Bases de datos SQLite en Android

Aprende todo lo necesario para EMPEZAR a usar bases de datos SQLite en tus proyectos Android.

¿Por qué usar bases de datos SQLite?

En Android existen varias formas, sobre cómo guardar datos de forma persistente.

Entre los métodos más comunes encontramos:

* Bases de datos SQLite
* SharedPreferences
* Almacenamiento en disco

**Guardar datos de forma persistente** significa que los datos persisten incluso si se apaga y se vuelve a prender el dispositivo.

Generalmente cuando cerramos una aplicación, Android va a liberar los recursos que ésta tenía asignados.

Nuestras variables que estaban definidas dejarán de existir. Es por eso que necesitamos guardar estos datos si son importantes.

Y, hay que tener en cuenta que estos datos se están registrando de forma local.

Es decir, estas 3 opciones almacenan los datos en el mismo dispositivo, sin requerir de una conexión a internet.

Si lo que estás buscando es registrar datos en una base de datos online, lo mejor es que [veas este tutorial](https://programacionymas.com/blog/consumir-una-api-usando-retrofit).

¿Cuándo usar SQLite en Android?

Las preferencias son muy fáciles de usar, y resultan muy prácticas.

Si tenemos que registrar datos simples como, el idioma que prefiere el usuario, el skin que quiere usar, fecha y hora de la última acción que realizó, podemos usar SharedPreferences sin problema alguno.

Las preferencias nos permiten guardar datos bien puntuales.

A diferencia de ello, las bases de datos nos permiten guardar información más elaborada.

Si tienes experiencia usando MySQL, SQLServer, Oracle, PostgreSQL o algún otro gestor de bases de datos relacionales, esto te resultará muy familiar.

Sucede que SQLite nos ofrece también la posibilidad de crear bases de datos relacionales.

Puedes imaginar a SQLite como un MySQL pequeño, o una simplificación de los antes mencionados.

Se escogió esta opción porque nos permitirá crear bases de datos muy ligeras, de forma que nuestra app podrá interactuar con los datos incluso desde dispositivos con pocos recursos.

Si solo necesitas guardar un entero, una cadena u otro valor en específico, basta con usar SharedPreferences.

Pero, si en cambio requieres de una estructura para tus datos, SQLite es una muy buena opción.

Para darte un ejemplo: Ahora mismo, yo he terminado de desarrollar una app. Esta app se conecta con una API, que le permite consumir y guardar datos. Todo OK. Pero ahora me comentan que se necesita usar la app en modo offline. SQLite es la solución.

Lo que vamos a hacer en este tutorial

A modo de ejemplo, vamos a:

* Crear una base de datos con una tabla llamada "areas".
* Definir las columnas (nombres y tipos de dato) que usará esta tabla.
* Implementar las operaciones de registro, edición y eliminación de filas en esta tabla.
* Guardar datos, editarlos y recuperarlos usando las operaciones definidas.

Paso 1: Definir un esquema inicial

Vamos a trabajar con una entidad "Area", que tendrá solo 2 datos: un ID y un nombre.

La idea es entender cómo funciona todo.

Si entendemos todo a la perfección entonces luego será sencillo usar tablas con más datos, y trabajar con varias tablas que están relacionadas.

He aquí nuestra clase Area:

public class Area {

private int id;

private String name;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Espera, ¿no es esto una clase común y corriente?

Sí que lo es. Esta clase únicamente expresa los datos que queremos registrar.

Nuestra base de datos tendrá una tabla, y las columnas de esta tabla tendrán un nombre y un tipo de dato.

Esto tenemos que expresarlo a través de un Contract.

Un contrato es simplemente una clase que nos permitirá guardar allí algunas constantes.

¿Eso significa que no es obligatorio su uso?

Exacto. Podríamos no definir constantes y usar directamente los valores en nuestras sentencias SQL.

El punto es que, si necesitamos hacer algún cambio en el futuro, vamos a tener que hacer cambios aquí y allá.

Al usar constantes, solo modificamos el valor de estas constantes, y todas nuestras sentencias usarán los nuevos valores.

He aquí el contrato:

public final class AreaContract {

private AreaContract() {}

public static class AreaEntry implements BaseColumns {

public static final String TABLE\_NAME = "areas";

public static final String COLUMN\_ID = "id";

public static final String COLUMN\_NAME = "name";

}

}

**Acerca de BaseColumns:**

Seguro notaste que la clase interna AreaEntry implementa una interfaz.

BaseColumns es una interfaz que define un atributo \_ID.

Este identificar lo genera Android, y es muy distinto al ID que vamos a guardar nosotros (el ID que guardamos por nuestra cuenta se asocia al ID que tiene cada área según nuestra API).

Es por eso que definimos el campo COLUMN\_ID por nuestra cuenta. Y usamos BaseColumns para tener nuestro código acorde con el estándar de Android.

Nos será útil posteriormente cuando usemos otras clases que requieran implementar esta interfaz.

Paso 2: Crear la base de datos

En el paso anterior hemos definido el esquema que tendrá nuestra base de datos.

Ahora vamos a escribir el código que se encargará de crear verdaderamente la base de datos.

Para ello vamos a crear una clase llamada MyDbHelper.

Esta clase va a heredar de SQLiteOpenHelper, que es una clase abstracta, que nos ayudará a gestionar nuestra base de datos.

Esta clase abstracta se encargará de usar la base de datos si ya existe, crearla si aun no existe, y actualizarla cuando sea necesario.

Gracias a que esta clase lleva un control del estado de nuestra base de datos, nosotros no tenemos que preocuparnos sobre cuándo ejecutar cada cosa.

Pero sí tenemos que decirle cómo hacerlo:

public class MyDbHelper extends SQLiteOpenHelper {

private static final int DATABASE\_VERSION = 1;

private static final String DATABASE\_NAME = "pym.db";

public MyDbHelper(Context context) {

super(context, DATABASE\_NAME, null, DATABASE\_VERSION);

}

public void onCreate(SQLiteDatabase db) {

db.execSQL("CREATE TABLE " + AreaEntry.TABLE\_NAME + " (" +

AreaEntry.\_ID + " INTEGER PRIMARY KEY," +

AreaEntry.COLUMN\_NAME + " TEXT)");

}

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

db.execSQL("DROP TABLE IF EXISTS " + AreaEntry.TABLE\_NAME);

onCreate(db);

}

}

Aquí encontramos:

* Un atributo entero DATABASE\_VERSION, que indica la versión de nuestra base de datos.
* Un atributo cadena DATABASE\_NAME que indica con qué nombre de archivo se guardará nuestra base de datos.
* Una implementación de los métodos onCreate y onUpgrade.

Partiendo de esta información debemos tener en cuenta que:

* El método onCreate se encargará de crear el esquema de nuestra base de datos (tablas, claves primarias, claves foráneas).
* Si altermos el esquema de nuestra base de datos, debemos aumentar el valor de DATABASE\_VERSION, para que nuestro helper ejecute onUpgrade la próxima vez.
* El método onUpgrade debería contener cierta lógica para migrar los datos existentes de una versión a otra.

En el ejemplo que estamos viendo, las Areas se obtienen desde una API, y quedan almacenadas para que la aplicación pueda trabajar de modo offline.

Si en algún momento cambia el esquema de la base de datos, no hay problema en borrar la tabla y volver a cargar la información.

Estamos considerando el escenario más simple. En un futuro post vamos a ver circunstancias que requieran de un método onUpgrade más elaborado.

¿Por qué no usamos una clave primaria autoincremental?

En este caso queremos que **el ID que devuelve la API** para cada Area sea igual al **ID almacenado de forma local**.

El esquema de nuestra base de datos es relativo a las operaciones que pensamos realizar.

Por ejemplo:

* Si estamos desarrollando una aplicación que funcionará únicamente de modo offline, tiene sentido que los ID de nuestras tablas sean todas auto-incrementales.
* Si tenemos una tabla de "reportes realizados por usuarios", y queremos que nuestra app funcione tanto de modo online como offline, podríamos tener 2 tablas, una para guardar los reportes descargados desde la API, y otra para guardar reportes que se han creado de modo offline y que hacen falta publicar en nuestro servidor online.
* Pero ese no es el único camino, otros van a preferir tener una columna adicional (que existe solo de forma local) para diferenciar los reportes que se han sincronizado de los que aún no. Aquí se podría tener un ID autoincremental como clave primaria (muy aparte del ID usado de forma online), y una columna adicional contendría el ID que tiene el reporte en el servidor. Si es null significaría que aun no se ha subido, y si tiene un valor quiere decir que ya se sincronizó con el servidor y tiene un ID online.

En fin. No hay que desviarnos mucho del tema.

Nuestro objetivo por ahora es explorar las operaciones básicas.

Paso 3: Registrar datos usando ContentValues

Veamos ahora cómo insertar datos.

Para hacer ello necesitamos:

* Instanciar nuestro DbHelper (pasándole como parámetro el contexto en que nos encontramos).
* Obtener una instancia de SQLiteDatabase en modo escritura apartir de nuestro helper.
* Crear un ContentValues y cargarlo con la información que deseamos registrar.
* Hacer efectiva la operación usando el método insert desde nuestra instancia SQLiteDatabase.

Si quieres que una tabla tenga valores desde su creación, entonces debes hacer la operación de inserción en el método **onCreate**. En ese caso los 2 primeros pasos se omiten porque ya se tiene a la mano una instancia de **SQLiteDatabase**.