UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Master of Science in
Artificial Intelligence and Data Engineering

ErasmusNest Project

Large Scale and Multi-Structured Databases

Giulio Capecchi
Jacopo Niccolai
Andrea Ruggieri

A.A. 2023/2024

Summary

1. Introduction

1.1. Abstract

ErasmusNest is Java-based application for easy and rapid **apartment's reservations**, thought for both helping exchange **students** looking for their lovely "*nest*" and hosts for house rent. Since most **hosts** always try to find possible customers using multiple social networks, the project's aim was providing an all-in-one simple solution where nothing but apartments and users are main characters, avoiding the latter to be lost in too much useless information. A fully personalized experience is given by **like and follows functionalities**; the first providing a rapid way to retrieve already visited favorite apartments and the latter for **ad-hoc suggestions**, both leveraging a continuously growing community.
Main actors are *User* and *Administrator*: since the first can exploit all application functionalities, such as reservations and networking with other users, Administrator's aim is to supervise application flow checking for the house upload phase and to access the **analytics** page **for advanced statistics**. Some ErasmusNest functionalities can be also exploited by a non-logged user, which can look for other users or apartments without making any relationship with them, until a sign-in preceded by a sign-up.

1.2. Frameworks and technologies

- IDE:
  - Pycharm for Python code
  - **IntelliJ** for Java, JavaFX, HTML and Javascript code

- Programming Languages:
  - Python
    - Script for overall dataset construction
    - Web scraping for images and description retrieval from AirBnB
  - **Java**
    - Application controllers implementation
    - Connection Manager classes, providing bridges between Java and databases
  - Java **FXML**
    - Application views design
  - HTML and **JavaScript**
    - Map view to show a location in a city
    - Heatmap visualization for specific analytic

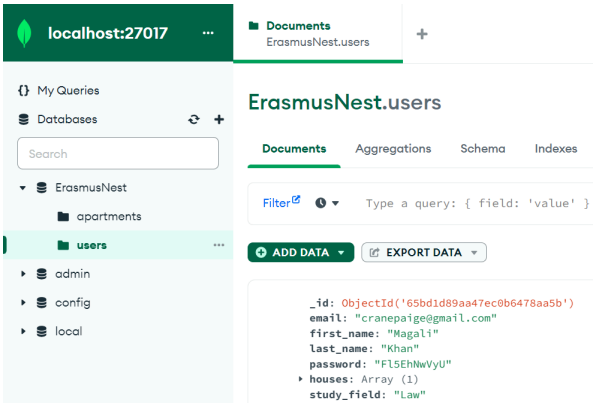
- **Maven**: dependencies management in pom.xml

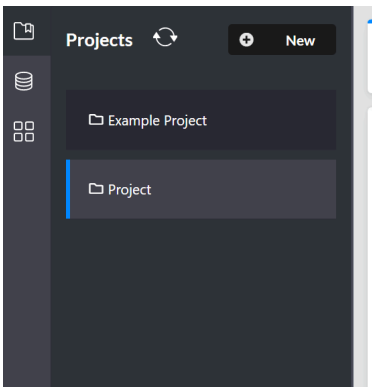
  Here are some main dependencies.

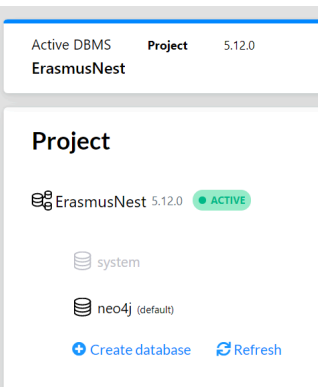
```
<dependency>
  <groupId>org.openjfx</groupId>
  <artifactId>javafx-controls</artifactId>
  <version>17.0.6</version>
</dependency>
<dependency>
  <groupId>org.openjfx</groupId>
  <artifactId>javafx-fxml</artifactId>
  <version>17.0.6</version>
</dependency>
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>4.3.0</version>
</dependency>
<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongodb-driver-sync</artifactId>
  <version>4.11.1</version>
</dependency>
```

```xml
<dependency>
  <groupId>org.openjfx</groupId>
  <artifactId>javafx-graphics</artifactId>
  <version>17.0.6</version>
</dependency>
<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongodb-driver-core</artifactId>
  <version>4.11.1</version>
</dependency>
<dependency>
  <groupId>org.controlsfx</groupId>
  <artifactId>controlsfx</artifactId>
  <version>11.1.2</version>
</dependency>
<dependency>
  <groupId>org.neo4j.driver</groupId>
  <artifactId>neo4j-java-driver</artifactId>
  <version>4.4.0</version>
</dependency>
<dependency>
  <groupId>org.openjfx</groupId>
  <artifactId>javafx-web</artifactId>
  <version>17.0.2</version>
</dependency>
<dependency>
  <groupId>org.openjfx</groupId>
  <artifactId>javafx-base</artifactId>
  <version>17.0.6</version>
</dependency>
```

- Database interfaces:
  - MongoDB Compass
  - Neo4J Desktop







2. Dataset

2.1. Data source

Regarding the data used, we utilized InsideAirBnB, a website whose aim is to provide datasets retrieved from the popular AirBnb website. Here, we could find various datasets for many **cities**, but particularly we focused on the following ones:

1. *Amsterdam*
2. *Barcelona*
3. *Berlin*
4. *Bologna*
5. *Florence*
6. *London*

For each one, we utilized the "*Detailed Listings Data*" and "*Detailed Review Data*" provided by the website; the first gives a complete overview of the **apartments** in the relative city, while the latter supplies many **reviews** for them.

2.2. Preprocessing

Since the datasets provided by our source were only raw data about apartments and reviews in a ".csv" format, some preprocessing **Python** operations were needed, and these has been the following:

1. Apartments file contains both apartment and host information, so was easy to use every entry to:
    a. Use apartment related fields to create an apartment object with name, position, number of accommodates, bathrooms, price and city composing the apartment object
    b. Use host related fields to create a user object with host name and email
2. Review file contains contains both review, apartment, reservation and user involved information, so was useful to:
    a. Use review information to create review object and associate it to corresponding apartment, considering date, comment and score
    b. Use user information to create a user object with host name and email
3. Password, unique email address and study field were randomly added for every user with the use of a Python library, since these information were not provided by the data source
4. A web scraping snippet to retrieve apartment's images URLs from AirBnB website was developed to have complete information also for apartments

5. At this point, we had the reviews and the apartment hosts. A merge between users from both Apartments and Review file was required to obtain the full *"User.csv"*, that was then added (in different ways of course) to both Neo4J and MongoDB
6. To conclude the MongoDB setup, a python script then proceed reading the ObjectsIDs assigned by Mongo to the apartments, and embedded them into their owner inside the user collection
7. Finally, in a similar way the "REVIEW" relations were added in Neo4J, by reading the precedent "*User.csv*" file, the *"Review.csv"* obtained from InsideAirBnB.com and the ObjectIds that Mongo assigned to the reviewed apartments.
8. To conclude, we generated some random data for the sake of the application purposes, and this included:
    a. Reservation for random users in random houses, and added them to Redis
    b. "FOLLOW" and "INTEREST" relations for Neo4J (these are used for the "suggestion" mechanism, so we needed at least some for testing purposes) .

2.3. Final Dataset

- **Users**: They are stored both in MongoDB and Neo4J, but in different ways. In the first we find a complete overview of the user entity, while in the latter this information is *lighter;* this is because in Neo4J we introduced users mainly to create relations on them ("FOLLOWS", "LIKE", "INTERESTS" and "REVIEW"), so we only required for the user the email field. An user with its password *can* also live in Redis, since this is used as a cache mechanism.
- **Apartments**: As for the users, the apartments can be found in both MongoDB and Neo4J, and for similar reasons as the ones from above. The full apartment can be found in the document database also in this case, while its (lighter) counterpart lives in the graph and is utilized mainly to check its relations; these can be LOCATED, INTEREST and LIKE.
- **Reviews**: the reviews are (as said) obtained from the starting csv, and in the end live only in Neo4J, as relations that connect Users and Apartments.
- **Reservations**: since these were not directly available from our data source, we generated a bunch synthetically through a Python script.
- **FOLLOWS, INTERESTS, LIKES**: these all represent relationships within our Neo4J graph database. Since they were not directly available in our initial data source in a form that met our requirements, we generated some synthetically. For each relationship type, we selected a random subset of users and established connections from these users to the relevant target entities (be it other users, cities, or apartments), according to their nature.

The final databases' sizes are the following:

- **MONGODB** (total ~140 MB):
    - 94k apartments
    - One million users (so, circa one out of ten users has an house)

- **NEO4J** (total ~405 MB):
    - 94k apartments
    - 725k users
    - 7 cities, which are the ones listed before plus Pisa, added for testing purposes
    - 300k FOLLOWS relations
    - 220k INTERESTS relations
    - 627k REVIEW relations
    - 10k LIKES relations

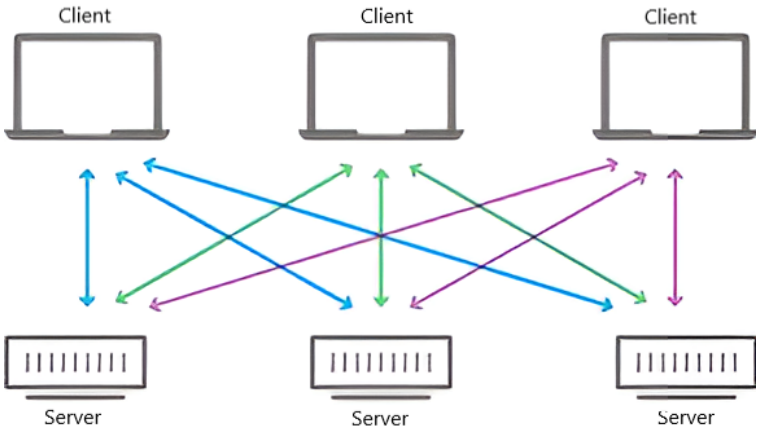
MongoDB size was found simply by checking the collections' sizes reported by Compass. Neo4J size was found through the following linux command

`du -hc /var/lib/neo4j/data/databases/neo4j/*store.db*`,

executed on the corresponding virtual machine (the one with ip 10.1.1.14).

3. Application design

First of all it is good to specify that a User for the client-side ErasmusNest application can be both a **student** and a **host**, there is **not a sharp separation between these roles**. So, once a User logs in can act as a student looking for an apartment to reserve for his erasmus experience, or as a host making available his own apartment. Also ErasmusNest provides **basic functionalities for not logged users**, requiring the sign up or login to enhance privileges and enjoy a better experience. Then the **administrator can consult statistics and run analytics** in order to extract useful information about the application data stored on servers.



Following chapter explains all application design choices, supported by Use case and UML class diagrams.

3.1. Actors and Functional Requirements

1) **UnLogged User**

Can look for an apartment and user without making any relationship with them. About functional requirements, system must allow UnLogged User to:

a) Sign-in, create an account providing personal information
b) Log-in, access the application using email and password
c) Search by users and view related profile page
    i) See reviews made by another user
    ii) See apartments made available by another user
d) Search by apartments and view related apartment page
    i) See related information
    ii) Access related reviews
    iii) See apartment on position map
    iv) Access host (other user) page
e) Access reviews made for an apartment

2) **Logged User**

After a sign-in, can browse-find-view for users and apartments and can also make relationships with them. About functional requirements, system must allow Registered User to:

a) Sign-in, create an account providing personal information
b) Log-in, access the application using email and password
c) Search by users and view related profile page
    i) See reviews made by another user
    ii) See apartments made available by another user
    iii) Follow another user
d) Search by apartments and view related apartment page
    i) See related information
    ii) Access related reviews
    iii) See apartment on position map
    iv) See calendar view for availability
    v) Make a reservation
    vi) Add to favorite ones (by clicking “like” button)
    vii) Get suggestions for similar apartments
    viii) Access host (other user) page
e) Access personal User page to:
    i) Modify personal information
    ii) Upload, Update or Remove an Apartment
    iii) See reservation requests for personal Apartment
        ➢ Accept or reject requests of reservations
    iv) See reservations made for other user Apartments
        ➢ Delete reservations he made
    v) See followers and follows
    vi) Remove other user from follows
    vii) Get suggestions for Users
f) Log-out

3) **Admin**

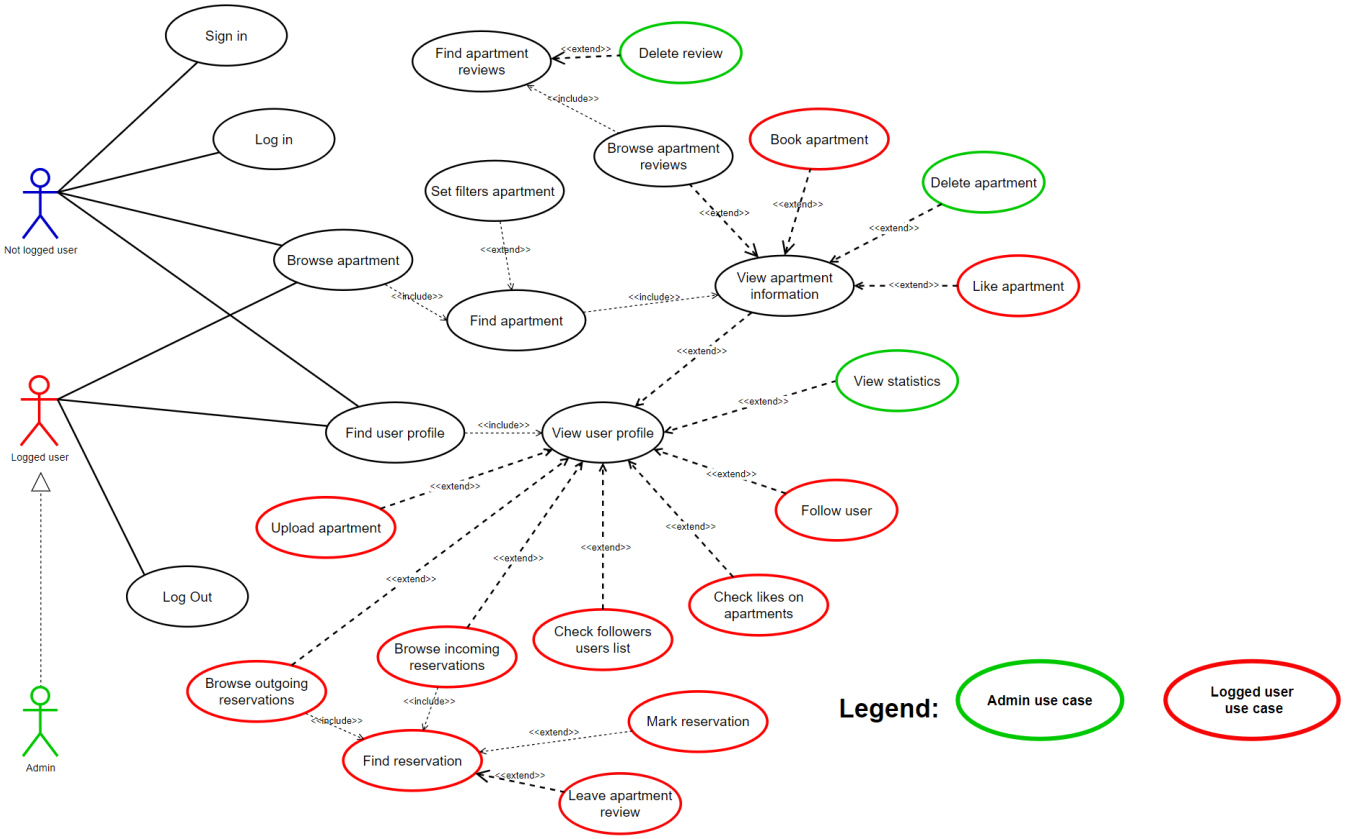
After a sign-in, can browse-find-view for users and apartments and make relationships with them, also responsible for guidelines checking. About functional requirements, system must allow Admin to:

a) All of the functional requirements allowed to Logged User
b) Remove an Apartment
c) Remove a Review
d) Access analytics page for statistics

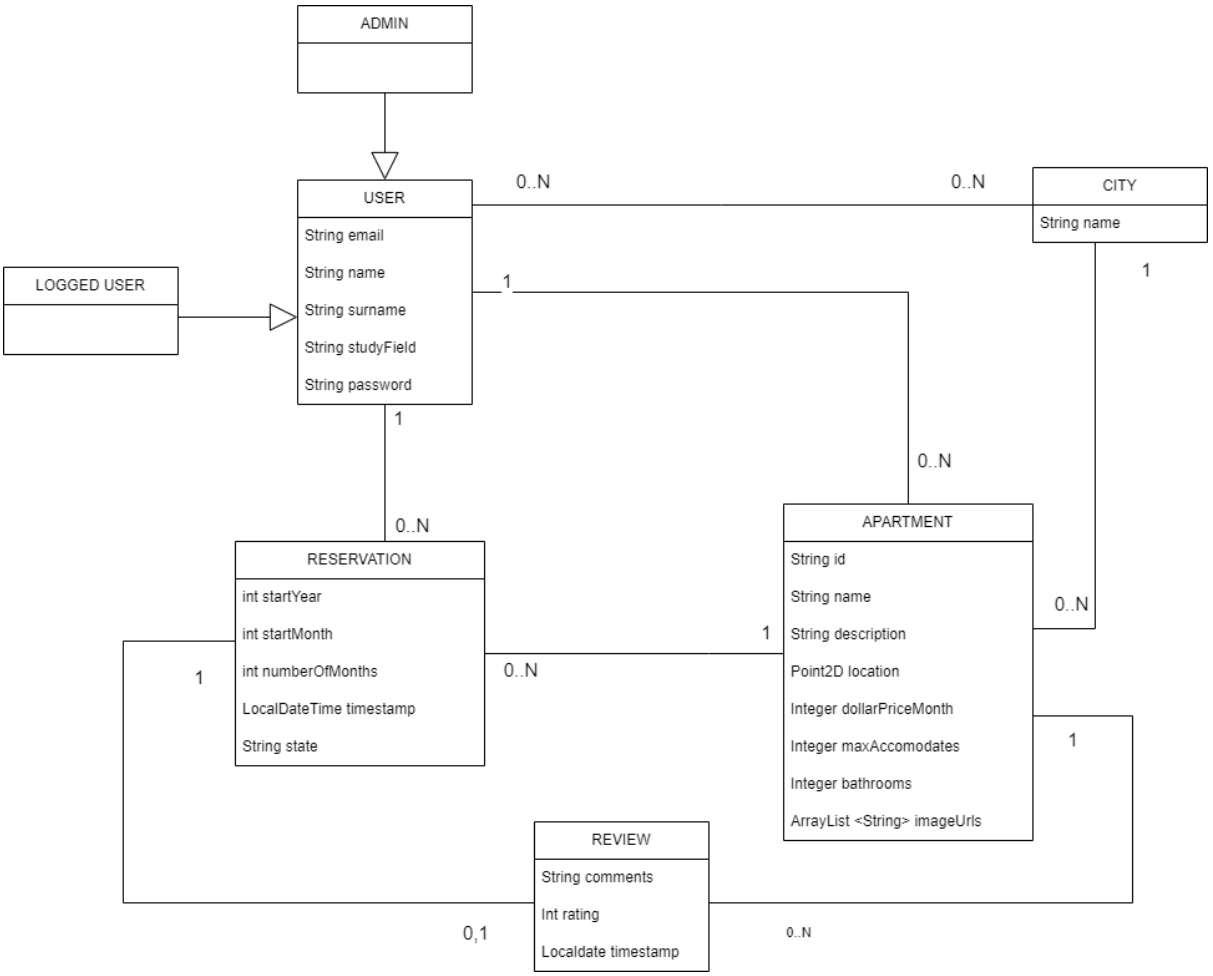
3.2. Non functional requirements

- Java application in a **user-friendly environment**, supported by simple guidelines
- **High availability** (following section will explain how to ensure this requirement)
- Avoid down time
- **Fault tolerance** for data loss
- **User caching** for simple and rapid application access
- The application is implemented using Java, an object-oriented programming language
- Appropriate **redundancy** to ensure rapid and low-latency CRUD
- User inter-connection through a **follower suggestion mechanism**, that allows to find other people that have been to the city the user is interested in an acceptable period of time
- The application manages **consistency** between databases of different types.

3.3. Use Case Diagram



3.4. UML Class Diagram



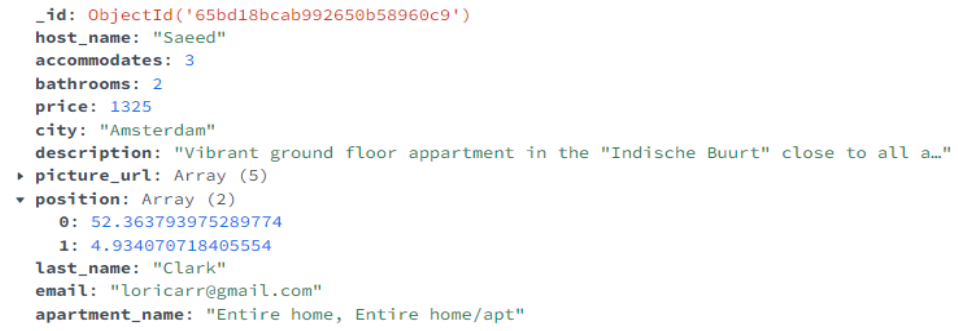
4. Database design

Handling many complex data structures and queries, the application was all developed over **non-relational databases** ensuring a rapid and simple way to access information even in a high volume of data.
Implementation choices are explained below.

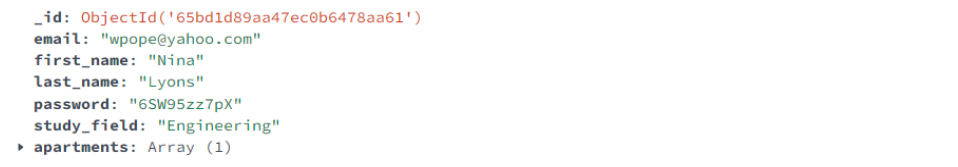
4.1. Document database on MongoDB

High performance, high availability and rich query language for CRUD operations let document databases deployed with MongoDB be the most suitable solution for User and Apartment objects handling.

- **Apartment, apartments collection**:



- **User, users collection**:

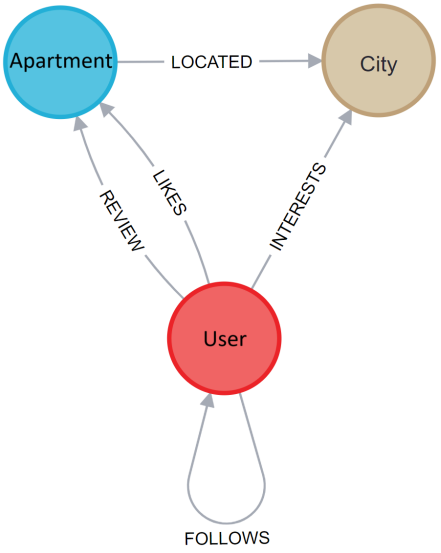


4.2. Graph database on Neo4J

As shown in the following scheme, Graph database deployed with Neo4J was used for follow, like, review and location relationships, exploiting main features as avoiding expensive join operations over high volume of data.

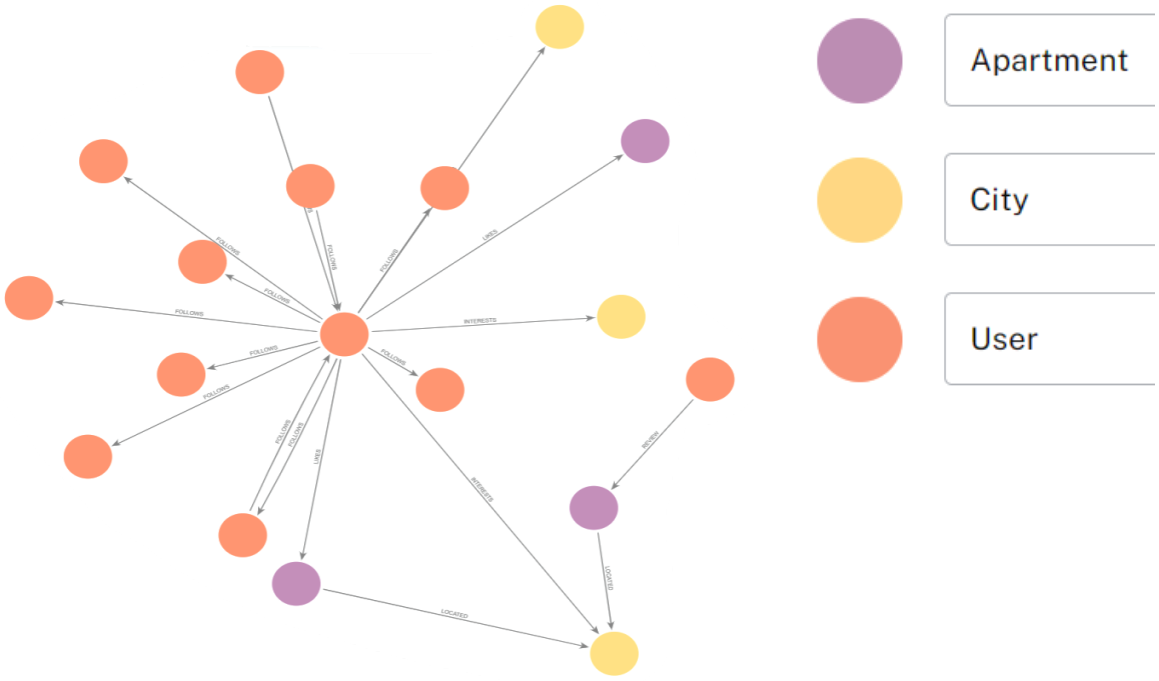
Relationship between entities:

- USER—FOLLOWS—>USER, no properties, user can follow multiple users
    - Relationship useful for easy user profile retrieval and user's suggestions implementation
- USER—INTERESTS—>CITY, no properties, user can be interested on multiple cities
    - Representation of user interests on a specific city, useful for other users looking for apartments in same city as your and for user's suggestions implementation
- USER—LIKE—>APARTMENT, no properties, user can like multiple apartments
    - Implemented to save favorite apartments and used for apartment's suggestions
- USER—-REVIEW—>APARTMENT, with date, score and comment properties, user can review multiple apartments
    - Easy way to have an experience evaluation, unlocked only while night stay is over. This relationship is also used for user's suggestions implementation
- APARTMENT—>LOCATED—>CITY, no properties, apartment can be located in one and only one city
    - Represents where apartment is



`CALL db.schema.visualization()` output

The image above is the output of CALL db.schema.visualization() in Neo4J desktop, which gives a visual representation on how the nodes inside the graph database are related.



4.3. Key-Value database on Redis

About the key-value database on Redis we reserved an entire chapter: Key-Value database on Redis.

4.4. Database redundancies

Since one of the main objectives of the project is to provide an easy and rapid way to access heterogeneous object data structures among different databases, one suitable way to satisfy such a requirement is redundancy among collection and databases.

1) Document database collections
    a) Since a Profile view provides information about both the host and the apartments owned, it was decided to replicate object_id, picture_url and house_name in order to avoid expensive search by clients whenever they want to operate over apartments collection

b) Name, surname and email are in both databases in order to show also host information over apartment view

2) Document database and Graph Database
    a) User is obviously replicated on Neo4J with only the unique email address
    b) Since Graph database is used for apartment browsing, and in that it's showed only an apartment resume, name and url for apartment picture are also present on Neo4J
    c) The city attribute is present in both Neo4J and MongoDB; in the first it's present as a node and it's mainly used for the INTERESTS relationship between a User and an Apartment, but also for the LOCATED relationship between an apartment and a city, while in the latter it's replicated inside the apartments collection mainly for analytics queries.

3) Document database and Key-Value database
    a) Using Redis in a *caching way* for user access, we replicated here usernames and their corresponding passwords.

More clear information about the redundancies we introduced, and how we manage them, can be found in the chapter Data distribution among databases.

5. Key-value database on Redis

For the purpose of storing data for some specific use cases we identify the key-value database as the most suitable architecture. So we decided to use Redis for the capabilities to store in-memory data structures, implement caching mechanisms thanks to eviction policies and persistence, and its speed for read operations.

5.1. Entities stored

The entities, identified considering the use cases for ErasmusNest, that can be lawfully stored on a key-value database are the **Reservations** for the apartments and **Users** that access the application. For our key-value system these entities are structured in the following way:

- Reservation
  - email [string] → the email of the student who made the reservation
  - apartment identifier [string] → the id associated with the apartment
  - start year [int] → it indicates the year in which the reservation starts
  - start month [int] → it refers to the month of the start year
  - number of months [int] → it expresses the duration in months
  - timestamp [datetime] → reports the moment in which it was booked
  - state [string] → it expresses the life stage of a reservation

- User
  - email [string] → the email that identifies a student
  - password [string] → the passcode used to log into the system

Note that we considered **only monthly reservations**, so the unit of a reservation coincides with one month that starts the first day of it and terminates the last one. This is a realistic scenario based on the use case referred to erasmus experience. Certainly in this way a student can reserve multiple consecutive months.

We could also store the reservation duration in a different way, for example using the end year and the last month related to it, but of course we preferred a solution that involves the fewest possible attributes, considering that they affect the memory usage or the key length, as reported in the next paragraph. The timestamp of a reservation is used to retrieve them in a sorted way.

About the state of a reservation we adopt a solution that expects **a specified set of states**: a single reservation can be *pending*, *approved*, *rejected*, *expired* or *reviewed*. This strategy allows us to manage all the use cases that involve a reservation. For example we permit a host to approve or reject a reservation, so until he does not mark it the reservation results as pending. After the last day of the whole reservation it will be signed as *expired*, then the student can review it and if he does it the reservation goes as *reviewed*. This last case permits us to allow only a single review of each reservation.

In reference to the User, considering the key-value architecture, we decide to store its credentials to implement a caching mechanism. All the details about the strategy we adopted are reported in the next section Caching Strategy and in the paragraph Impact of caching on user experience, please consult them for more.

5.2. Keys design

5.2.1. User key

Let's start from the simpler key we design, it is the one for the user entity.

user : *student_email* example → `user:johndoe@gmail.com`

For User entities we identify **the shortest meaningful solution** possible, this very short design **will use less memory**, but at the same time is also **strong enough not to cause conflicts**. The uniqueness of a User key is guaranteed by the presence of the email that is not replicable information among users. Observing the key structure it may seem that there is no reference to the user attributes, truly this is related to the choice we made to improve the key-value architecture efficiency, regarding memory, by adopting the Redis hashes (paragraph Redis hashes).

Let's move now to the more complex and interesting structure we designed for the reservations. For this one we thought of a frame strongly affected by the use cases that involve it.

5.2.2. Reservation key

reservation : *student_email* : *apartment_id* : *start_year* : *start_month* : *number_of_months*

example

```
reservation:johndoe@gmail.com:65bd18bcab992650b5896bf2:2024:3:2
```

As mentioned, guided by all the use cases that involve a reservation, we identify a key that contains a lot of useful information. First of all by reporting all this information into the key we ensure its **uniqueness**, minimizing at the same time the complexity and the amount of the code in charge of managing reservations.

To fully understand the utility of this structure we have to consider a specific use case: the one related to the **calendar construction for an apartment**.
The ErasmusNest application provides the capabilities for browsing apartments and once you have found an apartment of interest you can reserve it for a certain period only if it is available. To clarify, an apartment is considered available only if:

- who wants to reserve does not have yet a booking in the same period;
- the apartment is fully available or there are still free places (consider that each apartment has a maximum number of accommodates and we manage a solution that allows overlapped reservations);
- the start date and obviously the end date are not elapsed.

So each time we need to build the calendar for an apartment, which is a frequent operation, **we need only to retrieve all the** related **keys** to avoid accessing values. This is the real advantage of using this design, every time a user needs to reserve an apartment is sufficient to retrieve only keys **instead of scanning all the keys and checking values**. It impacts a lot on the user experience due to its speed, simplifying on the other hand the code written to build a calendar.

Of course the length of a key as this one will use more memory compared with the one designed for the user, but this permits to improve the efficiency of the most frequent read operations. To face the memory usage issue we decided to introduce a solution that allows us to **design shorter keys, reducing at the same time the number of keys** stored on Redis, thus simplifying also the search of a key considering that is a frequent operation in the case seen now for the calendar.

At this scope it is fundamental to consider that these **designs are strictly related to the efficient hashing mechanism** we adopt to store information in the key-value database. About the hashes and why we adopt them is explained in the following paragraph.

5.3. Redis hashes

Delving into the value store on Redis we came across hashes.
According to the Redis documentation "*Redis hashes are record types structured as collections of field-value pairs. You can use hashes to represent basic objects and to store groupings of counters, among other things.*"; about the performance they specify that "*Most Redis hash commands are **O(1)**.*" and that "*It is worth noting that small hashes (i.e., a few elements with small values) are encoded in a special way in memory that make them very **memory efficient**.*". Regarding limitations the only thing they highlight is that "*Every hash can store up to 4,294,967,295 (2^{32} - 1) field-value pairs. In practice, your hashes are **limited only by the overall memory** on the VMs hosting your Redis deployment.*"

Again in Redis documentation we found many best practices about memory optimization, most of these involve hashes. We report directly a short fundamental paragraph in relation to our use case.

Use hashes when possible

Small hashes are encoded in a very small space, so you should try representing your data using hashes whenever possible. For instance, if you have objects representing users in a web application, instead of using different keys for name, surname, email, password, use a single hash with all the required fields.

Assuming that it could be very useful for the advantages that we will list later, we looked into concrete cases that use redis hashes.

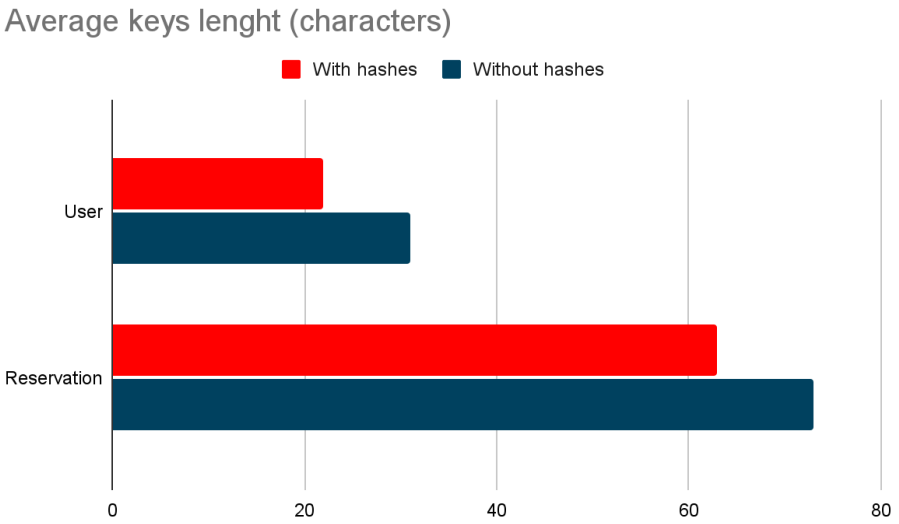
So we found interesting articles about memory efficiency.
For example "Hash Structure of Redis for Millions of Keys" in which is reported: "*Pieter Noordhuis, one of Redis' core developers, says that Redis hashes are one of the best ways to store key-value pairs.*"
Also Mike Krieger, Instagram co-founder, published an article in which he explains how Instagram uses Redis hashes to face memory usage issues.

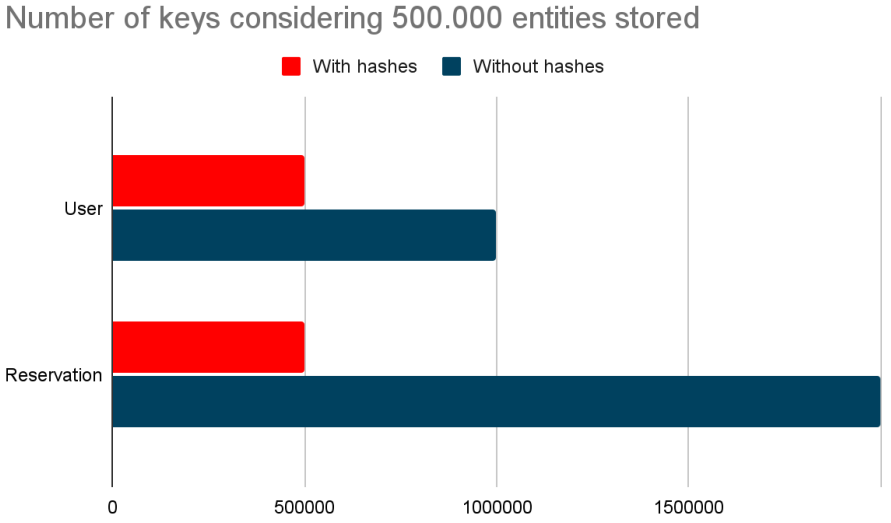
In sight of this, to store our entities in the key-value database we decide to adopt this approach, experiencing many advantages:

- ✔ reduced length of the key
- ✔ reduced number of keys, actually is enough a single key for each entity
- ✔ better memory usage and storing optimization for intrinsic hash properties
- ✔ simpler search among keys that means simpler code that works with them

Beyond words, let's see some diagrams that express the key's dimensions.

5.3.1. With or without hashes comparison





5.3.2. How to deal with Redis hashes

Here methods used to deal with a hash, analyzing the computational complexity.

- **hgetAll** (*key*)
  - time complexity → **O(N)** where N is the size of the hash.
  - we used it to retrieve all the hashed values for a single key

- **hget** (*key*, *field_name*)
  - time complexity → **O(1)**

- **hset** (*key*, *field_name*, *field_value*)
  - time complexity → **O(1)**

- **hdel** (*key*, *field_name*)
  - time complexity → **O(1)**

These are the main operations we exploit dealing with hashed values. So the **computational complexity is not increased** with respect to the use of *get* and *set* methods.

5.4. Redundancies introduced and final entities structure

Ascertained that the main attributes for the User and the Reservation entities are the ones listed in the paragraph Entities stored, we saw in the paragraph Keys design how for some attributes of a reservation it is better to embed them into the key. Starting from this, since we adopt Redis hashes, the remaining attributes will be stored in an encapsulated manner into the entity value.

Value associated with a User key user : *student_email*

- password
- reservedApartments [string] → contains the ids for apartments reserved by the user

Value associated with a Reservation key reservation : *student_email* : *apartment_id* : *start_year* : *start_month* : *number_of_months*

- timestamp
- city [String] → name of the city in which the apartment of the reservation is located
- apartmentImage [string] → URL of the main image corresponding to the reserved apartment
- state

The underlined attributes are the ones that represent **redundancy**, so now we will focus on them. Let's start right away by saying that *reservedApartments* is introduced to solve an issue related to the clustering we deploy, so the motivation of its introduction is reported in the next paragraph. About it here we can just say that contains the identifiers for the apartments actually reserved by the user, these ids are separated by ",". We assumed that a user can have one or however few reservations (people usually engage only once or twice in their lifetime with an Erasmus experience), so managing this attribute will never be too complicated. Note that until the user makes his first reservation this field does not exist, saving memory space.

Talking about the Reservation value the *city* and the *apartmentImage* are proposed in order to **retrieve all the necessary information only querying Redis** in these use cases:

➢ when a host consults incoming reservations for his apartments
➢ when a student consults reservations he made

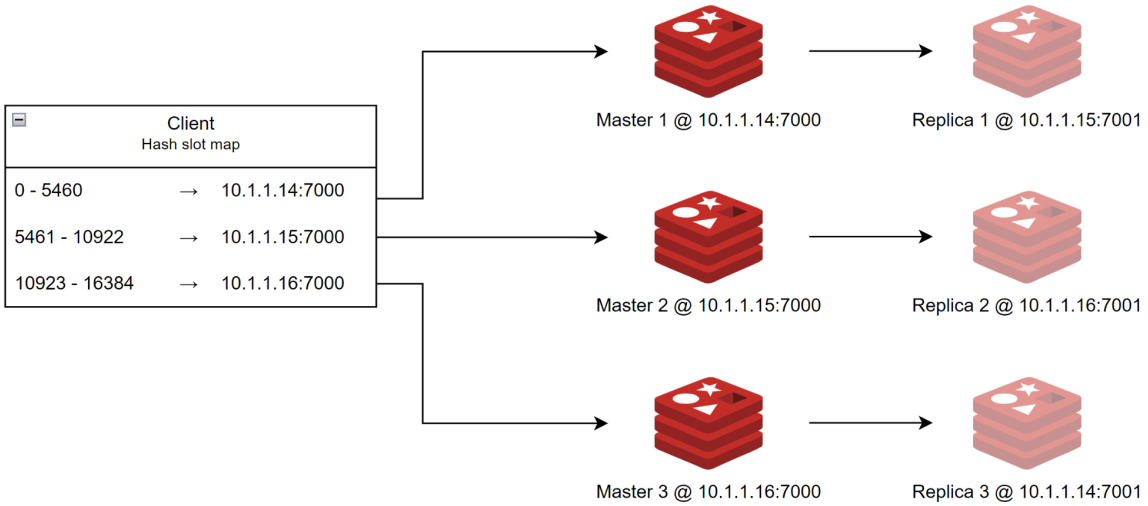
In these cases the name of the city where the apartment is located and an image of it permits to simply identify it. This is because the apartment's name could be not unique, instead the photo certainly is. It is also about information that require a low degree of consistency considering the user experience, we mean:

- the **city** will never changes, so this field **has no need to be updated**;
- the main **image** of an apartment certainly could be updated, but even if the user does not see the most recent one is not something that affects the experience with the application, so **can be updated eventually**.

5.5. Design motivations related to clustering

5.5.1. Clustering on Redis

As requested by the project guidelines, Redis was integrated in the project by exploiting its clustering functionalities. The deployment of the cluster was done by following this guide, and consisted in utilizing the three virtual machines kindly supplied by the University to do so. We decided to implement it by endorsing six Redis instances, organized in the following way :



Basically, to develop the cluster architecture, Redis required *at least one* replica for each master and so we provided the, as shown above; the virtual machines organization is also highlighted by the output of `redis-cli -p 7000 cluster nodes` that can be executed on any of the three virtual machines.

```
root@Large-Scale-Exam-2024:~# redis-cli -p 7000 cluster nodes
f134a6ed8d36ec0c5657677a13e9e4320c5724e4 10.1.1.16:7001@17001 slave 1788c21a7667fd0a34d361684749ae9f05b41659 0 1708034459861 3 connected
1788c21a7667fd0a34d361684749ae9f05b41659 10.1.1.15:7000@17000 master - 0 1708034459000 3 connected 5461-10922
65c7d12d2c100bbd8ab7113fa1c28167df78138e 10.1.1.14:7000@17000 master - 0 1708034459000 1 connected 0-5460
e976c0dcd2e2886f389924c218d6bc5e13301aef 10.1.1.15:7001@17001 slave 65c7d12d2c100bbd8ab7113fa1c28167df78138e 0 1708034459560 1 connected
0ea7aaa47490afc4560206cd56592aa36bbdbcc2 10.1.1.14:7001@17001 slave ae8b7d5aab9a51b3a303bad116c851a6f4bd8962 0 1708034459000 8 connected
ae8b7d5aab9a51b3a303bad116c851a6f4bd8962 10.1.1.16:7000@17000 myself,master - 0 1708034458000 8 connected 10923-16383
root@Large-Scale-Exam-2024:~#
```

As we know, by exploiting the cluster functionalities, Keys are distributed among nodes, and this is done by computing the hash slot to which the object in exam must be stored. We also know that the formula is the following:

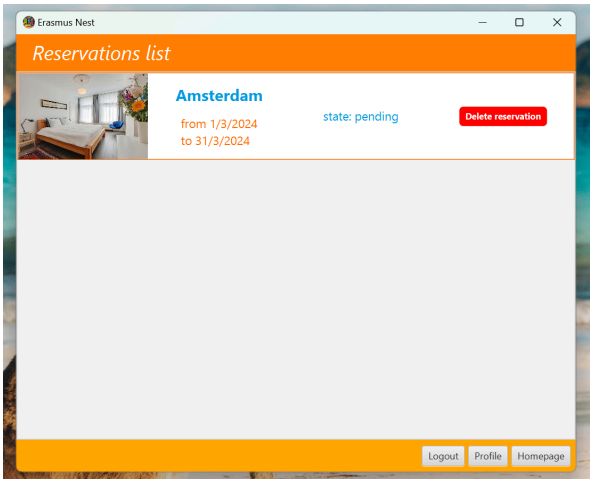
$$HashSlot = CRC16(Key) \bmod 16384$$

and this is coherent since we have available exactly 16384 slots as shown above.
The problem here is that the Redis client/connector *directly computes* the hash slot for a key to determine a Master node to work with; this raised a challenge for us, since we initially decided to introduce Redis only for reservations and for caching purposes.

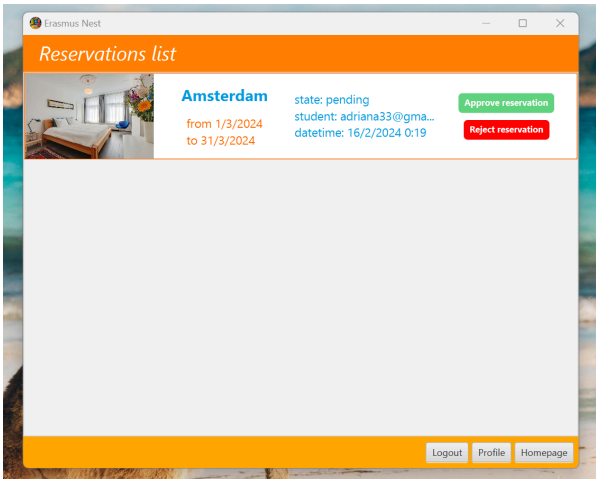
5.5.2. Strategy for searching keys among the cluster

5.5.2.1. Introduction

So, let's dive deeper into the problem; in our application users can book apartments, but users can also *host* apartments.



User reservations list page



Host incoming reservations page

The Redis key for the reservations is the following (as already said in its correspective paragraph, but reported here for the sake of simplicity):

reservation : *student_email* : *apartment_id* : *start_year* : *start_month* : *number_of_months*

so the questions are now the following:

1. How can hosts make a call to get all the reservations for their apartments? They of course don't have *a priori* access to the student_email field
2. How can students get all the reservations they booked? They don't have the information about the apartment_IDs.

This might seem trivial with the usage of the Redis' `KEYS` command, but it's really not when the clustering functionalities are introduced; it's not possible to use the command like this, to (e.g.) obtain all the reservations for an apartment

```
KEYS "*:apartment_id:*" ,
```

since the Redis client will try to hash `"*:apartment_id:*"` as the key, and will bring you to whatever *slot* this hash-es to. Facing this kind of problem, we analyzed the Redis documentation deeply and found out that it was possible to "inscribe" some parts of the keys between curly brackets, in order to ensure that the hash will be computed utilizing *only* that part of the key.

5.5.3. Curly brackets usage

Now, with this knowledge, we had to keep the key structure to ensure its unicity (for example, swapping the user email with the host's one would have not led to unicity). With this in mind, we were then forced to embed the apartment'IDs inside curly brackets, since this is the only common entity that both users and hosts have in common. This is also the reason why the redundant field reservedApartments is introduced in Redis, because otherwise users' clients wouldn't have any way to compute the correct keys' hashes and obtain the right slots to which read their reservations. With this setup, what happens is now the following:

1. Hosts can look up for the reservations on their apartments by simply looking for keys that have this structure → `" * {: apartment_id :} * "` , and this *guarantees* that they will fall inside the correct slot
2. Users upon login *will have to check* if they're inside the Redis cache, and if they're present their corresponding value is read. If the inserted password matches the one in the database, also the *reservedApartments* field is read, split on commas (since it's a string) and saved inside the current navigation session. Trough this mechanism:
    a. if it's empty and user requests his/her reservation during the application lifetime, Redis is not even called since we already know that there aren't any (the field was checked on login)
    b. If the current session contains apartments' IDs, it's because some were found through the login phase. If the user then requests to check for his reservations, we call the key-value database exactly as an host would do, by hashing `"*:student_email{:apartment_id:}*"`, and this again ensures that the correct slot is found, because of how the curly brackets mechanism works.

This redundancy can considered quite efficient, since we know that:

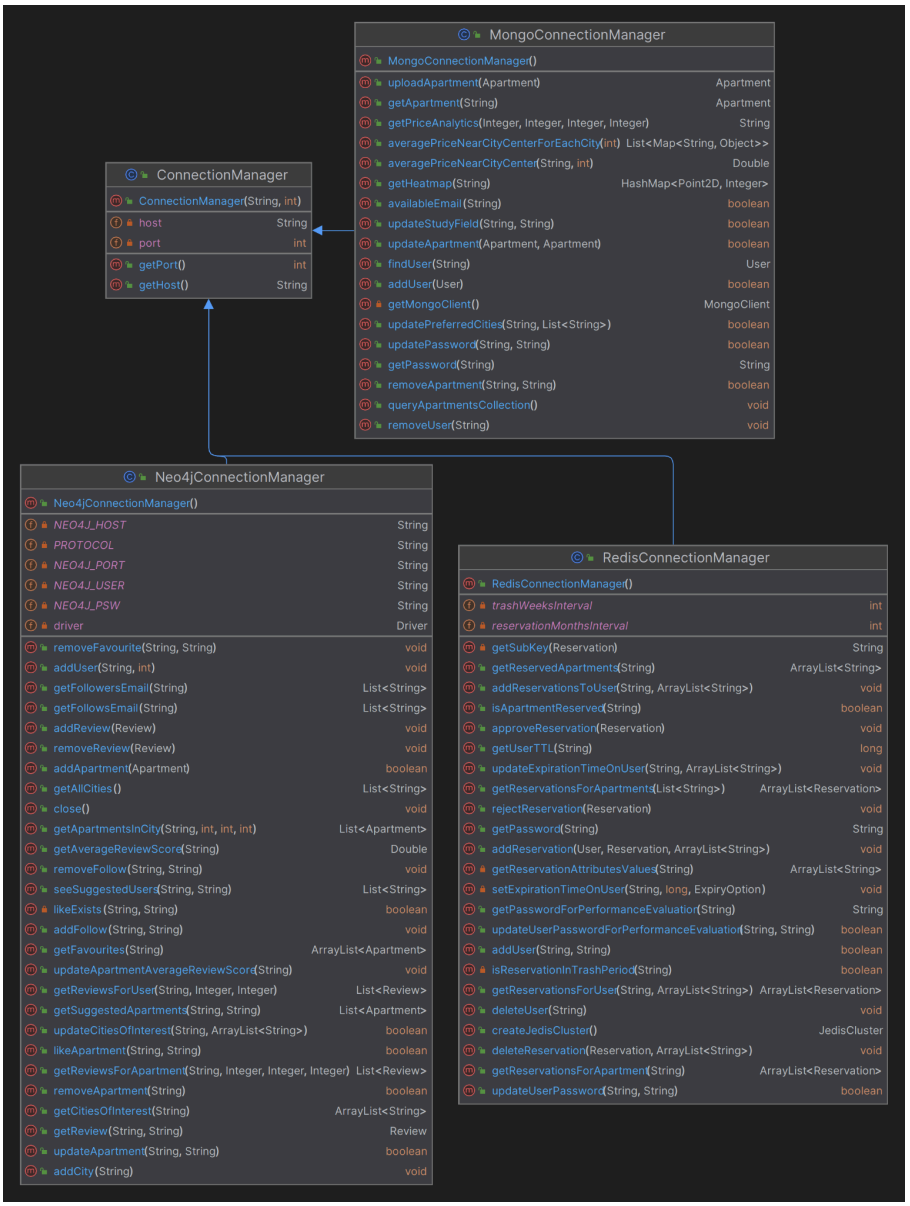
- If the user is booking a reservation, he surely logged in the application (so he will be inside Redis cache and will only be necessary to update one field through the `HSET` mechanism)
- Upon login, the application will call Redis first anyway to check for an existing cache entry.

6. Code design and implementation

6.1. Java Application Architecture

MVC design pattern was used to ensure simple development of the application, allowing Model View and Controller parts separation.
Since the application requires database access logic, this was implemented by adding another package, *dbconnectors*, that includes a class for every kind of database. All of the following classes extend the ConnectionManager.

- MongoConnectionManager: contains code for CRUD and analytics operations over MongoDB
- Neo4JConnectionManager: contains code for CRUD and analytics operations over Neo4J
- RedisConnectionManager: contains code for CRUD operations over Redis

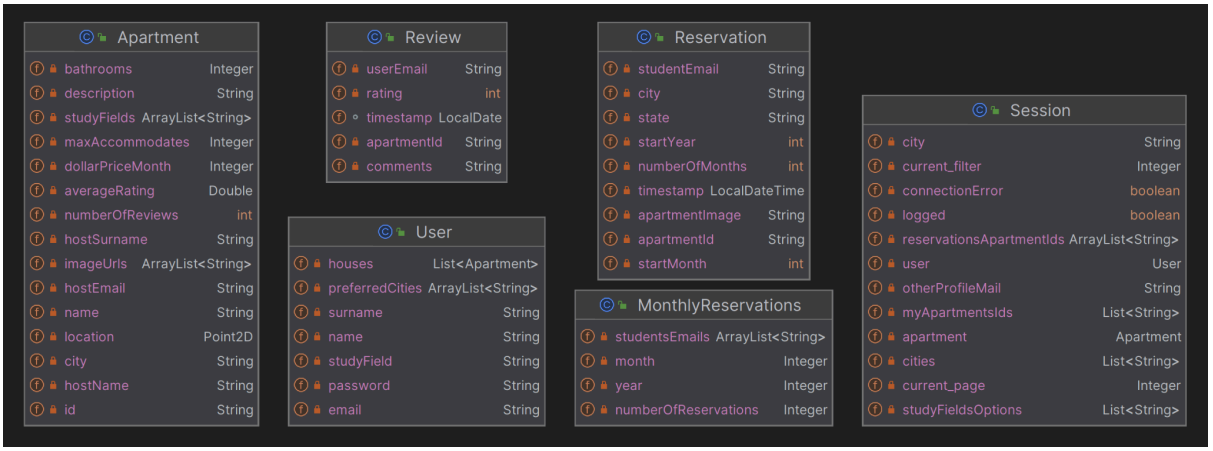


The *dbconnectors* package

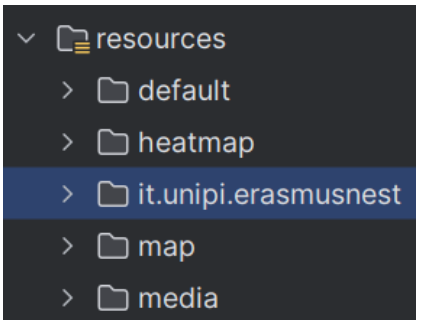
6.1.1. Model

This package represents the application's data and business logic and is responsible for retrieving and storing data and performing various operations on the data. Every class has related constructors, getters, setters and toString.

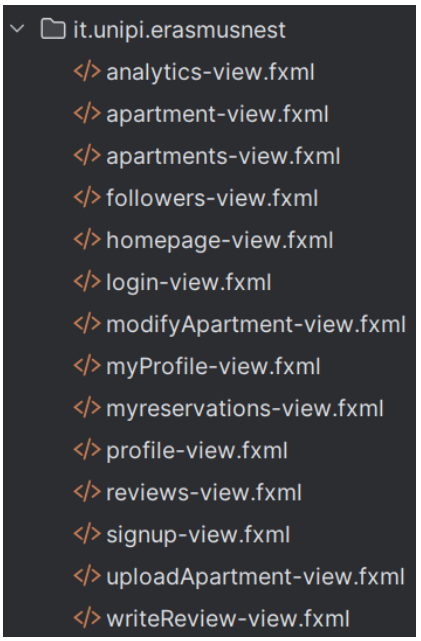
- User: represents User entity, with relative information
- Session: encapsulates not only session information for logged users but also list of all cities upload at homepage stage (List<String> cities), list of possible study fields (List<String> studyFields) and the logic for views flow (currentPage) using a stack
- Apartment: contains all apartment information
- Reservation: instance of reservation containing all information conceived
- Monthly Reservation: class for an easier Redis reservation updaiting, containing time and tenant information
- Review: contains information about who made the review (User), for which apartment (Apartment) and when (date)



6.1.2. View



View is responsible for presenting the data to the user and handling the user interface, combining FXML static elements and Java dynamic ones (embedded into Controller related class).

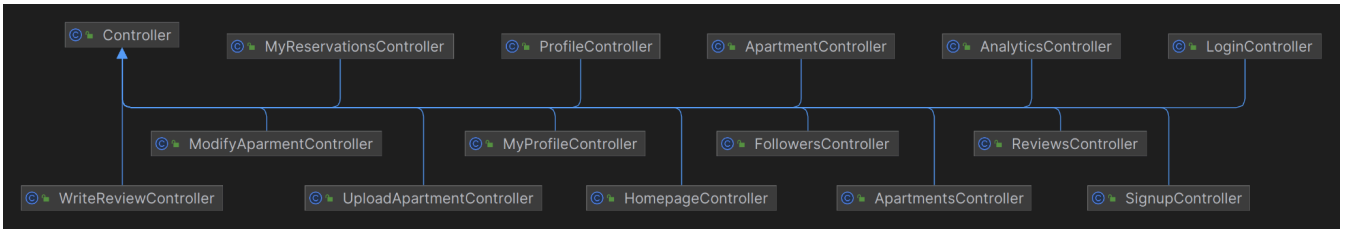


- login: first view for every instance of the user, in which he can choose between login, signup or even continue without logging
- signup: provides an easy registration form
- homepage: search bar for user and apartment browsing, including signup, login, profile buttons and a brief user guide instance
- myProfile: section for personal information and apartments updates, including easy way to see followers and follows relationship between other users, favorites apartments with related suggestions and linking for both reservations made for other-user-apartments and reservation requests for yours
- followers: separated section accessible by myProfile view for follows relationship
- myreservations: for both reservations made by the user for an apartment and other user reservations made for his apartment
- otherProfile: is for not-your profile page, includes a user's resume and related links to
    - See other profile information, including follow option with related suggestions
    - Look for other profile reviews
    - Search between other profile apartments
- apartments: shows the list of apartments the user is looking for
- apartment: shows all information about apartment, including a position map, a calendar view and options for
    - Make a reservation
    - See reviews
    - Go to host profile view
- modify and update apartment: providing related forms for uploading and updating apartment options
- reviews and writereview: the first is called whenever a user want to see apartment's review, the latter for writing one
- analytics: administrator-based view, for analytics queries

6.1.3. Controller

It interacts with the Model to retrieve or modify data and accordingly updates the View.
All of the following classes extend the Controller class, which provides methods for window changing and Connection Managers access.

- Login: check login fields and, if corrects, allows login and store needed information in Session object. Providing caching option, can search both Redis and Mongo if needed
- Signup: check user information fields, if corrects the account is created using MongoConnectionManager to insert a user instance into User collection
- Homepage: can browse for both User and Apartments accessing different collections via MongoConnectionManager, different filter options allowed
- MyProfile: access MongoDB for user information showing and updates and provides Analytics, Followers, MyReservations, Upload and Update Apartment page linking
- Analytics: recalls MongoConnectionManger to satisfy Admin advanced analytics will
- Followers: access Neo4J database for FOLLOWS relationship retrieval, provide function for add or remove one or access other user profile
- MyReservations: access Key-Value database via RedisConnectionManager for reservations retrieval
- UploadApartment and UpdateApartment: write on document database Apartment and User collections since document embedding is exploited
- Apartments: read on graph database Apartment node in order to retrieve partial apartment information
- Apartment: read on document database Apartment collection in order to retrieve complete apartment overview, including host page linking, and read and write operation among Redis for reservation handling. Also a way to read apartments reviews is also provided
- Reviews: retrieve Review accessing REVIEW edges on Neo4J
- WriteReview: in order to create a REVIEW edge for experience evaluation using an ad-hoc form, unlocked only at the end of the night stay
- Profile: simple way to show other user information accessing document database via MongoConnectionManager

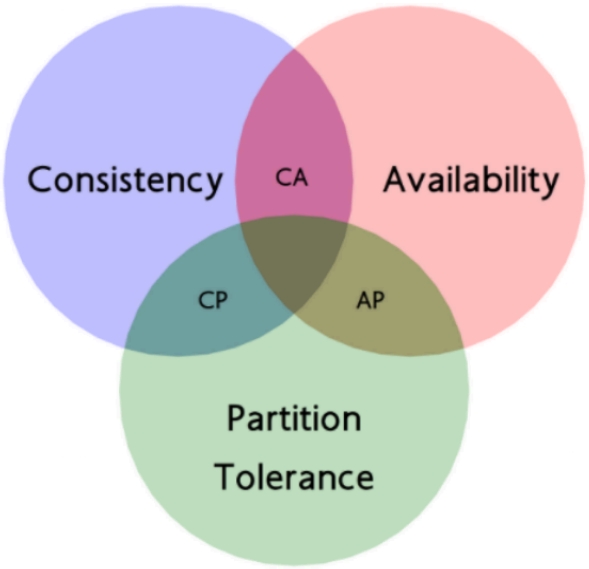


7. CAP Theorem Issue

ErasmusNest application's use cases require:

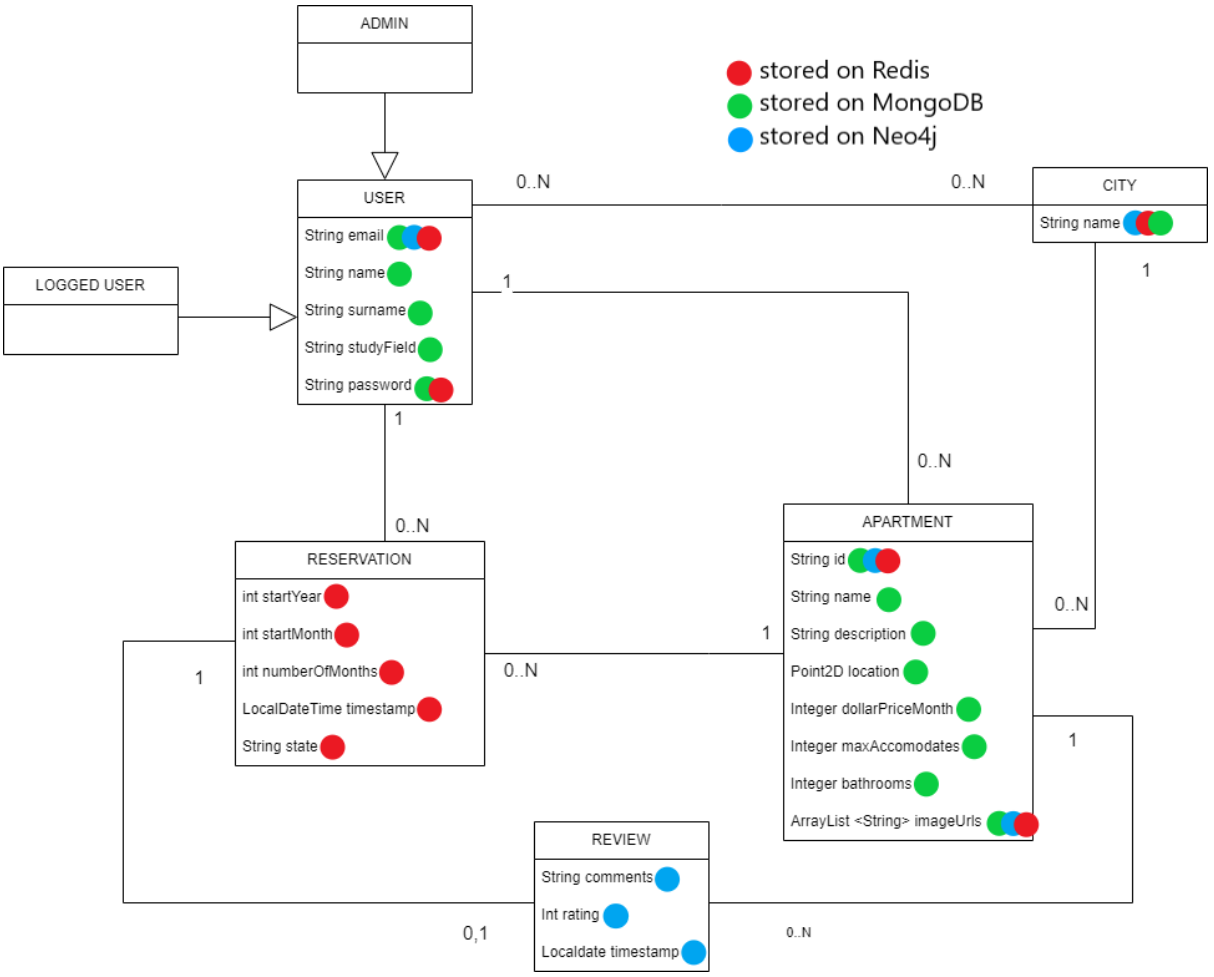
- the **high availability** to allow each client to access data in every moment;
- responsiveness, so **low latency**;
- the system remains **functional despite** physical **network partitions**.

To ensure all these characteristics, referring to the CAP theorem graphically reported below, the model to adopt is the "*AP*" so combine *Availability* with *Partition Tolerance*. In this manner the system can act respecting the **Eventual Consistency** paradigm.



The consequence of this choice is that the system will be available even if under partitioning, to the detriment of the complete consistency of data. It means the main drawback of this approach is that the **user could not read accurate information**, but *eventually* these will be made consistent.

7.1. Data distribution among databases



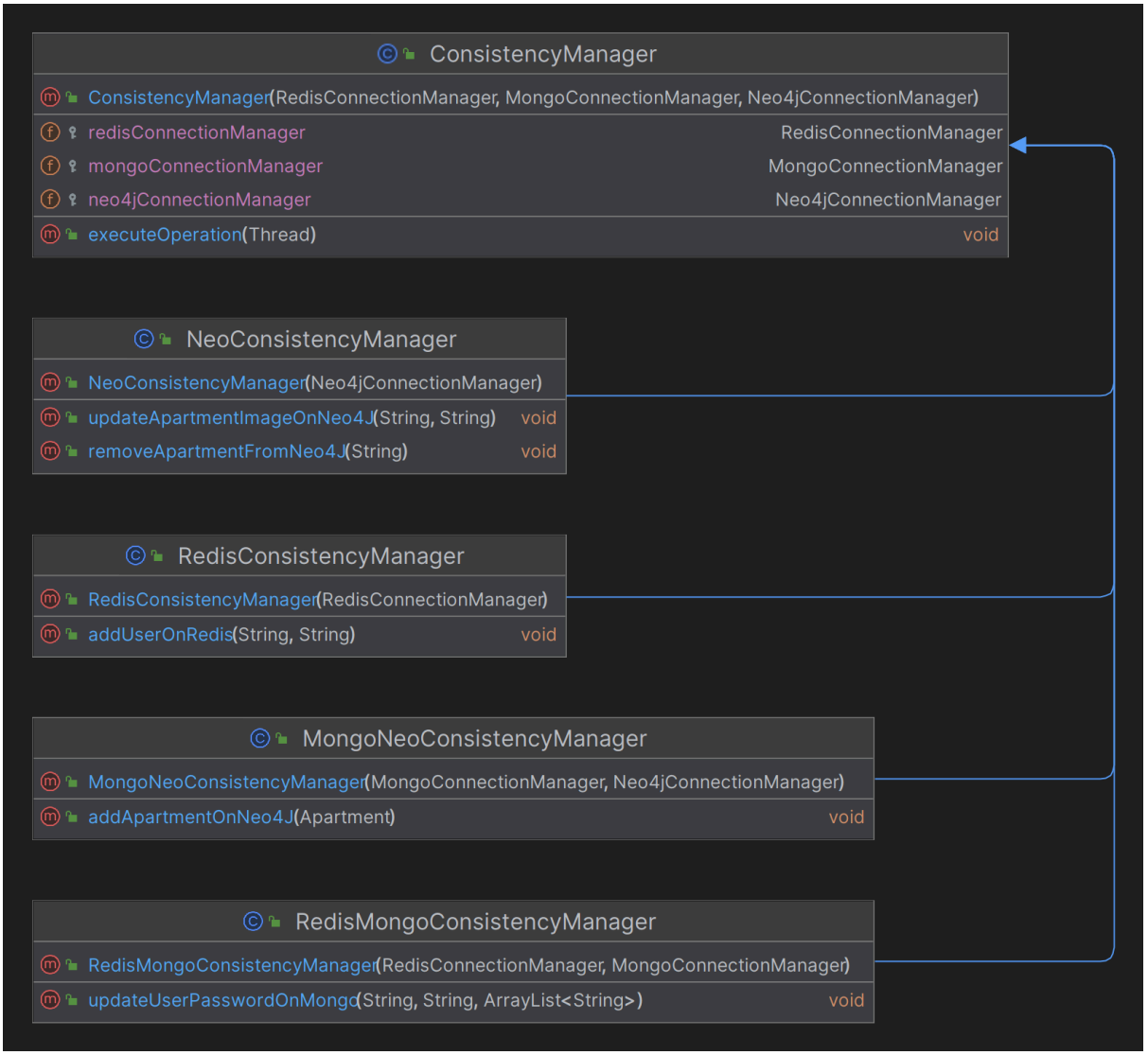
As we can observe, the storing strategy pursued introduces **some redundancies**, which allow **to improve the application efficiency** in relation to its use cases. Note that the redundancy of the image urls for an apartment is actually implemented only for the main picture, reducing the load of this data replication. User's credential are deployed also on Redis only in case of caching (this strategy is explained in the next chapters). It is therefore clear that an effective replication involves the city name and the main picture of an apartment. About this, it is fundamental to note that the city name is a static information, due to the fact the location of an apartment would not change, never.

The next paragraphs describe the solutions adopted to manage the consistency among data deployed on different database types.

7.2. Consistency between databases

7.2.1. Eventual consistency using Java Threads

Considering the redundancies among databases, a solution to keep the consistency of data distributed across different databases was needed.
Since ErasmusNest is a client-side application, a **client-side mechanism** was built to update modified data **with an eventual consistency logic**.
First of all we implemented some structured Java classes in charge of managing, using threads, all necessary operations. Here below the class-diagram for these.



As we can see we prepared a single class for each dependency among different database types depending on an initial design to modularize the eventual consistency strategy. In this way each ConsistencyManager can be instantiated "*on the fly*" to invoke the needed method that starts a thread.

As result observing the methods of the previous classes, the use cases involved in this strategy are:

a) when a host correctly updates the main image referred to an apartment;
b) when a host correctly removes an apartment;
c) when a user logs in and he is not already cached on the key-value db;
d) when a host correctly uploads a new apartment;
e) when a user updates his credentials.

Let's now analyze what happens in detail for each of the above cases:

a) the image will be updated on the document database;
b) the apartment will be removed from the document database;
c) the user's password will be read from the document database;
d) the apartment will be added to the document database;
e) the user's password will be updated on the key-value database;

What should happen in correspondence of these cases:

a) the image will be **eventually updated** on the graph database;
b) the apartment will be **eventually removed** from the graph database;
c) the user's credentials will be **eventually added** to the key-value database;
d) the apartment will be **eventually added** to the graph database;
e) the user's password will be **eventually updated** on the document db;

These operations, reported in the last one list, are performed eventually and in a ***non-blocking*** manner with respect to the user application experience flow. This property is obtained performing the operations using Java Threads. In particular what the system does is the following: once an operation is correctly executed on a database, a thread starts to perform the same operation on the other database on which data appears as redundancy. In this way, the **application flow for the user will be never paused** in order to wait for consistency among databases.

There is a specific case that is worth analyzing in detail, it is the case e) in which a user updates its credentials. Whose operation in relation to the caching mechanism is explained in paragraph Credentials consistency and caching.
As reported above, when a user updates his credentials the change will be executed immediately on Redis and the user will be marked as "*Mongo-inconsistent*" using the time-to-live associated with its key (more specifications in Credentials consistency and caching), then starts the thread in charge of updating it on MongoDB. At the *eventual moment* in which MongoDB will be correctly updated the same thread will return on Redis to mark again the user, this time as "*Mongo-consistent*" even working with the key's TTL.
But, what if the thread fails? There is no problem, eventually the update will be completed: the next time the same user logs into the application the system will detect that it is marked as "*Mongo-inconsistent*" and the thread will be started again. This will be repeated every time until a consistent situation is reached.

7.2.1.1. The rollback logic

Of course, to be fully consistent, we also considered the case in which fails the thread in charge of reflecting the operation on the redundancy. How is managed the case related to user credential is yet reported in the paragraph above.
Let's analyze what happens in any other case.

What the *consistency thread* does if its operation fails, in the cases d), is to take care of a rollback procedure in order to abort the insertion on the documentDB, it means that the newly added apartment will be removed.

If the thread that performs the operation in the case c) fails, nothing will be done. This is possible because the next time the same user logs in the application this thread will be started again. Obviously in case of reiterated errors this procedure will be revived until user credentials will be correctly stored on the key-value db.

Also about the case b), if the related thread fails, nothing will be done. Simply because the same thread will be invoked again when any user tries to access the apartment page for that one already removed from the document database. He will receive a warning stating that a server error was triggered, because the apartment is no longer available, and his page is simply reloaded.

The last case a) does not foresee anything if the related thread does not complete his task correctly. We assume that if the users still examine not the latest main photo for an apartment, the user experience will not be affected. So on the graph will be stored the previous single image until the host newly updates it and the thread correctly executes its work.

7.2.2. Other eventual consistency strategies: Document and Graph

Talking about other consistency strategies, we decided to utilize the following to manage graph and document databases user management; the main conclusion is that it is not mandatory or even necessary to have two user's instances on both databases at the same time unless the user has no relationship. Avoiding highly-coupled entities allows a **non-strict-consistency**, as described in following workflow: the main conclusion is that *it's not necessary* to have them at the same time in both, so we loosened this constraint in order to not have a strict-consistency between the two. Basically what happens is the following:

1. Upon user creation through the *signup* mechanism, a document is formed and placed inside MongoDB with all the information inserted by the client (credentials and optional study field)
2. At this stage, the user is then *not* already inserted inside Neo4j.
   User creation won't happen until an *explicit* request between one of these following:
   a. user starts following another user
   b. user is followed by another user
   c. user likes an apartment, adding it this way to related saved ones
   d. user adds some cities of interests through related profile page
   e. user reviews an apartment he has been in Erasmus in

This is done by using the "MERGE" command in the graph database queries instead of the "CREATE" one for the user nodes, since this checks for entity existence upon creation, and only if necessary creates them.
This way they can live separately in the databases.

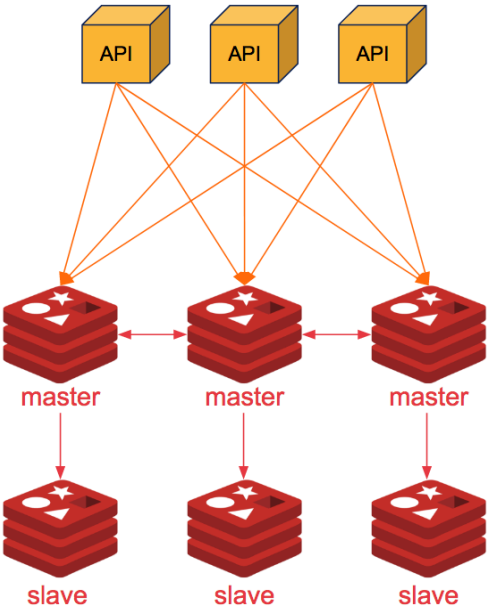
Is also important to explicitly describe apartment management between document and graph databases, described in following workflow:

1. After a removal request from the user application side (which can be done via the "modify" section on "my profile" for a specific apartment), the request is only handled on MongoDB
2. At this stage, if some other user is looking for an apartment in same city can see the apartment preview
3. The removal request will subsequently processed on Neo4J if and only if another user will try to look for further information about the apartment using specific apartment page: a not-found message will appear on application side (because apartment was previously removed from MongoDB, see above), then the GraphDB request will be process and the apartment removed also from Neo4J

7.3. Consistency between replicas

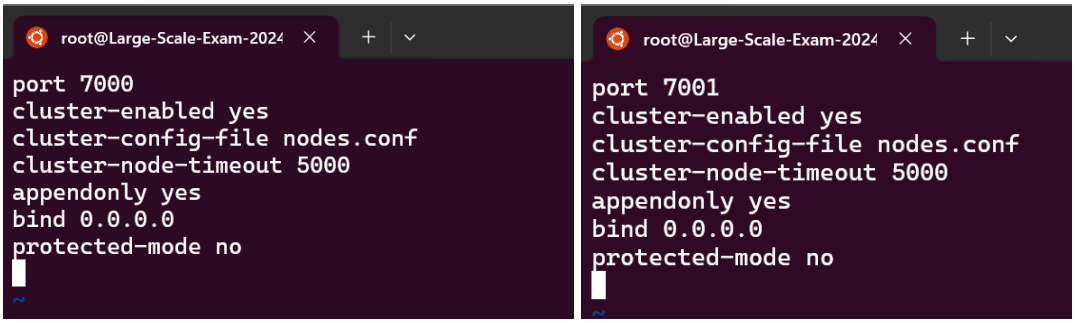
7.3.1. Redis

About replicas of key-value the database check the section Clustering on Redis, where it is explained the cluster architecture adopted.



It is important to note that Redis Cluster **does not guarantee strong consistency**. In practical terms this means that under certain conditions it is possible that Redis Cluster will lose writes that were acknowledged by the system to the client. This is because it uses **asynchronous replication**.

Here, our Redis cluster configuration for masters and slaves:



In this way we made, for each virtual machine, ports 7000 and 7001 available.

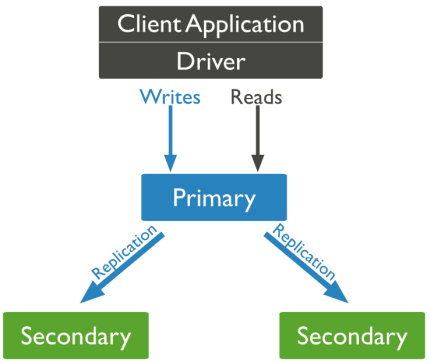
```
redis-cli --cluster-replicas 1 --cluster create 10.1.1.14:7000
10.1.1.15:7000 10.1.1.16:7000 10.1.1.14:7001 10.1.1.15:7001
10.1.1.16:7001
```

Then running the above command we built the replication mechanism on the cluster to deploy the cluster architecture that ensures eventual consistency among replicas.

7.3.2. MongoDB

Thanks to the cluster with three different machines provided by the University of Pisa, the most suitable choice was to share the Document database among three different machines in which one is the primary replica, acting as a server taking input requests, and two secondaries which are the copies of the primary.
Mind that decisions were made considering **High Availability** as one of the main non-functional requirements offered by the application, so considering following advantages:

- **Eventual Consistency**: a write operation is received by the first replica only, then eventually on the replicas so the system can keep work even if latest updates are not yet received by all replicas.
- **Protection against partitioning**: since a priority is given to both replicas, if a crash on the Primary node occurs transferred to one of the two secondary



Note that High Availability drawback cannot be avoided: since the replicas will be only eventually updated, it is not guaranteed that users will always be in front of the most recent data. Fortunately, it is not a critical issue considering the application domain, also considering the highest number of apartments available without updates.

Attention should be put on *Write Concern*, a specific parameter for write requests handling, which works as it follows:

- "W" = 0: no acknowledge mechanism adopted so the system tries to write on Primary but will never know if the operation was successful or not
- "W" >= 1 an acknowledgement is required from both Primary and as many replicas as needed by application requirements.
  Considering specific application use case, options available are:
  - "W" = 1: acknowledge required after a successful request only over the Primary
  - "W" = 2: acknowledge required after a successful request over both Primary and one of the two replicas
  - "W" = 3: acknowledge required after a successful request over both Primary and both of the two replicas

Considering that in this case W=3 set means the highest level of strict consistency requirement, which is not requested by the application domain, and considering only W=1 and W=2 as suitable solutions, W=2 was adopted to ensure consistency.

7.4. Sharding proposal

Our sharding proposal for this application is designed to work in conjunction with data replication, in order to achieve (or trying to achieve) optimal performance and scalability. The main purpose of sharding is to distribute the data across multiple servers, in order to handle the increased load and provide better performance. This is already effectively done for Redis, because it's required for the clustered implementation! Check the related paragraph. In order to implement sharding for the other DBs, we have considered two different sharding keys, one for each collection in our document database:

- For the *Apartments collection*, many options are available: since the *_id* field is unique for each document it can be eligible as a target candidate, but we can also use a geographical approach. Theoretically, one possible idea can be to divide the apartments by using as a sharding key the "*City*" field in which they're located, as this won't change through time and will divide them in as many slots as the number of cities. This is not truly suitable if the cities get really big or if some of them have many more apartments than others, since the load won't be equally shared between the machines. The alternative proposed is to divide the slots by utilizing *both the city and the square that contains it*, as we've seen in the heatmap aggregation query. This way, apartments that are in the same city and in the same logical square area are guaranteed to be stored on the same machine. This is doable since we have the position for each apartment in latitude and longitude form, so with some simple preliminary logic there won't be any problems.
- For the *Users* collection, we have a check upon user sign-up that guarantees its authenticity. So, we could implement this in a way that will produce a similar result to the Redis cluster.

In order to **balance the requests,** we would choose to use the *hashing method* as the partitioning technique for our sharding implementation. This works exactly as the Redis one, by using an hash function to determine the location of a document in the sharded cluster, resulting in evenly distributed data among the shards. This will allow for a more efficient load balancing among servers and will ensure that the system can handle large amounts of data and traffic. The **benefits** of a sharded implementation are multiple, since it allows handling *a large amount of data and traffic*, also providing better performance and scalability. It should be noted that sharding also comes with some **drawbacks**; it can make certain queries or updates more complicated and it increases system's complexity. Additionally, it requires additional resources for managing the shard servers and mongos router.

8. CRUD and Analytics on MongoDB

8.1. Create

8.1.1. User

```
db.users.insertOne({
email:"email@email.com",
first_name:"first_name",
last_name:"last_name",
password:"password",
})
```

8.1.2. Apartment into apartments collection:

```
db.apartments.insertOne({
house_name:"house_name",
host_name:"host_name",
last_name:"last_name",
email:"email@email.com",
accommodates:1,
bathrooms:1,
price:100,
city:"City",
description:"description",
picture_url:["picture_url1","picture_url2",
position:[23.11, 42.44]
})
```

8.1.3. Apartment into users collection

```
db.apartments.insertOne({
house_name:"house_name",
host_name:"host_name",
last_name:"last_name",
email:"email@email.com",
accommodates:1,
bathrooms:1,
price:100,
city:"City",
description:"description",
picture_url:["picture_url1","picture_url2"],
position:[23.11, 42.44]
})
```

8.2. Read

8.2.1. User by email

```
db.users.findOne(
{email:"email@email.com"})
```

8.2.2. Apartment by objectId

```
db.apartments.findOne(
{ _id:ObjectId( '65bd18bcab992650b58960c5' ) } )
```

8.3. Update

8.3.1. Update User password or study_field

```
db.users.updateOne(
  { email: "email@email.com" },
  { $set: { password: "newPassword" } }
)
```

8.3.2. Update User study_field

```
db.users.updateOne(
  { email: "email@email.com" },
  { $set: { study_field: "newStudyField" } }
)
```

```
db.users.updateOne(
  { email: "email@email.com" },
{ $unset: { study_field: "" } }
)
```

Application doesn't allow users to change Name, Surname or Email field.

8.3.3. Update Apartment fields into apartments collection

```
db.apartments.updateOne(
  { email: "email@email.com" },
  { $set: { "accommodates": 3 } }
)
```

8.3.4. Update Apartment picture_url field into users collection

```
db.users.updateOne(
  { email: "email@email.com" },
  { $set: { picture_url: "picture_url4" } }
)
```

8.4. Delete

8.4.1. Remove User from users collection

```
db.users.deleteOne({ email:"email@email.com" })
```

8.4.2. Remove Apartment from apartments collection

```
db.apartments.deleteOne(
{ _id:ObjectId('65bd18bcab992650b58960c5') }
)
```

8.4.3. Remove Apartment from users collection

```
db.users.updateOne(
  { email: "email@email.com" },
  { $push: {object_id:ObjectId('65bd18bcab992650b58960c5')}}
)
```

8.5. Analytics on MongoDB

8.5.1. Average lower and higher price for a given filter

Including the possibility to specify an input filter over number of accommodates, number of bathrooms or price range, the query returns the city name, related average price and number of apartments found of the both lowest and highest average price for apartments

MongoDB:

```
db.apartments.aggregate([
  {
    $match:
      {
        accommodates: {
          $gte: max_input_accommodates,
        },
        bathrooms: {
          $gte: max_input_bathrooms,
        },
        price: {
          $gte: min_input_price,
          $lte: max_input_price,
        },
      },
  },
  {
    $group: {
      _id: "$city",
      averagePrice: {
        $avg: "$price",
      },
      count: {
        $sum: 1,
      },
    },
  },
  {
    $sort:
      {
        averagePrice: 1,
      },
  },
  {
```

```
    $group:
      {
        _id: 0,
        lowestName: {
          $first: "$_id",
        },
        lowestCount: {
          $first: "$count",
        },
        lowestAveragePrice: {
          $first: "$averagePrice",
        },
        highestName: {
          $last: "$_id",
        },
        highestCount: {
          $last: "$count",
        },
        highestAveragePrice: {
          $last: "$averagePrice",
        },
      },
  },
  {
    $project:
      {
        _id: 0,
        lower: {
          name: "$lowestName",
          count: "$lowestCount",
          price: "$lowestAveragePrice",
        },
        higher: {
          name: "$highestName",
          count: "$highestCount",
          price: "$highestAveragePrice",
        },
      },
  },
])
```

Java function:

```
public String getPriceAnalytics(Integer accommodates, Integer bathrooms, Integer priceMin,
Integer priceMax) {
        MongoClient client = MongoClients.create("mongodb://localhost:27017");
        MongoDatabase database = client.getDatabase("ErasmusNest");
        MongoCollection<Document> collection = database.getCollection("apartments");

        List<Bson> matchFilters = new ArrayList<>();

        if (accommodates != 0) {
            matchFilters.add(Filters.gte("accommodates", accommodates));
        }
        if (bathrooms != 0) {
            matchFilters.add(Filters.gte("bathrooms", bathrooms));
        }
        if (priceMin != 0) {
            matchFilters.add(Filters.gte("price", priceMin));
        }
        if (priceMax != 0) {
            matchFilters.add(Filters.lte("price", priceMax));
        }
        Bson matchStage = matchFilters.isEmpty() ? match(new Document()) :
                                                   match(Filters.and(matchFilters));

        AggregateIterable<Document> result = collection.aggregate(Arrays.asList(
                matchStage,
                new Document("$group",
                        new Document("_id", "$city")
                                .append("averagePrice",
                                        new Document("$avg", "$price"))
                                .append("count",
                                        new Document("$sum", 1L))),
                new Document("$sort",
                        new Document("averagePrice", 1L)),
                new Document("$group",
                        new Document("_id", 0L)
                                .append("lowestName",
                                        new Document("$first", "$_id"))
                                .append("lowestCount",
                                        new Document("$first", "$count"))
                                .append("lowestAveragePrice",
                                        new Document("$first", "$averagePrice"))
                                .append("highestName",
                                        new Document("$last", "$_id"))
                                .append("highestCount",
                                        new Document("$last", "$count"))
                                .append("highestAveragePrice",
                                        new Document("$last", "$averagePrice"))),
                new Document("$project",
```

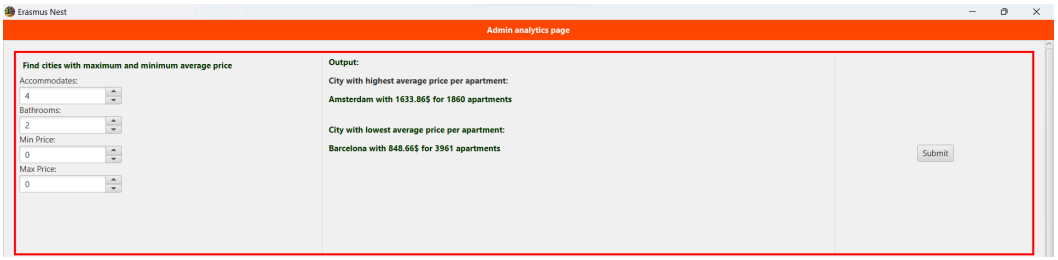
```
                        new Document("_id", 0L)
                                .append("lower",
                                        new Document("name", "$lowestName")
                                                .append("count", "$lowestCount")
                                                .append("price",
"$lowestAveragePrice"))
                                .append("higher",
                                        new Document("name", "$highestName")
                                                .append("count", "$highestCount")
                                                .append("price", "$highestAveragePrice"))));
        System.out.println("Price analytics result: ");
        StringBuilder resultString = new StringBuilder();
        for (Document doc : result) {
            resultString.append(doc.toJson()).append("\n");
        }
        String res = !resultString.toString().isEmpty() ? resultString.toString() : "No result
found";
        client.close();
        return res;
    }
```

This analytics can be used with or without filters specification: since in the first case a first $match stage is needed, in the second case this stage is avoided considering no filter by number of accommodates, bathrooms or minimum/maximum price.

Stages description:

1. **$match**: if present, act as a filter scanning all apartments collections to pass only the documents that match the specified condition(s) to the next pipeline stage
2. **$group**: groups documents using city attribute as group filed, computing average price and counting total number of apartments founded for each city
3. **$sort**: sorts document using ascending order
4. **$group**: creates both name, average price and number of apartments related fields for both most expensive and cheapest
5. **$project**: constructs embedded documents for both cities retrieved

Analytic result view:



8.5.2. Average price with a given distance as input

This query operates on all cities in the database and returns the average apartment price given an input distance from the cities' centers, calculated in the first stage as the average of latitudes and longitudes for each cities' apartments

- Introduction

The query might seem complicated at first, but this is due to the fact that we couldn't use MongoDB *geospatial queries*; these may seem like optimal for a query like this, but in reality (as in version 7.0.5) operators like $GeoNear cannot be used unless they're exactly the first aggregation stage, as reported by the database documentation. GeoNear in particular seems very good on paper since it can calculate the distance between points in an easy way, and can also select only the ones *at-max* distant "X" (with "X" given as an input) from a center point.

Behavior

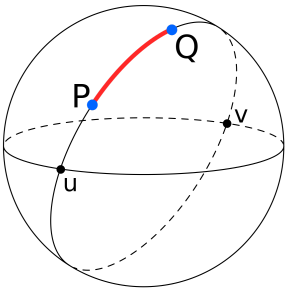
When using `$geoNear`, consider that:

- You can only use `$geoNear` as the first stage of a pipeline.

Because of this, it cannot be used, since it requires:

- To be the first aggregation stage
- The central point and distance "X" as inputs

but the first aggregation stage of the query needs to be the calculation of the center for each city. This led to the introduction of the strategy shown above to calculate the distance between points; as we know Earth *is a geoid*, but can be approximated to a sphere, so the usage of spherical coordinates was necessary for this purpose.



We utilized the Haversine formulas, suited for this objective. To do this, we, translated them as a MongoDB query, by knowing that :

$$\operatorname{hav}(\theta) = \operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)$$

where

- φ_1, φ_2 are the latitude of point 1 and latitude of point 2,
- λ_1, λ_2 are the longitudes of point 1 and point 2 respectively.

With this in mind, the Haversine formulas can be re-written as follows (with "*d*" as the distance between two points), and in MongoDb we just developed them by utilizing various stages:

$$
\begin{aligned}
d &= 2r \arcsin\left(\sqrt{\operatorname{hav}(\varphi_2 - \varphi_1) + (1 - \operatorname{hav}(\varphi_1 - \varphi_2) - \operatorname{hav}(\varphi_1 + \varphi_2)) \cdot \operatorname{hav}(\lambda_2 - \lambda_1)}\right) \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \left(1 - \sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) - \sin^2\left(\frac{\varphi_2 + \varphi_1}{2}\right)\right) \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) \cdot \cos^2\left(\frac{\lambda_2 - \lambda_1}{2}\right) + \cos^2\left(\frac{\varphi_2 + \varphi_1}{2}\right) \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right).
\end{aligned}
$$

Now, let's see the query.

MongoDB:

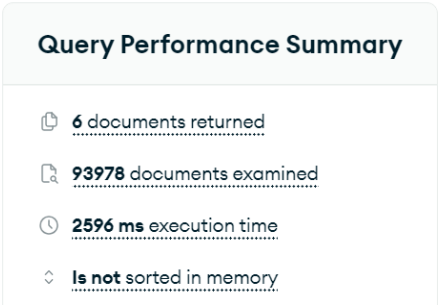
```
db.apartments.aggregate([
  {
    $group: {
      _id: "$city",
      avgLat: {
        $avg: {
          $arrayElemAt: ["$position", 0],
        },
      },
      avgLon: {
        $avg: {
          $arrayElemAt: ["$position", 1],
        },
      },
      apartments: {
        $push: "$$ROOT",
      },
    },
  },
  {
    $unwind: "$apartments",
  },
  {
    $set: {
      avgLatRad: {
        $degreesToRadians: "$avgLat",
```

```
    },
    avgLonRad: {
      $degreesToRadians: "$avgLon",
    },
    latRad: {
      $degreesToRadians: {
        $arrayElemAt: [
          "$apartments.position",
          0,
        ],
      },
    },
    lonRad: {
      $degreesToRadians: {
        $arrayElemAt: [
          "$apartments.position",
          1,
        ],
      },
    },
  },
},
{
  $addFields: {
    dLatRadians: {
      $subtract: ["$latRad", "$avgLatRad"],
    },
    dLonRadians: {
      $subtract: ["$lonRad", "$avgLonRad"],
    },
    r: 6372.8,
  },
},
{
  $addFields: {
    a: {
      $multiply: [
        {
          $sin: {
            $divide: ["$dLatRadians", 2],
          },
        },
        {
          $sin: {
            $divide: ["$dLatRadians", 2],
          },
```

```
          },
        ],
      },
      a2: {
        $multiply: [
          {
            $cos: "$latRad",
          },
          {
            $cos: "$avgLatRad",
          },
          {
            $sin: {
              $divide: ["$dLonRadians", 2],
            },
          },
          {
            $sin: {
              $divide: ["$dLonRadians", 2],
            },
          },
        ],
      },
    },
  },
  {
    $addFields: {
      a_out: {
        $add: ["$a", "$a2"],
      },
    },
  },
  {
    $addFields: {
      c: {
        $multiply: [
          2,
          {
            $asin: {
              $sqrt: {
                $max: [0, "$a_out"],
              },
            },
          },
        ],
      },
```

```
    },
  },
  {
    $addFields: {
      distance: {
        $multiply: ["$r", "$c"],
      },
    },
  },
  {
    $project: {
      distance: 1,
      "apartments.price": 1,
      "apartments.position": 1,
      "apartments.name": 1,
      "apartments.city": 1,
    },
  },
  {
    $match: {
      distance: {
        $lte: input_distance,
      },
    },
  },
  {
    $group: {
      _id: "$apartments.city",
      avgPrice: {
        $avg: "$apartments.price",
      },
    },
  },
  {
    $project: {
      _id: 1,
      avgPrice: {
        $round: ["$avgPrice", 2],
      },
    },
  }
]);
```

This query might seem the heaviest one we have available for the project; because of this, we thought it necessitated at least some insight on its execution times; in this, Compass came in handy, by providing an insightful visualization of the “explain” MongoDB command. As reported by the screenshot below, the query times are in the order of ~2 seconds, that can be considered acceptable if we observe that it will be runned by the system administrators, so not frequently.



Java function:

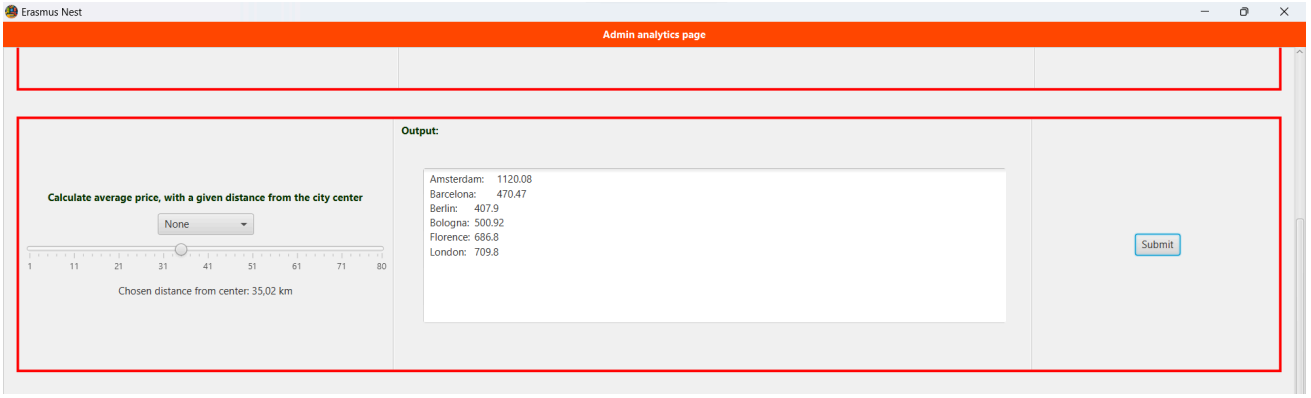
```
public List<Map<String, Object>> averagePriceNearCityCenterForEachCity(int
distance) {
        try (MongoClient mongoClient = MongoClients.create("mongodb://" +
super.getHost() + ":" + super.getPort())) {
            MongoDatabase database = mongoClient.getDatabase("ErasmusNest");
            MongoCollection<Document> apartmentsCollection =
database.getCollection("apartments");

            AggregateIterable<Document> result =
apartmentsCollection.aggregate(Arrays.asList(new Document("$group",
                            new Document("_id", "$city")
                                    .append("avgLat",
                                            new Document("$avg",
                                                    new
Document("$arrayElemAt", Arrays.asList("$position", 0L))))
                                    .append("avgLon",
                                            new Document("$avg",
                                                    new
Document("$arrayElemAt", Arrays.asList("$position", 1L))))
                                    .append("apartments",
                                            new Document("$push", "$$ROOT"))),
                    new Document("$unwind", "$apartments"),
                    new Document("$set",
                            new Document("avgLatRad",
                                    new Document("$degreesToRadians",
"$avgLat"))
```

```
                                    .append("avgLonRad",
                                            new Document("$degreesToRadians",
"$avgLon"))
                                    .append("latRad",
                                            new Document("$degreesToRadians",
                                                    new
Document("$arrayElemAt", Arrays.asList("$apartments.position", 0L))))
                                    .append("lonRad",
                                            new Document("$degreesToRadians",
                                                    new
Document("$arrayElemAt", Arrays.asList("$apartments.position", 1L)))))),
                    new Document("$addFields",
                            new Document("dLatRadians",
                                    new Document("$subtract",
Arrays.asList("$latRad", "$avgLatRad")))
                                    .append("dLonRadians",
                                            new Document("$subtract",
Arrays.asList("$lonRad", "$avgLonRad")))
                                    .append("r", 6372.8d)),
                    new Document("$addFields",
                            new Document("a",
                                    new Document("$multiply", Arrays.asList(new
Document("$sin",
                                                    new Document("$divide",
Arrays.asList("$dLatRadians", 2L))),
                                            new Document("$sin",
                                                    new Document("$divide",
Arrays.asList("$dLatRadians", 2L))))))
                                    .append("a2",
                                            new Document("$multiply",
Arrays.asList(new Document("$cos", "$latRad"),
                                                    new Document("$cos",
"$avgLatRad"),
                                                    new Document("$sin",
                                                            new
Document("$divide", Arrays.asList("$dLonRadians", 2L))),
                                                    new Document("$sin",
                                                            new
Document("$divide", Arrays.asList("$dLonRadians", 2L)))))))),
                    new Document("$addFields",
                            new Document("a_out",
                                    new Document("$add", Arrays.asList("$a",
"$a2")))),
                    new Document("$addFields",
                            new Document("c",
                                    new Document("$multiply", Arrays.asList(2L,
```

```
                                        new Document("$asin",
                                                new Document("$sqrt",
                                                        new
Document("$max", Arrays.asList(0L, "$a_out"))))))))),
                    new Document("$addFields",
                            new Document("distance",
                                    new Document("$multiply",
Arrays.asList("$r", "$c")))),
                    new Document("$project",
                            new Document("distance", 1L)
                                    .append("apartments.price", 1L)
                                    .append("apartments.position", 1L)
                                    .append("apartments.name", 1L)
                                    .append("apartments.city", 1L)),
                    new Document("$match",
                            new Document("distance",
                                    new Document("$lte", (double) distance /
1000))), // la distanza è fornita in metri ma serve in chilometri nella query
                    new Document("$group",
                            new Document("_id", "$apartments.city")
                                    .append("avgPrice",
                                            new Document("$avg",
"$apartments.price"))),
                    new Document("$project",
                            new Document("_id", 1L)
                                    .append("avgPrice",
                                            new Document("$round",
Arrays.asList("$avgPrice", 2L))))));



            List<Map<String, Object>> cityPrices = new ArrayList<>();
            for (Document doc : result) {
                String city = doc.getString("_id");
                double avgPrice = doc.getDouble("avgPrice");
                Map<String, Object> cityPriceMap = new HashMap<>();
                cityPriceMap.put("city", city);
                cityPriceMap.put("avgPrice", avgPrice);
                cityPrices.add(cityPriceMap);
            }

            cityPrices.sort((m1, m2) -> {
                String city1 = (String) m1.get("city");
                String city2 = (String) m2.get("city");
                return city1.compareTo(city2);
            });
```

```
            return cityPrices;
        }catch (Exception e){
            e.printStackTrace();
            new AlertDialogGraphicManager("MongoDB connection failed").show();
            System.out.println("Error in averagePriceNearCityCenterForEachCity:
" + e.getMessage());
            return null;
        }
 }
```

Analytic result view:



8.5.3. Heatmap analytic

Given an input city, query returns an heat map based on number of apartments per areas using red for highest number, green for lowest and yellow for average

MongoDB:

```
db.apartments.aggregate([
[
  {
    $match: {
      city: input_city,
    },
  },
  {
    $project: {
      position: 1,
    },
  },
  {
    $addFields: {
      cell: {
        $concat: [
          {
            $toString: {
              $trunc: {
                $multiply: [
                  10,
                  {
                    $arrayElemAt: [
                      "$position",
                      0,
                    ],
                  },
                ],
              },
            },
          },
          "_",
          {
            $toString: {
              $trunc: {
                $multiply: [
                  10,
                  {
                    $arrayElemAt: [
```

```
                    "$position",
                    1,
                  ],
                },
              ],
            },
          },
        },
      ],
    },
  },
  {
    $group: {
      _id: "$cell",
      count: {
        $sum: 1,
      },
    },
  },
]

])
```

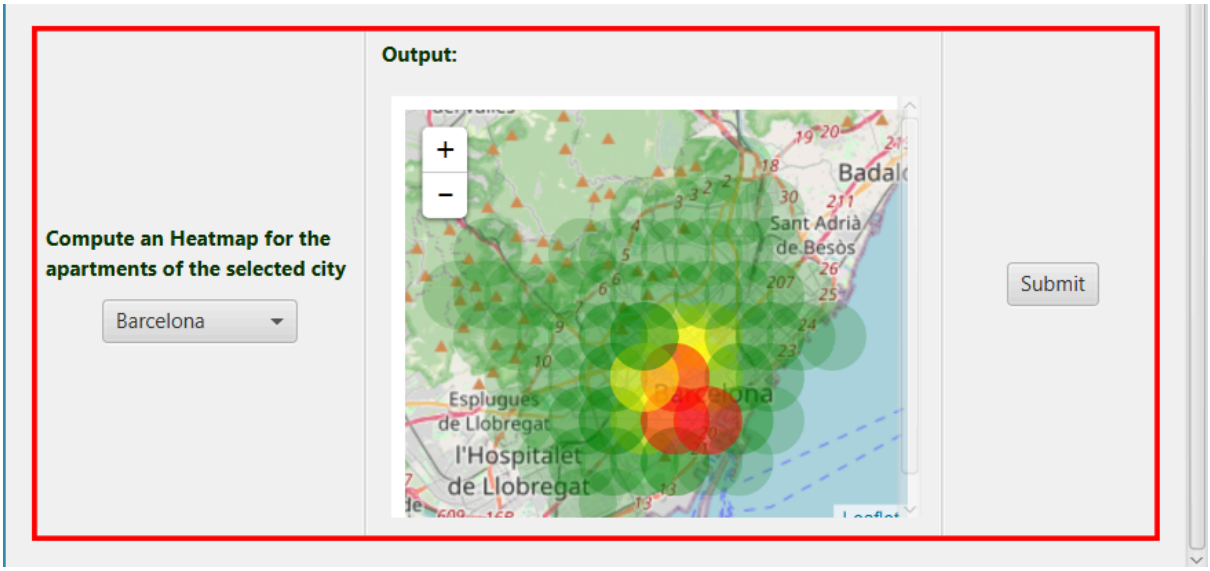
Stages description:

1. **$match**: first, we match on the city given in input by the client
2. **$projection**: in order to lighten the documents, only the “position” array is left for each apartment’ document
3. **$addFields**: “cells” are computed in this stage, multiplying by ten latitude and longitude, then condensing them in a string separated by “_”. This is done to output at the end only a singular string instead of an array for each computed square. We have all we need to proceed with a $group stage
4. **$group**: now we just group on the “cell”, and sum *+1* for each document that we find. This way we effectively find a heatmap representation, since the higher the count the higher the red color will be on the final map.

Java function:

```
public HashMap<Point2D, Integer> getHeatmap(String city) {
        long multiplier = 80L;
        try (MongoClient mongoClient = MongoClients.create("mongodb://"
+ super.getHost() + ":" + super.getPort())) {
            MongoDatabase database =
mongoClient.getDatabase("ErasmusNest");
            MongoCollection<Document> apartmentsCollection =
database.getCollection("apartments");
            AggregateIterable<Document> result =
apartmentsCollection.aggregate(Arrays.asList(new Document("$match",
                            new Document("city", city)),
                    new Document("$project",
                            new Document("position", 1L)),
                    new Document("$addFields",
                            new Document("cell",
                                    new Document("$concat",
Arrays.asList(new Document("$toString",

new Document("$trunc",

new Document("$multiply", Arrays.asList(multiplier,

new Document("$arrayElemAt", Arrays.asList("$position", 0L)))))), "_",
                                            new Document("$toString",

new Document("$trunc",

new Document("$multiply", Arrays.asList(multiplier,

new Document("$arrayElemAt", Arrays.asList("$position", 1L))))))))))),
                    new Document("$group",
                            new Document("_id", "$cell")
                                    .append("count",
                                            new Document("$sum",
1L)))));


            HashMap<Point2D, Integer> heatmap = new HashMap<>();
            for (Document doc : result) {
                String cell = doc.getString("_id");
                String[] coordinates = cell.split("_");
                double lat = Double.parseDouble(coordinates[0]);
                double lon = Double.parseDouble(coordinates[1]);
                Point2D point = new Point2D(lat/multiplier,
```

```
lon/multiplier);
                Integer count = doc.getLong("count").intValue();
                heatmap.put(point, count);
            }
            return heatmap;
        }catch (Exception e){
            e.printStackTrace();
            new AlertDialogGraphicManager("MongoDB connection
failed").show();
            System.out.println("Error in getHeatmap: " +
e.getMessage());
        }
        return null;
    }
```



9. CRUD and Analytics on Neo4J

9.1. Create

9.1.1. Create Apartment

```
MERGE(a:Apartment {
  apartment_id : $apartment_id,
  averageReviewScore : $averageReviewScore ,
  name : $name,
picture_url : $picture_url
})
MERGE (c:City {name: $city})
MERGE (a)-[:LOCATED]->(c)
```

9.1.2. Create City

```
MERGE(c:City { name:$name})
```

9.1.3. Create User

Create User code is not provided since a User instance is not create on Neo4J until he has no relationships

9.1.4. Create User—FOLLOWS—>User

```
MERGE (u:User {email: $email})
MERGE (u2:User {email: $otherEmail})
MERGE (u)-[:FOLLOWS]->(u2)
```

9.1.5. Create User—LIKE—>Apartment

```
MERGE (u:User {email: $email})
WITH u
MATCH (a:Apartment {apartmentId: $id})-[:LOCATED]->(c:City)
MERGE (u)-[:LIKES]->(a)
MERGE (u)-[:INTERESTS]->(c)
```

9.1.6. Create User—INTERESTS—>City

```
MERGE (u:User {email: $email})
WITH u
MATCH (c:City {name: $cityName})
MERGE (u)-[:INTERESTS]->(c)
```

9.1.7. Create User—REVIEW—>Apartment

```
MERGE (u:User {email: $email})
WITH u
MATCH (a:Apartment {apartmentId: $apartmentId})
MERGE (u)-[r:REVIEW]->(a)
SET r.comment = $comment, r.score = $score, r.date = $timestamp
```

9.1.8. Create Apartment—LOCATED—>City

This relationship cannot be created on demand since is established only while an apartment is created

9.2. Read

9.2.1. Read all apartments LOCATED in a given city

- By default

```
MATCH (a:Apartment)-[:LOCATED]->(c:City {name:$cityName})
OPTIONAL MATCH (a)<-[r:REVIEW]-()
RETURN a, reviewCount
SKIP $elementsToSkip LIMIT $elementsPerPage
```

- Ordered by average review score

```
MATCH (a:Apartment)-[:LOCATED]->(c:City {name:$cityName})
OPTIONAL MATCH (a)<-[r:REVIEW]-()
WITH a, COUNT(r) AS reviewCount
RETURN a, reviewCount
ORDER BY CASE WHEN a.averageReviewScore IS NULL THEN 1 ELSE 0 END,
a.averageReviewScore DESC
SKIP $elementsToSkip LIMIT $elementsPerPage
```

- Ordered by number of reviews

```
MATCH (a:Apartment)-[:LOCATED]->(c:City {name:$cityName})
OPTIONAL MATCH (a)<-[r:REVIEW]-()
WITH a, COUNT(r) AS reviewCount
RETURN a, reviewCount
ORDER BY CASE WHEN a.averageReviewScore IS NULL THEN 1 ELSE 0 END,
reviewCount DESC
SKIP $elementsToSkip
LIMIT $elementsPerPage"
```

9.2.2. Read average review score attribute for a specific apartment

```
MATCH (a:Apartment)-[:LOCATED]->(c:City {name:$cityName})
OPTIONAL MATCH (a)<-[r:REVIEW]-()
WITH a, COUNT(r) AS reviewCount
RETURN a, reviewCount
ORDER BY CASE WHEN a.averageReviewScore IS NULL THEN 1 ELSE 0 END,
reviewCount DESC
SKIP $elementsToSkip
LIMIT $elementsPerPage"
MATCH (a:Apartment {apartmentId: $apartmentId})
RETURN DISTINCT a.averageReviewScore
```

9.2.3. Read reviews for a specific apartment

- Filter 0

```
MATCH (u:User)-[r:REVIEW]->(a:Apartment {apartmentId: $apartmentId})
RETURN u.email,r
SKIP $elementsToSkip
LIMIT $elementsPerPage
```

- Ordered by score (e.g. descendent)

```
MATCH (u:User)-[r:REVIEW]->(a:Apartment {apartmentId: $apartmentId})
RETURN u.email,r
ORDER BY r.score DESC
SKIP $elementsToSkip
LIMIT $elementsPerPage
```

- Ordered by date (e.g. ascendent)

```
MATCH (u:User)-[r:REVIEW]->(a:Apartment {apartmentId: $apartmentId})
RETURN u.email,r
ORDER BY r.date ASC
SKIP $elementsToSkip
LIMIT $elementsPerPage
```

9.2.4. Read reviews made by a specific User

- All reviews for all apartments

```
MATCH (u:User {email: $email})-[r:REVIEW]->(a:Apartment)
RETURN r, a.apartmentId
SKIP $elementsToSkip
LIMIT $elementsPerPage
```

- For a specific Apartment

```
MATCH (u:User {email: $email})-[r:REVIEW]->(a:Apartment {apartmentId:
$apartmentId})
RETURN r
```

9.2.5. Read Cities

- All cities

```
MATCH (c:City) RETURN c.name
```

- Read all Cities a specific User is interested in

```
MATCH (u:User {email: $email})-[:INTERESTS]->(c:City)
RETURN c.name
```

9.2.6. Read number of User's followers

```
MATCH (u:User {email: $email})<-[f:FOLLOWS]-(u2:User)
RETURN u2.emaill1
```

9.2.7. Read if a like relationship exists

```
MATCH (u:User {email: $email})-[l:LIKES]->(a:Apartment {apartmentId:
$id}) RETURN l
```

9.2.8. Read all apartments a User likes

```
MATCH (u:User {email: $email})-[l:LIKES]->(a:Apartment)
RETURN a.apartmentId, a.name
```

9.3. Update

9.3.1. Update apartment average review score

```
MATCH ()-[r:REVIEW]->(a:Apartment {apartmentId:$apartmentId})
WITH a, AVG(r.score) AS averageScore
SET a.averageReviewScore = ROUND(averageScore * 100) / 100
```

9.3.2. Update apartment picture_url

```
MATCH (a:Apartment {apartmentId: $apartmentId})
SET a.pictureUrl = $pictureUrl
```

9.4. Delete

9.4.1. Remove apartment and all related relationships

```
MATCH (a:Apartment {apartmentId: $apartmentId})
DETACH DELETE a
```

9.4.2. Remove REVIEW relationship

```
MATCH (u:User {email: $email})-[r:REVIEW]->(a:Apartment {apartmentId:
$apartmentId})
WHERE r.comment = $comment AND r.score = $score AND r.date = $timestamp
DELETE r
```

9.4.3. Remove LIKES relationship

```
MATCH (u:User {email: $email})-[l:LIKES]->(a:Apartment {apartmentId:
$favourite})
DELETE l
```

9.4.4. Remove FOLLOWS relationship

```
MATCH (u:User {email: $email})-[f:FOLLOWS]->(u2:User {email:
$otherEmail})
DELETE f
```

9.4.5. Remove INTERESTS relationship

```
MATCH (u:User {email: $email})-[r:INTERESTS]->(c:City)
DELETE r
```

9.5. Analytics on Neo4J

9.5.1. Users suggestions

Triggered when a User_A starts following another User_B, retrieves all users who:

- Are followed by User_B
- Are not already followed by User_A
- Are interested in same Cities as User_A

Cypher:

```
MATCH
(a:User{email:$emailA})-[:INTERESTS]->(city:City)<-[:INTERESTS]-(suggested:User),
(b:User {email: $emailB})-[:FOLLOWS]->(suggested)
WHERE NOT (a)-[:FOLLOWS]->(suggested)
  AND suggested.email <> $emailA
RETURN DISTINCT suggested.email AS suggestedEmail
```

9.5.2. Apartments suggestions

Triggered when a User_A likes an Apartment_A, retrieves first three apartments which:

- Are located in same city as Apartment_A
- Have been reviewed with a positive score (>3) from a User_B
- User_A follows User_B

Cypher:

```
MATCH
(u:User{email:$email})-[:FOLLOWS]->(f:User)-[r:REVIEW]->(a:Apartment)-[:LOCATED]
->(c:City {name: $cityName})
WHERE r.score >= 3
WITH a, MAX(r.score) AS maxScore
RETURN a.apartmentId AS apartmentId,
       a.name AS name,
       a.pictureUrl AS pictureUrl,
       a.averageReviewScore AS averageReviewScore
ORDER BY maxScore DESC
LIMIT 3
```

10. CRUD on Redis

Java code of staple methods belonging to class RedisConnectionManager.java.

10.1. Create

10.1.1. Create a User

```
public boolean addUser(String email, String password) {

        boolean added = false;

        try (JedisCluster jedis = createJedisCluster()) {

            String key = "user:" + email ;

            // set the key
            Map<String, String> hash = new HashMap<>();;
            hash.put("password",password);
            jedis.hset(key, hash);

            // compute the seconds between now and trashWeeksInterval
            long seconds = getGarbageTimeInSeconds();
            // set expiration time on the key equal to the seconds
            jedis.expire(key, seconds);

            added = true;

        } catch (Exception e) {
            new AlertDialogGraphicManager(
                        "Redis connection failed").show();
        }
        return added;
    }
```

10.1.2. Create a Reservation

```
public void addReservation(User student, Reservation reservation,
                                          ArrayList<String> apartmentsIds) {

        try (JedisCluster jedis = createJedisCluster()) {

            String dateTime = LocalDateTime.now().toString();
            String subKey = "reservation:" + student.getEmail() + ":{" +
                   reservation.getApartmentId() + "}:" + reservation.getStartYear() + ":"
                   + reservation.getStartMonth() + ":" + reservation.getNumberOfMonths();

            Map<String, String> hash = new HashMap<>();;
            hash.put("timestamp", dateTime);
            hash.put("city", reservation.getCity());
            hash.put("apartmentImage", reservation.getApartmentImage());
            hash.put("state", "pending"); // pending | approved |
                                          rejected | expired | reviewed
            System.out.println("subKey: " + subKey);
            System.out.println("hash: " + hash);
            jedis.hset(subKey, hash);

            // get the first day after the whole reservation period is expired
            LocalDateTime expirationDate = LocalDateTime.of(reservation.getStartYear(),
                        reservation.getStartMonth(), 1, 0, 0)
                        .plusMonths(reservation.getNumberOfMonths());

            // add the "still alive months" and the "trash week" to the expiration date
            expirationDate = expirationDate
                    .plusMonths(reservationMonthsInterval).plusWeeks(trashWeeksInterval);

            // compute the seconds between the timestamp and the end of the reservation
            long seconds = LocalDateTime.now().until(expirationDate, ChronoUnit.SECONDS);

            // set expiration time on the key equal to the seconds
            jedis.expire(subKey, seconds);

            addReservationsToUser(student.getEmail(), apartmentsIds);
            setExpirationTimeOnUser(student.getEmail(), seconds, ExpiryOption.GT);

        } catch (Exception e) {
            System.out.println("Connection problem: " + e.getMessage());
            new AlertDialogGraphicManager("Redis connection failed").show();
        }
    }
```

10.2. Read

10.2.1. Read user's password

```
public String getPassword(String email) {

        String value = null;

        try (JedisCluster jedis = createJedisCluster()) {

            String key = "user:" + email;
            value = jedis.hget(key, "password");
            System.out.println("Password: " + value);

            return value;

        } catch (Exception e) {
            new AlertDialogGraphicManager("Redis connection failed")
                                                                    .show();
        }
        return value;
    }
```

10.2.2. Read reservations for an apartment

```
public ArrayList<Reservation> getReservationsForApartment(String houseIdToSearch){
        ArrayList<Reservation> reservations = new ArrayList<>();

        try (JedisCluster jedis = createJedisCluster()) {

            // Use the KEYS command to get all keys matching the pattern
            Set<String> keys = jedis.keys(
                                "reservation:*:{" + houseIdToSearch + "}:*:*:*");

            for (String key : keys) {
                if(!isReservationInTrashPeriod(key)) {
                    String[] keyParts = key.split(":");
                    // removing the curly braces from keyParts[2]
                    keyParts[2] = keyParts[2].substring(1, keyParts[2].length()-1);
                    reservations.add(new Reservation(keyParts[1], keyParts[2],
                                        Integer.parseInt(keyParts[3]),
                                        Integer.parseInt(keyParts[4]),
                                        Integer.parseInt(keyParts[5])));
                }
            }
        } catch (Exception e) {
            new AlertDialogGraphicManager("Redis connection failed").show();
        }
        return reservations;
    }
```

10.2.3. Read reservation's attributes

```
private ArrayList<String> getReservationAttributesValues(String subKey) {

    ArrayList<String> attributesValues = new ArrayList<>();
    try (JedisCluster jedis = createJedisCluster()) {

        Map<String, String>hash = jedis.hgetAll(subKey);
        attributesValues.add(hash.get("timestamp"));
        attributesValues.add(hash.get("city"));
        attributesValues.add(hash.get("apartmentImage"));
        attributesValues.add(hash.get("state"));

    } catch (Exception e) {
        new AlertDialogGraphicManager("Redis connection failed").show();
    }
    return attributesValues;
}
```

10.3. Update

10.3.1. Update user's password

```
    public boolean updateUserPassword(String email, String password) {

        try (JedisCluster jedis = createJedisCluster()) {

            String key = "user:" + email;
            // set the key
            jedis.hset(key, "password", password);

            // set the ttl to -1 to remove the expiration time
            jedis.persist(key);

            return true;
        } catch (Exception e) {
            new AlertDialogGraphicManager("Redis connection failed")
                                                                .show();
        }
        return false;
    }
```

10.3.2. Approve a reservation

```
public void approveReservation(Reservation reservation) {

        try (JedisCluster jedis = createJedisCluster()) {

            String subKey = getSubKey(reservation);
            Map<String, String> hash = new HashMap<>();;
            hash.put("timestamp", reservation
                                    .getTimestamp().toString());
            hash.put("city", reservation.getCity());
            hash.put("apartmentImage", reservation.getApartmentImage());
            hash.put("state", "approved"); // pending | approved
                                     | rejected | expired | reviewed
            jedis.hset(subKey, hash);

        } catch (Exception e) {
            new AlertDialogGraphicManager("Redis connection failed")
                                                                .show();
        }
    }
```

10.4. Delete

10.4.1. Delete reservation

```
public void deleteReservation(Reservation reservation,
                              ArrayList<String> apartmentsIds) {

    try (JedisCluster jedis = createJedisCluster()) {

        String subKey = getSubKey(reservation);
        jedis.del(subKey);

        if(apartmentsIds.isEmpty())
            jedis.hdel("user:" + reservation.getStudentEmail(),
                                                "reservedApartments");
        else
            jedis.hset("user:" + reservation.getStudentEmail(),
             "reservedApartments", String.join(",", apartmentsIds));

        updateExpirationTimeOnUser(reservation.getStudentEmail(),
                                                        apartmentsIds);

    } catch (Exception e) {
        new AlertDialogGraphicManager("Redis connection failed")
                                                            .show();
    }

}
```

11. Caching Strategy

To improve the performance of the ErasmusNest application we introduce a solution for caching user credentials. For this purpose we exploit the **in-memory property of Redis** that is designed to hold all the key-value pairs in memory. This approach allows access to data in a faster way during the **login phase**, avoiding possible bottlenecks due to data transfer in the document database primary node. Considering that a user can modify his password, we assume that the password update is not a frequent operation, so also the information cached will rarely need to be updated.

11.1. Entities involved

As mentioned before the caching involves passwords for a subset of users, in particular we consider users that recently logged-in with the application. According to the whole database design we store information of the **User entity** on the document database, so the caching mechanism is just related to the **credentials** because we don't need to store any other information that could reduce Redis memory space. In this regard, note that also Reservation entities are stored in the key-value database, even if they also have a time limit for their existence. Due to the possible existence of a relationship between one user and one or more reservations, users who have done at least one reservation are excluded from the short term caching cause they will be maintained in cache for the entire reservation lifetime. This behavior is part of the expiration mechanism explained below.

11.2. Eviction policy

About the eviction policy we adopted, the **volatile-ttl** was preferred to the others. Thanks to this choice we are able to provide hints to Redis about what are good candidates for expiration by using different TTL values when we create cache objects. Exactly volatile-ttl removes keys with the expire field set to true and the **shortest remaining time-to-live** value. This policy allows the introduction of the expiration mechanism explained below.

```
127.0.0.1:6379> CONFIG SET maxmemory-policy volatile-ttl
OK
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "volatile-ttl"
```

11.3. Expiration mechanism

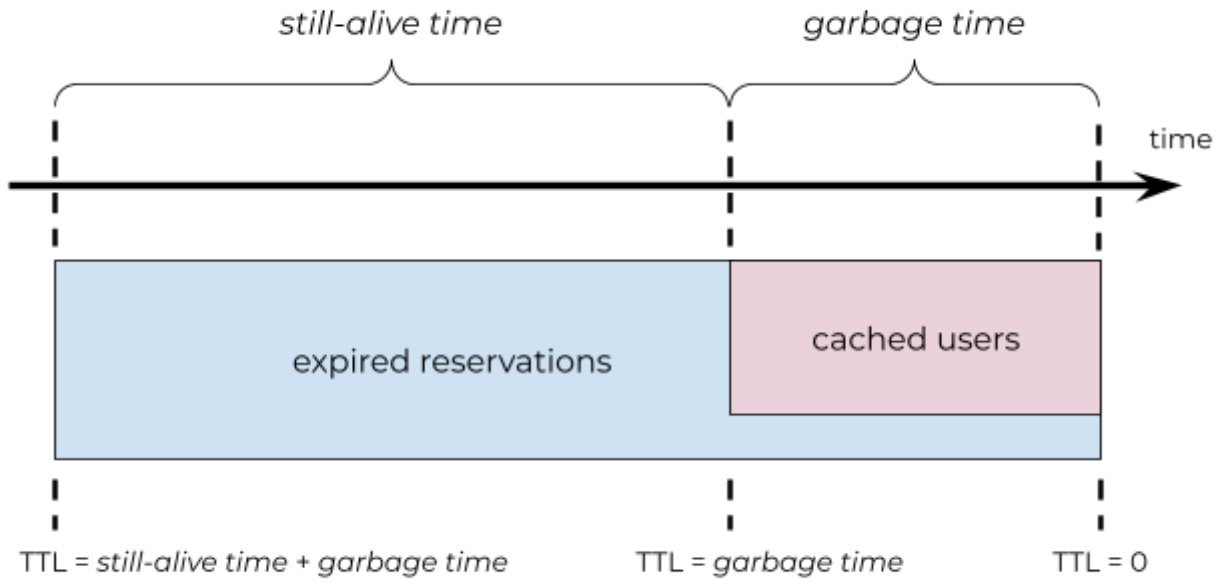
Considering that our key-value database stores both User and Reservation entities and both are involved by the volatile-ttl eviction policy, we built a solution that guarantees the expirations of only entities that can truly be removed. In order to do this we introduced a kind of "***garbage time***" in which all the cached users and only elapsed reservations will be stored. The main problem we faced is that users and reservations have different requirements in terms of time-to-live:

- users not related to reservations will be cached until the elapsion of the *garbage time*, started at the moment corresponding to the last login;
- users with at least one not expired reservation will be stored until the elapsion of the last related reservation;
- reservations will be stored until a "***still-alive time***" plus the *garbage time* period will be expired.

We assumed a *garbage time* of one week and a *still-alive time* of two months.
Note that the period a reservation spends in *still-alive time* is fundamental to allow its user to write a review for that one reservation.

So, when is the time to live of a key set or updated?

- When a reservation is rejected by the host, the TTL of the user will be updated in accordance with the elapsion of the most distant reservation for that user, if the rejected reservation was the unique reservation for that user the TTL will be set to the *garbage time*.
- When a user is loaded in cache, its TTL will be initialized with the *garbage time*.
- When a reservation is added, its TTL will be initialized with the reservation duration, plus *still-alive* time and the *garbage time*. The TTL of the related user will be updated in accordance with the expiration of the further away in time reservation for that user.



So, to give an example and wrap-up this documentation section, let's suppose that an user books a reservation for the period between 01/01/2024 and 01/06/2024; what happens is the following:

1. A reservation entry is added to redis, with TTL set to *six months* (the time of the reservation), *plus two months* (the time in which the user can write a review for his Erasmus' apartment), *plus one week* (the trash period).
2. The TTL of the user is updated with the same TTL of the reservation

This way, we can ensure that the reservation will have its corresponding user inside Redis for its duration. The things that now can happen are the following:

- When the TTL of the reservation is greater than two months plus one week, we know that the user is currently engaging his Erasmus experience
- When the TTL of the reservation is *exactly* two months plus one week, we know that the Erasmus ended for the user. From now on he can leave a review, and he has two months to do it.
- When the TTL is one week or less, the user cannot leave a review anymore, and the review enters the "trash area", in which the volatile TTL algorithm can possibly choose it as a candidate for removal. The same thing happens to the user entry, if he didn't book any other apartments in the meantime.

11.4. Credentials consistency and caching

Actually there is another case in which the TTL is modified for a user: when the modified password for that user has been successfully updated on the document database. In fact when a user updates its password this change is immediately registered on Redis then eventually will be updated on Mongo. Due to this, to not lose the password change, when a user modifies its credential the TTL corresponding to the user key is set to -1, in order to exclude it from the caching mechanism. So, when the updated password on the key-value database and the one stored in the document db will correspond, the TTL for the user credential key will be newly initialized with the *garbage time* and the user returns available for caching.

12. Performance Evaluation

12.1. Impact of caching on user experience

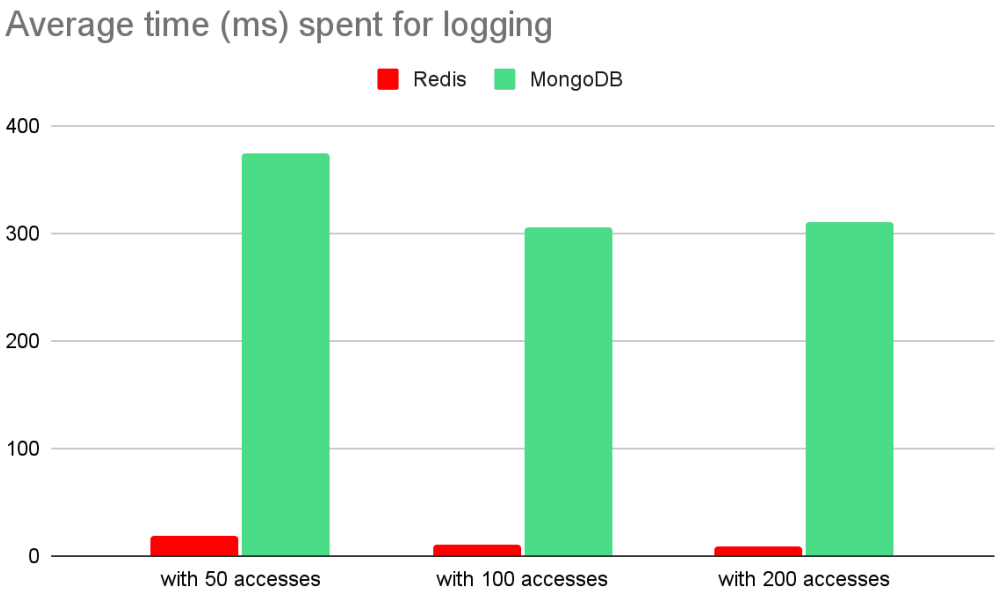
As reported in the previous chapter, we have applied a caching mechanism on user credentials with the main objective of **reducing** both **the load on the MongoDB node** and **the time needed to retrieve or modify passwords**, with the obvious consequent advantage of reducing the time spent in the login phase.

12.1.1. Reducing the load on the MongoDB node

All the information about Users and Apartments are stored in corresponding collections on the document database, so every time we need to interact with data of a User or an Apartment the MongoDB node is involved to compute these operations. We know that this type of operation could be very frequent, so we decide to exploit the caching mechanism to avoid, when it is possible, the access to the document database node during the **login phase**, considering that this phase **is common to most use cases**. Actually, when a user logs in, we first check if its credentials are in the cache yet and if so **we avoid accessing MongoDB**. Instead, if the user is not cached, the access to MongoDB is unavoidable but from the next time the user will be present in the cache.

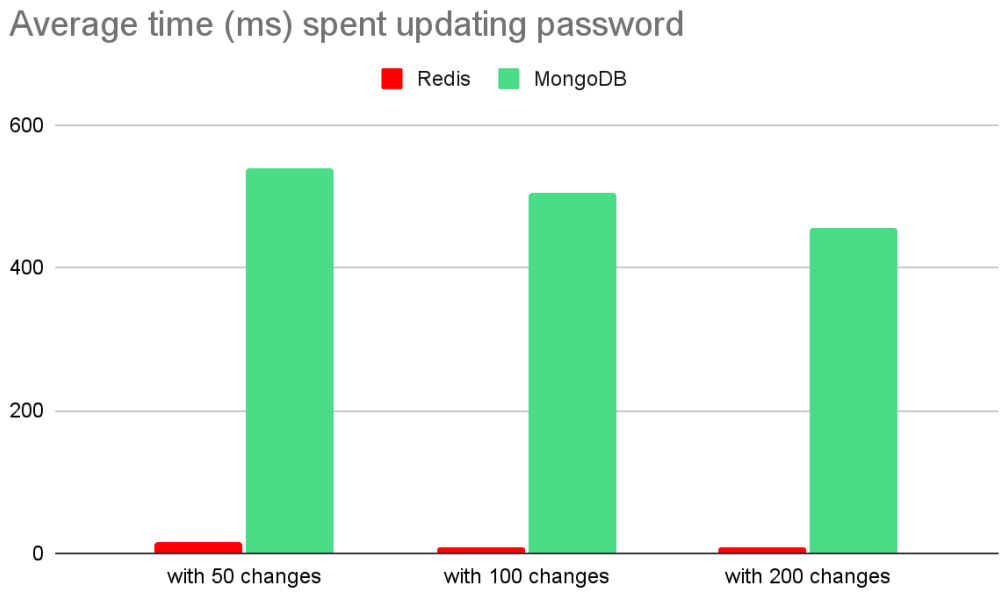
12.1.2. Reducing the time needed to retrieve or modify passwords

Exploiting the speed offered by using Redis, we are able to perform faster read operations than using MongoDB, so the password retrieval will require less time. In order to certify this, we implement a simple snippet of code to measure the logging performance both using only Redis and only MongoDB.



The previous chart was obtained by simulating the access of users stored on both Redis and MongoDB. As we can see there is a noticeable difference that justifies the caching of user credentials, using Redis **faster performances** were measured. To be honest, talking about milliseconds, also the time taken by MongoDB would have been an acceptable amount of time, but it is also necessary to consider that the same analysis must be done also for the password update. Also, we have to consider that in a fully operational phase for the application, the document database will be one of the *heaviest impacted* one of the three we have available, since it provides the most features for the final user.

Changing a password by a user is a much less frequent action than the one previously analyzed, but its speed of execution could be crucial to guarantee the access to the application. So, we adopt a strategy that executes the update immediately on Redis and eventually on MongoDB. To evaluate the effective difference about the time spent to perform this operation we repeat the same measurements done for the login.



Also here there is an evident dissimilarity in favor of Redis usage. But it is important to highlight that this time the operation will be eventually executed also on MongoDB, so the **real advantage** we introduced is in the case in which an heavy load on the document database node could not allow an immediate execution of the update: the password changed for the cached user in Redis **allow the user to access the application even if the password is not already updated on MongoDB**, we do this in the faster way possible. About Credentials consistency and caching see the related paragraph.

12.2. Queries frequency estimation

Query	Frequency	DB involved
Search Apartment by City	Very High	GraphDB
Search Apartment by apartment_id	High	MongoDB
Search Review by apartment_id	Medium	GraphDB
Search Reservation by apartment_id	Medium	Key-Value
Search User by email	Low	MongoDB
Search Reservation by email	Low	Key-Value

13. Indexing

Let's now discuss the indexes we introduced for our MongoDB and Neo4j deployments.

13.1. Indexes for MongoDB

Since many of the queries available to the user requires managing the email field of other users or even himself, we thought that an introduction of an index on it would have been quite good. Below, we report the results of an experiment we conducted by repeatedly calling the "`findUser`" method a hundred and a thousand times respectively, with and without an index on the "email" field of the users' documents:

Number of iterations	Time (without indexes)	Time (with indexes)
100	337.151 ms	10.306 ms
1000	361.904 ms	6.364 ms

As we can see the benefit is present, so we decided to keep it.

The applications also support analytics for the system administrator, and these mainly play out with the "city" field inside the documents. Because of this, an additional experiment was run with the introduction of an index on city field:

Numero di iterazioni	Time (without indexes)	Time (with indexes)
100	102.017 ms	71.422 ms
1000	97.647 ms	80.542 ms

As said, the queries that mainly utilize the *'city'* field in MongoDB are not executed frequently, as they are reserved for the system administrator. Due to this, we have decided *against* creating an index on the *'city'* field, especially since the time advantage was minimal, improving performance by only 20ms for an operation that is executed a few times per month.

Indexes were created for MongoDB by exploiting the rich Compass interface, which also allows easy index creation for the database.

13.2. Indexes for Neo4J

In the realm of Neo4J, the significance of efficient data retrieval cannot be overstated, particularly when it comes to fields that are heavily interacted with, such as the email field. This is a frequent target for client queries but also a critical component in the application's logic, especially for generating follow suggestions among users. Because of this, we conducted an experiment similar to precedents, calling a hundred and a thousand times the "`getReviewsForUser`" method.

Numero di iterazioni	Time (without indexes)	Time (with indexes)
100	678.272 ms	18.415 ms
1000	623.272 ms	12.130 ms

As shown by the test, time is quite lower using the index; because of this, we opted for its utilization in the graph database.

Another meaningful index we thought could be one for the "apartment" nodes, on their property apartmentId.

Numero di iterazioni	Time (without indexes)	Time (with indexes)
100	43.241 ms	9.427 ms
1000	40.102 ms	5.590 ms

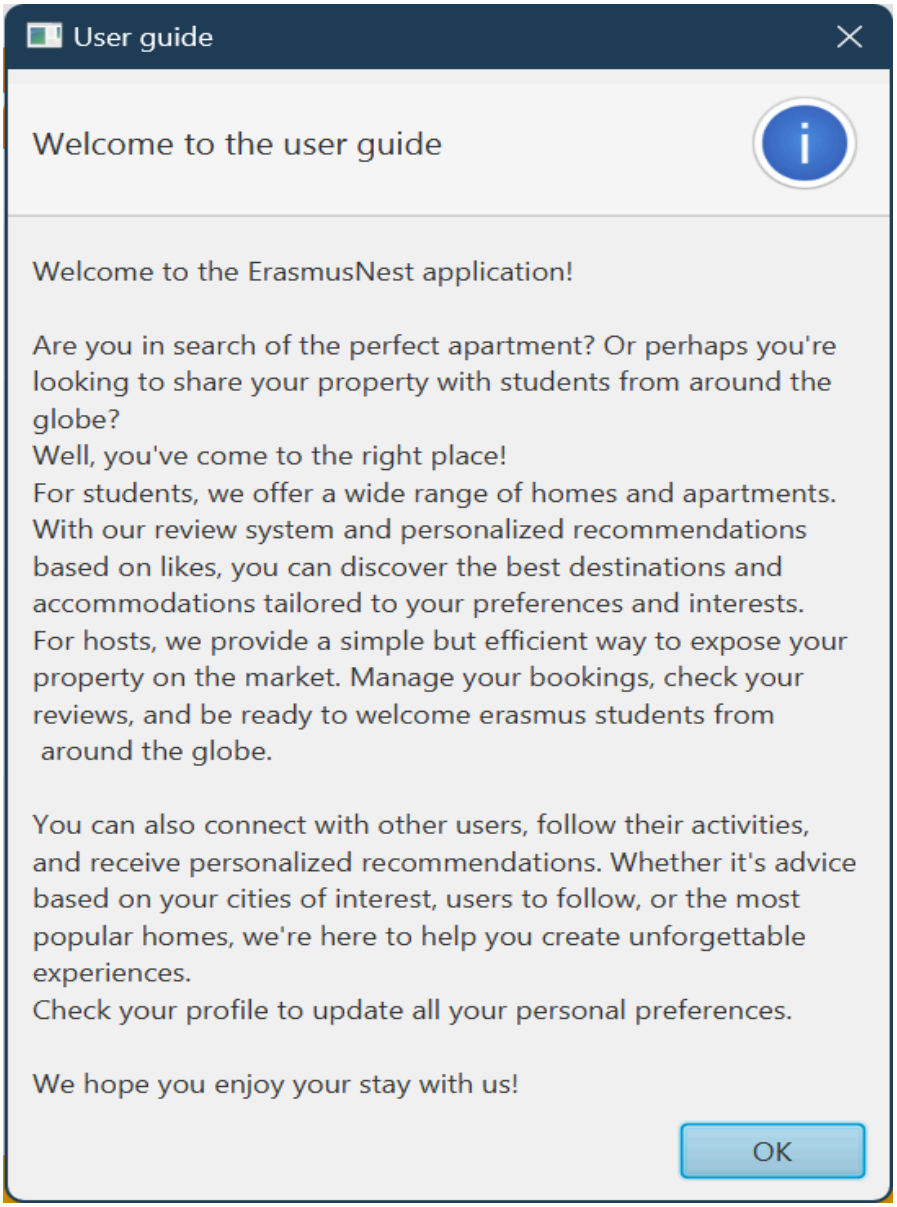
While the performance gains are less dramatic than those observed for the previous index, they are still substantial. In a production environment, where database traffic is exponentially higher, the efficiency brought by this index is quite significant to ignore. Therefore, we decided to implement it in our graph database, anticipating that it will contribute to smoother, more efficient database interactions in high-traffic scenarios.

For Neo4J indexes creation, we utilized some cypher queries, given to the database as input by utilizing the Neo4J desktop browser window.

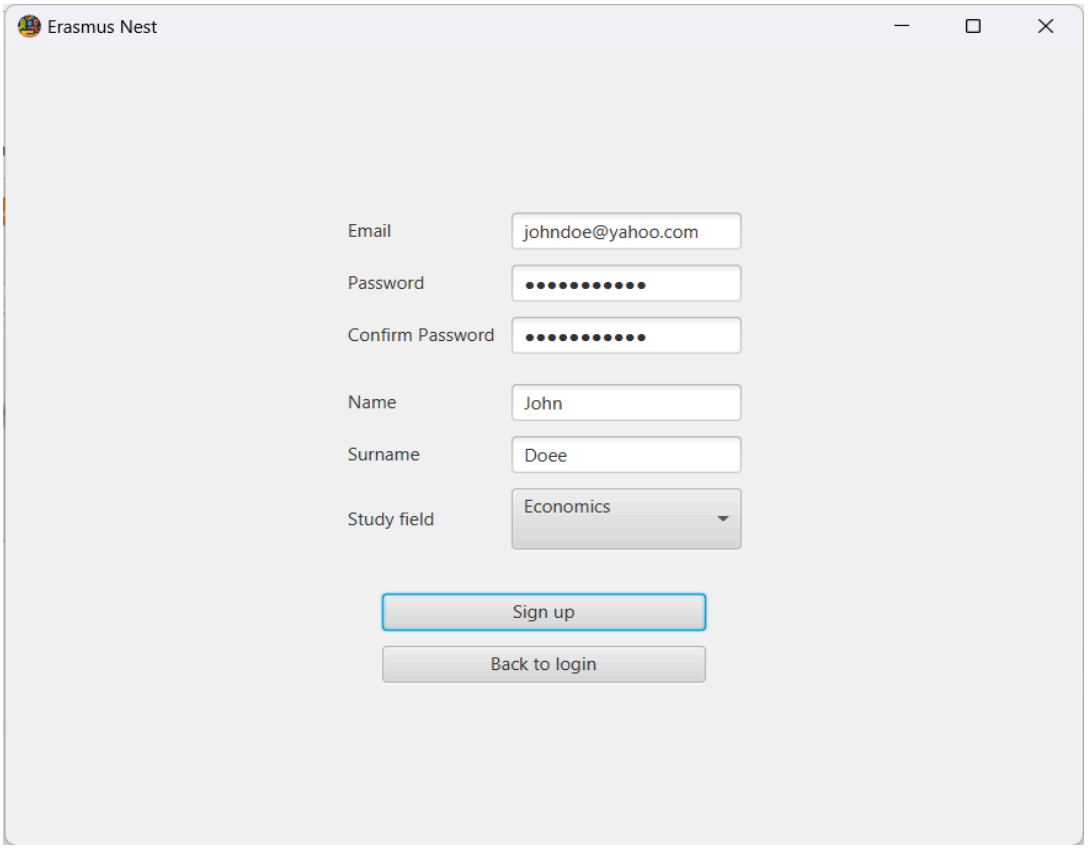
```
CREATE INDEX userEmail FOR (u:User) ON (u.email);
CREATE INDEX apartmentIdIndex FOR (a:Apartment) ON (a.apartmentId);
```

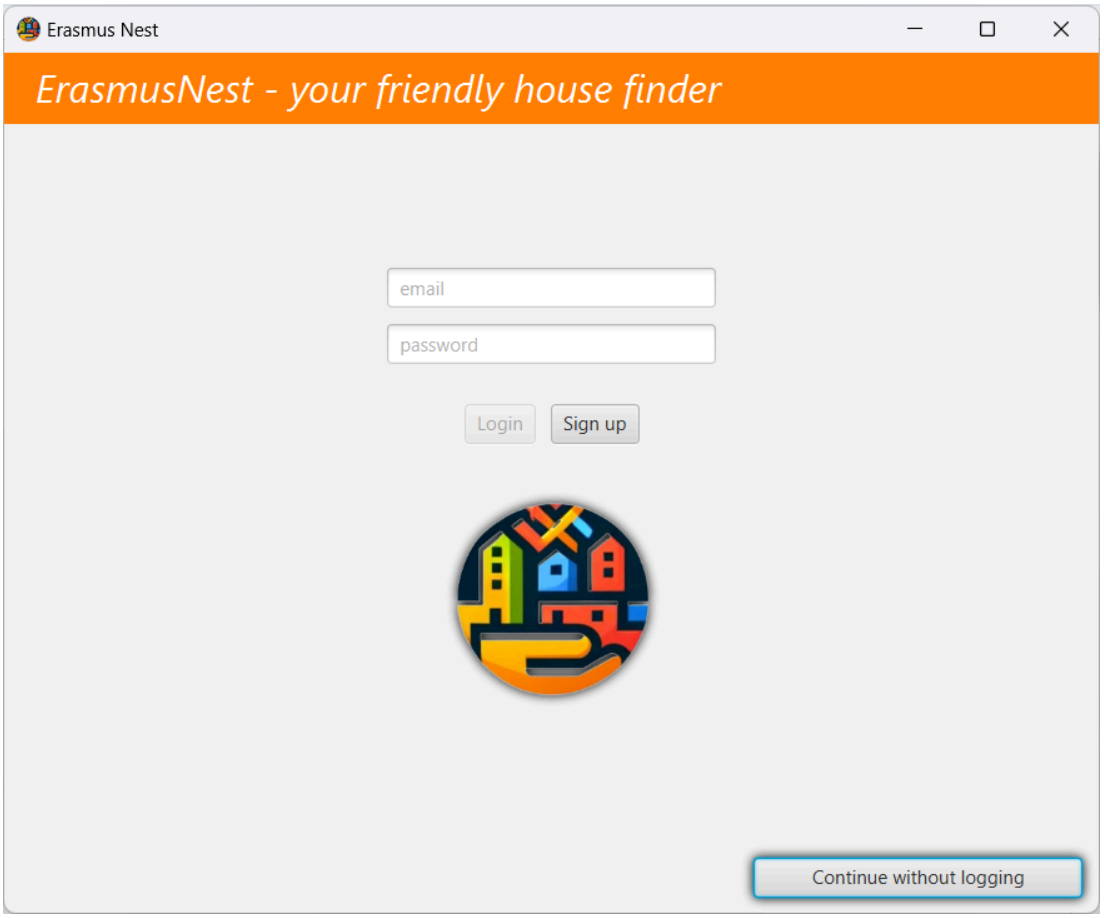
14. USER Manual and application workflow

14.1. User Guide



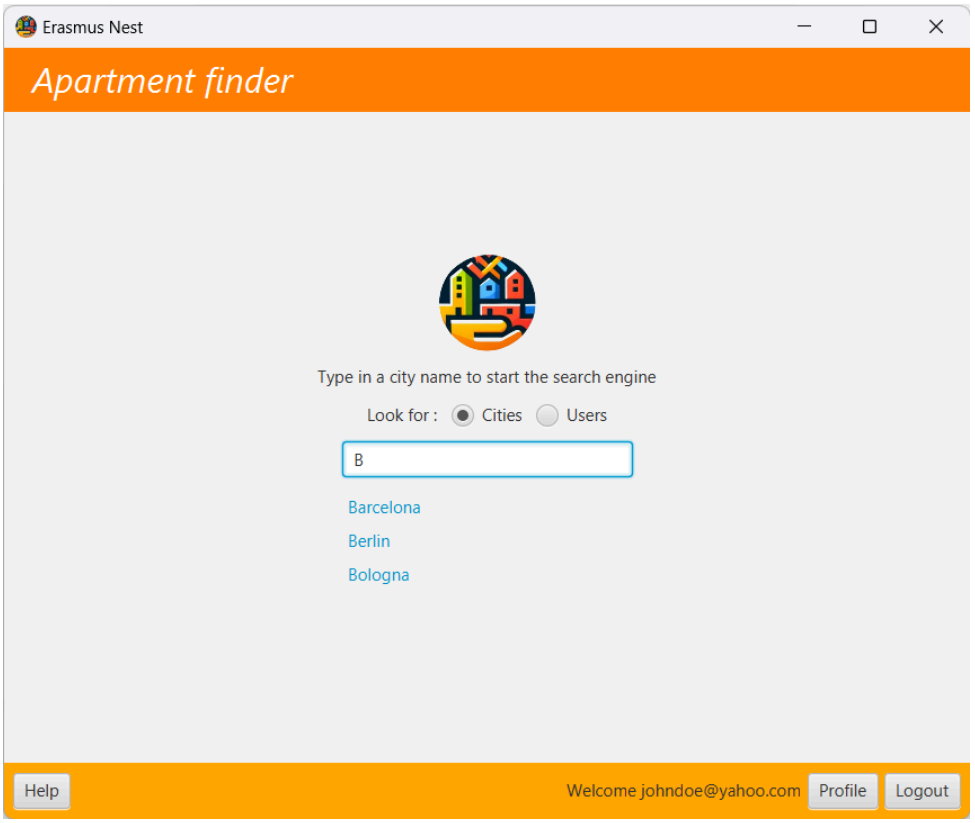
14.2. Login and Signup pages



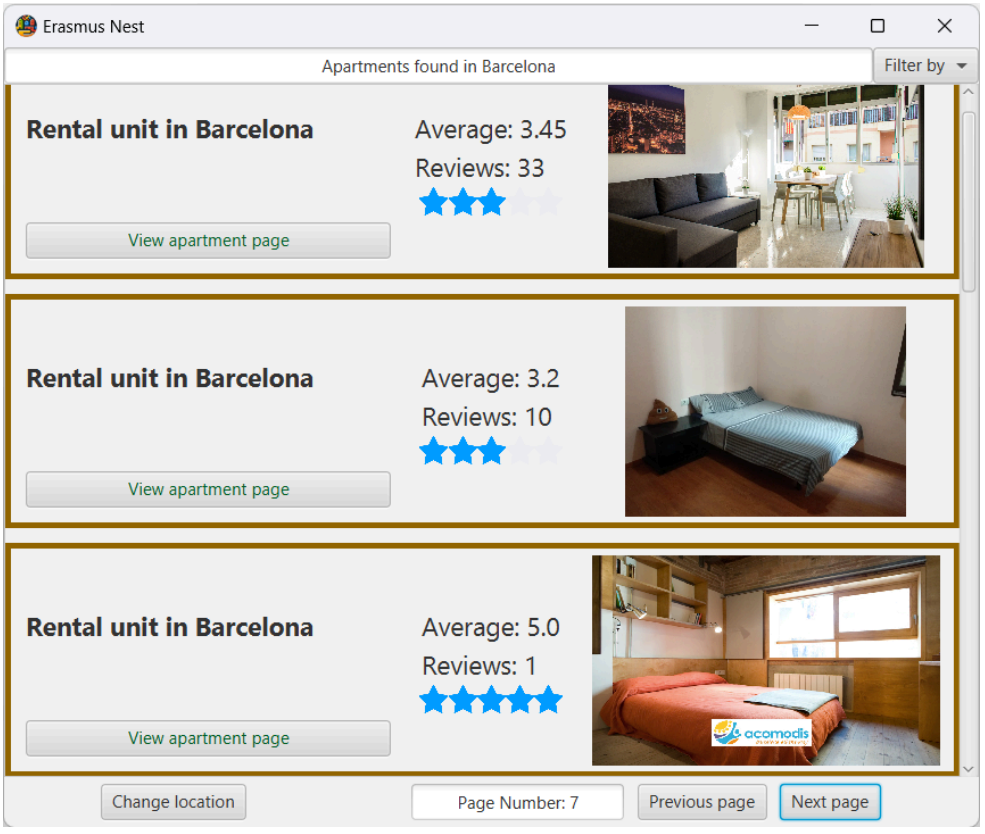


14.3. Homepage

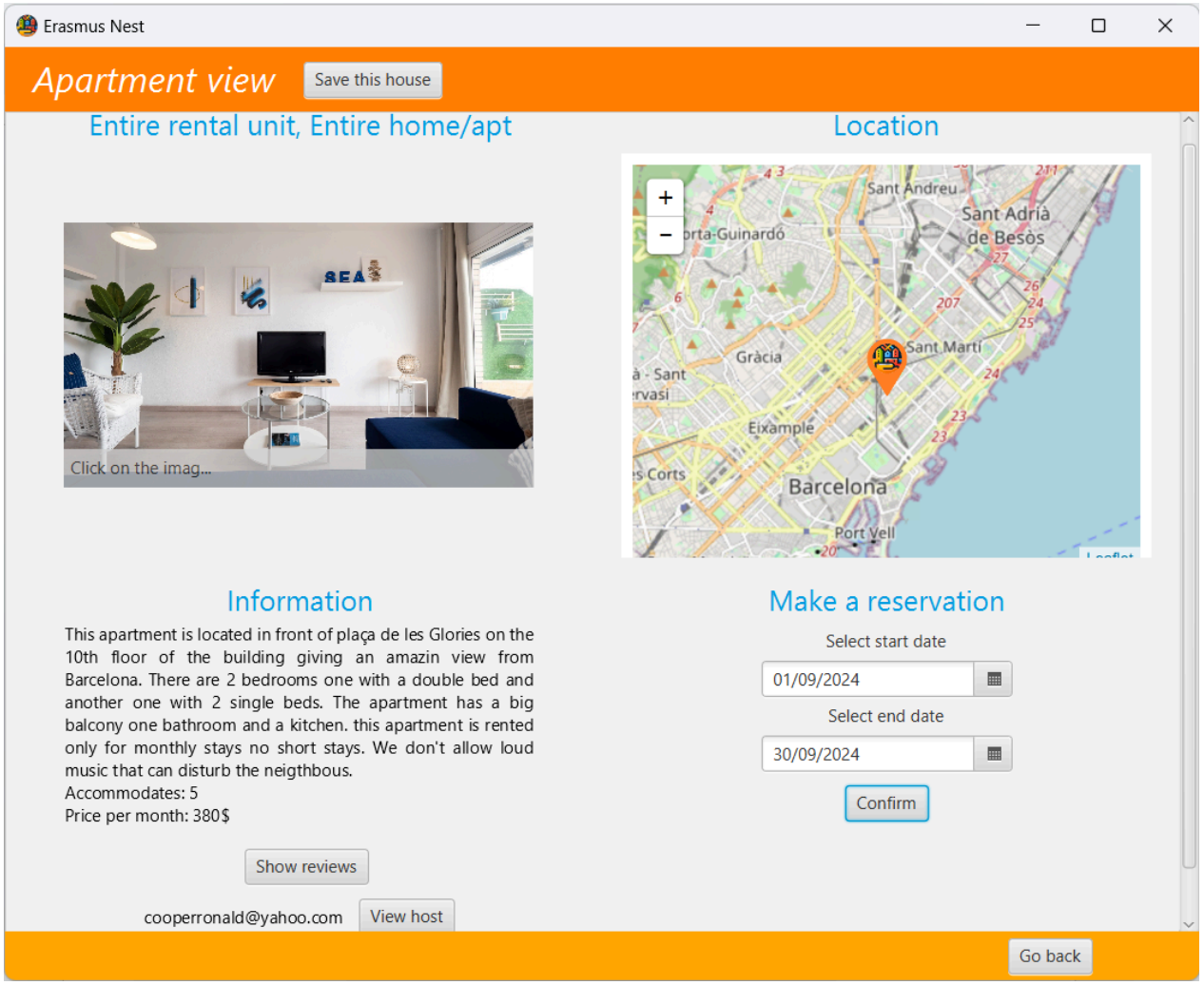
After successful registration, the user is redirected to the login page.
After successful login, the user is redirected to the homepage.



14.4. Browse apartments by city (e.g. Barcelona)



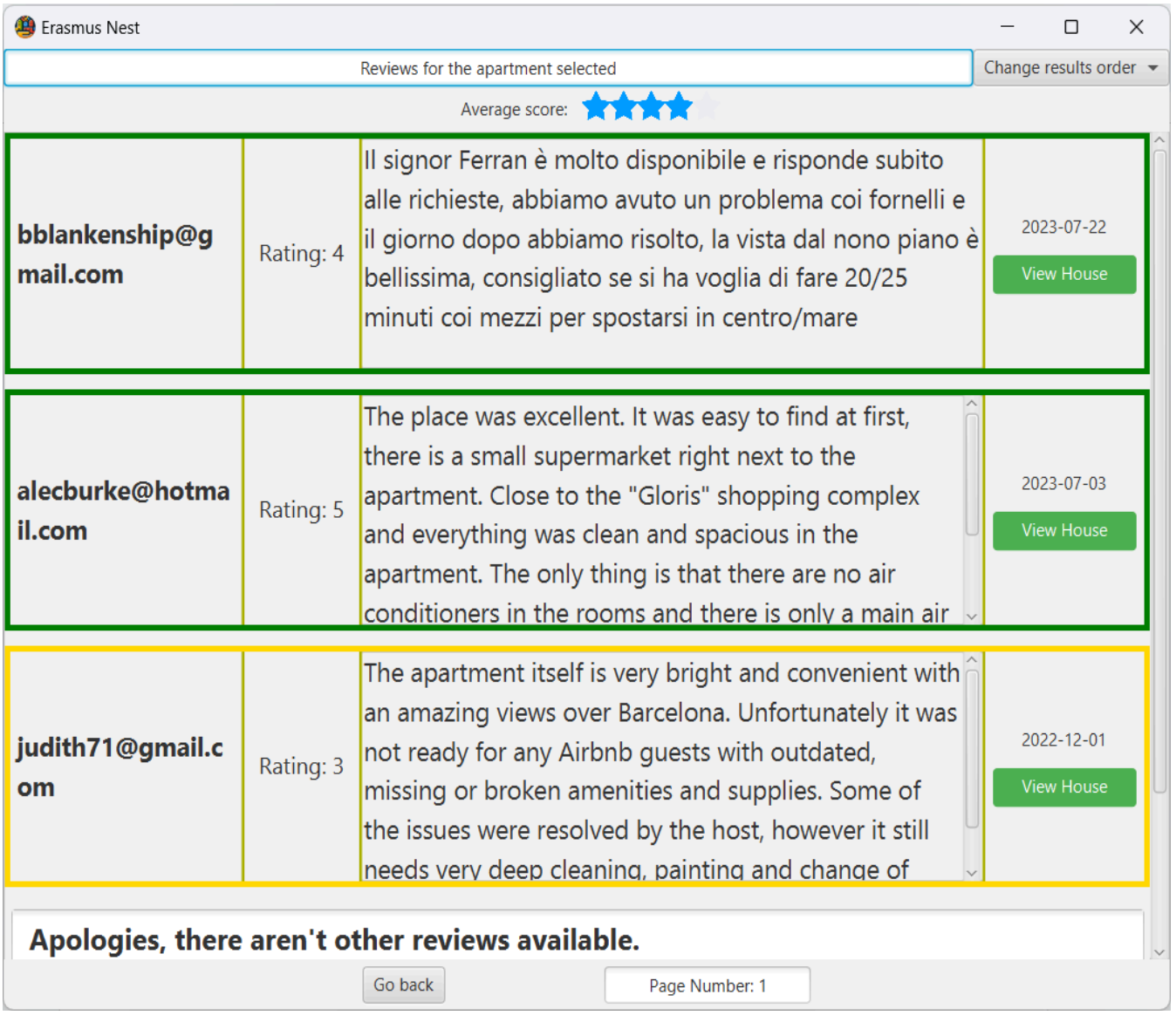
14.5. View specific apartment page



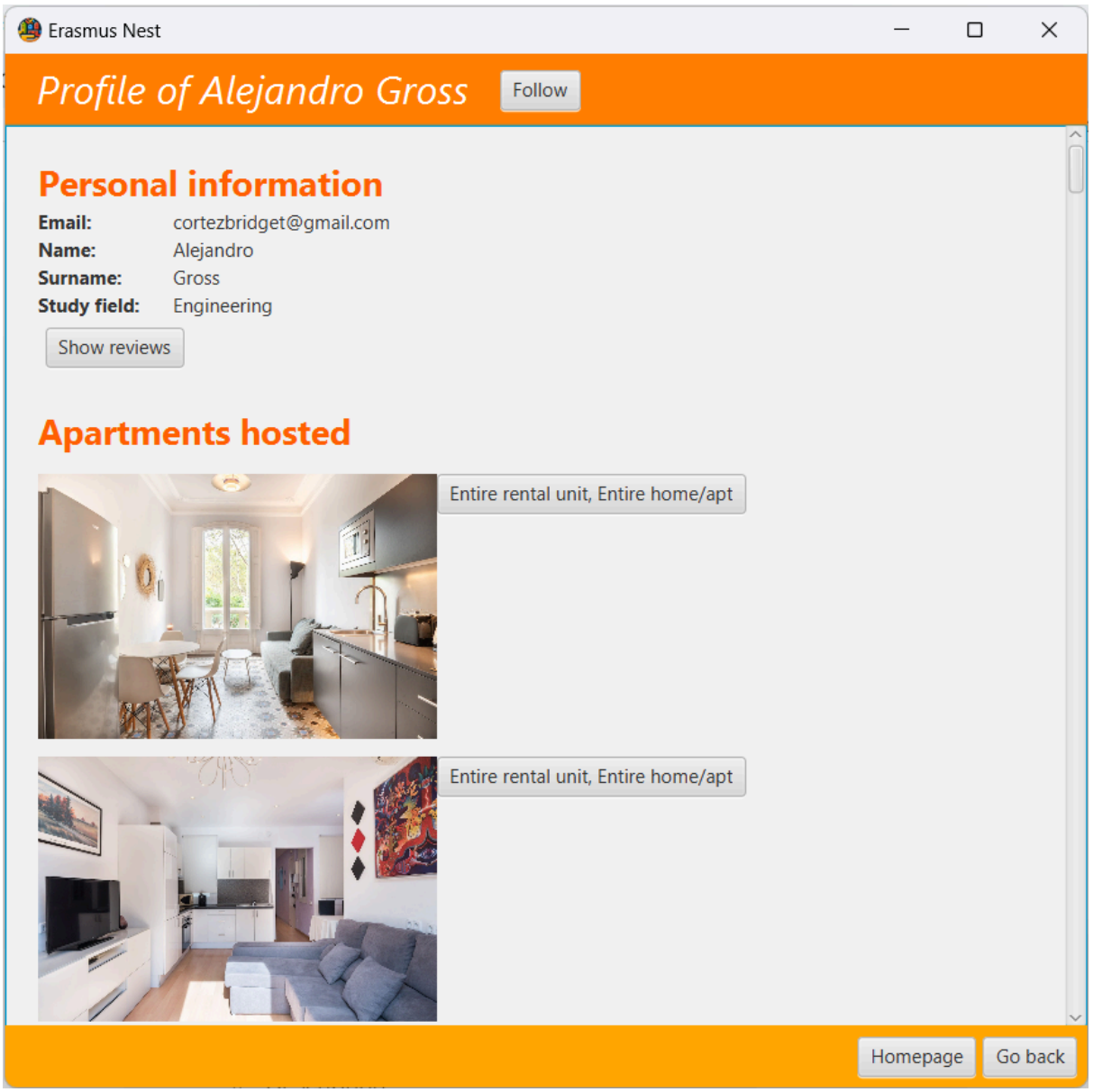
In this section is possible to:

- Access all apartment's information as
  - Description
  - Maximum number of accommodates
  - Price per month
  - Position on the map
- Access apartment's reviews list via "Show reviews" button
- Access host page via "View host" button
- Save the apartment via "Save the apartment button"
- Make a reservation specifying end and start date of the night stay

14.6. See reviews about the apartment



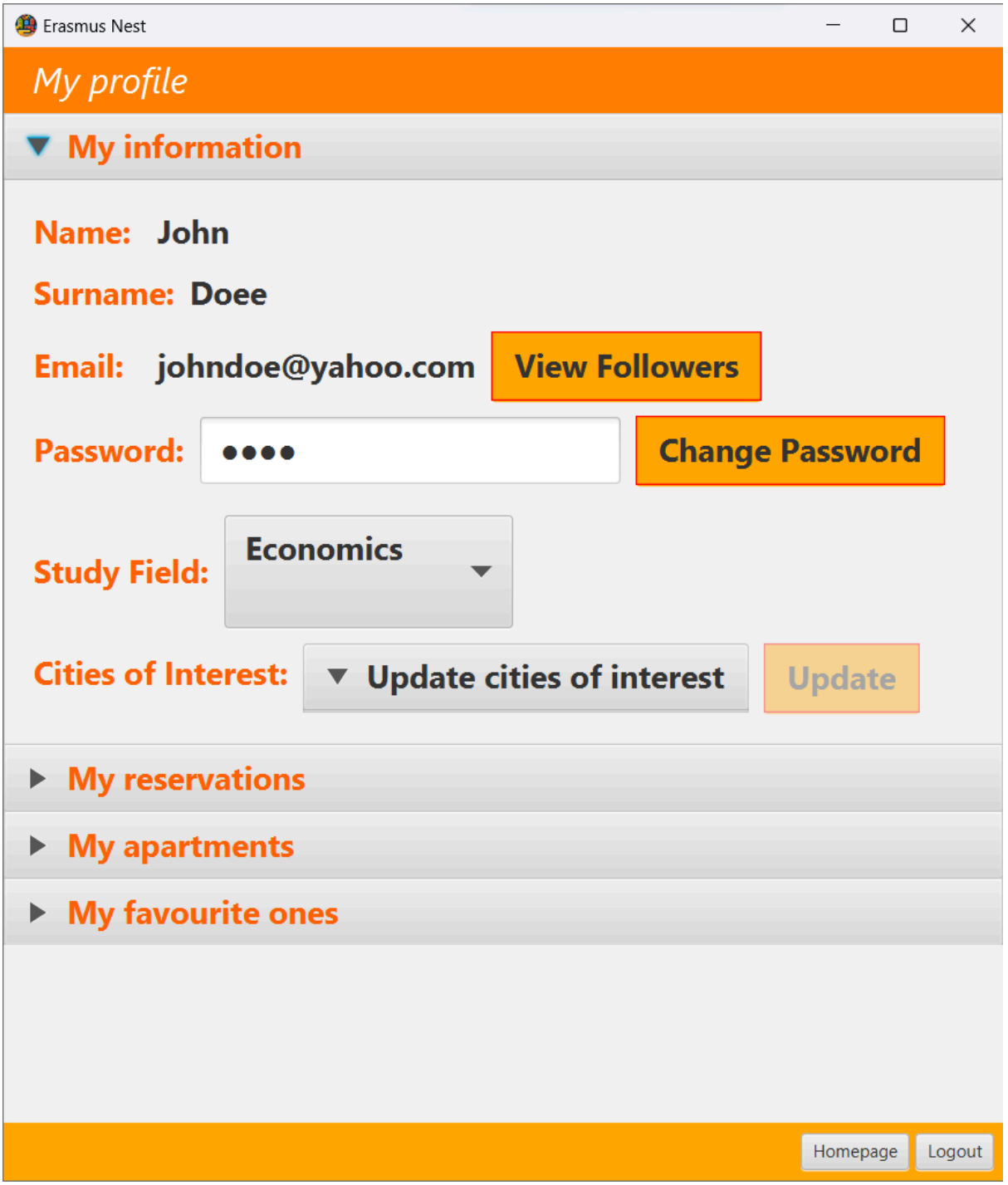
14.7. See other user personal page



Related page includes:

- User personal information
- Access reviews made by the user
- User's apartments on rent with specific button to access related apartment
- "Follow" button to start follow the user

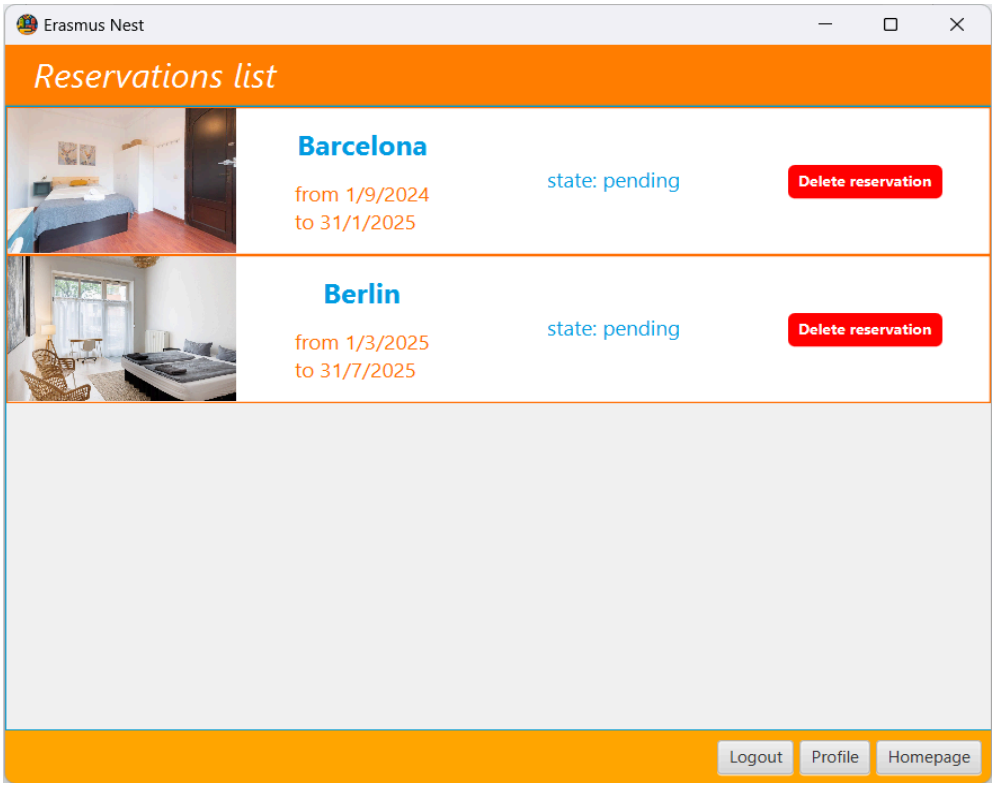
14.8. My profile page



In this page the user can:

- Change email, password, study field and list of cities interested in
- Access related sections illustrated above
- Back to homepage
- Logout

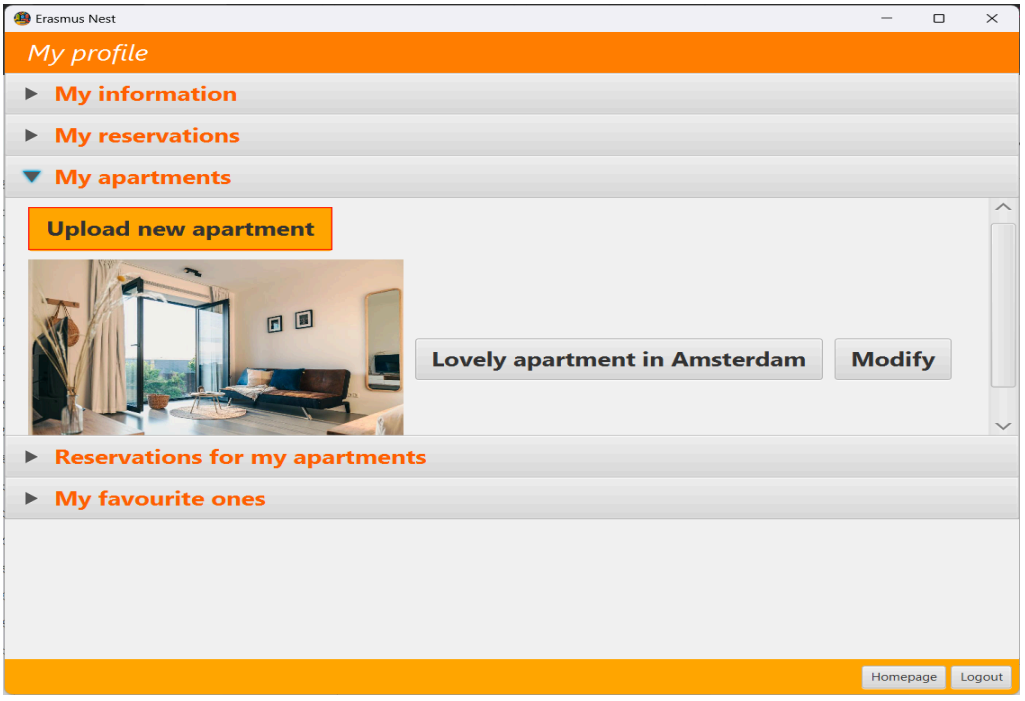
14.9. “My reservations” section (state: pending)



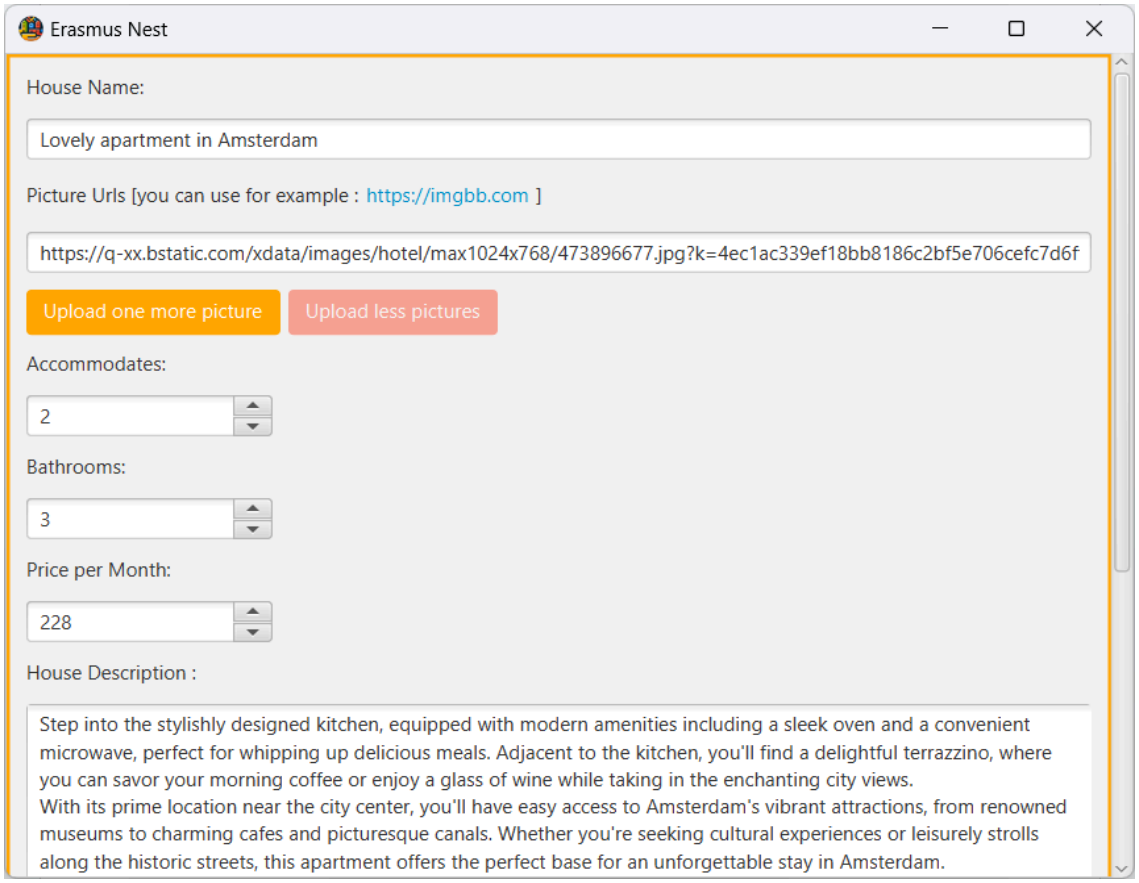
At the moment, the user is waiting for reservation confirmation and it is possible to delete it. While the host changes the state, this can be moved to “Approved” or “Rejected”.

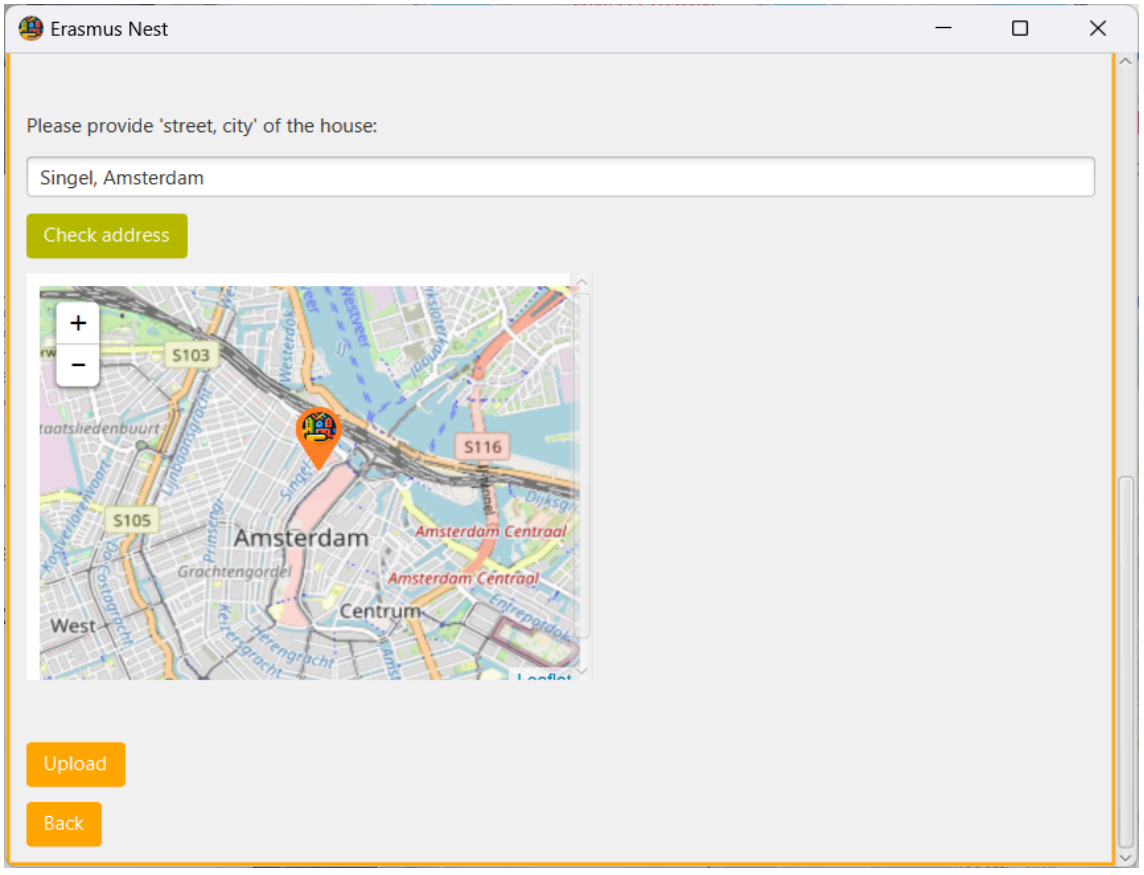
14.10. “My apartments” section

Button name redirects to apartment page while “Modify” to an updating form.

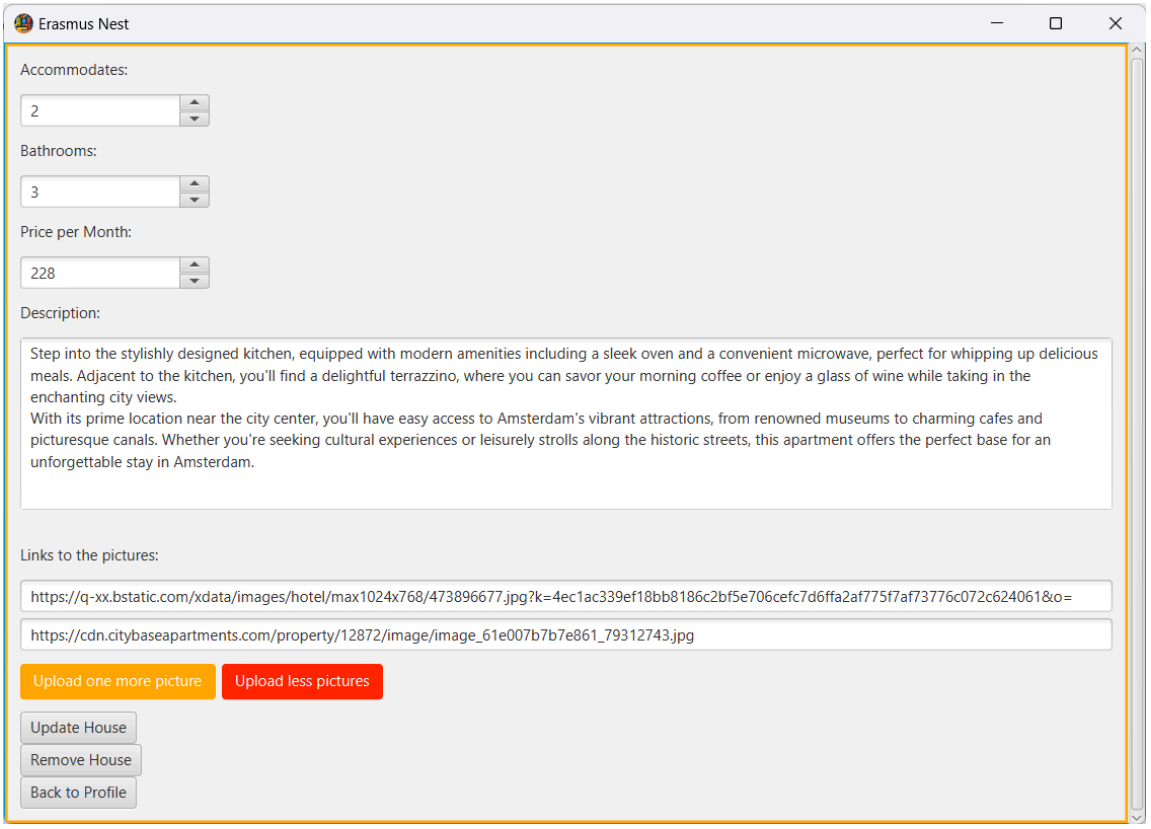


14.11. “Upload apartment” form



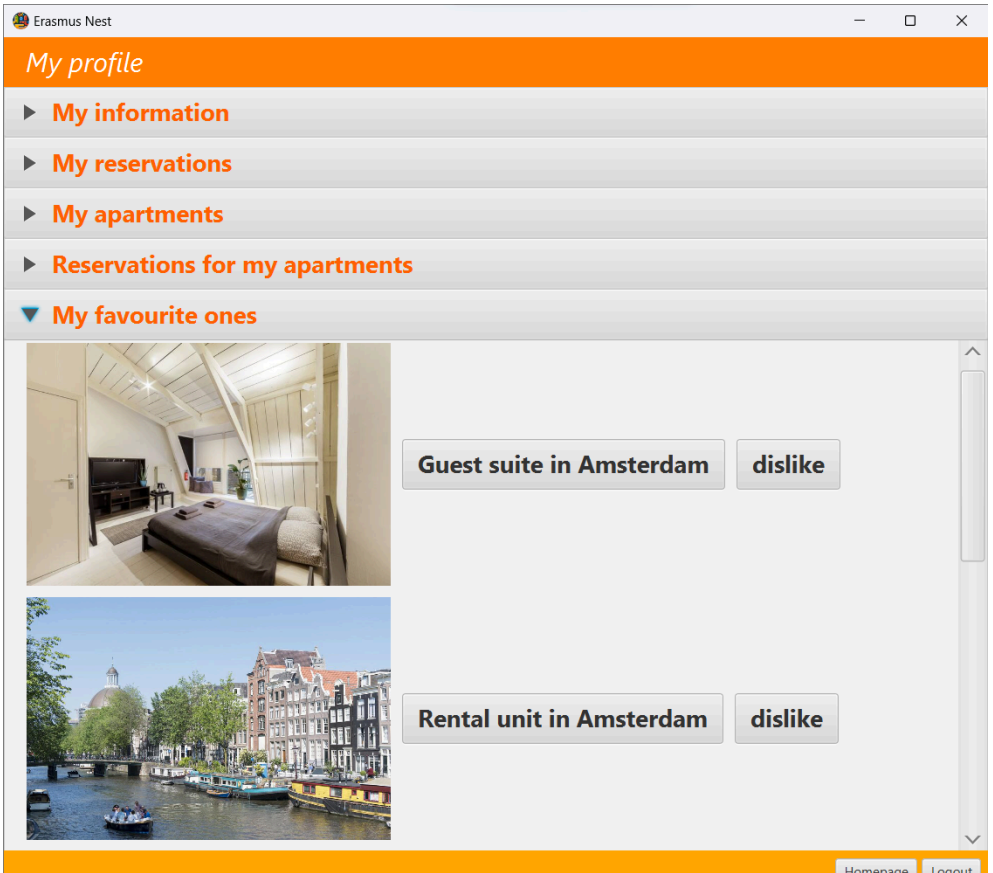


14.12. “Modify apartment” form

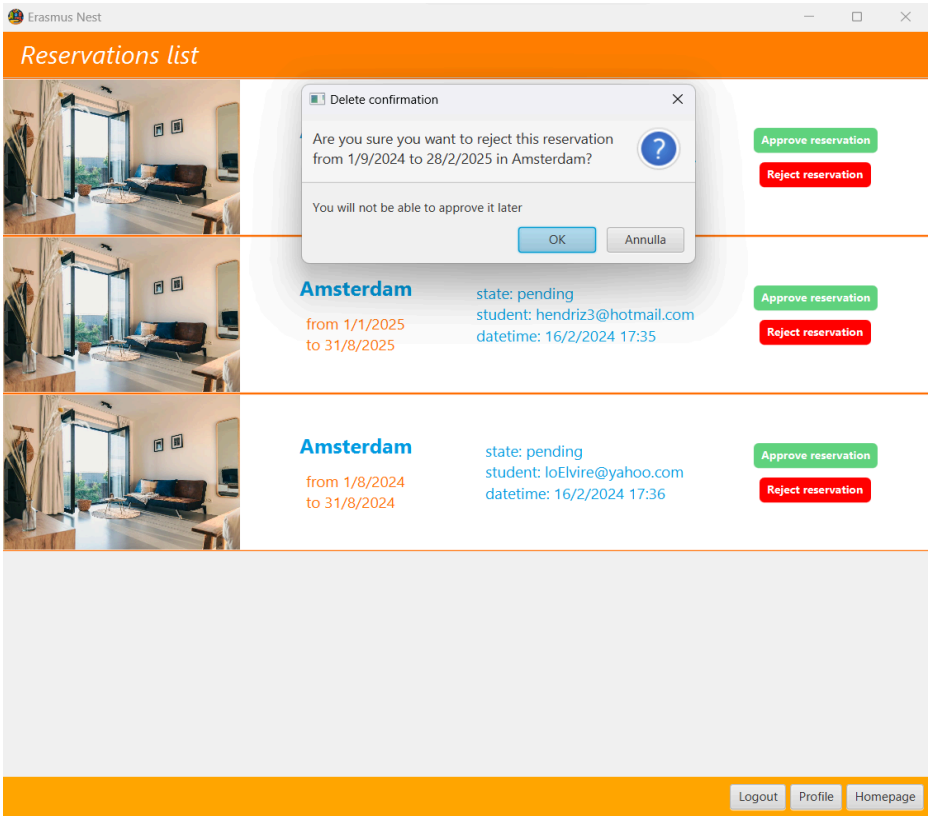


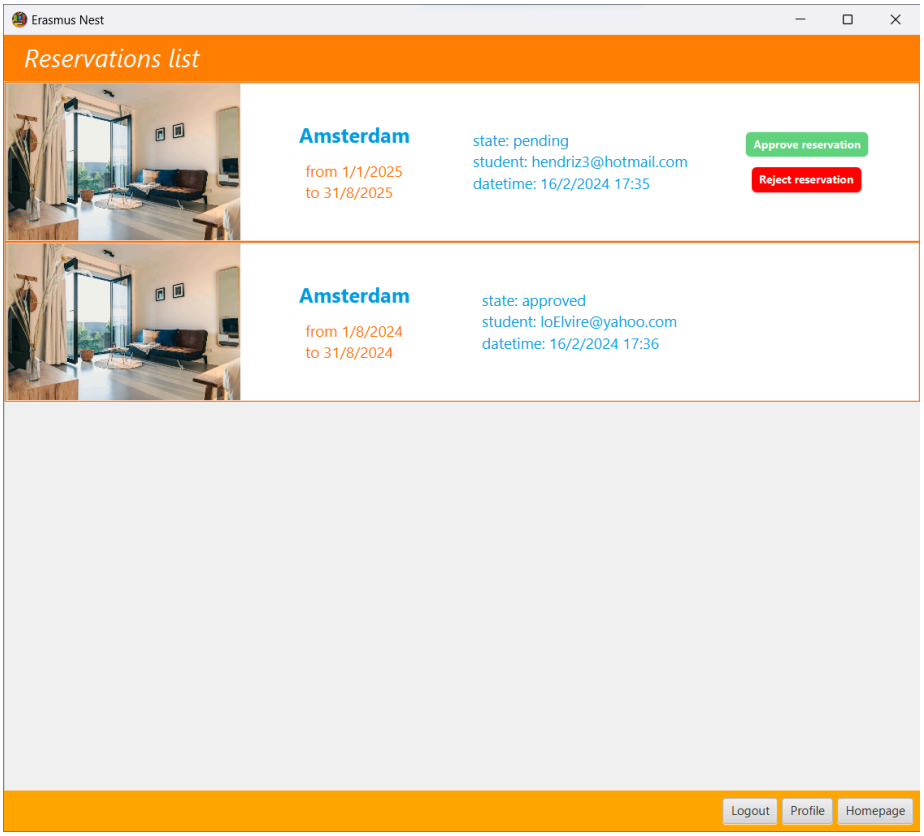
14.13. “My favourite ones” section

This section shows apartments which the user previously visited and decided to add in the specific section for an easy and rapid retrieval.
Pressing “dislike” button the apartment is removed from the section.

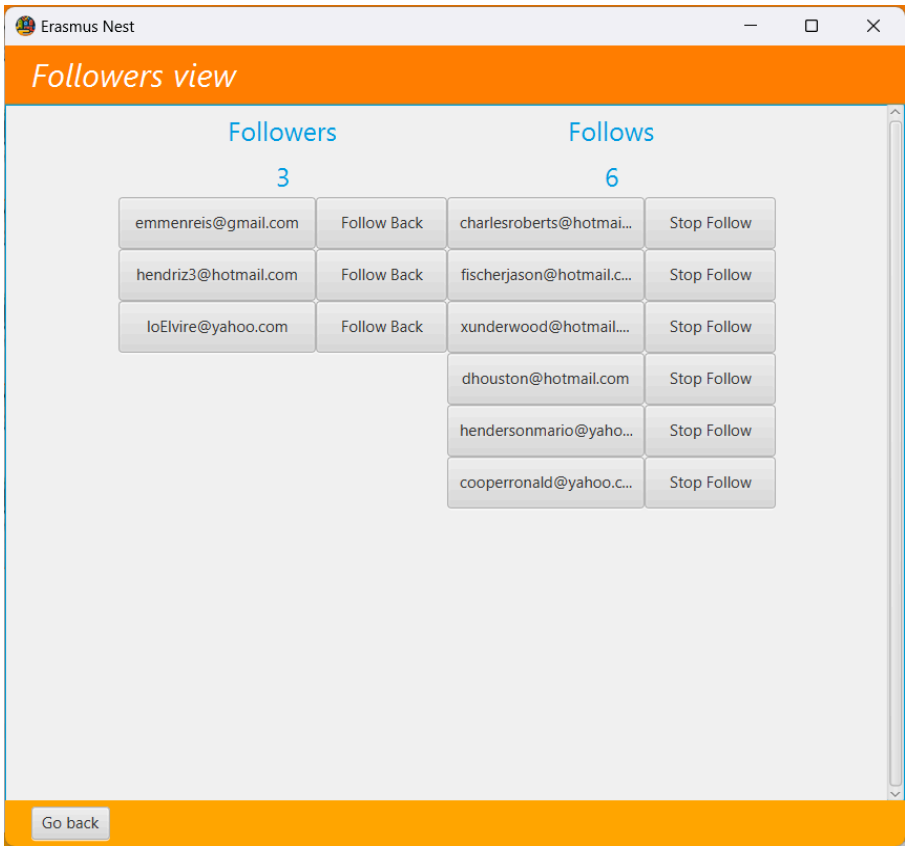


14.14. “Reservations for my apartments” section





14.15. “View followers” section



All names are embedded in related buttons to provide the possibility to access in an easy way to profile.

See section 8.5 for Analytic view, accessible from “Admin View” button in My profile section.

15. Project enrichment proposal

Here, we would like to provide a section in which we highlight some vulnerabilities of the application, and that would need an upgrade if deployed.

15.1. Password hashing

Currently, passwords within our document databases are stored in plain text, generated using Faker as outlined in the data source section. This decision was made to facilitate ease of use during development, allowing us to conveniently log in and assess the behavior of specific users by simply querying MongoDB for their password. In a production environment, of course this won't happen, so we propose a basic hashing solution that is not present in the code.

Login Java code

```
import org.mindrot.jbcrypt.BCrypt;
.
.
.
if(isEmailFieldValid(emailField) && isTextFieldValid(passwordField)) {
     String redisPassword =
getRedisConnectionManager().getPassword(emailField.getEmailAddress());

     String typedPassword = passwordField.getText().trim();

     if(redisPassword == null){ // Password not found in Redis
       System.out.println("Credentials not found in Redis.");

     String mongoPassword =
getMongoConnectionManager().getPassword(emailField.getEmailAddress());

     if(mongoPassword != null && BCrypt.checkpw(typedPassword,
mongoPassword)) {
            getSession().setLogged(true);

getSession().getUser().setEmail(emailField.getEmailAddress());

super.getRedisConnectionManager().addUser(emailField.getEmailAddress(),
BCrypt.hashpw(typedPassword, BCrypt.gensalt(12)));
     }else{
         showErrorMessage("Invalid email or password", errorTextFlow);
      }
      }else if(BCrypt.checkpw(typedPassword, redisPassword)) {
```

```
      // Password taken from Redis is equal to real pw
         getSession().setLogged(true);

getSession().getUser().setEmail(emailField.getEmailAddress());

      }else{ // Wrong password
          showErrorMessage("Invalid email or password", errorTextFlow);

      }
   }
.
.
.
```

Sign-up Java Code

```
import org.mindrot.jbcrypt.BCrypt;
.
.
.

// get User data and password from the sign-up form

String password = BCrypt.hashpw(password, BCrypt.gensalt(12));

// store password inside Redis and MongoDB

.
.
.
```

15.2. Notification message

Following application development, as it stands User experience suffers the absence of a messaging and notification system. Providing an easy way for information spreading and communication is important for all the relationships that can be instantiated among users and even for reservation state changes.

For example, consider following simple use cases with or without related notification reporting system:

User_B accept reservation made by User_A

- Actual application implementation:
    - User_A will discover to have the reservation approved only after enter the application, go to personal profile page and “My reservations” section
- With future update including notification system:
    - User_B will receive a notification

Obviously this update will simplify most of the operations made over the application, but an automatic email sending system (e.g. Bonita and Java interaction) will be even more efficient.

16. Github link for code review

All the Java code we utilized can be found in the following repository.

17. Link to the databases' files

The files used for database import can be found here. They were not uploaded in GitHub because of space limitations, so we opted for the upload inside a folder in the academic Google Drive personal account provided by the University.

If you made it this far, thanks for your attention. We hope everything was clear in this documentation. If necessary, contact us for more information.

The ErasmusNest team

- **Giulio Capecchi** <g.capecchi2@studenti.unipi.it>
- **Jacopo Niccolai** <j.niccolai@studenti.unipi.it>
- **Andrea ruggieri** <a.ruggieri2@studenti.unipi.it>

UNIVERSITÀ DI PISA