HOMEPASS

Theodoros Mandilaras
Team: **Μανδηλαράς**
Cs2.190018
2019-2020
An AirBnB-like application that was created for the purposes of the e-commerce technologies course.

Content

Introduction

Homepass is an AirBnB-like application created only by me (Theodoros Mandilaras) for the purposes of the eCommerce Technologies course of the MSc in Computer Science.

In this application, users can search available places to stay for their holidays or their trips, check images of a place with a lot of other information, read comments, communicate with the host and make reservations. Moreover, they can enable and switch as a **host** in order to add their own place for rent in the days that it could be available. They also can edit it or even delete it. So, the application contains two modes. One mode that the user can act like a ***GUEST***, and a mode that the user can act as a ***HOST***. Finally, users that stayed in a place can rate it by providing a comment and a degree from 1 to 5.

The users, firstly should be registered in the application by providing the requested information and then, after the backend server accepts the registration, they can login in the application. The users can also edit their profile information or upload a picture. In the homepage, the users in the beginning will see the most famous places sorted to the cost per person. After some searches the homepage view will contain places that the user had searched before.

Homepass is the official name of the application. Despite the fact that the project folders are called MyBnb (which was my initial name).

The delivered compressed file, contains two main folders:

- **myBnb:** Which contains the implementation of the backend and the file-based database (as we are going to see later)
- **frontMynBnb:** Which contains the implementation of the frontend part which is an android application.

Some data in the database have been left (with also some images in the storage), so the application won't be empty at the testing time.

All the stylistic decisions were made and implemented by me, and the background images are taken from a game called: **Firewatch**.

Development Details

The application follows the RESTful architecture, having a frontend application which communicates with the services provided by the backend server using HTTP(S) requests that contain the JSON objects. The backend's services create, read, update and delete entities from a database following the CRUD model.

Front-End

The front-end is an Android application and it is created in the **Android Studio**.

- It is compiled with SDK version 29,
- The target SDK version is 29
- And the minimum SDK version is 26

So, the application targets devices with Android 10 versions.

All the communications with the backend's services are achieved using the Retrofit tool and the conversation from JSONs to other objects or the opposite, is achieved with the gson tool.

Back-End

The back-end is implemented with the **Spring Boot Framework**, which is a Java framework. The code was written in the IntelliJ IDEA.

All the communication with the backend's services are encrypted with SSL(TLS) protocol, so all the requests on port 8443 are using the HTTPS protocol. That was achieved by generating a self-sign certification using the bellow *keytool* command:

```
keytool -genkeypair -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore mkyong.p12 -validity 365
```

The self-signed certification is located in the resources file called server.p12. Moreover, the backend has been set to accept multipart files having maximum size at 128MB in order to acceptes big images!

Database

At the beginning, the application was developed to have all its data in a **MySQL** database which was running on my machine. However, for making easier and faster deployment, the database was switched to a H2 database.

H2 is a relational database management system which in most of the cases is used as a memory based database. However, I am using it as a **file based** in order to maintain the data in multiple executions. So the database is stored in a file called ***bnb_db.mv.db*** which is located in the backend project main directory.

Server

As server I use the default server that is set by the IntelliJ IDEA (Ultimate Version) which is a **TomcatWebServer.**

Application Video & Screenshot Samples

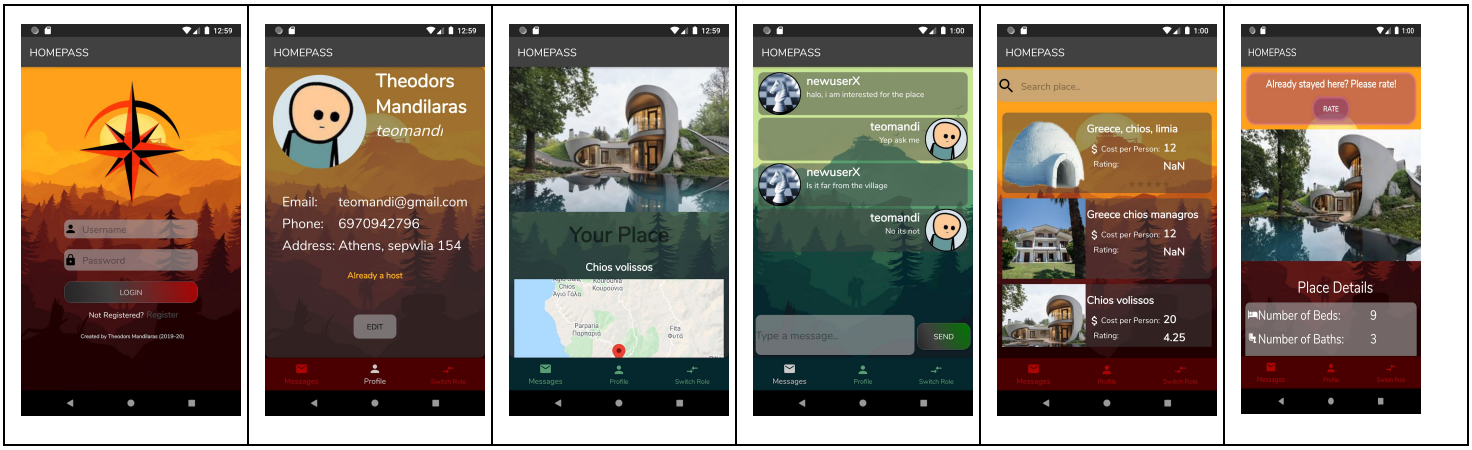
I have recorded some video samples with the main features of the application usage as an example of the user experience. Those video samples can be found in the below google drive url link:

https://drive.google.com/drive/folders/1VYS3XKicxrZFWLkWkRYI9qf38LxmfyRg?usp=sharing

The content of the videos is described below:

- 1_register_editprofile → contains Creation of a new user, basic exploration of the application and profile editing.
- 2_enabling_host → shows how to enable the **HOST** role.
- 3_checkingplace_message → checking places in details and sending the first message.
- 4_hostview_editplace → switching as a **HOST**, and editing my place.
- 5_messaging_with_host → messages between a **HOST** and a **GUEST.**
- 6_newuser_newplace → created a new place for the new user.
- 7_search_reservation_rating → searching for a place, making a reservation, and after that rating it. (The dates are selected in the past, so the rate feature is enabled).
- 8_search → more search.

Also, there is a directory with some screenshots.



I will try to explain in detail all the implementation decisions that I took for those three components in the below chapters.

Database - H2

The H2 database is configured in the *application.properties* file, which is located in the *resources* directory. In the follow lines, we can see how it is configured:

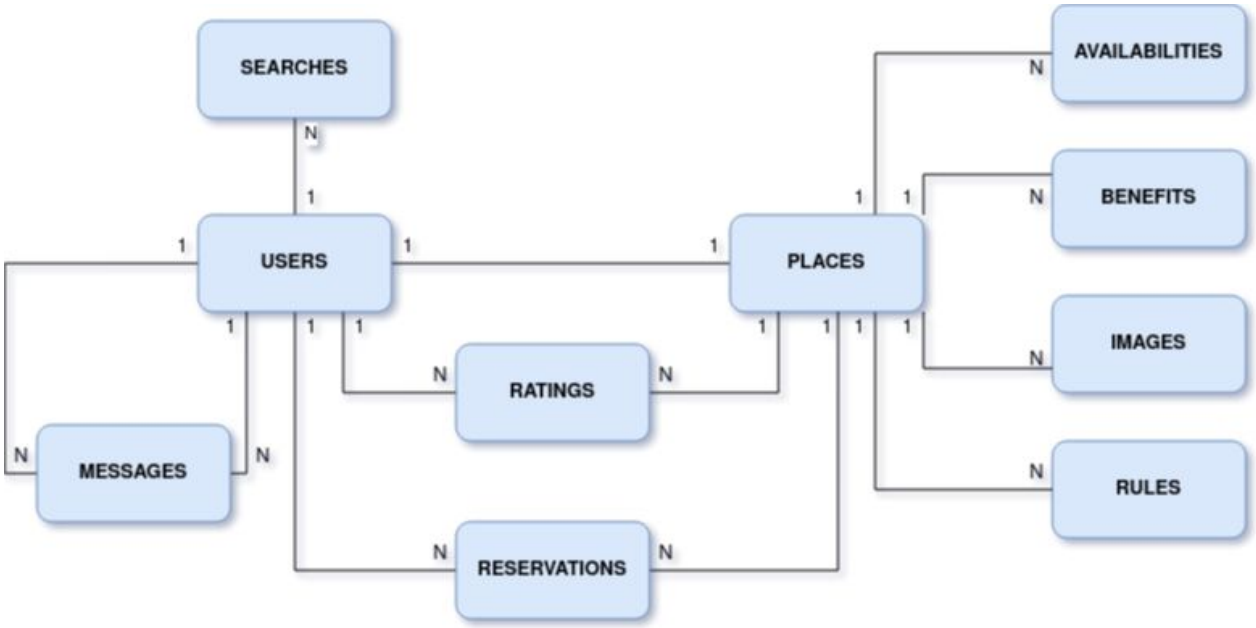
```
spring.datasource.url=jdbc:h2:file://home/teomandi/DI-M-Sc/eCommerceTechnologies/myBnb/bnb_db
spring.datasource.driverClassName=org.h2.Driver
spring.h2.console.enabled=true
spring.datasource.username=cs2180019
spring.datasource.password=cs2180019
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

In the first line we can see the path for the file where the database is located. **It is very important to set the path correctly in order the application could find the file and use the Database otherwise it will not be able to run!!!**

In the following lines, I set the Driver, I enable the h2-console in order to have a better view of my database, I set the credentials of the database (my registration number have been misspelled here), and lastly I set the JPA.

Schema

Now, let's see the database schema that I came up with, in order to fulfil all the requested requirements of the application.
The below figure, shows the database schema that the application uses:



Tables and Attributes

All the tables will be explained with their fields one by one in the following lines.

Users

This table contains all the information related with the users. Those selected information are:

- id (primary key)
- Username (which is unique)
- Password
- Email
- First name
- Last name
- Phone
- isHost (if he has enabled the host role)

- imageName (The name of the image only in order to find the image in the storeage)
- Address
- Place_id (links to the user's place)
- Ratings (the ratings the user made) (id)
- sendedMessages (the messages that he sent) (id)
- ReceivedMessages (the messages the he received) (id)
- MessagesAsHost (the messages that he is the host) (id)
- Searches (the searches that the user made) (id)

A user can be the owner of only **one** place, so the relationship is OneToOne, as it can be seen in the schema above.

Places

This table contains all the information related with the places. Those selected information are:

- id (primary key)
- User (owner of the place) (id)
- Main Image (the filename of the main image (string)): main image is the image that is previewed in the home view of the application.
- Address
- Geographic coordinates
    - Latitude (double)
    - Longitude (double)
- Max guest (the maximum number of guest a place can have according to the owner)
- Min Cost (in order to be rent)
- Cost per person
- Type (Room or house)
- Description
- Beds (number of beds)
- Baths (number of baths)
- Bedrooms (number of bedrooms)
- livingRoom (boolean: if it contains a living room)
- Area (how many square meter the place is)
- Benefits (the benefits that the place provides) (id)
- Rules (the rules that are set by the owner) (id)
- Availabilities (the date ranges that the place is available) (id)
- Ratings (Degree and comments from users that already stayed there) (id)
- Images (filenames of the images of the place) (id)

- Reservations (that the place have been in)(id)

Availabilities

An availability describes the date range at which a place can be available for a reservation. A place can contain multiple availabilities (OneToMany relation). The Availability entity contains:

- Id
- From (starting of the range) (date)
- To (ending of the range) (date)
- Place_id (the id of the place)

Benefits

Benefits are the features that a place provides according to the owner. A benefit can be free wifi or free parking. A place can have multiple benefits. I describe the benefit entity with the:

- Id
- Content (string)
- Place id

Rules

Rules are the rules that the users should follow as they stay in the place, according to the owner of that place. A place can have multiple rules. One rule can be: No parties are allowed. I describe the rules just like the benefits:

- Id
- Content (string)
- Place id

Images

When a user creates a place, he can upload multiple images to show the apartment. Those images after they get stored in the storage of the application, they have their filename stored in the database in the Images table. This table contains only the name of the file.

- Id
- Filename (string)
- Place id

Ratings

After a user stays in place, the frontend allows the user to rate this place. A rating for a place is described by:

- Id
- Degree (a float number that can be between 0 to 5)
- Comment (a comment that user had write)
- User_id
- Place_id

A user can have multiple ratings for different places.
A place can have multiple ratings for different users.

Reservations

The reservations table contains the information of the reservations all users did in which places and in which date ranges. Those things are described with:

- Id
- Start (starting date)
- End (ending date)
- User_id
- Place_id

Searches

Searches table contain the information about the searches a user made in order to suggest him new places. As we are going to see, each time a user makes a search request, the backend looks for places in a radius of the lat and long point. So, in each search I keep the location that the user is looking for a place. So a search is described with the:

- Id
- Geographic coordinates
    - Latitude (double)
    - Longitude (double)
- User_id (the one that made the search)

Messages

For the messages, in order to describe a message, I have a string that contains the content of the message. Also, I have two different columns that keep the sender user and the receiver user of a message. In normal cases those fields would be enough. However, the messages we see should be filtered, so in **GUEST** mode the user should

be able to see only messages from other users that are not referring to him as a host, and similarly in the **HOST** mode the user should see messages that referred to him as a host of a place. So, I had to store in each message who from the two users that are taking part is the host. So the table looks like this:

- Id
- Text (content of the message)
- Sender_id (the user that send the message)
- Receiver_id (the user the received the message)
- Hoster_id (the user between the two who is the host)

Many of the entities implement the **AudiModel,** that model adds in those tables the columns *created*At and *updated*At. Those columns contain the timestamps of the events that they describe.

BackEnd - Spring Boot

As I already have mentioned, the backend of the application has been implemented with the Spring Boot framework, which is a Java framework. The code for the backend is located in the directory called: *myBnb*

In the following chapters I will try to describe the main details and the programming decisions I took, designing the backend, to fulfill the project's requirements.

Directories Structure

All the implementation of the backend is located in the path: MyBnb/src/main/java/com.exercise.mybnb/ . Before that path the most important files or directories are:

- MyBnbApplication,java: This is the main file that starts the whole application.
- Resources: in this directory the *application.properties* file exists, which contains all the configuration of the application, and the generated certification named *server.p12*.
- Storage: in this directory, as we are going to see and below, is the location where all the media of the application are stored.

- Pom.xml: A maven file that contains all the dependencies of the application.
- Bnb.db: The file that contains the database of the application.

In the above mentioned path we will find the directories with:

- Controllers: In this directory are located all the controllers for the application.
- Exceptions: Some exceptions to handle unique cases in the application.
- Models: In this directory are implemented all the entities that are described in the database section, and they are represented with an Object class.
- Repositories: That directory has an interface for each model, which gave us the ability to make useful queries in the database easily.
- Response: some custom responses.
- Security: Contains all the security scripts for the application's requirements.
- Utils: Contains scripts with functionalities that are repeated and not belong to any model.

Now we are going to go deep inside to the most important directories and describe their functionalities.

Security

HTTPS - TLS

As I already mentioned the application responses only in HTTPS requests on the port 8443. If a request is not with the HTTPS protocol, then the request is redirected in with the correct protocol. All that configuration is written in the *MyBnbApplication.java* file.

Authentication - Authorization

The authentication and authorization is implemented with the JWT method. All the requests should be authorized with a given web-token. The token is taken by the users when they login in the application. The only request that it is allowed to pass without authorization is the registration of the users.

The files in the Security directory follow the same pattern as we learned in the course.

Model

In the model directory are located all the entities that are described in the database section. Each entity that points in a database table, has its own java class with the described fields and the necessary functionalities to handle them. The fields have been

declared with the correct types and annotations to serve the database schema that was shown in the figure.

Those models are:

- Availability
- Benefit
- Image
- Message
- Place
- Rating
- Reservation
- Rule
- Search
- User

And the AudiModel interface from which some the other models inherit the CreatedAt and UpdatedAt fields.

Repository

For each model a repository exists having for name: <model>Repo. Each repository is an Interface that extends the **JpaRepository**, which provides the interface with some functionalities in order to fetch and edit entities of the model. As we are going to see, in those repositories have been implemented some NamedQueries for functionalities that are not already inherited.

Controller

The controllers are the place where all the magic happens. Just like with repositories, each model has a controller in which the main functionalities are implemented to handle the resources according to the CRUD architecture. That is achieved with the help of the repositories of the models. The resource handle is done by requests in certain paths that triggers some functionalities called endpoints. All the controllers are REST controllers and the communication with the clients is done with JSONs. I am going to describe the most important endpoints in detail in the later chapter.

Exception

I have implemented three different exceptions which are thrown when there is an unexpected event in the run time of the application. All of those three exceptions

extend the RunTimeException and the goal is to respond to the client with a desired error message suitable to the error. Those exceptions are:

- ActionNotAllowed: When the client tries to make an action that is not allowed. For instance to delete a place that does not belong to him.
- ResourceNotFound: When we look for a resource that does not exist in the database.
- ResourseAlreadyExists: Mainly for the case in which the user chooses a username that already exists.

Storage

In the storage directory are stored all the images that the users upload. At the first run of the application, the storage folder is created with two folders inside. One for the users and one for the places. The users can have only one image. On the other side, a place can have multiple images, so each place has its one folder named by its unique id, where all the images are located.

This storage directory tree is generated in the *initApp()* in the *MybnbApplication.java* file if it does not already exist.

Endpoints

In this chapter I will describe the most important endpoints (with the request type and the route) for the application for each controller.
As endpoints I refer to all the services that the backend of the application provides, and they can be triggered by making requests to paths with the correct method. Those services can create, read, update and delete items from the database.

Now I will describe the main endpoint of the application, for each controller.

In each of the below cases:

- If a resource does not exist a *ResourceNotFound* exception is thrown.
- If an action is not allowed a *ActionNotAllowed* exception is thrown.
- If trying to create a duplicate of a unique field a *ResourceAlreadyExists* exception is thrown.

UserController - Create, Read, Update, Delete etc Users

- GET - /users/{id}: finds and returns the user by his id.

- GET - /users/{username}/username: finds and returns the user by his username which is unique for each user.
- GET - /users/{id}/image: Finds the user by his id and returns his image as a byte array. The images of the users are stored in the ‘users’ directory in the storage. However, the controller finds the image name from the database and parses the bytes with a function from the *Utils*.
- PUT - /users/{id}/host: Finds the user by his id and sets the host field to the opposite of the current value. Using this endpoint a user can enable the host role.
- POST - /users (register): Gets in the request body a user object, checks if the username does not exist, and stores it in the database. It also gets as multipart an Image, which it will be stored in the storage. If no image was provided, then the default image will be used. The default is shown below. It is important to mention that the user’s passwords are encrypted with a **bCrypt** encoder before they get stored in the database. The **bCrypt** encoder is initialized at the execution of the application.
- PUT - /users/{id}: Finds the user for the id in the path, also in the body of the response gets a User object, which will update the current user with the requested id. First, it checks the username if it already exists, if it does not, updates the entity with the received new user object. It also replaces the image if a new image was provided. However, it cannot update the password!
- DELETE - /users/{id}: Finds the user by his id, and deletes that entity.

Default user image.

PlaceController - Create, Read, Update, Delete, Search etc Places

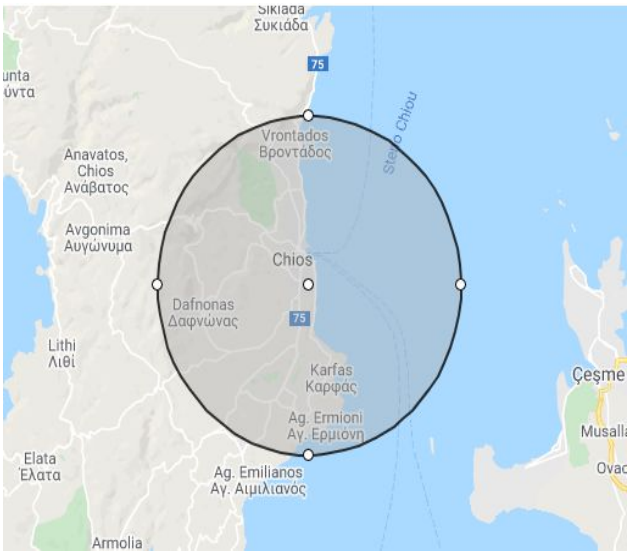
- GET - /places: Returns a page with places according to the request. Each page contains, by default, 10 places, and they are sorted with the *costPerPerson* field. The number of the current page is given in the request params. This endpoint is

used for the fresh users who don't have any search history so the algorithm has no data for suggestions.

- GET - /places/{pid}: Finds a place by its id, and returns it.
- GET - /users/{uid}/places: Finds the user by the given id, and returns its place, if he has already created it.
- GET - /users/username/{username}/places: Just like the one before, with the addition that finds the user by his username.
- POST - /users/{uid}/places: Firstly, it finds the user from the given id, then checks if the user already has a place. If he does an ActionNotAllowed exception is thrown. If he does not, it stores in the database the received place object from the body of the request, sets it to the target user, and if a main image is sent, creates a directory for the place in the storage and stores the image there.
- PUT - /users/{uid}/places/{pid}: The endpoint first validates that the user with the given id has the place with the given id, and if the validation is ok, updates all the fields of the given place from the request body, to the entity in the database. Also, if an image for the place is provided, then deletes the old (main) image and replaces it.
- DELETE - /users/{uid}/places/{pid}: Finds the user with the given id, then finds the place with the given id, checks if the place belongs to the user (validation), and if everything is ok, it firstly deletes all the related items with the place (ratings, searches, availabilities), then set the user without any place, and finally deletes the place.
- GET - /places/{pid}/mainimage: finds the place by the given id, and then builds an ok response entity having as body the bytes of the main image of the place.

Search Engine

- GET - /search: This endpoint implements the **search engine** of the application. It gets as parameters:
    - For the paging control, the page size and the requested page,
    - For the place search the: type of the place the user is looking for, the date range (‘from’, ‘to’) that the user is interested in, the number of the people, and the latitude and longitude of the area that the user typed.
    - Also gets the id of the user.



At first the request is stored as a search entity for future suggestions. Then the bellow process takes part:

1. The endpoint using a *named query* from the *placeRepo*, finds all the places that are 'close' in the geographic location that the user provided sorted by the costPerPerson field. As 'close', I define all the places in a radius of 0.3 degrees of the given geographic location. That corresponds to an area with a radius of 33 km.
2. Then, for each one of those places, the algorithm calculates if the place's minimum cost field is accepted by the number of the people that are interested to stay multiplied by the costPerPerson. So the condition is:

   $$minCost > num * costPerPerson$$

3. For the places that passed that condition, the endpoint checks the availabilities of the place. If the 'from' date is after or equal to the start an availability and the 'to' before or equal to the end, then the place is accepted!
4. All the accepted places are inserted in a list with the accepted places.
5. In the end, the accepted places should be sent in the group of 'pageSize' for implementing the paging. So according to the given 'pageNo' a sublist is cut and returned as a response to the user.

That was the algorithm for searching places.

AvailabilityController - Create, Read, Update, Delete etc Availabilities

- GET - /availabilities: Returns all the availabilities of the database.
- GET - /places/{pid}/availabilities: Finds the place from the given id and returns all its availabilities.
- POST - /places/{pid}/availabilities: Finds the place from the given id, and stores for that place the availability provided from the body of the request.
- POST - /places/{pid}/multi-availabilities: Finds the place from the given id, and stores a list of availabilities provided from the body of the request for that place. However, I was never able to make a request like this to work from the client.

- PUT - /availabilities/{aid}: Finds the availability with the given id, and updates its content with the given availability in the request body.
- DELETE - /availabilities/{aid}: Finds the availability with the given id, and deletes it.

BenefitController - Create, Read, Update, Delete etc Benefits

- GET - /benefits: Returns all the benefits of the database.
- GET - /places/{pid}/benefits: Finds the place from the given id and returns all its benefits.
- POST - /places/{pid}/benefits: Finds the place from the given id, and stores for that place the benefit provided from the body of the request.
- POST - /places/{pid}/multi-benefits: Finds the place from the given id, and stores a list of benefits provided from the body of the request for that place. However, I was never able to make a request like this to work from the client.
- DELETE - /places/{pid}/benefits: Finds the place from the given id, and deletes all the benefits that it has.
- PUT - /benefits/{aid}: Finds the benefit with the given id, and updates its content with the given benefit in the request body.
- DELETE - /benefits/{aid}: Finds the benefit with the given id, and deletes it.

RuleController - Create, Read, Update, Delete etc Rules

- GET - /rules: Returns all the benefits of the database.
- GET - /places/{pid}/rules: Finds the place from the given id and returns all its rules.
- POST - /places/{pid}/rules: Finds the place from the given id, and stores for that place the rule provided from the body of the request.
- POST - /places/{pid}/multi-rules: Finds the place from the given id, and stores a list of rules provided from the body of the request for that place. However, I was never able to make a request like this to work from the client.
- DELETE - /places/{pid}/rules: Finds the place from the given id, and deletes all the rules that it has.
- PUT - /rules/{aid}: Finds the rule with the given id, and updates its content with the given rule in the request body.
- DELETE - /rules/{aid}: Finds the rule with the given id, and deletes it.

ImageController - Store, Fetch, Update, Remove etc Images

- GET - /images/names: Returns all the images objects of the database.

- GET - /places/{pid}/images: Finds the place from the given id and returns all its images as objects (not the real images).
- POST - /places/{pid}/images: Finds the place from the given id, and stores for that place all the images provided in the multi part as a file array.
- GET - /images/{id}: Finds an image by its id and returns its content as a byte array.
- DELETE - /images/{id}: Finds an image by its id and remove it as a file and from the database.
- PUT - /places/{pid}/images: This endpoint, finds the place by its id, removes all the images that already exists from storage and the database, and stores the new images that provided from the multipart as a file array.

ReservationController - Creating, Reading and checking Reservations

- POST - /reservations/places/{pid}/users/{uid}: Creates a reservation for the place with the given id and the user also with the given id. In more details the algorithm is the below:
    - Gets from the request the 'start' and the 'end' dates for the reservation
    - Firstly finds the user with the given uid
    - Then finds the place with the given pid
    - Finds the availabilities of the place
    - If no availabilities was found, then returns a false response
    - Else it checks for all the availabilities if the 'start' is before or equal to 'av_start' and 'end' is before or equal to 'av_end'
    - If an availability is found, then the availability is split correctly, according to the case, and then the divided availabilities (if they exist) are stored and the old one is removed
    - Then, the algorithm checks if the user has already a reservation in that place. If he had, the old one gets deleted!
    - And last, the reservation is created and a true response is returned.

- GET - /reservations/places/{pid}/users/{uid}: using a *named query* checks if the specified user had a reservation in the specified place (after it finds them).
- GET - /reservations/places/{pid}/users/{uid}/canrate: Checks if the user can rate the place. That means that the user already has a reservation for that place and also that the 'end' of the reservation is before the current date. The endpoint response with a true or false.

RatingController - Creating, Reading and Deleting Ratings

- POST - /users/{uid}/places/{pid}/ratings: This endpoint, finds the user by the given uid, finds the place by the given pid and then creates the rating for the target place which is located in the body of the request.
- DELETE - /ratings/{rid}: Finds a rating by its id and deletes it.
- GET - /places/{pid}/ratings: Finds the place from the given id, and then returns all the ratings in that place as a set.

SearchController - Check and Fetch Searches

- GET - /search/{uid}/has: Returns a boolean value that shows if a user with the given id has made searches in the history.
- GET - /search/places: This endpoint returns suggested places in pages to the user according to their last searches. The algorithm is similar to the search end point.
    - Load last searches
    - For each search gets the geolocation (lat and long), and looks for places in radius of 0.3f (~33 km)
    - Then packs the places in a list sorted by the cost per person field
    - Using the paging controllers (pageSize and pageNo) creates the desired sublist

The user id, and the paging controllers are get from as params in the request.

MessageController - Create, Fetch, Handle etc Messages

- GET - /messages: Returns all messages
- POST - /messages/{sid}/{rid}/{hid}: Creates a new message in which the sender is the user with id the sid, the receiver is the user with id the rid, and the host user is the one with id hid.
- GET - /messages/host/{u1id}/{u2id}/chat: Returns all the messages between two users where the client that makes the request is the user with u1id and has the role of host. As I have already mentioned, *a user has different messages as a host and as a guest*. This function fetches the messages of the user1 with the user2 where the user1 is the hoster. This endpoint is using a named query, finds the messages between the two users where the user with u1id is the hoster in those messages. The messages are sorted in descending order according to the *createdA*t column.
- GET - /messages/{u1id}/{u2id}/chat: Returns all the messages between two users where the client that makes the request is the user with u1id and in the

conversation he is not the host So, it fetches the messages between those two users where the user 1 is not the hoster. This endpoint is using a named query, finds the messages between the two users where the user with u1id is not a host in those messages. The messages are sorted according to the createdAt column descent.

- GET - /messages/host/{uid}: This endpoint fetches the last message of the target user in each of his conversations as a host. It uses a named query and all the returned messages are parsed, and the last message for each conversation is picked and added in a list, which is returned in the end.
- GET - /messages/host/{uid}: Same as the last one, but in this case the target user is not a host.

NamedQueries

All the named queries are located in the repository of the model that they refer to. The named queries that are implemented are:

- For place searching

```
SELECT p FROM Place p WHERE p.latitude >= ?1 AND p.latitude <= ?2 AND p.longitude
>= ?3 AND p.longitude <= ?4 ORDER BY p.costPerPerson
```

- For fetching all the messages between two user where the user1 is a host

```
SELECT m FROM Message m WHERE m.hoster=?1 AND ((m.reciever=?1 AND m.sender=?2) OR
(m.reciever=?2 AND m.sender=?1)) ORDER BY m.createdAt
```

- For fetching all the messages between two user where the user1 is a NOT host

```
SELECT m FROM Message m WHERE m.hoster<>?1 AND ((m.reciever=?1 AND m.sender=?2) OR
(m.reciever=?2 AND m.sender=?1)) order by m.createdAt
```

- For fetching all the messages of a user in which he is not a host

```
SELECT m FROM Message m WHERE m.hoster<>?1 AND (m.reciever=?1 OR m.sender=?1) ORDER
BY  m.createdAt DESC
```

Bonus

The Bonus part has not been implemented due to the lack of time and data.

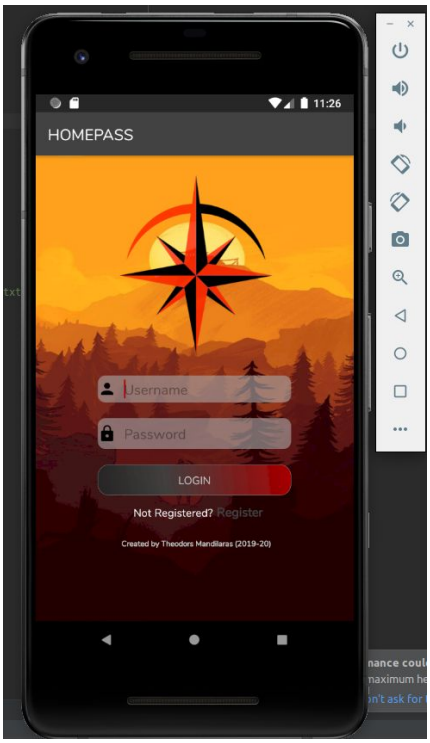
This is almost how the Backend of that application works. Lets now have a deep look in the android application of the Frontend.

FrontEnd - Android Application

Introduction

Now let's talk about the frontend which is the main client of the pre-described backend. Because the backend has been implemented as a REST API of the application the frontend would be able to have many forms. In this case, as frontend an android application was implemented.

The application targets smartphones with SDK version 29, which means they need to have android 10. However, the minimum SDK version in order to run in the smartphone is 26.



For the implementation and the debugging, a Pixel 2 emulator was used with API 27.

The communication with the backend is achieved with the **Retrofit** library (version 2.4.0), and for jsonify is used the gson-converter (version 2.4.0). Retrofit is used almost the same way as we learned in the course. However, for doing asynchronous requests, I used a method called ***enqueue()*** which makes the request and creates a callback either if we have a response or an error.

Moreover, a circle image view addon was added in order to have the CircleImageView, for stylistic reasons.

The application contains two different backgrounds according to the mode (**GUEST** or **HOST**). Both of those backgrounds have been taken from a game called **Firewatch.**

 **GUEST MODE**	 **HOST MODE**

Also, the font that was downloaded and used is the Nunito-light (like this one).

All the strings in the application, the main colors, and some custom background views just like the buttons and others, all are located in the ‘values’ folder.

Just like the backend application, the model directory contains the main entities that are used (Availability, benefit, rule, image, etc). All those objects have mostly the same structure as the other in the backend.

I will try to describe deeply the details and how things are designed and implemented in this android application, following the user experience path.

LoginActivity

By executing the application, the first view we get is the loginActivity. This activity contains a simple view, that waits for the user to fulfill his credentials in order to login in the application. Those fields are the username and the password. When the user tries to login, a POST request is made to the backend with the credentials stored in a Login object. The backend validates the request, and if the credentials match, then a success response is received with the JWToken in the header. The token is extracted by the loginActivity and is stored along with the username in a static class called

AppConstants with static fields. If the request failed then an Unauthorised message is shown as a Toast.

If the user has not an account, he can easily create one by clicking the Register text below. This will lead him to the RegisterActivity.

RegisterActivity

The registerActivity is a form which collects all the required information the application needs from a user. Those information are:

- Username
- Email
- Name & Username
- Password (& Confirm Password)
- Phone
- Address
- A checkbox to select if he wants to enable the **HOST** mode
- And a clickable image which allows the user to add his own image, from the gallery of the phone.

In order the user to submit this form, all the fields (except the image) should be field in order to create his account. If the image is not set, the default image will be used. Before the submit, the application also checks if the password and the confirm password are the same.

On submit, a user object is created and sent with a POST request to the backend. If the username is already used, then the request fails and a Toast informs the user to set a new username. If the request is successful, the application redirects the user to the loginActivity again, and presets the username of the user.

On the bottom of the RegisterActivity the user can press the login text to go back to the loginActivity.

MainActivity - HostActivity

The main and the host activities are the main activities that host the rest of the application. MainActivity hosts all the views for the **GUEST** mode and has a **red** theme. HostActivity hosts all the views for the **HOST** mode and has a **green** theme. Both of those activities have almost the same architecture. They contain a frame layout in which all the fragments with the views are taking place, and the bottom navigation bar.

Those activities handle the navigation bar and the back button events (using an interface for the Fragments called MyFragment) in order to set the correct view.

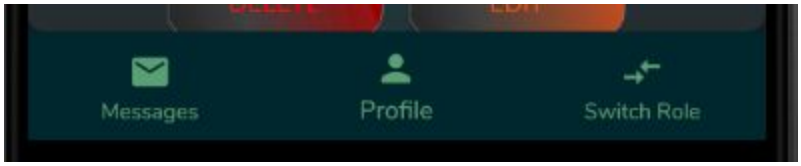
After logging in the application, the MainActivity takes place. In this activity the **GUEST** mode is set (as a default) and the user object is fetched from the backend. Both of them are stored in the AppConstants file, so each view can have access to them.

The landing fragment is the HomeFragment.

Let's now have a look in the NavigationBar and in the views with their functionalities.

BottomNavigationBar

Object: `com.google.android.material.bottomnavigation.BottomNavigationView`



Host

Guest

In the view there are three buttons,

- MESSAGES → sets the view with all of the messages according to the *mode (MessageFragment)*.
- PROFILE → sets the profile view in which we can edit our user details. *(ProfileFragment)*
- SWITCH ROLE → switches between the roles (allows only if the user has the **HOST** role enabled).

All those views have as 'back' the Home(fragment) view for the guest, and the Place(fragment) view for the host which are the landview of each mode. Also, in both of those views we get to the login view, and then to the exit of the app.

Starting from the views as a **GUEST.**

HomeFragment - Search, scroll and view places

The home fragment is the view that contains the expanding search component and a scrollable list view with fetched places.

Expanding Search View

When the search view is pressed then, the view is expanding (using a delayed transition for the expansion). The view contains the required fields that are necessary in order to search places in a location. Then the search button is clicked, then all the typed data are gathered, the then typed address is converted into a geographic location **latlong.** If geographic coordination is found (latlong) a POST request is done to the backend to fetch places.

The search view can be minimized again by clicking somewhere else.

PlacesContainer - Default Places

In this view all the places are shown using a view called *place_component*. When the view is set, and the user is fetched, the application asks the backend if the user has previous searches. If he has, then asks places that are close to the previous searches of the user and shows those places by default. If the user does not have previous searches, then random places are fetched.

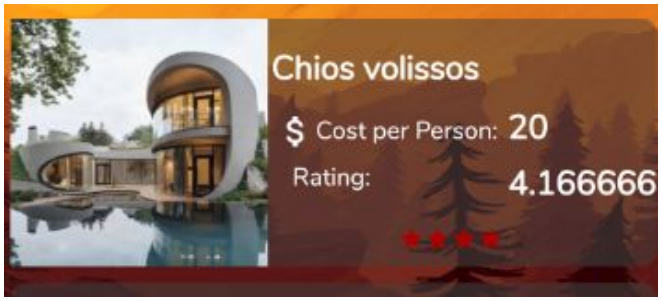
When the user makes a search, the results are shown in this view.

In all the cases, the places are fetched 10 by 10 as much as the user keeps scrolling down, and they are sorted according to the cost per person field.

Place Component

Contains the main image and the some information of the place. Those are:

- The address
- Cost per person
- And the mean rating



Minor issue: When no ratings have been provided, the mean rating cannot be calculated, so it has a rating of NaN value.

ProfileFragment - Check, edit your profile details and Enable Host

In this view, each user can check his user details and can edit them. It has the same functionality no matter the current mode. When it is created, the user fields are filled from the AppConstants class which contains the information of our user, and the image of the user is fetched with a GET request.

If the user has not enabled the **HOST** role, he can easily enable it there by clicking the text that says it.

By clicking the edit button, the user can edit his details. On CANCEL the changes are discarded. On SAVE the changes will be packed in a User object and with a PUT request the details will be updated in the database through the backend. The user can also set a new profile image by clicking the image view, and select from his gallery.

Minor issue: On edit, the fields at first seem empty. However they are not. Their content will appear when the user selects one of them (don't know why).

MessageFragment - Check inboxes

By clicking in the bottom navigation bar the messages option, you get a simple view that contains your correspondence according to the current mode. In GUEST mode, you get the messages that you have maybe from some hosts, and the opposite in the HOST message. The messages are visualised with a custom view called *message_component* into a list view. If no messages exist, the view is empty. The messages are fetched with a GET request from the backend. When they are received, then in one by one all the images of the users are fetched too. By clicking in one of those messages, the user is redirected to the Chat view (ChatFragment). That view gets the target user id as an argument from a **Bundle** object.

ChatFragment - Getting and Sending messages with others

The chat fragment contains the conversation you have with another user according to your current role. On the creation of the view, all your messages in this mode (which is found from the AppConstants class) with the other user are fetched in descending order by their timestamp. Also for both of the users, their main image is fetched.

Sending a message, creates a message object and makes a POST request to the correct route and then it is added in the view. Although, the chat fragment does not check for new messages periodically. It only searches in the creation of the view. So you have to go back and front to sync the new messages.

Minor issue: If there are no messages, and the user sends one, the image is not set correctly on the message component. So the default user image remains.

DetailedPlaceFragment - Inspect, reserve and rate a place

By clicking the image of a place in the results of the HomeFragment view, the user is redirected in the DetailedPlace View which contains all the stored information about the selected place. The view gets the place id as an argument from a **Bundle** object along with the date range if the user has selected one. Firstly a GET request is done, in order the place to get fetched and in the meanwhile a loading spinner is appeared. When it gets the place the view is set, the loading spinner gets invisible, and multiple GET requests start to fetch the image of the owner, and all the images of the place one by one (with the main image). The view contains:

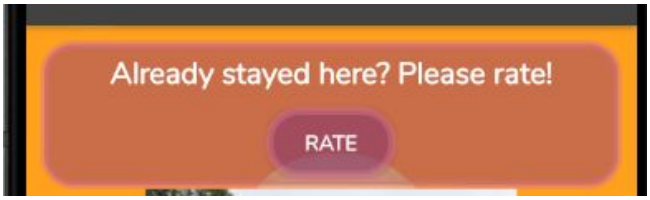
GalleryEffect

- **On Top:** On top an image gallery is located. When the images of the place are fetched, they are stored in a list with Bitmaps. In the meanwhile a **Thread** has been created, which iterates the list with the images and sets an image in the ImageView on the top using a **UiThread**. Between each set, it sleeps for 3 seconds. When all the images have been iterated, the process is repeated. That functionality creates the GalleryEffect that we see.
  Minor issue: I have tried to catch all the errors that can be generated by this effect, and to stop the thread smoothly when the user changes views, however sometimes it gets a **FATAL ERROR** because the view is destroyed and the **UIThread** is still active (that's my guess).

- **In the Middle:** The middle of the view contains all the details of the place. Those details are in the bellow categories:
    - Place details:
        - Number of beds,
        - number of baths,
        - number of bedrooms,
        - if it has a living room,
        - the area of the place,
        - and the type of the place
    - Description:
        - Free text provided by the owner and describes the place
    - Location
        - Typed address,
        - And a google map with a marker on the place
    - Benefits, Rules and Availabilities (each one has its own ListView)
- **On Bottom**:
    - Owner details:
        - Clickable image with the profile picture of the owner, which leads to the detailedOwnerFragment View (which will check right after this one).
        - A Button that leads to the ChatFragment view to start a conversation with the owner of the place.
    - Reservations:
        - If the date range has been provided and is accepted by the availability, by clicking the button the reservation is created, a Toast informs the user about the reservation and after the delay of three seconds, the user is redirected back in the HomeFragment. Else, the button is disabled.

Rating the Place

If the current user has already made a reservation on the place, and if the date range has passed, then he can rate the place. That information is fetched by the backend with a GET request, when the view is created. In that case, over the view, a purple box appears and informs the user that he can rate the place, and the owner.



By clicking the **RATE** button, the rating dialog appears. This dialog asks for the user to provide a degree about the place between 1-5 (floats are allowed) and to write a comment. By clicking the OK, the rating object is created and is sent to the backend with a POST request in order to be registered in the database. Also, the application thanks the user for the rating that he provided with a Toast.

The whole view is scrollable in order all the details to be contained.

DetailedOwnerFragment - Inspect Owner and the Ratings

By clicking the image view with the profile picture of the owner, the detailedOwnerFragment view is set. The view gets as arguments the owner id and the id of the place from the previous view with a bundle, and it uses them to fetch the details of the user, and the ratings of the place with GET request in the correct endpoints. Then everything is fetched, the view is set and a spinner component gets invisible.
The details about the user are:

- Name and surname
- Username
- His profile image (which is fetched separate)
- Email, Phone and address

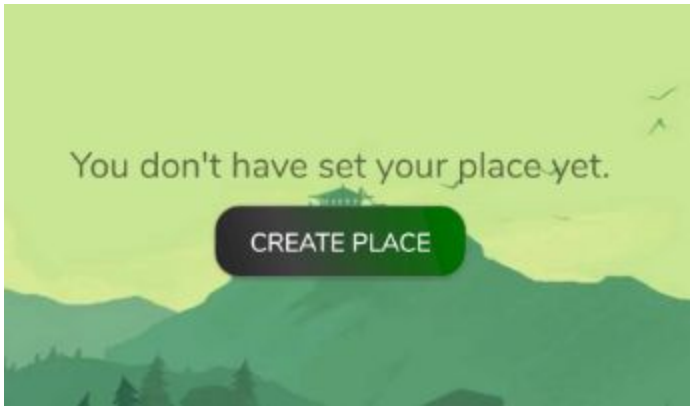
Also, below those details, are placed the ratings for the place and the owner, in a listView. This list view contains *rating_component*(s), which each one has for a rating the degree, as a value and as a RatingBar (with the stars) and also the comment of the rating.

PlaceFragment - Inspect, Edit or Delete your place

By pressing the Switch Role option in the bottom navigation bar (and having the HOST role enabled) the HostActivity takes place and sets the current mode to **HOST** in the AppConstants class. By default the activity sets the PlaceFragment in the view of the user.

The PlaceFragment firstly tries to fetch the place created by the user.

If the user has not created his place, then a “no place found” toast appears and a button that suggests you to create a place appears. By clicking this button, the user is redirected to the CreatePlaceFragment, in order to create his place.



If the user has already created a place, then, the place’s details are set in the view. Also its images are fetched one by one, and set to the gallery. The view is almost the same as the one in the DetailedPlaceFragment with a minor change in the order (the gallery

effect also takes part). This view is also scrollable in order to contain everything beautifully.

In the bottom of the view, there are two buttons which give the ability to the owner to either delete his place or to edit it.

- **Delete** creates a DELETE request to the backend and the place is deleted from the database within everything related just like images, benefits reservations etc. Then the app redirects the user to the same view, which of course it gets empty so the previous described view appears.
- **Edit** redirects the user to the CreatePlaceFragment view, which will allow him to edit his place's details.

CreatePlaceFragment - Create or Edit your Place

This view is used either to create a place or to edit an already existed place. If it's about editing a place, the fragments gets a boolean argument which informs it.

The view is divided in the below parts:

Setting Address

By typing an address on the search view on the top, and pressing the enter, the application looks for the address in the google map (which is below the search view) using the Geocoder class, and if the address is found, it selects the first result, it retrieves the geo-coordination and places a marker on that point using a zoom animation.

If no results are found, the user gets informed with a toast.
On a new search the last one is replaced and a new marker appears.

Place Details

Below the Google Map, exists a table in which the user will fill all the important details about his place. Those details are:

- Max number of guests
- Minimum cost to stay
- Cost per person (per night)
- Number of the beds in the place
- Number of the baths in the place
- Number of the bedrooms in the place
- If the place have a living room
- How much is the area of the place

Under that table, there is an input for the owner to provide free text with a description about his place, in order to advertise or to inform the guests.

After that, there is a radio group in order to set the type of the place (House or Room)

Images Part

Below the details, comes the images part. This is where the owner sets the main image (which is the image that is shown in the *place_componet*) and other images for the place.

By clicking the default house image (or the last image on edit) the application opens the gallery of the phone, and waits for the user to select **one** image. When the image is selected, from the URI of the image the application gets the Bitmap and sets it on the view. This is how the main image for a place is selected.

With the button below the users can select multiple images in order to show the place. The button opens the gallery and waits for many images, from which the app will get their URIs and sets them on the view with a similar to the gallery effect way (periodic iteration). **In the gallery for multiple image selection a long press click is required**.

Adding Rules and Benefits

Under the images part, there is the part for adding rules and benefits. For both of those features the functionality control is the same. There is an input text left and a button right. The owner writes on the left the rule or the benefit, and by pressing the button the rule or benefit object is created. After that is added in a list which contains them, and then using an adapterArray into the ListView below. An Entity in the ListView can be removed with a long press over it.

Setting the availabilities of the Place

Below those, there is the availability control. By availability is meant the date range when the place can host some guests. So it contains a date that the range starts and a date that the range ends. The owner can set ranges like that by pressing the ADD button. By pressing it two calendars will be opened, one after another.

- In the first one, the owner sets the start of the date range
- In the second, the owner sets the end of the date range

When both of the dates are inserted, an Availability object is created, it is inserted in a List with the availabilities, and then with an arrayAdapter to the ListView below.
With the same way as before, we can **remove** availabilities by long pressing over them.

Minor issue: dates on the past are allowed on purpose, in order to test and debug the rating feature.

Cancel

In the end, there is a button that allows the user to cancel the place creation or editing. This button discards everything and redirects the user back to the HomeFragment view, without storing anything.

Submitting an edited or a new Place

Above the cancel button, there is one that says POST. With this button the user can submit the place that he set. On submitting, all the items are validated, the application checks if the user forgot anything required. The non required features are the images, the benefits and the rules. If he does, the submit is canceled and a toast informs the user to set them required. If he does not then a Place object is created according with the users fills, the below pipeline takes part

1. The place object gets POSTed in the backend and it is stored in the database, after that the id of the fresh created place is returned as a response.
2. Using that id the availabilities, benefits, rules Objects are sent to be added also in the database using the place id.
3. The Images are also packed and sent in a multipart array.
4. When the POST request with the images (which is the most expensive) is complete, the application assumes that the Creation of the place is completed.
5. Then the user is redirected to the PlaceFragment which will fetch it as normally would do.

On Editing an existing Place

At first the place is fetched using the user id from the User object in the AppConstants, and along with the main image and all the other images too. When it is fetched, the map, the details, the benefit's view, the rule's view, the availability's view and all the images (main or not) are set with the fetched place.
The owner can make changes on the details, or in the address off the map by replacing the values in their fields.

Also, he can **add** new items on rules, on benefits or on availability with the same way as before, or he can **delete** an already existing item from the containers **by long pressing over the item**.

For the places images, if the user sets new images **all the old images will be deleted on submit and the new will take place.** This is something the user gets warned by the view. The default image also is replaced if it gets set.

The changes are submitted by the user in the same way as when he created the place. However, now a **PUT** request is done with the updated place object using the place id. The new created availabilities, benefits and rules are POSTed with the same way, and for the deleted one a DELETE request is done in order to get removed and form the database.
If the images are updated, similarly a DELETE request is done, so all the old ones to be deleted and the new one are POST again all together in a multi part array.

This is how a place gets created, edited or deleted in this Application.

And by finishing this fragment, I complete the description about the most important views and functionalities for the frontend part of the application. More information can be found in the implementation **code**.

Issues - 401 status code - Duplicate Places

1. Sometimes, in the first executions or on a fresh registration, after logging in, all the requests are getting a **401** status code in the response. I do not understand why this is happening sometimes, **but it is fixed after restarting the application.**
2. When the users make a search, that search is registered in the Database, and later it will be used for fetching places for the HomeView of the user. However, searches with common results may produce duplicate places for results.

Sum up - My opinion about the project

This project for me was very educational. Although that I already have worked with Android applications, not only it totally helped me to learn a lot of new things, just like fragments, google maps and calendars, but also helped me to exercise those skills even more.

Moreover, I got a very good taste about the Spring Boot Framework which is a very powerful web development framework, and with new database systems just like the H2.

As a next step, I aim to deploy the application over the web using an Amazon Instance, make some minor optimizations and fix the remaining issues and at last have some fun with it.

THE END