Base de datos con Room

Crearemos la aplicación **App_04_03**.

Vamos a crear una aplicación para gestionar una base de datos en **Room** de películas.

GRADLE

Antes de comenzar vamos a añadir las dependencias necesarias en nuestros fichero gradle.
En primer lugar, en el **gradle del proyecto** añade:

```
buildscript {
    dependencies {
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.5.3"
    }
}
```

En el **gradle del módulo** añadimos los plugins:

```
id 'kotlin-kapt'
id 'androidx.navigation.safeargs.kotlin'
```

Y las dependencias:

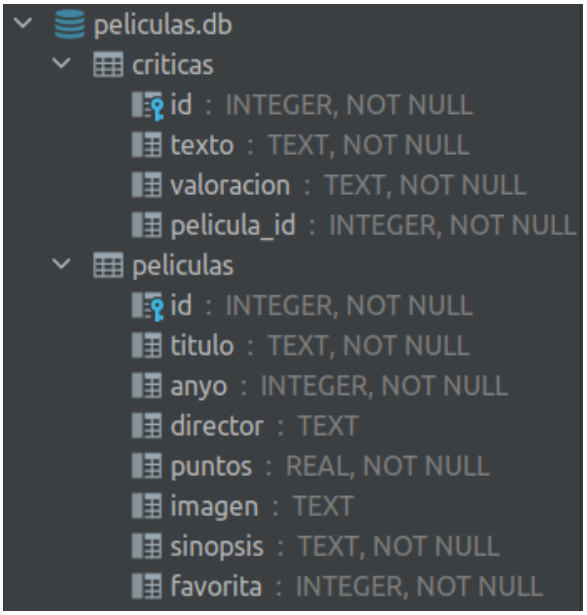
```
// LiveData
implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.5.1'
// ViewModel
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.5.1'
//Fragments
implementation 'androidx.fragment:fragment-ktx:1.5.5'

//Navegación
implementation "androidx.navigation:navigation-fragment-ktx:2.5.3"
implementation "androidx.navigation:navigation-ui-ktx:2.5.3"

//Coil
implementation "io.coil-kt:coil:2.2.2"

//Room
implementation "androidx.room:room-ktx:2.4.3"
kapt "androidx.room:room-compiler:2.4.3”
```

MANIFEST

Añade el permiso de Internet

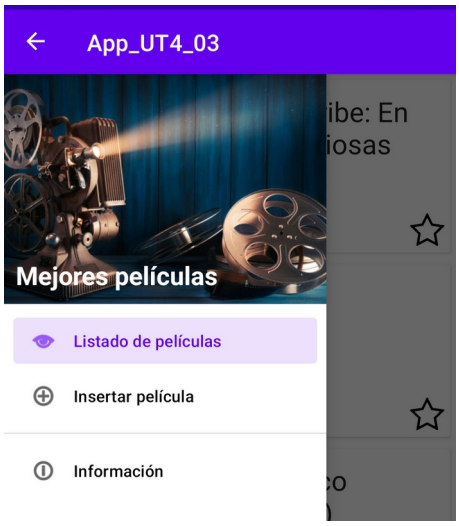
BASE DE DATOS

A lo largo de la aplicación se utiliza una base de datos con la siguiente estructura:



La tabla **peliculas** contiene información sobre películas: titulo, año, director, puntuación, URL de la imagen, sinopsis y si es favorita o no. Además, posee un identificador que coincide con el identificador de la película en Filmaffinity.

La tabla **criticas** tiene información sobre las críticas profesionales emitidas para una película. Contendrá un texto, una valoración (POSITIVA, NEGATIVA O NEUTRA) y el identificador de la película a la que pertenece la crítica. Además, tendrá un identificador numérico y autoincrementado.



Vemos entonces como existe una relación **uno a muchos** entre películas y críticas

VISTAS

En primer lugar vamos a crear dos **menús.** El primero de ellos lateral, al igual que en la hoja de ejercicios anterior. Contendrá 3 ítems como se puede ver en la imagen.
El archivo de recursos se llamará **menu_lateral.xml**

Por otro lado habrá un menú inferior con 2 ítems que llamaremos **menu_abajo.xml**

Al igual que en la hoja de ejercicios anterior, crearemos un layout para la cabecera del menú lateral. Dentro de la carpeta layout creamos un fichero llamado **cabecera_navegacion.xml**

Crearemos un **gráfico de navegación** como en los ejercicios anteriores llamado **grafico_navegacion.xml**

En el **activity_main.xml** crearemos un elemento **DrawerLayout** con un **NavigationView**. Además, habrá un **ConstraintLayout**. Este layout tendrá un **FrameContainerView** que será el contenedor de Fragments de nuestra aplicación y un **BottomNavigationView** para mostrar el menú inferior.

Para que el menú funcione tenemos que configurarlo en clase **MainActivity**. Además, habrá que configurar el navController y hacer que funcione el menú inferior.

Crea ahora 4 fragmentos vacíos y añádelos al gráfico de navegación:

- FragmentListaPeliculas
- FragmentListaFavoritas
- FragmentNuevaPelicula
- FragmentDetalle

FragmentListaPeliculas

Dentro del mapa de navegación este fragmento será el de inicio.
Únicamente tendrá un **RecyclerView** que ocupará todo el layout.

MODELO

Crea la clase **Pelicula** (data class). Será una entidad de la base de datos. Se mapeará en la tabla peliculas.
Además, tendrá como atributos id de tipo Long que será clave primaria no autogenerada. También tendrá un título (String), anyo (Int), director (String) que puede ser nulo, puntuacion (Float), imagen (String) que puede ser nula, sinopsis (String) y favorita (Boolean).

Añadiremos la clase **Critica**. Tendrá un identificador autoincrementado, un texto y una valoración.
Además, contendrá el identificador de la película a la que pertenece la crítica.
En la definición de la entidad añadiremos una clase foránea.

```
@Entity(tableName = "criticas",
      foreignKeys =
         [ForeignKey(
            entity = Pelicula::class,
            childColumns = ["pelicula_id"],
            parentColumns = ["id"],
            onDelete = CASCADE)
         ]
)
data class Critica(
   @PrimaryKey(autoGenerate = true)
   val id: Long,

   val texto: String,

   val valoracion: String,

   @ColumnInfo(name = "pelicula_id")
   val peliculaId: Long,

): java.io.Serializable
```

Las clases tienen que ser **Serializable**

DATOS.BASEDATOS

Crearemos una interface llamada **PeliculaDao** anotada como @Dao
Tendrá los siguientes métodos:

```
fun getPeliculas(): LiveData<List<Pelicula>>

fun getFavoritas(): LiveData<List<Pelicula>>

suspend fun insert(pelicula: Pelicula)

suspend fun deleteById(peliculaId: Long)

suspend fun cambiarFavorita(peliculaId: Long)
```

Para cada método **tendremos que realizar la consulta apropiada**.

Habrá que realizar también una clase **PeliculaDatabase** que herede de RoomDatabase

```
@Database(
    entities = [Pelicula::class, Critica::class],
    version = 1
)
abstract class PeliculaDatabase: RoomDatabase()
{
    abstract fun peliculaDao(): PeliculaDao

    //Patrón singleton para tener una única instancia de la clase
    companion object {
        private var INSTANCE: PeliculaDatabase? = null
        fun getDatabase(context: Context): PeliculaDatabase
        {
            if (INSTANCE == null) {
                synchronized(this) {
                    INSTANCE =
                     Room.databaseBuilder(context,PeliculaDatabase::class.java, "peliculas.db")
                        .createFromAsset("peliculas-copiar.db")
                        .build()
                }
            }
            return INSTANCE!!
        }
    }
}
```

Esta clase tendrá una patrón Singleton y, además, cargará el fichero peliculas-copiar.db (la base de datos) en la aplicación.
Para ello simplemente tendremos que crear la carpeta **assets** en el proyecto y copiar allí el fichero.

DATOS

Dentro del paquete Datos tendremos una clase llamada **PeliculaRepository**
Recibirá como parámetro un PeliculaDao y contendrá las siguientes funciones:

```
class PeliculaRepository(private val peliculaDao: PeliculaDao)
{

    fun get(): LiveData<List<Pelicula>> = peliculaDao.getPeliculas()

    fun getFavoritas(): LiveData<List<Pelicula>> = peliculaDao.getFavoritas()

    suspend fun insertar(pelicula: Pelicula) {
        peliculaDao.insert(pelicula)
    }

    suspend fun delete(pelicula: Pelicula)
    {
        peliculaDao.deleteById(pelicula.id)
```

```
    }

    suspend fun cambiarFavorita(peliculaId: Long)
    {
        peliculaDao.cambiarFavorita(peliculaId)
    }

}
```

VIEWMODEL

Se tendrá una instancia del repositorio y dos liveData. El primero contendrá la lista de películas de la base de datos y el segundo la lista de películas favoritas:

```
class PeliculasViewModel(context: Context): ViewModel()
{
    private val repositorio :PeliculaRepository
    val peliculas: LiveData<List<Pelicula>>
    val peliculasFavoritas: LiveData<List<Pelicula>>

    init
    {
        val peliculaDao = PeliculaDatabase.getDatabase(context).peliculaDao()
        repositorio = PeliculaRepository(peliculaDao)
        peliculas = repositorio.get()
        peliculasFavoritas = repositorio.getFavoritas()
    }


    fun insertarPelicula(pelicula: Pelicula)
    {
        viewModelScope.launch {
            repositorio.insertar(pelicula)
        }
    }

    fun borrarPelicula(pelicula: Pelicula)
    {
        viewModelScope.launch {
            repositorio.delete(pelicula)
        }
    }

    fun cambiarFavorita(peliculaId: Long)
    {
        viewModelScope.launch {
            repositorio.cambiarFavorita(peliculaId)
        }
    }
```

```
    // Define ViewModel factory in a companion object
    companion object {
        val Factory: ViewModelProvider.Factory = viewModelFactory {
            initializer {
                val contexto = (this[APPLICATION_KEY] as Context)
                PeliculasViewModel(
                    context = contexto
                )
            }
        }
    }

}
```

CODIFICACIÓN DE LAS VISTAS

En los fragments debemos crear los **viewModel** del siguiente modo:

```
private val peliculasViewModel: PeliculasViewModel by activityViewModels
{PeliculasViewModel.Factory}
```

Además, usaremos un **binding** para acceder a cada componente

Debemos completar nuestro **gráfico de navegación** para establecer los caminos a los que se puede navegar desde cada fragmento.
De momento, únicamente nos moveremos desde FragmentListaPeliculas a FragmentDetalle (aunque no implementemos este fragmento en la actividad) y de FragmentListaPeliculas a FragmentDetalle.
FragmentDetalle debe recibir el objeto película con **SafeArgs**. El argumento será de tipo Serializable.

Y también nos moveremos de FragmentNuevaPelicula a FragmentListaPeliculas (después de insertar una película iremos al listado general de películas).

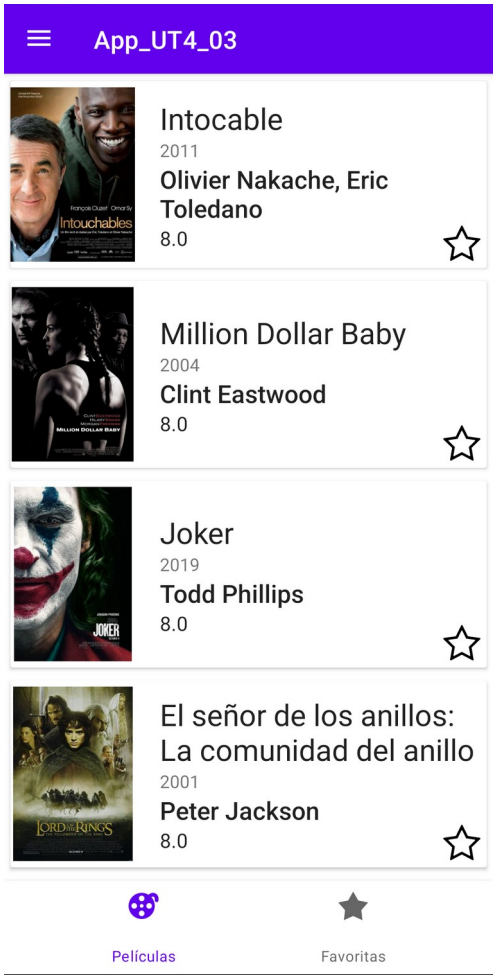
Luego, habrá que ir haciendo las acciones oportunas para **realizar la funcionalidad de cada uno**.

Para los fragment de lista de películas y lista de favoritas podemos utilizar el mismo Adapter. Lo podemos llamar **PeliculasAdapter**
Deberá recibir la lista de películas, además de dos métodos: uno para cambiar una película a favorita o no y otro cuando se hace clic en algún ítem:

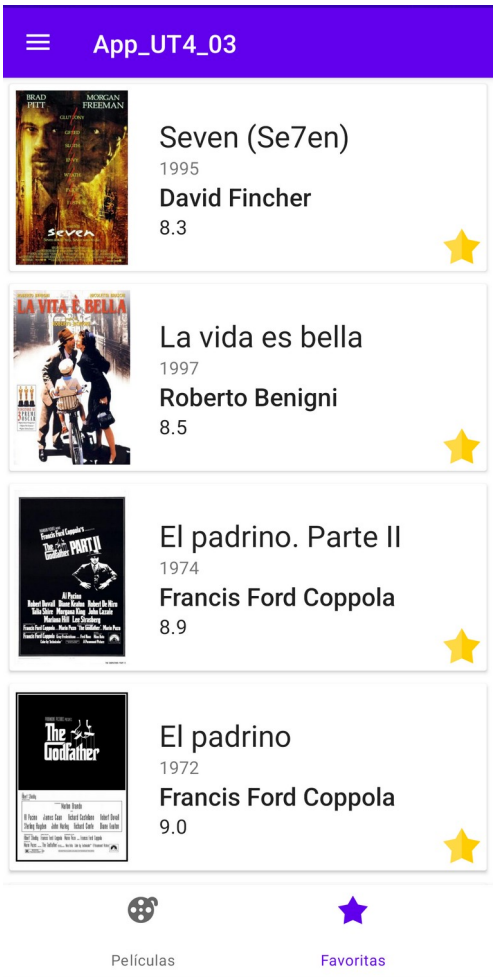
```
class PeliculasAdapter(private val listaPeliculas: List<Pelicula>,
                       private val onClickPelicula: (Pelicula) -> Unit,
                       private val cambiarFavorita: (Pelicula) -> Unit)
```

```
   : RecyclerView.Adapter<PeliculasAdapter.PeliculasViewHolder>()
{

// Resto de código


}
```

FragmentListaPeliculas



FragmentListaFavoritas



Se mostrarán todas las películas que NO sean favoritas.

Se deberá observar peliculas del viewModel y cuando cambie se cargará el RecyclerView

FragmentNuevaPelicula **FragmentDetalle**

Los haremos en la siguiente práctica

Para mostrar las imágenes usaremos **Coil**.
Las imágenes que he puesto en la base de datos están alojadas en Filmaffinity. Han restringido su acceso directamente desde Android.
Para conseguir descargarlos tenemos que cambiar el User Agent simulando como que somos un navegador.
El código sería el siguiente:

```
val userAgent = "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/108.0.0.0 Safari/537.36"
val url = binding.etImagen.text.toString()
binding.imagenPelicula.load(url)
{
   addHeader("User-Agent", userAgent)
}
```