UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

FIB

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

Fecha: Abril 2020

Índice

Listado de imágenes

1 Introducción

Hoy por hoy, una de las principales razones por las que se usan 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 se trata en este proyecto, la traducción automática de mensajes en tiempo real.

Esta funcionalidad no sería de suma importancia si toda la población mundial hablase 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 se ha 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, se ha propuesto realizar una aplicación de Android desde cero, donde el usuario será capaz de comunicar con cualquier usuario en uno de los idiomas permitidos (castellano, catalán, francés, inglés, alemán...), y hacerlo de forma transparente como si se tratase de una funcionalidad más, al igual que si se enviase una imagen, por poner un ejemplo.

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

Se comenzará explicando cuáles son las funcionalidades básicas y son:

1. **Conversaciones uno a uno**: como era de esperar, se ha de poder enviar mensajes al otro usuario de manera directa y al instante.

2. **Perfil de usuario**: con esta funcionalidad, el usuario podrá modificar su imagen de perfil, estado y el nombre de usuario.

3. **Pantalla de registro**: el usuario antes de comenzar a usar la aplicación, tendrá que tener una cuenta de usuario con la que se registrará.

4. **Envío de emoticonos**.

5. **Envío de archivos multimedia**: estos archivos son por ejemplo las imágenes o GIFs.

6. **Envío de localización**: se implementará también el hecho de poder enviar la localización actual o cualquier dirección deseada.

Cabe destacar que se han dejado otras ciertas funcionalidades fuera del alcance del proyecto como son la encriptación de los mensajes, las conversaciones grupales debido a una mayor complejidad y exclusión de ciertos tipos de documentos (PDF, formatos de audio).

Por otra parte, los objetivos a grandes rasgos son:

1. **Diseño de la arquitectura**: dentro de cualquier proyecto informático, se ha de diseñar y tener cierta estructura a la hora de realizar el proyecto. En el contexto del desarrollo Android, son dos las arquitecturas que se usan en la industria y son MVC y MVVM, que se explicarán en la sección 4.

2. **Diseño de la Interfaz de Usuario**: para que una aplicación móvil sea usada de manera correcta, se ha de crear una interfaz intuitiva y amigable para que el usuario esté cómodo en todo momento. Esto se explicará en la sección 7.

3. **Traducción de mensajes en tiempo real**: se intenta conseguir que entre dos usuarios, por poner un ejemplo, un hispanohablante y un usuario que hable mandarín, el usuario hispanohablante vea en su dispositivo los mensajes en mandarín traducidos al español sin saber qué lengua habla el otro usuario. En la sección 6 se explicará el proceso de implementación.

1.3. Estructura del trabajo

Para concluir la introducción, se ha 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, se explicará la arquitectura del proyecto que se ha seguido para organizar toda la lógica de la aplicación, seguida del porqué hemos seleccionado el modelo de traducción gratis sobre el de pago.

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

Finalmente, se hablará de qué tareas se podrían realizar en un futuro y un resumen final del proyecto.

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 se explicarán 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 [1].

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.

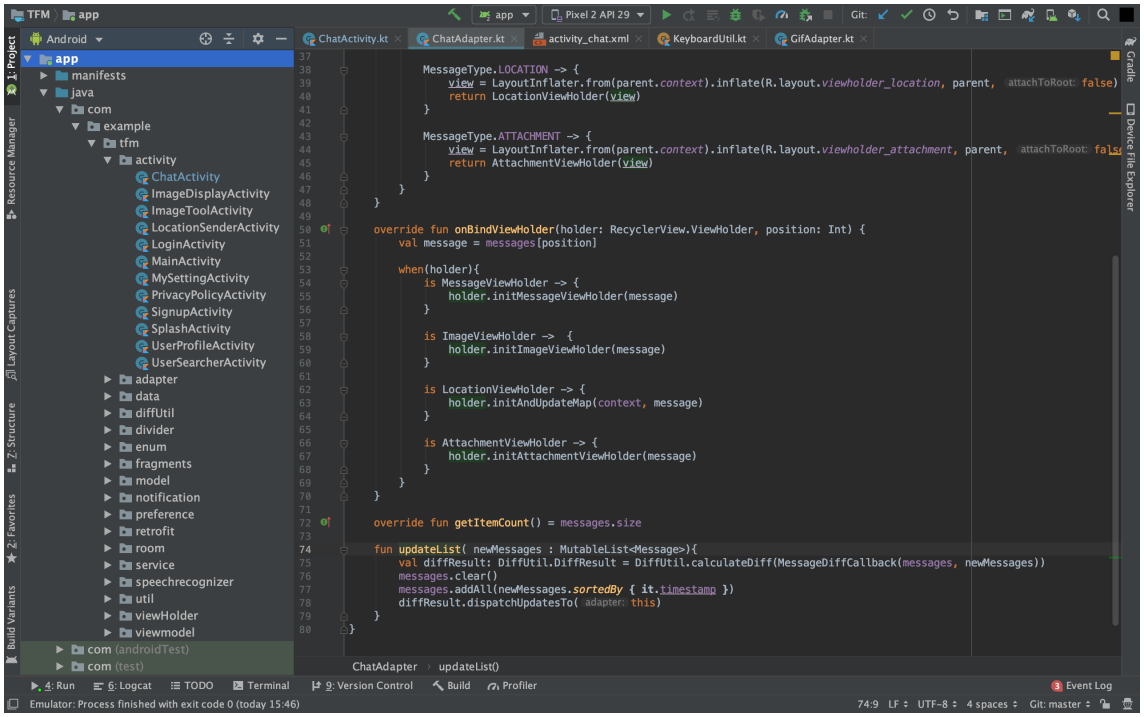


Figura 1: Captura de pantalla del Android Studio. Se puede observar que en la parte izquierda se encuentra la estructura de ficheros. En la parte central y derecha se muestra el contenido de los ficheros y donde el usuario programará.

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.

[1] 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 "I nnovación 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.

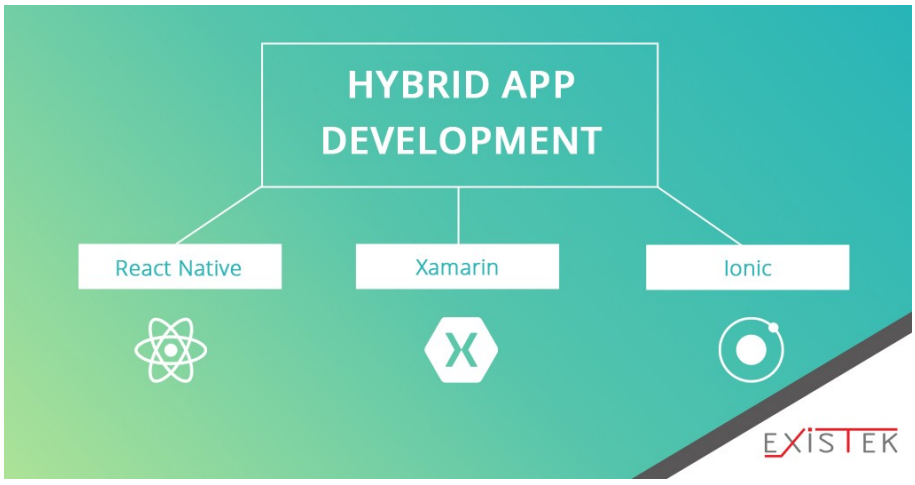


Figura 3: Desarrollo híbrido de aplicaciones móviles. Para desarrollo de aplicaciones híbridas, los más usados son React Native, Xamarin e Ionic. Cabe destacar que un nuevo lenguaje está cogiendo fuerza en cuanto a desarrollo Native se refiere denominado Flutter.

Entonces, ¿cuándo es mejor el desarrollo híbrido? Si se observa en la figura 4, se puede observar las diferencias entre ambas formas de crear aplicaciones móviles. Existen proyectos donde se intentan obtener el máximo número de usuarios posibles, por lo que al ser desarrollados en híbrido, simplemente con desarrollarlos una vez, podrán ser

[2]Las imágenes 2, 3, 4 se han extraído de [2]

utilizado en 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.

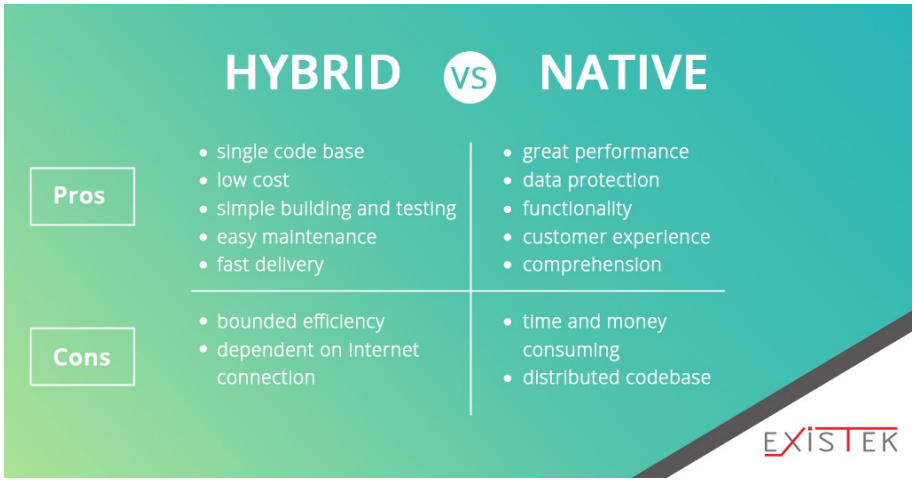


Figura 4: Tabla de comparación entre desarrollo nativo e híbrido

A modo de resumen, a continuación se listarán 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.

A continuación se mostrará un par de ejemplos de aplicaciones desarrolladas de manera nativa e híbrida. En la 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 se observa la figura 6, ambas aplicaciones tienen el mismo diseño y la funcionalidad es idéntica independientemente del sistema operativo que se usa en el dispositivo. Es por ello, que la decisión de elegir el tipo de desarrollo dependerá única y exclusivamente de los recursos del que se disponen.

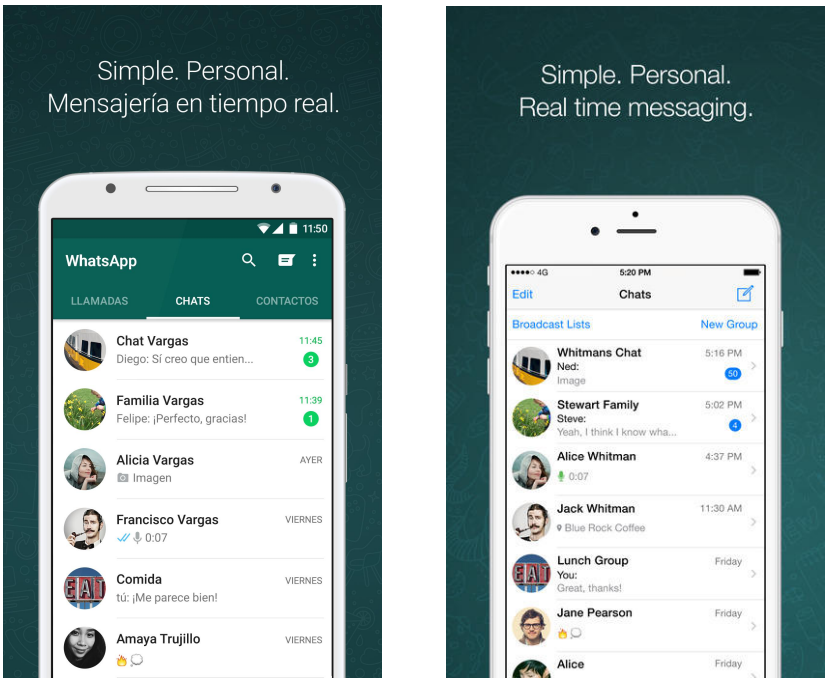


Figura 5: A su izquierda se encuentra la aplicación de Whatsapp desarrollada en Android, y a la derecha en iOS.

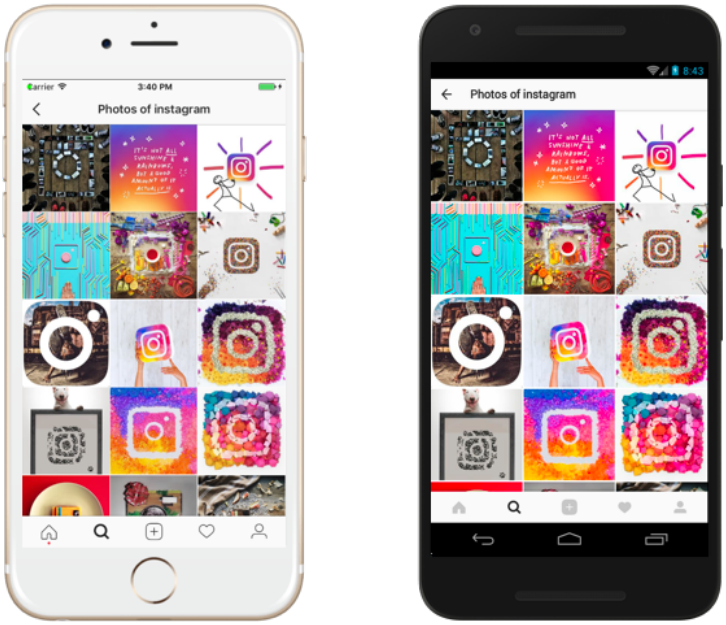


Figura 6: Aplicación de Instagram en cada una de sus versiones. A la izquierda se encuentra en su versión iOS, y a la derecha en su versión Android.

2.2. Kotlin vs Java

Kotlin 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 se ejecuta 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.

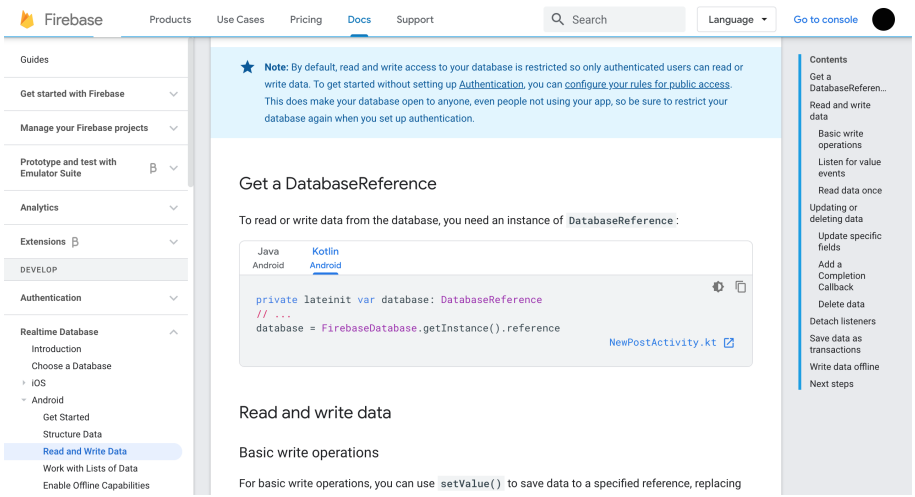


Figura 7: Opción de Java y Kotlin en la documentación de *Google* Firebase.

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. Un claro ejemplo es la creación de clases POJO (acrónimo de Plain Old Java Object) y se refiere a una clase que no extiende ni implementa ninguna función especial, es decir, es la clase más trivial. 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.

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

Listing 1: Clase POJO implementada en Kotlin

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

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 reducida en comparación a Java ya que este último cuenta con varios años de experiencia y existe información de todo tipo.

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 usados según Github.
3. Suele necesitar más líneas de código para realizar la misma función que en otros lenguajes de programación, necesitando «boilerplate code».

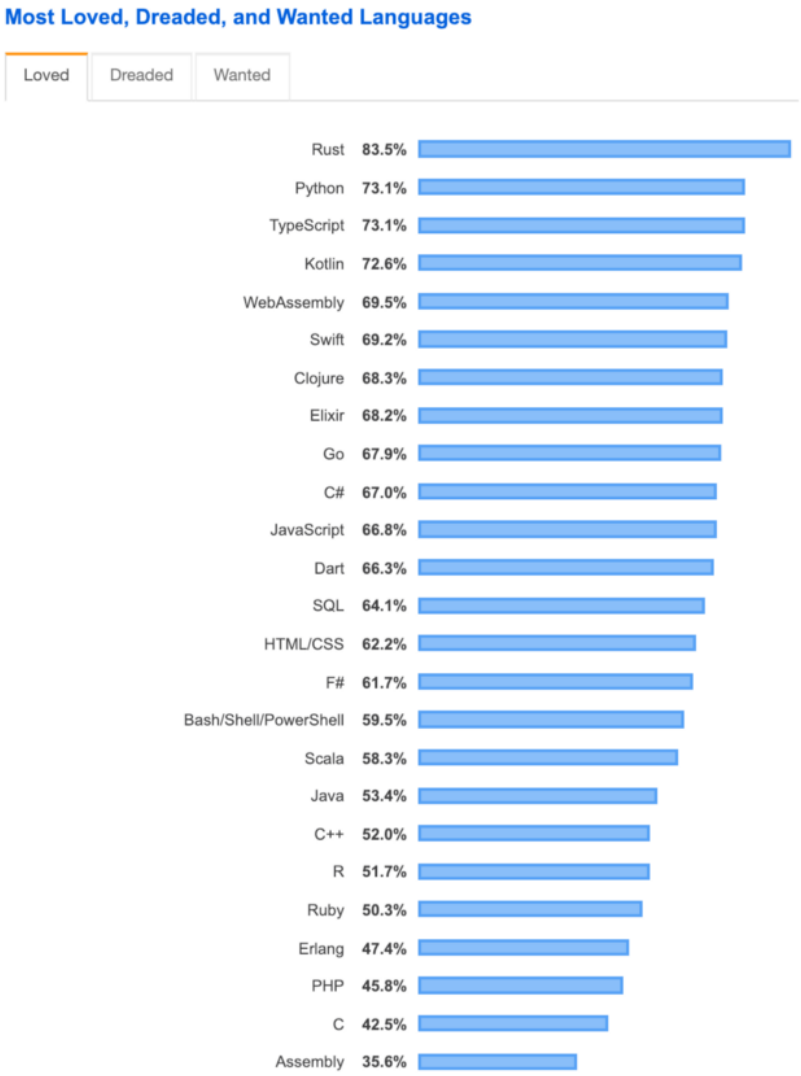


Figura 8: Encuesta de Stackoverflow sobre lenguajes más queridos en 2019.

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, 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 ofrece un servicio en la nube desarrollada por *Firebase Inc. (2011)* y posteriormente adquirida en 2014 por *Google*, ayuda a desarrollar aplicaciones web y móvil de alta calidad.

Dentro de Firebase se puede encontrar un sinfín de herramientas o servicios muy fáciles de usar a la hora de implementar una 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.

Una de las características principales de Firebase es que es en tiempo real, tal como se ha mencionado líneas más arriba. En nuestro caso, en el contexto de una aplicación de mensajería, cuando un usuario envía un mensaje se inserta este mensaje en la base de datos, notificándole al otro usuario que tiene un mensaje nuevo instantáneamente. El proceso más detallado se explicará en la sección 6.

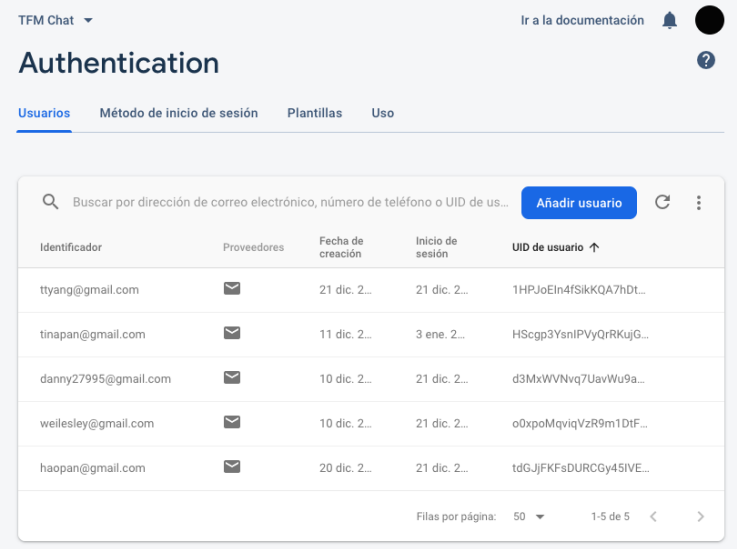


Figura 9: Captura de pantalla de Firebase Authentication.

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 el proyecto, solo se aceptan correos que tengan el dominio «@gmail.com» por sencillez. Más adelante, en la sección 7 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 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 del método *findViewById(R.id.nombre_widget)*, hoy en día existe varias opciones para evitar toda esta cantidad de «boilerplate code» como Dagger 2, Butter Knife y View Binding de Android.

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

Por supuesto, se ha de 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.

2.5. Picasso vs Glide

Cuando se tratan imágenes en un proyecto Android, son dos las librerías que más se usan y son Picasso y Glide. Dependiendo del uso y los recursos de los que se disponen, se usa uno u otro.

A continuación se explicarán las diferencias entre estas dos librerías. Ambas librerías son casi idénticas en cuanto a funciones se refieren, que permiten 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 también son prácticamente idénticas, por lo que a priori, si estás familiarizado con una de estas dos librerías, el uso de la otra es inmediato.

```
class ExampleActivity extends Activity {
  @BindView(R.id.user) EditText username;
  @BindView(R.id.pass) EditText password;

  @BindString(R.string.login_error) String loginErrorMessage;

  @OnClick(R.id.submit) void submit() {
    // TODO call server...
  }

  @Override public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.simple_activity);
    ButterKnife.bind(this);
    // TODO Use fields...
  }
}
```

Listing 3: Ejemplo de utilización de Butter Knife

```
Picasso.with(myFragment)
    .load(url)
    .centerCrop()
    .placeholder(R.drawable.loading_spinner)
    .into(myImageView);
```

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

```
Glide.with(myFragment)
    .load(url)
    .centerCrop()
    .placeholder(R.drawable.loading_spinner)
    .crossFade()
    .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).

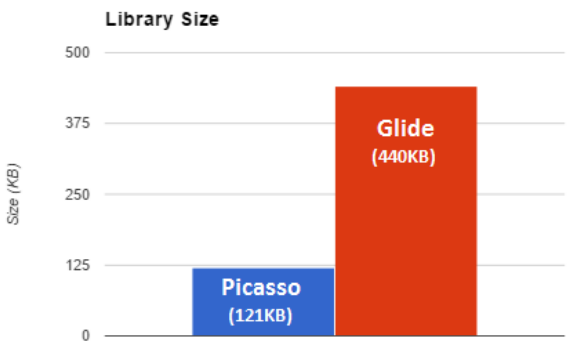


Figura 11: Tamaño de Glide y Picasso.

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.

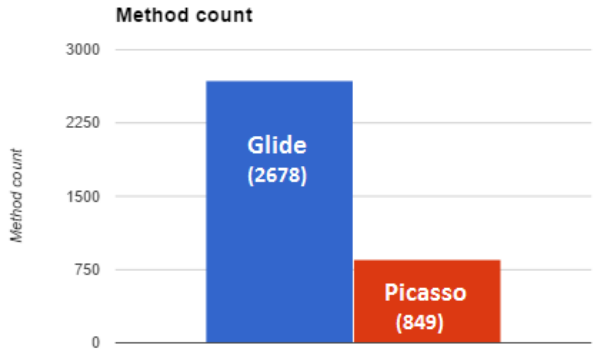


Figura 12: Métodos de Glide y Picasso.

En cuanto al uso de memoria, se puede observar 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.

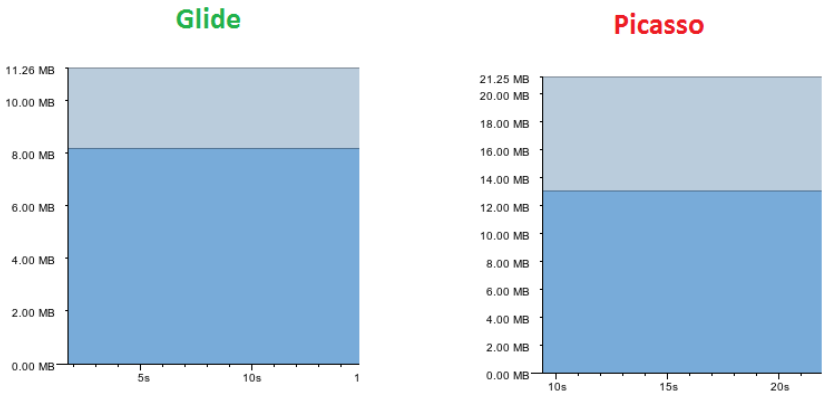


Figura 13: Uso de memoria de Glide y Picasso.

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, se ha 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. Un breve repaso de cuáles 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 praxis de programación, reducen el código «boilerplate» y simplifican 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 abstrae 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, de la 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.

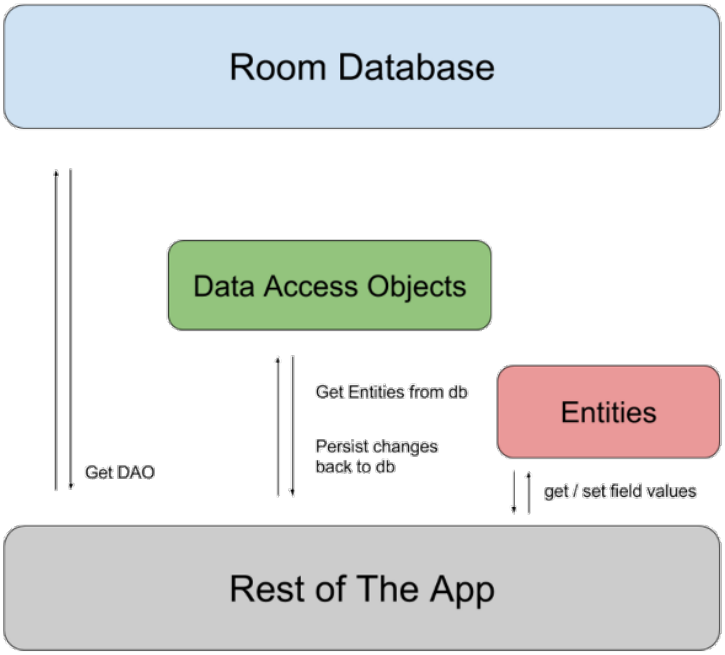


Figura 14: Componentes de Room.

2.7. Retrofit

En esta sección hablaremos de Retrofit, se trata de una librería de Android que permite abstraer 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.

```
{
    "name":"John",
    "age":30,
    "cars":[ "Ford", "BMW", "Fiat" ]
}
```

Listing 6: Ejemplo de un fichero JSON

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

```
implementation 'com.squareup.retrofit2:retrofit:(latest version)'
implementation 'com.Google.code.gson:gson:(latest version)'
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. Corrutinas

En este apartado de tecnologías utilizadas se hablará de las corrutinas. Las corrutinas introducen una nueva forma de concurrencia que puede ser usada en Android para simplificar el código asíncrono, a pesar de ser nuevo en Kotlin en su versión 1.3, este concepto de corrutinas 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 Kotlin, nos podemos encontrar las corrutinas 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 corrutinas 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 se bloquea 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 corrutinas.

Para implementar el Main-Safety mencionado anteriormente, Kotlin provee al desarrollador los llamados dispatchers, también se le conoce más vagamente como contextos de corrutinas. 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.

Se ha comentado que las corrutinas sirven para realizar peticiones a la red y acceso a la base de datos fuera del hilo principal. También se ha 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 corrutinas 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().

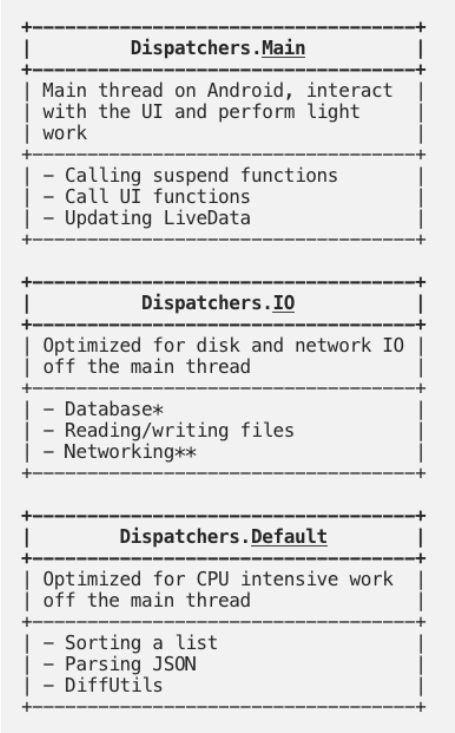


Figura 15: Diferencias entre los tres tipos de Dispatchers de las corrutinas.

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 se realizan en el proyecto y mantener un control de versiones. En el proyecto se ha utilizado *Github*. Existen otras herramientas como *GitLab* o *BitBucket*.

En la figura 16 se observa una captura de 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.

```
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 corrutinas y el uso de Dispatchers

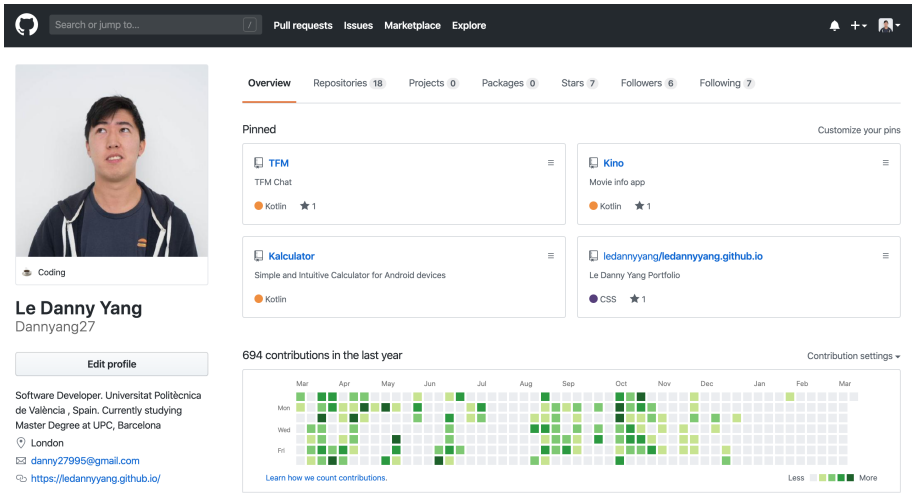


Figura 16: Captura de pantalla de Github.

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

3 Kotlin

En esta sección se ampliará lo que ya se ha explicado acerca de Kotlin en apartados anteriores. También se realizará 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 se conoce Kotlin previamente, se puede ir directamente a la sección 4.2.

3.1. Clases

Una de las principales diferencias entre Kotlin y Java a la hora de crear clases, es en las clases POJO. Poniendo el mismo ejemplo que en el apartado 2.2, 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 implementarla. 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 destacables cuando se programa en Kotlin, es que los ';' son innecesarios al igual que en otros lenguaje de programación como Javascript y Python, cuando el programador se acostumbra 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, 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 se ha mencionado todo es un objeto y es debido a esto, que no existe la conversión automática de tipos. Esto se 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 el proyecto se quiere 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 se ha presentado la librería Glide para el tratamiento de imágenes. Ahora bien, es posible que los desarrolladores, extiendan la clase ImageView y añadan una nueva función tal como se ve 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 declaran 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

Posiblemente dos de las características de Kotlin más interesantes si se viene de desarrollar aplicaciones en Java.

En Kotlin, cuando se quiere declarar que un objeto puede llegar a tener el valor *null*, se usa la cláusula '?' tal como se observa 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 se puede trabajar con objetos *null* sin realizar comprobaciones previas, en el fragmento 16 se muestra un pequeño código que no compila.

```
1    val a: Int? = null
2    a.toLong() // 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 se observa 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.toLong()
5    }
```

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

En este siguiente caso, se ejecutará la llamada *toLong()* 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?.toLong()
```

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*.

```
List<Gasto> gastos= new ArrayList<Gasto>();
gastos.add(new Gasto("A",80));
gastos.add(new Gasto("B",50));
gastos.add(new Gasto("C",70));
gastos.add(new Gasto("D",95));

double resultado=gastos.stream()
    .mapToDouble(gasto->gasto.getImporte()*1.21)
    .filter(gasto->gasto<100)
    .sum();

System.out.println(resultado);
```

Listing 21: Ejemplo de Java Streams

```
val a: Int? = null
...
val value: Long = a?.toLong() ?: 0L
```

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

Se puede lanzar un error cuando este caso ocurra tal como se ve en el fragmento 20:

```
val a: Int? = null
...
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 los desarrolladores saben con total certeza que en un momento dado, esa variable no es *null*, por lo que se puede 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 se incorpora en el uso de las colecciones o listas, se obtiene una potencia expresiva mayor. Con todo esto, se quiere decir que con la programación funcional en vez de decir cómo se implementa un cambio, se dice qué cambios se quiere obtener.

Esto se verá más fácilmente con el siguiente ejemplo en los fragmentos 21 y 22 que muestra los ejemplo en Java y Kotlin, respectivamente. [3]

[3] Ejemplo extraído de https://www.arquitecturajava.com/programacion-funcional-java-8-streams/

```
val gastos= mutableListOf<Gasto>()
gastos.add(Gasto("A", 80))
gastos.add(Gasto("B", 50))
gastos.add(Gasto("C", 70))
gastos.add(Gasto("D", 95))

val resultado = gastos.map { it.importe * 1.21 }.filter { it < 100 }.sum()
print(resultado)
```

Listing 22: Ejemplo de streams en Kotlin

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.

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

```
val list = listOf(1,2,3,4,5,6)

list.any { it % 2 == 0 } //comprueba si existe algun valor que sea par
list.all { it % 2 == 0 } //comprueba si todos los valores son numeros pares
list.count { it % 2 == 0 } //cuenta cuantos numeros son pares en la lista
list.max() //devuelve el numero maximo de la lista
list.min() //devuelve el numero minimo de la lista
list.filter { it % 2 == 0 } //devuelve los elementos que cumplan el
    predicado
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.

```
class Singleton{
    private static Singleton INSTANCE = null;

    public static Singleton getInstance(){
        if( INSTANCE == null ){
            INSTANCE = new Singleton();
        }

        return INSTANCE;
    }
}
```

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, se puede 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 se está desarrollando cualquier aplicación.

4 Arquitectura

En esta sección se hablará de dos patrones de arquitectura, MVC (Model-View-Controller) y MVVM (Model-View-ViewModel), cuando se habla de arquitectura en el contexto de la programación 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. Esta es la que nosotros hemos escogido para el desarrollo de este proyecto.

Pero antes de comenzar explicando el porqué de esta elección, se explicará brevemente la arquitectura MVC.

4.1. MVC

La arquitectura MVC separa los datos y la lógica de negocio. Para ello, MVC propone la construcción de tres componentes que son el modelo, la vista y el controlador, esto se puede observar en la figura 17. A continuación se explicará cada módulo por separado:

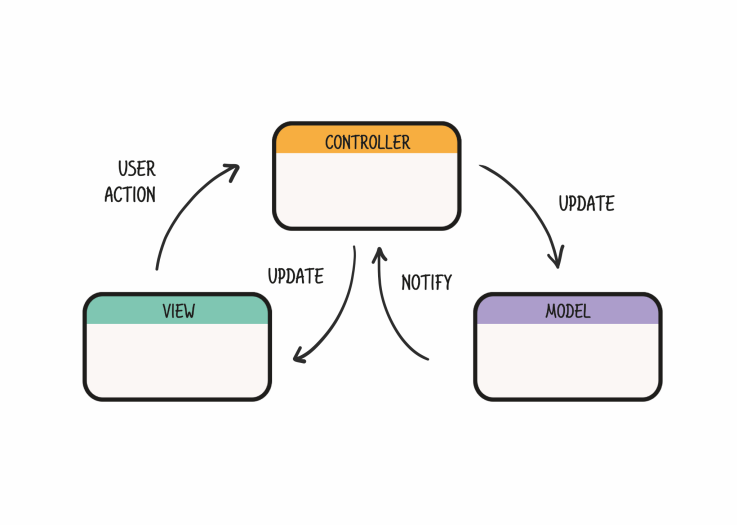


Figura 17: Esquema de la arquitectura MVC.

1. **Modelo**: caracterizada por un conjunto de clases que modelan objetos de la vida real que describen la lógica de negocio.
2. **Vista**: la vista representa los componentes visuales de la aplicación que mostrarán la información pertinente, es decir, aquello con el que el usuario final interactuará.
3. **Controlador**: es el encargado de procesar los eventos de la vista y procesar dicha información recogida con la ayuda de modelo. Una vez la información se haya procesado se actualizaría la vista.

4.2. Model View ViewModel

Esta nueva arquitectura se compone principalmente de tres componentes que se pueden observar en la figura 18:

1. **Model**: en este componente, es donde se escribe toda la lógica de negocio, es decir, si se realiza 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.

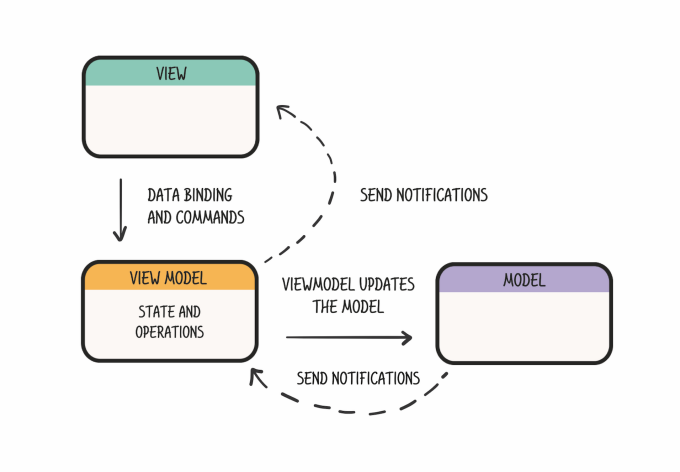


Figura 18: Esquema de la arquitectura MVVM.

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 esta 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 4.3), dichos componentes son *Activity Lifecycle aware* y este conlleva que en la práctica provoquen menos errores y *memory leaks* una vez se ejecuta la aplicación.

4.3. 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.

4.4. Ejemplo de MVVM en el proyecto

En esta sección se mostrará cómo funciona este patrón de diseño en la práctica. Lo que se ha conseguido es lo que se ve en la figura 19, 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 se obtenga el dato. Todo esto, se realiza de manera asíncrona sin bloquear el hilo principal. De esta manera, también se mejora la experiencia de usuario.

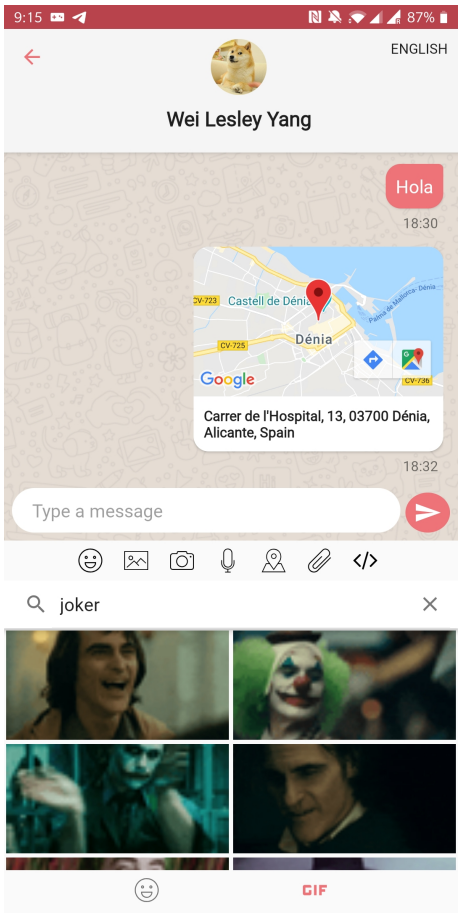


Figura 19: Interfaz del Chat Activity cuando el usuario quiere seleccionar un GIF.

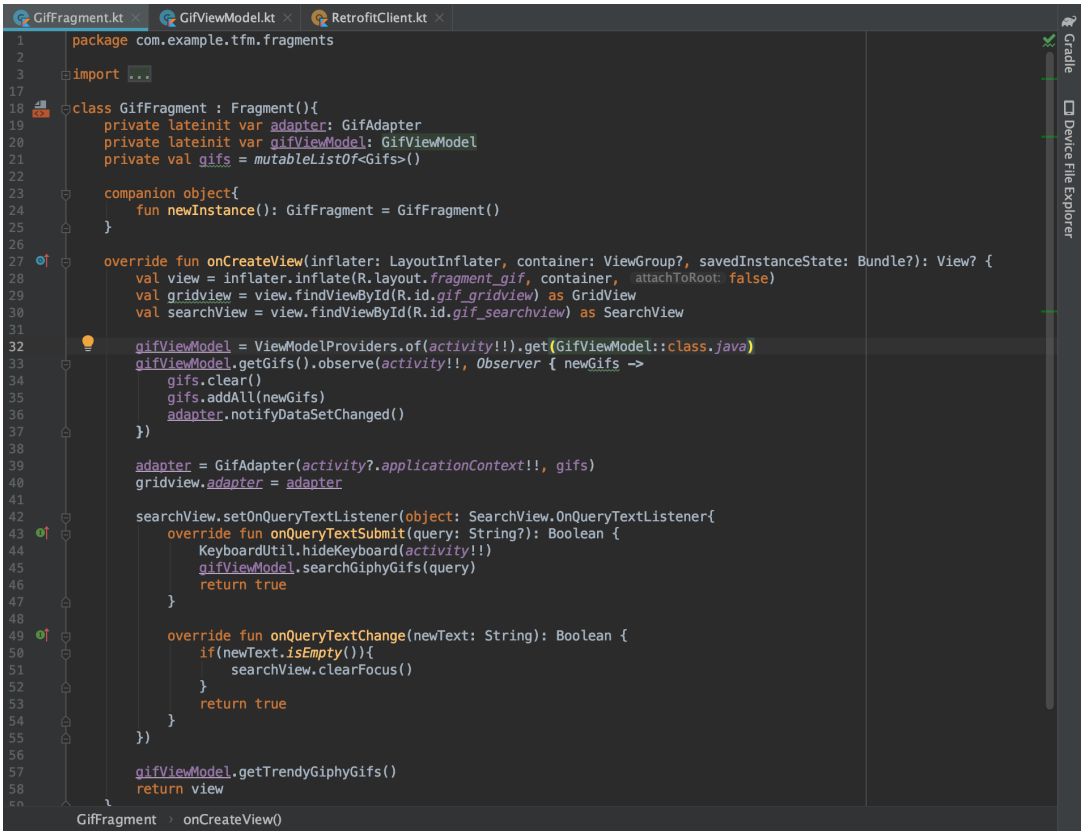
Comenzando con la figura 20, en dicha figura se puede observar que tiene una referencia a la clase *ViewModel*. Luego más adelante en la línea 32 se instancia la clase y se crea también el observable. Cualquier pequeño cambio en la lista de GIFs actualizaría la vista.

Ahora bien, la Figura 21 hace referencia a la clase *ViewModel* encargada del tratamiento de GIFs. Se puede observar que existe un objeto de tipo LiveData, y dos funciones que se encargan de llamar los métodos de nuestro *Model*, getTrendyGiphyGifs() y searchGiphyGifs(query: String?). Una vez se obtienen los datos a partir de estos métodos, se notificaría a la vista de que dicha lista se ha actualizado con el método postValue() (línea 20 y 28).

Por último, se hablará del *Model*, tal como se había mencionado en el *ViewModel*, desde dicha clase se llaman los métodos definidos en el *Model*.

En la figura 22, por ejemplo en el método getSearchGifFromGiphy() se realiza una llamada al WebService de *Giphy* que es el encargado de proporcionar los GIFs que nos interesan para nuestra aplicación.

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




```kotlin
package com.example.tfm.fragments

import ...

class GifFragment : Fragment(){
    private lateinit var adapter: GifAdapter
    private lateinit var gifViewModel: GifViewModel
    private val gifs = mutableListOf<Gifs>()

    companion object{
        fun newInstance(): GifFragment = GifFragment()
    }

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {
        val view = inflater.inflate(R.layout.fragment_gif, container,  attachToRoot: false)
        val gridview = view.findViewById(R.id.gif_gridview) as GridView
        val searchView = view.findViewById(R.id.gif_searchview) as SearchView

        gifViewModel = ViewModelProviders.of(activity!!).get(GifViewModel::class.java)
        gifViewModel.getGifs().observe(activity!!, Observer { newGifs ->
            gifs.clear()
            gifs.addAll(newGifs)
            adapter.notifyDataSetChanged()
        })

        adapter = GifAdapter(activity?.applicationContext!!, gifs)
        gridview.adapter = adapter

        searchView.setOnQueryTextListener(object: SearchView.OnQueryTextListener{
            override fun onQueryTextSubmit(query: String?): Boolean {
                KeyboardUtil.hideKeyboard(activity!!)
                gifViewModel.searchGiphyGifs(query)
                return true
            }

            override fun onQueryTextChange(newText: String): Boolean {
                if(newText.isEmpty()){
                    searchView.clearFocus()
                }
                return true
            }
        })

        gifViewModel.getTrendyGiphyGifs()
        return view
    }
```

Figura 20: Fragmento de código relacionado con la View.

de los distintos componentes, haciendo posible que si en un momento futuro queramos añadir nuevas funcionalidades, el cambio en nuestro código sería mínimo.

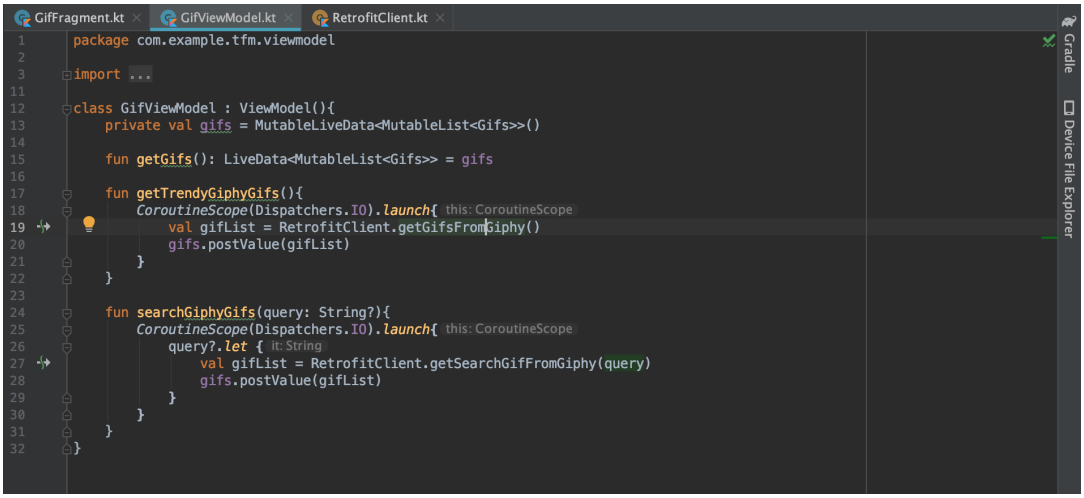


Figura 21: Fragmento de código relacionado con el ViewModel.

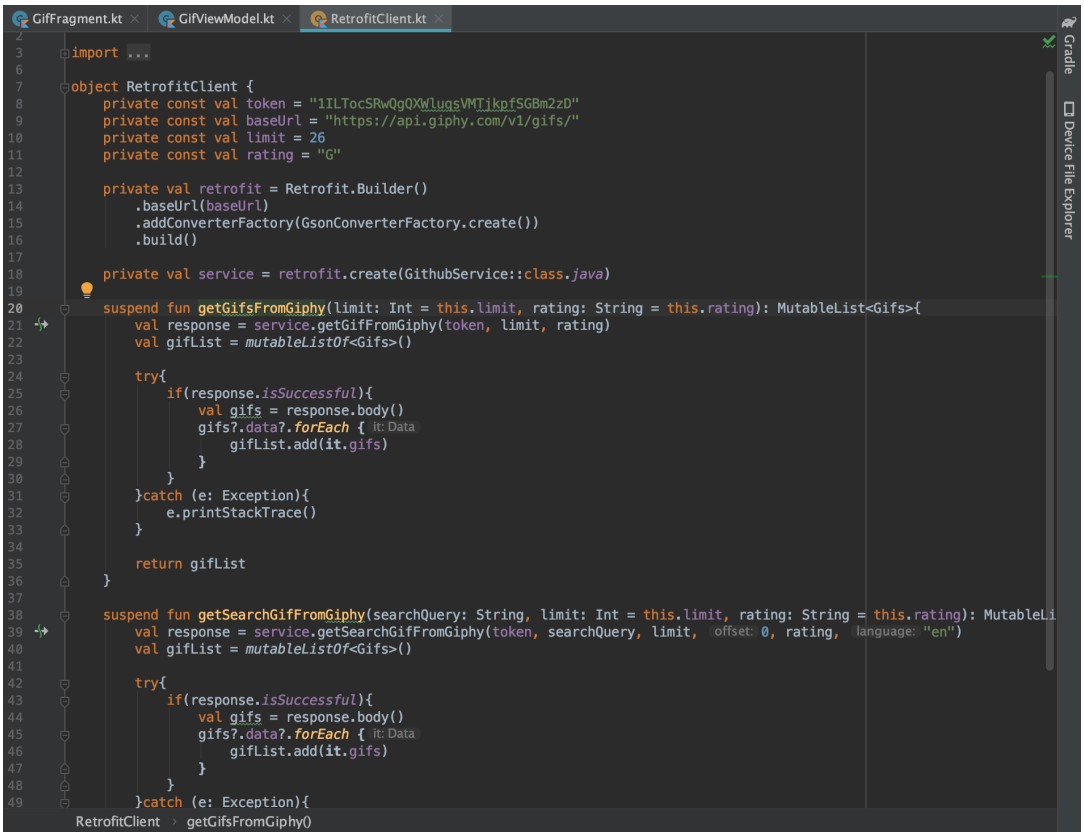


Figura 22: Fragmento de código relacionado con el Model.

5 Modelo de traducción local vs Cloud Translation

En esta sección se realizará una comparación entre dos modelos de traducción, el modelo gratis y el modelo de pago.

La principal razón del porqué se ha escogido el modelo gratis es que la traducción en tiempo real no es gratis y el desarrollador debe pagar una cuota por mensaje traducido (véase figura 23), es cierto que facilita mucho la vida ya que evita al usuario descargar los modelos de traducción en el dispositivo, con la consiguiente ocupación de memoria que conlleva, y debido a que no es un proyecto comercial, 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 23: 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 26: Fragmento de código para traducir un texto del español al alemán

Ahora bien, se ha extraído un par de ejemplo del link siguiente https://cloud.google.com/translate/docs/basic/translating-text#translate_text_with_model-java, estos ejemplos son los que se muestran en 24 y en 25.

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 se puede observar la calidad de traducción que se obtiene al usar el modelo de pago de Google, para un proyecto real comercial la opción de pago sería la opción correcta.

```
{
  "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 24: 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 25: Traducción realizada a partir del texto de la figura 24.

6 ¿Cómo realizamos ciertas funcionalidades?

En esta sección se explicará el porqué de ciertas funcionalidades de manera no muy detallada ya que para ello, se explicará en la sección 7, por ejemplo, la primera funcionalidad trata sobre cómo se traducen los mensajes para que los usuarios puedan comunicarse entre ellos de manera efectiva incluso sin hablar el mismo idioma.

6.1. ¿Cómo traducimos los mensajes?

Primeramente, cabe destacar que se utilizará el inglés como lenguaje intermedio, debido a que la aplicación soporta varias lenguas, es inviable tener guardado en el dispositivo varios modelos de traducción al idioma seleccionado, esto es, si se escogiese el castellano como lengua principal, habría que salvaguardar en el 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 este hecho es inviable y cabe recordar que cada modelo de traducción ocupa cerca de 30MB.

Es por ello que se necesita el uso de un idioma intermedio como puede ser el inglés (figura 26). De este modo, cada usuario tendría 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 se ahorra espacio de almacenamiento.

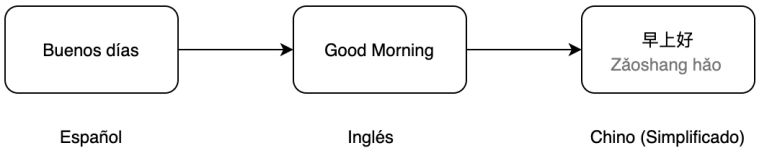


Figura 26: Esquema del proceso de traducción a un lenguaje intermedio como es el inglés.

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

Por otra parte, guardando relación con este hecho, se podría solventar este problema si se usara el servicio de pago de *Google* que traduce los mensajes en la nube (figura 27) y así evitar mantener dichos modelos de traducción en el dispositivo. Esto conlleva un coste que ya se ha explicado en la sección 5.

Figura 27: Esquema del proceso de traducción si se hubiese escogido la traducción en la nube usando Google Cloud Platform.

6.2. ¿Cómo enviamos los mensajes?

En este apartado se explicará el envío de mensajes dentro de la aplicación. Cuando el usuario escribe un mensaje en el cuadro de texto y pulsa el botón de enviar, ocurre cierto trabajo totalmente invisible para el usuario. Primeramente se obtienen todos los valores de los que se compondrán el mensajes 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 se guarda el mensaje dentro de la 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, se actualizaría el 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 puede 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, se guardan 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 cómo funcionan otras aplicaciones de mensajería (*WhatsApp*, *Telegram*...).

Todo lo mencionado se puede observar de manera gráfica en 28.

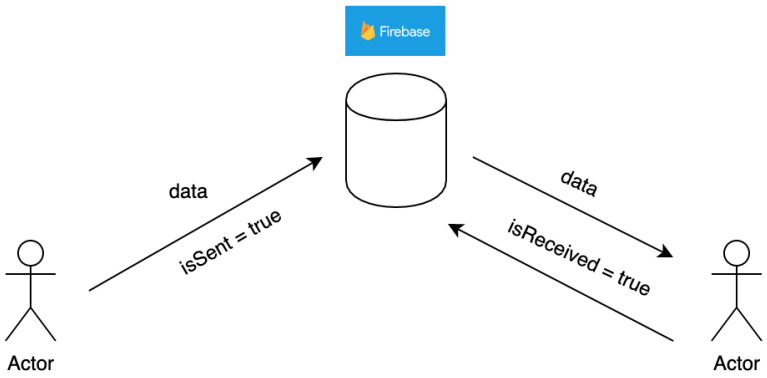


Figura 28: Esquema de envío de mensajes en la aplicación.

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

Cuando se inicia sesión en la aplicación y el usuario accede a **MainActivity**, 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 27: Fragmento de código para escuchar cambios en los usuarios de la aplicación

En el fragmento de código 27, 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.

En la figura 29 se observa el proceso en modo esquema cuando un usuario actualiza su perfil de usuario.

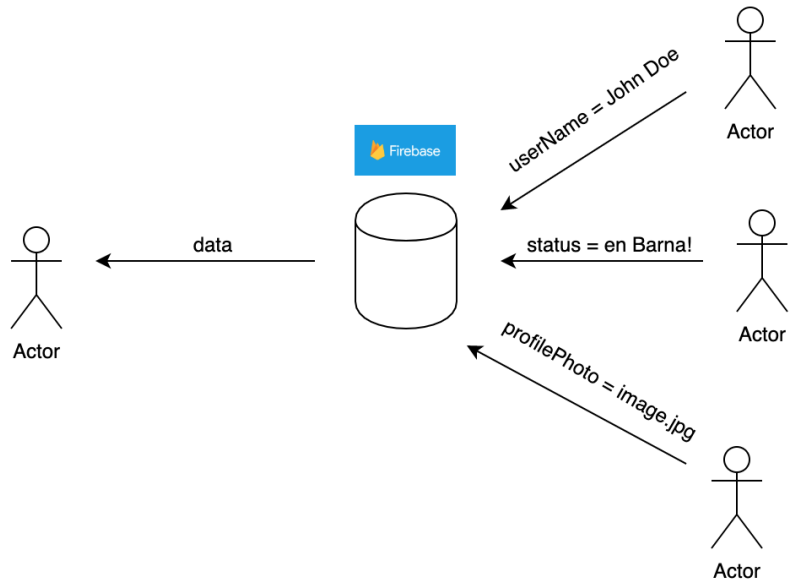


Figura 29: Esquema de notificación cuando un usuario actualiza su perfil de usuario.

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

Por otra parte, se ha comentado brevemente el proceso de guardado de las 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 se ha optado por transformar las imágenes en formate Base64[4] [11].

En relación a lo anterior, se codifica 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 30.

```
>>> import base64
>>> texto = "Prueba de codificación BASE64"
>>> textoCodificado = base64.encodestring(texto)
>>> textoCodificado
'UHJ1ZWJhIGRlIGNvZGlmaWNhY2nDs24gQkFTRTY0IQ=\n'
>>> base64.decodestring(textoCodificado)
'Prueba de codificación BASE64'
```

Figura 30: Ejemplo de código en Python para codificar y decodificar en base64

A modo de esquema, se puede observar en la figura 31 este proceso.

Figura 31: Esquema de traducción de una imagen en formato Bitmap a Base64.

Tal como se ha explicado en secciones anteriores, se usa una librería de Android para visualizar las imágenes de manera optimizada, está librería es Glide donde cuyas ventajas ya se mencionó en 2.5.

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

[4]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.

7 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 se ha 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 39).
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 se ha de olvidar 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 se tuviese dos TextViews, se puede 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 32) y mediante código XML (33).

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.

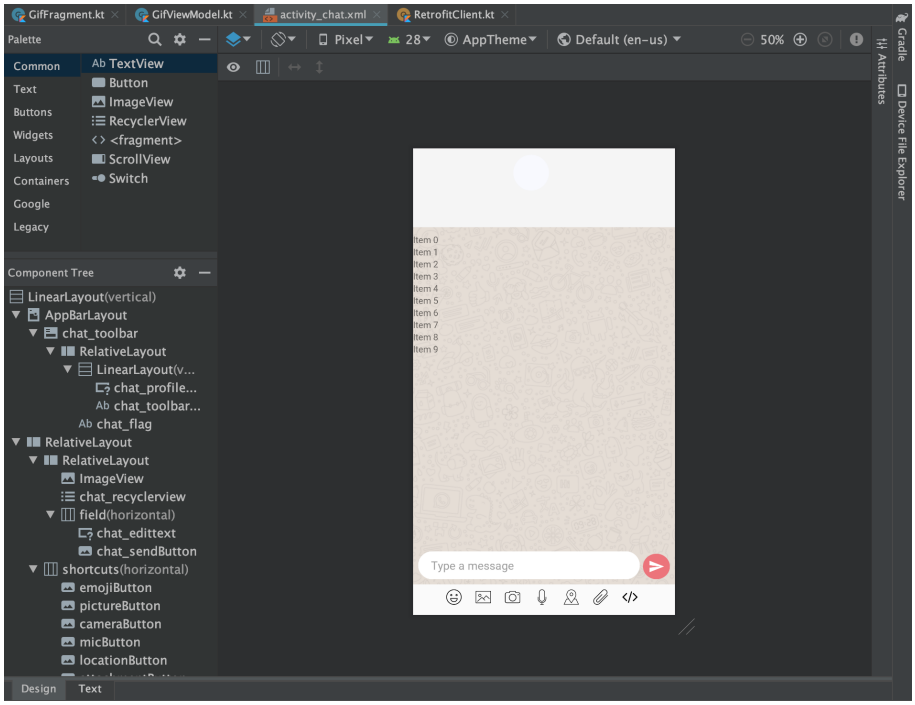


Figura 32: Diseñando la interfaz de la Actividad de manera gráfica. Esta vista corresponde a **ChatActivity** donde se mostrará la lista de mensajes enviados entre dos usuarios.

De ahora en adelante, cuando se hable de **Activity**, se hará referencia a una pantalla de la aplicación.

A continuación se nombra brevemente las distintas actividades que existen 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 aplicacion (imagen de perfil, nombre de usuario, estado).

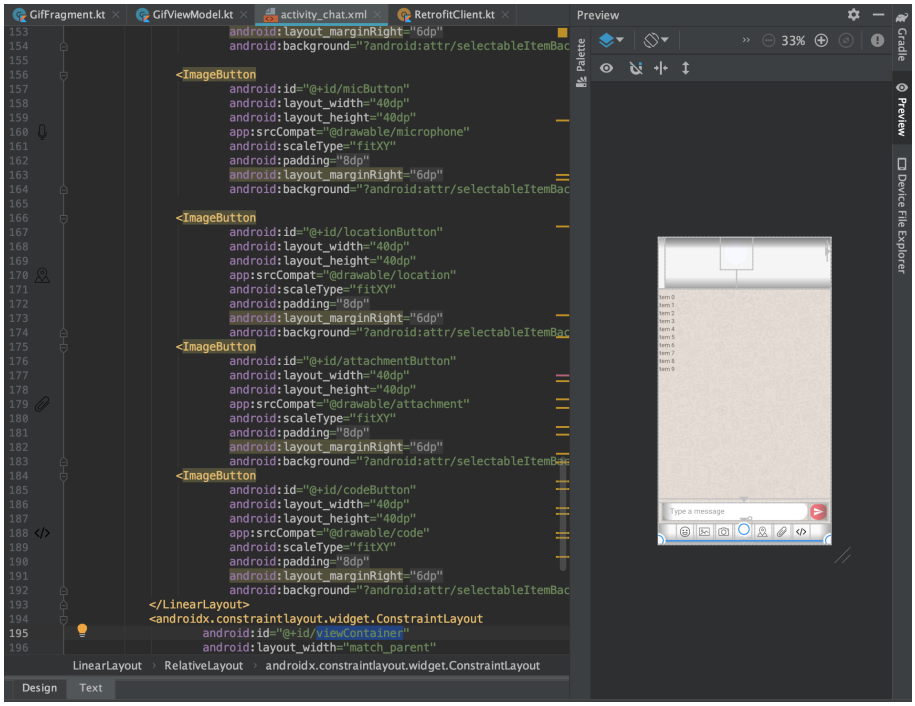


Figura 33: Diseñando la interfaz de la Actividad mediante código XML.

10. **SettingActivity**: permite al usuario descargar y seleccionar el traductor deseado de entre una lista de lenguajes (inglés, español, alemán, chino, japones, etc...).

En la figura 34 se puede observar el diagrama de flujos entre las distintas actividades de las que se componen el proyecto. Este diagrama refleja, tal como su propio nombre indica, el flujo de la interacciones del usuario. Entre cada pareja de actividades se muestra la acción necesaria a realizar para ir de una actividad a la otra. Por ejemplo, si el usuario se encuentra en la primera pantalla, tendrá dos opciones, registrarse en la aplicación o iniciar sesión.

Las interfaces se han diseñado de tal manera que se parezcan a las aplicaciones de mensajería existentes como son *WhatsApp* y *Telegram*. Esto se ha diseñado así debido a que desde la perspectiva del usuario es más intuitivo a la hora de usar la aplicación, por lo que la curva de aprendizaje es mínima.

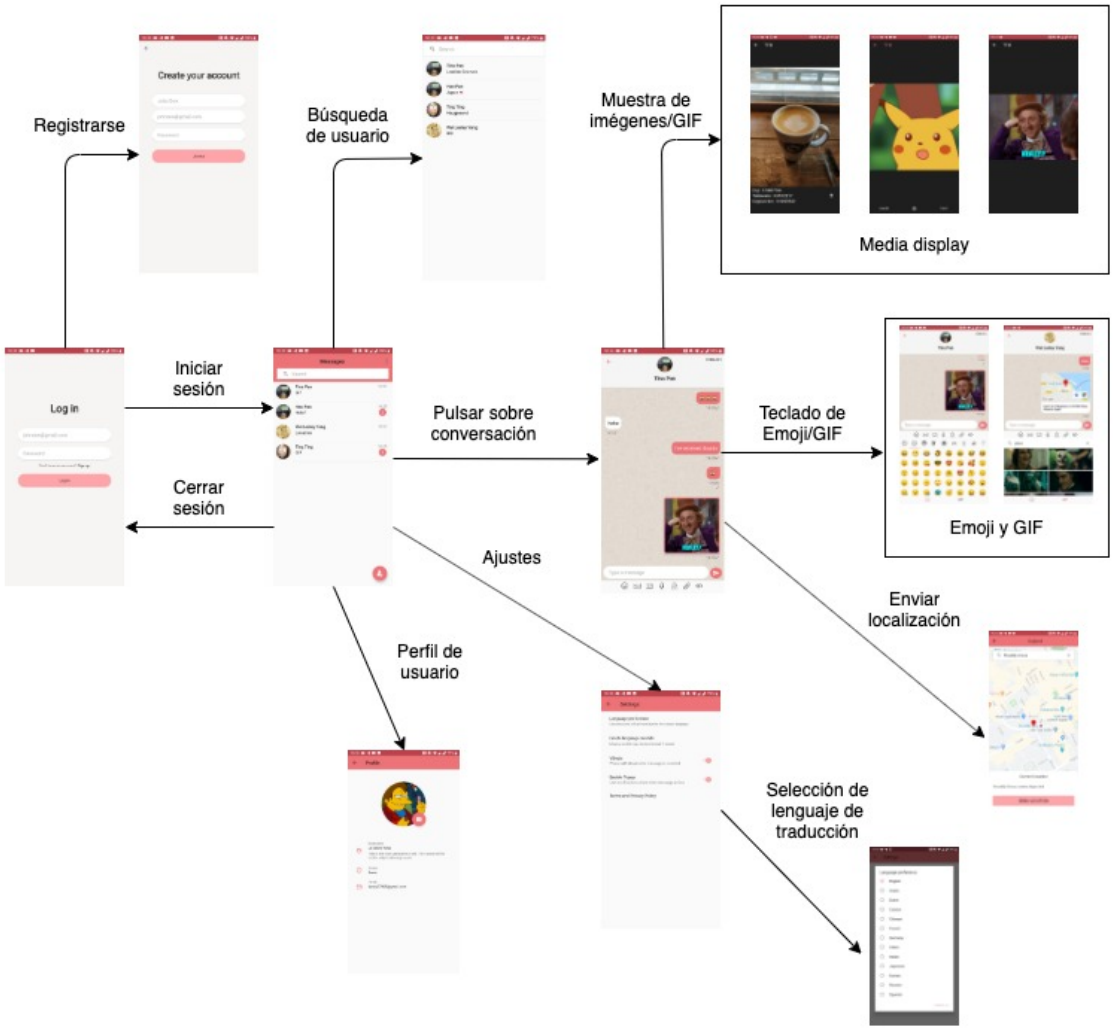


Figura 34: Diagrama de flujo de las actividades en el proyecto.

7.1. 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 35.

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.

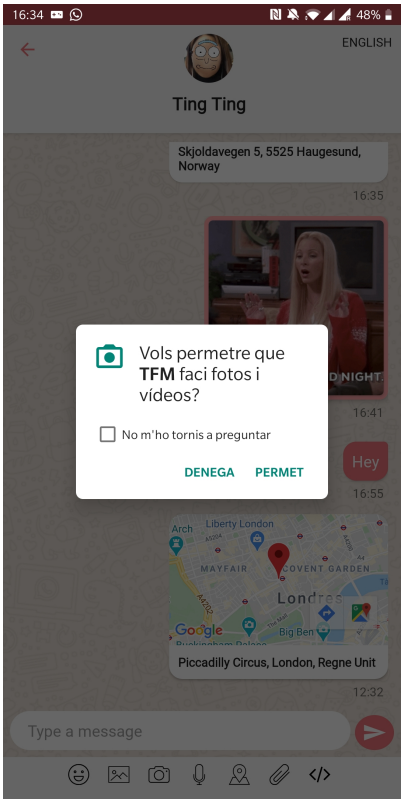


Figura 35: Petición de permisos en Android.

En el fragmento de código 28 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 35.

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.
                   PERMISSION_GRANTED){
5          ActivityCompat.requestPermissions(this,
6              arrayOf(Manifest.permission.CAMERA),
7              DataRepository.CAMERA_PERMISSION)
8      }else{
9          openCamera()
10     }
11 }
```

Listing 28: Método cameraBtn() para cuando el usuario pulsa sobre el icono de la cámara

```
override fun onRequestPermissionsResult(requestCode: Int, permissions:
    Array<out String>, grantResults: IntArray) {
    if ((grantResults.isNotEmpty() && grantResults[0] == PackageManager.
        PERMISSION_GRANTED)) {
        when(requestCode){
            DataRepository.CAMERA_PERMISSION -> {
                openCamera()
            }

            DataRepository.STORAGE_PERMISSION -> {
                openGallery()
            }

            DataRepository.LOCATION_PERMISSION -> {
                openLocation()
            }

            DataRepository.AUDIO_PERMISSION -> {
                openMic()
            }
        }
    }
}
```

Dentro de la 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.

7.2. Login

En la figura 36 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 se aceptan correos con la extensión **@gmail.com** por simplicidad, aunque si se quisiera 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 37 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

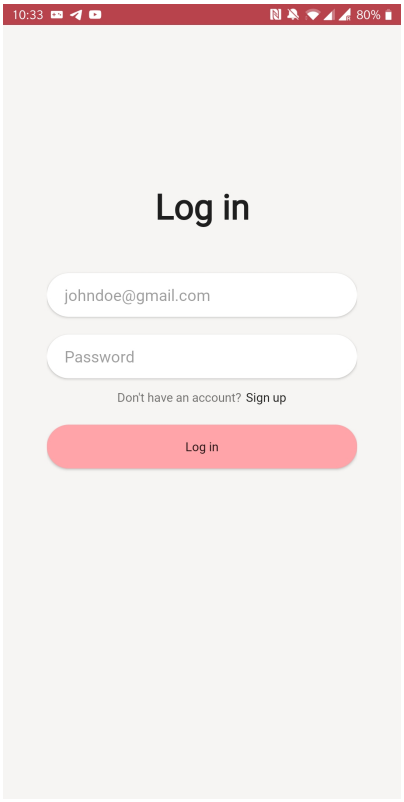


Figura 36: Pantalla de inicio donde el usuario accede a la aplicación mediante sus credenciales.

principal de la figura 38. Esto se consigue salvaguardando las credenciales en la aplicación.

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

Listing 29: Fragmento de código para accionar los botones de la pantalla de inicio

En el fragmento de código 29 se puede observar dos métodos, launchSignup() que al pulsarlo direcciona al usuario a la pantalla de registro (figura 37) 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 30). Por otra parte, en

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

Listing 30: Método login de la clase FirebaseUtil.kt

caso afirmativo se actualizarían las credenciales en la aplicación y las variables tales como **DataRepository.user** y **DataRepository.currentUserEmail** que serán usadas más adelante, como por ejemplo, para determinar qué perfil de usuario actualizar cuando el usuario cambie su imagen de perfil.

7.3. Sign up

Una vez en la pantalla de registro (*Sign up*), el usuario puede dar marcha atrás a la pantalla de *Log in* o crear una usuario nuevo, si se crea una cuenta nueva se deberá respetar una serie de condiciones para garantizar cierta seguridad y son:

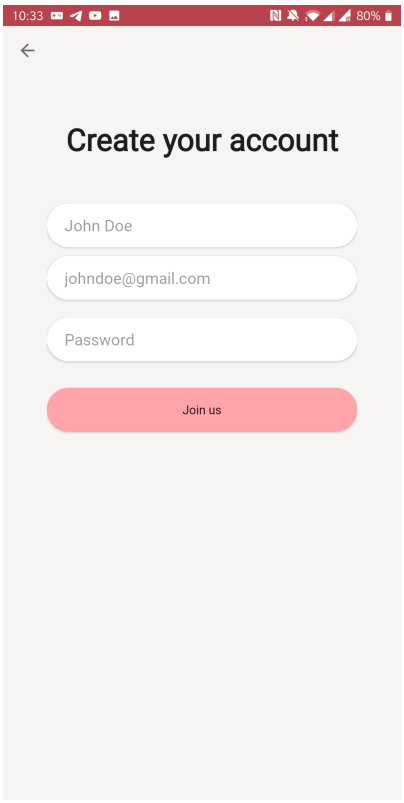


Figura 37: Pantalla de SignUpActivity donde permite al usuario crear una nueva cuenta de usuario.

1. El correo electrónico deberá terminar en *@gmail.com*.
2. El correo electrónico no debe existir ya en la base de datos.
3. La longitud de la contraseña deberá ser mayor de ocho 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 (véase la figura 38).

Esto se puede ver en el fragmento de código 31 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
         : String){
2        FirebaseUtil.createNewUser(context, username, email, password)
3    }
```

Listing 32: Método joinNewUser() de la clase SignupViewModel.kt

```
1
2     private fun isFormNotEmpty() = eUser.text.isNotEmpty() && eEmail.text.
          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 31: Función signup para registrar un nuevo usuario

En la sección 4.2, se ha hablado de cómo se estructuría el 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,
          password: String) {
2         val user = User("", email, username, "", "")
3         val hashcode = user.hashCode().toString()
4         user.id = hashcode
5
6         val auth = FirebaseAuth.getInstance()
7         auth.createUserWithEmailAndPassword(email, password)
8             .addOnSuccessListener {
9                 FirebaseFirestore.getInstance().addUser(context, user)
10                SignupActivity.currentUserEmail = email
11                SignupActivity.currentUserPassword = password
12            }.addOnFailureListener {
13                SignupViewModel.isJoinUsSuccessful.postValue(false)
14                context.toast("Cannot create user with those inputs")
15            }
16    }
```

Listing 33: Método createNewUser que se encarga de crear el nuevo usuario y direccionar al usuario a la pantalla principal

7.4. Main Activity

El **MainActivity** se encarga de organizar la lista de conversaciones que el usuario posee. En cuanto al diseño de dicha actividad (Véase la figura 38), se puede observar que existe un *Toolbar* en la parte superior con el título de *Messages* y un SearchView para filtrar la lista de conversaciones en un momento dado, muy útil en el caso de tener una cantidad elevada de conversaciones (Véase la figura 39).

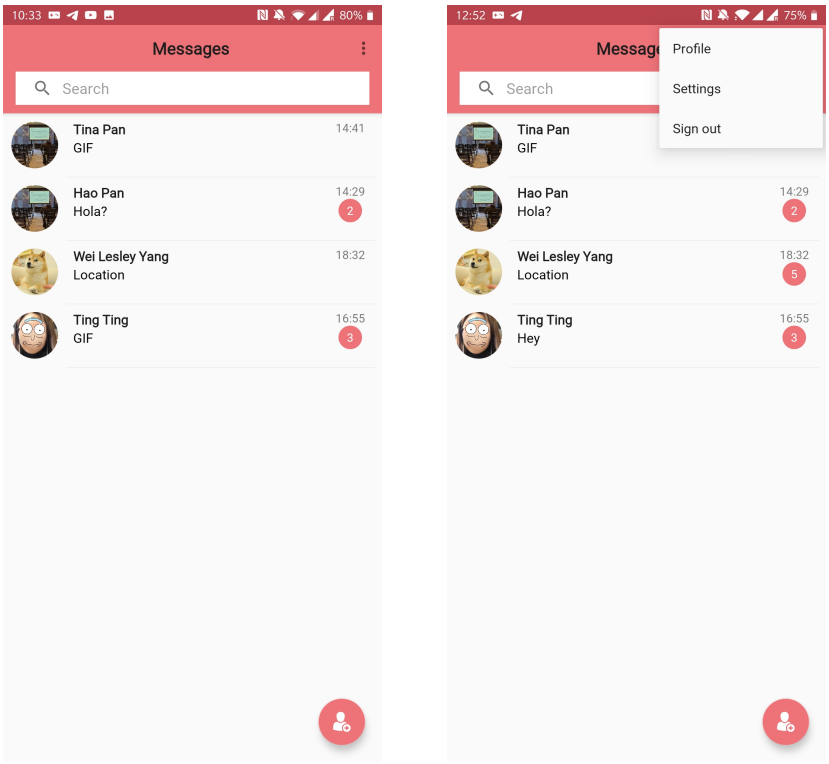


Figura 38: Pantalla principal donde se encuentran todas las conversaciones del usuario.

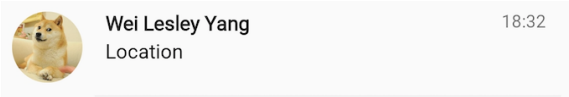


Figura 39: Formato que tiene una conversación en la aplicación. Dentro del contexto de un desarrollador, este *item* se le conoce como ViewHolder.

Dentro de esta pantalla tenemos varias opciones a realizar (Véase el diagrama de la figura 34):

1. Filtrar conversaciones gracias al SearchView del Toolbar.
2. Ir a la pantalla de la figura 49.
3. Ir a la pantalla de la figura 51.
4. Cerrar sesión e ir a a la pantalla de la figura 36.
5. Pulsando el FloatingActionButton de la esquina inferior derecha, la aplicación nos llevará a la pantalla de la figura 48.
6. Si se pulsa sobre una conversación, se mostrará la Activity de la figura 42 con los mensajes correspondientes.

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

1. La compatibilidad de los emoticonos, si no se inicializa el servicio, los emoticonos que se reciban de los usuarios puede causar errores en nuestra aplicación.
2. También se inicializa la base de datos para obtener la lista de conversaciones que se guardan en la base de datos SQLite (local).
3. El servicio de traducción, sin él cuando se requiera 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 34

```
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(
          applicationContext))
9
10    private fun initDatabase() = MyRoomDatabase.getMyRoomDatabase(this)
```

Listing 34: Inicialización de todos los métodos necesarios para el correcto funcionamiento de la aplicación

Tal como se puede imaginar, el **MainActivity** es una de las clases principales del proyecto, realiza una multitud de acciones mencionados anteriormente. A continuación se mostrará cómo funciona cada una de las distintas partes.

Comenzando con el SearchView, este sirve para filtrar las conversaciones dentro de la 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    })
```

Listing 35: Campo de búsqueda para filtrar las conversaciones

En el fragmento de código 35 se ha referenciado el SearchView mediante ButterKnife, el cual, evita en gran medida lo que se conoce como «boilerplate code» en el contexto de la programación.

Líneas más abajo, se observa 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()
                 , ignoreCase = true) ||
4                    it.userTwoEmail.removeAfter('@').contains(text.toString(),
                         ignoreCase = true)}
5            .toMutableList()
6
7        conversationList.postValue(list)
8    }
```

Listing 36: Método filterList() que se llama cada vez que el usuario añade un carácter en el SearchView

Por otra parte, toca explicar cómo se ha 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:
         RecyclerView
2
3    private fun initRecyclerView(){
4        viewManager = LinearLayoutManager(this)
5        viewAdapter = ConversationAdapter(mutableListOf())
6
7        conversations.apply {
8            setHasFixedSize(true)
9            addItemDecoration(HorizontalDivider(this.context))
10           layoutManager = viewManager
11           adapter = viewAdapter
12       }
13   }
```

Listing 37: Inicialización del RecyclerView en el MainActivity

En el fragmento de código 37 se puede observar como se referencia el RecyclerView y al llamar el método initRecyclerView(), tal como su nombre indica, se inicializa la lista, y el resto de clases necesarias para poder visualizar el conjunto de conversaciones (Véase la figura 40).

Por ello, para cada objeto personalizado que se quiera visualizar en una lista, se ha de crear una clase extendiendo de RecyclerView.Adapter<RecyclerView.ViewHolder>. El código asociado se puede observar en 38.

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

1. **onCreateViewHolder()**: este método especifica qué ViewHolder se ha de crear, en nuestro caso **ConversationViewHolder** que es el correspondiente a la figura 39.

2. **onBindViewHolder()**: este método se dedica a asociar el valor de las variables a los componentes que tenemos en **ConversationViewHolder**.

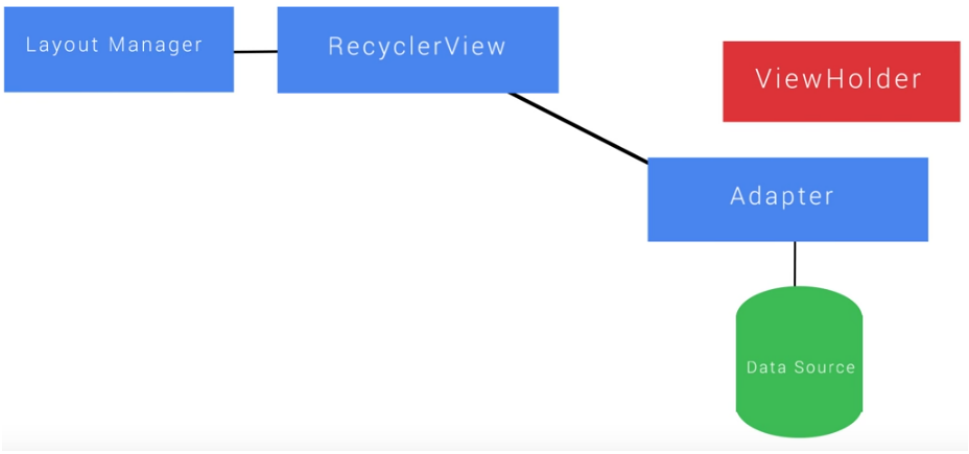


Figura 40: Esquema del funcionamiento de un RecyclerView

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 este caso, esta variación del clásico botón se utiliza para navegar a **UserSearcherActivity** (Véase la figura 48)

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

Listing 39: Método asociado al FloatingActionButton

Ahora bien, si se observa la figura 38, en la segunda imagen observamos que existe una pestaña con tres opciones, al pulsar en cualquier opción se navega a **UserProfileActivity**, **SettingsActivity** o **LoginActivity** respectivamente.

Una peculiaridad es si se pulsa en «Sign out», la aplicación nos mostrará una alerta (Véase la figura 41) de si se está seguro en cerrar la sesión ya que se borrarían 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 se pulsas una imagen de perfil,muestra un diálogo con dicha imagen y para ello se ha hecho uso de las *Function Extension* que se explicó en 14.

```
class ConversationAdapter(private val conversations: MutableList<Conversation>)
    : RecyclerView.Adapter<ConversationViewHolder>(){
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
        ConversationViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.
            viewholder_conversation, parent, false)
        return ConversationViewHolder(view)
    }

    override fun onBindViewHolder(holder: ConversationViewHolder, position: Int
        ) {
        val conversation = conversations[position]
        holder.bindViewHolder(conversation)
    }

    override fun getItemCount() = conversations.size

    fun updateList( newConversations : MutableList<Conversation>){
        val diffResult: DiffUtil.DiffResult = DiffUtil.calculateDiff(
            ConversationDiffCallback(conversations, newConversations))
        conversations.clear()
        conversations.addAll(newConversations.sortedByDescending { it.timestamp
            })
        diffResult.dispatchUpdatesTo(this)
    }
}
```

Listing 38: Código asociado al adaptador para poder visualizar la lista de conversaciones

```
    private fun createSignoutDialog(): AlertDialog{
        return AlertDialog.Builder(this)
            .setTitle(R.string.signout)
            .setMessage(R.string.signout_alert)
            .setPositiveButton(R.string.sure) { _, _ ->
                signout()
            }
            .setNegativeButton(R.string.cancel) { dialog, _ ->
                dialog.dismiss()
            }

            .create()
    }

    private fun signout(){
        FirebaseAuth.getInstance().signOut()
        PreferenceManager.getDefaultSharedPreferences(applicationContext).
            clearCredential()
        startActivity(Intent(this, LoginActivity::class.java))
        finish()
    }
```

Listing 40: Método createSignoutDialog() que mostrará al usuario una alerta

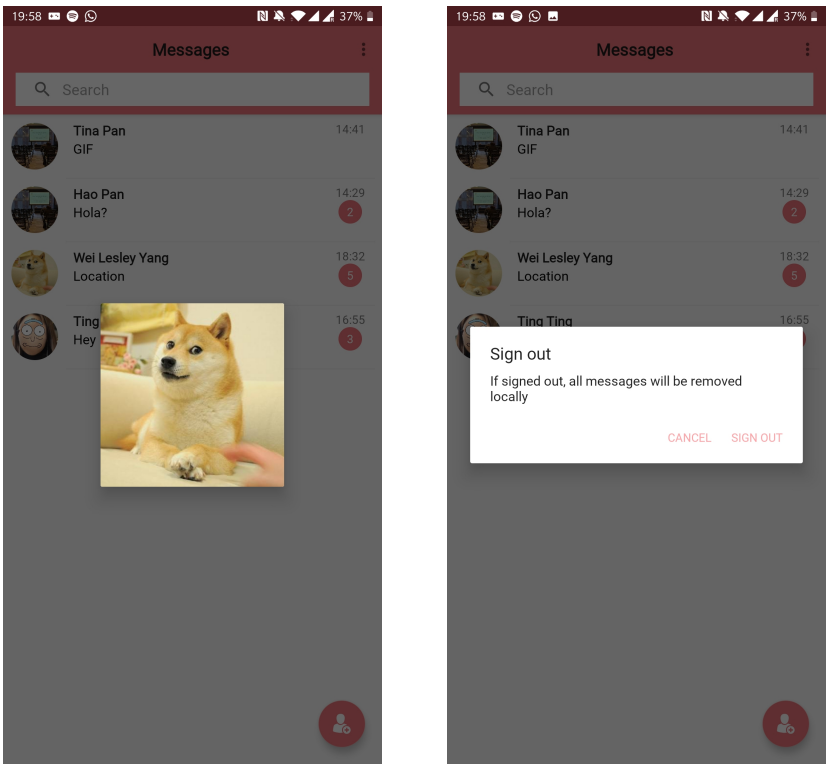


Figura 41: A la izquierda se puede observar cuando el usuario pulsa la imagen de perfil de una conversaciones. A la derecha se muestra la alerta cuando el usuario quiere cerrar sesión.

```
fun CircleImageView.showDialog(context: Context, imageBase64: String?){
    try{
        imageBase64?.let {
            val dialog  = Dialog(context)
            dialog.setContentView(R.layout.dialog_imagedisplay)
            val dialogPhoto = dialog.findViewById<ImageView>(R.id.
                dialog_imagedisplay)
            Glide.with(context).load(imageBase64.toBitmap()).into(dialogPhoto)
            dialog.show()
        }
    }catch (e: Exception){
        e.printStackTrace()
    }
}
```

Listing 41: 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

7.5. Chat Activity

Junto a **MainActivity**, esta es una de las pantallas más importantes dentro de la aplicación, ya que es aquí donde se encuentra la funcionalidad básica, enviar mensajes.

Esta actividad se encarga de toda la funcionalidad relacionada con los mensajes, tales como mostrar y enviar.

Como se puede observar en la figura 42, 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 la aplicación, es decir, si fuese «English» los mensajes que envían el resto de usuarios se traducirían de su idioma al inglés.

Luego más adelante, se encontrará 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.

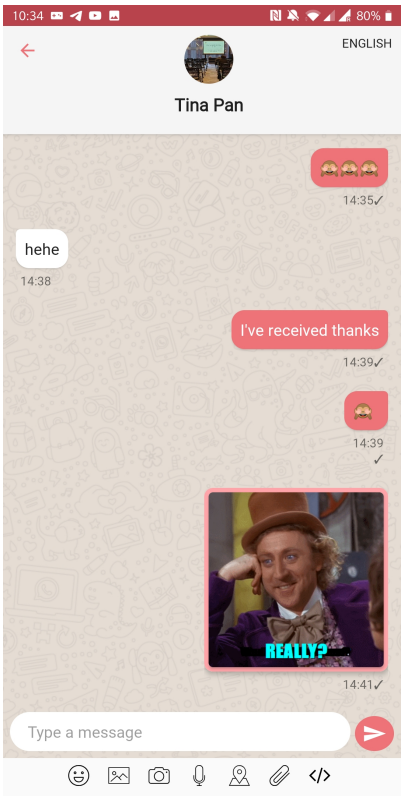


Figura 42: Captura de pantalla de ChatActivity.kt.

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

Además, dentro de esta pantalla existen varios fragmentos que se encargarán de añadir emoticonos (figura 43), GIFs, imágenes, localización (figura 47).

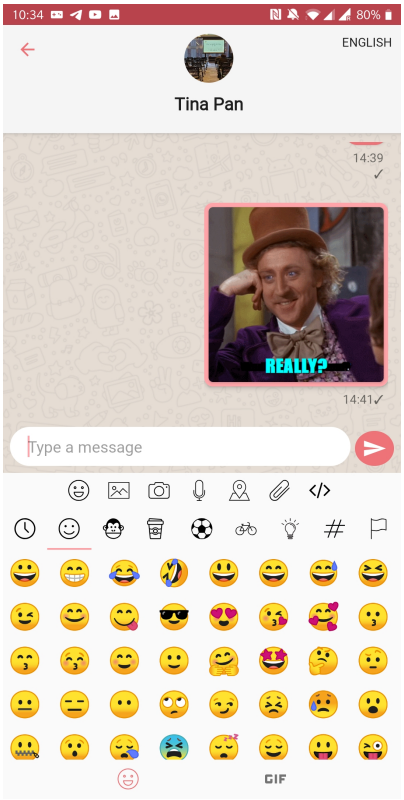


Figura 43: ChatActivity mostrando los emoticonos.

Se comenzará explicando cómo se ha implementado el Toolbar que se observa, en el cual se compone de un CircleImageView, un TextView para el nombre de usuario y otro para mostrar el usuario qué lenguaje de traducción ha escogido para traducir los mensajes.

Por otra parte, se tiene la lista de mensajes que como se ha explicado anteriormente, puede ser de varios tipos.

Se observa en el fragmento de código que dependiendo del tipo de mensaje (texto, imagen, GIF, localización) se crea un tipo de ViewHolder u otro, por otra parte, también se debe iniciar el contenido de dichos mensajes, tal como se explicó en la sección anterior.

En el fragmento de código 43 destaca una serie de puntos:

1. Se puede observar que dependiendo de si el emisor de mensaje es uno mismo o no (Véase la figura 45), se renderiza 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 se realiza la traducción del mensaje, dentro de la instancia de *Message*, se encuentra un atributo que indica en qué idioma se envió originalmente y también el mensaje traducido al inglés, si coincide con la preferencia escogida en la aplicación, entonces no se traduce, en caso contrario se traduciría del inglés al idioma escogido.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
    RecyclerView.ViewHolder {
    var view: View?

    when(MessageType.fromInt(viewType)){
        MessageType.MESSAGE -> {
            view = LayoutInflater.from(parent.context).inflate(R.layout.
                viewholder_message, parent, false)
            return MessageViewHolder(view)
        }

        MessageType.IMAGE, MessageType.GIF ->{
            view = LayoutInflater.from(parent.context).inflate(R.layout.
                viewholder_image, parent, false)
            return ImageViewHolder(view)
        }

        MessageType.LOCATION -> {
            view = LayoutInflater.from(parent.context).inflate(R.layout.
                viewholder_location, parent, false)
            return LocationViewHolder(view)
        }

        MessageType.ATTACHMENT -> {
            view = LayoutInflater.from(parent.context).inflate(R.layout.
                viewholder_attachment, parent, false)
            return AttachmentViewHolder(view)
        }
    }
}
```

Listing 42: Fragmento de código relacionado a la creación de un mensaje según su tipo

```
override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position:
    Int) {
    val message = messages[position]

    when(holder){
        is MessageViewHolder -> {
            holder.initMessageViewHolder(message)
        }

        is ImageViewHolder ->  {
            holder.initImageViewHolder(message)
        }

        is LocationViewHolder -> {
            holder.initAndUpdateMap(context, message)
        }

        is AttachmentViewHolder -> {
            holder.initAttachmentViewHolder(message)
        }
    }
}
```

Listing 43: Código asociado a inicializar un texto plano en el ViewHolder

```
    //texto plano
    fun initMessageViewHolder(message: Message){
        if(message.senderName == currentUserEmail){
            setSenderViewHolder()
        }else{
            setReceiverViewHolder()
        }

        initLayout(message)

        setTime(time, message.timestamp)
        setMessageCheckIfSeen(time, message.senderName == currentUserEmail,
            message.isSent)
    }

    private fun initLayout(message: Message){
        val code = message.body?.fieldThree?.toInt()

        if(code == languagePreferenceCode || languagePreferenceCode == null){
            body.text = message.body?.fieldOne
        }else if(code != LanguageCode.ENGLISH.code && (languagePreferenceCode
            == LanguageCode.ENGLISH.code || languagePreferenceCode == null)){
            body.text = message.body?.fieldTwo
        }else{
            val translator = DataRepository.fromEnglishTranslator
            translator?.let {
                var textToTranslate = if(message.body?.fieldThree?.toInt() ==
                    LanguageCode.ENGLISH.code){
                    message.body?.fieldOne
                }else{
                    message.body?.fieldTwo
                }

                translator.translate(textToTranslate.toString())
                    .addOnSuccessListener { translatedText ->
                        body.text = translatedText
                    }
            }
        }
    }
```

Listing 44: Código asociado a inicializar un mensaje de texto plano

```
    private fun setSenderViewHolder(){
        val context = layout.context
        val layoutParams = RelativeLayout.LayoutParams(
            RelativeLayout.LayoutParams.MATCH_PARENT,
            RelativeLayout.LayoutParams.WRAP_CONTENT)

        layoutParams.setMargins(getDpValue(40), 0, 0, 0)
        layout.layoutParams = layoutParams
        layout.gravity = Gravity.END
        placeholder.background = context.getDrawable(R.drawable.sender_message)
        layout.setPadding(0,getDpValue(10),getDpValue(15), getDpValue(10))
        body.setTextColor(context.getColor(R.color.colorWhite))
        body.gravity = Gravity.START
        time.gravity = Gravity.END
    }
```

Listing 45: 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.
              receiver_message)
11        layout.setPadding(getDpValue(15),getDpValue(10), 0, getDpValue(10))
12        body.setTextColor(context.getColor(R.color.colorPrimaryText))
13        body.gravity  = Gravity.START
14        time.gravity = Gravity.START
15    }
```

Listing 46: Fragmento de código para determinar si un mensaje es emisor o receptor

Por último en esta sección se hablará de los fragmentos que existen en la barra inferior, si se observa la figura 43 se puede observar que existe una serie de pestañas. La funcionalidad de cada pestaña se explicará a continuación:

1. **Emoticonos y GIFs**: esta primera pestaña muestra una lista de emoticonos (figura 44), básicamente estos emoticonos son caracteres Unicode. Luego en la parte inferior, permite seleccionar la opción de GIF (figura 19).
2. **Seleccionar imagen**: al pulsar sobre esta pestaña, llevará al usuario a su galería interna del dispositivo, al seleccionar una foto lanzará **ImageDisplayActivity**.
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**.
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**.
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, se explicará 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** rellenando 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 se traduce el mensaje, si está en cualquier otro idioma entonces se traduce este mensaje al inglés y se llama al método del fragmento 48.

```
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(
                  translateModel.toString())
6             val timestamp = System.currentTimeMillis()
7
8             val message = Message(timestamp, conversationId, DataRepository.
                  currentUserEmail, receiverUser,
9                 MessageType.MESSAGE.value, null, timestamp )
10
11            if(languageCode == LanguageCode.ENGLISH.code){
12                message.body = MessageContent(fieldText, "", languageCode.
                      toString())
13                chatViewModel.sendMessage(message)
14            }else{
15                val translator = DataRepository.toEnglishTranslator
16
17                translator?.let{
18                        it.translate(fieldText).addOnSuccessListener {
                              translatedText ->
19                            message.body = MessageContent(fieldText,
                                  translatedText, languageCode.toString())
20                            chatViewModel.sendMessage(message)
21                        }.addOnFailureListener {
22                            Log.d("TFM", "Cannot translate")
23                    }
24                }
25            }
26
27            chat_edittext.text.clear()
28        }
```

Listing 47: Método sendMessage del ChatActivity se traduce el mensaje al ingles al pulsar el botón

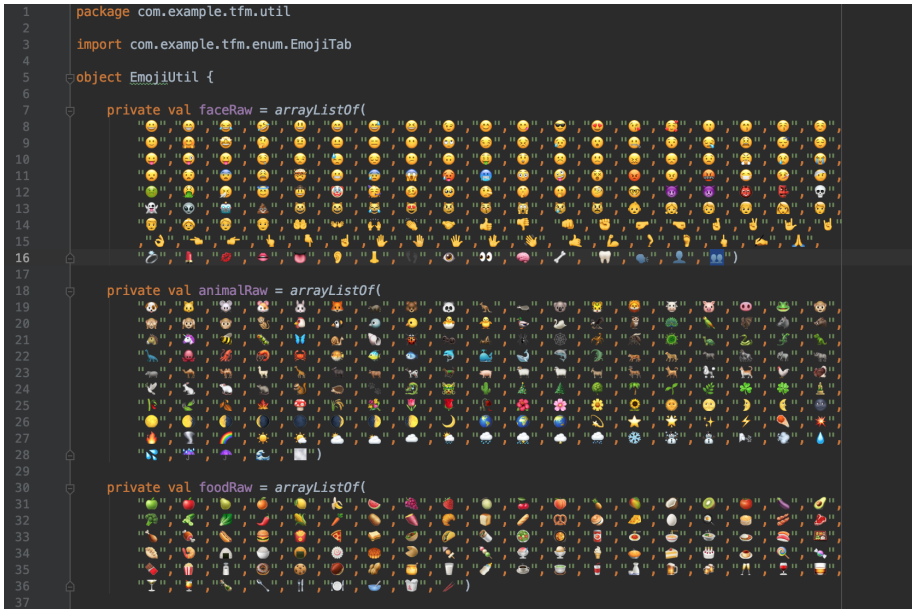


Figura 44: Listado de emoticonos dentro del proyecto en IntelliJ.

```
fun sendMessage(message: Message){
    FirebaseUtil.addMessageFirebase(message)
}
```

Listing 48: Método sendMessage de la clase ChatViewModel

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

En el fragmento de código 49 añadimos el mensaje en Firebase. Cuando el listener detecta que este mensaje se ha añadido correctamente, se procede a añadirlo también a la base de datos Room e inmediatamente se actualiza la conversación (por ejemplo el «último mensaje»).

```
fun addMessageFirebase(message: Message) {
    database.child(FIREBASE_PRIVATE_CHAT_PATH).child(message.ownerId)
        .child(FIREBASE_PRIVATE_MESSAGE_PATH)
        .child(message.timestamp.toString())
        .setValue(message)
        .addOnSuccessListener {
            message.isSent = true
            CoroutineScope(Dispatchers.IO).launch {
                roomDatabase.addMessage(message)
                updateConversation(message)
            }
        }
        .addOnFailureListener {
            Log.d(LogUtil.TAG, "Error while sending message")
        }
}
```

Listing 49: Método addMessageFirebase de la clase FirebaseUtil

7.6. Image Display Activity

Existen dos actividades encargadas de mostrar imágenes o GIFs al usuario, estas son **ImageDisplayActivity** e **ImageToolActivity**. La primera de ella se muestra cuando el usuario pulsa sobre una imagen o GIF (Véase la figura 45) y la segunda cuando el usuario envía.

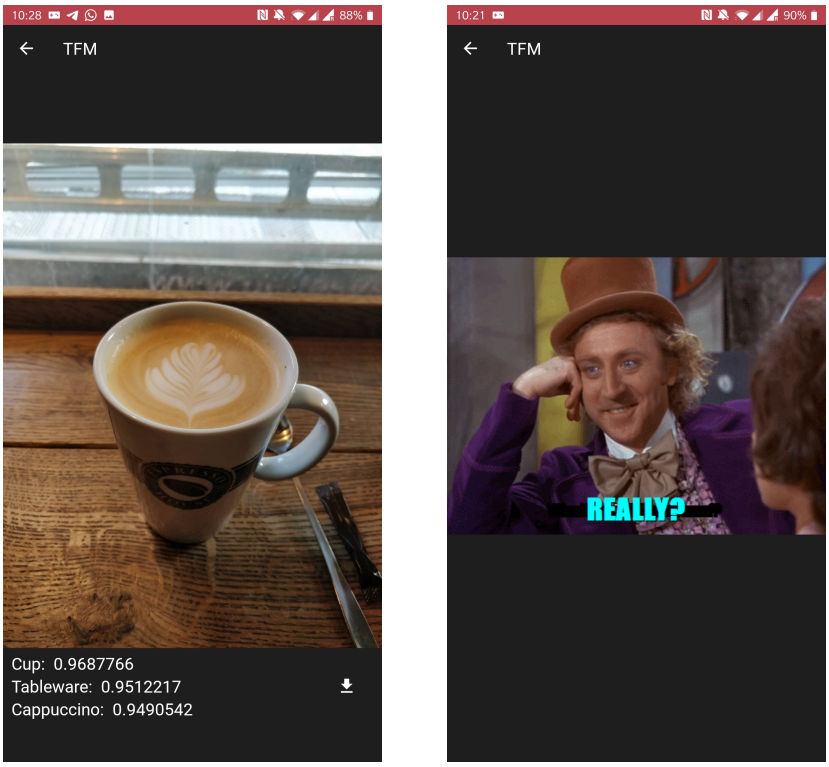


Figura 45: A la izquierda tenemos la visualización de una imagen y a la derecha de un GIF.

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.

En el fragmento de código 50 se puede observar ambos métodos que realizan esta acción.

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 45 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 51 se observa que se obtiene una instancia del DeviceImageLabeler de FirebaseVision y a partir de ahí, dada una imagen, el labeler en cuestión procesa la información y devuelve los cocientes de probabilidad.

Asimismo, existe la opción de descargar una imagen y se puede observar en el fragmento de código 52.

En cuanto al GIF, simplemente lo mostramos al usuario.

```
fun Bitmap.toBase64() : String{
    val outputStream = ByteArrayOutputStream()
    this.compress(Bitmap.CompressFormat.JPEG, 70, outputStream)
    return Base64.encodeToString(outputStream.toByteArray(), Base64.DEFAULT
        )
}

fun String.toBitmap() : Bitmap? {
    val bitmap = Base64.decode(this, Base64.DEFAULT)

    bitmap?.let {
        return BitmapFactory.decodeByteArray(bitmap, 0 , bitmap.size)
    }

    return null
}
```

Listing 50: Métodos para pasar de un Bitmap a una imagen en formato Base64 y viceversa

```
val labeler = FirebaseVision.getInstance().onDeviceImageLabeler

labeler.processImage(image)
    .addOnSuccessListener {
        it.forEachIndexed { index, label ->
            when(index){
                1 -> {  labelOne.text = "${label.text}:"
                        confidenceOne.text = label.confidence.toString() }
                2 -> {  labelTwo.text = "${label.text}:"
                        confidenceTwo.text = label.confidence.toString() }
                3 -> {  labelThree.text = "${label.text}:"
                        confidenceThree.text = label.confidence.toString()
                            }
            }
        }
    }
    .addOnFailureListener {
        Log.d(LogUtil.TAG, "failure")
    }
```

Listing 51: Código relacionado con FirebaseVision

```
@OnClick(R.id.imagedisplay_download)
fun downloadImage(){
    val randomNum = (Math.random() * 100000 + 1).toInt()
    bitmap?.let {
        val path = MediaStore.Images.Media.insertImage(contentResolver,
            bitmap, "IMAGE${randomNum}", "")
        Uri.parse(path)

        toast("Image saved")
    }

    finish()
}
```

Listing 52: Método downloadImage() en la clase ImageDisplayActivity.kt

7.7. 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** (Véase la figura 46). Dentro de esta actividad se puede realizar cuatro acciones:

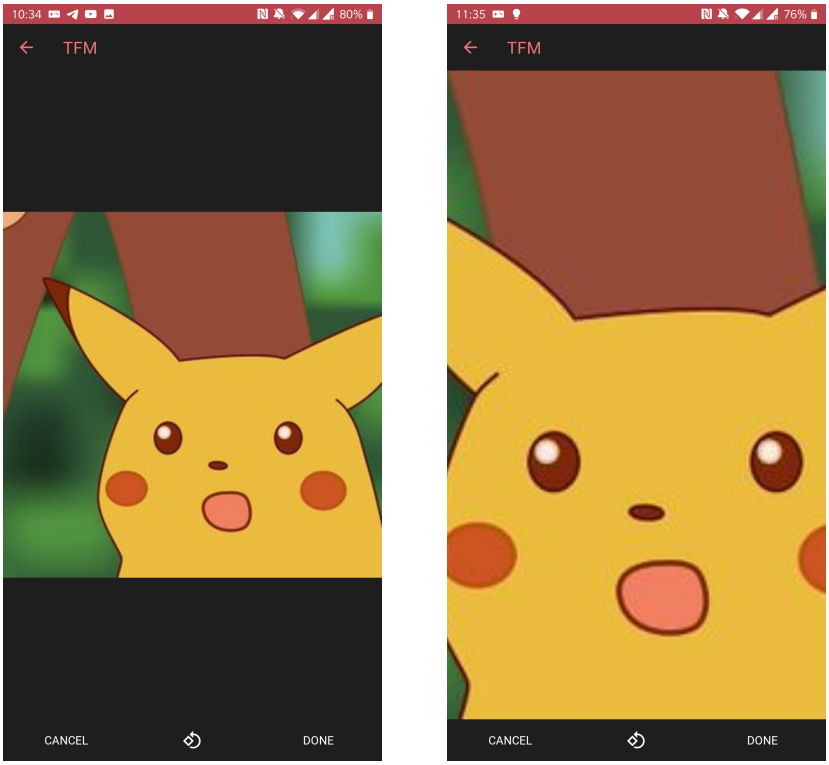


Figura 46: ImageToolActivity donde se encarga del tratamiento de imágenes. Permite al usuario realizar zoom y rotaciones.

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.

A continuación se muestra 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 53: 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**.

```
@OnClick(R.id.tool_rotate)
fun rotate(){
    content = touchImage.rotate()
}

//en ExtensionFunctionUtil.kt
fun ImageView.rotate(): String{
    val matrix = Matrix().apply {
        postRotate(-90F)
    }
    val bitmap = (drawable as BitmapDrawable).bitmap
    val rotatedBitmap = Bitmap.createBitmap(bitmap, 0,0, bitmap.width,
        bitmap.height, matrix, true)
    setImageBitmap(rotatedBitmap)

    return rotatedBitmap.toBase64()
}
```

Listing 54: Método rotate()

```
@OnClick(R.id.tool_accept)
fun accept(){
    val timestamp = System.currentTimeMillis()
    val message = Message(timestamp, ChatActivity.conversationId,
        currentUserEmail,
        ChatActivity.receiverUser, typeValue, MessageContent(fieldOne =
            content), timestamp)

    FirebaseUtil.addMessageFirebase(message)
    finish()
}
```

Listing 55: Método accept()

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 se crea 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 se observa de nuevo la segunda imagen en la figura 46, se ve 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).

7.8. Location Activity

Otra de las funcionalidades que existe en la aplicación es el envío de localización. Por ello, en la actividad **LocationSenderActivity** (Véase la figura 47) existe un SearchView con un método asignado (Fragmento de código 57) para que el usuario pueda introducir cualquier dirección. En la figura 47 se observa un ejemplo donde hemos introducido «Picadilly Circus» y la aplicación nos lleva a dicha dirección representándola con el puntero.

```
<com.ortiz.touchview.TouchImageView
    android:id="@+id/tool_touchimage"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scaleType="fitCenter"
    app:srcCompat="@drawable/white_placeholder"
    android:layout_centerVertical="true"
    android:layout_gravity="center"
    android:visibility="gone"/>
```

Listing 56: Inicializando atributos al TouchImageView

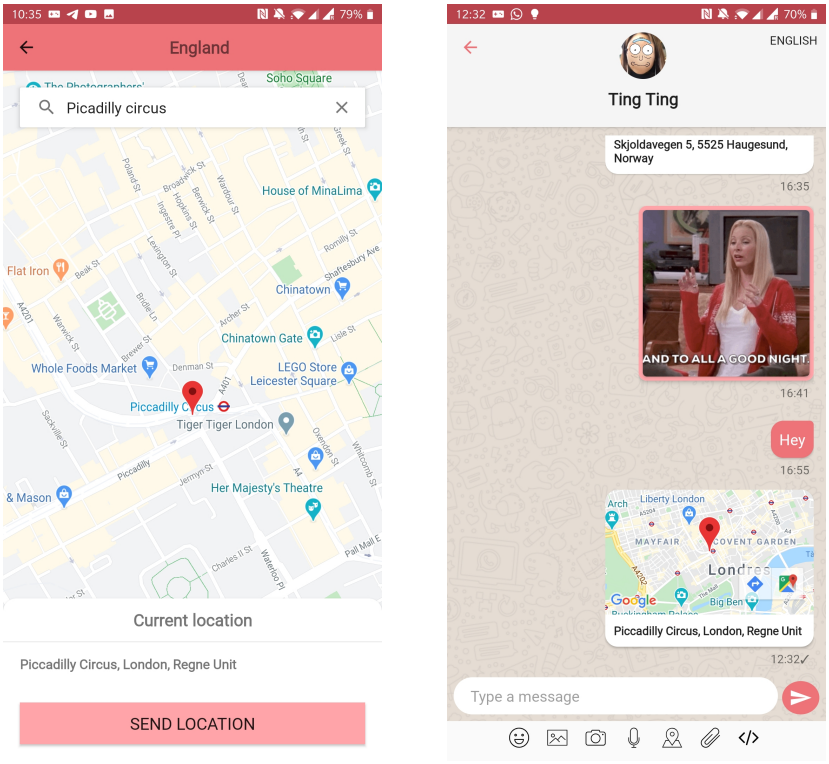


Figura 47: Pantalla de LocationSenderActivity donde se permite al usuario enviar su localización o cualquier localización deseada.

En la segunda imagen de la figura 47, se muestra un ejemplo de cómo se visualizaría la localización enviada en la conversación. Si se pulsa 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 **LocationSenderActivity** 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 58.

```
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 57: Método searchLocation() para cuando el usuario introduce una dirección en el SearchView

```
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(
                      BitmapDescriptorFactory.HUE_RED)))
8         }
9
10        locationViewModel.setNewLocation(applicationContext)
11    }
```

Listing 58: Método moveToLocation()

7.9. User Searcher Activity

A esta actividad se llega desde **MainActivity** 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.

Como se puede observar en la figura 48, 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 59, 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** 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 60 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 59 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.

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

Listing 59: 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 60: Clase Conversation dentro de nuestra aplicación

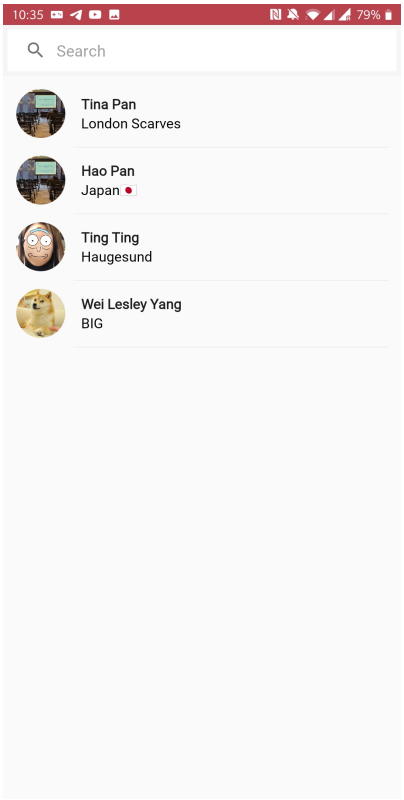


Figura 48: Captura de pantalla de UserSearcherActivity.

7.10. User Profile Activity

Esta actividad (Figura 49) 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 36).

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 49), si el usuario modifica dicho valor y posteriormente acepta los cambios, esto se verá reflejado en Firebase (Figura 50).

Si se observa el fragmento de código 61, se actualizaría la base de datos local y posteriormente en Firebase, esto se ha diseñado de esta manera para salvaguardar la consistencia de los datos.

7.11. 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** 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.

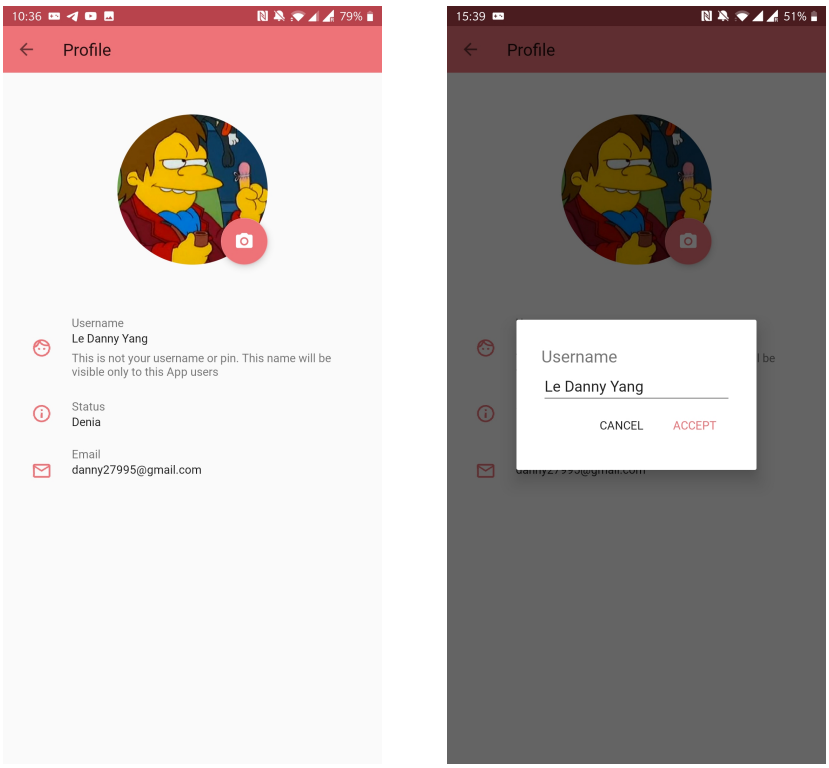


Figura 49: Actualización del nombre de usuario.

Figura 50: Diseñando la interfaz de la Actividad mediante código XML.

También a su vez, en esta actividad se permitirá al usuario borrar la lista de lenguajes debido a que cuando se selecciona 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 51 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.

```
fun FirebaseFirestore.updateCurrentUser(context: Context, user: User){
    collection(FirebaseUtil.FIREBASE_USER_PATH)
        .document(user.email)
        .set(user)
        .addOnSuccessListener {
            MyRoomDatabase.getMyRoomDatabase(context)?.updateUser(user)
            context.toast("User updated")
            FirebaseUtil.updateUser(user)
        }
        .addOnFailureListener {
            Log.d(LogUtil.TAG, "Error while updating user")
        }
}
```

Listing 61: Método updateCurrentUser() donde actualiza un usuario en la base de datos de Firebase

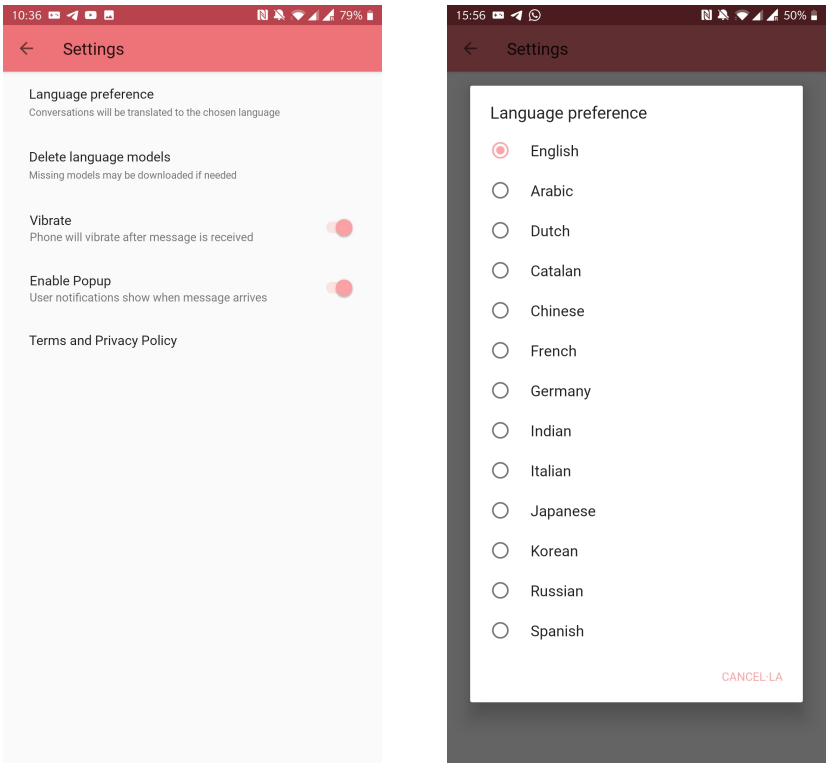


Figura 51: Listado de lenguajes permitidos.

8 Trabajo futuro

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

Para comenzar se enumera la lista de funcionalidades que se podrían haber añadido en un futuro a la aplicación:

1. **Envío de audio**: 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, en caso de tener la aplicación instalada en el dispositivo.
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 está funcionalidad puede ser de mucha utilidad para los programadores.
5. **Grupos de mensajería**: esta implementación se ha dejado de lado debido a que el objetivo del proyecto es crear una aplicación donde se puede 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 se ha desarrollado.

Por otra parte, otro aspecto no menos importante es el de la seguridad, en este proyecto no se ha implementado el cifrado de mensajes, por lo que se incumple la LOPD (Ley Orgánica de Protección de Datos) donde también se podría incluir como un trabajo futuro, tales como los ya mencionados anteriormente.

A falta de la seguridad, una vez implementadas todas las funcionalidades mencionadas, este proyecto podría ser libremente publicado en la Google Play Store y ser una aplicación lista para ser usada.

9 Conclusiones

Finalmente, en el aspecto de qué desarrollo es más conveniente a la hora de diseñar una aplicación de mensajería se puede decir que el desarrollo Nativo es, en este caso mejor, debido a que el objetivo del trabajo era desarrollar la aplicación para un sistema operativo.

Por otra parte, en el caso de 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 se ha escogido Kotlin como lenguaje principal debido a sus notables ventajas (Sección 3) que ofrecían de cara al desarrollo facilitando notablemente el trabajo.

En cuanto al resto de tecnologías, se puede decir que se han tomado las decisiones correctas al usar el nuevo patrón de diseño de Google (4.2) 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 5).

No obstante, hoy en día, se están mejorando las traducciones entre distintos idiomas a pasos agigantados, pero todavía queda un largo camino por recorrer. Cabe destacar, que es posible realizar aplicaciones de mensajería con dicha funcionalidad y una prueba de ello es el trabajo que se ha realizado con un tiempo de entrega establecido y con los recursos ínfimos con los que se contaban. 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 se han 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 trascurso del trabajo (Kotlin, MVVM, Firebase...) creo firmemente que este trabajo puede ser considerado como un éxito.

Bibliografía

[1] WhatsApp dice tener 2 000 millones de usuarios en el mundo https://www.elcomercio.com/tendencias/whatsapp-usuarios-activos-mundo-encriptacion.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