Escola Politècnica Superior d'Enginyeria de Manresa
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Missing Persons App

An Android app for missing people

Submitted by: Pol Moreno

Date: 2/06/2018

Supervisor: Dr. Francisco del Águila

Bachelor’s Degree in ICT Systems Engineering

Table of Contents

List of Figures

List of Tables

Abstract

During the last years, the smartphone industry has really advanced by leaps and bounds. In 2017, the number of smartphone users was 2.4 billion, with an increase of 10,8% respecting the year before. As we can see, there is a huge market, and more people get used to use the mobile phone to do daily things such as online shopping, visiting webpages or sending emails.

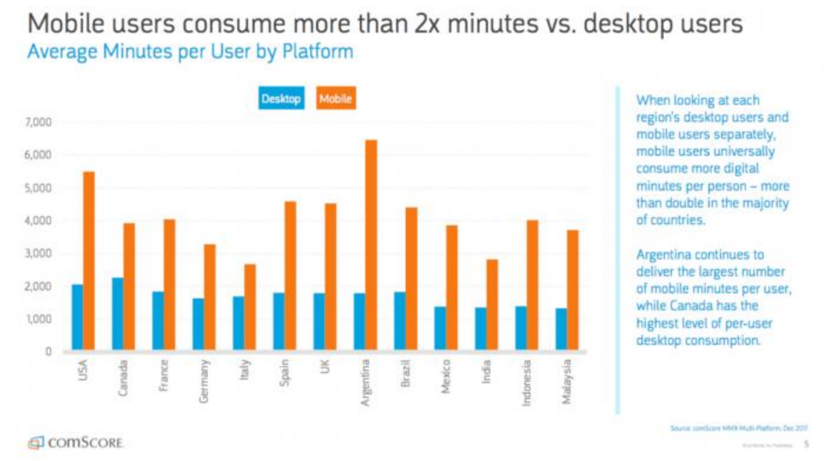
The following project covers an aspect that very few people worked in, the people who disappeared. Nowadays, the data of disappeared people has not been digitized, or if it has, is outdated in the current times.

The aim is to create an Android application for disappeared people. The app will collect and share information among users, whose use will be helping in the research of people who disappeared. Also, the idea is to offer a database and allow people to access data easily.

In this document is explained all the process of making the app, starting from analysing similar apps in the actual market, all the technologies that were used (from backend to frontend) and in which order was the app made. Finally, we will test the app and see how it works. Also, a manual for the backend and the app will be attached.

1. Introduction

There was an incredible increase of smartphones during the last years. According to a survey in 2015, most of the countries of Europe there is a percentage of 50% or more users that has an smartphone. Also, there is a tendency that says that users are replacing desktop PCs for smartphones when we are doing daily things such as visiting web pages, check for new email or chatting. Statistics say that mobile users consume more than 2x minutes vs desktop users.



When looking at each region's desktop users and mobile users separately, mobile users universally consume more digital minutes per person – more than double in the majority of countries.

Argentina continues to deliver the largest number of mobile minutes per user, while Canada has the highest level of per-user desktop consumption.

Figure 1: Graph of utilization between desktop devices and mobile phone

Consequently, mobile apps have also been increased. If we check Play Store (the app market of Android devices) in March 2018 it contains 3.600.000 apps. In the other side, App Store (the app market of Apple devices) in January 2017 has reached 2.220.000 apps.

1.1. Objectives

The main objectives of the app are:

- Make an app from scratch that allows mobile users help in the research of disappeared people
- Make a database (collecting data from users) from people who is disappeared
- Make a web service that allows users to get information from the database
- Understand the programming and general logic of a mobile application
- Make a 1.0 version that includes
    - Full map of the Earth and display, using markups, the location of disappeared people in live (update depending on the zone that is looking the user)
    - When a user clicks on the markup, show all the information
    - Let users add people

1.2. Choosing the operating system

In this section we will analyse the different operating systems that we have on mobile devices and see which one that suits best for our needs.

1.2.1. Windows Phone

Windows Phone it's a mobile operating system developed by Microsoft. It uses a user interface called Metro design language. It's built in C and C++ and released in November 10, 2010 in North America and October 21, 2010 in the rest of the world. Actually, it's discontinued.

Figure 2: Logo of Windows Phone

1.2.2. iOS

iOS is a mobile operating created and developed by Apple. It’s built in C,C++, Objective-C, Swift and it’s Unix based. As a matter of fact, only the devices that are from Apple can run this OS, and these devices are iPhone, iPad and iPod Touch.

Figure 3: Logo of iOS

1.2.3. Android

Android is the mobile operating system, now managed and maintained by Google. It’s based on a modification of the Linux kernel. It’s open source and, as iOS, it’s Unix based. Google bought Android in 2005 to *Android Inc*, and in 2008 it released the first version (Android 1.0). Actually, Android can be run in smartphones, tablets, smart TVs (Android TV), cars (Android Auto) or smartwatches (Wear OS).

Figure 4: Android logo

1.2.4. General overview

In this section we will analyse all the OS and see which one is the best for our needs. First of all, we will compare the market share of all mobile phone operating systems. According to a study, in April 2018, Android is installed in the 75,66% of the devices, followed by iOS with 19,23% and Windows Mobile with 0.54%.

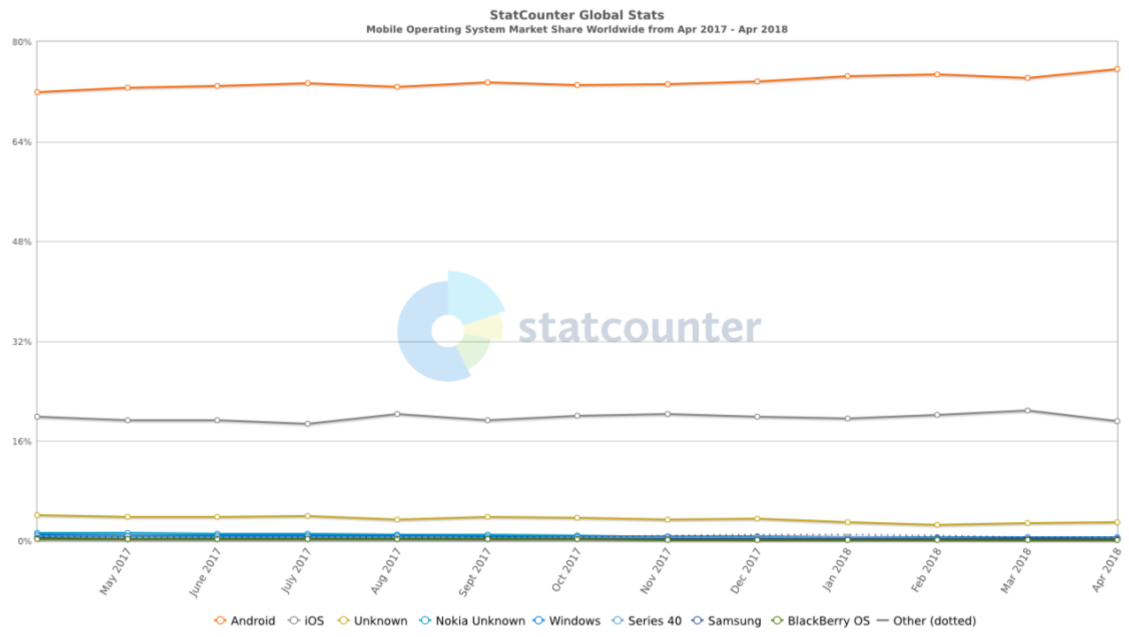


Figure 5: Graph containing the market share

As we can see, Windows Phone has a low market share, and also in September 2017 Microsoft decided to end support. Now we have to choose between Android and iOS.

Referring to the graph above, Android is the one that predominates in the mobile scenario, so I've chosen Android, and not only for the market share, but also for the following reasons:

- The tools for programming an Android app are in the most common desktop operating systems (macOS, Windows, Ubuntu…). In the case of iOS, we need the latest version of macOS with XCode.
- We will program in Java, which is very similar to C, that I've used in all of my career. If I have to program for iOS, I have to learn Swift. Also, it's one of the most used object oriented languages in the industry.

1.3. Android

Android is an open-source, Linux-based software created for different types of devices. Now in this section we will explain how it works Android.

1.3.1. Platform architecture

Android by default is structured in different layers:

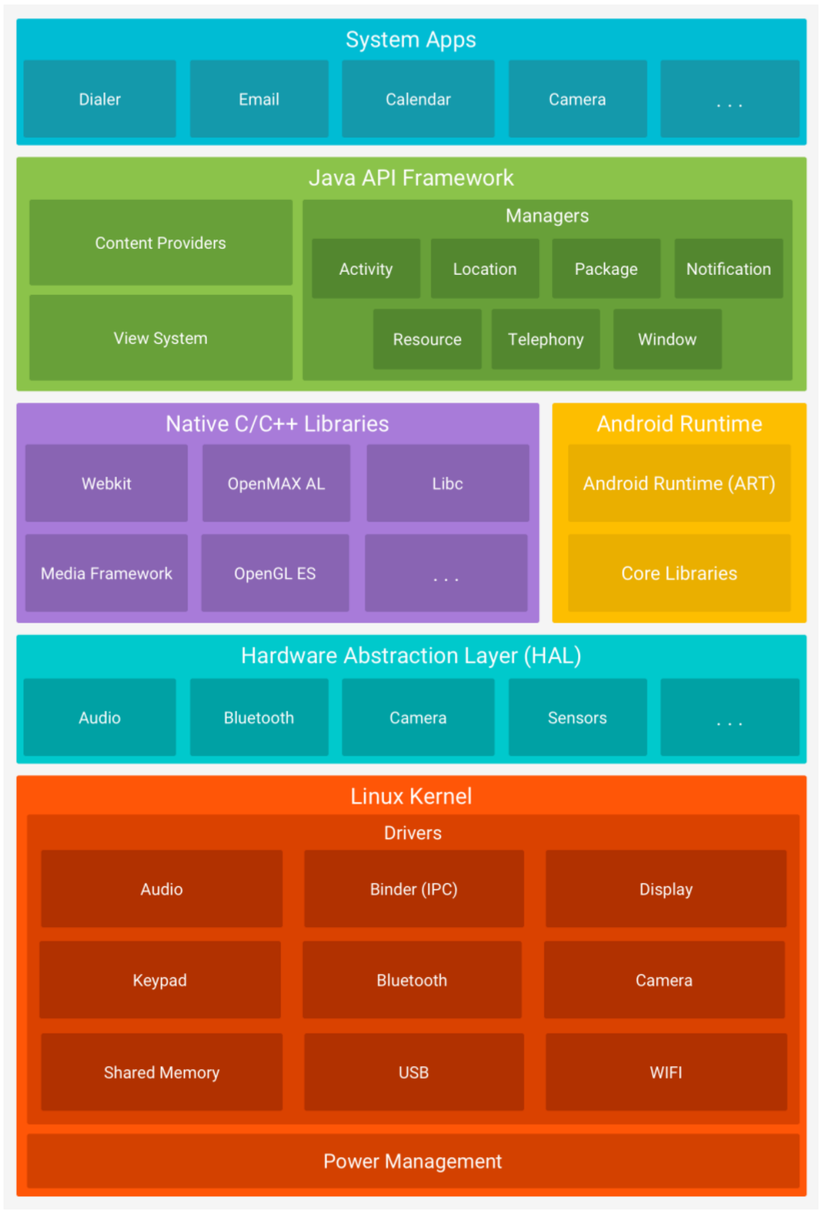


Figure 6: Android software stack

We will start with the layers from bottom to the top. The first layer Android is built is the Linux kernel. It is the layer that doesn't interact users and developers, but it is the heart of the system. It provides security, memory management programs, support for shared libraries, network stack…

The second layer is the Hardware Abstraction Layer (HAL). It defines a standard interface for hardware vendors to implement, and it implements functionality without affecting or modifying the higher level system.

The third layer is the Native C+/C++ Libraries and the Android Runtime. The Native Libraries carries a set of instructions for handling different types of data. An example will be the Media Framework library, that controls the playback and recording of different audio and video formats. At the same level there is the Android Runtime, that recently was changed to ART (Android used Dalvik before). ART is the one that translates the application bytecode into native instructions that are later executed by the device.

The next layer is the Java API Framework. It is the source code of Android that can be used by the developers. It gives us the building blocks we need to create the Android apps characterized for being a modular system. It includes the following:

- A View System for building a UI
- A Resource Manager, that provides access to non-code resources like strings, layouts, graphics…
- A Notification Manager for displaying custom alerts
- An Activity Manager that manages the lifecycle of apps
- Content Providers that enables apps to get data from other apps

And finally, we have the system apps. It is the layer where the user interacts most because it provides basic functions such as making phone calls, accessing the webbrowser etc.

1.4. Android SDK

1.4.1. Definition

Android SDK (software development kit) is a set of development tools released by Google that allows us to build applications in Android. It includes a lot of tools like debugging, the libraries needed, sample source code, the possibility of using an emulated device, etc.

1.4.2. Choosing the correct SDK

Choosing an Android SDK is an important step when we start developing in Android, because it can affect in the success of our app. All apps has three types of SDK versions:

- compileSdkVersion: It's in which version of Android SDK your app will be compiled. We usually use the latest version available.
- minSdkVersion: It's the lower bound for your app. It's the one that determinates if an user can install the app in his phone or not, so it's one of the most important parameters. There are some support libraries (compatibility between different versions) that require a minSdkVersion, if not, it can't be used.
- targetSdkVersion: It is the most optimal Android version for the app. It gives an idea to Android of how it should handle your app in terms of the performance.

Checking the API version distribution will help us to decide what version are we going to use.

ANDROID VERSION (API LEVEL)	**CUMULATIVE DISTRIBUTION**
4.0 Ice Cream Sandwich (15)	
4.1 Jelly Bean (16)	99.2%
4.2 Jelly Bean (17)	96.0%
4.3 Jelly Bean (18)	91.4%
4.4 KitKat (19)	90.1%
5.0 Lollipop (21)	71.3%
5.1 Lollipop (22)	62.6%
6.0 Marshmallow (23)	39.3%
7.0 Nougat (24)	8.1%
7.1 Nougat (25)	1.5%

Table 1: API version distribution (in May 2018)

The table above has 3 columns: the first one is the Android version, the second one is the API Level (each version is identified by a number) and the third value is the cumulative distribution, that tells us how many people has that API version. The idea is to get a balance between going down to support as many devices as possible and not going low for losing features that the app needs, because there are more features when you increase the API level.

The version chosen for the app is 21 (Android Lollipop) because we need basic functionality, such as GPS or file access, and it's the minimum version required for Material Design.

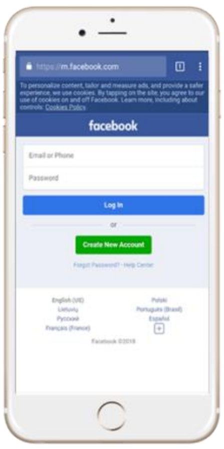
1.5. Choosing the architecture

When developing an app, we have different types of apps we can develop depending on the needs of the company. Normally, we have three types of apps: web apps, hybrid apps and native apps.

1.5.1. Web apps

A web app is a software application that tries to have the same appearance as a native app. It needs a browser to be run and they are programmed using a web language, such as HTML, and usually combined with Javascript and CSS. They work as the same way as a webpage, but it has a responsive design, so it adapts to the screen of the device, and the movement between screens is done by visiting different URL paths. As advantages, web apps require a minimum memory, so it may work in the majority of the devices. Also, the app will work in every SO that has a browser installed and, in relation with that, we have to maintain only one code, so it makes the development relatively cheap.

One of the main disadvantages of using a web app is that we need internet connection, resulting in a bad user experience if there is poor connection. As a developer point of view, we don't have too many API, so it's hard to build something with a little bit of complexity.



To personalize content, tailor and measure ads, and provide a safer experience, we use cookies. By tapping on the site, you agree to our use of cookies on and off Facebook. Learn more, including about controls: Cookies Policy

Figure 7: Example of a web app

1.5.2. Hybrid apps

Hybrids apps are a mix between web apps and native apps. They are built using a combination of technologies such as HTML, CSS and Javascript, but they are installed in a phone like a native app. The main difference with a native app is that hybrid apps are hosted in a native application that uses a WebView, that is a "browser" window that can be used in fullscreen mode. This allows us to access device capabilities like GPS, gyroscope, camera, etc.

Some cons are that the performance is not as good as a native app and even sharing the same code between platforms the UI (User Interface) may be different.

A lot of companies decided to use hybrid apps: Uber, Instagram, Evernote…

1.5.3. Native apps

Native apps, as its name says, are apps that they are "native" for a specific mobile operating system. The most important advantage of a native app is the high performance, because the code is optimized for a particular OS. Also, we ensure a good UX (user experience). Moreover, we have a full access of the API so we have more functions to use in comparison with hybrid apps.

Some cons to native apps are that the development cost is high, because if we want to build an app that extends the main targets (Android & iOS) we have to make two different apps.

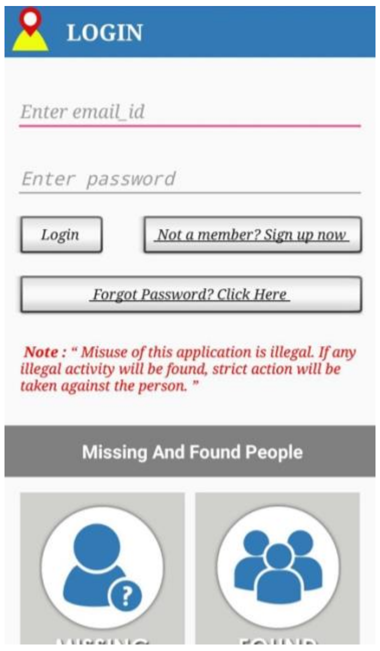
2. Implementation

In this section is explained all the process of making the app, and also all the problems that I have to solve during these months.

2.1. Checking similar apps

A good way to start developing the app is to check similar apps and see its weaknesses, and see how our app can improve it.

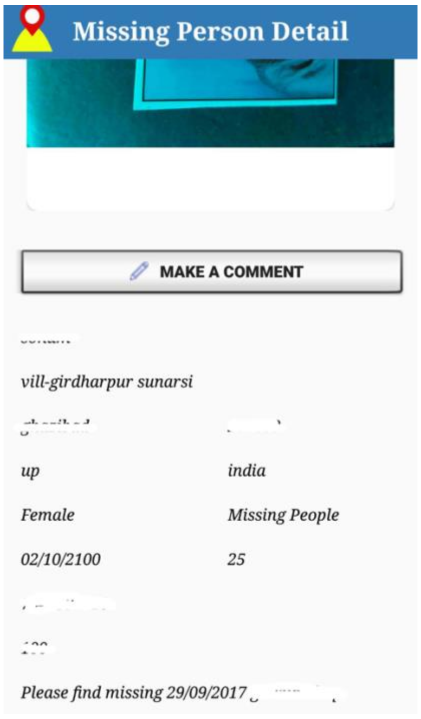
The following app is Missing People Find.



Note : " Misuse of this application is illegal. If any illegal activity will be found, strict action will be taken against the person. "

Figure 8: Screenshot of Missing People Find

We can see that it's a native app. The UI is not the most appropriate having a good user experience. In terms of the design, the font doesn't fit with the type of the app and obviously is not following the rules of Material Design. We also see that the we need to register to enter information of a user.



[redacted]

vill-girdharpur sunarsi

[redacted]

up	india
Female	Missing People
02/10/2100	25

[redacted]

Please find missing 29/09/2017 [redacted]

Figure 9: Screenshot details person Missing People Find

If we check in the details of a specific person, there is a disorganization of the information, so the user can be confused. In general terms, there is a need of improving the UI, it’s an important point because people will not use an app that is not well-structured.

Our app will be different in the concept. As this one tries to find disappeared people, our app will display the people disappeared but in a map. We will focus on the concept of finding people that is near the user, because we have more possibilities of finding it instead of having a list with all the people around the world.

2.2. General structure of the APP

The general structure of the app will be the following:

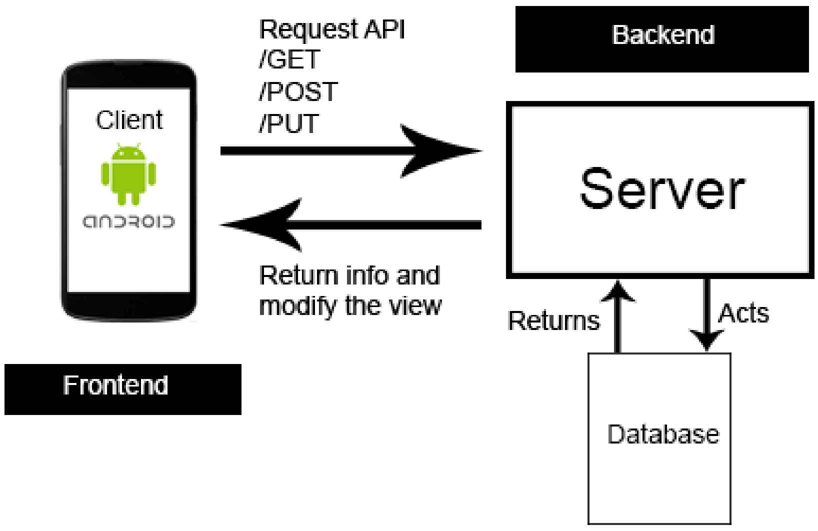


Figure 10: Schema of the app

In the schema above, we can distinguish two parts: the frontend and the backend. The idea is that the user interacts with the frontend. The user will move in different screens, and when an action occurs, we will make a request to a server and, if's necessary, it will modify the database and the server knows the modification that occurred. Then, the server returns the data in raw text (if it's demanded) and the view of the user is modified accordingly with the new information.

2.3. Backend (server side)

The backend is the part of the project that is not visible. It is the logical of the app and usually consists in three parts: the server, the application and a database. As an example, if we buy tickets for a concert, the backend will be how we store the information of the ticket so a specific user can access it after. The backend is also the logic in the Android app: For example, when we click x button, disable the text field and show an error. The three types of backend are explained here.

2.3.1. Server side

The server side is the one that the user will interact for getting or adding the information. In a production app, the server will be hosted anywhere in the internet. In our case, the tests will be done in local.

In the server side, we will use the Restful API architecture. The idea of a Restful API is to make HTTP requests and the server answers in a JSON format with the data specified. Also, we will use the CRUD principles. CRUD means "Create, read/retrieve, update, delete" and it's referred to the database, so we can create new entries, read the information, update information and delete entries. CRUD principles are mapped to Rest commands to obtain a full Restful architecture.

In this part we need to choose the programming language for the server. In my case, I decided to use NodeJS.

Figure 11: NodeJS logo

NodeJS is a open-source, Javascript runtine built on Chrome's V8 Javascript Engine. Javascript is usually used to program in the client side but NodeJS allows us to do it in server-side. It's main use is to build scalable network applications, so one of the characteristics of NodeJS is to handle concurrently a lot of requests. It's a very recent framework, but because of its lightweight and efficiency, a lot of companies are changing the code of the server side and migrating to NodeJS. Also, it's very used for full-stack[1] web developers, because they don't have the necessity to learn different languages for making a project (they can use Javascript for client side + server side).

[1] Developer that programs the backend and the frontend of a project

Anyway, NodeJS doesn’t have in-built libraries to make a web service. So we have to download any external packages with npm[2]. In this case, we will use Express.

Express is a minimal and flexible NodeJS framework that provides a lot of features for web and mobile applications, such as how an app responds to a client request to a particular endpoint, with a URI and a specific HTTP method (routing),the possibility of serving static files or writing our own middlewares.

Also we need the following packages:

- body-parser: It’s a middleware that allows us to parse the information of the user in as a JSON.
- connect-multiparty: A middleware for allowing the user to upload images in a HTTP request.
- moment: A package for dealing with datetimes.
- mongoose: A package that facilitates the use of interacting with the database.
- jsonwebtoken: Library used for using the jsonwebtoken technology.
- bcrypt: Library used for hashing passwords in the database.

For making a scalable server and with the possibility of adding more features in the future, it is needed to structure the code. This is how I organized the data.

The “package.json” file contains all the packages needed for running the server. It is useful if we want to move the server, because we can move only the source code files and with a command we can install all the packages, reducing the folder size.

I have two files in the root called “app.js” and “index.js”. In app.js, we initialise all that is needed. We load all the routes and the middlewares for getting a JSON response and we export the object app. In the index.js, we get

[2] Package manager for Javascript programming language. It’s the default package manager for NodeJS

the app object from app.js and we connect to the database. If it’s successful, we run the server.

The “uploads” folder contains all the images that the users uploaded from the app.

The “models” folder contains all the models of the database. In our case, we have the person model.

The “controllers” folder contains all the functions that are related with a route. This is where we interact with the database, getting, adding or updating data.

The “routes” folder contains all the routes of our server, divided by the types. In our case, we have all the routes of a person, that defines which function will be executed in every route.

An example route will be /showall, and it is going to return all the info of the database. The controller will be the function related to showall:

```
function showAll(req,res) {
    Person.find({},(err,people)=>{
        console.log(people);
        if(err) return res.status(500).send({message:"Unknown error!"});
        people.forEach((person)=>{
            //delete person._id;
            person.__v=undefined;
        });
        return res.status(200).send(people);
    })
}
```

As we can see, we make a query to the Person model, and if there’s not an error, we return successfully (200) and we send as a JSON all the people from the database.

The "utils" folder contains code not related with the MVC. In this case, we have que jwt file that manages all the encryption of the payloads for using web tokens.

The constants file contains all the constants related to the project, such as the port of the server, the secret key for encrypting the data, the path of the images, etc.

2.3.2. Database side

Because of the requirements of the app, we need any storage system that allows us to deal with all the information we will work with. We need only to storage text, as in the case of the photo we will save the name of the file.

When choosing a database, we have to choose first the type. Generally, we have two possibilities:

- Relational databases: Relational databases represent and store the data in rows and columns. Its principal advantage is to optimize data and don't use duplicated values. For making a query, we use Structure Querying Language (SQL). We can link information from different tables using foreign keys, that identify a unique row. MySQL, PostgreSQL and SQLite3 are the most famous ones.
- Non-relational databases: Non-relational databases represent data in collections of JSON documents. The main advantages are that the latency is low with small pieces of data, it is highly scalable and its performance is good. Also, it is very easy to work with this type of databases because it is the same as dealing with JSON. MongoDB is the most famous non-relational database.

The database that is using the project is MongoDB, because we need to deal with small pieces of data and also because we only need simple relationship between collections, and the library Mongoose of NodeJS allows us to do it such as a relational database.

This project has 3 collections: the "Person" model, the "Users" model and the "Verification" model.

The schema of the database is the following:

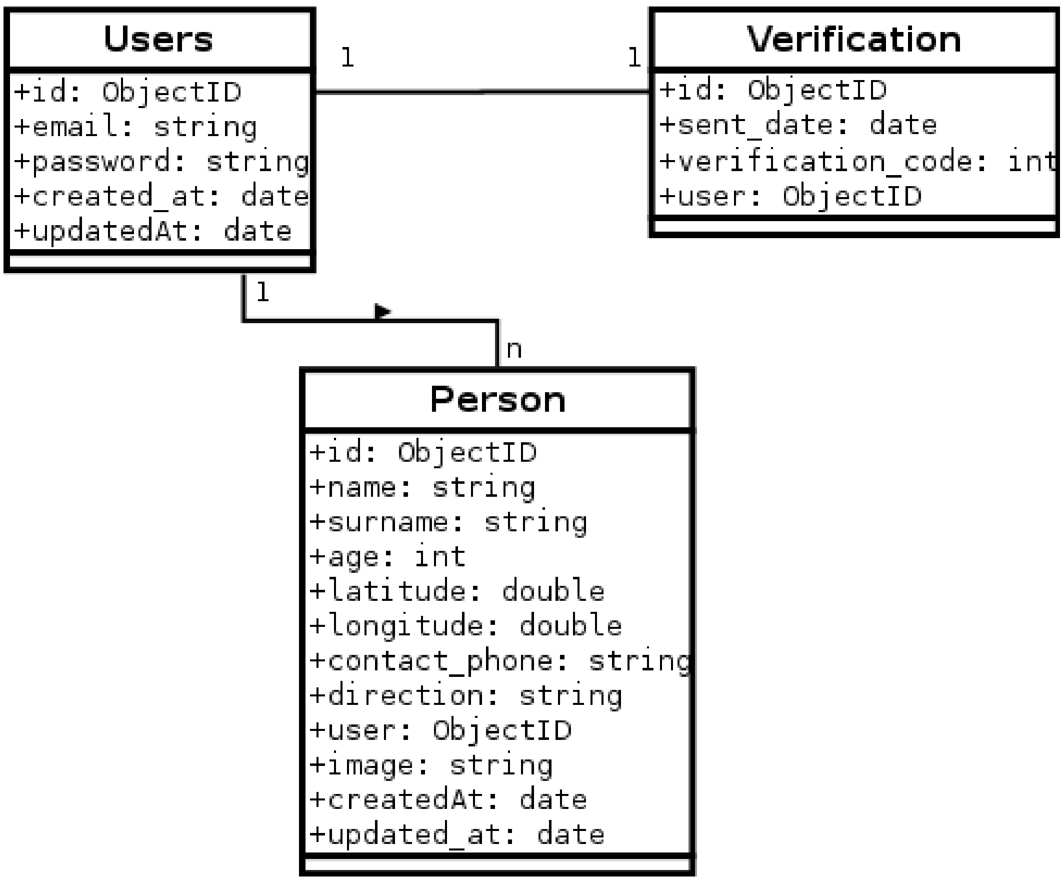


Figure 12: Schema of the database

The users collection is the one that stores the information of a user that is using the app and it is registered. In the register form we only ask for an email and a password, but in the future it can be implemented for asking more fields and make, for example, a "My profile" section. It is related with the verification schema, that stores the information of a user when it is not verified. The relationship is 1-1, that means that a user can only have one verification entry and one verification can be related to one user.

The Person schema contains all the information related to a disappeared person. As saving an image in the database is inefficient, we store the name of the file that is in the database. This can cause an integrity problem, because a person can point to a file that doesn't exist. For this reason, we have to take care of maintaining a relation and remove or update the information accordingly.

As we can't have any relationship between models in a non-relational database, we have to use an alternative. The NodeJS library "Mongoose" allows us to create different models like in a relational database, and create a link between collections. For example, creating a 1-n relationship is as simple as creating a field that references the other collection.

```
PersonSchema = new Schema({
    name: {type: String,required:true},
    surname: String,
    age: Number,
    latitude: {type:Number, required:true},
    longitude: {type:Number, required:true},
    image:{type:String,required:true},
    contact_phone:{type:String,required:true},
    direction:{type:String,required:true},
    user: {type: Schema.Types.ObjectId,ref:'User',required:true}
},{timestamps: {createdAt:'created_at'}});
```

And for creating a 1-1 relationship, we just need to the same as a 1-n but, in this case, we just need to pass a parameter called unique. The database will crash when we try to add two entries that have the same value.

2.3.3. Application side

Not all the logic is involved in the server side. The application also needs a logic for working properly. For moving into different screens, we will use "activities" and "fragments". The difference between them is that an activity is more slow and a fragment stays in memory. The rule is that we have to use fragments when we can, a fragment will be used when we change completely a screen.

The applications is divided in the following form:

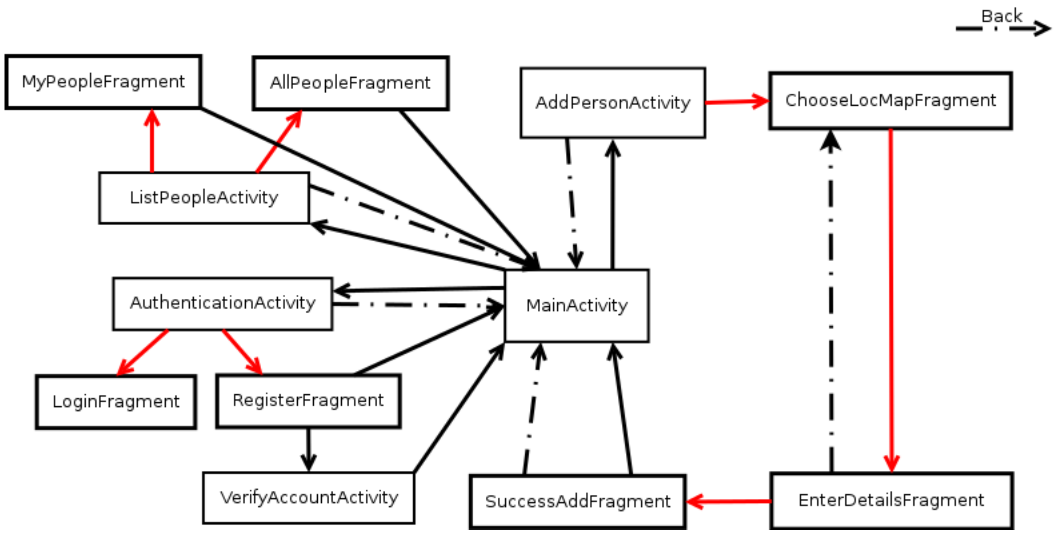


Figure 13: Diagram of the application

The black arrows represent the movement between activities. The red arrows represent the change of a fragment, so we stay at the same activity. In the schema, we see that "MainActivity" is the core of the application. It shows a map, controls all the gestures of the user (moving through the map, stopping the camera) for updating the information accordingly and making the necessary requests. Also, it contains all the logic for the navigation drawer. Depending if the user is logged or not, we will inflate a navigation drawer or another. In the case that the user is not logged, we will inflate a menu that contains the login and the people section for checking all the people that is in the app. The MainActivity also controls the topic of requesting permission for getting the position of the user.

In the "AddPersonActivity" we have a "FrameLayout", that is a container where you can load a fragment. In our case, we load first the "ChooseLocMapFragment" which allows the user to select an ubication. If the user selects a correct place, we move to the "EnterDetailsFragment" that asks for the basic information, that is uploading a photo, a name, the age and the contact phone. If we press back, we move again to the selection of the place. If the user enters all the information required, we move to the SuccessAddFragment that gives a information to the user that the user had been added. We have a button that finishes the "AddPersonActivity". The same happens when the user clicks the back button.

The “AuthenticationActivity” contains a bottom navigation view for moving into the different sections. Also it contains a frame layout that changes fragments between the “LoginFragment” that contains all the logic for logging in a user, such as dealing with different type of HTTP error codes and the “RegisterFragment” for allowing the user to register. When the user is logged in, that activity is not displayed in the navigation drawer until the user clicks the logout option.

The “VerifyAccountActivity” is used for managing the verification of an account.

The “ListPeopleActivity” contains the list of the people who is in the database. It contains a RecyclerView and, if the user is logged in, he can see the people who added himself and, if we click, it appears a popup with all the information, including a map with his/her location and with the possibility of removing the user in the case it is found. In this activity, we load two fragments at the same time, for improving the user experience and reduce the time of loading. It also includes pagination for not loading all the data.

2.3.4. Dealing with possible large amount of data

One problem that we have to solve is how we download all the data from the server efficiently, so we have to find a method that has a good performance without wasting resources or very consuming.

The first idea is to download all the data from the database and put all the markers. It is not a very suitable method because it will work very well if there is not a lot of data but the performance will decrease if we have tons of data.

A possible solution can be getting the data only from the area that the user is looking for with a certain zoom level, because if we do not restrict with zoom the user can zoom out and get the full map and we have the same problem as downloading all the data. By default, the zoom level is set to 15, that is a zoom level that we expect to see street names. Any zoom level that is below we will not load the data.

For knowing the visible area we need to get the latitude and longitude of the extremes. Google Maps has a specific function to get the latitude and longitude bounds of the northeast and the southwest zone. We have to do a little bit of maths to get the users that are in that area.

The world map is divided by the following coordinates:

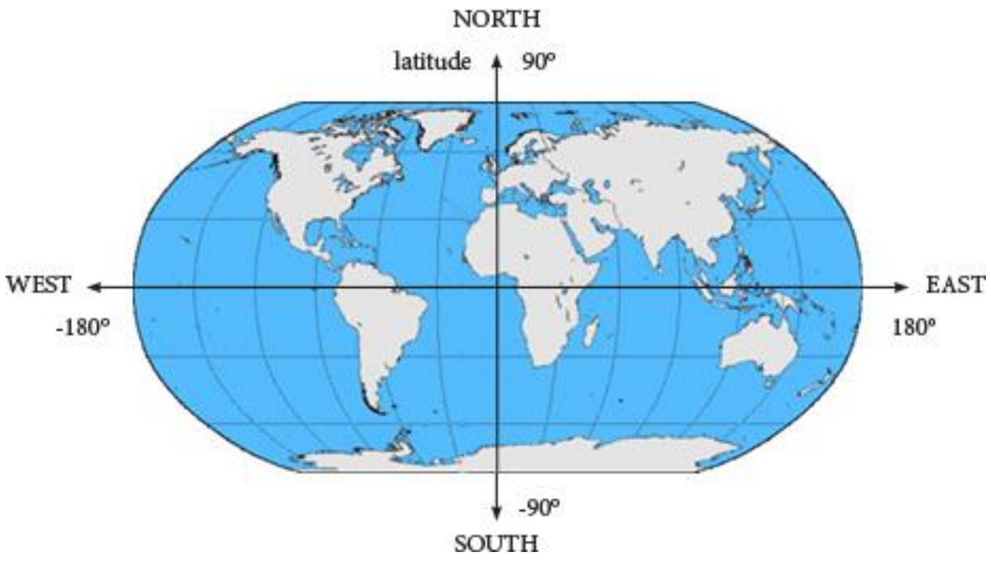


Figure 14: Geographic coordinate system

The longitude goes from -180 (west) to 180 (east) and the latitude goes from 90 (north) to -90 (south). If we have the northeast and southwest, we can do the following (considering that the map has no rotation, because the rotation gesture was disabled by code):

- First, check if latitude is smaller or equal to the northwest latitude and also check if latitude is larger or equal to the southwest latitude.
- If the condition above is true, check the longitude. If it's larger or equal to southwest longitude and smaller or equal to the northeast longitude, it's in range.

So the pseudocode will be like the following:

```
users = []
For user in getAllUsers():
if (user.latitude <= northeast_latitude && user.latitude >= southwest_
latitude)
    if (user.longitude >= southwest_longitude && user.longitude <= nor
theast_longitude) users.append(user)
return users;
```

Also, with this method, we have cons. The most important disadvantage is that we make a lot of requests. When the user is scrolling the map and the camera is stopped, we make a request to get the data, so another disadvantage related with the one above is that if the user moves to the zone of the map that is very next to we get repeated data in every request.

In the case of displaying all the people in a list, we have the same issue. We cannot load all the people in one request because, in the case of having a lot of data in the database, both mobile phone and server can reduce the performance. This is why we are going to use the pagination. We will achieve this using the predefined functions that Mongoose has, particularly the skip and the limit. Changing the page will be done by the url type. In this case, making a request to /allpeople/1 will output the contents of the page 1.

In the server case, we need to skip and limit:

$$Skip = (ITEMS\_PER\_PAGE * page) - ITEMS\_PER\_PAGE$$

$$Limit = ITEMS\_PER\_PAGE$$

For example, if we have 20 entries and we want to see the 1st page with a limitation of 5 entries, skip value is 0 (first entry) and the limit is 5, showing the values 1,2,3,4,5 from the database.

In the frontend, we need to implement some method to allow the user move between different pages. I've used the "load more" method. Initially, we load the first page and, if it's necessary, a load more button in the footer of the list is added. If the user clicks, it loads the following page, adding the new elements at the bottom of the list. We know the number of pages that we can load because the server gives us the total number of pages, so in the first request we save it in a local variable. When all pages are loaded, we hide the load more button.

```
int TOTAL_PAGES = 1;
int actual_page = 1;
LoadMoreButton.onClick(
    if (actual_page <= TOTAL_PAGES) {
        hide(LoadMoreButton);
        make(request, actual_page).OnNewRequest(actual_page++;
            if (firstRequest) TOTAL_PAGES = getFromRequest("total_pages"); updateList
WithContent();
            if (ACTUAL_PAGE <= TOTAL_PAGES) show(LoadMoreButton);)
    });
```

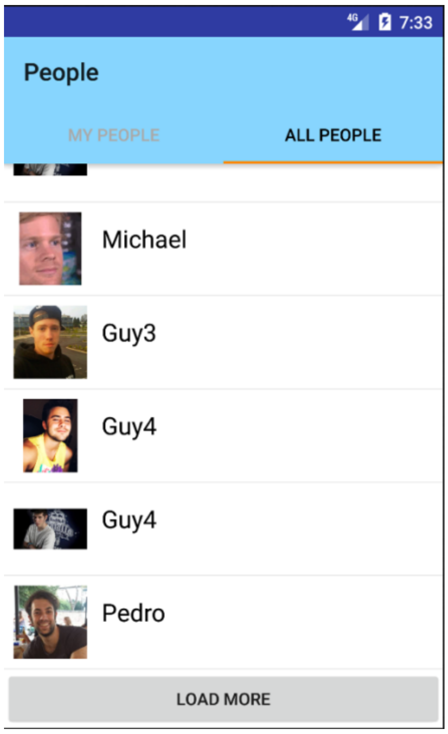


Figure 15: Design of pagination in the app

2.4. Frontend (general design)

In the section before we have talked of backend, that is the logic that the user can’t see. The frontend is the opposite. It is the part that the user can interact with. We make this distinction because there are two different parts and we usually divide the work for different profiles. The backend is usually made by a programmer and the frontend is usually made by a designer, but we also have a full stack developer that does both things. The design of the app was the hardest part of the project, because the career is more focused in programming instead of designing apps.

2.4.1. Material Design

Material Design is a design language that was released by Google in 2014. It is a set of rules that Google gives us for making the general design of an app for being intuitive and easy to work with all the types of users. Material Design can be used in Android with an API Level of 21 (Android Lollipop 5.0). Nowadays, Google is extending Material Design not only for mobile apps but also for the creation of webpages, providing a similar experience in all types of platforms. Creating an app using Material Design rules is very important because the majority of the people is used to that design, as all the Google applications such as Gmail, Calendar, Play Store… use that design.

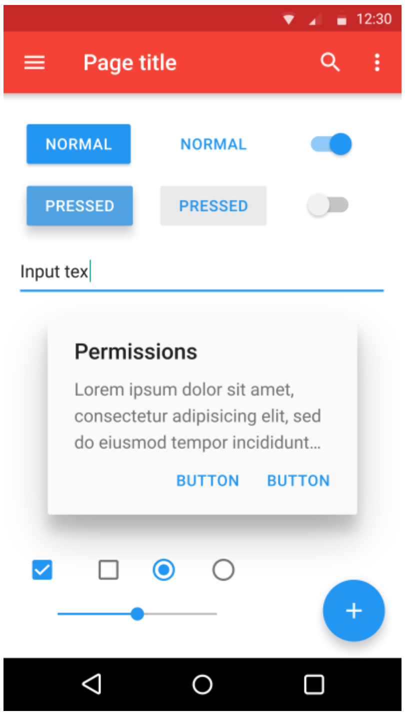

12:30
Page title
NORMAL
NORMAL
PRESSED
PRESSED
Input tex
Permissions
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt...
BUTTON
BUTTON

Figure 16: Example of a Material Design app

2.4.2. Design of the app

The design of the app is the following:

The "activity_main" layout is related with the "MainActivity". It contains the SlidingPanelLayout, used for displaying the user information in a sliding panel. We also have two floating buttons, one for getting the GPS position and the other for updating the information manually. We have a lateral Navigation View and a toolbar overlaid for opening it. In total, we have 5 layouts for making the main design: activity_main, app_bar_main, content_main, nav_header_main and panel_user.

The design of the choosing a location is very simple. We have a LinearLayout that groups all the elements, we have the principal fragment that loads a map, below we have a CardView that includes a TextView that is updated when a user selects a location and a next step button. Also we can click to the TextView and it appears a screen for finding a place. If the user selects one place, the map is automatically moved to that position.

In the “Entering more details” screen, we have the inputs of the user. A button for selecting the image, and all the data needed: name, surname, age and a contact phone. If we click next and not all the data is entered, the EditText turns red giving an error message.

The “Success adding a people screen” is simply a TextView informing the user that is added and a button for going to the main activity.

2.4.3. Dealing with phone numbers around the world

One problem that we can have is the phone number prefixes when. How do we know if some phone number is from certain country. We need to find a way to know from which country is the telephone number when we are asking for the contact phone. The solution is adding a widget that allows the user to enter the code of the country. With this, we will not have any problem. Google has not have any official library for dealing with this, so we need to use an external library. The library “Country Code Picker (CPP)” by shivam95 was used. It looks like the following:

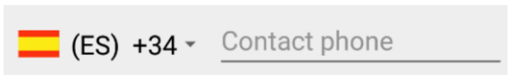


Figure 17: Country code picker widget

And when we click on the arrow:

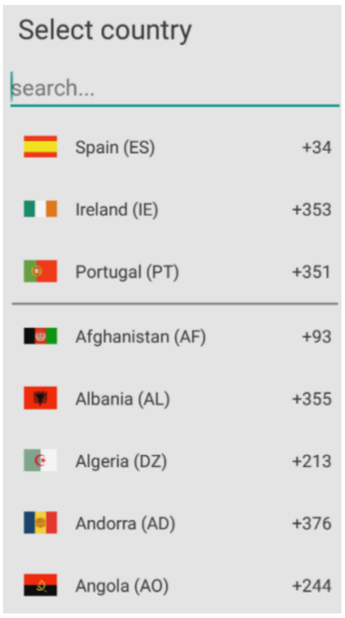


Figure 18: Extendable menu of country code picker

2.4.4. Choosing how the user inputs a location

The selection of how the user will choose a location is not as easy as it seems. We have to find an intuitive method for allowing the user to select a place. I have found two possible methods:

- Touch-based method: The touch-based method is based on the following: we have a map and the user can click in some place of the map and a marker will be added. If the user clicks another place, the old marker will be removed and changed for the new position. If we search for a place, we will move the camera and add a marker.
- Scroll-based method: The scroll-based method is the same concept as the touch-based method but with the difference that the user has a marker in the centre. The marker always remains in the centre and it's the position that the user chose. If the user searches a place, the camera is centred at the point so the marker points at the position.

The final implementation uses the scroll-based method, because it is more intuitive for the user and some important apps like *myTaxi* uses it.

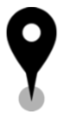


Figure 19: Icon of location

The icon above was designed with Photoshop.

The main problem now is how to put the marker in the centre. It can seem easy, but we don't want to put the marker in the centre, we want to put the shadow area. We cannot achieve this using only the XML design, so we have to do it programmatically.

The idea to solve this is to get the height of the Google Maps fragment. When we have that height, we can apply to the marker a margin top, doing this we achieve to have it in the centre, but it's the same case as before, so in that margin we have to minus the height of the icon itself.

$$Margin\ top\ marker = \frac{Height\ fragment}{2} - Height\ icon - constant$$

Constant is a value for fitting the centre of the shadow just in the point (It's approximately 1/13 part of the icon, the value has been achieved by trial and error).

To see if it works, I put a marker just in the centre and see if it matches with the marker we've added programmatically. This is what I get:

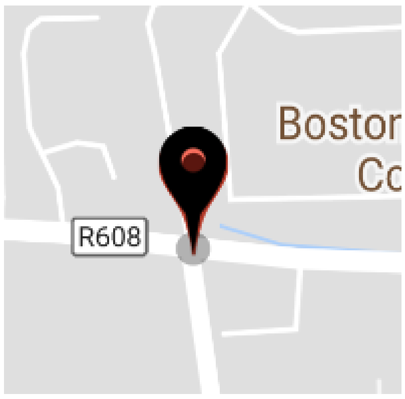


Figure 20: Test of the centre marker

As we can see, we could put the marker in the centre.

2.4.5. Using clusters for avoiding lag

The second aspect that I want to talk is about clusters. Imagine we have a map zone loaded and a lot of data is inside the map. We will have a lot of lag when scrolling. We need to find a solution for avoiding this. Google has a tool called "Marker Clustering". The idea of marker clustering is to group markers that are next to, and it appears a numbers that shows the count numbers in the area. If we zoom in the map in the cluster zone, we will see the markers, and if we zoom out, the cluster will group it again. With this method, we solve the overload of markers that cause a poor performance. If we have this:

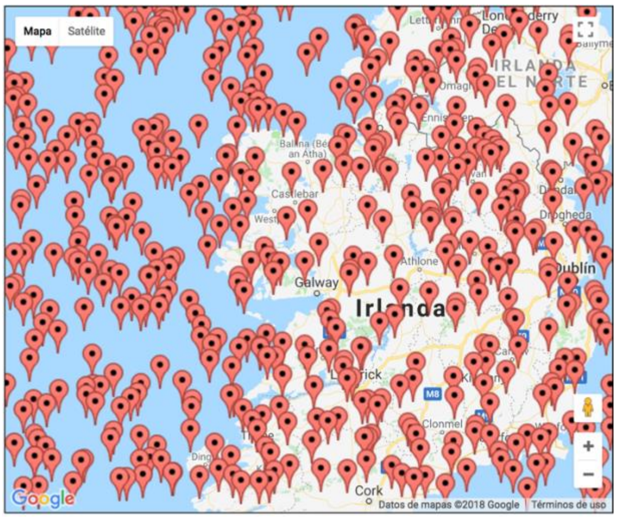


Figure 21: Map with an overload of markers

With a “Marker Cluster” is converted as the following:

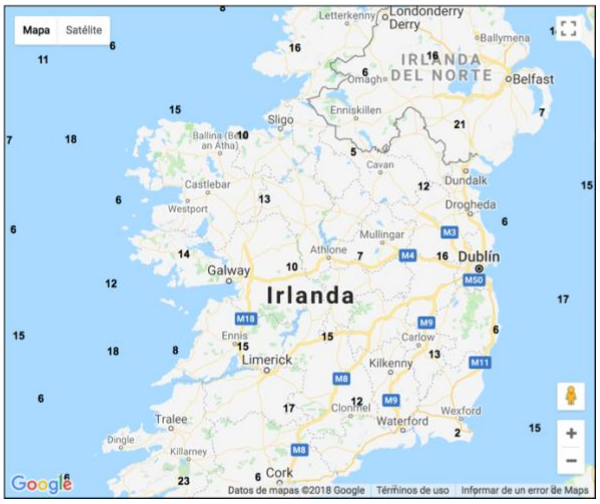


Figure 22: Map using a "Marker Cluster"

2.5. Adding a user role

Implementing a user role is important for managing the content. A user role, as a concept, defines permissions for users for performing different group of tasks. In the case of the app, we have only 1 role, that is the user that controls the app. Anyway, more roles can be added at the time we are improving the application, specially the admin role that allows full access to all of the functions, including editing or removing a person and also ban users. For controlling the actions that a specific role can do, we can use an integer, which each action has assigned a number that is power of 2 (1,2,4,8..). Then we make an and mask comparing if the value is equal. If it is, the user has access to that action.

For adding a user role, we need to use a type of authentication for HTTP requests. Nowadays, we have two types: cookies and JSON Web Token.

2.5.1. Cookies

In this section we will talk about cookies, more specifically with session cookies. Cookies are small piece of data that are stored in the user's computer. They are used for displaying a particular website to a specific user, so we are storing state information on the browser. Each cookie is a dictionary that contains pairs of key-data values, having the possibility of adding a expiration date.

Session cookies are used for the authentication of users. It works as the following way: the user makes a request to the server with the credentials (usually the username or email and password). The server receives the request and, in the response, we add a Set-Cookie header with a session id. The browser saves that session id and, in every request, we add that value. In the server side, the session id is stored along with all the data that is needed, so deserialize session and the server now has access to a specific user.

The biggest advantage of using session cookies is that they can be accessed on the browser or by using JavaScript. In our case, it is not needed, because we will use the Android app for making the requests. Also, they are restful, so requests are not related with the server state.

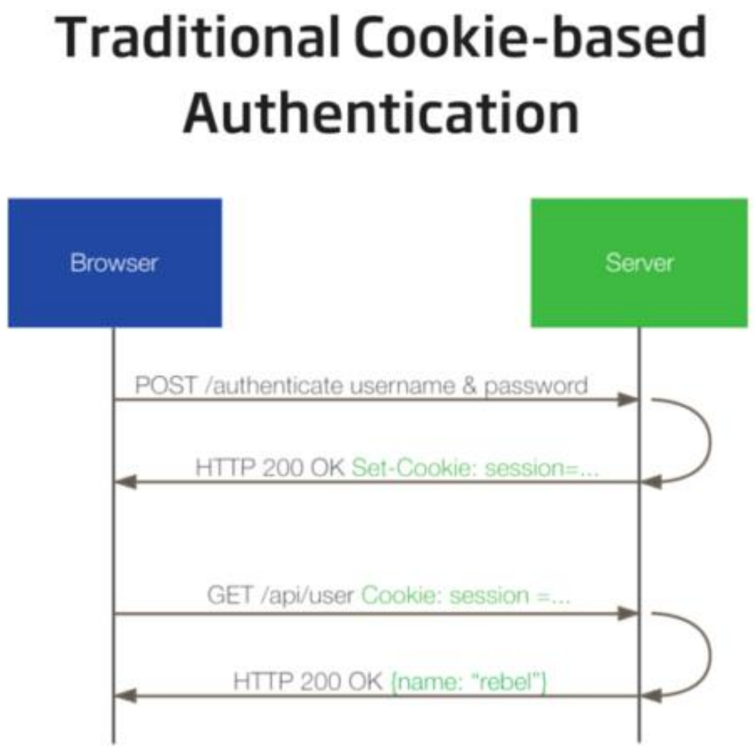

Traditional Cookie-based Authentication
Browser
Server
POST /authenticate username & password
HTTP 200 OK Set-Cookie: session=...
GET /api/user Cookie: session =...
HTTP 200 OK {name: "rebel"}

Figure 23: Traditional cookie session authentication

As the biggest drawbacks, we have that:

- Each cookie is limited to 4kb, with a total of 80kb.
- Difficulty if we want to use CORS. Cross Origin Resource Sharing is used for getting a resource of another webpage using a different origin.
- We need to implement protection for CSRF (Cross-Site Requests Forgery) attacks.
- The server has to save the session id.

2.5.2. JSON Web Token

JSON Web Token can be used as an alternative to cookies. JWT is a standard created for making access tokens. The token is encoded using an algorithm such as *HS256* and it ensures the connection between two parties securely. The final token is a string that contains three parts, separated using a dot. The first part is the header, that contains about how the JWT signature should be computed (algorithm and type). The second type is the payload, and it contains the data we want to store. It usually includes the expiration time (exp) or the issuer (iss). The payload is not encrypted so we can't save sensitive data because it can be easily decrypted. The third part is the signature, used for checking that the message hasn't been modified. It is computed in the following way:

$$datatoEncode = base64(header) + . + base64(payload)$$
$$Signature = HMACSHA256(datatoEncode, SECRETKEY)$$

We can use that technology for authentication. Let's assume that the user makes a request sending the credentials. If the data is correct, the server makes a token with the information needed, signing using a secret key. The user now saves that token and each time it wants to do a request, it will include that string in the "Authorization" header. The server will verify if it's not corrupted and will act accordingly.

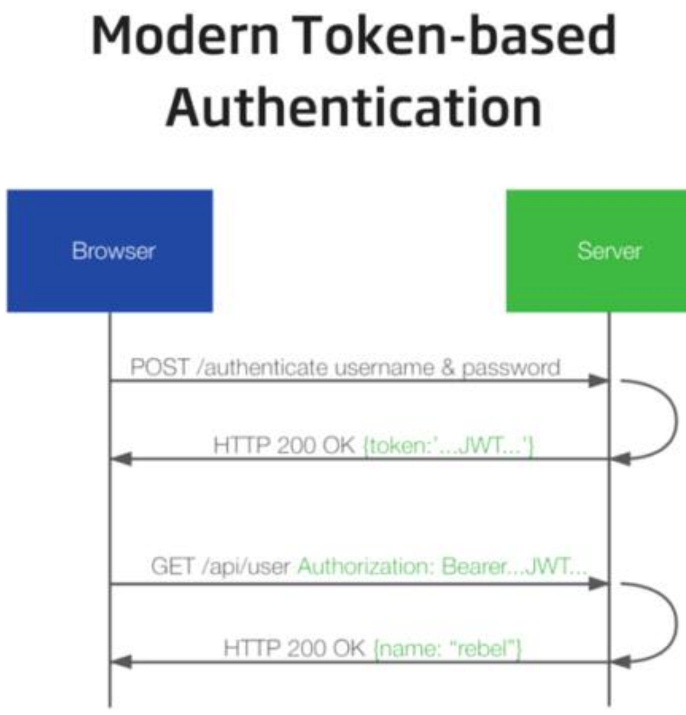

Modern Token-based Authentication
Browser
Server
POST /authenticate username & password
HTTP 200 OK {token:'...JWT...'}
GET /api/user Authorization: Bearer...JWT...
HTTP 200 OK {name: "rebel"}

Figure 24: Modern token based authentication

One of the biggest advantages is that the server doesn't need to save any extra data, we can get the data only decrypting, so it's statelessness and, for this reason, it is very scalable if we want to deal with multiple backends. Also, it's very useful for building a mobile-ready backend, because using tokens is more easy to deal in comparison with cookies in any mobile operating system.

As a disadvantage, as being a statelessness technology, we can't disable a token in the server side. The only method available is to put an expiration date in the payload and, when it's overpassed, force the user to login again. For logging out the user in the client side, we simply remove the token. Furthermore, we have to save the secret key securely, as if a malicious user gets it, he can generate different tokens and modify the behaviour of the app. As a example, he can change the user id in the payload and act a as a different user.

Using JSON Web Tokens is perfect for our application because it's hard to deal with cookies in both client and server side and it's more easy to make requests. We only need to save the token one time and add a new Authorization header in each request. In addition to this, JWT has a set of libraries for NodeJS that allows us to use the basic functions such as signing or verifying.

2.5.3. Implementing the authorization (server side)

In this section we will explain how it is implemented the authorization on the server side. The first issue that we have is that there are some routes that we don't have the necessity to authenticate, so we need a method for distinguishing between protected and non protected routes. The solution to this problem is to use *middlewares*. A *middleware*, in terms of NodeJS, as its name says, is a function that is called before the controller (in the middle). Using that concept, we can make a function to ensure authentication, and apply that middleware only to the routes that is strictly necessary. It is implemented in the following way:

```
if not requestHeader("Authorization") return response(400);
if verifyToken(token, SECRET_KEY) {
    request.append(decoded_token);
    next();
} else {
    if (verifyToken.error == TOKEN_EXPIRED) return response(401);
    return response(403);
}
```

First, we check if we have the authorization header. After, we verify the token with the secret key stored in constants. If there is an error or the token expired, we return a response with status code 400 or 401, that means bad request. If not, we put the decoded token in the request and call the next function.

2.5.4. Adding a user verification

Adding a method of verification a user is very important for avoiding an inappropriate use of the application. We will implement a method for checking if the email entered in the signup form is genuine. When the user is fully registered, we are going to send a verification code in the email account that the user has to enter correctly in the app. When the user enters the code, the user is verified and he can use the app normally. In the server side, we will control if the user is verified by using a boolean field called "isVerified", that is false by default. If the user tries to login and it is not verified, the server will response with a status code of 433. In the client side, it will redirect to the activity of verification. The pseudocode of the login is the following:

```
if (User.findBy(email == email, password == password)) {
    if (User.isVerified) return response(createToken());
    else {
        sendCodeToEmail(User);
        return response(433)
    }
}
```

Not always we send the code when the user logs in. In the verification table, we save the date when a code is sent. If the user demands another code, it will not be sent after they've passed 10 minutes before the last one.

However, the method implemented doesn't take in account disposable email addresses.

2.5.5. Securing the database

Securing the database is a important step if we want to deploy the server in a production server. The first thing to implement is hashing passwords. A hash, more specifically a cryptographic hash, is a special class of hash suitable for cryptography. It consists of mathematical algorithms that converts data in a string of fixed size designed for be on a one way. This means that we can convert a string to a hash, but not in the reverse way. The ideal cryptographic hash function has:

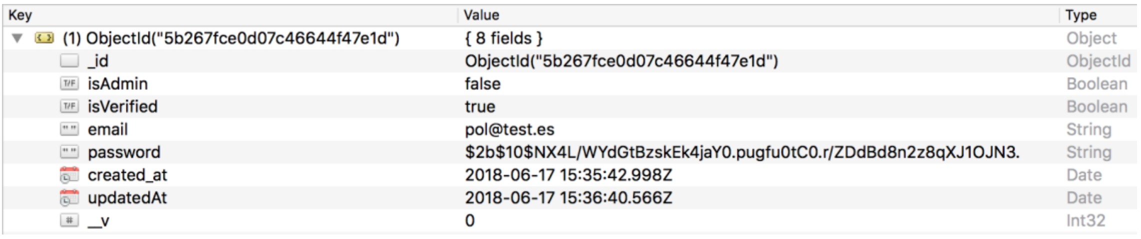
- Deterministic: The same message always has the same hash
- It is quick to compute the hash
- Infeasible: Two different messages can't have the same hash

Implemented cryptographic hashes are continuously checked in looking for collisions. Collisions occur when two strings have the same hash. The most important cryptographic hashes are MD5, SHA-1, SHA-256 and SHA-512.

For hashing passwords we will use the "bcrypt" library. In the user model, we will create a function that it will be executed before saving in the database, allowing us to save the hashed password. When we register the user, we save the password normally, and when the save method is called, we will put the hashed password. The code for doing this is the following:

```
UserSchema.pre('save', function(next) {
    var user = this;
    if (!user.isModified('password')) return next();
    bcrypt.hash(user.password, DEFAULT_SALT_PASSWORD, function(err, hash) {
        if (err) return next(err);
        user.password = hash;
        next();
    })
});
```

If we see the database, we have to check that is working:



Key	Value	Type
(1) ObjectId("5b267fce0d07c46644f47e1d")	{ 8 fields }	Object
_id	ObjectId("5b267fce0d07c46644f47e1d")	ObjectId
isAdmin	false	Boolean
isVerified	true	Boolean
email	pol@test.es	String
password	$2b$10$NX4L/WYdGtBzskEk4jaY0.pugfu0tC0.r/ZDdBd8n2z8qXJ1OJN3.	String
created_at	2018-06-17 15:35:42.998Z	Date
updatedAt	2018-06-17 15:36:40.566Z	Date
__v	0	Int32

Figure 25: Test of hashing password

3. Future implementations

The implementation of this course is the base for all that is coming. The project is very scalable in both server side and user side. Actually, the app is in an earlier release and doesn't have a lot of features, but we can the implement the following things:

- Improve the UI: This is the most important aspect. The app has to get an improvement of the User Interface because now is intuitive but at the same time very simple. For me, it's the most difficult aspect to improve because I'm not a designer and also because on my career I was more focused on the backend.
- Maximum requests per IP per hour: This feature can be implemented on the server side. Using this feature we can avoid that, if the user knows the location to make the request (or even using it inside the app) add, for example, 100 disappeared people every time it makes one request. This can cause a security issue in the app and have a lot of fake data in the database. If we set a maximum requests per hour, the user can do the same but only x times we specify per hour, and we can detect easily who enters fake data. Using this, we can ban them to not make more requests anymore.
- Add permission in user roles: In this project, we've implemented a simple user role, that allows adding or removing the people that they've previously added. Permissions can be also implemented, allowing, for example, an admin to remove all the people, or banning/removing users.
- Add https in the request: As being a development server, https is not implemented. If we want to implement this as a production server hosted on the internet, https is very important because all the login credentials in http goes in plain text and can be a potential issue for users when they are using the app in a public hotspot. When you purchase a hosting, the majority of companies offers you https for improving the security of your server. If not, there are free alternatives such as Let's Encrypt.

4. Conclusion

The project made achieves the purpose of creating the app, that is making a tool for help finding people who disappeared. Due to lack of time it could not be implemented all the functionality that I wished, but the initial objectives were achieved. This project is only the beginning for start making a production app, because as a summary it is needed to implement more security in the server side.

In terms of the app development, I did not have any experience with Android, only the basic aspects such as programming in Java, so I had to search a lot of information of how to do all the things required.

During all the project, there were a lot of changes to achieve the implementation required for improving the existent code, and also for making it compatible with different versions of Android, like the GPS permission that works different in Android 5.0 and Android 6.0.

In conclusion, it was a very rewarding project, because I learnt a lot in terms of programming, specifically with mobile programming, and also how to manage and design (with limited time) a project of this scale.

References

eMarketer (2017) ‘Statistics of people who owns a smartphone’, [online] https://www.emarketer.com

[accessed 3th of May 2018]

Statista (2017) ‘Number of available application in Play Store’, [online] https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[accessed 3th of May 2018]

Statista (2017) ‘Number of available apps in App Store’, [online] https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/

[accessed 3th of May 2018]

Wikipedia (2018) ‘Windows Phone’, [online] https://en.wikipedia.org/wiki/Windows_Phone

[accessed 4th of May 2018]

Wikipedia (2018) ‘iOS’, [online]

https://en.wikipedia.org/wiki/IOS

[accessed 4th of May 2018]

Wikipedia (2018) ‘Android’, [online] https://en.wikipedia.org/wiki/Android_(operating_system)

[accessed 5th of May 2018]

Statcounter (2018) ‘Mobile Operating System Market Share’, [online] http://gs.statcounter.com/os-market-share/mobile/worldwide

[accessed 5th of May 2018]

Thinkmobiles (2018) ‘Types of apps, different categories of mobile applications’, [online]

https://thinkmobiles.com/blog/popular-types-of-apps/

[accessed 5th of May 2018]

Android Developers (2018) ‘Platform Architecture’, [online]
https://developer.android.com/guide/platform/

[accessed 8th of May 2018]

Edureka (2017) ‘Android Architecture Guides’, [online]
https://www.edureka.co/blog/beginners-guide-android-architecture/

[accessed 10th of May 2018]

Android Open Source Project (2018) ‘Hardware Abstraction Layer (HAL)’, [online]

https://source.android.com/devices/architecture/hal

[accessed 10th of May 2018]

Wikipedia (2018) ‘Android Runtine’, [online]

https://en.wikipedia.org/wiki/Android_Runtime

[accessed 13th of May 2018]

Wikipedia (2018) ‘Cross-Site Request Forgery’, [online]

https://en.wikipedia.org/wiki/Cross-site_request_forgery

[accessed 16th of May 2018]

StormPath (2018) ‘Token authentication: The Secret to Scalable User Management’, [online]

https://stormpath.com/blog/token-authentication-scalable-user-mgmt
[accessed 20th of May 2018]

Wikipedia (2018) ‘JSON Web Token’, [online]

https://en.wikipedia.org/wiki/JSON_Web_Token

[accessed 20th of May 2018]

Wikipedia (2018) ‘Cryptographic hash function’, [online]

https://en.wikipedia.org/wiki/Cryptographic_hash_function

[accessed 20th of May 2018]

Appendix A: User Manual

In this appendix we will guide the user how to use the app.

The app was created for helping people in the research of disappeared people. As a general terms, we can see a map of the world and markers that indicates that somebody was disappeared in that zone. At first launch, we have the following screen:

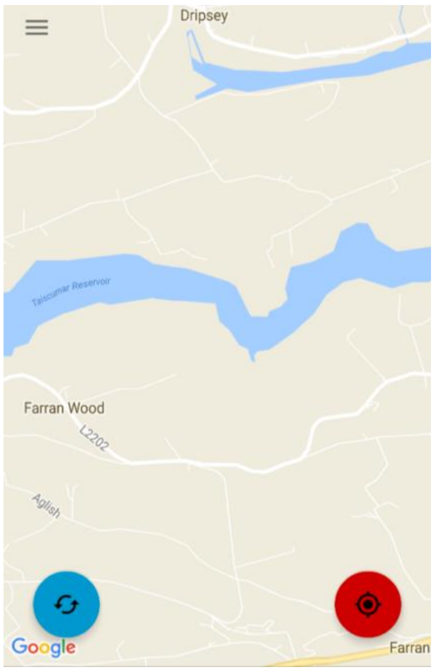


Figure 26: Main screen of the app

In the main screen, we see two buttons: the first one is used for updating the map manually if it failed, but by default it does it automatically. If we click the red button, we will move to the position we are actually. By default it goes to the position of the user. We can also move through the map and do the typical actions such as zoom in, zoom out and move.

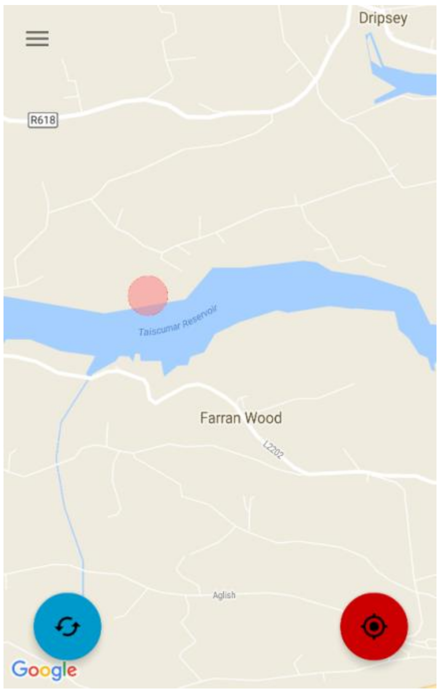
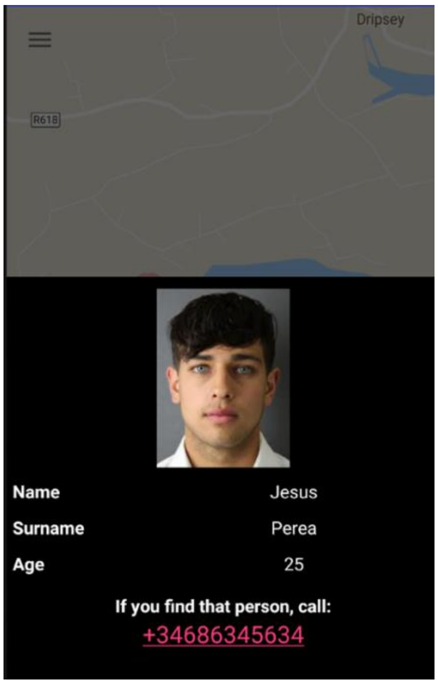


Figure 27: Main screen of the map with one marker

As we can see in the figure above, we see that somebody disappeared near the river. We can get all the information clicking in the red circle.



Name	Jesus
Surname	Perea
Age	25

If you find that person, call:
+34686345634

Figure 28: Panel of disappeared person

As we can see, a sliding panel appears with all the information needed. If we find that people, we can click to the contact phone and the phone dial will be shown.

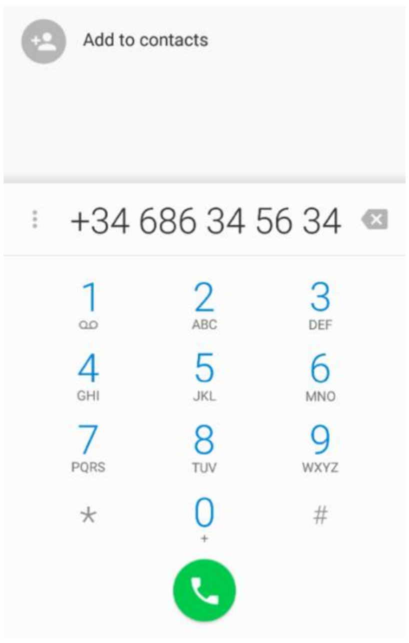


Figure 29: Phone dial

Now we are going to register a user for being able to add a person. In the navigation drawer, we click My Account, and using the bottom navigation, we select register.

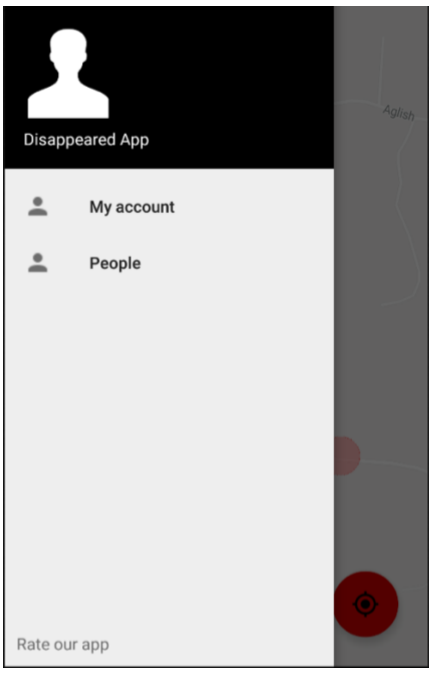


Figure 30: Navigation drawer of a non-logged user

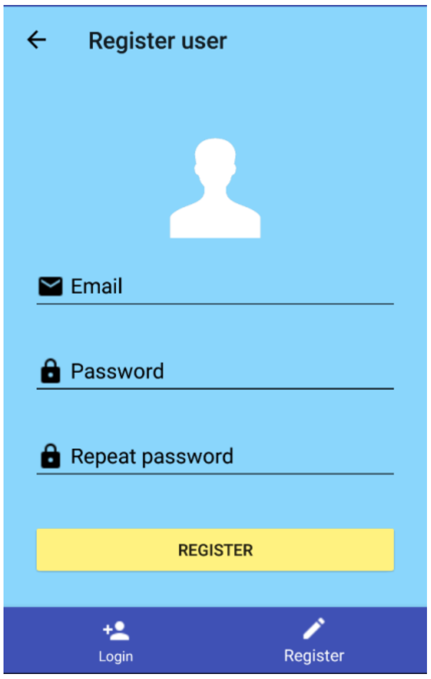


Figure 31: Register screen

When we've registered, a new screen will appear for verifying the account. An email will be sent with the verification code, but we can verify the account later. Simply we have to login and the verification screen will be redirected.

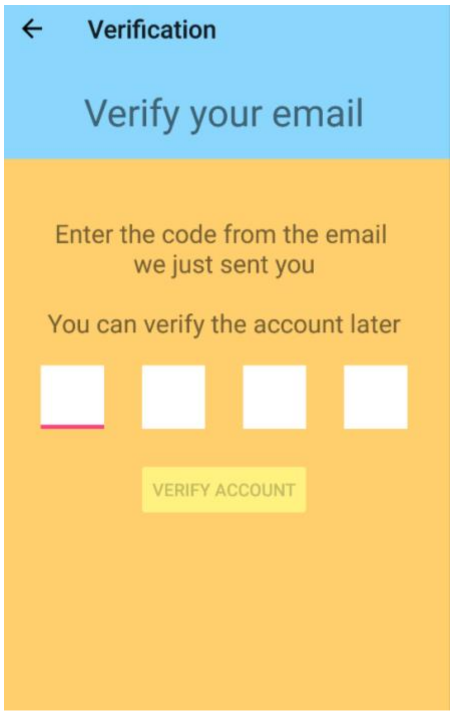

Verification
Verify your email
Enter the code from the email we just sent you
You can verify the account later
VERIFY ACCOUNT

Figure 32: Verification screen

When we are verificated, we can login normally.

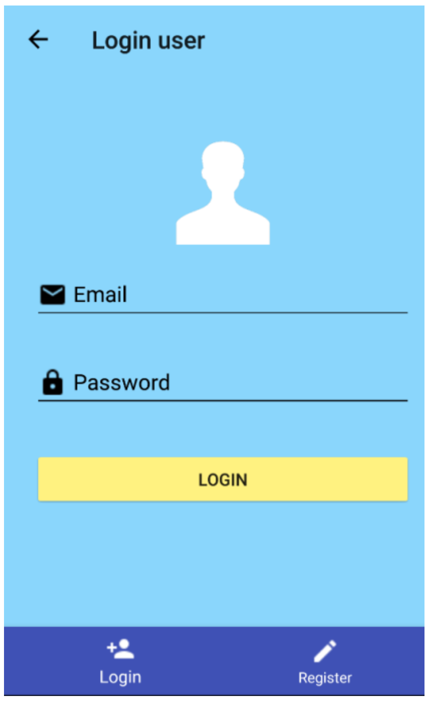


Figure 33: Login screen

Now that we’ve logged in, we can see the panel update, with the possibility of adding a person.

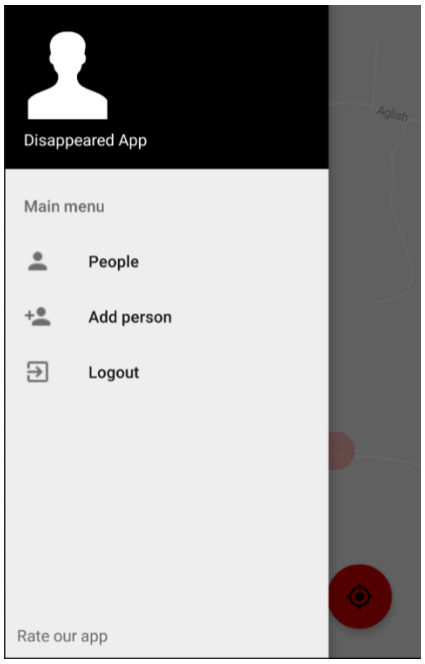


Figure 34: Navigation drawer of a logged user

Now we will add a user, clicking on the "Add person section". A new screen will be displayed asking for the location. By default, the camera is pointing in the user position. For marking a position, we just need to drag the centre pointer to the position we want. We can also click in the position and a new screen will appear asking for a direction.

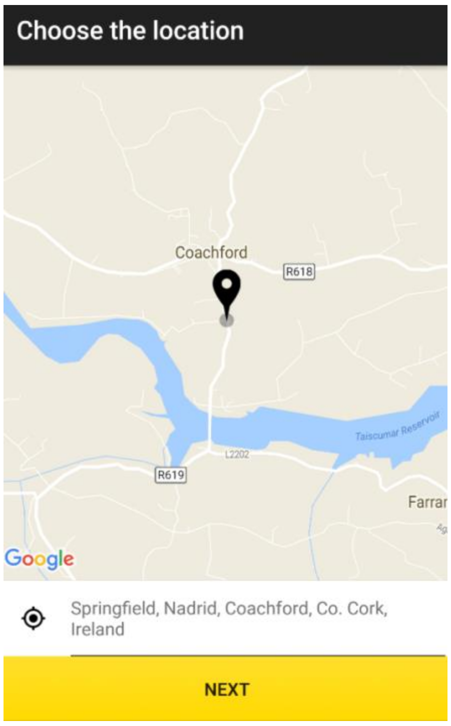


Figure 35: Choosing location screen

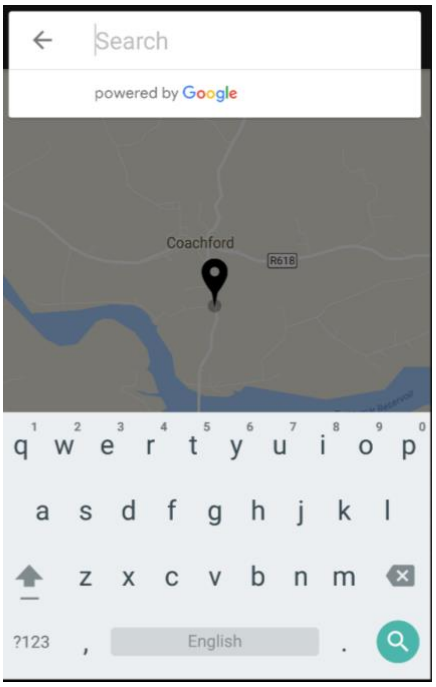


Figure 36: Choosing a location screen

If we input a direction, the map will be automatically moved. The next step is filling all the fields with the information of the user.

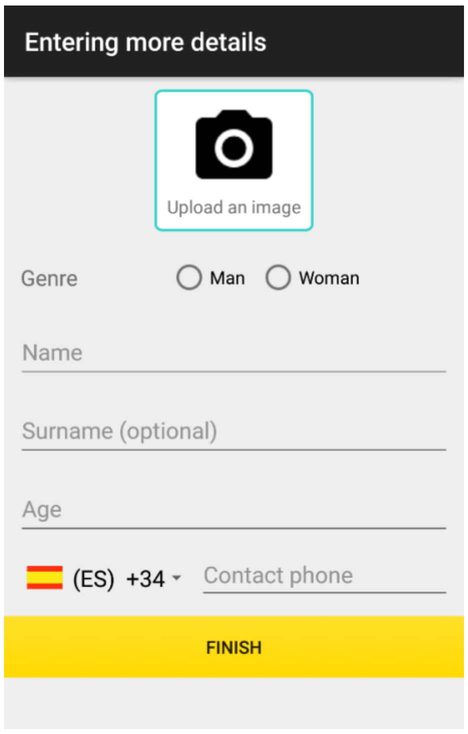


Figure 37: Entering details screen

If we click on upload an image, it will appear a new screen that allows choosing an image.

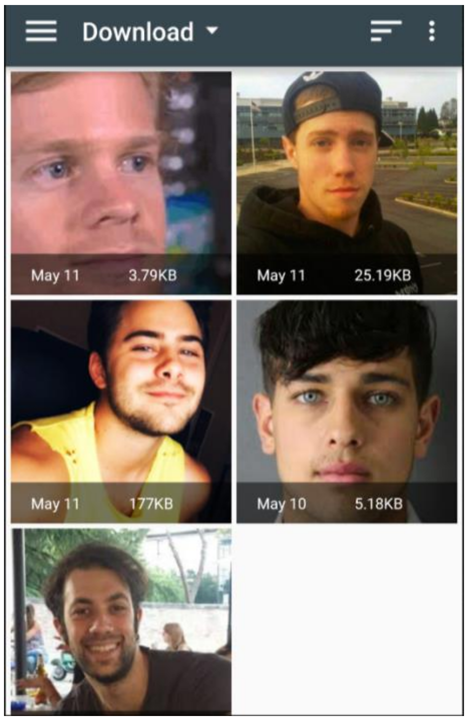
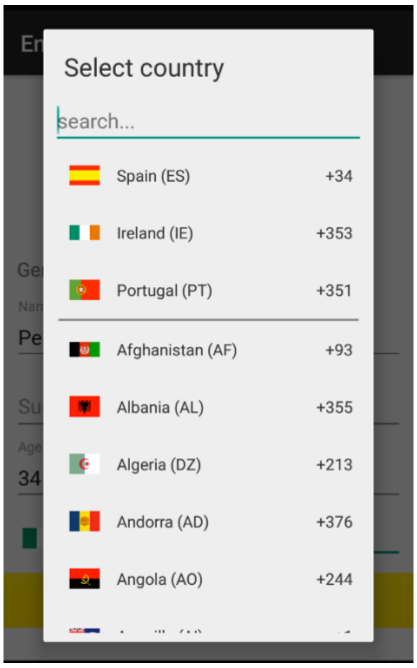


Figure 38: Screen choosing an image

In the contact phone, we can choose a different prefix depending on the country of the mobile phone.

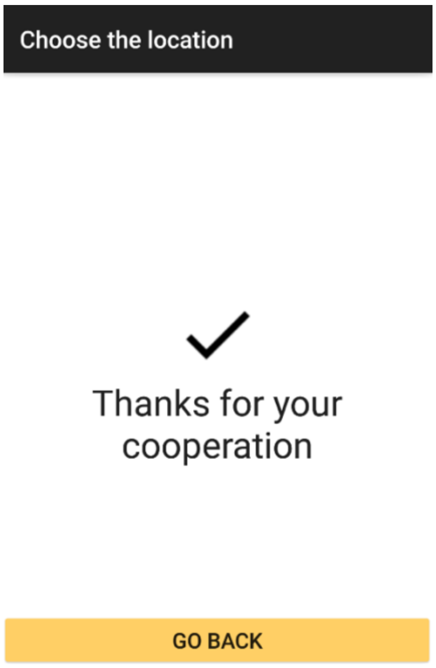

Select country

search...

🇪🇸	Spain (ES)	+34
🇮🇪	Ireland (IE)	+353
🇵🇹	Portugal (PT)	+351
🇦🇫	Afghanistan (AF)	+93
🇦🇱	Albania (AL)	+355
🇩🇿	Algeria (DZ)	+213
🇦🇩	Andorra (AD)	+376
🇦🇴	Angola (AO)	+244

Figure 39: Select country prefix screen

After filling all the information, if everything went well, a successful message will appear.


Choose the location

Thanks for your cooperation

Figure 40: Successful add screen

For managing the people we've added before, we simply go to "people" section in the lateral navigation drawer.

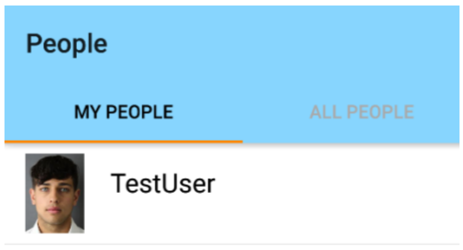


Figure 41: My people screen

If we click on the user, a popup will be displayed with all the information of the user, with the possibility of removing that user.

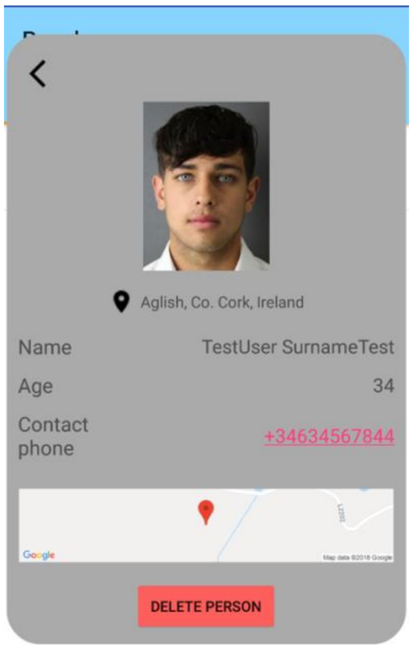

Aglish, Co. Cork, Ireland
Name TestUser SurnameTest
Age 34
Contact phone +34634567844
DELETE PERSON

Figure 42: User details section

If we click on the map, the user will be redirected to the position of that user in the map. The user is represented in the centre of the map.

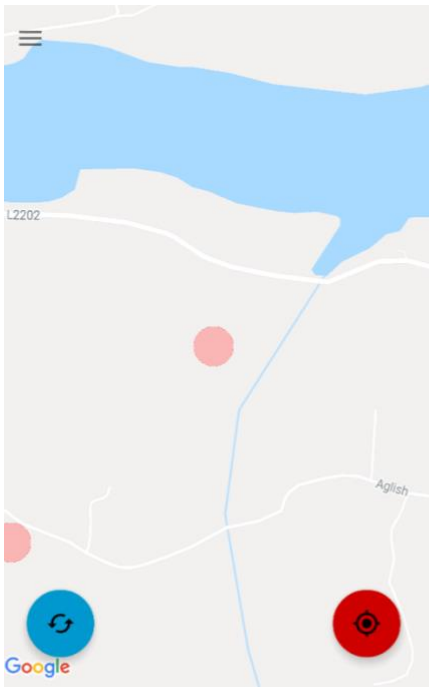
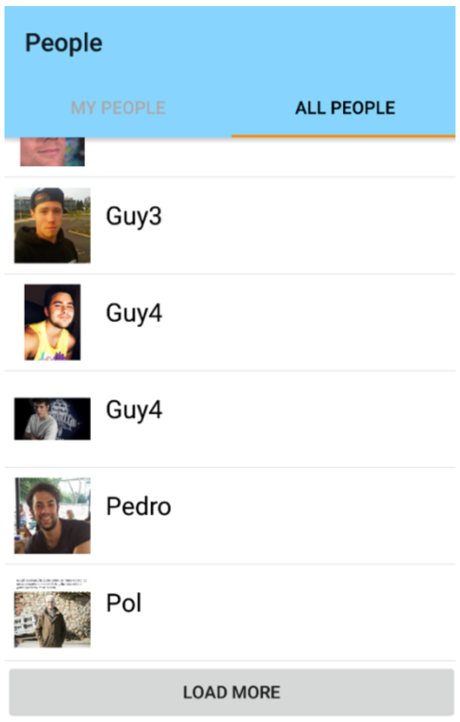


Figure 43: Redirection of user location screen

We can also see all the people who is added, and using pagination for not loading all the data.


People

MY PEOPLE | ALL PEOPLE

Guy3

Guy4

Guy4

Pedro

Pol

LOAD MORE

Figure 44: All people screen

Appendix B: API Manual

In this section I will describe all the methods that the user can interact with the server to get or add information.

GET /allpeople/:page

Description: Returns an array with all the people that is in the page page. Also it returns the total number of pages.

Input fields: Page (integer)

Example of output:

```
{
    "people": [
        {
            "_id": "5b2f7ec04d51a471447eb91b",
            "name": "ElDto",
            "latitude": 41.610899975838294,
            "longitude": 0.6486067548394203,
            "contact_phone": "+34653234566",
            "direction": "Carrer Palauet, 131, 25001 Lleida, Spain",
            "age": 24,
            "user": "5b267fce0d07c46644f47e1d",
            "image": "/users/K50UK1TDgbXMSUd33nvXDIB-.png"
        }
    ],
    "total_pages": 2
}
```

GET /mypeople

Description: Returns an array with all the people that are added by a user without pagination.

Input fields: Authorization header (string)

Example of output:

```
{
    "people": [
        {
            "_id": "5b2e59a1885d7360577c6187",
            "name": "Guy4",
            "latitude": 51.8790982930885,
            "longitude": -8.772150166332722,
            "contact_phone": "+34656345677",
            "direction": "Unnamed Road, Co. Cork, Ireland",
            "age": 34,
            "user": "5b267fce0d07c46644f47e1d",
            "image": "/users/3aWlKVRipNGPt7EI33_IfSdI.png"
        }
    ]
}
```

DELETE /deleteperson/:id_person

Description: Deletes a user with the id id_person. It needs the authorization header for checking if the user is the one who added the person.

Input fields: Authorization header (string), id_person (string)

Returns a 200 response if the user is removed successfully.

POST /addperson

Description: Adds a user in the database. The type of request has to be *multipart/form-data*.

Input fields: Authoritzation header (String), name (String), surname (String, optional), age (Number), contact_phone (String), latitude (Number), longitude (Number), image (Image)

Returns the data added.

POST /login

Description: Method user for logging in a user.

Input fields: verification_code (Number), email (String)

Returns the token if the login is successful. If the user is not verified, returns a 433 response.

POST /verificate

Description: Verificates a user

Input fields: verification_code (Number), email (String)

Returns a 200 response if the user is verificated correctly. Otherwise, a 400 response.

GET /getusersbylocation

Description: Returns all the users that are in a specific coordinates.

Input fields: southwest_latitude (Number), southwest_longitude (Number), northeast_latitude (Number), northeast_longitude (Number)

Example of output:

```
{
    "users": [
        {
            "latitude": 51.89494864376246,
            "longitude": -8.789026290178299
        },
        {
            "latitude": 51.888982267209165,
            "longitude": -8.790567554533482
        },
        {
            "latitude": 51.89243783412591,
            "longitude": -8.786109387874603
        }
    ]
}
```

GET /getuserbylocation

Description: Returns all the info of a user that is in a location

Input fields: latitude (Number), longitude (Number)

Example of output:

```
{
    "createdAt": "2018-05-10T19:36:15.861Z",
    "name": "Antonio",
    "contact_phone": "+34645345677",
    "age": 34,
    "surname": "Perea",
    "image": "UCxv3uaVk62Gcaa1NO2s-1CJ.jpeg"
}
```