ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

DESARROLLO DE UNA APLICACIÓN MÓVIL EN ANDROID PARA LA ADQUISICIÓN DE GAS LICUADO DE PETRÓLEO (GLP) DE USO DOMÉSTICO

TRABAJO DE TITULACIÓN PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y TELECOMUNICACIONES

FRANK ANDRES ERAS CAMACHO

DIRECTOR: MSc. PABLO WILIAN HIDALGO LASCANO

Quito, noviembre 2020

AVAL

Certifico que el presente trabajo fue desarrollado por Frank Andres Eras Camacho, bajo mi supervisión.

_____

MSc. PABLO WILIAN HIDALGO LASCANO
DIRECTOR DEL TRABAJO DE TITULACIÓN

DECLARACIÓN DE AUTORÍA

Yo, Frank Andres Eras Camacho, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración dejo constancia de que la Escuela Politécnica Nacional podrá hacer uso del presente trabajo según los términos estipulados en la Ley, Reglamentos y Normas vigentes.

_____

FRANK ANDRES ERAS CAMACHO

DEDICATORIA

A mi madre, que gracias a su apoyo y amor incondicional me ha motivado siempre a seguir hacia delante.

Frank Eras C.

AGRADECIMIENTO

A mi persona, por siempre superarme y encontrar los medios para alcanzar los objetivos propuestos; por convertir mis miedos en combustible para seguir adelante y ver lo positivo en cada situación.

A mi madre Jenny por la paciencia, los consejos, el amor infinito y los sacrificios hechos para que yo sea una persona de bien. A Adriano, a quien quiero y considero como un padre; su constante soporte ha constituido un pilar fundamental de lo que soy ahora. A mis hermanas Adriana y Cristina, sabiendo que querrán seguir mis pasos, me han motivado a no rendirme en el camino para que ellas tampoco lo hagan.

A mi segunda familia: Paquita, Patricio y Fernanda, quienes me acogieron desde el primer día en su hogar y me han sabido tener paciencia a lo largo de este viaje. No alcanzan las palabras para expresar la gratitud que siento por todo el apoyo brindado.

A mi director de tesis, Ing. Pablo Hidalgo por su ayuda y guía en el desarrollo de este Trabajo de Titulación, los conocimientos y directrices impartidas permitieron el alcance de este objetivo.

A las amistades forjadas en la Escuela Politécnica Nacional, tanto en la carrera como fuera de esta, excelentes personas con las cuales no solo he compartido gratos momentos si no que me han ayudado a crecer tanto a nivel personal como profesional. Ha sido un honor poder haber recorrido este sendero en su compañía.

Frank Eras C.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGOS

RESUMEN

El desarrollo de aplicaciones móviles se encuentra en constante crecimiento, facilitando y mejorando la calidad de vida de las personas alrededor del mundo. En el diario vivir las aplicaciones móviles han permitido acceder y optimizar casi todos los servicios existentes, lo cual ha convertido al celular inteligente en un dispositivo indispensable.

Sin embargo, algunos servicios no cuentan con una aplicación móvil, como es el caso del acceso al Gas Licuado de Petróleo (GLP) para el uso doméstico, cuya distribución se realiza mediante carros transportadores de gas que circulan por la ciudad. La comercialización de esta manera genera inconvenientes como: congestión vehicular, contaminación sonora y dependencia de horarios de los distribuidores de gas. El presente trabajo desarrolla una aplicación que permite al cliente realizar un pedido de gas y ser atendido por el distribuidor autorizado más cercano en la zona, reduciendo el tiempo de espera del cliente y aumentando la eficiencia en la comercialización de gas.

El presente trabajo de titulación se encuentra dividido en cuatro capítulos, en el Capítulo 1 se desarrolla el marco teórico, en el cual se incluyen los conceptos necesarios y herramientas a usar para el desarrollo de la aplicación. En el Capítulo 2 se detalla el diseño de las interfaces, el desarrollo e implementación de la aplicación móvil. En el Capítulo 3 se especifican las pruebas de funcionamiento realizadas y el correspondiente análisis de los resultados. En el Capítulo 4 se presentan las conclusiones y recomendaciones obtenidas en el desarrollo de este proyecto. Finalmente, se incluyen como anexos el código completo desarrollado en Android Studio, un manual de usuario para que se facilite el uso de la aplicación y el instalador de la aplicación.

PALABRAS CLAVE: Android Studio, Firebase, Servicios de red, API de Google, GLP.

ABSTRACT

The development of mobile applications is constantly growing, facilitating, and improving the quality of life of people around the world. In our daily lives, mobile applications have made it possible to access and optimize almost all existing services, which has made the smart phone an indispensable device

However, some services do not have a mobile application, as is the case of access to Liquefied Petroleum Gas (LPG) for domestic use, the distribution of which is carried out by gas transporters that circulate through the city. The distribution in this way generates inconveniences such as: vehicular congestion, noise pollution and dependence on gas distributors' schedules. This work presents the development of an application that allows the customer to place an order of gas and be served by the nearest authorized distributor in the area, reducing the customer's waiting time and increasing the efficiency in gas commercialization.

This document is divided into four chapters; Chapter 1 develops the theoretical framework, which includes the necessary concepts and tools to use in the development of the application. Chapter 2 details the design of the interfaces, the development and implementation of the mobile app. Chapter 3 specifies the performance tests carried out and the corresponding analysis of the results. Chapter 4 presents the conclusions and recommendations obtained in the development of this project. Finally, the annexes include the complete code developed in Android Studio, a user manual to facilitate the use of the application, and the application installer.

KEY WORDS: Android Studio, Firebase, Network services, Google API

1. INTRODUCCIÓN

Actualmente los avances tecnológicos han producido cambios culturales, económicos y sociales, migrando servicios de alimentación, limpieza, distribución, entre otros, a plataformas virtuales y aplicaciones móviles para satisfacer las necesidades del cliente y adaptarse a los cambios del mercado.

Un gran porcentaje de personas, especialmente los jóvenes, desean realizar la mayor cantidad de tareas desde su teléfono móvil, sin embargo, no todos los servicios que se ofrecen se gestionan a través de una aplicación móvil. Un claro ejemplo es el acceso al Gas Licuado de Petróleo (GLP), cuya comercialización para el uso doméstico se realiza de dos formas: que el usuario se acerque a los diferentes centros de acopio del combustible o que se reabastezca mediante los carros transportadores de gas que circulan por la ciudad. La segunda opción es la más usada, existiendo en la provincia de Pichincha aproximadamente 400 distribuidores, entregando cerca de 30 000 cilindros diarios [1].

El problema con la distribución de GLP en camiones autorizados es la dependencia de los horarios y las rutas de los distribuidores móviles de gas, teniendo que esperar a que un distribuidor esté cerca del domicilio del cliente para poder obtener un cilindro de gas, haciendo difícil el acceso a cilindros de gas en función a la necesidad del cliente. Además, existe desperdicio en el consumo de combustible de los camiones al recorrer infructuosamente las calles de la ciudad, trayendo consecuencias como: congestión vehicular, contaminación sonora y contaminación del medio ambiente.

Por lo tanto, el proyecto planteado intenta solventar este problema al ofrecer al cliente acceso al gas de uso doméstico cuando lo necesite, mediante una aplicación móvil. Con esta aplicación se realiza un pedido y los distribuidores de gas de la zona recibirán la notificación en sus teléfonos móviles, pudiendo atender o rechazar dicha solicitud, mejorando la eficiencia en la distribución de GLP de uso doméstico.

En el presente capítulo se aborda un análisis de los fundamentos teóricos necesarios para el desarrollo de la aplicación.

1.1 OBJETIVOS

El objetivo general del presente Trabajo de Titulación es desarrollar una aplicación móvil en Android para la adquisición de Gas Licuado de Petróleo (GLP) de uso doméstico.

Los objetivos específicos son:

- Analizar la documentación y funcionalidad de las diferentes herramientas que serán usadas para el desarrollo de la aplicación.
- Diseñar la aplicación para distribución de gas con el uso de Android Studio.
- Implementar la aplicación en los dispositivos móviles.
- Verificar el correcto funcionamiento de la aplicación.

1.2 ALCANCE

El presente proyecto de titulación plantea el desarrollo de una aplicación móvil que sea capaz de mejorar el proceso de distribución de GLP doméstico, la cual estará instalada tanto en el celular del cliente como del distribuidor.

La aplicación móvil es desarrollada en Android Studio, y tiene la capacidad de:

- Crear nuevos usuarios, tanto clientes como distribuidores.
- Autenticar usuarios anteriormente registrados.
- Realizar un pedido de gas en tiempo real y que sea atendido por un distribuidor cercano que se encuentre en la zona.
- Gestionar la petición del número de cilindros de gas que el cliente requiere.
- Cancelar el pedido en caso de ser requerido.

La interfaz de la aplicación cuenta con un menú el cual permite la modificación de los datos del usuario, tales como nombres, dirección, teléfono y foto de perfil, y un botón de ayuda en el cual se detallará cómo funciona la aplicación.

La interacción entre el distribuidor y el cliente es posible al establecer una comunicación vía Internet, la misma que se logra mediante el uso de la plataforma de desarrollo Firebase, la cual permite el desarrollo de aplicaciones web y móviles en la nube. Firebase permite la creación de una instancia para el uso de *Cloud Storage*, en donde se almacenarán datos multimedia de los usuarios.

En la Figura 1.1 se muestran los componentes involucrados para el funcionamiento de la aplicación.

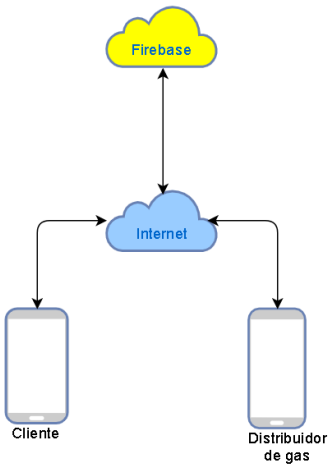

Figura 1.1 Estructura general de la aplicación

1.3 MARCO TEÓRICO

1.3.1 SISTEMA OPERATIVO ANDROID

Android es un sistema operativo móvil basado en código abierto y desarrollado por Google, que se encuentra disponible para celulares, *tablets, wearables*[1], relojes inteligentes entre otros, siendo utilizado actualmente en 2500 millones de dispositivos aproximadamente [2].

Android está basado en Linux, lo que significa que es un software de libre uso y permite a los usuarios trabajar sobre el código fuente con mínimas restricciones. Por esta razón, Android cuenta con una extensa comunidad de desarrolladores de aplicaciones, lo que permite ampliar la funcionalidad de los teléfonos móviles que cuentan con este sistema operativo.

1.3.1.1. Versiones del Sistema Operativo de Android

El sistema operativo de Android desde su lanzamiento en el año 2008 con Android 1.0 ha tenido varias actualizaciones hasta la actualidad, con Android 10, su última versión comercial y con una potencial versión Android 11 que se espera esté al alcance del público en general para finales del año 2020 [3]. Estas nuevas versiones del sistema operativo tienen como objetivo la corrección de errores de programa y la implementación de nuevas funcionalidades para los usuarios.

---

[1] *Wearables*: dispositivos tecnológicos que una persona incorpora a su cuerpo para interacción con el usuario u otro dispositivo.

A continuación, se realiza una pequeña reseña de algunas versiones del sistema operativo Android que se usó en este Trabajo de Titulación.

1.3.1.1.1 Android 4.0 Ice Cream Sandwich

Figura 1.2 Android 4.0 Ice Cream Sandwich [4]

Ice Cream Sandwich (ver Figura 1.2) fue lanzada en el año 2011 y es la novena versión del sistema operativo Android; a continuación se detallan sus características más importantes [4]:

- *Widgets*[2] disponibles con contenido en vivo.
- Corrector de ortografía mejorado.
- Capacidad de cerrar aplicaciones que consumen datos en segundo plano.
- Capacidad de agrupar notificaciones de una misma aplicación.
- Incorporación de tecnología NFC (Comunicación de campo cercano) para compartir contenido.

[2] *Widget*: son elementos que se usan en la personalización de la pantalla de inicio de dispositivos móviles. Su función es brindar información y permitir interactuar al usuario sin necesidad de ingresar a las aplicaciones asociadas.

1.3.1.1.2 Android 7.0 Nougat

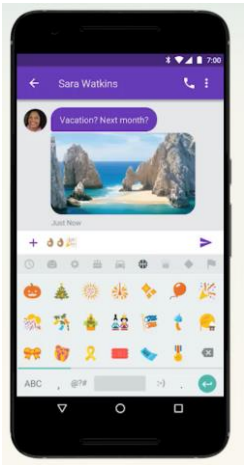

Figura 1.3 Android 7.0 Nougat [5]

Nougat (ver Figura 1.3), fue lanzada en el año 2016 y es la decimocuarta versión del sistema operativo Android. Se detallan a continuación las características más importantes [5]:

- Mejora en la velocidad de instalación de aplicaciones.
- Incorporación de la API (*Application Programming Interface*) Vulkan, con mejoras gráficas.
- Incorporación de multi ventana, permitiendo usar dos aplicaciones al mismo tiempo.
- Incorporación del modo DOZE, para el ahorro de batería del móvil.
- Incorporación de Daydream, plataforma de realidad virtual.

1.3.1.1.3 Android 8.0 Oreo

Figura 1.4 Android 8.0 Oreo [6]

Oreo (ver Figura 1.4), fue lanzada en el año 2017 y es la decimoquinta versión del sistema operativo Android. A continuación se detallan las características más importantes de esta versión [6]:

- Incorporación de íconos adaptativos.
- Incorporación de un sistema de activación automática de Wifi en redes conocidas.
- Mejora en el sistema de autocompletado.
- Mejora en la conectividad *Bluetooth*.
- Incorporación de "*Picture-in-Picture*", que permite visualizar dos aplicaciones al mismo tiempo.

1.3.1.2 Arquitectura Android

La arquitectura que emplea Android está compuesta de varias capas, las cuales usan los servicios que capas anteriores ofrecen y a su vez generan servicios para las capas superiores; se puede observar el *stack* de Android en la Figura 1.5.

Se describe la funcionalidad de cada capa a continuación:

- Kernel de Linux

  Se usa el núcleo de Linux como una capa de abstracción, es decir, para la interacción con los componentes de hardware de los dispositivos Android, esto mediante *drivers* necesarios que deben estar incluidos en el Kernel de Linux.

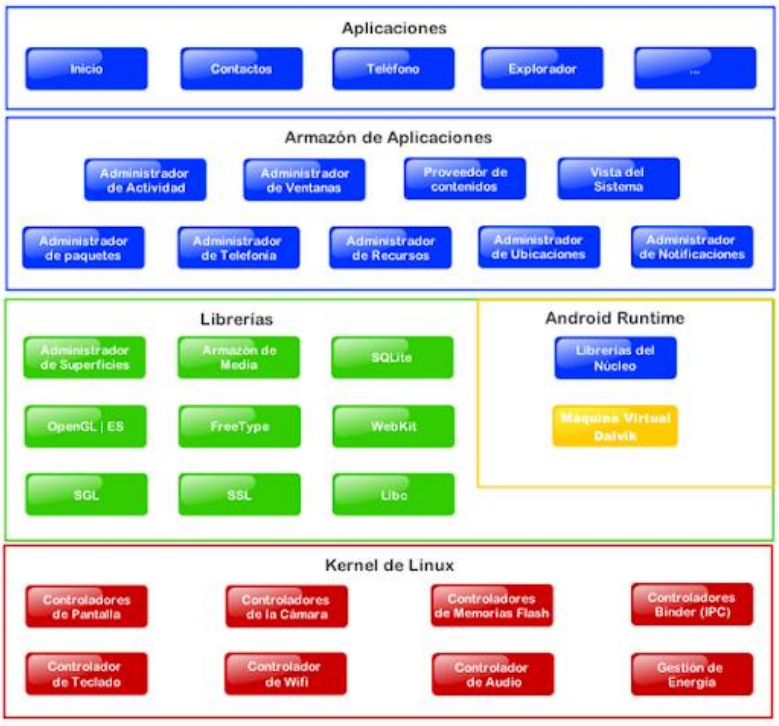

Figura 1.5 Arquitectura de Android [7]

- Librerías

Esta capa proporciona las librerías necesarias para la realización de determinadas tareas en el sistema Android; dichas librerías se encuentran escritas en lenguaje C/C++ y juntamente con el Kernel de Linux son la parte más importante de la arquitectura Android.

- Android Runtime

Se encuentra en la misma capa que las Librerías, y está constituida por varias librerías de *Core* escritas en lenguaje Java.

- Armazón de aplicaciones

Esta capa es la encargada de contener un conjunto de herramientas usadas para el desarrollo de las aplicaciones. Todas las aplicaciones que vayan a ser desarrolladas para Android usan un conjunto predeterminado de APIs (*Application Programming Interface*) que permiten acceder a diferentes servicios de Android.

- Aplicaciones

Esta capa contiene todas las aplicaciones del dispositivo, ya sean las propias de Android que se incluyen por defecto, o las aplicaciones que se instalan luego por parte de los usuarios en el dispositivo. Las aplicaciones usan los servicios, APIs y librerías generadas en las capas anteriores.

1.3.1.3 Android Studio

Android Studio es un IDE (*Integrated Development Environment*) o entorno de desarrollo integrado que fue diseñado por Google y es el encargado de proveer un editor de código y diferentes herramientas para el desarrollo de aplicaciones para dispositivos Android. Las principales características del IDE son las siguientes [8]:

- Sistema de compilación que tiene base en Gradle.
- Emulador rápido con diversidad de funciones.
- Entorno para desarrollo de cualquier dispositivo que cuente con Android.
- Aplicación para la inserción de cambios en el código sin necesidad de reiniciar la aplicación.
- Integración con la plataforma Github para lograr importar códigos ejemplo.
- Herramientas de Lint para la identificación de problemas de rendimiento, compatibilidad, entre otros.
- Compatibilidad con lenguajes como C++ y NDK.

- Compatibilidad con Google Cloud Plataform, para la integración con las plataformas Google Cloud Messaging y App Engine.
- Renderización en tiempo real.

Android Studio se encuentra disponible para varias plataformas, entre ellas: Windows 2003, 7, 8 y 10, GNU/Linux, macOS, etc. Para su instalación y correcto funcionamiento debe cumplir los requisitos del sistema, entre los cuales se destaca 4 GB de RAM mínimo, 4 GB de espacio en el disco duro para la aplicación y contar con la versión de Java JDK (*Java Development Kit*). En la Figura 1.6 se puede observar la interfaz gráfica que Android Studio ofrece a los desarrolladores.

1.3.2 JAVA

Java es un lenguaje de programación orientado a objetos e independiente de la plataforma hardware en la que se desarrolla, es decir, es multiplataforma; permite que el código desarrollado funcione en cualquier sistema operativo que tenga instalada la máquina virtual de Java [9].

Algunas de las principales características de este lenguaje de programación se mencionan a continuación [9]:

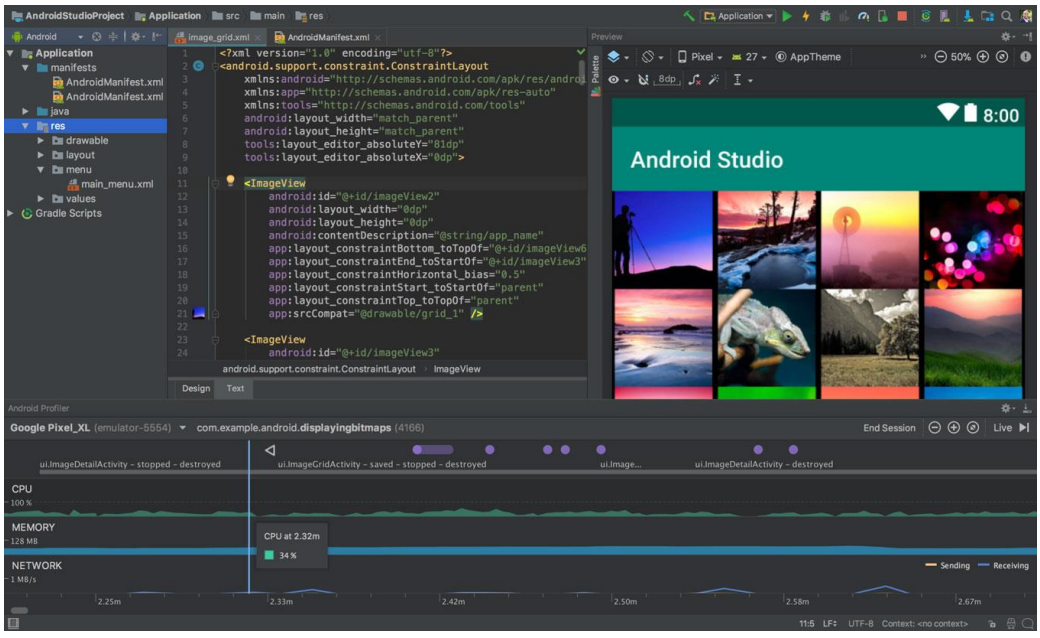

Figura 1.6 Interfaz gráfica de Android Studio [2]

- Es simple: fácil de leer, comprender y aprender.
- Es orientado a objetos: trabaja el código de los programas como si fueran objetos que se relacionan entre ellos para lograr el objetivo deseado.

- Independiente de la plataforma: puede ser ejecutado en cualquier dispositivo.
- Es multihilo: permite realizar varias tareas simultáneamente.
- Es seguro y sólido: suministra canales de comunicación para asegurar la privacidad de los datos.
- Recolector de basura: si un objeto no se está usando, Java lo borra para liberar memoria.

En la actualidad, Java es utilizado para programar páginas web, entre ellas destacan Facebook y Amazon debido a la robustez de su código. Además, al ser un lenguaje de programación de alto rendimiento y fácil de aprender, es bastante utilizado en el *Big Data* y en el desarrollo de aplicaciones para la nube.

1.3.3 FIREBASE

Firebase es una plataforma digital creada por Google utilizada en el desarrollo de aplicaciones móviles; proporciona funciones tales como: estadísticas, bases de datos, informes de mensajería, informes de fallas, haciendo que la aplicación a la que va dirigida sea más eficiente [10].

Los recursos con los que cuenta actualmente esta plataforma incluyen [10]:

- *Cloud Firestore:* permite la sincronización y almacenamiento de datos de las aplicaciones a nivel global.
- *ML Kit:* permite el aprendizaje automático dirigido para desarrolladores de aplicaciones móviles.
- *Cloud Functions:* permite crear funciones que son activadas en base a cambios en los datos, nuevos registros o eventos analíticos en los dispositivos móviles.
- *Authentication:* permite la autenticación de usuarios con un método sencillo y seguro.
- *Hosting:* permite la implementación de páginas web de destino para las aplicaciones móviles.
- *Cloud Storage:* permite el almacenamiento y procesamiento de contenido multimedia como fotos y videos en la nube.
- *Realtime Database:* permite el almacenamiento de información y la sincronización de ésta en milisegundos.

Firebase además de brindar las herramientas para el desarrollo de las aplicaciones permite el soporte para diferentes sistemas operativos, entre ellos: Android e iOS, esto a través de las APIs de Google.

1.3.3.1 Autenticación Firebase

Es un servicio que ofrece la plataforma de desarrollo de Firebase que facilita la creación de sistemas de autenticación para aplicaciones en dispositivos móviles o sitios web; este servicio permite mejorar la experiencia del usuario en la integración y acceso a un aplicativo, al proporcionar una solución de autenticación directa, personalizable y de código abierto para el acceso de los usuarios [11].

En las soluciones de autenticación que se puede escoger hay diferentes opciones: autenticación telefónica, mediante cuentas de correo electrónico, cuentas de Google, Twitter, Facebook entre otras. La Figura 1.7 muestra todas las opciones de registro e inicio de sesión que se pueden implementar en las aplicaciones a desarrollar.

1.3.3.2 Base de datos en Tiempo Real de Firebase

Este servicio de Firebase representa una base de datos NoSQL que se encuentra alojada en la nube para el almacenamiento y sincronización de datos de usuarios en tiempo real, pudiendo ser actualizada desde cualquier dispositivo web o móvil [12].

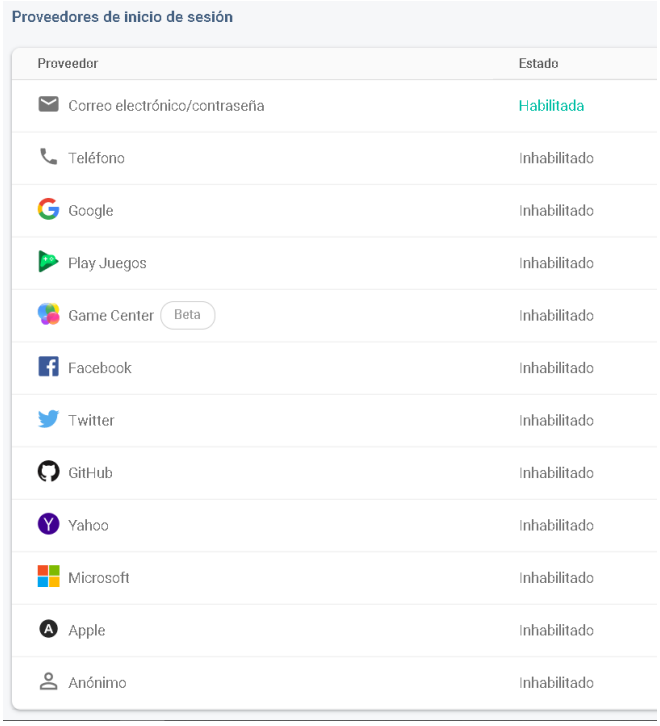

Figura 1.7 Opciones para autenticación en Firebase [11]

Esta base de datos cuenta con numerosas ventajas, entre las que se pueden destacar las siguientes:

- Creación de aplicaciones sin necesidad de servidores.
- Optimizada para usar aún sin tener conexión a Internet.
- Integración con Autenticación de Firebase para proporcionar seguridad.
- Acceso desde cualquier parte del mundo al tener conexión a Internet.

1.3.3.3 Cloud Storage

Cloud Storage es una de las herramientas con las que cuenta Firebase para el almacenamiento y publicación de contenido multimedia generado por usuarios, como fotos o videos. Este servicio almacena los archivos en un depósito de Google Cloud Storage, a los cuales se puede acceder a través de Firebase o de Google Cloud, permitiendo subir o descargar archivos de clientes móviles para luego poder realizar procesamiento en el servidor, tal como filtrado de imágenes o transcodificación de video [13].

A continuación, se describen algunas de las funciones claves de Cloud Storage para Firebase [13]:

- Operaciones robustas: Los SDK (*Software Development Kit*) de Firebase para Cloud Storage permiten subir o descargar archivos independientemente de la calidad de la red, reiniciando la operación en el punto en el cual se interrumpió.
- Seguridad sólida: Los SDK de Firebase son integrados con las funciones de autenticación de Firebase para simplificar los procesos.
- Gran escalabilidad: Cloud Storage es fácilmente escalable a niveles de exabytes de ser necesario, pasando de la fase de prototipo a la fase de producción con sencillez.

Con Cloud Storage es posible agregar reglas para que solo ciertos usuarios tengan acceso al contenido disponible, implementando los servicios de autenticación de Firebase para discriminar la visualización de contenido dependiendo del usuario, asegurando confidencialidad en los datos de los usuarios. En la Figura 1.8 se puede observar una representación de cómo se accede a la información almacenada en Cloud Storage, permitiendo solo a los dispositivos autenticados el acceso al contenido deseado.

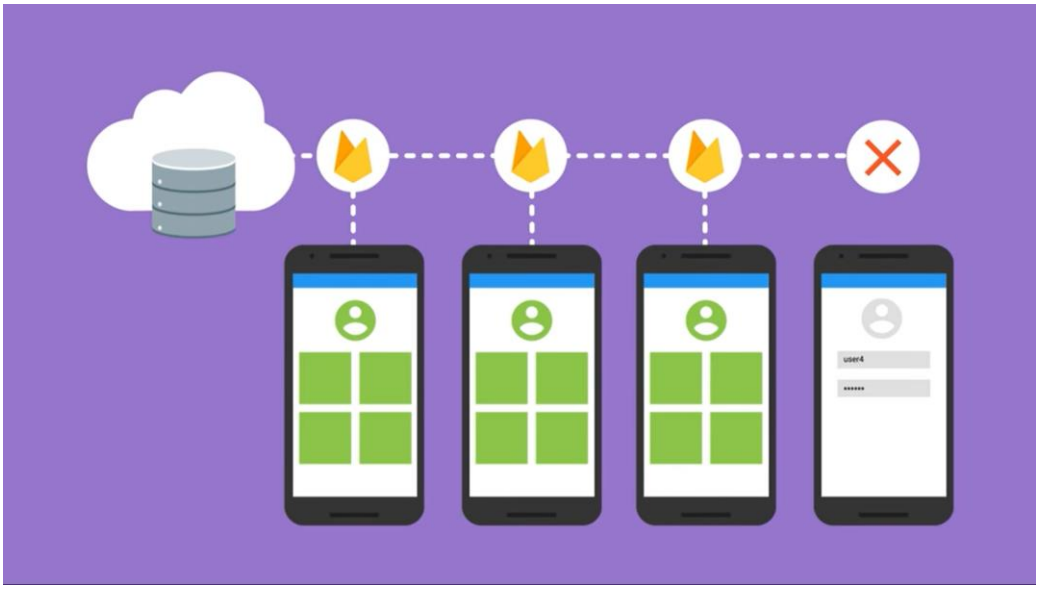

Figura 1.8 Representación de acceso al contenido en Cloud Storage [13]

1.3.4 GEOFIRE

GeoFire es una biblioteca de código abierto de Javascript que permite el almacenamiento y la consulta de datos, en función de la ubicación geográfica de un elemento. Es decir, realizar una consulta de los usuarios (distribuidores) que están dentro de un rango o cerca de una dirección origen (cliente). Sin embargo, lo que hace realmente atractiva a esta biblioteca de código abierto es la posibilidad de realizar consultas en tiempo real [14].

El funcionamiento de GeoFire se basa en la utilización de la base de datos en tiempo real de Firebase, que permite actualizar los resultados de las consultas realizadas en fracciones de milisegundos conforme los datos cambian. Es decir, si un usuario cambia su localización, las nuevas coordenadas se actualizarán automáticamente y en tiempo real en la consulta realizada. El beneficio es que carga selectivamente solamente los datos que se encuentran cerca de determinadas ubicaciones, con lo que se mantiene a las aplicaciones que usan esta librería, ligeras, incluso teniendo bases de datos de gran tamaño.

1.3.5 API DE GOOGLE

Se conoce como *Application Programming Interface* (API) a un conjunto de código que permite la integración y comunicación de los servicios de Google con aplicaciones a desarrollar. Estás APIs simplifican la programación y aumentan la base de usuarios con una serie de herramientas para la creación de nuevas apps [15]. Las API de Google son bastante populares entre los desarrolladores debido a que permiten usar funciones

existentes para otros software, evitando el volver a escribir un código funcional probado con anterioridad.

Es necesario tener una cuenta de desarrollador en Google para poder hacer uso de las APIs, las cuales al ser activadas generan un *token* de programador[3] único para el proyecto en el cual serán usadas. Cada vez que la aplicación necesite hacer uso de los servicios de una API, se realiza una solicitud *HTTPS* que permite la comunicación entre los servidores de Google con la aplicación registrada, siendo este proceso totalmente transparente para el usuario de la aplicación.

1.3.5.1 Maps SDK para Android

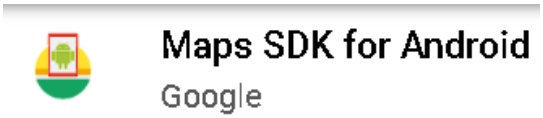

Figura 1.9 Logo Maps SDK para Android [16]

Esta API permite al desarrollador agregar mapas a la aplicación Android, basados en los datos existentes en Google Maps (ver Figura 1.9). La API se encarga de acceder a los servidores de Google automáticamente para lograr la descarga de los datos, visualización de datos y obtener respuesta a los gestos que el usuario realice sobre los mapas. Adicionalmente, es posible integrar objetos a los mapas, tales como marcadores, polilíneas, polígonos, superposiciones de mapas de bits anclados y superposiciones de mosaicos [16].

1.3.5.2 API de geocodificación

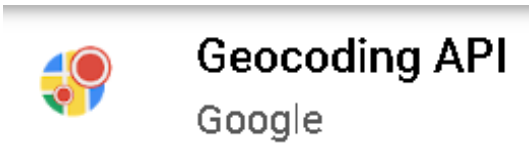

Figura 1.10 Logo de la API de geocodificación [17]

Esta API provee el servicio de geocodificación (ver Figura 1.10), es decir, convertir direcciones en coordenadas geográficas, usualmente longitud y latitud, para un mejor

[3] *Token* de programador: es una combinación exclusiva de letras, números y caracteres que permite identificar una actividad en las APIs.

manejo de los mapas. Adicional, tiene la función de geocodificación en reversa, la cual transforma coordenadas en direcciones fácilmente legibles para las personas [17].

1.3.5.3 API de geolocalización

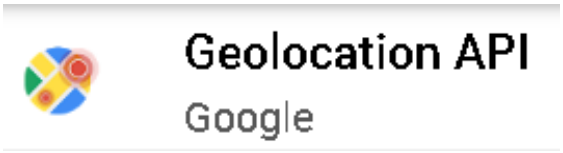

Figura 1.11 Logo API de geolocalización [18]

La API de geolocalización (ver Figura 1.11) retorna la ubicación del celular con un radio de precisión bastante alto; la localización del usuario es obtenida en base a la información de las torres celulares y los nodos Wi-fi a los que el móvil se conecta [18].

1.3.5.4 API de direcciones

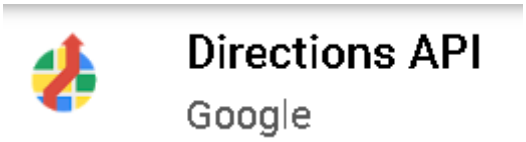

Figura 1.12 Logo API de direcciones [19]

La API de direcciones (ver Figura 1.12) ofrece un servicio que calcula las indicaciones para llegar de un punto a otro en un mapa. Permite obtener las rutas entre diferentes ubicaciones dependiendo del método de transporte, pudiendo ser transporte público, conduciendo, en bicicleta o caminando [19].

1.3.6 PLATAFORMA DE GOOGLE MAPS

Google ofrece entre sus múltiples herramientas de desarrollo, la plataforma Google Maps, la cual consta de 3 principales ramas: mapas, rutas y lugares. Cada una de estas opciones cuenta con características únicas que ayudan en el desarrollo de una aplicación móvil y cuentan con compatibilidad con sistemas operativos Android e iOS, haciendo fácil su implementación al añadir la API deseada al programa principal de una aplicación.

Los 3 productos que ofrece *Google Maps Plataform* son [20]:

- **Maps:** permite acceder a mapas estáticos y dinámicos, imágenes de *Street View* y vistas en 360º. Además, permite personalizar los mapas con marcadores, imágenes, colores y líneas .

- **Rutas:** permite a un usuario llegar de un punto a otro en un mapa con indicaciones y actualizaciones de tráfico en tiempo real, estableciendo rutas eficientes para optimizar el sistema a implementar.
- **Lugares:** permite a los usuarios explorar el mundo con información de más de 100 millones de sitios, pudiendo encontrar dichos sitios mediante direcciones, números de teléfono y señales en tiempo real.

1.3.7 HTTP (HYPERTEXT TRANSFER PROTOCOL)

HTTP o Protocolo de Transferencia de Hipertexto, es un protocolo de aplicación con el cual es posible realizar peticiones de información a través de la web. Es conocido como un protocolo cliente-servidor, puesto que un elemento (cliente) realiza una petición de recursos y otro elemento (servidor) contesta con una respuesta y los datos solicitados [21].

HTTP es un protocolo sencillo, fácilmente interpretable para los desarrolladores de programación, en el cual existen dos tipos de mensajes: *peticiones* y *respuestas.* Una petición está compuesta por un método HTTP que define la operación que el cliente va a realizar, usualmente GET o POST; la dirección del recurso que se pide, pudiendo ser una URL; la versión del protocolo HTTP, 1.1 o 2; y finalmente cabeceras HTTP con información adicional, las cuales pueden ser opcionales.

Por otro lado, una respuesta está compuesta por: la versión del protocolo que se está utilizando, un código de mensaje que sirve como indicador si la petición tuvo éxito o no, un mensaje de estado que describe el código de estado, cabeceras HTTP similares a las de las peticiones y finalmente el recurso solicitado, cuyo envío puede ser opcional [21].

2. METODOLOGÍA E IMPLEMENTACIÓN

En este capítulo se aborda la descripción del IDE de Android Studio y el desarrollo de la aplicación móvil para dispositivos con sistema operativo Android. Se realiza una explicación de los pasos a seguir para la creación del proyecto en Android Studio, la creación de la base de datos en Firebase y posterior implementación de los servicios en la aplicación móvil.

La aplicación fue creada en conjunto con el lenguaje de programación Java, la plataforma de desarrollo Firebase y los servicios de base de datos, autenticación y almacenamiento multimedia. Además, se hace uso de las APIs de Google con sus correspondientes librerías y dependencias. Los diferentes elementos que interactúan con la aplicación móvil pueden ser observados en la **Figura 2.1** Diagrama de estructura de la aplicación móvil

.

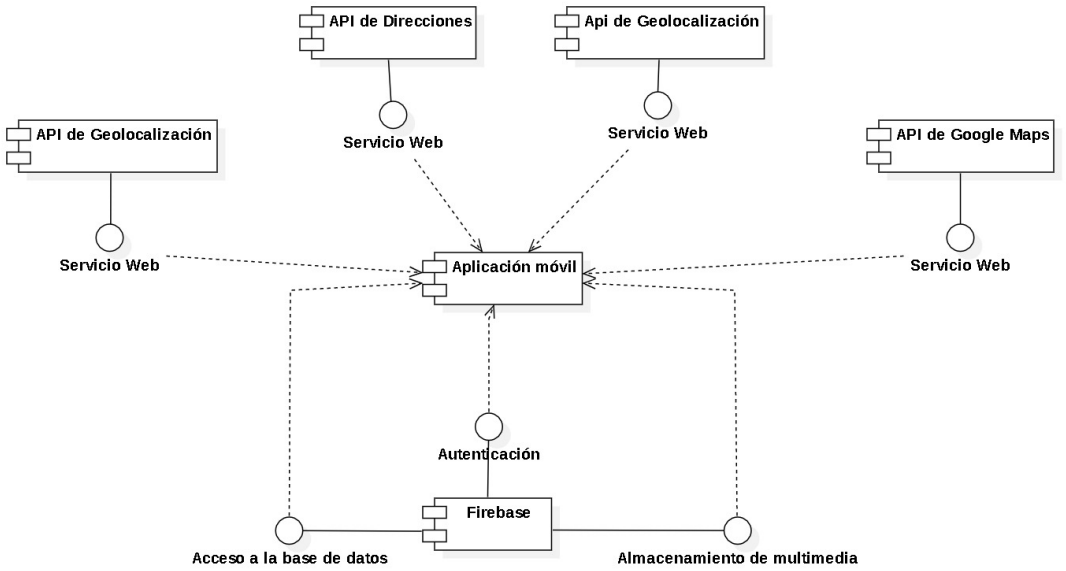

Figura 2.1 Diagrama de estructura de la aplicación móvil

2.1 REQUERIMIENTOS DE LA APLICACIÓN

Mediante entrevistas verbales a posibles usuarios, tanto clientes como distribuidores de gas de uso doméstico, y ha diversas conversaciones con el director del trabajo de titulación, se ha obtenido una lista de requerimientos funcionales y no funcionales que la aplicación debe cumplir para su implementación.

2.1.1 REQUERIMIENTOS FUNCIONALES

- Permitir al cliente realizar un pedido.
- Permitir que el sistema registre los datos de nuevos usuarios.
- Permitir que el sistema autentique las credenciales del usuario.
- Permitir al distribuidor ser notificado de un nuevo pedido.
- Permitir al distribuidor aceptar o rechazar un pedido.
- Permitir tanto al cliente como al distribuidor cancelar un pedido en progreso.
- Mostrar al cliente los detalles del pedido y la información del distribuidor que atenderá el pedido.
- Mostrar al distribuidor los detalles del pedido y la información del cliente que solicitó el pedido.
- Mostrar el historial de todos los pedidos en los que el usuario ha estado involucrado.
- Mostrar a detalle cada uno de los pedidos en los que el usuario ha estado involucrado.
- Acceder a los servicios de ubicación del móvil del usuario.

2.1.2 REQUERIMIENTOS NO FUNCIONALES

- Permitir que el sistema se conecte automáticamente a Internet.
- Permitir al usuario realizar cambios de los datos personales.
- Acceder a la base de datos de Firebase.
- Presentar un tiempo de respuesta relativamente rápido.

2.2 CREACIÓN DEL PROYECTO EN ANDROID STUDIO

Para la creación de la aplicación, es necesario contar con el programa de Android Studio previamente instalado en la computadora a trabajar, el cual puede ser descargado de la página oficial y ser instalado siguiendo las instrucciones que se detallan en la guía de usuario. Una vez instalado el IDE de Android Studio, la creación de un nuevo proyecto puede ser realizada mediante los pasos que se especifican en el Anexo D de este trabajo de titulación.

2.3 CREACIÓN DEL PROYECTO EN FIREBASE

La incorporación de los servicios de Firebase es indispensable para el funcionamiento de la aplicación móvil, por lo cual es necesario disponer de una cuenta de Google para el ingreso a la plataforma como se observa en la Figura 2.2.

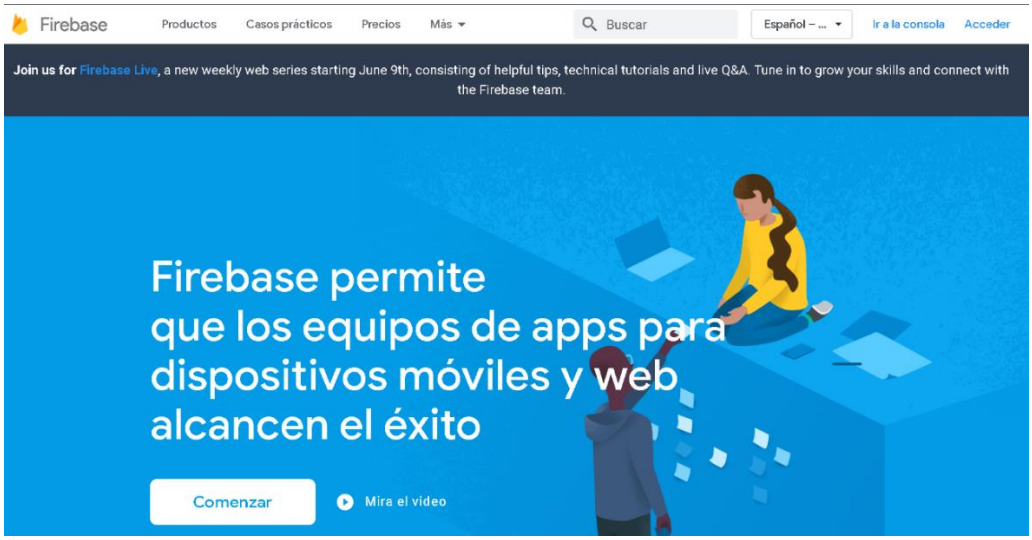

Figura 2.2 Página de inicio de Firebase

Luego de autenticadas las credenciales de Google e ingresar a Firebase, se presiona el botón *Ir a la consola*, el mismo que carga una nueva página en la que es posible visualizar los proyectos que al momento se tiene y crear nuevos de ser necesario, tal como se observa en la Figura 2.3.

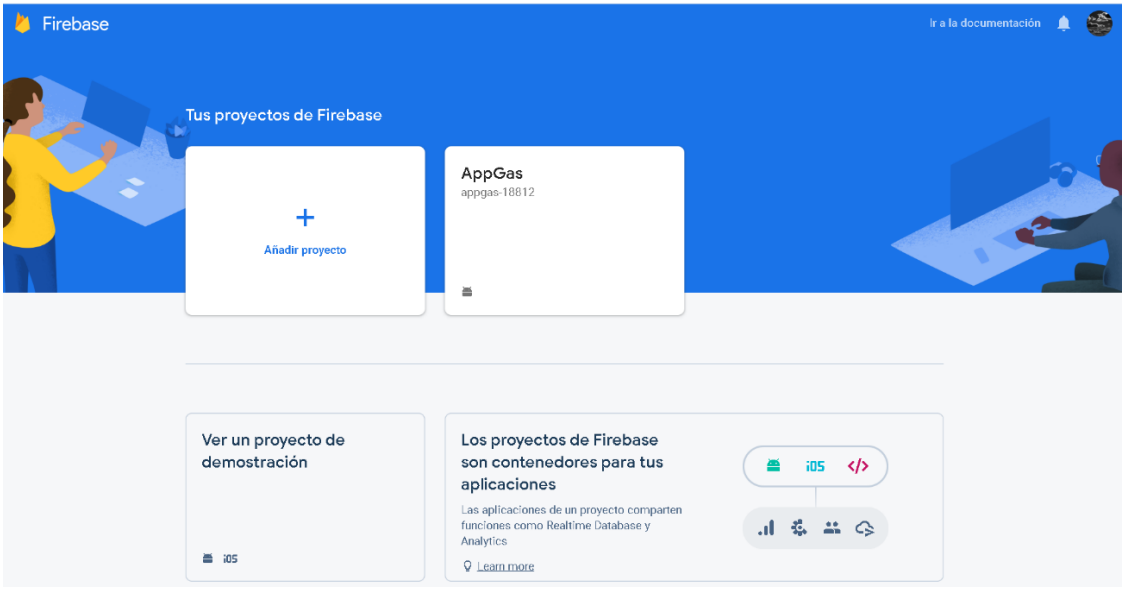

Figura 2.3 Proyectos del usuario de Firebase

La creación de un nuevo proyecto es posible al dar clic en *Añadir proyecto,* el cual desplegará algunos pasos a seguir. Será necesario ingresar la información requerida, siendo el primer paso la asignación del nombre del proyecto, como se muestra en la Figura 2.4.

Figura 2.4 Nombre del nuevo proyecto

Luego es necesario decidir si se realizará la activación de Google Analytics, la cual es una función de Google que permite a la plataforma realizar informes, predicciones o segmentaciones de los usuarios automáticamente, para mejorar la eficiencia de Firebase. Como se muestra en la Figura 2.5, se activa o desactiva esta opción y luego se procede con un clic en *Continuar*.

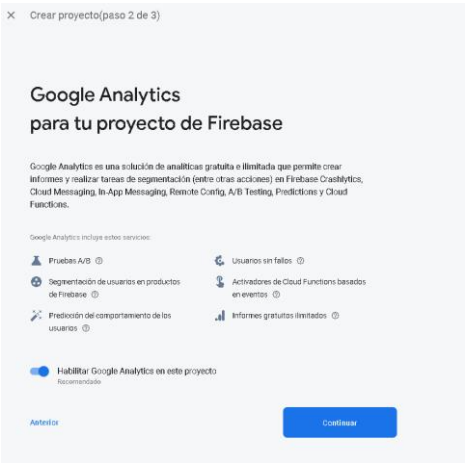

Figura 2.5 Activación de Google Analytics

Finalmente se selecciona o se crea una cuenta de Google Analytics para poder almacenar los datos recogidos durante el funcionamiento de la aplicación. Al seleccionar la cuenta se da clic en *Crear Proyecto* como se muestra en la Figura 2.6.

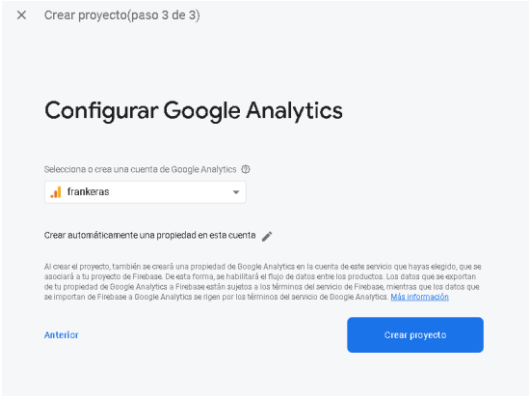

Figura 2.6 Selección de cuenta de Google Analytics

La creación del proyecto tomará unos segundos, luego de los cuales se informará que está listo para continuar, tal como se observa en la Figura 2.7.

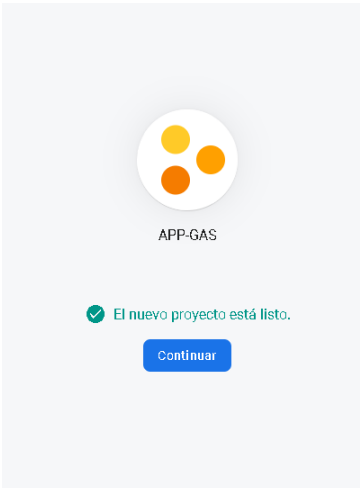

Figura 2.7 Creación del proyecto en Firebase

Al dar clic en *Continuar* se dará paso a la pantalla principal de la Consola de Firebase, en la cual estarán disponibles todas las funcionalidades de Firebase que se pueden implementar. Como se observa en la Figura 2.8, a la izquierda de la pantalla se tienen las herramientas de Desarrollo, Calidad y Analytics para el proyecto, mientras en el centro de la pantalla se observa las plataformas en las cuales es posible implementar Firebase.

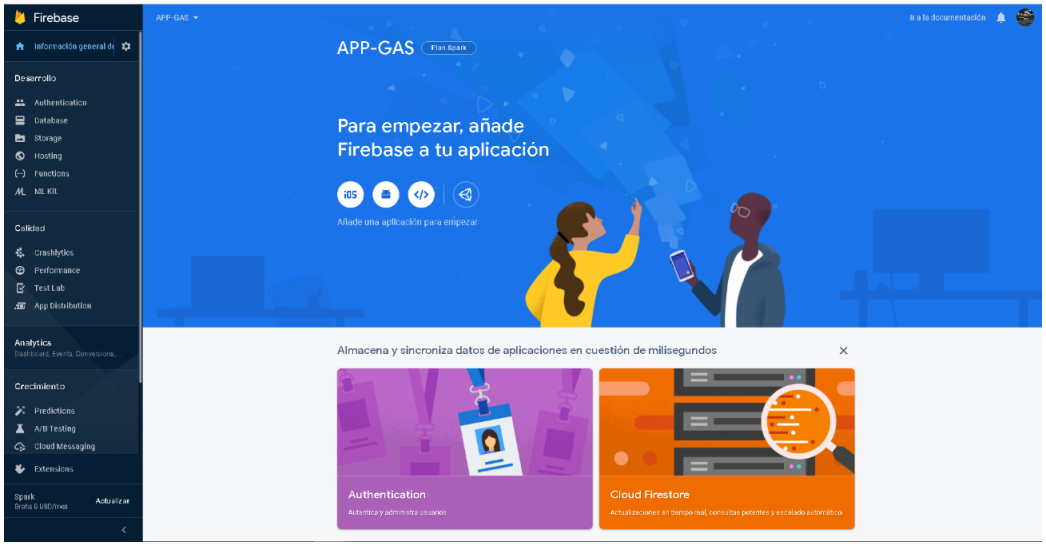

Figura 2.8 Consola de Firebase para un proyecto

Se selecciona el ícono de Android, que es la plataforma móvil en la que se desarrolla la aplicación. Se desplegará una nueva interfaz, como se muestra en la Figura 2.9, en la cual se pide registrar la aplicación que se vinculará a este proyecto, incluyendo el nombre del paquete de Android, apodo para nuestra aplicación (opcional) y el certificado de firma SHA-

1 (opcional). Dicha información puede ser encontrada en el archivo build.gradle (Module: app) en nuestro proyecto de Android Studio.

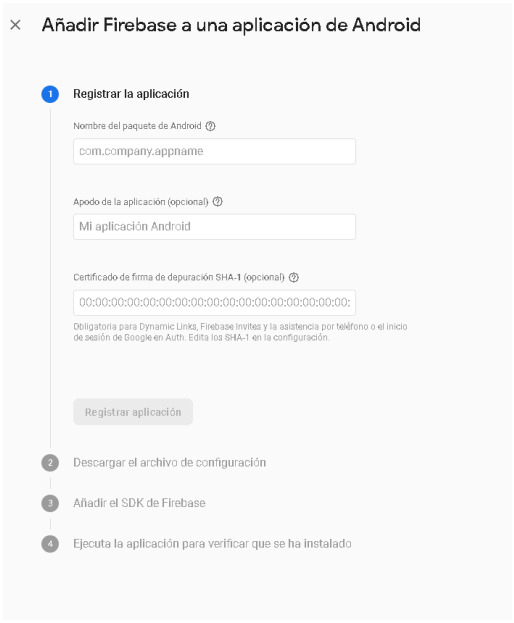

Figura 2.9 Información para añadir Firebase a una aplicación móvil

Una vez registrada la información, Firebase generará un archivo de configuración en formato .json que debe ser descargado y posteriormente agregado al proyecto en Android Studio, como se indica en las Figura 2.10 y 2.11.

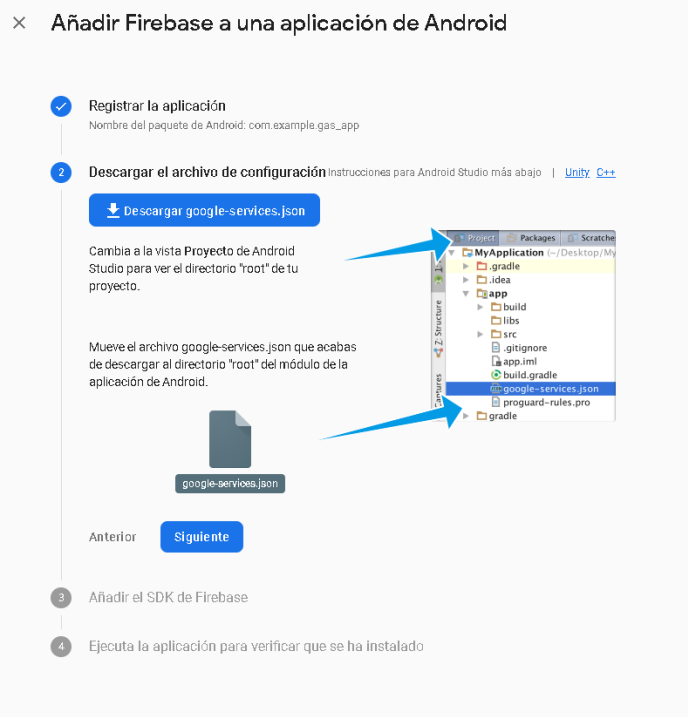

Figura 2.10 Generación de archivo de configuración

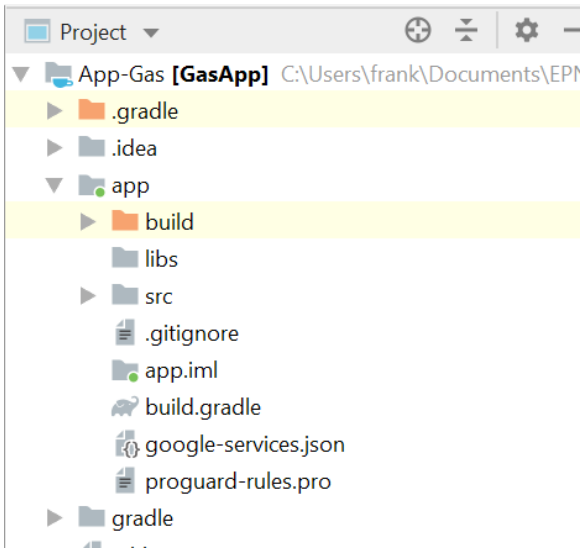

Figura 2.11 Archivo de configuración de Firebase en el proyecto en Android Studio

Luego que el archivo de configuración ha sido agregado al proyecto de Android Studio, se da clic en *Siguiente* para proceder a añadir el SDK de Firebase a la aplicación. Para poder hacer uso del archivo de configuración añadido anteriormente, es necesario modificar los archivos build.gradle en Android Studio. En primer lugar, hay que abrir al archivo *build.gradle (Project: GasApp)* y se añaden las líneas de código como se observa en la Figura 2.12.

Build.gradle de nivel de proyecto (<proyecto>/build.gradle):

```
buildscript {
  repositories {
    // Check that you have the following line (if not, add it):
    google()  // Google's Maven repository
  }
  dependencies {
    ...
    // Add this line
    classpath 'com.google.gms:google-services:4.3.3'
  }
}

allprojects {
  ...
  repositories {
    // Check that you have the following line (if not, add it):
    google()  // Google's Maven repository
    ...
  }
}
```

Figura 2.12 Dependencias de Firebase a nivel de proyecto

Adicionalmente, se debe abrir en Android Studio el archivo *build.gradle (Module: app)* y añadir las líneas de código conforme se indica en la Figura 2.13, las cuales son necesarias para la habilitación de los servicios de Firebase a nivel de aplicación.

Build.gradle de nivel de aplicación (<proyecto>/<app-module>/build.gradle):

```
apply plugin: 'com.android.application'
// Add this line
apply plugin: 'com.google.gms.google-services'

dependencies {
  // add the Firebase SDK for Google Analytics
  implementation 'com.google.firebase:firebase-analytics:17.2.2'
  // add SDKs for any other desired Firebase products
  // https://firebase.google.com/docs/android/setup#available-libraries
}
```

Figura 2.13 Dependencias de Firebase a nivel de aplicación

Finalmente, se procede a sincronizar el proyecto en Android Studio presionando *Sync Now,* que se encuentra en la parte superior derecha del archivo *build.gradle*. Una vez sincronizado el proyecto, los cambios realizados surtirán efecto y el SDK de Firebase estará listo, pudiendo hacer uso de los servicios de Firebase para el desarrollo de la aplicación móvil.

2.3.1 SERVICIO DE AUTENTICACIÓN DE USUARIOS DE FIREBASE

Una vez añadidos los servicios de Firebase a la aplicación, la implementación de éstos al proyecto de Android Studio es relativamente fácil e intuitiva. Uno de estos servicios es *Firebase Authentication,* el cual permite el uso de otras plataformas para el registro y posterior autenticación de usuarios en la aplicación. En la Figura 2.14, se observan algunas de las formas en las que se puede iniciar sesión en Firebase, las cuales se pueden habilitar o inhabilitar según el método de acceso que el desarrollador escoja.

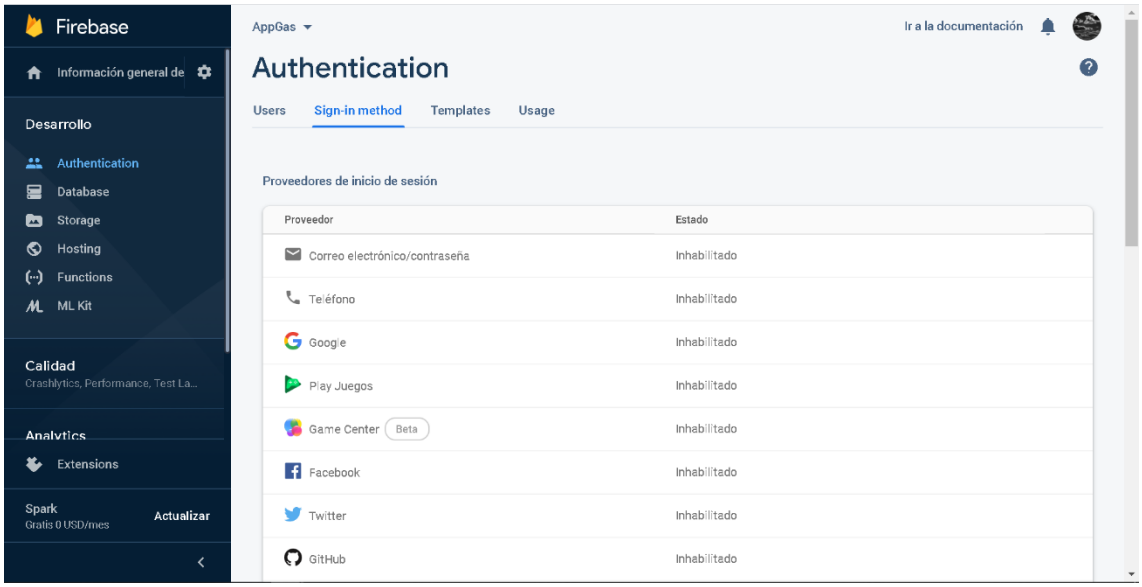

Figura 2.14 Proveedores de registro en Firebase para aplicaciones

La aplicación usará el método de autenticación de correo/contraseña, por lo que se procede a habilitar esta opción de autenticación. Como se observa en la Figura 2.15, el primer *switch* debe estar en color azul y luego se presiona *Guardar* para completar la habilitación de este servicio.

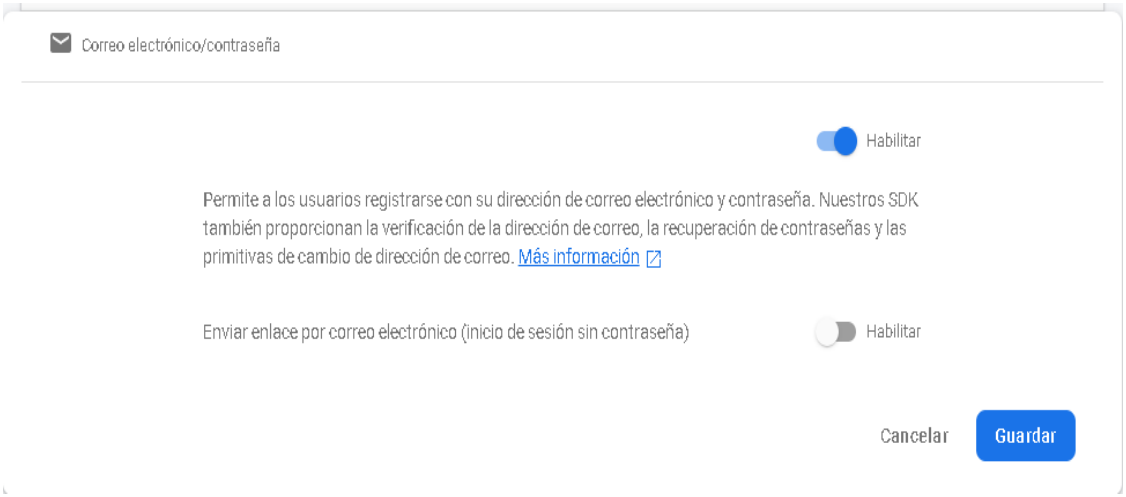

Figura 2.15 Habilitación del servicio de autenticación de Firebase

Adicional, para poder hacer uso del servicio es necesario añadir la biblioteca de Android para autenticación de Firebase; las diferentes bibliotecas pueden encontrarse en la documentación en la página oficial de Firebase. Para añadir esta biblioteca se debe abrir el archivo *build.gradle (Module: app)* y en la sección de dependencias añadir el código que se muestra en el Código 2.1.

```
implementation 'com.google.firebase:firebase-auth:18.0.0'
```

Código 2.1 Implementación de la biblioteca de Autenticación de Firebase

Finalmente se sincroniza el archivo *build.gradle (Module: app),* para que los cambios tengan efecto y el servicio pueda ser utilizado en la aplicación.

2.3.2 BASE DE DATOS EN TIEMPO REAL DE FIREBASE

El servicio de base de datos en tiempo real que ofrece Firebase servirá para almacenar a los usuarios y su información, pero también para manejar el sistema de pedidos y despachos de gas. Para activar este servicio es necesario dirigirse a la sección *Database* de la consola de Firebase, como se observa en la Figura 2.16, se despliegan las opciones para base de datos que están disponibles, de las cuales se dará clic en *Realtime Database* para su creación.

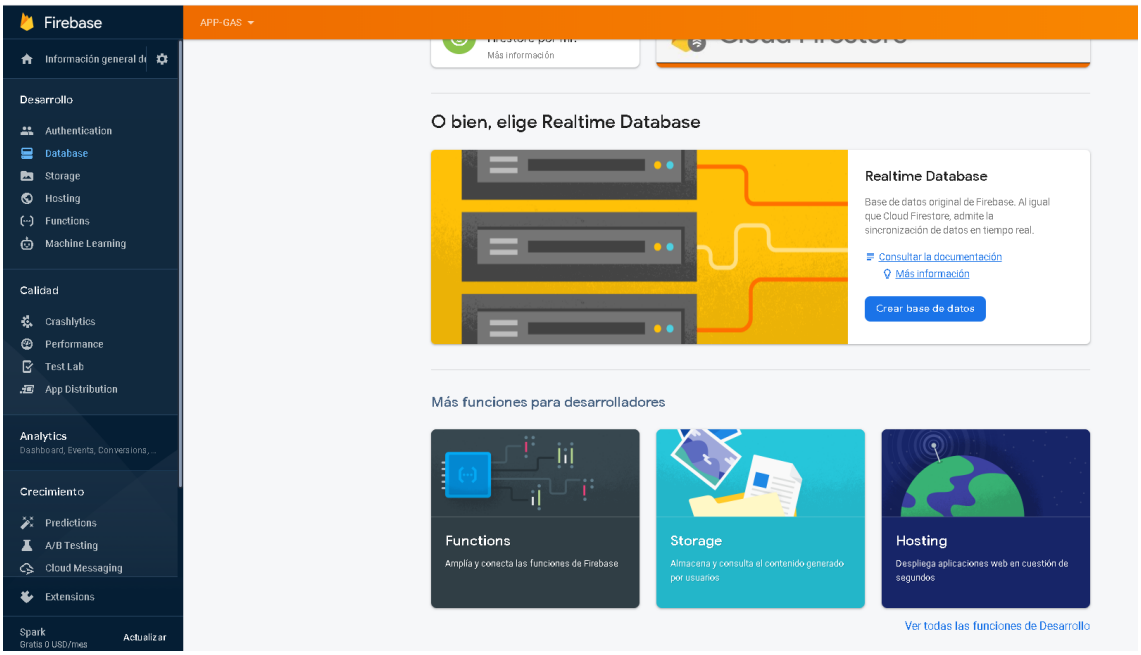

Figura 2.16 Opciones de base de datos en Firebase

Al presionar *Crear base de datos,* aparecerá una nueva ventana en la que pide definir las reglas de seguridad de la base de datos en tiempo real. En la Figura 2.17, se pueden observar las dos opciones que se tiene para la base de datos; la primera no permite la lectura o escritura de ningún usuario, solamente el administrador, mientras la segunda permite que los usuarios que estén ligados a este proyecto puedan hacer uso de la base de datos. Se escoge la segunda opción y se presiona el botón *Habilitar.*

Figura 2.17 Reglas de seguridad para la Base de Datos de Firebase

Al crear la base de datos aparecerá una pantalla como la mostrada en la Figura 2.18, en la que se observa la raíz de la base de datos con el identificador de la aplicación: *app-gas-4a185: null*, a partir de la cual es posible crear nuevos objetos que se almacenarán en forma de árbol de nodos.

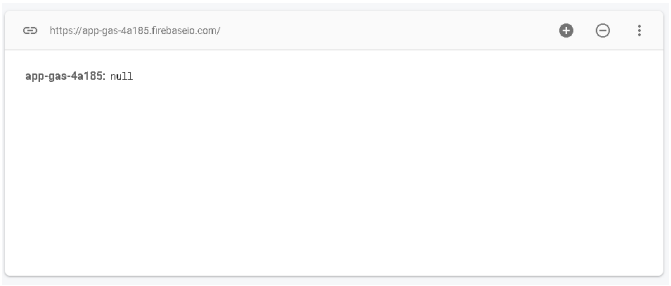

Figura 2.18 Base de datos por defecto de Firebase

Para la aplicación que se desarrolla como primer elemento se crea un nodo *Usuarios,* dentro del cual se subdividirá en *Clientes* y *Distribuidores,* para los dos roles que un usuario puede tener dentro de esta aplicación. Como se puede observar en la Figura 2.19, se crea la base de datos, dentro de la cual se crearán usuarios dependiendo del rol que escojan al registrarse cuando accedan por primera vez a la aplicación.

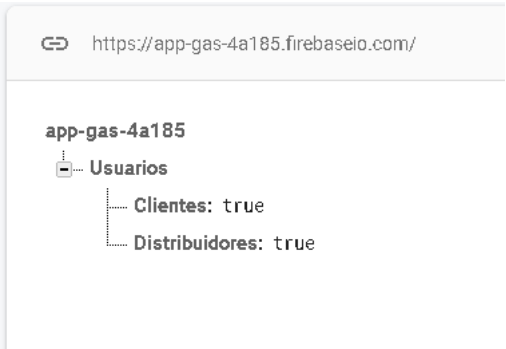

Figura 2.19 Base de datos en tiempo real de Firebase

Adicional, para poder hacer uso del servicio es necesario añadir la biblioteca de Android para el manejo de la base de datos en tiempo real de Firebase. Para añadir esta biblioteca se debe abrir el archivo *build.gradle (Module: app)* y en la sección de dependencias añadir el código que se muestra en el Código 2.2.

```
implementation 'com.google.firebase:firebase-database:18.0.0'
```

Código 2.2 Biblioteca para base de datos de Firebase

Finalmente se sincroniza el archivo *build.gradle (Module: app),* para que los cambios tengan efecto y el servicio pueda ser utilizado en la aplicación.

2.4 IMPLEMENTACIÓN DE GEOFIRE

Esta biblioteca es bastante útil para realizar las consultas de ubicación en tiempo real con la plataforma Firebase; permite entre otras cosas, al cliente, obtener la dirección del distribuidor de gas más cercano. Para su uso se debe instalar la dependencia en el archivo *build.gradle (Module: app),* insertando el código mostrado en el Código 2.3, finalmente se sincroniza el archivo *build.gradle,* para que los cambios surtan efecto.

```
implementation 'com.firebase:geofire-android:3.0.0'
```

Código 2.3 Dependencia para el uso de GeoFire

Adicional, en la base de datos en tiempo real se debe agregar un nuevo nodo para registrar los distribuidores de gas disponibles; dentro se distinguirá entre los diferentes distribuidores basados en el ID asignado al momento de registrarse en la aplicación. Como se observa en la Figura 2.20, el *ID_distribuidor* representa el ID único que tiene un distribuidor en la aplicación y los nodos hijos *Latitud* y *Longitud* representan la ubicación actual de dicho distribuidor.

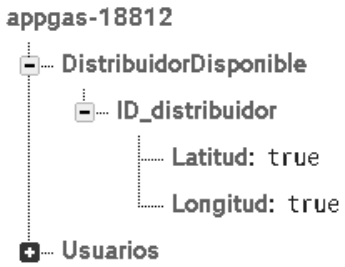

Figura 2.20 Base de datos actualizada

2.5 IMPLEMENTACIÓN DE APIS DE GOOGLE

Las APIs que ofrece Google para el desarrollo de aplicaciones son numerosas y se adaptan a las funcionalidades que se necesiten en una app. Para la implementación de una API a una aplicación de Android Studio es indispensable contar con una cuenta de Google, la cual es necesaria para acceder a la consola de desarrolladores de Google, que puede ser encontrada en la página oficial de Google.

Al ingresar por primera vez a la consola, se deben aceptar los términos y condiciones como se muestra en la Figura 2.21. En la siguiente ventana, se debe presionar *Crear Proyecto* para continuar con el proceso.

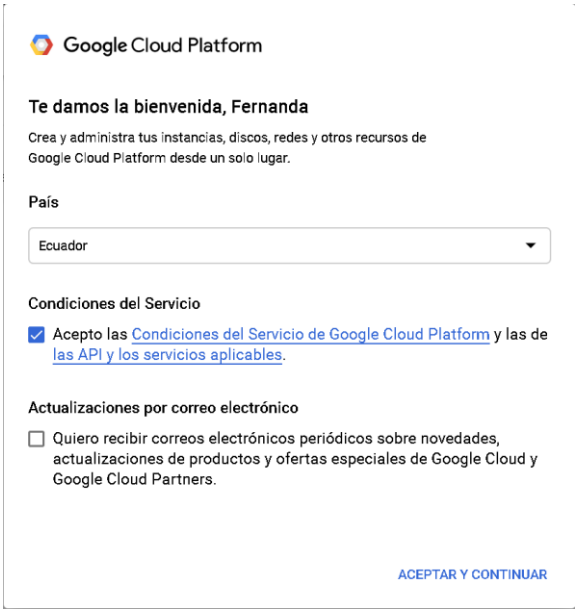

Figura 2.21 Condiciones de Servicio para el uso de la Consola de Desarrolladores de Google

Como se muestra en la Figura 2.22, es necesario asignar un nombre y una ubicación al proyecto, esta última se asigna por defecto en “Sin organización”; al finalizar se presiona *Crear* para finalizar la creación del nuevo proyecto.

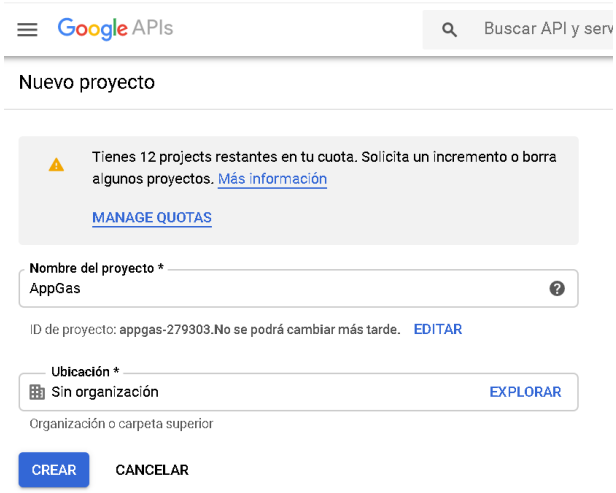

Figura 2.22 Creación de un nuevo proyecto en Google Desarrolladores

Una vez creado el proyecto, se procede a habilitar las APIs y servicios necesarios para la aplicación. En la Figura 2.23, se muestran algunas de las APIs que Google ofrece y que pueden ser añadidas a un proyecto.

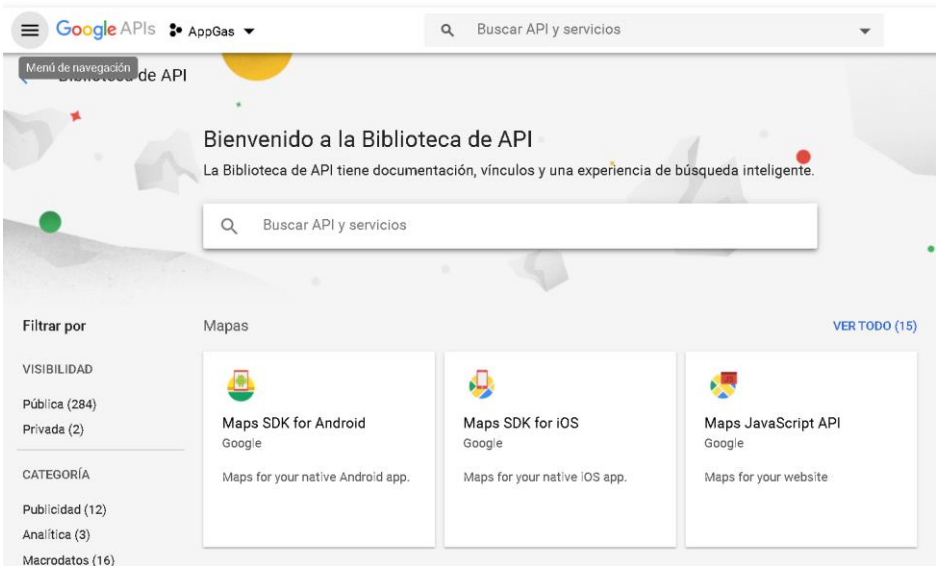

Figura 2.23 Biblioteca de APIs de Google Desarrolladores

2.5.1 GOOGLE MAPS API

El acceso a los Mapas que ofrece Google es factible mediante la instalación de las dependencias y librerías de Google Maps en Android Studio. Dentro de la Biblioteca de API se selecciona “Maps JavaScript API” y se da clic en *Habilitar*, para proveer los servicios que la aplicación en Android necesita, incluyendo mapas en tiempo real que pueden ser personalizados de acuerdo con el desarrollador. Además es necesario la creación de credenciales para poder ligar el proyecto de Google con Android Studio, por lo que en el menú de la consola de desarrolladores se selecciona credenciales y luego *Crear credenciales,* para obtener una clave de API, tal como se muestra en la Figura 2.24.

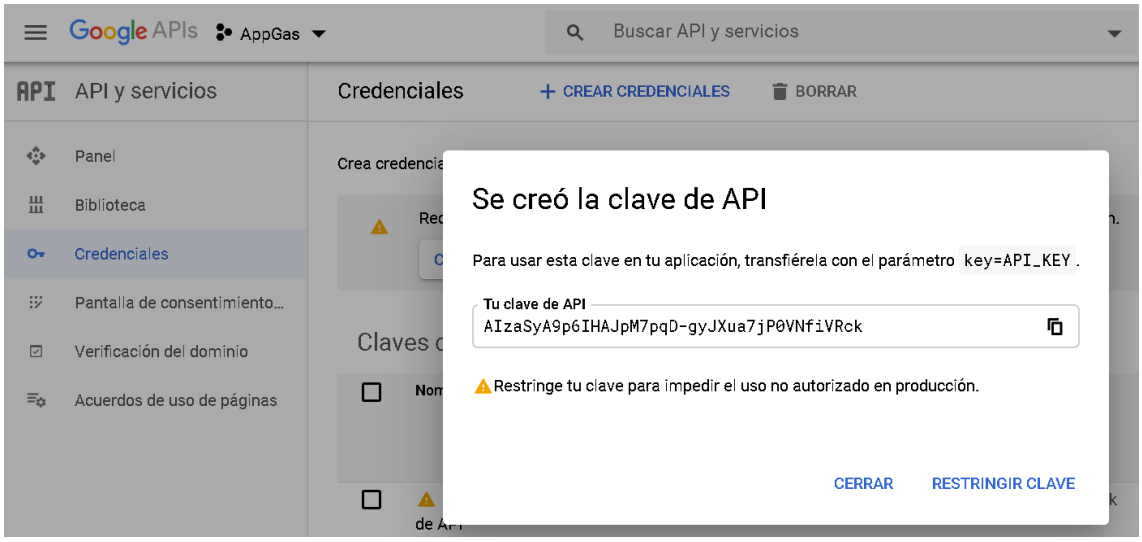

Figura 2.24 Creación de credenciales de API de Google

Para el soporte de Maps en el desarrollo de la aplicación es necesario crear una nueva actividad destinada exclusivamente para ello; Android Studio ofrece un módulo exclusivo para Google Maps, como se observa en la Figura 2.25, el cual contiene el código base para poder mostrar los mapas de Google.

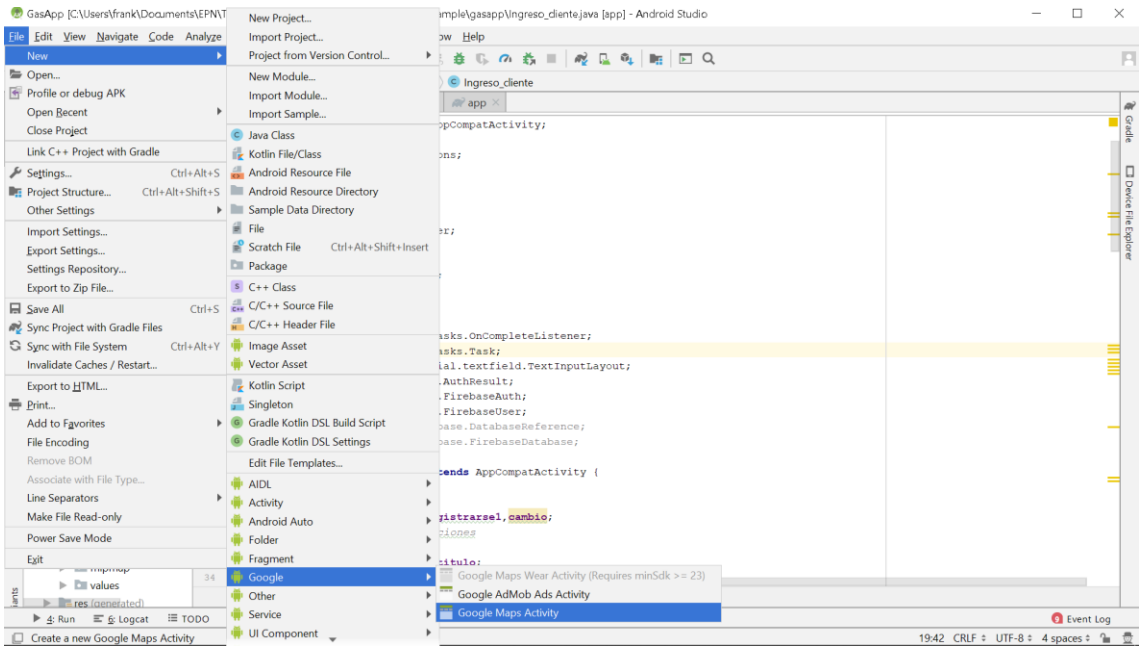

Figura 2.25 Creación de la actividad para soporte de Google Maps

Además, para hacer uso del servicio de Google Maps se debe enlazar el proyecto creado con Android Studio; esto se logra incluyendo la clave API generada anteriormente en la consola de desarrolladores de Google en el archivo *google_maps_api.xml,* que se encuentra en la carpeta **values** del módulo res. En la Figura 2.26, se muestra la clave ingresada en el archivo antes mencionado.

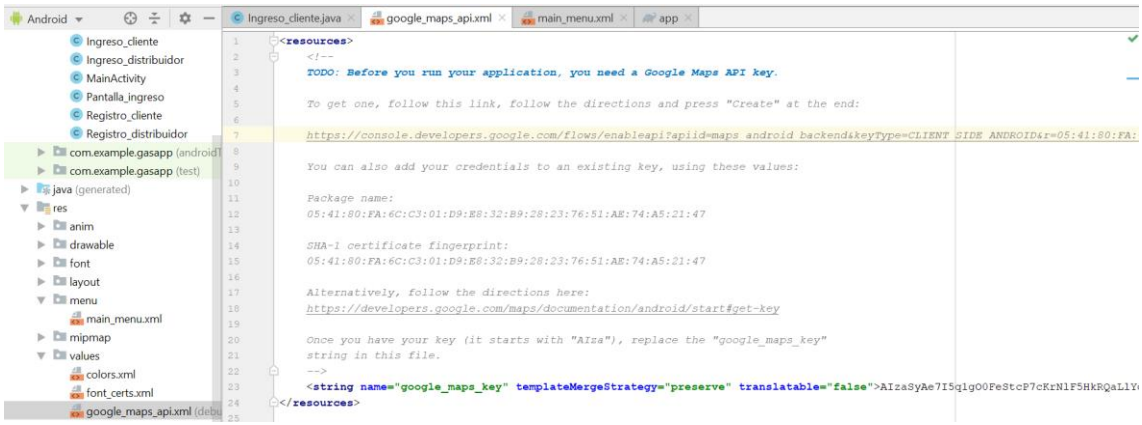

Figura 2.26 Enlace de Google Maps con Android Studio

Adicionalmente, las dependencias deben estar instaladas para el soporte de los mapas; la dependencia se añade en el archivo *build.gradle (Module: app),* ingresando el código

mostrado en el Código 2.4. Finalmente se sincroniza el archivo *build.gradle,* para que los cambios surtan efecto.

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'com.google.android.gms:play-services-maps:16.1.0'
```

Código 2.4 Dependencia para Google maps

2.5.2 API DE GEOCODIFICACIÓN Y GEOLOCALIZACIÓN

Para hacer uso de estas APIs hay que acceder a la consola de desarrolladores de Google y habilitarlas en la biblioteca de API. En la Figura 2.27, se muestra la interfaz donde es posible habilitar la API de Geocodificación y en la que se muestra la habilitación de la API de geolocalización (ver Figura 2.28).

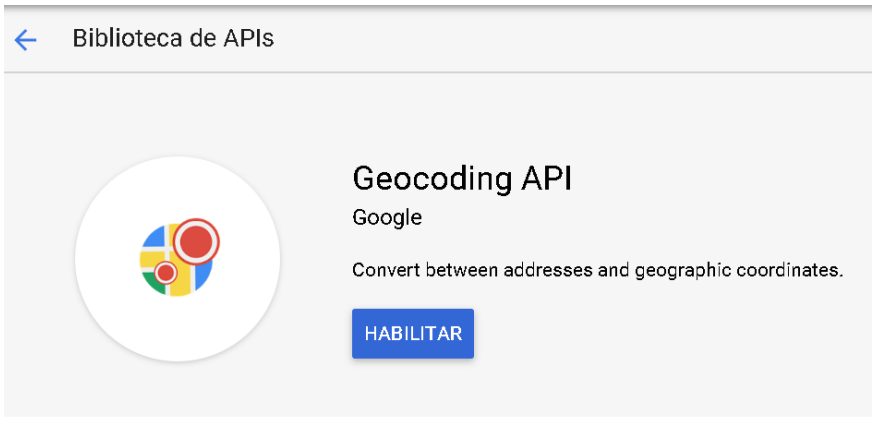

Figura 2.27 Habilitar API de Geocodificación

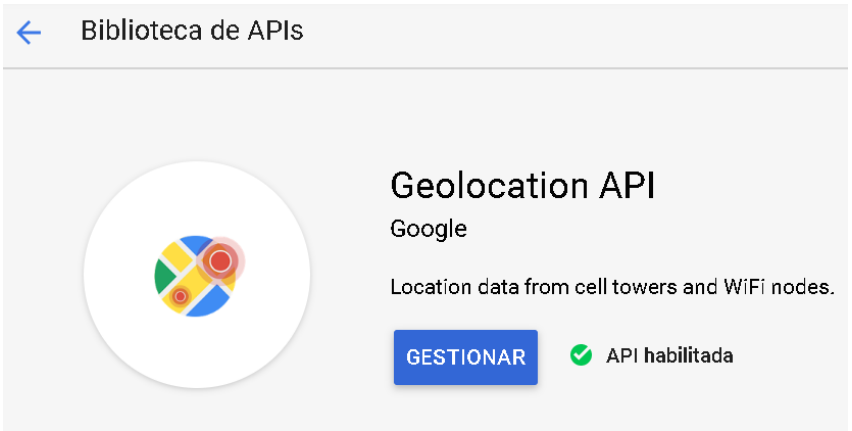

Figura 2.28 Habilitar API de Geolocalización

La implementación de estas APIs implica que la aplicación deba acceder a la ubicación del móvil del usuario, por lo cual se deben tener los permisos para hacer uso del servicio; estos permisos deben ser agregados en el archivo *AndroidManifest.xml,* con el código que se muestra en el Código 2.5.

```
<!-- Permisos para acceder a la localización del teléfono, para internet, para el estado del teléfono,
para vibración del teléfono.-->
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Código 2.5 Permisos para acceso a la localización del móvil

Además, las dependencias deben estar instaladas para el soporte de los servicios de localización en Android Studio; la dependencia se añade en el archivo *build.gradle (Module: app),* con el código mostrado en el Código 2.6. Finalmente se sincroniza el archivo *build.gradle,* para que los cambios surtan efecto.

```
implementation 'com.google.android.gms:play-services-location:17.0.0'
```

Código 2.6 Dependencia para acceso a ubicación del usuario

2.5.3 API DE DIRECCIONES

La API de direcciones se implementa mediante la consola de desarrolladores de Google. En la Figura 2.29, se muestra la interfaz donde es posible habilitar la API de Direcciones.

Figura 2.29 Habilitar API de direcciones

Además, las dependencias deben estar instaladas para el soporte de los servicios, la cual se añade en el archivo *build.gradle (Module: app),* con el código mostrado en Código 2.7. Finalmente se sincroniza el archivo *build.gradle,* para que los cambios surtan efecto.

```
apply plugin: 'com.google.gms.google-services'
```

Código 2.7 Dependencia para soporte de servicio de Direcciones

2.6 DISEÑO DE LA INTERFAZ DE USUARIO

La aplicación se encuentra compuesta de varias interfaces según la actividad que el usuario vaya a realizar; el desarrollo de las diferentes interfaces se realiza con Android Studio y usando el lenguaje XML. A continuación se aborda cada interfaz disponible en la aplicación.

2.6.1 INTERFAZ INICIO

Es la pantalla inicial mostrada al ingresar a la aplicación, se compone de un *splash screen*, que contiene el logo y título de la aplicación de manera animada; esta pantalla tiene una duración de 4 segundos luego de lo cual el usuario pasa a la siguiente interfaz. En la Figura 2.30, se puede apreciar la interfaz de inicio.

Figura 2.30 Interfaz de inicio

2.6.2 INTERFAZ SELECCIÓN DE ROL

En esta interfaz el usuario debe escoger el rol que tendrá en la aplicación, pudiendo ser cliente o distribuidor. En la Figura 2.31, se puede observar el diseño realizado, el cual consta de dos botones, para escoger el rol a desempeñar. Al presionar cualquiera de los botones, se envía al usuario a la siguiente interfaz dependiendo del rol escogido.

Figura 2.31 Interfaz de selección de rol

2.6.3 INTERFAZ INGRESO CREDENCIALES

Es la interfaz para el ingreso de credenciales del usuario, consta de 5 elementos interactivos: dos casillas de texto y tres botones. Las casillas de texto sirven para el ingreso de *email* y contraseña del usuario; mientras el botón *Ingresar,* para avanzar a la pantalla principal de la aplicación, el botón *Nuevo Usuario? Registrarse,* para cambiar a la pantalla de registro si es un nuevo usuario, y el último botón que varía según el rol del usuario.

Si el usuario es cliente, el botón refleja *Soy distribuidor,* para moverse a la pantalla de ingreso del distribuidor, en caso de que el usuario haya accedido erróneamente en la interfaz del distribuidor, como se puede observar en la Figura 2.32. Por otro lado, si el usuario es distribuidor, el botón refleja *Soy cliente,* para moverse a la pantalla de ingreso del cliente, en caso de que el usuario haya accedido erróneamente como se observa en la Figura 2.33.

Figura 2.32 Interfaz ingreso cliente

Figura 2.33 Interfaz ingreso distribuidor

2.6.4 INTERFAZ REGISTRO DE UN USUARIO

Es la interfaz para realizar el registro de nuevos usuarios en la aplicación, en la cual se pueden ingresar los datos necesarios para poder hacer uso de la app. Cuenta con varias casillas de texto para ingresar información, un botón tipo imagen que al ser presionado brinda la opción de agregar una foto de perfil que se encuentre en el dispositivo para que sea almacenada junto a la información del usuario; y finalmente el botón *Registrarse* para terminar el proceso y avanzar a la pantalla principal de la aplicación.

En la Figura 2.34, se muestra la interfaz para el cliente, que cuenta con las casillas para ingresar: nombre y apellido, número de teléfono, correo electrónico y contraseña; mientras en la Figura 2.35, se muestra la interfaz del distribuidor, la cual permite el ingreso de: nombre y apellido, teléfono, placa del vehículo que utiliza para realizar los despachos de pedidos, *email* y contraseña.

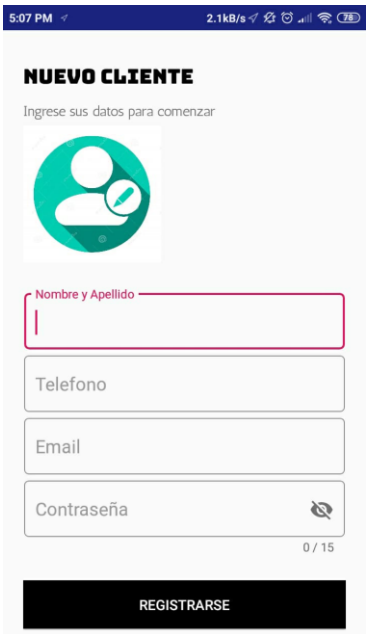

Figura 2.34 Interfaz registro cliente

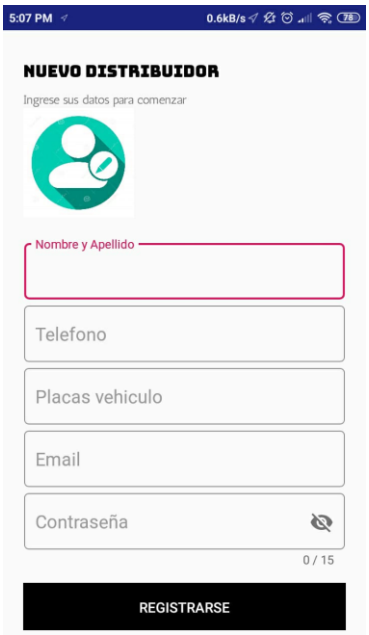

Figura 2.35 Interfaz registro distribuidor

2.6.5 INTERFAZ PRINCIPAL DEL CLIENTE

Es la interfaz principal de la aplicación para un usuario que ha ingresado como cliente, desde la cual se puede realizar el pedido de cilindros de gas hacia los distribuidores. En la Figura 2.36, se observa el diseño de la interfaz, en la que se encuentra configurado un mapa con la ubicación actual del cliente; en la parte inferior se muestran dos casillas de texto, la de la izquierda muestra la dirección del pedido cuando el cliente presiona un punto en el mapa, mientras que en la de la derecha se puede ingresar el número de cilindros de gas deseados. Finalmente se tiene el botón *Realizar pedido,* el cual al ser presionado sube

el pedido a la plataforma Firebase para que pueda ser atendido por los distribuidores cercanos en la zona.

Figura 2.36 Interfaz principal cliente

2.6.6 INTERFAZ PRINCIPAL DEL DISTRIBUIDOR

Es la interfaz principal de la aplicación para un usuario que ha ingresado como distribuidor, desde la cual se puede recibir pedidos de cilindros de gas de los clientes. En la Figura 2.37, se observa el diseño de la interfaz, en el que se encuentra configurado un mapa con la ubicación actual del distribuidor; en la parte superior se tiene un botón tipo *Switch* que permite el cambio de estado del conductor entre conectado y desconectado. Finalmente se tiene una casilla de texto: *Esperando pedido,* la cual tiene la función informativa que cambiará de estado al recibir un pedido de gas.

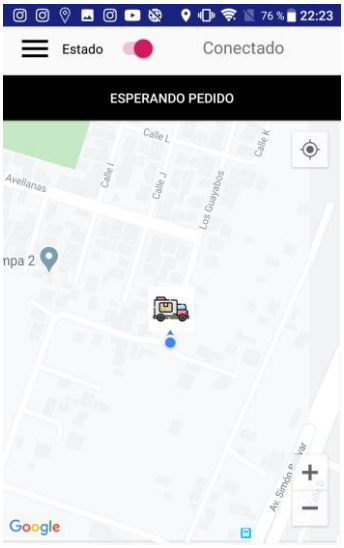

Figura 2.37 Interfaz principal distribuidor

2.6.7 INTERFAZ MENÚ LATERAL

Es un menú lateral al cual se tiene acceso deslizando de izquierda a derecha en la pantalla del móvil. En la Figura 2.38, se muestra el diseño del menú, el cual se encuentra compuesto por un encabezado informativo; en la parte inferior consta de cinco botones:

- **Principal:** dirige a la interfaz principal de la aplicación.
- **Historial:** transporta al usuario a la interfaz de historial de pedidos.
- **Ayuda:** conduce a la interfaz de ayuda de la aplicación.
- **Perfil:** lleva a la interfaz de configuración de datos del usuario.
- **Salir:** cierra la sesión del usuario actual y retrocede a la interfaz de inicio de la aplicación.

Figura 2.38 Interfaz menú aplicación

2.6.8 INTERFAZ HISTORIAL DE PEDIDOS

Es la interfaz que permite al usuario visualizar todos los pedidos en los que ha estado involucrado, desde la instalación de la aplicación hasta la fecha de consulta. En la Figura 2.39, se observa el diseño de esta interfaz, que cuenta con tarjetas que muestran la información de un pedido, las cuales al ser pulsadas redirigen a la interfaz de historial individual de pedidos. Además existe el botón *Atrás,* para regresar al menú lateral de la aplicación.

Figura 2.39 Interfaz Historial de pedidos

2.6.9 INTERFAZ HISTORIAL INDIVIDUAL

Es la interfaz que presenta toda la información referente a un pedido en el que haya estado involucrado un usuario. Como se observa en la Figura 2.40, la interfaz cuenta con un mapa donde se muestra la información de la ruta con la que se llevó a cabo el pedido. Se tienen además varias casillas de texto en las que se entrega información correspondiente al pedido, tal como: fecha, distancia, tiempo, precio del pedido y número de cilindros de gas. Finalmente, se cuenta con casillas de texto y una casilla de imagen para mostrar la información del usuario del otro extremo que estuvo involucrado en el pedido.

Figura 2.40 Interfaz historial individual

2.6.10 INTERFAZ DE AYUDA

Es la interfaz que muestra respuestas a las dudas que el usuario pueda tener acerca del uso de la aplicación. En la Figura 2.41, se observa la interfaz, la cual cuenta con una sección de “Preguntas frecuentes”, la que consta de casillas de texto, que al ser presionadas proveen respuestas a preguntas predeterminadas. Además se dispone de la sección “Contáctenos”, la cual despliega la información de los administradores de la aplicación. Finalmente existe el botón *Atrás,* para regresar al menú lateral de la aplicación.

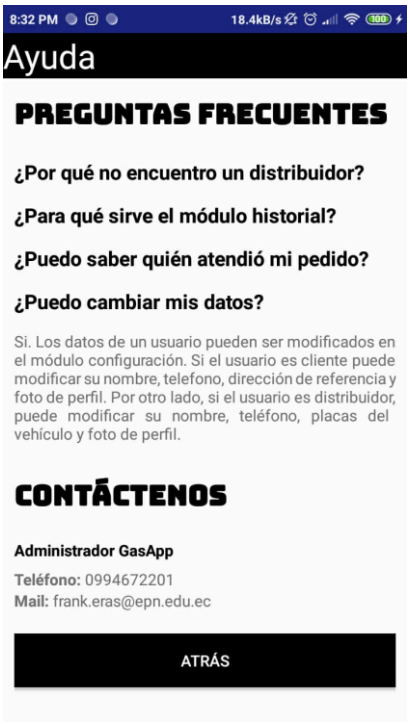

Figura 2.41 Interfaz Ayuda

2.6.11 INTERFAZ CONFIGURACIÓN DE DATOS

Es la interfaz para realizar el cambio de información del usuario que se encuentre dentro de la aplicación. Consta de tres casillas de texto, dos botones normales y un botón tipo imagen, el cual muestra la foto de perfil del usuario, misma que al presionar ofrece la opción de cambiarla. El botón *Guardar,* tiene la función de guardar los datos modificados en la interfaz y el botón *Atrás,* sirve para volver al menú lateral de la aplicación sin guardar los cambios efectuados.

En la Figura 2.42 se observa la interfaz para el cliente, en la que se puede editar: nombre y apellido, teléfono, y agregar una dirección de referencia; mientras que la Figura 2.43 muestra la interfaz para el distribuidor, en la cual se puede editar: nombre y apellido, teléfono y placas del vehículo.

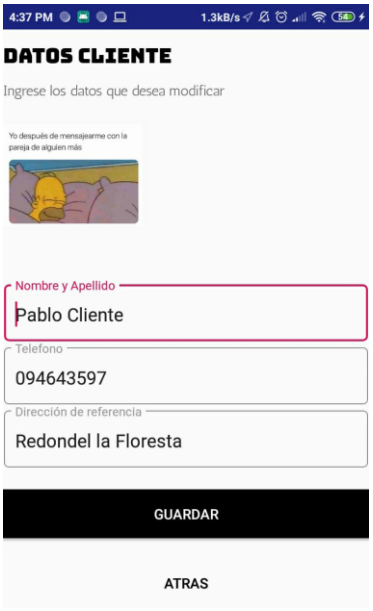

Figura 2.42 Interfaz configuración cliente

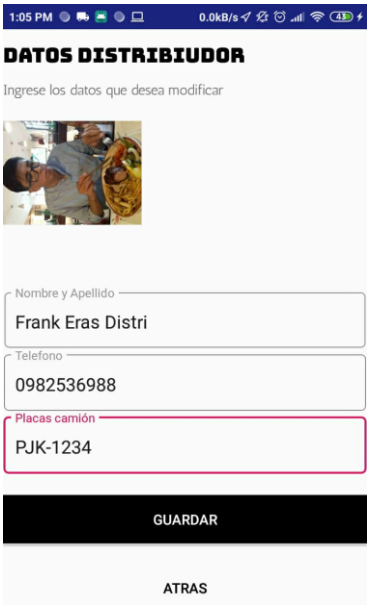

Figura 2.43 Interfaz configuración distribuidor

2.7 DESARROLLO DE LA APLICACIÓN

La aplicación es desarrollada utilizando el lenguaje de programación Java en Android Studio. En la Figura 2.44 se presenta la estructura del proyecto, en la cual los archivos dentro de la carpeta Java son las actividades que permitirán el correcto funcionamiento de la aplicación móvil y se describe cada actividad para entender la función que desempeña en la aplicación.

2.7.1 PANTALLA_ INICIO.JAVA

En esta actividad se programa la pantalla que se presenta al usuario al ingresar a la aplicación, la cual básicamente muestra un *splash screen* con duración de 4 segundos. Este archivo contiene código que permite cargar los elementos, animaciones y transiciones en la pantalla de inicio; además realiza el proceso de autenticación de usuario, es decir, si el usuario ya ha ingresado con anterioridad a la aplicación lo envía directamente a la pantalla principal de la aplicación, evitando que ingrese las credenciales nuevamente.

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

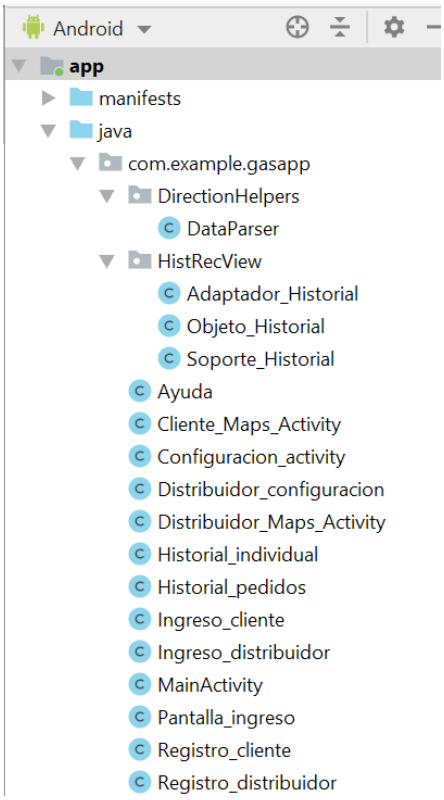

Figura 2.44 Estructura del proyecto en Android Studio

- **Código para inicializar los elementos y animaciones**

Se inicializan las variables necesarias para las animaciones, además de los elementos como el título, subtítulo e imagen presentes en la interfaz de inicio. Adicional se llama a la función geolocalizar para obtener la ubicación del usuario y finalmente se obtiene la instancia de Firebase para el proceso de autenticación, obteniendo el ID asignado al usuario para comparar con la base de datos de Firebase. Lo descrito se muestra en el Código 2.8.

```
48          //Inicializar las variables para las animaciones
49          Animation topAnima = AnimationUtils.loadAnimation( context: this, R.anim.top_animation);
50          Animation bottomAnima = AnimationUtils.loadAnimation( context: this, R.anim.bottom_animation);
51          //Inicializar las variables para los elementos
52          imagen= findViewById(R.id.imagenly);
53          titulo= (TextView) findViewById(R.id.tituloly);
54          TextView subtitulo = findViewById(R.id.subtituloly);
55          //Ubicar las animaciones
56          imagen.setAnimation(topAnima);
57          titulo.setAnimation(bottomAnima);
58          subtitulo.setAnimation(bottomAnima);
59          autenticar= FirebaseAuth.getInstance();
60          geolocalizar();
61          firebaseAuthListener =new FirebaseAuth.AuthStateListener() {
62              @Override
63              public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
64                  FirebaseUser user= FirebaseAuth.getInstance().getCurrentUser();
```

Código 2.8 Pantalla_inicio-Inicialización de variables

- **Código para autenticar si el usuario es cliente**

Antes de cargar la interfaz para el ingreso de credenciales, se autentica al usuario (Código 2.9); si el usuario ya ha ingresado con anterioridad a la aplicación, se valida si es cliente, comparando con la base de datos el ID del usuario actual (línea 76). Si existe coincidencia, avanza directamente a la interfaz principal de la aplicación para realizar pedidos, esto para evitar que se ingresen credenciales cada vez que se abre la aplicación.

```
68  if(user != null){
69      ID_usuario=FirebaseAuth.getInstance().getCurrentUser().getUid();
70      mclientedatabase= FirebaseDatabase.getInstance().getReference().child("Usuarios").child("Clientes");
71      mclientedatabase.addListenerForSingleValueEvent(new ValueEventListener() {
72          @Override
73          public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
74              if(dataSnapshot.exists()){
75                  for(DataSnapshot child:dataSnapshot.getChildren()){
76                      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
77                          if(Objects.equals(child.getKey(), ID_usuario)){
78                              new Handler().postDelayed(() -> {
81                                      Intent intent=new Intent( packageContext: Pantalla_principal.this,
82                                              Cliente_Maps_Activity.class);
83                                      startActivity(intent);
84                                      finish();
85                              },splash_scree);
```

Código 2.9 Pantalla_inicio-Autenticar si el usuario es cliente

- **Código para autenticar si el usuario es distribuidor**

Mediante el Código 2.10 se valida si el usuario actual es distribuidor, comparando con la base de datos el ID del usuario actual; si existe coincidencia, avanza directamente a la interfaz principal de la aplicación para recibir pedidos (línea 105), evitando el ingreso de credenciales.

```
95      mclientedatabase= FirebaseDatabase.getInstance().getReference().child("Usuarios").child("Distribuidores");
96      mclientedatabase.addListenerForSingleValueEvent(new ValueEventListener() {
97          @Override
98          public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
99              if(dataSnapshot.exists()){
100                 for(DataSnapshot child:dataSnapshot.getChildren()){
101                     if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
102                         if(Objects.equals(child.getKey(), ID_usuario)){
103                             new Handler().postDelayed(() -> {
106                                     Intent intent=new Intent( packageContext: Pantalla_principal.this,
107                                             Distribuidor_Maps_Activity.class);
108                                     startActivity(intent);
109                                     finish();
110                             },splash_scree);
```

Código 2.10 Pantalla_inicio-Autenticar si el usuario es distribuidor

- **Código para transiciones de los objetos**

El Código 2.11 muestra cómo se activan las transiciones del logo y título para pasar a la siguiente actividad luego de determinado tiempo. Estas transiciones animadas solo funcionan en versiones de sistema operativo mayor a Android Lollipop, por lo que si el móvil

tiene una versión anterior las transiciones se realizaran sin animación, lo cual no afecta el funcionamiento de la aplicación, esto se puede apreciar en las líneas de código 129 a 132.

```
121     // Paso de la pantalla de presentación a la de Ingreso
122     new Handler().postDelayed(() → {
125             Intent intent=new Intent( packageContext: Pantalla_principal.this, Pantalla_ingreso.class);
126             //Activo las transiciones
127             Pair[] pairs= new Pair[2];
128             pairs[0]=new Pair<View, String>(imagen, "logo-image");
129             pairs[1]=new Pair<View, String>(titulo, "logo-text");
130
131             if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.LOLLIPOP) {
132                 ActivityOptions options=ActivityOptions.makeSceneTransitionAnimation(
133                         activity: Pantalla_principal.this,pairs);
134                 startActivity(intent,options.toBundle());
135             }
136     },splash_scree);
```

Código 2.11 Pantalla_inicio-Transición mediante animaciones

- **Código para inicializar el servicio de autenticación de Firebase**

Al iniciar esta actividad, se activa el proceso de escucha entre el móvil y el servidor de autenticación de Firebase para poder verificar al usuario que ingresa a la aplicación en el móvil, esto mediante el Código 2.12.

```
//Cuando comienza la actividad el servidor FIREBASE comienza a escuchar
    @Override
    protected  void onStart(){
        super.onStart();
        autenticar.addAuthStateListener(firebaseAuthListener);
    }
```

Código 2.12 Pantalla_inicio-Comenzar escucha del servidor Firebase

- **Código para detener el servicio de autenticación de Firebase**

Cuando se ha autenticado al usuario, no es necesario que se siga usando el servicio de autenticación de Firebase, por lo que se detiene el proceso de escucha para economizar la batería del móvil, como se observa en las líneas del Código 2.13.

```
// Cuando se abandona la actividad el servidor FIREBASE deja de escuchar
    @Override
    protected void onStop() {
        super.onStop();
        autenticar.removeAuthStateListener(firebaseAuthListener);
    }
```

Código 2.13 Pantalla_inicio-Parar escucha del servidor Firebase

- **Código para obtener la posición actual del usuario**

Esta función (Código 2.14) verifica si la aplicación tiene permiso para acceder a la ubicación del móvil del usuario mediante el GPS; si el resultado es negativo se llama a otra función

para solicitar los permisos necesarios, como se observa en la línea 149. Por otro lado, si se cuenta con los permisos, se actualiza la posición del usuario, en intervalos de 1 minuto o cada 10 metros desplazados desde la última posición conocida.

```
140     // para verificar los permisos de GPS y obtener la posición actual en base a GPS
141     public void geolocalizar (){
142         //Verifica si se tiene permisos para el GPS
143         if(ActivityCompat.checkSelfPermission( context: this,
144                 Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED &&
145                 ActivityCompat.checkSelfPermission( context: this,Manifest.permission.
146                         ACCESS_COARSE_LOCATION) !=PackageManager.PERMISSION_GRANTED){
147             //si no tiene permisos solicito
148             if(Build.VERSION.SDK_INT>=Build.VERSION_CODES.M){
149                 ActivityCompat.requestPermissions( activity: this, new String[]{Manifest.permission.ACCESS_COARSE_LOCATION,Manifest.
150                         permission.ACCESS_FINE_LOCATION,Manifest.permission.INTERNET}, requestCode: 10); }
```

Código 2.14 Pantalla_inicio-Obtener posición del usuario mediante GPS

- **Código para obtener permisos para el uso del GPS del usuario**

Si la aplicación no tiene permisos para el uso del GPS, se realiza la solicitud de permisos al usuario; si el permiso es concedido se llama nuevamente a la función *geolocalizar(),* como se observa en la línea 160 del Código 2.15.

```
154     //permisos para acceder al GPS
155     @Override
156     public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResult){
157         if (requestCode == 10) {
158             if (grantResult.length > 0 && grantResult[0] == PackageManager.PERMISSION_GRANTED) {
159                 Toast.makeText( context: this,  text: "Permiso Concedido", Toast.LENGTH_SHORT).show();
160                 geolocalizar();
161             } else {
162                 Toast.makeText( context: this,  text: "Permiso Denegado", Toast.LENGTH_SHORT).show();
```

Código 2.15 Pantalla_inicio-Permiso para uso del GPS del usuario

2.7.2 PANTALLA_INGRESO.JAVA

En esta actividad el usuario de la aplicación decide si es distribuidor o cliente; se tienen dos botones para dicha selección, luego de la cual se pasará a diferentes actividades dependiendo de la decisión del usuario.

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para inicialización de los elementos a usarse**

Se inicializan las variables necesarias para el funcionamiento de esta actividad, incluyendo dos botones: cliente y distribuidor. Posteriormente se inicializan los elementos para las transiciones desde la actividad anterior hacia ésta y para las transiciones de ésta a la posterior actividad, como se observa en el Código 2.16.

```
// Inicialización de los botones
distribuidor= (Button) findViewById(R.id.distribuidor);
cliente= (Button) findViewById(R.id.cliente);
//Inicialización de los elementos
imagen= (ImageView) findViewById(R.id.imagenly);
titulo= (TextView) findViewById(R.id.tituloly);
subtitulo=(TextView) findViewById(R.id.titulo2ly);
//Inicialización de las transiciones
pairs= new Pair[5];
pairs[0]=new Pair<View, String>(imagen, "logo-image");
pairs[1]=new Pair<View, String>(titulo, "logo-text");
pairs[2]=new Pair<View, String>(subtitulo, "logo-subtext");
pairs[3]=new Pair<View, String>(distribuidor, "boton-1");
pairs[4]=new Pair<View, String>(cliente, "boton-2");
```

Código 2.16 Pantalla_ingreso-Inicialización elementos en Pantalla_ingreso

- **Código para selección del rol Distribuidor**

En el Código 2.17 se observa cómo el usuario al presionar el botón *Distribuidor* pasará de la actividad actual a la actividad *Ingreso_distribuidor*, en donde se hará el proceso de autenticación de credenciales.

```
distribuidor.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent=new Intent( packageContext: Pantalla_ingreso.this, Ingreso_distribuidor.class);
        startActivity(intent);
    }
});
```

Código 2.17 Pantalla_ingreso-Selección de rol de distribuidor

- **Código para selección del rol Cliente**

Como se muestra en el Código 2.18, el usuario al presionar el botón *Cliente* pasará de la actividad actual a la actividad *Ingreso_cliente*, en donde se hará el proceso de autenticación de credenciales .

```
cliente.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent2=new Intent( packageContext: Pantalla_ingreso.this, Ingreso_cliente.class);
        startActivity(intent2);
    }
});
```

Código 2.18 Pantalla_ingreso-Selección de rol de Cliente

2.7.3 INGRESO_DISTRIBUIDOR.JAVA

Si el usuario se ha identificado como *Distribuidor* accede a esta actividad, en la cual se pide el ingreso de las credenciales para avanzar a la interfaz principal y poder recibir pedidos de gas. Si el usuario se encuentra registrado en la base de datos de Firebase puede

ingresar a la aplicación, caso contrario existe un botón que permite al usuario registrarse como un nuevo *Distribuidor*. Además, si el usuario, siendo cliente accedió por equivocación a esta actividad de *Distribuidor*, se tiene un botón para movilizarse a la actividad *Ingreso_cliente*, destinada para la autenticación de clientes.

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para inicialización de elementos y transiciones**

Se inicializan los elementos necesarios para el cambio de actividades, incluyendo los botones entre los cuales se encuentran: ingresar, registrarse, cambiar a *Cliente*; además de las transiciones animadas para moverse entre actividades, como se observa en las líneas del Código 2.19.

```
95      //Inicialización de botones de ingresar y registrarse
96      ingresar1=(Button) findViewById(R.id.ingresar);
97      registrarse1=(Button) findViewById(R.id.registro);
98      cambio=(Button) findViewById(R.id.toclien);
99      contraolvi=(Button) findViewById(R.id.contradis);
100     //Inicialización de elementos para transiciones animadas
101     imagen= (ImageView) findViewById(R.id.logoly);
102     titulo=(TextView) findViewById(R.id.titulo1ly);
103     subtitulo=(TextView) findViewById(R.id.titulo2ly);
104     usuario=(TextInputLayout) findViewById(R.id.email1);
105     tcontrasena=(TextInputLayout) findViewById(R.id.contraseña1);
```

Código 2.19 Ingreso_distribuidor-Inicialización de elementos

- **Código para el botón *Soy Cliente***

El Código 2.20 se observa que al presionar el botón *Soy Cliente,* se abandona la actividad actual y se avanza a la actividad *Ingreso_cliente*, para la autenticación exclusiva de usuarios catalogados como clientes.

```
//cuando se presiona el botón de "Soy cliente"
cambio.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent1=new Intent( packageContext: Ingreso_distribuidor.this,Ingreso_cliente.class);
        startActivity(intent1);
        finish();
```

Código 2.20 Ingreso_distribuidor-Botón Soy Cliente

- **Código para el botón *Registrarse***

Al presionar el botón *Nuevo usuario? Registrarse*, se abandona la actividad y se avanza a *Registro_distribuidor* para realizar el proceso de registro en la aplicación; el paso a la

siguiente actividad incluye transiciones animadas de los títulos, imágenes y botones, como se muestra en el Código 2.21.

```
117         //Acciones a realizarse cuando se presiona el boton registrarse
118         registrarse1.setOnClickListener(new View.OnClickListener() {
119             @Override
120             public void onClick(View v) {
121                 Intent intent1=new Intent( packageContext: Ingreso_distribuidor.this,Registro_distribuidor.class);
122                 Pair[] pairs=new Pair[7];
123                 pairs[0]=new Pair<View, String>(imagen, "logo-image");
124                 pairs[1]=new Pair<View, String>(subtitulo, "logo-subtext");
125                 pairs[2]=new Pair<View, String>(usuario, "emailani");
126                 pairs[3]=new Pair<View, String>(tcontrasena, "contrasenaani");
127                 pairs[4]=new Pair<View, String>(ingresar1, "boton-1");
128                 pairs[5]=new Pair<View, String>(registrarse1, "boton-2");
129                 pairs[6]=new Pair<View, String>(titulo,"logo-text");
130
131                 if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.LOLLIPOP) {
132                     ActivityOptions options=ActivityOptions.makeSceneTransitionAnimation(
133                             activity: Ingreso_distribuidor.this,pairs);
134                     startActivity(intent1,options.toBundle());
```

Código 2.21 Ingreso_distribuidor-Botón registrarse

- **Código para el botón *Ingresar (Distribuidor)***

Mediante el Código 2.22 se autentican las credenciales ingresadas, tanto correo electrónico como contraseña al presionar el botón Ingresar; si las credenciales son correctas se avanza a la actividad principal del Distribuidor para que pueda recibir pedidos, caso contrario, si las credenciales no son las correctas se muestra un mensaje de error: “Error al ingresar. Revisar mail y contraseña” y se permanece en esta actividad, como se observa en la línea 158. Adicional, si el usuario es *Cliente* pero ingresa las credenciales en el rol de *Distribuidor* se le niega el acceso y se mantiene en la actividad esperando las credenciales correctas.

```
142         //Acciones a realizarse cuando se presiona el boton Ingresar
143         ingresar1.setOnClickListener(new View.OnClickListener() {
144             @RequiresApi(api = Build.VERSION_CODES.KITKAT)
145             @Override
146             public void onClick(View v) {
147                 final String email= Objects.requireNonNull(usuario.getEditText()).getText().toString();
148                 final String contrasena= Objects.requireNonNull(tcontrasena.getEditText()).getText().toString();
149                 if(email.isEmpty() || contrasena.isEmpty()){
150                     Toast.makeText(getBaseContext(), text: "Ingresar las credenciales", Toast.LENGTH_SHORT).show();
151                 }else {
152                     autenticar.signInWithEmailAndPassword(email, contrasena).
153                             addOnCompleteListener( activity: Ingreso_distribuidor.this, new OnCompleteListener<AuthResult>() {
154                                 @Override
155                                 public void onComplete(@NonNull Task<AuthResult> task) {
156                                     if (!task.isSuccessful()) {
157                                         Toast.makeText( context: Ingreso_distribuidor.this,
158                                                 text: "Error al ingresar. Revisar mail y contraseña", Toast.LENGTH_SHORT).show();
```

Código 2.22 Ingreso_distribuidor-Botón Ingresar

2.7.4 INGRESO_CLIENTE.JAVA

Si el usuario elige el rol de *cliente* accede a esta capa, en la cual se solicita el ingreso de las credenciales para autenticar al usuario. Si el *cliente* se encuentra registrado puede

avanzar a la actividad principal de la aplicación, caso contrario existe un botón que permite que el usuario se registre como un nuevo *cliente*. Adicionalmente, si el usuario, siendo *Distribuidor*, accedió por equivocación a esta actividad, existe un botón para movilizarse a la actividad *Ingreso_distribuidor*, destinada para la autenticación de distribuidores de gas.

Para esta actividad es posible reutilizar el código descrito en Ingreso_distribuidor.java, puesto que se emplean los mismos elementos y se sigue la misma lógica de funcionamiento; sin embargo se aplican pequeños cambios para adaptarse al rol del cliente. A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para el botón *Soy Distribuidor***

Como se muestra en el Código 2.23, al presionar el botón *Soy Distribuidor,* se abandona esta actividad y se avanza a la actividad *Ingreso_distribuidor*, para la autenticación exclusiva del distribuidor de gas.

```
//cuando se presiona el botón de "Soy distribuidor"
cambio.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent1=new Intent( packageContext: Ingreso_cliente.this,Ingreso_distribuidor.class);
        startActivity(intent1);
        finish();
    }
});
```

Código 2.23 Ingreso_cliente-Botón Soy Distribuidor

- **Código para el botón Ingresar (cliente)**

En el Código 2.24, se muestra que al presionar el botón *Ingresar,* se autentican las credenciales ingresadas, correo y contraseña (líneas 68-75); si las credenciales son correctas se avanza a la actividad principal del *cliente*, caso contrario, se muestra un mensaje de error: “Error al ingresar” y se permanece en esta actividad. Adicionalmente, si el usuario es *Distribuidor* pero ingresa las credenciales en el rol de *cliente* se le niega el acceso y se mantiene en la actividad esperando las credenciales correctas.

```
//Autenticación al ingresar
autenticar= FirebaseAuth.getInstance();
firebaseAuthListener = new FirebaseAuth.AuthStateListener() {
    @Override
    public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
        FirebaseUser user= FirebaseAuth.getInstance().getCurrentUser();
        if(user != null){
            ID_usuario=FirebaseAuth.getInstance().getCurrentUser().getUid();
            mclientedatabase= FirebaseDatabase.getInstance().getReference().child("Usuarios").child("Clientes");
            mclientedatabase.addListenerForSingleValueEvent(new ValueEventListener() {
                @RequiresApi(api = Build.VERSION_CODES.KITKAT)
                @Override
                public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
                    if(dataSnapshot.exists()){
                        for(DataSnapshot child:dataSnapshot.getChildren()){
                            if(Objects.equals(child.getKey(), ID_usuario)){
                                Toast.makeText( context: Ingreso_cliente.this,
                                        text: "Ingresando", Toast.LENGTH_SHORT).show();
                                Intent intent=new Intent( packageContext: Ingreso_cliente.this, Cliente_Maps_Activity.class);
                                startActivity(intent);
                                finish();
                                break;
                            }else{
                                if (aux==1) {
                                    aux=2;
                                }else{
                                    aux=1;
                                    Toast.makeText(getBaseContext(),
                                            text: "Acceso Denegado: Usuario es Distribuidor", Toast.LENGTH_SHORT).show();
```

Código 2.24 Ingreso_cliente-Botón ingresar

2.7.5 REGISTRO_DISTRIBUIDOR.JAVA

Esta actividad es la encargada de registrar nuevos distribuidores en la aplicación; se pide la información básica como: nombres, teléfono, placas del vehículo, correo electrónico y contraseña; adicional se debe ingresar una foto de perfil.

A continuación, se describen fragmentos de código para el funcionamiento de esta actividad.

- **Código para inicialización de elementos**

Se inicializan dos botones en la aplicación, uno para registrarse y otro para volver a la actividad anterior *Ingreso_distribuidor* en caso de haber ingresado erróneamente a esta actividad. Adicionalmente, se inicializan las casillas de texto donde se ingresarán los datos para el proceso de registro, como se observa en el Código 2.25.

```
//Inicialización de botones
bregistrarse=(Button) findViewById(R.id.registroly);
batras=(Button) findViewById(R.id.atrasly);
//Inicialización para las entradas de texto
nombre=(TextInputLayout) findViewById(R.id.nombrely);
telefono=(TextInputLayout) findViewById(R.id.telefonoly);
placas =(TextInputLayout) findViewById(R.id.placasly);
email=(TextInputLayout) findViewById(R.id.emailly);
contrasena=(TextInputLayout) findViewById(R.id.contraseñaly);
```

Código 2.25 Registro_distribuidor-Inicialización de elementos

- **Código para obtener la información ingresada**

En el Código 2.26 se muestra el proceso mediante el cual se recupera la información ingresada en las casillas de texto, y su almacenamiento en variables para su posterior uso.

```
//obtiene la información ingresada en las casillas de texto
private void obtener_info(){
    nombre_aux=nombre.getEditText().getText().toString();
    telefono_aux=telefono.getEditText().getText().toString();
    placas_aux=placas.getEditText().getText().toString();
    email_aux=email.getEditText().getText().toString();
    contrasena_aux=contrasena.getEditText().getText().toString();
```

Código 2.26 Registro_distribuidor-Obtener información del distribuidor

- **Código para la foto de perfil**

Al presionar la imagen, comienza un nuevo evento de selección de una imagen en el móvil, con el cual al seleccionar una imagen se almacena en la aplicación para posteriormente ser subida al servidor Firebase. Esto es posible visualizar en el Código 2.27.

```
//al presionar la foto de perfil
fotodeperfil.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent=new Intent(Intent.ACTION_PICK);
        intent.setType("image/*");
        startActivityForResult(intent, requestCode: 1);
```

Código 2.27 Registro_distribuidor-Botón de Foto de Perfil

Para guardar la foto de perfil se crea un mapa de bits y se almacena la imagen en éste, luego es comprimida en formato JPEG y almacenada en un *array* de bytes como se observa en las líneas 133 a 135 del Código 2.28. Posteriormente se sube esta fotografía al servicio de *Cloud Storage* que ofrece Firebase; la fotografía se sube con el ID del cliente a la carpeta Foto_Perfil.

```
133     ByteArrayOutputStream com = new ByteArrayOutputStream();
134     bitmap.compress(Bitmap.CompressFormat.JPEG, quality: 20, com);
135     byte[] data = com.toByteArray();
136     //guardar la foto
137     final UploadTask subir = fotopath.putBytes(data);
138     subir.addOnSuccessListener(new OnSuccessListener<UploadTask.TaskSnapshot>() {
139         @Override
140         public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
141             fotopath.getDownloadUrl().addOnSuccessListener(new OnSuccessListener<Uri>() {
142                 @Override
143                 public void onSuccess(Uri uri) {
144                     Map nueva_img = new HashMap();
145                     nueva_img.put( k: "fotoUrl", uri.toString());
146                     mclientedatabase.updateChildren(nueva_img);
147                     finish();
```

Código 2.28 Registro_distribuidor-Guardar foto de perfil del cliente

- **Código para el botón *Registrarse***

En el Código 2.29 se observa que al presionar el botón *Registrarse,* se crea un nuevo usuario en la plataforma Firebase (línea 86), al cual se le asigna un ID único dentro de la aplicación, el *email* y contraseña se asocian a dicho ID y se registran en los servicios de autenticación de Firebase para posteriores ingresos a la aplicación, esto validando que se haya ingresado toda la información correspondiente como se observa en la línea 80. Dentro del árbol de nodos de la base de datos de Firebase se crean las categorías *email*, nombre, teléfono, placas y se guarda la información registrada bajo el ID asignado anteriormente. En la Figura 2.45 se muestra cómo queda el árbol de nodos al registrarse un nuevo *Distribuidor*.

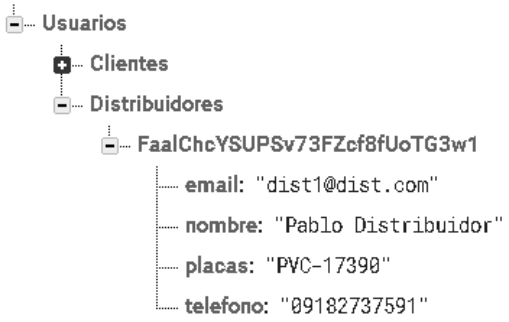

Figura 2.45 Árbol de nodos en Firebase con nuevo distribuidor

Si el proceso es exitoso se avanza a la actividad principal del *Distribuidor*, en la cual puede recibir pedidos de gas; si por el contrario, el proceso de registro no se lleva a cabo, aparece un mensaje de error: “Error en el registro” y el usuario debe intentarlo nuevamente.

```
76  bregistrarse.setOnClickListener(new View.OnClickListener() {
77      @Override
78      public void onClick(View view) {
79          obtener_info();
80          if(nombre_aux.isEmpty() || placas_aux.isEmpty() || telefono_aux.isEmpty() ||
81                  email_aux.isEmpty() || contrasena_aux.isEmpty() ){
82              Toast.makeText(getBaseContext(),
83                      text: "Ingresar toda la información solicitada", Toast.LENGTH_SHORT).show();
84          }else {
85              //se crea un nuevo usuario en FIREBASE
86              autenticar.createUserWithEmailAndPassword(email_aux, contrasena_aux).
87                      addOnCompleteListener( activity: Registro_distribuidor.this, new OnCompleteListener<AuthResult>() {
88                          @Override
89                          public void onComplete(@NonNull Task<AuthResult> task) {
90                              if (!task.isSuccessful()) {
91                                  autenticar.fetchSignInMethodsForEmail(email_aux)
92                                          .addOnCompleteListener(new OnCompleteListener<SignInMethodQueryResult>() {
93                                      @Override
94                                      public void onComplete(@NonNull Task<SignInMethodQueryResult> task) {
95                                          boolean check = !task.getResult().getSignInMethods().isEmpty();
96                                          if (!check) {
97                                              Toast.makeText( context: Registro_distribuidor.this,
98                                                      text: "Error en el registro", Toast.LENGTH_SHORT).show();
99                                          } else {
100                                             Toast.makeText( context: Registro_distribuidor.this,
101                                                     text: "El mail ingresado ya se encuentra registrado", Toast.LENGTH_SHORT).show();
```

Código 2.29 Registro_distribuidor-Botón registrarse

- **Código para el botón *Atrás***

Al presionar el botón *Atrás,* se finaliza esta actividad y se regresa a la actividad anterior (*Ingreso_distribuidor*) como se muestra en el Código 2.30.

```
//boton atras
batras.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent=new Intent( packageContext: Registro_distribuidor.this, Ingreso_distribuidor.class);
        startActivity(intent);
        finish();
    }
});
```

Código 2.30 Registro_distribuidor-Botón Atrás

2.7.6 REGISTRO_CLIENTE.JAVA

Esta actividad es la encargada de registrar nuevos clientes en la aplicación, solicitando la información de: nombres, teléfono, correo electrónico y contraseña; adicionalmente se debe cargar una imagen de perfil.

Para esta actividad es posible reutilizar el código descrito en Registro_distribuidor.java, puesto que se emplean los mismos elementos y se usa una lógica de funcionamiento similar. Sin embargo, se aplican pequeños cambios para adaptarse al rol del *cliente*. A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para el botón *Registrarse***

Al presionar el botón *Registrarse,* se realiza el proceso explicado anteriormente en el Código 2.29, con la diferencia que al momento de crear el nuevo usuario en la plataforma Firebase, el cliente es ubicado dentro del árbol de nodos de la base de datos de Firebase en la sección *Clientes,* y se crean las categorías nombre, teléfono, *email* y URL de la foto de perfil, en las cuales se guarda la información ingresada en la aplicación. En la Figura 2.46, se muestra cómo queda el árbol de nodos al registrarse un nuevo *cliente*.

Figura 2.46 Árbol de nodos en Firebase con nuevo cliente

- **Código para el botón *Atrás***

Al presionar el botón *Atrás,* se finaliza esta actividad y se regresa a la actividad anterior (*Ingreso_cliente*) como se muestra en el Código 2.31.

```
//boton atrás
batras.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent=new Intent( packageContext: Registro_cliente.this, Ingreso_cliente.class);
        startActivity(intent);
        finish();
    }
});
```

Código 2.31 Registro_cliente-Botón Atrás

2.7.7 CLIENTE_MAPS_ACTIVITY.JAVA

Es la actividad principal del *cliente*, en la cual es posible realizar el pedido de cilindros de gas al *Distribuidor* y recibir la retroalimentación del *Distribuidor* que va a atender el pedido.

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para inicializar las variables**

En el Código 2.32 se inicializan las variables que permiten el funcionamiento de la interfaz principal de la aplicación del *cliente*, entre ellas se tiene casillas de texto, una casilla de imagen, botones y el mapa con la ubicación del *cliente*; además se crean variables para mostrar el menú lateral con las diferentes opciones de funcionalidades.

```
mapFragment.getMapAsync( onMapReadyCallback: this);
//Inicialización para trazar la ruta
polylines = new ArrayList<>();
//Inicialización de variables para mostrar la información del distribuidor
info_distribuidor= (LinearLayout) findViewById(R.id.info_distribuidorly);
info_pedido= (LinearLayout) findViewById(R.id.info_pedidoly);
foto_distribuidor= (ImageView) findViewById(R.id.foto_distribuidorly);
nombre_distribuidor= (TextView) findViewById(R.id.nombre_distribuidorly);
telefono_distribuidor= (TextView) findViewById(R.id.telefono_distribuidorly);
carro_distribuidor= (TextView) findViewById(R.id.carro_distribuidorly);
costo_pedido= (TextView) findViewById(R.id.costo_ped);
numero_tanques= (TextView) findViewById(R.id.ntanquesly);
tiempo_pedido= (TextView) findViewById(R.id.tiempo_ped);
//Inicialización de variable para el número de cilindros
numero_clindros=(EditText) findViewById(R.id.ncilindros);
//Inicializaciòn de variable para dirección
mTapTextView = (TextView) findViewById(R.id.tap_text);
//Inicialización de variables para el menú
drawerLayout=findViewById(R.id.drawer_layout);
navigationView=findViewById(R.id.navigation_view);
menuIcon=findViewById(R.id.menu_icon);
contentView=findViewById(R.id.content);
```

Código 2.32 Cliente_Maps_Activity-Inicialización de variables

- **Código botón del menú principal**

Se configura la aparición del menú lateral mediante el Código 2.33, en el cual se especifica que al presionar el ícono de menú (≡), se desplegará un menú lateral (línea 760), el mismo que también puede mostrarse al deslizarse por la pantalla de la aplicación de izquierda a derecha. Adicionalmente, en la línea 773 se llama a la función *animateNavDraw(),* la cual aporta la aparición del menú de manera animada.

```
758     // Al presionar el icono de menú para que aparezca
759     private void navigationDraw() {
760         navigationView.bringToFront();
761         navigationView.setNavigationItemSelectedListener(this);
762         navigationView.setCheckedItem(R.id.nav_home);
763         menuIcon.setOnClickListener(new View.OnClickListener() {
764             @Override
765             public void onClick(View view) {
766                 if(drawerLayout.isDrawerVisible(GravityCompat.START)){
767                     drawerLayout.closeDrawer(GravityCompat.START);
768                 }else{
769                     drawerLayout.openDrawer(GravityCompat.START);
770                 }
771             }
772         });
773         animateNavDraw();
```

Código 2.33 Cliente_Maps_Activity-Desplegar el menú lateral de la aplicación

- **Código para seleccionar un ítem del menú lateral**

El menú lateral cuenta con diferentes opciones: la opción *home* muestra la pantalla principal de la aplicación (línea 805), la opción *configuración* abandona la actividad actual y conduce al usuario a *Perfil_activity (*línea 809*)*, la opción *ayuda* abandona la actividad actual y pasa a la actividad *Ayuda* (línea 812), la opción *historial* pasa a la actividad historial de pedidos (línea 816) y finalmente la opción *salir,* cierra la sesión del usuario actual y regresa a la pantalla inicial de la aplicación, lo cual se puede apreciar en el Código 2.34.

```
801     //Accion a realizarse al seleccionar un item del menu
802     @Override
803     public boolean onNavigationItemSelected(@NonNull MenuItem menuItem) {
804         switch (menuItem.getItemId()){
805             case R.id.nav_home:
806                 drawerLayout.closeDrawer(GravityCompat.START);
807                 break;
808             case R.id.nav_configuracion:
809                 Intent intent=new Intent( packageContext: Cliente_Maps_Activity.this, Perfil_activity.class);
810                 startActivity(intent);
811                 break;
812             case R.id.nav_ayuda:
813                 Intent intenta=new Intent( packageContext: Cliente_Maps_Activity.this, Ayuda.class);
814                 startActivity(intenta);
815                 break;
816             case R.id.nav_histor:
817                 Intent intenth=new Intent( packageContext: Cliente_Maps_Activity.this, Historial_pedidos.class);
818                 intenth.putExtra( name: "DistriOcliente", value: "Clientes");
819                 startActivity(intenth);
820                 break;
821             case R.id.nav_salir:
822                 FirebaseAuth.getInstance().signOut();
823                 Intent intents=new Intent( packageContext: Cliente_Maps_Activity.this,Pantalla_ingreso.class);
824                 startActivity(intents);
825                 finish();
826                 break;
```

Código 2.34 Cliente_Maps_Activity-Opciones del menú lateral del cliente

- **Código para mostrar el mapa en la pantalla del cliente**

En el Código 2.35 desarrolla la pantalla principal de la aplicación, se configura el mapa habilitando los botones de *zoom* y localización (líneas 604-607), y se llama a la función *geolocalizar()* en la línea 631, para encontrar la posición precisa del usuario en base al GPS del móvil.

```
602     @Override
603     public void onMapReady(GoogleMap googleMap) {
604         mMap = googleMap;
605         mMap.setMyLocationEnabled(true);
606         mMap.getUiSettings().setZoomControlsEnabled(true);
607         mMap.getUiSettings().setMyLocationButtonEnabled(true);
608         locationManager=((LocationManager) this.getSystemService(Context.LOCATION_SERVICE));
609         locationListener=new LocationListener() {
610             @Override
611             public void onLocationChanged(Location location) {
612                 //obtener coordenas de latitud y longitud de posición actual, y marcador morado.
613                 lat=location.getLatitude();
614                 lon=location.getLongitude();
615                 ubicacion=new LatLng(lat,lon);
616                 if (null !=marc_actual){
617                     marc_actual.remove(); }
618                 marc_actual=mMap.addMarker(new MarkerOptions().position(ubicacion).title("Ubicación actual").
619                         icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_BLUE)).alpha(0.3f));
620                 mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(ubicacion, v: 17));
621             }
622             @Override
623             public void onStatusChanged(String provider, int status, Bundle extras) { }
624             @Override
625             public void onProviderEnabled(String provider) { }
626             @Override
627             public void onProviderDisabled(String provider) {
628                 Intent i=new Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS);
629                 startActivity(i);
630             }
631         };geolocalizar();
```

Código 2.35 Cliente_Maps_Activity-Mostrar el mapa en la pantalla

- **Código para colocar la posición del cliente en el mapa del cliente**

En el Código 2.36 se aprecia que el usuario para realizar un pedido debe presionar un punto en el mapa para que el distribuidor conozca el lugar donde debe despachar el cilindro de gas. La función configurarMapa() en la línea 576, inicia un evento de escucha que se mantiene hasta que el usuario presiona un lugar en el mapa, luego de lo cual se activa la función onMapLongClick(), la cual agrega un marcador en el mapa en el lugar señalado y almacena las coordenadas en una variable (línea 587) para su posterior uso.

```
575     //Se mantiene a la escucha de presionar un punto en el mapa
576     private void configurarMapa() {
577         mMap.setOnMapLongClickListener((GoogleMap.OnMapLongClickListener) this);
578     }
579     // Cuando se presiona un lugar en el mapa obtiene coordenadas y pone marcador en el mapa
580     public void onMapLongClick(LatLng punto) {
581         if (null !=marc_fin){
582             marc_fin.remove(); }
583         marc_fin= mMap.addMarker(new MarkerOptions().position(punto));
584         this.setLocation(punto);
585         lat1 = punto.latitude;
586         lon1 = punto.longitude;
587         ubicacion_cl=new LatLng(lat1,lon1);
```

Código 2.36 Cliente_Maps_Activity-Presionar un punto en el mapa

- **Código para obtener la dirección del cliente**

La función que se observa en el Código 2.37, permite que el *cliente* al presionar un punto en el mapa active *setLocation(),* función que transforma las coordenadas de latitud y longitud en direcciones como se muestra en la línea 593, y ubica dicha dirección en una casilla de texto para visualización del *cliente* (línea 597).

```
589     // Para obtener la direccion del punto señalado
590 @   private void setLocation(LatLng punto) {
591         try {
592             Geocoder geocoder = new Geocoder( context: this, Locale.getDefault());
593             List<Address> list = geocoder.getFromLocation(
594                     punto.latitude, punto.longitude,  maxResults: 1);
595             if (!list.isEmpty()) {
596                 Address DirCalle = list.get(0);
597                 mTapTextView.setText(Html.fromHtml( source: "<b>Dirección: </b>"
598                         +DirCalle.getAddressLine( index: 0)));
```

Código 2.37 Cliente_Maps_Activity-Obtener dirección del punto presionado

- **Código para el botón *Pedir Gas***

Al presionar el botón *Pedir Gas,* primeramente se valida que los campos dirección y número de cilindros estén llenos, de no estarlos se envía un mensaje solicitando que la información esté completa. Si los campos están llenos, se obtiene la ubicación del *cliente* y se crea en la base de datos de Firebase, en el nodo *Pedidos,* un nuevo pedido con la latitud y longitud del usuario; luego se llama a la función *obtener_distribuidor,* para conseguir un *Distribuidor* que atienda el pedido. Si el pedido se encuentra en curso el botón *Pedir Gas,* cambia a *Cancelar Pedido,* en cuyo caso al presionar se llama a la función *fin_pedido(),* para terminar el requisito de pedido de gas. El proceso descrito anteriormente se observa en el Código 2.38.

```
//Botón pedir gas
pedir_gas =(Button)findViewById(R.id.pedir);
pedir_gas.setOnClickListener((v) → {
        String nc=numero_clindros.getText().toString();
        if (ubicacion_cl==null || nc.isEmpty()){
            Toast.makeText(getBaseContext(),  text: "Llenar la información solicitada", Toast.LENGTH_SHORT).show();
        }else{
            pedir_gas.setVisibility(View.GONE);
            cancelar_pedido.setVisibility(View.VISIBLE);
            cancelar = true;
            String ID_usuario = FirebaseAuth.getInstance().getCurrentUser().getUid();
            DatabaseReference ref = FirebaseDatabase.getInstance().getReference( path: "Pedidos");
            GeoFire geoFire = new GeoFire(ref);
            geoFire.setLocation(ID_usuario, new GeoLocation(lat1, lon1));//punto seleccionado en el mapa
            localizacion_pedido = new LatLng(lat1, lon1);
            if (marc_fin != null) {
                marc_fin.remove();
            }
            marc_fin = mMap.addMarker(new MarkerOptions().position(localizacion_pedido).title("Entregar aqui"));
            obtener_distribuidor();
        }
});
```

Código 2.38 Cliente_Maps_Activity-Botón Pedir Gas

- **Código para encontrar un distribuidor de gas**

Una vez realizado el pedido de gas, la función *obtener_distribuidor(),* permite encontrar el *Distribuidor* más cercano al cliente en base a las coordenadas del usuario, mediante una consulta realizada con GeoFire, empleando el Código 2.39. La búsqueda se realiza en base a los *Distribuidores* presentes en el nodo *DistribuidorDisponible,* en la base de datos de Firebase.

```
private void obtener_distribuidor(){
    //Obtener Distribuidores disponibles
    DatabaseReference ubicacion_distribuidor = FirebaseDatabase.getInstance().
            getReference().child("DistribuidorDisponible");
    GeoFire geoFire=new GeoFire(ubicacion_distribuidor);
    //realiza una busqueda entre la ubicación del pedido y la ubicación del distribuidor
    geoQuery= geoFire.queryAtLocation(new
            GeoLocation(localizacion_pedido.latitude,localizacion_pedido.longitude),radio);
    geoQuery.removeAllListeners();
    geoQuery.addGeoQueryEventListener(new GeoQueryEventListener() {
```

Código 2.39 Cliente_Maps_Activity-Obtener un distribuidor para el pedido (1)

Al encontrar un *Distribuidor* para el pedido, se obtiene el ID del *Distribuidor* y se accede a la base de datos de Firebase del *Distribuidor* para crear un nuevo nodo con el nombre *Pedido_cliente,* el cual contendrá el ID del cliente del cual viene el pedido y el número de cilindros de gas que se pedirá. Luego se llama a las demás funciones necesarias para que el proceso de pedido de gas se complete, como se muestra en el Código 2.40.

```
245  public void onKeyEntered(String key, GeoLocation location) {
246      if (!distribuidor_encontrado&&cancelar) { //Entra solo una vez, al encontrar a un distribuidor
247          //se obtiene la información del distribuidor encontrado
248          DatabaseReference clientedatabase=FirebaseDatabase.getInstance().getReference().child("Usuarios")
249                  .child("Distribuidores").child(key);
250          clientedatabase.addListenerForSingleValueEvent(new ValueEventListener() {
251              @Override
252              public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
253                  if(dataSnapshot.exists() && dataSnapshot.getChildrenCount()>0){
254                      Map<String, Object> Distr_map=(Map<String, Object>) dataSnapshot.getValue();
255                      if(distribuidor_encontrado){
256                          return;}
257                      distribuidor_encontrado = true;
258                      ID_distribuidor_encontrado=dataSnapshot.getKey();
259                      //Digo al distribuidor cual cliente va a atender
260                      DatabaseReference distr_ref=FirebaseDatabase.getInstance().getReference().child("Usuarios")
261                              .child("Distribuidores").child(ID_distribuidor_encontrado).child("Pedido_cliente");
262                      String ID_cliente=FirebaseAuth.getInstance().getCurrentUser().getUid();
263                      HashMap  map=new HashMap();
264                      map.put("ID_cliente",ID_cliente); //obtengo ID cliente
265                      map.put("Numero_cilindros",numero_clindros.getText().toString());//obtengo # de cilindros
266                      map.put("Estado",estado_ped);//estado del pedido
267                      distr_ref.updateChildren(map);//actualizo información bajo el Distribuidor
268                      obtener_info_distribuidor();
269                      obtener_direccion_distribuidor();
270                      fin_pedido_aux();
```

Código 2.40 Cliente_Maps_Activity-Obtener un distribuidor para el pedido (2)

- **Código para obtener la información del distribuidor asignado al pedido**

El Código 2.41 explica la función *obtener_info_distribuidor(),* la cual accede a la base de datos de Firebase y extrae la información del *Distribuidor* que atenderá el pedido. La

información recuperada es colocada en una ventana, y se comparte con el cliente: el nombre, teléfono y placas del *Distribuidor*.

```
294     //Obtener la informacion del distribuidor
295     private void obtener_info_distribuidor(){
296         //Se obtiene el ID del distribuidor que atenderá el pedido
297         DatabaseReference mclientedatabase=FirebaseDatabase.getInstance().
298                 getReference().child("Usuarios").child("Distribuidores")
299                 .child(ID_distribuidor_encontrado);
300         mclientedatabase.addListenerForSingleValueEvent(new ValueEventListener() {
301             @Override
302             public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
303                 if (dataSnapshot.exists()&&dataSnapshot.getChildrenCount()>0){
304                     Map<String, Object> map=(Map<String, Object>) dataSnapshot.getValue();
305 
306                     if (map.get("nombre")!=null){
307                         nombre_distribuidor.setText(Html.fromHtml( source: "<b>Distribuidor: </b>"+map.get("nombre").toString())); }
308                     if (map.get("telefono")!=null){
309                         telefono_distribuidor.setText(Html.fromHtml( source: "<b>Teléfono: </b>"+map.get("telefono").toString())); }
310                     if (map.get("placas")!=null){
311                         carro_distribuidor.setText(Html.fromHtml( source: "<b>Placas Camión: </b>"+map.get("placas").toString())); }
312                     if(map.get("fotoUrl")!=null){
313                         Glide.with(getApplication()).load(map.get("fotoUrl").toString()).into(foto_distribuidor); }
```

Código 2.41 Cliente_Maps_Activity-Obtener información del distribuidor

- **Código para obtener la dirección del distribuidor**

Se debe obtener la ubicación del *Distribuidor* que atenderá el pedido para mostrarla en el mapa al cliente y tenga una referencia de la distancia y tiempo aproximado en el que el pedido llegará, por lo cual se hace uso del Código 2.42. Se accede a la base de datos de Firebase, al nodo *DistribuidoresTrabajando* (línea 341), y con el ID del *Distribuidor* asignado se crea un nuevo evento que obtenga las coordenadas del *Distribuidor* cuando se actualicen en la base de datos (línea 348).

```
338     //Para obtener la ubicación del distribuidor
339     private void obtener_direccion_distribuidor(){
340         //se accede a la base de datos en Firebase y se escucha al distribuidor seleccionado
341         ubic_distrib_ref=FirebaseDatabase.getInstance().getReference().child("DistribuidorTrabajando")
342                 .child(ID_distribuidor_encontrado).child("l");
343         ubic_distrib_ref_listener=ubic_distrib_ref.addValueEventListener(new ValueEventListener() {
344             //Cuando el distribuidor este en movimiento
345              @Override
346             public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
347                 if (dataSnapshot.exists()) {
348                     List<Object> map = (List<Object>) dataSnapshot.getValue();
349                     double locationLat = 0;
350                     double locationLon = 0;
351                     //Se obtiene las coordenadas del distribuidor
352                     if (map.get(0) != null) {
353                         locationLat = Double.parseDouble(map.get(0).toString());
354                     }
355                     if (map.get(1) != null) {
356                         locationLon = Double.parseDouble(map.get(1).toString());
357                     }
358                     LatLng LatLng_distribuidor = new LatLng(locationLat, locationLon);
359                     //remover el marcador del distribuidor, de existir
360                     if (distribuidor_marker != null) {
361                         distribuidor_marker.remove();
```

Código 2.42 Cliente_Maps_Activity-Obtener dirección del distribuidor (1)

El Código 2.43 muestra cómo se construyen las ubicaciones del cliente y distribuidor a partir de las coordenadas de latitud y longitud (líneas 364 a 370); se calcula la distancia

entre el cliente y el distribuidor y cuando la distancia es menor a 50 metros salta una notificación en el móvil del cliente avisando que el distribuidor de gas llegó a entregar el pedido.

```
363         //coordenadas de la ubicacion cliente
364         Location location1 = new Location( provider: "");
365         location1.setLatitude(localizacion_pedido.latitude);
366         location1.setLongitude(localizacion_pedido.longitude);
367         //coordenadas de la ubicacion distribuidor
368         Location location2 = new Location( provider: "");
369         location2.setLatitude(LatLng_distribuidor.latitude);
370         location2.setLongitude(LatLng_distribuidor.longitude);
371         //la distancia entre el cliente y el distribuidor
372         float distancia = location1.distanceTo(location2);
373         if (distancia < 50) {
374             //crearnotifica();
375             vibrator.vibrate( milliseconds: 1000);
376             AlertDialog dialog = new AlertDialog.Builder( context: Cliente_Maps_Activity.this)
377                     .setTitle("GasAPP")
378                     .setMessage("Llegó el GAS")
379                     .setPositiveButton( text: "OK",  listener: null)
380                     .show();
```

Código 2.43 Cliente_Maps_Activity-Obtener ubicación del distribuidor (2)

- **Código para graficar las rutas entre cliente y distribuidor**

Al obtener la dirección del *Distribuidor*, se procede a graficar la ruta entre el cliente y el *Distribuidor* para que sea mostrada en la pantalla del usuario, esto mediante el Código 2.44. Se llama a la función *tiempo_pedido(),* para obtener la duración aproximada en la que se atenderá el pedido, luego se llama a la función *getUrl(),* para generar el URL (línea 469), y se ejecuta la tarea *TakeRequestDirection(),* para obtener la información y graficar la ruta en el mapa. Adicionalmente se enfocan los puntos de interés en la pantalla para lograr que la ruta se muestre en su totalidad adecuándose a las dimensiones del móvil (líneas 473 a 481).

```
466     //Para graficar las rutas
467     private void grafic_rutas(LatLng LatLng_distribuidor){
468         tiempo_pedido();
469         urla=getUrl(LatLng_distribuidor,ubicacion_cl,  directionMode: "driving");
470         TakeRequestDirection takeRequestDirection=new TakeRequestDirection();
471         String aux= String.valueOf(takeRequestDirection.execute(urla));
472         //Zoom en los puntoss
473         LatLngBounds.Builder builder= new LatLngBounds.Builder();
474         builder.include(LatLng_distribuidor); //ubica distribuidor
475         builder.include(ubicacion_cl); //ubica ciente
476         LatLngBounds bounds= builder.build();
477         //obtener las dimensiones de la pantalla
478         int width =getResources().getDisplayMetrics().widthPixels;
479         int padding= (int) (width*0.4);
480         CameraUpdate cameraUpdate=CameraUpdateFactory.newLatLngBounds(bounds, padding);
481         mMap.animateCamera(cameraUpdate);
```

Código 2.44 Cliente_Maps_Activity-Graficar la ruta distribuidor-cliente

- **Código para obtener el tiempo del pedido**

En el Código 2.45 se muestra la función *tiempo_pedido(),* la cual accede a la base de datos de Firebase, al subnodo *Pedido_cliente,* y obtiene el valor de la variable “Tiempo” (línea 452), la cual fue creada por el *Distribuidor*, este valor se coloca en una casilla de texto para que el cliente lo pueda visualizar (línea 460).

```
449     //Para obtener el tiempo del pedido
450     private void tiempo_pedido(){
451         //Se obtiene el tiempo del pedido
452         DatabaseReference drtiempo=FirebaseDatabase.getInstance().
453                 getReference().child("Usuarios").child("Distribuidores")
454                 .child(ID_distribuidor_encontrado).child("Pedido_cliente");
455         drtiempo.addListenerForSingleValueEvent(new ValueEventListener() {
456             @Override
457             public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
458                 Map<String, Object> map1=(Map<String, Object>) dataSnapshot.getValue();
459                 if (map1.get("Tiempo")!=null){
460                     tiempo_pedido.setText(Html.fromHtml( source: "<b>Tiempo pedido: </b>"
461                             +map1.get("Tiempo").toString())); }
```

Código 2.45 Cliente_Maps_Activity-Obtener tiempo del pedido

- **Código para construcción de la URL para la ruta entre cliente y distribuidor**

La URL sirve para realizar un requisito de dirección de Google Maps y se forma con los *strings* que se mencionan a continuación: coordenadas de origen, coordenadas de destino, modo de navegación, formato de salida y la clave de la API de Google Maps que se está usando en el proyecto, como se muestra en el Código 2.46.

```
637     //Construccion del URL
638     private String getUrl(LatLng origin, LatLng dest,String directionMode) {
639         // Coordenadas de origen
640         String str_origin = "origin=" + origin.latitude + "," + origin.longitude;
641         // Coordenadas de destino
642         String str_dest = "destination=" + dest.latitude + "," + dest.longitude;
643         // Modo
644         String mode = "mode="+directionMode;
645         // Construcción de los parametros
646         String parameters = str_origin + "&" + str_dest + "&" + mode;
647         // Formato de salida
648         String output = "json";
649         // Construcción del URL para el servicio web
650         String url = "https://maps.googleapis.com/maps/api/directions/" + output +
651                 "?" + parameters + "&key=" + "AIzaSyDejd0DRdx8Q4dZ3Fp_aPpI5VChwyH_d3w";
652         Log.d( tag: "mylo", msg: "inicio"+url);
653         return url;
```

Código 2.46 Cliente_Maps_Activity-Obtener URL para las rutas

- **Código para creación de Tarea Asíncrona para obtener y trazar las rutas**

Mediante el Código 2.47 es posible la obtención de las rutas desde la web y posterior trazado en el mapa del *cliente* con la tarea *AsyncTask,* la misma que ejecuta las acciones antes mencionadas en segundo plano (línea 688). En esta tarea se llama a la función

downloadUrl() en la línea 691, y solamente luego de ser ejecutada se llama a la función *taskParser()* para el trazado de rutas.

```
686     public class TakeRequestDirection extends AsyncTask<String, Void, String> {
687         @Override
688         protected String doInBackground(String... strings) {
689             String rsS="";
690             try {
691                 rsS=downloadUrl(strings[0]);
692             } catch (IOException e) {
693                 e.printStackTrace();
694             }
695             return rsS;
696         }
697         @Override
698         protected void onPostExecute(String s) {
699             super.onPostExecute(s);
700             TaskParser taskParser= new TaskParser();
701             taskParser.execute(s);
702         }
```

Código 2.47 Cliente_Maps_Activity-Creación tarea en segundo plano para descargar información de rutas

- **Código para obtener la información de la dirección del servicio web**

Con el Código 2.48 se muestra la creación de una conexión HTTP con la URL de los puntos entre los que se desea obtener la ruta (línea 661). Al conectarse al servicio web se lee la información que devuelve la conexión y se almacena en una variable dentro de la aplicación (líneas 667 a 674). Una vez se hayan extraído todos los datos del servicio web se cierra la conexión http y la función retorna la información obtenida (línea 681).

```
655 //Descargar información de rutas desde el servidor a la aplicación
656 private String downloadUrl(String strUrl) throws IOException {
657     String data = "";
658     InputStream iStream = null;
659     HttpURLConnection urlConnection = null;
660     try {
661         URL url = new URL(strUrl);
662         // Creación de una conexón http para comunicarse con la url
663         urlConnection = (HttpURLConnection) url.openConnection();
664         // Conectandose a la url
665         urlConnection.connect();
666         // Leyendo la información de la url
667         iStream = urlConnection.getInputStream();
668         BufferedReader br = new BufferedReader(new InputStreamReader(iStream));
669         StringBuffer sb = new StringBuffer();
670         String line = "";
671         while ((line = br.readLine()) != null) {
672             sb.append(line);
673         }
674         data = sb.toString();
675         br.close();
676     } catch (Exception e) {
677     } finally {
678         iStream.close();
679         urlConnection.disconnect();
680     }
681     return data;
```

Código 2.48 Cliente_Maps_Activity-Descargar información con una conexión HTTP

- **Código para creación de tarea asíncrona para análisis de datos de direcciones**

El Código 2.49 muestra el proceso de creación de una nueva tarea asíncrona para el análisis de los datos en segundo plano (línea 707). La tarea llama a la función *DataParser(),* y ésta devuelve el trazado de rutas en el mapa (línea 718).

```
705    public class TaskParser extends AsyncTask<String, Integer, List<List<HashMap<String, String>>>> {
706        @Override
707        protected List<List<HashMap<String, String>>> doInBackground(String... strings) {
708            JSONObject jObject=null;
709            List<List<HashMap<String, String>>> routes = null;
710            try {
711                jObject = new JSONObject(strings[0]);
712                DataParser parser = new DataParser();
713                // Análisis de datos
714                routes = parser.parse(jObject);
715            } catch (Exception e) {
716                e.printStackTrace();
717            }
718            return routes;
```

Código 2.49 Cliente_Maps_Activity-Creación de tarea en segundo plano para graficar rutas

- **Código para escuchar cuando finaliza un pedido**

Al realizar un pedido y encontrar un *Distribuidor*, la función mostrada en el Código 2.50 se activa y se mantiene escuchando la base de datos de Firebase (línea 494); si el subnodo *Pedido_cliente,* que se encuentra bajo el ID del *Distribuidor* desaparece, significa que el pedido ha terminado y se llama a la función *fin_pedido()* para terminar el pedido actual (línea 500).

```
484    //Para escuchar cuando termina el pedido
485    private void fin_pedido_aux(){
486        DatabaseReference distr_ref=FirebaseDatabase.getInstance().getReference().child("Usuarios")
487                .child("Distribuidores").child(ID_distribuidor_encontrado).child("Pedido_cliente");
488        HashMap  map=new HashMap();
489        map.put("Estado",estado_ped);//estado del pedido
490        distr_ref.updateChildren(map);//actualizo información bajo el Distribuidor
491
492        fin_ped_ref=FirebaseDatabase.getInstance().getReference().child("Usuarios").child("Distribuidores")
493                .child(ID_distribuidor_encontrado).child("Pedido_cliente");
494        fin_ped_ref_listener= fin_ped_ref.addValueEventListener(new ValueEventListener() {
495            @Override
496            public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
497                if(dataSnapshot.exists()){
498                }else{
499                    auxiliarfin=0;
500                    fin_pedido();
```

Código 2.50 Cliente_Maps_Activity-Escuhar fin de pedido

- **Código para finalizar el pedido**

Al finalizar un pedido, todos los valores de las variables usadas en el mismo vuelven a sus valores por *default*, se borra de la base de datos de Firebase el pedido del *cliente* para

evitar confusiones (líneas 521 a 524). Finalmente, se borran los datos del *Distribuidor* designado al pedido y se eliminan los marcadores y rutas trazadas en el mapa (línea 526), como se observa en el Código 2.51.

```
507     //para finalizar el pedido
508     private void fin_pedido(){
509         cancelar=false;
510         geoQuery.removeAllListeners();
511         if (auxiliarfin==1) {
512             //ubic_distrib_ref.removeEventListener(ubic_distrib_ref_listener);
513         }else {
514             ubic_distrib_ref.removeEventListener(ubic_distrib_ref_listener);
515             fin_ped_ref.removeEventListener(fin_ped_ref_listener);
516         };
517         distribuidor_encontrado=false;
518         radio=1;
519         pedir_gas.setText("Realizar Pedido");
520         //Borrar de la base de datos el pedido
521         String ID_usuario = FirebaseAuth.getInstance().getCurrentUser().getUid();
522         DatabaseReference ref = FirebaseDatabase.getInstance().getReference( path: "Pedidos");
523         GeoFire geoFire = new GeoFire(ref);
524         geoFire.removeLocation(ID_usuario);
525         //se quitan los marcadores del map
526         if (marc_fin!=null){
527             marc_fin.remove(); }
528         if (distribuidor_marker!=null){
529             distribuidor_marker.remove(); }
```

Código 2.51 Cliente_Maps_Activity-Finalizar pedido

- **Código para borrar la ruta del mapa del cliente**

La eliminación de las rutas entre el *Distribuidor* y el *cliente* del mapa es parte del proceso de finalización de un pedido; para esto se usa la función *borrarPolyline(),* que se muestra en el Código 2.52, la cual remueve todas las líneas trazadas en el mapa del *cliente*.

```
//Para borrar las rutas en el mapa
public void borrarPolyline(){
    for (Polyline line : polylines){
        line.remove();
    }
    polylines.clear();
}
```

Código 2.52 Cliente_Maps_Activity-Eliminar ruta del mapa del cliente

2.7.8 DISTRIBUIDOR_MAPS_ACTIVITY.JAVA

Es la actividad principal del *Distribuidor*, desde la cual es posible realizar la recepción y aceptación de pedidos de gas; además permite recibir la retroalimentación del *cliente* que va a realizar el pedido.

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para inicializar las variables**

En el Código 2.53 se muestra la inicialización de las variables que permitirán el funcionamiento de la interfaz principal de la aplicación del *Distribuidor*, entre ellas se tienen casillas de texto, botones para aceptar o rechazar un pedido y para habilitar al *Distribuidor* el estado de trabajando o no trabajando; además de variables para mostrar el menú lateral y el soporte para cargar el mapa (línea 150).

```
150         SupportMapFragment mapFragment = (SupportMapFragment) getSupportFragmentManager
151                 .findFragmentById(R.id.map);
152         mapFragment.getMapAsync( onMapReadyCallback: this);
153
154         polylines = new ArrayList<>();
155         //Variable para dirección
156         tedireccioncli = (TextView) findViewById(R.id.direccion_cliente);
157         //Variables para mostrar la informacion del cliente
158         info_cliente= (LinearLayout) findViewById(R.id.info_clientely);
159         info_cliente.setVisibility(View.GONE);
160         foto_cliente= (ImageView) findViewById(R.id.foto_clientely);
161         nombre_cli= (TextView) findViewById(R.id.nombre_clientely);
162         telefono_cli= (TextView) findViewById(R.id.telefono_clientely);
163         numero_tanques= (TextView) findViewById(R.id.ntanquesly);
164         costo_pedido= (TextView) findViewById(R.id.costo_ped);
165         tiempo_pedido= (TextView) findViewById(R.id.tiempo_ped);
166         //Variables para el botón para aceptar o rechazar el pedido
167         LY_aceptar_rechazar=(LinearLayout) findViewById(R.id.botones_pedido);
168         aceptar_pedido=(Button) findViewById(R.id.botaceptar);
169         rechazar_pedido=(Button) findViewById(R.id.botrechazar);
170         pedido_entrante=(Button) findViewById(R.id.botstatus_ped);
171         entreg_pedido=(Button) findViewById(R.id.botpedientr);
172         vibrator=(Vibrator) this.getSystemService(Context.VIBRATOR_SERVICE);
```

Código 2.53 Distribuidor_Maps_Activity-Inicialización de variables

- **Código para activar o desactivar la disponibilidad del distribuidor**

El *Distribuidor* al entrar a la aplicación tiene un botón tipo *Switch* que permite cambiar el estado del *Distribuidor* entre disponible o no disponible. En el Código 2.54 se aprecia el funcionamiento del botón *Switch*: si el *Switch* se encuentra encendido (valor *true*), el *Distribuidor* aparecerá en el nodo *DistribuidorDisponible,* en la base de datos de Firebase para poder recibir pedidos (línea 205); por otro lado si el *Switch* se encuentra apagado (valor *false*), el *Distribuidor* será removido del nodo *DistribuidorDisponible,* en la base de datos de Firebase y no podrá recibir pedidos (línea 219).

```
194     //para el botón switch de habilitado para trabajar o deshabilitado
195     switch1.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
196         @SuppressLint("SetTextI18n")
197         @Override
198         public void onCheckedChanged(CompoundButton compoundButton, boolean cambio) {
199             if (cambio){
200                 con_des=1;
201                 String ID_usuario = FirebaseAuth.getInstance().getCurrentUser().getUid();
202                 //Referencias para los conductores trabajando y los conductores disponibles
203                 DatabaseReference Disponible_ref= FirebaseDatabase.getInstance().
204                         getReference( path: "DistribuidorDisponible");
205                 DatabaseReference ref= FirebaseDatabase.getInstance().getReference( path: "DistribuidorDisponible");
206                 GeoFire geoFire=new GeoFire(ref);
207                 geoFire.setLocation(ID_usuario,new GeoLocation(lat,lon));
208                 Toast.makeText(getBaseContext(), text: "On", Toast.LENGTH_SHORT).show();
209                 estadosw.setText("Conectado");
216             }else{
217                 con_des=0;
218                 ID_distr = FirebaseAuth.getInstance().getCurrentUser().getUid();
219                 DatabaseReference ref = FirebaseDatabase.getInstance().getReference( path: "DistribuidorDisponible");
220                 GeoFire geoFire = new GeoFire(ref);
221                 geoFire.removeLocation(ID_distr);
222                 Toast.makeText(getBaseContext(), text: "Off",Toast.LENGTH_SHORT).show();
223                 estadosw.setText("Desconectado");
```

Código 2.54 Distribuidor_Maps_Activity-Botón disponibilidad del distribuidor

- **Código para aceptar un pedido**

Al ingresar una nueva solicitud de pedido de un *cliente*, si el *Distribuidor* decide aceptar el pedido, cambia su estado en la base de datos de Firebase a *DistribuidorTrabajando,* y se actualizan las coordenadas del usuario, como se observa en el Código 2.55.

```
//Boton para aceptar el pedido
aceptar_pedido.setOnClickListener((view) → {
        cancelar_pedido.setVisibility(View.VISIBLE);
        LY_aceptar_rechazar.setVisibility(View.GONE);
        String ID_usuario = FirebaseAuth.getInstance().getCurrentUser().getUid();
        //Referencias para los conductores trabajando y los conductores disponibles
        DatabaseReference Disponible_ref= FirebaseDatabase.getInstance().getReference( path: "DistribuidorDisponible");
        DatabaseReference Trabajando_ref= FirebaseDatabase.getInstance().getReference( path: "DistribuidorTrabajando");
        GeoFire geoFiredispo=new GeoFire(Disponible_ref);
        GeoFire geoFiretraba=new GeoFire(Trabajando_ref);
        geoFiredispo.removeLocation(ID_usuario);
        geoFiretraba.setLocation(ID_usuario,new GeoLocation(lat,lon));
});
```

Código 2.55 Distribuidor_Maps_Activity-Botón aceptar pedido

- **Código para rechazar un pedido**

Al ingresar una nueva solicitud de pedido de un *cliente*, si el *Distribuidor* decide rechazar el pedido, al presionar el botón se llama a la función *fin_pedido_dis(),* para finalizar el proceso del pedido, como se aprecia en el Código 2.56.

```
rechazar_pedido.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        fin_pedido_dis();
    }
```

Código 2.56 Distribuidor_Maps_Activity-Botón rechazar pedido

- **Código para el botón de cancelar el pedido**

En el Código 2.57, se muestra que al presionar el botón *Cancelar Pedido,* aparece un cuadro de texto pidiendo la confirmación del usuario para cancelar el pedido (líneas 257 a

261). Si se confirma la cancelación del pedido, se borra el pedido de la base de datos de Firebase (línea 267) y se llama a la función *fin_pedido_dis(),* para terminar el pedido (línea 273).

```
252     //Botón para cancelar pedido
253     cancelar_pedido =(Button)findViewById(R.id.botcancelar);
254     cancelar_pedido.setOnClickListener(new View.OnClickListener() {
255         @Override
256         public void onClick(View view) {
257             AlertDialog dialog = new AlertDialog.Builder( context: Distribuidor_Maps_Activity.this)
258                     .setTitle("Cancelar pedido")
259                     .setMessage("¿Desea cancelar el pedido actual?")
260                     .setNegativeButton( text: "No", listener: null)
261                     .setPositiveButton( text: "Si", new DialogInterface.OnClickListener() {
262                         @Override
263                         public void onClick(DialogInterface dialog, int which) {
264                             pedido_entrante.setVisibility(View.VISIBLE);
265                             cancelar_pedido.setVisibility(View.GONE);
266                             if(enpedido==1){ //si se cancela un pedido que está siendo atendido
267                                 DatabaseReference distr_ref=FirebaseDatabase.getInstance().
268                                         getReference().child("Usuarios").child("Distribuidores")
269                                         .child(ID_distr).child("Pedido_cliente");
270                                 HashMap  map1=new HashMap();
271                                 map1.put("Estado",3);//estado del pedido
272                                 distr_ref.updateChildren(map1);//actualizo información bajo el Distribuidor
273                                 fin_pedido_dis();
274                             }
275                             fin_pedido_dis();
```

Código 2.57 Distribuidor_Maps_Activity-Botón para cancelar el pedido

- **Código para el botón de entregar el pedido**

En el Código 2.58 se observa que al presionar el botón *Entregar Pedido,* se llama a dos funciones, la primera *guardar_viaje(),* la cual guarda la información del pedido y *fin_pedido_dis(),* para finalizar el pedido.

```
//Botón para entregar el pedido
entreg_pedido.setOnClickListener((view) → {
        guardar_viaje();
        fin_pedido_dis();
});
obtener_cliente();
```

Código 2.58 Distribuidor_Maps_Activity-Botón para entregar el pedido

- **Código para mostrar el mapa en la pantalla del distribuidor**

La pantalla principal de la aplicación muestra el mapa del sector donde se encuentra el *Distribuidor*, se reutiliza el Código 2.35 utilizando en Cliente_Maps_Activity.java, puesto que cumple la misma funcionalidad de configuración del mapa.

Adicionalmente, como se muestra en el Código 2.59, se crean referencias en la base de datos de Firebase en los nodos *DistribuidorDisponible y DistribuidorTrabajando* (líneas 679 a 682)*;* cuando el *Distribuidor* se encuentre atendiendo un pedido, la variable *trabajando* tiene el valor de 1, el ID del *Distribuidor* se incluirá en el nodo *DistribuidorTrabajando* y se eliminará del nodo *DistribuidorDisponible,* evitando que le lleguen más notificaciones de pedidos (línea 684 a 689). Mientras que, si la variable *trabajando* tiene un valor diferente a

1, significa que no está atendiendo ningún pedido al momento en cuyo caso el ID del distribuidor se incluirá en el nodo *DistribuidorDisponible,* y se eliminará del nodo *DistribuidorTrabajando,* permitiéndole recibir pedidos (línea 690 a 693).

```
677     if (con_des==1){
678         //Referencias para los conductores trabajando y los conductores disponibles
679         DatabaseReference Disponible_ref= FirebaseDatabase.getInstance().getReference( path: "DistribuidorDisponible");
680         DatabaseReference Trabajando_ref= FirebaseDatabase.getInstance().getReference( path: "DistribuidorTrabajando");
681         GeoFire geoFiredispo=new GeoFire(Disponible_ref);
682         GeoFire geoFiretraba=new GeoFire(Trabajando_ref);
683         //obtener y enviar las coordenadas al servidor FireBase
684         if (trabajando==1){
685             geoFiredispo.removeLocation(ID_usuario);
686             geoFiretraba.setLocation(ID_usuario,new GeoLocation(lat,lon));
687             borrarPolyline();
688             graficar_rutas(LatLng_cliente);
689             obtener_dist(LatLng_cliente, ubicacion);
690         }else{
691             mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(ubicacion,  v: 17));
692             geoFiretraba.removeLocation(ID_usuario);
693             geoFiredispo.setLocation(ID_usuario,new GeoLocation(lat,lon));
```

Código 2.59 Distribuidor_Maps_Activity-Mostrar el mapa en la pantalla del distribuidor

- **Código para obtener un cliente**

En el Código 2.60 se explica el proceso para obtener un cliente: el *Distribuidor* entra en estado de escucha, en busca de un *cliente* dentro de la base de datos de Firebase en el subnodo *Pedido_cliente()* (líneas 300 a 302). Al existir un pedido se obtiene el ID del *cliente* y se llama a las funciones *obtener_info_cliente(), obtener_ubic_cliente() y notificación(),* para continuar con el proceso de entrega del pedido solicitado (línea 306). Finalmente, se llama a la función *fin_pedido_dis(),* si el subnodo del pedido del cliente desaparece, indicando que se ha completado o se ha cancelado el pedido.

```
297     //Obtener cliente que hace pedido
298     private void obtener_cliente(){
299         final String ID_distribuidor=FirebaseAuth.getInstance().getCurrentUser().getUid();
300         DatabaseReference cliente_ref = FirebaseDatabase.getInstance().getReference().child("Usuarios")
301                 .child("Distribuidores").child(ID_distribuidor).child("Pedido_cliente").child("ID_cliente");
302         cliente_ref.addValueEventListener(new ValueEventListener() {
303             //Si existe un pedido se escucha la base de datos
304             @Override
305             public void  onDataChange(@NonNull DataSnapshot dataSnapshot) {
306                 if (dataSnapshot.exists()){
307                     ID_cliente=dataSnapshot.getValue().toString();
308                     //crearnotifica();
309                     notificacion();
310                     trabajando=1;
311                     obtener_info_cliente();
312                     obtener_ubic_cliente();
313                 }else {
314                     fin_pedido_dis();
```

Código 2.60 Distribuidor_Maps_Activity-Obtener un cliente

- **Código para recibir la notificación de un pedido**

Al registrarse un nuevo pedido por parte de un *cliente*, se notifica al *Distribuidor* más cercano mostrando una ventana desde la cual se puede aceptar o rechazar el pedido de gas, como se muestra en el Código 2.61.

```
//Notificacion que llego un pedido de gas
public void notificacion(){
    //crearnotifica();
    vibrator.vibrate( milliseconds: 1000);
    AlertDialog dialog = new AlertDialog.Builder( context: Distribuidor_Maps_Activity.this)
        .setTitle("GasAPP")
        .setMessage("Nuevo Pedido")
        .setPositiveButton( text: "Ver pedido",  listener: null)
        .show();
    pedido_entrante.setVisibility(View.GONE);
    LY_aceptar_rechazar.setVisibility(View.VISIBLE);
}
```

Código 2.61 Distribuidor_Maps_Activity-Notificación de pedido

- **Código para obtener información del cliente a atender**

En el Código 2.62 se procede a acceder a la base de datos de Firebase al nodo *Clientes,* con el ID del cliente que realizó el pedido (línea 448) y se extrae su información, la cual es compartida con el *Distribuidor*, incluyendo el nombre, teléfono y foto del cliente que recibirá el pedido (líneas 455 a 463).

```
446  //Obtener la informacion del cliente
447  private void obtener_info_cliente(){
448      DatabaseReference mclientedatabase=FirebaseDatabase.getInstance().
449              getReference().child("Usuarios").child("Clientes").child(ID_cliente);
450      mclientedatabase.addListenerForSingleValueEvent(new ValueEventListener() {
451          @Override
452          public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
453              if (dataSnapshot.exists()&&dataSnapshot.getChildrenCount()>0){
454                  Map<String, Object> map=(Map<String, Object>) dataSnapshot.getValue();
455                  if (map.get("nombre")!=null){
456                      nombre_aux=map.get("nombre").toString();
457                      nombre_cli.setText(Html.fromHtml( source: "<b>Cliente: </b>"+nombre_aux)); }
458                  if (map.get("telefono")!=null){
459                      telefono_cli.setText(Html.fromHtml( source: "<b>Teléfono: </b>"
460                              +map.get("telefono").toString())); }
461                  if(map.get("fotoUrl")!=null){
462                      Glide.with(getApplication()).load(map.get("fotoUrl").toString()).into(foto_cliente); }
463                  info_cliente.setVisibility(View.VISIBLE);
```

Código 2.62 Distribuidor_Maps_Activity-Obtener información del cliente (1)

Adicionalmente, se busca en la base de datos de Firebase el subnodo *Pedido_cliente,* dentro de *Distribuidores*, para poder obtener el número de cilindros de gas que el cliente solicita y mostrar al *Distribuidor*, tal como se aprecia en el Código 2.63. Finalmente, en el Código 2.64, con el valor del cilindro de gas recuperado, se multiplica por el número de cilindros solicitados y se obtiene el valor final del pedido (línea 494), para ser mostrado al *Distribuidor*, como se observa

```
469  String ID_dist=FirebaseAuth.getInstance().getCurrentUser().getUid();
470  DatabaseReference cilin=FirebaseDatabase.getInstance().getReference().child("Usuarios").child("Distribuidores")
471          .child(ID_dist).child("Pedido_cliente");
472  cilin.addListenerForSingleValueEvent(new ValueEventListener() {
473      @Override
474      public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
475          if (dataSnapshot.exists()&dataSnapshot.getChildrenCount()>0){
476              Map<String, Object> map1=(Map<String, Object>) dataSnapshot.getValue();
477              if (map1.get("Numero_cilindros")!=null){
478                  ncilcli=map1.get("Numero_cilindros").toString();
479                  numero_tanques.setText(Html.fromHtml( source: "<b>Cilindros: </b>"+ncilcli));
```

Código 2.63 Distribuidor_Maps_Activity-Obtener información del cliente (2)

```
486     DatabaseReference precio=FirebaseDatabase.getInstance().getReference();
487     precio.addListenerForSingleValueEvent(new ValueEventListener() {
488         @Override
489         public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
490             Map<String, Object> map2=(Map<String, Object>) dataSnapshot.getValue();
491             String preciouni=map2.get("Precio_tanque").toString();
492             int cilindros_fin=Integer.parseInt(ncilcli);
493             DecimalFormat formato= new DecimalFormat( pattern: "$0.00");
494             preciofin=cilindros_fin*Double.parseDouble(preciouni);
495             costo_pedido.setText(Html.fromHtml( source: "<b>Valor pedido: </b>"+formato.format(preciofin)));
```

Código 2.64 Distribuidor_Maps_Activity-Obtener información del cliente (3)

- **Código para obtener la ubicación del cliente**

La ubicación del *cliente* es necesaria para conocer donde será entregado el pedido, para conseguir esta ubicación se usa el Código 2.65. Se accede a la base de datos de Firebase, al nodo *Pedidos,* y con el ID del *cliente* se crea un nuevo evento que obtenga las coordenadas del *cliente* (líneas 504 a 519). Se llama a la función *obtener_dis(),* para obtener la distancia entre el *cliente* y el *distribuidor* y a la función *graficar_rutas(),* para proceder con el trazado de las rutas en el mapa (líneas 523 y 525).

```
502     //Obtener la ubicacion del cliente
503     private void obtener_ubic_cliente(){
504         ubic_client_ref=FirebaseDatabase.getInstance().getReference()
505                 .child("Pedidos").child(ID_cliente).child("1");
506         ubic_client_ref_listener=ubic_client_ref.addValueEventListener(new ValueEventListener() {
507             @Override
508             public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
509                 if (dataSnapshot.exists()&&!ID_cliente.equals("")){
510                     List<Object> map=(List<Object>) dataSnapshot.getValue();
511                     double locationLat=0;
512                     double locationLon=0;
513                     if(map.get(0)!=null){
514                         locationLat=Double.parseDouble(map.get(0).toString());
515                     }
516                     if (map.get(1)!=null){
517                         locationLon=Double.parseDouble(map.get(1).toString());
518                     }
519                     LatLng_cliente= new LatLng(locationLat,locationLon);
520                     setLocation(LatLng_cliente);
521                     cliente_marcador=mMap.addMarker(new MarkerOptions().position(LatLng_cliente).title("Cliente"));
522                     prim=1;
523                     obtener_dist(LatLng_cliente, ubicacion);
524                     //obtener las rutas desde el distribuidor al cliente
525                     graficar_rutas(LatLng_cliente);
```

Código 2.65 Distribuidor_Maps_Activity-Obtener ubicación del cliente

- **Código para obtener distancia entre el cliente y el distribuidor**

En el Código 2.66, se calcula la distancia de la misma forma que en Cliente_Maps_Activity.java. Sin embargo, en este caso, cuando la distancia es menor a 100 metros desaparece el botón de cancelar pedido y aparece en su lugar un botón para completar el pedido (línea 574).

```
556     //Obtener distancia entre puntos
557     private void obtener_dist(LatLng LatLng_cliente,LatLng ubicacion){
558         if (prim==1){
559             origen=ubicacion;
560             prim=2;
561         }
562         //ubicacion cliente
563         Location location1 = new Location( provider: "");
564         location1.setLatitude(LatLng_cliente.latitude);
565         location1.setLongitude(LatLng_cliente.longitude);
566         //ubicacion distribuidor
567         Location location2 = new Location( provider: "");
568         location2.setLatitude(ubicacion.latitude);
569         location2.setLongitude(ubicacion.longitude);
570         //la distancia entre el cliente y el distribuidor
571         distancia = location1.distanceTo(location2);
572         float distanciai=distancia;
573         //Cuando el distribuidor llega
574         if (distancia < 100) {
575             cancelar_pedido.setVisibility(View.GONE);
576             entreg_pedido.setVisibility(View.VISIBLE);
577         }
```

Código 2.66 Distribuidor_Maps_Activity-Obtener distancia entre cliente y distribuidor

- **Código para mostrar el tiempo del pedido**

Como se puede observar en el Código 2.67, cuando se obtiene el tiempo que tomará el pedido en llegar, se lo muestra al *Distribuidor* y se sube la información a la base de datos de Firebase (líneas 586 a 591), para que el *cliente* pueda tener acceso a esta información.

```
579     //Para el tiempo del pedido
580     private void tiempo_depedido(){
581         if(!duracion.equals("")){
582             if (prim==2){
583                 durapedido=duracion;
584                 tiempo_pedido.setText(Html.fromHtml( source: "<b>Tiempo pedido: </b>"+durapedido));
585                 //subo a firebase para que el usuario vea
586                 String ID_usuario = FirebaseAuth.getInstance().getCurrentUser().getUid();
587                 DatabaseReference distr_ref=FirebaseDatabase.getInstance().getReference().child("Usuarios")
588                         .child("Distribuidores").child(ID_usuario).child("Pedido_cliente");
589                 HashMap map=new HashMap();
590                 map.put("Tiempo",durapedido);
591                 distr_ref.updateChildren(map);
592                 prim=0;
```

Código 2.67 Distribuidor_Maps_Activity-Informar tiempo de pedido

- **Código para obtener la fecha y hora del pedido**

En el Código 2.68 se aprecia la obtención de la fecha del pedido mediante la función nativa de Android Studio, *currentTimeMillis()* (línea 378). Esta función devuelve la fecha y hora del sistema en milisegundos, la cual es almacenada en una variable del tipo Long debido a la naturaleza de la información. Finalmente, se pasa el *timestamp* a la función *obtenerFecha()* (línea 384) para transformarla en un formato legible para los usuarios, que incluye el día, mes, año y la hora del pedido.

```
377     //obtiene la fecha cuando se hace un pedido
378     private Long obtener_timestamp() {
379         Long timestamp=System.currentTimeMillis()/1000;
380         Log.d( tag: "timestamp",timestamp.toString());
381          return timestamp;
382     }
383     // para transformar el timestamp en una fecha legible
384     private String obtenerFecha(Long timestamp) {
385         @SuppressLint("SimpleDateFormat")
386         SimpleDateFormat dateFormat= new SimpleDateFormat( pattern: "E dd MMM yyyy HH:mm:ss");
387         Date dateT=new Date(timestamp*1000);
388         String fechapedido= dateFormat.format(dateT);
389         Log.d( tag: "fecha",  msg: "la fechas es "+fechapedido);
390         return fechapedido;
```

Código 2.68 Distribuidor_Maps_Activity-Obtener fecha del pedido

- **Código para guardar un pedido**

La función *guardar_pedido(),* se muestra en el Código 2.69; la cual crea un nuevo nodo con el ID del pedido atendido en la base de datos de Firebase, dentro de *Historial* (línea 359), y almacena la información referente incluyendo: el *Distribuidor* que atendió el pedido, el cliente que realizó el pedido, la fecha, el número de cilindros, el precio, el tiempo, duración y ubicaciones en las cuales tuvo lugar el pedido (líneas 364 a 375).

```
352 //guardar historial de pedidos
353 public void guardar_pedido(){
354     String ID_usuario = FirebaseAuth.getInstance().getCurrentUser().getUid();
355     DatabaseReference distr_ref=FirebaseDatabase.getInstance().getReference().child("Usuarios")
356             .child("Distribuidores").child(ID_usuario).child("Historial");
357     DatabaseReference clien_ref=FirebaseDatabase.getInstance().getReference().child("Usuarios")
358             .child("Clientes").child(ID_cliente).child("Historial");
359     DatabaseReference historialref=FirebaseDatabase.getInstance().getReference().child("Historial");
360     String ID_pedido=historialref.push().getKey();
361     distr_ref.child(ID_pedido).setValue(true);
362     clien_ref.child(ID_pedido).setValue(true);
363
364     HashMap map=new HashMap();
365     map.put("Distribuidor", ID_usuario);
366     map.put("Cliente",ID_cliente);
367     map.put("Fecha",obtenerFecha(obtener_timestamp()));
368     map.put("Cilindros",ncilcli);
369     map.put("Precio",preciofin);
370     map.put("Tiempo",durapedido);
371     map.put("Lugar/Destino/lat",LatLng_cliente.latitude);
372     map.put("Lugar/Destino/lon",LatLng_cliente.longitude);
373     map.put("Lugar/Origen/lat",origen.latitude);
374     map.put("Lugar/Origen/lon",origen.longitude);
375     historialref.child(ID_pedido).updateChildren(map);
```

Código 2.69 Distribuidor_Maps_Activity-Guardar información de un pedido

- **Código al finalizar un pedido**

Al igual que en Cliente_Maps_Activity.java, al finalizar un pedido, todos los valores de las variables usadas en el mismo vuelven a sus valores por *default* y se borra de la base de datos de Firebase la información que ya no es necesaria (líneas 401 a 407), como se muestra en el Código 2.70.

```
394     public void fin_pedido_dis(){
395         pedido_entrante.setText("Esperando Pedido");
396         pedido_entrante.setVisibility(View.VISIBLE);
397         cancelar_pedido.setVisibility(View.GONE);
398         LY_aceptar_rechazar.setVisibility(View.GONE);
399         borrarPolyline(); //Borrar ruta
400         //borrar el pedido de la base de datos de Firebase
401         String ID_usuario = FirebaseAuth.getInstance().getCurrentUser().getUid();
402         DatabaseReference distr_ref=FirebaseDatabase.getInstance().getReference().child("Usuarios")
403                 .child("Distribuidores").child(ID_usuario).child("Pedido_cliente");
404         distr_ref.removeValue();
405         DatabaseReference ref= FirebaseDatabase.getInstance().getReference( path: "Pedidos");
406         GeoFire geoFire=new GeoFire(ref);
407         geoFire.removeLocation(ID_cliente);
408         ID_cliente="";//asigna como ID del cliente nula
409         trabajando=0;
410         prim=0;
411         enpedido=0;
412         durapedido="";
413         //borra la info del cliente
414         info_cliente.setVisibility(View.GONE);
415         entreg_pedido.setVisibility(View.GONE);
416         tedireccioncli.setText("");
417         nombre_cli.setText("");
418         telefono_cli.setText("");
419         foto_cliente.setImageResource(R.mipmap.ic_perfil_pre);
420         //Borra el marcador del cliente
421         if(cliente_marcador!=null){
422             cliente_marcador.remove(); }
```

Código 2.70 Distribuidor_Maps_Activity-Fin de un pedido

2.7.9 DATA_PARSER

Con el fin de graficar la ruta entre el cliente y distribuidor en el mapa del usuario, se llama a esta función como parte de las tareas asíncronas desarrolladas en segundo plano en las actividades Cliente_Maps_Activity.java y Distribuidor_Maps_Activity.java. Utiliza los datos recuperados del servidor web, que se encuentran en formato JSON y los analiza para graficar las rutas.

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para obtener las polilíneas de los datos web**

En el Código 2.71 se muestra el análisis sobre los datos web recuperados, en las líneas 47 a 56 se obtienen los puntos de la polilínea que representa la ruta a seguir. Luego se llama a la función *decodePoly()* para transformar los puntos en coordenadas y se guarda la información en una lista como se observa en la línea 63, para proceder a devolverlos a las actividades principales del cliente y distribuidor.

```
public List<List<HashMap<String, String>>> parse(JSONObject jObject) {
    List<List<HashMap<String, String>>> routes = new ArrayList<>();
    JSONArray jRoutes;
    JSONArray jLegs;
    JSONArray jSteps;
    try {
        jRoutes = jObject.getJSONArray( name: "routes");
        /* Recorre all routes */
        for (int i = 0; i < jRoutes.length(); i++) {
            jLegs = ((JSONObject) jRoutes.get(i)).getJSONArray( name: "legs");
            List path = new ArrayList<>();
            /* recorre all legs */
            for (int j = 0; j < jLegs.length(); j++) {
                jSteps = ((JSONObject) jLegs.get(j)).getJSONArray( name: "steps");
                /* recorre all steps */
                for (int k = 0; k < jSteps.length(); k++) {
                    String polyline = "";
                    polyline = (String) ((JSONObject) ((JSONObject) jSteps.get(k)).get("polyline")).get("points");
                   List<LatLng> list = decodePoly(polyline);
                    /* recorre all points */
                    for (int l = 0; l < list.size(); l++) {
                        HashMap<String, String> hm = new HashMap<>();
                        hm.put("lat", Double.toString((list.get(l)).latitude));
                        hm.put("lng", Double.toString((list.get(l)).longitude));
                        path.add(hm);
                    }
                }
                routes.add(path);
```

Código 2.71 DataParser-Obtener polilíneas de ruta

- **Código para decodificar las polilíneas**

En el Código 2.72 se muestra la decodificación de las polilíneas recuperadas, en este proceso se mapea la información de polilíneas en coordenadas de latitud y longitud y se devuelve la información en una lista para su posterior uso. Esta función puede ser encontrada y analizada a mayor detalle en la documentación de la API de direcciones y mapas de Google.

```
@ private List<LatLng> decodePoly(String encoded) {
    List<LatLng> poly = new ArrayList<>();
    int index = 0, len = encoded.length();
    int lat = 0, lng = 0;
    while (index < len) {
        int b, shift = 0, result = 0;
        do {
            b = encoded.charAt(index++) - 63;
            result |= (b & 0x1f) << shift;
            shift += 5;
        } while (b >= 0x20);
        int dlat = ((result & 1) != 0 ? ~(result >> 1) : (result >> 1));
        lat += dlat;

        shift = 0;
        result = 0;
        do {
            b = encoded.charAt(index++) - 63;
            result |= (b & 0x1f) << shift;
            shift += 5;
        } while (b >= 0x20);
        int dlng = ((result & 1) != 0 ? ~(result >> 1) : (result >> 1));
        lng += dlng;
        LatLng p = new LatLng((((double) lat / 1E5)),
                (((double) lng / 1E5)));
        poly.add(p);
    }
    return poly;
```

Código 2.72 DataParser-Decodificar polilíneas en coordenadas

2.7.10 PERFIL_ACTIVITY.JAVA Y DISTRIBUIDOR_PERFIL.JAVA

Esta actividad permite modificar los datos del usuario ingresados anteriormente en el proceso de registro. Los datos que se pueden modificar por parte del *cliente* son: nombre, teléfono, dirección de referencia y foto de perfil del usuario. Mientras que los datos que pueden ser modificados por parte del *Distribuidor* son: nombre, teléfono, placas del vehículo y foto de perfil del usuario

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para inicializar las variables**

Se inicializan las variables para la modificación de datos; las casillas de texto son para el ingreso de nombre, teléfono y dirección de referencia. Además se inicializan los botones de guardar, atrás y un botón adicional para modificar la foto de perfil, como se observa en Código 2.73.

```
//Inicializar variables para casillas de texto
tenombre= (TextInputLayout) findViewById(R.id.nombrely);
tetelefono= (TextInputLayout)  findViewById(R.id.telefonoly);
tedireccion= (TextInputLayout)  findViewById(R.id.direccionly);
//Inicializar variables para botones
bguardar= (Button) findViewById(R.id.guardar);
batras=(Button) findViewById(R.id.atras);
fotodeperfil = (ImageView) findViewById(R.id.foto);
```

Código 2.73 Perfil_activity/Distribuidor_perfil-Inicialización de elementos

- **Código para obtener la información del servidor de Firebase**

En el Código 2.74 se muestran los comandos para obtener el ID del usuario que actualmente está en la aplicación (línea 64), luego se accede a la base de datos apuntando al ID obtenido (líneas 65 y 66) y se llama a la función *obtener_info().*

```
63      //obtener el ID del cliente de la base de datos
64      ID_usuario=FirebaseAuth.getInstance().getCurrentUser().getUid();
65      mclientedatabase=FirebaseDatabase.getInstance().getReference()
66              .child("Usuarios").child("Clientes").child(ID_usuario);
67      obtener_info();
```

Código 2.74 Perfil_activity/Distribuidor_perfil-Obtener ID cliente

Con el Código 2.75 se accede a la base de datos en tiempo real y se obtiene la información del usuario mediante el mapeo y extracción de todos los datos bajo el árbol de Firebase: *Usuarios->(Clientes o Distribuidores)->ID_usuario*. Luego se ubica dicha información en

las casillas de texto, además si existe una foto de perfil se recupera del servidor y se ubica en la casilla correspondiente, como se aprecia en las líneas 100 a 114.

```
93  //Obtener la informacion del usuario
94  private void obtener_info(){
95      mclientedatabase.addValueEventListener(new ValueEventListener() {
96          @Override
97          public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
98              if (dataSnapshot.exists()&&dataSnapshot.getChildrenCount()>0){
99                  Map<String, Object> map=(Map<String, Object>) dataSnapshot.getValue();
100                 if (map.get("nombre")!=null){
101                     nombre_aux=map.get("nombre").toString();
102                     tenombre.getEditText().setText(nombre_aux);
103                 }
104                 if (map.get("telefono")!=null){
105                     telefono_aux=map.get("telefono").toString();
106                     tetelefono.getEditText().setText(telefono_aux);
107                 }
108                 if (map.get("direccion")!=null){
109                     direccion_aux=map.get("direccion").toString();
110                     tedireccion.getEditText().setText(direccion_aux);
111                 }
112                 if(map.get("fotoUrl")!=null){
113                     foto_aux = map.get("fotoUrl").toString();
114                     Glide.with(getApplication()).load(foto_aux).into(fotodeperfil);
```

Código 2.75 Perfil_activity/Distribuidor_perfil-Obtener información del cliente desde Firebase

- **Código para el botón *Atrás***

Al presionar el botón *Atrás,* se termina la actividad y se regresa a la actividad principal del usuario, como se muestra en el Código 2.76.

```
//al presionar el boton atras
batras.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        finish();
    }
});
```

Código 2.76 Perfil_activity/Distribuidor_perfil-Botón Atrás

- **Código para el botón *Guardar***

Al presionar el botón *Guardar,* la información que se encuentre ingresada en las casillas de texto o en la casilla de imagen se subirá a la base de datos de Firebase. Para guardar la información se usa la función *guardar_info(),* que se aprecia en el Código 2.77. Esta función obtiene la información de las casillas de texto y la mapea de acuerdo con la categoría (línea 128), siendo nombre, teléfono o dirección y actualiza la base de datos del usuario mediante la línea 132.

```
122     // Guardar la información en Firebase
123     private void guardar_info(){
124         nombre_aux=tenombre.getEditText().getText().toString();
125         telefono_aux=tetelefono.getEditText().getText().toString();
126         direccion_aux=tedireccion.getEditText().getText().toString();
127
128         Map info_usuario= new HashMap();
129         info_usuario.put( k: "nombre",nombre_aux);
130         info_usuario.put( k: "telefono",telefono_aux);
131         info_usuario.put( k: "direccion",direccion_aux);
132         mclientedatabase.updateChildren(info_usuario);
```

Código 2.77 Perfil_activity-Guardar información del cliente en Firebase

2.7.11 AYUDA.JAVA

Esta actividad tiene la función de ser informativa, brinda respuestas a inquietudes que tengan los usuarios en relación con la aplicación y ofrece información del administrador de la aplicación en caso de necesitar contactarlo.

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para almacenar la respuesta a una pregunta**

En el Código 2.78 se muestran las respuestas previamente definidas para preguntas frecuentes. Estas respuestas son almacenadas en variables del tipo *string* y se ubican en las casillas de texto correspondientes para que el usuario pueda acceder a esta información.

```
respues0=(TextView) findViewById(R.id.Respuesta0);
String r0= "La busqueda de un distribuidor de gas se realiza de forma circular, tomando las " +
        "coordenadas del cliente como centro de la circunferencia y aumentando el radio hasta" +
        "encontrar un distribuidor disponible. Si no encuentra un distribuidor, probablemente " +
        "no existen distribuidores disponibles en su zona o los distribuidores existentes " +
        "se encuentran atendiendo otro pedido en ese momento.";
respues0.setText(Html.fromHtml(r0));
```

Código 2.78 Ayuda-Respuesta a pregunta frecuente

- **Código para mostrar las respuestas a las preguntas frecuentes**

Si el usuario presiona una de las preguntas que se encuentran en la sección de “Preguntas Frecuentes” se despliega una casilla de texto con la respuesta a dicha interrogante, como se observa en Código 2.79.

```
TextView preg3=(TextView) findViewById(R.id.Preg3);
preg3.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        respues3.setVisibility(View.VISIBLE);
    }
});
```

Código 2.79 Ayuda-Mostrar respuestas a preguntas frecuentes

2.7.12 HISTORIAL_PEDIDOS.JAVA

Esta actividad es la encargada de presentar al usuario todos los pedidos en los que ha estado involucrado. Para cumplir con el objetivo se crean las actividades: *Objeto_Historial, Soporte_Historial* y *Adaptador_Historial¸* para el soporte de las tarjetas y del *recyclerView* usado en esta actividad, las mismas que se encuentran dentro del paquete *HistRecView*.

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para inicializar las variables**

En el Código 2.80, se inicializan los elementos para el funcionamiento del *ScrollView*, se incluye el soporte para las tarjetas de cada pedido (línea 51). Además se obtiene el ID del usuario actual y se llama a la función *obtener_Hist_usuario(),* para recuperar los pedidos en los que estuvo involucrado el usuario (línea 56).

```
45          //Inicialización de elementos  para el scroll view
46          historialrecview=(RecyclerView) findViewById(R.id.hist_Rev_view);
47          historialrecview.setNestedScrollingEnabled(false);//el scroll sea fluido
48          historialrecview.setHasFixedSize(true);//
49          histmanag=new LinearLayoutManager( context: Historial_pedidos.this);
50          historialrecview.setLayoutManager(histmanag);
51          histadapter=new Adaptador_Historial(getDataSetHistory(), context: Historial_pedidos.this);
52          historialrecview.setAdapter(histadapter);
53
54          DistriOcliente=getIntent().getExtras().getString( key: "DistriOcliente");
55          ID_usuario= FirebaseAuth.getInstance().getCurrentUser().getUid();
56          obtener_Hist_usuaroi();
57          //al presionar el boton atras
58          Button batras = (Button) findViewById(R.id.atras);
59          batras.setOnClickListener(new View.OnClickListener() {
60              @Override
61              public void onClick(View view) {
62                  finish();
```

Código 2.80 Historial_pedido-Inicializar elementos de la actividad

- **Código para obtener los pedidos del usuario**

Con la línea 68 del Código 2.81, se accede a la base de datos de Firebase, al nodo *Historial,* y se compara el ID del usuario actual con los IDs de los usuarios involucrados en cada uno de los pedidos registrados en la aplicación. Si existe una coincidencia, se recupera el ID del pedido y se llama a la función de la línea 75*,* para obtener la información esencial del pedido.

```
66      //obtener el historial de todos los pedidos del usuario
67          private void obtener_Hist_usuaroi() {
68              DatabaseReference usuarioHistDatabase= FirebaseDatabase.getInstance().
69                      getReference().child("Usuarios").child(DistriOcliente).child(ID_usuario).child("Historial");
70              usuarioHistDatabase.addListenerForSingleValueEvent(new ValueEventListener() {
71                  @Override
72                  public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
73                      if(dataSnapshot.exists()){
74                          for (DataSnapshot historial : dataSnapshot.getChildren()){
75                              buscar_fecha_pedid(historial.getKey());
```

Código 2.81 Historial_pedido-Obtener pedidos del usuario

- **Código para obtener la fecha de los pedidos**

Con las líneas del Código 2.82 se ingresa a la base de datos de Firebase, al nodo *Historial,* y se seleccionan todos los pedidos antes recuperados, dentro de los cuales se extraerá el valor de la variable *Fecha* (línea 94), para mostrar al usuario. Se coloca la información de cada pedido en una tarjeta y se notifica al adaptador que el estado de una tarjeta ha cambiado.

```
84  //Busco la información del pedido
85      private void buscar_fecha_pedid(String Ident_pedido) {
86          DatabaseReference HistorialDatabase= FirebaseDatabase.getInstance().
87                  getReference().child("Historial").child(Ident_pedido);
88          HistorialDatabase.addListenerForSingleValueEvent(new ValueEventListener() {
89              @Override
90              public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
91                  if(dataSnapshot.exists()){
92                      String ID_pedido=dataSnapshot.getKey();
93                      for (DataSnapshot child :dataSnapshot.getChildren()){
94                          if(child.getKey().equals("Fecha")){
95                              Fecha=child.getValue().toString();
96                          }
97                      }
98                      Objeto_Historial objt= new Objeto_Historial(ID_pedido,Fecha);
99                      resultado_historial.add(objt);
100                     histadapter.notifyDataSetChanged();
```

Código 2.82 Historial_pedidos-Obtener fecha del pedido

2.7.13 OBJETO_HISTORIAL.JAVA

Esta actividad contiene toda la información que se encuentra en cada una de las tarjetas del *RecyclerView*; dicha información involucra el ID y la fecha del pedido. Se inicializan varias funciones, con las cuales es posible obtener y devolver cualquier ID o fecha de las tarjetas mostradas en la actividad *Historial_pedido,* tal como se observa en el Código 2.83.

```
3   public class Objeto_Historial {
4       private String ID_pedido;
5       private String fecha;
6
7       //Se ingresa los parametros
8   @   public Objeto_Historial(String ID_pedido,String fecha){
9           this.ID_pedido= ID_pedido;
10          this.fecha=fecha;
11      }
12      //para obtener el ID del objeto
13      public String getID_pedido(){
14          return ID_pedido;}
15      public void setID_pedido(String ID_pedido){
16          this.ID_pedido=ID_pedido;}
17      //para obtener la fecha del pedido
18      public String getfecha(){
19          return fecha; }
20      public void setfecha(String fecha){
21          this.fecha=fecha; }
```

Código 2.83 Objeto_HIstorial-Parámetros de las tarjetas del historial

2.7.14 ADAPTADOR_HISTORIAL.JAVA

En el Código 2.84 se observa el funcionamiento de esta actividad, la cual crea las tarjetas en las que se presenta el ID del pedido y la fecha; estas tarjetas son insertadas en una lista para ser ubicadas en el *RecyclerView* (línea 26) y mostrar al usuario la información acerca de los pedidos.

```
1  package com.example.gasapp.HistRecView;
2  import ...
12
13 public class Adaptador_Historial  extends RecyclerView.Adapter<Soporte_Historial>{
14     private List <Objeto_Historial> Lista_items;
15     private Context context;
16     public Adaptador_Historial(List <Objeto_Historial> Lista_items, Context context){
17         this.Lista_items=Lista_items;
18         this.context=context;
19     }
20     // creo la vista de las tarjetas para mostrar al usuario
21     @NonNull
22     @Override
23     public Soporte_Historial onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
24         @SuppressLint("InflateParams") View Layoutv= LayoutInflater.from(parent.getContext()).
25                 inflate(R.layout.item_historial,  root: null,  attachToRoot: false);
26         RecyclerView.LayoutParams laypar=new RecyclerView.LayoutParams(ViewGroup.
27                 LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.WRAP_CONTENT);
28         Layoutv.setLayoutParams(laypar);
29         return new Soporte_Historial(Layoutv);
30     }
31     //Ubica en las casillas de texto la información del viaje
32     @SuppressLint("SetTextI18n")
33     @Override
34     public void onBindViewHolder(@NonNull Soporte_Historial holder, int position) {
35         holder.fecha.setText("Fecha: "+Lista_items.get(position).getfecha());
36         holder.ID_pedido.setText("Código pedido: "+Lista_items.get(position).getID_pedido());
37     }
38     @Override
39     public int getItemCount() {
40         return Lista_items.size();
```

Código 2.84 Adaptador_Historial

2.7.15 SOPORTE_HISTORIAL.JAVA

Esta actividad brinda el soporte a la aplicación para ejecutar las tareas programadas sobre las tarjetas del *RecyclerView* de la actividad *Historial_Pedido*. Las líneas del Código 2.85 muestran el establecimiento de la acción a ser realizada cuando se presiona una tarjeta, la cual es desplazarse a la actividad *Historial_Individual,* y pasar como elemento el ID del pedido (líneas 31 a 35).

```
1  package com.example.gasapp.HistRecView;
2
3  import ...
14
15 public class Soporte_Historial extends RecyclerView.ViewHolder implements View.OnClickListener{
16
17     public TextView ID_pedido;
18     public TextView fecha;
19
20     //Para realizar acciones al presionar un card del historial
21     public Soporte_Historial(@NonNull View itemView) {
22         super(itemView);
23         itemView.setOnClickListener(this);
24         //Inicializa variables para mostrar los pedidos realizados
25         ID_pedido=(TextView) itemView.findViewById(R.id.ID_pedidol);
26         fecha=(TextView) itemView.findViewById(R.id.Fechaly);
27     }
28     //para saber cual tarjeta fue presionada
29     @Override
30     public void onClick(View view) {
31         Intent intent = new Intent(view.getContext(), Historial_individual.class);
32         Bundle b= new Bundle();
33         b.putString("ID_pedido",ID_pedido.getText().toString());
34         intent.putExtras(b);
35         view.getContext().startActivity(intent);
```

Código 2.85 Soporte_Historial-Acciones al presionar una tarjeta del RecyclerView

2.7.16 HISTORIAL_INDIVIDUAL.JAVA

Esta actividad se encuentra diseñada para mostrar al usuario toda la información referente a un pedido, incluyendo: distancia, fecha, cilindros, precio del pedido y las coordenadas de usuario y *Distribuidor* para ubicarlas en el mapa que será cargado en la interfaz; adicionalmente se importan los datos del otro usuario que estuvo involucrado en el pedido.

A continuación, se analizan fragmentos de código para el funcionamiento de esta actividad.

- **Código para inicializar las variables**

En el Código 2.86 se inicializan las casillas de texto para mostrar la información del pedido, además se carga el soporte para los mapas (línea 85 y 86) y se importa el ID del pedido de la actividad anterior (línea 88). Además se llama a la función *obtener_info_pedido(),* para obtener los datos y mostrarlos al usuario.

```
85      mmapFragment=(SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map);
86      mmapFragment.getMapAsync( onMapReadyCallback: this);
87      //Obtener el ID del pedido
88      ID_pedido=getIntent().getExtras().getString( key: "ID_pedido").substring(15);
89      //Inicialización para trazar la ruta
90      polylines = new ArrayList<>();
91      //Inicializar las variables para la información, obtener los objetos del XML
92      codigopedido=(TextView) findViewById(R.id.ID_pedidol);
93      distapedido=(TextView) findViewById(R.id.Distancia_pedi);
94      fechapedido=(TextView) findViewById(R.id.Fecha_pedi);
95      nombreusua=(TextView) findViewById(R.id.Nombre_dis);
96      telefonousua=(TextView) findViewById(R.id.Telefono_dis);
97      preciopedido=(TextView) findViewById(R.id.Precio_ped);
98      cilindrospedido=(TextView) findViewById(R.id.Numero_cil);
99      tiempopedido=(TextView) findViewById(R.id.Tiempo_pedi);
100     fotousua=(ImageView) findViewById(R.id.foto_dist);
101     //al presionar el boton atras
102     Button batras = (Button) findViewById(R.id.atras);
103     batras.setOnClickListener(new View.OnClickListener() {
104         @Override
105         public void onClick(View view) {
106             finish();}
107     });
108     //obtener el usuario actual de la aplicación y el historial del pedido
109     ID_usuar_actual= FirebaseAuth.getInstance().getCurrentUser().getUid();
110     info_dist_pedi= FirebaseDatabase.getInstance().getReference().child("Historial").child(ID_pedido);
111     codigopedido.setText("Código del pedido: "+ ID_pedido);
112     obtener info pedido();
```

Código 2.86 Historial_individual-Inicialización de variables

- **Código para obtener la información del pedido**

La función *obtener_info_pedido()* se aprecia en el Código 2.87 y discrimina si el usuario actual es *cliente* o *Distribuidor* (línea 121 y 129). Si es *cliente*, va a obtener la información del *Distribuidor* que atendió su pedido, por otro lado si el usuario es *Distribuidor*, va a obtener la información del *cliente* que realizó el pedido.

```
113  //Obtener información acerca del pedido
114      private void obtener_info_pedido() {
115          info_dist_pedi.addListenerForSingleValueEvent(new ValueEventListener() {
116              @Override
117              public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
118                  if(dataSnapshot.exists()){
119                      //Compara el ID del pedido con los ID del cliente y ID del distribuidor
120                      for (DataSnapshot child:dataSnapshot.getChildren()){
121                          if(child.getKey().equals("Cliente")){
122                              ID_cliente=child.getValue().toString(); //obtiene ID del cliente que hizo el pedido
123                              //Compara si es cliente o distribuidor con el ID del usuario actual
124                              if(!ID_cliente.equals(ID_usuar_actual)){
125                                  usuario_actual="Distribuidores";
126                                  obtener_info_usuario( usu_Aux_distOcli: "Clientes",ID_cliente);
127                              }
128                          }
129                          if(child.getKey().equals("Distribuidor")){
130                              ID_distribuidor=child.getValue().toString();
131                              if(!ID_distribuidor.equals(ID_usuar_actual)){
132                                  usuario_actual="Clientes";
133                                  obtener_info_usuario( usu_Aux_distOcli: "Distribuidores",ID_distribuidor);
```

Código 2.87 Historial_Individual-Obtener la información del pedido (1)

Luego se busca en la base de datos de Firebase, en el nodo *Historial,* el ID del pedido y se extrae toda la información requerida, para ubicarla en las casillas de texto y pueda ser visualizada por el usuario, como se observa en el Código 2.88.

```
136  //Obtiene la fecha
137  if(child.getKey().equals("Fecha")){
138      fechapedido.setText(Html.fromHtml( source: "<b>Fecha: </b>"+child.getValue().toString()));
139  }
140  //Obtener el número de cilindros
141  if (child.getKey().equals("Cilindros")){
142      cilindrospedido.setText(Html.fromHtml( source: "<b>Cilindros pedidos: </b>"+child.getValue().toString()));
143  }
144  //obtener el precio
145  if (child.getKey().equals("Precio")){
146      DecimalFormat formato= new DecimalFormat( pattern: "$0.00");
147      Double precio= Double.parseDouble(child.getValue().toString());
148      preciopedido.setText(Html.fromHtml( source: "<b>Precio del pedido: </b>"+formato.format(precio)));
149  }
150  // Obtener el tiempo que duro el pedido
151  if (child.getKey().equals("Tiempo")){
152      String tiempo= child.getValue().toString();
153      tiempopedido.setText(Html.fromHtml( source: "<b>Tiempo del pedido: </b>"+tiempo));
154  }
```

Código 2.88 Historial_Individual-Obtener la información del pedido (2)

Finalmente, en el Código 2.89 se obtienen las coordenadas de los puntos involucrados en el pedido (líneas 158 y 161) y se llama a la función *obtener_ruta(),* para obtener la ruta recorrida por el distribuidor.

```
156      //Obtiene las coordenadas de los viajes
157      if(child.getKey().equals("Lugar")){
158          LatLng_origen=new LatLng(Double.valueOf(child.child("Origen").
159                  child("lat").getValue().toString()),Double.valueOf(child.
160                  child("Origen").child("lon").getValue().toString()));
161          LatLng_destino=new LatLng(Double.valueOf(child.child("Destino").
162                  child("lat").getValue().toString()),Double.valueOf(child.
163                  child("Destino").child("lon").getValue().toString()));
164          if (!LatLng_destino.equals(new LatLng( v: 0,  v1: 0))){
165              mmap.addMarker(new MarkerOptions().
166                      position(LatLng_origen));
167              mmap.addMarker(new MarkerOptions().
168                      position(LatLng_destino));
169          }
170          obtener_ruta(LatLng_destino, LatLng_origen);
```

Código 2.89 Historial_Individual-Obtener la información del pedido (3)

- **Código para obtener los elementos del mapa**

En esta función (Código 2.90) se realiza todo lo correspondiente a la sección del mapa, en la cual se ubican los marcadores de las posiciones iniciales del *Distribuidor* y *cliente*, y se grafica la ruta que el *Distribuidor* siguió junto con la distancia recorrida para la entrega del pedido.

```
//Obtener distancia entre puntos
private void obtener_ruta(LatLng LatLng_cliente,LatLng ubicacion){
    //obtener las rutas desde el distribuidor al cliente
    urla=getUrl(ubicacion, LatLng_cliente, directionMode: "driving");
    TakeRequestDirection takeRequestDirection=new TakeRequestDirection();
    String aux= String.valueOf(takeRequestDirection.execute(urla));
    //ubicacion cliente
    Location location1 = new Location( provider: "");
    location1.setLatitude(LatLng_cliente.latitude);
    location1.setLongitude(LatLng_cliente.longitude);
    //ubicacion distribuidor
    Location location2 = new Location( provider: "");
    location2.setLatitude(ubicacion.latitude);
    location2.setLongitude(ubicacion.longitude);
    //la distancia entre el cliente y el distribuidor
    float distancia = location1.distanceTo(location2);
    DecimalFormat formato= new DecimalFormat( pattern: "0.00");
    distapedido.setText(Html.fromHtml( source: "<b>Distancia: </b> "+formato.format(distancia)+" metros."));
    //Zoom en los puntos
    LatLngBounds.Builder builder= new LatLngBounds.Builder();
    builder.include(LatLng_cliente); //ubicación cliente
    builder.include(ubicacion); //ubicación distribuidor
    LatLngBounds bounds= builder.build();
    //obtener las dimensiones de la pantalla
    int width =getResources().getDisplayMetrics().widthPixels;
    int padding= (int) (width*0.2);
    CameraUpdate cameraUpdate= CameraUpdateFactory.newLatLngBounds(bounds, padding);
    mmap.animateCamera(cameraUpdate);
}
```

Código 2.90 Historial_Individual-Obtener elementos para el mapa

- **Código para obtener la información del usuario**

En el Código 2.91 se accede a la base de datos de Firebase en busca el ID del usuario (línea 183) recuperado anteriormente, del cual se obtiene el nombre, teléfono y foto de perfil para mostrar en la interfaz actual.

```
181 //Obtengo la información del usuario
182     private void obtener_info_usuario(String usu_Aux_distOcli, String ID_usuar_aux) {
183         DatabaseReference DBusuaraux=FirebaseDatabase.getInstance().getReference().child("Usuarios").
184                 child(usu_Aux_distOcli).child(ID_usuar_aux);
185         DBusuaraux.addListenerForSingleValueEvent(new ValueEventListener() {
186             @Override
187             public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
188                 if(dataSnapshot.exists()){
189                     Map<String, Object> map=(Map<String, Object>) dataSnapshot.getValue();
190                     if (map.get("nombre")!=null){
191                         nombreusua.setText(Html.fromHtml( source: "<b>Usuario: </b>"+map.get("nombre").toString()));
192                     }
193                     if (map.get("telefono")!=null){
194                         telefonousua.setText(Html.fromHtml( source: "<b>Teléfono: </b>"+map.get("telefono").toString()));
195                     }
196                     if (map.get("fotoUrl")!=null){
197                         Glide.with(getApplication()).load(map.get("fotoUrl").toString()).into(fotousua);
```

Código 2.91 Historial_individual-Obtener la información del usuario

3. RESULTADOS Y DISCUSIÓN

Este capítulo presenta los resultados de las pruebas de funcionamiento realizadas con el objetivo de validar la aplicación desarrollada, por lo cual se instala la *app* en dos teléfonos celulares haciendo una emulación de un ambiente en el que un teléfono actuará como *cliente* mientras el otro actuará como *Distribuidor*. Las pruebas se ejecutaron en tres secciones: funcionamiento de los módulos de registro y autenticación, funcionamiento del sistema de pedido de gas, y funcionamiento de módulos adicionales.

3.1 EQUIPOS EMPLEADOS

Para lograr comprobar el funcionamiento del aplicativo móvil, se instala la APK de la aplicación en dos *Smartphones*. La *app* al estar desarrollada para el soporte de la versión Android 4.0 (*IceCreamSandwich*) en adelante, es compatible con aproximadamente el 100% de dispositivos Android [2]. Los dispositivos móviles utilizados para las pruebas son un Xiaomi Redmi 5A y un Sony Experia Z5, además se pueden utilizar emuladores de Android para PC .

3.1.1 SMARTPHONE XIAOMI REDMI 5A

Se selecciona este móvil debido a que cuenta con sistema operativo Android, por lo que es posible la implementación de la aplicación desarrollada en el mismo. El celular dispone de una pantalla de 5 pulgadas, 720 x 1280 pixels, procesador Snapdragon 425 de cuatro núcleos, 2GB de memoria RAM, soporta redes 2G, 3G, 4G y conectividad a Internet a través de WiFi o de datos móviles. En la Figura 3.1, se aprecia el modelo del equipo que es MCG3B y actualmente cuenta con el sistema operativo 8.1.0-Oreo [22].

Figura 3.1 Móvil Xiaomi Redmi 5A [22]

3.1.2 SMARTPHONE SONY XPERIA Z5

Al igual que el móvil anterior, este celular tiene como base un sistema operativo Android; cuenta con una pantalla de 5,2 pulgadas, resolución de 1920 x 1080 pixels, un procesador Qualcomm Snapdragon 801 V2.1 de ocho núcleos, memoria RAM de 3 GB y conectividad a Internet a través de datos móviles o WiFi 802.11 dual-band. En la Figura 3.2, se muestra el modelo del equipo, el cual actualmente se encuentra con el sistema operativo 7.1.1 Nougat [23].

Figura 3.2 Móvil Sony Xperia Z5 [23]

3.2 PRUEBAS DE FUNCIONALIDAD

3.2.1 PRUEBAS DE FUNCIONALIDAD DE LOS MÓDULOS DE REGISTRO E INGRESO

En este módulo, la funcionalidad es validada a través de dos interfaces, la primera permite ingresar a la aplicación con un usuario previamente registrado y la segunda, se usa para el registro de un nuevo usuario.

3.2.1.1 Prueba de funcionamiento del módulo de ingreso

Al seleccionar el rol de *Cliente,* para proceder a hacer uso de la aplicación se ingresan con su correo y contraseña, como se observa en la Figura 3.3. Se valida en la base de datos de Firebase si existe el usuario; de existir se da paso a la pantalla principal del *cliente*, en caso de no estar registrado o ingresar mal las credenciales, aparecerá un mensaje de error, como se aprecia en la Figura 3.4.

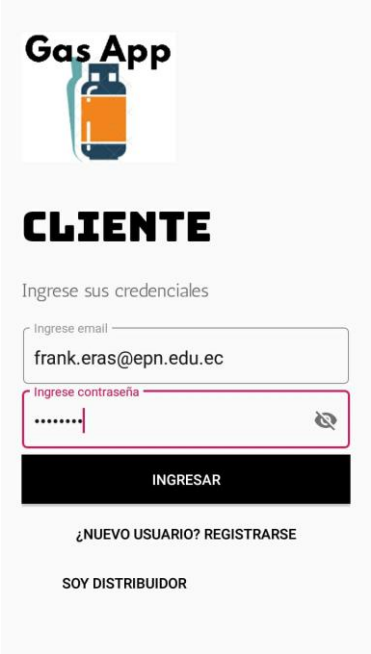

Figura 3.3 Ingreso de credenciales del cliente

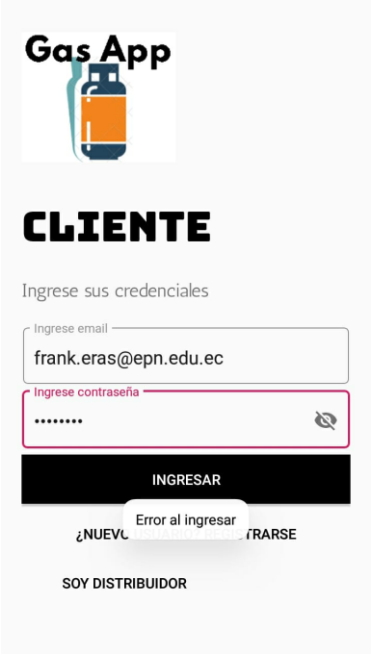

Figura 3.4 Mensaje de error de credenciales

Por otro lado si se elige el rol de *Distribuidor,* al igual que en el rol del *cliente*, se valida en la base de datos de Firebase la existencia del usuario. Si el *Distribuidor* es autenticado, automáticamente sus coordenadas se actualizarán en la base de datos de Firebase, en el nodo de *DistribuidorDisponible,* como se muestra en la Figura 3.5.

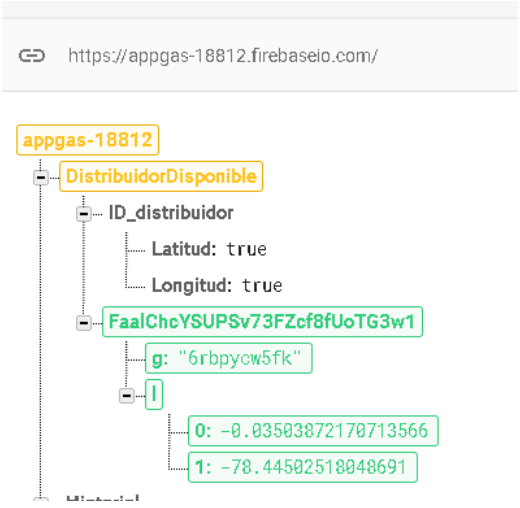

Figura 3.5 Actualización de la base de datos de Firebase al ingresar un distribuidor

3.2.1.2 Prueba de funcionamiento del módulo de registro

Si es la primera vez que un usuario accede a la aplicación, se debe efectuar el proceso de registro para hacer uso de los servicios de la *app*. Como se observa en la Figura 3.6, la

creación de un nuevo usuario se realiza llenando las casillas de texto con la información solicitada y se da clic en el botón *Registrarse* para continuar con el proceso. Si el usuario elige registrarse como *Distribuidor.* Adicionalmente, se pide las placas del vehículo que se usará para la distribución de cilindros de gas.

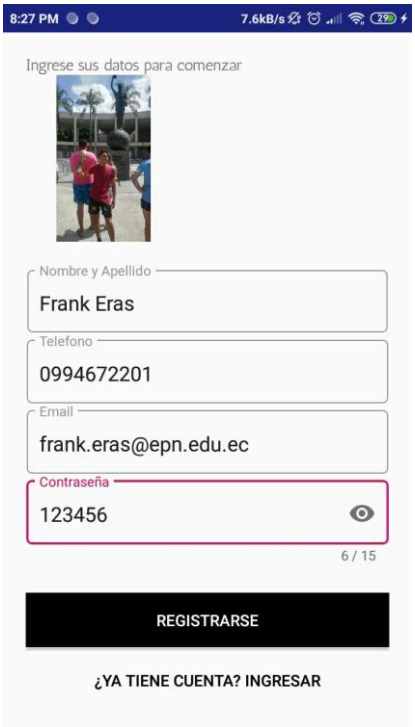

Figura 3.6 Registro de un nuevo cliente

En la base de datos de Firebase automáticamente se refleja la información del nuevo usuario. Primeramente se guarda el correo y contraseña ingresados, en el servicio de autenticación y se le asigna un ID único dentro de la aplicación para ser identificado, como se muestra en la Figura 3.7.

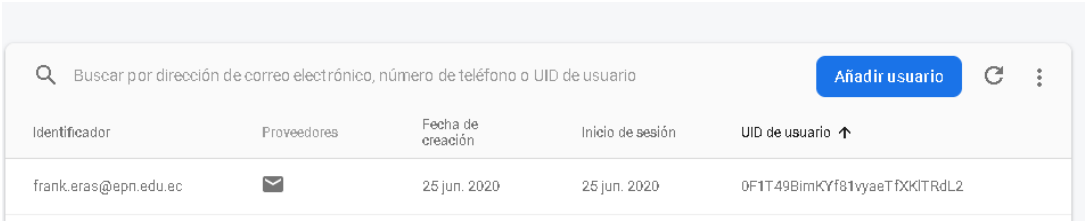

Figura 3.7 Usuarios registrado en el servicio de autenticación de Firebase

Luego, en la base de datos de Firebase se crea un nuevo usuario con el ID asignado en la fase de autenticación y dependiendo del rol elegido, el usuario se ubicará bajo el nodo *Clientes* o *Distribuidores.* Como se observa en la Figura 3.8, se almacenan los datos ingresados en la aplicación, incluyendo la URL de la foto de perfil, la cual apunta al servicio de *Storage* que ofrece Firebase.

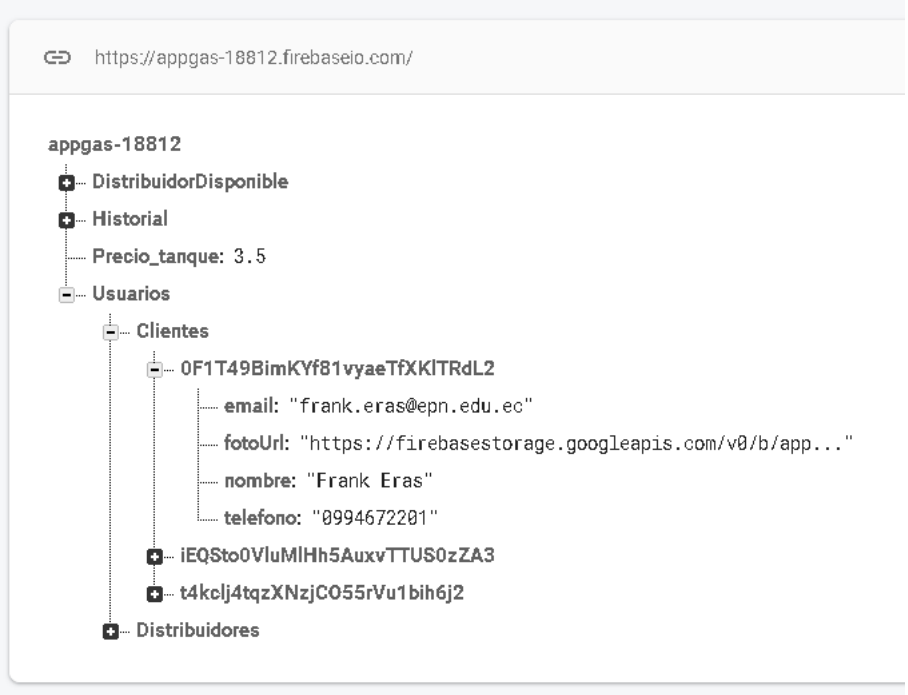

Figura 3.8 Creación de un nuevo usuario en la base de datos de Firebase

En la pestaña de *Storage* de la consola de Firebase se almacena la foto de perfil que el usuario subió a la aplicación. La Figura 3.9, muestra que la imagen es guardada con el ID asignado al usuario y con la fecha de modificación.

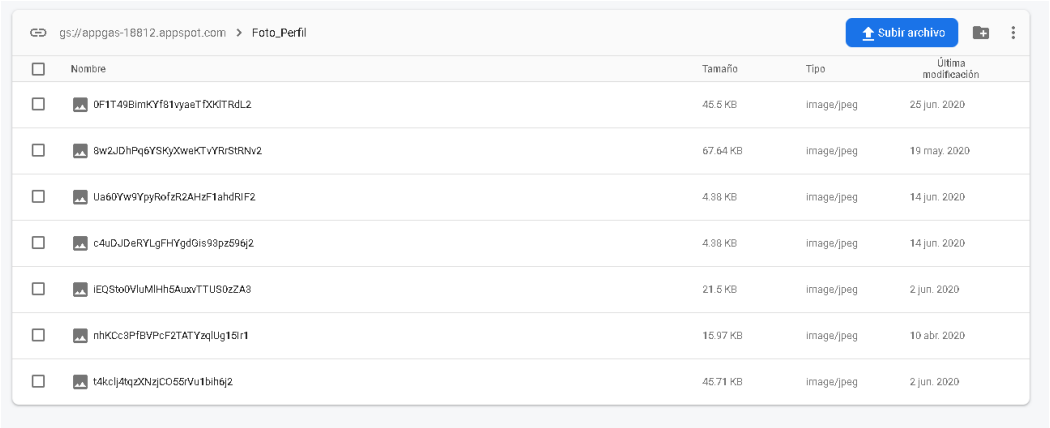

Figura 3.9 Almacenamiento de la foto de perfil de un usuario en Storage de Firebase

3.2.2 PRUEBAS DE FUNCIONAMIENTO DEL SISTEMA DE PEDIDO DE GAS

La funcionalidad de este módulo será validada mediante dos interfaces y la capacidad de un *cliente* de realizar un pedido de gas y que éste sea atendido por un *Distribuidor*. El sistema entra en funcionamiento cuando el *cliente* presiona el botón *Realizar Pedido,* en la interfaz del *cliente* y termina cuando el *Distribuidor* presiona el botón *Entregar Pedido,* en la interfaz del *Distribuidor*.

3.2.2.1 Realizar un pedido de gas

El *cliente* para realizar un pedido de gas puede hacerlo desde la interfaz principal de la aplicación, en la cual debe presionar una ubicación en el mapa en donde desea que se haga la entrega del pedido; esta ubicación es transformada a una dirección y se debe especificar el número de cilindros de gas que requiere. Si no se llena alguno de estos campos y se desea pedir gas, la aplicación mostrará un mensaje solicitando toda la información necesaria para el pedido, como se observa en la Figura 3.10.

Figura 3.10 Mensaje de error al no llenar los campos para un pedido de gas

Por otro lado, la interfaz del *Distribuidor* muestra el mensaje "*Esperando Pedido",* y un mapa con la ubicación actual del *Distribuidor*, como se observa en la Figura 3.11.

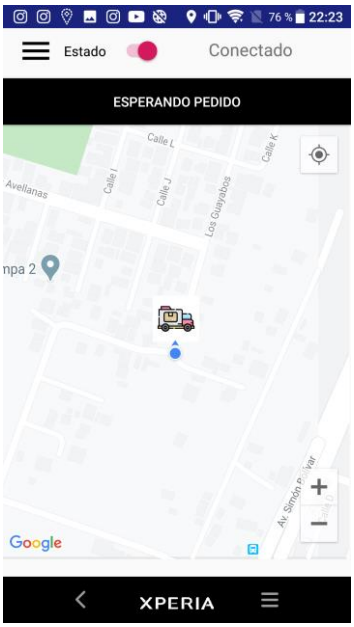

Figura 3.11 Distribuidor a la espera de un pedido

El *Distribuidor* se coloca en la base de datos de Firebase en el nodo de *DistribuidorDisponible*, junto a su ID único y sus coordenadas actuales, a la espera de un pedido, como se muestra en la Figura 3.12.

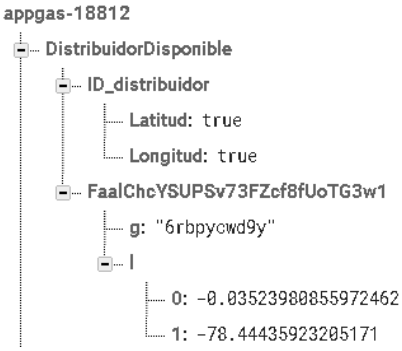

Figura 3.12 Distribuidores Disponibles mostrados en la base de datos de Firebase

Si el *cliente* llenó los campos requeridos para el pedido y presiona el botón *Realizar Pedido*, automáticamente se suben las coordenadas del cliente a la base de datos de Firebase, al nodo *Pedidos,* como se observa en la Figura 3.13.

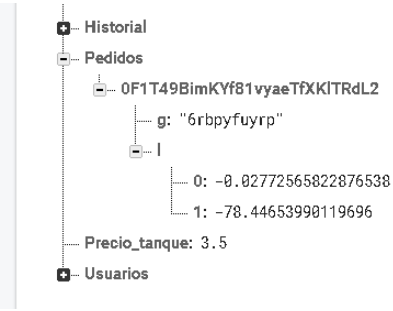

Figura 3.13 Nodo Pedidos en la base de datos de Firebase

La aplicación del lado del *cliente* comienza una búsqueda del *Distribuidor* más cercano y al encontrarlo, automáticamente se actualiza la base de datos de Firebase y el *Distribuidor* pasa de nodo *DistribuidorDisponible* a nodo *DistribuidorTrabajando,* como se observa en la Figura 3.14.

Figura 3.14 Actualización de la base de datos de Firebase cuando un distribuidor recibe un pedido

3.2.2.2 Atender un pedido de gas

Cuando a un *Distribuidor* se le asigna un pedido, se crea un nodo temporal dentro del ID del *Distribuidor*, llamado *Pedido_cliente,* como se puede observar en la Figura 3.15. Este nodo contiene la información relacionada al pedido, incluyendo el ID del cliente que solicitó el GLP, el número de cilindros y el tiempo que le tomará completar el pedido.

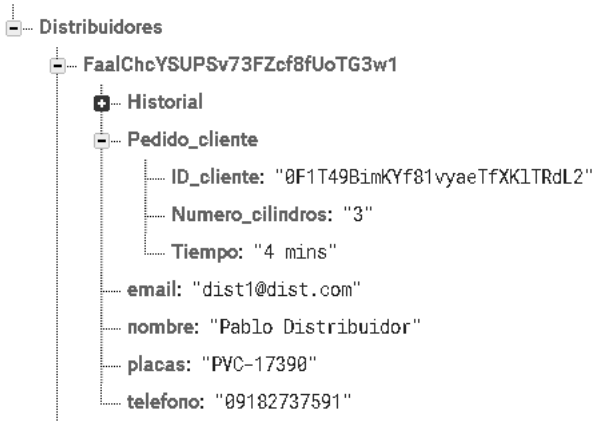

Figura 3.15 Nodo temporal con la información del pedido actual

En la interfaz de la aplicación correspondiente al *Distribuidor* aparece una notificación con el mensaje de un pedido entrante. Como se muestra en la Figura 3.16, se debe presionar el botón *Ver Pedido,* para continuar con el proceso.

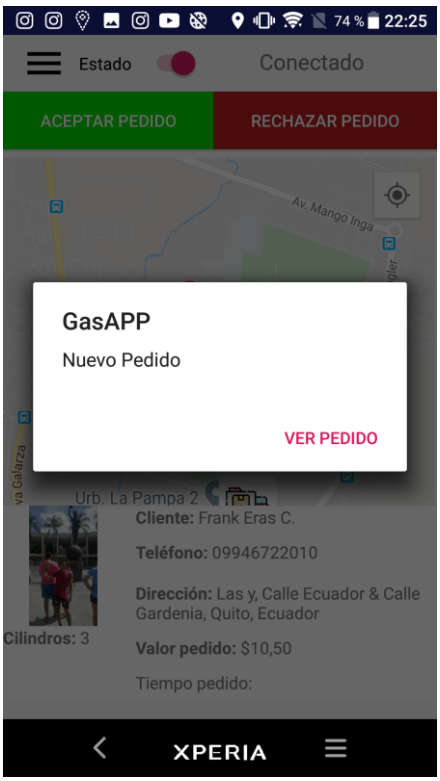

Figura 3.16 Notificación al distribuidor de un nuevo pedido de gas

En la interfaz del *Distribuidor* se despliega la información correspondiente al pedido, incluyendo la ruta en el mapa, el número de cilindros solicitados, el valor del pedido y el tiempo que le tomará llegar al *cliente*. Adicionalmente se muestran los datos del *cliente*: nombre, teléfono, dirección y foto de perfil. En la Figura 3.17, se puede observar la pantalla previa a atender un pedido, el cual puede ser aceptado o rechazado por el *Distribuidor* mediante los botones correspondientes.

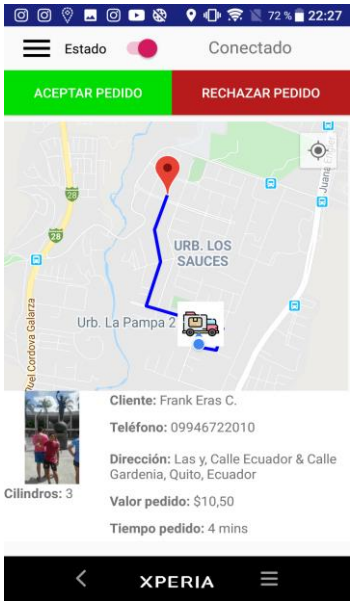

Figura 3.17 Pantalla del distribuidor previo la atención de un pedido de gas

Si el *Distribuidor* acepta el pedido, automáticamente en la interfaz del cliente se muestra la información de la persona que atenderá su pedido, junto al precio total, el tiempo que tomará y la ruta que el *Distribuidor* seguirá. En la Figura 3.18, se observa la pantalla del *cliente* cuando un *Distribuidor* se encuentra atendiendo el pedido y se complementa con un botón para cancelar el pedido en caso de ser necesario.

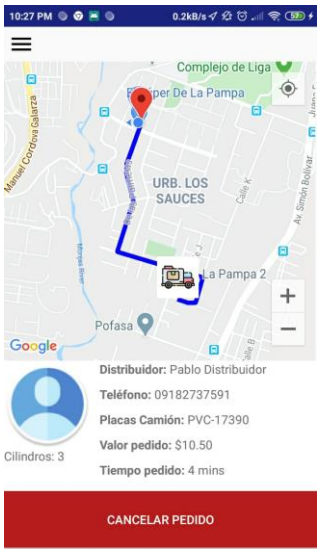

Figura 3.18 Pantalla del cliente cuando un pedido está en curso

La Figura 3.19 y la Figura 3.20, muestran las interfaces del *Distribuidor* y del *cliente* en tiempo real de la ubicación del *Distribuidor* y la ruta que debe seguir, misma que se va actualizando a medida que se mueve el *Distribuidor*.

Figura 3.19 Estado de la ruta del pedido del lado del distribuidor

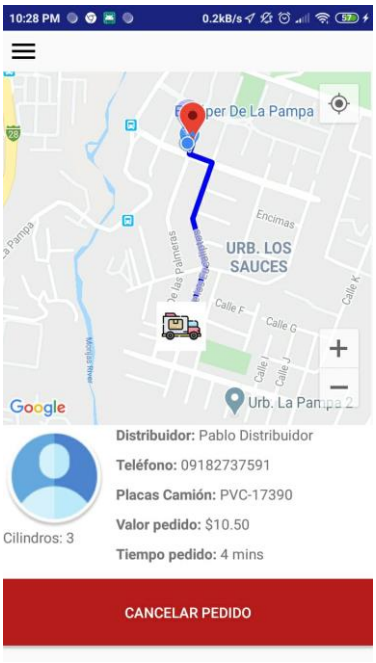

Figura 3.20 Estado de la ruta del pedido del lado del cliente

3.2.2.3 Finalizar un pedido de gas

Si el *Distribuidor* se encuentra a una distancia menor a 50 metros de la ubicación de entrega del pedido, en la interfaz del cliente aparece una notificación anunciando la llegada del gas, como se muestra en la Figura 3.21.

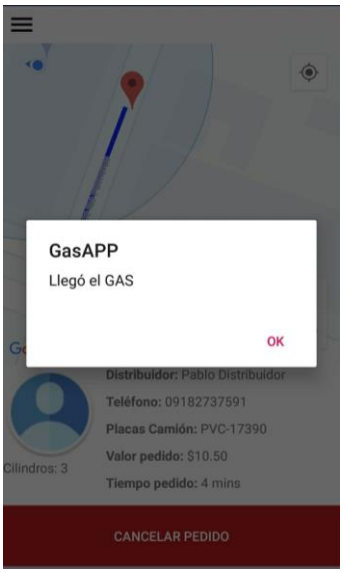

Figura 3.21 Notificación al cliente que llegó el pedido de gas

Por otra parte, en la interfaz del *Distribuidor* se habilita el botón de *Entregar Pedido,* como se muestra en la Figura 3.22. Al presionarse da por terminado el pedido, con lo cual se actualiza la base de datos de Firebase y como se observa en la Figura 3.23 se elimina al *Distribuidor* del nodo *DistribuidorTrabajando* y se añade al nodo *DistribuidorDisponible.*

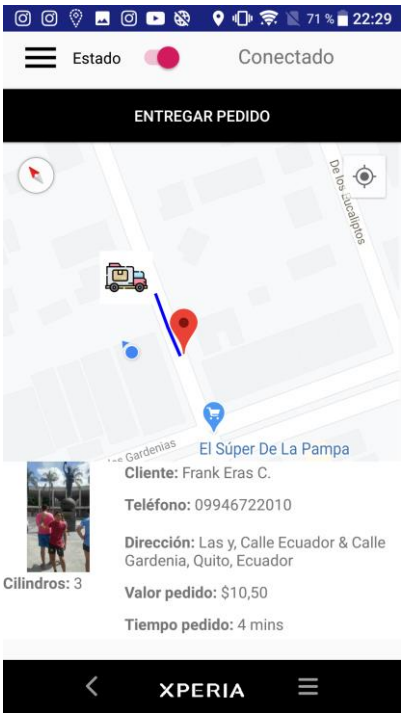

Figura 3.22 Pantalla del distribuidor para finalizar un pedido

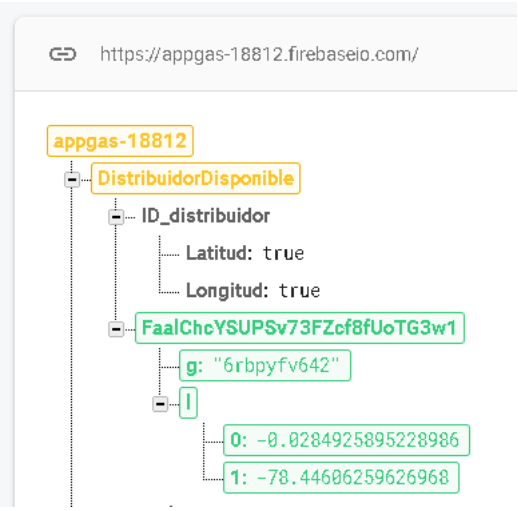

Figura 3.23 Actualización de la base de datos de Firebase al terminar un pedido

Finalmente, el pedido se guarda en la base de datos de Firebase, en el nodo *Historial.* En la Figura 3.24, se aprecia el almacenamiento del ID del pedido junto a la información de éste, para su posterior uso.

Figura 3.24 Almacenamiento de un pedido en la base de datos de Firebase

3.2.3 PRUEBAS DE FUNCIONAMIENTO DE MÓDULOS ADICIONALES

La aplicación cuenta con tres módulos que brindan funciones adicionales a los usuarios, las cuales serán validadas a través de las interfaces correspondientes.

3.2.3.1 Pruebas de funcionamiento del módulo de Configuración

El módulo de configuración permite realizar el cambio de la información del usuario, para el *Distribuidor* es posible cambiar el nombre, teléfono y placas del carro; mientras para el *cliente* se puede cambiar el nombre, teléfono, dirección de referencia y foto de perfil, como se observa en la Figura 3.25.

Figura 3.25 Interfaz de configuración para cambio de información del usuario

Al presionar el botón *Guardar,* la aplicación obtiene la información ingresada y la envía a la base de datos de Firebase, en donde se actualizan los datos previamente almacenados con los nuevos. En la Figura 3.26 se muestra la actualización realizada en tiempo real de la información del cliente.

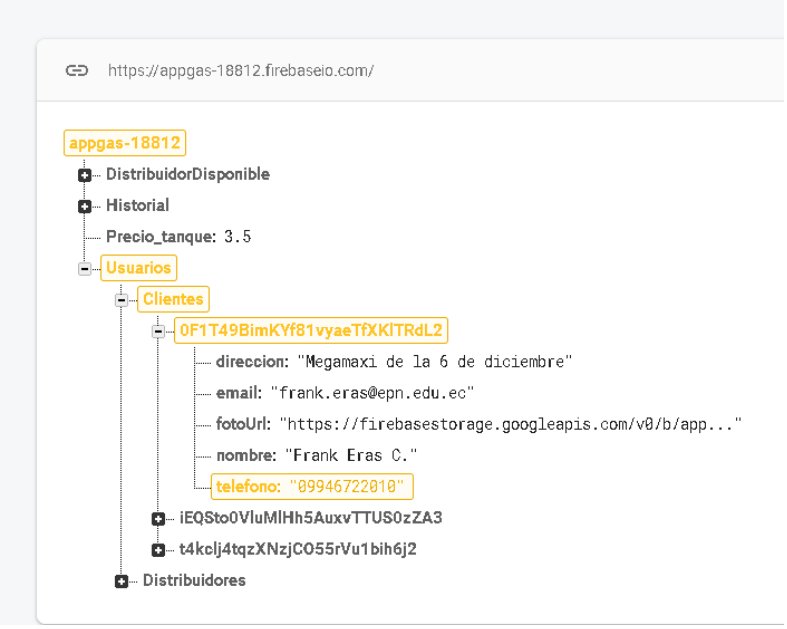

Figura 3.26 Actualización de la base de datos de un usuario

3.2.3.2 Pruebas de funcionamiento del módulo de Ayuda

El módulo *Ayuda,* tiene la función de ser informativo y responder a inquietudes básicas de los usuarios acerca del uso de la aplicación. En la Figura 3.27, se observa la interfaz que cuenta con respuestas referentes a la aplicación, las cuales son desplegadas al presionar una pregunta de las existentes.

Figura 3.27 Información presentada en el módulo Ayuda

3.2.3.3 Pruebas de funcionamiento del módulo de Historial

El módulo *Historial* se valida a través de dos interfaces; la primera interfaz recopila todos los pedidos de gas en los que el usuario estuvo involucrado, mientras la segunda interfaz recopila la información de un pedido en específico. La Figura 3.29, muestra la primera interfaz de historial, la cual ofrece el ID único del pedido y la fecha en la que el pedido de gas se realizó y atendió.

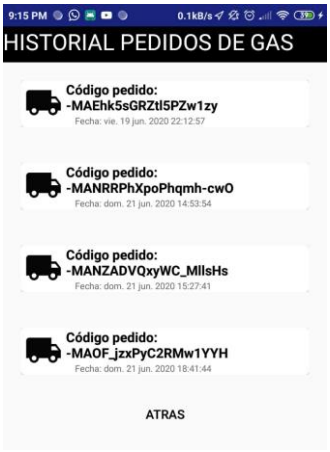

Figura 3.28 Historial de los pedidos de un usuario

Si se desea obtener información adicional de un pedido se presiona sobre el mismo y se desplegará la segunda interfaz del módulo *Historial,* en la cual es posible visualizar todo lo referente a dicho pedido. Como se observa en la Figura 3.29, la información del pedido se obtiene de la base de datos de Firebase mediante el ID único asignado a determinado pedido.

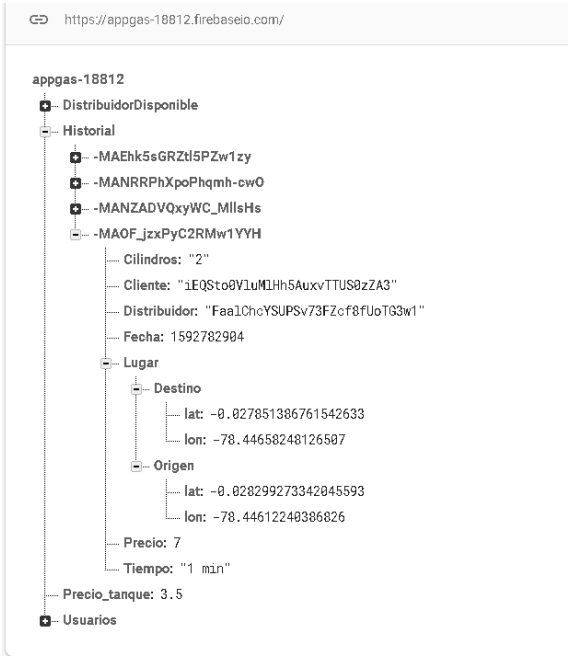

Figura 3.29 Información de un pedido en la base de datos de Firebase

La Figura 3.30 presenta la interfaz del historial individual de un pedido, en la cual se incluye un mapa con la posición de los usuarios involucrados en el pedido y la ruta que se realizó para la entrega del Gas Licuado de Petróleo (GLP). Se muestra el código del pedido, la fecha en que se realizó, la distancia entre el cliente y el distribuidor y el tiempo que tomó el pedido en ser despachado. Finalmente se entrega la información del otro usuario involucrado en el pedido, es decir, si el *cliente* accede al historial del pedido observará la información del *Distribuidor* que lo atendió; mientras si el *Distribuidor* accede al historial del pedido, observará la información del *cliente* al cual atendió.

Figura 3.30 Información de un pedido en el Historial

3.3 RESULTADOS DE LAS PRUEBAS DE FUNCIONALIDAD

Con el fin de evaluar la funcionalidad de la aplicación en su totalidad, se plantean diversos escenarios, los cuales deben cumplir con los criterios de aceptación propuestos para que la *app* se considere exitosa. Adicionalmente, se realiza un análisis de los recursos utilizados en el teléfono móvil durante la ejecución de la aplicación.

3.3.1 RESULTADOS DE LAS PRUEBAS DE LOS MÓDULOS DE REGISTRO E INGRESO

En la Tabla 3.1 se muestran los resultados de las pruebas realizadas en los módulos de autenticación y registro de la aplicación, las cuales superan los criterios de aceptación y se obtienen los resultados deseados.

Tabla 3.1 Resultados de funcionalidad del módulo de registro y autenticación

Escenario	Descripción de la prueba	Criterio de aceptación
Registro de un nuevo usuario.	Un usuario ingresa un correo electrónico, contraseña y datos personales para registrarse en la aplicación.	Se guarda la información en la base de datos de Firebase y avanza a la interfaz principal de la aplicación.
Ingreso de credenciales de un usuario previamente registrado.	Un usuario previamente registrado en la aplicación ingresa correctamente su correo y contraseña.	La aplicación avanza a la interfaz principal del usuario.
Ingreso de credenciales de un usuario no registrado.	Un usuario que no está registrado en la aplicación ingresa un correo y contraseña aleatorios.	Se muestra mensaje de error y se permanece en la interfaz de ingreso.
Ingreso de credenciales del *Distribuidor* en la interfaz de ingreso del *cliente*.	Un *cliente* ingresa su correo y contraseña en la interfaz de ingreso del *Distribuidor*.	Se niega el acceso, se muestra un mensaje de que se encuentra en la interfaz errónea y se mantiene al usuario en dicha interfaz.
Ingreso de credenciales del *cliente* en la interfaz de ingreso del *Distribuidor*.	Un *Distribuidor* ingresa su correo y contraseña en la interfaz de ingreso del *cliente*.	Se niega el acceso, se muestra un mensaje de que se encuentra en la interfaz errónea y se mantiene al usuario en dicha interfaz.

3.3.2 RESULTADOS PRUEBAS DEL SISTEMA DE PEDIDO DE GAS

En la Tabla 3.2 se observan los resultados de las pruebas realizadas en el Sistema de Pedido de gas de la aplicación, las cuales superan los criterios de aceptación y se obtienen los resultados deseados.

Tabla 3.2 Resultados de funcionalidad del Sistema de Pedido de gas

Escenario	Descripción de la prueba	Criterio de aceptación
El *cliente* realiza un pedido y el *Distribuidor* acepta el pedido.	El *cliente* ingresa su ubicación, el número de cilindros y realiza el pedido; el *Distribuidor* recibe una notificación del pedido y lo acepta.	El *cliente* observa la información del *distribuidor* que lo atenderá, la ubicación de este y la ruta que tomará para llegar.
El *cliente* realiza un pedido pero el *Distribuidor* rechaza el pedido.	El *cliente* ingresa su ubicación, el número de cilindros y realiza el pedido; el *Distribuidor* recibe una notificación del pedido y lo rechaza.	El pedido continúa en la base de datos de Firebase y se realiza una nueva búsqueda de otro *Distribuidor*.
Un pedido se encuentra en marcha y el *cliente* cancela el pedido.	Un pedido está en marcha, pero el *cliente* presiona el botón de cancelar pedido.	Se da por finalizado el pedido y no se registra en el historial de pedidos. El *Distribuidor* regresa a la categoría Distribuidor Disponible en Firebase, para receptar próximos pedidos.
Un pedido se encuentra en marcha y el *Distribuidor* cancela el pedido.	Un pedido está en marcha, pero el *Distribuidor* presiona el botón de cancelar pedido.	Se da por finalizado el pedido y no se registra en el historial de pedidos. El *Distribuidor* regresa a la categoría Distribuidor Disponible en Firebase, para receptar próximos pedidos.
El *cliente* realiza un pedido pero no encuentra *Distribuidor* disponible.	El *cliente* ingresa su ubicación, el número de cilindros y realiza el pedido; no hay *Distribuidores* disponibles en la zona.	Si no se encuentra un *Distribuidor* a 20 km a la redonda del cliente se muestra un mensaje que indica que no existen *Distribuidores* disponibles en el área y se cancela el pedido.

3.3.3 RESULTADOS PRUEBAS DE MÓDULOS ADICIONALES

En la Tabla 3.3 se aprecian los resultados de las pruebas de los módulos adicionales de la aplicación, tales como Configuración, Historial y Ayuda. Las pruebas realizadas superan los criterios de aceptación y se obtienen los resultados deseados.

Tabla 3.3 Resultados de funcionalidad de módulos adicionales

Escenario	Descripción de la prueba	Criterio de aceptación
El usuario cambia su información en el módulo de *Configuración.*	Un usuario modifica su nombre, número de teléfono, fotografía y presiona el botón *Guardar*.	Se actualiza la información ingresada en la aplicación en la base de datos de Firebase.
El usuario presiona una pregunta en el módulo de *Ayuda*.	Un usuario ingresa al módulo *Ayuda* y presiona las preguntas existentes.	Se despliega la respuesta a la pregunta presionada.
El usuario ingresa al módulo *Historial.*	Un usuario presiona el botón *Historial* que se encuentra en el menú lateral de la aplicación.	La aplicación recupera de la base de datos de Firebase los pedidos del usuario y los muestra en pantalla.
El usuario presiona un pedido en el módulo *Historial.*	Un usuario que se encuentra en el módulo *Historial* presiona sobre un pedido existente.	La aplicación recupera la información del pedido particular de la base de datos de Firebase y la muestra en una nueva interfaz desplegada en la app.
El usuario presiona el botón *Salir.*	Un usuario presiona el botón *Salir* que se encuentra en el menú lateral de la aplicación.	Se cierra la sesión del usuario actual y se muestra la interfaz de inicio de sesión para ingreso de credenciales.

3.3.4 ANÁLISIS DE LOS RECURSOS CONSUMIDOS POR LA APLICACIÓN

El análisis de los recursos utilizados en el teléfono móvil se realiza mediante *Android Profiler*, una de las herramientas de Android Studio, la cual permite obtener datos en tiempo real acerca del consumo de la memoria, red y batería de una aplicación en el móvil. El análisis se realiza en dos escenarios, en el primero se ingresa a la aplicación y se prueban cada uno de los módulos. Mientras que el segundo escenario se enfoca en el consumo de los recursos de red durante el proceso de un pedido.

En la Figura 3.31 se muestra la gráfica general de los recursos de memoria, red y energía que consume la aplicación en el móvil. Se observa que el uso de memoria no supera los 256 MB, el uso de la red se limita a máximo 2 MB/s y la carga de energía es variable dependiendo del uso que se esté dando a la aplicación. Para tener una mejor comprensión, se procede a analizar cada uno de estos recursos por separado, con el siguiente detalle entre tiempos y actividad realizada:

- Minuto 00:00 .- ingreso a la aplicación.
- Minuto 00:05 .- ingreso al módulo selección de rol.
- Minuto 00:15 .- autenticación de credenciales en el módulo de ingreso.
- Minuto 00:25-01:15 .- navegación en el módulo principal.
- Minuto 01:30 -02:00 .- se realiza un pedido y se obtiene un distribuidor luego de la búsqueda.
- Minuto 02:15 .- se ingresa al módulo de ayuda.
- Minuto 03:00 .- se ingresa al módulo de historial y se carga el historial de un pedido en específico.
- Minuto 04:00 .- se finaliza el pedido realizado.
- Minuto 04:30 .- se ingresa al módulo configuración.
- Minuto 05:00 .- se guardan los datos modificados en el módulo configuración.
-

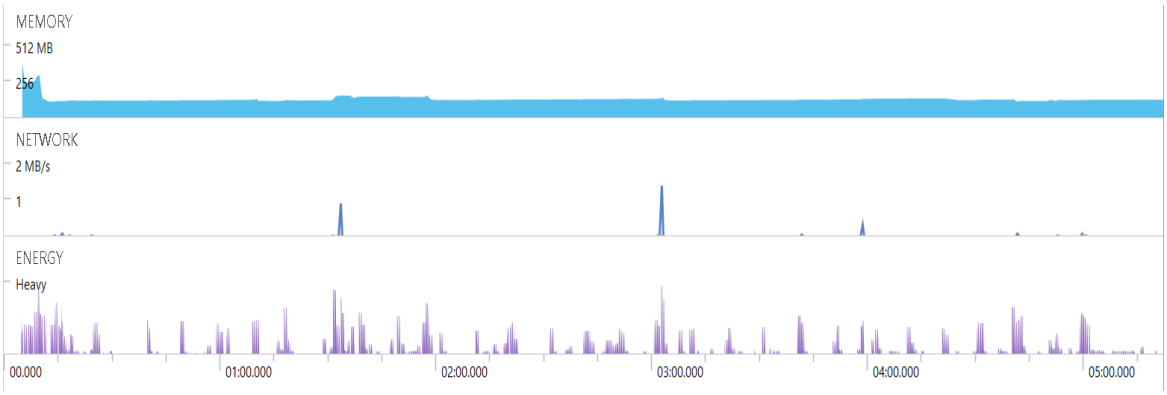

Figura 3.31 Consumo de recursos al usar la aplicación

La Figura 3.32 muestra el consumo de memoria en el teléfono móvil. Se aprecia un pico de uso al principio, producto de la inicialización de la aplicación, que involucra la carga de todos los elementos necesarios. Sin embargo, el uso de memoria se reduce y se mantiene entre 256 MB y 128 MB, variando en función del módulo que se esté usando.

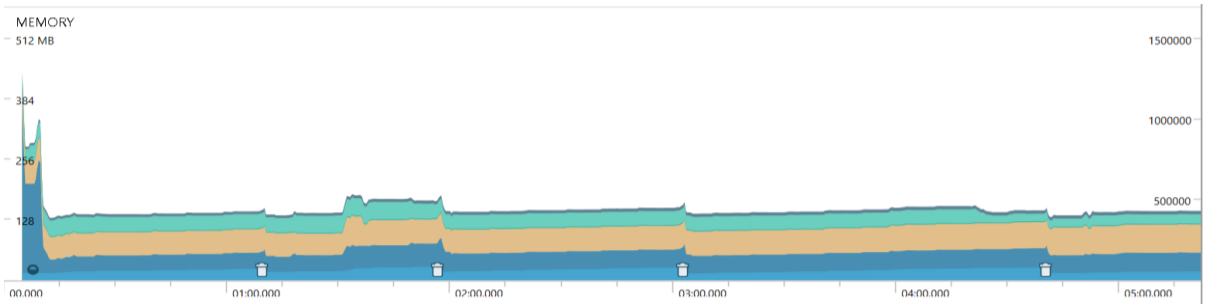

Figura 3.32 Consumo de memoria al usar la aplicación

En la Figura 3.33 se muestra los recursos de red que consume la aplicación. Al inicio se aprecia un pequeño pico, el cual simboliza la autenticación del usuario mediante Firebase; mientras el segundo pico notable, entre 0.8MB/s y 1MB/s aproximadamente, ocurre cuando se realiza un pedido e involucra el uso de los servicios GPS, envío y recepción de información de la base de datos. El tercer pico, que consume alrededor de 1.5 MB/s es producto del uso de los módulos historial e historial individual; se tiene mayor consumo debido a que se recupera toda la información de los pedidos del servidor. Al finalizar el pedido se tiene un nuevo pico, en el minuto 04:00:00, en el cual se consume recursos debido al cierre de la comunicación con Firebase. Los picos finales representan el uso del módulo de configuración debido que se obtienen los datos del usuario de la base de datos y al actualizar la información y presionar en Guardar, se subirán los nuevos datos a Firebase.

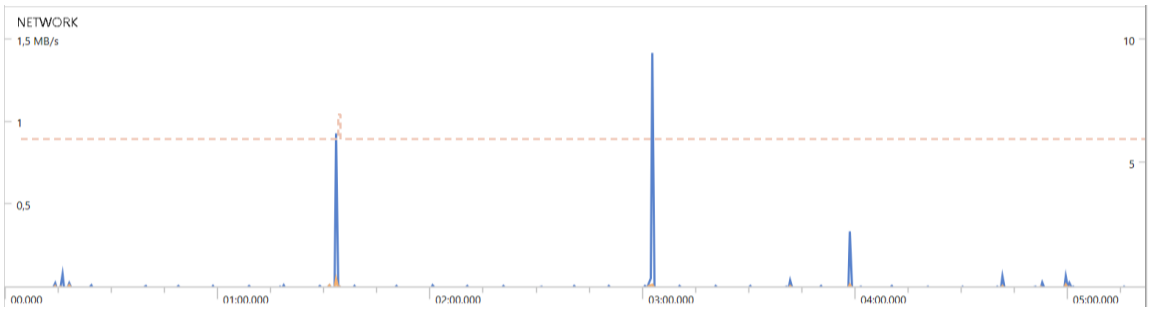

Figura 3.33 Consumo de red al usar la aplicación

La Figura 3.34 muestra el consumo de energía de la aplicación. Se tiene un uso de energía entre ligero y moderado; los picos más sobresalientes identifican procesos más pesados como el inicio de la aplicación, la realización de un pedido, uso de GPS y transiciones entre interfaces. La mayor parte del tiempo el uso de batería se mantiene bajo y constante en todos los módulos; sin embargo, aumenta al hacer uso de recursos de red para las diferentes actividades. Esto se evidencia en el minuto 01:30:00 en el cual se realiza una solicitud de pedido; así mismo, aproximadamente en el minuto 03:00:00 cuando se recupera un pedido de los módulos historial e historial individual se tiene otro pico de energía al obtener los datos desde la base de datos.

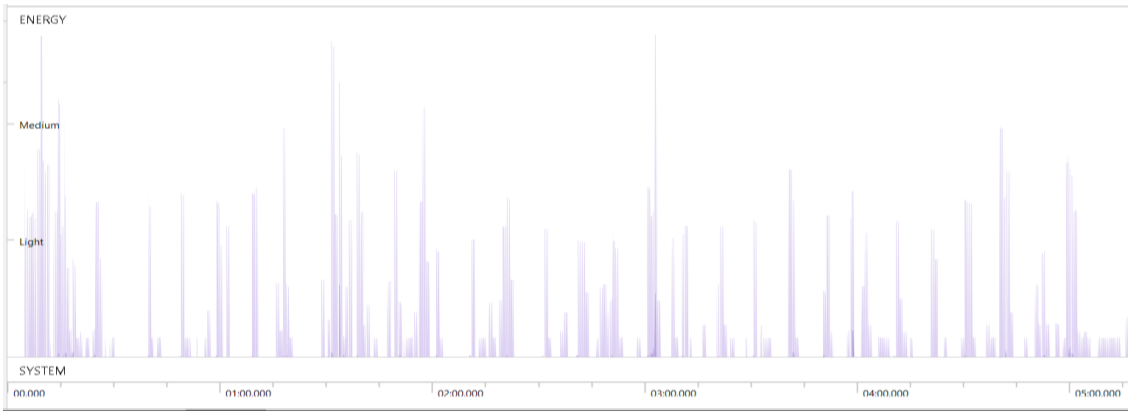

Figura 3.34 Consumo de energía al usar la aplicación

El detalle del consumo de recursos de red cuando un usuario está involucrado en un pedido se observa en la Figura 3.35.

El color azul representa la información recibida, el color naranja la información enviada y se toma como referencia un pedido con duración aproximada de 7 minutos. El primer consumo se da cuando el usuario es autenticado en la aplicación mediante el ingreso de credenciales. El pico más sobresaliente indica que el *cliente* ha realizado un pedido, y se encuentra recibiendo la información del *Distribuidor* que lo atenderá. Existe una tendencia de pequeños picos de información recibida, que son las constantes actualizaciones que se otorga al usuario referente a la ubicación del pedido, cada una de ellas consumiendo menos de 16 KB/s. Finalmente, se observa un pico al terminar el pedido, el cual indica que los datos del *Distribuidor* han sido borrados de la aplicación y se elimina de la interfaz principal del usuario la información referente al pedido.

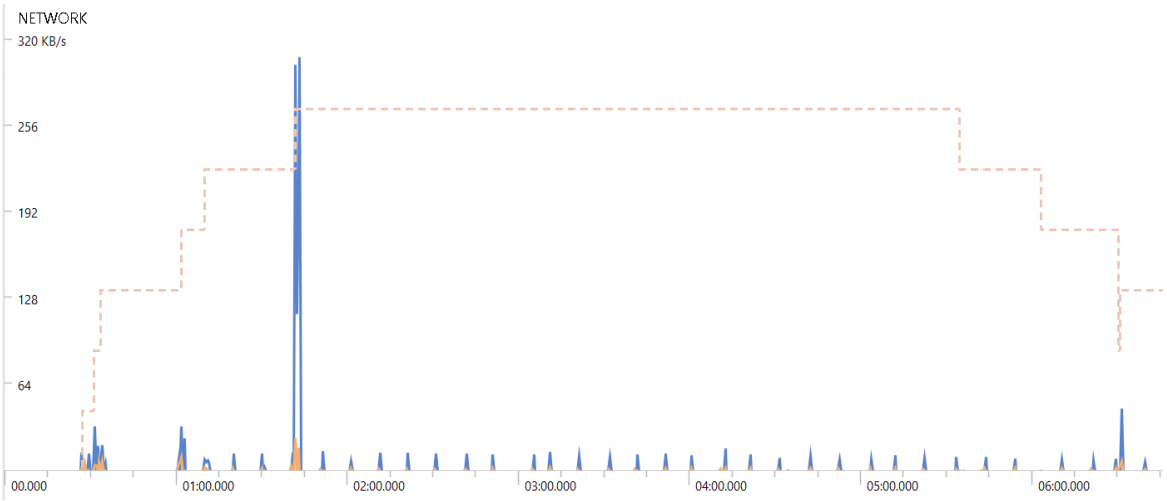

Figura 3.35 Consumo de recursos de red en un pedido

Finalmente, en la Tabla 3.4 se muestra el promedio de los recursos consumidos por cada uno de los módulos. Se observa que no existe mayor consumo en recursos de red, teniendo

un valor aproximado de 1.1 MB/s al realizar un pedido de gas, y valores bajos al usar los módulos de configuración de datos e historial de pedidos. El uso de la memoria del móvil para la aplicación es en promedio 146 MB/s, con una excepción, que como se menciona anteriormente, el consumo de recursos es mayor al inicio de la aplicación debido a la inicialización de los elementos necesarios. El consumo de la batería es ligero en la mayoría de los módulos de la aplicación, se tiene un consumo moderado en los módulos que hacen uso de recursos de red y un consumo alto al momento de inicializar la aplicación y de realizar un pedido.

Tabla 3.4 Recursos consumidos por los módulos de la aplicación

Recurso / Interfaz	Memoria	Red		Batería
		Recibido	Enviado	
Inicio de la aplicación	357,7MB/s	0 MB/s	0 MB/s	Alto
Interfaz de ingreso de credenciales	90,3MB/s	1,18 kB/s	1,1 kB/s	Ligero
Interfaz principal	109,7MB/s	1,1 MB/s	486.1 kB/s	Moderado
Módulo de ayuda	215,4 MB/s	0 MB/s	0 MB/s	Ligero
Módulo de configuración de datos	125,7MB/s	0.1 MB/s	0,3 MB/s	Ligero
Modulo Historial	222,1 MB/s	0,5 MB/s	0,1 MB/s	Moderado
Modulo Historial individual	154,8 MB/s	0,9 MB/s (por pedido)	2 kB/s	Moderado
Salir de la aplicación	104,2 MB/s	0 MB/s	0 MB/s	Ligero

4. CONCLUSIONES Y RECOMENDACIONES

4.1 CONCLUSIONES

- El presente trabajo de titulación se centra en el desarrollo de una aplicación para celulares con sistema operativo Android que permita mejorar la eficiencia en la distribución de GLP de uso doméstico, habiendo cumplido en su totalidad el objetivo planteado.

- En el desarrollo del proyecto de titulación se revisan diferentes fundamentos teóricos, los cuales incluyen: Javascript, Android Studio, Firebase, GeoFire, API de Google, entre otros. Gracias a estos conocimientos se obtuvieron nuevas habilidades en la creación de aplicaciones móviles con el uso del IDE Android Studio y de Firebase.

- Las extensas ventajas que Firebase ofrece para el desarrollo de aplicaciones móviles fueron claves importantes en la creación de la aplicación, ya que el uso de esta plataforma de desarrollo simplificó el proceso de implementación de funciones tales como: registro, autenticación y base de datos de los usuarios.

- La implementación de las APIs de Google es esencial para el funcionamiento de la aplicación, puesto que contribuyen no solo con la información acerca de los mapas, sino que además ofrecen datos exactos acerca de la ubicación del usuario en tiempo real, tanto en coordenadas como en direcciones.

- Las pruebas efectuadas en diferentes dispositivos móviles confirmaron el correcto funcionamiento de la aplicación móvil bajo ciertos escenarios, con el fin de verificar que no existan inconsistencias en cada una de las funcionalidades ofrecidas.

- El uso de la base datos de Firebase para realizar y atender pedidos por parte de los usuarios resultó exitosa y con las pruebas realizadas se comprobó que se retorna la información adecuada en tiempo real para el correcto funcionamiento de la aplicación.

- Al usar Firebase no es necesario tener servidores físicos, puesto que todos los procesos se realizan mediante un servidor virtual, lo cual ofrece la oportunidad de administrar la aplicación desde cualquier lugar, solamente teniendo acceso a Internet.

- Las pruebas de rendimiento realizadas muestran que el consumo de recursos de memoria, red y energía en el teléfono móvil tienen una relación proporcional a las actividades realizadas, existiendo un mayor consumo cuando se hace uso de servicios como GPS e Internet.

4.2 RECOMENDACIONES

- Para asegurar la veracidad de los datos ingresados por parte de los distribuidores de gas se recomienda que el administrador del sistema verifique que la información registrada concuerde con los registros de distribuidores de la Agencia de Regulación y Control Hidrocarburífero (ARCH).

- Al implementar los servicios de API de Google se recomienda verificar que se esté usando la última versión oficial, para evitar problemas de compatibilidad al momento de probar la aplicación.

- Se recomienda trabajar con la última versión estable de Android Studio para poder acceder a todas las funciones que ofrece y evitar complicaciones en el desarrollo de la aplicación, puesto que versiones anteriores podrían contener fragmentos de código base obsoletos para las aplicaciones a desarrollarse.

- Cuando el distribuidor no se encuentre trabajando se recomienda al usuario deshabilitar la aplicación en el móvil por medio del botón tipo *Switch* integrado; esto con el fin de evitar que se comparta su ubicación actual en la base de datos de Firebase.

- Las pruebas con la aplicación fueron realizadas mediante la emulación de los escenarios entre un *cliente* y un *Distribuidor*, debido a las medidas de restricción de movilidad impuestas por el Gobierno en época de pandemia por lo que se recomienda realizar las pruebas con distribuidores de gas autorizados.

- Se recomienda en un futuro agregar un sistema de pago en línea, a través de tarjeta de crédito o de débito, para agilizar el proceso de compra de cilindros de gas, puesto que la aplicación actualmente solo muestra el valor a pagar en efectivo cuando se despacha el pedido.

- La aplicación se encuentra desarrollada para funcionar solamente en teléfonos con sistema operativo Android, por lo que se recomienda el desarrollo de la app para dispositivos iOS y lograr así que una mayor población pueda acceder a este servicio.

- Para proveer mayores facilidades y opciones de identificación y autenticación de usuarios, se recomienda implementar más métodos de acceso además del correo electrónico, pudiendo ser autenticación mediante Facebook, Twitter, Google, SMS entre otros.

- Para que la aplicación sea escalable se recomienda contratar uno de los planes pagados que ofrece Firebase para el uso de más recursos y brindar accesibilidad a más usuarios. Además se recomienda implementar un servidor físico que sirva como *backup* en caso de que el servidor principal de Firebase sufra algún desperfecto.

5. REFERENCIAS

[1] C. A. El Universo, «Distribuidores de gas en Pichincha piden desinfección de cilindros,» 21 marzo 2020. [En línea]. Disponible: https://www.eluniverso.com/noticias/2020/03/21/nota/7790508/pichincha-distribuidores-gas-piden-desinfeccion-cilindros. [Último acceso: 18 abril 2020].

[2] Android, «Qué es Android,» [En línea]. Disponible: https://www.android.com/intl/es_es/what-is-android/. [Último acceso: 27 Abril 2020].

[3] Android S.O, «Android 11,» Android , Noviembre 2020. [En línea]. Disponible: https://www.android.com/android-11/.

[4] Android, «Android 4.0 Ice Cream Sandwich,» [En línea]. Disponible: https://www.android.com/intl/en_uk/history/#/icecream. [Último acceso: 27 Junio 2020].

[5] Android, «Android 7.0 Nougat,» [En línea]. Disponible: https://www.android.com/versions/nougat-7-0/. [Último acceso: 27 Junio 2020].

[6] Android, «Android 8.0 Oreo,» [En línea]. Disponible: https://www.android.com/versions/oreo-8-0/. [Último acceso: 27 Junio 2020].

[7] Software de Comunicaciones, «Arquitectura Android,» [En línea]. Disponible: https://sites.google.com/site/swcuc3m/home/android/generalidades/2-2-arquitectura-de-android. [Último acceso: 25 Abril 2020].

[8] Google Developers, «Introducción a Android Studio,» Google, [En línea]. Disponible: https://developer.android.com/studio/intro/?hl=es-419. [Último acceso: 21 Febrero 2020].

[9] EXES, Universidad a Distancia de Madrid, «Curso de Introducción a Java,» [En línea]. Disponible: https://www.mundojava.net/caracteristicas-del-lenguaje.html?Pg=java_inicial_4_1.html. [Último acceso: 18 Mayo 2020].

[10] Google Developers, «Firebase,» [En línea]. Disponible: https://firebase.google.com/?hl=es. [Último acceso: 20 abril 2020].

[11] Google Developers, «Firebase Authentication,» [En línea]. Disponible: https://firebase.google.com/products/auth?hl=es. [Último acceso: 18 Mayo 2020].

[12] Google Developers, «Firebase Realtime Database,» [En línea]. Disponible: https://firebase.google.com/products/realtime-database?hl=es. [Último acceso: 18 Mayo 2020].

[13] Google Developers, «Cloud Storage,» [En línea]. Disponible: https://firebase.google.com/docs/storage?hl=es-419. [Último acceso: 20 abril 2020].

[14] GitHub, Inc, «geofire-android,» [En línea]. Disponible: https://github.com/firebase/geofire-android. [Último acceso: 26 Junio 2020].

[15] Google Developers, «Las API de Google para Android,» [En línea]. Disponible: https://developers.google.com/android?hl=es. [Último acceso: 20 abril 2020].

[16] Google Developers, «Maps SDK for Android,» [En línea]. Disponible: https://developers.google.com/maps/documentation/android-sdk/intro?hl=es. [Último acceso: 26 Junio 2020].

[17] Google Developers, «Geocoding API,» [En línea]. Disponible: https://developers.google.com/maps/documentation/geocoding/start?hl=es. [Último acceso: 36 Junio 2020].

[18] Google Developers, «Geolocation API,» [En línea]. Disponible: https://developers.google.com/maps/documentation/geolocation/intro?hl=es. [Último acceso: 26 Junio 2020].

[19] Google Developers, «Directions API,» [En línea]. Disponible: https://developers.google.com/maps/documentation/directions/start?hl=es. [Último acceso: 26 Junio 2020].

[20] Google Developers, «Google Maps Platform Documents,» [En línea]. Disponible: https://developers.google.com/maps/documentation?hl=es. [Último acceso: 20 abril 2020].

[21] Mozilla and individual contributors, «Generalidades del protocolo HTTP,» [En línea]. Disponible: https://developer.mozilla.org/es/docs/Web/HTTP/Overview. [Último acceso: 26 Junio 2020].

[22] Xiaomi, «Redmi phones,» Xiaomi, [En línea]. Disponible: https://www.mi.com/global/list/. [Último acceso: 26 Junio 2020].

[23] Amóvil, «Sony Xperia Z5, Características,» [En línea]. Disponible: http://www.amovil.es/es/asistente/dispositivo/sony-xperia-z5/6. [Último acceso: 26 Junio 2020].

[24] Uber Technologies, Inc., «Uber,» [En línea]. Disponible: https://www.uber.com/us/es/about/. [Último acceso: 18 abril 2020].

6. ANEXOS

ANEXO A. Código de la aplicación

ANEXO B. Manual de usuario

ANEXO C. Instalador de la aplicación

ANEXO D. Creación del proyecto en Android Studio

Anexo A

Código de aplicación en digital.