UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Master's Degree in Artificial Intelligence and Data Engineering

LARGE SCALE AND MULTI STRUCTURED DATABASES PROJECT

Itinera

GitHub Repository: https://github.com/MenchiniDev/Itinera

Students:
Nicolò Bacherotti
Lorenzo Menchini
Rossana Antonella Sacco

ACADEMIC YEAR 2024/2025

Indice

1 Introduction

Itinera is a sleek and user-friendly application designed for travelers and locals alike, offering reviews of hotels, museums, restaurants, and monuments across Europe. With *Itinera*, users can:

- Explore reviews of landmarks, dining spots, and accommodations in various European cities.
- Select specific locations to browse detailed feedback and experiences shared by other users.
- Join community spaces dedicated to different cities, enabling users to connect and exchange tips.

Whether you are planning your next European adventure or simply exploring your local scene, *Itinera* serves as your guide to discovering the best that Europe has to offer. Share your experiences, connect with like-minded people and make every journey memorable with *Itinera*!

2 Design

2.1 Actors

In Itinera there are three main actors

- **Unregitered user**
- **Registered user**
- **Administrator**

An unregistered user will have limited access to the website's features, restricted primarily to creating an account. A registered user, on the other hand, will have full access to all functionalities, including searching for points of interest in a city, reading and writing reviews, and interacting with other travelers through communities by creating posts and comments.

The Admin has the responsibility of creating new communities and monitoring user interactions on the platform. They can review content flagged by users and take appropriate actions to ensure a healthy and respectful environment.

2.2 Main Mockups

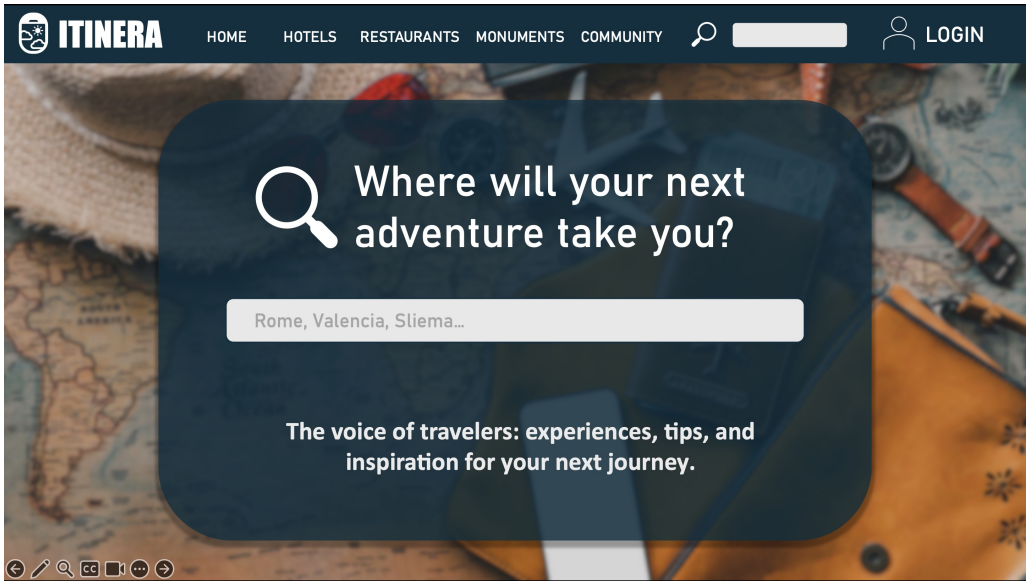


Figure 1: Home Page

Figure 1 shows the main page. Through this page a logged in user can search for a city and start browsing the site. If a user is not registered or logged in, they can access the login area.

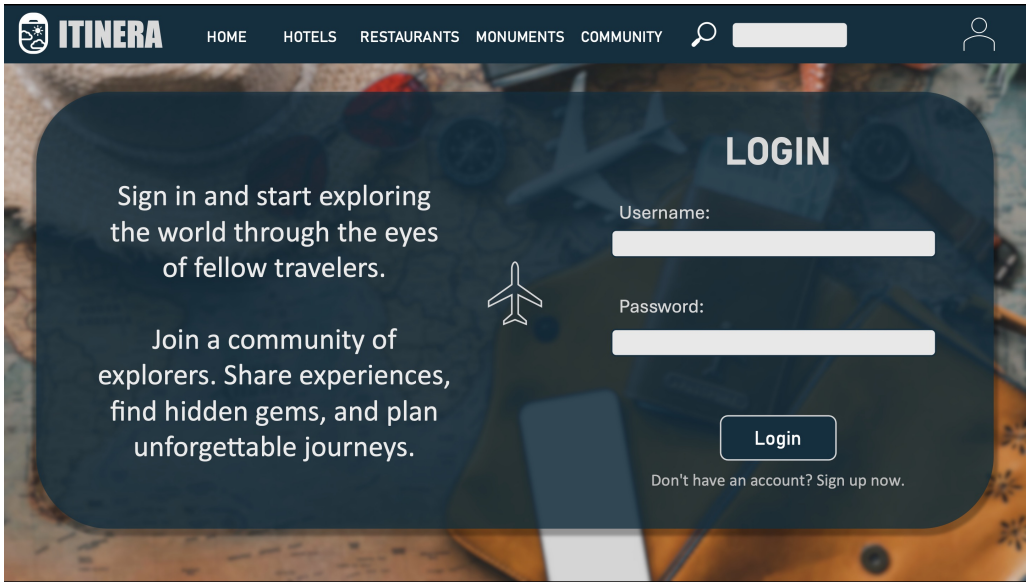


Figure 2: Login Page

Figure 2 shows the login page where a registered user can log in to their account or an unregistered user can sign up.

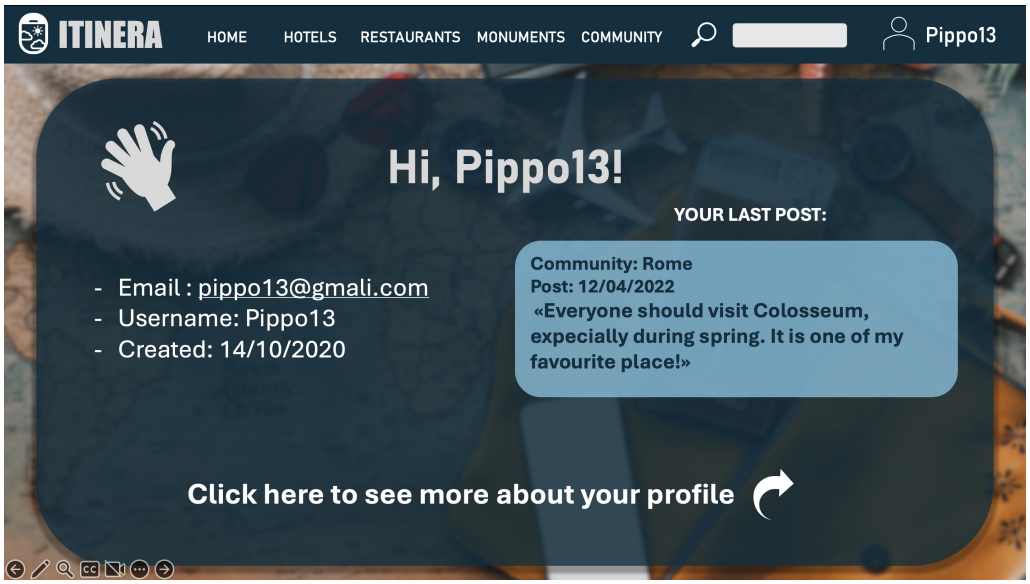


Figure 3: Logged User Profile.

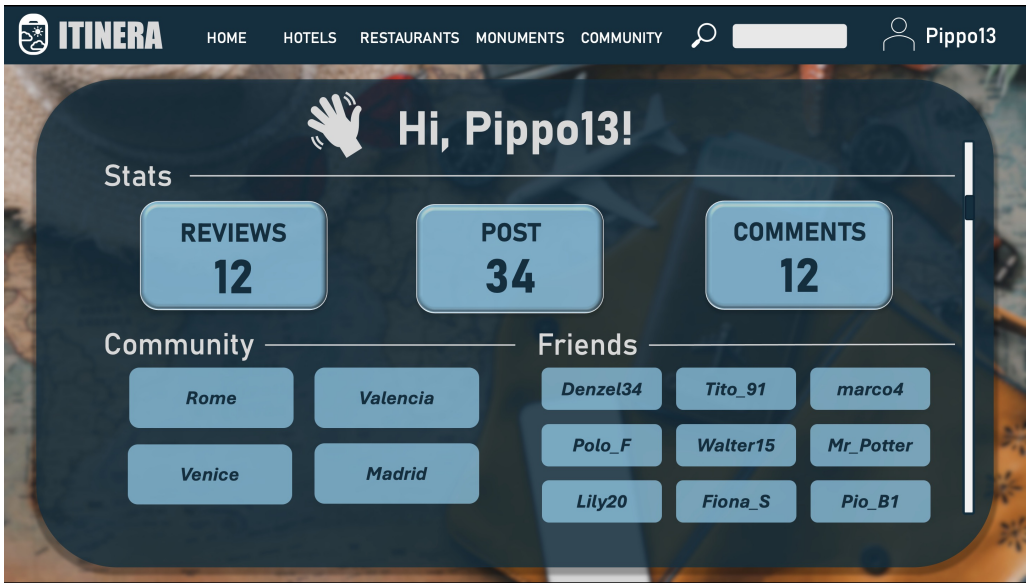


Figure 4: Logged User Profile.

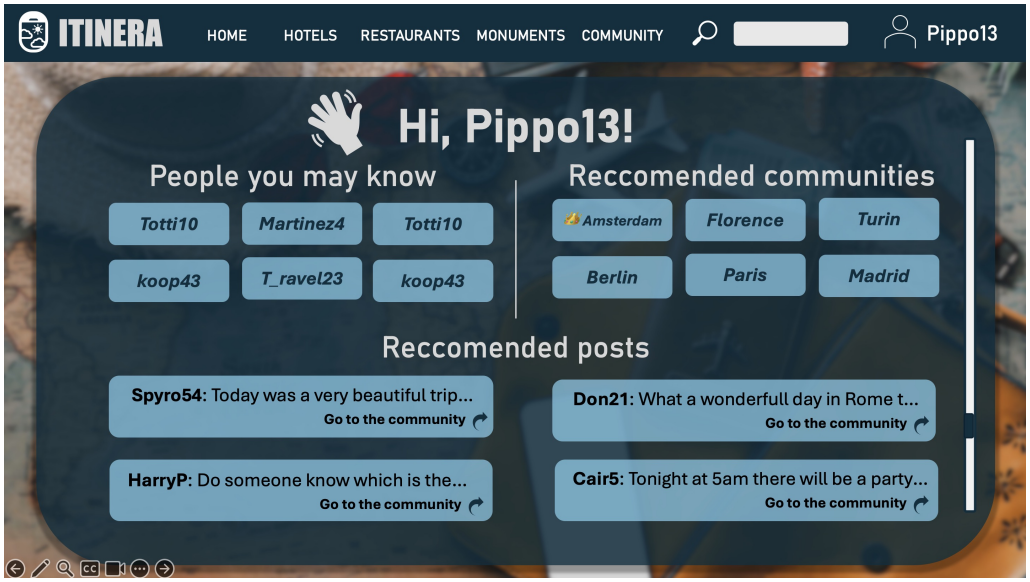


Figure 5: Logged User Profile.

Figure 3, *Figure 4* and *Figure 5* show what a user sees when they visit their profile:*Figure 3* shows the main page with some information about the user. If the user wants, he can decide to go more specifically and look at some statistics about himself. Specifically, *Figure 4* shows some statistics on the user's activities, the communities he has joined and the users he follows. *Figure 5* shows suggestions for the user that can further improve their experience on the site.

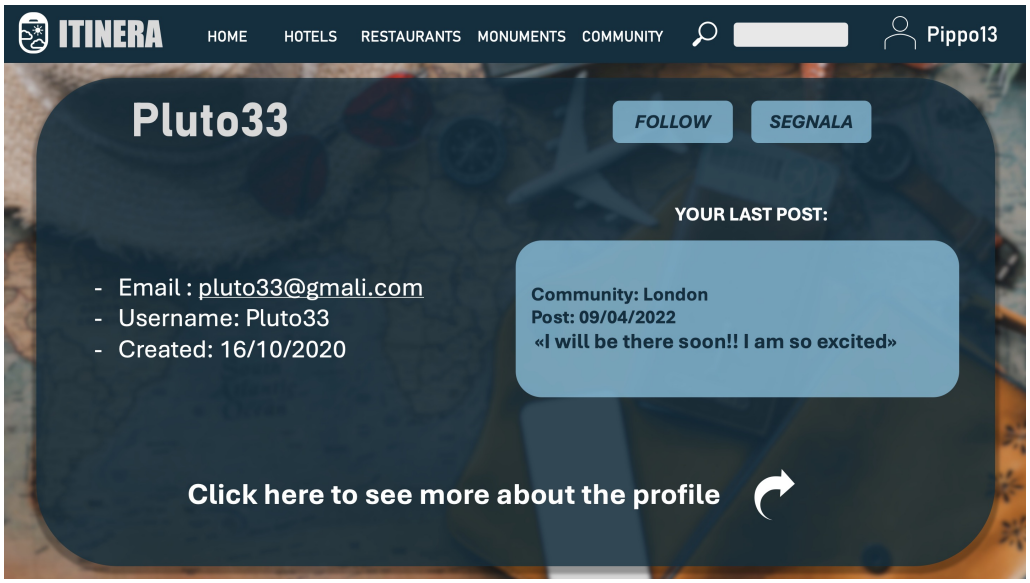


Figure 6: User Profile.

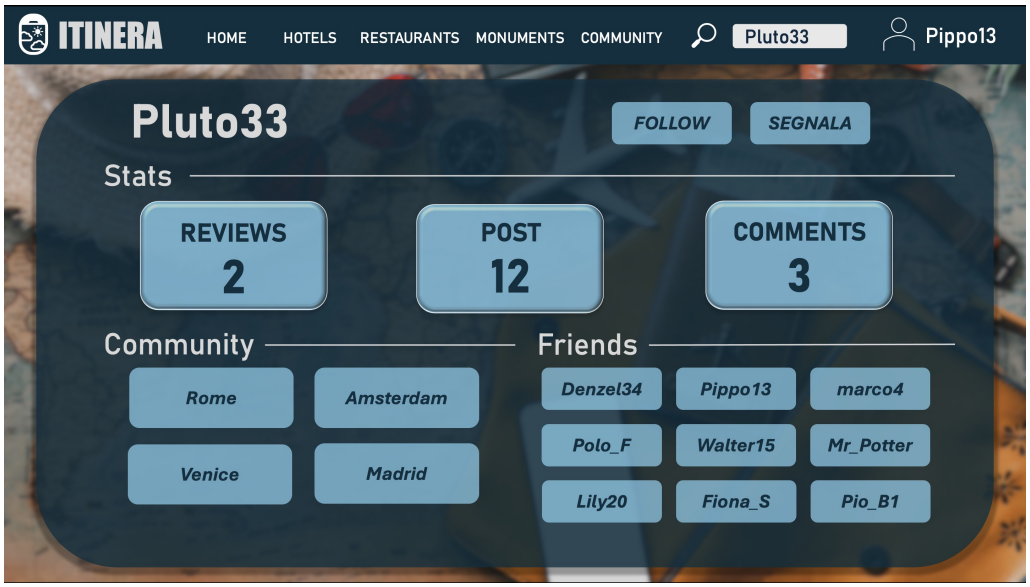


Figure 7: User Profile.

Figure 6 and *Figure 7* show what a user sees when they visit other user's profile. The structure is the same as previously shown, with the addition of the possibility to follow the user or report him.

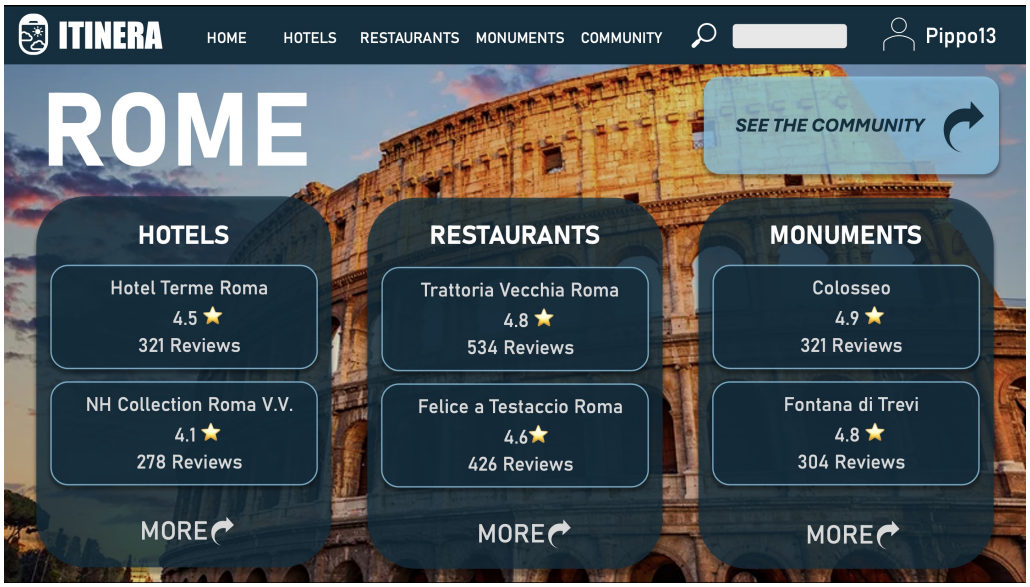


Figure 8: City browse page.

Figure 8 is what is shown once a user has searched for a city. A preview of the places of greatest interest is shown divided by category, which he can choose to learn more about if he is interested. He will also have the opportunity to visit the community page relating to the city searched for.

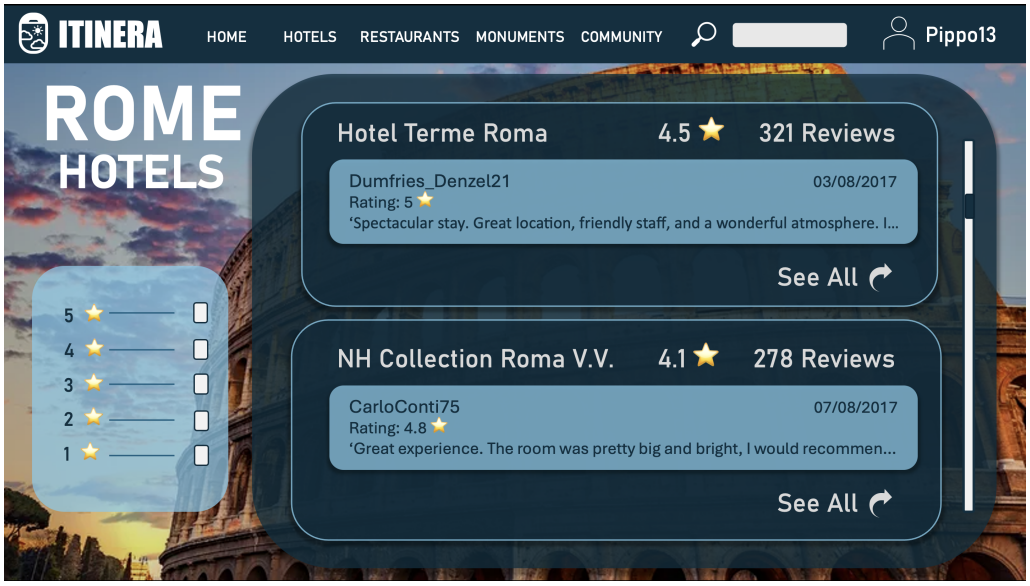


Figure 9: Category browse page.

Figure 9 illustrates what happens when a user selects a category—in this example, hotels. The user is presented with previews of various hotels along with their associated reviews. Additionally, a ratings filter is available, allowing the user to refine their search based on specific preferences

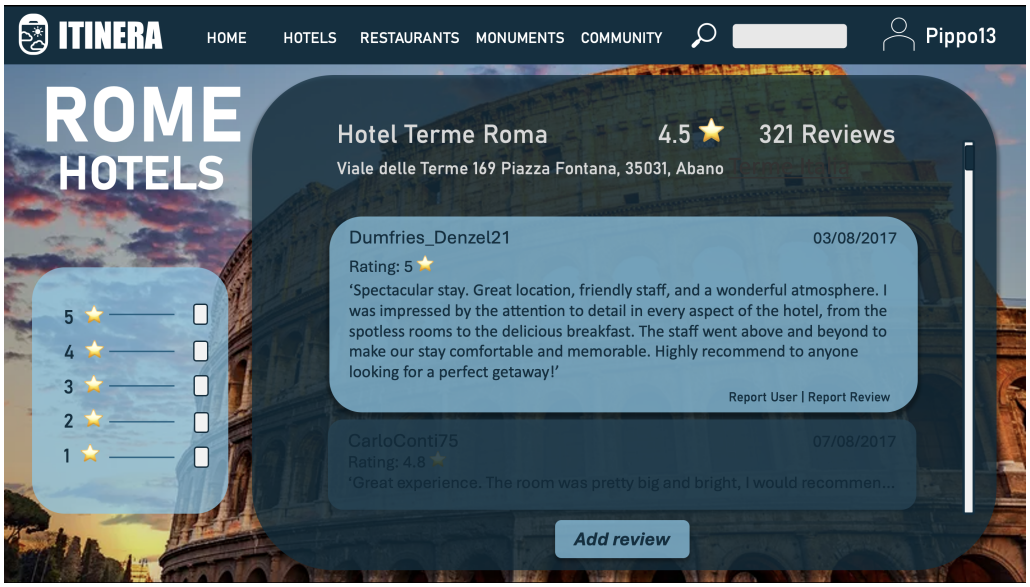


Figure 10: Reviews about a specific place.

Figure 10 illustrates what happens when a user selects a specific place. Detailed information about the location, along with its associated reviews, is displayed. Additionally, the user can filter the reviews using a star-rating filter to view feedback based on the given ratings.

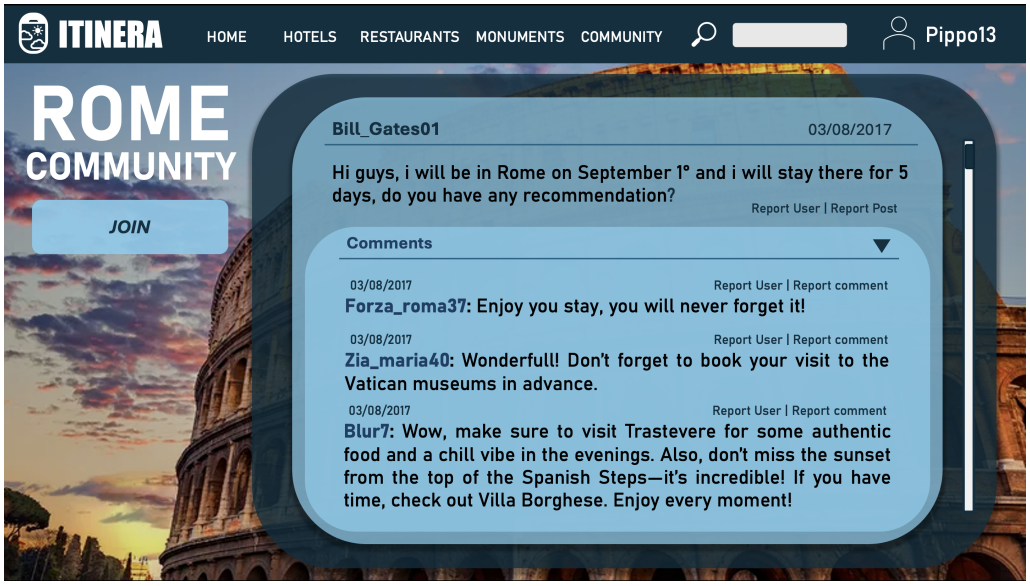


Figure 11: Community preview.

Figure 11 shows the community page as it appears to a user who is not a member. The page displays a preview of a post, but the user does not have the option to comment or create their own post.

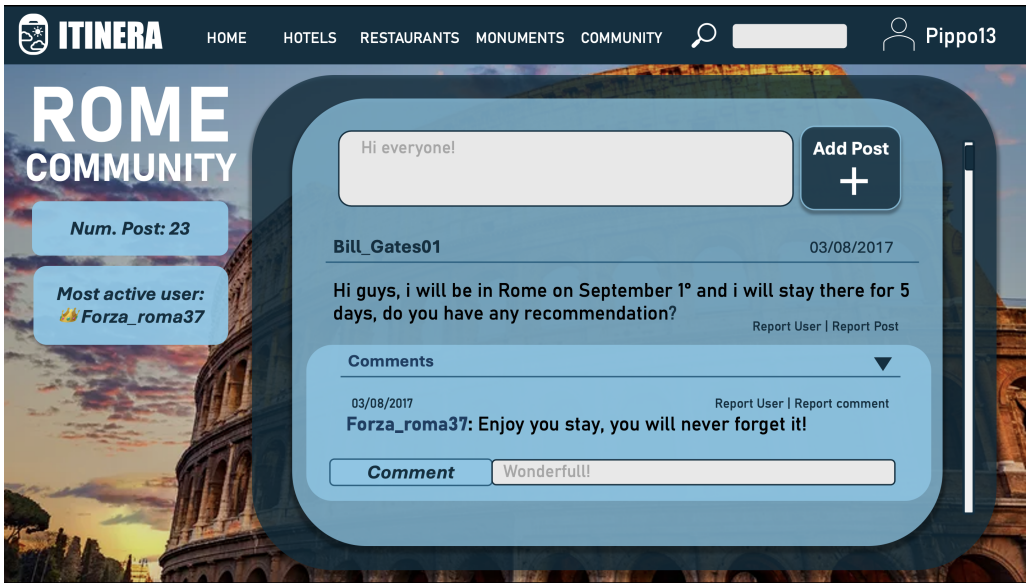


Figure 12: Community page for joined users.

Figure 12 shows the community page as it appears to a user who is a member. The user can view all posts, comment on them, and create their own posts.

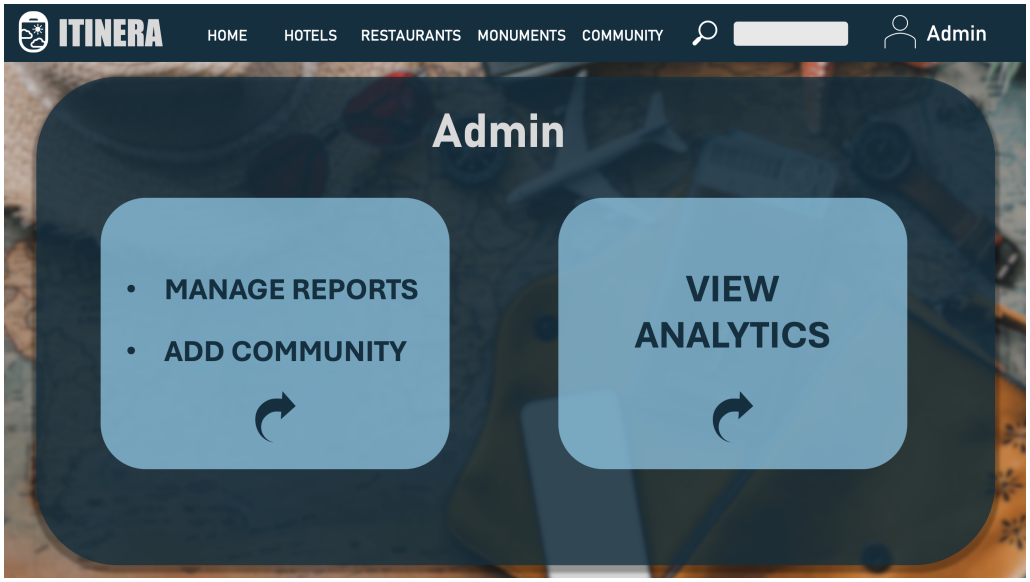


Figure 13: Admins home page.

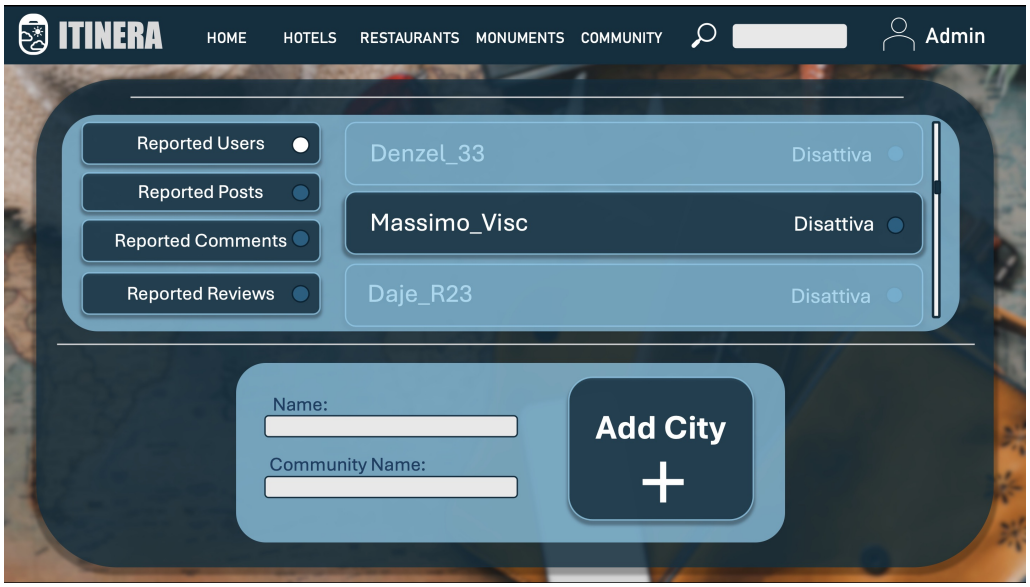


Figure 14: Admins administration page.

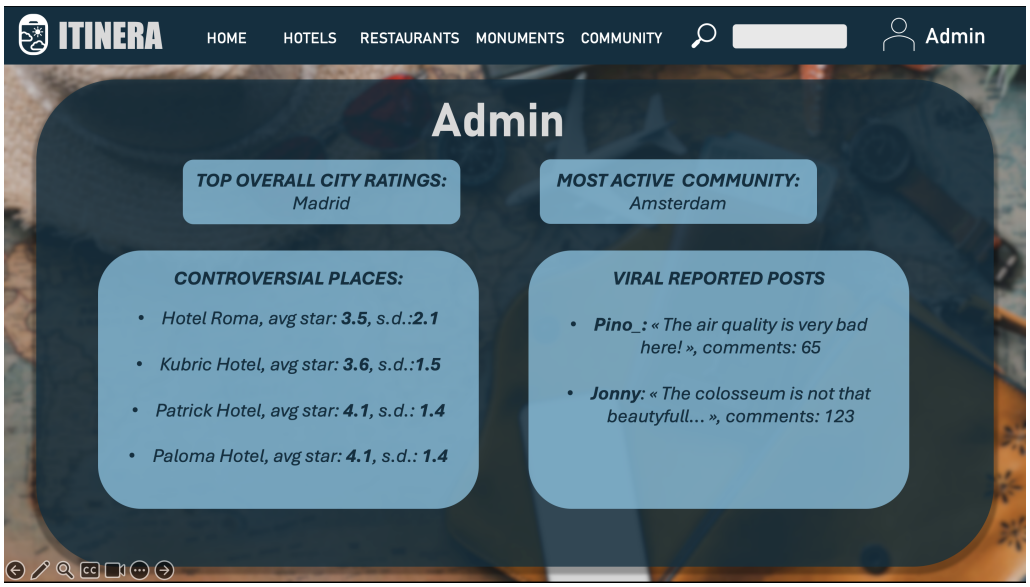


Figure 15: Admins Analitycs page.

Figure 13, *Figure 14* and *Figure 15* showcase the pages dedicated to the admin. These pages allow the admin to review and manage user reports, add new cities and communities, and view insightful statistics about the application.

2.3 Requirements

2.3.1 Functional Requirements

Unregistered user

1. The system must allow an unregistered user to register as a regular user of the application.
2. The system must allow an unregistered user to register as admin of the application

Registered user

1. The system must allow a registered user to log in the application.
2. The system must allow a registered user to see how many reviews other registered users have published.
3. The system must allow a registered user to find other registered users by username.
4. The system must allow a registered user to report other registered users by username.
5. The system must allow a registered user to see which is the last post of others registered users by username.
6. The system must allow a registered user to see which communities he joined.
7. The system must allow a registered user to see which communities another regular user has joined by username.
8. The system must allow a registered user to follow another registered user.
9. The system must allow a registered user to unfollow another registered user.
10. The system must allow a registered user to see all the users that he follows.
11. The system must allow a registered user to see all the followers of another registered user by username.
12. The system must allow a registered user to look for suggestions like "people he/she might know" or "recommended communities/post for you".
13. The system must allow a registered user to add a review to a place.
14. The system must allow a registered user to see all of the review a place has by id of the place.
15. The system must allow a registered user to report a review by id.
16. The system must allow a registered user to add a post on a community.
17. The system must allow a registered user to report a post by id.
18. The system must allow a registered user to report a comment by id.
19. The system must allow a registered user to add a comment to a post.
20. The system must allow a registered user to browse places by city, category and rating.
21. The system must allow a registered user to browse all the places in a particular city.
22. The system must allow a registered user to browse all the communities.
23. The system must allow a registered user to join a community.
24. The system must allow a registered user to see all the posts and comments of a community (if not joined to the community just 2 summary posts).
25. The system must allow a registered user to leave a community.
26. The system must allow a registered user to see the most active users of a community.
27. The system must allow a registered user to see the most active community.
28. The system must allow a registered user to see the number of posts in a community

Admin

1. The system must allow an admin to retrieve a list of all the users of the application.
2. The system must allow an admin to see all the reported users.

3. The system must allow an admin to ban an user from the application (the banned user can no longer log in).
4. The system must allow an admin to delete an user account from the application.
5. The system must allow an admin to find the top 10 active users on the application.
6. The system must allow an admin to delete a review by id.
7. The system must allow an admin to see all the reported reviews.
8. The system must allow an admin to see the most controversial places in the application.
9. The system must allow an admin to delete a post by id.
10. The system must allow an admin to delete a comment by id.
11. The system must allow an admin to see all the reported posts.
12. The system must allow an admin to see all the reported comments.
13. The system must allow an admin to see the most controversial posts.
14. The system must allow an admin to create a new place on the app.
15. The system must allow an admin to create a new community.
16. The system must allow an admin to delete a community.

2.3.2 Non Functional Requirements

1. The system must be capable of returning to the user quick responses.
2. The system should be able to scale to accommodate an increase in data traffic
3. The system must be developed using Object-Oriented Programming languages
4. The system must use an authentication protocol to secure users' accesses.
5. The system must encrypt users' passwords.

2.4 UML Class Diagram

2.5 Data Models

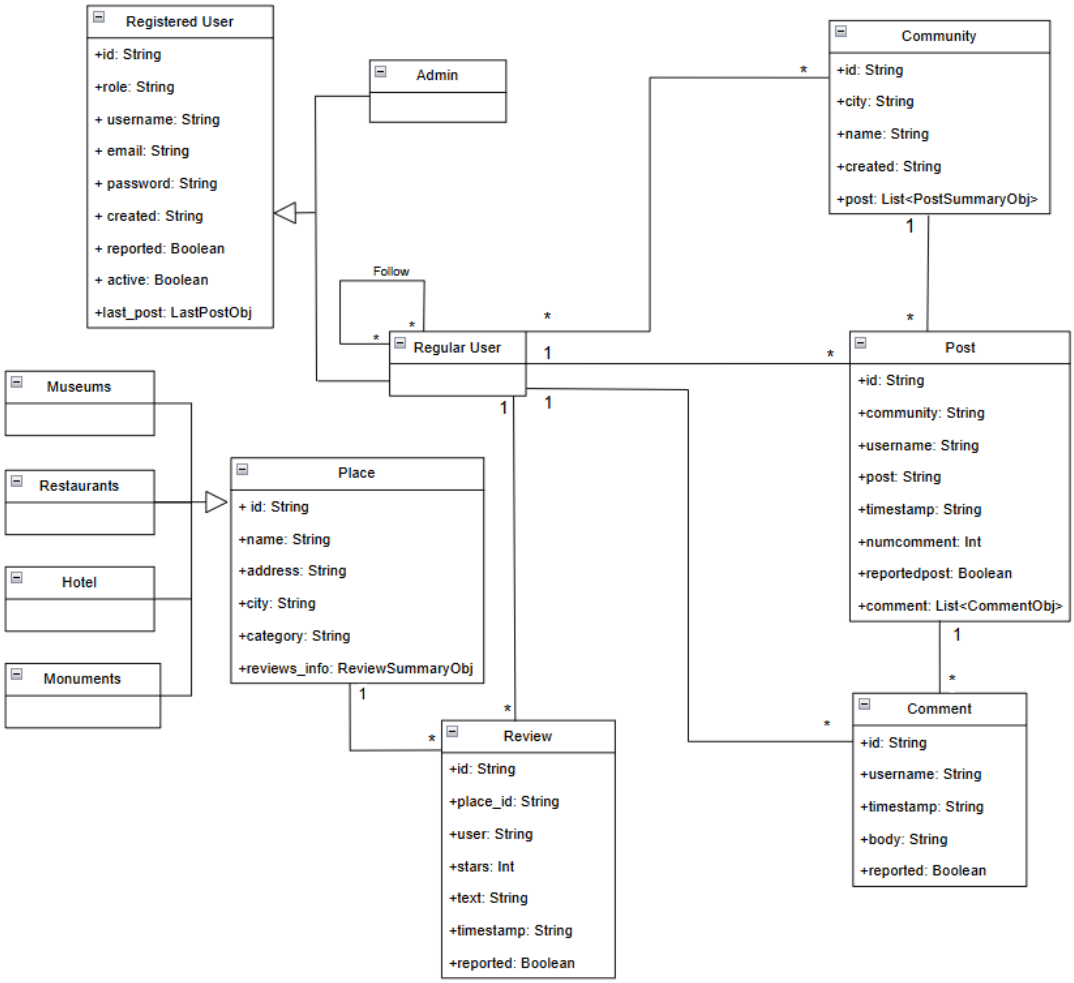
2.5.1 Document DB Structure

Our Document DB presents 5 different collections to store our data:

- *Users*
- *Community*
- *Places*
- *Post*
- *Reviews*

The **Users** collection stores the documents related to registered users and administrators. It stores the information provided at the moment of the registration and other information such as:

- **id**: the unique id of the user
- **username**: the username chosen at the moment of the registration, which also will be unique, in our application two users can not have the same username field.



- **email**: the email chosen
- **password**: the password chosen
- **created**: the timestamp when the account was created.
- **active**: a Boolean attribute: represents if the user is actually "active" on the application. If it is "false", the user is banned and can no longer perform any action on the application.
- **role**: the specific role of the user, the field checked when trying to perform any specific operation
- **reported**: a second Boolean attribute: a use can decide to report another user for some reason, in that case the admin can see all of the reported users and decide whether to ban some of them or not.

The **Places** collection stores the documents related to all those places that users can review. Each place has the following attributes:

- **id**: the unique id of the place
- **name**: the name of the place
- **address**: the address of the place
- **city**: the city in which the place is located
- **category**: our places can have different categories (only one for each place) such as:

  - *Hotel*
  - *Restaurant*
  - *Museum*
  - *Monument*

- reviews_info: an embedded object containing a summary of the reviews about the place stored in the DB. Its fields are :
  - overall_rating: indicates the average score of the place, calculated from the value of the field *"stars"* of those reviews referring to it.
  - tot_rev_number: the total number of reviews in the *Reviews* collection referring to the specific place.

The **Reviews** collection stores the documents related to all the reviews of places. We opted for this kind of implementation because in a real context, where the amount od data can increase significantly, a separate collection would be better to handle them and for the scalability of the database.

- **id**: the unique id of the review
- **place_id**: the id of the place which the review refers to
- **text**: the text the user chose to write
- **timestamp**: the timestamp when the review was created
- **user**: the username of the user who wrote the review
- **stars**: the integer number representing the evaluation of the review (from 1 to 5)
- **reported**: a boolean attribute indicating if the review is reported. For example a user can report a review if he thinks that the content of the review is inappropriate.

The **Community** collection stores the documents related to the communities of all the cities we consider. Each city has its own community and it's unique.

- **city**: the city to which the community refers.
- **name**: the name of the specific community, which can be also the same as the name of city itself
- **created**: the timestamp at the moment of the community's creation
- **post**: an embedded array containing the two most recent posts in the community

The **Post** collection stores the documents related to

- **id**: the unique id of the post
- **community**: the community in which the post is written
- **username**: the user who wrote the post
- **post**: the content of the post
- **timestamp**: the timestamp at the moment of the creation of the post
- **reportedpost**: a boolean field indicates if the post is reported
- **numcomment**: number of comment of the post, basically how many people "answer" to the post. A comment has less attributes than the "post" entity
- comment: in an embedded array containing all the comments that a post has. The **comment** entity has the following attributes:

- **id**: the unique id of the comment
- **body**: the content of the comment
- **username**: the user who wrote the comment
- **timestamp**: the timestamp at the moment of the creation of the comment

Contrary to what was done with the review collection, we preferred to store comments information in an embedded arrays in the Post collection. We did this so we could have not too many collections to work with, and also thinking about a real context, is rare for a message in a community to have a large number of comments responding to it.

2.5.2 Document Examples


```
{
  _id: '123e4567-e89b-12d3-a456-426614174001',
  username: 'Nico',
  email: 'NicoB@gmail.com',
  password: '$2a$10$GQQi/NpwjoSvnWovsc2TxefczPdWVENtsgKVM1cAJRvNe6G03xtBq',
  role: 'ADMIN',
  created: '2025-01-25T09:11:16.331507400',
  active: true,
  reported: false,
  _class: 'com.unipi.ItineraJava.model.User'
}
```


Figure 16: Example of user document.


```
{
  _id: '65036',
  text: ' the building meeting rooms modern style of my room park',
  timestamp: '2021-08-18T13:13:00.000000',
  user: '2dmaxxx',
  stars: 4,
  reported: false,
  place_id: '1250'
}
```


Figure 17: Example of review document.

```
{
  _id: '522200ba-3cc3-4d8f-b980-9c022483fb0e',
  community: 'Podgorica',
  username: 'edgalimov',
  post: 'Anyone up for playing chess offline?',
  timestamp: '2024-11-23T10:29:18Z',
  numcomment: 1,
  reportedpost: false,
  comment: [
    {
      reportedcomment: false,
      _id: 'd319f28b-a334-46f1-a623-325b3b6cd3d3',
      body: 'I sent you a PM :)',
      username: 'crazzywak',
      timestamp: '2024-11-23T10:29:18Z'
    }
  ]
}
```


Figure 18: Example of post document.


```
{
  _id: '1250',
  name: 'Hotel Arena',
  address: ' s Gravesandestraat 55 Oost 1092 AA Amsterdam Netherlands',
  city: 'Amsterdam',
  category: 'Hotel',
  reviews_info: {
    overall_rating: 3.96,
    tot_rev_number: 405
  }
}
```


Figure 19: Example of place document.


```
{
  _id: ObjectId('6796303dc9b98870aea09b22'),
  city: 'Amsterdam',
  name: 'Amsterdam',
  created: '2024-10-25T23:08:55Z',
  post: [
    {
      _id: '56000966-b7d5-4f91-82a0-6c3d504e9835',
      text: 'Kerstviering bezoeken in Amsterdam? Hier ben je welkom op kerstavond',
      user: 'dullestfranchise',
      timestamp: '2024-10-26T00:08:55Z'
    },
    {
      _id: 'c01b2b5d-78f7-4065-92dd-5c4cda9b37b5',
      text: 'Dancing in the Oosterpark',
      user: 'HetGewildeWesten',
      timestamp: '2024-12-22T08:26:29Z'
    }
  ]
}
```


Figure 20: Example of community document.

2.6 Graph DB Structure

The GraphDB was chosen as the most suitable solution to manage the dynamics of a social platform. This database is used to handle and track various relationships, such as user follow connections, the communities a user has joined, and the posts and comments they have created.

Through these relationships, it becomes possible to provide personalized suggestions to users, such as recommending posts, people, or communities that align with their interests, enhancing their experience within the app. Additionally, the GraphDB enables monitoring of user activity and extraction of valuable insights, such as identifying the most active user within a specific community.

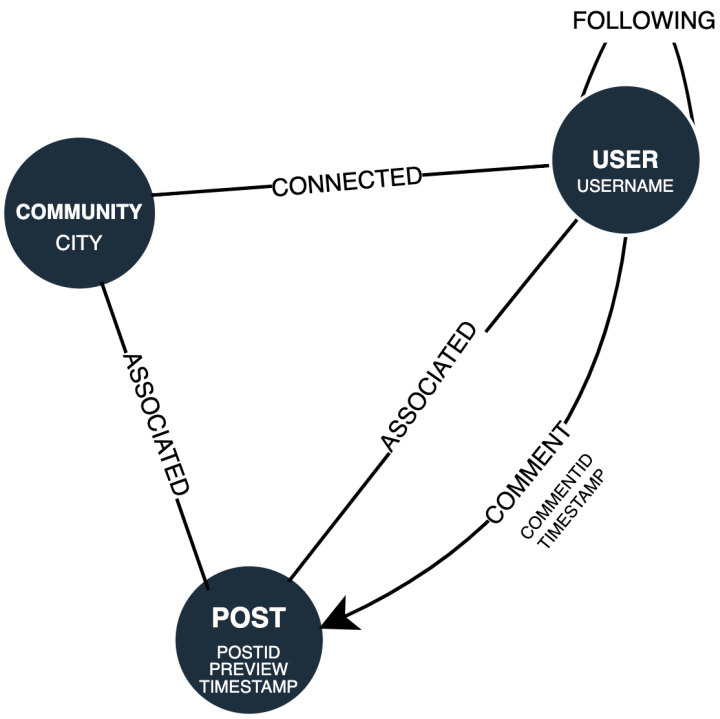


Figure 21: GraphDB Structure.

To create the GraphDB, a script was developed that processes the JSON documents related to posts. These documents contain all the necessary information to construct the GraphDB in Neo4j, as defined in *Figure 21*:, while ensuring there are no duplicates or inconsistencies.

A user is considered as "joined" to a community if they have written a post or comment within it.The "following" relationship between users was generated randomly, with the number of followed users ranging from 1 (since all users have already performed at least one activity on the site, they are presumed to be minimally active—new users start with zero followed users) to 25.

2.6.1 GraphDB Nodes and Relationship

In the graphDB there are three kind of **nodes** as shown in *Figure 21*:

- **User Node**: contains a user's unique username.
- **Community Node**: contains the city relating to the community through which it is possible to identify the specific community.
- **Post Node**: contains the postId, a preview of the post and the timestamp.

For posts, it was decided to retain only a preview to be used in the recommended posts feature. If a user is interested, they can search for the specific post and view its full content, provided they have joined the corresponding community.

The nodes are connected to each other through **relationships**:

- **:User - [:FOLLOWING] - :User**

This edge identifies the follow relationship of a user to another user.

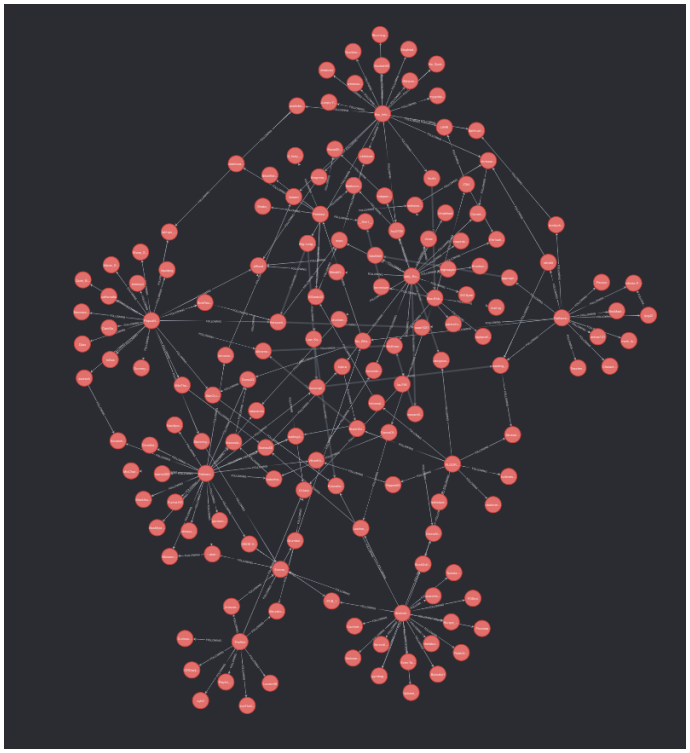


Figure 22: Following Relationship example

- **:Post - [:ASSOCIATED] - :Community**
- **:User - [:CONNECTED] - :Community**
- **:User - [:ASSOCIATED] - :Post**
- **:User - [:COMMENTED] - :Post**

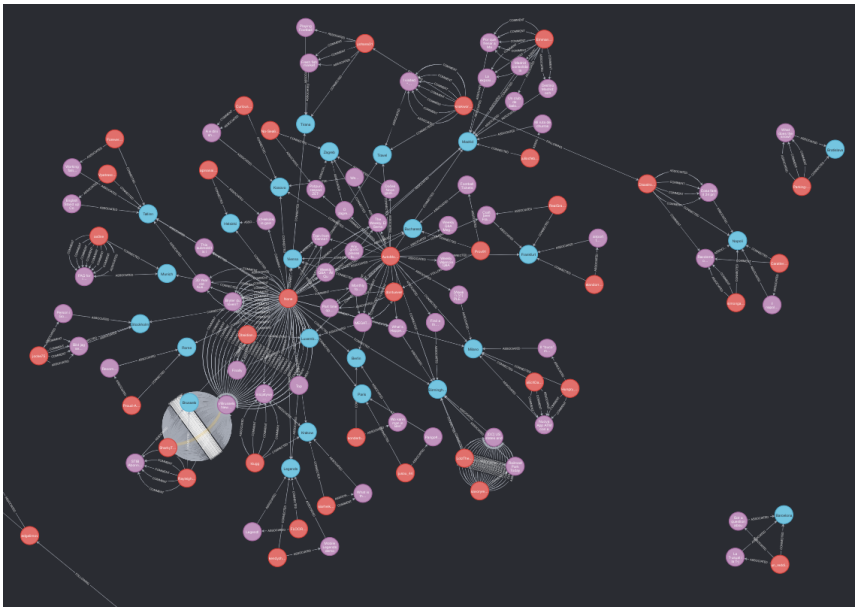


Figure 23: Relationship example

2.7 Distributed Database Design

Considering the requirements and the intrinsic nature of the app as a social network, we decided that implementing the app according to the AP characteristics of the CAP theorem was the most suitable choice. Maintaining high availability and partition tolerance was prioritized over strict consistency, resulting in an eventually consistent model.

This approach ensures that the application remains available to users even in the event of node failures or network partitions. To achieve this, the databases are replicated across multiple nodes, ensuring availability even if some nodes fail. While this replication may cause temporary inconsistencies, it is an acceptable trade-off in this context to provide persistent availability to the users.

2.7.1 Operation Volumes

Below is a table containing estimates of access to the main operations present in the app:

Action	%	# Daily	MongoDB	Neo4j	Daily Total
Register as regular user	0.01%	10	1	0	10
Register as admin	0.01%	10	1	0	10
Login user	100%	100000	1	0	100000
See reviews of others	5%	5000	1	0	5000
Find user by username	3%	3000	0	1	3000
Report a user	0.1%	100	1	0	100
See last post of user	3%	3000	1	0	3000
See joined communities	5%	5000	0	1	5000
See joined communities of others	5%	5000	0	1	5000
Follow a user	3%	3000	0	1	3000
Unfollow a user	2%	2000	0	1	2000
See users followed	10%	10000	0	1	10000
See followers of user	5%	5000	0	1	5000
Suggestions (people/community)	10%	10000	0	1	10000
Add review to a place	1%	1000	1	0	1000
See reviews of a place	2%	2000	1	0	2000
Report a review	0.01%	10	1	0	10
Add post to community	1%	1000	3	1	4000
Report a post	0.1%	100	1	0	100
Report a comment	0.1%	100	1	0	100
Add comment to a post	0.9%	900	1	1	1800
Browse places	15%	15000	1	0	15000
Browse places by city	5%	5000	1	0	5000
Browse communities	10%	10000	1	0	10000
Join a community	3%	3000	0	1	3000

Figure 24: Tabella dei Volumi stimati su 100.000 login giornalieri

2.7.2 Replicas

With three virtual machines available, the replicas were managed as follows:

- **Primary Node (Ip: 10.1.1.25) (Priority:5):** This machine serves as the primary server for MongoDB, managing all incoming requests. Its dedicated role ensures optimal efficiency and performance for write operations. The higher priority assigned to this node guarantees that it remains the primary server as long as it is functional.
- **Secondary Node (Ips: 10.1.1.24)(Priority:1):** This machine function as secondary nodes within the replica set, maintaining copies of the primary server's MongoDB data. It provide redundancy and ensure data durability in the event of a primary node failure.
- **Secondary Node (Ips: 10.1.1.23)(Priority:1):** This machine operates as a secondary node, similar to those described earlier, but with the addition of hosting a replica of the GraphDB. A secondary node was chosen for the **GraphDB** to avoid overloading the primary node and to balance the workload effectively.

In conclusion, we will have three replicas of MongoDB and one replica of the GraphDB. This replica set configuration reflects the choise to create a robust, scalable, and highly available system. By effectively distributing the workload and prioritizing node roles, we ensure a resilient architecture that can handle varying demands and recover swiftly from potential disruptions.

The replication operations were set as follows:

- **Read Operation:** Considering the high volume expected in our system and the need to provide fast responses to user queries, we configured the MongoDB replica set to use a **"nearest" read preference**. This setting allows data to be retrieved from the replica with the lowest latency, optimizing response times.

  By distributing read operations across secondary nodes, this approach not only alleviates the load on the primary node, preventing potential bottlenecks, but also ensures a more balanced and efficient workload. Furthermore, leveraging secondary nodes for read operations enhances the system's scalability, enabling it to accommodate a growing user base and maintain smooth performance even during peak usage periods.

- **Write Operation:** Given the nature of our app as a social network, we expect a high volume of write operations. To handle this, we have chosen the **majority write concern** for managing replicas. This configuration ensures strong consistency by requiring write operations to be acknowledged by at least two replicas before being considered successful.

  When combined with the nearest read preference, this setup guarantees that users, even when reading from secondary nodes, access up-to-date data. Considering the limited number of replicas, the slight increase in write latency is negligible in this context and does not impact overall performance.

2.7.3 Sharding

Compared to vertical scaling — such as upgrading a single server by adding more RAM, storage, or CPUs—sharding is a more cost-effective option. Instead of relying on increasingly powerful hardware, it leverages horizontal scaling by distributing the workload across multiple machines.

When combined with replication, sharding ensures high system availability and fast response times. Replication provides redundancy by maintaining copies of the data, while sharding optimizes performance by directing queries to the appropriate shard. Together, these mechanisms create a robust and scalable system capable of supporting both current demands and future growth.

Considering the specific case of our application and evaluating various possibilities, we concluded that the most suitable sharding strategy involves selecting shard keys tailored to the specific use case of each collection, based on the implemented queries. For the **Post, Place, and Review collections, the document ID is used as the shard key**, as queries for these collections are primarily ID-based, ensuring efficient and direct data retrieval.

For the **User collection, the username is used as the shard key**, as it is unique and serves as the primary attribute for user-related queries, such as login or profile lookups. Similarly,

for the **Community collection, the city attribute is used as the shard key**, aligning with the application's logic where community-related queries often target specific cities.

This sharding strategy, combined with a partitioning algorithm based on hash functions, ensures an even distribution of data across the shards, minimizing the risk of workload imbalance. By aligning the shard key selection with the nature of queries for each collection, the system achieves efficient data routing, low-latency queries, and scalability to accommodate a growing dataset and user base.

Using a hash-based partitioning algorithm provides several advantages. It evenly distributes the data across the shards, preventing scenarios where certain shards become overloaded while others remain underutilized. This uniform distribution minimizes the risk of performance bottlenecks and ensures a balanced workload, even as the dataset grows.

In summary, this combination offers a highly efficient, balanced, and scalable sharding solution tailored to the needs of our application.

2.7.4 Databases Consistency

When using two different databases, like in our case, it is crucial to ensure proper consistency management, especially given the redundancy of data between them. Particular attention must be paid to operations involving additions or modifications that affect both databases.

In our case, all operations are first performed on Neo4j, as it has only one replica and is therefore more prone to failures, making it the priority for verification. If the operation on Neo4j succeeds, we proceed with the corresponding operations on MongoDB, which is supported by three replicas. However, if an error occurs on MongoDB, there will be no rollback on Neo4j, potentially causing an inconsistency. This is a trade-off we were willing to accept, given the low likelihood of all three MongoDB replicas failing simultaneously.

To implement this logic, we utilized the **@Retryable** functionality. This allows us to define a set number of retry attempts (three in our case) and a delay between each attempt in case of execution errors. This approach ensures that transient issues, such as temporary network failures, are handled gracefully, reducing the risk of persistent inconsistencies between the two databases.

3 Dataset and data retrieval

We choose to retrieve our data from different sources, processing and saving them in JSON files with the python language: For **usernames, posts and comments** data we used the **Reddit API** via the PROW library. The script attempts to locate subreddits for each city. If a subreddit is inaccessible or does not exist, it searches for related subreddits or defaults to a generic one. For each discovered subreddit, the script retrieves up to 10 popular posts, extracting details such as the post title, author, URL, and comments. All comments are expanded to capture detailed information, including the author, content, and timestamp. To create the collection of **users**, we used usernames extracted from post and comments retrieved from subreddits. Then, we generated fake information about each user, using the Faker library of python, such as email and password.

- **Posts volume:** 3.6 MB.
- **Communities volume:** 23 KB.
- **Users volume:** 1.7 MB.

For the **Reviews and Places** collections we needed many places of different categories like "restaurants", "hotels" or "museums". We used data found on datasets about reviews on the website **"Kaggle.com"**. Those datasets contain just reviews about places all around the world, also, the attributes of the reviews were different from one dataset to another, so we needed to bring them all into the same format.

We used a different dataset for each category of our places: "hotels", "museums", "monuments" and "restaurants".

Since every single link below contains only reviews, we create the collection Places taking the ones to which the review refers and created different places based on those. Also, given the discrepancy between the date formats across the various Kaggle datasets, as well as the mismatch with the format we wanted, every timestamp in the Reviews collection has been artificially generated.

- **Link to the dataset containing restaurants reviews, final volume: 75.4 MB**

We took the text and the stars of the reviews along with restaurant names for the Places collection. Then we generated a fake address as well to redistribute places to the european cities we considered.

- **Link to the dataset containing monuments reviews, final volume: 15.9 MB**

From this dataset we kept just the monuments located in the cities of our interest. Since there was no address in the dataset, for the collection of the places we had to manually insert the addresses.

- **Link to the dataset containing museums reviews, final volume: 2.9 MB**

The linked dataset contains reviews about the British Museum, we redistributed them among other museums of our choice and, also for this category of places, insert their address manually.

- **Link to the dataset containing hotel reviews, final volume: 38.5 MB**

This was the most complete dataset for reviews. From the address, which was stored as an information, we were able to filter the hotels based on their city, then we took one of the two reviews in the dataset (positive or negative one) based on the rating of the review itself.

To each review was assigned a user took from those who wrote posts and comments, we did not consider usernames in the kaggle datasets.

4 Implementation

The application is implemented using the Spring framework, also using the Controller-Service-Repository structure to make the code more readable, maintainable and scalable.

4.1 Spring

The project has been built using the Spring framework for the initialization and to establish the connection between the server and the databases. Another useful feature used of the Spring framework is the Spring Security token management to allow a more fluid login of the users into the system, handling the login via a JWT (JSON Web Token), to be included in any request's header when the user is logged.

4.2 Model

The models for Itinera are divided into two main types: the MongoDB models and the Neo4j ones.

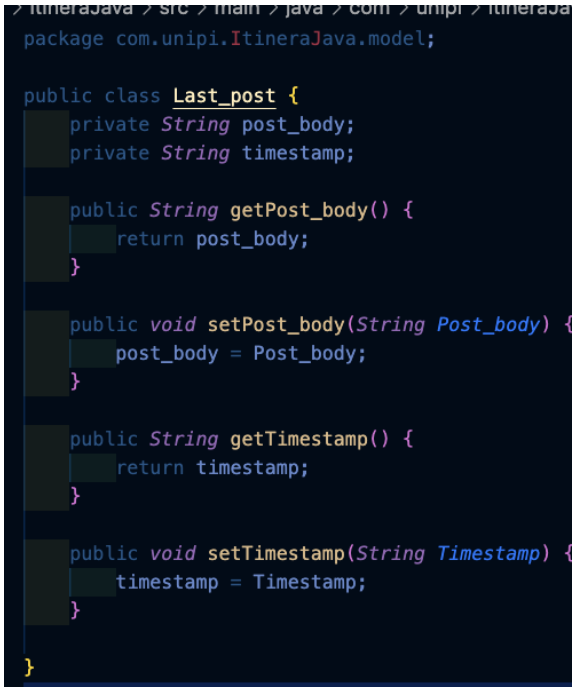
4.2.1 Mongo Models

```java
// USERS
@Document(collection = "Users")
@Component
@JsonIgnoreProperties({"jwtTokenProvider"})
public class User {
    private static JwtTokenProvider jwtTokenProvider;
    @Autowired
    public void setJwtTokenProvider(JwtTokenProvider provider) {
        jwtTokenProvider = provider;
    }

    @Id
    private String id;
    private String username;
    private String email;
    private String password;
    private String role; // "User" or "Admin"
    private String created;
    private boolean active;
    private boolean reported;
    private Last_post last_post;



    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }

    public void setEmail(String email) {
        this.email = email;
    }
    public String getEmail() {
        return email;
    }
    public void setCreated(String created) {
        this.created = created;
    }
    public String getCreated() {
        return created;
    }
    public void setActive(boolean active) {
        this.active = active;
    }
```

Figure 25: User Mongo Model 1

```
    public void setActive(boolean active) {
        this.active = active;
    }
    public boolean isActive() {
        return active;
    }
    public void setReported(boolean reported) {
        this.reported = reported;
    }
    public boolean isReported() {
        return reported;
    }

    public void setLastPost(Last_post last_post) {
        this.last_post = last_post;
    }
    public Last_post getLastPost() {
        return last_post;
    }

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }

    public static boolean isAdmin(String token) {
        try {
            String jwt = token.replace("Bearer ", "").trim();

            String username = JwtTokenProvider.getUsernameFromToken(jwt);
            System.out.println("Username: " + username);

            User user = UserService.findByUsername(username)
                    .orElseThrow(() -> new IllegalArgumentException("User not found: " + username));
            System.out.println("Role: " + user.getRole());

            if ("ADMIN".equals(user.getRole())) {
                System.out.println("L'utente è ADMIN");
                return true;
            } else {
                System.out.println("L'utente NON è ADMIN");
                return false;
            }

        } catch (Exception e) {
            System.out.println("L'user NON è ADMIN");
            e.printStackTrace();
            return false;
        }
    }
}
```

Figure 26: User Mongo Model 2



```
> ItineraJava > src > main > java > com > unipi > ItineraJa
package com.unipi.ItineraJava.model;

public class Last_post {
    private String post_body;
    private String timestamp;

    public String getPost_body() {
        return post_body;
    }

    public void setPost_body(String Post_body) {
        post_body = Post_body;
    }

    public String getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(String Timestamp) {
        timestamp = Timestamp;
    }

}
```

Figure 27: Last Post Mongo Model

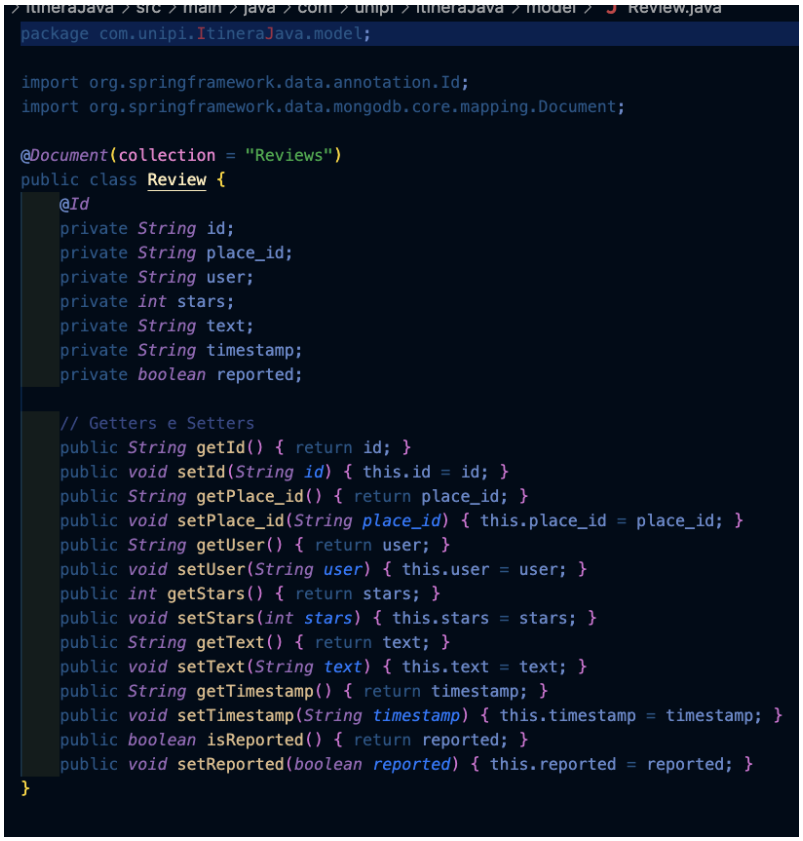
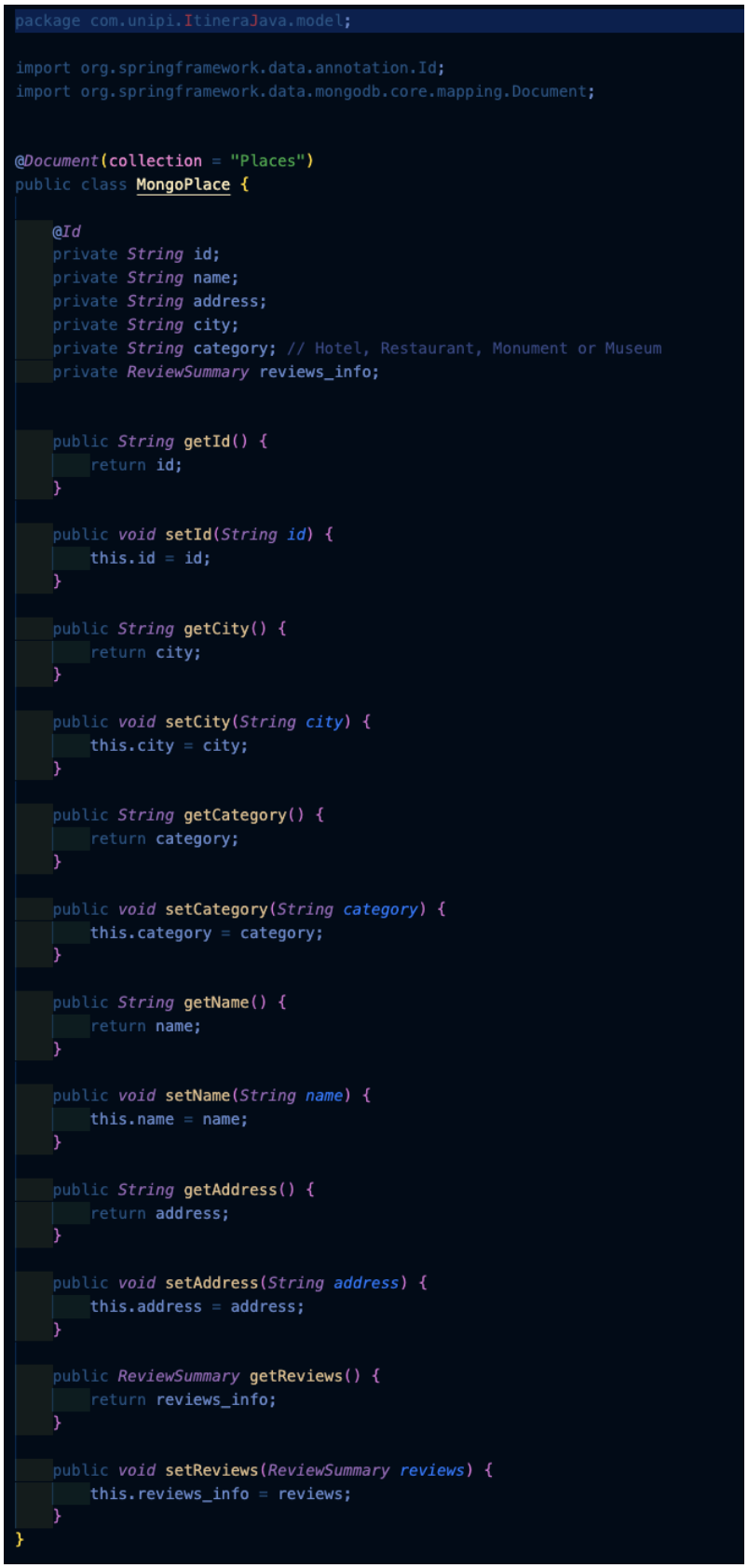


Figure 28: Review Mongo Model

Figure 29: Review Summary Mongo Model, embedded document



```java
package com.unipi.ItineraJava.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;


@Document(collection = "Places")
public class MongoPlace {

    @Id
    private String id;
    private String name;
    private String address;
    private String city;
    private String category; // Hotel, Restaurant, Monument or Museum
    private ReviewSummary reviews_info;


    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public ReviewSummary getReviews() {
        return reviews_info;
    }

    public void setReviews(ReviewSummary reviews) {
        this.reviews_info = reviews;
    }
}
```

Figure 30: Place Mongo Model

```java
package com.unipi.ItineraJava.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

@Document(collection = "Community")
public class MongoCommunity {
    @Id
    private String id;
    private String city;
    private String name;
    private String created;
    private List<PostSummary> post;

    // Getters and Setters
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCreated() {
        return created;
    }

    public void setCreated(String created) {
        this.created = String.valueOf(created);
    }

    public List<PostSummary> getPost() {
        return post;
    }

    public void setPost(List<PostSummary> post) {
        this.post = post;
    }

    public List<Post> getPosts() {
        return new ArrayList<Post>();
    }
}
```

Figure 31: Community Mongo Model

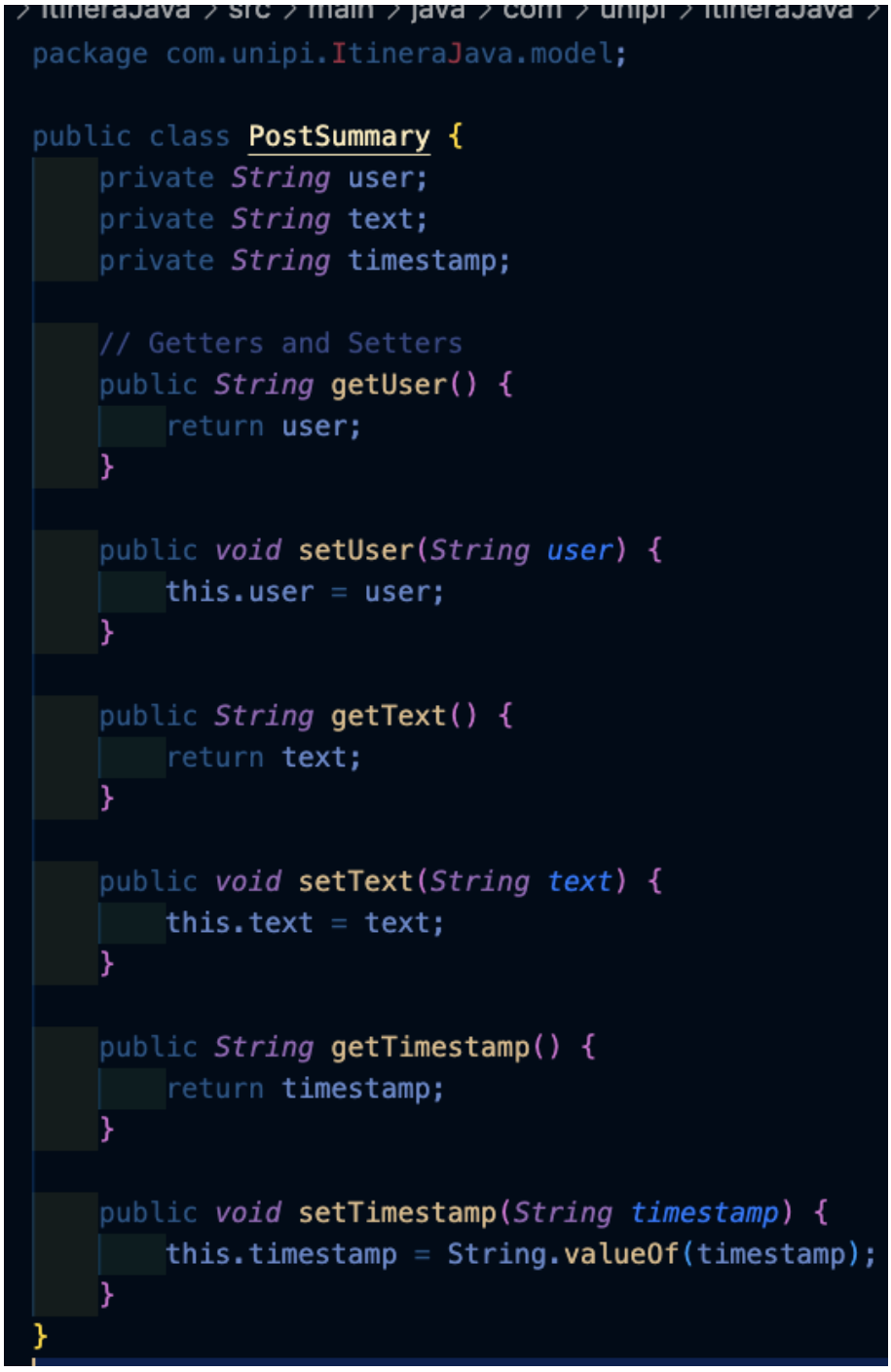


Figure 32: Post Summary Mongo Model, embedded document

```java
package com.unipi.ItineraJava.model;


import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.time.LocalDateTime;
import java.util.Date;
import java.util.List;

// POST
@Document(collection = "Post")
public class Post {

    @Id
    private String _id;

    private String community;
    private String username;
    private String post;
    private String timestamp;
    private int numcomment;
    private boolean reportedpost;
    private List<Comment> comment;

    public Post() {}

    public String getId() {
        return _id;
    }

    public void setId(String id) {
        this._id = id;
    }

    public String getCommunity() {
        return community;
    }
    public void setCommunity(String community) {
        this.community = community;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPost() {
        return post;
    }
    public void setPost(String post) {
        this.post = post;
    }
    public String getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }
    public int getNum_comment() {
        return numcomment;
    }
    public void setNum_comment(int num_comment) {
        this.numcomment = num_comment;
    }
    public boolean isReported_post() {
        return reportedpost;
    }
    public void setReported_post(boolean reported_post) {
        this.reportedpost = reported_post;
    }
    public List<Comment> getComment() {
        return comment;
    }
    public void setComment(List<Comment> commenti) {
        this.comment = commenti;
    }
```

Figure 33: Post Mongo Model

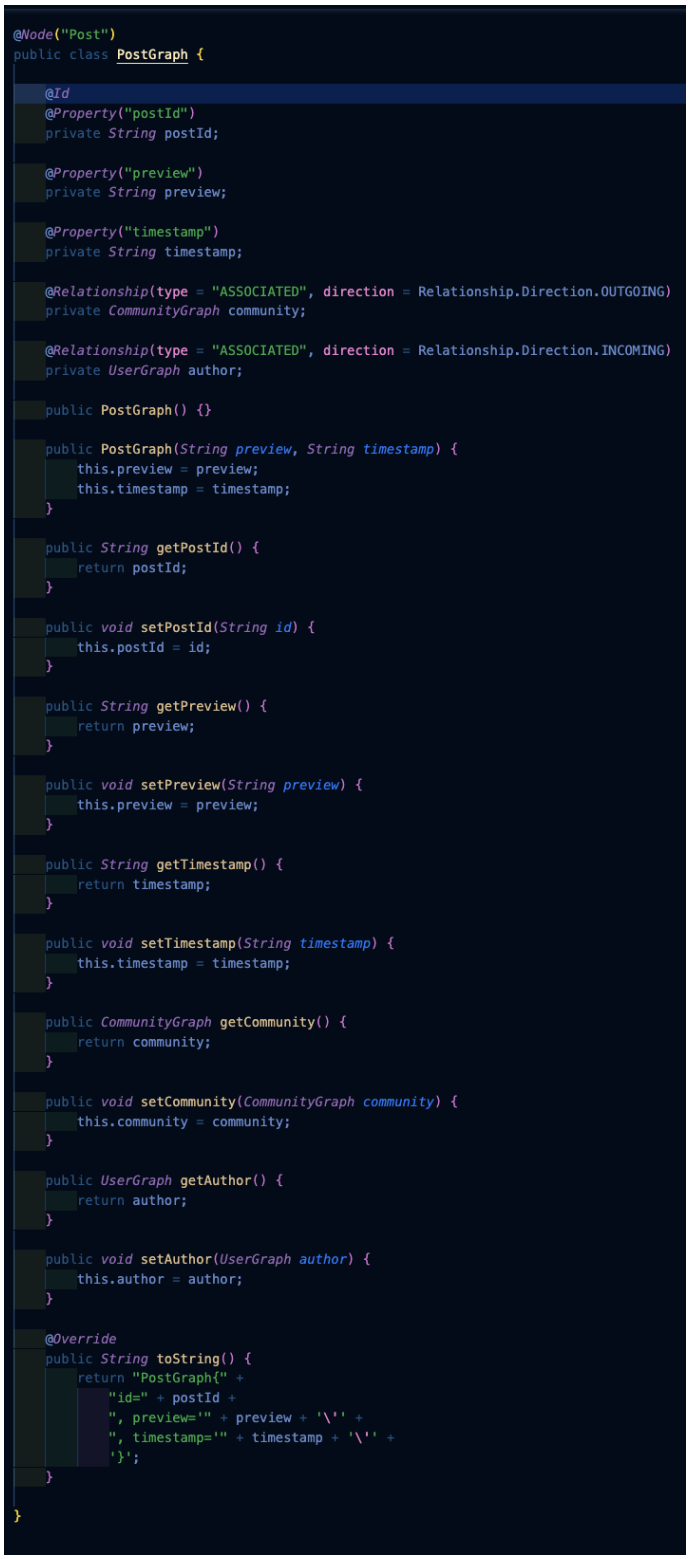
```java
package com.unipi.ItineraJava.model;

import org.springframework.data.mongodb.core.mapping.Field;

// COMMENT
public class Comment {

    @Field("_id")
    private String _id;
    @Field("username")
    private String username;
    @Field("timestamp")
    private String timestamp;
    @Field("body")
    private String body;
    @Field("reported")
    private boolean reported;

    public Comment() {}
    public void setCommentId(String commentId) {
        this._id = _id;
    }
    public String getCommentId() {
        return _id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String  getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(String  timestamp) {
        this.timestamp = timestamp;
    }
    public String getBody() {
        return body;
    }
    public void setBody(String body) {
        this.body = body;
    }
    public boolean isReported() {
        return reported;
    }
    public void setReported(boolean reported) {
        this.reported = reported;
    }
}
```

Figure 34: Comment Mongo Model

4.2.2 Neo4j Model

```java
@Node("User")
public class UserGraph {

    @Id
    @Property("username")
    private String username;

    @Relationship(type = "CONNECTED", direction = Relationship.Direction.OUTGOING)
    private CommunityGraph connectedCommunity;

    @Relationship(type = "ASSOCIATED", direction = Relationship.Direction.OUTGOING)
    private Set<PostGraph> posts;

    @Relationship(type = "COMMENT", direction = Relationship.Direction.OUTGOING)
    private Set<CommentRelationship> comments;

    @Relationship(type = "FOLLOWING", direction = Relationship.Direction.OUTGOING)
    private Set<UserGraph> following;

    public UserGraph() {}

    public UserGraph(String username) {
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public CommunityGraph getConnectedCommunity() {
        return connectedCommunity;
    }

    public void setConnectedCommunity(CommunityGraph connectedCommunity) {
        this.connectedCommunity = connectedCommunity;
    }

    public Set<PostGraph> getPosts() {
        return posts;
    }

    public void setPosts(Set<PostGraph> posts) {
        this.posts = posts;
    }

    public Set<CommentRelationship> getComments() {
        return comments;
    }

    public void setComments(Set<CommentRelationship> comments) {
        this.comments = comments;
    }

    public Set<UserGraph> getFollowing() {
        return following;
    }

    public void setFollowing(Set<UserGraph> following) {
        this.following = following;
    }
```

Figure 35: User Graph Model



```java
@Node("Post")
public class PostGraph {

    @Id
    @Property("postId")
    private String postId;

    @Property("preview")
    private String preview;

    @Property("timestamp")
    private String timestamp;

    @Relationship(type = "ASSOCIATED", direction = Relationship.Direction.OUTGOING)
    private CommunityGraph community;

    @Relationship(type = "ASSOCIATED", direction = Relationship.Direction.INCOMING)
    private UserGraph author;

    public PostGraph() {}

    public PostGraph(String preview, String timestamp) {
        this.preview = preview;
        this.timestamp = timestamp;
    }

    public String getPostId() {
        return postId;
    }

    public void setPostId(String id) {
        this.postId = id;
    }

    public String getPreview() {
        return preview;
    }

    public void setPreview(String preview) {
        this.preview = preview;
    }

    public String getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }

    public CommunityGraph getCommunity() {
        return community;
    }

    public void setCommunity(CommunityGraph community) {
        this.community = community;
    }

    public UserGraph getAuthor() {
        return author;
    }

    public void setAuthor(UserGraph author) {
        this.author = author;
    }

    @Override
    public String toString() {
        return "PostGraph{" +
            "id=" + postId +
            ", preview='" + preview + '\'' +
            ", timestamp='" + timestamp + '\'' +
            '}';
    }

}
```

Figure 36: Post Graph model

```java
package com.unipi.ItineraJava.model;

import org.springframework.data.neo4j.core.schema.*;

@RelationshipProperties
public class CommentRelationship {

    @Id
    @GeneratedValue
    private String internalId;

    @Property("commentId")
    private String commentId;

    @Property("timestamp")
    private String timestamp;

    @TargetNode
    private PostGraph post;

    // Costruttore
    public CommentRelationship(String commentId, String timestamp, PostGraph post) {
        this.commentId = commentId;
        this.timestamp = timestamp;
        this.post = post;
    }

    // Getter e setter
    public String getInternalId() {
        return internalId;
    }

    public void setInternalId(String internalId) {
        this.internalId = internalId;
    }

    public String getCommentId() {
        return commentId;
    }

    public void setCommentId(String commentId) {
        this.commentId = commentId;
    }

    public String getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }

    public PostGraph getPost() {
        return post;
    }

    public void setPost(PostGraph post) {
        this.post = post;
    }
}
```

Figure 37: Comment Graph Model

```java
package com.unipi.ItineraJava.model;

import org.springframework.data.neo4j.core.schema.GeneratedValue;
import org.springframework.data.neo4j.core.schema.Id;
import org.springframework.data.neo4j.core.schema.Node;
import org.springframework.data.neo4j.core.schema.Property;
import org.springframework.data.neo4j.core.schema.Relationship;

import java.util.Set;

@Node("Community")
public class CommunityGraph {

    @Id
    @GeneratedValue
    private Long id;

    @Property("city")
    private String city;

    @Relationship(type = "CONNECTED", direction = Relationship.Direction.INCOMING)
    private Set<UserGraph> connectedUsers;

    @Relationship(type = "ASSOCIATED", direction = Relationship.Direction.INCOMING)
    private Set<PostGraph> posts;

    public CommunityGraph() {}

    public CommunityGraph(String city) {
        this.city = city;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public Set<UserGraph> getConnectedUsers() {
        return connectedUsers;
    }

    public void setConnectedUsers(Set<UserGraph> connectedUsers) {
        this.connectedUsers = connectedUsers;
    }

    public Set<PostGraph> getPosts() {
        return posts;
    }

    public void setPosts(Set<PostGraph> posts) {
        this.posts = posts;
    }

    @Override
    public String toString() {
        return "CommunityGraph{" +
            "city='" + city + '\'' +
            ", connectedUsers=" + (connectedUsers != null ? connectedUsers.size() : "null") +
            ", posts=" + (posts != null ? posts.size() : "null") +
            '}';
}
```

Figure 38: Community Graph Model

4.3 Controller

The **Controller** layer is responsible for handling incoming HTTP requests from the client. It serves as the entry point into the application and acts as an intermediary between the client and the service layer. Each of the following controllers handles operations for both admins and users.

4.3.1 User Controller

It is responsible for handling user-related operations such as user registration, authentication, and profile management. This controller interacts with the User Service to execute the requests.

4.3.2 Review Controller

This controller handles operations related to add, report and browse reviews. It also handles admin operation such as ban reviews or see reported ones.

4.3.3 Community Controller

This controller handles operations related to join, leave and browse through communities.

4.3.4 Post Controller

This controller handles operations related to write, browse an delete posts and comments.

4.3.5 Place Controller

This controller handles operations related to add and browse places throughout the application.

4.4 Service

The **Service** layer contains the business logic of the application. It acts as a bridge between the controller and the repository layers, ensuring that the logic remains isolated and reusable. Additionally, the service modules allow better management of consistency and error handling, making the system more robust and maintainable.

4.4.1 Auth Service

This module handles the core functionalities of user registration and login. It validates user credentials, securely stores passwords using encryption, and generates JWT tokens upon successful authentication. During the registration process, it creates new user records in both MongoDB and Neo4j, maintaining data consistency across the two databases.

4.4.2 User Service

This module manages the core functionalities related to users in the application. It handles user management tasks, including retrieving user details, managing user activity status, and updating user information such as their last post. It ensures data consistency across MongoDB and Neo4j by implementing operations such as creating, updating, or deleting user records in both databases.

4.4.3 Review Service

This module handles all operations related to reviews within the application. It provides functionality for adding, reporting, retrieving, and deleting reviews, as well as managing review-related data for places.

4.4.4 Community Service

This module manages the core functionalities related to community management in the application. It integrates with both MongoDB and Neo4j to handle operations involving community data, ensuring consistency across the databases. This module is central to managing community activities, including user participation, post management, and activity monitoring.

4.4.5 Post Service

Thi module is responsible for managing all operations related to posts and comments within the application. It integrates MongoDB and Neo4j to ensure seamless functionality and data consistency for creating, retrieving, updating, and deleting posts and comments.

4.4.6 Place Service

This module is responsible for managing operations related to places in the application. It provides functionality for retrieving, updating, and managing place-related data stored in MongoDB. This module plays a vital role in ensuring that users can effectively interact with place-related features, such as reviews and ratings, within the application.

4.5 Repository

The **Repository** layer is responsible for interacting with the database. This layer handles operations such as querying, saving and deleting data. In our application we use repositories for MongoDB and Neo4j using simple and more complex queries.

4.6 Restful API endpoints

4.6.1 User API endpoints

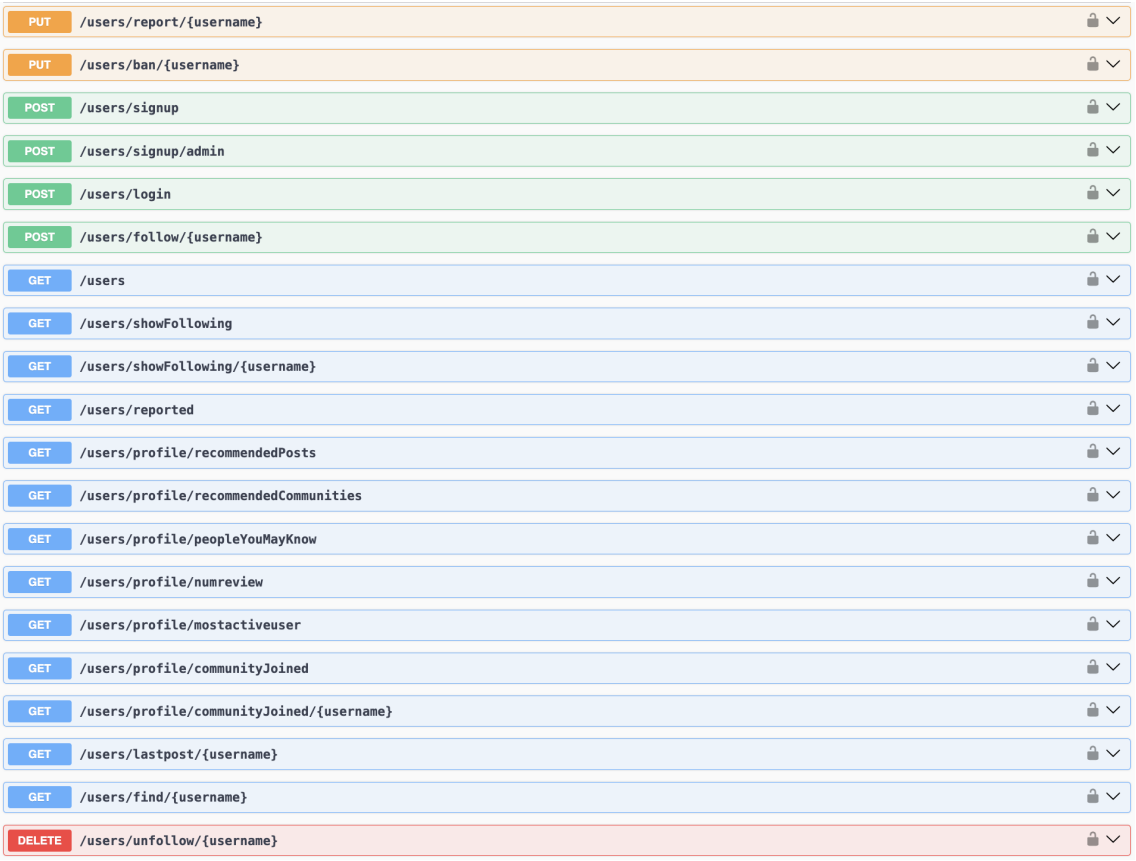


Figure 39: User API endpoints

4.6.2 Review API endpoints

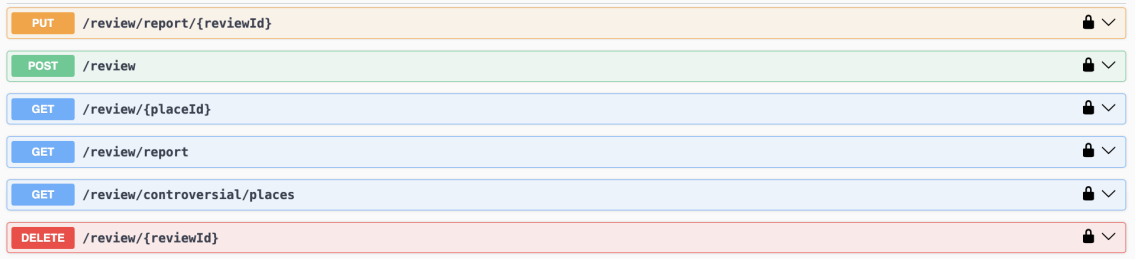


Figure 40: Review API endpoints

4.6.3 Place API endpoints

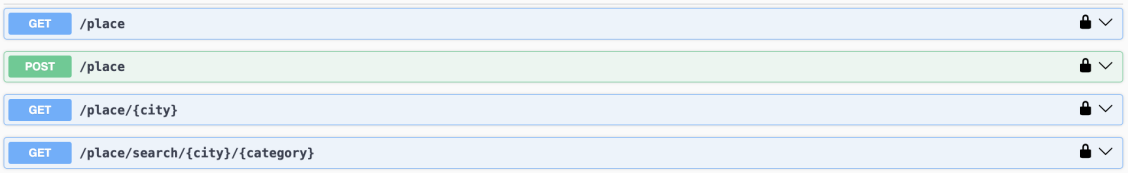


Figure 41: Place API endpoints

4.6.4 Community API endpoints

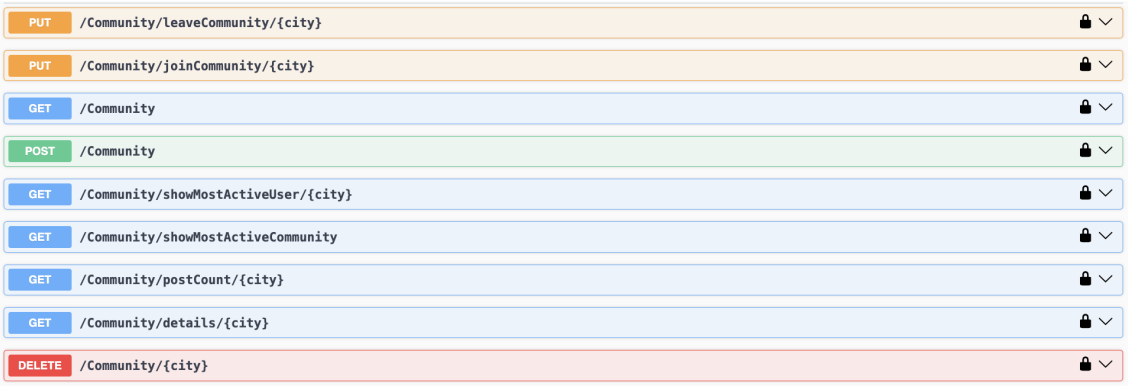


Figure 42: Community API endpoints

4.6.5 Posts API endpoints

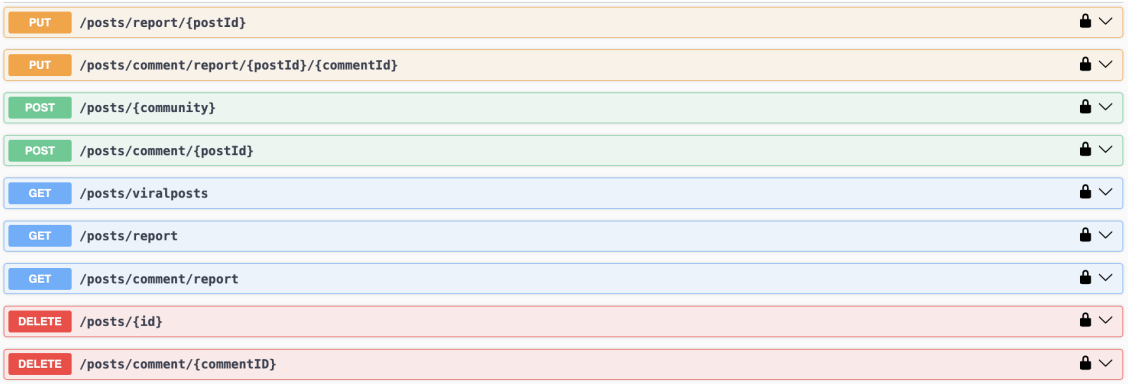


Figure 43: Post API endpoints

5 Most relevant queries

5.1 MongoDB queries

Most relevant queries in mongoDB are implemented using aggregation pipelines.

- **FIND MOST CONTROVERSIAL PLACES**

Figure 44: FindControversialPlaces Query

How It Works:

1. **Initial Filtering ($match):**
   - The first pipeline stage filters the reviews where the field ‘reported‘ is set to `false`.
   - This ensures that only unreported reviews are considered for further analysis.
2. **Grouping ($group):**
   - The reviews are grouped by the ‘place_id‘ field.
   - For each group:
     - `averageStars`: The average of the ‘stars‘ field is calculated.
     - `standardDeviation`: The standard deviation of the ‘stars‘ field is calculated using the `$stdDevPop` operator.
3. **Add Statistical Fields ($addFields):**
   - New fields are computed:
     - `variance`: Calculated as the square of the `standardDeviation` and rounded to 3 decimal places using `$pow` and `$round`.
     - `averageStars` and `standardDeviation`: Both are rounded to 3 decimal places.
4. **Filtering Based on Threshold ($match):**
   - The groups are filtered to retain only those with a `standardDeviation` greater than 1.3.
   - This step identifies places with high variability in user ratings.
5. **Sorting ($sort):**
   - The remaining results are sorted in descending order based on the `variance` field.
   - This ensures that places with the highest variability appear first in the final output.
6. **Final Projection ($project):**
   - The final output includes:
     - `id`: The `_id` field renamed as `id`.

- `variance`, `standardDeviation`, and `averageStars`.
- All other fields are excluded.

- **FIND REPORTED POSTS WITH HIGHEST NUMBER OF COMMENTS**

```
@Aggregation(pipeline = {
        "{ '$match': { 'reportedpost': true } }",
        "{ '$addFields': { 'reportedComments': { '$size': { '$filter': { " +
                "        'input': '$comment', " +
                "        'as': 'c', " +
                "        'cond': { '$eq': ['$$c.reported', true] } " +
                "    } } } } }",
        "{ '$sort': { 'reportedComments': -1 } }",
        "{ '$limit': 10 }",
        "{ '$project': { '_id': 0, 'id': '$_id', 'community': 1, 'username': 1, 'post': 1, 'reportedComments': 1, 'timestamp': 1 } }"
})
List<PostSummaryDto> findTopReportedPostsByCommentCount();
```

Figure 45: Reported posts with most comments Query

How It Works:

1. **`$match`:**
   - Filters the collection to include only posts where the `reportedpost` field is set to `true`.
   - This ensures that the aggregation process works exclusively on reported posts.
2. **`$addFieldse`:**
   - Adds a new field called `reportedComments` to each document.
   - This field calculates the number of reported comments for the post using the following steps:
     (a) Uses `$filter` to iterate through the `comment` array.
     (b) For each comment, checks if the `reported` field is `true`.
     (c) Counts the number of comments that meet this condition with the `$size` operator.
3. **`$sort`:**
   - Sorts the posts in descending order of the `reportedComments` field.
   - Posts with more reported comments will appear earlier in the result.
4. **`$limit`:**
   - Restricts the result to the top 10 posts based on the sorted order.
   - This stage ensures that only the most relevant posts are returned.
5. **`$project`:**
   - Specifies the fields to include in the final output:
     - `id`: The unique identifier of the post, renamed from `_id`.
     - `community`: The community where the post was made.
     - `username`: The username of the person who created the post.
     - `post`: The content of the post.
     - `reportedComments`: The calculated number of reported comments.
     - `timestamp`: The timestamp of the post's creation.
   - Excludes the default `_id` field from the output by setting it to `0`.

- **FIND TOP 3 CITIES BY RATING**

```
@Aggregation(pipeline = {
        "{ '$group': { " +
                "   '_id': '$city', " +
                "   'averageRating': { '$avg': '$reviews_info.overall_rating' }, " +
                "   'totalPlaces': { '$sum': 1 } " +
                "} }",
        "{ '$sort': { 'averageRating': -1 } }",
        "{ '$limit': 3 }",
        "{ '$project': { " +
                "   '_id': 0, " +
                "   'city': '$_id', " +
                "   'averageRating': 1, " +
                "   'totalPlaces': 1 } }"
})
List<TopRatedCitiesDTO> findTopRatedCitiesByAverageRating();
```

Figure 46: Top Cities By Rating

How It Works:

1. **Group by city:** The query groups documents by the `$city` field. For each city, it calculates the following:
   - `averageRating`: The average of the `$reviews_info.overall_rating` field across all places in the city.
   - `totalPlaces`: The total count of places (`$sum: 1`) in each city.
2. **Sort by average rating:** The cities are sorted in descending order of their `averageRating`, prioritizing cities with higher ratings.
3. **Limit to top 3 cities:** The query limits the results to only the top 3 cities with the highest average ratings.
4. **Project the fields:** The output includes the following fields:
   - `city`: The name of the city (mapped from the `_id` field in the grouping stage).
   - `averageRating`: The calculated average rating for the city.
   - `totalPlaces`: The total number of places in the city.

5.2 GraphDB queries

The GraphDB is primarily utilized to track user interactions with each other and within communities. Beyond basic CRUD operations, the most notable queries related to the graph are those designed to provide personalized recommendations to users and to identify the most active users or communities. These queries leverage the structure of the graph to uncover valuable insights and enhance the user experience.

- **FIND SUGGESTED USERNAMES**

```
@Query("MATCH (u:User {username: $username}) " +
       "OPTIONAL MATCH (u)-[:FOLLOWING]->(f:User)<-[:FOLLOWING]-(suggested1:User) " +
       "WHERE NOT (u)-[:FOLLOWING]->(suggested1) AND u <> suggested1 " +
       "WITH u, COLLECT(DISTINCT suggested1.username) AS firstLevelSuggestions " +
       "OPTIONAL MATCH (u)-[:CONNECTED]->(:Community)<-[:CONNECTED]-(suggested2:User) " +
       "WHERE NOT (u)-[:FOLLOWING]->(suggested2) AND u <> suggested2 " +
       "WITH u, firstLevelSuggestions, COLLECT(DISTINCT suggested2.username) AS secondLevelSuggestions " +
       "OPTIONAL MATCH (suggested3:User) " +
       "WHERE NOT (u)-[:FOLLOWING]->(suggested3) AND u <> suggested3 " +
       "WITH firstLevelSuggestions AS level1, secondLevelSuggestions AS level2, " +
       "     COLLECT(DISTINCT suggested3.username) AS level3 " +
       "WITH level1 + level2 + level3 AS finalSuggestions " +
       "UNWIND finalSuggestions AS suggestedUsername " +
       "RETURN DISTINCT suggestedUsername " +
       "LIMIT 10")
List<String> findSuggestedUsernames(@Param("username") String username);
```

Figure 47: FindSuggestedUser Query

This query retrieves a list of suggested users for a given user based on three levels of connections within the social graph.

1. **First-level suggestions:** Users followed by the people the given user follows but who are not yet followed by the user.
2. **Second-level suggestions:** Users connected to the same communities as the given user but who are not yet followed by them
3. **Third-level suggestions:** Additional users that are not yet followed by the user but are identified as potential recommendations based on indirect connections, useful when the User doesn't follow anyone or hasn't joined a community yet.

The query aggregates these suggestions, filters out duplicates, and returns up to 10 unique suggested usernames.

- **FIND SUGGESTED COMMUNITIES**

```
@Query("MATCH (u:User {username: $username}) " +
       "OPTIONAL MATCH (u)-[:FOLLOWING]->(f:User)-[:CONNECTED]->(suggestedCommunity:Community) " +
       "WHERE NOT (u)-[:CONNECTED]->(suggestedCommunity) " +
       "WITH u, COLLECT(DISTINCT suggestedCommunity.city) AS suggestedByFollowedUsers " +
       "OPTIONAL MATCH (randomCommunity:Community) " +
       "WHERE NOT (u)-[:CONNECTED]->(randomCommunity) " +
       "WITH suggestedByFollowedUsers AS level1, COLLECT(DISTINCT randomCommunity.city) AS level2 " +
       "WITH level1 + level2 AS allSuggestions " +
       "UNWIND allSuggestions AS suggestedCommunityName " +
       "RETURN DISTINCT suggestedCommunityName " +
       "LIMIT 5")
List<String> findSuggestedCommunities(@Param("username") String username);
```

Figure 48: FindSuggestedCommunities Query

This Neo4j query suggests communities that a user might be interested in joining. The suggestions are generated based on two levels:

1. **Communities Followed by the User's Followed Users:** Finds communities that are joined by users whom the target user follows but has not joined yet.

   If the user doesn't follow anyone yet:

2. **Random Communities:** Additionally, selects random communities that the user has not yet joined, providing diverse suggestions.

The query ensures that the user is not already connected to the suggested communities and limits the results to five distinct communities. Additionally, it helps in expanding user engagement by recommending relevant communities based on social connections and random exploration.

- **FIND SUGGESTED POSTS**

```
@Query("""
       MATCH (u:User {username: $username})
       OPTIONAL MATCH (u)-[:FOLLOWING]->(f:User)-[:ASSOCIATED]->(p:Post)
       OPTIONAL MATCH (p)-[:ASSOCIATED]->(c:Community)
       WITH collect({ post: p, community: c }) AS pac
       CALL {
       WITH pac
       WITH pac, size([x IN pac WHERE x.post IS NOT NULL]) AS postCount
       WHERE postCount > 0
       UNWIND pac AS row
       WITH row.post AS post, row.community AS community
       WHERE post IS NOT NULL
       RETURN post.postId       AS postId,
       post.preview   AS preview,
       community.city AS community
       UNION
       WITH pac
       WITH pac, size([x IN pac WHERE x.post IS NOT NULL]) AS postCount
       WHERE postCount = 0
       MATCH (randomPost:Post)
       OPTIONAL MATCH (randomPost)-[:ASSOCIATED]->(rc:Community)
       WITH randomPost, rc, rand() AS r
       ORDER BY r
       LIMIT 10
       RETURN randomPost.postId      AS postId,
       randomPost.preview AS preview,
       rc.city            AS community
       }
       RETURN postId, preview, community
       """)
List<PostSuggestionDto> findSuggestedPosts(@Param("username") String username);
```

Figure 49: FindSuggestedPosts Quey

This Neo4j query suggests posts that a user might be interested in reading. The recommendations are based on two strategies:

1. **Posts from Communities Followed by the User's Followed Users:** Identifies posts written by users that the target user follows and ensures that the posts belong to communities where these followed users have participated.
2. **Random Posts from Unrelated Communities:** If no posts are found using the first method, the query selects random posts from the database to ensure diversity in recommendations.This helps in discovering new communities and discussions

This query enhances user engagement by providing relevant post recommendations, encouraging exploration within familiar communities as well as introducing new ones.

- **FIND MOST ACTIVE COMMUNITY**

```
@Query("MATCH (c:Community) " +
    "OPTIONAL MATCH (c)<-[:CONNECTED]-(u:User) " +
    "OPTIONAL MATCH (c)<-[:ASSOCIATED]-(p:Post) " +
    "OPTIONAL MATCH (p)<-[:COMMENT]-(uCommenter:User) " +
    "WITH c.city AS communityName, " +
    "     COUNT(DISTINCT u) AS users, " +
    "     COUNT(DISTINCT p) AS posts, " +
    "     COUNT(DISTINCT uCommenter) AS comments " +
    "RETURN communityName " +
    "ORDER BY (users + posts + comments) DESC " +
    "LIMIT 1")
String findMostActiveCommunity();
```

Figure 50: Find Most Active Community Query

How It Works:

1. **Match** the `Community` nodes (`c:Community`).
2. **Optional Matches:**
   - Users (`u:User`) connected to the community (`CONNECTED` relationship).
   - Posts (`p:Post`) associated with the community (`ASSOCIATED` relationship).
   - Users (`uCommenter:User`) who commented on posts (`COMMENT` relationship).
3. **Count Distinct Contributions:**
   - Counts the number of unique users connected to the community.
   - Counts the number of unique posts in the community.
   - Counts the number of unique commenters.
4. **Sort by Engagement:**
   - Computes a total activity score as the sum of users, posts, and comments.
   - Orders communities **in descending order** based on activity.
5. **Returns** the most active community (`LIMIT 1`).

This query is useful for **analyzing user engagement** in different communities and **identifying the most active one**, which can be helpful for recommendations, moderation, or promotional activities.

- **FIND MOST ACTIVE USER IN A COMMUNITY**

```
@Query("MATCH (c:Community {city: $city})<-[:CONNECTED]-(u:User) " +
    "OPTIONAL MATCH (u)-[:ASSOCIATED]->(p:Post)-[:ASSOCIATED]->(c) " +
    "OPTIONAL MATCH (u)-[:COMMENT]->(com:Post)-[:ASSOCIATED]->(c) " +
    "WITH u, COUNT(DISTINCT p) AS posts, COUNT(DISTINCT com) AS comments " +
    "RETURN u.username AS username " +
    "ORDER BY (posts + comments) DESC " +
    "LIMIT 1")
String findMostActiveUserByCommunity(@Param("city") String city);
```

Figure 51: FindMostActiveUserByCommunity Quey

How It Works:

1. **Match** the `Community` node (`c:Community`) for a given city.
2. **Optional Matches:**
   - Users (`u:User`) connected to the community via the `CONNECTED` relationship.
   - Users who created posts in the community (`ASSOCIATED` relationship).
   - Users who commented on posts in the community (`COMMENT` relationship).
3. **Count Contributions:**
   - Counts the number of distinct posts a user has created.
   - Counts the number of distinct comments a user has made.
4. **Sort Users by Engagement:**
   - Orders users by the sum of posts and comments in descending order.
5. **Returns** the username of the most active user (`LIMIT 1`).

This query helps in **analyzing user activity within a specific community**, allowing insights into the most engaged members. It is useful for community moderation and user recommendations.

6 Indexes

6.1 MongoDB Indexes

In this section, we discuss the possible indexes that could improve the performance of read operations for each collection in the Itinera project. Subsequently, each index will be analyzed to determine whether it should be added.

6.1.1 Users Collection

- **Username**
  This index would assist with most of the queries the users will use, based on our estimation of the users actions inside of Itinera. So we dediced to use this type of index in this collection. To evaluate his impact, we conducted a performance test with and without. Other indexes, such as those on the `Active` field and `postCount`, were not selected because they would be used in infrequent queries. Additionally, these fields are frequently recalculated, which could result in a significant decrease in efficiency.

	Without Index	With Index
nReturned	1	1
executionTimeMillis	12	4
totalKeysExamined	0	1
totalDocsExamined	5662	1

Figure 52: User Index Statistic

6.1.2 Reviews Collection

- **place_id**
  This index is crucial for filtering reviews by place. A performance test was conducted to compare query execution times with and without the index.

  Another index applied was on the `reported` field. However, like other discarded indexes, this one is primarily involved in high-write operations and administrative queries.

	Without Index	With Index
nReturned	405	405
executionTime (ms)	138	5
totalKeysExamined	0	405
totalDocsExamined	138,871	405

Figure 53: review index statistics

6.1.3 Places Collection

- **Category and city**
  Indexing the `category` field would enhance the performance of filtering places by type. Additionally, indexing the `city` field would allow for more efficient and faster retrievals from this collection.

  A compound index on `city`, `category`, and `overall_rating` could be beneficial, but the `overall_rating` field is frequently recalculated and is primarily used in administrative queries, which are infrequent.

	Without Index	With Index
nReturned	27	27
executionTime (ms)	13	10
totalKeysExamined	0	27
totalDocsExamined	1402	27

Figure 54: Place index statistics

6.1.4 Community Collection

- **City**
  Adding an index on the city of the community would optimize searches when users look for specific communities by name.

	Without Index	With Index
nReturned	1	1
executionTime (ms)	4	2
totalKeysExamined	0	405
totalDocsExamined	47	1

Figure 55: community index statistic

6.1.5 Posts Collection

- The only index applied to `posts` is the default `_id` index, which is automatically created by MongoDB, and on the `community`, which speeds up the queries envolving this field. While the `reported` field is a potential candidate for indexing, it is primarily used in queries for administrative purposes (which account for only 0.01% of total queries) and in report-related operations (such as reporting posts or comments), which also constitute a negligible portion of the workload. Thus, the overhead of maintaining an index on `reported` is not justified by its limited utility.

	Without Index	With Index
nReturned	9	9
executionTime (ms)	7	1
totalKeysExamined	0	9
totalDocsExamined	436	9

Figure 56: Post index statistics

6.2 Neo4j Indexes

Considering the queries related to the GraphDB, we decided to use the attribute that uniquely defines each node as an index. Specifically, we implemented the following indexing strategy:

- **User Node** → indexed by `username`
- **Community Node** → indexed by `city`
- **Post Node** → indexed by `postId`

This indexing approach optimizes query performance and ensures efficient data retrieval.