

Laboratorio 07 - LiveData and DataBinding

1. Introducción

En este laboratorio se brindará un acercamiento a la clase de Android llamada **LiveData**, la cual permite crear objetos observables los cuales hacen posible que fragmentos o actividades se suscriban a cambios de datos o actualizaciones en tiempo real. Además se implementará **DataBinding** para establecer enlaces de datos entre los elementos de la interfaz y la capa de lógica de datos de nuestra aplicación. Eliminando el **Boilerplate code** al momento de referenciar los elementos de la vista.

1.1 ¿Qué se aprenderá?

- Implementación de LiveData
- Implementación de DataBinding

1.2. Requisitos

- Tener previamente instalado Android Studio en tu computadora
- Poder agregar una película usando Model, ViewModel & Repository
- Proyecto dividido en distintos paquetes según su funcionalidad

2. Live Data

Es una clase de contenedor de datos observables. LiveData está optimizado para ciclos de vida, lo que significa que respeta el ciclo de vida de otros componentes de las apps, como actividades, fragmentos o servicios. Esta optimización garantiza que LiveData solo actualice observadores de componentes de aplicaciones que tienen un estado de ciclo de vida activo.

LiveData proporciona un contenedor de datos observable que puede contener cualquier tipo de dato. Los observadores (normalmente, una actividad, fragmento o ViewModel) pueden suscribirse a LiveData y recibir notificaciones cuando se produce un cambio en los datos contenidos dentro de ella. Esto permite que la interfaz de usuario refleje automáticamente los cambios en los datos, sin tener que realizar actualizaciones manuales.

Además, se encuentra en estado activo si su ciclo de vida está en el estado **onStarted()** o **onResumed()**. Live Data solo notifica a los observadores activos sobre las actualizaciones.

2.1 Ventajas de usar LiveData

- **Garantiza que la IU coincida con el estado de los datos:** LiveData sigue el patrón de diseño observador. LiveData notifica a los objetos Observer cuando cambian los datos subyacentes. Puedes consolidar tu código para actualizar la IU en esos objetos Observer. De esa manera, no necesitas actualizar la IU cada vez que cambian los datos de la app porque el observador lo hace por ti.

- **Sin fugas de memoria:** Los observadores están vinculados a objetos Lifecycle y borran lo que crean cuando se destruye el ciclo de vida asociado.
- **Actividades detenidas para evitar las fallas:** Si el ciclo de vida del observador está inactivo, como en el caso de una actividad de la pila de actividades, no recibe ningún evento de LiveData.
- **No más control manual del ciclo de vida:** Los componentes de IU solo observan los datos relevantes y no detienen ni reanudan la observación. LiveData se ocupa automáticamente de todo esto, ya que está al tanto de los cambios de estado del ciclo de vida relevantes mientras lleva a cabo la observación.
- **Datos siempre actualizados:** Si un ciclo de vida queda inactivo, recibe los datos más recientes después de quedar activo de nuevo. Por ejemplo, una actividad que estuvo en segundo plano recibe los datos más recientes inmediatamente después de volver al primer plano.

- **Cambios de configuración apropiados:** Una actividad o un fragmento que se vuelve a crear debido a un cambio de configuración, como la rotación del dispositivo, recibe de inmediato los datos disponibles más recientes.


2.2 LiveData en la arquitectura de una aplicación

LiveData se adapta al ciclo de vida y sigue el ciclo de vida de entidades como actividades y fragmentos. Usa LiveData para establecer una comunicación entre estos propietarios del ciclo de vida y otros objetos con una vida útil diferente, como los objetos ViewModel. **La principal responsabilidad de ViewModel es cargar y administrar los datos relacionados con la IU, lo que lo convierte en un gran candidato para conservar objetos LiveData.** Crea objetos LiveData en ViewModel y úsalos para exponer el estado a la capa de IU.

Las actividades y los fragmentos no deben contener instancias de LiveData, ya que su función es mostrar datos, no mantener estados. Además, a fin de que las actividades y los fragmentos no puedan contener datos, facilita la escritura de pruebas de unidades.

2.3 ¿Cómo Implementar LiveData en nuestro proyecto?

LiveData es un wrapper que se puede usar con cualquier dato, incluidos los objetos que implementan Collections, cómo Listas. Por lo general, un objeto LiveData se almacena dentro de nuestro **ViewModel**, por lo que para poder implementar un LiveData se realiza de la siguiente manera:

A screenshot of a code editor with a dark background and light-colored text. The code is written in Kotlin and defines a class MovieViewModel that inherits from ViewModel. It has a private val repository of type MovieRepository. There are five var declarations for MutableLiveData: name, category, description, qualification, and status, all initialized with empty strings. There is also a fun declaration for getMovies() that calls repository.getMovies(). The code is numbered from 1 to 8 on the left side of the editor.

```
1 class MovieViewModel(private val repository: MovieRepository) : ViewModel() {  
2     var name = MutableLiveData("")  
3     var category = MutableLiveData("")  
4     var description = MutableLiveData("")  
5     var qualification = MutableLiveData("")  
6     var status = MutableLiveData("")  
7  
8     fun getMovies() = repository.getMovies()
```

Cada una de las variables de tipo **MutableLiveData** hacen referencia a los campos que se necesitan para crear una nueva **Movie**.

Además, cabe recalcar que el **value** dentro del **MutableLiveData** es el valor inicial que tomará nuestro LiveData, de esta forma podemos tener los datos de nuestra aplicación cambiando a tiempo real; pero, aún no se ha conectado a nuestro IU, para poder lograrlo debemos hacer uso de **DataBinding**

2.4 Validaciones en ViewModel


Ahora que nuestro **ViewModel** está suscrito a datos de tipo **LiveData** y que notificará cada vez que surja cierto cambio en su estado es necesario identificar los estados en los que nuestra información se encontrará. Por ende dentro del **Companion Object** que por medio del patrón de diseño **Factory** inicializa nuestro **ViewModel** se crearán una variables que nos permitirán ubicar el estado de nuestros datos.

```
1 companion object {
2     val Factory = viewModelFactory {
3         initializer {
4             val app = this[APPLICATION_KEY] as MovieReviewerApplication
5             MovieViewModel(app.movieRepository)
6         }
7     }
8
9     const val MOVIE_CREATED = "Movie created"
10    const val WRONG_INFORMATION = "Wrong information"
11    const val INACTIVE = ""
12
```

Previamente el proceso para crear una nueva película estaba siendo realizado dentro de un fragmento, lo que podía exponer nuestros datos. Ahora dicho proceso se realizará dentro del **ViewModel** de la siguiente manera:

```
1 private fun addMovies(movie: MovieModel) = repository.addMovies(movie)
2
3 fun createMovie() {
4     if (!validateData()) {
5         status.value = WRONG_INFORMATION
6         return
7     }
8
9     val movie = MovieModel(
10         name.value!!,
11         category.value!!,
12         description.value!!,
13         qualification.value!!
14     )
15
16     addMovies(movie)
17     clearData()
18
19     status.value = MOVIE_CREATED
20 }
```


Donde se debe crear una nueva función llamada **fun createMovie()** la cual primeramente verificará por medio de la función **validateData()**, si alguno de los datos se encuentra nulo o vacío, y si es así se encargará de actualizar el estado del LiveData a **Wrong Information** y la función hará **return** para evitar que la función siga ejecutándose.



```
1 private fun validateData(): Boolean {
2     when {
3         name.value.isNullOrEmpty() → return false
4         category.value.isNullOrEmpty() → return false
5         description.value.isNullOrEmpty() → return false
6         qualification.value.isNullOrEmpty() → return false
7     }
8     return true
9 }
```

Si dentro de la función **createMovie()** se pasó la validación de campos vacíos es posible proceder a construir una nueva película haciendo uso de **MovieModel**, donde se ocupan los datos de tipo LiveData accediendo a su valor y haciendo uso de **Not-Null Assertion (!)**, lo que evita que un dato sea nulo, y si lo es lanza una excepción.

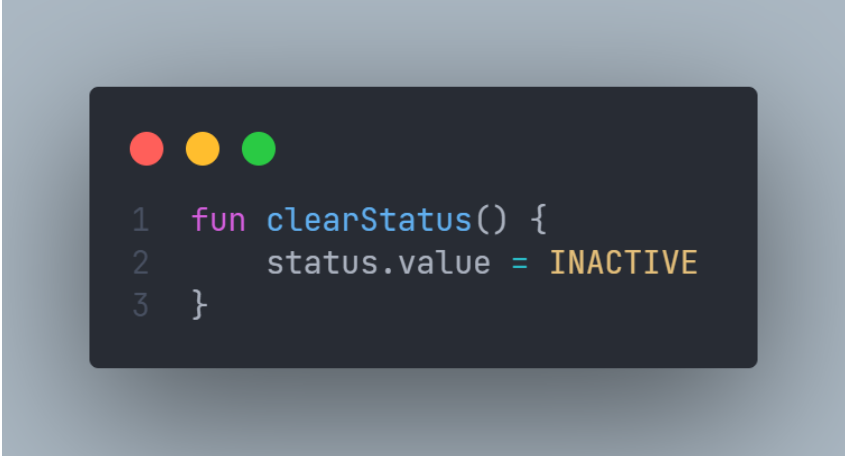
Una vez, se ha construido correctamente la película, se procede a utilizar la función de nuestro **MovieRepository** llamada **addMovies** donde recibe la película previamente construida. Además, se limpian las variables LiveData a su estado inicial por medio de la función **clearData()**



```
1 private fun clearData() {  
2     name.value = ""  
3     category.value = ""  
4     description.value = ""  
5     qualification.value = ""  
6 }
```

Ya que fue posible crear una película es posible asignarle al estado de nuestro LiveData la variable de nuestro **Companion Object** llamada **MOVIE_CREATED**

Y finalmente dentro de **MovieViewModel** se crea una función que se encargará de limpiar el estado del LiveData, ya que es posible que se encuentre en **WRONG_INFORMATION** o **MOVIE_CREATED** y setear un estado inactivo.



```
1 fun clearStatus() {  
2     status.value = INACTIVE  
3 }
```

3. DataBinding

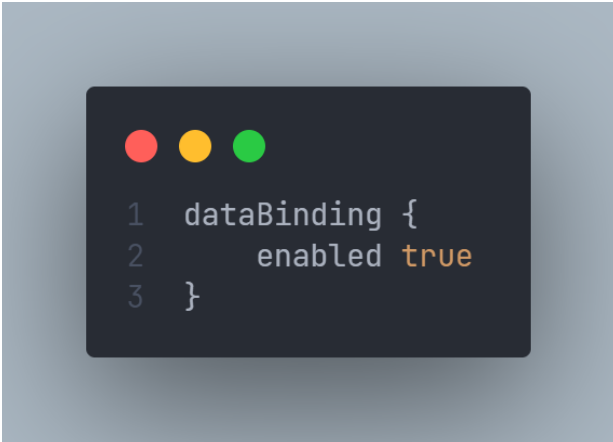
DataBinding es una función de Android que **permite vincular los elementos de la interfaz de usuario (UI) de una aplicación directamente a los datos de la aplicación**, lo que significa que no es necesario realizar actualizaciones manuales para que la interfaz refleje los cambios en los datos. DataBinding utiliza una sintaxis especial en XML para crear una conexión entre la UI y los datos, lo que facilita la implementación de patrones de arquitectura de software como el **patrón MVVM (Model-View-ViewModel)**.

En lugar de tener que actualizar manualmente los elementos de la UI en el código, DataBinding permite que la UI se actualice automáticamente a medida que los datos cambian en la aplicación. Esto significa que es posible centrarse en la lógica de la aplicación en lugar de preocuparse por la sincronización de los datos y la UI.

Además, DataBinding proporciona una mayor eficiencia y rendimiento, ya que reduce la necesidad de hacer referencia a los elementos de la interfaz de usuario en el código y también disminuye la cantidad de código repetitivo necesario para mantener la sincronización de los datos y la UI.

3.1 ¿Cómo Implementar DataBinding en nuestro proyecto?

Lo primero que se debe realizar para poder implementar DataBinding es preparar nuestro entorno de desarrollo agregando la librería en nuestro proyecto, por lo que debemos dirigirnos a nuestro **Gradle Scripts** seguido de **build.gradle(Module:app)**, dentro del build.gradle debemos dirigirnos a la sección que dice **android** y dentro de las llaves agregar el siguiente bloque de código

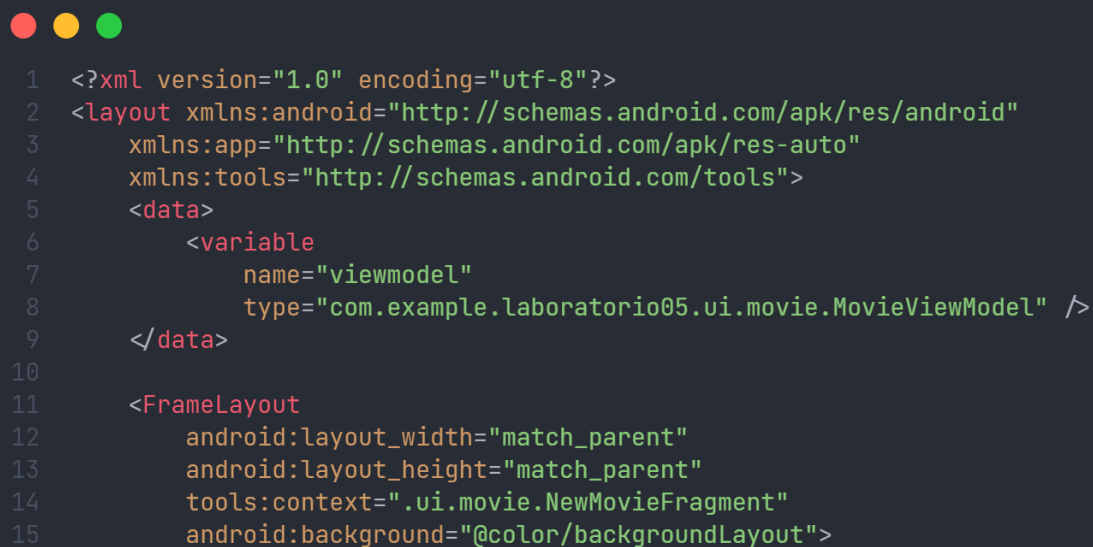
A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains three lines of code:

```
1  dataBinding {  
2      enabled true  
3  }
```

```
1  dataBinding {  
2      enabled true  
3  }
```

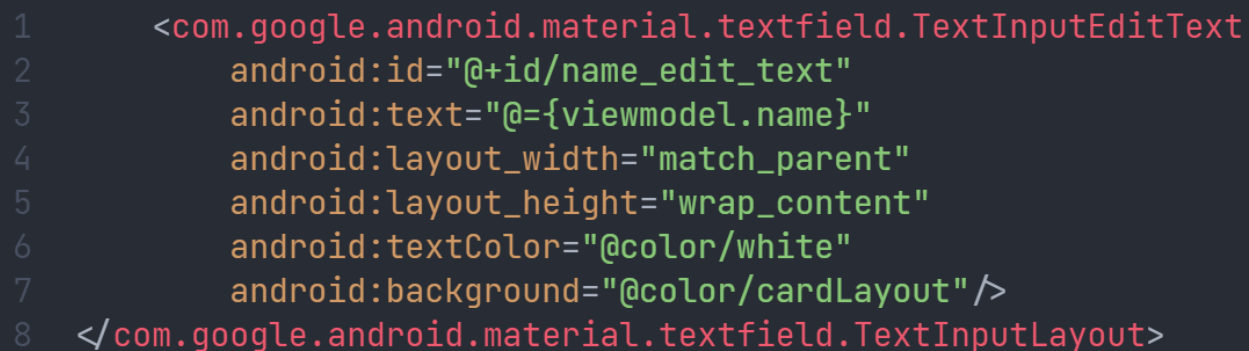
Ya tendremos activado el DataBinding en nuestro entorno de desarrollo, lo siguiente que se debe hacer es implementarlo dentro de nuestro archivo de diseño para vincular nuestros componentes con nuestra fuente de datos, en este caso estaremos vinculando nuestras vistas con el **ViewModel**

Ahora es necesario dirigirse a **fragment_new_movie.xml**, añadir toda nuestra vista dentro de un **layout** que a su vez tendrá dentro **<data>** que será utilizado para definir las variables que se usarán en la vinculación de datos en la interfaz de usuario. Y dentro de **<data>** se define una nueva **<variable>** que tendrá como nombre **"viewmodel"** y el **type** se le debe especificar la ruta en la que se encuentra nuestro **MovieViewModel**



```
1 <?xml version="1.0" encoding="utf-8"?>
2 <layout xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:app="http://schemas.android.com/apk/res-auto"
4     xmlns:tools="http://schemas.android.com/tools">
5     <data>
6         <variable
7             name="viewmodel"
8             type="com.example.laboratorio05.ui.movie.MovieViewModel" />
9     </data>
10
11     <FrameLayout
12         android:layout_width="match_parent"
13         android:layout_height="match_parent"
14         tools:context=".ui.movie.NewMovieFragment"
15         android:background="@color/backgroundLayout">
```

Ya que tenemos enlazado nuestro **MovieViewModel** con nuestro UI, ahora podremos acceder a todas las variables dentro que fueron declaradas de tipo **LiveData**. Para poder conectar el texto de nuestros componentes con nuestras variables se establece una vinculación bidireccional entre la vista y el ViewModel por medio del **DataBinding** haciendo uso de **@={viewmodel.name}**



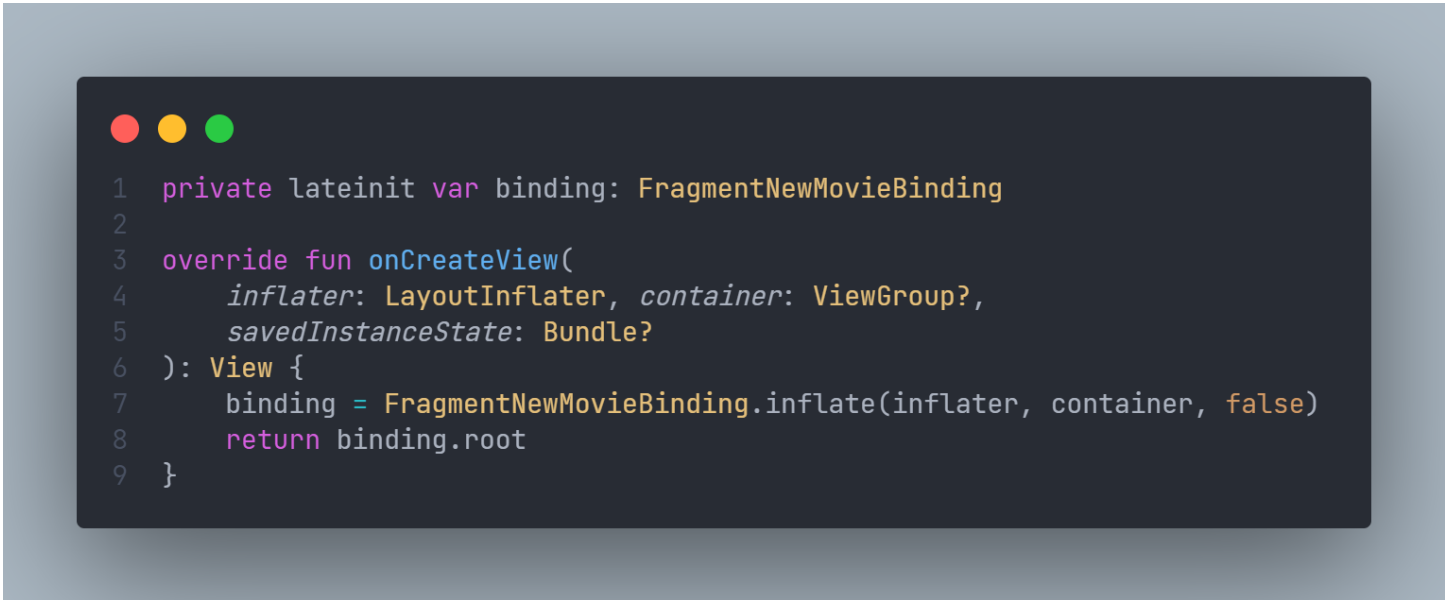
```
1      <com.google.android.material.textfield.TextInputEditText
2          android:id="@+id/name_edit_text"
3          android:text="@={viewmodel.name}"
4          android:layout_width="match_parent"
5          android:layout_height="wrap_content"
6          android:textColor="@color/white"
7          android:background="@color/cardLayout"/>
8  </com.google.android.material.textfield.TextInputLayout>
```

Realiza la vinculación bidireccional para Category, Description y Qualification

Ahora para asignarle una función a un botón ya no es necesario hacerlo por medio **setOnClickListener**, es posible vincular las acciones por medio del **DataBinding**, por medio de **android:onClick="@{() -> viewModel.createMovie()}"** se indica que cuando nuestro elemento de la vista sea presionado, se ejecutará la función **createMovie**, la cuál fue declarada dentro de nuestro **MovieViewModel**

```
1 <Button
2     android:id="@+id/submit_button"
3     android:onClick="@{() -> viewModel.createMovie()}"
4     android:layout_width="wrap_content"
5     android:layout_height="wrap_content"
6     android:layout_marginTop="@dimen/mediumGap"
7     android:backgroundTint="@color/buttonColor"
8     android:textColor="@color/black"
9     android:text="@string/submitButtonText"
10    app:layout_constraintEnd_toEndOf="@+id/CalificationTv"
11    app:layout_constraintStart_toStartOf="@+id/CalificationTv"
12    app:layout_constraintTop_toBottomOf="@+id/CalificationTv" />
```

Ya que nuestros archivos **.xml** de los fragmentos han sido preparados para utilizar **DataBinding** se procede a inflar nuestras vistas en los archivos **.kt** haciendo uso del mismo

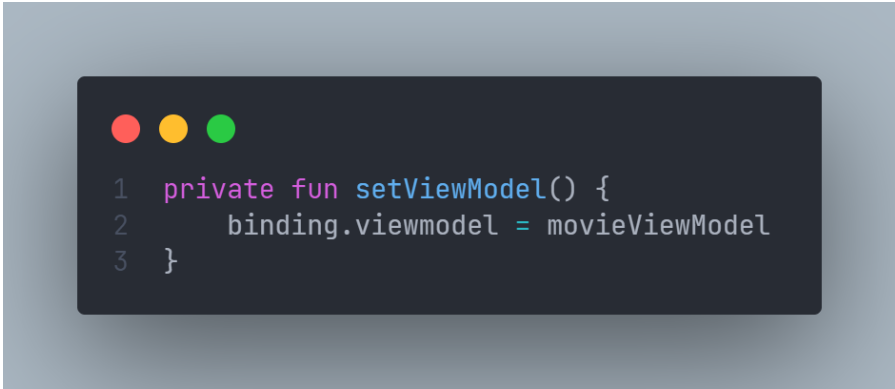
A screenshot of a code editor with a dark background and light-colored text. The code is written in Kotlin and is part of an Android application. It shows the onCreateView method of a Fragment, which is used to inflate the layout for the fragment. The code is as follows:

```
1 private lateinit var binding: FragmentNewMovieBinding
2
3 override fun onCreateView(
4     inflater: LayoutInflater, container: ViewGroup?,
5     savedInstanceState: Bundle?
6 ): View {
7     binding = FragmentNewMovieBinding.inflate(inflater, container, false)
8     return binding.root
9 }
```

Se declara una variable llamada **binding** de tipo **FragmentNewMovieBinding**, cabe recalcar que esta clase es generada por la función de **DataBinding** y su nombre varía según el fragmento. Y se procede a inflar las vistas de dicho fragmento y retornar la raíz de la interfaz de usuario, la cual permite mostrar las vistas en pantalla.

Realizar este proceso para los tres fragmentos de tu aplicación y para MainActivity

Ya que nuestro fragmento **NewMovieFragment** ha implementado **DataBinding**, se debe establecer el **MovieViewModel** en dicha vista haciendo uso del mismo. Lo que permite que la vista actualice su estado según los cambios en los datos del **ViewModel**, y envíe eventos y acciones del usuario al **VM**.

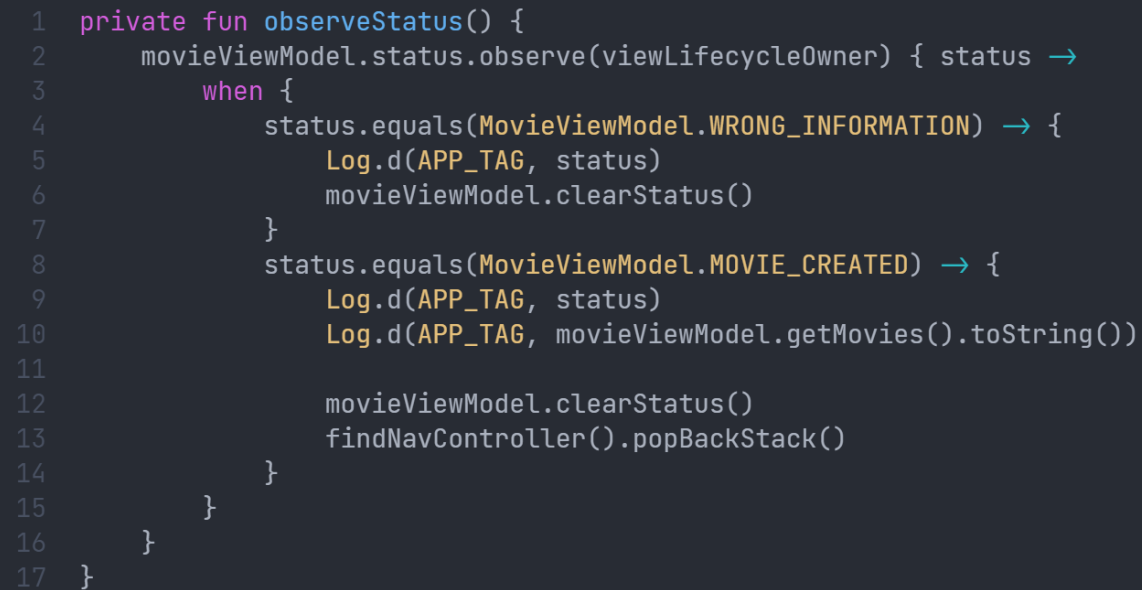
A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It contains three lines of Kotlin code:

```
1 private fun setViewModel() {  
2     binding.viewmodel = movieViewModel  
3 }
```

Y finalmente, dentro del fragmento se debe crear una función la cual se encargue de observar el estado del **ViewModel** y tomar decisiones en consecuencia. Por medio del **viewLifecycleOwner** nuestro observer estará activo únicamente mientras el ciclo de vida del Fragmento este activo.

Además, de acuerdo al estado que fue definido dentro de **MovieViewModel**, el cual se encarga de verificar si los datos están aptos para ser almacenados, se procede a evaluar dicho estado, y si la validación fue errónea se muestra el mensaje de error y se limpia el estado del ViewModel. Mientras que, si la película fue creada correctamente se

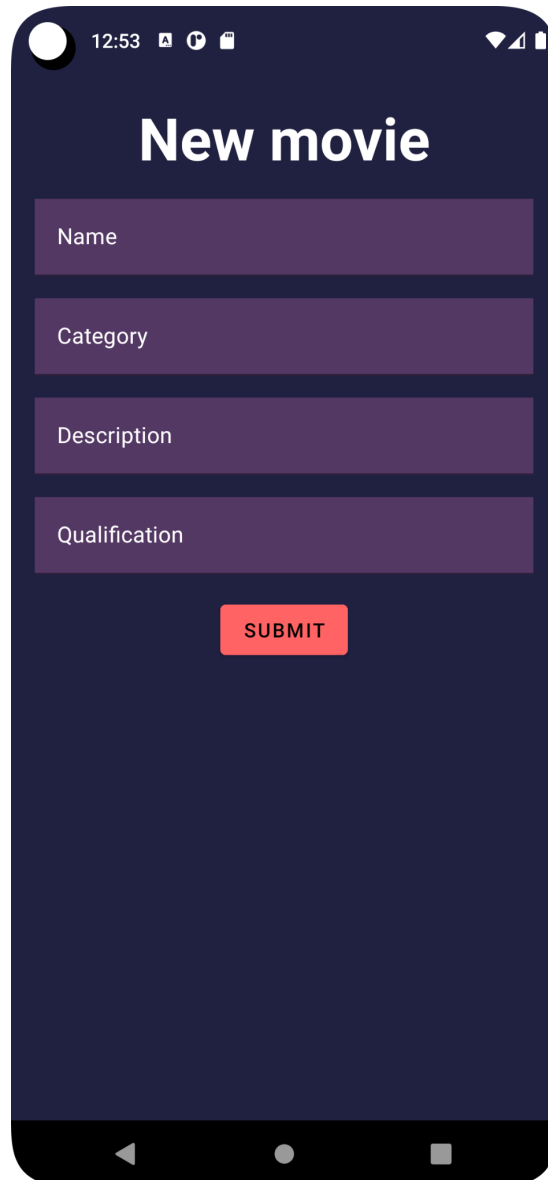
muestra el estado exitoso seguido de la lista completa de películas, se limpia el estado del ViewModel y se regresa al Fragmento anterior.



```
1 private fun observeStatus() {
2     movieViewModel.status.observe(viewLifecycleOwner) { status →
3         when {
4             status.equals(MovieViewModel.WRONG_INFORMATION) → {
5                 Log.d(APP_TAG, status)
6                 movieViewModel.clearStatus()
7             }
8             status.equals(MovieViewModel.MOVIE_CREATED) → {
9                 Log.d(APP_TAG, status)
10                Log.d(APP_TAG, movieViewModel.getMovies().toString())
11
12                movieViewModel.clearStatus()
13                findNavController().popBackStack()
14            }
15        }
16    }
17 }
```

4. Actividad a realizar

Se debe de agregar una nueva película haciendo uso de **LiveData** y **DataBinding**



The screenshot shows a mobile application interface with a dark blue background. At the top, the status bar displays the time 12:53 and various icons. The app's title bar is dark blue with the text "New movie" in white. Below the title bar, there are four input fields with light purple backgrounds and white text labels: "Name", "Category", "Description", and "Qualification". At the bottom of the form, there is a red button with the text "SUBMIT" in white. The bottom of the screen shows the Android navigation bar with three icons: a back arrow, a home circle, and a recent apps square.

5. To do:

1. ¿Qué **ventajas** tiene el utilizar **LiveData**?

2. ¿Qué **patrón de diseño** utiliza **LiveData**?

3. Menciona que se debe agregar al **Gradle** para poder implementar **DataBinding**

4. ¿Qué es **DataBinding**? Menciona sus ventajas frente a **findViewById**

9. Rúbrica

Actividad	Porcentaje	Nota
Uso de LiveData en MovieViewModel	10%	
Implementación en Gradle de DataBinding	5%	
Las interfaces de usuario son creadas a partir de DataBinding	25%	
Validación de datos dentro del ViewModel	15%	
Se verifica el estado del ViewModel para mostrar información en el Logcat	15%	
Preguntas teóricas	20%	
Puntualidad	10%	
Total	100%	