

**MP_0489. Programación
multimedia y dispositivos móviles**

**UF2. Programación de
aplicaciones para dispositivos móviles**

**2.7. Almacenamiento y
bases de datos**

Índice

☰	Objetivos	3
☰	Qué es el almacenamiento de datos	4
☰	Almacenamiento interno y externo	7
☰	SQLite	13
☰	SQLite Java Classes	20
☰	Ejemplo de bases de datos SQLite	24
☰	Resumen	42

Objetivos

Con esta unidad perseguimos los siguientes objetivos:

- 1 Aprender que Android ofrece varias opciones para guardar datos.
- 2 Descubrir el sistema de archivos SQLite.
- 3 Conocer cómo funciona SQLite.
- 4 Crear una aplicación que funcione con una base de datos.

¡Ánimo y adelante!

Qué es el almacenamiento de datos

Android ofrece varias opciones para que guardemos datos de aplicaciones persistentes.

La solución que elijamos dependerá de nuestras necesidades específicas.

Opciones de almacenamiento

Las opciones de almacenamiento de datos son las siguientes:

- 1 **Preferencias compartidas:** almacena datos primitivos privados en pares clave-valor.
- 2 **Almacenamiento interno:** almacena datos privados en la memoria del dispositivo.
- 3 **Almacenamiento externo:** almacena datos públicos en el almacenamiento externo compartido.
- 4 **Bases de datos SQLite:** almacenan datos estructurados en una base de datos privada.

5

Conexión de red: almacena datos en la web con su propio servidor de red.

6

Copia de seguridad en la nube: copia de seguridad de la aplicación y los datos del usuario en la nube.

7

Proveedores de contenido: almacenan los datos de forma privada y hacen que estén disponibles públicamente.

8

Base de datos en tiempo real de Firebase: almacena y sincroniza datos con una base de datos en la nube NoSQL. Los datos se sincronizan en todos los clientes en tiempo real y permanecen disponibles cuando su aplicación se desconecta.

Diferencia entre almacenamiento interno y externo

Android utiliza un sistema de archivos que es similar a los sistemas de archivos basados en disco en otras plataformas, como Linux. Las operaciones basadas en archivos son familiares para cualquiera que haya usado el uso de la E/S de archivos de Linux o el paquete java.io.

Todos los dispositivos Android tienen dos áreas de almacenamiento de archivos: almacenamiento "interno" y "externo".



Estos nombres provienen de los primeros días de Android, cuando **la mayoría de los dispositivos ofrecían memoria no volátil incorporada (almacenamiento interno)**, más **un medio de almacenamiento extraíble**, como una tarjeta micro SD (almacenamiento externo).

Hoy en día, algunos dispositivos dividen el espacio de almacenamiento permanente en particiones "internas" y "externas" por lo que, **incluso sin un medio de almacenamiento extraíble, siempre hay dos espacios de almacenamiento y el comportamiento de la API es el mismo**, sea el almacenamiento externo removible o no.

Resumen de las características de cada espacio de almacenamiento:

Almacenamiento interno

- **Siempre está disponible.**
- **Solo la aplicación puede acceder a los archivos.** Específicamente, el directorio de almacenamiento interno de la aplicación se especifica por el nombre de su paquete en una ubicación especial del sistema de archivos de Android. Otras aplicaciones no pueden navegar por los directorios internos, y no tienen acceso de lectura o escritura, a menos que se establezca explícitamente que los archivos sean legibles o grabables.
- Cuando el usuario desinstala la aplicación, el sistema elimina todos los archivos de la aplicación del almacenamiento interno.
- El almacenamiento interno **es mejor cuando se quiere estar seguro de que ni el usuario ni otras aplicaciones pueden acceder a los archivos.**

Almacenamiento externo

- **No siempre está disponible**, porque el usuario puede montar el almacenamiento externo como almacenamiento USB y, en algunos casos, eliminarlo del dispositivo.
- **Puede leer cualquier aplicación.**
- Cuando el usuario desinstala la aplicación, el sistema elimina los archivos de la aplicación desde aquí **solo si se guardan en el directorio desde `getExternalFilesDir()`**.
- El almacenamiento externo es el mejor lugar para los archivos que no requieren restricciones de acceso, que se desea compartir con otras aplicaciones o permitir que el usuario acceda con un ordenador.

Almacenamiento interno y externo

Vamos a analizar cómo funciona y cuáles son las funcionalidades de cada tipo de almacenamiento.

Comenzaremos por el **almacenamiento interno**.

Almacenamiento interno

No necesitamos ningún permiso para guardar archivos en el almacenamiento interno. La aplicación siempre tiene permiso para leer y escribir archivos en este directorio.

Podemos crear archivos en dos directorios diferentes:

1

Almacenamiento permanente: *getFilesDir()*.

2

Almacenamiento temporal: *getCacheDir()*. Recomendado para archivos pequeños, temporales, que suman menos de 1 MB.



Recordemos que el sistema puede eliminar archivos temporales si se queda sin memoria.

Para crear un nuevo archivo en uno de estos directorios podemos usar el constructor *File()*, pasando el archivo proporcionado por uno de los métodos anteriores, que especifica el directorio de almacenamiento interno. Por ejemplo:

```
File file = new File(context.getFilesDir(), filename);
```

Alternativamente, podemos llamar a *openFileOutput()* para obtener un *FileOutputStream*, que escribe en un archivo del directorio interno.

Por ejemplo, veamos cómo escribir texto en un archivo:

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
    outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
    outputStream.write(string.getBytes());
    outputStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Si necesitamos guardar en caché algunos archivos, usamos *createTempFile()*.

Veamos ahora cómo extraer el nombre de archivo de una URL y crear un archivo con ese nombre en el directorio de caché interno de la aplicación:

```
public File getTempFile(Context context, String url) {  
    File file;  
    try {  
        String fileName = Uri.parse(url).getLastPathSegment();  
        file = File.createTempFile(fileName, null, context.getCacheDir());  
    } catch (IOException e) {  
        // Error while creating file  
    }  
    return file;  
}
```

Pasemos ahora a analizar el **almacenamiento externo**.

Almacenamiento externo

Usamos el almacenamiento externo para los **archivos**, como imágenes, dibujos o documentos creados por su aplicación, que deben almacenarse permanentemente, si queremos que estén disponibles para otros usuarios y aplicaciones.

Algunos archivos privados que no tienen valor para otras aplicaciones también pueden almacenarse en un almacenamiento externo. Dichos archivos pueden ser recursos adicionales de aplicaciones descargadas o archivos de medios temporales.

1

Para escribir en el almacenamiento externo debemos solicitar el permiso **WRITE_EXTERNAL_STORAGE** en su archivo de manifiesto. Esto, implícitamente, incluye permiso para leer.

```
<manifest ...>  
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />  
    ...  
</manifest>
```

2

Si la aplicación necesita leer el almacenamiento externo, pero no escribir en él, debemos declarar el permiso **READ_EXTERNAL_STORAGE**.

```
<manifest ...>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    ...
</manifest>
```

Comprobar si el almacenamiento externo está montado

El almacenamiento externo puede no estar disponible, como, por ejemplo, cuando el usuario ha montado el almacenamiento en un PC o ha retirado la tarjeta SD con el almacenamiento externo. Por eso, siempre debemos verificar que el volumen esté disponible antes de acceder a él.

Podemos consultar el estado de almacenamiento externo llamando a *getExternalStorageState()*. Si el estado devuelto es igual a **MEDIA_MOUNTED**, podemos leer y escribir los archivos.

Por ejemplo, los siguientes métodos son útiles para determinar la disponibilidad de almacenamiento:

```
/* Comprobar si podemos leer y escribir */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Comprobar si podemos leer */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

Diferencia entre almacenamiento externo público y privado

El almacenamiento externo está muy estructurado por Android. Existen directorios públicos y directorios privados específicos para la aplicación. Cada uno de estos árboles de archivos tiene directorios identificados por constantes del sistema.

- Cualquier archivo que almacenemos en el directorio público de tonos de llamada DIRECTORY_RINGTONES está disponible para todas las demás aplicaciones de tonos de llamada.
- Cualquier archivo que almacenemos en un directorio privado de tonos de llamada DIRECTORY_RINGTONES puede, de forma predeterminada, ser visto solo por esa aplicación y eliminarse junto con esa aplicación.



[Aquí](#) puedes consultar el **listado de directorios**.

Veamos algunas acciones con almacenamiento externo público y privado.

- Obtener descriptores de archivo.

Para acceder a un directorio de almacenamiento externo público, obtenemos una ruta y creamos un archivo que llame a *getExternalStoragePublicDirectory()*.

```
File path = Environment.getExternalStoragePublicDirectory(  
    Environment.DIRECTORY_PICTURES);  
File file = new File(path, "DemoPicture.jpg");
```

Para acceder a un directorio de almacenamiento externo privado, obtenemos una ruta y creamos un archivo que llame a `getExternalFilesDir()`.

```
File file = new File(getExternalFilesDir(null), "DemoFile.jpg");
```



Consultar espacio de almacenamiento.

Si sabemos de antemano cuánto ocupan los datos que estamos guardando, podemos averiguar si hay suficiente espacio disponible sin causar una excepción `IOException`, llamando a `getFreeSpace()` o `getTotalSpace()`.

Estos métodos proporcionan el espacio disponible actual y el espacio total en el volumen de almacenamiento, respectivamente.



Borrar archivos.

Siempre debemos eliminar los archivos que ya no necesitamos. La forma más sencilla de eliminar un archivo es tener la referencia de archivo abierta llamando a `delete()` en la misma referencia.

```
myFile.delete();
```

Si el archivo se guarda en el almacenamiento interno, también podemos pedirle al `Context` que localice y elimine un archivo llamando a `deleteFile()`:

```
myContext.deleteFile(fileName);
```

SQLite

SQLite es un sistema de gestión de base de datos relacional (RDBMS) integrado, es decir, que se proporciona en forma de biblioteca vinculada a las aplicaciones.

Como tal, **no hay un servidor de base de datos autónomo ejecutándose en segundo plano**. Todas las operaciones de la base de datos se manejan dentro de la aplicación, a través de llamadas a funciones contenidas en la biblioteca SQLite.

Los desarrolladores de SQLite han liberado la tecnología al dominio público con el resultado de que ahora **es una solución de base de datos ampliamente implementada**.

SQLite está escrito en el lenguaje de programación C y, como tal, el SDK de Android proporciona un "contenedor" basado en Java alrededor de la interfaz de base de datos subyacente. Básicamente, esto consiste en un conjunto de clases que pueden utilizarse dentro del código Java de una aplicación para crear y administrar bases de datos basadas en SQLite.

Se accede a los datos en bases de datos SQLite utilizando un lenguaje de alto nivel, conocido como **lenguaje de consulta estructurado**, que generalmente se abrevia a SQL.

Es un lenguaje estándar utilizado por la mayoría de los sistemas de administración de bases de datos relacionales. Se ajusta principalmente al estándar SQL-92.



SQL es un lenguaje muy simple y fácil de usar, diseñado específicamente para permitir la lectura y escritura de datos de bases de datos.

Probar SQLite en un dispositivo virtual de Android (AVD)

Android se entrega con SQLite preinstalado, incluido un entorno interactivo para emitir comandos SQL desde una sesión de shell `adb` conectada a una instancia de un emulador AVD de Android en ejecución.

Esta es una forma útil de aprender sobre SQLite y SQL, y también una herramienta muy valiosa para identificar problemas con bases de datos creadas por aplicaciones que se ejecutan en un emulador.

1

Para iniciar una sesión interactiva de SQLite, **comenzamos ejecutando una sesión de AVD**. Esto se puede lograr desde Android Studio, iniciando el *Administrador de dispositivos virtuales de Android* (Tools -> Android -> AVD Admin). Ahí, seleccionamos un AVD previamente configurado y hacemos clic en *Iniciar*.

2

Una vez que el AVD esté en funcionamiento, **abrimos una ventana de Terminal o Símbolo del sistema y nos conectamos al emulador**, utilizando la herramienta de línea de comandos *adb* de la siguiente manera (ten en cuenta que el indicador *-e* dirige la herramienta para buscar un emulador con el que conectarse, en lugar de un dispositivo físico):

```
adb -e shell
```

3

Una vez conectado, el entorno de *shell* proporcionará un indicador de comando en el que se pueden ingresar los comandos:

```
root@android:/ #
```

Ruta del sistema de archivos de base de datos

Los datos almacenados en bases de datos SQLite se almacenan realmente en archivos de base de datos en el sistema de archivos del dispositivo Android en el que se ejecuta la aplicación. De forma predeterminada, la **ruta del sistema de archivos para estos archivos de base de datos** es la siguiente:

```
/data/data/<package name>/databases/<nombre archivo>.db
```

Por ejemplo, si una aplicación con el nombre del paquete *com.miejemplo.MiBDApp* crea una base de datos llamada **mibasedatos.db**, la ruta al archivo en el dispositivo se leerá de la siguiente manera:

```
/data/data/com.miejemplo.MiBDApp/databases/mibasedatos.db
```

Veamos un ejemplo:

1

Para este ejercicio, **cambiamos el directorio a /data/data dentro del shell adb y creamos una jerarquía de subdirectorios** adecuada para algunos experimentos de SQLite:

```
cd /data/data
mkdir com.miejemplo.ejemplobd
cd com.miejemplo.ejemplobd
mkdir databases
cd databases
```

2

Con una ubicación adecuada, creada para el archivo de base de datos, **iniciamos la herramienta interactiva SQLite** de la siguiente manera:

```
root@android:/data/data/databases # sqlite3 ./mibasedatos.db
sqlite3 ./mibasedatos.db
SQLite version 3.7.4
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

3

En el indicador `sqlite>` podemos ingresar comandos para realizar tareas, tales como crear tablas, e insertar y recuperar datos.

Por ejemplo, para crear una nueva tabla en nuestra base de datos con campos que contengan los campos de ID, nombre, dirección y número de teléfono, se requiere la siguiente declaración:

```
create table contactos (_id integer primary key autoincrement, nombre  
text, direccion text, telefono text);
```



Hay que tener en cuenta que **cada fila de una tabla debe tener una clave principal que sea única para esa fila.**

En el ejemplo anterior hemos designado el campo ID como clave principal, lo hemos declarado como de *tipo entero* y le pedimos a SQLite que incremente automáticamente el número cada vez que se agregue una fila. Esta es una forma común de asegurarse de que cada fila tenga una clave principal única.

En la mayoría de plataformas, la elección del nombre para la clave principal es arbitraria. Sin embargo, **en el caso de Android es esencial que la clave se llame `_id` para que la base de datos sea completamente accesible**, utilizando todas las clases relacionadas con la base de datos de Android. Los campos restantes se declaran como de tipo texto.

Acciones que podemos realizar sobre la base de datos

- Para **listar las tablas en la base de datos seleccionada** actualmente, usamos la declaración **.tables**:

```
sqlite> .tables
contactos
```

- Para **insertar registros en la tabla**:

```
sqlite> insert into contactos(nombre, direccion, telefono) values ("David P
sqlite> insert into contactos(nombre, direccion, telefono) values ("Pedro G
```

- Para **recuperar todas las filas de una tabla**:

```
sqlite> select * from contactos;
1|David Perez|C/Principal 4, Madrid|910000000
2|Pedro Gomez|Plaza Mayor 5, Albacete|967000000
```

- Para **extraer una fila que cumpla criterios específicos**:

```
sqlite> select * from contactos where nombre="Pedro Gomez";
2|Pedro Gomez|Plaza Mayor 5, Albacete|967000000
```

- Para salir del entorno interactivo sqlite3:

```
sqlite> .exit
```

Al ejecutar una aplicación de Android en el entorno del emulador, se crearán todos los archivos de la base de datos en el sistema de archivos del emulador, utilizando la convención de ruta anterior. Esto tiene la ventaja de que:

- Puede conectarse con *adb*.
- Puede navegar a la ubicación del archivo de base de datos.
- Se puede cargar en la herramienta interactiva *sqlite3*.
- Y puede realizar tareas en los datos para identificar posibles problemas que ocurren en el código de la aplicación.

También es importante tener en cuenta que, si bien es posible conectarse con un *shell adb* a un dispositivo físico con Android, al *shell* no se le otorgan privilegios suficientes de manera predeterminada para crear y administrar bases de datos SQLite. Por lo tanto, la depuración de problemas de la base de datos se realiza mejor usando una sesión AVD.

Continuaremos dentro de SQL, trabajando con **SQLite Java Classes**, en el siguiente apartado.

SQLite Java Classes

SQLite está escrito en el lenguaje de programación C, mientras que las aplicaciones de Android se desarrollan principalmente utilizando Java.

Para cerrar esta "brecha de idioma", el SDK de Android incluye un conjunto de **clases que proporciona una capa Java sobre el sistema de administración de bases de datos SQLite**.



Cursor.

Es una clase proporcionada específicamente para dar acceso a los resultados de una consulta de base de datos.

Por ejemplo, una operación SELECT de SQL realizada en una base de datos, posiblemente devolverá varias filas coincidentes de la base de datos. Se puede usar una instancia de Cursor para pasar por estos resultados, a los que luego se puede acceder desde el código de la aplicación, utilizando una variedad de métodos.

Algunos métodos clave de esta clase son los siguientes:

- `close()`: libera todos los recursos utilizados por el *cursor* y lo cierra.
- `getCount()`: devuelve el número de filas contenidas en el conjunto de resultados.
- `moveToFirst()`: se mueve a la primera fila dentro del conjunto de resultados.
- `moveToLast()`: se desplaza a la última fila del conjunto de resultados.
- `moveToNext()`: se mueve a la siguiente fila en el conjunto de resultados.
- `move()`: se mueve en un desplazamiento especificado desde la posición actual en el conjunto de resultados.
- `get<tipo>()`: devuelve el valor del <tipo> especificado en el índice de columna en la posición actual del cursor (las variaciones consisten en `getString()`, `getInt()`, `getShort()`, `getFloat()` y `getDouble()`).



SQLiteDatabase

Esta clase proporciona la interfaz principal entre el código de la aplicación y las bases de datos SQLite subyacentes, incluida la capacidad de crear, eliminar y realizar operaciones basadas en SQL en bases de datos.

Algunos métodos clave de esta clase son los siguientes:

- `insert()`: inserta una nueva fila en una tabla de base de datos.
- `delete()`: borra filas de una tabla de base de datos.
- `query()`: realiza una consulta de base de datos específica y devuelve resultados coincidentes mediante un objeto *Cursor*.
- `execSQL()`: ejecuta una sola instrucción SQL que no devuelve datos de resultados.
- `rawQuery()`: ejecuta una declaración de consulta SQL y devuelve resultados coincidentes en forma de un objeto *Cursor*.



SQLiteOpenHelper

Es una clase auxiliar diseñada para facilitar la creación y actualización de bases de datos. Esta clase debe ser subclaseada dentro del código de la aplicación que busca acceso a la base de datos, y los siguientes métodos de devolución de llamada implementados dentro de esa subclase:

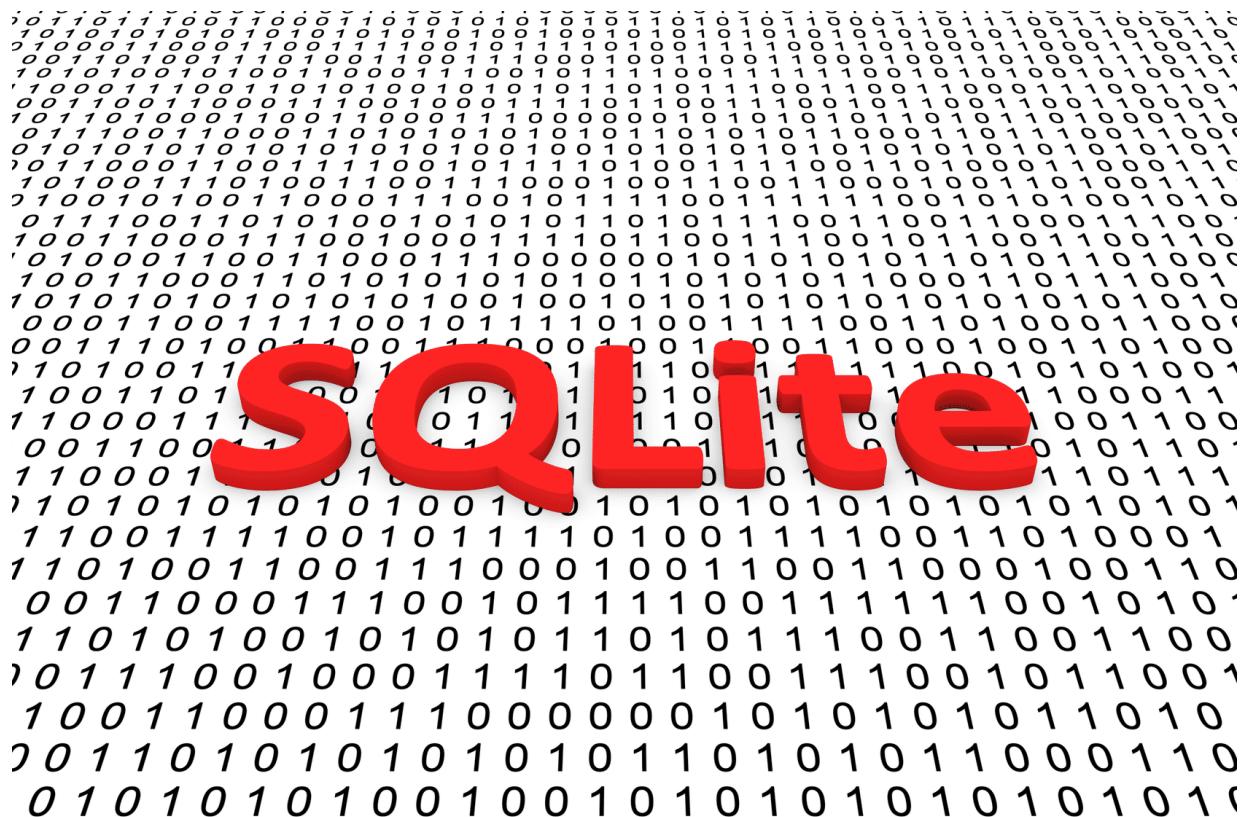
- ***onCreate()***: es llamado cuando se crea la base de datos por primera vez. Este método se pasa como argumento al objeto *SQLiteDatabase* para la base de datos recién creada. Esta es la ubicación ideal para inicializar la base de datos, en términos de crear una tabla e insertar cualquier fila de datos inicial.
- ***onUpgrade()***: es llamado en caso de que el código de la aplicación contenga una referencia de número de versión de la base de datos más reciente. Generalmente, se usa cuando una aplicación se actualiza en el dispositivo y requiere que el esquema de la base de datos también se actualice para manejar el almacenamiento de datos adicionales.

Además de estos métodos de devolución de llamada obligatorios, el método ***onOpen()***, llamado cuando se abre la base de datos, puede implementarse dentro de la subclase.

El constructor de la subclase también debe implementarse para llamar a la superclase, pasando por el contexto de la aplicación, el nombre de la base de datos y la versión de la base de datos.

Los métodos notables de la clase *SQLiteOpenHelper* incluyen:

- ***getWritableDatabase()***: abre o crea una base de datos para leer y escribir. Devuelve una referencia a la base de datos en forma de un objeto *SQLiteDatabase*.
- ***getReadableDatabase()***: crea o abre una base de datos solo para lectura. Devuelve una referencia a la base de datos en forma de un objeto *SQLiteDatabase*.
- ***SQLiteDatabase.close()***: cierra la base de datos.



ContentValues.

ContentValues es una clase de conveniencia que permite que se declaren pares clave–valor que consisten en identificadores de columna de tabla y los valores que se almacenan en cada columna.



La clase *ContentValues* es de uso particular cuando se insertan o actualizan entradas en una tabla de base de datos.

Ejemplo de bases de datos SQLite

Vamos a crear un sencillo ejemplo que nos permita agregar, consultar y eliminar entradas de una base de datos.

Nuestro proyecto es una **aplicación de ingreso y recuperación de datos** diseñada para permitir al usuario agregar, consultar y eliminar entradas de bases de datos. La idea es permitir el seguimiento del inventario de productos.

Con este código tendremos nuestra **interfaz de usuario**:

```
<LinearLayout
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <TableLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal">

        <TableRow
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:layout_margin="10dp">

            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textAppearance="?android:attr/textAppearanceLarge"
```

```
        android:text="@string/id_string"
        android:id="@+id/textView" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="Not assigned"
        android:id="@+id/productID" />
</TableRow>

<TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_margin="10dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="@string/product_string"
        android:id="@+id/textView3" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/productName" />
</TableRow>

<TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_margin="10dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="@string/quantity_string"
        android:id="@+id/textView2" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:inputType="numberDecimal"
        android:ems="10"
        android:id="@+id/productQuantity" />
</TableRow>
</TableLayout>

<LinearLayout
    android:orientation="horizontal"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:layout_gravity="center_horizontal"
    android:layout_margin="10dp">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/add_string"
        android:id="@+id/button" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/find_string"
        android:id="@+id/button2" />

```

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/delete_string"  
    android:id="@+id/button3" />  
</LinearLayout>  
</LinearLayout>
```

El nombre de la base de datos será **productID.db** que, a su vez, contendrá una sola tabla, llamada **products**.

Cada registro en la tabla de la base de datos contendrá un ID de producto único, una descripción del producto y la cantidad de ese producto actualmente en stock, correspondiente a los nombres de columna de "productid", "productname" y "productquantity", respectivamente. La columna **productid** actuará como clave principal y será automáticamente asignada e incrementada por el sistema de administración de la base de datos.

Crear el modelo de datos

Una vez completada, la aplicación constará de una actividad y una clase de **handler** de base de datos. El controlador de la base de datos será una subclase de **SQLiteOpenHelper** y proporcionará una capa abstracta entre la base de datos SQLite subyacente y la clase de actividad, y la actividad llamará al controlador de la base de datos a interactuar con la base de datos (agregar, eliminar y consultar las entradas).

Para implementar esta interacción de manera estructurada, será necesario implementar una tercera clase para mantener los datos de entrada de la base de datos a medida que se pasan entre la actividad y el **handler**. En realidad, esta es una clase muy simple, capaz de mantener los valores de ID de producto, nombre de producto y cantidad de producto, junto con los métodos de obtención y configuración para acceder a estos valores.

Las instancias de esta clase se pueden crear dentro de la actividad y el controlador de la base de datos, y se pueden pasar de un lado a otro, según sea necesario. Esencialmente, se puede considerar que **esta clase representa el modelo de base de datos**.

1

Creamos una nueva clase, tal como vemos en la imagen.



2

En el cuadro de diálogo *Crear nueva clase*, nombramos la clase **Product** antes de hacer clic en el botón *Aceptar*.

3

Una vez creado, el archivo fuente **Product.java** se cargará automáticamente en el editor de Android Studio. Cuando se cargue, **modificamos el código para agregar el siguiente:**

```
package com.miejemplo.basedatos;

public class Product {

    private int _id;
    private String _productname;
    private int _quantity;

    public Product() {
    }

    public Product(int id, String productname, int quantity) {
        this._id = id;
        this._productname = productname;
```

```
    this._quantity = quantity;
}

public Product(String productname, int quantity) {
    this._productname = productname;
    this._quantity = quantity;
}

public void setID(int id) {
    this._id = id;
}

public int getID() {
    return this._id;
}

public void setProductName(String productname) {
    this._productname = productname;
}

public String getProductName() {
    return this._productname;
}

public void setQuantity(int quantity) {
    this._quantity = quantity;
}

public int getQuantity() {
    return this._quantity;
}
```

La clase completada contiene variables de datos privados para el almacenamiento interno de las columnas de datos, de las entradas de la base de datos, y un conjunto de métodos para obtener y establecer esos valores.

Implementar el *Handler* de datos

El *Handler* de datos se implementará como subclase de la clase *SQLiteOpenHelper* de Android agregando el constructor, los métodos *onCreate()* y *onUpgrade()*. Dado que se requerirá que el controlador agregue, consulte y elimine datos en nombre del componente de la actividad, los métodos correspondientes también deberán agregarse a la clase.

1

Comenzaremos agregando una segunda nueva clase al proyecto, para que actúe como controlador, tal como hicimos antes (esta vez, lo denominaremos *MyDBHandler*).

2

Modificamos el código, añadiendo los métodos *template onCreate()* y *onUpgrade()* y las constantes para el nombre de la base de datos, el nombre de la tabla, las columnas de la tabla y la versión de la base de datos. También para agregar el método del constructor, de la siguiente manera:

```
package com.miejemplo.basedatos;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;

public class MyDBHandler extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "productDB.db";
    private static final String TABLE_PRODUCTS = "products";

    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_PRODUCTNAME = "productname";
    public static final String COLUMN_QUANTITY = "quantity";

    public MyDBHandler(Context context, String name,
                       SQLiteDatabase.CursorFactory factory, int version) {
        super(context, DATABASE_NAME, factory, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {

    }

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

    }
}
```

3

A continuación, debe implementarse el método `onCreate()`, de modo que la tabla de productos se cree cuando la base de datos se inicie por primera vez.

Esto implica construir una instrucción CREATE de SQL que contenga instrucciones para crear una nueva tabla con las columnas apropiadas, y luego pasarlal al método `execSQL()` del objeto `SQLiteDatabase`, pasando como argumento a `onCreate()`:

```
package com.miejemplo.basedatos;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;

public class MyDBHandler extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "productDB.db";
    private static final String TABLE_PRODUCTS = "products";

    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_PRODUCTNAME = "productname";
    public static final String COLUMN_QUANTITY = "quantity";

    public MyDBHandler(Context context, String name,
                       SQLiteDatabase.CursorFactory factory, int version) {
        super(context, DATABASE_NAME, factory, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String CREATE_PRODUCTS_TABLE = "CREATE TABLE " +
            TABLE_PRODUCTS + "("
            + COLUMN_ID + " INTEGER PRIMARY KEY," + COLUMN_PRODUCTNAME
            + " TEXT," + COLUMN_QUANTITY + " INTEGER" + ")";
        db.execSQL(CREATE_PRODUCTS_TABLE);
    }

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

    }
}
```

4

El método `onUpgrade()` se invoca cuando invitamos el controlador con un número de versión de base de datos mayor que el utilizado anteriormente. Los pasos exactos que se realizarán en esta instancia serán específicos de la aplicación, por lo que, a efectos de este ejemplo, simplemente **eliminaremos la base de datos anterior y crearemos una nueva**:

```
package com.miejemplo.basedatos;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;

public class MyDBHandler extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "productDB.db";
    private static final String TABLE_PRODUCTS = "products";

    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_PRODUCTNAME = "productname";
    public static final String COLUMN_QUANTITY = "quantity";

    public MyDBHandler(Context context, String name,
                       SQLiteDatabase.CursorFactory factory, int version) {
        super(context, DATABASE_NAME, factory, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String CREATE_PRODUCTS_TABLE = "CREATE TABLE " +
            TABLE_PRODUCTS + "("
            + COLUMN_ID + " INTEGER PRIMARY KEY," + COLUMN_PRODUCTNAME
            + " TEXT," + COLUMN_QUANTITY + " INTEGER" + ")";
        db.execSQL(CREATE_PRODUCTS_TABLE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_PRODUCTS);
        onCreate(db);
    }
}
```

5

Todo lo que ahora queda por implementar en la clase de controlador `MyDBHandler.java` son los **métodos para agregar, consultar y eliminar entradas de la tabla de base de datos**.

Añadir

El método para insertar registros en la base de datos se llamará `addProduct()` y tomará como argumento una instancia de nuestra clase de modelo de datos de producto.

1

Se creará un objeto `ContentValues` en el cuerpo del método y se cebará con pares clave–valor para las columnas de datos extraídas del objeto `Product`.

2

A continuación, se obtendrá una referencia a la base de datos mediante una llamada a `getWritableDatabase()`, seguida de una llamada al método `insert()` del objeto de base de datos devuelto.

3

Finalmente, una vez que se ha realizado la inserción, la base de datos debe cerrarse:

```
package com.miejemplo.basedatos;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;
import com.miejemplo.basedatos.Product;

public class MyDBHandler extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "productDB.db";
    private static final String TABLE_PRODUCTS = "products";

    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_PRODUCTNAME = "productname";
    public static final String COLUMN_QUANTITY = "quantity";

    public MyDBHandler(Context context, String name,
                       SQLiteDatabase.CursorFactory factory, int version) {
        super(context, DATABASE_NAME, factory, DATABASE_VERSION);
    }

    public void addProduct(Product product) {

        ContentValues values = new ContentValues();
        values.put(COLUMN_PRODUCTNAME, product.getProductName());
        values.put(COLUMN_QUANTITY, product.getQuantity());

        SQLiteDatabase db = this.getWritableDatabase();

        db.insert(TABLE_PRODUCTS, null, values);
        db.close();
    }

    @Override
```

```
public void onCreate(SQLiteDatabase db) {  
    String CREATE_PRODUCTS_TABLE = "CREATE TABLE " +  
        TABLE_PRODUCTS + "("  
        + COLUMN_ID + " INTEGER PRIMARY KEY," + COLUMN_PRODUCTNAME  
        + " TEXT," + COLUMN_QUANTITY + " INTEGER" + ")";  
    db.execSQL(CREATE_PRODUCTS_TABLE);  
}  
  
@Override  
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
    db.execSQL("DROP TABLE IF EXISTS " + TABLE_PRODUCTS);  
    onCreate(db);  
}  
  
}
```

Buscar

En este caso, el método para consultar la base de datos se llamará *findProduct()* y tomará como argumento un objeto *String*, que contiene el nombre del producto a ubicar. Usando esta cadena, se construirá una instrucción SQL SELECT para encontrar todos los registros coincidentes en la tabla.

Para este ejemplo, solo se devolverá la primera coincidencia, contenida en una nueva instancia de nuestra clase de modelo de datos de *Product*:

```
package com.miejemplo.basedatos;  
  
import android.database.sqlite.SQLiteDatabase;  
import android.database.sqlite.SQLiteOpenHelper;  
import android.content.Context;  
import android.content.ContentValues;  
import android.database.Cursor;  
import com.miejemplo.basedatos.Product;  
  
public class MyDBHandler extends SQLiteOpenHelper {  
  
    private static final int DATABASE_VERSION = 1;  
    private static final String DATABASE_NAME = "productDB.db";  
    private static final String TABLE_PRODUCTS = "products";  
  
    public static final String COLUMN_ID = "_id";  
    public static final String COLUMN_PRODUCTNAME = "productname";  
    public static final String COLUMN_QUANTITY = "quantity";
```

```
public MyDBHandler(Context context, String name,
    SQLiteDatabase.CursorFactory factory, int version) {
    super(context, DATABASE_NAME, factory, DATABASE_VERSION);

}

public void addProduct(Product product) {

    ContentValues values = new ContentValues();
    values.put(COLUMN_PRODUCTNAME, product.getProductName());
    values.put(COLUMN_QUANTITY, product.getQuantity());

    SQLiteDatabase db = this.getWritableDatabase();

    db.insert(TABLE_PRODUCTS, null, values);
    db.close();
}

public Product findProduct(String productname) {
    String query = "Select * FROM " + TABLE_PRODUCTS + " WHERE " + COLUMN_PRODUCTNAME + " = '" + productname + "'";

    SQLiteDatabase db = this.getWritableDatabase();

    Cursor cursor = db.rawQuery(query, null);

    Product product = new Product();

    if (cursor.moveToFirst()) {
        cursor.moveToFirst();
        product.setId(Integer.parseInt(cursor.getString(0)));
        product.setProductName(cursor.getString(1));
        product.setQuantity(Integer.parseInt(cursor.getString(2)));
        cursor.close();
    } else {
        product = null;
    }
    db.close();
    return product;
}

@Override
public void onCreate(SQLiteDatabase db) {
    String CREATE_PRODUCTS_TABLE = "CREATE TABLE " +
        TABLE_PRODUCTS + "("
        + COLUMN_ID + " INTEGER PRIMARY KEY," + COLUMN_PRODUCTNAME
        + " TEXT," + COLUMN_QUANTITY + " INTEGER" + ")";
    db.execSQL(CREATE_PRODUCTS_TABLE);
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS " + TABLE_PRODUCTS);
    onCreate(db);
}

}
```

Borrar

El método de eliminación se llamará `deleteProduct()` y aceptará como argumento la entrada que se eliminará en forma de un objeto `Product`.

El método utilizará una instrucción SELECT de SQL para buscar la entrada según el nombre del producto y, si lo encuentra, eliminarlo de la tabla. El éxito o no de la eliminación se reflejará en un valor de retorno booleano:

```
package com.miejemplo.basedatos;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.content.Context;
import android.content.ContentValues;
import android.database.Cursor;
import com.miejemplo.basedatos.Product;

public class MyDBHandler extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "productDB.db";
    private static final String TABLE_PRODUCTS = "products";

    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_PRODUCTNAME = "productname";
    public static final String COLUMN_QUANTITY = "quantity";

    public MyDBHandler(Context context, String name,
                       SQLiteDatabase.CursorFactory factory, int version) {
        super(context, DATABASE_NAME, factory, DATABASE_VERSION);
    }

    public void addProduct(Product product) {

        ContentValues values = new ContentValues();
        values.put(COLUMN_PRODUCTNAME, product.getProductName());
        values.put(COLUMN_QUANTITY, product.getQuantity());

        SQLiteDatabase db = this.getWritableDatabase();

        db.insert(TABLE_PRODUCTS, null, values);
        db.close();
    }

    public Product findProduct(String productname) {
        String query = "Select * FROM " + TABLE_PRODUCTS + " WHERE " + COLUMN_PRODUCTNAME + " = '" + productname + "'";

        SQLiteDatabase db = this.getWritableDatabase();

        Cursor cursor = db.rawQuery(query, null);
    }
}
```

```
Product product = new Product();

if (cursor.moveToFirst()) {
    cursor.moveToFirst();
    product.setID(Integer.parseInt(cursor.getString(0)));
    product.setProductName(cursor.getString(1));
    product.setQuantity(Integer.parseInt(cursor.getString(2)));
    cursor.close();
} else {
    product = null;
}
db.close();
return product;
}

public boolean deleteProduct(String productname) {

    boolean result = false;

    String query = "Select * FROM " + TABLE_PRODUCTS + " WHERE " + COLUMN_PRODUCTNAME + " = \'" + productname + "\'";

    SQLiteDatabase db = this.getWritableDatabase();

    Cursor cursor = db.rawQuery(query, null);

    Product product = new Product();

    if (cursor.moveToFirst()) {
        product.setID(Integer.parseInt(cursor.getString(0)));
        db.delete(TABLE_PRODUCTS, COLUMN_ID + " = ?",
            new String[] { String.valueOf(product.getID()) });
        cursor.close();
        result = true;
    }
    db.close();
    return result;
}

@Override
public void onCreate(SQLiteDatabase db) {
    String CREATE_PRODUCTS_TABLE = "CREATE TABLE " +
        TABLE_PRODUCTS + "("
        + COLUMN_ID + " INTEGER PRIMARY KEY," + COLUMN_PRODUCTNAME
        + " TEXT," + COLUMN_QUANTITY + " INTEGER" + ")";
    db.execSQL(CREATE_PRODUCTS_TABLE);
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS " + TABLE_PRODUCTS);
    onCreate(db);
}

}
```

Implementar los eventos

Por último, debemos conectar los controladores de eventos `onClick` en los tres botones de la interfaz de usuario, e implementar los métodos correspondientes para esos eventos.

1

En el archivo `activity_main.xml` modificamos los tres elementos `button` para agregar las propiedades de `onClick`:

```
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Añadir"  
    android:id="@+id/button"  
    android:onClick="newProduct" />  
  
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Buscar"  
    android:id="@+id/button2"  
    android:onClick="lookupProduct" />  
  
<Button  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Borrar"  
    android:id="@+id/button3"  
    android:onClick="removeProduct" />
```

2

En el archivo `MainActivity.java` implementamos el código para identificar las `views` en la interfaz de usuario y para implementar los tres métodos de destino "onClick":

```
package com.miejemplo.basedatos;  
  
import android.support.v7.app.AppCompatActivity;  
import android.os.Bundle;  
import android.view.View;  
import android.widget.EditText;  
import android.widget.TextView;
```

```
public class MainActivity extends AppCompatActivity {  
  
    TextView idView;  
    EditText productBox;  
    EditText quantityBox;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        idView = (TextView) findViewById(R.id.productID);  
        productBox = (EditText) findViewById(R.id.productName);  
        quantityBox =  
            (EditText) findViewById(R.id.productQuantity);  
    }  
  
    public void newProduct (View view) {  
        MyDBHandler dbHandler = new MyDBHandler(this, null, null, 1);  
  
        int quantity =  
            Integer.parseInt(quantityBox.getText().toString());  
  
        Product product =  
            new Product(productBox.getText().toString(), quantity);  
  
        dbHandler.addProduct(product);  
        productBox.setText("");  
        quantityBox.setText("");  
    }  
  
    public void lookupProduct (View view) {  
        MyDBHandler dbHandler = new MyDBHandler(this, null, null, 1);  
  
        Product product =  
            dbHandler.findProduct(productBox.getText().toString());  
  
        if (product != null) {  
            idView.setText(String.valueOf(product.getID()));  
  
            quantityBox.setText(String.valueOf(product.getQuantity()));  
        } else {  
            idView.setText("No se encontraron resultados");  
        }  
    }  
  
    public void removeProduct (View view) {  
        MyDBHandler dbHandler = new MyDBHandler(this, null,  
            null, 1);  
  
        boolean result = dbHandler.deleteProduct(  
            productBox.getText().toString());  
  
        if (result)  
        {  
            idView.setText("Producto eliminado");  
            productBox.setText("");  
            quantityBox.setText("");  
        } else  
            idView.setText("No se encontraron resultados");  
    }  
}
```

De esta manera, **habremos completado nuestra aplicación, que tendrá este aspecto en funcionamiento:**



Añadimos un producto *Ejemplo 1*.



Añadimos un segundo producto *Ejemplo 2*.



Buscamos *Ejemplo 2* y nos muestra este resultado.



Eliminamos *Ejemplo 2*.



Si ahora buscamos *Ejemplo 2*, no muestra resultados.

Resumen

Hemos terminado la lección, repasemos los puntos más importantes que hemos tratado.

- A lo largo de esta unidad hemos conocido algunos **métodos de almacenamiento en Android** y su funcionamiento.
- Después hemos aprendido en qué consisten el **almacenamiento interno y el almacenamiento externo**, así como sus **utilidades y diferencias**. Además, hemos visto cómo acceder a cada uno de ellos, sus permisos y los modos de utilizarlos.
- Hemos descubierto **SQLite**, una forma de acceder a las bases de datos desde la consola para poder trabajar con ellas.
- Y, para finalizar, hemos descubierto las **clases que tiene Android para gestionar SQLite**, desarrollando, paso a paso, un ejemplo básico con el que podemos añadir, buscar y borrar entradas en la base de datos.



PROEDUCA