



UNIVERSITAT POLITÈCNICA  
DE CATALUNYA  
BARCELONATECH



Facultat d'Informàtica de Barcelona  
Universitat Politècnica de Catalunya

## Chat application for Android with multiple languages

TRABAJO DE FIN DE MÁSTER

Máster en Ingeniería Informática - MEI

*Autor:* Le Danny Yang

*Tutor:* Pere Pau Vázquez Alcocer

Curso 2018-2019



# Índice

---

<b>Índice</b>	iii
<b>Listado de imágenes</b>	iv
1    Introducción . . . . .	2
1.1    Motivación . . . . .	2
1.2    Objetivos . . . . .	2
1.3    Estructura del trabajo . . . . .	2
2    Tecnologías . . . . .	4
2.1    Android Studio . . . . .	4
2.2    Kotlin vs Java . . . . .	8
2.3    Google Firebase . . . . .	10
2.4    Butter Knife . . . . .	11
2.5    Picasso vs Glide . . . . .	12
2.6    Room . . . . .	14
2.7    Retrofit . . . . .	15
2.8    Corutinas . . . . .	16
2.9    GitHub . . . . .	17
3    Kotlin . . . . .	20
3.1    Clases . . . . .	20
3.2    Herencia de clases . . . . .	20
3.3    Funciones en Kotlin . . . . .	20
3.4    Variables . . . . .	21
3.5    Extensión de funciones . . . . .	21
3.6    Null Safety y Elvis Operator . . . . .	22
3.7    Collections . . . . .	23
3.8    Singleton en Kotlin . . . . .	24
4    Arquitectura . . . . .	26
4.1    Model View ViewModel . . . . .	26
4.2    LiveData . . . . .	27
4.3    Ejemplo de MVVM en el proyecto . . . . .	27
5    Diseño de Interfaces y código . . . . .	30
5.1    Permisos . . . . .	33
5.2    Login . . . . .	34
5.3    Sign up . . . . .	37
5.4    Main Activity . . . . .	39
5.5    Chat Activity . . . . .	42
5.6    Image Display Activity . . . . .	51
5.7    Image Tool Activity . . . . .	53
5.8    Location Activity . . . . .	56
5.9    User Searcher Activity . . . . .	57
5.10    User Profile Activity . . . . .	58
5.11    Settings Activity . . . . .	60
6    ¿Cómo realizamos ciertas acciones? . . . . .	62
6.1    ¿Cómo traducimos los mensajes? . . . . .	62

6.2	¿Cómo enviamos los mensajes? . . . . .	62
6.3	¿Cómo se le notifica al resto de usuarios que hemos actualizado nuestro usuario (foto, nombre de usuario, estado) . . . . .	63
6.4	¿Cómo guardamos las imágenes en nuestra base de datos? . . . . .	63
7	Modelo de traducción local vs Cloud Translation . . . . .	65
8	Trabajo futuro . . . . .	67
9	Conclusiones . . . . .	68
	<b>Bibliography</b>	<b>69</b>

## Listado de imágenes

---

1	Captura de pantalla del Android Studio . . . . .	4
2	Desarrollo nativo de aplicaciones móviles . . . . .	5
3	Desarrollo híbrido de aplicaciones móviles . . . . .	5
4	Tabla de comparación entre desarrollo nativo e híbrido . . . . .	6
5	WhatsApp en Android e iOS . . . . .	7
6	Instagram en Android e iOS . . . . .	7
7	Opción de Java y Kotlin en la documentación de <i>Google Firebase</i> . . . . .	8
8	Encuesta de Stackoverflow sobre lenguajes más queridos en 2019 . . . . .	10
9	Captura de pantalla de Firebase Authentication . . . . .	11
10	Base de datos NoSQL de Firebase Realtime Database . . . . .	12
11	Tamaño de Glide y Picasso . . . . .	13
12	Métodos de Glide y Picasso . . . . .	14
13	Uso de memoria de Glide y Picasso . . . . .	14
14	Componentes de Room . . . . .	16
15	Diferencias entre los tres tipos de Dispatchers de las corutinas . . . . .	18
16	Captura de pantalla de Github . . . . .	19
17	Esquema de la arquitectura MVVM . . . . .	26
18	Interfaz del Chat Activity cuando el usuario quiere seleccionar un GIF . . . . .	27
19	Fragmento de código relacionado con la View . . . . .	28
20	Fragmento de código relacionado con el ViewModel . . . . .	28
21	Fragmento de código relacionado con el Model . . . . .	29
22	Diseñando la IU en Android Studio . . . . .	31
23	Diseñando la IU mediante código XML . . . . .	32
24	Petición de permisos en Android . . . . .	33
25	Pantalla de login . . . . .	35
26	Diseñando la interfaz de la Actividad mediante código XML . . . . .	37
27	Actividad principal - MainActivity . . . . .	39
28	ViewHolder de una conversación . . . . .	39
29	Esquema del funcionamiento de un RecyclerView . . . . .	42
30	Diálogos en MainActivity . . . . .	44
31	Captura de pantalla de ChatActivity.kt . . . . .	45
32	ChatActivity mostrando los emoticonos . . . . .	47
33	Captura de pantalla de ChatActivity.kt . . . . .	51
34	A la izquierda tenemos una imagen y a la derecha un GIF . . . . .	52
35	Pantalla principal donde se encuentran todas las conversaciones del usuario . . . . .	54
36	Pantalla principal donde se encuentran todas las conversaciones del usuario . . . . .	56
37	Diseñando la interfaz de la Actividad mediante código XML . . . . .	57

38	Actualización del nombre de usuario . . . . .	58
39	Diseñando la interfaz de la Actividad mediante código XML . . . . .	60
40	Listado de lenguajes permitidos . . . . .	61
41	Ejemplo de código en Python para codificar y decodificar en base64 . . . . .	64
42	Captura de pantalla de la tabla de precios de Google Cloud Platform . . . . .	65
43	Texto que el usuario envía para ser traducido . . . . .	66
44	Traducción realizada a partir del texto de la figura 43 . . . . .	66

## 1 Introducción

---

Hoy por hoy, una de las principales razones por las que usamos los *smartphones* son las aplicaciones de mensajería, dentro de estas aplicaciones se nos viene a la mente *Whatsapp*, *Telegram* y *WeChat* sobre todo en el continente asiático. Se dice que WhatsApp cuenta con alrededor de 2000 millones de usuarios [1].

Generalmente, estas aplicaciones comparten cierta funcionalidad básica como es el envío de mensajes planos, imágenes, vídeos, GIFs y la localización, pero normalmente olvidan una bastante interesante, que es la que trataremos en este proyecto, la traducción automática de mensajes en tiempo real.

Esta funcionalidad no sería de suma importancia si toda la población mundial hablásemos el mismo idioma, la realidad es que vivimos cada vez en un mundo más globalizado así pues, este proyecto intenta facilitar la vida a aquellos usuarios que son incapaces o no han tenido la suerte de aprender otro idioma sea por el motivo que sea.

### 1.1. Motivación

La principal motivación de la realización de este proyecto es como hemos mencionado anteriormente, en poder permitir a aquellos usuarios más desfavorecidos en el apartado lingüístico, debido a varios factores, a poder comunicarse con cualquier persona en su propio idioma haciéndole transparente esta funcionalidad y también el hecho y el reto de demostrar la posibilidad de implementar dicha función en los dispositivos móviles que poseen una potencia limitada debido a su reducido tamaño.

### 1.2. Objetivos

Para este proyecto, nos hemos propuesto realizar una aplicación de Android desde cero, donde el usuario podrá ser capaz de comunicar con cualquier usuario en uno de los idiomas permitidos (castellano, catalán, francés, inglés, alemán...), y hacerlo de forma transparente como si fuese una funcionalidad más, al igual que si enviásemos una imagen, por poner un ejemplo.

Por supuesto, esta aplicación contará con las funcionalidades básicas de cualquier «App» de mensajería que más adelante iremos desglosando y explicando con sumo detalle.

### 1.3. Estructura del trabajo

Para concluir la introducción, hemos decidido estructurar el documento en secciones o capítulos de manera lógica, empezando con la introducción seguida de las tecnologías usadas y el porqué de las decisiones.

Posteriormente, explicaremos la arquitectura del proyecto que hemos seguido para organizar toda la lógica de la aplicación.

Después, explicaremos cómo hemos diseñado las interfaces de usuario de manera que mejore la experiencia intentando siempre que dicha aplicación sea lo más intuitiva posible y la lógica interna de cada módulo de nuestro proyecto, qué realiza cada fichero del proyecto y su impacto al realizarla de esta manera. También incluiremos fragmentos de código a lo largo de la explicación, para que el lector, en este caso usted, pueda seguir la redacción más fácilmente.

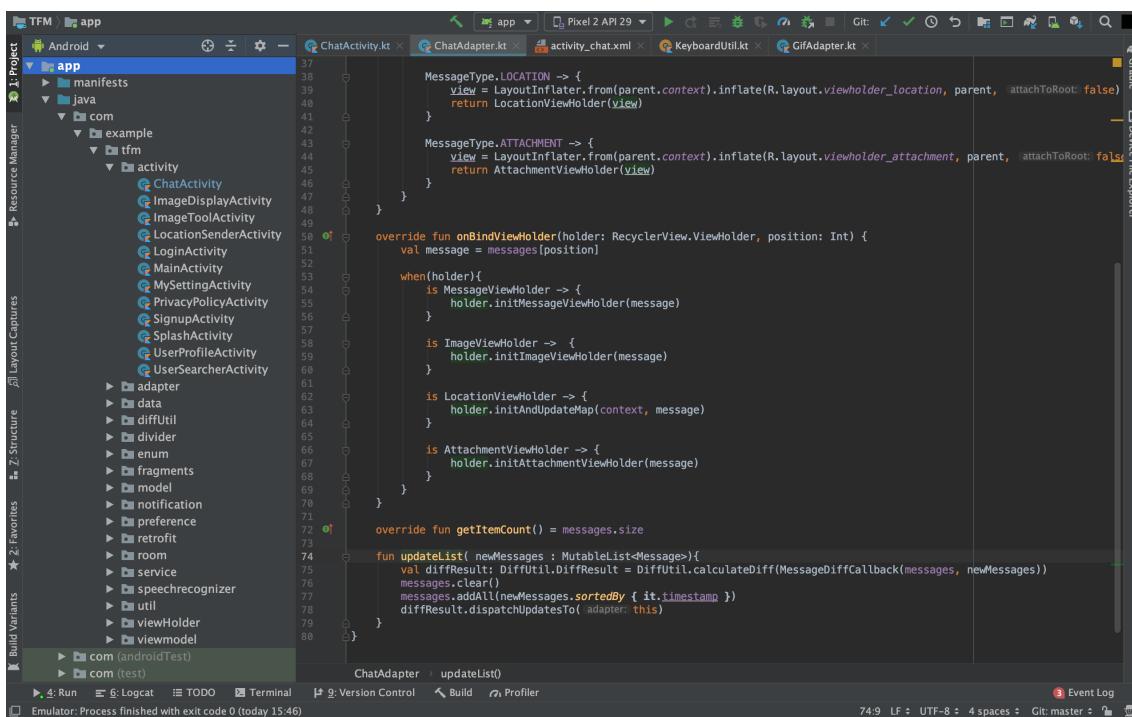
Finalmente, acabaremos contrastando el hecho de haber escogido las librerías gratuitas por encima de las de pago con el consiguiente resultado que se ha obtenido y un resumen final.

## 2 Tecnologías

Primeramente, para empezar a desarrollar una aplicación de Android nativo, existe un programa que se denomina *Android Studio* proporcionado por *Google*. Por otra parte, existen dos lenguajes soportados por el propio *Google* que son: Java y Kotlin. Más adelante explicaremos las diferencias entre ambos lenguajes pero os adelantamos que para este proyecto se ha escogido finalmente Kotlin para el desarrollo, una de las razones del porqué es que en el pasado I/O 2019 Developer Conference, *Google* anunció Kotlin como el lenguaje preferido para los desarrolladores y una prueba de ello es que todas las posteriores librerías de Jetpack API se publicarán primeramente en Kotlin. Es esto, sumado a las ganas de aprender un nuevo lenguajes los responsables de haber escogido dicho lenguaje de programación <sup>1</sup>.

### 2.1. Android Studio

Es el Integrated Development Environment o también llamado IDE preferido para el desarrollo nativo de aplicaciones en Android, este se lanzó el 16 de Mayo de 2013 y desde entonces, reemplazó a Eclipse. Desde el momento en el que se escribe este documento, *Android Studio* está en su versión 3.5.



**Figura 1:** Captura de pantalla del *Android Studio*. Se puede observar que en la parte izquierda se encuentra la estructura de ficheros. En la parte central y derecha se muestra el contenido de los ficheros y donde el usuario programará.

Tal como se puede intuir, este proyecto se ha desarrollado de manera nativa y eso quiere decir, que esta aplicación solo podrá ser usada en dispositivos Android por lo que aquellos usuarios que usen iOS serán incapaces de usarla.

<sup>1</sup>Google I/O es una conferencia anual de desarrolladores realizada por Google en Mountain View, California. I/O se inauguró en 2008 y está organizado por el equipo ejecutivo. "I/O" es sinónimo de entrada/salida, así como el lema "Innovación en la pluma O". El formato del evento es similar al de Google Developer Day.

2



**Figura 2:** Desarrollo nativo de aplicaciones móviles. Para desarrollo de aplicaciones iOS tenemos dos lenguajes Objective-C y Swift. Para Android los más usados son Java y Kotlin.

Existen otras formas de desarrollar aplicaciones móviles que se denominan híbridas, donde los desarrolladores usan HTML5, JavaScript y CSS en combinación con elementos nativos. En este caso, se usa un componente llamado WebView de Android que sirve para reproducir contenido Web e incrustar la aplicación que hayamos creado usando tecnologías como Flutter o React para ser usado en nuestros dispositivos móviles.

Una de las ventajas en desarrollar en nativo es el rendimiento que se obtiene ya que se estará implementando funcionalidades que el propio Android te ofrece mientras que en el híbrido se añade una capa extra entre la plataforma y el código fuente, donde el rendimiento se resiente. Por otra parte, en el aspecto de UX (User Experience) también sale vencedor el desarrollo nativo por todo el sinfín de animaciones que trae de serie.



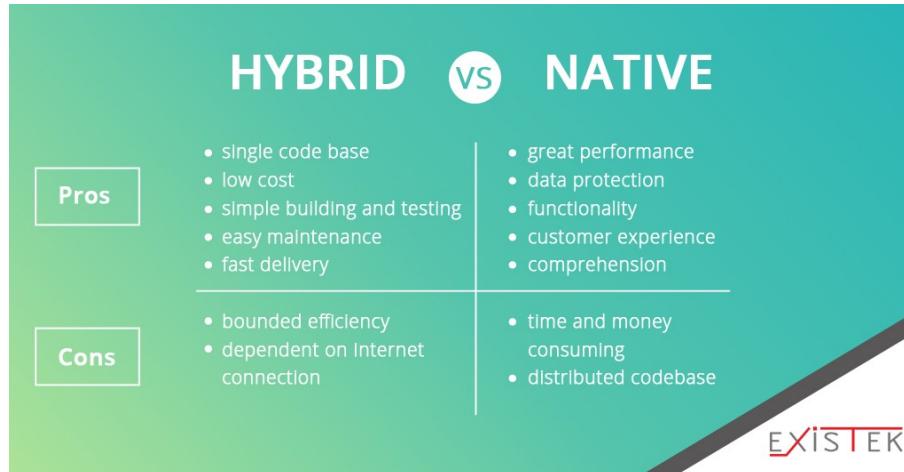
**Figura 3:** Desarrollo híbrido de aplicaciones móviles. Para desarrollo de aplicaciones híbridas, los más usados son React Native, Xamarin e Ionic. Cabe destacar que un nuevo lenguaje está cogiendo fuerza en cuanto a desarrollo Native se refiere denominado Flutter.

Entonces, ¿cuando es mejor el desarrollo híbrido? Si nos fijamos en la Figura 4, se puede observar las diferencias entre ambas formas de crear aplicaciones móviles. Existen proyectos donde se intenta obtener el máximo número de usuarios posibles, por lo que al ser desarrollado en nativo, simplemente con desarrollarla una vez, podrá ser utilizado en

<sup>2</sup>Las imágenes 2, 3, 4 se han extraído de [2]

Android e iOS (realizando pequeños cambios en el código), por lo que las empresas ven con bastantes buenos ojos el hecho de ahorrar tiempo y dinero. Opción bastante común en pequeñas Start-ups.

Otro aspecto no menos importante es el hecho del mantenimiento de la aplicación, cuantas menos líneas de código se escriban, más fácil será de mantener la aplicación. Así pues, el desarrollo híbrido también cuenta con esta ventaja.



**Figura 4:** Tabla de comparación entre desarrollo nativo e híbrido

A modo de resumen, listaremos las ventajas y desventajas de usar nativo e híbrido

#### **Desarrollo híbrido - Ventajas:**

1. Un único código base
2. Un único mantenimiento
3. Coste de producción menor
4. Tiempos de entrega más cortos

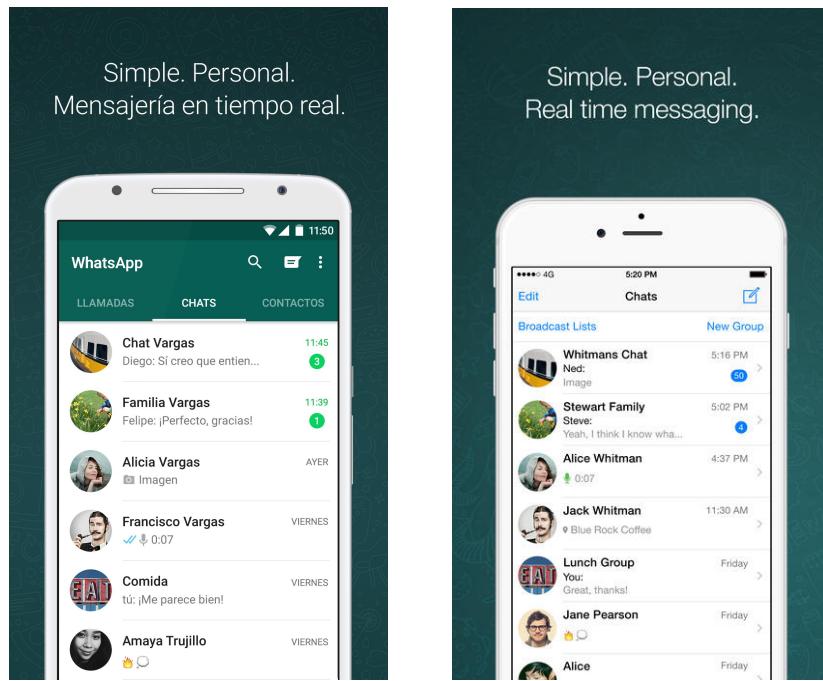
#### **Desarrollo nativo - Ventajas:**

1. Rendimiento elevado en comparación con híbrido
2. Mayor abanico de posibilidades al usar componentes nativos
3. Mayor control en el desarrollo de la aplicación (ciclo de vida de los componentes)

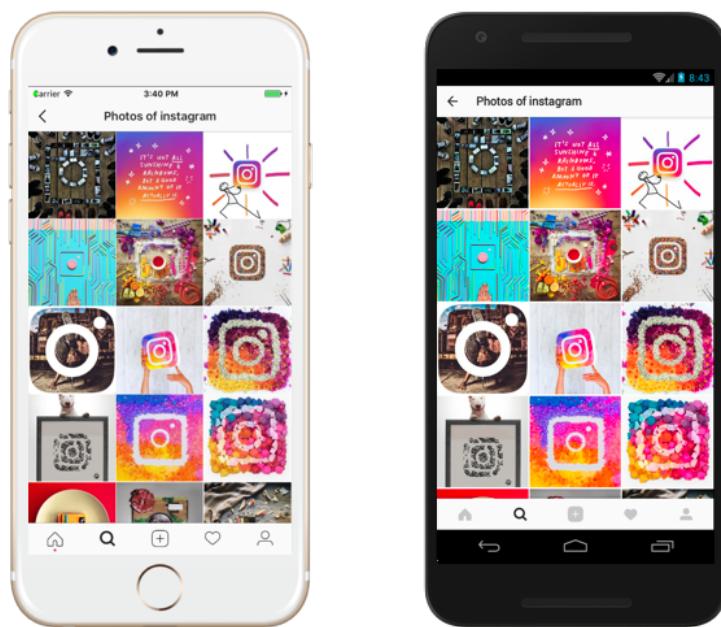
Como todo en esta vida, cualquiera de las dos opciones tienen sus ventajas e inconvenientes y la decisión se tomará según los requisitos y recursos del que se dispone.

En adelante se mostrará un par de ejemplos de aplicaciones desarrollada de manera nativa y de manera híbrida. En las Figura 5 se puede observar la aplicación de WhatsApp en sus dos versiones. Se puede diferenciar los distintos componentes y el estilo que se le ha aplicado a cada aplicación.

Por otra parte, si observamos en la Figura 6, ambas aplicaciones tienen el mismo diseño y la funcionalidad es idéntica independientemente del sistema operativo que usemos en nuestro dispositivo. Es por ello, que la decisión de elegir el tipo de desarrollo dependerá única y exclusivamente de los recursos que tengamos.



**Figura 5:** A su izquierda se encuentra la aplicación de WhatsApp desarrollada en Android, y a la derecha en iOS



**Figura 6:** Aplicación de Instagram en cada una de sus versiones. A la izquierda se encuentra en su versión iOS, y a la derecha en su versión Android

## 2.2. Kotlin vs Java

Este lenguaje fue desarrollado por JetBrains, la misma compañía que también creó Android Studio e IntelliJ en sus oficinas de San Petersburgo, Rusia. Una curiosidad, es que el nombre de Kotlin proviene de la isla de Kotlin localizado cerca de las oficinas donde surgió.

Lo que hace interesante este lenguaje es que corre sobre la máquina virtual de java o también denominado JVM (Java Virtual Machine), por lo que es totalmente compatible con Java y sus librerías. Citando al líder de desarrollo Andrey Breslav, Kotlin se concebió como un lenguaje de programación orientado a objetos de calidad industrial, cuyo propósito es mejorar el propio Java y seguir siendo interoperable con Java por lo que se permite una migración gradual de proyectos implementados en su totalidad en Java, añadir código Kotlin en él.

Una de las razones del auge de Kotlin en los últimos años es que en el *Google I/O* de 2018, se anunció como lenguaje oficial para el desarrollo en Android. Es por ello, que en las documentaciones de Android hoy en día también esté en Java y Kotlin (Véase en la Figura 7), facilitando mucho la labor de buscar información al desarrollador.

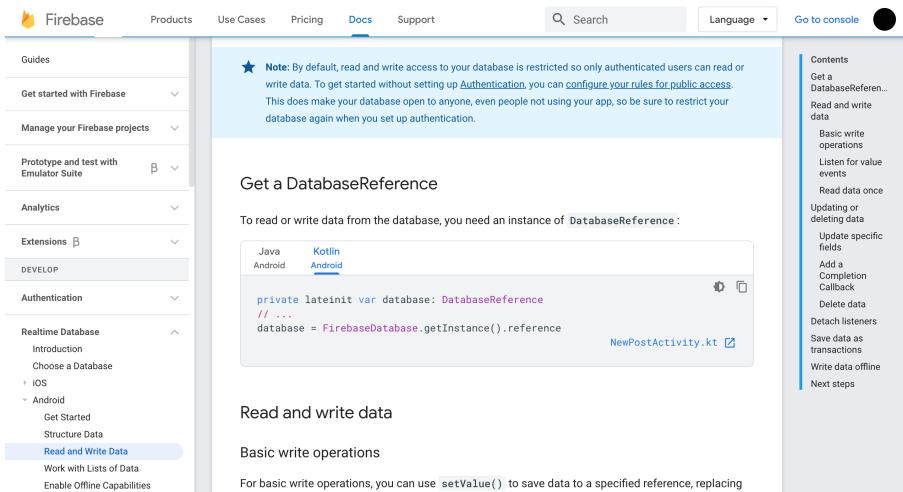


Figura 7: Opción de Java y Kotlin en la documentación de *Google Firebase*

### Ventajas de usar Kotlin:

1. Soporta varios conceptos de la programación moderna como: extensión de funciones, funciones de orden superior, delegación
2. Null-Safety: Kotlin se creó con esta idea en mente, se trata de especificar en cada creación de variable, si dicha variable puede obtener el valor nulo o no, esto lo que provoca es reducir un quebradero de cabeza más para el programador ya que evita el conocido *NullPointerException*. Tony Hoare, «computer scientist» británico, también conocido por la invención del *QuickSort* (algoritmo de ordenación) y el «null reference», denominó este último como el «Billion-Dollar mistake» debido a que por la culpa de este, causó innumerables errores en el código provocando a su vez, vulnerabilidades en el sistema.
3. Kotlin es más conciso y expresivo que Java, dando lugar a menos errores. Este hecho se puede observar en 1 y 2.
4. Adaptar Kotlin al proyecto en curso no implica ningún coste extra. Es normal encontrarse proyectos con código Java y Kotlin funcionando conjuntamente, sobre todo en aquellos proyectos que están en fase de migración.

```
1 data class Person(var name: String, var age: Int)
```

Listing 1: Clase POJO implementada en Kotlin

```
1 public class Person {  
2     private String name;  
3     private int age;  
4  
5     public Person(String name, int age) {  
6         this.name = name;  
7         this.age = age;  
8     }  
9  
10    public String getName() {  
11        return name;  
12    }  
13  
14    public void setName(String name) {  
15        this.name = name;  
16    }  
17  
18    public int getAge() {  
19        return age;  
20    }  
21  
22    public void setAge(int age) {  
23        this.age = age;  
24    }  
25}
```

Listing 2: Clase POJO implementada en Java

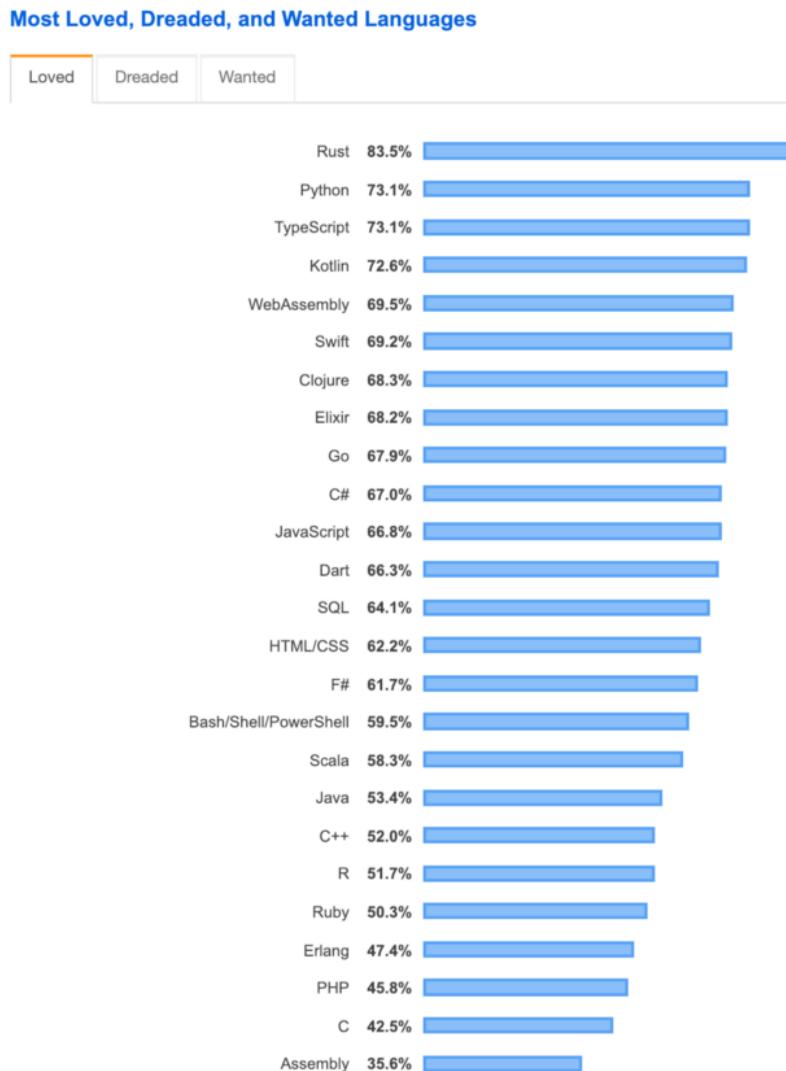
### Desventajas del uso de Kotlin:

1. En ciertas compañías, puede provocar un aumento en el gasto al formar a los trabajadores en el uso de Kotlin, esto también se puede ver como un reto para los propios programadores.
2. Hoy en día, la comunidad de Kotlin es notablemente más pequeña que la de Java ya que cuenta con muchísimos años de experiencia y existe información de todo tipo para este lenguaje.

Por otra parte, analizando los aspectos positivos y negativos de Java, nos podemos encontrar:

1. Fácil de aprender y entender
2. Android fue escrito en Java y a día de hoy, sigue siendo uno de los tres lenguajes de programación más usado según Github
3. Suele necesitar más líneas de código para realizar la misma función que en otros lenguajes de programación, necesitando «boilerplate code».

Para concluir esta sección, ¿cuáles son los beneficios principales de usar Kotlin en vez de Java? Una de las razones es que Kotlin se concibió para subsanar problemas de Java como el *NullPointerException*, mencionado anteriormente, otro aspecto a tener en cuenta,



**Figura 8:** Encuesta de Stackoverflow sobre lenguajes más queridos en 2019

es la tendencia en auge que está teniendo dentro de la comunidad de Android y que seguramente irá en aumento en los próximos años, según una encuesta realizada por la famosa página <https://www.stackoverflow.com>, Kotlin se sitúa en el cuarto puesto de los lenguajes de programación más queridos (Véase la Figura 8).

### 2.3. Google Firebase

Firebase es una plataforma móvil de *Google* que te ayuda a desarrollar aplicaciones de alta calidad. Una vez dicho esto, dentro de Firebase nos podemos encontrar un sinfín de herramientas para llevar a cabo, a la hora de implementar cualquier aplicación. En este proyecto se ha hecho uso de:

1. **Authentication:** autenticación de usuarios de manera simple y segura
2. **ML Kit:** se ha hecho uso de la traducción automática de idiomas

3. **Realtime Database**: almacenamiento y sincronización de datos de manera asíncrona para la aplicación. Los archivos se guardan en formato JSON. Es una base de datos NoSQL.

4. **Cloud Firestore**: almacenamiento y sincronización de datos de manera síncrona para la aplicación

Identificador	Proveedores	Fecha de creación	Inicio de sesión	UID de usuario
ttyang@gmail.com	✉	21 dic. 2...	21 dic. 2...	1HPJoEln4fSikKQA7hDt...
tinapan@gmail.com	✉	11 dic. 2...	3 ene. 2...	HScgp3YsnlPVyQrKujG...
danny27995@gmail.com	✉	10 dic. 2...	21 dic. 2...	d3MxWVNvq7UavWu9a...
weilesley@gmail.com	✉	10 dic. 2...	21 dic. 2...	o0xpoMqvqVzR9m1DtF...
haopen@gmail.com	✉	20 dic. 2...	21 dic. 2...	tdGJjFKFsDURCGy4SIVE...

Figura 9: Captura de pantalla de Firebase Authentication

En la Figura 9 se puede observar una captura de pantalla de Firebase Authentication, en ella se muestra al programador la lista de correos que están registrados en la aplicación, por otra parte, se puede especificar qué dominios de correo son válidos. En nuestro caso, solo aceptamos correos que tengan el dominio «@gmail.com» por sencillez. Más adelante, en la sección 5 se explicará con más detalle el cómo se ha implementado esta funcionalidad en la aplicación.

Por otra parte, en la Figura 10, se muestra la base de datos NoSQL con la lista de conversaciones que existen en la aplicación de mensajería y el cómo está estructurado. Esto también se explicará más adelante de manera detallada todos los campos de la conversación. La sección de ahora es una mera y sencilla presentación de Firebase.

## 2.4. Butter Knife

Primeramente, toda la información acerca de esta herramienta se ha extraído de [4]. Butter Knife es una librería que te ayuda y automatiza todo el proceso de inyección de las vistas y recursos de una forma elegante y limpia a tu proyecto.

Antiguamente, para referenciar un componente de alguna vista en Android, se hacía uso de del método `findViewById(R.id.nombre_widget)`, hoy en día existe varias opciones para evitar toda esta cantidad de «boilerplate code» como Dagger 2 y Butter Knife.

Por supuesto, habrá que instalar las dependencias pertinentes y para ello Android Studio usa Gradle como principal herramienta para instalarlas. Para más información consulte <https://github.com/JakeWharton/butterknife>.



**Figura 10:** Captura de pantalla de la base de datos NoSQL de Firebase Realtime Database

## 2.5. Picasso vs Glide

En este apartado hablaremos de dos librerías de tratamiento de imágenes que posee Android. Ambas librerías son casi idénticas que permite al programador realizar las principales tareas de tratamiento de imágenes como son:

1. Se encargan de «reciclar» las imágenes que formen parte de una lista y evitar la creación y destrucción continuada e innecesaria de las imágenes
2. Transformaciones de imágenes complejas con el mínimo uso de memoria (zoom-in, zoom-out, rotación...)
3. Realizar «caching» de las imágenes

La sintaxis entre estas dos librerías son prácticamente idénticas, por lo que a priori, si estás familiarizado con una de estas dos librerías, el uso de la otra es inmediato.

```

1 Picasso .with(myFragment)
2     .load(url)
3     .centerCrop()
4     .placeholder(R.drawable.loading_spinner)
5     .into(myImageView);

```

**Listing 4:** Sintaxis de la librería Picasso al cargar una imagen

```

1 class ExampleActivity extends Activity {
2     @BindView(R.id.user) EditText username;
3     @BindView(R.id.pass) EditText password;
4
5     @BindString(R.string.login_error) String loginErrorMessage;
6
7     @OnClick(R.id.submit) void submit() {
8         // TODO call server ...
9     }
10
11    @Override public void onCreate(Bundle savedInstanceState) {
12        super.onCreate(savedInstanceState);
13        setContentView(R.layout.simple_activity);
14        ButterKnife.bind(this);
15        // TODO Use fields ...
16    }
17}

```

**Listing 3:** Ejemplo de utilización de Butter Knife

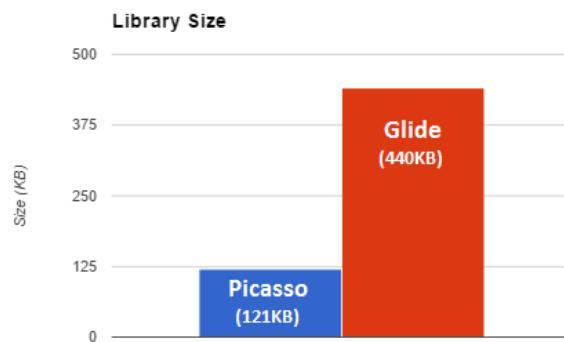
```

1 Glide.with(myFragment)
2     .load(url)
3     .centerCrop()
4     .placeholder(R.drawable.loading_spinner)
5     .crossFade()
6     .into(myImageView);

```

**Listing 5:** Sintaxis de la librería Glide al cargar una imagen

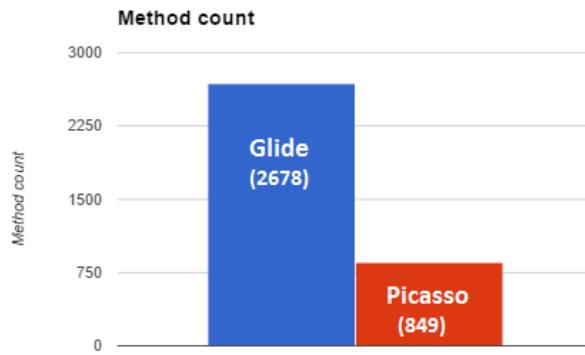
Cuando se comparan los tamaños entre estas dos librerías, nos encontramos que el tamaño de Glide es 3.5 veces mayor que el de Picasso (Véase la Figura 11)



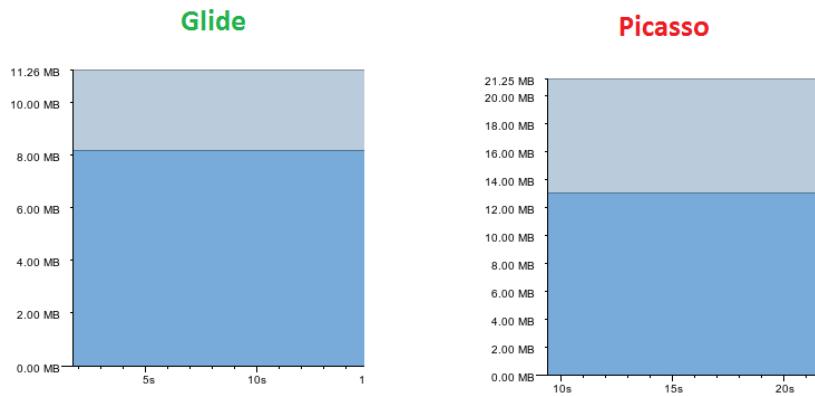
**Figura 11:** Tamaño de Glide y Picasso

Tiene sentido que a mayor tamaño de librería, mayor será el número de métodos del que se dispone. Es por ello que la gráfica de la Figura 12, observamos que Glide tiene un total de 2678 métodos frente a los 849 de Picasso.

En cuanto al uso de memoria, vemos en la Figura 13 que en un primer instante, Glide consume 8MB de memoria frente a los 13MB de Picasso para una misma imagen. Este hecho es así debido a que Picasso carga la imagen en la máxima calidad posible para posteriormente delegar la tarea de redimensionar la imagen a la tarjeta gráfica.



**Figura 12:** Métodos de Glide y Picasso



**Figura 13:** Uso de memoria de Glide y Picasso

Ambas librerías poseen la opción de cachear las imágenes en memoria. Para el caso de Picasso, una vez se descargan las imágenes en la calidad existente, se guarda una copia en memoria y cada vez que se requiera el uso de dicha imagen, se obtiene la imagen y se redimensiona. Ahora bien, con Glide, al descargar la imagen se guarda la copia redimensionada que será cargada a la *ImageView*, es por ello, que si en nuestra aplicación tuviésemos dos copias de una misma imagen de diferentes dimensiones, con Glide se guardarían dos copias distintas mientras que en Picasso solo existiría una única copia.

Una vez mostradas las diferencias y similitudes entre ambas librerías, en este proyecto se ha hecho uso de Glide, debido a que Glide provee al programador el uso de GIFs (Graphic Interchange Format) y para una aplicación de mensajería esta funcionalidad es vital para mejorar la experiencia de usuario.

La conclusión es que ninguna de las dos librerías son perfectas con sus pros y contras, y depende de cada proyecto, si se requiere de una aplicación con el menor tamaño posible, se recomienda el uso de Picasso, pero si se requiere el uso de GIFs, este es un aspecto diferenciador y el cual, hemos escogido Glide por encima de Picasso.

## 2.6. Room

Room es una librería que forma parte de Android Jetpack. Para aquellas personas que no estén familiarizadas, Jetpack es una colección de componentes de Android que facilitan

la vida al desarrollador a la hora de implementar las aplicaciones. Un breve repaso de qué son los componentes de Android Jetpack, se pueden dividir en:

1. **Foundation**: ofrecen funcionalidades interrelacionadas, test, y compatibilidad con Kotlin
2. **Arquitectura**: ayudan a diseñar aplicaciones sólidas
3. **Comportamiento**: ayudan a la aplicación a integrarse con los servicios de Android, como notificaciones, permisos, uso compartido y asistente de voz
4. **IU**: ayuda al desarrollador a implementar aplicaciones que sean fáciles de usar, intuitivas así como atractivas

Estos componentes de Jetpack siguen las buenas prácticas de programación, reducen el código «boilerplate» y simplifica tareas complejas para que el programador se centre únicamente en lo esencial.

Una vez repasado lo que es Android Jetpack, Room provee al desarrollador una capa extra de abstracción sobre SQLite, es decir, simplifica y abstracta las tareas complejas de SQLite pero sin perder rendimiento. Actualmente, Room se considera la mejor elección en cuanto a manejo de base de datos en Android y una de las ventajas de la abstracción es que se elimina notablemente el código repetitivo.

Existen tres razones principales por las que se usa Room en las aplicaciones y son:

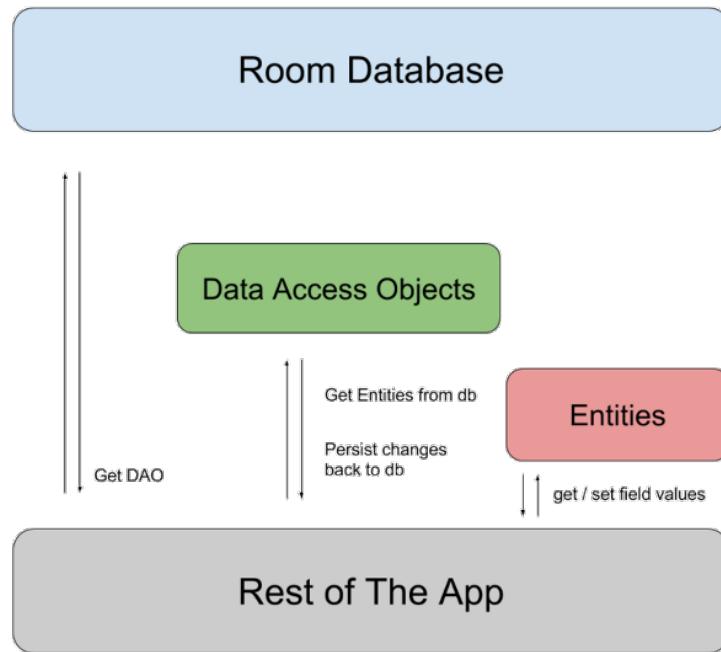
1. Todas las anotaciones **@Query** y **@Entity** se verifican en tiempo de compilación, por lo que evita errores en el código y no solo comprueban la sintaxis del código sino si se han eliminado o faltan tablas en la base de datos.
2. Como se ha mencionado anteriormente, evitar código repetitivo (*boilerplate code*)
3. Fácil integración con otros componentes de arquitectura en Jetpack como es LiveData, que hablaremos en posteriores secciones.

Como se puede observar en la Figura 14, Room tiene tres componentes principales:

1. **Entity**: la entidad se refiere a la tabla que se va a crear en la base de datos, se podría considerar una clase POJO con los atributos básicos
2. **DAO**: también conocido como Data Access Object, y tal como indican las siglas, son los responsables de definir los métodos para acceder a la base de datos, es decir, las queries.
3. **Database**: contiene el conjunto de Dao y en Java o Kotlin, debe extender de RoomDatabase.

## 2.7. Retrofit

En esta sección hablaremos de Retrofit, se trata de una librería de Android que permite abstractar al desarrollador de las llamadas **REST** y obtener el resultado en formato **JSON** (Véase el ejemplo en el fragmento de código 6), lo cual facilita la tarea que en el pasado podían ser un quebradero de cabeza para el programador.



**Figura 14:** Componentes de Room

```

1  {
2      "name": "John",
3      "age": 30,
4      "cars": [ "Ford", "BMW", "Fiat" ]
5  }
  
```

**Listing 6:** Ejemplo de un fichero JSON

Para instalar Retrofit, simplemente se ha de añadir las dependencias necesarias en **build.gradle**

```

1 implementation 'com.squareup.retrofit2:retrofit:(latest version)'
2 implementation 'com.google.code.gson:gson:(latest version)'
3 implementation 'com.squareup.retrofit2:converter-gson:(latest version)'
  
```

**Listing 7:** Dependencias de Retrofit en build.gradle

Para usar Retrofit, se debe crear clases POJO que representen las entidades del archivo JSON, esto se puede obtener fácilmente en multitud de páginas webs que se encuentran en Internet como <http://www.jsonschema2pojo.org/>, donde simplemente se añade el fichero JSON que se transforma en una clase Java o Kotlin para ser usado con Retrofit.

## 2.8. Corutinas

En este apartado de tecnologías utilizadas hablaremos de las corutinas. Las corutinas introducen una nueva forma de concurrencia que puede ser usado en Android para simplificar el código asíncrono, a pesar de ser nuevo en Kotlin en su versión 1.3, este concepto de corutinas ha estado siempre presente en los lenguajes de programación. En 1967, existió un lenguaje de programación denominado Simula que introdujo este término. Actualmente, son muchos los lenguajes que incluyen este concepto, tales como Javascript, Python, Go, entre otros muchos.

En Android, nos podemos encontrar las corutinas para:

1. **Tareas de larga duración:** como puede ser las consultas a bases de datos, se debe evitar bloquear el hilo principal por lo que las corutinas son una excelente solución.
2. **Main-Safety:** que en castellano se podría traducir como seguro para ser ejecutado en el hilo principal. Esto quiere decir que cualquier método con la etiqueta *suspend* puede ser llamado desde el hilo principal sin bloquearlo con total seguridad.

¿Qué denominamos tareas de larga duración? Puede ser confuso cómo de rápido puede llegar a ser un procesador moderno comparado con las peticiones en la red, en [7] pone de ejemplo que un dispositivo *Google Pixel 2*, un ciclo del procesador tarda 0.0000000004 segundos, un número ínfimo para los seres humanos. Por otra parte, una petición en la red tarda de media 0.4 segundos, que en este contexto ya se podría considerar una tarea de larga duración.

En Android, el hilo principal se encarga de la generación de las vistas y coordinar las acciones del usuario. Por lo que si bloqueamos el hilo principal, seguramente provoque una sensación de torpeza en las acciones y empeore la experiencia de usuario. Es por ello que se recomienda que las tareas de larga duración como puede ser acceder a la base de datos, ordenar una lista considerable de instancias deban ser realizadas con corutinas.

Para implementar el Main-Safety mencionado anteriormente, Kotlin provee al desarrollador los llamados dispatchers, también se le conoce más vagamente como contextos de corutinas. Existen tres dispatchers (Véase la Figura 15) y cuya función depende de la acción que queramos implementar:

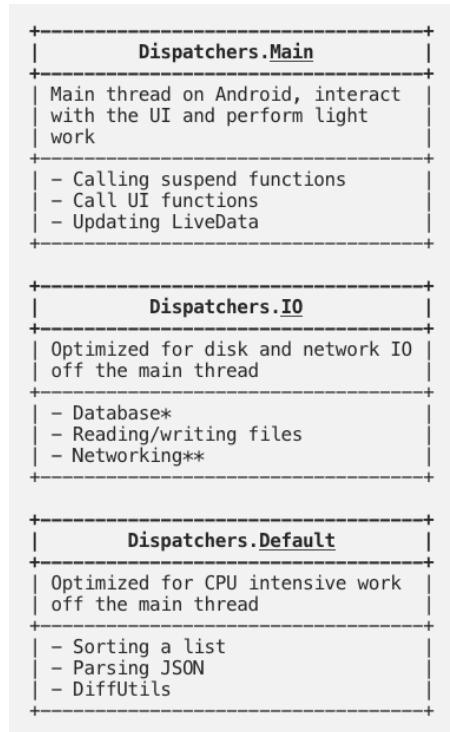
1. **Main:** se encarga de llamar a las suspend functions, actualización de las vistas y las variables
2. **IO:** acceso a base de datos, lectura y escritura de ficheros y peticiones a la red
3. **Default:** ordenar listas, parsear archivos JSON y ejecutar DiffUtils que básicamente consiste en determinar las diferencias entre dos colecciones y actualizar una lista.

Hemos comentado que las corutinas sirven para realizar peticiones a la red y acceso a la base de datos fuera del hilo principal. También hemos mencionado que Room y Retrofit se encargaban de estas acciones, respectivamente, por lo que una ventaja de Android Jetpack es que son compatibles ambas librerías. Así pues, Room y Retrofit implementan las corutinas de manera interna, mejorando el rendimiento de nuestro proyecto en este aspecto.

Como se observa en el fragmento de código 8, el método `fetchDocs()` se ejecuta en el hilo principal hasta que realiza una llamada `get()` por lo que cambia de contexto a IO con el fin de realizar una tarea de larga duración, que una vez finalizada vuelve al contexto inicial (hilo principal) para actualizar los datos recogidos del método `get()`.

## 2.9. GitHub

Por último y no menos importante, otro aspecto fundamental en el desarrollo de un proyecto informático, ya sea móvil o web, es necesario tener un repositorio donde salvaguardar los cambios que realizamos en el proyecto y mantener un control de versiones. En nuestro caso hemos hecho uso de *Github*. Existen otras herramientas como *GitLab* o *BitBucket*.



**Figura 15:** Diferencias entre los tres tipos de Dispatchers de las corutinas

En la Figura 16 tenemos la pantalla de *GitHub*, en ella se puede visualizar los proyectos que dicho usuario participa y en la parte inferior están las contribuciones que el usuario ha realizado a lo largo del tiempo.

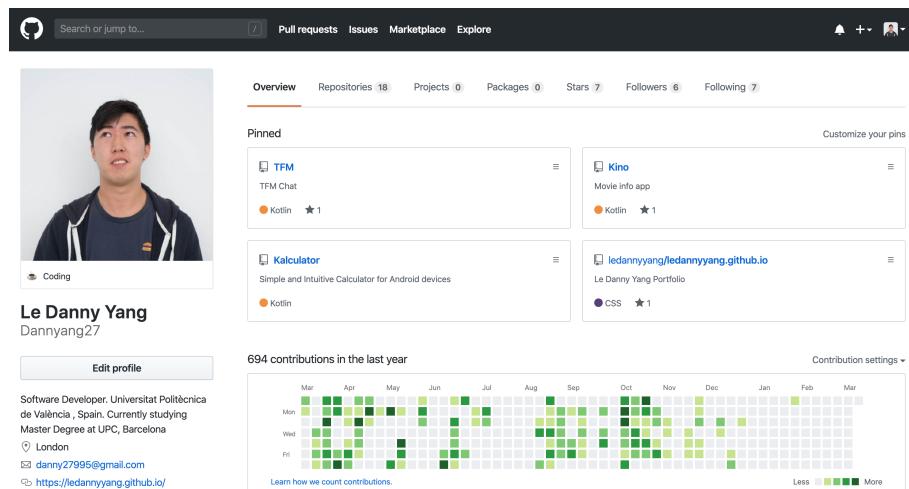
Para realizar cambios en los repositorios de *Github* se usan los comandos de *Git*. Si quieres aprender más acerca de *Git* puedes hacerlo en el siguiente link <https://dev.to/dhruv/essential-git-commands-every-developer-should-know-2f1>

```

1 // Dispatchers.Main
2 suspend fun fetchDocs() {
3     // Dispatchers.Main
4     val result = get("...")
5     // Dispatchers.Main
6     show(result)
7 }
8
9 // Dispatchers.Main
10 suspend fun get(url: String) =
11     // Dispatchers.Main
12     withContext(Dispatchers.IO) {
13         // Dispatchers.IO
14         /* perform blocking network IO here */
15     }
16 // Dispatchers.Main

```

**Listing 8:** Código de ejemplo de las corutinas y el uso de Dispatchers



**Figura 16:** Captura de pantalla de Github

## 3 Kotlin

En esta sección ampliaremos lo que ya hemos explicado acerca de Kotlin en apartados anteriores. Realizaremos un breve repaso a la sintaxis de Kotlin y todas las nuevas funcionalidades que este lenguaje aporta frente a Java, por lo que si ya conocéis Kotlin de antemano podéis saltar esta sección e ir directamente a [4](#).

### 3.1. Clases

Unas de las principales diferencias entre Kotlin y Java a la hora de crear clases, es en los POJOs o también denominado Plain Old Java Objects, que son las clases más triviales que existen. Para poner el mismo ejemplo que en el apartado [2.2](#), en los fragmentos de código [\(1, 2\)](#) permiten observar la facilidad que tiene Kotlin en crear dicha clase, más concretamente, se necesita una línea de código mientras que en Java, se necesita código repetitivo para implementar dicha clase. Es aquí, donde se observa la expresividad de Kotlin.

### 3.2. Herencia de clases

Por defecto, cualquier clase de Kotlin extiende de *Any*, bastante similar al *Object* en el lenguaje Java, por defecto en Kotlin las clases están cerradas y para que otra clase pueda extender de ella se necesita añadir *open* en la declaración.

```
1 open class Animal(name: String)
2
3 class Person(firstName: String, lastName: String): Animal(firstName)
```

**Listing 9:** Creación de una clase padre y su hijo

### 3.3. Funciones en Kotlin

Las funciones en Kotlin se crean usando la cláusula *fun*, al igual que en Java, se debe especificar el tipo de retorno (en el caso de no especificarse, devuelve *Unit*, equiparable al *void* en Java).

En los fragmentos de código [10](#) y [11](#) se puede observar dos formas totalmente equivalentes de crear la función suma, en el segundo caso, el propio lenguaje infiere que el tipo devuelto va a ser un *Int*, y por ello, es posible para el desarrollador explotar la expresividad que nos da Kotlin y crear la función en una simple línea.

```
1 fun add(x: Int, y: Int): Int {
2     return x + y
3 }
```

**Listing 10:** Creación de una función en Kotlin

```
1 fun add(x: Int, y: Int) = x + y
```

**Listing 11:** Creación de una función inline en Kotlin

Una de las cosas que nos podemos dar cuenta cuando programamos en Kotlin, es que los ';' son innecesarios al igual que en otros lenguaje de programación como Javascript y Python, cuando te acostumbras a ello, se ahorra una cantidad de tiempo notable.

Para acabar la sección de funciones, cabe mencionar que en Kotlin se puede especificar valores predeterminados en los argumentos, para seguir con el ejemplo del fragmento 11, en este caso hemos especificado que si no se aporta un valor al parámetro x, se le asignará un valor predeterminado de 0, por lo que la llamada al método posterior, devolvería un total de 20.

```
1 fun add(x: Int = 0, y: Int) = x + y
2
3 add(y = 20) // return 20
```

**Listing 12:** Creación de una función inline con valor predeterminado en la X

### 3.4. Variables

Para empezar esta sección, empezaremos diciendo que en Kotlin, absolutamente todo son objetos, es decir, en Java nos podíamos encontrar con tipos primitivos como eran los *int*, *float*, *long*, *char* y un sinfín de tipos. Esto es diferente en Kotlin, tal como hemos mencionado todo es un objeto y es debido a esto, que no existe la conversión automática de tipos. Esto se puede observar de manera más clara en el fragmento 13

```
1 val i: Int = 7
2 val d: Double = i.toDouble()
```

**Listing 13:** Conversión no automática de los tipos numéricos

En 13, se observa también otro detalle y es la cláusula *val*, en Kotlin existen las variables mutables (*var*) e inmutables (*val*), similar al **final** de Java. La ventaja de esto es que las variables inmutables son *thread-safe* por definición ya que una vez instanciadas, su valor no va a ser modificado a lo largo de toda la ejecución. Esto mejora la robustez del programa.

### 3.5. Extensión de funciones

La extensión de funciones, tal como su propio nombre indica, añade nuevos métodos a una clase incluso sin que tengamos acceso al código fuente de esa clase, por ejemplo si en nuestro proyecto queremos añadir una nueva función a la clase String, es posible gracias a la extensión de funciones que nos provee Kotlin.

Esta nueva funcionalidad puede llegar a ser de gran utilidad, y mejora de manera considerable la legibilidad del proyecto. En secciones anteriores habíamos presentado la librería Glide para el tratamiento de imágenes. Ahora bien, es posible que nosotros como desarrolladores, extendamos la clase ImageView y añadamos una nueva función tal que la que vemos en el fragmento 14

```
1 fun ImageView.loadUrl(url: String){
2     Glide.with(context).load(url).into(this)
3 }
```

**Listing 14:** Extendiendo la clase ImageView para realizar nuevos métodos de manera más intuitiva

Las funciones extensión no modifican la clase original, y se pueden declarar en cualquier parte del proyecto por lo que es una buena práctica crear ficheros específicos para llevar a cabo esta funcionalidad.

### 3.6. Null Safety y Elvis Operator

Quizás sea una de las funcionalidades más interesantes en Kotlin si vienes de desarrollar aplicaciones en Java.

En Kotlin, cuando queremos declarar que un objeto puede llegar a tener el valor *null*, se usa la cláusula '?' tal como vemos en el fragmento 15

```
1 val a: Int? = null
```

**Listing 15:** Uso de la cláusula ? para especificar que un objeto puede ser null

Por otra parte, no puedes trabajar con objetos *null* sin realizar comprobaciones previos, en el fragmento 16 se muestra un pequeño código que no compila.

```
1 val a: Int? = null
2 a.toInt() // error when compiling
```

**Listing 16:** Uso de la cláusula ? para especificar que un objeto puede ser null

En el ejemplo siguiente, se muestra la forma a la que los programadores en Java están acostumbrados a trabajar, en algunos proyectos existen innumerables casos de null checks mientras que si nos fijamos en el fragmento 18 la forma que tiene Kotlin en tratar estas comprobaciones es mucho más concisa.

```
1 val a: Int? = null
2
3 if(a != null){
4     a.toInt()
5 }
```

**Listing 17:** Comprobación de la variable si es null o no

En este siguiente caso, se ejecutará la llamada *toInt()* solo en el caso que la variable *a* no sea *null*. Este operando en Kotlin se denomina *safe call operator*. Ahora bien, qué pasa en los casos que queramos tratar cuando la variable *a* sea *null*. Existe otro operador denominado *Elvis operator* (? :), se llama así debido a que tiene cierta similitud con el «tupé» del famoso cantante estadounidense Elvis Presley y considerado por muchos el «Rey del Rock and Roll»

```
1 val a: Int? = null
2 a?.toInt()
```

**Listing 18:** Comprobación null con la cláusula ?

Siguiendo con los ejemplo, vamos a asignar un valor a otra variable en caso de que la variable *a* sea *null*

```

1 val a: Int? = null
2 ...
3 val value: Long = a?.toLong() ?: 0L

```

**Listing 19:** Asignación de valor cuando la variable a es null

También podemos lanzar un error cuando este caso ocurra tal como vemos en el fragmento 20:

```

1 val a: Int? = null
2 ...
3 val value: Long = a?.toLong() ?: throw IllegalStateException()

```

**Listing 20:** Lanzando error cuando la variable a es null

Por último, existen casos cuando una variable puede tener el valor *null* pero nosotros como desarrolladores sabemos con total certeza que en un momento dado, esa variable no es *null*, por lo que podemos forzar que dicha variable no sea considerado null por el compilador. Es lo que viene siendo el operador **!!**.

Hay que tener cuidado que un proyecto lleno de **!!** es un «smell», que en el contexto de desarrollador significa que puede ser un foco de errores y por ello hay que evitar usarlo lo máximo posible.

### 3.7. Collections

En 2014, la empresa Oracle, cuando lanzó la octava versión de Java, dicha versión incorporaba importantes cambios al lenguaje donde uno de ellos eran los Java Streams. La programación funcional está cada vez más de moda, y si la incorporamos en el uso de las colecciones o listas, obtenemos una potencia expresiva mayor. Con todo esto, se quiere decir que con la programación funcional en vez de decir cómo implementamos los cambios, decimos qué cambios queremos.

Esto se verá más fácilmente con el siguiente ejemplo en los fragmentos 21 y 22 que muestra los ejemplo en Java y Kotlin, respectivamente.<sup>3</sup>

```

1 List<Gasto> gastos= new ArrayList<Gasto>();
2 gastos.add(new Gasto("A",80));
3 gastos.add(new Gasto("B",50));
4 gastos.add(new Gasto("C",70));
5 gastos.add(new Gasto("D",95));
6
7 double resultado=gastos.stream()
8     .mapToDouble(gasto->gasto.getImporte()*1.21)
9     .filter(gasto->gasto<100)
10    .sum();
11
12 System.out.println(resultado);

```

**Listing 21:** Ejemplo de Java Streams

Si bien es cierto que Java en su octava versión ha mejorado notablemente el tratamiento de las colecciones, a pesar de todo, Kotlin sigue siendo claro vencedor en cuanto a potencia expresiva se refiere.

<sup>3</sup>Ejemplo extraído de <https://www.arquitecturajava.com/programacion-funcional-java-8-streams/>

```

1 val gastos= mutableListOf<Gasto>()
2 gastos.add(Gasto("A", 80))
3 gastos.add(Gasto("B", 50))
4 gastos.add(Gasto("C", 70))
5 gastos.add(Gasto("D", 95))
6
7 val resultado = gastos.map { it.importe * 1.21 }.filter { it < 100 }.sum()
8 print(resultado)

```

**Listing 22:** Ejemplo de streams en Kotlin

Además de los métodos usados en los fragmentos anteriores, existen un gran conjunto de operaciones sobre listas y que a continuación os mostraremos:

```

1 val list = listOf(1,2,3,4,5,6)
2
3 list.any { it % 2 == 0 } //comprueba si existe algun valor que sea par
4 list.all { it % 2 == 0 } //comprueba si todos los valores son numeros pares
5 list.count { it % 2 == 0 } //cuenta cuantos numeros son pares en la lista
6 list.max() //devuelve el numero maximo de la lista
7 list.min() //devuelve el numero minimo de la lista
8 list.filter { it % 2 == 0 } //devuelve los elementos que cumplan el
9     predicado
10 list.foreach{ print(it) } //realiza la operacion especificada para cada
    elemento de la lista

```

**Listing 23:** Lista de operaciones al tratar colecciones

### 3.8. Singleton en Kotlin

En la programación, el patrón de diseño **Singleton** resulta muy útil cuando se requiere que únicamente una instancia de ese objeto sea creada, y de esta manera evitamos la creación innecesaria de ese objeto ahorrando recursos vitales. También resulta muy útil cuando se requiere que un objeto se encargue de coordinar acciones en el sistema.

En los fragmentos de código siguientes, se muestra la diferencia entre la creación de una clase Singleton en Java y Kotlin, haciendo hincapié de nuevo en la potencia expresiva del lenguaje Kotlin.

```

1 class Singleton{
2     private static Singleton INSTANCE = null;
3
4     public static Singleton getInstance(){
5         if( INSTANCE == null ){
6             INSTANCE = new Singleton();
7         }
8
9         return INSTANCE;
10    }
11}

```

**Listing 24:** Ejemplo de creación de una clase Singleton en Java

```

1 object Singleton {}

```

**Listing 25:** Ejemplo de creación de una clase Singleton en Kotlin

Para concluir esta sección, podemos observar que Kotlin es mucho más expresivo (se necesita simplemente una línea de código) y libre de errores evitando código repetitivo que es una fuente casual de errores cuando estamos desarrollando cualquier aplicación.

## 4 Arquitectura

En esta sección hablaremos de la arquitectura en el contexto de la programación, cuando se habla de este término normalmente se refiere a los aspectos internos del diseño de un sistema software. Una buena arquitectura es fundamental en el proceso de creación de una aplicación. De otro modo, la arquitectura que hayamos escogido afectará considerablemente el éxito de la aplicación.

Durante la Google I/O del año 2017, Google anunció una nueva arquitectura para diseñar aplicaciones Android, denominada *MVVM* o también conocida como *Model View ViewModel*. Esta es la que nosotros hemos escogido para el desarrollo de este proyecto.

### 4.1. Model View ViewModel

Esta nueva arquitectura se compone principalmente de tres componentes que se pueden observar en la Figura 17:

1. **Model:** en este componente, es donde escribimos toda la lógica de negocio, es decir, si realizamos una llamada REST, este sería el encargado de procesar toda la información.
2. **View:** en el contexto de una aplicación Android, la view sería nuestra Activity o Fragment, a grandes rasgos, es la interfaz donde se visualiza el estado actual de la aplicación.
3. **ViewModel:** este componente es la pieza fundamental de la arquitectura, es el encargado de vincular la información obtenida a partir del Model y actualizar la View.

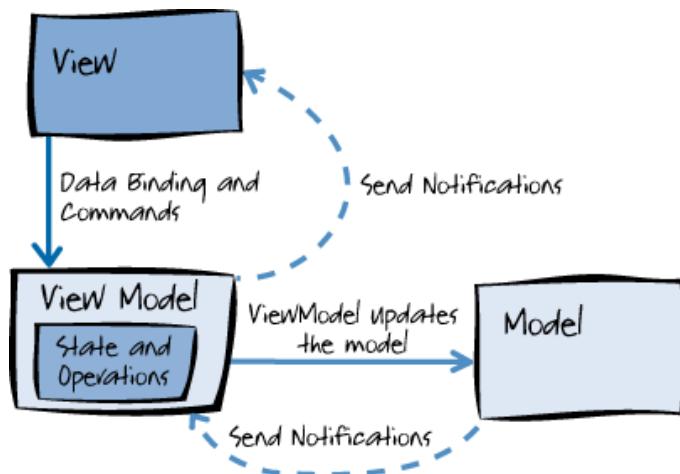


Figura 17: Esquema de la arquitectura MVVM

Dicho esto, ¿qué ventajas aporta seguir este patrón de diseño?

1. Realiza una separación entre la capa de negocio y la capa de presentación.
2. La vista no tiene constancia de lo que la capa de negocio está procesando en ningún momento, lo que facilita enormemente el trabajo del desarrollo a la hora de actualizar cualquier elemento de la vista.

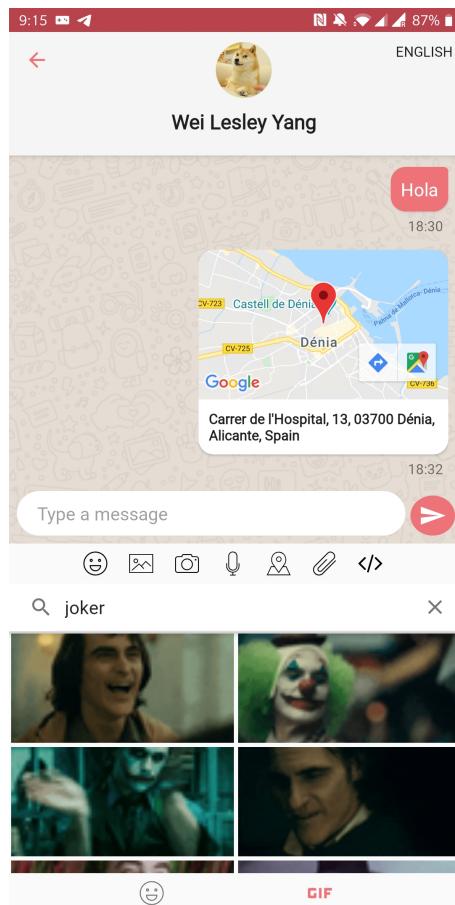
3. Entre el Model y el ViewModel, este último se olvida del trabajo del Model a la hora de obtener los datos, un ejemplo muy claro es que al ViewModel no le importa de donde se obtenga el dato, sino simplemente el dato, de esta manera el Model podría obtenerlo realizando llamadas REST o directamente de la base de datos.
4. Última gran ventaja, es que al estar usando ViewModel y LiveData (explicado en [4.2](#)), dichos componentes son *Activity Lifecycle aware* y este conlleva que en la práctica provoquen menos errores y *memory leaks* una vez ejecutamos la aplicación.

## 4.2. LiveData

Esta clase se le denomina una *Observable data holder class*, con respecto a otras clases observables, esta tiene en cuenta el ciclo de vida de las actividades en Android, esto quiere decir por ejemplo, que actualiza la vista solamente en caso de que la vista esté visible

## 4.3. Ejemplo de MVVM en el proyecto

En esta sección mostraremos cómo funciona este patrón de diseño en la práctica. Lo que se ha conseguido es lo que se ve en la Figura [18](#), desde el punto de vista del usuario, simplemente tendría que buscar el GIF que está buscando en ese momento y la vista se actualizaría cuando obtengamos el dato. Todo esto, se realiza de manera asíncrona sin bloquear el hilo principal. De esta manera, también mejoramos la experiencia de usuario.

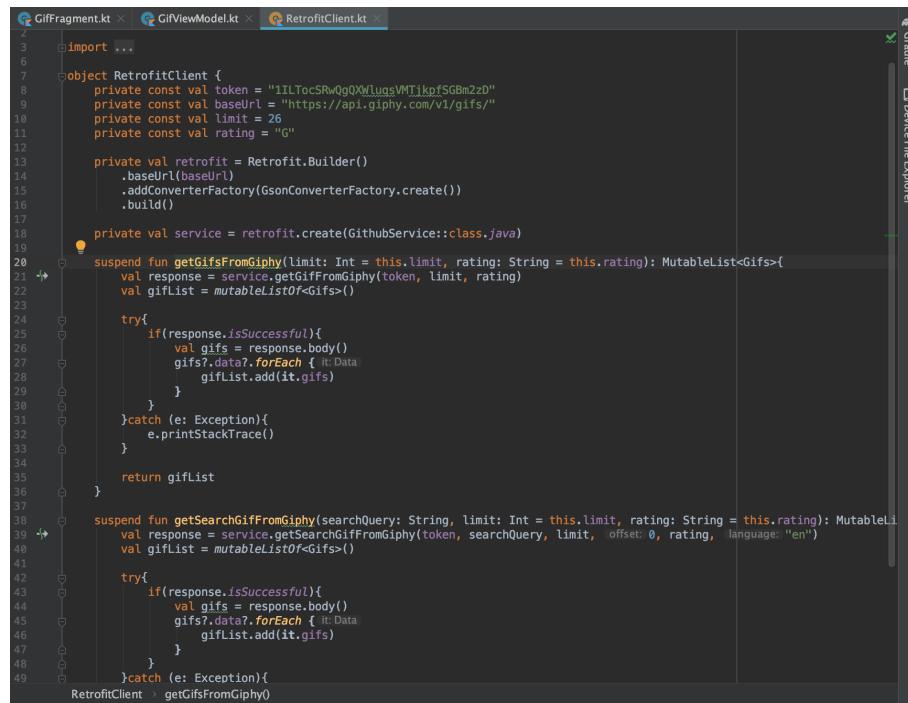


**Figura 18:** Interfaz del Chat Activity cuando el usuario quiere seleccionar un GIF

Comenzaremos con la Figura 19, en esta figura se puede observar que tiene una referencia a la clase *ViewModel*. Luego más adelante en la línea 32 instanciamos la clase y creamos el observable. Cualquier pequeño cambio en la lista de GIFs actualizaría la vista.

```

1 package com.example.tfm.fragments
2
3 import ...
4
5 class GifFragment : Fragment(){
6     private lateinit var adapter: GifAdapter
7     private lateinit var gifViewModel: GifViewModel
8     private val gifs = mutableListOf<Gifs>()
9
10    companion object{
11        fun newInstance(): GifFragment = GifFragment()
12    }
13
14    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {
15        val view = inflater.inflate(R.layout.fragment_gif, container, attachToRoot: false)
16        val gridView = view.findViewById(R.id.gif_gridview) as GridView
17        val searchView = view.findViewById(R.id.gif_searchview) as SearchView
18
19        gifViewModel = ViewModelProviders.of(activity!!).get(GifViewModel::class.java)
20        gifViewModel.getGifs().observe(activity!!, Observer { newGifs ->
21            gifs.clear()
22            gifs.addAll(newGifs)
23            adapter.notifyDataSetChanged()
24        })
25
26        adapter = GifAdapter(activity?.applicationContext!!, gifs)
27        gridView.adapter = adapter
28
29        searchView.setOnQueryTextListener(object: SearchView.OnQueryTextListener{
30            override fun onQueryTextSubmit(query: String?): Boolean {
31                KeyboardUtil.hideKeyboard(activity!!)
32                gifViewModel.searchGiphyGifs(query)
33                return true
34            }
35
36            override fun onQueryTextChange(newText: String): Boolean {
37                if(newText.isNotEmpty()){
38                    searchView.clearFocus()
39                }
40                return true
41            }
42        })
43
44        gifViewModel.getTrendyGiphyGifs()
45
46    }
47
48    GifFragment :: onCreateView
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
717
718
719
719
720
721
722
723
724
725
726
727
727
728
729
729
730
731
732
733
734
735
736
736
737
738
738
739
739
740
741
742
743
744
745
745
746
747
747
748
748
749
749
750
751
752
753
754
755
755
756
757
757
758
758
759
759
760
761
762
763
764
764
765
765
766
766
767
767
768
768
769
769
770
771
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
781
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
791
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
15
```



```
import ...

object RetrofitClient {
    private const val token = "IILToSRw0g0XWlugsVMTjpkfSGBm2zD"
    private const val baseUrl = "https://api.giphy.com/v1/gifs/"
    private const val limit = 26
    private const val rating = "g"

    private val retrofit = Retrofit.Builder()
        .baseUrl(baseUrl)
        .addConverterFactory(GsonConverterFactory.create())
        .build()

    private val service = retrofit.create(GithubService::class.java)

    suspend fun getGifsFromGiphy(limit: Int = this.limit, rating: String = this.rating): MutableList<Gifs>{
        val response = service.getGifFromGiphy(token, limit, rating)
        val gifList = mutableListOf<Gifs>()

        try{
            if(response.isSuccessful){
                val gifs = response.body()
                gifs?.data?.forEach { it.Data
                    gifList.add(it.gifs)
                }
            }
        }catch (e: Exception){
            e.printStackTrace()
        }
        return gifList
    }

    suspend fun getSearchGifFromGiphy(searchQuery: String, limit: Int = this.limit, rating: String = this.rating): MutableList<Gifs>{
        val response = service.getSearchGifFromGiphy(token, searchQuery, limit, offset: 0, rating, language: "en")
        val gifList = mutableListOf<Gifs>()

        try{
            if(response.isSuccessful){
                val gifs = response.body()
                gifs?.data?.forEach { it.Data
                    gifList.add(it.gifs)
                }
            }
        }catch (e: Exception){
            e.printStackTrace()
        }
        return gifList
    }
}
```

Figura 21: Fragmento de código relacionado con el Model

Y para concluir esta sección, se puede observar las ventajas que nos da a la hora de usar esta arquitectura en el proyecto, y el grado de separación que existe en nuestra aplicación de los distintos componentes, haciendo posible que si en un momento futuro queramos añadir nuevas funcionalidades, el cambio en nuestro código sería mínimo.

## 5 Diseño de Interfaces y código

---

La interfaz de usuario en una aplicación Android, es todo lo que el usuario puede ver e interactuar. Android provee una serie de componentes tales como Buttons, TextView, EditText que permiten al desarrollador diseñar las vistas. Por otra parte, Android provee también componentes especiales como son los diálogos, notificaciones y menús.

Antes de explicar cómo hemos realizado el diseño de todas las interfaces, cabe destacar los componentes más utilizados a la hora de desarrollar la aplicación:

1. **TextView**: elemento que muestra texto al usuario
2. **EditText**: elemento que permite al usuario añadir texto
3. **Button**: elemento que permite al usuario pulsarlo y realizar una acción asignada
4. **ImageView**: elemento que permite visualizar imágenes al usuario
5. **RecyclerView**: elemento de Android que permite mostrar al usuario una lista con componentes, denominados viewHolders (Véase la Figura 28)
6. **Toolbar**: elemento de la librería Android que se trata de una barra superior en la interfaz gráfica.
7. **SearchView**: elemento que permite al usuario introducir texto y realizar acciones de búsqueda
8. **MapView**: elemento de la librería Android que permite al usuario visualizar mapas, por ejemplo de la API de Google

Además de los componentes mencionados, existe un sinfín de componentes personalizados creados por la comunidad de Android.

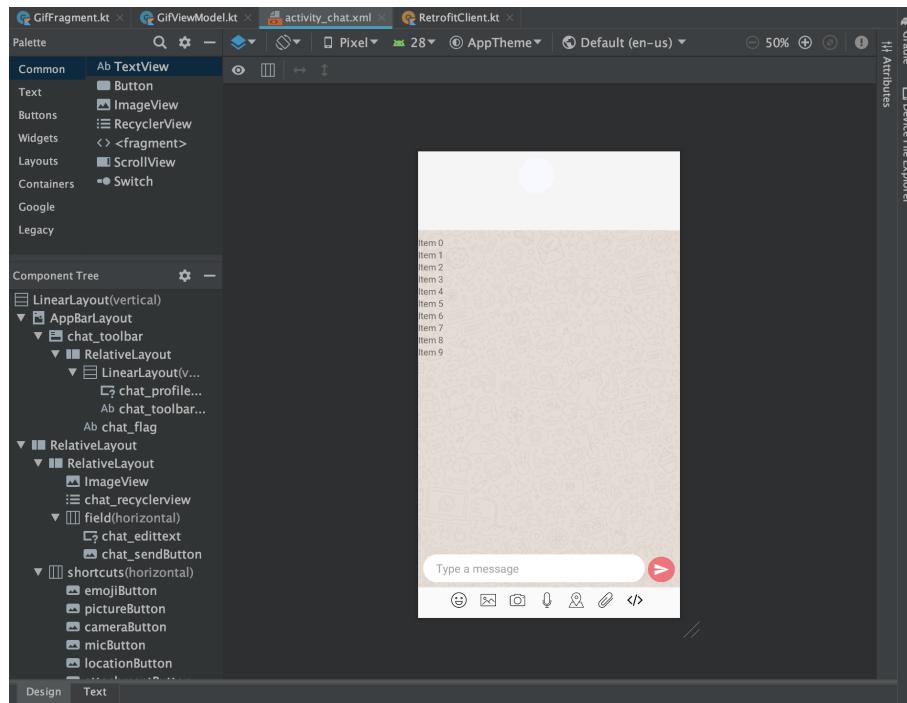
Tampoco nos podemos olvidar de los layouts que se utilizan en Android para estructurar el diseño de las pantallas:

1. **LinearLayout**: esquema usado que permite al desarrollar determinar la orientación (vertical u horizontal) de los componentes. También se permite concatenar diversos LinearLayouts, se ha de tener cuidado ya que reduce el rendimiento de la aplicación.
2. **RelativeLayout**: este esquema permite al usuario añadir componentes en relación a otros componentes, de ahí su nombre, esto quiere decir que si tuviésemos dos TextViews, podemos especificar que el segundo TextView esté posicionado a la derecha del primero, por ejemplo.

Existen diversos tipos de *layouts* en Android. Para más información consulte en <https://medium.com/androiddevelopers/layouts-attributes-and-you-9e5a4b4fe32c>

Por otra parte, una vez descritos brevemente los distintos tipos de componentes, existen dos formas de definir las interfaces, mediante una interfaz gráfica (Figura 22) y mediante código XML (23)

Normalmente a la hora de diseñar las interfaz, se empieza añadiendo los componentes de manera gráfica y posteriormente acceder al código para ajustar los atributos con un grado mayor de detalle. Por ejemplo, darle un valor de 400 píxeles a la anchura de un botón.



**Figura 22:** Diseñando la interfaz de la Actividad de manera gráfica. Esta vista corresponde a ChatActivity.kt donde se mostrará la lista de mensajes enviados entre dos usuarios.

De ahora en adelante, cuando hablamos de Activity, nos referiremos a una pantalla de nuestra aplicación.

A continuación nombraremos brevemente las distintas actividades que tenemos dentro del proyecto:

1. **LoginActivity:** permite al usuario iniciar sesión mediante unas credenciales
2. **SignupActivity:** permite al usuario crear nuevas cuentas
3. **MainActivity:** en esta actividad se muestra la lista de conversaciones que posee un usuario, también servirá para moverse por las distintas actividades dentro de la aplicación
4. **ChatActivity:** muestra la lista de mensajes con el otro usuario, permite una serie de distintos tipos de mensajes como texto plano, emoticonos, imágenes, GIF, localización.
5. **ImageDisplayActivity:** muestra información relacionado con la multimedia (imagen y GIF)
6. **ImageToolActivity:** permite al usuario rotar la imagen antes de ser enviada como mensaje.
7. **LocationSenderActivity:** permite al usuario navegar alrededor de un mapa y seleccionar la dirección deseada antes de ser enviada
8. **UserSearcherActivity:** permite buscar a cualquier usuario registrado en la aplicación
9. **UserProfileActivity:** permite al usuario actualizar datos que serán visualizados dentro de la aplicación (imagen de perfil, nombre de usuario, estado)
10. **SettingActivity:** permite al usuario descargar y seleccionar el traductor deseado de entre una lista de lenguajes (inglés, español, alemán, chino, japones, etc...)

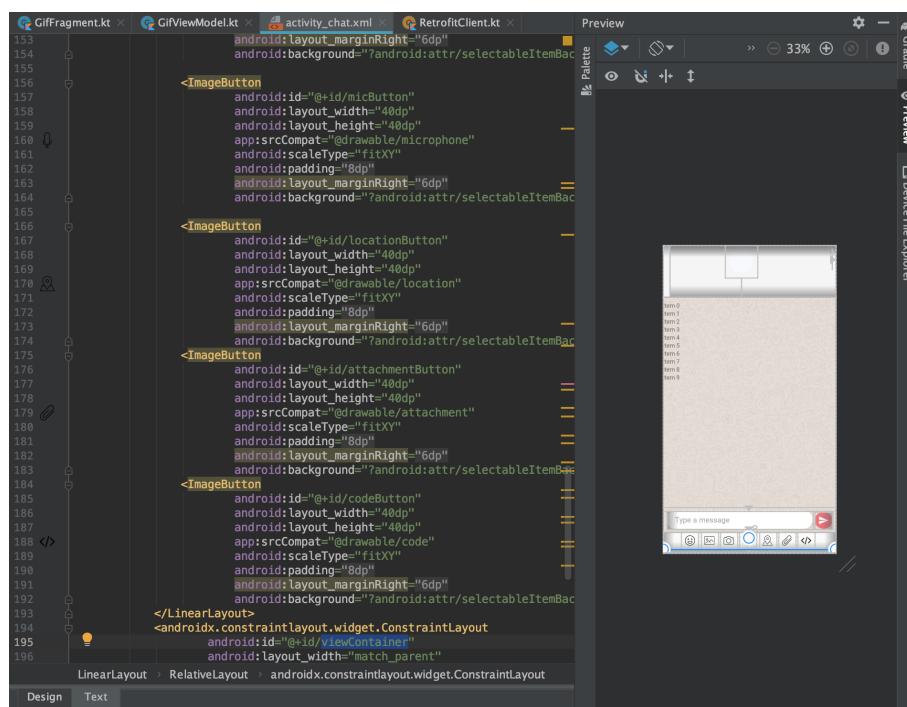


Figura 23: Diseñando la interfaz de la Actividad mediante código XML

## 5.1. Permisos

En una aplicación Android, un usuario debe aceptar explícitamente los permisos que este concede a la aplicación, si el usuario no está de acuerdo con ciertos permisos, entonces dicha funcionalidad deberá ser desactivada, por ejemplo, véase en la figura 24.

Por otra parte, los permisos que se les pide al usuario deberán ser lógicos, no estaría bien visto preguntar al usuario permisos de localización si dicha aplicación no hace uso de esta funcionalidad.

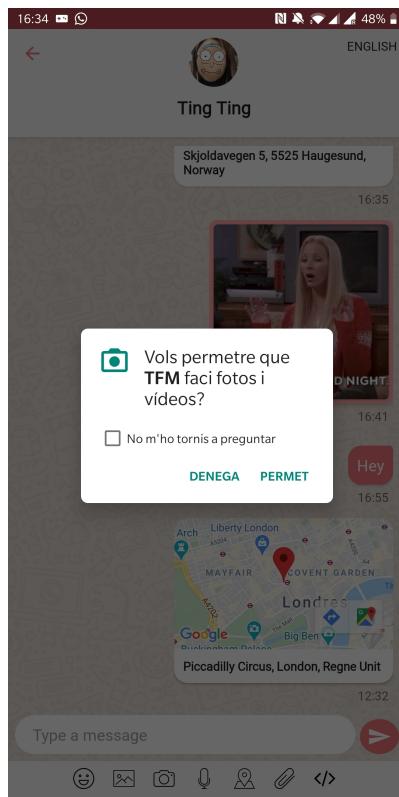


Figura 24: Petición de permisos en Android

En el fragmento de código 26 se observa que la aplicación comprueba si dichos permisos están aceptados por el usuario, en caso contrario se realiza una petición en forma de alerta tal como se ve en la figura 24.

Si los permisos ya han sido aceptados previamente, entonces en este ejemplo en concreto, se procede a activar la cámara.

```

1 @OnClick(R.id.cameraButton)
2 fun cameraBtn() {
3     if(ContextCompat.checkSelfPermission(this,
4         Manifest.permission.CAMERA) != PackageManager.
5             PERMISSION_GRANTED){
6         ActivityCompat.requestPermissions(this,
7             arrayOf(Manifest.permission.CAMERA),
8             DataRepository.CAMERA_PERMISSION)
9     } else{
10         openCamera()
11     }
12 }
```

Listing 26: Método cameraBtn() para cuando el usuario pulsa sobre el ícono de la cámara

```

1  override fun onRequestPermissionsResult(requestCode: Int, permissions:
2      Array<out String>, grantResults: IntArray) {
3      if ((grantResults.isNotEmpty() && grantResults[0] == PackageManager.
4          PERMISSION_GRANTED)) {
5          when(requestCode){
6              DataRepository.CAMERA_PERMISSION -> {
7                  openCamera()
8              }
9
10             DataRepository.STORAGE_PERMISSION -> {
11                 openGallery()
12             }
13
14             DataRepository.LOCATION_PERMISSION -> {
15                 openLocation()
16             }
17
18             DataRepository.AUDIO_PERMISSION -> {
19                 openMic()
20             }
21         }
22     }
23 }
```

Dentro de nuestra aplicación existen varios permisos:

1. **Acceso a la memoria:** para poder acceder a las imágenes del usuario y poder enviar dichas imágenes al otro usuario.
2. **Localización:** para poder enviar la localización seleccionada al otro usuario.
3. **Grabación de voz:** para poder dictar el texto que se enviará al otro usuario.
4. **Acceso a Internet:** para poder hacer uso de la conectividad de internet ya que el envío y recepción de mensajes se realiza a través de Internet.

Por otra parte, en Android existe un sinfín de permisos creados para cada una de las diferentes situaciones que se pueden dar al crear una aplicación, la lista de los permisos se puede encontrar en <https://gist.github.com/Arinerron/1bcaadc7b1cbeae77de0263f4e15156f>

## 5.2. Login

En la Figura 25 se puede observar una pantalla de inicio muy sencilla, en la aplicación el usuario se deberá acceder mediante un correo electrónico y una contraseña, estos datos se guardan en Firebase debido a que este servicio es el encargado de tratar todo el tema de autenticación. En este caso, solo aceptamos correos con la extensión **@gmail.com** por simplicidad, aunque si quisiésemos aceptar otros dominios tales como **@hotmail.com** se podría activar en un instante desde la consola de Google Firebase.

A partir de esta Activity, existen dos opciones para el usuario, si dicho usuario no tiene ninguna cuenta tendrá la opción de crear una cuenta pulsamos el botón de *Sign up* que le llevará al Activity de la Figura 26 o bien, introducir sus credenciales y pulsar el botón de *Log in*.

Cabe mencionar, que si el usuario ya inició sesión previamente en la aplicación, la próxima vez que lance la aplicación, esta dirigirá al usuario directamente a la pantalla principal de la figura 27. Esto se consigue salvaguardando las credenciales en la aplicación.

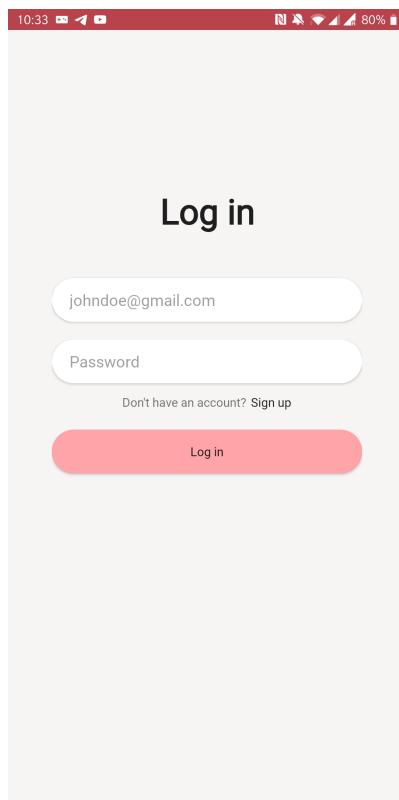


Figura 25: Pantalla de inicio donde el usuario accede a la aplicación mediante sus credenciales

```
1  @OnClick(R.id.login_signup_button)
2      fun launchSignup() {
3          startActivity(Intent(this, SignupActivity::class.java))
4      }
5
6  @OnClick(R.id.login_button)
7      fun login() {
8          if (eEmail.text.isNotEmpty() && ePassword.text.isNotEmpty()) {
9              login(eEmail.text.toString().trimBothSides(), ePassword.text.
10                  toString())
11          } else {
12              toast(R.string.field_not_empty)
13          }
14      }
```

Listing 27: Fragmento de código para accionar los botones de la pantalla de inicio

En el fragmento de código 27 se puede observar dos métodos, launchSignup() que al pulsarlo direcciona al usuario a la pantalla de registro (Figura 26) y login() donde primeramente realiza una comprobación de que los campos del correo electrónico y contraseña no estén vacíos. Si se cumplen estas condiciones Firebase en este caso comprobaría que las credenciales son correctas.

En caso de que las credenciales no fuesen correctas, la aplicación mostraría una alerta al usuario con el mensaje de *Wrong user/password* (Código 28). Por otra parte, en caso afirmativo actualizaríamos las credenciales en la aplicación y las variables tales como **DataRepository.user** y **DataRepository.currentUserEmail** que serán usadas más adelante,

```
1 fun login(context: Context, email: String, password: String) {
2     firebaseAuth?.signInWithEmailAndPassword(email, password)
3         ?.addOnCompleteListener { task ->
4             if (task.isSuccessful) {
5                 prefs = PreferenceManager.getDefaultSharedPreferences(
6                     context)
7                 prefs.updateCurrentUser(email, password)
8
9                 CoroutineScope(Dispatchers.IO).launch {
10                     val loginTask = Firebase Firestore.getInstance() .
11                         collection(FIREBASE_USER_PATH)
12                             .document(email).get().await()
13                     DataRepository.user = loginTask.toObject(User::class.
14                         java)
15                     DataRepository.currentUserEmail = email
16                     LoginViewModel.isSuccessful.postValue(true)
17                 }
18             } else {
19                 context.toast("Wrong user/password")
20                 LoginViewModel.isLoading.postValue(false)
21             }
22 }
```

**Listing 28:** Método login de la clase FirebaseUtil.kt

como por ejemplo, para determinar qué user profile actualizar cuando el usuario cambie su imagen de perfil.

### 5.3. Sign up

Una vez estamos en la pantalla de registro (*Sign up*), podemos dar marcha atrás a la pantalla de *Log in* o crear una usuario nuevo, estas deberán respetar una serie de condiciones para garantizar cierta seguridad y son:

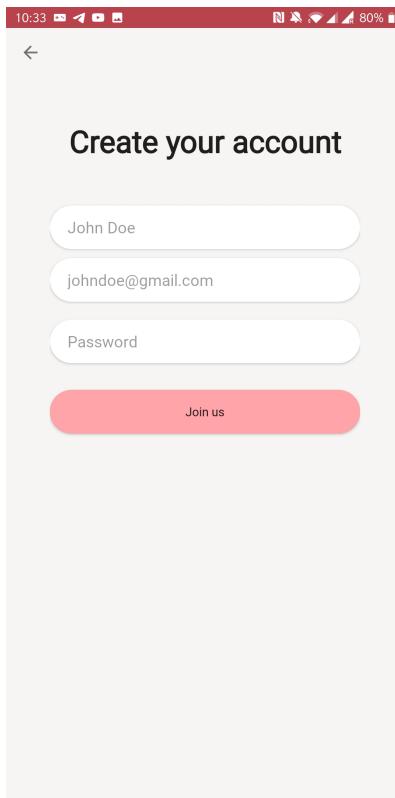


Figura 26: Diseñando la interfaz de la Actividad mediante código XML

1. El correo electrónico deberá terminar en `@gmail.com`
2. El correo electrónico no debe existir
3. La longitud de la contraseña deberá ser mayor de 8 caracteres

Ahora bien, en caso de haber iniciado sesión en la pantalla inicial o desde la pantalla de registro, si las credenciales son válidas, la aplicación guiará al usuario a la siguiente Activity (véas la figura 27).

Esto se puede ver en el fragmento de código 29 en el método `signup()`, primero comprueba que los campos de textos no estén vacíos y posteriormente hace una llamada al método `joinNewUser()` del `ViewModel`, que este a su vez realiza una llamada al método `createNewUser()` de la clase `FirebaseUtil.kt`.

```
1 fun joinNewUser(context: Context, username: String, email: String, password
2     : String){
3     FirebaseUtil.createNewUser(context, username, email, password)
}
```

Listing 30: Método `joinNewUser()` de la clase `SignupViewModel.kt`

```

1  private fun isFormNotEmpty() = eUser.text.isNotEmpty() && eEmail.text.
2      isNotEmpty() && ePassword.text.isNotEmpty()
3
4  @OnClick(R.id.signup_joinus)
5  fun signup() {
6      if(isFormNotEmpty()) {
7          disableViews()
8          val username = eUser.text.toString().trimBothSides()
9          val email = eEmail.text.toString().trimBothSides()
10         val password = ePassword.text.toString().trimBothSides()
11         signupViewModel.joinNewUser(this, username, email, password)
12     } else {
13         toast(R.string.field_not_empty)
14     }
15 }
```

**Listing 29:** Función signup para registrar un nuevo usuario

Si recordamos de secciones anteriores, concretamente en la sección 4, hemos hablado de que estructuraríamos nuestro proyecto siguiendo los patrones de diseño del Model View ViewModel. En este caso, la vista sería la clase SignupActivity.kt, el viewModel sería la clase SignupViewModel.kt y el model sería la clase FirebaseUtil.kt.

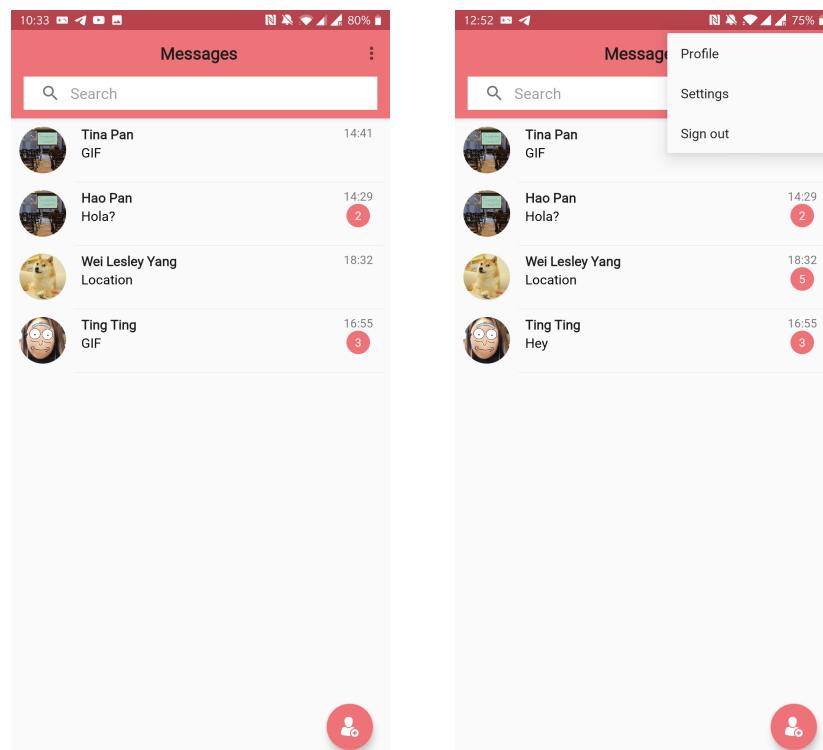
```

1  fun createNewUser(context: Context, username: String, email: String,
2      password: String) {
3      val user = User("", email, username, "", "")
4      val hashCode = user.hashCode().toString()
5      user.id = hashCode
6
7      val auth = FirebaseAuth.getInstance()
8      auth.createUserWithEmailAndPassword(email, password)
9          .addOnSuccessListener {
10             FirebaseFirestore.getInstance().addUser(context, user)
11             SignupActivity.currentUserEmail = email
12             SignupActivity.currentUserPassword = password
13         }.addOnFailureListener {
14             SignupViewModel.isJoinUsSuccessful.postValue(false)
15             context.toast("Cannot create user with those inputs")
16         }
17 }
```

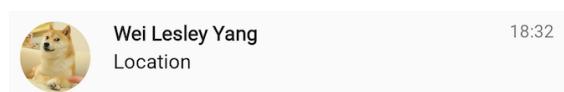
**Listing 31:** Método createNewUser que se encarga de crear el nuevo usuario y direccionar al usuario a la pantalla principal

## 5.4. Main Activity

Una vez en la pantalla principal (Figura 27), se puede observar que tenemos una *Toolbar* en la parte superior con el título de *Messages* y un *SearchView* para filtrar la lista de conversaciones que tenemos en un momento dado. Esto viene muy útil en el caso que tuviésemos una cantidad elevada de conversaciones. Luego por otra parte, en nuestra lista vemos que tenemos un conjunto de *ViewHolder* (Figura 28).



**Figura 27:** Pantalla principal donde se encuentran todas las conversaciones del usuario



**Figura 28:** Formato que tiene una conversación en la aplicación. Dentro del contexto de un desarrollador, este *item* se le conoce como *ViewHolder*

Dentro de esta pantalla tenemos varias opciones a realizar:

1. Filtrar conversaciones gracias al *SearchView* del *Toolbar*
2. Ir a la pantalla de la Figura 38
3. Ir a la pantalla de la Figura 40
4. Cerrar sesión e ir a la pantalla de la Figura 25
5. Pulsando el *FloatingActionButton* de la esquina inferior derecha, la aplicación nos llevará a la pantalla de la Figura 37
6. Si pulsamos en una conversación, se mostrará la *Activity* de la Figura 31 con los mensajes correspondientes.

Ahora bien, esta pantalla será la encargada de inicializar una serie de servicios que explicaremos a continuación:

1. La compatibilidad de los emoticonos, si no inicializamos este servicio, los emoticonos que recibamos de los usuarios puede causar errores en nuestra aplicación.
2. También inicializamos la base de datos para obtener la lista de conversaciones que tenemos guardados en nuestra base de datos SQLite (local).
3. El servicio de traducción, sin él cuando queramos traducir un mensaje de otro lenguaje al nuestro no se podría llevar a cabo.
4. El Service que se ejecuta en segundo plano que se encarga de leer los nuevos mensajes.

Todo lo mencionado se puede observar en el fragmento de código 32

```

1  private fun setupInitialiser() {
2      initEmoji()
3      initDatabase()
4      DataRepository.initTranslator(applicationContext)
5      MyNotificationManager.createNotificationChannel(this)
6  }
7
8  private fun initEmoji() = EmojiCompat.init(BundledEmojiCompatConfig(
9      applicationContext))
10
11 private fun initDatabase() = MyRoomDatabase.getMyRoomDatabase(this)

```

**Listing 32:** Inicialización de todos los métodos necesarios para el correcto funcionamiento de la aplicación

Tal como os podrías imaginar, el MainActivity.kt es una de las clases principales de nuestro proyecto, realiza una multitud de acciones mencionados anteriormente. Ahora nos dirigimos a cómo funciona cada una de las distintas partes.

Comenzaremos con el SearchView, este sirve para filtrar las conversaciones que tengamos en nuestra aplicación.

```

1  @BindView(R.id.search_chat) lateinit var searcher: SearchView
2
3  searcher.setOnQueryTextListener(object: SearchView.OnQueryTextListener{
4      override fun onQueryTextSubmit(query: String?) = true
5
6      override fun onQueryTextChange(newText: String?): Boolean {
7          if(newText.isNullOrEmpty()){
8              search_chat.clearFocus()
9          }
10
11         conversationViewModel.filterList(newText)
12         return true
13     }
14 })
15

```

**Listing 33:** Campo de búsqueda para filtrar las conversaciones

En el fragmento de código 33 hemos referenciado el SearchView mediante ButterKnife, el cual, nos evitamos en gran medida lo que se conoce como «boilerplate code» en el contexto de la programación.

Líneas más abajo, se puede observar que se le ha asociado un listener que filtrará cuando el usuario añada cierto texto. Por cada nuevo carácter que se añada, se llamará al método filterList(). Esto lo realiza el ViewModel del MainActivity siguiendo el modelo MVVM, tal como hemos explicado en secciones anteriores.

```

1  fun filterList(text: String?) {
2      val list = conversations
3          .filter { it.userOneEmail.removeAfter('@').contains(text.toString())
4              , ignoreCase = true) ||
5                  it.userTwoEmail.removeAfter('@').contains(text.toString(),
6                      ignoreCase = true)}
7          .toMutableList()
8
9      conversationList.postValue(list)
10 }
```

**Listing 34:** Método filterList() que se llama cada vez que el usuario añade un carácter en el SearchView

Por otra parte, toca explicar cómo hemos implementado la lista de conversaciones. Esto se ha realizado gracias a un componente de Android denominado RecyclerView

```

1  @BindView(R.id.conversations_recyclerview) lateinit var conversations:
2      RecyclerView
3
4  private fun initRecyclerView() {
5      viewManager = LinearLayoutManager(this)
6      viewAdapter = ConversationAdapter(mutableListOf())
7
8      conversations.apply {
9          setHasFixedSize(true)
10         addItemDecoration(HorizontalDivider(this.context))
11         layoutManager = viewManager
12         adapter = viewAdapter
13     }
14 }
```

**Listing 35:** Inicialización del RecyclerView en el MainActivity

En el fragmento de código 35 se puede observar como referenciamos el RecyclerView y al llamar el método initRecyclerView(), tal como su nombre indica, inicializamos la lista, y el resto de clases necesarias para poder visualizar el conjunto de conversaciones (Véase la Figura 29)

Por ello, para cada objeto personalizado que queramos visualizar en una lista, hay que crear una clase extendiendo de RecyclerView.Adapter<RecyclerView.ViewHolder>. El código asociado se puede observar en 36.

Cuando se extiende de RecyclerView.Adapter es necesario siempre sobreescribir tres métodos que son:

1. **onCreateViewHolder()**: este método especifica qué ViewHolder hay que crear, en nuestro caso ConversationViewHolder.kt que es el correspondiente a la Figura 28
2. **onBindViewHolder()**: este método se dedica a «mappear» el valor de las variables a los componentes que tenemos en ConversationViewHolder.kt.

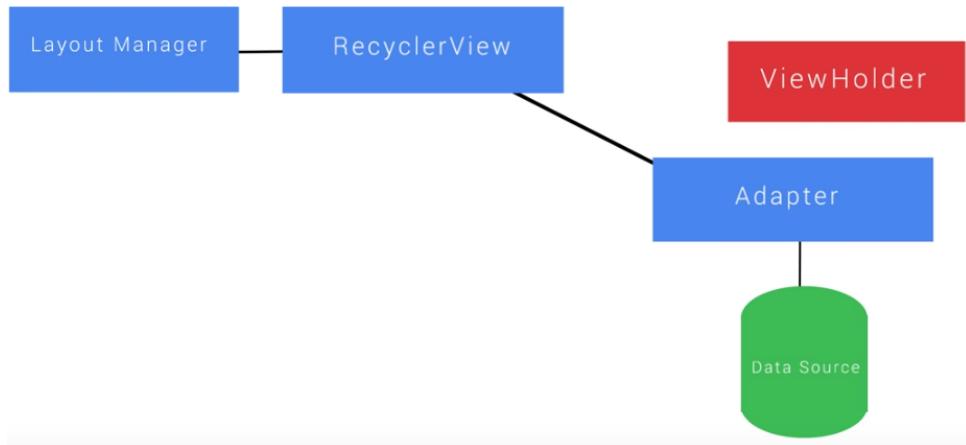


Figura 29: Esquema del funcionamiento de un RecyclerView

3. **getItemCount()**: este sencillo método devuelve la talla de la lista que le pasamos, en nuestro caso, el número de conversaciones que tenemos.

El FloatingActionButton, también conocido como fab comúnmente, se comporta de manera muy similar a un botón, de hecho, el fab extiende de la clase ImageButton, que este a su vez extiende de Button. En nuestro caso, esta variación del clásico botón se utiliza para navegar a UserSearcherActivity.kt (Véase la Figura 37)

```

1 @OnClick(R.id.fab)
2 fun fabClick() {
3     startActivity(Intent(this, UserSearcherActivity::class.java))
4 }
  
```

Listing 37: Método asociado al FloatingActionButton

Ahora bien, si nos fijamos en 27, en la segunda imagen observamos que tenemos una pestaña con tres opciones, al pulsar en cualquier opción navegaremos a UserProfileActivity.kt, SettingsActivity.kt o LoginActivity.kt respectivamente.

Una peculiaridad es si pulsamos en «Sign out», la aplicación nos mostrará una alerta (Véase la Figura 30) de si estamos seguro en cerrar ya que borraremos los datos asociados a la cuenta particular. Esto se ha decidido de esta manera para que el siguiente usuario al iniciar sesión, no tenga en su dispositivo información de la sesión previa liberando de esta manera memoria.

Existe otra funcionalidad, si pulsamos una imagen de perfil se nos muestra un diálogo con dicha imagen, hemos hecho uso de las *Function Extension* que hemos explicado en 14.

## 5.5. Chat Activity

Junto a MainActivity.kt, esta es una de las pantallas más importantes dentro de la aplicación, ya que es aquí donde se encuentra la funcionalidad básica, enviar mensajes. Como se puede observar en la Figura 31, en la parte superior, en el Toolbar se encuentra la imagen de perfil seleccionada por el otro usuario y su nombre. En la parte superior derecha nos indica el lenguaje en el que tenemos configurado nuestra aplicación, es decir, si fuese

```

1 class ConversationAdapter(private val conversations: MutableList<Conversation>)
2     : RecyclerView.Adapter<ConversationViewHolder>(){
3     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
4         ConversationViewHolder {
5             val view = LayoutInflater.from(parent.context).inflate(R.layout.
6                 viewholder_conversation, parent, false)
7             return ConversationViewHolder(view)
8         }
9
10    override fun onBindViewHolder(holder: ConversationViewHolder, position: Int
11        ) {
12        val conversation = conversations[position]
13        holder.bindViewHolder(conversation)
14    }
15
16    override fun getItemCount() = conversations.size
17
18    fun updateList( newConversations : MutableList<Conversation>){
19        val diffResult: DiffUtil.DiffResult = DiffUtil.calculateDiff(
20            ConversationDiffCallback(conversations, newConversations))
21        conversations.clear()
22        conversations.addAll(newConversations.sortedByDescending { it.timestamp
23            })
24        diffResult.dispatchUpdatesTo(this)
25    }
26}

```

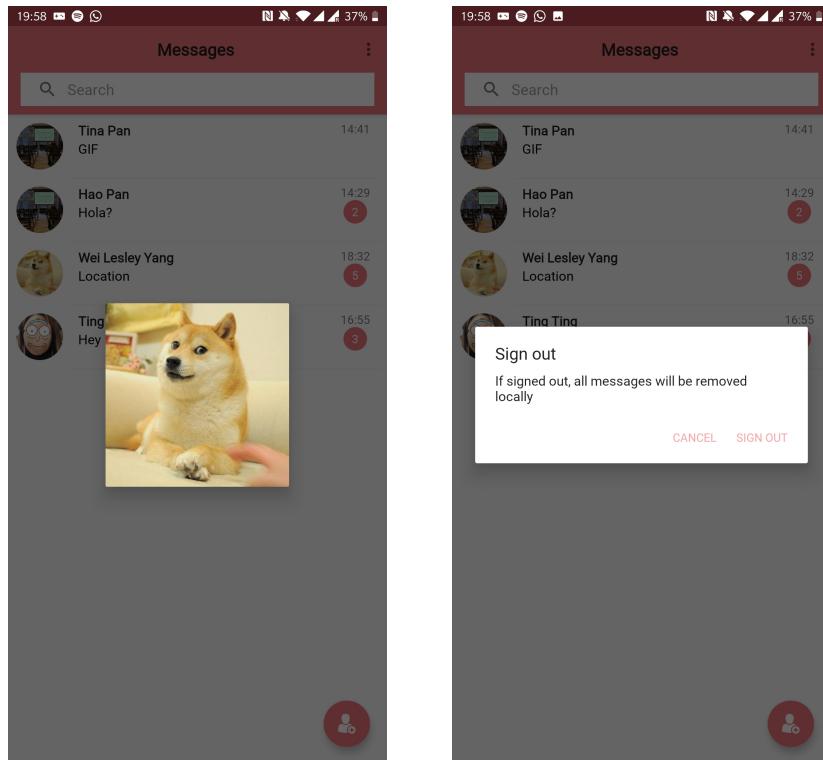
**Listing 36:** Código asociado al adaptador para poder visualizar la lista de conversaciones

```

1 private fun createSignoutDialog(): AlertDialog{
2     return AlertDialog.Builder(this)
3         .setTitle(R.string.signout)
4         .setMessage(R.string.signout_alert)
5         .setPositiveButton(R.string.sure) { _, _ ->
6             signout()
7         }
8         .setNegativeButton(R.string.cancel) { dialog, _ ->
9             dialog.dismiss()
10        }
11
12        .create()
13    }
14
15    private fun signout(){
16        FirebaseAuth.getInstance().signOut()
17        PreferenceManager.getDefaultSharedPreferences(applicationContext).
18            clearCredential()
19        startActivity(Intent(this, LoginActivity::class.java))
20        finish()
21    }

```

**Listing 38:** Método createSignoutDialog() que mostrará al usuario una alerta



**Figura 30:** A la izquierda se puede observar cuando el usuario pulsa la imagen de perfil de una conversaciones. A la derecha se muestra la alerta cuando el usuario quiere cerrar sesión

```

1 fun CircleImageView.showDialog(context: Context, imageBase64: String?) {
2     try {
3         imageBase64?.let {
4             val dialog = Dialog(context)
5             dialog.setContentView(R.layout.dialog_imagedisplay)
6             val dialogPhoto = dialog.findViewById<ImageView>(R.id.
7                 dialog_imagedisplay)
8             Glide.with(context).load(imageBase64.toBitmap()).into(dialogPhoto)
9             dialog.show()
10        }
11    } catch (e: Exception) {
12        e.printStackTrace()
13    }
}

```

**Listing 39:** Extensión de método a la clase CircleImageView con esto conseguimos que el método showDialog pueda ser utilizado por cualquier instancia de CircleImageView dentro de la aplicación

«English» los mensajes provenientes del resto de usuarios se traducirían de su idioma al inglés.

Luego más adelante, nos encontramos con nuestra lista de mensajes, y estos mensajes pueden ser de varios tipos:

1. Texto plano
2. Imagen
3. GIF (Graphic Interchange Format)
4. Localización



Figura 31: Captura de pantalla de ChatActivity.kt

Por simplicidad, hemos evitado los formatos de vídeos y voz ya que no estaba dentro de nuestros objetivos.

Además, dentro de esta pantalla tenemos varios fragmentos que se encargarán de añadir emoticonos (Figura 32), GIFs, imágenes, localización (Figura 36)

Comenzaremos explicando cómo hemos implementado el Toolbar que podemos observar, se componen de un CircleImageView, un TextView para el nombre de usuario y otro para mostrar el usuario que lenguaje de traducción ha escogido para traducir los mensajes.

Por otra parte, tenemos la lista de mensajes que como hemos explicado anteriormente, puede ser de varios tipos. A continuación veremos en 40

Vemos en el fragmento de código que dependiendo del tipo de mensaje (texto, imagen, gif, localización) creamos un tipo de ViewHolder u otro, por otra parte, también tendremos

```
1 override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
2     RecyclerView.ViewHolder {  
3         var view: View?  
4  
5         when(MessageType.fromInt(viewType)){  
6             MessageType.MESSAGE -> {  
7                 view = LayoutInflater.from(parent.context).inflate(R.layout.  
8                     viewholder_message, parent, false)  
9                 return MessageViewHolder(view)  
10            }  
11            MessageType.IMAGE, MessageType.GIF ->{  
12                view = LayoutInflater.from(parent.context).inflate(R.layout.  
13                    viewholder_image, parent, false)  
14                return ImageViewHolder(view)  
15            }  
16            MessageType.LOCATION -> {  
17                view = LayoutInflater.from(parent.context).inflate(R.layout.  
18                    viewholder_location, parent, false)  
19                return LocationViewHolder(view)  
20            }  
21            MessageType.ATTACHMENT -> {  
22                view = LayoutInflater.from(parent.context).inflate(R.layout.  
23                    viewholder_attachment, parent, false)  
24                return AttachmentViewHolder(view)  
25            }  
26        }  
27    }
```

**Listing 40:** Fragmento de código relacionado a la creación de un mensaje según su tipo

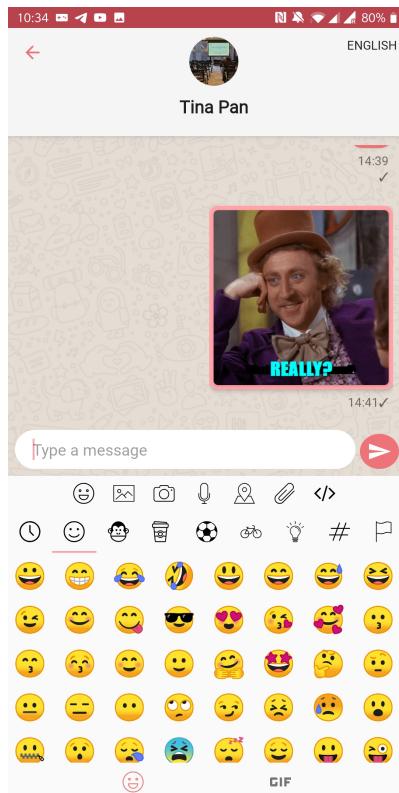


Figura 32: ChatActivity mostrando los emoticonos

que iniciar el contenido de dichos mensajes, tal como explicamos en la sección anterior, esto se consigue dentro de la clase adaptador, en nuestro caso ChatAdapter.kt

En el fragmento de código 41 destaca una serie de puntos:

1. Podemos observar que dependiendo de si el emisor de mensaje es uno mismo o no (Véase la Figura 43), renderizamos el texto plano como emisor o receptor, esto se ha conseguido modificando los atributos del TextView (alineación, color, gravedad, etc...)
2. El segundo punto es cómo realizamos la traducción del mensaje, dentro de la instancia de *Message*, se encuentra un atributo que nos indica en qué idioma se envió originalmente y también el mensaje traducido al inglés, si coincide con tu preferencia entonces no se traduce, en caso contrario se traduciría del inglés al idioma escogido.

Por último en esta sección hablaremos de los fragmentos que tenemos en la barra inferior, si nos fijamos en la Figura 32 se puede observar que tenemos una serie de pestaña y las funcionalidades las explicaremos a continuación:

1. **Emoticonos y GIFs:** esta primera pestaña nos muestra una serie de emoticonos (Figura 33), básicamente estos emoticonos son caracteres Unicode. Luego en la parte inferior, podemos seleccionar la opción de GIF (Figura 18)
2. **Seleccionar imagen:** al pulsar sobre esta pestaña, llevará al usuario a su galería interna del dispositivo, al seleccionar una foto lanzará ImageDisplayActivity.kt

```

1 override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position:
2     Int) {
3     val message = messages[position]
4
5     when(holder) {
6         is MessageViewHolder -> {
7             holder.initMessageViewHolder(message)
8         }
9
10        is ImageViewHolder -> {
11            holder.initImageViewHolder(message)
12        }
13
14        is LocationViewHolder -> {
15            holder.initAndUpdateMap(context, message)
16        }
17
18        is AttachmentViewHolder -> {
19            holder.initAttachmentViewHolder(message)
20        }
21    }
}

```

**Listing 41:** Código asociado a inicializar un texto plano en el ViewHolder

3. **Cámara:** muy similar a la selección de imágenes, el usuario cuando realice una foto con la cámara del dispositivo, la aplicación se mostrará en ImageDisplayActivity.kt
4. **Dictado por voz:** al pulsar esta pestaña, el dispositivo comenzará a escuchar lo que el usuario le dicte y mostrará el texto en *EditText* para mandar el mensaje posteriormente.
5. **Localización:** al pulsar sobre esta pestaña, el dispositivo dirigirá al usuario a la pantalla de LocationSenderActivity.kt
6. **Archivos adjuntos:** se ha dejado la implementación para versiones futuras
7. **Bloque de código:** se ha dejado la implementación para versiones futuras

A continuación, explicaremos el funcionamiento del envío de un mensaje ya que, dependiendo del tipo de mensaje que queramos enviar en un momento dado será de una forma u otra. En este ejemplo trataremos el envío de un texto normal.

Cuando el usuario escribe un mensaje en el *EditText* y pulsa el botón ocurre una serie de acciones. La primera acción es que comprueba que dicho campo no esté vacío, si el contenido no es vacío entonces procedemos a crear una instancia de **Message** llenando los campos pertinentes (*id*, *conversationId*, *senderEmail*, *receiverEmail*, *MessageType*, *MessageBody*, *timestamp*). También comprobamos si el lenguaje del usuario es inglés (lenguaje intermedio), en caso afirmativo, no traduciremos el mensaje, si está en cualquier otro idioma entonces se traduce este mensaje al inglés y se llama al método del fragmento [46](#).

ChatViewModel solamente llama a otro método haciendo de intermediario tal como manda el patrón MVVM.

En el fragmento de código [47](#) añadimos el mensaje en Firebase. Cuando nuestro listener detecta que este mensaje se ha añadido correctamente, procederemos a añadirlo

```

1 //texto plano
2 fun initMessageViewHolder(message: Message){
3     if (message.senderName == currentUserEmail){
4         setSenderViewHolder()
5     } else {
6         setReceiverViewHolder()
7     }
8
9     initLayout(message)
10
11    setTime(time, message.timestamp)
12    setMessageCheckIfSeen(time, message.senderName == currentUserEmail,
13                           message.isSent)
14
15    private fun initLayout(message: Message){
16        val code = message.body?.fieldThree?.toInt()
17
18        if (code == languagePreferenceCode || languagePreferenceCode == null){
19            body.text = message.body?.fieldOne
20        } else if (code != LanguageCode.ENGLISH.code && (languagePreferenceCode
21                  == LanguageCode.ENGLISH.code || languagePreferenceCode == null)){
22            body.text = message.body?.fieldTwo
23        } else {
24            val translator = DataRepository.fromEnglishTranslator
25            translator?.let {
26                var textToTranslate = if (message.body?.fieldThree?.toInt() ==
27                    LanguageCode.ENGLISH.code){
28                    message.body?.fieldOne
29                } else {
30                    message.body?.fieldTwo
31                }
32
33                translator.translate(textToTranslate.toString())
34                    .addOnSuccessListener { translatedText -
35                        body.text = translatedText
36                    }
37            }
38        }
39    }
}

```

**Listing 42:** Código asociado a inicializar un mensaje de texto plano

```

1 private fun setSenderViewHolder() {
2     val context = layout.context
3     val layoutParams = RelativeLayout.LayoutParams(
4         RelativeLayout.LayoutParams.MATCH_PARENT,
5         RelativeLayout.LayoutParams.WRAP_CONTENT)
6
7     layoutParams.setMargins(getDpValue(40), 0, 0, 0)
8     layout.layoutParams = layoutParams
9     layout.gravity = Gravity.END
10    placeholder.background = context.getDrawable(R.drawable.sender_message)
11    layout.setPadding(0, getDpValue(10), getDpValue(15), getDpValue(10))
12    body.setTextColor(context.getColor(R.color.colorWhite))
13    body.gravity = Gravity.START
14    time.gravity = Gravity.END
15 }

```

**Listing 43:** Fragmento de código para determinar si un mensaje es emisor o receptor

```

1  private fun setReceiverViewHolder() {
2      val context = layout.context
3      val layoutParams = RelativeLayout.LayoutParams(
4          RelativeLayout.LayoutParams.MATCH_PARENT,
5          RelativeLayout.LayoutParams.WRAP_CONTENT)
6
7      layoutParams.setMargins(0, 0, getDpValue(40), 0)
8      layout.layoutParams = layoutParams
9      layout.gravity = Gravity.START
10     placeholder.background = context.getDrawable(R.drawable.
11         receiver_message)
12     layout.setPadding(getDpValue(15),getDpValue(10), 0, getDpValue(10))
13     body.setTextColor(context.getColor(R.color.colorPrimaryText))
14     body.gravity = Gravity.START
15     time.gravity = Gravity.START
16 }

```

**Listing 44:** Fragmento de código para determinar si un mensaje es emisor o receptor

```

1 @OnClick(R.id.chat_sendButton)
2 fun sendMessage() {
3     val fieldText = chat_edittext.text.toString()
4     if(fieldText.isNotEmpty()){
5         val languageCode = FirebaseTranslator.languageCodeFromString(
6             translateModel.toString())
7         val timestamp = System.currentTimeMillis()
8
9         val message = Message(timestamp, conversationId, DataRepository.
10             currentUserEmail, receiverUser,
11             MessageType.MESSAGE.value, null, timestamp )
12
13         if(languageCode == LanguageCode.ENGLISH.code){
14             message.body = MessageContent(fieldText, "", languageCode.
15                 toString())
16             chatViewModel.sendMessage(message)
17         } else{
18             val translator = DataRepository.toEnglishTranslator
19
20             translator?.let{
21                 it.translate(fieldText).addOnSuccessListener {
22                     translatedText ->
23                     message.body = MessageContent(fieldText,
24                         translatedText, languageCode.toString())
25                     chatViewModel.sendMessage(message)
26                 }.addOnFailureListener {
27                     Log.d("TFM", "Cannot translate")
28                 }
29             }
30
31             chat_edittext.text.clear()
32         }
33     }
34 }

```

**Listing 45:** Método sendMessage del ChatActivity se traduce el mensaje al inglés al pulsar el botón

```

1 fun sendMessage(message: Message) {
2     FirebaseUtil.addMessageFirebase(message)
3 }

```

**Listing 46:** Método sendMessage de la clase ChatViewModel



**Figura 33:** Captura de pantalla de ChatActivity.kt

también a nuestra base de datos Room e inmediatamente actualizamos la conversación (por ejemplo el «último mensaje»)

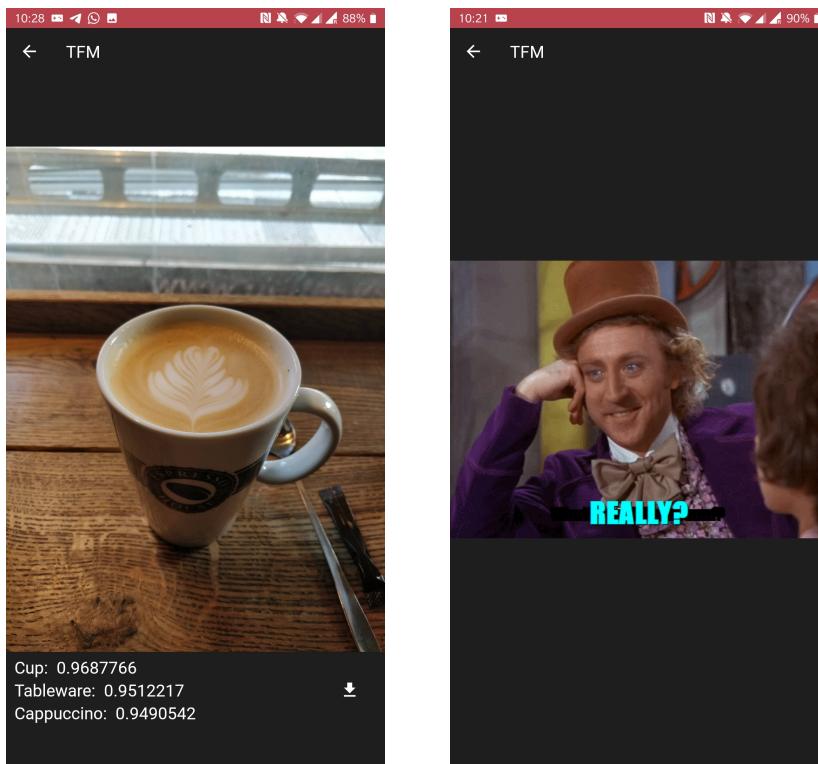
```
1 fun addMessageFirebase (message: Message) {
2     database . child (FIREBASE_PRIVATE_CHAT_PATH) . child (message . ownerId)
3         . child (FIREBASE_PRIVATE_MESSAGE_PATH)
4         . child (message . timestamp . toString ())
5         . setValue (message)
6         . addOnSuccessListener {
7             message . isSent = true
8             CoroutineScope (Dispatchers . IO) . launch {
9                 roomDatabase . addMessage (message)
10                updateConversation (message)
11            }
12        }
13        . addOnFailureListener {
14            Log . d (LogUtil . TAG, "Error while sending message")
15        }
16    }
```

**Listing 47:** Método addMessageFirebase de la clase FirebaseUtil

### **5.6. Image Display Activity**

Existen dos actividades encargadas de mostrar imágenes o GIFs al usuario, estas son ImageDisplayActivity.kt e ImageToolActivity.kt. La primera de ella se muestra cuando el usuario pulsa sobre una imagen o GIF (Véase la Figura 34) y la segunda cuando el usuario envía.

Cabe destacar que en el proyecto, las imágenes se guardan en base64 (si se desea más información acerca de él puede consultar en <https://es.wikipedia.org/wiki/Base64>), a grandes rasgos, este formato permite salvaguardar imágenes de tipo Bitmap a un tipo String en formato base64. Se ha realizado de esta manera para mantener cierta sencillez a la hora de trabajar con imágenes.



**Figura 34:** A la izquierda tenemos una imagen y a la derecha un GIF

En el fragmento de código 48 se puede observar ambos métodos que realizan esta acción.

```

1 fun Bitmap.toBase64() : String {
2     val outputStream = ByteArrayOutputStream()
3     this.compress(Bitmap.CompressFormat.JPEG, 70, outputStream)
4     return Base64.encodeToString(outputStream.toByteArray(), Base64.DEFAULT)
5 }
6
7 fun String.toBitmap() : Bitmap? {
8     val bitmap = Base64.decode(this, Base64.DEFAULT)
9
10    bitmap?.let {
11        return BitmapFactory.decodeByteArray(bitmap, 0, bitmap.size)
12    }
13
14    return null
15 }
```

**Listing 48:** Métodos para pasar de un Bitmap a una imagen en formato Base64 y viceversa

Por otra parte, el modo de funcionamiento entre una imagen o GIF difiere en algunos aspectos, en el primero se muestra información respecto al contenido de la imagen con una cierta probabilidad, en la Figura 34 se puede observar que el contenido de dicha imagen es una «cup» con una certeza del 96.88%.

Por ejemplo, en el fragmento de código 49 se observa que obtenemos una instancia del DeviceImageLabeler de FirebaseVision y a partir de ahí, dada una imagen, el labeler en cuestión procesa la información y nos devuelve los cocientes de probabilidad.

```

1  val labeler = FirebaseVision.getInstance().onDeviceImageLabeler
2
3  labeler.processImage(image)
4      .addOnSuccessListener {
5          it.forEachIndexed { index, label ->
6              when(index){
7                  1 -> { labelOne.text = "${label.text}:"
8                      confidenceOne.text = label.confidence.toString() }
9                  2 -> { labelTwo.text = "${label.text}:"
10                     confidenceTwo.text = label.confidence.toString() }
11                  3 -> { labelThree.text = "${label.text}:"
12                      confidenceThree.text = label.confidence.toString()
13                          }
14          }
15      }
16      .addOnFailureListener {
17          Log.d(LogUtil.TAG, "failure")
18      }

```

**Listing 49:** Código relacionado con FirebaseVision

Asimismo, para las imágenes tenemos la funcionalidad de descargarlas, esto se puede observar en el fragmento de código 50

```

1  @OnClick(R.id.imageDisplay_download)
2  fun downloadImage(){
3      val randomNum = (Math.random() * 100000 + 1).toInt()
4      bitmap?.let {
5          val path = MediaStore.Images.Media.insertImage(contentResolver,
6              bitmap, "IMAGE${randomNum}", "")
7          Uri.parse(path)
8          toast("Image saved")
9      }
10     finish()
11 }

```

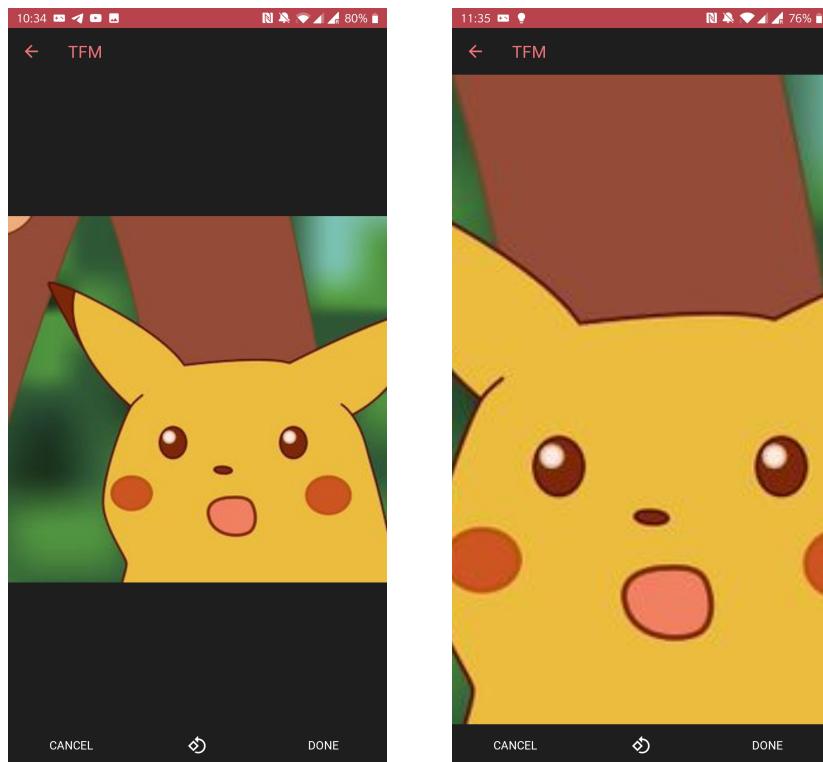
**Listing 50:** Método downloadImage() en la clase ImageDisplayActivity.kt

En cuanto al GIF, simplemente lo mostramos al usuario.

## 5.7. Image Tool Activity

Cuando un usuario selecciona una imagen desde la galería de su móvil, o bien, realiza una foto, se le dirige a ImageToolActivity.kt (Véase la Figura 35). Dentro de esta actividad se puede realizar cuatro acciones:

1. Ir a la actividad anterior (cancelar)
2. Rotar la imagen 90 grados en sentido antihorario
3. Enviar dicha imagen al otro usuario
4. Zoom a la imagen



**Figura 35:** Pantalla principal donde se encuentran todas las conversaciones del usuario

A continuación os mostraremos el código relacionado con cada una de las acciones junto a una breve explicación sobre el funcionamiento.

```
1 @OnClick(R.id.tool_cancel)
2     fun cancel() {
3         finish()
4     }
```

**Listing 51:** Método cancel()

La función del método `cancel()` es terminar la Activity, por lo que se muestra la actividad anterior, en este caso, la actividad desde donde se lanzó, que es `ChatActivity.kt`.

El método `rotate()`, como su nombre indica rota la imagen en un ángulo de 90 grados en sentido antihorario, en el código se puede observar que creamos un nuevo Bitmap rotado.

cuando el usuario pulsa «accept», se crea una instancia de Message con el correspondiente MessageType y realiza una llamada a la clase encargada de enviar/guardar los mensajes, en este caso FirebaseUtil y se procede a terminar la actividad.

Por último, si nos fijamos de nuevo en la segunda imagen en la Figura 35, vemos que el usuario también puede realizar un aumento sobre la imagen, esto se ha conseguido gracias a un componente externo a Android creado por MikeOrtiz (puede obtener más información en <https://github.com/MikeOrtiz/TouchImageView>)

```
1 @OnClick(R.id.tool_rotate)
2 fun rotate() {
3     content = touchImage.rotate()
4 }
5
6 //en ExtensionFunctionUtil.kt
7 fun ImageView.rotate(): String {
8     val matrix = Matrix().apply {
9         postRotate(-90F)
10    }
11    val bitmap = (drawable as BitmapDrawable).bitmap
12    val rotatedBitmap = Bitmap.createBitmap(bitmap, 0, 0, bitmap.width,
13        bitmap.height, matrix, true)
14    setImageBitmap(rotatedBitmap)
15
16    return rotatedBitmap.toBase64()
17 }
```

Listing 52: Método rotate()

```
1 @OnClick(R.id.tool_accept)
2 fun accept() {
3     val timestamp = System.currentTimeMillis()
4     val message = Message(timestamp, ChatActivity.conversationId,
5         currentUserEmail,
6         ChatActivity.receiverUser, typeValue, MessageContent(fieldOne =
7             content), timestamp)
8
9     FirebaseUtil.addMessageFirebase(message)
10    finish()
11 }
```

Listing 53: Método accept()

```
1 <com.ortiz.touchview.TouchImageView
2     android:id="@+id/tool_touchimage"
3     android:layout_width="match_parent"
4     android:layout_height="match_parent"
5     android:scaleType="fitCenter"
6     app:srcCompat="@drawable/white_placeholder"
7     android:layout_centerVertical="true"
8     android:layout_gravity="center"
9     android:visibility="gone"/>
```

Listing 54: Inicializando atributos al TouchImageView

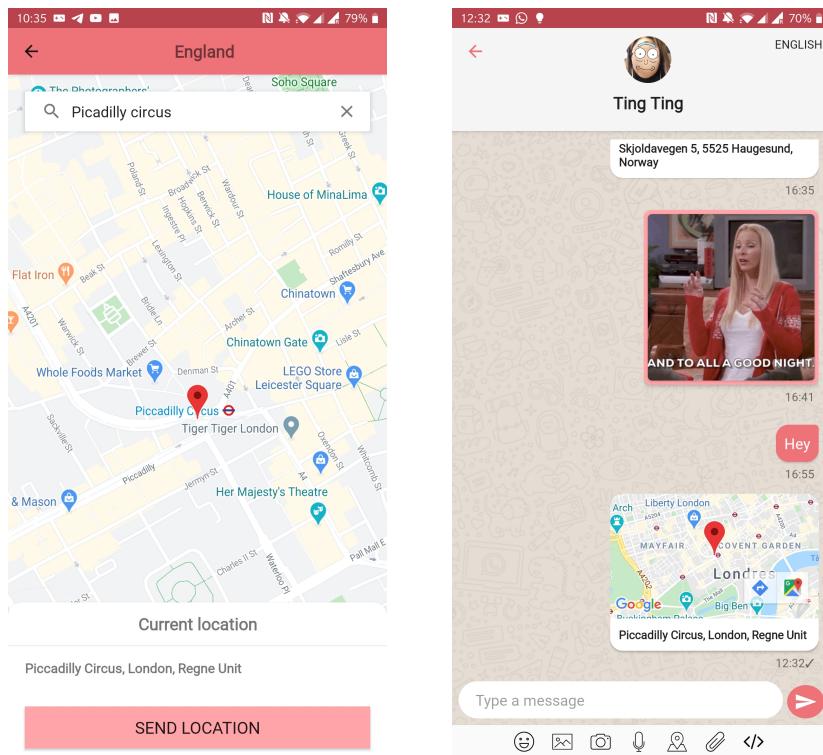
```

1 fun searchLocation(context: Context, query: String?) {
2     val address = Geocoder(context, Locale.getDefault()).getFromLocationName(query, 1)
3     if(address.isNotEmpty()) {
4         latLng.postValue(LatLng(address[0].latitude, address[0].longitude))
5     } else {
6         context.toast("No results")
7     }
8 }
```

**Listing 55:** Método searchLocation() para cuando el usuario introduce una dirección en el SearchView

## 5.8. Location Activity

Otra de las funcionalidades que tenemos en nuestra aplicación es el envío de localización. Por ello, en la actividad **LocationSenderIdActivity.kt** (Véase la Figura 36) tenemos un SearchView con un método asignado (Fragmento de código 55) para que el usuario pueda introducir cualquier dirección. En la Figura 36 se observa un ejemplo donde hemos introducido «Picadilly Circus» y la aplicación nos lleva a dicha dirección representándola con el puntero.



**Figura 36:** Pantalla principal donde se encuentran todas las conversaciones del usuario

En la segunda imagen de la Figura 36, se muestra un ejemplo de cómo se visualizaría la localización enviada en la conversación. Si pulsásemos en la localización, el sistema operativo de Android abriría un diálogo donde se mostraría la lista de aplicaciones capaces de abrir mapas. Por ejemplo, Google Maps.

Otra acción que el usuario puede realizar en **LocationSenderIdActivity.kt** es el hecho de hacer zoom-in y zoom-out, también el usuario tiene la posibilidad de moverse alrededor

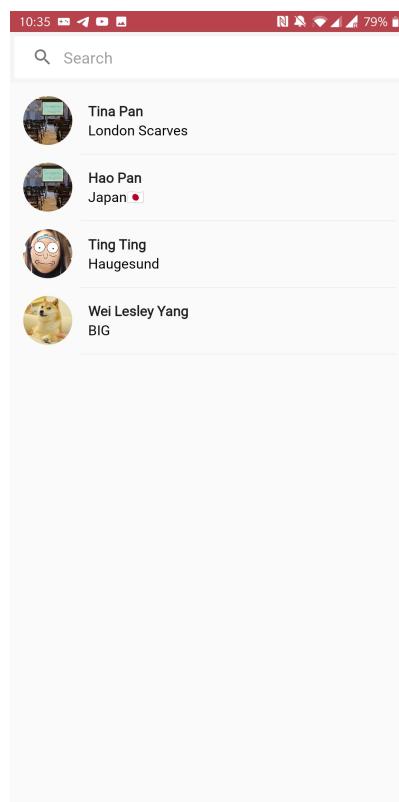
del mapa realizando una pulsación larga sobre la dirección que desea ir. En ese caso llamaría al método `moveToLocation()` que se observa en [56](#)

```
1  private fun moveToLocation(location: LatLng) {
2      googleMap?.apply {
3          clear()
4          animateCamera(CameraUpdateFactory.newLatLngZoom(location, 16F))
5          addMarker(MarkerOptions()
6              .position(location)
7              .icon(BitmapDescriptorFactory.defaultMarker(
8                  BitmapDescriptorFactory.HUE_RED)))
9      }
10     locationViewModel.setNewLocation(applicationContext)
11 }
```

**Listing 56:** Método `moveToLocation()`

## 5.9. User Searcher Activity

A esta actividad se llega desde `MainActivity.kt` al pulsar sobre el Floating Action Button. Principalmente esta actividad está compuesta por dos componentes, un `SearchView` para filtrar y realizar búsquedas de usuarios (en caso de tener muchos) y una lista de usuarios.



**Figura 37:** Diseñando la interfaz de la Actividad mediante código XML

Como se puede observar en la figura [37](#), un elemento de la lista está formada por una imagen de perfil, nombre de usuario y un estado. Cuando el usuario pulse sobre un elemento, nuestra aplicación comprobará si existe una conversación con dicho usuario, en caso negativo, se crearía la conversación añadiéndola a nuestra base de datos local y a Firebase.

Esto se puede observar en el fragmento de código 57, básicamente primero comprueba si dicha conversación se encuentra en la base de datos local (línea 9), si existe entonces la aplicación dirigirá al usuario a ChatActivity.kt donde se muestra la lista de mensajes. Ahora bien, en caso de no existir la conversación se procederá a crearla.

En el fragmento de código 58 se puede observar los atributos de los que se componen una conversación. Se necesita una id, que en nuestro caso es la concatenación de los *hashcodes* de ambos usuarios. En 57 está presente en la línea 23. Otro atributo que puede requerir un poco más de explicación puede ser el *timestamp*, este valor denota el tiempo en segundos desde el 1 de enero de 1970, a las cero horas. Esta medida o valor de tiempo se conoce también como el timestamp de Linux.

La anotación `@Ignore` significa que cuando la aplicación procede a guardar la conversación en Firebase, omite este campo.

## 5.10. User Profile Activity

Esta actividad (Figura 38) muestra información relacionada con el usuario de la sesión. Dentro de esta actividad el usuario tiene total libertad para actualizar la foto de perfil (pulsando sobre el Floating Action Button), para modificar su nombre de perfil y también el estado. Lo que el usuario no puede modificar es su correo electrónico debido a que se usa como credencial a la hora de iniciar sesión (Véase la Figura 25)

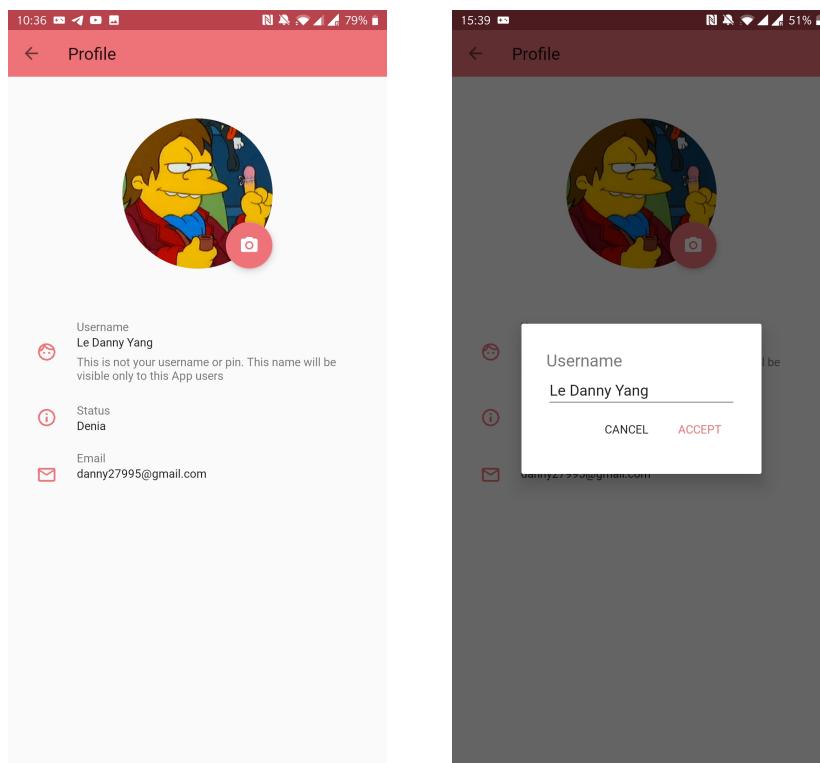


Figura 38: Actualización del nombre de usuario

Ahora bien, cuando el usuario pulsa sobre el nombre de usuario o bien sobre el estado, se muestra un diálogo (Véase la segunda imagen de 38), si el usuario modifica dicho valor y posteriormente acepta los cambios, esto se verá reflejado en Firebase (Figura 39)

```

1  private fun createConversationIfNone(context: Context, holder:
2      UserSearchViewHolder){
3      val user = DataRepository.user?.id?.toLong()
4      val conversationId = user?.getConversation(holder.id)
5
6      CoroutineScope(Dispatchers.IO).launch {
7          val roomDatabase = MyRoomDatabase.getMyRoomDatabase(context)
8          val conversation = roomDatabase?.conversationDao()?.getById(
9              conversationId.toString())
10
11         if(conversation != null){
12             context.launchChatActivity(conversationId.toString(), holder.
13                 email, holder.user, holder.photoBase64, false )
14         } else{
15             val myself = DataRepository.user
16             val friend = roomDatabase?.getUserByEmail(holder.email)
17             var userOneHash = myself?.id?.toLong()!!
18             var userTwoHash = friend?.id?.toLong()!!
19
20             if(userOneHash > userTwoHash){
21                 val tmp = userOneHash
22                 userOneHash = userTwoHash
23                 userTwoHash = tmp
24             }
25
26             val hashCode = userOneHash.toString().plus(userTwoHash.toString()
27                 ())
28             val conversation = Conversation(hashCode, myself.email, myself.
29                 name, myself.profilePhoto, friend.email, friend.name, friend
30                 .profilePhoto,
31                 mutableListOf(), "", System.currentTimeMillis() )
32
33             roomDatabase.addConversation(conversation)
34             FirebaseFirestore.getInstance().addConversation(context,
35                 conversation)
36
37             withContext(Dispatchers.Main){
38                 context.toast("Creating conversation ... ")
39             }
40         }
41     }
42 }
```

**Listing 57:** Método createConversationIfNone() que se encarga de crear una conversación en caso de no existir

```

1  @Entity(tableName = "Conversation")
2  data class Conversation (@PrimaryKey
3      var id: String = "",
4      var userOneEmail: String = "",
5      var userOneUsername: String = "",
6      var userOnePhoto: String = "",
7      var userTwoEmail : String = "",
8      var userTwoUsername: String = "",
9      var userTwoPhoto: String = "",
10     @Ignore @get:Exclude
11     var messages: MutableList<Message> = mutableListOf(),
12     var lastMessage: String? = "",
13     var timestamp: Long = -1)
```

**Listing 58:** Clase Conversation dentro de nuestra aplicación



**Figura 39:** Diseñando la interfaz de la Actividad mediante código XML

```

1  fun FirebaseFirestore.updateCurrentUser(context: Context, user: User) {
2      collection(FirebaseUtil.FIREBASE_USER_PATH)
3          .document(user.email)
4          .set(user)
5          .addOnSuccessListener {
6              MyRoomDatabase.getMyRoomDatabase(context)?.updateUser(user)
7              context.toast("User updated")
8              FirebaseUtil.updateUser(user)
9          }
10         .addOnFailureListener {
11             Log.d(LogUtil.TAG, "Error while updating user")
12         }
13     }

```

**Listing 59:** Método updateCurrentUser() donde actualiza un usuario en la base de datos de Firebase

Si observamos el fragmento de código 59, vemos que se actualiza en la base de datos local y posteriormente en Firebase, esto se ha diseñado de esta manera para salvaguardar la consistencia de los datos.

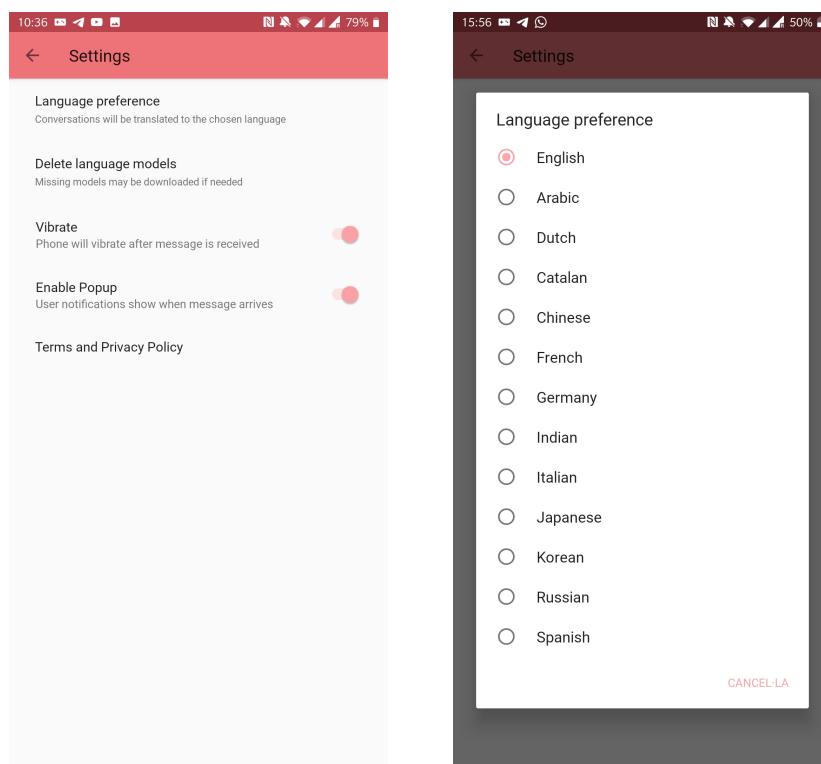
### 5.11. Settings Activity

En esta actividad se encarga de seleccionar qué idiomas escoge el usuario como el predeterminado, por ejemplo, si el usuario escoge como lenguaje el español, luego cuando esté en **ChatActivity.kt** y reciba mensajes del resto de usuario, estos mensajes se verán traducidos al español.

Por otra parte, cuando el usuario envíe un mensaje de texto, dicho mensaje se traducirá al inglés como idioma común para ser posteriormente traducido al idioma pertinente de cada usuario.

También a su vez, en esta actividad se permitirá al usuario borrar la lista de lenguajes debido a que cuando seleccionamos un idioma este se descarga y se guarda en nuestro dispositivo móvil y normalmente estos traductores ocupan alrededor de 30MB, cantidad lo suficientemente grande para que el usuario decida borrarlos.

En la figura 40 existe dos funcionalidades como son *Vibrate* y *Enable Popup*, estos no se han implementado en la aplicación debido a la falta de tiempo y no entran dentro de los objetivos del trabajo fin de Máster. Pueden ser vistos como una funcionalidad de vista al futuro.



**Figura 40:** Listado de lenguajes permitidos

## 6 ¿Cómo realizamos ciertas acciones?

---

### 6.1. ¿Cómo traducimos los mensajes?

Primeramente, cabe destacar que utilizaremos el inglés como lenguaje intermedio, debido a que nuestra aplicación soporta varias lenguas, es inviable tener guardado en nuestro dispositivo varios modelos de traducción a nuestro idioma, esto es, si escogiésemos el castellano como lengua principal, tendríamos que salvaguardar en nuestro dispositivo Android un modelo de traducción del italiano al castellano, francés al castellano y así sucesivamente con todos los lenguajes que soportamos.

Es obvio que esté hecho es inviable y recordamos que cada modelo de traducción ocupa cerca de 30MB.

Es por ello que necesitamos el uso de un idioma intermedio como puede ser el inglés. De este modo, cada usuario tendrá como mínimo un modelo de traducción tal que el idioma fuente sea el inglés y el idioma destino el lenguaje escogido para traducir el mensaje y de esta manera ahorramos espacio de almacenamiento.

Es verdad que sí escogemos esta manera de traducir los mensajes, el mensaje final no estará traducido de manera perfecto y en ciertos ocasiones puede que el mensaje tenga un significado distinto a lo que se quería transmitir originalmente, ya que con cada paso, añadimos un error.

Por otra parte, guardando relación con este hecho, podríamos solventar este problema si usásemos el servicio de pago de Google que traduce los mensajes en la nube y así evitamos mantener dichos modelos de traducción en nuestro dispositivo. Esto conlleva un coste que más adelante explicaremos con más detalle en [7](#).

### 6.2. ¿Cómo enviamos los mensajes?

En este apartado explicaremos el envío de mensajes dentro de nuestra aplicación. Cuando el usuario escribe un mensaje en el cuadro de texto y pulsa el botón de enviar, ocurre un cierto trabajo totalmente invisible para el usuario. Primeramente obtenemos todos los valores de los que se compondrán el mensaje tales como el id de la conversación pertinente, el emisor, receptor, timestamp etc, con esos valores creamos una instancia de **Message** en la aplicación y guardamos el mensaje dentro de nuestra base de datos local (SQLite).

Posteriormente, la aplicación intentará guardar este mensaje en Firebase, donde una vez conseguido guardarlo llamará un «callback» (`addOnSuccessListener`) que confirmará que dicho mensaje se ha guardado con éxito. En este caso, actualizamos nuestro mensaje en local asignando el atributo `isSent` a `True`, por lo que a ojos del usuario se verá reflejada con un signo en la parte inferior del mensaje.

Luego cuando el usuario reciba el mensaje, guardará dicho mensaje en la base de datos local de su dispositivo y se dispondrá a borrar el mensaje en Firebase (simulando una conexión punto a punto). De este modo ambos usuarios habrán establecido el envío y recibo de un mensaje.

Ahora bien, ¿cómo se le notifica al usuario receptor de que le ha llegado un mensaje nuevo?

Esta funcionalidad se ha implementado usando Services en Android, este componente se podría explicar como una Actividad pero sin la parte visual que se ejecuta en segundo plano.

En nuestro caso, cuando el usuario inicia la aplicación y llega a **MainActivity** crea un Service donde dicho servicio lanza una serie de listeners para cada una de las conversación, a grandes rasgos esto es que la aplicación se pone a escuchar para cada conversación si tiene mensajes nuevos o no. En caso de tener mensajes nuevos, los guardamos en local. Existe una peculiaridad y es que si el usuario no está en esos momentos dentro de la aplicación, recibirá una notificación en forma de pop-up tal como funciona otras aplicaciones de mensajería (WhatsApp, Telegram...)

### 6.3. ¿Cómo se le notifica al resto de usuarios que hemos actualizado nuestro usuario (foto, nombre de usuario, estado)

Cuando se inicia sesión en la aplicación, cuando el usuario accede a **MainActivity.kt** la aplicación se pone a "escuchar" la lista de usuarios por si se produce algún cambio.

```

1 // listen for user updates such as Profile photo, status, username
2 fun launchUserListener() {
3     database.child(FIREBASE_USER_PATH)
4         .addValueEventListener( object : ValueEventListener {
5             override fun onCancelled(p0: DatabaseError) {}
6             override fun onDataChange(dataSnapshot: DataSnapshot) {
7                 dataSnapshot.children.forEach { userSnapshot ->
8                     val user = userSnapshot.getValue(User :: class.java)
9                     user?.let {
10                         roomDatabase.addUser(user)
11                         updateConversation(user)
12                     }
13                 }
14             }
15         })
16 }
```

**Listing 60:** Fragmento de código para escuchar cambios en los usuarios de la aplicación

En el fragmento de código **60**, se observa que cuando un usuario modifica ya sea su foto de perfil, estado o nombre de usuario, dichos cambios se notifican al resto de usuarios, por lo que en cada dispositivo de estos usuarios se actualiza el valor del usuario que ha cambiado su valor. Al actualizar el usuario, se actualiza la base de datos local de cada uno de estos usuarios e inmediatamente actualiza la lista de conversaciones.

### 6.4. ¿Cómo guardamos las imágenes en nuestra base de datos?

Por otra parte, hemos comentado brevemente el proceso de guardar imágenes en la base de datos, es verdad que Firebase cuenta con una opción de salvaguardar imágenes en formato JPG y PNG, pero dentro del proyecto por sencillez transformamos las imágenes en dichos formatos en un texto, este formato se denomina Base64<sup>4</sup> [11].

En relación a lo anterior, codificamos la imagen en formato Bitmap en un texto, el proceso para recuperar desde un texto codificado en base64 a la imagen original es trivial. Un ejemplo de ello se puede ver en la figura **41**.

<sup>4</sup>Base 64 es un sistema de numeración posicional que usa 64 como base. Es la mayor potencia que puede ser representada usando únicamente los caracteres imprimibles de ASCII. Esto ha propiciado su uso para codificación de correos electrónicos, PGP y otras aplicaciones. Todas las variantes famosas que se conocen con el nombre de Base64 usan el rango de caracteres A-Z, a-z y 0-9 en este orden para los primeros 62 dígitos, pero los símbolos escogidos para los últimos dos dígitos varían considerablemente de unas a otras. Otros métodos de codificación como UUEncode y las últimas versiones de binhex usan un conjunto diferente de 64 caracteres para representar 6 dígitos binarios, pero estos nunca son llamados Base64.

```
>>> import base64
>>> texto = "Prueba de codificación BASE64"
>>> textoCodificado = base64.encodestring(texto)
>>> textoCodificado
'UHJ1ZWJhIGRLIGNvZGlmaWNhY2nDs24gQkFTRTY0IQ=\n'
>>> base64.decodestring(textoCodificado)
'Prueba de codificación BASE64'
```

Figura 41: Ejemplo de código en Python para codificar y decodificar en base64

Tal como hemos explicado en secciones anteriores, usamos una librería de Android para visualizar las imágenes de manera optimizada, esta librería es Glide donde cuyas ventajas se mencionó en [2.5](#)

Asimismo, hemos extendido una función para los ImageView por lo que si pulsamos en la imagen de perfil ya sea en MainActivity, ChatActivity o UserSearcherActivity, se creará un diálogo mostrando dicha imagen.

## 7 Modelo de traducción local vs Cloud Translation

La principal razón del porqué es que la traducción en tiempo real no es gratis, el usuario debe pagar una cuota (véase figura 42), es cierto que nos facilita mucho la vida ya que nos olvidamos de descargar los modelos de traducción en nuestro dispositivo, con la consiguiente ocupación de memoria que conlleva, hemos hecho un breve cálculo sobre el dinero y debido a que no es un proyecto real, hemos preferido optar por la opción gratuita.

### Precios mensuales

#### API Translation: edición básica

Función	Puede optar al uso gratuito	Precio
Detección de idioma	✓	20 USD por millón de caracteres*
Traducción de texto (modelos generales de NMT)	✓	20 USD por millón de caracteres*
Traducción de texto (modelos generales de PBMT)	✓	20 USD por millón de caracteres*

**Figura 42:** Captura de pantalla de la tabla de precios de Google Cloud Platform

Para poder traducir texto usando las funcionalidades de GCP (Google Cloud Platform) se realiza de la siguiente manera.

```

1 Translation translation = translate.translate(
2   "Hola Mundo!",
3   Translate.TranslateOption.sourceLanguage("es"),
4   Translate.TranslateOption.targetLanguage("de"),
5   // Use "base" for standard edition, "nmt" for the premium model.
6   Translate.TranslateOption.model("nmt"));
7
8 System.out.printf("TranslatedText:\nText: %s\n", translation.
getTranslatedText());

```

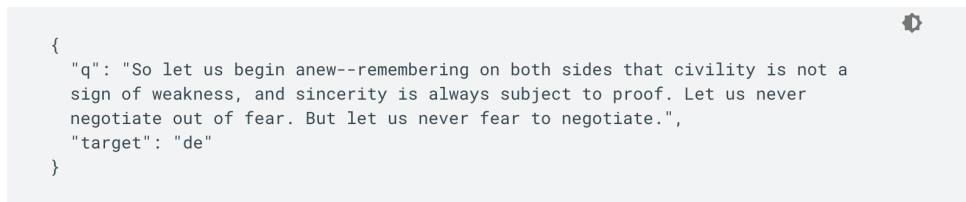
**Listing 61:** Fragmento de código para traducir un texto del español al alemán

Ahora bien, hemos extraído un par de ejemplo del link siguiente [https://cloud.google.com/translate/docs/basic/translating-text#translate\\_text\\_with\\_model-java](https://cloud.google.com/translate/docs/basic/translating-text#translate_text_with_model-java), estos ejemplos son los que se muestran en 43 y en 44

Cuando el usuario traduce un texto necesita cuatro parámetros:

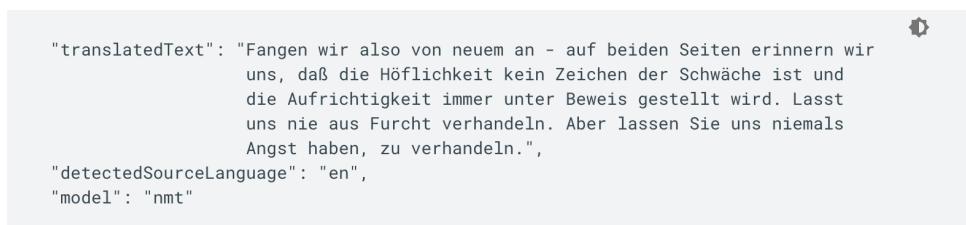
1. Texto fuente
2. Lenguaje fuente
3. Lenguaje destino
4. Modelo, que puede ser *base* o *nmt*

En dichos ejemplos podemos ver la calidad de traducción que se obtiene al usar el modelo de pago de Google, para un proyecto real sí hubiésemos escogido este modelo.



```
{  
  "q": "So let us begin anew--remembering on both sides that civility is not a  
  sign of weakness, and sincerity is always subject to proof. Let us never  
  negotiate out of fear. But let us never fear to negotiate.",  
  "target": "de"  
}
```

**Figura 43:** Texto que el usuario envía para ser traducido



```
{"translatedText": "Fangen wir also von neuem an - auf beiden Seiten erinnern wir  
uns, daß die Höflichkeit kein Zeichen der Schwäche ist und  
die Aufrichtigkeit immer unter Beweis gestellt wird. Lasst  
uns nie aus Furcht verhandeln. Aber lassen Sie uns niemals  
Angst haben, zu verhandeln.",  
"detectedSourceLanguage": "en",  
"model": "nmt"}
```

**Figura 44:** Traducción realizada a partir del texto de la figura 43

## 8 Trabajo futuro

---

En este apartado hablaremos de las funcionalidades que no hemos implementado ya sea por falta de tiempo o debido a que está fuera del alcance de los objetivos del proyecto.

Para comenzar enumeraremos la lista de funcionalidades que se podrían añadir en un futuro a la aplicación:

1. **Envío de audios:** esta funcionalidad puede parecer sencillo de implementar a priori ya que se debe tener en cuenta bastantes aspectos como por ejemplo, a primera vista se necesitaría investigar qué formatos de audio existen, posteriormente salvaguardarlos en la base de datos y realizar todas las acciones oportunas como pausar, resumir el audio, por mencionar unas cuantas
2. **Envío de documentos:** otra funcionalidad interesante de implementar sería el envío de documentos ya sea en formato PDF, excel, docx por nombrar algunas
3. **Envío de canciones:** se podría usar la API de Spotify para implementar dicha funcionalidad
4. **Envío de código:** desde el punto de vista de un programador sería interesante la capacidad de enviar bloques de código, indentados según el lenguaje de programación ya que ninguna aplicación de mensajería tiene esta capacidad. Si bien es un trabajo que conllevaría tiempo, también es verdad que esta funcionalidad puede ser de mucha utilidad para los programadores
5. **Grupos de mensajería:** esta implementación se ha dejado de lado debido a que nuestro objetivo es crear una aplicación donde podamos traducir en tiempo real los mensajes enviados en cualquier idioma y que el usuario lo perciba en el idioma deseados.
6. **Implementación de llamadas y videollamadas**
7. **Localización en directo (*Live location*):** de momento se ha implementado solamente el envío de la localización de manera estática.

Dicho todo esto, todas estas ideas de implementación podrían mejorar considerablemente la experiencia de usuario al usar la aplicación que hemos creado.

## 9 Conclusiones

---

Para finalizar, llegamos a las conclusiones de nuestro trabajo fin de máster. En el aspecto de qué desarrollo es más conveniente a la hora de diseñar una aplicación de mensajería podemos decir que el desarrollo Nativo es, en este caso mejor, debido a que el objetivo de nuestro trabajo era desarrollar la aplicación para un sistema operativo.

Por otra parte si fuésemos una *start-up* quizás entonces la decisión hubiese sido diferente. Asimismo dentro del desarrollo nativo existen dos lenguajes principales, tal como explicamos en la sección 2.2, y escogimos Kotlin como lenguaje principal debido a sus notables ventajas (Sección 3) que nos ofrecían de cara al desarrollo facilitándonos notablemente el trabajo.

En cuanto al resto de tecnologías, creemos que hemos realizado las decisiones correctas al usar el nuevo patrón de diseño de Google (4) y las librerías más comunes que se usan en Android (Glide, Retrofit, Room...)

La segunda parte que queríamos concluir es el tema de la calidad de las traducciones. Hemos observado a lo largo de este proyecto que aunque dichas traducciones no son perfectas, se puede decir que hacen su función, es decir, es posible mantener una conversación entre dos personas con idiomas totalmente diferentes. Además hay que decir que en el proyecto se ha usado la librería de traducción gratuita, por lo que las traducciones son relativamente peores en comparación a la librería de pago (discutido en 7).

No obstante, hoy en día aunque se están mejorando las traducciones entre distintos idiomas a pasos agigantados, pero todavía queda un largo camino por recorrer. Cabe decir, que es posible realizar aplicaciones de mensajería con dicha funcionalidad y una prueba de ello es el trabajo que hemos realizado con un tiempo de entrega establecido y con los recursos ínfimos con los que contábamos. Es por ello, y me mantengo bastante positivo, que en un futuro cercano las grandes empresas comiencen a implementar la traducción en tiempo real en aplicaciones de este estilo.

En cuanto a los objetivos que nos hemos propuesto al principio del documento, creo muy positivamente que se han cumplido con los objetivos propuestos:

1. Traducción en tiempo real
2. Creación de una pantalla de iniciar sesión
3. Creación de una pantalla donde modificar datos personales (foto de perfil, estado)
4. Funcionalidad extra tales como envío de emoticonos, GIFs, imágenes, localización...

Por todo lo mencionado anteriormente, y también, lo aprendido durante el tránscurso del trabajo (Kotlin, MVVM, Firebase...) creo positivamente que este trabajo se puede considerar como un éxito.

# Bibliography

---

- [1] WhatsApp dice tener 2 000 millones de usuarios en el mundo  
<https://www.elcomercio.com/tendencias/whatsapp-usuarios-activos-mundo-criptacion.html>
- [2] Hybrid VS Native App: Which one to choose for your business?  
<https://medium.com/existek/hybrid-vs-native-app-which-one-to-choose-for-your-business-e51542554078>
- [3] Kotlin vs. Java: Which One You Should Choose for Your Next Android App  
<https://medium.com/netguru/kotlin-vs-java-which-one-you-should-choose-for-your-next-android-app-1d08c4d3a22f>
- [4] Butter Knife by Jake Wharton. <https://github.com/JakeWharton/butterknife>
- [5] Glide vs Picasso - Multidots <https://medium.com/@multidots/glide-vs-picasso-930eed42b81d>
- [6] Using Room Database | Android Jetpack <https://medium.com/mindorks/using-room-database-android-jetpack-675a89a0e942>
- [7] Coroutines on Android (Part I): Getting the background  
<https://medium.com/androiddevelopers/coroutines-on-android-part-i-getting-the-background-3e0e54d20bb>
- [8] MVVM (Model View ViewModel) + Kotlin + Google Jetpack  
<https://medium.com/@er.ankitbisht/mvvm-model-view-viewmodel-kotlin-google-jetpack-f02ec7754854>
- [9] Software Architecture Guide <https://www.martinfowler.com/architecture/>
- [10] Qué es la Unix epoch (época Unix) <https://desarrolloweb.com/faq/unix-epoch-epoca>
- [11] Wikipedia base64 encoding <https://es.wikipedia.org/wiki/Base64>
- [12] Google Cloud Platform - Translate\_text\_with\_model-java  
[https://cloud.google.com/translate/docs/basic/translating-text#translate\\_text\\_with\\_model-java](https://cloud.google.com/translate/docs/basic/translating-text#translate_text_with_model-java)