Motor SQLLite

|  |
| --- |
|  |
| UD 5: Base de Datos SQLLite |
|  |
| Desarrollo de aplicaciones para Dispositivos Móviles |



**cENTRO DE ESTUDIOS SEIM**

1 de julio de 2012

Autor: Daniel Miguel

SQLite es un motor de bases de datos de pequeño tamaño, que no necesitar servidor, precisar poca configuración, es transaccional 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.

El procedimiento habitual para acceder a una base de datos SQLLite desde una aplicación Android, consiste en utilizar una clase derivada de *SQLiteOpenHelper*, que personalizaremos con el código necesario para crear y manejar nuestra base de datos según nuestras necesidades.

Creación de una base de datos

Para crear una base de datos en SQLLite desde programa usaremos una clase derivada de *SQLiteOpenHelper* redefiniendo los métodos *OnCreate* y *OnUpgrade*.

Para manipular la base de datos crearemos una instancia de nuestra clase derivada y ocurrirá lo siguiente:

* Si la base de datos ya existe, simplemente se establecerá una conexión con ella.
* Si la base de datos existe pero la versión no coincide con la solicitada, simplemente se ejecutará el método redefinido *OnUpgrade.*
* Si la base de datos no existe, se ejecutará el método *OnCreate.*

Una vez que tenemos una referencia a una instancia de la clase derivada de *SQLiteOpenHelper* 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.

|  |
| --- |
| Ejemplo: import android.content.Context;  import android.database.sqlite.SQLiteDatabase;  import android.database.sqlite.SQLiteDatabase.CursorFactory;  import android.database.sqlite.SQLiteOpenHelper;  public class UsuariosSQLiteHelper extends SQLiteOpenHelper {  //Sentencia SQL para crear la tabla de Usuarios  String sqlCreate = "CREATE TABLE Usuarios (codigo INTEGER, nombre 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 versionAnterior, int versionNueva) {  //NOTA: Por simplicidad del ejemplo aquí utilizamos directamente la opción de  // eliminar la tabla anterior y crearla de nuevo vacía 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 debería ser más elaborado.  //Se elimina la versión anterior de la tabla  db.execSQL("DROP TABLE IF EXISTS Usuarios");  //Se crea la nueva versión de la tabla  db.execSQL(sqlCreate);  } |

La sintaxis de las sentencias de SQLite la podemos consultar en el siguiente vínculo: <http://www.sqlite.org/lang.html>

A continuación podemos crearíamos una actividad que al cargarse cree una nueva instancia de la clase DBHelper. A partir de la misma obtenga una base de datos de lectura o escritura, según proceda, y si ha conseguido conectar ya podremos trabajar con ella.

En el momento de instanciar nuestra clase, le pasamos el contexto (la Actividad actual), el nombre de la base de datos, *null* en CursorFactory y el número de versión.

Si no encuentra la base de datos, ejcutará onCreate, si la encuentra con otro número de versión ejecutará onUpgrade.

|  |
| --- |
| Ejemplo: import android.app.Activity;  import android.database.sqlite.SQLiteDatabase;  import android.os.Bundle;  public class AndroidBaseDatos extends Activity  {  @Override  public void onCreate(Bundle savedInstanceState)  {  super.onCreate(savedInstanceState);  setContentView(R.layout.main);  //Abrimos la base de datos 'DBUsuarios' en modo escritura  UsuariosSQLiteHelper usdbh =  new UsuariosSQLiteHelper(this, "DBUsuarios", null, 1);  SQLiteDatabase db = usdbh.getWritableDatabase();  //Si hemos abierto correctamente la base de datos  if(db != null)  {  //Insertamos 5 usuarios de ejemplo  for(int i=1; i<=5; i++)  {  //Generamos los datos  int codigo = i;  String nombre = "Usuario" + i;  //Insertamos los datos en la tabla Usuarios  db.execSQL("INSERT INTO Usuarios (codigo, nombre) " +  "VALUES (" + codigo + ", '" + nombre +"')");  }  //Cerramos la base de datos  db.close();  }  }  } |

En nuestro caso, una vez abierta, añadimos 5 registros mediante ejecución de instrucciones INSERT. No obstante podemos hacer uso de los métodos de la clase OpenHelper para manipular los datos.

Manipulación de datos mediante consultas sql

El primer método de manipulación de datos desde Android es la ejecución directa de sentencias SQL mediante la orden execSQL().

|  |
| --- |
| Ejemplos: db execSQL("INSERT INTO Usuarios (codigo, nombre) VALUES (1,’UNO’)");.  db.execSQL("DELETE FROM Usuarios WHERE codigo= 1");  db.execSQL("UPDATE Usuarios SET nombre='dos' WHERE codigo =1"); |

Si lo que queremos es utilizar parámetros:

db.execSQL("DELETE FROM Usuarios WHERE usuario=?", args);

Donde args representa la lista de argumentos.

Manipulación de datos mediante métodos de SQLiteDataBase

# El método insert:

Se utiliza para insertar nuevos registros y tiene la siguiente estructura:

db.insert(String nombreTabla, String param2, ContentValues datos);

Donde el primer parámetro es el nombre de la tabla, el segundo de momento lo dejaremos a null y el tercero es de tipo *ContentValues*.

*ContentValues* es una colección que almacena los datos del registro a insertar en forma de pares **columna, valor**.

Un ejemplo de inserción sería el siguiente:

|  |
| --- |
| Ejemplo: //Creamos el registro a insertar como objeto ContentValues  ContentValues nuevoRegistro = new ContentValues();  nuevoRegistro.put("usuario", "usu10");  nuevoRegistro.put("email","usu10@email.com");  //Insertamos el registro en la base de datos  db.insert("Usuarios", null, nuevoRegistro); |

# El método update:

Funciona de manera similar al anterior pero incluyendo un parámetro adicional que representa la condición del **WHERE.**

db.update(String nombreTabla, ContentValues datos , String condicion);

|  |
| --- |
| //Establecemos los campos-valores a actualizar  ContentValues valores = new ContentValues();  valores.put("email","usu1\_nuevo@email.com");  //Actualizamos el registro en la base de datos  db.update("Usuarios", valores, "usuario='usu1'"); |

# El método delete.

En este caso tendría dos parámetros, el nombre de la tabla y la condición.

db.delete(String nombreTabla,String condicion);

|  |
| --- |
| //Eliminamos el registro del usuario 'usu2'  db.delete("Usuarios", "usuario='usu2'"); |

En caso de que la condición lleve parámetros, estos se añaden después del último argumento:

|  |
| --- |
| Ejemplo //Actualizar dos registros con update(), utilizando argumentos  ContentValues valores = new ContentValues();  valores.put("email","usu1\_nuevo@email.com");  String[] args = new String[]{"usu1", "usu2"};  db.update("Usuarios", valores, "usuario=? OR usuario=?", args); |

Seleccionar registros.

# Método rawQuery.

Este método recibe como parámetros una sentencia SQL complete y devuelve un objeto de tipo cursor.

|  |
| --- |
| Cursor c = db.rawQuery(" SELECT usuario,email FROM Usuarios WHERE usuario='usu1' ");  //con parámetros  String[] args = new String[] {"usu1"};  Cursor c = db.rawQuery(" SELECT usuario,email FROM Usuarios WHERE usuario=? ", args); |

# Método Query:

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 al final más indicando el número máximo de registros que queremos que nos devuelva la consulta.

|  |
| --- |
| Ejemplo String[] campos = new String[] {"usuario", "email"};  String[] args = new String[] {"usu1"};  Cursor c = db.query("Usuarios", campos, "usuario=?", args, null, null, null); |

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), en caso de contener un dato de tipo real llamaríamos a getDouble(1), y de forma análoga para todos los tipos de datos existentes.

|  |
| --- |
| String[] campos = new String[] {"usuario", "email"};  String[] args = new String[] {"usu1"};  Cursor c = db.query("Usuarios", campos, "usuario=?", args, null, null, null);  //Nos aseguramos de que existe al menos un registro  if (c.moveToFirst()) {  //Recorremos el cursor hasta que no haya más registros  do {  String usuario = c.getString(0);  String email = c.getString(1);  } while(c.moveToNext());  } |

Controles de selección: listview

Para rellenar un control compuesto tipo ListView, necesitamos 3 componentes:

# Array de palabras:

Será el conjunto de datos que aparecerán como opciones seleccionables de la listview:

String [] palabras={“enero”, ”febrero”, ”marzo”, abril” ,”mayo” ,”junio” ,”julio” ,”agosto” ,”septiembre” ,”octubre” ,”noviembre”,”diciembre };

Puede ser sustituido por un recurso de tipo Array de strings y guardado como recurso .xml.

# Plantilla de datos.

Será un nuevo layout almacenado como recurso .xml donde se describirán los recursos y el formato en el que se mostrarán los ítems de la lista.

En caso de no querer personalizar dicho formato, podemos utilizar el que contiene la clase R por defecto (android.R.layout.simple\_list\_item\_1 ó android.R.layout.simple\_list\_item\_2)

Un ejemplo de plantilla sería:

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"utf-8"*?>  <TextView xmlns:android=*"http://schemas.android.com/apk/res/android"*  android:id=*"@+id/txtplantilla"*  android:layout\_width=*"match\_parent"*  android:layout\_height=*"wrap\_content"*  /> |

# Adaptador de datos.

A continuación tenemos que crear un adaptador de datos basado en el Array de palabras del apartado anterior. Utilizaremos la clase ArrayAdapter que sirve para convertir un array en origen de datos de un control de selección.

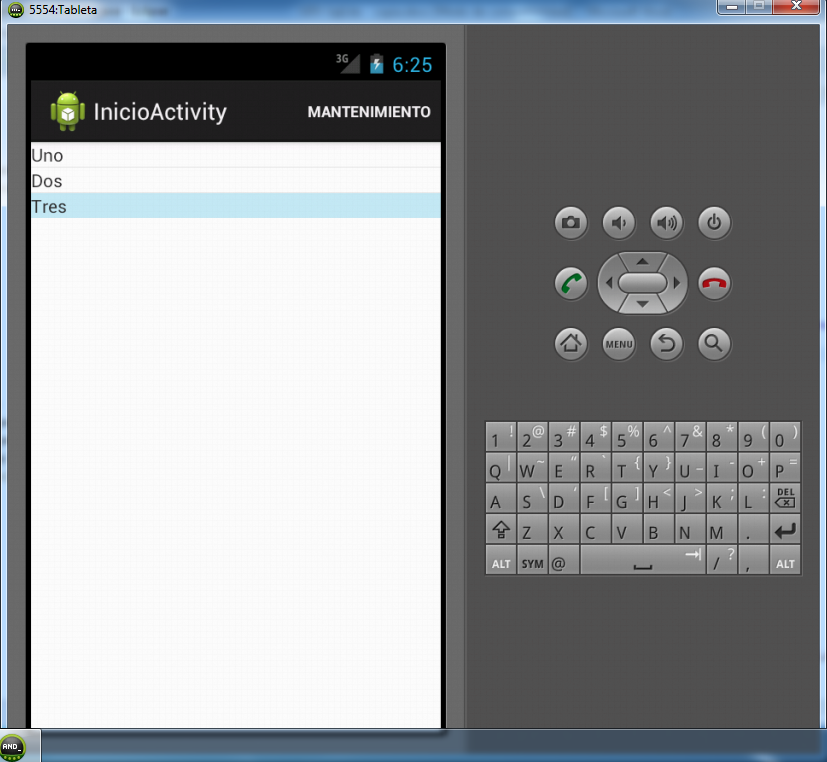
Tiene dos posibles sobrecargas de su constructor.

* public ArrayAdapter ([Context](file:///C:\Users\dani\android-sdks\docs\reference\android\content\Context.html) context, int textViewResourceId): el origen de datos es una colección sin elementos, se utiliza en los casos en los que prefiramos llenar el adaptador elemento a elemento.
* public ArrayAdapter ([Context](file:///C:\Users\dani\android-sdks\docs\reference\android\content\Context.html) context, int textViewResourceId, T[] objects): el origen de datos se vincula a un array de objetos de la clase que necesitemos, en nuestro caso de Strings.

En nuestro ejemplo utilizaremos la segunda opción dado que hemos inicializado las cadenas que aparecerán en la ListView.

|  |
| --- |
| lv=(ListView)findViewById(R.id.*listView1*);  origen=**new** ArrayAdapter<String>(**this**,R.layout.*plantilla\_lista*,**new** String[] {"Uno","Dos","Tres"});    lv.setAdapter(origen); |

El resultado es el de la figura.



Rellenar un listView con los datos de una consulta.

En el ejemplo anterior vimos que eran necesarios 3 componentes para rellenar una ListView: el origen de datos, el adaptador de datos y la plantilla.

Vamos a mantener la plantilla del ejercicio anterior y vamos a utilizar la primera sobrecarga del constructor, esto es, la que no recibía el tercer argumento.

Al no pasarle las palabras a mostrar, lo que crearemos será un origen con la colección de palabras inicialmente vacía e iremos añadiendo dinámicamente, mediante código, los ítems a visualizar en la lista.

En nuestro caso, además, los ítems los obtendremos del resultado de una consulta contra una base de datos de SQLITE:

|  |
| --- |
| lv=(ListView)findViewById(R.id.*listView1*);  origen=**new** ArrayAdapter<String>(**this**,R.layout.*plantilla\_lista*);  Cursor c=db.query("Motivos",**new** String [] {"IdMotivo as \_id"},**null**, **null**, **null**,**null**, **null**);  **while**(c.moveToNext())  {origen.add(c.getString(0));  }  listView1.setAdapter(origen); |

Vinculación de datos

En lugar de rellenar el ArrayAdapter establecido como origen de una listview, con los datos de una consulta, podemos directamente crear un objeto de tipo SimpleCursorAdapter (derivada de CursorAdapter), para vincular directamente un cursor a la lista, sin necesidad de tener que recorrerlo para añadir los elementos al adaptador.

El ejemplo siguiente muestra cómo hacerlo

|  |
| --- |
| DataBaseHelper dbh=**new** DataBaseHelper(**this**,"DBFestivos",**null**,1);  SQLiteDatabase db=dbh.getReadableDatabase();  Cursor c=db.query("Motivos",**new** String [] {"IdMotivo as \_id"},**null**, **null**, **null**,**null**, **null**); SimpleCursorAdapter sca = new SimpleCursorAdapter(this,R.layout.*plantilla\_lista*,c,new String[]{"\_id"},new int [] {R.id.*txtplantilla*}, cursorAdapter.*FLAG\_REGISTER\_CONTENT\_OBSERVER*);lv.setAdapter(sca); |

Hay que tener en cuenta varias cuestiones:

* Al menos una de las columnas tiene que llamarse “\_id”, esto se consigue poniendo un alias a la columna que haga de Identificador, a la hora de hacer la query.
* Todas las columnas de la query, tienen que tener un Alias que es el que se utilizará en la colección de campos de vinculación, en concreto, en el parámetro FROM.
* La forma general del constructor de SimpleCursorAdapter es:

##### SimpleCursorAdapter([Context](eclipse-javadoc:%E2%98%82=SqlLite/C:%5C/Users%5C/dani%5C/android-sdks%5C/platforms%5C/android-14%5C/android.jar%3Candroid.widget(SimpleCursorAdapter.class%E2%98%83SimpleCursorAdapter~SimpleCursorAdapter~Landroid.content.Context;~I~Landroid.database.Cursor;~%5C%E2%98%83Ljava.lang.String;~%5C%E2%98%83I~I%E2%98%82Context) context, int layout, [Cursor](eclipse-javadoc:%E2%98%82=SqlLite/C:%5C/Users%5C/dani%5C/android-sdks%5C/platforms%5C/android-14%5C/android.jar%3Candroid.widget(SimpleCursorAdapter.class%E2%98%83SimpleCursorAdapter~SimpleCursorAdapter~Landroid.content.Context;~I~Landroid.database.Cursor;~%5C%E2%98%83Ljava.lang.String;~%5C%E2%98%83I~I%E2%98%82Cursor) c, [String](eclipse-javadoc:%E2%98%82=SqlLite/C:%5C/Users%5C/dani%5C/android-sdks%5C/platforms%5C/android-14%5C/android.jar%3Candroid.widget(SimpleCursorAdapter.class%E2%98%83SimpleCursorAdapter~SimpleCursorAdapter~Landroid.content.Context;~I~Landroid.database.Cursor;~%5C%E2%98%83Ljava.lang.String;~%5C%E2%98%83I~I%E2%98%82String)[] from, int[] to, int flags);

Donde:

* Context es el contexto, esto es, la actividad actual.
* Layout es el .xml que contiene la plantilla.
* Cursos es el cursor previamente cargado mediante una query.
* From es un array de Strings formado por los nombres de los alias que hayamos dado a las columnas seleccionadas de la base de datos.
* To es un array de enteros, con los identificadores de los controles de la plantilla que hemos diseñado para mostrar los datos (tendrá más de un elemento en el caso de mostrar más de una columna).
* Flags es un valor entero que representa las opciones de configuración del Adaptador, lo habitual es poner CursorAdapter.*FLAG\_REGISTER\_CONTENT\_OBSERVER.*