UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Large-Scale and Multi-Structured Database

Second Chance

Project Documentation

Adelmo Brunelli

Martina Marino

Roberta Matrella

Sommario

Abstract

SecondChance is a Java oriented application developed by using different No-SQL databases.

In this documentation are described the main aspects of the development process from the requirements through the models selected and the strategies adopted. In the implementation chapter is explained the general organization of the code and also some relevant functions are reported. For what concerns the databases chosen, there is a description of how data are organized and of the most important queries that exploit them. Some indices were introduced in order to speed up of the application and the performance is there discussed and measured.

In conclusion there is a brief discussion about sharding solutions to discuss what can be the best way to realize database partition according to this application characteristics.

GitHub link to the repository:

https://github.com/martimarino29/Large-Scale-Project

Introduction

SecondChance is an application that gives you the possibility of buying and selling new or vinted items. A normal user can use it as an e-commerce to update his/her products, make purchases, browse the feed and do searches about articles and users. There is also an admin panel in which the administrator can suspend users and obtain statistical results about data.

It was developed using IntelliJ IDEA as IDE and Java as programming language with the support of FXML files. The dataset was taken from different sources that were merged using some Python scripts and all the data were reorganized and stored in two different databases: MongoDB and Neo4j. In the first are stored all the information regarding users, insertions, orders while the second contains all that regards social connections and suggestions.

Analysis

Actors

There are two types of actors that can interact with the application:

- **Normal user**: can act both like a buyer and a seller.
- **Admin**: can visualize statistics, suspend users and delete an insertion.

Functional requirements

- An **unregistered user** can only sign up and then sign in.

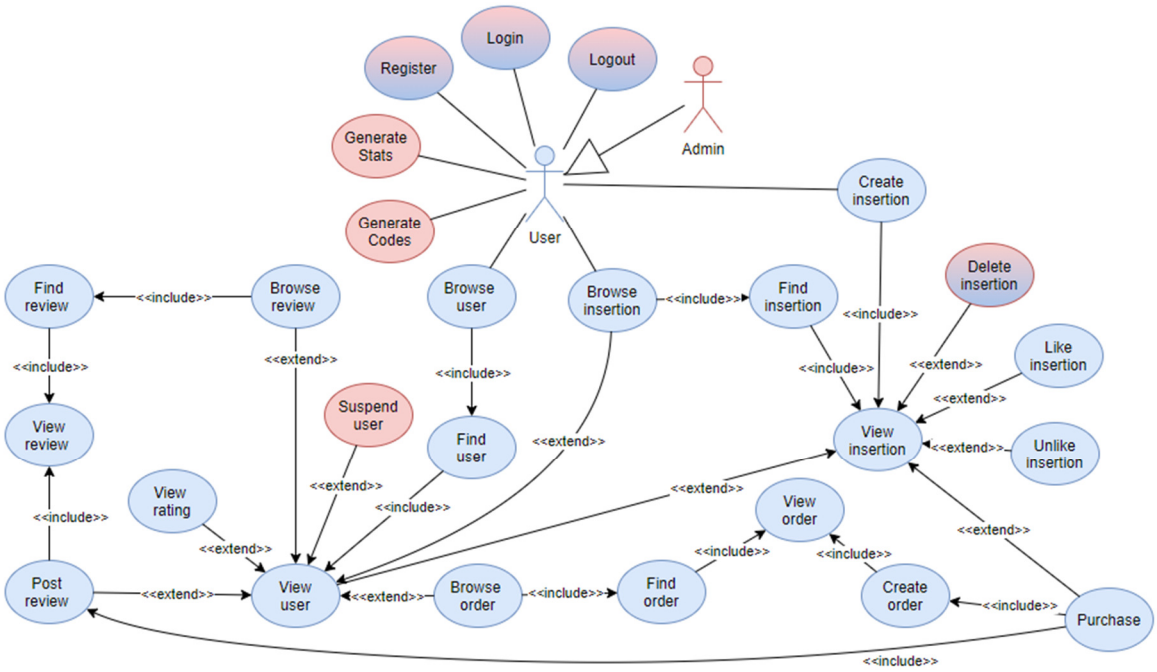
- A **registered user** can:
    - *Sign in* in the application
    - *Visualize* home
        - Visualize viral insertions
        - Visualize feed
    - *Visualize* his/her profile
        - Visualize personal information
        - Browse his/her insertions
        - Visualize insertions he/she is interested in
        - Browse his/her orders
        - Visualize followers and following users
        - Add credit to the personal balance account
    - *Create* a new insertion
    - *Delete an insertion of your own*
    - *Search* insertions by seller, brand or using filters
    - Visualize an insertion
        - *Like* an insertion
        - Buy relative item
    - *Search* a user
    - *Follow/unfollow* a user
    - *Visualize* suggested users
    - *Logout* from the application

- The **administrator** is able to:
    - *Visualize* statistics (specifying a k parameter)
        - View top k rated user per country
        - View top k interesting insertions per category
        - View top k viewed insertions per category
        - View top k users with more purchased items
        - View top k users with more sold items
        - Number of likes per category
    - *Suspend* a user
    - *Delete* an insertion

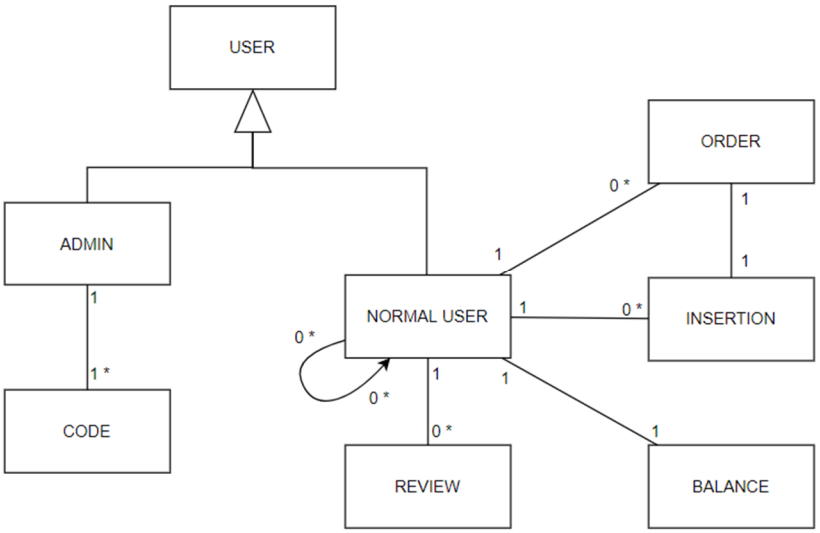
Non-functional requirements

- The application must be user-friendly so a GUI has been implemented
- The username must be unique
- The password must be stored in a protected form
- The code must be modular, easy to change and maintain
- The data has to be consistent among the different databases
- The data must be consistent in order to display always the correct information
- The system needs to be tolerant to data lost and to single point of failure
- The application should guarantees an adequate quality of performance

Use Cases



Class Analysis



User class

FIELD	TYPE	DESCRIPTION
address	String	Address of the user
city	String	City of the user
country	String	Country of the user (Italy, Canada, Spain, Austria, Germany, France, Brazil, Netherlands, Poland, Ireland, United Kingdom (Great Britain))
email	String	Email of the user
img_profile	String	Name of the image file
name	String	Name and surname of the user
password	String	Password chosen by the user
suspended	String	'Y' if the user is suspended, 'N' if not
username	String	Username chosen by the user

Insertion class

FIELD	TYPE	DESCRIPTION
brand	String	Brand of the item
category	String	Category of the item (clothing, accessories, bags, beauty, house, jewelry, kids, shoes)
color	String	Color of the item
country	String	Country in which the item is sold (Italy, Canada, Spain, Austria, Germany, France, Brazil, Netherlands, Poland, Ireland, United Kingdom (Great Britain)
description	String	Description of the item
gender	String	Gender of the item (F, M, U)
image_url	String	Link of the image of the item
interested	Int32	How many people are interested in the insertion
price	Double	Price of the item
seller	String	Username of who sells the item
size	String	XS, S, M, L, XL, U
status	String	Status of the item (new, excellent, good, used, very used)
timestamp	String	Timestamp of the publication
views	Int32	How many people clicked on the insertion

Order class

FIELD	TYPE	DESCRIPTION
buyer	String	Who bought the item
timestamp	String	Timestamp of the conclusion of the order
user	String	Username of the buyer or the seller

Admin class

FIELD	TYPE	DESCRIPTION
username	String	Admin's username
password	String	Admin's password

Balance class

FIELD	TYPE	DESCRIPTION
username	String	Username of the user
credit	double	Credit of the user

Codes class

FIELD	TYPE	DESCRIPTION
code	String	Code of the card
credit	double	Credit of the card

Design

Dataset

Sources

In order to create a realistic large scale database, different sources were merged together and uniformed to have coherent data.

All information about users were randomly generated online, while all the insertions were taken from three different sources:

https://data.world/wordlift/shopping-demo/workspace/file?filename=amazon-fashion.csv

https://data.world/jfreex/products-catalog-from-newchiccom .

https://data.world/promptcloud/amazon-australia-product-listing/workspace/project-summary?agentid=promptcloud&datasetid=amazon-australia-product-listing

The reviews were taken from *https://www.kaggle.com/asmaoueslati/womensclothingecommerce* .

Volume and variety

For what concern volume we have that the original overall dataset was about 50 MB composed by:

- ~ 100 k users
- ~ 116 k insertions
- ~ 1k codes
- ~ 100 k balances

The variety is given by the fact that the final dataset is a merged version of the different sources.

Data organization

All the information are stored into two different databases: MongoDB as document database and Neo4j as graph database.

Neo4j was chosen as second database because the application gives the possibility to follow/unfollow other users and receive suggestions about popular sellers or users that can be interesting considering those who are followed by the user's followed ones and coming from the same country.

MongoDB stores the basic information about Users, Insertions and Orders, while in Neo4j there are all the relationships between entities, like social interactions and posting.

MongoDB – Documents organization

In the document DB are stored four different collections:

- User
- Insertion
- Code
- Balance

The **user** collection is organized in this way:


```
"_id": {
    "$oid": "61fd8cc3edf5f2f4bd1366bc"
},
"address": "Ap #958-5394 Aliquam Ave",
"city": "Ebenthal in Kärnten",
"country": "Austria",
"email": "pellentesque.ultricies.dignissim@protonmail.couk",
"img_profile": "image.png",
"name": "Elisa Littel",
"password": "81dc9bdb52d04dc2036dbd8313ed055",
"suspended": false,
"username": "Aaltje",
"reviews": [{
    "timestamp": "2021-12-23 20:57:14 ",
    "reviewer": "Cadalso",
    "title": "Another tiny success",
    "text": "Tiny are experts at making busy bohemian shirtdresses that look casual but retain a feminine drape.  it skims over my trouble spots without adding bulk.
    "rating": 5
}],
"rating": 5,
"sold": [{
    "timestamp": "2021-06-22 15:56:48",
    "buyer": "Cadalso",
    "insertion": {
        "image": "https://imgaz1.chiccdn.com/thumb/view/oaupload/newchic/images/C4/3D/d856b0c6-9e36-4798-874e-6717b0780e2d.jpg",
        "price": 45.75,
        "size": "XL",
        "status": "new",
        "category": "clothing"
    }
}],
"purchased": [{
    "timestamp": "2021-06-27 04:42:20",
    "seller": "Cubillas",
    "reviewed": true,
    "insertion": {
        "image": "https://imgaz1.chiccdn.com/thumb/view/oaupload/ser1/newchic/images/59/76/89dd637b-d567-4d17-aead-ee83b5b8e68b.jpg",
        "price": 49.49,
        "size": "S",
        "status": "very used",
        "category": "clothing"
    }
}, {...
}, {...
}, {...}]
```


In the user document are stored all the personal information, the credentials for the login and a suspended field. If the admin has suspended a user he/she can't join the application. For every user there are three arrays for storing the reviews, all the sold items and the purchases he/she did. The rating field is the average value of all the ratings of the reviews.

NOTE: in this application the review is associated to the user as seller and is not correlate to the specific item sold. The reason is that the insertions are relative to a unique object that cannot be bought again. In fact, most of them are vinted, so a review about an article would have been a pretty useless information.

The **insertion** collection is organized in this way:


```
{
    "_id": {
        "$oid": "61fc54c2ce8ed77a2c6b5ad4"
    },
    "brand": "Alien Storehouse",
    "category": "house",
    "color": "Pink",
    "country": "Austria",
    "description": " 5Pairs Girl Clip-on Earrings Child Pendant Ear Clips for Pretend Play Princess Girl Birthday Gift, H ",
    "gender": "U",
    "image_url": "https://images-na.ssl-images-amazon.com/images/I/41-NOgsUf2L._.jpg|https://images-na.ssl-images-amazon.com/images/I/510rIQG2ssL._.jpg",
    "interested": 42,
    "price": 23.06,
    "seller": "Aali",
    "size": "M",
    "status": "very used",
    "timestamp": "2021-07-15 02:48:47 ",
    "views": 66
}
```


In each document there are all the details of the item to sold and the number of users interested in it and the number of views it has received.

The **code** collection is organized in this way:


```
{
    "_id": {
        "$oid": "61fc54aecb8caf31782a6140"
    },
    "code": "IFFVJV88SF",
    "credit": 50
}
```


For each code is associated the relative value.

The collection **balance** is designed in this way:


```
{
    "_id": {
        "$oid": "61fd081b530448255c65e738"
    },
    "username": "Aadil",
    "credit": 0
}
```


This collection has associated, for every user, the relative credit of the account.

Neo4j – Nodes organization

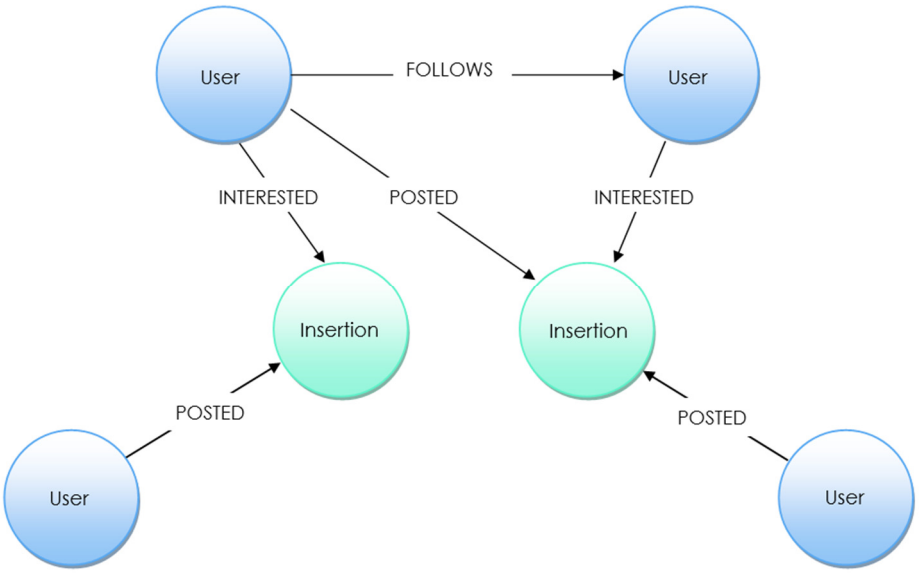
In the graph database there are about 219.000 nodes and 760.000 relationships.

The structure of the nodes with their relations is described below:

- A **User** node represents a registered user with the relative username and country.
- An **Insertion** node represents the insertion entity with the id and category.

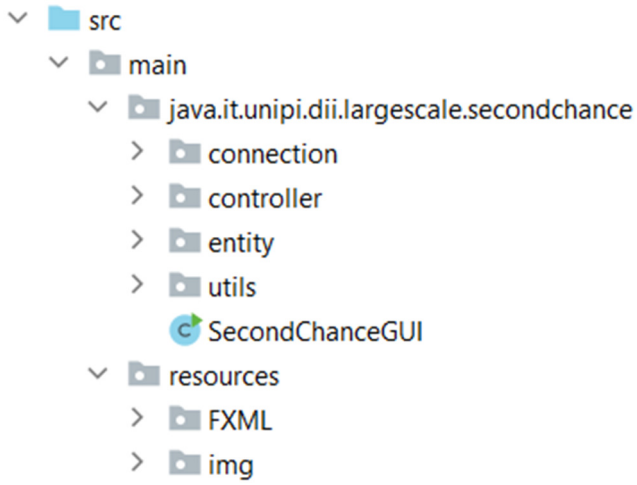
About relationships there are:

- **User – [:FOLLOWS] -> User**, which represents a user that follows another.
- **User – [:POSTED] -> Insertion**, which represents an insertion published by a user.
- **User – [:INTERESTED] -> Insertion**, which represents that a user has put the insertion in his/her favourites list.



Implementation

As we can see in the figure below, the root package contains five children: connection, controller, entity, utils.



Package structure

The *entity* package contains the classes needed to represent in OOP style every entity of the database: User, Order, Review, Insertion, Admin, General User.

These classes contain only getter and setter methods to load and read data.

The *controller* package contains the classes that control the data flow and update the view whenever data changes. These classes handle the corresponded FXML document used to implement the graphic interface.

The *connection* package contains the driver for MongoDB and Neo4j. The classes ConnectionMongoDB and ConnectionNeo4jDB contain as a member the queries implemented for the MongoDB and Neo4j database.

The class Util, inside the util package, contains methods util for the other classes, such as the control of the images. The class Session, contained into the same package, has the function to maintain information about the logged user.

Entities structure

```
public class Insertion {

    String id;
    String category;
    String description;
    String gender;
    double price;
    int interested;
    int views;
    String status;
    String color;
    String size;
    String brand;
    String country;
    String image_url;
    String timestamp;
    String seller;
```

```
public class Review {

    String reviewer;
    String seller;
    String text;
    String timestamp;
    String title;
    int rating;
```

```
public class Order {

    String buyer;
    String timestamp;
    Insertion insertion;
    boolean reviewed;
```

```
public interface GeneralUser {

    public void setUsername(String username);

    public void setPassword(String username);

    public String getUsername();

    public String getPassword();

}
```

```
public class User implements GeneralUser {

    String email;
    String username;
    String password;
    String name;
    String country;
    String city;
    String image;
    String address;
    String suspended;
    double rating;
```

```
public class Admin implements GeneralUser {

    String username;
    String password;
```

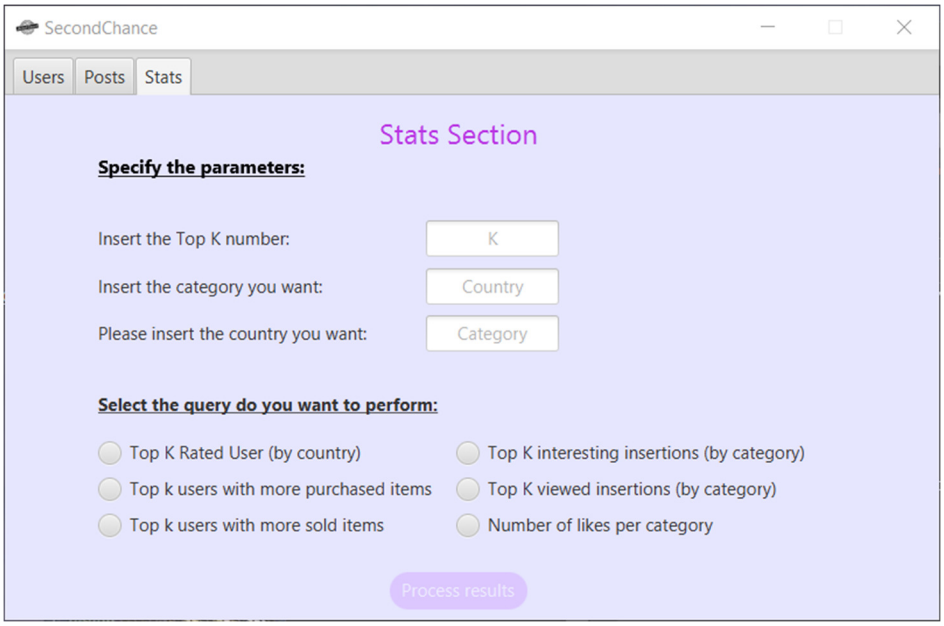
MongoDB: queries and analytics

Queries

The function is used to get the viral insertions to show in the home. These are the k insertions with the highest interested number and views.

```
public ArrayList<Document> findViralInsertions(int k) {

    ArrayList<Document> insertions = new ArrayList<>();
    Bson sort = sort(descending( ...fieldNames: "interested", "views"));
    Bson limit = limit(k);

    AggregateIterable<Document> r = insertionColl.aggregate(Arrays.asList(sort ,limit));

    for (Document document : r)
        insertions.add(document);

    return insertions;
}
```

Statistics



This panel can be accessed only by the administrator and allows to show some interesting statistics about the usage of the application.

The function below is used to retrieve the k best rated users of a specific country.

```
//find the k users with higher rate for a specified country
public ArrayList<Document> findTopKRatedUser(int k, String country) {

    ArrayList<Document> array = new ArrayList<>();

    Bson limit = limit(k);
    Bson project = project(fields(excludeId(), include( ...fieldNames: "username"), include( ...fieldNames: "rating")));
    AggregateIterable<Document> aggr  = userColl.aggregate(
            Arrays.asList(
                    Aggregates.match(Filters.eq( fieldName: "country", country)),
                    Aggregates.match(exists( fieldName: "rating")),
                    Aggregates.sort(descending( ...fieldNames: "rating")),
                    project,
                    limit
            )
    );

    for (Document document : aggr)
        array.add(document);

    return array;
}
```

This function returns the k users that have more sold orders or more insertion published, depending on the value of the variable choice.

```
//find the users with most purchased or sold orders
public ArrayList<Document> findMostActiveUsers(int k, boolean choice) {

    ArrayList<Document> orders = new ArrayList<>();
    AggregateIterable<Document> aggr;
    // true = select the top k users with more purchased orders
    if(choice) {

        Bson match = match(exists( fieldName: "purchased.0"));
        Bson project = project(fields(computed( fieldName: "count", new Document("$size", "$purchased")), include( ...fieldNames: "username"), include( ...fieldNames: "purchased")));
        Bson sort = sort(descending( ...fieldNames: "count"));
        Bson limit = limit(k);
        aggr = userColl.aggregate(
                Arrays.asList(
                        match, project, sort, limit
                )
        );
    }
```

```
    else        // false = select the top k with more sold orders
    {
        Bson match = match(exists( fieldName: "sold.0"));
        Bson project = project(fields(computed( fieldName: "count", new Document("$size", "$sold")), include( ...fieldNames: "username")));
        Bson sort = sort(descending( ...fieldNames: "count"));
        Bson limit = limit(k);
        aggr = userColl.aggregate(
                Arrays.asList(
                        match, project, sort, limit
                )
        );
    }
    for (Document d : aggr) {
        System.out.println(d);
        orders.add(d);
    }

    return orders;
}
```

The following method returns the k insertions of a specific category with the highest interested value.

```java
public ArrayList<Document> findTopKInterestingInsertion(int k, String category) {

    this.openConnection();
    ArrayList<Document> array = new ArrayList<>();
    MongoCollection<Document> myColl = db.getCollection( s: "insertion");

    Bson limit = limit(k);
    AggregateIterable<Document> aggr  = myColl.aggregate(
            Arrays.asList(
                    Aggregates.match(Filters.eq( fieldName: "category", category)),
                    Aggregates.sort(descending( ...fieldNames: "interested")),
                    limit
            )
    );

    for (Document document : aggr)
        array.add(document);

    this.closeConnection();
    return array;
}
```

The method below, given a specific category, retrieves the most viewed insertions.

```java
public ArrayList<Document> findTopKViewedInsertion(int k, String category) {

    ArrayList<Document> array = new ArrayList<>();

    Bson limit = limit(k);
    AggregateIterable<Document> aggr  = insertionColl.aggregate(
            Arrays.asList(
                    Aggregates.match(Filters.eq( fieldName: "category", category)),
                    Aggregates.sort(descending( ...fieldNames: "views")),
                    limit
            )
    );

    for (Document document : aggr)
        array.add(document);

    return array;
}
```

MongoDB Queries Analysis

READ OPERATIONS		
Operation	**Expected Frequency**	**Cost**
Find user by username	High	Low (1 read)
Find user details	High	Low (1 read)
Find insertion of followed users by id	Average	Average (multiple reads)
Find viral insertions	Average	Very high (aggregation)
Find users by filter	Average	Average (multiple reads)
Find insertions by filter	Average	Average (multiple reads)
Find insertion of a specific user	Average	Average (multiple reads)
Find insertion details by id	High	Low (1 read)
Find most active users (user with most sold items)	Average	Very high (aggregation)
Find the best reviewed users	Average	Very high (aggregation)
Find the most interested insertion of a specific category	Average	Very high (aggregation)
Find the most viewed insertion of a specific category	Average	Very high (aggregation)
Retrieve all the orders, sold or purchased, of a specific user	Average	Average (multiple reads)
Retrieve all the review of a user	High	Average (multiple reads)

WRITE OPERATIONS		
Operation	**Expected Frequency**	**Cost**
Buy an insertion	High	Average (add and delete a document, update two fields)
Suspend user	Low	Low (update 1 field)
Add insertion	High	Low (add document)
Add review	Average	Average (add document)
Update seller rating	Average	Low (update 1 field)
Update insertion view count	High	Low (update 1 field)
Add funds to wallet	Average	Average (update 2 fields)
Remove insertion	Low	Low (delete a document)

Neo4j: queries and analytics

Queries

The function in the following figure returns k suggested users, basing on the users followed by the followed users of the same country.

```
//suggests the users followed by the follower of the logged user and of its same country
public ArrayList<String> getSuggestedUsers(String username, String country, int k) {
    this.open();
    ArrayList<String> suggestions = new ArrayList<>();
    try (Session session = driver.session()) {
        session.readTransaction((TransactionWork<List<String>>) tx -> {
            Result result = tx.run( s: "MATCH (u:User)-[:FOLLOWS]->(m)-[:FOLLOWS]->(others) " +
                            "WHERE u.username = $username AND others.country = $country AND NOT (u)-[:FOLLOWS]->(others) " +
                            "RETURN others.username as SuggUsers " +
                            "LIMIT $k",
                    parameters( ...keysAndValues: "username", username,
                            "country", country,
                            "k", k));
            while (result.hasNext()) {
                Record r = result.next();
                suggestions.add(r.get("SuggUsers").asString());
            }
            return suggestions;
        });
        this.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return suggestions;
}
```

The function in the figure below returns k insertions posted by the followers of a specific user.

```
//get the insertion of the followed users of the specified users
public ArrayList<String> getFollowedInsertions(String username, int k) {

    this.open();
    ArrayList<String> followed = new ArrayList<>();
    try (Session session = driver.session()) {
        session.readTransaction((TransactionWork<List<String>>) tx -> {
            Result result = tx.run( s: "MATCH (u:User)-[:FOLLOWS]->(m)-[:POSTED]->(i:Insertion) " +
                            "WHERE u.username = $username " +
                            "RETURN i.uniq_id as SuggIns " +
                            "LIMIT $k",
                    parameters( ...keysAndValues: "username", username,
                            "k", k));

            while (result.hasNext()) {
                Record r = result.next();
                followed.add(r.get("SuggIns").asString());
            }
            return followed;
        });
        this.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return followed;

}
```

The function below retrieves all the followers of a specific user.

```java
//retrieve all the followers of the specified user
public ArrayList<String> retrieveFollowersByUser(String user) {

    this.open();
    ArrayList<String> followers = new ArrayList<>();
    try (Session session = driver.session()) {
        session.readTransaction((TransactionWork<List<String>>) tx -> {
            Result result = tx.run( s: "MATCH (u:User) <- [r:FOLLOWS] - (u1:User) " +
                    "WHERE u.username = $username " +
                    "RETURN u1.username as name ",
                parameters( ...keysAndValues: "username", user));

            while (result.hasNext()) {
                Record r = result.next();
                followers.add(r.get("name").asString());
            }
            return followers;
        });
        this.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return followers;
}
```

This function retrieves the users followed by a specific user.

```java
//retrieves all the following users of the specified user (shown in profile page)
public ArrayList<String> retrieveFollowingByUser(String user) {

    this.open();
    ArrayList<String> following = new ArrayList<>();

    try (Session session = driver.session()) {
        session.readTransaction((TransactionWork<List<String>>) tx -> {
            Result result = tx.run( s: "MATCH (u:User) - [r:FOLLOWS] -> (u1:User) " +
                    "WHERE u.username = $username " +
                    "RETURN u1.username as name ",
                parameters( ...keysAndValues: "username", user));

            while (result.hasNext()) {
                Record r = result.next();
                following.add(r.get("name").asString());
            }
            return following;
        });
        this.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return following;
}
```

The function below, indeed, retrieves all the insertions of the user.

```
//returns all the interested insertion of the specified user (logged user or selected user)
public ArrayList<String> retrieveInterestedInsertionByUser(String user) {

    this.open();
    ArrayList<String> followed_ins = new ArrayList<>();

    try (Session session = driver.session()) {
        session.readTransaction((TransactionWork<List<String>>) tx -> {
            Result result = tx.run( s: "MATCH (u:User) - [r:INTERESTED] -> (i:Insertion)" +
                            "WHERE u.username = $username " +
                            "RETURN i.uniq_id as uniq_id",
                    parameters( ...keysAndValues: "username", user));

            while (result.hasNext()) {
                Record r = result.next();
                followed_ins.add(r.get("uniq_id").asString());
            }
            return followed_ins;
        });
        this.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return followed_ins;
}
```

Statistics

The following function shows to the admin the number of likes given a specific category.

```
public ArrayList<String> findNumberInterestingForCategory() {

    this.open();
    ArrayList<String> interesting = new ArrayList<>();

    try (Session session = driver.session()) {

        session.readTransaction((TransactionWork<List<String>>) tx -> {
            Result result = tx.run( s: "match(u:User)-[r:INTERESTED]->(i:Insertion) " +
                    "RETURN DISTINCT i.category as category, count(r) AS counter" +
                    " ORDER BY counter DESC");

            while (result.hasNext()) {
                Record r = result.next();
                String category = r.get("category").asString();
                int count = r.get("counter").asInt();
                String ins = category + ":" + count;
                interesting.add(ins);
            }
            return interesting;
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
    return interesting;

}
```

Password encryption

All the passwords are not stored as they are submitted by users in the registration phase, but they are encrypted using MD5 hash function. This operation ensure the privacy of the credentials.

```
public class CryptWithMD5 {

    private static MessageDigest md;

    public static String cryptWithMD5(String pass){
        try {
            md = MessageDigest.getInstance("MD5");
            byte[] passBytes = pass.getBytes();
            md.reset();
            byte[] digested = md.digest(passBytes);
            StringBuffer sb = new StringBuffer();
            for (byte b : digested) {
                sb.append(Integer.toHexString( i: 0xff & b));
            }
            return sb.toString();
        } catch (NoSuchAlgorithmException ex) {
            Logger.getLogger(CryptWithMD5.class.getName()).log(Level.SEVERE,  msg: null, ex);
        }
        return null;

    }
}
```

Crud Operations (MongoDB)

In this section there are some examples of the most common crud operations performed by the application.

Create

The following method is called when a user is registered into the application.

```
public boolean registerUser(User u) {

    if (userAlreadyPresent(u.getUsername())) {
        Utility.infoBox( infoMessage: "Please, choose another username and try again.",
                         titleBar: "Error",  headerMessage: "Username already used!");
        return false;
    }

    Document user = new Document("address", u.getAddress())
            .append("city", u.getCity())
            .append("country", u.getCountry())
            .append("email", u.getEmail())
            .append("img_profile", u.getImage())
            .append("name", u.getName())
            .append("password", u.getPassword())
            .append("suspended", u.getSuspended())
            .append("username", u.getUsername());

    Document balanceUser = new Document("username", u.getUsername())
            .append("credit", 0);

    userColl.insertOne(user);
    balanceColl.insertOne(balanceUser);

    return true;
}
```

Read

This function gets the seller by input and returns all the insertions posted by the seller with that username.

```
public ArrayList<Document> findInsertionBySeller(String seller) {

    ArrayList<Document> insertions = new ArrayList<>();

    cursor = insertionColl.find(eq( fieldName: "seller", seller)).iterator();
    while (cursor.hasNext())
        insertions.add(cursor.next());

    return insertions;
}
```

Update

The following function updates the rating of a specific user, given by the average of all the reviews written to that user.

```
public void updateSellerRating(String seller) {

    Document d = userColl.find(eq( fieldName: "username", seller)).first();
    List<Document> list = d.getList( key: "reviews", Document.class);

    Double avg;
    int sum = 0;

    for (Document document : list)
        sum += document.getInteger( key: "rating");

    avg = (double) sum / (double) list.size();

    // {$set: {"rating": avg}}
    Bson filter = eq( fieldName: "username", d.getString( key: "username"));
    Bson update = set("rating", avg);;

    userColl.findOneAndUpdate(filter, update);
}
```

Delete

This function allows the user to delete an insertion.

```
public boolean deleteInsertionMongo(String id) {

    Bson query = eq( fieldName: "_id", new ObjectId(id));

    try {
        DeleteResult result = insertionColl.deleteOne(query);
        return (result.getDeletedCount() == 1);
    } catch (MongoException me) {
        System.err.println("Unable to delete due to an error: " + me);
        return false;
    }
}
```

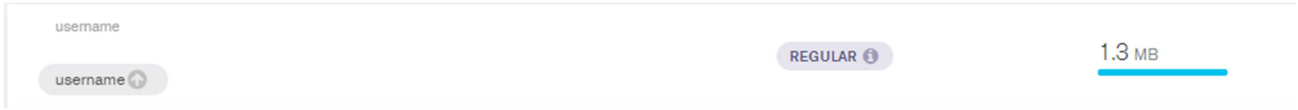
MongoDB indexes

These indexes below were used in order to speed up the application. Some tests were used to measure the speed improvement obtained by using indexes. The tests were carried out using the explain() function offered by MongoDB.

INDEX NAME	INDEX TYPE	COLLECTION	ATTRIBUTES
username	*Single*	*User*	*username*
country	*Single*	*User*	*country*
category	*Single*	*Insertion*	*category*

User Collection Test

1. Username




```
db.user.find({username: "Aali"})
```

INDEX	DOCUMENT RETURNED	INDEX KEY EXAMINED	DOCUMENT EXAMINED	EXECUTION TIME (ms)
False	*1*	*0*	*99506*	*63*
True	*1*	*1*	*1*	*11*

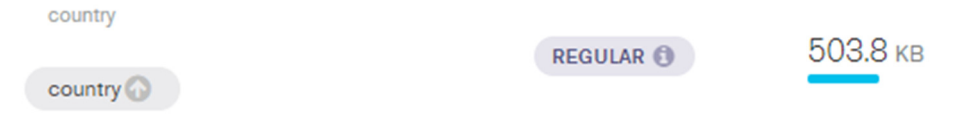
2. Uniqueness of username

This index guarantees that in the user collection there are not two documents with the same username.

Index	Document Returned	Index Key Examined	Document Examined	Execution Time (ms)
False	*1*	*0*	*99506*	*63*
True	*1*	*1*	*1*	*11*

3. Country

```
db.user.aggregate({$match :{country: "Italy"}}, {$sort: {interested: 1}})
```

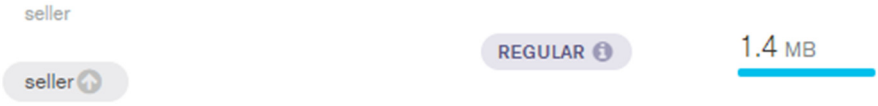


INDEX	DOCUMENT RETURNED	INDEX KEY EXAMINED	DOCUMENT EXAMINED	EXECUTION TIME (ms)
False	*10004*	*0*	*99505*	*66*
True	*10004*	*1*	*10004*	*26*

Insertion Collection Test

1. Seller

```
db.insertion.find({seller: "Aali"})
```



INDEX	DOCUMENT RETURNED	INDEX KEY EXAMINED	DOCUMENT EXAMINED	EXECUTION TIME (ms)
False	*2*	*0*	*120031*	*70*
True	*2*	*1*	*2*	*0*

Neo4j Indexes

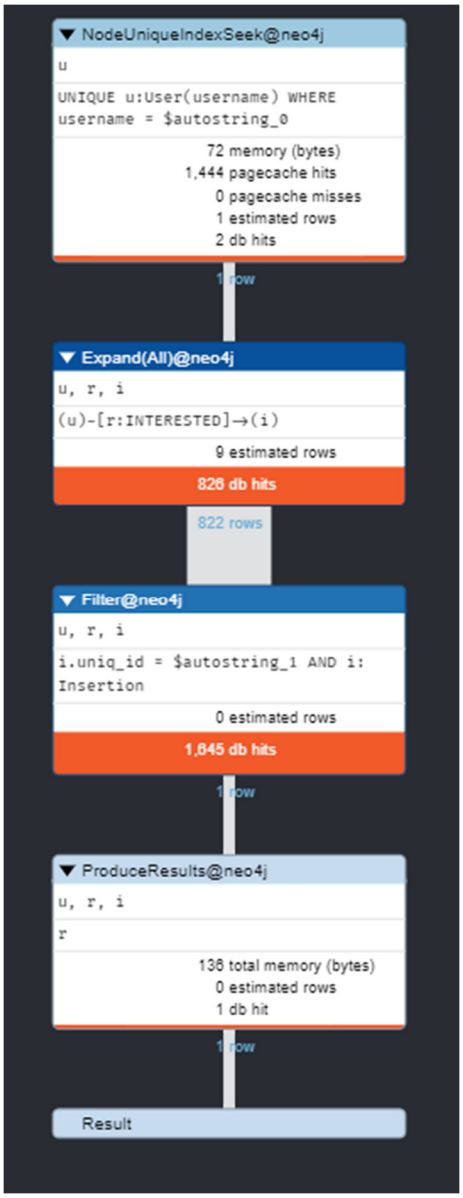
INDEX NAME	INDEX TYPE	INDEX LABEL	INDEX PROPERTIES
username_const	*BTREE[UNIQUE]*	*:User*	*username*
uniq_id	*BTREE*	*:Insertion*	*uniq_id*

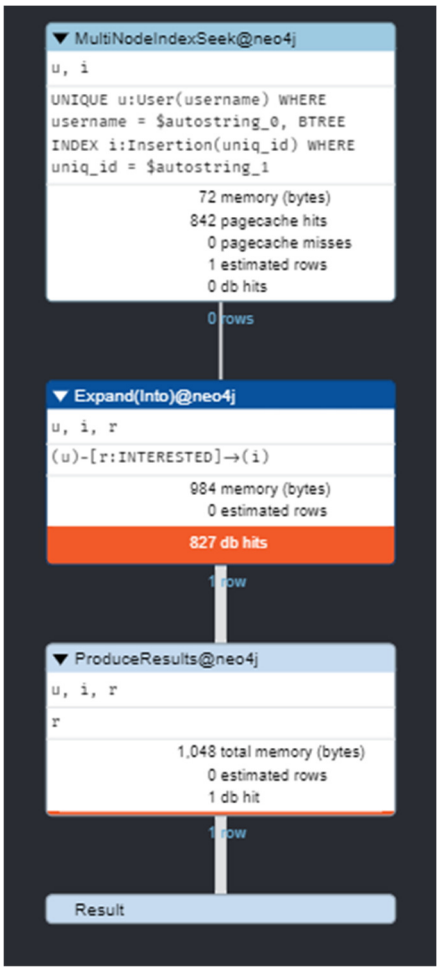
- uniq_id

There are some functions on Neo4j that use the *uniq_id* field of the *Insertion* that show the advantages of adding an index on this field. One of them is **showIfInterested**() shown below.

```
PROFILE
MATCH (u:User { username: "Aamir"})-[r:INTERESTED]->(i :Insertion {
uniq_id:"61fc54c2ce8ed77a2c6b5b01"})
RETURN r;
```

Executing the query without the index, as shown in the left figure, it was necessary to scan 822 rows and wait 29ms.





The right figure shows the execution of the query with the index: the time decreases to 2ms and the number of rows to scan becomes 1. So, this index helps the query not only in terms of time but also in terms of number of rows to scan.

Brief consideration about the CAP theorem

All the choices that were made during the design and implementation phases were aimed at fulfilling the functional and non-functional requirements.

As known from the CAP theorem, it is not possible for a distributed database to provide availability, consistency and partition protection at the same time.

In order to ensure the non-functional requirements, it was necessary to priorize avaiability and partition protection over consistency.

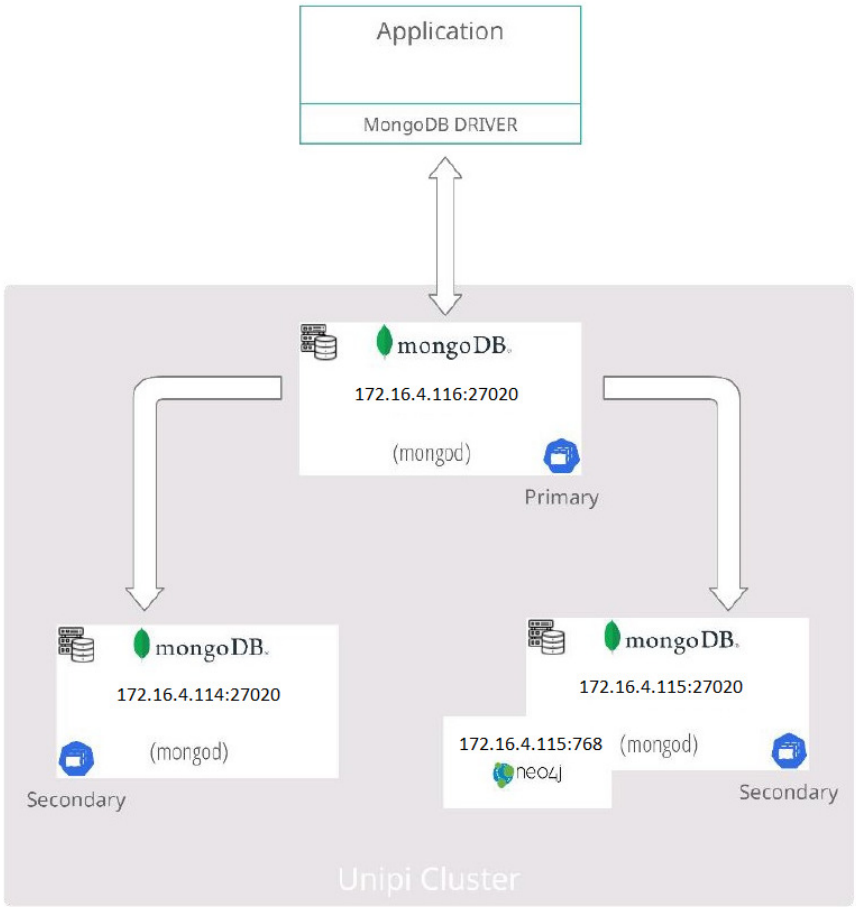
Infact, the main purpose of this application, as an e-commerce, is to guarantee to the users the possibility to use the application in any moment they want.

In any case, when necessary, especially reguarding the balance of the users, the consistency was mantained. Generally, in case of a failure, the user will see inconsistent data, but the application will still work.

MongoDB replica set

A replica set has been set up in order to avoid the "single point of failure": the cluster is composed by three virtual machines provided by UniPi that host three different instances of mongod. One of them is the primary that acts like a server receiving all the requests and the other two are the secondary ones keeping copies of the data stored on the primary.

VM	IP ADDRESS	PORT	OS
Replica-0	*172.16.4.114*	*27020*	*Ubuntu*
Replica-1	*172.16.4.115*	*27020*	*Ubuntu*
Repica-2	*172.16.4.116*	*27020*	*Ubuntu*



Replica configuration

The configuration of the cluster is shown below:

```
rsconf = {
          _id: "lsmdb",
          members:
          [
               {_id:3, host:"172.16.4.114:27020", priority:1},
               {_id:4, host:"172.16.4.115:27020", priority:2},
               {_id:2, host:"172.16.4.116:27020", priority:3}
          ]
};
```

The application is read oriented because is more frequent that a logged user browse the various insertions and sellers than doing an order or publish a new insertion, so reads predominate over writes.

Regarding Write Concern and Read Preferences has been set the configuration below:

```
"mongodb://172.16.4.114:27020,172.16.4.115:27020,172.16.4.116:27020/" +
        "?retryWrites=true&w=1&readPreferences=nearest&wtimeout=10000"
```

- Write Concern
  - w = 1 : wait for acknowledgement from a single member.
- Read Preferences
  - readPreference=nearest : read from the member of the replica set with the least network latency to have the fastest response

At collection level there is a difference between *Balance* collection and the others: it is the only one that has w = 3, because it manages the money that every user have on his/her account so it is important to always have the strict consistency. All the others have w = 1;

The idea is to manage the balance part with a Key-Value database in the future in order to improve the performance of the application and reduce the accesses on different collections of MongoDB when a purchase is done.

Replica crash

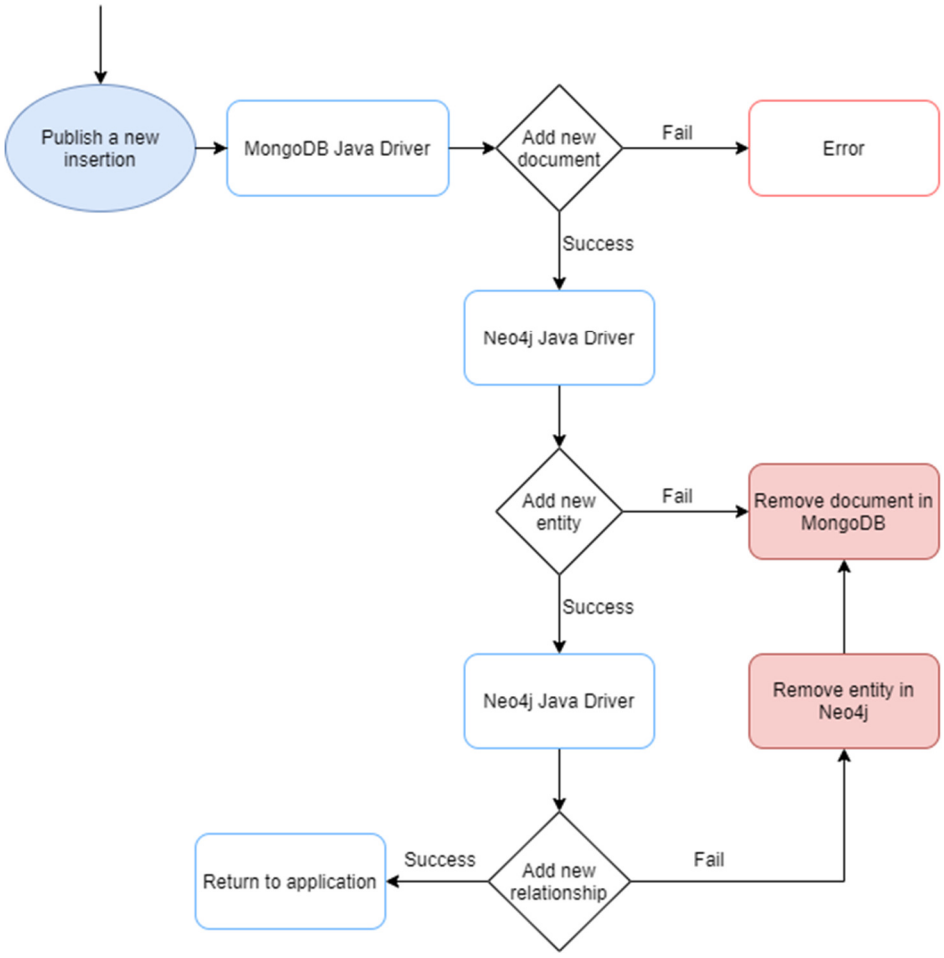
If the primary fails there will be an election to decide the new primary one among the two secondaries. Given the priority assigned on each replica, the behaviour of the election algorithm can be predicted to know which of the two secondaries will be promoted. In the specific case, when the primary is not available, the VM 172.16.4.115 will be marked as the new primary.

MongoDB and Neo4j interactions

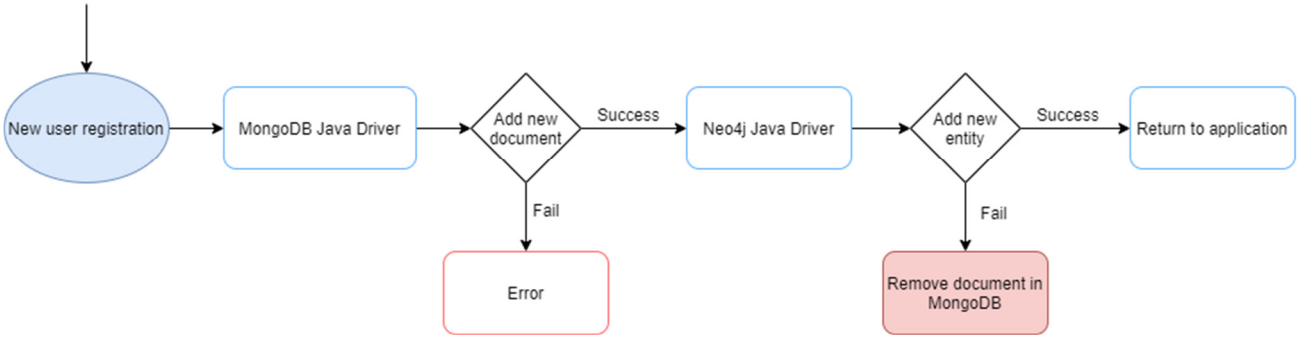
There are some redundant attribute fields into the two databases. For this reason, it is necessary to manage possible inconsistencies between the documents stored into MongoDB and the entities into Neo4j. To guarantee the consistency of the data, has been provided to update, when necessary, the data both in MongoDB and in Neo4j.

The operations that regard information stored in both databases are:

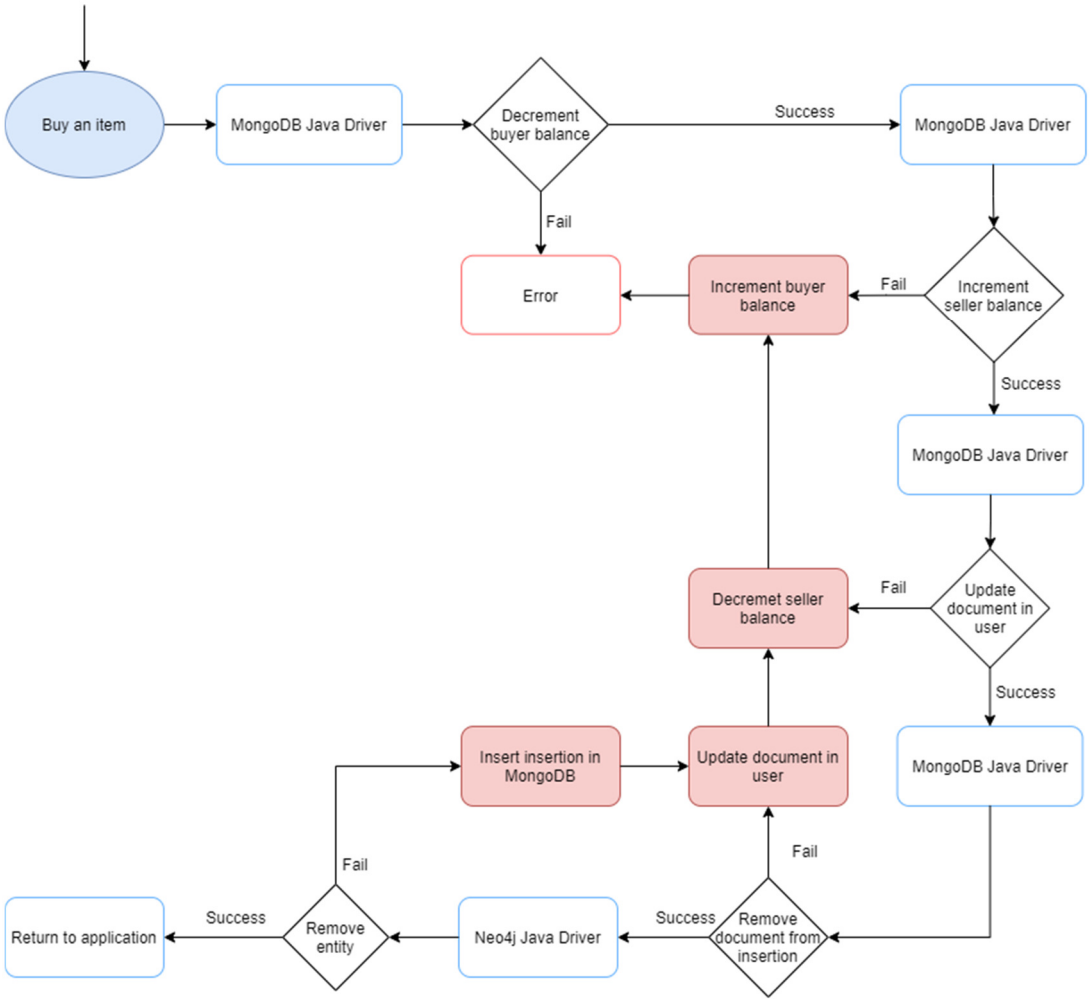
- Add new insertion



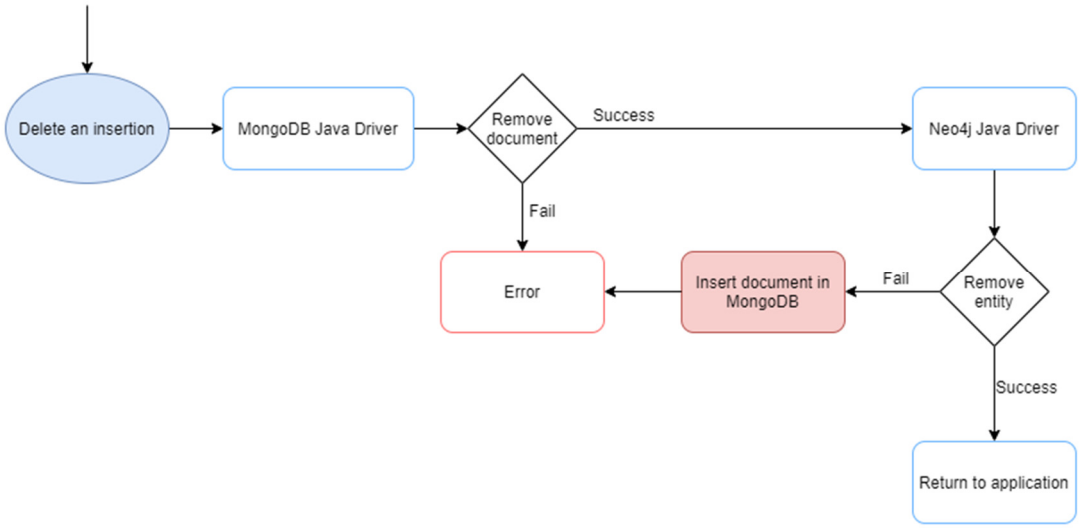
- Add new user



- Buy insertion



- Delete insertion



Sharding discussion

Database systems with large data sets or high throughput applications can challenge the capacity of a single server. For example, high query rates can exhaust the CPU capacity of the server. Working set sizes larger than the system's RAM stress the I/O capacity of disk drives.

MongoDB supports horizontal scaling through sharding.

Horizontal Scaling involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required. While the overall speed or capacity of a single machine may not be high, each machine handles a subset of the overall workload, potentially providing better efficiency than a single high-speed high-capacity server. Expanding the capacity of the deployment only requires adding additional servers as needed, which can be a lower overall cost than high-end hardware for a single machine. The trade-off is between increasing complexity in infrastructure and maintenance for the deployment.

Regarding the application proposed, if the read or write requests increase too much the number of mongos can be incremented to distribute the load. For what concern the sharding, considering the fact that the e-commerce is mainly based on local buying and selling, the main proposal is a local solution. With the sharding mechanism data can be retrieved in the using a sharding key.

The sharding keys for a hashed strategy could be:

- **username** for user collection
- **uniq_id** for insertion collection
- **code_string** for code collection

Another solution is list strategy using the **country** field of User and Insertion for a Geographical Sharding in which each partition can manage the specific region assigned.

A third option could be to shard insertions using the list strategy for dividing them by category in order to allocate documents across different servers. This can be combined with a hashed sharding for users.

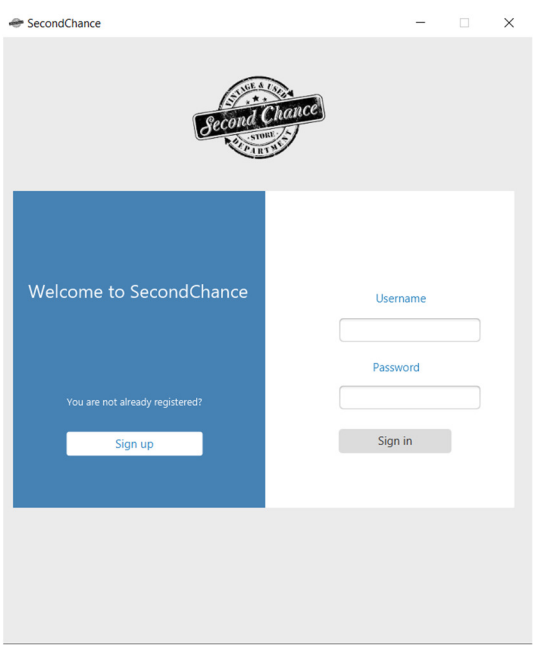
Improvements

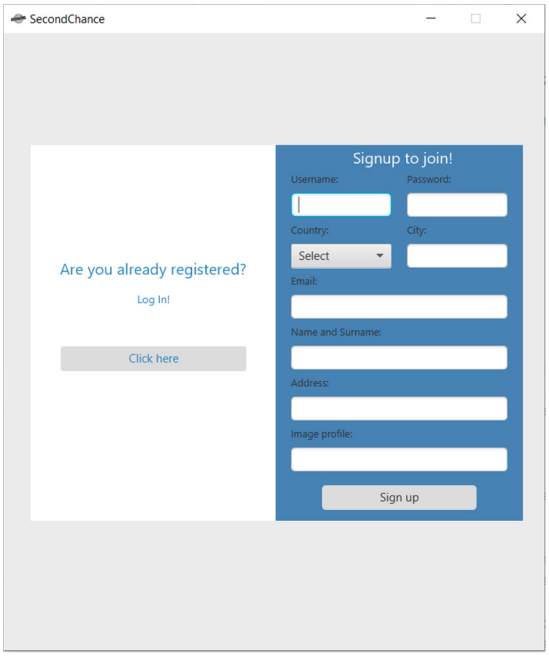
The application proposed is mainly based on displaying images so it could be useful to store the images in a cache that can be realized with a Key-Value database. Given its simplicity, also the balance entity could be implemented as a Key-Value database with the advantage of avoiding the access to different collections when an order is processed.

Key-value stores scale out by implementing partitioning (storing data on more than one node), replication and auto recovery. They can scale up by maintaining the database in RAM and minimize the effects of ACID guarantees (a guarantee that committed transactions persist somewhere) by avoiding locks, latches and low-overhead server calls.

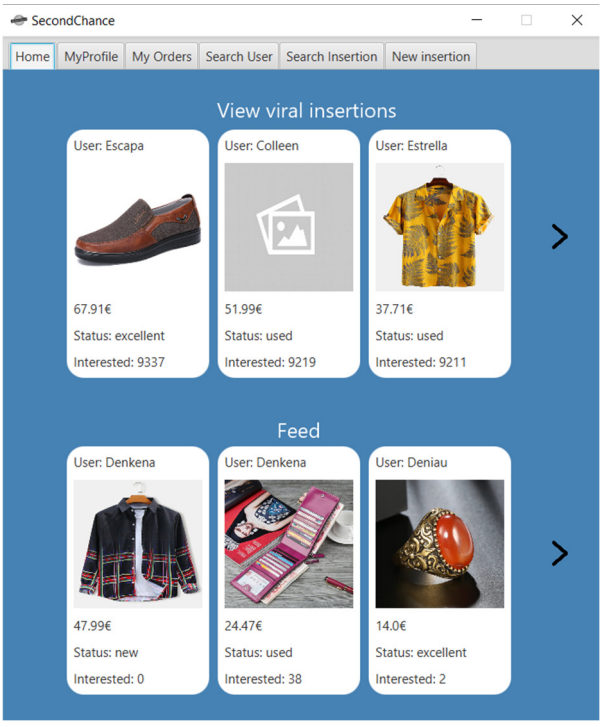
User Manual

Starting the application, the user will see the initial page where he can sign in if he/she is already registered, or sign up if he/she is not already.





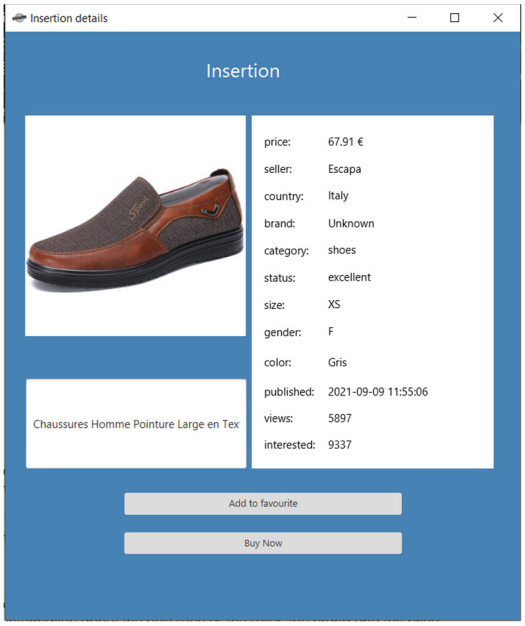
Once logged in, the user will see the *Home* screen with two sections: Viral Insertion and Feed.



In the *Viral Insertion section* there are those ads that have a very high number of likes and that are still present in the user's country of reference.

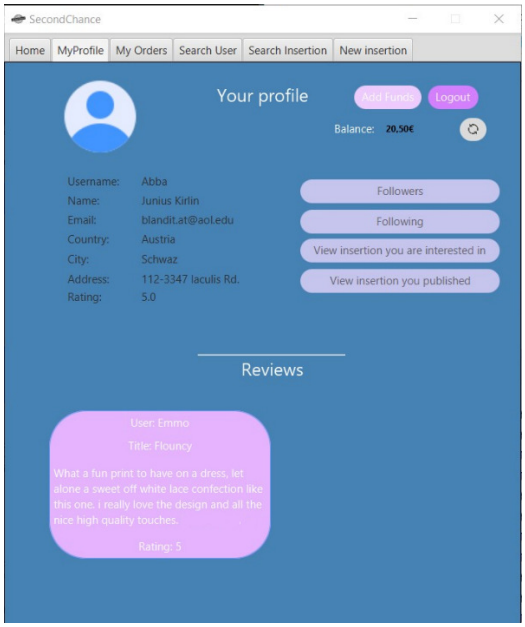
In the *Feed section*, only posts from users that a person follows will appear. If a user does not follow any other user in the feed section, the most interesting insertions will appear.

To open an insertion it is possible to click on it: after this action a window will be opened containing all the information about the post such as the seller, the brand and the price.

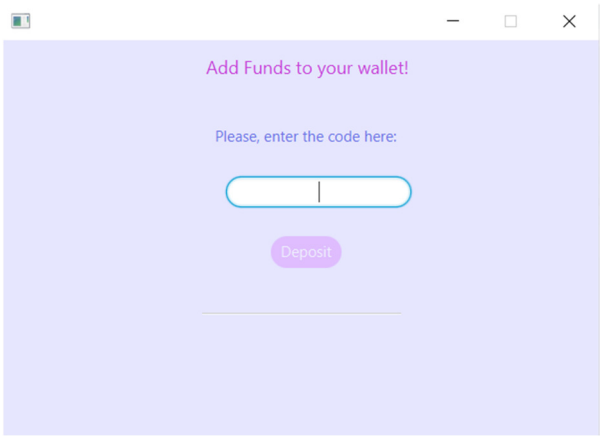


A user can decide to buy the object on sale if the balance of his profile allows it, or he can insert the post in his favourites.

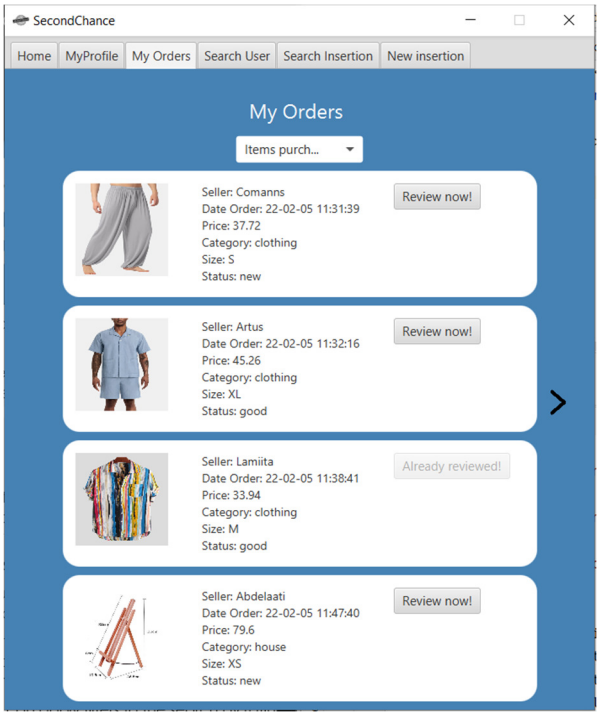
In *MyProfile section* it is possible to see one's own basic information, followers and e following and to see the listings in which a user is interested or the ones he has published. In both cases a window containing the advertisements will be opened. There is also a logout button to end the session.



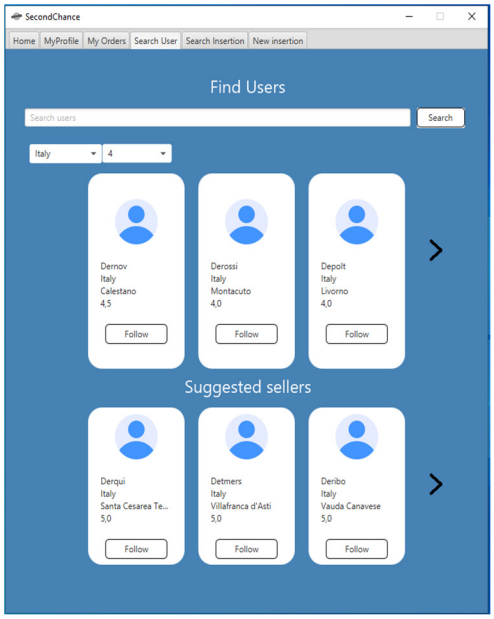
In the upper right corner there is also a button *Add Funds* that allows, through the previous purchase of a code from a third party dealer, to insert the code inside the appropriate box and recharge the account of the amount decided at the time of purchase.



In the *MyOrders section* it is possible to visualize the purchased and sold items. According to the selection will be displayed boxes containing the listings.

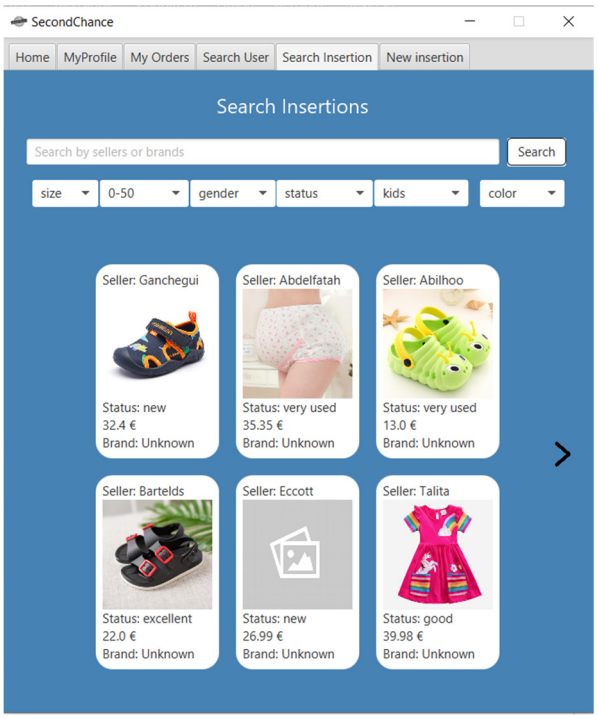


Analyzing instead the *SearchUser section* it's possible, through the search bar, to insert the username of a seller that we want to search and then we can follow him. In this way his ads will be added to the Feed section previously mentioned.

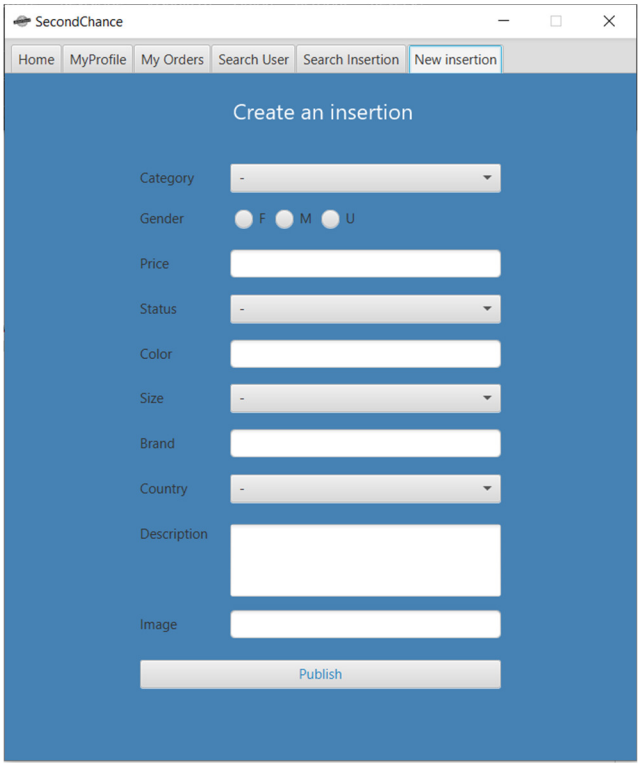


Through the two drop-down menus a user can apply filters to the search by rating or country.

The *SearchInsertion section* is apparently similar: in fact it always presents itself with a search bar in which a user can search for listings by seller or by brand. Also through the drop-down menu you can apply filters to the search.



Finally, in the *NewInsertion section* it is possible to create and publish a new advertisement by filling in all the fields present. It is also possible to insert an image of the goods you want to sell.

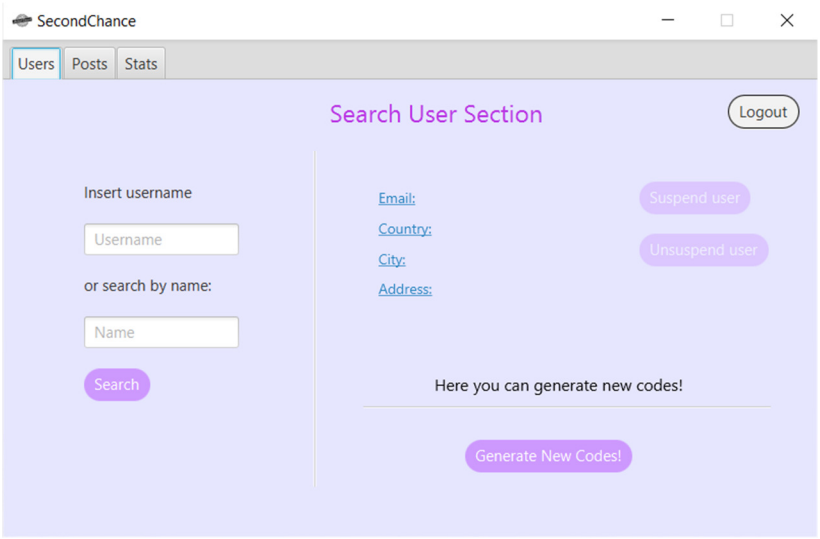


Admin Manual

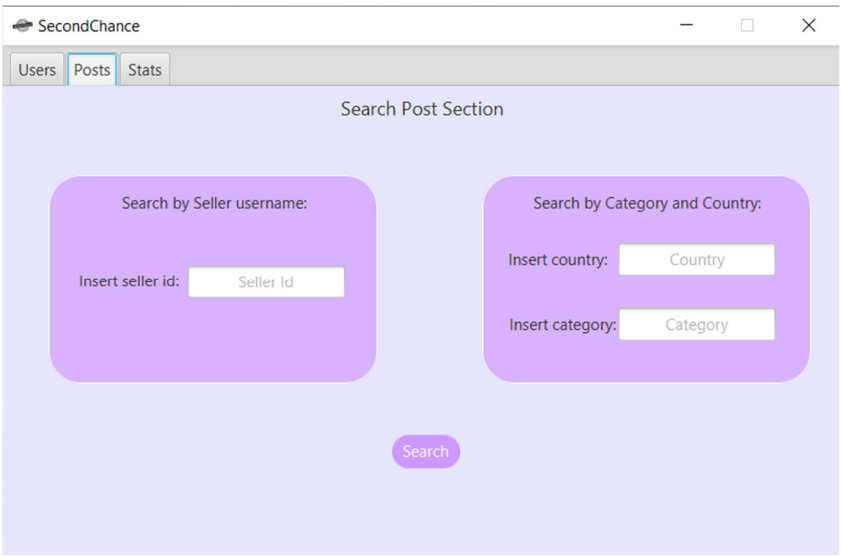
Starting the application the initial screen will be the same of the login of a normal user.

Initially the *SearchUser page* is shown which allows the admin to search for a user either by username or by name and surname. Once the search is done and the user has been found, it is possible to suspend or reactivate his/her account.

It is fundamental to specify that the admin is an entity that can suspend other users and therefore can perform this action based on misbehavior by a user.

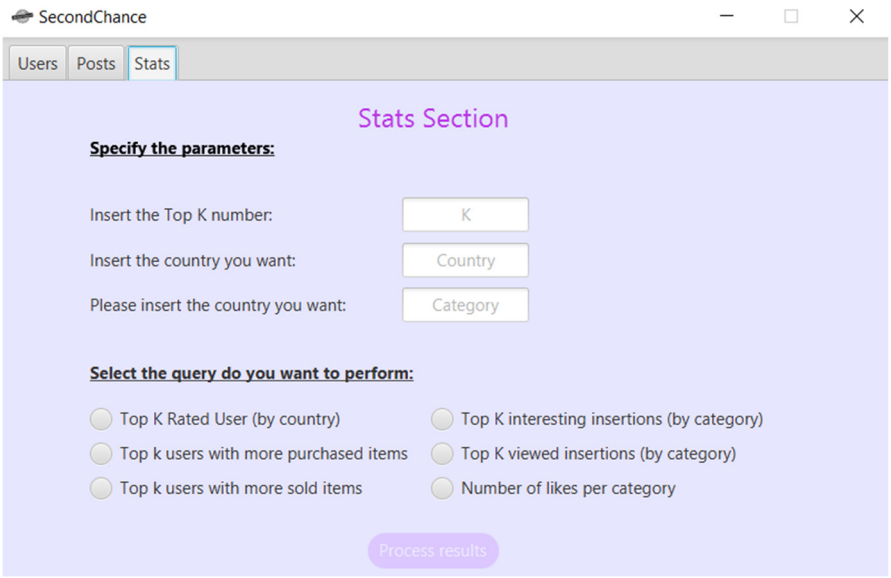


In the *SearchPost section* instead the admin can search through the search fields for seller username an ad and, if appropriate, delete it from the MongoDB database. Obviously will be updated also all the arcs on Neo4J that represent the relationship "**INTERESTED**" by other users and the only existing arc "**POSTED**".



The Stats section is the section dedicated to statistics and deserves a more in-depth look.

We basically have the window vertically divided in two parts: the "***Top K Section***" part and the "***Most Item Stats***" part.



The first part, in detail, provides that the admin initially enters a K number.

After that it is possible to choose one of the 3 functionalities through the menu below:

Top K Rated User (by country): this function provides a ranking of the K users who have the highest rating for the country entered.

Top K Most Interesting Insertions (by category): this function provides a ranking of the K insertions that have the highest interested field based on the category entered.

Top K Most Viewed Insertions (by category): this function provides a ranking of the K insertions that have the highest viewed field according to the category entered.

In the section Most Items Stats it is possible to make two queries: to verify which are the *Most Active Users*, that is those users who have published more ads and to verify which are the *Most Active Sellers*, that is those profiles that have sold more items through the public ads.