

Laboratorio 06 - Model, Repository and ViewModel

1. Introducción

En este laboratorio se comenzará a implementar la arquitectura **MVVM** junto con el patrón de diseño **Repository**. Se añadirá Model para representar y almacenar los datos de la aplicación, ViewModel para comunicar la vista con la capa de Model y Repository que se encargará de abstraer y encapsular el acceso a los datos de la aplicación.

1.1 ¿Qué se aprenderá?

- Implementación de Model y ViewModel
- Implementación del patrón de diseño Repository
- Implementación de una subclase de Application

1.2. Requisitos

- Tener previamente instalado Android Studio en tu computadora
- Conocimiento previo sobre Navigation Component
- Ejecutar acciones al presionar un botón

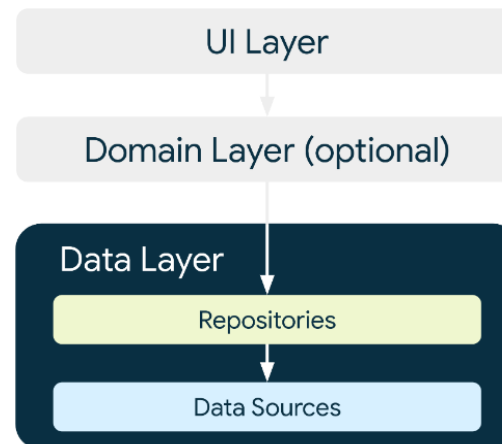
2. Capa de datos

La capa de datos contiene *datos de la aplicación y lógica empresarial*. La lógica empresarial es la que aporta valor a la app: está compuesta por reglas empresariales reales que determinan cómo se deben crear, almacenar y cambiar los datos de la aplicación.

Esta separación de problemas permite que la capa de datos se use en varias pantallas, comparta información entre diferentes partes de la app y reproduce la lógica empresarial fuera de la IU a fin de realizar pruebas de unidades.

2.1 Arquitectura de capa de datos

La capa de datos está formada por **repositorios** que pueden contener de cero a muchas **fuentes de datos**. Se debe crear una clase de repositorio para cada tipo de datos diferente que administre tu app.

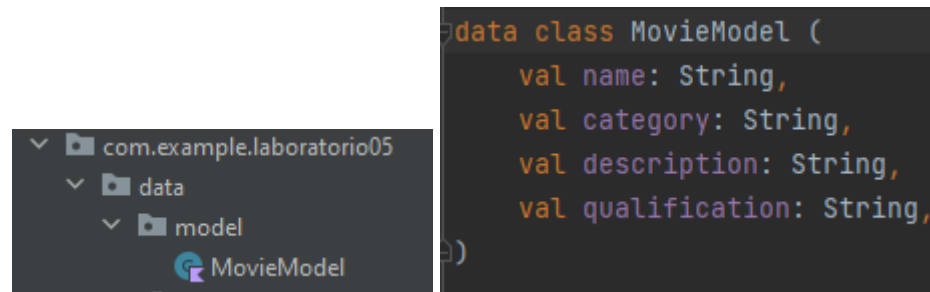


3. Modelos de Datos

Los modelos de datos son representaciones abstractas de los datos que serán utilizados en una aplicación. En otras palabras, los modelos de datos describen la estructura y los tipos de datos que serán utilizados. Es posible verlos como una entidad que cuenta con una serie de atributos.

3.1 Cómo Implementar un modelo de datos

De ahora en adelante se dividirán toda la lógica de negocio de nuestra aplicación en sus respectivas carpetas para poder separar en capas y así poder darle más escalabilidad a nuestras aplicaciones, para poder implementar un modelo de datos se debe crear un paquete llamado **data** el cual contendrá otro paquete llamado **model** donde crearemos todos los modelos que ocupará la aplicación.



3.2 DummyData

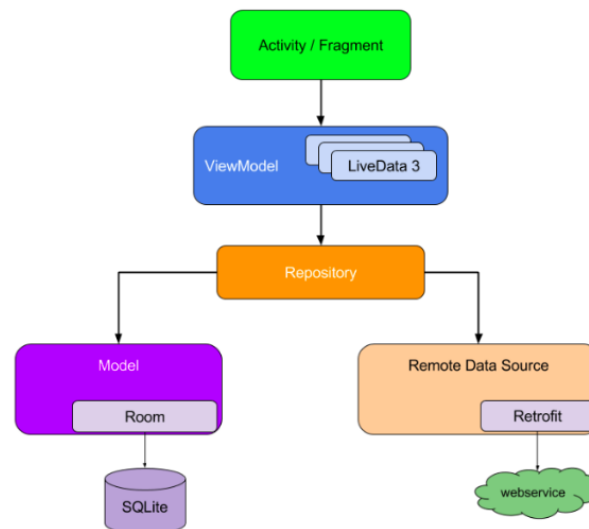
Ahora toda la información de los modelos de datos deberían ser guardadas dentro de una base de datos a la cual se pueda consultar cuando se necesite, en este laboratorio se trabajará con una **MutableList**, ya que es un tipo de lista que puede ser modificada después de su creación, lo que significa que puedes agregar, quitar o actualizar sus elementos. Crearemos el archivo **DummyData** como un archivo kotlin dentro de nuestro paquete **data**, este será una **MutableList** que tendrá como elementos nuestro modelo de datos y tendría la siguiente estructura

```
var movies = mutableListOf(  
    MovieModel(name, category, description, qualification),  
    MovieModel(name2, category2, description2, qualification2)  
)
```

4. Patrón de repositorio

Es un patrón de diseño que aísla la capa de datos del resto de la aplicación. Dicha capa hace referencia a la parte que controla los datos y la lógica empresarial, separando los componentes visuales, lo que expone APIs coherentes de modo que el resto de la aplicación acceda a esos datos. Mientras que la IU presenta información al usuario, la capa de datos incluye elementos como el código de red, las bases de datos de Room, el manejo de errores y cualquier código que lea o manipule datos.

Un repositorio puede resolver conflictos entre fuentes de datos (como modelos persistentes, servicios web y cachés) y centralizar los cambios en estos datos.



Las clases de repositorio son responsables de las siguientes tareas:

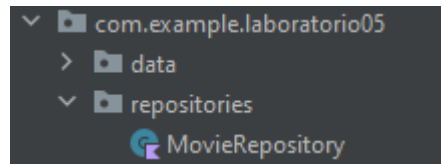
- Exponer datos al resto de la app
- Centralizar los cambios en los datos
- Resolver conflictos entre múltiples fuentes de datos
- Abstraer fuentes de datos del resto de la app
- Contener la lógica empresarial

4.1 Ventajas de usar un repositorio

Un módulo de repositorio controla operaciones de datos y te permite usar varios backends. En una app real típica, el repositorio implementa la lógica para decidir si debe recuperar datos de una red o usar resultados almacenados en caché de una base de datos local. Con un repositorio, puedes intercambiar los detalles de la implementación, como la migración a una biblioteca de persistencia diferente, sin afectar el código de llamada, como los modelos de vista. Esto también permite que tu código sea modular y se pueda probar. Puedes simular con facilidad el repositorio y probar el resto del código.

4.2 ¿Cómo implementar un repositorio?

Dentro De nuestro paquete principal se debe crear un paquete llamado **repositories** el cual contendrá todos los repositorios de la aplicación



Se debe crear una clase Kotlin llamada **MovieRepository** que tendrá como parámetro la base de datos o en este caso la **MutableList**, dentro del repositorio se tendrán todas las funcionalidades que le podremos dar a nuestros datos y las que utilizaremos junto a nuestra UI.

```
Bryan31542 *  
class MovieRepository(private val movies: MutableList<MovieModel>) {  
  
    Bryan31542  
    fun getMovies() = movies  
  
    Bryan31542  
    fun addMovies(movie: MovieModel) = movies.add(movie)  
  
}
```

5. ViewModel

5.1 ¿Qué es el ViewModel?

Es una lógica empresarial o un contenedor de estado a nivel de pantalla que expone el estado a la IU y encapsula la lógica empresarial relacionada. Su principal ventaja es que almacena en caché el estado y lo conserva durante los cambios de configuración, esto significa que la IU no tiene que recuperar datos cuando navegas entre actividades o si sigues cambios de configuración, como cuando rotas la pantalla. Además, actúa como intermediario entre la capa de datos (Model) y entre la capa de la interfaz de usuario (View).

Asimismo, el ViewModel se comunica con el modelo de datos a través de un **repository** u otro proveedor de datos y brinda esos datos a la vista de la aplicación.

5.2 Beneficios de ViewModel

La alternativa a un ViewModel es una clase simple que contiene los datos que muestras en tu IU. Esto puede convertirse en un problema cuando navegas entre actividades o destinos de Navigation. Si lo haces, esos datos se destruirán si no los almacenan con el mecanismo de guardado de estado de instancias. ViewModel proporciona una API conveniente para la persistencia de datos que resuelve este problema.

Los beneficios clave de la clase ViewModel son básicamente dos:

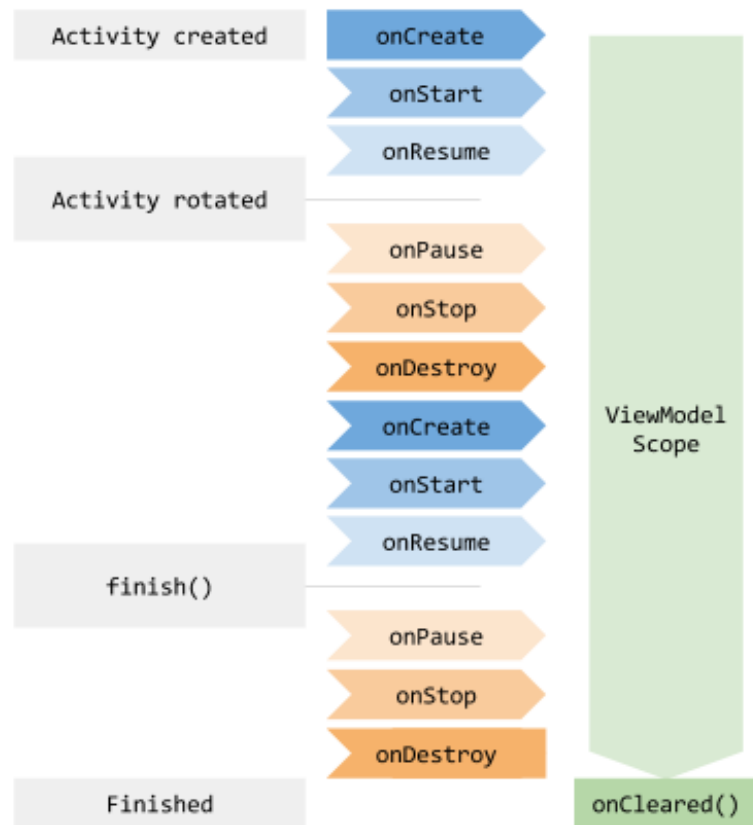
- Te permite conservar el estado de la IU.
- Proporciona acceso a la lógica empresarial.

5.4 Ciclo de vida de un ViewModel

El ciclo de vida de un ViewModel está vinculado directamente a su alcance. Un ViewModel permanece en la memoria hasta que desaparece el **ViewModelStoreOwner** que determina su alcance. Esto puede ocurrir en los siguientes contextos:

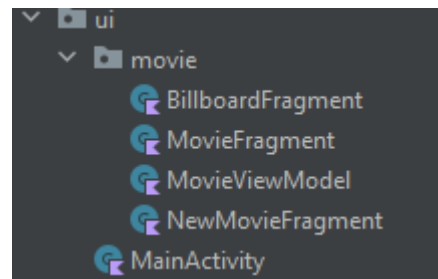
- En el caso de una actividad, cuando termina.
- En el caso de un fragmento, cuando se desvincula.
- En el caso de una entrada de Navigation, cuando se quita de la pila de actividades.

Esto hace que los ViewModels sean una gran solución para almacenar datos que sobreviven a cambios de configuración.



5.5 ¿Cómo implementar ViewModel?

Dentro del paquete llamado **ui** se debe crear un nuevo paquete llamado **Movie** el cual contendrá todos los fragmentos relacionados a movie, así como el **MovieViewModel**



La clase contendrá como parámetro el repositorio creado con el cual podremos acceder a todas las funciones definidas del mismo, además se debe definir una función llamada **getMovies** que se encargará de mostrar todas las películas dentro de la **MutableList**, y otra función llamada **addMovies** que permitirá añadir nueva películas a la lista.

Asimismo, se hace uso de un **CompanionObject** para poder crear una instancia de nuestro **MovieViewModel** por medio de **viewModelFactory**, donde a su vez se obtiene una instancia de **MovieReviewerApplication** haciendo uso de una clave de valor **APPLICATION_KEY** que sirve para identificar el objeto de la aplicación en el mapa de valores de la aplicación. Con esa instancia se inicializa el viewModel y se brinda la fuente de datos por medio del repositorio.

```
class MovieViewModel(private val repository: MovieRepository) : ViewModel() {
```

• Bryan31542

```
fun getMovies() = repository.getMovies()
```

• Bryan31542

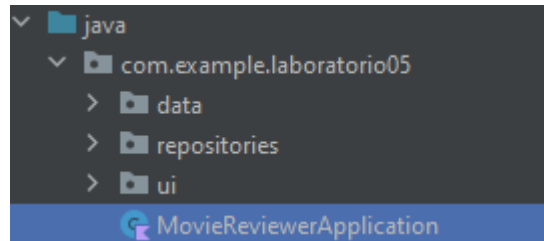
```
fun addMovies(movie: MovieModel) = repository.addMovies(movie)
```

• Bryan31542

```
companion object {  
    val Factory = viewModelFactory { this: InitializerViewModelFactoryBuilder  
        initializer { this: CreationExtras  
            val app = this[APPLICATION_KEY] as MovieReviewerApplication  
            MovieViewModel(app.movieRepository) ^initializer  
        }  
    }  
}
```

6. MovieReviewerApplication

Ahora se procede a crear una nueva clase llamada **MovieReviewerApplication** dentro de la raíz de nuestro proyecto



Dicha clase heredará de **Application()** y dentro ella hace uso de **by lazy** que pospone la creación de dicho repositorio hasta el momento en el que será utilizado y se le brinda la lista mutable llamada **movies** para su inicialización

```
class MovieReviewerApplication : Application() {  
    val movieRepository: MovieRepository by lazy {  
        MovieRepository(movies)  
    }  
}
```

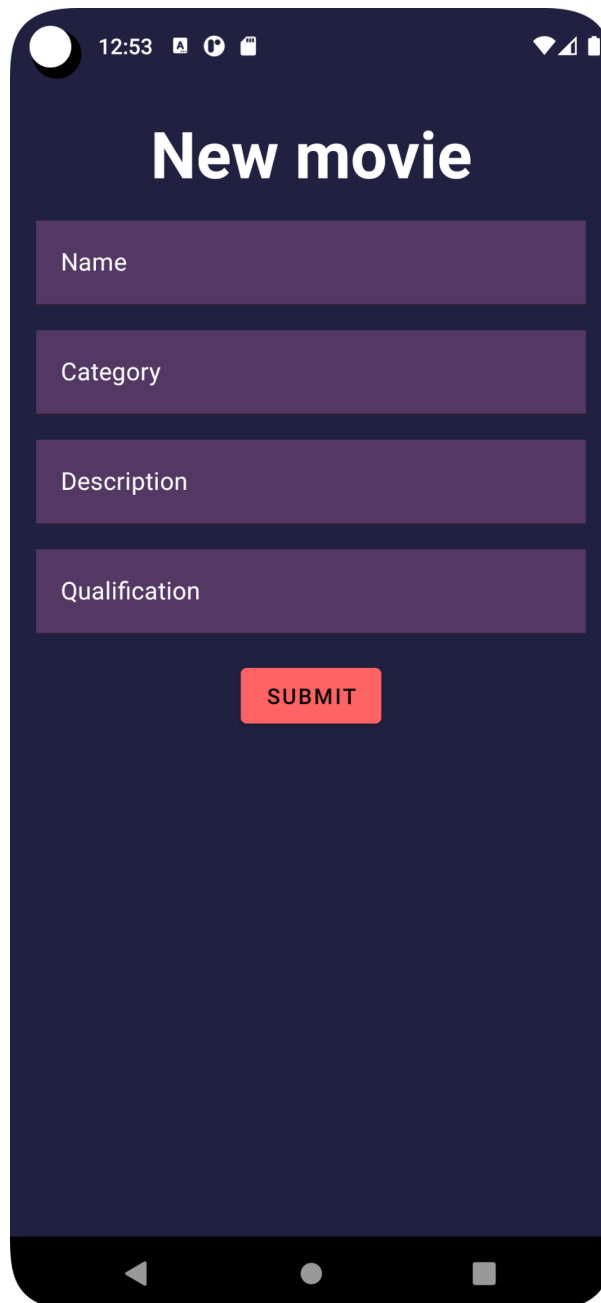
Cabe recalcar que es necesario agregar **android:name=".MovieReviewerApplication"** ya que por defecto se busca la clase **"Application"**, pero al existir una subclase es necesario indicar que utilice dicha subclase para evitar errores

```
<application
    android:name=".MovieReviewerApplication"
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
```

El hacer uso de una subclase de **Application** es de suma importancia ya que permite almacenar objetos y recursos que **serán utilizados en toda la aplicación** y que deben tener un ciclo de vida que abarque todo el scope de la misma.

8. Actividad a realizar

A la aplicación realizada en el laboratorio 05 se le dará la funcionalidad al formulario de **crear una nueva película**,



The image shows a mobile application interface on a dark blue background. At the top, there is a status bar with the time 12:53 and various icons. Below the status bar, the title "New movie" is displayed in white. The form consists of four input fields, each with a light purple background and a white label: "Name", "Category", "Description", and "Qualification". Below these fields is a red "SUBMIT" button. The bottom of the screen shows the Android navigation bar with back, home, and recent apps icons.

9. To do:

1. ¿En qué consiste el patrón de diseño **Repository**?

2. ¿Qué es el **ViewModel**?

9. Rúbrica

Actividad	Porcentaje	Nota
División de los archivos de la aplicación en sus respectivos paquetes	10%	
Uso de los modelos de datos	10%	
Uso del patrón de repositorio	15%	
Implementación del ViewModel	20%	
Se agrega correctamente una nueva película	25%	
Preguntas teóricas	10%	
Puntualidad	10%	
Total	100%	