POLITECNICO
MILANO 1863

Design and Implementation of Mobile Applications - Polaris project VINCENZO MANTO, ROBERT MEDVEDEC

Design Document

Deliverable:	DD
Title:	Design Document
Authors:	VINCENZO MANTO, ROBERT MEDVEDEC
Version:	1.0
Date:	02-01-2022
Download page:	https://github.com/robertodavinci/android-dev-travel-app/tree/main
Copyright:	Copyright © 2022, VINCENZO MANTO & ROBERT MEDVEDEC – All rights reserved

Contents

Table of Contents . . . **3**

List of Figures . . . **5**

List of Tables . . . **5**

1 Introduction . . . **6**
- 1.1 Purpose . . . 6
- 1.2 Scope . . . 7
- 1.3 Definitions, Acronyms, Abbreviations . . . 8
  - 1.3.1 Definitions . . . 8
  - 1.3.2 Acronyms . . . 8
  - 1.3.3 Abbreviations . . . 8

2 Overall Description . . . **9**
- 2.1 Product perspective . . . 9
  - 2.1.1 Technologies overview . . . 9
  - 2.1.2 Internal structure . . . 10
  - 2.1.3 Scenarios . . . 13
- 2.2 Product functions . . . 13
  - 2.2.1 Adding a trip . . . 13
  - 2.2.2 Editing a trip . . . 14
  - 2.2.3 Finding a trip . . . 14
  - 2.2.4 Following a trip . . . 14
  - 2.2.5 Updating account settings . . . 14
  - 2.2.6 Offline function . . . 15
- 2.3 Additional features . . . 15
  - 2.3.1 Translation . . . 15
  - 2.3.2 Multiple language support . . . 15

3 Architectural Design . . . **16**
- 3.1 Overview . . . 16
- 3.2 Frontend architecture . . . 17
- 3.3 Backend architecture . . . 19
  - 3.3.1 User database . . . 19
  - 3.3.2 Trips database . . . 20
- 3.4 Middleware architecture . . . 22
  - 3.4.1 Communication with the backend . . . 22
  - 3.4.2 Handling unexpected actions . . . 22
- 3.5 Runtime view . . . 22
  - 3.5.1 Create a trip . . . 23
  - 3.5.2 Edit a trip . . . 24
  - 3.5.3 Add a destination . . . 25
  - 3.5.4 Search for a trip/explore . . . 26
- 3.6 Design constraints . . . 30
  - 3.6.1 Hardware limitations . . . 30
  - 3.6.2 Privacy limitations . . . 30

4 User Interface Design . . . **31**
- 4.1 Identity and colours . . . 31

4.2 User interface design . . . 32
4.3 Login screen . . . 33
4.3.1 Main screen . . . 34
4.3.2 Map screen . . . 35
4.3.3 Saved trips screen . . . 36
4.3.4 Explore screen . . . 37
4.3.5 Settings screen . . . 38
4.3.6 Trip creation screen . . . 39

5 Testing . . . 40
5.1 Unit testing . . . 40
5.1.1 Login unit testing . . . 40
5.1.2 Trip creation activity unit testing . . . 41
5.1.3 Navigation unit testing . . . 44
5.1.4 Map screen unit testing . . . 45
5.2 Deep linking testing . . . 46
5.3 Failure test . . . 47
5.4 Support of different devices . . . 48

List of Figures

1 **Internal structure of the system** . . . 10
2 **Structure of the NoSQL user database** . . . 19
3 **Structure of the "Destination" entities** . . . 20
4 **Structure of the "Trip" entities** . . . 21
5 **Sequence diagram 1 - Create a Trip** . . . 23
6 **Sequence diagram 2 - Edit a Trip** . . . 24
7 **Sequence diagram 3 - Add a destination** . . . 25
8 **Search by name and ID** . . . 26
9 **Search by map drawing** . . . 27
10 **Search by location** . . . 28
11 **Search by destination** . . . 29
12 **Destination details** . . . 29
13 **'Polaris' icon featuring an astronaut on a blue background** . . . 31
14 **Primary color #0083FF** . . . 31
15 **Secondary color #0460D9** . . . 31
16 **Background colours for light theme** . . . 32
17 **Background colours for dark theme** . . . 32
18 **Login screen** . . . 33
19 **Main screen in light style** . . . 34
20 **Main screen in dark style** . . . 34
21 **Map screen in light style** . . . 35
22 **Map screen in dark style** . . . 35
23 **Saved trips screen in light style** . . . 36
24 **Saved trips screen in dark style** . . . 36
25 **Explore screen in light style** . . . 37
26 **Explore screen in dark style** . . . 37
27 **Settings screen in light style** . . . 38
28 **Settings screen in dark style** . . . 38
29 **Trip creation screen in light style** . . . 39
30 **Trip creation screen in dark style** . . . 39
31 **Login screen - Tablet** . . . 48
32 **Login screen - Phone** . . . 48
33 **Home screen - Tablet** . . . 49
34 **Home screen - phone** . . . 49
35 **Explore screen - Tablet** . . . 50
36 **Explore screen - Phone** . . . 50
37 **Trip create screen - Tablet** . . . 51
38 **Trip create sreen - Phone** . . . 51

List of Tables

1 Introduction

1.1 Purpose

This document provides a detailed description, mainly of the architecture and the UI, of the 'Polaris' mobile application.
'Polaris' application is a system used to enhance travelling experience through idea sharing among people, exploration of other users' travels, documentation and easy manipulation of trip destinations and ideas. The system itself is made so the user has every single information regarding his travel in one place, without having to remember or worry about any details.
'Polaris' is mainly an Android mobile application, with a possibility of being also expanded as a web application and/or iOS application in the future.

1.2 Scope

'Polaris' is an application that helps all travellers around the world to easily and efficiently manage their travels and thus enhance their travelling experience. The target audience for this application is everyone who uses a smartphone and has at least some experience in using mobile applications, as some of the patterns and application usages might not be intended for the novices in the field. Thus the target audience ranges from 15 to 50 years old, although there are no strict boundaries.

The application is mainly intended to be used when a user is planning and making their own trip and has a liberty to organize their free time and places to visit.

Another important usage of the application is sharing trip ideas with friends and other users around the world. Planning trips by itself is a daunting and time consuming task, and with the lack of adequate applications on the market, we wanted to create something that is going to allow users to easily share and modify their previous trips and therefore improve the overall travelling experience for others.
The most important functions of the application are arranging a trip, finding points of interest in the area, organizing visits to those points, exploring accommodation and restaurants in the area, and crafting your own views of the travel by providing additional comments on the whole experience.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Application**: a computer (mobile) program that is designed for a particular purpose.
- **Smartphone**: a mobile phone that performs many of the functions of a computer, typically having a touchscreen interface, internet access, and an operating system capable of running downloaded apps.
- **Google Maps**: a web mapping service developed by Google, used both as a standalone app and as an integrated mapping solution in most of the apps.
- **iOS**: operating system developed by Apple, used by their portable devices like iPads and iPhones.
- **Android**: most popular operating system for smartphones and tablets, developed by Google and partners.
- **Backend**: the part of a computer system or application that is not directly accessed by the user, typically responsible for storing and manipulating data.

1.3.2 Acronyms

- **API**: Application programming interface, computing interface which defines interactions between multiple software intermediaries
- **UI**: User interface
- **GUI**: Graphical user interface
- **DB**: Database
- **REST**: Representational state transfer - software architectural style used in web services

1.3.3 Abbreviations

- **App**: Application.

2 Overall Description

2.1 Product perspective

2.1.1 Technologies overview

- Android Studio Development Kit
  - Kotlin
  - Android Compose
- Data management
  - Google Firestore
  - Google Firebase
  - Database
  - Room
  - Shared preferences
- Authentication
  - Google Authentication
  - Facebook Authentication
  - Email Authentication
- External APIs
  - Google Maps
  - Google Translate
  - Google Cloud Messaging
  - Google Places
  - OpenWeatherMap
  - GPT 3 - OpenAI

2.1.2 Internal structure

The proposed system uses an application on the server side that is connected with the APIs to the Android on the phone.

The Way of communication between the parts of the system and logic behind the system is presented in the following figure:

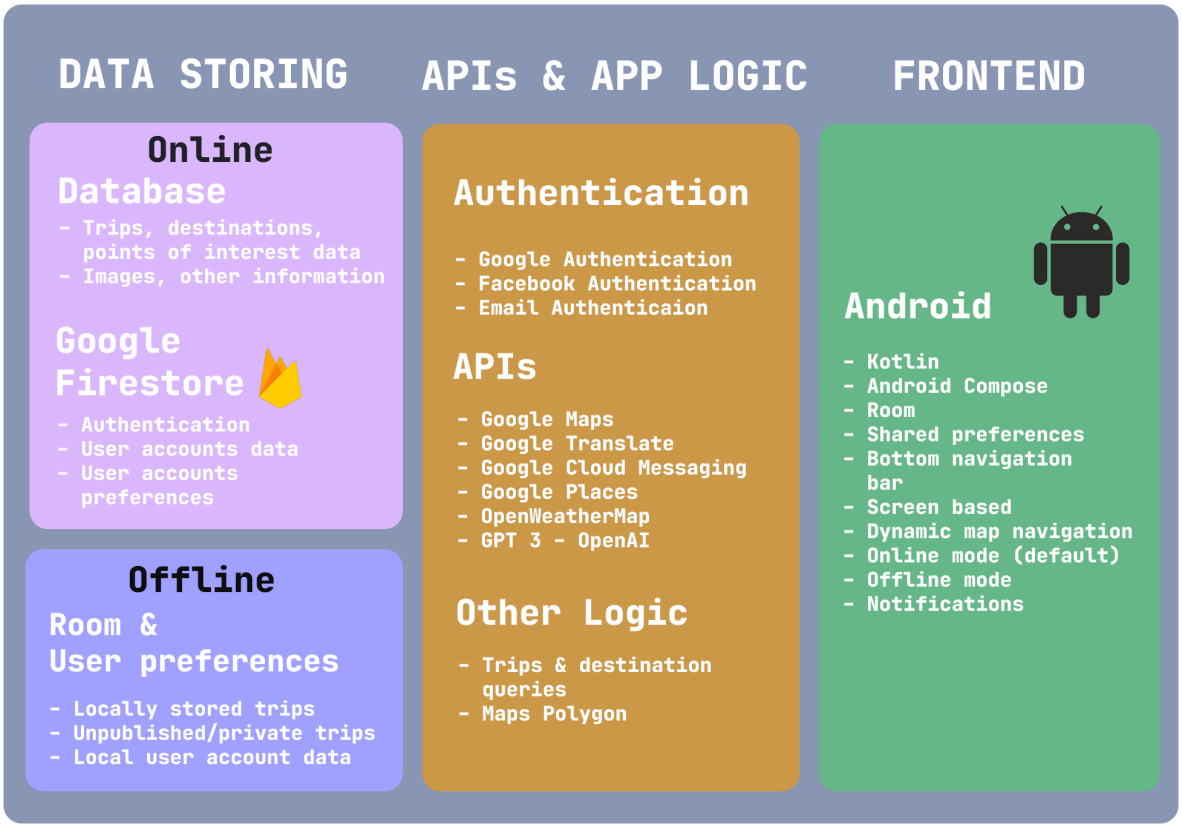

Figure 1: **Internal structure of the system**

Data storing

The database part is divided into two separate entities. Most of the authentication part is done directly through Google Firebase and Firestore frameworks, which offer great support for Google and Facebook accounts login integration, as well as native support for email based accounts. Controlling authentication via Google services is fast, easy, and seamless, so we have decided to keep most of our logic, as well as account data, stored here.

User account details, such as username, ID, and user preferences, are stored in the Google Firestore database, and are fetched after every login, so that the local user account data is consistent with the one on the servers.

Images that are used for trips and destinations are also stored in the Google Firestore database.

The other part of the database, which stores the data about the destinations, trips, points of interests, and other features related directly to the workflow of the app, are stored in another, separate database. This decision was due to the fact that Google Firestore is a NoSQL database, so putting relational data there would make it inefficient and slow compared to the traditional SQL database.

This is what made us decide to transfer this data to completely other place and use a different technology than with the accounts. Account data storing and fetching works exceptionally well and fast with Google services, so we had no real reason to move it from there, especially since that data is more sensitive and Google services provide high level of protection.

Android framework also supports numerous features for working locally. We have used these features to have an additional "offline" mode of our application, which allows full functionalities despite the user not having internet connection. We use Android Room to store all the desired trips downloaded and edited by the user, as well as the ones that the user has created locally and decided not to publish yet. "Offline" mode is then synced as soon as the user gets Internet connection, if any additional changes have been made to the local or public trips.

Finally, shared preferences are used to store the user's local information such as username, theme preferences, display name, and other features that help user while navigating the app. These are updated with the data located in the Google Firestore framework.

APIs & App Logic

Authentication used for the application is done directly through Google Firbase. Account creation can be done directly through email, after which it has to be confirmed via confirmation email. The other way of creating account is logging in with either Facebook or Google account. This way account gets automatically created, or joined if it has already been previously created with the same email. This allows users to not have to worry with which account they have logged in before, as it is all connected to the same email. Users would have to be logged in to either their Google or Facebook account before using those methods to log in to the app.

Google and Facebook "button" directly communicate with their respective APIs and only after the conversation has successfully ended do communicate with Google Firebase servers.

Another thing that is used from Google framework is the Google Maps API. From that API, several different features are used, such as Maps presentation, location searching, cities searching, distance calculation, and others. Google Maps are dynamically presented in the app and allow users to search through them either by text or by touch.

Another important thing is a small PHP script that is running on the server. It serves as a backend of the app, handling query fetching, control, and execution, as well as handling and displaying data that is coming and going through Google Maps API.

This PHP script is returning all the necessary data in the form of JSON files, which are then handled by the app accordingly and to its needs.

Frontend

On the frontend side, we have decided to use all of the latest technologies recommended by Google for Android app development.

Kotlin is used as the programming language, which allows better data and variable control in Android environment than the previous official language Java.

From the pure design perspective, Android Compose is used instead of the previous template/xml based design. Compose allows for creation of certain elements, as well as their dynamic modelling, which allows for reusability and code cleanliness. Compose allows for elements to by dynamically inserted into specific places, as well as to be dynamically changed, hidden, or manipulated in a much faster and easier fashion. It also allows for faster development since the previews of those compose components can be rendered much faster.

As the base of the system, bottom bar with different pages has been used. More on that part is explained in latter chapters of this document.

The app supports both online and offline mode, with shared preferences used for storing local display preferences of the app.

2.1.3 Scenarios

Scenario 1

Marco wants to go to a trip to France. His friend, Federico, already went on a trip in a similar area and visited points of interest that Marco likes. Marco asked Federico to tell him all of the places he visited, how did he travel, and the order in which he visited these places. Federico made some notes for Marco, but it still left Marco with the problem of having everything in his phone organized and in one place. That's when Marco found out about the app and asked Federico to put everything he remembers from the trip here. That way Marco had all of the locations in the correct order, as well as comments about certain places and other detailed information all in one place, which allowed him to easier follow the trip during his time in France.

Scenario 2

Rebecca, the friend of Marco and Federico, heard about their trip to France and decided to go there as well. However, she is not as interested into visiting so many French castles like Marco and Federico, and wants to add some other destinations to her trip. However she still wants to keep the first half of the trip, where guys visited the southern part of France. She also uses the app and copies the trip, after which she edits it and replaces castle destinations with beaches on the northern part of the country. She renames the trip and publishes it with a different name. This way Rebecca saved some time and effort on planning the entire first half of the trip, whilst being able to modify the experience in the latter half.

Scenario 3

Giovanna wants to on a trip but she already visited France. She needs some new ideas. By using the "explore" function in the app she finds some of the top rated trips around Europe - her favourite of those is a trip to Iceland. Giovanna follows all of the guidelines in the trip and has an amazing time.

2.2 Product functions

Functions of the system provide easy and intuitive ways to use the app. They are somewhat connected and have overlapping features. Nevertheless, the users may use only certain parts of the system and still get the full functionality they need from the app. These functions are mentioned in several places in the document, but their most thorough explanation can be found here.

2.2.1 Adding a trip

Adding a trip is the most essential part of the system. The function features several parameters and allows for high level of customizability in order to create a fully unique travel experience.
Each point of interest is featured as a specific "destination", although even places that are not regarded as specific destinations can be inserted into a trip. Destination fetching is done through the Google Maps API, with the users also having the ability to add and edit their own destinations. The function features the following:

- Selecting a central point of the trip
- Adding other destinations to the trip
- Writing a trip description
- Adding preferred ways of travel between destinations

- Adding comments to trip destinations
- Adding images to destinations and trips
- Adding multiple days to trip
- Determining optimal arrival time and visit time for every specific destination
- Choosing whether the trip is public or not

2.2.2 Editing a trip

Editing a trip can be done on two different types of trips - public or private. If the trip is public, either published by the user editing it or someone else, by starting to editing an exact copy of that trip is created, which can then be published again under different name and different trip ID.
If the trip is private and has not yet been published, then a new trip is not created, but rather the trip ID stays the same. If the trip is then published and edited again, the new edited version of the trip has a new trip ID and is a whole new entity.
Editing a trip features all of the same functions that adding a trip does, which means that everything from a small comment to the whole trip can be modified.

2.2.3 Finding a trip

Finding a trip can be done in several different ways. The first way features a search bar which then searches a trip by the central point name or by the city that is the trip based on. The second way is a search by the trip ID/name, which can be directly received from other users. Trips can also be found by looking at the interactive map, selecting the area of the desired starting point of the trip, and finding the trip through the distinct trip name and photo. Trips can also be scrolled through on the main page of the app, which offers different suggestions. There is also an option of searching through the specific destinations used by other users in their trips in a certain area close to the current user, which can be a good starting point for new trip ideas.

2.2.4 Following a trip

This function mainly uses a specific trip and sets it as an active trip of the user. This makes it easily accessible by the user at all times by using the bottom navigation bar, and allows him to follow certain steps of the trip without losing progress. This function also allows the user to change the current active trip and still keep the progress of an old trip, so that the progress can be easily restored when that trip is again set as an active trip.

2.2.5 Updating account settings

Only a few settings can be changed in the user account. List is the following:

- Changing a username
- Changing a profile picture
- Switching between dark and light colour scheme
- Turning notifications on or off
- Switching between offline mode and online mode

2.2.6 Offline function

The system allows the users to be disconnected from the Internet and still use the app fully. Individual trips can be downloaded and store in the phone internal storage, and then accessed at any time. If any changes are made to the trip during the offline time, a new iteration of the trip gets created and published as soon as the connection is restored and the user has come back to the online mode. Multiple trips can be downloaded and kept in the storage of the phone at all times.

2.3 Additional features

2.3.1 Translation

Translation API is used to translate some information about the trips. The app will detect the language of the phone and translate the description of a specific trip to this language. This allows users to quickly see the overview of the trip and decide whether they like it or not, as the app is meant to be used on the international level, it is highly possible that a large number of users will not use English as their primary language.

2.3.2 Multiple language support

Besides the translating descriptions, the app offers a variety of different languages in which it can be used. Every single function and word in the app is translated into the language of the system, and is completely usable without needing to know a single word of English language. The app currently supports only two additional languages, Italian and Croatian, to go along with the original English, but the infrastructure is set so that more languages can be added swiftly and easily.

3 Architectural Design

3.1 Overview

Architectural design of the application is based on the three-layer model used in most applications. The three layers are the presentation layer, or frontend, the application layer, or middleware, and the data layer, or backend. Each of those layers does its own part of the job and communicates with other layers in order to present the correct information to the user.
The architecture is also a typical client-server implementation where server holds the data and the client is accessing it through requests (besides the offline mode where the user stored some of the data from the server locally and is accessing it without using online requests).

3.2 Frontend architecture

Frontend part of the application is using viewmodels, which are more commonly known as **model-view-viewmodel** architecture.

The base of the Android app technology is the Android Compose, which uses different approach to the app development, as it draws specific components based on the state that the app is in. This makes for much more lightweight app, as the activity or intent of the Android app is much less frequently changed as only some of the components that are being drawn out on the screen are being interchanged, which saves a lot of time and resources. This approach also allows for code reusability as components can be used in either completely identical shapes, or just slightly altered depending on the use.

This is where the viewmodel architecture comes in. Instead of directly handling the data and communicating with the data layer of the application, which in this case is done through the middleware, the frontend has additional sub layer which handles all the data, communication, and calculations of which parts of the interface should be shown. There are many advantages to this, with the main ones being complete independent handling of the communication and the presentation.

Making big changes or adding additional functions in the middle of the app development can exponentially increase the production time and can quite often lead to rewriting the same parts of the code multiple times. Viewmodel architecture provides a certain kind of stability of the code, as adding additional function will usually mean writing either only the presentational part or the communication part, and then just connecting it with other already existing functions. It dramatically improves clarity and speed of development once everything is set up. Lastly, detecting bugs and unintended behaviour in the app is far easier, as we always know in which layer the problem is located. This makes it not only easier to find, but also easier to fix, as there is a good chance that we are only going to have to change a small part of the code, assuming that the other part is written correctly. In other types of architectures this often leads to rewriting entire parts of the frontend.

One of the main disadvantages of this model is it's complexity, as there is another additional layer of communication in which the data needs to be handled and adapted in order to be passed through. This of course increases the time of development and the possibility of bugs, but if everything is done correctly, it provides a great tradeoff.
Multithreading is also used in a lot of different places in the app, specifically in fetching the data from the database and Google Services and working with it in the background while the user can easily navigate the app.

Almost every screen/activity has it's own view model, with the focus being on ones that have the most communication with the backend. The list of all the activities and their respective viewmodels is the following:

- Main Activity
- Trip Activity - Trip View Model
- Trips Screen - Trips View Model
- Trip Creation Activity - Trip Creation View Model
- Map Screen - Map View Model
- Location Screen - Location View Model
- Location Selection Screen - Location Selection View Model

- Home Screen - Home View Model
- Google Place Screen - Google Place View Model
- Login Activity
- Inspiration Activity
- Explore Screen
- Around Me Activity
- Active Trip Activity
- Profile Screen

Other types of design elements included in front development, besides the common ones used in pretty much every single app, include the following:

- Bars
- Buttons
- Cards
- Switches
- Icons
- Images
- Map

The general design and look of the UI is shown in more detail in the Chapter 4 of the document.

3.3 Backend architecture

3.3.1 User database

As previously mentioned, user database is located in the Google Firestore service. This service uses real-time NoSQL database. This database is organized in collection->document system. Everything starts with one collection, which can hold as many documents (entities) as possible. Each document can have an unlimited number of attribute fields, which can be with a specified type or without one, and at most one collection. Then the cycle repeats again.

Working with NoSQL has its advantages when it doesn't have a lot of relational and connected data. We can extract only the small amount of data we want, it is very fast, and also very efficient. Since accounts are not connected in any way, using NoSQL database seemed like a viable choice since it saves our users time and data.

The architecture of this database is the following: the first collection contains all the users as their unique IDs, where they are represented by a document. The first level of the document holds only the info of the display name and the ID. The second collection is made for storing user preferences, and every user has their own. In that collection there is a document that holds all of the parameters needed for the user's usage of the app, such as colour mode, economy level, thumbnail URL; and two optional ones, real name and real surname

The following figure represents the structure of the explained database.

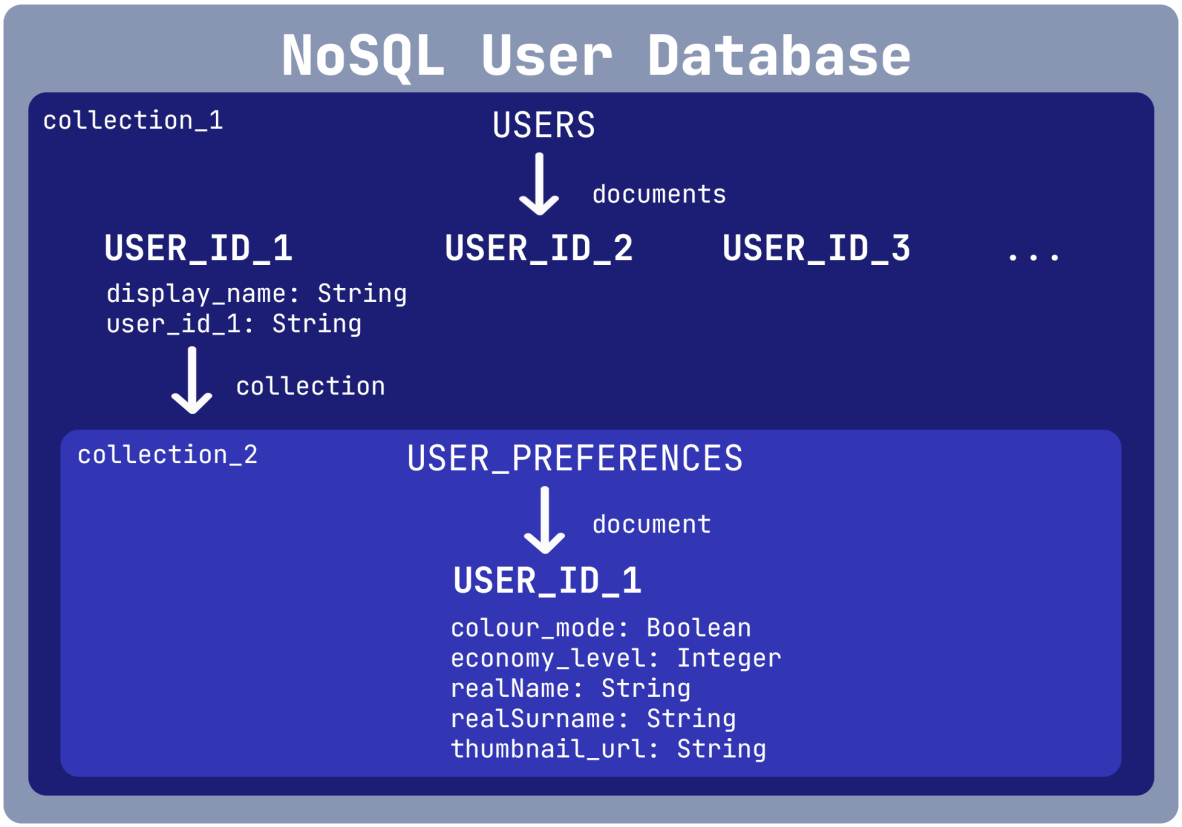

Figure 2: **Structure of the NoSQL user database**

3.3.2 Trips database

Database regarding trips has several different parts that make the whole system work. The most integral part of every trip are destinations. There are two types of destinations in the database - destinations retrieved from Google Places API and user created destinations.

When adding a destination to the trip, users can either find a destination using Google Places API, or if they are not satisfied with the search, they can create their own destination, with a unique name and a location on the map. This destination can then be provided with a specific name, GPS coordinates, rating, and an thumbnail image. Newly added destinations receive an ID generated by the backend, while Google Places destinations have their own unique ID, which has a different form and shape. Users are always going to be urged to use already existent destinations, so unless there is not a really specific location in question, they would have no need to create their own, especially since the Google database is quite large.

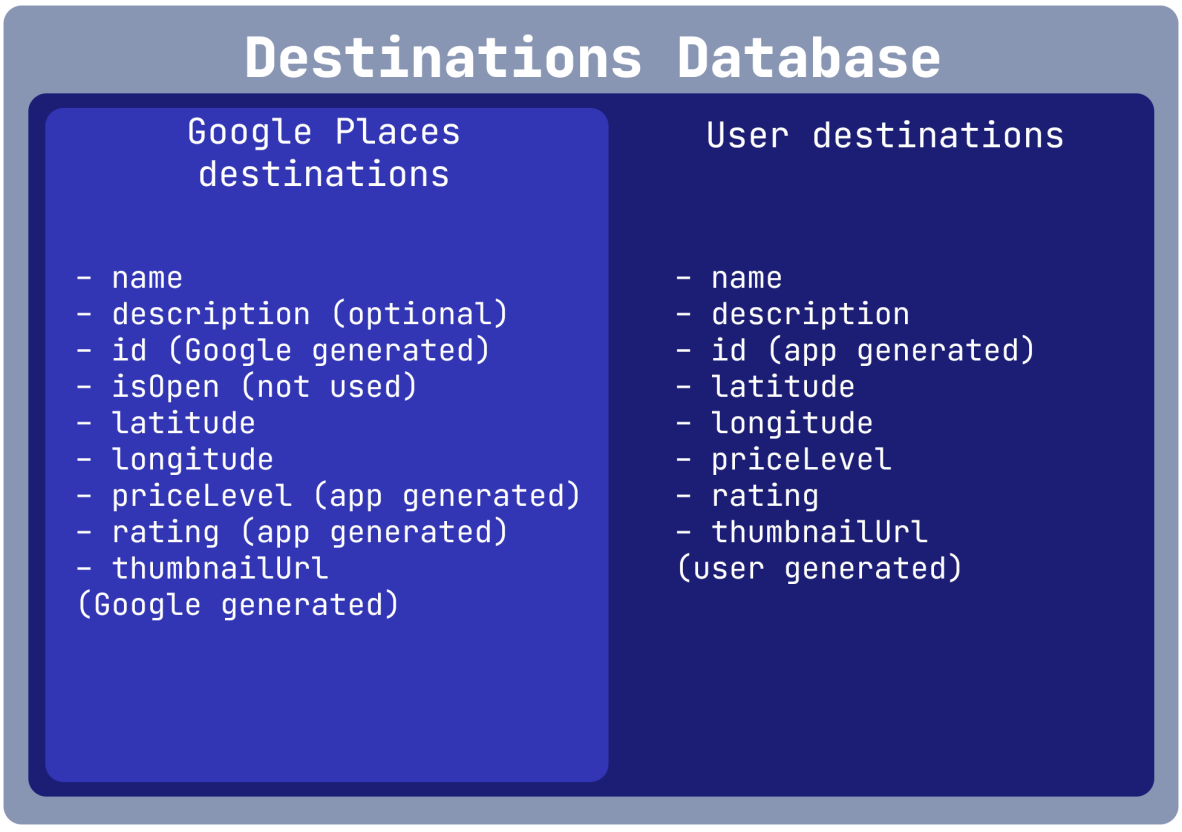

Figure 3: **Structure of the "Destination" entities**

Trips structure are much more complex and detailed. Every trip is defined by several important attributes. Besides the basic information like name, id, description, and thumbnail URL, trips are mainly defined by destinations. Central point destination is the most important point of the trip, which necessarily doesn't have to be a beginning or an ending destination. Then for every single day of the trip there could be one or more destinations defined. Along with the regular destination attributes, those destinations feature the distance to the next destination (both in minutes and kilometers) taken with the recommended mean of transport, which is calculated through Google Maps and Places APIs. Those destinations also feature approximate hours of arrival and a recommended time to be spent there so that a plan could be fully followed. Every destination can also feature notes which could give users additional information about it or just provide some tips on what to do and what to see. A list of image URLs can also be added for every specific destination in the list.

Besides all of the destinations and the basic info, trip entity also features creator of the trip, creation date, recommended season of travel, rating, and public boolean, which defines whether the trip is private (and therefore hidden from other users) or public (and can be seen by everyone).

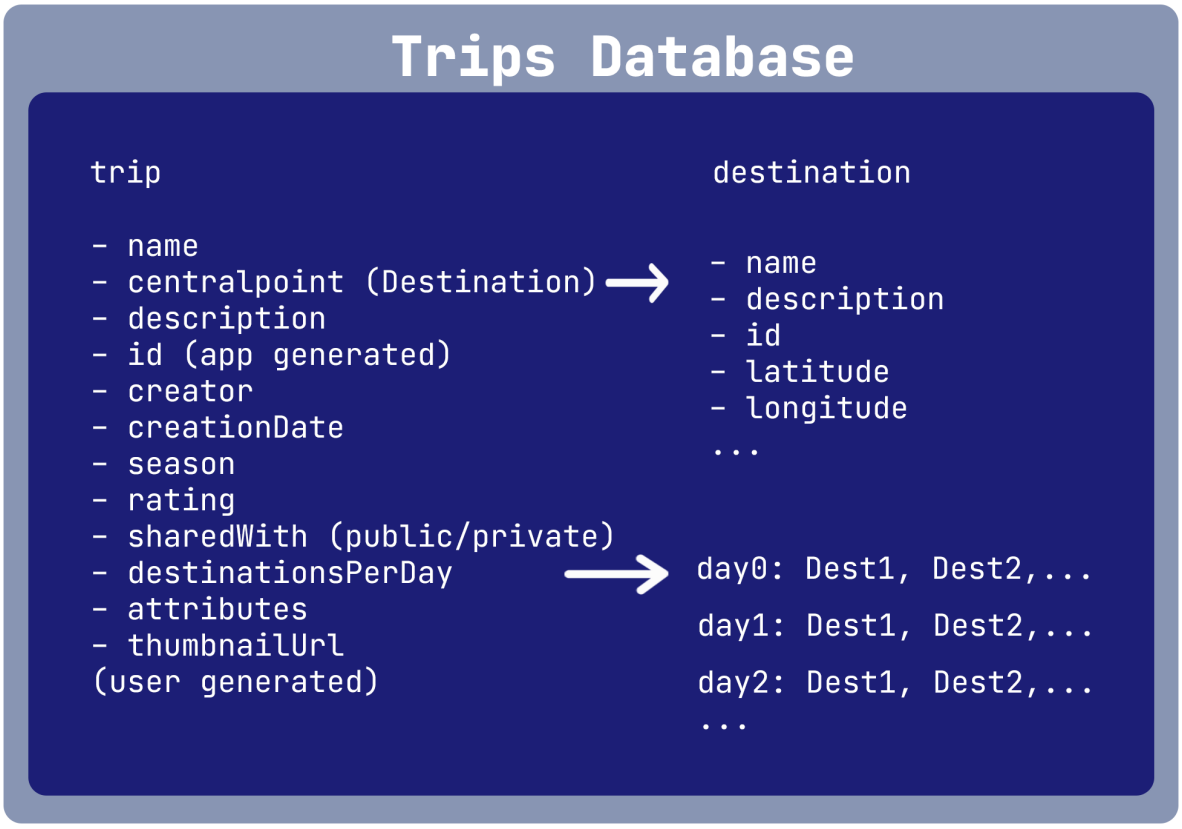

Figure 4: **Structure of the "Trip" entities**

3.4 Middleware architecture

In order to connect the data on the server to the screen of the phone and to allow the user to properly see the data, we have implemented a complicated layer of functions and classes in order to create easy-to-use and esthetically pleasing experience for the user. In this section the most important functions of this layer, and everything that does not belong to the frontend and backend will be explained.

In general, it works as a separator or a controller of the three-layer infrastructure that is being used by the app.

Notifications are also enabled and allow users to see new trips that hold the destinations they have favourited. These can be easily turned on or off in the settings screen.

3.4.1 Communication with the backend

App comunicates with the backend through the queries. After user's input, they are generated within the app, and the query is sent to the database. Database then collects all the needed data and sends it back to the application in the form of JSON files, which are easily readable and are rather simple to manipulate.

3.4.2 Handling unexpected actions

There are several instances in which the unexpected behaviour of the app or the phone can cause users to lose a lot of their previous work. As creating a new trip from scratch can be a long process, especially for longer trips, there is always a problem of consistency. If the user visits some other apps in the meantime, the risk of 'Polaris' app changing lifecycle state can destroy all of the previous work. The same can also happen with the phone's battery being depleted or if the user accidentally closes the app.

The solution of this problem is found in the premature saving of the trip. Every time the user leaves the trip unfinished or exits the app without either saving or discarding the trip, the app will save the trip with the "unfinished" attribute. This means that the trip will exist just like normal trips, but will not be published in neither private or public mode. The reason for that is that the trip might not even have the most basic attributes like name, which is a crucial attribute for the trip to be published, but is quite often left as the final piece of the puzzle when being created by the user.

This way whenever the users return to the library of created/saved trips, they will found the previously unfinished trip and will be able to continue from where they left off. The middleware handles this part on its own, storing the trip in the database and taking care of all of the necessary actions, so the user doesn't have to worry about storing or fetching the trip.

3.5 Runtime view

In the following section, sequence diagrams that represent the most important use cases are shown and explained in detail. We have focused only on the most commonly used use cases in the app, as well as their crucial parts. We do not go into too much detail when it comes to specific parts of interaction between the layers, but rather want to present the vague idea and how it all works.

3.5.1 Create a trip

Create a trip use case starts with opening the app and pressing on the "Create a trip" button. A new screen opens up which allows us to change certain attributes about the trip and add different destinations. Every trip needs to have a name, which doesn't have to be unique, since there is an ID that is automatically allocated to the every trip. Every trip needs to have at least two destinations. We can search through destinations via the search bar or by using the interactive map and selecting a destination from the map. After selecting a destination, we add it to the trip. Destinations can be reordered and removed from the trip at any time, which is not shown in the diagram due to simplicity and concision. Trip can also be discarded. In the end, if the user is satisfied with its trip, they can publish it in which case it will be stored directly in the database. Local database is not used for storing unpublished trips, since this would mean that by losing local data, all of the unpublished trips would be lost. All of the trips are stored in the online database and are can be found by other users only if published.

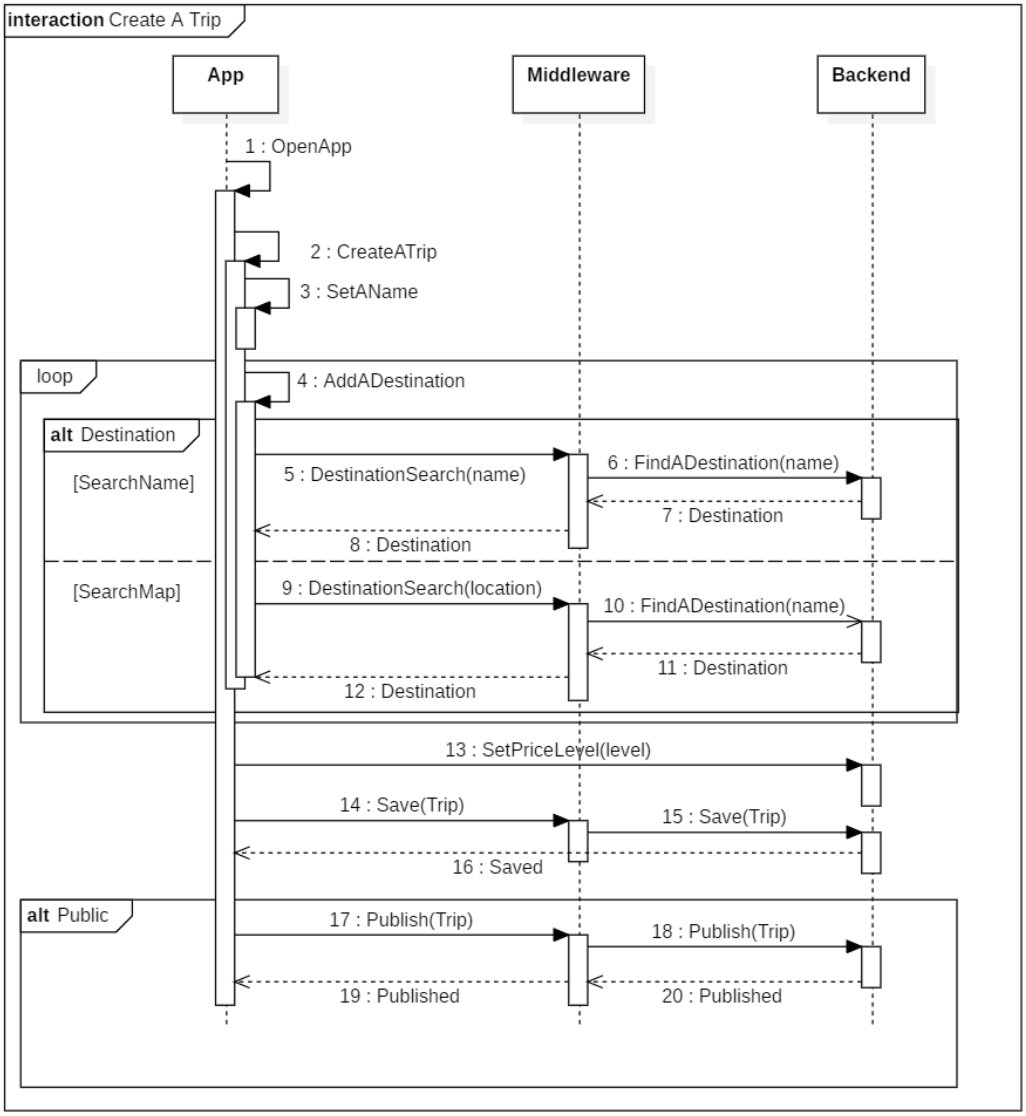

Figure 5: **Sequence diagram 1 - Create a Trip**

3.5.2 Edit a trip

Edit a trip use case starts with opening the app and choosing one of two ways of searching for a trip. One way is searching for a trip name/ID directly through the search bar, and another is by searching for a trip on the interactive map. After the trip is found, it is somewhat "copied" to the instance of the user. User can then edit every single bit of the trip - change the name, and add/remove destinations (again, destination removal is not shown in the graph due to simplicity). Finally the user can choose to save the trip without publishing it, or publish it so that everyone else is able to find it. Either way, the trip gets a brand new ID only if it has been changed in any way (changing only the trip name is not regarded as a change). Again, due to data safety, in either case it is stored online.

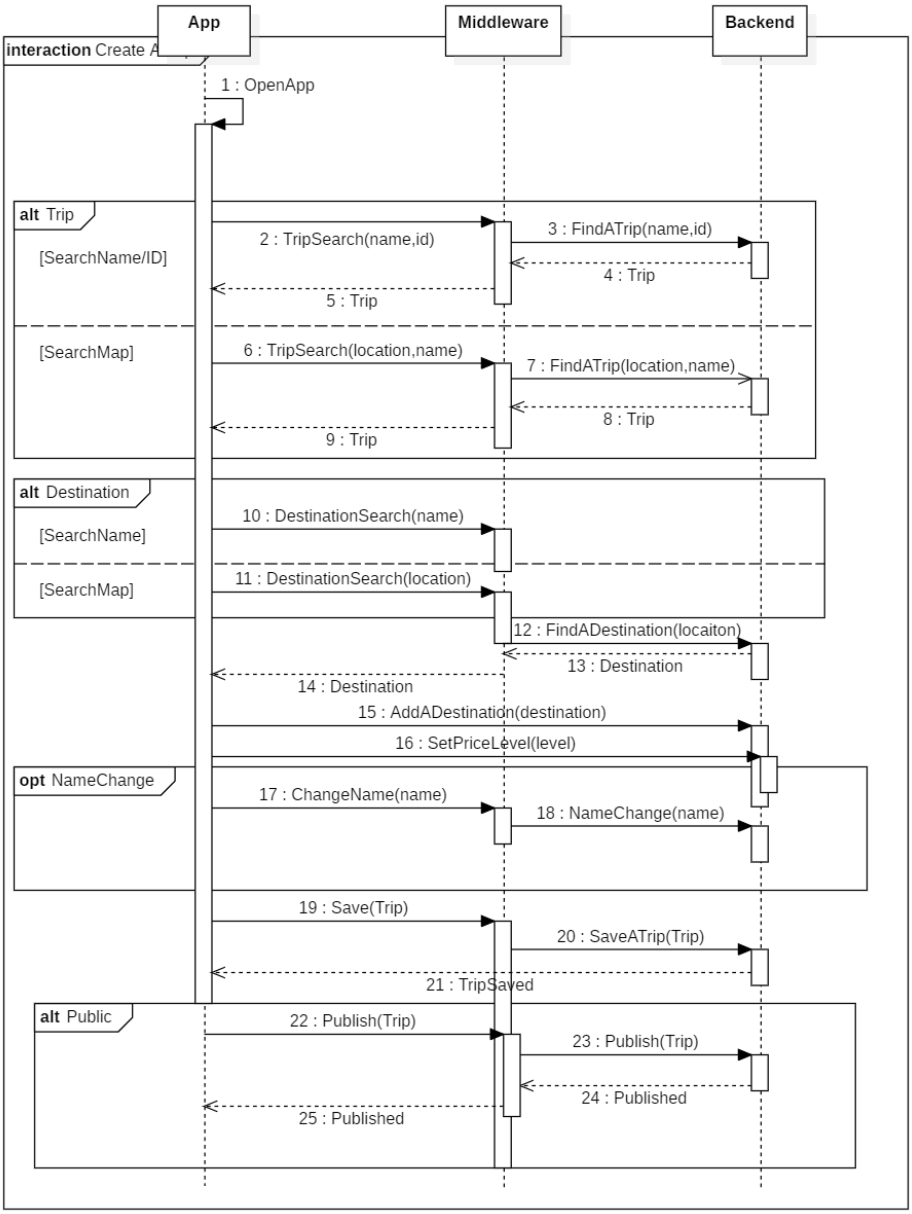

Figure 6: **Sequence diagram 2 - Edit a Trip**

3.5.3 Add a destination

Add a destination use case starts with opening the app and choosing one of two ways of adding destination - either by using the dynamic Google Maps API, or by using a search function which searches for specific places and locations in the Google database. By using a Google Maps way, the user can specify the exact location, while by using the search way, the user can pick of the previous existing locations in the database. After picking a location, the user can change a name and/or add an specific image of this location. This way the user is allowed to personalize locations for their trip needs. After saving all of the data in the form of a destination, the new destination is published and can be found by other users and added to their trips. There are no local and unpublished destinations.

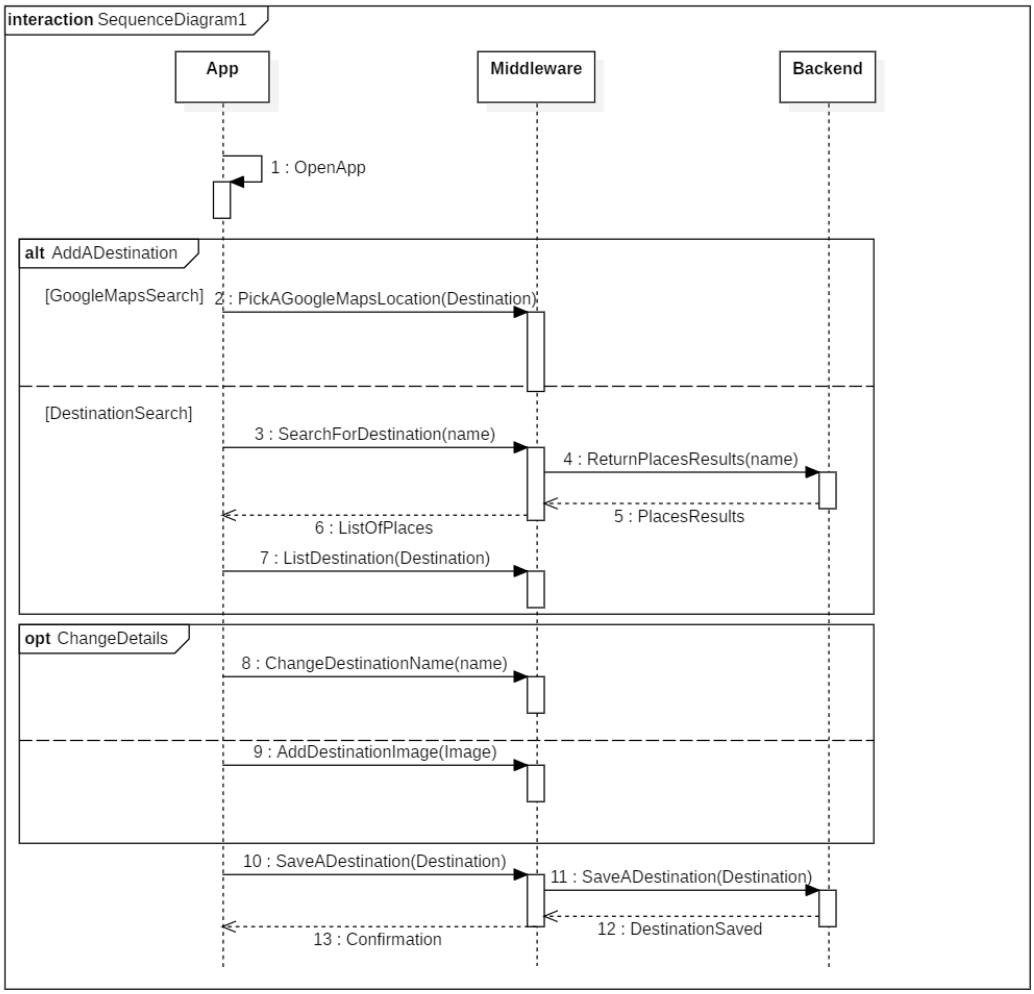

Figure 7: **Sequence diagram 3 - Add a destination**

3.5.4 Search for a trip/explore

There are several ways of searching for a trip which are going to be explained rather than shown in graphs, due to the complexity and confusion they would cause.

Search by name and ID

The app features a search bar right above the navigation bar, which allows users to search through the database of saved trips and destinations by name. Both destinations created by the users and Google Places destinations are featured, just like trips created by both the current user and all other users. Trips can also be searched by the unique ID that has been assigned to every public trip. The user can also share the link of a trip (via the trip screen) that can then be clicked from outside of the app, and via trip ID, is brought to the trip screen of a specific trip.

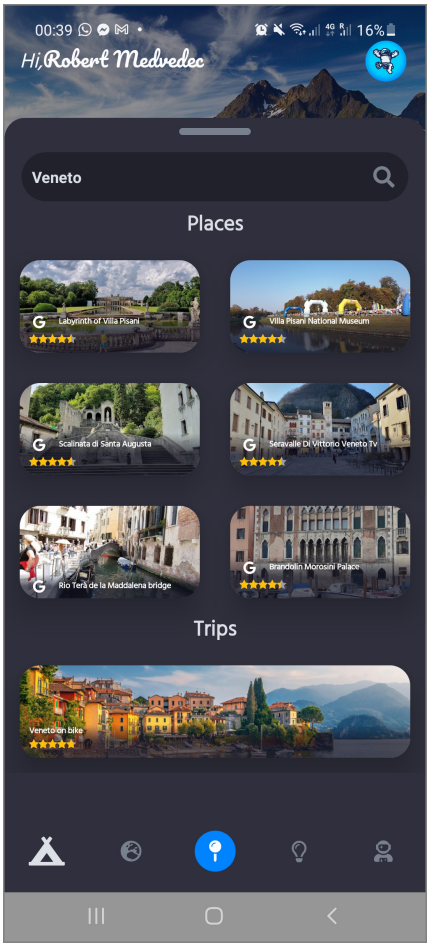

Figure 8: **Search by name and ID**

Search by map drawing

Map searching is the most interesting and interactive function of the app. It allows the user to focus on the specific area of the world on the app, circle the desired part of it with finger input, on which the map will generate interesting locations on the map in the form of bubbles. This can help users search through specific areas, which can be extremely useful when visiting unknown places or having a vacation in a new place.

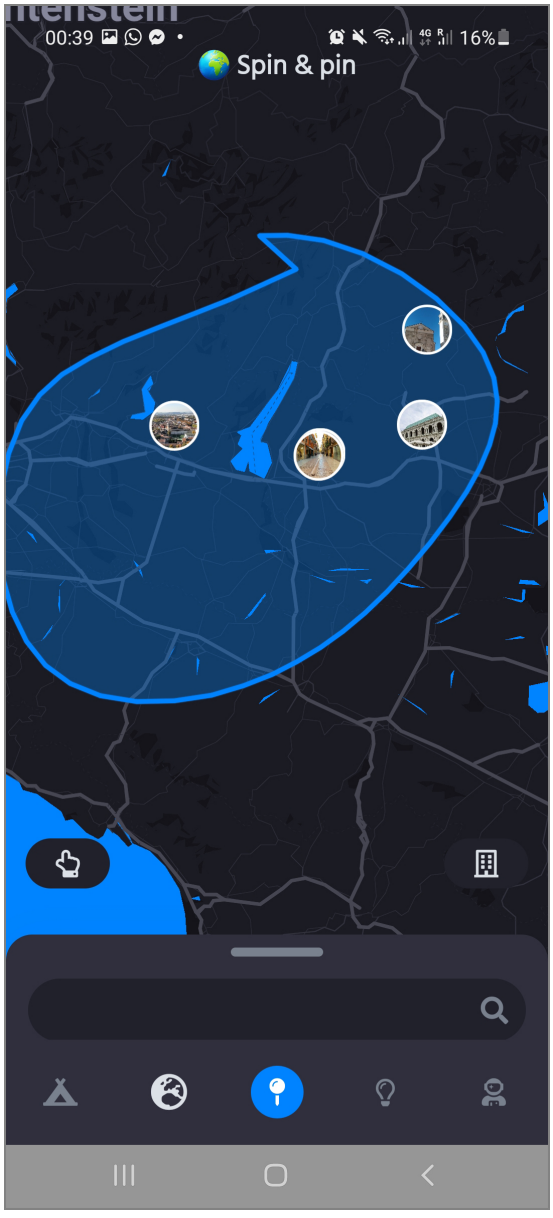

Figure 9: **Search by map drawing**

Search by location

Similar function to the previous one is used when searching based on the current location. When the user allows the app to access the location of the device, it provides it with interesting destinations in the radius around the device, which can range all the way from zero to several thousand kilometers.

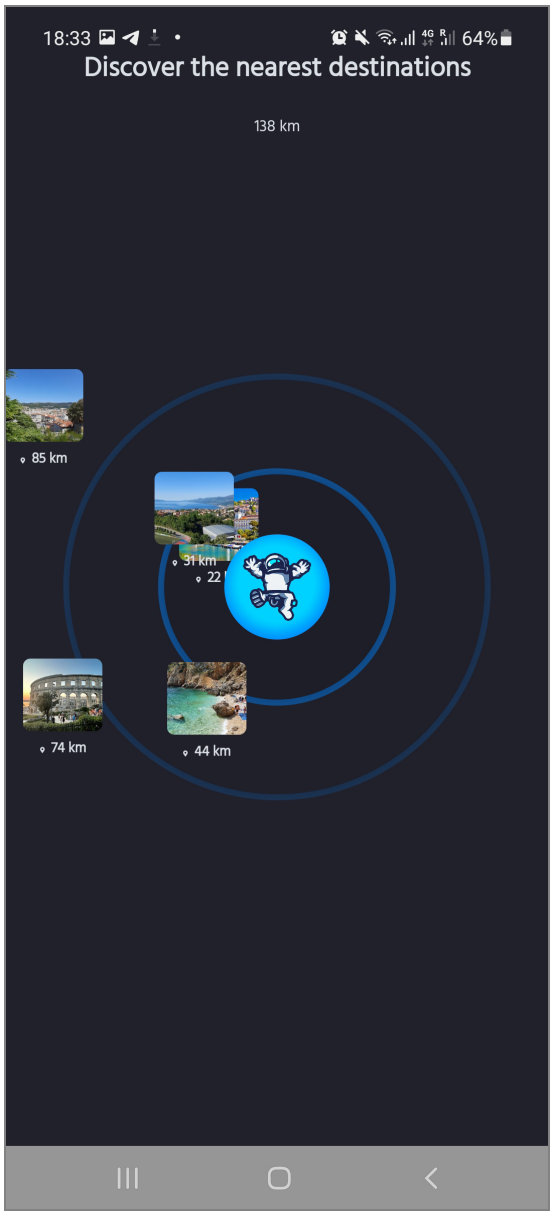

Figure 10: **Search by location**

Search on the wall

App also features a "wall" of top destinations and trips, based on an algorithm, that features the newest and the highest rated trips and destinations. Users can easily scroll through and pick whatever they find interesting. When searching through locations and cities, the app will fetch other points of interest in the vicinity and provide user with both trips and locations (which are downloaded from Google Places) in and around the area.

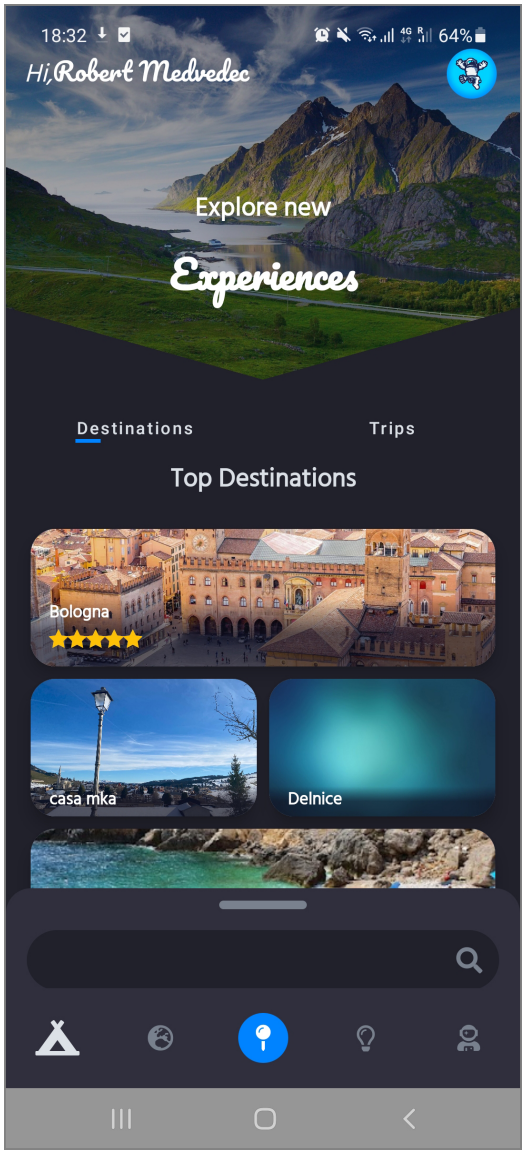

Figure 11: **Search by destination**

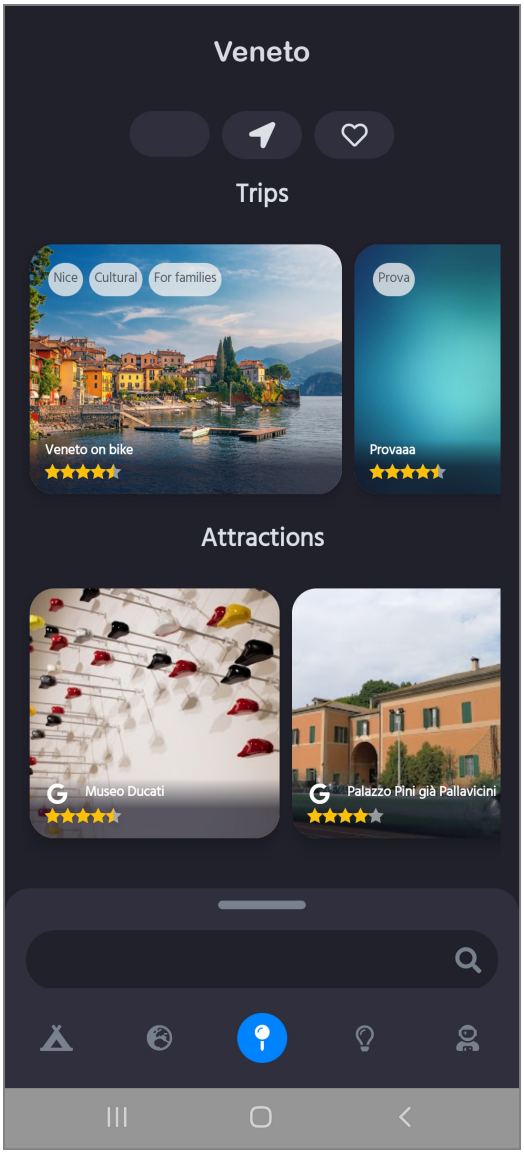

Figure 12: **Destination details**

3.6 Design constraints

3.6.1 Hardware limitations

As previously mentioned, the application is currently only made for Android OS, using native Android development. iOS devices, as well as Windows Mobile OS devices, and Web/Desktop version of the software is currently not available.

The application will only work on Android devices that support Android 7.0(Nougat) which is Android API 24 or above, due to security reasons, and due to specific software features that are not available in the previous versions of this software. As this version is over six years old, and vast majority (over 90%) of Android users are currently running this or higher version, we estimated this to be a viable choice for the lowest supported version. As of right now, the latest version of the Android OS is 12 or API level 31, which is also the target build for the app.

Device is use is required to have access to the Internet for the initial(and any subsequent) login, and for the initial data fetching. The Internet connection is not required when using the application in offline mode, after the desired trips have been downloaded and locally stored. In case of using the application without Internet connection, the full features of the app may not be available to the user.

3.6.2 Privacy limitations

The user is not required to insert any personal information in the app, besides their email or social media account which is necessary for login. This email will not be visible to the other users. The user may provide his real name and surname, which would make them easier to find by other users, although the primary identification will still be in the form of username, which is initially set by the user. User may also provide an image that will be used for the thumbnail of their account, which is also optional. User's comments on trips will be seen by everyone if the trip has been made public. If the user wants to keep the comments to themselves, it is possible to leave the trip unpublished.

4 User Interface Design

4.1 Identity and colours

The official name of the app is 'Polaris'. The app identity is an astronaut, with the idea of a user acting as an astronaut, which has an overview of the planet Earth and can just pick a place and travel wherever. This feature is also presented in one of the screens in the app, as user searches for already existing trips by searching the globe.

Figure 13: **'Polaris' icon featuring an astronaut on a blue background**

The main colours used for the app are two slightly different blues. Primary blue has the RBG value of #0083FF while secondary blue has the RBG value of #0460D9. They are used interchangeably throughout the app.

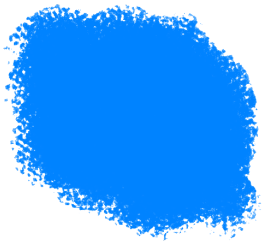

Figure 14: **Primary color #0083FF**

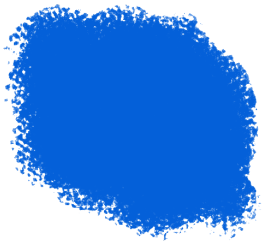

Figure 15: **Secondary color #0460D9**

As app features both light and dark mode, there are also colour palettes used for those instances. The same primary and secondary colours are used for accents, but for backgrounds and texts, the colours presented in the following two figures are used.

Figure 16: **Background colours for light theme**

Figure 17: **Background colours for dark theme**

4.2 User interface design

This section features the main design choices of the entire UI of the app, which includes navigation bars, menus, buttons, and other choices. It will not go into too much detail about every component, as all of the screens will be shown with a short comment about it.

The main part of the app is divided into four major components, with additional screens available for other functionalities, such as adding and editing destinations and trips. At the bottom of the app there is a navigation bar that leads the user to all of the main screens. Additionally, users can navigate to the settings/account screen either by pressing the 'Polaris' icon in the upper right corner of the screen, or by pressing the last icon in the bottom navigation bar. Right above the navigation bar, a search bar is located which allows the users to search for their previously saved trips.

As previously mentioned, the app features both light and dark themes. The following figures present the same screens for both of the themes.

4.3 Login screen

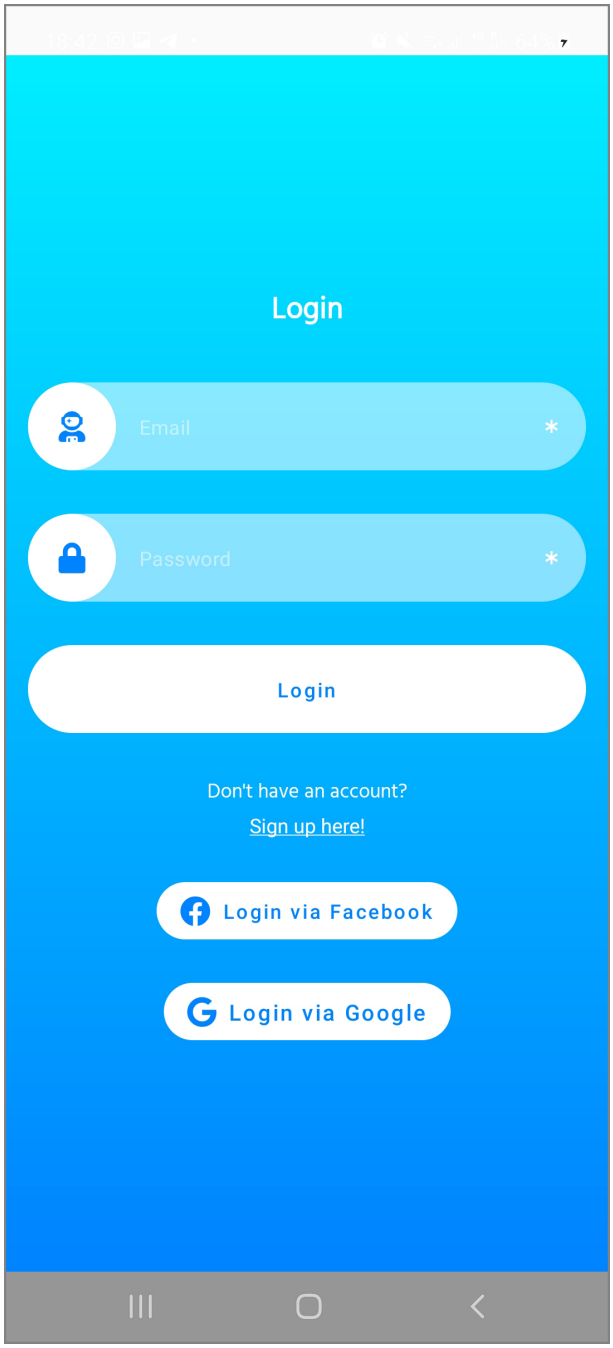

Figure 18: **Login screen**

Login screen provides users with the option of logging in or registering with email, Google account, or Facebook account. Successfully logging in leads the user to the Main screen of the app.

4.3.1 Main screen

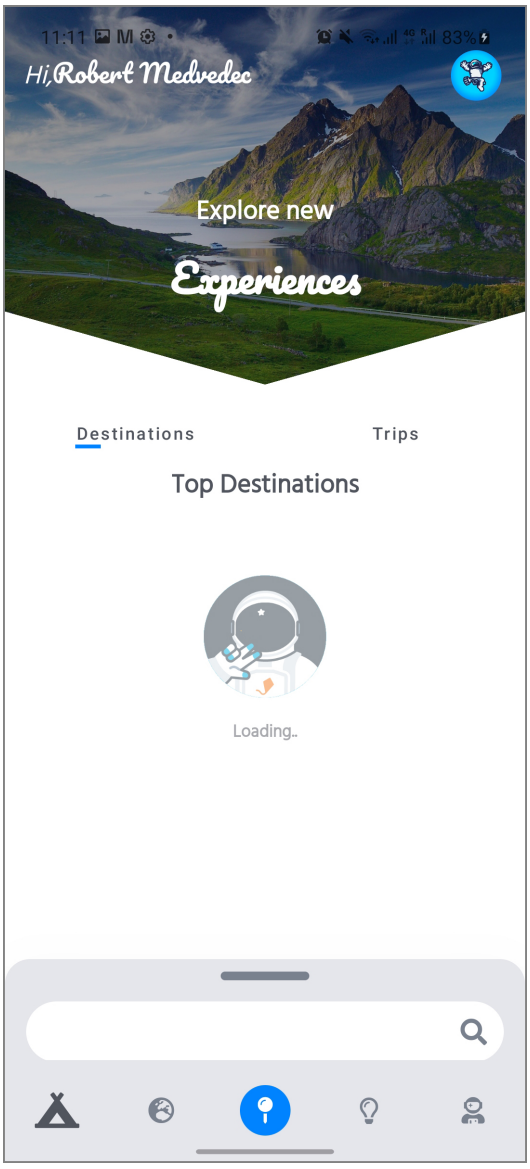

Figure 19: **Main screen in light style**

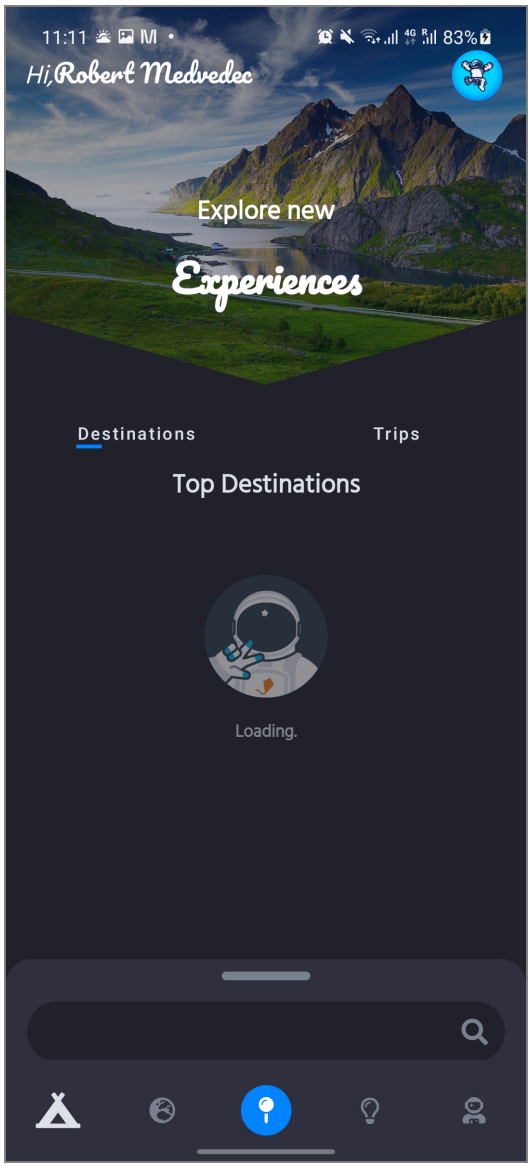

Figure 20: **Main screen in dark style**

Main screen provides user with the information about the existing destinations and trips. It offers the user a fast way to access some of the information about the locations that they might find useful to build on or to explore.

4.3.2 Map screen

Figure 21: **Map screen in light style**

Figure 22: **Map screen in dark style**

Map screen allows users to interactively go through the desired locations on the map and find small icons that represent trips in that area. Map takes into account the boundaries that are shown on the screen and actively adjusts the search parameters, fetching from the database only the trips that are in the current area. Map helps users explore certain areas when they are not sure which specific destination to center their trip around.

4.3.3 Saved trips screen

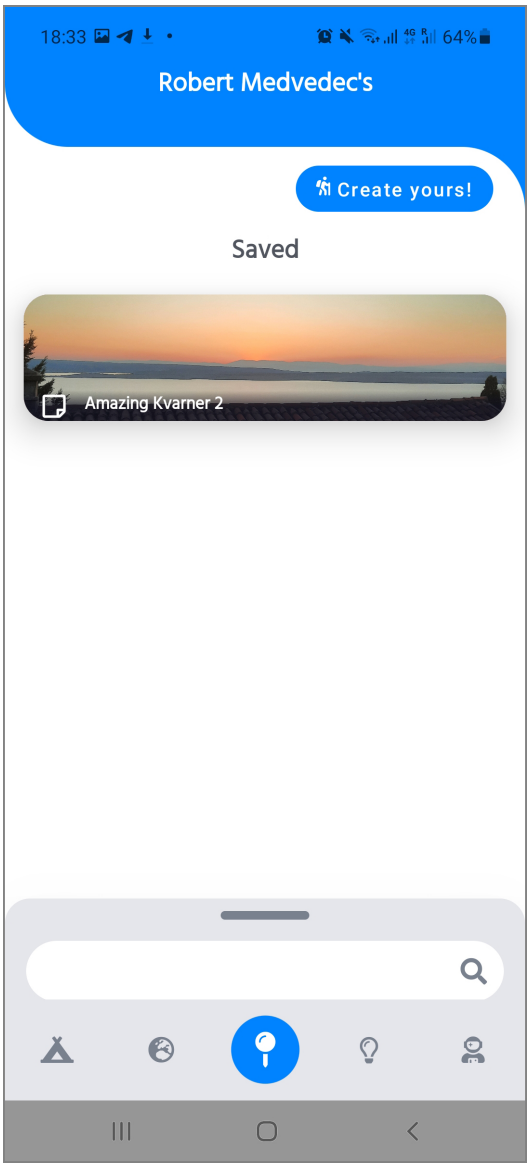

Figure 23: **Saved trips screen in light style**

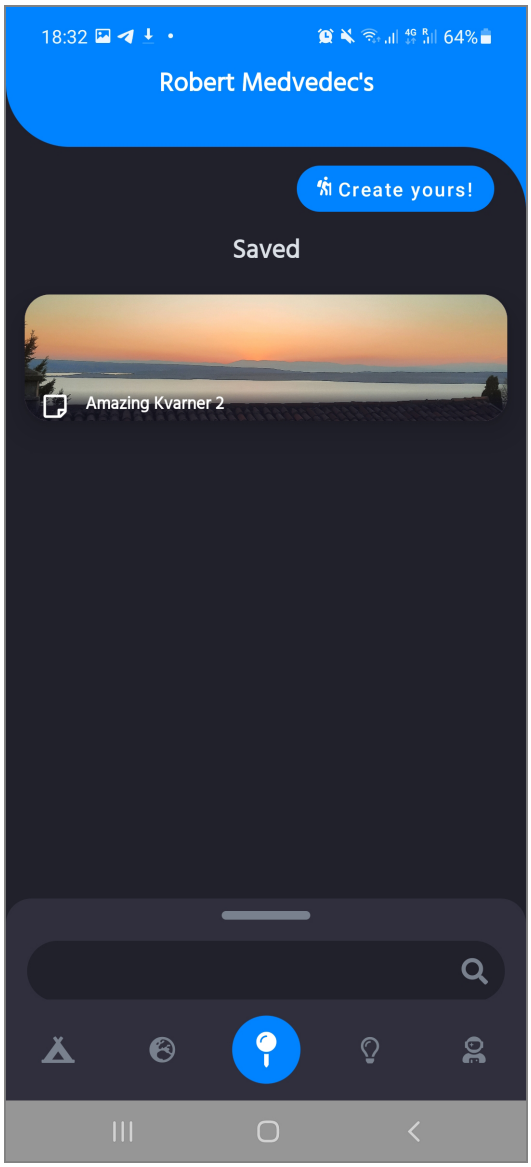

Figure 24: **Saved trips screen in dark style**

Saved trips screen holds the information of all of the trips the user has created or saved. It is also a quick way of finding older trips that users found interesting and allows them to organize their travels in an easy and effective way.

4.3.4 Explore screen

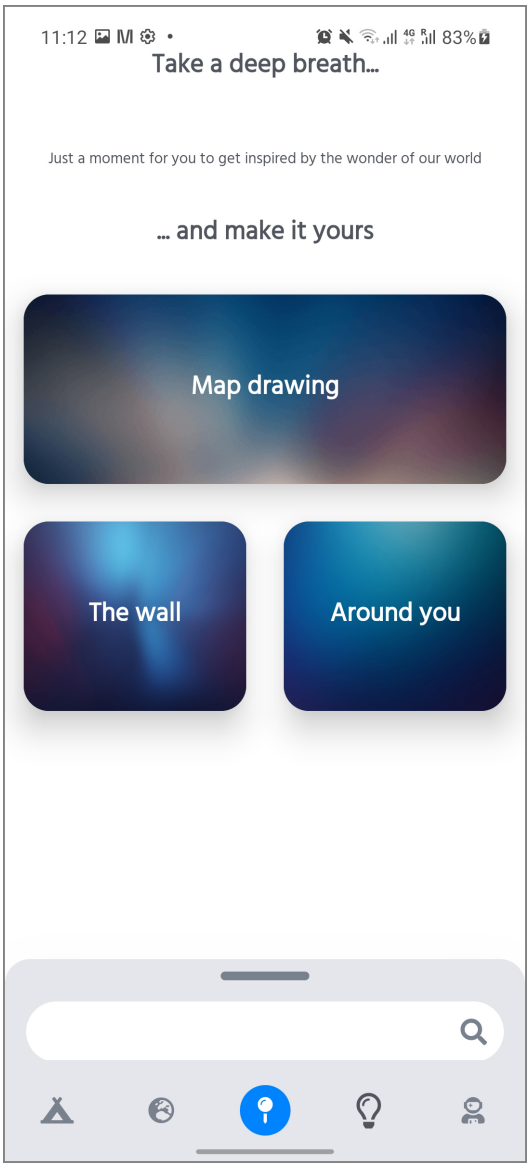

Figure 25: **Explore screen in light style**

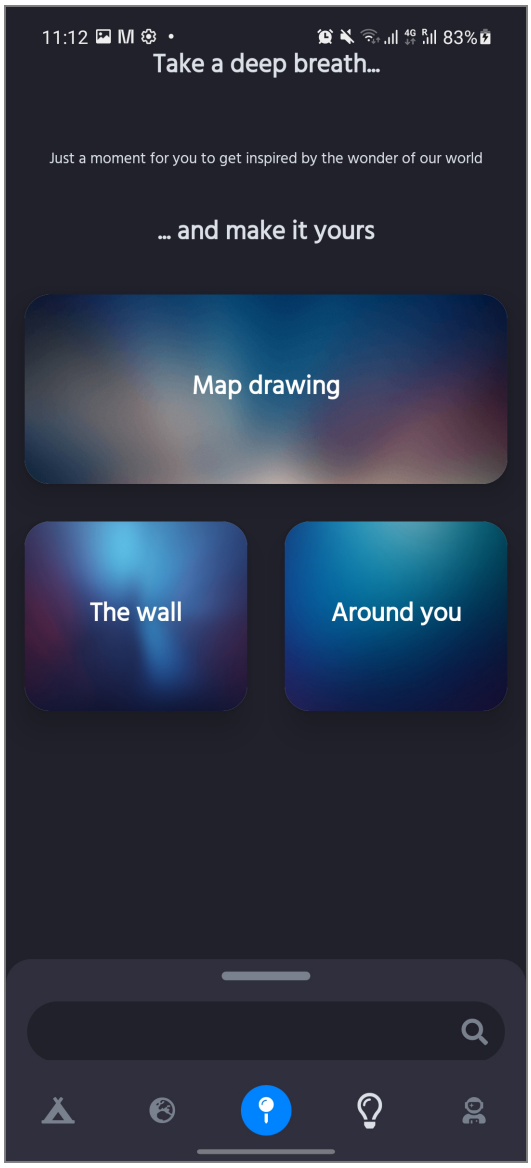

Figure 26: **Explore screen in dark style**

Explore screen allows users to search for their future trips in multiple ways. The first way is by searching on the map, and by clicking the 'Map drawing' button, the app takes the user to the previously explained screen.

The next option, 'The wall', - DOESN?T DO ANYTHING NOW.

Finally, 'Around you' button asks the user for location permission, and if granted, searches for interesting destinations and trips in the user's vicinity.

4.3.5 Settings screen

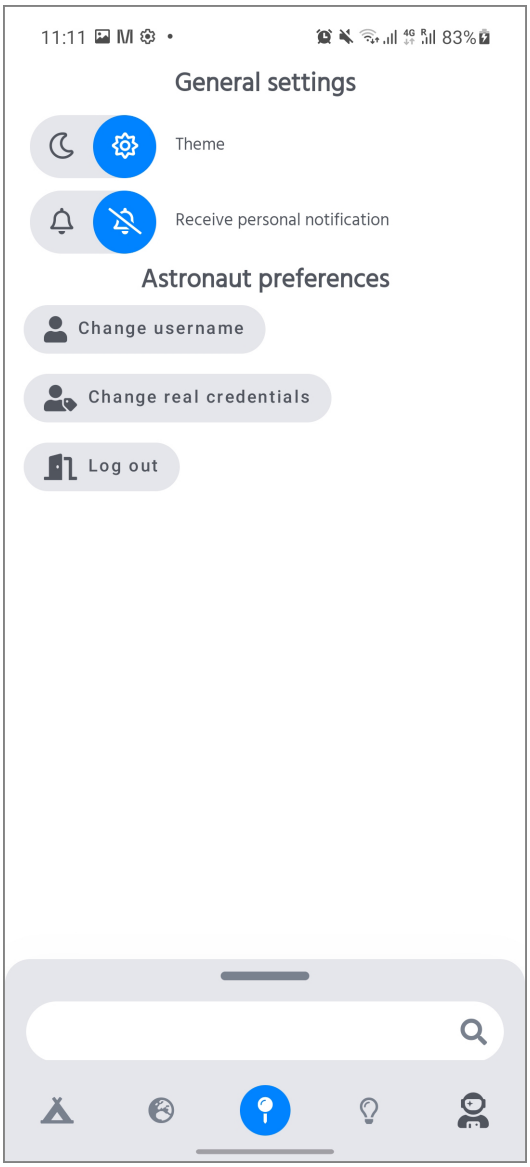

Figure 27: **Settings screen in light style**

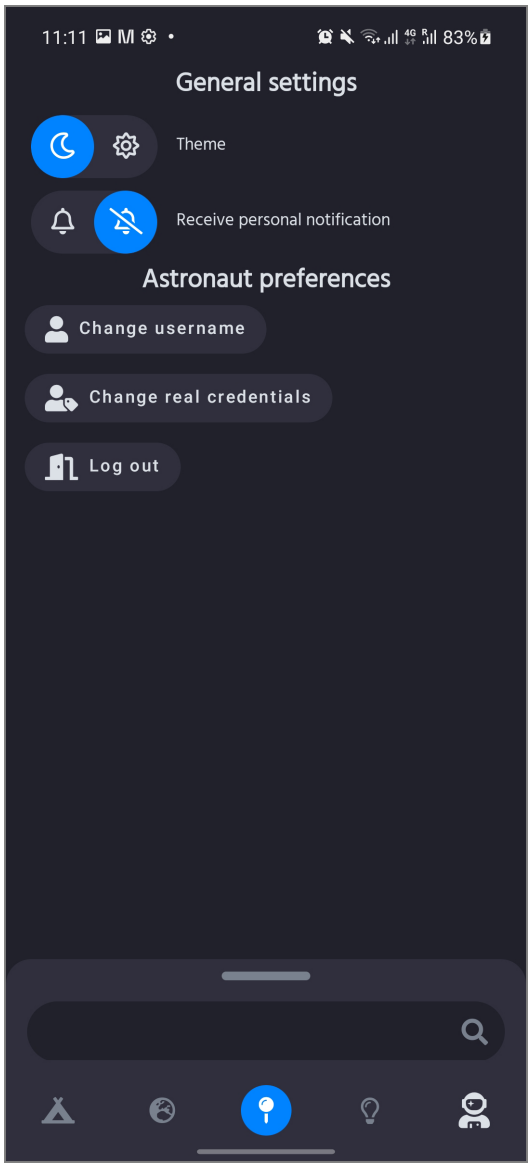

Figure 28: **Settings screen in dark style**

Settings screen allows users to change between dark and light theme, to turn the notifications on or off, and to change user preferences such as username and real credentials, which are optional. It also features an option to log out, if user ever wishes to change their account.

4.3.6 Trip creation screen

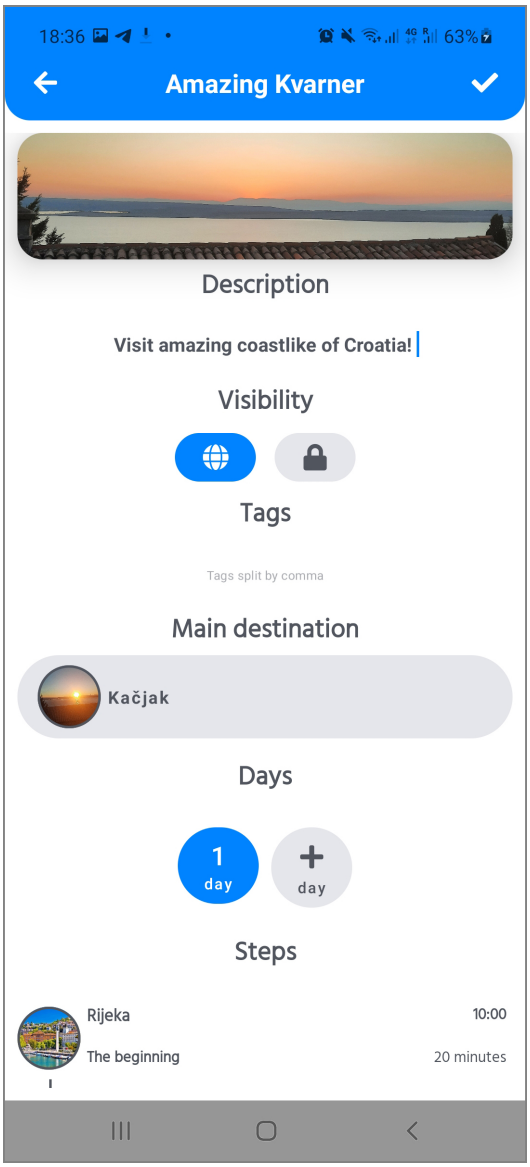

Figure 29: **Trip creation screen in light style**

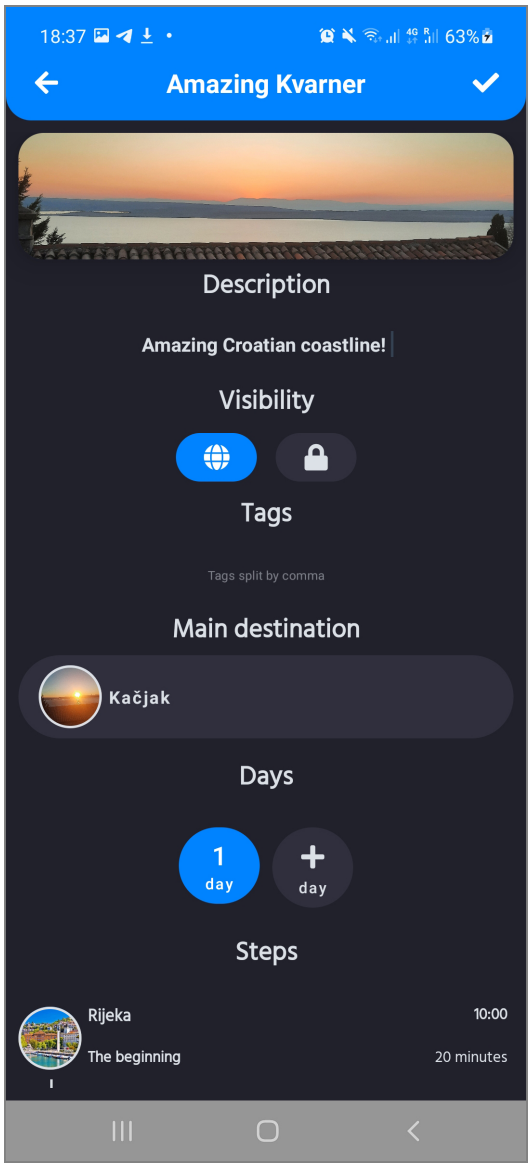

Figure 30: **Trip creation screen in dark style**

This is the way the trip creation looks like, with its functionality previously explained in the document. This screen also features several sub screens, which are not presented in the document due to conciseness.

5 Testing

Several test types have been done for the app. Most of the testing was done during the app development, so further Unit testing was reduced as most of it was done before.

5.1 Unit testing

The main part of the testing was done by unit testing. Four major unit tests have been created that test functionalities of the main parts of the app. Each main test has several subtests. For code conciseness purposes main tests are not a set of small tests, but just one large test that tests multiple different parts of the subsystem.

When counting the total number of tested subcomponents, we have done approximately XY small tests, separated into four major groups, which are:

- Login tests (7)
- Trip Creation Activity tests (12)
- Navigation tests (6)
- Map screen tests (5)

These tests are now going to be showcased and presented.

5.1.1 Login unit testing

The tests done for login testing:

- Inputting username
- Inputting password
- Testing login
- Testing login with false authentication
- Testing login with correct authentication
- Logging in with Google Account
- Logging in with Facebook acocunt

The code:

```
@RunWith(AndroidJUnit4::class)
class LoginTest : TestCase() {

    @OptIn(
        ExperimentalFoundationApi::class,
    kotlinx.coroutines.ExperimentalCoroutinesApi::class,
        androidx.compose.animation.ExperimentalAnimationApi::class,
        androidx.compose.material.ExperimentalMaterialApi::class
    )
    @get:Rule
    val composeTestRule = createAndroidComposeRule<LoginActivity>()
```



```
    @OptIn(
        ExperimentalCoroutinesApi::class,
        androidx.compose.foundation.ExperimentalFoundationApi::class,
        androidx.compose.animation.ExperimentalAnimationApi::class,
        androidx.compose.material.ExperimentalMaterialApi::class
    )
    @Test
    fun checkLogin() {
        val login = composeTestRule.onNodeWithTag("login")
        val usr = composeTestRule.onNodeWithTag("username")
        val pwd = composeTestRule.onNodeWithTag("password")
            usr.performTextInput("vins@vins.com")
            pwd.performTextInput("vins")
            login.performClick()
        composeTestRule.onNode(hasText("Authentication failed"))
        assertFalse(composeTestRule.activity.isNewUser ?: false)
        pwd.performTextClearance()
        pwd.performTextInput("vinsvins")
        login.performClick()
        Thread.sleep(5000)
        getInstrumentation().runOnMainSync {
            val activity =
                ActivityLifecycleMonitorRegistry.getInstance().getActivitiesInStage(
                    Stage.RESUMED
                ).first()
            assertTrue(activity is MainActivity)
        }
    }
}
```

5.1.2 Trip creation activity unit testing

The tests done for trip creation activity testing:

- Tests whether the trip can be created correctly.
- The following elements of the trip creation screen are tested:
- Inputting text to name field
- Inputting text to description field
- Inputting multiple tags
- Searchig destinations with city name
- Adding steps
- Adding main destinations
- Exiting without saving
- Exiting with saving

- Trying to save trip without filling all the required fields
- Failing to save trip

```
-@RunWith(AndroidJUnit4::class)
class TripCreationActivityTest : TestCase() {
    @get:Rule
    val composeTestRule = createAndroidComposeRule<TripCreationActivity>()
    @OptIn(ExperimentalTestApi::class)
    @Test
    fun create() {
        composeTestRule.onNodeWithTag("name").performTextInput("Nome")
        composeTestRule.onNodeWithTag("description").
        performTextInput("Description")
        val tag = composeTestRule.onNodeWithTag("tag")
        tag.performTextInput("tag1")
        tag.performTextInput(",")
        tag.performTextInput("tag2")
        tag.performTextInput(",")
        composeTestRule.onNodeWithTag("main").performClick()
        Thread.sleep(1000)
        composeTestRule.onNodeWithTag("searchText").
        performTextInput("verona")
        composeTestRule.onNodeWithTag("searchText").
        performImeAction()
        Thread.sleep(4000)
        composeTestRule.
        onAllNodes(hasText("Verona",true,true)).
        onFirst().assertExists()
        val addStep = composeTestRule.onAllNodesWithTag("add").onFirst()
        addStep.assertExists()
        addStep.performClick()
        val setMain = composeTestRule.onAllNodesWithTag("main").onFirst()
        setMain.assertExists()
        setMain.performClick()
        composeTestRule.onNodeWithTag("back").performClick()
        Thread.sleep(500)
        val nodes = composeTestRule.onAllNodes(hasText("verona",true,true))
        assertNotNull(nodes)
        nodes.assertCountEquals(2)
        composeTestRule.onNodeWithTag("confirm").performClick()
        Thread.sleep(2000)
        InstrumentationRegistry.getInstrumentation().runOnMainSync {
            val activity =
                ActivityLifecycleMonitorRegistry.getInstance().
            getActivitiesInStage(
                    Stage.DESTROYED
                ).first()

            assertTrue(activity is TripCreationActivity)
```



```
        }
    }

    @Test
    fun creationFailure() {
        for (i in 0..3) {
            if (i == 0)
                composeTestRule.onNodeWithTag("name").
            performTextInput("Nome")
            if (i == 1)
                composeTestRule.onNodeWithTag("description")
            .performTextInput("Description")
            if (i == 2) {
                val tag = composeTestRule.onNodeWithTag("tag")
                tag.performTextInput("tag1")
                tag.performTextInput(",")
                tag.performTextInput("tag2")
                tag.performTextInput(",")
            }
            if (i == 3) {
                composeTestRule.onNodeWithTag("main").performClick()
                Thread.sleep(1000)
                composeTestRule.onNodeWithTag("searchText").
                performTextInput("verona")
                composeTestRule.onNodeWithTag("searchText").
                performImeAction()
                Thread.sleep(4000)
                composeTestRule.onAllNodes(hasText("Verona", true, true)).onFirst().assertEx
                val addStep = composeTestRule.onAllNodesWithTag("add").onFirst()
                addStep.assertExists()
                addStep.performClick()
                val setMain = composeTestRule.onAllNodesWithTag("main").onFirst()
                setMain.assertExists()
                setMain.performClick()
                composeTestRule.onNodeWithTag("back").performClick()
                Thread.sleep(500)
                val nodes = composeTestRule.onAllNodes(hasText("verona", true, true))
                assertNotNull(nodes)
                nodes.assertCountEquals(2)
            }
            composeTestRule.onNodeWithTag("confirm").performClick()
            Thread.sleep(1000)
            val text = composeTestRule.activity.getString(R.string.not_enough_info)
            if (i != 3)
                onView(withText(text)).check(matches(isDisplayed()))
            Thread.sleep(2000)
        }
    }
}
```


5.1.3 Navigation unit testing

The tests done for trip creation activity testing:

- Navigation of the bottom tab
- Navigation of home screen
- Navigation of map screen
- Navigation of trip screen
- Navigation of explore screen
- Navigation of profile screen

```
@RunWith(AndroidJUnit4::class)
class NavigationTest : TestCase() {
    @get:Rule
    val composeTestRule = createAndroidComposeRule<MainActivity>()

    @Test
    fun navigation() {
        for (i in 0..6) {
            composeTestRule.onNodeWithTag("tab" + nextInt(0, 4)).performClick()
            Thread.sleep(3000)
        }
        composeTestRule.onNodeWithTag("tab4").performClick()
        assertNotNull(composeTestRule.onNode(hasText("Astro",true,true)))
        composeTestRule.onAllNodesWithTag("positive").onFirst().performClick()
        Thread.sleep(2000)
        composeTestRule.onAllNodesWithTag("negative").onFirst().performClick()

        composeTestRule.onNodeWithTag("tab3").performClick()

        composeTestRule.onNodeWithTag("around").performClick()
        Thread.sleep(3000)
        getInstrumentation().runOnMainSync {


            val activity =
                ActivityLifecycleMonitorRegistry.getInstance().
                getActivitiesInStage(
                    Stage.RESUMED
                ).first()

            assertTrue(activity is AroundMeActivity)
            // seems to not work in Debug test mode
            // works in release
        }
    }
}
```


5.1.4 Map screen unit testing

The tests done for trip creation activity testing:

- Navigate to tab
- Draw a polygon in the map tab
- Retrieve results of the drawn polygon
- Selection of the shown destination
- Detail displaying of the destination

```
@RunWith(AndroidJUnit4::class)
class MapScreenTest : TestCase() {


    @get:Rule
    val composeTestRule = createAndroidComposeRule<MainActivity>()

    @OptIn(ExperimentalTestApi::class)
    @Test
    fun checkSelection() {
        composeTestRule.onNodeWithTag("tab1").performClick()
        Thread.sleep(1000)
        composeTestRule.onNodeWithTag("draw").performClick()
        Thread.sleep(1000)
        val device = UiDevice.getInstance(getInstrumentation())


        // It requires to disable animations by phone settings
        // + declare injection_event permission
        // For the release, this requirements have been turned off
        try {
            composeTestRule.onRoot().performGesture {
                down(Offset(50f,50f))
                moveBy(Offset(500f,500f))
                moveBy(Offset(500f,0f))
                moveBy(Offset(-500f,500f))
            }
            composeTestRule.onNodeWithTag("draw").performClick()
        }
        catch (e: Exception) {
            Log.e("TEST",e.localizedMessage)
        }
        val marker = device.findObject(UiSelector().descriptionContains("Venice"))
        marker.click()
    }

}
```


5.2 Deep linking testing

Deep linking testing has been done by providing the application with three different types of links - the correct type of link accepted by the application (schema, host, and attribute part are accepted), semi-correct type of link (schema and host are accepted, the attribute is not), and the correct link with false attribute value.

For testing purposes, Android Debug Bridge (adb) and manual link clicking have been used. Here are the results of several tests done via adb:

Test 1

```
adb shell am start
>adb shell am start -W -a android.intent.action.VIEW
Starting: Intent { act=android.intent.action.VIEW }
Status: ok
LaunchState: COLD
Activity: android/com.android.internal.app.ResolverActivity
TotalTime: 457
WaitTime: 527
Complete
```

Test 2

```
>adb shell am start -W -a
android.intent.action.VIEW -d "https://polaris.travel.app/find/tripID"
Starting: Intent { act=android.intent.action.VIEW dat=https://polaris.travel.app/... }
Status: ok
LaunchState: WARM
Activity: android/com.android.internal.app.ResolverActivity
TotalTime: 251
WaitTime: 258
Complete
```

Test 3

```
>adb shell am start -W -a
android.intent.action.VIEW -d "https://polaris.travel.app/find/tripID"
Starting: Intent { act=android.intent.action.VIEW dat=https://polaris.travel.app/... }
Status: ok
LaunchState: COLD
Activity: web/android.default.webApp
TotalTime: 251
WaitTime: 258
```

The tests that have correct deep links have opened the Polaris activity, while the link with incorrect link has opened the link directly in the browser.

5.3 Failure test

We have done a process of a failure test when if comes to the app functioning in both online and offline mode. As the app features both states, it is important that the app works in offline mode properly even when there is no Internet connection, and that it restores to a proper flow once the Internet connection is present.

This is the following set of actions done for this test:

1. Shut down the server
2. Enter the app and navigate it with internet connection enabled
3. Check the app doesn't crash and gracefully alerts the user of the problem
4. Put the server up
5. Check everything is correctly received
6. Change endpoint
7. Redo points 2. and 3.
8. Put the server in a inconsistent status of responses
9. Redo points 2. and 3.
10. Restore server in a consistent configuration
11. Disabling local connectivity (offline status)
12. Redo points 2. and 3.

All of the tests regarding failure have been passed successfully.

5.4 Support of different devices

The app has been tested on several different device types and screen sizes. The testing has been done either on real world devices or on the virtual devices, ranging from Android 8 to Android 11. The list of the devices the app has been tested is the following:

- Samsung Galaxy S6
- Samsung Galaxy A50
- Samsung Galaxy A51
- Google Pixel 3a
- Samsung Galaxy Tablet S5 (Tablet)
- Google Nexus 7 (Tablet)
- Google Pixel C (Tablet)

Here are a few screenshot examples of the same screen working on devices of two different sizes (phone and tablet):

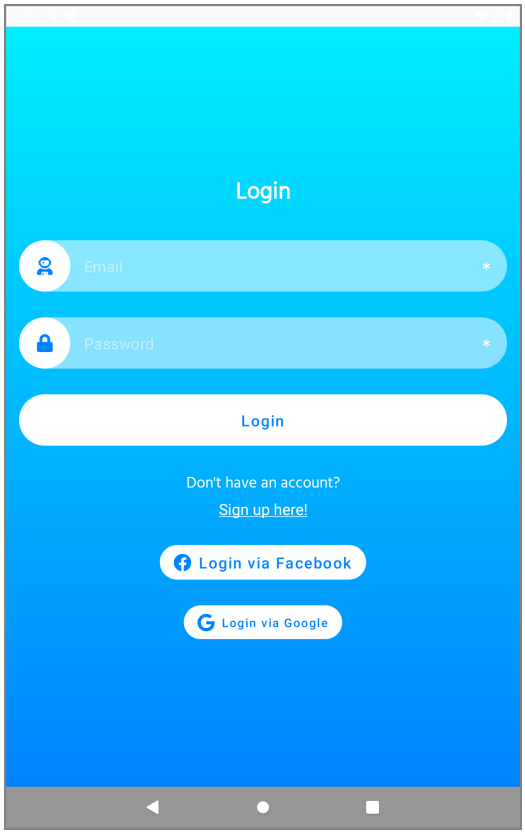

Figure 31: **Login screen - Tablet**

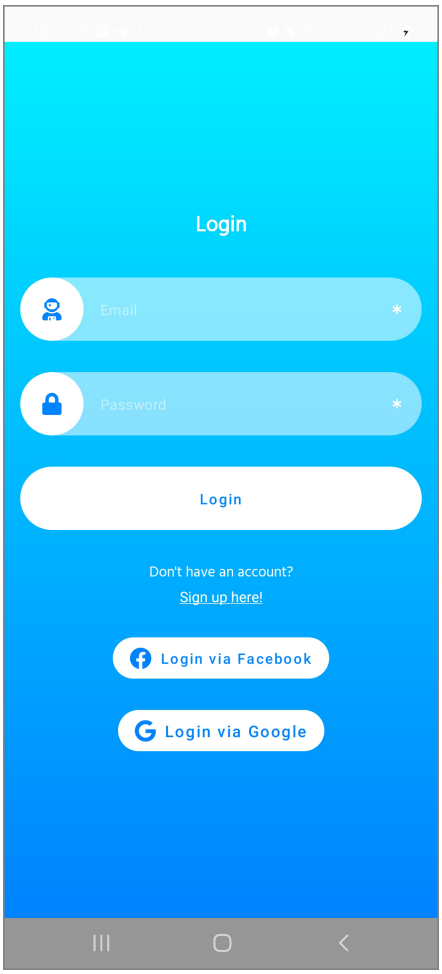

Figure 32: **Login screen - Phone**

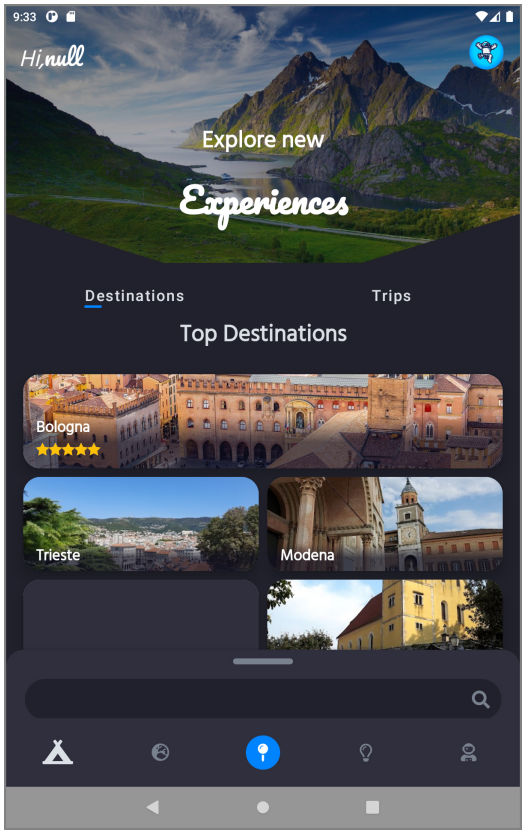

Figure 33: **Home screen - Tablet**

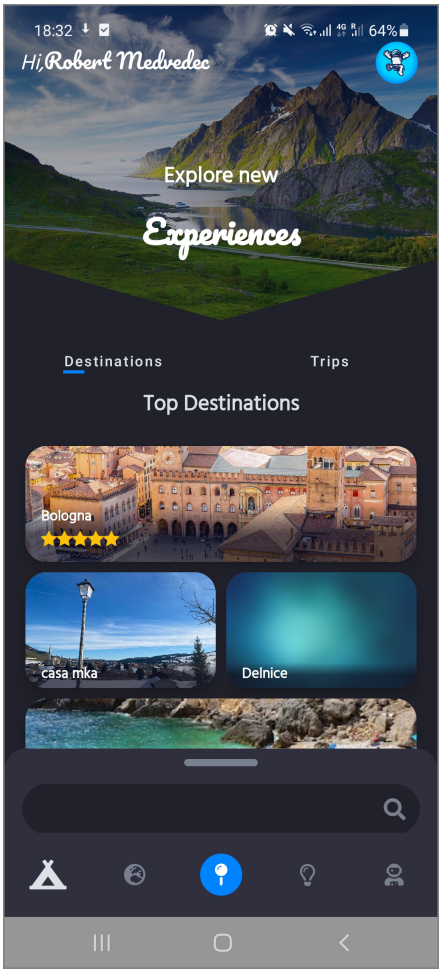

Figure 34: **Home screen - phone**

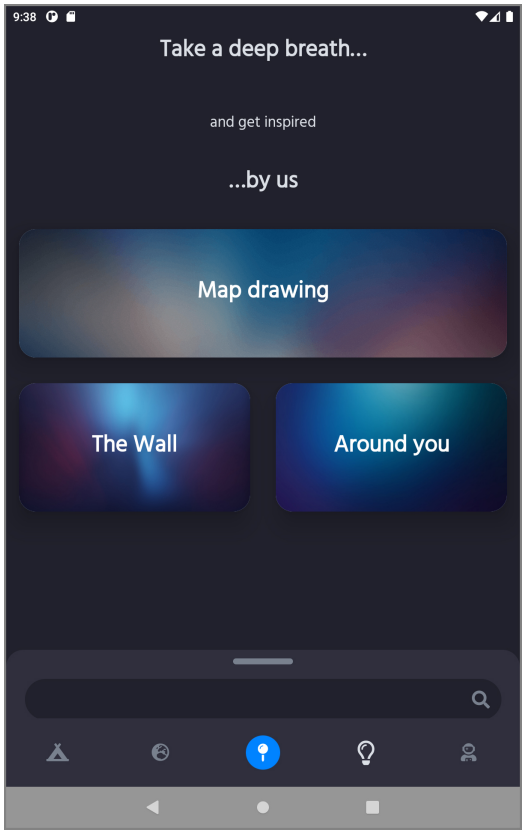

Figure 35: **Explore screen - Tablet**

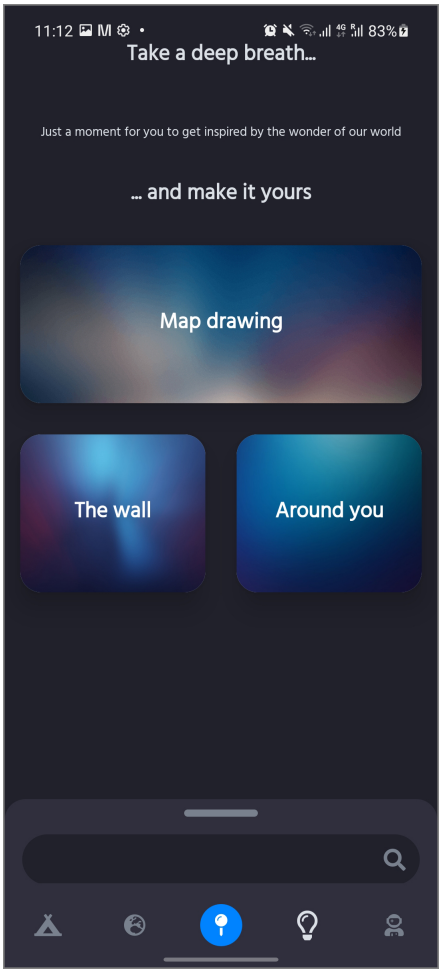

Figure 36: **Explore screen - Phone**

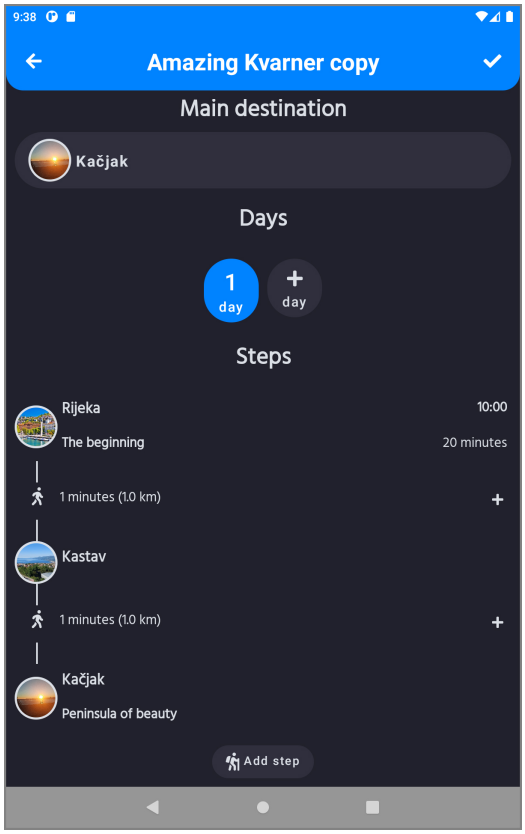

Figure 37: **Trip create screen - Tablet**

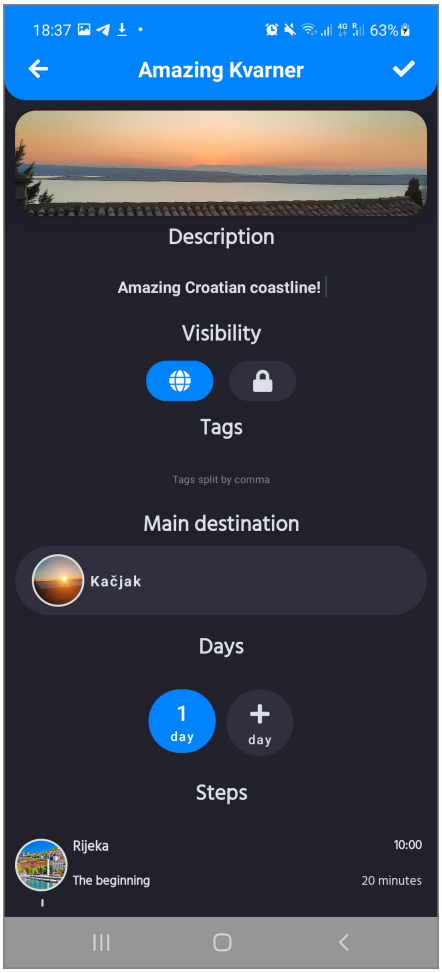

Figure 38: **Trip create sreen - Phone**

