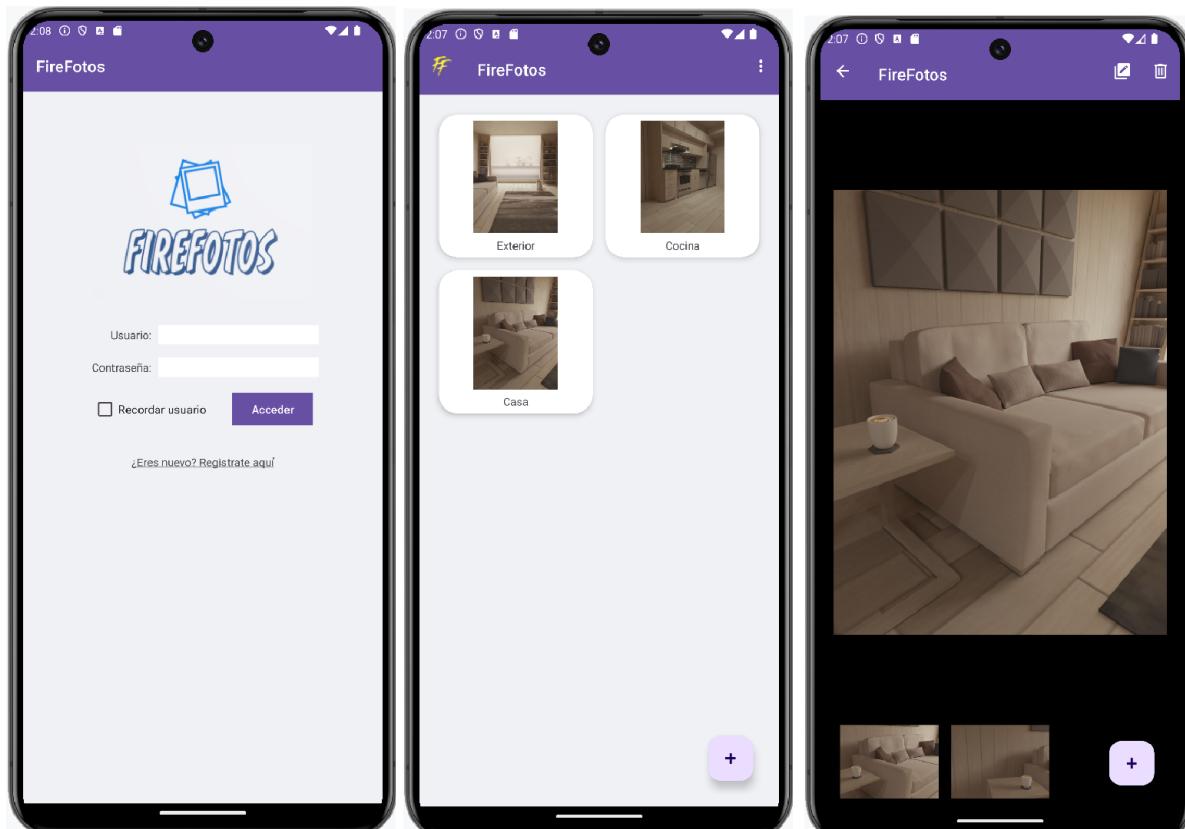


# PROYECTO 10

## FIRE FOTOS



En este proyecto haremos una app que usará el sistema Firebase de Google para almacenar álbumes de fotos de usuarios. Cada usuario registrado se identificará en la ubicación y podrá crear nuevos álbumes que rellenará con fotos tomadas por la cámara de su dispositivo móvil.

Durante su desarrollo se tratarán estos conceptos:

- Firebase
- Implementación de un sistema de login
- Shared Preferences
- Firestore
- Dialog Fragment
- RecyclerView con carousel
- Iconos SVG
- Acceso a la cámara
- DatePickerDialog y TimePickerDialog
- Cloud storage
- MaskableFrameLayout

## 1 – Firebase

Firebase es una plataforma para el desarrollo de aplicaciones (web, móvil, etc) creada por Google. Ofrece un montón de herramientas que utilizan las aplicaciones actuales, como las siguientes, que usaremos en este proyecto:

- Sistema de registro de usuarios: se admite la creación y validación de usuarios mediante correo y contraseña, identificación con una cuenta externa (google, facebook, github, etc)
- Firestore: Es una base de datos documental en la que se pueden crear colecciones de documentos. Cada documento posee parejas clave-valor y otras colecciones, siendo así apta para guardar datos estructurados jerárquicamente
- Cloud storage: Es un sistema que permite subir documentos, como imágenes, a la nube. El uso de esta herramienta consume créditos que hay que comprar, aunque Google da 300 créditos gratis para la fase de desarrollo. Puesto que hay que introducir datos bancarios, no usaremos esta herramienta aunque dejaremos el proyecto preparado para su posible uso.

Firebase dispone de muchas más herramientas, para monitorización, obtención de estadísticas, uso de IA, etc.

- Crea el proyecto en AndroidStudio
- Abre el archivo **AndroidManifest.xml** e incluye el permiso de acceso a internet:

```
1. <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2.   xmlns:tools="http://schemas.android.com/tools">
3.     <uses-permission android:name="android.permission.INTERNET"/>
4.     <!-- resto del archivo omitido -->
5.   </manifest>
```

- Abre el archivo **build.gradle** (versión project) y agrega las dependencias para usar el plugin **kapt** y los **servicios de google**

```
1. buildscript{
2.   repositories{
3.     google()
4.   }
5.   dependencies{
6.     classpath("androidx.navigation:navigation-safe-args-gradle-plugin:2.8.5")
7.   }
8. }
9. plugins {
10.   alias(libs.plugins.android.application) apply false
11.   alias(libs.plugins.kotlin.android) apply false
12.   kotlin("kapt") version "2.0.21"
13.   id("com.google.gms.google-services") version "4.4.2" apply false
14. }
```

En Logcat suelen aparecer constantemente excepciones lanzadas por los servicios de Google, pero pueden ser ignorados

- Abre el archivo **build.gradle** (versión module) y añade dependencias para usar el view binding, kapt, el plugin de navegación, la librería Coil, los servicios de google, el sistema de registro e identificación de usuarios de Firebase (Firebase auth) y la base de datos Firestore

```

1. plugins {
2.     alias(libs.plugins.android.application)
3.     alias(libs.plugins.kotlin.android)
4.     id("org.jetbrains.kotlin.kapt") // plugin kapt
5.     id("androidx.navigation.safeargs.kotlin") // plugin safe-args
6.     id("com.google.gms.google-services") // servicios de google
7. }
8. android {
9.     buildFeatures{
10.         viewBinding=true
11.     }
12.     // resto de la sección "android" omitida
13. }
14. dependencies {
15.     implementation("androidx.navigation:navigation-fragment-ktx:2.8.5") // plugin de navegación
16.     implementation(platform("com.google.firebase:firebase-bom:33.8.0")) // firebase
17.     implementation("io.coil-kt.coil3:coil:3.0.4") // coil
18.     implementation("io.coil-kt.coil3:coil-network-okhttp:3.0.4") // coil
19.     implementation("com.google.firebase:firebase-auth") // firebase auth
20.     // resto de las dependencias omitidas
21. }

```

- Accede a la plataforma de desarrollo Firebase (<https://firebase.google.com>) con una cuenta de Google y entra en la consola



- Pulsa el botón “Crear un proyecto” y crea un proyecto llamado “firefotos”, pulsando el botón de continuar al escribir el nombre

## Comencemos con el nombre de tu proyecto<sup>?</sup>

Nombre del proyecto

**firefotos**



Ya tienes un proyecto de Google Cloud?  
[Agregar Firebase al proyecto de Google Cloud](#)

Continuar

- Deshabilita Gemini, ya que no queremos el uso de IA en el proyecto. Cuando lo hagas, pulsa el botón de continuar



Habilitar Gemini en Firebase  
Recomendado

- Deshabilita las analíticas de Google, ya que no estamos interesados en obtener estadísticas de uso ni informes. Pulsa el botón para crear el proyecto cuando termines



- Una vez que el proyecto esté listo, pulsa el botón de “Android” en la imagen que aparece para agregar Firebase a una app

The screenshot shows the Firebase homepage. At the top left is the 'firefotos' logo. Next to it is a button labeled 'Plan Spark'. To the right is a large button with a play icon and the text '¿Todo listo para comenzar? Cuéntale a Gemini sobre tu proyecto'. Below these are two 3D-style characters: a black man on the left wearing glasses and a grey hoodie, and a white woman on the right with long brown hair tied back, wearing a yellow sweater. They are both gesturing towards a large orange and yellow flame icon. The background is dark grey. On the left side, there is text in Spanish: 'Comienza por agregar Firebase a tu app'. Below this are icons for iOS+, Android, and various development tools like React Native, Node.js, and Java. At the bottom left is the text 'Agrega una app para comenzar'.

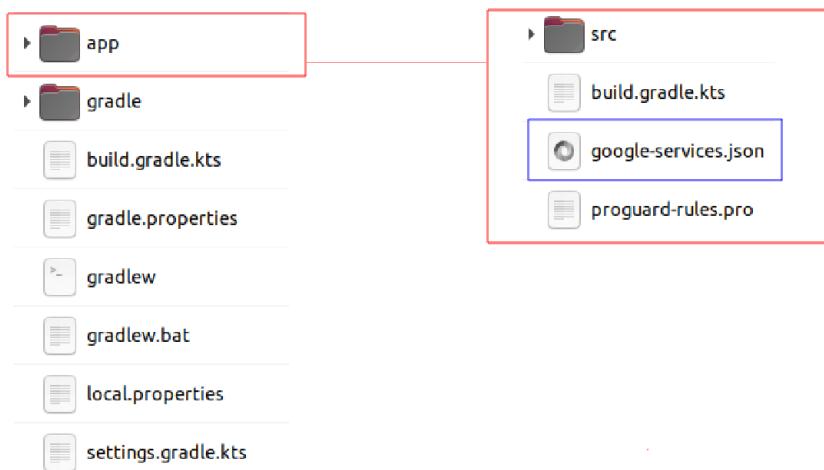
- En la pantalla que aparece, escribe el paquete, el nombre de la app y pulsa el botón para registrarla

- Una vez registrada la app, pulsa el enlace para descargar el archivo de configuración **google-services.json**

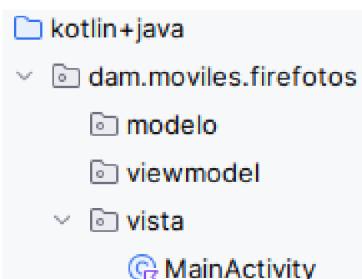
## 2 Descargar y, luego, agregar el archivo de configuración

[↓ Descargar google-services.json](#)

- Cuando hayas descargado el archivo, usa el explorador del sistema operativo para cortarlo y pegarlo dentro de la carpeta **app** del proyecto



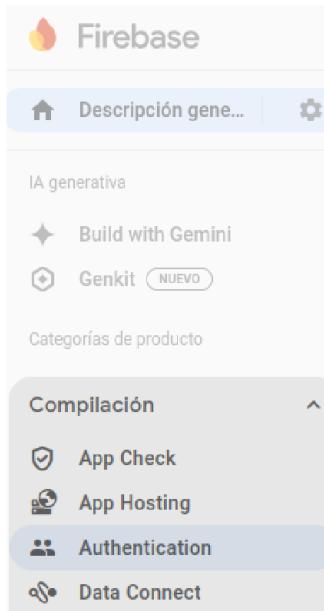
- Crea en el proyecto los paquetes **modelo**, **vista** y **viewmodel**, arrastrando **MainActivity** al paquete **vista**



## 2 – Implementación de un sistema de login

Firebase permite implementar fácilmente un completo sistema de login con el que podemos registrar y validar usuarios en el sistema. Esos usuarios pueden usar el resto de las herramientas de Firebase, como Firestore o el Cloud Storage.

- Entra en la consola de Firebase
- En el menú de la izquierda, pulsa sobre **Compilación** y después **Authentication**



- En la imagen central, pulsa el botón de comenzar

## Authentication

Autentica y administra usuarios de una gran variedad de proveedores sin ejecutar código en el servidor.

[Comenzar](#) [Preguntarle a Gemini](#)

- Elige como método de acceso **Correo electrónico/contraseña**

Agrega tu primer método de acceso y comienza a utilizar Firebase Auth

Proveedores nativos	Proveedores adicionales	Proveedores personalizados
Correo electrónico/contraseña	Google  Facebook	OpenID Connect
Teléfono	Play Juegos  Game Center	SAML
Anónimo	Apple  GitHub  Microsoft	
	Twitter  Yahoo	

La opción de correo y contraseña permite que los usuarios creen una cuenta usando como nombre de usuario su dirección de correo. Por defecto, los usuarios pueden entrar directamente, pero el sistema nos da la posibilidad de enviarle un correo electrónico para verificarlo, y solo dar acceso a los correos verificados.

- Pulsa la opción **Plantillas** y abajo a la izquierda, elige como idioma de la plantilla el **español**

Plantillas ⓘ

Correo electrónico

**Verificación de dirección de correo...** ⓘ

Restablecer contraseña

Cambio de dirección de correo ele...

Notificación sobre la inscripción d...

Configuración del SMTP

SMS

Verificación por SMS

Idioma de la plantilla  
español

Verificación de dirección de correo electrónico

Cuando un usuario se registra con una dirección de correo electrónico y una contraseña, puedes enviarle un mensaje de confirmación para verificar la dirección de correo electrónico que registró.  
[Más información](#)

Nombre del remitente  
no proporcionado

De  
noreply@firefotos-90dc8.firebaseioapp.com

Responder a  
noreply

Asunto  
Verifica tu dirección de correo electrónico de %APP\_NAME%

Mensaje

Hola, %DISPLAY\_NAME%:  
Haz clic en este enlace para verificar tu dirección de correo electrónico.  
[https://firefotos-90dc8.firebaseioapp.com/\\_auth/action?  
mode=action&oobCode=code](https://firefotos-90dc8.firebaseioapp.com/_auth/action?mode=action&oobCode=code)

Si no has emitido esta solicitud, ignora este mensaje.

Gracias,

El equipo de %APP\_NAME%

En esta ventana es posible configurar la plantilla del mensaje de correo que se envía a los usuarios para completar el registro, las posibilidades de recuperación de contraseña y otras opciones avanzadas, como la verificación en dos pasos.

- Abre **Android Studio** y en el paquete **vista** crea un fragmento llamado **LoginFragment** (el archivo xml asociado será **fragment\_login.xml**). Dicho fragmento tendrá dos capas:
  - Capa 1 → mostrará una imagen de carga (splash screen), que se verá solo cuando la app entre directamente con una cuenta de usuario que esté siendo recordada.
  - Capa 2 → mostrará un formulario para hacer login, dará la opción de recordar la contraseña y también tendrá una opción para que un usuario nuevo pueda registrarse



CAPA 1



CAPA 2

- Abre `fragment_login.xml` y añade un **FrameLayout** con dos capas. Diseña la primera capa, arrastrando un **ImageView** con el logo de la app y un **ProgressBar** centrados verticalmente. No es necesario poner ningún id a estos elementos, ya que son decorativos y no se usarán en el código fuente.

```

1. <FrameLayout xmlns:tools="http://schemas.android.com/tools"
2.   xmlns:app="http://schemas.android.com/apk/res-auto"
3.   xmlns:android="http://schemas.android.com/apk/res/android"
4.   android:layout_width="match_parent"
5.   android:layout_height="match_parent">
6.   <!-- capa 1 -->
7.   <LinearLayout
8.     android:id="@+id/capa1"
9.     android:orientation="vertical"
10.    android:layout_width="match_parent"
11.    android:gravity="center"
12.    android:layout_height="match_parent">
13.      <ImageView
14.        android:layout_width="wrap_content"
15.        android:layout_height="wrap_content"
16.        android:src="@drawable/logo"/>
17.      <ProgressBar
18.        android:layout_width="wrap_content"
19.        android:layout_height="wrap_content"
20.        android:layout_marginTop="32dp"/>
21.    </LinearLayout>
22. </FrameLayout>
```

- Tras el **LinearLayout**, añade la capa 2 de manera que no se vea (su **visibility** debe ser **gone**). Para poner el subrayado en el **TextView** llamado **txtNuevoUsuario**, es necesario que primero añadas su mensaje al archivo **strings.xml**

```

1.      <!-- capa 2 -->
2.      <LinearLayout
3.          android:id="@+id/capa2"
4.          android:visibility="gone"
5.          android:orientation="vertical"
6.          android:layout_width="match_parent" android:layout_height="match_parent"
7.          android:layout_marginTop="64dp" android:gravity="center_horizontal">
8.          <ImageView
9.              android:layout_width="wrap_content" android:layout_height="wrap_content"
10.             android:src="@drawable/logo"/>
11.          <LinearLayout
12.              android:layout_width="wrap_content" android:orientation="horizontal"
13.              android:gravity="center" android:layout_marginTop="32dp"
14.              android:layout_height="wrap_content">
15.              <TextView
16.                  android:layout_width="80dp" android:layout_height="wrap_content"
17.                  android:layout_marginEnd="8dp" android:textAlignment="textEnd"
18.                  android:text="Usuario:" />
19.              <EditText
20.                  android:id="@+id/txtUsuario" android:layout_width="200dp"
21.                  android:layout_height="wrap_content" />
22.          </LinearLayout>
23.          <LinearLayout
24.              android:layout_width="wrap_content" android:orientation="horizontal"
25.              android:gravity="center" android:layout_marginTop="16dp"
26.              android:layout_height="wrap_content">
27.              <TextView
28.                  android:layout_width="80dp" android:layout_height="wrap_content"
29.                  android:layout_marginEnd="8dp" android:textAlignment="textEnd"
30.                  android:text="Contraseña:" />
31.              <EditText
32.                  android:id="@+id/txtClave" android:layout_width="200dp"
33.                  android:inputType="textPassword"
34.                  android:layout_height="wrap_content"/>
35.          </LinearLayout>
36.          <LinearLayout
37.              android:layout_width="wrap_content" android:orientation="horizontal"
38.              android:gravity="center" android:layout_marginTop="16dp"
39.              android:layout_height="wrap_content">
40.              <CheckBox
41.                  android:id="@+id/chkRecordar" android:layout_width="wrap_content"
42.                  android:layout_height="wrap_content"
43.                  android:text="Recordar usuario"/>
44.              <Button
45.                  android:layout_width="wrap_content"
46.                  android:layout_height="wrap_content"
47.                  android:id="@+id/btnLogin" android:layout_marginLeft="32dp"
48.                  app:cornerRadius="0dp" android:layout_gravity="end"
49.                  android:text="Acceder"/>
50.          </LinearLayout>
51.          <TextView
52.              android:layout_width="wrap_content"
53.              android:layout_height="wrap_content"
54.              android:id="@+id/txtNuevoUsuario"
55.              android:layout_marginTop="32dp"
56.              android:text="@string/nuevo_usuario"/>
57.      </LinearLayout>

```

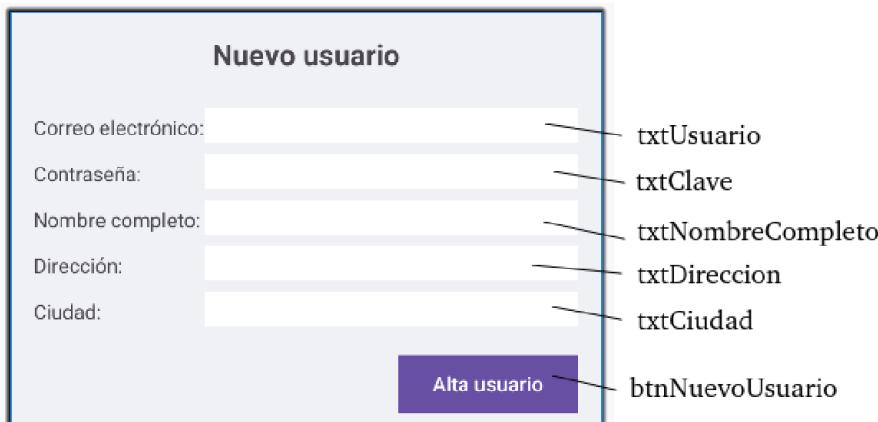
- Abre el archivo **strings.xml** y añade las etiquetas **<u>** y **</u>** al mensaje de clave **nuevo\_usuario**, con el fin de subrayar su contenido.

```

1.  <resources>
2.      <string name="nuevo_usuario"><u>¿Eres nuevo? Regístrate aquí</u></string>
3.      <!-- resto del archivo omitido -->
4.  </resources>

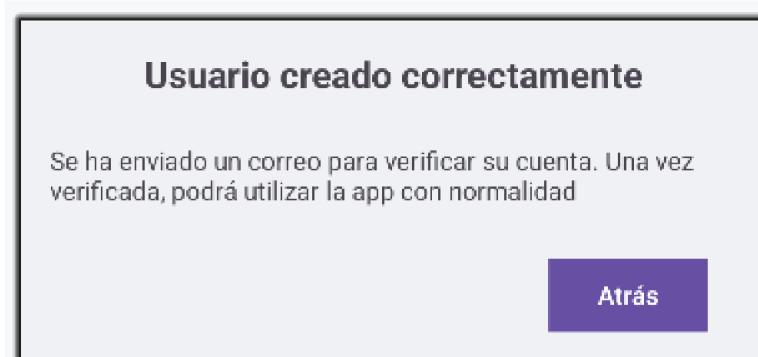
```

- Crea otro fragmento llamado **NuevoUsuarioFragment** (su vista será el archivo **fragment\_nuevo\_usuario.xml**) y dale el siguiente aspecto:



```
1. <LinearLayout
2.     xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:app="http://schemas.android.com/apk/res-auto"
4.     xmlns:tools="http://schemas.android.com/tools"
5.     android:layout_width="match_parent" android:layout_height="match_parent"
6.     android:orientation="vertical" android:padding="16dp">
7.     <TextView
8.         android:layout_width="match_parent" android:layout_height="wrap_content"
9.         android:text="Nuevo usuario" android:textAlignment="center"
10.        android:textSize="20sp" android:textStyle="bold" />
11.     <TableLayout
12.         android:layout_width="match_parent" android:layout_marginTop="24dp"
13.         android:layout_height="wrap_content" android:stretchColumns="1">
14.         <TableRow>
15.             <TextView
16.                 android:text="@string/correo_electronico"/>
17.             <EditText
18.                 android:id="@+id/txtUsuario"/>
19.         </TableRow>
20.         <TableRow android:layout_marginTop="8dp">
21.             <TextView
22.                 android:layout_marginRight="8dp"
23.                 android:text="@string/clave"/>
24.             <EditText
25.                 android:inputType="textPassword"
26.                 android:id="@+id/txtClave"/>
27.         </TableRow>
28.         <TableRow android:layout_marginTop="8dp">
29.             <TextView
30.                 android:text="@string/nombre_completo"/>
31.             <EditText
32.                 android:id="@+id/txtNombreCompleto"/>
33.         </TableRow>
34.         <TableRow android:layout_marginTop="8dp">
35.             <TextView
36.                 android:layout_marginRight="8dp"
37.                 android:text="@string/direcci_n"/>
38.             <EditText
39.                 android:id="@+id/txtDireccion"/>
40.         </TableRow>
41.         <TableRow android:layout_marginTop="8dp">
42.             <TextView
43.                 android:text="@string/ciudad"
44.                 android:layout_marginRight="8dp" />
45.             <EditText
46.                 android:id="@+id/txtCiudad"/>
47.         </TableRow>
48.     </TableLayout>
49.     <Button
50.         android:layout_height="wrap_content" android:layout_width="wrap_content"
51.         android:text="@string/alta_usuario" app:cornerRadius="0dp"
52.         android:layout_marginTop="16dp" android:layout_gravity="right"
53.         android:id="@+id/btnNuevoUsuario"/>
54.     </LinearLayout>
```

- Crea un nuevo fragment llamado **CorreoEnviadoFragment** (su vista será el archivo **fragment\_correo\_enviado.xml**) y dale el siguiente aspecto (sus elementos son decorativos y no llevan id. En el archivo **strings.xml** añade un string llamado **correo\_enviado** con el mensaje central):



```

1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.   xmlns:tools="http://schemas.android.com/tools"
3.   android:layout_width="match_parent"
4.   android:layout_height="match_parent"
5.   xmlns:app="http://schemas.android.com/apk/res-auto"
6.   android:orientation="vertical"
7.   android:padding="16dp">
8.   <TextView
9.     android:id="@+id/textView2"
10.    android:layout_width="match_parent"
11.    android:layout_height="wrap_content"
12.    android:text="@string/usuario_creado_correctamente"
13.    android:textAlignment="center"
14.    android:textSize="20sp"
15.    android:textStyle="bold" />
16.   <TextView
17.     android:layout_width="wrap_content"
18.     android:layout_height="wrap_content"
19.     android:layout_marginTop="24dp"
20.     android:text="@string/correo_enviado"/>
21.   <Button
22.     android:id="@+id/btnAtras"
23.     app:cornerRadius="0dp"
24.     android:layout_width="wrap_content"
25.     android:layout_height="wrap_content"
26.     android:text="Atrás"
27.     android:layout_gravity="right"
28.     android:layout_marginRight="16dp"
29.     android:layout_marginTop="24dp"/>
30. </LinearLayout>
```

- Crea un nuevo fragment llamado **ErrorFragment** (su vista asociada será el archivo **fragment\_error.xml**) y dale este aspecto:

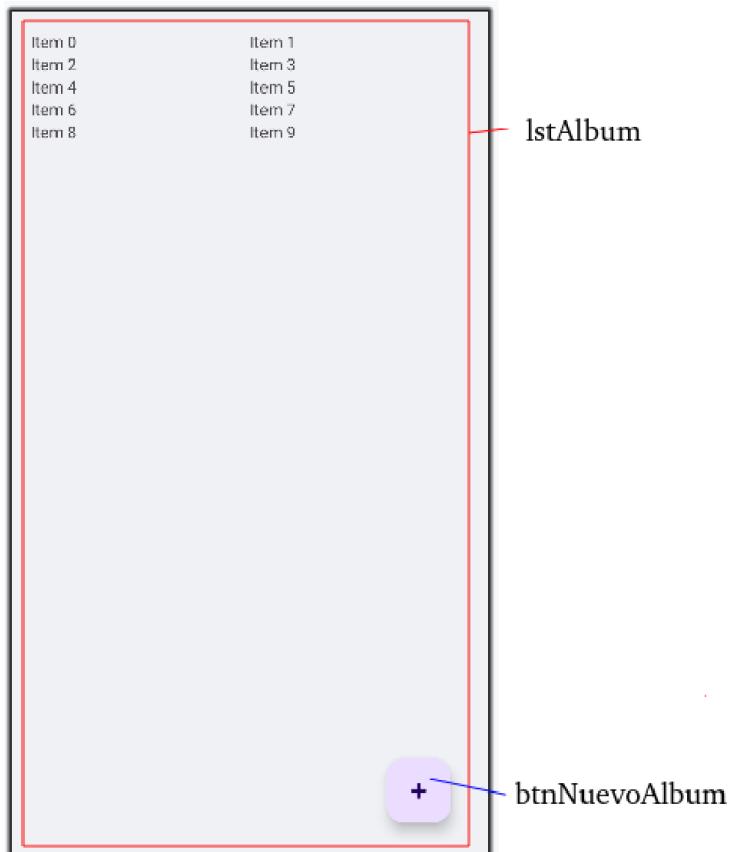


```

1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     xmlns:tools="http://schemas.android.com/tools"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     xmlns:app="http://schemas.android.com/apk/res-auto"
6.     android:orientation="vertical"
7.     android:padding="16dp">
8.     <TextView
9.         android:layout_width="match_parent"
10.        android:layout_height="wrap_content"
11.        android:text="Se ha producido un error"
12.        android:textAlignment="center"
13.        android:textSize="20sp"
14.        android:textStyle="bold" />
15.     <TextView
16.         android:id="@+id/txtMensajeError"
17.         android:layout_width="wrap_content"
18.         android:layout_height="wrap_content"
19.         android:layout_marginTop="24dp"
20.         tools:text="Aquí viene el mensaje de error"/>
21.     <Button
22.         android:id="@+id/btnAtras"
23.         android:layout_width="wrap_content"
24.         android:layout_height="wrap_content"
25.         android:layout_marginTop="24dp"
26.         android:layout_gravity="right"
27.         app:cornerRadius="0dp"
28.         android:text="@string/atras"/>
29. </LinearLayout>

```

- Añade un nuevo fragment llamado **PrincipalFragment** (su vista xml será **fragment\_principal.xml**) con dos capas: una tendrá un **RecyclerView** (con un **GridLayoutManager**) y la otra un **FloatingActionButton** situado en la esquina inferior derecha



```

1. <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     xmlns:tools="http://schemas.android.com/tools"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     xmlns:app="http://schemas.android.com/apk/res-auto"
6.     android:padding="16dp">
7.     <!-- CAPA 1 : RECYCLER VIEW -->
8.     <androidx.recyclerview.widget.RecyclerView
9.         android:layout_width="match_parent"
10.        android:layout_height="match_parent"
11.        app:layoutManager="androidx.recyclerview.widget.GridLayoutManager"
12.        app:spanCount="2"
13.        android:id="@+id/lstAlbum"/>
14.     <!-- CAPA 2 : FLOATING ACTION BUTTON -->
15.     <LinearLayout
16.         android:layout_width="match_parent"
17.         android:layout_height="match_parent"
18.         android:gravity="bottom|end"
19.         android:orientation="vertical">
20.         <com.google.android.material.floatingactionbutton.FloatingActionButton
21.             android:id="@+id/btnNuevoAlbum"
22.             android:layout_width="wrap_content"
23.             android:layout_height="wrap_content"
24.             android:layout_margin="16dp"
25.             android:src="@android:drawable/ic_input_add" />
26.     </LinearLayout>
27. </FrameLayout>

```

- Elimina el código obsoleto de todos los fragments y habilita el view binding en todos ellos. Por ejemplo, en **LoginFragment** sería así:

```

1. class LoginFragment : Fragment(), NavegadorError {
2.     private var _binding: FragmentLoginBinding? = null
3.     val binding: FragmentLoginBinding
4.         get() = checkNotNull(_binding)
5.     override fun onCreateView(
6.         inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
7.     ): View? {
8.         inicializarBinding(inflater, container)
9.         return binding.root
10.    }
11.    private fun inicializarBinding(inflater: LayoutInflater, container: ViewGroup?) {
12.        _binding = FragmentLoginBinding.inflate(inflater, container, false)
13.    }
14.    override fun onDestroyView() {
15.        super.onDestroyView()
16.        _binding=null
17.    }
18. }

```

- Abre **MainActivity** y habilita el view binding

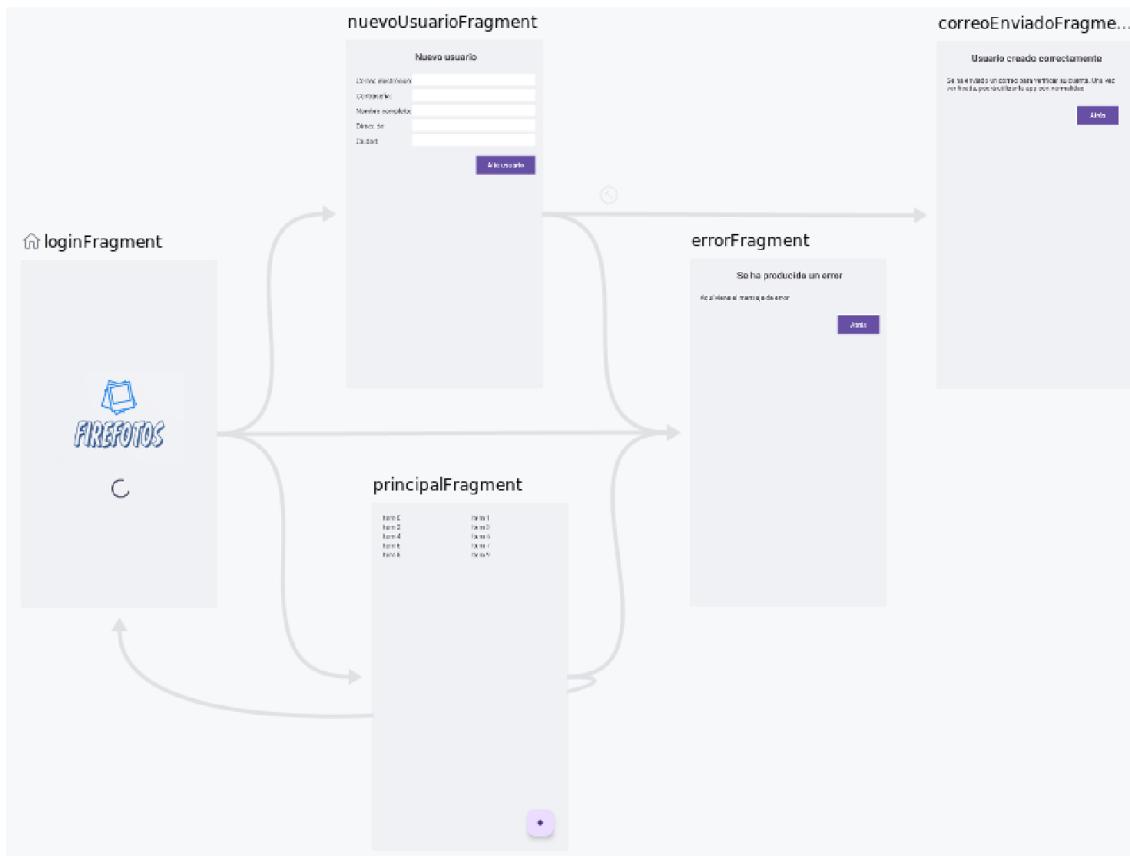
```

1. class MainActivity : AppCompatActivity() {
2.     lateinit var binding: ActivityMainBinding
3.     override fun onCreate(savedInstanceState: Bundle?) {
4.         super.onCreate(savedInstanceState)
5.         inicializarBinding()
6.         setContentView(binding.root)
7.     }
8.     fun inicializarBinding(){
9.         binding=ActivityMainBinding.inflate(layoutInflater)
10.    }
11. }

```

- En la carpeta **res** pulsa el botón derecho y crea un recurso de tipo **Navigation** llamado **nav\_graph.xml**

- Abre el **nav\_graph.xml** con el diseñador gráfico y añade todos los fragments que has creado (el primer fragment a añadir debe ser **LoginFragment**, conectándolos de esta forma:



- Sobre el **nav\_graph.xml** y añade a **errorFragment** un parámetro de tipo **String** llamado **mensajeError**



- Abre **ErrorFragment** y crea en él estos dos métodos, que deberán ser llamados en **onCreateView**:

- inicializarMensajeError**: su objetivo es recoger el argumento **mensajeError** y mostrarlo en **txtMensajeError**
- inicializarBoton**: hace que al pulsar **btnAtras** se navegue hacia la pantalla previa

```

1. class ErrorFragment : Fragment() {
2.     // resto de la clase omitido
3.     override fun onCreateView(
4.         inflater: LayoutInflater, container: ViewGroup?,
5.         savedInstanceState: Bundle?
6.     ): View? {
7.         inicializarBinding(inflater,container)
8.         inicializarMensajeError()
9.         inicializarBoton()
10.        return binding.root
11.    }
12.    fun inicializarBoton(){
13.        binding.btnAtras.setOnClickListener {
14.            findNavController().popBackStack()
15.        }
16.    }
17.    fun inicializarMensajeError(){
18.        val mensaje = ErrorFragmentArgs.fromBundle(requireArguments()).mensajeError
19.        binding.txtMensajeError.text = mensaje
20.    }
21. }

```

Como hay muchos fragments que pueden conducirnos a la **ErrorFragment**, para no duplicar constantemente el código para navegar a dicha pantalla, vamos a hacer una interfaz que nos permita navegar a dicha pantalla a todos los fragments que la implementen.

- En el paquete **modelo**, programa una interfaz llamada **NavegadorError** que permita navegar hacia la pantalla de error a cualquier fragment que la implemente.

```

1. interface NavegadorError {
2.     fun getNavController():NavController
3.     fun getFlechaNavegacionPantallaError(m:String):NavDirections
4.     fun navegarPantallaError(m:String){
5.         val nc = getNavController()
6.         val flecha = getFlechaNavegacionPantallaError(m)
7.         nc.navigate(flecha)
8.     }
9. }

```

- Haz que **LoginFragment**, **NuevoUsuarioFragment** y **PrincipalFragment** implementen la interfaz **NavegadorError**, sobreescribiendo sus métodos abstractos de esta forma:

```

1. // XXX es uno de estos fragment: LoginFragment,PrincipalFragment,NuevoUsuarioFragment
2. class XXX : Fragment(),NavegadorError {
3.     // inicio de la clase omitido
4.     override fun getNavController(): NavController = findNavController()
5.     override fun getFlechaNavegacionPantallaError(m: String): NavDirections
6.         = XXXDirections.actionXXXToErrorFragment(m)
7. }

```

Tras hacer todo esto, **LoginFragment**, **NuevoUsuarioFragment** y **PrincipalFragment** adquieren un método llamado **navegarPantallaError** que les lleva automáticamente a **ErrorFragment**

- Abre el **nav\_graph.xml**, pulsa la flecha que conecta **nuevoUsuarioFragment** con **correoEnviadoFragment** y en la propiedad **popUpTo** elige **loginFragment**

Al hacer esto, cuando el usuario se encuentre en **CorreoEnviadoFragment** y pulse el botón de atrás, será conducido directamente a **LoginFragment**. Es como si la flecha de atrás se “saltara” a **NuevoUsuarioFragment**

- Abre **activity\_main.xml**, borra su contenido y añade de elemento principal un **LinearLayout** vertical que contenga una **MaterialToolbar** oculta (pon su **visibility a gone**) y un **FragmentContainerView** que navegue por el **nav\_graph**

```

1. <LinearLayout
2.     xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:app="http://schemas.android.com/apk/res-auto"
4.     android:layout_width="match_parent"
5.     android:layout_height="match_parent" android:orientation="vertical">
6.     <com.google.android.material.appbar.MaterialToolbar
7.         android:layout_width="match_parent" android:layout_height="?attr/actionBarSize"
8.         android:visibility="gone" style="@style/Widget.MaterialComponents.Toolbar.Primary"
9.         android:id="@+id/material_toolbar"/>
10.    <androidx.fragment.app.FragmentContainerView
11.        xmlns:android="http://schemas.android.com/apk/res/android"
12.        xmlns:app="http://schemas.android.com/apk/res-auto"
13.        android:layout_width="match_parent" android:layout_height="match_parent"
14.        android:id="@+id/fragment_container_view"
15.        android:name="androidx.navigation.fragment.NavHostFragment"
16.        app:navGraph="@navigation/nav_graph"
17.        app:defaultNavHost="true"/>
18. </LinearLayout>
```

- Añade a **MainActivity** un método llamado **inicializarToolbar** cuya misión sea sustituir la toolbar que viene por defecto por la **MaterialToolbar**, y llámalo dentro de **onCreate**

```

1. class MainActivity : AppCompatActivity() {
2.     // resto de la clase omitida
3.     override fun onCreate(savedInstanceState: Bundle?) {
4.         // resto del método omitido
5.         inicializarToolbar()
6.     }
7.     fun inicializarToolbar(){
8.         setSupportActionBar(binding.materialToolbar)
9.     }
10. }
```

- Abre el archivo **themes.xml** y añade reglas de estilo para que el fondo de todas las ventanas sea el color **#F0F1F5** y el color de fondo de los cuadros de texto sea blanco

```

1. <resources xmlns:tools="http://schemas.android.com/tools">
2.     <style name="Base.Theme.FireFotos" parent="Theme.Material3.DayNight.NoActionBar">
3.         <item name="android:windowBackgroundColor">#F0F1F5</item>
4.         <item name="android:editTextBackgroundColor">@color/white</item>
5.     </style>
6.     <style name="Theme.FireFotos" parent="Base.Theme.FireFotos" />
7. </resources>
```

- Ejecuta la app y comprueba que se ve la pantalla de carga de **LoginFragment**
- En el paquete **modelo** crea una **data class** llamada **Usuario**, que guardará toda la información que necesitamos sobre los usuarios de nuestra app

```

1. data class Usuario(
2.     val id:String = "",
3.     val correo:String = "",
4.     val clave:String = "",
5.     val nombre:String = "",
6.     val direccion:String = "",
7.     val ciudad:String = "",
8. )

```

Para que la clase **Usuario** pueda ser almacenada en la base de datos **Firestore**, es necesario que todos los parámetros de su constructor deban tener un valor por defecto (da igual el valor, aquí ponemos una cadena de texto vacía).

- A continuación, también en el paquete **modelo**, crea una interfaz llamada **LoginManager**, que va a tener métodos para dar de alta un usuario y para identificar un usuario mediante su correo y contraseña.

```

1. interface LoginManager {
2.     // crea un usuario en la app
3.     fun crearUsuario(
4.         correo: String,
5.         clave: String,
6.         nombre: String,
7.         direccion: String,
8.         ciudad: String,
9.         lambdaExito: (Usuario) -> Unit,
10.        lambdaError: (String) -> Unit
11.    )
12.    // identifica un usuario en la app
13.    fun login(
14.        correo: String,
15.        clave: String,
16.        lambdaExito: (Usuario) -> Unit,
17.        lambdaError: (String) -> Unit
18.    )
19.    // cierra la sesión del usuario activo
20.    fun logout()
21. }

```

Haciendo una interfaz para gestionar el login, hacemos que nuestra app sea independiente del sistema de gestión de usuarios usado. Así, si alguna vez, en lugar de Firebase usamos otro sistema para registrar usuarios, bastará con hacer una nueva implementación de esta interfaz, sin necesidad de reprogramar el cuerpo de la app

La interfaz está diseñada para trabajar de forma asíncrona, de manera que una vez que el usuario es creado, o identificado, se ejecutan expresiones lambda en caso de éxito o error de la operación. Esto se debe a que los procesos de creación de usuario y login pueden consumir tiempo en completarse<sup>1</sup>.

- **lambdaExito:** Es una expresión lambda que recibe el objeto **Usuario** que ha sido creado (o identificado) y hace algo con él
- **lambdaError:** Es una expresión lambda que recibe un mensaje de error y hace algo con él (normalmente, navegar a la pantalla de error).

---

<sup>1</sup> Un enfoque alternativo sería el uso de **corrutinas**, pero aquí se ha preferido el enfoque de pasar lambdas, debido a que Firebase trabaja de esta forma para la realización del login

- En el paquete **modelo**, crea una clase llamada **FirebaseLogin**, que implemente la interfaz **LoginManager** usando **Firebase**.

```

1. class FirebaseLogin : LoginManager{
2.     override fun crearUsuario(
3.         correo: String,
4.         clave: String,
5.         nombre: String,
6.         direccion: String,
7.         ciudad: String,
8.         lambdaExito: (Usuario) -> Unit,
9.         lambdaError: (String) -> Unit
10.    ) {
11.        // creamos un usuario en Firebase
12.    }
13.    override fun login(
14.        correo: String,
15.        clave: String,
16.        lambdaExito: (Usuario) -> Unit,
17.        lambdaError: (String) -> Unit
18.    ) {
19.        // hacemos que el usuario se identifique en Firebase
20.    }
21.    override fun logout() {
22.        // cerramos la sesión Firebase
23.    }
24. }
```

- Añade al archivo **LoginManager.kt** una función llamada **getLoginManager** que nos devuelva un objeto **FirebaseLogin**

```

1. interface LoginManager {
2.     // código omitido
3. }
4. fun getLoginManager():LoginManager = FirebaseLogin()
```

La función **getLoginManager** nos devuelve la implementación de **LoginManager** que vaya a usar nuestra app. Aquí usamos la implementación de **Firebase**, pero si en el futuro se cambiase a otro sistema, haríamos una nueva clase y la devolveríamos ahí.

- En **FirebaseLogin**, programa el método **crearUsuario** para que siga estos pasos:

- Crea en **Firebase** el usuario cuyo correo y contraseña se recibe como parámetro, haciendo el uso del objeto **Firebase.auth** y su método **createUserWithEmailAndPassword**
- Una vez creado, obtenemos el usuario registrado en **Firebase** y le enviamos un correo para verificar su cuenta. Eso se hace con el método **sendEmailVerification**
- Si todo va bien, se creará un objeto **Usuario** con todos los datos del usuario, y más adelante, ese objeto se guardará en la base de datos **Firestore** (ya que Firebase no guarda información de usuarios como su ciudad o dirección)
- En caso de fallo, se llamará a **lambdaError**

```

1. override fun crearUsuario(
2.     correo: String,
3.     clave: String,
4.     nombre: String,
5.     direccion: String,
6.     ciudad: String,
7.     lambdaExito: (Usuario) -> Unit,
8.     lambdaError: (String) -> Unit
9. ) {
10.     Firebase.auth
11.         .createUserWithEmailAndPassword(correo, clave)
12.         .addOnSuccessListener {
13.             val usuarioFB = checkNotNull(Firebase.auth.currentUser)
14.             usuarioFB
15.                 .sendEmailVerification()
16.                 .addOnSuccessListener {
17.                     val u = Usuario(usuarioFB.uid, correo, clave, nombre, direccion, ciudad)
18.                     // PENDIENTE: nos falta guardar el usuario creado en Firestore
19.                     lambdaExito(u)
20.                 }.addOnFailureListener {
21.                     lambdaError("No se ha podido enviar un correo de verificación")
22.                 }
23.             }.addOnFailureListener { excepcion ->
24.                 lambdaError(excepcion.message.toString())
25.             }
26. }

```

Firebase pone a nuestra disposición el objeto **Firebase.auth**, que contiene los métodos para crear y validar usuarios que hemos usado. Cabe destacar que todas las operaciones de **Firebase** son asíncronas y se hacen con llamadas a métodos encadenados, que al final terminan con una de estas opciones:

- **addOnSuccessListener**: Permite definir una lambda que se lanza si todo va bien
- **addOnFailureListener**: Permite definir una lambda que se lanza si se produce un error. Dicha lambda recibe como parámetro un objeto **Exception**
- **await**: Si en lugar de los dos métodos anteriores, queremos usar **corrutinas**, el método **await** es un punto de suspensión que hace que el hilo que ejecuta la corrutina quede libre hasta que la operación finalice.

La elección entre pasar lambdas como parámetros o usar corrutinas dependerá de lo que más interese en cada momento. Para el sistema de login es más interesante el enfoque de pasar lambdas como parámetros.

- En **FirebaseLogin**, programa el método **login** para que siga estos pasos:
  - Usa el método **signInWithEmailAndPassword** del objeto **Firebase.auth** para que el usuario cuyo correo y contraseña accedan al sistema
  - En caso de que se acceda con éxito, se comprobará si el usuario ha verificado su cuenta de correo, y en dicho caso ejecutaremos la expresión **lambdaExito**. Provisionalmente, crearemos un objeto **Usuario** que solo tiene el correo y la contraseña. Más adelante, recuperaremos de **Firestore** el resto de su información.
  - Si en algún momento algo falla, se ejecutará **lambdaError** pasándole un mensaje con el error detectado.

```

1. override fun login(
2.     correo: String,
3.     clave: String,
4.     lambdaExito: (Usuario) -> Unit,
5.     lambdaError: (String) -> Unit
6. ) {
7.     try {
8.         Firebase.auth
9.             .signInWithEmailAndPassword(correo, clave)
10.            .addOnSuccessListener {
11.                val usuarioFB = Firebase.auth.currentUser
12.                if (usuarioFB != null) {
13.                    if (usuarioFB.isEmailVerified) {
14.                        // PROVISIONAL: Aquí más adelante recuperaremos el usuario de Firestore
15.                        val u = Usuario("",correo,clave,"","","")
16.                        lambdaExito(u)
17.                    } else {
18.                        lambdaError("El correo del usuario no está validado")
19.                    }
20.                } else {
21.                    lambdaError("No se pudo identificar al usuario")
22.                }
23.            }.addOnFailureListener { excepcion ->
24.                lambdaError(excepcion.message.toString())
25.            }
26.        }catch(e:Exception){
27.            lambdaError(e.message.toString())
28.        }
29.    }

```

Si el método **signInWithEmailAndPassword** tiene éxito, en su **SuccessListener** recuperamos el objeto que representa al usuario conectado a **Firebase**, y entonces se llama sobre él a **isEmailVerified**, que nos dirá si ha verificado su cuenta.

- En **FirebaseLogin**, programa el método **logout** de forma que se cierre la conexión con **Firebase** y el usuario que ha accedido cierre su sesión.

```

1. override fun logout() {
2.     Firebase.auth.signOut()
3. }

```

- En el paquete **modelo**, crea una clase llamada **Sesion**, cuya misión va a ser almacenar el objeto **Usuario** que ha accedido al sistema y hacerlo disponible a todos los fragments. En caso de que no haya ningún usuario en el sistema, dicha variable valdrá **null**

```

1. class Sesion {
2.     private var _usuario:Usuario? = null
3.     val usuario:Usuario
4.         get() = checkNotNull(_usuario) { "No hay ningún usuario en la sesión" }
5. }

```

La variable de instancia **\_usuario** guarda el usuario identificado en el sistema, o **null** si no hay ninguno. La variable de instancia **usuario** solo sirve para obtener el valor de **\_usuario** cuando este no es nulo, o lanzar una excepción en caso de que lo sea.

- Convierte la clase **Sesion** en **singleton**, esto es, una clase de la que solo puede existir un objeto en la memoria, que se guardará en una variable estática llamada **INSTANCIA**. Los pasos son:

- Se hace privado el constructor de **Sesion**
- Se añade un **companion object** (equivalente al contenido estático en Java) que contenga la variable **INSTANCIA** con valor inicial **null**
- Se añade al **companion object** un método llamado **getInstancia** que devuelve la variable **INSTANCIA**, o la crea si es **null**

```

1. class Sesion private constructor() {
2.     // inicio de la clase omitido
3.     companion object {
4.         private var INSTANCIA:Sesion? = null
5.         fun getInstancia():Sesion {
6.             if (INSTANCIA == null) {
7.                 INSTANCIA = Sesion()
8.             }
9.             return checkNotNull(INSTANCIA)
10.        }
11.    }
12. }
```

Para poner privado el constructor de una clase, se añade **private constructor()** a su cabecera.

- Añade a la clase **Sesion** los siguientes métodos:
  - **iniciarSesion**: Permite a un usuario iniciar sesión usando el **LoginManager** que esté configurado en la app y ejecutar un **lambdaExito** o **lambdaError** según el resultado del inicio de la sesión
  - **cerrarSesion**: Permite cerrar la sesión del usuario que previamente haya sido identificado
  - **sesionIniciada**: Devuelve true si hay un usuario en la sesión

```

1. class Sesion private constructor() {
2.     // resto de la clase omitido
3.     fun iniciarSesion(
4.         correo:String,
5.         clave:String,
6.         lambdaExito: (Sesion) -> Unit,
7.         lambdaError: (String) -> Unit
8.     ){
9.         getLoginManager().login(
10.             correo,
11.             clave,
12.             lambdaExito = { usuario ->
13.                 _usuario = usuario
14.                 lambdaExito(this)
15.             },
16.             lambdaError = { m -> lambdaError(m)}
17.         )
18.     }
19.     fun cerrarSesion(){
20.         _usuario = null
21.         getLoginManager().logout()
22.     }
23.     fun sesionIniciada() = _usuario!=null
24. }
```

Ahora que ya tenemos hechas las clases que hacen el registro e identificación del usuario, vamos a integrarlas en el **LoginFragment**

- Abre **LoginFragment** y añade un método llamado **inicializarInterfaz**, que se llamará en **onCreateView** y se comportará de esta forma:
  - Si en la sesión no hay ningún objeto **Usuario**, se llamará a un método llamado **mostrarCapaLogin**, que hará visible la capa 2 de **fragment\_login.xml**
  - Si hay un objeto **Usuario** en la sesión, se finalizará la app. Esto se debe a que si ya existe un objeto **Usuario** en la sesión es porque estamos volviendo hacia atrás, y hemos llegado a la pantalla de login. En este caso, lo más natural es finalizar la app.

```

1. fun inicializarInterfaz() {
2.     if(Sesion.getInstancia().sesionIniciada()){
3.         activity?.finish()
4.         System.exit(0)
5.     }else{
6.         mostrarCapaLogin()
7.     }
8. }
9. }
10. fun mostrarCapaLogin(){
11.     binding.capa1.visibility=View.GONE
12.     binding.capa2.visibility = View.VISIBLE
13. }
```

- Ejecuta la app y comprueba que se ve el formulario de login pero sus botones no hacen nada
- Añade a **LoginFragment** un método llamado **inicializarBotones**, que llamarás dentro de **mostrarCapaLogin**. En **inicializarBotones** haz que al pulsar el mensaje **txtNuevoUsuario**, se navegue hacia **NuevoUsuarioFragment**

```

1. private fun inicializarBotones() {
2.     binding.txtNuevoUsuario.setOnClickListener{
3.         val nc = findNavController()
4.         val flecha = LoginFragmentDirections.actionLoginFragmentToNuevoUsuarioFragment()
5.         nc.navigate(flecha)
6.     }
7. }
```

- Ejecuta la app y comprueba que al pulsar **txtNuevoUsuario** se navega hacia la ventana de registro. Comprueba también que al volver atrás se vuelve a la pantalla de login.
- Abre **NuevoUsuarioFragment** y añade un método llamado **inicializarBoton** que llamarás en **onCreateView**. Dentro de **inicializarBoton** haz que al pulsar **btnNuevoUsuario** se llame al método **crearUsuario** del objeto **LoginManager** pasándole todos los datos escritos en los cuadros de texto. Si todo va bien, se navegará a **CorreoEnviadoFragment** y en caso de error, a la pantalla de error.

```

1. fun inicializarBoton(){
2.     binding.btnNuevoUsuario.setOnClickListener {
3.         getLoginManager().crearUsuario(
4.             binding.txtUsuario.text.toString(),
5.             binding.txtClave.text.toString(),
6.             binding.txtNombreCompleto.text.toString(),
7.             binding.txtDireccion.text.toString(),
8.             binding.txtCiudad.text.toString(),
9.             lambdaExito = {
10.                 val nc=findNavController()
11.                 val flecha = NuevoUsuarioFragmentDirections
12.                     .actionNuevoUsuarioFragmentToCorreoEnviadoFragment()
13.                 nc.navigate(flecha)
14.             },
15.             lambdaError = { m -> navegarPantallaError(m) }
16.         )
17.     }
18. }

```

- Abre **CorreoEnviadoFragment** y añade un método **inicializarBoton** que será llamado en **onCreateView**. En ese método se programará que al pulsar **btnAtras** se vuelva a la pantalla previa.

```

1. fun inicializarBoton(){
2.     binding.btnAtras.setOnClickListener {
3.         findNavController().popBackStack()
4.     }
5. }

```

- Ejecuta la app y crea un usuario con tu cuenta de correo electrónico. Comprueba que al darlo de alta, se le envía un mensaje de correo a esa cuenta pidiendo su verificación. Comprueba también que al pulsar el botón de atrás en **CorreoEnviadoFragment** se navega directamente hacia **LoginFragment**, sin pasar otra vez por **NuevoUsuarioFragment**
- Abre **LoginFragment** y añade un método llamado **hacerLogin** que reciba un usuario, una contraseña e inicie sesión. Si las credenciales son correctas, se navegará a **PrincipalFragment**, y si no, a la pantalla de error. De momento, se ignorará la casilla para recordar el usuario.

```

1. fun hacerLogin(nombre:String,clave:String){
2.     Sesion.getInstancia().iniciarSesion(
3.         nombre,
4.         clave,
5.         lambdaExito = { sesion ->
6.             val nc=findNavController()
7.             val flecha = LoginFragmentDirections.actionLoginFragmentToPrincipalFragment()
8.             nc.navigate(flecha)
9.         },
10.         lambdaError = { error -> navegarPantallaError(error) }
11.     )
12. }

```

- Añade al método **inicializarBotones** de **LoginFragment** código para que al pulsar **btnLogin** se llame al método **hacerLogin** pasando el usuario y contraseña escritos en los cuadros de texto.

```

1. private fun inicializarBotones() {
2.     // inicio del método omitido
3.     binding.btnLogin.setOnClickListener {
4.         hacerLogin(
5.             binding.txtUsuario.text.toString(),
6.             binding.txtClave.text.toString()
7.         )
8.     }
9. }

```

- Ejecuta la app y comprueba que si accedes con un usuario/contraseña correcta, se navega hacia **PrincipalFragment** y si no, se navega a la pantalla de error con un mensaje descriptivo de lo que ha sucedido (cuenta no verificada, usuario/contraseña incorrecto, etc).

### 3 – Shared Preferences

Para no tener que estar constantemente identificando al usuario cada vez que ejecuta la app, vamos a implementar un sistema para guardar sus credenciales en el almacenamiento, de forma que al iniciar la app, estas se recuperen y automáticamente se inicie la sesión. Aunque no es la forma más segura, usaremos **SharedPreferences** para implementar este sistema, debido a su facilidad.

**SharedPreferences** es un sistema que permite a una app almacenar parejas **clave-valor** en el sistema de almacenamiento dedicado a la app.

Al igual que hicimos con el login, usaremos una interfaz para definir las necesidades de almacenamiento de la app, de manera que si en el futuro queremos cambiar de **SharedPreferences** a otro sistema, solo tengamos que programar una nueva implementación de dicha interfaz.

- En el paquete **modelo**, crea una data class llamada **Credenciales**, que guardará un nombre de usuario y una contraseña. Esta clase se usará para manejar conjuntamente ambos datos.

```

1. data class Credenciales(
2.     val usuario:String,
3.     val clave:String
4. )

```

- Añade también al paquete **modelo** una nueva interfaz llamada **GestorCredenciales**, que tendrá métodos para guardar un par usuario/contraseña en el almacenamiento, recuperarlo (en caso de que haya alguno guardado), y borrarlo.

```

1. interface GestorCredenciales {
2.     fun guardarCredenciales(nombre:String,clave:String):Boolean
3.     fun borrarCredenciales():Boolean
4.     fun getCredencialesGuardadas():Credenciales?
5. }

```

- En el paquete **modelo** crea la clase **GestorCredencialesSharedPreferences**, que será la implementación de **GestorCredenciales** usando **SharedPreferences**

```

1. class GestorCredencialesSharedPreferences(
2.     val sharedPreferences:SharedPreferences
3. ) : GestorCredenciales {
4.     override fun guardarCredenciales(nombre: String, clave: String): Boolean {
5.         // guardamos el par usuario/clave usando SharedPreferences
6.     }
7.     override fun getCredencialesGuardadas(): Credenciales? {
8.         // obtenemos el par usuario/clave guardado, o null si no hay ninguno
9.     }
10.    override fun borrarCredenciales(): Boolean = this.guardarCredenciales("", "")
11. }
```

El constructor de la clase recibe un objeto **SharedPreferences**, que permite almacenar y recuperar parejas clave-valor en el sistema de almacenamiento.

Para borrar las credenciales, basta con guardar como clave y contraseña una cadena de texto vacía.

- Programa el método **guardarCredenciales** para que se almacene el nombre y la clave en el sistema de almacenamiento reservado a la app. Se usarán las claves “usuario” y “clave” para almacenarlas, de manera que los valores sean los parámetros recibidos por el método.

```

1. override fun guardarCredenciales(nombre: String, clave: String): Boolean {
2.     var r=true
3.     try {
4.         sharedPreferences.edit()
5.             .putString("usuario", nombre)
6.             .putString("clave", clave)
7.             .commit()
8.     }catch(e:Exception){
9.         r=false
10.    }
11.    return r
12. }
```

Para guardar una pareja clave-valor, lo primero que se hace es obtener un editor, usando el método **edit**. Después, se usan consecutivamente métodos como **putString** para almacenar todas las parejas que se deseen, y finalmente se usa **commit** para que los cambios se hagan efectivos.

- Programa el método **getCredencialesGuardadas** para recuperar la clave y contraseña del usuario, o devolver **null** en caso de que no existan.

```

1. override fun getCredencialesGuardadas(): Credenciales? {
2.     var c = Credenciales(
3.         checkNotNull(sharedPreferences.getString("usuario", "")),
4.         checkNotNull(sharedPreferences.getString("clave", "")))
5.     )
6.     return if(c.usuario.isEmpty()) null else c
7. }
```

El método **getString** permite recuperar el valor de una clave como una cadena de caracteres, y en caso de que no exista, proporcionar un valor alternativo (aquí “”)

- Añade a **GestorCredenciales** una función llamada **getGestorCredenciales**, que devuelva un objeto **GestorCredencialesSharedPreferences**. Para obtenerlo, se necesita pasar como parámetro la **Activity** de la app

```

1. fun getGestorCredenciales(activity:Activity):GestorCredenciales{
2.     val preferencias = activity.getSharedPreferences("firefotos", Context.MODE_PRIVATE)
3.     return GestorCredencialesSharedPreferences(preferencias)
4. }
```

Una **Activity** permite obtener acceso a un objeto **SharedPreferences** con su método **getSharedPreferences**. Como parámetro recibe el nombre que se le da al conjunto de parejas clave-valor al que se desea acceder.

- Abre **LoginFragment** y en el método **hacerLogin** haz que si se inicia correctamente la sesión, se compruebe la casilla **chkRecordar**. En caso de que esté activada, se guardará el nombre de usuario y la contraseña en el sistema de almacenamiento de la app usando el gestor de credenciales configurado

```

1. fun hacerLogin(nombre:String,clave:String){
2.     Sesion.getInstancia().iniciarSesion(
3.         nombre,
4.         clave,
5.         lambdaExito = { sesion ->
6.             if(binding.chkRecordar.isChecked){
7.                 getGestorCredenciales(requireActivity()).guardarCredenciales(nombre,clave)
8.             }
9.             val nc=findNavController()
10.            val flecha = LoginFragmentDirections.actionLoginFragmentToPrincipalFragment()
11.            nc.navigate(flecha)
12.        },
13.        lambdaError = { error -> navegarPantallaError(error)}
14.    )
15. }
```

- Por último, modifica **inicializarInterfaz** de manera que antes de mostrar la capa de login, se compruebe si hay credenciales guardadas, y en dicho caso, recuperarlas y llamar con ellas directamente al método **hacerLogin**

```

1. fun inicializarInterfaz() {
2.     if(Sesion.getInstancia().sesionIniciada()){
3.         activity?.finish()
4.         System.exit(0)
5.     }else{
6.         val credenciales = getGestorCredenciales(requireActivity()).getCredencialesGuardadas()
7.         if (credenciales==null) {
8.             mostrarCapaLogin()
9.         } else {
10.             hacerLogin(credenciales.usuario, credenciales.clave)
11.         }
12.     }
13. }
```

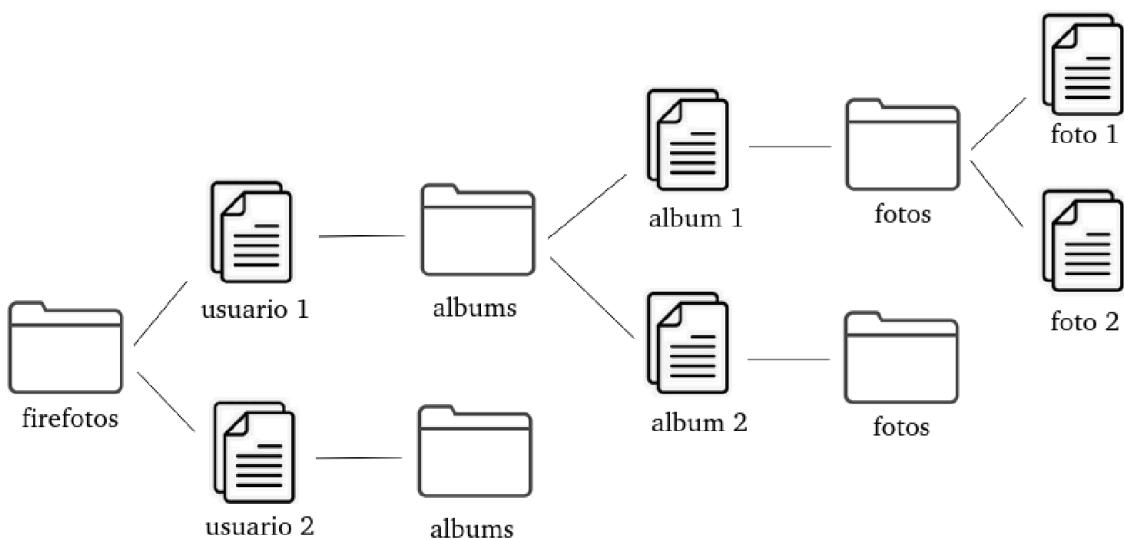
- Ejecuta la app y comprueba que si marcamos la casilla para recordar al usuario, al volver a poner la app ya se accede directamente a **PrincipalFragment**. Comprueba también que durante el proceso de login solo se muestra la pantalla de carga.

## 4 – Firestore

Firestore es una base de datos documental que forma parte de la plataforma Firebase. Los usuarios registrados en Firebase pueden almacenar en ella **documentos** agrupados en **colecciones**, de forma parecida a como los archivos se almacenan en carpetas en un medio de almacenamiento convencional. Las reglas son:

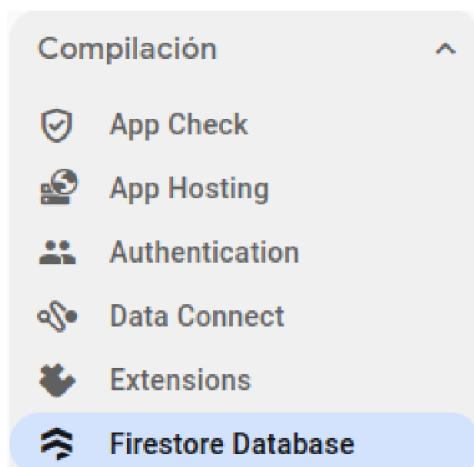
- Una **colección** puede contener **documentos**
- Un **documento** es un conjunto de pares clave-valor
- Un **documento** puede incluir **colecciones** en su interior

Por tanto, Firestore es muy útil para almacenar datos estructurados de forma jerárquica, como ocurre en nuestro proyecto, donde usaremos la siguiente estructura:



Al igual que hemos hecho para el login, usaremos una interfaz para poner en ella todos los accesos a datos que necesita nuestra app, y luego haremos una implementación que use Firestore.

- Entra en la consola de **Firebase** y en el menú la izquierda entra en **Firestore**



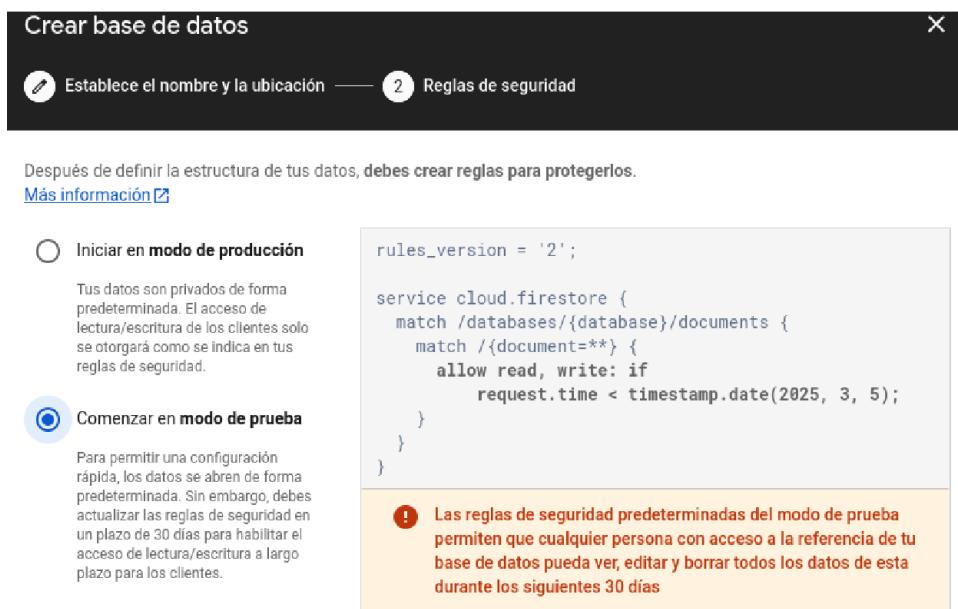
- En la imagen que aparece, pulsa el botón **crear base de datos**



- Se abre una ventana donde puedes elegir la ubicación de la base de datos



- Elige el **modo de prueba** y pulsa el botón de **crear**



- En **modelo**, crea la data class **Album**, que es un álbum que puede contener fotos. El álbum está hecho por un usuario y tiene un título y una descripción.

```
1. data class Album(
2.     val id:String = "", // id del álbum, que asigna Firestore para identificarlo de forma única
3.     val idUsuario:String = "", // id del usuario que crea el album
4.     val titulo:String = "", // título del álbum
5.     val descripcion:String = "" // descripción del album
6. ):Serializable
```

En **Firestore** los documentos se identifican por un id, pero estos se almacenan a nivel interno, y si queremos trabajar con ellos, deberemos almacenarlos en el propio documento. Por ese motivo, **Album** incluye su propio id.

- Crea también la siguiente data class **Foto**, que representa una foto tomada por el usuario. Cada foto está hecha por un usuario, pertenece a un álbum y posee un título, fecha y hora de realización y la url donde se accede a la foto.

```

1. data class Foto (
2.     var id:String = "", // id de la foto, que asigna Firestore para identificarla
3.     var idAlbum:String = "", // id del álbum que contiene la foto
4.     var idUsuario:String = "", // id del usuario que toma la foto y posee el álbum
5.     var titulo:String = "", // título de la foto
6.     var fecha:String = "", // fecha de realización de la foto
7.     var hora: String = "", // hora de realización de la foto
8.     var urlFoto:String = "" // url donde es posible encontrar la foto
9. ): Serializable

```

- Crea una interfaz llamada **FireFotosRepository**, que contiene la definición de los métodos de acceso a datos que va a necesitar nuestra app.

```

1. interface FireFotosRepository {
2.     // métodos para gestionar usuarios (enfoque basado en lambdas)
3.     fun crearUsuario(
4.         u:Usuario,
5.         lambdaExito: () ->Unit,
6.         lambdaError: (String)->Unit
7.     )
8.     fun getUsuario(
9.         id:String,
10.        lambdaExito: (Usuario) -> Unit,
11.        lambdaError: (String) -> Unit
12.    )
13.    // métodos para hacer crud sobre álbumes y fotos (enfoque basado en corrutinas)
14.    suspend fun crearAlbum(u:Usuario,titulo:String,descripcion:String):Album
15.    suspend fun getAlbumes(u:Usuario):List<Album>
16.    suspend fun getFotos(a:Album):List<Foto>
17.    suspend fun crearFoto(
18.        a:Album, titulo:String, fecha: String, hora: String, uriFoto:String):Foto
19.    suspend fun actualizarFoto(f:Foto)
20.    suspend fun borrarFoto(f:Foto)
21. }

```

Para los métodos para gestionar usuarios conviene usar el enfoque de lambdas, ya que se usan en las clases **Sesion** y **FirebaseLogin**, que no están preparadas para lanzar corrutinas. Los métodos para hacer crud (create, read, update, delete) se van a usar en los **view model**, y allí si es posible lanzar corrutinas. Por ese motivo, es mejor plantearlos de esa forma.

- Crea una clase llamada **FirestoreRepository** que implemente **FireFotosRepository** usando **Firestore** como medio de almacenamiento. De momento deja todos sus métodos en **stub** (esqueleto que lanza excepción)

```

1. class FirestoreRepository : FireFotosRepository{
2.     override fun crearUsuario(u: Usuario, lambdaExito:() -> Unit,lambdaError:(String) -> Unit) {
3.         TODO("Not yet implemented")
4.     }
5.     override fun getUsuario(id:String,lambdaExito:(Usuario)-> Unit,lambdaError:(String)->Unit) {
6.         TODO("Not yet implemented")
7.     }
8.     override suspend fun crearAlbum(u: Usuario, titulo: String, descripcion: String): Album {
9.         TODO("Not yet implemented")
10.    }
11.    // resto omitido (es similar)
12. }

```

- Programa en **FirestoreRepository** el método **crearUsuario**, que recibe un objeto **Usuario** y dentro de la colección **firefotos** crea un documento con toda la información de dicho usuario (id, nombre, clave, dirección, ciudad, etc)

```

1. override fun crearUsuario(
2.     u:Usuario,
3.     lambdaExito: () -> Unit,
4.     lambdaError: (String) -> Unit
5. ){
6.     Firebase.firestore // Firebase.firestore es el objeto para trabajar con Firestore
7.         .collection("firefotos") // accedo a la colección llamada "firefotos"
8.         .document(u.id) // creo un nuevo documento cuyo id es "u.id"
9.         .set(u) // pongo en ese documento pares clave-valor para el usuario "u"
10.        .addOnSuccessListener { // código a ejecutar si todo va bien
11.            lambdaExito()
12.        }.addOnFailureListener { excepcion -> // código a ejecutar si se produce un error
13.            lambdaError(excepcion.message.toString())
14.        }
15. }

```

Para trabajar con **Firebase** se usa el objeto **Firebase.firestore**, con estos métodos:

- **collection**: Accede a una colección a partir de su nombre, o la crea si no existe.
- **document**: Crea un documento con el id que pasamos como parámetro. Si no se pasa ningún id, **Firebase** le pone uno automáticamente.

Una vez que estamos en un documento, su método **set** nos permite poner todas las variables de instancia de un objeto en formato clave-valor. Al igual que sucedía con el login, los métodos **addOnSuccessListener** y **addOnFailureListener** permiten definir el código que se ejecuta si todo va bien o si se produce un error.

- Programa en **FirestoreRepository** el método **getUsuario**, que recibe un id y recupera de **Firebase** un objeto **Usuario** con toda la información del usuario que tiene ese id

```

1. override fun getUsuario(
2.     id:String,
3.     lambdaExito: (Usuario) -> Unit,
4.     lambdaError: (String) -> Unit
5. ){
6.     Firebase.firestore
7.         .collection("firefotos") // accedo a la colección "firefotos"
8.         .document(id) // accedo al documento cuyo id es el del usuario que hay que consultar
9.         .get() // obtengo el documento de dicho id
10.        .addOnSuccessListener { resultado -> // resultado es el documento consultado
11.            val u = resultado.toObject(Usuario::class.java) // pasamos el documento a objeto
12.            if(u!=null){
13.                lambdaExito(u) // si todo va bien, ejecutamos lambdaExito
14.            }else{
15.                lambdaError("No se pudo obtener el usuario")
16.            }
17.        }.addOnFailureListener { excepcion ->
18.            lambdaError(excepcion.message.toString())
19.        }
20. }

```

Para realizar consultas, el objeto `Firebase.firestore` tiene los siguientes métodos:

- **collection:** Accede a la colección cuyo nombre se pasa como parámetro
- **document:** Accede al documento cuyo id se pasa como parámetro
- **get:** Recupera los datos del documento

Si todo va bien, se activa el código definido en `addOnSuccessListener`. Dicha lambda recibe como parámetro el resultado de la consulta, que es posible transformar a un objeto con el método `toObject`

- Añade a `FireFotosRepository` una función llamada `getFireFotosRepository` que nos devuelva un `FirestoreRepository`

```
1. interface FireFotosRepository {  
2.     // código omitido  
3. }  
4. fun getFireFotosRepository():FireFotosRepository = FirestoreRepository()
```

Ahora que ya tenemos clases que gestionan los usuarios, podemos terminar las tareas que se quedaron pendientes en la clase `FirebaseLogin`

- Abre `FirebaseLogin` y modifica el método `crearUsuario` para terminar la parte que estaba pendiente, que es grabar la información del usuario que se quiere crear en `Firestore`.

```
1. override fun crearUsuario(  
2.     correo: String,  
3.     clave: String,  
4.     nombre: String,  
5.     direccion: String,  
6.     ciudad: String,  
7.     lambdaExito: (Usuario) -> Unit,  
8.     lambdaError: (String) -> Unit  
9. ) {  
10.     Firebase.auth  
11.         .createUserWithEmailAndPassword(correo,clave)  
12.         .addOnSuccessListener {  
13.             val usuarioFB = checkNotNull(Firebase.auth.currentUser)  
14.             usuarioFB  
15.                 .sendEmailVerification()  
16.                 .addOnSuccessListener {  
17.                     val u = Usuario(usuarioFB.uid,correo,clave,nombre,direccion,ciudad)  
18.                     getFireFotosRepository().crearUsuarid(u,  
19.                         lambdaExito= { lambdaExito(u) },  
20.                         lambdaError = { m -> lambdaError(m) }  
21.                     )  
22.                 }.addOnFailureListener {  
23.                     lambdaError("No se ha podido enviar un correo de verificación")  
24.                 }  
25.             }.addOnFailureListener { excepcion ->  
26.                 lambdaError(excepcion.message.toString())  
27.             }  
28. }
```

El diseño de `crearUsuario` basado en lambdas nos permite ir pasando las lambdas entre los distintos métodos que están siendo llamados

- En **FirebaseLogin** modifica el método **login** para recuperar los datos del usuario que entra al sistema, una vez que se ha comprobado que su cuenta de correo está verificada

```

1. override fun login(
2.     correo: String,
3.     clave: String,
4.     lambdaExito: (Usuario) -> Unit,
5.     lambdaError: (String) -> Unit
6. ) {
7.     try {
8.         Firebase.auth
9.             .signInWithEmailAndPassword(correo, clave)
10.            .addOnSuccessListener {
11.                val usuarioFB = Firebase.auth.currentUser
12.                if (usuarioFB != null) {
13.                    if (usuarioFB.isEmailVerified) {
14.                        getFireFotosRepository().getUsuario(
15.                            usuarioFB.uid,
16.                            lambdaExito = { u -> lambdaExito(u) },
17.                            lambdaError = { m -> lambdaError(m) }
18.                        )
19.                        val u = Usuario("",correo,clave,"","","")
20.                    } else {
21.                        lambdaError("El correo del usuario no está validado")
22.                    }
23.                } else {
24.                    lambdaError("No se pudo identificar al usuario")
25.                }
26.            }.addOnFailureListener { excepcion ->
27.                lambdaError(excepcion.message.toString())
28.            }
29.        }catch(e:Exception){
30.            lambdaError(e.message.toString())
31.        }
32.    }

```

- Ejecuta la app y comprueba que el sistema de creación usuarios funciona correctamente. Crea un usuario y entra en Firebase para ver que dentro de la colección “firefotos” hay un documento con el usuario creado, con todos sus datos. Puedes comprobar también que el usuario es recordado si se marca la casilla correspondiente. Si es necesario, puedes borrar todos los usuarios que previamente hubieran sido creados.

 (default)	 firefotos		 ceD2Dd2LhQVuyBixDwBFjNlxE6i1
<a href="#">+ Iniciar colección</a>	<a href="#">+ Agregar documento</a>	<a href="#">+ Iniciar colección</a>	
firefotos >	ceD2Dd2LhQVuyBixDwB... >	<a href="#">+ Agregar campo</a>	ciudad: "Granada" clave: "1234567" correo: "jdbolivar@ieshlanz.es" dirección: "Calle Roble" id: "ceD2Dd2LhQVuyBixDwBFjNlxE6i1" nombre: "Juan Diego"

A continuación, vamos a añadir una opción a la toolbar en **PrincipalFragment** para que el usuario pueda cerrar sesión

- Pulsa el botón derecho en la carpeta **res** y crea un **android resource file** de tipo **Menu** con el nombre **menu\_toolbar\_fragment\_principal**
- Abre el archivo **menu\_toolbar\_fragment\_principal.xml** y añade un **MenuItem** con id **btnCerrarSesion** y mensaje “Cerrar sesión”

```

1. <menu xmlns:android="http://schemas.android.com/apk/res/android">
2.     <item
3.         android:id="@+id/btnCerrarSesion"
4.         android:title="Cerrar sesión" />
5. </menu>
```

- Abre **PrincipalFragment** y añade en las variables de instancia un objeto que implemente **MenuProvider**, de forma que:
  - En su método **onCreateMenu**, ponga a la toolbar el menú creado anteriormente
  - En su método **onMenuItemSelected** haga que si se pulsa **btnCerrarSesion**, se llame a un método llamado **cerrarSesion**, donde se programarán las acciones encaminadas a cerrar la sesión del usuario

```

1. class PrincipalFragment : Fragment(), NavegadorError {
2.     val menuProvider = object: MenuProvider {
3.         override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
4.             menuInflater.inflate(R.menu.menu_toolbar_fragment_principal, menu)
5.         }
6.         override fun onMenuItemSelected(menuItem: MenuItem): Boolean {
7.             when(menuItem.itemId){
8.                 R.id.btnCerrarSesion -> cerrarSesion()
9.             }
10.            return true
11.        }
12.    }
13.    // resto de la clase omitido
14. }
```

- Programa el método **cerrarSesion** de forma que el usuario activo cierre su sesión, se borren las credenciales que puedan estar almacenadas, se oculte la toolbar y se navegue hacia la pantalla principal.

```

1. fun cerrarSesion(){
2.     Sesion.getInstancia().cerrarSesion()
3.     val mainActivity = activity as MainActivity
4.     mainActivity.binding.materialToolbar.visibility = View.GONE
5.     getGestorCredenciales(requireActivity()).borrarCredenciales()
6.     val nc = findNavController()
7.     nc.navigate(PrincipalFragmentDirections.actionPrincipalFragmentToLoginFragment())
8. }
```

- Añade a **PrincipalFragment** un método para poner el **MenuProvider** en la toolbar y otro para quitarlo

```

1. fun ponerMenuToolbar(){
2.     val mainActivity = activity as MainActivity
3.     mainActivity.addMenuProvider(menuProvider)
4. }
5. fun quitarMenuToolbar(){
6.     val mainActivity = activity as MainActivity
7.     mainActivity.removeMenuProvider(menuProvider)
8. }
```

- Añade un método **inicializarToolbar** que haga visible la **MaterialToolbar** y también llame a **ponerMenuToolbar**

```

1. fun inicializarToolbar(){
2.     val mainActivity = activity as MainActivity
3.     mainActivity.binding.materialToolbar.visibility=View.VISIBLE
4.     ponerMenuToolbar()
5. }
```

- Llama a **inicializarToolbar** en **onCreateView** y a **quitarMenuToolbar** en **onDestroyView**

```

1. override fun onCreateView(
2.     inflater: LayoutInflater, container: ViewGroup?,
3.     savedInstanceState: Bundle?
4. ): View? {
5.     inicializarBinding(inflater, container)
6.     inicializarToolbar()
7.     return binding.root
8. }
9. override fun onDestroyView() {
10.    super.onDestroyView()
11.    quitarMenuToolbar()
12. }
```

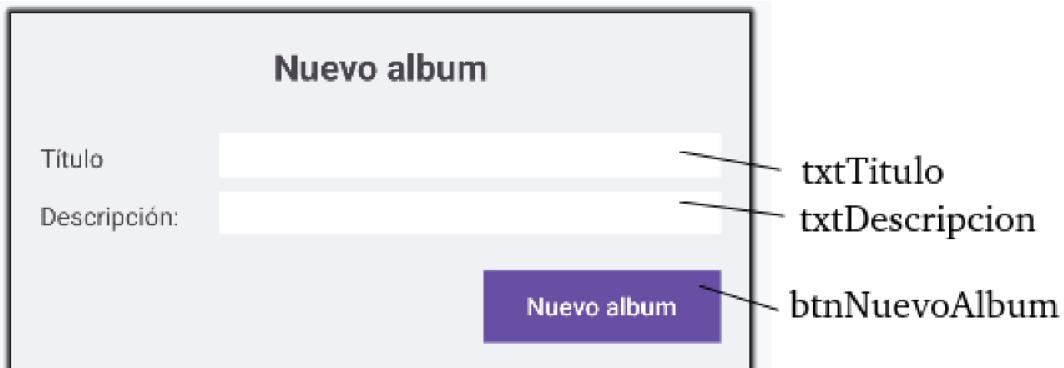
- Ejecuta la app y comprueba que aparece la toolbar y que el usuario ya puede cerrar sesión. Comprueba también que si hay credenciales guardadas, estas se borran y en el siguiente acceso el usuario debe volver a identificarse.

## 5 – DialogFragment

Todos los **Fragment** que hemos programado siempre aparecen ocupando la pantalla entera del dispositivo. Sin embargo, es posible abrir un **Fragment** encima de otro, que queda parcialmente oculto. Esto se conoce como un **DialogFragment**.

Vamos a hacer que la pantalla para crear un nuevo álbum sea un **DialogFragment**

- En el paquete **vista** añade un **Fragment** llamado **NuevoAlbumFragment** (su vista asociada será **fragment\_nuevo\_album.xml**)
- Abre **fragment\_nuevo\_album.xml** y diseña su interfaz así:



```

1. <LinearLayout
2.     xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:app="http://schemas.android.com/apk/res-auto"
4.     xmlns:tools="http://schemas.android.com/tools"
5.     android:layout_width="match_parent"
6.     android:layout_height="wrap_content"
7.     android:minWidth="400dp" android:orientation="vertical" android:padding="16dp">
8.     <TextView
9.         android:layout_width="match_parent" android:layout_height="wrap_content"
10.        android:text="Nuevo album" android:textAlignment="center"
11.        android:textSize="20sp" android:layout_marginBottom="24dp"
12.        android:textStyle="bold" />
13.     <LinearLayout
14.         android:layout_width="match_parent" android:orientation="vertical"
15.         android:layout_height="wrap_content">
16.         <LinearLayout
17.             android:layout_width="match_parent" android:layout_height="wrap_content"
18.             android:layout_marginBottom="8dp" android:orientation="horizontal">
19.                 <TextView
20.                     android:layout_width="100dp" android:layout_height="wrap_content"
21.                     android:text="@string/t_tulo"/>
22.                 <EditText
23.                     android:layout_width="match_parent" android:layout_height="wrap_content"
24.                     android:background="@color/white"
25.                     android:id="@+id/txtTitulo"/>
26.             </LinearLayout>
27.             <LinearLayout
28.                 android:layout_width="match_parent" android:layout_height="wrap_content"
29.                 android:orientation="horizontal">
30.                 <TextView
31.                     android:layout_width="100dp" android:layout_height="wrap_content"
32.                     android:text="@string/descripcion"/>
33.                 <EditText
34.                     android:layout_width="match_parent" android:layout_height="wrap_content"
35.                     android:background="@color/white" android:inputType="textMultiLine"
36.                     android:id="@+id/txtDescripcion"/>
37.             </LinearLayout>
38.         </LinearLayout>
39.         <Button
40.             android:layout_height="wrap_content" android:layout_width="wrap_content"
41.             android:text="@string/nuevo_album" app:cornerRadius="0dp"
42.             android:layout_marginTop="16dp" android:layout_gravity="right"
43.             android:id="@+id/btnNuevoAlbum"/>
44.     </LinearLayout>

```

Es muy importante poner a **wrap\_content** la altura del **Fragment**, para que así pueda ocupar solo una porción de la pantalla y ponerse encima del **Fragment** que lo abre.

- Abre **NuevoAlbumFragment**, habilita el view binding, haz que herede de **DialogFragment** y que su constructor reciba una expresión lambda que se lance cuando se pulse **btnNuevoAlbum**. Más adelante, verás que dicha lambda recibirá como parámetros el título y descripción escritos por el usuario.

```

1. class NuevoAlbumFragment(val lambda: (String,String) -> Unit) : DialogFragment() {
2.     lateinit var binding:FragmentNuevoAlbumBinding
3.     override fun onCreateView(
4.         inflater: LayoutInflater, container: ViewGroup?,
5.         savedInstanceState: Bundle?
6.     ): View? {
7.         inicializarBinding(inflater,container)
8.         return binding.root
9.     }
10.    private fun inicializarBinding(inflater: LayoutInflater, container: ViewGroup?) {
11.        binding = FragmentNuevoAlbumBinding.inflate(inflater,container,false)
12.    }
13. }

```

- Añade a **NuevoAlbumFragment** un método llamado **inicializarBoton**, que será llamado en **onCreateView**. En **inicializarBoton** simplemente llamaremos a la lambda que hay en el constructor pasando como parámetro el título y descripción escritos por el usuario, y después se cerrará el **Fragment**

```

1. private fun inicializarBoton(){
2.     binding.btnNuevoAlbum.setOnClickListener {
3.         lambda(
4.             binding.txtTitulo.text.toString(),
5.             binding.txtDescripcion.text.toString()
6.         )
7.         dismiss()
8.     }
9. }
```

El método **dismiss** hace que se cierre el **DialogFragment**

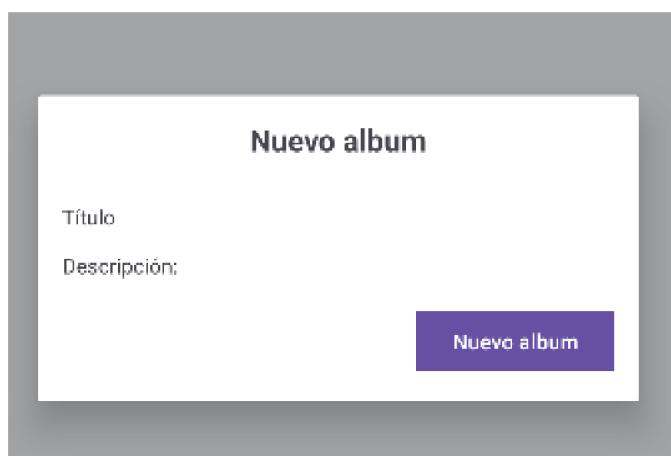
- En **PrincipalFragment** haz un método llamado **inicializarBoton**, que llame en **onCreateView**. En **inicializarBoton** crea un **NuevoAlbumFragment** pasando como parámetro una lambda que procese el título y la descripción del álbum, y las muestre en Logcat. El objeto **NuevoAlbumFragment** (que recordemos, es un **DialogFragment**) se mostrará en pantalla con su método **show**

```

1. fun configurarBoton(){
2.     binding.btnNuevoAlbum.setOnClickListener{
3.         NuevoAlbumFragment { titulo, descripcion ->
4.             Log.d("creando_album","Título=$titulo Descripción=$descripcion")
5.             }.show(childFragmentManager, "Nuevo album")
6.     }
7. }
```

El método **show** de los **DialogFragment** hace que se muestren por encima del **Fragment** que los crea. Dicho método necesita un objeto llamado **childFragmentManager** y un nombre que lo identifica

- Ejecuta la app y comprueba que cuando pulsas el botón para crear un nuevo álbum, aparece **NuevoAlbumFragment** por encima del **PrincipalFragment** y que al pulsar su botón de aceptar, en Logcat aparece el título y descripción que han sido introducidos en ella.



- Ahora abre **FirestoreRepository** y programa el método **crearAlbum** de forma que se grabe en **Firestore** el objeto **Album** que se pasa como parámetro.

```

1. override suspend fun crearAlbum(usuario: Usuario,titulo: String,descripcion: String):Album {
2.     val nuevoDocumento = Firestore.firestore
3.         .collection("firefotos") // accedo a la colección "firefotos"
4.         .document(usuario.id) // accedo al documento cuyo id es el del usuario
5.         .collection("albums") // obtengo la colección "albums" del usuario
6.         .document() // creo un documento con un id, pero sin datos
7.         // como el documento creado tiene un id, puedo crear el objeto Album con dicho id
8.         val album = Album(nuevoDocumento.id,usuario.id,titulo,descripcion)
9.         nuevoDocumento
10.            .set(album) // pongo al documento creado los datos del album
11.            .await() // punto de suspensión para esperar la finalización de la tarea
12.        return album
13.    }

```

Para crear el **Album** es necesario el id que **Firestore** le pone. Por ese motivo, primero se obtiene el **documento** que se va a crear, que estará vacío pero ya tendrá un id. En ese momento, se crea el **Album** y después, se inserta dicho **Album** en el documento creado. En esta ocasión se usa **await**, que es un punto de suspensión que hace que el hilo que ejecuta la corriente quede libre hasta que se haya creado el álbum.

- En el paquete **viewmodel** crea una clase **PrincipalFragmentViewModel**, que herede de **ViewModel** y añádele como variable de instancia una lista mutable que guardará la lista de albums del usuario, que inicialmente estará vacía.

```

1. class PrincipalFragmentViewModel : ViewModel() {
2.     var listaAlbums:MutableList<Album> = mutableListOf()
3. }

```

- En **PrincipalFragment** crea una variable de instancia de tipo **PrincipalFragmentViewModel** y un método **inicializarViewModel** para inicializarla. Ese método será llamado en **onCreateView**

```

1. class PrincipalFragment : Fragment(),NavegadorError {
2.     // resto de la clase omitido
3.     lateinit var viewModel: PrincipalFragmentViewModel
4.     override fun onCreateView(
5.         inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
6.     ): View? { inicializarBinding(inflater, container)
7.         inicializarViewModel()
8.         inicializarToolbar()
9.         return binding.root
10.    }
11.    fun inicializarViewModel(){
12.        viewModel = ViewModelProvider(this).get(PrincipalFragmentViewModel::class.java)
13.    }
14. }

```

- En **PrincipalFragmentViewModel** añade un método llamado **crearAlbum**, que reciba como parámetros el título de un álbum, su descripción, y dos expresiones **lambdaExito** (que procesará el objeto **Album** creado) y **lambdaError** (que procesará el mensaje de error). Dicho método usará **FireFotosRepository** para crear un álbum y lanzar las lambda adecuadas. Si todo va bien, el **Album** creado se guardará en la lista de albums del usuario

```

1. class PrincipalFragmentViewModel : ViewModel() {
2.     fun crearAlbum(
3.         titulo:String,
4.         descripcion:String,
5.         lambdaExito:(Album) -> Unit,
6.         lambdaError:(String) -> Unit
7.     ){
8.         viewModelScope.launch { // lanza una corrutina en un view model
9.             try {
10.                 val album:Album = getFireFotosRepository().crearAlbum(
11.                     Sesion.getInstancia().usuario,
12.                     titulo,
13.                     descripcion,
14.                     )
15.                 listaAlbums.add(album)
16.                 lambdaExito(album)
17.             }catch(e:Exception){
18.                 lambdaError(e.message.toString())
19.             }
20.         }
21.     }
22. }
23. }
```

Para lanzar una corrutina dentro de un **viewmodel** se usa el objeto **viewModelScope**

- Abre **PrincipalFragment** y en el método **configurarBoton**, retira el mensaje de Logcat y en su lugar, pon una llamada al método **crearAlbum** del **viewModel**, de forma que si todo va bien se llame a un método **actualizarRecyclerView** y si se produce un error se navegue a la pantalla de error

```

1. fun configurarBoton(){
2.     binding.btnNuevoAlbum.setOnClickListener{
3.         NuevoAlbumFragment { titulo, descripcion ->
4.             viewModel.crearAlbum(
5.                 titulo,
6.                 descripcion,
7.                 lambdaExito = { a -> actualizarRecyclerView()},
8.                 lambdaError = { m -> navegarPantallaError(m)}
9.             )
10.        }.show(childFragmentManager, "Nuevo album")
11.    }
12. }
13. fun actualizarRecyclerView(){
14.     // aquí recargaremos los álbumes
15. }
```

A continuación, vamos a hacer toda la parte para hacer un **RecyclerView** que muestre la colección de álbums del usuario que ha iniciado sesión

- Pulsa el botón derecho sobre **res/layout**, crea un **layout resource file** llamado **album.xml** y diseña en él la siguiente interfaz:



```

1. <androidx.cardview.widget.CardView xmlns:app="http://schemas.android.com/apk/res-auto"
2.     xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:layout_height="wrap_content"
4.     android:layout_width="match_parent"
5.     app:cardCornerRadius="24dp"
6.     app:cardElevation="4dp"
7.     android:layout_margin="8dp"
8.     xmlns:tools="http://schemas.android.com/tools">
9.     <LinearLayout
10.         android:layout_width="match_parent"
11.         android:layout_height="match_parent"
12.         android:orientation="vertical">
13.         <ImageView
14.             android:id="@+id/imgFoto"
15.             tools:srcCompat="@tools:sample/backgrounds/scenic"
16.             android:layout_width="match_parent"
17.             android:layout_marginTop="8dp"
18.             android:layout_height="140dp"/>
19.         <TextView
20.             android:id="@+id/txtTituloAlbum"
21.             android:layout_width="match_parent"
22.             android:layout_height="30sp"
23.             android:gravity="center"
24.             tools:text="Nombre del album"/>
25.     </LinearLayout>
26. </androidx.cardview.widget.CardView>
```

- Copia a la carpeta **drawable** el archivo **sin\_imagen.jpg** adjunto al proyecto.
- En **Funciones.kt**, crea una nueva **extension function** para la clase **ImageView** llamada **ponerFotoAlbum**. Esta función recibirá un objeto **Album** y, de momento, le pondrá la imagen anterior. Más adelante, esta función pondrá una foto aleatoria del album, y como la obtención de las fotos tardará tiempo en completarse, haremos que **ponerFotoAlbum** sea una función de suspensión.

```

1. suspend fun ImageView.ponerFotoAlbum(a:Album) {
2.     // PROVISIONAL: ponemos la imagen "sin_imagen.jpg" por defecto al album
3.     setImageResource(R.drawable.sin_imagen)
4. }
```

- Crea una clase llamada **AlbumHolder** que herede de **RecyclerView.ViewHolder** y añádele una variable de instancia de tipo **Album**

```

1. class AlbumHolder(val binding:AlbumBinding) : RecyclerView.ViewHolder(binding.root){
2.     lateinit var album: Album
3. }
```

- Añade a **AlbumHolder** un método llamado **mostrarAlbum**, que reciba un objeto **Album**, lo ponga en la variable de instancia, y rellene los campos de la pantalla con su información

```

1. class AlbumHolder(val binding:AlbumBinding) : RecyclerView.ViewHolder(binding.root)ope {
2.     lateinit var album: Album
3.     fun mostrarAlbum(a:Album){
4.         album = a
5.         binding.txtTituloAlbum.text = a.titulo
6.         binding.imgFoto.ponerFotoAlbum(a)    // no compila
7.     }
8. }
```

La llamada a **ponerFotoAlbum** no compila porque es un punto de suspensión

- Haz que **AlbumHolder** implemente la interfaz **CoroutineScope** y sobreescribe su propiedad **coroutineContext** para que devuelva **Dispatchers.Main**. Con esto, ya puedes llamar a **ponerFotoAlbum** en una corutina.

```

1. class AlbumHolder(
2.     val binding:AlbumBinding
3. ) : RecyclerView.ViewHolder(binding.root), CoroutineScope {
4.     lateinit var album: Album
5.     fun mostrarAlbum(a:Album){
6.         album = a
7.         binding.txtTituloAlbum.text = a.titulo
8.         launch {
9.             binding.imgFoto.ponerFotoAlbum(a)
10.        }
11.    }
12. }
13. override val coroutineContext: CoroutineContext
14.     get() = Dispatchers.Main
15. }
```

Para que una clase cualquiera tenga la habilidad de lanzar corrutinas, debe implementar la interfaz **CoroutineScope** y sobreescribir su propiedad **coroutineContext** para devolver el “almacén de hilos” que ejecutará la corrutina:

- **Dispatchers.Default** → almacén que usa tantos hilos como núcleos tiene el procesador, y es útil para realizar en segundo plano tareas intensivas
- **Dispatchers.IO** → almacén que usa infinitos hilos, y es útil para realizar tareas de entrada y salida, las cuales están casi siempre en espera
- **Dispatchers.Main** → almacén que solo tiene un hilo, que es el que dibuja la interfaz de usuario. Es el que usamos aquí, ya que la clase **AlbumHolder** es usada por dicho hilo en nuestra app.

- En el paquete **vista** crea una clase llamada **AlbumAdapter**, que herede de **RecyclerView.Adapter<AlbumHolder>**. Esta clase tendrá como propiedades una lista de albums y un lambda que procesa un **AlbumHolder**. Los métodos abstractos de esta clase se programarán de la forma habitual

```

1. class AlbumAdapter(
2.     val albumes:List<Album>,
3.     val lambda:(AlbumHolder) -> Unit
4. ) : RecyclerView.Adapter<AlbumHolder>() {
5.     // devuelve un objeto AlbumHolder
6.     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): AlbumHolder {
7.         val inflater = LayoutInflater.from(parent.context)
8.         val binding = AlbumBinding.inflate(inflater,parent,false)
9.         return AlbumHolder(binding)
10.    }
11.    // devuelve el número de albums
12.    override fun getItemCount(): Int = albumes.size
13.    // pone un objeto Album en un AlbumHolder y hace que al pulsarlo se ejecute una lambda
14.    override fun onBindViewHolder(holder: AlbumHolder, position: Int) {
15.        val album = albumes.get(position)
16.        holder.mostrarAlbum(album)
17.        holder.binding.root.setOnClickListener{
18.            lambda(holder)
19.        }
20.    }
21. }
```

- Abre **PrincipalFragment** y programa el método **actualizarRecyclerView** para que se ponga a **lstAlbums** la lista de albums que hay en el **viewModel**. Además, se hará que al pulsar un **AlbumHolder** se llame a un método **navegarPantallaAlbum** pasando el objeto **Album** que hay en el **AlbumHolder**

```

1. fun actualizarRecyclerView(){
2.     binding.lstAlbum.adapter = AlbumAdapter(viewModel.listaAlbums){ holder ->
3.         navegarPantallaAlbum(holder.album)
4.     }
5. }
6. fun navegarPantallaAlbum(a: Album){
7.     // aquí navegaremos a la pantalla que nos mostrará las fotos del álbum
8. }
```

- Ejecuta la app y crea varios álbumes de fotos. Comprueba que los álbumes creados se añaden al **RecyclerView**, pero si reinicias la app ya no se ven. Esto se debe a que **PrincipalFragment** no carga en su inicio la lista de álbumes del usuario.
- Abre **FirestoreRepository** y programa el método **getAlbumes**, de manera que se recupere de **Firebase** la lista de álbumes del usuario

```

1. override suspend fun getAlbumes(u: Usuario): List<Album> =
2.     Firebase.firestore
3.         .collection("firefotos") // accedo a la colección "firefotos"
4.         .document(u.id) // accedo al documento cuyo id es el del usuario
5.         .collection("albums") // accedo a su colección "albums"
6.         .get() // obtengo la colección de albums
7.         .await() // punto de suspensión para esperar la obtención de la lista de álbumes
8.         .toObjects(Album::class.java) // convierte los documentos recuperados en objetos Album
```

- Vete a **PrincipalFragmentViewModel** y añade un método llamado **cargarListaAlbumes** que rellene la lista de álbumes que hay en sus variables de instancia con los álbumes que tiene el usuario que ha iniciado sesión. Como es habitual, el método recibirá lambdas para ejecutar en caso de éxito o de error

```

1. fun cargarListaAlbumes(
2.     lambdaExito:()->Unit,
3.     lambdaError:(String)->Unit
4. ){
5.     viewModelScope.launch {
6.         try{
7.             val usuario = Sesion.getInstancia().usuario
8.             listaAlbums = getFireFotosRepository().getAlbumes(usuario).toMutableList()
9.             lambdaExito()
10.        }catch(e:Exception){
11.            lambdaError(e.message.toString())
12.        }
13.    }
14. }
```

- Abre **PrincipalFragment** y actualiza el método **inicializarViewModel** para que cargue la lista de álbumes. En caso de que vaya bien, se actualizará el **RecyclerView** y en caso de error, se navegará a la pantalla de error.

```

1. fun inicializarViewModel(){
2.     viewModel = ViewModelProvider(this).get(PrincipalFragmentViewModel::class.java)
3.     viewModel.cargarListaAlbumes()
4.     lambdaExito = { actualizarRecyclerView() },
5.     lambdaError = { m -> navegarPantallaError(m) }
6. }
7. }
```

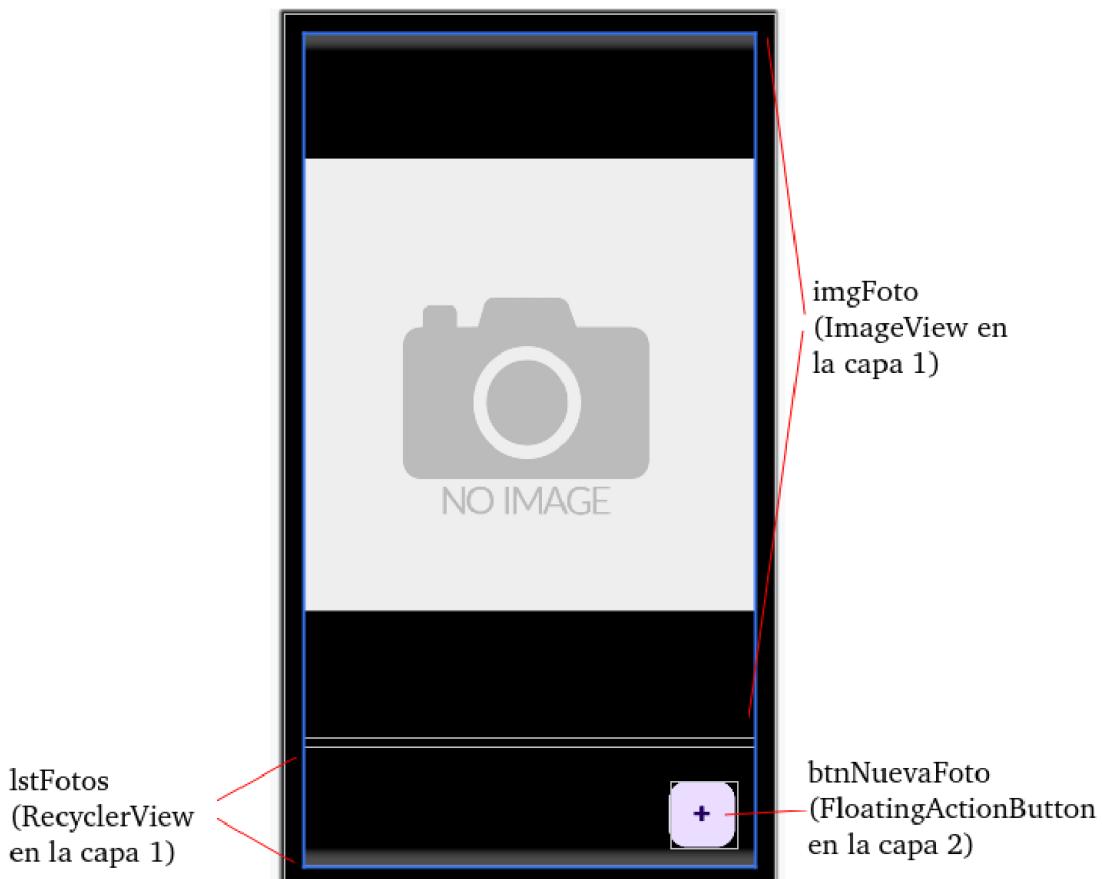
- Ejecuta la app y comprueba que ya se ven los álbums del usuario que inicia sesión, junto con los nuevos que va creando.

## ***6 – RecyclerView con carousel***

A continuación, vamos a empezar la programación de la pantalla **AlbumFragment**, que será un visor de las fotos del album seleccionado en **PrincipalFragment**

Las fotos se mostrarán en **RecyclerView** que usará un **CarouselLayoutManager**. Esto significa que las vistas que muestra el **RecyclerView** se dispondrán de izquierda a derecha y será posible desplazarse por ellas de manera horizontal.

- En el paquete **vista** crea un nuevo **Fragment** llamado **AlbumFragment**
- Abre el archivo **fragment\_album.xml** y dale el siguiente diseño por capas:



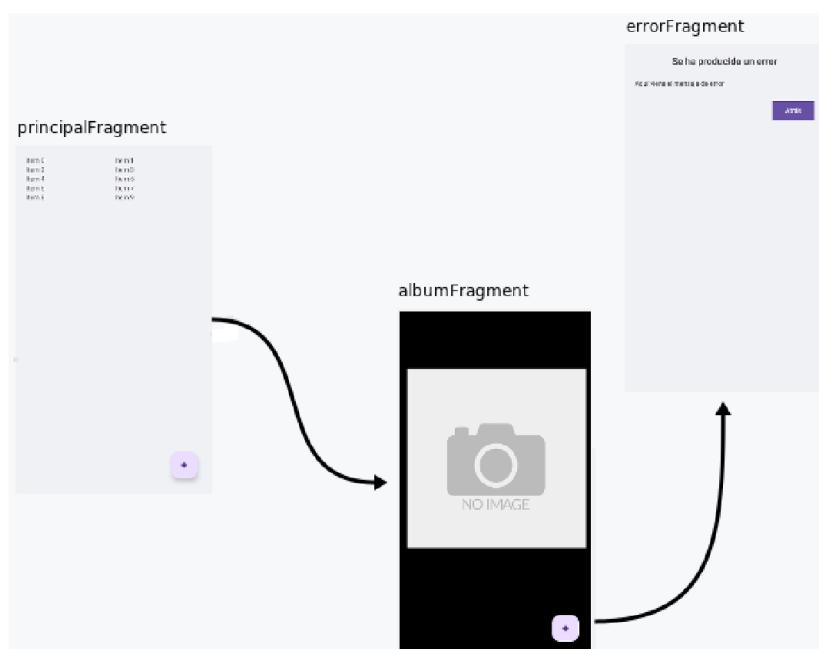
```

1. <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     xmlns:tools="http://schemas.android.com/tools"
3.     android:layout_width="match_parent" android:layout_height="match_parent"
4.     android:padding="16dp" android:background="@color/black"
5.     xmlns:app="http://schemas.android.com/apk/res-auto">
6.     <!-- CAPA 1 : RECYCLER VIEW -->
7.     <LinearLayout
8.         android:layout_width="match_parent" android:layout_height="match_parent"
9.         android:orientation="vertical">
10.        <ImageView
11.            android:layout_width="match_parent" android:layout_height="0dp"
12.            android:layout_weight="1" android:src="@drawable/sin_imagen"
13.            android:id="@+id/imgFoto"/>
14.        <androidx.recyclerview.widget.RecyclerView
15.            android:layout_width="match_parent"
16.            android:layout_height="100dp"
17.            app:layoutManager="com.google.android.material.carousel.CarouselLayoutManager"
18.            android:clipChildren="false"
19.            android:clipToPadding="false"
20.            android:layout_marginTop="8dp"
21.            android:id="@+id/lstFotos"/>
22.    </LinearLayout>
23.    <!-- CAPA 2 : FLOATING ACTION BUTTON -->
24.    <LinearLayout
25.        android:layout_width="match_parent" android:layout_height="match_parent"
26.        android:gravity="bottom|end" android:orientation="vertical">
27.        <com.google.android.material.floatingactionbutton.FloatingActionButton
28.            android:id="@+id/btnNuevaFoto"
29.            android:layout_width="wrap_content" android:layout_height="wrap_content"
30.            android:layout_margin="16dp" android:src="@android:drawable/ic_input_add" />
31.    </LinearLayout>
32. </FrameLayout>

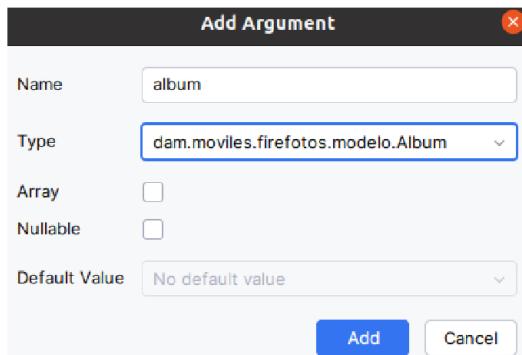
```

Para poner el **RecyclerView** en modo carousel, le asociamos en su atributo **layoutManager** un **CarouselLayoutManager** y configuramos a false sus atributos **clipChildren** y **clipToPadding** para que las imágenes que sobresalen puedan verse parcialmente.

- Abre el archivo **nav\_graph.xml** y agrega el **AlbumFragment**, haciendo que sea posible navegar a él desde **PrincipalFragment**, y que desde **AlbumFragment** se pueda navegar a la pantalla de error



- Añade a **albumFragment** un argumento llamado **album** de tipo **Custom Serializable**, y en el desplegable, elige **Album**



- En el paquete **viewmodel** crea una clase llamada **AlbumFragmentViewModel**, que herede de **ViewModel** y tenga como variables de instancia el **Album**, la foto que se mostrada en el centro, y una lista mutable con sus fotos.

```

1. class AlbumFragmentViewModel : ViewModel() {
2.     lateinit var album: Album
3.     var fotoVisible:Foto? = null // inicialmente no se muestra ninguna foto en el centro
4.     var fotos:MutableList<Foto> = mutableListOf()
5. }
```

- Abre el código de **AlbumFragment**, borra el código obsoleto, habilita el **view binding**, implementa **NavegadorError** e inicializa el **viewModel**, rellenando el álbum con el parámetro recibido de la pantalla previa.

```

1. class AlbumFragment : Fragment(),NavegadorError{
2.     lateinit var binding:FragmentAlbumBinding
3.     lateinit var viewModel:AlbumFragmentViewModel
4.     override fun onCreateView(
5.         inflater: LayoutInflater, container: ViewGroup?,
6.         savedInstanceState: Bundle?
7.     ): View? {
8.         inicializarBinding(inflater,container)
9.         inicializarViewModel()
10.        return binding.root
11.    }
12.    private fun inicializarBinding(inflater: LayoutInflater, container: ViewGroup?) {
13.        binding = FragmentAlbumBinding.inflate(inflater,container,false)
14.    }
15.    private fun inicializarViewModel() {
16.        viewModel = ViewModelProvider(this).get(AlbumFragmentViewModel::class.java)
17.        viewModel.album = AlbumFragmentArgs.fromBundle(requireArguments()).album
18.    }
19.    override fun getNavController(): NavController = findNavController()
20.    override fun getFlechaNavegacionPantallaError(m: String): NavDirections =
21.        AlbumFragmentDirections.actionAlbumFragmentToErrorFragment(m)
22. }
```

- Abre **PrincipalFragment** y programa el método **navegarPantallaAlbum** para que se navegue hacia **AlbumFragment**

```

1. fun navegarPantallaAlbum(a: Album){
2.     findNavController().navigate(
3.         PrincipalFragmentDirections.actionPrincipalFragmentToAlbumFragment(a)
4.     )
5. }
```

- Ejecuta la app y comprueba que al pulsar un álbum, se navega hacia **AlbumFragment**

## 7 – Iconos SVG

Ahora vamos a poner a la toolbar de **AlbumFragment** un menú que tenga opciones para modificar y borrar la foto que esté siendo mostrada.

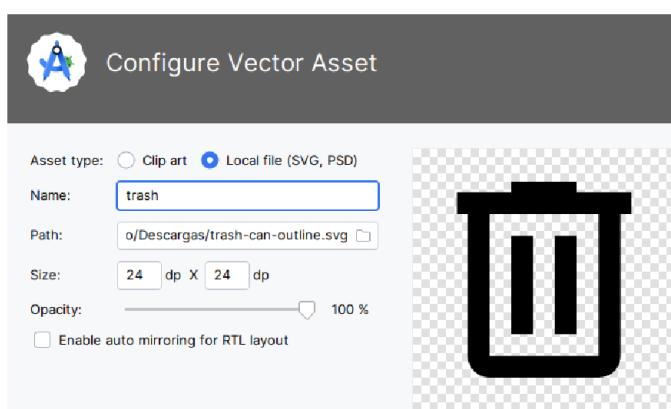
Los iconos que usaremos serán de tipo **svg**, que en lugar de una imagen, contienen órdenes para que el ordenador pinte el icono. Su ventaja es que pueden ser escalados a cualquier tamaño sin pérdida de calidad.

Android Studio no permite añadir directamente imágenes **svg**, sino que deben ser convertidas a un **drawable xml**. Afortunadamente, Android Studio integra dicha herramienta y el proceso de conversión es muy sencillo.

- Entra en la web <https://pictogrammers.com/>
- En el buscador, escribe **trash** y elige el icono **trash-can-outline**
- Descarga el icono a tu ordenador

Para convertir el **svg** en un **drawable xml** usaremos la herramienta **Asset Studio**

- Pulsa el botón derecho en la carpeta **res** y después **new → Vector Asset**.
- En la ventana que aparece, rellena estos datos:
  - Asset type: **local file (svg,psd)**
  - Name (*es el nombre que pondremos a nuestro drawable*): **trash**
  - Path: la ruta del archivo **trash-can-outline.svg**



- Pulsa el botón **Next** y en la siguiente pantalla confirma que el archivo convertido **trash.xml** se incluirá en la carpeta **drawable** y pulsa el botón de finalizar.
- Repite el mismo proceso con un ícono llamado **pencil-box-multiple.svg**, conviértelo en **pencil.xml** y guárdalo en la carpeta **drawable**
- Pulsa el botón derecho en **res/menu** y crea un **menu resource file** llamado **menu\_toolbar\_fragment\_album**

- Abre el archivo **menu\_toolbar\_fragment\_album.xml** y añade dos **Item** para actualizar y borrar fotos. Ambos tendrán los iconos **pencil.xml** y **trash.xml** respectivamente, y sus id serán **btnActualizarFoto** y **btnBorrarFoto**.

```

1. <menu xmlns:app="http://schemas.android.com/apk/res-auto"
2.       xmlns:android="http://schemas.android.com/apk/res/android">
3.     <item
4.         android:id="@+id(btnActualizarFoto"
5.         android:icon="@drawable/pencil"
6.         android:title="Item"
7.         app:showAsAction="always" />
8.     <item
9.         android:id="@+id(btnBorrarFoto"
10.        android:icon="@drawable/trash"
11.        android:title="Item"
12.        app:showAsAction="always" />
13. </menu>
```

- Abre **AlbumFragment** y añade a sus variables de instancia un objeto **MenuProvider** que añada a la toolbar el menú diseñado anteriormente, y haga que al pulsar los botones de actualizar y borrar, se llamen a métodos **borrarFoto** y **navegarPantallaDetalleFoto**, pasándole a este último como parámetro el objeto **fotoVisible** del **viewModel**

```

1. class AlbumFragment : Fragment(),NavegadorError{
2.     // resto de la clase omitido
3.     val menuProvider:MenuProvider = object: MenuProvider{
4.         override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
5.             menuInflater.inflate(R.menu.menu_toolbar_fragment_album,menu)
6.         }
7.         override fun onMenuItemSelected(menuItem: MenuItem): Boolean {
8.             when(menuItem.itemId){
9.                 R.id.btnBorrarFoto -> borrarFoto()
10.                R.id.btnActualizarFoto -> navegarPantallaDetalleFoto(viewModel.fotoVisible)
11.            }
12.            return true
13.        }
14.    }
15.    fun borrarFoto(){
16.        // aquí borraremos la foto que se ve en el centro de la pantalla
17.    }
18.    fun navegarPantallaDetalleFoto(f:Foto?){
19.        // aquí navegaremos a la pantalla para mostrar el detalle de una foto, o crear una nueva
20.    }
21. }
```

- En **AlbumFragment** pon métodos **ponerMenuToolbar** y **quitarMenuToolbar**.

```

1. class AlbumFragment : Fragment(),NavegadorError{ 
2.     // resto omitido
3.     fun ponerMenuToolbar(){
4.         val mainActivity = activity as MainActivity
5.         mainActivity.addMenuProvider(menuProvider)
6.     }
7.     fun quitarMenuToolbar(){
8.         val mainActivity = activity as MainActivity
9.         mainActivity.removeMenuProvider(menuProvider)
10.    }
11. }
```

- Llama a **ponerMenuToolbar** en **onCreateView** y a **quitarMenuToolbar** en **onPause**

```

1. class AlbumFragment : Fragment(), NavegadorError{
2.     // resto omitido
3.     override fun onCreateView(
4.         inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
5.     ): View? {
6.         inicializarBinding(inflater, container)
7.         inicializarViewModel()
8.         ponerMenuToolbar()
9.         return binding.root
10.    }
11.    override fun onPause() {
12.        super.onPause()
13.        quitarMenuToolbar()
14.    }
15. }

```

El método **onPause** es llamado inmediatamente después de que **AlbumFragment** deje de ser visible y es un buen lugar para quitar el menú de la toolbar, ya que el siguiente **Fragment** de la navegación no ha sido cargado

- Ejecuta la app y comprueba que al entrar en **AlbumFragment** se muestra el menú de opciones de la toolbar y que al volver hacia atrás, desaparece ese menú y se pone a la toolbar el menú de cerrar sesión.

Observa que los iconos se dibujan de color negro, y ese color no pega con la toolbar. Afortunadamente, se puede cambiar porque los iconos no son imágenes, sino instrucciones de dibujado y una de ellas define el color con el que se pinta.

- En la carpeta **drawable** abre el archivo **trash.xml** y cambia el atributo **android:fillColor** por **#FFFFFF**

```

1. <vector xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:width="24dp" android:height="24dp"
3.     android:viewportWidth="24" android:viewportHeight="24">
4.     <path
5.         android:fillColor="#FFFFFF"
6.         android:pathData="M9,3V4H4V6H5V19A2,2 0,0 0,7 21H17A2,..... 8V17H15V8H13Z"/>
7. </vector>

```

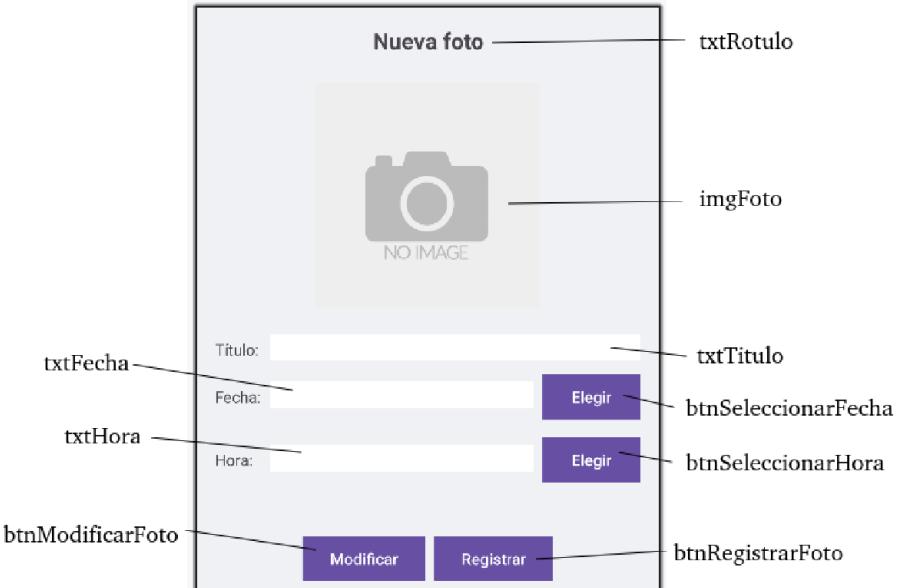
- Repite lo mismo con el archivo **pencil.xml**
- Ejecuta la app y comprueba que los iconos ahora se pintan en blanco

## **8 – Acceso a la cámara**

Ahora vamos a diseñar una pantalla llamada **FotoFragment**, que tendrá un doble comportamiento (es un formulario adaptable). Por un lado, permitirá tomar una foto usando la cámara del dispositivo, y por otro, permitirá modificar una foto que ya exista.

Para saber cómo debe actuar, **FotoFragment** recibirá como parámetro un objeto **Foto?**, que cuando sea **null** permitirá registrar una nueva foto, y si no, editarla.

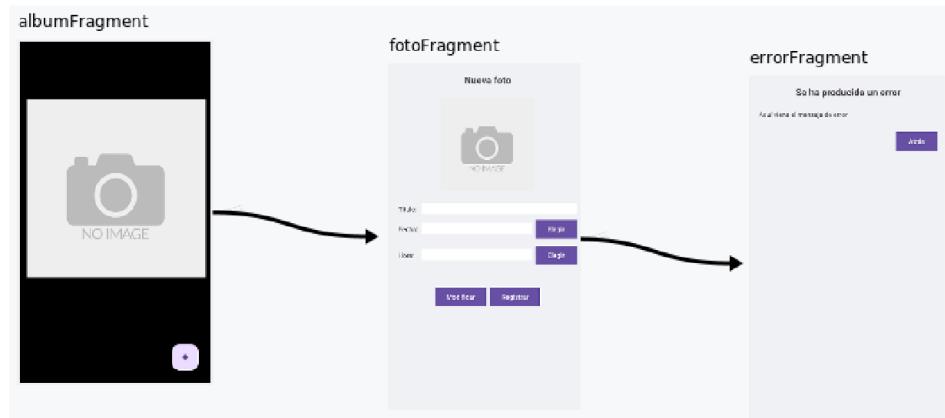
- En el paquete **vista** crea un nuevo **Fragment** llamado **FotoFragment**
- Abre el archivo **fragment\_foto.xml** y diseña la siguiente interfaz:



```

1. <?xml version="1.0" encoding="utf-8"?>
2. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:tools="http://schemas.android.com/tools"
4.     xmlns:app="http://schemas.android.com/apk/res-auto"
5.     android:layout_width="match_parent" android:orientation="vertical"
6.     android:padding="16dp" android:layout_height="match_parent">
7.     <TextView android:id="@+id/txtRotulo"
8.         android:layout_width="match_parent" android:layout_height="wrap_content"
9.         android:text="Nueva foto" android:textAlignment="center"
10.        android:textSize="20sp" android:textStyle="bold" />
11.     <ImageView android:id="@+id/imgFoto"
12.         android:layout_width="200dp" android:layout_height="200dp"
13.         android:layout_gravity="center" android:layout_marginTop="24dp"
14.         android:scaleType="fitXY" android:src="@drawable/sin_imagen" />
15.     <TableLayout android:layout_width="match_parent" android:layout_marginTop="24dp"
16.         android:layout_height="wrap_content" android:stretchColumns="1">
17.         <TableRow>
18.             <TextView android:text="@string/titulo"/>
19.             <EditText android:layout_span="2" android:id="@+id/txtTitulo"/>
20.         </TableRow>
21.         <TableRow android:layout_marginTop="8dp">
22.             <TextView android:layout_marginRight="8dp" android:text="@string/fecha"/>
23.             <EditText android:editable="false" android:id="@+id/txtFecha"/>
24.             <Button android:id="@+id(btnSeleccionarFecha" app:cornerRadius="0dp"
25.                 android:layout_width="wrap_content" android:layout_height="wrap_content"
26.                 android:layout_marginLeft="8dp" android:text="@string/elegir"/>
27.         </TableRow>
28.         <TableRow android:layout_marginTop="8dp">
29.             <TextView android:text="@string/hora" />
30.             <EditText android:editable="false" android:id="@+id/txtHora"/>
31.             <Button android:id="@+id(btnSeleccionarHora"
32.                 android:layout_width="wrap_content" android:layout_height="wrap_content"
33.                 android:layout_marginLeft="8dp" android:text="@string/elegir"
34.                 app:cornerRadius="0dp"/>
35.         </TableRow>
36.     </TableLayout>
37.     <LinearLayout android:layout_width="match_parent"
38.         android:layout_height="wrap_content" android:gravity="center_horizontal"
39.         android:layout_marginTop="24dp" android:orientation="horizontal">
40.         <Button android:layout_height="wrap_content"
41.             android:layout_width="wrap_content" android:text="@string/modificar_foto"
42.             app:cornerRadius="0dp" android:layout_marginTop="16dp"
43.             android:layout_marginRight="8dp" android:id="@+id	btnModificarFoto"/>
44.         <Button android:layout_height="wrap_content" android:layout_width="wrap_content"
45.             android:text="@string/nueva_foto" app:cornerRadius="0dp"
46.             android:layout_marginTop="16dp" android:id="@+id	btnRegistrarFoto"/>
47.     </LinearLayout>
48. </LinearLayout>
```

- Añade **FotoFragment** al **nav\_graph** y haz que se pueda navegar desde **AlbumFragment** hacia **FotoFragment**, y desde **FotoFragment** a **ErrorFragment**



- Añade a **fotoFragment** estos dos argumentos:
  - **album**: de tipo **custom serializable**, y elige **Album**
  - **foto**: de tipo **custom serializable**, con posibilidad de poder ser nulo (**nullable**) y de tipo **Foto**
- Crea una clase llamada **FotoFragmentViewModel**, que herede de **ViewModel** y que tenga como variable de instancia el objeto **Foto?** que está siendo editado (en caso de que sea **null**, es porque se está creando una nueva foto) y el objeto **Album** en el que se encuentra (o en el que se guardará la nueva foto)

```

1. class FotoFragmentViewModel : ViewModel() {
2.     lateinit var album: Album
3.     var foto: Foto? = null // inicialmente no se está editando ninguna foto
4. }
```

- Abre **FotoFragment**, borra todo el código obsoleto, inicializa **view binding**, implemente **NavegadorError**, e inicialice **viewModel**, guardando en sus variables de instancia los parámetros recibidos desde la pantalla previa.

```

1. class FotoFragment : Fragment(), NavegadorError{
2.     lateinit var binding: FragmentFotoBinding
3.     lateinit var viewModel: FotoFragmentViewModel
4.     override fun onCreateView(
5.         inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
6.     ): View? {
7.         inicializarBinding(inflater, container)
8.         inicializarViewModel()
9.         return binding.root
10.    }
11.    private fun inicializarBinding(inflater: LayoutInflater, container: ViewGroup?) {
12.        binding=FragmentFotoBinding.inflate(inflater,container,false)
13.    }
14.    private fun inicializarViewModel() {
15.        viewModel=ViewModelProvider(this).get(FotoFragmentViewModel::class.java)
16.        viewModel.album = FotoFragmentArgs.fromBundle(requireArguments()).album
17.        viewModel.foto = FotoFragmentArgs.fromBundle(requireArguments()).foto
18.    }
19.    override fun getNavController(): NavController = findNavController()
20.    override fun getFlechaNavegacionPantallaError(m: String): NavDirections =
21.        FotoFragmentDirections.actionFotoFragmentToErrorFragment(m)
22. }
```

- Añade al archivo **Funciones.kt** una **extension function** para la clase **ImageView**, llamada **ponerFoto**. Esta función, recibirá un objeto **Foto** y lo pondrá en el **ImageView** que está siendo extendido, usando la librería **Coil** (que añade el método **load** al **ImageView**)

```

1. fun ImageView.ponerFoto(f:Foto){
2.     load(f.urlFoto.toUri()){
3.         crossfade(true)
4.         placeholder(R.drawable.loading)
5.         error(R.drawable.error)
6.     }
7. }
```

- Añade a **FotoFragment** un método llamado **inicializarInterfaz**, que se llamará en **onCreateView**, justo después de **inicializarBotones**. El método **inicializarInterfaz** mirará si el objeto **Foto** del **viewModel** es **null**, y en ese caso llamará a nuevos métodos **inicializarModoInsercion**, y en caso contrario, a **inicializarModoEdicion**

```

1. fun inicializarInterfaz(){
2.     if(viewModel.foto!=null){
3.         inicializarModoEdicion()
4.     }else{
5.         inicializarModoInsercion()
6.     }
7. }
8. fun inicializarModoInsercion(){
9.     // aquí preparamos la pantalla para tomar una nueva foto
10. }
11. fun inicializarModoEdicion(foto: Foto){
12.     // aquí preparamos la pantalla para editar una foto
13. }
```

- Programa el método **inicializarModoInsercion** para ocultar **btnModificarFoto** y poner en **txtRotulo** el mensaje “*Nueva foto*”

```

1. fun inicializarModoInsercion(){
2.     binding.apply{
3.         txtRotulo.text = "Nueva foto"
4.         btnModificarFoto.visibility = View.GONE
5.     }
6. }
```

- Programa el método **inicializarModoEdicion** para ocultar **btnRegistrarFoto**, poner en **txtRotulo** el mensaje “*Modificar foto*” y mostrar en todos los campos de texto los valores del objeto **Foto** recibido como parámetro

```

1. fun inicializarModoEdicion(){
2.     val foto = checkNotNull(viewModel.foto)
3.     binding.apply {
4.         txtRotulo.text = "Modificar foto"
5.         btnRegistrarFoto.visibility=View.GONE
6.         txtTitulo.setText(foto.titulo)
7.         txtFecha.setText(foto.fecha)
8.         txtHora.setText(foto.hora)
9.         imgFoto.ponerFoto(foto)
10.    }
11. }
```

- Añade a **FotoFragment** un método **inicializarBotones** que será llamado en **onCreateView**. Dentro de **inicializarBotones** llama a métodos para inicializar todos los botones que se ven en **FotoFragment**. También se hará que al pulsar en **imgFoto** se llame a un método **tomarFoto**

```

1. class FotoFragment : Fragment(), NavegadorError{
2.     // resto omitido
3.     override fun onCreateView(
4.         inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
5.     ): View? {
6.         inicializarBinding(inflater,container)
7.         inicializarViewModel()
8.         inicializarBotones()
9.         return binding.root
10.    }
11.    fun inicializarBotones(){
12.        binding.apply {
13.            btnRegistrarFoto.setOnClickListener{ registrarFoto() }
14.            btnModificarFoto.setOnClickListener{ modificarFoto() }
15.            btnSeleccionarFecha.setOnClickListener{ pedirFecha() }
16.            btnSeleccionarHora.setOnClickListener{ pedirHora() }
17.            imgFoto.setOnClickListener { tomarFoto() }
18.        }
19.    }
20.    private fun registrarFoto(){
21.        // aquí subiremos la foto tomada a la base de datos (disponible en modo de nueva foto)
22.    }
23.    private fun modificarFoto(){
24.        // aquí modificaremos la foto (disponible en modo edición)
25.    }
26.    private fun pedirFecha(){
27.        // aquí mostraremos una ventana para elegir una fecha (disponible en ambos modos)
28.    }
29.    private fun pedirHora(){
30.        // aquí mostraremos una ventana para elegir una hora (disponible en ambos modos)
31.    }
32.    private fun tomarFoto(){
33.        // aquí abriremos la pantalla para tomar una foto (disponible en modo de nueva foto)
34.    }
35. }
```

- Vete a **AlbumFragment** y completa el método **navegarPantallaDetalleFoto** para que se navegue hacia **FotoFragment**

```

1. fun navegarPantallaDetalleFoto(f:Foto?){
2.     findNavController().navigate(
3.         AlbumFragmentDirections.actionAlbumFragmentToFotoFragment(
4.             viewModel.album,f
5.         )
6.     )
```

- Añade a **AlbumFragment** un método **inicializarBoton**, que llamarás en **onCreateView**. Allí haz que al pulsar **btnNuevaFoto** se navegue hacia **FotoFragment** pasándole **null** (esto pone **FotoFragment** en modo inserción)

```

1. private fun inicializarBoton() {
2.     binding.btnNuevaFoto.setOnClickListener{
3.         navegarPantallaDetalleFoto(null)
4.     }
5. }
```

- Ejecuta la app y comprueba que al pulsar en **btnNuevaFoto** se navega hacia **FotoFragment** y que solo aparece el botón de registrar una nueva foto.

Vamos ahora a hacer que al pulsar **imgFoto** se abra la pantalla para tomar una foto. Comenzaremos haciendo una clase llamada **TomadorFoto**, cuya misión será usar la cámara del dispositivo para tomar una foto.

- En la carpeta **res/xml** crea un **xml resource file** llamado **files.xml**
- Abre el archivo **files.xml**, borra lo que hay y pon el siguiente contenido

```

1. <paths>
2.   <files-path
3.     name="carpeta_fotos"
4.     path=". />
5. </paths>
```

El código anterior está asociando el nombre **carpeta\_fotos** al directorio donde la app almacena sus archivos locales (su ruta es **./**)

- Abre el archivo **AndroidManifest.xml** y añade al final del bloque **application** el siguiente código:

```

1. <application>
2.   <!-- resto omitido -->
3.   <provider
4.     android:authorities="firefotos.fileprovider"
5.     android:name="androidx.core.content.FileProvider"
6.     android:exported="false"
7.     android:grantUriPermissions="true">
8.       <meta-data
9.         android:name="android.support.FILE_PROVIDER_PATHS"
10.        android:resource="@xml/files"/>
11.   </provider>
12. </application>
```

En Android cada app tiene un espacio aislado para sus archivos, de forma que ninguna app puede acceder a los archivos de otra app, a no ser que esta se los exponga.

Nuestra app necesita exponer a la app de la cámara de fotos el archivo donde queremos tomar la foto. Al añadir el código anterior al **AndroidManifest.xml** hacemos que nuestra app pueda exponer los archivos de la carpeta definida en el archivo **files.xml** a otras aplicaciones.

- En el paquete **modelo**, crea una clase llamada **TomadorFoto**, que en su constructor reciba como parámetros:
  - El **Fragment** que va a utilizar la clase
  - Una lambda que procesará un objeto **File** con la foto capturada

```

1. class TomadorFoto(
2.   val fragment: Fragment,
3.   val lambda:(File) -> Unit
4. ) {
5. }
```

- Añade a **TomadorFoto**, dos nuevas variables de instancia:
  - Una variable con un objeto **File** que tendrá la ruta donde se guardará la foto que se tomará con la cámara. En nuestra app la cámara siempre guardará la foto en un archivo llamado **foto.jpg**
  - Un objeto **Uri** que permitirá a la app de la cámara acceder al archivo **foto.jpg**.

```

1. class TomadorFoto(val fragment: Fragment, val lambda:(File) -> Unit) {
2.     val rutaFotoLocal:File = fragment.requireContext().filesDir.resolve("foto.jpg")
3.     lateinit var uriFoto: Uri
4. }
```

Debido a que **foto.jpg** pertenece a nuestra app, la app de la cámara no puede acceder a él directamente, y se necesita “proporcionar” una **url** (variable **uriFoto**) a la app de la cámara para que pueda acceder a él.

- Añade a **TomadorFoto** una nueva variable de instancia, que guardará el proceso que se encargará de lanzar la app de la cámara de fotos y tomar la foto. Si la foto se toma correctamente, se lanzará la lambda que procesa el archivo **foto.jpg**

```

1. class TomadorFoto(val fragment: Fragment, val lambda:(File) -> Unit) {
2.     // resto omitido
3.     private val procesoTomarFoto =
4.         fragment.registerForActivityResult(ActivityResultContracts.TakePicture()){ resultado ->
5.             if(resultado){ // resultado es un boolean
6.                 lambda(rutaFotoLocal)
7.             }
8.         }
9. }
```

Los **Fragment** tienen un método llamado **registerForActivityResult**, que permite lanzar un nuevo **Activity** (en este caso usamos un **Activity** predefinido en **ActivityResultContracts** que abre la app de la cámara de fotos) y procesar el resultado dentro de una lambda.

- Por último, añade un método llamado **tomarFoto** a **TomadorFoto**. En dicho método, se borrará el archivo **foto.jpg** (si ya existe), se llenará **uriFoto** con una url para que otras apps (en nuestro caso la app de la cámara de fotos) puedan acceder a **foto.jpg** y se lanzará el proceso para tomar fotos en **uriFoto**

```

1. fun tomarFoto(){
2.     rutaFotoLocal.delete()
3.     uriFoto = FileProvider.getUriFromFile(
4.         fragment.requireContext(),
5.         "firefotos.fileprovider",
6.         rutaFotoLocal
7.     )
8.     procesoTomarFoto.launch(uriFoto)
9. }
```

El método **FileProvider.getUriForFile** permite obtener una **Uri** para que otras apps puedan acceder a un **File** de nuestra app. La condición es que dicho archivo esté dentro de la carpeta a la que hayamos dado permiso en el **AndroidManifest.xml**

- Abre **FotoFragment** y añádele una variable de instancia de tipo **TomadorFoto**, que será inicializada en un método llamado **inicializarTomadorFoto**, que será llamado en **onCreateView**. El método **inicializarTomadorFoto** creará un objeto **TomadorFoto** y mostrará en **imgFoto** la foto capturada

```
1. class FotoFragment : Fragment(), NavegadorError{
2.     // resto omitido
3.     lateinit var tomadorFoto: TomadorFoto
4.     override fun onCreateView(
5.         inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
6.     ): View? {
7.         inicializarBinding(inflater,container)
8.         inicializarViewModel()
9.         inicializarTomadorFoto()
10.        inicializarBotones()
11.        inicializarInterfaz()
12.        return binding.root
13.    }
14.    private fun inicializarTomadorFoto(){
15.        tomadorFoto = TomadorFoto(this){ rutaFotoLocal ->
16.            binding.imgFoto.load(rutaFotoLocal.toUri()){
17.                diskCachePolicy(CachePolicy.DISABLED)
18.                memoryCachePolicy(CachePolicy.DISABLED)
19.            }
20.        }
21.    }
22. }
```

Para poner la foto capturada en **imgFoto** usamos el método **load**. Las líneas que hay dentro de su expresión lambda deshabilitan la memoria caché, y esto es necesario porque la foto capturada siempre está en la ruta **foto.jpg**, y si activamos la caché, muchas veces no se actualizará correctamente.

- En **FotoFragment**, programa el método **tomarFoto** para que llame al método **tomarFoto** del **tomadorFoto** si el objeto **foto** de su **viewModel** es **null**

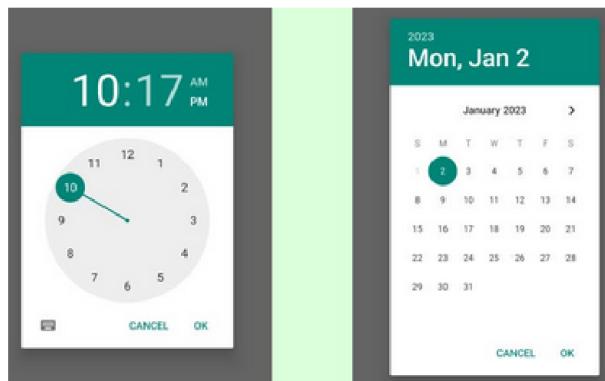
```
1. private fun tomarFoto(){
2.     if(viewModel.foto==null) {
3.         tomadorFoto.tomarFoto()
4.     }
5. }
```

Hacemos esa comprobación porque estamos aceptando que en el modo edición (la variable **foto** del **viewModel** no es **null**), la foto tomada no puede ser cambiada. Si no queremos este comportamiento, quitaríamos esa comprobación.

- Ejecuta la app y comprueba que al pulsar el botón para tomar fotos, se abre la app de la cámara y es posible tirar una foto con ella. Comprueba también que la foto capturada se ve en **imgFoto** dentro de **FotoFragment**

## 9 – DatePickerDialog y TimePickerDialog

**FotoFragment** posee dos botones para poner una fecha y una hora. Lo ideal en esta situación es que se abra una ventana con un calendario o un reloj donde podamos elegirlos de forma gráfica. Android proporciona dos **DialogFragment** ya predefinidos, llamados **DatePickerDialog** y **TimePickerDialog** para esta finalidad.



- En el archivo **Funciones.kt** añade una **extension function** a la clase **LocalDate** que permita transformarla en un **String** con formato **día/mes/año**

```
1. fun LocalDate.darFormato():String =  
2.     DateTimeFormatter.ofPattern("dd/MM/yyyy").format(this)
```

- En **FotoFragment** programa el método **pedirFecha**, de forma que se cree un **DatePickerDialog** que sirva para llenar **txtFecha**

```
1. private fun pedirFecha() {  
2.     val hoy = LocalDate.now()  
3.     DatePickerDialog(  
4.         requireContext(),  
5.         { v, a, m,d ->  
6.             val f = LocalDate.of(a,m,d)  
7.             binding.txtFecha.setText(f.darFormato())  
8.         },  
9.         hoy.year,  
10.        hoy.monthValue,  
11.        hoy.dayOfMonth  
12.    ).show()  
13. }
```

Para mostrar el **DatePickerDialog** basta con usar su constructor pasándole estos parámetros:

- El **context** de la app
- Una lambda que indica lo que se va a hacer con el año (a), mes (m) y día (d) que ha elegido el usuario. En nuestro caso, lo ponemos en **txtFecha** con formato **día/mes/año**
- Los valores del año, mes y día que aparece marcado por defecto en el calendario

Una vez que se usa el constructor, el método **show** permite mostrarlo

- Ejecuta la app y comprueba que es posible elegir una fecha con un calendario.
- En **FotoFragment** programa el método **pedirHora**, de forma que se cree un **TimePickerDialog** que sirva para llenar **txtHora**

```

1. private fun pedirHora() {
2.     val hora = LocalTime.now()
3.     TimePickerDialog(
4.         context,
5.         { v, h, m -> binding.txtHora.setText(LocalTime.of(h,m).toString())},
6.         hora.hour,
7.         hora.minute,
8.         true // formato 24 horas
9.     ).show()
10. }
```

Para mostrar el **TimePickerDialog** basta con usar su constructor pasándole estos parámetros:

- El **context** de la app
- Una lambda que indica lo que se va a hacer con las horas (h) y minutos (m) que ha elegido el usuario. En nuestro caso, lo ponemos en **txtHora** con formato **horas:minutos**
- Los valores de hora y minutos que aparecerán marcados por defecto

Una vez que se usa el constructor, el método **show** permite mostrarlo

- Ejecuta la app y comprueba que es posible elegir una hora con un reloj.

## 10 – Cloud storage

**Cloud storage** es el nombre que recibe el sistema de almacenamiento en la nube de **Firebase**. Con él, los usuarios registrados pueden subir archivos como por ejemplo, fotografías. **Cloud storage** es el sistema ideal para subir las fotos que captura nuestra app, pero no es gratuito, y utiliza un sistema de pago por accesos. Para poder utilizarlo, es necesario crear un sistema de pago y dar un número de cuenta corriente.

Por este motivo, no utilizaremos **cloud storage**, pero simularemos cómo se haría la app si se utilizase ese sistema. En lugar de subir las fotos a la nube, mantendremos las fotos en el dispositivo móvil, pero el diseño que haremos es el mismo que si usáramos **cloud storage**.

- En el paquete **modelo** crea una interfaz **AlmacenFotos**, que representa un lugar (local o en la nube) donde podemos almacenar fotos. En esa interfaz, pondremos dos métodos:
  - **subirFoto**: sube una foto al lugar de almacenamiento
  - **getUriFoto**: recibe el **id** de una foto y nos permite obtener una dirección (en forma de objeto **Uri**) para consultarla.

```

1. interface AlmacenFotos {
2.     fun subirFoto(
3.         rutaFoto: File,
4.         lambdaExito: (Uri) -> Unit,
5.         lambdaError: (String) -> Unit
6.     )
7.     fun getUriFoto(
8.         idFoto: String,
9.         lambdaExito: (Uri) -> Unit,
10.        lambdaError: (String) -> Unit
11.    )
12. }

```

El uso de la interfaz **AlmacenFotos** hace nuestra app independiente del sistema usado para almacenar las fotos. Como es habitual en este proyecto, **lambdaExito** y **lambdaError** se usan para ejecutar código en caso de éxito o error.

Para no proporcionar datos bancarios, programaremos una implementación que simule cómo se subirían las fotos, pero haciendo uso del almacenamiento local. Si usáramos el **cloud storage** de **Firebase**, bastaría con proporcionar una nueva implementación de esta interfaz y el resto de la app seguiría intacta.

- En el paquete **modelo** crea una clase llamada **AlmacenFotosLocal** que implemente la interfaz anterior

```

1. class AlmacenFotosLocal() : AlmacenFotos {
2.     override fun subirFoto(
3.         rutaFoto: File,
4.         lambdaExito: (Uri) -> Unit,
5.         lambdaError: (String) -> Unit
6.     ){
7.         // aquí simularemos la subida de una foto al cloud storage
8.     }
9.     override fun getUriFoto(
10.         idFoto: String,
11.         lambdaExito: (Uri) -> Unit,
12.         lambdaError: (String) -> Unit
13.     ){
14.         // aquí obtendremos una url para una foto del cloud storage
15.     }
16. }

```

La clase **AlmacenFotosLocal** simula el almacenamiento en **cloud storage**, pero en local, de la siguiente forma:

- Las fotos “subidas” se quedarán en la carpeta de archivos de la app.
- Cada foto “subida” tendrá un nombre que será un número correlativo (y que será su **id**). Por ejemplo, la primera foto será **1.jpg**, la segunda será **2.jpg** y así sucesivamente.
- La “url” para acceder a la foto, será simplemente su ruta, pero en lugar de proporcionarse como un objeto **File**, se proporcionará como un **Uri**
- Programa **subirFoto** de manera que reciba la foto tomada por la cámara (archivo **foto.jpg**) y en caso de que exista, la “suba” cambiándole su nombre al siguiente número consecutivo.

```

1. override fun subirFoto(
2.     rutaFoto: File,
3.     lambdaExito: (Uri) -> Unit,
4.     lambdaError: (String) -> Unit
5. ){
6.     try {
7.         if(rutaFoto.exists()) {
8.             val carpetaFotos = rutaFoto.parentFile
9.             val numero = carpetaFotos.listFiles()
10.                .filter { archivo -> archivo.nameWithoutExtension.matches("\d+".toRegex()) }
11.                .map { archivo -> archivo.nameWithoutExtension.toInt() }
12.                .maxOrNull() ?: 0
13.             val ruta = carpetaFotos.resolve("${numero + 1}.jpg")
14.             rutaFoto.renameTo(ruta)
15.             lambdaExito(ruta.toUri())
16.         }else{
17.             lambdaError("No hay ninguna foto para registrar")
18.         }
19.     }catch(e:Exception){
20.         lambdaError(e.message.toString())
21.     }
22. }

```

Para obtener el número (**o id**) con el que se guardará la siguiente foto, partimos de la lista de archivos de la carpeta de la app. Con **filter**, nos quedamos con aquellos archivos cuyo nombre sea un número (se usa una **expresión regular** para ello) y con **map** pasamos, de una lista de archivos, a una lista con dichos números. El método **maxOrNull** nos devuelve el máximo de todos esos números, o **null** en caso de que la lista de números esté vacía.

En caso de que **maxOrNull** devuelva **null**, el operador **Elvis ( ?: )** nos permite dar un valor concreto al **null**, que en este caso, será **0** (por supuesto, si **maxOrNull** no devuelve **null**, entonces el operador Elvis no hace nada).

- Programa **getUriFoto** para que obtenga un objeto **Uri** a partir del archivo cuya ruta es el **id** pasado como parámetro, y ejecute **lambdaExito** con él.

```

1. override fun getUriFoto(
2.     idFoto: String,
3.     lambdaExito: (Uri) -> Unit,
4.     lambdaError: (String) -> Unit
5. ) {
6.     try{
7.         val ruta = File("${idFoto}.jpg")
8.         val u=ruta.toUri()
9.         lambdaExito(u)
10.     }catch(e:Exception){
11.         lambdaError(e.message.toString())
12.     }
13. }

```

- Abre el archivo **AlmacenFotos.kt** y añade al final una función llamada **getAlmacenFotos** que nos devuelva la implementación que hemos hecho

```

1. interface AlmacenFotos {
2.     // código omitido
3. }
4. fun getAlmacenFotos() = AlmacenFotosLocal()

```

Si se desea usar **cloud storage** en lugar del almacén local, será necesario programar otra clase que implemente **AlmacenFotos** y devolverla en la función **getAlmacenFotos**

- Vete a **FotoFragmentViewModel** y programa un método llamado **registrarFoto**, que reciba un objeto **File** con la ruta de una foto y la grabe en el **AlmacenFotos** que usa nuestra app (que nos lo devuelve la función **getAlmacenFotos**). Si todo va bien, se ejecutará un **lambdaExito** y si falla, se ejecutará un **lambdaError**

```
1. fun registrarFoto(
2.     titulo:String, fecha:String, hora:String, rutaLocal: File,
3.     lambdaExito:(Foto) -> Unit,
4.     lambdaError:(String)->Unit
5. ){
6.     try {
7.         getAlmacenFotos().subirFoto(
8.             rutaLocal,
9.             lambdaExito = { uriFotoAlmacen ->
10.                 // aquí crearemos un objeto Foto y lo subiremos a Firestore
11.             },
12.             lambdaError = { m -> lambdaError(m)}
13.         )
14.     }catch(e:Exception){
15.         lambdaError(e.message.toString())
16.     }
17. }
```

- En **FotoFragment**, programa el método **registrarFoto**, de forma que use el método **registrarFoto** de su **viewModel** para subir la foto tomada por la cámara (es el archivo **foto.jpg**, cuya ruta ya tenemos en la propiedad **rutaFotoLocal** del objeto **tomadorFoto**) al **AlmacenFotos** que usa nuestra app. Si todo va bien, se mostrará un mensaje en Logcat y se volverá a la pantalla previa. Si hay algún error, se navegará a la pantalla de error.

```
1. private fun registrarFoto(){
2.     viewModel.registrarFoto(
3.         binding.txtTitulo.text.toString(),
4.         binding.txtFecha.text.toString(),
5.         binding.txtHora.text.toString(),
6.         tomadorFoto.rutaFotoLocal,
7.         lambdaExito = { foto ->
8.             Log.d("registrar foto", "foto registrada")
9.             findNavController().popBackStack()
10.        },
11.        lambdaError = { m -> navegarPantallaError(m)}
12.    )
13. }
```

- Ejecuta la app y registra varias fotos. Pulsa el ícono del **device explorer** y entra en la carpeta **/data/data/dam.moviles.firefotos/files**. Allí podrás ver las fotos que se van almacenando y observa que todas tienen números consecutivos.

El **device explorer**  es una herramienta que nos permite ver el sistema de almacenamiento del dispositivo. La carpeta de archivos asociada a nuestra app tiene la ruta **/data/data/(paquete de la app)/files**.

- Vete a **FirestoreRepository** y programa el método **crearFoto**, para que cree en **Firebase**, dentro de la carpeta **fotos** del **Album** que se le pasa como parámetro, un nuevo documento con la información de la fotografía

```

1. override suspend fun crearFoto(
2.         a:Album,titulo:String, fecha:String, hora:String, uriFoto:String
3. ):Foto {
4.     val doc=Firebase.firestore
5.         .collection("firefotos") // accedo a la colección "firefotos"
6.         .document(a.idUsuario) // accedo al documento cuyo id es el usuario del álbum
7.         .collection("albums") // accedo a la colección "albums" de ese usuario
8.         .document(a.id) // accedo al álbum que tiene el id proporcionado
9.         .collection("fotos") // accedo a su colección "fotos"
10.        .document() // creo un documento vacío, con un id puesto por Firestore
11.        val foto = Foto(doc.id,a.id,a.idUsuario,titulo,fecha,hora,uriFoto ) // creo el objeto Foto
12.        doc.set(foto) // inserto el objeto Foto en el documento
13.        .await() // punto de suspensión para la corriputina que llame a crearFoto
14.    return foto // al finalizar el punto de suspensión, el hilo devuelve el objeto Foto
15. }
```

- Abre **FotoFragmentViewModel** y completa el método **registrarFoto**, de forma que si la foto se sube correctamente al **AlmacenFotos**, se llame en una corriputina al método anterior para crear en **Firebase** un documento **Foto** con la información de la foto, y dicho objeto **Foto** sea procesado por **lambdaExito**

```

1. fun registrarFoto(
2.     titulo:String,fecha:String,hora:String,rutaLocal: File,
3.     lambdaExito:(Foto) -> Unit,
4.     lambdaError:(String)->Unit
5. ){
6.     try {
7.         getAlmacenFotos().subirFoto(
8.             rutaLocal,
9.             lambdaExito = { uriFotoAlmacen ->
10.                 viewModelScope.launch {
11.                     val foto = getFireFotosRepository().crearFoto(
12.                         album,titulo,fecha,hora,uriFotoAlmacen.toString()
13.                     )
14.                     lambdaExito(foto)
15.                 }
16.             },
17.             lambdaError = { m -> lambdaError(m)}
18.         )
19.     }catch(e:Exception){
20.         lambdaError(e.message.toString())
21.     }
22. }
```

- Ejecuta la app y comprueba que puedes registrar fotos y que toda la información de la foto tomada se sube a **Firebase**

## 11 – MaskableFrameLayout

Para realizar un carousel de imágenes ponemos un **RecyclerView** con un **CarouselLayoutManager**, como ya hicimos en apartados anteriores. El resto de su configuración se hace como siempre, creando una vista, un **view holder** y un **adapter**

La novedad está en que la vista, para que pueda ser parcialmente visible cuando va por las esquinas, debe estar encerrada en un **MaskableFrameLayout**

- Abre **FirestoreRepository** y programa el método **getFotos** para que nos devuelva la lista de objetos **Foto** que hay en **Firestore**

```

1. override suspend fun getFotos(a: Album): List<Foto> =
2.     Firebase.firestore
3.         .collection("firefotos") // accedo a la colección "fotos"
4.         .document(a.idUsuario) // recupero el documento cuyo id es el del usuario
5.         .collection("albums") // accedo a la colección "albums"
6.         .document(a.id) // recupero al album cuyo id es el del album que queremos consultar
7.         .collection("fotos") // accedo a la colección "fotos"
8.         .get()// recuperar TODOS los documentos de la colección "fotos"
9.         .await()//interrumpe la corriente hasta que está disponible la lista de documentos
10.        .toObjects(Foto::class.java) // convierte cada documento consultado en un objeto Foto

```

El método **get**, cuando estamos situados en una colección, consulta todos los documentos de esa colección. Una vez que están disponibles (tras la finalización de la suspensión producida en **await**), el método **toObjects** convierte cada documento consultado en un objeto **Foto**.

- En **AlbumFragmentViewModel** añade un método llamado **cargarFotos**, que rellene las dos variables de instancia de esa clase:
  - o **fotos**, con la lista de fotos del album que se está viendo en **AlbumFragmant**
  - o **fotoVisible**, con la primera foto de la lista del álbum, o **null** si dicha lista está vacía (*esto se hace para que cuando se abra **AlbumFragmant** se vea automáticamente la primera foto del álbum, si la hay*)

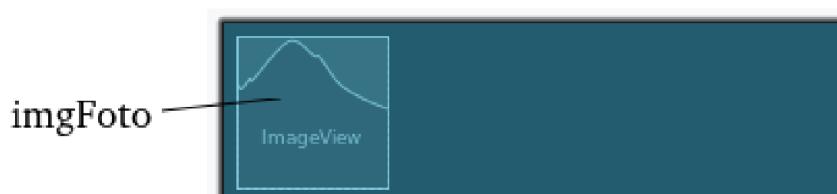
El método también recibirá como parámetros **lambdaExito** y **lambdaError** para ejecutar en caso de que todo vaya bien o se produzca un error.

```

1. fun cargarFotos(lambdaExito:() -> Unit, lambdaError:(String)->Unit){
2.     viewModelScope.launch {
3.         try {
4.             fotos = getFireFotosRepository().getFotos(album).toMutableList()
5.             fotoVisible = if (fotos.isNotEmpty()) fotos.first() else null
6.             lambdaExito()
7.         }catch(e:Exception){
8.             lambdaError(e.message.toString())
9.         }
10.    }
11. }

```

- En la carpeta **res** pulsa el botón derecho y crea un **layout resource file** llamado **foto.xml** y dale la siguiente apariencia:



```

1. <com.google.android.material.carousel.MaskableFrameLayout
2.     xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:app="http://schemas.android.com/apk/res-auto"
4.     xmlns:tools="http://schemas.android.com/tools"
5.     android:layout_width="100dp"
6.     android:layout_height="100dp"
7.     android:layout_margin="8dp">
8.     <ImageView
9.         android:id="@+id/imgFoto"
10.        android:layout_width="match_parent"
11.        android:layout_height="match_parent"
12.        android:scaleType="centerCrop"/>
13. </com.google.android.material.carousel.MaskableFrameLayout>
```

- En el paquete **vista**, crea una clase llamada **FotoHolder**, que herede de **RecyclerView.ViewHolder** y que reciba en su constructor el objeto **binding** de un **foto.xml**. Esta clase tendrá una variable de instancia de tipo **Foto**, que será la foto cuyos datos se mostrarán en su vista, y un método **mostrarFoto** para establecerla. Puesto que el método **ponerFoto** que añadimos al **ImageView** es un punto de suspensión, tendremos que llamarlo en una corutina y para ello, **FotoHolder** deberá implementar **CoroutineScope**, como ya se hizo en **AlbumHolder**

```

1. class FotoHolder(
2.     val binding:FotoBinding
3. ) : RecyclerView.ViewHolder(binding.root), CoroutineScope{
4.     lateinit var foto:Foto
5.     fun mostrarFoto(f: Foto){
6.         foto = f
7.         launch{
8.             binding.imgFoto.ponerFoto(f)
9.         }
10.    }
11.    override val coroutineContext: CoroutineContext
12.        get() = Dispatchers.Main
13. }
```

- En el paquete **vista**, crea una clase llamada **FotoAdapter**, que herede de **RecyclerView.Adapter<FotoHolder>** y reciba como variables de instancia en su constructor una lista de fotos y el código que se lanza cuando se pulsa un **FotoHolder**. Su programación es análoga a la de **AlbumAdapter**

```

1. class FotoAdapter(
2.     val fotos:List<Foto>,
3.     val lambda:(FotoHolder) -> Unit
4. ) : RecyclerView.Adapter<FotoHolder>(){
5.     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): FotoHolder {
6.         val inflater = LayoutInflater.from(parent.context)
7.         val binding = FotoBinding.inflate(inflater,parent,false)
8.         return FotoHolder(binding)
9.     }
10.    override fun getItemCount(): Int = fotos.size
11.    override fun onBindViewHolder(holder: FotoHolder, position: Int) {
12.        holder.mostrarFoto(fotos.get(position))
13.        holder.binding.root.setOnClickListener {
14.            lambda(holder)
15.        }
16.    }
17. }
```

- En **AlbumFragment** pon un método llamado **seleccionarFoto**, que reciba un objeto **Foto?** y si no es **null**, lo ponga como imagen central (en **imgFoto**). En caso contrario, mostrará la imagen por defecto del drawable **sin\_imagen**

```

1. private fun seleccionarFoto(f:Foto?){
2.     viewModel.fotoVisible = f
3.     if(f!=null) {
4.         binding.imgFoto.ponerFoto(f)
5.     }else{
6.         binding.imgFoto.setImageResource(R.drawable.sin_imagen)
7.     }
8. }
```

- En **AlbumFragment** pon un método llamado **inicializarRecyclerView** que ponga al **RecyclerView** un **FotoAdapter** que muestre la lista de fotos de su **viewModel**

```

1. private fun inicializarRecyclerView() {
2.     binding.lstFotos.adapter = FotoAdapter(viewModel.fotos){ holder ->
3.         seleccionarFoto(holder.foto)
4.         binding.imgFoto.ponerFoto(holder.foto)
5.     }
6. }
```

- En **AlbumFragment** termina el método **inicializarViewModel**, para que se cargue la lista de fotos, y en caso de que no haya error, se ponga en la imagen central la imagen que se pinche en el carousel e inicialice el **RecyclerView**.

```

1. private fun inicializarViewModel() {
2.     viewModel = ViewModelProvider(this).get(AlbumFragmentViewModel::class.java)
3.     viewModel.album = AlbumFragmentArgs.fromBundle(requireArguments()).album
4.     viewModel.cargarFotos(
5.         lambdaExito = {
6.             seleccionarFoto(viewModel.fotoVisible)
7.             inicializarRecyclerView()
8.         },
9.         lambdaError = { m -> navegarPantallaError(m)}
10.    )
11. }
```

- Ejecuta la app y comprueba que **AlbumFragment** muestra un carousel con todas las fotos que se van tomando. Comprueba también que al pulsar sobre una foto, esta se muestra en la zona central de la pantalla.
- Abre el archivo **Funciones.kt** y modifica la función **ponerFotoAlbum** para que, en caso de que el álbum no esté vacío, se coja la primera foto del álbum y se muestre en el **ImageView** que está siendo extendido

```

1. suspend fun ImageView.ponerFotoAlbum(a:Album) {
2.     val listaFotos = getFireFotosRepository().getFotos(a)
3.     if(!listaFotos.isEmpty()) {
4.         ponerFoto(listaFotos.first())
5.     }else{
6.         setImageResource(R.drawable.sin_imagen)
7.     }
8.     setImageResource(R.drawable.sin_imagen)
9. }
```

- Ejecuta la app y comprueba que en el **RecyclerView** de **PrincipalFragment** se ve la primera foto de cada álbum.