Desarrollo para dispositivos móviles con Android: Almacenamiento con SQLite.

Contenido

1 Introducción

Si bien ya se han explicado las múltiples posibilidades de almacenamiento que proporciona Android, una de ellas merece una especial atención: acceso a bases de datos SQLite. Las bases de datos son muy importantes, ya que permiten manejar información que potencialmente puede superar la memoria RAM del dispositivo. Además, SQL permite el manejo de información con múltiples referencias cruzadas de manera sistemática. Por último, **Android** tiene incorporado el manejo de bases de datos SQLite sin necesidad de añadir ninguna librería.

2 Las clases SQLiteOpenHelper y SQLiteDatabase

La clase **SQLiteOpenHelper** permite el acceso inicial a la base de datos, así como una referencia **SQLiteDatabase** para poder manejarla una vez preparada, siendo muy útil para crear a partir de ella nuestro propio gestor de la base de datos. Así, la forma indicada de manejarse con SQLite es crear una clase derivada de **SQLiteOpenHelper**, en la cual crearemos las tablas de la base de datos, mediante la reescritura de los métodos **SQLiteOpenHelper**.*onCreate()* y **SQLiteOpenHelper**.*onUpgrade()*. Durante este documento, la clase derivada de **SQLiteOpenHelper** se denominará **DBManager**.

Una vez las tablas han sido creadas, es posible acceder a una referencia a la base datos de tipo **SQLiteDatabase**. Mediante esta referencia, es posible realizar modificaciones y consultas (*queries*) contra la base de datos. Estas *queries* devuelven información que pueden manejarse con cursores, como se verá más adelante.

3 Cómo utilizar la clase de acceso a SQLite

Una vez creada la clase derivada de **SQLiteOpenHelper**, es necesario hacer un objeto de esta clase accesible para todas las actividades de la aplicación (a no ser que la aplicación solo tenga una actividad), por lo que una opción es suplantar una clase aplicación (derivada de **Application**), y guardar en ella una referencia a la primera.

```
public class App extends Application {
    private DBManager dbman;

    @Override
    public void onCreate()
    {
        super.onCreate();
        this.dbman = new DBManager( this.getApplicationContext() );
    }

    /** @return the database open */
    public DBManager gmanetDBManager() {
        return this.db;
    }

    // más cosas...
}
```

En el código más arriba, el objeto e la clase **App** mantiene una referencia a la clase **DBManager**, que como se verá más adelante, deriva de **SQLiteOpenHelper**. Siguiendo los ejemplos anteriores, podríamos vernos tentados de guardar los datos en la base de datos al final de la aplicación, y cargarlos al comienzo. En realidad, utilizar así una base de datos no es la mejor estrategia. El gestor es capaz de manejar todas las modificaciones posibles a los datos, incluyendo altas, bajas y modificaciones, por lo que mantener a mayores la estructura de datos en memoria sería redundante.

La segunda opción es crear **DBManager** como un *singleton*, de manera que exista tan solo una instancia de ella, y que esta sea siempre accesible.

```
public class DBManager extends SQLiteOpenHelper {
    private DBManager(Context c, …

    // más cosas…

    public static DBManager getManager(Context c)
    {
        if ( instancia == null ) {
            instancia = new DBManager(c, …
        }

        return instancia;
    }

    private static DBManager instancia;
}
```

4 Derivando de la clase SQLiteOpenHelper

Como hemos discutido antes, es necesario crear una clase derivada de la clase **SQLiteOpenHelper** para poder indicar cómo se crea la base de datos, y cómo se debe comportar el gestor cuando se pasa a una nueva versión de la misma. Si bien es más o menos sencillo crear las tablas que componen la base de datos, la estrategia de actualización de una versión de la misma a otra suele ser complejo. En este texto, se asumirá que la base de datos simplemente se actualiza perdiendo todos sus datos y recreando las tablas.

4.1 Métodos básicos

El gestor SQLite que emplea **Android** (la clase **SQLiteDatabase**), admite varios métodos para aquellos casos más comunes, como inserciones, modificaciones o eliminaciones, tal y como se verá más adelante. Sin embargo, ofrece dos métodos básicos para interaccionar con la base de datos: **SQLiteDatabase**.*execSQL*(*s*, *datos*), y **SQLiteDatabase**.*rawQuery*(*s*, *datos*), siendo *s* una sentencia SQL, y *datos* un vector de cadenas con los valores necesarios para la consulta (puede ser **null** si no se necesitan valores específicos). Como se puede deducir de los nombres de ambos métodos, el segundo lanza consultas sobre la base de datos, mientras el primero se encarga del resto de posibles sentencias SQL. Se citan varios ejemplos a continuación.

```
db.execSQL( “DELETE FROM notas” );
Cursor cursor1 = db.rawQuery( “SELECT * FROM estudiantes”, null );
Cursor cursor2 = db.rawQuery( “SELECT * FROM notas WHERE calificacion > ?”, new String[]{ “7” } );
```

Los valores o datos que se proporcionan como vectores de cadenas, a sustituir por los interrogantes (‘?’), son necesarios debido a que componiendo las órdenes SQL como texto, es posible permitir ataques por inyección de SQL en nuestra aplicación. El ejemplo más arriba es artificioso, pues codifica igualmente un 7 literal que podría estar incluido sin problema en la misma orden. La complicación real viene cuando se toman estos datos del usuario, y este puede haber tecleado sentencias SQL como valores para esos datos. El uso de valores por separado elimina este problema.

En cuanto a los cursores, como se verá más adelante, se trata de objetos que permiten recorrer todos los registros que ha devuelto una consulta sobre la base de datos.

4.2 Transacciones

El código mostrado a continuación utilizará los métodos **SQLiteDatabase**.*beginTransaction()*, *endTransaction()* y *setTransactionSuccessful()*. La idea tras estos tres métodos es permitir el uso de transacciones, de manera que las operaciones sobre la base de datos que impliquen modificaciones, se lleven a cabo totalmente o no se lleven a cabo en absoluto. En caso de llegar a ejecutarse el método *setTransactionSuccessful()*, al ejecutarse a continuación el método *endTransaction()*, las operaciones se escribirán en la base de datos, es decir, se hace un *commit*. En caso de que el método

setTransactionSucessful() no llegue a ejecutarse (porque se ha producido una excepción), las operaciones realizadas desde la ejecución del método *beginTransaction()* se cancelan. Técnicamente, se efectúa el llamado *roll back* de la base datos.

Así, la forma correcta de utilizar estos tres métodos siempre es la siguiente:

```
try {
        db.beginTransaction();
        // modificaciones sobre la base de datos
        db.setTransactionSucessful();
} finally {
        db.endTransaction();
}
```

De forma natural, se obtiene el efecto deseado: la modificación deseada solamente tiene éxito si se llega a marcar la misma como correcta.

Nótese que además de la cláusula *finally*, también se puede reaccionar con una o varias cláusulas *catch()* a las excepciones concretas que se desee. Específicamente, la excepción **SQLException** es la que se lanza en caso de no poder ejecutar correctamente una operación sobre la base de datos.

4.3 Creando las tablas

El primer paso es la creación de las tablas que componen la base de datos. Esto se especifica mediante la reescritura (obligatoria), del método **SQLiteOpenHelper**.*onCreate()*. Se utiliza entonces el método **SQLiteDatabase**.*execSQL()* para ordenar la creación de las tablas.

```
public class DBManager extends SQLiteOpenHelper {
    public static final String DB_NOMBRE = "estudiantes";
    public static final int DB_VERSION = 1;

    public static String ESTUDIANTES_DNI = “_id”;
    public static String ESTUDIANTES_NOMBRE = “nombre”;

    public DBManager(Context context) {
        super( context, DB_NOMBRE, null, DB_VERSION );
    }

    @Override
    public void onCreate(SQLiteDatabase db)
    {
        Log.i(  "DBManager", "Creando BBDD " + DB_NOMBRE + " v" + DB_VERSION);

        try {
            db.beginTransaction();
            db.execSQL( "CREATE TABLE IF NOT EXISTS “ + TABLA_ESTUDIANTES + “("
                    + ESTUDIANTES_DNI + " int PRIMARY KEY,"
                    + ESTUDIANES_NOMBRE + " string(255) NOT NULL"
                    + ")" );
            db.setTransactionSuccessful();
        } catch(SQLException exc) {
            Log.e( “DBManager.onCreate”, “Creando “ + TABLA_ESTUDIANTES + “: “ + exc.getMessage() );
        } finally {
            db.endTransaction();
        }
    }

    // más cosas…
}
```

Para crear la base de datos, se utiliza la sentencia SQL *create table* (a través del método **SQLiteOpenHelper**.*execSQL(s)*), que permite especificar el nombre de la tabla y los campos que la componen. Nótese que se utiliza la variante *if not exists*, de manera que la tabla no se vuelve a crear en caso de que ya exista. La sentencia SQL ejecutada, es por tanto, la siguiente:

```
CREATE TABLE IF NOT EXISTS estudiantes(
        _id int PRIMARY KEY,
        nombre string(255) NOT NULL)
```

Los modificadores *primary key* y *not null* significan clave primaria (permite identificar unívocamente a un registro en esta tabla) y que el contenido del campo no puede ser vacío, respectivamente.

Quizás llame la atención el uso de *_id* en lugar de *dni*, por ejemplo. Lo cierto es que, por convención, la clave primaria de una tabla debe ser llamada así, o en caso contrario, ciertas características (como el uso de **SimpleCursorAdapter**), no funcionarán correctamente.

La tabla a continuación contiene una introducción a los tipos de datos que soporta SQL/SQLite y su comparación con los tipos de Java.

Tipo	Tipo Java	Tipo SQL
Cadena de texto	String	string
Número entero	int	int
Número real	double	real
Fecha	Calendar/Date	string/int
Hora	Calendar/Time	string/int
Booleanos	boolean	int (0 ó 1)

Los valores booleanos se pueden leer y escribir directamente, SQLite se encarga de realizar la conversión entre uno y otro tipo.

Una mención aparte merece el tratamiento de fechas u horas. En SQLite no existe un tipo específico, si bien la documentación recomienda el uso de las siguientes posibilidades.

- **string**: Una cadena de caracteres que debe contener la fecha/hora en el formato ISO-8601: YYYY-MM-DD HH:MM:SS. Si se requiere una precisión de milisegundos, se puede utilizar: YYYY-MM-DD HH:MM:SS.SSS.
- **int**: Un entero que se calcula como el número de segundos desde 1970-01-01 00:00:00, convención UTC.

Para convertir entre fechas de Java y fechas a almacenar en la base de datos, por lo tanto, es necesario hacer una codificación/decodificación. En el caso de elegir una cadena de caracteres como soporte, es posible utilizar código como el siguiente para guardar una fecha (donde *fechaNacimiento* es la fecha en cuestión, y *fecha_nacimiento* el nombre de la columna de la tabla):

```
SimpleDateFormat isoDateFormat = new SimpleDateFormat( "yyyy-MM-dd HH:mm:ss",Locale.ROOT );
isoDateFormat.setTimeZone( TimeZone.getTimeZone( "UTC" ) );
String strFechaNacimiento = isoDateFormat.format( fechaNacimiento );
…
```

Si lo que se necesita es recuperar una fecha de la base de datos, entonces

```
SimpleDateFormat isoDateFormat = new SimpleDateFormat( "yyyy-MM-dd HH:mm:ss", Locale.ROOT );
isoDateFormat.setTimeZone( TimeZone.getTimeZone( "UTC" ) );
String strFechaNacimiento = cursor.getString( cursor.getColumnIndex( “fecha_nacimiento” ) );
Date fechaNacimiento = isoDateFormat.parse( strFechaNacimiento );
…
```

En caso de elegir el número de segundos desde el 1 de enero de 1970 (habitualmente conocido como *UNIX Epoch*), entonces para guardar solo es necesario llamar al método **Date**.*getTime()*, mientras que para recuperar basta con crear un objeto **Date** pasándole al constructor los milisegundos, es decir, los segundos multiplicados por mil.

```
int intFechaNacimiento = fechaNacimiento.getTime() / 1000;
```

```
int intFechaNacimiento = cursor.getInt( cursor.getColumnIndex( “fecha_nacimiento” ) );
Date fechaNacimiento = new Date( intFechaNacimiento * 1000 );
…
```

4.4 Actualizando la base de datos

En el ejemplo más arriba, se crea la base de datos *estudiantes* cuya versión es la 1. La necesidad de especificar una versión está relacionada con el método **SQLiteOpenHelper**.*onUpgrade(db, v1, v2)*, que es llamado automáticamente cuando cambia la versión de la base de datos *db* de *v1* a *v2*. Por ejemplo, supongamos que añadimos la columna *email* a la tabla *estudiantes*. En tal caso, deberemos incrementar la constante DB_VERSION, que pasará de 1 a 2. Así, se llamará automáticamente a *onUpgrade()*, con *v1* siendo 1, y *v2* siendo 2. Allí especificaremos lo que sucede con los registros que ya existen, añadiéndoles probablemente un valor por defecto a su e.mail. En el caso que nos ocupa, solo crearemos (ya que la inclusión del método es obligatorio), un método que elimina la tabla en cuestión, y la vuelve a crear.

```
public class DBManager extends SQLiteOpenHelper {
    // más cosas…
    @Override
    public void onUpgrade(SQLiteDatabase db, int v1, int v2)
    {
        Log.i(  "DBManager", "DB: " + DB_NOMBRE + ": v" + v1 + " -> v" + v2 );
        try {
            db.beginTransaction();
            db.execSQL( "DROP TABLE IF EXISTS " + TABLA_COMPRA );
            db.setTransactionSuccessful();
        }  catch(SQLException exc) {
            Log.e( "DBManager.onUpgrade", exc.getMessage() );
        } finally {
            db.endTransaction();
        }
        this.onCreate( db );
    }
}
```

Este método es, podríamos decir, genérico, en el sentido de que vale para cualquier cambio de versión. También es expeditivo: al eliminar la tabla y volver a crearla (a través de la llamada a **SQLiteOpenHelper**.*onCreate(db)*), se pierden todas las filas almacenadas en la tabla, así que sirve como recurso de última instancia pero no como una forma genérica de actualización para cualquier base de datos.

5 Realizando modificaciones sobre la base de datos

Para realizar modificaciones sobre la base de datos, se obtiene primero una referencia a un objeto **SQLiteDatabase** mediante el método **SQLiteOpenHelper**.*getWritableDatabase()*., y entonces se ejecuta la operación mediante uno de los métodos disponibles: **SQLiteDatabase**.*execSQL()*, que es el más genérico, permitiendo cualquier sentencia SQL, y los más específicos *insert()*, *update()*, y *delete()*. Es preferible utilizar siempre estos últimos, pues supone una menor oportunidad para realizar ataques por inyección de SQL o cualquier otro tipo de error.

En esta sección se realiza un pequeño y sucinto repaso a las posibilidades de SQL, si bien la discusión en profundidad de este lenguaje de manejo de base de datos está fuera del alcance de este texto.

5.1 Inserciones de registros

Para insertar registros nuevos, se utiliza la orden SQL *insert into*, de manera que se especifica la tabla donde insertar, así como los valores en sí a insertar. Nótese que siempre que una sentencia SQL tiene parámetros (especialmente si contienen información obtenida del usuario), siempre deben especificarse como se hace en el código más abajo, de forma que será necesario pasar un vector de cadenas que contendrá los datos reales.

En el siguiente código, se asume que se tiene acceso al objeto *gestorDB* de la clase **DBManager** (derivada de **SQLiteOpenHelper**), bien debido a que se ha creado como un *singleton*, porque está disponible de cualquier manera en la actividad, o porque se obtiene a través de la clase derivada de **Application** que sustituye a la aplicación por defecto de **Android**.

```
SQLiteDatabase db = gestorDB.getWritableDatabase();

try {
    db.beginTransaction();
    db.execSql( “INSERT OR IGNORE INTO “ + TABLA_ESTUDIANTES
                + “(“ + ESTUDIANTES_DNI
                + “,” + ESTUDIANTES_NOMBRE + “) VALUES( ?, ? )”,
            new String[]{ “12345678”, “Baltasar” } );
    db.setTransactionSuccesful();
} catch(…
```

Así, la sentencia anterior crea dentro de la tabla *students* un nuevo registro para un tal Baltasar, siempre y cuando dicho registro no exista ya (cláusula *or ignore*). Para saber si existe o no, el gestor utiliza el DNI, que es la clave primaria en la tabla.

Aunque *execSQL()* es una herramienta muy poderosa, existe también el método *insert(t, ch, v)*, que es mucho más específico, siendo *t* el nombre de la tabla, *ch* el llamado *columnHack*, para el caso en el que realmente se desee crear filas con valores nulos, y *v* la colección de valores a insertar como nuevo registro.

```
SQLiteDatabase db = gestorDB.getWritableDatabase();
ContentValues valores = new ContentValues();

valores.put( ESTUDIANTES_DNI, dni );
valores.put( ESTUDIANTES_NOMBRE, nombre );

try {
    db.beginTransaction();
    db.insert( TABLA_ESTUDIANTES, null, valores );
    db.setTransactionSuccessful();
} catch(…
```

5.2 Actualización de registros

Los registros ya existentes se actualizan mediante la sentencia SQL *update set*. Como en el caso anterior, está disponible un método más específico en **SQLDatabase**.

```
SQLiteDatabase db = gestorDB.getWritableDatabase();

try {
    db.beginTransaction();
    db.execSql( “UPDATE “ + TABLA_ESTUDIANTES
                + “SET “ + ESTUDIANTES_DNI + “ = ? “
                + “,” + ESTUDIANTES_NOMBRE + “ = ? “
                + “WHERE “ + ESTUDIANTES_DNI + “ = ? VALUES( ?, ?, ? )”,
            new String[]{ “12345678”, “Baltasar”, “12345678” } );
    db.setTransactionSuccesful();
} catch(…
```

En el código anterior se emplea execSQL() con la sentencia SQL apropiada, mientras en el siguiente ejemplo se usa el método **SQLDatabase**.*update(t, v, w)* específico. En cuanto a los parámetros, *t* es el nombre de la tabla, *v* la colección de valores a actualizar, y *w* la cláusula *where* para restringir el efecto de *update*: téngase en cuenta que sin cláusula *where*, actualizaría todos los registros a los valores dados.

```
SQLiteDatabase db = gestorDB.getWritableDatabase();
ContentValues valores = new ContentValues();

valores.put( ESTUDIANTES_NOMBRE, nombre );
valores.put( ESTUDIANTES_DNI, dni );

try {
    db.beginTransaction();
    db.update( TABLA_ESTUDIANTES, valores, ESTUDIANTES_DNI + "= ?",
                                                new String[]{ Integer.toString( dni ) } );
    db.setTransactionSuccessful();
} catch(…
```

Es muy posible encontrarse con la necesidad de realizar una inserción o actualización en función de que el registro se encuentre o no. El siguiente método realiza esta función mediante una consulta previa, que se explican en mayor detalle en las secciones siguientes.

```
class DBManager {
   // más cosas….
   public void guarda(int dni, String nombre)
    {
        Cursor cursor = null;
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues values = new ContentValues();

        values.put( ESTUDIANTES_NOMBRE, nombre );
        values.put( ESTUDIANTES_DNI, dni );

        try {
            db.beginTransaction();
            cursor = db.query( TABLA_ESTUDIANTES, null, ESTUDIANTES_DNI + "=?",
                                new String[]{ Integer.toString( dni ) },
                                null, null, null, null );

            if ( cursor.getCount() > 0 ) {
                db.update( TABLA_ESTUDIANTES,
                           values, ESTUDIANTES_NOMBRE + "= ?", new String[]{ nombre } );
            } else {
                db.insert( TABLA_ESTUDIANTES, null, values );
            }

            db.setTransactionSuccessful();
        } catch(SQLException exc) {
            Log.e( "DBManager.guarda", exc.getMessage() );
        } finally {
            if ( cursor != null ) {
                cursor.close();
            }

            db.endTransaction();
        }
    }
}
```

5.3 Borrado de registros

Para eliminar registros, se utiliza la orden SQL *delete from*, de manera que se especifica la tabla donde borrar y la cláusula *where* que indica los registros a borrar (en caso de no aportar una cláusula *where*, se borran todos en la tabla).

```
SQLiteDatabase db = gestorDB.getWritableDatabase();
db.execSql( “DELETE FROM “ + TABLA_ESTUDIANTES + “ WHERE nombre <> ?”, new String[]{ “Baltasar” } );
```

Al igual que en los casos anteriores, también es posible utilizar el método específico **SQLDatabase**.*delete(t, w, v)*, siendo *t* la tabla donde eliminar, *w* la cláusula *where* a utilizar, y *v* la colección de valores para *where*.

```
SQLiteDatabase db = this.getWritableDatabase();

try {
       db.beginTransaction();
       db.delete( TABLA_COMPRA, COMPRA_COL_NOMBRE + "= ?", new String[]{ nombre } );
       db.setTransactionSuccessful();
} catch(SQLException exc) {
        Log.e( "DBManager.elimina", exc.getMessage() );
} finally {
        db.endTransaction();
}
```

6 Consultas

Las consultas (órdenes SQL tipo *select... from*), no precisan realizar modificaciones en la base de datos, por lo que es posible utilizar el método **SQLiteOpenHelper**.*getReadableDatabase()*, ligeramente más eficiente en recursos. Este es el mecanismo más poderoso de SQL devolviendo conjuntos de datos que pueden ser recorridos por un objeto de la clase android.database.**Cursor**.

```
SQLiteDatabase db = this.getReadableDatabase();
Cursor cursor = db.rawQuery( “SELECT “ + ESTUDIANTES_DNI + “, “ + ESTUDIANTES_NOMBRE
                             + ” FROM “ + TABLA_ESTUDIANTES” );

// Muestra todos los estudiantes por la consola
if ( studentsCursor.moveToFirst() ) {
    do {
        Estudiante estudiante = new Estudiante( cursor.getInt( 0 ), cursor.getString( 1 ) );
        System.out.println( estudiante );
    } while ( cursor.moveToNext() );
}

cursor.close();
```

El código anterior vuelvc por consola el contenido de toda la tabla de estudiantes. En primer lugar, se ejecuta el método **Cursor**.*moveToFirst()*, que mueve el cursor a comienzo del conjunto de datos devuelto, y devuelve *false* en caso de que el conjunto no contenga datos. De una forma de trabajo similar, **Cursor**.*moveToNext()* mueve el cursor a la siguiente fila del conjunto de datos, y devuelve *false* cuando ya se estaba en la última.

Para cada fila, el cursor permite obtener datos de distintos tipos pasando el número de columna (basado en cero). La primera columna, según la sentencia SQL *select* ejecutada, es un DNI que ha sido definido como entero. La segunda columna contiene el nombre del alumno, que debe ser obtenido como **String**. De ahí que el objeto **Student** se cree pasándole al constructor la primera columna como entero, y la segunda como cadena: ***new*** *Student(studentsCursor.getInt(0), studentsCursor.getString(1))*.

```
SQLiteDatabase db = this.getReadableDatabase();
Cursor cursor = db.query( TABLA_ESTUDIANTES, null, null, null, null, null, null, null );

// Muestra todos los estudiantes por la consola
if ( cursor.moveToFirst() ) {
    do {
        Estudiante estudiante = new Estudiante( cursor.getInt( 0 ), cursor.getString( 1 ) );
        System.out.println( estudiante );
    } while ( cursor.moveToNext() );
}

cursor.close();
```

Como se puede observar, el método **SQLDatabase**.*query(t, cols, w, wargs, gr, h, or, l)* tiene una lista de parámetros importante, y muchos de ellos no siempre se usan. La tabla es el primer argumento, *t*, siguiéndole un vector de cadenas con los nombres de los campos a recuperar (si es **null**, se asume que todos), *w* contendrá la cláusula *where* en formato texto, mientras *wargs* es un vector de cadenas con los valores a sustituir en lugar de las interrogaciones en el parámetro anterior (‘?’). El parámetro *gr* es también una cadena con el valor de la cláusula *group by*, siguiéndole igualmente los valores para las cláusulas *having*, *order by* y *limit* (esta última es opcional, puede omitirse en lugar de mandar **null**).

Un ejemplo algo más común es el siguiente, en el que se obtienen todas las filas cuya columna nombre es parecida (operador de SQL *like*) al contenido de la variable *nombre*.

```
SQLiteDatabase db = this.getReadableDatabase();
Cursor cursor = db.query( TABLA_ESTUDIANTES, null,
                          ESTUDIANTES_NOMBRE + " LIKE ?",
                          new String[]{ nombre }, null, null, null, null );

// Muestra todos los estudiantes por la consola
if ( cursor.moveToFirst() ) {
    do {
        Estudiante estudiante = new Estudiante( cursor.getInt( 0 ), cursor.getString( 1 ) );
        System.out.println( estudiante );
    } while ( cursor.moveToNext() );
}

cursor.close();
```

7 Mostrando el contenido de la tabla con un ListView

Una necesidad muy típica que puede surgir al crear una aplicación **Android** es conectar una tabla de la base de datos a un **ListView**. En realidad, las posibilidades que proporciona el sistema son todavía más poderosas, pues se puede conectar una consulta cualquiera con un **ListView**. La gran ventaja de utilizar estas posibilidades es la de no tener que preocuparnos de desbordar la memoria del dispositivo, pues es el propio sistema operativo el que lee de la base de datos a medida que necesita nuevas filas para mostrárselas al usuario.

Esta posibilidad se articula sobre dos pilares: una *layout* que le explique al sistema cómo visualizar la información en el **ListView**, y un **CursorAdapter** con el que conectar la consulta a la base de datos con el **ListView**. Existe ya un **CursorAdapter** preparado para las necesidades más básicas, en la forma de la clase **SimpleCursorAdapter**.

En lugar de crear el adaptador en **Activity**.*onCreate()*, se crea en **Activity**.*onStart()*, de manera que esté preparado no solo cuando comienza la aplicación, sino también cuando se restaura tras un tiempo inactiva. Como veremos, se crea con el parámetro relacionado con el cursor a **null**, lo cual es posible debido a que todavía no está disponible el mismo. Inmediatamente se invoca el método *actualizaEstudiantes()*, que crea el cursor y lo asigna al adaptador, abriendo automáticamente la base de datos. La contrapartida se encuentra en el método *onPause()*, que antes de hacer pasar la aplicación a estado inactivo, cierra el cursor y la base de datos.

```
public class MainActivity extends AppCompatActivity {
    // más cosas…

    @Override
    public void onStart()
    {
        super.onStart();

        // Configurar lista
        final ListView lvLista = this.findViewById( R.id.lvLista );

        this.adaptadorDB = new SimpleCursorAdapter(
                this,
                R.layout.lvlista_item,
                null,
                new String[] { DBManager.ESTUDIANTES_NOMBRE, DBManager.ESTUDIANTES_DNI },
                new int[] { R.id.lvLista_Item_Nombre, R.id.lvLista_Item_Dni }
        );

        lvLista.setAdapter( this.adaptadorDB );
        this.actualizaEstudiantes();
    }

    @Override
    public void onPause()
    {
        super.onPause();

        this.gestorDB.close();
        this.adaptadorDB.getCursor().close();
    }

    /** Actualiza el num. de elementos existentes en la vista. */
    private void actualizaEstudiantes()
    {
        final TextView lblNum = this.findViewById( R.id.lblNum );

        this.adaptadorDB.changeCursor( this.gestorDB.getEstudiantes() );
        lblNum.setText( String.format( Locale.getDefault(),"%d", this.adaptadorDB.getCount() ) );
    }

    private SimpleCursorAdapter adaptadorDB;
    private DBManager gestorDB;
}

public class DBManager extends SQLiteOpenHelper {
    public static final String TABLA_ESTUDIANTES = "estudiantes";
    // más cosas…

    /** Devuelve todos los estudiantes en la BD
      * @return Un Cursor con los estudiantes. */
    public Cursor getEstudiantes()
    {
        return this.getReadableDatabase().query( TABLA_ESTUDIANTES,
                null, null, null, null, null, null );
    }
}
```

Buscando un layout sencillo, que muestre la información sin excesivos adornos:

```
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" >

    <TextView android:id="@+id/lvLista_Item_Dni"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="0.25"
        android:textSize="18sp"
        android:textStyle="bold"/>

    <TextView android:id="@+id/lvLista_Item_Nombre"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="0.75"
        android:textStyle="italic"
        android:textSize="16sp" />
</LinearLayout>
```

Mientras con **ListAdapter<>** se empleaba el método *notifyDatasetChanged()*, con una base de datos es necesario volver a crear el cursor, de manera que se refleje cualquier cambio realizado. De ahí que en *actualizaEstudiantes()* (llamado después de cuaqluier cambio a la base de datos), se invoque a **CursorAdapter**.*changeCursor(c)*, siendo *c* un nuevo cursor creado en **DBManager.***getEstudiantes()*.

La parte relativa al cursor sobre la base de datos es la más importante, ya que es la que hace de puente entre la información en la consulta y la información a mostrar en el **ListView**.

```
… new SimpleCursorAdapter(
          this,
          R.layout.lvlista_item,
          null,
          new String[] { DBManager.ESTUDIANTES_NOMBRE, DBManager.ESTUDIANTES_DNI },
          new int[] { R.id.lvLista_Item_Nombre, R.id.lvLista_Item_Dni }
 );

 lvLista.setAdapter( this.adaptadorDB );
```

Como se puede observar sobre estas líneas, el constructor de la clase **SimpleCursorAdapter**(*cntxt, ly, cr, cols, tvs*) soporta los argumentos: *cntxt*, el contexto de la actividad donde se sitúa el **ListView** (en este caso, *lvLista*); *ly* es el *layout* a emplear con el **ListView**, mientras que *cr* es el cursor de la base de datos (que puede ser *null*, como en este caso, si no se dispone de él, hasta invocar **CursorAdapter**.*changeCursor(cr)*; un vector de cadenas *cols* con los nombres de las columnas a tomar de la consulta, y finalmente un vector de enteros *tvs* con los identificadores de los controles gráficos que van a visualizar la información dentro del *layout ly*. Nótese que estos últimos parámetros son vectores paralelos: la posición 0 del vector *cols* contiene el campo a visualizar en el control indicado en la posición 0 del vector *tvs*, la columna en la posición 1 con el control en la posición 1, y así sucesivamente.

8 Aplicaciones de ejemplo

Comprueba la sección de referencias para encontrar dos aplicaciones en *GitHub* que muestran un uso simple de bases de datos en Android:

- ListaCompra4: Se trata de la aplicación con la que se ha ido ejemplificando cada sección, esta vez cambiada para manejar una base de datos en la que almacenar los artículos a comprar.
- ConTacto: Un gestor de contactos simple (solo teléfonos), que utiliza una base de datos para almacenar los datos de los contactos.

La última está especialmente diseñada para ser pulida y expandida, favoreciendo la experimentación.

9 Referencias

- Documentación y recursos de Android para desarrolladores (accedido en sept. 2015)
  http://developer.android.com/
- Almacenamiento con SQLite
  http://developer.android.com/intl/es/guide/topics/data/data-storage.html#db
- SQLiteOpenHelper
  https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html
- SQLiteDatabase
  http://developer.android.com/intl/es/reference/android/database/sqlite/SQLiteDatabase.html
- Documentación oficial de SQLite
  https://www.sqlite.org/doclist.html
- SQL en la Wikipedia
  https://es.wikipedia.org/wiki/SQL
- Aplicación “Lista de la compra” con acceso a base de datos
  https://github.com/Baltasarq/ListaCompra4/
- Aplicación “Gestor de contactos telefónicos” con acceso a base de datos
  https://github.com/Baltasarq/AndroidConTacto