

Tema 3.4 - ViewModel

Descargar estos apuntes [pdf](#) o [html](#)

Índice

▼ Introducción

- ▼ Alcance de un ViewModel (ViewModel Scope)
 - Ciclo de vida de un ViewModel
- Instanciando un ViewModel
- Ejemplo de uso y definición de un ViewModel
- 🚫 Prácticas no recomendadas

Introducción

En anteriores temas cuando hablamos de la **arquitectura de aplicaciones de Android** ya mencionamos este componente como parte de la Capa de UI. En este tema vamos a centrarnos en él para concretar la implementación de MVVM en nuestra capa UI de la arquitectura.

ViewModel es una de esas clases que **Google** definió, allá por 2018, en la primera versión de **Jetpack** para ayudar a los desarrolladores a crear aplicaciones de Android más robustas y fáciles de mantener. **ViewModel** es una clase que está **diseñada para almacenar y administrar datos relacionados con la interfaz de usuario de una manera que sobrevive a los cambios de configuración**, como la rotación de la pantalla.

Los beneficios clave de la clase ViewModel son básicamente dos:


- Te permite conservar el estado de la IU.
- Proporciona acceso a la lógica empresarial, idealmente a través de casos de uso definidos en el dominio.

Alcance de un ViewModel (ViewModel Scope)


Cuando se crea una instancia de **ViewModel**, se pasa un objeto que implementa la interfaz **ViewModelStoreOwner**.

Los objetos que implementan **ViewModelStoreOwner** puede ser por ejemplo:

- Una **Activity** o **ComponentActivity**.
- Un **Fragment**.
- Un destino de Navigation **NavBackStackEntry**.

 **Importante:** El alcance de tu ViewModel se define en el Ciclo de vida del **ViewModelStoreOwner**. Esto es, Continúa en la memoria hasta que su **ViewModelStoreOwner** desaparece de forma permanente.

Cuando se destruye el fragmento o la actividad para los que se definió el alcance del ViewModel, el trabajo asíncrono continúa en el ViewModel específico. Esta es la clave de la persistencia.

 **Nota:** Cuando se definió la clase **ViewModel** en la primera versión de **Jetpack**, aún no existía **Compose** y se usaba asociado a una vista como un **Activity** o a un **Fragment**. Nuestras aplicaciones de Android se componían de una o más **Activity** o **Fragment** y cada uno de ellos tenía su propio **ViewModel** que se creaba como una **instancia única**. **ViewModel**

se usaba para almacenar datos que se necesitaban en la interfaz de usuario de la **Activity/Fragment** o el **Fragment** y queríamos que sobrevivieran a los cambios de configuración o queríamos compartir datos entre ellos.

Con la llegada de **Jetpack Compose** este enfoque ha cambiado y Google ahora **recomienda aplicaciones de actividad única** donde las diferentes pantallas se cargan como contenido dentro de la misma actividad. Por tanto, un **ViewModel** utilizado por una actividad permanece en memoria hasta que la actividad finalice esto es hasta que la aplicación finalice.

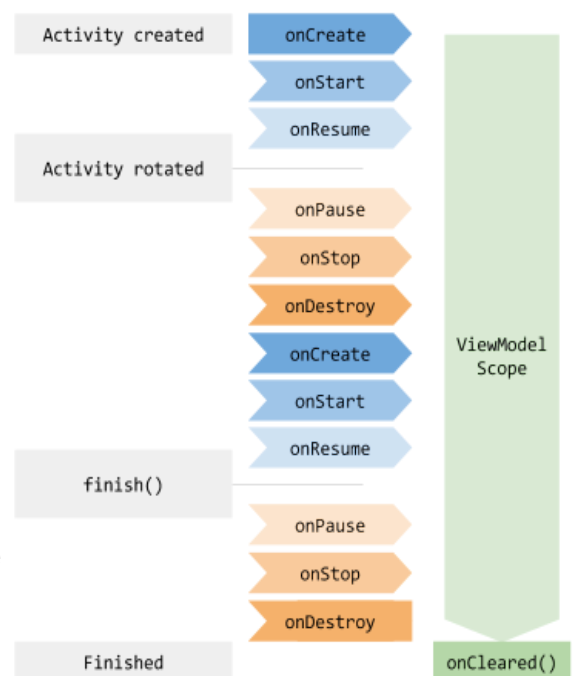
Ciclo de vida de un ViewModel

- En el caso de una **Activity**, hasta que termina. (**El más común**)
- ~~En el caso de un **Fragment**, cuando se desvincula.~~ (No se usa en Compose)
- En el caso de un **NavBackStackEntry**, hasta volvemos atrás en el grafo.

En la imagen de la derecha vemos que el objeto

ViewModel cuyo '**propietario**' (el **ViewModelStoreOwner**) es una **Activity** que tiene un cambio de configuración (rotación de pantalla) y por tanto se destruye y vuelve a crearse su vista asociada.

1. Se crea al llamarse al método **onCreate()** de la **Activity** **solo la primera vez** y aunque se vuelva a llamar a **onCreate()** por un cambio de configuración, **no se vuelve a crear** permanece el mismo objeto **ViewModel** que se creó en la primera llamada.
2. **No se destruye** aunque la **Activity**, destruya su vista y permanece en memoria hasta la finalización de la **Activity**. En ese momento se llama al método **onCleared()** del **ViewModel** para que realice las tareas de limpieza necesarias.



Instanciando un ViewModel

Hay muchas formas de instanciarlo y eso nos puede llevar a confusión. Pero nosotros vamos a usar la más sencilla y recomendada por Google. De todas formas, si quieres profundizar puedes ver el siguiente *'cheatsheet'* creado por los desarrolladores de Google.

1. Para definirlo deberemos crear una clase que herede de `ViewModel` y que implemente la lógica de negocio que necesitemos.

```
class MiViewModel : ViewModel() { ... }
```

2. ❌ No debemos hacer jamás una instancia directa.

```
val miVm = MiViewModel() // 💀 💀
```

Ya que el ciclo de vida de un ViewModel está asociado a un `ViewModelStoreOwner` y por tanto es la clase `ViewModelProvider` la que se encarga de crearlo y mantenerlo en memoria.

```
// activity es el objeto que implementa ViewModelStoreOwner
val miVm = ViewModelProvider(activity).get(MiViewModel::class.java)
```

👉 **Importante:** Sin embargo nosotros **no tendremos que crearlo así nunca** pues **Jetpack** nos proporciona formas más sencillas de hacerlo.

3. ⚠️ Como en el fondo debería ser `ViewModelProvider` quien lo cree, **si pasamos parámetros al constructor** de la clase `MiViewModel`, no podremos tener forma más sencilla de crearlo. Por tanto, de momento, no deberíamos pasar parámetros al constructor.

```
class MiViewModel(val param: String) : ViewModel() { ... } // 💀 💀
```

Si necesitáramos pasar parámetros al constructor, deberemos crear una clase que implemente `ViewModelProvider.Factory` y que se encargue de crear el objeto `MiViewModel` como se indica en la [documentación oficial](#).

👉 **Importante:** Ya veremos más adelante que la librería **Hilt** (Jetpack) para inyección de dependencias me facilita mucho esta tarea. Así pues, no tendremos nunca que crear una clase que implemente `ViewModelProvider.Factory` cuando pasemos parámetros.

4. Delegado de creación `by viewModels()` , si el propietario es una `Activity` .

```
class MainActivity : ComponentActivity() {  
    2    // Opción 1  
        // Lo defino como propiedad de la clase y delego su creación que será al ser usado  
        // primera vez después de llamarse el método onCreate() de la Activity y puede ser  
        // accedido desde cualquier método de la Activity que usemos después del onCreate  
        // El delegado internamente llama a ViewModelProvider  
    7    val miVm: MiViewModel by viewModels()  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
    12    // Opción 2  
        // Lo defino en el método onCreate() y lo clausuramos al definir el árbol de  
        // seguirá siendo destruido por ViewModelProvider al destruirse la  
        // Activity que es el ViewModelStoreOwner  
    16    val miVm: MiViewModel by viewModels()  
  
        setContent {  
            // Clausura de miVm  
        }  
    }  
}
```

5. Creación con `viewModel()` , si estamos dentro de una función `@Composable`

El `ViewModelStoreOwner` será la `Activity` que renderiza la composición. No importa que `viewModel()` sea llamado más veces a lo largo de las recomposiciones pues solo se instanciará en la primera llamada.

```
@Composable  
fun MiScreen(miVm: MiViewModel = viewModel()) { ... }
```

🚩 **Nota:** Existen más formas de crear un `ViewModel` que veremos más adelante.

Ejemplo de uso y definición de un ViewModel

TODO: AQUI VA AHORCADO EXPLICADO

👉 Prácticas no recomendadas

Las siguientes son varias prácticas recomendadas clave que debes seguir cuando implementes

`ViewModel` :

- ❌ **NO** defines `ViewModel` para *composables* reutilizables en tu UI o para componentes de IU que no sean de nivel superior. Deberíamos definirlos a nivel de Screen (pantalla).
- ❌ Los ViewModels **NO deberían conocer los detalles de implementación de la IU**. Mantén los nombres de los métodos que expone la API de ViewModel y los de los campos del UIState lo más genéricos posible.
- ❌ Como pueden tener una vida más larga que el `ViewModelStoreOwner`, los ViewModels **NO deberían contener ninguna referencia de APIs relacionadas con el ciclo de vida**, como `Context` o `Resources` para evitar fugas de memoria.
- ❌ **NO pases ViewModels a funciones ni otros componentes de la IU**. Esto evita que los componentes de nivel inferior accedan a más datos y lógica de los que necesitan.
- ❌ Derivado del anterior **NO instances directamente un objeto ViewModels**. En su lugar, usa algún tipo de '*proveedor de ViewModels*' para obtener una instancia del mismo.