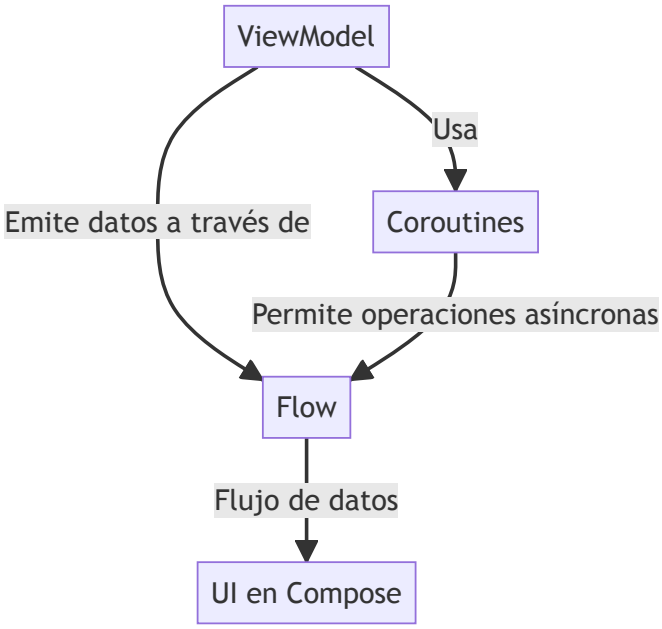
90. ViewModel

Seguir en Android Developer AD

¿Que necesitamos conocer para usar ViewModel en Android ?

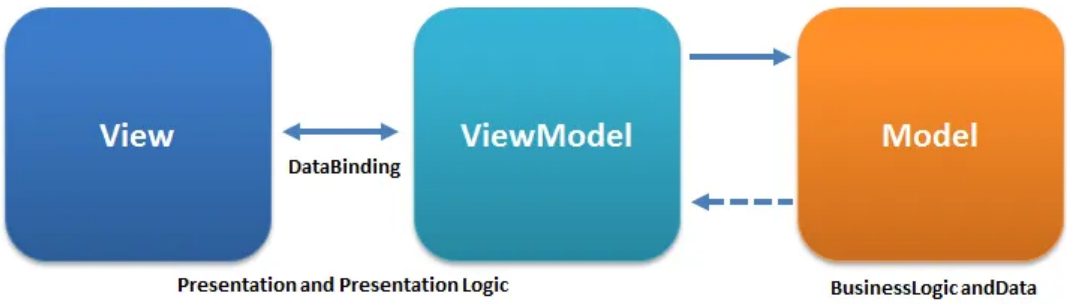
- Coroutines ver en el tama 8.2
- State
- Flow más detalles en el tema 8.3 y StateFlow

En el siguiete esquema se presentan las relaciones entre estos elementos.



90.1 Introducción

Sigue el patrón de diseño MVVM (modelo-vista-vista-modelo)



El principal objetivo de este patrón de diseño, es separar la presentación gráfica de la lógica de negocio (modelo).

Resumen:

Aspecto	Detalles
¿Qué es ViewModel?	**Clase** de Android Jetpack para almacenar y gestionar datos de la UI de manera eficiente y *consciente del ciclo de vida*. Permite que los datos sobrevivan a cambios de configuración como rotaciones de pantalla.
Características	- **Persistencia de Datos**: Mantiene los datos durante cambios de configuración. - **Separación** de Responsabilidades: Separa la lógica de la UI de la lógica empresarial. - **Gestión del Ciclo de Vida**: Consciente del ciclo de vida de actividades y fragmentos, previene fugas de memoria.
Uso Básico	1. Creación: Se crea una instancia de `ViewModel` utilizando `ViewModelProviders` (forma clásica) o derivando de la clase `ViewModel`, usada con Compose. 2. Observación: Los datos en `ViewModel` pueden ser observados por la UI. 3. Sobrevive a Cambios: Mantiene los datos a pesar de cambios de configuración.

90.2 La clase ViewModel

ViewModel es la clase responsable de preparar los datos y la lógica de negocio para la Activity.
Un ViewModel incluye un `scope` que mantendrá durante todo la vida, hasta finalizar la Activity. Esto significa que el ViewModel persiste al cambio de configuración (estado `destroyed` de la Activity) como ocurre con la rotación de la pantalla. En pocas palabras es independiente del ciclo de vida de la Activity.

La Activity puede observar los cambios en el ViewModel usando `LiveData` o `StateFlow`. ViewModel es el responsable de tratar con los datos y **nunca debe acceder al IU**.

Ejemplo de uso con Compose y kotlin:

```
class MiActividad : ComponentActivity()() {

    private val viewModel: MiViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.mi_actividad_layout)
        // Aquí puedes usar tu viewModel
    }
}
```

Un ejemplo de uso **tradicional** con Vistas y java:

```
public class UserActivity extends Activity {

    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.user_activity_layout);

        final UserModel viewModel = new ViewModelProvider(this).get(UserModel.class);

        viewModel.getUser().observe(this, new Observer<User>() {
            @Override
            public void onChanged(@Nullable User data) {
                // update ui.
            }
        });
        findViewById(R.id.button).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                 viewModel.doAction();
            }
        });
    }
}
```

90.3 Clases e Inteface relacionados con ViewModel

1. **ViewModel** : Es la clase principal que extiendes para crear tu propio ViewModel. Guarda tus datos y los sobrevive a cambios de configuración.
2. ```
   class MyViewModel : ViewModel() {
       // Tus datos y lógica aquí
   }
   ```

Poe oreo lado, en el uso **tradicional** tenemos las siguientes clases:

1. **ViewModelProvider**: Se utiliza para crear y recuperar ViewModels. Se encarga de proporcionar ViewModels para una scope específica, como una actividad o un fragmento.
2. **ViewModelStore**: Proporciona el almacenamiento para ViewModels. Cada actividad o fragmento tiene su propio ViewModelStore.
3. **ViewModelFactory**: Si necesitas pasar parámetros al ViewModel, puedes usar una Factory para crear instancias de ViewModel con esos parámetros.

```
class MyViewModelFactory(private val myParameter: String) : ViewModelProvider.Factory {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return MyViewModel(myParameter) as T
    }
}
```

90.4 LiveData y StateFlow

Para la comunicación entre viewModel y las pantallas gráficos podemos usar LiveDat o StateFlow. Actualmente el preferido para usar es ComponentActivity()

1. **LiveData y StateFlow**: A menudo, los ViewModels se usan con LiveData o StateFlow para observar los datos. LiveData es una clase observable que sabe en qué ciclo de vida se encuentra, mientras que
```

StateFlow es parte de Kotlin Coroutines y es más moderno.

Hay otros elementos que veremos en el tema de persistencia como es los **Repositories**: Los ViewModels a menudo interactúan con los Repositories para separar la lógica de negocio de la UI. Los Repositories se encargan de manejar datos, como operaciones de base de datos o llamadas a la red.

StateFlow es una clase especializada de Flow. Se utiliza cuando hay un único valor que cambia. Veamos en el siguiente ejemplo como se implementa:

- Se utiliza **MutableStateFlow** para mantener un estado mutable dentro del ViewModel. La propiedad es privada para evitar cualquier modificación fuera del ViewModel.

```
class ExampleViewModel : ViewModel() {

    // MutableStateFlow para actualizar los datos internamente
    private val _data = MutableStateFlow("Initial Value")

    // StateFlow para exponer los datos inmutables a la UI
    val data: StateFlow<String> = _data

    fun updateData(newValue: String) {
         // Actualizar el valor de _data
         _data.value = newValue

    }
}
```

- **StateFlow** se expone a la UI para observar los cambios. Es inmutable desde el punto de vista de los consumidores.

```
El objetivo de comunicaciones unidireccionales se obtiene forzando a que toda
modificación del estado se hace exclusivamente deede el ViewModel
Y mediante estados observables en los Composable se obtienn los valores
```

En el Composable:

Podemos observar el `stateFlow` en compose:

- Creamos un `ViewModel` utilizando un delegado `viewModels<>` que permanece entre cambios de configuración.
- `val message by viewModel.mensajeFlujoEstado.collectAsState()`
  Aquí se observa un StateFlow del viewModel llamado a una de sus propiedades de clase: `mensajeFlujoEstado`. El método `collectAsState` convierte el StateFlow en un estado componible que se actualiza automáticamente cuando el StateFlow emite nuevos valores.

```
@Composable
fun MainScreen(viewModel: MainViewModel) {

    val message by viewModel.stateFlowMessage.collectAsState()

    Text(
        text = message
     )
}
```

`Flow<T>.collectAsState()` establece una subscripción como observador de `viewModel`. Este método convierte el StateFlow en un estado observable por Compose. Cada vez que el StateFlow emite un nuevo valor, el Composable se recompondrá con el valor actualizado.

El ViewModel sobrevive a los cambios de configuración y por tanto conserva los valores en `StateFlow`

90.5 Otro forma de usar Flow

Es necesario una programación asíncrona en ciertos casos como cuando accedemos a Internet o a una base de datos.

En estos casos usamos una corrutina para obtener los datos y evitar que toda la aplicación quede pendiente a la espera.

Por ejemplo, leemos un sensor de temperatura cada cierto periodo de tiempo y lo mostramos en la pantallla.

Para el emisor en el ViewModel:

```
class SensorVM: ViewModel() {
    var empezado by mutableStateOf(false)

    // flows
    private val  _temperatura = MutableStateFlow(0f)
    val temperaturaFlujoEstado = _temperatura.asStateFlow()

    fun empezar(){
        empezado = false
        viewModelScope.launch {
            while(true) {
                val numero = Random.nextFloat()
                if(empezado)
                    _temperatura.emit(numero)
                delay(1000)
            }
        }

    }
    fun onConmutar(){
        empezado = !empezado
    }
}
```

* `by viewModels()` proporciona una instancia del ViewModel. * `lifecycleScope.launchWhenStarted` lanza una corutina que observa el StateFlow. * `collect` se utiliza para recibir los valores actualizados del StateFlow.

En la parte de IU se conecta el StateFlow

```
@Composable
fun PantallaVM(vm: FrutasVM, sensor: SensorVM){
// conectamos el estdado de "mensaje" de la pantalla con el modelo
    val mensaje by vm.mensajeFlujoEstado.collectAsState()
    val temperatura by sensor.temperaturaFlujoEstado.collectAsState()
```

```
    Column(
        modifier = Modifier.fillMaxWidth(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.SpaceEvenly,
    ) {


        Button(onClick = { sensor.onConmutar() }) {
            Text(text ="Leer Temp ")
        }
        Text( text= "Temperatura : " +temperatura )
    }
}
```

90.6 Diferencias entre CollectAsState y collectAsStateWithLifecycle

(de este artículo https://blog.protein.tech/exploring-differences-collectasstate-collectasstatewithlifecycle-7fde491110c0o)

90.7 Ejemplos Calculadora con estados,

Ejemplo con varios casos en villablanca github

Ejemplo 1 con estados: En una ventana con dos campos de entrada y un Text de salida. Cada vez que se modifica los campos de entrada se refleja su suma en el de salida.

Ejemplo 2 con ViewModel:
Misma calculadora implementada con viewModel

Ejemplo 3 con ViewModel

Dos emisores, uno de temperaturas y otro un elemento de una lista aleatoria

90.8 Tutorial viewmodel "unscrumble"

tutorial : Se utiliza para crear y recuperar ViewModels. Se encarga de proporcionar ViewModels para una scope específica, como una actividad o un fragmento

90.9 Resumen para utilización de ViewModel y StateFlow

1. Creamos una clase para los estados: **MyUiState**
2. Creamos la clase MyViewModel derivada de ViewModel
3. Definición de StateFlow: Primero, se define un StateFlow para ser usado en ViewModel.

   Por ejemplo:

```
private val _uiState = MutableStateFlow(MyUiState())
val uiState: StateFlow<MyUiState> = _uiState.asStateFlow()
```

Aquí, MyUiState es una clase que representa el estado de la UI. 2. Actualización del Estado: Para actualizar el estado, se modifica el valor del _uiState, que es un MutableStateFlow. Esto se hace generalmente en respuesta a eventos de la UI o cambios en los datos. En la clase de ViewModel

```
fun updateData(newData: Data) {
    _uiState.update{ it.copy(data = newData)}
}
```

3. *Observación en Compose*: En su UI Compose, observa los cambios en el StateFlow usando el método `collectAsState()`. Esto se hace generalmente en un @Composable:

```
@Composable
fun MyScreen(viewModel: MyViewModel) {
    val uiState = viewModel.uiState.collectAsState()

    // Use uiState.value para construir la UI
}
```

4. Consideraciones de **Concurrency**: StateFlow maneja la concurrencia internamente, asegurando que las actualizaciones de estado sean seguras en términos de hilos. 5. Uso de StateFlow en Jetpack Compose: StateFlow es especialmente útil en Jetpack Compose debido a su naturaleza reactiva, lo que facilita la creación de interfaces de usuario que responden dinámicamente a los cambios en el estado de la aplicación. ## Nota y observaciones. No necesitamos pasar a cada función el viewModel. disponemos de varias formas : 1. Obtener el ViewModel Directamente en el Composable: Puedes llamar la función `viewModel()` directamente dentro del Composable. Esto creará una instancia del ViewModel o proporcionará la existente si ya ha sido creada. La función viewModel() es inteligente y se asegurará de que el ViewModel sobreviva a los cambios de configuración, como las rotaciones de pantalla. Ejemplo:

```
val appBarViewModel: AppBarViewModel = viewModel()
```

2. Uso de **ViewModelProvider.Factory ** (Opcional): Si tu ViewModel necesita parámetros específicos, puedes usar un `ViewModelProvider.Factory` para crearlo. Esto es útil si necesitas pasar argumentos al constructor del ViewModel:

```
val appBarViewModel: AppBarViewModel = viewModel(factory = MiFactory)
```

Es importante asegurarse de la Coincidencia del Alcance del ViewModel: Si tu ViewModel debe compartirse entre varios composables, asegúrate de que todos accedan al mismo alcance. Por ejemplo, si el ViewModel debe ser compartido a nivel de actividad, todos los composables deberían obtenerlo de la misma forma. ## Apendice Enlaces: * https://decode.agency/article/kotlin-flows-guide/ * [Descripcion general ViewModel] (https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419) * https://developer.android.com/kotlin/flow/stateflow-and-sharedflow?hl=es-419 * Terminos usados: * IU : Interfaz de usuario * AD : Web de Android Developer Versión 0.5 10-12-23 Versión 0.9 8-12-24