Desarrollo de un prototipo de una aplicación móvil mediante el lenguaje Kotlin para el agendamiento de máquinas y la gestión documental del Laboratorio de Prototipado de la Universidad de Antioquia.

Ana María Velasco Montenegro

Proyecto de grado para optar al título de Ingeniera Electrónica

Tutor

Orlando Carrillo Perilla, Ingeniero electrónico

Universidad de Antioquia
Facultad de Ingeniería
Ingeniería Electrónica
Medellín
2024

Cita	(Velasco, A. 2024)
Referencia **Estilo APA 7 (2020)**	Velasco, A. (2024). *Desarrollo de un prototipo de una aplicación móvil mediante el lenguaje Kotlin para el agendamiento de máquinas y la gestión documental del Laboratorio de Prototipado de la Universidad de Antioquia.* Proyecto de grado. Universidad de Antioquia, Medellín.

Contenido

Resumen

En un mundo digital el cual se encuentra en constante evolución, la presencia de las aplicaciones móviles ha permitido mejorar significativamente nuestra forma de interactuar con el entorno. De acuerdo con esto, se presenta el proyecto de desarrollo de "LabProApp", una aplicación móvil realizada en Kotlin y respaldada por Firebase, con el propósito de gestionar el agendamiento de máquinas y la documentación en el Laboratorio de Prototipado de la Universidad de Antioquia.

La aplicación busca satisfacer la necesidad de realizar la reserva de máquinas, entre las cuales se tienen las impresoras 3D, la cortadora láser y CNC, al igual que permite realizar la gestión eficiente de la información generada en el laboratorio. Basada en el éxito de aplicaciones de reserva de citas en el ámbito de la salud, clases de yoga, gimnasios, entre otras, "LabProApp" contará con roles diferenciados: uno para el usuario, permitiendo solicitar turnos y visualizar la disponibilidad del espacio y de las máquinas, y otro para el administrador, permitiendo la administración de las citas y la actualización de información del laboratorio.

Este proyecto permite el desarrollo de un prototipo funcional, el cual tiene funcionalidades específicas para cada rol, el diseño de una interfaz de usuario intuitiva, secciones destinadas para visualizar diferentes actividades del laboratorio, y además la realización de pruebas para garantizar el correcto funcionamiento de la aplicación.

Se opta por realizar la aplicación en el lenguaje de programación Kotlin, el cual es reconocido por su compatibilidad con Java y su preferencia en el desarrollo de aplicaciones para Android.

La metodología propuesta se desglosa en varias etapas, desde la comprensión de las necesidades de cada rol y la definición de la estructura de la base de datos en Firebase, hasta el desarrollo de la gestión de turnos, la visualización de actividades anteriores y la evaluación del funcionamiento de la aplicación.

La aplicación LabProApp ha potenciado significativamente la eficiencia y accesibilidad en la gestión de recursos del Laboratorio de Prototipado, proporcionando una solución ágil y ajustada a las necesidades de los usuarios. Esta implementación establece una base robusta que facilitará futuras mejoras y evoluciones de la aplicación.

Introducción

En la era digital actual, la presencia de las aplicaciones móviles ha cambiado la manera en la que interactuamos con el mundo que nos rodea, debido a que son un componente importante a nivel de desarrollo y reconocimiento de las empresas en diversos sectores. Estas aplicaciones permiten realizar soluciones para mejorar la eficiencia, conectividad y la experiencia de los usuarios relacionados con ciertos productos o servicios.

En el Laboratorio de Prototipado de la Universidad de Antioquia se debe llevar el control de algunas tareas, como lo son la reserva de las máquinas tales como las impresoras 3D, la CNC y la cortadora láser. Además de poder implementar un método de gestión de la información que permita darle un manejo adecuado a esta, facilitando el acceso a los usuarios y llevando un registro de las actividades.

Una manera de atender este requerimiento es mediante el desarrollo de una aplicación móvil ya que permite cubrir una necesidad o desarrollar una tarea específica, además de facilitar los procesos recurrentes que se lleven a cabo en el contexto definido.

Una de las áreas en las que las aplicaciones han demostrado ser particularmente útiles es en la reserva de citas, entre ellas se encuentran Sanitas, MindBody, Savia, entre otras, las cuales ofrecen diferentes servicios como citas médicas, clases de yoga, gimnasios. Por ejemplo, en una clínica dental de salud en Lima se concluyó que las aplicaciones móviles multiplataformas mejoran la gestión de citas médicas [1].

Como se mencionó, existen varias aplicaciones que permiten la reserva de citas, pero estas suelen enfocarse al área de la salud. Estas aplicaciones ofrecen una serie de ventajas, como la comodidad, la flexibilidad y la accesibilidad. Las cuales también pueden ser beneficiosas en otros ámbitos, como el académico.

Se plantea la aplicación “LabProApp”, la cual se realizará en el lenguaje de programación Kotlin implementando también Firebase, la cual es una herramienta que facilita el desarrollo de aplicaciones web y móviles. Estará compuesta por un rol de cliente en el cual se podrán solicitar turnos, visualizar la agenda disponible y diferentes contenidos que se publiquen por parte del laboratorio, y un rol del laboratorio en donde se podrán realizar diferentes acciones como modificar disponibilidad, turnos, actualizar diferentes novedades del laboratorio y visualizar los datos ingresados por parte de los usuarios.

El desarrollo de esta aplicación surge debido a la necesidad de llevar a cabo un control respecto a las actividades realizadas en el laboratorio para posteriormente ser analizadas administrativamente.

Objetivos

Objetivo general:

Desarrollar un prototipo de aplicación móvil mediante el lenguaje de programación Kotlin integrado con Firebase, el cual permita realizar la reserva de diferentes máquinas que se encuentran en el laboratorio de prototipado de la Universidad de Antioquia.

Objetivos específicos:

- Desarrollar funcionalidades para los dos roles en la aplicación, un rol que permita visualizar, modificar la disponibilidad y turnos del espacio, además de almacenar los datos. El otro, que permita visualizar la disponibilidad y turnos del espacio, además de reservar las máquinas.

- Diseñar una interfaz que sea intuitiva la cual permita a los usuarios visualizar y añadir los turnos correspondientes.

- Implementar una sección en donde se visualicen las actividades previamente realizadas en el espacio.

- Implementar una sección en donde se visualicen los requisitos necesarios para acceder al espacio y manipular las máquinas.

- Realizar pruebas del funcionamiento de la aplicación.

Marco Teórico

En las últimas décadas la tecnología ha avanzado de una forma impresionante y las aplicaciones móviles son el claro ejemplo de esta evolución. Las aplicaciones móviles son programas que se ejecutan en dispositivos móviles, ya sea en teléfonos inteligentes o en tabletas. Estas aplicaciones permiten tener una gran variedad de funciones y servicios, desde el entretenimiento hasta a nivel empresarial o académico.

Las aplicaciones móviles son aquellas que fueron desarrolladas para ejecutarse en dispositivos móviles. El término móvil se refiere a poder acceder a los datos, las aplicaciones y los dispositivos desde cualquier lugar [3]. Es importante resaltar que se eligió una aplicación móvil debido a que a comparación con una aplicación web esta permite tener una mejor accesibilidad y experiencia del usuario, al estar instalada directamente en el dispositivo móvil esta proporciona un acceso rápido, además estas se pueden aprovechar de otras herramientas del dispositivo como lo son el GPS, la cámara, notificaciones, calendario, entre otras.

El simple hecho de tener la aplicación instalada en el terminar ya es una ventaja. El icono de la misma es un acceso directo gracias al cual no tenemos que teclear la dirección web para acceder a ella. Por otro lado, las tiendas de aplicaciones móviles como el App Store o Google Play nos permiten encontrar las aplicaciones de manera más sencilla que si buscáramos en la web [4].

En el Laboratorio de Prototipado de la Universidad de Antioquia, una aplicación móvil es la solución más adecuada para el agendamiento de máquinas y la gestión documental, debido a que, los usuarios del laboratorio necesitan acceder a la información y realizar sus reservas de manera rápida y sencilla, además esta aplicación permite organizar y gestionar la información de manera rápida y eficiente.

El prototipo de la aplicación se realizará en lenguaje Kotlin, el cual es un lenguaje de programación multiplataforma de código abierto desarrollado por JetBrains. Es un lenguaje compatible con Java. Kotlin es una buena opción para el desarrollo de aplicaciones móviles, debido a que tiene una serie de ventajas, es amigable, conciso y flexible, tiene un aprendizaje rápido, apoyo completo de Google, es interoperable 100% Java, es el preferido para los desarrolladores y programadores de Android, tiene funciones de alto orden y técnicas que Java no contiene [5]. Además, Kotlin ofrece una sintaxis clara, la cual reduce la cantidad de código requerido para lograr funcionalidades similares en comparación con Java.

La estructuración del código debe dividirse dependiendo de las acciones que se realicen en cada una de las fases de la aplicación, por ejemplo, la fase del registro del usuario, el inicio de sesión, el agendamiento de citas, entre otros.

El agendamiento de diferentes recursos, en este caso las máquinas de laboratorio, es un aspecto importante para las operaciones del Laboratorio de Prototipado, debido a que esto permite maximizar la productividad y reducir el tiempo de espera o de inactividad. Uno de los ejemplos es el agendamiento de citas en peluquerías, debido a que en Ecuador es un

proceso que la mayoría de los locales no ha automatizado, ya que el servicio se brinda de acuerdo a cómo lleguen los clientes, esto es un problema, ya que puede tomar mucho tiempo en recibir el servicio de peluquería [6]. Al automatizar este proceso permite que diferentes usuarios accedan al recurso sin tener que estar físicamente en el lugar, además de promover la equidad en el acceso al laboratorio, asegurando que todos los usuarios tengan la oportunidad de utilizar las máquinas de manera justa y eficaz.

Debido a que se debe realizar el despliegue de la aplicación y el almacenamiento de los datos de los usuarios que agendan las máquinas se debe utilizar una plataforma para esto, en este caso será FireBase, la cual es una plataforma móvil creada por Google, cuya función es facilitar la creación de aplicaciones mediante el almacenamiento de datos, crecimiento, monetización y análisis.

Es importante que el Laboratorio de Prototipado de la Universidad de Antioquia tenga un sistema de gestión documental ya que esto implica la creación, clasificación, almacenamiento y recuperación eficiente de documentos, asegurando que la información sea accesible, que se conserve en el tiempo y la información se mantenga segura. Al igual que en otros laboratorios del país, como por ejemplo el laboratorio de Ingeniería Industrial de la Universidad Cooperativa de Colombia, el cual requiere una implementación de un sistema de gestión documental que se adapte a las necesidades de los docentes, estudiantes, y en general a todas las personas involucradas en los procesos que se dan en este espacio [7]. Es de suma importancia que existan dos roles, ya que el rol de administrador debe tener ciertos permisos para realizar cambios frente a la disponibilidad, el acceso y la descarga de datos, mientras que el rol de usuario permite visualizar y agendar las citas de las máquinas del laboratorio.

Por último, se debe tener en cuenta una interfaz de usuario, la cual permite la interacción de los usuarios con la aplicación, para esto se realizó un mockup, el cual es un maquetado que permite realizar modificaciones previas en el diseño antes de realizar el desarrollo en el entorno de programación, permitiendo así la optimización del tiempo.

Metodología

Para la implementación de la aplicación "LabProApp", se optó por el lenguaje de programación Kotlin, reconocido por su compatibilidad con Java y su preferencia en el desarrollo de aplicaciones para Android. También, se usa Firebase para almacenar los datos de la aplicación de forma eficiente.

La estructuración del código se realizó de manera modular, adaptándose a las diversas fases de la aplicación, como el registro de usuarios, inicio de sesión, agenda del laboratorio, reserva de máquinas, historial de reservas y perfil.

La gestión de datos y almacenamiento de usuarios que agendan máquinas se realizará mediante Firebase, la cual es respaldada por Google, esto asegura una implementación eficiente del sistema de gestión documental necesario para el Laboratorio de Prototipado de la Universidad de Antioquia.

Así, la metodología se dividirá en etapas, desde la comprensión de las necesidades y requerimientos de los roles hasta la implementación de las funcionalidades específicas de la aplicación. Se lleva a cabo un enfoque modular y escalonado para garantizar un desarrollo efectivo y una aplicación móvil integral.

Dichas etapas son el registro de usuario, inicio de sesión, inicio de la aplicación, sobre nosotros, agenda, reservas, historia de reservas y perfil. Además de el almacenamiento mediante Firebase de los datos de dichas etapas.

Firebase

Firebase es una plataforma de desarrollo móvil y web ofrecida por Google la cual ha sido muy importante para el funcionamiento de la aplicación "LabPro". Este conjunto de servicios en la nube proporciona herramientas robustas para el desarrollo eficiente y la administración efectiva de aplicaciones.

Firebase abarca diversas áreas clave, algunas de las cuales han sido esenciales para la implementación de la aplicación, en este caso se utilizó Firebase Authentication, la cual facilita el registro e inicio de sesión de los usuarios en la aplicación, además de mantener la integridad de la información del usuario y Firestore Database, el cual sirve para almacenar los datos permitiendo una gestión eficiente y escalable de la información.

Para el manejo de Firebase se creó un archivo llamado *"ResourceRemote.kt",* el cual proporciona una implementación de gestión de recursos remotos en el contexto de la aplicación, con el objetivo de mejorar la interacción con Firebase. Esta clase está diseñada para encapsular y representar de manera estructurada los diferentes estados de las operaciones remotas, como éxitos, errores y procesos de carga.

```
sealed class ResourceRemote<T>(
    var data: T? = null,
    var message: String? = null,
    var status: Status? = null,
    var errorCode: Int? = null
) {
    // Clase para representar un recurso remoto exitoso
    anamvelasco
    class Success<T>(data:T): ResourceRemote<T>(data = data, status = Status.Success)

    // Clase para representar un recurso remoto en proceso de carga
    anamvelasco
    class Loading<T>(message: String = ""): ResourceRemote<T>(message = message, status = Status.Loading)

    // Clase para representar un recurso remoto con error
    anamvelasco
    class Error<T>(errorCode: Int? = null, message: String? = null): ResourceRemote <T>(errorCode = errorCode, message = message, status = Status.Error )
}
```

Figura 1. Definición de la clase ResourceRemote

Se implementa una enumeración `Status` que define los posibles estados de un recurso remoto, como Éxito, Error y Carga. Esto proporciona una manera clara y estructurada de manejar las respuestas de Firebase, mejorando la legibilidad y mantenibilidad del código.

```
enum class Status{
    Success{
        override fun toString(): String {
            return this.name
        }

    },
    Error{
        override fun toString(): String {
            return this.name
        }
    },

    Loading{
        override fun toString(): String {
            return this.name
        }
    }
}
```

Figura 2. Estados para las respuestas de Firebase

La creación de esta estructura se hace con el propósito de obtener un código más limpio, organizado y fácilmente comprensible, lo cual permite gestionar las respuestas provenientes de las operaciones en Firebase dentro de la aplicación.

Colecciones Firebase

Se utilizó Firestore para gestionar diferentes colecciones que representan distintas entidades clave. Cada entidad tiene su propia estructura de datos, y estas entidades se almacenan en colecciones específicas dentro de Firestore. Estas colecciones y sus respectivas estructuras de datos definen la organización y el flujo de información en la aplicación, permitiendo un acceso eficiente y una gestión de datos coherente a través de Firebase Firestore. Se definieron tres colecciones, entre las cuales están:

- **Colección *"Galeria":***
  La colección *"Galeria"* almacena información relacionada con imágenes y sus propietarios en la aplicación. Cada documento dentro de esta colección representa una imagen en la galería y contiene atributos como "id" (identificador único), *"name"* (nombre de la imagen), *"ownerUid"* (identificador único del propietario), *"ownerRole"* (rol del propietario), y *"urlPicture"* (URL de la imagen).

```
package com.ana.labpro.model

//Variables definidas para la galeria en Firebase.
anamvelasco
data class Galeria(
    var id: String? = null,
    var name: String? = null,
    var ownerUid: String? = null,
    var ownerRole: String = "user",
    var urlPicture: String? = null
)
```

Figura 3. Documento "Galeria.kt"

***Figura 4. Colección** "Galeria" en Firebase*

- **Colección *"Reservas":***
  Se encarga de almacenar la información relacionada con las reservas realizadas por los usuarios. Cada documento en esta colección representa una reserva y contiene atributos como *"id"* (identificador único), *"name"* (nombre del usuario), *"cedula"* (número de identificación), *"email"* (correo electrónico del usuario), *"programa"* (programa académico), *"maquina"* (máquina reservada), *"date"* (fecha de la reserva), *"hour"* (hora de la reserva) y *"uid"* (identificador único del usuario).

```kotlin
package com.ana.labpro.model

//Variables definidas para reservas en Firebase.
anamvelasco
data class Reservas(
    var id: String? = null,
    var name: String? = null,
    var cedula: Int? = null,
    var email: String? = null,
    var programa: String? = null,
    var maquina: String? = null,
    var date: String? = null,
    var hour: String? = null,
    var uid: String? = null
)
```

Figura 5. Documento "Reservas.kt"

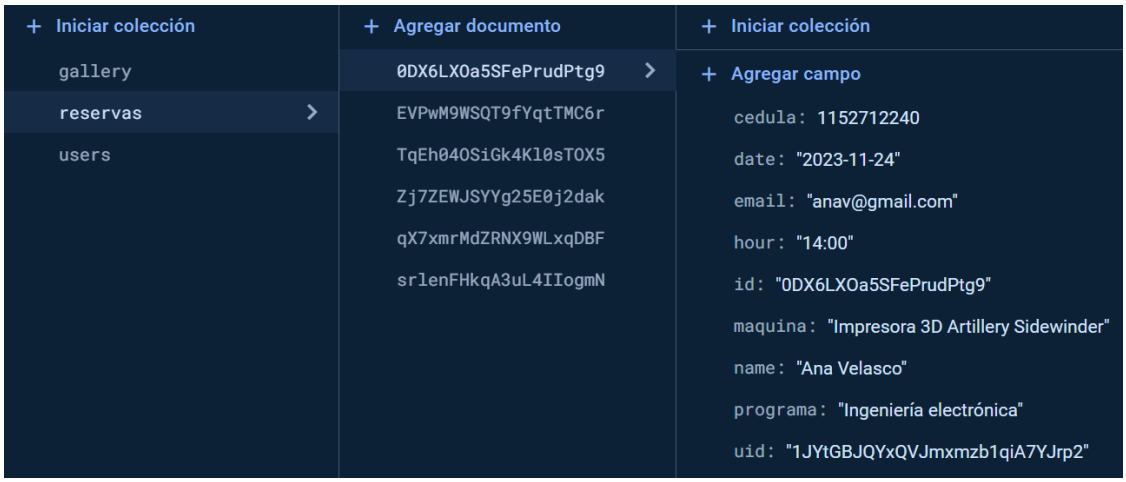


Figura 6. Colección "reservas" en Firebase

- **Colección *"User"*:**
  Esta colección almacena información sobre los usuarios registrados en la aplicación. Cada documento en esta colección representa un usuario y contiene atributos como *"uid"* (identificador único), *"namer"* (nombre del usuario), *"lastnamer"* (apellido del usuario), *"identir"* (número de identificación), *"programar"* (programa académico), *"email"* (correo electrónico del usuario), *"numReservas"* (número de reservas realizadas por el usuario) y *"role"* (rol del usuario).

```
data class User(
    var uid: String? = null,
    var namer: String? = null,
    var lastnamer: String? = null,
    var identir: String? = null,
    var programar: String? = null,
    var email: String? = null,
    var numReservas: Int = 0,
    var role: String = "user"
)
```

Figura 7. Documento "User.kt"

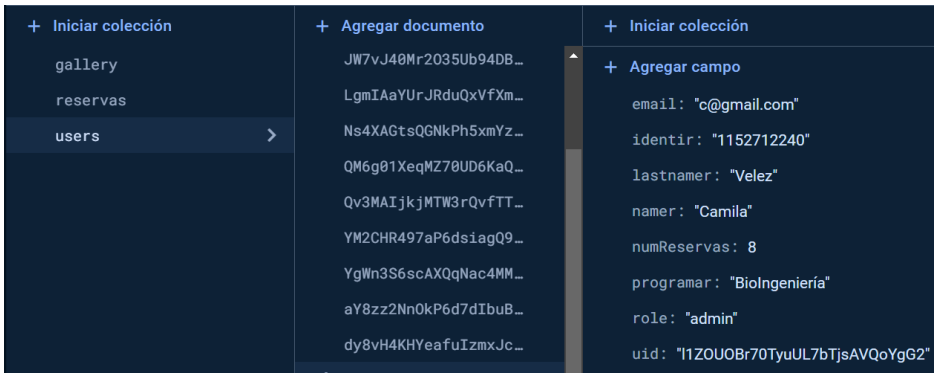


Figura 8. Colección "users" en Firebase

Registro de usuario

Para el registro de cada uno de los usuarios se realiza utilizando Firebase Authentication, y específicamente, la clase FirebaseAuth que forma parte de Firebase SDK.

Se crea el archivo llamado *"User.kt*" el cual se usa para representar la estructura de los datos de usuario que se almacenan en Firebase Firestore además de manejar la información del usuario durante el registro, inicio de sesión y otras operaciones relacionadas con el usuario. En este archivo se definen diferentes variables como el id, nombre, apellido, programa, email, contraseña, entre otros.

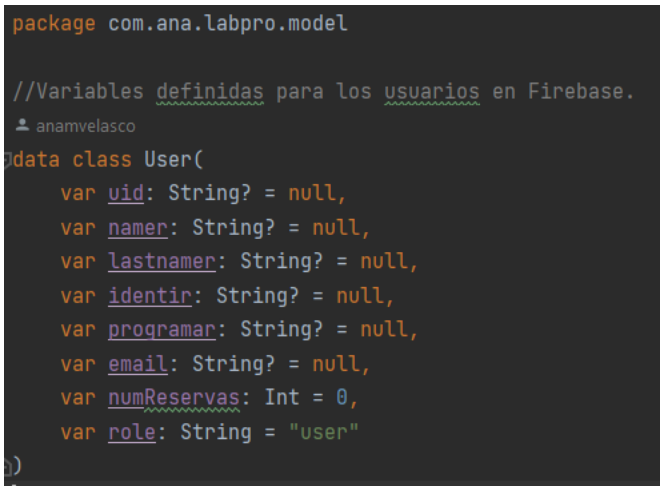


Figura 9. Documento "User.kt"

Se crea también un archivo llamado *"UserRepository.kt"* el cual tiene diferentes funciones que se realizaron para la gestión de usuarios en la aplicación, especialmente en lo que respecta a la autenticación y el registro en Firebase.

Entre las funciones se encuentran:

- **registerUser***:* La cual registra un nuevo usuario en Firebase Authentication utilizando el correo electrónico y la contraseña proporcionados. Si la operación es exitosa, devuelve el UID del usuario recién registrado. En caso de errores, se manejan excepciones específicas de Firebase Authentication y se registran en la consola.

```
suspend fun registerUser(email: String, password: String): ResourceRemote<String?> {
    return try {
        val result = auth.createUserWithEmailAndPassword(email, password).await()
        ResourceRemote.Success(data = result.user?.uid)
    } catch (e: FirebaseAuthException) {
        Log.e( tag: "Register", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "RegisterNetwork", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 10.Función "registerUser"

- **createUser**: Después de registrar el usuario en Firebase Authentication, esta función crea un documento correspondiente en Firestore con la información adicional del usuario, como nombre, apellido, etc.

```
suspend fun createUser(user: User): ResourceRemote<String?> {
    return try {
        user.uid?.let { db.collection( collectionPath: "users").document(it).set(user).await() }
        ResourceRemote.Success(data = user?.uid)
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

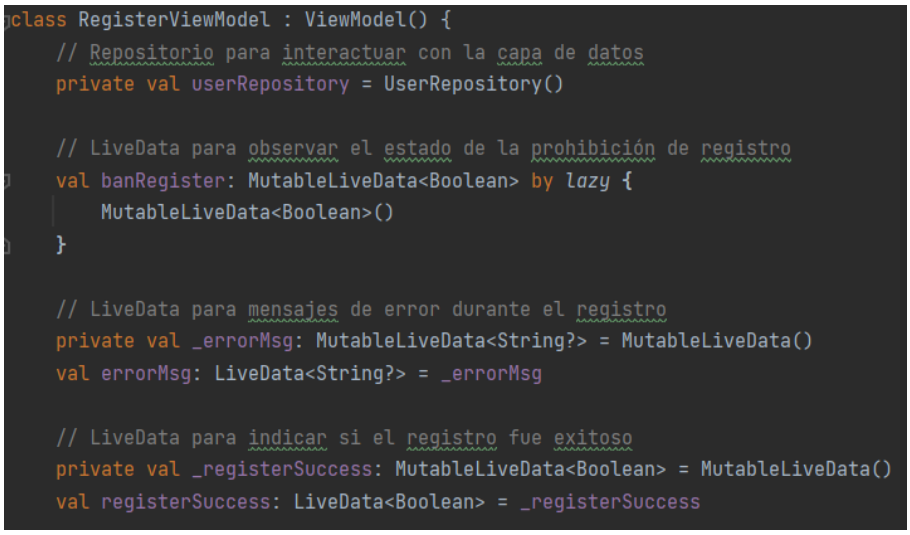
Figura 11. Función "createUser"

El *"RegisterActivity.kt"* es un archivo que contiene una clase llamada *"RegisterActivity"* que forma parte del proceso de registro de usuarios en la aplicación. Su función principal es gestionar la interfaz de usuario definida en activity_register.xml y orquestar las interacciones del usuario con dicha interfaz. Esta actividad inicializa un objeto llamado el RegisterViewModel que mediante ViewModelProvider maneja la lógica del registro. Además, observa eventos relevantes como el éxito del registro, mensajes de error y la prohibición de registro a través de LiveData provenientes del RegisterViewModel. Al registrarse con éxito, la actividad navega a la pantalla principal de la aplicación mediante un intent.

```
// Observador para el registro exitoso, navegar a la pantalla principal de la aplicación
val banRegisterObserver = Observer<Boolean> { banRegister ->
    if (banRegister) {
        val intent = Intent( packageContext: this, NavigationDrawerActivity::class.java)
        startActivity(intent)
        finish()
    }
}

registerViewModel.banRegister.observe( owner: this, banRegisterObserver)

// Observador para mensajes de error, muestra Snackbar con opción para continuar
val errorMsgObserver = Observer<String?> { errorMsg ->
    errorMsg?.let { it: String
        Snackbar.make(view, it, Snackbar.LENGTH_INDEFINITE)
            .setAction( text: "Continuar") {}
            .show()
    }
}
registerViewModel.errorMsg.observe( owner: this, errorMsgObserver)

// Observador para el éxito del registro, retroceder al fragmento anterior
registerViewModel.registerSuccess.observe( owner: this) { it: Boolean!
    onBackPressedDispatcher.onBackPressed()
}
```

Figura 12. Documento "RegisterAcitivity.kt"

El *"RegisterViewModel.kt"* es un archivo que contiene una clase llamada *"RegisterViewModel"* que se encarga de la lógica del registro de usuarios. Utiliza un objeto UserRepository para interactuar con la capa de datos, gestionando las operaciones relacionadas con la autenticación y el registro en Firebase. Este ViewModel contiene LiveData para observar el estado del registro, mensajes de error y la prohibición de registro. Además, valida los datos ingresados por el usuario, realiza la autenticación y crea un nuevo usuario en la base de datos Firestore. El RegisterViewModel se comunica con la actividad (RegisterActivity.kt) y el repositorio para coordinar las acciones necesarias durante el proceso de registro.



```
class RegisterViewModel : ViewModel() {
    // Repositorio para interactuar con la capa de datos
    private val userRepository = UserRepository()

    // LiveData para observar el estado de la prohibición de registro
    val banRegister: MutableLiveData<Boolean> by lazy {
        MutableLiveData<Boolean>()
    }

    // LiveData para mensajes de error durante el registro
    private val _errorMsg: MutableLiveData<String?> = MutableLiveData()
    val errorMsg: LiveData<String?> = _errorMsg

    // LiveData para indicar si el registro fue exitoso
    private val _registerSuccess: MutableLiveData<Boolean> = MutableLiveData()
    val registerSuccess: LiveData<Boolean> = _registerSuccess
```

Figura 13. Documento "RegisterViewModel.kt"

En el archivo *"RegisterViewModel.kt"* se encuentra una función *"validateData"* la cual es llamada cuando el usuario intenta registrarse, y realiza diversas validaciones, como verificar la completitud de los campos, la coincidencia de contraseñas y la longitud mínima de la contraseña. Además, valida el formato correcto del correo electrónico. En caso de que alguna validación no sea exitosa, la función actualiza LiveData para notificar a la interfaz de usuario sobre cualquier mensaje de error. Si todas las validaciones son exitosas, la función procede a intentar registrar al usuario utilizando el UserRepository.

```
fun validateData(
    email: String,
    password: String,
    repeatPassword: String,
    name: String,
    lastName: String,
    cedula: String,
    programa: String
) {
    if (email.isEmpty() || password.isEmpty() || repeatPassword.isEmpty()) {
        // Validar campos obligatorios
        _errorMsg.value = "Debe escribir todos los datos de registro"
        banRegister.value = false
    } else if (password != repeatPassword) {
        // Validar coincidencia de contraseñas
        _errorMsg.value = "Las contraseñas no coinciden"
        banRegister.value = false
    } else {
        if (password.length < 6) {
            // Validar longitud mínima de la contraseña
            _errorMsg.value = "Las contraseña debe tener mínimo 6 dígitos"
        } else {
            if (!emailValidator(email)) {
                // Validar formato correcto del correo electrónico
                _errorMsg.value = "El correo electrónico está mal escrito, revise su formato"
```

Figura 14. Función validateData

El archivo *"RegistroFragment.kt"* es un fragmento que forma parte de la interfaz de usuario para el proceso de registro. Se utiliza para mostrar una parte específica de la interfaz de registro en la aplicación. Este fragmento se infla en la actividad de registro (RegisterActivity.kt). Su función principal es contribuir a la modularidad y organización de la interfaz, dividiéndola en fragmentos reutilizables y manejables. Estos componentes, junto con User.kt y UserRepository.kt, colaboran para implementar un flujo de registro coherente y eficiente en la aplicación. La actividad gestiona la interfaz, el ViewModel controla la lógica y la comunicación con el repositorio, y el fragmento es utilizado para estructurar la interfaz de usuario de manera modular. La interconexión de estos elementos permite una implementación robusta del registro de usuarios en la aplicación.

```
class RegistroFragment : Fragment() {

    anamvelasco *
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflar el diseño de este fragmento
        return inflater.inflate(R.layout.fragment_registro, container, attachToRoot: false)
    }

}
```

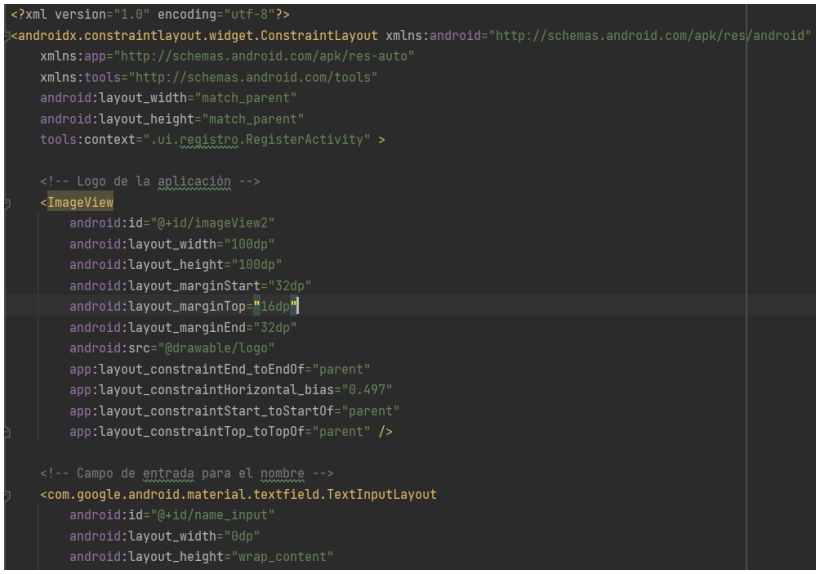
Figura 15. Documento "RegistroFragment.kt"

Luego de completar el proceso de registro de los datos de los usuarios en la aplicación, esta información se refleja en la sección de autenticación de Firebase. En este apartado, se exhiben de manera detallada los correos electrónicos asociados, las fechas de creación y acceso, así como el UID (Identificador Único) correspondiente a cada usuario registrado. Este enfoque permite una gestión centralizada de los usuarios dentro del entorno de autenticación de Firebase, brindando una visión clara y organizada de la información esencial relacionada con los usuarios de la aplicación.

Identificador	Proveedores	Fecha de creación	Fecha de acceso	UID de usuario
edwinacubillos@gmail....		27 nov 2023	27 nov 2023	LgmlAaYUrJRduQxVfXmV0Gq...
anitamar2@gmail.com		27 nov 2023	27 nov 2023	s7PZtZU8SjWWmpWYsTprAof...

Figura 16. Datos de usuarios en Firebase

El archivo “activity_register.xml” define la interfaz de usuario para la actividad de registro (“RegisterActivity”). En esta interfaz, se encuentran diversos componentes que permiten a los usuarios ingresar información personal, como nombre, apellido, cédula, programa académico, correo electrónico y contraseña. El diseño se organiza mediante un “ConstraintLayout”, asegurando una disposición ordenada y amigable. Cada campo de entrada está encapsulado en un “TextInputLayout”, lo que facilita la validación y mejora la experiencia del usuario. Además, se incluyen funcionalidades como la opción de mostrar/ocultar contraseñas. El botón de registro (“register_button”) el cual activa el proceso de registro cuando se selecciona. En conjunto, esta disposición proporciona una interfaz intuitiva y efectiva para que los usuarios completen el registro en la aplicación.



```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.registro.RegisterActivity" >

    <!-- Logo de la aplicación -->
    <ImageView
        android:id="@+id/imageView2"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_marginStart="32dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="32dp"
        android:src="@drawable/logo"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.497"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <!-- Campo de entrada para el nombre -->
    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/name_input"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
```

Figura 17. Documento "activity_register.xml"

Inicio de sesión de usuario

El inicio de sesión se realiza mediante Firebase Authentication, que se encarga de la gestión de la autenticación de usuarios. Este servicio de Firebase facilita la creación, verificación y gestión de cuentas de usuario, asegurando un proceso de inicio de sesión eficiente y protegido contra amenazas de seguridad.

Durante el proceso de inicio de sesión, se solicita al usuario que ingrese su dirección de correo electrónico y contraseña, que son verificados mediante Firebase Authentication. En caso de éxito, se permite el acceso a la aplicación.

Para el inicio de sesión se usa el archivo "UserRepository.kt" el cual se mencionó previamente para el registro de usuario UserRepository.kt, por ende, las funciones relacionadas con el proceso de inicio de sesión son loginUser y verifyUser.

Entre las funciones se encuentran:

- **loginUser**: Esta función se encarga de iniciar sesión en Firebase Authentication utilizando el correo electrónico y la contraseña proporcionados y retorna un objeto ResourceRemote que encapsula el resultado de la operación.
  Si el inicio de sesión es exitoso, devuelve el UID del usuario. En caso de error, devuelve un objeto ResourceRemote.Error con un mensaje descriptivo del error.

```
suspend fun loginUser(email: String, password: String): ResourceRemote<String?> {
    return try {
        val result = auth.signInWithEmailAndPassword(email, password).await()
        ResourceRemote.Success(data = result.user?.uid)
    } catch (e: FirebaseAuthException) {
        Log.e( tag: "FirebaseAuthError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 18. Función "loginUser"

- **verifyUser:** Esta función verifica si hay un usuario autenticado en Firebase Authentication y retorna un objeto ResourceRemote que almacena el resultado de la verificación. Si hay un usuario autenticado, devuelve true; de lo contrario, devuelve false. En caso de error, devuelve un objeto ResourceRemote.Error con un mensaje descriptivo del error.

```
suspend fun verifyUser(): ResourceRemote<Boolean> {
    return try {
        val userLoaded = auth.currentUser != null
        ResourceRemote.Success(data = userLoaded)
    } catch (e: FirebaseAuthException) {
        Log.e( tag: "FirebaseAuthError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 19. Función "verifyUser"

En el archivo “LoginActivity.kt” está la actividad principal encargada de gestionar el proceso de inicio de sesión. Al abrir la aplicación, esta actividad verifica si el usuario ya está autenticado. Si es así, redirige al usuario a la NavigationDrawerActivity, que representa la interfaz principal de la aplicación. En caso contrario, muestra la interfaz de inicio de sesión. La actividad utiliza LoginViewModel para realizar la lógica de inicio de sesión y manejar eventos relacionados.

```
class LoginActivity : AppCompatActivity() {
    private lateinit var activityLoginBinding: ActivityLoginBinding
    private lateinit var loginViewModel: LoginViewModel

    // LiveData para mensajes de error durante el login
    private val _errorMsg: MutableLiveData<String?> = MutableLiveData()
    val errorMsg: MutableLiveData<String?> = _errorMsg

    anamvelasco *
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Inflar el diseño de la actividad
        activityLoginBinding = ActivityLoginBinding.inflate(layoutInflater)
        loginViewModel = ViewModelProvider( owner: this)[LoginViewModel::class.java]
        val view = activityLoginBinding.root
        setContentView(view)

        // Observador para el login exitoso, navegar a la pantalla principal de la aplicación
        val banLoginObserver = Observer<Boolean> { banLogin ->
            if (banLogin) {
                val intent = Intent( packageContext: this, NavigationDrawerActivity::class.java)
                startActivity(intent)
                finish()
            }
        }

        loginViewModel.banLogin.observe( owner: this, banLoginObserver)
```

Figura 20. Documento "LoginActivity"

Las funciones que se encuentran en el archivo *"LoginActivity"* son las siguientes:

- **forgotPasswordClicked:** Es la función que maneja el clic en el enlace "¿Olvidaste tu contraseña?". Verifica si se ha ingresado un correo electrónico y, si es así, verifica la existencia del correo en Firestore y envía un correo de restablecimiento de contraseña si es válido.

```
fun forgotPasswordClicked(view: View) {
    val email = activityLoginBinding.emailLoginEditText.text.toString()

    if (email.isNotEmpty()) {
        // Verificar si el correo electrónico existe en Firestore
        checkIfUserExists(email)
    } else {
        Toast.makeText( context: this,  text: "Ingrese su dirección de correo electrónico primero.", Toast.LENGTH_LONG).show()
    }
}
```

Figura 21. Función "forgotPasswordClicked"

- **checkIfUserExists y sendPasswordResetEmail:** Son funciones usadas para verificar la existencia de un usuario con el correo electrónico proporcionado y enviar un correo electrónico de restablecimiento de contraseña, respectivamente.

```
private fun checkIfUserExists(email: String) {
    val db = FirebaseFirestore.getInstance()
    val usersCollection = db.collection( collectionPath: "users")

    usersCollection.whereEqualTo( field: "email", email).get()
        .addOnSuccessListener { documents ->
            if (documents.isEmpty) {
                // El correo electrónico no está asociado con ninguna cuenta en Firestore
                Toast.makeText(
                    context: this,
                    text: "No hay ninguna cuenta asociada con este correo electrónico.",
                    Toast.LENGTH_LONG
                ).show()
            } else {
                // El correo electrónico está asociado con una cuenta, enviar el correo de restablecimiento
                sendPasswordResetEmail(email)
            }
        }
        .addOnFailureListener { exception ->
            // Error al consultar Firestore
            Toast.makeText(
                context: this,
                text: "Error al verificar la existencia del usuario. Inténtalo de nuevo.",
                Toast.LENGTH_LONG
            ).show()
        }
}
```

Figura 22. Función "checkIfUserExists"

```
private fun sendPasswordResetEmail(email: String) {
    FirebaseAuth.getInstance().sendPasswordResetEmail(email)
        .addOnCompleteListener { task ->
            if (task.isSuccessful) {
                Toast.makeText(
                    context: this,
                    text: "Correo de recuperación enviado. Revise su bandeja de entrada.",
                    Toast.LENGTH_LONG
                ).show()
            } else {
                Toast.makeText(
                    context: this,
                    text: "Error al enviar el correo de recuperación. Verifique su dirección de correo electrónico
                    Toast.LENGTH_LONG
                ).show()
            }
        }
}
```

Figura 23.Función "sendPasswordResetEmail"

El archivo *"LoginFragment.kt"* es un fragmento que se infla dentro de la interfaz de *"LoginActivity.kt"*. Su función principal es proporcionar la interfaz gráfica para el proceso de inicio de sesión. En este fragmento, se muestran campos de entrada de correo electrónico y contraseña, así como botones para realizar el inicio de sesión y la navegación hacia la pantalla de registro.

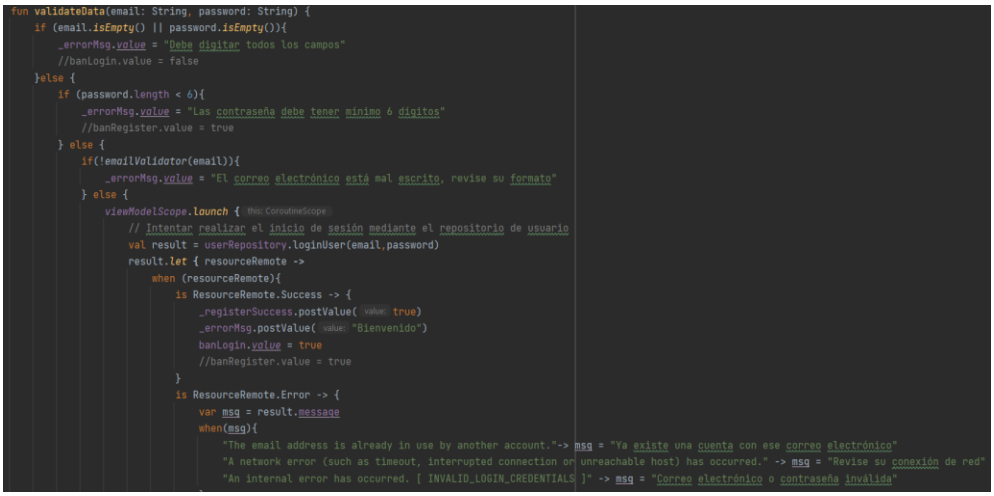
```
class LoginFragment : Fragment() {

    // Método llamado al crear la vista del fragmento
    anamvelasco *
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflar el diseño del fragmento utilizando el archivo XML fragment_login
        return inflater.inflate(R.layout.fragment_login, container,  attachToRoot: false)
    }
}
```

Figura 24. Documento "LoginFragment.kt"

El archivo *"LoginViewModel.kt"* contiene una clase ViewModel que actúa como intermediario entre la interfaz de usuario y el repositorio de datos (UserRepository). Además de contener funciones para validar los datos de inicio de sesión, verificar si el usuario ya está autenticado, y realizar el proceso de inicio de sesión. También gestiona eventos y errores relacionados con el inicio de sesión, utilizando LiveData para notificar a la interfaz de usuario sobre cambios en los datos.

Las funciones que se encuentran en el archivo *"LoginViewModel.kt"* son las siguientes:

- **validateData:** Es una función muy importante para la validación de datos antes de intentar el inicio de sesión. Se asegura de que se ingresen tanto el correo electrónico como la contraseña, verifica si la contraseña cumple con los requisitos mínimos y utiliza la función *"emailValidator"* para comprobar el formato del correo electrónico. Si pasa todas las validaciones, intenta iniciar sesión llamando a la función *"loginUser"* del repositorio.



```
fun validateData(email: String, password: String) {
    if (email.isEmpty() || password.isEmpty()){
        _errorMsg.value = "Debe digitar todos los campos"
        //banLogin.value = false
    }else {
        if (password.length < 6){
            _errorMsg.value = "Las contraseña debe tener mínimo 6 dígitos"
            //banRegister.value = true
        } else {
            if(!emailValidator(email)){
                _errorMsg.value = "El correo electrónico está mal escrito, revise su formato"
            } else {
                viewModelScope.launch { this: CoroutineScope
                    // Intentar realizar el inicio de sesión mediante el repositorio de usuario
                    val result = userRepository.loginUser(email,password)
                    result.let { resourceRemote ->
                        when (resourceRemote){
                            is ResourceRemote.Success -> {
                                _registerSuccess.postValue( value: true)
                                _errorMsg.postValue( value: "Bienvenido")
                                banLogin.value = true
                                //banRegister.value = true
                            }
                            is ResourceRemote.Error -> {
                                var msg = result.message
                                when(msg){
                                    "The email address is already in use by another account."-> msg = "Ya existe una cuenta con ese correo electrónico"
                                    "A network error (such as timeout, interrupted connection or unreachable host) has occurred." -> msg = "Revise su conexión de red"
                                    "An internal error has occurred. [ INVALID_LOGIN_CREDENTIALS ]" -> msg = "Correo electrónico o contraseña inválida"
```

Figura 25. Función "validateData"

- **emailValidator:** Es una función que utiliza una expresión regular para validar el formato del correo electrónico.

- **verifyUser:** Esta función verifica si ya hay un usuario autenticado. Si es así, establece *"_userLoggedIn"* en verdadero, lo cual indica que el usuario ya está autenticado.

```
fun verifyUser() {
    viewModelScope.launch { this: CoroutineScope
        val result = userRepository.verifyUser()
        result.let { resourceRemote ->
            when (resourceRemote) {
                is ResourceRemote.Success -> {
                    if (result.data == false) {
                        _userLoggedIn.postValue( value: true)
                        Log.d( tag: "LoginViewModel",  msg: "Usuario verificado")
                    }
                }
                is ResourceRemote.Error -> {
                    var msg = result.message
                    when (msg) {
                        "A network error (such as timeout, interrupted connection or unreachable host) has occurred." -> {
                            msg = "Revise su conexión de red al verificar el usuario"
                            Log.d( tag: "LoginViewModel",  msg: "Error de red al verificar el usuario")
                        }
                        "An internal error has occurred." -> {
                            msg = "Error interno al verificar el usuario"
                            Log.d( tag: "LoginViewModel",  msg: "Error interno al verificar el usuario")
                        }
                        else -> {
                            // Puedes agregar más casos según sea necesario
                        }
                    }
                    _errorMsg.postValue(msg!!)
                }
                else -> {
                    // No uses este caso si no es necesario
                }
```

Figura 26. Función "verifyUser"

En el archivo *"activity_login.xml"* se define la interfaz de usuario para la pantalla de inicio de sesión de la aplicación. Contiene elementos como un campo de entrada para el correo electrónico, otro para la contraseña, botones para iniciar sesión y registrarse, y un enlace para recuperar la contraseña. Además, presenta un diseño basado en ConstraintLayout para garantizar una disposición flexible y adaptable en diferentes pantallas.

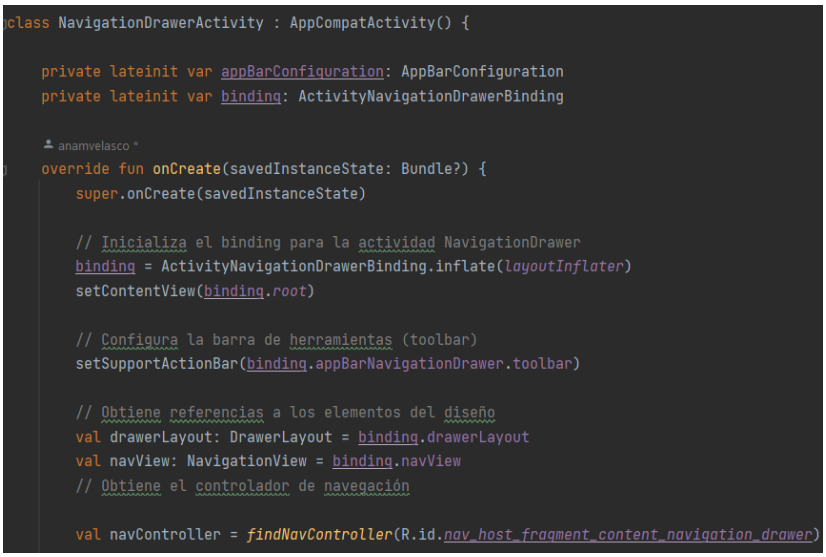
```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.login.LoginActivity" >

    <!-- Logo de la aplicación -->
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:layout_marginStart="32dp"
        android:layout_marginTop="32dp"
        android:layout_marginEnd="32dp"
        android:src="@drawable/logo"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />


    <!-- Campo de entrada para el correo electrónico -->
    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/email_login_input"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
```

Figura 27. Documento "activity_login.xml"

Caja de navegación (Navigation Drawer)

El Navigation Drawer es un componente de interfaz de usuario que permite acceder a las secciones principales de la aplicación. Se implementó para ofrecer una experiencia de navegación fluida y organizada. Es un menú lateral que se despliega desde el borde izquierdo de la pantalla, revelando opciones de navegación, entre las cuales se encuentran: inicio, sobre nosotros, agenda, reservas, historial de reservas y perfil.

En el archivo *"NavigationDrawerActivity.kt"* se define la lógica para la actividad principal de la aplicación que utiliza un Navigation Drawer.



```
class NavigationDrawerActivity : AppCompatActivity() {

    private lateinit var appBarConfiguration: AppBarConfiguration
    private lateinit var binding: ActivityNavigationDrawerBinding

    anamvelasco *
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Inicializa el binding para la actividad NavigationDrawer
        binding = ActivityNavigationDrawerBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Configura la barra de herramientas (toolbar)
        setSupportActionBar(binding.appBarNavigationDrawer.toolbar)

        // Obtiene referencias a los elementos del diseño
        val drawerLayout: DrawerLayout = binding.drawerLayout
        val navView: NavigationView = binding.navView
        // Obtiene el controlador de navegación

        val navController = findNavController(R.id.nav_host_fragment_content_navigation_drawer)
```

Figura 28. Documento "NavigationDrawerActivity.kt"

Las funciones más destacadas de *"NavigationDrawerActivity.kt"* son las siguientes:

- **onCreate:** Este método se llama al crear la actividad. Configura el diseño y la barra de herramientas (toolbar), obtiene referencias a los elementos de diseño como el DrawerLayout y NavigationView, y configura el controlador de navegación con sus destinaciones principales.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    // Inicializa el binding para la actividad NavigationDrawer
    binding = ActivityNavigationDrawerBinding.inflate(layoutInflater)
    setContentView(binding.root)

    // Configura la barra de herramientas (toolbar)
    setSupportActionBar(binding.appBarNavigationDrawer.toolbar)

    // Obtiene referencias a los elementos del diseño
    val drawerLayout: DrawerLayout = binding.drawerLayout
    val navView: NavigationView = binding.navView
    // Obtiene el controlador de navegación
    val navController = findNavController(R.id.nav_host_fragment_content_navigation_drawer)
    // Define las destinaciones principales del AppBar
    appBarConfiguration = AppBarConfiguration(
        setOf(
            R.id.navigation_homee,
            R.id.navigation_about,
            R.id.navigation_horario,
            R.id.navigation_reservas,
            R.id.navigation_perfil,
            R.id.navigation_perfil
        ), drawerLayout
    )
    // Configura la barra de herramientas con el controlador de navegación y la configuración del AppBar
    setupActionBarWithNavController(navController, appBarConfiguration)
    navView.setupWithNavController(navController)
}
```

Figura 29. Función "onCreate"

- **onSupportNavigateUp**: Este método se llama cuando se presiona el botón de navegación hacia arriba en la barra de herramientas. Permite navegar hacia arriba en la jerarquía de destinaciones.

```
override fun onSupportNavigateUp(): Boolean {
    val navController = findNavController(R.id.nav_host_fragment_content_navigation_drawer)
    return navController.navigateUp(appBarConfiguration) || super.onSupportNavigateUp()
}
```

Figura 30. Función "onSupportNavigateUp"

En el archivo *"activity_navigation_drawer.xml"* se muestra el diseño principal de la actividad que utiliza un DrawerLayout. Un DrawerLayout es un contenedor que permite tener un diseño de cajón (drawer) deslizable desde un borde de la pantalla para mostrar opciones de navegación. Este archivo incluye dos elementos principales:

- **app_bar_navigation_drawer:** Este componente incluye el diseño de la barra de la aplicación (app_bar_navigation_drawer.xml). La barra de la aplicación toemeelementos como el título de la pantalla y el ícono de menú para abrir y cerrar el drawer.

- **NavigationView:** Este componente representa el menú de navegación lateral. Contiene un encabezado (nav_header_navigation_drawer.xml) que puede incluir información del usuario y un menú (bottom_nav_menu.xml) que define las opciones de navegación disponibles en el drawer.

```
<androidx.drawerlayout.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <include
        android:id="@+id/app_bar_navigation_drawer"
        layout="@layout/app_bar_navigation_drawer"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_navigation_drawer"
        app:menu="@menu/bottom_nav_menu" />
</androidx.drawerlayout.widget.DrawerLayout>
```

Figura 31. Documento "activity_navigation_drawer.xml"

Sección inicio

La sección de inicio ofrece a los usuarios una interfaz atractiva y funcional. Esta sección proporciona acceso a las funcionalidades clave, permitiendo a los usuarios explorar y administrar galerías. En esta sección se implementaron características específicas para los usuarios con rol de administrador, permitiéndole crear y eliminar galerías. Esta sección depende de algunos archivos los cuales son: *"GaleriaRepository.kt", "HomeeFragment.kt", "HomeeViewModel.kt", "AdapterDeHome.kt", "fragment_homee.xml"* y *"card_view_gallery_item.xml"*.

En el archivo *"GaleriaRepository.kt"* se gestiona la interacción con la base de datos Firestore de Firebase para las entidades relacionadas con galerías. Las galerías representan conjuntos de imágenes. Es importante resaltar que la creación de nuevas galerías está sujeta a validaciones de roles, asegurando que solo los administradores tengan la capacidad de realizar esto. La eliminación de galerías, por otro lado, involucra una cuidadosa validación de roles y propietarios, garantizando que solo aquellos con los privilegios adecuados puedan llevar a cabo esta acción.

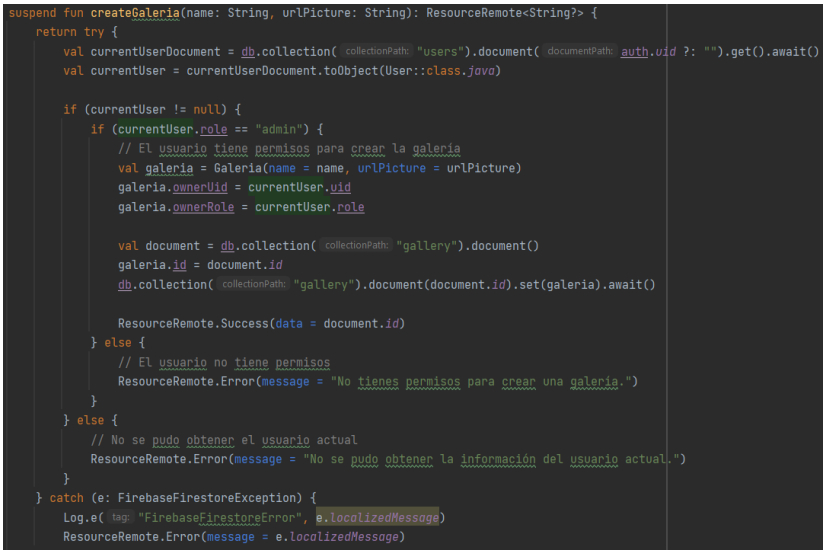
```
class GaleriaRepository {

    private var db = Firebase.firestore
    private var auth: FirebaseAuth = Firebase.auth

    // Función para crear una galeria en Firebase Firestore
    anamvelasco
    suspend fun createGaleria(name: String, urlPicture: String): ResourceRemote<String?> {
        return try {
            val currentUserDocument = db.collection( collectionPath: "users").document( documentPath: auth.uid ?: "").get().await()
            val currentUser = currentUserDocument.toObject(User::class.java)

            if (currentUser != null) {
                if (currentUser.role == "admin") {
                    // El usuario tiene permisos para crear la galeria
                    val galeria = Galeria(name = name, urlPicture = urlPicture)
                    galeria.ownerUid = currentUser.uid
                    galeria.ownerRole = currentUser.role

                    val document = db.collection( collectionPath: "gallery").document()
                    galeria.id = document.id
                    db.collection( collectionPath: "gallery").document(document.id).set(galeria).await()

                    ResourceRemote.Success(data = document.id)
                } else {
                    // El usuario no tiene permisos
                    ResourceRemote.Error(message = "No tienes permisos para crear una galeria.")
                }
```

Figura 32. Documento "GaleriaRepository.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **createGaleria:** Esta función permite la creación de una nueva galería en Firestore. Se debe verificar el rol del usuario actual para asegurar que solo los administradores puedan crear galerías. Posteriormente, se crea una instancia de la clase Galeria con los detalles proporcionados, se asigna el propietario y se genera un ID único para la

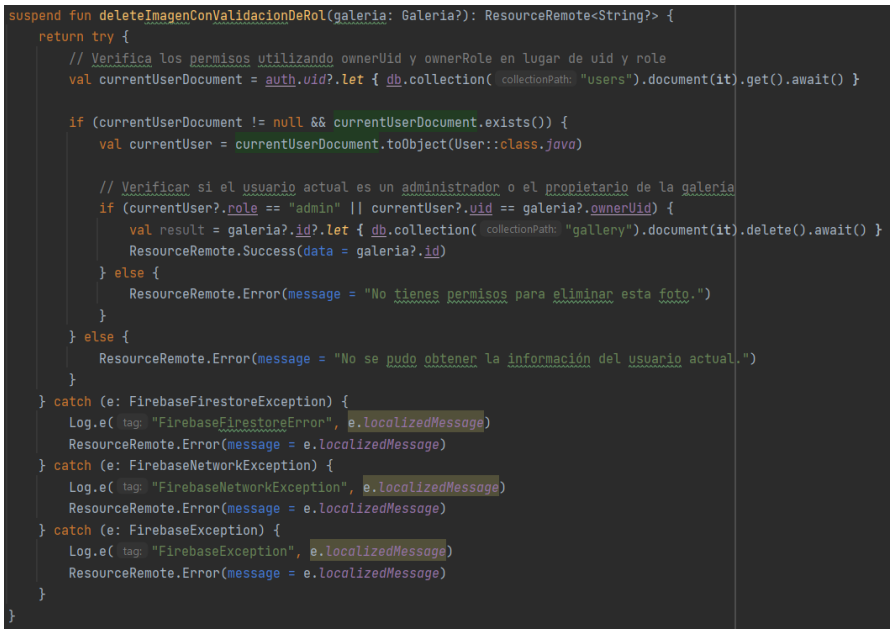
galería en Firestore. En caso de éxito, retorna el ID de la galería recién creada. En situaciones de error, devuelve un mensaje descriptivo indicando el fallo.



```
suspend fun createGaleria(name: String, urlPicture: String): ResourceRemote<String?> {
    return try {
        val currentUserDocument = db.collection( collectionPath: "users").document( documentPath: auth.uid ?: "").get().await()
        val currentUser = currentUserDocument.toObject(User::class.java)

        if (currentUser != null) {
            if (currentUser.role == "admin") {
                // El usuario tiene permisos para crear la galería
                val galeria = Galeria(name = name, urlPicture = urlPicture)
                galeria.ownerUid = currentUser.uid
                galeria.ownerRole = currentUser.role

                val document = db.collection( collectionPath: "gallery").document()
                galeria.id = document.id
                db.collection( collectionPath: "gallery").document(document.id).set(galeria).await()

                ResourceRemote.Success(data = document.id)
            } else {
                // El usuario no tiene permisos
                ResourceRemote.Error(message = "No tienes permisos para crear una galería.")
            }
        } else {
            // No se pudo obtener el usuario actual
            ResourceRemote.Error(message = "No se pudo obtener la información del usuario actual.")
        }
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
```

Figura 33. Función "createGaleria"

- **deleteImagenConValidacionDeRol:** Esta función es la encargada de eliminar una galería en Firestore. Se verifica si el usuario tiene los permisos necesarios para llevar a cabo la eliminación debido a que solo los administradores o el propietario de la galería puedan realizar esta acción. Si los permisos son válidos, se procede a eliminar la galería de Firestore. En caso de algún inconveniente durante el proceso se retorna un mensaje detallado sobre el error ocurrido.



```
suspend fun deleteImagenConValidacionDeRol(galeria: Galeria?): ResourceRemote<String?> {
    return try {
        // Verifica los permisos utilizando ownerUid y ownerRole en lugar de uid y role
        val currentUserDocument = auth.uid?.let { db.collection( collectionPath: "users").document(it).get().await() }

        if (currentUserDocument != null && currentUserDocument.exists()) {
            val currentUser = currentUserDocument.toObject(User::class.java)

            // Verificar si el usuario actual es un administrador o el propietario de la galeria
            if (currentUser?.role == "admin" || currentUser?.uid == galeria?.ownerUid) {
                val result = galeria?.id?.let { db.collection( collectionPath: "gallery").document(it).delete().await() }
                ResourceRemote.Success(data = galeria?.id)
            } else {
                ResourceRemote.Error(message = "No tienes permisos para eliminar esta foto.")
            }
        } else {
            ResourceRemote.Error(message = "No se pudo obtener la información del usuario actual.")
        }
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 34. Función "deleteImagenConValidacionDeRol"

- **loadGalery:** Esta función es la encargada de cargar la lista completa de galerías desde Firestore.

```
suspend fun loadGalery(): ResourceRemote<QuerySnapshot?> {
    return try {
        val result = db.collection( collectionPath: "gallery").get().await()
        ResourceRemote.Success(data = result)
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

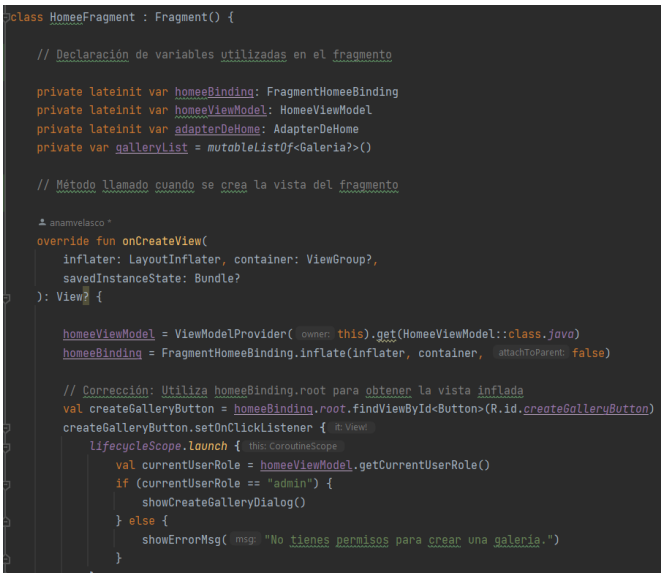
Figura 35. Función "loadGalery"

- **obtenerRolDelUsuarioActual:** Esta función es la encargada de obtener el rol del usuario actual. Se utiliza la información del usuario almacenada en Firestore para extraer su rol y se devuelve esta información como una cadena de texto. Este rol es importante para determinar si el usuario tiene permisos de administrador para crear o eliminar galerías.

```
suspend fun obtenerRolDelUsuarioActual(): String? {
    val currentUserEmail = auth.currentUser?.email
    if (currentUserEmail != null) {
        val querySnapshot = db.collection( collectionPath: "users").whereEqualTo( field: "email", currentUserEmail).get().await()
        if (!querySnapshot.isEmpty) {
            val userDocument = querySnapshot.documents[0]
            return userDocument.getString( field: "role")
        }
    }
    return null
}
```

Figura 36. Función "obtenerRolDelUsuarioActual"

El archivo *"HomeeFragment.kt"* posee un interfaz para la gestión de galerías. Este fragmento se integra en la sección de inicio de la aplicación y sirve como el espacio principal donde los usuarios, especialmente los administradores, pueden interactuar con las galerías existentes.



```
class HomeeFragment : Fragment() {

    // Declaración de variables utilizadas en el fragmento

    private lateinit var homeeBinding: FragmentHomeeBinding
    private lateinit var homeeViewModel: HomeeViewModel
    private lateinit var adapterDeHome: AdapterDeHome
    private var galleryList = mutableListOf<Galeria?>()

    // Método llamado cuando se crea la vista del fragmento

    anamvelasco *
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {

        homeeViewModel = ViewModelProvider( owner: this).get(HomeeViewModel::class.java)
        homeeBinding = FragmentHomeeBinding.inflate(inflater, container, attachToParent: false)

        // Corrección: Utiliza homeeBinding.root para obtener la vista inflada
        val createGalleryButton = homeeBinding.root.findViewById<Button>(R.id.createGalleryButton)
        createGalleryButton.setOnClickListener { it: View!
            lifecycleScope.launch { this: CoroutineScope
                val currentUserRole = homeeViewModel.getCurrentUserRole()
                if (currentUserRole == "admin") {
                    showCreateGalleryDialog()
                } else {
                    showErrorMsg( msg: "No tienes permisos para crear una galeria.")
                }
```

Figura 37. Documento "HomeeFragment.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **onCreateView:** Este método infla la interfaz de usuario asociada al fragmento, permitiendo tener botones y lista de galerías. La creación de nuevas galerías se realiza con ciertas condiciones.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {

    homeeViewModel = ViewModelProvider( owner: this).get(HomeeViewModel::class.java)
    homeeBinding = FragmentHomeeBinding.inflate(inflater, container, attachToParent: false)

    val createGalleryButton = homeeBinding.root.findViewById<Button>(R.id.createGalleryButton)
    createGalleryButton.setOnClickListener { it: View!
        lifecycleScope.launch { this: CoroutineScope
            val currentUserRole = homeeViewModel.getCurrentUserRole()
            if (currentUserRole == "admin") {
                showCreateGalleryDialog()
            } else {
                showErrorMsg( msg: "No tienes permisos para crear una galeria.")
            }
        }
    }

    // Inicialización del RecyclerView y su adaptador
    val view = homeeBinding.root
    adapterDeHome = AdapterDeHome(
        galleryList,
        onItemClicked = { onGaleriaItemClicked(it) },
        onItemLongClicked = { onGaleriaLongItemClicked(it) }
    )
```

Figura 38. Función "onCreateView"

- **showCreateGalleryDialog:** Esta función muestra un cuadro de diálogo interactivo que guía a los administradores a crear nuevas galerías. Se realizan verificaciones de roles para garantizar que solo los administradores puedan realizar esta acción.

```
private fun showCreateGalleryDialog() {
    val alertDialogBuilder = AlertDialog.Builder(requireContext())
    val inflater = LayoutInflater.from(requireContext())
    val dialogView = inflater.inflate(R.layout.dialog_create_galeria,  root: null)

    val editTextName = dialogView.findViewById<EditText>(R.id.editTextName)
    val editTextUrl = dialogView.findViewById<EditText>(R.id.editTextUrl)

    alertDialogBuilder.setView(dialogView)
    alertDialogBuilder.setTitle("Crear Nueva Galeria")

    alertDialogBuilder.setPositiveButton( text: "Crear") { dialog, _ ->
        val name = editTextName.text.toString()
        val urlPicture = editTextUrl.text.toString()

        lifecycleScope.launch { this: CoroutineScope
            if (!homeeViewModel.createGaleria(name, urlPicture)) {
                showErrorMsg( msg: "No tienes permisos para crear una galeria.")
            }
        }

        dialog.dismiss()
    }

    alertDialogBuilder.setNegativeButton( text: "Cancelar") { dialog, _ ->
        dialog.dismiss()
    }

    alertDialogBuilder.create().show()
}
```

Figura 39. Función "showCreateGalleryDialog"

- **onGaleriaLongItemClicked:** Esta función permite realizar un clic largo en una galería específica, esto es con el fin de que solo los administradores tengan el privilegio de eliminar galerías.

```
private fun onGaleriaLongItemClicked(galeria: Galeria?) {
    galeria?.let { it: Galeria
        lifecycleScope.launch { this: CoroutineScope
            // Realizar validación del rol
            val userRole = homeeViewModel.getCurrentUserRole()

            if (userRole == "admin") {
                // Si tiene permisos, mostrar el AlertDialog
                val alertDialogBuilder = AlertDialog.Builder(requireContext())
                alertDialogBuilder.setTitle("Eliminar Galeria")
                alertDialogBuilder.setMessage("¿Estás seguro de que quieres eliminar esta galeria?")

                alertDialogBuilder.setPositiveButton( text: "Si") { _, _ ->
                    // Lógica para eliminar la galeria
                    lifecycleScope.launch { this: CoroutineScope
                        homeeViewModel.deleteImagenConValidacionDeRol(galeria)
                    }
                }
                alertDialogBuilder.setNegativeButton( text: "No") { dialog, _ ->
                    dialog.dismiss()
                }
                alertDialogBuilder.create().show()
            } else {
                // Si no tiene permisos, mostrar mensaje de error
                showErrorMsg( msg: "No tienes permisos para eliminar una galeria.")
            }
        }
    }
}
```

Figura 40. Función "onGaleriaLongItemClicked"

- **loadGalery:** Se encarga de cargar y actualizar la lista de galerías en el fragmento lo que permite a los usuarios visualizar las galerías existentes.

```
homeeViewModel.loadGalery()

homeeViewModel.errorMsg.observe(viewLifecycleOwner) { msg ->
    showErrorMsg(msg)
}
homeeViewModel.galleryList.observe(viewLifecycleOwner) { galeriaList ->
    // Actualiza la lista de galerías en el adaptador y notifica los cambios
    adapterDeHome.appendItems(galeriaList)
}

return homeeBinding.root
```

Figura 41. Función "loadGalery"

El archivo *"HomeeViewModel.kt"* permite gestionar la lógica y la interacción con el repositorio de galerías, facilitando la comunicación entre la interfaz de usuario y los datos subyacentes.

```
class HomeeViewModel : ViewModel() {
    // Instancia del repositorio de la galería
    private val galeriaRepository = GaleriaRepository()
    private var galeriaListLocal = mutableListOf<Galeria?>()

    private val _errorMsg: MutableLiveData<String?> = MutableLiveData()
    val errorMsg: LiveData<String?> = _errorMsg

    private val _pictureErased: MutableLiveData<Boolean> = MutableLiveData()
    val pictureErased: LiveData<Boolean> = _pictureErased

    private val _galleryList: MutableLiveData<MutableList<Galeria?>> = MutableLiveData()
    val galleryList: MutableLiveData<MutableList<Galeria?>> = _galleryList
```

Figura 42. Documento "HomeeViewModel.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **loadGalery:** Esta función se encarga de coordinar la carga de la lista de galerías desde el repositorio, asegurando que la interfaz de usuario refleje de manera precisa la información más reciente.

```
fun loadGalery() {
    galeriaListLocal.clear()
    viewModelScope.launch { this: CoroutineScope
        val result = galeriaRepository.loadGalery()
        result.let { resourceRemote ->
            when (resourceRemote) {
                is ResourceRemote.Success -> {
                    result.data?.documents?.forEach { document ->
                        val galeria = document.toObject<Galeria>()
                        galeriaListLocal.add(galeria)
                    }
                    _galleryList.postValue(galeriaListLocal)
                }
                is ResourceRemote.Error -> {
                    val msg = result.message
                    _errorMsg.postValue(msg)
                }
                else -> {
                    //don't use
                }
            }
        }
    }
}
```

Figura 43. Función "loadGalery"

- **deleteImagenConValidacionDeRol:** Esta función se encarga de eliminar una imagen de la galería, implementa validaciones de roles para garantizar que solo los administradores puedan llevar a cabo esta acción.

```
fun deleteImagenConValidacionDeRol(galeria: Galeria?) {
    viewModelScope.launch { this: CoroutineScope
        val result = galeriaRepository.deleteImagenConValidacionDeRol(galeria)
        result.let { resourceRemote ->
            when (resourceRemote) {
                is ResourceRemote.Success -> {
                    _pictureErased.postValue( value: true)
                    _errorMsg.postValue( value: "Foto eliminada con éxito")
                }
                is ResourceRemote.Error -> {
                    val msg = result.message
                    _errorMsg.postValue(msg)
                }
                else -> {
                    // no se utiliza
                }
            }
        }
    }
}
```

Figura 44. Función "deleteImagenConValidacionDeRol"

- **createGaleria:** Esta función facilita la creación de nuevas galerías, con una validación de roles para asegurar que solo los administradores puedan realizar este proceso.

```
suspend fun createGaleria(name: String, urlPicture: String): Boolean {
    val currentUserRole = getCurrentUserRole()
    return if (currentUserRole == "admin") {
        val result = galeriaRepository.createGaleria(name, urlPicture)
        handleCreateGaleriaResult(result)
        true // Indica que la operación fue exitosa
    } else {
        _errorMsg.postValue( value: "No tienes permisos para crear una galería.")
        false // Indica que la operación no fue exitosa
    }
}
```

Figura 45. Función "createGaleria"

- **getCurrentUserRole:** Esta función proporciona el rol actual del usuario lo cual es muy importante para determinar las acciones permitidas en la interfaz de usuario.

```
suspend fun getCurrentUserRole(): String? {
    return galeriaRepository.obtenerRolDelUsuarioActual()
}
```

Figura 46. Función "getCurrentUserRole"

- **handleCreateGaleriaResult:** Esta función se encarga de gestionar los resultados de la creación de galerías teniendo en cuenta los casos de éxito y error de manera adecuada.

```
private fun handleCreateGaleriaResult(result: ResourceRemote<String?>) {
    when (result) {
        is ResourceRemote.Success -> {
            // Manejar el caso de éxito
        }
        is ResourceRemote.Error -> {
            val errorMsg = result.message
            _errorMsg.postValue(errorMsg)
        }

        else -> {
            //Nothing
        }
    }
}
```

Figura 47. Función "handleCreateGaleriaResult"

El archivo “AdapterDeHome.kt” es un adaptador que define cómo se deben mostrar las galerías en la lista permitiendo la interacción con el usuario y asegurando una representación visual ordenada.

Figura 48. Documento "AdapterDeHome.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **onCreateViewHolder:** Esta función se encarga de representar los elementos individuales en la lista de galerías. Además se define la estructura visual de cada elemento utilizando el diseño *"card_view_gallery_item.xml"*

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): HomeeViewHolder {
    val view = LayoutInflater.from(parent.context)
        .inflate(R.layout.card_view_gallery_item, parent, attachToRoot: false)
    return HomeeViewHolder(view)
}
```

Figura 49. Función "onCreateViewHolder"

- **onBindViewHolder:** En esta función se establece el contenido visual y se gestionan los eventos de clic para la interactividad del usuario.

```
override fun onBindViewHolder(holder: HomeeViewHolder, position: Int) {
    val galeria = galeryList[position]
    holder.bind(galeria)
    holder.itemView.setOnClickListener { onItemClicked(galeria) }
    holder.itemView.setOnLongClickListener { it: View!
        onItemLongClicked(galeria)
        Log.d( tag: "AdapterDeHome", msg: "Clic largo ejecutado para posición: $position")
        true ^setOnLongClickListener
    }
}
```

Figura 50. Función "onBindViewHolder"

- **getItemCount:** Esta función le indica al RecyclerView cuántos elementos debe mostrar, basándose en la cantidad de galerías disponibles.

```
override fun getItemCount(): Int = galeryList.size
```

Figura 51. Función "getItemCount"

- **appendItems:** Esta función actualiza la lista de galerías en el adaptador, permitiendo la adición de nuevas galerías y notificando al RecyclerView de los cambios.

```
fun appendItems(newList: MutableList<Galeria?>) {
    galeryList.clear()
    galeryList.addAll(newList)
    notifyDataSetChanged()
}
```

Figura 52. Función "appendItems"

El archivo *"fragment_homee.xml"* define la interfaz de usuario para mostrar y gestionar galerías. Proporciona una estructura visual organizada, la cual tiene elementos para textos, un botón y un RecyclerView que sirve como contenedor dinámico para mostrar la lista de galerías.

```
<!-- Título de la sección -->
<TextView
    android:id="@+id/textView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="GALERÍA"
    android:textAlignment="center"
    android:textColor="#026937"
    android:textSize="34sp" />

<!-- Descripción de la sección -->
<TextView
    android:id="@+id/textView2"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="En este espacio se muestran proyectos previamente reali..."
    android:textAlignment="center" />

<!-- Botón para crear una nueva galería -->
<Button
    android:id="@+id/createGalleryButton"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Crear Galería"
    android:textColor="@color/white"
    android:textColorLink="#026937" />
```

Figura 53. Documento "fragment_homee.xml"

En el archivo *"card_view_gallery_item.xml"* se define la apariencia visual de cada elemento individual dentro del RecyclerView en la sección de inicio. Este diseño se utiliza en AdapterDeHome.kt para representar cada galería.

```
<androidx.cardview.widget.CardView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    app:cardCornerRadius="8dp"
    android:longClickable="true"
    app:cardElevation="4dp">

    <!-- Contenedor principal utilizando ConstraintLayout -->
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#026937">

        <!-- ImageView para mostrar la imagen de la galería -->
        <ImageView
            android:id="@+id/gallery_image_view"
            android:layout_width="200dp"
            android:layout_height="200dp"
            android:layout_marginStart="32dp"
            android:layout_marginTop="8dp"
            android:layout_marginBottom="8dp"
            android:src="@drawable/logo"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintStart_toStartOf="parent"
```

Figura 54. Documento "card_view_gallery_item.xml"

Sección sobre nosotros

En esta sección se proporciona información sobre el Laboratorio de Prototipado y su propósito general. De esta sección existen dos archivos principales: *"AboutFragment.kt"* que representa la lógica del fragmento de la página "Acerca de" y *"fragment_about.xml"* que define la interfaz de usuario correspondiente.

En el archivo *"AboutFragment.kt"* se controla la lógica asociada a la sección "Acerca de" de la aplicación. En el método *"onCreateView"*, se infla el diseño definido en *"fragment_about.xml"*, el cual contiene la interfaz visual para esta sección.

```
class AboutFragment : Fragment() {

    anamvelasco *
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Infla el diseño XML para este fragmento
        return inflater.inflate(R.layout.fragment_about, container,  attachToRoot: false)
    }

}
```

Figura 55. Documento "AboutFragment.kt"

En el archivo *"fragment_about.xml"* se define la estructura visual de la sección "Acerca de". Se realizó un diseño de tipo ConstraintLayout para organizar los elementos en la pantalla. Además, contiene un ImageView que muestra el logotipo de la aplicación y un TextView que presenta el texto descriptivo sobre la aplicación.

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/white"
    android:textAlignment="center"
    tools:context=".ui.about.AboutFragment">

    <!-- ImageView para mostrar el logotipo -->
    <ImageView
        android:id="@+id/imageView3"
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:layout_marginStart="32dp"
        android:layout_marginTop="32dp"
        android:layout_marginEnd="32dp"
        android:src="@drawable/logo"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <!-- TextView para mostrar el texto descriptivo -->
    <TextView
        android:id="@+id/about_text_view"
        android:layout_width="0dp"
```

Figura 56. Documento "fragment_about.xml"

Sección agenda

La implementación de la sección de agenda en la aplicación es fundamental, ya que esta área permite la visualización de las reservas previamente realizadas por los usuarios a través de un calendario. Además, brinda la capacidad de eliminar las reservas, ya sea de un usuario autenticado o de manera global, en el caso del administrador. Esta sección contiene archivos que permiten su correcto funcionamiento, entre los cuales están: *"ReservasRepository.kt", "AgendaFragment.kt", "AgendaViewModel.kt", "AgendaAdapterkt", "fragment_agenda.xml" y "card_view_reserva_item.xml"*

El archivo *"ReservasRepository.kt"* actúa como un puente entre la aplicación y la base de datos Firebase Firestore. Se encuentran funciones fundamentales para la manipulación de datos relacionados con las reservas. Desde la creación hasta la eliminación de reservas además de encargarse de las operaciones de Firebase.

```
class ReservasRepository {

    private var db = Firebase.firestore
    private var auth: FirebaseAuth = Firebase.auth

    // Función para crear una reserva en Firebase Firestore
    anamvelasco
    suspend fun createReserva(reserva: Reservas): ResourceRemote<String?> {
        return try {
            val document = db.collection( collectionPath: "reservas").document()
            reserva.id = document.id
            reserva.uid = auth.uid
            db.collection( collectionPath: "reservas").document(document.id).set(reserva).await()
            ResourceRemote.Success(document.id)
```

Figura 57. Documento "ReservasRepository.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **createReserva:** Esta función se encarga de crear una nueva reserva en Firebase Firestore. Primero, genera un nuevo documento en la colección de "reservas" y asigna un identificador único a la reserva. Luego, actualiza la información de la reserva en la base de datos.

```
suspend fun createReserva(reserva: Reservas): ResourceRemote<String?> {
    return try {
        val document = db.collection( collectionPath: "reservas").document()
        reserva.id = document.id
        reserva.uid = auth.uid
        db.collection( collectionPath: "reservas").document(document.id).set(reserva).await()
        ResourceRemote.Success(document.id)

    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 58. Función "createReserva"

- **getReservasByUserEmail**: Esta función permite recuperar las reservas asociadas a un usuario específico mediante su dirección de correo electrónico. Realiza una consulta en la colección "reservas" filtrando por el campo "email" y devuelve la lista de reservas correspondientes.

```
suspend fun getReservasByUserEmail(email: String): ResourceRemote<List<Reservas>> {
    return try {
        val querySnapshot =
            db.collection( collectionPath: "reservas").whereEqualTo( field: "email", email).get().await()

        val reservasList = querySnapshot.toObjects(Reservas::class.java)

        ResourceRemote.Success(data = reservasList)
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 59. Función "getReservasByUserEmail"

- **loadReservas:** Esta función recupera todas las reservas del usuario actual. También realiza una consulta en la colección "reservas" filtrando por el identificador único del usuario autenticado y devuelve el resultado.

```
suspend fun loadReservas(): ResourceRemote<QuerySnapshot?> {
    return try {
        val result = db.collection( collectionPath: "reservas").whereEqualTo( field: "uid",auth.uid).get().await()
        ResourceRemote.Success(data = result)
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }

}
```

Figura 60. Función "loadReservas"

- **loadReservasbyDate:** Esta función obtiene las reservas filtradas por una fecha específica. Realiza una consulta en la colección "reservas" filtrando por el campo "date" y devuelve el resultado.

```
suspend fun loadReservasbyDate(fecha:String) : ResourceRemote<QuerySnapshot?> {
    return try {
        val result = db.collection( collectionPath: "reservas").whereEqualTo( field: "date",fecha).get().await()
        ResourceRemote.Success(data = result)
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }

}
```

Figura 61. Función "loadReservasbyDate"

- **deleteReserva:** Función encargada de eliminar una reserva específica de la base de datos. Utiliza el identificador de la reserva para buscar y eliminar el documento correspondiente en la colección "reservas".

```
suspend fun deleteReserva(reserva: Reservas?): ResourceRemote<String?> {
    return try {
        val result = reserva?.id?.let { db.collection( collectionPath: "reservas").document(it).delete().await() }
        ResourceRemote.Success(data = reserva?.id)
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 62. Función "deleteReserva"

- **deleteReservaConValidacionDeRol:** Función que permite la eliminación de una reserva con validación de roles. Antes de eliminar la reserva, verifica si el usuario actual tiene los permisos adecuados. Un administrador o el propio usuario pueden eliminar la reserva, y se verifica su rol antes de proceder.

```
suspend fun deleteReservaConValidacionDeRol(reserva: Reservas?): ResourceRemote<String?> {
    return try {
        val currentUserDocument = db.collection( collectionPath: "users").document( documentPath: auth.uid ?: "").get().await()
        val currentUser = currentUserDocument.toObject(User::class.java)

        if (currentUser != null) {
            if (currentUser.role == "admin" || currentUser.uid == reserva?.uid) {
                // El usuario tiene permisos para eliminar la reserva
                val result = reserva?.id?.let { db.collection( collectionPath: "reservas").document(it).delete().await() }
                ResourceRemote.Success(data = reserva?.id)
            } else {
                // El usuario no tiene permisos
                ResourceRemote.Error(message = "No tienes permisos para eliminar esta reserva.")
            }
        } else {
            // No se pudo obtener el usuario actual
            ResourceRemote.Error(message = "No se pudo obtener la información del usuario actual.")
        }
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 63. Función "deleteReservaConValidacionDeRol"

- **loadReservasByDateAndHour:** Recupera reservas filtradas por fecha, hora y máquina. Realiza una consulta en la colección "reservas" considerando los tres criterios mencionados y devuelve la lista de reservas correspondientes.

```
suspend fun loadReservasByDateAndHour(date: String, hour: String, maquina: String): ResourceRemote<List<Reservas>> {
    return try {
        val result = db.collection( collectionPath: "reservas") CollectionReference
            .whereEqualTo( field: "date", date) Query
            .whereEqualTo( field: "hour", hour)
            .whereEqualTo( field: "maquina", maquina)
            .get() Task<QuerySnapshot!>
            .await()

        val reservasList = result.toObjects(Reservas::class.java)
        ResourceRemote.Success(data = reservasList)
    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 64. Función "loadReservasByDateAndHour"

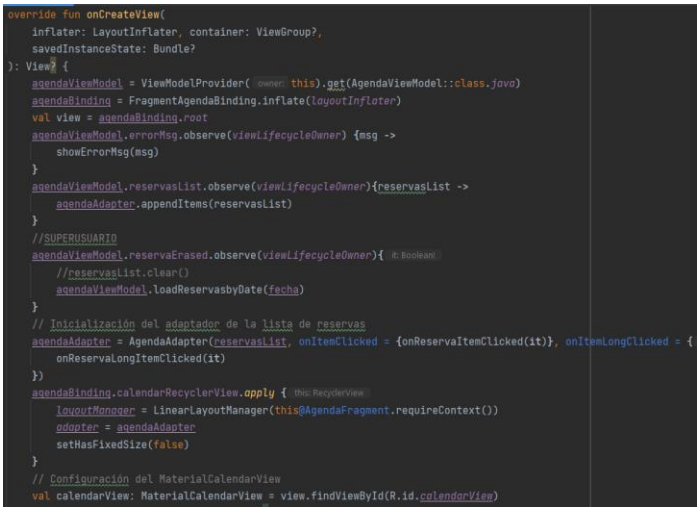
El archivo *"AgendaFragment.kt"* representa la interfaz de usuario de la sección de agenda. Este fragmento interactúa con el modelo de vista (*"AgendaViewModel.kt"*) y presenta los datos de las reservas. Al incorporar un calendario interactivo (*"calendarView"*) y una lista de reservas (*"calendarRecyclerView"*)

```
class AgendaFragment : Fragment() {

    private lateinit var agendaBinding: FragmentAgendaBinding
    private lateinit var agendaViewModel: AgendaViewModel
    private lateinit var agendaAdapter: AgendaAdapter
    private var fecha: String = ""
    private var reservasList = mutableListOf<Reservas?>()
```

Figura 65. Documento "AgendaFragment.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **onCreateView:** Esta función se ejecuta cuando el fragmento es inflado. Se inicializan instancias importantes, como el ViewModel, el Adapter para la lista de reservas y el MaterialCalendarView para la selección de fechas. También se configuran observadores para manejar actualizaciones en la interfaz de usuario.



```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    agendaViewModel = ViewModelProvider( owner: this).get(AgendaViewModel::class.java)
    agendaBinding = FragmentAgendaBinding.inflate(layoutInflater)
    val view = agendaBinding.root
    agendaViewModel.errorMsg.observe(viewLifecycleOwner) {msg ->
        showErrorMsg(msg)
    }
    agendaViewModel.reservasList.observe(viewLifecycleOwner){reservasList ->
        agendaAdapter.appendItems(reservasList)
    }
    //SUPERUSUARIO
    agendaViewModel.reservaErased.observe(viewLifecycleOwner){ it: Boolean
        //reservasList.clear()
        agendaViewModel.loadReservasbyDate(fecha)
    }
    // Inicialización del adaptador de la lista de reservas
    agendaAdapter = AgendaAdapter(reservasList, onItemClicked = {onReservaItemClicked(it)}, onItemLongClicked = {
        onReservaLongItemClicked(it)
    })
    agendaBinding.calendarRecyclerView.apply { this: RecyclerView
        layoutManager = LinearLayoutManager(this@AgendaFragment.requireContext())
        adapter = agendaAdapter
        setHasFixedSize(false)
    }
    // Configuración del MaterialCalendarView
    val calendarView: MaterialCalendarView = view.findViewById(R.id.calendarView)
```

Figura 66. Función "onCreateView"

- **showErrorMsg:** Es un método utilizado para mostrar mensajes de error en forma de notificaciones Toast. Estos mensajes pueden informar al usuario sobre posibles problemas durante la ejecución.

```
private fun showErrorMsg(msg: String?) {
    Toast.makeText(requireActivity(), msg, Toast.LENGTH_LONG).show()
}
```

Figura 67. Función "showErrorMsg"

- **onReservaLongItemClicked:** Es un método que se activa cuando se mantiene presionado un elemento de la lista de reservas. Se encarga de iniciar el proceso de eliminación de la reserva, invocando la correspondiente función del ViewModel.

```
private fun onReservaLongItemClicked(reserva: Reservas?) {
    agendaViewModel.deleteReservaConValidacionDeRol(reserva)
}
```

Figura 68. Función "onReservaLongItemClicked"

- **calendarView.setOnDateChangedListener:** Establece un escuchador para la selección de fechas en el MaterialCalendarView. Esto permite que cuando se selecciona una fecha, se actualiza la variable fecha y se inicia la carga de las reservas correspondientes a esa fecha mediante el ViewModel.

```
calendarView.setOnDateChangedListener { widget, date, selected ->
    fecha = SimpleDateFormat( pattern: "yyyy-MM-dd", Locale.getDefault()).format(date.date)
    Log.d( tag: "Fecha", fecha)
    Log.d( tag: "Fecha Selected", selected.toString())
    // Cargar las reservas para la fecha seleccionada
    agendaViewModel.loadReservasbyDate(fecha)
}
return view
```

Figura 69. Método "calendarView.setOnDateChangedListener"

El archivo *"AgendaViewModel.kt"* actúa como intermediario entre el fragmento y el repositorio de datos. Este modelo de vista proporciona datos observables y funciones para cargar reservas según la fecha seleccionada, manejar la eliminación de reservas y gestionar posibles errores.

```
class AgendaViewModel: ViewModel() {
    private val reservasRepository = ReservasRepository()
    private var reservasListLocal = mutableListOf<Reservas?>()

    private val _errorMsg : MutableLiveData<String?> = MutableLiveData()
    val errorMsg: LiveData<String?> = _errorMsg

    private val _reservaErased : MutableLiveData<Boolean> = MutableLiveData()
    val reservaErased : LiveData<Boolean> = _reservaErased

    private val _reservasList: MutableLiveData<MutableList<Reservas?>> = MutableLiveData()
    val reservasList: LiveData<MutableList<Reservas?>> = _reservasList
```

Figura 70. Documento "AgendaViewModel.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **loadReservasbyDate:** Es una función que carga las reservas según una fecha proporcionada. Utiliza el repositorio para realizar la consulta a Firebase y actualiza la lista de reservas local.

Figura 71. Función "loadReservasbyDate"

- **deleteReservaConValidacionDeRol:** Es una función que elimina una reserva con validación de roles. Verifica si el usuario actual tiene permisos de administrador o es el propietario de la reserva antes de realizar la eliminación.

Figura 72. Función "deleteReservaConValidacionDeRol"

En el archivo *"AgendaAdapter.kt"* se define la adaptación de datos para la presentación en la interfaz de usuario. Este adaptador se encarga de inflar las vistas de cada elemento de reserva y gestionar distintos eventos.

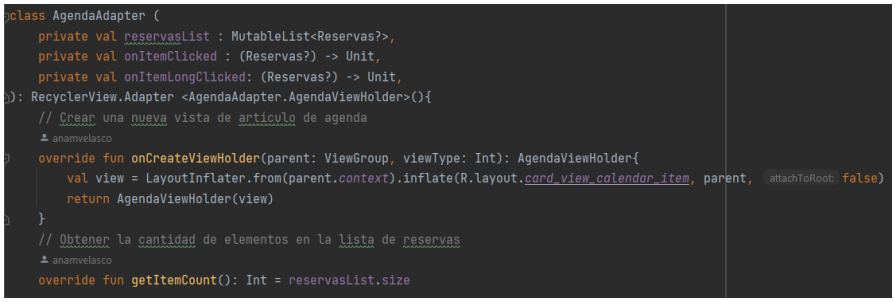


Figura 73. Documento "AgendaAdapter.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **onItemLongClicked:** Es una función que se ejecuta cuando se mantiene pulsado un elemento de la lista de reservas.

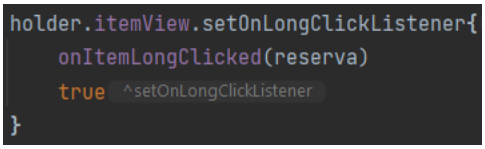


Figura 74. Función "onItemLongClicked"

- **appendItems:** Es una función que reemplaza los elementos existentes en la lista de reservas con una nueva lista proporcionada y notifica al adaptador del cambio.

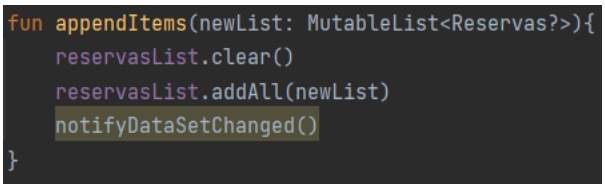


Figura 75. Función "appendItems"

- **AgendaViewHolder:** Es una clase interna que actúa como un contenedor para las vistas de un elemento de la lista de reservas.

```
class AgendaViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView){
    private val binding = CardViewCalendarItemBinding.bind(itemView)
    anamvelasco
    fun bind(reserva: Reservas?){
        with(binding){ this: CardViewCalendarItemBinding
            nameTextView.text = reserva?.name
            //emailTextView.text = reserva?.email
            //programaTextView.text = reserva?.programa
            maquinaTextView.text = reserva?.maquina
            //fechatextView.text= reserva?.date
            horatextView.text= reserva?.hour
        }
    }
}
```

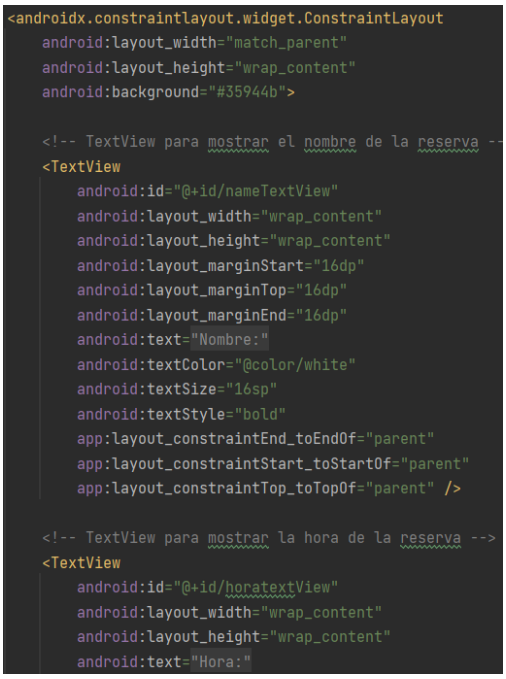
Figura 76. Clase "AgendaViewHolder"

El archivo *"fragment_agenda.xml"* tiene la estructura visual del fragmento de agenda. Se utiliza un *"MaterialCalendarView"* interactivo y un *"RecyclerView"* para mostrar las reservas.

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- Sección del calendario, utilizando la biblioteca MaterialCalendarView -->
    <com.prolificinteractive.materialcalendarview.MaterialCalendarView
        android:id="@+id/calendarView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:mcv_selectionColor="#026937" />

    <!-- RecyclerView para mostrar la lista de reservas -->
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/calendarRecyclerView"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/calendarView"
        tools:listitem="@layout/card_view_reserva_item" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Figura 77. Documento "fragment_agenda.xml"

El archivo "*card_view_calendar_item.xml"* contiene los campos de información de reserva, como nombre, correo electrónico, hora y máquina. Este archivo define la apariencia de un elemento de lista en la sección de agenda al seleccionar una fecha con reservas asociadas en el calendario.



```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="#35944b">

    <!-- TextView para mostrar el nombre de la reserva --
    <TextView
        android:id="@+id/nameTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
        android:text="Nombre:"
        android:textColor="@color/white"
        android:textSize="16sp"
        android:textStyle="bold"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <!-- TextView para mostrar la hora de la reserva -->
    <TextView
        android:id="@+id/horatextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hora:"
```

Figura 78. Documento "card_view_calendar_item.xml"

Sección reservas

La sección de reservas en la aplicación es muy importante, debido a que en esta se realiza una de las funciones más importantes de la aplicación la cual es facilitar a los usuarios el proceso de reserva de máquinas mediante un formulario. La implementación eficiente de esta sección se compone de varios archivos cruciales, incluyendo *"ReservasRepository.kt"*, *"ReservasFragment.kt", "ReservasViewModel.kt" y "fragment_reservas.xml".*

El archivo *"ReservasRepository.kt"* contiene la implementación de las funciones cruciales para gestionar las reservas en la aplicación a través de Firebase Firestore.

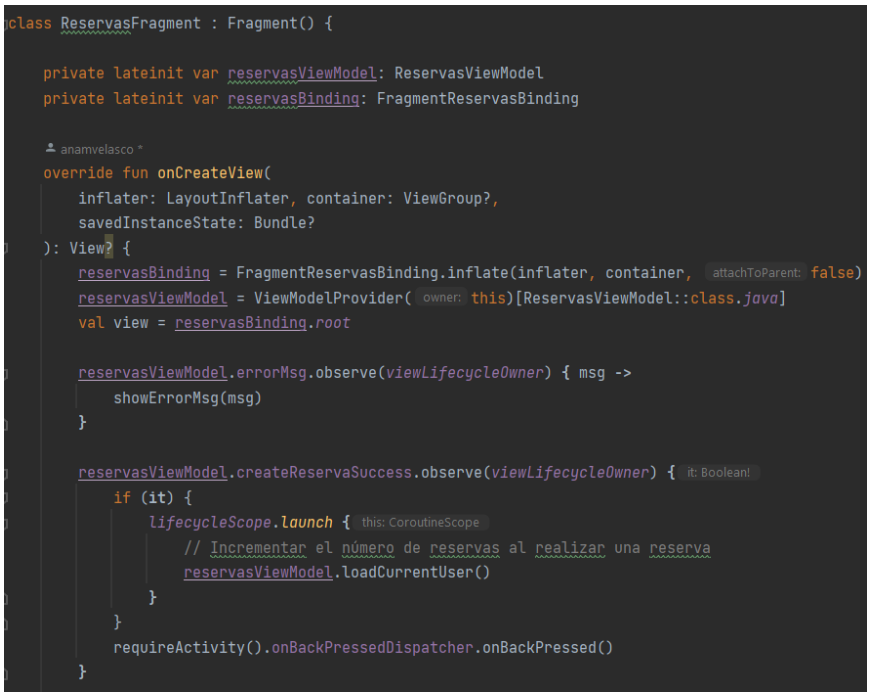
Es relevante señalar que las funciones de este archivo han sido explicadas con anterioridad. Una de las funciones destacadas en este contexto es la siguiente:

- **createReserva:** Esta función se encarga de crear una nueva reserva en Firebase Firestore. Recibe como parámetro un objeto de la clase Reservas que contiene toda la información necesaria para la reserva. El resultado de la operación se encapsula en un objeto ResourceRemote, el cual proporciona información sobre la operación realizada, incluyendo el ID de la reserva recién creada en caso de éxito.

```
suspend fun createReserva(reserva: Reservas): ResourceRemote<String?> {
    return try {
        val document = db.collection( collectionPath: "reservas").document()
        reserva.id = document.id
        reserva.uid = auth.uid
        db.collection( collectionPath: "reservas").document(document.id).set(reserva).await()
        ResourceRemote.Success(document.id)

    } catch (e: FirebaseFirestoreException) {
        Log.e( tag: "FirebaseFirestoreError", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseNetworkException) {
        Log.e( tag: "FirebaseNetworkException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    } catch (e: FirebaseException) {
        Log.e( tag: "FirebaseException", e.localizedMessage)
        ResourceRemote.Error(message = e.localizedMessage)
    }
}
```

Figura 79. Función "createReserva"

El archivo *"ReservasFragment.kt"* se encarga de la interfaz de usuario para crear nuevas reservas, ofreciendo funcionalidades como la selección de fecha y hora, elección de programas académicos y máquinas.

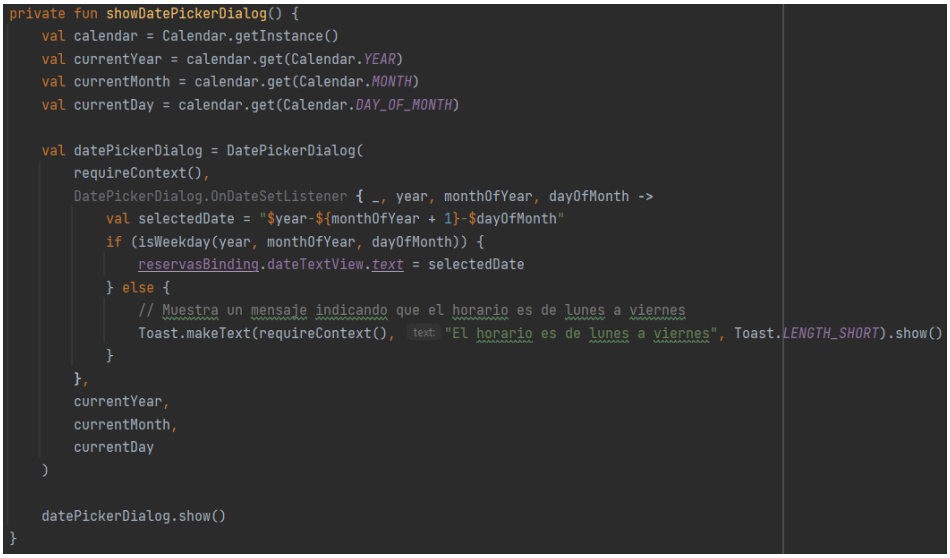


```
class ReservasFragment : Fragment() {

    private lateinit var reservasViewModel: ReservasViewModel
    private lateinit var reservasBinding: FragmentReservasBinding

    anamvelasco *
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        reservasBinding = FragmentReservasBinding.inflate(inflater, container, attachToParent: false)
        reservasViewModel = ViewModelProvider( owner: this)[ReservasViewModel::class.java]
        val view = reservasBinding.root

        reservasViewModel.errorMsg.observe(viewLifecycleOwner) { msg ->
            showErrorMsg(msg)
        }

        reservasViewModel.createReservaSuccess.observe(viewLifecycleOwner) { it: Boolean!
            if (it) {
                lifecycleScope.launch { this: CoroutineScope
                    // Incrementar el número de reservas al realizar una reserva
                    reservasViewModel.loadCurrentUser()
                }
            }
            requireActivity().onBackPressedDispatcher.onBackPressed()
        }
```

Figura 80. Documento "ReservasFragment.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

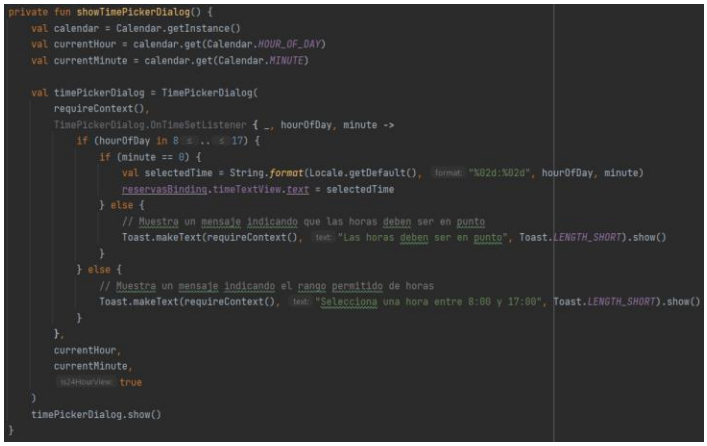
- **showDatePickerDialog:** Esta función presenta diálogo de selección de fecha, lo cual permite que los usuarios especifiquen cuándo desean realizar una reserva.



```
private fun showDatePickerDialog() {
    val calendar = Calendar.getInstance()
    val currentYear = calendar.get(Calendar.YEAR)
    val currentMonth = calendar.get(Calendar.MONTH)
    val currentDay = calendar.get(Calendar.DAY_OF_MONTH)

    val datePickerDialog = DatePickerDialog(
        requireContext(),
        DatePickerDialog.OnDateSetListener { _, year, monthOfYear, dayOfMonth ->
            val selectedDate = "$year-${monthOfYear + 1}-$dayOfMonth"
            if (isWeekday(year, monthOfYear, dayOfMonth)) {
                reservasBinding.dateTextView.text = selectedDate
            } else {
                // Muestra un mensaje indicando que el horario es de lunes a viernes
                Toast.makeText(requireContext(), text: "El horario es de lunes a viernes", Toast.LENGTH_SHORT).show()
            }
        },
        currentYear,
        currentMonth,
        currentDay
    )

    datePickerDialog.show()
}
```

Figura 81. Función "showDatePickerDialog"

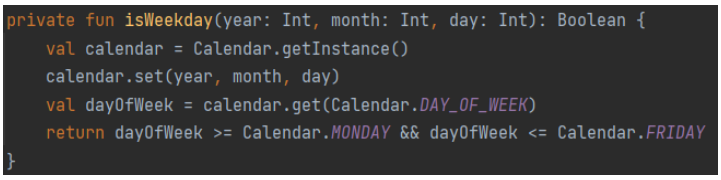
- **showTimePickerDialog:** Esta función presenta diálogo de selección de hora, lo cual permite que los usuarios especifiquen cuándo desean realizar una reserva.



```
private fun showTimePickerDialog() {
    val calendar = Calendar.getInstance()
    val currentHour = calendar.get(Calendar.HOUR_OF_DAY)
    val currentMinute = calendar.get(Calendar.MINUTE)

    val timePickerDialog = TimePickerDialog(
        requireContext(),
        TimePickerDialog.OnTimeSetListener { _, hourOfDay, minute ->
            if (hourOfDay in 8 .. 17) {
                if (minute == 0) {
                    val selectedTime = String.format(Locale.getDefault(), format: "%02d:%02d", hourOfDay, minute)
                    reservasBinding.timeTextView.text = selectedTime
                } else {
                    // Muestra un mensaje indicando que las horas deben ser en punto
                    Toast.makeText(requireContext(), text: "Las horas deben ser en punto", Toast.LENGTH_SHORT).show()
                }
            } else {
                // Muestra un mensaje indicando el rango permitido de horas
                Toast.makeText(requireContext(), text: "Selecciona una hora entre 8:00 y 17:00", Toast.LENGTH_SHORT).show()
            }
        },
        currentHour,
        currentMinute,
        is24HourView: true
    )
    timePickerDialog.show()
}
```

Figura 82. Función "showTimePickerDialog"

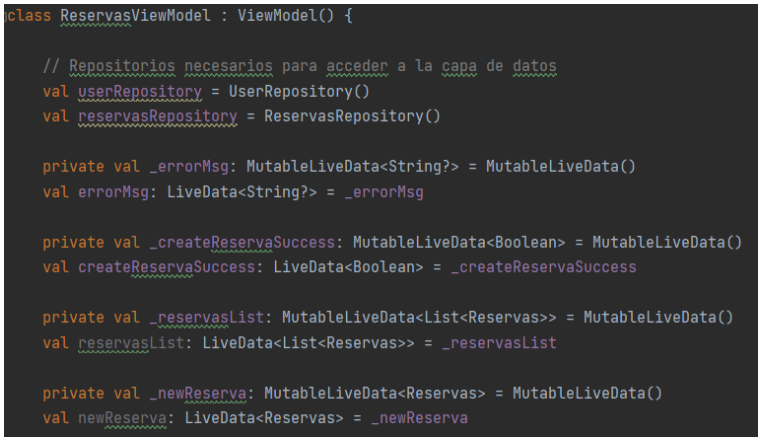
- **isWeekday:** Esta función verifica si la fecha seleccionada corresponde a un día laborable (de lunes a viernes), debido a que es una de las restricciones para la reserva.



```
private fun isWeekday(year: Int, month: Int, day: Int): Boolean {
    val calendar = Calendar.getInstance()
    calendar.set(year, month, day)
    val dayOfWeek = calendar.get(Calendar.DAY_OF_WEEK)
    return dayOfWeek >= Calendar.MONDAY && dayOfWeek <= Calendar.FRIDAY
}
```

Figura 83. Función "isWeekday"

El archivo *"ReservasViewModel.kt"* actúa como un intermediario entre la interfaz de usuario y el repositorio de reservas, facilitando la interacción con la capa de datos y garantizando una experiencia de usuario fluida y coherente.

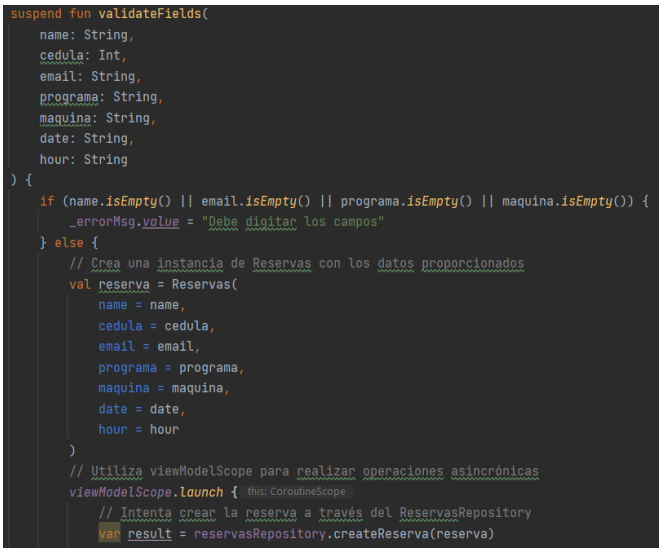


```
class ReservasViewModel : ViewModel() {

    // Repositorios necesarios para acceder a la capa de datos
    val userRepository = UserRepository()
    val reservasRepository = ReservasRepository()

    private val _errorMsg: MutableLiveData<String?> = MutableLiveData()
    val errorMsg: LiveData<String?> = _errorMsg

    private val _createReservaSuccess: MutableLiveData<Boolean> = MutableLiveData()
    val createReservaSuccess: LiveData<Boolean> = _createReservaSuccess

    private val _reservasList: MutableLiveData<List<Reservas>> = MutableLiveData()
    val reservasList: LiveData<List<Reservas>> = _reservasList

    private val _newReserva: MutableLiveData<Reservas> = MutableLiveData()
    val newReserva: LiveData<Reservas> = _newReserva
```

Figura 84. Documento "ReservasViewModel.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

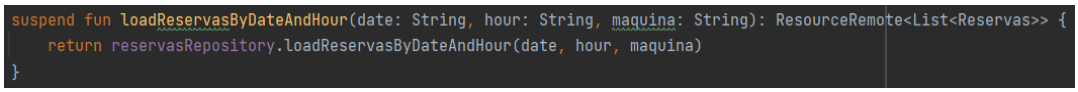
- **validateFields:** Esta función se encarga de validar los campos esenciales requeridos para realizar una reserva, como el nombre, la cédula, el correo electrónico, el programa y la máquina. En caso de campos incompletos, notifica al usuario a través de mensajes de error. Si la validación es exitosa, crea una instancia de Reservas, la almacena en Firebase Firestore a través del *"ReservasRepository"* y realiza acciones adicionales, como la actualización de la lista de reservas y la notificación de éxito.



```
suspend fun validateFields(
    name: String,
    cedula: Int,
    email: String,
    programa: String,
    maquina: String,
    date: String,
    hour: String
) {
    if (name.isEmpty() || email.isEmpty() || programa.isEmpty() || maquina.isEmpty()) {
        _errorMsg.value = "Debe digitar los campos"
    } else {
        // Crea una instancia de Reservas con los datos proporcionados
        val reserva = Reservas(
            name = name,
            cedula = cedula,
            email = email,
            programa = programa,
            maquina = maquina,
            date = date,
            hour = hour
        )
        // Utiliza viewModelScope para realizar operaciones asincrónicas
        viewModelScope.launch { this: CoroutineScope
            // Intenta crear la reserva a través del ReservasRepository
            var result = reservasRepository.createReserva(reserva)
```

Figura 85. Función "validateFields"

- **loadReservasByDateAndHour:** Esta función permite cargar reservas filtradas por fecha, hora y máquina, utilizando el método correspondiente del *"ReservasRepository"*. Además, permite verificar la disponibilidad antes de realizar una nueva reserva y garantizar la integridad del sistema de reservas.



```
suspend fun loadReservasByDateAndHour(date: String, hour: String, maquina: String): ResourceRemote<List<Reservas>> {
    return reservasRepository.loadReservasByDateAndHour(date, hour, maquina)
}
```

Figura 86. Función "loadReservasByDateAndHour"

- **loadCurrentUser:** Esta función carga la información del usuario actual desde Firebase, utilizando el *"UserRepository"*. También permite mantener actualizada la información del usuario, especialmente después de realizar una reserva, donde se actualiza el número total de reservas asociadas al usuario.

```
fun loadCurrentUser() {
    viewModelScope.launch { this: CoroutineScope
        val currentUser = FirebaseAuth.getInstance().currentUser
        currentUser?.uid?.let { uid ->
            userRepository.loadUser(uid)
        }
    }
}
```

Figura 87. Función "loadCurrentUser"

El archivo *"fragment_reservas.xml"* proporciona la interfaz de usuario para la funcionalidad de creación de reservas en la aplicación. En él, se utilizan varios componentes como TextInputLayouts, Spinners y Buttons, para recopilar información clave necesaria para la reserva. Además de tener campos de entrada, como nombre, cédula, correo electrónico, programa académico, máquina, fecha y hora.

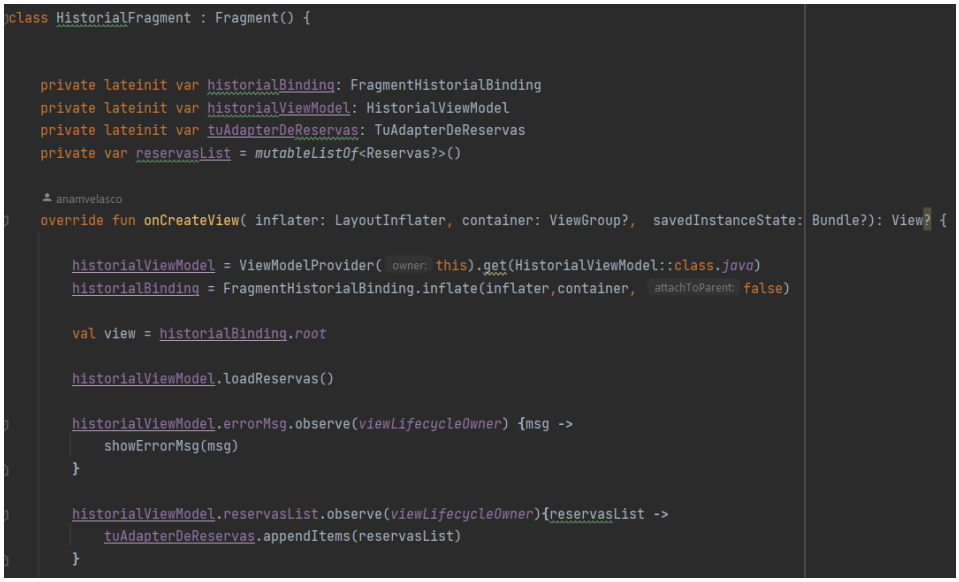
```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/white"
    tools:context=".ui.reservas.ReservasFragment">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/name_text_input_layout"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginStart="32dp"
            android:layout_marginTop="32dp"
            android:layout_marginEnd="32dp"
            android:textColorHint="@color/black"
            app:helperText="*Campo obligatorio"
            app:helperTextTextColor="@color/black"
            app:hintTextColor="@color/white"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent">
```

Figura 88. Documento "fragment_reservas.xml"

Sección historial de reservas

La sección de historial de reservas permite que los usuarios puedan visualizar sus reservas hechas previamente. La implementación eficiente de esta sección se compone de varios archivos cruciales, incluyendo *"HistorialFragment.kt", "HistorialViewModel.kt", "TuAdapterDeReservas"y "fragment_historial.xml".*

El archivo *"HistorialFragment.kt"* interactúa con el *"HistorialViewModel"* para obtener y gestionar datos, así como con un adaptador personalizado (*"TuAdapterDeReservas"*) para mostrar la información de las reservas en un formato adecuado.

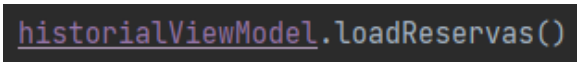


```
class HistorialFragment : Fragment() {


    private lateinit var historialBinding: FragmentHistorialBinding
    private lateinit var historialViewModel: HistorialViewModel
    private lateinit var tuAdapterDeReservas: TuAdapterDeReservas
    private var reservasList = mutableListOf<Reservas?>()

    anamvelasco
    override fun onCreateView( inflater: LayoutInflater, container: ViewGroup?,  savedInstanceState: Bundle?): View? {

        historialViewModel = ViewModelProvider( owner: this).get(HistorialViewModel::class.java)
        historialBinding = FragmentHistorialBinding.inflate(inflater,container,  attachToParent: false)

        val view = historialBinding.root

        historialViewModel.loadReservas()

        historialViewModel.errorMsg.observe(viewLifecycleOwner) {msg ->
            showErrorMsg(msg)
        }

        historialViewModel.reservasList.observe(viewLifecycleOwner){reservasList ->
            tuAdapterDeReservas.appendItems(reservasList)
        }
```

Figura 89. Documento "HistorialFragment.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **loadReservas:** Este método permite cargar las reservas y actualizar la interfaz de usuario con la información más reciente.



```
historialViewModel.loadReservas()
```

Figura 90. Método "loadReservas"

- **showErrorMsg:** Esta función permite mensajes de error al usuario mediante un Toast.

```
private fun showErrorMsg(msg: String?) {
    Toast.makeText(requireActivity(), msg, Toast.LENGTH_LONG).show()
}
```

Figura 91. Función "showErrorMsg"

El *"HistorialViewModel.kt"* despliega la lógica y la funcionalidad asociada con la sección de historial de reservas en la aplicación. Este archivo implementa la manipulación de datos y la comunicación con el repositorio de reservas para cargar y eliminar reservas. Además, se encarga de gestionar y proporcionar datos actualizados a la interfaz de usuario.

```
class HistorialViewModel : ViewModel() {

    private val reservasRepository = ReservasRepository()
    private var reservasListLocal = mutableListOf<Reservas?>()

    private val _errorMsg : MutableLiveData<String?> = MutableLiveData()
    val errorMsg: LiveData<String?> = _errorMsg

    private val _reservaErased : MutableLiveData<Boolean> = MutableLiveData()
    val reservaErased : LiveData<Boolean> = _reservaErased

    private val _reservasList: MutableLiveData<MutableList<Reservas?>> = MutableLiveData()
    val reservasList: LiveData<MutableList<Reservas?>> = _reservasList
```

Figura 92. Documento "HistorialViewModel.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **loadReservas:** Esta función se encarga de cargar la lista de reservas almacenadas. Usando el *"ReservasRepository"*, realiza una solicitud y maneja los resultados a través de LiveData. Si la operación es exitosa, las reservas se transforman y actualizan en el LiveData. En caso de error, se notifica a través del *"_errorMsg"*.

```
fun loadReservas() {
    reservasListLocal.clear()
    viewModelScope.launch { this: CoroutineScope
        val result = reservasRepository.loadReservas()
        result.let { resourceRemote ->
            when(resourceRemote){
                is ResourceRemote.Success ->{
                    result.data?.documents?.forEach{document ->
                        val reserva = document.toObject<Reservas>()
                        reservasListLocal.add(reserva)
                    }
                    _reservasList.postValue(reservasListLocal)
                }
                is ResourceRemote.Error -> {
                    val msg = result.message
                    _errorMsg.postValue(msg)


                }
                else -> {
                    //don't use
                }
            }
        }

    }

}
```

Figura 93. Función "loadReservas"

- **deleteReserva:** Esta función permite eliminar una reserva específica. Al igual que en *"loadReservas"*, utiliza el *"ReservasRepository"* para realizar la eliminación de estas.

```
fun deleteReserva(reserva: Reservas?) {
    viewModelScope.launch { this: CoroutineScope
        val result = reservasRepository.deleteReserva(reserva)
        result.let { resourceRemote ->
            when(resourceRemote){
                is ResourceRemote.Success ->{
                    _reservaErased.postValue( value: true)
                    _errorMsg.postValue( value: "Reserva eliminada con exito")
                }
                is ResourceRemote.Error -> {
                    val msg = result.message
                    _errorMsg.postValue(msg)
                }
                else -> {
                    //don't use
                }
            }
        }
    }
}
```

Figura 94. Función "deleteReserva"

El archivo *"TuAdapterDeReservas.kt"* permite la visualización de la lista de reservas en la interfaz de usuario de la sección de historial.

```
class TuAdapterDeReservas (
    private val reservasList : MutableList<Reservas?>,
    private val onItemClicked : (Reservas?) -> Unit,
    private val onItemLongClicked: (Reservas?) -> Unit,
): RecyclerView.Adapter <TuAdapterDeReservas.HistorialViewHolder>(){

    anamvelasco
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): HistorialViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.card_view_reserva_item, parent, attachToRoot: false)
        return HistorialViewHolder(view)
    }

    anamvelasco
    override fun getItemCount(): Int = reservasList.size
    anamvelasco
    override fun onBindViewHolder(holder: HistorialViewHolder, position: Int) {
        val reserva = reservasList[position]
        holder.bind(reserva)
        holder.itemView.setOnClickListener { onItemClicked(reserva)}
        holder.itemView.setOnLongClickListener{ it: View!
            onItemLongClicked(reserva)
            true ^setOnLongClickListener
        }
    }
```

Figura 95. Documento "TuAdapterDeReservas.kt"

Entre las funciones más importantes de este archivo se encuentra la siguiente:

- **appendItems:** Esta función actualiza la lista de reservas con una nueva lista proporcionada. Esto se logra mediante la limpieza de la lista actual (*"reservasList"*) y la adición de los elementos de la nueva lista. Luego, se notifica al adaptador para que actualice la interfaz de usuario.

```
fun appendItems(newList: MutableList<Reservas?>){
    reservasList.clear()
    reservasList.addAll(newList)
    notifyDataSetChanged()
}
```

Figura 96. Función "appendItems"

El archivo *"fragment_historial.xml"* proporciona la estructura visual que permite la presentación ordenada y estilizada del historial de reservas.

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/white"
    tools:context=".ui.historial.HistorialFragment">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/reservasRecyclerView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="16dp"
            tools:listitem="@layout/card_view_reserva_item">

        </androidx.recyclerview.widget.RecyclerView>
    </LinearLayout>
```

Figura 97. Documento "fragment_historial.xml"

El archivo *"card_view_reserva_item.xml"* contiene los campos de información de reserva, como nombre, correo electrónico, programa, máquina, fecha y hora.

```
<!-- Campos de texto para mostrar la información de la reserva -->
<TextView
    android:id="@+id/nameTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="16dp"
    android:text="Nombre:"
    android:textColor="@color/white"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/emailTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="16dp"
    android:text="Email:"
    android:textColor="@color/white"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/nameTextView" />
```

Figura 98. Documento "card_view_reserva_item.xml"

Sección perfil

La sección de perfil permite que los usuarios puedan visualizar su perfil con su información personal. La implementación eficiente de esta sección se compone de varios archivos los cuales son *"PerfilFragment.kt", "EditPerfilFragment.kt", "PerfilViewModel.kt", "fragment_perfil.xml"* y *"fragment_edit_perfil.xml".*

El archivo *"PerfilFragment.kt"* se encarga de la interfaz de usuario y la lógica asociada a la sección de perfil. Esta sección proporciona a los usuarios una visión detallada de su información personal almacenada en el sistema, permitiendo realizar acciones como la edición de su perfil y el cierre de sesión.

```
class PerfilFragment : Fragment() {

    private lateinit var perfilViewModel: PerfilViewModel
    private lateinit var binding: FragmentPerfilBinding
    anamvelasco
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = FragmentPerfilBinding.inflate(inflater, container,  attachToParent: false)
        return binding.root
    }
    anamvelasco
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        perfilViewModel = ViewModelProvider( owner: this).get(PerfilViewModel::class.java)

        binding.editButton.setOnClickListener {  it: View!
            abrirFragmentoEdicion()
        }
        observeViewModel()


        perfilViewModel.errorMsg.observe(viewLifecycleOwner) { msg ->
            showErrorMsg(msg)
        }
```

Figura 99. Documento "PerfilFragment.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **observeViewModel:** Esta función observa los cambios en el PerfilViewModel y actualiza la interfaz de usuario en consecuencia. Controla la visualización de datos importantes como el nombre, apellido, cédula, programa, correo electrónico y el número de reservas del usuario.

```
private fun observeViewModel() {
    perfilViewModel.errorMsg.observe(viewLifecycleOwner) { msg ->
        showErrorMsg(msg)
    }

    perfilViewModel.userData.observe(viewLifecycleOwner) { user ->
        user?.let { it: User
            binding.nameTextView.text = "Nombre: ${user.namer}"
            binding.lastNameTextView.text = "Apellido: ${user.lastnamer}"
            binding.identiTextView.text = "Cédula: ${user.identir}"
            binding.programaTextView.text = "Programa: ${user.programar}"
            binding.emailTextView.text = "Email: ${user.email}"
            binding.numreservasTextView.text = "Numero de Reservas: ${user.numReservas.toString()}"
        }
    }

    perfilViewModel.loadCurrentUser()
}
```

Figura 100. Función "observeViewModel"

- **abrirFragmentoEdicion:** Esta función navega hacia el fragmento de edición cuando el usuario desea modificar su perfil. Esta función utiliza el componente de navegación para cambiar dinámicamente entre fragmentos.

```
private fun abrirFragmentoEdicion() {
    findNavController().navigate(R.id.action_perfilFragment_to_editPerfilFragment)
}
```

Figura 101. Función "abrirFragmentoEdicion"

El archivo *"PerfilViewModel.kt"* permite realizar gestión del perfil del usuario. Este ViewModel interactúa con el repositorio de usuarios para cargar y actualizar la información del usuario, además de gestionar la funcionalidad de cierre de sesión.

```
class PerfilViewModel : ViewModel() {
    private val userRepository = UserRepository()

    private val _errorMsg: MutableLiveData<String?> = MutableLiveData()
    val errorMsg: LiveData<String?> = _errorMsg

    private val _userData: MutableLiveData<User?> = MutableLiveData()
    val userData: LiveData<User?> = _userData

    private val _userLoggedOut: MutableLiveData<Boolean> = MutableLiveData()
    val userLoggedOut: LiveData<Boolean> = _userLoggedOut
```

Figura 102. Documento "PerfilViewModel.kt"

Entre las funciones más importantes de este archivo se encuentran las siguientes:

- **loadCurrentUser:** Esta función verifica la existencia de un usuario autenticado y obtiene su UID. Posteriormente, invoca la función *"loadUser"* para cargar la información asociada al usuario.

```
fun loadCurrentUser() {
    val currentUser = FirebaseAuth.getInstance().currentUser
    currentUser?.uid?.let { uid ->
        Log.d( tag: "PerfilViewModel",  msg: "UID del usuario: $uid")
        loadUser(uid)
    } ?: run {  this: PerfilViewModel
        _errorMsg.postValue( value: "El usuario actual no tiene un UID válido.")
    }
}
```

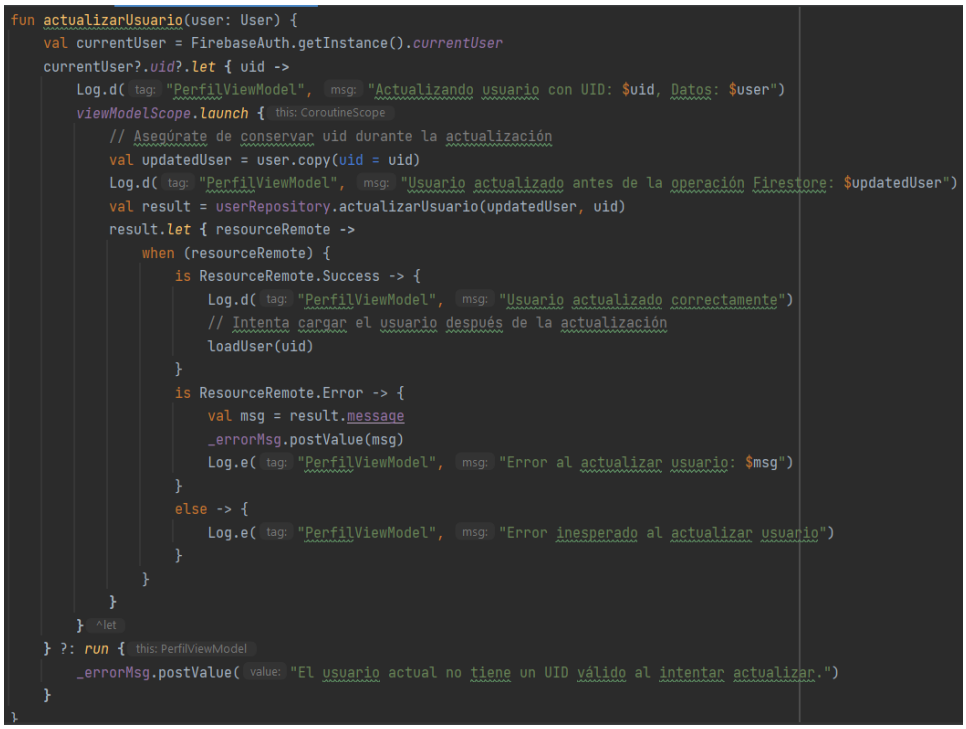
Figura 103. Función "loadCurrentUser"

- **loadUser:** Esta función utiliza el repositorio de usuarios para cargar la información del usuario actual.

```
private fun loadUser(uid: String) {
    viewModelScope.launch {  this: CoroutineScope
        val result = userRepository.loadUser(uid)
        result.let { resourceRemote ->
            when (resourceRemote) {
                is ResourceRemote.Success -> {
                    result.data?.let { user ->
                        Log.d( tag: "PerfilViewModel",  msg: "Datos del usuario después de la actualización: $user")
                        _userData.postValue(user)
                    }
                }
                is ResourceRemote.Error -> {
                    val msg = result.message
                    _errorMsg.postValue(msg)
                    Log.e( tag: "PerfilViewModel",  msg: "Error al cargar usuario después de la actualización: $msg")
                }
                else -> {
                    Log.e( tag: "PerfilViewModel",  msg: "Error inesperado al cargar usuario después de la actualización")
                }
            }
        }
    }
}
```

Figura 104. Función "loadUser"

- **actualizarUsuario:** Esta función facilita la actualización de la información del usuario. Verifica el UID del usuario actual, realiza la actualización en el repositorio y vuelve a cargar la información actualizada del usuario.



```
fun actualizarUsuario(user: User) {
    val currentUser = FirebaseAuth.getInstance().currentUser
    currentUser?.uid?.let { uid ->
        Log.d( tag: "PerfilViewModel",  msg: "Actualizando usuario con UID: $uid, Datos: $user")
        viewModelScope.launch {  this: CoroutineScope
            // Asegúrate de conservar uid durante la actualización
            val updatedUser = user.copy(uid = uid)
            Log.d( tag: "PerfilViewModel",  msg: "Usuario actualizado antes de la operación Firestore: $updatedUser")
            val result = userRepository.actualizarUsuario(updatedUser, uid)
            result.let { resourceRemote ->
                when (resourceRemote) {
                    is ResourceRemote.Success -> {
                        Log.d( tag: "PerfilViewModel",  msg: "Usuario actualizado correctamente")
                        // Intenta cargar el usuario después de la actualización
                        loadUser(uid)
                    }
                    is ResourceRemote.Error -> {
                        val msg = result.message
                        _errorMsg.postValue(msg)
                        Log.e( tag: "PerfilViewModel",  msg: "Error al actualizar usuario: $msg")
                    }
                    else -> {
                        Log.e( tag: "PerfilViewModel",  msg: "Error inesperado al actualizar usuario")
                    }
                }
            }
        }  ^let
    } ?: run {  this: PerfilViewModel
        _errorMsg.postValue( value: "El usuario actual no tiene un UID válido al intentar actualizar.")
    }
}
```

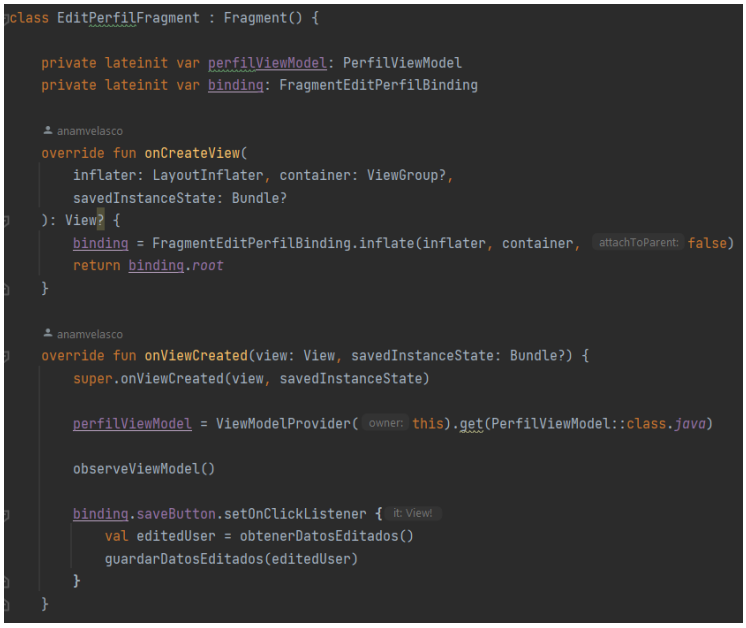
Figura 105. Función "actualizarUsuario"

- **signOut:** Esta función inicia el proceso de cierre de sesión para el usuario actual. Utiliza el repositorio de usuarios para realizar la operación de cierre de sesión.

```
fun signOut() {
    viewModelScope.launch {  this: CoroutineScope
        val result = userRepository.signOut()
        result.let { resourceRemote ->
            when (resourceRemote) {
                is ResourceRemote.Success -> {
                    if (result.data == true) {
                        _userLoggedOut.postValue( value: true)
                    }
                }
                is ResourceRemote.Error -> {
                    var msg = result.message
                    when (msg) {
                        "A network error (such as timeout, interrupted connection or unreachable host) has occurred." -> {
                            msg = "Revise su conexión de red al verificar el usuario"
                        }

                        "An internal error has occurred." -> {
                            msg = "Error interno al verificar el usuario"
                        }

                        else -> {
                            // Puedes agregar más casos según sea necesario
                        }
                    }
                    _errorMsg.postValue(msg!!)
                }
                else -> {
                    // No uses este caso si no es necesario
                }
            }
```

Figura 106. Función "signOut"

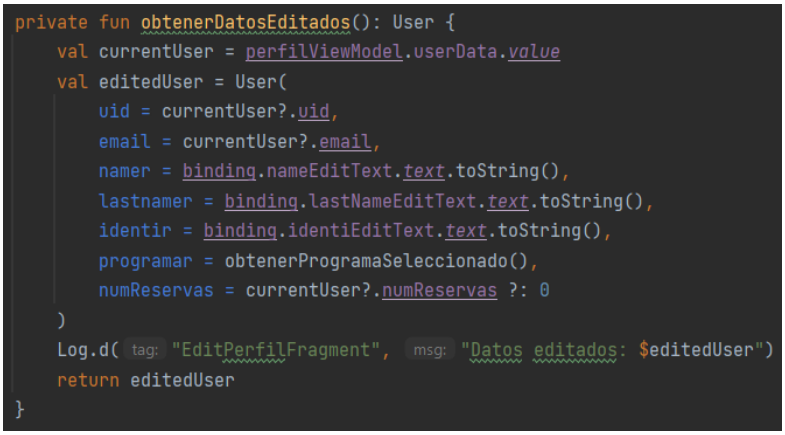
El archivo *"EditPerfilFragment.kt"* interactúa con *"PerfilViewModel"* para obtener, modificar y almacenar la información del usuario. Proporciona una interfaz de usuario para la edición de campos y la posterior actualización de datos en el perfil.



```
class EditPerfilFragment : Fragment() {

    private lateinit var perfilViewModel: PerfilViewModel
    private lateinit var binding: FragmentEditPerfilBinding

    anamvelasco
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = FragmentEditPerfilBinding.inflate(inflater, container, attachToParent: false)
        return binding.root
    }

    anamvelasco
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        perfilViewModel = ViewModelProvider( owner: this).get(PerfilViewModel::class.java)

        observeViewModel()

        binding.saveButton.setOnClickListener { it: View!
            val editedUser = obtenerDatosEditados()
            guardarDatosEditados(editedUser)
        }
    }
```

Figura 107. Documento "EditPerfilFragment.kt"

- **obtenerDatosEditados:** Esta función obtiene los datos editados por el usuario desde los campos de entrada de texto y elige el programa seleccionado. Retorna un objeto User con los datos editados.



```
private fun obtenerDatosEditados(): User {
    val currentUser = perfilViewModel.userData.value
    val editedUser = User(
        uid = currentUser?.uid,
        email = currentUser?.email,
        namer = binding.nameEditText.text.toString(),
        lastnamer = binding.lastNameEditText.text.toString(),
        identir = binding.identiEditText.text.toString(),
        programar = obtenerProgramaSeleccionado(),
        numReservas = currentUser?.numReservas ?: 0
    )
    Log.d( tag: "EditPerfilFragment", msg: "Datos editados: $editedUser")
    return editedUser
}
```

Figura 108. Función "obtenerDatosEditados"

- **obtenerProgramaSeleccionado:** Esta función recupera el programa seleccionado por el usuario desde el spinner de opciones en la interfaz.

```
private fun obtenerProgramaSeleccionado(): String {
    return binding.programaSpinnerEdit.selectedItem.toString()
}
```

Figura 109. Función "obtenerProgramaSeleccionado"

- **guardarDatosEditados:** Esta función utiliza *"PerfilViewModel"* para actualizar la información del usuario con los datos editados. Posteriormente, navega hacia atrás en la pila de fragmentos.

```
private fun guardarDatosEditados(user: User) {
    Log.d( tag: "EditPerfilFragment",  msg: "Guardando datos editados: $user")
    perfilViewModel.actualizarUsuario(user)
    // Navegar hacia atrás en la pila de fragmentos
    findNavController().navigateUp()
}
```

Figura 110. Función "guardarDatosEditados"

- **actualizarInterfazUsuario:** Esta función actualiza la interfaz de usuario con los datos del usuario proporcionados, permitiendo que el usuario visualice la información actual al abrir la pantalla de edición.

```
private fun actualizarInterfazUsuario(user: User) {
    binding.nameEditText.setText(user.namer)
    binding.lastNameEditText.setText(user.lastnamer)
    binding.identiEditText.setText(user.identir)
    binding.programaSpinnerEdit.setSelection(obtenerIndicePrograma(user.programar))
}
```

Figura 111. Función "actualizarInterfazUsuario"

- **obtenerIndicePrograma:** Esta función obtiene el índice del programa seleccionado en el spinner, útil para establecer la selección al cargar la interfaz de usuario.

```
private fun obtenerIndicePrograma(programa: String?): Int {
    val programaOptions = resources.getStringArray(R.array.programa_options)
    return programaOptions.indexOf(programa)
}
```

Figura 112. Función "obtenerIndicePrograma"

El archivo *"fragment_perfil.xml"* define la interfaz de usuario para la pantalla de perfil en la aplicación. Además, contiene elementos para mostrar información del usuario, y dos botones para permitir la edición del perfil o cerrar sesión.

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/white"
    tools:context=".ui.perfil.PerfilFragment">


    <TextView
        android:id="@+id/name_text_view"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="32dp"
        android:layout_marginTop="32dp"
        android:layout_marginEnd="32dp"
        android:text=""
        android:textColor="@color/black"
        android:textSize="16sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```

Figura 113. Documento "fragment_perfil.xml"

En el archivo *"fragment_edit_perfil.xml"* se define la interfaz de usuario para la pantalla de edición de perfil en la aplicación. En este archivo el usuario puede ingresar o editar información personal.

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#43b649"
    tools:context=".ui.perfil.EditPerfilFragment">

    <EditText
        android:id="@+id/name_edit_text"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="32dp"
        android:layout_marginTop="32dp"
        android:layout_marginEnd="32dp"
        android:hint="Ingrese su nombre"
        android:inputType="text|textPersonName|textAutoCorrect"
        android:textColor="@color/white"
        android:textColorHint="@color/white"
        android:textSize="20sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```

Figura 114. Documento "fragment_edit_perfil.xml"

Resultados y análisis

La aplicación ha logrado implementar las funcionalidades esenciales para sus dos roles distintivos: administrador y usuario.

En primera instancia, se crea la interfaz para el registro e inicio de sesión, las cuales se pueden observar en las siguientes figuras respectivamente:

Figura 115. Sección de registro

Figura 116. Sección inicio de sesión

Posteriormente, se logra visualizar en Firebase Authenticator la creación de dicho usuario, con su respecto UID de usuario.

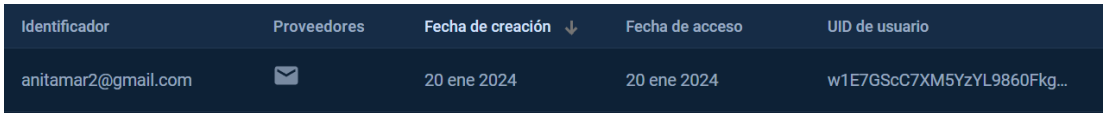


Figura 117. Usuario creado en Firebase Authentication

Los datos personales diligenciados en la sección de registro se pueden visualizar en la colección de users, la cual se encuentra en Firestore Database.

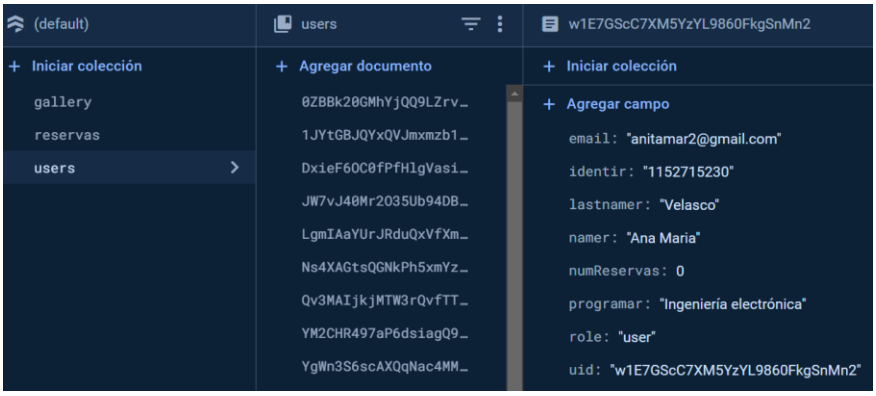


Figura 118. Colección "users"

Al inicio de la aplicación se puede observar una galería en donde se encuentran los trabajos previamente realizados en el laboratorio de prototipado, teniendo en cuenta que solo el administrador puede crear y eliminar elementos de esta galería.

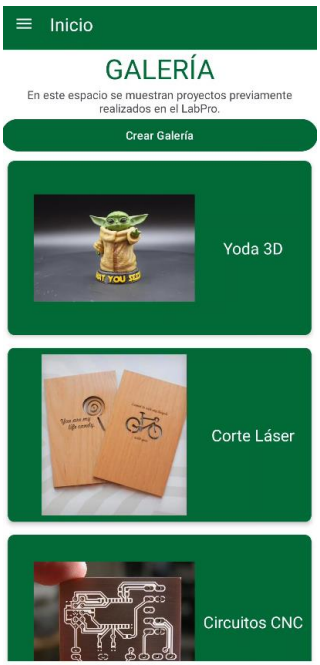


Figura 119. Sección inicio de la aplicación

Cuando se crea un nuevo elemento de la galería este se puede visualizar en la colección de gallery, la cual se encuentra en Firestore Database.

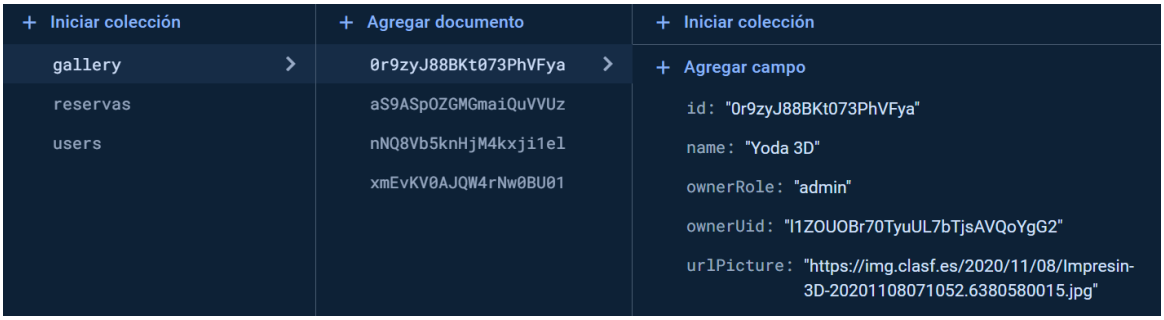


Figura 120. Colección "gallery"

El menú desplegable permite navegar en distintos espacios de la aplicación, entre estos espacios se encuentra: inicio, sobre nosotros, agenda, reservas, historial de reservas y perfil.

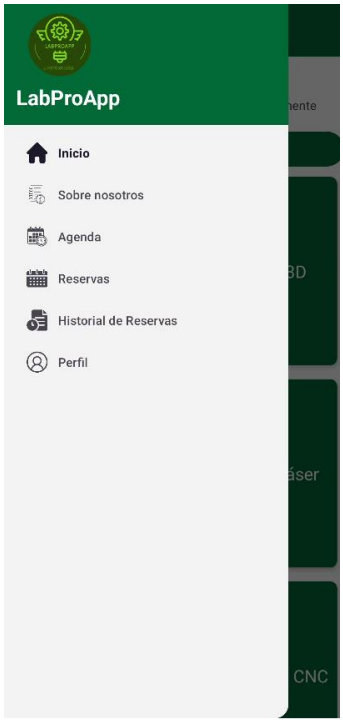


Figura 121. Menú de la aplicación

En la sección de sobre nosotros se encuentra el logo del laboratorio y un texto corto que permite conocer un poco más sobre el laboratorio.

Figura 122. Sección "sobre nosotros"

En la sección de la agenda, los usuarios tienen la capacidad de visualizar las reservas programadas para el espacio. Además, cada usuario cuenta con la opción de eliminar su propia reserva. No obstante, se destaca que el administrador posee el privilegio de eliminar cualquier reserva, brindándole un mayor control y capacidad de gestión en comparación con los usuarios regulares.

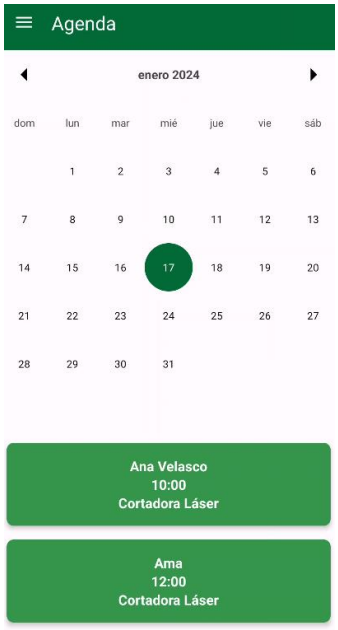


Figura 123. Sección "agenda"

En la sección de reservas, los usuarios deben completar el formulario correspondiente con la información necesaria para llevar a cabo su reserva de manera exitosa.

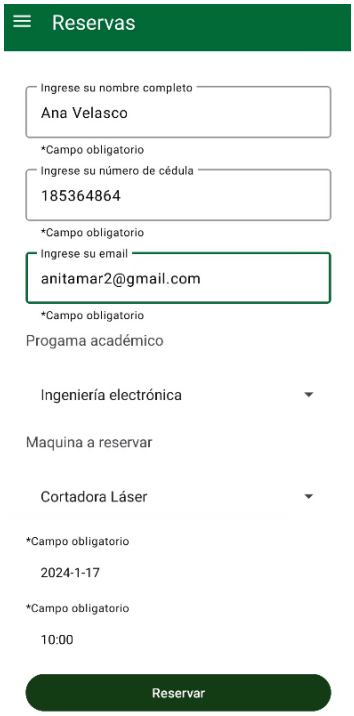


Figura 124. Sección "reservas"

Cuando se realiza una nueva reserva esta se puede visualizar en la colección de reservas, la cual se encuentra en Firestore Database.

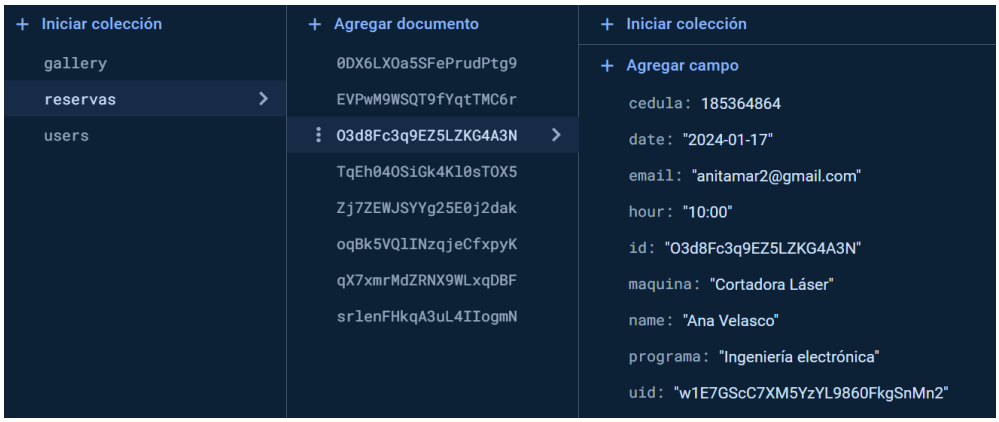


Figura 125. Colección "reservas"

En la sección de historial de reservas, los usuarios tienen la posibilidad de acceder y revisar todas las reservas que han realizado a lo largo del tiempo. Este historial proporciona una perspectiva cronológica de las actividades pasadas y futuras, permitiendo a los usuarios realizar un seguimiento efectivo de su historial de reservas en el laboratorio de prototipado.

Figura 126. Sección "historial de reservas"

En la sección del perfil de usuario se pueden visualizar los datos personales previamente diligenciados en la sección de registro, adicionando el número de reservas realizadas en el laboratorio. Esta sección tiene dos botones los cuales son “editar perfil” y “cerrar sesión”

Figura 127. Sección "perfil"

En la sección de editar perfil se pueden modificar algunos de los datos personales como lo son el nombre, apellido, programa y cédula.

Figura 128. Sección "editar perfil"

Posteriormente, se pueden visualizar estos cambios en el perfil del usuario

Figura 129. Sección "perfil" modificada

También, se pueden visualizar estos cambios en la base de datos de “users” en Firestone Database.

Figura 130. Colección "users" (datos modificados)

La interfaz desarrollada se caracteriza por ser muy intuitiva, permitiendo a los usuarios agendar citas de manera efectiva y visualizar las programadas por otros usuarios. Asimismo, la galería integrada en la interfaz proporciona un registro visual de las actividades previas realizadas en el laboratorio de prototipado.

La evaluación del rendimiento de "LabProApp" arroja resultados positivos en diversas áreas, destacándose la eficaz gestión y almacenamiento de datos, la facilidad de uso de la aplicación, así como el diseño general de la interfaz. A pesar de algunos desafíos encontrados, la aplicación demuestra su valioso aporte al ámbito de reservas y gestión de espacios en el laboratorio de prototipado de la Universidad.

Conclusiones

El presente trabajo logró alcanzar los objetivos inicialmente planteados, evidenciando un significativo avance de la aplicación “LabProApp”. Se desarrollaron las funcionalidades propuestas para el rol de administrador y el rol de usuario.

Se diseñó la interfaz de “LabProApp” de forma intuitiva, lo cual facilita a los usuarios la programación de citas y la visualización de las citas programadas por otros usuarios. Una funcionalidad adicional en esta interfaz es la galería que exhibe actividades previamente realizadas en el Laboratorio de Prototipado.

A pesar de los desafíos encontrados durante el proceso, es preciso mencionar que, la implementación de una sección donde se visualicen los requisitos necesarios para acceder al espacio y manipular las máquinas no se desarrolló debido a circunstancias externas. Este inconveniente se origina en la falta de claridad sobre dichos requisitos hasta el momento actual.

Se realizaron por parte del programador pruebas funcionales de la aplicación para evaluar el desempeño de “LabProApp” la cual arrojó resultados positivos en la capacidad de almacenamiento de datos en donde se demostró ser eficiente, respaldando la información ingresada por cada usuario. También, en la facilidad de uso en cuanto a la experiencia del usuario. El diseño general de la aplicación permite una experiencia visual agradable y coherente. Además, se proporciona el repositorio con el código fuente del proyecto, el cual se puede encontrar en el siguiente link: https://github.com/anamvelasco/LabPro

En conclusión, “LabProApp” en este contexto académico se avanza en un proceso estructurado para la gestión de reservas de citas del espacio. Aunque algunos objetivos específicos no se desarrollaron debido a circunstancias externas, los resultados positivos obtenidos en términos de funcionalidad, interfaz y evaluación general respaldan la contribución significativa de esta aplicación al ámbito de reservas y gestión de espacios en el laboratorio de prototipado de la Universidad.

Referencias

[1] Boza Rosas, J. H. (2022). Aplicativo móvil multiplataforma para la gestión de citas médicas en una clínica dental de salud, Lima 2022.

[2] Acosta Espinoza, Jorge Lenin, Lenin León Yacelga, Andrés Roberto, & Sanafria Michilena, Widman Germánico. (2022). Mobile applications and their impact on society. *Revista Universidad y Sociedad*, *14*(2), 237-243. Epub 02 de abril de 2022. Recuperado en 28 de septiembre de 2023, de http://scielo.sld.cu/scielo.php?script=sci_arttext&pid=S2218-36202022000200237&lng=es&tlng=en.

[3] Enriquez, J. G., & Casas, S. I. (2013). Usabilidad en aplicaciones móviles. *Informes científicos técnicos-UNPA*, *5*(2), 25-47.

[4] Navarro, A., & Lutty, R. (2014). Desarrollo de aplicaciones móviles.

[5] Martínez Vaca, D. A. (2021). *Estudio comparativo de las mejoras del lenguaje de programación kotlin y el lenguaje java en el desarrollo de aplicaciones android* (Bachelor's thesis, BABAHOYO: UTB, 2021).

[6] Andrade, M. A. P., Mora, A. W. C., & Robles, J. D. C. (2022). Automatización con aplicación móvil para agendamiento de cita en peluquerías. *Revista Odigos*, *3*(1), 25-47.

[7] Leyva González, L. L., & Moreno Sánchez, D. C. (2016). Diseño del sistema de gestión documental para el laboratorio de ingenieria industrial de la Universidad Cooperativa de Colombia.