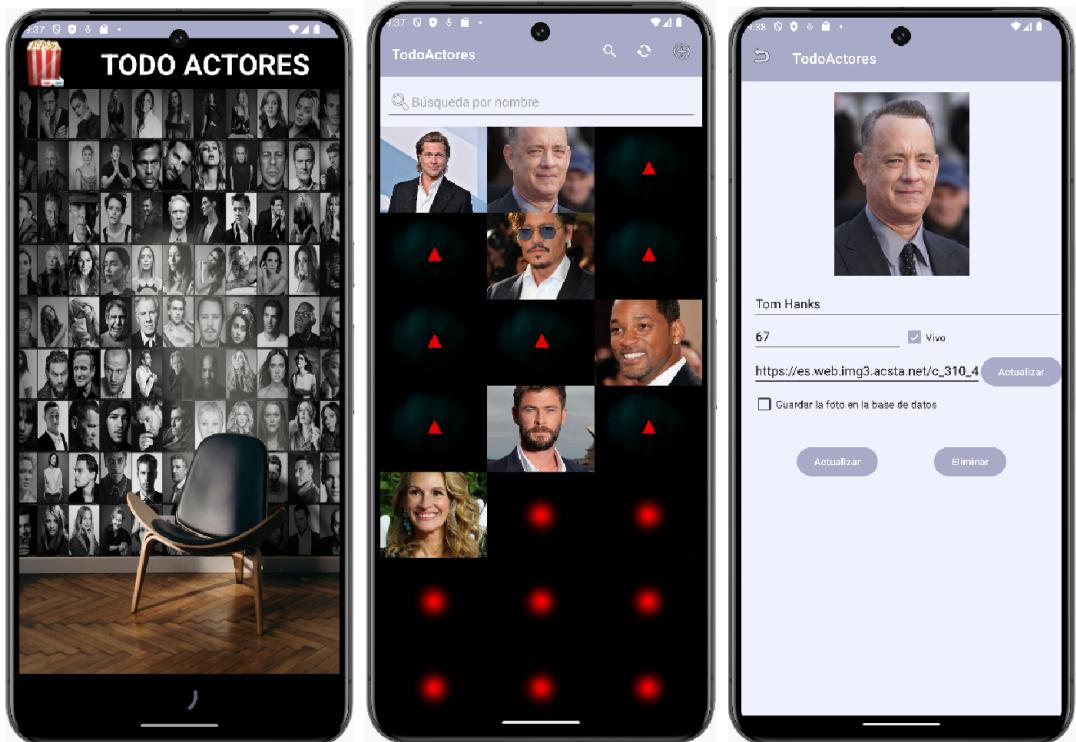


PROYECTO 8

TODO ACTORES



En este proyecto haremos una app para acceder a un sencillo API Rest de actores usando la librería Retrofit y su integración con Moshi y Coil. Durante su desarrollo exploraremos las distintas formas de hacer operaciones crud con un API.

Durante su desarrollo se tratarán estos conceptos:

- Permiso de internet
- Navegación común a la pantalla de error
- Acceso a un API Rest
- Operaciones CRUD sobre un API Rest
- Splash screen de carga
- RecyclerView con datos remotos
- Formularios adaptables
- Toolbar dinámica
- SearchView simple
- AutoCompleteTextView

1 – Permiso de internet

La app que vamos a desarrollar necesita acceder a internet. Como este es un permiso muy común, no es necesario aprobación explícita por parte del usuario para usarlo, y basta con incluir en el archivo **AndroidManifest.xml** que la app lo necesita, para obtenerlo.

Puesto que en esta app es necesario acceder a un servidor en local, en el que esté instalada el API de actores que vamos a usar, es necesario modificar las políticas de seguridad para que la app pueda conectarse a direcciones locales (por defecto, esto no es posible por motivos de seguridad).

Ambas cosas se realizan en el archivo **AndroidManifest.xml**

- Crea el proyecto en AndroidStudio, y ponle como mínimo el **API 33 (Tiramisu)**
- Abre el archivo **AndroidManifest.xml** e incluye las líneas indicadas para solicitar el acceso a internet y para poder conectarse a direcciones locales.

```
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.     <application
5.       android:usesCleartextTraffic="true"
6.       android:allowBackup="true"
7.       android:dataExtractionRules="@xml/data_extraction_rules"
8.       android:fullBackupContent="@xml/backup_rules"
9.       android:icon="@mipmap/ic_launcher"
10.      android:label="@string/app_name"
11.      android:roundIcon="@mipmap/ic_launcher_round"
12.      android:supportsRtl="true"
13.      android:theme="@style/Theme.ElTiempo"
14.      tools:targetApi="31">
15.       <activity
16.         android:name=".MainActivity"
17.         android:exported="true">
18.           <intent-filter>
19.             <action android:name="android.intent.action.MAIN" />
20.
21.             <category android:name="android.intent.category.LAUNCHER" />
22.           </intent-filter>
23.         </activity>
24.       </application>
25.     </manifest>
```

- Abre el archivo **build.gradle.kts (Project)** y agrega los plugin **kapt** y **safe-args**

```
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. }
```

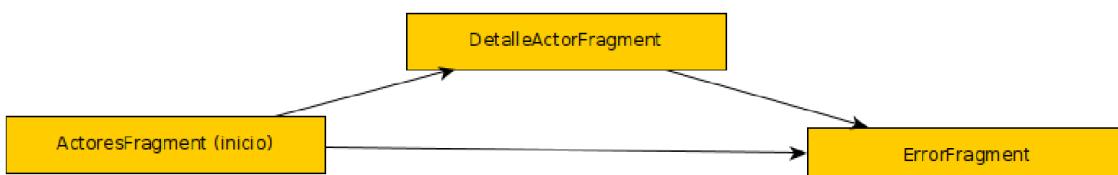
- Abre el archivo **build.gradle.kts (Module)**, habilita el **view binding** y agrega las dependencias para usar las librerías **Moshi**, **Retrofit**, **Coil** y la integración de Moshi con Retrofit

```

1. plugins {
2.     alias(libs.plugins.android.application)
3.     alias(libs.plugins.kotlin.android)
4.     id("org.jetbrains.kotlin.kapt")
5.     id("androidx.navigation.safeargs.kotlin")
6. }
7. android {
8.     namespace = "dam.moviles.todoactores"
9.     compileSdk = 35
10.    buildFeatures{
11.        viewBinding=true
12.    }
13.    // resto de la sección android omitido
14. }
15. dependencies {
16.     // navigation component
17.     implementation("androidx.navigation:navigation-fragment-ktx:2.8.5")
18.     // moshi
19.     implementation("com.squareup.moshi:moshi:1.15.1")
20.     implementation("com.squareup.moshi:moshi-kotlin-codegen:1.15.1")
21.     kapt("com.squareup.moshi:moshi-kotlin-codegen:1.15.1")
22.     // retrofit
23.     implementation("com.squareup.retrofit2:retrofit:2.9.0")
24.     // integración de retrofit con moshi
25.     implementation("com.squareup.retrofit2:converter-moshi:2.9.0")
26.     // coil
27.     implementation("io.coil-kt.coil3:coil:3.0.4")
28.     implementation("io.coil-kt.coil3:coil-network-okhttp:3.0.4")
29.     // resto de la sección de dependencias omitido
30. }

```

- Crea, dentro del paquete de la app, los siguientes subpaquetes:
 - **modelo**
 - **viewmodel**
 - **vista**
- Mueve el archivo **MainActivity** al paquete **vista**
- Añade al paquete **vista** los siguientes **Fragment**, eliminando del código fuente generado por Android Studio las partes obsoletas.
 - **ActoresFragment** → Pantalla principal, con una lista de actores
 - **DetalleActorFragment** → Pantalla para gestionar los datos de un actor
 - **ErrorFragment** → Pantalla que se muestra si se produce un error
- Diseña el siguiente grafo de navegación y guárdalo con el nombre **nav_graph**



- Añade un argumento a **ErrorFragment** llamado **mensajeError** de tipo **String**
- Abre **fragment_error.xml** con la vista xml, borra todo lo que hay y añade un **LinearLayout** que contenga dos **TextView** de forma que la interfaz quede así:



```

1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.   xmlns:tools="http://schemas.android.com/tools"
3.   android:padding="16dp"
4.   android:orientation="vertical"
5.   android:layout_width="match_parent"
6.   android:layout_height="match_parent">
7.   <TextView
8.     android:layout_width="wrap_content"
9.     android:layout_height="wrap_content"
10.    android:textStyle="bold"
11.    android:textSize="16sp"
12.    android:text="Error al iniciar la app"/>
13.   <TextView
14.     android:id="@+id/txtError"
15.     android:layout_width="wrap_content"
16.     android:layout_height="wrap_content"
17.     android:layout_marginTop="16dp"
18.     tools:text="Mensaje de error"/>
19. </LinearLayout>

```

- Abre **ErrorFragment** y haz que al iniciarse, se muestre en **txtError** el valor del argumento **mensajeError** que recibe de la pantalla previa en el **nav_graph**.

```

1. class ErrorFragment : Fragment() {
2.     var _binding:FragmentErrorBinding? = null
3.     val binding:FragmentErrorBinding
4.         get()= checkNotNull(_binding)
5.     override fun onCreateView(
6.         inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
7.     ): View? {
8.         inicializarBinding(inflater,container)
9.         inicializarMensajeError()
10.        return binding.root
11.    }
12.    private fun inicializarBinding(inflater: LayoutInflater, container: ViewGroup?) {
13.        _binding=FragmentErrorBinding.inflate(inflater,container,false)
14.    }
15.    private fun inicializarMensajeError() {
16.        val mensaje=ErrorFragmentArgs.fromBundle(requireArguments()).mensajeError
17.        binding.txtError.text = mensaje
18.    }
19.    override fun onDestroy() {
20.        super.onDestroy()
21.        _binding=null
22.    }
23. }

```

- Inicializa **view binding** en **MainActivity**

```

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. }

```

- Abre **activity_main.xml**, borra todo lo que contiene y haz que su elemento principal sea un **LinearLayout** con dos elementos:
 - Un **FragmentContainerView**, que muestre el **NavHostFragment** y use el **nav_graph**
 - Una **MaterialToolbar**

```

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

- Abre el código fuente de **MainActivity** y haz un método llamado **inicializarToolbar**, que será llamado en **onCreate** y en el que Harás que la **MaterialToolbar** sustituya la barra de tareas que traen por defecto las apps

```

1. fun inicializarToolbar(){
2.   setSupportActionBar(binding.materialToolbar)
3. }
```

2 – Navegación común a la pantalla de error

Como todos los **Fragment** de la app pueden necesitar navegar hacia **ErrorFragment**, vamos a hacer una interfaz en la que se recoja en un método la “capacidad de navegar hacia **ErrorFragment**”. De esta forma, el resto de **Fragment** implementarán dicha interfaz y tendrán disponible el método que les permite navegar hacia **ErrorFragment**

- En el paquete **vista** crea una interfaz llamada **NavegadorError**, que tenga:
 - Una propiedad abstracta que nos devuelva el **NavController**
 - Un método abstracto que nos devuelva la “flecha” (su tipo es **NavDirections**) que une en el **nav_graph** la pantalla con **ErrorFragment**
 - Un método no abstracto, llamado **navegarError**, que use los dos métodos anteriores para navegar hacia la pantalla de error.

```

1. interface NavegadorError {
2.   val navController:NavController
3.   fun getFlecha(mensaje:String):NavDirections
4.   fun navegarError(mensaje:String){
5.     val flecha = getFlecha(mensaje)
6.     navController.navigate(flecha)
7.   }
8. }
```

- Haz que **ActoresFragment** implemente la interfaz **NavegadorError**, sobreescribiendo **navController** y **getFlecha**

```

1. class ActoresFragment : Fragment(), NavegadorError {
2.     override val navController: NavController
3.         get() = findNavController()
4.     override fun getFlecha(mensaje: String): NavDirections =
5.         ActoresFragmentDirections.actionActoresFragmentToErrorFragment(mensaje)
6.     // resto omitido
7. }
```

En Kotlin, cuando una variable de instancia (como **navController**) lleva un **get**, estamos haciendo que su valor no se guarde en memoria, sino que se calcule siempre que se necesite (en este caso, siempre que se acceda a **navController** se estará haciendo una llamada a **findNavController**).

Los bloques **get()** como el que vemos aquí, se llaman **getters** en Kotlin.

- Habilita **view binding** y **view model** en **ActoresFragment**

```

1. class ActoresFragment : Fragment(), NavegadorError {
2.     lateinit var viewModel:ActoresFragmentViewModel
3.     override val navController: NavController = findNavController()
4.     override fun getFlecha(mensaje: String): NavDirections =
5.         ActoresFragmentDirections.actionActoresFragmentToErrorFragment(mensaje)
6.     var _binding:FragmentActoresBinding? = null
7.     val binding:FragmentActoresBinding
8.         get()= checkNotNull(_binding)
9.     override fun onCreateView(
10.         inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
11.     ): View? {
12.         inicializarBinding(inflater,container)
13.         inicializarViewModel()
14.         return binding.root
15.     }
16.     private fun inicializarViewModel{
17.         viewModel = ViewModelProvider(this).get(ActoresFragmentViewModel::class.java)
18.     }
19.     private fun inicializarBinding(inflater: LayoutInflater, container: ViewGroup?) {
20.         _binding=FragmentActoresBinding.inflate(inflater,container,false)
21.     }
22.     override fun onDestroy() {
23.         super.onDestroy()
24.         _binding=null
25.     }
26. }
```

- Repite los dos últimos pasos **DetalleActorFragment**
- Cambia el estilo de la app para que el color principal sea #A9AAC7 y el color de fondo de las ventanas sea #F0F1FF

```

1. <resources xmlns:tools="http://schemas.android.com/tools">
2.     <style name="Base.Theme.MyApplication" parent="Theme.Material3.DayNight.NoActionBar">
3.         <item name="colorPrimary">#A9AAC7</item>
4.         <item name="android:windowBackground">#F0F1FF</item>
5.     </style>
6.     <style name="Theme.MyApplication" parent="Base.Theme.MyApplication" />
7. </resources>
```

3 – Acceso a un API Rest

El API Rest de actores que usamos en este proyecto posee los siguientes endpoints:

- **http://localhost/crud/leer.php** → Usa el método GET y permite consultar todos los actores de la base de datos
- **http://localhost/crud/leer.php?id=ID** → Usa el método GET y permite consultar el actor cuyo id se pasa como parámetro en la URL
- **http://localhost/crud/insertar.php** → Usa el método POST y permite insertar un actor cuyos datos se pasan en un documento json en el cuerpo del mensaje (no en la URL)
- **http://localhost/crud/borrar.php** → Usa el método GET y permite borrar un actor cuyo id se pasa en un documento json en el cuerpo del mensaje
- **http://localhost/crud/actualizar.php** → Usa el método GET y permite modificar un actor con los datos que se pasan en un documento json en el cuerpo del mensaje. Dicho json incluirá el id del actor que se desea modificar.

Usaremos la librería **Retrofit** para enviar las peticiones anteriores al API y gracias a su integración con **Moshi**, podremos obtener objetos de Kotlin para las respuestas

- En el paquete **modelo** crea una interfaz vacía llamada **ActoresApi**, en la que iremos añadiendo métodos que realicen las peticiones que queramos hacer.

```
1. interface ActoresApi {  
2.     // aquí pondremos métodos que llamen a los endpoints del api y recojan sus resultados  
3. }
```

No tenemos que hacer ninguna clase que implemente la interfaz **ActoresApi**, ya que **Retrofit** se encargará de generar automáticamente dicha clase, que implementará la interfaz y que tenga los métodos que acceden al api y recogen sus resultados.

Gracias a **Retrofit**, nosotros solamente tenemos que limitarnos a escribir los métodos que necesitamos en la interfaz, y **Retrofit** generará su código automáticamente.

- En el paquete **modelo** crea una clase llamada **ActoresRepository**, cuya misión será alojar en sus variables de instancia un objeto que implemente **ActoresApi** y proporcionar a la app métodos que usen dicho objeto para realizar las tareas de acceso a datos que necesita la app.

```
1. class ActoresRepository {  
2.     val actoresApi:ActoresApi  
3. }
```

En general, las clases **Repository** proporcionan a las aplicaciones métodos para acceder a los datos que necesitan, de forma independiente del lugar donde están alojados (base de datos relacional, ficheros, base de datos nosql, api, etc)

- Añade a **ActoresRepository** un bloque **init** para pedir a la librería **Retrofit** que nos rellene la variable **actoresApi**

```

1. class ActoresRepository {
2.     val actoresApi:ActoresApi
3.     init{
4.         actoresApi = Retrofit.Builder() // creamos un builder para construir el objeto Retrofit
5.             .baseUrl("http://192.168.1.38/crud/") //sustituir por la ip del servidor
6.             .addConverterFactory(MoshiConverterFactory.create()) // usa Moshi para pasar json a objeto
7.             .build() // crea el objeto Retrofit
8.             .create() // usa el objeto Retrofit para obtener el objeto ActoresApi
9.     }
10. }
```

En Kotlin el bloque **init** es un complemento del constructor, que se llama para terminar de inicializar el objeto. En nuestro caso, en él estamos usando **Retrofit** para inicializar **actoresApi**, de la forma que indican los comentarios.

4 – Operaciones CRUD sobre un API Rest

Siempre hay que leer la documentación del API para saber cómo usarlo. En nuestro caso, nuestro API nos permite consultar todos los actores o consultar solo un actor a partir de su id. Cuando consultamos todos los actores obtenemos un json con una lista de actores, y cuando consultamos solo un actor, obtenemos un json con los datos de un solo actor.

En ambos casos, un actor tiene un id, un nombre, su edad, un número que es 1 si está vivo o 0 si está muerto, la url de su foto, y en algunos actores, la propia foto está codificada en base 64 en otro campo.

```
{
    "id":10,
    "nombre":"Will Smith",
    "edad":56,
    "activo":1,
    "fotoUrl":"https://cdn.britannica.com/66/103066-050-B89D5EAF/Will-Smith-actor-musician-2006.jpg",
    "fotocodif":"
}
```

- En el paquete **modelo** crea una data class **Serializable** llamada **Actor** y añádele la anotación **@JsonClass** para que **Moshi** sepa trabajar con ella. Esta clase tendrá como variables de instancia los mismos atributos que vemos en el json de respuesta del API de actores

```

1. @JsonClass(generateAdapter = true)
2. data class Actor(
3.     val id:Int,
4.     val nombre:String,
5.     val edad:Int,
6.     @Json(name="activo") val vivo:Int,
7.     @Json(name="fotoUrl") val fotoUrl:String,
8.     @Json(name="fotocodif") val fotoCodif:String
9. ):Serializable
```

Actor implementa la interfaz **Serializable**, lo que le permite ser pasada de un **Fragment** a otro cuando se navega por el **nav_graph**

- Añade a **ActoresApi** un método punto de suspensión de corrutinas, que se llame **consultarTodosActores** y que permita obtener una lista con todos los actores que nos devuelve el endpoint **leer.php**

```

1. interface ActoresApi {
2.     @GET("leer.php")
3.     suspend fun consultarTodosActores():List<Actor>
4. }
```

El método **consultarTodosActores** está anotado con la anotación **@GET** porque el endpoint **leer.php** usa dicho método (otros métodos posibles serían **POST**, **DELETE**, etc) y la documentación del API nos dice cuál debemos usar). Como parámetros, el método recibe la terminación del endpoint a partir de la url base que se introdujo cuando se creó el objeto **Retrofit**

Además, el método **consultarTodosActores** es un punto de suspensión de corrutinas, lo que significa que solo puede ser llamado dentro de una corriente. Esto es así porque dicho método tarda una cantidad de tiempo apreciable en terminar, y el hilo que ejecute la corriente (que será el hilo que dibuja la interfaz de usuario) debe poder quedarse libre para hacer otras cosas hasta que la lista de actores está disponible.

- Añade a **ActoresApi** otro método punto de suspensión de corrutinas, que se llame **consultarActor** y que nos devuelva un objeto **Actor** a partir de un número entero **id** que se pasa como parámetro.

```

1. interface ActoresApi {
2.     // resto omitido
3.     @GET("leer.php")
4.     suspend fun consultarActor(
5.         @Query("id") id:Int
6.     ):Actor
7. }
```

El parámetro **id** del método **consultarActor** tiene la anotación **@Query**, que significa que el valor de ese parámetro (marcado en rojo) forma parte de la URL de la consulta al API. Dentro de **@Query** aparece entre paréntesis el nombre del parámetro (marcado en azul) tal y como debe añadirse en la llamada al API.

<http://localhost/crud/leer.php?id=ID>

- Añade a **ActoresRepository** dos métodos, puntos de suspensión, llamados **consultarTodosActores** y **consultarActor**. Ambos usarán el objeto **actoresApi** para obtener la lista de todos los actores y un solo actor a partir de un id, respectivamente.

```

1. class ActoresRepository {
2.     // inicio de la clase omitido
3.     suspend fun consultarTodosActores():List<Actor> = actoresApi.consultarTodosActores()
4.     suspend fun consultarActor(id:Int):Actor = actoresApi.consultarActor(id)
5. }
```

Aunque el uso de estos métodos parece que está “repetido”, porque **ActoresRepository** hace uso de **ActoresApi** para hacer exactamente lo mismo, en realidad es bastante frecuente hacer esto, que se llama **delegación** (un objeto “delega” en otro para hacer una tarea).

- Añade a **ActoresRepository** un método que permita consultar una lista de actores cuyos nombres incluyen un patrón de búsqueda pasado como parámetro. Dicho patrón puede ser **null**, y eso significa que se buscarán todos los actores.

```
1. suspend fun consultarActores(patronNombre:String?) =  
2.     if(patronNombre==null || patronNombre.isEmpty())  
3.         consultarTodosActores()  
4.     else  
5.         consultarTodosActores()  
6.             .filter { actor -> actor.nombre.contains(patronNombre, ignoreCase = true) }  
7.             .toList()
```

El método **consultarActores** no está directamente soportado por el api, así que lo incluimos en **ActoresRepository** y lo programamos con los medios que nos proporciona el api (que en nuestro caso, es partir de **consultarTodosActores**)

- Para comprobar que los métodos funcionan, añade a **ActoresFragment** un método **testConsultaTodosActores**, que use **ActoresRepository** para consultar los actores en una corutina. Dicho método se llamará en **onCreateView**

```
1. class ActoresFragment : Fragment(), NavegadorError {  
2.     // resto de la clase omitido  
3.     override fun onCreateView(  
4.         inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?  
5.     ): View? {  
6.         inicializarBinding(inflater,container)  
7.         testTodosActores()  
8.         return binding.root  
9.     }  
10.    fun testTodosActores(){  
11.        lifecycleScope.launch {  
12.            try {  
13.                ActoresRepository()  
14.                    .consultarTodosActores() // punto de suspensión de la corrutina  
15.                    .forEach { actor -> Log.d("actores", actor.toString()) }  
16.            }catch(e:Exception){  
17.                navegarError("No se pudieron consultar los actores: ${e.message}")  
18.            }  
19.        }  
20.    }  
21. }
```

El objeto **lifecycleScope** permite poner en marcha con su método **launch** una corutina que es ejecutada por el hilo principal de la app (el que dibuja la interfaz y procesa la interacción con el usuario). Cuando dicho hilo llega a un punto de suspensión (como la llamada a **consultarTodosActores**), la corutina queda en espera, el hilo principal se pone a hacer otra cosa (como seguir dibujando e interactuando con el usuario) y cuando los datos están disponibles, el hilo principal retoma la corutina y muestra los mensajes en Logcat.

- Si ejecutas la app, podrás ver que en Logcat se ven todos los actores que nos devuelve el API

```
D Actor(id=4, nombre=Meryl Streep, edad=75, vivo=1, fotoUrl=https://upload.wikimedia.org/wikipedia/con
D Actor(id=5, nombre=Leonardo DiCaprio, edad=50, vivo=1, fotoUrl=https://estaticos-cdn.prensaiberica.e
D Actor(id=6, nombre=Johnny Depp, edad=60, vivo=1, fotoUrl=https://m.media-amazon.com/images/M/MV5BZj/
```

- Puedes comprobar que el método **consultarActor** funciona correctamente haciendo lo mismo, con este método en **ActoresFragment**

```
1. fun testActor(){
2.     lifecycleScope.launch {
3.         try {
4.             val actor = ActoresRepository()
5.             .consultarActor(5) // punto de suspensión de La corutina
6.             Log.d("actores", actor.toString())
7.         }catch(e:Exception){
8.             navegarError("No se pudieron consultar los actores: ${e.message}")
9.         }
10.    }
11. }
```

- Si apagas el servidor donde está el API y lanzas la app, puedes comprobar que se navega correctamente a la pantalla de error y se muestra un mensaje indicando el error de conexión

Nuestro API de actores también permite insertar en su base de datos un nuevo actor, pero a diferencia de la consulta, los datos del autor no se pasan al API en la URL, sino que viajan en el cuerpo de la petición en forma de documento json

```
juandiego@jd-desktop:~/docker_images$ curl \
> --header "Content-Type: application/json" \
> --request GET \
> --data '[{"id":-1,"nombre":"Antonio López","edad":30,"activo":1}]' \
> http://localhost/crud/insertar.php
["info":"Actor/Actriz Creado!"]
```

petición HTTP → JSON con los datos
respuesta del API →

Así mismo, vemos que la respuesta también es un documento json que tiene el formato establecido por el diseñador del API.

- Programa en el paquete **modelo** una data class llamada **Respuesta**, que tenga una variable de instancia llamada **info**, como corresponde al json de la respuesta del API de actores. La petición hará uso del método **POST**, que normalmente se usa para añadir un nuevo objeto a una base de datos

```
1. @JsonClass(generateAdapter = true)
2. data class Respuesta(
3.     val info:String
4. )
```

- Añade a **ActoresApi** la definición de un método, punto de suspensión, llamado **insertarActor**, que reciba un objeto **Actor** y lo pase como parámetro en el cuerpo de la petición HTTP al API de actores. El método devolverá un objeto **Respuesta** como el creado anteriormente

```

1. interface ActoresApi {
2.     // inicio omitido
3.     @POST("insertar.php")
4.     suspend fun insertarActor(
5.         @Body actor:Actor
6.     ):Respuesta
7. }

```

La anotación **@Body** se usa para que el parámetro se envíe como objeto json dentro del cuerpo de la petición al API.

- Añade a **ActoresRepository** un método punto de suspensión, llamado **insertarActor**, que reciba como parámetros un nombre, edad, entero (para indicar si está vivo/muerto), url de la foto y foto codificada. El método creará un objeto **Actor** con esos datos e id -1 (nuestro API lo ignorará) y lo pasará al endpoint **insertar.php** usando **ActoresApi**

```

1. class ActoresRepository {
2.     // resto omitido
3.     suspend fun insertarActor(
4.         nombre:String,edad:Int,vivo:Int,urlFoto:String,fotoCodificada:String
5.     ):Respuesta =
6.         actoresApi.insertarActor(Actor(-1,nombre,edad,vivo,urlFoto,fotoCodificada))
7. }

```

- Para comprobar si funciona, puedes crear este método **testInsertarActor** en **ActoresFragment** y llamarlo desde **onCreateView**

```

1. fun testInsertarActor(){
2.     lifecycleScope.launch {
3.         try {
4.             val respuesta:Respuesta = ActoresRepository()
5.                 .insertarActor("Juan Diego",20,1,"","")
6.             if(respuesta.info.contains("Creado!")){
7.                 Log.d("actores","El actor se ha creado con éxito")
8.             }else{
9.                 navegarError("El api no ha podido insertar el actor")
10.            }
11.        }catch(e:Exception){
12.            navegarError("No se pudo insertar el actor: ${e.message}")
13.        }
14.    }
15. }

```

El API de actores también tiene un endpoint para modificar un objeto. En nuestro caso, se necesita hacer una petición con el método **PUT** al endpoint **actualizar.php** y pasarle un objeto **Actor** cuyo **id** sea el identificador del actor que se va a modificar, y el resto de parámetros sean los datos de la modificación.

```

juandiego@jd-desktop:~/docker_images$ curl \
> --header "Content-Type: application/json" \
> --request PUT \
> --data '{"id":122,"nombre":"JD","edad":25,"activo":1}' \
> http://localhost/crud/actualizar.php
{"info":"Actor\Actriz actualizado"}

```

- Añade a **ActoresApi** la definición de un método punto de suspensión, llamado **actualizarActor**, que reciba como parámetro un objeto **Actor** y lo pase con el método **PUT** al endpoint **actualizar.php** en el cuerpo de la petición

```

1. interface ActoresApi {
2.     // resto omitido
3.     @PUT("actualizar.php")
4.     suspend fun actualizarActor(
5.         @Body actor:Actor
6.     ):Respuesta
7. }
```

- Añade a **ActoresRepository** un método punto de suspensión, llamado **actualizarActor**, que reciba como parámetros el id de un actor, el resto de sus datos (nombre, edad, activo, url de la foto y foto codificada) y use **ActoresApi** para pasarlo al endpoint **actualizar.php** del API de actores

```

1. class ActoresRepository {
2.     // resto omitido
3.     suspend fun actualizarActor(
4.         id:Int,nombre:String,edad:Int,vivo:Int,urlFoto:String,fotoCodificada:String
5.     ):Respuesta =
6.         actoresApi.actualizarActor(Actor(id,nombre,edad,vivo,urlFoto,fotoCodificada))
7. }
```

- Para comprobar si funciona, puedes crear este método **testActualizarActor** en **ActoresFragment** y llamarlo desde **onCreateView**

```

1. fun testActualizarActor(){
2.     lifecycleScope.launch {
3.         try {
4.             val respuesta:Respuesta = ActoresRepository()
5.                 .insertarActor("Juan Diego",20,1,"","")
6.             if(respuesta.info.contains("Creado!")){
7.                 Log.d("actores","El actor se ha actualizado con éxito")
8.             }else{
9.                 navegarError("El api no ha podido insertar el actor")
10.            }
11.        }catch(e:Exception){
12.            navegarError("No se pudo insertar el actor: ${e.message}")
13.        }
14.    }
15. }
```

El API de actores permite la operación de borrar un actor mediante el endpoint **borrar.php**. Funciona mediante el envío de una petición de tipo **POST**, en cuyo cuerpo debe ir un json que encierra el id del actor que se desea borrar. La respuesta obtenida es similar a las de las peticiones anteriores.

```

juandiego@jd-desktop:~/docker_images$ curl \
> --header "Content-Type: application/json" \
> --request DELETE \
> --data '{"id":124}' \
> http://localhost/crud/borrar.php
{"info":"Actor borrado con \u000e9xito o no est\u000e9 en el sistema!"}
```

- En el paquete **modelo** crea una data class llamada **Id** que poseerá una variable de instancia llamada **id** de tipo entero, para ajustarse al formato del json que enviaremos en la petición de borrado

```

1. @JsonClass(generateAdapter = true)
2. data class Id(
3.     val id:Int
4. )

```

- Añade a **ActoresApi** la definición de un método punto de suspensión, llamado **borrarActor**, que reciba como parámetro un objeto **Id** y lo pase con el método **POST** al endpoint **borrar.php** en el cuerpo de la petición

```

1. interface ActoresApi {
2.     // resto omitido
3.     @POST("borrar.php")
4.     suspend fun borrarActor(
5.         @Body id:Id
6.     ):Respuesta
7. }

```

- Añade a **ActoresRepository** un método punto de suspensión, llamado **borrarActor**, que reciba como parámetro un entero con el id de un actor, lo meta en un objeto **Id** y después use **ActoresApi** para pasarlo al endpoint **borrar.php** del API de actores

```

1. class ActoresRepository {
2.     // resto omitido
3.     suspend fun borrarActor(id:Int):Respuesta =
4.         actoresApi.borrarActor(Id(id))
5. }

```

- Para comprobar si funciona, puedes crear este método **testBorrarActor** en **ActoresFragment** y llamarlo desde **onCreateView**

```

1. fun testBorrarActor(){
2.     lifecycleScope.launch {
3.         try {
4.             val respuesta:Respuesta = ActoresRepository()
5.                 .borrarActor(122)
6.             if(respuesta.info.contains("éxito")){
7.                 Log.d("actores","El actor se ha borrado con éxito")
8.             }else{
9.                 navegarError("El api no ha podido borrar el actor")
10.            }
11.        }catch(e:Exception){
12.            navegarError("No se pudo borrar el actor: ${e.message}")
13.        }
14.    }
15. }

```

Todos estos métodos comprueban que los métodos de **ActoresRepository** funcionan correctamente. A continuación, vamos a meter en el **view model** la lista de actores que se verá en la app y haremos un método para rellenarla

- Abre el archivo **ActoresFragmentViewModel** y añade una variable de instancia, que será la lista de actores con la que se trabajará en **ActoresFragment**

```
1. class ActoresFragmentViewModel : ViewModel() {
2.     var listaActores:List<Actor> = emptyList()
3. }
```

- Añade a **ActoresFragmentViewModel** un método, punto de suspensión de corrutinas, llamado **rellenarListaActores** que use el método **consultarActores** de **ActoresRepository** para llenar la lista con todos los actores (se pasará null al método **consultarActores** para ello). Para programarlo, se usará un diseño basado en programación funcional, que significa:
 - o Si todo va bien, el método ejecutará una expresión lambda llamada **lambdaExito**
 - o Si se produce un fallo, el método ejecutará otra expresión lambda llamada **lambdaError**, que procesará el mensaje de error producido.
 - o Ambas expresiones lambda se pasan como parámetro al método **rellenarListaActores**

```
1. suspend fun rellenarListaActores(
2.     lambdaExito:() -> Unit,
3.     lambdaError:(String) -> Unit,
4. ){
5.     try{
6.         listaActores=ActoresRepository().consultarActores(null)
7.         lambdaExito()
8.     }catch(e:Exception){
9.         lambdaError("Error al consultar la lista de actores: ${e.message}")
10.    }
11. }
```

☞ **¿Por qué un diseño de programación funcional?** Podríamos haber optado por un diseño orientado a objeto, que hubiera sido simplemente un método que devuelva la lista de actores, o lance una excepción en caso de error:

```
suspend fun rellenarListaActores(patron:String?):List<Actor>
```

Sin embargo, no lo hemos hecho así por un motivo: llamar a **consultarActores** es una operación asíncrona, lo que significa que tarda un tiempo en completarse y se realiza en segundo plano dentro de una corrutina. O sea, no se sabe en qué momento está disponible la lista de actores. Con un enfoque orientado a objetos sería necesario sincronizar las corrutinas para que la principal, esperase a que la lista de actores esté disponible, y eso complica el código de la app. Por ese motivo, es más sencillo pasar a **rellenarListaActores** el código que se ejecutará en caso de éxito o de error, para que sea dicho método quien lo invoque.

- Modifica el método **rellenarListaActores** para que el parámetro **lambdaExito** tome por defecto una expresión lambda que no haga nada y el parámetro **lambdaError** tome por defecto una lambda que muestre error en Logcat

```

1. suspend fun rellenarListaActores(
2.     lambdaExito:() -> Unit = {},
3.     lambdaError:(String) -> Unit = {mensaje ->
4.         Log.d("ERROR","Error al consultar los actores: $mensaje")}
5. ){
6.     try{
7.         listaActores=ActoresRepository().consultarActores(patron)
8.         lambdaExito()
9.     }catch(e:Exception){
10.        lambdaError("Error al consultar la lista de actores: ${e.message}")
11.    }
12. }

```

Kotlin permite asignar un valor por defecto a los últimos parámetros de un método, de manera que si estos se omiten cuando luego se llama al método, esos parámetros toman el valor por defecto. En nuestro caso, si llamamos a `rellenarListaActores` con `lambdaExito`, entonces `lambdaError` se convierte en una lambda que muestra el error en Logcat

- Añade en `ActoresFragment` un método llamado `testRellenarListaActores`, que compruebe que el método `rellenarListaActores` funciona correctamente

```

1. fun testRellenarListaActores(){
2.     lifecycleScope.launch {
3.         viewModel.rellenarListaActores(
4.             { Log.d("testRellenarListaActores","Lista de actores cargada: ${viewModel.listaActores}") },
5.             { error -> Log.d("testRellenarListaActores","Error al cargar la lista de actores: $error") }
6.         )
7.     }
8. }

```

La expresión lambda que procesa el error no es necesaria en este ejemplo y la podemos quitar, porque `lambdaError` toma por defecto una lambda que ya usa Logcat

- Para que la llamada al método anterior quede más clara, incluye el nombre de los parámetros en la llamada al método

```

1. fun testRellenarListaActores(){
2.     lifecycleScope.launch {
3.         viewModel.rellenarListaActores(
4.             lambdaExito = { Log.d("testRellenarListaActores","Lista de actores cargada:
5.                             ${viewModel.listaActores}") },
6.             lambdaError = { error -> Log.d("testRellenarListaActores","Error al cargar la lista
7.                             de actores: $error") }
8.         )
9.     }
10. }

```

Cuando se llama a un método, Kotlin permite añadir el nombre de los parámetros a la llamada, de forma que así se vea más claro qué es lo que hace cada valor pasado al método. En nuestro caso, si solo pasamos las lambdas, el código es más difícil de leer porque no se sabe lo que hace cada una. Al añadir el nombre de los parámetros, el código se vuelve más legible y además, el método se vuelve más fácil de usar.

Cuando se añade el nombre de los parámetros, se les puede cambiar el orden. O sea, podemos pasar primero `lambdaError` y luego `lambdaExito` si ponemos los nombres.

- Una vez que has comprobado que todos los métodos de **ActoresRepository** funcionan correctamente, puedes borrar las llamadas que has insertado en **onCreateView** (incluso puedes borrar los métodos de test si quieres)
- Añade a **ActoresFragment** un método que permita hacer visible o invisible la **MaterialToolbar** de **MainActivity**, según un parámetro **Boolean** que reciba

```

1. fun mostrarToolbar(b:Boolean){
2.     val mainActivity = activity as MainActivity
3.     mainActivity.binding.materialToolbar.visibility =
4.         if(b) View.VISIBLE else View.GONE
5. }
```

5 – Splash screen de carga

En nuestra app la primera consulta al API de actores se va a realizar dentro de una **splash screen**, que mostrará una barra de progreso indeterminada (da vueltas cíclicamente) mientras dura el proceso

Para realizarla, haremos un diseño de **ActoresFragment** en capas, de forma que haya un **FrameLayout** que tenga una primera capa con la splash screen, que se haga invisible cuando la carga se haya finalizado, y en ese momento, se muestre otra segunda capa con el **RecyclerView** de resultados y la interfaz principal. Una variable del **view model** controlará cuál de las dos capas debe mostrarse.

- Abre **fragment_actores** con la vista xml, borra todo lo que contiene y añade un **FrameLayout** que contenga dentro un **LinearLayout** (sería la capa 1) con el diseño de la splash screen:
 - **TextView**
 - Letras de tamaño 40sp, negrita, centrado y color blanco
 - Icono a la izquierda (**android:drawableLeft**) **@drawable/cinema**
 - **ImageView**
 - Anchura: la de su contenedor
 - Altura: **0dp** (para que así se extienda lo que indique su peso)
 - Peso: 1
 - Image de fondo: **@drawable/fondosplash**
 - **ProgressBar**
 - Anchura: la de su contenedor
 - Altura: **60dp**
- Añade a continuación otro **LinearLayout** (sería la capa 2) que esté **invisible**, y que tenga el diseño de la interfaz principal este **RecyclerView**
 - **RecyclerView**
 - Id: **listActores**
 - Anchura y altura: La de su contenedor
 - layoutManager: **GridLayoutManager**
 - Elementos por fila (**app:spanCount**): 3

```

1. <FrameLayout
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">
6.     <LinearLayout
7.         android:id="@+id/capa1"
8.         android:layout_width="match_parent"
9.         android:layout_height="match_parent"
10.        android:background="@color/black"
11.        android:orientation="vertical">
12.            <TextView
13.                android:layout_width="match_parent"
14.                android:layout_height="wrap_content"
15.                android:drawableLeft="@drawable/cinema"
16.                android:gravity="center"
17.                android:textColor="@color/white"
18.                android:textSize="40sp"
19.                android:textStyle="bold"
20.                android:text="TODO ACTORES"/>
21.            <ImageView
22.                android:layout_width="wrap_content"
23.                android:layout_height="0dp"
24.                android:layout_weight="1"
25.                android:scaleType="fitXY"
26.                android:src="@drawable/fondosplash" />
27.            <ProgressBar
28.                android:layout_width="match_parent"
29.                android:layout_height="60dp"/>
30.        </LinearLayout>
31.        <LinearLayout
32.            android:id="@+id/capa2"
33.            android:orientation="vertical"
34.            android:visibility="gone"
35.            android:layout_width="match_parent"
36.            android:layout_height="match_parent">
37.            <com.google.android.material.appbar.MaterialToolbar
38.                android:id="@+id/material_toolbar"
39.                android:layout_width="match_parent"
40.                android:layout_height="?attr/actionBarSize"
41.                style="@style/Widget.MaterialComponents.Toolbar.Primary"/>
42.            <androidx.recyclerview.widget.RecyclerView
43.                android:id="@+id/lstActores"
44.                android:layout_width="match_parent"
45.                android:layout_height="match_parent"
46.                app:layoutManager="androidx.recyclerview.widget.GridLayoutManager"
47.                app:spanCount="3"/>
48.        </LinearLayout>
49.    </FrameLayout>

```

- Añade a **ActoresFragmentViewModel** una variable de instancia **Boolean** que valga **true** si hay que mostrar la capa1 (splash screen) y **false** en caso contrario

```

1. class ActoresFragmentViewModel : ViewModel() {
2.     var mostrarSplashScreen:Boolean = true
3.     // resto omitido
4. }

```

- Abre **ActoresFragment** y añade cinco métodos llamados **inicializarInterfaz**, **mostrarSplashScreen**, **inicializarInterfazPrincipal**, e **inicializarRecyclerView**
- Programa el método **inicializarInterfaz**, para que, según el valor de la variable **mostrarSplashScreen**, llame a **inicializarSplashScreen** o a **inicializarInterfazPrincipal**

```

1. class ActoresFragment : 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.         inicializarInterfaz()
9.         return binding.root
10.    }
11.    fun inicializarInterfaz(){
12.        if(viewModel.mostrarSplashScreen){
13.            inicializarSplashScreen()
14.        }else{
15.            inicializarInterfazPrincipal()
16.        }
17.    }
18.    fun inicializarSplashScreen(){
19.        // aquí haremos visible la splash screen y cargaremos los actores
20.    }
21.    fun inicializarInterfazPrincipal(){
22.        // aquí haremos visible la interfaz principal
23.    }
24.    fun inicializarRecyclerView(){
25.        // aquí inicializaremos el RecyclerView
26.    }
27. }

```

- Programa el método **inicializarSplashScreen** para que lance una corriputina que oculte la toolbar, cargue los actores, y tras haberlo hecho, llame a **inicializarInterfazPrincipal**

```

1. fun inicializarSplashScreen(){
2.     lifecycleScope.launch {
3.         mostrarToolbar(false)
4.         viewModel.rellenarListaActores(
5.             lambdaExito = { inicializarInterfazPrincipal() },
6.             lambdaError = { mensajeError -> navegarError(mensajeError) }
7.         )
8.     }
9. }

```

- Programa **inicializarInterfazPrincipal** para que haga invisible la **capa1**, visible la toolbar y la **capa2** y ponga a **false** la variable **mostrarSplashScreen**. A continuación, llamará a **inicializarToolbar** e **inicializarRecyclerView**

```

1. fun inicializarInterfazPrincipal(){
2.     mostrarToolbar(true)
3.     binding.capa1.visibility=View.GONE
4.     binding.capa2.visibility=View.VISIBLE
5.     inicializarRecyclerView()
6.     viewModel.mostrarSplashScreen=false // será importante poner esta línea la última
7. }

```

- Inicia la app y comprueba que aparece la interfaz principal (ahora mismo vacía) de **ActoresFragment**

Como la API está funcionando en local, es posible que la splash screen no aparezca porque la consulta al API es muy rápida y tarda muy poco tiempo en completarse. Si quieras el splash screen, añade una llamada a **delay(1000)** en **inicializarSplashScreen**, justo tras ocultar la toolbar y antes de cargar la lista de actores.

6 – RecyclerView con datos remotos

El **RecyclerView** puede ser rellenado con datos obtenidos del API de actores, de la forma habitual, programando las correspondientes clases **ActorHolder** y **ActorAdapter**

- Pulsa el botón derecho en **res/layout** y elige **new → layout resource file**
- Crea un archivo llamado **actor.xml**, ábrelo con la vista xml, borra lo que hay y deja como elemento principal un **ImageView** de altura **120dp**, con altura, la de su contenedor, e id **imgActor** que muestre la imagen **desconocido.png** (que debe estar en la carpeta **drawable**)

```
1. <ImageView  
2.     xmlns:android="http://schemas.android.com/apk/res/android"  
3.     xmlns:app="http://schemas.android.com/apk/res-auto"  
4.     android:id="@+id/imgActor"  
5.     android:src="@drawable/desconocido"  
6.     android:scaleType="fitXY"  
7.     android:layout_height="120dp"  
8.     android:layout_width="match_parent"/>
```

- En el paquete **vista** crea la siguiente clase **ActorHolder**, cuya misión es portar un objeto **Actor** y su correspondiente vista **actor.xml**

```
1. class ActorHolder(val binding:ActorBinding) : RecyclerView.ViewHolder(binding.root) {  
2.     lateinit var actor: Actor  
3. }
```

- Añade a **ActorHolder** un método **mostrarActor** que reciba un **Actor** y lo guarde en la variable de instancia **actor**

```
1. fun mostrarActor(a:Actor) {  
2.     actor = a  
3. }
```

- En el paquete **modelo** crea un archivo llamado **Funciones** y usa el mecanismo de **extension functions** de Kotlin para añadir a la clase **ImageView** un método llamado **setFotoActor**, que reciba un objeto **Actor** y ponga su foto en él.

```
1. fun ImageView.setFotoActor(a:Actor){  
2.     /* aqui podemos usar this para referirnos al ImageView  
3.     y poner la foto del actor en él */  
4. }
```

Kotlin permite añadir métodos a una clase ya hecha mediante las **extension functions**.

Al escribir **ImageView.setFotoActor** estamos añadiendo a la clase **ImageView** un método llamado **setFotoActor**, que se programa usando **this** para referirnos al **ImageView** y llamar a sus métodos.

- Completa el método **setFotoActor** para poner sobre el **ImageView** (objeto **this**) la foto del actor pasado como parámetro. Se mirará primero si el actor lleva ya la foto codificada en Base 64 en su variable **fotoCodif**. En dicho caso, usará dicha foto, y si no, usará la foto de su variable **fotoUrl**. Si no hay ninguna foto disponible, se pondrá una imagen por defecto.

```

1. fun ImageView.setFotoActor(a:Actor){
2.     val imagen:Any = when{
3.         a.fotoCodif.isNotEmpty() -> Base64.Default.decode(a.fotoCodif)
4.         a.fotoUrl.isNotEmpty() -> a.fotoUrl
5.         else -> R.drawable.desconocido
6.     }
7.     this.load(imagen) {
8.         this.crossfade(true)
9.         this.placeholder(R.drawable.loading)
10.        this.error(R.drawable.error)
11.    }
12. }
```

La librería **Coil** añade a la clase **ImageView** el método **load**, que carga en segundo plano la imagen que se pasa como parámetro (admite cualquier cosa, como un entero con el id de un drawable, una url, un array de bytes, etc)

El método **load** recibe como segundo parámetro una expresión lambda que sirve para configurar el proceso de carga, con estos métodos:

- placeholder** → imagen que se muestra durante la carga de la foto
- error** → imagen que se muestra en caso de que haya error en la carga
- crossfade** → activa una animación para pasar del placeholder a la foto

- Completa el método **mostrarActor** de **ActorHolder** haciendo que se muestre la foto del actor, usando el método que hemos añadido a la clase **ImageView**

```

1. class ActorHolder(val binding: ActorBinding) : RecyclerView.ViewHolder(binding.root) {
2.     lateinit var actor: Actor
3.     fun mostrarActor(a:Actor) {
4.         actor = a
5.         binding.imgActor.setFotoActor(actor)
6.     }
7. }
```

- En el paquete **vista** crea la siguiente clase **ActorAdapter**, cuya misión es definir la lista de actores que va a mostrarse en el **RecyclerView**, crear los **ActorHolder**, asignar un **Actor** a un **ActorHolder** y asignar una expresión lambda que se ejecutará al pulsar un **ActorHolder**

```

1. class ActorAdapter(
2.     var actores:List<Actor>,
3.     val lambda:(ActorHolder)->Unit
4. ):RecyclerView.Adapter<ActorHolder>(){
5.     override fun getItemCount(): Int = actores.size
6.     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ActorHolder {
7.         val inflater = LayoutInflater.from(parent.context)
8.         val binding = ActorBinding.inflate(inflater,parent,false)
9.         return ActorHolder(binding)
```

```

10.    }
11.    override fun onBindViewHolder(holder: ActorHolder, position: Int) {
12.        val actor = actores.get(position)
13.        holder.mostrarActor(actor)
14.        holder.binding.root.setOnClickListener{
15.            lambda(holder)
16.        }
17.    }
18. }

```

- Abre el código de **ActoresFragment** y programa el método **inicializarRecyclerView** para que ponga al **RecyclerView** un **ActorAdapter** con la lista de actores que hay en el view model.

```

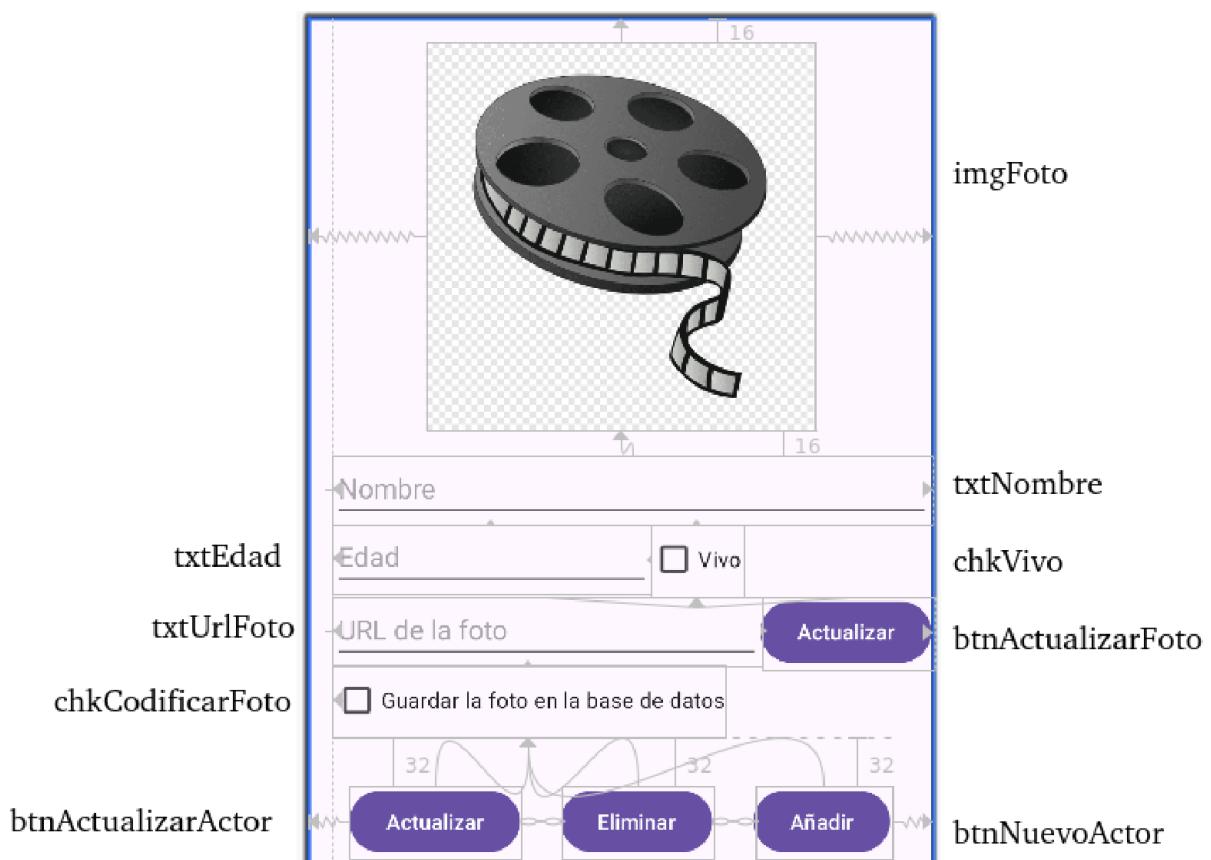
1. fun inicializarRecyclerView(){
2.     binding.lstActores.adapter = ActorAdapter(viewModel.actores){ holder ->
3.         Log.d("actores","Pulsado sobre el actor ${holder.actor.nombre}")
4.     }
5. }

```

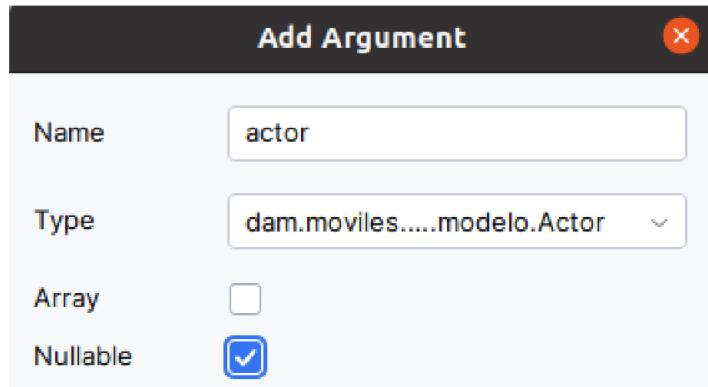
- Ejecuta la app y comprueba que el **RecyclerView** se rellena con las fotos de los actores

7 – Formularios adaptables

- Abre **fragment_detalle_actor.xml** y diseña la siguiente interfaz, usando el diseñador de Android Studio



- Abre el `nav_graph` y haz que `detalleActorFragment` tenga como argumento un objeto **Serializable** llamado `actor`, que pueda tomar el valor `null`



La pantalla `fragment_detalle_actor` hace dos cosas en una. Si el **Actor** recibido es `null`, esta pantalla servirá para dar de alta un nuevo actor y solo se mostrará el botón de añadir. En cambio, si el **Actor** recibido no es `null`, la pantalla servirá para modificarlo y solo se mostrarán los otros dos botones.

Este tipo de pantallas, que cambian su comportamiento según los parámetros que reciban, se llaman **formularios adaptables**

- Vete a `ActoresFragment` y crea un método llamado `navegarDetalleActor`, que reciba un objeto **Actor** (con la posibilidad de que sea nulo) y navegue hacia `FragmentDetalleActor` pasando como argumento dicho actor.

```
1. fun navegarDetalleActor(a:Actor?){
2.     val flecha = ActoresFragmentDirections.actionActoresFragmentToDetalleActorFragment(a)
3.     findNavController().navigate(flecha)
4. }
```

- Modifica el método `inicializarRecyclerView` para que al pulsar en un **ActorHolder** se navegue hacia la pantalla de detalle de actor, pasándole como parámetro el objeto **Actor** contenido en el **ActorHolder**

```
1. fun inicializarRecyclerView(){
2.     lifecycleScope.launch {
3.         val actores:List<Actor> = ActoresRepository().consultarTodosActores();
4.         binding.lstActores.adapter = ActorAdapter(actores){ holder ->
5.             navegarDetalleActor(holder.actor)
6.         }
7.     }
8. }
```

- Añade a `DetalleActorFragment` una variable de instancia llamada `idActorModificado` de tipo `Int?` que sea el id del actor que se desea modificar, o `null` en caso de que la pantalla esté en modo de creación de actor.

```
1. class DetalleActorFragment : Fragment(), NavegadorError {
2.     var idActorModificado:Int? = null
3.     // resto de la clase omitido
4. }
```

- En **DetalleActorFragment** añade un método llamado **inicializarInterfaz**, que recogerá el objeto **Actor?** recibido desde **ActoresFragment** y si no es null, rellenará con sus datos todos los elementos de la interfaz, mostrando los botones de actualizar y borrar. En caso de que sea null, no rellenará nada y solo mostrará el botón de crear un nuevo actor

```

1. fun inicializarInterfaz(){
2.     val actor:Actor? = DetalleActorFragmentArgs.fromBundle(requireArguments()).actor
3.     if(actor!=null) {
4.         idActorModificado = actor.id
5.         binding.txtNombre.setText(actor.nombre)
6.         binding.txtEdad.setText(actor.edad.toString())
7.         binding.chkVivo.isChecked = actor.vivo==1
8.         binding.txtUrlFoto.setText(actor.fotoUrl)
9.         binding.chkCodificarFoto.isChecked=actor.fotoCodif.isNotEmpty()
10.        binding.imgFoto.setFotoActor(actor)
11.        binding.btnNuevoActor.visibility = View.GONE
12.    }else {
13.        binding.btnNuevoActor.visibility = View.VISIBLE
14.    }
15.    binding.btnActualizar.visibility =
16.        if (binding.btnNuevoActor.visibility==View.GONE) View.VISIBLE else View.GONE
17.    binding.btnDelete.visibility = binding.btnActualizar.visibility
18. }
```

- Llama al método **inicializarInterfaz** dentro de **onCreateView**, justo después de inicializar el **view binding**.
- Ejecuta la app y comprueba que al pulsar un actor se navega hacia su pantalla de detalle, mostrando sus datos y los botones de actualizar y eliminar
- Añade a **DetalleActorFragment** el método **inicializarBotonActualizarActor** y llama allí al método **setOnClickListener** de **btnActualizarActor**, para que **ActoresRepository** modifique el actor cuyo id está en **idActorModificado**

```

1. fun inicializarBotonActualizarActor(){
2.     binding.btnActualizarActor.setOnClickListener {
3.         lifecycleScope.launch {
4.             try {
5.                 val respuesta = ActoresRepository().actualizarActor(
6.                     checkNotNull(idActorModificado),
7.                     binding.txtNombre.text.toString(),
8.                     binding.txtEdad.text.toInt(),
9.                     if (binding.chkVivo.isChecked) 1 else 0,
10.                     binding.txtUrlFoto.text.toString(),
11.                     if (binding.chkCodificarFoto.isChecked) getFotoCodificada() else 0,
12.                 )
13.                 if (respuesta.info.contains("actualizado")) {
14.                     findNavController().popBackStack()
15.                 } else {
16.                     navegarError("El api no pudo modificar el actor")
17.                 }
18.             }catch(e:Exception){
19.                 navegarError("Error al consultar el api: ${e.message}")
20.             }
21.         }
22.     }
23. }
```

El método **popBackStack** del **navigation controller** vuelve a la pantalla previa

- Programa el método **getFotoCodificada**, que en caso de que haya una url escrita en **txtUrlFoto**, la descargará y la devolverá codificada en base 64.

```

1. suspend fun getFotoCodificada():String{
2.     var resultado=""
3.     val urlFoto = binding.txtUrlFoto.text.toString()
4.     if(urlFoto.isNotEmpty()){
5.         try {
6.             lifecycleScope.launch(Dispatchers.IO){
7.                 val bytes:ByteArray = URL(urlFoto).readBytes()
8.                 resultado = Base64.Default.encode(bytes)
9.             }.join()
10.        }catch(e:IOException){
11.            Log.d("actores","No se pudo descargar la foto: ${e.message}")
12.        }
13.    }
14.    return resultado
15. }
```

El método **readBytes** no puede ser llamado desde el hilo main por motivos de seguridad (puede bloquear la interfaz porque tarda tiempo en completarse) y por ese motivo, debe ser lanzado en una corutina que se ejecute en un hilo diferente

El método **join** hace que el hilo que lanza la corutina espere su finalización

- Añade a **DetalleActorFragment** el método **inicializarBotonActualizarFoto**, y llama allí al método **setOnClickListener** de **btnActualizarFoto** para actualizar **imgFoto** con la foto cuya url esté escrita en **txtUrlFoto**

```

1. private fun inicializarBotonActualizarFoto() {
2.     binding.btnActualizarFoto.setOnClickListener {
3.         binding.imgFoto.setFotoActor(
4.             Actor(0,"",0,0,binding.txtUrlFoto.text.toString(),""))
5.     }
6. }
```

Para usar el efecto de animación y carga en segundo plano, llamamos al método **setFotoActor** pasando un “actor de mentira” cuya foto es la URL escrita en **txtUrlFoto**

- Añade a **DetalleActorFragment** el método **inicializarBotonBorrarActor** y llama allí al método **setOnClickListener** de **btnBorrarActor**, para que **ActoresRepository** borre el actor cuyo id está en **idActorModificado**

```

1. fun inicializarBotonBorrarActor(){
2.     binding.btnEliminar.setOnClickListener {
3.         lifecycleScope.launch {
4.             try {
5.                 val respuesta = ActoresRepository().borrarActor.checkNotNull(idActorModificado))
6.                 if (respuesta.info.contains("éxito")) {
7.                     findNavController().popBackStack()
8.                 } else {
9.                     navegarError("El api no pudo borrar el actor")
10.                }
11.            }catch(e:Exception){
12.                navegarError("Error al consultar el api: ${e.message}")
13.            }
14.        }
15.    }
16. }
```

- Añade a **DetalleActorFragment** el método **inicializarBotonNuevoActor** y llama allí al método **setOnClickListener** de **btnNuevoActor**, para que **ActoresRepository** cree un nuevo actor cuyos datos estén en los componentes de la interfaz

```

1. fun inicializarBotonNuevoActor(){
2.     binding.btnNuevoActor.setOnClickListener {
3.         lifecycleScope.launch {
4.             try {
5.                 val respuesta = ActoresRepository().insertarActor(
6.                     binding.txtNombre.text.toString(),
7.                     binding.txtEdad.text.toString().toInt(),
8.                     if (binding.chkVivo.isChecked) 1 else 0,
9.                     binding.txtUrlFoto.toString(),
10.                    if(binding.chkCodificarFoto.isChecked) getFotoCodificada() else ""
11.                )
12.                if (respuesta.info.contains("Creado!")) {
13.                    findNavController().popBackStack()
14.                } else {
15.                    navegarError("El api no pudo crear el actor")
16.                }
17.            }catch(e:Exception){
18.                navegarError("Error al consultar el api: ${e.message}")
19.            }
20.        }
21.    }
22. }
```

- Añade a **DetalleActorFragment** un método llamado **inicializarBotones** que llame a los cuatro métodos de inicializar botones que has hecho, y llama a **inicializarBotones** en **onCreateView** tras inicializar el binding

```

1. fun inicializarBotones(){
2.     inicializarBotonNuevoActor()
3.     inicializarBotonActualizarActor()
4.     inicializarBotonBorrarActor()
5.     inicializarBotonActualizarFoto()
6. }
```

7 – Toolbar dinámica

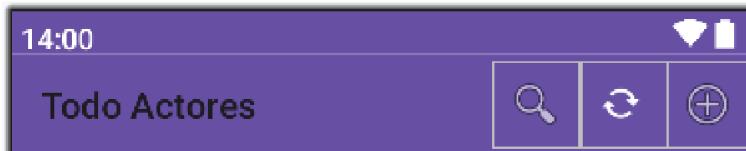
En nuestra app la toolbar está en **MainActivity**, pero cambia de opciones según el **Fragment** en el que se encuentre. Esto se conoce como **toolbar dinámica**.

Para implementar una **toolbar dinámica** programaremos dos métodos en cada **Fragment** llamados **iniciarOpcionesToolbar** y **retirarOpcionesToolbar**, que le den y quiten la configuración que necesita. Esos métodos se llamarán en **onCreateView** y **onDestroyView**.

Puesto que la configuración de la **toolbar** se define en un objeto de tipo **MenuProvider**, para poder quitarlo es necesario tenerlo guardado en una variable de instancia en el **Fragment**

- Pulsa el botón derecho sobre **res** y elige **new → android resource file**
- En la ventana que aparece crea un archivo de tipo **Menu** llamado **menu_toolbar**

- Abre el archivo **menu_toolbar.xml** con el diseñador, y arrastra hacia interior, en este orden, estos elementos:
 - Un **MenuItem** con id **btnInsertarActor**, la opción **showAsAction** puesta en **always** y el icono **@android:drawable/ic_menu_add**
 - Un **MenuItem** con id **btnActualizarDatos**, la opción **showAsAction** puesta en **always** y el icono **@android:drawable/stat_notify_sync**
 - Un **SearchView** con id **txtBuscador**, la opción **showAsAction** puesta en **always** y el icono **@android:drawable/ic_menu_search**



Para poner a la toolbar el menú que hemos diseñado y configurar el comportamiento de los botones es necesario crear un objeto que implemente la interfaz **MenuProvider** y ponérselo al objeto **MainActivity**

- Añade a **ActoresFragment** una variable de instancia llamada **menuProvider**, que contendrá un objeto que implemente **MenuProvider**. Dicho objeto sobreescribirá los métodos abstractos de **MenuProvider** de esta forma:
 - **onCreateMenu** → coloca al menú recibido como parámetro la vista definida en **menu_toolbar.xml**
 - **onMenuItemSelected** → Mirará el id del **MenuItem** pulsado y hará esta acción, según el caso:
 - pulsado **btnInsertarActor** → navegará hacia **DetalleActorFragment** pasando como parámetro **null**
 - pulsado **btnActualizarDatos** → llamará a **inicializarRecyclerView**, para que así se recarguen los datos

```

1. class ActoresFragment : Fragment(), NavegadorError {
2.     val menuProvider = object:MenuProvider{
3.         override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
4.             menuInflater.inflate(R.menu.menu_toolbar,menu)
5.         }
6.         override fun onMenuItemSelected(menuItem: MenuItem): Boolean {
7.             when(menuItem.itemId){
8.                 R.id.btnInsertarActor -> {
9.                     val accion = ActoresFragmentDirections
10.                         .actionActoresFragmentToDetalleActorFragment(null)
11.                         findNavController().navigate(accion)
12.                 }
13.                 R.id.btnActualizarDatos -> inicializarRecyclerView()
14.             }
15.             return true
16.         }
17.     }
18.     // resto de la clase omitido
19. }
```

- Añade dos métodos llamados **iniciarOpcionesToolbar** y **retirarOpcionesToolbar** en los que añadas y retires, el objeto **menuProvider** al objeto **MainActivity**

```

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

- Llama a los métodos anteriores en **onCreateView** y **onDestroyView**

```

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

- Ejecuta la app y comprueba que el menú se muestra correctamente en **ActoresFragment** y que al pulsar un actor se borra cuando aparece **DetalleActorFragment**
- Comprueba también que ya es posible crear nuevos actores

El **DetalleActorFragment** no tiene opciones propias, así que no necesita un **MenuProvider** para incializarlo. Solo tendrá un ícono (que aparece a la izquierda del título de la app) que al pulsarlo volverá hacia atrás.

- Abre **DetalleActorFragment** y añade los siguientes métodos:
 - **iniciarOpcionesToolbar** → pone un botón de atrás como ícono de la toolbar y al pulsarlo, navega a la pantalla previa.
 - **retirarOpcionesToolbar** → pone a **null** el ícono de la toolbar

```

1. fun iniciarOpcionesToolbar(){
2.     val mainActivity = activity as MainActivity
3.     mainActivity.binding.materialToolbar.navigationIcon =
4.         resources.getDrawable(R.drawable.ic_menu_revert)
5.     mainActivity.binding.materialToolbar.setNavigationOnClickListener {
6.         findNavController().popBackStack()
7.     }
8. }
9. fun retirarOpcionesToolbar() {
10.    val mainActivity = activity as MainActivity
11.    mainActivity.binding.materialToolbar.navigationIcon=null
12. }
```

La **MaterialToolbar** tiene una propiedad **navigationIcon** y un método **setNavigationOnClickListener** con los que podemos configurar su ícono y lo que pasa al pulsarlo.

- Llama a los métodos anteriores en **onCreateView** y **onDestroyView**
- Ejecuta la app y comprueba que al pulsar en un actor, aparece la pantalla de detalle y la toolbar muestra un botón para volver atrás, que funciona correctamente.

8 – SearchView simple

Vamos a hacer que al escribir un texto en el **SearchView** de la toolbar, se muestre el listado de actores cuyo nombre contiene dicho texto. Lo ideal es que conforme se escribe, se muestren sugerencias, pero eso es bastante más complejo.

- Añade a **ActoresFragmentViewModel** una variable de instancia llamada **patrón**, que será el texto que se desea buscar para el nombre de los actores. Si vale **null** es porque se buscan todos los actores.

```
1. class ActoresFragmentViewModel : ViewModel(){
2.     var patron:String? = null
3.     // resto omitido
4. }
```

- Actualiza el método **rellenarListaActores** para que utilice la variable **patrón** del view model

```
1. suspend fun rellenarListaActores(
2.     lambdaExito:() -> Unit,
3.     lambdaError:(String) -> Unit
4. ){
5.     try{
6.         listaActores=ActoresRepository().consultarActores(patron)
7.         lambdaExito()
8.     }catch(e:Exception){
9.         lambdaError("Error al consultar la lista de actores: ${e.message}")
10.    }
11. }
```

- Añade en **ActoresFragment** un método llamado **actualizarListaActores** que llame al método **rellenarListaActores** del view model, y que en caso de que todo vaya bien, actualice el **RecyclerView**, y si se produce un error, navegue hacia la pantalla de error.

```
1. fun actualizarListaActores(){
2.     lifecycleScope.launch {
3.         viewModel.rellenarListaActores(
4.             lambdaExito = { inicializarRecyclerView()},
5.             lambdaError = { mensaje -> navegarError(mensaje)}
6.         )
7.     }
8. }
```

- Modifica el método **inicializarInterfazPrincipal** para que cuando este no se llame desde la splash screen (por ejemplo, al crear un actor, girar el móvil, pulsar el botón de refrescar de la toolbar, etc) se llame al método **actualizarListaActores**

```

1. fun inicializarInterfazPrincipal(){
2.     mostrarToolbar(true)
3.     binding.capa1.visibility=View.GONE
4.     binding.capa2.visibility=View.VISIBLE
5.     if(!viewModel.mostrarSplashScreen){
6.         // no venimos de la splash screen
7.         actualizarListaActores()
8.     }else {
9.         // venimos de la splash screen
10.        inicializarRecyclerView()
11.        viewModel.mostrarSplashScreen = false
12.    }
13. }
```

- Vete al método **onCreateMenu** del **MenuProvider** y usa el método **findItem** del objeto **menu** para recuperar el objeto **txtBuscador**

```

1. val menuProvider = object:MenuProvider{
2.     override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
3.         menuInflater.inflate(R.menu.menu_toolbar,menu)
4.         val txtBuscador:SearchView = menu.findItem(R.id.txtBuscador).actionView as SearchView
5.     }
6.     // resto omitido
7. }
```

- A continuación, llama al método **setQuery** de **txtBuscador** para pasarle como parámetro un objeto que implemente **OnQueryTextListener**. Dicha interfaz tiene dos métodos abstractos que sobreescribirás así:
 - o **onQueryTextSubmit** → Se llama cuando se escribe algo en el buscador y se pulsa enviar. Aquí actualizaremos la variable **patrón** del view model y llamaremos a **actualizarListaActores** para recargar la lista de actores con el nuevo patrón y mostrarla en el **RecyclerView**
 - o **onQueryTextChange** → Se llama cada vez que cambia el texto escrito en el buscador y es útil para obtener las sugerencias y mostrarlas. No haremos nada con él en este proyecto.

```

1. val menuProvider = object:MenuProvider{
2.     override fun onCreateMenu(menu: Menu, menuInflater: MenuInflater) {
3.         menuInflater.inflate(R.menu.menu_toolbar,menu)
4.         val txtBuscador:SearchView = menu.findItem(R.id.txtBuscador).actionView as SearchView
5.         txtBuscador.setOnQueryTextListener(
6.             object:OnQueryTextListener{
7.                 override fun onQueryTextSubmit(query: String?): Boolean {
8.                     viewModel.patron = query
9.                     actualizarListaActores()
10.                    txtBuscador.setQuery("",false)
11.                    txtBuscador.setIconified = true
12.                    return true
13.                }
14.                override fun onQueryTextChange(newText: String?): Boolean = true
15.            }
16.        )
17.    }
18.    // resto omitido
19. }
```

- Ejecuta la app y comprueba que el buscador funciona correctamente

9 – AutoCompleteTextView

Un **AutoCompetableTextView** es un cuadro de texto que es capaz de mostrar sugerencias cuando se escribe dentro de él.

Para que funcione correctamente, es necesario ponerle un **ArrayAdapter**, que es un objeto que permite obtener la lista de sugerencias a partir del texto que el **AutoCompetableTextView** tiene escrito en él.

A diferencia del **SearchView**, el **AutoCompetableTextView** no puede integrarse en la toolbar.

- Abre **fragment_actores.xml** y añade, al principio del **LinearLayout** de la **capa2**, un **AutoCompetableTextView** con estas características:
 - Id: **txtBuscador**
 - Anchura: la de su contenedor
 - Altura: la de su contenido
 - Admite una sola línea de texto (atributo **android:inputType="text"**)
 - Hint: Búsqueda por nombre
 - Margen: **8dp**
 - Icono (atributo **android:drawableLeft**): El recurso **drawable** definido en el sistema Android, llamado **ic_menu_search**

```
1. <AutoCompleteTextView  
2.     android:id="@+id/txtBuscador2"  
3.     android:inputType="text"  
4.     android:layout_margin="8dp"  
5.     android:drawableLeft="@android:drawable/ic_menu_search"  
6.     android:hint="Búsqueda por nombre"  
7.     android:layout_width="match_parent"  
8.     android:layout_height="wrap_content"/>
```

- Añade a **FragmentActores** un método llamado **inicializarBuscador2**, en el que se creará un **ArrayAdapter**, que es el objeto que contiene la lista de datos que tendrá en cuenta el **AutoCompleteTextView** (en nuestro caso, la lista de nombres de actores) y mostrar sugerencias a partir de ellos.

```
1. fun inicializarBuscador2(){  
2.     // obtenemos una lista con los nombres de los actores (sin que se repitan)  
3.     val nombresActores = viewModel.listaActores  
4.     .map{ actor -> actor.nombre}  
5.     .distinct()  
6.     // creamos un ArrayAdapter que muestra String  
7.     val adapter:ArrayAdapter<String> = ArrayAdapter(  
8.         requireContext(), // contexto de la app  
9.         android.R.layout.simple_spinner_item, // tipo de layout para las sugerencias  
10.        nombresActores // lista con los datos que usará el AutoCompleteTextView  
11.    )  
12.    // ponemos el adapter al AutoCompleteTextView  
13.    binding.txtBuscador2.setAdapter(adapter)  
14. }
```

Creamos un **ArrayAdapter** pasando como parámetros:

- o El **context** en el que se ejecuta la app
- o El layout que se usará para dibujar las sugerencias. En este caso indicamos un layout definido en el sistema Android que las muestra en forma de lista desplegable
- o La lista con los nombres de los actores, que será lo que el **AutoCompleteTextView** utilice para mostrar las sugerencias

- Si queremos que el **RecyclerView** se actualice cada vez que cambia el texto escrito en el **AutoCompletableEditText**, programaremos su método **doOnTextChanged**, al que pasaremos una expresión lambda que llamará al método **actualizarListaActores**

```
1. fun inicializarBuscador2(){  
2.     // inicio del método omitido  
3.     binding.txtBuscador2.doOnTextChanged { text, start, before, count ->  
4.         viewModel.patron=binding.txtBuscador2.text.toString()  
5.         actualizarListaActores()  
6.     }  
7. }
```

- Por último, llama a **inicializarBuscador2** dentro del método **inicializarInterfazPrincipal**, en el momento de iniciar la interfaz tras la splash screen

```
1. fun inicializarInterfazPrincipal(){  
2.     mostrarToolbar(true)  
3.     binding.capa1.visibility=View.GONE  
4.     binding.capa2.visibility=View.VISIBLE  
5.     if(!viewModel.mostrarSplashScreen){  
6.         // no venimos de la splash screen  
7.         actualizarListaActores()  
8.     }else {  
9.         // venimos de la splash screen  
10.        inicializarRecyclerView()  
11.        inicializarBuscador2()  
12.        viewModel.mostrarSplashScreen = false  
13.    }  
14. }
```