MyVacation

Large-Scale and Multi-Structured Databases

Bataloni Lorenzo
Daka Claudio
Pantè Edoardo

Academic Year: 2022/2023

MyVacation

project repo: https://github.com/bata26/MyVacation-LSMSDB

1. Project Idea

In recent years, travel has played an increasingly important role in our lives. During the pandemic, the spread of smart working has allowed people to realise that it is not essential to be in the office every day to be truly productive, and more importantly that the environment in which one works is a very important aspect. MyVacation was created with the goal of offering all its users the opportunity to discover new places and experience them through experiences and accommodations offered by the very people who live those places every day, away from the routine and hustle and bustle.
Whether you are looking for temporary housing or a home for your travels, MyVacation has what you need.

1.1. Application and Requirements

Our idea is to provide a platform where users can offer accommodation and activities to users who want to experience the city in a different way from the usual tourist vacation.
The application allows users to view accommodations and activities, book them, and leave a review. It then allows users to follow users and have suggested listings based on the people being followed.
Finally, for people who post listings, the application allows them to have additional information related to the cities in their listings.

1.1.1 Functional Requirements

The users of the application will be divided into three categories: *Unregistered Users*, *Registered Users* and *Admin*.
Registered Users are allowed to use the main functionalities of the application.
A Registered User can be **standard** or a **host**.
Access to the application is guaranteed to everyone. It's provided a login system using username and password, through which the user will be correctly identified.
A registration form will allow new users to register within the application as Registered Users.

Unregistered User:

- Unregistered User can register to the platform.
- Unregistered User can browse an accommodation/activity filtered by city, period and number of guests.
- Unregistered User can view an accommodation/activity.

- Unregistered User can browse the top 3 cities.
- Unregistered User can browse top 3 accommodations.
- Unregistered User can browse the top 3 activities.
- Unregistered User can view a review.

Registered User:

- Registered User can log in to the platform.
- Registered User can log out from the platform.
- Registered User can browse accommodation/activity filtered by city, period and number of guests.
- Registered User can insert an accommodation/activity.
- Registered User can view an accommodation/activity.
- Registered User can update an accommodation/activity.
- Registered User can delete an accommodation/activity.
- Registered User can browse the top 3 cities.
- Registered User can browse the top 3 accommodations.
- Registered User can browse the top 3 activities.
- Registered User can browse a recommended accommodation/activity.
- Registered User can browse a recommended user.
- Registered User can reserve an accommodation/activity.
- Registered User can view their own reservations.
- Registered User can update a reservation.
- Registered User can delete a reservation.
- Registered User can insert a review.
- Registered User can view a review.
- Registered User can delete a review.
- Registered User can view a user profile.
- Registered User can follow/unfollow an user.
- Registered User can like/unlike an accommodation/activity.
- Registered User can view their own user profile.
- Registered User can update their own user profile.
- Registered User can see the booking trend for each month.

Admin:

- Admin can log in to the platform.
- Admin can log out from the platform.
- Admin can browse an accommodation/activity filtered by city, period and number of guests.
- Admin can view an accommodation/activity.
- Admin can update an accommodation/activity.
- Admin can delete an accommodation/activity.

- Admin can browse the top 3 cities.
- Admin can browse the top 3 accommodations.
- Admin can browse the top 3 activities.
- Admin can view all reservations.
- Admin can delete a reservation.
- Admin can view a user profile.
- Admin can update a user profile.
- Admin can delete an user.
- Admin can approve/refuse an accommodation/activity insertion.
- Admin can view all the stats and analytics on the admin page.

1.1.2. Non-Functional Requirements

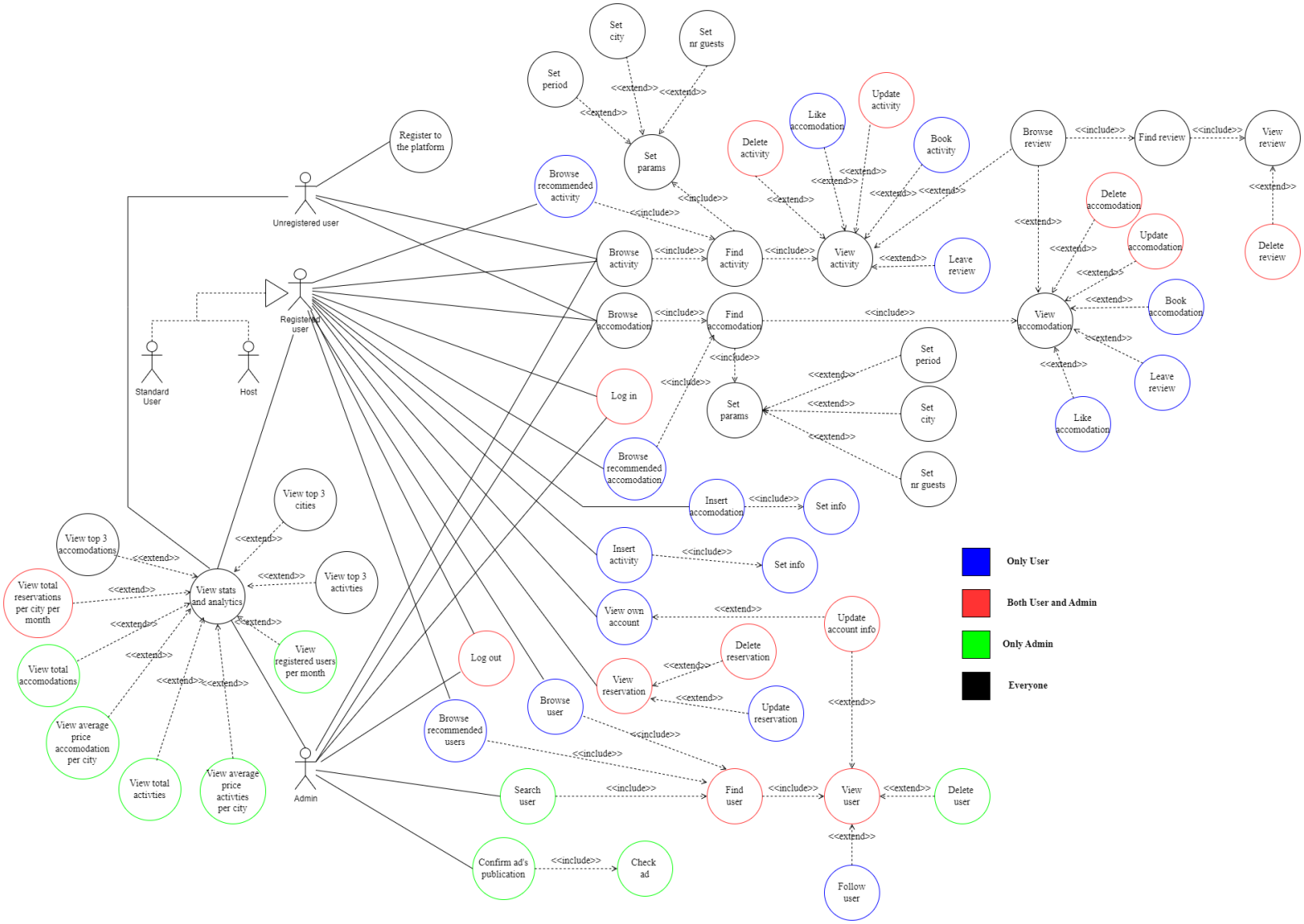
- The application needs to provide low latency, high availability and tolerance to single points of failures (**SPOF**) and network partitions, for which the application has been designed in order to prefer the Availability (**A**) and Partition Protection (**P**).
- The *Eventual Consistency* paradigm must be adopted.
- The application's backend must be written in Python.
- The application's frontend must be written in Javascript.
- Operations on the databases have to be atomic.
- Registered Users are identified by their username.
- Registered Users can leave only one review for a reserved accommodation/activity.
- An accommodation/activity has to be approved by the Admin.
- When an accommodation/activity is updated, it has to be approved.
- Every accommodation/activity must have at least one picture.
- An accommodation/activity cannot have different reservations for the same period.
- When an User has been deleted from the platform, his reservations remain on the database.
- *Usability*: The application needs to be user friendly, providing a GUI.
- *Privacy*: The application shall provide the security of users' credentials.
- *Portability*: The system shall be environment independent.
- *Maintainability*: The code shall be readable and easy to maintain.
- The client side and server side have to use HTTP protocol to communicate.

1.2. Specification

1.2.1. Actors and Use Case Diagram

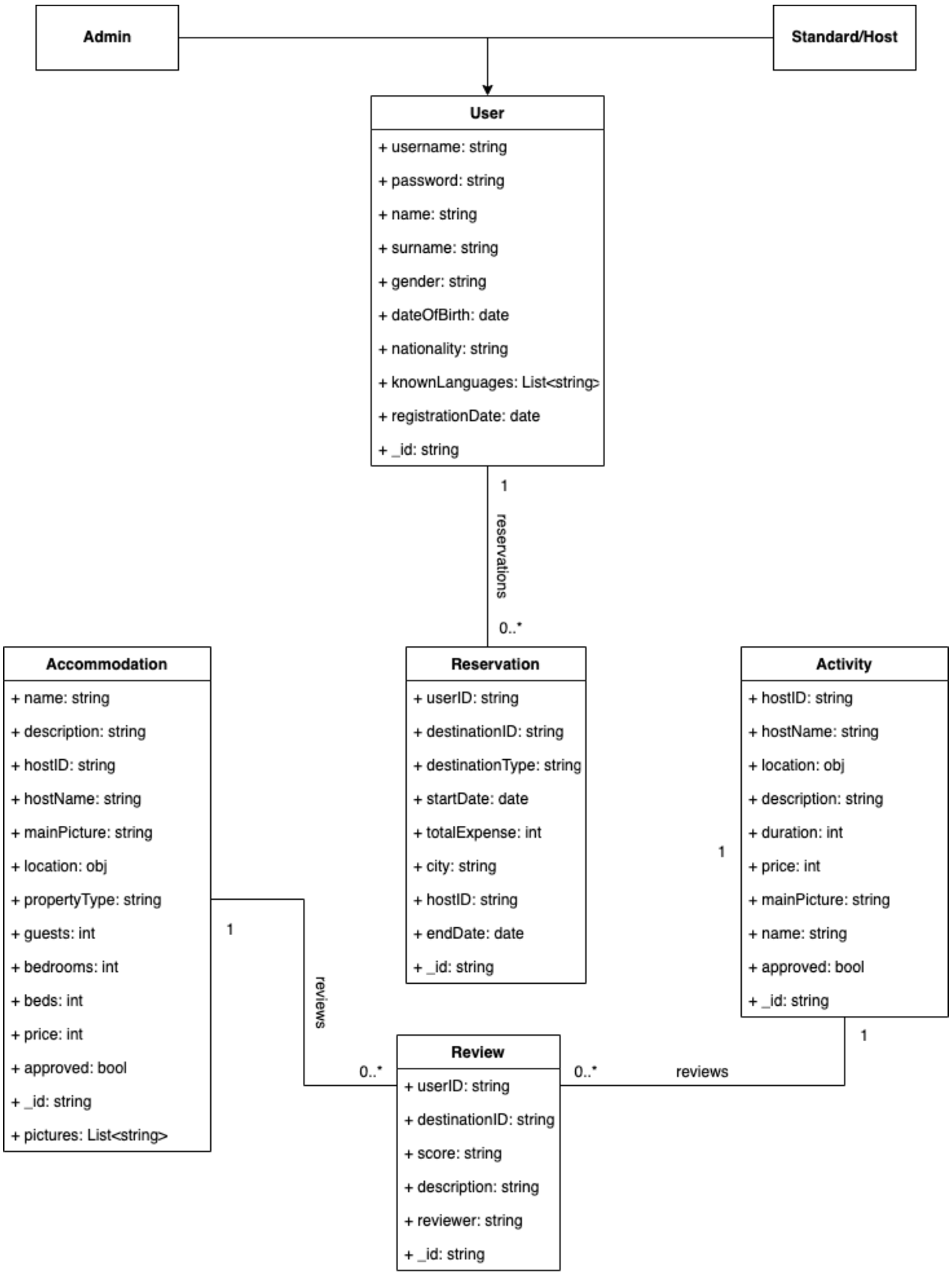
Based on software's functional requirements, we can distinguish three different actors:

- *Administrators*, who are allowed to manage users, reservations and to view analytics about all applications's info.
- *Registered User,* who are allowed to interact with all applications's functionalities. They can browse, search, book and review accommodations/activities. They can follow and unfollow users and view user's profile info. They can like and unlike an accommodation/activity and they can check and edit their own account info.
- *Unregistered User*, who are allowed only to browse and search for different accommodations/activities but they can't book or review them. They also can register or login.



1.2.2. Analysis Classes Diagram

Based on software's functional requirements, we can see in the following diagram the main entities of the application and the relation between them.

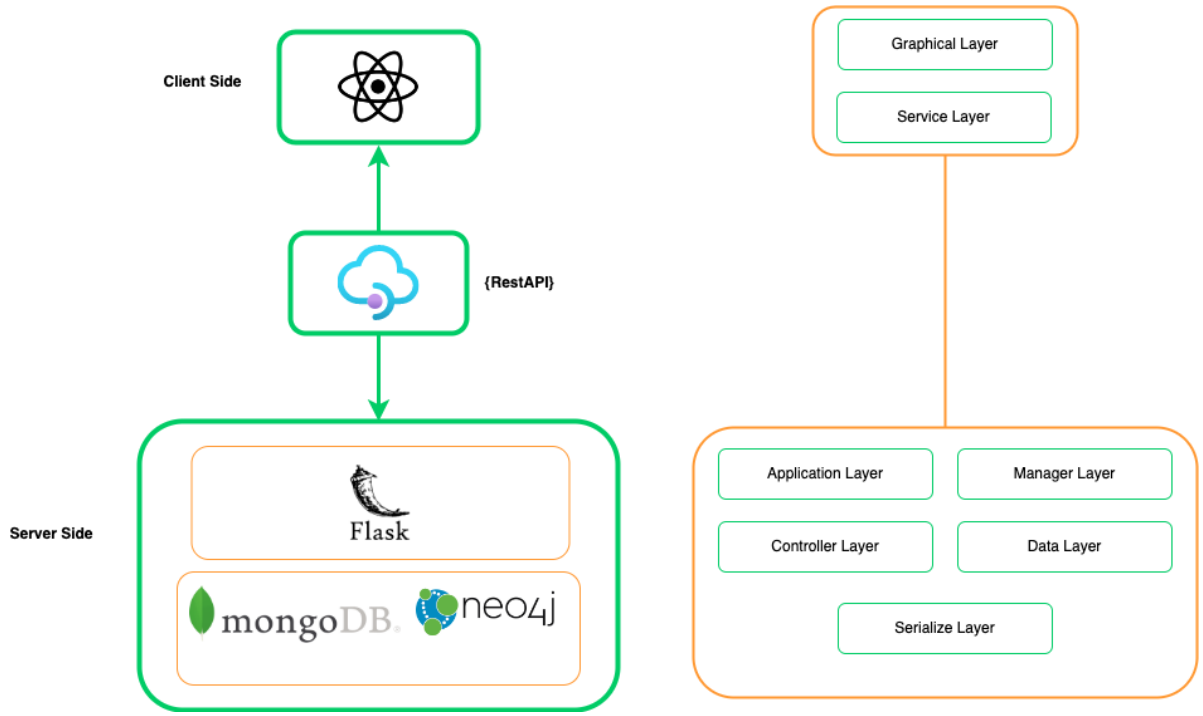


Each user can follow/unfollow other users. Each user may like or dislike multiple accommodations or activities. Each user can be a standard/host user or admin.

Each user can do multiple reservations. Each user can leave only a review for an accommodation or an activity. Each reservation can involve only one accommodation/activity and one user. Each user can publish multiple accommodations/activities.

Each accommodation/activity can have zero to multiple reviews.

1.2.3. System Architecture



1.2.4. Server Side

The server implements a REST API with the Flask Python Framework. Each functionality is reachable via an endpoint accessed using a URI. The server has different layers, each layer has a different task, so the execution flow goes to involve each of them.

- The application layer handles the request from the time it arrives to the time the response is sent to the client. This layer calls the controller related to the entity that has to be handled and in case of errors reports them to the client.

- The controller layer handles all application logic. Each controller deals with a specific logic entity.

- The manager layer implements all operations that access the database. Each manager deals with a specific entity for each database in which it is represented, while in the controller layer, entities are managed at the logical level.

- The Model layer contains the entities that represent the data stored in the database.

- The serializer layer is responsible for serializing all the results of the processing that is done by the manager layer to make them manageable for the client.

For access to MongoDB, *pymongo* driver was used, which allows a client to be created to manage the connection and all access to the database. For access to Neo4j, on the other hand, the official *neo4j* driver was used. Both connections are handled using the **singleton pattern**, this allows creating one connection and reusing it without going to create new connections that would degrade the performance of the whole system.

The server side uses a cluster of 3 virtual machines offered by University of Pisa, they are used to implement a MongoDB Replica Set with a primary and two secondary actors and to run a single instance of a Neo4j database.

- **172.16.5.32**: MongoDB Server (**PRIMARY**) and Neo4j instance.
- **172.16.5.33**: MongoDB Server (**SECONDARY**).
- **172.16.5.34**: MongoDB Server (**SECONDARY**).

1.2.5. Client Side

The client side has two main functions: to provide a Graphical User Interface to make the experience pleasant and usable for all, and to handle all requests for the server and the responses received. The Graphical Layer takes care of the graphical interface while the Service Layer takes care of request handling.

1.2.6. Frameworks

The application has been developed using two different programming languages. Python has been used for the server side while Javascript has been used for the client side. The React framework has been used for the Graphical User Interface. The server side is implemented using the Python Flask framework. Concerning the database management systems, Neo4j and MongoDB have been used. For version control, Git has been used.

1.2.6. Dataset Organization and Database Population

Data collection was done through datasets found online regarding accommodations, activities and users. Reservations and reviews, on the other hand, were generated through scripts.

1.2.6.1. User Information

User information was created through https://randomuser.me. Some users have been turned into hosts while others are standard users. This information was then saved in both the Document Database and the Graph Database.

1.2.6.2. Accommodation Information

Information on accommodations was taken from the dataset http://data.insideairbnb.com/italy/puglia/puglia/2022-09-26/data/listings.csv.gz. The dataset contains all the accommodations present in Puglia; given the size of the dataset, the city of some accommodations was modified in order to have a uniform distribution for all Italian cities. In the dataset there was only one image for each accommodation, because in a real context the images could be more, default photos were added for each accommodation. The information collected is related to the type of accommodation, guests, bedrooms, beds present, and more.

1.2.6.3. Activity Information

The information on activities was taken from the dataset https://data.world/dcopendata/9ad16c12e50e4664a95a016766b5ad51-18/workspace/file?filename=All_Summer_Events.dbf. The database contains information related to summer activities in some American cities. Again, the city was modified to have a uniform distribution in Italy. Unfortunately, there were no photos or posters related to the activities in the dataset, so a default one was included. The information collected is related to the name of the activity, duration, location and price of them.

1.2.6.4. Reservation filler

As mentioned, reservations information was created from the above datasets via scripts in python.

```
for i in range(0 , 1200):
  startTimestamp = lastTimestamp + 86400 # aggiungo un giorno
  startDatetime = datetime.fromtimestamp(startTimestamp).replace(hour=0, minute=0, second=0)


  reservation = {}
```

```
reservation = {
    "userID" : usersList[randint(0 , len(usersList) - 1)],
    "startDate" : startDatetime
}

if(i % 5 == 0): #activity
    activity = activityList[randint(0 , len(activityList) - 1)]
    destinationID = activity["_id"]
    totalExpense = activity["price"]
    city = activity["city"]
    type = "activity"
    hostID = activity["hostID"]
else:
    numberOfDays = randint(0 , 20)
    endDatetime = startDatetime + timedelta(days=numberOfDays)
    lastTimestamp = endDatetime.timestamp()
    accommodation = accommodationList[randint(0 , len(accommodationList) - 1)]
    destinationID = accommodation["_id"]
    totalExpense = accommodation["price"] * (numberOfDays - 1)
    reservation["endDate"] = endDatetime
    city = accommodation["city"]
    type = "accommodation"
    hostID = accommodation["hostID"]

reservation["totalExpense"] = totalExpense
reservation["destinationID"] = destinationID
reservation["city"] = city
reservation["destinationType"] = type
reservation["hostID"] = hostID

userID = str(hostID)
while(str(userID) == str(hostID)):
    userID = usersList[random.randint( 0 , len(usersList) - 1)]
reservation["userID"] = userID
print(reservation)
reservationList.append(reservation)


try:
with client.start_session() as session:
    with session.start_transaction():
        reservationCollection.insert_many(reservationList , session=session)
```

```
        for reservation in reservationList:
            userID = str(reservation["userID"])
            toInsertReservation = reservation
            toInsertReservation.pop("userID")

            userCollection.update_one({"_id" : ObjectId(userID)} ,  {"$push" : {"reservations" : toInsertReservation}} ,
session=session)
```

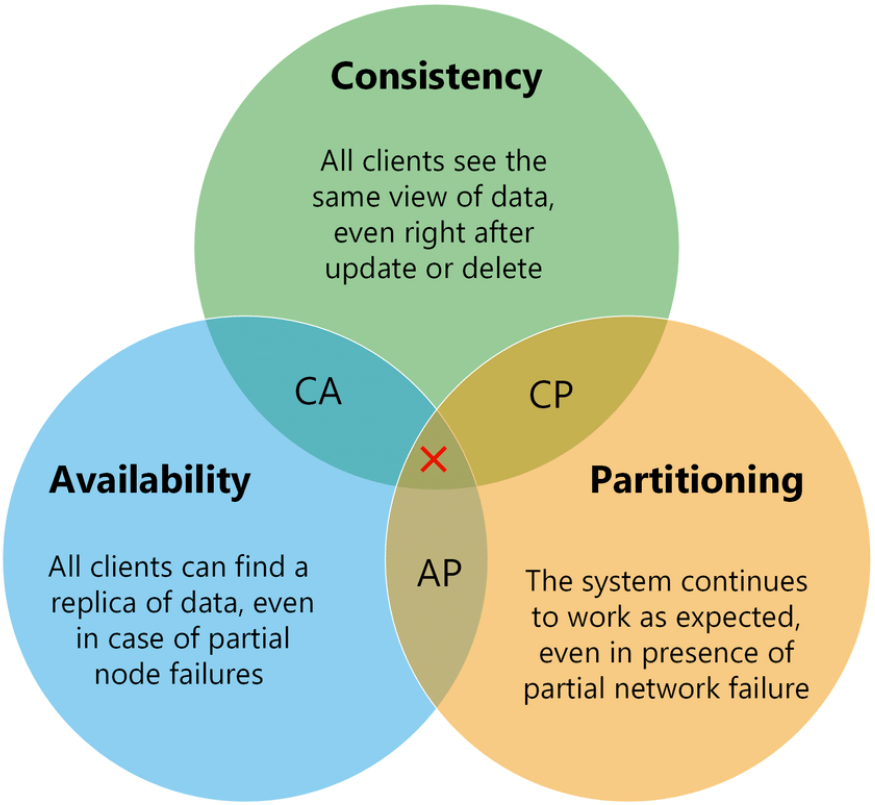
1.2.6.5. Review Information

As mentioned, reviews information was created from the above datasets via scripts in python.

```
for reservation in reservationList:
  vote = comments[randint(0 , len(comments) - 1)]
  review = {
   "score" : vote["score"],
   "description"  :vote["comment"],
   "userID" : reservation["userID"],
   "destinationID" : reservation["destinationID"],
   "reviewer" : getUsernameFromID(str(reservation["userID"]))
  }
  reviewList.append(review)
  try:
     with client.start_session() as session:
        with session.start_transaction():
            reviewCollection.insert_one(review , session=session)
            if reservation["destinationType"] == "accommodation":
                accommodationCollection.update_one({"_id" : ObjectId(str(reservation["destinationID"]))} , { "$push" : {"reviews" :
review}} , session=session)
                accommodationCollection.update_one({"_id" : ObjectId(str(reservation["destinationID"]))} , {"$push" : {"reviews"
:{"$each" : []  , "$slice" : -15}}} , session=session)
            else:
                activityCollection.update_one({"_id" : ObjectId(str(reservation["destinationID"]))} , { "$push" : {"reviews" : review}} ,
session=session)
                activityCollection.update_one({"_id" : ObjectId(str(reservation["destinationID"]))} , {"$push" : {"reviews" :{"$each" :
[]  , "$slice" : -15}}}, session=session)

  except Exception as e:
     print("Impossibile inserire tutte : " + str(e))
```

1.2.7. Handling CAP theorem issue



AP Solution (Availability and Partition Tolerance): The application should be accessible to the users at any given point in time and the expectation is to have a lot of read operations. It needs to continue to function regardless of system failures and network partitions. It may result in inconsistency at some points however, this is a small cost compared to the benefit of availability in the case of this application. In order to respect the non-functional requirements, we decided to guarantee high availability of data, even if an error occurs on the network layer, accepting that the content returned to the user cannot always be accurate, meaning that data displayed are shown temporarily in an old version. This is directly linked to the adoption of the Eventual Consistency paradigm for our distributed system which is better detailed further.

1.2.7.1. Replica Set

To ensure greater consistency and robustness in case of failures, a MongoDB replica set was implemented thanks to machines offered by the University of Pisa. This system involves using one primary replica on the primary machine (172.16.5.32) and two secondary replicas on the remaining machines (172.16.5.33/34). This allows, in case of failure of the primary server, to have the database always reachable making the service available even in the face of failures of multiple machines.

1.2.7.2. Replica Set settings

To create the replica set, the default settings were changed. Specifically, the settings for the replica set are:

```
{
 _id: 'myVacationRS',
 version: 102193,
 members: [
  {
    _id: 0,
    host: '172.16.5.32:27018',
    priority: 1,
  },
  {
    _id: 1,
    host: '172.16.5.33:27017',
    priority: 1,
  },
  {
    _id: 2,
    host: '172.16.5.34:27017',
    priority: 1,
  }
 ]
}
```

1.2.7.3. Replica Set Read Preferences

To ensure service availability, the reading preferences were changed from the default configuration. In fact, by default the reads occur only on the primary while in our case we want that if the primary is not available the queries will be performed on one of the secondaries.

```
myVacationRS [direct: primary] myvacation> db.getMongo().getReadPref()
ReadPreference {
 mode: 'primary',
 tags: undefined,
 hedge: undefined,
 maxStalenessSeconds: undefined,
 minWireVersion: undefined
}
```

1.2.7.4. Replica Set Write Concern

Although consistency is not a key parameter for application implementation choices, a best-effort approach was attempted by going to modify the settings related to the Write Concern of the replica set. By setting the write to "majority," we ensure that when a write is made it is propagated to at least one of the secondaries, thus in case of failures of the primary we limit data loss.

```
myVacationRS [direct: primary] myvacation> db.adminCommand(
... {
...   getDefaultRWConcern: 1 ,
... }
... )
{
  defaultReadConcern: { level: 'local' },
  defaultWriteConcern: { w: 'majority', wtimeout: 10000 },
  updateOpTime: Timestamp({ t: 1673359819, i: 1 }),
  updateWallClockTime: ISODate("2023-01-10T14:10:20.067Z"),
  defaultWriteConcernSource: 'global',
  defaultReadConcernSource: 'global',
  localUpdateWallClockTime: ISODate("2023-01-10T14:10:20.240Z"),
  ...
}
```

2. NoSQL Databases

We decided to use a Document Database, with MongoDB as DBMS, to store all main information about the entities of our application. We also decided to exploit a Graph Database, with Neo4J as DBMS, to store information about following among registered users and for storing the adv's preferences of our users.

2.1. MongoDB Design

Information about accommodations, activities, users, reservations and reviews are stored in MongoDB in five different collections.

2.1.1. Users Collection

The document of a user contains all the personal information of the user and also an array of all the reservations made by the user. This collection is used at the login, for checking if the credentials are correct. To satisfy the privacy non-functional requirement, the password is encrypted using BCrypt hashing. We decided to embed an array of reservation documents in

every user document (except for the Admin) due to the one-to-many relationship between these two entities. In fact, in this way, every time a user profile is displayed, the system has only access to one collection (user collection) and not also to the reservations collection. In fact, as told in the functional requirements, every user can view his reservations and the admin can view all the reservations.

Users's Collection Example

2.1.2. Accommodations Collection

The document of an accommodation contains all the general information of the accommodation and also an array of embedded documents describing the reviews created by the users. It has been decided to embed an array of review documents inside the accommodation's collection due to the one-to-many relationship between these two entities. In this way, each time an accommodation is displayed, users will be able to see the reviews of other users to the advertisement. We decided to embed the username of the user, allowing the general information of a review to be retrieved with a single read operation and avoiding join operations, at the cost of an additional small redundancy within the database. In fact, for each review we need to be able to see the username of the user who wrote it. We embedded the 5 most recent reviews and we decided to store all the reviews in another collection, since we will show them in the interface only if the user will require it, using the 'More reviews' button. We have chosen to do this because we allow the user of the application to view each advertisement with its detailed specifications and the reviews written for it. MongoDB keeps frequently accessed data in RAM. When the working set of data and indexes grow beyond the physical assigned RAM performance is reduced because disk accesses start to occur. To solve this issue and avoid having unbounded arrays that could exceed the maximum document size limit we decided to store a subset of the reviews in advertisement collections, and the older ones only in the reviews collection, as a backup. So we decided to embed the 5 most recent reviews in both cases to improve the performances of the application, while offering as many features as possible to the user. In this way we introduced redundancies and denormalized data, but at the same time, as just said, we improved the performances because in most cases we don't have to do join operations to see ads reviews. In fact, generally, a user reads only a few of the most recent reviews and we think that 5 reviews are, generally, enough.

```
_id: ObjectId('637bb26c945bed1e66467421')
name: "Dimore del Borgo Antico - La Zaffara"
description: "Nel centro storico di Monopoli, caratteristica Dimora finemente ristru…"
location: Object
  city: "Milano"
  address: "Lungomare Santa Maria"
  country: "Italy"
bedrooms: "2"
beds: "3"
price: "174"
mainPicture: BinData(0, 'LzlqLzRRUEVSWGhwWmdBQVNVa3FBQWdBQUFBS0FBOEJBZ0FTQUFBQWhnQUFBQkFCQWdBTEFBQUFtQUFBQUJvQkJRQUJBQUFBcEFB…')
pictures: Array
reviews: Array
guests: 5
minimumNights: "3"
propertyType: "Entire home"
approved: true
hostID: ObjectId('637ce1a04ed62608566c5fa8')
hostName: "Lucia"
```


Accommodations's Collection Example

2.1.3. Activities Collection

The document of an activity contains all the general information of the accommodation and, like the Accommodations Collection, also an array of embedded documents describing the reviews created by the users. It has been decided to embed the array of review documents inside the activity's collection due to the same reasons.


```
_id: ObjectId('63b812a40e81ef0a42af8ccf')
name: "COLORS: Adult Coloring Program"
description: "Wind down on Wednesday with art therapy at Capitol View Library. If yo…"
location: Object
  address: "5001 CENTRAL AVENUE SE"
  city: "Parma"
  country: "Italy"
price: 32
duration: 6
hostID: "63b7fcfdd4f5a1c5cbb417ea"
hostName: "Lorenzo"
reviews: Array
approved: true
mainPicture: BinData(0, 'LzlqLzRBQVFTa1pKUmdBQkFRRUFTQUJJQUFELzRnSWNTVU5EWDFCU1QwWkpURVVBQVFFQUFBSU1iR050Y3dJUUFBQnRiblJ5VWtk…')
```


Activities's Collection Example

2.1.4. Reviews Collection

If a user wants to see additional reviews, the application will consult the review's collection, in which we will find all general information of a review, again with a small redundancy to avoid join operations. This collection will also store the most recent 5 reviews which are already contained in the advertisement, in such a way that if an advertisement has more than 5 reviews, each time a new review arrives we don't need to move the oldest review among the previous 5 in this collection. This will improve our performances, with the cost of another small redundancy.

```
_id: ObjectId('63bd34222889320636398l4b')
score: 5
description: "Excellent"
userID: ObjectId('637ce1a04ed62608566c5fa9')
destinationID: ObjectId('63b812a40e81ef0a42af8e97')
reviewer: "Bradley"
```

Reviews's Collection Example

2.1.5. Reservations Collection

In this collection are contained all the reservations made by users. The document of a reservation contains all the general information of the reservation and also the type of advertisement reserved. It has been decided to embed the type inside the reservation's collection due to the one-to-many relationship between these two entities. In this way we can retrieve the type of the destination, and all other information of the reservation, with a single read operation and avoiding join operations, at the cost of an additional small redundancy within the database. In fact, doing this, we have improved very much the computation of some stats and analytics related to the reservations, and we have also made some easier checks on the frontend and backend sides. This collection is very important because it allows us to retrieve all the information and stats related to the reservations with a single read operation, avoiding the access to all the embedded documents on the users collections.

```
_id: ObjectId('63b83374a88d87c8550d347c')
userID: ObjectId('637ce1a04ed62608566c5fad')
startDate: 2020-08-15T00:00:00.000+00:00
endDate: 2020-08-31T00:00:00.000+00:00
totalExpense: 750
destinationID: ObjectId('637bb2bc945bed1e6646747a')
city: "Prato"
destinationType: "accommodation"
hostID: ObjectId('637ce1a04ed62608566c5fab')
```

2.1.5. Queries Analysis

Starting from the analysis of the use case, we determined the following read and write queries involving our Document Database. Each operation is presented with its expected frequency and cost.

2.1.5.1. Read Operations (Unregistered User – Registered User)

Operation	Expected Frequency	Cost
Show User Information	Medium	Low (1 Read)
Show Accommodation/Activity Information	Very High	Low (1 Read)
Show First Accommodation/Activity Reviews	Very High	Low (1 Read)

Show All Accommodation/Activity Reviews	Low	High (Many Reads)
Show Filtered Accommodations/Activities	Very High	Very High (Many Reads)
Check login attempt	High	Low (1 Read)
Retrieve User Information at Login	High	Low (1 Read)
Show Own Reservations	Medium	Medium (1 Read)
Show Own Accommodations/Activities	Medium	High (Many Reads)
Show Top 3 Accommodations/Activities/Cities	Medium	High (Aggregation)
Show Total Reservations per City per Month	Medium	High (Aggregation)

2.1.5.2. Write Operations (Unregistered User – Registered User)

Operation	Expected Frequency	Cost
Registration to the platform	Medium	Medium (Document Insertion)
Add a Review	Low	Medium (Document Insertion)
Insert Accommodation/Activity	Low	Medium (Document Insertion)
Reserve an Accommodation/Activity	High	Medium (Document Insertion)
Update Own Information	Very Low	Medium (Few Attribute Writes)
Update Reservation	Very Low	Medium (2 Attribute Writes)
Update Accommodation/Activity	Low	Medium (Few Attribute Writes)
Delete Own Accommodation/Activity	Very Low	Medium (Document Deletion)
Delete Own Review	Very Low	Medium (Document Deletion)
Delete Reservation	Very Low	Medium (Document Deletion)

2.1.5.3. Read Operations (Admin)

Operation	Expected Frequency	Cost
Search User	Low	Low (1 Read)
Show Accommodations/Activities To Be Approved	High	High (Many Reads)
Show Registered Users per Month	High	High (Aggregation)

Show Number of Accommodations/Activities	High	High (Aggregation)
Show Average Price Accommodations/Activities per City	High	High (Aggregation)

2.1.5.4 Write Operations (Admin)

Operation	Expected Frequency	Cost
Delete User	Low	Very High (Many Document Deletions)
Confirm Accommodation/Activity	High	Low (Update one attribute)

2.1.5. CRUD Operations

In this section, we can find CRUD Operations

2.1.5.1. Create

Operation	
Create Accommodation	collection.insert_one(accommodation.getDictToUpload());
Create Activity	collection.insert_one(activity.getDictToUpload());
Create Reservaion	collection.insert_one(reservation.getDictToUpload());
Create Review	collection.insert_one(review.getDictToUpload());
Create User	collection.insert_one(user.getDictToUpload());

2.1.5.2. Read

Operation	
Get Activities From Id List	collection.find({"_id": {"$in": idList}});
Get Activity From Id	collection.find_one({“_id”: ObjectId(activityID)});
Get Occupied Activities	query = {'destinationType': 'activity', 'startDate': startDate}; occupiedActivitiesID = collection.distinct("destinationId", query);
Get Filtered Activities	query = {'location.city': city, '_id': {'$nin': occupiedActivitiesID}, 'approved': True}; projection = {"reservations": 0, "reviews": 0}; activities = list(collection.find(query, projection).sort('_id', 1).limit(int(os.getenv("PAGE_SIZE"))));
Get Activities By UserID	collection.find({"hostID": ObjectId(userID)});
Get Accommodations From Id List	collection.find({"_id": {"$in": idList}});

Get Accommodation From Id	collection.find_one({“_id”: ObjectId(accommodationID)});
Get Occupied Accommodations	query = {'destinationType': 'accommodation', '$or': [{'$or': [{'$and': [{'startDate': {'$lte': endDate}}, {'startDate': {'$gte': startDate}}]}, {'$and': [{'endDate': {'$lte': endDate}}, {'endDate': {'$gte': startDate}}]}]}, {'$and': [{'startDate': {'$lte': startDate}}, {'endDate': {'$gte': endDate}}]}]} occupiedAccommodationsID = collection.distinct("destinationID", query,)
Get Filtered Accommodations	query = {'location.city': city, 'guests': {'$gte': nrGuests}, '_id': {'$nin': occipiedAccomodationsID}, 'approved': True}; projection = {"reservations": 0, "reviews": 0}; accommodations = list(collection.find(query, projection).sort('_id', 1).limit(int(os.getenv("PAGE_SIZE"))));
Get Accommodations By UserID	collection.find({"hostID": ObjectId(userID)})
Get Announcements To Approve	query = {“approved”: False}; projection = {"pictures": 0, "mainPicture": 0}; items = list(collection.find(query, projection) .sort("_id", 1) .limit(int(os.getenv("ADMIN_PAGE_SIZE"))));
Get Announcement To Approve By ID	collection.find_one({"_id": ObjectId(announcementID)});
Get Reservations By User	collection.find({"userID" : ObjectId(userID)});
Get Review From ID	collection.find_one({"_id": ObjectId(reviewID)});
Get Review From DestinationID	collection.find({"destinationID": ObjectId(destinationID)});
Get User From ID	collection.find_one({"_id" : ObjectId(userID)});

2.1.5.3. Update

Operation	
Update Accommodation	collection.update_one({"_id": ObjectId(accommodationID)}, {"$set": accommodation});
Add Review to an Accommodation	collection.update_one({"_id": ObjectId(review.destinationID)},

	{"$push": {"reviews": review.getDictForAdvertisement()}});
Update Activity	collection.update_one({"_id": ObjectId(activityID)}, {"$set": activity});
Add Review to an Activity	collection.update_one({"_id": ObjectId(review.destinationID)}, {"$push": {"reviews": review.getDictForAdvertisement()}});
Approve Announcement	collection.update_one({"_id": ObjectId(announcementID)}, {"$set": {"approved": True}});
Remove Approval Announcement	collection.update_one({"_id": ObjectId(announcementID)}, {"$set": {"approved": False}});
Operation	
Update Reservation	collection.update_one('_id': ObjectId(reservationID), {"$set": {'startDate':newStartDate, 'endDate': newEndDate, "totalExpense" : newTotalExpense}})
Update Reservation to an User	collection.update_one({"_id" : ObjectId(user["_id"]) , "reservations._id" : ObjectId(reservation["_id"])} , {"$set" : {"reservations.$.startDate" : newStartDate }})
Add Reservation to an User	usersCollection.update_one({"_id" : ObjectId(reservation.userID)} , {"$push" : {"reservations" : reservation.getDictForUser()}});
Remove Reservation to an User	usersCollection.update_one({"_id" : ObjectId(reservation["userID"])} , {"$pull" : {"reservations": {"_id" : ObjectId(reservationID)}}});
Update User	collection.update_one({"_id" : ObjectId(userID)} , {"$set" : user});

2.1.5.4. Delete

Operation	
Delete Accommodation	collection.delete_one({"_id":ObjectId(accommodationID)});
Delete Activity	collection.delete_one({"_id": ObjectId(activityID)});
Delete Review	collection.delete_one({"_id" : ObjectId(reviewID)});
Delet Reservation	collection.delete_one({"_id" : ObjectId(reservationID)});
Delete User	collection.delete_one({"_id": ObjectId(userID)});

2.1.6. Analytics and Statistics

This section contains different aggregations where our Document Database is involved for analytics and statistics purposes.

2.1.6.1. Reservations by month

The following aggregation returns the host reservations number by city and month.

```
result = list(collection.aggregate(([
            {"$match": {"hostID": ObjectId(user["_id"])}},
            {"$group": {
               "_id": {
                  "city": "$city",
                  "month": {"$month": "$startDate"}
               },
               "count": {"$count": {}}
            }},
            {"$project": {"total": "$count", "month": "$_id.month",
                        "_id": 0, "city": "$_id.city"}},
            {"$group": {
               "_id": "$city",
               "stats": {
                  "$push": {
                     "month": "$month",
                     "total": "$total"
                  }
               }
            }},
            {"$project": {"_id": 0, "city": "$_id", "stats": "$stats"}}
         ]))
```

2.1.6.2. Monthly Subscribed Users

The following aggregation returns the number of the subscribed users for each month.

```
result = list(collection.aggregate([
            {"$match":
             {"$expr":
              {
                 "$eq": [{"$year": "$registrationDate"}, year]
```

```
        }

        }

        },

        {

          "$group": {

              "_id": {"$month": "$registrationDate"},

              "users": {

                 "$count": {}

              }

          }

        },

        {'$sort': {'users': -1}},

        {

          "$project": {"month": "$_id", "users": "$users", "_id": 0}

        }]))
```

2.1.6.3. Top three cities for each month

The following aggregation returns the three most booked cities of the last month.

```
result = list(collection.aggregate([
            {"$match":
             {"$expr":
              {
                 "$eq": [{"$month": "$startDate"}, month]
              }
              }
             },
            {"$group": {"_id": "$city", "count": {"$sum": 1}}},
            {"$sort": {"count": -1, "_id": 1}},
            {"$project": {"city": "$_id", "_id": 0}},
            {"$limit": 3}
        ]))
```

2.1.6.4. Accommodations Average Costs

The following aggregation returns the accommodation average price for each city.

```
result = list(collection.aggregate([
        {"$match" :  {"_id" : {"$in" : approvedAccommodations}}},
        {"$group":
         {
            "_id": "$location.city",
            "averageCost": {"$avg": "$price"}
         }
         },
        {'$sort': {'averageCost': -1}},
        {"$project": {"_id": 0, "city": "$_id", "averageCost": {"$round": ["$averageCost", 2]}}}


    ]))
```

2.1.6.5. Activities Average Costs

The following aggregation returns the activities average price for each city.

```
result = list(collection.aggregate([
        {"$match" :  {"_id" : {"$in" : approvedActivities}}},
        {"$group":
         {
            "_id": "$location.city",
            "averageCost": {"$avg": "$price"}
         }
         },
        {'$sort': {'averageCost': -1}},
        {"$project": {"_id": 0, "city": "$_id", "averageCost": {"$round": ["$averageCost", 2]}}}


    ]))
```

2.1.6.6. Top 3 Accommodations/Activities

The following aggregations returns the three most booked activities and accommodations of all times.

```
    serializedAccommodations = []
     serializedActivities = []
     approvedAccommodations = AccommodationManager.getApprovedAccommodationsID()
```

```
    accommodationsResult = list(collection.aggregate([
        {"$match": {"destinationType": "accommodation" , "destinationID" : {"$in" : approvedAccommodations}}},
        {"$group": {"_id": "$destinationID", "count": {"$sum": 1}}},
        {"$sort": {"count": -1, "_id": 1}},
        {"$limit": 3},
        {"$project": {"count": 0}}
    ]))

    for accommodation in accommodationsResult:
        stringId = str(accommodation["_id"])
        accommodation["_id"] = stringId
        serializedAccommodations.append(accommodation)

    approvedActivities = ActivityManager.getApprovedActivitiesID()
    activitiesResult = list(collection.aggregate([
        {"$match": {"destinationType": "activity" , "destinationID" : {"$in" : approvedActivities}}},
        {"$group": {"_id": "$destinationID", "count": {"$sum": 1}}},
        {"$sort": {"count": -1, "_id": 1}},
        {"$limit": 3},
        {"$project": {"count": 0}}
    ]))

    for activity in activitiesResult:
        stringId = str(activity["_id"])
        activity["_id"] = stringId
        serializedActivities.append(activity)

    return {"accommodationsID": serializedAccommodations,
            "activitiesID": serializedActivities}
```

2.1.6.7. Total reservations

The following aggregation returns the total number of reservations.

```
        result = list(collection.aggregate([
            {
                '$count': 'totalReservation'
            }
        ]))
    else:
        result = list(collection.aggregate([
```

```
            {
                '$match': {
                    'hostID': ObjectId(user["_id"])
                }
            }, {
                '$count': 'totalReservation'
            }
        ]))
```

2.1.6.8. Total Accommodations/Activities

The following aggregations returns the total number of the published accommodations and activities of all users if the authenticated user is an admin; otherwise returns the total number of the published accommodations and activities of the logged user if host.

```
if (user["role"] == "admin"):
        resultAcc = list(accommodationCollection.aggregate([
            {
                '$count': 'totalAccommodations'
            }
        ]))
        resultAct = list(activityCollection.aggregate([
            {
                '$count': 'totalActivities'
            }
        ]))
    else:
        resultAcc = list(accommodationCollection.aggregate([
            {
                '$match': {
                    'hostID': ObjectId(user["_id"])
                }
            }, {
                '$count': 'totalAccommodations'
            }
        ]))
        resultAct = list(activityCollection.aggregate([
            {
                '$match': {
                    'hostID': ObjectId(user["_id"])
                }
            }, {
                '$count': 'totalActivities'
```

```
        }
    ]))
result = {
    "totalAccommodations": resultAcc[0]["totalAccommodations"] ,
    "totalActivities": resultAct[0]["totalActivities"]
}
```

2.1.7. Indexes

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

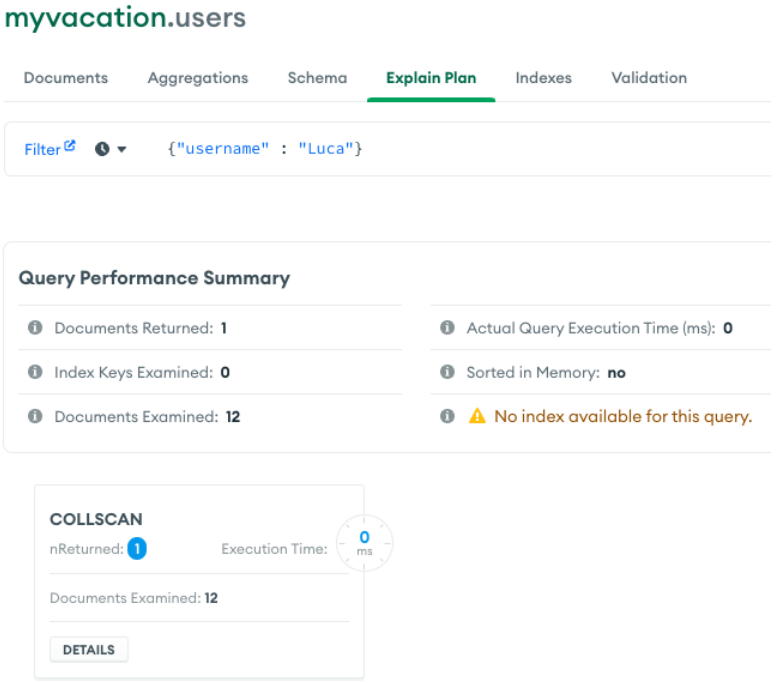
Collection	Index name	Index field(s)	Properties
Users	Username	Username	Unique
Reservations	ReservedAccommodations/ Activities	startDate, endDate	Not unique
Accommodations	CityAccommodation	location.city	Not unique
Activities	CityActivity	location.city	Not unique
Reviews	AdvReviewed	destinationID	Not unique
Accommodations	Approved Accommodation	approved	Not unique
Activities	ApprovedActivity	approved	Not unique

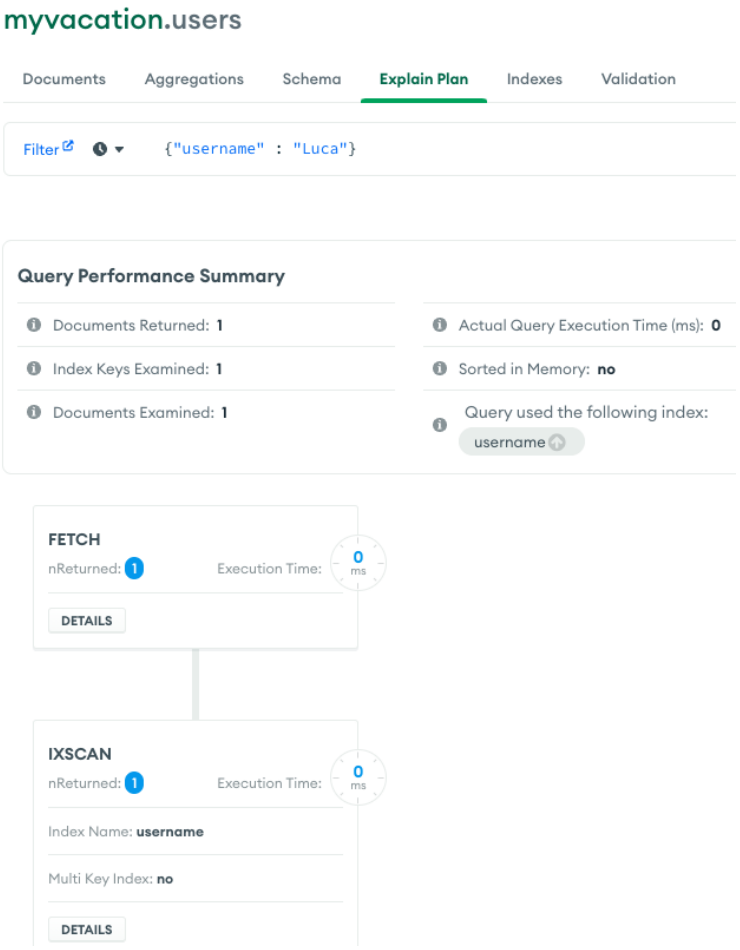
2.1.7.1 Indexes Performance Analysis

The tests carried out on the Document DBMS MongoDb Compass are done using the “Explain Plan” section. Cannot be shown a huge improvement in terms of Execution Speed in all the collections because some of them have few documents, but can be shown an important improvement in terms of Documents Examined in all of them.

2.1.7.1.1. Username Index

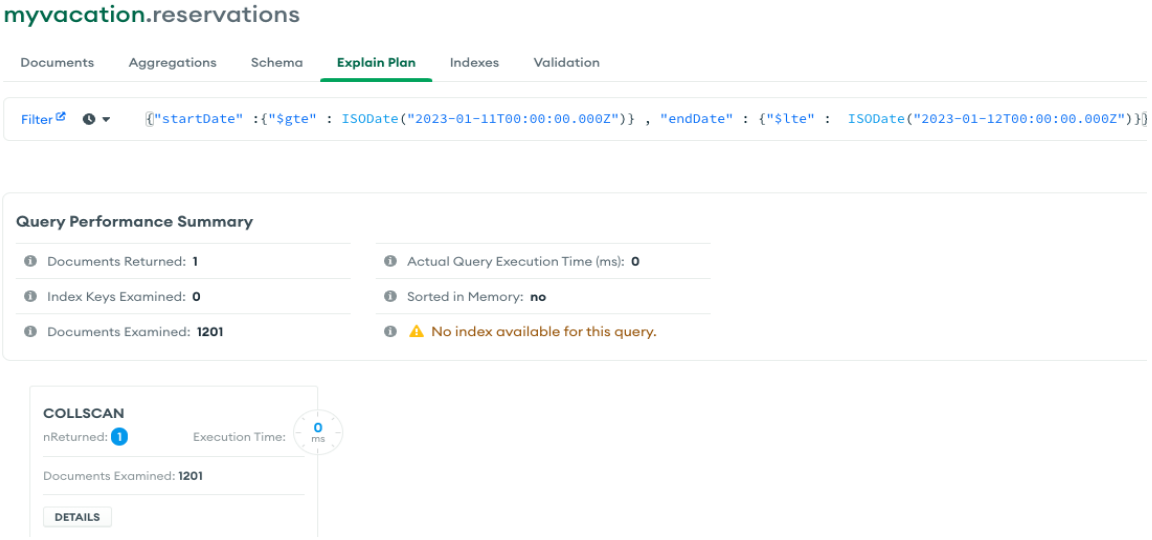
In this performance analysis we can see the improvements on equality matches using the username Index. Using the index, only documents that match the value or regular expression are examined, with an execution time that strongly decreases. This can be very useful every time that a user tries to log in, and also when the admin searches an user filtering by username.

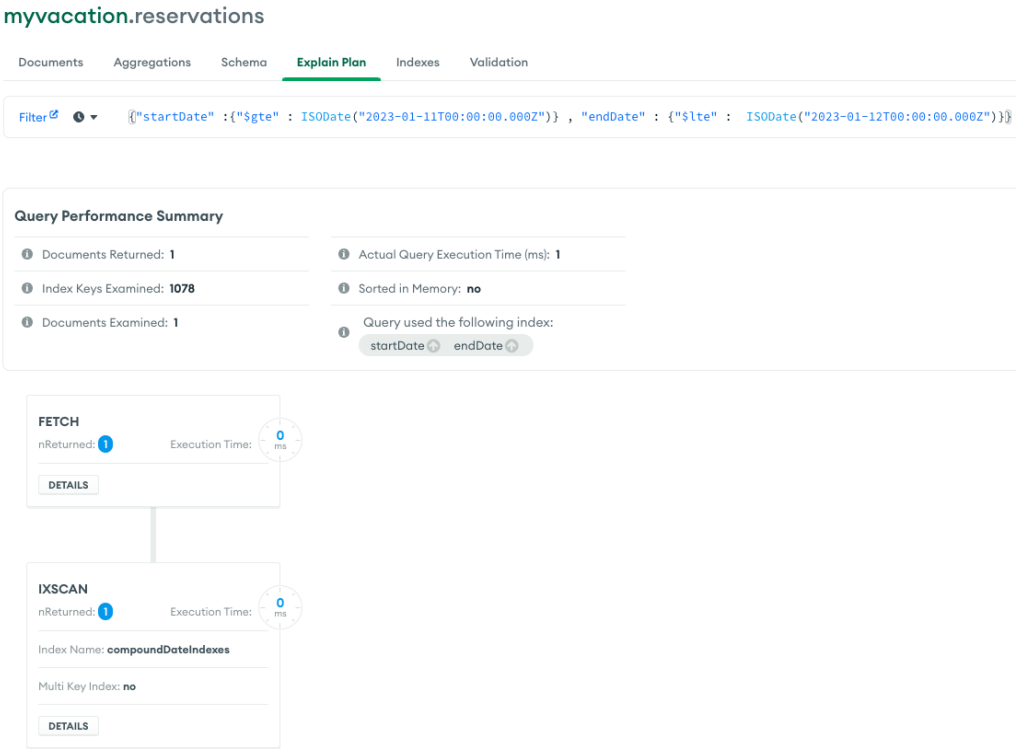




2.1.7.1.2. ReservedAccommodations/Activities Index

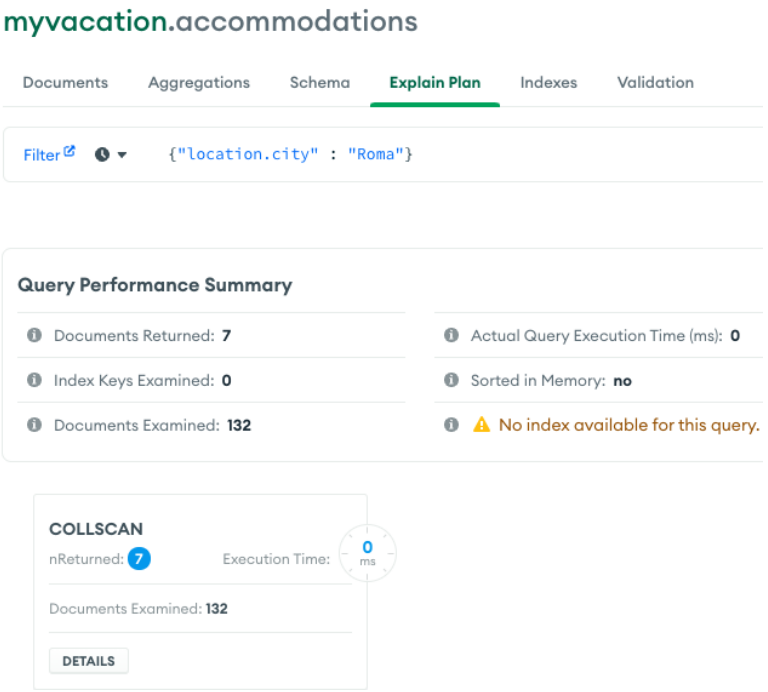
This index is used very intensively during the Search and Reserve functionalities, two of the most used and important of the system. In fact, thanks to this index, we can optimise the searching of occupied accommodations/activities in order to not create conflicts on reservations.

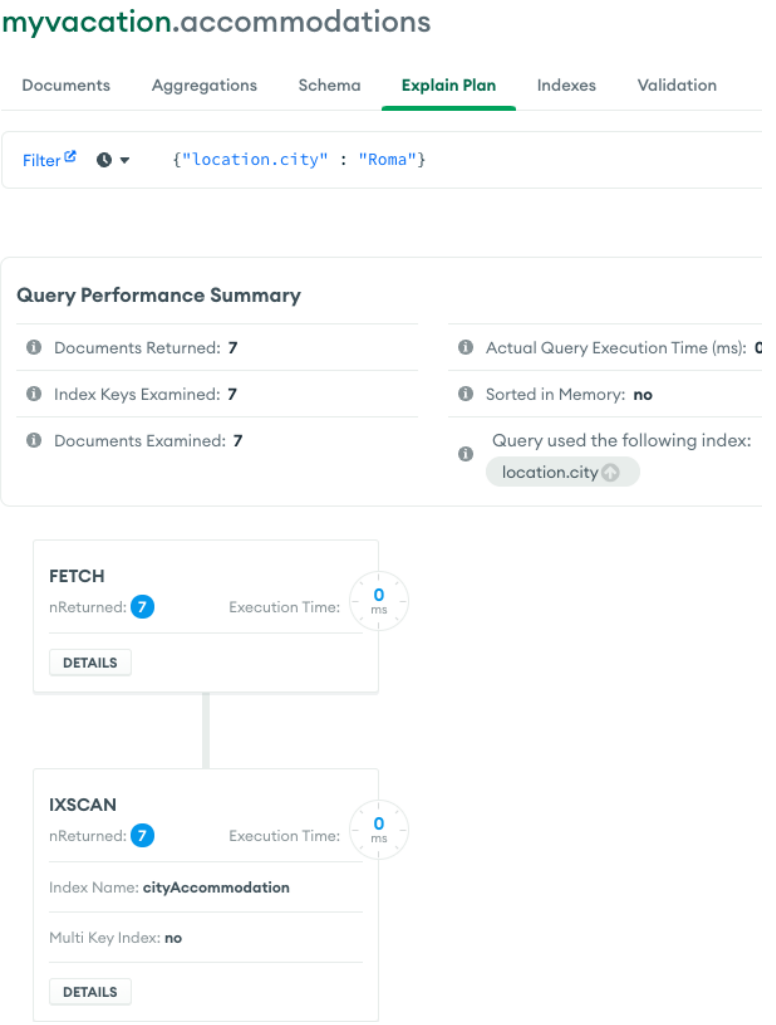




2.1.7.1.4. CityAccommodation Index

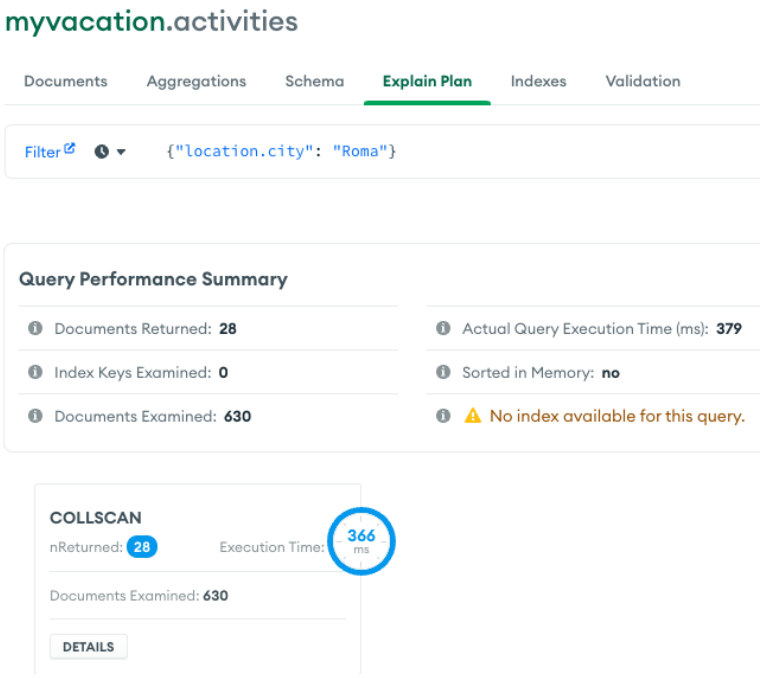
In this performance analysis we can see the improvements on equality matches using the CityAccommodation Index. In fact, this index is used intensively during the Search functionality, one of the most used and important of the system. During searching an user can filter by the City of the accommodation.

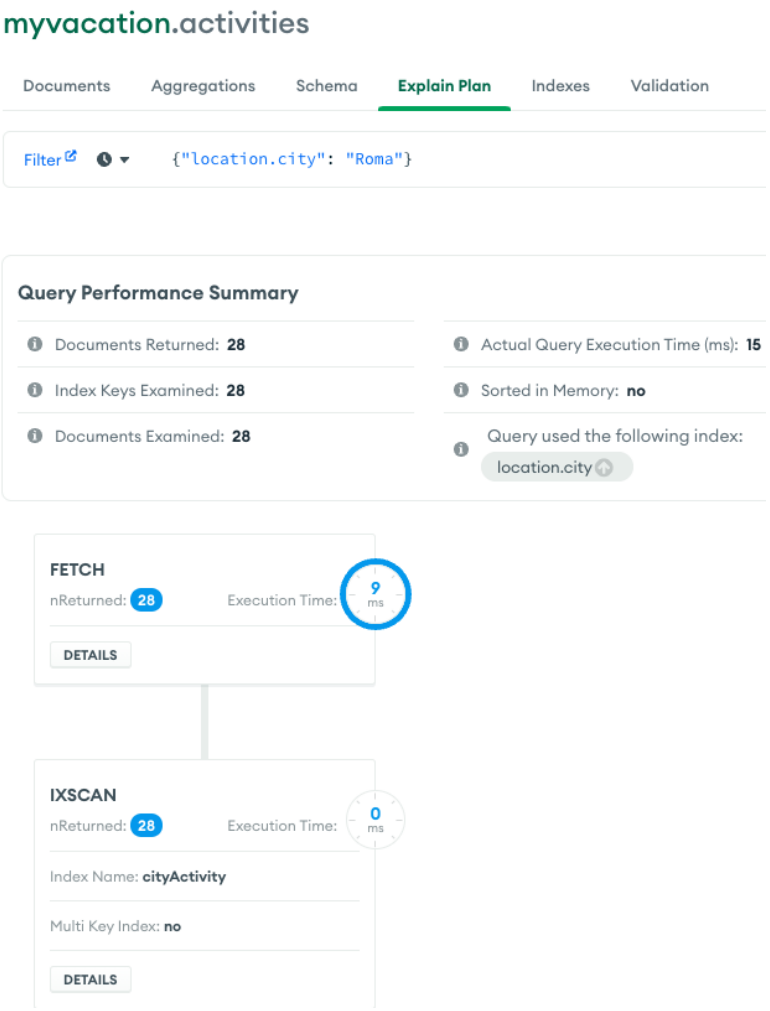




2.1.7.1.5. CityActivity Index

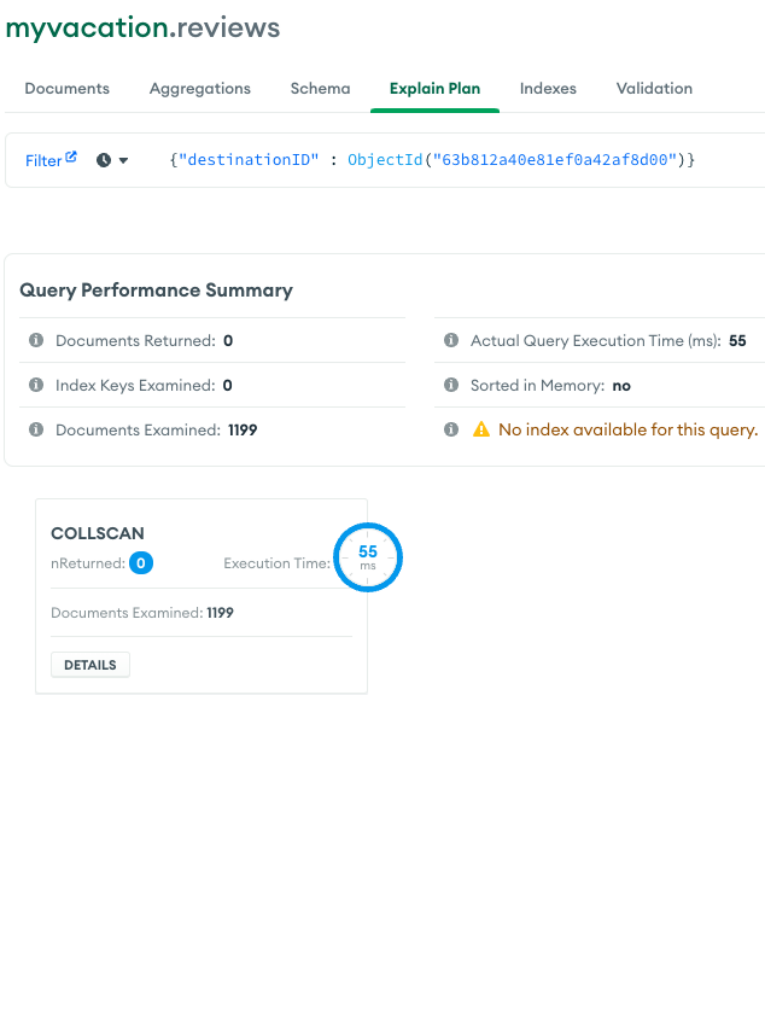
In this performance analysis we can see the improvements on equality matches using the CityActivity Index. The importance of this index is the same as the CityAccommodation Index.

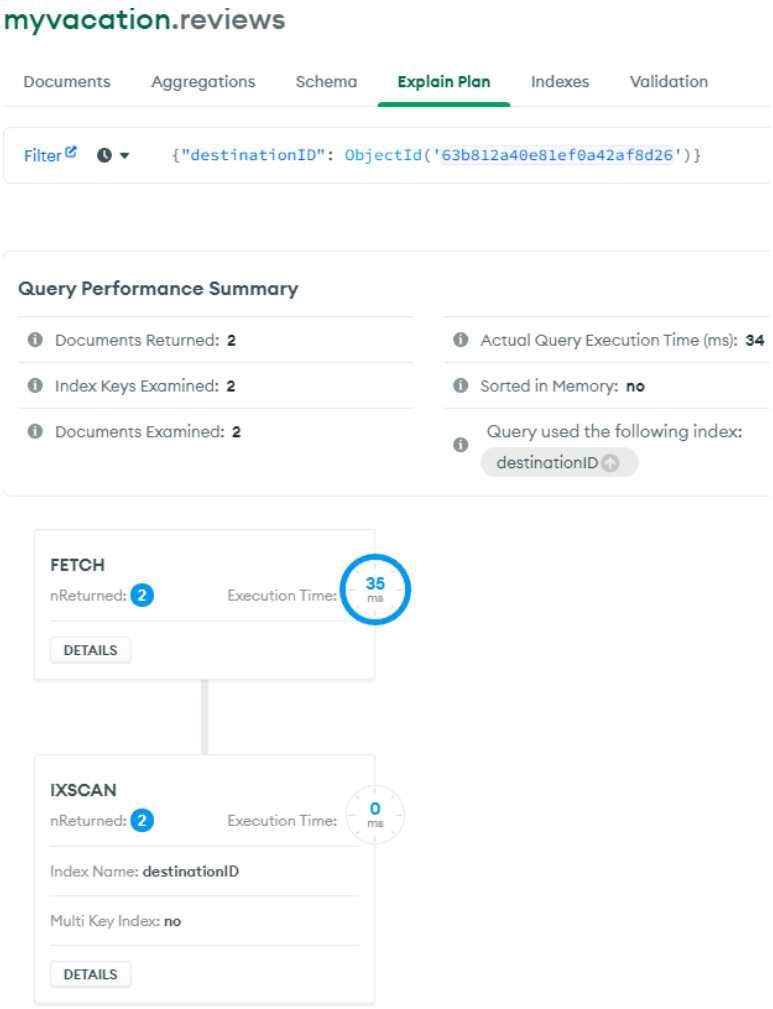




2.1.7.1.6. AdvReviewed Index

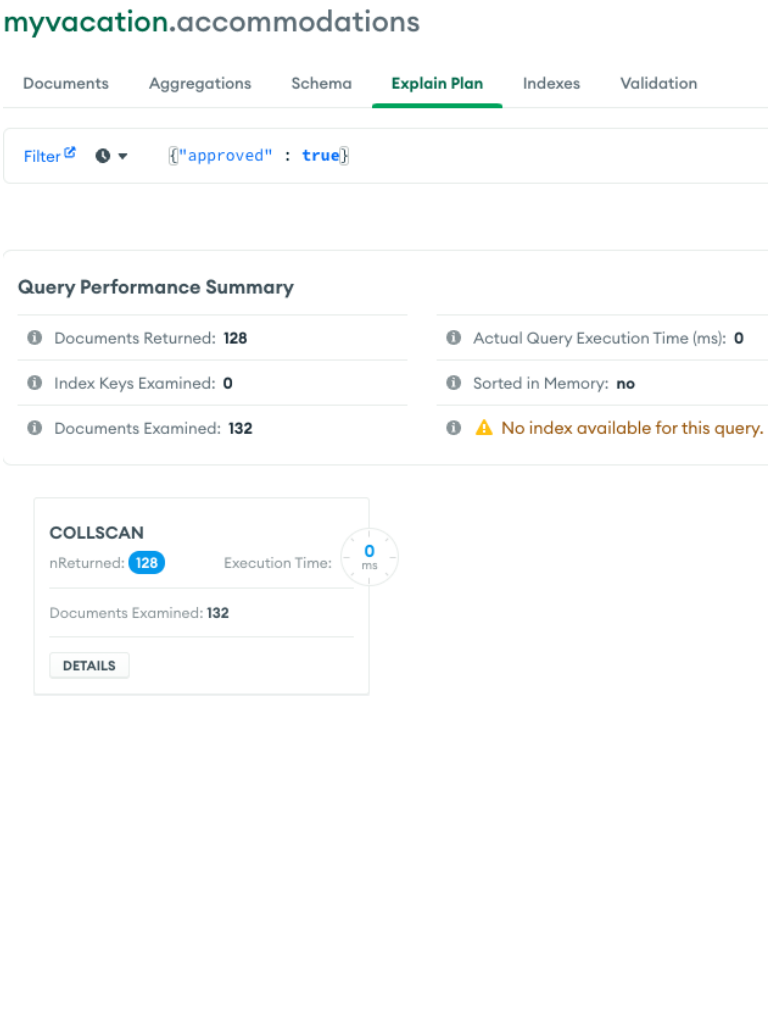
In this performance analysis we can see the improvements on equality matches using the AdvReviewed Index. This index optimises the finding of all the reviews when a user clicks the “More reviews” button present in the ad's viewing.

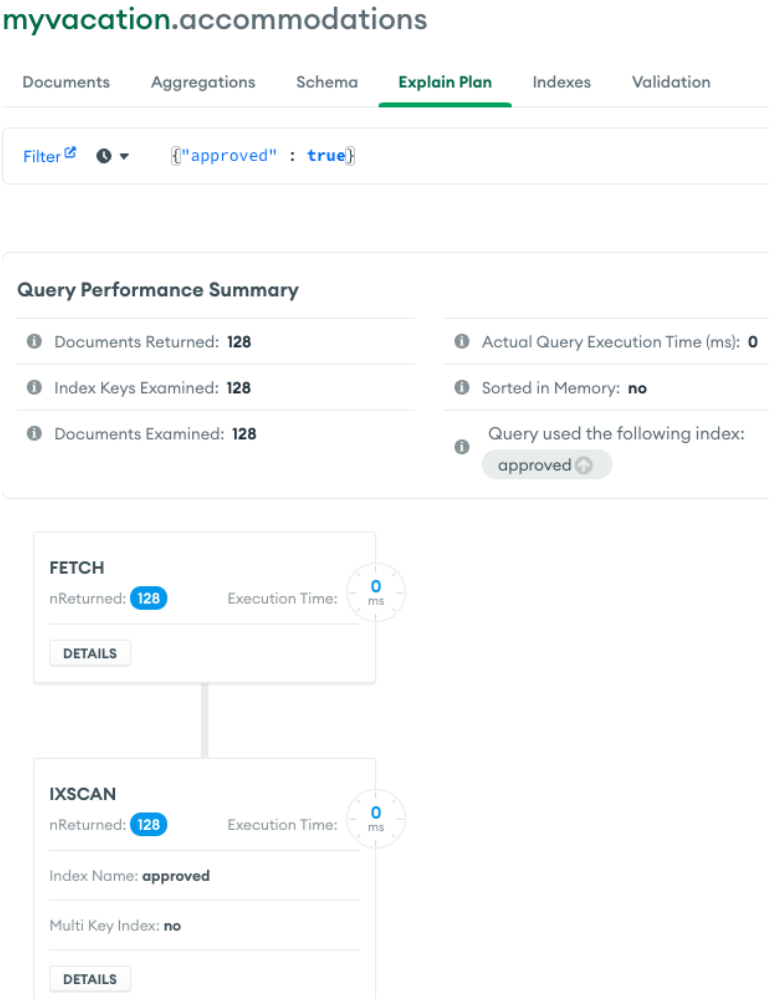




2.1.7.1.7. ApprovedAccommodation Index

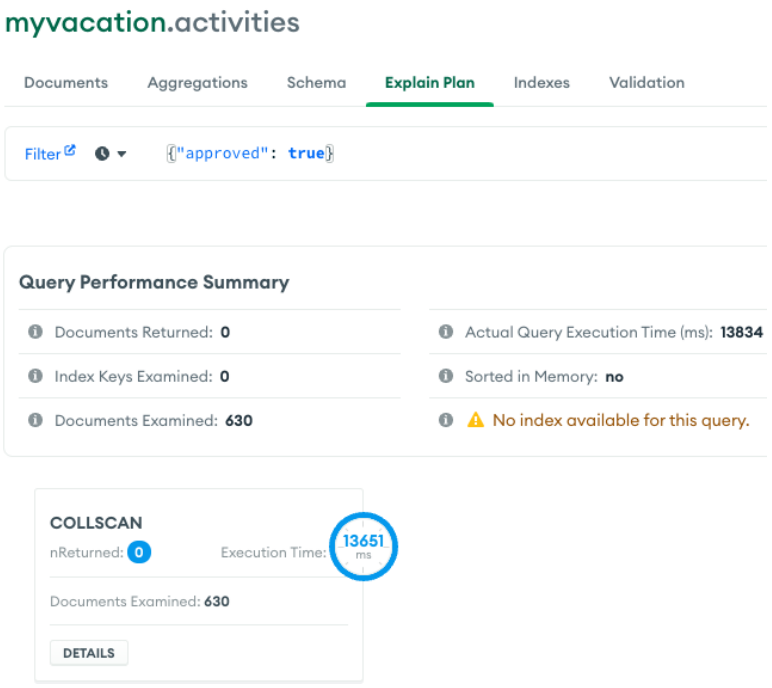
In this performance analysis we can see the improvements on equality matches using the ApprovedAccommodation Index. This index optimises the finding of all approved accommodations. This index will be also very useful during the Search functionality and all the analytics/stats regarding accommodations.

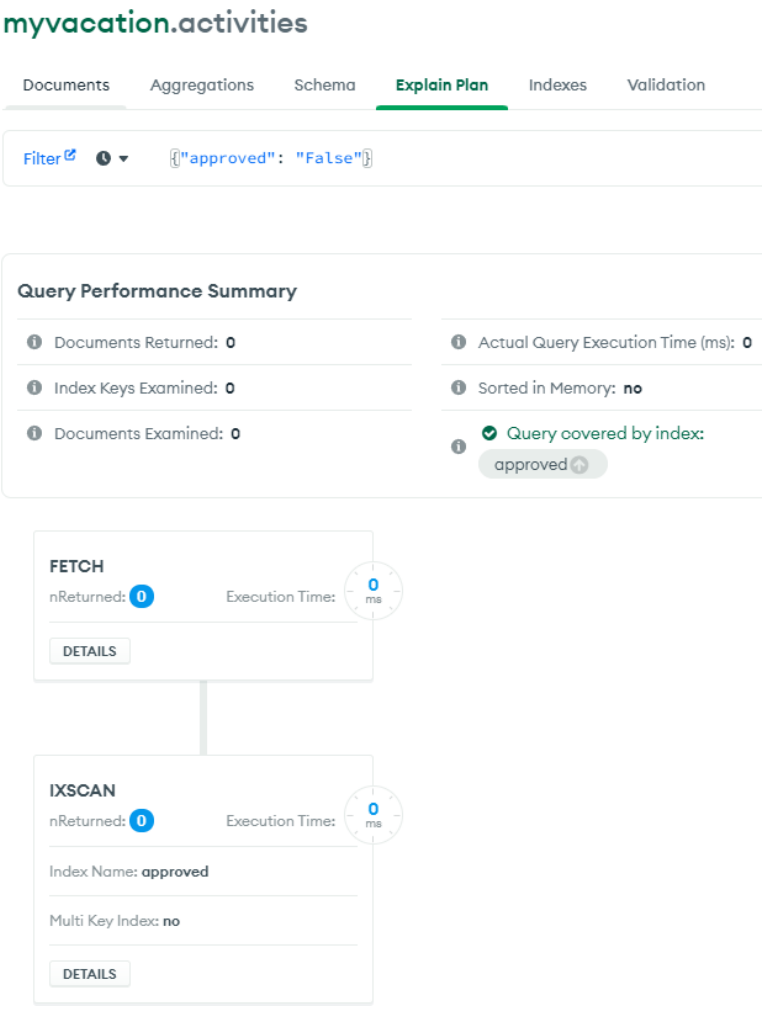




2.1.7.1.8. ApprovedActivity Index

In this performance analysis we can see the improvements on equality matches using the ApprovedActivity Index. This index optimises the finding of all approved activities. This index will be also very useful during the Search functionality and all the analytics/stats regarding activities.





This query examines 0 Documents because all of them have the “approved” field equal to True, so we have reached the maximum optimisation.

2.1.8. Sharding Data

For the moment, sharding on Mongodb is not convenient due to the reduced set size (GBs, not TBs), but, imagining that the size of the collections increases significantly, we thought of proposing a hypothetical horizontal partitioning, based on:

- The distribution of reads and writes.
- The size of the chunks.
- The number of chunks each query uses

To implement the sharding, we thought about selecting three different sharding keys, one for 3 of our 5 collections of our document database:

Collection	**Sharding Key**	**Description**
Accommodations	_id	Field automatically generated by MongoDB
Activities	_id	Field automatically generated by MongoDB
Users	_id	Field automatically generated by MongoDB

We choose as a sharding partitioning method the hashing, so to ensure a good load balance among servers. The hash function returns an hash value to determine where to place a document, resulting in evenly distributed data among the shards. As for the other two collections, Reviews and Reservations, we thought of:

1. Shard reviews associated with an accommodation/activity into the same shard of the accommodation/activity;
2. Shard reservations associated with an accommodation/activity into the same shard of the accommodation/activity.

The first allows us to optimise the retrieval of reviews of an accommodation after a user has pressed the "More Reviews" button.
The second one allows us to optimise the check regarding the occupation of the accommodation/activity. Furthermore, it is easy to recover the reservations of each user as each user has his own reservations embedded in the document.

2.2. Neo4J Design

We used Neo4j to store information about following among users and preferences of users on the reserved advertisements, information that we can't find on our document DB.

2.2.1. Nodes

In Neo4j we stored just the necessary information to perform the queries we wanted to execute and to display previews of the entity in the application. All the entities hold always enough information to find the corresponding entity and the missing values in the other database in necessary, in particular:

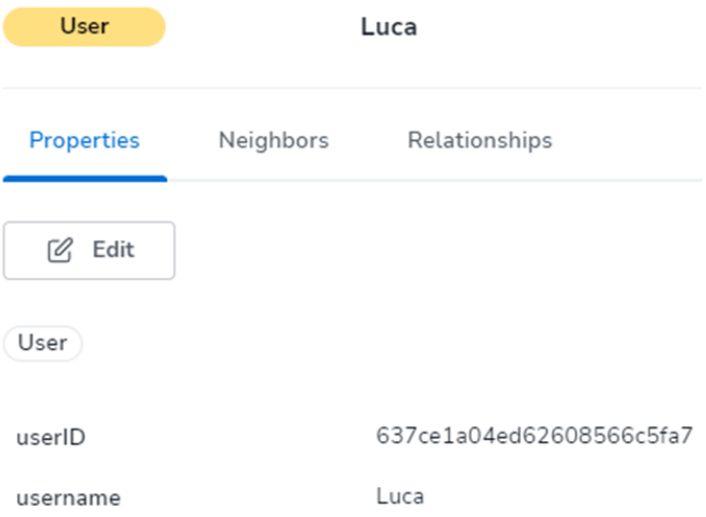
- A field userID containing the MongoDB ObjectId was added to Neo4j;
- A field accommodationID containing the MongoDB ObjectId was added to Neo4j;
- A field activityID containing the MongoDB ObjectId was added to Neo4j.

The complete list of entities we used with Neo4j is the following:

- Accommodation: {accommodationID: string, approved: boolean, name: string}
- Activity: {activityID: string, approved: boolean, name: string}
- User: {userID: string, username: string}

2.2.1.1. User Node

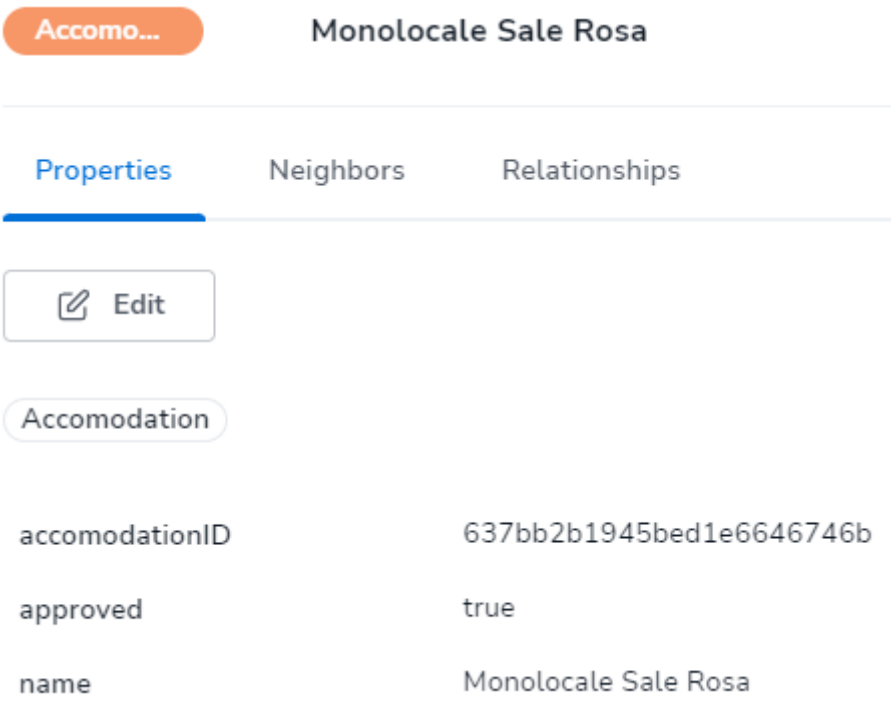
The user node will maintain as properties just the essential information to show a preview of the user. In fact a user can just see essential information about recommended users, and only once he navigates to the profile of a recommended user, the user can see the other user's information.



User Node Properties Example

2.2.1.2. Accommodation Node

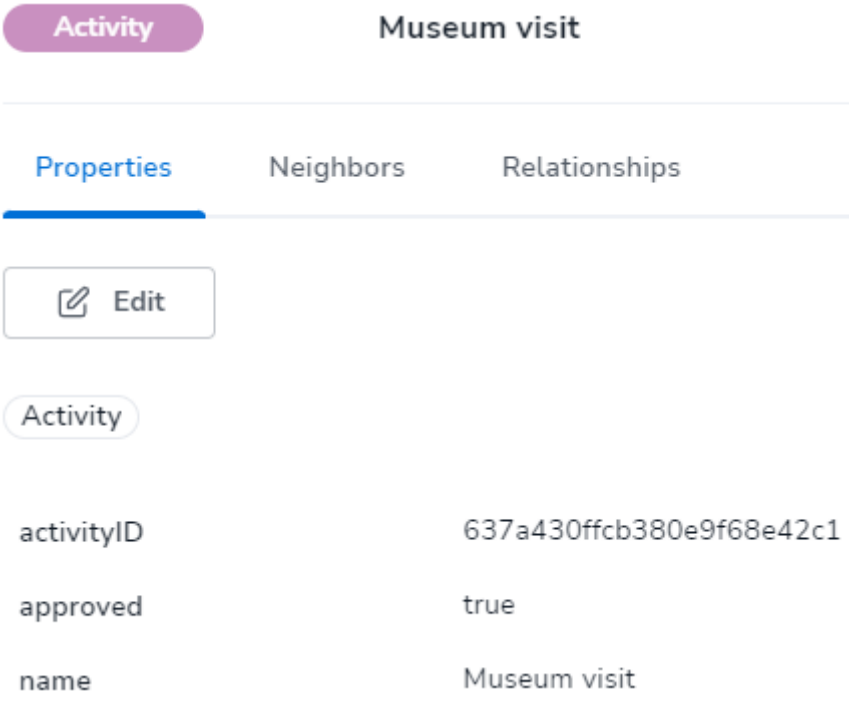
The accommodation node will store as properties just the essential information to show a preview of the recommended accommodations to the user. Once interested, the user will be able to access the document database to obtain more information.



Accommodation Node Properties Example

2.2.1.3. Activity Node

The activity node will store as properties just the essential information to show a preview of the recommended activities to the user. Once interested, the user will be able to access the document database to obtain more information.



Activity Node Properties Example

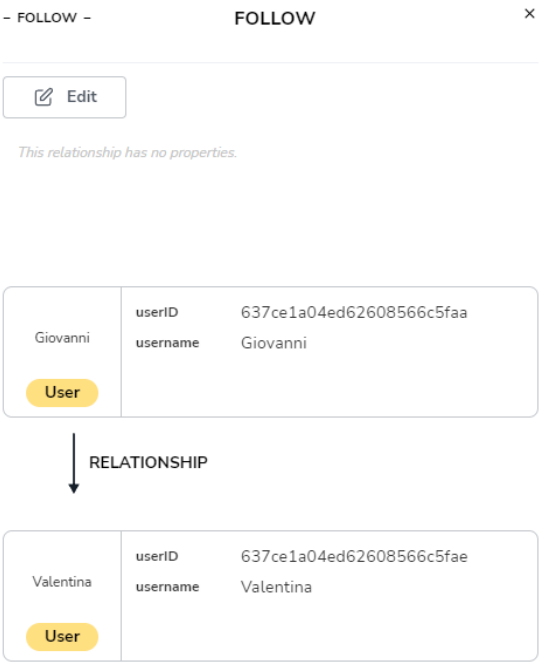
2.2.2. Relations

The relationships involved are:

- User - [:FOLLOW] -> User: which represents a user following another user.
- User - [:LIKE] -> Accommodation: which represents a user liking an accommodation.
- User - [:LIKE] -> Activity: which represents a user liking an activity.

2.2.2.1. Follow

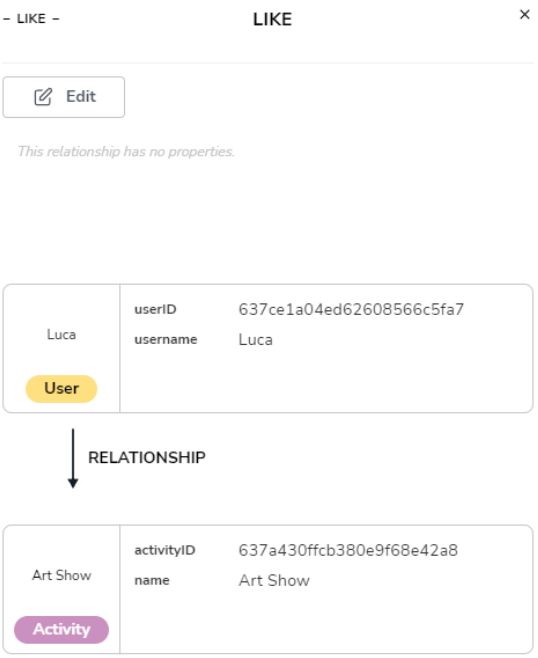
The directed relation FOLLOW will be added between users if one of them wants to follow one other. Each user, excluding the admin, can follow another user. The relation has not a status property. The relation will also be used to enhance the recommendation of users and accommodations/activities.



Follow Relation Properties Example

2.2.2.2. Like

The directed relation LIKE will be added between a user and an accommodation/activity if the user expresses a preference (like/unlike) on the accommodation/activity. The relation will be used to realise the advertisement's recommendation system and also to show, for example, the common liked accommodations/activities between two users.



Like Relation Properties Example

2.2.3 Queries Analysis Neo4J

The following read and write queries involve the Graph Database. Each operation is presented with its expected frequency and cost.

2.2.3.1 Read Operations (Unregistered User – Registered User)

Operation	Expected Frequency	Cost
Check if the Logged User is Following an User	Low	Low (1 Read)
Show Followed Users	Medium	Medium
Check if User Likes a Destination	High	Low
Show Liked Accommodations/Activities	Medium	High
Show Recommended Accommodations/Activities	Medium	High
Show Recommended Users	Medium	High
Show Total Likes of an Accommodation/Activity	High	Low (1 Read)
Show Common Liked Accommodations/Activities	Medium	Medium

2.2.3.2 Write Operations (Unregistered User – Registered User)

Operation	Expected Frequency	Cost
Create a User Node	Medium	Low (Node Creation)
Delete a User Node	Low	Medium (Nodes and Relations Deletion)
Create an Accommodation/Activity Node	Medium/High	Low (Node Creation)
Delete an Accommodation/Activity Node	Low	Medium (Node Deletion)
Update an Accommodation/Activity Node	Medium	Medium (Node Update)
Follow a User	Medium	Low (1 Relation Creation)
Unfollow a User	Medium	Low (1 Relation Deletion)
Like an Accommodation/Activity	Medium	Low (1 Relation Creation)
Unlike an Accommodation/Activity	Medium	Low (1 Relation Deletion)

2.2.4. CRUD Operations

2.2.4.1. Create

Operation	
Create an Accommodation	query = "CREATE (a:Accommodation {accommodationID: '%s', name: '%s'})" %(accommodationNode.accommodationID, accommodationNode.name);
Create an Activity	query = "CREATE (a:Activity {activityID: '%s', name: '%s'})" % (activityNode.userID, activityNode.name);
Create an User	query = "CREATE (u:User {userID: '%s', username: '%s'})" % (userNode.userID, userNode.username);
Create a Follow Relation	query = "MATCH (u:User {userID: '%s'}) MATCH (u1: User{userID : '%s'}) CREATE (u)-[:FOLLOW]->(u1)" %(followRelation.user.userID , followRelation.followedUser.userID);
Create a Like Relation	"MATCH (u:User {userID: '%s'}) MATCH (a: Accommodation{accommodationID : '%s'}) CREATE (u)-[:LIKE]->(a)"

	%(likeRelation.user.userID, likeRelation.accommodation.accommodationID);

2.2.4.2. Read

Operation	
Get Total Likes for Accommodation	query = "MATCH(a: Accommodation)<-[r:LIKE]-(u: User) WHERE a.accommodationID = '%s' return COUNT(r) as total" % accommodationID;
Get Total Likes for Activity	query = "MATCH(a: Activity)<-[r:LIKE]-(u:User) WHERE a.activityID = '%s' return COUNT(r) as total" % activityNodeID;
Get Followed Users	query = "MATCH(u:User {userID: '%s' })-[:FOLLOW]-> (followed: User) return followed" % userNodeID; queryResult = list(session.run(query));
Get Liked Accommodations	query = "MATCH(u:User {userID: '%s' })-[:LIKE]->(liked: Accommodation) ; return liked" % userNodeID;
Get Liked Activities	query = "MATCH(u:User {userID: '%s' })-[:LIKE]->(liked: Activity) return liked" % userNodeID;

2.2.4.4. Update

Operation	
Update the Accommodation's name	query = "MATCH (a:Accommodation {accommodationID: '%s'}) SET a.name='%s'" % (accommodationNode.accommodationID, accommodationNode.name);
Update the Activity's name	query = "MATCH (a:Activity {activityID: '%s'}) SET a.name='%s'" % (activityNode.activityID, activityNode.name);

2.2.4.4. Delete

Operation	
Delete an Accommodation	query = "MATCH (a:Accommodation {accommodationID: '%s'}) DELETE a" %accommodationNode.accommodationID;
Delete an Activity	query = "MATCH (a:Activity {activityID: '%s'}) DELETE a" %activityNodeID;
Delete an User	query = "MATCH (u:User {userID: '%s'}) DETACH DELETE u" % userNodeID;
Delete a Follow Relation	query = "MATCH (u: User {userID : '%s' })-[r:FOLLOW]-> (u1:User {userID: '%s'}) DELETE r" %(followRelation.user.userID , followRelation.followedUser.userID);

Delete a Like Relation to an Accommodation	query = "MATCH (u:User {userID: '%s'})-[r:LIKE]-> (a: Accommodation{accommodationID : '%s'}) DELETE r""%(likeRelation.user.userID, likeRelation.accommodation.accommodationID);
Delete a Like Relation to an Activity	query = "MATCH (u:User {userID: '%s'})-[r:LIKE]-> (a: Activity{activityID: '%s'}) DELETE r"%(likeRelation.user.userID, likeRelation.activity.activityID);

2.2.5. On-graph queries

2.2.5.1. Recommended Accommodations/Activities

For the recommendation of accommodations/activities our system uses a simple and clear recommendation system. In particular, for each registered user, the system shows in the home page the top 3 recommended accommodations and the top 3 recommended activities. The system computes these advertisements through this query:

```
if destinationType == "accommodation":
    if totalFollowed == 0:
        query = "MATCH (u:User)-[r:LIKE]->(a:Accommodation) " \
                "MATCH (u2:User {userID: '%s'})  " \
                "WHERE NOT (u2)-[:LIKE]->(a)  "\
                "AND a.approved=True" \
                "return a , " \
                "COUNT(r) as liked " \
                "ORDER BY liked DESC " \
                "LIMIT 3" % userNode.userID
    else:
        query = "MATCH (u:User {userID: '%s'})-[:FOLLOW]->(u2:User) " \
                "MATCH (u2)-[:LIKE]->(a:Accommodation) " \
                "MATCH (u3:User)-[r:LIKE]->(a)  " \
                "WHERE NOT (u)-[:LIKE]->(a)  " \
                "AND a.approved=True "\
                "return a , " \
                "COUNT(r) as liked " \
                "ORDER BY liked DESC " \
                "LIMIT 3" % userNode.userID
```

Recommended accommodations query

For simplicity is shown only the query related to accommodations, but in the activities case is almost the same thing.

With this query, the system shows the 3 accommodations/activities most liked by the users, followed by the logged user, that are not liked by himself yet.

In case the logged user does not follow anyone, the system shows the 3 accommodations/activities most liked by all the users, that are not liked by himself yet.

2.2.5.2. Recommended Users

For the recommendation of users our system uses a simple and clear recommendation system. In particular, for each registered user, the system shows in the homepage the top 3 recommended users. The system computes these users through this query:

```
if totalFollowed == 0:
    query = "MATCH (u : User) " \
            "MATCH (u2)-[r:FOLLOW]->(u)  " \
            "WHERE NOT u.userID = '%s' " \
            "return u ," \
            "COUNT(r) as followed " \
            "ORDER BY followed DESC " \
            "LIMIT 3" % userNode.userID
else:
    query = "MATCH (u:User {userID: '%s'})-[:FOLLOW]->(u2:User) " \
            "MATCH (u2)-[:FOLLOW]->(u3:User) " \
            "MATCH (u4)-[r:FOLLOW]->(u3)  " \
            "WHERE NOT (u)-[:FOLLOW]->(u3) " \
            "return u3 , " \
            "COUNT(r) as followed " \
            "ORDER BY followed DESC " \
            "LIMIT 3" % userNode.userID
```

Recommended users query

With this query, the system shows the 3 users most liked by the users, followed by the logged user, that are not followed by himself yet.
In case the logged user does not follow no one, the system shows 3 random users that are not followed by himself yet.

2.2.5.3. Common liked Accommodations/Activities

With this on-graph query, the system shows in every user profile the liked accommodations/activities in common between the logged user and the viewed profile. The system computes these advertisements through this query:

```
query = "MATCH(u1: User {userID: '%s' })-[:LIKE]->(a: Accommodation)<-[:LIKE]-(u2: User {userID: '%s' }) " \
        "WHERE a.approved=True " \
        "return a" % (firstUserNode.userID, secondUserID)
```

Common liked accommodations query

For simplicity is shown only the query related to accommodations, but in the activities case is almost the same thing.

2.2.6. Neo4J Indexes

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

Index Name	Index Label	Index Field
ApprovedAccomodation Index	:Accommodation	approved
ApprovedActivity Index	:Activity	approved

2.3 Inter-DBMS Consistency

Using two different DBMSs, it is critical to maintain consistency between the two databases. Therefore, special controls and procedures have been implemented for all operations that modify, create, or delete entities that are maintained in both DBMSs. In particular, the entities in common are user, accommodation and activity. In particular, three levels of management have been implemented to try to cope with failures.

2.3.1. Consistency Layers

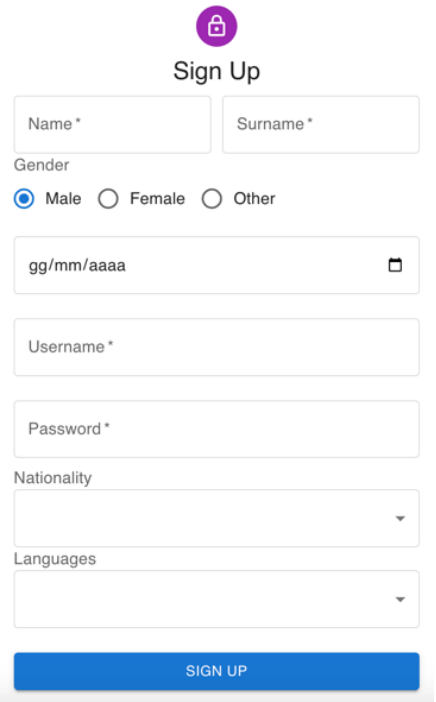
The first level of management is concerned with maintaining consistency at the time when one of the operations requiring updating of both DBMSs is performed. Each operation is first performed on the DocumentDB, and then, once the operation is successfully completed, it is replicated to the graph DB.
Should the operation fail, a JSON containing information about the failed operation is saved in a log file related to the current day. At midnight each day, a worker then takes care of reading the log file and going to re-execute the operation that failed during the day. Should the worker also fail, it writes a warning containing information about the failed operation to the log file. It will then be up to the admin to read the log file and possibly manually correct the errors to maintain consistency.

3. User Manual

This paragraph contains information regarding the user capabilities.

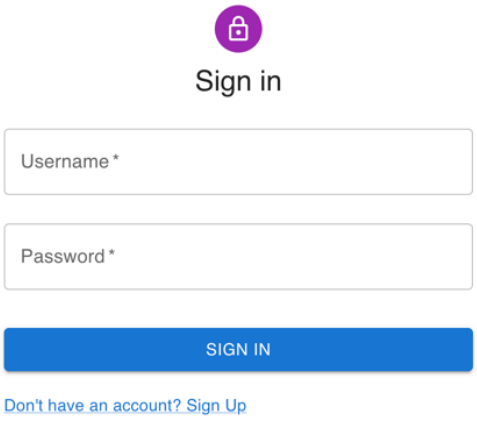
3.1. Sign up

The user can register to the site by filling the form with the required information.



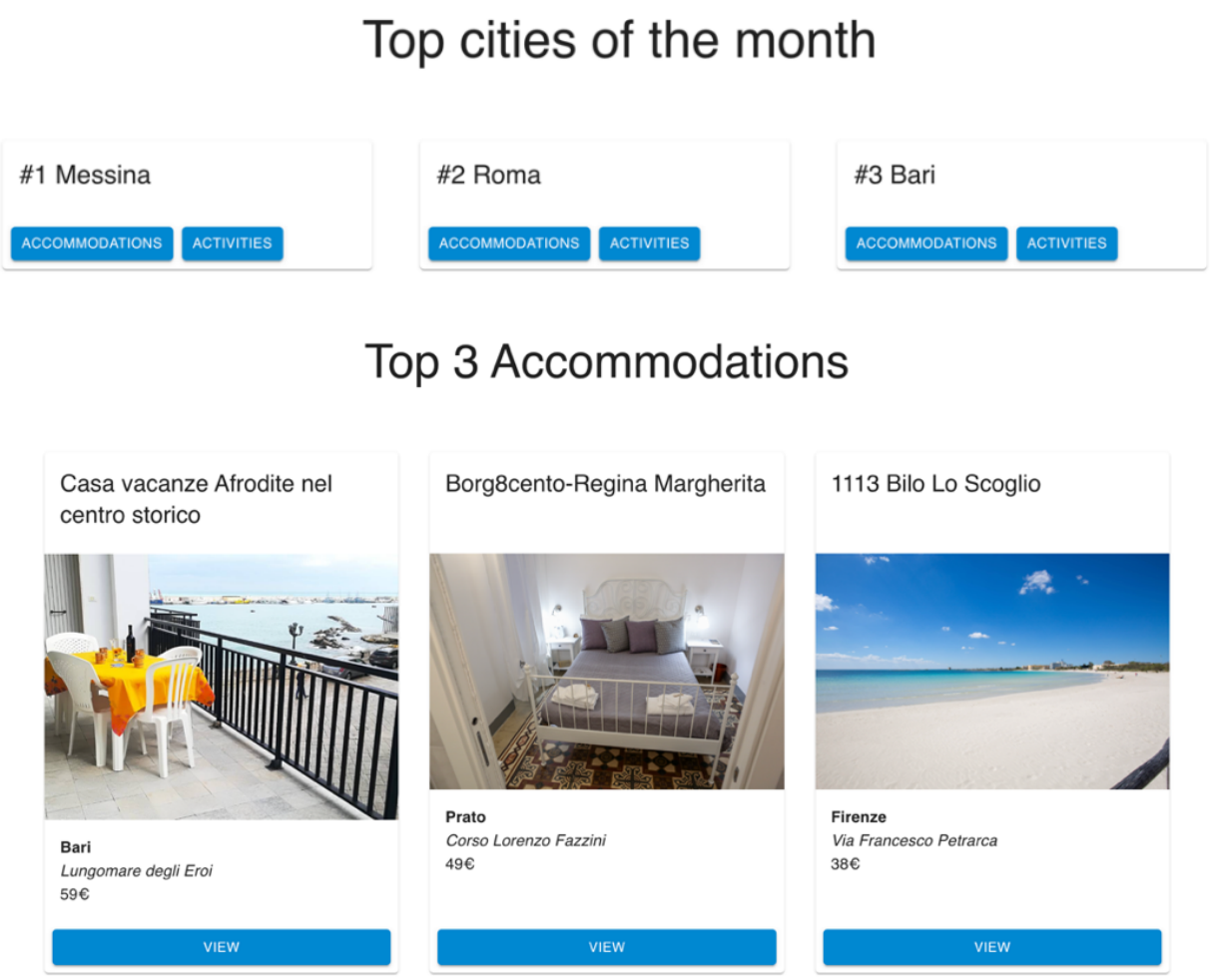
3.2. Sign in

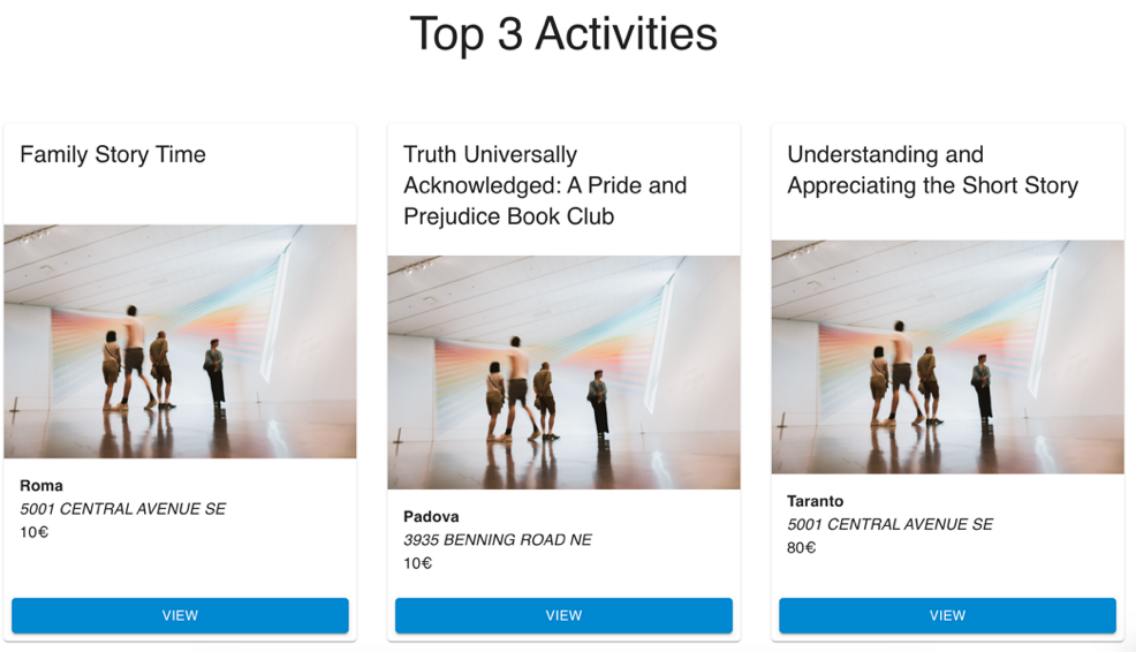
The user can log in to the site filling the form with the credential previously provided during the registration phase (username/password).



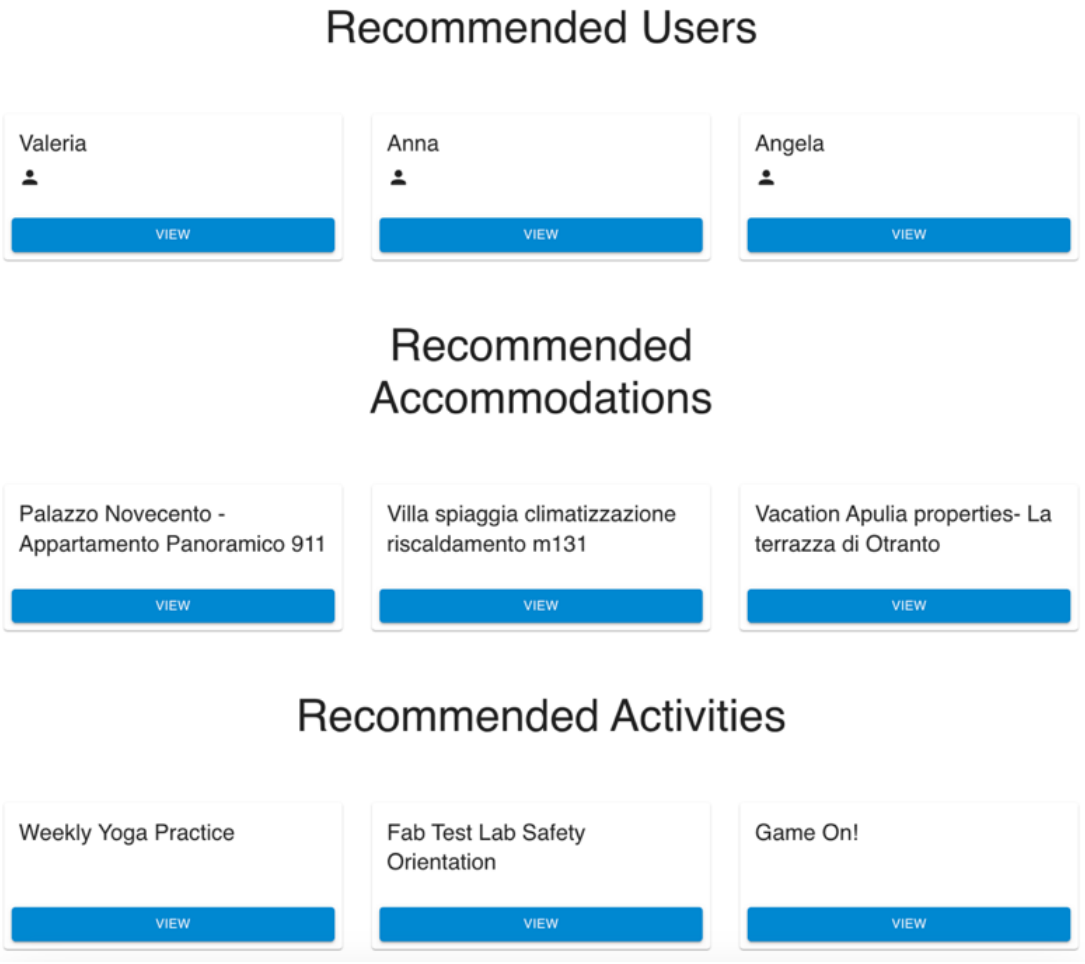
3.3. Home

In the home page we have different scenarios based on if the user is authenticated or not. In case of a non-authenticated user, the home shows the three most famous cities of the month based on the reservations they have, then by clicking on the appropriate button the user can check accommodations and activities based on the chosen city. The non-authenticated user can also check the top three accommodations and activities of all times based on the reservations number.





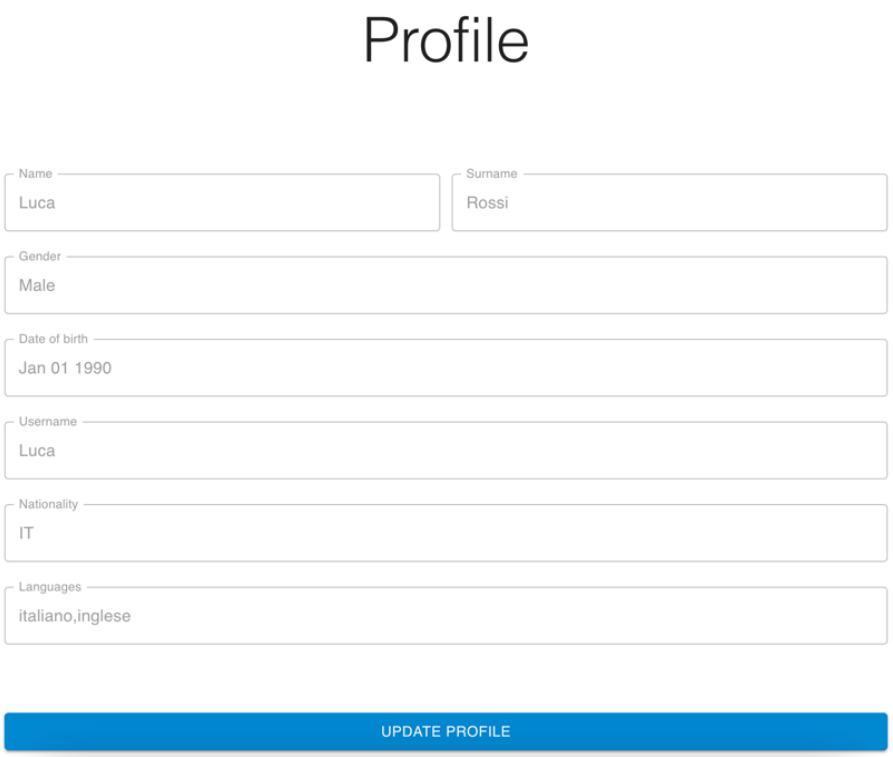
If the user is authenticated, beyond the already described functionalities, can check the recommended users based on the following ones and check the recommended accommodations and activities based on the most liked ones of the following ones.

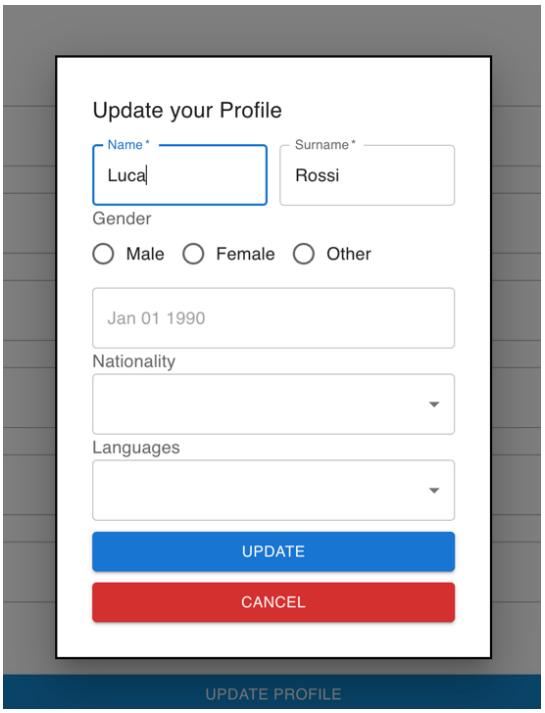


3.4. Profile

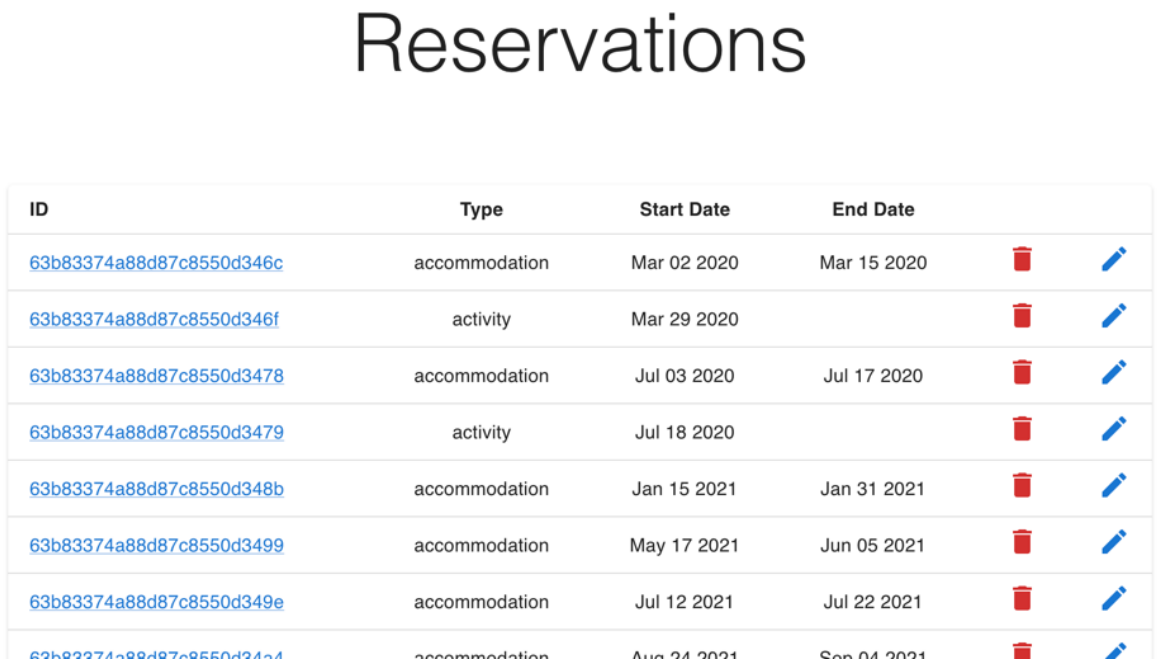
The authenticated user can:

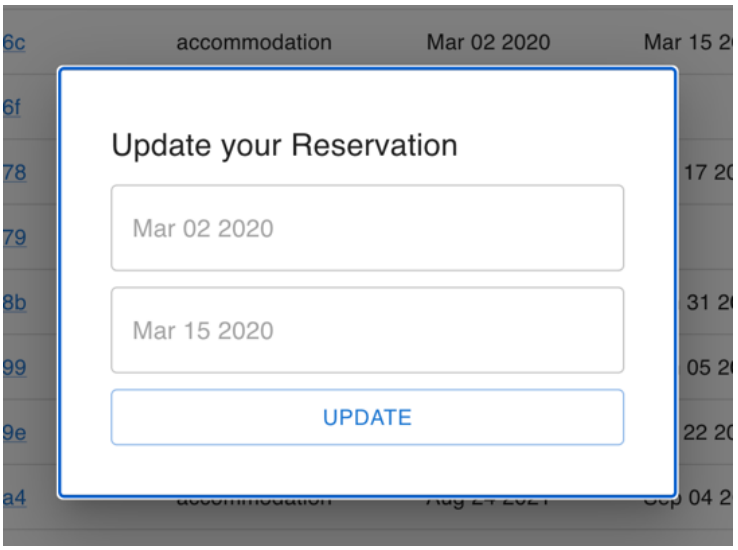
- Check and update his personal profile.



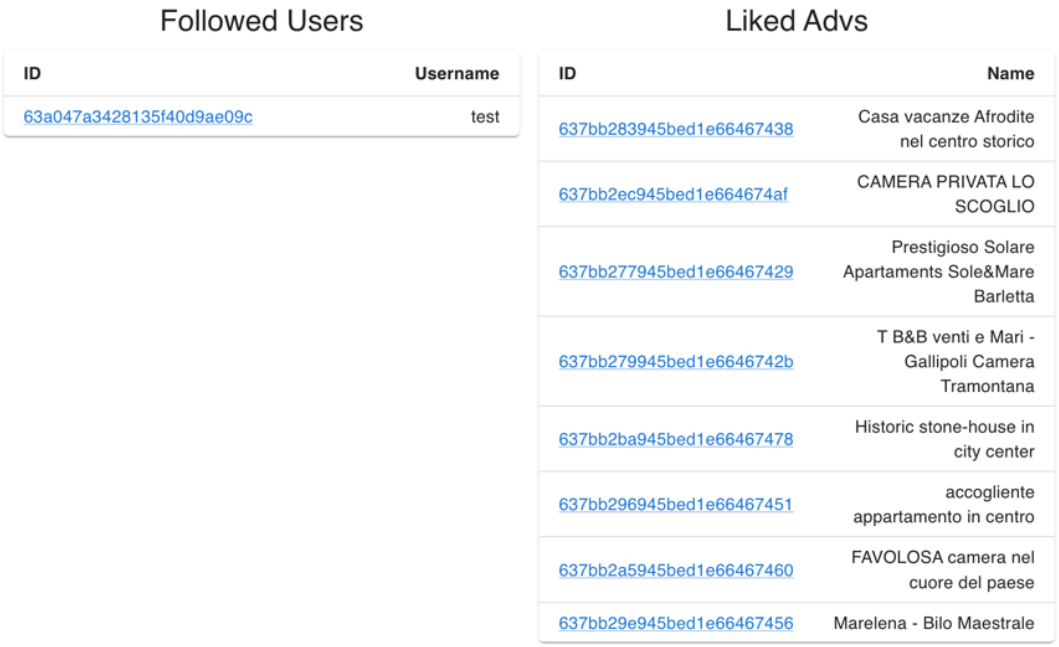


- Check/delete/edit reservations of the booked activities or accommodations.

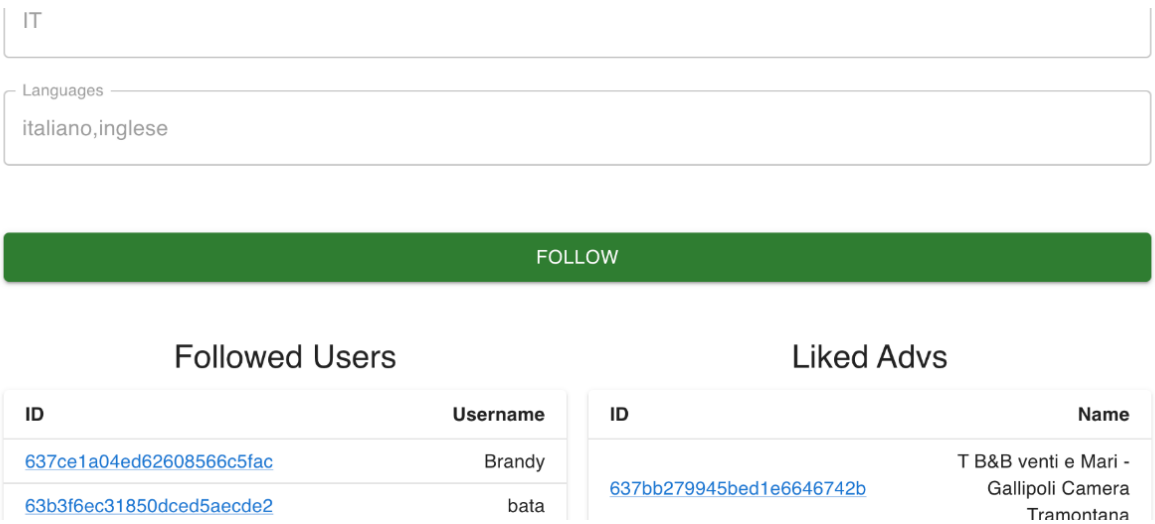




- Check the followed users and the liked accommodations/activities.



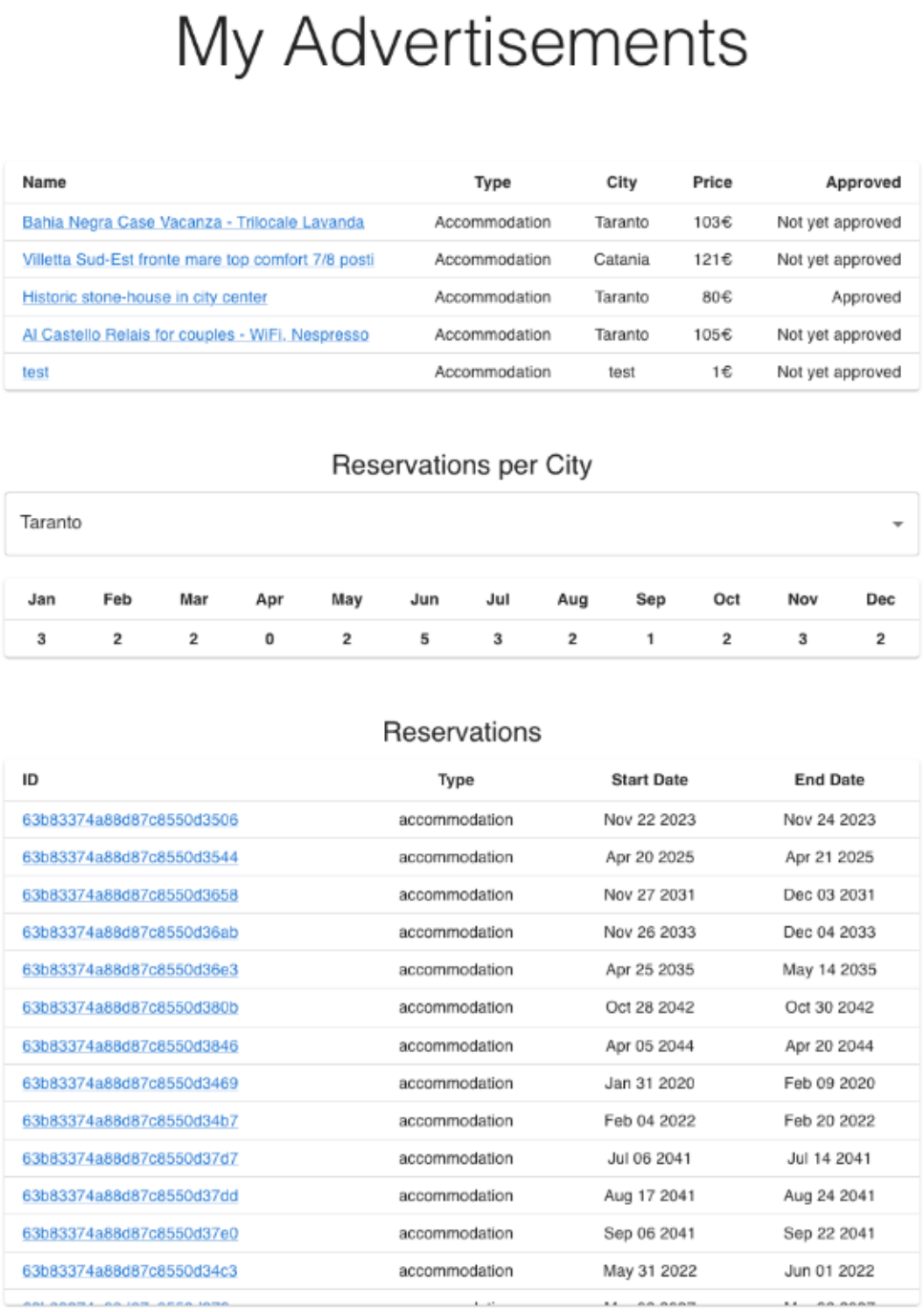
- In the profile page of another user the authenticated one can check the profile information and follow or unfollow the specific user



3.5. My Adv

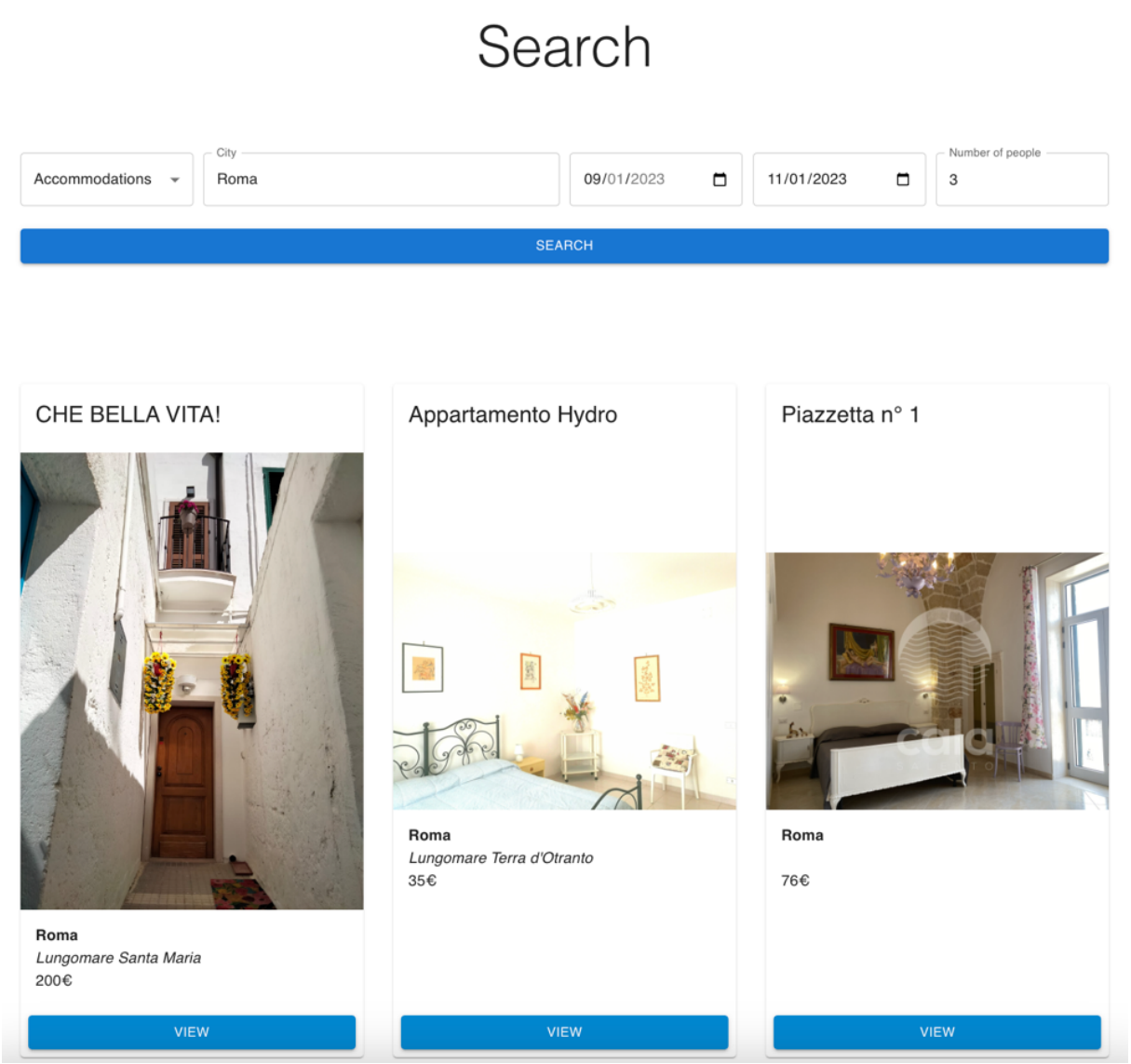
The authenticated user can:

- Check their own accommodations and activities and their statistics based on the monthly reservations of the selected city.
- Check the reservations of their accommodations or activities



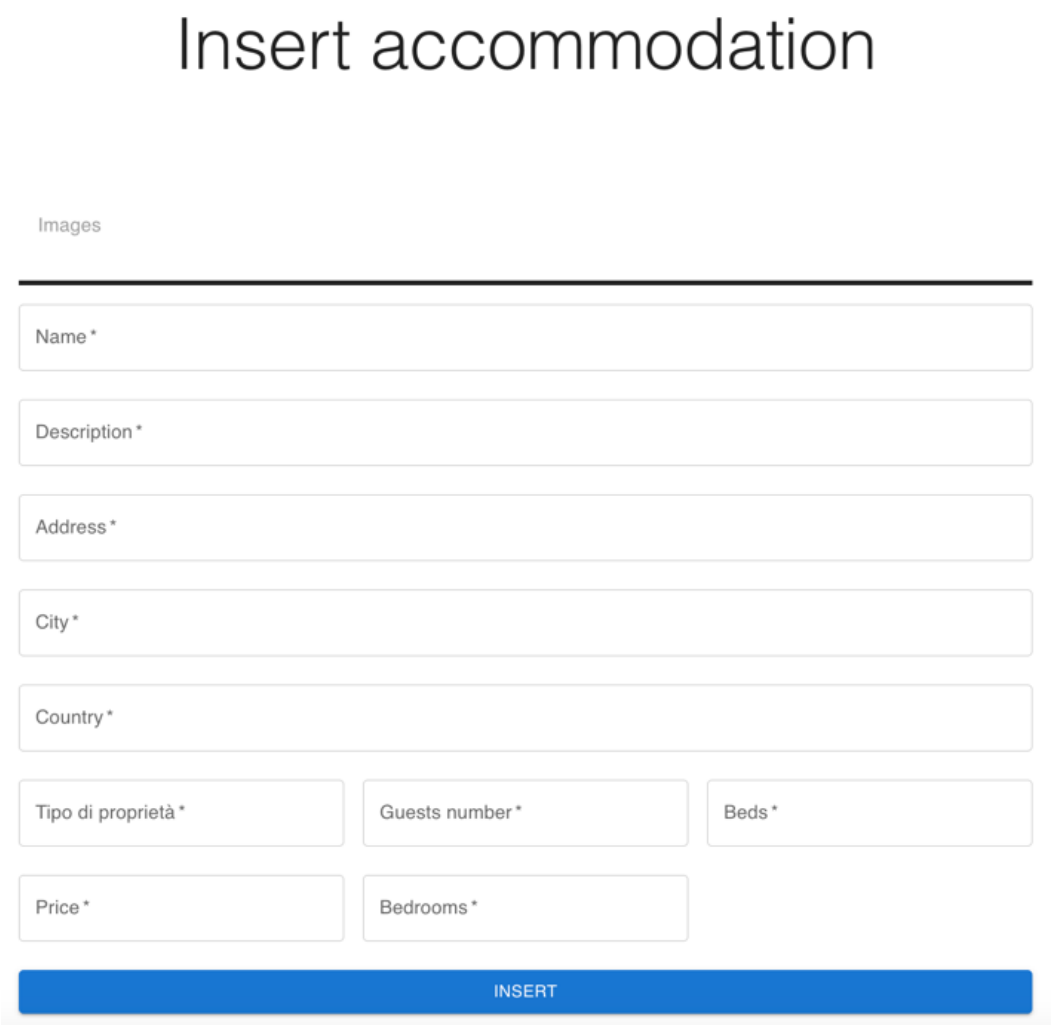
3.6. Search

Both the authenticated and non-authenticated user can browse accommodations/activities available on the site, based on the filters inserted previously in the search form.



3.9. Insert Accommodation

The authenticated user can insert a new accommodation by filling the form; then the admin may approve or disapprove it.

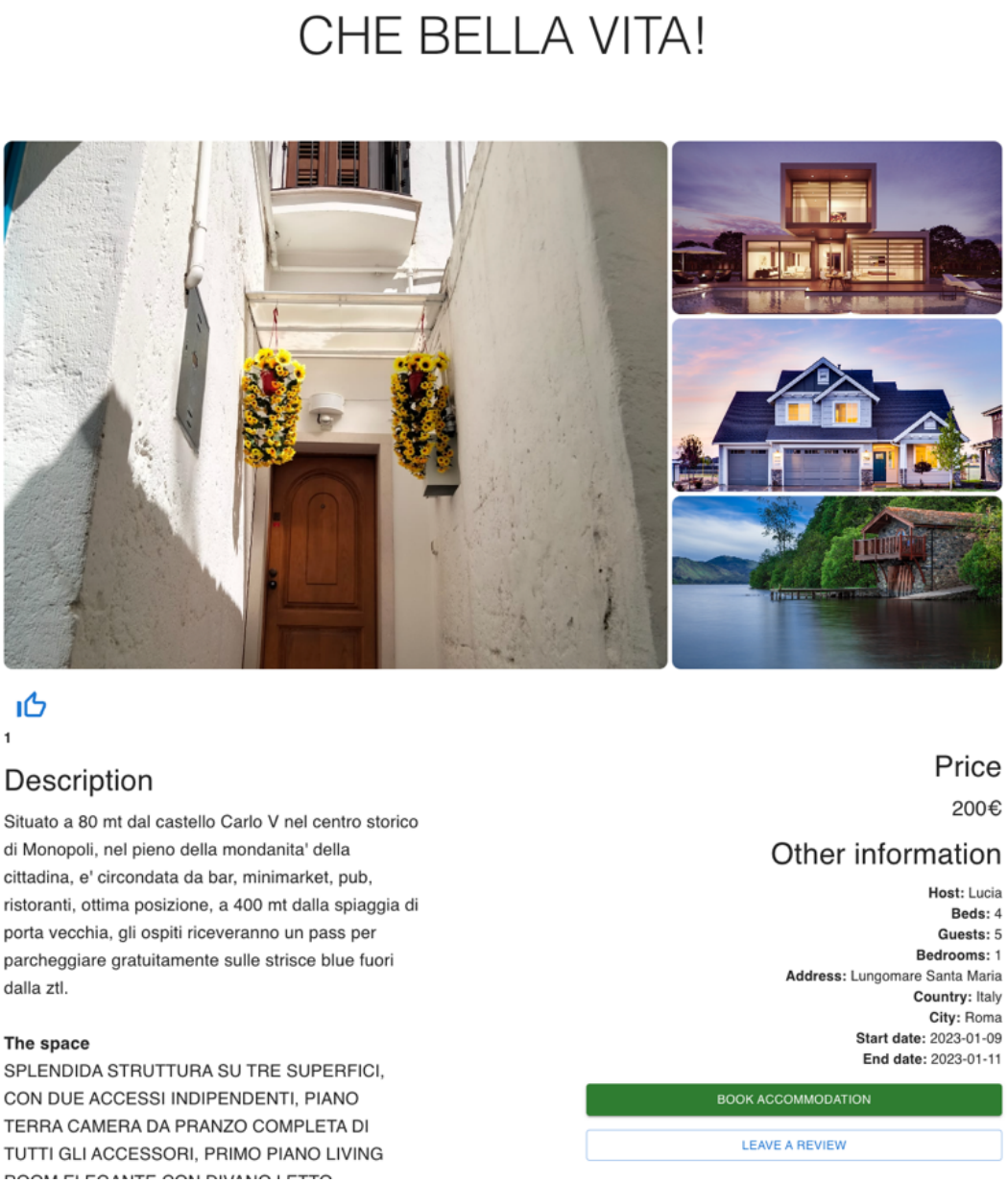


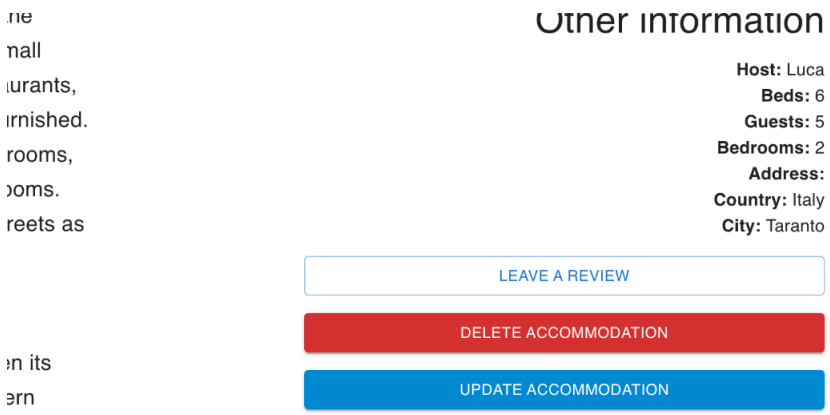
3.8. Insert Activity

The authenticated user can insert a new activity by filling the form; then the admin may approve or disapprove it (very similar to the accommodation one).

3.9. Accommodation Page

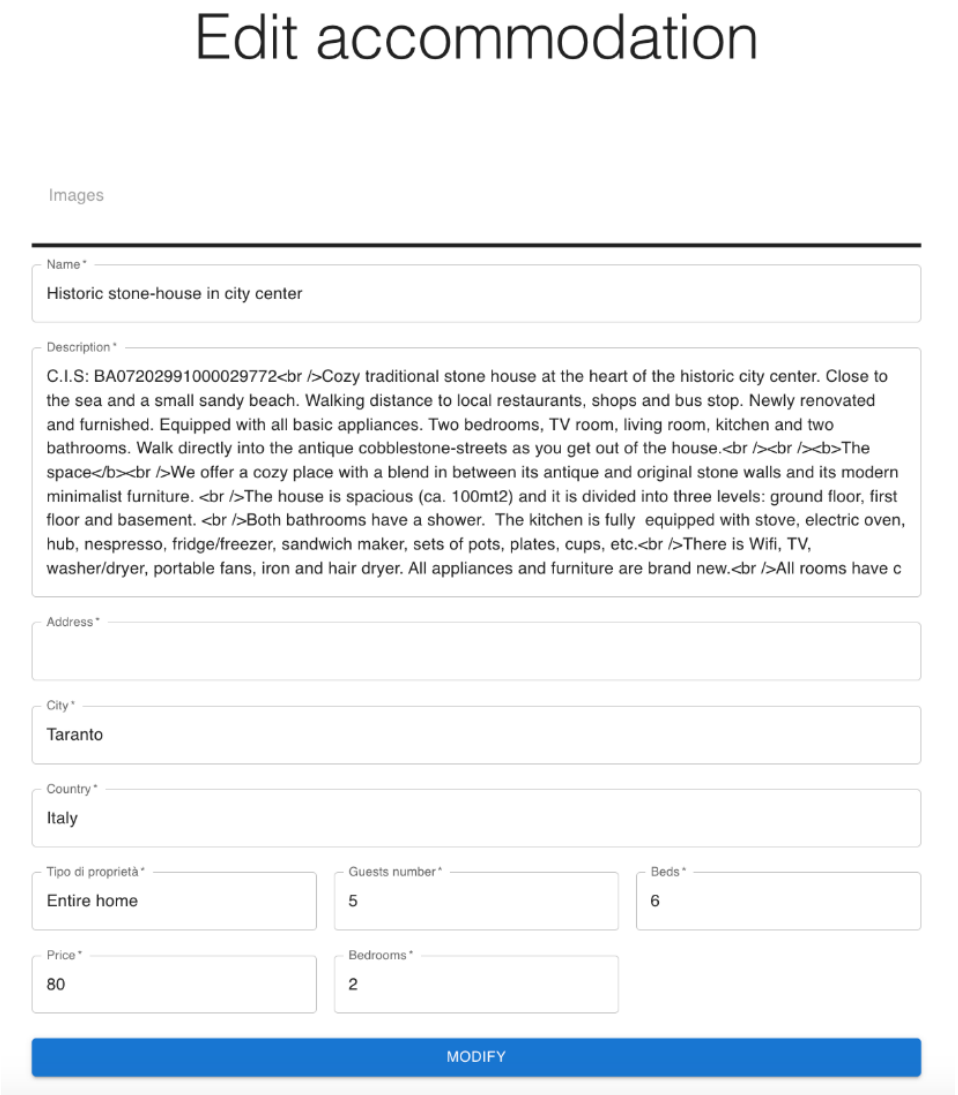
In the accommodation page we have different scenarios based on if the user is authenticated, non-authenticated or if it's the owner of the accommodation. The user can, if non-authenticated, check the information of the accommodation otherwise the authenticated one can also book the accommodation based on the information provided during the search phase, can leave a review with a score and a comment if not already done and if the accommodation is booked and leave a like to the accommodation. In the case the user is the owner he can edit or delete the accommodation.





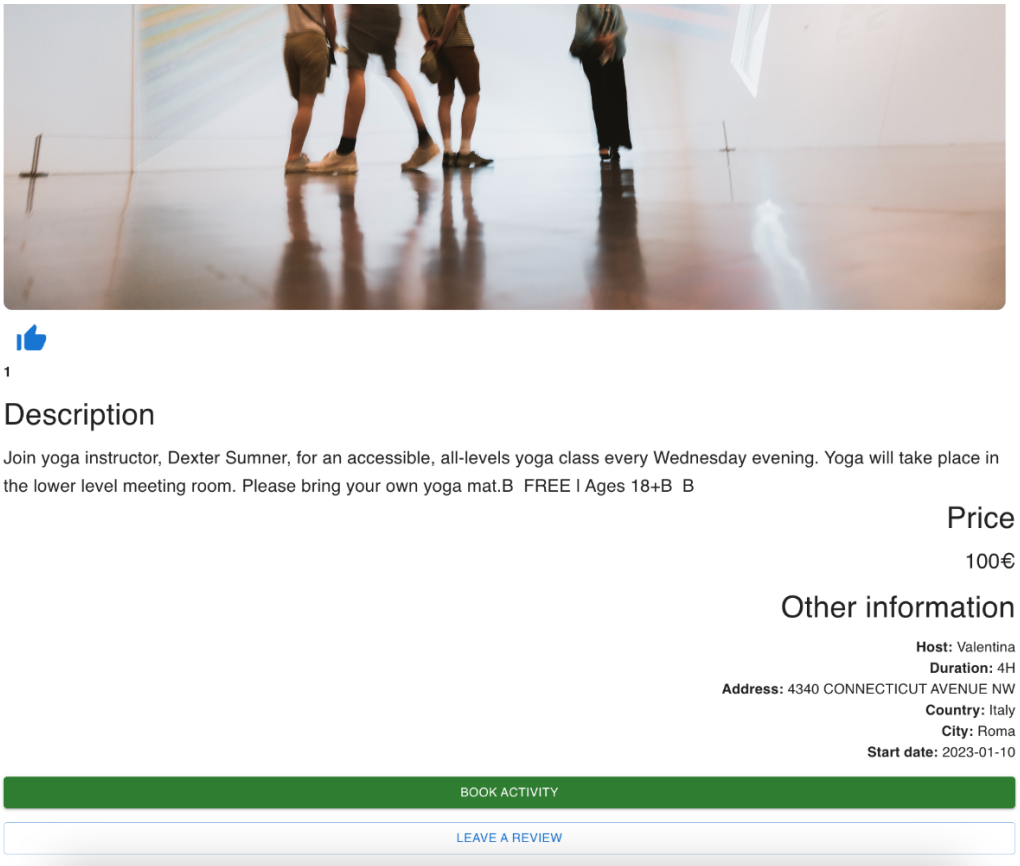
3.10. Edit Accommodation Page

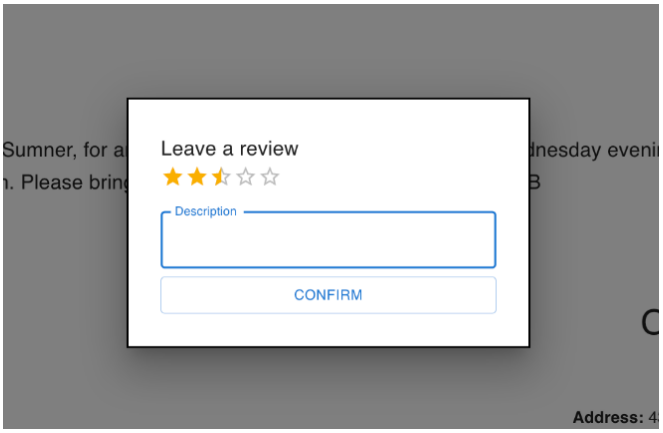
The user can update the previously inserted information of the accommodation by filling the form.



3.11. Activity Page

In the activity page we have different scenarios based on if the user is authenticated, non-authenticated or if it's the owner of the activity. The user can, if non-authenticated, check the information of the activity otherwise the authenticated one can also book the activity based on the information provided during the search phase, can leave a review with a score and a comment if not already done and if the activity is booked and leave a like to the activity. In the case the user is the owner he can edit or delete the activity.



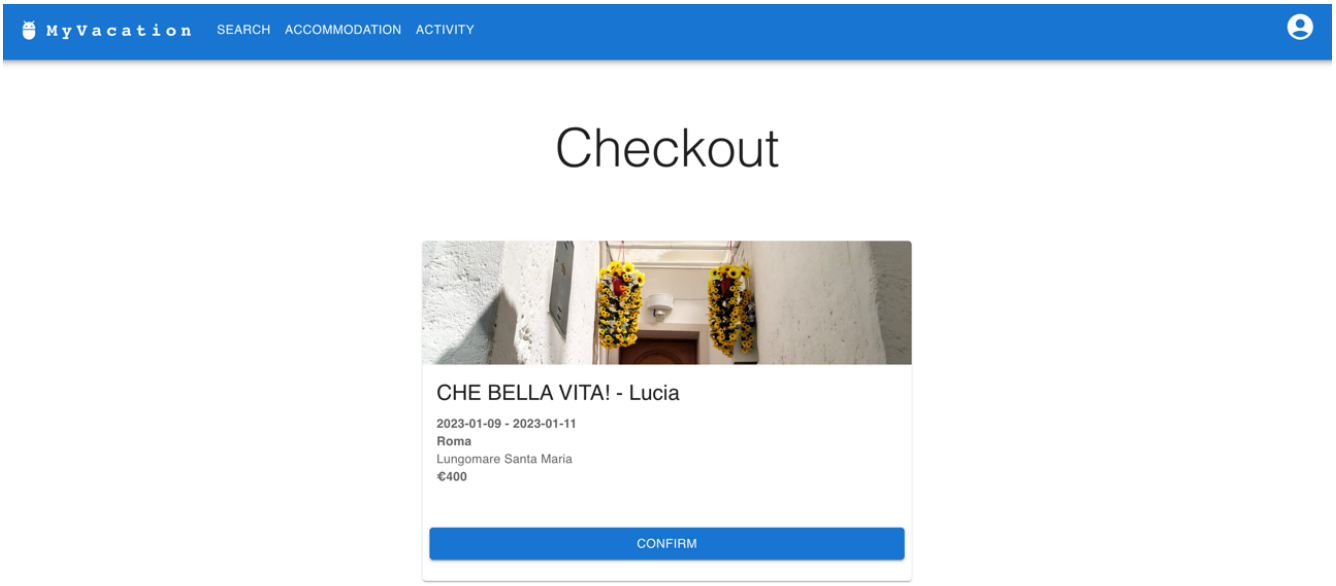


3.12. Edit Activity Page

The user can update the previously inserted information of the activity by filling the form (very similar to the accommodation one).

3.13. Confirm Booking

The user can check the booked activity/accommodation and confirm the reservation.



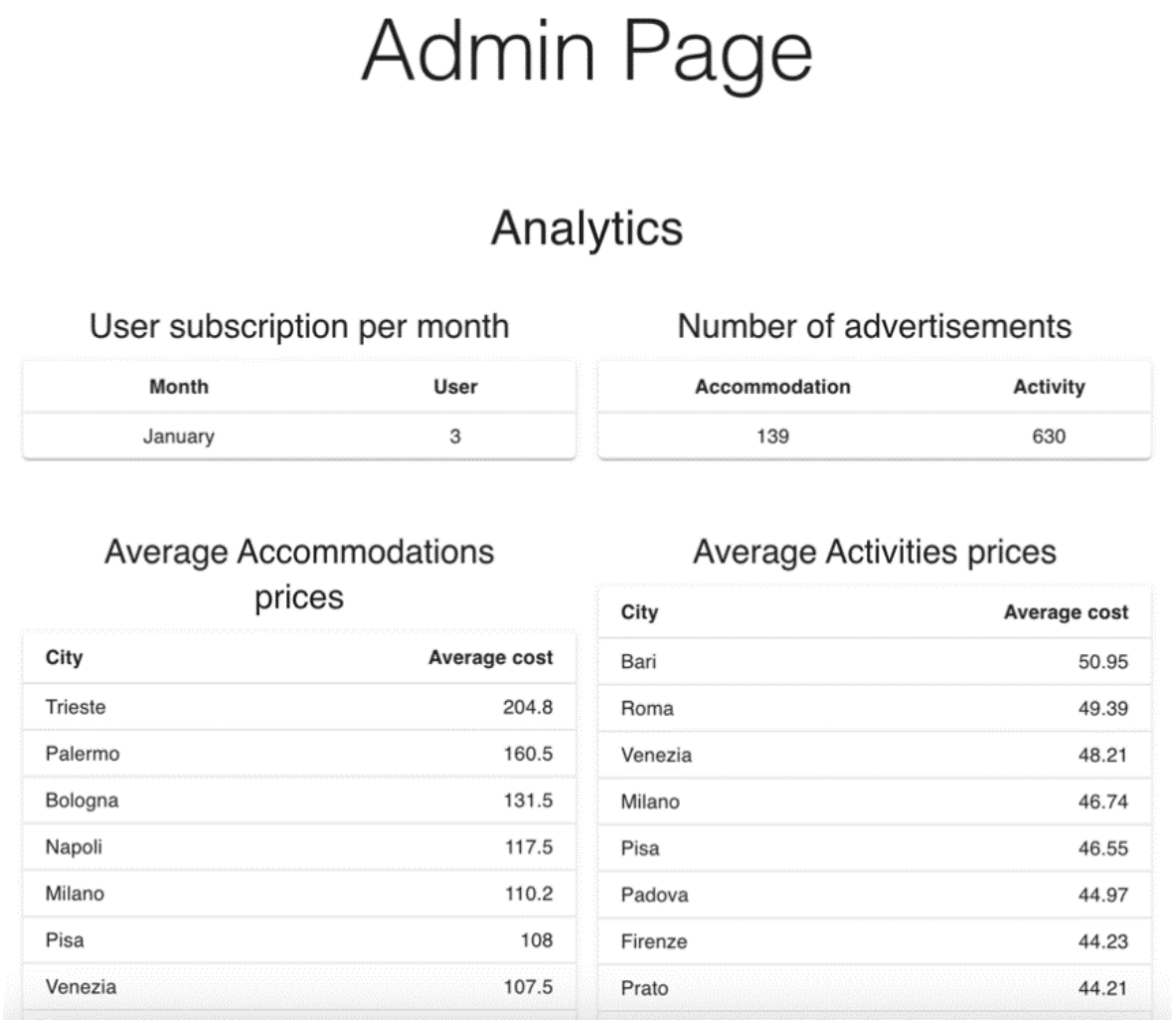
4. Admin Manual

This paragraph contains information regarding the admin capabilities.

4.1. Admin Page

The admin user can:

- Check the analytics of the site based on user subscription per month, number of Accommodations and Activities, average Accommodations prices and average Activities prices.



Admin Page

Analytics

User subscription per month

Month	User
January	3

Number of advertisements

Accommodation	Activity
139	630

Average Accommodations prices

City	Average cost
Trieste	204.8
Palermo	160.5
Bologna	131.5
Napoli	117.5
Milano	110.2
Pisa	108
Venezia	107.5

Average Activities prices

City	Average cost
Bari	50.95
Roma	49.39
Venezia	48.21
Milano	46.74
Pisa	46.55
Padova	44.97
Firenze	44.23
Prato	44.21

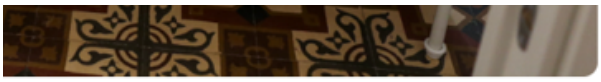
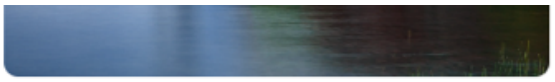
- Approve or disapprove an accommodation or an activity by clicking on the id of the specific one, that will redirect the admin to the page with all the information regarding the selected one, and then by clicking on the specific button approve/disapprove.

Accommodations to be approved

HostID	Title	City	Price
637ce1a04ed62608566c5fa9	Borg8cento-Regina Margherita	Prato	49
637ce1a04ed62608566c5fac	mini apt/ monolocale	Barbarano	73
637ce1a04ed62608566c5fa7	h	h	4
63b7fcfdd4f5a1c5cbb417ea	test	test	1

Activities to be approved

HostID	Title	City	Price

Price

49€

lall'aspetto
menti e le
a del borgo

Other information

Host: Anna

Beds: 1
Minimum nights:
Guests: 2
Bedrooms: 1

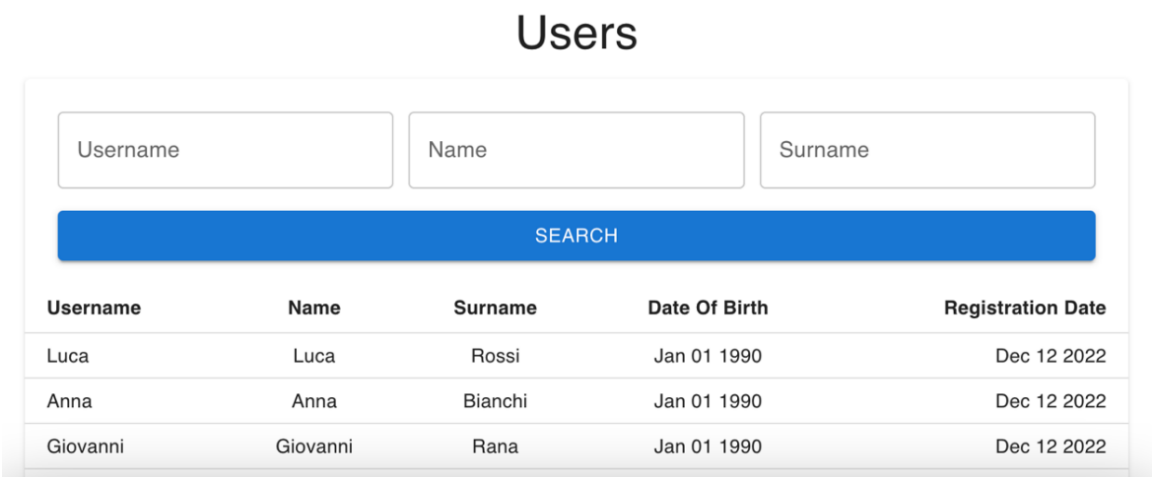
Country: Italy
City: Prato
Address: Corso Lorenzo Fazzini

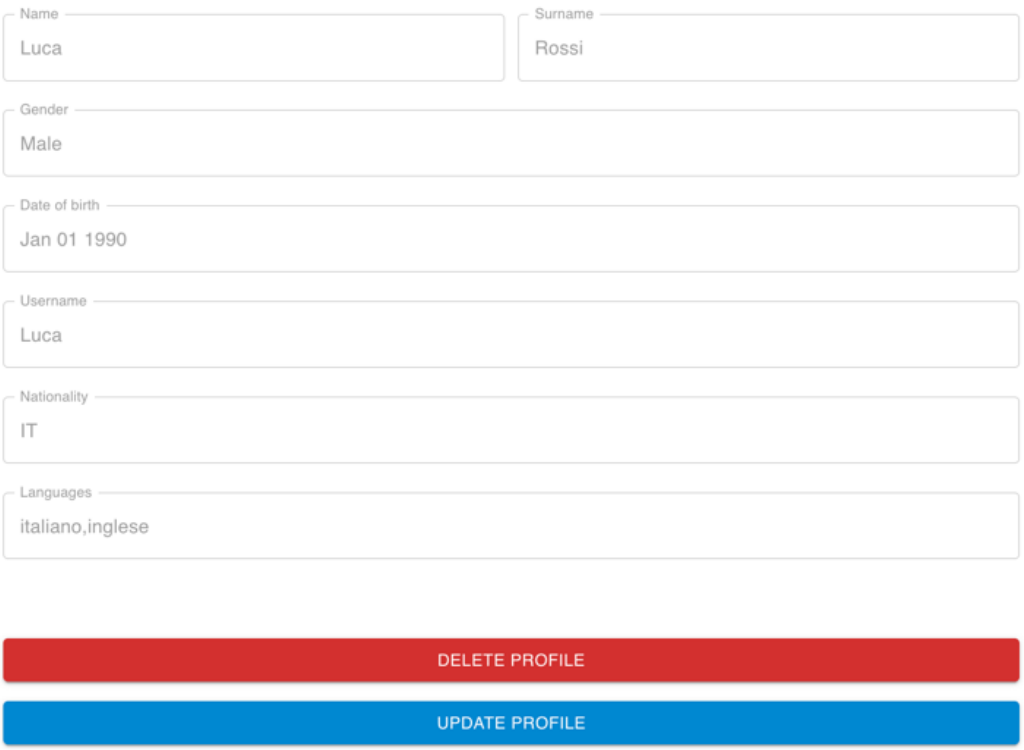
APPROVE

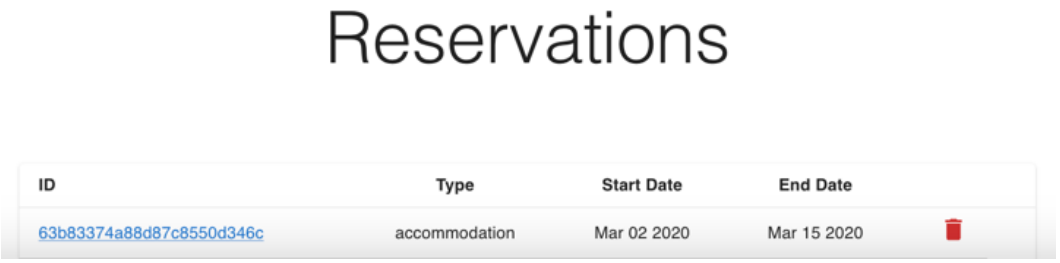
NOT APPROVE

afarmacia,

- Search a user from the specific table and then by clicking on one of them the admin can check/update the information of the specific user and can also delete it.







4.2. Search Page

The admin user can perform a search as previously described in the user manual.

4.3. Accommodation Page

The admin can Edit or Delete an Accommodation as previously described in the user manual.

4.4. Activity page

The admin can Edit or Delete an Activity as previously described in the user manual.