

Programación Multimedia y Dispositivos Móviles

UD 9. DB Adapter

Javier Carrasco

Curso 2021 / 2022



Este obra está bajo una licencia de Creative Commons Reconocimiento 4.0 Internacional.
Última actualización: septiembre de 2021.

Versión impresa en Amazon: <https://www.amazon.es/dp/B08NM4XV4Q>

DB Adapter

9. DB Adapter.....	3
9.1. <i>SQLiteOpenHelper.....</i>	3
Insertar.....	4
Eliminar.....	5
Actualizar.....	5
9.2. <i>Cursor.....</i>	8
9.2.1. <i>SimpleCursorAdapter.....</i>	9
9.3. <i>CursorAdapter.....</i>	11
9.4. Mostrar en un <i>RecyclerView.....</i>	15
9.5. <i>Database Inspector.....</i>	20

9. DB Adapter

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 de texto 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 para Android.

9.1. *SQLiteOpenHelper*

La clase que se tratará en este punto ayuda con la creación y control de versión de la base de datos. 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  class MyDBOpenHelper(context: Context, factory: SQLiteDatabase.CursorFactory?) :
2      SQLiteOpenHelper(context, DATABASE_NAME, factory, DATABASE_VERSION) {
3
4     val TAG = "SQLite"
5     companion object {
6         val DATABASE_VERSION = 1
7         val DATABASE_NAME = "personas.db"
8         val TABLA_AMIGOS = "amigos"
9         val COLUMNA_ID = "_id"
10        val COLUMNA_NOMBRE = "nombre"
11        val COLUMNA_APELLIDOS = "apellidos"
12    }
13
14    /**
15     * Este método es llamado cuando se crea la base por primera vez. Debe
16     * producirse la creación de todas las tablas que formen la base de datos.
17     */
18    override fun onCreate(db: SQLiteDatabase?) {
19        try {
20            val crearTablaAmigos = "CREATE TABLE $TABLA_AMIGOS " +
21                "($COLUMNA_ID INTEGER PRIMARY KEY AUTOINCREMENT, " +
22                "$COLUMNA_NOMBRE TEXT, " +
23                "$COLUMNA_APELLIDOS TEXT)"
24            db.execSQL(crearTablaAmigos)
25        } catch (e: SQLiteException) {
26            Log.e("$TAG (onCreate)", e.message.toString())
27        }
28    }

```

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

4 UNIDAD 9 DB ADAPTER

```
29     /**
30      * Este método se invocará cuando la base de datos necesite ser actualizada.
31      * Se utiliza para hacer DROPs, añadir tablas o cualquier acción que
32      * actualice el esquema de la BD.
33     */
34     override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion: Int){
35         try {
36             val dropTablaAmigos = "DROP TABLE IF EXISTS $TABLA_AMIGOS"
37             db!!.execSQL(dropTablaAmigos)
38             onCreate(db)
39         } catch (e: SQLiteException) {
40             Log.e("$TAG (onUpgrade)", e.message.toString())
41         }
42     }
43
44     /**
45      * Método opcional. Se llamará a este método después de abrir la base de
46      * datos, antes de ello, comprobará si está en modo lectura. Se llama justo
47      * después de establecer la conexión y crear el esquema.
48     */
49     override fun onOpen(db: SQLiteDatabase?) {
50         super.onOpen(db)
51         Log.d("$TAG (onOpen)", "¡¡Base de datos abierta!!")
52     }
53 }
```

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 la s tablas, el nombre de las tablas y de la propia base de datos, que pueden causarnos problemas si se escriben incorrectamente.

Insertar

El método para añadir un amigo a la tabla *amigos* podría quedar de la siguiente manera dentro de la propia clase `MyDBOpenHelper()`.

```
1 /**
2  * Método para añadir un amigo a la tabla amigos.
3 */
```

```

4  fun addAmigo(name: String, surname: String) {
5      // Se crea un ArrayMap<String, String> haciendo uso de ContentValues().
6      val data = ContentValues()
7      data.put(COLUMN_NOMBRE, name)
8      data.put(COLUMN_APELLIDOS, surname)
9
10     // Se abre la BD en modo escritura.
11     val db = this.writableDatabase
12     db.insert(TABLA_AMIGOS, null, data)
13     db.close()
14 }
```

Eliminar

```

1  /**
2  * Método para eliminar un amigo de la tabla por el identificador.
3  */
4  fun delAmigo(identifier: Int): Int {
5      val args = arrayOf(identifier.toString())
6
7      // Se abre la BD en modo escritura.
8      val db = this.writableDatabase
9
10     // Se puede elegir un sistema u otro.
11     val result = db.delete(TABLA_AMIGOS, "$COLUMN_ID = ?", args)
12     // db.execSQL("DELETE FROM $TABLA_AMIGOS WHERE $COLUMN_ID = ?", args)
13
14     db.close()
15     return result
16 }
```

Actualizar

```

1  /**
2  * Método para actualizar el nombre de un amigo de la tabla por el id.
3  */
4  fun updateAmigo(identifier: Int, newName: String) {
5      val args = arrayOf(identifier.toString())
6
7      // Se crea un ArrayMap<String, String> con los datos nuevos.
8      val data = ContentValues()
9      data.put(COLUMN_NOMBRE, newName)
10
11     val db = this.writableDatabase
12     db.update(TABLA_AMIGOS, data, "$COLUMN_ID = ?", args)
13     db.close()
14 }
```

6 UNIDAD 9 DB ADAPTER

Si no te gusta utilizar los métodos `insert()`, `delete()` y `update()`, puedes hacer uso del método `execSQL()`, este método necesitará de la instrucción SQL 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.

Con la clase `Helper` ya casi terminada, ahora deberá crearse la interfaz de la actividad `activity_main.xml` para hacer uso de las acciones establecidas en el código.

A continuación, se añadirá el siguiente código a la clase principal para darle la funcionalidad adecuada a los botones que se han añadido para lanzar las diferentes acciones.

```
1 class MainActivity : AppCompatActivity() {
2     private lateinit var binding: ActivityMainBinding
3     private lateinit var amigosDBHelper: MyDBOpenHelper
4
5     override fun onCreate(savedInstanceState: Bundle?) {
6         super.onCreate(savedInstanceState)
7         binding = ActivityMainBinding.inflate(layoutInflater)
8         setContentView(binding.root)
9
10        // Se instancia el objeto MyDBOpenHelper.
11        amigosDBHelper = MyDBOpenHelper(this, null)
12        with(binding) {
13            // Botón INSERTAR.
14            btnInsertar.setOnClickListener {
15                if (etNombre.text.isNotBlank() && etApes.text.isNotBlank()) {
16                    // Se inserta en la tabla.
17                    amigosDBHelper.addAmigo(
18                        etNombre.text.toString(),
19                        etApes.text.toString()
20                    )
21                    // Se limpian los EditText después de la inserción.
22                    etNombre.text.clear()
23                    etApes.text.clear()
24                } else {
25                    myToast("Los campos no pueden estar vacíos.")
26                }
27            }
28
29            // Botón ACTUALIZAR.
30            btnActualizar.setOnClickListener {
31                if (etNombre.text.isNotBlank()) {
32                    // Se lanza el dialogo para solicitar el id del registro,
33                    // además, se indica el tipo de operación.
34                    solicitaIdentificador(UPDATE)
35                } else {
```



Figura 1

```
36             myToast("El campo nombre no debe estar vacío.")
37         }
38     }
39
40     // Botón ELIMINAR.
41     btnEliminar.setOnClickListener {
42         // Se lanza el dialogo para solicitar el id del registro,
43         // además, se indica el tipo de operación.
44         solicitaIdentificador(DELETE)
45     }
46 }
47 }
48
49 /**
50 * Método encargado de mostrar un cuadro de diálogo para pedir el
51 * identificador al usuario y realizar la acción correspondiente según
52 * la acción requerida.
53 */
54 fun solicitaIdentificador(accion: String) {
55     // Se infla la vista para el diálogo.
56     val myDialogView = LayoutInflater.from(this@MainActivity)
57         .inflate(R.layout.dialogo, null)
58     // Se crea el builder.
59     val builder = AlertDialog.Builder(this)
60         .setView(myDialogView)
61
62     builder.apply {
63         setPositiveButton(android.R.string.ok) { dialog, _ ->
64             val valor = myDialogView
65                 .findViewById<EditText>(R.id.identificador).text
66             val identificador = valor.toString().toInt()
67
68             // Se realiza la acción.
69             when (accion) {
70                 UPDATE -> {
71                     val nombre = binding.etNombre.text.toString()
72                     amigosDBHelper.updateAmigo(identificador, nombre)
73                     // Se limpian los EditText después de la inserción.
74                     binding.etNombre.text.clear()
75                     binding.etApes.text.clear()
76                 }
77                 DELETE -> {
78                     myToast(
79                         "Eliminado/s " +
80                         "${amigosDBHelper.delAmigo(identificador)} " +
81                         "registro/s"
82                     )
83                 }
84             }
85         }
86     }
87 }
```

8 UNIDAD 9 DB ADAPTER

```
86         setNegativeButton(android.R.string.cancel) { dialog, _ ->
87             dialog.dismiss()
88         }
89     }.show()
90 }
91
92 fun myToast(mensaje: String) {
93     Toast.makeText(
94         this@MainActivity,
95         mensaje,
96         Toast.LENGTH_SHORT
97     ).show()
98 }
99 }
```

Si te fijas en la instanciación del objeto `MyDBOpenHelper()`, el segundo parámetro, `factory`, se establece a `null` porque se busca el comportamiento por defecto (según documentación de Google), es decir, para indicar que se sabe trabajar con bases de datos.

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**² para ver su contenido.

Como habrás podido observar en el ejemplo, tanto para actualizar como para eliminar, se necesita conocer el identificador del registro. A continuación, se añadirá la posibilidad de consultar los datos almacenados en la base de datos, para eso se hará uso de la clase `Cursor`.

9.2. Cursor

Para poder mostrar la información almacenada en la base de datos, se añadirá un nuevo botón para poder lanzar una consulta, así como un `TextView` donde mostrar la información recuperada, la UI quedará como muestra la figura.

En el `setOnClickListener()` del nuevo botón se añadirá el siguiente código para lanzar una consulta sobre la base de datos.

```
1 with(binding) {
2     ...
3     // Botón CONSULTAR.
4     btnConsultar.setOnClickListener {
5         tvResult.text = ""
6
7         // Se instancia la BD en modo lectura y se crea la SELECT.
8         val db: SQLiteDatabase = amigosDBHelper.readableDatabase
```



Figura 2

```

9     val cursor: Cursor = db.rawQuery(
10        "SELECT * FROM ${MyDBOpenHelper.TABLA_AMIGOS};",
11        null
12    )
13
14    // Se comprueba que al menos exista un registro.
15    if (cursor.moveToFirst()) {
16        do {
17            tvResult.append(cursor.getInt(0).toString() + " - ")
18            tvResult.append(cursor.getString(1).toString() + " ")
19            tvResult.append(cursor.getString(2).toString() + "\n")
20        } while (cursor.moveToNext())
21    } else {
22        myToast("No existen datos a mostrar.")
23    }
24    db.close()
25 }
26 }
```

En este caso, se ejecuta la *query* desde la `MainActivity.kt` ya que, si se hace desde la clase `MyDBOpenHelper()` y se cierra la base de datos desde ella, se perdería el *cursor*.

En este ejemplo, se puede ver como se hace uso de la clase `SQLiteDatabase()` para instanciar la base de datos mediante el *helper* creado. También se utiliza la clase `Cursor()` al crear la variable `cursor`, ésta permite recoger la información devuelta por la ejecución de la sentencia SQL mediante el método `rawQuery()`, permitiendo el acceso de manera aleatoria 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.

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



Figura 3

9.2.1. SimpleCursorAdapter

En el siguiente punto se verá como hacer uso de un `CursorAdapter`, mediante el cual es posible personalizar el adaptador, pero 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`³, 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.

3 `SimpleCursorAdapter` (<https://developer.android.com/reference/kotlin/android/widget/SimpleCursorAdapter>)

10 UNIDAD 9 DB ADAPTER

El siguiente ejemplo muestra como montar un `SimpleCursorAdapter` y asignarlo a un `Spinner`, así como capturar el elemento seleccionado.

```
1 // Se obtiene el Curso con la información a cargar.  
2 val cursor = superHeroDBHelper.getAllEditorials()  
3  
4 // Se crea el adaptador mediante SimpleCursorAdapter.  
5 val adapter = SimpleCursorAdapter(  
6     this,  
7     android.R.layout.simple_list_item_2,  
8     cursor,  
9     arrayOf(cursor.columnNames[0], cursor.columnNames[1]),  
10    intArrayOf(android.R.id.text1, android.R.id.text2),  
11    SimpleCursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER  
12 )  
13  
14 // Se carga el adaptador en el Spinner.  
15 binding.spinner.adapter = adapter
```

Como puedes observar, la obtención del `Cursor`, y la asignación del adaptador a un elemento es algo ya visto y que no cambia para la creación de este tipo de adaptador. 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 represente los elementos, en este caso, se utiliza uno predefinido.
- **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 injectar.
- **to**: será el `array` de los identificadores de las vistas que contiene el `layout` donde se injectarán los campos indicados en `from`. Al utilizarse un `layout` predefinido, el nombre 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`.

```

1 binding.spinner.onItemSelectedListener = object :
2     AdapterView.OnItemSelectedListener {
3
4         override fun onItemSelected(
5             adapterView: AdapterView<*>?,
6             view: View?,
7             pos: Int,
8             id: Long
9         ) {
10            val cursorEd = binding.spinner.getItemAtPosition(pos) as Cursor
11            editorialSeleccionada = cursorEd.getInt(0)
12            Log.d(
13                "Spinner",
14                "${cursorEd.getString(0)} - ${cursorEd.getString(1)}"
15            )
16        }
17
18        override fun onNothingSelected(p0: AdapterView<*>?) {}
19    }

```

La sobrecarga del método `onItemSelected()` es la encargada de recoger la posición seleccionada, la variable `cursorEd` 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.

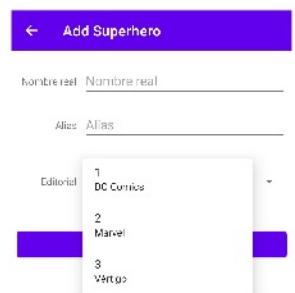


Figura 4

9.3. CursorAdapter

Si se pretende rellenar un `ListView` con datos obtenidos de una base de datos, no se puede utilizar un `cursor` directamente, deberá hacerse uso de la clase `CursorAdapter`.

`CursorAdapter` 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()`.

`bindView()` se encargará de llenar 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, se hace internamente.

Para actualizar esta aplicación, se añadirá un nuevo botón en la parte baja, éste permitirá ver los datos en un `ListView` abriendo una nueva `activity`.

Se creará una nueva actividad vacía (`ListviewActivity.kt` y `activity_listview.xml`), ésta contendrá la lista que mostrará los datos.

12 UNIDAD 9 DB ADAPTER

activity_listview.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:app="http://schemas.android.com/apk/res-auto"
5     xmlns:tools="http://schemas.android.com/tools"
6     android:layout_width="match_parent"
7     android:layout_height="match_parent"
8     tools:context=".ListviewActivity">
9
10    <ListView
11        android:id="@+id/myListview"
12        android:layout_width="match_parent"
13        android:layout_height="match_parent"
14        android:layout_marginStart="8dp"
15        android:layout_marginEnd="8dp"
16        app:layout_constraintBottom_toBottomOf="parent"
17        app:layout_constraintEnd_toEndOf="parent"
18        app:layout_constraintStart_toStartOf="parent"
19        app:layout_constraintTop_toTopOf="parent" />
20 </androidx.constraintlayout.widget.ConstraintLayout>
```

También será necesario crear el *layout* para los elementos que formen la lista, item_listview.xml .

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:tools="http://schemas.android.com/tools"
5     android:layout_width="match_parent"
6     android:layout_height="match_parent"
7     android:orientation="vertical">
8
9    <TextView
10        android:id="@+id/tvItemNombre"
11        android:layout_width="match_parent"
12        android:layout_height="wrap_content"
13        android:textSize="24sp"
14        android:textStyle="bold"
15        tools:text="Nombre" />
16    <TextView
17        android:id="@+id/tvItemApes"
18        android:layout_width="match_parent"
19        android:layout_height="wrap_content"
20        android:layout_marginBottom="10dp"
21        android:textSize="18sp"
22        android:textStyle="italic"
23        tools:text="Apellidos" />
24 </LinearLayout>
```

A continuación, se añade el código a la clase `ListviewActivity.kt`, donde se creará el `cursor adapter` personalizado, en este caso no se hará en una clase a parte, ya que el contenido es muy sencillo, y esta `activity` únicamente se encargará de completar la lista.

```
1 class ListviewActivity : AppCompatActivity() {
2     private lateinit var binding: ActivityListviewBinding
3     private val amigosDBHelper = MyDBOpenHelper(this, null)
4
5     override fun onCreate(savedInstanceState: Bundle?) {
6         super.onCreate(savedInstanceState)
7         binding = ActivityListviewBinding.inflate(layoutInflater)
8         setContentView(binding.root)
9         setTitle(R.string.bt_ver_listview)
10
11         // Se instancia la BD en modo lectura y se crea la SELECT.
12         val db: SQLiteDatabase = amigosDBHelper.readableDatabase
13         val cursor: Cursor = db.rawQuery(
14             "SELECT * FROM ${MyDBOpenHelper.TABLA_AMIGOS};",
15             null
16         )
17         // Se crea el CursorAdapter.
18         val myListCursorAdapter = MyListCursorAdapter(this, cursor)
19         // Se carga los datos en el ListView.
20         binding myListview.adapter = myListCursorAdapter
21         db.close()
22     }
23
24     inner class MyListCursorAdapter(context: Context, cursor: Cursor) :
25         CursorAdapter(context, cursor, FLAG_REGISTER_CONTENT_OBSERVER) {
26         /**
27          * "Infla" cada uno de los elementos de la lista.
28          */
29         override fun newView(
30             context: Context?,
31             cursor: Cursor?,
32             parent: ViewGroup?
33         ): View {
34             val inflater = LayoutInflator.from(context)
35             return inflater.inflate(R.layout.item_listview, parent, false)
36         }
37         /**
38          * Rellena el ListView.
39          */
40         override fun bindView(view: View?, context: Context?, cursor: Cursor?) {
41             val bindingItems = ItemListviewBinding.bind(view!!)
42
43             with(bindingItems) {
44                 tvItemNombre.text = cursor!!.getString(1)
45                 tvItemApes.text = cursor.getString(2)
46                 view.setOnClickListener {
```

14 UNIDAD 9 DB ADAPTER

```
47         Toast.makeText(
48             this@ListviewActivity,
49             "${tvItemNombre.text} ${tvItemApes.text}",
50             Toast.LENGTH_SHORT
51         ).show()
52     }
53 }
54 }
55 }
56 }
```

Ya se puede añadir el código para lanzar la nueva *activity* desde la *MainActivity*. Recuerda modificar el fichero *manifest* si quieres activar el botón "atrás" en la barra de aplicación.

```
1 with(binding) {
2     ...
3     // Botón VER EN LISTVIEW.
4     btnVerListview.setOnClickListener {
5         val myIntent = Intent(this@MainActivity, ListviewActivity::class.java)
6         startActivity(myIntent)
7     }
8 }
```

El resultado obtenido con esta modificación del código se muestra en las siguientes imágenes.

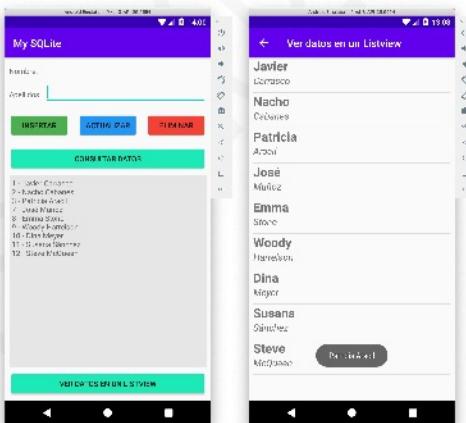


Figura 5

9.4. Mostrar en un *RecyclerView*

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*.

Se añadirá un nuevo botón en la parte baja de la actividad principal, de forma que cuando se pulse sobre este nuevo botón, la aplicación abrirá una *activity* nueva que mostrará la información de la consulta en un *RecyclerView*.

Crea una nueva *activity* vacía (`RecyclerviewActivity.kt` y `activity_recyclerview.xml`), además se deberá añadir un nuevo *layout* para los elementos del *RecyclerView*.

`activity_recyclerview.xml`

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:app="http://schemas.android.com/apk/res-auto"
5     xmlns:tools="http://schemas.android.com/tools"
6     android:layout_width="match_parent"
7     android:layout_height="match_parent"
8     tools:context=".RecyclerviewActivity">
9
10    <androidx.recyclerview.widget.RecyclerView
11        android:id="@+id/myRecyclerview"
12        android:layout_width="match_parent"
13        android:layout_height="match_parent"
14        app:layout_constraintBottom_toBottomOf="parent"
15        app:layout_constraintEnd_toEndOf="parent"
16        app:layout_constraintStart_toStartOf="parent"
17        app:layout_constraintTop_toTopOf="parent" />
18 </androidx.constraintlayout.widget.ConstraintLayout>
```

`item_recyclerview.xml`

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.cardview.widget.CardView
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:app="http://schemas.android.com/apk/res-auto"
5     xmlns:tools="http://schemas.android.com/tools"
6     android:id="@+id/cvItem"
7     android:layout_width="match_parent"
8     android:layout_height="wrap_content"
9     android:layout_margin="5dp"
10    android:clickable="true"
11    android:focusable="true"
12    android:foreground="?attr/selectableItemBackground"
```

16 UNIDAD 9 DB ADAPTER

```
13     android:foregroundTint="#F0F0F0"
14     android:padding="15dp"
15     app:cardCornerRadius="10dp"
16     app:cardElevation="8dp">
17 <RelativeLayout
18     android:id="@+id/rootLayout"
19     android:layout_width="match_parent"
20     android:layout_height="match_parent"
21     android:background="@color/cardview_light_background"
22     android:padding="10dp">
23     <ImageView
24         android:id="@+id/imgAmigo"
25         android:layout_width="100dp"
26         android:layout_height="100dp"
27         android:layout_marginEnd="10dp"
28         android:contentDescription="@string/txt_imagen"
29         app:srcCompat="@mipmap/ic_launcher_round" />
30     <TextView
31         android:id="@+id/tvIdentificador"
32         android:layout_width="20dp"
33         android:layout_height="20dp"
34         android:layout_alignParentTop="true"
35         android:layout_alignParentEnd="true"
36         android:layout_marginTop="6dp"
37         android:layout_marginEnd="6dp"
38         android:textAlignment="center"
39         android:textStyle="italic"
40         tools:text="id" />
41     <TextView
42         android:id="@+id/tvNombre"
43         android:layout_width="match_parent"
44         android:layout_height="wrap_content"
45         android:layout_marginStart="5dp"
46         android:layout_marginTop="10dp"
47         android:layout_toEndOf="@+id/imgAmigo"
48         android:textSize="30sp"
49         android:textStyle="bold"
50         tools:text="Nombre" />
51     <TextView
52         android:id="@+id/tvApes"
53         android:layout_width="match_parent"
54         android:layout_height="wrap_content"
55         android:layout_below="@+id/tvNombre"
56         android:layout_toEndOf="@+id/imgAmigo"
57         android:textSize="24sp"
58         android:textStyle="italic"
59         tools:text="Apellidos" />
60     </RelativeLayout>
61 </androidx.cardview.widget.CardView>
```

A continuación, se creará la clase `MyRecyclerViewAdapter.kt`, en la que se implementará el adaptador del `RecyclerView`. Básicamente es igual a los vistos anteriormente, será en el método `onBindViewHolder()` donde se deberán añadir las líneas que permita recorrer el *cursor*.

```
1 class MyRecyclerViewAdapter
2     : RecyclerView.Adapter<MyRecyclerViewAdapter.ViewHolder>() {
3
4     private lateinit var context: Context
5     private lateinit var cursor: Cursor
6
7     fun MyRecyclerViewAdapter(context: Context, cursor: Cursor) {
8         this.context = context
9         this.cursor = cursor
10    }
11
12    /**
13     * Se "infla" la vista de los items.
14     */
15    override fun onCreateViewHolder(
16        parent: ViewGroup,
17        viewType: Int
18    ): ViewHolder {
19        Log.d("RECYCLERVIEW", "onCreateViewHolder")
20        val inflater = LayoutInflater.from(parent.context)
21        return ViewHolder(
22            inflater.inflate(
23                R.layout.item_recyclerview,
24                parent,
25                false
26            )
27        )
28    }
29
30    override fun getItemCount(): Int {
31        return cursor.count
32    }
33
34    /**
35     * Se completan los datos de cada vista mediante ViewHolder.
36     */
37    override fun onBindViewHolder(
38        holder: MyRecyclerViewAdapter.ViewHolder,
39        position: Int
40    ) {
41        // Importante para recorrer el cursor.
42        cursor.moveToPosition(position)
43        Log.d("RECYCLERVIEW", "onBindViewHolder")
44
45        // Se asignan los valores a los elementos de la UI.
46        holder.id.text = cursor.getString(0)
```

18 UNIDAD 9 DB ADAPTER

```
47     holder.nombre.text = cursor.getString(1)
48     holder.apes.text = cursor.getString(2)
49 }
50
51     inner class ViewHolder : RecyclerView.ViewHolder {
52         // Creamos las referencias de la UI.
53         val id: TextView
54         val nombre: TextView
55         val apes: TextView
56
57         constructor(view: View) : super(view) {
58             // Se enlazan los elementos de la UI mediante ViewBinding.
59             val bindingItemsRV = ItemRecyclerviewBinding.bind(view)
60             this.id = bindingItemsRV.tvIdentificador
61             this.nombre = bindingItemsRV.tvNombre
62             this.apes = bindingItemsRV.tvApes
63
64             view.setOnClickListener {
65                 Toast.makeText(
66                     context,
67                     "${this.id.text}-${this.nombre.text} ${this.apes.text}",
68                     Toast.LENGTH_SHORT
69                 ).show()
70             }
71         }
72     }
73 }
```

Con este adaptador ya estaría relleno el *RecyclerView*, además de añadir un *listener* para la pulsación sobre algún elemento. Por último, en la clase *RecyclerViewActivity.kt* se añadirá el siguiente código.

```
1 class RecyclerviewActivity : AppCompatActivity() {
2     private lateinit var binding: ActivityRecyclerviewBinding
3     private val amigosDBHelper = MyDBOpenHelper(this, null)
4     private lateinit var db: SQLiteDatabase
5
6     override fun onCreate(savedInstanceState: Bundle?) {
7         super.onCreate(savedInstanceState)
8         binding = ActivityRecyclerviewBinding.inflate(layoutInflater)
9         setContentView(binding.root)
10        setTitle(R.string.bt_ver_recyclerview)
11
12        // Se instancia la BD en modo lectura y se crea la SELECT.
13        db = amigosDBHelper.readableDatabase
14        val cursor: Cursor = db.rawQuery(
15            "SELECT * FROM ${MyDBOpenHelper.TABLA_AMIGOS};", null)
16        // Se crea el adaptador con el resultado del cursor.
17        val myRecyclerViewAdapter = MyRecyclerViewAdapter()
18        myRecyclerViewAdapter.MyRecyclerViewAdapter(this, cursor)
```

```

19     // Montamos el RecyclerView.
20     binding.myRecyclerview.setHasFixedSize(true)
21     binding.myRecyclerview.setLayoutManager(this)
22     binding.myRecyclerview.setAdapter = myRecyclerViewAdapter
23 }
24
25     override fun onDestroy() {
26         super.onDestroy()
27         // Cerramos la conexión al terminar la activity.
28         Log.d("onDestroy", "Cerramos la conexión")
29         db.close()
30     }
31 }
```

La creación del adaptador y del *RecyclerView* sigue siendo igual, no cambia nada. Resaltar al necesidad de cerrar la conexión de la base de datos en el método `onDestroy()`, ya que si se cierra al final del método `onCreate()` el cursor se destruiría y la aplicación fallaría. El resultado de todo esto lo puedes ver en la siguiente figura.

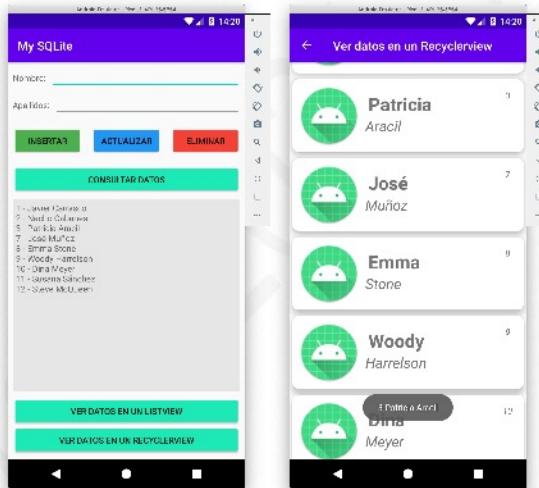


Figura 6

Ejercicios propuestos

9.1. Crea una aplicación utilizando **Android Studio + Kotlin**, similar a la vista en este capítulo, que permita insertar, modificar, eliminar y consultar sobre una base de datos que almacene información sobre animales. Puedes hacer uso de las imágenes de los recursos y almacenar la información que consideres necesaria.

9.5. Database Inspector

Desde la versión 4.1. de Android Studio, se dispone de **Database Inspector**⁴ 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. 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 > Database Inspector**, o la opción que encontrarás en la barra inferior.

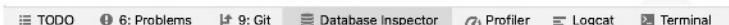


Figura 7

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.

Database Inspector			
Xiaomi M2004J19C > es.javiercarrasco.sqlite			
Databases			
personas.db	amigos	New Query [3]	Live updates
	_id : INTEGER	_id	nombre
	nombre : TEXT	1	Antonio
	apellidos : TEXT	2	Javier
		3	Patricia
		4	Nacho
			Muñoz
			Carrasco
			Aracil
			Cabanes

Figura 8

Como puedes ver en la imagen, 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 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.

Database Inspector			
Xiaomi M2004J19C > es.javiercarrasco.sqlite			
Databases			
personas.db			
amigos			
	_id : INTEGER	_id	nombre
	nombre : TEXT	1	Javier
	apellidos : TEXT		Carrasco
			Results are read-only

Figura 9