2025

Programación Multimedia y Dispositivos Móviles (PMDM)

[ANDROID: CONEXIÓN A BASE DE DATOS LOCAL]

Tabla de contenido

1. Introducción

Vamos a ver el uso de base de datos SQLite en Android para trabajar sobre base de datos local, es decir, Android almacena la base de datos en una carpeta privada de tu aplicación (directorio *data/data/APP_Name/databases/DATABASE_NAME*). Los datos están seguros porque, de forma predeterminada, esta área no es accesible para otros usuarios y aplicaciones.

Las APIs que necesitarás para utilizar una base de datos en Android están disponibles en el paquete **android.database.sqlite**.

Para crear, actualizar y otras operaciones hay que crear una clase o subclase **SQLiteOpenHelper**. Después de extender SQLiteOpenHelper, habrá que implementar sus métodos **onCreate()**, **onUpgrade()** y **constructor**. También se pueden implementar los métodos ***onDowngrade()*** y ***onOpen()***, pero no son obligatorios.

2. SQLiteOpenHelper

Es una clase auxiliar para administrar la creación de bases de datos y la administración de versiones.

Proporciona dos métodos:

- onCreate(SQLiteDatabase db)
- onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)

Es responsable de:

- abrir la base de datos si existe.
- crear la base de datos si no existe. SQLiteOpenHelper sólo requiere el nombre de la base de datos para poder crearla.
- actualizar la base de datos si es necesario.

La clase SQLiteOpenHelper contiene un conjunto útil de APIs para administrar la base de datos. El sistema realiza operaciones de larga duración para crear y actualizar la base de datos sólo cuando es necesario y no durante el inicio de la aplicación. Para obtener una referencia a la base de datos lo único que hay que hacer es llamar a **writableDatabase** o **readableDatabase**.

El método **onCreate() se llama sólo una vez durante todo el ciclo de vida de la aplicación**. Se llamará cada vez que haya una primera llamada a readableDatabase o writableDatabase. Por lo tanto, la clase SQLiteOpenHelper llama al método onCreate() **después de crear la base de datos** y crea una instancia del objeto SQLiteDatabase. El nombre de la base de datos se pasa en la llamada al constructor.

El método **onUpgrade()** sólo se llama cuando hay una **actualización en la versión** existente. Para actualizar una versión tenemos que incrementar el valor de la variable versión pasada en el constructor de la superclase.
En este método se pueden escribir consultas para realizar cualquier acción necesaria. En la mayoría de las ocasiones lo que se hace es **eliminar las tablas existentes y llamar de nuevo al método onCreate()** para que las vuelva a crear. No es obligatorio y todo depende de las necesidades que se tenga.
Hay que cambiar la versión de la base de datos **si se agrega un nuevo campo en la tabla**. Si existe el requisito de no perder los datos existentes en la tabla, se puede escribir una consulta de modificación de la tabla en el método onUpgrade().

Una posible **implementación** de la clase **SQLiteOpenHelper** sería:

```
class BBDD_Helper(context: Context, name: String, factory:
CursorFactory?, version: Int): SQLiteOpenHelper(context, name, factory,
version){
      private val SQL_CREATE_ENTRIES = "CREATE TABLE login(id INT
      PRIMARY KEY, email VARCHAR(50), password VARCHAR(50))"
      private val SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS login"

      override fun onCreate(db: SQLiteDatabase){
            db.execSQL(SQL_CREATE_ENTRIES)
      }

      override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
      newVersion: Int){
            db.execSQL(SQL_DELETE_ENTRIES)
            this.onCreate(db)
      }
}
```

Para acceder a la base de datos hay que **crear una instancia** de la subclase de SQLiteOpenHelper. Por ejemplo:

```
      val conexion = BBDD_Helper(this, "usuario", null, 1)
```

3. Operaciones sobre la base de datos

INSERTAR INFORMACIÓN

Para insertar datos en la base de datos, lo primero que hay que hacer es abrir la base de datos para escritura a través de **writableDatabase** y llamar al método **insert()** pasándole un **objeto ContentValues**.

Una vez acabemos las operaciones sobre la base de datos hay que **cerrarla** llamando al método **close()**.

Ejemplo:

```
val conexion = BBDD_Helper(this, "usuario", null, 1)
// Obtiene la base de datos en modo escritura
val bd = conexion.writableDatabase
// Crea un mapa de valores donde los nombres de las columnas son
// las claves
val registro = ContentValues()
registro.put("id", 1)
registro.put("email", "patriciasfo@educastur.org")
registro.put("password", "prueba")
// Otra forma de crear el registro
/*
      val registro = ContentValues().apply{
            put("id", 1)
            put("email", "patriciasfo@educastur.org")
            put("password", "prueba")
      }
*/
// Inserta una nueva fila, retornando la clave primaria de la
// nueva fila
val nuevoId = bd.insert("login", null, registro)
bd.close()
```

Los **argumentos** del método **insert()** son:

- El primer argumento es simplemente **el nombre de la tabla**.
- El segundo, le indica **qué hacer** en caso de que el **ContentValues** esté **vacío**. Está vacío si no se añade ningún valor con ***put***:
    - Si se especifica **null**: NO se insertará una fila cuando no haya valores.
    - Si se especifica **el nombre de una columna**: se inserta una fila y establece el valor de esa columna como nulo.
- El tercer argumento es **el registro**.

El **retorno** del método **insert()** es:

- El ID o clave primaria de la fila recién creada
- -1 si hubo algún error al insertar los datos. Esto puede suceder si hay conflicto con los datos preexistentes en la base de datos.

CONSULTAR INFORMACIÓN

Para leer información de una base de datos, lo primero que hay que hacer es abrir la base de datos para lectura a través de **readableDatabase** y llamar al método **query()** o **rawQuery()** pasándole los criterios de selección y/o columnas deseadas. Los resultados de la consulta se devuelven en un objeto **Cursor**.

Una vez acabemos las operaciones sobre la base de datos hay que **cerrarla** llamando al método **close()**.

Ejemplo con query():

```
val conexion = BBDD_Helper(this, "usuario", null, 1)
val bd = conexión.readableDatabase
// Definir una proyección con las columnas de la base de datos
// que se desean recuperar
val proyeccion = arrayOf("id", "email", "password")
// Filtro de resultados
val seleccion = "email = ?"
val seleccioArgs = arrayOf("patriciasfo@educastur.org")
// Cómo se desean ordenar los resultados
val orden = "password DESC"
val filas = bd.query(
      "login",      // La tabla
      proyeccion,   // El array con las columnas a retornar. Si
                    // queremos todos los datos -> poner null.
      seleccion,    // Las columnas de la cláusula WHERE
      seleccioArgs, // Los valores de la cláusula WHERE
      null,         // Para no agrupar las filas
      null,         // Para no filtrar los grupos de filas
      orden         // El orden de los registros
)
```

El tercer (selection) y cuarto (selectionArgs) argumento de la query se combinan para crear la cláusula WHERE.

Como los argumentos se proporcionan por separado de la consulta de selección, se escapan antes de combinarse, lo que hace que las instrucciones de selección sean inmunes a la inyección de SQL.

Ejemplo con rawQuery():

```
val conexion = BBDD_Helper(this, "usuario", null, 1)
val bd = conexion.readableDatabase
val filas = bd.rawQuery("select id from login where email =
'$correo' and password = '$passw'", null)
```

Para **recorrer los registros** de la consulta realizada debemos utilizar los métodos de movimiento de la clase **Cursor**. Dado que el cursor comienza en la posición -1, la llamada a **moveToNext()** coloca la "posición de lectura" en la primera entrada de los resultados y muestra si el cursor ya pasó la última entrada del conjunto de resultados.

Para cada fila, se puede leer el valor de una columna llamando a uno de los **métodos GET de Cursor**: **getString()**, **getLong()**, etc. Para cada uno de esos métodos hay que indicarle la posición del índice de la columna, y se puede obtener llamando a los métodos **getColumnIndex()** o **getColumnIndexOrThrow()**. Al terminar la iteración de los resultados hay que llamar al método **close()** del cursor para liberar sus recursos.

Ejemplo:

```
while (filas.moveToNext()){
     val itemId = filas.getInt(filas.getColumnIndexOrThrow("id"))
      Log.println(Log.DEBUG, null, itemId.toString())
}
filas.close()
bd.close()
```

ELIMINAR INFORMACIÓN

Para borrar filas de una tabla hay que proporcionar los criterios de selección que identifiquen las filas a eliminar y llamar al método **delete()**.

El mecanismo funciona igual que los argumentos de selección del método **query()**: divide la especificación de selección en una cláusula de selección y argumentos de selección. La cláusula define las columnas que se comprobarán. Los argumentos son valores para probar que están vinculadas a la cláusula.

Ejemplo:

```
val conexion = BBDD_Helper(this, "usuario", null, 1)
val bd = conexion.writableDatabase
// Defina la parte WHERE de la query
val seleccion = "email like ?"
// Especifica los argumentos en el orden adecuado
val seleccionArgs = arrayOf("patriciasfo@educastur.org")
val filasEliminadas = bd.delete("login", seleccion, seleccionArgs)
bd.close()
```

El valor que **devuelve** el método **delete()** indica el **número de filas que se borraron** de la base de datos.

ACTUALIZAR INFORMACIÓN

Para actualizar un subconjunto de valores de la base de datos se utiliza el método **update()**.

La actualización de la tabla combina la sintaxis del **ContentValues** de la inserción de datos con la sintaxis **WHERE** de la eliminación.

Ejemplo:

```
val conexion = BBDD_Helper(this, "usuario", null, 1)
val bd = conexion.writableDatabase
// Nuevo valor para una columna
val passw = "presencial"
val valores = ContentValues().apply{
      put("password", passw)
}
// Especifica la fila a actualizar
val seleccion = "password like ?"
val seleccionArgs = arrayOf("prueba")
val filasAfectadas = bd.update("login", valores, seleccion, seleccionArgs)
bd.close()
```

El valor que **devuelve** el método **update()** indica el **número de filas afectadas** por la actualización en la base de datos.

4. Conexión persistente a la base de datos

Dado que llamar a **writableDatabase** y **readableDatabase** es **costoso** cuando la base de datos está cerrada, se debe dejar abierta la conexión con la base de datos durante el tiempo que necesites acceder a ella. Por lo general, lo óptimo es **cerrar la base de datos** en el método **onDestroy()** de la actividad de llamada.

Ejemplo:

```
override fun onDestroy(){
      conexion.close()
      super.onDestroy()
}
```