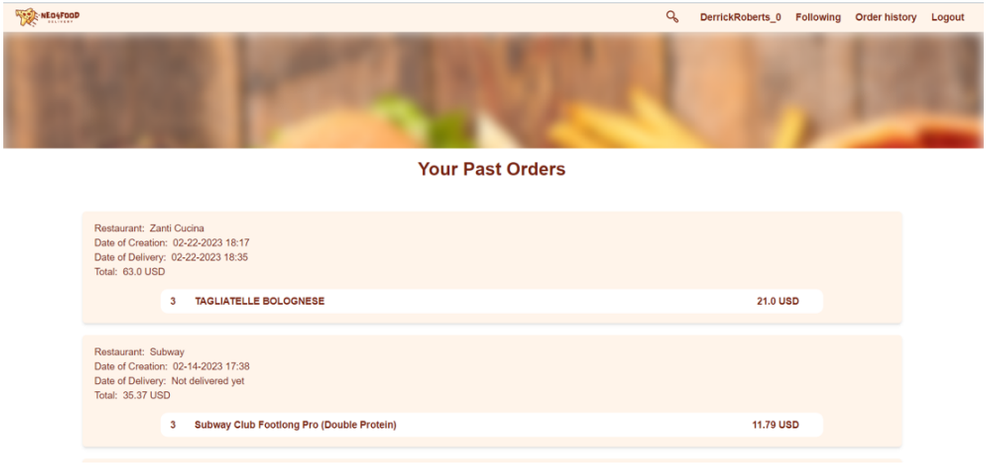


Once the user is satisfied with the order, they can continue with the checkout by pressing the big red button on the bottom of the screen.



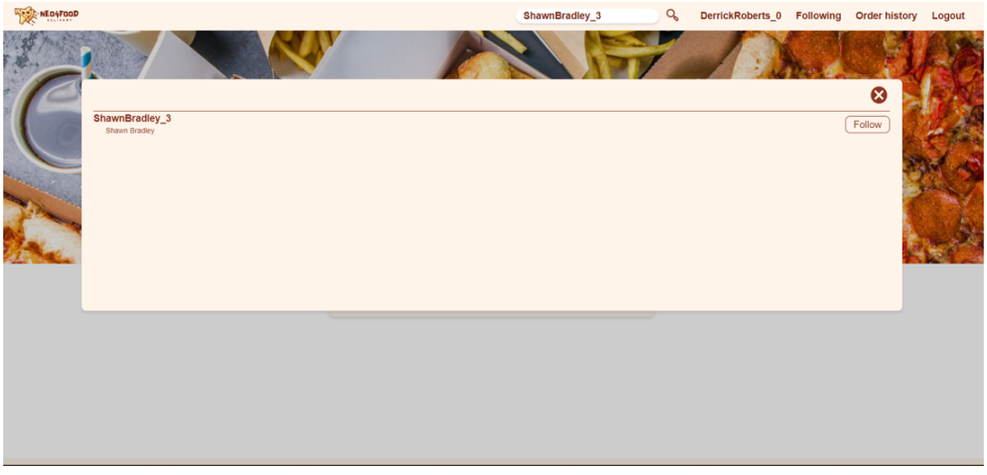
On the next page, the user can see a summary of the order and continue with the checkout by clicking on the Confirm button. As said previously, the user doesn't need to put a payment method and number it he set it on their profile.

By clicking Order history on the top right of the screen, they can see all ever-made orders.



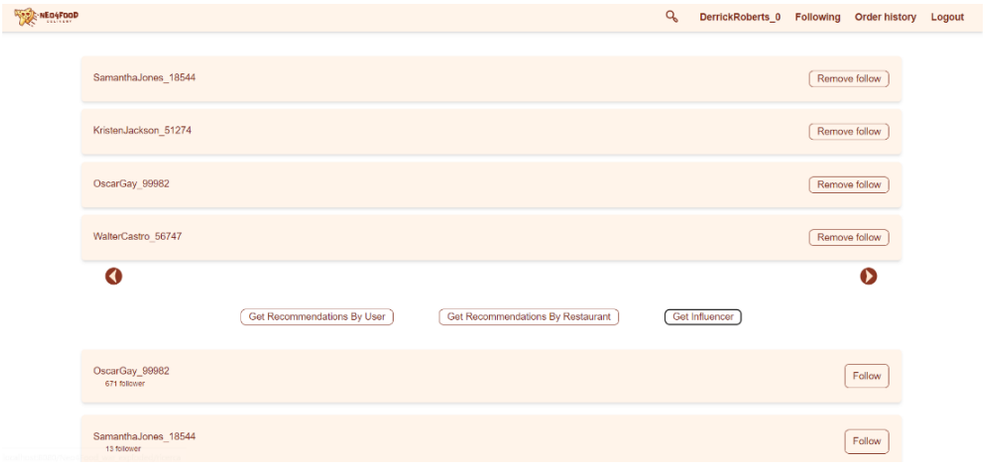
4.1.5. Follower feature

On the top of the screen there is a small lens. If hover with the mouse, it can used to find username of users to follow them:



Also, by clicking on "*Following*" button on top right of the screen, they will be redirected to a page where they can see all the people followed. In addition, by using the other buttons, they will get some recommendations on other users they might be interested in following.

One of the most important feature available is the button to see all influencer users in the system.



Users can be unfollowed at any time by clicking the unfollow button.

Remember that the following people are important to get recommendations on restaurants.

4.2. Restaurant

4.2.1. Login

Because Restaurants are special entities that offer a service for money, they are not able to register on the website from nothing. By choice, they must contact an admin to create an account.

To login, they can press the button on top right of the screen. To login they can use the same form used by Customers, but in addition they must check the box to login.

4.2.2. Orders confirmation

Once logged in, they will be redirected to the page they will use the most. There they can view reviews by clicking on the proper button.

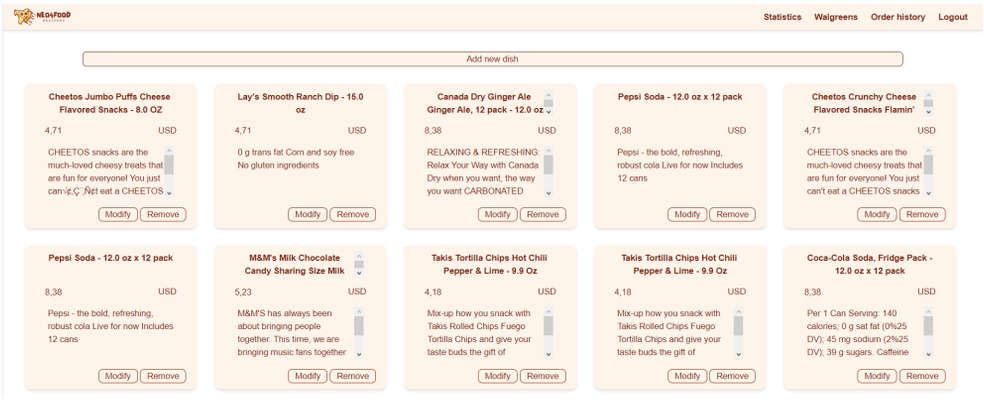
Below, there are displayed all pending orders along with some useful information.

By pressing the proper button, they can confirm the order, marking it as ready.

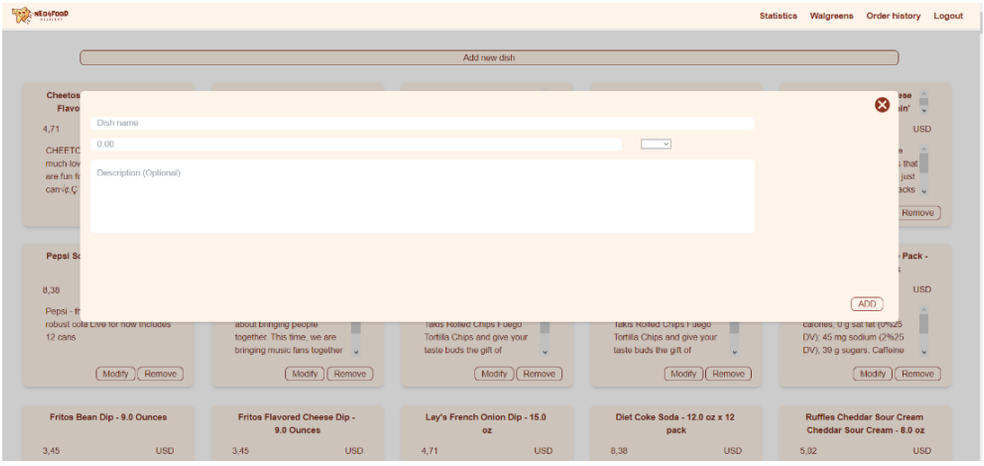
By clicking on Order history in the top right of the screen, they can see all ever-received orders.

4.2.3. Dishes management

Clicking on restaurant name, they will be redirected to a page where they can manage all available dishes.



There, they can either add, modify, or remove dishes.



4.2.4. *View Reviews*

In a comparable manner for the user, even the restaurant can look at his reviews.

4.2.5. *View Analytics*

On the header of the page, the restaurant can use the button “Statistics”, to see some statistics on his activity.

4.3. Admin

4.3.1. Access method

As the access for the admin is meant to be used by more technical people, you can only access those features only if in possess of a special link with authorization token.

4.3.2. Features and Analytics

There the admin can have an overview of the general trend of the application, by seeing the number of restaurants, orders, and users in the system.

By clicking two special buttons, he can start the process of calculating new ratings and price range of restaurants.

In addition, he has 5 buttons to load some of the analytics created for the system: Most expansive dishes, zip codes where Neo4Food is used the most, Restaurants which gained the most during last 30 days, most active registered users and Restaurants with best Delivery Time by zip code.