University of Pisa

Large Scale and Multi-Structured Databases

Project

Trip4Share

Bruchi Caterina

Bruni Davide

Grillea Francesco

A.Y. 2022-2023

Abstract

Trip4Share is a web-application that connects people around the world to travel together. The idea behind Trip4Share is that if a user planned to go somewhere, he/she can share the trip plan with the network in such a way to find other people with the same interest that want to join him/her. So on one side the user can create him/her own trip and get in touch with other users that sent a join request and the user could also receive reviews from them. On the other side, users can browse through available trips shared by other users, search using filters as price, date, destination and popularity and, if interested, can send a join request to the trip organizer.
Trip4Share can also show to the users the most popular destination based on criteria like price, period and tags.

The application includes also a social network side where people can follow each others in such a way to see trips shared by friends or to receive suggestion based on people they might know or places they might like to visit.

The code is available on GitHub at:
`https://github.com/DavideBruni/Trip4Share`

Contents

Chapter 1

Dataset

To populate the databases we extract data from three different sources through web scraping:

1. sivola.it to retrieve trip information;
2. ketrip.it to retrieve trip information as well;
3. it.trustpilot.com to retrieve user information and user reviews.

After the data retrieval phase, we noticed that some trip information were missing, for instance some trip may have any tag: we decided to accept this kind of "incompleteness" because it may happen that when a user creates a trip doesn't specify any tag or some optional information.

Also we assumed that username is unique for each user, so we had to manage duplicate data just adding to the username a number and removing all white spaces in it.

After this first step we merged all data in three different dataset:

1. Trips Dataset saved as `dataset_trips.json` that holds 589 different trips;
2. Users Dataset saved as `dataset_users.json` that holds 1135 different users of which one is the admin;
3. Reviews Dataset saved as `dataset_reviews.json` that holds 5356 different reviews.

1.1 Data Generation

Since the amount of data retrieved were approximately 7MB, we decided to simulate trips organized in the past years as follows:

- 2023: data retrieved through web scraping
- 2022: 10% more than 2023[1]
- 2021: 10% less than 2022[2]
- 2020: 35% less than 2021
- 2019: 73% more than 2020[3]
- 2018: 10% more than 2019[4]
- 2017: 20% less than 2018[5]

We selected trip using random sampling without replacement given by the python function `random.sample`, then we updated the year and we decreased the month by one in such a way to not have the same trip always on the same date. Also to simulate a real usage of the application during the time, we add to the trip dataset a `last_modified` field just subtracting to the departure date random integers given by the following python functions `randint(0,1)` for month, `randint(0,23)` for hours, `randint(0,59)` for minutes and `randint(0,59)` for seconds.

Also, for each trip, we generated the number of likes (i.e. the number of users that add the trip to wishlist) using a normal distribution with $\mu = 50$ and $\sigma = 10$ for expired trips (the current date is after the departure date) and a normal distribution with $\mu = 20$ and $\sigma = 4$ for not expired trips.

Since web sources scraped didn't hold nationality and spoken languages for

[1] We haven't got the report about 2022, so since we have scraped travels until the end of 2023, we expect about the 10% more, supposing not all the trips are already organized for the 2023

[2] https://www.europassistance.it/press/estate-2022-il-numero-degli-italiani-in-viaggio-cresce-e-supera-i-livelli-pre-pandemia

[3] This source is used for 2020 and 2019 `https://www.italiaatavola.net/flash/tendenze-mercato/horeca-turismo/2022/12/10/viaggi-in-aereo-nel-2021-in-aumento-del-35-rispetto-al-2020/92411/#:~:text=Nel%202021%2C%20il%20numero,alle%20restrizioni%20dovute%20alla%20pandemia.`

[4] `https://www.istat.it/it/files//2020/12/C19.pdf`

[5] `https://www.istat.it/it/archivio/227018#:~:text=Nel%202018%20si%20stima%20che,registrata%20a%20partire%20dal%202016.`

users, we decided to generate them randomly choosing from the following set `["Italian", "English", "French", "German", "Dutch", "Greek", "Spanish"]`. The nationality is chosen by generating a random index using the python function `random.randint`; the spoken languages are chosen using random sampling without replacement (given by the python function `random.sample`) where each sample has random size given by `random.randint`, but at least greater than 1.

1.2 Data Linking

In this section will be explained how we connected each other different dataset.

User can organize a trip We didn't found a real use case similar to Trip4Share, so we made an hypothesis: since a lot of people organize trip for work, we supposed that about the 20% of the total users are "organizer", i.e. users that mainly organize trips. A sort of "influencer" of our application, with many organized trips, many follower and few following.

User can follow another User We supposed that "organizers" follow a number of other users described by a normal distribution with $\mu = 45$ and $\sigma = 10$. "Normal users" instead, have a number of following people described by a normal distribution with $\mu = 10$ and $\sigma = 2$.

User can leave a review to another User We associated each "organizer" 20 reviews chosen randomly using python sampling without replacement `random.sample`; for other users we selected 2 reviews each with the same strategy of the previous, until there are no more left. The author of a review is chosen randomly using the index given by `random.randint`: note that also here we start counting from 1 because the 0th user is the admin.

User can add a trip to wishlist First of all we selected all trips that were not expired yet i.e. the departure date is further than the current date, then we selected `#likes` users using the python function `random.randint` and then we added the selected trip in the wishlist of each of these users. Note that each selected user should not be the organizer or a participant of the trip.

Chapter 2

Design

2.1 Main Actors

- *Registered User*: user that is already registered to the application. He/she can create or join a trip after the login.
- *Unregistered User*: user that accesses to the application for the first time. He/she have to sign up to become a registered user.
- *Admin*: can manage the application, banning users and managing other admins.

2.2 Functional Requirements

We will now explain in detail all the operation allowed to each of the main actors of the application

2.2.1 Unregistered user

The system must allow unregistered users to:

- **View most popular trip** according to the number of likes i.e. the most added to wishlist by users.
- **View cheapest trips starting from the current date**
- **Search for trips**: this can be done using 3 different filters
  1. *Destination* with or without date
  2. *Tag* with or without date

3. *Price Range* with or without date

- **View trip details**
- **Search for popular destinations** This can be done in 5 different way
  1. *Tag*
  2. *Price Range*
  3. *Date Intervals*
  4. *Overall*: destination where at least 25 trips were organized in the last year and they received more than 5000 likes in total
  5. *Most Exclusive*: destinations where at most 5 trips were organized in the last year, and they received more than 1000 like in total
- **See cheaper destinations on average**
- **Login**: if already registered a user can log in using an username and a password to unlock further functionalities.
- **Sign Up**

2.2.2 Registered user

In addition to all the functionalities explained for the unregistered users after the registration process the system must allow the registered user to:

- **View, edit and delete their profile**
- **View user's detailed profile**
- **View user's reviews**
- **View user's average rating**
- **View user's followings and followers**
- **View user's followings and followers number**
- **View user's organized trips**
- **View own past trips**: this list contains the trips for which the user made a join request.

- **View own wishlist**: the wishlist is a list of favourite trips the user likes, the number of people who added a trip to their wishlist is intended in the application as a metric of popularity.
- **Follow and unfollow other users**
- **Leave reviews to other users**
- **Delete own reviews**
- **Create, edit and delete own trips**
- **Add or remove a Trip from wishlist**
- **Send/Unsend a join request to a trip**
- **View join request status**
- **Delete his participation to a trip**
- **View trip participants**
- **View trip pending requests**
- **Accept/Reject trip pending requests**
- **Remove participant**
- **Visualize personalized suggested users to follow**
- **Visualize personalized suggested trips they might like**
- **See trips organized by people they follow**

2.2.3 Admin

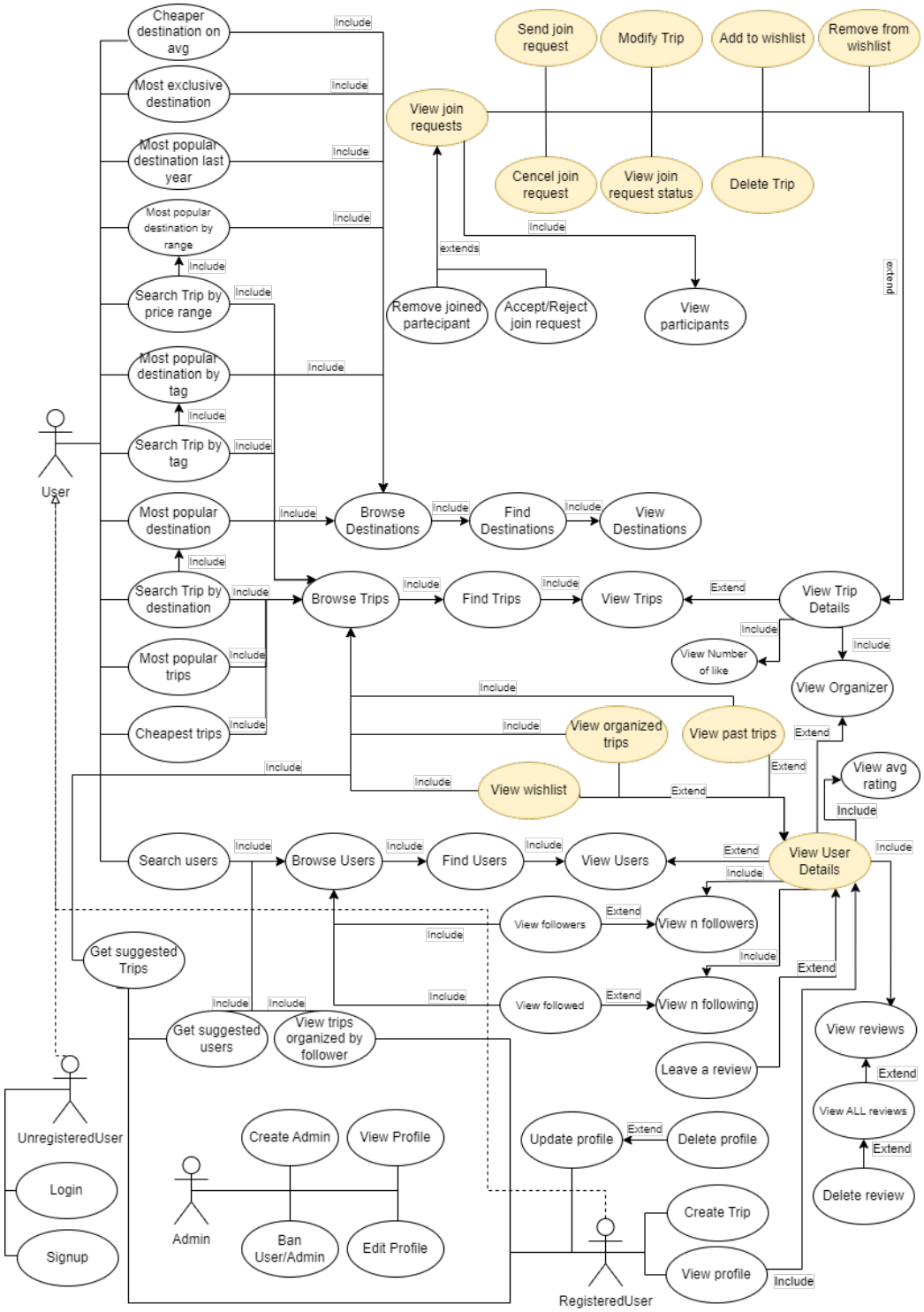
Admins are not regular users, the system must allow them to:

- **Ban other users/admin**
- **Create other admins profile**
- **View and Edit its profile information**

2.3 Non-Functional Requirements

- The system must be a website application.
- The system will be available 24 * 7.
- High availability: the service must be always available for users in order to guarantee the best possible user experience, providing a response to any query.
- Partition protection, the failure of an individual node must not impact the system as a whole.
- Low latency: the response to the requests should be fast.
- Usability: the application must be user-friendly and must ensure a low response-time.

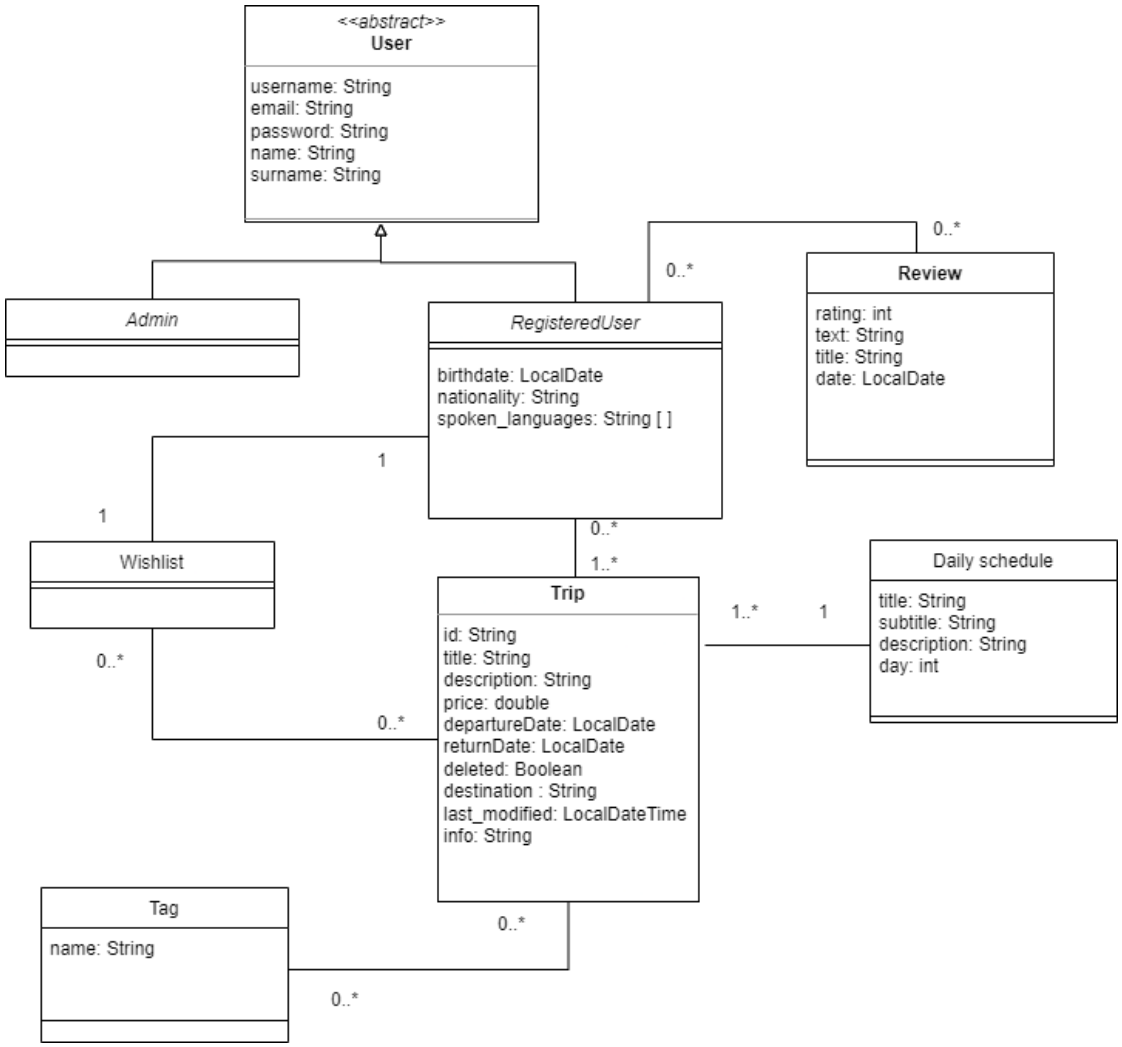
2.4 Use Case


Cheaper destination on avg
Most exclusive destination
Most popular destination last year
Most popular destination by range
Search Trip by price range
Most popular destination by tag
Search Trip by tag
Most popular destination
Search Trip by destination
Most popular trips
Cheapest trips
Search users
Get suggested Trips
Get suggested users
View trips organized by follower
Send join request
Modify Trip
Add to wishlist
Remove from wishlist
View join requests
Cencel join request
View join request status
Delete Trip
Remove joined partecipant
Accept/Reject join request
View participants
Browse Destinations
Find Destinations
View Destinations
Browse Trips
Find Trips
View Trips
View Trip Details
View Number of like
View Organizer
View organized trips
View past trips
View wishlist
View avg rating
Browse Users
Find Users
View Users
View User Details
View followers
View n followers
View followed
View n following
View reviews
Leave a review
View ALL reviews
Delete review
Update profile
Delete profile
Create Trip
View profile
Create Admin
View Profile
Ban User/Admin
Edit Profile
User
UnregisteredUser
Login
Signup
Admin
RegisteredUser
Include
Extend
extends

2.4.1 Notes about use case diagram

- All the operation "search trip by destination", "search trip by tag", "search trip by price range" include the possibility to filter results also by specifying departure date and return date. Note that, if not specified, the interval range is [current date, ∞)
- View join requests, modify and delete trip are allowed only if the user who see the trip details, is also the trip organizer
- View join requests status and cancel join requests are allowed only if the user who see the trip details, has performed a requests previously.
- View past trips and view wishlist are allowed only if the logged user is the same as the visualized profile owner.
- Get suggested trip and get suggested users works only if the user follow someone else.
- Send a join request and add/remove a Trip from the wishlist are allowed only if the user isn't also the Trip organizer.
- Sending a join request will remove the trip from wishlist if present.

2.5 UML Class Diagram



2.5.1 Classes Design

User

- **username**: represents users in the website, it's picked by the user at the sign up and it's then used to perform the login. It must be unique
- **email**: email address of the user
- **password**: password for the account, necessary to log in
- **name**: first name of the user
- **surname**: last name of the user

Registered User

It includes all of the User fields an some additional ones:

- **Birthday**: contains the date of birth of the user
- **Nationality**: contains the nationality of the user (optional)
- **Spoken languages**: all different languages spoken by the user (optional)

Admin

It includes all of the User fields

Review

- **Rating**: number from 1 to 5 representing the vote of the user to the organizer of a trip
- **Title**: a written title to review
- **Text**: a written comment to the vote (optional)
- **Date**: the date in which the review was made
- **Author**: author's username.

Trip

- **id**: is an ID that represents uniquely the trip in the whole application.
- **title**: the title for the trip given by the organizer.
- **description**: a description of the trip given by the organizer. (optional)
- **price**: the price that is planned for the trip.
- **departureDate**: date in which the trip will start.
- **returnDate**: date in which the trip will end.
- **deleted**: a value that represents if the trip has been cancelled by the creator, it will be displayed differently by the application
- **destination**: the destination of the trip.
- **last modified**: date in which the information about the trip were modified the last time.
- **info**: additional information field (optional)

Tag

- **name**: the name for the tag.
  Tags can be used by multiple trips.

Daily Schedule

- **Title**: is the title chosen to describe the dayily schedule.
- **Subtitle**: is the title chosen to describe the day. (optional)
- **Description**: a detailed description of the daily activities. (optional)
- **Day**: the day number inside the vacation.

Chapter 3

Data Model

We decided to use two types of noSql databases: Document and Graph DB.

3.1 Document database

We decide to store Users (both Registered and Admin), Trips, Reviews, Tag, and DailySchedule into the Document DB, because we need flexibility: several attributes (like nationality or birthdate for the User or description and info for the Trip) may be not always present. Since admin and registered user are different kind of users, they share only the following fields: `username`, `name`, `surname`, `email`, `password` and `type`.
We use Document DB also because we need to handle a huge amount of data as support of our website.

Main Queries on Document database

- Search for popular destinations. This can be done in 5 different way:
  - Based on tag
  - Based on price
  - Based on date intervals
  - See the most exclusive destination, i.e: Destinations where at most 5 trips were organized in the last year, and they received more than 150 like in total
  - See the most popular destination during the last year, i.e: destinations where at least 40 trips were organized in the last year, and they received more than 1200 like in total
- View cheapest trips starting from the current date

- View cheaper destination on average
- Search for trips based on
  - Destination (optionally specifying the period)
  - Price (optionally specifying the period)
  - Tag (optionally specifying the period)
- User's average rating

Other Queries on DocumentDB

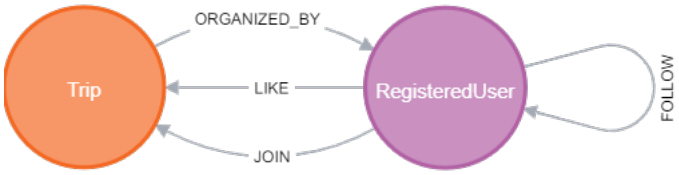
- Create, view, update or delete a trip
- Create, view or delete a review
- Create view, update or delete a user

3.2 Graph database

We saved on graph DB RegisteredUser and Trips because is very useful to represent follow, join and organization relationship as a graph between RegisteredUsers and/or Trips entities, even though this implicates some redundancies. We also saved the Wishlist entity on graphDB representing it with the relationship "LIKE" between User and Trip, this choice was lead by two main factor:

- From the dataset obtained we can immediately notice that, on average, each user adds around 10 trips to wishlist: note that expired trips (i.e. trips whose departure date hasn't passed yet) will be deleted by a daemon process that will start in background once a day in such a way to delete all the "LIKE" relationship with passed trip. Under this assumption we can esteem the number of relationship to add to cover this aspect as around 11k.
- This solution allows us to keep wishlist's trips always updated and remove the deleted ones easily. We'll explained in the section about inter database consistency 3.4.4 how this is automatically guaranteed by the delete/update trip operation.
- The LIKE relationship, used as an engagement metric, allow us to leave further implementation for suggestion on trips or other ideas open.

On the other hand while using a graph database, we have to consider the growth. This could become a problem if the users got 10 times more and the average number of likes doubled, then we would go from 10k relationships to 300k making this not a scalable solution. In that case we may have to consider other solutions, discussed in 3.3.2.



Main Queries on Graph Database

#	Domain Specific	Graph-centric
1	Show to the user "username" other suggested users to follow	Which user vertices are exactly two hop away from "username" vertex using the FOLLOW edges?
2	Show to "username" user the number of followers	How many FOLLOW edges are incoming into the "username" vertex?
3	Show to the user "username" trips he joined	Which vertexes with JOIN edges are incident to "username" vertex?
4	Show to the user "username", suggested trips	Which "trip" vertexes doesn't have linking with "username" vertex within Trip vertexes exactly one hop away using JOIN edges from vertexes one hop away from "username" vertex using FOLLOW edges vertex?

Other similar queries are:

- Show to "username" user the number of following
- Show to "username" user his/her followers/following
- Show to "username" user trips organized by people he/she is following
- Show to "username" user trips organized by the user itself

For the sake of the readability, these queries have not been reported in the table.

CRUD Operations

- View, Create, update or delete a trip
- Create or delete a user
- View, Create or delete a follow relationship
- View, Create, delete or Update a Join request.

3.3 Database Design

3.3.1 DocumentDB

As DocumentDB DBMS we choose MongoDB. We decided to create two collections:

Trips In this collection we store all the documents containing Trips information, Tags and DailySchedule. Inside each document, an array of tag (represented as string) and an array of DailySchedules (represented as Documents) are present. It is very useful, because, every time we accede to a Trip, we are also always interested in retrieve tags information and itinerary (i.e. the array of DailySchedule). A document inside the Trip collection can be represented as the following document structure

```
{
    title: "TripTitle",
    description: "TripDescription",
    destination: "TripDestination",
    departureDate: 2023-01-05,
    returnDate: 2023-01-25,
    price: 1000.00,
    itinerary:  [
                    {
                        day: 1,
                        title: "TripTitle",
                        subtitle: "TripSubtitle",
                        description: "TripDescription"
                    }
                ],
    tags:   [
                "TripTag"
            ],
    info: "TripInfo",
    likes: 100,
    whatsIncluded:  [
                        "TripWhatsIncluded"
                    ],
    whatsNotIncluded:   [
                            "TripWhatsNotIncluded
                        ],
    last_modified: 2023-01-01
}
```

thanks to the flexibility of the architecture, fields such `tags`, `info`, `whatsIncluded` and `whatsNotIncluded` can be optional. Also fields as `itinerary`, `tags`, `whatsIncluded` and `whatsNotIncluded` can have variable size.

Users In this collection we store all the documents containing Users (RegisteredUsers and Admin) and reviews information. We decide to embed Reviews documents inside User documents because the application accede to them mostly when it want to retrieve user information, for example when we want to show user's profile.

```
{
    username: "myUsername",
    name: "myName",
    surname: "mySurname",
    email: "user@trip4share.com",
    password: "myPassword",
    type: "adminOrUser",
    birthdate: 1990-01-01,
    reviews:    [
                    {
                        title: "myTitle",
                        text: "myText",
                        value: 5,
                        author: "myAuthor"
                    }
                ],
    nationality: "myNationality",
    spoken_languages:   [
                            "language_1"
                        ]
}
```

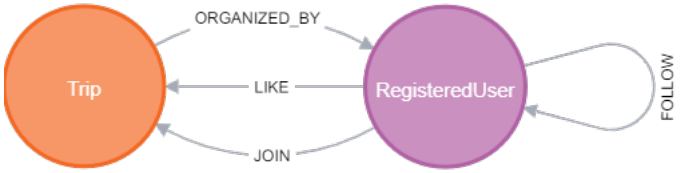
Inside the reviews, we save also the author, because we are interested in who makes the reviews only when the reviews is shown. We aren't interested in retrieve all the reviews written by a single user.

What if the embedded documents are too much?

We rejected the hypothesis that Itinerary or Reviews can grow too much for the following reasons:

- Reviews: since in the worst case, travels are joined by about 20 people and a single user organizes about 20 trips and supposing that each user that participate to the Trip leave a review, then a single user may have 400 review. To reach the 16MB size limit, each of the 400 reviews should be about 400KB. It's not a problem the growth of the document.
- Itinerary: the longest trip we have it's about 20 days, so it is very unlikely to exceed 16MB dimension limit of MongoDB for a single Document.

3.3.2 GraphDB



To store the relationships between users and trips were used Neo4J as DBMS. Since storing all document's information inside a node will grow the complexity of the graph, we decided to select a subset of attributes for each entity.

Trips

- _id, a string that holds the ObjectID of the respective MongoDB document
- title
- destination
- departureDate
- returnDate

Users

- username

We decided to store only the username because it refers uniquely to a user and we've assumed that the username is immutable, even if the user one day will delete the account.

Wishlist

We explained in the section 3.2 the motivation that led us to implement the wishlist with the relation LIKE on the GraphDB. During the design we also analyzed two further alternatives, that involved a DocumentDB to store the wishlist persistently and a Key-ValueDB as cache to store user's wishlist from the beginning to the end of user's session. Before going into details let's assume that keys would have been as `trip4share:username:trip_id`.

- The first option regarded keeping the wishlist persistently sotred in the Users collection, saving for each user a list of trip ids belonging to the user wishlist, so a list of references to elements of the Trips collection. Since we want to use a Key-ValueDB as cache, we should retrieving all trips in wishlist performing in fact a join operation between the two collection.

  Moreover, the documents we want to join could contain fields potentially large, for example Users contain all the reviews and Trips contain itinerary and variable fields like whatsIncluded and whatsNotIncluded. And also in further implementations the size could grow more due to the inclusion of the profile picture for the user and for the trip itself. This would make the join operation even more inconvenient.

- Our second option regarded creating a dedicated collection for the wishlist; it would have looked something like this:

```
{
    user_id: id
    trip_id: id
    value: {
        destination: tripDestination,
        title: tripTitle,
        departureDate: tripReturnDate,
        returnDate tripReturnDate,
        organizer: tripOrganizer
    }
}
```

  The first two fields are needed to recreate the key of the Key-ValueDB and the field value would have been a copy of the trip memorized on Redis. Using this approach, when user logs out or when the session

ends, we should move all the trips stored in Redis to MongoDB, so the system wouldn't be able to keep the document inside the Wishlist collection updated if the trip was been deleted or modified.

We didn't regard this choice as a good one (considering the current dataset) because firstly trips are usually in the wishlist for months, so they should be always updated otherwise the user could be mislead by wrong information and secondly the delete operation would have been difficult given our use case.

3.4 Distributed Database design

3.4.1 fCAP Theorem

For this application, the expectation is to have `#Reads >> #Writes` because is more likely for users to browse trips than create a trip[1]. We decided to give the priority to high availability and low latency, with a system still available under partitioning.

Considering the CAP theorem, the application is more oriented to the AP side of the triangle, favouring Availability (A) and Partition Tolerance (P) in spite of data Consistency (C), but, for the Sign-in operation, we decide to "sacrifice" Availability in spite of Consistency, we'll discuss this the paragraph 3.4.2. In order to respect the non functional requirements, we decided to guarantee high availability of data, even if an error occurs on the network layer, accepting that the content returned to the user cannot always be accurate, i.e. data displayed are shown temporarily in an old version: we can accept for example an unseen reviews or trip, because this are information write a lot a time before the needed, so it isn't much important as high availability that an user could see data update in "real time".

3.4.2 Replicas

We have access to 3 virtual machines provided by the University of Pisa:

- IP: 172.16.5.20, we decide to put an instance of MongoDB.
- IP: 172.16.5.21, we decide to put an instance of MongoDB and an instance of Neo4j server.
- IP: 172.16.5.22, we decide to put an instance of MongoDB.

[1] considering the 589 trips scraped, we have about 1.6 new trips per day

MongoDB

The configuration is shown below:

```
rsconf = {
    _id: "lsmdb",
    members: [
        {_id: 0, host: "172.16.5.20", priority: 10},
        {_id: 1, host: "172.16.5.21", priority: 2},
        {_id: 2, host: "172.16.5.22", priority: 5}
    ]
};
```

We decided to give the highest priority to the VM 172.16.5.20, so unless a problem arises, the replica on this machine will be elected as primary.

About write and read concern:

Write concern: W1 The control return from the server to the application once the first copy is written, without waiting that the primary server propagate the write to the secondary ones.
This situation could let to lost some data if the primary nodes crashes before propagate the writing and after acknowledge the write request to the application. We accept it, because we want to advantage Availability over Consistency.
About the write operation, we decide to set `retryWrties=true` in the connection options for MongoDB. In this way, we allow MongoDB drivers to automatically retry write operations a single time if they encounter network errors, or if they cannot find a healthy primary in the replica sets.

Write concern for signup: W3 The control return from the server to the application once all of the replicas acknowledge the write operation to the primary node.
This situation favour consistency over availability.
We accept it, because we want avoid a rare, but possible situation like the following:

- User A registered itself to the platform
- Mongo primary node goes down before propagate the creation of a new user on the DB

- User A cannot login again, because it is not present in replicas

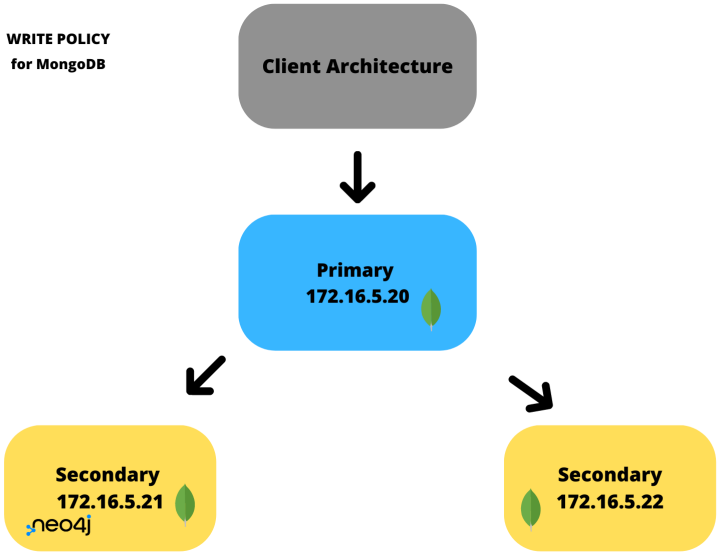
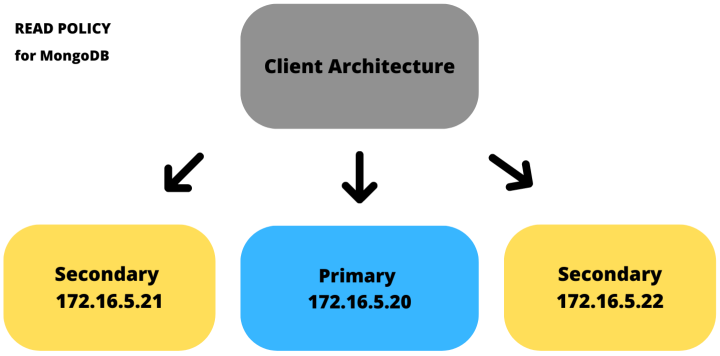


Figure 3.1: General write concern

Read preferences: nearest We allow the MongoDB driver to read from any of the replica because of the non-functional requirements low latency and high availability. We accept the fact that data obtained from the MongoDB server could be not updated to the latest version.



Replica crash In the case of failure of the primary node, since we have assigned priority to each replica, the VM 172.16.5.22 is elected as the new

primary one unless some network problem occurred.
We choose it, because the VM 172.16.5.21 have already the Neo4j instances on it, and we want to try to perform a sort of load balancing since Neo4J only has one replica and will be used more we tried to make the load on the server more severe for Mongo.

Neo4j

Replicas in Neo4j is a feature of the Enterprise edition and we have the community one.

3.4.3 Sharding proposal

In order to improve the availability of the system, we propose to adopt sharding concurrently to the data replication. Let's analyze our sharding proposal for each database.

MongoDB

Users collection We propose to use hashing as sharding algorithm and username field as shard key because we are used to retrieve the user document by username, that is always present. This sharding proposal doesn't affect the actual collection structure. Username field is also unique, and sharding support unique index forcing all the shard key as unique.

Trips collection We propose to use hashing as sharding algorithm and destination field as shard key because MongoDB doesn't support list as sharding algorithm, but even with the hashing we are able to have trips with the same destination in the same chunk, because, for the same destination, hash function return the same value. This sharding proposal doesn't affect the collection structure.

We could have used as shard key other mandatory fields, but they could force the system to perform all the query through multiple nodes. It improve also operations like: "Show most popular destination based on price/date/tag" because all of them are grouped by destination, so the pipeline could be executed in parallel.

Neo4j

Sharding in Neo4j is a feature of the Enterprise edition and we have the community one.

Even though is a premium feature we still wanted to discuss this possibility: according to Neo4j official documentation, we noticed 2 different reasons we could be interested in for graph database sharding:

- Growth of the dimension of the graph to tens bilions of nodes for example.
- Minimize latency of queries in various geographical regions.

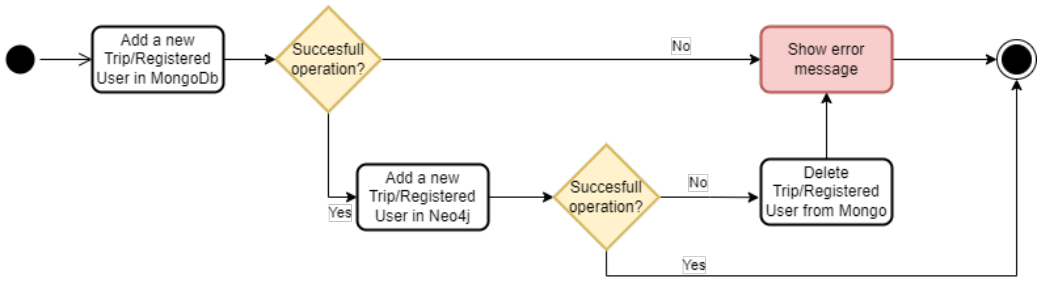
Since the dimension of the dataset stored on neo4j is much smaller than billions of nodes and since we supposed to serve the application mainly in a European context, it's useless shard the GraphDB.

In a future scenario in which the graph size grow we could take in consideration to change the license of Neo4j to use the replica and sharding features offered by the enterprise version.

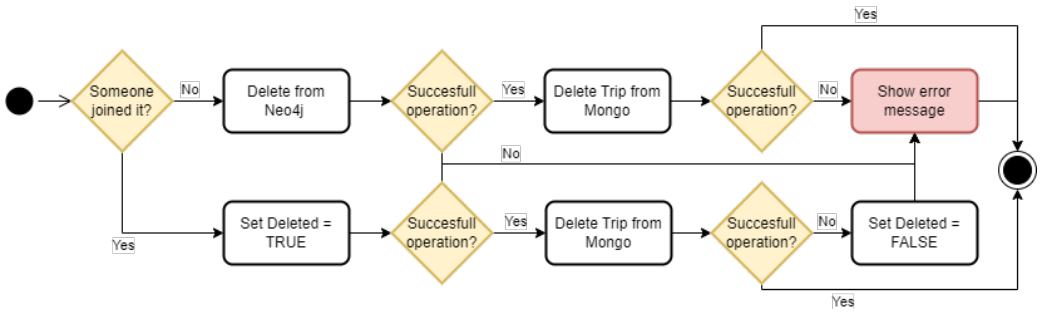
3.4.4 Inter-Database Consistency

Since we have redundant information through different databases, we need to manage the inter-database consistency for the operations that influence duplicated data.

Create Trip or RegisteredUser

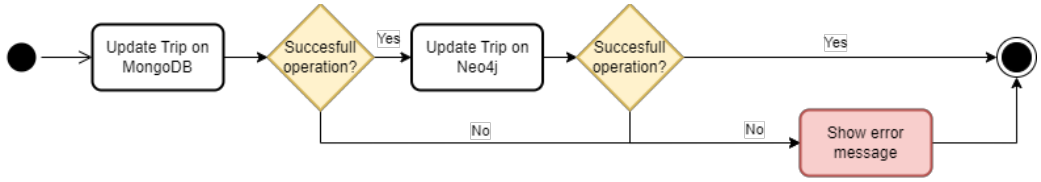


Delete Trip

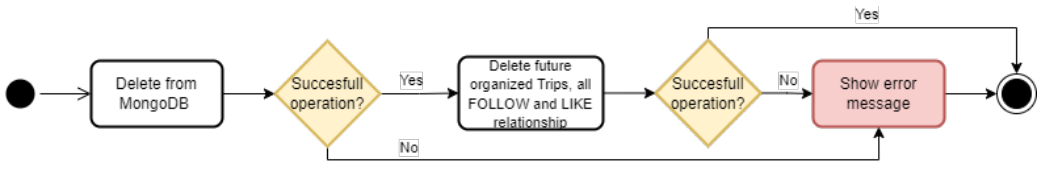


In this operation we want to keep a deleted trip on Neo4j (even if was deleted from MongoDB) if there's at least one user who joined it: this was made in such a way to leave a sort of receipt that "certificates" that user joined the trip.

Update Trip

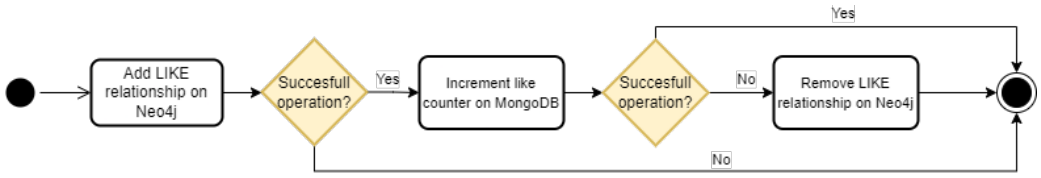


Delete Registered User

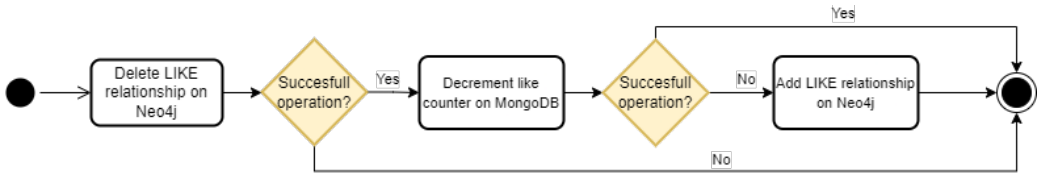


When a User is deleted, are deleted also all the future trips he/she organized.

Add Trip to wishlist



Remove Trip from wishlist



Chapter 4

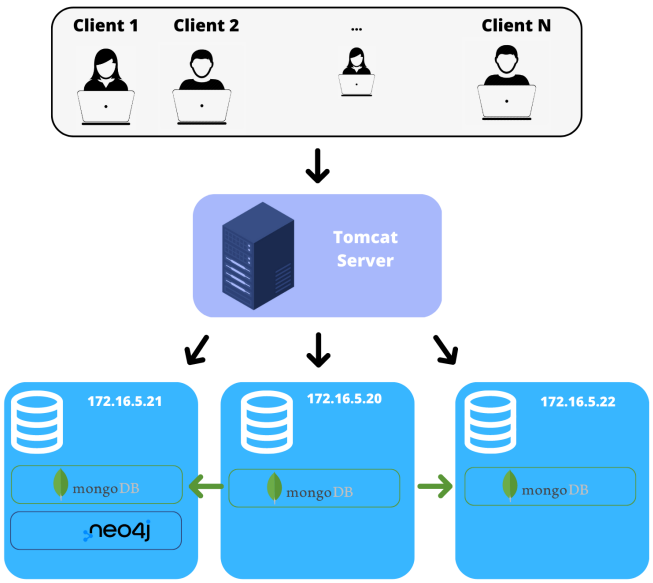
Software Architecture

The application is implemented using a client-server paradigm and following the Model-View-Controller pattern.

4.1 Client-Server Architecture

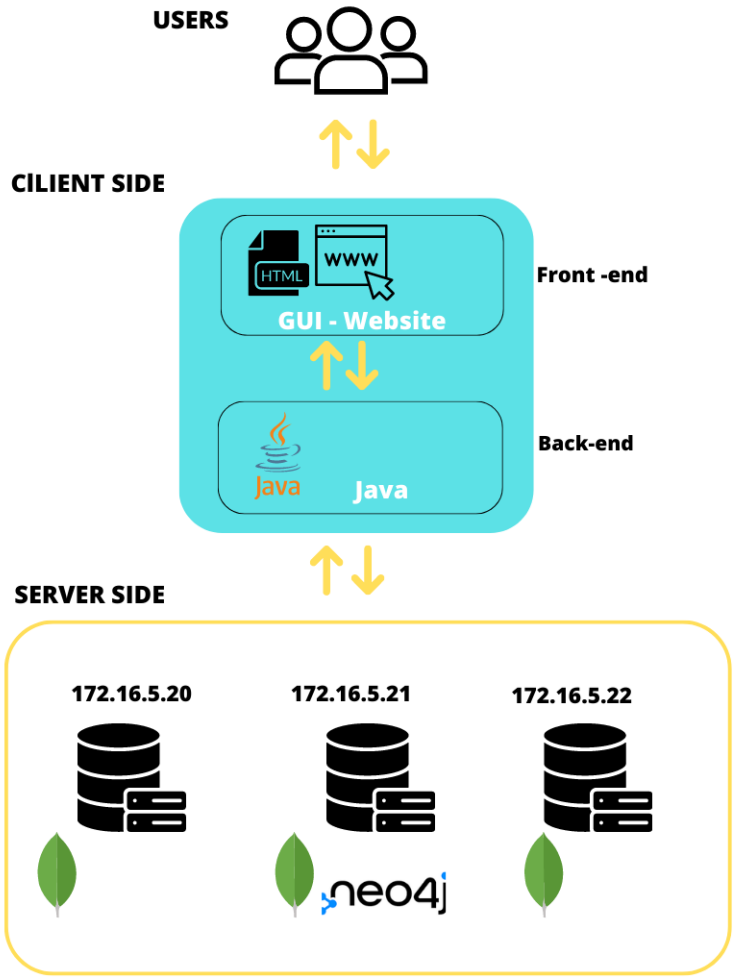
Users client Side

Client can use Trip4Share through a web application which allows the user to interact intuitively with the system.



Server Side

On the server side we have Apache Tomcat as Web Server, that using Java perform request to the database servers (acting like a client) and then generate the web pages to respond to the users requests.
Looking at the Web Server as a Client to the Database Server, we can schematize our software architecture as:



Chapter 5

Implementation and Test

In this section will be presented the implementation of the packages and the java classes used for the application.

5.1 Main Packages and Classes

it.unipi.lsmd.config

This package contains only the `AppServletContextListener.java` class that will be invoked to initialize the connections with the databases.

it.unipi.lsmd.controller

This package contains the controllers of all the different pages that can be shown to the user. In particular the file contained are:

- *AddAdminServlet*: this class gets the necessary information for the creation of a new admin. Then redirect the admin to his/her own profile.
- *AddTripServlet*: this class manages the necessary information to create a new trip, then redirects the user to a page showing the outcome of the operation.
- *AdminServlet*: this class just shows admin's own profile, if who logged in is an admin.
- *BanUserServlet*: this class allows admin to ban users removing his/her profile.
- *DeleteProfileServlet*: this class allows users to remove own profile.

- *DeleteReviewServlet*: this class gets the request from the GUI to delete a user review. It responds with a String message containing the outcome of the operation.
- *DeleteTripServlet*: this class gets the request from the GUI to delete a user trip, then redirects the user to a page showing the outcome of the operation.
- *DestinationServlet*: this class gets parameters to retrieve the most popular destination according to parameters got, then responds with the outcome of the operation.
- *ExploreDestinationsServlet*: this class manages the redirection of the user to the page designated to the search of popular destinations.
- *ExploreServlet*: this class gets parameter to search trips and suggested destination based on what parameters got, then redirect to the page showing these trips and destination. This servlet could manage also the search of other users.
- *HomeServlet*: this class handles the requests for the homepage of a registered user.
- *IndexServlet*: this class handles the requests for the website homepage.
- *JoinManagerServlet*: this class gets the parameters to accept or reject a join request to a Trip, then responds with a String containing the outcome of the accept/reject/remove operation.
- *JoinRequestServlet*: this class gets the parameters to perform a join request to a Trip, then responds with a String containing the outcome of the operation.
- *LeaveReviewServlet*: this class gets the parameters to leave a review to another registered user, then redirect to the page showing the outcome of the operation.
- *LoginServlet*: this class gets the parameters to handle the authentication functionality and redirects the user to his/her own profile (if correctly authenticated).
- *LogoutServlet*: this class handles the logout functionality and redirects the user to index.

- *PastTripsServlet*: this class gets parameter to show user's own past trips and redirect him/her to the page where all results are listed.
- *RequestsViewServlet*: this class gets parameters to obtain all users that want to join the trip, then redirects to the page showing the result.
- *SearchServlet*: this class manages the requests from the search pages, then redirect to the explore servlet.
- *SignupServlet*: this class manages gets the necessary information for the registration of a new user, then redirect to the user profile (if correctly registered).
- *TripServlet*: this class allows users to view trip details and, if a parameter is present, can manage the adding or removing from the wishlist.
- *UpdateProfileServlet*: this class gets parameters to handle the update profile functionality for both registered user and admin, then redirect to the page showing the outcome of the operation.
- *UpdateTripServlet*: this class gets parameters to handle the update trip functionality, then redirect to the page showing the outcome of the operation.
- *UserServlet*: this class allows users to view own or other users' profile.
- *WishlistServlet*: this class allows users to retrieve own wishlist.

it.unipi.lsmd.dao

This package have the interfaces to interact with databases and implements the DAOLocator.

it.unipi.lsmd.dao.base

This package establish the connection with the databases.

it.unipi.lsmd.dao.mongo

This package implement the classes to interact with MongoDB. All the classes derive from *it.unipi.lsmd.dao.base.BaseDAOMongo*

- *TripMongoDAO*: this class manages all the requests for interacting with the Trip Collection stored on MongoDB and to perform all the CRUD operations and all the aggregations regarding Trip.

- *UserMongoDAO*: this class manages all the requests for interacting with the User Collection stored on MongoDB and to perform all the CRUD operations and all the aggregations regarding User.
- *WishlistMongoDAO*: this class manages the redundancies of the trip like counter in MongoDB.

it.unipi.lsmd.dao.neo4j

This package implement the classes to interact with Neo4j. All the classes derive from *it.unipi.lsmd.dao.base.Neo4j*

- *RegisteredUserNeo4jDAO*: this class manages all the read/write operation regarding RegisteredUser nodes and the Follow relationships on Neo4j.
- *TripNeo4jDAO*: this class manages all the read/write operation regarding Trip nodes and the Join and Organized_by relationships on Neo4j.
- *WishlistNeo4jDAO*: this class manages all the read/write operation regarding Wishlist information stored on Neo4j.

it.unipi.lsmd.dto

The classes presented in this packages are used by the view and by the controller part of the application.

- *AdminDTO*
- *AuthenticatedUserDTO*
- *DailyScheduleDTO*
- *DestinationsDTO* contains information about destination, number of like and number of trips for the destination. It's used when it's necessary display the result of the aggregation operation with list of destinations as result.
- *InvolvedPeopleDTO* contains organizer and list of people who performed a join request with the respective request status. It's used for the management operation of the join request.
- *OtehrUserDTO* contains basic information about other users. Used mainly when a list of users is shown.

- *PriceDestinationDTO* contains the information showed by the aggregation returning prices and destinations.
- *RegisteredUserDTO* contains information about the registered user. It contains, for example, the avgRating, a flag for the follow relationship, the number of followers and following.
- *ReviewDTO*
- *TripDetailsDTO* contains information about the Trip, used when the details of a Trip are displayed and when a Trip is created or updated.
- *TripSummaryDTO* contains only partial information about the trip. It's used when a list of trips is shown, for example in the home page.

it.unipi.lsmd.model

In this package are really implemented the entity models

- *Admin*: extends User, containing the basic info and operation for an Admin.
- *Daily Schedule*: contains the information necessary for a daily schedule.
- *Registered User*: extends User, containing the basic info and operation for a Registered User.
- *Review*: contains the information necessary for a review to a user
- *Tag*: contains the name of a tag
- *Trip*: contains the information necessary for a trip.
- *User*: abstract Class containing the basic information for a user.
- *Wishlist*: contains the wishlist of a user.

For more detailed information, refer to the section 5.2.

it.unipi.lsmd.model.enums

This package contains the enum Status. It's represent the status of a join request.

it.unipi.lsmd.service

This packages contains the interfaces `TripService, UserService, WishlistService` and the Class `ServiceLocator`, useful to get instances of the classes implementing the interfaces. The 3 interfaces must be implemented containing the logic of the application.

it.unipi.lsmd.service.impl

This packages contains the classes `TripServiceImpl, UserServiceImpl, WishlistServiceImpl`: the implementation of the interfaces presented into the package it.unip.lsmd.service. These class manage the constraint of the application, manage consistency between databases as described in the section 3.4.4 and convert DTO objects obtained from the View Layer to the Model ones to pass at DAO classes and vice versa.

it.unipi.lsmd.utils

This class implements some utility function used by more other classes

- *LocalDateAdapter*: this class allows to produce a LocalDate object from timestamp.
- *LocalDateTimeAdapter*: this class allows to produce a LocalDateTime object from timestamp.
- *PageUtils*: this interface contains some constant values used trough the site, for the sake of platform maintainability.
- *ReviewUtils*: this class handles the conversion of a `org.bson.Document` into a review model entity and from a review model entity into a review DTO.
- *SecurityUtils*: this class handles all the utilities that controls page urls to move trough the website and the function to get the user from the session.
- *TripUtils*: this class handles the conversion from `org.bson.Document` to different types of trip information DTO classes (details and summary) and to the Trip model class and vice versa.
- *UserUtils*: this class handles the conversion from `org.bson.Document` to different types of user information DTO classes (admin) and to the admin and user model class and vice versa.

it.unipi.lsmd.utils.exceptions

This package contains the class IncompleteTripException that extends java.lang.Exception. This Exception is thrown when a create or modify trip is operation is performed and there aren't all the mandatory fields.

5.2 Java Model Classes

In this section we'll illustrate in details classes belonging to the Java Model introduced previously.

User

```
package it.unipi.lsmd.model;


public abstract class User {

    private String name;
    private String surname;
    private String username;
    private String email;
    private String password;
```

RegisteredUser

```
package it.unipi.lsmd.model;

import ...
public class RegisteredUser extends User{

    private LocalDate birthdate;
    private List<Review> reviews;
    private List<String> spoken_languages;
    private String nationality;
    private List<RegisteredUser> following;
    private List<RegisteredUser> followers;
    private List<Trip> past_trips;
    private List<Trip> organized_trips;
    private Wishlist wishlist;
```

Admin

```java
package it.unipi.lsmd.model;

public class Admin extends User{

    public Admin (){
        super();
    }

    public Admin(String username) { super.setUsername(username); }


}
```

Trip

```java
package it.unipi.lsmd.model;

import ...

public class Trip {

    private String id;
    private String title;
    private String description;
    private String destination;
    private double price;
    private LocalDate departureDate;
    private LocalDate returnDate;
    private List<Tag> tags;
    private List<DailySchedule> itinerary;
    private List<String> whatsIncluded;
    private List<String> whatsNotIncluded;
    private String info;
    private Boolean deleted;
    private int like_counter;
    private LocalDateTime last_modified;
    private RegisteredUser organizer;
    private List<Pair<RegisteredUser, Status>> joiners;
```

DailySchedule

```java
package it.unipi.lsmd.model;

public class DailySchedule {

    private String title;
    private String subtitle;
    private String description;
    private int day;
```

Review

```java
package it.unipi.lsmd.model;

import java.time.LocalDate;

public class Review {

    private String text;
    private String title;
    private LocalDate date;
    private int rating;
    private RegisteredUser author;
```

Tag

```java
package it.unipi.lsmd.model;

public class Tag {

    String tag;
```

Wishlist

```java
package it.unipi.lsmd.model;

import ...

public class Wishlist {

    private List<Trip> wishlist;
```

5.3 Constraints

- Username identify uniquely a user, even if his/her profile has been deleted.
- Trip id must be unique.
- While creating or editing a trip, `currentDate < departureDate < returnDate`.
- It's possible to perform a join request only if the user isn't the the organizer of the trip and if `currentDate < departureDate`.
- A trip can be removed only before the Departure Date.
- If a trip is deleted, and some users already joined it, they must be able to see some information about the trip (not all the details).

5.4 Redundancies

We decided to store some information about trip in both MongoDB and Neo4j:

- Trip id
- Destination
- Departure date
- Return date
- Title

These information are stored to avoid to perform an access to MongoDB each time we perform queries returning Trip on Neo4j.

5.5 Most Relevant Queries

5.5.1 MongoDB

We use MongoDB mostly to display detailed information about trips and users.

- Search by destination in period

```
public List<Trip> getTripsByDestination(String destination, LocalDate departureDate, LocalDate returnDate, int size, int page) {

    Bson m1;
    if (returnDate == null) {
        m1 = match(and(eq( fieldName: "destination", destination.toLowerCase()), gte( fieldName: "departureDate", departureDate)));
    } else {
        m1 = match(and(eq( fieldName: "destination", destination.toLowerCase()), gte( fieldName: "departureDate", departureDate),
                lte( fieldName: "returnDate", returnDate)));
    }
    Bson srt = sort(ascending( ...fieldNames: "departureDate"));
    Bson l1 = limit(size);
    Bson p1 = project(fields(include( ...fieldNames: "_id", "destination", "title", "departureDate", "returnDate", "likes")));


    AggregateIterable<Document> res;
    if (page != 1) {
        Bson s1 = skip((page - 1) * size);
        res = collection.aggregate(Arrays.asList(m1, srt, s1, l1, p1));
    } else {
        res = collection.aggregate(Arrays.asList(m1, srt, l1, p1));
    }
```

```
db.trips.aggregate([
    {$match: {$and: [
        {destination: "islanda"},
        {departureDate: {$gte : new Date("2023-01-01")}},
        {returnDate : {$lt : new Date ("2023-08-01")}}
        ]}},
    {$sort:{departureDate : 1}},
    {$skip:5},
    {$limit:5},
    {$project: {"_id": 1, "destination": 1, "title": 1,
            "departureDate": 1, "returnDate": 1, "likes": 1}}
])
```

We decided to use `$skip` and `$limit` to show a reduced number of trips per page (5). Note that we allow to not specify the return date, so the period range becomes from the departure date on.

- Search by tag in period

```java
public List<Trip> getTripsByTag(Tag tag, LocalDate departureDate, LocalDate returnDate, int size, int page) {

    Bson m1;
    if (returnDate == null) {
        m1 = match(and(eq( fieldName: "tags", tag.getTag()), gte( fieldName: "departureDate", departureDate)));
    } else {
        m1 = match(and(eq( fieldName: "tags", tag.getTag()), gte( fieldName: "departureDate", departureDate),
                lte( fieldName: "returnDate", returnDate)));
    }
    Bson srt = sort(ascending( ...fieldNames: "departureDate"));
    Bson l1 = limit(size);
    Bson p1 = project(fields(include( ...fieldNames: "_id", "destination", "title", "departureDate", "returnDate", "likes")));


    AggregateIterable<Document> res;
    if (page != 1) {
        Bson s1 = skip((page - 1) * size);
        res = collection.aggregate(Arrays.asList(m1, srt, s1, l1,  p1));
    } else {
        res = collection.aggregate(Arrays.asList(m1, srt, l1, p1));
    }
```

```
db.trips.aggregate([
    {$match: {$and: [
        {tags: "Avventura"},
        {departureDate: {$gte : new Date("2023-01-01")}},
        {returnDate : {$lt : new Date ("2023-08-01")}}
        ]}},
    {$sort:{departureDate : 1}},
    {$skip:5},
    {$limit:5},
    {$project: {"_id": 1, "destination": 1, "title": 1,
        "departureDate": 1, "returnDate": 1, "likes": 1}}
])
```

- Search by price in period

```
public List<Trip> getTripsByPrice(double min_price, double max_price, LocalDate departureDate, LocalDate returnDate, int size, int page) {

    Bson m1;
    if (returnDate == null) {
        if(max_price > 0){
            m1 = match(and(gte( fieldName: "price", min_price), lte( fieldName: "price", max_price), gte( fieldName: "departureDate", departureDate)));
        }else{
            m1 = match(and(gte( fieldName: "price", min_price), gte( fieldName: "departureDate", departureDate)));
        }
    } else {
        if(max_price > 0){
            m1 = match(and(gte( fieldName: "price", min_price), lte( fieldName: "price", max_price), gte( fieldName: "departureDate", departureDate),
                    lte( fieldName: "returnDate", returnDate)));
        }else{
            m1 = match(and(gte( fieldName: "price", min_price), gte( fieldName: "departureDate", departureDate), lte( fieldName: "returnDate", returnDate)));
        }

    }
    Bson l1 = limit(size);
    Bson p1 = project(fields(include( ...fieldNames: "_id", "destination", "title", "departureDate", "returnDate", "likes")));
    Bson srt = sort(ascending( ...fieldNames: "price"));


    AggregateIterable<Document> res;
    if (page != 1) {
        Bson s1 = skip((page - 1) * size);
        res = collection.aggregate(Arrays.asList(m1, srt, s1, l1,  p1));
    } else {
        res = collection.aggregate(Arrays.asList(m1, srt, l1, p1));
    }
```

```
db.trips.aggregate([
    {$match: {$and: [
        {price: {$gt : 1000}},
        {price: {$lt : 4000}},
        {departureDate: {$gte : new Date("2023-01-01")}},
        {returnDate : {$lt : new Date ("2023-08-01")}}
        ]}},
    {$sort:{departureDate : 1}},
    {$skip:5},
    {$limit:5},
    {$project: {"_id": 1, "destination": 1, "title": 1,
        "departureDate": 1, "returnDate": 1, "likes": 1}}
])
```

Also we allow to not specify the max price, so the price range becomes from the min price "to infinity".

- Cheapest destinations by average price:

```java
@Override
public List<Trip> cheapestDestinationsByAvg(int objectPerPageSearch) {

    Bson g1 = group( id: "$destination",avg( fieldName: "agg", expression: "$price"));
    Bson s1 = sort(ascending( ...fieldNames: "agg"));
    Bson l1 = limit(objectPerPageSearch);
    AggregateIterable<Document> res = collection.aggregate(Arrays.asList(g1, s1, l1));
```

```
db.trips.aggregate([
    {$group: {_id: "$destination", agg: {$avg: "$price" }}},
    {$sort: {agg: 1}},
    {$limit: 10}
])
```

- Cheapest trips for each destination in a given period

```java
@Override
public List<Trip> cheapestTripForDestinationInPeriod(LocalDate start, LocalDate end, int objectPerPageSearch) {

    Bson m1;
    if(end != null)
        m1  = match(and(gte( fieldName: "departureDate",start),lte( fieldName: "returnDate",end)));
    else
        m1 = match(gte( fieldName: "departureDate",start));

    Bson g1 = group( id: "$destination",min( fieldName: "min_price", expression: "$price"),first( fieldName: "doc_with_max_ver", expression: "$$ROOT"))
    Bson r1 = replaceWith( value: "$doc_with_max_ver");
    Bson s1 = sort(ascending( ...fieldNames: "price"));
    Bson l1 = limit(objectPerPageSearch);
    Bson p1 = project(fields(include( ...fieldNames: "_id", "destination", "title", "departureDate", "returnDate", "likes","price")));
    AggregateIterable<Document> res = collection.aggregate(Arrays.asList(m1, g1,r1, s1, l1, p1));
```

```
db.trips.aggregate([
    {$match : {$and: [
        {"departureDate" : {$gte : ISODate("2023-12-01")}},
        {"returnDate" : {$lte : ISODate("2023-12-31")}}
    ]}},
    {$sort: {price : 1}},
    {$group: {_id: "$destination",
        doc_with_max_ver: {$first: "$$ROOT" }}},
    {$replaceWith: "$doc_with_max_ver" }
])
```

This query returns, for each destination, the cheapest trip: in the group

stage, it takes the first document within the grouped and then replace the result with the document itself.[1]

- Most popular destinations

```
@Override
public List<Pair<String, Integer>> mostPopularDestinations(int limit) {

    Bson g1 = group( id: "$destination",sum( fieldName: "total_like", expression: "$likes"));
    Bson s1 = sort(descending( ...fieldNames: "total_like"));
    Bson l1 = limit(limit);
    AggregateIterable<Document> res = collection.aggregate(Arrays.asList(g1, s1, l1));
```

```
db.trips.aggregate([
    {$group : {"_id": "$destination",
        total_like: {$sum: "$likes"}}},
    {$sort: {total_like : -1}},
    {$limit : 5}
])
```

We assumed to base the popularity on the number of likes, so the most popular destinations are those with the highest number of likes for all trips in that destination.

- Most popular destinations for a given tag

```
@Override
public List<Pair<String, Integer>> mostPopularDestinationsByTag(Tag tag, int limit) {

    Bson m1 = match(eq( fieldName: "tags",tag.getTag()));
    Bson g1 = group( id: "$destination",sum( fieldName: "total_like", expression: "$likes"));
    Bson s1 = sort(descending( ...fieldNames: "total_like"));
    Bson l1 = limit(limit);
    AggregateIterable<Document> res = collection.aggregate(Arrays.asList(m1, g1, s1, l1));
```

```
db.trips.aggregate([
    {$match: {tags: {$eq: "tag_name"}}},
    {$group: {"_id": "$destination",
```

[1]From MongoDB documentation: "First: Returns the value that results from applying an expression to the first document in a group of documents."
"ReplaceWith: Replaces the input document with the specified document. [...] You can also specify a new document as the replacement."

```
            total_like: {$sum: "$likes"}}},
        {$sort: {total_like: -1}},
        {$limit: 5}
    ])
```

- Most popular destinations for a given price range

```
@Override
public List<Pair<String, Integer>> mostPopularDestinationsByPrice(double start, double end, int limit) {

    Bson m1;
    if(end > 0){
        m1 =  match(and(gte( fieldName: "price",start),lte( fieldName: "price",end)));
    }else{
        m1 =  match(gte( fieldName: "price",start));
    }

    Bson g1 = group( id: "$destination",sum( fieldName: "total_like", expression: "$likes"));
    Bson s1 = sort(descending( ...fieldNames: "total_like"));
    Bson l1 = limit(limit);
    AggregateIterable<Document> res = collection.aggregate(Arrays.asList(m1, g1, s1, l1));
```

```
    db.trips.aggregate([
        {$match : {$and: [{"price": {$gte: 1000}},
            {"price": {$lte: 2000}}]}},
        {$group : {"_id": "$destination",
            total_like: {$sum: "$likes"}}},
        {$sort: {total_like : -1}},
        {$limit : 5}
    ])
```

- Most popular destinations for a given period

```
@Override
public List<Pair<String, Integer>> mostPopularDestinationsByPeriod(LocalDate depDate, LocalDate retDate, int limit) {

    Bson m1 = match(and(gte( fieldName: "departureDate",depDate),lte( fieldName: "returnDate",retDate)));
    Bson g1 = group( id: "$destination",sum( fieldName: "total_like", expression: "$likes"));
    Bson s1 = sort(descending( ...fieldNames: "total_like"));
    Bson l1 = limit(limit);
    AggregateIterable<Document> res;
    res = collection.aggregate(Arrays.asList(m1, g1, s1, l1));
```

```
    db.trips.aggregate([
        {$match : {$and: [
            {"departureDate": {$gte: ISODate("2023-01-01")}},
            {"returnDate": {$lte: ISODate("2023-01-31")}}]}},
        {$group : {"_id": "$destination", total_like: {$sum: "$likes"}}},
```

```
        {$sort: {total_like : -1}},
        {$limit : 5}
    ])
```

- Most popular destinations in the last year

```
@Override
public List<Triplet<String, Integer, Integer>> mostPopularDestinationsOverall(int limit){

    Bson m1 = match(gte( fieldName: "departureDate",LocalDate.now().minusYears( yearsToSubtract: 1)));
    Bson g1 = group( id: "$destination",sum( fieldName: "total_like", expression: "$likes"),sum( fieldName: "total_trips", expression: 1));
    Bson m2 = match((and(gte( fieldName: "total_trips", value: 40),gte( fieldName: "total_like", value: 1200))));
    Bson s1 = sort(descending( ...fieldNames: "total_like"));
    Bson l1 = limit(limit);
    AggregateIterable<Document> res;
    res = collection.aggregate(Arrays.asList(m1, g1, m2, s1, l1));
```

```
    db.trips.aggregate([
        {$match : {departureDate :
            {$gte : new Date(Date.now() - 24*60*60*1000*365)}}},
        {$group : {"_id": "$destination",
            "total_trips": {$sum : 1},
            "total_likes": {$sum : "$likes"}}},
        {$match : { $and : [
            {total_trips : {$gte : 40}},
            {total_likes : {$gte : 1200}}
        ]}},
        {$sort : {total_likes : -1}},
        {$limit : 5}
    ])
```

In this case we based the popularity on the number of likes as before and on the number of trips organized in that destination. So a destination was popular last year if were organized at least 40 trips in it and the sum of likes for all trips was at least 1200.

- Most exclusive destinations in the last year

```
@Override
public List<Triplet<String, Integer, Integer>> mostExclusive(int limit){
    Bson m1 = match(gte( fieldName: "departureDate",LocalDate.now().minusYears( yearsToSubtract: 1)));
    Bson g1 = group( id: "$destination",sum( fieldName: "total_like", expression: "$likes"),sum( fieldName: "total_trips", expression: 1));
    Bson m2 = match((and(lte( fieldName: "total_trips", value: 5),gte( fieldName: "total_like", value: 150))));
    Bson s1 = sort(descending( ...fieldNames: "total_like"));
    Bson l1 = limit(limit);
    AggregateIterable<Document> res;
    res = collection.aggregate(Arrays.asList(m1, g1, m2, s1, l1));
```

```
db.trips.aggregate([
    {$match : {departureDate:
        {$gte : new Date(Date.now() - 24*60*60*1000*365)}}},
    {$group : {"_id":"$destination",
        "tot_trips": {$sum : 1},
        "tot_likes":{$sum : "$likes"}}},
    {$match: { $and : [
        {tot_trips : {$lte : 5}},
        {tot_likes : {$gte : 150}}
    ]}},
    {$sort : {tot_likes : -1}},
    {$limit : 5}])
```

As the popularity, we based the "exclusiveness" on the number of likes and the number of trips organized in that destination. So a destination is exclusive if there are few trips organized in it but those trips received a lot of likes.

- User's average rating

```
@Override
public double avgRating(RegisteredUser registeredUser){

    if(registeredUser == null)
        return 0.0;

    Bson m1 = match(eq( fieldName: "username", registeredUser.getUsername()));
    Bson u1 = unwind( fieldName: "$reviews");
    Bson g1 = group( id: "$username",avg( fieldName: "rating", expression: "$reviews.value"));
    AggregateIterable<Document> res = collection.aggregate(Arrays.asList(m1,u1,g1));
```

```
db.users.aggregate([
    {$match: {username: "Vincenzo0"}},
    {$unwind: "$reviews"},
    {$group : {"_id":"$username", rating:{$avg:"$reviews.value"}}},
])
```

The average rating given by the reviews left on the user's profile.

5.5.2 Neo4J

On Neo4J we save information about the follow in the application, the trip participants and request to join trips. The main queries are

- Show suggested users

```
public List<RegisteredUser> getSuggestedUser(RegisteredUser user, int nUser){

    if(user == null)
        return null;

    List<RegisteredUser> suggested;
    try (Session session = getConnection().session()) {
        suggested = session.readTransaction(tx -> {
            Result result = tx.run( s: "MATCH (u1:RegisteredUser {username:$username})-[:FOLLOW]->(u2:RegisteredUser)," +
                                    "(u2)-[:FOLLOW]->(u3:RegisteredUser) WHERE u1.username <> u3.username AND " +
                                    "(NOT (u1)-[:FOLLOW]->(u3)) RETURN DISTINCT u3.username, rand() as r " +
                                    "ORDER BY r LIMIT $limit",
                    parameters( ...keysAndValues: "username", user.getUsername(), "limit",nUser));
```

```
MATCH (u1:RegisteredUser {username:"Vincenzo0"})
    -[:FOLLOW]->(u2:RegisteredUser),
    (u2)-[:FOLLOW]->(u3:RegisteredUser)
WHERE u1.username <> u3.username AND
    (NOT (u1)-[:FOLLOW]->(u3))
RETURN DISTINCT u3.username, rand() as r
ORDER BY r LIMIT 10
```

We assumed as "suggested users" all users followed by someone we're following.

- Show followers (or followings)

```
@Override
public List<RegisteredUser> getFollower(RegisteredUser user, int size, int page){

    if(user == null)
        return null;

    List<RegisteredUser> followers;
    try (Session session = getConnection().session()) {
        followers = session.readTransaction(tx -> {
            Result result = tx.run( s: "MATCH (u1:RegisteredUser {username:$username})<-[:FOLLOW]-(u2:RegisteredUser)"+
                                    "RETURN u2.username " +
                                    "SKIP $skip " +
                                    "LIMIT $limit",
                    parameters( ...keysAndValues: "username", user.getUsername(), "skip", ((page-1)*size), "limit", size));
```

```
MATCH (u1:RegisteredUser {username:"Vincenzo0"})
    <-[:FOLLOW]-(u2:RegisteredUser)
RETURN u2.username
SKIP 10
LIMIT 10
```

Note that is possible to show following just changing the arrow's direction of the relationship `FOLLOW`. As we did in MongoDB, we used `SKIP` and `LIMIT` to show a reduced number of users per page (10).

- Show number of followers (and followings)

```
@Override
public int getNumberOfFollower(RegisteredUser user){

    if(user == null)
        return 0;

    int followers;
    try (Session session = getConnection().session()) {
        followers = session.readTransaction(tx -> {
            Result result = tx.run( s: " MATCH (user:RegisteredUser {username:$username})<-[:FOLLOW]-()"+
                                    " RETURN count(*) as in",
                    parameters( ...keysAndValues: "username", user.getUsername()));
```

```
MATCH (user:RegisteredUser {username:"Vincenzo0"})
    <-[:FOLLOW]-()
RETURN count(*) as in
```

As discussed on the previous point, is possible to show the number of following just changing the arrow's direction of the relationship `FOLLOW`.

- Show suggested trips

```
@Override
public List<Trip> getSuggestedTrip(RegisteredUser registeredUser, int numTrips){

    if(registeredUser == null)
        return null;

    List<Trip> tripsList;
    try (Session session = getConnection().session()) {
        tripsList = session.readTransaction(tx -> {
            Result result = tx.run( s: "MATCH (r1:RegisteredUser{username : $username})-[:FOLLOW]->(r2:RegisteredUser) -[:JOIN]->(t:Trip) " +
                                    "WHERE t.departureDate > date() AND (NOT (r1)-[:JOIN]->(t)) AND (NOT (t)-[:ORGANIZED_BY] -> (r1)) AND t.deleted = FALSE " +
                                    "RETURN t._id, t.destination, t.departureDate, t.returnDate,t.title, t.deleted, r2.username as organizer, rand() as ord " +
                                    "ORDER BY ord LIMIT $limit",
                    parameters( ...keysAndValues: "username", registeredUser.getUsername(), "limit", numTrips));
```

```
MATCH (r1:RegisteredUser{username: "Vincenzo0"})
    -[:FOLLOW]->(r2:RegisteredUser) -[:JOIN]->(t:Trip)
WHERE t.departureDate > date() AND (NOT (r1)-[:JOIN]->(t))
    AND (NOT (t)-[:ORGANIZED_BY] -> (r1))
    AND t.deleted = FALSE
RETURN t._id, t.destination, t.departureDate, t.returnDate,
    t.title, t.deleted,
    r2.username as organizer, rand() as ord
ORDER BY ord LIMIT 10
```

We assumed as "suggested trips" all trips joined by someone user's following.

- Show trips organized by followed users

```
@Override
public List<Trip> getTripsOrganizedByFollower(RegisteredUser registeredUser, int size, int page) {

    if(registeredUser == null)
        return null;

    List<Trip> tripsList;
    try (Session session = getConnection().session()) {
        tripsList = session.readTransaction(tx -> {
            Result result = tx.run( s: "MATCH (r1:RegisteredUser{username : $username})-[:FOLLOW]->(r2:RegisteredUser) <-" +
                    "[:ORGANIZED_BY]-(t:Trip) WHERE t.deleted = FALSE AND t.departureDate > date() " +
                    " RETURN t._id, t.destination, t.departureDate, t.returnDate, t.title, t.deleted, r2.username as organizer" +
                    " ORDER BY t.departureDate SKIP $skip LIMIT $limit",
                    parameters( ...keysAndValues: "username", registeredUser.getUsername(), "skip", ((page-1)*size),"limit",size));
```

```
MATCH (r1:RegisteredUser{username : "Vincenzo0")-[:FOLLOW]
    ->(r2:RegisteredUser)<-[:ORGANIZED_BY]-(t:Trip)
WHERE t.deleted = FALSE AND t.departureDate > date()
RETURN t._id, t.destination, t.departureDate, t.returnDate,
    t.title, t.deleted, r2.username as organizer
ORDER BY t.departureDate
SKIP 10 LIMIT 10
```

- Show user's past trips

```
public List<Trip> getPastTrips(RegisteredUser registeredUser, int size, int page){

    if(registeredUser == null)
        return null;

    List<Trip> trip_list;

    try (Session session = getConnection().session()) {
        trip_list = session.readTransaction(tx -> {
            Result result = tx.run( s: "MATCH (r:RegisteredUser{username: $username})-[:JOIN]->(t:Trip)-[:ORGANIZED_BY]->(r1:RegisteredUser) " +
                        "RETURN t._id, t.destination, t.departureDate, t.returnDate, t.title, t.deleted, r1.username as organizer " +
                        "ORDER BY t.departureDate DESC " +
                        "SKIP $skip " +
                        "LIMIT $limit",
                    parameters( ...keysAndValues: "username", registeredUser.getUsername(), "skip", ((page-1)*size), "limit", size));
```

```
MATCH (r:RegisteredUser{username: "Vincenzo0"})-[:JOIN]->
    (t:Trip)-[:ORGANIZED_BY]->(r1:RegisteredUser)
RETURN t._id, t.destination, t.departureDate, t.returnDate,
    t.title, t.deleted, r1.username as organizer
ORDER BY t.departureDate DESC
SKIP 10
LIMIT 10
```

- Show trips organized by user

```java
public List<Trip> getTripOrganizedByUser(RegisteredUser organizer, int size, int page){

    if(organizer == null)
        return null;

    List<Trip> trip_list;

    try (Session session = getConnection().session()) {
        trip_list = session.readTransaction(tx -> {
            Result result = tx.run( s: "MATCH (t:Trip)-[:ORGANIZED_BY]->(r:RegisteredUser{username: $username}) " +
                            "RETURN t._id, t.destination, t.departureDate, t.returnDate, t.title, t.deleted, r.username as organizer " +
                            "ORDER BY t.departureDate DESC " +
                            "SKIP $skip " +
                            "LIMIT $limit",
                    parameters( ...keysAndValues: "username", organizer.getUsername(), "skip", ((page-1)*size), "limit", size));
```

```
MATCH (t:Trip)-[:ORGANIZED_BY]->(r:RegisteredUser{username: "Vincenzo0"})
RETURN t._id, t.destination, t.departureDate, t.returnDate, t.title,
    t.deleted, r.username as organizer
ORDER BY t.departureDate
SKIP 10
LIMIT 10
```

5.5.3 Indexes

In this section we evaluate the indexes we used in different databases.

MongoDB

Users collection

#	Index Name	Index Type	Properties	Attribute
0	username_1	Single, Ascending	Unique	username

Queries who benefit of it:

- Authenticate:

```
db.users.find(
    {username:"Vincenzo0", password: "vldcvwgj"},
    {reviews:{$slice: [0,3]}}
).explain("executionStats")
```

- GetUser:

```
db.users.find(
    {username:"Vincenzo0"},
    {reviews:{$slice: [0,3]}}
).explain("executionStats")
```

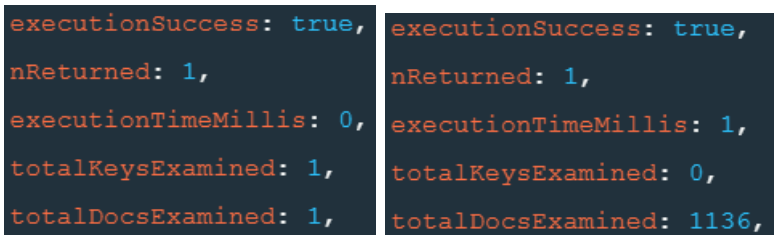


Figure 5.1: using index vs don't use it

We create this index because username must be unique and because, as we can see from the stats report by executing the queries with and without index, the performance are much more better with, because MongoDB has not to scan all the collection.

Trips collection

#	Index Name	Index Type	Properties	Attribute
0	departureDate_1	Single, Ascending	Not unique	departureDate

Queries who benefit of it:

- Search by destination in period

```
db.trips.aggregate([
    {$match: {$and: [
        {destination: "islanda"},
        {departureDate: {$gte : new Date("2023-01-01")}},
        {returnDate : {$lt : new Date ("2023-08-01")}}
        ]}},
    {$sort:{departureDate : 1}},{$limit:5}
]).explain("executionStats")
```

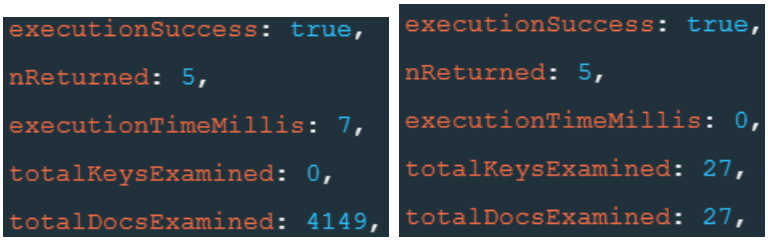


Figure 5.2: With and without index

- Search by tag in period

```
db.trips.aggregate([
    {$match: {$and: [
        {tags: "Avventura"},
        {departureDate: {$gte : new Date("2023-01-01")}},
        {returnDate : {$lt : new Date ("2023-08-01")}}
        ]}},
    {$sort:{departureDate : 1}},
    {$skip:5},
    {$limit:5},
    {$project: {"_id": 1, "destination": 1, "title": 1,
        "departureDate": 1, "returnDate": 1, "likes": 1}}
]).explain("executionStats")
```

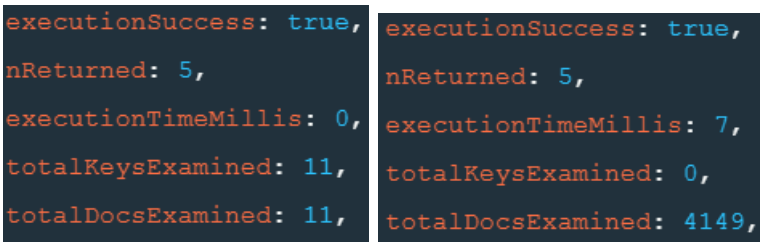


Figure 5.3: With and without index

- Search by price in period

```
db.trips.aggregate([
    {$match: {$and: [
        {price: {$gt : 1000}},
        {price: {$lt : 4000}},
```

```
        {departureDate: {$gte : new Date("2023-01-01")}},
        {returnDate : {$lt : new Date ("2023-08-01")}}
        ]}},
    {$sort:{departureDate : 1}},
    {$skip:5},
    {$limit:5},
    {$project: {"_id": 1, "destination": 1, "title": 1,
        "departureDate": 1, "returnDate": 1, "likes": 1}}
]).explain("executionStats")
```

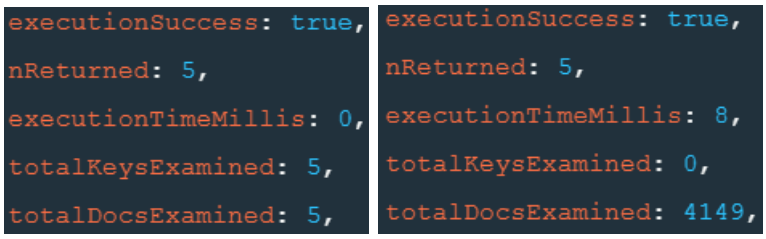


Figure 5.4: With and without index

- Cheapest trips for each destination in a given period

```
db.trips.aggregate([
    {$match : {$and: [
        {"departureDate" : {$gte : ISODate("2023-12-01")}},
        {"returnDate" : {$lte : ISODate("2023-12-31")}}
    ]}},
    {$sort: {price : 1}},
    {$group: {_id: "$destination",
        doc_with_max_ver: {$first: "$$ROOT" }}},
    {$replaceWith: "$doc_with_max_ver" }
]).explain("executionStats")
```

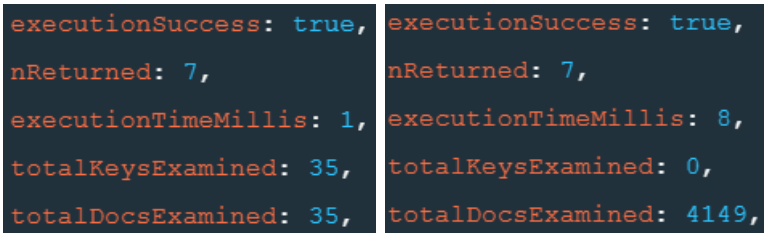


Figure 5.5: With and without index

- Most popular destinations for a given period

```
db.trips.aggregate([
    {$match : {$and: [
        {"departureDate": {$gte: ISODate("2023-01-01")}},
        {"returnDate": {$lte: ISODate("2023-01-31")}}]}},
    {$group : {"_id": "$destination", total_like: {$sum: "$likes"}}},
    {$sort: {total_like : -1}},
    {$limit : 5}
]).explain("executionStats")
```

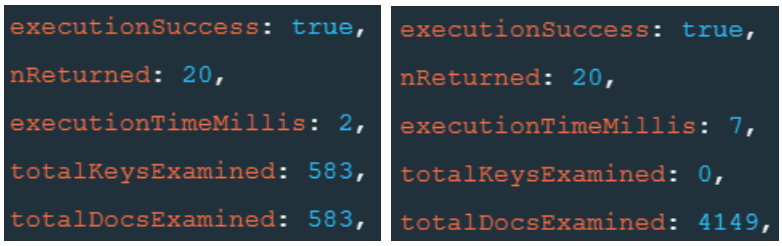


Figure 5.6: With and without index

- Most popular trips

```
db.trips.aggregate([
    {$match: {departureDate: {$gte: new ISODate()}}},
    {$sort: {likes: -1}},
    {$limit: 9}
])
```

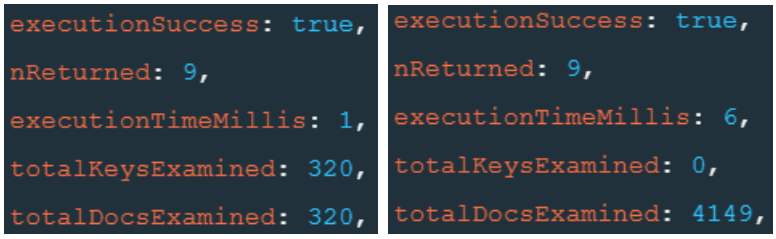


Figure 5.7: With and without index

- Most popular destinations in the last year

```
db.trips.aggregate([
    {$match : {departureDate :
        {$gte : new Date(Date.now() - 24*60*60*1000*365)}}},
    {$group : {"_id": "$destination",
        "total_trips": {$sum : 1},
        "total_likes": {$sum : "$likes"}}},
    {$match : { $and : [
        {total_trips : {$gte : 40}},
        {total_likes : {$gte : 1200}}
    ]}},
    {$sort : {total_likes : -1}},
    {$limit : 5}
]).explain("executionStats")
```

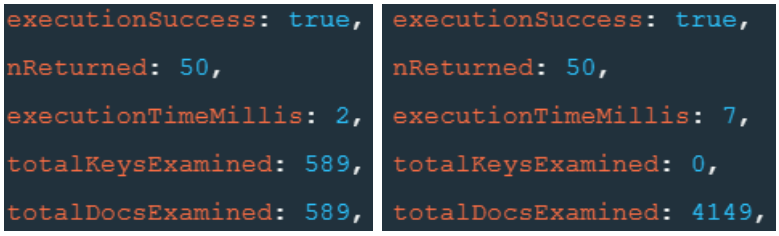


Figure 5.8: With and without index

- Most exclusive destinations in the last year

```
db.trips.aggregate([
    {$match : {departureDate:
        {$gte : new Date(Date.now() - 24*60*60*1000*365)}}},
    {$group : {"_id":"$destination",
        "tot_trips": {$sum : 1},
        "tot_likes":{$sum : "$likes"}}},
    {$match: { $and : [
        {tot_trips : {$lte : 5}},
        {tot_likes : {$gte : 150}}
    ]}},
    {$sort : {tot_likes : -1}},
    {$limit : 5}]).explain("executionStats")
```

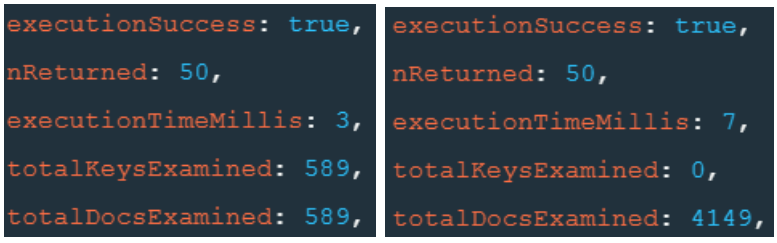


Figure 5.9: With and without index

We create this index because, as we can see from the reports, it increase many queries performance. We took also in considerations the following option for creating indexes:

- Index compound by Price and departureDate: we rejected it because prices have a low standard deviation, and this index would be useful if prices inserted in price range searching had very similar value.
  With the current data, even with the use of the index, the query examine about half of the total documents.
  However, the index used (`departureDate_1`), improve also the performances.

- Index compound by departureDate:1, returnDate:1: we rejected it because we obtained almost always the same result of the index departureDate_1, and the few improvements are in the order of tens of documents analyzed.

Neo4j

Index name	Type	Entity Type	Label	Properties	Owning Constraint
username_unique	range	node	RegisteredUser	username	username_unique
_id_unique	range	node	Trip	_id	_id_unique

The two indexes are introduced for two reasons:

1. username and trip _id must be unique, and to add this constraints, Neo4j force us to introduce the two indexes.
2. as we see from the reports, the queries improve their performances in considerable way. All the reading queries benefit of the introduction of the two indexes.

To make readable the stats reports, part of them are cut, and there are reported only the main changes.

- Get Suggested User

```
PROFILE MATCH (u1:RegisteredUser {username:"Antonio14"})-
[:FOLLOW]->(u2:RegisteredUser),(u2)-[:FOLLOW]->
(u3:RegisteredUser) WHERE u1.username <> u3.username AND
(NOT (u1)-[:FOLLOW]->(u3))
RETURN DISTINCT u3.username, rand() as r ORDER BY r LIMIT 5
```

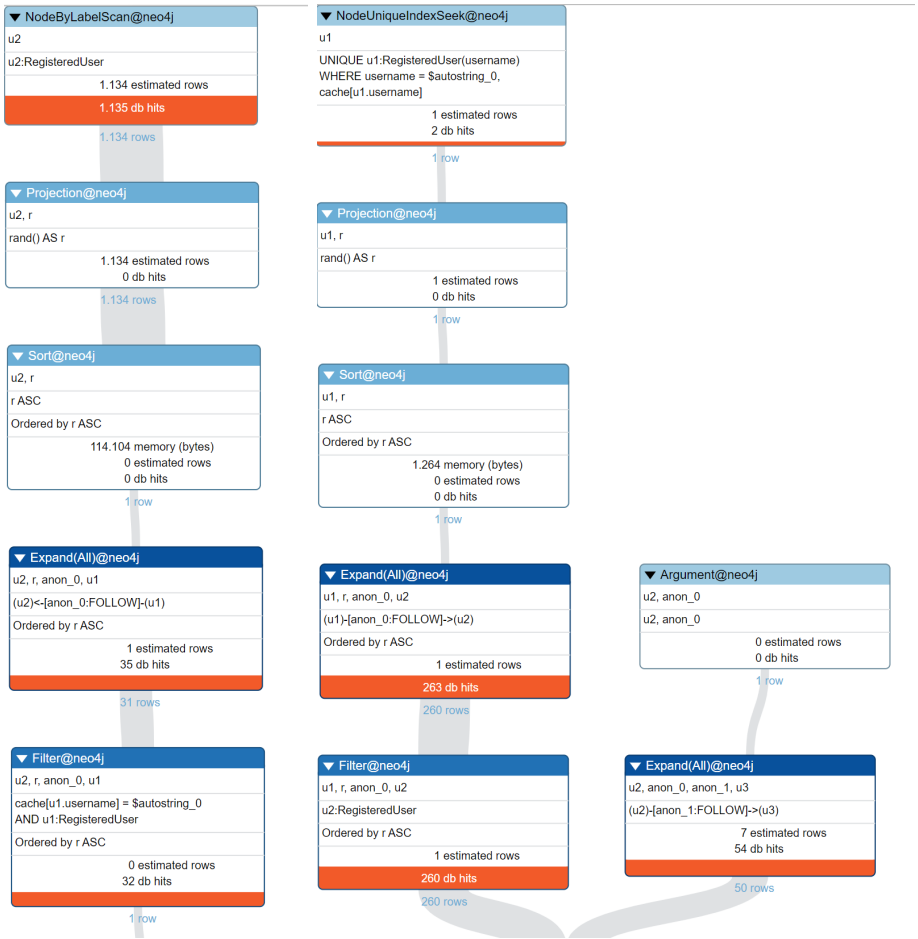


Figure 5.10: Without and with index

In the image reporting the without stats report, an important stats is missing: to perform the following part of the total query `(u1)-[anon_2:FOLLOW]->(u3)`, Neo4J perform 2.064 db hits vs the 54 we find in the same query executed using the index.

- Get Following Users

```
PROFILE MATCH (u1:RegisteredUser {username:"Antonio14"})-
[:FOLLOW]->(u2:RegisteredUser) RETURN u2.username
SKIP 0 LIMIT 10
```

- Get Follower Users

```
PROFILE MATCH (u1:RegisteredUser {username:"Antonio14"})<-
[:FOLLOW]-(u2:RegisteredUser) RETURN u2.username
SKIP 0 LIMIT 10
```

Since the two previous queries have the same db hits, for the sake of the readability, only the stats about "Get following users" is reported.

Figure 5.11: With and without index

- Exists follow relationship?

```
PROFILE MATCH (u1:RegisteredUser {username: "Loni_0"})-
[f:FOLLOW]->(u2:RegisteredUser{username: "Antonio14"})
RETURN f
```

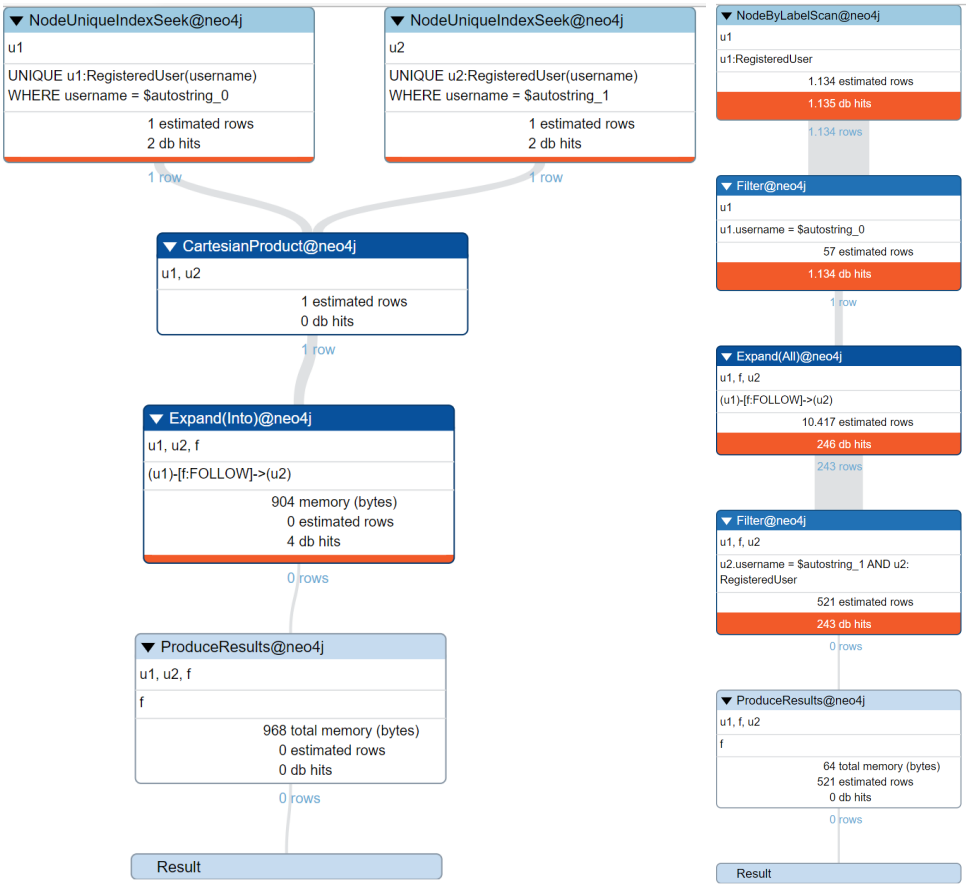


Figure 5.12: With and without index

- Get future trips organized by user

```
PROFILE MATCH (x:RegisteredUser {username: "Vincenzo0"})<-
[:ORGANIZED_BY]-(t:Trip) WHERE t.departureDate > date()
return t._id
```

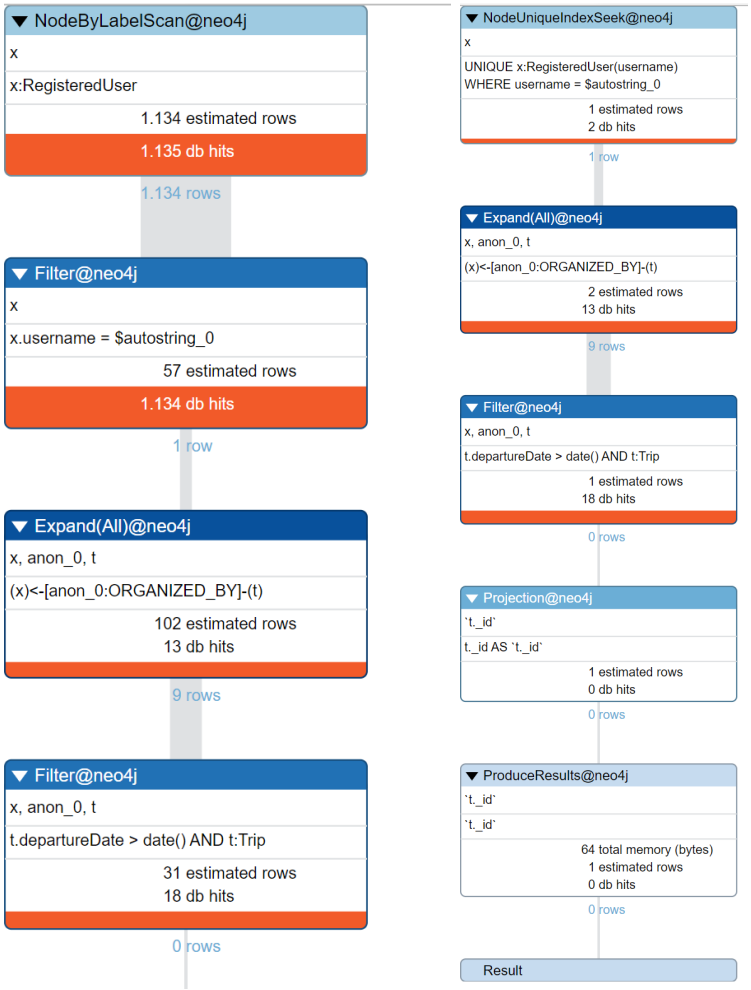


Figure 5.13: Without and with index

- Get trips organized by followers

```
PROFILE MATCH (r1:RegisteredUser{username : "Antonio14"})-
[:FOLLOW]->(r2:RegisteredUser) <-[:ORGANIZED_BY]-(t:Trip)
WHERE t.deleted = FALSE AND t.departureDate > date()
RETURN t._id, t.destination, t.departureDate, t.returnDate,
t.title, t.deleted, r2.username as organizer
ORDER BY t.departureDate SKIP 0 LIMIT 5
```

Figure 5.14: Without and with index

In the image reporting the without stats report, an important stats is missing: to perform the following part of the total query `(r2)<-[anon_1:ORGANIZED_BY] -(t)` and the next one `cache[t.deleted] = false AND cache[t.departureDate] > date() AND t:Trip`, Neo4J perform 5669 and 6449 db hits respectively vs the 2030 and 3695 we find in the same query executed using the index.

- Get suggested trips

```
PROFILE MATCH (r1:RegisteredUser{username : "Antonio14"})-
[:FOLLOW]->(r2:RegisteredUser) -[:JOIN]->(t:Trip)
WHERE t.departureDate > date() AND (NOT (r1)-[:JOIN]->(t))
```

```
AND (NOT (t)-[:ORGANIZED_BY] -> (r1)) AND t.deleted = FALSE
RETURN t._id, t.destination, t.departureDate, t.returnDate,
t.title, t.deleted, r2.username as organizer,
rand() as ord ORDER BY ord LIMIT 5
```

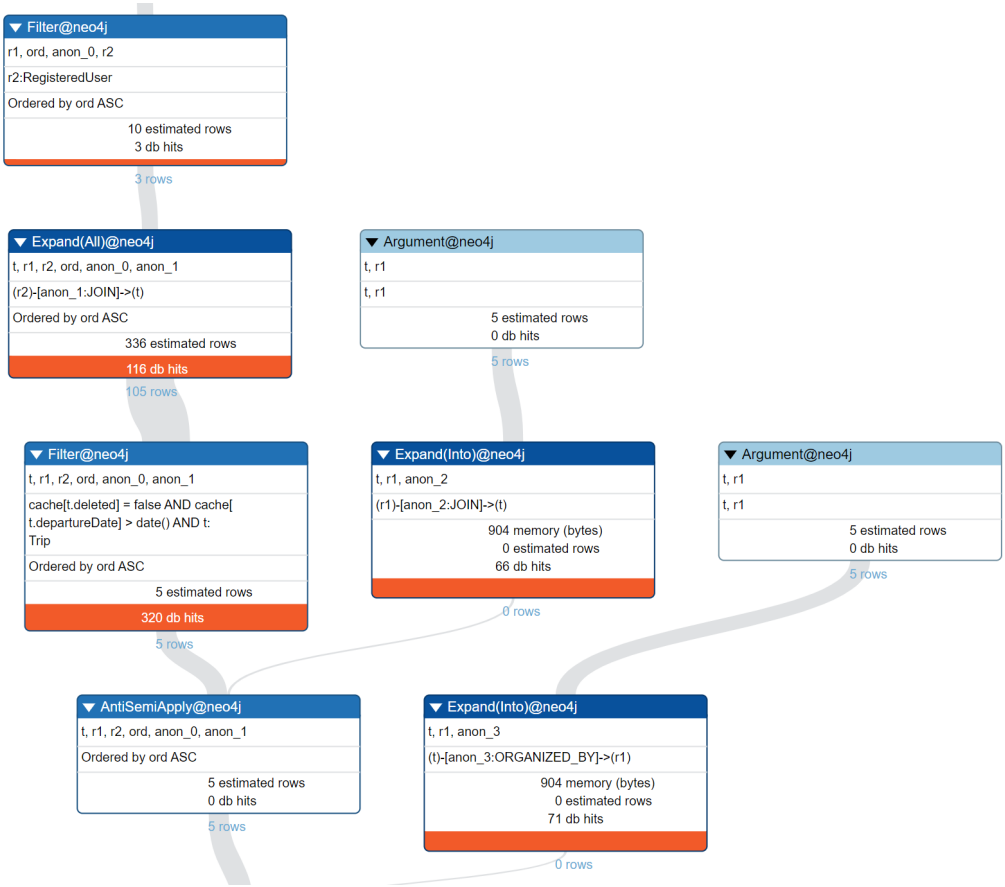


Figure 5.15: With index

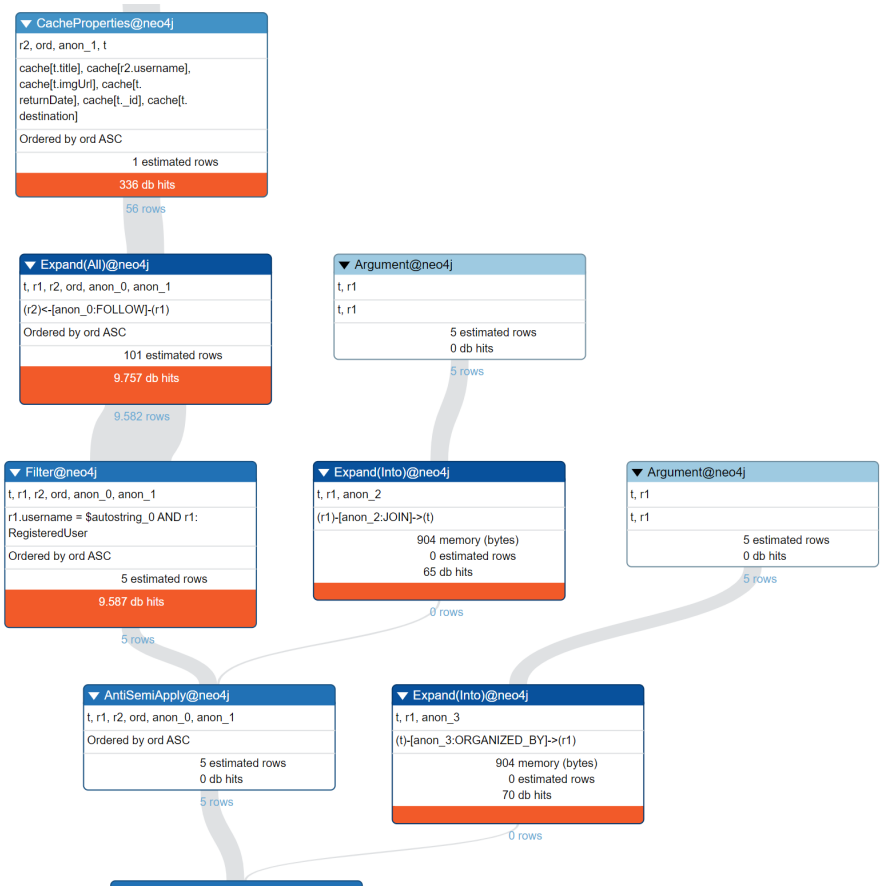


Figure 5.16: Without index

- Get joiners and organizer

```
PROFILE MATCH (r1:RegisteredUser) -[j:JOIN]->
(:Trip {_id :"63b9d588f1bd123a89704b69"}) -
[:ORGANIZED_BY]->(r2:RegisteredUser)
RETURN r1.username, r2.username,j.status
ORDER BY j.status
```

- Get organizer

```
PROFILE MATCH (t:Trip{_id: "63b9d588f1bd123a89704b69"})-
[:ORGANIZED_BY]->(r:RegisteredUser)
RETURN r.username as organizer
```

Since the two previous queries have the same db hits, for the sake of the readability, only the stats about "Get organizer" is reported

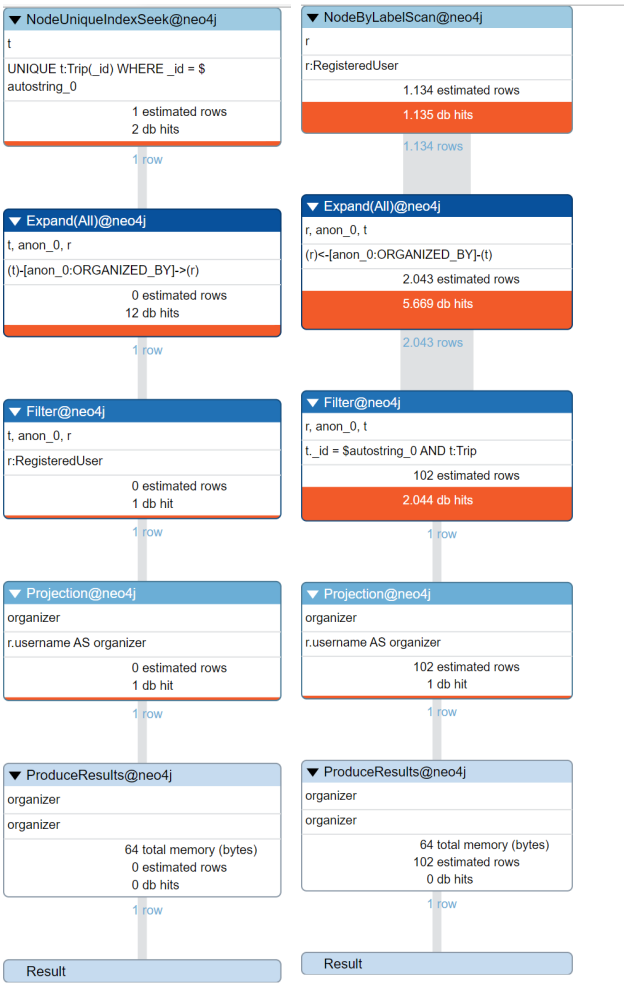


Figure 5.17: With and without index

- Get trips organized by a specific User

```
PROFILE MATCH (t:Trip)-[:ORGANIZED_BY]->(r:RegisteredUser
{username: "Vincenzo0"}) RETURN t._id, t.destination,
t.departureDate, t.returnDate, t.title, t.deleted,
r.username as organizer ORDER BY t.departureDate SKIP 0 LIMIT 8
```

Figure 5.18: With and without index

- Get past trips

```
PROFILE MATCH (r:RegisteredUser{username: "Antonio14"})-
[:JOIN]->(t:Trip)-[:ORGANIZED_BY]->(r1:RegisteredUser)
RETURN t._id, t.destination, t.departureDate, t.returnDate,
t.title, t.deleted, r1.username as organizer
ORDER BY t.departureDate DESC SKIP 0 LIMIT 8
```

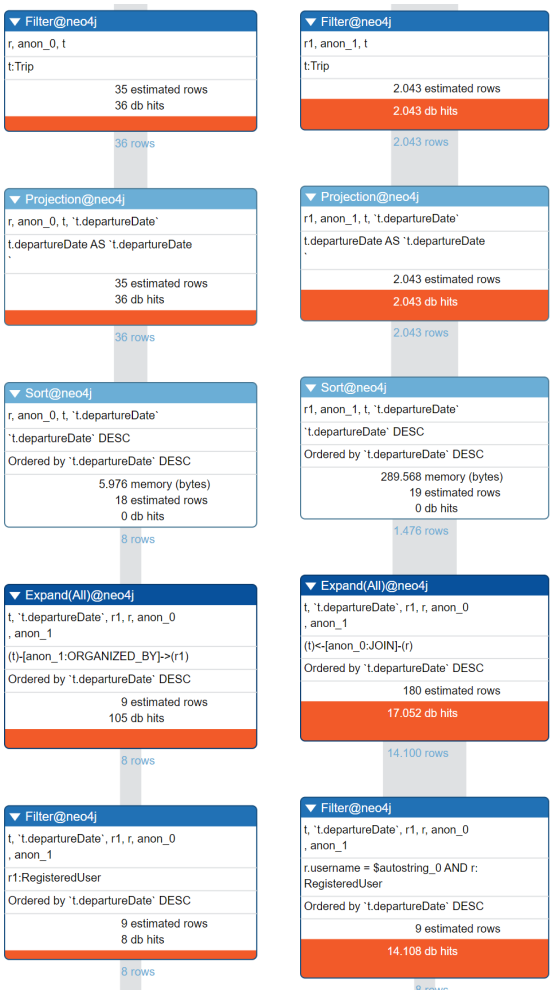


Figure 5.19: With and without index

- Get join status

```
PROFILE MATCH (t:Trip{_id:"63b9d588f1bd123a89704b69"})<-
[j:JOIN]-(r:RegisteredUser{username:"Antonio14"})
RETURN j.status
```

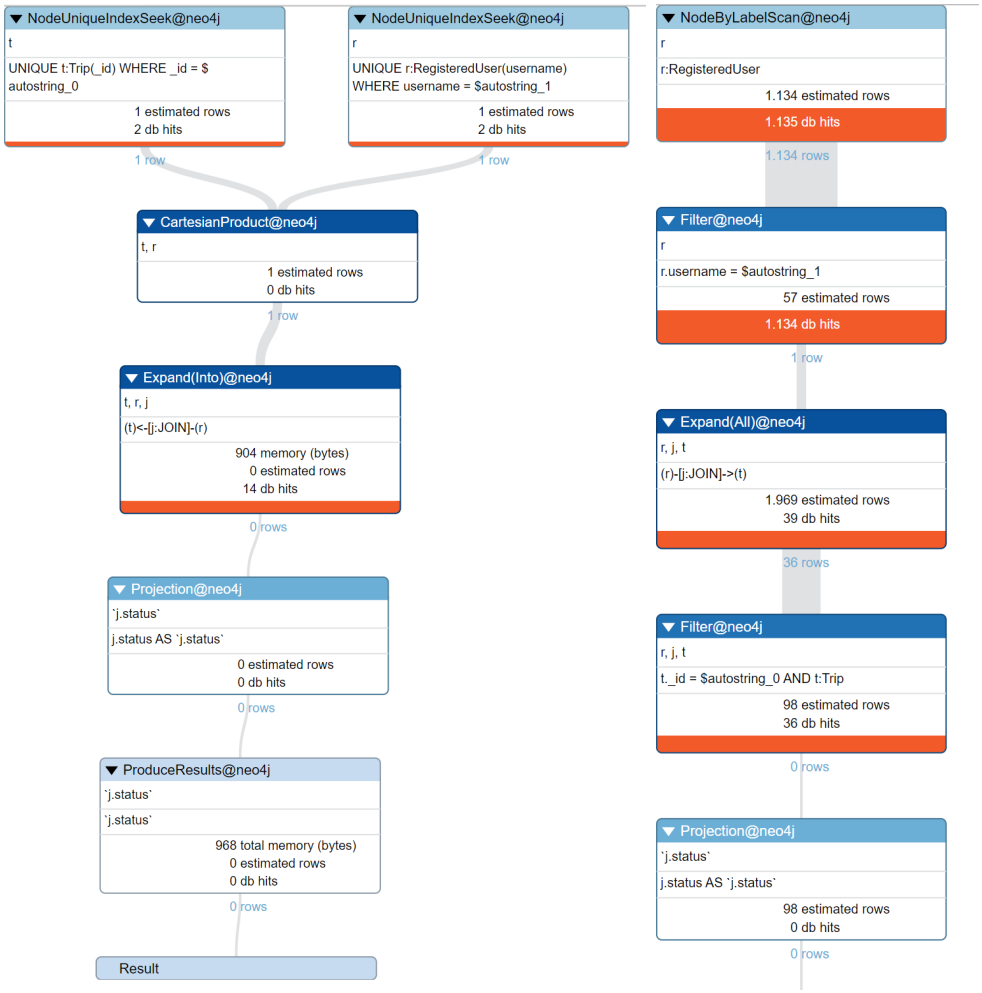


Figure 5.20: With and without index

Chapter 6

User Manual

6.1 Unregistered User

Until the login, the Unregistered User will see at the top of the page this bar:

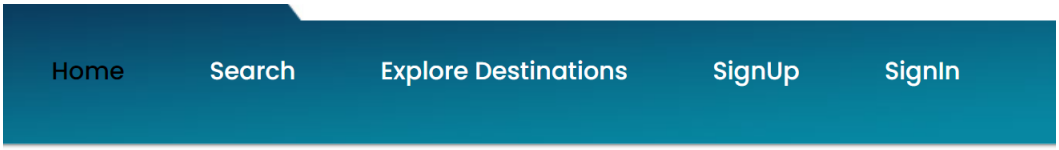


Figure 6.1: Unregistered Navbar

Clicking the buttons will lead the user respectively to:

1. **Home** General Homepage (Section 6.1.1)
2. **Search** Page to search for trips (Section 6.1.4)
3. **Explore Destinations** Page to search for popular destination (Section 6.1.6)
4. **Signup** Signup page (Section 6.1.2)
5. **Login** Login page (Section 6.1.3)

6.1.1 General Homepage

It's the first page that will be show to the user when he/she enter the site and din't logged in yet. Also user will be redirected here when clicking on the website logo.

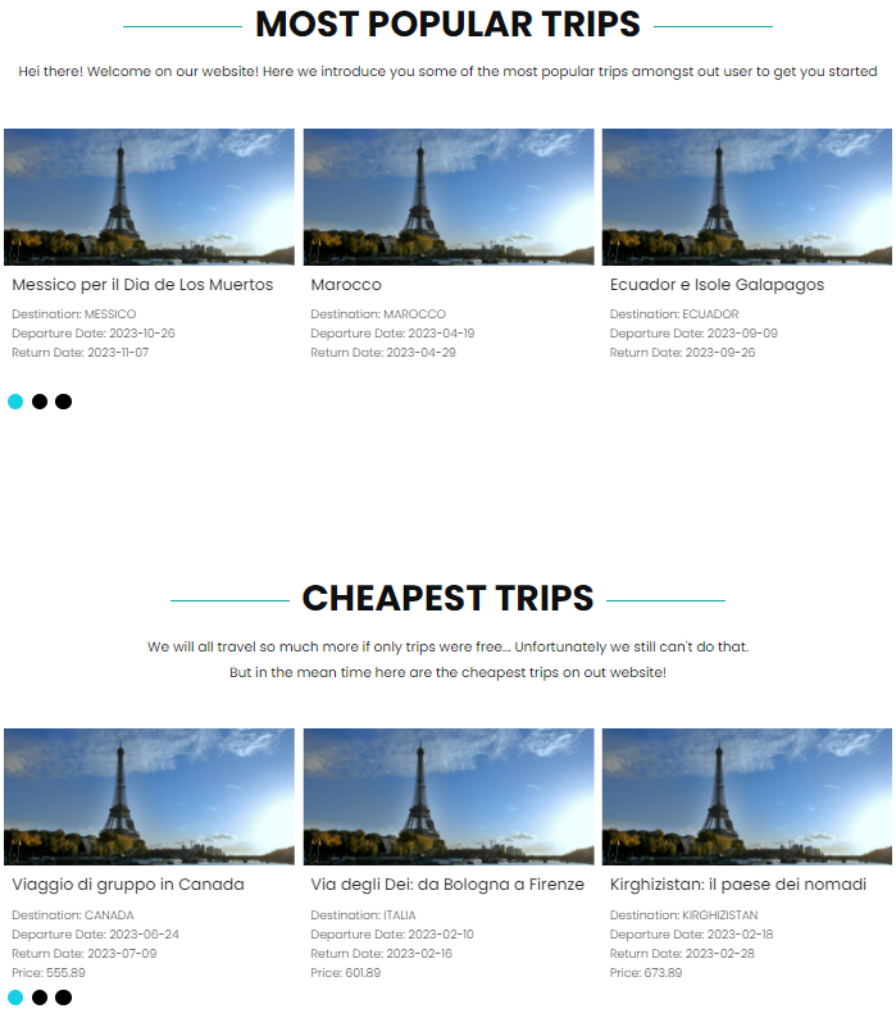


Figure 6.2: General Homepage

6.1.2 SignUp

To signup, the user must click on the related button ("Sign Up") on the navigation bar and then will appear the following registration form.

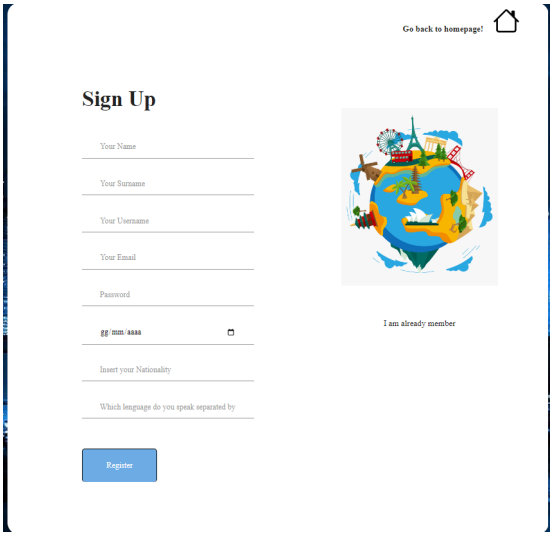


Figure 6.3: Signup Form

When user submits the filled form, his/her own account will be created and then can login the website.

6.1.3 Login

If login is successful, user will be automatically redirected to his/her personal homepage. (See 6.2 for more information).

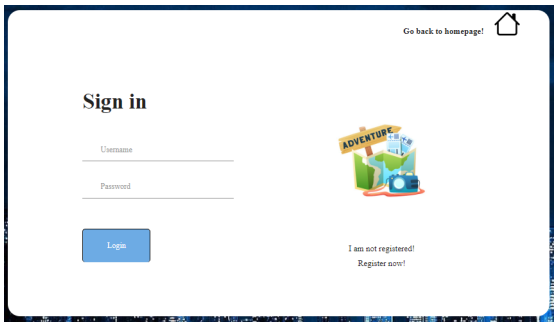


Figure 6.4: Login Form

(See registered user manual for more information)

6.1.4 Search Trips

When clicking on the navbar the user will be redirected to this page:

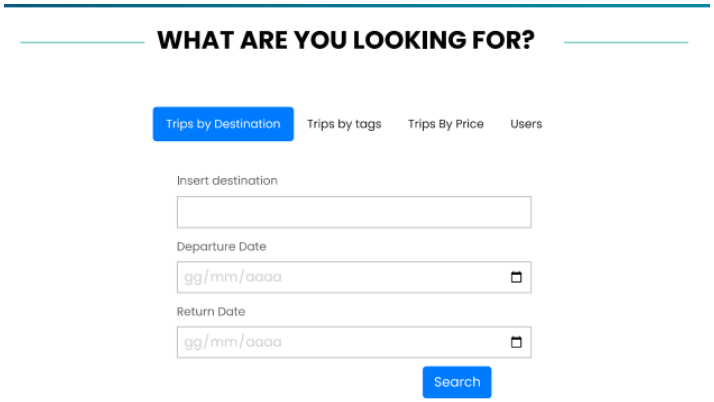


Figure 6.5: Search Trips Form

Using the navigation bar in the search page, users can select the parameters who wants to search by and, for each of them the user can additionally insert the period of time as a further filter:

- destination
- tag
- price

After hitting the "search" button a list of results will be displayed to users as following :

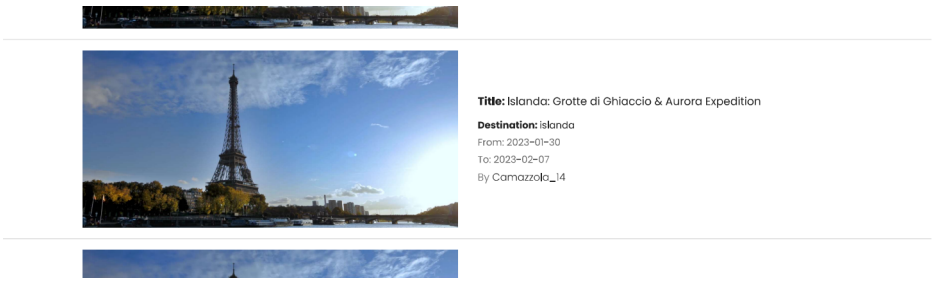


Figure 6.6: Trip Summary

There There are 2 cases that can verify while visualizing the result of searching for trips: the trip could be expired or deleted. In both cases there will be a mark at the top left corner of the post that will display the status of the trip. Here is an example for the expired trip:

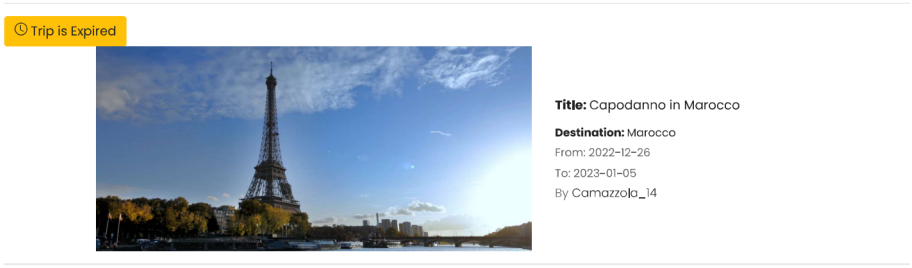


Figure 6.7: Expired Trip

Using the page buttons at the bottom of the bar the user can see more result, moving trough different pages using previous and next. Note that what the user is seeing now is just a summary of the whole information of the single trip, including:

- destination
- title
- representative picture
- departure date
- return
- organizer

6.1.5 Trip Details

By clicking on the title user can visualize more details on the trip:

Figure 6.8: Trip Details

In addition to the information already displayed in the summary, the user will also see a description of the trip, the price, a detailed description of the itinerary day by day, a list of what is included and not included and, if present, some additional information.

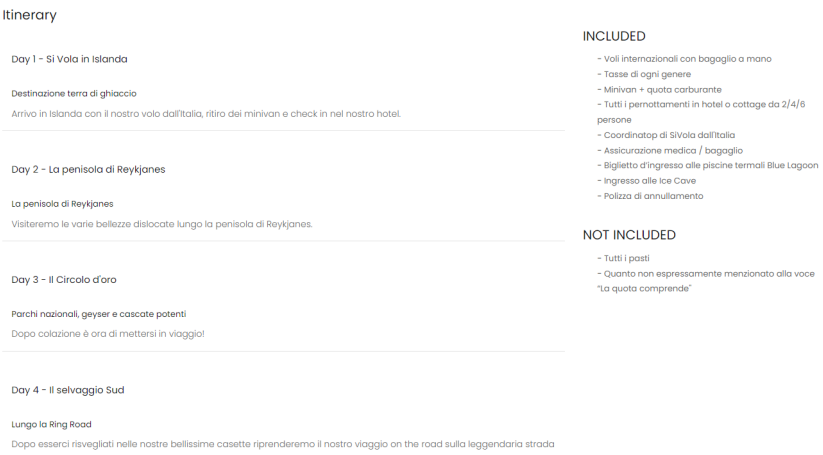


Figure 6.9: Itinerary and Included

6.1.6 Search Popular Destination

Using the related button on the Navbar, users can also search for popular destination filtering by

- Overall: will display the most popular destination in the last year
- Tag: will display the most popular destination for a given tag
- Price: will display the most popular destination for a given price range
- Period: will display the most popular destination for a given date rage
- Exclusive: will display the most exclusive destinations
- Cheaper: will display the cheapest destination

For the *Overall*, *Cheaper on average* and *Most Exclusive* filters, the page will look like these:

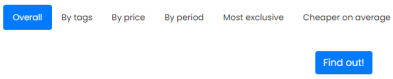


Figure 6.10: Popular Destination Overall Search

For the other parameters the user will have an input form like the one below:

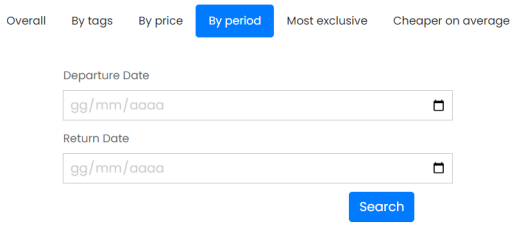


Figure 6.11: Popular Destination by Period Search

After hitting the *Find out / Search* button the results will be shown to the user.

6.1.7 Search Users

In the same page used for searching trips the user can also serach other users of the website by username using the following form

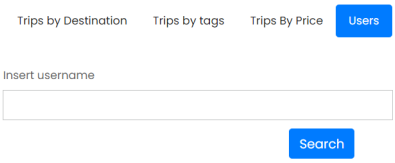


Figure 6.12: User Search Form

By clicking search, the results will be shown in a page like the following one and the user will be able to browse all the results with pagination.

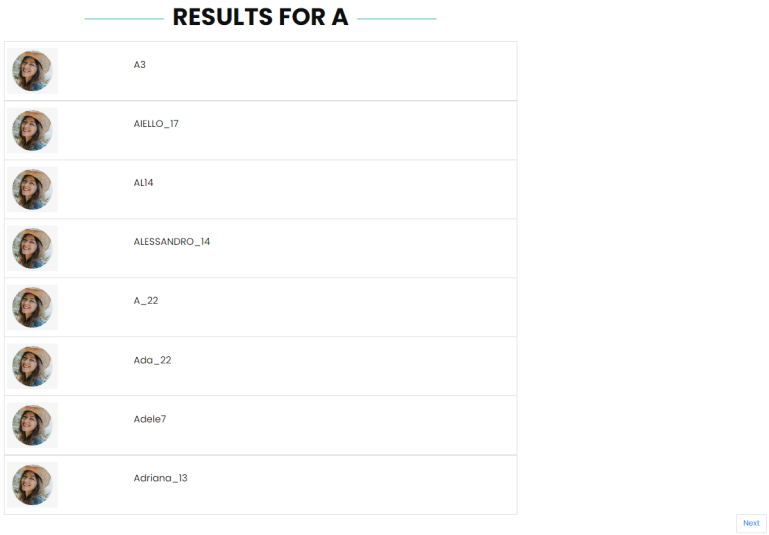


Figure 6.13: User Search Result

6.2 Registered User

Forthermore from the functionalities just explained for the unregistered user the registered user can do some more activities, also the navbar will look slightly different

6.2.1 Registered User Navbar

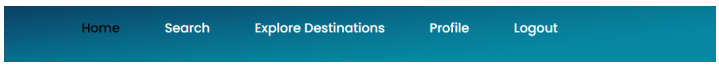


Figure 6.14: Registered User Navbar

6.2.2 Registered User Homepage

Clicking on the home button in the navbar the user will be now redirected to his/her personalized home. The registered user homepage will display personalized recommendation as shown in picture

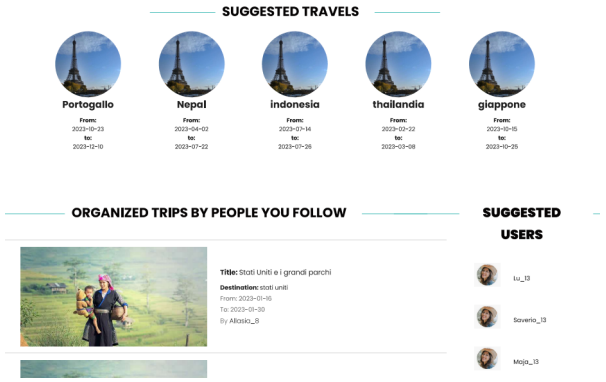


Figure 6.15: Registered User Homepage

In the current page the user will visualize

1. Suggested Trips
2. Suggested user he/she might want to start following
3. trips organized by people he/she follows

Note that since these functionalities are based on followers, if user don't follow anyone, these suggestions will not be displayed.

6.2.3 View own profile

After the login, the user will be redirect to own profile, but can also see it by clicking on the profile button in the navbar:

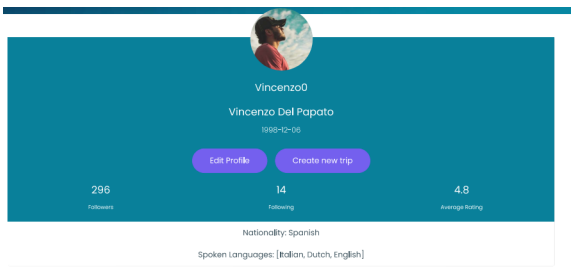


Figure 6.16: Personal Profile Banner

In his/her own profile, user can also edit personal information, create a new trip, display his/her reviews, past trips (i.e trips the user's already joined), organized trips and wishlist.

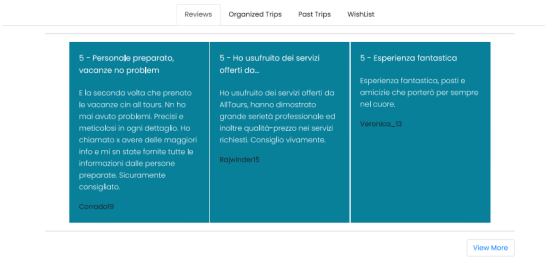


Figure 6.17: Personal Profile Navbar

In this page are displayed only the first 3 reviews: by clicking on "View More" he/she can visualize all reviews and browse them all trough pages. As for the other fields of the bar, if he/she click on it a page like the trip summary page 6.6 will open.

6.2.4 View Other Users Profile

The visualization of other users profile will be slightly different since some information are private.

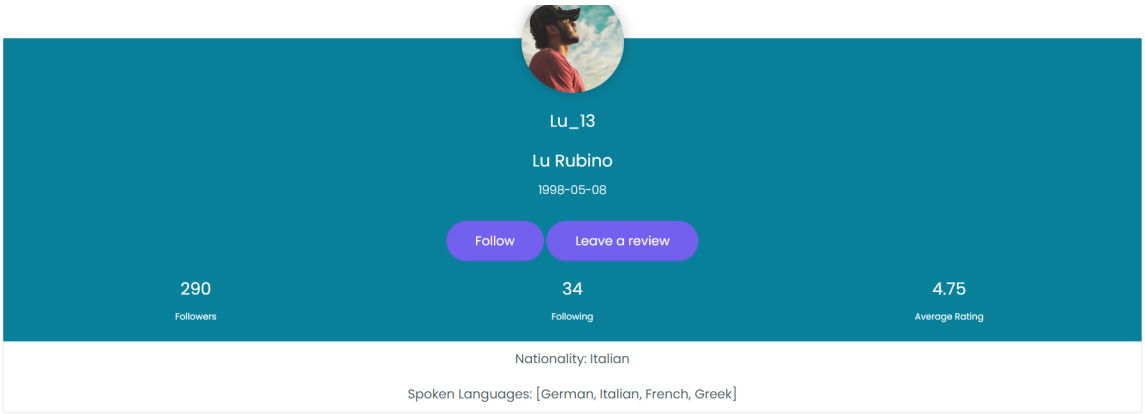


Figure 6.18: Other User Profile

As can be noticed, the buttons are different since a registered user can follow other users, and also leave them a review by clicking on the button.

The navbar is similar and works the same way but past trips and wishlist are not shown since they are private.

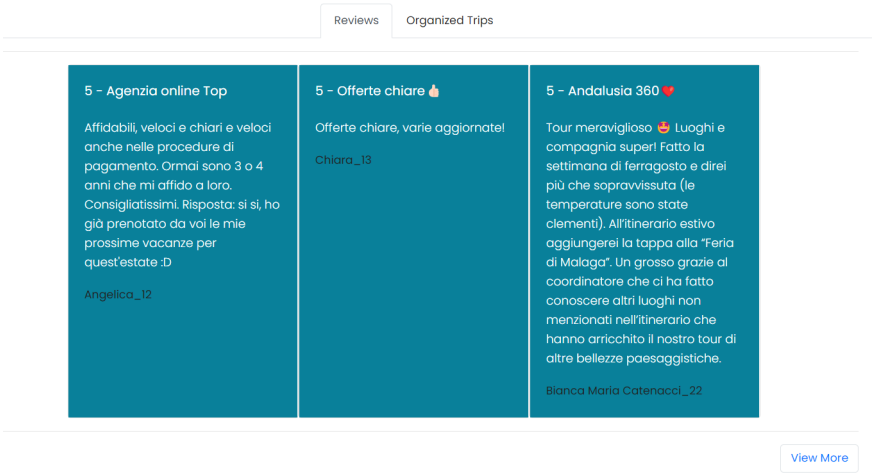


Figure 6.19: Other User NavBar

6.2.5 Edit user Information

By clicking on the related button the user will access this form to modify his/her information.

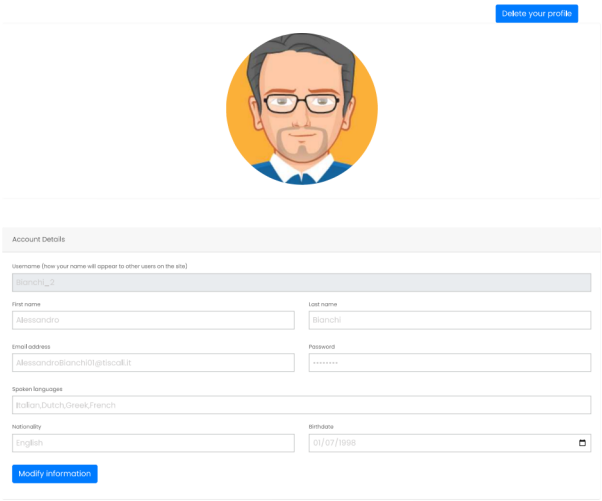


Figure 6.20: Edit Profile Form

6.2.6 Create a Trip

With registration he/she are now allowed to create a new trip, by clicking on the button he/she will be redirected to this page and he/she will be able to "build" the structure for the trip as the user prefer using the add and remove buttons and compiling the input fields.

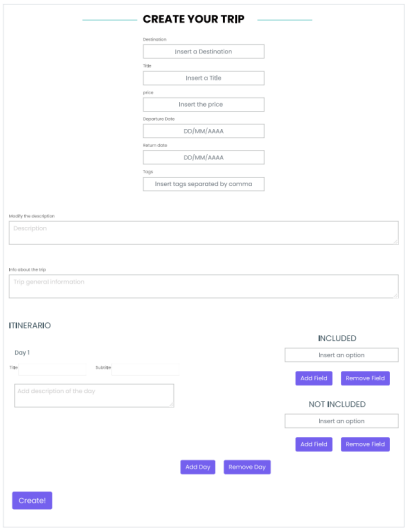


Figure 6.21: Create Trip Form

6.2.7 Edit or delete a Trip

If the user already created a trip, he/she can modify it. If he/she are the owner of a trip while visualizing the trip details 6.8 these buttons will appear:

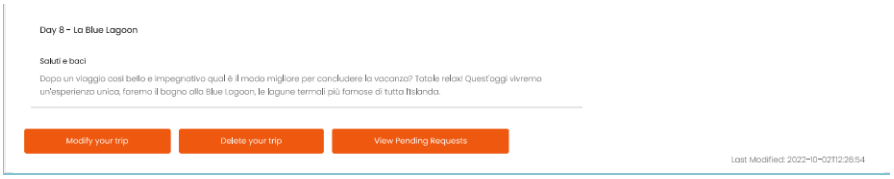


Figure 6.22: Edit Trip Buttons

By clicking on delete the user can delete the trip, and by clicking edit, a page just like the one in the section 6.21 will be opened, but with the form inside the page already filled with the previous data.

6.2.8 Send a join request

A registered user can send a request to join a trip, to do so in the trip details page at the bottom of 6.8 these buttons will appear

Figure 6.23: Join Request Buttons

After submitting a request the user can visualize the status of their request or cancel the request itself.

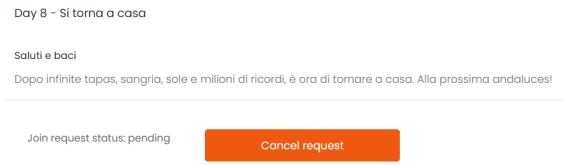


Figure 6.24: Join Status Customer

6.2.9 Visualize the join request

On the other hand the organizer of a trip can visualize the status of the pending request and also the already accepted ones by using the buttons at the end of the trip detail page at section 6.22.
By clicking on the *View pending request* button this page will appear:

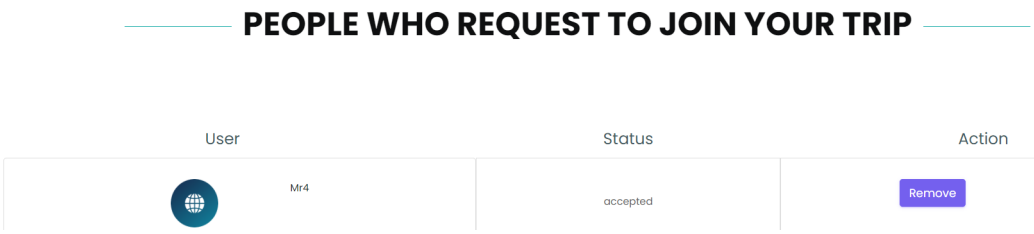


Figure 6.25: Join Request Visualization

For each request the user will visualize these buttons to manage them: he/she can accept or decline a join request or remove an already accepted participant.

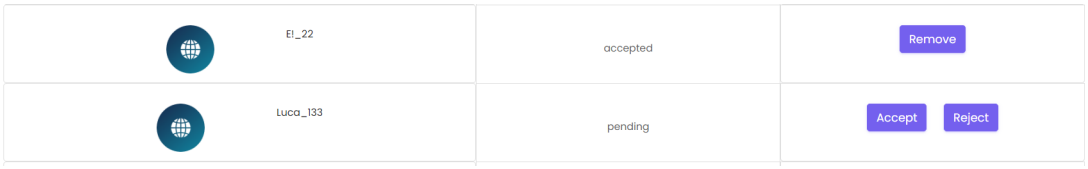


Figure 6.26: Join Request Approval

6.2.10 Add a trip to wishlist or remove it

If the user visualize the trip details 6.8 of a trip he/she didn't organize, on the top of the page, the user will see a star. By clicking on it, the user can add that trip to his/her wishlist or remove it, if the trip was already in the user wishlist.

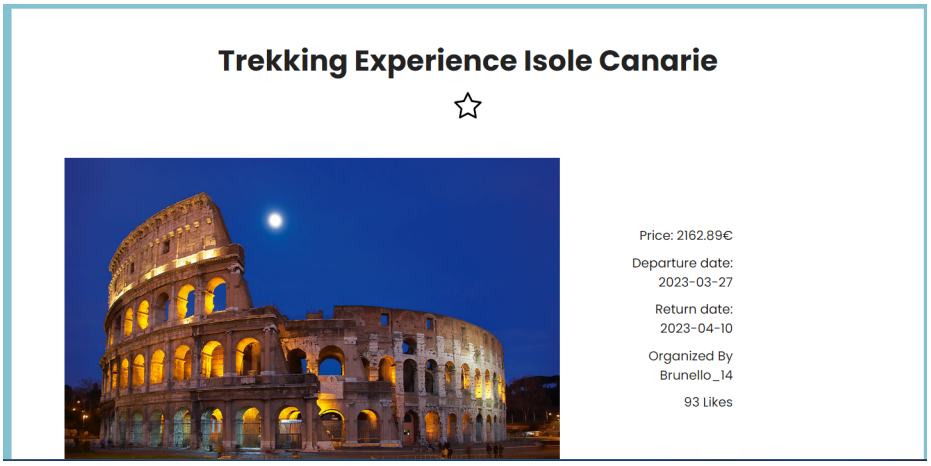


Figure 6.27: Wishlist Star

6.2.11 Write a review

If the user click on the related button placed into the user profile page, the following page will appear.

Figure 6.28: Write a Review Form

6.3 Admin

The admin isn't a normal user, he has different functionalities mainly connected to controlling the users and other admin.

The navbar is different and looks like this:

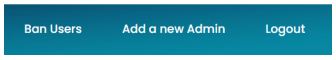


Figure 6.29: Admin Navbar

The login form is the same for a normal user but after the login succeded the admin will be redirected to another page.

6.3.1 Visualize admin information

The admin information are the first page shown after the login.

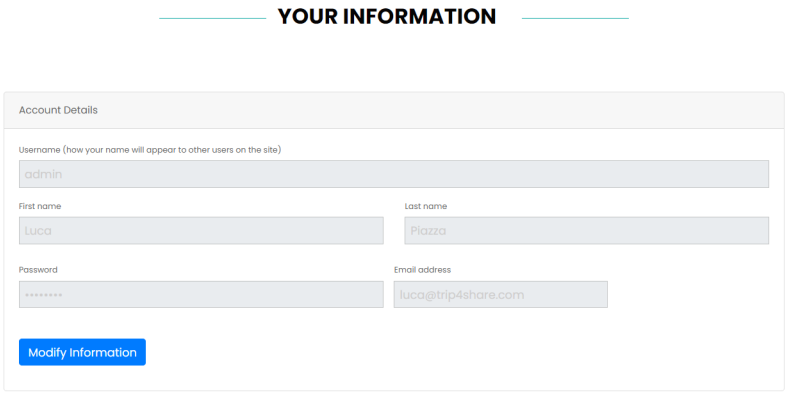


Figure 6.30: Admin Homepage

By clicking on *Modify the user profile* the fields will become clickable.

6.3.2 Add another admin

The admin can also add another admin, by clicking on the related button on the navbar this form will appear:

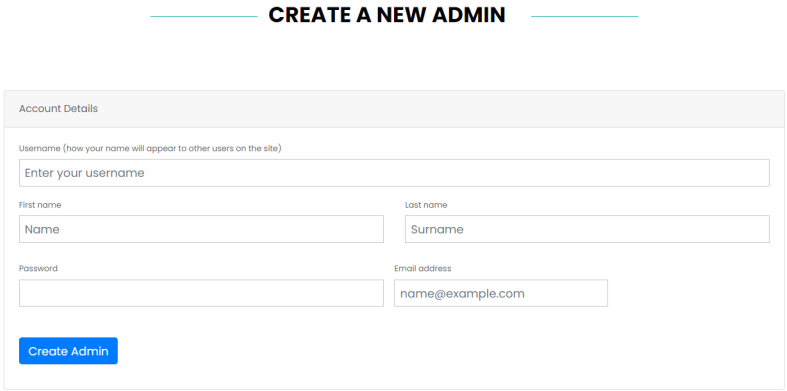


Figure 6.31: Admin Creation Form

6.3.3 Ban Users and Admin

The admin can ban other users and admin, a search bar will appear:

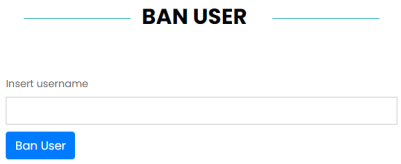


Figure 6.32: User Ban Form
After inserting the username, the admin can ban the user just clicking the button.

Chapter 7

Future Implementation

The current architecture made us think about possible future implementation without changing the architecture itself or by making small adjustments that doesn't impact the whole system.
Such developments cuold be:

- Show travel-mates to the user when the trip details are shown.
- Perform suggestion based on the Trip currently add to the user Wish-list.
- Temporary trips: users can create trip available for a limited period of time.