

Apuntes

Descargar estos apuntes [pdf](#) y [html](#)

Tema 11.1. Persistencia de Datos I SQLite

Índice

1. [Acceso a bases de datos locales. SQLite](#)
 1. [Introducción](#)
 2. [Creación y apertura de la BD](#)
 3. [Acceso y modificación de los datos en la BD](#)
 1. [Acceso a la BD mediante sentencias sql](#)
 2. [Acceso a la BD mediante SQLiteDatabase](#)
 3. [Recuperación de datos con rawQuery](#)
 4. [Recuperación de datos con query](#)
 4. [Uso de SimpleCursorAdapter](#)
 1. [Cursores con RecyclerView](#)

Acceso a bases de datos locales. SQLite

Introducción

La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados: la base de datos **SQLite** y **Content Providers** (lo trabajaremos en temas posteriores).

Vamos a centrarnos en **SQLite**, aunque no entraremos ni en el diseño de BBDD relacionales ni en el uso avanzado de la BBDD. Para conocer toda la funcionalidad de SQLite se recomienda usar la documentación oficial.


Para el acceso a las bases de datos tendremos tres clases que tenemos que conocer. La clase **SQLiteOpenHelper**, que encapsula todas las funciones de creación de la base de datos y versiones de la misma. La clase **SQLiteDatabase**, que incorpora la gestión de tablas y datos. Y por último la clase **Cursor**, que usaremos para movernos por el recordset que nos devuelve una consulta *SELECT*.

Creación y apertura de la BD

El método recomendado para la creación de la base de datos es extender la clase `SQLiteOpenHelper` y sobrescribir los métodos `onCreate` y `onUpgrade`. El primer método se utiliza para crear la base de datos por primera vez y el segundo para actualizaciones de la misma. Si la base de datos ya existe y su versión actual coincide con la solicitada, se realizará la conexión a ella.

Para ejecutar la sentencia SQL utilizaremos el método `execSQL` proporcionado por la API para SQLite de Android.

```
internal inner class BDClientes(  
    context: Context?,  
    name: String?,  
    factory: CursorFactory?,  
    version: Int) : SQLiteOpenHelper(context, name, factory, version)  
{  
    var sentencia =  
        "create table if not exists clientes" +  
        "(dni VARCHAR PRIMARY KEY NOT NULL, nombre TEXT, apellidos TEXT);"  
    override fun onCreate(sqliteDatabase: SQLiteDatabase) {  
        sqliteDatabase.execSQL(sentencia)  
    }  
    override fun onUpgrade(sqliteDatabase: SQLiteDatabase, i: Int, i2: Int) {}  
}
```

 **Nota:** Si la base de datos existe pero su versión actual es anterior a la solicitada, se llamará automáticamente al método `onUpgrade()`. Si la base de datos no existe se llama automáticamente a `onCreate()`.

Acceso y modificación de los datos en la BD

Una vez creada la clase, instanciaremos un objeto del tipo creado y usaremos uno de los dos siguientes métodos para tener acceso a la gestión de datos:

`getWritableDatabase()` para acceso de escritura y `getReadableDatabase()` para acceso de solo lectura. En ambos casos se devolverá un objeto de la clase `SQLiteDatabase` para el acceso a los datos.


Para realizar una codificación limpia y reutilizable, lo mejor es crear una clase con todos los métodos que necesitemos para trabajar con la Base de Datos, además es útil tener la clase derivada de `SQLiteOpenHelper` interna a esta.

```
class BDAAdapter(context: Context?)
{
    private var clientes: BDClientes

    init {
        clientes = BDClientes(context, "BDClientes", null, 1)
    }

    ...

    internal inner class BDClientes(
        context: Context?,
        name: String?,
        factory: CursorFactory?,
        version: Int) : SQLiteOpenHelper(context, name, factory, version)
    {
        var sentencia =
            "create table if not exists clientes" +
            "(dni VARCHAR PRIMARY KEY NOT NULL, nombre TEXT, apellidos TEXT);"
        override fun onCreate(sqliteDatabase: SQLiteDatabase) {
            sqliteDatabase.execSQL(sentencia)
        }
        override fun onUpgrade(sqliteDatabase: SQLiteDatabase, i: Int, i2: Int)
    }
}
```

 **Nota:** Como podemos ver en el código, creamos un objeto del tipo de `BDClientes` con el que vamos a trabajar llamando al constructor `BDClientes()`, a este método le pasamos el contexto, el nombre de la BBDD, un objeto cursor (con valor null) y la versión de la BD que necesitamos.

Para poder usar los métodos creados en la clase, tan solo instanciaremos un objeto de esta pasándole el contexto (de tipo `activity`). Con esta instancia podremos llamar a

cualquiera de los métodos que crearemos en la clase de utilidad.

```
var bdAdapter = BDAdapter(this)
bdAdapter.insertarDatos()
```

Acceso a la BD mediante sentencias sql

El acceso a la BBDD se hace en dos fases, primero se consigue un objeto del tipo **SQLiteDatabase** y posteriormente usamos sus funciones para gestionar los datos. Si hemos abierto correctamente la base de datos entonces podremos empezar con las inserciones, actualizaciones, borrado, etc. No deberemos olvidar cerrar el acceso a la BD, siempre y cuando no se esté usando un cursor.

*Por ejemplo, vamos a insertar 10 registros en nuestra tabla clientes. Para ello podremos crear un método en el **BDAdapter** parecido al siguiente:*

```
fun insertarDatos() {
    var dbClientes = clientes.writableDatabase
    if (dbClientes != null) {
        for (i in 0..9) {
            val sentencia = "INSERT INTO Clientes (dni, nombre, apellidos) V
                            " ('" + i + "', 'nombre" + i + "', 'apellido" + i + "');"
            dbClientes.execSQL(sentencia)
        }
        dbClientes.close()
    }
}
```

👉 Vale, ¿y ahora qué? ¿Dónde está la base de datos que acabamos de crear? ¿Cómo podemos comprobar que todo ha ido bien y que los registros se han insertado correctamente?, en primer lugar veamos dónde se ha creado nuestra base de datos. Todas las bases de datos SQLite creadas por aplicaciones Android se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón: **/data/data/paquete.java.de.la.aplicacion/databases/nombre_base_datos**

En el caso de nuestro ejemplo, la base de datos se almacenaría por tanto en la ruta siguiente: **/data/data/ejemplo.basedatos/databases/DBClientes**

Para comprobar que los datos se han grabado podemos acceder de forma remota al emulador a través de su consola de comandos (shell). Pero es mas sencillo abrir el archivo creado con un bloc de notas y echar un vistazo a su contenido.

Acceso a la BD mediante SQLiteDatabase

La API de Android nos ofrece dos métodos para acceder a los datos. El primero de ellos ya lo hemos visto, se trata de ejecutar sentencias SQL a través de **execSql**. Este método tiene como parámetro una cadena con cualquier instrucción SQL válida. La otra forma es utilizar los métodos específicos **insert()**, **update()** y **delete()** de la clase **SQLiteDatabase**.

Veamos a continuación cada uno de estos métodos.

- **insert()** : recibe tres parámetros `insert(table, nullColumnHack, values)` , el primero es el nombre de la tabla, el segundo se utiliza en caso de que necesitemos insertar valores nulos en la tabla "nullColumnHack" en este caso lo dejaremos pasar ya que no lo vamos a usar y por lo tanto lo ponemos a null y el tercero son los valores del registro a insertar. Los valores a insertar los pasaremos a su vez como elementos de una colección de tipo **ContentValues** . Estos elementos se almacenan como parejas **clave-valor**, donde la clave será el nombre de cada campo y el valor será el dato que se va a insertar.
- **update()** : Prácticamente es igual que el anterior método pero con la excepción de que aquí estamos usando el método `update(table, values, whereClause, whereArgs)` para actualizar/modificar registros de nuestra tabla. Este método nos pide el nombre de la tabla "table", los valores a modificar/actualizar "values" (ContentValues), una condición WHERE "whereClause" que nos sirve para indicarle que valor queremos que actualice y como último parámetro "whereArgs" podemos pasarle los valores nuevos a insertar, en este caso no lo vamos a necesitar por lo tanto lo ponemos a null.
- **delete()** : el método `delete(table, whereClause, whereArgs)` nos pide el nombre de la tabla "table", el registro a borrar "whereClause" que tomaremos como referencia su id y como último parámetro "whereArgs" los valores a borrar.

Insertando con ContentValues

```
val dbClientes = clientes.writableDatabase
val valores = ContentValues()
valores.put("nombre", "Xavi")
valores.put("dni", "22222111")
valores.put("apellidos", "Perez Rico")
dbClientes.update("clientes", valores, "dni=4", null)
dbClientes.close()
```

Borrando con ContentValues

```
val dbClientes = clientes.writableDatabase
if (dbClientes != null) {
    val valores = ContentValues()
    valores.put("nombre", cliente.nombre)
    valores.put("dni", cliente.dni)
    valores.put("apellidos", cliente.apellidos)
    dbClientes.insert("clientes", null, valores)
    dbClientes.close()
}
```


Modificando con ContentValues

```
val dbClientes = clientes.writableDatabase
val arg = arrayOf("1")
dbClientes.delete("clientes", "dni=?", arg)
dbClientes.close()
```

Los valores que hemos dejado anteriormente como null son realmente argumentos que podemos utilizar en la sentencia SQL. Veámoslo con un ejemplo:

```
val dbClientes = clientes.writableDatabase
val valores = ContentValues()
valores.put("nombre", "Carla")
val arg = arrayOf("6", "7")
dbClientes.update("clientes", valores, "dni=? OR dni=?", arg)
dbClientes.close()
```

Donde las `?` indican los emplazamientos de los argumentos.

 **Crea un ejercicio EjercicioResueltoBD en el que pruebes el código visto anteriormente: Creación de la BD, insertar elementos con SQL y modificar, eliminar e insertar mediante ContentValues.**

Recuperación de datos con rawQuery

Existen dos formas de recuperar información (SELECT de la base de datos), aunque ambas se apoyan en el concepto de **cursor**, que es en definitiva el objeto que recoge los resultados de la consulta. Vamos a completar el ejercicio anterior, para que nos permita ir insertando registros en la BD y visualizándolos en un listView.

La primera forma de obtener datos es utilizar el método `rawQuery()` de la clase `SQLiteDatabase`. Este método recibe como parámetro la sentencia SQL, donde se indican los campos a recuperar y los criterios de selección.

EjemploBD	
dni	
nombre	
apellidos	
<div>añadir</div> <div>mostrar</div>	
nombre5	apellido5
5	
nombre0	apellido0
0	
nombre3	apellido3
3	
nombre1	apellido1
1	

```
fun seleccionarDatosSelect(sentencia: String?): Boolean {  
    val listaCliente: ArrayList<Clientes>?  
    val dbClientes = clientes.readableDatabase  
    if (dbClientes != null) {  
        val cursor: Cursor = dbClientes.rawQuery(sentencia, null)  
        listaCliente = getClientes(cursor)  
        dbClientes.close()  
        return if (listaCliente == null) false else true  
    }  
    return false  
}
```

Evidentemente, ahora tendremos que recorrer el cursor y visualizar los datos devueltos. Para ello se dispone de los métodos `moveToFirst()`, `moveToNext()`, `moveToLast()`, `moveToPrevious()`, `isFirst()` y `isLast()`.

Existen métodos específicos para la recuperación de datos `getXXX(indice)`, donde XXX indica el tipo de dato (String, Blob, Float,...) y el parámetro índice permite recuperar la columna indicada en el mismo, teniendo en cuenta que comienza en 0.

El método `getClientes` (implementado por nosotros) es el que nos permite leer los datos existentes en la BD y llevarlos al `arrayList`:


```

fun getClientes(cursor: Cursor): ArrayList<Clientes>? {
    val clientes: ArrayList<Clientes>
    var cliente: Clientes
    cursor.moveToFirst()
    if (!cursor.isAfterLast()) {
        clientes = ArrayList()
        while (!cursor.isAfterLast()) {
            cliente =
                Clientes(cursor.getString(0), cursor.getString(1), cursor.getStrin
            clientes.add(cliente)
            cursor.moveToNext()
        }
        return clientes
    }
    return null
}

```

Recuperación de datos con query

La segunda forma de recuperar información de la BD es utilizar el método `query()`. Recibe como parámetros el nombre de la tabla, un string con los campos a recuperar, un string donde especificar las condiciones del WHERE, otro para los argumentos si los hubiera, otro para el GROUP BY, otro para HAVING y finalmente otro para ORDER BY.


```

fun seleccionarDatosCodigo(
    columnas: Array<String?>?,
    where: String?,
    valores: Array<String?>?,
    orderBy: String?): ArrayList<Clientes>?
{
    var listaCliente: ArrayList<Clientes>? = ArrayList()
    val dbClientes = clientes.readableDatabase
    if (dbClientes != null) {
        val cursor: Cursor =
            dbClientes.query("clientes", columnas, where, valores, null, null,
        listaCliente = getClientes(cursor)
        dbClientes.close()
        return listaCliente
    }
    return null
}

```

Donde la llamada al método podría ser la siguiente:

```
val listaCliente = bdAdapter.seleccionarDatosCodigo(  
    arrayOf("dni", "nombre", "apellidos"),  
    null,  
    null,  
    "apellidos")
```

 **Crea un ejercicio ejemploBD en el que pruebes el código visto anteriormente: Recuperación de los datos con rawQuery y con query. Para ello crea la aplicación como en la imagen, de forma que con un botón se guarde la información en la BD y con el otro se extraiga y se muestre en una lista o en un recycler. Si vais a utilizar una lista como se muestra en la imagen, debereis crear una clase adaptador como la de la siguiente imagen y asociarla a la lista con el setAdapter:**

```
binding.listView.setAdapter(  
    AdaptadorClientes(  
        this@MainActivity,  
        R.layout.list_layout,  
        listaCliente!! ))
```

Clase Adaptador para un ListView:

```
class AdaptadorClientes(var activitycontext: Activity,  
    resource: Int,  
    objects: ArrayList<Clientes>  
    ) :  
    ArrayAdapter<Any>(activitycontext, resource, objects as List<Any>) {  
    var objects: ArrayList<Clientes>  
    override fun getView(position: Int,  
        convertView: View?,  
        parent: ViewGroup): View {  
        var vista: View? = convertView  
        if (vista == null) {  
            val inflater = activitycontext.layoutInflater  
            vista = inflater.inflate(R.layout.list_layout, null)  
            (vista?.findViewById(R.id.apellidolist) as TextView)  
                .setText(objects[position].apellidos)  
            (vista?.findViewById(R.id.nombrelist) as TextView)  
                .setText(objects[position].nombre)  
            (vista?.findViewById(R.id.dnिलist) as TextView)  
                .setText(objects[position].dni)  
        }  
        return vista  
    }  
  
    init {  
        this.objects = objects  
    }  
}
```

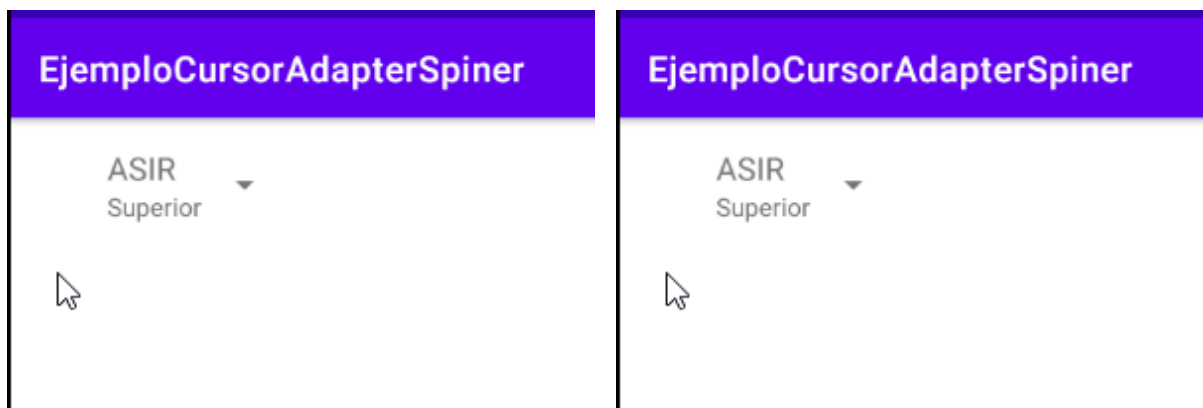
 **Ejercicio Propuesto Recorrer BD**

Uso de SimpleCursorAdapter

Se utiliza para mapear las columnas de un cursor abierto en una base de datos a los diferentes elementos visuales contenidos en el control de selección. Así se evita el volcado de todos los elementos descargados de la BD a una colección, con el ahorro de memoria que conlleva.

Nos encontramos con dos casos: cuando usamos **Spinner** o **ListView** sencillos el adaptador que gestiona este volcado ya está creado, si usamos **RecyclerView** tendremos que crear y gestionar el adaptador.

Vamos a programar un ejemplo para practicar este concepto, será una sencilla actividad con un **Spinner** que mostrará los ciclos informáticos con la categoría a la que pertenecen:



Para ello nos creamos la base de datos e insertamos elementos como ya hemos visto hasta el momento:

```

class DBAdapter(context: Context) {
    private var ohCategoria: OHCategoria
    init {
        ohCategoria = OHCategoria(context, "BBDCategorias", null, 1)
    }
    fun insertarDatosCodigo() {
        val sqLiteDatabase = ohCategoria.writableDatabase
        if (sqLiteDatabase != null) {
            val valores = ContentValues()
            valores.put("nombre", "ASIR")
            valores.put("cate", "Superior")
            valores.put("idcategoria", 1)
            sqLiteDatabase.insert("categoria", null, valores)
            valores.put("nombre", "DAM")
            valores.put("cate", "Superior")
            valores.put("idcategoria", 2)
            sqLiteDatabase.insert("categoria", null, valores)
            valores.put("nombre", "SMR")
            valores.put("cate", "Medio")
            valores.put("idcategoria", 3)
            sqLiteDatabase.insert("categoria", null, valores)
            sqLiteDatabase.close()
        }
    }
    fun leerDatos():Cursor?
    {
        val sqLiteDatabase = ohCategoria.readableDatabase
        if (sqLiteDatabase != null) {
            return sqLiteDatabase.rawQuery(
                "select idcategoria as _id, nombre, cate from categoria",
                null )
        }
        return null
    }

    inner class OHCategoria(
        context: Context?, name: String?,
        factory: SQLiteDatabase.CursorFactory?,
        version: Int):QLiteOpenHelper(context, name, factory, version)
    {
        var cadena =
            "create table if not exists categoria(idcategoria
            INTEGER PRIMARY KEY NOT NULL, nombre TEXT, cate TEXT);"
        override fun onCreate(db: SQLiteDatabase) {
            db.execSQL(cadena)
        }
        override fun onUpgrade(db: SQLiteDatabase,
                                oldVersion: Int, newVersion: Int) {}
    }
}

```

Posteriormente pasamos a crear el `cursorAdapter`, pero para ello vamos a usar un `Spinner` para mostrar la información, lo incluimos en el `layout` principal.

```
1  override fun onCreate(savedInstanceState: Bundle?) {  
2      super.onCreate(savedInstanceState)  
3      val from = arrayOf("nombre", "cate")  
4      val to = intArrayOf(R.id.ciclo, R.id.cate)  
5      setContentView(R.layout.activity_my)  
6      val dbAdapter=DBAdapter(this)  
7      dbAdapter.insertarDatosCodigo()  
8      val desplegable = findViewById<Spinner>(R.id.spinner)  
9      val cursor=dbAdapter.leerDatos()  
10     val mAdapter = SimpleCursorAdapter(this, R.layout.spinner_layout,  
11                                     cursor, from, to, 0x0)  
12     desplegable.setAdapter(mAdapter)  
13 }
```

✦ **Nota:** En la **Línea 9** se devuelve el cursor con la selección de los datos de la BD (cuidado!! no se deberá cerrar el cursor). **Línea 19** para el funcionamiento del `SimpleCursorAdapter`, es necesario indicar en su definición los siguientes elementos: el contexto, el layout sobre el que se va a definir la vista, el cursor que va a ser origen de los datos, lista con el nombre de las columnas que se quiere enlazar, una lista con los id de las vistas donde se van a mostrar los campos (el orden es importante) y un flag (0 por defecto)

✍ **Prueba la aplicación del ejemplo anterior**

Cursores con RecyclerView

Como era de esperar, no se puede aplicar directamente un adaptador de tipo

`SimpleCursorAdapter` a un tipo `RecyclerView`. Si queremos aprovechar los beneficios de los cursores en los recylcers, tendremos que fabricarlos a medida. Los pasos son los siguientes:


1. Crear una clase **Abstracta** base que derive de **RecyclerView.Adapter** y en la que implementaremos los métodos sobrescritos derivados de `RecyclerView.Adapter`:

```
1  abstract class CursorRecyclerViewAdapter(cursor: Cursor) :
2  RecyclerView.Adapter<RecyclerView.ViewHolder>() {
3  var mCursor: Cursor
4  override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
5                                position: Int)
6  {
7      checkNotNull(mCursor) { "ERROR, cursos vacio" }
8      check(mCursor.moveToPosition(position)) { "ERROR, no se puede" +
9          " encontrar la posicion: $position" }
10     onBindViewHolder(holder, mCursor)
11 }
12 abstract fun onBindViewHolder(holder: RecyclerView.ViewHolder,
13                                cursor: Cursor)
14 override fun getItemCount(): Int
15 {
16     return if (mCursor != null)    mCursor.getCount()
17         else 0
18 }
19 init {
20     mCursor = cursor
21 }
22 }
```




🚩 **Nota:** Al heredar de `RecyclerView.Adapter` habrá que implementar los métodos `onBindViewHolder()` al que le llega la posición de la línea del recycler que está activa en ese momento y `getItemCount()` que tendrá que devolver los elementos que tiene el cursor **Línea 16**, le llega en el constructor. En el método `onBindViewHolder` se comprobará que el cursor no está vacío y que se puede acceder a la posición que llega como parámetro **Línea 8**, y en ese caso se llamará al método `onBindViewHolder(holder: RecyclerView.ViewHolder, cursor: Cursor)` abstracto que debemos crear en esta clase **Línea 12**, pero esta vez con un `Cursor` como parámetro.

2. Ahora tendremos que crear nuestra clase RecyclerView derivada de la anterior.

```
1  class RecyclerViewAdapter(c: Cursor) : CursorRecyclerViewAdapter(c) {
2      override fun onCreateViewHolder(parent: ViewGroup,
3                                     viewType: Int): RecyclerView.ViewHolder
4      {
5          val v: View = LayoutInflater.from(parent.context)
6              .inflate(R.layout.recycler_layout, parent, false)
7          return SimpleViewHolder(v)
8      }
9      override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
10                                  cursor: Cursor)
11      {
12          (holder as SimpleViewHolder).bind(cursor)
13      }
14      internal inner class SimpleViewHolder(itemView: View) :
15          RecyclerView.ViewHolder(itemView)
16      {
17          var nombre: TextView
18          var cate: TextView
19          var imagen: ImageView
20          fun bind(dato: Cursor) {
21              nombre.setText(dato.getString(0))
22              cate.setText(dato.getString(1))
23              val theImage: Bitmap = MainActivity.
24                  convertirStringBitmap(dato.getString(3))
25              imagen.setImageBitmap(theImage)
26          }
27          init {
28              nombre = itemView.findViewById(R.id.ciclo)
29              cate = itemView.findViewById(R.id.cate)
30              imagen = itemView.findViewById(R.id.imagen) as ImageView
31          }
32      }
33  }
```

 **Nota:** A esta clase le llegará el cursor por el constructor que a su vez será pasado a la clase padre. Al heredar de la clase abstracta, tendremos que implementar su método abstracto y será ahí donde aprovecharemos que el cursor se ha posicionado en el elemento correcto de la BD, para llamar al método **bind(cursor)** **Línea 12** del **Holder**, con ese estado del cursor. En el **Holder** trataremos los elementos que saquemos del cursor de forma correcta.

3. Como esta App es un poco diferente a la anterior, se han incluido imágenes para dar más funcionalidad, hay dos métodos que nos permiten pasar imágenes de Bitmap a String y viceversa para poderlas guardar en la BD a través del cursor (En la BD se guardarán como Blob). Están localizados en MyActivity y son estáticos para poder acceder a ellos sin necesidad de objeto.

EjemploCursorAdapter	
	ASIR Superior
	DAM Superior
	SMR Medio

```
companion object
{
    fun convertirStringBitmap(imagen: String?): Bitmap {
        val decodedString: ByteArray = Base64.decode(imagen, Base64.DEFAULT)
        return BitmapFactory.decodeByteArray(decodedString, 0, decodedString.size)
    }

    fun ConvertirImagenString(bitmap: Bitmap): String {
        val stream = ByteArrayOutputStream()
        bitmap.compress(Bitmap.CompressFormat.PNG, 90, stream)
        val byte_arr: ByteArray = stream.toByteArray()
        return Base64.encodeToString(byte_arr, Base64.DEFAULT)
    }
}
```

 **Prueba la aplicación del ejemplo anterior**

 **Ejercicio propuesto cursor con recycler**