Apuntes

Tema 5 ViewModel: Conexión del modelo y la vista

Descargar estos apuntes pdf o html

Índice

Recordando nuestra arquitectura

El *MVVM* es una arquitectura que se ha hecho muy popular desde que *Google* la convirtiera en su arquitectura oficial al lanzar la guía de arquitecturas.

La hemos explicado y personalizado en un tema anterior.

De forma muy resumida *MVVM* se compone de tres componentes principales:

- *Model*: representa la capa de datos, que se encarga de la obtención y gestión de datos, ya sea a través de servicios web, bases de datos, archivos, etc.
- *View*: representa la capa de interfaz de usuario, que se encarga de mostrar los datos al usuario y de gestionar las interacciones de éste con la aplicación.Nosotros la desarrollamos con *Jetpack Compose*.
- *ViewModel*: actúa como intermediario entre la capa de datos (*Model*) y la capa de interfaz de usuario (*View*), exponiendo los datos y comportamientos necesarios para la vista. También es responsable de manejar las acciones del usuario y actualizar el modelo en consecuencia.

La idea principal detrás de *MVVM* es que los desarrolladores puedan crear una interfaz de usuario más modular, escalable y fácilmente testeable. Al separar la lógica de la interfaz de usuario y la lógica de negocio, se hace más fácil mantener y actualizar el código. Además, *ViewModel* permite reutilizar la lógica de negocio entre varias vistas, lo que puede reducir el tiempo de desarrollo y aumentar la calidad del código.

¿Qué es ViewModel?

El `ViewModel` es un componente de la arquitectura de *Android* que se utiliza para almacenar y administrar los datos relacionados con la *UI*, y sobrevive a los cambios de configuración de la actividad o fragmento que lo contiene.

El objetivo principal del ViewModel es separar la lógica de la vista (*Activity, Fragment*) y permitir que la vista se centre en la presentación de los datos. Además, los datos almacenados en el `ViewModel` pueden sobrevivir a los cambios de configuración, como rotaciones de pantalla, lo que hace que las aplicaciones sean más robustas y evita la pérdida de datos.

El `ViewModel` se puede utilizar para compartir información entre *Activitys*, *Fragments* o funciones *composables* en una aplicación. Cuando se utiliza un `ViewModel`, se crea una instancia del mismo que es compartida por todos los componentes que lo soliciten, por lo que puede utilizarse para mantener datos y estado de la aplicación de forma centralizada.

El uso de `ViewModel` requiere de la intervención de `LiveData` que es una clase de *Android* que se utiliza para emitir datos de forma reactiva y observable. La idea principal es que un objeto `LiveData` emite datos y los componentes de la aplicación que están suscritos a él reaccionan a esos cambios. Esto ayuda a mantener una separación adecuada entre la lógica de la aplicación y la interfaz de usuario.

La combinación de estos dos elementos nos permite actualizar automáticamente la interfaz de usuario con los datos más recientes, sin necesidad de preocuparnos por la gestión manual de la sincronización de datos y la actualización de la interfaz de usuario.

```
class MyViewModel : ViewModel() {
    private val _counter = MutableLiveData(0)
    val counter: LiveData<Int> = _counter
    //val counter: LiveData<Int> get() = _counter

    fun incrementCounter() {
        _counter.value = _counter.value?.plus(1) ?: 0
    }
}

/*
MutableLiveData es una clase proporcionada por el framework de Android que es una variante de
LiveData. A diferencia de LiveData, que es inmutable y solo puede ser observado, MutableLiveD
permite cambios en su valor.

_counter, que es un MutableLiveData<Int> inicializado con un valor de 0.

counter de tipo LiveData<Int> se inicializa con la instancia _counter, lo que permite que otr
componentes observen su valor pero no lo modifiquen directamente.
*/
```

Llevar el estado al ViewModel

Vamos a partir de nuestra aplicación *¡Hola mundo! con los dos botones* + y - que permitían incrementar o decrementar el contador. He creado un nuevo proyecto y adaptado el código a nuestra arquitectura creando un *feature* llamado *SaludoDosBotones* que incluye la definición de la vista y el *ViewMModel* para los datos.

Si recordamos, teníamos nuestra función `@Composable` *MyApp* con el siguiente código:

```
@Composable
fun MyApp(){
   var cont = remember { mutableStateOf(0) }

   MyScreen(
       cont,
       onButtonIncrement={cont.value++},
       onButtonDecrement={cont.value--})
}
```

Habíamos aplicado el patrón *State Hoisting* y definido el estado de la variable `cont` y definido las funciones lambda que se pasábamos como parámetros para la recomposición de la vista.

El siguiente paso para que nuestra aplicación siga el patrón de *MVVM* es llevar este estado al `ViewModel`.

Necesitamos crear una instancia para el `ViewModel` y modificar la referencia que teníamos de nuestro `State`.

```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //Se crea una instancia del ViewModel (nuestro ContadorViewModel)  y se asocia al cic
        //vida del componente, en nuestro caso nuestra Activity

        val viewModel by viewModels<SaludoDosBotonesViewModel>()

        setContent {
            B5_ViewModel_1Theme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    //Referenciamos la variable counter de nuestro Viewmodel para controlar e
                    //a través de la función observableState(), de esta forma cont permitirá
                    // recomposición de la View donde se encuentre.
                    val cont by viewModel.counter.observeAsState(0)
                    SaludoDosBotones(
                        cont,
                        onButtonIncrement={viewModel.incrementCounter()},
                        onButtonDecrement={viewModel.decrementCounter()}
                    )
                }
            }
        }
    }
}
```

📌 **NOTA**: Recordar añadir la función **`decrementCounter()`** al **`ViewModel`** :

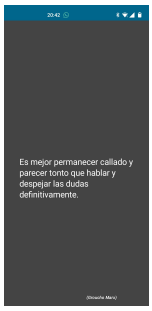
```
fun decrementCounter() {
        _counter.value = _counter.value?.minus(1) ?: 0
    }
```

Capa de datos

Vamos a ampliar la utilización de los elementos que hemos definido en nuestra arquitectura, concretamente vamos a introducir *datos* a nuestra aplicación.

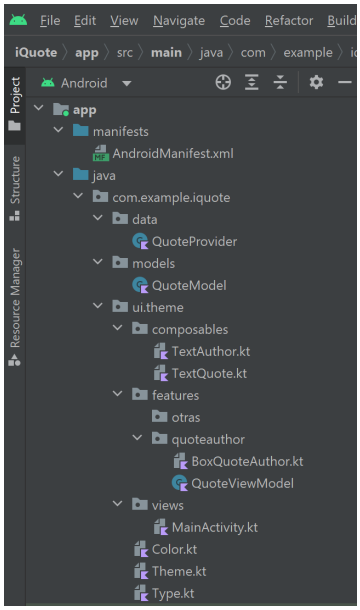
Vamos a diseñar una aplicación que muestre por pantalla una cita y el autor, de forma que cuando pulsemos sobre la cita, nos muestre otra cita aleatoria.

El diseño de la *UI* denuestra aplicación será el siguiente:



Creamos un nuevo proyecto que se llame **iQuote*.

Para el diseño de nuestra *UI* y basándonos en el modelo de arquitectura que vimos en el tema3, necesitaremos la siguiente estructura:



Vamos a ir comentando cada uno de estos elementos.

Nuestra *MainActivity.kt* que ahora está almacenada en la carpeta *com.example.iquote\ui.theme\views*

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            IQuoteTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                   modifier = Modifier.fillMaxSize(),
                    color = Color.DarkGray
                ) {
                    //Este elemento sería un features
                    BoxQuoteAuthor()
                }
            }
        }
    }
}
```

La función `BoxQuoteAuthor` sería un elemento del tipo *features* de nuestra arquitectura. Recoge el diseño de nuestra *UI* aunque podría ser solo de una parte de ella. Está definida en el archivo *BoxQuoteAuthor.kt*. En esta función llamamos a dos elementos *composables* `TextQuote()` y `TextAuthor`. Y se almacena con ruta y nombre *\ui.theme\features\quoteauthor\BoxQuoteAuthor.kt*.

```
@Composable
fun BoxQuoteAuthor(){
    Box(
        modifier = Modifier.fillMaxSize(),
    ){
        TextQuote(quote="La tapa del cementerio es una insenatez, los que están dentro no pue

        Spacer(modifier =  Modifier.height(90.dp))

        TextAuthor(author = "Anónimo",modifier = Modifier.align(Alignment.BottomEnd))
    }
}
```

La definición de cada elemento *composable* está en un archivo distinto dentro de la carpeta *\composables*.

TextQuote.kt

```
@Composable
fun TextQuote(quote:String,modifier:Modifier) {
    //Introducimos ya la propiedad .clickable del Text para posteriormente añadirle la funcic
    Text(
        modifier=modifier
            .clickable {  }
            .padding(40.dp),
        text = quote,
        style=MaterialTheme.typography.titleLarge,
        color= Color.White
    )
}
```

TextAuthor

```
@Composable
fun TextAuthor(author:String,modifier:Modifier) {
    Text(
        modifier=modifier.padding(90.dp),
        text = author,
        style = MaterialTheme.typography.labelSmall+
                TextStyle(
                    fontStyle = FontStyle.Italic
                    //fontWeight = FontWeight.Bold
                ),
        textAlign = TextAlign.End,
        color= Color.White
    )
}
```

Si probamos nuestro proyecto debaría salir la cita que se está pasando como parámetro por pantalla. Al hacer *click* sobre ella no sucede nda porque no hemos añadido ninguna funcionalidad.

Lo que queremos ahora es que cada vez que se pulse sobre la cita, la aplicación nos muestre otra distinta. Empecemos.

Vamos a definir una `data class` para nuestros datos. En Kotlin, una `data class`, es una clase especial que se utiliza para representar un modelo de datos que contiene solo propiedades. La clase de datos es muy útil para crear objetos inmutables y garantizar que las propiedades no puedan ser modificadas después de que se creen los objetos. Además, la `data class` proporciona varios métodos de utilidad, como `toString()`, `equals()`, `hashCode()`, que se generan automáticamente y simplifican el trabajo con objetos de datos.

Nuestra `data class` es muy sencilla porque solamente necesitamos almacenar la cita y el autor. Y se almacena con ruta y nombre *\models\QuoteModel.kt*

```
data class QuoteModel (val quote:String, val author:String)
```

Para simplificar esta primera aproximación a una aplicación con datos, definiremos nuestro propio proveedor de citas, es decir, una clase con varias citas con un método que aleatoriamente devolverá una u otra. Lo no más normal sería obtener estos datos de una *BD* o de una página *Web* leyendo una estructura *XML* o *JSON*. Llamaremos a nuestra clase proveedora de datos `QuoteProvider`. Y se almacena con ruta y nombre *\data\QuoteProvider.kt*

```
class QuoteProvider {
    //companion object es un objeto dentro de una clase que permite acceder a sus propiedades
    companion object {
        fun random(): QuoteModel {
            val position = (0..10).random()
            return quotes[position]
        }

        private val quotes = listOf(
            QuoteModel(
                quote = "La vida es lo que sucede cuando estás ocupado haciendo otros planes.
                author = "(John Lennon)"
            ),
            ...
            QuoteModel(
                quote = "Hay 10 tipos de personas, los que leen binario y los que no.",
                author = "(Yo mismo)"
            )
        )
    }
}
```

Es el momento de definir nuestro `ViewModel` y una vez definido utilizarlo en niestra aplicación como intermediario entre la *UI* y los datos.
Se almacena en la ruta donde tenemos esta *feature* implementada, con ruta y nombre *\ui.theme\features\quoteauthor\QuoteViewModel.kt*

```
class QuoteViewModel : ViewModel() {
    //Almacena la cita actual
    val _quoteModel = MutableLiveData<QuoteModel>()
    //Variable para acceder a la cita actual, actúa como un get
    var quoteModel: LiveData<QuoteModel> =_quoteModel

    init {
        randomQuote()
    }
    //Esta función nos devuelve una cita aleatoria de nuestro proveedor y la almacena en el c
    //La cita actualizada se notifica automáticamente a cualquier componente que esté observa
    fun randomQuote() {
        val currentQuote = QuoteProvider.random()
        _quoteModel.postValue(currentQuote)
    }
}
```

Empecemos a utilizar el `ViewModel` y a jugar con `State`.

```
class MainActivity : ComponentActivity() {
    //Referenciamos nuestro ViewModel
    private val quoteViewModel by viewModels<QuoteViewModel>()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            CitasyfrasesTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                   modifier = Modifier.fillMaxSize(),
                    color = Color.DarkGray
                ) {
                    //Aplicamos patrón Stete Hoisting, pasando en ViewModel para recordar el
                    //Y una función lambda donde indicamos qué haremos cuando pulsemos sobre
                    BoxQuoteAuthor(quoteViewModel){
                        quoteViewModel.randomQuote()
                    }
                }
            }
        }
    }
}
```

Vamos a modificar ahora el código de *BoxQuoteAuthor.kt* con la nueva definición de su cabecera con los parámetros vistos anteriormente y definiendo el `State`

```
@Composable
fun BoxQuoteAuthor(viewModel:QuoteViewModel,onQuoteNew:()->Unit){
    // Definimos o convertinos a State el valor del LiveData quoteModel de nuestro ViewModel.
    //Cuando se produce un cambio en la emisión de quoteModel, la función observeAsState() se
    //Luego, cualquier composición que haga uso de q se volverá a dibujar automáticamente par
    val q by viewModel.quoteModel.observeAsState()

    Box(
        modifier = Modifier.fillMaxSize(),
    ) {
        //Pasamos a TextCita la cita (String)
        TextCita(quote = q?.quote ?: "Cargando cita...",
            modifier = Modifier.align(Alignment.Center),
           ){
            onQuoteNew()
        }
        Spacer(modifier =  Modifier.height(90.dp))
        //Pasamos a TextAuthor el autor de la cita (String)
        TextAuthor(author = q?.author ?: "...",modifier = Modifier.align(Alignment.BottomEnd)
    }
}
```

Y ahora los *composable*.

TextQuote.kt

```
@Composable
fun TextCita(quote: String,modifier:Modifier,onQuoteNew:()->Unit) {
    Text(
        modifier=modifier
                .padding(40.dp)
                .clickable {onQuoteNew()},
        text = quote,
        style=MaterialTheme.typography.titleLarge,
        color= Color.White
    )
}
```

TextAuthor.kt

```
@Composable
fun TextAuthor(author:String,modifier:Modifier) {
    Text(
        modifier=modifier.padding(90.dp),
        text = author,
        style = MaterialTheme.typography.labelSmall+
            TextStyle(
                fontStyle = FontStyle.Italic
            //fontWeight = FontWeight.Bold
            ),
        textAlign = TextAlign.End,
        color= Color.White
    )
}
```

✏️ Ejercicio Propuesto CardRecipe. Con pulsación sobre *Likes* y elevar el estado al padreViewModel.

👍
📌
✋
💡
📣