Bases de Datos SQLite en Android

Primeros pasos con SQLite

La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados:

- *Bases de Datos SQLite*
- *Content Providers* .

SQLite es un motor de bases de datos muy popular en la actualidad por ofrecer características tan interesantes como su pequeño tamaño, no necesitar servidor, precisar poca configuración, ser transaccional y por supuesto ser de **código libre**.

Android incorpora de serie todas las herramientas necesarias para la creación y gestión de bases de datos SQLite, y entre ellas una **completa API** para llevar a cabo de manera sencilla todas las tareas necesarias.

En Android, la forma típica para crear, actualizar, y conectar con una base de datos SQLite será a través de una clase auxiliar llamada `SQLiteOpenHelper`, o para ser más exactos, de una clase propia que derive de ella y que debemos personalizar para adaptarnos a las necesidades concretas de nuestra aplicación.

La clase `SQLiteOpenHelper` tiene tan sólo **un constructor**, que normalmente no necesitaremos sobrescribir, y **dos métodos abstractos**, `onCreate()` y `onUpgrade()`, que deberemos personalizar con el código necesario para crear nuestra base de datos y para actualizar su estructura respectivamente.

Como ejemplo, vamos a crear una base de datos muy sencilla llamada **`BDUsuarios`**, con una sola tabla llamada **`Usuarios`** que contendrá sólo tres campos: `idUsuario`, `nombre` e `email`. Para ellos, vamos a crear una clase derivada de `SQLiteOpenHelper` que llamaremos `UsuariosSQLiteHelper`, donde sobrescribiremos los métodos `onCreate()` y `onUpgrade()` para adaptarlos a la estructura de datos indicada:

```
package com.example.ej_basesdatos;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase.CursorFactory;

public class UsuariosSQLiteHelper extends SQLiteOpenHelper {

    //Sentencia SQL para crear la tabla de Usuarios
    String sqlCreate =
           "CREATE TABLE Usuarios (idUsuario INTEGER PRIMARY KEY," +
                   " nombre TEXT, email TEXT)";

    public UsuariosSQLiteHelper(Context contexto, String nombre,
                                CursorFactory factory, int version) {
        super(contexto, nombre, factory, version);
    }
```

```
    @Override
    public void onCreate(SQLiteDatabase db) {
        //Se ejecuta la sentencia SQL de creación de la tabla
        db.execSQL(sqlCreate);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int i, int i1) {
        /*NOTA: Por simplicidad del ejemplo aquí utilizamos directamente
         la opción de eliminar la tabla anterior y crearla de nuevo vacia
         con el nuevo formato.
         Sin embargo lo normal será que haya que migrar datos de la
         tabla antigua a la nueva, por lo que este método deberia
         ser más elaborado.
        */

        //Se elimina la versión anterior de la tabla
        db.execSQL("DROP TABLE IF EXISTS Usuarios");

        //Se crea la nueva version de la tabla
        db.execSQL(sqlCreate);
    }
}
```

Lo primero que hacemos es definir una variable llamado `sqlCreate` donde almacenamos la sentencia SQL para crear una tabla llamada `Usuarios` con los campos alfanuméricos *idUsuario*, *nombre* y *email*.

El método `onCreate()` será ejecutado automáticamente por nuestra clase `UsuariosSQLiteHelper` cuando sea necesaria la creación de la base de datos, es decir, cuando aún no exista. Las tareas típicas que deben hacerse en este método serán la creación de todas las tablas necesarias y la inserción de los datos iniciales si son necesarios. En nuestro caso y por el momento, sólo vamos a crear la tabla `Usuarios` descrita anteriormente. Para la creación de la tabla utilizaremos la sentencia SQL ya definida y la ejecutaremos contra la base de datos utilizando el método más sencillo de los disponibles en la API de SQLite proporcionada por Android, llamado `execSQL()`. Este método se limita a ejecutar directamente el código SQL que le pasemos como parámetro.

Por su parte, el método `onUpgrade()` se lanzará automáticamente cuando sea necesaria una actualización de la estructura de la base de datos o una conversión de los datos. Un ejemplo práctico: imaginemos que publicamos una aplicación que utiliza una tabla con los campos idUsuario, nombre e email (llamémoslo versión 1 de la base de datos). Más adelante, ampliamos la funcionalidad de nuestra aplicación y necesitamos que la tabla también incluya un campo adicional como por ejemplo la edad del usuario (versión 2 de nuestra base de datos). Pues bien, para que todo funcione correctamente, la primera vez que ejecutemos la versión ampliada de la aplicación necesitaremos modificar la estructura de la tabla `Usuarios` para añadir el nuevo campo `edad`. Pues este tipo de cosas son las que se encargará de hacer automáticamente el método `onUpgrade()` cuando intentemos abrir una versión concreta de la base de datos que aún no exista. Para ello, como parámetros recibe la versión actual de la base de datos en el sistema, y la nueva versión a la que se quiere convertir. En función de esta pareja de datos necesitaremos realizar unas acciones u otras. En

nuestro caso de ejemplo optamos por la opción más sencilla: borrar la tabla actual y volver a crearla con la nueva estructura, pero como se indica en los comentarios del código, lo habitual será que necesitemos algo más de lógica para convertir la base de datos de una versión a otra y por supuesto para conservar los datos registrados hasta el momento.

Una vez definida nuestra clase *helper*, la apertura de la base de datos desde nuestra aplicación resulta ser algo de lo más sencillo. Lo primero será crear un objeto de la clase `UsuariosSQLiteHelper` al que pasaremos el contexto de la aplicación (en el ejemplo una referencia a la actividad principal), el nombre de la base de datos, un objeto `CursorFactory` que típicamente no será necesario (en ese caso pasaremos el valor `null`), y por último la versión de la base de datos que necesitamos. La simple creación de este objeto puede tener varios efectos:

- Si la base de datos ya existe y su versión actual coincide con la solicitada simplemente se realizará la conexión con ella.
- Si la base de datos existe pero su versión actual es anterior a la solicitada, se llamará automáticamente al método `onUpgrade()` para convertir la base de datos a la nueva versión y se conectará con la base de datos convertida.
- Si la base de datos no existe, se llamará automáticamente al método `onCreate()` para crearla y se conectará con la base de datos creada.

Una vez tenemos una referencia al objeto `UsuariosSQLiteHelper`, llamaremos a su método `getReadableDatabase()` o `getWritableDatabase()` para obtener una referencia a la base de datos, dependiendo si sólo necesitamos consultar los datos o también necesitamos realizar modificaciones, respectivamente.

Ahora que ya hemos conseguido una referencia a la base de datos (objeto de tipo `SQLiteDatabase`) ya podemos realizar todas las acciones que queramos sobre ella. Para nuestro ejemplo nos limitaremos a insertar 5 registros de prueba, utilizando para ello el método ya comentado `execSQL()` con las sentencias `INSERT` correspondientes. Por último cerramos la conexión con la base de datos llamando al método `close()`.

```
package com.example.ej_basesdatos;

import android.database.sqlite.SQLiteDatabase;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;


public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //Abrimos la base de datos "DBUsuarios" en modo de escritura
        UsuariosSQLiteHelper usdbh =
                new UsuariosSQLiteHelper(this, "DBUsuarios", null, 1);

        SQLiteDatabase db = usdbh.getWritableDatabase();

        //Si hemos abierto correctamente la base de datos
        if (db != null){
```

```
                //Insertamoa 5 usuarios de ejemplo
                for (int i =1; i<=5; i++){
                    //Generamos los datos
                    int codigo = i;
                    String nombre ="Usuario "+i;
                    String email = "usuario"+i+"@usuarios.com";

                    //Insertamos los datos en la tabla de Usuarios
                    db.execSQL("INSERT INTO Usuarios (idUsuario, nombre, email)" +
                          " VALUES ("+ codigo +" ,'"+ nombre +"','" + email +"')");
                }

                //Cerramos la base de datos
                db.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?

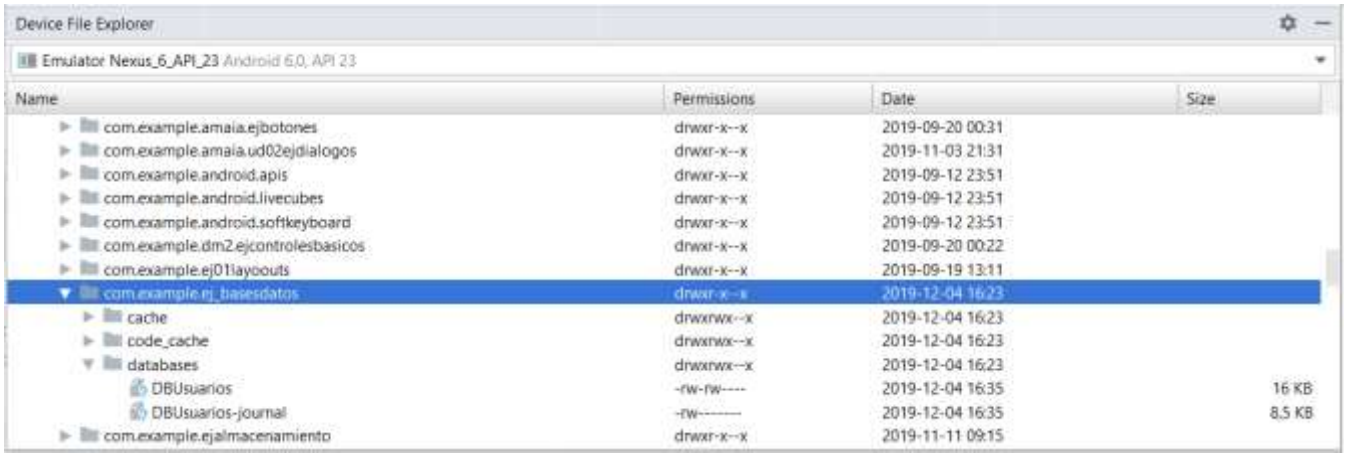
En primer lugar veamos dónde se ha creado nuestra base de datos. Todas las bases de datos SQLite creadas por aplicaciones Android utilizando este método se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón:

`/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/com.example.ej_basesdatos/databases/DBUsuarios`

Para comprobar esto podemos hacer lo siguiente. Una vez ejecutada por primera vez la aplicación de ejemplo sobre el emulador de Android (y por supuesto antes de cerrarlo) podemos ir al Device File Explorer (View → Tools Windows → Device File Explorer) y acceder al sistema de archivos del emulador, donde podremos buscar en la ruta indicada la base de datos. Podemos ver esto en la siguiente imagen:

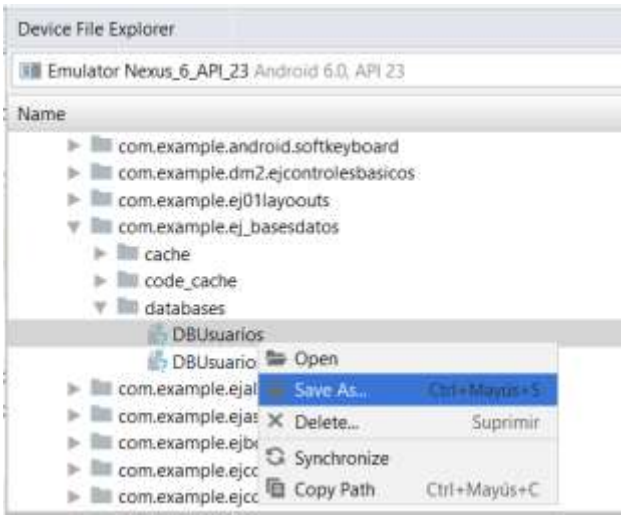


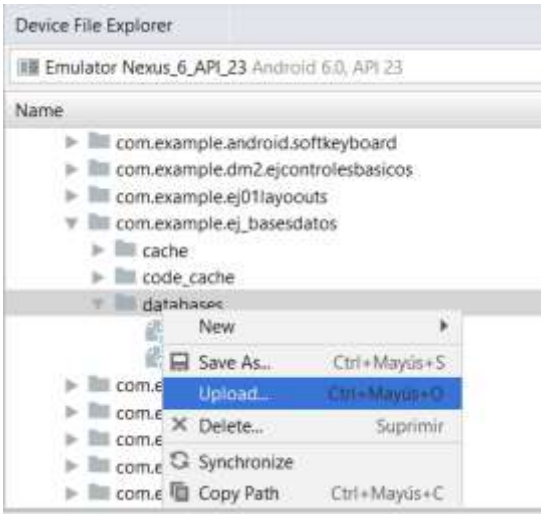
Con esto comprobamos al menos que el fichero de nuestra base de datos se ha creado en la ruta correcta. Ya sólo nos queda comprobar que tanto las tablas creadas como los datos insertados

también se han incluido correctamente en la base de datos. Para ello podemos recurrir a dos posibles métodos:

1. Transferir la base de datos a nuestro PC y consultarla con cualquier administrador de bases de datos SQLite.
2. Acceder directamente a la consola de comandos del emulador de Android y utilizar los comandos existentes para acceder y consultar la base de datos SQLite.

El primero de los métodos es sencillo. El fichero de la base de datos podemos guardarlo en nuestro PC haciendo click con el botón derecho del ratón, seleccionando la opción *Save As* y por último indicando/seleccionando donde queremos guardarlo.





Por otro lado, si lo que se quiere hacer es la operación contraria, copiar un fichero local al sistema de archivos del emulador, en el menú contextual de la carpeta, se dispone de la opción *Upload*, y para eliminar ficheros del emulador, la opción *Delete*.

Una vez descargado el fichero a nuestro sistema local, podemos utilizar cualquier administrador de SQLite para abrir y consultar la base de datos, por ejemplo SQLite Administrator (freeware).

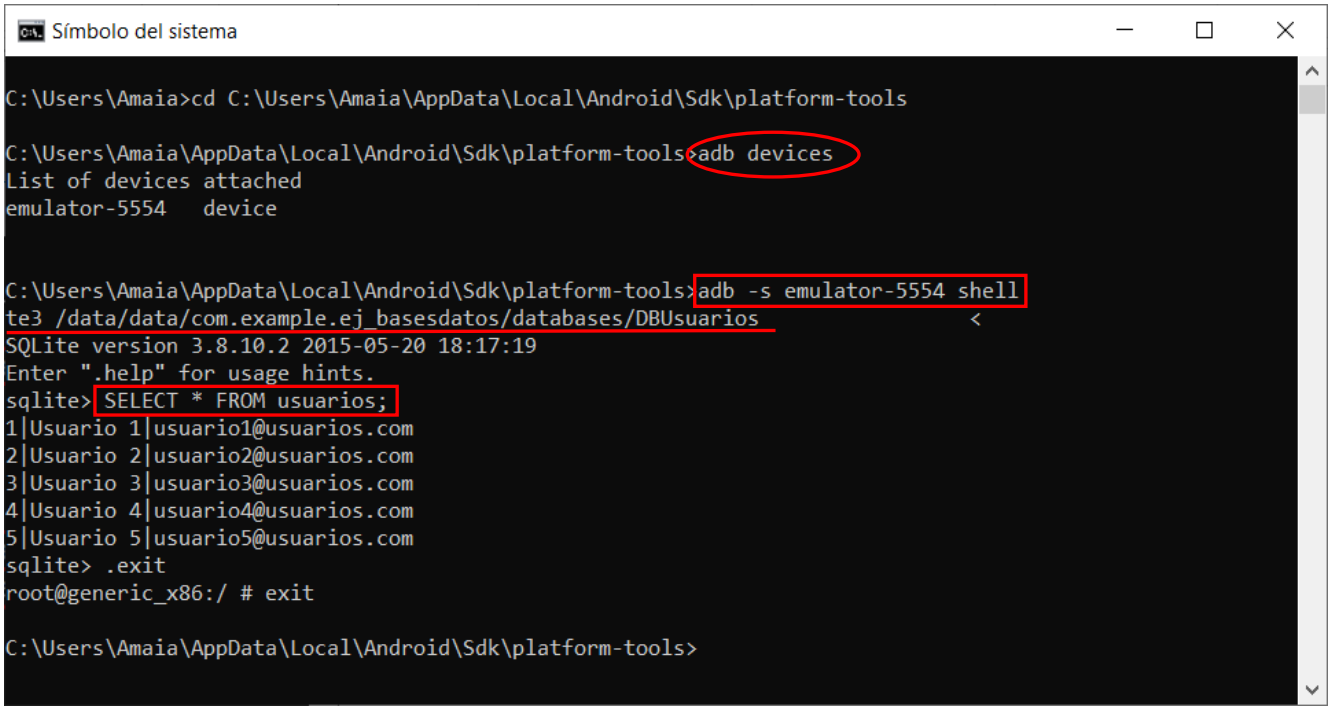
El segundo método utiliza una estrategia diferente. En vez de descargar la base de datos a nuestro sistema local, somos nosotros los que accedemos de forma remota al emulador a través de su consola de comandos (*shell*). Para ello, con el emulador de Android aún abierto, debemos abrir una consola de MS-DOS y utilizar la utilidad `adb.exe` (*Android Debug Bridge*) situada en la carpeta `platform-tools` del SDK de Android (en mi caso: `C:\Users\Amaia\AppData\Local\Android\Sdk\platform-tools\`). En primer lugar consultaremos los identificadores de todos los emuladores en ejecución mediante el comando "`adb devices`". Esto nos debe devolver una única instancia si sólo tenemos un emulador abierto, que en mi caso particular se llama "`emulator-5554`".

Tras conocer el identificador de nuestro emulador, vamos a acceder a su shell mediante el comando "`adb -s identificador-del-emulador shell`" (`adb -s emulator-5554 shell`). Una vez conectados, ya podemos acceder a nuestra base de datos utilizando el comando **sqlite3** pasándole la ruta del fichero, para nuestro ejemplo

"`sqlite3 /data/data/com.example.ej_basesdatos/databases/DBUsuarios`".

Si todo ha ido bien, debe aparecernos el *prompt* de SQLite "`sqlite>`", lo que nos indicará que ya

podemos escribir las consultas SQL necesarias sobre nuestra base de datos. Vamos a comprobar que existe la tabla `Usuarios` y que se han insertado los cinco registros de ejemplo. Para ello haremos la siguiente consulta: "`SELECT * FROM Usuarios;`". Si todo es correcto esta instrucción debe devolvernos los cinco usuarios existentes en la tabla. En la imagen siguiente se muestra todo el proceso descrito:



Con esto ya hemos comprobado que nuestra base de datos se ha creado correctamente, que se han insertado todos los registros de ejemplo y que todo funciona según se espera.

Insertar/Actualizar/Eliminar

La API de SQLite de Android proporciona dos alternativas para realizar operaciones sobre la base de datos que no devuelven resultados (entre ellas la inserción/actualización/eliminación de registros, pero también la creación de tablas, de índices, etc).

El primero de ellos, que hemos visto anteriormente, es el método `execSQL()` de la clase `SQLiteDatabase`. Este método permite ejecutar cualquier sentencia SQL sobre la base de datos, siempre que ésta no devuelva resultados. Para ello, simplemente aportaremos como parámetro de entrada de este método la cadena de texto correspondiente con la sentencia SQL. Cuando hemos creado la base de datos hemos visto algún ejemplo de esto para insertar los registros de prueba. Otros ejemplos podrían ser los siguientes:

- Insertar un registro

```
db.execSQL("INSERT INTO Usuarios (idUsuario, nombre, email)" +
        " VALUES ('7','Usuario7','usuario7@usuarios.com') ");
```

- Eliminar un registro

```
db.execSQL("DELETE FROM Usuarios WHERE idUsuario=7");
```

- Actualizar un registro

```
db.execSQL("UPDATE Usuarios SET nombre='usuarioNuevo' WHERE idUsuario=7");
```

La **segunda de las alternativas** disponibles en la API de Android es utilizar los métodos `insert()`, `update()` y `delete()` proporcionados también con la clase `SQLiteDatabase`. Estos métodos permiten realizar las tareas de inserción, actualización y eliminación de registros de una forma algo más paramétrica que `execSQL()`, separando tablas, valores y condiciones en parámetros independientes de estos métodos.

Empecemos por el método `insert()` para insertar nuevos registros en la base de datos. Este método recibe tres parámetros, el primero de ellos será el *nombre de la tabla*, el tercero serán *los valores del registro a insertar*, y el segundo lo obviaremos por el momento ya que tan sólo se hace necesario en casos muy puntuales (por ejemplo para poder insertar registros completamente vacíos), en cualquier otro caso pasaremos con valor `null` este segundo parámetro.

Los valores a insertar los pasaremos como elementos de una colección de tipo `ContentValues`. Esta colección es de tipo *diccionario*, donde almacenaremos parejas de *clave-valor*, donde la *clave* será el nombre de cada campo y el *valor* será el dato correspondiente a insertar en dicho campo. Vamos con un ejemplo:

```
//Creamos el registro a insertar como Objeto ContentValues
ContentValues nuevoRegistro = new ContentValues();
nuevoRegistro.put("idUsuario", "7");
nuevoRegistro.put("nombre", "Usuario 7");
nuevoRegistro.put("email", "usuario7@usuarios.com");

//Insertamos el registro en la base de datos
db.insert("Usuarios", null, nuevoRegistro);
```

Los métodos `update()` y `delete()` se utilizarán de forma muy parecida a ésta, con la salvedad de que recibirán un parámetro adicional con la condición *WHERE* de la sentencia SQL. Por ejemplo, para actualizar el nombre del usuario con código ‘7’ haríamos lo siguiente:

```
//Establecemos los campos-valores a actualizar
ContentValues valores = new ContentValues();
valores.put("nombre", "usuario Nuevo");

//Actualizamos el registro en la base de datos
db.update("Usuarios", valores, "idUsuario=7" , null);
```

Como podemos ver, como tercer parámetro del método `update()` pasamos directamente la condición del *UPDATE* tal como lo haríamos en la cláusula *WHERE* en una sentencia SQL normal.

El método `delete()` se utilizaría de forma análoga. Por ejemplo para eliminar el registro del usuario con código ‘7’ haríamos lo siguiente:

- Eliminamos el registro del usuario '7'

```
db.delete("Usuarios", "idUsuario=7", null);
```

Como vemos, volvemos a pasar como primer parámetro el nombre de la tabla y en segundo lugar la condición *WHERE*. Por supuesto, si no necesitáramos ninguna condición, podríamos dejar como valor `null` en este parámetro (lo que eliminaría todos los registros de la tabla).

Un último detalle sobre estos métodos. Tanto en el caso de `execSQL()` como en los casos de `update()` o `delete()` podemos utilizar argumentos dentro de las condiciones de la sentencia SQL.

Éstos no son más que partes *variables* de la sentencia SQL que aportaremos en un array de valores aparte, lo que nos evitará pasar por la situación típica en la que tenemos que construir una sentencia SQL concatenando cadenas de texto y variables para formar el comando SQL final. Estos argumentos SQL se indicarán con el símbolo '?', y los valores de dichos argumentos deben pasarse en el array en el mismo orden que aparecen en la sentencia SQL. Así, por ejemplo, podemos escribir instrucciones como la siguiente:

```
//Eliminar un registro con execSQL(), utilizando argumentos
String[] args = new String[]{"usuario1"};
db.execSQL("DELETE FROM Usuarios WHERE nombre=?", args);

//Actualizar dos registros con update(), utilizando argumentos
ContentValues valores = new ContentValues();
valores.put("nombre","usunuevo");

String[] args2 = new String[]{"usuario1", "usuario2"};
db.update("Usuarios", valores, "nombre=? OR nombre=?", args2);
```

Esta forma de pasar a la sentencia SQL determinados datos variables puede ayudarnos además a escribir código más limpio y evitar posibles errores.

Continuando con el ejemplo anterior, hemos añadido tres cuadros de texto para poder introducir el código, nombre e email de un usuario y tres botones para insertar, actualizar o eliminar dicha información.

Consultar/Recuperar registros

Por último vamos a describir la última de las tareas más importantes de tratamiento de datos que nos queda por ver, la selección y recuperación de datos.

De forma análoga a lo que hemos visto para las sentencias de modificación de datos, vamos a tener dos opciones principales para recuperar registros de una base de datos SQLite en Android. La primera de ellas utilizando directamente un comando de selección SQL, y como segunda opción utilizando un método específico donde *parametrizaremos* la consulta a la base de datos.

Para la primera opción utilizaremos el método `rawQuery()` de la clase `SQLiteDatabase`. Este método recibe directamente como parámetro un comando SQL completo, donde indicamos los campos a recuperar y los criterios de selección. El resultado de la consulta lo obtendremos en forma de cursor, que posteriormente podremos recorrer para procesar los registros recuperados. Sirva la siguiente consulta a modo de ejemplo:

```
Cursor c =db.rawQuery("SELECT idUsuario, nombre, email FROM Usuarios " +
        "WHERE nombre = 'Usuario 3'", null);
```

Como en el caso de los métodos de modificación de datos, también podemos añadir a este método una lista de argumentos variables que hayamos indicado en el comando SQL con el símbolo '?', por ejemplo así:

```
String[] args = new String[] {"Usuario 2"};
Cursor c = db.rawQuery("SELECT idUsuario, nombre, email FROM Usuarios " +
        "WHERE nombre=? ", args);
```

Como segunda opción para recuperar datos podemos utilizar el método `query()` de la clase `SQLiteDatabase`. Este método recibe varios parámetros: el nombre de la tabla, un array con los nombre de campos a recuperar, la cláusula *WHERE*, un array con los argumentos variables incluidos en el *WHERE* (si los hay, `null` en caso contrario), la cláusula *GROUP BY* si existe, la cláusula *HAVING* si existe, y por último la cláusula *ORDER BY* si existe. Opcionalmente, se puede incluir un parámetro más al final indicando el número máximo de registros que queremos que nos devuelva la consulta. Veamos el mismo ejemplo anterior utilizando el método `query()`:

```
String[] campos = new String[] {"idUsuario", "nombre", "email"};
String[] args = new String[] {"Usuario 3"};

Cursor c = db.query("Usuarios", campos, "nombre=?", args, null, null, null);
```

Como vemos, los resultados se devuelven nuevamente en un objeto `Cursor` que deberemos recorrer para procesar los datos obtenidos.

Para recorrer y manipular el cursor devuelto por cualquiera de los dos métodos mencionados tenemos a nuestra disposición varios métodos de la clase `Cursor`, entre los que destacamos dos de los dedicados a recorrer el cursor de forma secuencial y en orden natural:

- **`moveToFirst()`:** mueve el puntero del cursor al primer registro devuelto.
- **`moveToNext()`:** mueve el puntero del cursor al siguiente registro devuelto.

Los métodos `moveToFirst()` y `moveToNext()` devuelven TRUE en caso de haber realizado el movimiento correspondiente del puntero sin errores, es decir, siempre que exista un primer registro o un registro siguiente, respectivamente.

Una vez posicionados en cada registro, podremos utilizar cualquiera de los métodos `getXXX(índice_columna)` existentes para cada tipo de dato, para recuperar el dato de cada campo del registro actual del cursor. Así, si queremos recuperar por ejemplo la segunda columna del registro actual, y ésta contiene un campo alfanumérico, haremos la llamada `getString(1)` (los índices comienzan por 0 (cero), por lo que la segunda columna tiene índice 1), en caso de contener un dato de tipo real llamaríamos a `getDouble(1)`, y de forma análoga para todos los tipos de datos existentes.

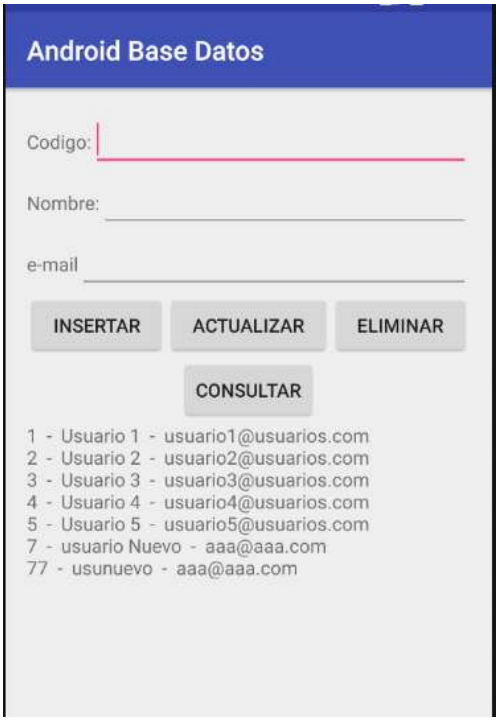
Con todo esto en cuenta, veamos cómo podríamos recorrer el cursor devuelto por el ejemplo anterior:

```
String[] campos = new String[] {"idUsuario", "nombre", "email"};
String[] args = new String[] {"Usuario 3"};

Cursor c = db.query("Usuarios", campos, "nombre=?", args, null, null, null);

//Recorremos los resultados para mostrarlos en pantalla
txtResultado.setText("");
```

```
//Nos aseguramos de que existe al menos un registro
if (c.moveToFirst()){
    //Recorremos el cursor hasta que no haya más registros.
    do {
        int codigo = c.getInt(0);
        String nombre =c.getString(1);
        String email = c.getString(2);
        txtResultado.append (codigo +" - "+nombre + " - " +email+"\n" );
    }while (c.moveToNext());
}
```

Además de los métodos comentados de la clase `Cursor` existen muchos otros más que nos pueden ser útiles en muchas ocasiones. Por ejemplo, `getCount()` dirá el número total de registros devueltos en el cursor, `getColumnName(i)` devuelve el nombre de la columna con índice `i`, `moveToPosition(i)` mueve el puntero del cursor al registro con índice `i`, etc. Podemos consultar la lista completa de métodos disponibles en la clase Cursor en la documentación oficial de Android.

Continuando con la aplicación anterior, añadimos un nuevo botón de consulta, que nos da la posibilidad de recuperar todos los registros de la tabla Usuarios.



ACTIVIDADES SQLite

1.- Realizar una Agenda de Contactos con SQLite

2.- Crea una base de Datos SQLite para administrar los libros de una biblioteca, en la que se puedan incluir, borrar, modificar y consultar libros (listar, buscar, …), asi como prestarlos a usuarios de la biblioteca.
