Programación Multimedia y Dispositivos Móviles

UD 11. Bases de datos

Javier Carrasco

Curso 2024 / 2025

Bases de datos

11. Bases de datos

En este capítulo se introducirá el almacenamiento local en el dispositivo mediante bases de datos. El nombre de la base de datos que se utilizará en **Android** es **SQLite**[1], ésta es una base de datos de código abierto, *Open Source*, que almacena la información en un fichero en el mismo dispositivo.

SQLite es un motor de base de datos transaccional, ligero y autónomo, de fácil configuración y sin la necesidad de utilizar un servidor. Dejando a un lado todas las características de **SQLite**, en este capítulo se introducirá su uso en aplicaciones Android. En primer lugar se verá como trabajar directamente sobre SQLite, para más adelante introducir **Room**, que es una biblioteca de persistencia que proporciona una capa de abstracción sobre SQLite.

Para ambos ejemplos se planteará el siguiente esquema, una base de datos que se llamará *"SuperHeros.db"* que contendrá dos tablas, una llamada *"Supers"* que contendrá información básica como el nombre del superhéroe, su nombre real y si es favorito o no. La segunda tabla se llamará *"Editorials"*, que contendrá el nombre de las editoriales, entre ambas tablas existirá una relación de uno a muchos, una editorial puede contener muchos superhéroes, pero un superhéroe únicamente podrá pertenecer a una editorial.

Las aplicaciones básicamente contendrán un formulario para añadir superhéroes y otro para añadir las editoriales. La pantalla principal mostrará un listado con la información añadida que, además, dará acceso al detalle del elemento pulsado. El diseño se dejará de lado en este punto.

11.1. SQLiteOpenHelper

Para este primer ejemplo se utilizarán las siguientes data class para utilizarlas como modelo.

```
// Modelo para las editoriales.
data class Editorial(val id: Int, val name: String)

// Modelo para los superhéroes.
data class SuperHero(
    val id: Int,
    val superName: String,
    val realName: String,
    val favorite: Int,
    val editorial: Int
)
```

La clase más importante que tratará este punto ayuda con la creación y control de versión de la base de datos de forma manual. Cuando se crea una clase que implementa `SQLiteOpenHelper` deberán sobrecargarse los métodos `onCreate()`, `onUpgrade()` y de manera opcional el método `onOpen()`. Por ejemplo, se puede crear la clase `MyDBOpenHelper()` que implemente `SQLiteOpenHelper` de la siguiente forma.

1 SQLite (https://developer.android.com/reference/android/database/sqlite/package-summary)

```
class SupersDBHelper(context: Context, factory: SQLiteDatabase.CursorFactory?) :
    SQLiteOpenHelper(context, DATABASE_NAME, factory, DATABASE_VER) {
    private val TAG = "SQLite"

    companion object {
        val DATABASE_VER = 1
        val DATABASE_NAME = "SuperHeros.db"

        val TABLE_EDITORIAL = "Editorials"
        val COLUMN_ID = "_id" // valido para las dos tablas
        val COLUMN_EDITORIAL = "editorial"

        val TABLE_SUPER = "Supers"
        val COLUMN_SUPERNAME = "supername"
        val COLUMN_REALNAME = "realname"
        val COLUMN_FAV = "favorite"
        val COLUMN_ID_ED = "idEd"
    }

    // Este método es llamado cuando se crea la base por primera vez. Debe producirse
    // la creación de todas las tablas que formen la base de datos.
    override fun onCreate(db: SQLiteDatabase?) {
        try {
            var query = "CREATE TABLE $TABLE_EDITORIAL " +
                    "($COLUMN_ID INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    "$COLUMN_EDITORIAL TEXT)"
            db!!.execSQL(query)

            query = "CREATE TABLE $TABLE_SUPER " +
                    "($COLUMN_ID INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    "$COLUMN_SUPERNAME TEXT," +
                    "$COLUMN_REALNAME TEXT," +
                    "$COLUMN_FAV INTEGER," +
                    "$COLUMN_ID_ED INTEGER," +
                    "FOREIGN KEY($COLUMN_ID_ED) REFERENCES $TABLE_EDITORIAL)"
            db.execSQL(query)
        } catch (e: SQLiteException) {
            Log.e("$TAG onCreate", e.message.toString())
        }
    }

    // Este método se invocará cuando la base de datos necesite ser actualizada. Se
    // utiliza para hacer DROPs, añadir tablas o cualquier acción que actualice la BD.
    override fun onUpgrade(db: SQLiteDatabase?, p1: Int, p2: Int) {
        try {
            var query = "DROP TABLE IF EXISTS $TABLE_SUPER"
            db!!.execSQL(query)

            query = "DROP TABLE IF EXISTS $TABLE_EDITORIAL"
            db.execSQL(query)
```

```
            onCreate(db)
        } catch (e: SQLiteException) {
            Log.e("$TAG onUpgrade", e.message.toString())
        }
    }

    // Método opcional. Se llamará a este método después de abrir la base de datos,
    // antes de ello, comprobará si está en modo lectura. Se llama justo después de
    // establecer la conexión y crear el esquema.
    override fun onOpen(db: SQLiteDatabase?) {
        super.onOpen(db)
        Log.d("$TAG onOpen", "¡¡Base de datos abierta!!")
    }
}
```

Debes saber para que se utiliza el parámetro que indica la versión, en el ejemplo aparece representado mediante la constante `DATABASE_VERSION`. Empezará con valor a 1, cuando se incremente el número (2, 3, 4, etc), se ejecutará el método `onUpgrade()` para actualizar, pero si se indica un valor inferior, 1 es el mínimo, entonces se ejecutará el método `onDowngrade()`, para volver a una versión anterior.

El implementar la clase `SQLiteOpenHelper` en la clase creada, permitirá hacer uso en esta nueva clase las operaciones **CRUD** que se necesiten, muy recomendable en la propia clase para poder separar las vistas del controlador.

Fíjate que también se han creado una serie de constantes para identificar las columnas de las tablas, el nombre de las tablas y de la propia base de datos, que pueden causar problemas si se escriben incorrectamente.

11.1.1. Establecer conexión con la base de datos

Una vez creada la base de datos, es necesario establecer la conexión para poder acceder a ella. Para evitar estar creando conexiones según vaya haciendo falta, puedes optar por la siguiente estrategia. Se creará una nueva clase que extienda de la clase *Application* con el siguiente contenido.

```
class MySQLiteApplication : Application() {
    lateinit var supersDBHelper: SupersDBHelper
        private set

    override fun onCreate() {
        super.onCreate()
        supersDBHelper = SupersDBHelper(this, null)
    }
}
```

El segundo parámetro, `factory`, se establece a `null`, ya que se busca el comportamiento por defecto (según documentación de Google), es decir, para indicar que se sabe trabajar con bases de datos.

Esta clase se ejecutará la primera, lo que permitirá establecer el objeto *supersDBHelper*, siendo accesible desde cualquier punto de la aplicación. Para que funcione, será necesario indicar en el *Manifest* la existencia de esta clase.

```
<application
    android:name=".MySQLiteApplication"
    ...
```

Si te fijas en la clase, el objeto *supersDBHelper* es público, pero su *setter* es privado, por lo que solo se instanciará en el *onCreate* de esta clase.

Para hacer uso del objeto que apunta a la base de datos, deberás utilizar la siguiente línea, para añadir una editorial, por ejemplo.

```
(application as MySQLiteApplication).supersDBHelper.addEditorial(name)
```

11.1.2. Insertar

El método para añadir una editorial a la tabla *"editorials"* podría quedar de la siguiente manera dentro de la propia clase `SupersDBHelper`.

```
// Se añade una editorial.
fun addEditorial(name: String): Long {
    // Se crea un ArrayMap<>() haciendo uso de ContentValues().
    val data = ContentValues().apply {
        put(COLUMN_EDITORIAL, name)
    }

    // Se abre la BD en modo escritura.
    val db = this.writableDatabase

    // Devuelve el id del registro insertado, -1 en caso de error.
    val result = db.insert(TABLE_EDITORIAL, null, data)

    db.close()
    return result
}
```

Para añadir un superhéroe, no cambiaría mucho, la diferencia está principalmente en los datos que deben pasarse para insertar, se podría hacer uso de la *data class* creada.

```
fun addSuperHero(superHero: SuperHero): Long {
    val data = ContentValues().apply {
        put(COLUMN_SUPERNAME, superHero.superName)
        put(COLUMN_REALNAME, superHero.realName)
        put(COLUMN_FAV, superHero.favorite)
        put(COLUMN_ID_ED, superHero.editorial)
    }
    val db = this.writableDatabase
    val result = db.insert(TABLE_SUPER, null, data)
```

```
    db.close()
    return result
}
```

Observa que se omite el identificador, ya que será asignado de manera automática por la base de datos.

11.1.3. Eliminar

```
// Método para eliminar una editorial de la tabla por el identificador.
fun delEditorial(identifier: Int): Int {
    val args = arrayOf(identifier.toString())

    // Se abre la BD en modo escritura.
    val db = this.writableDatabase

    // Se puede elegir un sistema u otro, teniendo en cuenta que execSQL no devuelve nada.
    val result = db.delete(TABLE_EDITORIAL, "$COLUMN_ID = ?", args)
    // db.execSQL("DELETE FROM TABLE_EDITORIAL WHERE COLUMN_ID = ?", args)

    db.close()
    return result
}
```

En este caso, el método *delete* devolverá el número de filas afectadas que pasen el filtro de la cláusula *where*.

11.1.4. Actualizar

```
// Método para actualizar el nombre de una editorial de la tabla por el id.
fun updateEditorial(idEditorial: Int, newName: String) {
    val args = arrayOf(idEditorial.toString())
    val data = ContentValues()
    data.put(COLUMN_EDITORIAL, newName)

    val db = this.writableDatabase
    db.update(TABLE_EDITORIAL, data, "$COLUMN_ID = ?", args)

    db.close()
}
```

Si no te gusta utilizar los métodos `insert()`, `delete()` y `update()`, puedes hacer uso del método `execSQL()`, este método necesitará de la instrucción SQL[2] necesaria para la acción que se desee realizar, pero deberás tener en cuenta que este método no devuelve información acerca de las filas afectadas por la instrucción.

Figura 1

2 SQLite (https://www.sqlite.org/lang.html)

Fíjate ahora como quedaría la acción de guardar una editorial desde el formulario diseñado para ello, en este caso, desde el botón guardar del formulario.

```
binding.button.setOnClickListener {
    if (binding.etEditorial.text.isNullOrBlank())
        binding.labelEtEditorial.error = getString(R.string.warning_empty_field)
    else {
        binding.labelEtEditorial.error = null

        val name = binding.etEditorial.text!!.trim().toString()
        (application as MySQLiteApplication).supersDBHelper.addEditorial(name)

        finish()
    }
}
```

Es posible utilizar un visor para comprobar en que estado se encuentra base de datos, para ello, una vez descargada desde el emulador (*Device File Explorer*), se puede utilizar, por ejemplo **DB Browser for SQLite**[3] para ver su contenido. Pero como se detallará más adelante, puedes hacer uso del *App Inspection* que incorpora Android Studio.

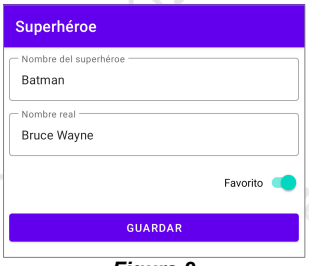

Figura 2

Siguiendo los pasos vistos para añadir una editorial, se seguirán los mismos para añadir un superhéroe a la base datos.

```
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

        (application as MySQLiteApplication).supersDBHelper.addSuperHero(
            SuperHero(0, supername, realname, fab, 0)
        )

        finish()
    }
}
```

Observa que al crear el objeto *SuperHero*, tanto el identificador como el id de la editorial se ponen a cero. El primero es porque será la base de datos quien se lo asigne, el segundo es porque todavía no se ha añadido un sistema para establecerlo.

3 *DB Browser for SQLite* (https://sqlitebrowser.org/)

11.1.5. Leer

Para poder mostrar la información almacenada en la base de datos, desde la clase *helper* creada se hará uso de la clase `Cursor()`, ésta permite recoger la información devuelta por la ejecución de la sentencia SQL mediante el método `rawQuery()`, permitiendo el acceso de manera aleatorio a los resultados devueltos.

Los métodos que se utilizarán para desplazarse por el *cursor* en este ejemplo son, `moveToFirst()` para desplazarse al inicio y `moveToNext()` para ir avanzando. Devolverán *false* si no hay entrada de datos. También se dispone de `moveToLast()`, `moveToPosition(pos)` y `moveToPrevious()` para desplazarse.

Se recomienda que, tras obtener la información de la base de datos, devolverla en el formato que mejor se adapte a la vista que vaya a utilizarse para mostrar.

SimpleCursorAdapter

En ocasiones, por el motivo que sea, no es necesario o no se quiere crear un adaptador personalizado, para ello se dispone de la clase `SimpleCursorAdapter`[4], que permite pasar directamente un `Cursor` a un adaptador sencillo, ya definido, y que se puede utilizar para salir del paso sin necesidad de crear nuevas clases.

El siguiente ejemplo muestra como montar un `SimpleCursorAdapter` y asignarlo a un `Spinner`, así como capturar el elemento seleccionado. En el formulario para añadir superhéroes se añadirá la nueva vista.

```
...
<Spinner
    android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    ...
    android:spinnerMode="dropdown" />
```

El método encargado de recoger la información de la base de datos en la clase *helper* será como se muestra a continuación, para este caso, se devuelve el *cursor* completo.

```
// Se obtienen todas las editoriales.
fun getEditorials(): Cursor {
    val db = this.readableDatabase

    return db.rawQuery("SELECT * FROM $TABLE_EDITORIAL;", null)
}
```

A continuación, en el método `onCreate()` de la clase se obtiene la información que se quiere mostrar en el desplegable *(Spinner)* y se monta con el *SimpleCursorAdapter*.

4 *SimpleCursorAdapter* (https://developer.android.com/reference/kotlin/android/widget/SimpleCursorAdapter)

```
val cursor = (application as MySQLiteApplication).supersDBHelper.getEditorials()

// Se crea el adaptador mediante SimpleCursorAdapter.
val adapter = SimpleCursorAdapter(
    this,
    android.R.layout.simple_list_item_2,
    cursor,
    arrayOf(cursor.columnNames[0], cursor.columnNames[1]),
    intArrayOf(android.R.id.text1, android.R.id.text2),
    SimpleCursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER
)

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

Como puedes observar, la obtención del `Cursor` y la asignación del adaptador a un elemento es algo ya visto, y no cambia para la creación de este tipo de adaptadores. Simplemente destacar el uso de `SimpleCursorAdapter` y sus parámetros:

```
SimpleCursorAdapter(context: Context!, layout: Int, c: Cursor!,
                    from: Array<String!>!, to: IntArray!, flags: Int)
```

- ***context***: contexto en el que se mostrará el listado asociado al *adapter*, en este ejemplo será donde se muestre el *Spinner*.
- ***layout***: identificador del recurso que representará los elementos, en este caso, se utiliza uno predefinido. Hay varios tipos, puedes probar y elegir el que más te guste.
- ***c***: cursor de la base de datos que contiene la información a mostrar.
- ***from***: es un *array* con la lista de los campos (columnas) que contienen la información a inyectar.
- ***to***: será el array de los identificadores de las vistas que contiene el *layout* donde se inyectarán lo campos indicados en *from*. Al utilizarse un *layout* predefinido, los nombres de las vistas son *text1*, *text2*, etc.
- ***flags***: determinará el comportamiento del adaptador, se suele utilizar FLAG_REGISTER_CONTENT_OBSERVER para crear un observador que registre y notifique cada cambio.

Ya solo queda recoger la selección del usuario, para ello, se utilizará el método `onItemSelectedListener` al cual se le asignará un objeto `AdapterView.OnItemSelectedListener`.

```
var cursorPos: Cursor? = null

binding.spinner.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
    override fun onItemSelected(
        adapterView: AdapterView<*>?,
```

```
            view: View?,
            pos: Int,
            id: Long
        ) {
            cursorPos = binding.spinner.getItemAtPosition(pos) as Cursor
            Log.d("Spinner", "${cursorPos!!.getString(0)} - ${cursorPos!!.getString(1)}")
        }

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

La sobrecarga del método `onItemSelected()` es la encargada de recoger la posición seleccionada, la variable `cursorPos` contiene la posición del cursor, a partir de la cual ya se puede acceder a su información. Para este ejemplo, el método `onNothingSelected()` no se ha sobrecargado, pero debe estar preparado.

Una vez obtenida toda la información, la llamada para guardar el registro podría quedar como se muestra en esta línea. Aunque se inicialice a *null* la variable *cursorPos*, al crear el *listener* del *Spinner* se asignará el primer valor siempre, por lo que nunca llegará a ser *null*.

```
(application as MySQLiteApplication).supersDBHelper.addSuperHero(
    SuperHero(0, supername, realname, fab, cursorPos!!.getInt(0))
)
```

El resultado de ejecutar la consulta debería ser algo parecido a lo que se muestra en la imagen de la figura.

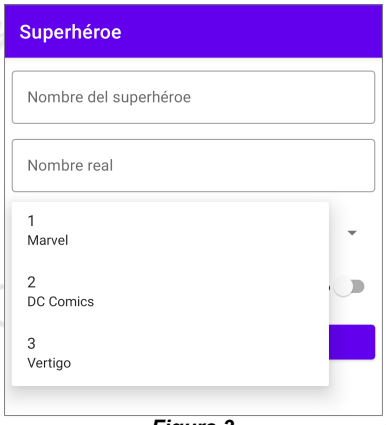

Figura 3

ArrayAdapter

Si quieres utilizar un `AutoCompleteTextView`, por ejemplo, deberás transformar el *cursor* en un *array*, y que la fuente de datos para este tipo de elementos suele de tipo *array*.

El método *getEditorials* quedaría de la siguiente forma para devolver directamente la información en el formato que interesa.

```
// Se obtiene el nombre de las editoriales.
fun getEditorials(): ArrayList<String> {
    val db = this.readableDatabase
    val cursor = db.rawQuery("SELECT * FROM $TABLE_EDITORIAL;", null)
    val datos = ArrayList<String>()

    cursor.moveToFirst()
    do {
        datos.add(cursor.getString(1))
        cursor.moveToNext()
    } while (!cursor.isAfterLast) // Se convierte el Cursor en un ArrayList.

    cursor.close()
    return datos
}
```

Observa que este caso, al no necesitarse más el cursor, se puede cerrar y se devuelve la información en una *ArrayList*. Ahora, ya en la clase de la actividad que contenga el AutoCompleteTextView se crea el adaptador y se asigna a la vista.

```
val datos = (application as MySQLiteApplication).supersDBHelper.getEditorials()

ArrayAdapter(this, android.R.layout.simple_list_item_1, datos).also {
    binding.autoTextView.setAdapter(it)
}
```

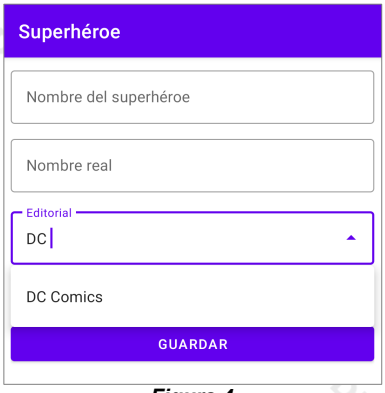

Figura 4

CursorAdapter

Si quieres rellenar un `ListView` con datos de una base de datos, puede hacerse de dos formas. Para un *ListView* simple, puedes usar *SimpleCursorAdapter*, pero si quieres personalizarlo, no se puede utilizar un *cursor* directamente, para ello se hará uso de la clase `CursorAdapter`.

Para el primer caso, el método encargado de obtener todos los superhéroes podrías ser como se muestra a continuación.

```
// Muestra todos los Supers.
fun getAllSuperHeros(): Cursor {
    val db = this.readableDatabase
    return db.rawQuery(
        "SELECT * FROM $TABLE_SUPER INNER JOIN $TABLE_EDITORIAL ON " +
                "$TABLE_SUPER.$COLUMN_ID_ED = $TABLE_EDITORIAL.$COLUMN_ID " +
                "ORDER BY $COLUMN_SUPERNAME;", null
    )
}
```

El siguiente código muestra como poblar un *ListView* simple que se encuentra en un *fragment*.

```
class ListviewFragment(private val db: SupersDBHelper) : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        val cursor = db.getAllSuperHerosCursor()
        val adapter = SimpleCursorAdapter(
            requireContext(),
            android.R.layout.simple_list_item_2,
            cursor,
            arrayOf(cursor.columnNames[1], cursor.columnNames[2]),
            intArrayOf(android.R.id.text1, android.R.id.text2),
            SimpleCursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER
        )
        binding.listView.adapter = adapter
    }
    ...
}
```

Este sería el caso sencillo. Para personalizar un *ListView*, como se ha dicho, se utilizará `CursorAdapter`, que es una clase abstracta mediante la cual se podrá crear un adaptador personalizado para la lista. Si con *ArrayAdapter* se debía sobrecargar el método `getView()` para *"inflar"* la lista, con `CursorAdapter` se deberán sobrescribir los métodos `bindView()` y `newView()`.

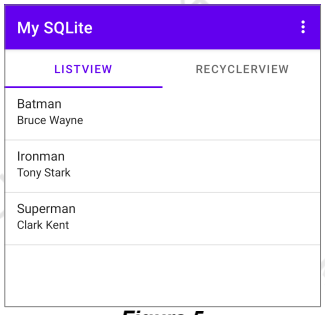

Figura 5

Aunque se mostrará como se hace, no se recomienda utilizar este sistema si tu listado va a permitir cambios, como eliminar elementos o realizar actualizaciones continuas.

`bindView()` se encargará de rellenar la lista con los datos pasados en el *cursor* y, `newView()` *"inflará"* cada elemento (*view*) de la lista. Cuando se implementan estos métodos, no hay que preocuparse por iterar el cursor, esto se hace internamente.

Como se trata de personalizar la vista de los ítems, una vez se haya creado el *layout* personalizado, se creará el *adapter* para el *ListView*, este adaptador extenderá de la clase *CursorAdapter* como ya se ha comentado. El método *getAllSuperHeros* quedará como se ha visto en su última versión.

La creación y asignación del *adapter* al *ListView* puede quedar como se muestra en estas líneas.

```
val cursor = db.getAllSuperHerosCursor()
val adapter = SupersCursorAdapter(requireContext(), cursor)
binding.listView.adapter = adapter
```

La clase `SupersCursorAdapter` puede ser como se muestra a continuación.

```
class SupersCursorAdapter(context: Context, cursor: Cursor) :
    CursorAdapter(context, cursor, FLAG_REGISTER_CONTENT_OBSERVER) {

    // "Infla" cada uno de los elementos de la lista.
    override fun newView(
        context: Context?,
        cursor: Cursor?,
        parent: ViewGroup?
    ): View = ItemListviewBinding.inflate(LayoutInflater.from(context), parent, false).root

    // Rellena el ListView.
    override fun bindView(view: View?, context: Context?, cursor: Cursor?) {
        val bindingItems = ItemListviewBinding.bind(view!!)

        with(bindingItems) {
            tvSuperName.text = cursor!!.getString(1)
            tvRealName.text = cursor.getString(2)
            tvEditorial.text = cursor.getString(6)

            view.setOnClickListener {
                Toast.makeText(context, "${tvSuperName.text}", Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

El resultado a obtener será una lista personalizada con más información de cada elemento a mostrar.

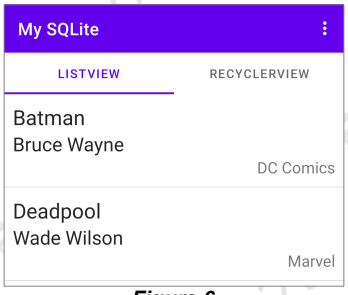

Figura 6

11.1.6. Poblar un RecyclerView con ListAdapter

Si se pretende mostrar la información de la base de datos en un *RecyclerView* no se puede hacer uso de la clase `CursorAdapter`, deberá utilizarse la clase `Cursor` directamente. Deberá prestarse especial atención a los métodos `onBindViewHolder()`, `onCreateViewHolder()` y `getCount()` al implementar el adaptador para el `RecyclerView`. Pero, como se ha comentado con anterioridad, lo ideal es pasar esa información obtenida como *Cursor* a una colección más manejable, de forma que puedan evitarse transacciones contra la base de datos.

Para este ejemplo, además, se utilizará la clase abstracta `ListAdapter`[5] para crear el adaptador del *RecyclerView*, esta clase simplificará notablemente el código necesario para crear el adaptador.

El primer paso será hacer una modificación de las clases del modelo, esto facilitará la recogida de información y su manejo.

```
data class Editorial(
    val id: Int = 0,
    val name: String? = null
)
```

```
data class SuperHero(
  val id: Int = 0,
  val superName: String? = null,
  val realName: String? = null,
  val favorite: Int = 0,
  val editorial: Editorial = Editorial()
)
```

5 ListAdapter (https://developer.android.com/reference/androidx/recyclerview/widget/ListAdapter)

El siguiente paso será adaptar el método *getAllSuperHeros* para que devuelva en este caso una *MutableList* de *SuperHero* con los datos obtenidos.

```
fun getAllSuperHeros(): MutableList<SuperHero> {
    val result: MutableList<SuperHero> = mutableListOf()
    val db = this.readableDatabase
    val cursor: Cursor =
        db.rawQuery(
            "SELECT * FROM $TABLE_SUPER INNER JOIN $TABLE_EDITORIAL ON " +
                    "$TABLE_SUPER.$COLUMN_ID_ED = $TABLE_EDITORIAL.$COLUMN_ID " +
                    "ORDER BY $COLUMN_SUPERNAME;", null
        )

    if (cursor.moveToFirst()) {
        do {
            result.add(
                SuperHero(
                    cursor.getInt(0), cursor.getString(1),
                    cursor.getString(2), cursor.getInt(3),
                    Editorial(cursor.getInt(5), cursor.getString(6))
                )
            )
        } while (cursor.moveToNext())
    }

    cursor.close()
    return result
}
```

A continuación, se creará el adaptador para el *RecyclerView* utilizando *ListAdapter*. Además, podrás observar como delegar ciertas acciones a la clase que lo implementa, haciendo que el adaptador sea más estándar.

```
class SupersRecyclerAdapter(
    private val onSuperHeroClick: (SuperHero) -> Unit,
    private val onSuperHeroLongClick: (SuperHero) -> Unit,
    private val onFabClick: (SuperHero) -> Unit
) : ListAdapter<SuperHero, SupersViewHolder>(SupersDiffCallback()) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): SupersViewHolder {
        val binding = ItemRecyclerviewBinding.inflate(
            LayoutInflater.from(parent.context), parent, false
        )
        return SupersViewHolder(binding.root)
    }

    override fun onBindViewHolder(holder: SupersViewHolder, position: Int) {
       holder.bind(getItem(position), onSuperHeroClick, onSuperHeroLongClick, onFabClick)
    }
}
```

Como puedes ver, el adaptador para el *RecyclerView* queda bastante sencillo. Como parámetros se van a pasar tres eventos que se controlarán desde la clase que haga uso del adaptador. Observa como extiende de ListAdapter, donde habrá que indicar el tipo de la lista (*SuperHero*) y la clase *ViewHolder* que se detallará a continuación, junto la clase *SupersDiffCallback*, que se encargará de comprobar las diferencias entre los elementos de la lista, este segunda es bastante sencilla como puedes ver.

```
class SupersDiffCallback : DiffUtil.ItemCallback<SuperHero>() {
    override fun areItemsTheSame(oldItem: SuperHero, newItem: SuperHero): Boolean {
        return oldItem.id == newItem.id
    }

    override fun areContentsTheSame(oldItem: SuperHero, newItem: SuperHero): Boolean {
        return oldItem == newItem
    }
}
```

Esta clase extiende de *DiffUtil*, del *callback ItemCallback*, donde se indicará el sistema que se utilizará para diferenciar los elementos, el primero para indicar si son el mismo, y el segundo para indicar si contiene lo mismo, básicamente suele ser así de simple, a no ser que necesites indicar otros criterios.

La clase *ViewHolder* se parece más a las vistas anteriormente con ligeras modificaciones.

```
class SupersViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    val binding = ItemRecyclerviewBinding.bind(view)

    fun bind(
        superHero: SuperHero,
        onSuperHeroClick: (SuperHero) -> Unit,
        onSuperHeroLongClick: (SuperHero) -> Unit,
        onFabClick: (SuperHero) -> Unit
    ) {
        binding.tvSuperName.text = superHero.superName
        binding.tvRealName.text = superHero.realName
        binding.tvEditorial.text = superHero.editorial.name

        binding.ivFab.setImageState(
            intArrayOf(R.attr.state_fab_on), superHero.favorite == 1
        )

        itemView.setOnClickListener { onSuperHeroClick(superHero) }
        itemView.setOnLongClickListener {
            onSuperHeroLongClick(superHero)
            true
        }
        binding.ivFab.setOnClickListener { onFabClick(superHero) }
    }
}
```

Ya solo faltaría crear el adaptador desde el *fragment* que contiene el *RecyclerView*. En el método *onViewCreated* se añadirán las siguiente líneas.

```
adapter = SupersRecyclerAdapter(
    onSuperHeroClick = {
        SuperheroActivity.navigate(
            (requireActivity() as AppCompatActivity),
            it.id
        )
    },
    onSuperHeroLongClick = {
        if (db.delSuperHero(it.id) != 0)
            adapter.submitList(db.getAllSuperHeros())
    },
    onFabClick = {
        db.updateFab(it.id, it.favorite)
        adapter.submitList(db.getAllSuperHeros())
    }
)

binding.recycler.adapter = adapter
adapter.submitList(db.getAllSuperHeros())
```

A destacar, las acciones de los eventos se ejecutan desde esta clase, se libera al adaptador de esta tarea. Además, ya no es necesario hacer uso de la notificación sobre el adaptador, se utiliza el método *submitList* para añadir y refrescar los datos. Fíjate también que se ha modificado el método *navigate* de la clase *SuperheroActivity* de forma que, ahora, se le debe pasar como segundo parámetro el identificador del superhéroe que se ha pulsado.

También se sobrecarga el método *onResume* del *fragment* de la siguiente forma para asegurar mostrar los cambios.

```
override fun onResume() {
    super.onResume()
    adapter.submitList(db.getAllSuperHeros())
}
```

Con el código visto, ya dispones del acceso a la vista detalle con un click, el borrado del ítem mediante una pulsación larga y la opción favorito totalmente funcional.

```
// Actualiza el estado favorito.
fun updateFab(idSuper: Int, stateFav: Int) {
    val args = arrayOf(idSuper.toString())
    val data = ContentValues()

    data.put(COLUMN_FAV, (if (stateFav == 1) 0 else 1))

    val db = this.writableDatabase
    db.update(TABLE_SUPER, data, "$COLUMN_ID = ?", args)
```

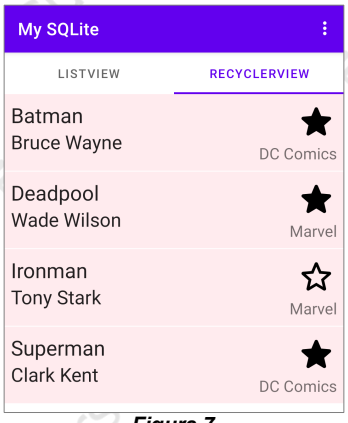

Figura 7

```
10      db.close()
11  }
```

Para la estrella utilizada para marcar como favorito un ítem se ha hecho uso de varios recursos. Se han añadido dos *drawables* con las estrellas, una rellena (*ic_star.xml*) y otra con la silueta (*ic_star_border.xml*), puedes hacerlo añadiéndolas desde la opción imagen vectorial (*New > Vector Asset*).

Seguidamente se creará un recurso ***attr.xml*** en *res/values*, este permite definir variables para utilizar a nivel de diseño, en este caso, se creará una variable de tipo *boolean*.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <attr name="state_fab_on" format="boolean" />
4  </resources>
```

Esta variable se utilizará para indicar en que estado se encuentra el favorito y asignarle la imagen que debe representarse.

Para terminar con el diseño, se añadirá un nuevo *drawable* de tipo *selector* (*New > Drawable Resouce File*) que se llamará *fab_icon.xml*.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <selector xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:app="http://schemas.android.com/apk/res-auto">
4
5      <item android:drawable="@drawable/ic_star_border"
6          android:state_enabled="false" app:state_fab_on="false" />
7
8      <item android:drawable="@drawable/ic_star"
9          android:state_enabled="true" app:state_fab_on="true" />
10 </selector>
```

Como puedes ver en este recurso, según el valor de la variable *state_fab_on* se mostrará una imagen u otra.

A continuación, en le *layout* que contenga el *ImageView* que vaya a indicar el estado de favorito, en su propiedad ***srcCompat*** contendrá el valor ***@drawable/fab_icon***.

Ya por último, desde código se indicará que imagen debe mostrar en cada momento, recuerda que dependerá del valor que contenga el campo favorito de la base de datos.

```
1  binding.ivFab.setImageState(intArrayOf(R.attr.state_fab_on), superHero.favorite == 1)
```

Observa como se establece el estado de la imagen con la propiedad *setImageState* de la vista *ImageView*, indicando, mediante un *array* de enteros la variable creada y, como segundo parámetro, el estado.

11.2. Database Inspector

Desde la versión 4.1. de Android Studio, se dispone de ***Database Inspector***[6] como herramienta para la inspección, búsqueda y modificación de las bases de datos empleadas en las aplicaciones durante su ejecución. Es importante saber que únicamente se podrá utilizar esta herramienta cuando se esté trabajando con la **API 26** o superior y, la base de datos sea SQLite o Room. Lo interesante es que funciona lanzando la aplicación tanto en el emulador, como en un dispositivo físico. Para mostrar el inspector, cuando hayas lanzado la aplicación puedes utilizar la opción de menú ***View > Tool Windows > App Inspection***, y seleccionar la pestaña ***Database Inspector***. o la opción que encontrarás en la barra inferior.

Figura 8

El inspector mostrará un listado con todas las bases de datos que se estén utilizando en la aplicación, permitiendo modificar los registros, añadir o eliminarlos.

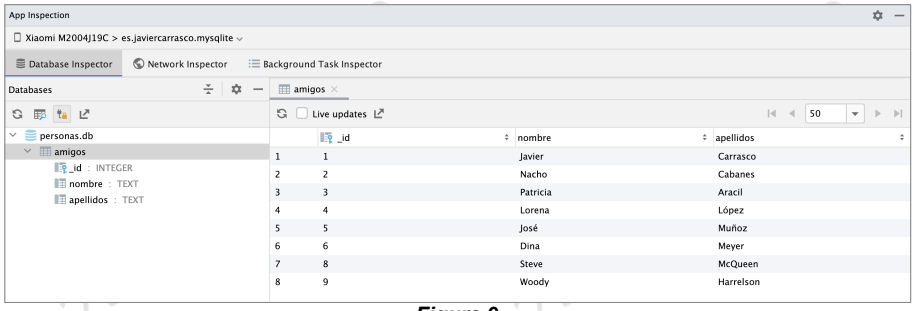

Figura 9

Como puedes ver en la figura, se podrá consultar la estructura de la tabla y su contenido. Según se haya programado la conexión a la base de datos, es posible que se conecte y desconecte según el uso en la aplicación, para evitar esto, puedes utilizar la opción que ofrece el botón  para mantener la conexión de la base de datos abiertas y que esta no se desconecte.

Otra opción que puede resultar interesante del inspector es la opción *New Query*  , que permite lanzar sentencias SQL sobre la base de datos desde el propio Android Studio.

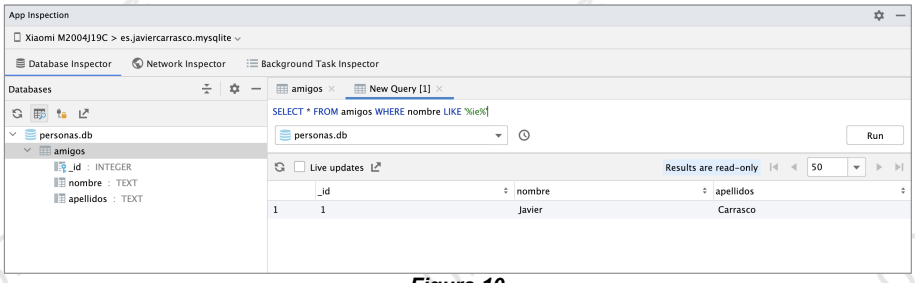

Figura 10

6 *Database Inspector* (https://developer.android.com/studio/inspect/database)

11.3. Room

Ahora que ya sabes como funciona la base de SQLite en Android, se pasará a ver la biblioteca de persistencia **Room**[7]. Este biblioteca ofrece una capa de abstracción sobre la base de datos SQLite, permitiendo un acceso sin problemas, facilitando la creación del esquema, las relaciones existentes y las operaciones SQL que deban realizarse.

Básicamente, se pedirá a Room el DAO (*Data Access Objects*) que se quiere utilizar, y este devolverá una serie de *entities* que se podrán utilizar para modificar la base de datos.

A continuación, se reproducirá el ejemplo del punto anterior, pero esta vez, aplicando la librería Room para la persistencia de datos. Para comenzar, en un nuevo proyecto, deberán añadirse las siguientes dependencias en el `build.gradle` de app, se añadirán únicamente las necesarias, si consultas la documentación, verás que hay otras de carácter opcional, pero para iniciarse en Room bastarán con las mínimas.

```
// 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'
```

Para añadir la biblioteca de anotaciones mediante ***kapt***, deberá añadirse el plugin al *Gradle*.

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

Ahora, se configurarán una serie de opciones del compilador que permiten la actualización de versiones y la exportación del esquema, necesario para el cambio de versión. En el mismo archivo *Gradle* que se está modificando, en la sección *android* en *defaultConfig* se añadirán las siguientes líneas.

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

Existen más opciones, pero para el objetivo buscado, servirán las opciones básicas de

7 Room (https://developer.android.com/topic/libraries/architecture/room)

configuración. Una vez añadidas todas las dependencias, ya puedes sincronizar el proyecto para continuar. Como en el ejemplo anterior, el primer paso será crear las clases modelo, estas serán prácticamente iguales, en este caso, se añadirán las etiquetas necesarias para que Room pueda hacer uso de ellas.

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

```kotlin
@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
)
```

Observa que cada clase está etiquetada con la etiqueta *@Entity*, puedes añadir el atributo *tableName* si quieres cambiar el nombre de la tabla con respecto al nombre de la clase. También se indica que atributo será la clave primaria con la etiqueta *@PrimaryKey*, indicando además que es auto-generada. Existen más etiquetas, como *@ColumnInfo* para indicar el nombre del campo en la tabla si se quiere cambiar.

Si te fijas en la *data class SuperHero*, verás que ya no contiene una propiedad de tipo *Editorial*, esto es debido a que en Room, para obtener datos de varias tablas, pueden emplearse dos enfoques[8].

- **Clase de datos intermedia**, en este enfoque deberá crearse una clase de datos intermedia que definirá el tipo de relación.
- **Multimapa**, se evita el uso de clases intermedias, pero deberá definirse el mapa que devolver y definir las relaciones mediante SQL.

En el punto “Relaciones” que encontrarás más adelante se tratarán ambos enfoques mediante el ejemplo que se está desarrollando.

11.3.1. Creación del DAO

El siguiente paso será la creación de la base de datos e indicar su DAO, los objetos que permitirán el acceso a los datos. Para que esté organizado, todo estará dentro de la clase `SupersDatabase`.

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

8 Relaciones Room (https://developer.android.com/training/data-storage/room/relationships#approaches)

```
}

@Dao
interface SupersDAO { }
```

En este caso se utilizará la etiqueta *@Database*, donde se indicará en un *array* las clases que formarán las entidades de la base de datos. El atributo versión tiene la misma utilidad que en el ejemplo anterior. Esta clase debe ser una clase abstracta que extienda de `RoomDatabase`. La *interface*, que se desarrollará a continuación, simplemente deberá etiquetarse con la etiqueta *@Dao*.

Los DAO son las interfaces en los que cada método será una operación de acceso o modificación de la base de datos. Como verás más adelante, cada método se etiquetará en función de la operación a realizar.

Antes de establecer la conexión, se hará un alto en la actualización de la base de datos, ya que durante el desarrollo, seguramente esta cambie. Para no complicar el primer contacto con Room, se verá como hacer la migración de versión de manera automática. El primer paso ya se ha hecho con las anotaciones del compilador en el punto anterior. Para indicar un cambio de versión, en la etiqueta *@Database*, se añadirán los siguientes argumentos. Supón que se añaden dos tablas nuevas.

```
@Database(
    entities = [
        SuperHero::class,
        Editorial::class,
        Illustrator::class,
        EditorialsIllustrators::class
    ],
    version = 2,
    autoMigrations = [AutoMigration(from = 1, to = 2)]
)
```

Observa como se añaden dos tablas nuevas, se cambia el número de versión y, con el parámetro *autoMigrations* se especifica la migración a realizar, de la versión 1 a la 2. En este último puedes indicar el tipo de migración, por ejemplo, de la 1 a 3 directamente, siempre y cuando la versión 3 ya existiese, o hacer un *downgrade*, de la 2 a la 1.

Si no quieres añadir las anotaciones del compilador y/o estás completamente seguro del diseño y no quieres utilizar este sistema, en la definición de la base de datos puedes utilizar el parámetro *exportSchema* a falso para que no se compruebe.

11.3.2. Establecer conexión con la base de datos

Como se hizo en el ejemplo anterior, se creará una nueva clase que extienda de la clase *Application* con el siguiente contenido para establecer la conexión.

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

```
        private set

    override fun onCreate() {
        super.onCreate()

        supersDatabase = Room.databaseBuilder(
            this, SupersDatabase::class.java, "SuperHeros.db"
        ).build()
    }
}
```

Recuerda añadir la clase al *Manifest* en la propiedad *name* de la etiqueta *application*.

```
<application
    android:name=".MyRoomApplication"
    ...
```

Ahora se continuará con las operaciones CRUD desde el DAO.

11.3.3. Insertar

Para insertar datos se utilizará la etiqueta *@Insert*, además, gracias al atributo *onConflict* se puede elegir que estrategia seguir en caso de conflicto, IGNORE, ABORT o REPLACE.

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

Observa que se utiliza la palabra reservada *suspend*, esto se hace para que todas las operaciones contra la base de datos se hagan fuera del hilo principal. Si no se hiciese así la aplicación fallaría debido a un bloqueo de la UI.

Para insertar ahora una editorial, siguiendo la estructura del ejemplo anterior, se haría de la siguiente manera.

```
class EditorialActivity : AppCompatActivity() {
    ...
    private lateinit var db: SupersDatabase
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        db = (application as MyRoomApplication).supersDatabase
        ...
        binding.button.setOnClickListener {
            ...
            val name = binding.etEditorial.text!!.trim().toString()
            val newEditorial = Editorial(name = name)

            CoroutineScope(Dispatchers.IO).launch {
                db.supersDAO().insertEditorial(newEditorial)
            }

            finish()
```

```
        }
    }
}
```

Observa como se ha sacado del hilo principal la ejecución de la tarea de inserción sobre la base de datos. También se ha creado la variable *db*, a la que se le asigna el valor en el método *onCreate*, esto ha sido simplemente para no repetir constantemente la llamada la clase *MyRoomApplication*.

11.3.4. Eliminar

Siguiendo la estructura utilizada para insertar, para eliminar se utilizará la etiqueta *@Delete*.

```
@Delete
suspend fun deleteEditorial(editorial: Editorial)
```

11.3.5. Actualizar

Para actualizar registros de la base de datos se utilizará la etiqueta *@Update*.

```
@Update
suspend fun updateSuperHero(superHero: SuperHero)
```

11.3.6. Leer

Para recolectar información de la base de datos se utilizará la etiqueta *@Query*, esta deberá contener la sentencia SQL ha ejecutar.

```
@Query("SELECT * FROM SuperHero ORDER BY superName")
suspend fun getAllSuperHeros() : MutableList<SuperHero>
```

También es posible pasarle parámetros a las queries, simplemente habrá que indicar el parámetro en la función y pasarlo a la sentencia, indicándolo con dos puntos ":" delante.

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

Fíjate que en estos métodos bastará con indicar el tipo de dato devuelto, una lista de superhéroes, un único superhéroe, etc.

SimpleCursorAdapter

Si por ejemplo quisieras poblar un *Spinner* mediante un *SimpleCursorAdapter* con las editoriales, una posible solución para conseguir un *Cursor* sería la siguiente.

```
@Query("SELECT idEd as _id, name FROM Editorial")
fun getAllEditorials(): Cursor
```

Fíjate que en la *query* hay que modificar el nombre del campo identificador para que el adaptador lo encuentre, además, no es una función *suspend*, aunque deberás realizar la llamada en

un *coroutine*.

```
CoroutineScope(Dispatchers.Main).launch {
    withContext(Dispatchers.IO) {
        cursor = db.supersDAO().getAllEditorials()
    }

    // Se crea el adaptador mediante SimpleCursorAdapter.
    val adapter = SimpleCursorAdapter(
        this@SuperheroActivity,
        android.R.layout.simple_list_item_2,
        cursor,
        arrayOf(cursor.columnNames[0], cursor.columnNames[1]),
        intArrayOf(android.R.id.text1, android.R.id.text2),
        SimpleCursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER
    )

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

La creación del adaptador en este caso no variará con respecto al ejemplo visto con anterioridad.

ArrayAdapter

Si prefieres utilizar un *ArrayAdapter*, habrá que modificar tanto la recogida como la carga de datos.

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

En este caso, la *query* es más sencilla, no se necesita cambiar el nombre de ninguna columna, pero, deberá adaptarse la lista de editoriales al tipo de datos necesario para cargarlos con el *ArrayAdapter*, esto se hace para poder mostrar el nombre de la editorial en el *Spinner*.

```
CoroutineScope(Dispatchers.Main).launch {
    val adapter: ArrayAdapter<String>

    withContext(Dispatchers.IO) {
        editorialsList = db.supersDAO().getAllEditorials()
    }

    // Se crea un ArrayList<String> con los nombres de las editoriales.
    val editorialsArray = ArrayList<String>().apply {
        editorialsList.forEach {
            this.add(it.name!!)
        }
    }

    // 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
}
```

Podría eliminarse la creación de *editorialArray* y crear el adaptador pasándole directamente *editorialList*. La instanciación del objeto *adapter* sería *ArrayAdapter<Editorial>*, pero en el *Spinner* se obtendría la lista como un objeto, y no quedaría muy bien.

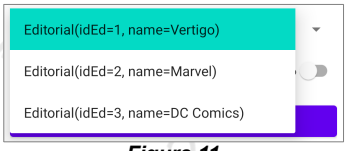

Figura 11

11.3.7. Poblar un ListView con SimpleAdapter

Para la pestaña que muestra el *ListView* de los superhéroes se utilizará esta vez un *SimpleAdapter*. Para este caso, habrá que pasarle una colección de tipo *HashMap*, por lo que habrá que adaptar los datos que se obtengan de la base de datos. Recuerda que el método *getAllSuperHeros* devuelve una *MutableList*.

```
CoroutineScope(Dispatchers.Main).launch {
    val supersList: MutableList<SuperHero>

    withContext(Dispatchers.IO) {
        supersList = db.supersDAO().getAllSuperHeros()
    }

    // Se crea el ArrayList con el HashMap de los superhéroes.
    val supersHashMap: ArrayList<HashMap<String, String>> = ArrayList()
    supersList.forEach {
        val map: HashMap<String, String> = HashMap()
        map.put("superName", it.superName!!)
        map.put("realName", it.realName!!)
        supersHashMap.add(map)
    }

    val adapter = SimpleAdapter(
        context,
        supersHashMap,
        android.R.layout.simple_list_item_2,
        arrayOf("superName", "realName"),
        intArrayOf(android.R.id.text1, android.R.id.text2)
    )
    binding.listView.adapter = adapter
}
```

Como puedes ver, se ha utilizado un *layout* de Android, pero puedes personalizar el tuyo e

indicarlo como tercer parámetro del *SimpleAdapter*.

11.3.8. Poblar un RecyclerView con ListAdapter

En este caso, la única modificación que hay que hacer el adaptador es la forma de obtener el nombre de la editorial, de momento, se mostrará el id de la editorial, que es lo que se obtiene, de momento. La instanciación habrá que adaptarla al uso de corrutinas.

```
adapter = SupersRecyclerAdapter(
    onSuperHeroClick = {
        SuperheroActivity.navigate(
            (requireActivity() as AppCompatActivity),
            it.idSuper
        )
    },
    onSuperHeroLongClick = {
        CoroutineScope(Dispatchers.IO).launch {
            db.supersDAO().deleteSuperHero(it)
            adapter.submitList(db.supersDAO().getAllSuperHeros())
        }
    },
    onFabClick = {
        CoroutineScope(Dispatchers.IO).launch {
            val updated = it.copy(
                favorite = if (it.favorite == 0) 1 else 0
            )
            db.supersDAO().insertSuperHero(updated)
            adapter.submitList(db.supersDAO().getAllSuperHeros())
        }
    }
)
binding.recycler.adapter = adapter
```

Para asegurar la actualización de la información, tras asignar el adaptador al *RecyclerView* se llamará al siguiente método.

```
private fun updateRecycler() {
    CoroutineScope(Dispatchers.Main).launch {
        withContext(Dispatchers.IO) {
            db.supersDAO().getAllSuperHeros()
        }.run {
            adapter.submitList(this)
        }
    }
}
```

También se llamará a este método en *onResume* para actualizar tras volver de añadir o modificar un registro.

11.3.9. Relaciones

Como se comento al inicio de este punto, Room ofrece dos enfoques para afrontar las relaciones entre tablas. A continuación, se verán ambos sistemas, según las necesidades de la aplicación se optará por un sistema u otro, aunque puede que tus preferencias hacia uno de ellos determine la elección.

Enfoque multimapa

Este enfoque es sencillo de implementar, no requiere clases adicionales como en siguiente enfoque, pero requiere escribir instrucciones SQL más complejas, de esta forma, es la sentencia SQL la que se encarga de realizar la mayor parte del trabajo. Por ejemplo.

```
@Query(
    "SELECT * FROM SuperHero " +
            "INNER JOIN Editorial ON idEditorial = idEd ORDER BY superName"
)
suspend fun getSuperHerosWithEditorials(): Map<SuperHero, Editorial>
```

Con esta instrucción se obtiene cada superhéroe junto con la editorial a la que pertenece. Para obtener el resultado y mostrarlo, bastaría con llamar al método en cuestión.

```
CoroutineScope(Dispatchers.Main).launch {
    binding.textView.setText("")

    withContext(Dispatchers.IO) {
        db.supersDAO().getSuperHerosWithEditorials()
    }.run {
        this.forEach {
            binding.textView.append("${it}\n\n")
        }
    }
}
```

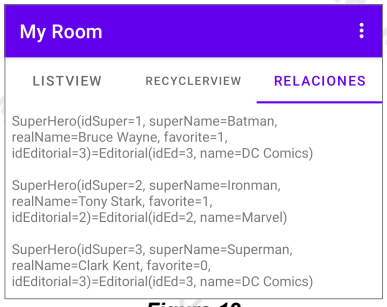

Figura 12

Enfoque de clases intermedias

Se establece la relación entre las tablas *SuperHero* y *Editorials*, según el diseño, se sabe que un Superhéroe únicamente pertenece a una Editorial, y que una Editorial tiene más de un Superhéroe, una relación de 1 a N.

Relaciones de 1 a 1

Se añadirá una clase intermedia al diseño planteado anteriormente. En este caso se plantea como una relación de 1 a 1, ya que se obtendrán los superhéroes y la editorial a la que pertenecen.

```
// Relación de 1 a 1.
data class SupersWithEditorial(
    @Embedded val supers: SuperHero,
    @Relation(
```

```
        parentColumn = "idEditorial",
        entityColumn = "idEd"
    ) val editorial: Editorial
)
```

Fíjate que esta nueva data class no está etiquetada con *@Entity*, pero sí se aplica la etiqueta *@Embedded* para indicar que tabla tendrá datos incluidos y la etiqueta *@Relation* que indicará los atributos que establecen la relación.

En el DAO se añadirá el método encargado de recuperar la información basándose en la clase intermedia.

```
@Transaction
@Query("SELECT * FROM SuperHero ORDER BY superName")
suspend fun getAllSuperHerosWithEditorials(): MutableList<SupersWithEditorial>
```

Como esta información se mostrará en el *RecyclerView*, hay que realizar una serie de modificaciones, en *SupersRecyclerAdapter*, únicamente hay que cambiar el tipo que devuelve el *ListAdapter*.

```
ListAdapter<SupersWithEditorial, SupersViewHolder>(SupersDiffCallback()) { ...
```

La clase *SupersDiffCallback* también necesitará un cambio de tipo.

```
class SupersDiffCallback : DiffUtil.ItemCallback<SupersWithEditorial>() {
    override fun areItemsTheSame(
        oldItem: SupersWithEditorial,
        newItem: SupersWithEditorial
    ): Boolean {
        return oldItem.supers.idSuper == newItem.supers.idSuper
    }

    override fun areContentsTheSame(
        oldItem: SupersWithEditorial,
        newItem: SupersWithEditorial
    ): Boolean {
        return oldItem == newItem
    }
}
```

Observa como hay que cambiar también la forma en la que se hace referencia al identificador del superhéroe. Por último, se modificará la clase *SupersViewHolder*, adaptándolo a la nueva clase.

```
class SupersViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    val binding = ItemRecyclerviewBinding.bind(view)

    fun bind(
        superHero: SupersWithEditorial,
        onSuperHeroClick: (SuperHero) -> Unit,
        onSuperHeroLongClick: (SuperHero) -> Unit,
        onFabClick: (SuperHero) -> Unit
```

```
    ) {
        binding.tvSuperName.text = superHero.supers.superName
        binding.tvRealName.text = superHero.supers.realName
        binding.tvEditorial.text = superHero.editorial.name

        binding.ivFab.setImageState(
            intArrayOf(R.attr.state_fab_on), superHero.supers.favorite == 1
        )

        itemView.setOnClickListener { onSuperHeroClick(superHero.supers) }
        itemView.setOnLongClickListener {
            onSuperHeroLongClick(superHero.supers)
            true
        }
        binding.ivFab.setOnClickListener { onFabClick(superHero.supers) }
    }
}
```

La instanciación del adaptador no cambiará, por lo que quedará como se vio al poblar el *RecyclerView*, ya que la mayor adaptación se ha hecho en la clase *SupersRecyclerAdapter*, haciendo uso de la clase *SuperHero*.

Relaciones de 1 a N

La clase intermedia que representará la relación de 1 a N variará con respecto a la anterior. En este caso se obtendrá la relación entre editoriales y superhéroes, teniendo en cuenta que una editorial puede contener uno o más superhéroes.

```
// Relación de 1 a N.
data class EditorialWithSupers(
    @Embedded val editorial: Editorial,
    @Relation(
        parentColumn = "idEd",
        entityColumn = "idEditorial"
    ) val supers: List<SuperHero>
)
```

En este caso, la variable *supers* será la lista de los superhéroes que pertenecen a la editorial que corresponda. En el DAO se añadirá el método encargado de recuperar la información basándose en la clase intermedia.

```
@Transaction
@Query("SELECT * FROM Editorial ORDER BY name")
suspend fun getAllEditorialWithSupers(): MutableList<EditorialWithSupers>
```

La forma de mostrar la información ya dependerá de donde y como mostrarla, por ejemplo, en un *TextView*.

```
CoroutineScope(Dispatchers.Main).launch {
    withContext(Dispatchers.IO) {
```

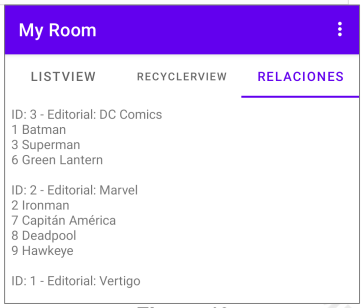

Figura 13

```
            db.supersDAO().getAllEditorialWithSupers()
        }.run {
            if (this.isNotEmpty()) {
                this.forEach {
                    binding.textView.append("${it}\n")
                }
            }
        }
    }
```

Es importante realizar la comprobación de nulo para no tener problemas durante la ejecución, también se ha sobrecargado el método *toString* de la clase *EditorialWithSupers* para facilitar la representación de la información.

Relaciones de N a N

Para las relaciones de N a N será necesario crear una tercera tabla, la que se encargue de relacionar las dos tablas principales, lo que se conoce como tabla o entidad asociativa o de referencias cruzadas.

Para crear una relación de N a N en este ejemplo se creará una tercera tabla para guardar los dibujantes de cómics. La idea es conocer que dibujantes trabajan en la editoriales, de tal forma que una editorial puede tener cero o muchos dibujantes y un dibujante puede trabajar para más de una editorial.

```
@Entity
data class Illustrator(
    @PrimaryKey(autoGenerate = true)
    val idIllustrator: Int = 0,
    val nameIllustrator: String? = null
)
```

También deberá crearse la *data class* que establecerá las referencias cruzadas.

```
@Entity(primaryKeys = ["idIllustrator", "idEd"])
data class EditorialsIllustrators(
    val idIllustrator: Int,
    val idEd: Int
)
```

Recuerda que deberás añadir estas nuevas entidades a la definición de la base de datos en la clase *SupersDatabase*. Para evitar hacer una migración o cambio de versión, puedes desinstalar la aplicación del dispositivo o emulador y volver a instalarla, pero recuerda que se perderán los datos. Otra opción es aplicar la migración de versiones[9].

Para este tipo de relaciones, debes plantearte como quieres obtener la información para crear las clases intermedias. Puedes obtener las editoriales y todos los dibujantes que tengan, obtener los dibujantes y todas las editoriales para las que trabajen, o ambas.

9 Migración de versiones en Room (https://developer.android.com/training/data-storage/room/migrating-db-versions)

```
data class EditorialsWithIllustrators(
    @Embedded val editorial: Editorial,
    @Relation(
        parentColumn = "idEd",
        entityColumn = "idIllustrator",
        associateBy = Junction(EditorialsIllustrators::class)
    ) val illustrator: List<Illustrator>
)

data class IllustratrosWithEditorials(
    @Embedded val illustrator: Illustrator,
    @Relation(
        parentColumn = "idIllustrator",
        entityColumn = "idEd",
        associateBy = Junction(EditorialsIllustrators::class)
    ) val editorial: List<Editorial>
)
```

Ya solo faltaría añadir los métodos al DAO para obtener la información. No se ha comentado con anterioridad, la etiqueta *@Transaction* asegura que la operación se realice automáticamente.

```
@Transaction
@Query("SELECT * FROM Editorial")
suspend fun getAllEditorialsWithIllustrators(): MutableList<EditorialsWithIllustrators>

@Transaction
@Query("SELECT * FROM Illustrator")
suspend fun getAllIllustratorsWithEditorials(): MutableList<IllustratrosWithEditorials>
```

La forma de mostrar la información ya dependerá de donde y qué se quiere mostrar, por ejemplo, mostrar en un *TextView* las editoriales y sus dibujantes.

```
CoroutineScope(Dispatchers.Main).launch {
    withContext(Dispatchers.IO) {
        db.supersDAO().getAllEditorialsWithIllustrators()
    }.run {
        if (this.isNotEmpty()) {
            this.forEach {
                binding.textView.append("${it}\n")
            }
        }
    }
}
```

Figura 14

O mostrar todos los ilustradores y las editoriales para las que trabajan, cambiando simplemente el método llamado del DAO por *getAllIllustratorsWithEditorials*.

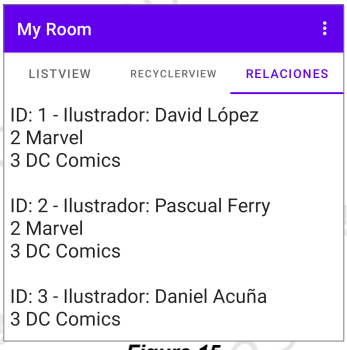

Figura 15