Bases de datos

SQLite

Datos estructurados

- La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados:
  - Bases de Datos SQLite: que abarca todas las tareas relacionadas con el almacenamiento de los datos propios de nuestra aplicación.
  - Content Providers: nos facilita la tarea de hacer visibles esos datos a otras aplicaciones y, de forma recíproca, de permitir la consulta de datos publicados por terceros desde nuestra aplicación.

SGBB para móviles más populares

Database	Type of data stored	License	Supported platforms
BerkeleyDB	relational, objects, key-value pairs, documents	AGPL 3.0	Android, iOS
Couchbase Lite	documents	Apache 2.0	Android, iOS
LevelDB	key-value pairs	New BSD	Android, iOS
SQLite	relational	Public Domain	Android, iOS, Windows Phone, Blackberry
UnQLite	key-value pairs, documents	BSD 2-Clause	Android, iOS, Windows Phone

- Más información

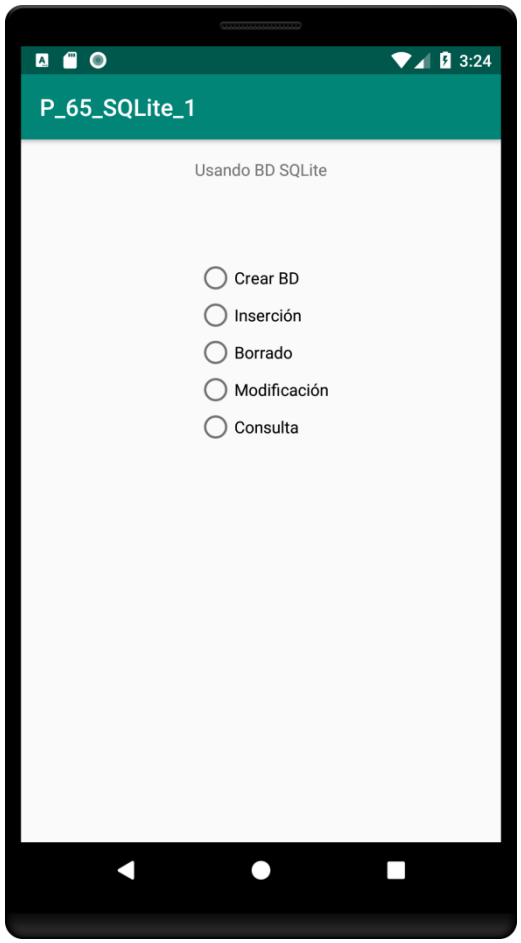
SQLite

- SQLite es una base de datos Open Source, muy popular en muchos dispositivos pequeños, como Android.
- Las ventajas que presenta utilizar SQLite es que no requiere configuración, no tiene un servidor de base de datos ejecutándose en un proceso separado y es relativamente simple su empleo ya que utiliza el lenguaje SQL (con pequeñas variaciones del estándar) para realizar consultas de los datos contenidos en sus tablas.
- SQL en SQLite

SQLite y Android

- Android incorpora de serie una herramienta muy potente para crear y gestionar bases de datos SQLite.
- Esa herramienta viene definida a través de la clase **SQLiteOpenHelper** y su función no es otra que la de comportarse como una plantilla sobre la que moldearemos la base de datos que necesitemos en nuestra aplicación.
- Por esta razón, para cada aplicación crearemos una clase que herede de SQLiteOpenHelper y personalizaremos la misma con nuestras necesidades: una estructura determinada, cambios a efectuar ante determinados eventos, etc.

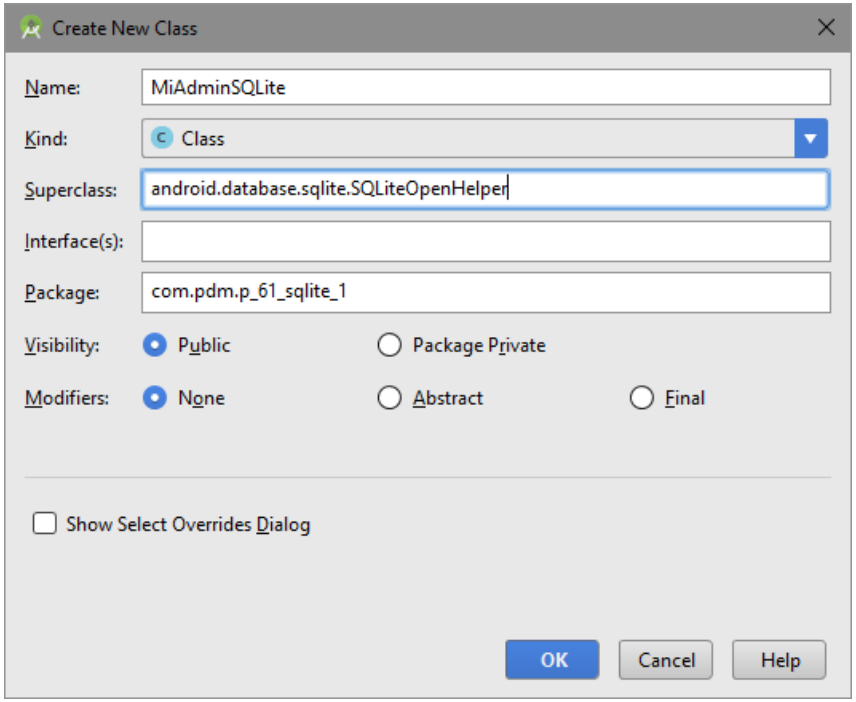
Proyecto P_70_SQLite_1



- Se desea contar con una BD SQLite de una agenda para guardar en una tabla los nombres y los email de personas.

Creación de la Clase SQLiteOpenHelper

- Creamos una nueva clase a la que llamaremos MiAdminSQLite (o el nombre que se quiera) y que **debe heredar de la clase SQLiteOpenHelper**, la cual nos ayudará a crear y conectarnos a la base de datos.
- La clase creada por el IDE tiene como mínimo:
  - 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** (por ejemplo añadir una tabla nueva o eliminar un campo de una de nuestras tablas) respectivamente.



class AdminSQLite extends SQLiteOpenHelper


```
public class MiAdminSQLite extends SQLiteOpenHelper {
   public MiAdminSQLite(@Nullable Context context, @Nullable String name, @Nullable SQLiteDatabase.CursorFactory factory, int version) {
      super(context, name, factory, version);
   }

   @Override
   public void onCreate(SQLiteDatabase sqLiteDatabase) {
      //Se ejecuta la sentencia SQL de creación de la tabla
      sqLiteDatabase.execSQL("CREATE TABLE usuarios (nombre TEXT primary key, email TEXT)");
   }

   @Override
   public void onUpgrade(SQLiteDatabase sqLiteDatabase, int oldVersion, int newVersion) {
      // NOTA: Por simplicidad del ejemplo aquí utilizamos directamente la opción
      //de eliminar la tabla anterior y crearla de nuevo vacía con el nuevo formato.
      sqLiteDatabase.execSQL("DROP TABLE IF EXISTS usuarios");
      sqLiteDatabase.execSQL("CREATE TABLE usuarios (nombre TEXT primary key, email TEXT)");
      // 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.
   }

}
```

En el constructor solo llamamos al constructor de la clase padre pasando los datos que llegan en los parámetros

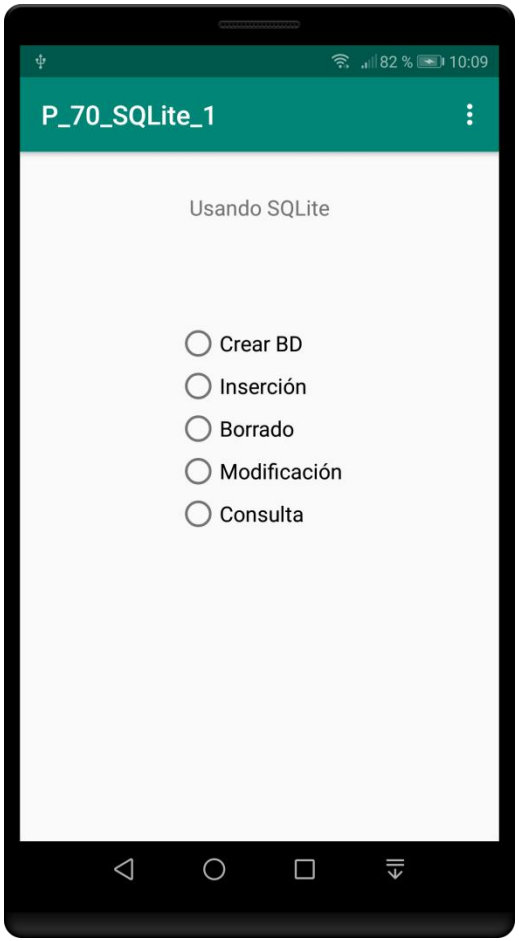
En el método onCreate procedemos a crear la tabla usuarios con los campos respectivos y definiendo el campo nombre como primary key

En el método onUpgrade procedemos a borrar la tabla usuarios y crear nuevamente la tabla (en este caso con la misma estructura pero podría ser otra en un caso más real)

Versión de la BD

- Android nos permite a la hora de diseñar una base de datos que la estructura de la misma pueda cambiar en caso de que nos surja la necesidad en un momento posterior de nuestro proyecto.
- Dicho de otra manera, Android nos permite tener distintas versiones de una misma base de datos y el método onUpgrade() se encargará de actualizar la base de datos en caso de que se produzca una migración de una a otra.
- El método onUpgrade() se encarga por tanto de hacer los cambios oportunos automáticamente cuando intentemos abrir una versión concreta de la base de datos que aún no existe.
- En nuestro caso de la base de datos, por ejemplo, podríamos tener una nueva versión de la misma que, en la tabla "usuarios", además del nombre y email, incluyera el teléfono.
- Una vez que conocemos esto, podemos explicar que la creación de un objeto de la clase SQLiteOpenHelper tiene las siguientes consecuencias:
  - Si la base de datos ya existe y su versión actual coincide con la indicada en el constructor simplemente se realizará la conexión con la misma.
  - Si la base de datos existe pero su versión actual es anterior a la indicada en el constructor, 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, entonces se llamará automáticamente al método onCreate() para crearla y se conectará con la base de datos creada con el nombre que especifiquemos en el constructor.

Proyecto SQLite_1: MainActivity

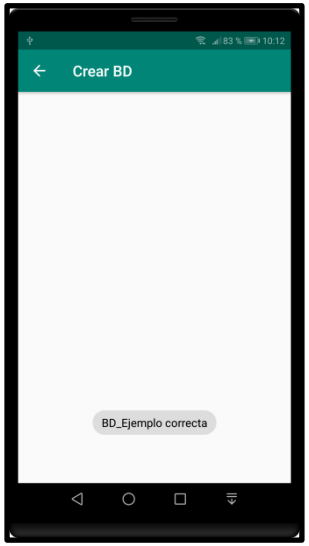



```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        RadioGroup rg = findViewById(R.id.radioGroup);
        final RadioButton r1 = findViewById(R.id.radioButton);
        final RadioButton r2 = findViewById(R.id.radioButton2);
        final RadioButton r3 =  findViewById(R.id.radioButton3);
        final RadioButton r4 =  findViewById(R.id.radioButton4);
        final RadioButton r5 =  findViewById(R.id.radioButton5);
        rg.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(RadioGroup group, int checkedId) {
                Intent intent = null;
                if (r1.isChecked())
                    intent = new Intent(getApplicationContext(), CrearActivity.class);
                if (r2.isChecked())
                    intent = new Intent(getApplicationContext(), AltaActivity.class);
                if (r3.isChecked())
                    intent = new Intent(getApplicationContext(), BajaActivity.class);
                if (r4.isChecked())
                    intent = new Intent(getApplicationContext(), ModificacActivity.class);
                if (r5.isChecked())
                    intent = new Intent(getApplicationContext(), ConsultarActivity.class);
                startActivity(intent);
            }
        });
    }
```

Crear BD


```
public class CrearBDActivity extends AppCompatActivity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      …
      getSupportActionBar().setDisplayHomeAsUpEnabled(true);
      MiAdminSQLite admin = new MiAdminSQLite (getApplicationContext(), "miBD.db", null, 1);
      SQLiteDatabase db = admin.getWritableDatabase();
      //Si hemos abierto correctamente la base de datos
      if(db != null)
      {
         Toast.makeText(this, "BD_Ejemplo correcta", Toast.LENGTH_SHORT).show();
         //Cerramos la base de datos
         db.close();
      }
   }
}
```

No necesitamos tener una Activity propia para la Creación de la BD. Lo he hecho así por clarificar conceptos, pero lo lógico sería que este código formase parte del método onCreate de la MainActivity.



Explicación crear

- Lo primero que hacemos en este método es crear un objeto de la clase que planteamos anteriormente y le pasamos al constructor el contexto de la Activity actual, "miBD.db" (es el nombre de la base de datos que crearemos en el caso que no exista, las extensiones más utilizadas son .db o .sqlite, pero podría ser la que quisiéramos), luego pasamos null (no vamos a usar CursorFactory, de momento) y un 1 indicando que es la primera versión de la base de datos (en caso que cambiemos la estructura o agreguemos tablas por ejemplo podemos pasar un 2 en lugar de un 1 para que se ejecute el método onUpgrade donde indicamos la nueva estructura de la base de datos):

  MiAdminSQLite admin= new MiAdminSQLite (getApplicationContext(), "miBD.db", null, 1);

- Después de crear un objeto admin de la clase AdminSqLite procedemos a crear un objeto de la clase SQLiteDataBase llamando al método getWritableDatabase (la base de datos se abre en modo lectura y escritura).

  SQLiteDatabase bd=admin.getWritableDatabase();

Clase SQLiteDatabase y métodos getReadableDatabase() y getWritableDatabase()

- Una vez que tengamos un objeto de la clase, podremos recuperar la base de datos asociada, representada mediante la clase SQLiteDatabase, a través de los métodos heredados **getReadableDatabase()** y **getWritableDatabase()**, en función de si necesitamos únicamente consultar la base de datos o si además necesitamos realizar cambios o modificaciones sobre la misma.

Patrón Singleton

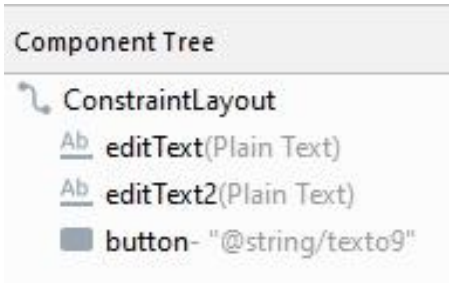
- A menudo, la base de datos SQLite se utilizará en toda la aplicación; dentro de servicios, actividades, fragmentos, etc. Por esta razón, las buenas prácticas aconsejan a aplicar el patrón Singleton en las instancias SQLiteOpenHelper para evitar pérdidas de memoria y reasignaciones innecesarias.:

```
public class MiAdminSQLite extends SQLiteOpenHelper {
   private static MiAdminSQLite sInstance;
   public static synchronized MiAdminSQLite getInstance(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)
        {
      if (sInstance == null) {
         sInstance = new MiAdminSQLite(context.getApplicationContext(),name,factory,version);
      }
      return sInstance;
   }
 public MiAdminSQLite(@Nullable Context context, @Nullable String name, @Nullable SQLiteDatabase.CursorFactory factory, int version) {
      super(context, name, factory, version);
   }
...
```

- El método estático getInstance() asegura que sólo un objeto de tipo MiAdminSQLite existirá en un momento dado. Si el objeto sInstance no se ha inicializado, se creará uno. Si uno ya ha sido creado entonces simplemente será devuelto.
- Por tanto, no se debe inicializar el objeto utilizando **MiAdminSQLite admin = new MiAdminSQLite (getApplicationContext(), "miBD.db", null, 1);.**, hay que utilizar

  MiAdminSQLite admin = MiAdminSQLite.getInstance(getApplicationContext(), "miBD.db", null, 1);
- Patrón Singleton

Insertar registros




```
public class AltaBDActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    …
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    final EditText nombre= findViewById(R.id.editText);
    final EditText email= findViewById(R.id.editText2);
    Button button= findViewById(R.id.button);
    button.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        MiAdminSQLite admin = MiAdminSQLite.getInstance(getApplicationContext(),
        "miBD.db", null, 1);
        SQLiteDatabase bd=admin.getWritableDatabase();
        ContentValues registro=new ContentValues();
        registro.put("nombre", nombre.getText().toString());
        registro.put("email", email.getText().toString());
        bd.insert("usuarios", null, registro);
        bd.close();
        Toast.makeText(getApplicationContext(), "Se cargaron los datos de la persona",
      Toast.LENGTH_SHORT).show();
        nombre.setText("");
        email.setText("");
      }
    });
  }
}
```

Observa que si el nombre (PK) ya existe no nos lo advierte. Deberíamos mejorar (primero busca y si no existe graba)!

Explicación insertar

- Instanciamos los objetos de las clases AdminSQLite y SQLiteDatabase necesarios

```
MiAdminSQLite admin = MiAdminSQLite.getInstance(getApplicationContext(), "miBD.db", null, 1);
SQLiteDatabase bd=admin.getWritableDatabase();
```

- Creamos un objeto de la **clase ContentValues** y mediante **su método put** inicializamos los campos a cargar. 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.

```
ContentValues registro=new ContentValues();
registro.put("nombre", nombre.getText().toString());
registro.put("email", email.getText().toString());
```

- Seguidamente llamamos al **método insert()** de la clase SQLiteDatabase pasando en el primer parámetro el nombre de la tabla, como segundo parámetro un null y por último el objeto de la clase ContentValues

```
bd.insert("usuarios", null, registro);
```

- Por último, cerramos la base de datos

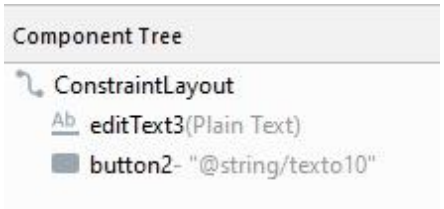
```
bd.close();
```

Método insert() de la clase SQLiteDatabase

- public long insert (String table, String nullColumnHack, ContentValues values);
- 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.

Suprimir registros


```
public class BajaBDActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    …
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    final EditText nombre = findViewById(R.id.editText3);
    Button button = findViewById(R.id.button2);
    button.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        MiAdminSQLite admin = MiAdminSQLite.getInstance(getApplicationContext(), "miBD.db", null, 1);
        SQLiteDatabase bd = admin.getWritableDatabase();
        String buscado = nombre.getText().toString();
        int cant = bd.delete("usuarios", "nombre='" + buscado + "'", null);
        if (cant > 0)
          Toast.makeText(getApplicationContext(), "Se borró la persona con dicho nombre",
              Toast.LENGTH_SHORT).show();
        else
          Toast.makeText(getApplicationContext(), "No existe una persona con dicho nombre",
              Toast.LENGTH_SHORT).show();
        bd.close();
        nombre.setText("");
      }
    });
  }
}
```



Explicación suprimir

- Creamos los objetos de las clases AdminSQLite y SQLiteDatabase necesarios

```
MiAdminSQLite admin = MiAdminSQLite.getInstance(getApplicationContext(), "miBD.db", null, 1);
SQLiteDatabase bd=admin.getWritableDatabase();
```

- Para borrar uno o más registros la clase SQLiteDatabase tiene el **método delete()** al que le pasamos en el primer parámetro el nombre de la tabla y en el segundo la condición que debe cumplirse para que se borre la fila de la tabla.

```
int cant=bd.delete("usuarios", "nombre='"+buscado+"'",null);
```

- El método devuelve un entero que indica la cantidad de registros borrados

```
if (cant>0)
    Toast.makeText(this, "Se borró la persona con dicho nombre",
        Toast.LENGTH_SHORT).show();
else
    Toast.makeText(this, "No existe una persona con dicho nombre",
        Toast.LENGTH_SHORT).show();
```

- Por último, cerramos la base de datos

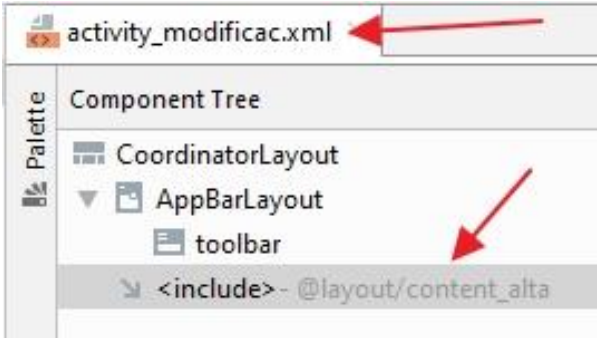
```
bd.close();
```

Método delete() de la clase SQLiteDatabase

- public int delete (String table, String whereClause, String[] whereArgs);
- Para eliminar los registros de la base de datos que cumplan una condición.
- Este método recibe tres parámetros:
  - el primero de ellos será el nombre de la tabla,
  - el segundo la condición *WHERE* de la sentencia SQL, si no necesitáramos ninguna condición, podríamos dejar como *null* este parámetro (y se borrarán todos los registros, recuerda la canción!).
  - y el tercero lo obviaremos de momento ya que tan sólo se hace necesario en casos de utilizar argumentos dentro de las condiciones de la sentencia SQL, en cualquier otro caso pasaremos con valor *null* este parámetro.

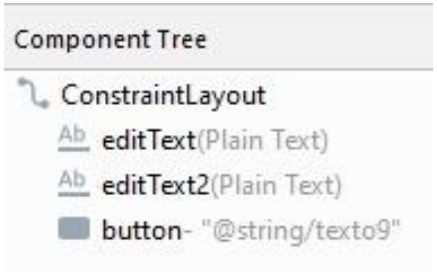
"Truco" de optimización

- Las pantallas de altas y modificaciones (en este ejemplo) son iguales solo difieren en el título que aparece en la AppBar.
- Por tanto, en este ejemplo (repito! ☺), podemos borrar el layout content_modific e incluir en el layout activity_modificac el de content_alta.
- Aunque, lo más correcto sería trabajar con fragmentos!



Modificar registros

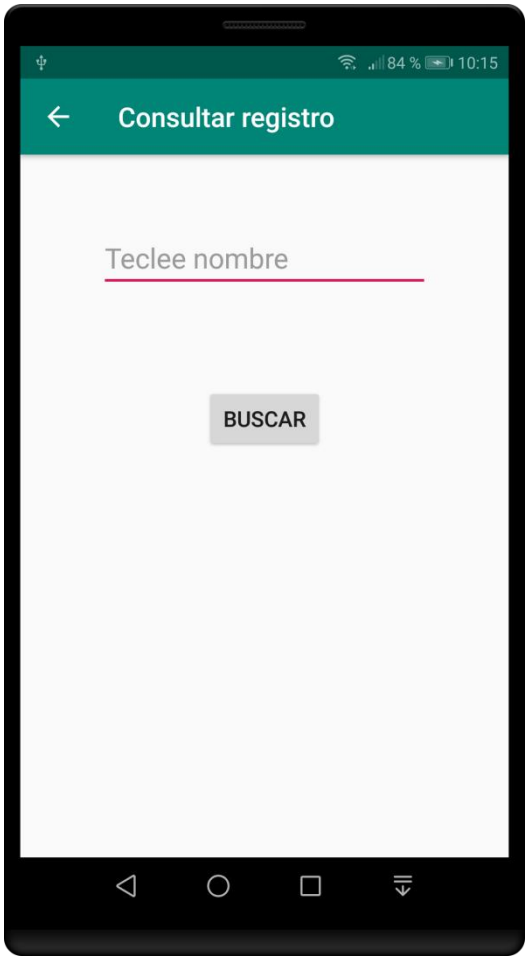

```
public class ModificacionBDActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      …
      getSupportActionBar().setDisplayHomeAsUpEnabled(true);
      final EditText nombre=findViewById(R.id.editText);
      final EditText nuevomail=findViewById(R.id.editText2);
      Button button=findViewById(R.id.button);
      button.setOnClickListener(new View.OnClickListener() {
         @Override
         public void onClick(View v) {
            MiAdminSQLite admin = MiAdminSQLite.getInstance(getApplicationContext(), "miBD.db", null, 1);
            SQLiteDatabase bd = admin.getWritableDatabase();
            String buscado=nombre.getText().toString();
            ContentValues registro=new ContentValues();
            registro.put("email",nuevomail.getText().toString());
            int cant = bd.update("usuarios", registro, "nombre='"+buscado+"'",  null);
            bd.close();
            if (cant>0)
               Toast.makeText(getApplicationContext(), "Se modificaron los datos", Toast.LENGTH_SHORT).show();
            else
               Toast.makeText(getApplicationContext(), "No existe una persona con dicho nombre", Toast.LENGTH_SHORT).show();

         }
      });…
```

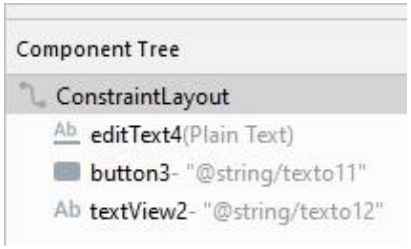


Método update() de la clase SQLiteDatabase

- public int update (String table, ContentValues values, String whereClause, String[] whereArgs);
- Para modificar los registros de la base de datos que cumplan una condición.
- Este método recibe cuatro parámetros:
  - el primero de ellos será el nombre de la tabla,
  - el segundo los nuevos valores del registro
  - el tercero la condición *WHERE* de la sentencia SQL, si no necesitáramos ninguna condición, podríamos dejar como *null* este parámetro (y se modificarían todos los registros!).
  - y el cuarto lo obviaremos de momento ya que tan sólo se hace necesario en casos de utilizar argumentos dentro de las condiciones de la sentencia SQL, en cualquier otro caso pasaremos con valor *null*.

Consultar registros






```
public class ConsultaBDActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    …
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
     final EditText nombre = findViewById(R.id.editText4);
    final TextView email = findViewById(R.id.textView2);
    email.setText("");
    Button button = findViewById(R.id.button3);
    button.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        MiAdminSQLite admin = MiAdminSQLite.getInstance(getApplicationContext(), "miBD.db", null, 1);
        SQLiteDatabase bd = admin.getWritableDatabase();
        String buscado = nombre.getText().toString();
        Cursor fila = bd.rawQuery("select email  from usuarios where nombre='" + buscado + "'", null);
        if (fila.moveToFirst())
          email.setText(fila.getString(0));
        else
          email.setText("No existe una persona con dicho nombre");
        fila.close();
        bd.close();
      }
    });
  }
}
```

Explicación consulta

- Definimos una variable de la **clase Cursor** y la inicializamos con el valor devuelto por el método llamado **rawQuery()**.

```
Cursor fila =
bd.rawQuery("select email  from usuarios where nombre=' "+ nombre + " ' ", null);
```

- La clase Cursor almacena en este caso una fila (recuerda que nombre es PK) en caso que hayamos tecleado un nombre existente en la tabla usuarios o cero filas en caso contrario. El método **moveToFirst()** de la clase Cursor y retorna true en caso de existir una persona con el nombre tecleado.

```
if (fila.moveToFirst())
```

- Para recuperar los datos propiamente dichos que queremos consultar llamamos al método **getString()** y le pasamos la posición del campo de la consulta a recuperar (comienza a numerarse en cero, en este ejemplo la columna cero representa el campo email del select fijado)

```
tv2.setText(fila.getString(0));
```

Métodos rawQuery() y query() de la clase SQLiteDatabase

- public Cursor rawQuery (String sql, String[] selectionArgs);

  El resultado se devuelve en una variable de tipo Cursor que es una estructura pensada para almacenar los resultados de las consultas, similar a una lista y que se puede recorrer fácilmente.

- public Cursor query (String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit);

  Permite especificar de una forma mas clara todo lo necesario para completar la consulta.

Métodos moveToFirst() y moveToNext()

- Para recorrer los resultados devueltos en la variable Cursor, tenemos a nuestra disposición varios métodos pero destacamos dos que recorren el mismo 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.

Método db.exec

- Además de los métodos insert, delete y update de la clase SQLiteDatabase ya vistos, también puede utilizarse el método exec que permite ejecutar directamente el código SQL que le pasemos como parámetro **siempre que éste no devuelva resultados**:
  - Insercion de registros en la tabla

    db.execSQL("INSERT INTO usuarios(nombre, email) VALUES ('Pepe', 'correo@tele.es')");
  - Actualizacion de registros de la tabla

    db.execSQL("UPDATE usuarios SET email='otro@qq.es' WHERE nombre='Pepe'");
  - Eliminacion de un registro de la tabla

    db.execSQL("DELETE FROM usuarios WHERE nombre='Pepe'");

Utilizar argumentos dentro de las condiciones de la sentencia SQL

- 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.
- Esto no son más que partes variables de la sentencia SQL que pasaremos 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 siguientes:

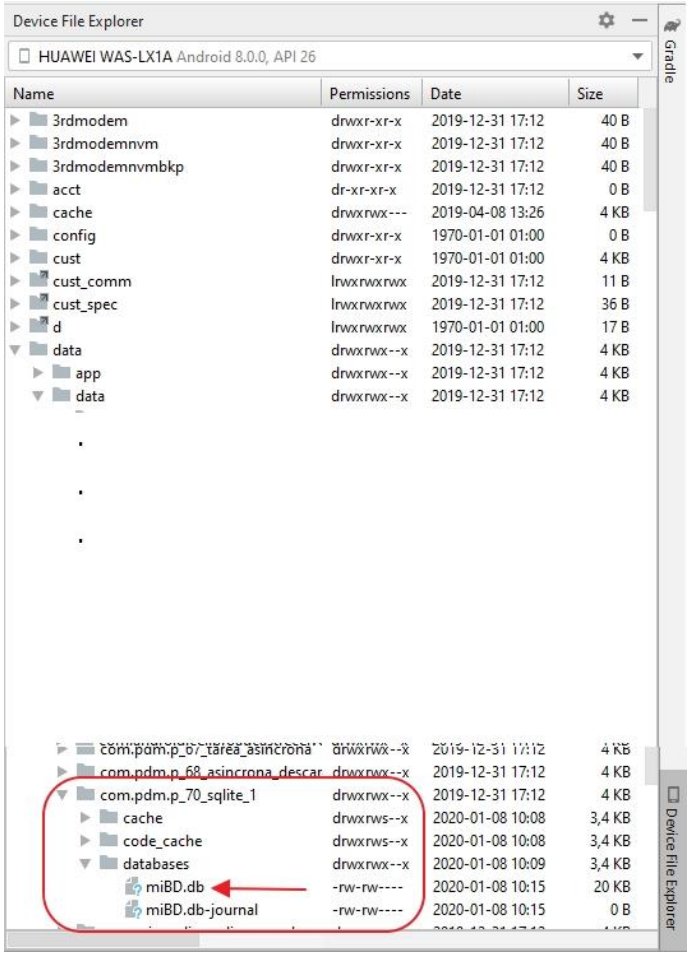
```
//Eliminar un registro con execSQL(), utilizando argumentos
String[] args = new String[]{"usu1"};
db.execSQL("DELETE FROM usuarios WHERE usuario=?", args);

//Actualizar dos registros con update(), utilizando argumentos
ContentValues valores = new ContentValues();
valores.put("email","usu1_nuevo@email.com");
String[] args = new String[]{"Pepe", "Juan"};
db.update("usuarios", valores, "usuario=? OR usuario=?", args);
```

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

Ruta de almacenamiento de la BD

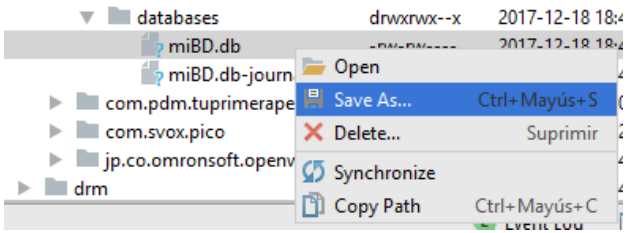
- Todas las bases de datos SQLite creadas por aplicaciones Android se almacenan en la memoria del teléfono en un fichero con el mismo nombre que la base de datos y situado en una ruta que sigue el siguiente patrón:

  /data/data/paquete.java.de.la.aplicacion/databases/nombre_base_datos



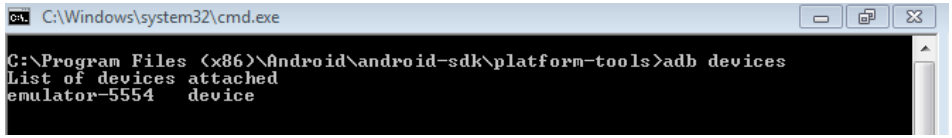
Trabajar directamente con la BD

- Para comprobar que tanto las tablas creadas como los datos insertados también se han incluido correctamente en la base de datos podemos recurrir a dos posibles vías:



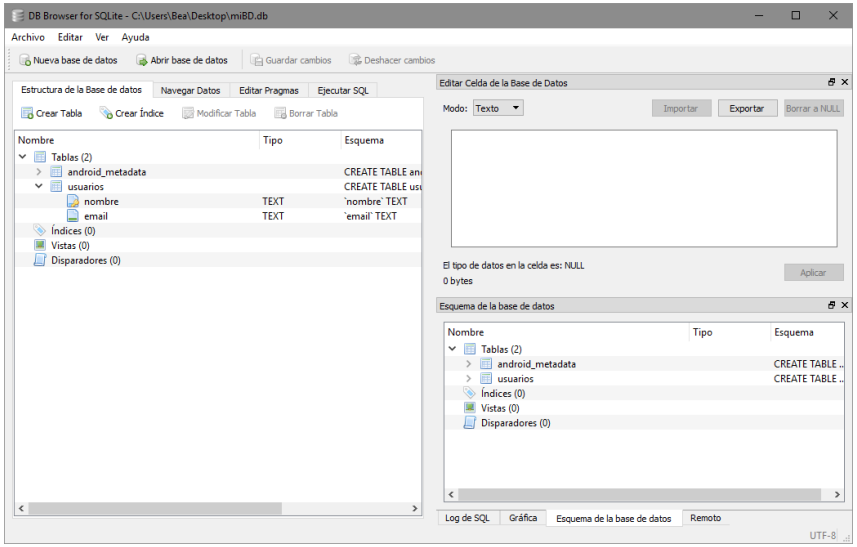
  - Transferir la base de datos a nuestro ordenador y consultarla con cualquier administrador de bases de datos SQLite.

  - 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.



Administradores de BD SQLite

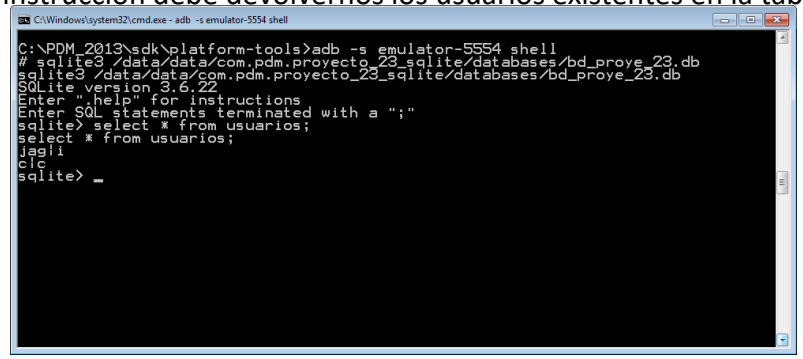
- DB Browser para SQLite



- SQLiteAdministrator
- sqlitestudio
- Sqlitedeveloper
- Extensiones instaladas en los navegadores

adb

- Con el emulador de Android aún abierto o el dispositivo real conectado, debemos abrir una consola de MS-DOS (o bien utilizar el terminal de AS) y utilizar la utilidad adb.exe (*Android Debug Bridge*) situada en la carpeta platform-tools del SDK de Android.
- 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" (es el identificador del emulador).
- Tras conocer el identificador de nuestro emulador, vamos a acceder a su shell mediante el comando **adb -s identificador-del-emulador 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.pdm.p_65_sqlite_1/databases/miBD.db**
- 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.
- Para comprobar que existe la tabla Usuarios y que se han insertado los registros de ejemplo haremos la siguiente consulta: **SELECT * FROM usuarios;**
- Si todo es correcto esta instrucción debe devolvernos los usuarios existentes en la tabla.



Prácticas propuestas

- Realiza la hoja de ejercicios Ejercicios_16_SQLite