MyVacation

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 nr of guests.
- Unregistered User can view an accommodation/activity.
- Unregistered User can browse top 3 cities.
- Unregistered User can browse top 3 accommodations.
- Unregistered User can browse top 3 activities.
- Uregistered 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 nr 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 top 3 cities.
- Registered User can browse top 3 accommodations.
- Registered User can browse top 3 activities.
- Registered User can browse a recommended accommodation/activity.
- Registered User can browse a recommended user.
- Registered User can reserve an accommodation/actvity.
- Registered User can view 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 an user profile.
- Registered User can follow/unfollow an user.
- Registered User can like/unlike an accommodation/activity.
- Registered User can view the own user profile.
- Registered User can update the own user profile.

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

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 must be written in python.
- 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
- Every accommodation/activity must have at least one picture.
- An accommodation/activity cannot have different reservations for the same period.
- 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.

Actors

The users of the application will be divided into three categories: Unregistered Users, Registered Users and Admin.

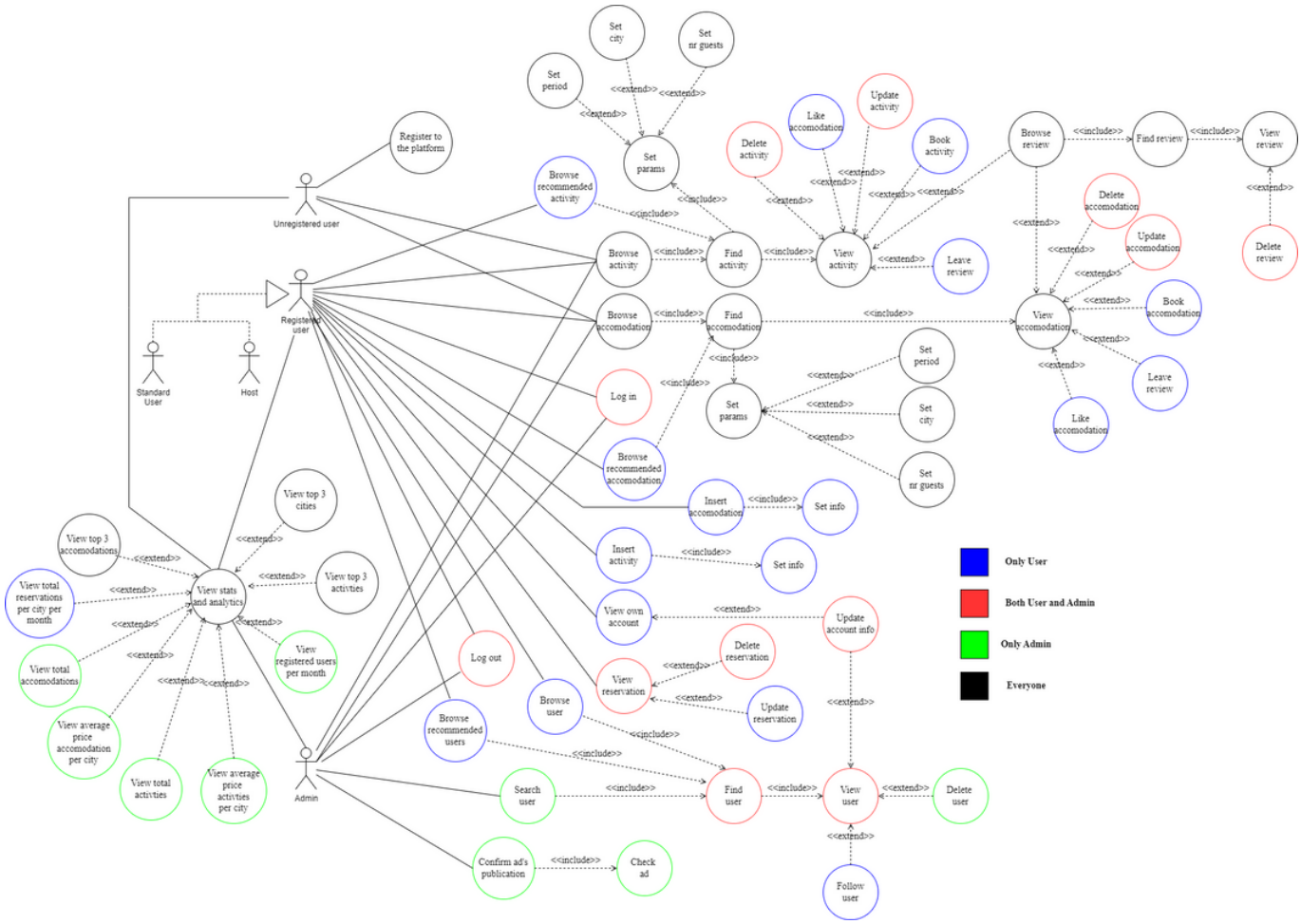
CAP Theorem

AP + eventual consistency

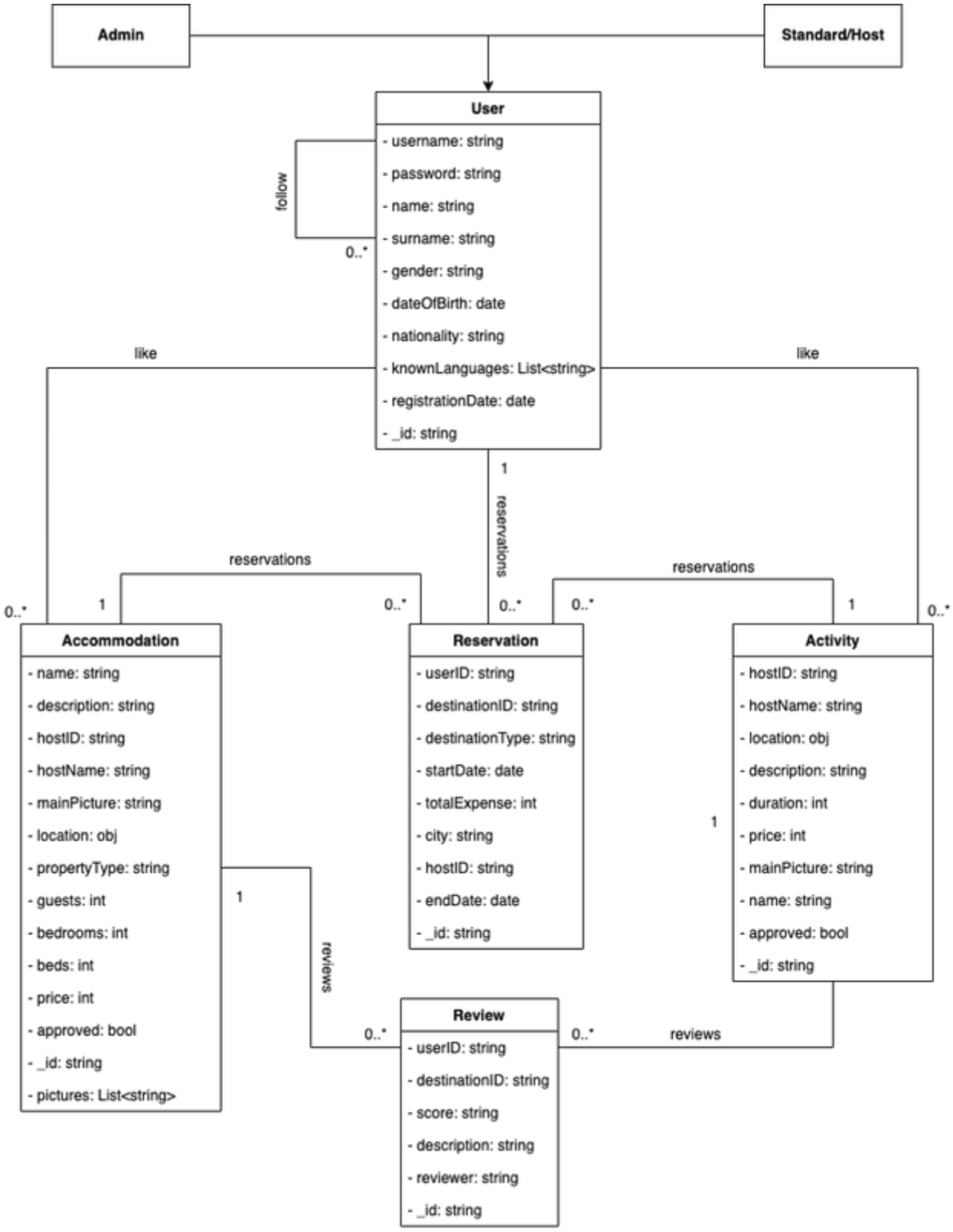
Reservation

Nella collection User mantenere l'array reservations solo per le prenotazioni future. Con ridondanza con la collection reservations. Stessa gestione con le reviews.

Use Cases


Set city
Set nr guests
Set period
<<extend>>
Set params
Register to the platform
Unregistered user
Browse recommended activity
<<include>>
Delete activity
Like accomodation
Update activity
Book activity
Browse review
<<include>>
Find review
<<include>>
View review
<<extend>>
Delete review
Delete accomodation
Update accomodation
Browse activity
Find activity
View activity
Leave review
Registered user
Standard User
Host
Browse accomodation
Find accomodation
View accomodation
Book accomodation
Leave review
Like accomodation
Log in
Set period
Set city
Set nr guests
Browse recommended accomodation
Insert accomodation
Set info
View top 3 cities
View top 3 accomodations
View stats and analytics
View top 3 activities
View total reservations per city per month
View total accomodations
View average price accomodation per city
View registered users per month
View total activities
View average price activities per city
Admin
Insert activity
Set info
View own account
Update account info
Delete reservation
View reservation
Update reservation
Log out
Browse user
Browse recommended users
Search user
Find user
View user
Delete user
Follow user
Confirm ad's publication
Check ad
Only User
Both User and Admin
Only Admin
Everyone

UML



Dataset Source

https://www.kaggle.com/datasets/alessiocrisafulli/airbnb-italy

Roba a caso utile

- https://github.com/VictorOmondi1997/airbnb-dataset-cleaning/blob/master/Cleaning_Airbnb_Data_in_Python.ipynb

appunti claudio

As presented, into the accommodations and activities collections we decided to embed the reviews. 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 cause 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 10/15 most recent reviews in both cases to improve the performances of the application, while offering as many features as possible to the user. In this way we introduced redundancies and denormalized data, but at the same time we improved the performances because in most cases we don't have to do join operations to see ads reviews. In fact, generally, a user reads only few of the most recent reviews and we think that 10/15 reviews are, generally, enough. Furthermore we were able to improve read operations, that are the most frequent in an application like ours, with the use of indexes.

//DA FARE IL DISCORSO RELATIVO ALLE RESERVATIONS

AP Solution (Availability and Partition Tolerance): The application should be accessible to the users at any given point in time. 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 comparing to the benefit of availability in the case of this application

We decided to prefer availability over consistency in a accommodation/activity booking system to provide a better experience to the customers. To gain more availability, we might allow both the nodes to keep accepting accommodation/activity reservations even if the communication line breaks. The worst possible outcome of this approach is that 2 customers will end up making the same accommodation/activity reservation. However, such situations can be resolved using domain knowledge. It's a pretty common occurrence that the room/activity are overbooked and then the company address such cases by taking the appropriate measures (e.g., Refunds, moving to another accommodation/activity, etc.). This is directly linked to the adoption of the Eventual Consistency paradigm for our distributed system which is better detailed further.

INDEXES MONGODB INDEXES FOR USERS Oltre ad _id abbiamo: Indexes on "users" We decided to force the uniqueness of the username chosen by the members of our community to avoid the unpleasant "who-is-who" problem. To obtain this we defined a unique index on the Username field of the user documents. We did this in our application code performing an action equivalent to de following one in mongosh syntax: db.users.createIndex({ "Username": 1 },{ unique: true, }) INDEXES FOR RESERVATIONS: _id, start_date, end_date, destinationType, destinationID INDEXES FOR ACCOMODATION e ACTIVITY: _id, city

We defined and implemented some analytics in our mongo implementation to analyse the user interaction with our system and gain insight on their behaviour to verify the hypothesis that led to implementation decisions match the real system usage. We admit there are simple, but they are useful and grant valuable information about our system and this is the only point that matters. Community Growth How does out community evolve? This is probably one of the most interesting information a website owner could ask for, both for planning future system upgrades and – why not – to proudly display it to new visitors and partners.

To obtain this information we perform an aggregation on the collection “users” to count, for each day in which a new user became a member in our community, how many users joined us in that day. Having the possibility to plot diagrams, the result could be displayed in a time graph to easily recognise patterns in the community evolution. Not handling diagrams in our front-end, we display the results of this analytics in a table in the Statistics section of our website, showing for each day for which data are available the absolute and relative growth in that day.

Neo4j [5] is a graph database management system, an extremely interesting technology that allows extremely fast queries on graph-like data. We chose it to maintain a subset, without replicating the posts content to limit waste of space, of the data stored in our mongo replicas to permit fast relation-based searches. Data are organised in nodes and relationships just as in many graph systems and properties (values associated to objects) can be created on demand. We did not plan to store massive objects in this database but many small entities, so we do not expect many relocations or shrinks of stored items to be required and, consequently of our choice of not storing voluminous texts here, we do not expect the limitations of the community version (e. g.: no replication support) to be a problem for us. 4.3.1 Entities In Neo4j we stored just the necessary information to perform the queries we wanted to execute. All the entities hold always enough information to find the corresponding entity and the missing values in the other db. The complete list of entities we used with neo4j is the following: • Accommodation: {AccommodationID, name: String} • Activity: {activityID: string, ecc, name: String} • User: {userID: string, username: string ecc} 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

It should be clear that they correspond almost to the same elements in the mongo database, but they keep only a reduced set of the original properties. In this database, we are going to store informations that concern following and liking, infos that we can't find on our document DB [METTO UN MOCKUP DI TUTTE LE RELAZIONI]

[INSERIAMO GRAPHDB DI ESEMPIO]

RECOMMENDATION QUERY: public static List recommendRestaurant(User user, Cuisine cuisine, City city, int distance, LikesFrom depth, Price price, int page, int perPage) { Map<String, Object> parameters = new HashMap<String,Object>(); parameters.put("city", city.getName()); parameters.put("distance", distance * 1000); parameters.put("depth", depth.ordinal()+1); parameters.put("price", price.getLowerValues()); parameters.put("skip", page*perPage); parameters.put("limit", perPage); parameters.put("username", user.getUsername());

```
    String cuisineFilter = "";
    String depthFilter =  depth == LikesFrom.FRIENDS_OF_FRIENDS ? "*1..2"
: "";
    if(cuisine != null) {
        cuisineFilter = "-[:SERVES]->(:Cuisine{name:$cuisine})";
        parameters.put("cuisine", cuisine.getName());
    }

    String query = "MATCH (ctarget:City{name:$city}), (c:City)<-
[:LOCATED]-(r:Restaurant)" + cuisineFilter + ", " +
           "(me:User{username:$username})-[:FOLLOWS" + depthFilter + "]->
```

```
(f:User)-[:LIKES]->(r) " +
            "WHERE f.username <> $username AND r.price IN $price " +
            "WITH ctarget, c, r, f, point({longitude: ctarget.longitude,
latitude:ctarget.latitude}) as p1, " +
            "   point({longitude: c.longitude, latitude:c.latitude}) as p2
" +
            "WITH distance(p1,p2) as dist, r, f " +
            "WHERE dist <= $distance AND NOT EXISTS ((r)-[:LIKES|OWNS]-
(:User{username:$username})) " +
            "RETURN r, (r)--(), count(DISTINCT f) as likes " +
            "ORDER BY likes DESC, r.name " +
            "SKIP $skip " +
            "LIMIT $limit ";

    Iterable<Restaurant> recommended =
DBManager.session().query(Restaurant.class, query ,parameters);
    List<Restaurant> restaurants = new ArrayList<Restaurant>();
    recommended.forEach(restaurants::add);
    return restaurants;
}
```
