UNIVERSITÀ DI PISA

Large-Scale and Multi-Structured Databases
PhoneWorld
Application Report

Application developed by
Daniele Laporta, Francesco Martoccia,
Salvatore Lombardi

Summary

Abstract

PhoneWorld is an application that everyone should use before buying a new phone or just to feed the desire to know which smartphone is considered the best by the community.

The application's main functionalities are collecting, organizing and presenting to users information related to all the phone models from 1999 until nowadays, with all their specifications and the reviews of the users.

Users can browse phones, read their specifications, add them to their watchlist and write a review for them, but also interact with other users, following them and viewing their watchlist.

For this project, the application is developed in Java using Intellij with Spring Data MongoDB to facilitate the mapping of the java objects with documents in the database and JavaFX for the graphic interface.

The data are stored in two different databases as we have two different typologies of databases. We decided to use MongoDB to store all the information about the phones, the users and the reviews. The users and the phones are both represented by a node in Neo4j, in order to better manage the social part, where users can add phones to their watchlist and follow other users.

Repository link: FrankMartoccia/phoneworld (github.com)

Feasibility Study

The first thing to do is a feasibility study of the idea of the project in order to write down the pros and cons, with all the limitations and the possible future developments.

Dataset analysis

Firstly, we focused our attention on the data we need to develop the application and we found these 5 different datasets from different sources with total storage involved around 110 MB:

- 20191226-items.csv
- 20191226-reviews.csv
- brands.json
- devices.json
- users.json

We found the first two datasets on Kaggle.com. They are composed of a lot of phones, each one associated with around 70 thousand reviews, scraped from amazon.com.

The third and fourth dataset, instead, are found on github and are scraped from gsmarena.com. They consist of more than 10 thousand device models with all the specifications and are associated with their brands.

The last one is a random users dataset generated with the randomuser.me API, which counts exactly 50 thousand unique users.

Looking generally at these datasets we found there was the need for some data pre-processing. So we analyse each dataset carefully with the proposed final aim of having 3 datasets representing Reviews, Phones and Users, the three main prospected entities of our application. We decided to use python as programming language for this step to be able to easily manage data modifications on a shared Google Colaboratory file.

We started our journey looking at the items and reviews data which brings us to obtain the Reviews.csv file.

We checked and managed the presence of duplicated rows and null values and resolved some cases of incorrect data that could lead to problems during the analysis and the implementation.

We also drop some useless attributes and rename some others in order to be better understandable.

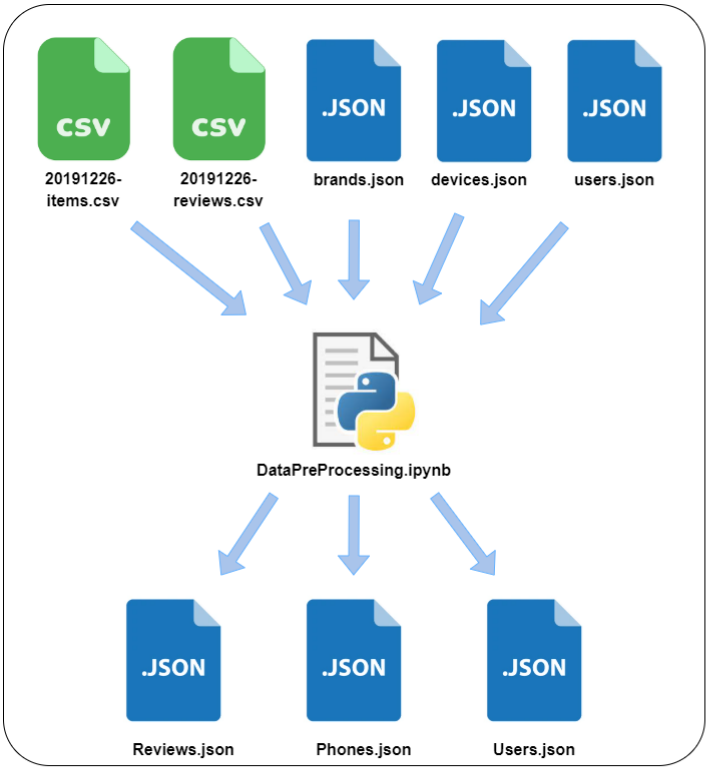
Then we step into brands and devices datasets where we did almost the same data cleaning steps in order to get the Phones.csv file.

The need of creating a dataset just composed of random users came from a problem encountered visualizing the initial data about the reviews, where each review is associated with a person name, but it became insufficient for our purposes. So we decided to build the Users.csv which is made up of so much information for each user, giving us the opportunity to exploit a lot of new analysis.

In the end we made some attribute adjustments in order to facilitate their import and use on the database management systems.

The process is available on the file *DataPreProcessing.ipynb* available in the git repository.

It's important to state that for the whole project we experience the Agile method, so there were some changes done in progress in order to keep the consistency of data and the code.



Analysis results

Looking carefully at how the data were at first, allows us to better understand what we wanted to do and how to reach our objective.

At the end of the analysis we can confirm that we reached the aim of obtaining 3 connected datasets for reviews, phones and users starting from 5 apparently distinct datasets.

This step was also fundamental for the preparation of the data model part that we will take on later in this document, above all to take care of the attributes used to link the different datasets.

During this preliminary phase we immerse ourself in data rows, acquiring general knowledge about them and just starting thinking about the next steps of this project.

So now the documentation goes on with the two main sections, the Design and the Implementation section.

As future developments, we want to include in the application a new function for the administrators which permits them to scrape new smartphone models and new reviews from defined sources on defined time intervals.

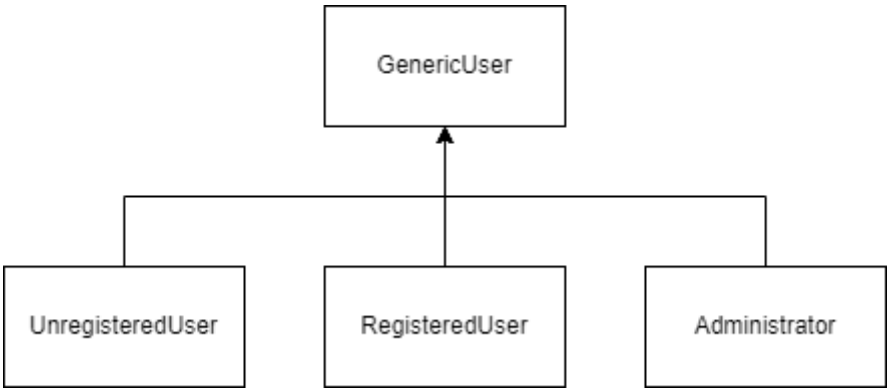
Design

The Design of the application is the direct consecutive step that follows the approval of the feasibility study.

Main actors

As already mentioned, the data on which the application is based on, are exactly related to the three entities wanted to be represented, that are: reviews, phones and users.

In particular, the application is meant to be used by the users, divided into 3 main actors: unregistered users, registered users and administrators.



Functional requirements

This section describes the functional requirements that the application must provide, divided with respect of the three main actors.

The system must allow to carry out functions such as, for:

- **Unregistered users**:
  - to log in or register within the application with their username and password. If they register, they become Registered Users,
  - to browse phones and read the associated reviews. The decision to allow unregistered users to visualize some phones is taken in order to favour the diffusion of the application, encouraging them to register.
- **Registered users** (after having done the login):
  - to browse phones,
  - to view phone reviews,

- to add a review for the phone,
- to add phones to their watchlist,
- to remove phones from their watchlist,
- to view suggested phones based on their favourite brand and, distinctly, based on friends' watchlist,
- to view suggested users based on their favourite brand and, distinctly, based on friends' watch list,
- to view suggestions on phones and users directly after log in,
- to show top rated phones,
- to show top rated brand based on review ratings,
- to show most appreciated brands (most present in user's watchlist),
- to browse users,
- to view users reviews,
- to follow and unfollow other users,
- to view the most followed users,
- to view their profile's informations,
- to update their profile's informations,
- to view their watchlists,
- to view their reviews,
- to delete or update their reviews.

- **Administrators** are users who are allowed to do some operations in common with registered users and others available only to them. Specifically, they can:
  - browse a phone,
  - add a phone,
  - remove a phone,
  - update a phone,
  - view phone reviews,
  - delete reviews,
  - find reviews by word,
  - add a new admin,
  - browse a user,
  - delete a user,
  - view analytics, which are:
    - ⇛ top rated brands,
    - ⇛ top phones by average rating of their reviews,
    - ⇛ most appreciated brands (most present in user's watchlist),
    - ⇛ most followed users,
    - ⇛ most active users,

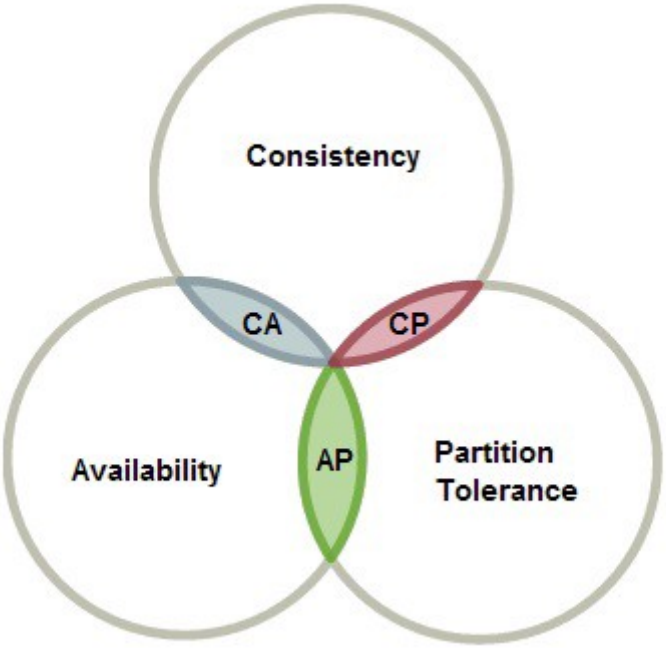
⇛ younger countries by average age of their users,
⇛ top countries by number of their users.

Non-functional requirements

The non-functional requirements of the application are outlined below:

- usability, the application must be simple and intuitive, providing a fluid experience for the user,
- the code must be readable and easy to maintain,
- high availability, accepting data displayed temporarily in an old version,
- low latency in accessing the database,
- tolerance to the loss of data, avoiding a single point of failure.

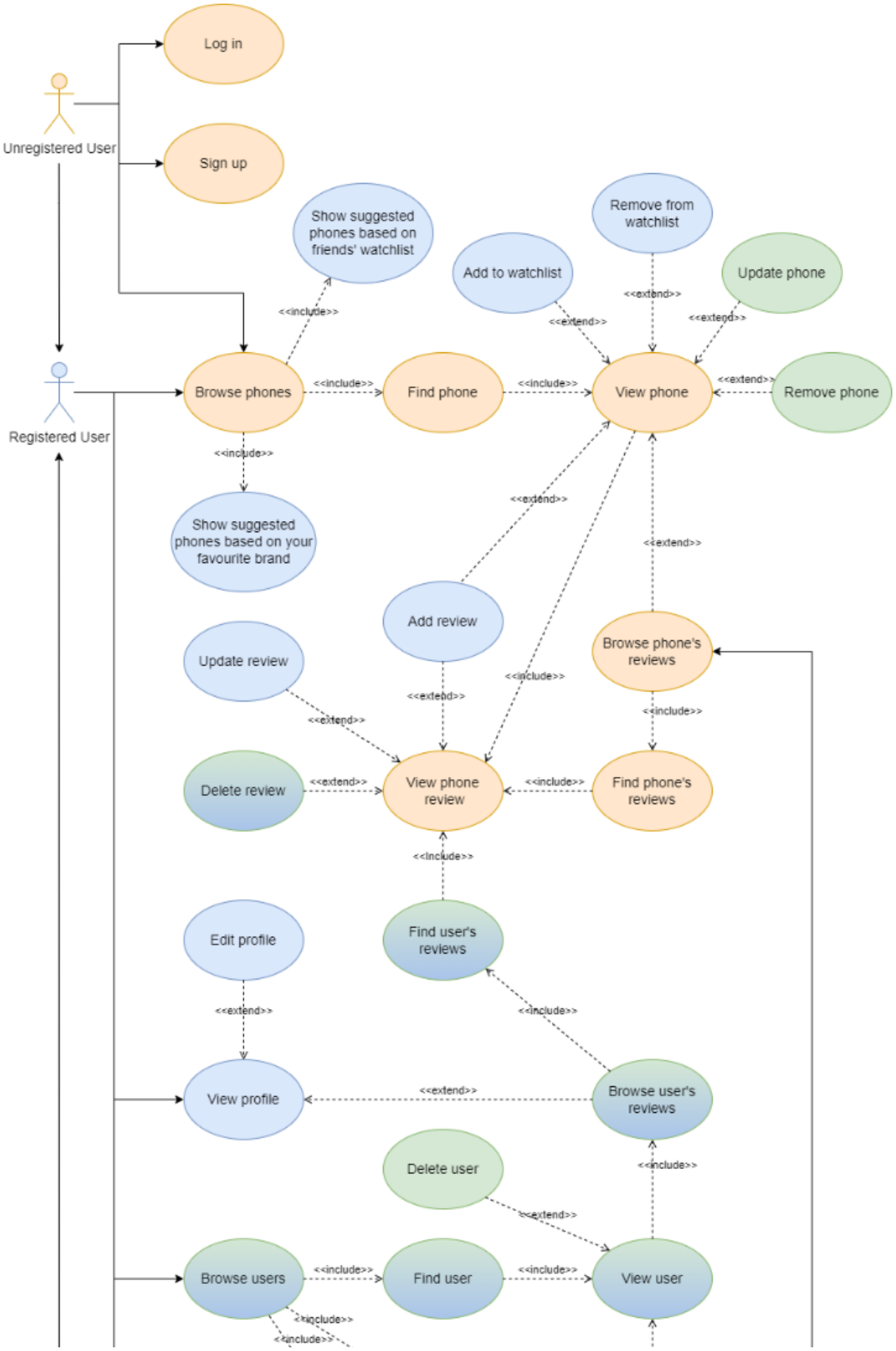
Handling CAP theorem issue

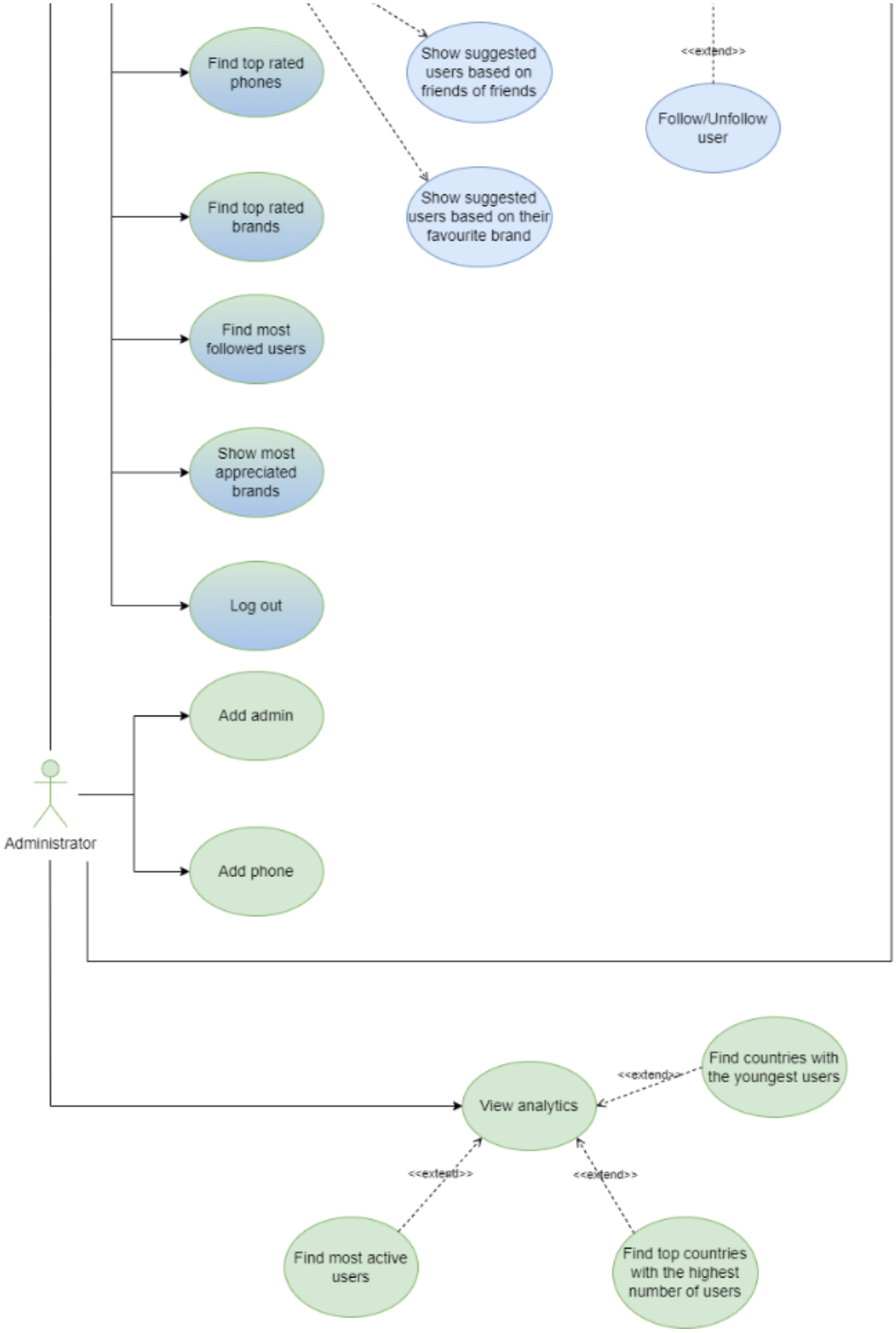


For this application, the expectation is to have a lot of read operations, so it's a priority to guarantee high availability and low latency, with a system still available under partitioning.
Considering the CAP theorem, the application is more oriented to the **AP** side of the triangle, favouring Availability (A) and Partition Tolerance (P) in spite of data Consistency (C).
In order to respect the non functional requirements, we decided to guarantee high availability of data, even if an error occurs on the network layer, accepting that the content returned to the user cannot always be accurate, meaning that data

displayed are shown temporarily in an old version.
This is directly linked to the adoption of the Eventual Consistency paradigm for our distributed system which is better detailed in the Distributed database design section.

Use Cases


Log in
Unregistered User
Sign up
Show suggested phones based on friends' watchlist
Remove from watchlist
Add to watchlist
Update phone
<<include>>
<<extend>>
<<extend>>
<<extend>>
Browse phones
<<include>>
Find phone
<<include>>
View phone
<<extend>>
Remove phone
Registered User
<<include>>
<<extend>>
Show suggested phones based on your favourite brand
<<extend>>
Add review
Browse phone's reviews
Update review
<<include>>
<<extend>>
<<extend>>
<<include>>
Delete review
<<extend>>
View phone review
<<include>>
Find phone's reviews
<<include>>
Edit profile
Find user's reviews
<<extend>>
<<include>>
View profile
<<extend>>
Browse user's reviews
Delete user
<<include>>
<<extend>>
Browse users
<<include>>
Find user
<<include>>
View user
<<include>>
<<include>>



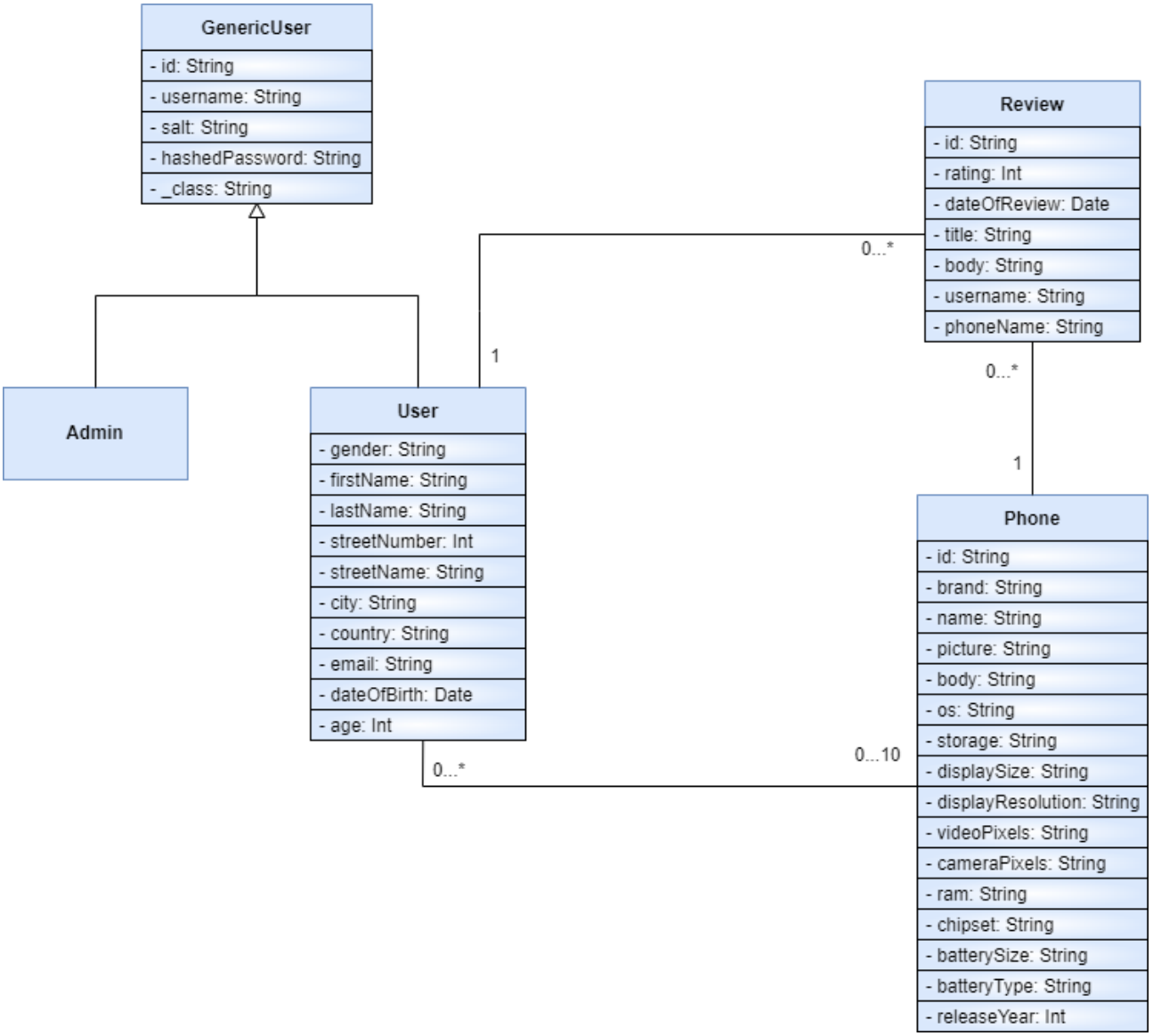
The above use case diagram sums up the actions that the different types of users can do.

To enhance the comprehension of the diagram, here below there are some clarifications:

- we identify each user typology and the action they can do with a specific colour:
  - orange for UnregisteredUser,
  - light blue for RegisteredUser,
  - green for Administrator,
- each action that an unregistered user can do, can be done also by registered users and administrators,
- each action highlighted both in green and light blue can be done by registered users and administrators,
- an Administrator doesn't have a collection of phones and reviews and doesn't have a profile (just username and password), so can't have suggestions,
- "Delete review" and "Delete user" have both the green and light blue colour because:
  - a Registered user can delete one or more reviews written by him/herself,
  - an Administrator can delete one or more reviews of other users,
  - an Administrator can delete the profile of other users
- "View profile" brings the user to his/her personal profile, instead "View user" brings the user to the profile of another user

Class Analysis

The UML class diagram is the following:



The diagram can be described by the following:

- a User can write from zero to many reviews
- a User can add to the watchlist from zero to 10 phones
- a Phone can have from zero to many reviews
- a Phone can be added in zero to many users' watchlists
- a Review can be written from an user
- a review can be related to a phone

Now the classes with their attributes and data types are displayed on appropriate tables.

GenericUser class		
Attribute Name	**Type**	**Description**

id	String	Unique string that identifies an user
username	String	Username chosen by the user
salt	String	Randomly generated salt
hashedPassword	String	User password in an encrypted form
_class	String	Identifies if an user is an admin or not

The Admin class ereditates all the attributes of the GenericUser class.

Same does the User class, but for this one there are also the following attributes:

User class		
Attribute Name	**Type**	**Description**
gender	String	Identifies the gender of the user
firstName	String	User first name
lastName	String	User last name
streetNumber	int	User address street number
streetName	String	User address street name
city	String	User address city name
country	String	User address country name
email	String	User email address
dateOfBirth	Date	User date of birth
age	int	User age

Review class		
Attribute Name	**Type**	**Description**
id	String	Unique string that identifies an user
rating	int	Rating of the review
dateOfReview	Date	Date on which the review has been written
title	String	Title of the review
body	String	Body text of the review
username	String	Username chosen by the user
phoneName	String	Phone commercial name

Phone class		
Attribute Name	**Type**	**Description**
id	String	Unique string that identifies a phone
brand	String	Phone brand
name	String	Phone commercial name
picture	String	Phone image
body	String	Phone dimensions
os	String	Phone os
storage	String	Phone storage capacity
displaySize	String	Phone display size
displayResolution	String	Phone display resolution
cameraPixels	String	Phone camera Pixels
videoPixels	String	Phone video resolution
ram	String	Phone ram capacity
chipset	String	Phone chipset
batterySize	String	Phone battery size
batteryType	String	Phone battery typology
releaseYear	int	Phone release year

Data model

The data model section includes a presentation of the document collections and the graph nodes stored in the database.

The types of database chosen are two.

We decided to use MongoDB as DBMS for the document database part in order to store the information about the phones, the reviews and the users.

The reason why we chose a document database is reflected in its flexibility and capacity to make complex queries.

To better manage the social part where users can add phones to their watchlist and follow other users we opted for a graph database managed using Neo4j.

DocumentDB collections

In MongoDB we created the following three collections:

- reviews
- phones
- users

The **reviews collection** is composed by more than 65 thousand documents and it has the structure below:

```
{
      "_id": "",
      "rating": "",
      "dateOfReview": "",
      "title": "",
      "body": "",
      "username": "",
      "phoneName": ""
}
```

The **phones collection** figures more than 10 thousand elements with the following structure, which includes embedded documents for the reviews:

```
{
      "_id": "",
      "brand": "",
      "name": "",
      "picture": "",
      "body": "",
      "os": "",
      "storage": "",
      "displaySize": "",
      "displayResolution": "",
      "cameraPixels": "",
      "videoPixels": "",
      "ram": "",
      "chipset": "",
      "batterySize": "",
      "batteryType": "",
      "releaseYear": "",
      "reviews": [ { "_id": "",
                     "rating": "",
                     "dateOfReview": "",
                     "title": "",
```

```
                    "body": "",
                    "username": "" },

                  { "_id": "",
                    "rating": "",
                    "dateOfReview": "",
                    "title": "",
                    "body": "",
                    "username": "" } ] }
```

The **users collection** instead has 50 thousand documents and is organised in the following way:

```
{
      "_id": "",
      "username": "",
      "salt": "",
      "hashedPassword": "",
      "_class": "",
      "gender": "",
      "firstName": "",
      "lastName": "",
      "streetNumber": "",
      "streetName": "",
      "city": "",
      "country": "",
      "email": "",
      "dateOfBirth": "",
      "age": "",
      "reviews": [ {"_id": "",
                    "rating": "",
                    "dateOfReview": "",
                    "title": "",
                    "body": "",
                    "phoneName": "" },

                   {"_id": "",
                    "rating": "",
                    "dateOfReview": "",
                    "title": "",
                    "body": "",
                    "phoneName": "" } ] }
```

As presented, into the phones and users collections we decided to embed the reviews. We have chosen to do this because we allow the user of the application to view each phone with its detailed specifications and the reviews written for it. At the same time we show the details of a user with its reviews.

MongoDB keeps frequently accessed data, referred to as the working set, in RAM. When the working set of data and indexes grow beyond the physical assigned RAM, performance is reduced as disk accesses start to occur.
To solve this issue and avoid having unbounded arrays that could exceed the maximum document size limit, we exploited the subset pattern, storing a subset of the reviews in phones and users collections, and the older ones only in the reviews collection.
So we decided to embed the 50 most recent reviews in both cases to improve the performances of the application, while offering as many features as possible to the user.
In this way we introduced redundancies and denormalized data, but at the same time we improved the performances, because in most cases we don't have to do join operations to see user reviews and phone reviews.
Furthermore we were able to improve read operations, that are the most frequent in an application like ours, with the use of indexes.

GraphDB entities and relations

In Neo4j we stored just the necessary information to perform the queries we wanted to execute. These are the required entities represented by nodes:

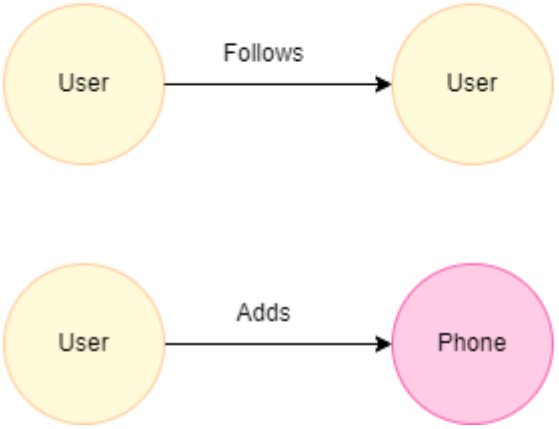
- User
- Phone

Of course we don't store all the attributes and we take only the following ones:

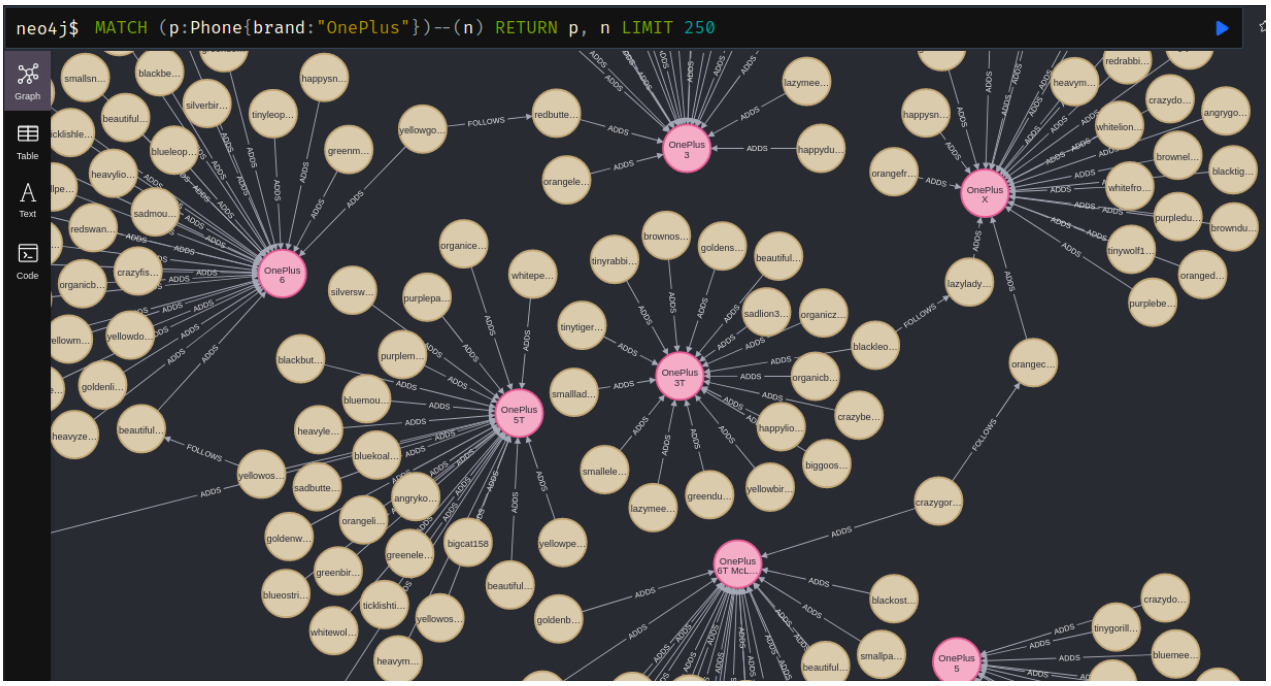
- for the users: "id", "username"
- for the phones: "id", "name", "brand", "picture", "releaseYear"

The relationships involved are:

- **User - [:FOLLOWS] -> User**, which represents a user following another user
- **User - [:ADDS] -> Phone**, which represents a user adding a phone to the watchlist



Here below is presented a snapshot of the graph database taken from Neo4j:



Distributed database design

The design of the distributed database is suggested by the requirements of the application.

Replica set

The decision we made is to have 3 virtual replicas with on each one a MongoDB instance hosted. The Neo4j part is present only on one replica because at the moment we have the free edition.

The three virtual machines are provided by the University of Pisa.

The replica set is composed of a primary replica that acts as the server that takes

client requests, and two secondaries which are the servers that keep copies of the primary's data.

Virtual Machine	IP Address	Port	OS
Replica-0	172.16.4.105	27020	Ubuntu
Replica-1	172.16.4.106	27020	Ubuntu
Replica-2	172.16.4.107	27020	Ubuntu

Replica configuration

The configuration is shown below:

```
rsconfig = {
    _id: "phoneworld",
    members: [
        {
            _id:0,
            host: "172.16.4.105:27020",
            priority:1
        },
        {
            _id:1,
            host: "172.16.4.106:27020",
            priority:2
        },
        {
            _id:2,
            host: "172.16.4.107:27020",
            priority:5
        }
    ]
};
```

We decided to give the highest priority to the VM 172.16.4.107, so unless a problem arises, the replica on this machine will be elected as primary.

As already mentioned when Handling the CAP theorem issue, the application is read-heavy, so we wanted to ensure high availability and partition protection.

Since we are oriented on the AP side of the CAP theorem, we may return data not updated to the latest version in case of partitioning. For this reason, we decided to adopt the Eventual Consistency paradigm.

We decided to introduce both *Write Concern* and *Read Preferences* constraints and set this configuration:

mongodb://admin:password@172.16.4.105:27020,172.16.4.106:27020,172.16.4.107:27020/?authSource=PhoneWorld&replicaSet=lsmdb&w=1&readPreference=nearest&retryWrites=true&ssl=false

- **write concern**:

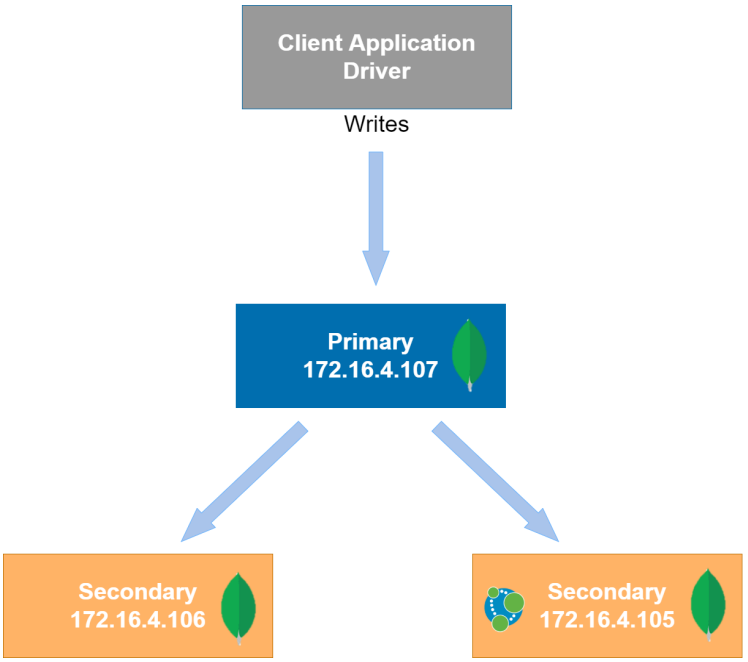
  w=1

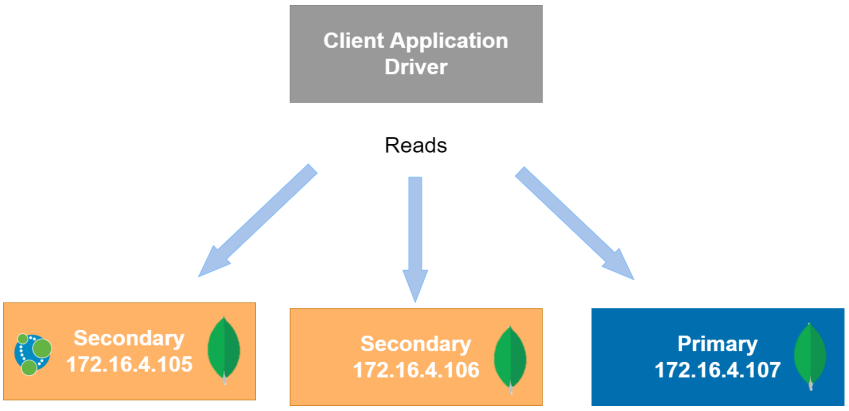
  A write request returns once the first copy is written. The other two writes can happen later. However, data might be lost if a node fails before the second write. To partially remedy this problem, we have decided to include the *retryWrites=true* option in the connection string for MongoDB. In this way the Retryable writes allow MongoDB drivers to automatically retry certain write operations a single time if they encounter network errors, or if they cannot find a healthy primary in the replica sets.

- **read concern**:

  readPreference = nearest

  Read from any member node from the set of nodes which respond the fastest, so the one with the least network latency. A read request reads only a single version of data, so the application might not get the latest copy.

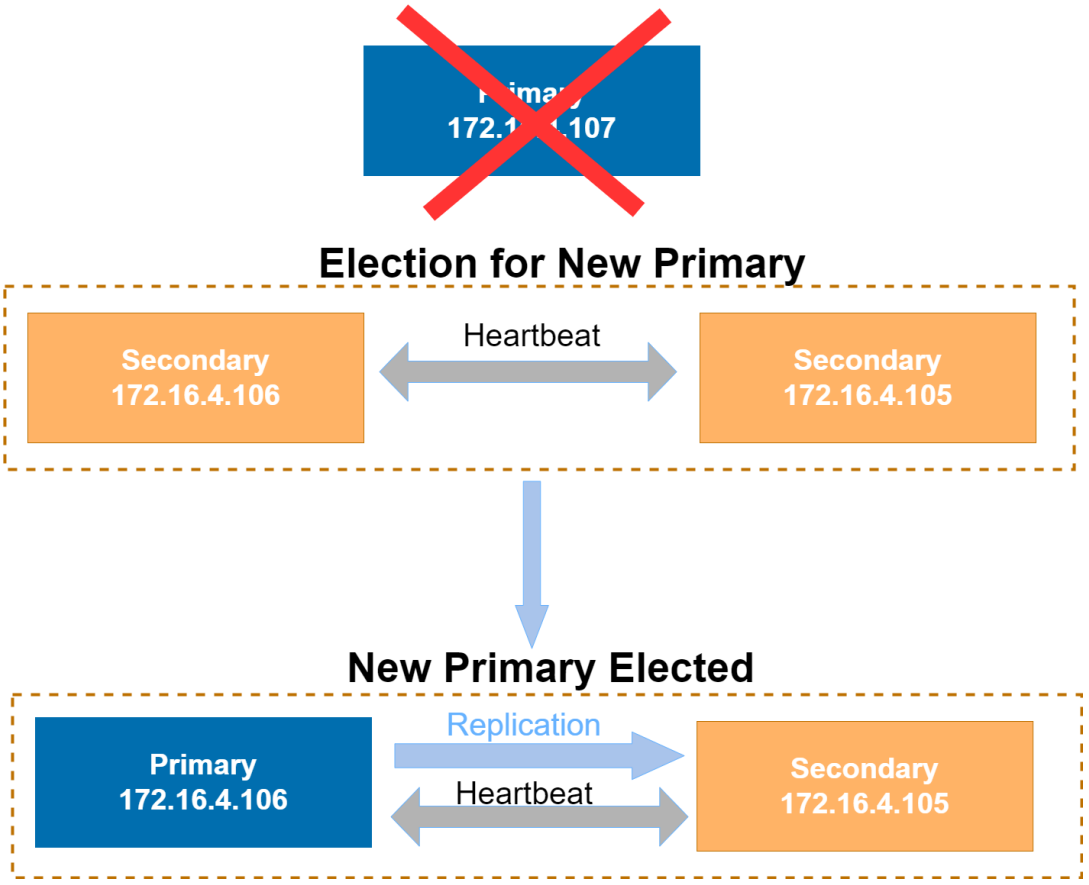




Replica crash

In the case of failure of the primary node, one of the two secondary ones is elected as the new primary one.

Since we have assigned a priority on each replica, we already know which of the two secondaries will be promoted. In our case, when the primary is not available, the VM 172.16.4.106 will be marked as the new primary.



Sharding proposal

Our sharding proposal for this application, which is concurrent with data replication, would allow for load-balancing purposes, in order to achieve better performances.

To implement the sharding, we thought about selecting **three different sharding keys**, one for each collection of our document database:

- phones collection, key: “phoneName”, which is unique among the phones,
- users collection, key: “username”, which is unique among the users,
- reviews collection, key: “_id”, a field automatically generated by MongoDB.

We chose as a sharding partitioning method the **hashing**, so to ensure a good load balance among servers. The hash function returns an hash value to determine where to place a document, resulting in evenly distributed data among the shards.

In the case of adoption of the sharding, it can be a possibility also to rethink about the collections organization.

Since designing and using sharding with our current database design results not optimal, a possible solution could be to embed all the reviews into the phones collection, splitting the documents if the size of the array grows too much. In each review, we would have the username of the user who wrote the review, which is unique and cannot be changed over time. In this way, we could also access the profile of the user from the reviews. Linked to this, we could remove the embedded reviews in the user collection and delete the reviews collection.

Pursuing this solution, we would avoid the possibility of accessing different shards, for example in the case of addition of a new review. On the other hand, comparing with the actual solution, it would be more difficult to access all the reviews written by a specific user.

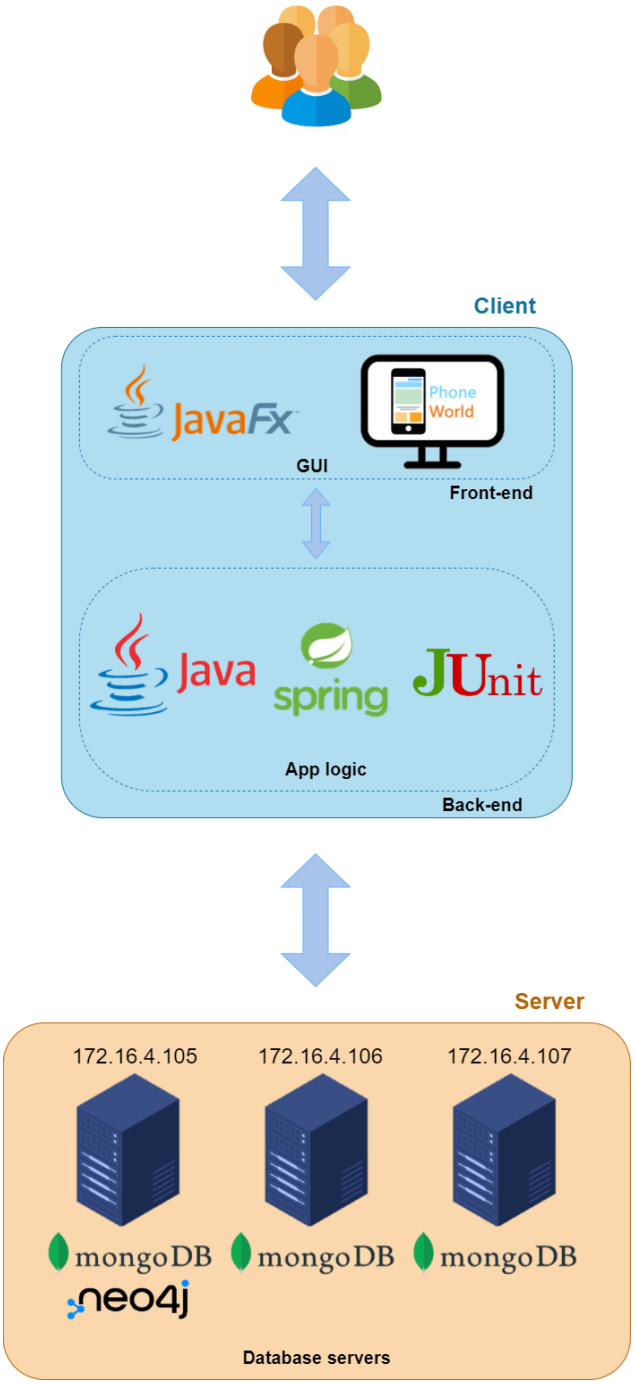
Overall platform architecture

The application is developed in Java using Intellij as IDE.

To store and manage data, as explained in the Data model section, we used MongoDB and Neo4j as our NoSQL DBMSs.

As mentioned in the previous section, there are 3 replicas for mongoDB and just one for Neo4j.

Below there's a graphic representation of the overall platform architecture.



Framework used

Spring Data MongoDB is the framework used to facilitate the mapping of the java objects with documents in the database.

In addition, is used JavaFx for the graphic interface and JUnit as unit testing framework.


spring


JavaFx


JUnit

Implementation

In this section is presented the implementation of the packages and the java classes necessary for the application, including some snapshots of the code.

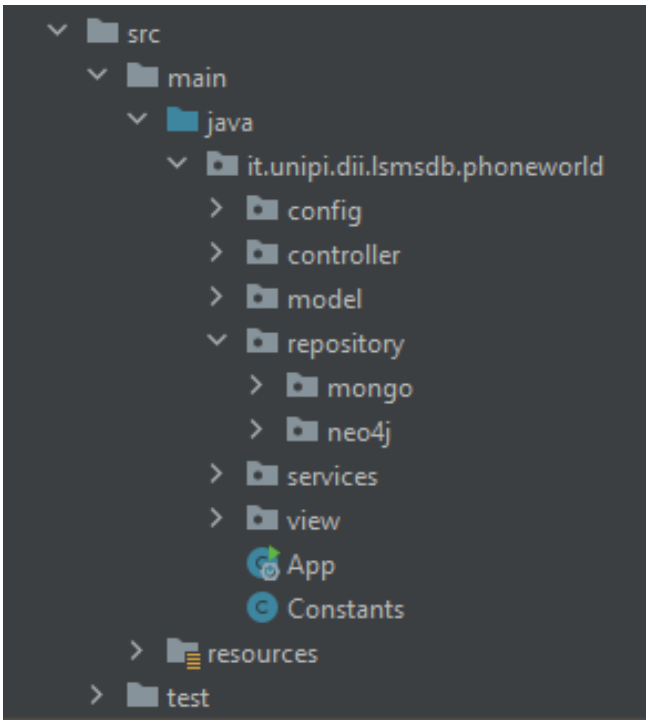
Is also highlighted the implementation of the queries done for the two different databases and the CRUD operations for MongoDB.

In addition, because this is a maven project, in this section there is also the POM file, in order to show all the dependencies needed.

To conclude, we exploit the use of some indexes and make some statistics and performance tests on the cluster.

Application project source organization

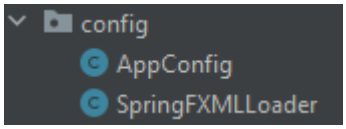
The picture below represents the packages as are presented into IntelliJ, the IDE used for the development of the application.



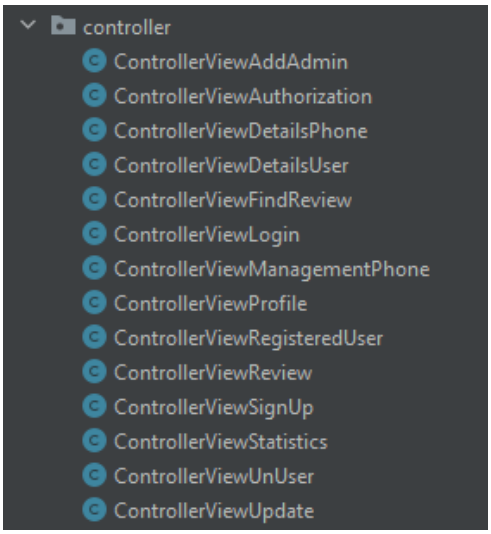
First of all, it is due to say that we implemented as package prefix the reverse domain of the University of Pisa and organized packages by layers.

Focusing on the path *src/main/java/it.unipi.dii.lsmsdb.phoneworld* we have the following packages:

- *config*: this package is responsible for running the view and configuring the application with the use of Spring and JavaFx.



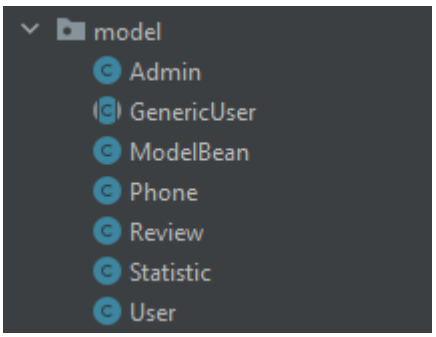
- *controller*: here there are all the controllers useful to manage the view shown in the application.



- *model*: there are all the classes representing the entities needed for the application.

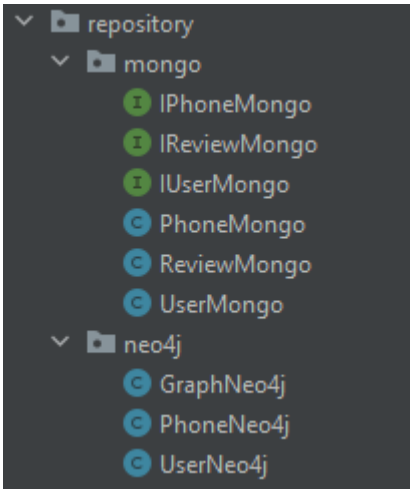
  ModelBean is responsible for keeping the objects in memory. The putBean method allows to insert an object into the map. The getBean method allows you to retrieve an object from it.

  Statistic is the class that represents the structure of the statistics showed in the application.

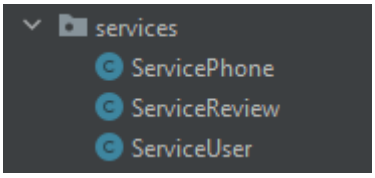


- *repository*: is divided into the two packages, *mongo* and *neo4j,* which contain the classes to manage the persistence. They include the code of the CRUD operations done in MongoDB and in Neo4j and all the queries to both

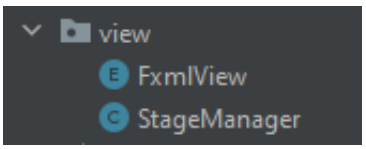
databases. Inside *mongo* there are also three interfaces that extend the MongoRepository interface from Spring Data MongoDB.



- *services*: three classes to reflect the three entities of the application. They offer to them many services which are used by many views, ready to be called where needed by the controllers. So they are not directly written into the controllers because they are not straight connected to them.



- *view*: here there is *FxmlView* that is an enum class representing a group of constants that map the views with their respective fxml files. *StageManager*, instead, implements methods to manage the view.



Inside this structure there are also:

- *App*: this is the main class, where we establish the connection with Neo4j and we run the application.
- *Constants*: a class that has constant strings that are used as keys to get the object's values from the map in the ModelBean class.

The folder *resources* contains all the fxml files which represent the views that are loaded with the use of the constants within the FxmlView class.

The path *src/test* have the unit tests classes better described into the Unit tests section.

Java model classes

Here the declaration of the java model classes built, firstly with a list and then with the code.

Of course the classes have getters and setters which are not included in the snapshots for the sake of simplicity.

Java class	Description
GenericUser	IdentifiesManages the abstract class for the user
Admin	Identifies the administrator
User	Identifies the normal registered user
Review	Identifies the reviews
Phone	Identifies the phones
ModelBean	Manages objects
Statistic	Represents a generic statistic result

GenericUser.java class

```
@Document(collection = "users")
public abstract class GenericUser {

    @Id
    protected String id;
    protected String username;
    protected String salt;
    protected String hashedPassword;
    protected String _class;

    protected GenericUser() {
    }

    protected GenericUser(String username, String salt, String hashedPassword, String _class) {
        this.username = username;
        this.salt = salt;
        this.hashedPassword = hashedPassword;
        this._class = _class;
    }
```

Admin.java class

```java
@Document(collection = "users")
@TypeAlias("admin")
public class Admin extends GenericUser{

    public Admin() {
    }

    public Admin(String username, String salt, String hashedPassword, String _class) {
        super(username, salt, hashedPassword, _class);
    }
```

User.java class

```java
@Document(collection = "users")
@TypeAlias("user")
public class User extends GenericUser{

    public User() {
    }

    public User(String username, String salt, String hashedPassword, String _class, String gender,
                String firstName, String lastName, int streetNumber, String streetName, String city,
                String country, String email, Date dateOfBirth, int age) {
        super(username, salt, hashedPassword, _class);
        this.gender = gender;
        this.firstName = firstName;
        this.lastName = lastName;
        this.streetNumber = streetNumber;
        this.streetName = streetName;
        this.city = city;
        this.country = country;
        this.email = email;
        this.dateOfBirth = dateOfBirth;
        this.age = age;
    }

    private String gender;
    private String firstName;
    private String lastName;
    private int streetNumber;
    private String streetName;
    private String city;
    private String country;
    private String email;
    private Date dateOfBirth;
    private int age;
    private List<Review> reviews = new ArrayList<>();
```

Review.java class

```
@Document(collection = "reviews")
public class Review {

    @Id
    private String id;
    private int rating;
    private Date dateOfReview;
    private String title;
    private String body;
    private String username;
    private String phoneName;

    @PersistenceConstructor
    public Review(String id, int rating, Date dateOfReview, String title, String body, String username,
                  String phoneName) {
        this.id = id;
        this.rating = rating;
        this.dateOfReview = dateOfReview;
        this.title = title;
        this.body = body;
        this.username = username;
        this.phoneName = phoneName;
    }
```

Phone.java class

```java
@Document(collection = "phones")
public class Phone {

    @Id
    private String id;
    private String brand;
    private String name;
    private String picture;
    private String body;
    private String os;
    private String storage;
    private String displaySize;
    private String displayResolution;
    private String cameraPixels;
    private String videoPixels;
    private String ram;
    private String chipset;
    private String batterySize;
    private String batteryType;
    private int releaseYear;
    private List<Review> reviews = new ArrayList<>();

    public Phone() {
    }

    public Phone(String brand, String name, String picture, String body, String os, String storage,
                 String displaySize, String displayResolution, String cameraPixels, String videoPixels,
                 String ram, String chipset, String batterySize, String batteryType, int releaseYear) {
        this.brand = brand;
        this.name = name;
        this.picture = picture;
        this.releaseYear = releaseYear;
        this.body = body;
        this.os = os;
        this.storage = storage;
        this.displaySize = displaySize;
        this.displayResolution = displayResolution;
        this.cameraPixels = cameraPixels;
        this.videoPixels = videoPixels;
        this.ram = ram;
        this.chipset = chipset;
        this.batterySize = batterySize;
        this.batteryType = batteryType;
    }
```

ModelBean.java

```
public class ModelBean {

    private Map<String, Object> mapBean = new HashMap<String, Object>();

    public void putBean(String key, Object o) {
        this.mapBean.put(key, o);
    }

    public Object getBean(String key) {
        return mapBean.get(key);
    }
}
```

This class exists to maintain the state of the objects throughout the use of the application, for example to keep the current user.

Statistic.java

```
public class Statistic {

    private SimpleStringProperty name;
    private SimpleIntegerProperty param2;
    private SimpleDoubleProperty param3;

    public Statistic(String name, int param2, Double param3) {
        this.name = new SimpleStringProperty(name);
        this.param2 = new SimpleIntegerProperty(param2);
        this.param3 = new SimpleDoubleProperty(param3);
    }
```

We need this class in order to be able to easily take the usual three parameters we use to show statistics.

Project Object Model (POM) File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>it.unipi.dii.lsmsdb</groupId>
    <artifactId>phoneworld</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>phoneworld</name>
    <description>Project LSMSDB Group 24</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>

        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-graphics</artifactId>
            <version>17-ea+13</version>
        </dependency>

        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-controls</artifactId>
            <version>17-ea+13</version>
        </dependency>

        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-fxml</artifactId>
            <version>17-ea+13</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
```

```
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>de.flapdoodle.embed</groupId>
            <artifactId>de.flapdoodle.embed.mongo</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.neo4j.driver</groupId>
            <artifactId>neo4j-java-driver</artifactId>
            <version>4.4.2</version>
        </dependency>

        <dependency>
            <groupId>org.neo4j.test</groupId>
            <artifactId>neo4j-harness</artifactId>
            <version>4.4.2</version>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.slf4j</groupId>
                    <artifactId>slf4j-nop</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-mongodb</artifactId>
            <version>3.3.0</version>
        </dependency>

        <dependency>
            <groupId>org.mongodb</groupId>
            <artifactId>mongodb-driver-sync</artifactId>
            <version>4.4.0</version>
        </dependency>
        <dependency>
```

```
            <groupId>org.neo4j</groupId>
            <artifactId>neo4j-graphdb-api</artifactId>
            <version>4.3.7</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>

    <repositories>
        <repository>
            <id>spring-milestone</id>
            <name>Spring Maven MILESTONE Repository</name>
            <url>https://repo.spring.io/libs-milestone</url>
        </repository>
    </repositories>

</project>
```

DocumentDB CRUD operations

Here below are displayed the most common CRUD operations.

Create

The following method is called every time a user writes a review in order to save it into the database.

```
public boolean addReview(Review review) {
    boolean result = true;
    try {
        reviewMongo.save(review);
    } catch (Exception e) {
        e.printStackTrace();
        result = false;
    }
    return result;
}
```

Read

This method returns a review thanks to the input of an id.

```java
public Optional<Review> findReviewById(String id) {
    Optional<Review> review = Optional.empty();
    try {
        review = reviewMongo.findById(id);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return review;
}
```

Update

This method updates the fields of a review, firstly found thanks to the id.

```java
public boolean updateReview(String id, Review newReview) {
    boolean result = true;
    try {
        Optional<Review> review = reviewMongo.findById(id);
        if (review.isPresent()) {
            Review resultReview = review.get();
            Review.ReviewBuilder builder = new Review.ReviewBuilder(newReview);
            builder.id(id).phoneName(resultReview.getPhoneName()).username(resultReview.getUsername());
            this.addReview(builder.build());
        }
    } catch (Exception e) {
        e.printStackTrace();
        result = false;
    }
    return result;
}
```

Delete

This method deletes a review found taking as input the id.

```java
public boolean deleteReviewById(String id) {
    boolean result = true;
    try {
        reviewMongo.deleteById(id);
    } catch (Exception e) {
        e.printStackTrace();
        result = false;
    }
    return result;
}
```

MongoDB query analysis

Here are some of the most relevant queries implemented in Java for MongoDB.

- **Display countries with the youngest users (admin)**

```java
public Document findYoungerCountriesByUsers(int number) {
    MatchOperation matchOperation = match(new Criteria("_class").is("user"));
    GroupOperation groupOperation = group("$country").avg("$age").as("avgAge");
    SortOperation sortOperation = sort(Sort.by(Sort.Direction.ASC, "avgAge"));
    LimitOperation limitOperation = limit(number);
    ProjectionOperation projectionOperation = project()
            .andExpression("_id").as("country")
            .and(ArithmeticOperators.Round.roundValueOf("avgAge").place(1)).as("age");
    Aggregation aggregation = newAggregation(matchOperation, groupOperation, sortOperation,
            limitOperation, projectionOperation);
    AggregationResults<User> result = mongoOperations
            .aggregate(aggregation, "users", User.class);
    return result.getRawResults();
}
```

```
db.users.aggregate(
[ {
   $group: {
      _id: '$country',
      avgAge: { $avg: '$age' }
   }
}, {
   $sort: {
      avgAge: 1
   }
}, {
   $limit: 5
}, {
   $project: {
      _id: 0,
      country: '$_id',
      age: { $round: ['$avgAge', 1] }
   }}])
```

This is an aggregation pipeline to recover, from the users collection, the younger countries, taking into account the age of each user and making the average of it, with respect to their country. This insight can tell us where the

application is more popular among young people.

- **Top countries with the highest number of users (admin)**

```
public Document findTopCountriesByUsers(int number) {
    MatchOperation matchOperation = match(new Criteria("_class").is("user"));
    GroupOperation groupOperation = group("$country").count().as("numUsers");
    SortOperation sortOperation = sort(Sort.by(Sort.Direction.DESC, "numUsers"));
    LimitOperation limitOperation = limit(number);
    ProjectionOperation projectionOperation = project()
            .andExpression("_id").as("country")
            .andExpression("numUsers").as("users").andExclude("_id");
    Aggregation aggregation = newAggregation(matchOperation, groupOperation, sortOperation,
            limitOperation, projectionOperation);
    AggregationResults<User> result = mongoOperations
            .aggregate(aggregation, "users", User.class);
    return result.getRawResults();
}
```

```
db.users.aggregate(
[{
   $group: {
      _id: '$country',
      numUsers: {
         $sum: 1
      }
   }
}, {
   $sort: {
      numUsers: -1
   }
}, {
   $limit: 5
}, {
   $project: {
      _id: 0,
      country: '$_id',
      users: '$numUsers'
   }
}])
```

This is another query to analyse users and their country, but this time is able to

recover the first "n" top countries, identified by the ones with the highest number of users. It can be useful to discover geographically which is the most active country for this application.

- **find top rated phones**

```
public Document findTopPhonesByRating(int minReviews, int results) {
    GroupOperation groupOperation = group("$phoneName").avg("$rating")
            .as("avgRating").count().as("numReviews");
    MatchOperation matchOperation = match(new Criteria("numReviews").gte(minReviews));
    ProjectionOperation projectionOperation = project()
            .andExpression("_id").as("phoneName").andExclude("_id")
            .andExpression("numReviews").as("reviews")
            .and(ArithmeticOperators.Round.roundValueOf("avgRating").place(1)).as("rating");
    SortOperation sortOperation = sort(Sort.by(Sort.Direction.DESC, "rating", "reviews"));
    LimitOperation limitOperation = limit(results);
    Aggregation aggregation = newAggregation(groupOperation, matchOperation, projectionOperation,
            sortOperation, limitOperation);
    AggregationResults<Review> result = mongoOperations
            .aggregate(aggregation, "reviews", Review.class);
    return result.getRawResults();
}
```

```
db.reviews.aggregate(
[{
        $group: {
        _id: '$phoneName',
        avgRating: {
        $avg: '$rating'
        },
        numReviews: {
        $sum: 1
        }}
}, {
        $match: {
        numReviews: {
        $gte: 10
        }}
}, {
        $project: {
        _id: 0,
        phoneName: '$_id',
        rating: {
        $round: [ '$avgRating', 1 ]},
```

```
        reviews: '$numReviews'
        }
}, {
        $sort: {
        rating: -1,
        reviews: -1
        }
}, {
        $limit: 10
}])
```

This query interests the reviews collection and shows the first ten phones with the higher average ratings. To increase the validity of the result, only phones with at least ten reviews are included. The results are sorted descending firstly by rating and secondly by number of reviews, also descending.

- **find most active users (admin)**

```
public Document findMostActiveUsers(int results) {
    GroupOperation groupOperation = group("$username").count().as("numReviews").
            avg("$rating").as("ratingUser");
    SortOperation sortOperation = sort(Sort.by(Sort.Direction.DESC, "numReviews"));
    LimitOperation limitOperation = limit(results);
    ProjectionOperation projectionOperation = project()
            .andExpression("_id").as("username")
            .andExpression("numReviews").as("reviews").andExclude("_id")
            .and(ArithmeticOperators.Round.roundValueOf("ratingUser").place(1)).as("rating");
    Aggregation aggregation = newAggregation(groupOperation, sortOperation, limitOperation,
            projectionOperation);
    AggregationResults<Review> result = mongoOperations
            .aggregate(aggregation, "reviews", Review.class);
    return result.getRawResults();
}
```

```
db.reviews.aggregate(
[{
        $group: {
        _id: '$username',
        numReviews: {
        $sum: 1
        },
        ratingUser: {
```

```
            $avg: '$rating'
            }}
}, {
            $sort: {
            numReviews: -1
            }
}, {
            $limit: 5
}, {
            $project: {
            _id: 0,
            username: '$_id',
            reviews: '$numReviews',
            rating: {
            $round: [
            '$ratingUser',
            1
            ]}}
}])
```

This aggregation pipeline shows the most active users in terms of the number of reviews that they have written.

- **find top rated brands**

```java
public Document findTopRatedBrands(int minReviews, int results) {
    UnwindOperation unwindOperation = unwind("reviews");
    GroupOperation groupOperation = group("$brand").avg("$reviews.rating")
            .as("avgRating").count().as("numReviews");
    MatchOperation matchOperation = match(new Criteria("numReviews").gte(minReviews));
    ProjectionOperation projectionOperation = project().andExpression("_id").as("brand")
            .andExclude("_id").andExpression("numReviews").as("reviews")
            .and(ArithmeticOperators.Round.roundValueOf("avgRating").place(1)).as("rating");
    SortOperation sortOperation = sort(Sort.by(Sort.Direction.DESC, "rating",
            "reviews"));
    LimitOperation limitOperation = limit(results);
    Aggregation aggregation = newAggregation(unwindOperation, groupOperation, matchOperation,
            projectionOperation, sortOperation, limitOperation);
    AggregationResults<Phone> result = mongoOperations
            .aggregate(aggregation, "phones", Phone.class);
    return result.getRawResults();
}
```

```
db.phones.aggregate(
[{ $unwind: {
      path: '$reviews'
   }},
{ $group: {
      _id: '$brand',
      avgRating: {
         $avg: '$reviews.rating'
      },
      numReviews: {
         $sum: 1
      }}},
{ $match: {
      numReviews: {
         $gte: 10
      }}},
{ $project: {
      _id: 0,
      brand: '$_id',
      rating: {
         $round: [ '$avgRating', 1 ]
      },
      reviews: '$numReviews'
```

```
    }},
{ $sort: {
      rating: -1,
      reviews: -1
    }
}, {
    $limit: 5
}])
```

This query searches into the phone collection and looking at the reviews' ratings of the embedded review documents, it takes only the ones with at least 10 reviews and matching with the phones, in the end shows just five brands of the phones with the highest average ratings.

Neo4j query analysis

In this paragraph are presented the relevant queries implemented in Neo4j.

- **Display most followed users**

```
MATCH (u1:User)<-[r:FOLLOWS]-(u2:User)
RETURN u1.username AS username, u1.id AS id, COUNT(r) AS followers
ORDER BY followers DESC
LIMIT 10
```

This query shows the first ten users with the higher number of followers.

- **Display suggested users based on friends of friends**

```
MATCH (u1:User{id: $id})-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(u3:User)
WHERE NOT EXISTS ((u1)-[:FOLLOWS]->(u3)) AND u1.id <> u3.id
WITH u3, rand() AS number
ORDER BY number
RETURN DISTINCT u3.id AS id, u3.username AS username
LIMIT 10
```

This aggregation shows a list of 10 suggested users, taken from the users followed by the following of the current user.

- **Display suggested users based on their favourite brand**

```
MATCH (u1:User{id:"$id"})-[:ADDS]->(p:Phone)
WITH COUNT (p.brand) AS numPhones, p.brand AS favBrand, u1
ORDER BY numPhones DESC
LIMIT 1
MATCH(u2:User)-[:ADDS]->(p:Phone{brand:favBrand})
WHERE u2.id <> u1.id AND NOT EXISTS ((u1)-[:FOLLOWS]->(u2))
RETURN COUNT (p.brand) AS phones, p.brand AS favouriteBrand, u2.id AS id, u2.username AS username
ORDER BY phones DESC
LIMIT 10
```

This, instead, firstly identifies the most present brand in a user watchlist to identify it as his/her favourite brand. After that we take the 10 users that have the highest number of ADDS relationships towards a phone of the favourite brand of the current user, that are not already followed by him/her.

- **Display suggested phones based on friends' watchlist**

```
MATCH (u1:User{id:"$id"})-[:FOLLOWS]->(u2:User)-[:ADDS]->(p:Phone)
WHERE u1.id <> u2.id AND NOT EXISTS ((u1)-[:ADDS]->(p))
WITH p
ORDER BY p.releaseYear DESC
RETURN DISTINCT p
LIMIT 10
```

This query returns, ordered from the most to the less recent, 10 phones which have been added to the watchlist of a user followed by the current user. These phones must not be already present in the watchlist of the current user.

- **Display suggested phones based on your favourite brand**

  MATCH (u1:User{id:"$id"})-[:ADDS]->(p:Phone)

  WITH COUNT (p.brand) AS numPhones, p.brand AS favBrand, u1

  ORDER BY numPhones DESC

  LIMIT 1

  MATCH (newPhone:Phone{brand:favBrand})

  WHERE NOT EXISTS ((u1)-[:ADDS]->(newPhone))

  WITH newPhone

  ORDER BY newPhone.releaseYear DESC

  RETURN DISTINCT newPhone

  LIMIT 10

  This query firstly identifies the most present brand in a user watchlist to identify it as his/her favourite brand. Then, this is used to return 10 suggested phones of that favourite brand, of course not already added in the watchlist.

- **Display most appreciated brands**

  MATCH (:User)-[:ADDS]->(p:Phone)

  RETURN p.brand AS brand, COUNT(p.brand) AS numPhones

  ORDER BY numPhones DESC

  LIMIT 10

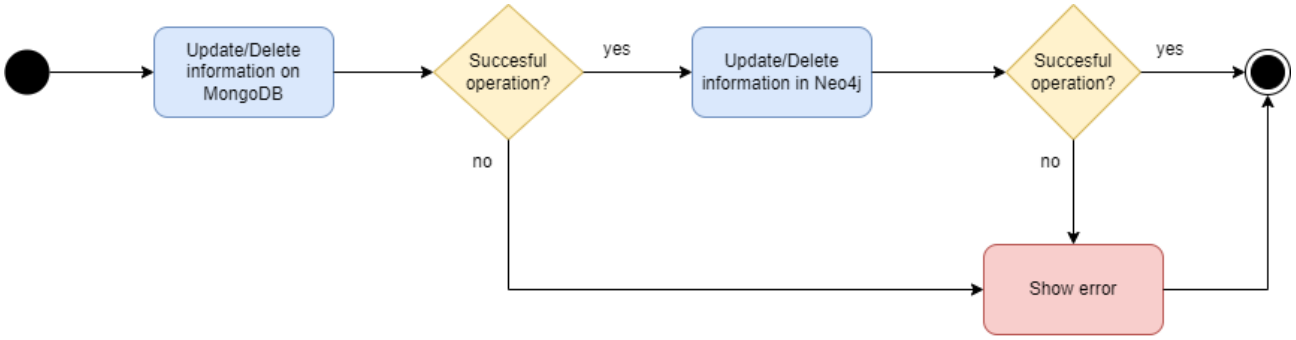
  This query returns the 10 most present brands in users' watchlists.

Database consistency management

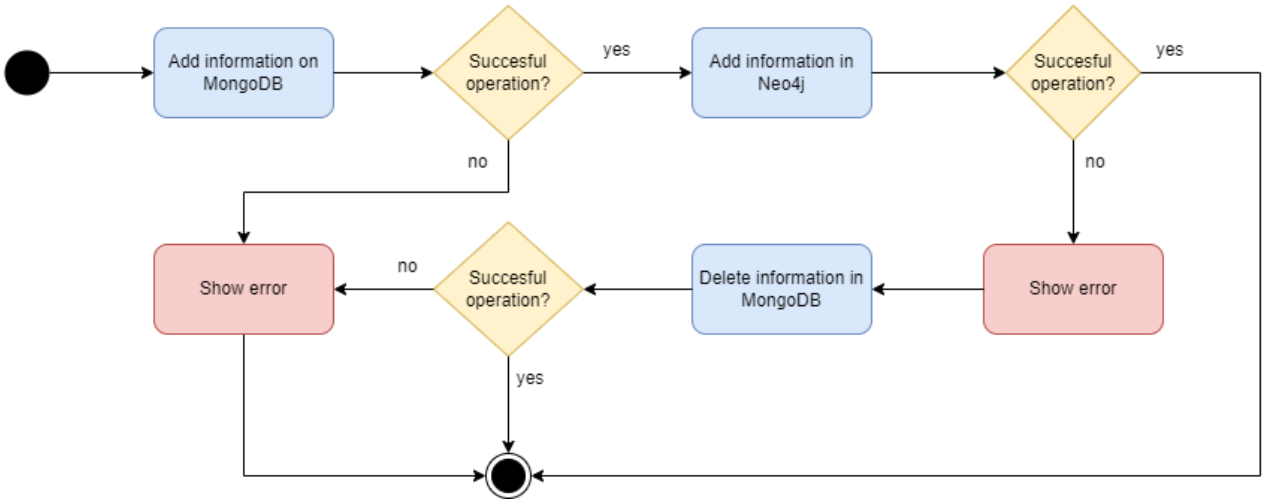
We decided to have data on two different databases, meaning that some information is duplicated, so it became necessary to manage the possible data inconsistency.

In particular, this can happen only for some operations regarding data present both in MongoDB and in Neo4j. This is the case of:

- update/delete a user/phone



- add a user/phone



Index analysis

In order to enhance the read operations, we tried to use some indexes firstly on MongoDB and then on Neo4j too.

MongoDB Compass index analysis

The tests carried out on the Document DBMS MongoDb Compass are done using the "Explain Plan" section. The results are shown in the two tables below, where in the first one are listed all the indexes adopted and in the second one are presented the performance compared with (True) and without (False) the use of indexes.

#	Collection	Index Name	Index Type	Properties	Attribute
1	**phones**	**phoneName**	Single	Unique	name
2	**phones**	**releaseYear**	Single	Not Unique	releaseYear

3	reviews	phoneName	Single	Not Unique	phoneName
4	reviews	username	Single	Not Unique	username
5	users	username	Single	Unique	username

1. **phones → phoneName**

   db.phones.find({name: "Nokia 3210"})

Index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	10332	9
True	1	1	1	0

2. **phones → releaseYear**

   db.phones.find().sort({releaseYear: -1}).limit(10)

Index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	10	0	10332	14
True	10	10	10	0

3. **reviews → phoneName**

   db.reviews.find({phoneName: "Nokia 3210"})

Index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	27	0	64912	40
True	27	27	27	0

4. **reviews → username**

   db.reviews.find({username: "Paolo"})

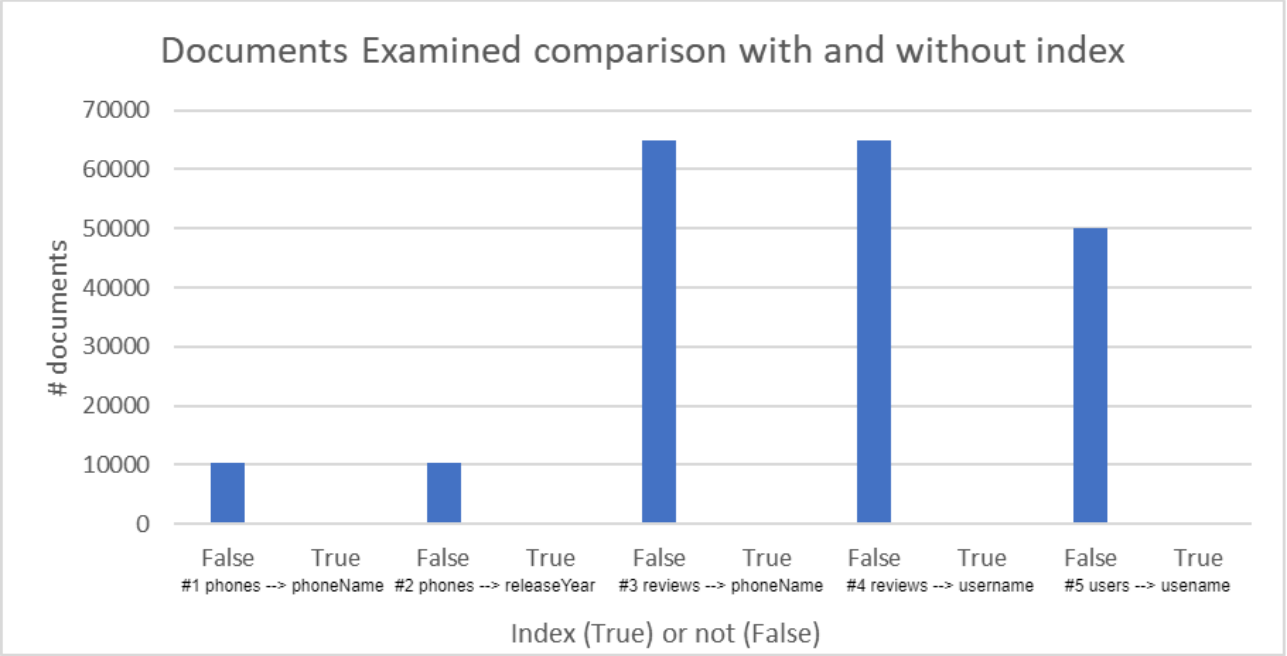
Index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	64912	46
True	1	1	1	0

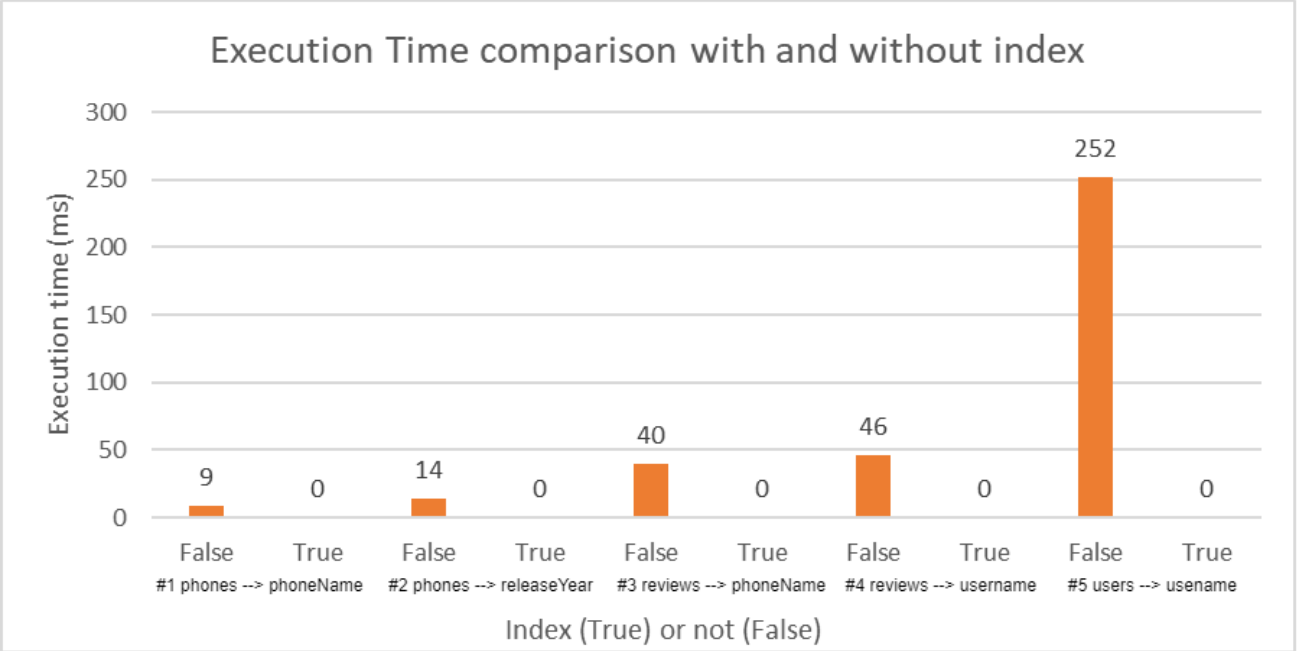
5. users → username

db.users.find({username: "Paolo"})

Index	Document Returned	Index Keys Examined	Documents Examined	Execution Time (ms)
False	1	0	50004	252
True	1	1	1	0

The bar plots below exploit for each index showed the number of documents examined and the execution times in both cases with (True) and without (False) index.





Looking at this analysis we can brightly say that using these indexes speeds up the execution time a lot, of course accepting some more copies of data and so increasing the storage size a bit.

Neo4j index analysis

On Neo4j, instead, we adopted these three different indexes:

#	Entity	Index Name	Type	Uniqueness	EntityType	LabelsOrTypes	Properties
1	**phones**	**brand**	BTREE	NONUNIQUE	NODE	[“Phone”]	[“brand”]
2	**phones**	**phone_id**	BTREE	UNIQUE	NODE	[“Phone”]	[“id”]
3	**users**	**user_id**	BTREE	UNIQUE	NODE	[“User”]	[“id”]

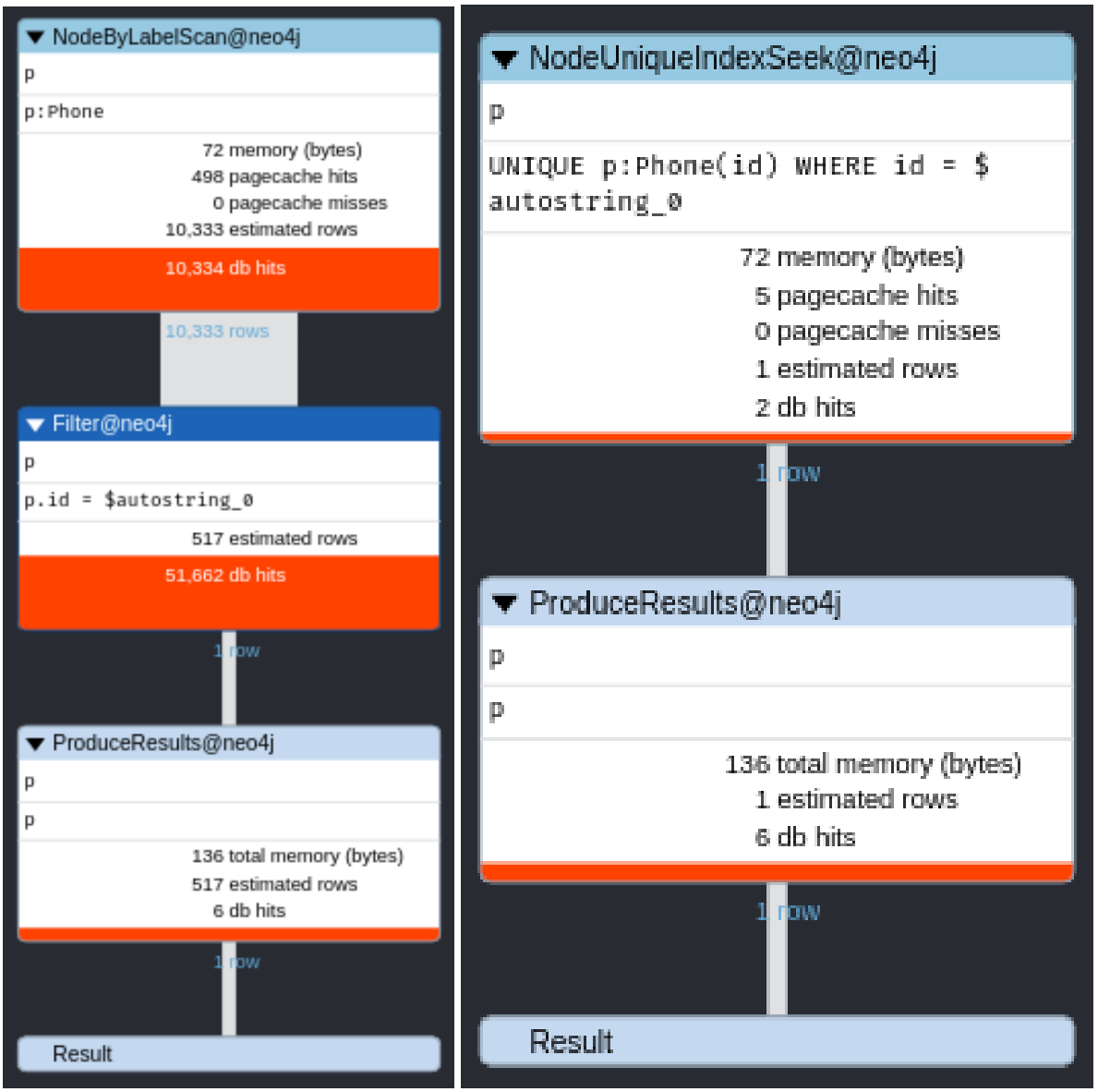
We made queries with and without indexes obtaining the following results:

1) **phones → brand**

   neo4j$ PROFILE MATCH(p:phone{brand: “Nokia”}) RETURN p

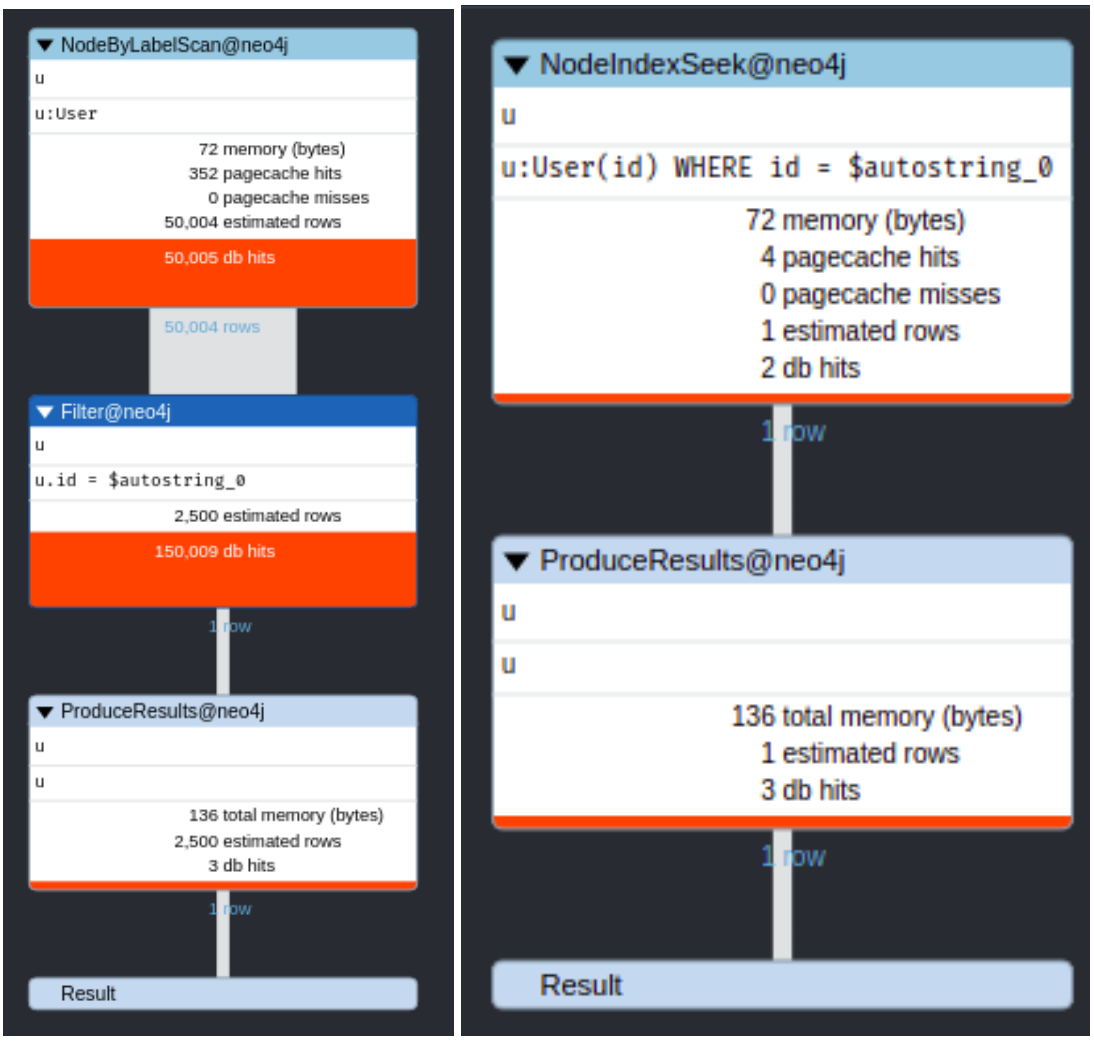
2) phones → phone_id

neo4j$ PROFILE MATCH(p:phone{id:"61dc0d5e6776357ac980be03"}) RETURN p



3) users → user_id

neo4j$ PROFILE MATCH(u:user{id:"61dc0d5e6776357ac980be03"}) RETURN u



We can confirm that also for Neo4j the performances improve a lot using indexes.

Unit tests

To test the application we created 5 classes, where 3 are for MongoDB and the other 2 for Neo4j:

- TestUserMongo.java
- TestReviewMongo.java
- TestPhoneMongo.java
- TestUserNeo4j.java
- TestPhoneNeo4j.java

These classes import the junit framework and the necessary packages.

Here is illustrated only a method included into the TestPhoneMongo.java class as an example.

It shows the addition to the database of a new phone.

Not all the test methods implemented are displayed just for the sake of simplicity.

```
@Test
public void testAddPhone() {
    List<Phone> phones = phoneMongo.getPhoneMongo().findAll();
    phones.forEach(System.out::println);
    assertEquals("resolution", phones.get(0).getDisplayResolution());
    assertEquals(3, phones.size());
}
```

GUI

There are 14 fxml documents, one for each page of the application and each one describe the objects of the relative page in the GUI interface:

- viewAddAdmin.fxml
- viewAuthorization.fxml
- viewDetailsPhone.fxml
- viewDetailsUser.fxml
- viewFindReviews.fxml
- viewLogin.fxml
- viewManagementPhone.fxml
- viewProfile.fxml
- viewRegisteredUser.fxml
- viewReview.fxml
- viewSignUp.fxml
- viewStatistics.fxml
- viewUnregisteredUser.fxml
- viewUpdate.fxml

Associated to these documents there are the following controllers in terms of java classes, responsible to handle the events of the objects in the fxml documents:

- ControllerViewAddAdmin.java
- ControllerViewAuthorization.java
- ControllerViewDetailsPhone.java
- ControllerViewDetailsUser.java
- ControllerViewFindReview.java
- ControllerViewLogin.java
- ControllerViewManagementPhone.java

- ControllerViewProfile.java
- ControllerViewRegisteredUser.java
- ControllerViewReview.java
- ControllerViewSignUp.java
- ControllerViewStatistics.java
- ControllerViewUnUser.java
- ControllerViewUpdate.java

Usage manual

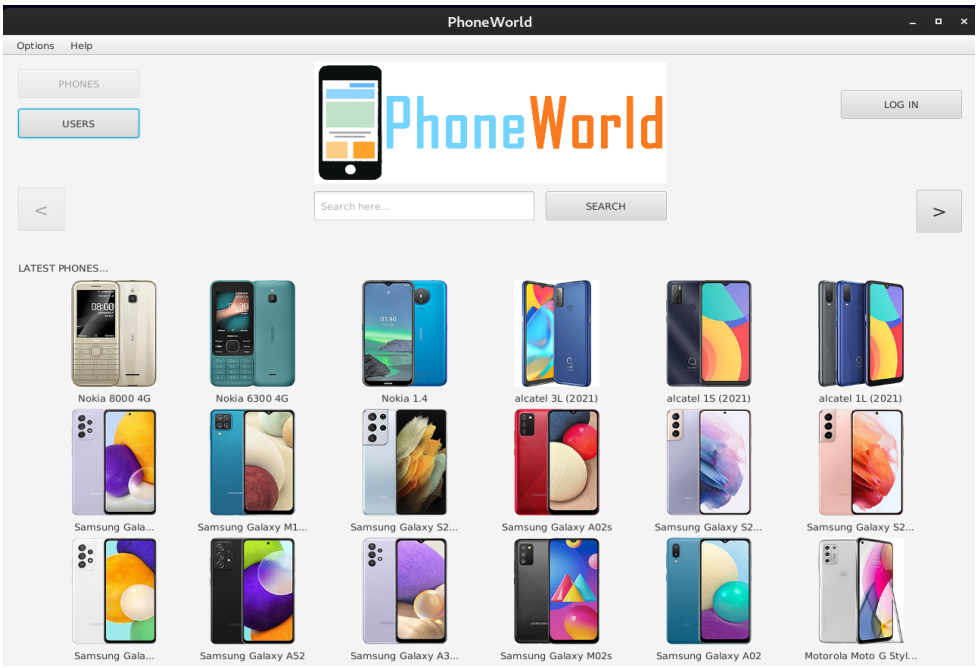
This section is designated as a short manual of usage of the application.

Due to the fact that the Login can be done by users or admins, below are highlighted the GUI windows along with the explanation of the functionalities offered.

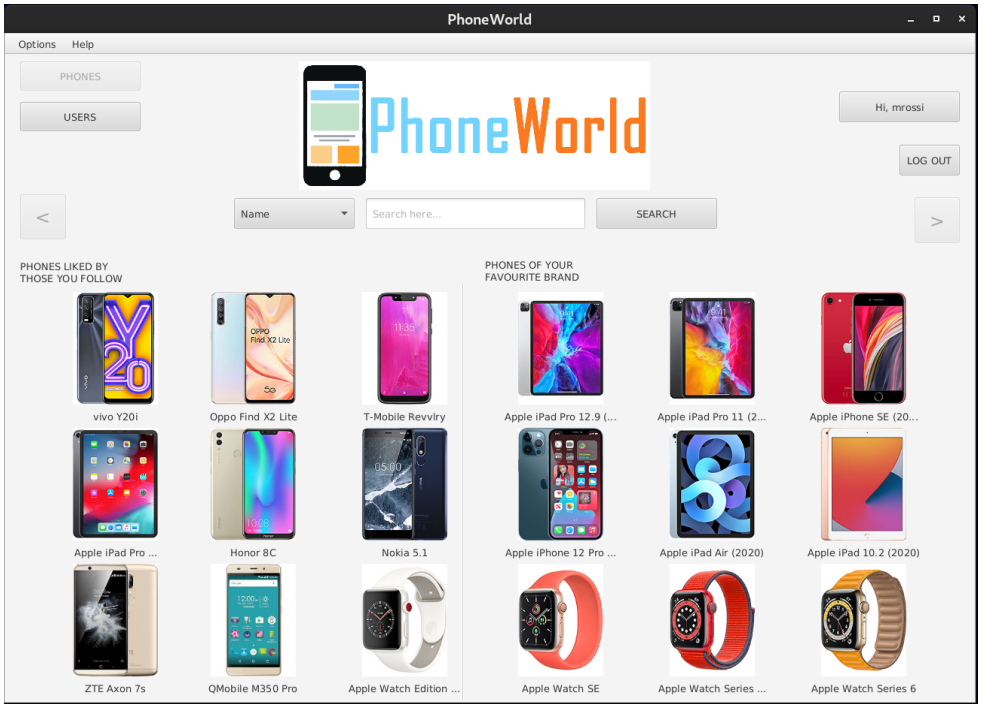
User manual

Firstly, is presented what an unregistered user can see just when he/she runs the application. We can call this the *Home Page* of the application.

For an unregistered users, are shown the latest phone:

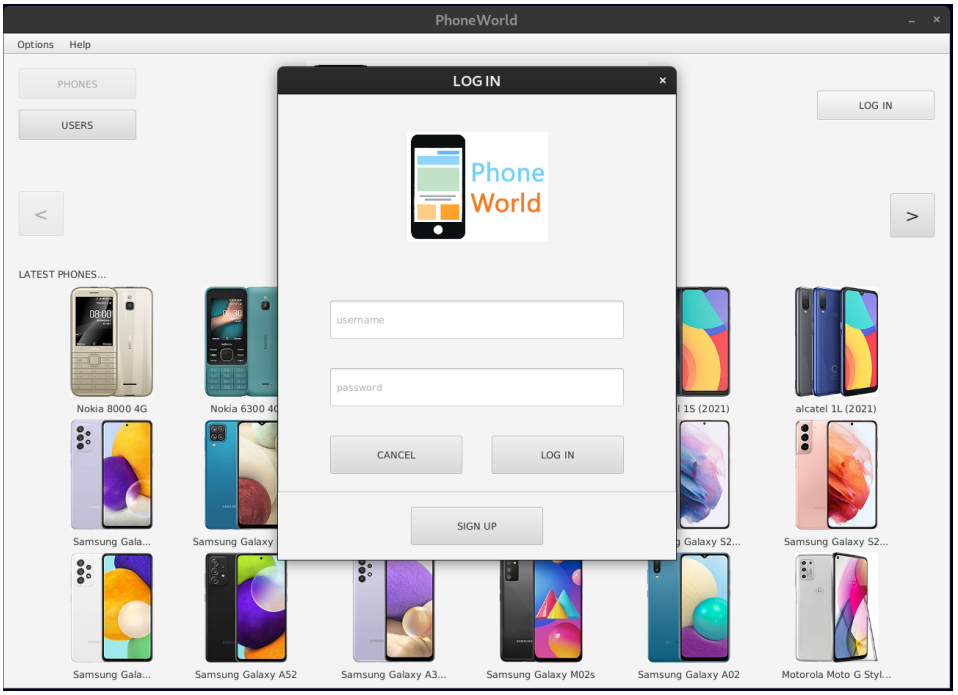


For a logged user this page shows the suggested phones:

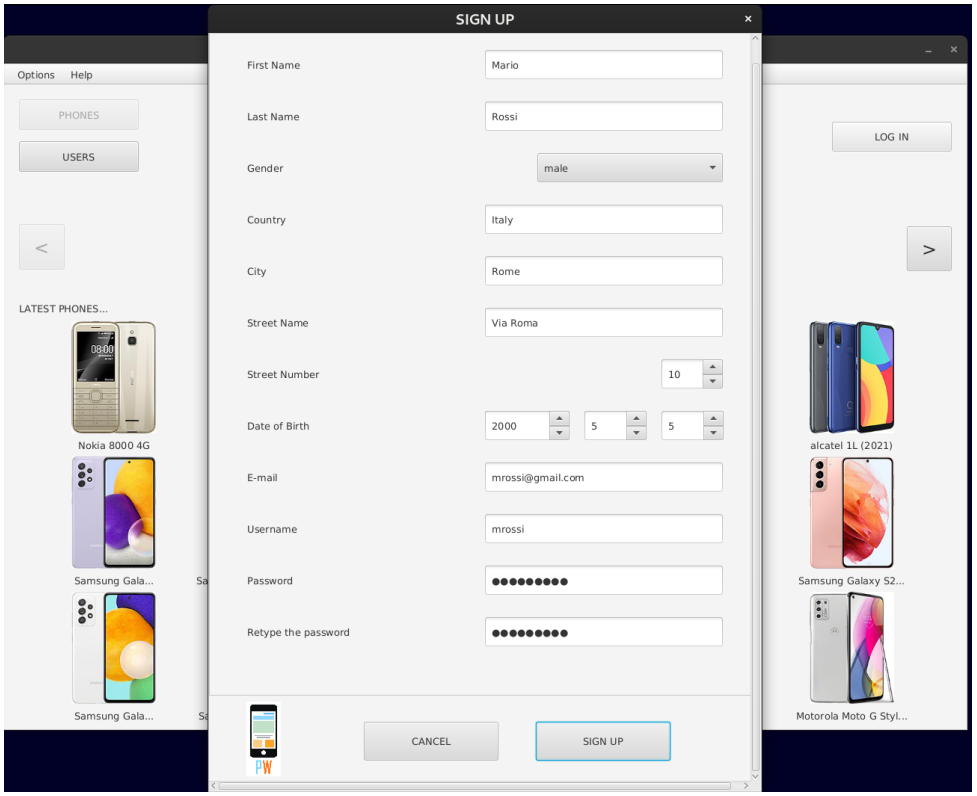


The *USERS* button redirects the unregistered user to the log in. Instead, the registered user can see all the other users as it will be shown later.

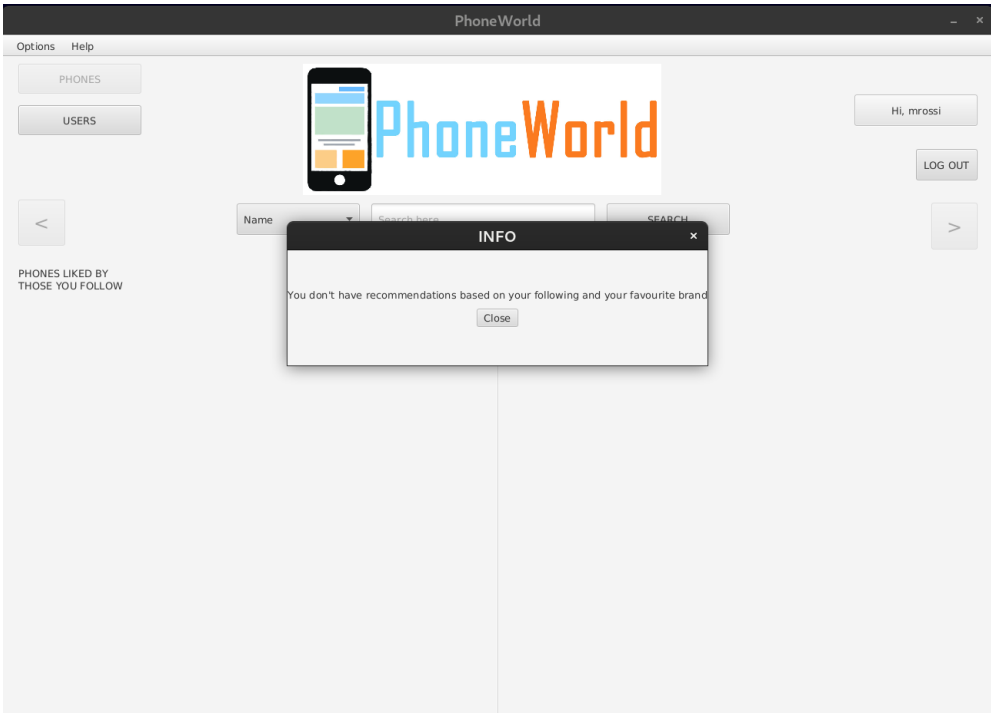
Clicking on *LOG IN*, the user can log in or sign up if not already registered, like in the window below:



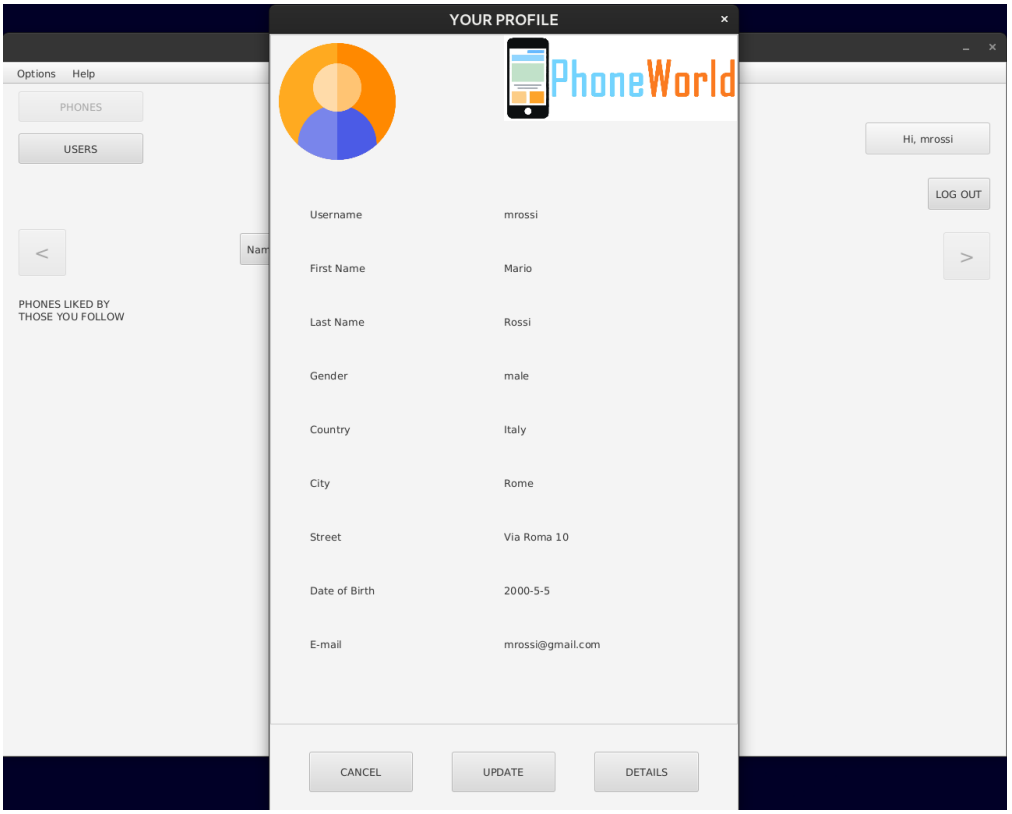
Clicking on *SIGN UP*, the user must compile all the fields to successfully complete the registration.



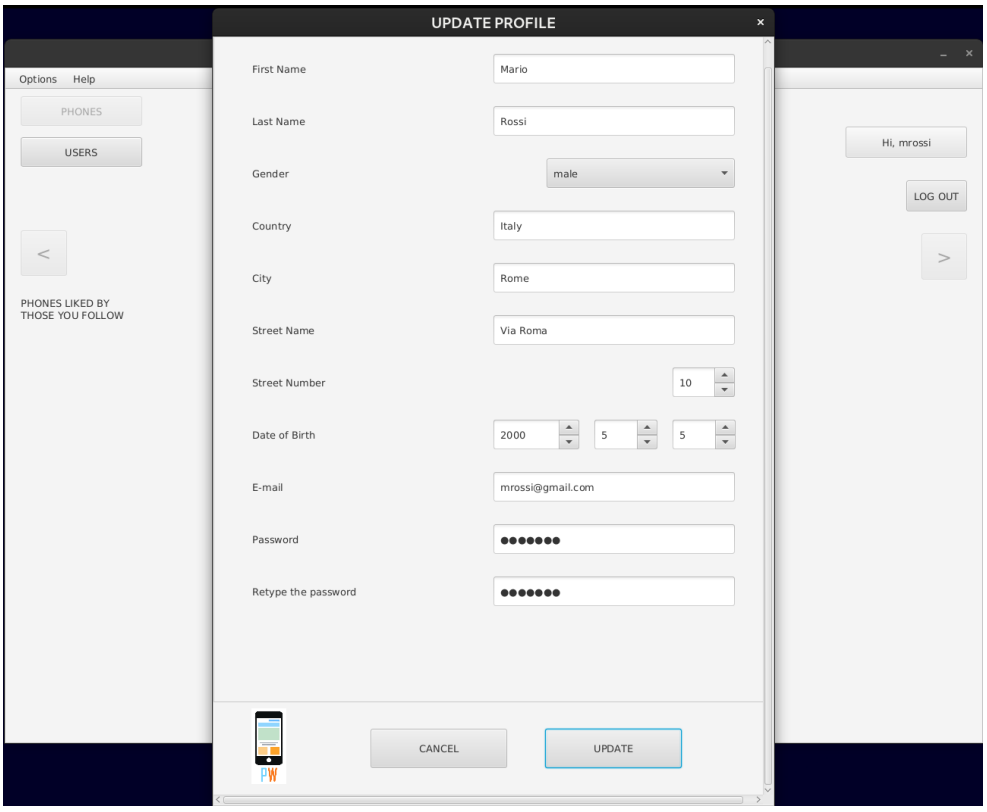
Completed the first sign up, the user can see the following:



The user profile page is:



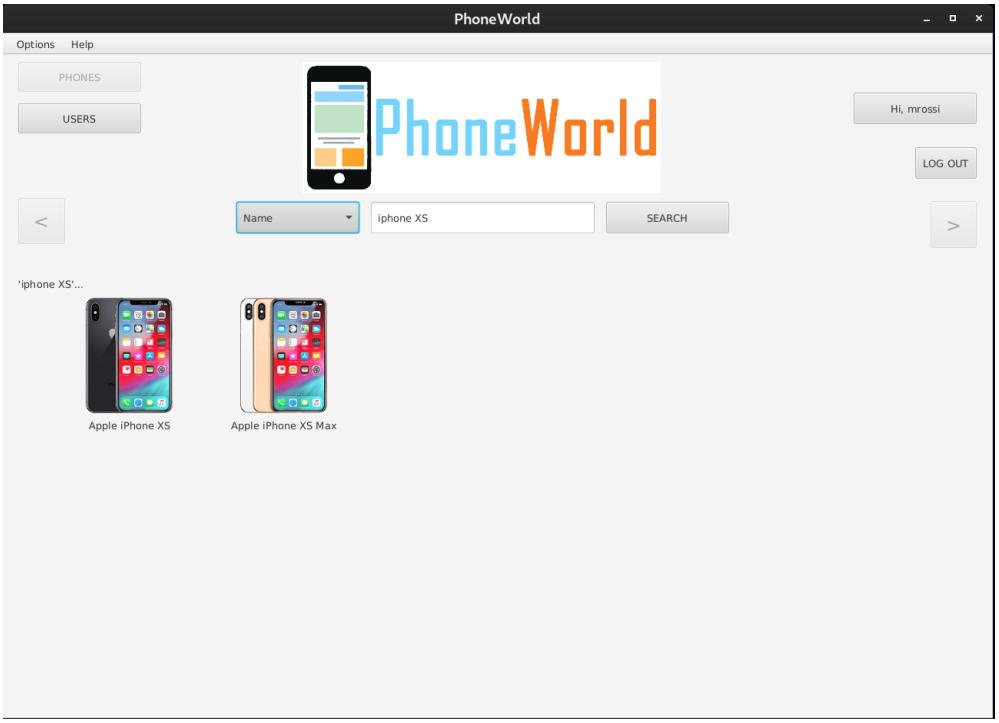
Clicking on UPDATE there's the update profile page:



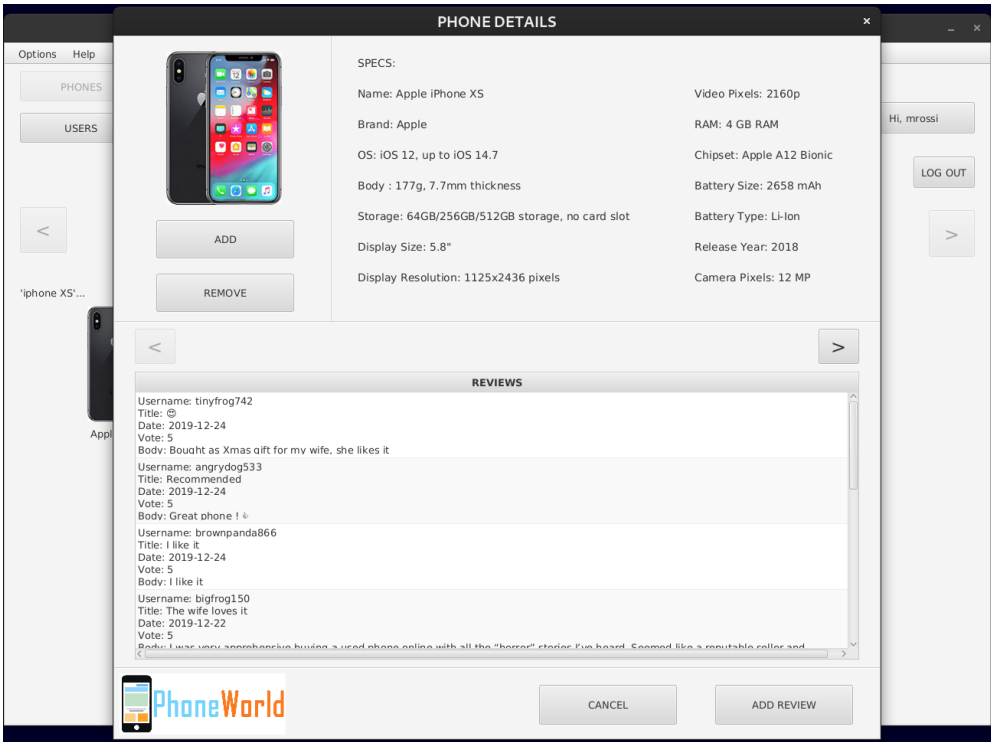
The phone search implements the following possible filters:

- Name
- Ram
- Storage
- Chipset
- Battery Size
- Camera Pixel
- Release year

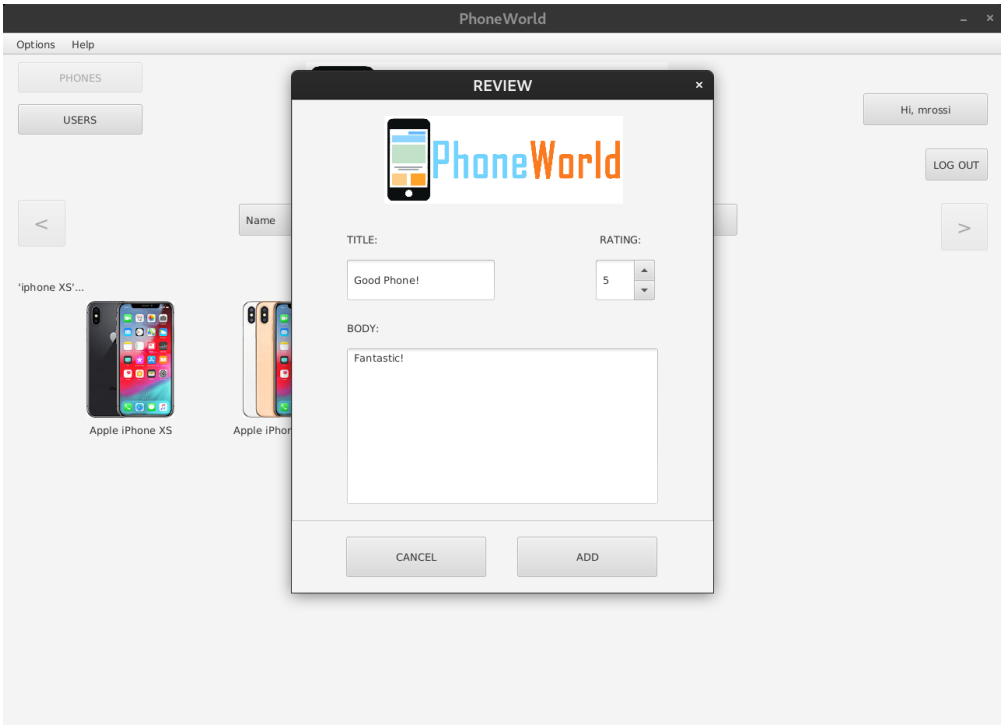
Below there's an example of research of a phone by name:



Selecting the phone, all the specifications and the reviews are showed in the phone page:



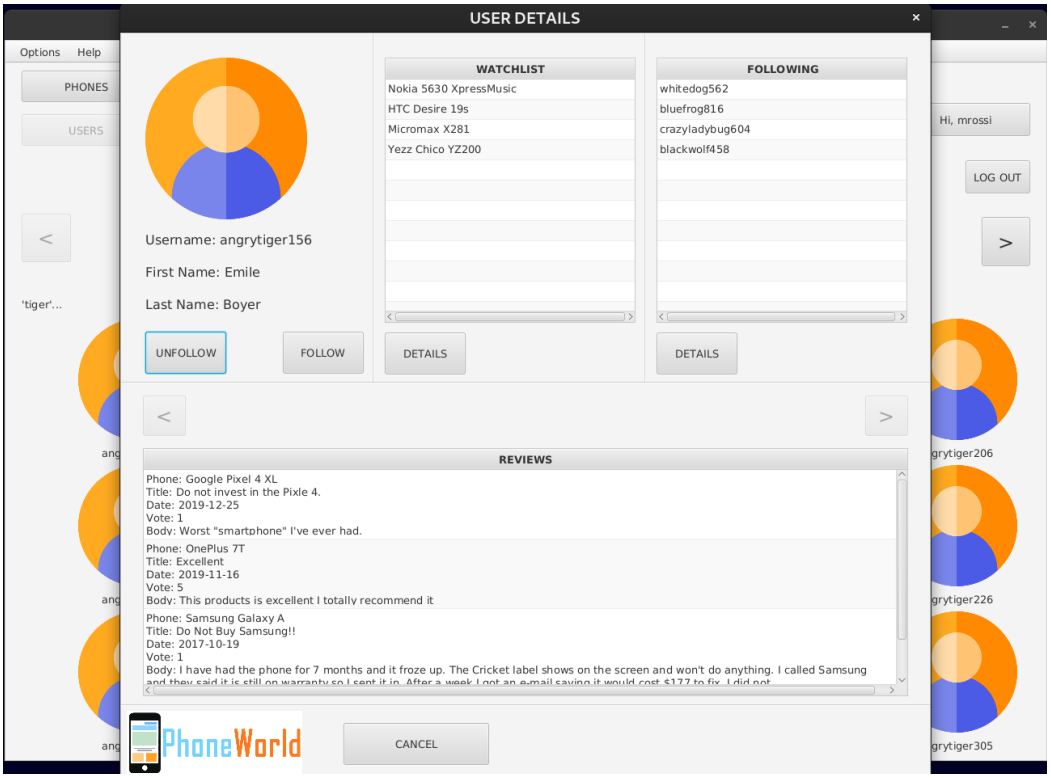
With the button ADD REVIEW it opens the following window:



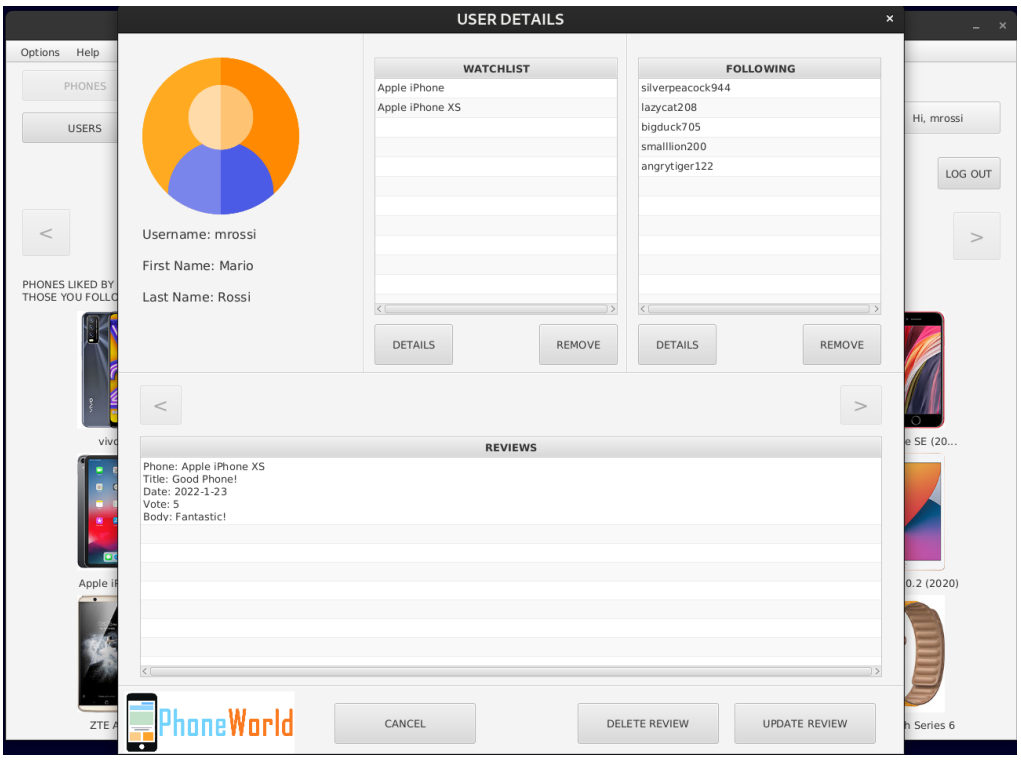
Back to the Home Page, clicking on the USERS button, the suggested users are showed (for a logged user):

Here is possible to search just by name:

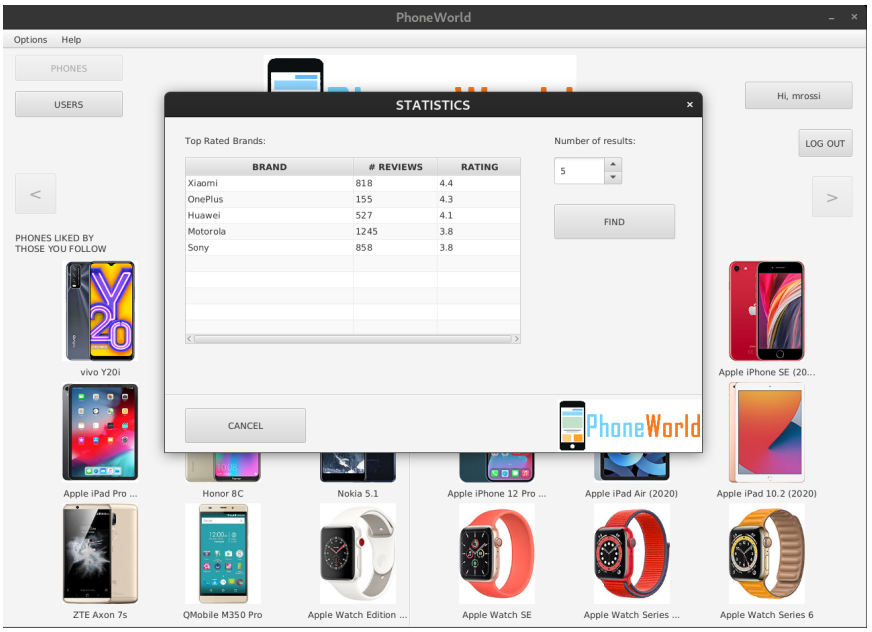
Clicking on a user, the user page opens with all the user information, the watchlist, the following users and the reviews. Is possible to follow or unfollow the user.



If the current user is on his/her personal profile, the page is different because here are present the *REMOVE* buttons for watchlist, following users and reviews. Is also possible to update a review.



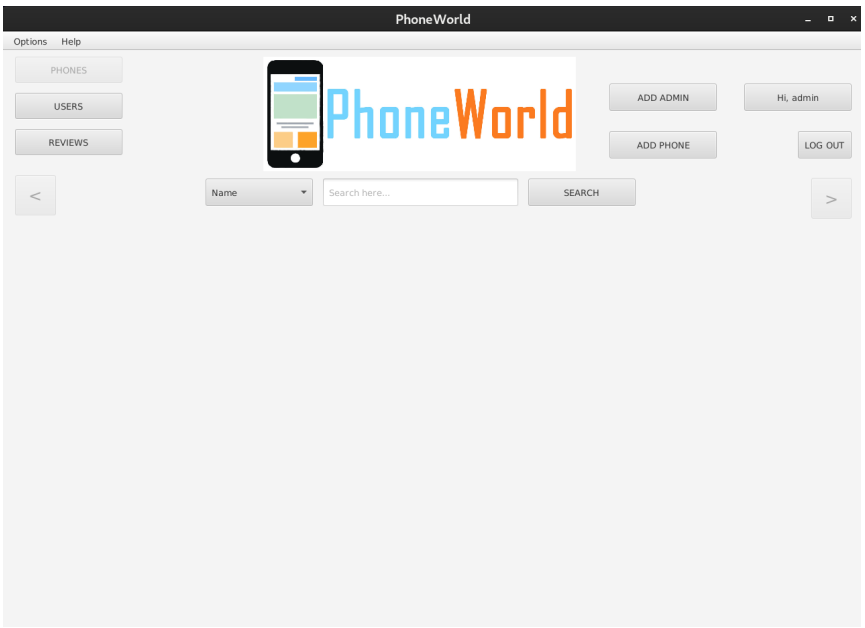
Clicking on *Options* on the left high corner, the user can access the statistics available and selecting one it opens the relative panel. Here is shown, as an example, the *Top Rated Brands* statistic.



Admin manual

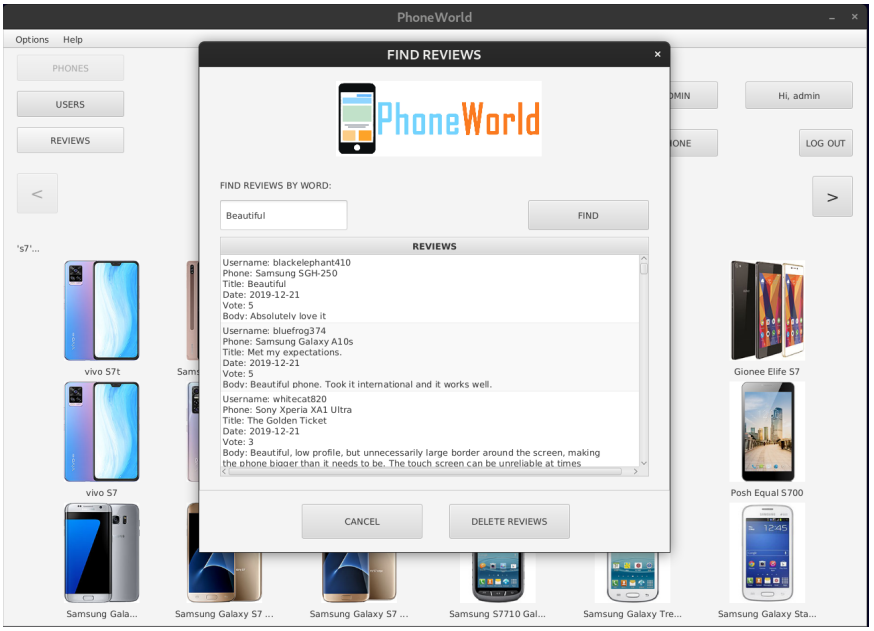
The admin windows are a bit different.

A first time log in with an admin profile shows the following window:

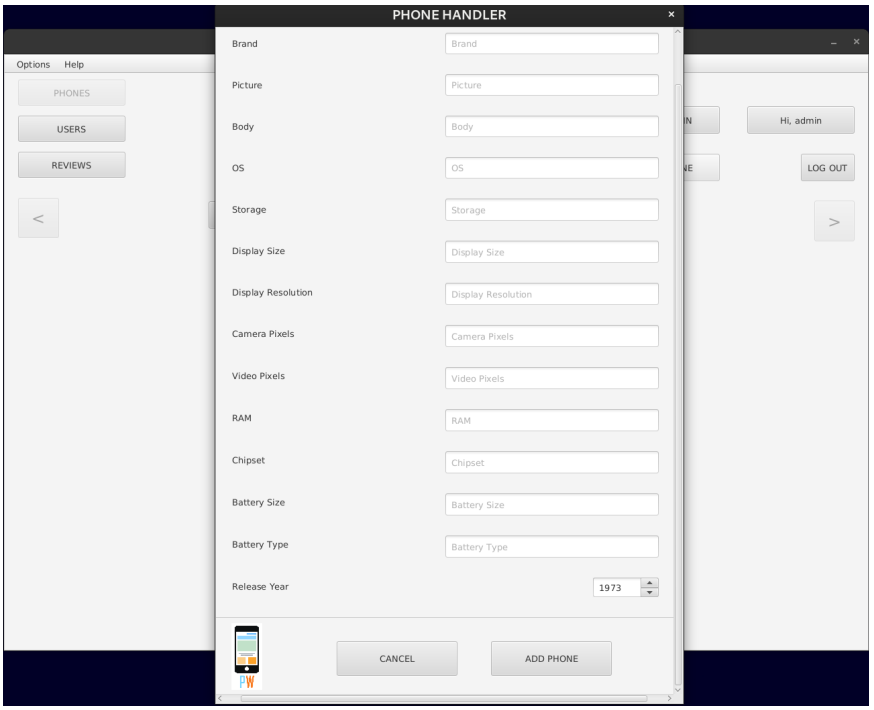


It has these special buttons:

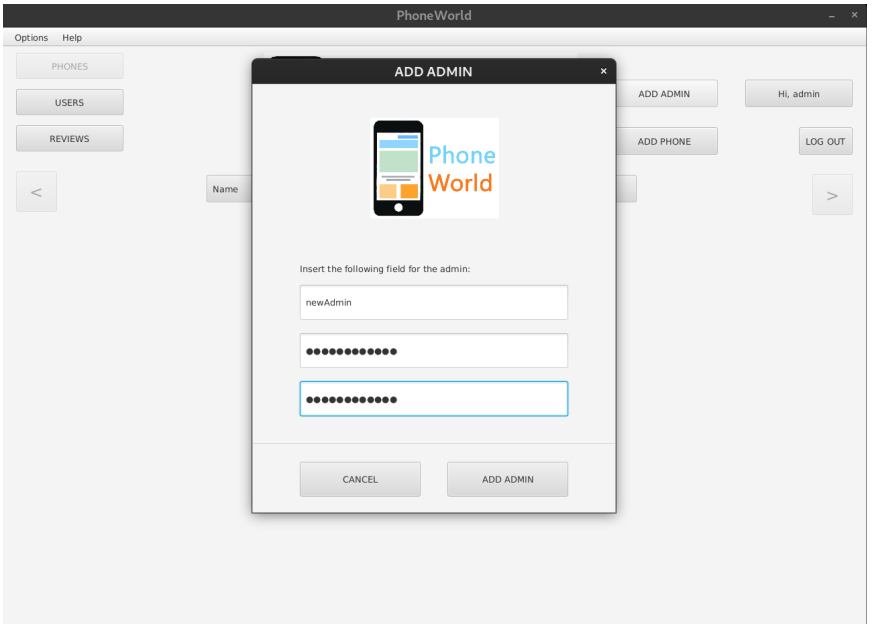
- *REVIEWS*, to search reviews by word and delete them:



- ADD PHONE, to add a phone to the database:

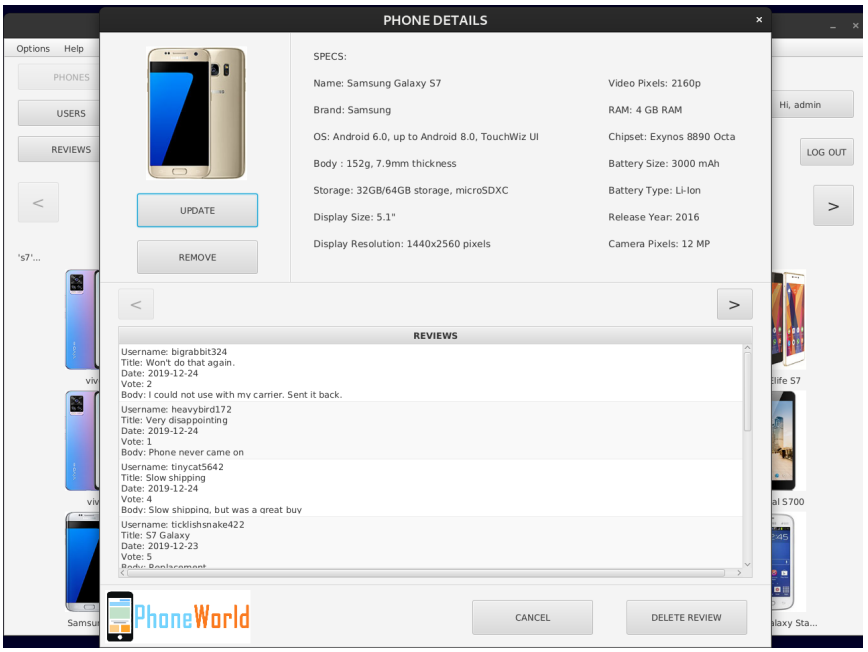


- ADD ADMIN, to add a new administrator, for who is required just an username and a password:

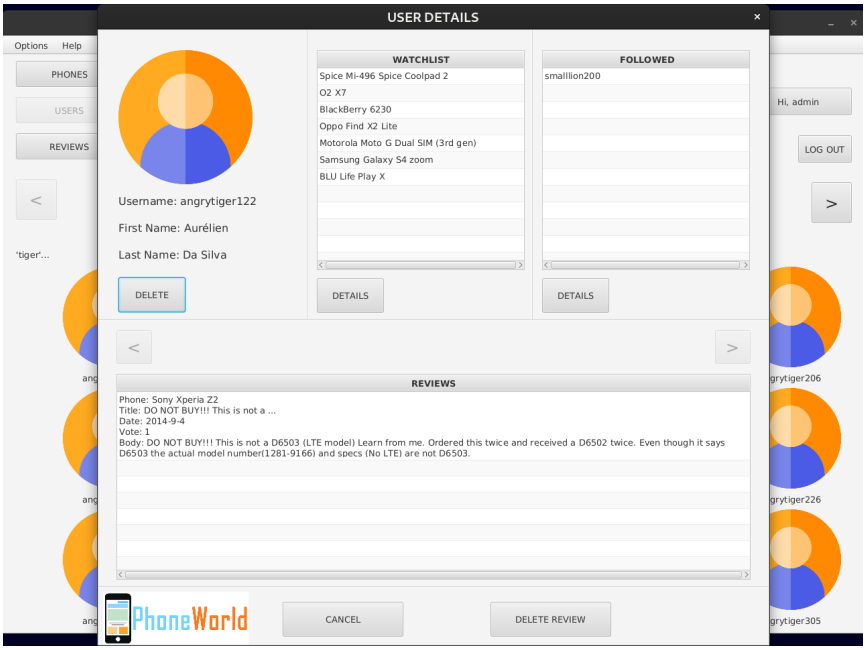


The PHONES and USERS buttons allow the search on them like a normal user, but here the admin can't have suggestions or interactions with other users.

In addition, when the admin searches a phone he/she can *REMOVE* or *UPDATE* it and delete reviews:



When the admin searches a user he/she can *DELETE* it or delete a review as it is displayed below:



Lastly, as already said for a registered user, from the *Options* button the admin can access the statistics. Here the phone and user suggestions are not present, but in addition to statistics available to users, there are three more statistics exclusively for the admin like explained in the Use Cases.