Persistencia de datos con SQLite

Una cuestión fundamental en cualquier aplicación informática (sea móvil, WEB o de escritorio, sea para Windows, Mac o Linux, sea para Android, Windows Phone o iOS), es la ***persistencia de datos***. ***La persistencia de datos*** quiere decir, básicamente, que los datos permanezcan registrados luego que la aplicación termine su ejecución, y el dispositivo en que se ejecuta se apague.

En particular, en una aplicación Android, la forma más simple, más difundida, y más eficiente, es usar ***SQLite***. El ***SQLite*** es un pequeño motorcito de base de datos, diseñado especialmente para dispositivos móviles, open source, y disponible para ***Android*** y ***iOS*** (Windows Phone se cortó solo e implementó su propio motor, mala onda).

Es muy simple de implementar y usar (ok, si, para mi alumnos todo es fácil, pero esa es otra cuestión), ya que tiene una excelente integración con el lenguaje ***Java*** para ***Android***. Lets go.

Todo lo que hagamos respecto al ***SQLite*** se realiza por medio de una clase nativa del lenguaje, llamada ***SQLiteOpenHelper***.

La forma de utilizarla es implementarla, es decir, crear una clase nuestra heredada de esa clase, y escribir los métodos obligatorios que la clase dispone. En este caso, esta clase dispone de solo dos métodos obligatorios: ***onCreate*** y ***onUpgrade***. Por lo tanto, vamos a crear una clase nueva, heredada de ***SQLiteOpenHelper***, y escribir esos métodos.

Para crear una clase nueva, ya dentro del ***Android Studio***, vamos a hacer ***File*** - ***New...*** - ***Java Class***. Vamos a ponerle un nombre a nuestra clase, como siempre, en castellano, e ilustrativo. En mi ejemplo, lo llamé ***baseTP3SQLiteHelper***, (confío en que ustedes NO van a usar ese mismo nombre, sino que se van a tomar el trabajo de pensar un nombre aún mejor, no? NO???)

La clase se crea con ese nombre, y vacía.

```
public class baseTP3SQLiteHelper {

}
```

Nosotros vamos a indicarle que esa clase hereda de ***SQLiteOpenHelper***, por lo que deberemos agregar el ***extends SQLiteOpenHelper***. Nuestra clase entonces quedará así:

```
public class baseTP3SQLiteHelper extends SQLiteOpenHelper {

}
```

Por qué ahora, que indicamos que nuestra clase no es abstracta, sino que hereda de ***SQLiteOpenHelper***, Android Studio se enojó?

Porque Android Studio sabe que las clases heredadas de ***SQLiteOpenHelper*** deben tener un constructor, deben implementar un ***onCreate***, y deben implementar un ***onUpgrade***, ambos métodos obligatorios.

Vamos a ir haciéndolos de a uno. Primero, el constructor. Como siempre, el constructor de una clase debe llamarse igual que la clase, ser ***Public***, y no tener tipo. En el caso de esta clase, debe incluir una llamada al constructor original de la clase de la cuál heredamos.

```
public baseTP3SQLiteHelper(Context contexto, String Nombre, SQLiteDatabase.CursorFactory fabrica, int Version) {
    super(contexto, Nombre, fabrica, Version);

}
```

Como vemos, ese constructor recibe cuatro parámetros, es decir que, cuando instanciemos esa clase, deberemos mandarle esos cuatro parámetros. Nos ocuparemos de ellos más luego. También vemos que el constructor tiene una invocación al constructor de la clase de la cuál heredamos, enviándole esos mismos parámetros. El constructor no será algo que usemos más en detalle, así que no vale la pena ahondar más.

Listo el constructor, pasemos a los otros dos métodos.

```
@Override
public void onCreate(SQLiteDatabase baseDeDatos) {

}

@Override
public void onUpgrade(SQLiteDatabase baseDeDatos, int versionAnterior, int versionNueva) {

}
```

El ***onCreate*** lo vamos a usar frecuentemente, así en breve vamos a escribir alguito de código dentro de él. El ***onUpgrade*** también quedará a la espera de tiempos mejores, ya que no nos interesa por ahora.

La estructura de nuestra clase entonces, queda terminada, lista para ser *codeada*:

```
public class baseTP3SQLiteHelper extends SQLiteOpenHelper {

    public baseTP3SQLiteHelper(Context contexto, String Nombre, SQLiteDatabase.CursorFactory fabrica, int Version) {
        super(contexto, Nombre, fabrica, Version);

    }

    @Override
    public void onCreate(SQLiteDatabase baseDeDatos) {

    }

    @Override
    public void onUpgrade(SQLiteDatabase baseDeDatos, int versionAnterior, int versionNueva) {

    }
}
```

Tal como dijimos, el constructor y el ***onUpgrade*** no se van a tocar por ahora.

Pero el ***onCreate*** si. En él, vamos a escribir el código necesario para crear la base y sus tablas, de forma que cuando queramos usar la base, tanto la base en sí, como sus tablas, estén ya creadas.

Vamos a usar la sana costumbre de usar ***Log.d*** a modo de comentarios, de forma que no solo nos ayude a leer el código, sino que nos sirva para *debuggear* la aplicación.

Nuestro ***onCreate*** entonces, comienza así:

```
Log.d("SQLite", "Declaro e inicializo la variable para crear la tabla Personas");
String sqlCrearTablaPersonas;
sqlCrearTablaPersonas="create table personas (nombre text, edad integer)";
```

Acá no hicimos nada nuevo: tan solo declaramos una variable ***String***, a la que le asignamos una instrucción ***SQL*** para crear una tabla. Esta instrucción ***SQL*** no es un invento de Android, ni de ***SQLite***: es absolutamente estándar del lenguaje SQL, que existe en todos los motores de bases de datos del universo (***Microsoft Access***, ***Microsoft SQL Server***, ***Oracle***, ***MySQL***, ***Informix***, ***DB2***, ***VFP***, etc.).

Lo que sí es propio de ***SQLite*** es lo reducido de los tipos de campos que existen: mientras que en los motores más sofisticados suele haber 20 o 30 tipos de datos diferentes, en ***SQLite*** hay solo dos: ***TEXT*** e ***INTEGER***. Cool!

Lo que nos falta es decirle al motor de ***SQLite*** que ejecute esa instrucción sobre una base de datos determinada.

```
Log.d("SQLite", "Invoco al creador de la tabla");
baseDeDatos.execSQL(sqlCrearTablaPersonas);
```

Vemos dos cosas acá: por un lado, que estamos usando el objeto ***baseDeDatos*** que recibimos como parámetro. Por otro lado, estamos invocando a su método ***execSQL***, que ejecuta la instrucción ***SQL*** que le mandamos, siempre que no devuelva resultados (es decir, instrucciones que solo sean de escritura).

Con eso, tenemos la clase manejadora de la base de datos completa. Quedaría así:

```
public class baseTP3SQLiteHelper extends SQLiteOpenHelper {

    public baseTP3SQLiteHelper(Context contexto, String Nombre, SQLiteDatabase.CursorFactory fabrica, int Version) {
        super(contexto, Nombre, fabrica, Version);

    }

    @Override
    public void onCreate(SQLiteDatabase baseDeDatos) {

        Log.d("SQLite", "Declaro e inicializo la variable para crear la tabla Personas");
        String sqlCrearTablaPersonas;
        sqlCrearTablaPersonas="create table personas (nombre text, edad integer)";

        Log.d("SQLite", "Invoco al creador de la tabla");
        baseDeDatos.execSQL(sqlCrearTablaPersonas);

        Log.d("SQLite", "Fin de la creación de la tabla Personas");

    }

    @Override
    public void onUpgrade(SQLiteDatabase baseDeDatos, int versionAnterior, int versionNueva) {

    }
}
```

Ahora sí, vamos a la ***activity*** principal. Primeramente, vamos a declarar dos objetos públicos a toda la clase, ya que las vamos a usar en distintos lugares.

```
//Objeto para manipular las operaciones de la base de datos
baseTP3SQLiteHelper accesoBaseTP3;

//Objeto de la base de datos
SQLiteDatabase baseDatos;
```

El primero de estos dos elementos será el que usaremos para instanciar la clase que habíamos creado recientemente, por lo que será definida de ese tipo de datos. El segundo objeto es el que contendrá la referencia directa a la base de datos que usaremos, por lo que será de un tipo de datos nativo, llamado ***SQLiteDatabase***.

Esos dos objetos los podemos instanciar cuando queramos, pero dado que los vamos a usar de distintos puntos de nuestra aplicación, tal vez sea conveniente hacer una pequeña función que se ocupe de instanciarlas e inicializarlas cada vez. Esa función realizará tres cosas: instanciar la clase que habíamos creado, asignar una referencia al objeto de acceso a la base de datos, y verificar que la base se haya podido abrir sin problemas. Vamos a hacerla.

Empezamos por instanciar la clase:

```
accesoBaseTP3=new baseTP3SQLiteHelper(this, "baseTP3", null, 1);
```

El único parámetro que nos interesa es el segundo, que indica el nombre de nuestra base de datos, ya que podemos tener varias bases en el mismo dispositivo.

Luego, asignamos un vínculo al objeto de la base:

```
baseDatos=accesoBaseTP3.getWritableDatabase();
```

Estamos vinculando el objeto ***baseDatos*** a una base, en modo de escritura.

Por último, verificamos que la base se haya podido abrir:

```
if (baseDatos != null) {
```

Si ***baseDatos*** no es ***null***, quiere decir que se pudo abrir sin problemas.

Sería útil escribir esto en una función, de forma de usarla cuando querramos:

```
Boolean baseDeDatosAbierta() {

    Boolean responder;

    Log.d("SQLite", "Inicializo el accesor a la base mandándole, como segundo parámetro, el nombre de la base");
    accesoBaseTP3=new baseTP3SQLiteHelper(this, "baseTP3", null, 1);

    Log.d("SQLite", "Inicializo accediendo a la base en modo escritura");
    baseDatos=accesoBaseTP3.getWritableDatabase();

    Log.d("SQLite", "Verifico que la base se haya abierto correctamente");
    if (baseDatos != null) {
        Log.d("SQLite", "Base de datos abierta correctamente");

        responder=true;
    } else {
        Log.d("SQLite", "La base NO pudo ser correctamente abierta");
        responder=false;
    }

    return responder;
}
```

Teniendo ya la función lista, estamos en condiciones de operar sobre la base de datos. Sabemos que sobre las tablas hay tres operaciones de escritura y una de lectura. Comencemos por el ***Insert***. El código autocomentado por los ***Log.d*** ayuda muchísimo.

```
Log.d("SQLite", "Intento abrir la base");
if (baseDeDatosAbierta() == true) {

    Log.d("SQLite", "Voy a hacer un par de Inserts de ejemplo");

    Log.d("SQLite", "Creo un registro, especificando cada campo, y el valor que recibirá");
    ContentValues nuevoRegistro;

    nuevoRegistro=new ContentValues();
    nuevoRegistro.put("Nombre", "Mónica Galindo");
    nuevoRegistro.put("Edad", 18);

    Log.d("SQLite", "Ejecuto el insert, indicándole el nombre de la tabla y el registro a agregar");
    baseDatos.insert("Personas", null, nuevoRegistro);

    Log.d("SQLite", "Agrego otro registro, para lo cual vuelvo a instanciar el objeto, y vuelvo a llamar al Insert");
    nuevoRegistro=new ContentValues();
    nuevoRegistro.put("Nombre", "Rosamel Fierro");
    nuevoRegistro.put("Edad", 25);
    baseDatos.insert("Personas", null, nuevoRegistro);

    Log.d("SQLite", "Agrego un tercer registro");
    nuevoRegistro=new ContentValues();
    nuevoRegistro.put("Nombre", "Ana Lisa Meltrozo");
    nuevoRegistro.put("Edad", 38);
    baseDatos.insert("Personas", null, nuevoRegistro);

    Log.d("SQLite", "Cierro la base");
    baseDatos.close();

    Log.d("SQLite", "Base de datos cerrada");
}
```

Lo que hacemos es bastante simple. Primero que nada, usando la función que habíamos creado, abrimos la base. Sabemos que si esa función nos devuelve un ***true***, quiere decir que la base existe, está abierta, y que en nuestro ***baseDatos*** tenemos acceso a ella.

El paso siguiente es crear el objeto en el que indicaremos cada uno de los campos a los que vamos a dar valor, y sus valores. Esto lo hacemos con un objeto de tipo ***ContentValues***. Como vemos, es parecido al ***Bundle*** que conocemos, que tiene estructura de diccionario, y donde cada elemento corresponde a un campo de la tabla.

Una vez armado ese registro, usamos el método ***insert*** del objeto ***baseDatos***, al que le mandamos en el primer parámetro el nombre de la tabla sobre la que queremos hacer el ***insert***, y en el tercer parámetro el diccionario de valores para esos registros.

De la misma forma que hacemos el ***insert***, podemos borrar registros:

```
Log.d("SQLite","Intento abrir la base");
if (baseDeDatosAbierta() == true) {
    Log.d("SQLite","Ejecuto el vaciado de la base");

    baseDatos.delete("Personas", "", null);

    Log.d("SQLite","Base vaciada");
}
```

En este caso, el segundo parámetro de ***delete*** es la condición de borrado, es decir, lo que iría en el ***Where*** del ***delete***. Al dejarlo vacío, estamos indicando que queremos borrar todos los registros

de la tabla. Pero, por ejemplo, podíamos haber puesto un "***edad>20***" si nos hubiera venido en gana.

Por último, tenemos el ***update***, que funciona como una mezcla entre el ***insert*** y el ***delete***: requiere un diccionario de parámetros, para indicar qué campos serán reemplazados con qué valores, pero también puede llevar un ***string*** con una condición para indicar sobre cuáles registros aplica. De esta forma, el primer parámetro del ***update*** será el nombre de la tabla, el segundo será el nombre del diccionario de datos, el tercero será el string de condición, y el cuarto será nuestro querido ***null***.

Ya sabido cómo escribir, nos queda saber cómo leer. Esto tampoco es difícil:

```
Log.d("SQLite", "Declaro el objeto Cursor que recibirá el set de registros");
Cursor conjuntoDeRegistros;

Log.d("SQLite", "Ejecuto la lectura");
conjuntoDeRegistros=baseDatos.rawQuery("select nombre, edad from personas", null);
```

Primero, declaramos un objeto de tipo ***cursor***. Es importante aclarar que la palabra "***cursor***" no se refiere a esa rayita parpadeante que nos indica dónde estamos escribiendo, sino que son las siglas de ***Current Set Of Records***, o sea, "***Conjunto actual de registros***".

Una vez declarado, siempre usando métodos de ***baseDatos***, llamamos a la función ***rawQuery***, que ejecuta la consulta, y nos devuelve ese conjunto de registros.

Acto seguido, tenemos que verificar si obtuvimos al menos un registro. Por qué podríamos no obtener registros? Porque la tabla estuviera vacía, o porque hayamos puesto una condición en el ***select*** que no generara ningún registro como resultado.

```
Log.d("SQLite", "Verifico si trajo al menos un registro");
if (conjuntoDeRegistros.moveToFirst() == true){
```

Qué hace la función ***moveToFirst()***? Tal como suponemos, mueve el puntero de registro al primer registro. Si pudo hacerlo, devuelve ***true***. Si no pudo, porque no existe un primer registro, devuelve ***false***.

A partir de ahí, si hay registros, tengo que entrar en una repetitiva que lea el registro en el que estoy parado (el primero), y avance al próximo. Si pudo, porque había más registros, vuelvo a leer donde estoy parado (ahora el segundo registro), y vuelvo a avanzar. Cuando no puede avanzar más, porque no hay más registros, la repetitiva debe terminar.

```
Log.d("SQLite", "Inicializo un contador para saber cuántos registros hay, si es que me interesara");
int cantidadRegistros=0;

Log.d("SQLite", "Hay al menos un registro, comienzo a recorrerlos");
do  {
    cantidadRegistros++;

    Log.d("SQLite", "Obtengo el campo que está en la primera columna, en forma de string");
    String Nombre=conjuntoDeRegistros.getString(0);
    listaDatos.add(Nombre);

} while (conjuntoDeRegistros.moveToNext() == true);
```

Ahí tenemos lo que necesitamos saber. La función que avanza al próximo registro se llama ***moveToNext***, y al igual que ***moveToFirst***, devuelve ***true*** si pudo hacerlo, o ***false*** en caso contrario.

También tenemos el **getString** (y ***getInt***, y get...) para obtener el campo que está en la columna ***X***, y guardarlo en una variable. En nuestro ejemplo, estamos, luego agregándolo a un **ArrayList<String>** que, por supuesto, declaramos e inicializamos previamente.

Dos aclaraciones importantes.

Por un lado, recordemos que SOLO podemos leer campos del registro en el que estamos parados. NUNCA podemos leer campos de otros registros, a menos que nos desplacemos hasta ellos. De forma que, si o si, debemos posicionarnos en un registro para poder leer campos de él.

Cómo nos movemos a un registro? En forma secuencial. ***MoveToFirst()*** nos lleva al primero, y ***MoveToNext()*** nos lleva al siguiente. Ambas funciones nos devuelven un ***True*** si pudieron desplazarse, de forma que si ***MoveToFirst()*** devuelve False, es porque NO hay un primer registro, porque el Cursor NO tiene registros. Y si ***MoveToNext()*** devuelve ***False***, es porque no pudo ir a un registro siguiente, porque ya llego al final de la recorrida.

Por otro lado, cuando estoy parado en un determinado registro, puedo leer cualquier campo de ese registro. Supongamos que nuestro select es: "***Select nombre, domicilio, edad, mayorDeEdad from personas***", es decir, nuestro ***Cursor*** tiene N registros, pero todos los registros tienen 4 campos.

Entonces, para leer el campo ***Nombre*** del registro en el que estamos parados, usaremos un ***.getString(0)***. Para leer la edad será un ***.getInt(2)***, es decir, "***dame el campo que está en la posición 2 (0=Nombre, 1=Domicilio, 2=Edad) en forma de Int***".

Recordemos que SOLO podemos leer campos del registro en el que estamos parados.