Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

EmergenciAPP – Red social de emergencias

EmergenciAPP – Emergencies social network

Francisco J. Mendoza Álvarez

La Laguna, 9 de septiembre de 2020

D. **Alejandro Pérez Nava**, con N.I.F. 43821179-S profesor Asociado de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Fernando Pérez Nava**, con N.I.F. 42091420-V profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A N

Que la presente memoria titulada:

"EmergenciAPP – Red Social de emergencias"

ha sido realizada bajo su dirección por D. **Francisco J. Mendoza Álvarez**,

con N.I.F. 43832003-Y.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 9 de septiembre de 2020

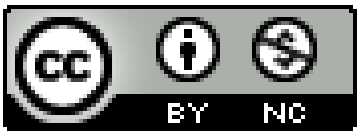
Agradecimientos

Agradecer la ayuda prestada por los tutores de este proyecto y por la paciencia demostrada durante el proceso.

Gracias a mi familia por su amor y apoyo incondicional desde mi nacimiento, que se mantiene siendo un adulto.

Y finalmente agradecer a mi pareja por su apoyo incondicional en este largo camino recorrido juntos.

Licencia



Resumen

El objetivo de este trabajo, como su nombre indica, es generar lo que llamamos una "red social de emergencias". Esto se traduce en la implementación de un modelo que permita conectar a personas con conocimientos en primeros auxilios, generalmente personal sanitario, que desean ayudar desinteresadamente, con personas que necesitan dicha ayuda al encontrase en un estado de emergencia.

La idea es simple, un usuario "general" con una emergencia sanitaria envía una alerta, ésta es enviada a todos los usuarios "sanitarios" que se encuentren dentro de un radio determinado, indicando la ubicación de la alerta y la distancia estimada desde su ubicación actual. Si un usuario decide aceptar la solicitud de asistencia, se muestran las indicaciones necesarias de navegación para llegar hasta la ubicación donde se ha realizado la petición y, por otro lado, se notifica al usuario solicitante de que la ayuda está en camino. Una vez en el lugar, el usuario "sanitario", puede consultar cierta información de interés sobre el usuario afectado.

Para ello se ha desarrollado una aplicación móvil para el sistema operativo Android, empleando el lenguaje de programación Java. Y por otro lado una API (interfaz de programación de aplicaciones) desde la que se proporcionan las funciones y datos necesarios para el funcionamiento de la aplicación, desarrollado en PHP y haciendo uso del framework Slim. Que a su vez se puede emplear para desarrollar aplicaciones para otros sistemas operativos, minimizando el esfuerzo.

Como recomendación dicha implementación podría ser empleada a su vez por los servicios de emergencia para facilitar su labor, ya que se conoce la ubicación del afectado, además de datos importantes para su correcto tratamiento. Para ello bastaría desarrollar una aplicación web que conecte con la API existente.

Palabras clave: *Java, PHP, API, Framework, Slim.*

Abstract

The objective of this work, as its name indicates, is to generate what we call an "emergencies social network". This translates into the implementation of a model that allows connecting people with knowledge of first aid, generally health personnel, who wish to help selflessly, with people who need such help when they are in a state of emergency.

The idea is simple, a "general" user with a health emergency sends an alert, this is sent to all "health personnel" users who are within a certain radius, indicating the location of the alert and the estimated distance from their current location. If a user decides to accept the request for assistance, the necessary navigation directions are displayed to reach the location where the request was made and, on the other hand, the requesting user is notified that help is on the way. Once at the site, the "health personnel" user can consult certain information of interest about the affected user.

For this, a mobile application has been developed for the Android operating system, using the Java programming language. And on the other hand, an API (application programming interface) from which the functions and data necessary for the operation of the application are provided, developed in PHP and using the Slim framework. Which in turn can be used to develop applications for other operating systems, minimizing effort.

As a recommendation, said implementation could be used in turn by the emergency services to facilitate their work, since the location of the affected person is known, as well as important data for their correct treatment. For this, it would be enough to develop a web application that connects with the existing API.

Keywords: Java, PHP, API, Framework, Slim.

Índice general

Introducción **1**
Objetivos **2**
Tecnologías empleadas **3**
1.1 Diseño de interfaz 3
1.2 Desarrollo Android 3
1.3 Servidor 3
1.3.1 API 3
1.3.2 Base de datos 3
Planificación y diseño **4**
1.1 Modelo 4
1.2 Interfaz gráfica 4
Desarrollo **11**
1.1 Base de datos 11
1.1.1 Datos del usuario: 11
1.1.2 Preferencias y datos de notificación 11
1.1.3 Alertas 12
1.2 API 12
1.2.1 /users 12
1.2.2 /alert 13
1.3 Implementación Android 13
Conclusiones y líneas futuras **23**
Conclusions **24**

Índice de figuras

Figura 1: Pantalla de inicio 4
Figura 2: Pantalla de login 5
Figura 3: Paso 1 del registro 5
Figura 4: Paso 2 del registro 6
Figura 5: Paso 3 del registro 6
Figura 6: Pantalla principal 7
Figura 7: Menú 7
Figura 8: Solicitud de asistencia 8
Figura 9: Solicitud enviada 8
Figura 10: Cancelar solicitud enviada 9
Figura 11: Solicitud aceptada 9
Figura 12: ER Users 11
Figura 13: ER Notification_preferences 12
Figura 14: ER Alerts 12
Figura 15: Estructura del proyecto 14
Figura 16: APIEngine.java 15
Figura 17: UserService.java 16
Figura 18: AlertService.java 16
Figura 19: Función startDots() 18
Figura 20: Funciones addDotsIndicator(position)
y setTransparence(position, transparence) 19
Figura 21: Funciones checkTime() y getEstimatedTime() 20
Figura 22: Función changeTime(time) 20
Figura 23: Función onNewIntent 21

Introducción

En la actualidad las personas conviven rodeadas de tecnología, empleándola a diario en su día a día, incluso sin percatarse en ocasiones, facilitando muchas tareas cotidianas y mejorando su calidad de vida.

Más de la mitad de la población mundial, 5.190 millones de personas a inicios de 2020, dispone de un teléfono móvil, por lo que parece lógico aprovechar esta infraestructura ya disponible para el fin que nos atañe.

La intención de este proyecto es desarrollar una aplicación móvil que permita a usuarios en situaciones en las que la asistencia sanitaria por parte de las autoridades locales no sea factible, recibir asistencia de personas con la formación adecuada como médicos, enfermeros, socorristas, etc… que deseen ayudar de forma altruista y se encuentren en un rango de distancia cercana al usuario solicitante. Generando así una “red social” de emergencias.

En los últimos años se han desarrollado aplicaciones enfocadas en asistir a personas en estado de emergencia, sin embargo, todas se enfocan en un publico especifico, como por ejemplo deportistas de riesgo como escaladores o montañistas, personas con enfermedades diagnosticadas para llevar un seguimiento de las mismas, o para evitar situaciones de violencia de genero. A su vez es bastante habitual que los propios servicios de emergencia tengan sus propias apps para gestionar las incidencias, pero sólo son validas para ese servicio en concreto.

Sin embargo, el objetivo de este proyecto es crear una herramienta simple, para que pueda ser usada aún teniendo poca familiaridad con dispositivos móviles, y para un público general, pudiendo ser empleada en todas las situaciones mencionadas con anterioridad y cualquier situación en la que los usuarios necesiten asistencia urgente que se presente.

Objetivos

El objetivo como ya se ha comentado, es el de conectar usuarios en situaciones de emergencia donde se necesita asistencia profesional, con otros usuarios que posean conocimientos en primeros auxilios y asistencia sanitaria que actúan de manera voluntaria y altruista proveyendo la asistencia necesaria. Es por ello que se requiere desarrollar una aplicación móvil para el sistema operativo Android que cumpla con los siguientes requisitos:

- Pueden existir dos tipos de usuario, el público en general y el personal sanitario, siendo usuarios con necesidad de asistencia y los usuarios capaces de proveerla, respectivamente.
- Debe existir un proceso de alta de usuarios para poder gestionar correctamente su información.
- Es necesario que los usuarios puedan ser geolocalizados, tanto para poder realizar avisos, como para acudir a los mismos.
- La solicitud de asistencia, debe ser un proceso simple y fácil de ejecutar.
- Una vez que un usuario envía una alerta, los usuarios “sanitarios” dentro de un radio específico, deben recibir una notificación. En ésta podrán decidir si aceptar o reclinar la solicitud.
- Los usuarios “sanitarios” podrán indicar el radio desde su ubicación en el que desean recibir alertas.

Tecnologías empleadas

1.1 Diseño de interfaz

Para la planificación y el diseño de la interfaz gráfica de la aplicación se ha hecho uso de la herramienta **Adobe XD**.

Por otro lado, se ha empleado **Adobe Photoshop** para el diseño de logos e iconos y algunos recursos gráficos extraídos de la web **Material Desing** de Google.

1.2 Desarrollo Android

En cuanto a la parte de desarrollo, se ha utilizado **Android Studio**, el IDE de desarrollo oficial de Google para el desarrollo Android, empleando como lenguaje de programación **Java**. A su vez se ha hecho uso de usado varias librerías como:

- **Firebase messaging**. Empleado para poder enviar notificaciones a los usuarios
- **Retrofit 2**. Para consumir recursos desde el API
- **Gson converter**. Para poder manejar fácilmente las respuestas del servidor convirtiendo ésta de formato JSON a objetos manipulables desde la aplicación.

1.3 Servidor

Para el despliegue del servidor, en un inicio se ha empleado el conjunto de herramientas **XAMPP** para su desarrollo en local, una vez finalizado el mismo, se ha migrado al host gratuito **000webhostapp**, ya que aparte de ser gratuito, posee los requisitos necesarios para el funcionamiento de la API diseñada.

1.3.1 API

La API ha sido desarrollada con el IDE **Visual Studio Code**, empleando el leguaje de programación **PHP** y la librería **Slim** que facilita la creación de APIs ya que posee métodos específicos para llamadas GET, POST, etc., funciones para la gestión de las URLs y los parámetros enviados al servidor y la gestión de las respuestas devueltas por parte del mismo.

1.3.2 Base de datos

Para almacenar la información de los usuarios y otros datos de importancia para el correcto funcionamiento de la aplicación se ha utilizado el gestor de bases de datos relacionales **MariaDB** y para la administración de la misma, durante la fase de desarrollo se ha empleado la herramienta **phpMyAdmin**.

Planificación y diseño

1.1 Modelo

Durante el inicio del proyecto se ha llevado a cabo una reunión con el tutor del proyecto, en la que se comentaron los detalles del mismo y los requisitos de implementación. Acordando emplear SCRUM como metodología de desarrollo.

Tras ello, se realizó un primer modelo de lo que se deseaba implementar, determinando cuales sería los pasos a seguir, las características a implementar y el funcionamiento esperado de la aplicación.

1.2 Interfaz gráfica

En cuanto a la interfaz gráfica se ha realizado un diseño de como se desea que fuese la misma, dando como resultado las siguientes pantallas:

Figura 1: Pantalla de inicio

Esta pantalla se muestra al iniciar la aplicación.

La función de dicha pantalla es mostrar al usuario que la aplicación se encuentra en ejecución, pero se están realizando tareas para poder continuar.

Está compuesta por dos campos donde se solicita el email y la contraseña para realizar el proceso de login al pulsar el botón "*Entrar*".
En caso de que el usuario no posea cuenta, puede realizar el registro pulsando sobre "¿Aún no tienes cuenta?"

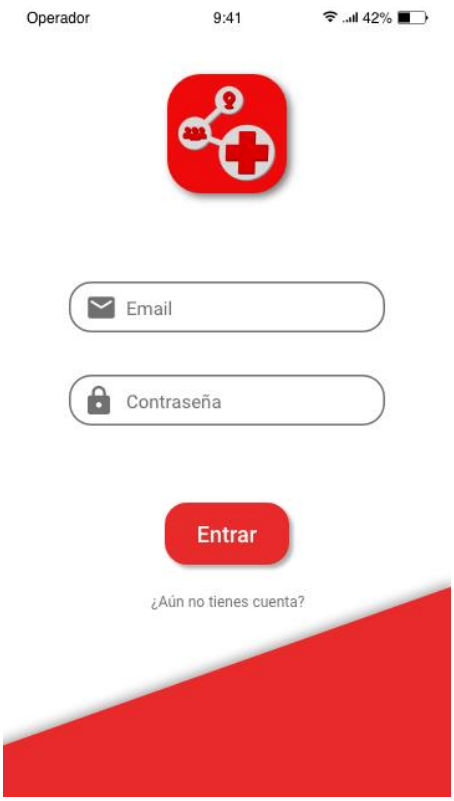


Figura 2: Pantalla de login

Figura 3: Paso 1 del registro

El usuario debe introducir los datos requeridos para la creación de la cuenta.

Esta pantalla está compuesta por varias vistas que componen cada uno de los tres pasos que componen el registro.

En el primer paso del proceso, se requiere indicar email, contraseña y seleccionar el tipo de usuario al que se pertenece.

En caso de que los emails y/o la contraseña no coincidan, se muestran los correspondientes mensajes de error, en caso contrario, se permite avanzar al siguiente paso, pulsado el botón.

En el segundo paso del registro se requieren los datos personsales del usuario. Entre ellos Nombre, apellidos, DNI, fecha de nacimiento, y el sexo del usuario.
Si no se rellenan los campos requeridos, se muestra un mensaje de error e impide avanzar al siguiente paso.

Figura 4: Paso 2 del registro

Figura 5: Paso 3 del registro

Tercer y último paso del registro, donde se deben indicar datos de interés clínico, como son el grupo sanguineo, alergias y cualquier otro detalle de interés, como tratamientos, enfermedades crónicas, etc.
Completado este paso finaliza el proceso de registro y se redirige al login.

Desde la pantalla principal se pueden realizar dos acciones.
Por una parte pulsando el botón central con el mensaje "*SOS*" se inicia un apetición de asistencia.
Y por otro lado, pulsado el botón de la esquina superior izquierda se despliega el menú lateral.

Figura 6: Pantalla principal

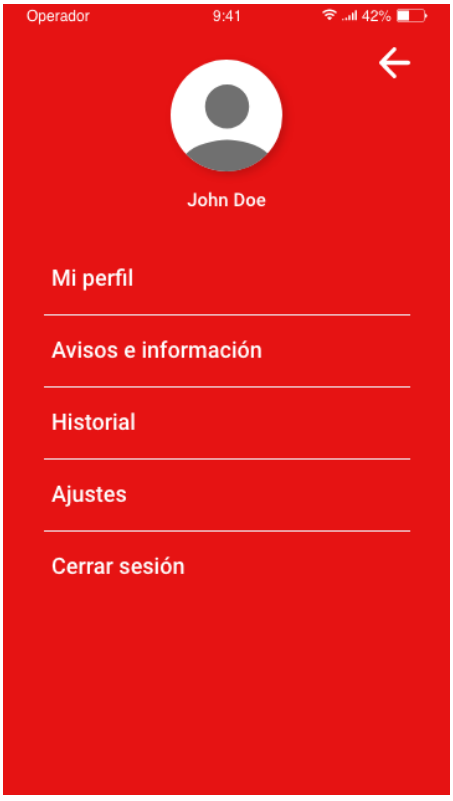
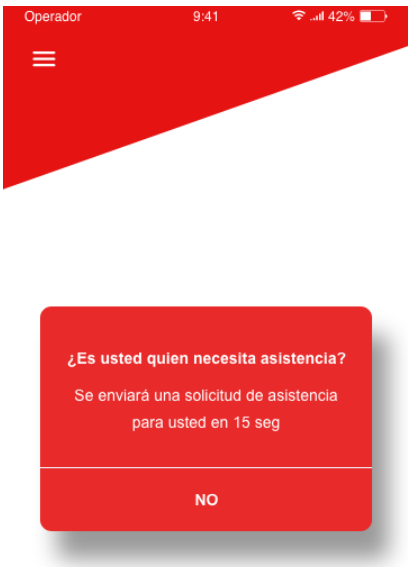


Figura 7: Menú

Desde el menú lateral se pueden acceder a distintas pantallas que no son impresindibles para el funcionamiendo de la aplicación pero confieren ciertas funcionalidades extras.

- "*Mi perfil*", donde se puede consultar la información personal ingresada en el registro.
- "*Avisos e información*" desde aquí se pueden consultar canales de comunicación de organismos locales oficiales.
- "*Historial*", para poder consultar el historial de peticiones realizadas por el usuario
- "*Ajustes*", para cambiar ciertas características de la aplicación, como por ejemplo la frecuencia de muestreo de la localización o el radio para recibir alertas.
- "*Cerrar sesión*", permite finalizar la sesión actual y redirige al login.



Esta pantalla se muestra al pulsar el botón de solicitud de asistencia.
Tras pulsar se inicia un contador con un tiempo determinado, transcurrido éste, se envía una solicitud de asistencia si no se cancela previamente la misma.
Pulsando en el botón de "*NO*" o fuera de la ventana emergente, se cancela el envío.

Figura 8: Solicitud de asistencia

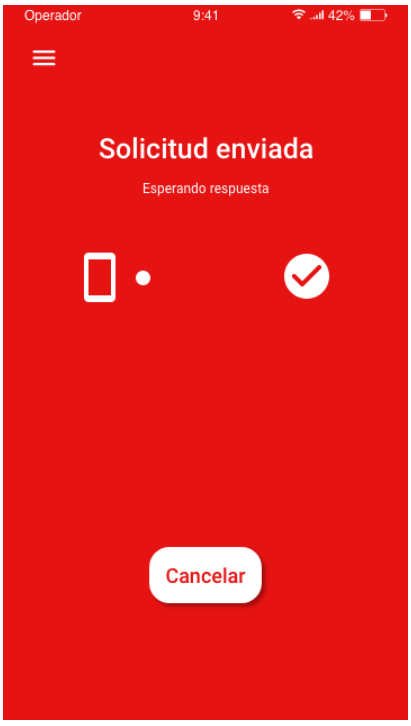


Figura 9: Solicitud enviada

Esta pantalla se muestra tras enviar una solicitud de asistencia mientras el usuario espera a que un usuario del tipo sanitario acepte la misma.
Durante éste periodo aún es posible cancelar la solicitud, pulsando sobre el botón "*Cancelar*". Ésta es la unica acción que puede realizar el usuario en esta pantalla.

Mientras no haya sido aceptada la solicitud, es posible cancelarla pulsando el botón correspondiente.
Tras ello se muestra una ventana emergente de confirmación, al pulsar sobre el botón "*SI*", se envía un apetición al servidor para cancelar la solicitud, en caso contrario, se cierra la ventana emergente y continua la espera.

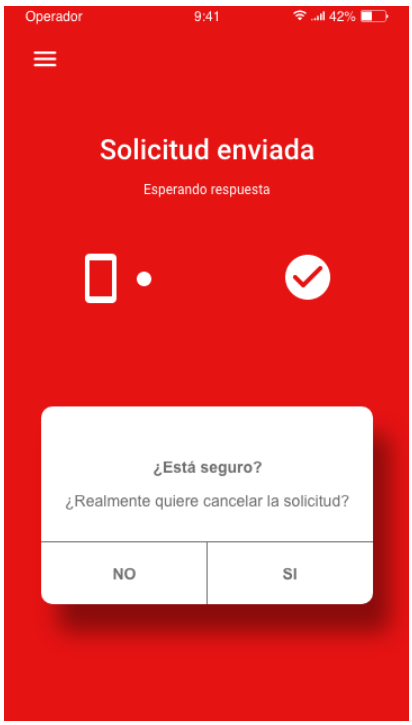


Figura 10: Cancelar solicitud enviada

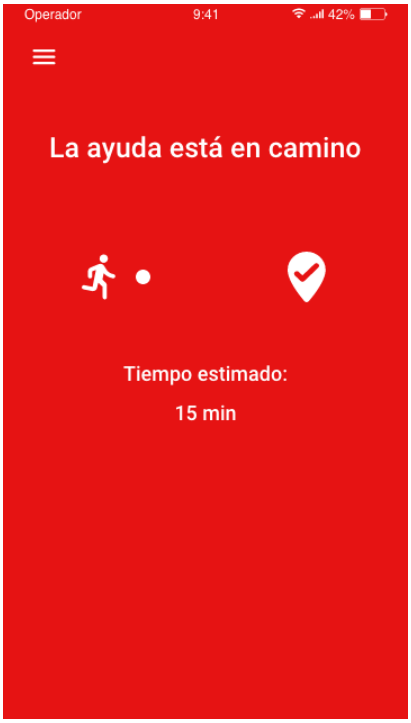
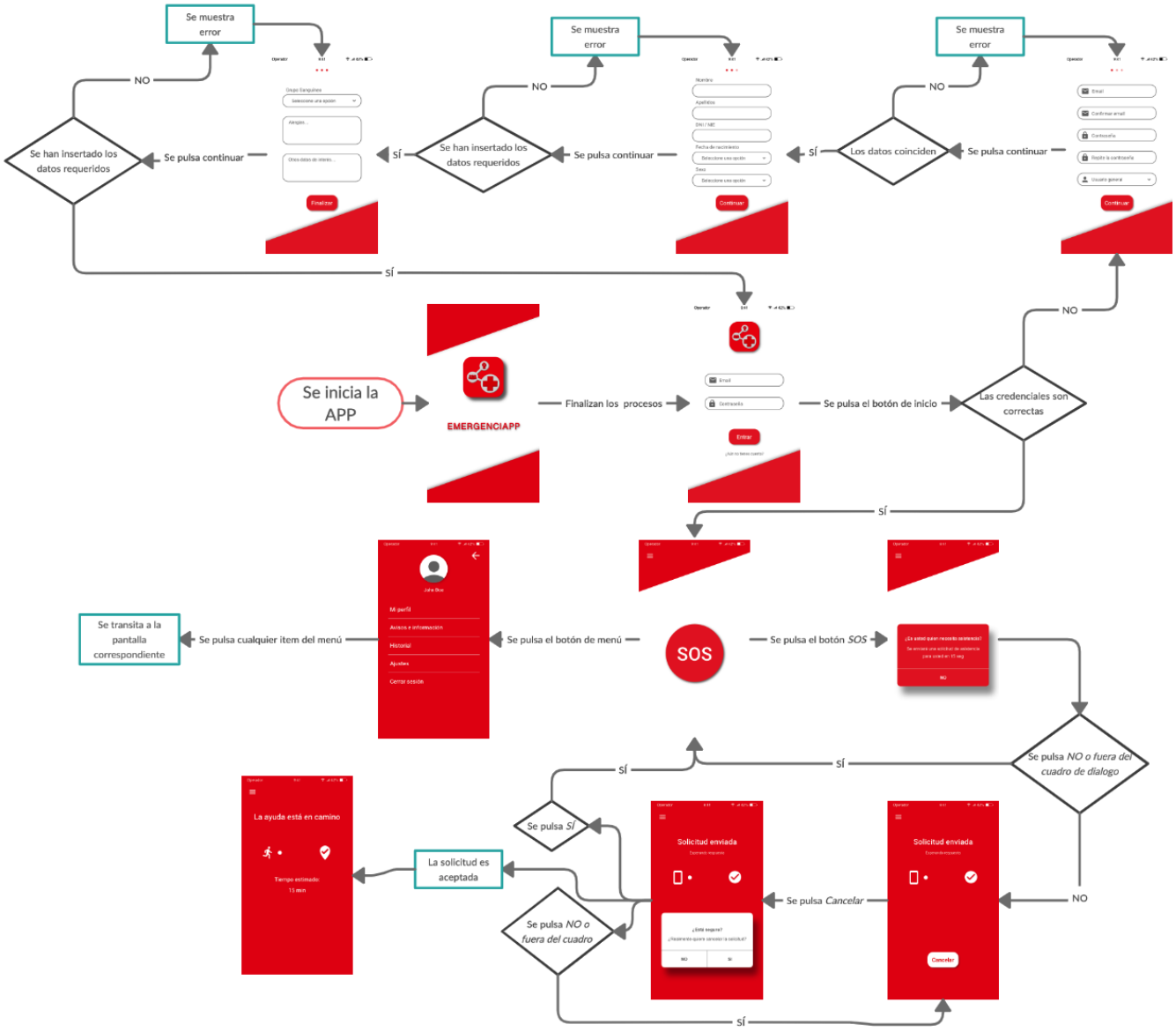


Figura 11: Solicitud aceptada

Una vez se ha aceptado la solicitud, se muestra ésta pantalla donde se indica el tiempo en el que se estima que llegue la ayuda.
Éste tiempo se va actualizando periodicamente mientras el usuario del tipo sanitario que atiende la sulicitud se aproxima hasta el lugar.

Para comprender mejor el planteamiento del funcionamiento que debe presentar la aplicación se ha diseñado el siguiente diagrama de flujo:



Desarrollo

Todo el código desarrollado, puede ser consultado en el siguiente enlace.

1.1 Base de datos

Para la base de datos se ha diseñado e implementado el siguiente modelo:

1.1.1 Datos del usuario:

Los datos referentes al perfil de cada usuario se ha optado por dividirlos en tres tablas dividiendo los datos según su ámbito. Para ello se ha creado una tabla *Users* donde se almacenan un identificador de usuario, su email y contraseña, que son empleados como credenciales de acceso, y finalmente un identificador del tipo de usuario. Otra llamada *Users_Data*, que contiene la información personal del usuario, como, nombre, apellidos, dni, etc… Y finalmente *Users_medicalData* donde se almacenan los datos médicos del usuario, como alergias, grupo sanguíneo o cualquier otra información relevante como enfermedades crónicas o tratamientos.

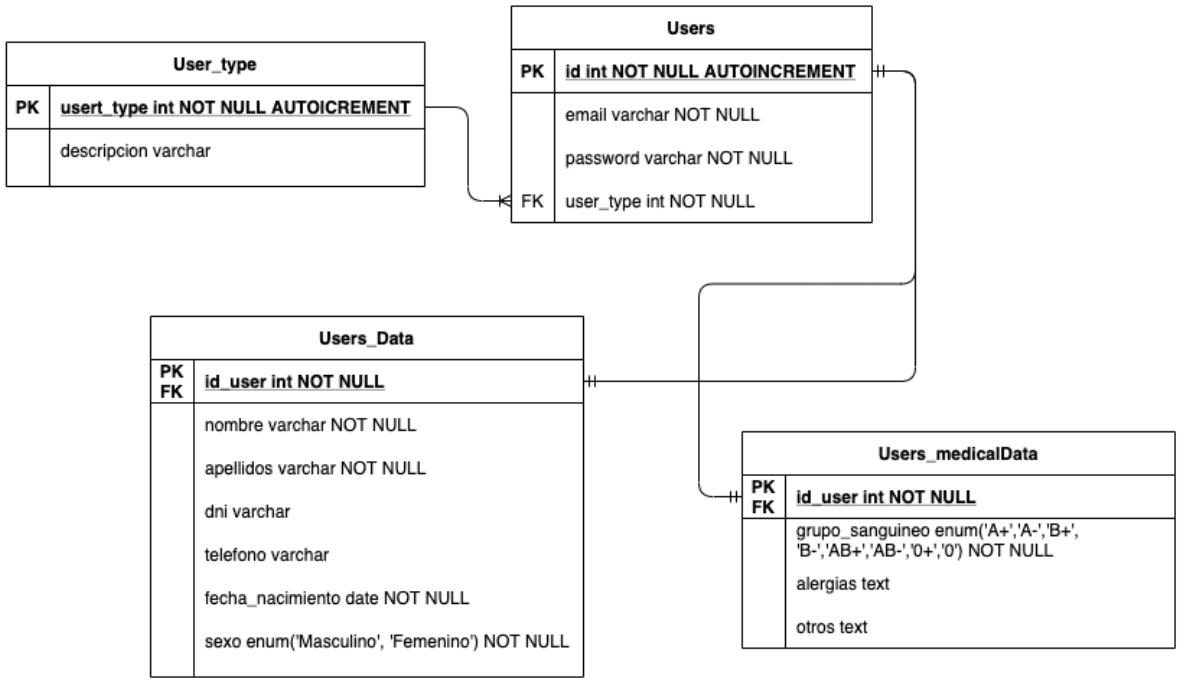


Figura 12: ER Users

1.1.2 Preferencias y datos de notificación

Se ha optado por emplear una tabla para almacenar los datos necesarios para notificar a los usuarios, así como sus preferencias en cuanto a las mismas. Por ello *Notifications_preferences* contiene el identificador del usuario al que pertenece la información, su token para notificaciones, su última ubicación conocida, así como su marca de tiempo, y por último el radio en el que el usuario desea recibir las alertas.

Notifications_preferences	
PK FK	id_user int NOT NULL
	notif_token varchar
	latitude decimal
	longitude decimal
	lastLocationUpdate timestamp
	radius int

Figura 13: ER Notification_preferences

1.1.3 Alertas

Las alertas son almacenadas en la tabla *Alerts*, en la misma se almacenan un identificador de alerta, el identificador de usuario y la ubicación en el momento de la alerta para ambos usuarios, tanto el usuario que envía la alerta como el que la recibe, y el momento en el que es aceptada la misma. Dando como resultado el siguiente esquema:

Alerts	
PK	id int NOT NULL AUTOINCREMENT
FK	id_user int NOT NULL
FK	id_sanitary int
	user_latitude decimal NOT NULL
	user_longitude decimal NOT NULL
	sanitary_latitude decimal
	sanitary_longitude decimal
	accepted timestamp

Figura 14: ER Alerts

1.2 API

En cuanto a la API, se ha empleado la librería Slim para facilitar su desarrollo. Se ha dividido en varias rutas para que su uso sea mas intuitivo.

La API se encentra publicada en la siguiente URL: https://emergenciapp.000webhostapp.com/

1.2.1 /users

En esta ruta se han añadido los métodos necesarios para manejar la información de los usuarios.

- **GET /login/{email}/{password}**

  Permite realizar el login en la aplicación, indicando el email y la contraseña.

- **POST /register**

  Permite crear un nuevo usuario, para ello se deben indicar los datos requeridos en las Figuras 3,4,5.

- **GET /id/{email}**

  Devuelve el identificador de un usuario especificando su email.

- **GET /data/{id}**

  Retorna la información sobre un determinado usuario

- **/notificationPreferences**
  - **POST /token**

    Actualiza el token de notificaciones, proporcionado por Firebase Cloud Messaging, almacenado en el servidor de un usuario determinado.

  - **POST /location**

    Actualiza la localización almacenada en el servidor de un usuario determinado.

  - **POST /radius**

    Determina el radio en el que un usuario desea recibir alertas.

1.2.2 /alert

En esta ruta se agrupan los métodos necesarios para la gestión de las alertas

- **POST /sendAlert**

  Genera una nueva alerta y envía una notificación a los usuarios que se encuentren dentro del radio determinado.

- **POST /deleteAlert**

  Cancela una solicitud de asistencia ya creada que aún no ha sido aceptada.

- **POST /aceptAlert**

  Se emplea para aceptar una solicitud de asistencia recibida.

- **GET /check/{id}**

  Devuelve el estado de una solicitud de asistencia, indicando si ésta ya ha sido aceptada o no.

- **GET /getEstimatedTime/{id}**

  Retorna el tiempo estimado en el que se espera que el usuario que ha aceptado una determinada alerta llegue al lugar de la misma.

1.3 Implementación Android

En cuanto al desarrollo de la propia aplicación Android, la estructura final obtenida en el proyecto es la siguiente:

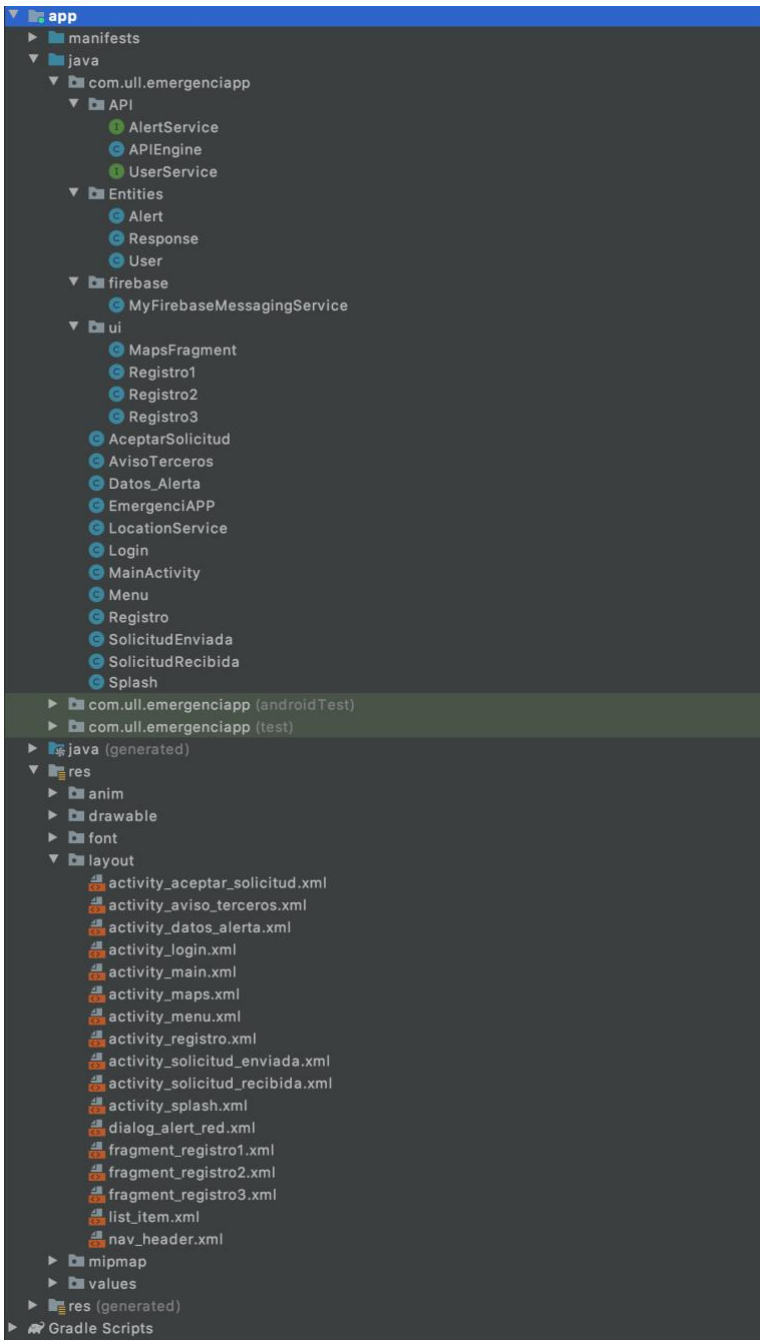


Figura 15: Estructura del proyecto

Como se puede observar, el paquete *com.ull.emergenciapp* contiene multitud de clases, pertenecientes cada una a cada una de las pantallas que componen la aplicación. Estas clases pertenecientes a Activitys no poseen ninguna característica relevante excepto algunas particularidades que se comentarán mas adelante. Por otro lado, se pueden contemplar otros paquetes cuyo contenido detallaremos a continuación:

- **API**

  En este paquete se encuentran las clases desarrolladas para poder comunicarse con la API.

  La principal es *APIEngine*, donde se crea un objeto del tipo *Retrofit* con los argumentos correspondientes, para poder comunicarse con la API implementada.

Su contenido es el siguiente:

```java
package com.ull.emergenciapp.API;

import okhttp3.OkHttpClient;
import okhttp3.logging.HttpLoggingInterceptor;
import retrofit2.Retrofit;
import retrofit2.converter.gson.GsonConverterFactory;

public class APIEngine {

    private static Retrofit retrofit= null;

    public static Retrofit getAPIEngine(){
        if(retrofit == null) {
            HttpLoggingInterceptor interceptor = new HttpLoggingInterceptor();
            interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);
            OkHttpClient client = new OkHttpClient.Builder().addInterceptor(interceptor).build();

            retrofit = new Retrofit.Builder()
                    .baseUrl("https://emergenciapp.000webhostapp.com/")
                    .addConverterFactory(GsonConverterFactory.create())
                    .client(client)
                    .build();
        }
        return retrofit;
    }
}

```

Figura 16: APIEngine.java

A su vez se debe crear una interfaz para indicar los métodos que pueden ser llamados por el objeto *Retrofit.* En este caso se ha optado por implementar una por cada grupo de rutas de la API, llamados *UserService* y *AlertService.* Sus implementaciones serían las siguientes:

```java
package com.ull.emergenciapp.API;

import ...

public interface UserService {
    String group = "users/";
    String subgroup = "notificationPreferences/";

    @GET(group + "login/{email}/{password}")
    Call<Response<User>> login(@Path("email") String email, @Path("password") String password);

    @Multipart
    @POST(group + "register")
    Call<Response<User>> register(@Part("email") RequestBody email, @Part("password") RequestBody password,
                                  @Part("user_type") RequestBody user_type, @Part("nombre") RequestBody nombre,
                                  @Part("apellidos") RequestBody apellidos, @Part("dni") RequestBody dni,
                                  @Part("telefono") RequestBody telefono, @Part("fecha_nacimiento") RequestBody fecha_nacimiento,
                                  @Part("sexo") RequestBody sexo, @Part("grupo_sanguineo") RequestBody grupo_sanguineo,
                                  @Part("alergias") RequestBody alergias, @Part("otros") RequestBody otros);

    @Multipart
    @POST(group + subgroup + "token")
    Call<Response<String>> setToken(@Part("id") RequestBody id, @Part("token") RequestBody token);

    @Multipart
    @POST(group + subgroup + "location")
    Call<Response<String>> setLocation(@Part("id") RequestBody id, @Part("latitude") RequestBody latitude,
                                       @Part("longitude") RequestBody longitude);

    @Multipart
    @POST(group + subgroup + "radius")
    Call<Response<String>> setRadius(@Part("id") RequestBody id, @Part("radius") RequestBody radius);
}
```

Figura 17: UserService.java

```java
package com.ull.emergenciapp.API;

import com.ull.emergenciapp.Entities.Alert;
import com.ull.emergenciapp.Entities.Response;

import okhttp3.RequestBody;
import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Multipart;
import retrofit2.http.POST;
import retrofit2.http.Part;
import retrofit2.http.Path;

public interface AlertService {
    String group = "alert/";

    @Multipart
    @POST(group + "sendAlert")
    Call<Response<Alert>> sendAlert(@Part("id") RequestBody id, @Part("latitude") RequestBody latitude, @Part("longitude") RequestBody longitude);

    @Multipart
    @POST(group + "acceptAlert")
    Call<Response<Alert>> acceptAlert(@Part("id_alert") RequestBody id_alert, @Part("id_sanitary") RequestBody id_sanitary,
                                      @Part("latitude") RequestBody latitude, @Part("longitude") RequestBody longitude, @Part("time") RequestBody time);

    @GET(group + "check/{id}")
    Call<Response<Alert>> checkAlert(@Path("id") String id);

    @GET(group + "getEstimatedTime/{id}")
    Call<Response<String>> getEstimatedTime(@Path("id") String id);
}
```

Figura 18: AlertService.java

- **Entities**

  Contiene clases empleadas para poder manejar con mayor facilidad las respuestas del servidor. Haciendo uso de la librería *GSON Converter*, se realiza la conversión desde el formato JSON a un objeto del tipo pasado como argumento.

En este caso, existen tres clases, *Response*, que representa el objeto general de toda respuesta del servidor. Está compuesto por un *boolean* de donde se obtiene si el resultado de la llamada a la API ha sido satisfactorio, un *string* donde se almacena el mensaje obtenido tras la llamada, y finalmente un objeto de tipo parametrizado llamado *data*, que abarca los datos útiles, si es que existen, retornados. Es aquí donde entra en juego las otras dos clases *Alert* y *User*, ya que son empleadas adjuntándolas como parámetro a un objeto de tipo *Response,* para llamadas que retornan datos sobre alertas y usuarios respectivamente.

- **firebase**

  Observamos la clase MyFirebaseMessagingService, que como bien sugiere la documentación del propio *Firebase Cloud Messaging* extiende de *FirebaseMessagingService* y sobrescribe los siguientes métodos:

  - *onNewToken:* Que es llamado cada vez que la aplicación recibe un nuevo token, generalmente tras la instalación de la misma. Es por ello que se debe sobrescribir el mismo para dar el tratamiento deseado al token obtenido, en nuestro caso, lo almacenamos, para que una vez el usuario haya realizado un login correcto sea enviado al servidor para su almacenamiento.
  - *onMessageRecieved:* Que se llama cada vez que se recibe un mensaje desde el servicio. En nuestra implementación, existen dos casos a considerar, dependiendo de si la aplicación se encuentra en ejecución en primer plano o no y a su vez del tipo de mensaje recibido. Normalmente si la aplicación se encuentra en segundo plano, se trataría de solicitudes de asistencia, por lo que se muestra la notificación correspondiente al usuario, por otro lado si se encuentra en primer plano, puede tratarse de una respuesta a una solicitud, por lo que la aplicación debe realizar una transición a una determinada pantalla dependiendo del contenido del mensaje.

- **ui**

  Este paquete contiene fragmentos que son mutables dentro de una misma pantalla, como por ejemplo en el caso del registro, ya que está diseñado de tal forma que consta de tres pasos y para cada uno de ellos existe un Fragment que lo representa, por lo que el registro estaría compuesto por un layout donde mostrar dichos Fragments, botones para avanzar o retroceder entre los pasos y un indicador del paso en el que se encuentra actualmente el usuario.

  Por otro lado, se encuentra el Fragment *MapsFragment* que como indica la documentación de Google extiende de *SupportMapFragment* e implementa *OnMapReadyCallback* para poder visualizar mapas de la API de *Google Maps.*

En cuanto a las clases mencionadas con anterioridad pertenecientes a cada una de las pantallas anteriores, algunos aspectos de la implementación a destacar serían:

- En las pantallas donde existe una animación para indicar al usuario que la aplicación sigue funcionando, aunque no se realice ninguna acción aparentemente, como pueden ser las de

las Figuras 9 y 11. Lo que realmente sucede es que se crea un hilo paralelo de ejecución en el que cada 0,2 segundos se altera la transparencia de los puntos para simular la animación. El código es el siguiente:

```
private void startDots(){
    addDotsIndicator( position: 2);
    dotsThread = (Thread) run() → {
            while(true) {
                for (int i = 0; i < (mDots.length + 2); i++) {
                    try {
                        final int a = i;
                        runOnUiThread(new Runnable() {
                            @Override
                            public void run() { addDotsIndicator(a); }
                        });
                        sleep( millis: 200);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
    };

    dotsThread.start();
}
```

Figura 19: Función startDots()

```
private void addDotsIndicator(int position){
    mDots = new TextView[5];
    mDotsLayout.removeAllViews();

    for (int i = 0; i < mDots.length; i++) {
        mDots[i] = new TextView( context: this);
        mDots[i].setText(Html.fromHtml( source: "&#8226"));
        mDots[i].setTextSize(65);
        mDots[i].setTextColor(getResources().getColor(R.color.white));
        mDots[i].setAlpha(0);

        mDotsLayout.addView(mDots[i]);
    }
    if(mDots.length > 0){
        setTransparence(position,  transparence: 1);
        setTransparence( position: position - 1, (float)0.5);
        setTransparence( position: position - 2, (float)0.25);
    }
}

private void setTransparence(int position, float transparence){
    if(position < mDots.length && position >= 0){
        mDots[position].setAlpha(transparence);
    }
}
```

Figura 20: Funciones addDotsIndicator(position) y setTransparence(position, transparence)

- A su vez nuevamente en la pantalla de la Figura 11, implementada en la clase *SolicitudRecibida.java*, existe otro hilo de ejecución en el que se comprueba con el servidor cada minuto el tiempo estimado en el que debería llegar la ayuda.

  Como se puede observar en la siguiente figura, se realiza una llamada síncrona a la API desde de la función *getEstimatedTime*, se actualiza el tiempo mostrado en la interfaz gráfica con la función *changeTime* y se mantiene inactivo el hilo durante un minuto, este proceso se realiza mientras el tiempo estimado sea mayor que 1 minuto, una vez detenido el hilo se muestra el mensaje “Menos de 1 min”, lo que indica que el usuario “sanitario” se encontraría muy cerca del usuario que ha solicitado la asistencia.

```
private void checkTime(){
    int sleepTime = 1000 * 60 * 1;
    timeThread = (Thread) run() → {
            try {
            double time;
            do{
                time = getEstimatedTime();
                changeTime(time);
                sleep(sleepTime);
            } while (time > 1.0);
            changeTime(time);
            timeThread.interrupt();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
    };
    timeThread.start();
}

private double getEstimatedTime() {
    AlertService alertService = APIEngine.getAPIEngine().create(AlertService.class);
    Call<Response<String>> call = alertService.getEstimatedTime(id_alert);
    try {
        return Double.valueOf(call.execute().body().getData());
    }catch (Exception e){
        Log.d( tag: "mylog", e.getMessage());
    }

    return 0.0;
}
```

Figura 21: Funciones checkTime() y getEstimatedTime()

```
private void changeTime(Double time) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            if(time > 60.0){
                int horas = (int) (time / 60.0);
                int minutos = (int) (time % 60.0);
                txt_Distancia.setText(horas + " h y " + minutos + " min");
            }else if(time > 1.0){
                txt_Distancia.setText(round(time) + " min");
            }else{
                txt_Distancia.setText("Menos de 1 min");
            }
        }
    });
}
```

Figura 22: Función changeTime(time)

- Otra parte destacable de la implementación se encuentra en la clase *Splash.java* o en *Login.java*. *Splash* es la primera pantalla que se muestra en la ejecución de la app y por ello, puede llegar a tener distintas formas de ejecución dependiendo de la situación actual. Por ejemplo, cuando un usuario "sanitario" recibe una solicitud de asistencia como notificación, cuando el usuario pulsa sobre la notificación, se ejecuta una llamada a *Splash*, y éste a su vez debe determinar si el usuario se encuentra logueado, para redirigirlo para poder aceptar o rechazar la solicitud, o si por el contrario no ha iniciado sesión, debe redirigir a la pantalla de login y una vez completado el mismo, ésta a su vez debe redirigir nuevamente para responder a la solicitud. Este comportamiento lo logramos sobrescribiendo la función *onNewIntent* que recibe como argumento el *Intent* con el que se ha llamado al *Activity*. Teniendo esto en cuenta es tan fácil como suministrar la acción a ejecutar en el *Intent* pasado al *Activity* y una vez en éste, actuar en consecuencia.

```
@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    Log.d( tag: "mylog",  msg: "New Intent");
    if(intent.hasExtra( name: "action")){
        switch (intent.getStringExtra( name: "action")){
            case "Alert":
                Log.d( tag: "mylog",  msg: "Se ha pulsado una notificación de ayuda");
                Intent aceptarIntent = new Intent( packageContext: Splash.this, AceptarSolicitud.class);
                aceptarIntent.putExtras(intent.getExtras());
                Intent failIntent1 = new Intent( packageContext: Splash.this, Login.class);
                failIntent1.setAction("Redirect");
                failIntent1.putExtra( name: "redirect",  value: "aceptarSolicitud");
                failIntent1.putExtras(intent.getExtras());
                checkSesion(aceptarIntent, failIntent1);
                break;
            case "Response":
                Log.d( tag: "mylog",  msg: "Se ha recibido una notificación de aceptación de alerta");
                Intent solicitudRecibidaIntent = new Intent( packageContext: Splash.this, SolicitudRecibida.class);
                solicitudRecibidaIntent.putExtras(intent.getExtras());

                Intent failIntent2 = new Intent( packageContext: Splash.this, Login.class);
                failIntent2.setAction("Redirect");
                failIntent2.putExtra( name: "redirect",  value: "SolicitudRecibida");
                failIntent2.putExtras(intent.getExtras());
                checkSesion(solicitudRecibidaIntent, failIntent2);
                break;
            default:
                defaultAction();
                break;
        }
    }else{
        Log.d( tag: "mylog",  msg: "MAIN Intent");
        defaultAction();
```

Figura 23: Función onNewIntent

- Finalmente se ha desarrollado el servicio *LocationService*, que se encarga de obtener la ubicación actual del usuario, tanto cuando la aplicación se ejecuta en primer plano como en segundo plano. En caso de que la aplicación se ejecute en segundo plano, debido a los requerimientos de la API 26 y superiores, correspondiente a la versión Oreo del sistema operativo Android. Es necesario notificar al usuario de que la aplicación continúa

ejecutando servicios por lo que la implementación propuesta, comprueba la versión del sistema operativo desde el que se está ejecutando la aplicación y genera la correspondiente notificación.

Para obtener la ubicación se emplea el tipo *FusedLocationProviderClient* proveído por Google que combina la obtención de ubicación a través del GPS y el suministrado por las antenas de radiofrecuencia o estaciones Wifi cercanas, dependiendo del estado y características del sistema. Facilitando así obtener la ubicación y dotando a la misma de una mayor precisión y rapidez de obtención.

Conclusiones y líneas futuras

Como conclusión podemos afirmar que los resultados han sido satisfactorios, ya que se ha logrado la implementación propuesta con un funcionamiento más que aceptable. Cumpliendo además con todos los objetivos y requerimientos formulados en el planteamiento del proyecto.

Por otro lado, cabría destacar que, como consideración personal, parece bastante más complicado implementar cualquier solución en la actualidad empleando como lenguaje de programación *Java*, en vez del recomendado por Google, *Kotlin*, ya que éste último simplifica en gran medida el desarrollo, además de que existe infinidad de documentación y tutoriales para el mismo, a diferencia de para *Java*, que lentamente va quedando obsoleto.

Como líneas futuras de trabajo, sería interesante adecuar la API para que se funcional independientemente de la plataforma desde la que se realizan las peticiones, lo cual simplemente sería gestionar correctamente el envío de notificaciones dependiendo de la plataforma a la que se envíen, sin embargo, como se ha empleado la plataforma de *Firebase Cloud Messaging* y ésta se encuentra disponible para todas las plataformas donde se podría emplear la app (Web, IOS, etc…), probablemente implementando las librerías de la plataforma para cada una de ellas, no sería necesario realizar ningún cambio en la API.

Una vez adecuada la API, si fuese necesario, el siguiente paso sería implementar una aplicación para IOS y su contraparte para navegadores web, cubriendo así la totalidad de dispositivos disponibles en el mercado. Pudiendo incluso ser operado directamente por los servicios oficiales de emergencias, añadiendo un plus a la aplicación y facilitando el trabajo de los servicios de emergencias ya que dispondrían de datos de los que carecen con el actual modelo de actuación que poseen.

Conclusions

In conclusion, we can affirm that the results have been satisfactory, since the proposed implementation has been achieved with a more than acceptable operation. Also complying with all the objectives and requirements formulated in the project approach.

On the other hand, it should be noted that, as a personal consideration, it seems much more complicated to implement any solution nowadays using Java as the programming language, instead of the one recommended by Google, Kotlin, since the latter greatly simplifies development, in addition that there is plenty of documentation and tutorials for it, unlike for Java, which is slowly becoming obsolete.

As future lines of work, it would be interesting to adapt the API so that it works independently of the platform from which the requests are made, which would simply be to correctly manage the sending of notifications depending on the platform to which they are sent, however, As the Firebase Cloud Messaging platform has been used and this it is available for all platforms where the app could be used (Web, IOS, etc ...), probably implementing the platform libraries for each ones, it would not be necessary to perform any change in API.

Once the API is adequate, if necessary, the next step would be to implement an application for IOS and its counterpart for web browsers, thus covering all devices available on the market. It could even be operated directly by the official emergency services, adding a plus to the application and facilitating the work of the emergency services since they would have data that they lack with the current model of action they have.

Bibliografía y recursos

https://developer.android.com/reference/android/app/Activity
https://developer.android.com/training/location/receive-location-updates
https://developer.android.com/reference/android/app/Service
https://developer.android.com/training/maps
https://developers.google.com/maps/documentation/android-sdk/map
https://firebase.google.com/docs/cloud-messaging/android/receive?hl=es
https://developer.android.com/guide/topics/location/strategies.html
https://openrouteservice.org/dev/#/api-docs/v2/matrix/{profile}/post
https://www.pabloblanco.es/sql-obtener-coordenadas-en-radio-de-accion/
https://square.github.io/retrofit/
http://www.slimframework.com/docs/v4/