University of Pisa

School of Engineering

Master of Science in Artificial Intelligence and Data Engineering

---

Wine Winery App

Massimo Valentino Caroti
Simone Landi
Academic year 2022-2023

INDEX

INTRODUCTION

WineWinery is a Java application designed for wine lovers to share experiences and feelings about bottles.
Users can search and comment wine bottles in the application. Bottle informations come from two different sources and are easily and quickly available through the application's interface.
Each user can create their own wineries and add the bottles present in the application to them.
In addition, WineWinery has a social component that allows users to interact by going to like bottles and offering the ability to follow users and wineries.

To keep the bottle database updated, the project includes DB_update, a Python script, which can be run from the command line. This script allows you to initialize the database or update it with the latest available bottle information.

The project is available on GitHub :
Link : https://github.com/max423/Wine-Winery-App

DATASET

To develop the WineWinery application, we had to create a large database of wines and comments. Through web scraping, we created the datasets using information found on the Internet from two sources for wine bottles and one source for comments.

Users, social relationships between users, and user relationships with bottles and wineries were randomly generated.

- https://www.vivino.com
- https://www.glugulp.com
- https://randomuser.me

	Data Size
Wine	60 MB
Comment	26 MB
User	1 MB

WINE

To generate the bottle dataset, an integration was performed from the data obtained by web scraping on two websites: the first one Vivino to obtain most of the bottles and the second one on Glugulp to obtain the information on champagne bottles.

The two datasets were then integrated together by going to remove any duplicate values from the two sources.

Wines info obtained by performing data scraping :

- Vivino: vivino_id, name, winemaker, country, varietal, grapes, year, price, info, description
- Glugulp: glugulp_id, name, winemaker, country, varietal, grapes, year, price, info, description

COMMENT

For each bottle found on Vivino, another request was made to the site to obtain comments under the bottle page. From this the text of the comments, date, and username were extracted.
This step was done to obtain as much real information as possible for greater consistency of the application.

USER AND SOCIAL NETWORK RELATION

From the extracted comments, user usernames were obtained and we filled the personal information of each profile by making a request to randomuser.me. By doing this, we kept the original relation user-comment from the Vivino site.
We also populated the social network part of the application by adding Likes and Follows relations between users, wine and winery.

DESIGN

MAIN ACTORS

The main actors of the application are:

- **Unregistered user**

User that accesses to the application for the first time. They have to sign up to become a registered user

- **Registered user**

User that is already registered to the application. They can use the application after the login.

- **Moderator**

User that can delete comment of Bad user.

- **Administrator**

They are the most powerful actor of the application since they can delete users and comments in order to avoid offensive content and toxic behavior.

- **DB_updater**

It is responsible for initializing and keeping the database up to date.

FUNCTIONAL REQUIREMENTS

Features offered to the **unregistered user**:

- Sign in, to have access to the community, a user must be registered.

Features offered to the **registered user**:

- Login/Logout
- Update profile informations
- Search wine by parameters (winemaker, country, varietal, grapes, minimum year, maximum year, minimum price, maximum price) or by name
- Search winery by name
- Search user by username
- Manage their wineries
  - Create a new winery
  - Delete a winery
  - Add or Remove wine from a winery
- Comment Wine
- Update / delete comment
- Like / Remove Like from Wine page
- Browse follow / following Users
- Browse Suggestion
  - Suggested wines
  - Suggested users
  - Suggested wineries
- Browse Analytics
  - Most commented wines
  - Most liked wines
  - Most followed users
  - Most followed wineries
  - Most versatile users

Features offered to the **moderator**:

- Moderator can do all the operations that a Registered user can do
- Remove comments from a wine
- Browse Suggestion
  - Suggested wines
  - Suggested users
  - Suggested wineries
- Browse Analytics
  - Most commented wines
  - Most liked wines
  - Most followed users
  - Most followed wineries
  - Most versatile users

Features offered to the **admin:**

- Browse Analytics
  - Most commented wines
  - Most liked wines
  - Most followed users
  - Most followed wineries
  - Most versatile users
- Browse Summary
  - Wine varietals sorted by comments
  - Wine varietals sorted by number of wines
  - Wine varietals sorted by number of likes
- Remove comments from a wine
- Search Bad user
- Search user by username
- Elect/dismiss User as Moderator

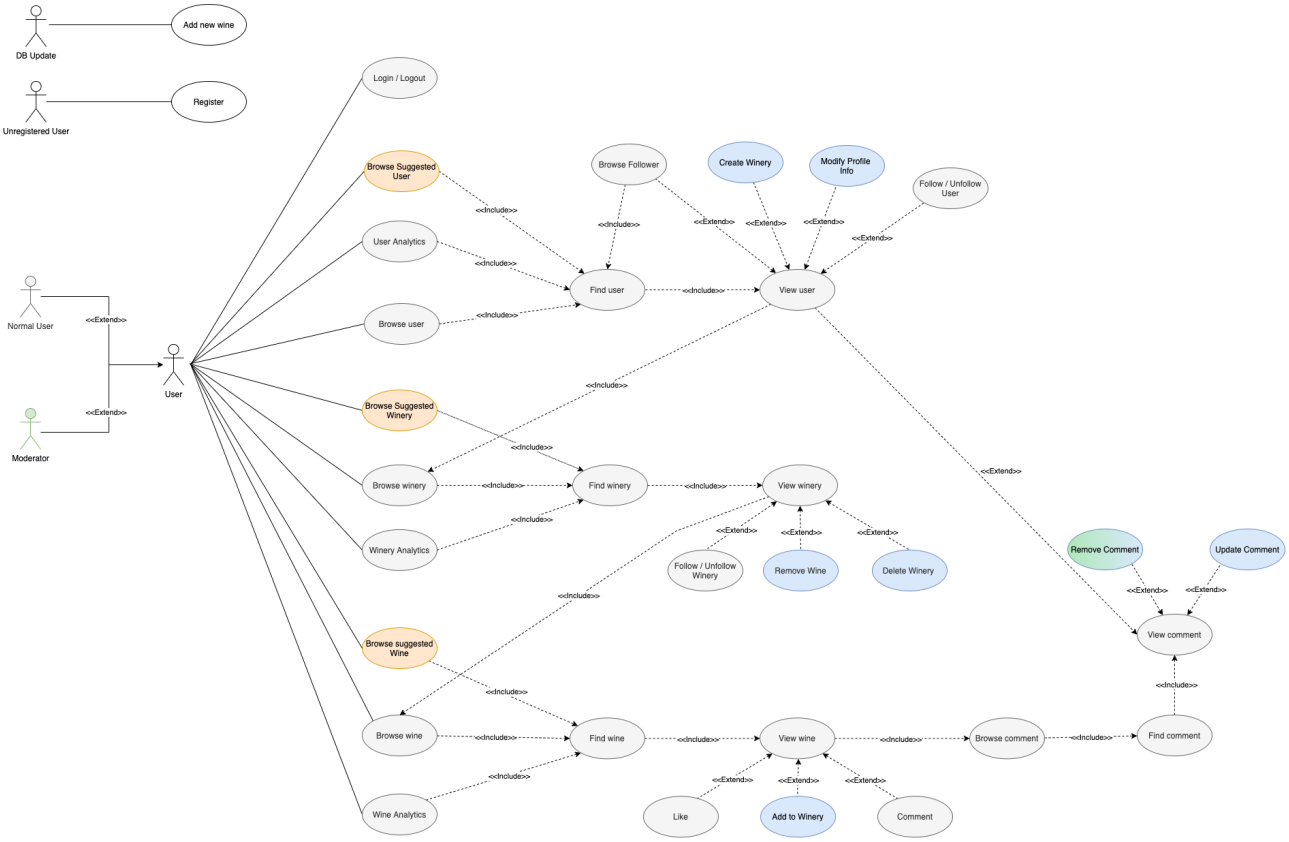
- Remove Users

NON-FUNCTIONAL REQUIREMENTS

- The system needs to be tolerant to data lost and to single point of failure.
- The service must be always available for users in order to guarantee the best possible user experience, providing a response to any query.
- The user credentials must be handled securely.
- Flexibility is needed for manage different sources of data.
- The application must be user-friendly and must ensure a low response-time.

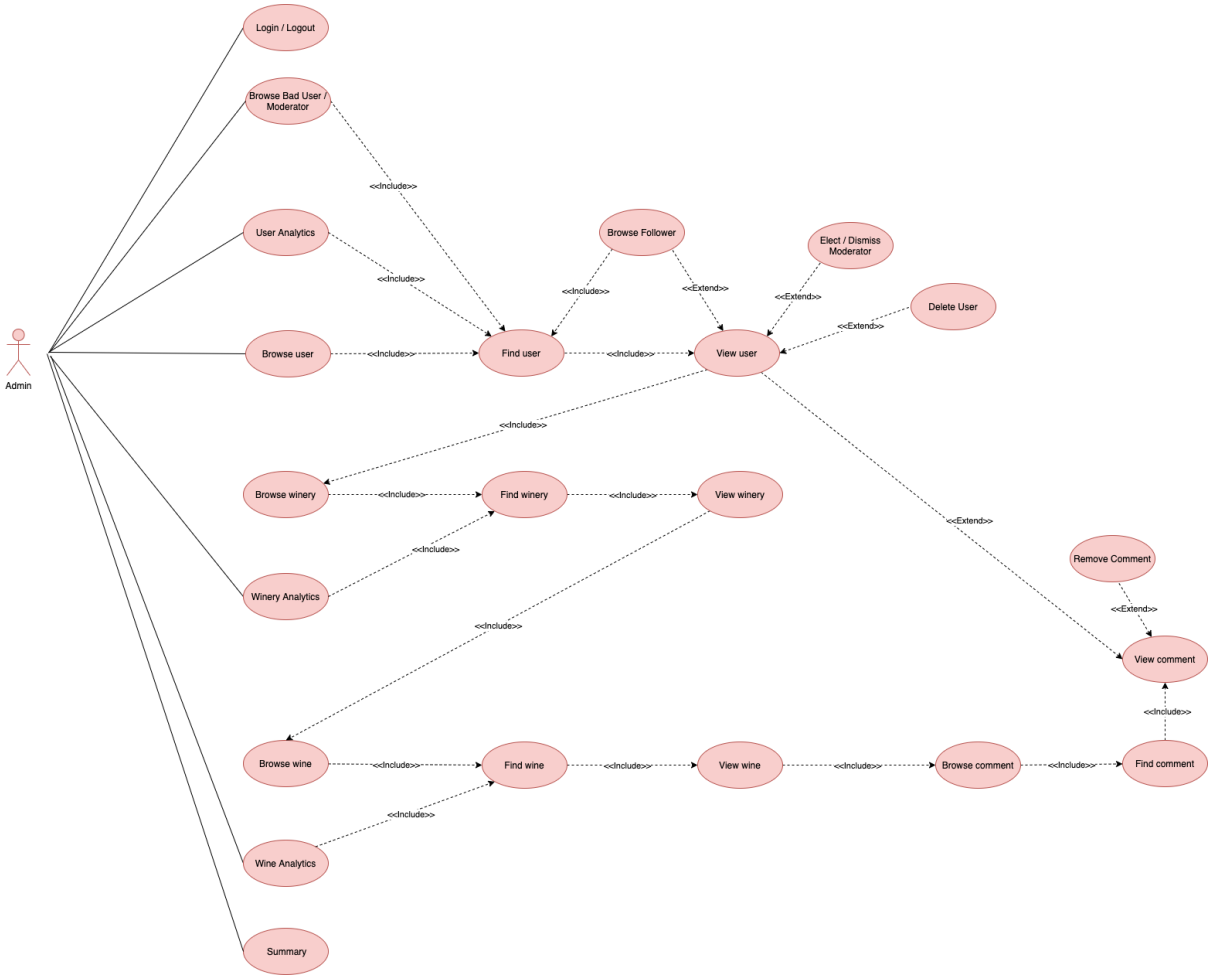
USE CASE DIAGRAM

The use case diagram of the application is described in Figure



- The circles in grey describe actions available to normal users and moderators
- The circles in green describe actions available to moderators
- The circles in blue describe actions available only to the owner of the entity
- The circles in orange describe graph database typical actions

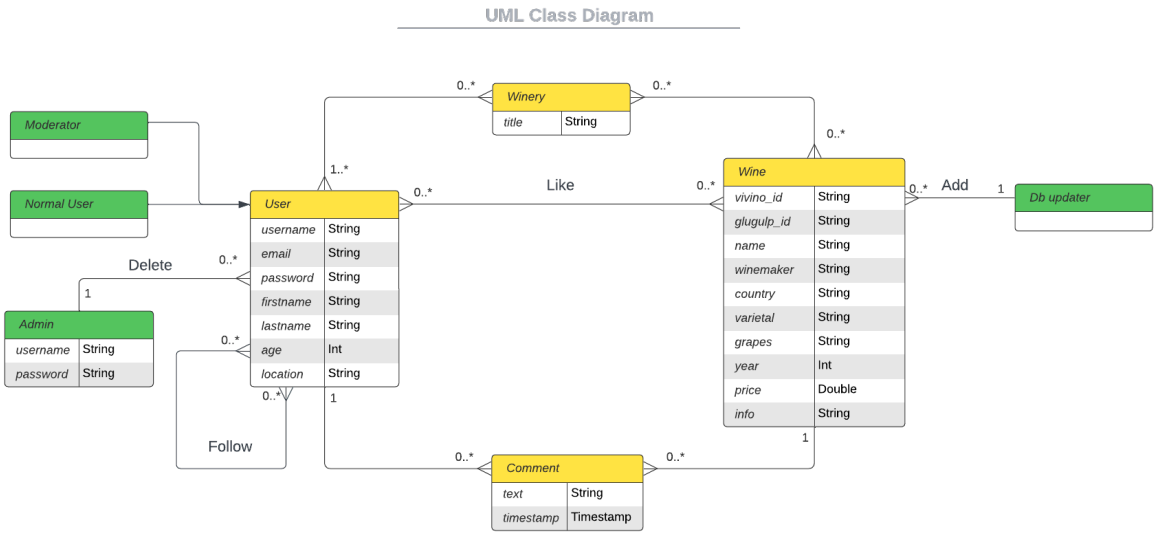
The following diagram describe the actions available to the admin.



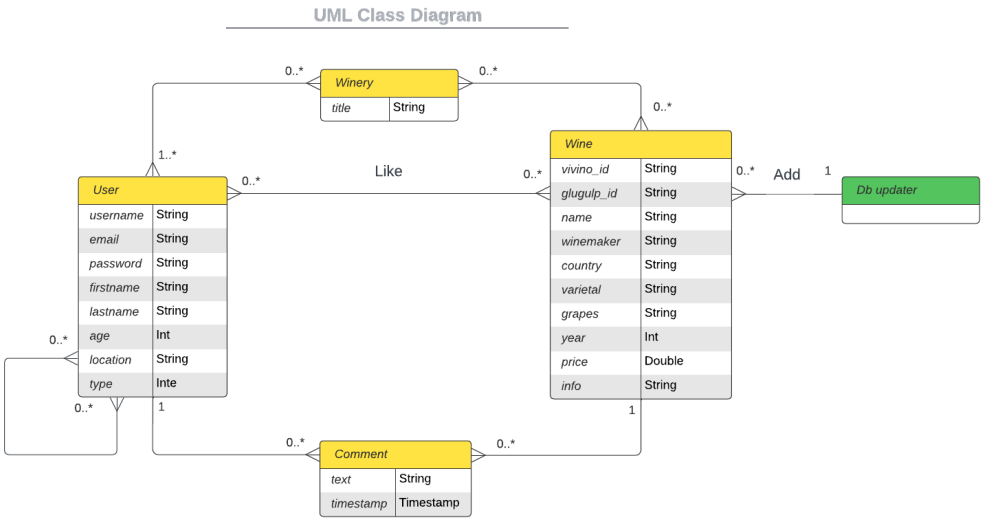
UML CLASS DIAGRAM

The class diagram of the application is described in Figure.

In the diagram, User is a generalization of the two main actors (Registered User and Moderator) of the use case diagram, each actor can perform in addition to its own action, the same action of User.



The restructured version of the class diagram of the application is shown in Figure. We resolve the generalization adding one attribute to the class user for specify the role. Since we have many attributes in common between User and Admin, we decided to merge the two classes while still being able to distinguish the various functionalities through the type attribute.



RELATIONSHIPS BETWEEN CLASSES

Depending on type attribute of the User, the user can be considered a Normal User , a Moderator or an Admin. The actions they can perform are different and are reported here below.

- Normal User
  - A User can create zero or more Wineries. An additional constraint is that a User can’t create more than 10 Wineries.
  - A Winery can be created only by one User/Moderator.
  - A Winery can contain zero or more Wines.
  - A Wine can be contained in zero or more Wineries.
  - A User can write zero or more Comments. An additional constraint is that a Comment can’t be longer than 100 characters.
  - A Comment can be written only by one User/Moderator.
  - A Comment can only refer to one Wine.
  - A Wine can have zero or more Comments.
  - A User can add like to zero or more Wines.
  - A Wine can have zero or more likes from Users/Moderators.
  - A User can follow zero or more Users/Moderators.
  - A User can be followed by zero or more Users/Moderators.
- Moderator
  - A Moderator can create zero or more Wineries.
  - A Winery can be created only by one User/Moderator.
  - A Moderator can write zero or more Comments.
  - A Comment can be written only by one User/Moderator.
  - A Moderator can add like to zero or more Wines.
  - A Wine can have zero or more likes from Users/Moderators.
  - A Moderator can follow zero or more Users/Moderators.
  - A Moderator can be followed by zero or more Users/Moderators.
  - A Moderator can delete zero or more Comments.

- A Comment can be deleted by one Moderator/Admin.

- Admin
  - An Admin can delete zero or more Comments.
  - A Comment can be deleted by one Moderator/Admin.
  - An Admin can delete zero or more Users/Moderators.
  - A User/Moderator can be deleted by one Admin.
  - An Admin can elect zero or more Users as Moderator.
  - A User can be elected as Moderator only by one Admin.

DESIGN OF THE CLASSES

- User's attributes
  - Username: contains the username of the user
  - Email: contains the email of the user
  - Password: contains the password of the user
  - Firstname: contains the first name of the user
  - Lastname: contains the last name of the user
  - Age: contains the age of the user
  - Location: contains the country of the user
  - Type: contains the role of the user (0: User, 1: Moderator, 2: Administrator)
- Winery's attributes
  - Title: contains the name of the winery
- Comment's attributes
  - Text: contains the text of the comment
  - Timestamp: contains the creation timestamp of the comment
- Wine's attributes
  - vivino_id: contains the id of vivino bottle (if the entity is a wine of Vivino)
  - glugulp_id: contains the id of glugulp bottle (if the entity is a wine of glugulp)
  - Name: contains the name of the wine

- Winemaker: contains the name of the wine producers
- Country: contains the country of the wine
- Varietal: contains the category of the wine
- Grapes: contains the grapes of the wine
- Year: contains the year of production of the wine
- Price: contains the price of the wine
- Info: contains info about the wine
- Description: contains the description of the wine

DATA MODEL

DOCUMENT DB

The document database was chosen because WineWinery needs to efficiently handle large amounts of data with flexibility. The use of a document database allows us to accommodate missing fields without wasting memory on null values. In our application, we utilize different fields to identify the source of a wine, such as the "vivino_id" for wines from vivino.com and "glugulp_id" for wines from glugulp.com. With a document database, we don't need to set unused values to null. The schema less approach offered by document database also allows us to handle in a flexible way users without wineries, wineries without wines or wines without comments.

We decided to implant three collections: Users, Wines and Comments.

Example of json document representing a **User**


```
{
    "_id":{"$oid":"64ff540d0d9fccb79cdf5456"},
    "username":"lapo.baglini",
    "email":"jayne.middelburg@example.com",
    "password":"RH9Ru1veSiPJ2LL/gXRtow==",
    "firstname":"Jayne",
    "lastname":"Middelburg",
    "age":73,
    "location":"Russia",
    "type":0,
    "winerys":[
        {
            "title":"winery0",
            "wines":
                [{"vivino_id":"170480226.0",
                "name":"Cantina Tollo 'Peco' Pecorino Terre di Chieti 2022",
                "winemaker":"Cantina Tollo",
                "varietal":"Bianco"}]
        },
        {
            "title":"winery1",
            "wines":
                [{"vivino_id":"152459152.0",
                "name":"Coppo Chardonnay Piemonte Riserva della Famiglia 2017",
                "winemaker":"Coppo",
                "varietal":"Bianco"},
                {"vivino_id":"15069771.0",
                "name":"Luigi Einaudi Dogliani 1976",
                "winemaker":"Luigi Einaudi",
                "varietal":"Rosso"}
                ]
        }
    ]
}
```

In terms of performance, employing a document database enables us to incorporate objects that are frequently accessed together, reducing the need for expensive join operations. We made the decision to include wineries within the user's document, as all the wineries created by a user are displayed on their user page. Each winery document only contains a subset of information necessary for previewing, while complete information about wine can be retrieved from the corresponding wine document. It's worth noting that the maximum number of wineries a user can create is limited to ten.

Example of json document representing a **Wine**


```
{
    "_id":{"$oid":"64ff52e40d9fccb79cdec8e2"},
    "vivino_id":"18480339.0",
    "name":"Tenute Al Bano Carrisi Negroamaro 2015",
    "winemaker":"Tenute Al Bano Carrisi",
    "country":"Italia",
    "varietal":"Rosso",
    "grapes":"Primitivo, Aglianico, Negroamaro, Nero d'Avola",
    "year":2015,
    "price":1.47,
    "info":"The soil in Southern Italy is largely volcanic and also granitic.\nSouthern Italy...",
    "comments":[
        {"username":"pietro-rizzo1",
        "text":"Rosso rubino. Naso floreale. Palato con nuances erbacea. Sorso di media ...",
        "timestamp":"2022-04-06 14:42:05"},
        {"username":"morganste1",
        "text":"Strutturato pieno polposo succoso persistente ..:: sentori tostati fruttati...",
        "timestamp":"2021-08-29 20:04:24"},
        {"username":"andrea.viglianti",
        "text":"Un vino pugliese , sarà firmato carrisi , ma non mi è piaciuto per nulla .",
        "timestamp":"2020-11-13 15:22:52"},
        {"username":"emiliano-pia",
        "text":"Mi ha deluso la corposità,  mi aspettavo quella tipica dei negramari...",
        "timestamp":"2017-12-08 14:35:52"}
        ]
    }
```


We applied the embedding of the comments in an array in the document of the corresponding Wine. The embedding of the comments makes faster the information retrieval. Embedding comments makes information retrieval faster. because when we view a bottle page, the most recent comments are also shown.

Since we opted to include comments within the wine document, we needed a solution for handling situations where a wine document becomes excessively

large due to a high number of embedded comments. To address this issue, we introduced a threshold to limit the number of comments stored within a wine document. If this threshold is reached and a new comment is written, a dedicated function of our application comes into play, which removes the oldest comment to make space for the new one. In particular, we have chosen to maintain a maximum of ten embedded comments.

It's important to note that all comments will still be available in the Comments collection. This approach serves two purposes: it allows users to access all comments related to the wine they are viewing by selecting the "show all comments" option, and it enables us to perform analytics and generate summaries based on the complete set of comments.

Example of json document representing a **Comment**

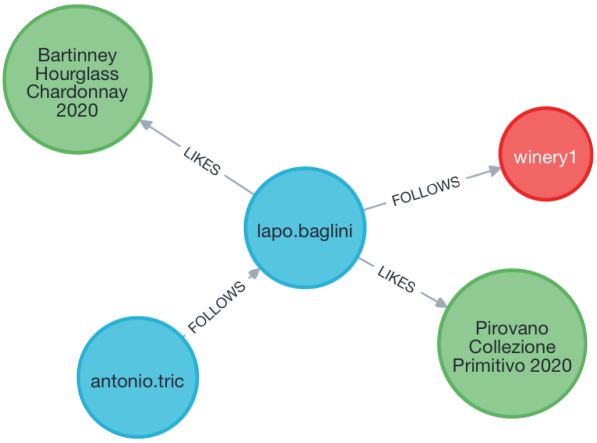

```json
{
    "_id":{"$oid":"64ff52e30d9fccb79cdda301"},
    "username":"andrea_oggioni",
    "vivino_id":"161471613.0",
    "varietal":"Rosso",
    "timestamp":"2023-09-05 17:31:34",
    "text":"Un rosso elegante di Masi, un supervenetian, ..."
}
```


As explained earlier we also implemented a collection related to comments in order to save the information related to them. Within this collection we also chose to save the varietal information so that we would have the ability to perform summaries more efficiently. This was done primarily for the main reason that the varietal information related to a bottle of wine cannot vary, just as a bottle of wine cannot be deleted.

GRAPH DB

We've implemented a graph database to efficiently execute rapid and insightful queries that involve traversing various relationships among users, users and wines, or users and wineries. This approach helps us avoid the need for multiple join operations, which could be computationally intensive and wouldn't meet our stringent low latency requirements. Our choice was to keep the graph database as “light” as possible and to use it only for handling social network relationships and social network-based analytics.

Example of graph



The GraphDB nodes are the following:

- User: username, email
- Wine: vivino_id/glugulp_id, name, winemaker, varietal
- Winery: title, owner

The GraphDB relationships are the following:

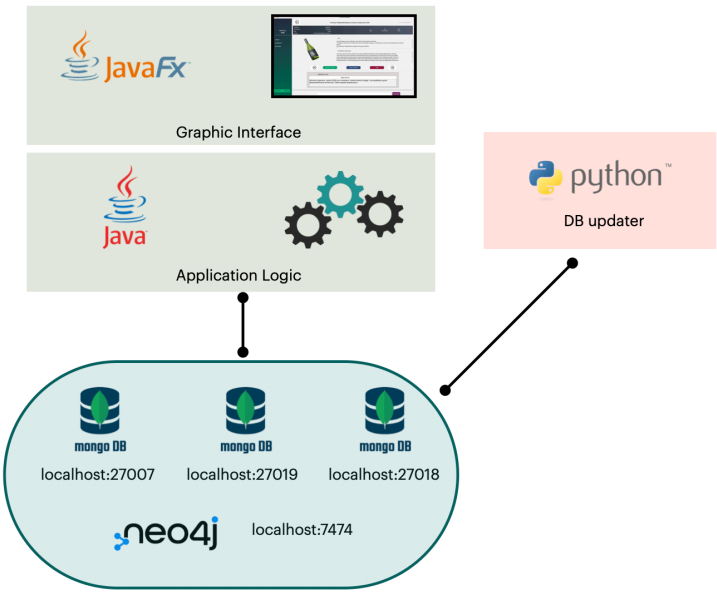
- Follows
    - If a user follows another user (:User)-[:FOLLOWS]->(:User)
    - If a user follows a winery (:User)-[:FOLLOWS]->(:Winery)
- Likes
    - If a user likes a wine (:User)-[:LIKES]->(:Wine)

We decided to duplicate certain essential information in both databases. Specifically, we stored data required to generate previews for various entities in the graph database. This strategic decision allows us to showcase the results of our social network analytics within the application efficiently, without the necessity of accessing MongoDB for this purpose.
By having this duplicated data in the graph database, we can easily retrieve and present these previews, enhancing the user experience and avoiding potential performance bottlenecks associated with cross-database queries.

ARCHITECTURAL DESIGN

SOFTWARE ARCHITECTURE

The application was implemented as client-server architecture, with middleware implemented on the client side.



Client Side

The client is composed of this modules:

- Front-end: this part consists of the GUI, allowing the users to have an intuitive interaction with the application of a graphical interface, developed in JavaFX and FXML files. User can use the GUI to interact with the application and managed by controllers used to handle events and views of the informations.

- Middleware: the duty of this part is to handle the connections and communications with the MongoDB and Neo4j.

Server Side

Unfortunately, for server availability reasons, we deployed the databases on the local host. Consists of three instances of mongoDB and one instance of Neo4j

Db_updater Side

The Wine Updater is an command line application composed by several python scripts:

- db_updater.py
  - command line Python application used to do create and update
- glugulp_scaper.py:
  - Gets glugulp links: glugulp_link.csv
  - Gets glugulp bottles: glugulp.json
- vivino_scraper.py:
  - Gets vivino bottles: vivino.json
- wine_generator.py:
  - Merges the two wine databases by checking duplicate: wines.json
- user_generator.py:
  - Gets 5000 users with multiple comments and generates info: users.json
- comment_scraper.py
  - Gets vivino comments: comment.json
  - Gets glugulp comments: glugulp_comment.json
  - Merges comments: filtered_comment.json

The DB_updater.py accept this action:

A. initDB: initializes the database of the application by populates the database with wines, users, wineries, comments, likes and follows.
B. UpdateDB: downloads the latest wine and it uploads them in the database. This command use a Bloom Filter to speed up the process avoiding the access on db to check if a bottle is already present in the current database.
C. Exit: close the command line application

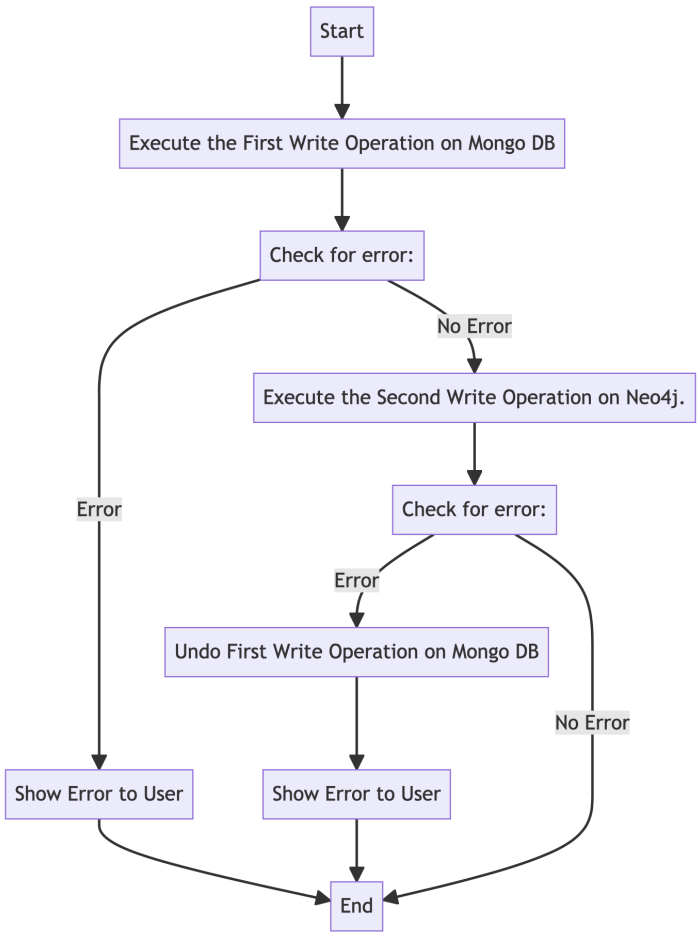
HANDLING INTER-DATABASE CONSISTENCY

We need to avoid possible inconsistencies between the databases when queries are performed on the same information that is on both databases.

The operation succeeds only if both operations on the two databases are successfully completed; if either operation fails, we un-do the operation on the first databases and show an error to the user.

The operations that require managing consistency between databases are Add/ Remove/Update a user Add/Remove a winery.

For updating users, we need to maintain consistency only on the "email" field, because it is the only information that can be updated in the Neo4J entity.



We also need to maintain consistency in inserting/updating/ and deleting comments since having a redundancy on these if they are among the last ten comments. We manage these using the same mechanism but doing both operations on mongoDB.

DISTRIBUTED DATABASE DESIGN

According to the Non-Functional Requirements, our system must provide high availability, fast response times and to be tolerant to data lost and single point of failure.
So the application has been designed in order to prefer the Availability (A) and Partition Protection (P) vertices of the CAP triangle.
For this reason, the *Eventual Consistency* paradigm has been adopted.

To achieve fast response times and availability our architecture when a user performs a write, the response time is very low since this is completed only on one replica, in a second time then the new information will also be transmitted to the other replicas. In this way writes operation don't keep the server busy for too much time.
Partition Protection of the service is guarantee by the presence of replicas in our cluster, if one server is down, we can continue to offer our service by searching the content in the replicas. Unfortunately, we can't ensure that the user will retrieve the most updated content.

- Write concern Type = W1 so wait for acknowledgement from a single member
- Read Preferences = Nearest, in such a way that read operations are performed on the nearest node, considering responsiveness, which is measured in pings.

When the wine update process runs, we want to guarantee that all the replicas will have the same information about new wine, so only in this case we need a consistency constraint, and the write operations will take more time.
On the localhost we implemented three replicas for mongoDB and one instance of Neo4j, although in practice for our application it was necessary to distribute the various instances on different servers to ensure non-functional requirement.

Sharding

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

In mongo DB we have three collections, User, Wine, Comment. Our idea is to implement Hash partition method so if the number of nodes changes it is easier to reallocate data between servers.
For the User collection we chose the username field as shard key, and for the Wine collection we chose the _id field which is automatically generated by MongoDB, as sharding key.
Regarding comments, we thought that to speed up the action of displaying all the comments for a wine when you are on that wine's page, it is convenient to shard comments associated with a wine into the same shard of that wine.

DATABASE DESIGN AND IMPLEMENTATION

MONGO DB

Within our MongoDB database, there are three collections: Users, Wines and Comments. For each document in these collections, we've made the decision to include all the necessary data required for building informational pages. At times, we also utilize MongoDB to display advanced information obtained through analytic queries.

CRUD Operation

Operation	Implementation
Create User	db.Users.InsertOne({ username: “username”, email: “email”, password: “password”, firstname: “firstname”, lastname: “lastname”, age: “age”, location: “location” })
Create Comment	db.Comments.inserOne({ text: “text”, username: “username”, timestamp: “timestamp”, vivino_id: “vivino_id” })
Get User	db.Users.findOne({ username: “username” })
Get Comment	db.Comments.findOne({ username: “username”, timestamp: “timestamp”, vivino_id: “vivino_id”, glugulp_id: “glugulp_id” })
Get Wine	db.Wines.findOne({ vivino_id: “vivino_id”, glugulp_id: “glugulp_id” })

Operation	Implementation
Update User	db.Users.updateOne({ username: “username”, email: “email”, password: “password”, firstname: “firstname”, lastname: “lastname”, age: “age”, location: “location”, type: “type” })
Update Comment	db.Comments.updateOne({ text: “text”, username: “username”, timestamp: “timestamp”, vivino_id: “vivino_id”, glugulp_id: “glugulp_id” })
Update Wine	db.Wines.updateOne({ vivino_id: “vivino_id”, comments: “comments” })
Delete User	db.Users.deleteOne({ username: “username” })
Delete Comment	db.Comments.deleteOne({ username: “username”, timestamp: “timestamp”, vivino_id: “vivino_id”, glugulp_id: “glugulp_id” })

NEO4J DB

In Neo4j database there are the following entities: User, Wine, and Winery that contain only the basic information needed to show a preview of the entity.

CRUD Operation

Operation	Implementation
Create User	CREATE (u:User {username: $username, email: $email}
Create Winery	CREATE (:Winery {title: $title, owner: $owner})
Create a follows relationship between a user and a winery	MATCH (u:User {username: $username}), (w:Winery {title: $title, owner: $owner}) MERGE (u)-[p:FOLLOWS]->(w) ON CREATE SET p.date = datetime()
Create a follows relationship between two users	MATCH (u:User {username: $username}), (t:User {username: $target}) MERGE (u)-[p:FOLLOWS]->(t) ON CREATE SET p.date = datetime()
Create a likes relationship between a user and a wine	MATCH (a:User), (b:Wine) WHERE a.username = $username AND (b.vivino_id = $vivino_id OR b.glugulp_id = $glugulp_id) MERGE (a)-[r:LIKES]->(b)
Get the number of likes of a wine	MATCH (w:Wine)<-[r:LIKES]-() WHERE w.vivino_id = $vivino_id OR w.glugulp_id = $glugulp_id RETURN COUNT(r)
Get the number of followed users	MATCH (:User {username: $username})- [r:FOLLOWS]->(u:User) RETURN COUNT(r)
Get the number of followers of a user	MATCH (:User {username: $username})<- [r:FOLLOWS]-(u:User) RETURN COUNT(r)
Get the number of followers of a winery	MATCH (:Winery {title: $title, owner: $owner})<-[r:FOLLOWS]-() RETURN COUNT(r)

Operation	Implementation
Get the followers of a users	MATCH (:User {username: $username})<-[:FOLLOWS]-(u:User) RETURN u.username, u.email ORDER BY u.username DESC
Get the users followed by a User	MATCH (:User {username: $username})-[:FOLLOWS]->(u:User) RETURN u.username, u.email ORDER BY u.username DESC
Check if user a follows user b	MATCH (a:User{username:$userA})-[r:FOLLOWS]->(b:User{username:$userB}) RETURN COUNT(*)
Check if a user is following a winery	MATCH (a:User{username:$user})-[r:FOLLOWS]->(b:Winery{title:$title, owner: $owner }) RETURN COUNT(*)
Check if exist a likes relationship between a user and a wine	MATCH (:User{username:$user})-[r:LIKES]->(w:Wine) WHERE (w.vivino_id = $vivino_id OR w.glugulp_id =$glugulp_id) RETURN COUNT(*)
Update User	MATCH (u:User {username: $username}) SET u.email = $newEmail
Delete a follows relationship between two users	MATCH (:User {username: $username})-[r:FOLLOWS]->(:User {username: $target}) DELETE r
Delete a follows relationship between a user and a winery	MATCH (:User {username: $username})-[r:FOLLOWS]->(:Winery {title: $title, owner: $owner}) DELETE r
Delete Winery	MATCH (w:Winery {title: $title, owner: $owner}) DETACH DELETE w
Delete User	MATCH (u:User) WHERE u.username = $username DETACH DELETE u
Delete a likes relationship between a user and a wine	MATCH (u:User{username:$username})-[r:LIKES]->(w:Wine) WHERE w.vivino_id = $vivino_id OR w.glugulp_id = $glugulp_id DELETE r

TEST AND STATISTICAL ANALYSIS

JAVA MODEL OF THE ENTITIES

Here we report the Java Model used for handling the entities of the databases.

For readability setters, getters and constructors were omitted.

```
public class User {
    4 usages
    private String username;
    4 usages
    private String email;
    4 usages
    private String password;
    4 usages
    private String firstname;
    4 usages
    private String lastname;
    4 usages
    private int age;
    4 usages
    private String location;
    4 usages
    private int type;
    4 usages
    private List<Winery> winerys;
```

```
public class Wine {
    4 usages
    private String vivino_id;
    4 usages
    private String glugulp_id;
    4 usages
    private String name;
    4 usages
    private String winemaker;
    4 usages
    private String country;
    4 usages
    private String varietal;
    4 usages
    private String grapes;
    4 usages
    private Integer year;
    4 usages
    private Double price;
    4 usages
    private String info;
    4 usages
    private String description;
    4 usages
    private List<Comment> comments;
```

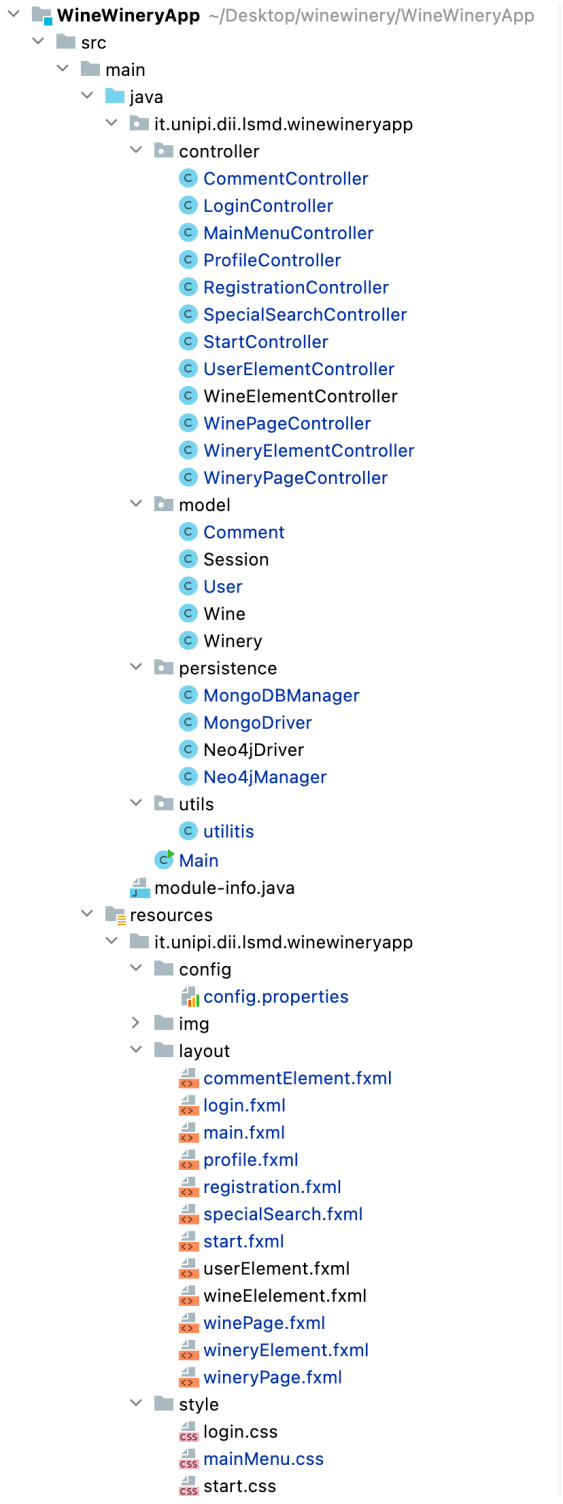
```
public class Comment {
    4 usages
    private String wine;
    4 usages
    private String username;
    4 usages
    private String text;
    4 usages
    private Date timestamp;
```

```
public class Winery {
    3 usages
    private String title;
    4 usages
    private List<Wine> wines;
```

JAVA PACKAGE STRUCTURE

Main Packages and Classes

In this section we describe the packages and the classes of which is composed the Java application.



it.unipi.dii.lsmd.winewineryapp.model

This package contains the classes needed for the java model of the application.

- User: class that contains user's information
- Wine: class that contains wine's information
- Winery: class that contains winery's information
- Comment: class that contains comment's information
- Session: class that contains information about the session

it.unipi.dii.lsmd.winewineryapp.persistence

This package contains the classes that are used to interface with the database.

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

it.unipi.dii.lsmd.winewineryapp.utilitis

This package contains utility functions.

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

it.unipi.dii.lsmd.winewineryapp.controller

- RegistrationController: this class manage the register page of application.
- LoginController: this class manage the login page of application.
- WinePageController: this class manage the page of a generic wine and allows the user to comment, like or add wine to their winery.
- WineryPageController: this class manage the page of a generic winery and allow the user to see the information about follower and main varietal, also the user who owns the winery can remove wine.
- ProfileController: this class manage the generic user page of the application and allows the user to view other users or edit their own data.
- CommentController: this class manage the view of comment.
- StartController: this class mange the initial start page of the application where user can do login or sign up.
- MainMenuController: this class manage the landing page after login and main menu that allow user to do search operation.
- SpecialSearchController: this class manage the special search page and allow to retrive suggestion, analytics and summary.
- WineryElementController: this class manage the view of winery element.
- UserElementController: this class manage the view of user element.
- WineElementController: this class manage the view of wine element.

MOST RELEVANT QUERIES

In this section, we present some of the most relevant queries, offering concise explanations of their functions, the data they take as input, the results they produce, and their execution procedures.

MONGODB

Returns wines by parameters

The query searches for wines given parameters

- Input: name, winemaker, country, varietal, grapes, min_year, max_year, min_price, max_price, skip, limit
- Output: list of wines that match the parameters

Mongo Java Manager:

```
public List<Wine> searchWinesByParameters (String name, String winemaker, String country,
                                           String varietal, String grapes, int min_year, int max_year,
                                           double min_price, double max_price, int skip, int limit) {
    List<Wine> wines = new ArrayList<>();
    Gson gson = new GsonBuilder().setDateFormat("yyyy-MM-dd hh:mm:ss").create();
    List<Bson> pipeline = new ArrayList<>();
    if (!name.isEmpty()) {
        Pattern pattern1 = Pattern.compile( regex: "^.*" + name + ".*$", Pattern.CASE_INSENSITIVE);
        pipeline.add(match(regex( fieldName: "name", pattern1)));}
    if (!winemaker.isEmpty()) {
        Pattern pattern2 = Pattern.compile( regex: "^.*" + winemaker + ".*$", Pattern.CASE_INSENSITIVE);
        pipeline.add(match(regex( fieldName: "winemaker", pattern2)));}
    if (!country.isEmpty()) {
        pipeline.add(match(eq( fieldName: "country", country)));}
    if (!varietal.isEmpty()) {
        pipeline.add(match(eq( fieldName: "varietal", varietal)));}
    if (!grapes.isEmpty()) {
        pipeline.add(match(eq( fieldName: "grapes", grapes)));}
    if (min_year != 0) {
        pipeline.add(match(and(gte( fieldName: "year", min_year))));}
    if(max_year != 0) {
        pipeline.add(match(and(lte( fieldName: "year", max_year))));}
    if (min_price != 0)
        pipeline.add(match(and(gte( fieldName: "price", min_price))));
    if(max_price != 0) {
        pipeline.add(match(and(lte( fieldName: "price", max_price))));}
    pipeline.add(sort(ascending( ...fieldNames: "price")));
    pipeline.add(skip(skip * limit));
    pipeline.add(limit(limit));
    List<Document> results = (List<Document>) winesCollection.aggregate(pipeline).into(new ArrayList<>());
    Type winesListType = new TypeToken<ArrayList<Wine>>(){}.getType();
    wines = gson.fromJson(gson.toJson(results), winesListType);
    return wines;
}
```

Mongo:

```
db.Wines.aggregate(
    {
        $match: {
          name: "Lepore Trebbiano d'Abruzzo 2014",
          winemaker: 'Lepore',
          country: 'Italia',
          varietal: 'Bianco',
          price: { $gt: 1, $lt: 20 },
          year: { $gt: 2010, $lt: 2020 }}
    },
    { $sort: { price: -1 } },
    { $limit: 5 },
    { $skip: 0 }
);
```

Returns bad users

The query searches for all the bad users

- Input: skipDoc, limitDoc
- Output: list of users whose comments have been deleted at least one time

Mongo Java Manager:

```
public List<User> getBadUsers(int skipDoc, int limitDoc) {
    List<User> results = new ArrayList<>();
    Gson gson = new GsonBuilder().serializeSpecialFloatingPointValues().create();
    Consumer<Document> convertInUser = doc -> {
        User user = gson.fromJson(gson.toJson(doc), User.class);
        results.add(user);
    };
    Bson filter = match(gte( fieldName: "deletedComments",  value: 1));
    Bson skip = skip(skipDoc);
    Bson limit = limit(limitDoc);
    usersCollection.aggregate(Arrays.asList(filter, skip, limit)).forEach(convertInUser);
    return results;
}
```

Mongo

```
db.Users.aggregate(
      { $match: { deletedComment: { $gte: 1 } } },
      { $skip: 0 },
      { $limit: 5 }
)
```

Returns all the comments

The query searches for all the comments of a certain wine

- Input: Wine, skipDoc, limitDoc
- Output: list of all the comments

Mongo Java Manager:

```
public List<Comment> getAllComments (Wine wine, int skipDoc, int limitDoc) {
    List<Comment> results = new ArrayList<>();
    Gson gson = new GsonBuilder().setDateFormat("yyyy-MM-dd hh:mm:ss").create();
    Consumer<Document> convertInComment = doc -> {
        Comment comment = gson.fromJson(gson.toJson(doc), Comment.class);
        results.add(comment);
    };
    Bson filter = match(and(eq( fieldName: "vivino_id", wine.getVivino_id()),
            eq( fieldName: "glugulp_id", wine.getGlugulp_id())));
    Bson sort = sort(Indexes.ascending( ...fieldNames: "timestamp"));
    Bson skip = skip(skipDoc);
    Bson limit = limit(limitDoc);

    commentsCollection.aggregate(Arrays.asList(filter, sort, skip, limit)).forEach(convertInComment);
    return results;
}
```

Mongo:

```
db.Comments.aggregate(
    {
        $match: {
          $and: [
            { vivino_id: '161471613.0' },
            { gluglup_id: null }]}
    },
    { $sort: { timestamp: 1 } },
    { $skip: 0 },
    { $limit: 5 }
)
```

Returns wines with the highest number of comments

The query searches for wines with the highest number of comments within a specified time frame. The search can be based on the last week, the last month, or all-time.

- Input: period (all, month, week), skipDoc, limitDoc
- Output: list with the name of the wines and their number of comments

Mongo Java Manager:

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

    Gson gson = new GsonBuilder().setDateFormat("yyyy-MM-dd hh:mm:ss").create();

    Bson filter = match(gte( fieldName: "timestamp", filterDate));
    Bson group = group(
            new Document("$ifNull", Arrays.asList("$vivino_id", "$glugulp_id")),
            sum( fieldName: "totalComment",  expression: 1)
    );
    Bson sort = sort(Indexes.descending( ...fieldNames: "totalComment"));
    Bson skip = skip(skipDoc);
    Bson limit = limit(limitDoc);

    AggregateIterable<Document> countResults = commentsCollection.aggregate(Arrays.asList(filter,
            group, sort, skip, limit));
    List<Pair<Wine, Integer>> resultList = new ArrayList<>();

    for (Document doc : countResults) {
        String wineId = doc.getString( key: "_id");
        Document myDoc = (Document) winesCollection.find(or(eq( fieldName: "vivino_id", wineId),
                eq( fieldName: "glugulp_id", wineId))).first();
        Wine wine = gson.fromJson(gson.toJson(myDoc), Wine.class);

        Integer count = doc.getInteger( key: "totalComment");
        resultList.add(new Pair<>(wine, count));
    }
    return resultList;
}
```

Mongo:

```
db.Comments.aggregate(
    {
        $match: {
          timestamp: { $gte: '0001-01-01 00:00:00' }}
    },
    {
        $group: {
          _id: {
            $ifNull: ['$vivino_id', '$glugulp_id']
          },
          totalComment: { $sum: 1 }}
    },
    { $sort: { totalComment: -1 } },
    { $skip: 0 },
    { $limit: 5 }
);
```

Returns the top varietals with more comments

The query searches for the varietals with the highest number of comments within a specified time frame. The search can be based on the last week, the last month, or all-time.

- Input: period (all, month, week)
- Output: list of varietals and their number of comments

Mongo Java Manager:

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

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

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

    return results;
}
```

Mongo:

```
db.Comments.aggregate(
    {
      $match: {
        timestamp: { $gte: '0001-01-01 00:00:00' }}
    },
    {
      $group: {
        _id: '$varietal',
        tots: { $sum: 1 }}
    },
    { $sort: { tots: -1 }
    }
);
```

Returns the top varietals by the number of wines

The query searches for the varietals with the highest number of wines.

- Output: list of varietals and their number of wines
- Mongo Java Manager:

```
public List<Pair<String, Integer>> getVarietalsSummaryByNumberOfWines (){
    List<Pair<String,Integer>> varietals = new ArrayList<>();

    Bson group = group( id: "$varietal", sum( fieldName: "totalWine",  expression: 1));
    Bson project = project(fields(excludeId(), computed( fieldName: "varietal",  expression: "$_id"),
            include( ...fieldNames: "totalWine")));
    Bson sort = sort(descending( ...fieldNames: "totalWine"));

    List<Document> results = (List<Document>) winesCollection.aggregate(Arrays.asList(group,
            project, sort)).into(new ArrayList<>());

    for (Document document: results)
    {
        varietals.add(new Pair(document.getString( key: "varietal"),
                document.getInteger( key: "totalWine")));
    }
    return varietals;
}
```

Mongo:

```
db.Wines.aggregate(
    {
        $group: {
          _id: '$varietal',
          totalWine: { $sum: 1 }}
    },
    {
        $project: {
          _id: 0,
          varietal: '$_id',
          totalWine: 1}
    },
    {
        $sort: { totalWine: -1 }
    }
);
```

Returns users with the highest number of varietals in their wineries

The query searches for the users with the highest number of varietals in their wineries. In this query we incorporated the $unwind operator because we had to calculate aggregated values for individual elements within an array of

embedded documents. This was necessary to perform computations separately for each element within array (array of wineries).

- Input: limitDoc, skipDoc
- Output: list of Users with the highest number of varietals in their wineries

Mongo Java Manager:

```
public List<Pair<User, Integer>> getTopVersatileUsers (int skipDoc, int limitDoc) {
    List<Pair<User, Integer>> results = new ArrayList<>();
    Gson gson = new GsonBuilder().serializeSpecialFloatingPointValues().create();
    Consumer<Document> convertInUser = doc -> {
        User user = gson.fromJson(gson.toJson(doc), User.class);
        results.add(new Pair(user, doc.getInteger( key: "totalWine")));
    };
    Bson unwind1 = unwind( fieldName: "$winerys");
    Bson unwind2 = unwind( fieldName: "$winerys.wines");
    // Distinct occurrences
    Bson groupMultiple = new Document("$group",
            new Document("_id", new Document("username", "$username")
                    .append("email", "$email")
                    .append("password", "$password")
                    .append("firstname", "$firstname")
                    .append("lastname", "$lastname")
                    .append("age", "$age")
                    .append("location", "$location")
                    .append("varietals", "$winerys.wines.varietals")
            ));
    // Sum all occurrences
    Bson group = new Document("$group",
            new Document("_id",
                    new Document("username", "$_id.username")
                            .append("email", "$_id.email")
                            .append("password", "$_id.password")
                            .append("firstname", "$_id.firstname")
                            .append("lastname", "$_id.lastname")
                            .append("age", "$_id.age")
                            .append("location", "$_id.location"))
                    .append("totalVarietals",
                            new Document("$sum", 1)));
    Bson project = project(fields(excludeId(),
            computed( fieldName: "username",  expression: "$_id.username"),
            computed( fieldName: "email",  expression: "$_id.email"),
            computed( fieldName: "password",  expression: "$_id.password"),
            computed( fieldName: "firstname",  expression: "$_id.firstname"),
            computed( fieldName: "lastname",  expression: "$_id.lastname"),
            computed( fieldName: "age",  expression: "$_id.age"),
            computed( fieldName: "location",  expression: "$_id.location"),
            include( ...fieldNames: "totalVarietals")));
    Bson sort = sort(descending( ...fieldNames: "totalVarietals"));
    Bson skip = skip(skipDoc);
    Bson limit = limit(limitDoc);
    usersCollection.aggregate(Arrays.asList(unwind1, unwind2, groupMultiple, group,
            project, sort, skip, limit)).forEach(convertInUser);
    return results;
```

Mongo:

```
db.Users.aggregate(
    { $unwind: { path: '$winerys' } },
    { $unwind: { path: '$winerys.wines' } },
    { $group: {
        _id: {
          username: '$username',
          email: '$email',
          password: '$password',
          firstname: '$firstname',
          lastname: '$lastname',
          age: '$age',
          location: '$location',
          varietal: '$winerys.wines.varietal'
        }}
    },
    { $group: {
        _id: {
          username: '$_id.username',
          email: '$_id.email',
          password: '$_id.password',
          firstname: '$_id.firstname',
          lastname: '$_id.lastname',
          age: '$_id.age',
          location: '$_id.location'
        },
        totalVarietals: { $sum: 1 }}
    },
    { $project: {
        _id: 0,
        username: '$_id.username',
        email: '$_id.email',
        password: '$_id.password',
        firstName: '$_id.firstName',
        lastName: '$_id.lastName',
        age: '$_id.age',
        location: '$_id.location',
        totalVarietals: 1}
    },
    { $sort: { totalVarietals: -1 } },
    { $skip: 0 },
    { $limit: 5 }
);
```

NEO4J

Most liked wine

This query returns wines with the highest number of likes.

- Input: how many wines to skip and how many wines to show
- Output: list of the most liked wines and their number of likes

```
MATCH ( :User)-[l:LIKES]->(w:Wine)
RETURN w. vivino_id AS VivinoId, w. glugulp_id AS GlugulpId,
       w. name AS Name, w.winemaker AS Winemaker,
       w. varietal AS Varietal
COUNT(l) AS like_count
ORDER BY like count DESC
SKIP $skip
LIMIT $limit
```

Most followed user

This query returns users with the highest number of followers.

- Input: how many users to skip and how many users to show
- Output: list of the most followed users and their number of followers

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

Most followed winery

This query returns wineries with the highest number of followers.

- Input: how many wineries to skip and how many wineries to show
- Output: list of the most followed wineries and their number of followers

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

Get varietals summary by likes

The query returns varietals with the highest number of likes

- Output: list of the most liked varietals and their number of likes

```
MATCH (w:Wine)<-[l:LIKES]-(:User)
RETURN count(l) AS likes, w. varietal AS Varietal
ORDER BY Likes DESC
```

Suggested wines

The query returns a list of suggested wines, the suggestion is based on:

- first level: wines liked by followed users
- second level: likes liked by users that are 2 FOLLOWS hops far from the logged user
- Input: username of the user, how many wines to skip and how many wines to show from first level, how many wines to skip and how many wines to show from second level
- Output: list of the suggested wines

```
MATCH (target:Wine)<-[r:LIKES]-(u:User)<-[:FOLLOWS]-(me:User{username:$username})
WHERE NOT EXISTS((me)-[:LIKES]->(target))
RETURN target.vivino_id AS VivinoId, target.glugulp_id AS GlugulpId, target.name as Name,
       target.winemaker AS Winemaker, target.varietal AS Varietal, COUNT(*) AS nOccurences
ORDER BY nOccurences DESC
SKIP $skipFirstLevel
LIMIT $firstlevel
UNION
MATCH (target:Wine)<-[r:LIKES]-(u:User)<-[:FOLLOWS*2..2]-(me:User{username:$username})
WHERE NOT EXISTS((me)-[:LIKES]->(target))
RETURN target.vivino_id AS VivinoId, target.glugulp_id AS GlugulpId, target.name as Name,
       target.winemaker AS Winemaker, target.varietal AS Varietal, COUNT(*) AS nOccurences
ORDER BY nOccurences DESC
SKIP $skipSecondLevel
LIMIT $secondLevel
```

Suggested wineries

The query returns a list of suggested wineries. The suggestion is based on:

- first level: most followed wineries followed by followed users
- second level: most followed wineries followed by users that are 2 FOLLOWS hops far from the logged user
- Input: username of the user, how many wineries to skip and how many wineries to show from first level, how many wineries to skip and how many wineries to show from second level
- Output: list of the suggested wineries

```
MATCH (target:Winery)<-[f:FOLLOWS]-(u:User)<-[:FOLLOWS]-(me:User{username:$username}),
      (target)<-[r:FOLLOWS]-(n:User) WITH DISTINCT me, target
COUNT(DISTINCT r) AS numFollower, COUNT(DISTINCT u) AS follow
WHERE NOT EXISTS((me)-[:FOLLOWS]->(target))
RETURN target.owner AS Owner, target.title AS Title, numFollower + follow AS followers
ORDER BY followers DESC
SKIP $skipFirstLevel
LIMIT $firstLevel
UNION
MATCH (target:Winery)<-[f:FOLLOWS]-(u:User)<-[:FOLLOWS*2..2]-(me:User{username:$username}),
      (target)<-[r:FOLLOWS]-(n:User) WITH DISTINCT me, target,
COUNT(DISTINCT r) AS numFollower, COUNT(DISTINCT u) AS follow
WHERE NOT EXISTS((me)-[:FOLLOWS]->(target))
RETURN target.owner AS Owner, target.title AS Title, numFollower + follow AS followers
ORDER BY followers DESC
SKIP $skipSecondLevel
LIMIT $secondLevel
```

Suggested users

The query returns a list of suggested users. The suggestion is based on:

- first level: most followed users who are 2 FOLLOWS hops far from the logged user
- second level: most followed users that have likes in common with the logged user
- Input: username of the user, how many users to skip and how many users to show from first level, how many users to skip and how many users to show from second level
- Output: list of the suggested users

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

QUERIES ANALYSIS

The most queries consist in read operations and so we can say that the application is read-heavy. In the table, we can see the frequency of each query.

Query	Frequency
Search Wine by name	High
Search Wine by winemaker	High
Search Wine by year	Medium
Search Wine by price	High
Search Wine by varietal	Medium
Search winery by name	Medium
Search User by username	Medium

As we can see from the application's code, the first five queries in the table are all integrated into the single "searchWineByParameters" query.
This consolidated query is utilized in the application's search feature and can be executed separately by specifying only the relevant parameter.

INDEX ANALYSIS

To improve the performances of the application we performed index analysis, in this subsection we will discuss the introduction of indexes for both MongoDB and Neo4J.

INDEX MONGODB

Index Type	Collection	Index Name	Index Field
Single Field Index	Wines	name_1	name
Single Field Index	Wines	winemaker_1	winemaker
Single Field Index	Wines	price_1	price
Single Field Index	Wines	varietal_1	varietal
Single Field Index	Wines	year_1	year
Single Field Index	User	winerys.title_1	winery.title
Single Field Index	User	username_1	username

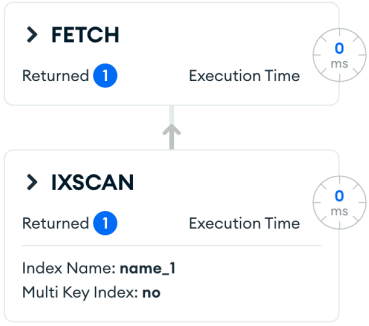
Indexes regarding Wine collection are particularly convenient because wine documents are almost not updatable, only the comments field can be updated but the application does not require frequent queries involving this field.
There will not be any overhead related to updates operations and the overhead related to insert operation will affect only the DB Updater and not the users (only the DB_Updater can create and insert new wine).

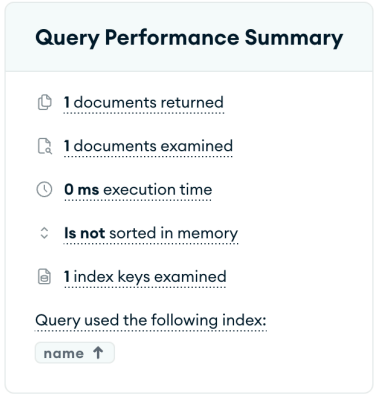
We will perform statistical tests to understand whether the insertion of an index is beneficial or not in terms of query performance.

Index on wine name

In this performance analysis we can see the improvements on equality matches using the wine name index.

	Results without Index	Results with index
Execution time	23	0
Documents examined	35707	1
Index keys examined	0	1

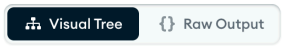


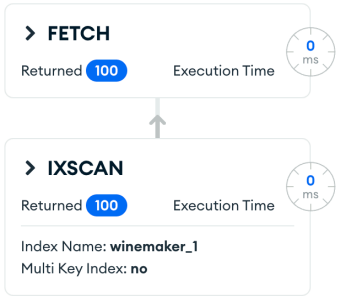


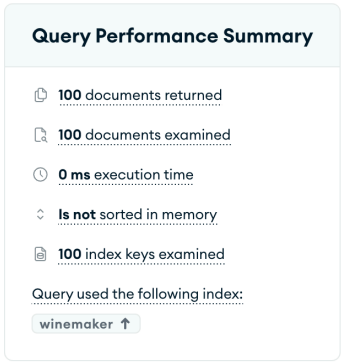
Index on winemaker

In this performance analysis we can see the improvements on equality matches using the winemaker index.

	Results without Index	Results with index
Execution time	18	0
Documents examined	35707	100
Index keys examined	0	100



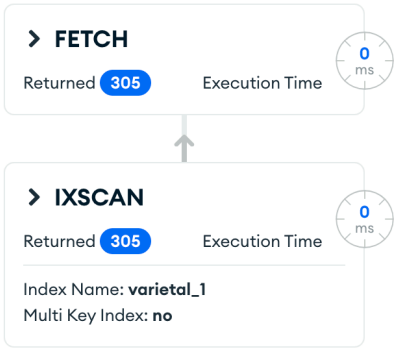


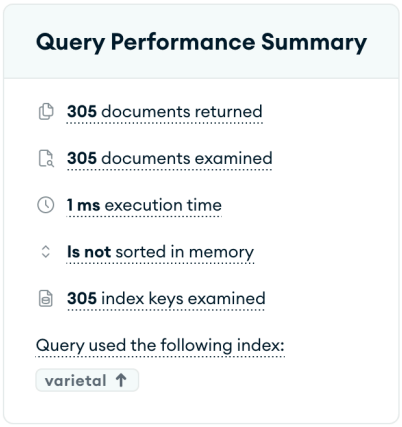


Index on Varietal

In this performance analysis we can see the improvements on equality matches using the varietal index.

	Results without Index	Results with index
Execution time	19	1
Documents examined	35707	305
Index keys examined	0	305

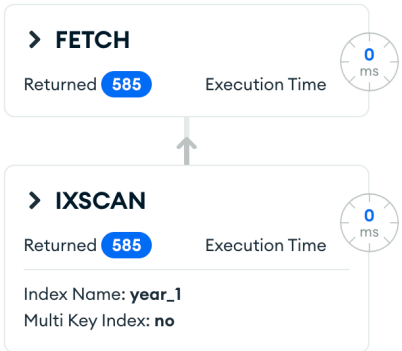


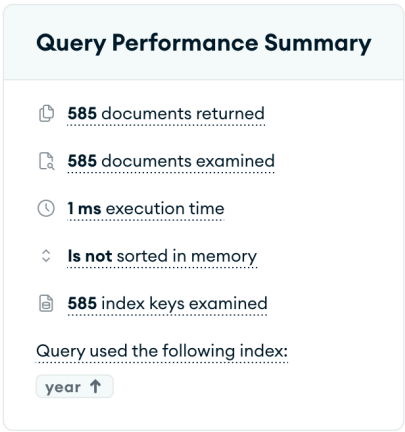


Index on year

In this performance analysis we can see the improvements on equality matches using the year index.

	Results without Index	Results with index
Execution time	19	1
Documents examined	35707	585
Index keys examined	0	585

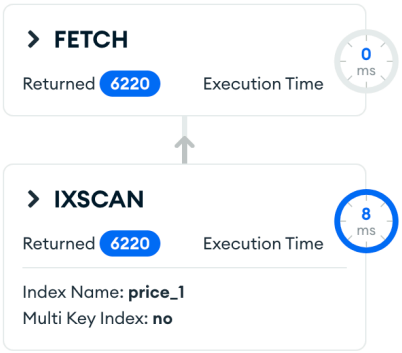


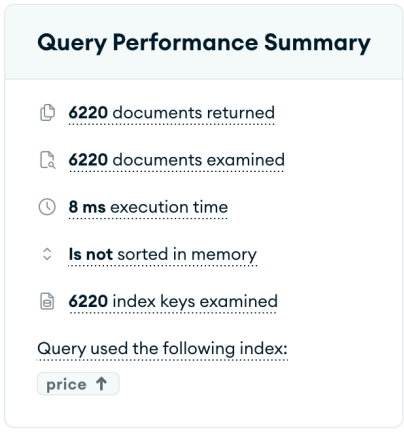


Index on price

In this performance analysis we can see the improvements on equality matches using the price index.

	Results without Index	Results with index
Execution time	21	8
Documents examined	35707	6220
Index keys examined	0	6220

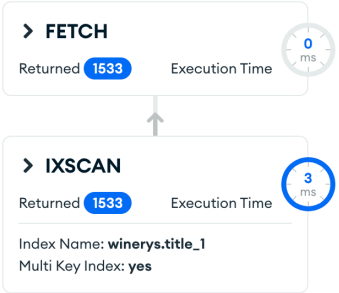


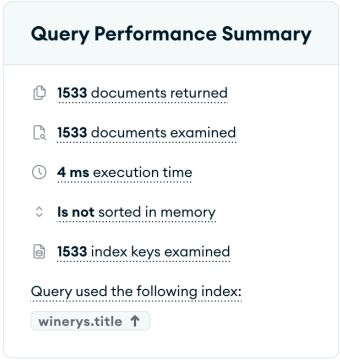


Index on Winery name

Regarding winery, we analyzed an index on the “name” field to increase the performance of the SearchWinery_by_Name query, the performances were good but the index was not included in the application. The main reason is that winery can be updated by the user, so the insertion of the index will slow down some of the user’s write operations (add and remove winery)

	Results without Index	Results with index
Execution time	133	4
Documents examined	5008	1553
Index keys examined	0	1553

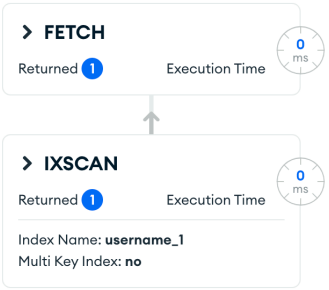


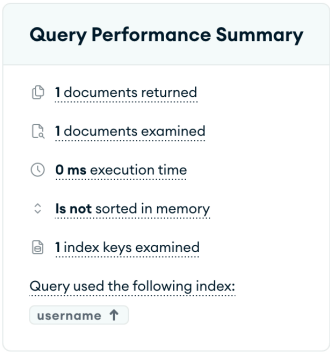


Index on username

In this performance analysis we can see the improvements on equality matches using the price index.

	Results without Index	Results with index
Execution time	7	1
Documents examined	5008	1
Index keys examined	0	1





INDEX NEO4J

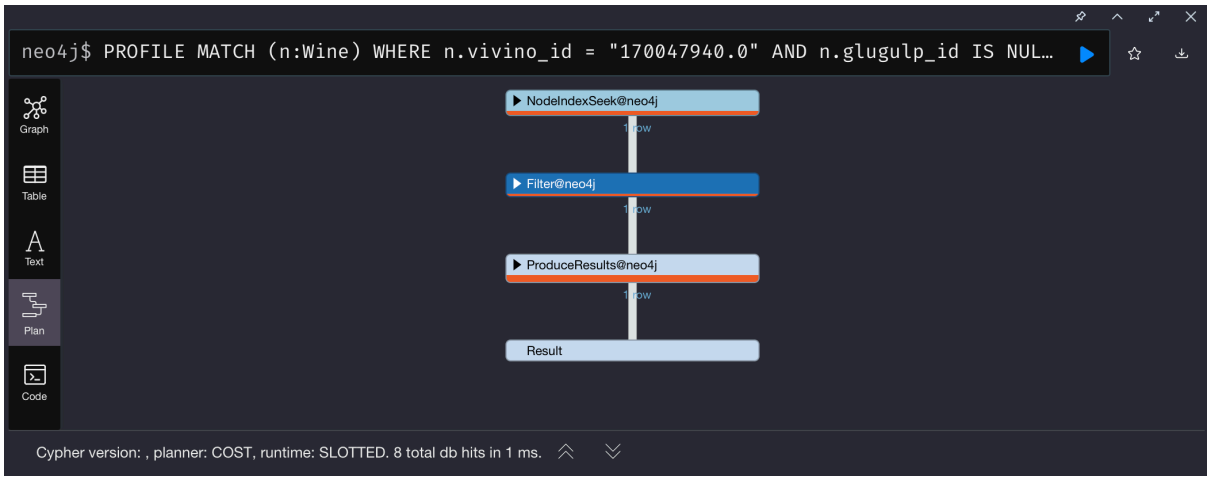
In order to improve the read operations' performance, the following indexes are introduced:

Index Type	Index Label	Index Name	Index Field
RANGE	:User	username_index	username
RANGE	:Wine	vivino_index	vivino_id
RANGE	:Wine	glugulp_index	glugulp_id

Index on vivino_id and glugulp_id

In the performance analysis, we can observe an improvement in exact matches using the bottles index. Thanks to this index, only items that match the specified value or pattern are examined, greatly reducing the time required to perform the operations.

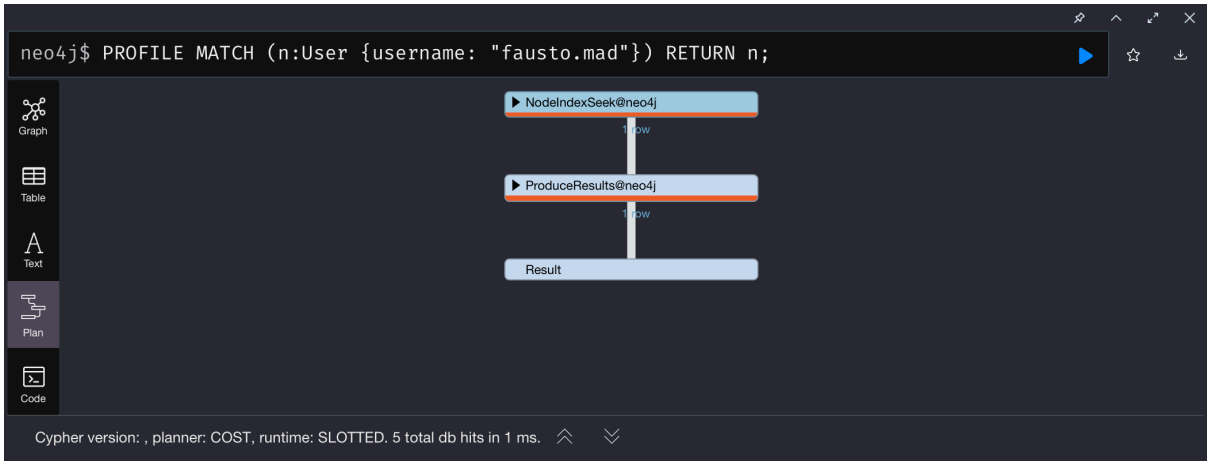
	Results without Index	Results with index
Execution time	34	1
Hits	73647	8



Index on username

As we can observe, in this case as well, adding an index on the username to Neo4j's database is beneficial for improving performance.

	Results without Index	Results with index
Execution time	26	1
Hits	10016	5

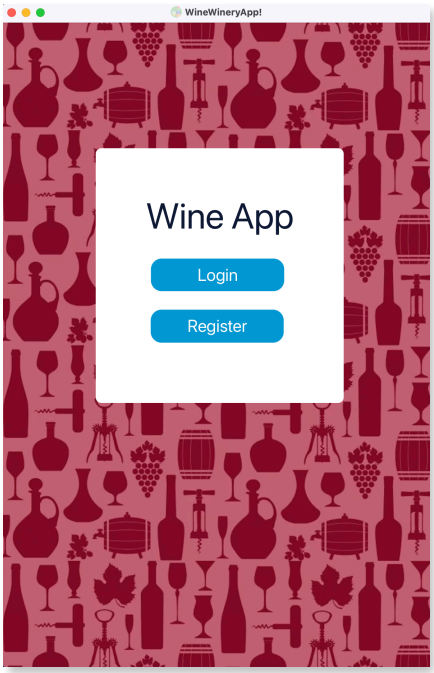


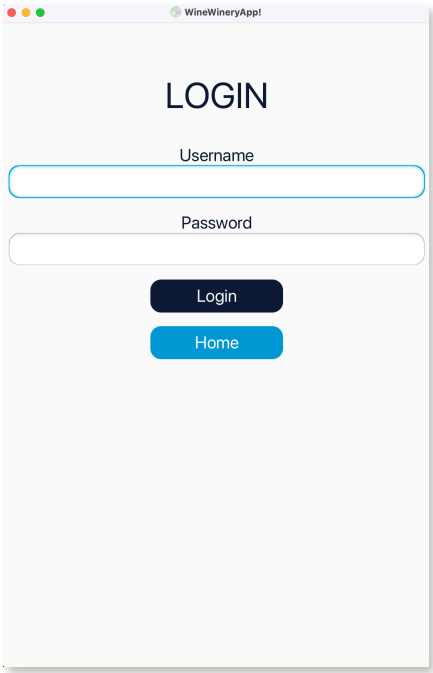
APPLICATION MANUAL

In this section we explain how a user can us WineWineryApp

LOGIN & REGISTRATION

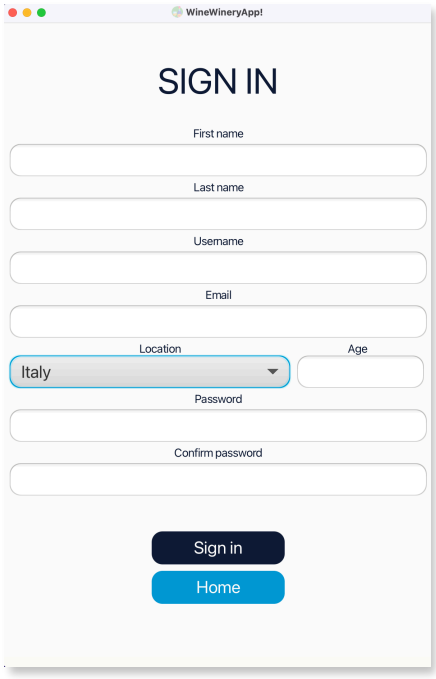
To have access to all the functionalities of the application, the user must be logged; after the login, the Main Page is shown.





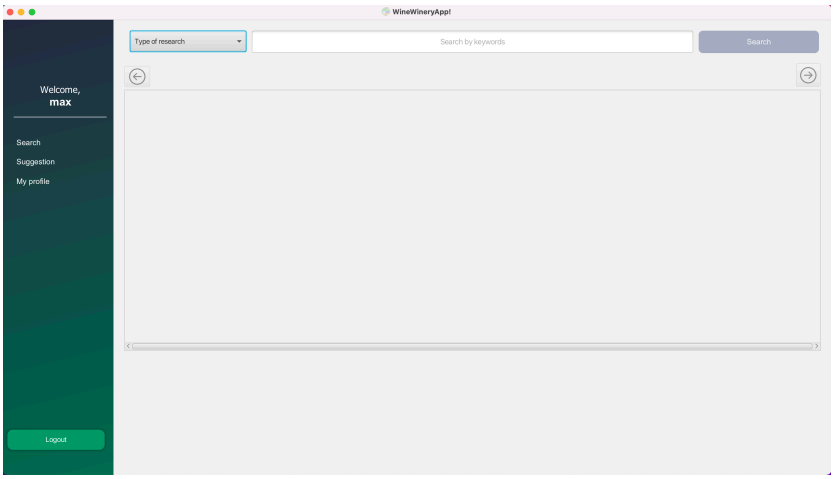
Registered users can login to WineWineryApp after they inserted their username and password on the login form and they clicked the login button.

If a user is not registered to the application, can join the community after clicking on the sign up button, inserting their personal information on the registration form and clicking on the registration button. To ensure security , all the information is checked before the registration of a user and if there is a field that does not respect the rules, the registration is rejected with a message.

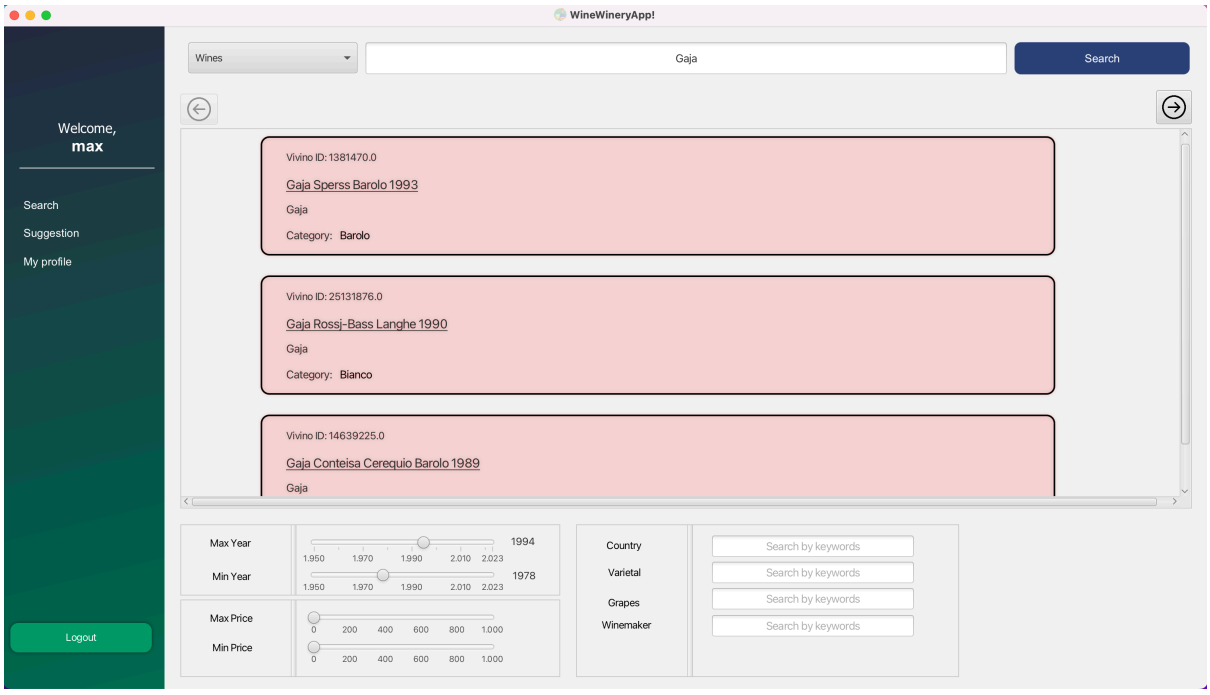


MAIN PAGE

From this page, users can search for wine, winery or users by selecting the type of research they are interested to and by typing the keyword of username on the search bar and pressing the search button.



During a wine search, users can specify some filters of search by selecting min and max price min and max year, varietal, winemaker, country or grapes.

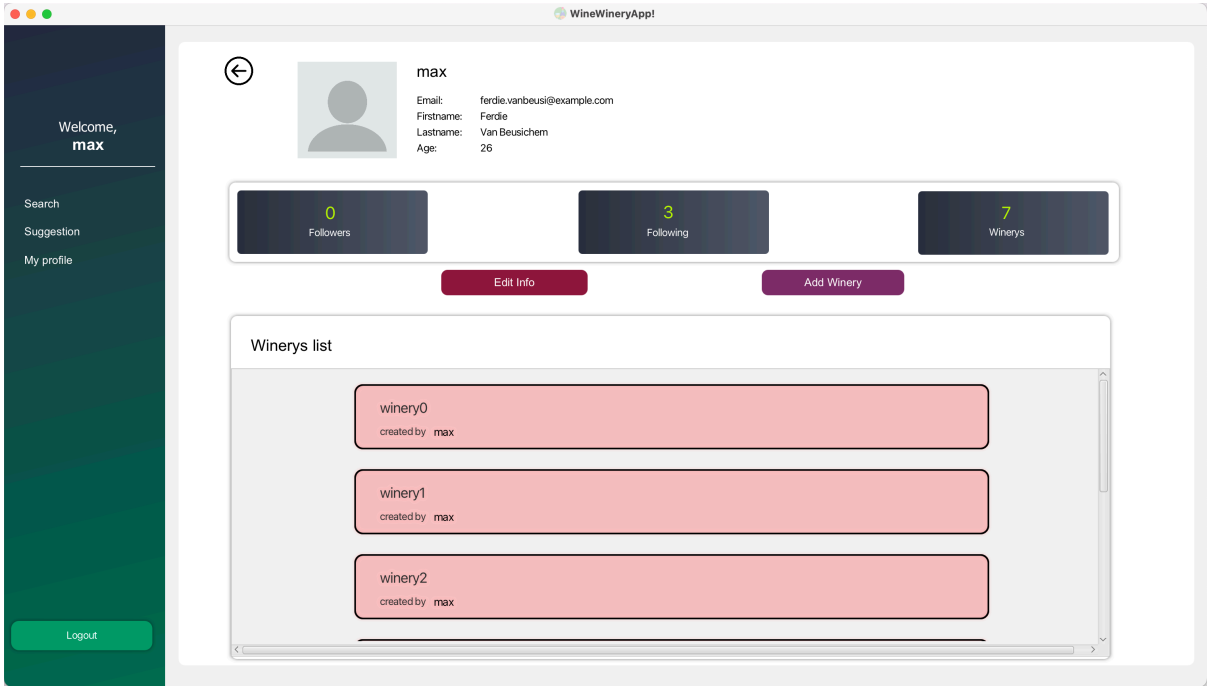


The user can navigate through his result with two buttons on the top-left and top- right. If we want to get all the information just click on the title of the bottle/winery/user to open the dedicated page with all the information.
In the sidebar there is a column menu : the first one brings us to the main page, the second one to the suggestion and analytics part, the third one brings

directly to the logged user profile page and at last the logout button to exit.

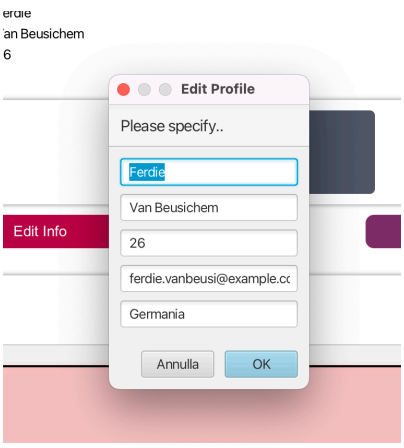
MY PROFILE

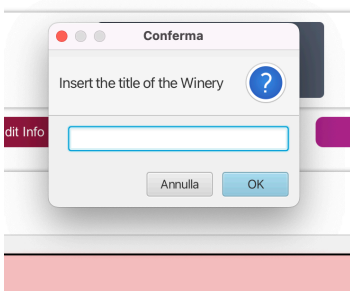
The profile page shows the user's main information, as well as the number of user follows, followers, and winery created.



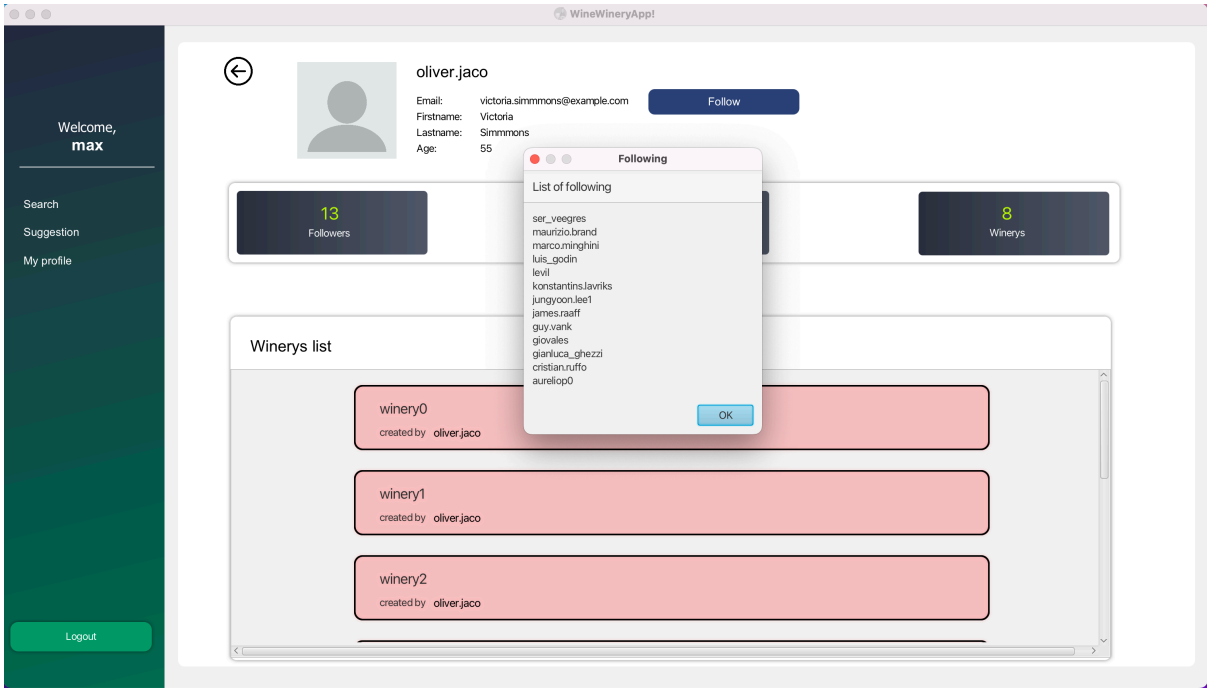
The user can update personal information or create a new winery.

There is a preview of the wineries created.



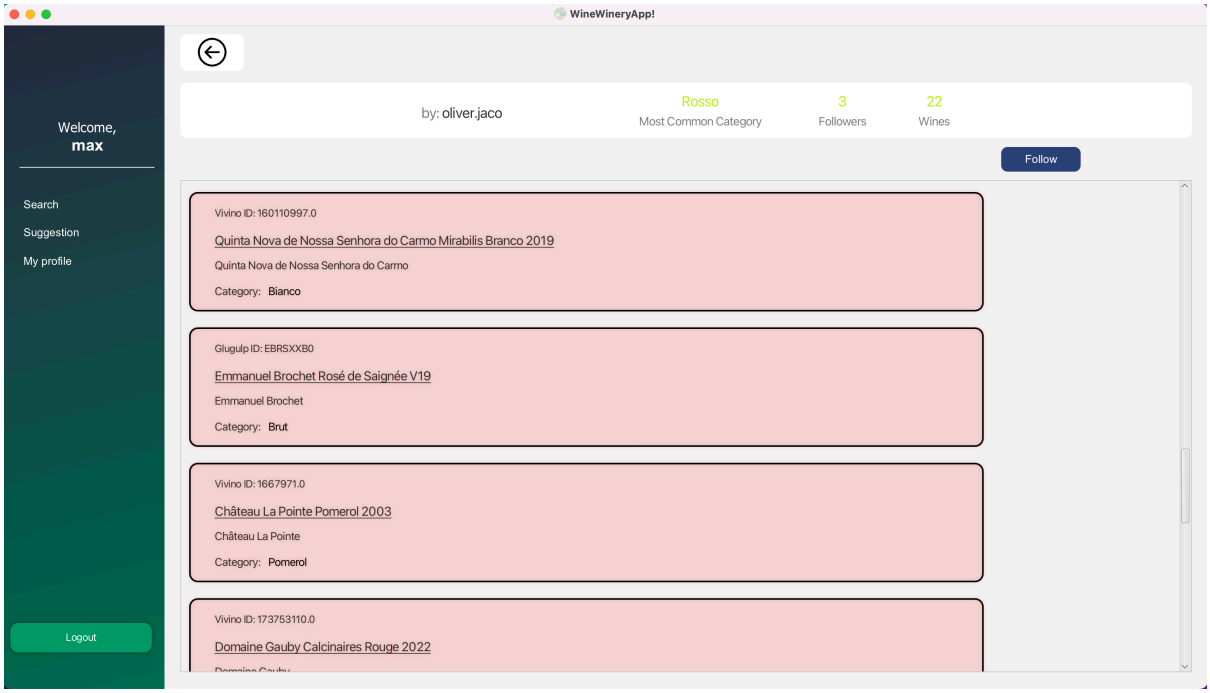


On the user's page there is a Follow button to add the user to our list of followers. We can browse both his followers and the users' followers, when we click on the user name the user's profile page is displayed.



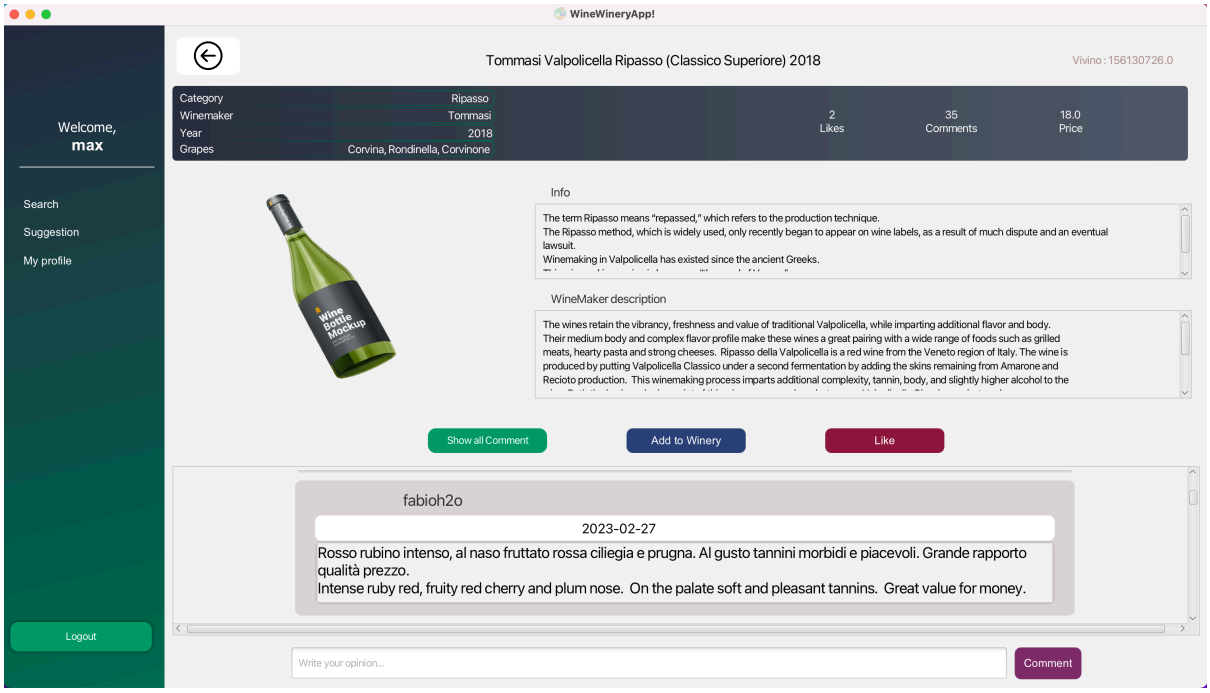
WINERY PAGE

On the page of a winery we can see the list of wines inside it and general information about the winery, we can do follow to the winery via the button.

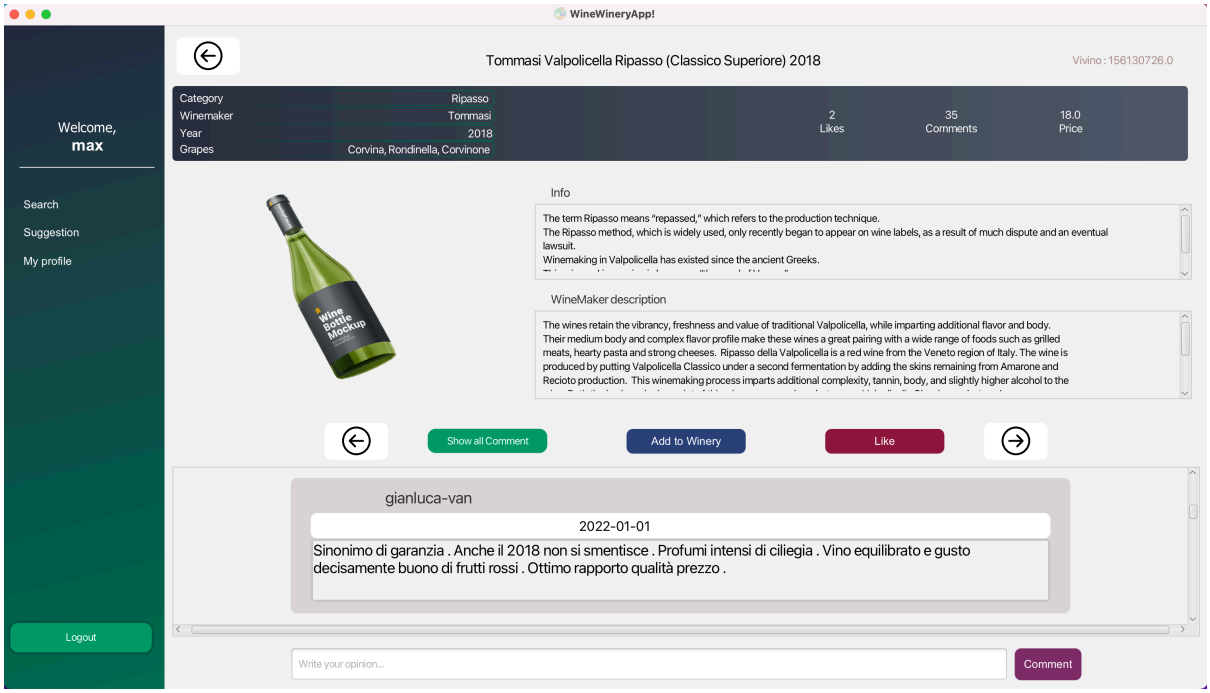


WINE PAGE

On the page of a bottle all the information and the list of recent comments are shown, if we want to visualize all the comments we can click on the view all comment button which will show us the list of the least recent comments in blocks of 10 comments. We can like, add the bottle to one of our winery or comment. We can visit the profile of a user who commented by clicking on the username

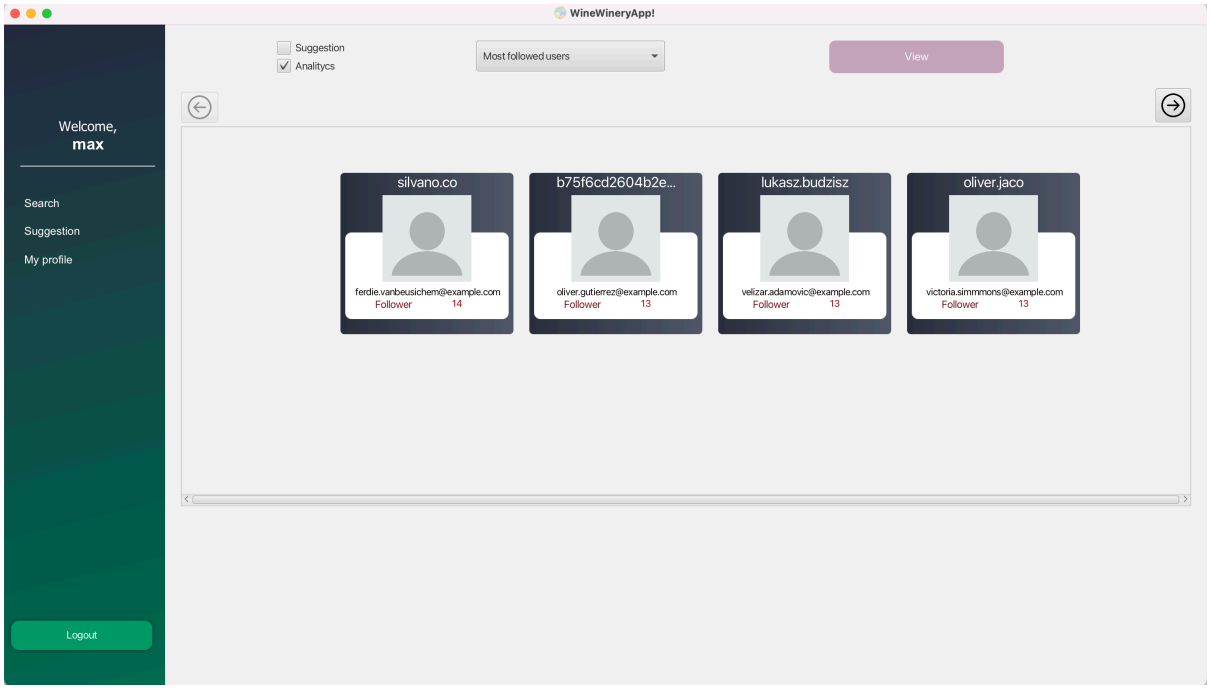


Once we have commented, we can edit the comment or delete it.

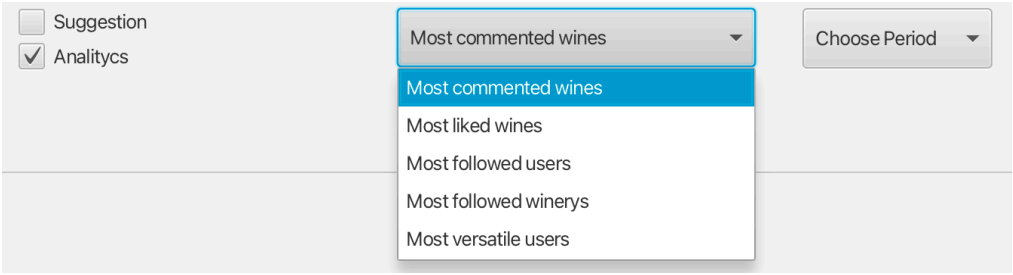


SUGGESTION PAGE

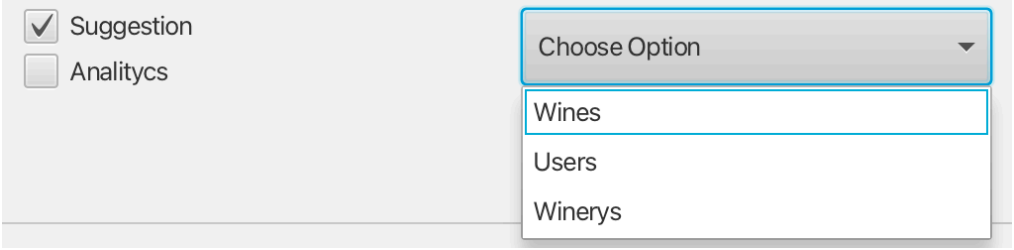
Clicking the Suggestion button on the side menu will open the page where it is possible for the normal user to view both Suggestions of Wines, Users and Wineries and analytics on the platform so they can explore new content.



The analytics available are:

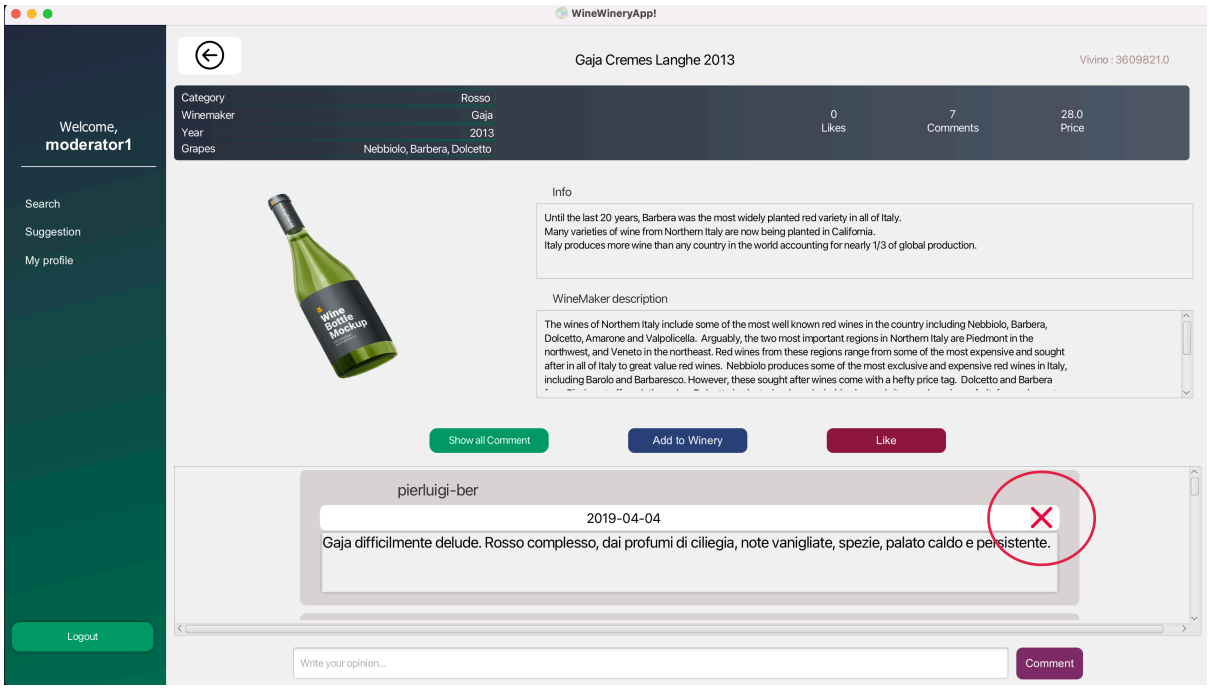


The suggestions available are:

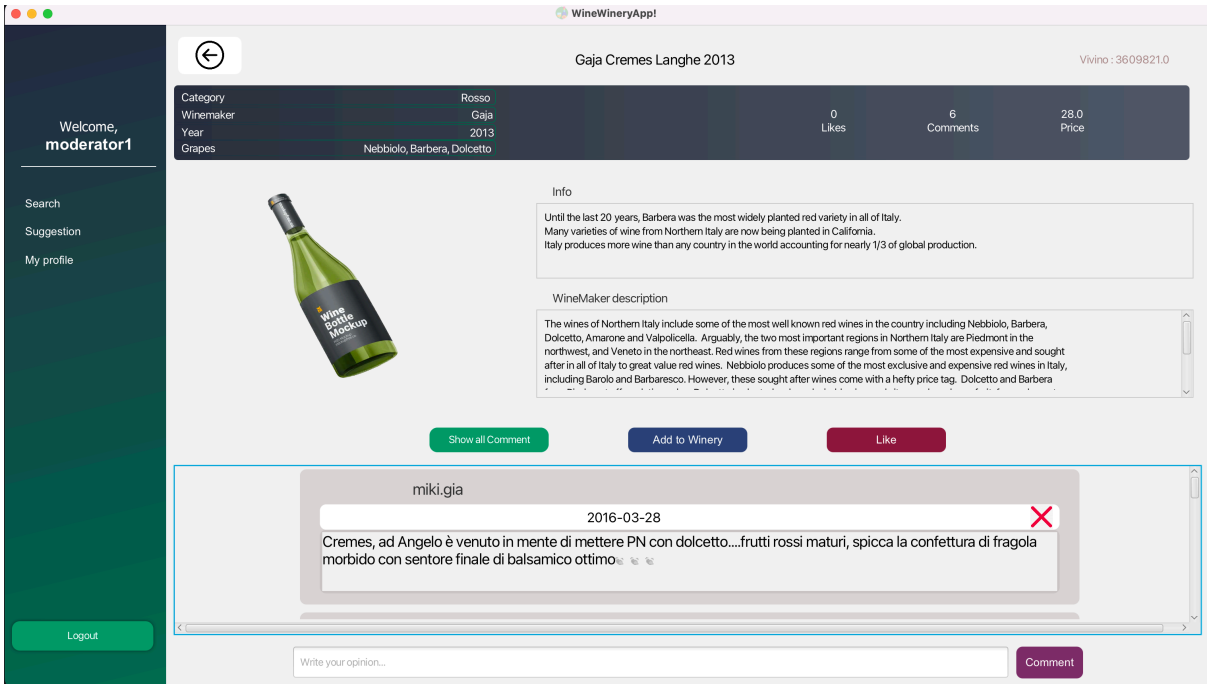


MODERATOR

Moderators have a similar view to normal users.



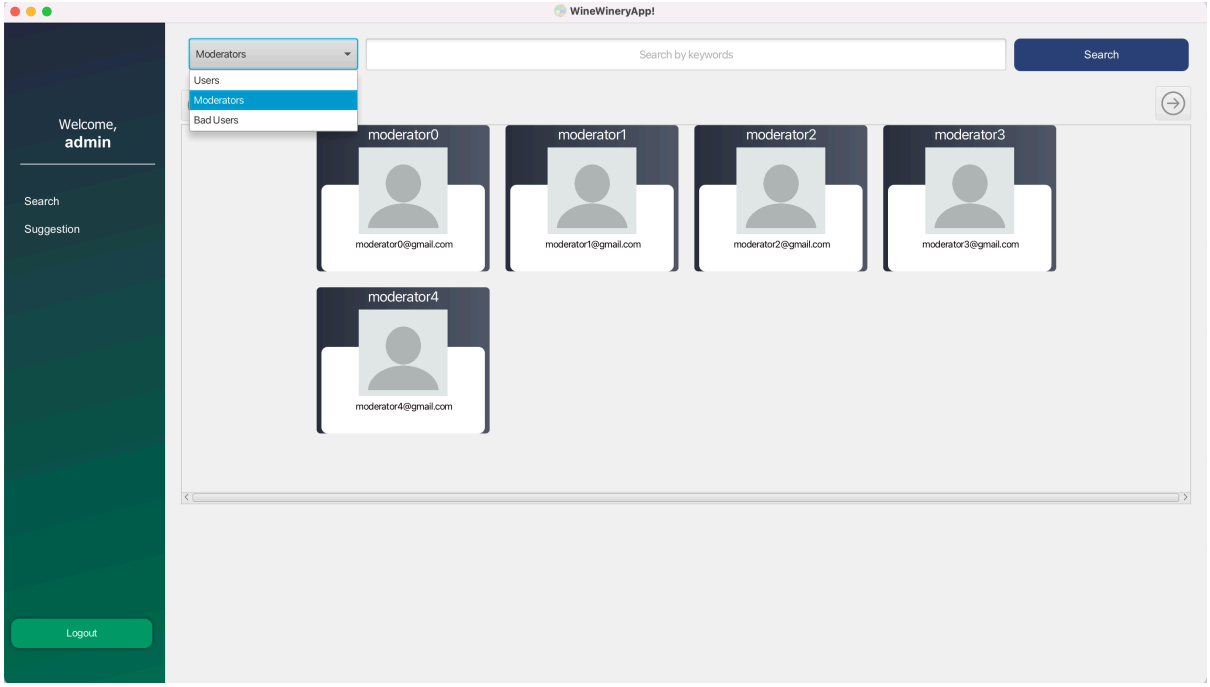
The function that distinguishes them from the latter lies in the ability to be able to delete user comments, thus reporting them to the admin.



ADMIN

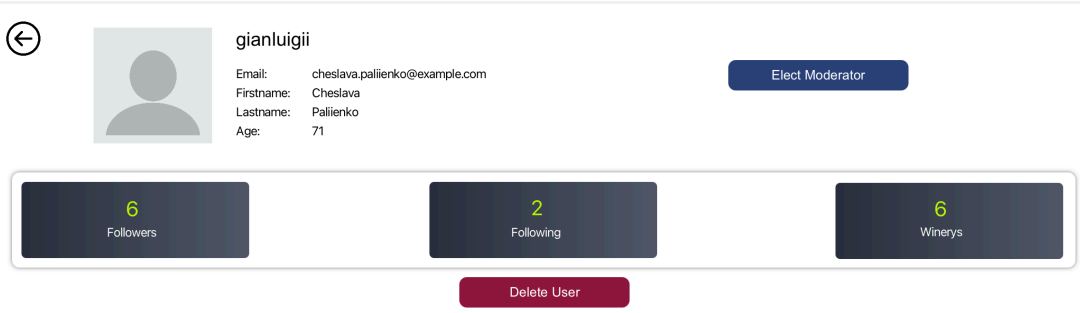
The system administrator has a different view than normal users and moderators.

The standard search part only allows him to search for users by filtering them by normal users, moderators, or Bad users.

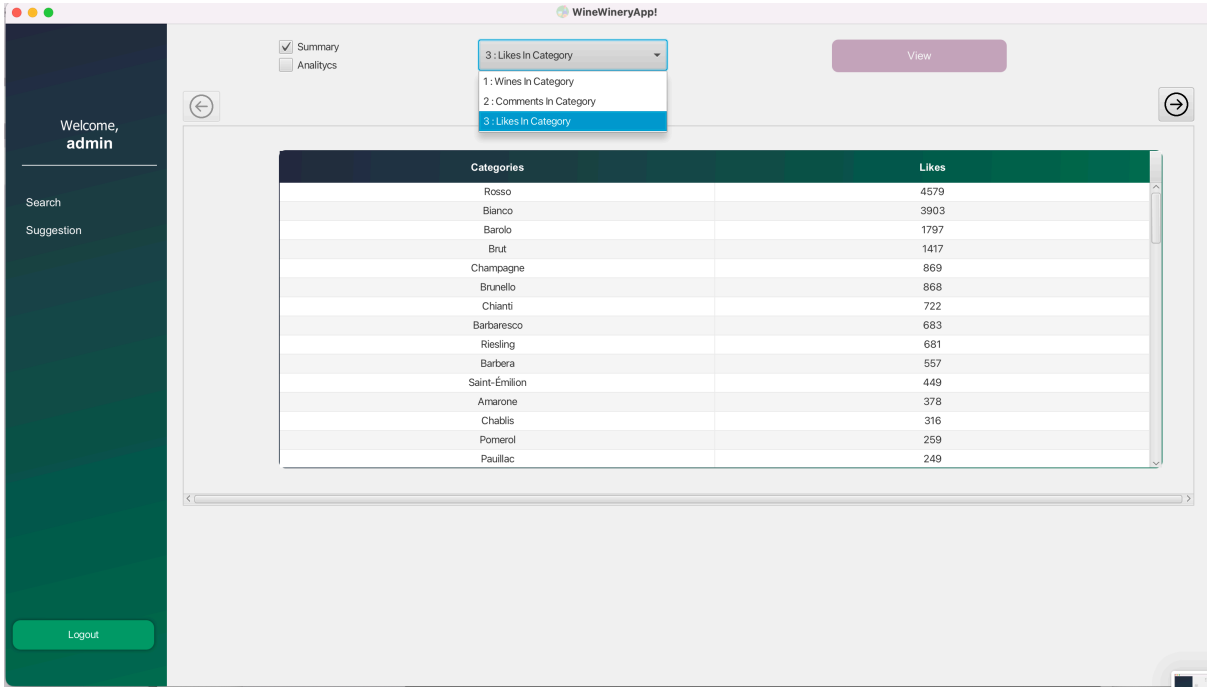


On the user profile page, the administrator via the provided buttons can delete the user or elect him as a moderator.

The admin cannot create a winery, view bottles or put like or follow.



Using the second button in the sidebar the admin can access the advanced search part, here he can see the analytics (like normal users) and instead of suggestions there is a summary section where he can see summaries of the data in the system.



The summary available are: