

DOCUMENTACIÓN DE LA FASE DE DESARROLLO

Let's Do iT app



Autor:

Jose Enrique Jordan Moreno

Índice

Índice

1. Creación del Proyecto en Android Studio

1.1. Fichero colors.xml

1.2. Creación carpeta de recursos raw para audio

1.3. Imágenes e Icono de la app

2. Creación de la base de datos

2.1 Firestore

2.2 Conexión de Firebase con Android Studio

2.3 Servicios de Firebase

2.3.1 Autenticación

2.3.2 Storage

2.3.3 Creación de la base de datos Cloud Firestore

3. Diseño del Layout

3.1 Modificación del AndroidManifest.xml

3.2 Creación del archivo styles.xml

3.3 Creación del archivo strings.xml

3.4 Personalización del fichero themes.xml

3.5 Creación del login layout

3.6 UX Experiencia del usuario en Inicio de Sesión

4. Implementación de Login y Autenticación del Usuario

4.1 Función loginUser()

4.2 Función register()

4.3 Función goHome()

4.4 Función onStart()

4.5 Función onBackPressed()

4.6 Función resetPassword()

4.7 Función signOut()

5. Implementación de Validación de Usuario y Contraseña

5.1 Función manageButtonLogin()

5.2 Clase ValidateEmail()

5.2.1 Expresión Regular utilizada:

5.3 Clase ValidatePassword()

5.4 Función manageButtonLogin()

5.5 Manejando el foco

5.6 Función showFormatAlertDialog - Diálogo de Formato de email y contraseña

6. Implementación de Login con Google

6.1 Función signInGoogle()

6.2 Función onActivityResult()

7. Implementación del Menú

7.1 Menú Toolbar

7.2 Opciones del Menú.

- [7.2.1 Opción del Menú. Autor. Función showAutorInfoAndApps\(\)](#)
- [7.2.2 Opción del Menú. Premium. Función alertPremium\(\)](#)
- [7.2.3 Opción del Menú. Premium. Función launchPaymentCard\(\)](#)
- [7.2.4 Opción del Menú. Premium. Función becamePremium\(\)](#)
- [7.2.5 Opción del Menú. Premium. CheckoutActivity](#)
- [7.3 Implementación del Menu NavigationView](#)
- [7.4 Diseño de la Cabecera del Menú](#)
- [7.5 Añadiendo funcionalidad al menú](#)
 - [7.5.1 Función initToolbar](#)
 - [7.5.2 Función initNavigationView](#)
 - [7.5.3 Función onNavigationItemSelected](#)
- [7.6 Diseño de la pantalla del Historial](#)
 - [7.6.1 Diseño del Menú Emergente en Toolbar](#)
 - [7.6.2 Función initToolbar](#)
 - [7.6.3 Función onOptionsItemSelected](#)
- [8. Diseño del Menú Principal](#)
 - [8.1 Elementos del menú fijos](#)
 - [8.2 Marcadores](#)
 - [8.3 Cronómetro](#)
 - [8.4 Elementos del menú móviles](#)
 - [8.4.1 Mapa](#)
 - [8.4.2 Botón Start / Stop](#)
 - [8.4.3 Botón Cámara Floating Action Button](#)
 - [8.4.4 Layout Elección de Actividad](#)
 - [8.4.5 Carrera con Intervalos](#)
 - [8.4.6 Marcar Objetivo](#)
 - [8.4.7 Ajustes de Audio](#)
 - [8.4.7 Pistas de audio](#)
 - [8.5 Pantalla de Resultados PopUp](#)
- [9. Animaciones](#)
 - [9.1 ViewBinding](#)
 - [9.2 Función de cambio de atributos de un layout](#)
 - [9.3 Función de ocultación de layouts](#)
 - [9.4 Función inflateVolumes](#)
 - [9.4.1 Función animateViewofFloat y animateViewofInt](#)
 - [9.5 Función inflateChallenges](#)
 - [9.6 TranslationZ](#)
 - [9.6.1 Funciones showChallenge, showDuration, showDistance, getChallengeDuration](#)
 - [9.7 Función inflateIntervalMode](#)
 - [9.8 Control del Carrera con Intervalos.](#)
 - [9.8.1 Función initIntervalMode](#)
 - [9.8.2 Función getFormattedStopWatch](#)
 - [9.9 Control del Carrera con Intervalos.](#)
 - [9.9.1 Marcar Objetivo . Función initChallengeMode](#)

[10. Implementación del Cronómetro y sus opciones](#)

- [10.1 Objetos Handler y Runnable](#)
- [10.2 Función manageRun. Lanzar y Detener el Cronómetro](#)
- [10.3 Botón del Cronómetro. Activar/Desactivar Funciones](#)
- [10.4 Botón de Reset del Cronómetro](#)
- [10.5 Habilitar/Deshabilitar Opciones de Carrera](#)
- [10.6 Control de Intervalos](#)

[11. Implementación de la Música](#)

- [11.1 MediaPlayer](#)
- [11.2 Función initMusic. Inicialización de objetos MediaPlayer](#)
- [11.3 Función setVolumes. Control del volumen](#)
- [11.4 Función setProgressTracks. Control de pistas de audio](#)
- [11.5 Función updateTimeTracks](#)
- [11.6 Reproducción / Parar Música](#)

[12. Implementación del GPS](#)

- [12.1 Consultar y Activar GPS](#)
- [12.2 Objeto FusedLocationProviderClient](#)
- [12.3 Administración de Localización y Datos de ubicación.](#)
- [12.4 Cálculo de Distancia y Velocidad](#)

[13. Implementación de Maps de Google](#)

- [13.1 Creación del Mapa](#)
- [13.2 Control de Velocidades Admitidas](#)
- [13.3 Función selectSport](#)
- [13.4 Trazado de la Trayectoria de la actividad en el Mapa](#)

[14. Implementación de Preferencias del Usuario](#)

- [14.1 Función savePreferences. Guardar Preferencias](#)
- [14.2 Función recoveryPreferences. RecuperarPreferencias](#)
- [14.3 Restablecer Preferencias](#)

[15. Implementación de Pantalla de Resultados](#)

- [15.1 Mostrar los datos de la actividad en la pantalla de resultados](#)

[16. Implementación de la Base de Datos](#)

- [16.1 Clases de datos . Niveles y Totales.](#)
- [16.2 Instancias de lectura y escritura en la base de datos.](#)
- [16.3 Configuración de Niveles](#)
- [16.4 Carga de Datos Personales](#)
- [16.5 Establecer Records Según Deporte](#)
- [16.6 Actualizar Totales. updateTotalsUser](#)
- [16.7 Mostrar Resultados Ventana Emergente](#)
- [16.8 Mostrar Records y Medallas](#)

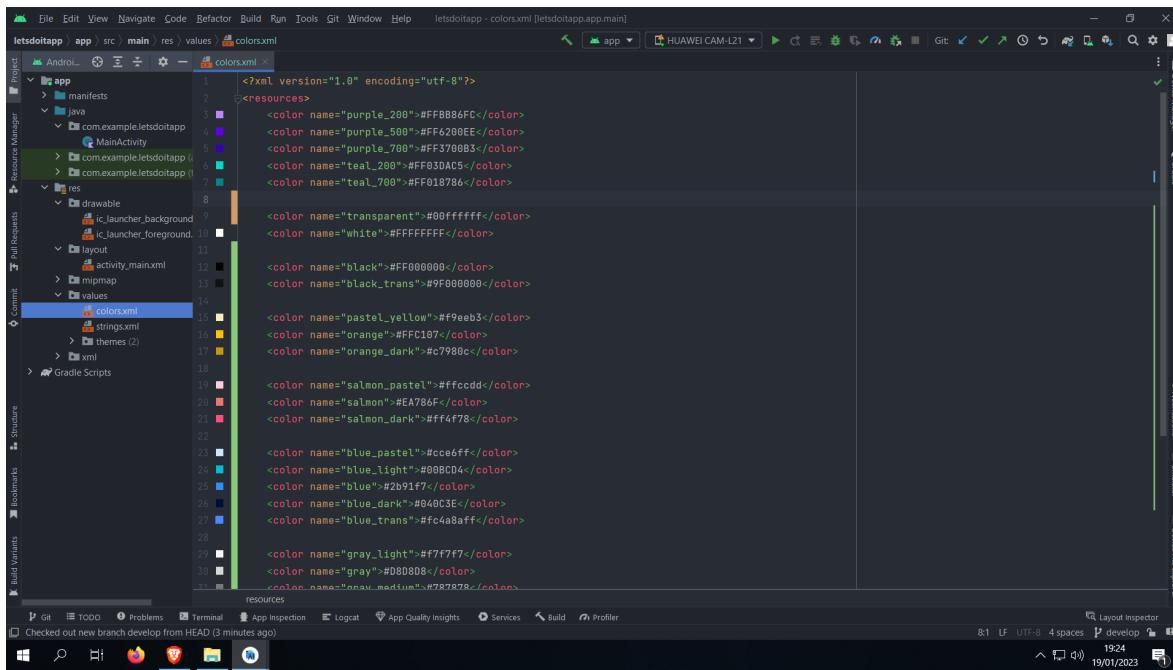
[17. Administración de la Base de Datos](#)

- [17.1 Guardar Actividad. Identificador único.](#)
- [17.2 Guardar Datos. Función saveDataRun](#)
 - [17.2.1 Problema al añadir datos con barras y puntos](#)
- [17.3 Actualizar Datos. Función update](#)

- [17.4 Consultas a la Base de Datos](#)
 - [17.4.1 Función loadMedalsBike.](#)
- [17.5 Creación de SubColecciones con Ubicaciones](#)
- [17.6 Borrado de Datos](#)
 - [17.6.1 Función deleteRun](#)
- [17.7 Actualización de los Totales](#)
- [17.8 Consulta Record de Distancia de un Usuario en particular](#)
 - [17.8.1 Creación de Indices compuestos en Firestore](#)
- [17.9 Borrado de Localizaciones](#)
- [18. Notificaciones](#)
 - [18.1 Notificación de Récord](#)
 - [18.2 Consultar y Mostrar Medallas. Función checkMedals](#)
- [19. Implementación de la Cámara](#)
 - [19.1 Importación en Gradle de las librerías de Cámara](#)
 - [19.2 Creación de la pantalla de la Cámara](#)
 - [19.3 Función TakePicture](#)
 - [19.4 Ejecución de la Cámara en un hilo separado](#)
 - [19.5 Creación de directorios externos. externalMediaDirs](#)
 - [19.6 Selector de la Cámara Delantera Trasera](#)
 - [19.7 Petición de Permisos de la Cámara](#)
 - [19.8 Iniciar Cámara. startCamara](#)
 - [19.9 Vinculación con la Cámara. bindCamara](#)
 - [19.10 Tomar Fotos y Guardar Imágenes. Función takePhoto](#)
- [20. Implementación del Historial de Actividades](#)
 - [20.1 Carga de Datos. Función loadRecyclerView](#)
 - [20.2 Función onOptionsItemSelected](#)
 - [20.3 Función onResume](#)
- [21. Almacenamiento en la nube. Firebase Storage](#)
 - [21.1 Integración de Storage](#)
 - [21.2 Subir archivos a Storage](#)
 - [21.3 Crear Metadata Personalizado](#)
 - [21.4 Descargar archivos de Storage](#)
 - [21.5 Borrado de archivos de Storage](#)
- [22. Reproducción de la actividad](#)
 - [22.1 Envío, recepción y reproducción de parámetros de Actividad](#)
- [23. Widget](#)
- [24. Pasarela de Pagos de Paypal](#)
- [25. Snackbar Personalizado](#)
- [26. Intro y Animaciones en SplashScreen](#)
- [27. Referencias](#)

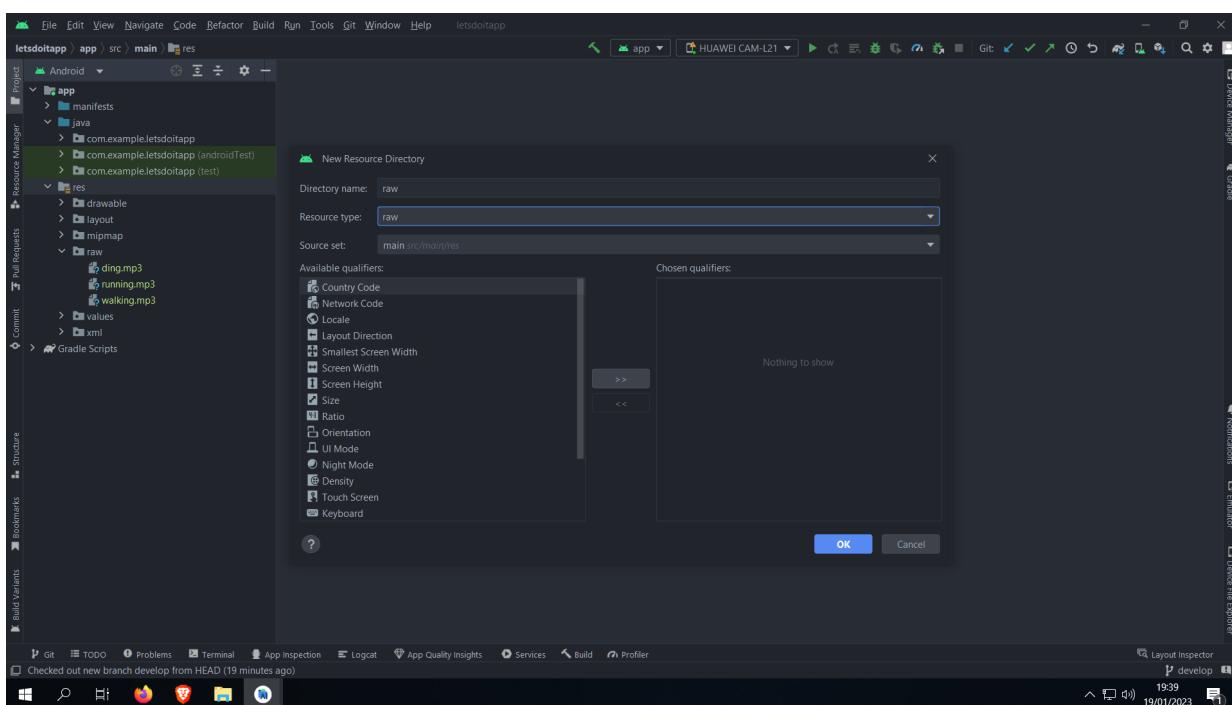
1. Creación del Proyecto en Android Studio

1.1. Fichero colors.xml



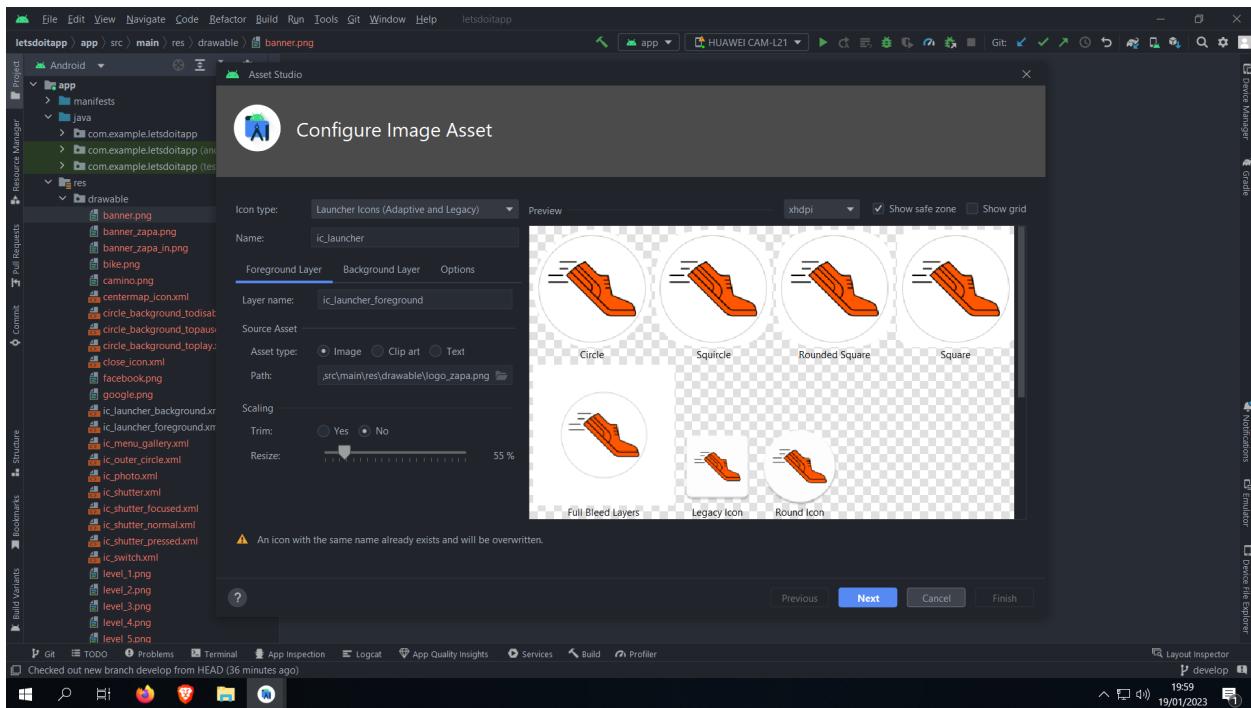
En este fichero se reúne toda la gama de colores que se va a utilizar en la app a desarrollar. Una gama de amarillos, salmon , azules ,grises.

1.2. Creación carpeta de recursos raw para audio

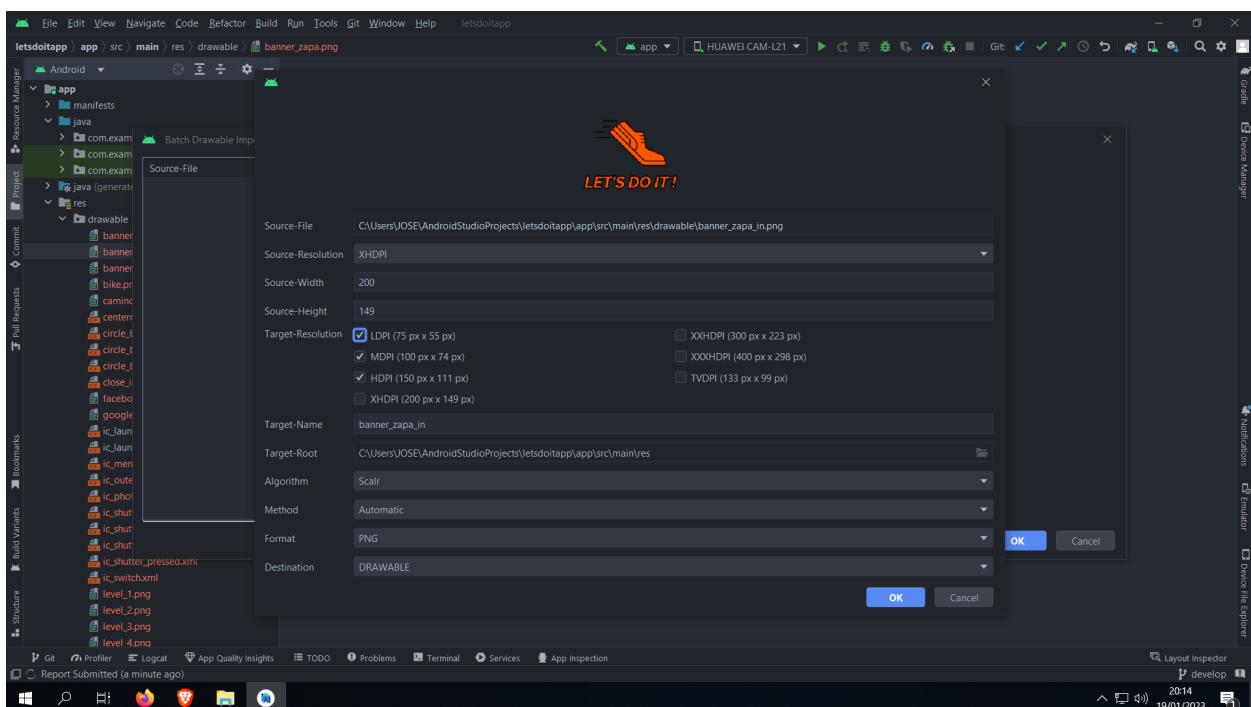


Para guardar los archivos de audio que se van a utilizar, se crea una carpeta raw dentro de recursos. Los audios son 3 archivos en formato mp3.

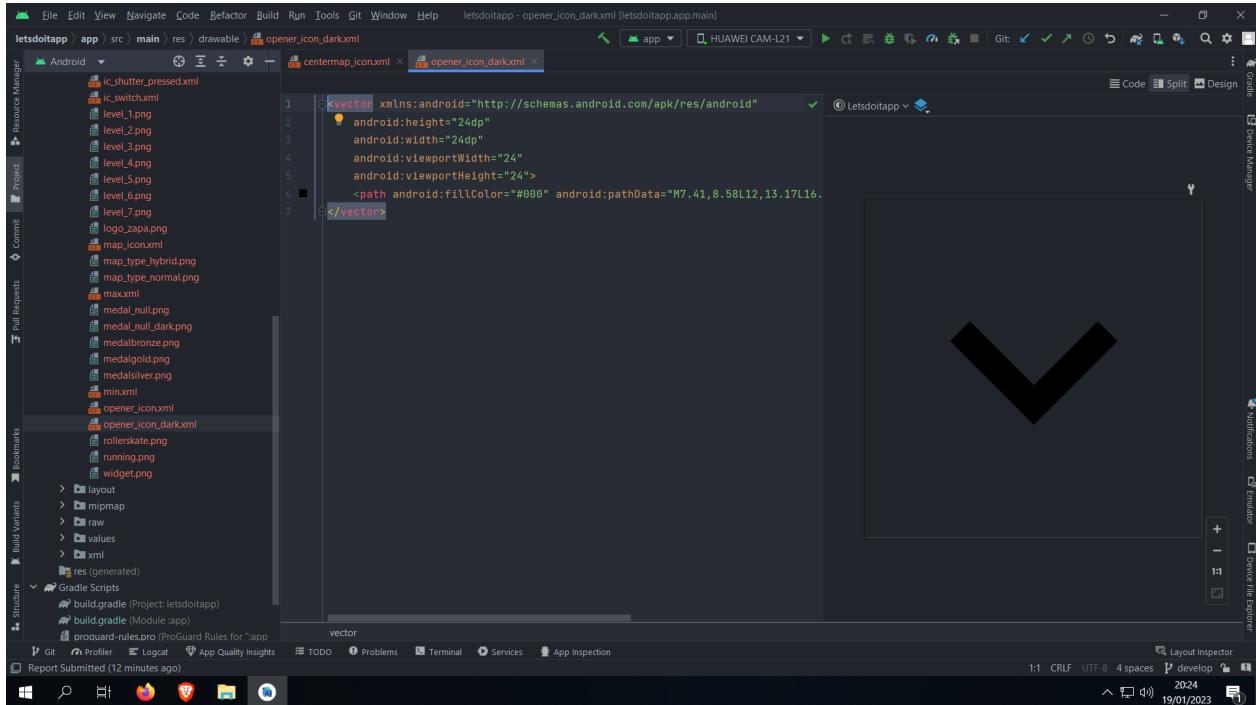
1.3. Imágenes e Icono de la app



Pantalla de Configuración del ícono. Para la creación del logo se usó la web <https://logomakr.com>



Importación de imágenes en distintas resoluciones con el plugin [Android Drawable Importer](#)



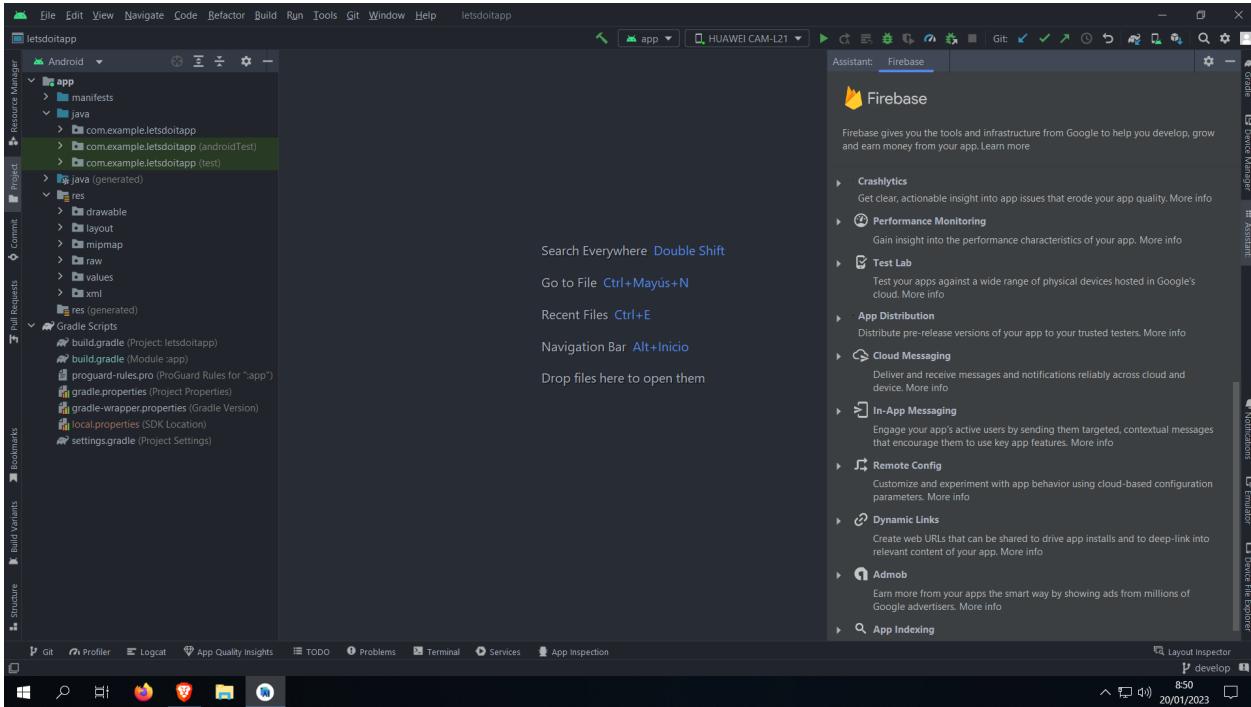
Los recursos de las imágenes vectoriales y los iconos están disponibles en materialdesignicons.com

2. Creación de la base de datos

2.1 Firestore

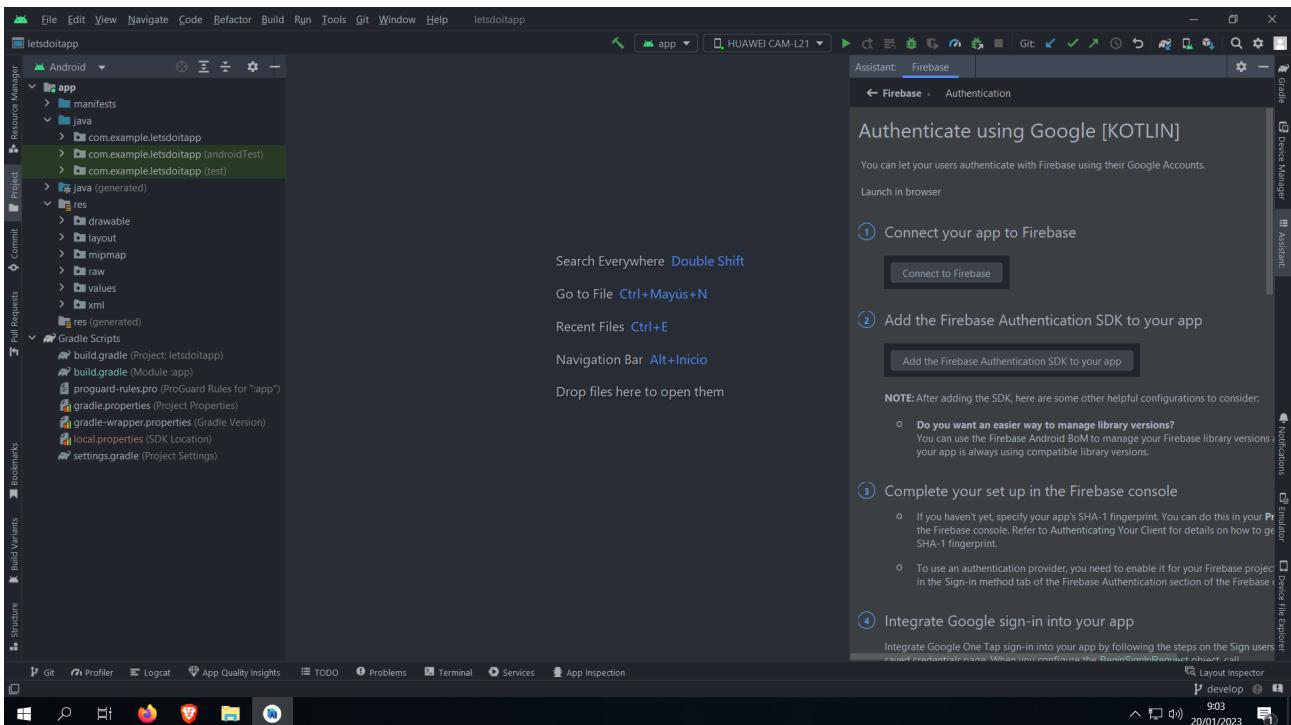
La creación de la base de datos es fundamental. Podría trabajar con una base de datos local , pero este sería un inconveniente porque ese tipo de datos solo estaría vinculado al dispositivo donde se instala la app. Si un usuario inicia sesión o pretende hacer uso de la app en otro dispositivo nuevo, todos los datos que tiene asociados a su perfil no podrían cargarse.

Será necesario guardar el usuario y la contraseña, así que siempre será necesario que haya una base de datos independiente del dispositivo. De esta manera cuando el usuario quiera hacer uso de la app en otro dispositivo simplemente iniciando sesión ya tendrá sus datos vinculados a su perfil. Evidentemente necesitaremos conexión a Internet para manejar esos datos.

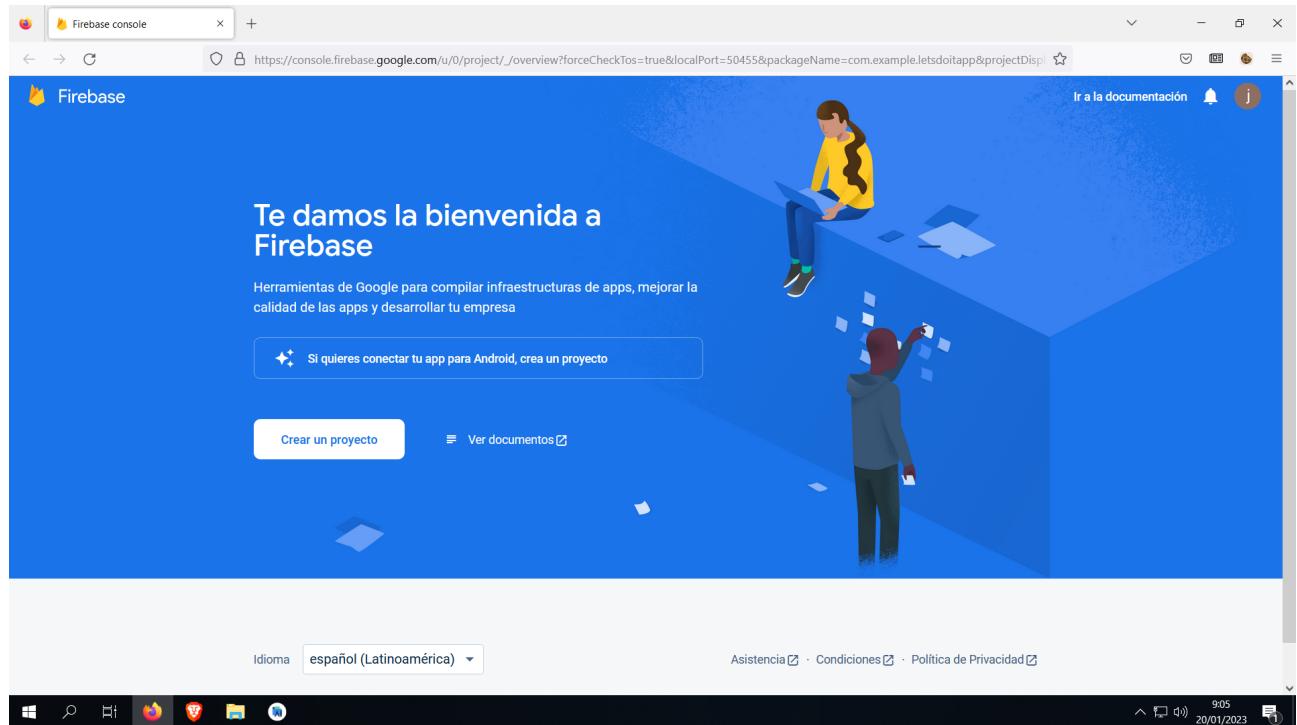


Android Studio nos ofrece la opción de configurar fácilmente una base de datos en Firebase a través de su asistente. Hay una documentación muy extensa y detallada de cómo utilizar esta base de datos para todos los servicios que vamos a utilizar en esta app. Entre los servicios más importantes tenemos **Analytics**, **Authentication**, **Cloud Storage** etc.

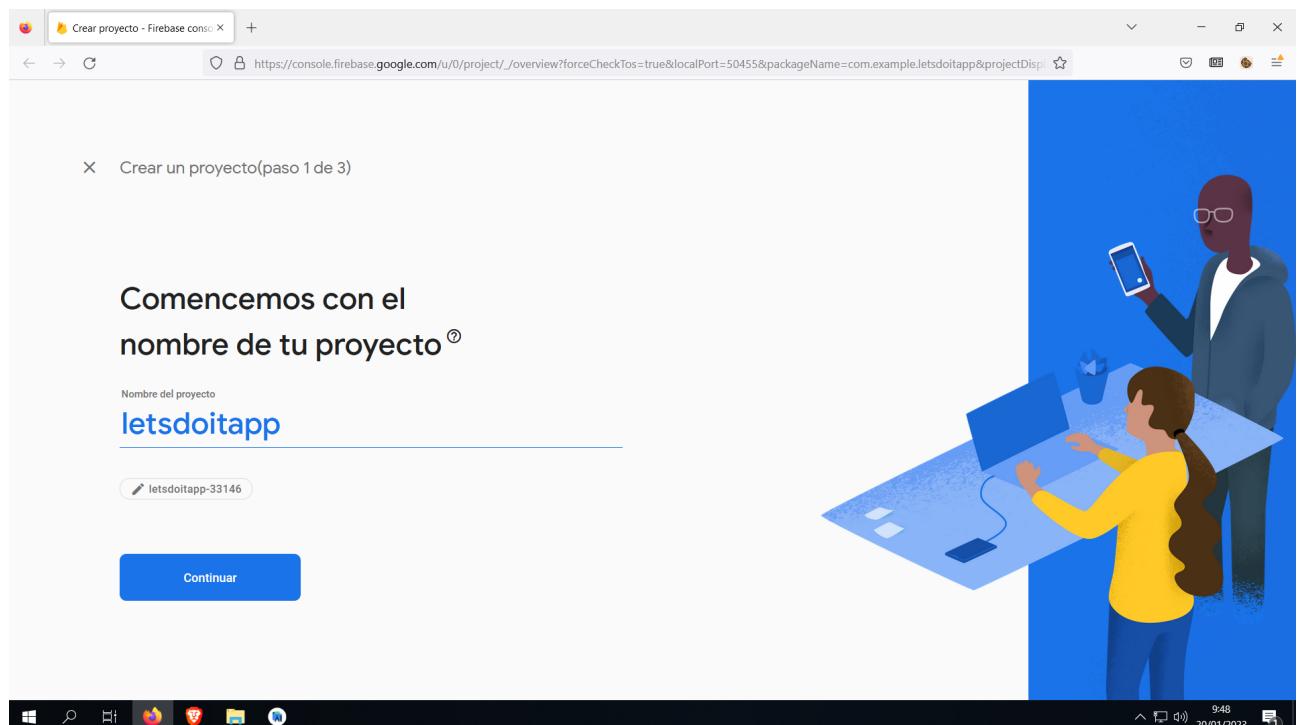
2.2 Conexión de Firebase con Android Studio



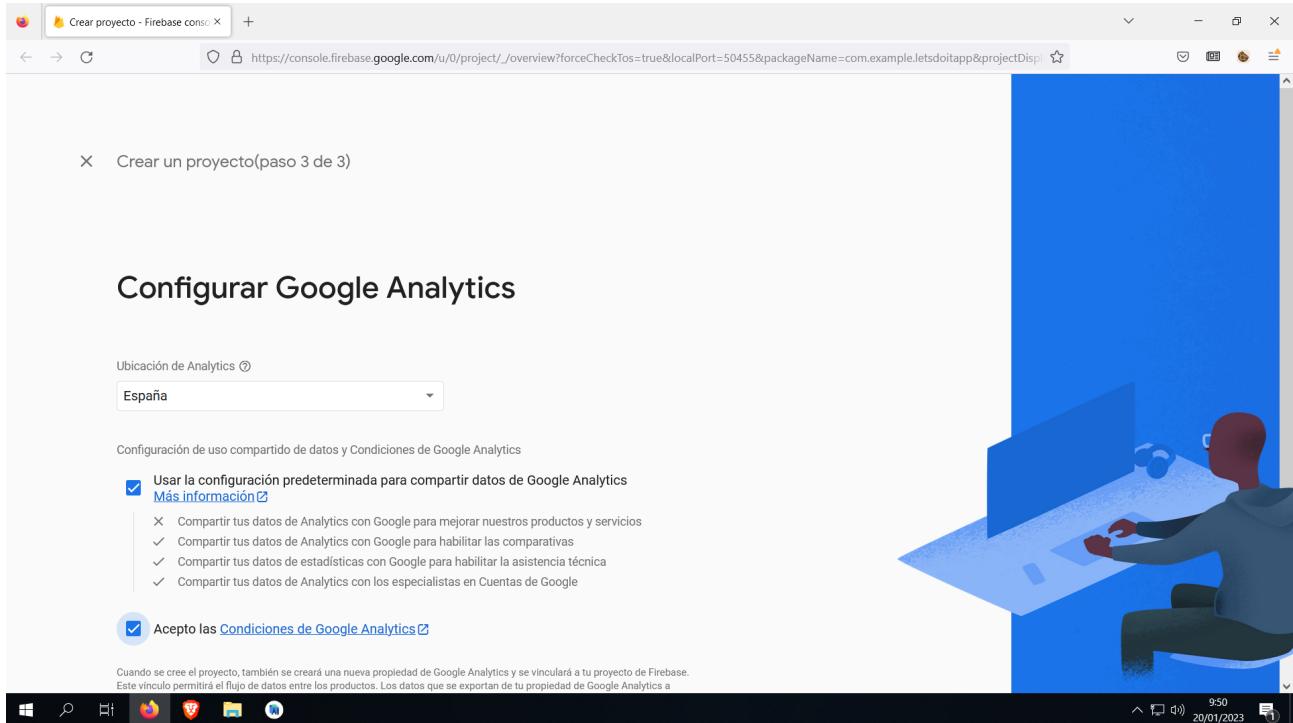
El asistente de Android Studio nos indica los pasos que tenemos que seguir para conectar nuestra app con Firestore. <https://console.firebaseio.google.com>



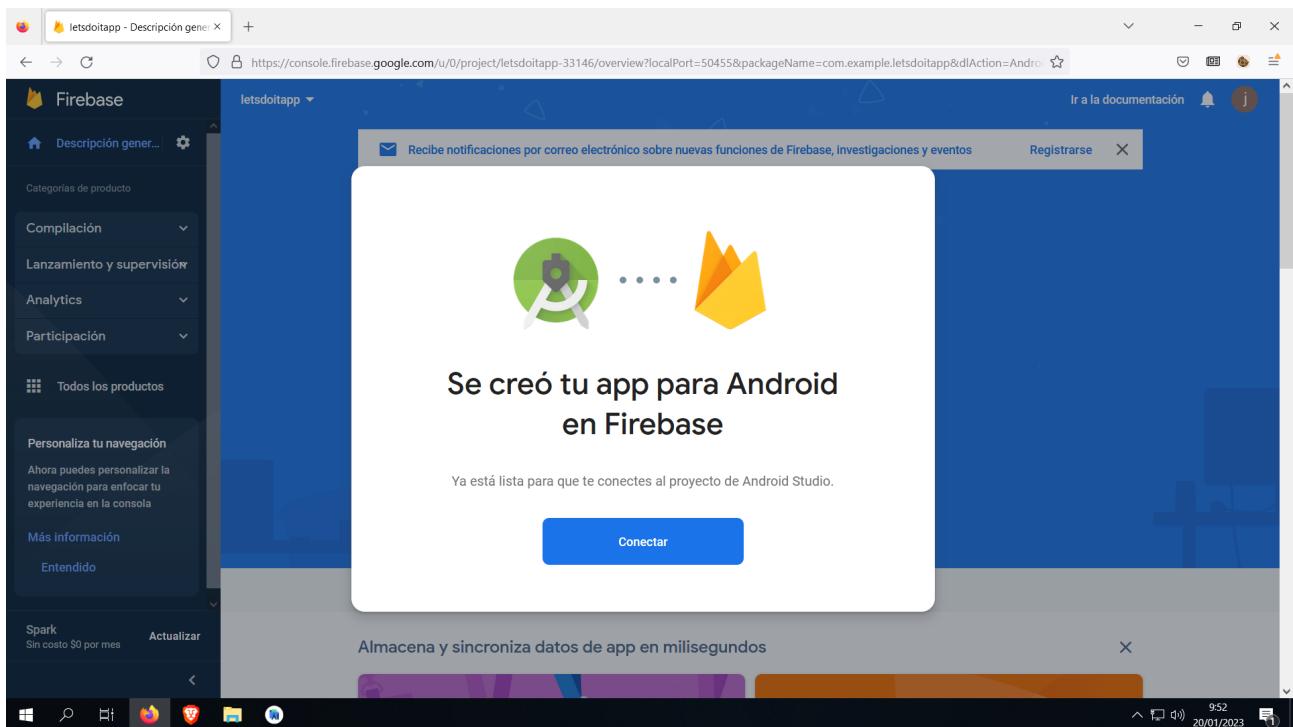
Desde la web desde donde nos da la bienvenida Firebase



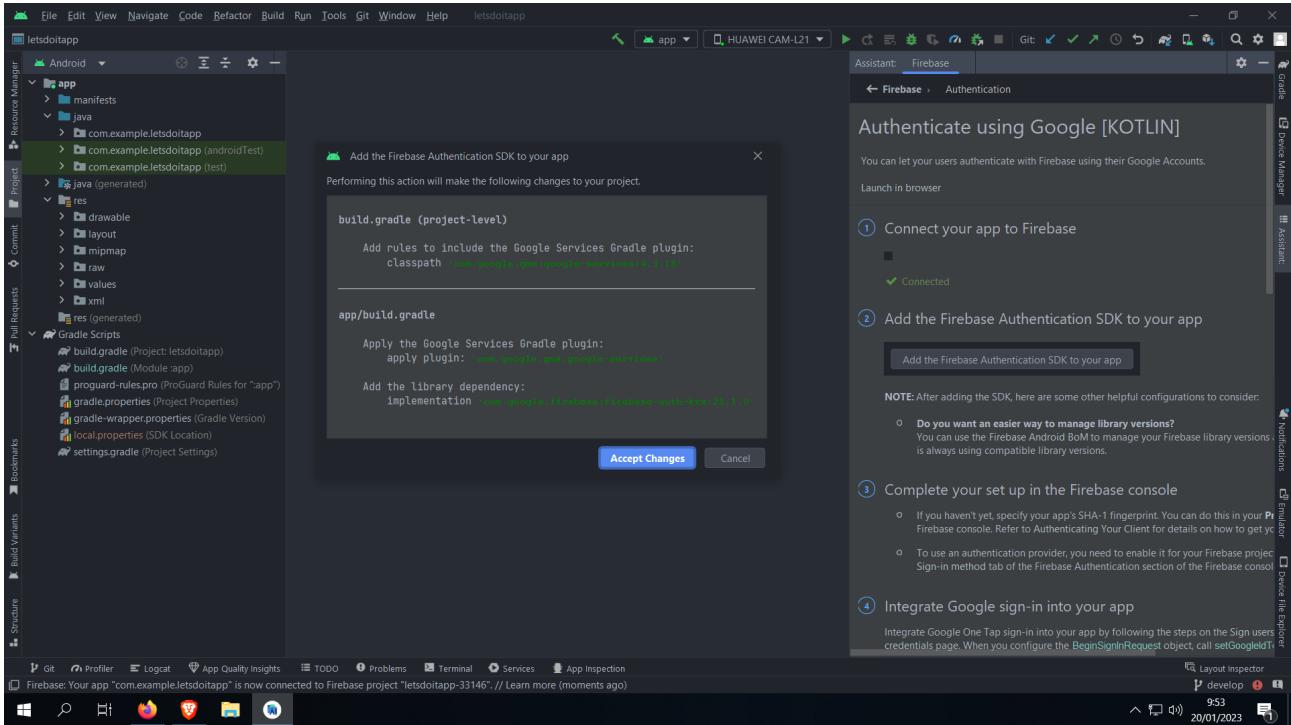
Crearemos paso a paso nuestra base de datos.



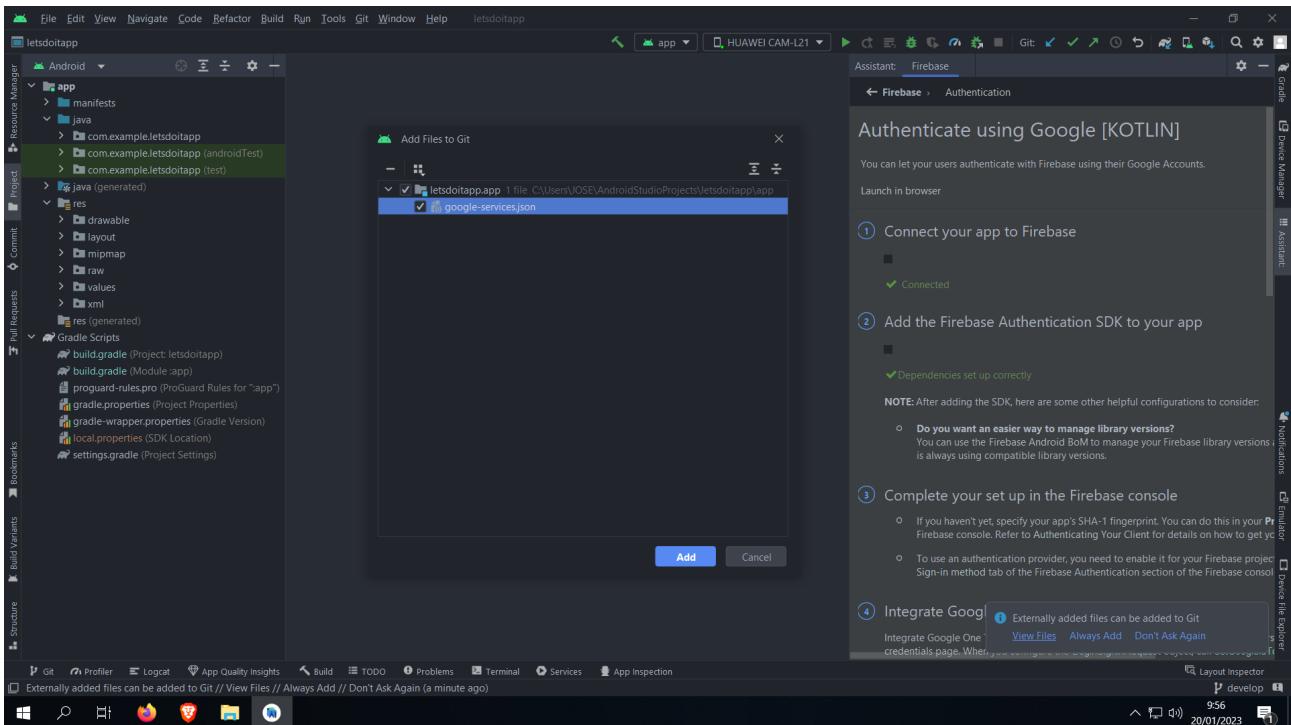
Configuramos Google Analytics



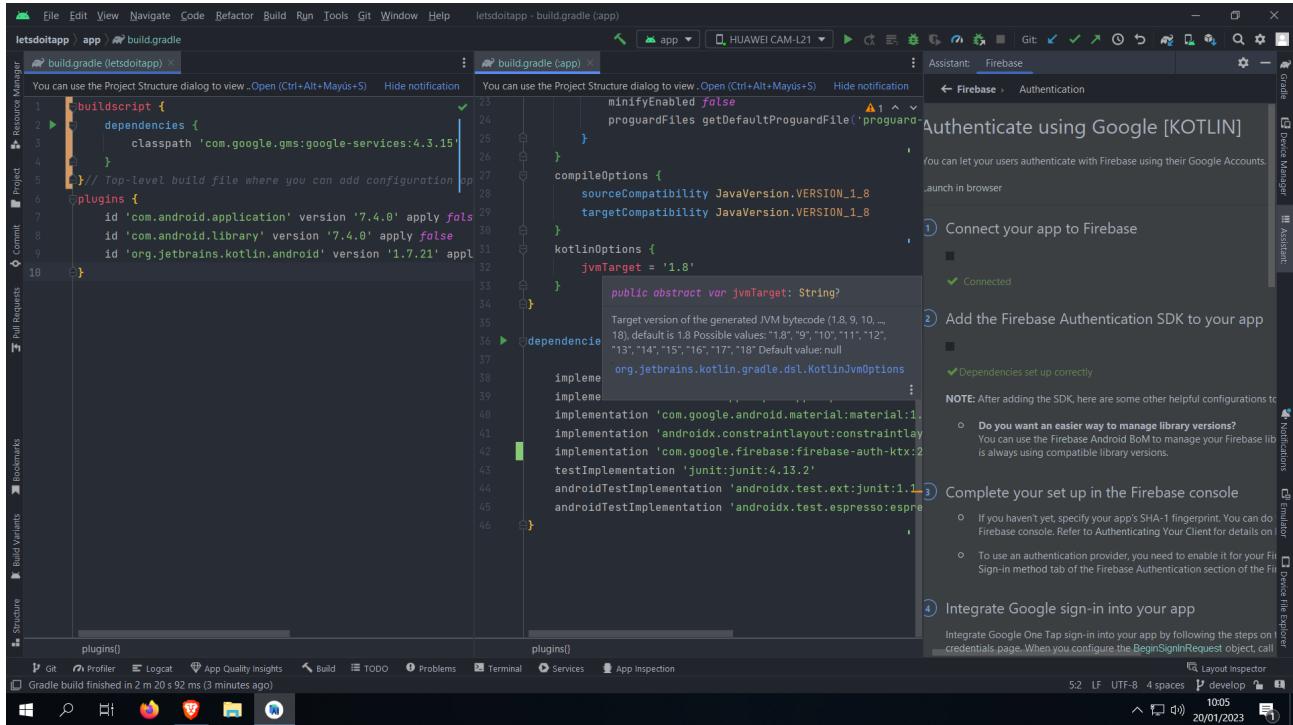
Conectamos Firebase con Android



Agregamos las dependencias de Firebase en los archivos build.gradle de la app y el proyecto.



Como podemos ver también nos ha añadido el fichero google-services.json

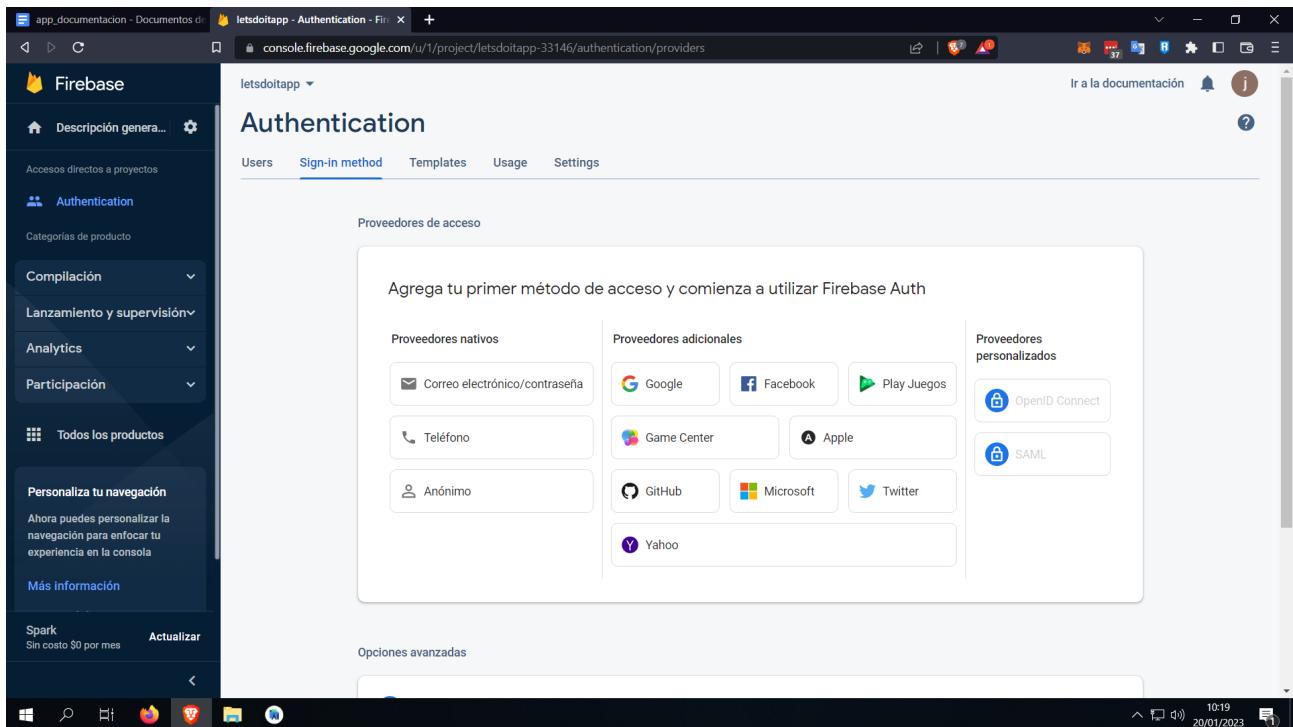


Ya tenemos nuestra base de datos conectada a nuestro proyecto con las dependencias necesarias que nos ha añadido y sincronizado automáticamente.

2.3 Servicios de Firebase

2.3.1 Autenticación

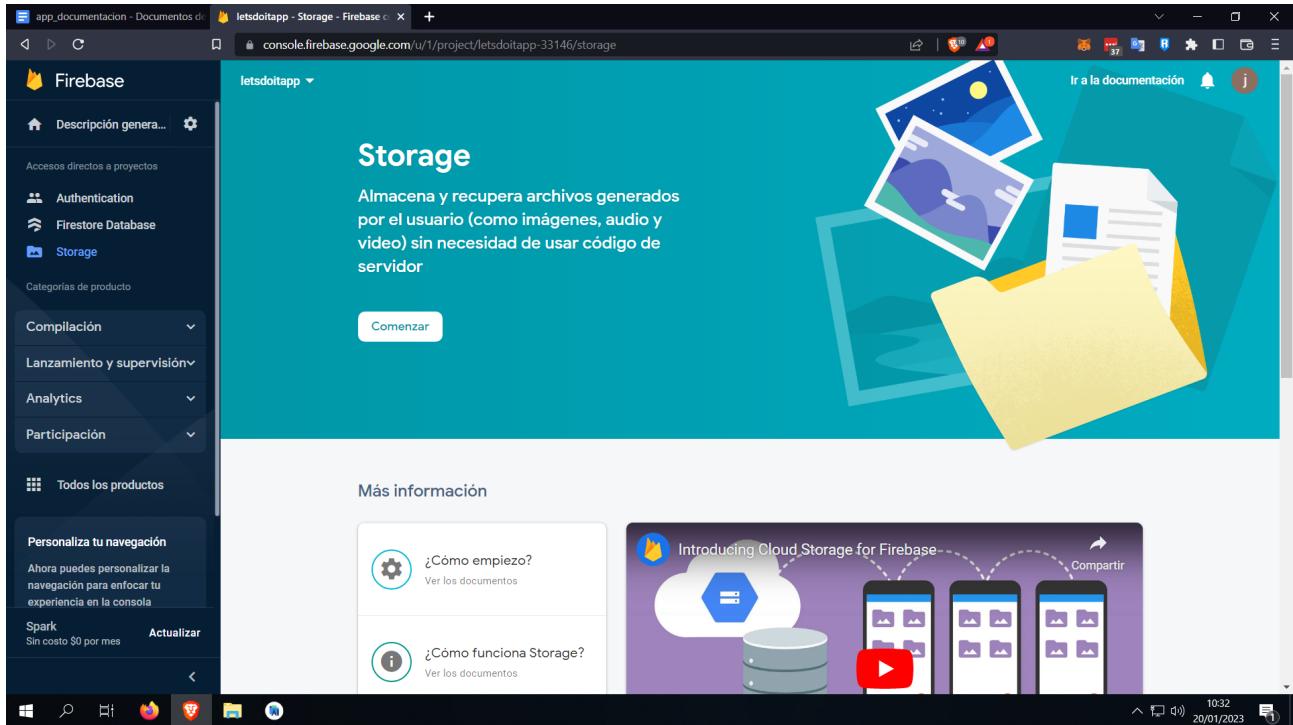
Firebase no solo es una base de datos , también nos proporciona una serie de servicios, por ejemplo tenemos el servicio de autenticación.Esto nos va a permitir tener usuarios y decidir cómo van a iniciar sesión.



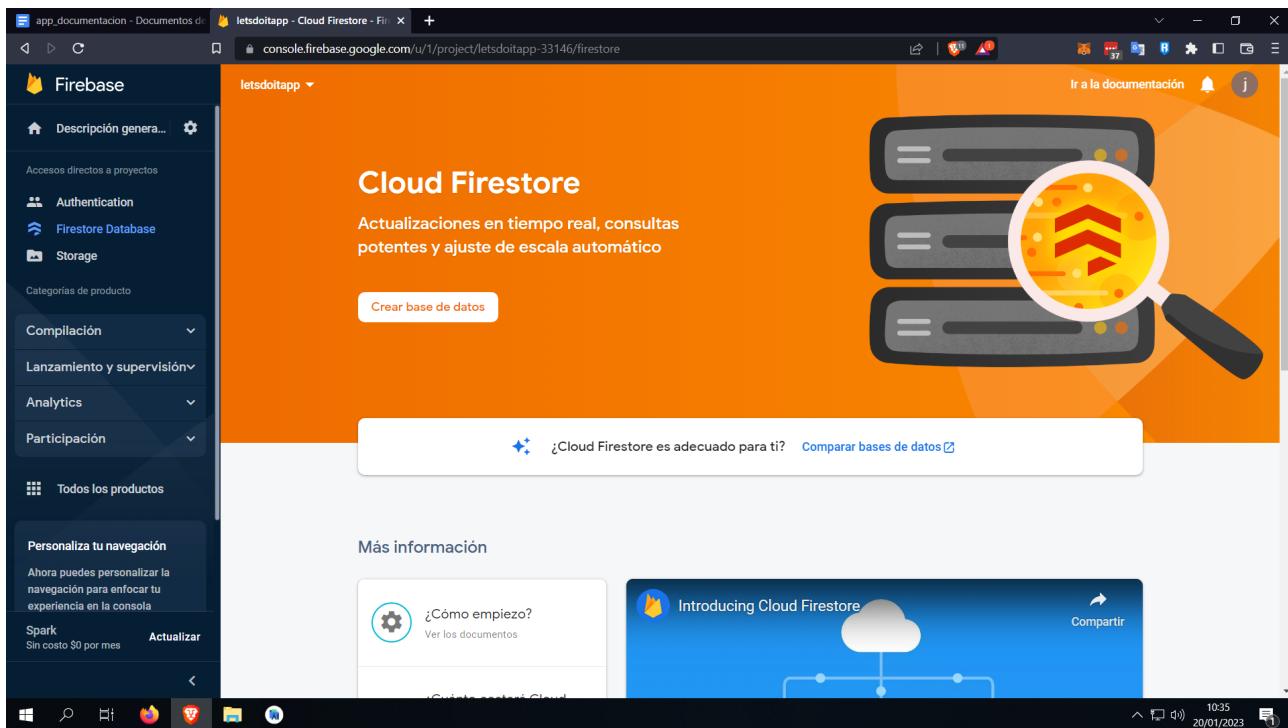
El servicio de Autenticación de Firebase nos provee una serie de servicios como proveedores de acceso como por correo electronico , telefono , Google, Facebook , Github etc.

2.3.2 Storage

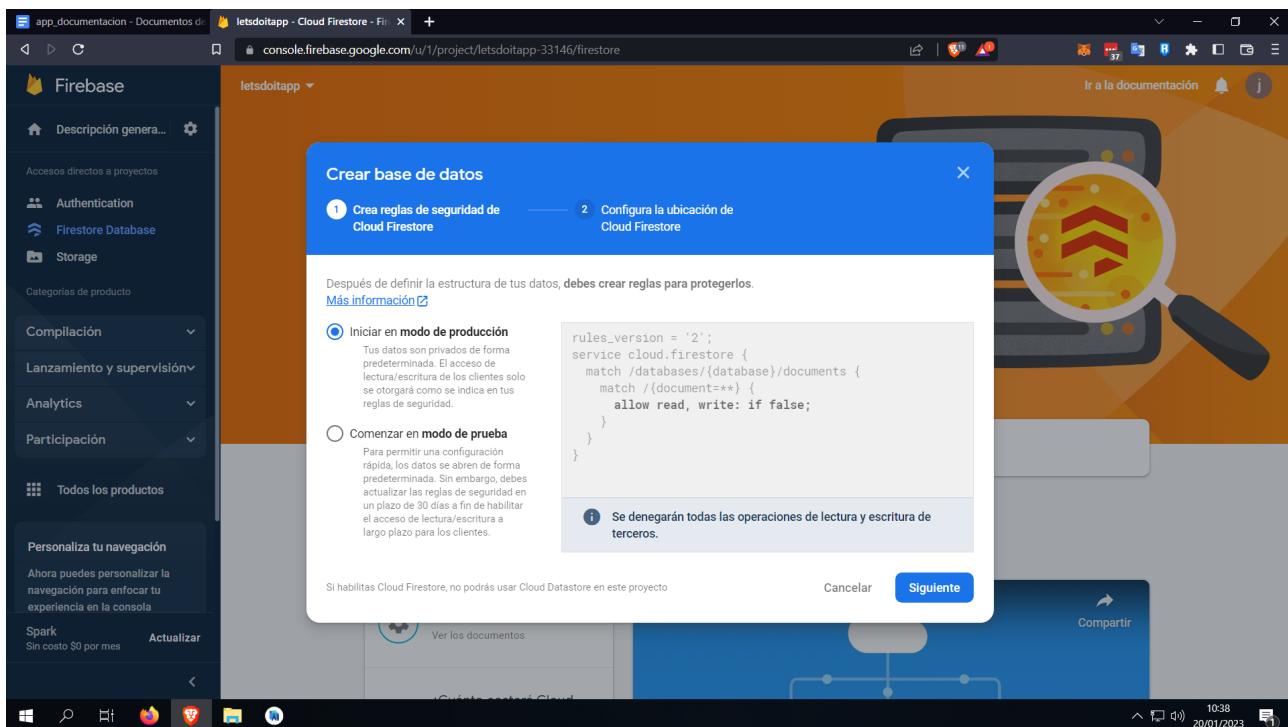
Con este servicio Firebase nos permite guardar archivos y esto es algo que no todas las bases de datos lo permiten. El servicio a un nivel básico es gratuito pero evidentemente cuando el nivel de uso es superior a cierta cantidad hay que pagar el servicio. Para nuestra app será suficiente con este servicio ya que guardaremos algunos archivos de fotos.



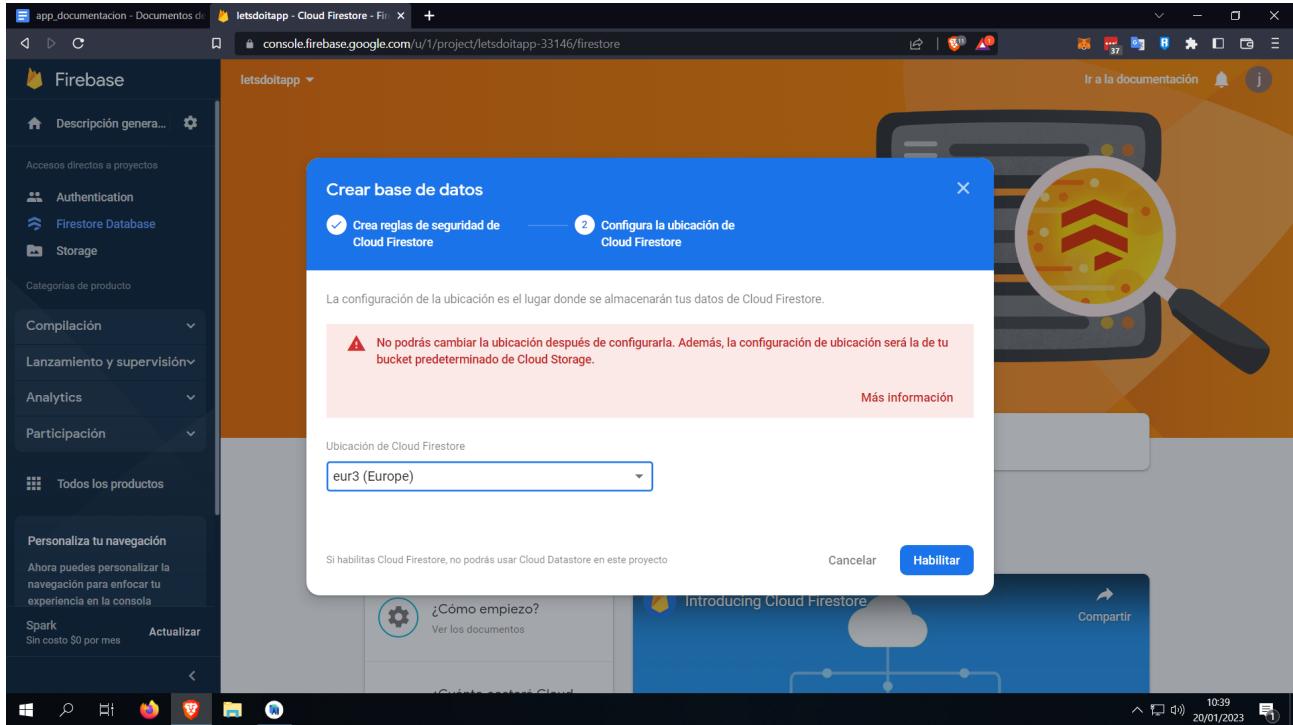
2.3.3 Creación de la base de datos Cloud Firestore



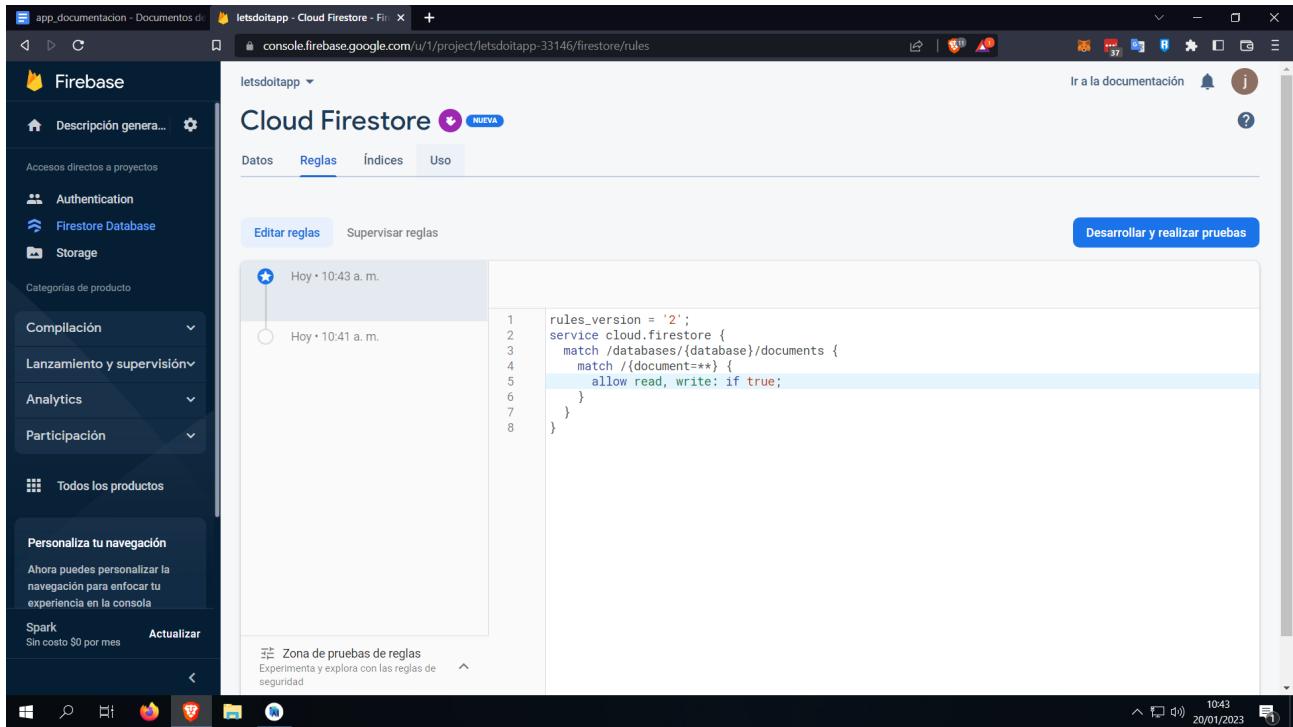
Desde la pantalla Firestore Database podemos crear la base de datos clicando en “Crear base de datos”



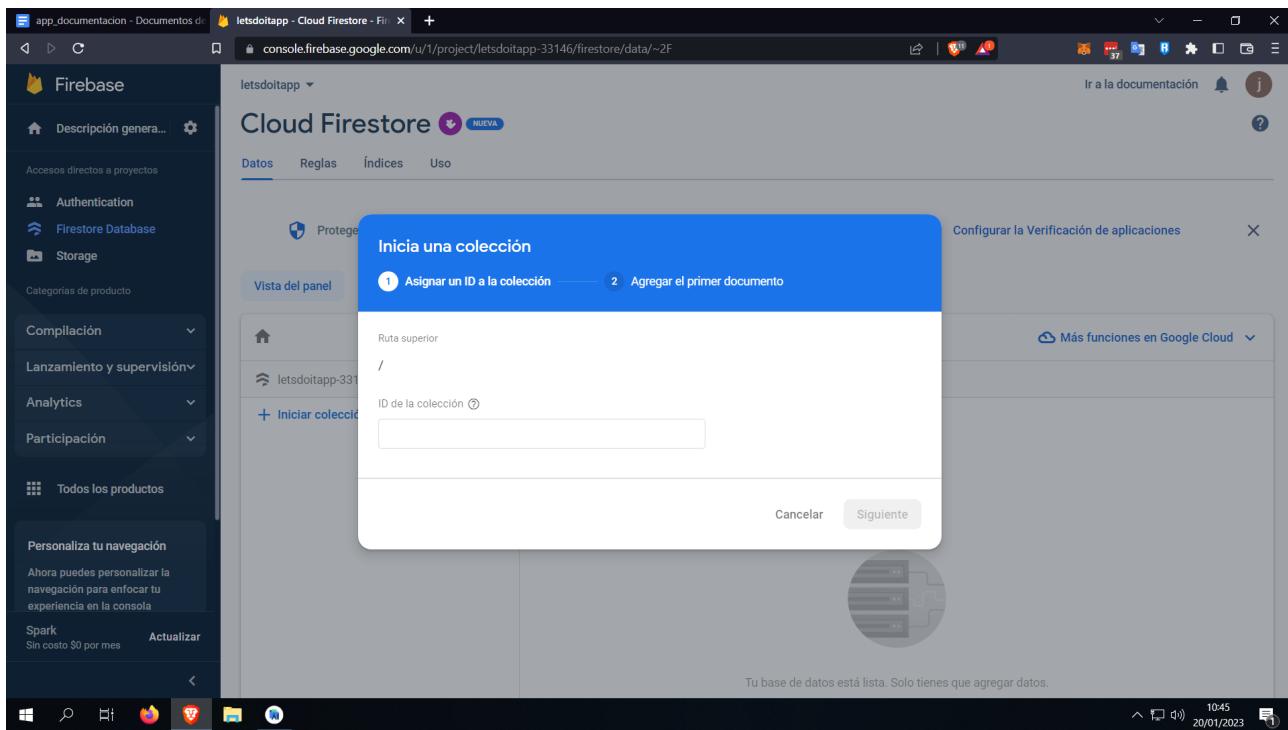
Al crear la base de datos se puede elegir entre el modo prueba (30 días) y el modo producción que será el que seleccionemos.



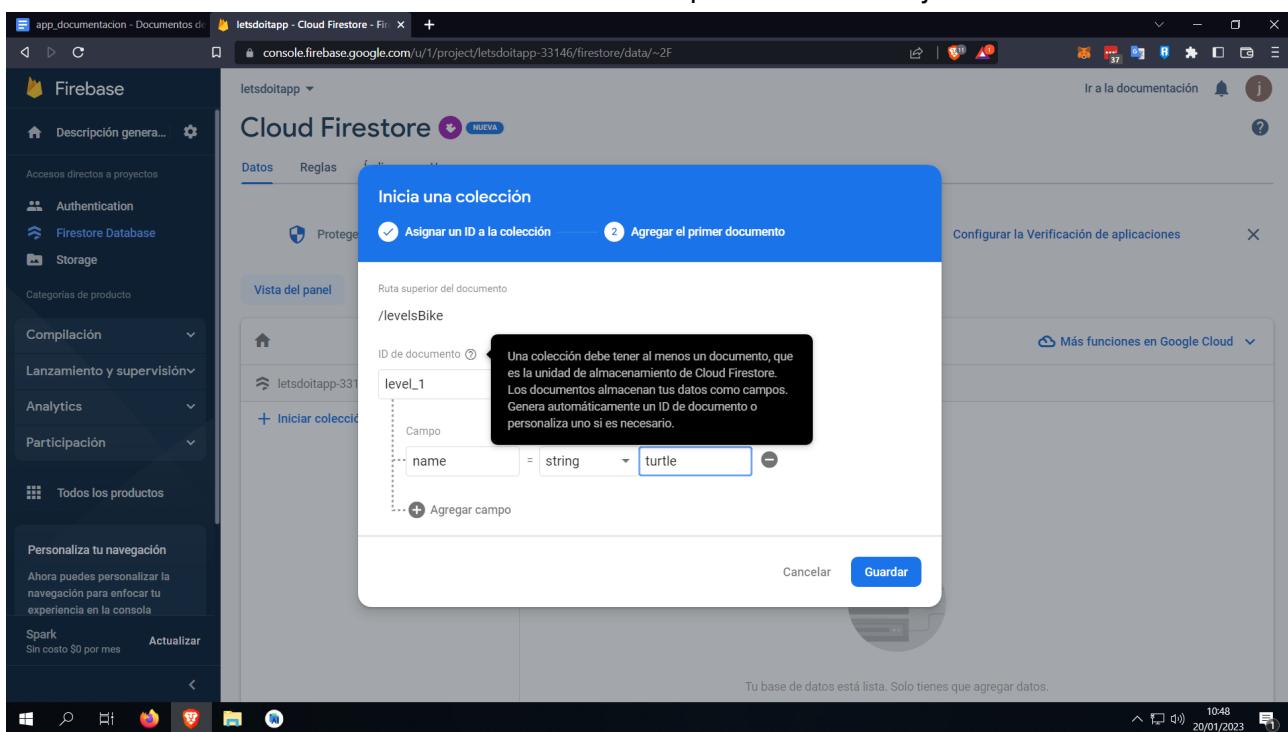
Elegiremos eur3 que corresponde a la zona europea.



Cambiamos las reglas que vienen por defecto para que nos permita leer y escribir.



En Datos se irán añadiendo las colecciones con las que vamos a trabajar.



Según indica Firestore , una colección debe tener al menos un documento, que es la unidad de almacenamiento de Cloud Firestore. Los documentos almacenan los datos en campos.

Inicia una colección

Asignar un ID a la colección: level_1

Agregar el primer documento

Ruta superior del documento: /levelsBike

ID de documento: level_1

Campo	Tipo	Valor
DistanceTarget	number	40
RunsTarget	number	4
image	string	level_1
name	string	turtle

... Agregar campo

En nuestra app tenemos por ejemplo: la Colección levelsBike, que consta de:

Ruta superior del documento = /levelsBike

ID de documento = level_1

Campo: DistanceTarget, Tipo = number, Valor = 40

Campo: RunsTarget, Tipo = number, Valor = 4

Campo: Image, Tipo = string, Valor = level_1

Campo = name , Tipo =string, Valor = turtle

La colección Locations, donde vamos a guardar los puntos de ubicación por los que el usuario va pasando a medida que va haciendo su carrera , que nos servirá para después poder recrear toda la ruta que ha seguido y marcarla en el mapa.

Las colecciones RunsBike, RunsRunning, RunsRoller, son las carreras que se han hecho con bici, patin o corriendo. Se ha hecho una colección distinta por cada deporte, para que cuando haya que recorrer la colección para localizar algún usuario, el tiempo de consulta va a ser más corto.

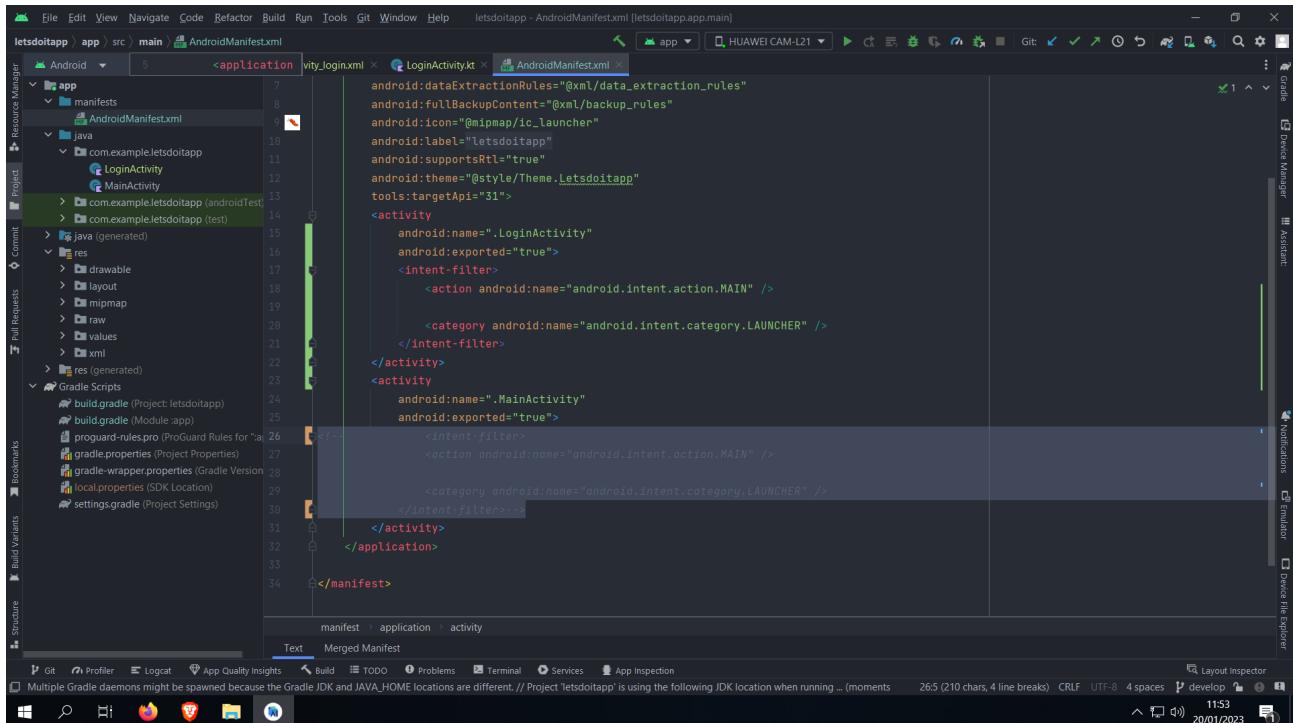
Las colecciones Totals, son donde tendremos los récords de velocidad media, distancia , velocidad, total distancia , total carreras y total tiempo.

La colección Users, donde tendremos control de los usuarios, aunque ya controlamos el acceso a los usuarios desde la autenticación , está colección nos servirá para saber datos más específicos del usuario , por ejemplo si es premium, o algún otro tipo de dato como nombres , apellidos , dirección etc.

Podría haber hecho todo junto el nombre del nivel y la imagen pero a la hora de utilizar código tendría que hacer dos consultas , una para saber cual es la imagen otra para saber cuales son los requisitos, por tanto al hacer dos consultas sería algo más lento.

3. Diseño del Layout

3.1 Modificación del AndroidManifest.xml



```

<application>
    <activity android:name=".LoginActivity" android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".MainActivity" android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>

```

Nos vamos a crear una nueva Activity que va a ser nuestra pantalla de Login. Indicaremos en el AndroidManifest.xml que el LoginActivity será nuestra pantalla de inicio cuando se lance la aplicación. Comentamos las líneas que corresponden al main activity para que se inicie LoginActivity. Además vamos a poner un parámetro extra para que la navegación sea vertical.

3.2 Creación del archivo styles.xml

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="viewCustom">
        <item name="android:layout_width">match_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:layout_margin">8dp</item>
    </style>

    <style name="NumberPickerTheme" parent="Theme.AppCompat.Light">
        <item name="colorAccent">@color/white</item>
        <!-- To remove divider of the number picker -->
        <item name="colorControlNormal">@android:color/transparent</item>
        <!--item name="colorControlNormal">@color/grey_dark</item-->
        <item name="textColorPrimary">@color/white</item>
        <item name="android:background">@color/grey_dark</item>
        <item name="android:textSize">30sp</item>
    </style>
</resources>

```

Para la creación del layout de login vamos a definir un nuevo archivo de estilos donde personalizamos algunos parámetros en común en todos los layouts. Para ello creamos el archivo **styles.xml** dentro de values.

3.3 Creación del archivo strings.xml

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">letsdoitapp</string>
    <string name="email">Email</string>
    <string name="password">Contraseña</string>
    <string name="forgotPassword">¿ Olvidaste tu contraseña ?</string>
    <string name="login">Iniciar Sesión</string>
    <string name="resetPassword">Enviar Contraseña</string>
    <string name="invitation">¿ Que eres capaz de hacer ?</string>
    <string name="accept">He leido y acepto los<u>s</u></string>
    <string name="terms">Términos y condiciones de uso.</string>
    <string name="empty_email">Introduce un email por favor</string>
    <string name="invalid_email">Parece que el email esta mal escrito</string>
    <string name="google">Google</string>
</resources>

```

También vamos a crear el archivo string.xml donde vamos a guardar diferentes textos o palabras que vamos a utilizar en la app.

3.4 Personalización del fichero themes.xml

```

<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.Letsdoitapp" parent="Theme.MaterialComponents.DayNight.NoActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor" >@color/orange_strong</item>
        <!--<item name="android:statusBarColor">?attr/colorPrimaryVariant</item>-->
        <!-- Customize your theme here. -->
    </style>
</resources>

```

En este fichero modificamos la status bar para que muestre la barra de status en el color deseado y cambiamos a .NoActionBar para que no muestre la barra de acción.

3.5 Creación del login layout

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/white"
    android:orientation="vertical"
    tools:context=".LoginActivity">

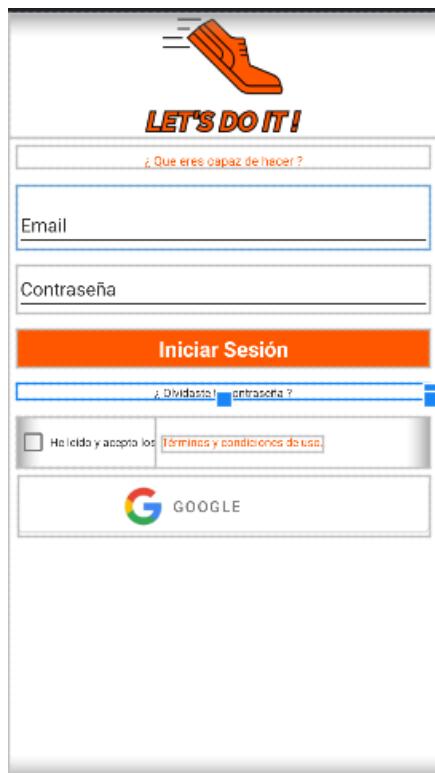
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="280dp"
        app:srcCompat="@drawable/banner_zapa_in" />

    <TextView
        android:id="@+id/txtInvitation"
        style="@style/viewCustom"
        android:gravity="center_horizontal"
        android:paddingTop="5dp"
        android:text="@string/invitation"
        android:textColor="@color/orange_strong"
        android:textSize="12sp" />

    <EditText
        android:id="@+id/etEmail"
        style="@style/viewCustom"
        android:layout_marginTop="0dp"
        ...>

```

Nuestra pantalla de Login está compuesta de un LinearLayout principal que contiene un ImageView con el logo, un textView con una frase, dos editText para el email y la contraseña. Un textView para iniciar sesión. Un textView con la frase de olvidar contraseña . Dentro tenemos otro LinearLayout horizontal que contiene los textView con los términos y condiciones y finalmente un botón de inicio de Google.



Pantalla de Login.

3.6 UX Experiencia del usuario en Inicio de Sesión

En el login solamente hay opción de iniciar sesión, no hay botón de registro. El diseño se ha simplificado. Por defecto la parte de “Términos y condiciones” no se va a visualizar. Desde el código manejaremos la opción de mostrarlo.

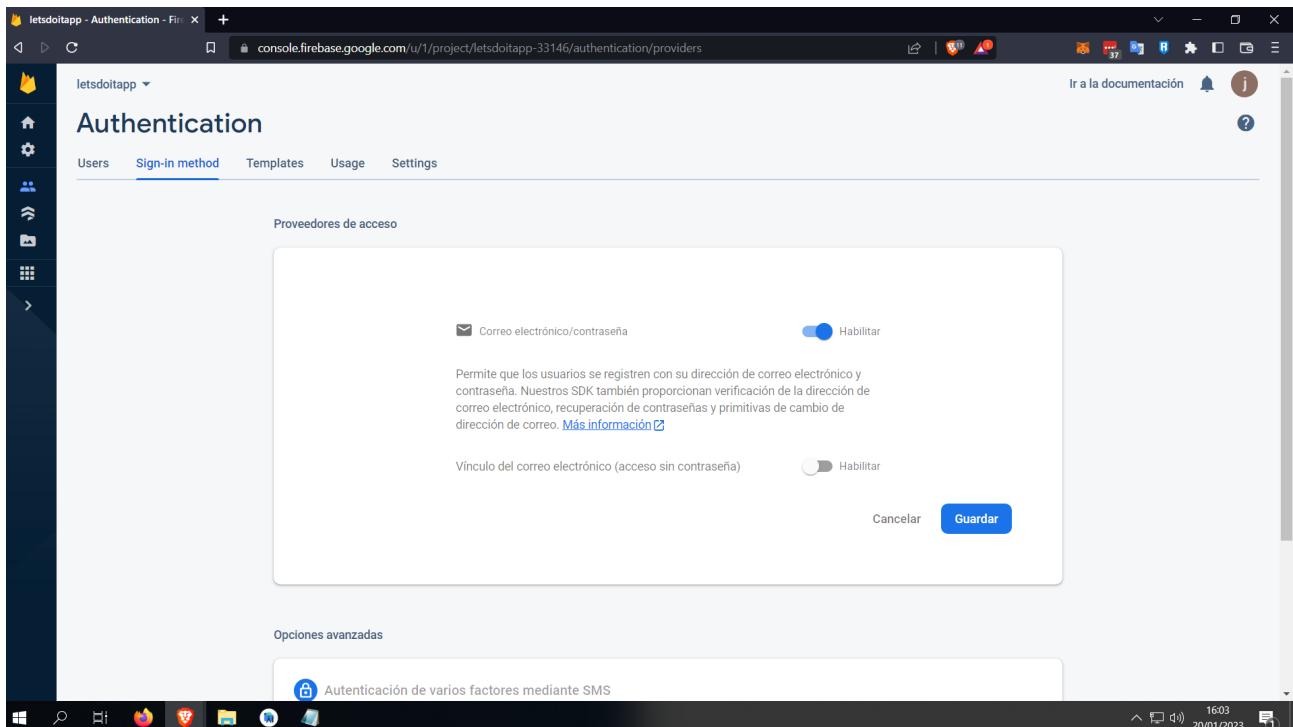
Cuando un usuario entra por primera vez y necesita crearse un usuario introducirá un email y una contraseña y cuando pulse el botón de iniciar sesión aparecerán los Términos y condiciones y entonces tendrá que aceptar para crear un nuevo usuario , pero solamente si el usuario no existe, entonces automáticamente el usuario se crea e inicia sesión.

Desde el botón de Iniciar Sesión se hará todo el proceso. Se simplifica el uso reduciendo los elementos en pantalla y mejora la experiencia del usuario.

Otra cosa es cuando el usuario ha cerrado la aplicación , la próxima vez que la abra debe de acceder directamente a la aplicación sin tener que volver a iniciar sesión.

Además cuando el usuario ha entrado ya en la aplicación e intenta ir hacia atrás , no debe de ir al menú de Login.

4. Implementación de Login y Autenticación del Usuario



Para que nuestra app permita el inicio de sesión lo primero que tenemos que hacer es habilitar el acceso por correo electrónico/ contraseña en Firebase.

4.1 Función loginUser()

```

private fun loginUser() {
    email = etEmail.text.toString() //Capturamos los datos de los Editext
    password = etPassword.text.toString()
    mAuth.signInWithEmailAndPassword(email, password) //Hacemos Inicio de Sesión
        .addOnCompleteListener(this) { task -> //Añadimos un Evento en task tenemos información
            if (task.isSuccessful) goHome(email, provider: "email") // Si ha tenido éxito vamos a Home
            else { // Si no ha tenido éxito
                if (lyTerms.visibility == View.INVISIBLE) //Si está invisible
                    lyTerms.visibility = View.VISIBLE //los ponemos visibles
                else { //Si no hay que aceptar los términos
                    val cbAcept =
                        findViewById<CheckBox>(R.id.cbAcept) //Mostramos nuestro checkbox
                    if (cbAcept.isChecked) register() // Si está marcado registramos usuario
                }
            }
        }
}

```

La función comienza capturando el correo electrónico y la contraseña ingresados por el usuario en los campos EditText. Luego, utiliza el método "signInWithEmailAndPassword" de la autenticación de Firebase para intentar iniciar sesión en el usuario con el correo electrónico y la contraseña proporcionados. Si la tarea tiene éxito, el usuario es dirigido a la pantalla "home". Si la tarea no tiene éxito, el código verifica si ciertos elementos de diseño son visibles y si una casilla de verificación está marcada, y si es así, llama a una función de "registro" separada.

4.2 Función register()

```

private fun register() {
    email = etEmail.text.toString() //Capturamos los datos de los Editext
    password = etPassword.text.toString() //Capturamos los datos de los Editext
    FirebaseAuth.getInstance()
        .createUserWithEmailAndPassword(email, password) //Registramos usuario
        .addOnCompleteListener { task -> //Creamos un evento
            if (task.isSuccessful) { //Si tiene éxito
                val dateRegister =
                    SimpleDateFormat("dd/MM/yyyy").format(Date()) //Capturamos fecha
                val dbRegister = FirebaseFirestore.getInstance() //Creamos instancia de la bd
                dbRegister.collection("users") //creamos colección
                    .document(email) //creamos documento con email
                    .set(
                        hashMapOf(
                            "user" to email,
                            "dateRegister" to dateRegister
                        )
                    ) //creamos un set con user y dateregister
                goHome(email, provider: "email") //Vamos a home con el email
            } else Toast.makeText(context: this, text: "Error, algo ha ido mal :(", Toast.LENGTH_SHORT).show()
        }
}

```

La función comienza capturando el correo electrónico y la contraseña ingresados por el usuario en los campos EditText. Luego, utiliza el método "**createUserWithEmailAndPassword**" de la autenticación de Firebase para registrar un nuevo usuario con el correo electrónico y la contraseña proporcionados. Si la tarea tiene éxito, captura la fecha actual, crea una instancia de Firebase Firestore, crea un nuevo documento en una colección "users", con el correo electrónico como nombre del documento, y establece los valores de "user" y "dateRegister" en el documento con el correo electrónico y la fecha capturada respectivamente. Finalmente, el usuario es dirigido a la pantalla "home". Si la tarea no tiene éxito, muestra un mensaje toast con un mensaje de error.

4.3 Función goHome()

```
private fun goHome(email: String, provider: String) {
    useremail = email
    providerSession = provider
    val intent = Intent(packageContext: this, MainActivity::class.java)
    startActivity(intent)
```

Esta función en Kotlin se llama "goHome" y tiene dos parámetros de entrada: una cadena de correo electrónico y una cadena de proveedor. La función comienza asignando los valores de los parámetros de entrada a dos variables globales llamadas "useremail" y "providerSession". Luego, crea una nueva intención (un objeto que representa una acción que se va a realizar) y especifica que la intención es para abrir la "MainActivity" (una clase o pantalla en la aplicación). Finalmente, utiliza el método "startActivity" para iniciar la nueva intención y navegar a la pantalla "MainActivity". En resumen, esta función se utiliza para navegar desde la pantalla actual a la pantalla "MainActivity" y pasar la información del correo electrónico y el proveedor de sesión al siguiente pantalla.

4.4 Función onStart()

```
public override fun onStart() {
    super.onStart()
    val currentUser = mAuth.currentUser
    if (currentUser != null) goHome(currentUser.email.toString(), currentUser.providerId)
```

Este código se ejecuta cuando se inicia la actividad (**onStart**). Lo que hace es comprobar si hay un usuario actualmente autenticado en Firebase. Utiliza la clase **FirebaseAuth** para obtener una instancia del usuario actualmente autenticado y almacenarlo en la variable "**currentUser**". Si **currentUser** no es nulo significa que hay un usuario autenticado, por lo que se llama a la función **goHome** pasando el correo electrónico del usuario y el proveedor de autenticación como parámetros.

```

override fun onBackPressed() {
    val startMain = Intent(Intent.ACTION_MAIN)
    startMain.addCategory(Intent.CATEGORY_HOME)
    startMain.flags = Intent.FLAG_ACTIVITY_NEW_TASK
    startActivity(startMain)
}

```

4.5 Función onBackPressed()

Este código se ejecuta cuando se presiona el botón de retorno del dispositivo (onBackPressed). Lo que hace es crear una nueva intención (Intent) utilizando Intent.ACTION_MAIN y añade la categoría Intent.CATEGORY_HOME. Esto significa que se está lanzando una intención para regresar al inicio o pantalla principal del dispositivo.

La bandera Intent.FLAG_ACTIVITY_NEW_TASK indica que se debe crear una nueva tarea para la actividad y se establece en la intención. Finalmente, se llama a startActivity para iniciar la nueva intención y regresar al inicio del dispositivo.

4.6 Función resetPassword()

```

117     private fun resetPassword() {
118         var e = etEmail.text.toString()
119         if (!TextUtils.isEmpty(e)) {
120             mAuth.sendPasswordResetEmail(e)
121                 .addOnCompleteListener { task ->
122                     if (task.isSuccessful) Toast.makeText(
123                         context: this,
124                         text: "Email Enviado a $e",
125                         Toast.LENGTH_SHORT
126                     ).show()
127                     else Toast.makeText(
128                         context: this,
129                         text: "No se encontró el usuario con este correo",
130                         Toast.LENGTH_SHORT
131                     ).show()
132                 }
133             else Toast.makeText( context: this, text: "Indica un email", Toast.LENGTH_SHORT).show()
134         }

```

La función primero obtiene el correo electrónico ingresado por el usuario en un campo de edición de texto y verifica si no está vacío. Si no está vacío, envía un correo electrónico para restablecer la contraseña a la dirección de correo electrónico utilizando el método

sendPasswordResetEmail() de la biblioteca de autenticación de Firebase. La función luego muestra un mensaje Toast al usuario indicando si el correo electrónico se envió con éxito o si hubo un error. Si el campo de correo electrónico está vacío, muestra un mensaje Toast pidiendo al usuario que ingrese un correo electrónico.

4.7 Función signOut()



```

new *
fun callSignOut(view: View) {
    signOut()
}

new *
private fun signOut() {
    FirebaseAuth.getInstance().signOut()
    startActivity(Intent(packageContext: this, LoginActivity::class.java))
}

```

La función utiliza el método `signOut()` de la instancia de `FirebaseAuth` para cerrar la sesión del usuario actual. Luego, inicia una nueva actividad utilizando un `Intent`, pasando como parámetro la clase `LoginActivity`, lo que hace que se rediriga al usuario a la pantalla de inicio de sesión de la aplicación.

5. Implementación de Validación de Usuario y Contraseña

5.1 Función manageButtonLogin()



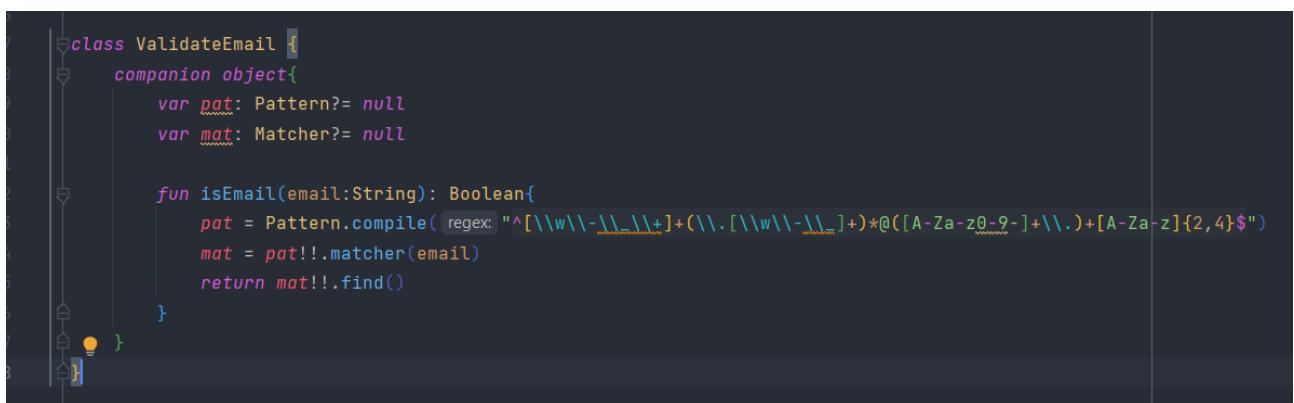
```

@ JordanPro *
private fun manageButtonLogin() [
    var tvLogin = findViewById<TextView>(R.id.tvLogin)
    email = etEmail.text.toString()
    password = etPassword.text.toString()
    //if (TextUtils.isEmpty(password) || ValidateEmail.isEmail(email) == false) {
    if (TextUtils.isEmpty(password) || !ValidateEmail.isEmail(email)) {
        tvLogin.setBackgroundColor(ContextCompat.getColor(context: this, R.color.gray))
        tvLogin.isEnabled = false
    } else {
        tvLogin.setBackgroundColor(ContextCompat.getColor(context: this, R.color.green))
        tvLogin.isEnabled = true
    }
]

```

La función **manageButtonLogin** recupera una vista de tipo TextView llamada **tvLogin**, y obtiene el contenido de los campos de correo electrónico y contraseña ingresados por el usuario en dos variables llamadas **email** y **password**. Luego, comprueba si el campo de contraseña está vacío o si el correo electrónico no es válido utilizando una expresión regular. Si alguna de estas condiciones se cumple, cambia el color de fondo del botón y lo deshabilita, de lo contrario, cambia el color de fondo del botón y lo habilita.

5.2 Clase ValidateEmail()



La clase **ValidateEmail** contiene un objeto **companion**. El objeto **companion** tiene dos propiedades llamadas **pat** y **mat**, que son de los tipos **Pattern** y **Matcher** respectivamente. El objeto **companion** también tiene un método estático llamado **isEmail()** que toma un parámetro **String** llamado **email**. El método utiliza el método **compile()** de la clase **Pattern** para crear un patrón de expresión regular para comparar con la cadena de correo electrónico. La expresión regular se utiliza para verificar si la cadena de correo electrónico está en el formato correcto (es decir, coincide con el patrón de una dirección de correo electrónico válida). Luego se utiliza el método **matcher()** para crear un objeto **Matcher** utilizando la cadena de correo electrónico y el patrón. El método **find()** es llamado sobre el objeto **matcher** y el resultado se devuelve como un booleano. Este método verificará si la dirección de correo electrónico está en el formato correcto y devolverá **true** si es correcto y **false** si no lo es.

5.2.1 Expresión Regular utilizada:

```
(^[\w\.-\\_\\]+(\. [\w\.-\\_]+)*@[A-Za-z0-9-]+\.\.[A-Za-z]{2,4}$)
```

Esta expresión regular comprueba si la dirección de correo electrónico proporcionada está en un formato válido. La expresión regular especifica comprueba lo siguiente:

- "^" comienzo de línea.

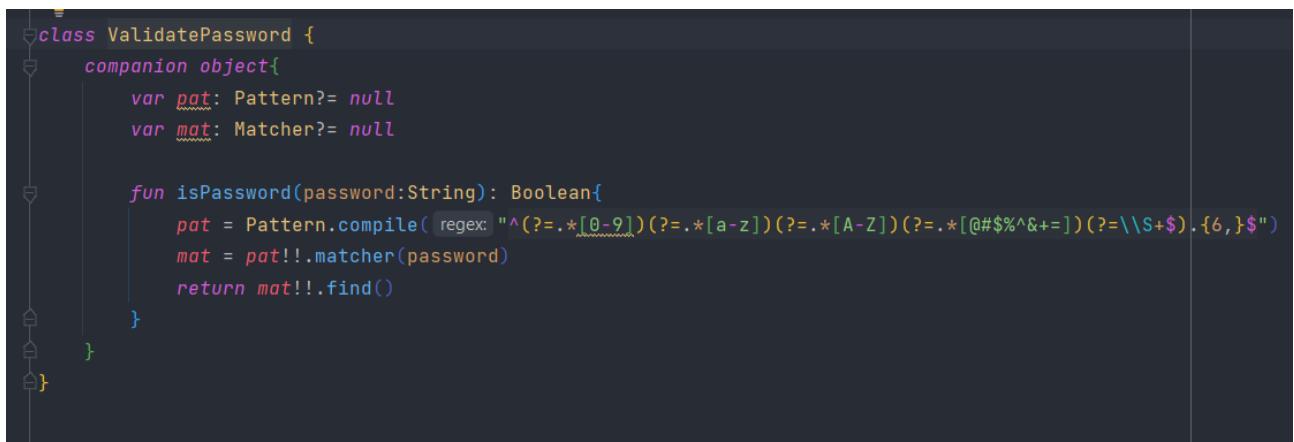
- "[\w\-_\+\+]" uno o más caracteres de palabra, guiones, subrayados o signos más.
- "(.\[\w\-_\+\])*" cero o más apariciones de un punto seguido de uno o más caracteres de palabra, guiones o subrayados.
- "@" un símbolo arroba.
- "([A-Za-z0-9-]+\.)+" una o más apariciones de caracteres de palabra, números o guiones, seguidos de un punto.
- "[A-Za-z]{2,4}\$\$" dos a cuatro caracteres de palabra al final de la línea.
-

Esta expresión regular coincidirá con la mayoría de los formatos de correo electrónico estándar, como name@domain.com, name+tag@sub.domains.com

No coincidirá con algunas otras variaciones como name@sub.domains.travel, name@sub.domains.travel.travel, etc.

Entonces, esta expresión regular comprueba si la dirección de correo electrónico proporcionada comienza con uno o más caracteres de palabra, guiones, subrayados o signos más, seguidos de cero o más apariciones de un punto y uno o más caracteres de palabra, guiones o subrayados, luego un símbolo arroba, una o más apariciones de caracteres de palabra, números o guiones, seguidos de un punto y dos a cuatro caracteres de palabra al final.

5.3 Clase ValidatePassword()



```

class ValidatePassword {
    companion object{
        var pat: Pattern?= null
        var mat: Matcher?= null

        fun isPassword(password:String): Boolean{
            pat = Pattern.compile( regex: "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%^&+=])(?=\\S+).{6,}$")
            mat = pat!!.matcher(password)
            return mat!!.find()
        }
    }
}

```

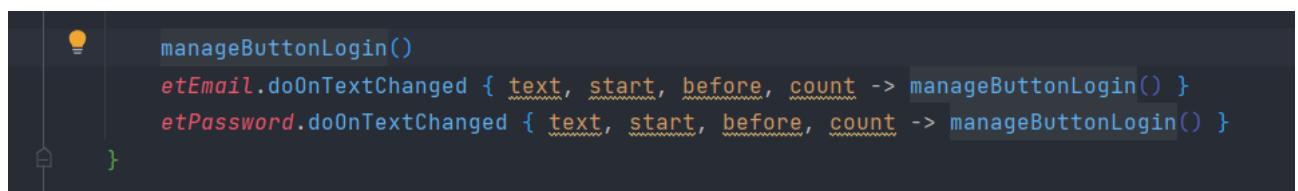
La clase **ValidatePassword()** contiene un objeto “**companion**”. El objeto “**companion**” tiene dos propiedades llamadas pat y mat, que son de los tipos **Pattern** y **Matcher** respectivamente. El objeto “**companion**” también tiene un método estático llamado **isPassword()** que toma en un parámetro String llamado password. El método utiliza el método **compile()** de la clase Pattern para crear un patrón de expresión regular para comparar con la cadena de contraseña. La expresión regular se utiliza para verificar si la cadena de contraseña está en el formato correcto, verifica si la contraseña cumple con los criterios de complejidad como:

- Al menos 1 dígito
- Al menos 1 letra minúscula
- Al menos 1 letra mayúscula
- Al menos 1 carácter especial (@#\$%^&+=)
- Al menos 6 caracteres de longitud
- Sin espacios en blanco

Luego se utiliza el método matcher() para crear un objeto Matcher utilizando la cadena de contraseña y el patrón. El método find() es llamado sobre el objeto matcher y el resultado se devuelve como un booleano. Este método verificará si la contraseña proporcionada cumple con todos los criterios de complejidad y devolverá true si es correcto y false si no lo es.

En resumen, esta clase contiene un método estático que verifica si la contraseña proporcionada cumple con ciertos criterios de complejidad utilizando una expresión regular y devuelve un valor booleano indicando si cumple o no con los criterios de complejidad requeridos.

5.4 Función manageButtonLogin()



Este código está llamando a la función **manageButtonLogin**, y también establece dos “listeners” de cambios de texto en las vistas de edición **etEmail** y **etPassword**. El método **doOnTextChanged** es una función de extensión que permite establecer un listener para los cambios de texto en las vistas **EditText**. El listener se activa cada vez que el texto en el **EditText** cambia, y llama nuevamente a la función **manageButtonLogin**. Esto permite que la función **manageButtonLogin** verifique los campos de correo electrónico y contraseña cada vez que el usuario escribe algo, y actualice el botón de inicio de sesión de acuerdo, habilitando o deshabilitando el botón y cambiando su color si los campos no están vacíos y el correo electrónico es válido o no.

```

private fun manageButtonLogin() {
    val tvLogin = findViewById<TextView>(R.id.tvLogin)
    email = etEmail.text.toString()
    password = etPassword.text.toString()
    val isValidEmail = ValidateEmail.isEmail(email)
    val isValidPassword = ValidatePassword.isPassword(password)

    if (email.isNotEmpty()) {
        if (!isValidEmail && !hasEmailErrorShown) {
            Toast.makeText(context: this, text: "El correo electrónico no es válido", Toast.LENGTH_SHORT)
                .show()
            hasEmailErrorShown = true
        } else if (isValidEmail) {
            hasEmailErrorShown = false
        }
    }

    if (password.isNotEmpty()) {
        if (!isValidPassword && !hasPasswordErrorShown) {
            Toast.makeText(context: this, text: "La contraseña no es válida", Toast.LENGTH_SHORT).show()
            hasPasswordErrorShown = true
        } else if (isValidPassword) {
            hasPasswordErrorShown = false
        }
    }
}

```

```

if (!isValidPassword || !isValidEmail) {
    tvLogin.setBackgroundColor(ContextCompat.getColor(context: this, R.color.gray))
    tvLogin.isEnabled = false
} else {
    tvLogin.setBackgroundColor(ContextCompat.getColor(context: this, R.color.orange_strong))
    tvLogin.isEnabled = true

    // Obtener el objeto InputMethodManager
    val imm = getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager

    // Ocultar el teclado virtual
    imm.hideSoftInputFromWindow(currentFocus?.windowToken, flags: 0)
}

```

La función `manageButtonLogin()` es responsable de validar el correo electrónico y la contraseña ingresados por el usuario y de habilitar o deshabilitar un botón de inicio de sesión en consecuencia. La función también oculta el teclado virtual después de que se hayan ingresado los datos y se haya habilitado el botón de inicio de sesión.

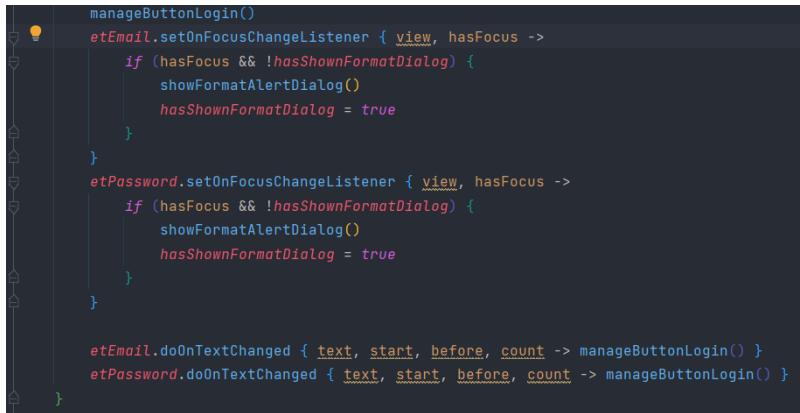
Primero, la función recupera el valor actual del correo electrónico y la contraseña de las vistas `EditText` correspondientes. Luego, utiliza las clases `ValidateEmail` y `ValidatePassword` para validar el correo electrónico y la contraseña, respectivamente. Si el correo electrónico o la contraseña no cumplen con los requisitos de formato, se muestra un mensaje de error en forma de `Toast` y se establece una bandera correspondiente (`hasEmailErrorShown` o `hasPasswordErrorShown`) para evitar que se muestre el mensaje de error repetidamente. Si el correo electrónico o la contraseña son válidos, se restablece la bandera correspondiente.

Luego, la función verifica si el correo electrónico y la contraseña son válidos. Si alguno de los dos no es válido, se deshabilita el botón de inicio de sesión y se establece su fondo en gris. Si ambos son válidos, se habilita el botón de inicio de sesión y se establece su fondo en naranja.

La función también oculta el teclado virtual utilizando el objeto **InputMethodManager**.

En resumen, la función **manageButtonLogin()** es esencialmente un validador de correo electrónico y contraseña que actualiza el estado del botón de inicio de sesión y oculta el teclado virtual.

5.5 Manejando el foco



```

manageButtonLogin()
    etEmail.setOnFocusChangeListener { view, hasFocus ->
        if (hasFocus && !hasShownFormatDialog) {
            showFormatAlertDialog()
            hasShownFormatDialog = true
        }
    }
    etPassword.setOnFocusChangeListener { view, hasFocus ->
        if (hasFocus && !hasShownFormatDialog) {
            showFormatAlertDialog()
            hasShownFormatDialog = true
        }
    }

    etEmail.doOnTextChanged { text, start, before, count -> manageButtonLogin() }
    etPassword.doOnTextChanged { text, start, before, count -> manageButtonLogin() }
}

```

Aquí se configuran los “**setOnFocusChangeListener**” para dos vistas **EditText** diferentes: **etEmail** y **etPassword**. Los “**setOnFocusChangeListener**” o escuchadores de cambio de enfoque están configurados para activar un cuadro de diálogo de alerta de formato si la vista **EditText** gana el enfoque y el indicador **hasShownFormatDialog** no está establecido. Esto sugiere que el propósito del cuadro de diálogo es pedirle al usuario que ingrese el texto en un formato específico, como una dirección de correo electrónico o una contraseña. Los escuchadores de cambio de texto están configurados para llamar a la función **manageButtonLogin()** cada vez que se cambia el texto en cualquiera de las vistas **EditText**. Esta función puede ser responsable de habilitar o deshabilitar un botón de inicio de sesión en función del contenido de las vistas **EditText**.

En general, este código maneja la autenticación del usuario usando un correo electrónico y una contraseña. El propósito del fragmento de código es asegurarse de que el usuario ingrese su correo electrónico y contraseña en el formato correcto y actualizar el estado de un botón de inicio de sesión en función del contenido de las vistas de **EditText**.

5.6 Función showFormatAlertDialog - Diálogo de Formato de email y contraseña

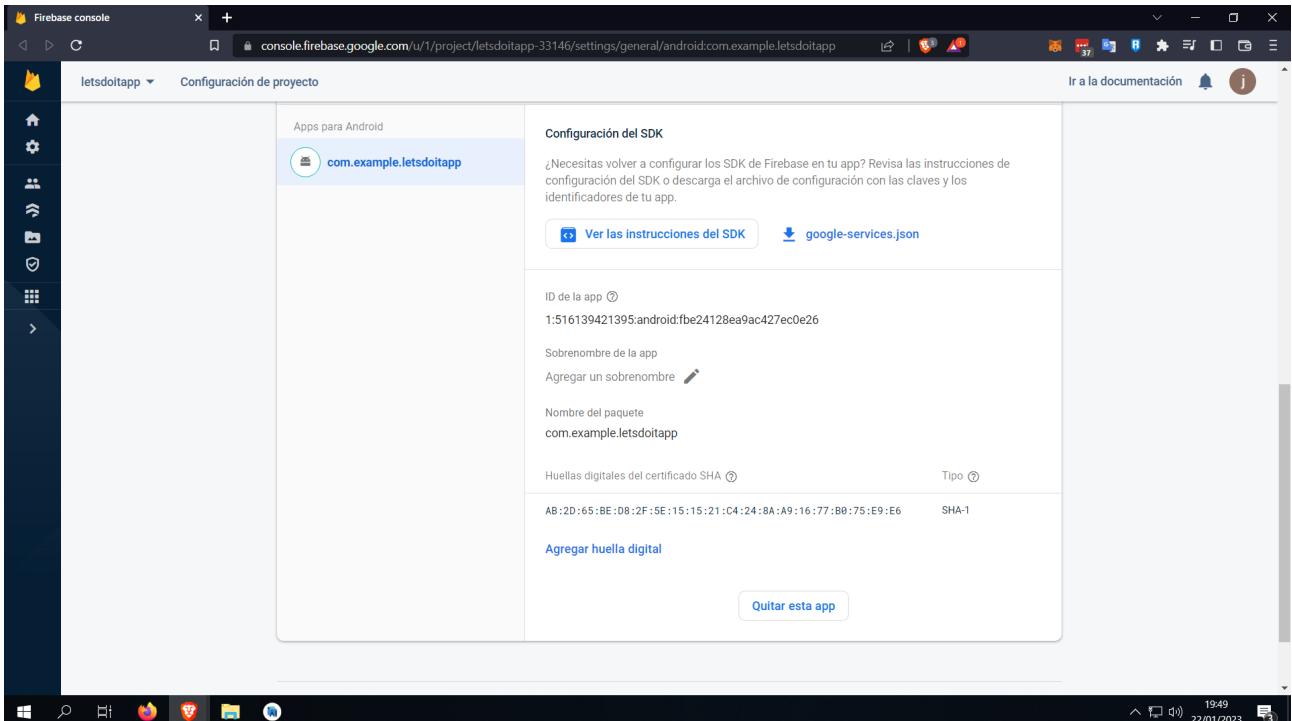
```
private fun showFormatAlertDialog() {  
    val builder = AlertDialog.Builder(context: this)  
    builder.setTitle(getString(R.string.formato))  
        .setMessage(getString(R.string.correo))  
        .setPositiveButton(R.string.acept) { dialog, which -> dialog.dismiss() }  
        .show()  
}
```

La función **showFormatAlertDialog()** muestra un cuadro de diálogo de alerta con un mensaje que indica el formato válido para el correo electrónico y la contraseña. El cuadro de diálogo contiene un título, un mensaje y un botón de aceptar.

El título del cuadro de diálogo indica que se trata del formato válido para el correo electrónico y la contraseña. El mensaje del cuadro de diálogo proporciona información detallada sobre los requisitos de formato que deben cumplirse para que un correo electrónico o una contraseña sean considerados válidos. Específicamente, para el correo electrónico, el formato debe ser nombre@dominio.com, mientras que para la contraseña, debe tener al menos una letra mayúscula, una letra minúscula, un número y un carácter especial (@#\$%^&+=), y debe tener al menos 9 caracteres.

El botón de aceptar del cuadro de diálogo simplemente cierra el cuadro de diálogo cuando se hace clic en él.

6. Implementación de Login con Google



Para configurar el huella digital SHA1 en un proyecto de Firebase, hay que seguir estos pasos:

1. Ve al Firebase Console y selecciona tu proyecto.
2. En el panel de navegación izquierdo, haz clic en el icono "Configuración" (se ve como un engranaje).
3. Bajo la sección "Tus aplicaciones", haz clic en el botón "Agregar aplicación".
4. Selecciona "Android" como la plataforma y ingresa tu nombre de paquete de Android.
5. En la sección "Huellas digitales de certificado SHA", ingresa la huella digital SHA1 de tu keystore de lanzamiento. Puedes obtener esta huella digital ejecutando el comando "keytool -list -v -keystore my-release-key.keystore" en la terminal, donde "my-release-key.keystore" es el nombre del archivo de tu keystore de lanzamiento.
6. Haz clic en "Agregar aplicación" para completar la configuración.

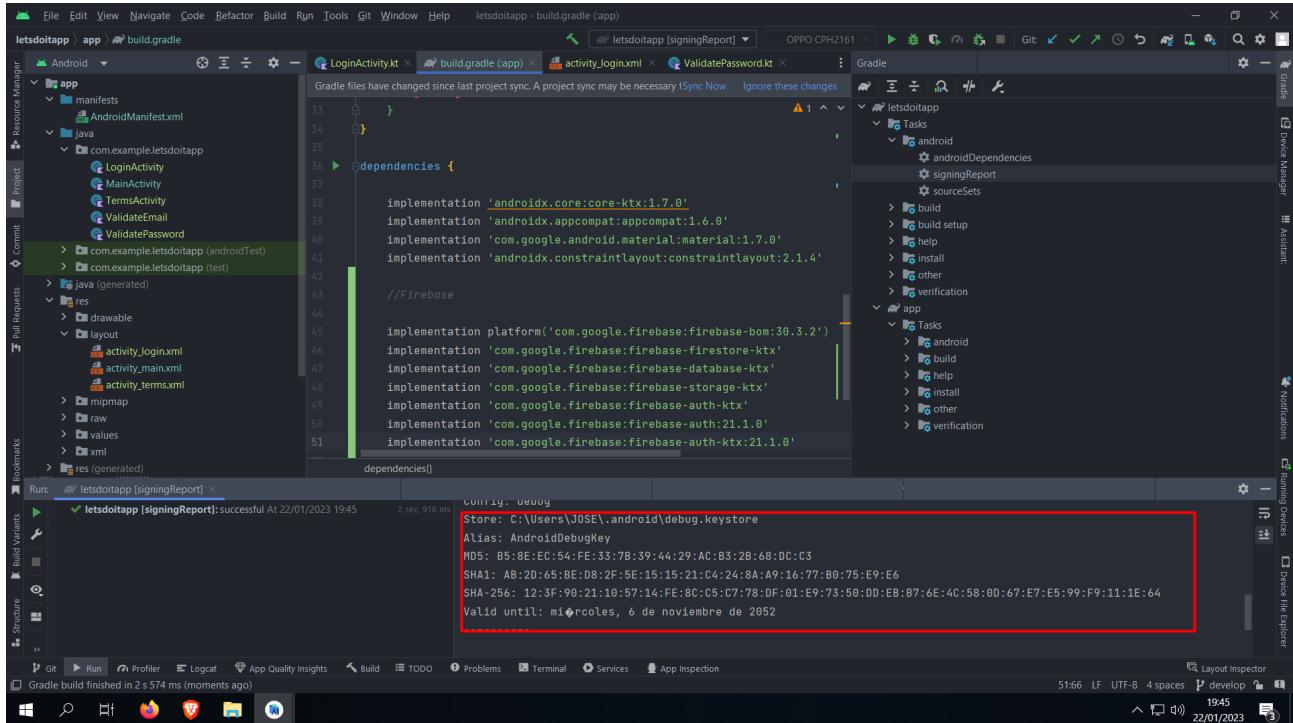
Es importante mencionar que también puedes agregar múltiples huellas digitales si tienes múltiples aplicaciones o diferentes entornos, por ejemplo, desarrollo, pruebas o producción.

Una vez configurada la huella digital SHA1 en Firebase, se utilizará para verificar la autenticidad de tu aplicación cuando se instala en un dispositivo de usuario. Esto garantiza que la aplicación es de una fuente confiable y no ha sido manipulada.

Enlaces:

<https://console.firebaseio.google.com/u/0/>

<https://firebase.google.com/docs/auth/android/google-signin>



6.1 Función signInGoogle()

```

fun callSignInGoogle(view: View) {
    signInGoogle()
}

private fun signInGoogle() {
    val gso = GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
        .requestIdToken("516139421395-t72hf6t4v74ersqsc38e1ecc98di99pg.apps.googleusercontent.com")
        .requestEmail()
        .build()
    googleSignInClient = GoogleSignIn.getClient(activity, gso)
    //googleSignInClient.signOut() // Enviamos un cierre de sesión
    val singInIntent = googleSignInClient.signInIntent
    startActivityForResult(singInIntent, RESULT_CODE_GOOGLE_SIGN_IN)
}

```

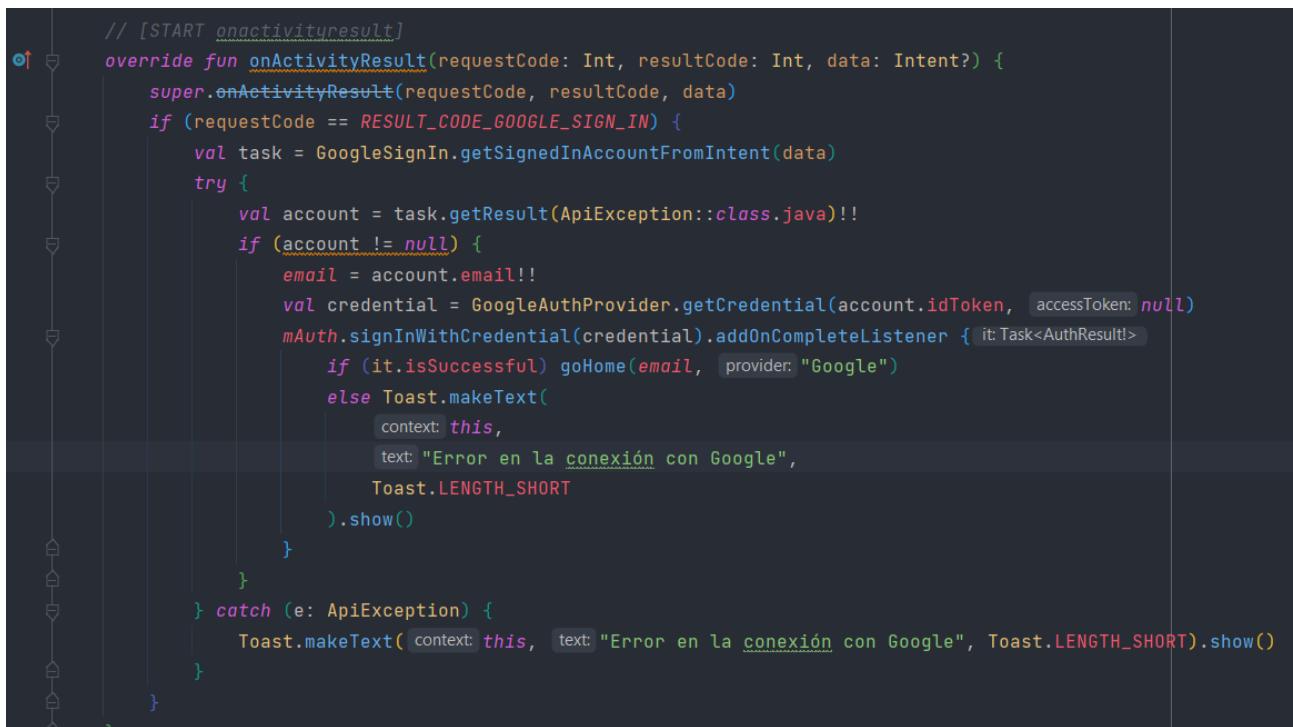
Este código se utiliza para implementar una función de inicio de sesión con Google en la aplicación de Android. La función **signInGoogle()** se utiliza para iniciar el proceso de inicio de sesión.

Empieza creando un objeto **GoogleSignInOptions** usando el **GoogleSignInOptions.Builder** y estableciendo las opciones **requestIdToken** y **requestEmail**. El **requestIdToken** se utiliza para solicitar un token de ID de la API de inicio de sesión de Google, que se puede utilizar para autenticar al usuario con Firebase, el **requestEmail** se utiliza para solicitar la dirección de correo electrónico del usuario.

Luego, crea un objeto **GoogleSignInClient** utilizando el método **GoogleSignIn.getClient()** y pasando el contexto actual y el objeto **GoogleSignInOptions**.

Después de eso, crea un objeto Intent para el proceso de inicio de sesión de Google utilizando el **googleSignInClient.signInIntent**, y luego inicia la actividad para el proceso de inicio de sesión llamando a **startActivityForResult()** y pasando el objeto **Intent**, así como un código de solicitud para identificar el resultado del proceso de inicio de sesión más tarde.

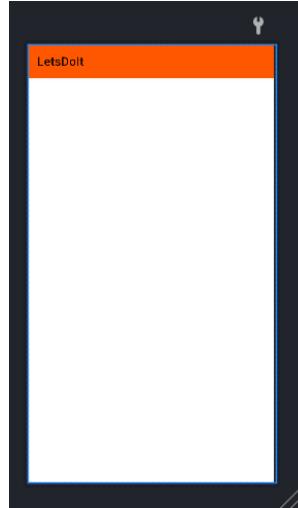
6.2 Función onActivityResult()



El método **onActivityResult()** se sobreescribe entonces para manejar el resultado del proceso de inicio de sesión de Google. Primero verifica si el código de solicitud coincide con el código de solicitud para el proceso de inicio de sesión de Google. Si es así, luego obtiene el resultado del proceso de inicio de sesión utilizando el método **GoogleSignIn.getSignedInAccountFromIntent()** y pasando el objeto Intent de datos.

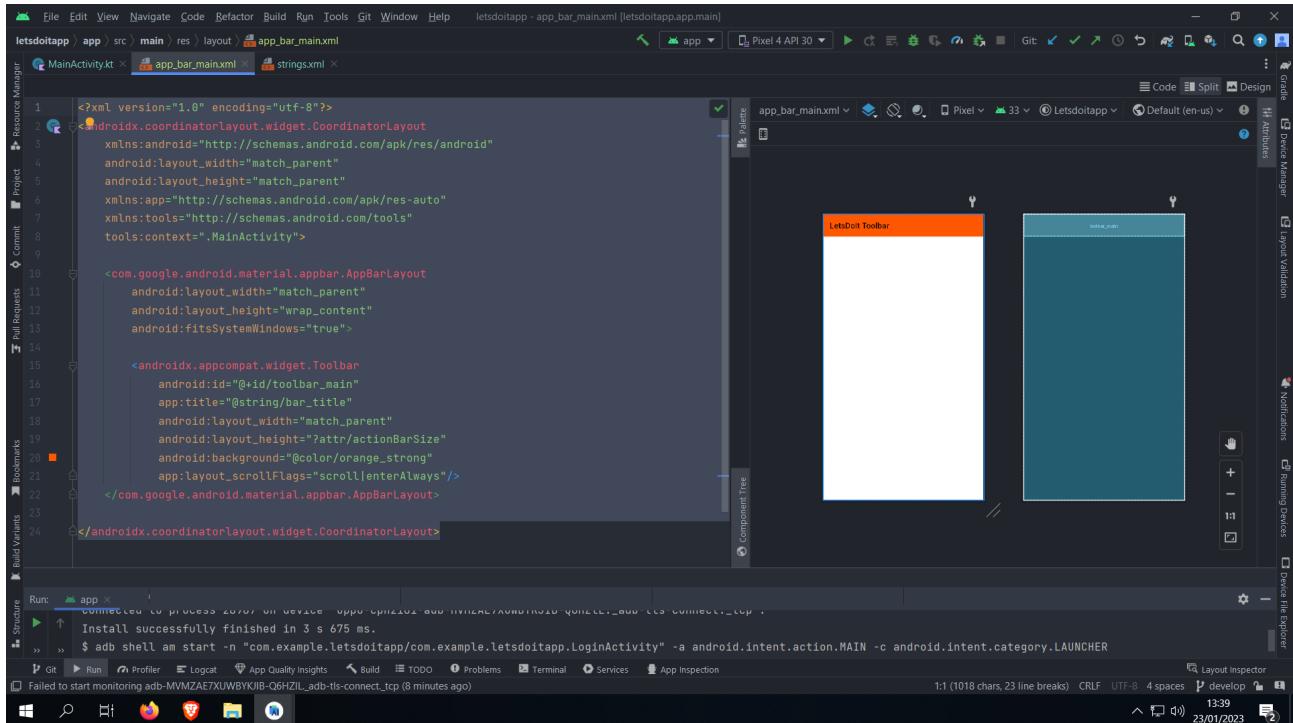
Entonces intenta obtener el objeto de cuenta de la tarea, si el objeto de cuenta no es nulo, obtiene la dirección de correo electrónico del usuario y un token de ID, que se utiliza para autenticar al usuario con Firebase. Si el proceso de inicio de sesión es exitoso, el usuario es redirigido a la pantalla principal de la aplicación, si no, se muestra un mensaje de error indicando que hubo un problema al conectarse con Google.

7. Implementación del Menú



7.1 Menú Toolbar

Aunque en Android Studio ya incluye la posibilidad de crear una Activity con un menú incluido. Este menú es personalizado y se ha hecho desde cero.



archivo app_bar_menu.xml

Este layout utiliza los widgets **CoordinatorLayout**, **AppBarLayout** y **Toolbar** de la biblioteca **AndroidX** para crear un diseño de una **Toolbar**. El diseño está configurado para que su ancho y alto coincidan con los del parent, y el Toolbar tiene como título "**bar_title**", un color de fondo

naranja y se desplaza y siempre entra. El Toolbar también tiene un ID de "toolbar_main" que se puede utilizar para hacer referencia a él en el código.

7.2 Opciones del Menú.

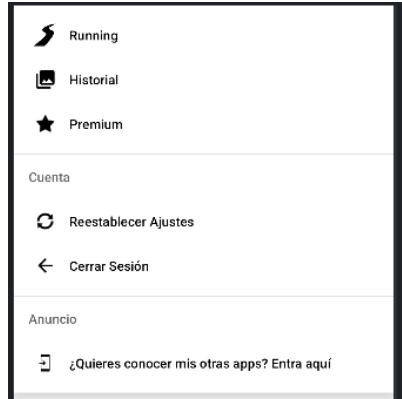


Imagen de las opciones del menú

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group>
        <item android:id="@+id/nav_item_main"
            android:icon="@drawable/camino"
            android:title="@string/menu_running" />
        <item android:id="@+id/nav_item_record"
            android:icon="@drawable/ic_menu_gallery"
            android:title="@string/menu_record" />
        <item android:id="@+id/nav_item_premium"
            android:icon="@drawable/star_on"
            android:title="@string/menu_becomepremium" />
    </group>

    <item android:title="Cuenta">
        <menu>
            <item android:id="@+id/nav_item_clearpreferences"
                android:icon="@android:drawable/ic_popup_sync"
                android:title="@string/menu_clearpreferences" />
            <item android:id="@+id/nav_item_signout"
                android:icon="@drawable/abc_vector_test"
                android:title="@string/menu_signout" />
        </menu>
    </item>
    <item android:title="Anuncio">
        <menu>
            <item android:id="@+id/nav_item_offer"
                android:icon="@drawable/common_full_open_on_phone"
                android:title="@string/menu_offercourse" />
        </menu>
    </item>
</menu>
```

activity_main_drawer.xml

En este layout se definen los elementos para el menú de navegación. Contiene un grupo de elementos con iconos y títulos. Cada elemento tiene un ID único, que se puede utilizar para hacer referencia a él en el código, y un recurso de cadena para el título. Los elementos están agrupados en dos secciones: "Cuenta" y "Anuncio", cada sección tiene uno o más elementos, cada elemento tiene un id único, un ícono y un título. Todos los elementos y las secciones se definen utilizando las etiquetas `<item>` y `<menu>` respectivamente, y todos están anidados dentro de la etiqueta `<menu>` raíz.

7.2.1 Opción del Menú. Autor. Función showAutorInfoAndApps()

```
private fun showAutorInfoAndApps(context: Context) {
    //val appName = context.getString(R.string.app_name)
    val appName = "Let's Do iT"
    val emailAddress = "mundodigital.pro@gmail.com"
    val authorName = "Jose Jordan\n"
    val centro = "IES - Trassierra - Córdoba\n"
    val contact = "<a href='mailto:$emailAddress'>$emailAddress</a>"
    val message = "Hola soy $authorName!\n\n" +
        "He desarrollado $appName en el marco de mi formación de DAM en $centro.\n\n" +
        "Espero que encuentres útil esta app.\n\n" +
        "He intentado que sea fácil de usar.\n\n" +
        "Si tienes algún problema o sugerencia, por favor házmelo saber.\n\n" +
        "Email: $contact\n\n" +
        "¡Gracias por utilizar $appName!"

    // Crea el diálogo con la información del autor
    val dialogBuilder = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {
        AlertDialog.Builder(context).AlertDialogBuilder()
            .setTitle(appName).AlertDialogBuilder!
            //.setMessage("Autor: $authorName")
            //.setMessage(message)
            //.setMessage(Html.fromHtml(message))
            .setMessage(Html.fromHtml(message, Html.FROM_HTML_MODE_LEGACY))
            .setPositiveButton( text: "Aceptar", listener: null)
    } else {
        TODO( reason: "VERSION.SDK_INT < N")
    }
}
```

```
// Agrega el enlace a tus otras aplicaciones en la PlayStore
//val playStoreLink = "https://play.google.com/store/apps/developer?id=TU_NOMBRE_DE_DESARROLLADOR"
val playStoreLink = "https://play.google.com/store/apps/"
dialogBuilder.setNeutralButton( text: "Mis apps") { _, _ ->
    val intent = Intent(Intent.ACTION_VIEW, Uri.parse(playStoreLink))
    context.startActivity(intent)
}

// Agrega un listener al mensaje para abrir la actividad de redacción de correo electrónico de Gmail
val messageView = dialogBuilder.show().findViewById<TextView>(android.R.id.message)
messageView?.movementMethod = LinkMovementMethod.getInstance()
messageView?.setOnClickListener { it: View!
    val intent = Intent(Intent.ACTION_SENDTO)
    intent.data = Uri.parse(uriString: "mailto:$emailAddress")
    intent.putExtra(Intent.EXTRA_SUBJECT, value: "Sugerencias o problemas con la aplicación")
    context.startActivity(Intent.createChooser(intent, title: "Enviar correo electrónico"))
}

// Muestra el diálogo
//dialogBuilder.show()
}
```

Esta función muestra un cuadro de diálogo con información sobre el autor de una aplicación, así como un enlace a las otras aplicaciones del autor en **Google Play Store**. La función toma un objeto Context como argumento, que se utiliza para crear el cuadro de diálogo y comenzar actividades.

La función primero define algunas variables de cadena, incluyendo el nombre de la aplicación ("Let's Do iT"), la dirección de correo electrónico del autor, el nombre del autor y el nombre de la institución educativa donde el autor estudió. Luego crea una cadena de mensaje que incluye esta información y algunos textos adicionales, y lo establece como el mensaje del cuadro de diálogo.

A continuación, la función crea un objeto **AlertDialog.Builder** con el título establecido en el nombre de la aplicación y el mensaje establecido en la cadena de mensaje. También establece un botón neutral etiquetado como "**Mis apps**" que abre la página de **Google Play Store** para las otras aplicaciones del autor cuando se hace clic.

La función luego obtiene una referencia al **TextView** que muestra el mensaje en el cuadro de diálogo y establece un listener de clic en él. Cuando se hace clic en el mensaje, la función abre la pantalla de composición de la aplicación **Gmail** con la dirección de correo electrónico del autor **prellenada** y una línea de asunto que indica que el usuario quiere informar un problema o sugerir una función para la aplicación.

7.2.2 Opción del Menú. Premium. Función alertPremium()

```
private fun alertPremium() {
    if (!isPremium) {
        AlertDialog.Builder(context: this) AlertDialog.Builder!
            .setTitle(" ¿ Quieres ser usuario Premium ? ") AlertDialog.Builder!
            .setMessage(" Al ser Premium no verás publicidad y tendrás opciones extra ")
            .setPositiveButton(
                android.R.string.ok
            ) { _, _ ->
                //botón OK pulsado
                launchPaymentCard()
                //becamePremium()
                showBanner()

            }
            .setNegativeButton(
                android.R.string.cancel
            ) { _, _ ->
                //botón cancel pulsado
            }
            .setCancelable(true)
            .show()
    } else {
        AlertDialog.Builder(context: this) AlertDialog.Builder!
            .setTitle(" Usuario Premium") AlertDialog.Builder!
            .setMessage(" Ya eres usuario Premium y disfrutas de las opciones extras de la app y sin publicidad !!!")
            .setPositiveButton(android.R.string.ok) { _, _ -> }
            .setCancelable(true)
            .show()
    }
}
```

Esta función muestra un cuadro de diálogo para que el usuario decida si quiere convertirse en un usuario Premium o no. La función verifica si el usuario ya es un usuario Premium o no y muestra diferentes mensajes según el estado.

Si el usuario aún no es un usuario Premium, la función muestra un cuadro de diálogo con un título y un mensaje que describen los beneficios de ser un usuario Premium. El cuadro de diálogo tiene un botón "OK" que el usuario puede pulsar si desea convertirse en un usuario Premium. Si el

usuario pulsa el botón "OK", se llama a la función **launchPaymentCard()** para que el usuario pueda realizar un pago y convertirse en usuario Premium. Después de realizar el pago, se llama a la función **becamePremium()** para actualizar el estado del usuario y se muestra un banner.

Si el usuario ya es un usuario Premium, la función muestra un cuadro de diálogo con un mensaje que indica que el usuario ya es Premium y disfruta de las opciones extra de la app sin publicidad. El cuadro de diálogo tiene un botón "OK" que el usuario puede pulsar para cerrar el cuadro de diálogo.

En ambos casos, el cuadro de diálogo es cancelable, lo que significa que el usuario puede cerrarlo haciendo clic fuera del cuadro de diálogo o pulsando el botón de "volver" en el dispositivo.

7.2.3 Opción del Menú. Premium. Función launchPaymentCard()

```
private fun launchPaymentCard() {
    val intent = Intent(this, CheckoutActivity::class.java)
    startActivity(intent)
}
```

Esta función inicia una actividad llamada **ChecoutActivity**. La actividad **ChecoutActivity** está diseñada para permitir al usuario realizar un pago para convertirse en un usuario Premium.

La función crea un objeto Intent que especifica que la actividad **ChecoutActivity** debe iniciarse. Luego, llama al método `startActivity()` del objeto Intent para iniciar la actividad **ChecoutActivity**. Cuando se inicia la actividad, el usuario debería ver una pantalla para realizar el pago y convertirse en usuario Premium.

7.2.4 Opción del Menú. Premium. Función becamePremium()

```
fun becamePremium() {//GUARDA EL USUARIO PREMIUM
    editor.apply {
        putBoolean(key_premium, true)
    }.apply()
}
```

Esta función actualiza el estado del usuario a **Premium**. La función utiliza el objeto **SharedPreferences** para guardar el estado del usuario en una clave booleana llamada "**key_premium**" en el archivo de preferencias.

Primero, la función obtiene una referencia al objeto **SharedPreferences.Editor** a través del objeto **SharedPreferences**. Luego, utiliza el método `putBoolean()` del objeto Editor para establecer el valor de "key_premium" en "true". Finalmente, llama al método `apply()` del objeto Editor para guardar los cambios en el archivo de preferencias.

En resumen, esta función es responsable de guardar el estado del usuario como Premium en el archivo de preferencias.

7.2.5 Opción del Menú. Premium. CheckoutActivity

```
JordanPro

class CheckoutActivity : AppCompatActivity() {
    private lateinit var paymentButtonContainer: PaymentButtonContainer
    private lateinit var lyCheckout: LinearLayout
    //private var checkPremium: Boolean = false

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_checkout)

        lyCheckout = findViewById(R.id.activity_checkout)
        paymentButtonContainer = findViewById(R.id.payment_button_container)
        paymentButtonContainer.setup(
            createOrder =
            CreateOrder { createOrderActions ->
                val order =
                    Order(
                        intent = OrderIntent.CAPTURE,
                        applicationContext = ApplicationContext(userAction = UserAction.PAY_NOW),
                        purchaseUnitList =
                        listOf(
                            PurchaseUnit(
                                amount =
                                Amount(currencyCode = CurrencyCode.USD, value = "5.00")
                        )
                )
                createOrderActions.create(order)
            }
        }
    }
}
```

```
        createOrderActions.create(order)
    },
    onApprove =
    OnApprove { approval ->
        approval.orderActions.capture { captureOrderResult ->
            //here you will get the result (Success)
            Log.i( tag: "CaptureOrder", msg: "CaptureOrderResult: $captureOrderResult")
            showCustomSnackbar(lyCheckout, message: "Resultado del Pedido: $captureOrderResult", R.color.orange_strong, duration: 10000)
            MainActivity.becamePremium()
            //checkPremium = true
        }
    },
    onCancel = OnCancel {
        Log.d( tag: "OnCancel", msg: "Buyer canceled the PayPal experience.")
        showCustomSnackbar(lyCheckout, message: "Se ha cancelado el pago por PayPal", R.color.orange_strong, duration: 8000)
        //checkPremium = false
    },
    onError = OnError { errorInfo ->
        Log.d( tag: "OnError", msg: "Error: $errorInfo")
        showCustomSnackbar(lyCheckout, message: "Error: $errorInfo", R.color.orange_strong, duration: 8000)
    }
} //Setup
}
```

Esta actividad utiliza la biblioteca de **PayPal Checkout** para procesar un pago y capturar el resultado del pedido.

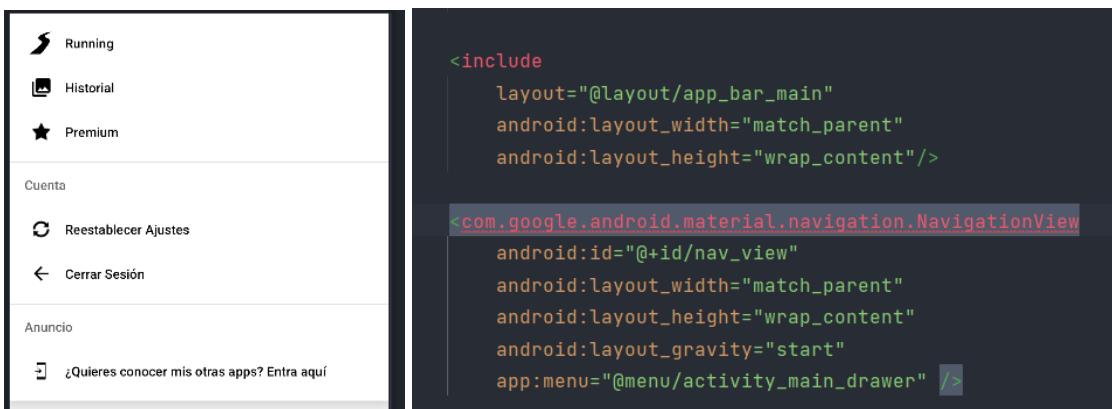
La actividad **CheckoutActivity** extiende la clase `AppCompatActivity` y tiene un método `onCreate()` que se llama cuando se crea la actividad. En el método `onCreate()`, la actividad infla su diseño a partir del archivo XML `activity_checkout.xml` y configura el botón de pago de PayPal.

La actividad crea una referencia a **PaymentButtonContainer** y `LinearLayout` en la que se mostrará el botón de pago. Luego, utiliza el método `setup()` del objeto **PaymentButtonContainer** para configurar el botón de pago. En el método `setup()`, se establece una acción para crear un pedido de PayPal utilizando la clase **CreateOrder**. El pedido se crea con un objeto `Order` que especifica la intención de capturar el pago y la unidad de compra con un valor de **USD 5.00**.

La actividad también define los manejadores de eventos para los casos en que el usuario aprueba el pago (**OnApprove**), cancela el pago (**OnCancel**) o hay algún error (**OnError**). El manejador de evento **OnApprove** llama al método `capture()` del objeto **OrderActions** para capturar el resultado del pedido. Si la captura es exitosa, se muestra un mensaje en la pantalla y se llama a la función `becamePremium()` de `MainActivity` para actualizar el estado del usuario. Si hay un error o el usuario cancela el pago, se muestra un mensaje de error o cancelación en la pantalla.

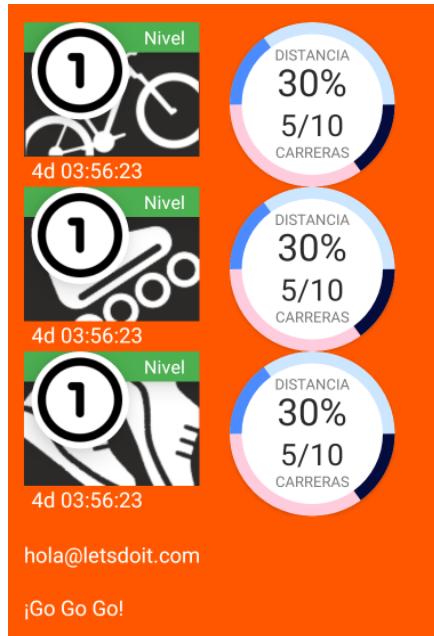
En resumen, la actividad **CheckoutActivity** utiliza la biblioteca de PayPal Checkout para procesar un pago y actualizar el estado del usuario.

7.3 Implementación del Menú NavigationView

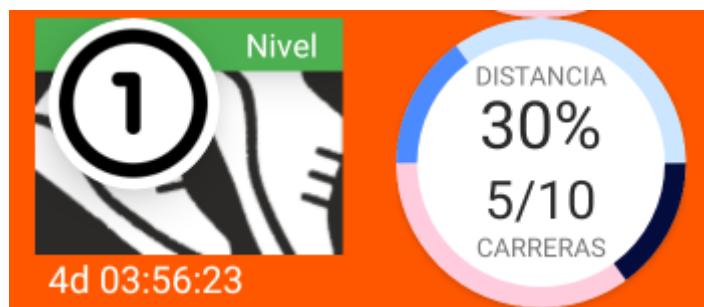


Este layout XML incluye el diseño `app_bar_main` y un **NavigationView** va incluido dentro del layout del Main. La etiqueta **include** se utiliza para incluir otro archivo de diseño, en este caso el diseño `app_bar_main`. El **NavigationView** es un widget proporcionado por la biblioteca de diseño **Android Material** que crea un menú de navegación, que a menudo se coloca en un diseño de cajón (drawer). El ID del **NavigationView** se establece en "`nav_view`" y el ancho y la altura se establecen para que coincidan con el padre. La gravedad del diseño se establece en "`start`" para que el **NavigationView** aparezca al principio del diseño. El atributo `app:menu` se establece en "`activity_main_drawer`" que hace referencia a un archivo de recurso de menú que define los elementos que se deben mostrar en el menú de navegación.

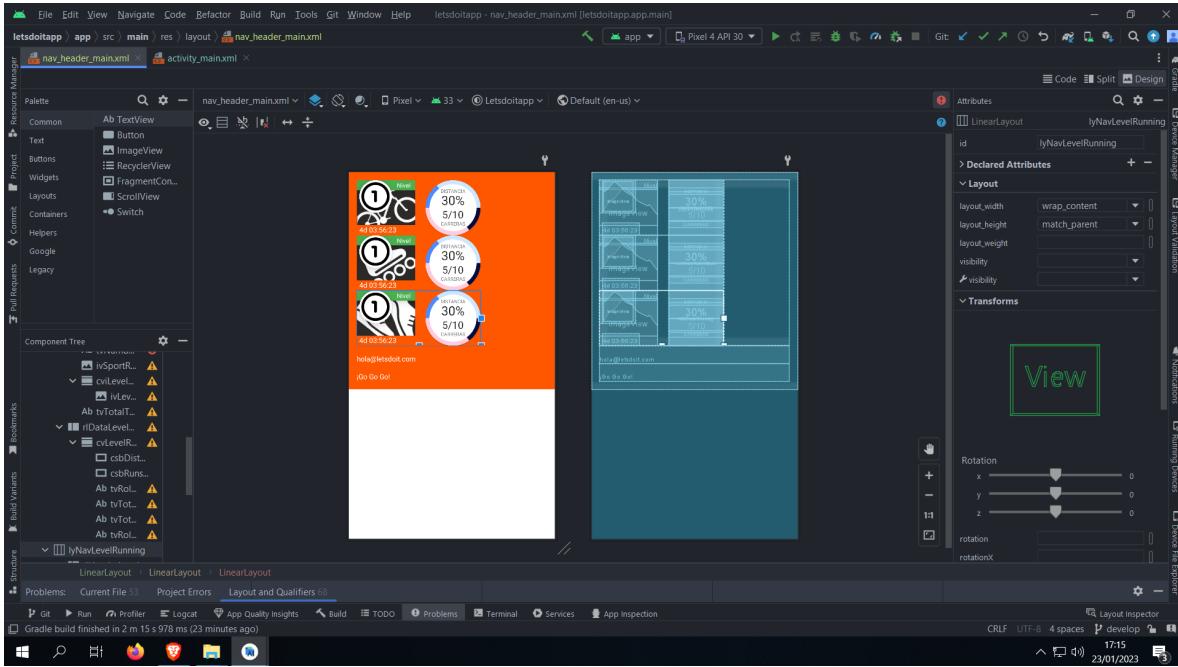
7.4 Diseño de la Cabecera del Menú



Cada usuario entra con su correo y aquí en la cabecera se muestra el email del usuario. También se muestra una frase motivadora “Go, Go , Go!”. En un futuro, esta característica se podría implementar y esta frase podría cambiar cada día . Se muestran datos del usuario, registros de cada actividad que haya practicado. Cada uno de estos deportes puede tener diferentes niveles. En un círculo con un número se muestra el nivel que tiene. La imagen identifica el deporte y el total de tiempo acumulado de ese deporte.



Distancia y Carreras. En el círculo hay indicadores en modo texto y de forma gráfica en colores , se indica también el progreso de la distancia y las carreras que lleva el usuario . Por ejemplo , lleva “30%” en Azul y en azul más claro es lo que le falta para pasar al siguiente nivel, cuando la distancia llegue al 100% y haya hecho el número de carreras entonces pasará al siguiente nivel. Luego al usuario solo se le mostrará información de los deportes que haya practicado.

**nav_header_main.xml**

El diseño del header tiene varios elementos anidados, LinearLayout, RelativeLayout, TextView, ImageView y CardView. Cada uno tiene un ancho y alto específico y con atributos de estilo adicionales. Además se utiliza un elemento **CircularSeekBar**. En resumen, esta función se utiliza para configurar el encabezado de navegación con el correo electrónico del usuario y también para establecer el escucha de los elementos de navegación, de manera que el escucha se notificará cuando se seleccione un elemento.

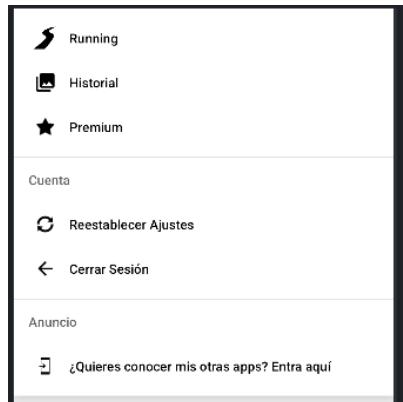
7.5 Añadiendo funcionalidad al menú

7.5.1 Función initToolbar

```
private fun initToolBar() {
    val toolbar: androidx.appcompat.widget.Toolbar = findViewById(R.id.toolbar_main)
    setSupportActionBar(toolbar)
    drawer = findViewById(R.id.drawer_layout)
    val toggle = ActionBarDrawerToggle(
        activity: this,
        drawer,
        toolbar,
        R.string.bar_title,
        R.string.navigation_drawer_close
    )
    drawer.addDrawerListener(toggle)
    toggle.syncState()
}
```

La función **initToolBar** establece la barra de herramientas de la actividad, establece la barra de acción de soporte en la barra de herramientas, inicializa un **DrawerLayout**, crea un **ActionBarDrawerToggle** y agrega el toggle como un escucha al **DrawerLayout**. El **ActionBarDrawerToggle** se utiliza para controlar el estado de apertura y cierre del cajón en conjunto con la barra de herramientas. El método **syncState** del toggle sincroniza el estado del toggle con el estado del cajón.

7.5.2 Función initNavigationView

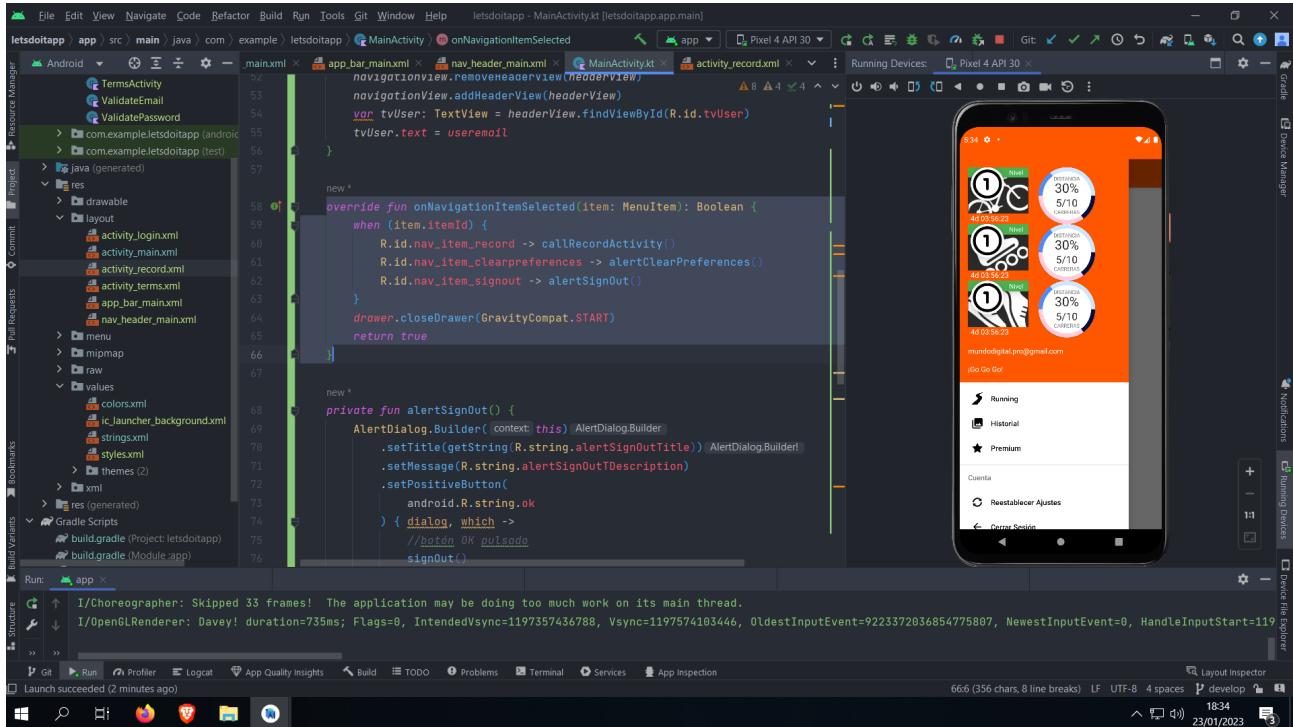


```
new *

private fun initNavigationView() {
    var navigationView: NavigationView = findViewById(R.id.nav_view)
    navigationView.setNavigationItemSelectedListener(this)
    var headerView: View = LayoutInflater
        .from(context)
        .inflate(R.layout.nav_header_main, navigationView, attachToRoot: false)
    navigationView.removeHeaderView(headerView)
    navigationView.addHeaderView(headerView)
    var tvUser: TextView = headerView.findViewById(R.id.tvUser)
    tvUser.text = useremail
}
```

La función **initNavigationView**, se utiliza para inicializar y personalizar la **NavigationView**.

- Primero, obtiene una referencia a la **NavigationView** en el diseño llamando al método **findViewById**, pasando el id de la vista.
- Luego, establece un escucha para la **NavigationView** llamando a **setNavigationItemSelectedListener** y pasando la actividad actual como el escucha.
- Después de eso, infla el diseño **nav_header_main** y lo agrega como una vista de encabezado para la **NavigationView**.
- Finalmente, obtiene una referencia al **TextView** dentro de la vista de encabezado llamando a **findViewById** y establece su texto con el valor de la variable **useremail**.



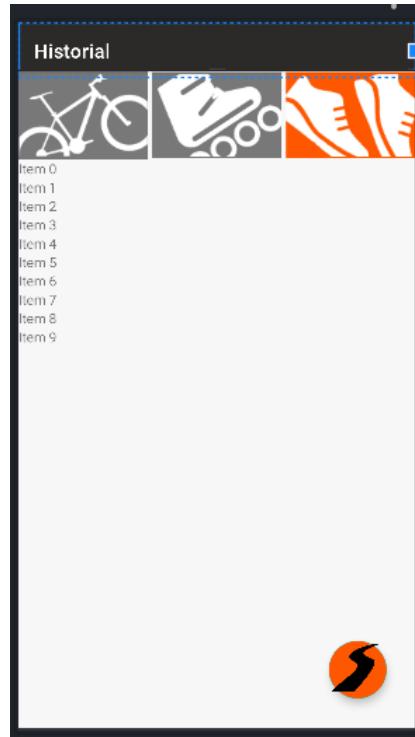
7.5.3 Función onNavigationItemSelected

La función "**onNavigationItemSelected**" toma como parámetro un objeto "**MenuItem**".

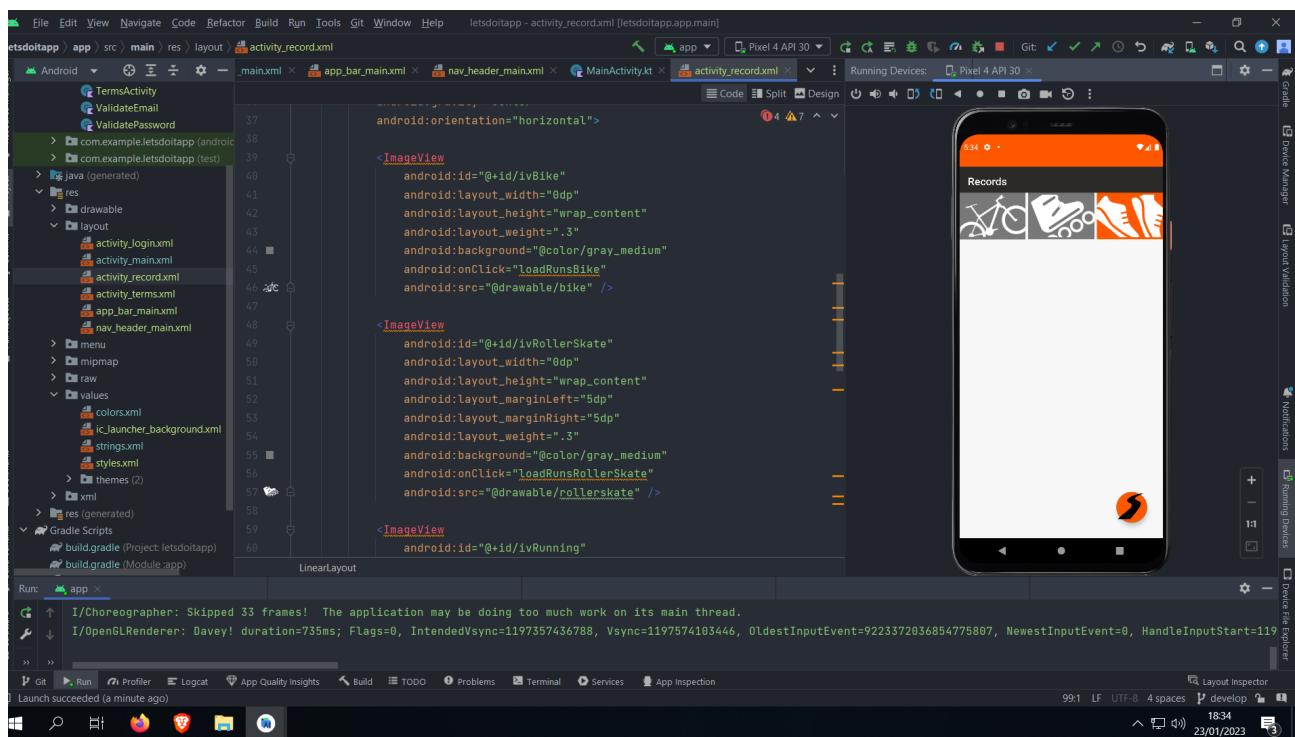
Dentro de la función, se utiliza una estructura "**when**" para evaluar el ID del elemento seleccionado en el menú. Si el ID es "**R.id.nav_item_record**", se llama a una función llamada "**callRecordActivity()**". Si el ID es "**R.id.nav_item_clearpreferences**", se llama a una función llamada "**alertClearPreferences()**". Y si el ID es "**R.id.nav_item_signout**", se llama a una función llamada "**alertSignOut()**".

Finalmente, se cierra el menú de navegación y se devuelve "true" para indicar que el evento fue manejado correctamente.

7.6 Diseño de la pantalla del Historial



En esta pantalla diseñada en una Activity podemos ver el registro de las carreras clasificadas por cada deporte y podemos seleccionar cómo queremos que se ordenen. Por fecha, distancia, velocidad de mayor a menor. Ese menú saldría de la derecha de la barra ToolBar.

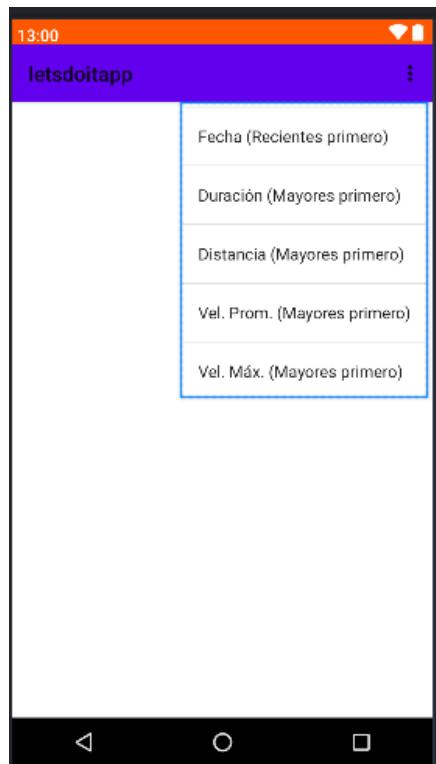


archivo `activity_records.xml`

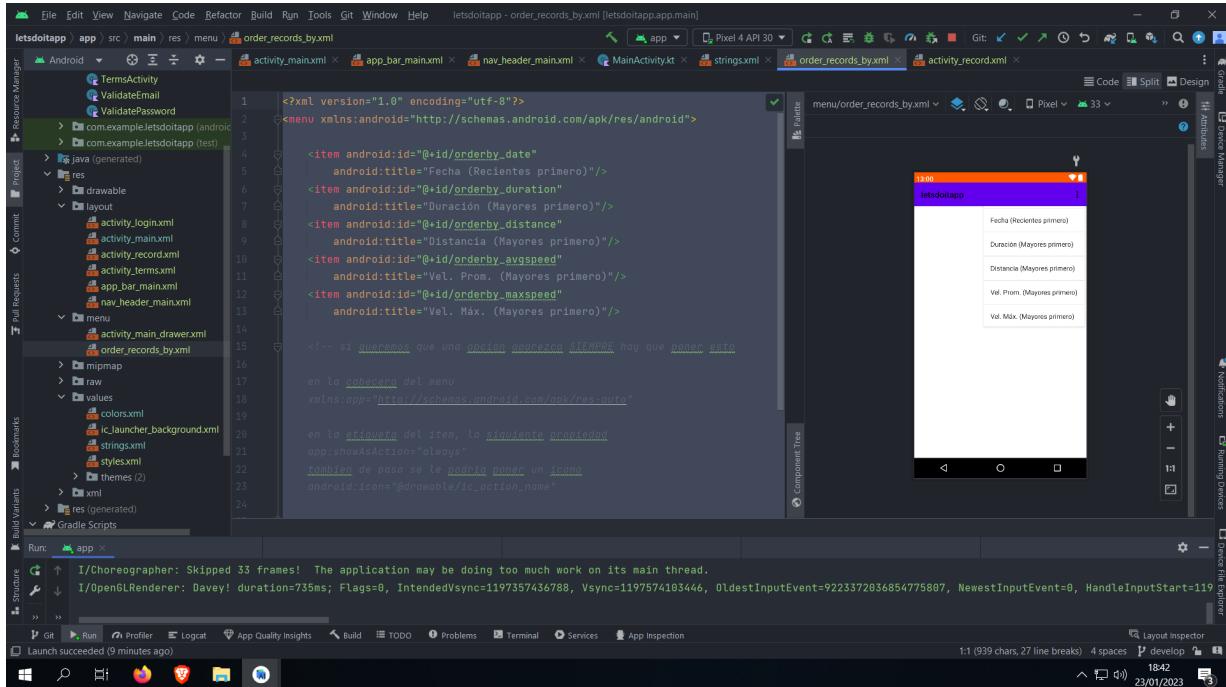
Este layout XML utiliza varios elementos de diseño **LinearLayout** y **AppBarLayout**, así como herramientas de la biblioteca de material design de Google como Toolbar y

FloatingActionButton. El layout tiene un fondo blanco y establece varios elementos dentro de él, como una barra de herramientas con un título y un botón flotante en la parte inferior derecha. También tiene varias imágenes que parecen ser iconos de deportes y un **RecyclerView**. Estas imágenes tienen eventos **onClick** que cargan funciones específicas en el código.

7.6.1 Diseño del Menú Emergente en Toolbar



Este menú XML en Android define varios elementos de menú, cada uno con un identificador único y un título. Los títulos de cada elemento de menú se refieren a un recurso de cadena en el archivo de recursos de la aplicación. Estos elementos de menú se utilizarán para ordenar diferentes elementos en la aplicación. El comentario en el código también sugiere la posibilidad de mostrar un elemento de menú siempre en la interfaz de usuario.



archivo order_records_by.xml

7.6.2 Función initToolbar

```
private fun initToolbar() {
    val toolbar: androidx.appcompat.widget.Toolbar = findViewById(R.id.toolbar_record)
    setSupportActionBar(toolbar)
    toolbar.title = getString(R.string.bar_title_record)
    toolbar.setTitleTextColor(ContextCompat.getColor(context: this, R.color.gray_dark))
    toolbar.setBackgroundColor(ContextCompat.getColor(context: this, R.color.gray_light))
    supportActionBar?.setDisplayHomeAsUpEnabled(true)
    supportActionBar?.setHomeButtonEnabled(true)
}
```

La función **initToolbar** se encarga de inicializar la barra de herramientas. La función comienza declarando una variable "toolbar" de tipo **androidx.appcompat.widget.Toolbar** y asignándole el valor de una vista con el id "**toolbar_record**" que se encuentra en el archivo de diseño de la actividad. Luego, se establece la toolbar como la barra de herramientas de la actividad con el método **setSupportActionBar(toolbar)**. Sigue asignando un título a la toolbar, tomando el texto del recurso strings.xml con la función **getString(R.string.bar_title_record)** y cambiando el color del título y del fondo de la barra. Finalmente, se habilita la opción de "**ir hacia atrás**" en la barra de herramientas con **supportActionBar?.setDisplayHomeAsUpEnabled(true)** y se habilita el botón de "**ir hacia atrás**" con **supportActionBar?.setHomeButtonEnabled(true)**.

7.6.3 Función onOptionsItemSelected

```

<string name="orderby_dateZA">Fecha (Recientes primero)</string>
<string name="orderby_durationZA">Duración (Mayores primero)</string>
<string name="orderby_distanceZA">Distancia (Mayores primero)</string>
<string name="orderby_avgspeedZA">Vel. Prom. (Mayores primero)</string>
<string name="orderby_maxspeedZA">Vel. Máx. (Mayores primero)</string>
|
<string name="orderby_dateAZ">Fecha (Antiguos primero)</string>
<string name="orderby_durationAZ">Duración (Menores primero)</string>
<string name="orderby_distanceAZ">Distancia (Menores primero)</string>
<string name="orderby_avgspeedAZ">Vel. Prom. (Menores primero)</string>
<string name="orderby_maxspeedAZ">Vel. Máx. (Menores primero)</string>

```

En el archivo strings.xml tenemos las cadenas de texto que se van a mostrar en el menú de la toolbar del Historial. Estas cadenas las vamos a cambiar cada vez que en el menú hacemos click en una de ellas. Cargaremos los datos con el parámetro que se haya elegido y la frase cambiará. Por ejemplo si pulsas en ver “Distancia Mayores Primero”, la siguiente vez tendrá que ser al revés cambiando a ver “Distancia Menores Primero”. Así que cada vez que se haga clic en una de las opciones , esta opción se cambiará por su frase opuesta.

```

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    var order: Query.Direction = Query.Direction.DESCENDING
    when (item.itemId) {
        R.id.orderby_date -> {
            if (item.title == "Fecha (Recientes primero)") {
                item.title = "Fecha (Antiguos primero)"
                order = Query.Direction.DESCENDING
            } else {
                item.title = "Fecha (Recientes primero)"
                order = Query.Direction.ASCENDING
            }
            loadRecyclerView(field: "date", order)
            return true
        }
    }
}

```

Esta función es la que se ejecuta cuando se selecciona una opción en el menú de opciones de nuestra app. La función es una sobreescritura (override) del método **onOptionsItemSelected** de la clase Activity.

La función comienza declarando una variable "**order**" de tipo **Query.Direction**, que es inicializada con el valor **Query.Direction.DESCENDING**.

Luego, hay una estructura de control "**when**" que se encarga de determinar qué opción del menú fue seleccionada. Dependiendo del id del elemento del menú seleccionado (**item.itemId**), se realizan diferentes acciones.

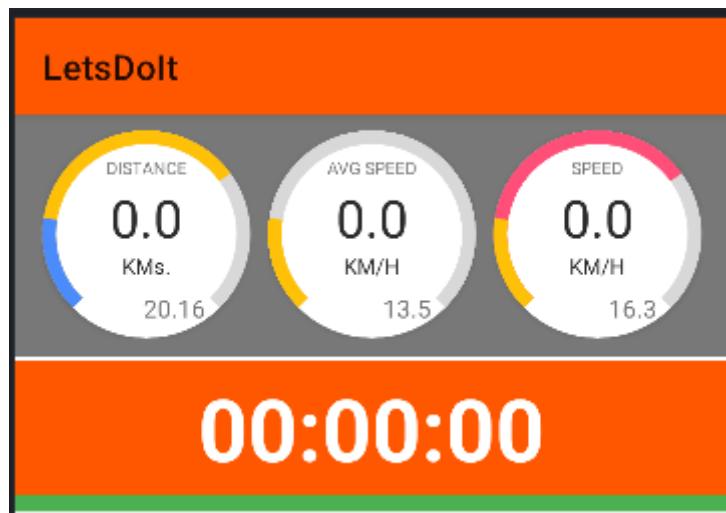
Para cada opción del menú, se verifica si el título del elemento del menú es igual al título de un recurso de cadena específico (**getString(R.string.orderby_dateZA)**), y si lo es, cambia el título del elemento del menú y la variable "order" a la opuesta (**ASCENDING** o **DESCENDING**).

Finalmente, se llama a una función "**loadRecyclerView**" pasándole el nombre del campo y la orden a utilizar.

La función devuelve "true" si se ha procesado la opción del menú y **"super.onOptionsItemSelected(item)"** si no se ha procesado la opción del menú.

8. Diseño del Menú Principal

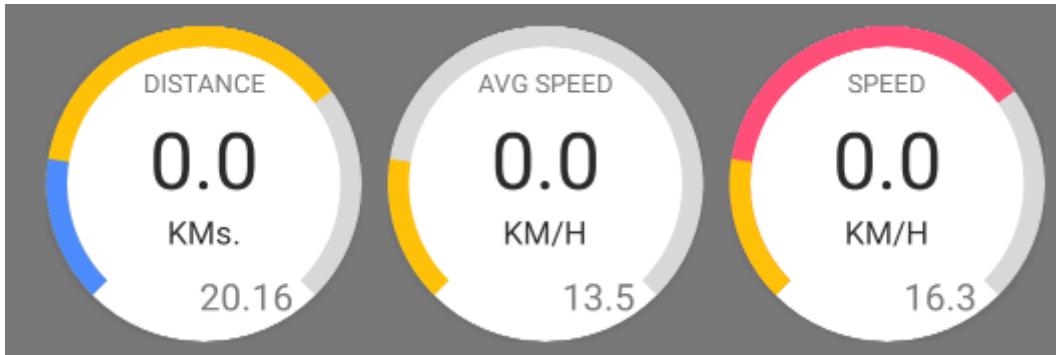
8.1 Elementos del menú fijos



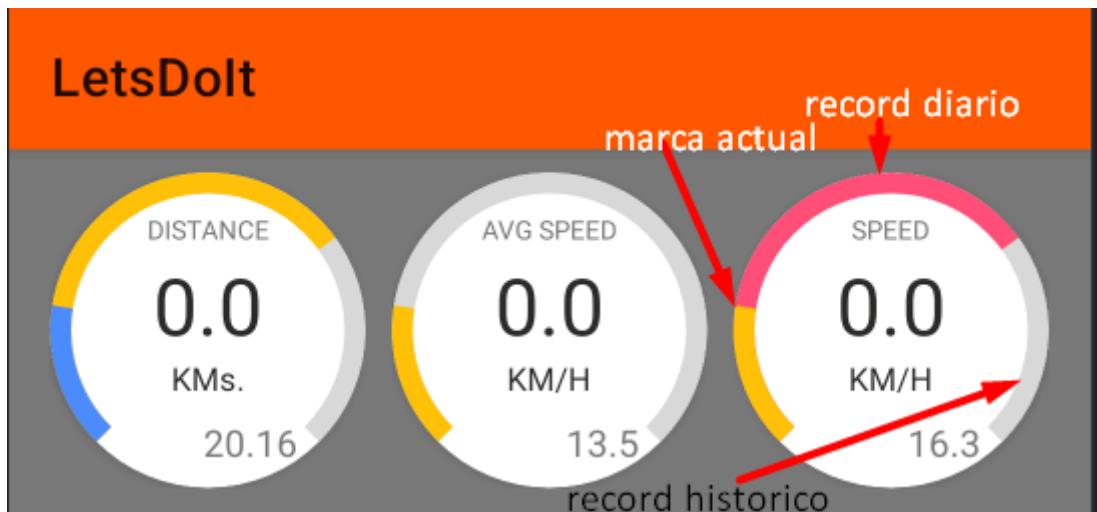
Esta parte del layout la vamos a tener siempre visible.

Para establecer el diseño principal del Main se utiliza un "**RelativeLayout**" como contenedor principal y dentro tenemos varios Layouts anidados.

8.2 Marcadores



En la parte de arriba tenemos un layout fijo que contiene varios layout mostrando información sobre la distancia, velocidad media y velocidad actual.



En este caso se muestra un **LinearLayout** donde hay tres “**CardView**” que contienen varios “**CircularSeekBar**” que se utilizan para mostrar una barra de progreso circular. Cada “**CircularSeekBar**” tiene diferentes colores de progreso (azul, naranja y gris). Los tres están configurados para no mostrar un puntero, ya que en realidad son **ProgressBar**. Los “**CardView**” se utilizan para albergar la información y dar un borde redondeado a los tres **CircularSeekBar**.



Dentro de cada **CardView** tenemos varios **Texviews** que muestran Distancia, Velocidad media y Velocidad Máxima junto con sus récords históricos.

8.3 Cronómetro



En este layout tenemos el diseño de un **cronómetro**, para su diseño se utiliza un elemento **"RelativeLayout"** como contenedor principal, con un tamaño de ancho de "match_parent" (se ajusta al ancho de la pantalla) y un tamaño de alto de 87dp. El RelativeLayout tiene un margen superior de 198dp y un fondo de color gris oscuro. Dentro del RelativeLayout hay tres elementos **"LinearLayout"** utilizados para mostrar el progreso del cronómetro:

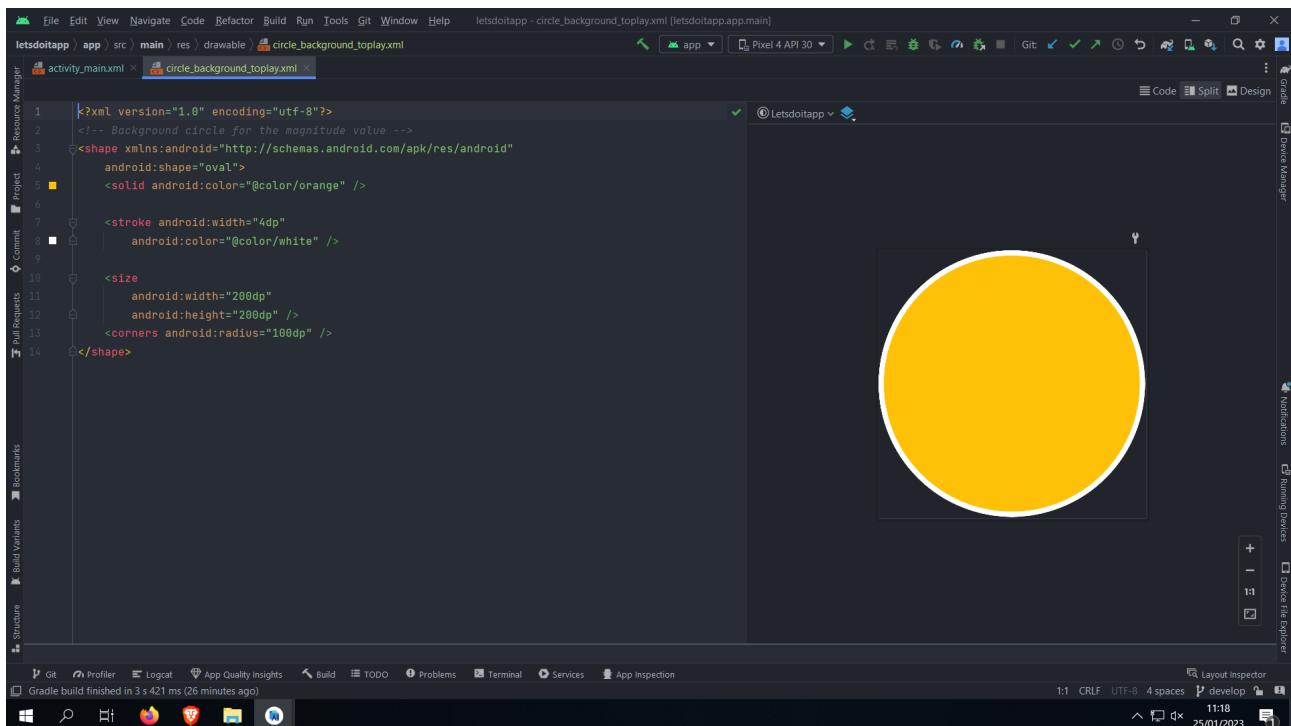
- El primer "LinearLayout" tiene un fondo de color naranja fuerte y se utiliza como fondo para el progreso del cronómetro.
- El segundo "LinearLayout" tiene un fondo verde y se utiliza para mostrar el progreso de las rondas.
- El tercer "LinearLayout" contiene dos "TextView" uno muestra el tiempo del cronómetro en un formato de texto y otro muestra el número de rondas. Los dos TextView tienen un tamaño de letra grande y están alineados en el centro del LinearLayout.

8.4 Elementos del menú móviles

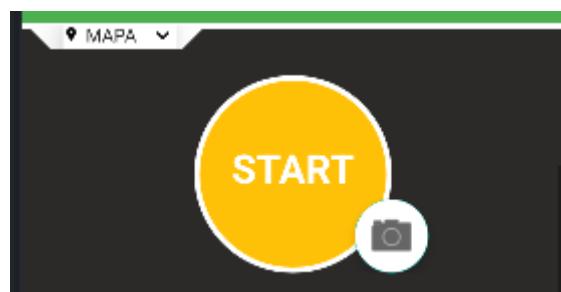
Dentro de un elemento ScrollView se incorporan elementos que se verán bajando o subiendo la pantalla. Todos los elementos dentro del ScrollView se van a organizar con LinearLayouts ya que están uno debajo de otro.

8.4.1 Mapa

8.4.2 Botón Start / Stop



Este botón se ha hecho en código XML. Se utiliza para establecer un diseño para un elemento de forma circular que se usará como botón. La forma es un círculo (`shape="oval"`) y tiene un color de fondo sólido de color naranja. También tiene un borde blanco con un ancho de 4dp. El tamaño del círculo es de 200dp x 200dp y tiene un radio de esquina de 100dp.



Resultado final con un texto y un Floating Action button

8.4.3 Botón Cámara Floating Action Button

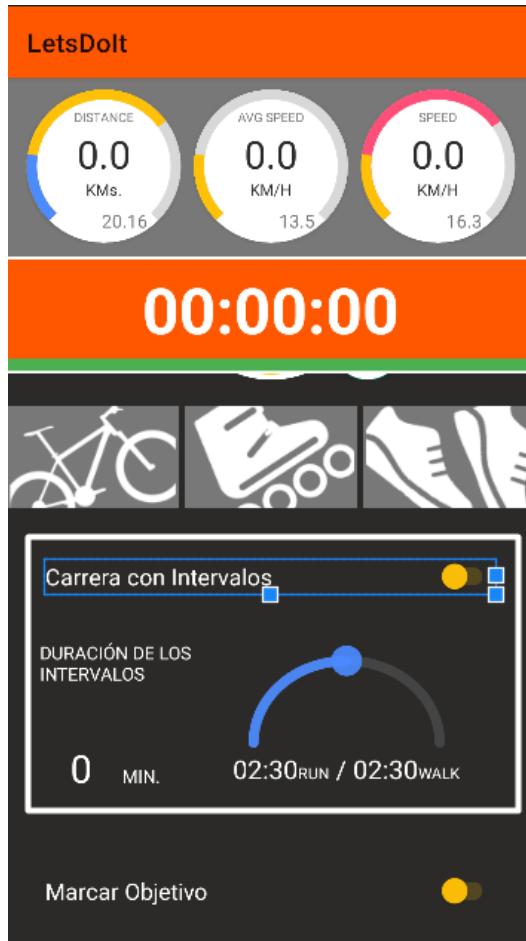
Se utiliza un elemento "**FloatingActionButton**" de la biblioteca de material de diseño de Google. El botón tiene un tamaño "wrap_content" y se ubica en la parte inferior del diseño (layout_alignParentBottom="true") y en el centro del diseño (layout_centerInParent="true"). El botón tiene un fondo blanco y un ícono de cámara (src="@android:drawable/ic_menu_camera") y se encuentra con un tamaño de imagen de 40dp. El botón se encuentra con una traducción de 12dp en el eje Z y 75dp en el eje X. También se agrega una función onClick llamada "takePicture" que se activa al presionar el botón.

8.4.4 Layout Elección de Actividad



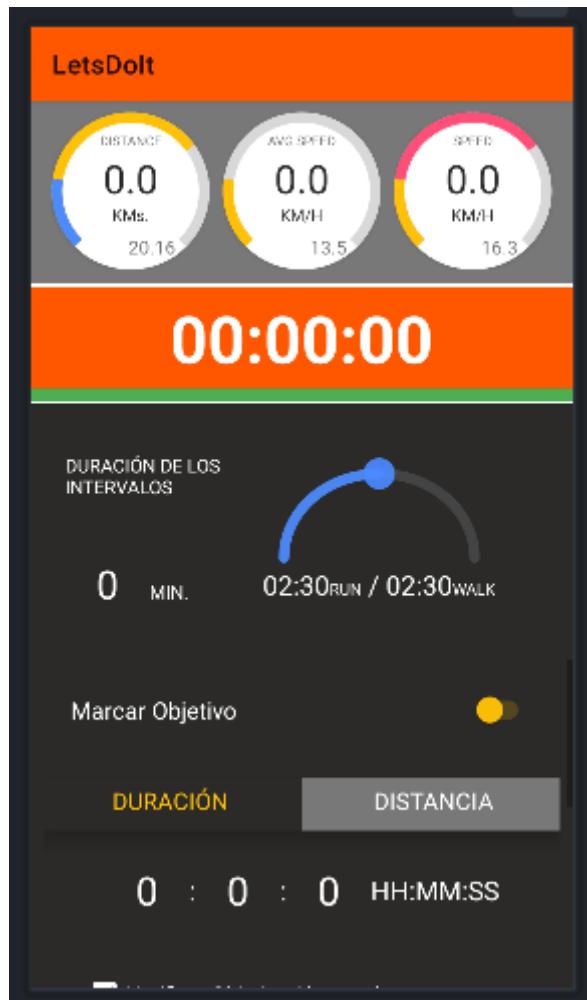
Este es un diseño de pantalla con un LinearLayout que contiene tres LinearLayout hijos. Cada uno de estos hijos contiene un ImageView y un fondo de color gris. Los tres LinearLayout hijos tienen diferentes imágenes y una función onClick diferente. Los tres LinearLayout hijos tienen un tamaño proporcional y están dispuestos horizontalmente. El LinearLayout padre tiene un fondo de color gris oscuro y está centrado. El layout tiene un margen superior de 20dp.

8.4.5 Carrera con Intervalos



Para crear la opción “**Carrera con Intervalos**” se utiliza un switch. En este layout mediante un elemento **NumberPicker** y un **CircularProgressBar** también podemos indicar el tiempo total de la actividad que vamos a desarrollar y el tiempo que vamos a utilizar para correr y el tiempo para caminar .

8.4.6 Marcar Objetivo

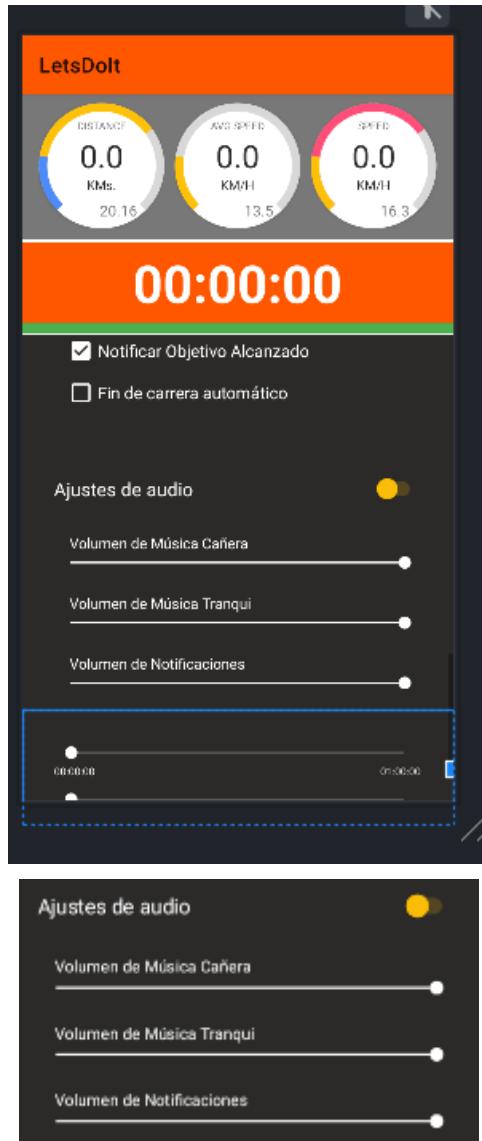


Mediante el switch “Marcar Objetivo” tenemos las opciones de elegir la duración y la distancia que queremos recorrer.



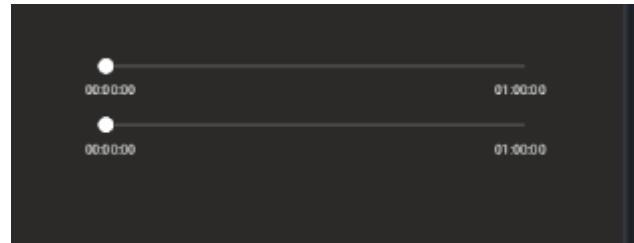
Mediante los Number Pickers escogemos el tiempo que queremos desarrollar la actividad en horas, minutos y segundos. Tenemos varios checkboxes para activar notificaciones o fin de carrera automático.

8.4.7 Ajustes de Audio



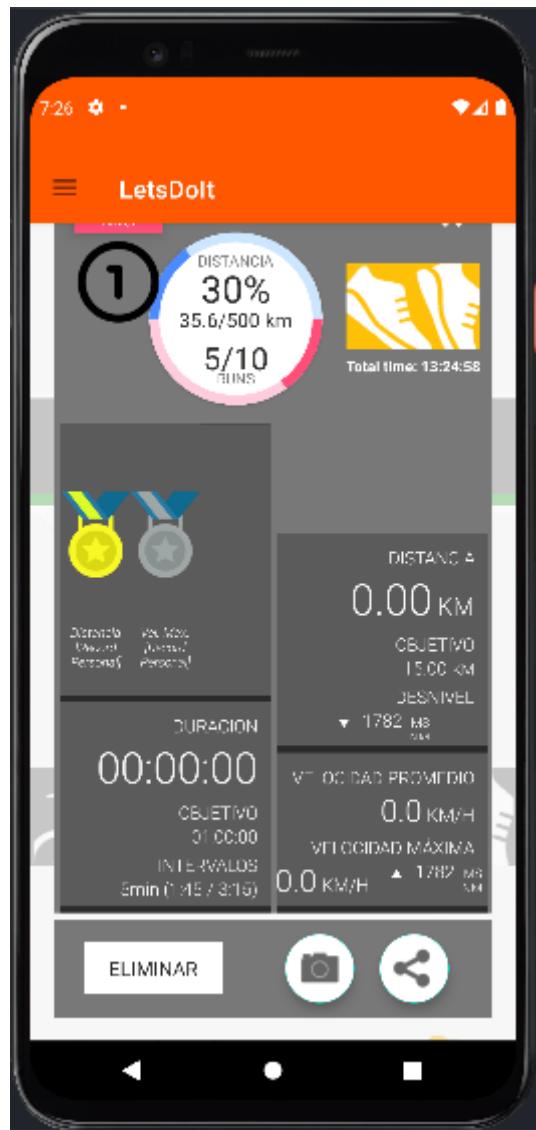
Activando el switch de ajustes de audio tenemos la opción de cambiar el volumen de la música que elijamos y de las notificaciones. Cada ajuste de audio está compuesto de un TextView y un Seekbar.

8.4.7 Pistas de audio



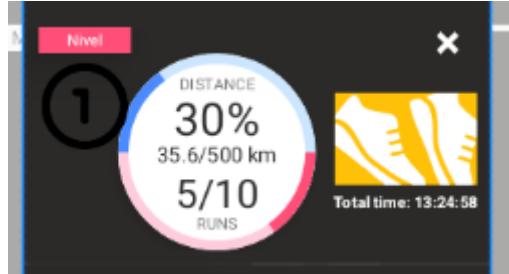
Tenemos dos Seekbar que marcan el tiempo de la pista de audio

8.5 Pantalla de Resultados PopUp



Pantalla PopUp de Resultados

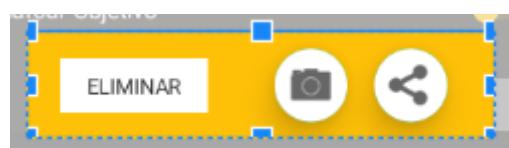
Esta pantalla se muestra cuando finaliza la carrera, pero mientras tanto permanece en el Menú Principal, está oculta por defecto, solo la mostraremos cuando la carrera haya finalizado, entonces en ésta pantalla mostrará todos los resultados de la carrera, progreso, nivel , distancia, tiempo total etc.



En esta parte de la pantalla mostramos un gráfico con el nivel actual del deporte que se ha elegido y un resumen de los totales con las marcas conseguidas en la actividad que se ha desarrollado.



Layout de la carrera actual está compuesta por varios bloques. Por un lado tenemos las medallas de distancia , velocidad media y velocidad máxima. Por otro lado la duración , distancia y velocidad media.



En la parte de abajo de este layout tenemos un pequeño menú de varias opciones con varios botones que nos permiten eliminar el registro, hacer una captura de la pantalla o bien compartir la captura de la pantalla en otras aplicaciones.

9. Animaciones

Las animaciones dentro del menú hacen que se desplieguen opciones y se hagan visibles o invisibles , dependiendo de lo que queremos mostrar en pantalla, como en caso anterior del menú PopUp. Para ello los elementos tienen que tener un estado inicial , en este caso deben de estar ocultos para mostrarlos cuando sean necesarios. Las animaciones pueden consistir en un cambio de tamaño, propiedad, color etc.

9.1 ViewBinding

Para trabajar con las vistas y realizar cambios se implementa **ViewBinding**. Es una característica de Android que permite acceder a los elementos de la interfaz de usuario de una actividad o fragmento de manera más rápida y segura. En lugar de usar el método **findViewById** para buscar una vista en el layout, se genera una clase que contiene referencias directas a las vistas. Esto mejora el rendimiento y reduce la posibilidad de errores debido a que los nombres de las vistas ya están verificados en tiempo de compilación.

9.2 Función de cambio de atributos de un layout

```
fun setHeightLinearLayout(lv: LinearLayout, value: Int) {  
    val params: LinearLayout.LayoutParams = lv.layoutParams as LinearLayout.LayoutParams  
    params.height = value  
    lv.setLayoutParams(params)  
}
```

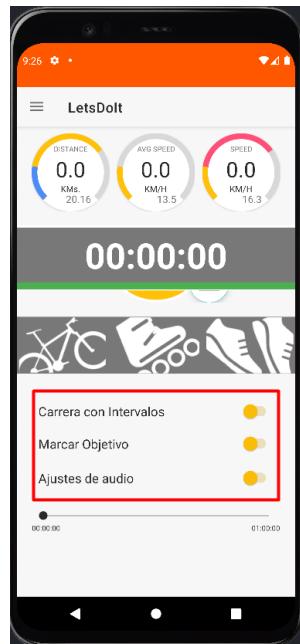
Función **setHeightLinearLayout** .Dentro del menú se le va a dar un estado inicial a ciertos elementos del menú, para ello se programa una función de cambio de atributos que establece la altura de un LinearLayout específico (dado por el parámetro "lv") a un valor específico (dado por el parámetro "value"). Primero, se obtienen los parámetros actuales del LinearLayout y se asignan a una variable "params". Luego se establece la altura de esa variable a el valor dado, y finalmente se establecen estos parámetros actualizados en el LinearLayout especificado.

9.3 Función de ocultación de layouts

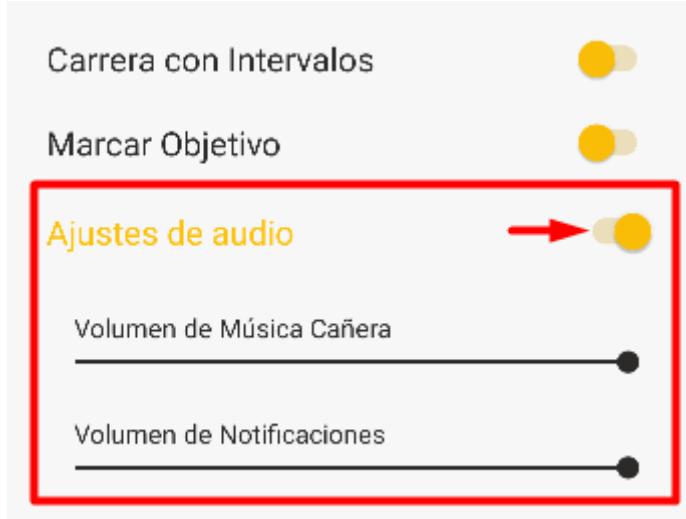
```
private fun hideLayouts() {
    setHeightLinearLayout(binding.lyMap, value: 0)
    setHeightLinearLayout(binding.lyIntervalModeSpace, value: 0)
    setHeightLinearLayout(binding.lyChallengesSpace, value: 0)
    setHeightLinearLayout(binding.lySettingsVolumesSpace, value: 0)
    setHeightLinearLayout(binding.lySoftTrack, value: 0)
    setHeightLinearLayout(binding.lySoftVolume, value: 0)
    binding.lyFragmentMap.translationY = -300f
    binding.lyIntervalMode.translationY = -300f
    binding.lyChallenges.translationY = -300f
    binding.lySettingsVolumes.translationY = -300f
}
```

Esta función llamada "**hideLayouts**" oculta varios layouts específicos. Esto se hace estableciendo la altura de cada layout en 0 utilizando la función "**setHeightLinearLayout**" y moviendo cada layout hacia arriba fuera de la pantalla utilizando la propiedad "**translationY**" con un valor negativo. **Esto hará que los layouts se oculten de la vista del usuario.**

9.4 Función inflateVolumes



Dentro de nuestro layout principal tenemos algunos switches que manejan activar/desactivar opciones del menú principal.



Al pulsar en el switch se despliegan distintas opciones, como en este caso el volumen de la música y además se cambia el color de las letras cada vez que se habilita el switch .

```
new *
fun inflateVolumes(v: View) {
    if (binding.swVolumes.isChecked) {
        animateViewofInt(binding.swVolumes, attr: "textColor", ContextCompat.getColor(context: this, R.color.orange), time: 500)
        var value = 400
        if (binding.swIntervalMode.isChecked) value = 600
        setHeightLinearLayout(binding.lySettingsVolumesSpace, value)
        animateViewofFloat(binding.lySettingsVolumes, attr: "translationY", value: 0f, time: 500)
    } else {
        binding.swVolumes.setTextColor(ContextCompat.getColor(context: this, R.color.gray_dark))
        setHeightLinearLayout(binding.lySettingsVolumesSpace, value: 0)
        binding.lySettingsVolumes.translationY = -300f
    }
}
```

con la función `inflateVolumes` manejamos la visualización de una sección específica de una interfaz de usuario. La función toma un parámetro "v" de tipo View.

Si el switch "`swVolumes`" está marcado (`isChecked` es verdadero), la función cambia el color del texto del switch a naranja utilizando la animación `animateViewofInt` y aumenta la altura de un LinearLayout llamado "`lySettingsVolumesSpace`" utilizando la función `setHeightLinearLayout`. También anima el "`lySettingsVolumes`" moviéndolo hacia arriba utilizando `animateViewofFloat`. Si el switch no está marcado, la función cambia el color del texto del switch a gris oscuro, establece la altura del LinearLayout en 0 y mueve el "`lySettingsVolumes`" hacia abajo utilizando la propiedad "`translationY`".

9.4.1 Función `animateViewofFloat` y `animateViewofInt`

```
fun animateViewofFloat(v: View, attr: String, value: Float, time: Long) {
    ObjectAnimator.ofFloat(v, attr, value).apply { this: ObjectAnimator!
        duration = time
        start()
    }
}
```

Estas funciones animan una vista (v) cambiando un atributo de punto flotante (attr) a un valor específico (value) en un cierto período de tiempo (time). La función utiliza la clase **ObjectAnimator** de Android para animar la vista y establece la duración de la animación al tiempo especificado en los argumentos de la función. Luego se llama al método start() para iniciar la animación.

En inflatevolumes llamamos a esta función en la línea de código:

```
animateViewofFloat(binding.lySettingsVolumes, "translationY", 0f, 500)
```

Esta línea de código utiliza la función **animateViewofFloat** para animar la vista llamada **lySettingsVolumes**. El atributo **translationY** de la vista se establece en 0.0 y la animación tardará 500 milisegundos en completarse. **Esta animación hará que la vista se deslice hacia arriba, mostrándola en la pantalla desde su posición actual hasta la posición de y = 0.**

```
fun animateViewofInt(v: View, attr: String, value: Int, time: Long) {
    ObjectAnimator.ofInt(v, attr, value).apply { this: ObjectAnimator!
        duration = time
        start()
    }
}
```

En inflatevolumes llamamos también a esta función en la línea de código:

```
animateViewofInt(binding.swVolumes,"textColor", ContextCompat.getColor(this,
R.color.orange),500)
```

Esta línea de código utiliza la función **animateViewofInt** para animar el atributo **textColor** de una vista llamada **swVolumes**. La función cambiará el color del texto de la vista a naranja mediante el uso de un recurso de color ubicado en **R.color.orange** y **ContextCompat.getColor(this, R.color.orange)** para obtener el valor del color. La animación tardará 500 milisegundos en completarse.

9.5 Función inflateChallenges

```
fun inflateChallenges(v: View) {
    if (binding.swChallenges.isChecked) {
        animateViewofInt(binding.swChallenges, attr: "textColor", ContextCompat.getColor(context: this, R.color.orange), time: 500)
        setHeightLinearLayout(binding.lyChallengesSpace, value: 750)
        animateViewofFloat(binding.lyChallenges, attr: "translationY", value: 0f, time: 500)
    } else {
        binding.swChallenges.setTextColor(ContextCompat.getColor(context: this, R.color.white))
        setHeightLinearLayout(binding.lyChallengesSpace, value: 0)
        binding.lyChallenges.translationY = -300f
        challengeDistance = 0f
        challengeDuration = 0
    }
}
```

La función **inflateChallenges** verifica el estado de una vista y realiza acciones específicas dependiendo si está marcada o no. Si está marcada, anima el color del texto y la posición de otra vista y establece una altura específica. Si no está marcada, establece el color del texto y la posición de una vista y establece una altura específica. Esta función es utilizada en nuestra pantalla principal para mostrar y ocultar las configuraciones relacionadas con los desafíos.

9.6 TranslationZ

En Android, la propiedad "**translationZ**" de un layout XML determina la distancia del layout con respecto al usuario, a lo largo del eje Z. Se utiliza para dar al layout un sentido de profundidad y puede utilizarse para crear un efecto 3D.

Es posible cambiar la propiedad **translationZ** de un layout dinámicamente, utilizando la clase ObjectAnimator, con el fin de realizar animaciones que cambian la distancia de un layout con respecto al usuario.

También es importante tener en cuenta que también puede utilizarse para administrar el orden de los elementos en la pantalla cuando se superponen, el que tiene el translationZ más alto se mostrará en la parte superior de los demás.

9.6.1 Funciones showChallenge, showDuration, showDistance, getChallengeDuration

```

new *
fun showDuration(v: View) {
    if (timeInSeconds == 0L) showChallenge(option: "duration")
}

new *
fun showDistance(v: View) {
    if (timeInSeconds == 0L) showChallenge(option: "distance")
}

```

```

new *
private fun showChallenge(option: String) {
    when (option) {
        "duration" -> {
            binding.lyChallengeDuration.translationZ = 5f
            binding.lyChallengeDistance.translationZ = 0f
            binding.tvChallengeDuration.setTextColor(ContextCompat.getColor(context: this, R.color.orange))
            binding.tvChallengeDuration.setBackgroundColor(ContextCompat.getColor(context: this, R.color.gray_dark))
            binding.tvChallengeDistance.setTextColor(ContextCompat.getColor(context: this, R.color.white))
            binding.tvChallengeDistance.setBackgroundColor(ContextCompat.getColor(context: this, R.color.gray_medium))
            challengeDuration = 0f
            getChallengeDuration(binding.npChallengeDurationHH.value, binding.npChallengeDurationMM.value, binding.npChallengeDurationSS.value)
        }
        "distance" -> {
            binding.lyChallengeDuration.translationZ = 0f
            binding.lyChallengeDistance.translationZ = 5f
            binding.tvChallengeDuration.setTextColor(ContextCompat.getColor(context: this, R.color.white))
            binding.tvChallengeDuration.setBackgroundColor(ContextCompat.getColor(context: this, R.color.gray_medium))
            binding.tvChallengeDistance.setTextColor(ContextCompat.getColor(context: this, R.color.orange))
            binding.tvChallengeDistance.setBackgroundColor(ContextCompat.getColor(context: this, R.color.gray_dark))
            challengeDuration = 0
            challengeDistance = binding.npChallengeDistance.value.toFloat()
        }
    }
}

```

función showChallenge

```

new *
private fun getChallengeDuration(hh: Int, mm: Int, ss: Int) {
    var hours: String = hh.toString()
    if (hh < 10) hours = "0" + hours
    var minutes: String = mm.toString()
    if (mm < 10) minutes = "0" + minutes
    var seconds: String = ss.toString()
    if (ss < 10) seconds = "0" + seconds
    challengeDuration = getSecFromWatch(watch: "${hours}:${minutes}:${seconds}")
}

```

función getChallengeDuration

El código anterior permite al usuario elegir entre dos desafíos: duración o distancia. Cuando el usuario elige "duración", se activa la opción de duración y se desactiva la opción de distancia, y viceversa. También se establece un valor de color y fondo para cada opción, y se llama a una función "**getChallengeDuration**" que convierte la duración elegida por el usuario en segundos.

```
fun getSecFromWatch(watch: String): Int {
    var secs = 0
    var w: String = watch
    if (w.length == 5) w = "00:" + w
    // 00:00:00
    secs += w.subSequence(0, 2).toString().toInt() * 3600
    secs += w.subSequence(3, 5).toString().toInt() * 60
    secs += w.subSequence(6, 8).toString().toInt()
    return secs
}
```

función getSecFromWatch

La función "**getSecFromWatch**" tiene como objetivo convertir una hora en formato de texto (en el formato "HH:MM:SS") a segundos. El argumento "watch" es una cadena de texto que representa la hora. La función divide esta cadena en tres partes: horas, minutos y segundos. Luego, multiplica las horas por 3600, los minutos por 60 y suma los segundos para obtener un total en segundos. Finalmente, la función devuelve el total de segundos.

```
fun getSecFromWatch(watch: String): Int {
    val time = watch.split(":")
    return (time[0].toInt() * 3600) + (time[1].toInt() * 60) + time[2].toInt()
}
```

otra versión de función getSecFromWatch

Este código es una función en Kotlin que recibe un parámetro de tipo String llamado "watch", que representa una hora en formato "HH:MM:SS". La función tiene un único propósito: convertir esa hora en segundos.

La primera línea utiliza el método "split" en el objeto "watch" para dividir el string en un arreglo de tres elementos, cada uno representando las horas, minutos y segundos respectivamente.

La segunda línea es el return de la función. Aquí se realiza la conversión de horas a segundos multiplicando el primer elemento del arreglo (horas) por 3600, se suma el resultado de multiplicar el segundo elemento (minutos) por 60 y se suma el tercer elemento (segundos) para obtener el total de segundos.

9.7 Función inflateIntervalMode

```
fun inflateIntervalMode(v: View) {
    if (binding.swIntervalMode.isChecked) {
        animateViewofInt(
            binding.swIntervalMode, attr: "textColor", ContextCompat.getColor(context: this, R.color.orange),
            value: 0f, time: 500
        )
        setHeightLinearLayout(binding.lyIntervalModeSpace, value: 600)
        animateViewofFloat(binding.lyIntervalMode, attr: "translationY", value: 0f, time: 500)
        animateViewofFloat(binding.tvChrono, attr: "translationX", value: -110f, time: 500)
        binding.tvRounds.setText(R.string.rounds)
        animateViewofInt(
            binding.tvRounds,
            attr: "textColor",
            ContextCompat.getColor(context: this, R.color.white),
            value: 0f, time: 500
        )
        setHeightLinearLayout(binding.lySoftTrack, value: 120)
        setHeightLinearLayout(binding.lySoftVolume, value: 200)
        if (binding.swVolumes.isChecked) {
            setHeightLinearLayout(binding.lySettingsVolumesSpace, value: 600)
        }
        TIME_RUNNING = getSecFromWatch(binding.tvRunningTime.text.toString())
    } else {
        binding.swIntervalMode.setTextColor(ContextCompat.getColor(context: this, R.color.gray_dark))
        setHeightLinearLayout(binding.lyIntervalModeSpace, value: 0)
        binding.lyIntervalMode.translationY = -200f
        animateViewofFloat(binding.tvChrono, attr: "translationX", value: 0f, time: 500)
        binding.tvRounds.text = ""
    }
}
```

función inflateIntervalMode

Con ésta función se combinan las animaciones que hemos programado antes, además se encarga de inflar una vista (View) llamada "**IntervalMode**". La función se activa cuando se cambia el estado de un interruptor (**binding.swIntervalMode.isChecked**). Si el interruptor está activado, se realizan animaciones y cambios en la vista, cómo cambiar el color del texto, aumentar la altura de algunos elementos, mover elementos en la pantalla y cambiar el contenido de algunos textos. Si el interruptor está desactivado, se realizan acciones opuestas a las mencionadas anteriormente, como cambiar el color del texto de nuevo, reducir la altura de elementos, mover elementos de nuevo a su posición original y vaciar algunos textos. También se controla el estado de otro interruptor en la vista (**binding.swVolumes.isChecked**) para ajustar la altura de algunos elementos adicionales en función de su estado como el cronómetro.

```
private fun initStopWatch() {
    binding.tvChrono.text = getString(R.string.init_stop_watch_value)
}
```

Además con esta pequeña función **initStopWatch** solo ponemos la cadena del cronómetro a 00:00:00 .

9.8 Control del Carrera con Intervalos.

9.8.1 Función initIntervalMode

La función "**initIntervalMode**" se encarga de inicializar y configurar varios elementos de la interfaz de usuario relacionados con el modo **Carrera con Intervalos**. Estos elementos incluyen un **NumberPicker** para seleccionar el tiempo de duración del intervalo, una barra de progreso circular para indicar el tiempo de ejecución y caminando, y varios **TextViews** para mostrar el tiempo de ejecución y caminando. Además, cuando el usuario cambia el valor del **NumberPicker** o interactúa con la barra de progreso circular, el código actualiza los elementos de la interfaz de usuario y variables de tiempo necesarias para el modo de intervalo.

La línea de código "npDurationInterval.wrapSelectorWheel = true" indica que el selector de rueda (wheel selector) del NumberPicker debe envolverse. Esto significa que si el usuario continúa desplazando el selector más allá del valor máximo o mínimo, el selector volverá al otro extremo del rango.

```
npDurationInterval.setFormatter { i -> String.format("%02d", i) }
```

El `setFormatter` es una función lambda que recibe un entero "i" y lo formatea como una cadena con dos dígitos con un cero a la izquierda si es necesario. La función "String.format" es utilizada para dar formato a la salida. Esto significa que si se proporciona un número menor de 10, se añadirá un cero a la izquierda.

```
new *
private fun initIntervalMode() {
    npDurationInterval.minValue = 1
    npDurationInterval maxValue = 60
    npDurationInterval.value = 5
    npDurationInterval.wrapSelectorWheel = true
    npDurationInterval.setFormatter { i -> String.format("%02d", i) }
    npDurationInterval.setOnValueChangedListener { _, _, newVal ->
        csbRunWalk.max = (newVal * 60).toFloat()
        csbRunWalk.progress = csbRunWalk.max / 2
        tvRunningTime.text = getFormattedStopWatch((newVal * 60 / 2) * 1000).toLong().subSequence(3, 8)
        tvWalkingTime.text = tvRunningTime.text
        ROUND_INTERVAL = newVal * 60
        TIME_RUNNING = ROUND_INTERVAL / 2
    }
}
```

Este fragmento de código establece un "listener" o "escuchador" en el objeto "**npDurationInterval**", el cual es un "**NumberPicker**" o selector numérico. Cuando el valor seleccionado en este objeto cambia, se ejecutan varias acciones.

Primero, se establece el valor máximo del objeto "**csbRunWalk**" (un "**CircularSeekBar**" o barra de progreso circular) en `(newVal * 60)` (donde `newVal` es el nuevo valor seleccionado en el "**npDurationInterval**"), y el valor actual del progreso se establece en la mitad del valor máximo.

Luego, se establecen los valores de texto de los objetos "**tvRunningTime**" y "**tvWalkingTime**" utilizando la función "**getFormattedStopWatch**" y un valor calculado.

Finalmente, se establecen los valores de las variables "**ROUND_INTERVAL**" y "**TIME_RUNNING**" en base al nuevo valor seleccionado en el "**npDurationInterval**". En resumen, este fragmento de código establece una relación entre el valor seleccionado en el "**npDurationInterval**" y varios otros objetos y variables en el código, actualizando sus valores en consecuencia.

```

csbRunWalk.max = 300f
csbRunWalk.progress = 150f
csbRunWalk.setOnSeekBarChangeListener(object :
    CircularSeekBar.OnCircularSeekBarChangeListener {
    override fun onProgressChanged(
        circularSeekBar: CircularSeekBar?,
        progress: Float,
        fromUser: Boolean
    ) {
        if (fromUser) {
            var steps = 15
            if (ROUND_INTERVAL > 600) steps = 60
            if (ROUND_INTERVAL > 1800) steps = 300
            var set = 0
            var p = progress.toInt()
            var limit = 60
            if (ROUND_INTERVAL > 1800) limit = 300
            if (p % steps != 0 && progress != csbRunWalk.max) {
                while (p >= limit) p -= limit
                while (p >= steps) p -= steps
                //if (steps - p > steps / 2) set = -1 * p
                set = if (steps - p > steps / 2) -1 * p
                else steps - p
                if (csbRunWalk.progress + set > csbRunWalk.max)
                    csbRunWalk.progress = csbRunWalk.max
                else
                    csbRunWalk.progress = csbRunWalk.progress + set
            }
        }
        //Si el progress está a cero se pone a false y si no se habilita correr
        if (csbRunWalk.progress == 0f) manageEnableButtonsRun( e_reset:false, e_run:false)
        else manageEnableButtonsRun( e_reset:false, e_run:true)

        tvRunningTime.text =
            getFormattedStopWatch((csbRunWalk.progress.toInt() * 1000).toLong()).subSequence(
                3,
                8
            )
        tvWalkingTime.text =
            getFormattedStopWatch(((ROUND_INTERVAL - csbRunWalk.progress.toInt()) * 1000).toLong()).subSequence(
                3,
                8
            )
        TIME_RUNNING = getSecFromWatch(tvRunningTime.text.toString())
    }

    override fun onStopTrackingTouch(seekBar: CircularSeekBar?) {
    }

    override fun onStartTrackingTouch(seekBar: CircularSeekBar?) {
    }
})
}

```

Este fragmento de código establece el valor máximo y el progreso inicial del **CircularSeekBar** llamado "**csbRunWalk**". Luego, establece un listener para la barra de búsqueda para que cuando el usuario haga clic y cambie el progreso, se ejecute una acción.

La acción en sí tiene varias partes:

1. Primero, se establecen algunas variables, como "steps", "set", y "limit", que se utilizan en el cálculo del progreso.
2. Luego, se comprueba si el progreso no es divisible por "steps" y si el progreso no es igual al valor máximo de la barra de búsqueda. Si se cumple alguna de estas condiciones, se realizan algunos cálculos para asegurar que el progreso sea divisible por "steps".
3. El progreso se ajusta según los cálculos anteriores y se comprueba si el progreso es igual a cero, en cuyo caso se deshabilitan algunos botones.

4. Se actualizan los valores de dos TextViews llamados "tvRunningTime" y "tvWalkingTime" mostrando el tiempo transcurrido en formato hh:mm:ss.
5. Se actualiza el valor de "TIME_RUNNING" con el tiempo transcurrido.

En resumen, este fragmento de código establece y controla el progreso de la barra de búsqueda y actualiza los valores de tiempo y habilita/deshabilita los botones en función del progreso.

9.8.2 Función getFormattedStopWatch

```
fun getFormattedStopWatch(ms: Long): String {
    val hours = TimeUnit.MILLISECONDS.toHours(ms) % 24
    val minutes = TimeUnit.MILLISECONDS.toMinutes(ms) % 60
    val seconds = TimeUnit.MILLISECONDS.toSeconds(ms) % 60
    return String.format("%02d:%02d:%02d", hours, minutes, seconds)
}
```

La función "getFormattedStopWatch" convierte un valor en milisegundos (ms) en una cadena de tiempo con formato horas:minutos:segundos. Se utiliza el operador módulo para obtener el valor restante después de convertir los milisegundos a horas, minutos y segundos. También se utiliza el método de formato de cadena para asegurar que los valores sean siempre de dos dígitos.

9.9 Control del Carrera con Intervalos.

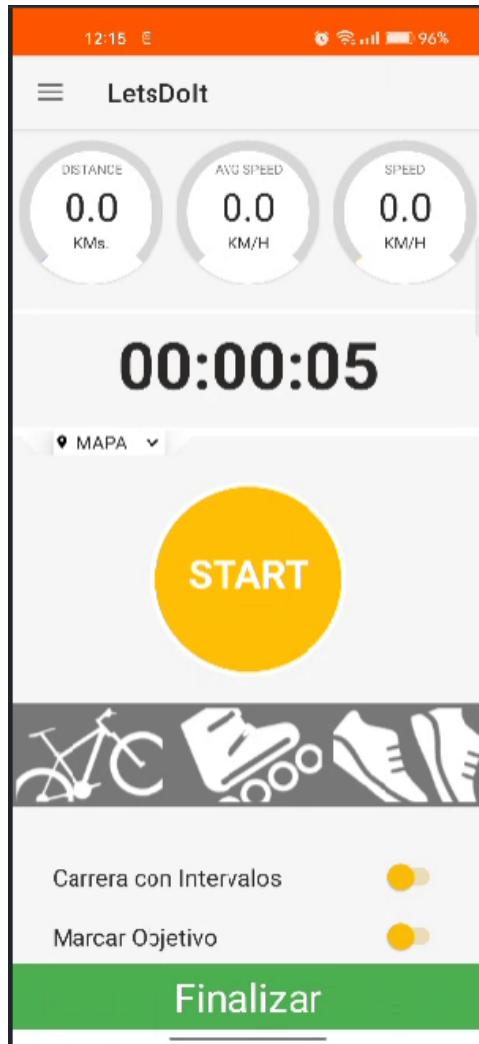
9.9.1 Marcar Objetivo . Función initChallengeMode

Al igual que la función anterior initIntervalMode , en esta función el código inicializa la funcionalidad de Marcar Objetivo. Se establecen valores mínimos y máximos para los **NumberPicker**, se establece el valor inicial y se habilita el "wrapSelectorWheel". También se establecen formateadores para mostrar los valores con dos dígitos. Los **NumberPicker** se utilizan para seleccionar la distancia y la duración del desafío. Se establecen listeners para cada **NumberPicker** de la duración para actualizar la variable "challengeDuration" con la duración seleccionada.

Para optimizar este código, se puede crear una función para inicializar cada NumberPicker, en lugar de escribir las mismas líneas de código varias veces para cada NumberPicker. También se podría utilizar una clase para encapsular la lógica del modo desafío y así separarlo del código principal de la aplicación.

En cuanto a la funcionalidad, se podría añadir una validación para asegurar que la distancia seleccionada sea menor o igual a la distancia del récord, de lo contrario, se le podría mostrar un mensaje de error al usuario.

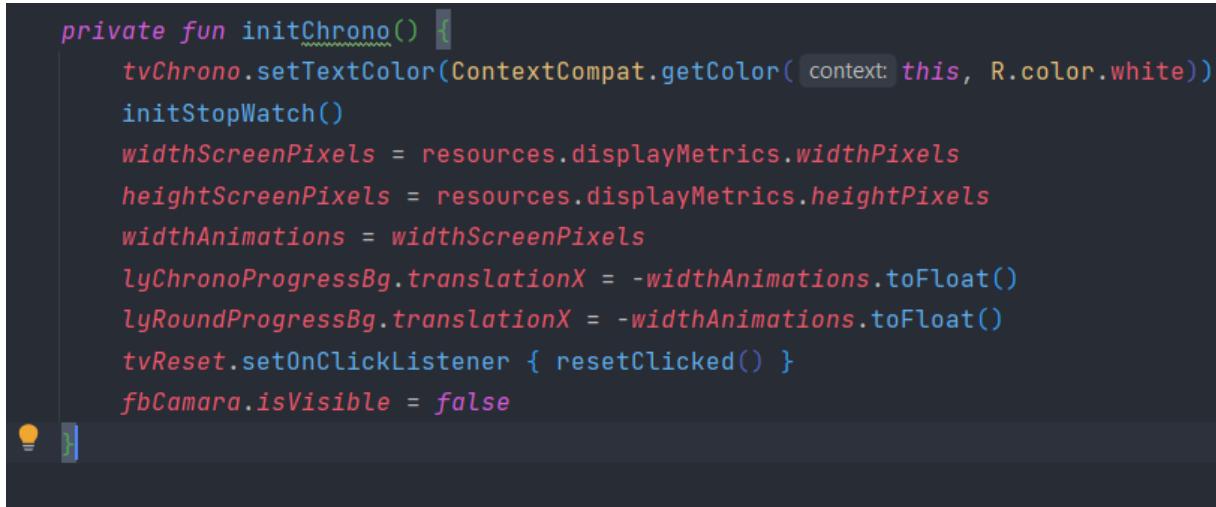
10. Implementación del Cronómetro y sus opciones



El cronómetro es clave en el proyecto y controla el tiempo, música, intervalos, ubicación GPS y cálculo de distancias, velocidades y marcas. Además, muestra y oculta opciones en el menú según su uso. Todo se basa en el cronómetro y se ejecutan funciones según los permisos y configuraciones del usuario, es necesario verificar permisos para GPS y cámara.

Puntos importantes:

- El cronómetro es el componente clave del proyecto.
- Funciones del cronómetro: iniciar, pausar, finalizar y ocultar opciones.
- Calcula y lanza música, intervalos, ubicación del GPS y notificaciones.
- Comprobación de permisos para usar la cámara y localización GPS.



La función "initChrono" inicializa el cronómetro y hace lo siguiente:

1. Establece el color de texto del elemento "tvChrono" en blanco utilizando el contexto de la aplicación.
2. Llama a la función "initStopWatch" para inicializar un reloj de parada.
3. Obtiene las medidas de ancho y alto de la pantalla en pixels y las almacena en las variables "widthScreenPixels" y "heightScreenPixels".
4. Asigna el ancho de la pantalla a la variable "widthAnimations".
5. Translada los elementos "lyChronoProgressBg" y "lyRoundProgressBg" hacia la izquierda fuera de la pantalla.
6. Establece un listener en el elemento "tvReset" para que al ser presionado llame a la función "resetClicked"
7. Oculta el elemento "fbCamara".

10.1 Objetos Handler y Runnable

El cronómetro en la app utiliza una variable de tipo **Handler**, y además implementará la interfaz **Runnable**. La lógica de actualizar el contador de tiempo se implementa en un objeto de tipo **Runnable**. La clase se envía al objeto **Handler** mediante **post** o **postDelayed** y se agenda para su ejecución en el momento adecuado. Además, la app realiza cálculos como ubicación y cálculo de distancias y velocidades.

```

private var chronometer: Runnable = object : Runnable {
    override fun run() {
        try {
            timeInSeconds += 1
            updateStopWatchView()

        } finally {
            mHandler!!.postDelayed(this, mInterval.toLong())
        }
    }
}

private fun updateStopWatchView() {
}

```

```
tvChrono.text = getFormattedStopWatch(timeInSeconds * 1000)
}
```

Este código crea un objeto Runnable llamado "**chronometer**". La clase anónima que implementa Runnable tiene un método llamado "run()". Dentro de este método, se actualiza el valor de "**timeInSeconds**" en 1 y luego se llama a "**updateStopWatchView()**". Finalmente, se utiliza el método "postDelayed()" de un objeto Handler previamente definido para programar una llamada al método "run()" cada "**mInterval**" milisegundos.

La función "**updateStopWatchView()**" actualiza la vista del cronómetro en la pantalla, estableciendo el texto de "**tvChrono**" con el resultado de "**getFormattedStopWatch()**", que recibe el tiempo en milisegundos actual y lo formatea para mostrarlo en la pantalla.

10.2 Función manageRun. Lanzar y Detener el Cronómetro

La función **manageRun** es utilizada para iniciar o detener el cronómetro de la aplicación. Al hacer clic en el botón que llama a esta función, se realiza una evaluación para verificar si el cronómetro se ha iniciado previamente.

Si el cronómetro no se ha iniciado previamente (**startButtonClicked** es falso), entonces se establece **startButtonClicked** como verdadero y se invoca la función **startTime** para iniciar el cronómetro.

Si el cronómetro ya está en ejecución (**startButtonClicked** es verdadero), entonces se establece **startButtonClicked** como falso y se invoca la función **stopTime** para detener el cronómetro.

La función **startTime** crea un nuevo objeto Handler y llama a **chronometer.run()** para iniciar la ejecución del cronómetro.

La función **stopTime** detiene el cronómetro invocando **mHandler?.removeCallbacks(chronometer)**, que remueve la llamada a **chronometer** de la cola de mensajes del Handler.

```
private var chronometer: Runnable = object : Runnable {...}

new *
private fun updateStopWatchView() {
    tvChrono.text = getFormattedStopWatch(ms: timeInSeconds * 1000)
}

new *
fun startOrStopButtonClicked(v: View) {
    manageRun()
}

new *
private fun startTime() {
    mHandler = Handler(Looper.getMainLooper())
    chronometer.run()
}

new *
private fun stopTime() {
    mHandler?.removeCallbacks(chronometer)
}

new *
private fun manageRun() {
    if (!startButtonClicked) {
        startButtonClicked = true
        startTime()
    } else {
        startButtonClicked = false
    }
    stopTime()
}
```

El método "run" lleva a cabo la función de aumentar el tiempo en 1 y llamar a la función "**updateStopWatchView**" para actualizar la vista de la pantalla y establecer el texto "**tvChrono**" con el tiempo formateado. La función "**startOrStopButtonClicked**" controla si el cronómetro se inicia o detiene. La función "**startTime**" crea un objeto **Handler** y ejecuta "**chronometer**", mientras que "**stopTime**" detiene la ejecución de "**chronometer**" usando "**removeCallbacks**". La función "**manageRun**" controla el estado del cronómetro, iniciándolo o deteniéndolo, dependiendo del estado actual.

10.3 Botón del Cronómetro. Activar/Desactivar Funciones

En ciertos momentos hay que limitar el uso de los botones mediante código. La función `manageEnableButtonsRun()` va administrar lo que está activado o desactivado.

```
new
private fun manageEnableButtonsRun(e_reset: Boolean, e_run: Boolean) {
    tvReset.isEnabled = e_reset
    btStart.isEnabled = e_run

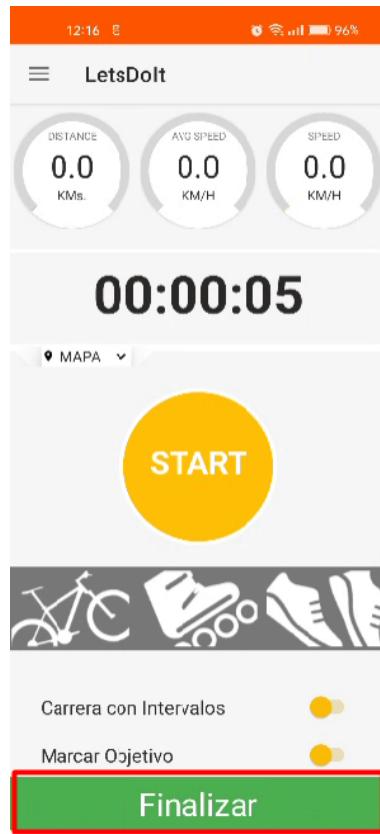
    if (e_reset) { // Si el reset está habilitado se pone en color verde y se eleva
        tvReset.setBackgroundColor(ContextCompat.getColor(context: this, R.color.green))
        animateViewoffFloat(tvReset, attr: "translationY", value: 0f, time: 500)
    } else { // En caso contrario se pone en gris y se oculta
        tvReset.setBackgroundColor(ContextCompat.getColor(context: this, R.color.gray))
        animateViewoffFloat(tvReset, attr: "translationY", value: 150f, time: 500)
    }

    if (e_run) { // El botón de la carrera está activo cambiamos el color del círculo
        if (startButtonClicked) {
            btStart.background = AppCompatResources.getDrawable(context: this, R.drawable.circle_background_topause)
            btStartLabel.setText(R.string.stop)
        } else {
            btStart.background = AppCompatResources.getDrawable(context: this, R.drawable.circle_background_toplay)
            btStartLabel.setText(R.string.start)
        }
    } else btStart.background = AppCompatResources.getDrawable(context: this, R.drawable.circle_background_todisable)
}
```

La función `manageEnableButtonsRun` cambia la apariencia y habilitación de los botones "reset" y "run". Si el parámetro `e_reset` es true, entonces el botón "reset" se vuelve habilitado y su apariencia cambia a verde y se eleva (mediante la función `animateViewoffFloat`). Si `e_reset` es false, el botón "reset" se deshabilita y su apariencia cambia a gris y se oculta.

Si el parámetro `e_run` es true, entonces el botón "run" se vuelve habilitado y su apariencia cambia dependiendo si el cronómetro está corriendo o detenido. Si el cronómetro está corriendo, el botón "run" cambia a una imagen de pausa. Si el cronómetro está detenido, el botón "run" cambia a una imagen de play. Si `e_run` es false, el botón "run" se deshabilita y su apariencia cambia a un fondo gris.

10.4 Botón de Reset del Cronómetro



El botón finalizar / reset en realidad es un TextView, en esta ocasión no se utiliza el método onClick, en su lugar se utiliza setOnClickListener

```
<TextView
    android:id="@+id/tvReset"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:layout_alignParentEnd="true"
    android:layout_alignParentRight="true"
    android:layout_alignParentBottom="true"
    android:layout_gravity="center|bottom"
    android:background="@color/gray_light"
    android:enabled="true"
    android:paddingTop="5dp"
    android:text="@string/finish"
    android:textAlignment="center"
    android:textColor="@color/white"
    android:textSize="30sp"
    android:translationY="150dp" />
```

```
private fun initChrono() {
    tvChrono.setTextColor(ContextCompat.getColor(context, R.color.gray_dark))
    initStopWatch()
    widthScreenPixels = resources.displayMetrics.widthPixels
    heightScreenPixels = resources.displayMetrics.heightPixels
    widthAnimations = widthScreenPixels
    lyChronoProgressBg.translationX = -widthAnimations.toFloat()
    lyRoundProgressBg.translationX = -widthAnimations.toFloat()
    tvReset.setOnClickListener { resetClicked() }
    fbCamara.isVisible = false
}
```

setOnClickListener es un método de la clase View que se utiliza para establecer un listener o escuchador para un evento de clic en una vista o elemento visual en una aplicación Android. Cada vez que el usuario toca o hace clic en la vista, el método asignado en el listener se ejecutará. Este método recibe como parámetro una clase anónima que implementa la interfaz **OnClickListener**. En este caso, se está utilizando una expresión **lambda** como argumento para establecer la acción a realizar cuando se hace clic en el elemento **tvReset**.

10.5 Habilitar/Deshabilitar Opciones de Carrera

```
private fun resetInterface() {  
    fbCamara.isVisible = false  
    tvCurrentDistance.text = "0.0"  
    tvCurrentAvgSpeed.text = "0.0"  
    tvCurrentSpeed.text = "0.0"  
    tvDistanceRecord.setTextColor(ContextCompat.getColor(context: this, R.color.gray_dark))  
    tvAvgSpeedRecord.setTextColor(ContextCompat.getColor(context: this, R.color.gray_dark))  
    tvMaxSpeedRecord.setTextColor(ContextCompat.getColor(context: this, R.color.gray_dark))  
    csbCurrentDistance.progress = 0f  
    csbCurrentAvgSpeed.progress = 0f  
    csbCurrentSpeed.progress = 0f  
    csbCurrentMaxSpeed.progress = 0f  
    tvRounds.text = "Round 1"  
    lyChronoProgressBg.translationX = -widthAnimations.toFloat()  
    lyRoundProgressBg.translationX = -widthAnimations.toFloat()  
    swIntervalMode.isClickable = true  
    npDurationInterval.isEnabled = true  
    csbRunWalk.isEnabled = true  
    swChallenges.isClickable = true  
    npChallengeDistance.isEnabled = true  
    npChallengeDurationHH.isEnabled = true  
    npChallengeDurationMM.isEnabled = true  
    npChallengeDurationSS.isEnabled = true  
    sbHardTrack.isEnabled = false  
    sbSoftTrack.isEnabled = false  
}
```

La función "**resetInterface**" reinicia los valores y estados de los elementos de la interfaz de usuario para que la aplicación vuelva a su estado inicial. Establece textos en 0.0, deshabilita elementos de la interfaz como **npDurationInterval**, **npChallengeDistance**, etc., y establece las propiedades de los elementos como el color de texto, la posición de los elementos, la capacidad de hacer clic en elementos, etc.

10.6 Control de Intervalos

El método "**checkStopRun**" verifica si es necesario detener la carrera en base a la cantidad de tiempo transcurrido (en segundos). Si la duración en segundos es mayor que el intervalo de tiempo establecido, se resta el intervalo de tiempo hasta que la duración restante sea igual al tiempo de ejecución. Si la duración restante es igual al tiempo de ejecución, se detiene la carrera y cambian algunos elementos de la interfaz de usuario (como el color del texto y el color del progreso de la ronda), así como también se cambia la música. Si la duración restante no es igual al tiempo de ejecución, se actualiza el progreso de la ronda.

La función **checkNewRound** verifica si se ha alcanzado una nueva ronda (definida por el valor de "roundInterval"). Si el tiempo transcurrido (Secs) es un múltiplo del intervalo de la ronda y es mayor que cero, entonces se aumenta el número de rondas, se actualiza el texto de la pantalla para reflejar la nueva ronda y se cambian los colores del cronómetro y el progreso de la ronda a rojo (que indica que es el momento de correr). También se reinicia el progreso de la ronda y se cambia la música a una más rápida. Si no se ha alcanzado una nueva ronda, entonces solo se actualiza el progreso de la ronda en la pantalla.

La función **updateProgressBarRound** actualiza la barra de progreso de la ronda. El tiempo en segundos se pasa como un parámetro "secs". Luego, se utiliza un bucle "while" para calcular la cantidad de tiempo restante después de haber restado el intervalo de la ronda. Después, se verifica el color actual del texto del cronómetro y, dependiendo de si es de correr o de caminar, se realiza un cálculo para mover la barra de progreso. Finalmente, se anima la barra de progreso con la función "**animateViewofFloat**".

11. Implementación de la Música

11.1 MediaPlayer

MediaPlayer es una clase en Android que permite reproducir diferentes tipos de medios, como audio y video. Es una herramienta fácil de usar y personalizar para agregar música y sonidos a una aplicación. Permite reproducir archivos locales y en línea, controlar la reproducción (pausa, reanudación, detener), establecer el volumen y mucho más.

11.2 Función initMusic. Inicialización de objetos MediaPlayer

```

new *

private fun initMusic() {
    mpNotify = MediaPlayer.create(context: this, R.raw.ding)
    mpHard = MediaPlayer.create(context: this, R.raw.running)
    mpSoft = MediaPlayer.create(context: this, R.raw.walking)
    mpHard?.isLooping = true
    mpSoft?.isLooping = true
    setVolumes()
    setProgressTracks()
}

```

La función "initMusic" se encarga de inicializar los objetos **MediaPlayer** para las tres pistas de audio. Se crea un objeto **MediaPlayer** para cada una de las tres pistas de audio (ding, running y walking) y se asocian con los recursos correspondientes. Se establece que las dos pistas de audio (running y walking) deben repetirse en bucle. Luego, se llaman las funciones "**setVolumes**" y "**setProgressTracks**" para establecer los volúmenes y el progreso de las pistas de audio, respectivamente.

11.3 Función setVolumes. Control del volumen

```

private fun setVolumes() {
    sbHardVolume.setOnSeekBarChangeListener(object : SeekBar.OnSeekBarChangeListener {
        override fun onProgressChanged(p0: SeekBar?, i: Int, p2: Boolean) {
            mpHard?.setVolume(leftVolume: i / 100.0f, rightVolume: i / 100.0f)
        }

        override fun onStartTrackingTouch(p0: SeekBar?) {}
        override fun onStopTrackingTouch(p0: SeekBar?) {}
    })

    sbSoftVolume.setOnSeekBarChangeListener(object : SeekBar.OnSeekBarChangeListener {...})
    sbNotifyVolume.setOnSeekBarChangeListener(object : SeekBar.OnSeekBarChangeListener {...})
}

```

La función **setVolumes** asigna controladores de eventos de **SeekBar** (**OnSeekBarChangeListener**) a tres barras deslizantes (**SeekBar**) **sbHardVolume**, **sbSoftVolume**, y **sbNotifyVolume**.

Los controladores de eventos **OnSeekBarChangeListener** actualizan el volumen de tres objetos **MediaPlayer** **mpHard**, **mpSoft** y **mpNotify** cuando los usuarios deslizan las barras deslizantes. Los métodos **onStartTrackingTouch** y **onStopTrackingTouch** no tienen ninguna acción en este código.

11.4 Función setProgressTracks. Control de pistas de audio

```

private fun setProgressTracks() {
    sbHardTrack.max = mpHard!!.duration
    sbSoftTrack.max = mpSoft!!.duration
    updateTimesTrack(timesH: true, timesS: true)
    sbHardTrack.setOnSeekBarChangeListener(object : SeekBar.OnSeekBarChangeListener {
        override fun onProgressChanged(p0: SeekBar?, i: Int, fromUser: Boolean) {
            if (fromUser) {
                mpHard?.pause() // Pausamos la canción
                mpHard?.seekTo(i) // Nos desplazamos al punto que ha indicado el usuario
                mpHard?.start() // Reanudamos la canción de nuevo
                updateTimesTrack(timesH: true, timesS: false) // Actualizamos los tracks
            }
        }

        override fun onStartTrackingTouch(p0: SeekBar?) {}
        override fun onStopTrackingTouch(p0: SeekBar?) {}
    })

    sbSoftTrack.setOnSeekBarChangeListener(object : SeekBar.OnSeekBarChangeListener {...})
}

```

Con la función **setProgressTracks** se configuran las barras de progreso de las dos pistas de música (**mpHard** y **mpSoft**) y una notificación (**mpNotify**). Cada barra de volumen (**sbHardVolume**, **sbSoftVolume**, **sbNotifyVolume**) tiene un listener que cambia el volumen de su correspondiente pista/notificación según la posición en la barra. Las barras de progreso (**sbHardTrack**, **sbSoftTrack**) también tienen un listener que permite al usuario mover el progreso de la canción y actualiza el tiempo de la canción.

11.5 Función updateTimeTracks

La función **updateTimeTracks** es utilizada para controlar la posición y el tiempo restante de dos pistas de audio, la pista de música más rápida y la pista de música más suave. La función toma dos argumentos booleanos **timesH** y **timesS**, los cuales indican si se deben actualizar los valores para la música más rápida o para la música más suave.

Dentro de la función, se utilizan los argumentos para determinar si se actualizan los valores para la rápida o suave. Primero, se comprueba si **timesH** es verdadero. Si es así, se actualizan los valores para la rápida: el tiempo actual en la pista se obtiene mediante **sbHardTrack.progress** y se formatea mediante la función **getFormattedStopWatch**, el tiempo restante se obtiene restando la duración total de la pista menos la posición actual en la misma.

Luego, se comprueba si **timesS** es verdadero. Si es así, se realiza un proceso similar para la pista suave.

En resumen, la función **updateTimeTracks** es utilizada para actualizar la posición y el tiempo restante de las pistas de audio y muestra esa información en la interfaz de usuario.

11.6 Reproducción / Parar Música

```

if (!startButtonClicked) {
    startButtonClicked = true
    startTime()
    manageEnableButtonsRun(e_reset = false, e_run = true)//Desactiva Finalizar/Reset y Activa Carrera
    (if (isRunning) mpHard else mpSoft)?.start() //si esta en carrera reproduce una musica u otra
} else {
    startButtonClicked = false
    stopTime()
    manageEnableButtonsRun(e_reset = true, e_run = true)//Activa Finalizar/reset y Activa Carrera
    (if (isRunning) mpHard else mpSoft)?.pause() //si esta en carrera pausa una musica u otra
}

```

Dentro de la función manageRun controlamos si se está corriendo reproducir una música u otra.

Este código representa una lógica de alternancia para un botón Start/Stop. Si el botón no ha sido pulsado, se activa la función startTime(), se desactiva el botón "Finalizar/Reset" y se activa el botón "Carrera". También se reproduce una música fuerte o suave dependiendo del estado de isRunning. Si el botón ha sido pulsado, se detiene la función startTime(), se activa el botón "Finalizar/Reset" y se pausa la música fuerte o suave.

12. Implementación del GPS

Con la implementación del GPS vamos a registrar las posiciones del usuario mediante geolocalización. La app captura las coordenadas que luego nos permitirá hacer distintos cálculos como la distancia entre diferentes coordenadas, velocidad, velocidad media etc.

Primero hay que importar en el Gradle la librería de Google Play Services Location

```

dependencies { implementation
    'com.google.android.gms:play-services-location:17.0.0' implementation
    'com.google.android.gms:play-services-maps:17.0.0' }

```

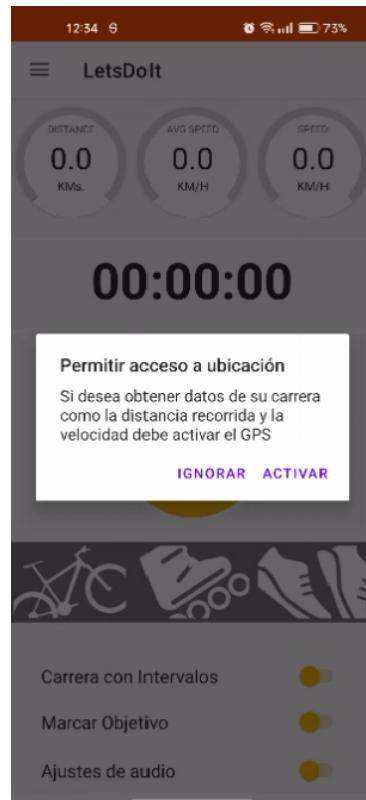
```

<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-feature android:name="android.hardware.location.gps" />

```

También será necesario añadir permisos en el archivo AndroidManifest.xml

12.1 Consultar y Activar GPS



Cuando el usuario pulse en el botón de iniciar carrera la app comprobará si tiene activado el servicio de localización, en el caso de que no lo tenga activado llevará a la pantalla de activación.

```
private fun activationLocation() {
    val intent = Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS)
    startActivity(intent)
}

new *
private fun isLocationEnabled(): Boolean {
    val locationManager: LocationManager = getSystemService(Context.LOCATION_SERVICE) as LocationManager
    //Devuelve un true si alguno de los dos servicios está habilitado
    return locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER)
        || locationManager.isProviderEnabled(LocationManager.NETWORK_PROVIDER)
}
```

Con "activationLocation" se abre la configuración de la ubicación en el dispositivo. Crea un objeto "Intent" con la acción "**Settings.ACTION_LOCATION_SOURCE_SETTINGS**" para abrir la configuración de la ubicación. Luego, se inicia una nueva actividad utilizando "startActivity" con el objeto "Intent" creado. Esto abrirá la pantalla de configuración de la ubicación en el dispositivo, permitiendo al usuario activar o desactivar el servicio de ubicación.

isLocationEnabled verifica si el servicio de ubicación está habilitado en el dispositivo. Utiliza la clase **LocationManager** para obtener el servicio de ubicación del sistema y luego verifica si alguno de los dos proveedores de ubicación (GPS o red) está habilitado. Si alguno de ellos está habilitado, devuelve un valor de verdadero, de lo contrario, devuelve falso.

```

private fun manageStartStop() {
    //Si TimeInSeconds no es cero es que la carrera está iniciada
    // Y Tambien Comprobamos si tiene habilitada la localización
    if (timeInSeconds == 0L && !isLocationEnabled()) {
        AlertDialog.Builder(context: this) AlertDialog.Builder()
            .setTitle(getString(R.string.alertActivationGPSTitle)) AlertDialog.Builder()
            .setMessage(getString(R.string.alertActivationGPSDescription))
            .setPositiveButton(R.string.aceptActivationGPS, //En caso de que pulse positivo
                DialogInterface.OnClickListener { dialog, which ->
                    activationLocation() //Activamos la ubicación
                })
            .setNegativeButton(R.string.ignoreActivationGPS, //En el caso de que pulse negativo
                DialogInterface.OnClickListener { dialog, which ->
                    activatedGPS = false // No hay GPS
                    manageRun() // Empieza la carrera
                })
            .setCancelable(true) //Que se pueda cancelar el mensaje
            .show() // Que lo muestre
    } else manageRun() //Se ejecuta si ya la he dado al botón empezar
}

```

Primero vamos a verificar si el servicio de ubicación si está habilitado y muestra un cuadro de diálogo para activarlo si no lo está. La función principal "**manageStartStop**" controla el inicio y detención de la carrera, y se inicia si el servicio de ubicación está habilitado o si el tiempo en segundos no es igual a cero.

12.2 Objeto FusedLocationProviderClient

FusedLocationProviderClient es una clase proporcionada por la biblioteca de ubicación de Google Play Services. Es una clase de nivel superior que permite a los desarrolladores obtener la ubicación actual del dispositivo de manera eficiente y precisa. La clase utiliza la combinación de diferentes tecnologías de ubicación, como GPS, Wi-Fi y datos móviles, para obtener la ubicación más precisa posible. La clase es fácil de usar y proporciona una forma simple y eficiente de obtener la ubicación en una aplicación.

```

val REQUIRED_PERMISSIONS_GPS =
    arrayOf(
        Manifest.permission.ACCESS_COARSE_LOCATION,
        Manifest.permission.ACCESS_FINE_LOCATION
    )

```

```

private fun initPermissionsGPS() {
    //Comprobamos si estan aprobados los permisos
    if (allPermissionsGrantedGPS())
        //Nos da todos los datos a través del servicio de localización
        fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
    else
        requestPermissionLocation() //en caso contrario vamos a solicitar al usuario permisos
}
new*
private fun allPermissionsGrantedGPS() = REQUIRED_PERMISSIONS_GPS.all { it String
    //Si todos los componentes del array REQUIRED..
    ContextCompat.checkSelfPermission(baseContext, it) == PackageManager.PERMISSION_GRANTED
}
new*
private fun requestPermissionLocation() {
    ActivityCompat.requestPermissions(
        activity: this, arrayOf(
            Manifest.permission.ACCESS_COARSE_LOCATION,
            Manifest.permission.ACCESS_FINE_LOCATION
        ), PERMISSION_ID
    )
}

```

Con estas funciones se controla los permisos de ubicación GPS . La función "**initPermissionsGPS**" verifica si los permisos necesarios (**ACCESS_COARSE_LOCATION** y **ACCESS_FINE_LOCATION**) están aprobados y en caso de que sí, proporciona todos los datos de ubicación a través del servicio de localización "**fusedLocationClient**". Si los permisos no están aprobados, la función "**requestPermissionLocation**" solicita al usuario los permisos necesarios. La función "**allPermissionsGrantedGPS**" verifica si todos los permisos necesarios en el array "**REQUIRED_PERMISSIONS_GPS**" han sido otorgados.

12.3 Administración de Localización y Datos de ubicación.

```

//Controlamos la administración localización en el intervalo indicado
//En este caso por motivos de no sobrecargar el sistema , se comprueba cada 4 segundos
// (los segundos se indican en la constante INTERVAL_LOCATION)
if (activatedGPS && timeInSeconds.toInt() % INTERVAL_LOCATION == 0) manageLocation() //Administraremos la localización

```

Dentro del cronómetro vamos a controlar la administración de la localización en un determinado intervalo, se podría hacer una localización más exhaustiva pero por motivos de no sobrecargar el sistema se comprueba cada 4 segundos.

```

private fun manageLocation() {
    if (checkPermission()) { //comprobamos permisos
        if (isLocationEnabled()) {
            if (ActivityCompat.checkSelfPermission(
                context: this,
                Manifest.permission.ACCESS_FINE_LOCATION
            ) ==
                PackageManager.PERMISSION_GRANTED
                && ActivityCompat.checkSelfPermission(
                    context: this,
                    Manifest.permission.ACCESS_COARSE_LOCATION
                ) ==
                PackageManager.PERMISSION_GRANTED
            ) {
                fusedLocationClient.lastLocation.addOnSuccessListener { it: Location! } //Solicitamos la ultima localización
                requestNewLocationData()
            }
        }
    } else activationLocation()
} else requestPermissionLocation()
}

```

ManageLocation

```

new *

private fun checkPermission(): Boolean {
    return ActivityCompat.checkSelfPermission(
        context: this,
        Manifest.permission.ACCESS_COARSE_LOCATION
    ) ==
        PackageManager.PERMISSION_GRANTED && ActivityCompat.checkSelfPermission(
            context: this,
            Manifest.permission.ACCESS_FINE_LOCATION
        ) ==
        PackageManager.PERMISSION_GRANTED
}

```

checkpermission

```

new *
@SuppressLint("MissingPermission", "VisibleForTests") //Olvida los permisos la comprobacion esta hecha en manageLocation()
private fun requestNewLocationData() {
    val mLocationRequest = LocationRequest()
    mLocationRequest.pr val mLocationRequest: LocationRequest ACCURACY
    mLocationRequest.in letsdoitapp.app.main ...
    mLocationRequest.fa ...
    mLocationRequest.numUpdates = 1
    fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
    fusedLocationClient.requestLocationUpdates(
        mLocationRequest,
        mLocationCallBack,
        Looper.myLooper()
    )
}

```

requestNewLocationData

```

private val mLocationCallBack = object : LocationCallback() {
    override fun onLocationResult(locationResult: LocationResult) {
        //Variable donde guardamos todos los datos (Ubicación del Usuario)
        val mLastLocation: Location =
            locationResult.lastLocation // Solo funciona con la version 'com.google.android.gms:play-services-location:17.0.
        init_lt = mLastLocation.latitude
        init_ln = mLastLocation.longitude

        //Si nos encontramos en una posición de la carrera mayor de cero registramos la posición
        if (timeInSeconds > 0L) registerNewLocation(mLastLocation)
    }
}

```

mLocationCallback

En resumen :

- Primero se consulta si el GPS está activado y se hace una comprobación cada x segundos.
- Despues , manageLocation , administra la localización que es donde tenemos toda la secuencia de comprobaciones necesarias para el GPS.
- Hacemos uso de checkPermission, donde hacemos las comprobaciones necesarias de los permisos de Android.
- Si los permisos están dados, verifica si el servicio de localización está activado usando isLocationEnabled().
- Si los permisos no están dados, se solicitan los permisos de localización.
- Si el servicio está activado,con fusedLocationClient.lastLocation, obtenemos la última localización haciendo uso de requestNewLocationData.
- Se configura una petición de ubicación en requestNewLocationData, y se inicia el seguimiento de la ubicación del usuario con un LocationCallback.
- Usamos fusedLocationClient.requestLocationUpdates con un callback donde con el LocationResult conseguimos el dato necesario.

12.4 Cálculo de Distancia y Velocidad

Ahora que ya tenemos datos de la posición del dispositivo , necesitamos variables para almacenar los datos de altitud, longitud, latitud , etc.

```

new *

private fun calculateDistance(n_lt: Double, n_lg: Double): Double {
    //Formula Calcula distancia entre dos coordenadas
    val radioTierra = 6371.0 //en kilómetros
    val dLat = Math.toRadians(n_lt - latitude)
    val dLng = Math.toRadians(n_lg - longitude)
    val sindLat = sin(x: dLat / 2)
    val sindLng = sin(x: dLng / 2)
    val va1 =
        sindLat.pow(x: 2.0) + (sindLng.pow(x: 2.0)
            * cos(Math.toRadians(latitude)) * cos(
            Math.toRadians(n_lt)
        ))
    val va2 = 2 * atan2(sqrt(va1), sqrt(x: 1 - va1))
    var n_distance = radioTierra * va2 //Distancia que hemos recorrido en el intervalo

    //if (n_distance < LIMIT_DISTANCE_ACCEPTED) distance += n_distance

    //La distancia acumulada la voy registrando . Distancia total + la distancia que en cada
    //Intervalo hemos registrado
    distance += n_distance
    return n_distance
}

```

La función "calculateDistance" se utiliza para calcular la distancia entre dos coordenadas geográficas. Utiliza la **fórmula de Haversine** para calcular la distancia en kilómetros entre dos puntos en la superficie terrestre. Toma como entrada las coordenadas geográficas n_lt y n_lg y las compara con las coordenadas guardadas previamente en las variables "latitude" y "longitude". La función devuelve la distancia entre los dos puntos. También mantiene una variable "distance" que acumula la distancia total recorrida.

La fórmula de Haversine es una fórmula matemática que se utiliza para calcular la distancia entre dos puntos en una esfera, como la Tierra. La fórmula toma en cuenta la curvatura de la Tierra y permite calcular la distancia en kilómetros o millas entre dos puntos con coordenadas geográficas (latitud y longitud).

La fórmula es:

$$\begin{aligned}
 a &= \sin^2(\Delta\text{lat}/2) + \cos(\text{lat1}) \cdot \cos(\text{lat2}) \cdot \sin^2(\Delta\text{long}/2) \\
 c &= 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a}) \\
 d &= R \cdot c
 \end{aligned}$$

donde R es el radio de la Tierra (6,371 km), lat1 y lat2 son las latitudes de los puntos en radianes, Δlat y Δlong son los cambios en latitud y longitud, y d es la distancia entre los puntos.

```
private fun updateSpeeds(d: Double) {
    //Funcion de calculo de distancias Velocidad = Distancia / Tiempo
    //Paso la distancia como parámetro , km se pasan a metros
    //la distancia se calcula en km, así que la pasamos a metros para el calculo de velocidad
    //convertirmos m/s a km/h multiplicando por 3.6
    speed = ((d * 1000) / INTERVAL_LOCATION) * 3.6
    if (speed > maxSpeed) maxSpeed = speed
    avgSpeed = ((distance * 1000) / timeInSeconds) * 3.6
}
```

La función "updateSpeeds" se utiliza para calcular la velocidad en km/h. Toma un parámetro "d" que representa la distancia en kilómetros. La función convierte "d" a metros y luego divide la distancia por el tiempo de intervalo para obtener la velocidad en m/s. Finalmente, la velocidad se multiplica por 3.6 para convertirla en km/h. La función también actualiza la velocidad máxima y la velocidad promedio.

```
private fun refreshInterfaceData() {
    //Para refrescar los datos asignamos a todos los circular seekbar los datos
    // Tambien se actualizan los textView Si hemos hecho algún record
    tvCurrentDistance.text = roundNumber(distance.toString(), decimals: 2)
    tvCurrentAvgSpeed.text = roundNumber(avgSpeed.toString(), decimals: 1)
    tvCurrentSpeed.text = roundNumber(speed.toString(), decimals: 1)

    //Actualizamos el color del TextView para mostrar que hemos superado un record
    csbCurrentDistance.progress = distance.toFloat() //Actualizamos Distancia
    if (distance > totalsSelectedSport.recordDistance!!) {...}

    csbCurrentAvgSpeed.progress = avgSpeed.toFloat() //Actualizamos velocidad Media
    if (avgSpeed > totalsSelectedSport.recordAvgSpeed!!) {...}

    if (speed > totalsSelectedSport.recordSpeed!!) {...} else {...}
    csbCurrentSpeed.progress = speed.toFloat() //Actualizamos velocidad
}
```

13. Implementación de Maps de Google

Después de geolocalizar cada punto de la carrera ahora se va implementar la representación del recorrido en el mapa.

Para implementar Google Maps en nuestra app vamos a seguir estos pasos generales:

1. Registrar en la Consola de Desarrolladores de Google y habilitar Google Maps Platform para obtener una clave API.
2. Agregar la dependencia de Google Maps en el archivo build.gradle.

3. En el archivo XML, agregar un fragmento de mapa que actúe como contenedor de Google Maps.
4. En la clase Java, obtener una referencia al fragmento y configurar la ubicación y el tipo de mapa deseados.
5. Implementar la lógica necesaria para manipular el mapa, como marcadores, cámaras y eventos de clic, utilizando las API de Google Maps disponibles.
6. <https://developer.android.com/training/maps?hl=es-419>

13.1 Creación del Mapa

```
private fun createMapFragment() {
    val mapFragment =
        supportFragmentManager.findFragmentById(R.id.fragmentMap) as SupportMapFragment?
    mapFragment?.getMapAsync(callback: this)
}
```

Este código crea un fragmento de mapa en una aplicación Android. Primero, busca un fragmento con un ID específico en el administrador de fragmentos de la actividad y lo guarda en una variable. Luego, verifica si el fragmento encontrado es nulo y si no lo es, invoca un método para cargar el mapa de forma asíncrona y específica la actividad que implementa una interfaz para recibir una notificación cuando el mapa esté listo para ser utilizado.

```
private fun centerMap(lt: Double, ln: Double) {
    val posMap = LatLng(lt, ln)
    map.animateCamera(CameraUpdateFactory.newLatLngZoom(posMap, zoom: 16f), durationMs: 1000, callback: null)
}

new *
override fun onMapReady(googleMap: GoogleMap) {
    map = googleMap
    googleMap.mapType = GoogleMap.MAP_TYPE_HYBRID
    enableMyLocation()
    map.setOnMyLocationButtonClickListener(this)
    map.setOnMyLocationClickListener(this)
    map.setOnMapLongClickListener { mapCentered = false }
    map.setOnMapClickListener { mapCentered = false }
    manageLocation() //Capturamos los datos
    centerMap(initialLatitude, initialLongitude)// Desplaza la cámara al punto que le hayamos indicado
}
```

Tenemos una función llamada **centerMap** que toma coordenadas de latitud y longitud y mueve la cámara del mapa hacia esa posición con un nivel de zoom y tiempo determinados. También tenemos la función **onMapReady** que se llama cuando el mapa está listo para ser usado y donde se realiza la configuración básica del mapa, como establecer el tipo de mapa, habilitar la ubicación del usuario y definir los listeners para eventos en el mapa. Finalmente, se invoca la función **centerMap** para mover la cámara del mapa a unas coordenadas específicas.

```

fun callShowHideMap(v: View) {
    if (allPermissionsGrantedGPS()) { // Están los permisos ?
        if (lyMap.height == 0) { // Si el layout tiene cero es que está oculto
            setHeightLinearLayout(lyMap, value: 1250)//medida que hay que darle para ver el mapa
            animateViewofFloat(
                lyFragmentMap,
                attr: "translationY",
                value: 0f,
                time: 0
            )//Animamos la vista del hijo
            ivOpenClose.rotation = 180f// le damos la vuelta al imageview
        } else { //en el caso contrario ( que este desplegado ) lo encogemos
            setHeightLinearLayout(lyMap, value: 0)
            lyFragmentMap.translationY = -300f
            ivOpenClose.rotation = 0f
        }
    } else requestPermissionLocation()//Si no están los permisos hay que pedirlos
}

```

La función **callShowHideMap** controla la visibilidad del fragmento de mapa en la interfaz de usuario. Primero se comprueba si todos los permisos de ubicación están otorgados, y si es así, se determina si el fragmento está oculto o visible. Si está oculto, se le da un alto específico y se anima hacia abajo para ser visible. Si está visible, se encoge hacia arriba. Si los permisos no están otorgados, se solicitan.

```

private fun enableMyLocation() {
    if (!::map.isInitialized) return//si el mapa no está inicializado se sale
    //Si no están aprobados los permisos se sale ( Comprobacion de Google obligatoria )
    if (ActivityCompat.checkSelfPermission(context: this, Manifest.permission.ACCESS_FINE_LOCATION)
        != PackageManager.PERMISSION_GRANTED

        && ActivityCompat.checkSelfPermission(context: this, Manifest.permission.ACCESS_COARSE_LOCATION)
        != PackageManager.PERMISSION_GRANTED
    ) {
        requestPermissionLocation()
        return
    } else map.isMyLocationEnabled = true
}

```

La función **enableMyLocation** activa la ubicación del usuario en el mapa. Primero se verifica si el mapa ha sido inicializado, y si no es así, la función vuelve. Luego, se comprueba si se han otorgado los permisos de ubicación necesarios y, si no es así, se solicitan los permisos. Si los permisos están otorgados, se habilita la ubicación del usuario en el mapa.

```

new *
fun changeTypeMap(v: View) {// Cambia los modos y las imageview de modos híbrido/normal
    if (map.mapType == GoogleMap.MAP_TYPE_HYBRID) {
        map.mapType = GoogleMap.MAP_TYPE_NORMAL
        ivTypeMap.setImageResource(R.drawable.map_type_hybrid)
    } else {
        map.mapType = GoogleMap.MAP_TYPE_HYBRID
        ivTypeMap.setImageResource(R.drawable.map_type_normal)
    }
}

new *
fun callCenterMap(v: View) {
    mapCentered = true
    if (latitude == 0.0) centerMap(
        initialLatitude,
        initialLongitude
    ) //si hace al inicio esta en cero lo centramos con datos iniciales
    else centerMap(latitude, longitude) // Si tenemos datos recuperamos datos
}

```

La función **changeTypeMap** permite cambiar el tipo de mapa que se muestra en la aplicación. Si el mapa actual es de tipo híbrido, se cambia a un mapa normal y se actualiza la imagen que representa el tipo de mapa. Si el mapa actual es normal, se cambia a un mapa híbrido y se actualiza la imagen que representa el tipo de mapa.

La función **callCenterMap** permite centrar el mapa en la ubicación actual. Si la ubicación actual es 0.0, se centra el mapa en las coordenadas iniciales, pero si se tiene una ubicación actual válida, se centra el mapa en esa ubicación.

```

//cuando se piden permisos sale la ventana y regresa con nuevos permisos tenemos que saber que ha cambiado
new *
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<out String>,
    grantResults: IntArray
) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    when (requestCode) {
        //Analizamos el permiso de la ubicación
        LOCATION_PERMISSION_REQ_CODE -> {
            if (grantResults.isNotEmpty() && grantResults[0] == PackageManager.PERMISSION_GRANTED) // si ha sido aprobado
                lyOpenerButton.isEnabled = true //habilitamos el layout
            else {
                if (lyMap.height > 0) { //layout padre// Si es mayor de 0 está desplegado
                    setHeightLinearLayout(lyMap, value: 0) //encogemos y el hijo lo desplazamos hacia arriba
                    lyFragmentMap.translationY = -300f
                    ivOpenClose.rotation = 0f
                }
                lyOpenerButton.isEnabled = false// En caso contrario lo desabilitamos
            }
        }
    }
}

```

Esta es una implementación de la función `onRequestPermissionsResult` que maneja el resultado de una solicitud de permisos. Cuando se solicitan permisos, una ventana aparece y regresa con nuevos permisos. En esta función, se comprueba el código de permiso (LOCATION_PERMISSION_REQ_CODE) y, si ha sido aprobado (`grantResults[0] == PackageManager.PERMISSION_GRANTED`), se habilita un LinearLayout (`lyOpenerButton`). Si no ha sido aprobado, se verifica si un layout padre (`lyMap`) es mayor que cero y, si lo es, se reduce su tamaño y se desplaza hacia arriba un layout hijo (`lyFragmentMap`) y se rota un ImageView (`ivOpenClose`). Finalmente, se deshabilita el `lyOpenerButton` si no ha sido aprobado.

```
private fun updateLocationLimits(location: Location) {
    // Si es nulo le ponemos todos los valores
    if (minLatitude == null) {
        minLatitude = latitude
        maxLatitude = latitude
        minLongitude = longitude
        maxLongitude = longitude
    }

    //Controlamos si es un valor mínimo/maximo de todos los valores
    if (latitude < minLatitude!!) minLatitude = latitude
    if (latitude > maxLatitude!!) maxLatitude = latitude
    if (longitude < minLongitude!!) minLongitude = longitude
    if (longitude > maxLongitude!!) maxLongitude = longitude

    //Si la altitud es por encima de la maxima o minima
    if (location.hasAltitude()) {
        if (maxAltitude == null) {
            maxAltitude = location.altitude
            minAltitude = location.altitude
        }
        if (location.latitude > maxAltitude!!) maxAltitude = location.altitude
        if (location.latitude < minAltitude!!) minAltitude = location.altitude
    }
}
```

Se verifica si la variable `minLatitude` es nula, y si lo es, se asignan los valores `latitude` a todas las variables relacionadas con la ubicación (`minLatitude`, `maxLatitude`, `minLongitude`, `maxLongitude`). Luego, se comparan `latitude` y `longitude` con sus respectivos valores mínimos y máximos para verificar si son mínimos o máximos de todos los valores. Finalmente, si `location` tiene información de altitud, se comparan `location.altitude` con los valores mínimo y máximo de la altitud (`minAltitude` y `maxAltitude`) y se actualizan según corresponda.

13.2 Control de Velocidades Admitidas

```

const val INTERVAL_LOCATION = 4
const val LIMIT_DISTANCE_ACCEPTED_BIKE = 0.04 * INTERVAL_LOCATION
//LIMIT_DISTANCE_ACCEPTED correcto sería 0.04 (40m)
//40m son 144km/h. Un ciclista profesional en descenso pudiera alcanzar 130

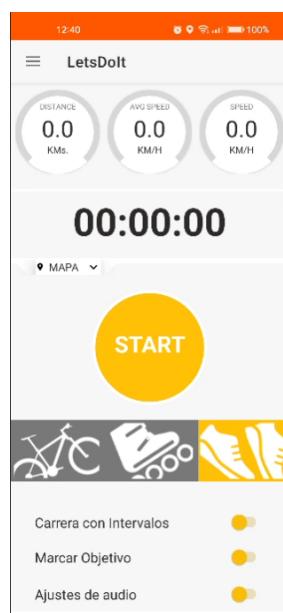
const val LIMIT_DISTANCE_ACCEPTED_ROLLERSKATE = 0.035 * INTERVAL_LOCATION
//LIMIT_DISTANCE_ACCEPTED correcto sería 0.035 (35m)
//35m son 126km/h.
//In 21 February, 2016 in Brazil, an Italian skater named Sandro Bovo entered the
//Guinness Book of World Records by finishing with an average speed of 124.67 km/h
💡 //in downhill roller skating      https://www.youtube.com/watch?v=zL53mi3kIrA

const val LIMIT_DISTANCE_ACCEPTED_RUNNING = 0.012 * INTERVAL_LOCATION
//LIMIT_DISTANCE_ACCEPTED correcto sería 0.012 (12 m)
//12m son 43.2km/h. Usain Bolt alcanza los 42km/h

```

Se puede considerar que estos valores constantes están relacionados con la velocidad máxima esperada para cada deporte/modo de transporte. Estas constantes pueden ser útiles para determinar si una ubicación es inválida o no, dependiendo de la actividad en cuestión. Por ejemplo, se puede calcular la distancia entre dos puntos de ubicación y compararla con la constante correspondiente para determinar si se trata de una velocidad válida o no para esa actividad.

13.3 Función selectSport



función selectSport

Con esta función se implementa la lógica para seleccionar un deporte, como andar en bicicleta, patinar con patines en línea o correr. Cuando un usuario toca uno de los tres botones para cada deporte, se selecciona el deporte correspondiente y se establecen sus propiedades (por ejemplo, color de fondo, límite de distancia). La información del deporte seleccionado se utiliza entonces para actualizar los datos en la aplicación.

13.4 Trazado de la Trayectoria de la actividad en el Mapa

```
private fun createPolyLines(listPosition: Iterable<LatLng>) {
    //creamos la polyline
    val polylineOptions = PolylineOptions()
        .width(25f)
        .color(ContextCompat.getColor(context: this, R.color.salmon_dark))
        .addAll(listPosition)

    //la pintamos en el mapa // devuelve un valor y lo guardamos y redondeamos
    val polyline = map.addPolyline(polylineOptions)
    polyline.startCap = RoundCap()

}
```

La función "**createPolyLines**" toma una lista de objetos "**LatLng**" que representan las posiciones a conectar mediante la línea.

La variable "**polylineOptions**" se inicializa con un objeto "**PolylineOptions**" que define la apariencia de la línea, incluyendo su ancho, color y los puntos a conectar. El método "**addAll**" se utiliza para agregar todos los puntos en la lista "**listPosition**" a la Polyline.

Después de configurar las opciones, se utiliza el método "**addPolyline**" en el objeto "**map**" para añadir la Polyline al mapa. Finalmente, se establece el "**startCap**" de la Polyline para que tenga una forma redonda en su extremo inicial.

14. Implementación de Preferencias del Usuario

SharedPreferences en Android es una clase que permite a las aplicaciones guardar y recuperar pequeñas cantidades de datos persistentes, como pares clave-valor, en el sistema. Estos datos se almacenan en un archivo XML en el dispositivo y se pueden recuperar cuando la aplicación se ejecuta de nuevo en el futuro.

SharedPreferences es útil para guardar configuraciones de usuario, estados de la aplicación o cualquier otro tipo de información que se deba mantener aun después de que la aplicación haya sido cerrada. Es una forma fácil y eficiente de almacenar y recuperar datos en Android sin tener que utilizar una base de datos.

Para utilizar **SharedPreferences**, primero se debe obtener una instancia a través del método "**getSharedPreferences**" de la clase "Context". Luego, se pueden usar métodos como "**putInt**",

"**putString**", etc. para guardar valores y métodos como "**getInt**", "**getString**", etc. para recuperar valores previamente guardados.

14.1 Función savePreferences. Guardar Preferencias

La función comienza limpiando cualquier valor previamente guardado con el método "clear". Luego, se utiliza el método "apply" para agregar los nuevos valores a SharedPreferences.

Dentro del método "apply", se guardan diferentes valores, como correos electrónicos de usuario, proveedores de sesión, deportes seleccionados, valores booleanos para diferentes modos, valores enteros para diferentes duraciones y distancias, valores flotantes para progreso y máximo en una barra circular, y valores de volumen para diferentes tipos de notificaciones.

Todos estos valores se guardan con pares clave-valor, donde la clave es una constante de cadena definida previamente y el valor es el valor actual del elemento de la interfaz de usuario o de otras variables.

Finalmente, se llama de nuevo al método "apply" para guardar los cambios en SharedPreferences.

14.2 Función recoveryPreferences. RecuperarPreferencias

Su objetivo es recuperar los valores previamente guardados en "**savePreferences**". Si el usuario guardado en SharedPreferences coincide con el usuario actual, se recuperan los valores de los distintos campos, como el deporte seleccionado, las configuraciones del modo de intervalo, las configuraciones de los desafíos, etc. y se aplican a la aplicación. Si no coincide, se establece un valor por defecto para algunos campos, como el deporte seleccionado.

14.3 Restablecer Preferencias

```
private fun alertClearPreferences() {
    AlertDialog.Builder( context: this ) AlertDialogBuilder
        .setTitle( getString( R.string.alertClearPreferencesTitle ) ) AlertDialogBuilder!
        .setMessage( getString( R.string.alertClearPreferencesDescription ) )
        .setPositiveButton( android.R.string.ok,
            DialogInterface.OnClickListener { dialogo, which ->
                callClearPreferences()
            }
        )
        .setNegativeButton( android.R.string.cancel,
            DialogInterface.OnClickListener { dialogo, which ->
                ...
            }
        )
        .setCancelable( true )
        .show()
}

new *
private fun callClearPreferences() {
    editor.clear().apply()
    Toast.makeText( context: this, text: "Tus ajustes han sido reestablecidos :)", Toast.LENGTH_SHORT ).show()
}
```

La función **alertViewClearPreferences** muestra un cuadro de diálogo con título y mensaje especificados. Tiene dos botones: "OK" y "Cancelar". Si se pulsa "OK", se llama a la función **callClearPreferences** que borra todas las preferencias almacenadas y muestra un mensaje Toast indicando que las preferencias han sido restablecidas.

15. Implementación de Pantalla de Resultados



Hemos ido capturando los datos de la actividad, ahora el siguiente paso es mostrar la pantalla de resultados donde se visualizan los datos generados.

De esto se encarga la función llamada "**showPopUp**". La función deshabilita el menú principal llamado "**rIMain**" y habilita un elemento llamado "**lyPopupRun**". Luego, usa una animación para mover un layout llamado "**lyWindow**" hacia el centro de la pantalla. Finalmente, llama a la función "**loadDataPopUp**" para cargar los datos en el PopUp.

15.1 Mostrar los datos de la actividad en la pantalla de resultados

La función **showDataRun** se encarga de actualizar los valores en la interfaz de usuario .La función recupera referencias a varios elementos de la interfaz de usuario, como textviews y layouts, y actualiza sus valores en función de los datos de una carrera. Los datos incluyen la duración de la carrera, el desafío de duración (si existe), el modo de intervalos, la distancia recorrida, el desafío de distancia, la desigualdad (altura máxima y mínima) y la velocidad media y máxima. La altura de los layouts se establece en 0 si no hay datos disponibles o en 120 si hay datos disponibles para mostrar.

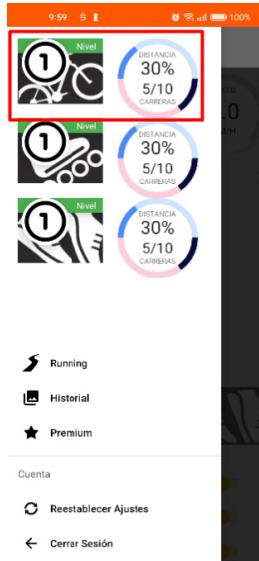
16. Implementación de la Base de Datos

Firebase Firestore es una base de datos en tiempo real que permite una gestión eficiente de los datos de los usuarios en la app de deportes. Al permitir la lectura, escritura y actualización de los datos, los usuarios podrán personalizar su experiencia en la app y ver su progreso en tiempo real. La base de datos se utilizará para calcular los niveles del usuario en función de la distancia recorrida, el número de carreras realizadas y otros indicadores relevantes en cada deporte.

Path	Value	Operations
letsdoitapp-33146	Document reference	+ Iniciar colección
levelsBike		+ Agregar documento
levelsRollerSkate		+ Iniciar colección
levelsRunning	Collection reference	+ Agregar campo
locations		DistanceTarget: 10
runsBike		RunsTarget: 1
runsRollerSkate		image: "level_1"
runsRunning		name: "turtle"
totalsBike		
totalsRollerSkate		
totalsRunning		
users		

Con los datos cargados del usuario en la pantalla de menú podemos ver la distancia, carreras , nivel de cada actividad.Para mostrar datos en pantalla se van a utilizar distintas clases de datos Run, Location, Levels Totals.

Además de los totales, los niveles se asociarán a una imagen que se mostrará en la pantalla de menú, permitiendo a los usuarios ver su progreso visualmente. Esto también aumentará la motivación de los usuarios al ver su progreso y alcanzar nuevos niveles.



La app también utilizará clases de datos específicas, como Run, Location y Levels Totals, para mostrar los datos en pantalla. La clase Run contendrá información sobre cada carrera realizada por el usuario, mientras que la clase Location permitirá almacenar la información de la ubicación geográfica de cada carrera. La clase Levels Totals, por su parte, mantendrá la información de los totales necesarios para calcular los niveles.

En resumen, la implementación de Firebase Firestore en la app mejorará significativamente la experiencia de usuario al permitir una gestión de datos personalizada y en tiempo real. Los usuarios podrán ver su progreso y alcanzar nuevos niveles de una manera visual y motivadora. La base de datos será un componente clave para el éxito de la app y para que los usuarios disfruten de una experiencia única y personalizada.

16.1 Clases de datos . Niveles y Totales.

```

data class Totals(
    var recordAvgSpeed: Double? = null,
    var recordDistance: Double? = null,
    var recordSpeed: Double? = null,
    var totalDistance: Double? = null,
    var totalRuns: Int? = null,
    var totalTime: Int? = null
)

data class Level(
    var name: String? = null,
    var image: String? = null,
    var RunsTarget: Int? = null,
    var DistanceTarget: Int? = null
)

```

Creación de clases de datos para almacenar datos de niveles y totales. Cada una tendrá campos con valores enteros y cadenas de texto o decimales. Es importante asegurarse de que los tipos de datos en ambos lados correspondan exactamente y manejarlos de la misma manera para evitar errores que causen la interrupción de la aplicación. Se crean estas dos clases y se definen los campos correspondientes en cada una de ellas.

16.2 Instancias de lectura y escritura en la base de datos.



```

object Database {

    fun loadTotalSport(sport: String) {
        var collection = "totals$sport"
        var dbTotalsUser = FirebaseFirestore.getInstance()
        dbTotalsUser.collection(collection).document(useremail).get()
            .addOnSuccessListener { document ->
                if (document.data?.size != null) {
                    var total = document.toObject(Totals::class.java)
                    when (sport) {
                        "Bike" -> totalsBike = total!!
                        "RollerSkate" -> totalsRollerSkate = total!!
                        "Running" -> totalsRunning = total!!
                    }
                } else {
                    val dbTotal: FirebaseFirestore = FirebaseFirestore.getInstance()
                    dbTotal.collection(collection).document(useremail).set(
                        hashMapOf // guardamos los datos
                            "recordAvgSpeed" to 0.0,
                            "recordDistance" to 0.0,
                            "recordSpeed" to 0.0,
                            "totalDistance" to 0.0,
                            "totalRuns" to 0,
                            "totalTime" to 0
                    )
                }
            }
        /* */
        sportsAdded++
    }
}

```

La función **"loadTotalSport"** toma un parámetro **"sport"** que especifica el deporte para el que se cargarán los datos. La variable **"collection"** se utiliza para determinar la colección en Firestore que se usará para obtener los datos. Luego, se crea una instancia de Firebase Firestore y se utiliza para recibir los datos de la colección correspondiente.

Si los datos se reciben correctamente, se verifica si existe algún documento y, en caso contrario, se guardan los datos en Firestore. En caso de error, se registra un mensaje de error.

En resumen, esta función se utiliza para cargar y guardar los datos totales de un deporte específico en Firebase Firestore.

La línea de código **"var total = document.toObject(Totals::class.java)"** es un ejemplo de cómo se puede convertir un documento de **Firebase Firestore** en un objeto de una clase Kotlin.

La función **"toObject"** pertenece a la biblioteca de Firebase para Android y se utiliza para convertir un documento de Firebase Firestore en un objeto de una clase determinada. En este caso, la clase es **"Totals"**, que se especifica como parámetro en **"Totals::class.java"**.

A la variable **"total"** se le asigna el resultado de la conversión, de modo que después se puede acceder a sus propiedades y métodos como a cualquier otro objeto en Kotlin.

```
data class Totals{  
    var recordAvgSpeed: Double? = null,  
    var recordDistance: Double? = null,  
    var recordSpeed: Double? = null,  
    var totalDistance: Double? = null,  
    var totalRuns: Int? = null,  
    var totalTime: Int? = null  
}
```

Ahora la variable total es un objeto de la clase Totals

```
// Enviamos datos de bbdd a las variables globales  
when (sport) {  
    "Bike" -> totalsBike = total!!  
    "RollerSkate" -> totalsRollerSkate = total!!  
    "Running" -> totalsRunning = total!!  
}
```

La estructura "when" que se utiliza en el código es una forma de realizar una selección múltiple en Kotlin. La estructura "when" evalúa una expresión y ejecuta un bloque de código correspondiente a la primera rama que se ajusta a la condición.

En este caso, la expresión evaluada es la variable "sport", que especifica el deporte para el que se han cargado los datos totales. Si "sport" es igual a "Bike", se asigna el objeto "total" a la variable "totalsBike". Si "sport" es igual a "RollerSkate", se asigna el objeto "total" a la variable "totalsRollerSkate", y si "sport" es igual a "Running", se asigna el objeto "total" a la variable "totalsRunning".

La doble exclamación (!!)" en "total!!" se utiliza para hacer una llamada de seguridad. Esto significa que se fuerza la conversión a un tipo no nulo. Ya hemos hecho la comprobación antes en la línea de código "if (document.data?.size != null)" que se evalúa como verdadera si el documento devuelto por Firebase Firestore contiene datos y como falsa si está vacío.

16.3 Configuración de Niveles

```

| private fun setLevelSport(sport: String) {
|     val dbLevels: FirebaseFirestore = FirebaseFirestore.getInstance()
|     dbLevels.collection( collectionPath: "levels$sport" ) CollectionReference
|         .get() Task<QuerySnapshot!>
|             .addOnSuccessListener { documents ->
|                 for (document in documents) {
|                     when (sport) {
|                         "Bike" -> levelsListBike.add(document.toObject(Level::class.java))
|                         "RollerSkate" -> levelsListRollerSkate.add(document.toObject(Level::class.java))
|                         "Running" -> levelsListRunning.add(document.toObject(Level::class.java))
|                     }
|                 }
|                 when (sport) {
|                     "Bike" -> setLevelBike()
|                     "RollerSkate" -> setLevelRollerSkate()
|                     "Running" -> setLevelRunning()
|                 }
|             }
|             .addOnFailureListener { exception ->
|                 Log.w(ContentValues.TAG, msg: "Error getting documents: ", exception)
|             }
| }
|
| 
```

En esta función se está utilizando la base de datos de Firebase Firestore para recuperar una colección de documentos con el nombre "levels\$sport".

Luego, para cada documento en la colección, se agrega a una lista levelsList correspondiente al deporte.

Por último, se llama a una función setLevel correspondiente al deporte para configurar el nivel. Todos los errores son registrados en el registro de Android con la etiqueta "ContentValues.TAG".

16.4 Carga de Datos Personales

La función setLevelBike() establece un nivel , en este caso el de la bicicleta , basado en la cantidad de tiempo, distancia y número de carreras completadas por el usuario.

El nivel se determina comparando los valores totales de tiempo, distancia y carreras completadas con los valores objetivo especificados en una lista de niveles previamente definida.

Si los valores totales son mayores o iguales a los valores objetivo para un determinado nivel, entonces el usuario es considerado un nivel más alto.

Una vez que se ha determinado el nivel, se actualiza la interfaz de usuario con el nombre y la imagen del nivel, así como con información sobre la cantidad de tiempo, distancia y carreras completadas y los valores objetivo para el nivel actual.

16.5 Establecer Records Según Deporte

La función refreshCBSSSport() establece el valor máximo y el progreso para cuatro barras de progreso circulares (csbRecordDistance, csbRecordAvgSpeed, csbCurrentDistance,

csbCurrentAvgSpeed, csbRecordSpeed, csbCurrentSpeed y csbCurrentMaxSpeed) en base al objeto totalsSelectedSport.

La función **refreshRecords()** actualiza el texto de tres vistas de texto (tvDistanceRecord, tvAvgSpeedRecord y tvMaxSpeedRecord) en base al objeto totalsSelectedSport. Si el valor de registro (distancia, velocidad media o velocidad máxima) es mayor que 0, la vista de texto correspondiente muestra ese valor como una cadena. Si no, la vista de texto se establece en una cadena vacía.

16.6 Actualizar Totales. updateTotalsUser

Esta función actualiza los totales para un deporte seleccionado por un usuario. La función actualiza los valores de la cantidad de carreras, la distancia total, el tiempo total, la distancia récord, la velocidad máxima récord y la velocidad promedio récord. Además, actualiza estos valores en una base de datos de Firebase Firestore.

16.7 Mostrar Resultados Ventana Emergente

Esta función se llama "**showHeaderPopUp**" y muestra una ventana emergente con información sobre un deporte seleccionado.

La ventana emergente incluye información sobre el nivel actual del deporte, la cantidad de carreras totales y la distancia total recorrida, el tiempo total invertido y una imagen del deporte seleccionado. La información se muestra en diferentes elementos de la interfaz de usuario, como ImageViews, TextViews y CircularSeekBar.

La función también actualiza el nivel actual del deporte y los datos relacionados (como el número de carreras y la distancia total) en función del deporte seleccionado. La información se actualiza y se muestra en la ventana emergente.

16.8 Mostrar Records y Medallas

Se necesita crear una lista de medallas para tres deportes (bicicleta, patinaje y carrera) y tres categorías (distancia, velocidad máxima, velocidad promedio). Para cada deporte y categoría, se desea identificar las tres marcas con el mejor rendimiento y asignar una medalla de oro, plata o bronce.

Para hacer esto, se usan arrays para almacenar los datos y luego utilizar algoritmos de ordenamiento para clasificar los datos y asignar las medallas correspondientes.

17. Administración de la Base de Datos

En esta sección se abordarán temas avanzados de administración de base de datos en nuestra app. Se verán consultas complejas y cómo solucionar los errores que pueden surgir. También se verá cómo generar índices para hacer consultas más eficientes y cómo mantener la coherencia de datos al borrar información. Se explicará cómo crear y borrar colecciones. Esta sección es más avanzada y se evolucionará desde conceptos básicos hasta temas más complejos.

17.1 Guardar Actividad. Identificador único.

¿ Cómo crear un identificador único para cada carrera ?

Utilizando una combinación de información del usuario, fecha y hora de inicio. Se puede guardar la fecha y la hora en variables al inicio de la carrera y luego utilizarlas para generar el identificador, asegurando que sea único y no se repita. El formato recomendado es Usuario + Fecha + Hora de inicio de carrera, lo que garantiza que dos carreras no tengan el mismo identificador. La información de fecha y hora debe ser capturada y guardada al comienzo de la carrera para tener un registro preciso.

17.2 Guardar Datos. Función saveDataRun

```
private fun saveDataRun() {  
    var id: String = useremail + dateRun + startTimeRun  
}
```

Se utiliza la clase **SimpleDateFormat** para dar formato a la fecha y hora actual en los formatos yyyy/MM/dd y HH:mm:ss respectivamente. La clase Date se utiliza para obtener la fecha y hora actual. La fecha con formato se almacena en la variable **dateRun** y la hora con formato se almacena en la variable **startTimeRun**.

En la función **saveDataRun** se define una variable local llamada **id** y le asigna un valor que es la concatenación de los valores de **useremail**, **dateRun**, y **startTimeRun**. Esto significa que el valor de **id** será una cadena que contiene la combinación de estos tres valores. Por ejemplo, si useremail es "user@example.com", dateRun es "2022/12/25" y startTimeRun es "12:00:00", entonces id será "**user@example.com2022/12/2512:00:00**".

17.2.1 Problema al añadir datos con barras y puntos

¿Qué problema habría si añado "user@example.com2022/12/2512:00:00" en la base de datos Firestore?

En Firebase Firestore, las barras "/" en un nombre de colección o documento pueden causar problemas porque representan una ruta absoluta a una colección o documento específico. Si se agrega "user@example.com2022/12/2512:00:00" en la base de datos Firestore, podría haber conflictos con las rutas existentes si ya existen colecciones o documentos con nombres similares. Además, los caracteres ":" y "/" son caracteres especiales en Firestore y pueden causar problemas a la hora de consultar o realizar operaciones en documentos con esos nombres. Por lo tanto, es una buena práctica evitar utilizar estos caracteres en los nombres de las colecciones y documentos.

En este caso, las funciones `id.replace(":", "")` y `id.replace("/", "")` se utilizan para remover los caracteres ":" y "/" de la cadena id.

```
id = id.replace(":", "") // Eliminamos los puntos:  
id = id.replace("/", "") // Eliminamos la barra/
```

Después de definir id, este código llama a los métodos replace en dos ocasiones. El método replace es usado para reemplazar una subcadena de id con otra.

En la primera llamada, `id.replace(":", "")`, se reemplazan todas las apariciones de ":" en id con una cadena vacía. Esto significa que todos los ":" serán eliminados de id.

En la segunda llamada, `id.replace("/", "")`, se reemplazan todas las apariciones de "/" en id con una cadena vacía. Esto significa que todas las "/" serán eliminadas de id. Después de estas dos llamadas, id será una cadena que no contiene ":" ni "/". Utilizando el código anterior

¿cuál sería el resultado de "user@example.com2022/12/2512:00:00". ?

Si se aplica el código anterior a la cadena "user@example.com2022/12/2512:00:00", el resultado sería "**user@example.com20221225120000**". Este sería el valor final de la variable id después de ejecutar las dos llamadas a replace.

```

var collection = "runs$sportSelected"
var dbRun = FirebaseFirestore.getInstance()
dbRun.collection(collection).document(id).set(
    hashMapOf(
        "user" to useremail,
        "date" to dateRun,
        "startTime" to startTimeRun,
        "sport" to sportSelected,
        "activatedGPS" to activatedGPS,
        "duration" to saveDuration,
        "distance" to saveDistance.toDouble(),
        "avgSpeed" to saveAvgSpeed.toDouble(),
        "maxSpeed" to saveMaxSpeed.toDouble(),
        "minAltitude" to minAltitude,
        "maxAltitude" to maxAltitude,
        "minLatitude" to minLatitude,
        "maxLatitude" to maxLatitude,
        "minLongitude" to minLongitude,
        "maxLongitude" to maxLongitude,
        "centerLatitude" to centerLatitude,
        "centerLongitude" to centerLongitude,
    )
)

```

Para guardar los datos de la carrera en la base de datos. En primer lugar, se define una variable collection que almacena el nombre de la colección donde se guardarán los datos de la carrera. El nombre de la colección es una cadena que contiene el texto "runs" seguido del deporte seleccionado (**sportSelected**).

Luego, se obtiene una instancia de Firebase Firestore y se almacenan los datos de la carrera en un documento dentro de la colección especificada. El identificador del documento es id, que se definió previamente.

Los datos de la carrera se guardan en un **hashMapOf** que contiene las claves y valores.

17.3 Actualizar Datos. Función update

¿ Por qué utilizamos "update" ?

"update" es un método en Firebase Firestore que permite actualizar los valores de un documento específico dentro de una colección en la base de datos sin sobrescribir los valores previos. Sólo los valores que se especifican en la llamada a "update" se modifican. Es útil cuando se quiere mantener cierta información en el documento y solo actualizar ciertos campos específicos.

```
dbRun.collection(collection).document(id).update
```

Este método toma como argumento el nombre del campo y el valor que se desea actualizar, y actualiza el documento correspondiente en la colección. Si el campo específico no existe en el documento, entonces será creado con el valor especificado. Es importante destacar que, a diferencia de la función set, update no reemplazará todo el documento, sino que sólo actualizará los campos especificados.

```
//Si el modo intervalos está activado mandamos a actualizar campos , si no existen los crea
//Hacemos un update para añadir sin borrar lo anterior

if (swIntervalMode.isChecked) {
    dbRun.collection(collection).document(id).update( field: "intervalMode", value: true)
    dbRun.collection(collection).document(id)
        .update( field: "intervalDuration", npDurationInterval.value)
    dbRun.collection(collection).document(id)
        .update( field: "runningTime", tvRunningTime.text.toString())
    dbRun.collection(collection).document(id)
        .update( field: "walkingTime", tvWalkingTime.text.toString())
}
```

En este caso primero, se verifica si el modo intervalos está activado. Si está activado, se actualizan los campos en el documento correspondiente con el identificador (id) especificado. El método "update" se utiliza para añadir nuevos valores sin borrar los valores existentes. Por ejemplo, el valor "**intervalMode**" se establece en "true".

```
if (swChallenges.isChecked) {
    if (challengeDistance > 0f)
        dbRun.collection(collection).document(id).update(
            field: "challengeDistance",
            roundNumber(challengeDistance.toString(), decimals: 1).toDouble()
        )
    if (challengeDuration > 0)
        dbRun.collection(collection).document(id)
            .update( field: "challengeDuration", getFormattedStopWatch(challengeDuration.toLong()))
}
```

La misma lógica se utiliza para comprobar y guardar si había retos.

Este código verifica si la opción "Desafíos" está activada. Si está activada, verifica si la distancia y la duración del desafío son mayores a 0. Si es así, se actualiza el documento correspondiente en la colección especificada con los valores de la distancia y la duración del desafío. La función "roundNumber" redondea el número especificado a un decimal. La función "getFormattedStopWatch" devuelve la duración en un formato de tiempo legible. El método "update" se utiliza para añadir los valores sin sobrescribir los valores previos en el documento.

The screenshot shows the Google Cloud Firestore interface. On the left, the sidebar lists collections: levelsBike, levelsRollerSkate, levelsRunning, locations, runsBike, runsRollerSkate, runsRunning (which is selected), totalsBike, totalsRollerSkate, totalsRunning, and users. The main area shows a document in the 'runsRunning' collection with the key 'mundodigital.pro@gmail.com202302031'. The document contains the following fields and values:

- activatedGPS: true
- avgSpeed: 3.5
- centerLatitude: 37.8727167
- centerLongitude: -4.772024950000005
- date: "2023/02/03"
- distance: 0
- duration: "00:00:43"
- maxAltitude: 147.29998779296875
- maxLatitude: 37.8727541
- maxLongitude: -4.7719146
- maxSpeed: 10.6
- minAltitude: 147.39999389648438
- minLatitude: 37.8726793
- minLongitude: -4.7721353
- sport: "Running"
- startTime: "10:00:03"
- user: "mundodigital.pro@gmail.com"

Imagen de la bbdd con los datos de una actividad

17.4 Consultas a la Base de Datos

17.4.1 Función loadMedalsBike.

```
private fun loadMedalsBike() {
    var dbRecords = FirebaseFirestore.getInstance()

    //Calculamos el Top 3 de Distancia
    dbRecords.collection( collectionPath: "runsBike" ) CollectionReference
        .orderBy( field: "distance", Query.Direction.DESCENDING ) //Ordenamos de mayor a menor
        .get() Task<QuerySnapshot!>
        .addOnSuccessListener { documents ->
            for (document in documents) {
                if (document["user"] == useremail) // Compruebo el usuario
                    medalsListBikeDistance.add(
                        document["distance"].toString().toDouble()
                    ) // Añado el usuario al array
                if (medalsListBikeDistance.size == 3) break
            }
            while (medalsListBikeDistance.size < 3) medalsListBikeDistance.add(0.0) //Cuando sea menor de 3 añadimos 0.0
        }
        .addOnFailureListener { exception ->
            Log.w(ContentValues.TAG, msg: "Error getting documents: ", exception)
        }
}
```

Con esta función nos conectamos a la base de datos y recuperamos información sobre carreras en bicicleta.

La función "**loadMedalsBike**" recupera los registros de tres colecciones en Firestore: "runsBike", "runsBikeAvgSpeed" y "runsBikeMaxSpeed".

Para cada colección, la función ordena los registros en orden descendente por la distancia, igual que se hace otra consulta para la distancia se hace también para la velocidad promedio y la velocidad máxima, respectivamente. Luego, para cada documento en la colección, se comprueba

si el usuario es igual al usuario actual (especificado por la variable "useremail") y, en caso afirmativo, se agrega el valor correspondiente a la lista de medallas correspondiente.

Si la lista de medallas no contiene 3 elementos, se agrega 0.0 hasta que la lista contenga 3 elementos. Esto se hace con el código **"while (medalsListBikeDistance.size < 3) medalsListBikeDistance.add(0.0)"**.

En caso de error al obtener los documentos de Firestore, se registra un mensaje de error en el registro.

17.5 Creación de SubColecciones con Ubicaciones

```
private fun saveLocation(location: Location) {
    var dirName = dateRun + startTimeRun // Identificador de carrera Fecha y Hora de Inicio
    dirName = dirName.replace( oldValue: "/", newValue: "")
    dirName = dirName.replace( oldValue: ":", newValue: "")
    var docName = timeInSeconds.toString()
    while (docName.length < 4) docName = "0" + docName //Añadimos el formato de 00001
    var ms: Boolean
    ms = speed == maxSpeed && speed > 0
    var dbLocation = FirebaseFirestore.getInstance()
    //Dentro de location estaran todos los nombres de los usuarios y dentro de cada carpeta
    //Tendremos los identificadores de carrera que seran su fecha y hora de inicio
    //Dentro tendremos las localizaciones que estamos añadiendo
    dbLocation.collection( collectionPath: "locations/$useremail/$dirName" ).document(docName).set(
        hashMapOf(
            "time" to SimpleDateFormat( pattern: "HH:mm:ss" ).format(Date()),
            "latitude" to location.latitude,
            "longitude" to location.longitude,
            "altitude" to location.altitude,
            "hasAltitude" to location.hasAltitude(),
            "speedFromGoogle" to location.speed,
            "speedFromMe" to speed,
            "maxSpeed" to ms,
            "color" to tvChrono.currentTextColor
        )
    )
}
```

Vamos a guardar cada ubicación de la carrera para futuras funcionalidades, para ello vamos a guardar todos los puntos de la carrera que estamos localizando. Vamos a guardar la información en la base de datos, de cada usuario como un documento y cada carrera realizada por el usuario como una subcolección dentro del documento. Además, es importante tener los datos bien organizados para poder trabajar con ellos de manera eficiente.

La función **saveLocation** toma como entrada un objeto de tipo "**Location**".

La variable "**dirName**" es el identificador de la carrera, que se crea combinando la fecha y la hora de inicio de la carrera y eliminando caracteres especiales.

La variable "**docName**" es el nombre del documento que se utiliza para guardar la información de ubicación. Este nombre se crea a partir del tiempo en segundos y se ajusta para tener un formato de 00001.

La función luego utiliza la instancia de Firestore para acceder a la base de datos y agregar la información de ubicación a la colección correspondiente.

La ubicación se guarda como un mapa de clave-valor que contiene información sobre la hora, la latitud, la longitud, la altitud, la velocidad, entre otros datos.

The screenshot shows the MongoDB interface with two panels. The left panel displays a list of collections under the database 'letsdoitapp-33146', including 'levelsBike', 'levelsRollerSkate', 'levelsRunning', and 'locations'. The 'locations' collection is highlighted with a red box. The right panel shows the 'locations' collection details, with a sub-panel for a specific document identified by the email 'mundodigital.pro@gmail.com'. This document has fields 'zzz' and '20230203131214'. A red box highlights the email field. Below this, there's a 'Agregar campo' (Add field) button.

The screenshot shows the MongoDB interface with a path: 'locations > mundodigital.pr... > 20230203131214 > 0006'. The right panel shows the document '0006' with fields: 'altitude: 147.29998779296875', 'color: -2743028', 'hasAltitude: true', 'latitude: 37.8727525', 'longitude: -4.7720466', 'maxSpeed: true', 'speedFromGoogle: 0.04592176899313927', 'speedFromMe: 0.22943865531852026', and 'time: "13:12:20"'. A note at the bottom left states: 'Este documento no existe, por lo que no aparecerá en las consultas ni en las instantáneas' (This document does not exist, so it will not appear in queries or snapshots). A 'Más información' link is also present.

17.6 Borrado de Datos

El borrado de registros en una base de datos no es solo una tarea simple, sino que también requiere considerar todos los efectos colaterales que se derivan de la eliminación de ese registro. Hay que pensar en la coherencia de los datos y en los ajustes necesarios para mantener la integridad de los mismos. Esto incluye el borrado de todas las ubicaciones relacionadas con la carrera, la actualización de récords y totales, el borrado de fotos relacionadas con la carrera, y la revisión de los datos para asegurarse de que todo esté en línea con la eliminación de ese registro. Un programador debe tener en cuenta todos estos aspectos y ser consciente de los impactos de la eliminación de un registro en la base de datos.

17.6.1 Función deleteRun

```
private fun deleteRun(idRun: String, sport: String, ly: LinearLayout) { //Borramos la carrera
    var dbRun = FirebaseFirestore.getInstance()
    dbRun.collection( collectionPath: "runs$sport").document(idRun).delete()
        .addOnSuccessListener { it: Void!
            //ly es el layout donde quiero que se genere el snackbar
            Snackbar.make(ly, text: "Registro Borrado", Snackbar.LENGTH_LONG).setAction( text: "OK") { it: View!
                ly.setBackgroundColor(Color.CYAN)
            }.show()
        }
        .addOnFailureListener { it: Exception
            Snackbar.make(ly, text: "Error al Borrar Registro", Snackbar.LENGTH_LONG)
                .setAction( text: "OK") { it: View!
                    ly.setBackgroundColor(Color.CYAN)
                }.show()
        }
    }
}
```

La función toma como entrada el **ID** de la carrera a borrar, el deporte asociado con la carrera y un objeto **LinearLayout** que se utiliza como vista para mostrar un **Snackbar**.

La función primero obtiene una instancia de la base de datos en Firebase y luego usa la instancia para acceder a la colección de carreras del deporte específico. Luego, llama a **delete** en el documento con el ID de la carrera especificado.

Si la eliminación es exitosa, se muestra un **Snackbar** con un mensaje "**Registro Borrado**". Si falla, se muestra un mensaje "**Error al Borrar Registro**". En ambos casos, la acción del **Snackbar** cambia el color de fondo del objeto **LinearLayout** a cian.

17.7 Actualización de los Totales

El sistema lleva un registro de carreras y actualiza los totales y récords de cada carrera. Si se crea una nueva carrera, se actualizan los totales y los récords y si se borra una carrera, los totales y los récords deben ser revisados y volver a su estado anterior.

```
private fun updateTotals(cr: Runs) {
    totalsSelectedSport.totalDistance = totalsSelectedSport.totalDistance!! - cr.distance!!
    totalsSelectedSport.totalRuns = totalsSelectedSport.totalRuns!! - 1
    totalsSelectedSport.totalTime =
        totalsSelectedSport.totalTime!! - getSecFromWatch(cr.duration!!)
}
```

"**updateTotals**" que toma un objeto "cr" de tipo "Runs" como entrada. La función actualiza los totales almacenados en un objeto "**totalsSelectedSport**" restando la distancia, la cantidad de carreras y el tiempo de duración de la carrera específica "cr". La función utiliza la función "**getSecFromWatch**" para convertir la duración de la carrera a segundos.

17.8 Consulta Record de Distancia de un Usuario en particular

```

private fun checkDistanceRecord(cr: Runs, sport: String, user: String) {
    if (cr.distance!! == totalsSelectedSport.recordDistance) {
        var dbRecords = FirebaseFirestore.getInstance()
        dbRecords.collection(collectionPath: "runs$sport") CollectionReference
            .orderBy(field: "distance", Query.Direction.DESCENDING) Query
            .whereEqualTo(field: "user", user) // Los registros del usuario particular
            .get() Task<QuerySnapshot!
        .addOnSuccessListener { documents ->
            if (documents.size() == 1) totalsSelectedSport.recordDistance = 0.0
            else totalsSelectedSport.recordDistance =
                documents.documents[1].get("distance").toString().toDouble()

            var collection = "totals$sport"
            var dbUpdateTotals = FirebaseFirestore.getInstance()
            dbUpdateTotals.collection(collection).document(user)
                .update(field: "recordDistance", totalsSelectedSport.recordDistance)

            totalsChecked++
            if (totalsChecked == 3) refreshTotalsSport(sport)
        }
        .addOnFailureListener { exception ->
            Log.w(ContentValues.TAG, msg: "Error getting documents WHERE EQUAL TO: ", exception)
        }
    }
}

```

Toma como entrada un objeto "cr" de tipo "Runs", una cadena "sport" y una cadena "user". La función verifica si la distancia de la carrera "cr" es igual a la distancia del récord almacenada en "totalsSelectedSport".

Si es igual, se utiliza Firebase Firestore para buscar los registros de la carrera de ese deporte y usuario en particular en la base de datos y ordenarlos por distancia en orden descendente. Si solo hay un registro, se establece el récord en 0.0, de lo contrario, se establece en el segundo documento más largo.

Luego, se actualiza el récord en la base de datos y se aumenta un contador "totalsChecked". Si "totalsChecked" es igual a 3, se llama a la función "refreshTotalsSport" pasándole el deporte como entrada. En caso de un error durante la recuperación de los documentos, se registra en los registros de la aplicación.

.whereEqualTo("user", user) // Los registros del usuario particular

La línea "whereEqualTo("user", user)" especifica que se está filtrando solo los documentos que tienen el valor "user" igual al valor "user" pasado como entrada a la función. Esto significa que solo se recuperarán los registros de la carrera para un usuario particular.

17.8.1 Creación de Indices compuestos en Firestore

ID de la colección	Campos indexados	Alcance de la consulta	Estado
runsRunning	user Ascendente distance Descendente __name__ Descendente	Colección	Habilitado
runsRunning	user Ascendente avgSpeed Descendente __name__ Descendente	Colección	Habilitado
runsRunning	user Ascendente maxSpeed Descendente __name__ Descendente	Colección	Habilitado
runsBike	user Ascendente distance Descendente __name__ Descendente	Colección	Habilitado
runsBike	user Ascendente avgSpeed Descendente __name__ Descendente	Colección	Habilitado
runsBike	user Ascendente maxSpeed Descendente __name__ Descendente	Colección	Habilitado
runsRollerSkate	user Ascendente distance Descendente __name__ Descendente	Colección	Habilitado
runsRollerSkate	user Ascendente avgSpeed Descendente __name__ Descendente	Colección	Habilitado
runsRollerSkate	user Ascendente maxSpeed Descendente __name__ Descendente	Colección	Habilitado

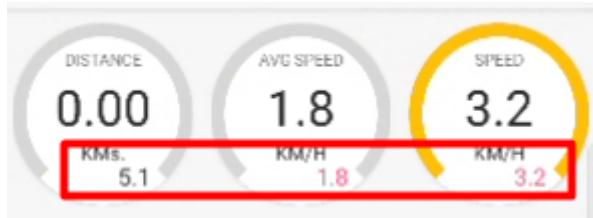
Los índices en Cloud Firestore se utilizan para mejorar el rendimiento de las consultas y para evitar problemas de escala cuando el volumen de datos es muy grande. Al crear índices, puedes definir cómo se deben organizar tus datos en la base de datos y cómo deben ser recuperados para cumplir con los requisitos de tus consultas. De esta manera, la base de datos puede acceder rápidamente a los datos necesarios para responder a tus consultas, lo que mejora el rendimiento de tu aplicación. Además, los índices también permiten realizar búsquedas complejas, filtrar los datos y ordenarlos, lo que puede ser útil para mostrar información relevante a tus usuarios.

17.9 Borrado de Localizaciones

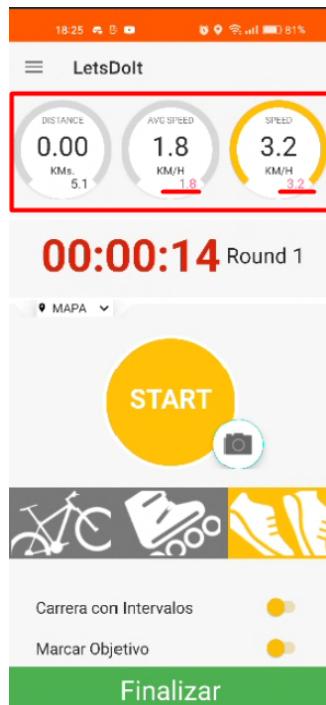
```
private fun deleteLocations(idRun: String, user: String) {
    var idLocations = idRun.subSequence(user.length, idRun.length).toString()
    var dbLocations = FirebaseFirestore.getInstance()
    dbLocations.collection(collectionPath: "locations/$user/$idLocations")
        .get()
        .addOnSuccessListener { documents ->
            for (docLocation in documents) {
                var dbLoc = FirebaseFirestore.getInstance()
                dbLoc.collection(collectionPath: "locations/$user/$idLocations").document(docLocation.id)
                    .delete()
            }
        }
        .addOnFailureListener { exception ->
            Log.w(ContentValues.TAG, msg: "Error getting documents: ", exception)
        }
}
```

La función "**deleteLocations**" elimina todos los documentos dentro de la colección "**locations/\$user/\$idLocations**". La variable "**idLocations**" es obtenida a partir del "**idRun**" recibido como parámetro y contiene una subsecuencia de caracteres que representan el identificador de la ubicación. La eliminación se realiza mediante el método "**delete()**" de **FirebaseFirestore** para cada documento obtenido en la consulta exitosa. En caso de falla, se registra un error en los logs.

18. Notificaciones



18.1 Notificación de Récord



Con la función **refreshInterfaceData** actualiza los datos mostrados en la interfaz de la aplicación. Actualiza las vistas de texto para la distancia, la velocidad promedio y la velocidad actual, así como los valores para varios seekbars circulares que se utilizan para representar estos valores. La función verifica si los nuevos valores para la distancia, la velocidad promedio y la velocidad actual son récords, y en ese caso, actualiza las vistas de texto y los seekbars correspondientes. La función también actualiza los límites para los seekbars basados en los nuevos valores.

18.2 Consultar y Mostrar Medallas. Función checkMedals

```
private fun checkMedals(distance: Double, averageSpeed: Double, maxSpeed: Double) {
    if (distance > 0) { //Preguntamos si Distancia es mayor que cero
        if (distance >= medalsListSportSelectedDistance.get(0)) { //Es mayor que el primer elemento ?
            recDistanceGold = true; recDistanceSilver = false; recDistanceBronze =
                false //Ha conseguido Oro
            notifyMedal(category: "distance", metal: "gold", scope: "PERSONAL") // Enviamos la notificación
        } else { // Si no es mayor que el siguiente
            if (distance >= medalsListSportSelectedDistance.get(1)) { //Es mayor que el segundo elemento ?
                recDistanceGold = false; recDistanceSilver = true; recDistanceBronze =
                    false // Ha conseguido Plata
                notifyMedal(category: "distance", metal: "silver", scope: "PERSONAL")
            } else {
                if (distance >= medalsListSportSelectedDistance.get(2)) { //Es mayor que el tercer elemento ?
                    recDistanceGold = false; recDistanceSilver = false; recDistanceBronze =
                        true // Medalla Bronce
                    notifyMedal(category: "distance", metal: "bronze", scope: "PERSONAL")
                }
            }
        }
    }
}
```

La función **checkMedals** califica el rendimiento de un usuario y le otorga medallas basadas en diferentes categorías (distancia, velocidad promedio y velocidad máxima). Se verifica si cada una de estas categorías tiene un valor mayor que cero y, si es así, se comparan con los límites de medallas previamente establecidos (oro, plata y bronce) y se notifica al usuario si ha ganado alguna medalla.

```
@SuppressLint("MissingPermission")
private fun notifyMedal(category: String, metal: String, scope: String) {
    val CHANNEL_NAME = "notifyMedal"
    val IMPORTANCE = NotificationManager.IMPORTANCE_HIGH
    var CHANNEL_ID = "NEW $scope RECORD - $sportSelected"

    var textNotification = ""
    when (metal) {
        "gold" -> textNotification = "1º "
        "silver" -> textNotification = "2º "
        "bronze" -> textNotification = "3º "
    }
    textNotification += "mejor marca personal en "
    when (category) {
        "distance" -> textNotification += "distancia recorrida"
        "avgSpeed" -> textNotification += "velocidad promedio"
        "maxSpeed" -> textNotification += "velocidad máxima alcanzada"
    }

    //Guardamos las medallas en una variable
    var iconNotificacion: Int = 0
    when (metal) {
        "gold" -> iconNotificacion = R.drawable.medalgold
        "silver" -> iconNotificacion = R.drawable.medalsilver
        "bronze" -> iconNotificacion = R.drawable.medalbronze
    }
}
```

La función "**notifyMedal**" es la responsable de mostrar al usuario la medalla ganada.

La notificación se muestra cuando se alcanza un récord en un deporte determinado, según la distancia recorrida, la velocidad promedio y la velocidad máxima alcanzada. La función recibe como parámetros la categoría, el metal y el alcance.

La función crea un canal de notificación si la versión de Android es mayor o igual a 8.0 (Oreo). Luego, crea una notificación con el título y el texto determinados por la categoría y el metal. Además, la función utiliza una imagen para representar el metal, ya sea oro, plata o bronce.

La función utiliza un switch para determinar el texto y la imagen de la notificación, y otro switch para determinar el ID de la notificación. Finalmente, se muestra la notificación con la función notify de la clase **NotificationManagerCompat**.

Documentación oficial Notificaciones:

<https://developer.android.com/training/notify-user/build-notification?hl=es-419>

19. Implementación de la Cámara

En esta sección se describe el proceso para implementar la funcionalidad de la cámara en nuestra app. Se va a hacer una integración personalizada, creando toda la conexión con la cámara desde cero, incluyendo reconocimiento de frontal y trasera, permisos, captura de previsualización, captura de imagen y guardado en la galería. Todo el código se desarrollará desde cero, incluyendo proporciones, conexión con la cámara, consulta de cámaras disponibles y muchas otras funciones.

19.1 Importación en Gradle de las librerías de Cámara

Siguiendo las instrucciones de la página oficial se importan las siguientes librerías:

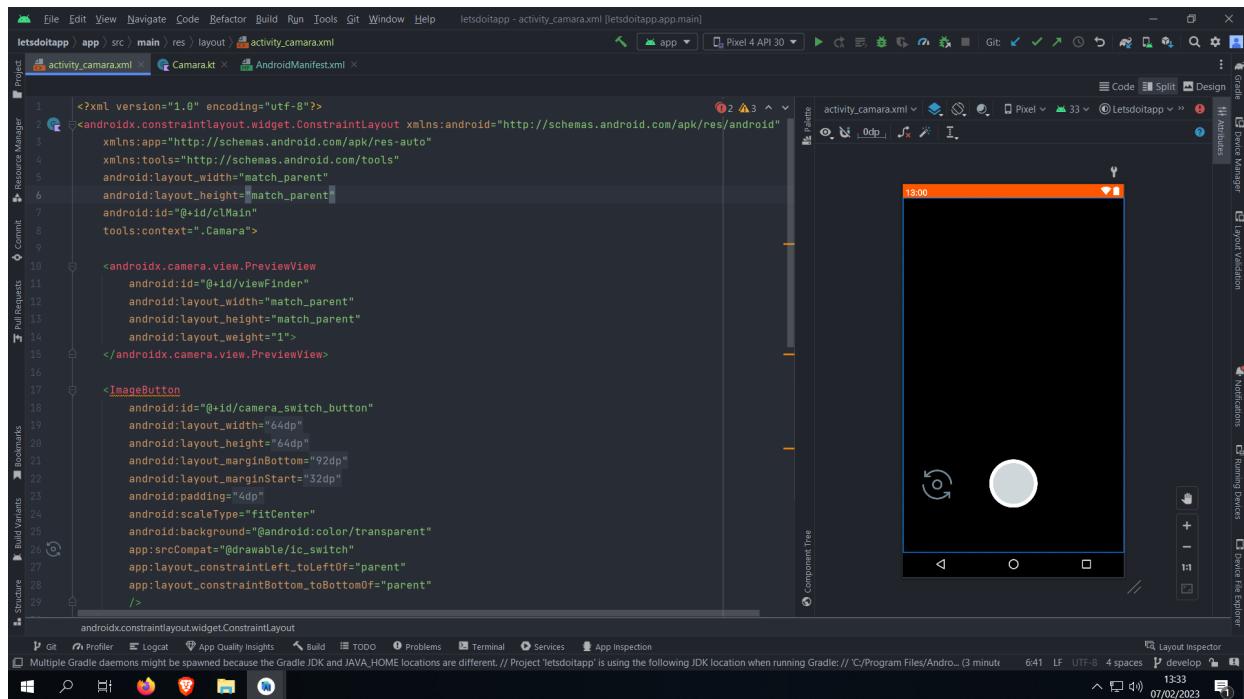
```
💡 //Camara
implementation 'androidx.camera:camera-view:1.0.0-alpha23'
implementation("androidx.camera:camera-core:1.0.1")
implementation("androidx.camera:camera-camera2:1.0.1")

// If you want to additionally use the CameraX Lifecycle library
implementation("androidx.camera:camera-lifecycle:1.0.1")
// If you want to additionally use the CameraX View class
implementation("androidx.camera:camera-view:1.0.0-alpha27")
// If you want to additionally use the CameraX Extensions library
implementation("androidx.camera:camera-extensions:1.0.0-alpha27")
```

Documentación Oficial Camara y Video:

<https://developer.android.com/training/camera/videobasics?hl=es-419>

19.2 Creación de la pantalla de la Cámara



Para el diseño de la pantalla de la cámara se crea una actividad nueva . Este es un archivo de diseño XML en Android usando ConstraintLayout. Define el diseño de una vista de cámara que consta de una vista previa de la cámara (**PreviewView**), tres botones (**ImageButton**) para cambiar de cámara, capturar y ver fotos, y restricciones para posicionar los botones en la parte inferior de la pantalla con márgenes específicos. El botón de vista de fotos tiene una visibilidad inicial de "gone", lo que significa que no es visible cuando se carga por primera vez el diseño.

19.3 Función TakePicture

```

new *
fun takePicture(v: View) { //Envia los datos a Camara
    val intent = Intent(packageContext: this, Camara::class.java)
    val inParameter = intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP)
    inParameter.putExtra(name: "dateRun", dateRun)
    inParameter.putExtra(name: "startTimeRun", startTimeRun)
    startActivityForResult(intent)
}

```

La función se activa cuando se hace clic en un objeto "View" "v" y toma una foto utilizando la cámara. Crea un objeto Intent que inicia la actividad "Cámara", agrega una bandera para borrar la actividad superior y agrega datos adicionales al intent en forma de las variables "dateRun" y "startTimeRun". Finalmente, la función inicia la actividad "Camara" utilizando el método "startActivity" y pasando el objeto Intent.

19.4 Ejecución de la Cámara en un hilo separado

`cameraExecutor = Executors.newSingleThreadExecutor()`

Este código crea un nuevo objeto de ejecutor de hilo único, que se utiliza para ejecutar una tarea de cámara en un hilo separado. La clase "**Executors**" es una clase de utilidad que proporciona métodos de fábrica para crear diferentes tipos de ejecutores. En este caso, se utiliza el método "**newSingleThreadExecutor**" para crear un ejecutor de hilo único, lo que significa que todas las tareas enviadas al ejecutor se ejecutan en un solo hilo de fondo.

El propósito de usar un hilo separado para la tarea de cámara es evitar que la interfaz de usuario se bloquee o se vuelva no receptiva mientras la cámara está en funcionamiento. Al ejecutar la tarea de cámara en un hilo separado, la aplicación puede seguir respondiendo a la entrada del usuario y actualizando la interfaz de usuario mientras la cámara se ejecuta en segundo plano.

19.5 Creación de directorios externos. `externalMediaDirs`

```
private fun getOutputDirectory(): File {
    val mediaDir = externalMediaDirs.firstOrNull()?.let {
        File(it, "letsdoit").apply { mkdirs() }
    }
    return if (mediaDir != null && mediaDir.exists()) mediaDir else
filesDir
}
```

Esta función devuelve un objeto "File" que representa un directorio de salida para almacenar archivos. La función comienza buscando un directorio a través de la propiedad "**externalMediaDirs**" que nos proporciona un array de objetos "File" que a su vez representa los directorios externos disponibles que podemos utilizar para almacenar archivos.

Estos directorios pueden ser tarjetas SD externas o almacenamiento en la nube, como Google Drive o Dropbox, dependiendo de la configuración y disponibilidad del dispositivo. También pueden estar protegidos por permisos de acceso que la aplicación debe solicitar antes de poder acceder a ellos.

El propósito de utilizar directorios externos para almacenar archivos de medios es permitir que la aplicación tenga más espacio para almacenar grandes archivos, como imágenes y videos, que pueden llenar rápidamente el espacio de almacenamiento interno. Al acceder a estos directorios externos, la aplicación puede aprovechar el espacio de almacenamiento adicional disponible en el dispositivo.

Se usa "**firstOrNull()**" para obtener el primer elemento de este arreglo o "**null**" si el arreglo está vacío.

Si se encuentra un directorio válido, la función crea un subdirectorio "**letsdoit**" dentro de él a través del método "**mkdirs()**". El método "**apply**" permite realizar varias operaciones en el objeto "File" antes de devolverlo.

Si no se encuentra un directorio válido, la función devuelve el directorio de archivos internos de la aplicación a través de la propiedad "**filesDir**".

En resumen, esta función proporciona una forma de acceder a un directorio de salida para almacenar archivos en la aplicación, primero buscando un directorio externo si está disponible y luego devolviendo el directorio interno si no está disponible.

19.6 Selector de la Cámara Delantera Trasera

```
private var lensFacing: Int = CameraSelector.LENS_FACING_BACK

//Gestionamos la Lente
binding.cameraSwitchButton.setOnClickListener {
    lensFacing = if (CameraSelector.LENS_FACING_FRONT == lensFacing) {
        CameraSelector.LENS_FACING_BACK
    } else {
        CameraSelector.LENS_FACING_FRONT
    }
    bindCamera()//vincular camara
}
```

Este código establece un botón para cambiar entre la cámara frontal y trasera en Android. El botón de cambio de cámara está definido en el archivo de diseño con el ID **cameraSwitchButton**. Cuando se hace clic en el botón, el código cambia el valor de la variable **lensFacing** a **CameraSelector.LENS_FACING_BACK** o **CameraSelector.LENS_FACING_FRONT**, dependiendo de su valor actual. Finalmente, se llama al método **bindCamera()** para vincular la cámara seleccionada.

19.7 Petición de Permisos de la Cámara

private val REQUIRED_PERMISSIONS = arrayOf(Manifest.permission.CAMERA): Este es un arreglo que contiene el permiso necesario para acceder a la cámara.

private val REQUEST_CODE_PERMISSIONS = 10: Este es un código de solicitud que se utiliza cuando se solicitan permisos al usuario.

private fun hasCameraPermissions(): Boolean: Esta función verifica si el permiso de cámara ha sido otorgado. Si todos los permisos en el arreglo **REQUIRED_PERMISSIONS** están otorgados, la función devuelve verdadero, de lo contrario, devuelve falso.

private fun requestCameraPermissions(): Esta función solicita permisos al usuario si los permisos de la cámara no están otorgados. Si ya se han otorgado los permisos, se llama a la función **startCamera()**.

override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>, grantResults: IntArray): Este método se ejecuta después de que el usuario haya proporcionado una respuesta a la solicitud de permisos. Si el código de solicitud coincide con **REQUEST_CODE_PERMISSIONS**, se verifica si se han otorgado los permisos necesarios mediante la llamada a la función **hasCameraPermissions()**. Si se han otorgado los permisos, se llama a la función **startCamera()**. De lo contrario, se muestra un mensaje de error al usuario.

19.8 Iniciar Cámara. startCamara

La función **startCamara** es la que inicia la cámara. Primero, obtiene una instancia del proveedor de procesamiento de cámara (**ProcessCameraProvider**), que proporciona la funcionalidad de la cámara a la aplicación. Luego, el código agrega un listener a la instancia de **cameraProviderFinnaly**, que ejecutará un **Runnable** cuando la cámara esté lista para usarse. En el **Runnable**, el código verifica si el dispositivo tiene una cámara trasera o frontal, y establece **lensFacing** de acuerdo a ello. Si no hay ni cámara trasera ni frontal, la función lanza una excepción. Finalmente, el código llama a dos funciones, **manageSwitchButton** y **bindCamera**, que tienen funcionalidades adicionales relacionadas con la cámara.

19.9 Vinculación con la Cámara. bindCamara

La función **bindCamara** realiza la configuración y vinculación de la cámara. Se establecen las configuraciones de visualización previa y captura de imagen, seleccionando la lente deseada y el modo de captura. También se realiza un desvinculamiento previo de cualquier otra vinculación previa y, finalmente, se vincula la cámara a la vida útil de la aplicación y se establece el proveedor de superficie para la vista previa.

La función **aspectRatio** determina el aspecto de una imagen en base a sus dimensiones (width y height). Primero, se calcula el "**previewRatio**" que es la relación de aspecto entre la mayor y la menor dimensión. Luego, se compara "**previewRatio**" con los valores estáticos **RATIO_4_3_VALUE** y **RATIO_16_9_VALUE** para determinar cuál se acerca más a "previewRatio". Finalmente, se devuelve **AspectRatio.RATIO_4_3** si "previewRatio" es más cercano a **RATIO_4_3_VALUE** o **AspectRatio.RATIO_16_9** en caso contrario.

19.10 Tomar Fotos y Guardar Imagenes. Función takePhoto

La función `takePhoto` toma una foto utilizando la cámara. El nombre del archivo de la foto se genera utilizando el nombre de la aplicación, el correo electrónico del usuario, la fecha de la ejecución y la hora de inicio de la ejecución. La metadata de la foto incluye la orientación de la foto, ya sea horizontal o vertical. La foto se guarda como un archivo .jpg y se agrega a la galería del dispositivo. Si hay un error durante el proceso de guardado, se muestra un snack bar con el mensaje "Error al guardar la imagen". La foto también se carga en un almacenamiento remoto.

20. Implementación del Historial de Actividades

En esta pantalla se va implementar el historial de actividades donde se va a cargar el registro de actividades generado por un usuario. Los datos de las actividades serán cargados en una lista y cada uno de los elementos mostrará datos de la actividad. Además se podrá borrar o reproducir.

20.1 Carga de Datos. Función loadRecyclerView

```
private fun loadRecyclerView(field: String, order: Query.Direction) {
    runsArrayList.clear() //Limpiamos el Array
    val dbRuns = FirebaseFirestore.getInstance()
    dbRuns.collection( collectionPath: "runs$sportSelected").orderBy(field, order) Query
        .whereEqualTo( field: "user", useremail)
        .get() Task<QuerySnapshot>
        .addOnSuccessListener { documents ->
            for (run in documents)
                runsArrayList.add(run.toObject(Runs::class.java))
            myAdapter.notifyDataSetChanged()
        }
        .addOnFailureListener { exception ->
            Log.w(ContentValues.TAG, msg: "Error getting documents WHERE EQUAL TO: ", exception)
        }
}
```

La función "`loadRecyclerView`" carga los datos de una colección de `FirebaseFirestore` con el nombre "**runs + deporte seleccionado**". Luego, ordena los documentos en la colección en función del campo especificado y solo muestra los documentos en los que el campo "**user**" sea igual al correo electrónico del usuario. Finalmente, los datos se agregaron a una lista y se notificó al adaptador para actualizar la vista. En caso de un error al obtener los documentos, se registra un mensaje de error en el registro.

20.2 Función onOptionsItemSelected

La función **onOptionsItemSelected** es llamada cuando un elemento en el menú de opciones es seleccionado. La función está implementando la lógica para dar una acción dependiendo del elemento seleccionado en el menú de opciones. Hay varios elementos en el menú, cada uno con un identificador único. Para cada elemento, se realiza una comparación para ver si es el elemento seleccionado y luego se llama a la función **loadRecyclerView** con un parámetro que indica qué campo para ordenar y en qué dirección.

Además, también se actualiza el título del elemento seleccionado para indicar si está en orden ascendente o descendente.

Por último, se devuelve **super.onOptionsItemSelected (item)** para que la función pueda ser manejada por la clase padre si es necesario.

20.3 Función onResume

La función **onResume** es un método en el ciclo de vida de una actividad en Android que se llama justo después de que la actividad ha vuelto a ser visible en la pantalla. Esto significa que la actividad ha sido resumida después de haber sido minimizada o ocultada por otra actividad.

En este caso, **onResume** es sobreescrito en la clase y llama al método **loadRecyclerView** con los parámetros "date" y `Query.Direction.DESCENDING`, lo que significa que el RecyclerView se cargará con los datos ordenados por fecha en orden descendente cuando la actividad vuelva a estar visible.

21. Almacenamiento en la nube. Firebase Storage

Los puntos más importantes de la implementación Storage en la app :

1. Cómo subir archivos a la nube, guardarlos en la carpeta y especificar metadatos de los archivos.
2. Cómo capturar y leer los archivos que se han guardado en la carpeta.
3. Cómo eliminar los archivos de la carpeta.
4. La importancia del uso del storage en una aplicación para guardar y manejar archivos.

Además, vamos a utilizar Storage para guardar imágenes de una carrera y además vamos a capturar, leer y eliminar los archivos de la carpeta. También destacar la importancia de especificar metadatos para los archivos que se guardan en la carpeta, ya que esto puede ayudar a identificarlos y organizarlos de manera más eficiente.

21.1 Integración de Storage

Hay que tener en cuenta diferentes aspectos para la implementación de Storage. Firebase ofrece la posibilidad de almacenar diferentes tipos de archivos, como imágenes, PDF, vídeos y documentos de texto. También se pueden organizar los archivos en carpetas. Antes hay que configurar las reglas de lectura y escritura. Además, es necesario dar acceso a internet a la aplicación e integrar algunas librerías propias del Firebase Storage.

21.2 Subir archivos a Storage

```
private fun uploadFile(image: File) {
    var dirName = dateRun + startTimeRun
    dirName = dirName.replace( oldValue: ":", newValue: "").replace( oldValue: "/", newValue: "")
    val fileName = "$dirName-$countPhotos"
    val storageReference = FirebaseStorage.getInstance().getReference( location: "images/$useremail/$dirName/$fileName")

    storageReference.putFile(Uri.fromFile(image)) UploadTask
        .addOnSuccessListener { it:UploadTask.TaskSnapshot!
            lastImage = "images/$useremail/$dirName/$fileName"
            countPhotos++
            val myFile = File(image.getAbsolutePath)
            myFile.delete() //Borraremos el fichero del telefono

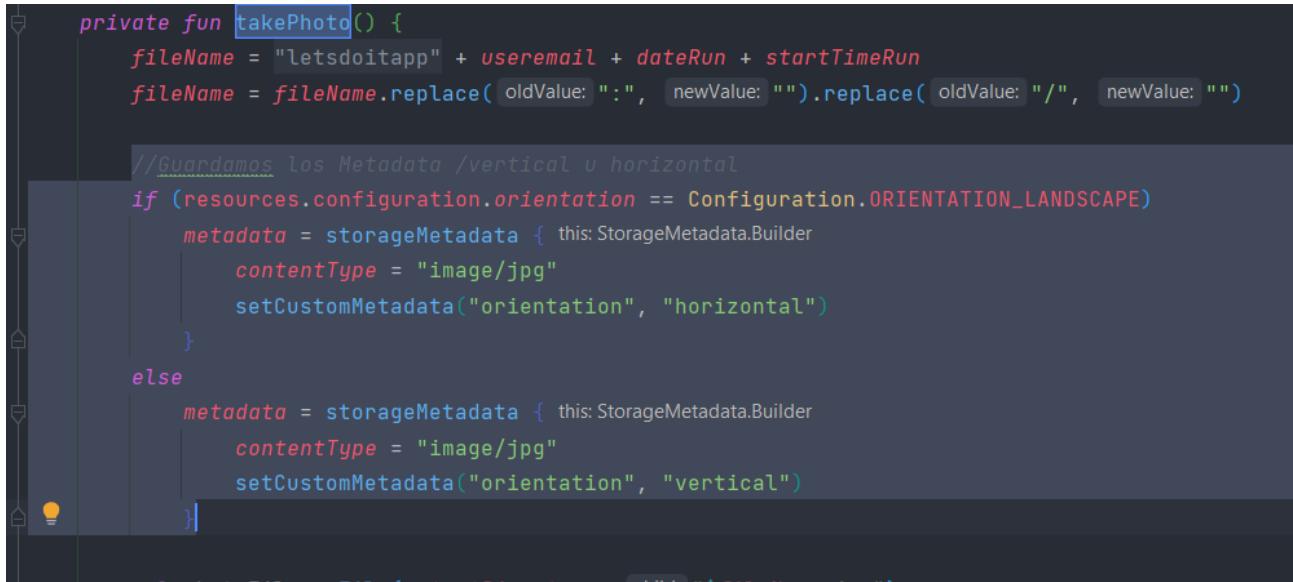
            //una vez que se ha subido el archivo le asignamos los metadatos
            val metaRef = FirebaseStorage.getInstance()
                .getReference( location: "images/$useremail/$dirName/$fileName")
            metaRef.updateMetadata(metadata)
                .addOnSuccessListener { it:StorageMetadata!
                    }
                .addOnFailureListener { it:Exception |
                    }
            }
        //var clMain = findViewById<ConstraintLayout>(R.id.clMain)
        Snackbar.make(binding.clMain, text: "Imagen Subida a la nube", Snackbar.LENGTH_LONG)
            .setAction( text: "OK" ) { it:View!
                binding.clMain.setBackgroundColor(Color.CYAN)
            }
}
```

Este es un código de Kotlin que sube un archivo de imagen a Firebase Storage. En resumen, lo que hace es:

1. Crear un nombre de directorio basado en la fecha y hora actual.
2. Crear un nombre de archivo único basado en el nombre de directorio y el número de imágenes que se han subido.
3. Crear una referencia a la ubicación donde se guardará el archivo en Firebase Storage.
4. Subir el archivo al servidor de Firebase Storage.
5. Una vez que se ha subido el archivo, se le asignan metadatos, como el nombre de archivo y la fecha de creación.

6. Mostrar un mensaje de éxito o de error dependiendo del resultado de la operación de subida del archivo. Si tiene éxito, se muestra un mensaje emergente indicando que la imagen se ha subido correctamente a la nube. Si falla, se muestra un mensaje emergente indicando que la imagen no se ha subido correctamente.

21.3 Crear Metadata Personalizado



```
private fun takePhoto() {
    fileName = "letsdoitapp" + useremail + dateRun + startTimeRun
    fileName = fileName.replace( oldValue: ":" , newValue: "" ).replace( oldValue: "/" , newValue: "" )

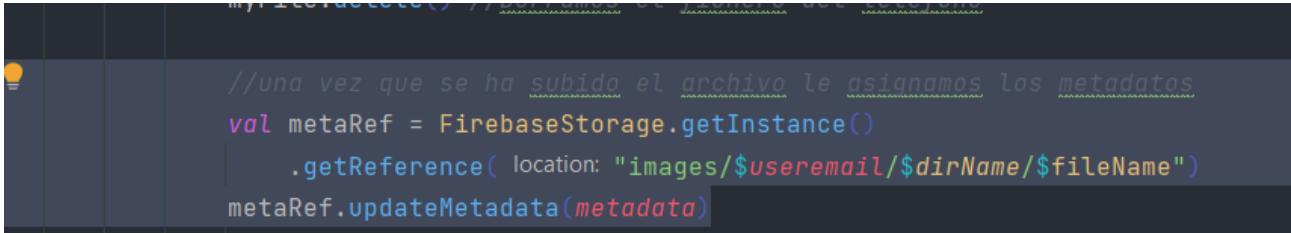
    //Guardamos los Metadata /vertical u horizontal
    if (resources.configuration.orientation == Configuration.ORIENTATION_LANDSCAPE)
        metadata = storageMetadata { this: StorageMetadata.Builder
            contentType = "image/jpg"
            setCustomMetadata("orientation", "horizontal")
        }
    else
        metadata = storageMetadata { this: StorageMetadata.Builder
            contentType = "image/jpg"
            setCustomMetadata("orientation", "vertical")
        }
}
```

Este código se utiliza para almacenar metadatos de un archivo de imagen en Firebase Storage. Los metadatos incluyen el tipo de contenido del archivo y si la imagen está en una orientación horizontal o vertical.

El código primero verifica la orientación del dispositivo usando la constante Configuration.ORIENTATION_LANDSCAPE. Si el dispositivo está en modo paisaje, lo que significa que el ancho de la pantalla es mayor que la altura, se establece los metadatos para indicar una orientación horizontal. De lo contrario, si el dispositivo está en modo retrato, se establece los metadatos para indicar una orientación vertical.

Los metadatos se crean usando el constructor storageMetadata, que le permite establecer varias propiedades de los metadatos, como el tipo de contenido y los metadatos personalizados. En este caso, el tipo de contenido se establece en "image/jpg" y se establece un par clave-valor de metadatos personalizados con la clave "orientación" y el valor "horizontal" o "vertical" dependiendo de la orientación del dispositivo.

Estos metadatos se pueden recuperar más adelante y utilizar para proporcionar contexto adicional sobre el archivo de imagen.



```
//una vez que se ha subido el archivo le asignamos los metadatos
val metaRef = FirebaseStorage.getInstance()
    .getReference( location: "images/${useremail}/${dirName}/${fileName}")
metaRef.updateMetadata(metadata)
```

Primero, se obtiene una referencia al archivo recién subido en Firebase Storage, especificando la ruta del archivo mediante la función `getReference()`. Esta ruta está compuesta por la cadena "images/" seguida del correo electrónico del usuario, el nombre del directorio y el nombre del archivo.

Luego, se utiliza la referencia a los metadatos recién creada, que se llama `metadata`, y se llama a la función `updateMetadata()` en la referencia a la imagen para actualizar sus metadatos.

La actualización de metadatos se lleva a cabo mediante los valores almacenados en la variable `metadatos`. Esta variable contiene información sobre el archivo que se subió, como el tipo de contenido y los metadatos personalizados, que se establecieron previamente en el código.

La actualización de los metadatos es útil porque proporciona información adicional sobre el archivo que se subió, lo que puede ser útil para realizar búsquedas o filtrar archivos.

21.4 Descargar archivos de Storage

```
//Código para cargar la imagen en el Historial y gestionar la foto
//Si hay foto cargamos la Imagen de Storage
if (run.lastimage != "") {
    val path = run.lastimage
    val storageRef = FirebaseStorage.getInstance().reference.child(path!!)
    val localfile = File.createTempFile(prefix: "tempImage", suffix: "jpg") //La ruta la convertimos en un archivo
    storageRef.getFile(localfile)//Capturamos el archivo
        .addOnSuccessListener { it: FileDownloadTask.TaskSnapshot! }
            val bitmap = BitmapFactory.decodeFile(localfile.getAbsolutePath)
            val metaRef = FirebaseStorage.getInstance().getReference(run.lastimage!!)
            val metadata: Task<StorageMetadata> = metaRef.metadata
            metadata.addOnSuccessListener { it: StorageMetadata! }
                val or = it.getCustomMetadata(key: "orientation")
                if (or == "horizontal") {
                    val porcent = 100 / bitmap.width.toFloat()
                    setHeightLinearLayout(holder.lyPicture, (bitmap.width * porcent).toInt())
                    holder.ivPicture.setImageBitmap(bitmap)
                } else {
                    val porcent = 100 / bitmap.height.toFloat()
                    setHeightLinearLayout(holder.lyPicture, (bitmap.width * porcent).toInt())
                    holder.ivPicture.setImageBitmap(bitmap)
                    holder.ivPicture.rotation = 90f //Rotamos la imagen
                }
            }
            metadata.addOnFailureListener { it: Exception
                Toast.makeText(context, text: "falló al cargar los metadatos", Toast.LENGTH_SHORT)
                    .show()
            }
        }
    .addOnFailureListener { it: Exception
        Toast.makeText(context, text: "falló al cargar la imagen", Toast.LENGTH_SHORT).show()
    }
}
```

Este código carga una imagen de Firebase Storage y la muestra en un ImageView en la interfaz de usuario.

Primero, se verifica si hay una imagen cargada en la variable run.lastimage. Si la variable contiene una cadena no vacía, se obtiene una referencia al archivo de imagen en Firebase Storage mediante la función `getReference()` y se crea un archivo temporal para almacenar la imagen descargada utilizando la función `createTempFile()`.

Luego, se llama a la función `getFile()` en la referencia del archivo de imagen para descargar la imagen de Firebase Storage en el archivo temporal. Si la descarga tiene éxito, se decodifica el archivo de imagen en un bitmap y se establece en el ImageView.

Además, se recuperan los metadatos de la imagen mediante la función `metadata`. Si se obtienen los metadatos correctamente, se verifica si la imagen está en orientación horizontal o vertical, utilizando la clave "orientation" en los metadatos personalizados. Si la imagen es horizontal, se ajusta la altura de la vista de imagen en función de la relación de aspecto de la imagen y se muestra la imagen. Si la imagen es vertical, se ajusta la altura de la vista de imagen, se rota la imagen 90 grados y se muestra la imagen.

Finalmente, se manejan los errores mediante el uso de funciones `addOnSuccessListener()` y `addOnFailureListener()` para proporcionar información de error y mensajes de usuario adecuados.

21.3 Borrado de archivos de Storage

```
💡 private fun deletePicturesRun(idRun: String) {
    val idFolder = idRun.subSequence(useremail.length, idRun.length).toString()
    // val delRef = FirebaseStorage.getInstance().getReference("images/$useremail/$idFolder")
    var delRef: StorageReference
    val storage = Firebase.storage
    val listRef = storage.reference.child(pathString: "images/$useremail/$idFolder")
    listRef.listAll()
        // .addOnSuccessListener { (items, prefixes) ->
        .addOnSuccessListener { listResult ->
            //items.forEach { item ->
            listResult.items.forEach { item ->
                val storageRef = storage.reference
                //val deleteRef = storageRef.child((item.path))
                delRef = storageRef.child(item.path)
                delRef.delete()
            }
        }
        .addOnFailureListener { it: Exception |
    }
```

Esta es una función en Kotlin que borra todas las imágenes de una carpeta específica en Firebase Storage.

El parámetro de entrada idRun es una cadena que representa el ID de una carrera y la función lo usa para determinar la carpeta de la que se deben eliminar las imágenes.

La función primero extrae el ID de la carpeta tomando una subcadena de idRun que comienza desde la longitud de useremail y termina en el final de idRun.

Luego crea un objeto StorageReference que apunta a la carpeta de la que se deben eliminar las imágenes, utilizando la instancia de Firebase Storage y el ID de la carpeta extraído. Se llama al método listAll() en este objeto para recuperar una lista de todas las imágenes en la carpeta.

La función luego recorre la lista de imágenes y crea un objeto StorageReference para cada imagen, utilizando la ruta de la imagen. Finalmente, llama al método delete() en cada objeto StorageReference para eliminar la imagen correspondiente.

Si ocurre un error durante el proceso de eliminación, la función no lo maneja y el fallo se ignora silenciosamente.

22. Reproducción de la actividad

En esta implementación se va integrar la opción de reproducir el recorrido que se ha hecho en una actividad. También se implementará la opción de compartir la carrera en una pantalla final con datos correspondientes como velocidades y posiciones.

22.1 Envío , recepción y reproducción de parámetros de Actividad

```
private fun callShareRun() {
    var tvDurationRun = findViewById<TextView>(R.id.tvDurationRun)
    var idRun = dateRun + startTimeRun
    idRun = idRun.replace( oldValue: ":", newValue: "")
    idRun = idRun.replace( oldValue: "/", newValue: "")

    var centerLatitude: Double = 0.0
    var centerLongitude: Double = 0.0

    if (activatedGPS == true) {
        centerLatitude = ((minLatitude!! + maxLatitude!!) / 2)
        centerLongitude = ((minLongitude!! + maxLongitude!!) / 2)
    }

    //var saveDuration = tvChrono.text.toString()
    var saveDuration = tvDurationRun.text.toString()
    var saveDistance = roundNumber(distance.toString(), decimals: 1)
    var saveMaxSpeed = roundNumber(maxSpeed.toString(), decimals: 1)
    var saveAvgSpeed = roundNumber(avgSpeed.toString(), decimals: 1)

    var medalDistance = "none"
    var medalAvgSpeed = "none"
    var medalMaxSpeed = "none"
```

La función shareRun se llama al compartir una carrera. La función prepara y envía los parámetros necesarios para iniciar la actividad RunActivity que hemos creado previamente.

Primero, la función genera un ID de carrera único concatenando la fecha y hora de inicio de la carrera y eliminando los caracteres ":" , "/". Luego, la función calcula la latitud y longitud central de la carrera a partir de la latitud y longitud mínimas y máximas, si se ha activado el GPS.

A continuación, la función redondea varios valores, incluyendo la duración de la carrera, la distancia recorrida, la velocidad máxima y la velocidad media. También determina qué medallas se han ganado en función de los tiempos de referencia.

Luego, la función crea un objeto Intent y agrega los parámetros necesarios a él. Los parámetros incluyen información sobre el usuario, el ID de la carrera, la ubicación central, las estadísticas de la carrera, la medalla ganada, el estado del GPS, el deporte seleccionado y otros detalles relevantes.

Finalmente, la función inicia la actividad RunActivity con el objeto Intent y los parámetros adjuntos. En resumen, la función recopila información sobre la carrera y los resultados, y los envía a la actividad RunActivity para su visualización y procesamiento.

```
private fun createMapFragment(){
    val mapFragment = supportFragmentManager
        .findFragmentById(R.id.map) as SupportMapFragment?
    mapFragment?.getMapAsync(callback: this)
}
```

La función llamada `createMapFragment()` se encarga de crear y configurar un fragmento de mapa en la actividad actual.

Primero, la función utiliza `supportFragmentManager` para buscar el fragmento de mapa que tiene el ID "map" en el archivo de diseño de la actividad. Luego, la función utiliza el método `getMapAsync()` en el objeto `mapFragment` para obtener una instancia del objeto `GoogleMap`.

La función espera que esta instancia del objeto `GoogleMap` se llame a través del método de devolución de llamada `onMapReady()` que se define en la actividad actual y que implementa la interfaz `OnMapReadyCallback`.

En resumen, la función crea y configura un fragmento de mapa en la actividad actual, que se utiliza para mostrar un mapa interactivo a los usuarios.

```
override fun onMapReady(googleMap: GoogleMap) {
    map = googleMap
    googleMap.mapType = GoogleMap.MAP_TYPE_HYBRID
    map.animateCamera(
        CameraUpdateFactory.newLatLngZoom(
            LatLng(centerLat!!, centerLong!!),
            zoom: 16f, durationMs: 1000, callback: null)
    loadLocations()
}
```

Esta es la función de devolución de llamada en Kotlin que se llama cuando el objeto GoogleMap está listo para ser utilizado en la actividad. Esta función se implementa en la actividad actual y se activa a través del método `getMapAsync()`.

La función de devolución de llamada tiene un parámetro `googleMap` que representa el objeto GoogleMap que se ha inicializado.

En esta función, el objeto `map` (que es un objeto GoogleMap) se inicializa con el objeto `googleMap`.

La función establece el tipo de mapa en "híbrido" utilizando la constante `GoogleMap.MAP_TYPE_HYBRID`. Luego, la función anima la cámara del mapa para que se centre en una ubicación específica, utilizando la latitud y longitud central de la carrera.

Finalmente, la función llama a la función `loadLocations()`, que se utiliza para cargar los marcadores en el mapa.

En resumen, esta función de devolución de llamada se llama cuando el objeto GoogleMap está listo para ser utilizado en la actividad y se utiliza para configurar el mapa y cargar los marcadores en él.

```
private fun loadLocations(){
    val collection = "locations/$user/$idRun"
    var point: LatLng
    val listPoints: Iterable<LatLng>
    listPoints = arrayListOf()
    listPoints.clear()
    val dbLocations: FirebaseFirestore = FirebaseFirestore.getInstance()
    dbLocations.collection(collection).CollectionReference
        .orderBy( field: "time")//Ordenamos las ubicaciones por el orden que se produjeron
        .get() Task<QuerySnapshot>{
            .addOnSuccessListener { documents ->
                for (docLocation in documents) {
                    var position = docLocation.toObject(Location::class.java)
                    //listPosition.add(position!!)
                    point = LatLng(position?.latitude!!, position?.longitude!!)
                    listPoints.add(point)
                }
                paintRun(listPoints)
            }
            .addOnFailureListener { exception ->
                Log.w(ContentValues.TAG, msg: "Error getting locations: ", exception)
            }
        }
}
```

La función `loadLocations()` se utiliza para cargar las ubicaciones de una carrera específica del usuario a partir de la base de datos Firebase Firestore.

La función construye una ruta de colección para la carrera del usuario utilizando el correo electrónico del usuario y el ID de la carrera. Luego, la función declara una lista mutable de puntos de ubicación llamada listPoints.

La función utiliza el objeto dbLocations de la clase FirebaseFirestore para obtener los documentos de ubicación en la colección. La función ordena los documentos por su tiempo de creación (para que se muestren en orden en el mapa) y los recorre uno por uno.

Para cada documento, la función extrae el objeto Location y crea un objeto LatLng con la latitud y longitud almacenadas en el objeto Location. La función agrega cada objeto LatLng a la lista listPoints.

Finalmente, la función llama a la función paintRun() y pasa la lista de puntos de ubicación como un parámetro. La función paintRun() se utiliza para trazar la ruta de la carrera en el mapa utilizando los puntos de ubicación.

En resumen, la función loadLocations() carga las ubicaciones de una carrera específica del usuario a partir de la base de datos Firebase Firestore y crea una lista de puntos de ubicación que se utilizarán para dibujar la ruta de la carrera en el mapa.

```
data class Location (
    var hasAltitude: Boolean? = null,
    var altitude: Double? = null,
    var latitude: Double? = null,
    var longitude: Double? = null,
    var color: Int? = null,
    var maxSpeed: Boolean? = null,
    var speedFromMe: Double? = null,
    var speedFromLocation: Double? = null,
    var time: String? = null
)
```

Este es un ejemplo de una clase de datos llamada "Location" que puede ser usada para almacenar información relacionada a la ubicación de un dispositivo. La clase contiene varias propiedades, incluyendo la altitud, la latitud, la longitud, el color, la velocidad máxima y la hora. Cada propiedad puede ser null (nula) en caso de no estar disponible.

```
private fun loadData(){
    val bundle = intent.extras //RECEPCION DE PARAMETROS
    user = bundle?.getString( key: "user")
    idRun = bundle?.getString( key: "idRun")
    centerLat = bundle?.getDouble( key: "centerLatitude")
    centerLong = bundle?.getDouble( key: "centerLongitude")

    //Si no tiene datos se cierra el Layout
    if (bundle?.getDouble( key: "distanceTarget") == 0.0){
        val lyCurrentLevel = findViewById<LinearLayout>(R.id.lyCurrentLevel)
        setHeightLinearLayout(lyCurrentLevel, value: 0)

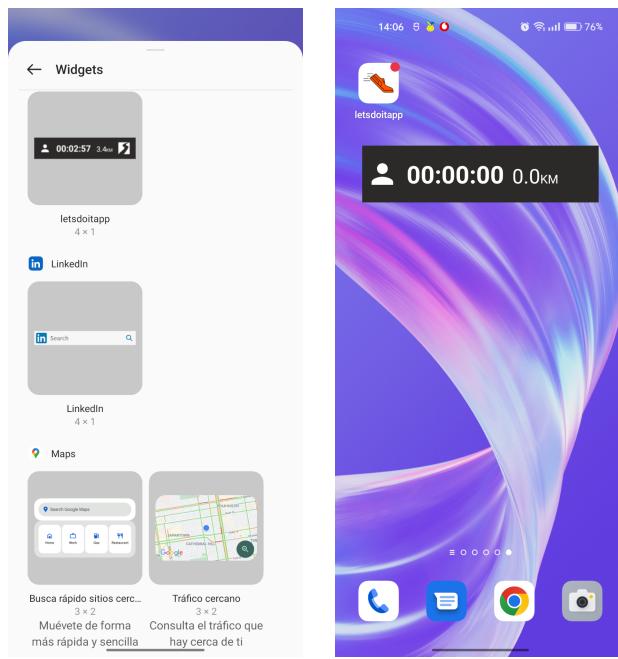
    } else{ // En caso contrario Tiene datos le damos valores

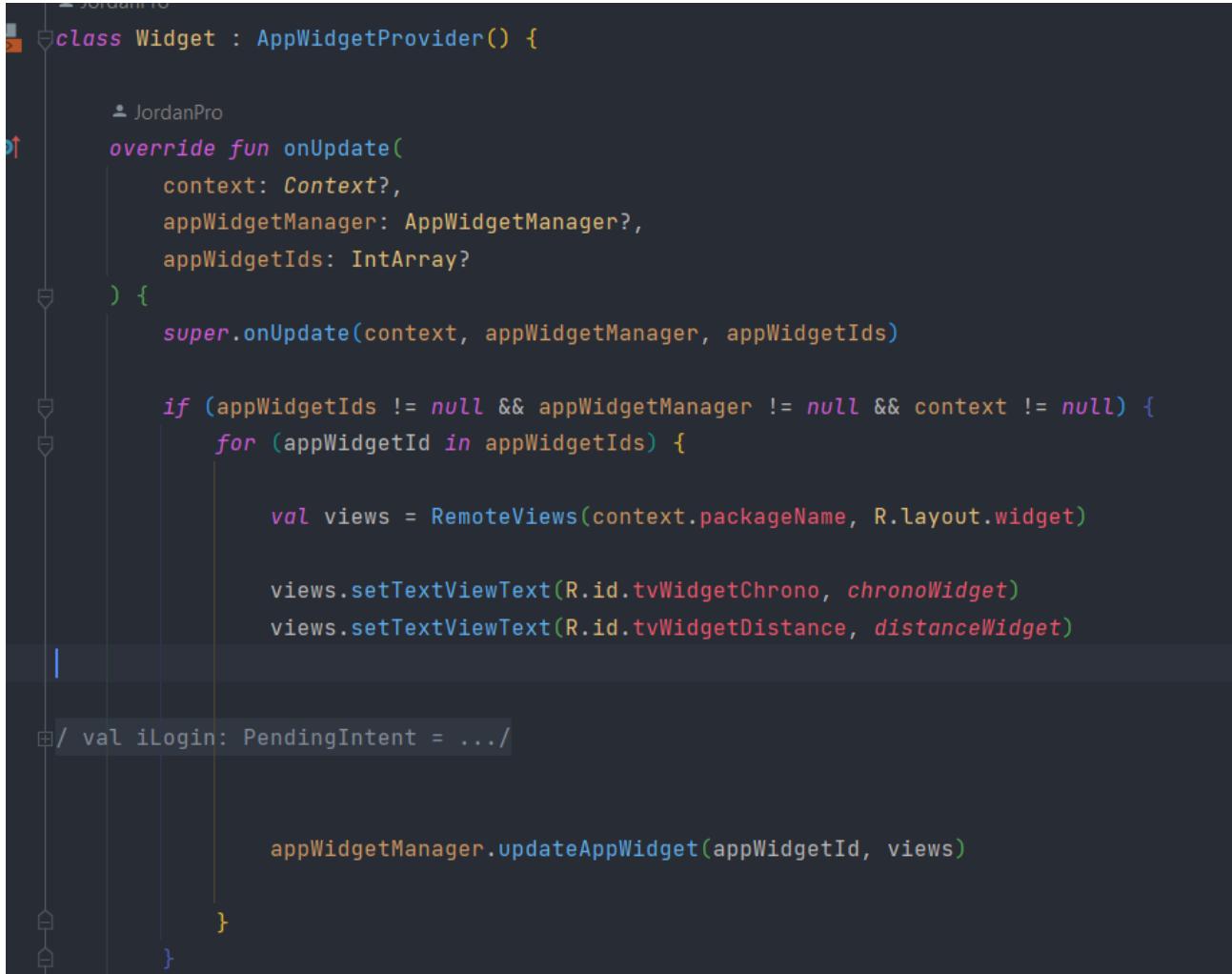
        //NIVEL
        val levelText = "${"Nivel"} ${bundle?.getString( key: "image_level")!!..subSequence(6,7).toString()}"
        val tvNumberLevel = findViewById<TextView>(R.id.tvNumberLevel)
        tvNumberLevel.text = levelText
        val ivCurrentLevel = findViewById<ImageView>(R.id.ivCurrentLevel)
        when (bundle.getString( key: "image_level")){
            "level_1" -> ivCurrentLevel.setImageResource(R.drawable.level_1)
            "level_2" -> ivCurrentLevel.setImageResource(R.drawable.level_2)
            "level_3" -> ivCurrentLevel.setImageResource(R.drawable.level_3)
        }
    }
}
```

La función `loadData()` se encarga de cargar los datos y mostrarlos en la interfaz de usuario. Primero, recibe los parámetros enviados por otra actividad mediante la función `"intent.extras"`, que se almacenan en variables locales. Luego, se utilizan estas variables para mostrar información al usuario en la pantalla. Si el usuario no tiene datos de distancia de objetivo, se oculta el `LinearLayout` que contiene la información correspondiente. Si el usuario tiene datos, se establecen los valores para cada campo de la vista de la aplicación.

El código verifica si hay imágenes asociadas al recorrido y las carga si es necesario. A continuación, establece la imagen del deporte seleccionado por el usuario. Si el GPS está desactivado, se oculta la sección que muestra los datos de medición. De lo contrario, los datos de medición se muestran en la interfaz de usuario. Finalmente, el código actualiza la altura de los elementos de `LinearLayout` para mostrar u ocultar la información correspondiente.

23. Widget





```

class Widget : AppWidgetProvider() {

    override fun onUpdate(
        context: Context?,
        appWidgetManager: AppWidgetManager?,
        appWidgetIds: IntArray?
    ) {
        super.onUpdate(context, appWidgetManager, appWidgetIds)

        if (appWidgetIds != null && appWidgetManager != null && context != null) {
            for (appWidgetId in appWidgetIds) {

                val views = RemoteViews(context.packageName, R.layout.widget)

                views.setTextViewText(R.id.tvWidgetChrono, chronoWidget)
                views.setTextViewText(R.id.tvWidgetDistance, distanceWidget)

                // val iLogin: PendingIntent = ...

                appWidgetManager.updateAppWidget(appWidgetId, views)
            }
        }
    }
}

```

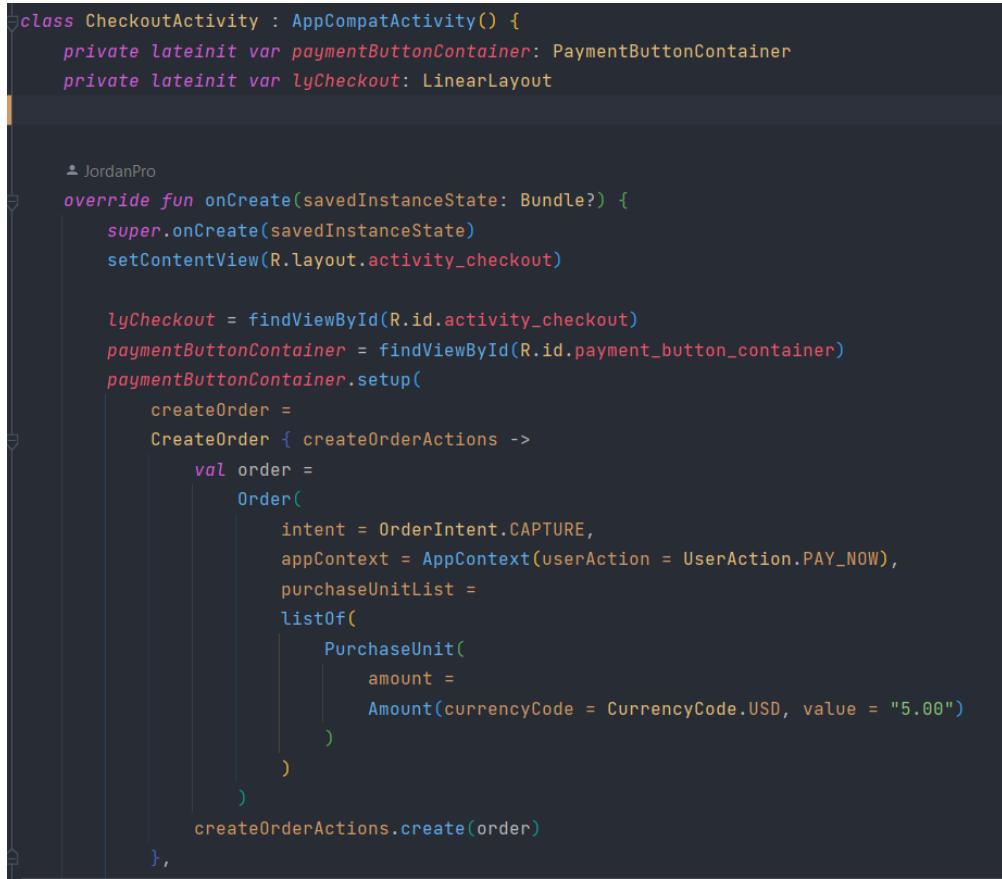
Este es un archivo de código Kotlin para una clase **AppWidgetProvider** de Android llamada "Widget". Un **AppWidgetProvider** es un tipo especial de **BroadcastReceiver** que se utiliza para manejar actualizaciones de widgets de aplicaciones.

La función "**onUpdate**" es el punto de entrada para manejar las actualizaciones del widget. Recibe un "**Contexto**", un "**AppWidgetManager**" y una matriz de "**appWidgetIds**". La función recorre todos los IDs en la matriz y actualiza las vistas para cada widget utilizando un objeto "**RemoteViews**".

En este código, el objeto "**RemoteViews**" se crea con el archivo de diseño "**R.layout.widget**". Dos **TextView** en el diseño se actualizan con los valores de "**chronoWidget**" y "**distanceWidget**", respectivamente.

El código dentro del código comentado es un ejemplo de cómo agregar eventos de clic a las vistas en el widget. Los objetos "**PendingIntent**" creados por "Intent" se utilizan para iniciar "**LoginActivity**" y "**MainActivity**" cuando se hacen clic en las vistas correspondientes. Sin embargo, en la implementación actual, estos eventos de clic no se agregan al widget.

24. Pasarela de Pagos de Paypal



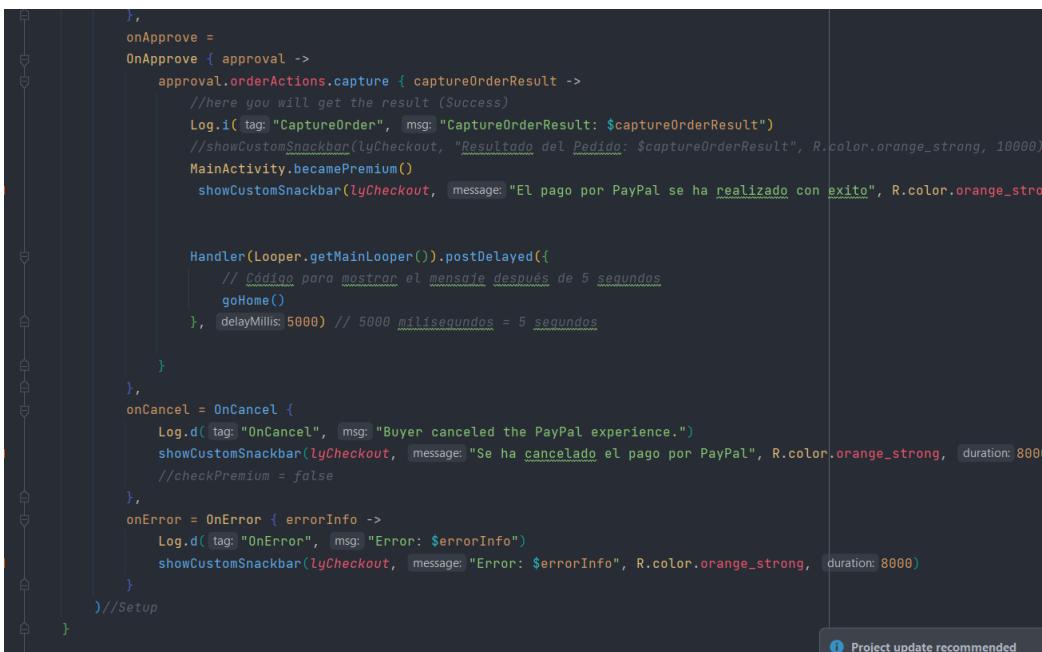
```

class CheckoutActivity : AppCompatActivity() {
    private lateinit var paymentButtonContainer: PaymentButtonContainer
    private lateinit var lyCheckout: LinearLayout

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_checkout)

        lyCheckout = findViewById(R.id.activity_checkout)
        paymentButtonContainer = findViewById(R.id.payment_button_container)
        paymentButtonContainer.setup(
            createOrder =
            CreateOrder { createOrderActions ->
                val order =
                    Order(
                        intent = OrderIntent.CAPTURE,
                        applicationContext = ApplicationContext(userAction = UserAction.PAY_NOW),
                        purchaseUnitList =
                        listOf(
                            PurchaseUnit(
                                amount =
                                Amount(currencyCode = CurrencyCode.USD, value = "5.00")
                            )
                        )
                )
                createOrderActions.create(order)
            },
            onApprove =
            OnApprove { approval ->
                approval.orderActions.capture { captureOrderResult ->
                    //here you will get the result (Success)
                    Log.i( tag: "CaptureOrder", msg: "CaptureOrderResult: $captureOrderResult")
                    //showCustomSnackbar(lyCheckout, "Resultado del Pedido: $captureOrderResult", R.color.orange_strong, 10000)
                    MainActivity.becamePremium()
                    showCustomSnackbar(lyCheckout, message: "El pago por PayPal se ha realizado con éxito", R.color.orange_strong)
                }
            },
            onCancel =
            OnCancel {
                Log.d( tag: "OnCancel", msg: "Buyer canceled the PayPal experience.")
                showCustomSnackbar(lyCheckout, message: "Se ha cancelado el pago por PayPal", R.color.orange_strong, duration: 8000)
                //checkPremium = false
            },
            onError =
            OnError { errorInfo ->
                Log.d( tag: "OnError", msg: "Error: $errorInfo")
                showCustomSnackbar(lyCheckout, message: "Error: $errorInfo", R.color.orange_strong, duration: 8000)
            }
        )//Setup
    }
}

```



```

    },
    onApprove =
    OnApprove { approval ->
        approval.orderActions.capture { captureOrderResult ->
            //here you will get the result (Success)
            Log.i( tag: "CaptureOrder", msg: "CaptureOrderResult: $captureOrderResult")
            //showCustomSnackbar(lyCheckout, "Resultado del Pedido: $captureOrderResult", R.color.orange_strong, 10000)
            MainActivity.becamePremium()
            showCustomSnackbar(lyCheckout, message: "El pago por PayPal se ha realizado con éxito", R.color.orange_strong)

            Handler(Looper.getMainLooper()).postDelayed({
                // Código para mostrar el mensaje después de 5 segundos
                goHome()
            }, delayMillis: 5000) // 5000 milisegundos = 5 segundos
        }
    },
    onCancel =
    OnCancel {
        Log.d( tag: "OnCancel", msg: "Buyer canceled the PayPal experience.")
        showCustomSnackbar(lyCheckout, message: "Se ha cancelado el pago por PayPal", R.color.orange_strong, duration: 8000)
        //checkPremium = false
    },
    onError =
    OnError { errorInfo ->
        Log.d( tag: "OnError", msg: "Error: $errorInfo")
        showCustomSnackbar(lyCheckout, message: "Error: $errorInfo", R.color.orange_strong, duration: 8000)
    }
} //Setup
}

```

La implementación del proceso de pago mediante PayPal utilizando la biblioteca de PayPal Checkout. Consiste en la creación de la actividad **CheckoutActivity**, que extiende

la clase `AppCompatActivity`, se implementa el método `onCreate()` para inflar el diseño de la actividad y configurar el botón de pago de PayPal. La actividad crea una referencia al objeto `PaymentButtonContainer` y al objeto `LinearLayout` en el que se mostrará el botón de pago, y utiliza el método `setup()` del objeto `PaymentButtonContainer` para configurar el botón de pago y crear un pedido de PayPal mediante la clase `CreateOrder`. La actividad también define manejadores de eventos para los casos en que el usuario aprueba el pago, cancela el pago o se produce algún error. Si el pago es aprobado, el manejador de eventos llama al método `capture()` del objeto `OrderActions` para capturar el resultado del pedido, actualizar el estado del usuario y mostrar un mensaje en la pantalla. Si hay un error o el usuario cancela el pago, se muestra un mensaje de error o cancelación en la pantalla.

En resumen, la implementación de pago por PayPal utiliza la biblioteca de PayPal Checkout para procesar un pago, y la actividad `CheckoutActivity` implementa los métodos necesarios para crear un pedido de PayPal, capturar el resultado del pedido y actualizar el estado del usuario.

25. Snackbar Personalizado

```
private fun showCustomSnackbar(view: View, message: String, backgroundColor: Int, duration: Int) {
    val snack = Snackbar.make(view, "", Snackbar.LENGTH_LONG)
    val customView = LayoutInflater.from(view.context)
        .inflate(R.layout.custom_snackbar, view.rootView as ViewGroup, false)

    // Cambiar el color de fondo
    snack.view.setBackgroundColor(ContextCompat.getColor(view.context, backgroundColor))

    // Cambiar el texto del Snackbar
    val tvMessage = customView.findViewById<TextView>(R.id.tvMessage)
    tvMessage.text = message

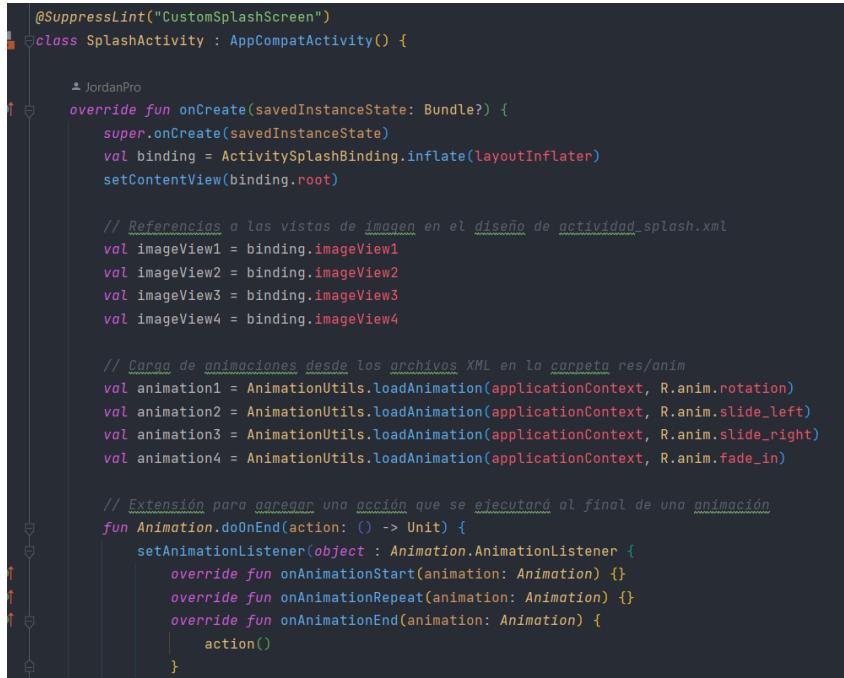
    // Cambiar la fuente del Snackbar
    tvMessage.typeface = Typeface.create("sans-serif-light", Typeface.BOLD)

    // Agregar la vista personalizada a la Snackbar
    val snackbarLayout = snack.view as Snackbar.SnackbarLayout
    snackbarLayout.addView(customView, 0)
```

```
// Agregar un efecto de animación
val animation = AnimationUtils.loadAnimation(view.context,
R.anim.slide_up)
snack.view.startAnimation(animation)
snack.duration = duration
snack.show()
}
```

El código anterior define una función llamada **showCustomSnackbar** que toma como entrada una vista, un mensaje de texto, un color de fondo y una duración. La función crea una instancia de **Snackbar** utilizando la vista de entrada y luego infla una vista personalizada a partir de un archivo de diseño XML. Luego, se actualizan los atributos de la vista personalizada con el mensaje y el color de fondo proporcionados, se establece la duración de **Snackbar** y se muestra la **Snackbar**. El resultado final es una **Snackbar** personalizada con un mensaje y un color de fondo definidos por el usuario, y una duración que se puede establecer de forma dinámica.

26. Intro y Animaciones en SplashScreen



```
@SuppressLint("CustomSplashScreen")
class SplashActivity : AppCompatActivity() {

    @JordanPro
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding = ActivitySplashBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Referencias a las vistas de imagen en el diseño de actividad_splash.xml
        val imageView1 = binding.imageView1
        val imageView2 = binding.imageView2
        val imageView3 = binding.imageView3
        val imageView4 = binding.imageView4

        // Carga de animaciones desde los archivos XML en la carpeta res/anim
        val animation1 = AnimationUtils.loadAnimation(applicationContext, R.anim.rotation)
        val animation2 = AnimationUtils.loadAnimation(applicationContext, R.anim.slide_left)
        val animation3 = AnimationUtils.loadAnimation(applicationContext, R.anim.slide_right)
        val animation4 = AnimationUtils.loadAnimation(applicationContext, R.anim.fade_in)

        // Extensión para agregar una acción que se ejecutará al final de una animación
        fun Animation.doOnEnd(action: () -> Unit) {
            setAnimationListener(object : Animation.AnimationListener {
                override fun onAnimationStart(animation: Animation) {}
                override fun onAnimationRepeat(animation: Animation) {}
                override fun onAnimationEnd(animation: Animation) {
                    action()
                }
            })
        }
    }
}
```

Un splash screen es una pantalla de inicio que se muestra mientras se carga la aplicación o antes de mostrar la pantalla principal. En este código, se utiliza la biblioteca View Binding para vincular las vistas del archivo de diseño "**activity_splash.xml**" a la actividad. Se definen cuatro vistas de imagen y se cargan animaciones desde archivos XML en la carpeta **res/anim**.

Se define una función de extensión "**doOnEnd**" que agrega una acción que se ejecutará al final de una animación. Las acciones se asignan a cada animación para mostrar y ocultar las vistas de imagen en secuencia.

La primera animación se inicia en "**imageView1**". Después de que finalice, se muestra "**imageView2**" y se inicia una segunda animación. Este proceso se repite para "**imageView3**" y "**imageView4**".

Se define un tiempo de espera de 4 segundos antes de lanzar la siguiente actividad. Se crea un objeto Handler con el looper principal y se ejecuta una tarea para iniciar la actividad "LoginActivity" después del tiempo de espera especificado.

```
// Asignación de acciones a ejecutar al final de cada animación
animation1.setOnEnd {
    imageView1.setVisibility(View.GONE)
    imageView2.setVisibility(View.VISIBLE)
    imageView2.startAnimation(animation2)
}

animation2.setOnEnd {
    imageView2.setVisibility(View.GONE)
    imageView3.setVisibility(View.VISIBLE)
    imageView3.startAnimation(animation3)
}

animation3.setOnEnd {
    imageView3.setVisibility(View.GONE)
    imageView4.setVisibility(View.VISIBLE)
    imageView4.startAnimation(animation4)
}

// Inicio de la primera animación
imageView1.startAnimation(animation1)

// Tiempo de espera antes de lanzar la siguiente actividad
val SPLASH_TIME_OUT = 4000L // Tiempo de espera en milisegundos

// Creación de un objeto Handler con el looper principal y ejecución de la tarea después de
Handler(Looper.getMainLooper()).postDelayed({
    val i = Intent(packageContext: this@SplashActivity, LoginActivity::class.java)
    startActivity(i)
    finish()
}, SPLASH_TIME_OUT)
```

27. Referencias

Documentación oficial Notificaciones:

<https://developer.android.com/training/notify-user/build-notification?hl=es-419>

Documentación oficial de Widget:

<https://developer.android.com/guide/topics/appwidgets?hl=es-419>

Inyección de dependencias con Hilt

<https://developer.android.com/training/dependency-injection/hilt-android?hl=es-419&authuser=1>

<https://www.youtube.com/watch?v=RFVnCCTY-j0>