Programación Multimedia y Dispositivos Móviles

UD 12. Patrón MVVM y Clean Architecture

Javier Carrasco

Curso 2024 / 2025

Patrón MVVM y Clean Architecture

12. Patrón MVVM y Clean Architecture

El patrón de arquitectura software **MVVM**, de las siglas en inglés *Model-View-ViewModel*, se convirtió en un estándar para las aplicaciones Android desde que Google lanzó su guía de arquitectura de aplicaciones[1].

Este patrón es una evolución de MVC (*Model-View-Controller*), en el cual, la filosofía principal de este patrón sigue siendo la separación de la lógica de negocio de la interfaz de usuario, esto facilitará el desacoplamiento del código, su mantenimiento y escalabilidad, además de la realización de pruebas.

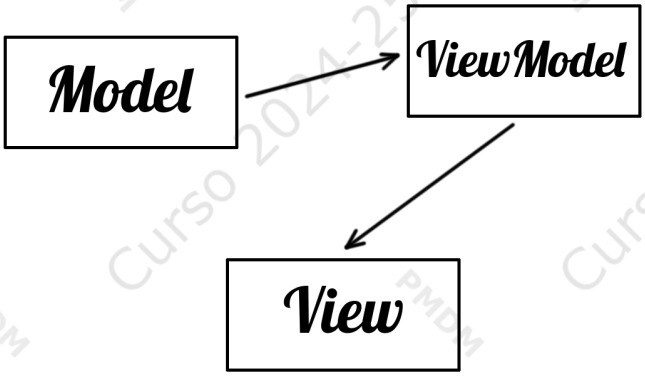

Figura 1

Los tres componentes que forman el patrón de arquitectura MVVM son:

- **Modelo** (*Model*): representará la capa de datos o lógica de negocio. Únicamente contendrá la información, no habrán métodos o acciones que manipulen los datos y, no tendrá ninguna dependencia de la vista.
- **Vista** (*View*): será la parte encargada de representar la información al usuario. En el patrón MVVM, las vistas son activas, reaccionando a eventos o cambios de los datos.
- **Modelo de vista** (*ViewModel*): es el intermediario entre el modelo y la vista, conteniendo la lógica de presentación y abstracción de la interfaz. El enlace con la vista se realizará mediante el enlace de datos.

Adaptando el patrón MVVM al desarrollo de aplicaciones móviles en Android, sus principios básicos son dos:

- **Separación de problemas**, debe evitarse algo muy común, escribir todo el código en las actividades o fragmentos, dejando ahí únicamente el código necesario para representar la información. Debes tener en cuenta que, las actividades y fragmentos tienen su propio ciclo de vida, por tanto pueden verse afectadas por cambios del SO, como falta de memoria, por ejemplo. Se recomienda reducir la dependencia de estas vistas.

1 Guía de arquitectura de apps (https://developer.android.com/jetpack/guide)

- **Controlar la UI a partir de un modelo**, ten en cuenta que los modelos proporcionarán la información a las vistas, y serán independientes de las vistas y otros componentes de la aplicación, por lo tanto no se ven afectados por los ciclos de vida.

En ocasiones, al intentar aplicar el MVC o MVP en una aplicación, se acaba cometiendo errores, como añadir funciones en lugares donde no corresponden, como en las vistas. La aplicación del modelo MVVM, como se ha descrito, separa las vistas de la lógica de negocio de la aplicación, esto funciona muy bien para aplicaciones no muy grandes.

Si además de la aplicación del patrón MVVM, se aplican conceptos de *Clean Architecture* se conseguirá mayor independencia entre módulos y proyectos más compactos. El uso de *Clean Architecture* se basa en la estructuración del código por capas, donde cada una de estas capas se comunicará con sus capas más cercanas. Además, cada una de estas capas tendrá un único objetivo, separando responsabilidades. Esta combinación permitirá soportar el crecimiento de la aplicación de manera más fiable.

Las capas comunes de *Clean Architecture* son:

1. **Presentación**, esta es la capa que interactúa directamente con el usuario.
2. **Casos de uso**, capa que suele contener las acciones que el usuario puede activar.
3. **Dominio**, contiene la lógica de negocio, suele contener los modelos, por ejemplo las clases *SuperHero* y *Editorial*.
4. **Datos**, esta capa contiene las definiciones de la fuente de datos y cómo se utilizará. Puede no limitarse a una única fuente de datos. Se suele utilizar el patrón repositorio para decidir que fuente de datos (*DataSource*) utilizar.
5. ***Framework***, esta capa define las distintas fuentes de datos, por ejemplo, Room en modo local o una API de forma remota.

El diagrama clásico que representa la *Clean Architecture* creado por Robert C. Martin es posible que ya lo hayas visto.

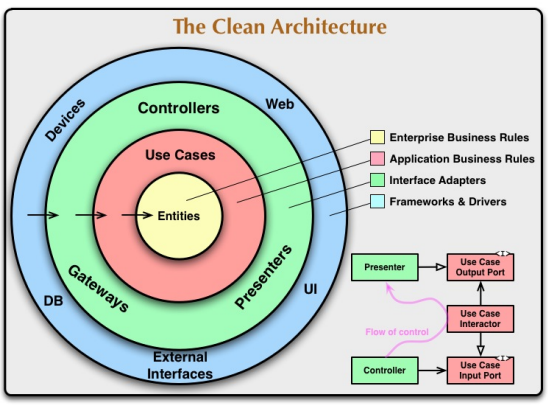

Figura 2

Evidentemente, puede hacerse una libre interpretación de la arquitectura, eliminando capas, unificando, etc. Esto es así porque no es realmente una arquitectura como tal, sino una guía con recomendaciones a seguir. En Android es muy común unificar las capas, lo que además permitirá simplificar el modelo.

- Capa **presentación**, donde se aplicará MVVM.
- Capa **dominio**, contendrá el modelo de negocio y los casos de uso.
- Capa **datos**, se utilizará el modelo repositorio y el acceso a datos.

La comunicación entre todos los componentes de las capas será la siguiente.

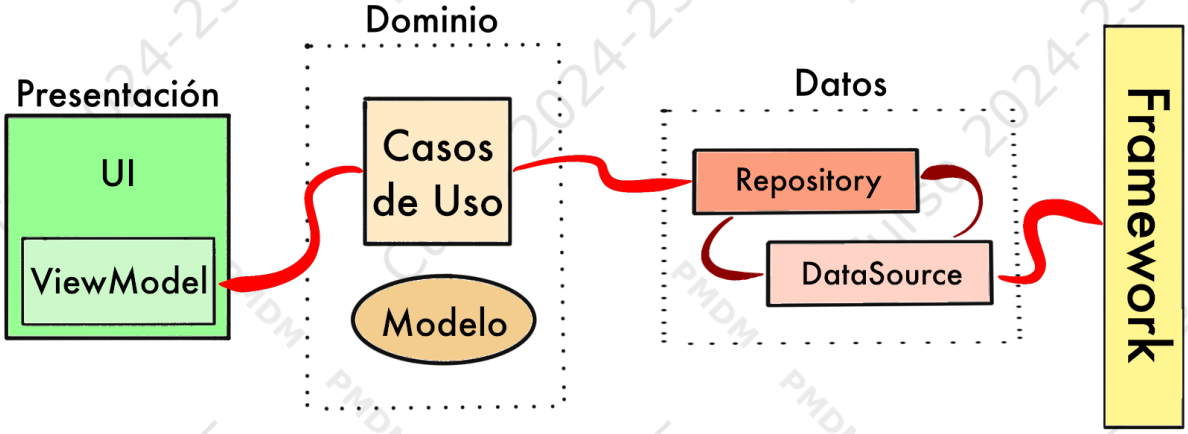

Figura 3

Para intentar comprender el uso y aplicación del patrón MVVM con *Clean Architecture*, se partirá de la aplicación de superhéroes vista anteriormente y que no cumple con dicho patrón.

Antes de comenzar a codificar, se recordarán las dependencias necesarias que deben añadirse al `build.gradle(:app)` para hacer uso de *Room*. También se añadirá una más para utilizar el ciclo de vida de *ViewModel*.

```
// Activityx, permite el uso de viewModelScope.
implementation 'androidx.activity:activity-ktx:1.7.1'

// Room
implementation 'androidx.room:room-ktx:2.5.1'
implementation 'androidx.room:room-runtime:2.5.1'
annotationProcessor 'androidx.room:room-compiler:2.5.1'
kapt 'androidx.room:room-compiler:2.5.1'
```

También se añadirá el plugin *kotlin-kapt*.

```
plugins {
    ...
    id 'kotlin-kapt'
```

```
}
```

Por último, añade en el bloque *defaultConfig* la configuración de esquemas para *Room*.

```
defaultConfig {
    ...
    javaCompileOptions {
        annotationProcessorOptions {
            arguments += [
                    "room.schemaLocation":"$projectDir/schemas".toString(),
                    "room.incremental":"true"
            ]
        }
    }
}
```

Y recuerda activar el *binding*. Con todo esto ya puedes sincronizar el *Gradle* y comenzar con el proyecto. Observa la figura adjunta, en ella puedes ver una aproximación de la organización que se aplicará al proyecto.

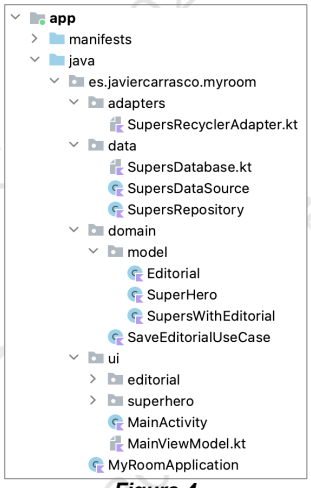

Figura 4

12.1. El modelo

Aunque el modelo, según el esquema planteado, se ha incluido en la capa dominio, suele ser uno de los primeros pasos a implementar, facilitando así el desarrollo del resto de componentes de las demás capas.

Dentro del *package* ***domain.model*** se crearán las clases necesarias, ya vistas en el capítulo anterior, pero que se recordarán a continuación.

Data Class Editorial

```
@Entity(tableName = "Editorial")
data class Editorial(
    @PrimaryKey(autoGenerate = true)
    val idEd: Int = 0,
    val name: String? = null
)
```

Data Class SuperHero

```
@Entity
data class SuperHero(
    @PrimaryKey(autoGenerate = true)
    val idSuper: Int = 0,
    val superName: String? = null,
    val realName: String? = null,
```

```
    val favorite: Int = 0,
    val idEditorial: Int = 0
)
```

Data Class SupersWithEditorial

```
data class SupersWithEditorial(
    @Embedded val supers: SuperHero,
    @Relation(
        parentColumn = "idEditorial",
        entityColumn = "idEd"
    ) val editorial: Editorial
)
```

Y ahora sí, ya se puede comenzar con la capa de datos.

12.2. Capa de datos

La capa de datos, *Data*, contendrá las dependencias de persistencia, como *Room*, *Firebase*, etc. Además, como se implementará el patrón repositorio, aquí se encontrarán las implementaciones para el repositorio y los *data sources*, las fuentes de datos.

Se comenzará por el componente más cercando al *Framework*, en esta caso, como ya se ha dicho, se hará uso de *Room*. Recuerda añadir la clase *MyRoomApplication* que extiende de *Application* para establecer la conexión con la base de datos, junto con la propiedad *name* en el *tag application* del fichero *Manifest*.

```
class MyRoomApplication : Application() {
    lateinit var supersDatabase: SupersDatabase
        private set

    override fun onCreate() {
        super.onCreate()
        supersDatabase = Room.databaseBuilder(
            this, SupersDatabase::class.java, "SuperHeros.db"
        ).build()
    }
}
```

El siguiente paso será crear el DAO, donde se establecerán las acciones que se lanzarán contra el *Framework*. Para esto, se creará el fichero `SupersDatabase.kt` en el *package* ***data***.

```
@Database(entities = [SuperHero::class, Editorial::class], version = 1)
abstract class SupersDatabase : RoomDatabase() {
    abstract fun supersDAO(): SupersDAO
}

@Dao
interface SupersDAO {
```

```
    @Transaction
    @Query("SELECT * FROM SuperHero ORDER BY superName")
    fun getSuperHerosWithEditorials(): Flow<List<SupersWithEditorial>>

    @Query("SELECT * FROM SuperHero WHERE idSuper = :idSuper")
    suspend fun getSuperById(idSuper: Int): SuperHero?

    @Query("SELECT * FROM Editorial")
    fun getAllEditorials(): Flow<List<Editorial>>

    @Query("SELECT count(idEd) FROM Editorial")
    fun getNumEditorials(): Flow<Int>

    @Query("SELECT * FROM Editorial WHERE idEd = :editorialId")
    suspend fun getEditorialById(editorialId: Int): Editorial?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertEditorial(editorial: Editorial)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertSuperHero(superHero: SuperHero)

    @Delete
    suspend fun deleteEditorial(editorial: Editorial)

    @Delete
    suspend fun deleteSuperHero(superHero: SuperHero)
}
```

Si te fijas con detenimiento en este DAO, verás que no varía mucho con respecto al desarrollado en el capítulo anterior, pero, si encontrará una diferencia notable en algunos de los tipos devueltos, concretamente el uso del tipo *Flow*.

En versiones anteriores se utilizaba el tipo *LiveData*, ***Flow***[2] es una evolución de éste. Los *Flow* son un componente que pertenece a la librería de corrutinas y permiten añadir programación reactiva. Básicamente se utilizan para recibir información en vivo, pueden devolver más de un valor y trabajan sobre corrutinas. Además, puede trabajarse con ellos de forma asíncrona.

Para el ejemplo que se está tratando, se establecerá un flujo de datos entre la fuente, que será *Room* y la representación de los datos en la UI. Como podrás ver más adelante, deberán establecerse los *"subscriptores"* sobre el flujo para obtener la información.

El siguiente componente a desarrollar será el *DataSource*, el cual se llamará `SupersDataSource` y se encontrará también dentro del *package* ***data***. Los *data sources* son una abstracción de la fuente de datos. Por lo general, se suelen crear en base a una interface, muy útil cuando se emplean diferentes fuentes de datos, por ejemplo *Room* y una API. En este ejemplo no será necesario, ya que la única fuente de datos será *Room*.

2 Flow (https://developer.android.com/kotlin/flow)

```
class SupersDataSource(private val db: SupersDAO) {
    val currentSupers: Flow<List<SupersWithEditorial>> = db.getSuperHerosWithEditorials()
    val currentEditorials: Flow<List<Editorial>> = db.getAllEditorials()
    val currentNumEditorials: Flow<Int> = db.getNumEditorials()

    suspend fun deleteSuper(superHero: SuperHero) {
        db.deleteSuperHero(superHero)
    }

   suspend fun saveSuper(superHero: SuperHero) {
        db.insertSuperHero(superHero)
   }

   suspend fun getSuperById(superId: Int): SuperHero? = db.getSuperById(superId)

   suspend fun deleteEditorial(editorial: Editorial) {
        db.deleteEditorial(editorial)
   }

   suspend fun saveEditorial(editorial: Editorial) {
        db.insertEditorial(editorial)
   }

   suspend fun getEdById(editorialId: Int): Editorial? = db.getEditorialById(editorialId)
}
```

Esta clase contiene tres propiedades *currents* en las que se recogerá la información de los superhéroes junto con su editorial, las editoriales y otra para el número de editoriales existentes. Esta última es simplemente para controlas si existen editoriales para poder añadir superhéroes.

Como puedes observar, estas propiedades son de tipo *Flow*, lo que permitirá añadir programación reactiva y, como se verá a más adelante, se mantendrá una escucha activa mediante subscripción, haciendo que sea más fácil la actualización.

El resto son funciones *suspend*, ya que se utilizarán para acceder a los datos, deben ser de este tipo ya que no puede accederse a *Room* desde el hilo principal.

Ahora deberá crearse el *repository*, en este caso observarás que prácticamente es idéntico al *data source*, la diferencia radica en que el segundo accede directamente al *framework*, podrías pensar que es innecesario, pero es aquí dónde se elegirá la fuente de datos a la que acceder (en el ejemplo que se está realizando sólo hay una), por ejemplo, cuando se tienen dos, o más, fuentes de datos y según la condición se debe acceder a una u a otra, como, cuando no hay conexión, una comprobación de caché, etc.

```
class SupersRepository(private val supersRoomDataSource: SupersDataSource) {
    val currentSupers: Flow<List<SupersWithEditorial>> = supersRoomDataSource.currentSupers
    val currentEditorials: Flow<List<Editorial>> = supersRoomDataSource.currentEditorials
    val currentNumEditorials: Flow<Int> = supersRoomDataSource.currentNumEditorials

    suspend fun deleteSuper(superHero: SuperHero) {
```

```
        supersRoomDataSource.deleteSuper(superHero)
    }

    suspend fun saveSuper(superHero: SuperHero) {
        supersRoomDataSource.saveSuper(superHero)
    }

    suspend fun getSuperById(superId: Int): SuperHero? =
        supersRoomDataSource.getSuperById(superId)

    suspend fun deleteEditorial(editorial: Editorial) {
        supersRoomDataSource.deleteEditorial(editorial)
    }

    suspend fun saveEditorial(editorial: Editorial) {
        supersRoomDataSource.saveEditorial(editorial)
    }

    suspend fun getEditorialById(editorialId: Int): Editorial? =
        supersRoomDataSource.getEdById(editorialId)
}
```

El objetivo de las propiedades *current* y los métodos sigue siendo el mismo que el planteado para el *data source*.

Con todo esto, la capa de datos para el proyecto ya estaría lista, veamos ahora como quedaría la capa dominio.

12.3. Capa dominio

De la capa dominio, el modelo ya está planteado, como se comentó anteriormente, suele ser una de las primeras tareas a resolver. En esta capa se encuentran los casos de uso, que según que autor, puede plantearse como algo opcional en el desarrollo. Para que veas ambas opciones, se creará un caso de uso para las editoriales y, para los superhéroes se trabajará directamente contra el repositorio.

En el *package* ***domain*** se creará la siguiente clase que representará el caso de uso para guardar una editorial. El nombre de estas clases suele acabar en *UseCase*, por ejemplo, `SaveEditorialUseCase`.

```
class SaveEditorialUseCase constructor(private val supersRepository: SupersRepository) {
    suspend operator fun invoke(editorial: Editorial) {
        supersRepository.saveEditorial(editorial)
    }
}
```

Particularidades, la palabra ***constructor*** especifica el constructor en línea utilizado, es opcional. En el método *suspend* ***operator*** *fun* ***invoke****()*, la palabra reservada ***operator***, junto con el nombre ***invoke***, hace que este método sea llamado directamente, así:

```
val saveEditorialUseCase = SaveEditorialUseCase(supersRepository)
```

Sin necesidad de indicar el método. Por lo general, las clases para los casos de uso únicamente contienen un método, el encargado de ejecutar el caso de uso propiamente dicho.

12.4. Capa presentación

Esta capa será la encargada de mostrar la información al usuario. Es aquí dónde se aplicará el uso de *ViewModel* para la recuperación de información y se buscará la máxima independencia posible entre interface y lógica.

Todas las clases que intervienen en esta capa se encuentran dentro del *package* ***ui***, buscando así una mayor organización del proyecto.

12.4.1. UI Editorial

En primer lugar se creará el *ViewModel* para la actividad editoriales, como se indicó en el punto anterior, es aquí dónde se utilizará el caso de uso creado. La clase en cuestión será `EditorialViewModel`.

```
class EditorialViewModel(
    private val saveEditorialUseCase: SaveEditorialUseCase
) : ViewModel() {

    private val _state = MutableStateFlow(Editorial())

    fun save(name: String) {
        viewModelScope.launch {
            val editorial = _state.value.copy(name = name)
            saveEditorialUseCase(editorial)
        }
    }
}
```

En primer lugar se crea la propiedad ***_state***, a la cual se le asigna una editorial vacía mediante *MutableStateFlow*, esto se utilizará más adelante con los superhéroes, permitiendo comprobar si se está creando uno nuevo o se está actualizando uno ya existente.

El método ***save*** se encargará de guardar la editorial en la base de datos, fíjate que se crea una copia del objeto modifican únicamente la propiedad deseada, de esta forma se hace uso de `@Insert(onConflict = OnConflictStrategy.REPLACE)` utilizado en *SupersDatabase*, así no es necesario crear un método para actualizar. Por último, se llama al caso de uso pasándole la editorial. Con *ViewModelScope* se lanza la corrutina dentro del *ViewModel* separándolo de la UI.

Ahora bien, este tipo de *ViewModels* que requieren parámetros, necesitan de un *ViewModelFactory* para poder establecer esos parámetros. Un *ViewModel* sin parámetros no necesita de un *Factory* para poder trabajar. En la misma clase, se puede añadir a continuación.

```
@Suppress("UNCHECKED_CAST")
class EditorialViewModelFactory(
    private val saveEditorialUseCase: SaveEditorialUseCase
) : ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return EditorialViewModel(saveEditorialUseCase) as T
    }
}
```

Ahora, cuando se instancie el *ViewModel* en la clase de la actividad, se hará a través del *Factory*. Observa como quedará ahora la clase de la actividad de editoriales.

```
class EditorialActivity : AppCompatActivity() {
    private lateinit var binding: ActivityEditorialBinding

    private val vm: EditorialViewModel by viewModels {
        val db = (application as MyRoomApplication).supersDatabase
        val supersDataSource = SupersDataSource(db.supersDAO())
        val supersRepository = SupersRepository(supersDataSource)
        val saveEditorialUseCase = SaveEditorialUseCase(supersRepository)
        EditorialViewModelFactory(saveEditorialUseCase)
    }

    companion object {
        fun navigate(activity: AppCompatActivity) {
            activity.startActivity(
                Intent(
                    activity,
                    EditorialActivity::class.java
                ).addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP),
                ActivityOptions.makeSceneTransitionAnimation(activity).toBundle()
            )
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityEditorialBinding.inflate(layoutInflater)
        setContentView(binding.root)

        supportActionBar!!.title = getString(R.string.txt_editorial)

        binding.button.setOnClickListener {
            if (binding.etEditorial.text.isNullOrBlank())
                binding.labelEtEditorial.error = getString(R.string.warning_empty_field)
            else {
                binding.labelEtEditorial.error = null
                vm.save(binding.etEditorial.text!!.trim().toString())
                finishAfterTransition()
```

```
            }
        }
    }
}
```

Esta clase es la de menor dificultad, únicamente se utiliza el *ViewModel* para guardar en la base de datos, pero ya no es necesario crear corrutinas en ella, ya se encarga el *ViewModel*.

12.4.2. UI Superhero

Esta ya contiene algo más de complejidad, además de comprobar si se está creando o editando un superhéroe, deberá cargarse una lista de editoriales en el *Spinner*. Para esta actividad no se ha creado un caso de uso, de esta forma puedes ver las dos opciones.

En primer lugar observa como quedará la clase *ViewModel* asociada a la actividad, `SuperheroViewModel`.

```
class SuperheroViewModel(
    private val supersRepository: SupersRepository, private val superId: Int
) : ViewModel() {

    private val _stateSuper = MutableStateFlow(SuperHero(0))
    val stateSuper = _stateSuper.asStateFlow()

    // Recoge las editoriales disponibles.
    val stateEd = supersRepository.currentEditorials

    init {  // Se intenta recuperar el superhéroe que se pasa por id.
        viewModelScope.launch {
            val superhero = supersRepository.getSuperById(superId)
            if (superhero != null)
                _stateSuper.value = superhero
        }
    }

    fun save(superhero: SuperHero) {
        viewModelScope.launch {
            val superheroAux = _stateSuper.value.copy(
                superName = superhero.superName,
                realName = superhero.realName,
                favorite = superhero.favorite,
                idEditorial = superhero.idEditorial
            )
            supersRepository.saveSuper(superheroAux)
        }
    }
}
```

En primer lugar, se crea en la variable ***_stateSuper*** un objeto *SuperHero* vacío con identificador a cero, se utilizará para saber si se está consultando o creando.

La variable ***stateEd*** recogerá todas las editoriales disponibles de la base de datos, recuerda el ciclo, desde aquí se llamará a la variable *currentEditorials* del repositorio que a su vez llamará a la variable *currentEditorials* del *DataSource*, que llamará al método *getAllEditorials()*.

En el constructor inicial se comprueba si se ha pasado un valor al identificador (*superId*), de tal forma que si es distinto de 0, si el identificador existe, no será nulo, por lo que se estará ante una consulta, se está viendo un superhéroe que sí existe en la base de datos.

Por último, el método ***save***, que será el encargado tanto de guardar un superhéroe nuevo, como de actualizar uno ya existente.

Además, como este *ViewModel* requiere parámetros, deberá crearse un *ViewModelFactory* parecido al visto anteriormente.

```
@Suppress("UNCHECKED_CAST")
class SuperheroViewModelFactory(
    private val supersRepository: SupersRepository, private val superId: Int
) : ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return SuperheroViewModel(supersRepository, superId) as T
    }
}
```

Ahora, en la clase `SuperheroActivity` deberá instanciarse el *ViewModel* y crear la lógica de la actividad, entre ella, el código necesario para el *Spinner*.

```
class SuperheroActivity : AppCompatActivity() {
    private lateinit var binding: ActivitySuperheroBinding
    private lateinit var adapter: ArrayAdapter<String>
    private var editorialsList: List<Editorial> = emptyList()
    private var editorialsArray = ArrayList<String>()
    private var editorialId = -1 // Versión ArrayAdapter para el Spinner.

    private val vm: SuperheroViewModel by viewModels {
        val db = (application as MyRoomApplication).supersDatabase
        val supersDataSource = SupersDataSource(db.supersDAO())
        val supersRepository = SupersRepository(supersDataSource)
        val superId = intent.getIntExtra(EXTRA_SUPER_ID, 0)
        SuperheroViewModelFactory(supersRepository, superId)
    }

    companion object {
        const val EXTRA_SUPER_ID = "superId"
        fun navigate(activity: AppCompatActivity, superId: Int = -1) {
            activity.startActivity(
                Intent(activity, SuperheroActivity::class.java)
                    .addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP).apply {
                        putExtra(EXTRA_SUPER_ID, superId)
                    }, ActivityOptions.makeSceneTransitionAnimation(activity).toBundle()
```

```
            )
        }
    }
…
}
```

Este primer bloque muestra las propiedades necesarias para esta clase, la instanciación del *ViewModel* y el *companion object* con el método necesario para abrir la actividad, dónde se le pasará el identificador del superhéroe.

A continuación, en el método *onCreate*, se creará el *ArrayAdapter* para el *Spinner* y, haciendo uso de corrutinas y el ciclo de vida, se recogerán los datos para el *Spinner* y los datos del superhéroe si los hubiese.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivitySuperheroBinding.inflate(layoutInflater)
    setContentView(binding.root)
    supportActionBar!!.title = getString(R.string.txt_superhero)

    // Se crea el adaptador mediante ArrayAdapter.
    adapter = ArrayAdapter(
        this@SuperheroActivity,
        android.R.layout.simple_list_item_activated_1,
        editorialsArray
    )

    // Se carga el adaptador en el Spinner.
    binding.spinner.adapter = adapter

    lifecycleScope.launch { this: CoroutineScope

        repeatOnLifecycle(Lifecycle.State.STARTED) { this: CoroutineScope
            vm.stateEd.collect { it: List<Editorial>
                editorialsList = it
                editorialsArray.clear()

                it.forEach {
                    editorialsArray.add(it.name!!)
                }
                adapter.notifyDataSetChanged()

                // Se recoge el Superhéroe si existe.
                vm.stateSuper.collect { superCollect -> superCollect: SuperHero
                    binding.etSuperName.setText(superCollect.superName)
                    binding.etRealName.setText(superCollect.realName)

                    if (superCollect.idSuper != 0) {
                        binding.spinner.setSelection( // Selección del Spinner.
                            editorialsList.withIndex().first {
```

```
                                it.value.idEd == superCollect.idEditorial
                            }.index
                        )
                    }
                    binding.switchFab.isChecked = superCollect.favorite == 1
                }
            }
        }
    }
}
```

Detalles importantes, se utiliza ***lifecycleScope*** para evitar problemas con la pérdida de la suscripción, existen otros métodos, pero no garantizan la suscripción a los elementos recogidos.

Con ***repeatOnLifecycle(Lifecycle.State.STARTED)*** se indicará en que estado comienza la recolección de datos, viene a continuación. El opuesto de *STARTED* será *ON_STOP*. Muy importante, cuando se utiliza *repeatOnLifecycle* + *collect*, suspende al finalizar, por lo que todo lo que quede debajo de la llave de cierre del *repeatOnLifecycle* no se ejecutará.

Los métodos ***collect*** son los encargados de recoger la información que se obtiene a través del *ViewModel*, y se tratarán en consecuencia.

En el método *onStart* se comprueban los campos para guardar y la selección de la editorial en el *Spinner*.

```
override fun onStart() {
    super.onStart()

    binding.button.setOnClickListener {
        if (binding.etSuperName.text.isNullOrBlank())
            binding.labelEtSuperName.error = getString(R.string.warning_empty_field)
        else if (binding.etRealName.text.isNullOrBlank())
            binding.labelEtRealName.error = getString(R.string.warning_empty_field)
        else {
            binding.labelEtSuperName.error = null
            val supername = binding.etSuperName.text!!.trim().toString()
            val realname = binding.etRealName.text!!.trim().toString()
            val fab = if (binding.switchFab.isChecked) 1 else 0

            vm.save(
                SuperHero(
                    superName = supername,
                    realName = realname,
                    favorite = fab,
                    idEditorial = editorialId
                )
            )
            finishAfterTransition()
        }
    }
```

```
    binding.spinner.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
        override fun onItemSelected(
            adapterView: AdapterView<*>?,
            view: View?,
            pos: Int,
            id: Long
        ) {
           editorialId = editorialsList[pos].idEd
           Log.d("Spinner", "${editorialsList[pos].idEd} - ${editorialsList[pos].name}")
        }

        override fun onNothingSelected(p0: AdapterView<*>?) {}
    }
}
```

12.4.3. UI MainActivity

La actividad principal se encargará de mostrar el listado de superhéroes en un *RecyclerView* y dará acceso a diferentes acciones. El *ViewModel* asociado a esta actividad se encargará de eliminar un superhéroe, añadir (para deshacer la operación de borrado) y el cambio de estado de favorito. Además de dos propiedades, ***state*** para obtener una lista de superhéroes, y ***stateNumEd***, que contiene el número de editoriales. Esta última se utilizará para saber si ya existen editoriales añadidas, si no hubiesen, no se permitirá añadir un superhéroe.

```
class MainViewModel(private val supersRepository: SupersRepository) : ViewModel() {
    val state = supersRepository.currentSupers
    var stateNumEd = supersRepository.currentNumEditorials

    fun onSuperDelete(superHero: SuperHero) {
        viewModelScope.launch {
            supersRepository.deleteSuper(superHero)
        }
    }

    fun onSuperInsert(superHero: SuperHero) {
        viewModelScope.launch {
            supersRepository.saveSuper(superHero)
        }
    }

    fun onFabSuper(superHero: SuperHero) {
        viewModelScope.launch {
            val superheroAux = superHero.copy(
                favorite = if (superHero.favorite == 0) 1 else 0
            )
            supersRepository.saveSuper(superheroAux)
        }
    }
}
```

Como con los anteriores *ViewModels*, será necesario crear un *factory* para poder pasarle el repositorio como parámetro.

```
@Suppress("UNCHECKED_CAST")
class MainViewModelFactory(private val supersRepository: SupersRepository) :
    ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return MainViewModel(supersRepository) as T
    }
}
```

Ya en la `MainActivity`, se declararán las siguientes propiedades, entre ellas, la asociación con el *ViewModel*.

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    private lateinit var adapter: SupersRecyclerAdapter
    private var numEditorials = 0

    private val vm: MainViewModel by viewModels {
        val db = (application as MyRoomApplication).supersDatabase
        val supersDataSource = SupersDataSource(db.supersDAO())
        val supersRepository = SupersRepository(supersDataSource)
        MainViewModelFactory(supersRepository)
    }
    ...
}
```

En el método *onCreate* de la actividad se configura el adaptador para el *RecyclerView* y la recolección de los datos. Para la recolección se utilizará el mismo sistema que el empleado en la *UI Superhero*.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    adapter = SupersRecyclerAdapter(
        onSuperHeroClick = { it: SuperHero
            SuperheroActivity.navigate(this@MainActivity, it.idSuper)
        },
        onSuperHeroLongClick = {
            vm.onSuperDelete(it)
            Snackbar.make(
                binding.root,
                getString(R.string.warning_delete, it.superName),
                Snackbar.LENGTH_LONG
            ).setAction(R.string.warning_undo) { v ->
                vm.onSuperInsert(it)
            }.show()
```

```
        },
        onFabClick = { vm.onFabSuper(it) }
    )
    binding.recycler.adapter = adapter

    lifecycleScope.launch {
        repeatOnLifecycle(Lifecycle.State.STARTED) {
            vm.state.collect {
                adapter.submitList(it)
            }
        }
    }
}
```

En el método *onStart* se obtendrá el número de editoriales y se actualizará la variable *numEditorials*, esta permitirá controlar si se pueden añadir superhéroes o no.

```
override fun onStart() {
    super.onStart()

    lifecycleScope.launch {
        vm.stateNumEd.collect {
            numEditorials = it
        }
    }
}
```

Por último, se inflará el menú y se hará el control de la opción pulsada, dónde se comprobará si existen editoriales para añadir un superhéroe.

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.main_menu, menu)
    return true
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.opAddEditorial -> {
            EditorialActivity.navigate(this@MainActivity)
            true
        }
        R.id.opAddSuper -> {
            if (numEditorials > 0)
                SuperheroActivity.navigate(this@MainActivity)

            true
        }
        else -> false
    }
}
```

Con todas estas modificaciones, ya se dispondría de la aplicación de superhéroes adaptada al patrón MVVM con *Clean Architecture*.

Es posible que pienses que ahora el proyecto tiene más clases creadas, o mayor complejidad, pero si observas las clases relacionadas con la UI, verás que se ha producido un desacoplamiento, lo que permitiría, entre otras cosas, cambiar la fuente de datos sin necesidad de tocar la parte visual.