Control de cambios:		
Fecha:	Versión:	Descripción:
27/01/2025	R0.1	Versión inicial. Inicio demo SQLite

PARTE 1. PERSISTENCIA EN ANDROID

Los programas, ya sean para escritorio, web o móviles, necesitan guardar datos de una ejecución a otra para su correcto funcionamiento. Quizás necesiten almacenar opciones de usuario, quizá guarden logs de depuración con errores y alertas, quizá estemos hablando de un juego en el que debemos guardar los jugadores y los puntos que han conseguido en cada partida. Veamos los métodos más usuales para guardar información en el dispositivo móvil.

Opciones para la persistencia en Android:

A. Preferencias compartidas (SharedPreferences)
B. Ficheros:
    a. Sistema de almacenamiento interno.
    b. Sistema de almacenamiento externo (por ejemplo, tarjeta SD).
C. XML
D. Bases de datos: por ejemplo, SQLite, Room ("parecido" a Hibernate)
E. Internet: Almacenamiento en la nube (por ejemplo, Google Firebase), acceso a servicios web (API REST,...)

A. Preferencias compartidas (Clase SharedPreferences)

- Las Shared Preferences o preferencias compartidas son un mecanismo proporcionado por Android que permite almacenar y recuperar ***datos primitivos*** en forma de pares clave/valor.
- Este mecanismo se suele utilizar, por ejemplo, para almacenar los parámetros de configuración de una aplicación.
- Desaparecen (se eliminan) cuando desinstalamos una aplicación.

B. Ficheros

- Podemos utilizar 3 tipos de ficheros en Android:
    - En la memoria interna del dispositivo.
    - En la memoria externa (tarjeta SD)
    - En los recursos de la aplicación (solo lectura, por lo que no son útiles para almacenar información desde la aplicación). Utilizan las carpetas `res` o `assets`.
- Android hereda el sistema de ficheros de Linux:
    - Cuando se instala una aplicación se crea un nuevo usuario.
    - Por defecto, los ficheros solo son accesibles por la aplicación que los crea.
    - También podemos dar acceso de lectura o escritura al resto de aplicaciones.

C. XML

XML es uno de los estándares más utilizados en la actualidad para codificar información. Es ampliamente utilizado en Internet; además, como has comprobado en este módulo, tiene muchos usos en el SDK de Android. Entre otras cosas, es utilizado para definir layouts, animaciones, el ficheros AndroidManifest.xml, etc.

Android tiene varias librerías para trabajar con XML en Kotlin. Algunas de las más populares incluyen:

- ***Android KTX***: Es una colección de extensiones de Kotlin que se han diseñado específicamente para hacer que el desarrollo de Android sea más fácil y eficiente. Con Android KTX, puedes trabajar con XML de una manera más simple y concisa.
- ***XmlPullParser***: Es una clase que proporciona una manera fácil de analizar documentos XML en Android. XmlPullParser es una librería liviana y rápida que puedes utilizar para analizar y extraer información de los documentos XML.
- ***Jsoup***: Es una biblioteca Java que te permite analizar y extraer información de documentos HTML y XML. Jsoup es una de las librerías más populares y potentes para trabajar con documentos XML en Android.
- ***Moshi:*** Es una biblioteca de análisis de JSON para Android que también admite la lectura y escritura de XML.

D. Bases de datos

Como seguro sabes gracias al módulo de Acceso a Datos, SQLite es un ligero motor de bases de datos de código abierto, que se caracteriza por mantener el almacenamiento de información persistente de forma sencilla.

A diferencia de otros Sistemas gestores de bases de datos como MySQL, SQL Server y Oracle DB, SQLite:

- **No requiere el soporte de un servidor**: SQLite no ejecuta un proceso para administrar la información, si no que implementa un conjunto de librerías encargadas de la gestión.
- **No necesita configuración**: Libera al programador de todo tipo de configuraciones de puertos, tamaños, ubicaciones, etc.
- **Usa un archivo para el esquema**: Crea un archivo para el esquema completo de una base de datos, lo que permite ahorrarse preocupaciones de seguridad, ya que los datos de las aplicaciones Android no pueden ser accedidos por contextos externos.
- **Es de Código Abierto**: Está disponible al dominio público de los desarrolladores al igual que sus archivos de compilación e instrucciones de escalabilidad.

Por todo esto SQLite es una tecnología cómoda para los dispositivos móviles. Por defecto, la base de datos SQLite está integrada en Android, por lo que no es necesario realizar ninguna tarea de configuración o administración de la base de datos.

E. Almacenamiento en la nube

En ocasiones, una base de datos local puede no ser suficiente para nuestros propósitos. Imaginemos una aplicación de chat que guarde las conversaciones y los contactos. Si almacenásemos estos datos en una BBDD local, un mismo usuario no podría acceder a ellos desde dos terminales distintos. Unas conversaciones estarían en el móvil y las otras en la tablet, y sería un verdadero engorro. Para que el usuario se conecte desde cualquier dispositivo y acceda a todos sus datos indistintamente, tendríamos que realizar un complejo sistema de sincronización entre las bases de datos locales de todos los dispositivos, lo que implicaría un servidor en internet con la base de datos central y un software de control muy complejo.

Hoy en día, hay sistemas como los descritos bien diseñados, eficientes y disponibles de forma gratuita y/o de pago. Sus API de desarrollo son bastante sencillas para los programadores, y el usuario solo necesitará una

conexión a internet para acceder a sus datos desde cualquier terminal. En ocasiones, estos sistemas permiten el cacheo y el almacenamiento local cuando el dispositivo ha perdido acceso a la red, para luego sincronizarse cuando obtenga conexión de nuevo. Hablamos por ejemplo de servicios en la nube como ***Firebase*** (https://firebase.google.com/?hl=es) o ***Backendless*** (https://backendless.com/). *Firebase* es una potente herramienta diseñada por Google, por lo que parece el candidato ideal para el uso en Android. Las bases de datos *Firebase* tienen opciones gratuitas y también de pago, como muchos otros servicios en la nube, dependiendo del volumen de tráfico y otros servicios añadidos. Tanto *Firebase* como *Backendless* y otros permiten un modo básico gratuito que nos permitirá hacer pruebas o apps sencillas.

Las bases de datos de *Firebase* son no relacionales (o **NoSQL, Not only SQL**), lo que significa que no tienen una estructura de tablas y filas ni utilizan SQL como lenguaje de definición o acceso a datos. Sus tablas son más bien colecciones de objetos con campos anidados de forma jerárquica en estructura de árbol de directorios que tendremos la posibilidad definir como creamos oportuno. Para acceder al servicio de base de datos de Firebase, necesitamos crear una cuenta en la consola de desarrollo disponible en https://console.firebase.google.com . Desde la consola podremos gestionar usuarios, diferentes modos de autenticación de usuario, reglas de seguridad, bases de datos, almacenamiento de archivos, hosting de webs estáticas, funciones, etcétera.

DEMO 5.1. SharedPreferences

Introducción

Las *SharedPreferences* son un mecanismo que ofrece Android para almacenar datos simples en formato **clave-valor** en un archivo privado de nuestra aplicación.

Algunas de las cosas que se suelen almacenar en las *SharedPreferences* son:

- Configuraciones de la aplicación, como el idioma, la fuente, el tema, el sonido y la vibración.
- Preferencias del usuario, como las notificaciones, el modo nocturno, la orientación de la pantalla, el tamaño de la fuente,...
- Información de sesión, como el nombre de usuario, la fecha de inicio de sesión y el estado de autenticación.
- Datos de estado de la aplicación, como la última pantalla visitada, los elementos seleccionados y las acciones realizadas.
- Otros datos pequeños y simples que se necesiten en la aplicación, como una lista de elementos favoritos, la frecuencia de actualización de los datos o el tiempo de espera para una tarea.

En general, las *SharedPreferences* son útiles para almacenar datos que se necesiten de forma rápida y fácil en diferentes partes de la aplicación, pero que no son críticos para el funcionamiento de la misma. Es importante tener en cuenta que no se deben guardar en las *SharedPreferences* datos sensibles o críticos de seguridad, como contraseñas o información financiera.

Ejemplos:

1. Guardar una ***cadena de texto*** en las SharedPreferences:

```
val prefs = getSharedPreferences("MisPrefs", Context.MODE_PRIVATE)
val editor = prefs.edit()
editor.putString("clave", "valor")
editor.commit()
```

   Leer una ***cadena de texto*** de las SharedPreferences:

```
 val prefs = getSharedPreferences("MisPrefs", Context.MODE_PRIVATE)
 val valor = prefs.getString("clave", "valor predeterminado")
// La segunda cadena de getString sería el valor a utilizar si no se encuentra
// esa clave.
```

2. Guardar un ***valor booleano*** en las SharedPreferences:

```
val prefs = getSharedPreferences("MisPrefs", Context.MODE_PRIVATE)
val editor = prefs.edit()
editor.putBoolean("clave", true)
// Podemos utilizar la función apply() en lugar de commit().
// apply() funciona de manera asíncrona, lo que quiere decir
// que no bloquea el hilo principal de la aplicación (commit sí).
// La operación de guardar los datos en las SharedPreferences con
// apply() se realiza por tanto en segundo plano.
editor.apply()
```

   Leer un ***valor booleano*** de las SharedPreferences:

```
val prefs = getSharedPreferences("MisPrefs", Context.MODE_PRIVATE)
val valor = prefs.getBoolean("clave", false)
```

3. Guardar un ***valor entero*** en las SharedPreferences:

```
val prefs = getSharedPreferences("MisPrefs", Context.MODE_PRIVATE)
val editor = prefs.edit()
editor.putInt("clave", 123)
editor.apply()
```

Leer un ***valor entero*** de las SharedPreferences:

```
val prefs = getSharedPreferences("MisPrefs", Context.MODE_PRIVATE)
val valor = prefs.getInt("clave", 0)
```

Context.MODE_PRIVATE

Context.MODE_PRIVATE es una constante que se utiliza para abrir las SharedPreferences en modo privado en Android. Este modo indica que las SharedPreferences solo son accesibles por la aplicación que las crea y que ningún otro proceso o aplicación puede acceder a ellas.

Cuando se utiliza Context.MODE_PRIVATE para abrir las SharedPreferences, se crea un archivo en la memoria interna del dispositivo, que se almacena en la carpeta data/data/[nombre de paquete]/shared_prefs/[nombre del archivo].xml. Este archivo sólo es accesible por la aplicación que lo creó y no puede ser visto ni modificado por otras aplicaciones o usuarios.

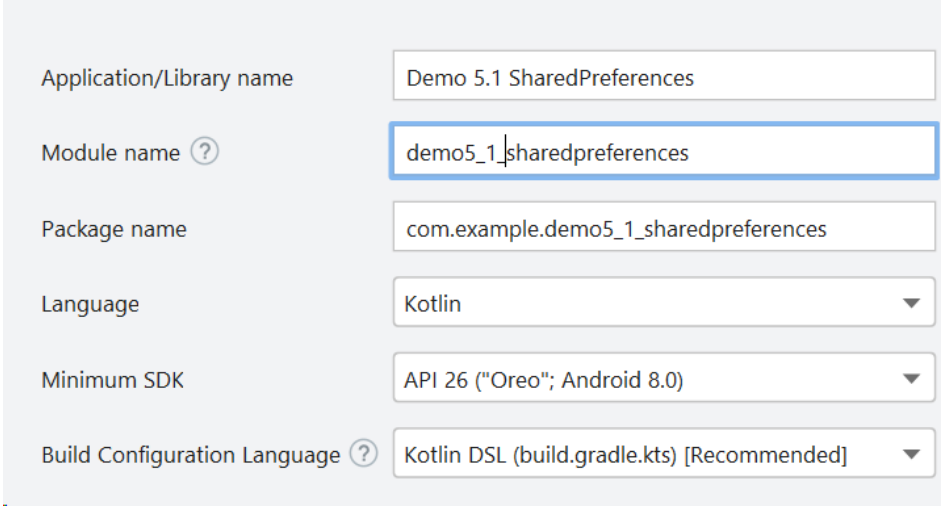
Además del modo privado, existen otros modos que se pueden utilizar al abrir las SharedPreferences. Aquí mostramos los modos más comunes:

- Context.MODE_WORLD_READABLE: Permite que otras aplicaciones puedan leer las SharedPreferences, pero no modificarlas.
- Context.MODE_WORLD_WRITEABLE: Permite que otras aplicaciones puedan leer y modificar las SharedPreferences.
- Context.MODE_MULTI_PROCESS: Indica que se permitirá el acceso concurrente a las SharedPreferences desde varios procesos. Este modo no se recomienda ya que puede causar problemas de sincronización y seguridad.
- Context.MODE_APPEND: Indica que se agregarán nuevos datos a los ya existentes en las SharedPreferences, en lugar de sobrescribirlos.

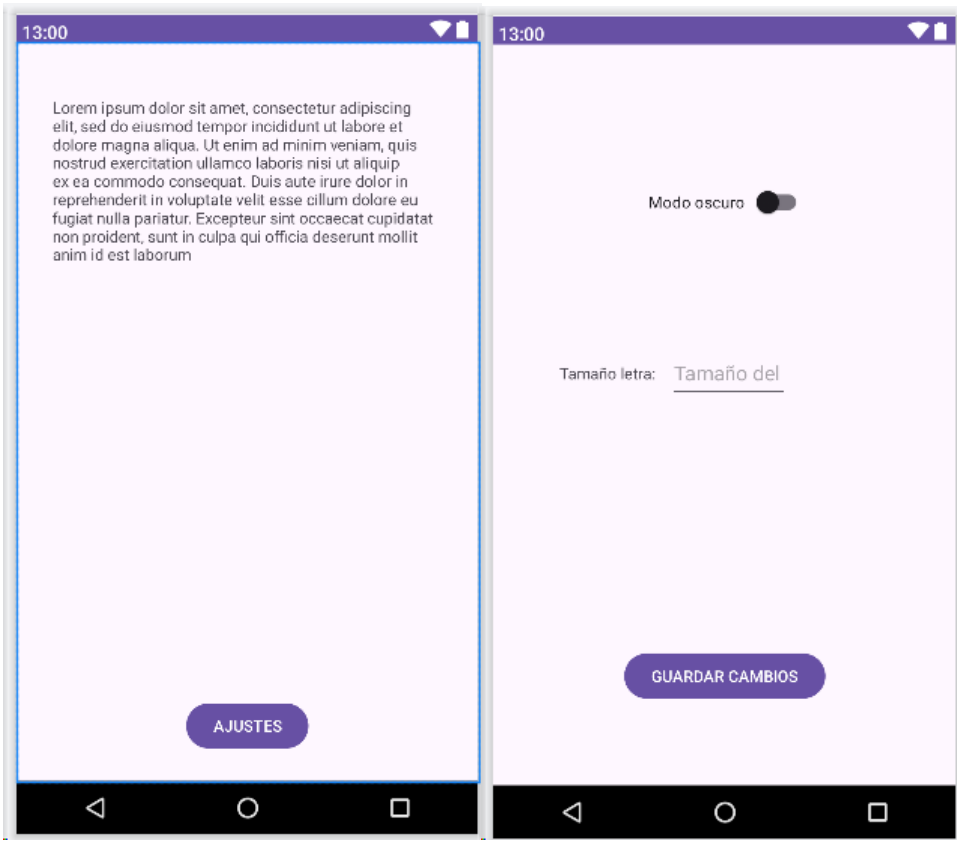
Es importante tener en cuenta que algunos modos, como Context.MODE_WORLD_READABLE y Context.MODE_WORLD_WRITEABLE, no son seguros y pueden causar problemas de seguridad en la aplicación. Por lo tanto, se recomienda utilizar siempre el modo privado (Context.MODE_PRIVATE) a menos que sea absolutamente necesario otro modo y se tenga en cuenta los riesgos potenciales.

Objetivo de la demo

Vamos a crear un nuevo proyecto en Android Studio para esta unidad 5.



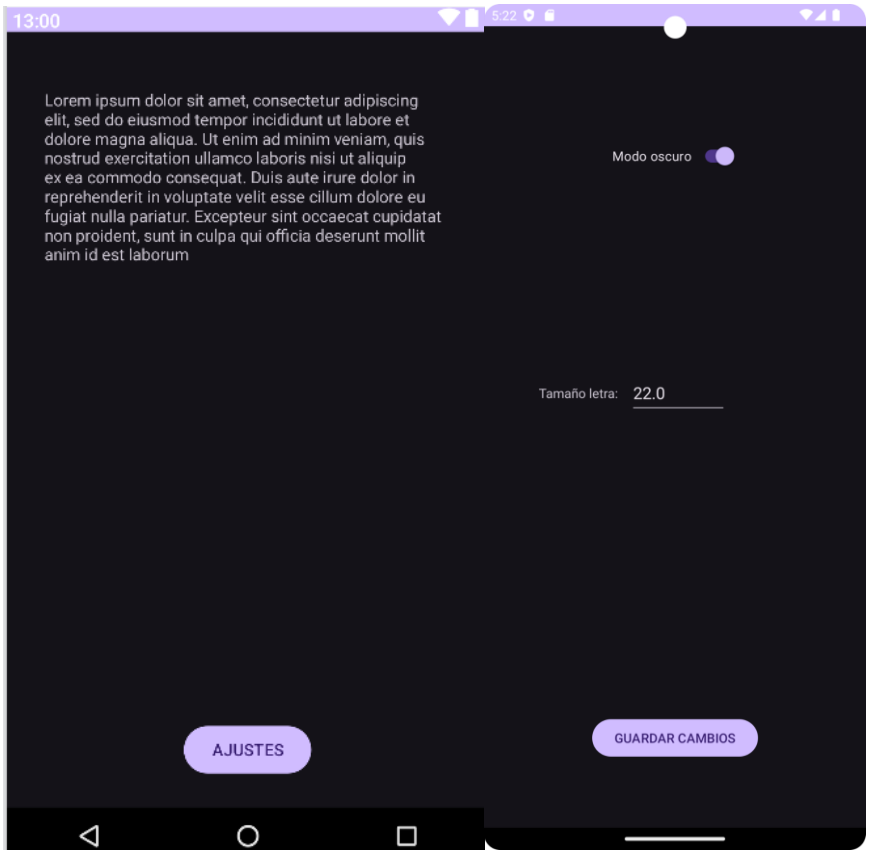
En el primer módulo, que renombraremos de **app** a **demo5_1_shared_preferences**, vamos a crear 2 activities, con el aspecto que podemos ver a continuación:



Al pinchar en el botón AJUSTES de la MainActivity se abrirá una segunda ventana (Ajustes) que nos permitirá indicar el modo de la aplicación (OSCURO/CLARO) y el tamaño de letra del texto que se muestra.

Al pinchar en 'Guardar cambios' en la activity 'Ajustes' se guardarán los valores del switch y del campo de texto que indica el tamaño de letra en el archivo SharedPreferences.

Al volver a la MainActivity o si cerramos y volvemos a abrir la aplicación, se leerán los valores del fichero SharedPreferences y se mostrará el tamaño de letra y el modo que corresponda:



Desarrollo

Archivo activity_main.xml:

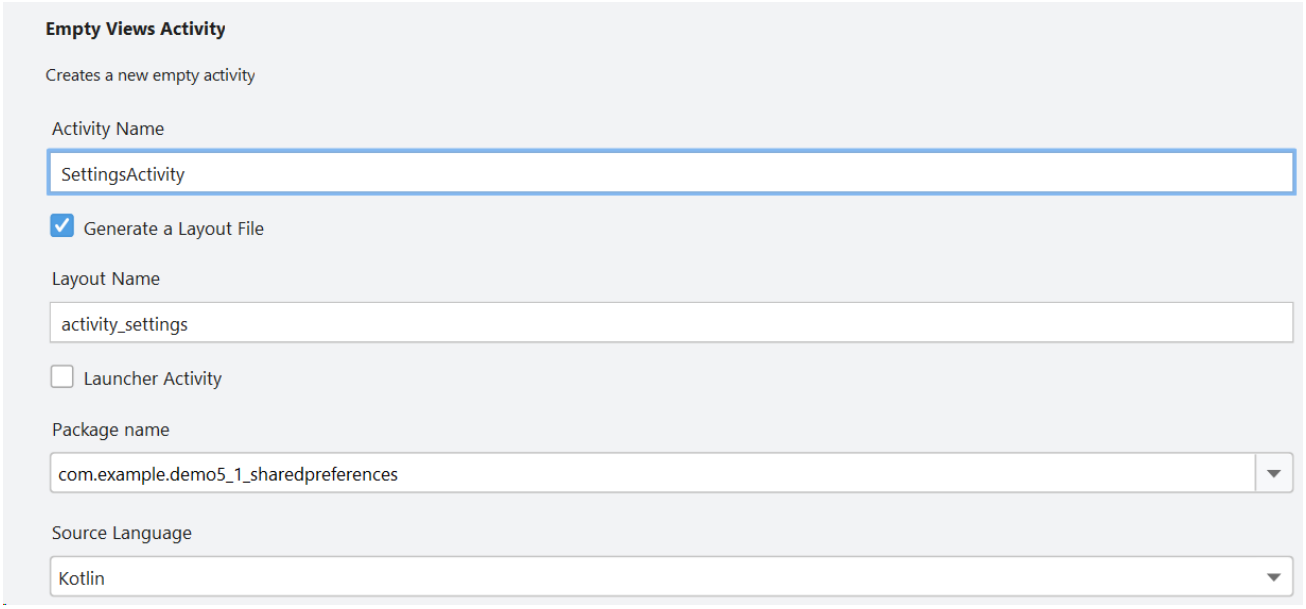
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textoInicial"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginStart="32dp"
        android:layout_marginTop="48dp"
        android:layout_marginEnd="32dp"
        android:text="@string/texto_inicial"
        app:layout_constraintBottom_toTopOf="@+id/btnAjustes"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
```

```
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/btnAjustes"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="150dp"
        android:layout_marginEnd="150dp"
        android:layout_marginBottom="25dp"
        android:text="AJUSTES"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Archivo Strings.xml

```
<resources>

    <string name="app_name">Demo 5.1 SharedPreferences</string>

    <string name="texto_inicial">Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute
irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit
anim id est laborum</string>

</resources>
```

Creamos una segunda Activity denominada SettingsActivity...



Archivo MainActivity.kt

De momento sólo activamos ViewBinding y añadimos el código para que se abra la 2ª activity:

```
package com.example.demo5_1_sharedpreferences

import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import com.example.demo5_1_sharedpreferences.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

   private lateinit var binding: ActivityMainBinding
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       binding = ActivityMainBinding.inflate(layoutInflater) // inflate the layout
       setContentView(binding.root) // set the content view

       // Asociamos un listener al botón AJUSTES que abrirá SettingsActivity
       binding.btnAjustes.setOnClickListener {
           startActivity(Intent(this, SettingsActivity::class.java))
       }
   }

} // Fin de la clase MainActivity
```

Archivo activity_settings.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.appcompat.widget.SwitchCompat
        android:id="@+id/switch1"
        android:layout_width="135dp"
        android:layout_height="48dp"
        android:layout_marginStart="142dp"
        android:layout_marginTop="116dp"
        android:layout_marginEnd="142dp"
        android:text="Modo oscuro"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/btnGuardar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="96dp"
        android:layout_marginEnd="96dp"
        android:layout_marginBottom="73dp"
        android:text="GUARDAR CAMBIOS"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="59dp"
        android:layout_marginTop="91dp"
        android:layout_marginBottom="208dp"
        android:text="Tamaño letra:"
        app:layout_constraintBottom_toTopOf="@+id/btnGuardar"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/switch1" />

    <EditText
        android:id="@+id/etTamLetra"
        android:layout_width="wrap_content"
        android:layout_height="48dp"
        android:layout_marginStart="12dp"
        android:layout_marginEnd="150dp"
        android:ems="5"
        android:hint="@string/noSpeakableText"
        android:inputType="number"
        app:layout_constraintBottom_toBottomOf="@+id/textView"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
```

```
        app:layout_constraintStart_toEndOf="@+id/textView"
        app:layout_constraintTop_toTopOf="@+id/textView" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Archivo SettingsActivity.kt

```
package com.example.demo5_1_sharedpreferences

import android.content.Context
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import com.example.demo5_1_sharedpreferences.databinding.ActivitySettingsBinding


class SettingsActivity : AppCompatActivity() {

    // Activamos ViewBinding en SettingsActivity
    private lateinit var binding: ActivitySettingsBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivitySettingsBinding.inflate(layoutInflater) // inflate the layout

        // LEEMOS LAS PREFERENCIAS PARA QUE APAREZCAN CORRECTAMENTE EN EL
        // SWITCH Y EN EL EDITTEXT DEL TAMAÑO DE LETRA
        val prefs = getSharedPreferences("MisPrefs", Context.MODE_PRIVATE)  // MisPrefs:
nombre del fichero de preferencias
        binding.switch1.isChecked = prefs.getBoolean("oscuro", false)
        binding.etTamLetra.setText(prefs.getFloat("tam_letra", 12F).toString())

        setContentView(binding.root) // set the content view

        // Asociamos un listener al botón GUARDAR que guardará los datos en
SharedPreferences
        // (valores actuales del switch y del tamaño de letra)
        binding.btnGuardar.setOnClickListener {
            // Guardamos los datos en SharedPreferences
            val prefs = getSharedPreferences("MisPrefs", MODE_PRIVATE).edit()
            prefs.putBoolean("oscuro", binding.switch1.isChecked)
            prefs.putFloat("tam_letra", binding.etTamLetra.text.toString().toFloat())
            prefs.apply()
            finish() // Cerramos SettingsActivity
        }
    }
} // Fin de la clase SettingsActivity
```

Modificación de MainActivity.kt

```
package com.example.demo5_1_sharedpreferences

import android.content.Context
import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import androidx.appcompat.app.AppCompatDelegate
import com.example.demo5_1_sharedpreferences.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d("MIAPP", "onCreate")
        binding = ActivityMainBinding.inflate(layoutInflater) // inflate the layout
        setContentView(binding.root) // set the content view

        // Asociamos un listener al botón AJUSTES que abrirá SettingsActivity
        binding.btnAjustes.setOnClickListener {
            startActivity(Intent(this, SettingsActivity::class.java))
        }
    }

    // Sobreescribimos el método onResume para que al inicio y cuando volvamos de
    // SettingsActivity se actualice el tamaño de letra y el modo oscuro o claro
    override fun onResume() {
        super.onResume()
        Log.d("MIAPP", "onResume")
        leerPreferencias()
    }

    // Función que lee las preferencias y establece el tamaño de letra y el modo oscuro o
    // claro en la aplicación
    private fun leerPreferencias() {
        val prefs = getSharedPreferences("MisPrefs", Context.MODE_PRIVATE)
        val tamLetra = prefs.getFloat("tam_letra", 12F)
        val oscuro = prefs.getBoolean("oscuro", false)
        binding.textoInicial.textSize = tamLetra
        if (oscuro) {
            // Establecemos el modo noche en la aplicación
            AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_YES)
        } else {
            // Establecemos el modo día en la aplicación
            AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_NO)
        }
    }

} // Fin de la clase MainActivity
```

Para ver el fichero, desde Device Explorer vamos a la ruta:

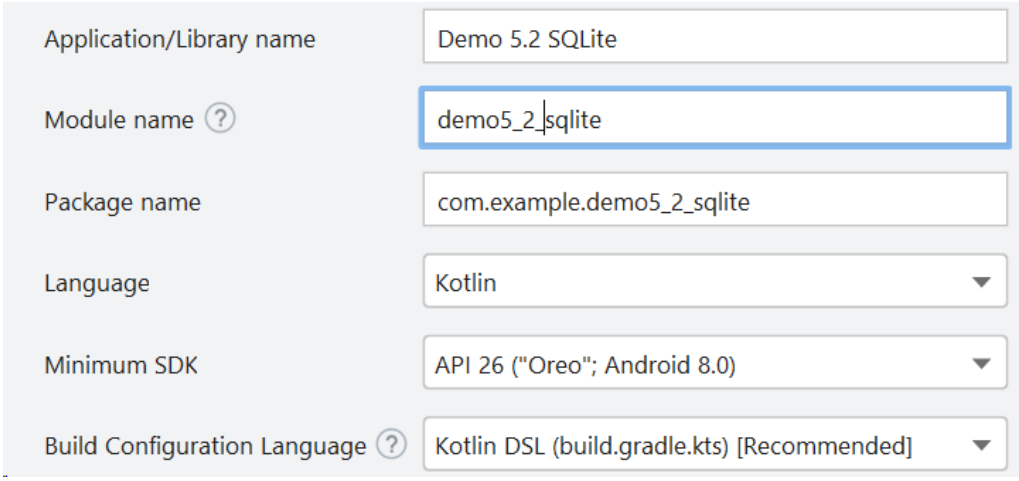
_\data\data\com.example.demo5_1_sharedpreferences\shared_prefs\MisPrefs.xml

MisPrefs.xml

```xml
<?xml version='1.0' encoding='utf-8' standalone='yes'
<map>
    <float name="tam_letra" value="18.0" />
    <boolean name="oscuro" value="true" />
</map>
```

DEMO 5.2. ALMACENAMIENTO EN SQLITE

Creamos un nuevo módulo:



EJEMPLO SQLITE TABLA USUARIOS

En este apartado vamos a ver el funcionamiento básico de SQLite en Android: cómo crear una base de datos, insertar algún dato de prueba y comprobar que todo funciona correctamente.

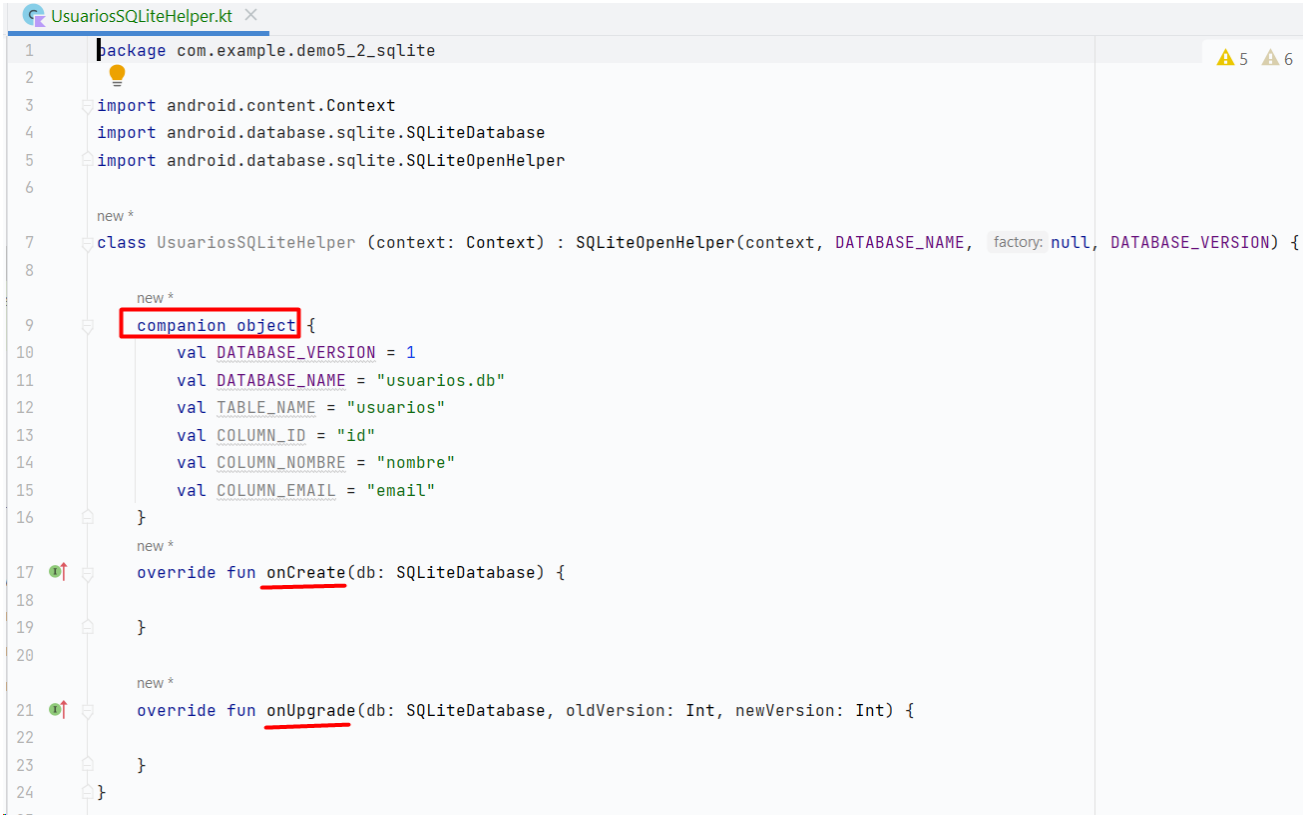
Clases SQLiteOpenHelper y UsuariosSQLiteHelper.kt

En Android, la forma típica para crear, actualizar, y conectar con una base de datos SQLite es a través de una clase creada por nosotros que herede de `SQLiteOpenHelper` que se adapte a las necesidades concretas de nuestra aplicación.

La clase `SQLiteOpenHelper` tiene tan sólo 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 respectivamente.

Como ejemplo, vamos a crear una base de datos muy sencilla llamada `usuarios.db`, con una sóla tabla llamada `usuarios` que contendrá 3 campos: `id`, `nombre` y `email`. Para ellos, vamos a crear una clase derivada de `SQLiteOpenHelper` que llamaremos `UsuariosSQLiteHelper`, donde sobrescribiremos los métodos `onCreate()` y `onUpgrade()` para adaptarlos a la estructura de datos indicada:

Esqueleto de la clase ***UsuariosSQLiteHelper.kt***:



En la captura podemos ver que la clase **UsuariosSQLiteHelper** hereda de **SQLiteOpenHelper** y define un **companion object** (equivalente a **static**) con el nombre y la versión de la base de datos, el nombre de la tabla y las columnas de la misma. Además, se sobreescriben los métodos **onCreate** y **onUpgrade** para realizar las acciones necesarias al crear y actualizar la base de datos, respectivamente.

```
package com.example.demo5_2_sqlite.db

import android.content.Context
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper

class UsuariosSQLiteHelper (context: Context) : SQLiteOpenHelper(context, DATABASE_NAME,
null, DATABASE_VERSION) {

   companion object {
       const val DATABASE_VERSION = 1
       const val DATABASE_NAME = "usuarios.db"
       const val TABLE_NAME = "usuarios"
       const val COLUMN_ID = "id"
       const val COLUMN_NOMBRE = "nombre"
       const val COLUMN_EMAIL = "email"
   }

   private val sqlCreateTable = "CREATE TABLE $TABLE_NAME (" +
           "$COLUMN_ID INTEGER PRIMARY KEY AUTOINCREMENT, " +
           "$COLUMN_NOMBRE TEXT, " +
           "$COLUMN_EMAIL TEXT)"
   override fun onCreate(db: SQLiteDatabase?) {
       // Aquí creamos la tabla usuarios (id, nombre, email)
```

```
        db?.execSQL(sqlCreateTable)
    }

    override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion: Int) {
        // Se elimina la versión anterior de la tabla
        db?.execSQL("DROP TABLE IF EXISTS $TABLE_NAME")
        // Se crea la nueva versión de la tabla
        db?.execSQL(sqlCreateTable)
    }
}
```

La simple creación de un objeto `UsuariosSQLiteHelper` puede tener varios efectos:

- Si la **base de datos ya existe** y su versión actual coincide con la solicitada simplemente se realizará la conexión con ella.
- Si la **base de datos existe pero su versión actual es anterior**, 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**, se llamará automáticamente al método onCreate() para crearla y se conectará con la base de datos creada.

Estructura del proyecto: patrón DAO (Data Access Object)

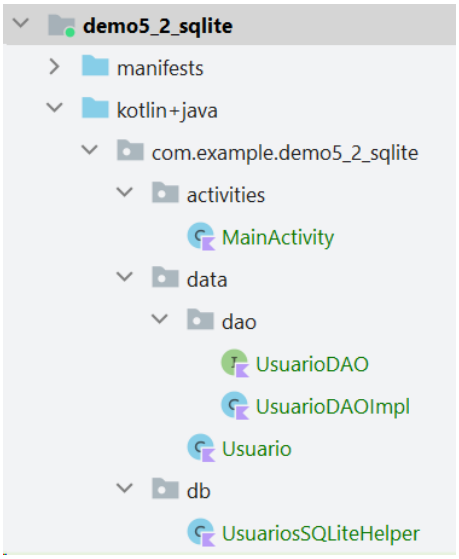
La estructura que vamos a dar a nuestro proyecto va a seguir un patrón de diseño muy utilizado denominado DAO (Data Access Object).

1. La clase **`UsuariosSQLiteOpenHelper`** la vamos a mover a un nuevo paquete denominado **`db`** (de database).
2. Creamos un paquete **`data`** y en él una *data class* llamada **`Usuario`** en la que vamos a declarar 3 atributos que se corresponden con las 3 columnas de la tabla usuarios: **`id`**, **`nombre`** y **`email`**:

   ```
   package com.example.demo5_2_sqlite.data
   data class Usuario(
       var id: Int? = null,  // Dejarlo como null permite que SQLite lo genere automáticamente (auto-increment)
       var nombre: String,
       var email: String
   )
   ```

   Como hemos definido el id como AUTOINCREMENT, cuando creemos Usuarios lo haremos sin incluir el ID. Para que SQLite entienda que lo debe autogenerar, debemos poner el tipo como nullable (que admite el valor null, Int?) y asignarle el valor por defecto null.

3. Dentro del paquete **`data`** creamos otro llamado **`dao`** y en él una interfaz llamada **`UsuarioDAO`** y una clase **`UsuarioDAOImpl`** (Impl, de implementación). La estructura del proyecto en este punto sería algo así:



4. `UsuarioDAO.kt` va a contener únicamente la declaración de las cabeceras de los métodos más habituales que utilizamos para trabajar con bases de datos (CRUD):

```
package com.example.demo5_2_sqlite.data.dao

import com.example.demo5_2_sqlite.data.Usuario

// Interfaz que define las operaciones que se pueden realizar sobre la tabla
usuarios
interface UsuarioDAO {
   fun insertarUsuario(usuario: Usuario)
   fun leerUsuarioPorId(id: Int): Usuario
   fun leerUsuarios(): List<Usuario>
   fun actualizarUsuario(usuario: Usuario)
   fun borrarUsuario(id: Int)
}
```

5. Creamos otra clase en el paquete `UsuarioDAOImpl.kt` va a contener la implementación de los métodos definidos por la interfaz UsuarioDAO, en este caso personalizada para SQLite. Necesita en el constructor una instancia de la clase UsuariosSQLiteHelper e implementar la interfaz del punto anterior, UsuarioDAO.

```
package com.example.demo5_2_sqlite.data.dao

import android.content.ContentValues
import android.view.View
import com.example.demo5_2_sqlite.data.Usuario
import com.example.demo5_2_sqlite.db.UsuariosSQLiteHelper
import com.google.android.material.snackbar.Snackbar

/**
* Clase que implementa la interfaz UsuarioDAO
* @param view: una vista de la MainActivity, necesaria para mostrar Snackbars
* @param dbHelper: instancia de la clase UsuariosSQLiteHelper
*/
class UsuarioDAOImpl (
   private val view: View,
```

```
    private val dbHelper: UsuariosSQLiteHelper) : UsuarioDAO {

    override fun insertarUsuario(usuario: Usuario) {
        TODO()
    }

    override fun leerUsuarios(): List<Usuario> {
        TODO()
    }

    override fun actualizarUsuario(usuario: Usuario) {
        TODO()
    }

    override fun borrarUsuario(id: Int) {
        TODO()
    }

    override fun leerUsuarioPorId(id: Int): Usuario {
        TODO()
    }

}
```

6. Para realizar una prueba, vamos a codificar la función **`insertarUsuario()`**.

   En primer lugar, necesitamos un objeto **`SQLiteDatabase`**, que es una referencia a la propia base de datos. Lo obtendremos utilizando la propiedad **`writableDatabase`** (equivalente a llamar a la función **`getWritableDatabase()`**) de la clase **`SQLiteOpenHelper`**.

   A continuación tenemos 2 opciones:

   A. Llamar al método **`execSQL`**.

   B. Llamar al método **`insert`**, que tiene la ventaja de que nos devuelve el id generado por SQLite (ojo, como un Long en lugar de un Int).

   El método **`insertarUsuario()`** quedaría así:

```
    override fun insertarUsuario(usuario: Usuario) {
        // Abrimos la base de datos para lectura/escritura
        val db = dbHelper.writableDatabase
        // Opción A) Insertamos los datos en la tabla Usuarios
        // -----------------------------------------------
        // Forma abreviada (sin usar las constantes definidas en
UsuariosSQLiteHelper):
        // db.execSQL("INSERT INTO usuarios (nombre, email) +
        //         "VALUES ('${usuario.nombre}', '${usuario.email}')")
        // Forma más correcta
        db.execSQL("INSERT INTO ${UsuariosSQLiteHelper.TABLE_NAME} (" +
                "${UsuariosSQLiteHelper.COLUMN_NOMBRE}, " +
                "${UsuariosSQLiteHelper.COLUMN_EMAIL}) " +
                "VALUES ('${usuario.nombre}', '${usuario.email}')")

        // Opción B) Insertar un registro en la tabla utilizando el método insert...
```

```
        // ------------------------------------------------
        // Necesitamos un objeto de tipo ContentValues. ContentValues es una
coleccion de
        // pares clave/valor, donde la clave es el nombre de la columna y el valor
es el
        // valor a insertar en dicha columna
        val nuevoUsuario = ContentValues()
        nuevoUsuario.put(UsuariosSQLiteHelper.COLUMN_NOMBRE, usuario.nombre)
        nuevoUsuario.put(UsuariosSQLiteHelper.COLUMN_EMAIL, usuario.email)
        // Insertamos el usuario
        val idInsertado = db.insert(UsuariosSQLiteHelper.TABLE_NAME, null,
nuevoUsuario)

        // Otra forma de añadir valores al ContentValues: utilizando apply
//        val nuevoUsuario = ContentValues().apply {
//            put(UsuariosSQLiteHelper.COLUMN_NOMBRE, usuario.nombre)
//            put(UsuariosSQLiteHelper.COLUMN_EMAIL, usuario.email)
//        }

        // Mostramos el id del usuario insertado
        Snackbar.make(view, "Usuario insertado con id: $idInsertado",
            Snackbar.LENGTH_LONG).show()

    } // Fin de insertarUsuario
```

Nota: Si nos fijamos, al utilizar ContentValues no nos hace falta crear la sentencia SQL.

7. Modificamos lo necesario en MainActivity para crear una instancia de UsuarioDAOImpl, insertar un usuario de prueba y mostrar el resultado:

```
package com.example.demo5_2_sqlite.activities

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.example.demo5_2_sqlite.data.Usuario
import com.example.demo5_2_sqlite.data.dao.UsuarioDAOImpl
import com.example.demo5_2_sqlite.databinding.ActivityMainBinding
import com.example.demo5_2_sqlite.db.UsuariosSQLiteHelper

class MainActivity : AppCompatActivity() {
    // Variable para el viewBinding
    private lateinit var binding: ActivityMainBinding

    // Creamos una instancia de la clase UsuariosSQLiteHelper
    private val dbHelper = UsuariosSQLiteHelper(this)
    // Declaramos una variable de tipo UsuarioDAOImpl (donde se implementan las
    // operaciones con la base de datos)
    private lateinit var operaciones: UsuarioDAOImpl
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
```

```
        setContentView(binding.root)
        // Creamos una instancia de la clase UsuarioDAOImpl.
        // Pasamos como parámetros una vista (necesaria para Snackbars)
        // y el dbHelper (instancia de UsuariosSQLiteHelper)
        operaciones = UsuarioDAOImpl(binding.root, dbHelper)

        // Creamos el usuario a insertar
        val usuario = Usuario(
            nombre = "Ernesto Pérez",
            email = "ernesto@a.com"
        )
        // Insertamos el usuario en la base de datos
        operaciones.insertarUsuario(usuario)
    } // Fin onCreate
```

8. Para comprobar si se ha insertado correctamente, ejecutamos la aplicación en el emulador. Veremos que simplemente aparece el TextView por defecto que se crea al utilizar la plantilla EmptyViewsActivity.

   ¿Y ***dónde está la base de datos SQLite que se supone acabamos de crear***? ¿Cómo podemos comprobar que todo ha ido bien y que los registros se han insertado correctamente?

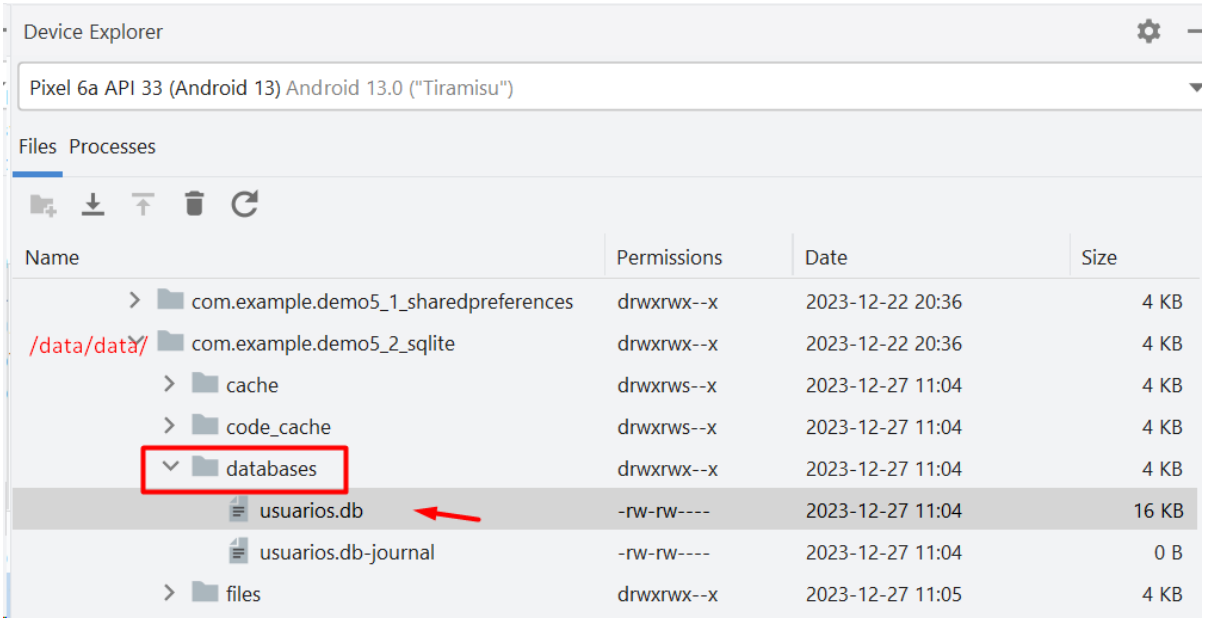
   Todas las bases de datos SQLite creadas por aplicaciones Android utilizando este método se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón:

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

   En mi caso, la base de datos se almacena en la ruta siguiente:

   ```
   /data/data/com.example.demo5_2_sqlite/databases/usuarios.db
   ```

   Podemos ir desde Android Studio a la pestaña '*Device File Explorer*', que generalmente se encuentra en la parte lateral derecha, y buscar nuestra ruta:



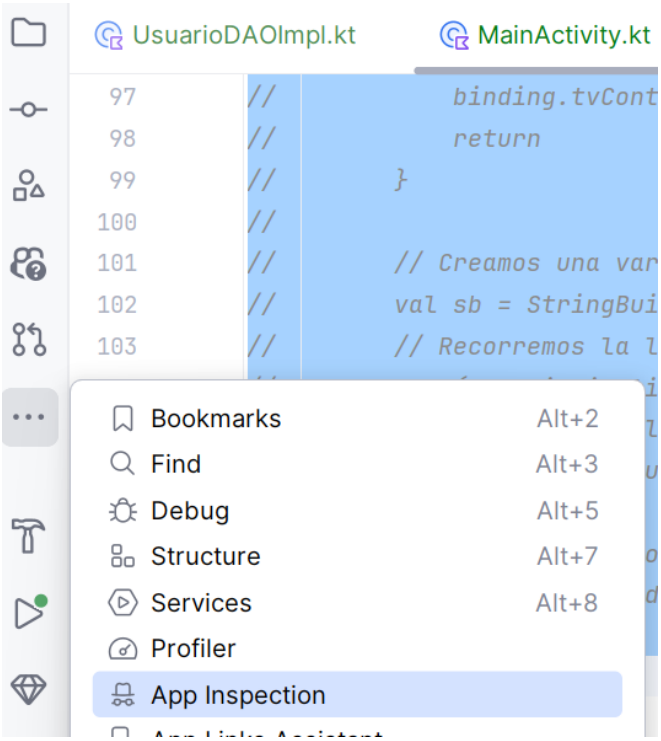
   Con esto ya comprobamos al menos que el fichero de nuestra base de datos se ha creado en la ruta correcta. Ya sólo nos queda comprobar que tanto la tabla creada como el usuario insertado también

se han incluido correctamente en la base de datos. Para ello podemos recurrir a dos posibles métodos:

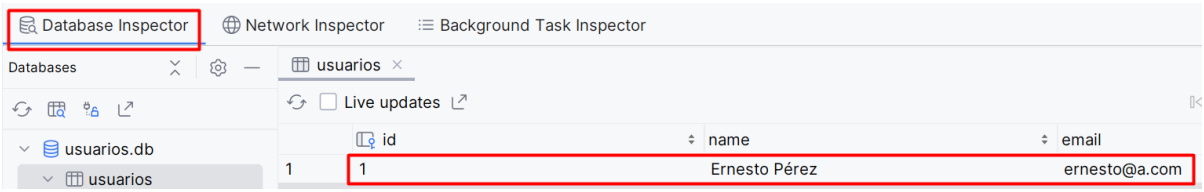
A. Transferir la base de datos a nuestro PC y consultarla con cualquier administrador de bases de datos SQLite (por ejemplo, DBBrowser for SQLite).
B. Acceder directamente desde Android Studio utilizando la herramienta ***Database Inspector***.

Vamos a realizar la comprobación con el segundo método...

En la lateral izquierda de Android Studio podemos encontrar el botón ***App Inspection***:



En la pestaña Database Inspector, veremos que se ha insertado un registro en la tabla usuarios de la base de datos usuarios.db.



La clase ContentValues

ContentValues es una clase de Android que se usa para almacenar pares clave-valor. Es como un "diccionario" donde se asocian nombres de columnas con valores.

Se utiliza principalmente cuando queremos insertar o actualizar datos en una base de datos SQLite sin escribir directamente una consulta SQL, lo que permite evitar errores y hacer el código más seguro y fácil de leer.

execSQL()	**con ContentValues y método insert()**
`val db = dbHelper.writableDatabase` `val sql = "INSERT INTO usuarios (name,`	`val db = dbHelper.writableDatabase` `val valores = ContentValues().apply {`

`email) VALUES ('Juan Pérez', 'juan@example.com')"`
`db.execSQL(sql)`
	`put("name", "Juan Pérez")`
	`put("email", "juan@example.com")`
	`}`
	`db.insert("usuarios", null, valores)`

Principal problema: Estamos concatenando valores manualmente en la consulta SQL, lo cual es peligroso y puede generar inyección SQL (un ataque donde alguien introduce código malicioso en la base de datos).	**Ventajas:** - Evita inyección SQL, ya que los datos se pasan como parámetros y no dentro de la consulta SQL. - Maneja automáticamente comillas y caracteres especiales, evitando errores inesperados. - Código más legible y organizado.

Método insert():

```
fun insert(
    table: String,             // Nombre de la tabla donde insertar
    nullColumnHack: String?,   // Columna a la que asignar NULL si no hay valores
    values: ContentValues      // Datos a insertar en la tabla (pares clave-valor)
): Long   // Devuelve el ID de la nueva fila insertada o -1 si hubo error
```

El parámetro `nullColumnHack` permite insertar una fila con valores nulos en caso de que el ContentValues esté vacío.

Normalmente, indicaremos el valor `null` para este parámetro, ya que lo habitual es insertar datos en al menos una columna. Sin embargo, si intentamos insertar una fila completamente vacía (sin datos en `ContentValues`), SQLite generaría un error, ya que no se pueden insertar filas sin valores.

Ejemplo:

Kotlin:
`val valores = ContentValues() // Sin datos`
 	
`val id = db.insert("usuarios", "name", valores)`
	SQLite ejecutaría...
	`INSERT INTO usuarios (name) VALUES (NULL);`

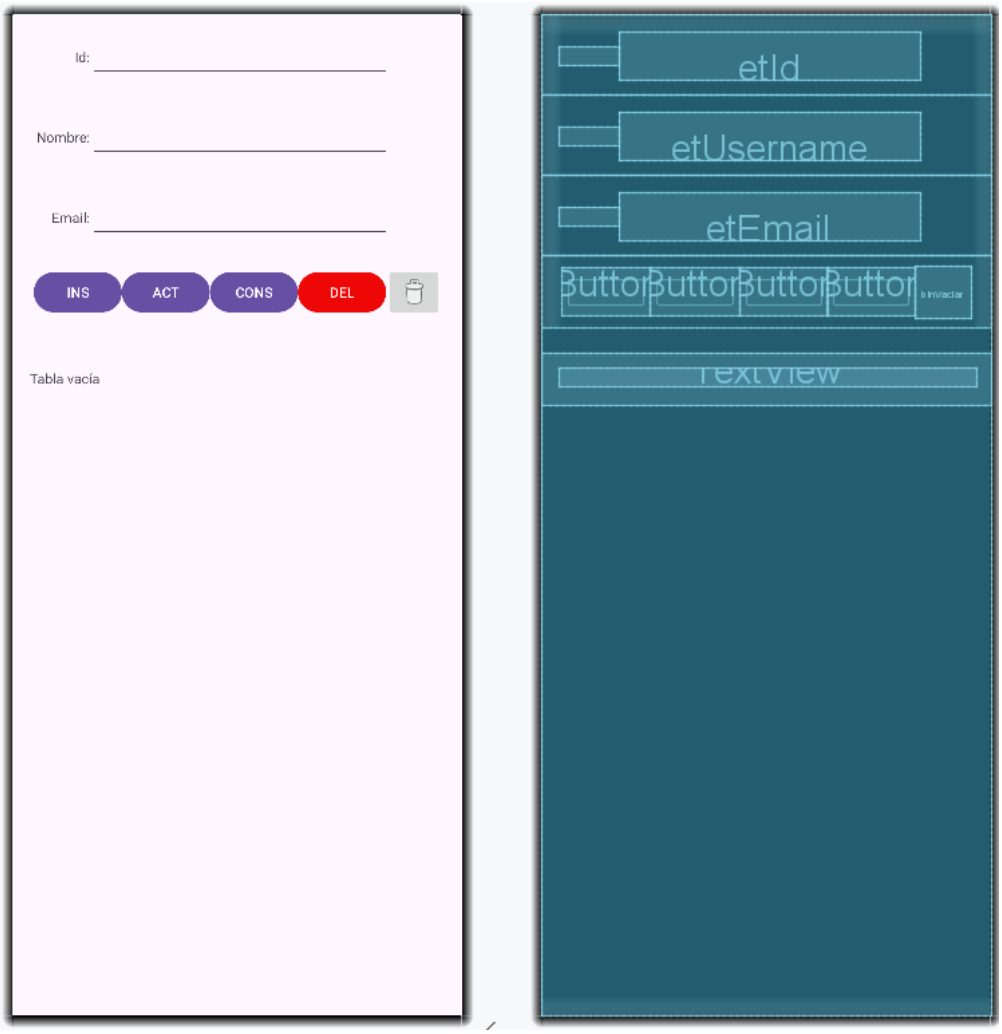
OPERACIONES CRUD

Como vimos en el apartado anterior, el método **`execSQL()`** nos permite ejecutar sentencias como insertar registros, crear una tabla o borrarla. También podemos utilizarlo para borrar o actualizar registros de la tabla:

```
//Eliminar un registro

db.execSQL("DELETE FROM usuarios WHERE id=6")

//Actualizar un registro

db.execSQL("UPDATE usuarios SET nombre='usunuevo' WHERE id=6")
```

De la misma manera que disponemos del método **`insert()`**, también tenemos los métodos **`update()`** y **`delete()`** de la clase **`SQLiteDatabase`**.

Antes de empezar a utilizarlos, vamos a modificar el layout `activity_main.xml` para que se muestre de la siguiente manera:



El archivo xml quedaría de la siguiente manera:

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:padding="16dp">
        <TextView
            android:id="@+id/textView3"
            android:layout_width="60dp"
            android:gravity="end"
            android:layout_height="wrap_content"
```

```
            android:text="Id:" />

        <EditText
            android:id="@+id/etId"
            android:layout_width="300dp"
            android:layout_height="48dp"
            android:ems="10"
            android:inputType="number" />

    </LinearLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:padding="16dp">

        <TextView
            android:id="@+id/textView"
            android:layout_width="60dp"
            android:gravity="end"
            android:layout_height="wrap_content"
            android:text="Nombre:" />

        <EditText
            android:id="@+id/etName"
            android:layout_width="300dp"
            android:layout_height="48dp"
            android:ems="10"
            android:inputType="textPersonName" />
    </LinearLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:padding="16dp">

        <TextView
            android:id="@+id/textView2"
            android:layout_width="60dp"
            android:gravity="end"
            android:layout_height="wrap_content"
            android:text="Email:" />

        <EditText
            android:id="@+id/etEmail"
            android:layout_width="300dp"
            android:layout_height="48dp"
            android:ems="10"
            android:inputType="textEmailAddress" />
    </LinearLayout>

    <LinearLayout
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:paddingVertical="10dp"
        android:gravity="center">

        <Button
            android:id="@+id/btnInsertar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"

            android:text="INS" />

        <Button
            android:id="@+id/btnActualizar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="ACT" />

        <Button
            android:id="@+id/btnConsultar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="CONS" />

        <Button
            android:id="@+id/btnEliminar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:backgroundTint="#EE0707"
            android:text="DEL" />

        <ImageButton
            android:id="@+id/btnVaciar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:srcCompat="@android:drawable/ic_menu_delete" />

    </LinearLayout>

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp"
        android:layout_marginTop="24dp">
        <TextView
            android:id="@+id/tvContenidoTabla"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Tabla vacía" />

    </ScrollView>
</LinearLayout>
```

Método para consultar todos los usuarios

Añadimos la codificación de la función leerUsuarios() que nos devolverá una Lista de Usuarios:

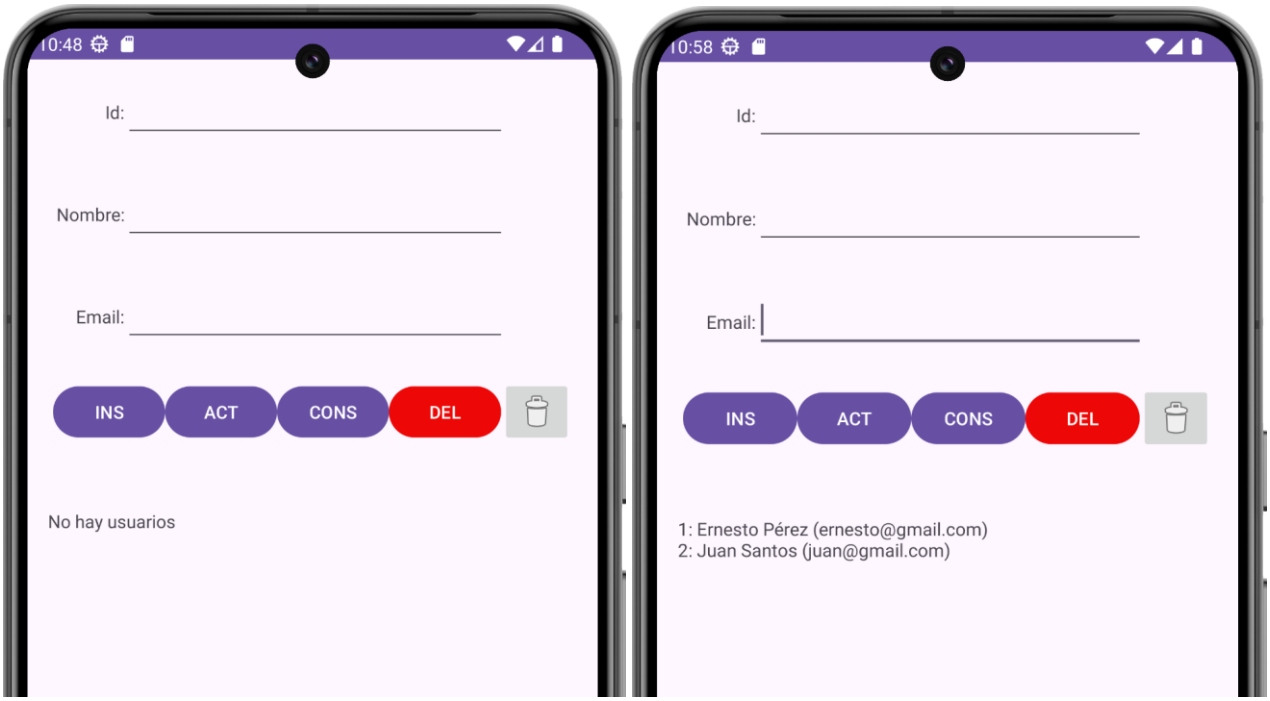
```
    override fun leerUsuarios(): List<Usuario> {
        val listaUsuarios = mutableListOf<Usuario>()

        // Abrimos la base de datos en modo lectura
        val db = dbHelper.readableDatabase

        // Ejecutamos la consulta
        val cursor = db.rawQuery(
            "SELECT * FROM ${UsuariosSQLiteHelper.TABLE_NAME}",
            null
        )
        // El segundo parámetro de rawQuery es un array de Strings con los valores de los
        // parámetros (placeholders) de la consulta. Lo veremos en leerUsuarioPorId.
        // En este caso no hay parámetros, por lo que se pasa null.

        // ------------------------------------------------
        // Opción A.
        // ------------------------------------------------
        // Recorremos el cursor
        while (cursor.moveToNext()) {
            // Obtenemos los valores de los campos del cursor
            val id = cursor.getInt(0)
            val nombre = cursor.getString(1)
            val email = cursor.getString(2)

            // Añadimos el usuario a la lista
            listaUsuarios.add(Usuario(id, nombre, email))
        }
        // Cerramos el cursor
        cursor.close()

        // ------------------------------------------------
        // Opción B.
        // Usamos use{} para asegurarnos de que el cursor se cierre automáticamente
        // ------------------------------------------------
//        cursor.use {
//            // Recorrer el cursor
//            while (cursor.moveToNext()) {
//                // Obtenemos los valores de los campos del cursor
//                val id = cursor.getInt(0)
//                val nombre = cursor.getString(1)
//                val email = cursor.getString(2)
//
//                listaUsuarios.add(Usuario(id, nombre, email))
//            }
//        }

        return listaUsuarios
    } // Fin función leerUsuarios()
```

En MainActivity.kt, añadimos el código necesario para responder al clic en el botón **INS** (Insertar usuario) y **CONS** (Consultar usuarios):

```
package com.example.demo5_2_sqlite.activities

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.example.demo5_2_sqlite.data.Usuario
import com.example.demo5_2_sqlite.data.dao.UsuarioDAOImpl
import com.example.demo5_2_sqlite.databinding.ActivityMainBinding
import com.example.demo5_2_sqlite.db.UsuariosSQLiteHelper

class MainActivity : AppCompatActivity() {
    // ViewBinding
    private lateinit var binding: ActivityMainBinding

    // Creamos una instancia de la clase UsuariosSQLiteHelper
    private val dbHelper = UsuariosSQLiteHelper(this)
    // Declaramos una variable de tipo UsuarioDAOImpl (donde se implementan las operaciones
    // con la base de datos)
    private lateinit var operaciones: UsuarioDAOImpl

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        // Creamos una instancia de la clase UsuarioDAOImpl (que implementa las
        // operaciones con la bdd).
        // Pasamos como parámetros una vista (necesaria para Snackbars)
        // y el dbHelper
        operaciones = UsuarioDAOImpl(binding.root, dbHelper)

        // Llamamos a la función mostrarUsuarios para mostrar al inicio los usuarios
        // en el TextView tvContenidoTabla.
        // mostrarUsuarios() llama a su vez al método leerUsuarios de UsuarioDAOImpl.
        mostrarUsuarios()

        binding.btnInsertar.setOnClickListener {
            // TODO: Comprobar que los campos no están vacíos
            // Creamos un objeto de tipo Usuario
            val usuario = Usuario(
                nombre = binding.etName.text.toString(),
                email = binding.etEmail.text.toString()
            )
            // Insertamos el usuario en la base de datos
            operaciones.insertarUsuario(usuario)

            // Reseteamos los EditText
            resetearEditText()
        }

        binding.btnConsultar.setOnClickListener {
            mostrarUsuarios()
        }
```

```
    } // Fin onCreate

    private fun resetearEditText() {
        binding.etId.setText("")
        binding.etName.setText("")
        binding.etEmail.setText("")
    }

    // Método para modificar el TextView tvContenidoTabla con el formato de los usuarios:
    // id: nombre (email)
    private fun mostrarUsuarios() {
        // Obtenemos la lista de usuarios
        val listaUsuarios = operaciones.leerUsuarios()

        if (listaUsuarios.isEmpty()) {
            binding.tvContenidoTabla.text = "No hay usuarios"
            return
        }

        // Creamos una variable de tipo StringBuilder para ir añadiendo los usuarios
        val sb = StringBuilder()
        // Recorremos la lista de usuarios
        for (usuario in listaUsuarios) {
            // Añadimos el usuario al StringBuilder con el formato indicado
            sb.append("${usuario.id}: ${usuario.nombre} (${usuario.email})\n")
        }
        // Mostramos el contenido del StringBuilder en el TextView tvContenidoTabla
        binding.tvContenidoTabla.text = sb.toString()

    } // Fin mostrarUsuarios

    // Sobrescribimos el método onDestroy para cerrar la conexión con la base de datos
    override fun onDestroy() {
        super.onDestroy()
        // Cerramos la conexión con la base de datos utilizando el método cerrarConexion
        dbHelper.close()
    }

} // Fin MainActivity
```

Ejecución:



Métodos para actualizar y borrar

Los métodos **`update()`** y **`delete()`** se utilizarán de forma muy parecida a **`insert()`**, con la salvedad de que recibirán un parámetro adicional con la condición *WHERE* de la sentencia SQL.

Fichero UsuarioDAOImpl.kt.

actualizarUsuario

```
override fun actualizarUsuario(usuario: Usuario) {
    val db = dbHelper.writableDatabase

    // Comprobamos si el usuario con el ID indicado existe
    if (!comprobarId(usuario.id)) {
        Log.w("SQLite", "No se encontró el usuario con ID ${usuario.id}. No se puede
actualizar.")
        Snackbar.make(
            view,
            "No se encontró el usuario con ID ${usuario.id}. No se puede actualizar.",
            Snackbar.LENGTH_LONG
        ).show()
        return
    }

    // Creamos un objeto de tipo ContentValues con los valores a actualizar (nombre y
email)
    val valoresActualizados = ContentValues().apply {
        put(UsuariosSQLiteHelper.COLUMN_NOMBRE, usuario.nombre)
        put(UsuariosSQLiteHelper.COLUMN_EMAIL, usuario.email)
```

```
    }
    // Actualizamos el usuario en la base de datos
    val clausulaWhere = "${UsuariosSQLiteHelper.COLUMN_ID} = ?"
    val whereArgs = arrayOf(usuario.id.toString())
    val filasActualizadas = db.update(
        UsuariosSQLiteHelper.TABLE_NAME,   // Nombre de la tabla
        valoresActualizados,               // Objeto ContentValues con los valores nuevos
        clausulaWhere,  // Clausula WHERE
        whereArgs       // Array de Strings con los valores de los parámetros de WHERE
    )

    if (filasActualizadas == 0) {
        Log.e("SQLite", "Error inesperado al actualizar el usuario con ID ${usuario.id}")
    } else {
        Log.d("SQLite", "Usuario con ID ${usuario.id} actualizado correctamente")
        Snackbar.make(
            view,
            "Usuario con ID ${usuario.id} actualizado correctamente",
            Snackbar.LENGTH_LONG
        ).show()
    }
} // Fin actualizarUsuario
```

borrarUsuario

```
override fun borrarUsuario(id: Int) {
    val db = dbHelper.writableDatabase

    if (!comprobarId(id)) {
        Log.w("Miapp", "El usuario $id no existe.")
        Snackbar.make(
            vista,
            "El usuario $id no existe.",
            Snackbar.LENGTH_LONG
        ).show()
        return
    }
    val clausulaWhere = "${UsuariosSQLiteOpenHelper.COLUMN_ID} = ?"
    val whereArgs = arrayOf(id.toString())
    val filasBorradas = db.delete(
        UsuariosSQLiteOpenHelper.TABLE_NAME,
        clausulaWhere,
        whereArgs
    )
    if (filasBorradas == 0) {
        Log.e("Miapp", "Error inesperado en borrar")
    } else {
        Log.d("Miapp", "El usuario $id borrado correctamente.")
        Snackbar.make(
            vista,
            "El usuario $id borrado correctamente.",
```

```
            Snackbar.LENGTH_LONG
        ).show()
    }
} // Fin de borrarUsuario
```

comprobarID

Esta es una función auxiliar que nos va a permitir comprobar si el id a utilizar en actualizar/borrar existe realmente en la base de datos.

```
private fun comprobarId(id: Int?): Boolean {
    val db = dbHelper.readableDatabase
    val query = "SELECT * FROM ${UsuariosSQLiteHelper.TABLE_NAME} " +
            "WHERE ${UsuariosSQLiteHelper.COLUMN_ID} = ?"
    val cursor = db.rawQuery(query, arrayOf(id.toString()))

    var existe = cursor.count > 0  // Si hay filas, el usuario existe
    cursor.close()  // Cerramos el cursor para liberar recursos
    return existe
}
```

borrarTodosLosUsuarios

Esta función nos va a permitir borrar todos los registros de la base de datos usuarios. Se ejecutará al pulsar el icono de la papelera en la MainActivity (a la derecha de DEL).

```
override fun borrarTodosLosUsuarios() {
    dbHelper.writableDatabase.delete(
        UsuariosSQLiteHelper.TABLE_NAME,
        null,
        null
    )
}
```

leerUsuarioPorId

Este método nos va a permitir leer un usuario de la base de datos dado su ID y mostrar su nombre y correo electrónico en los EditText correspondientes.

```
override fun leerUsuarioPorId(id: Int): Usuario {
    val db = dbHelper.readableDatabase
    val sql = "SELECT * FROM usuarios WHERE id=?"
    val cursor = db.rawQuery(sql, arrayOf(id.toString()))
    var usuario: Usuario
    // Recorremos el cursor
    if (cursor.moveToNext()) {
        // Obtener valores del cursor
        val nombre = cursor.getString(1)
        val email = cursor.getString(2)

        // Crear el objeto Usuario
        usuario = Usuario(id, nombre, email)
    } else {  // No se ha encontrado el id
        Log.w("Miapp", "NO se encontró el id")
        usuario = Usuario(-1, "No encontrado", "No encontrado")
```

```
    }
    cursor.close()
    return usuario
}
```

MainActivity.kt

```
package com.example.demo5_2_sqlite

import android.os.Bundle
import androidx.activity.enableEdgeToEdge
import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.ViewCompat
import androidx.core.view.WindowInsetsCompat
import com.example.demo5_2_sqlite.data.dao.UsuarioDAOImpl
import com.example.demo5_2_sqlite.databinding.ActivityMainBinding
import com.example.demo5_2_sqlite.model.Usuario

class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    private val dbHelper = UsuariosSQLiteHelper(this)
    private lateinit var operaciones: UsuarioDAOImpl

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        operaciones = UsuarioDAOImpl(binding.root, dbHelper)

        mostrarUsuarios()

        binding.btnInsertar.setOnClickListener {
            //TODO: Comprobar que los campos no estén vacíos
            val usuario = Usuario(
                nombre = binding.etName.text.toString(),
                email = binding.etEmail.text.toString()
            )
            operaciones.insertarUsuario(usuario)
            // Resetear editText
            resetearEditText()
        }

        binding.btnActualizar.setOnClickListener {
            //TODO: Comprobar que los campos no estén vacíos
            val usuario = Usuario(
                binding.etId.text.toString().toInt(),
                binding.etName.text.toString(),
                binding.etEmail.text.toString()
            )
            operaciones.actualizarUsuario(usuario)
            // Resetear editText
```

```
            resetearEditText()
        }

        binding.btnEliminar.setOnClickListener {
            val campoId = binding.etId.text.toString().toInt()
            operaciones.borrarUsuario(campoId)
            resetearEditText()
        }

        binding.btnConsultar.setOnClickListener {
            val campoId = binding.etId.text.toString()
            if (campoId.isEmpty()) {
                resetearEditText()
                mostrarUsuarios()
            } else {
                val usuario = operaciones.leerUsuarioPorId(campoId.toInt())
                // TODO: si es null, mostrar Snackbar
                binding.etName.setText(usuario.nombre)
                binding.etEmail.setText(usuario.email)
            }
        }

        binding.btnVaciar.setOnClickListener {
            operaciones.borrarTodosLosUsuarios()
        }
    } // Fin onCreate

    private fun resetearEditText() {
        binding.etId.setText("")
        binding.etName.setText("")
        binding.etEmail.setText("")
    }

    private fun mostrarUsuarios() {
        val listaUsuarios = operaciones.leerUsuarios()
        if (listaUsuarios.isEmpty()) {
            binding.tvContenidoTabla.text = "No hay usuarios"
            return
        }
        val sb = StringBuilder()
        for (usuario in listaUsuarios) {
            sb.append("${usuario.id}: ${usuario.nombre} ${usuario.email}\n")
        }
        binding.tvContenidoTabla.text = sb.toString()
    }

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

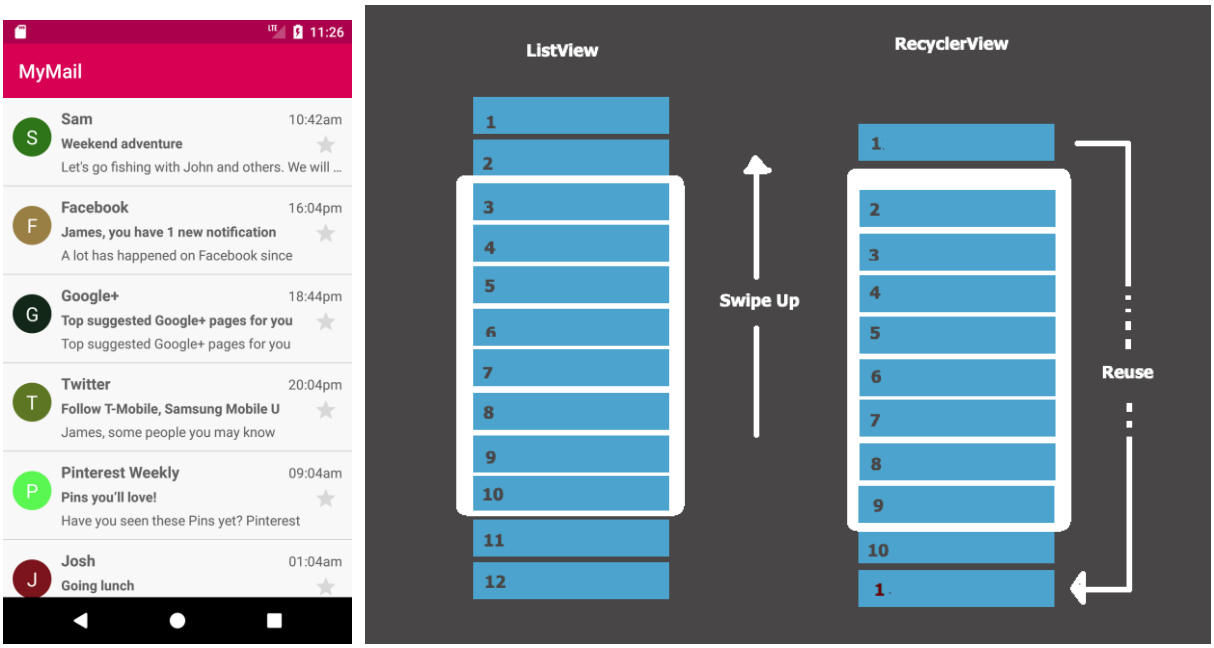
DEMO 5.3. RECYCLERVIEW

Vamos a modificar la demo anterior para que, en lugar de mostrar una cadena (StringBuilder) concatenando los diferentes campos y usuarios, muestre una lista utilizando un elemento que se denomina `RecyclerView`.

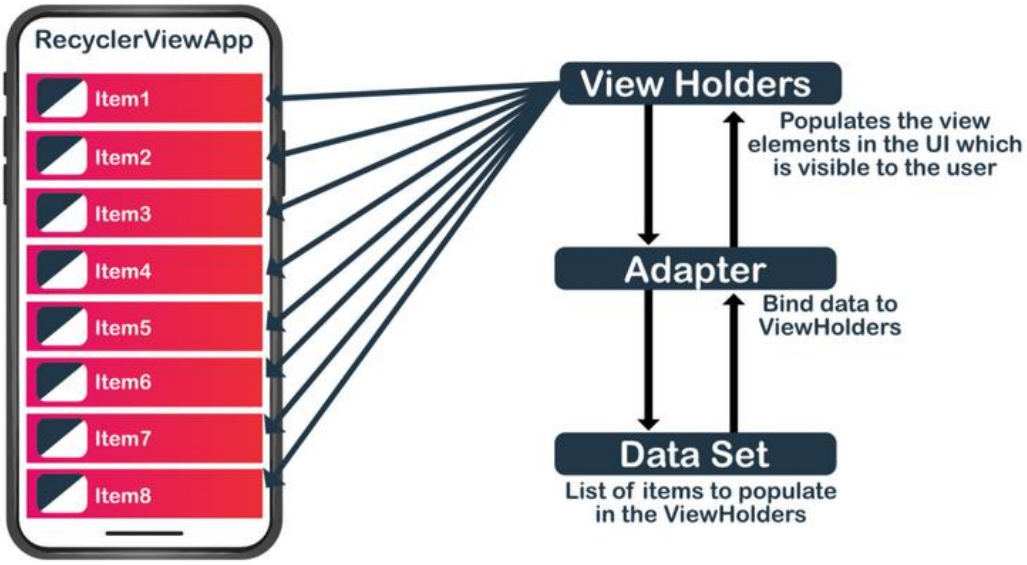
Introducción

El widget `RecyclerView` de Android es una versión más avanzada y flexible de un control más antiguo denominado ListView.

Se utiliza para mostrar una lista desplazable (*scrollable*) de elementos.



Para crear un RecyclerView:



De forma genérica, los pasos a seguir son los siguientes:

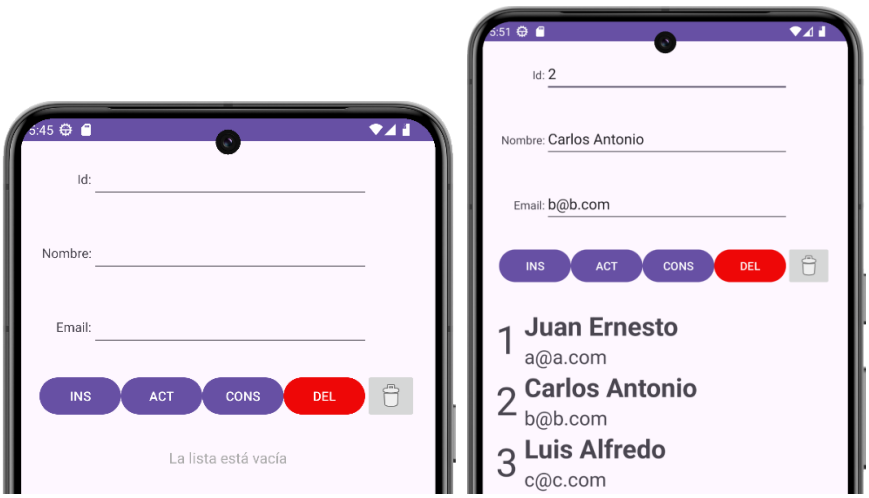
1. Agregamos un RecyclerView en el layout XML:

```
<androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
```

2. Creamos un modelo de datos (Usuario, Producto, Empleado, Coche,... lo que sea) para almacenar la información que se mostrará en el RecyclerView.
3. Creamos una clase `ViewHolder` para describir cómo se verá cada elemento de datos en el `RecyclerView`.
4. Creamos un **adaptador** para enlazar los datos del modelo con el ViewHolder y para manejar la creación y asociación de elementos en el RecyclerView.
5. Configuramos el RecyclerView en nuestra Activity o Fragment:

```
recyclerview.layoutManager = LinearLayoutManager(this)
recyclerview.adapter = YourAdapter(data)
```

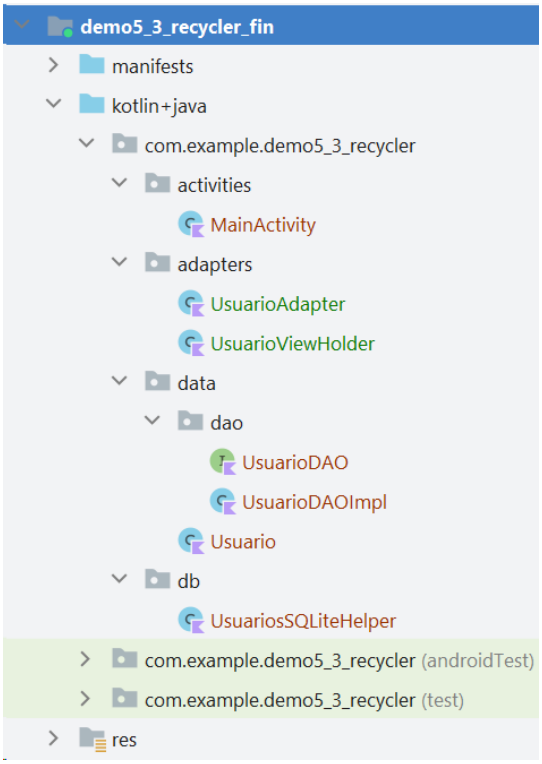
Objetivo final



Desarrollo

Estructura del proyecto

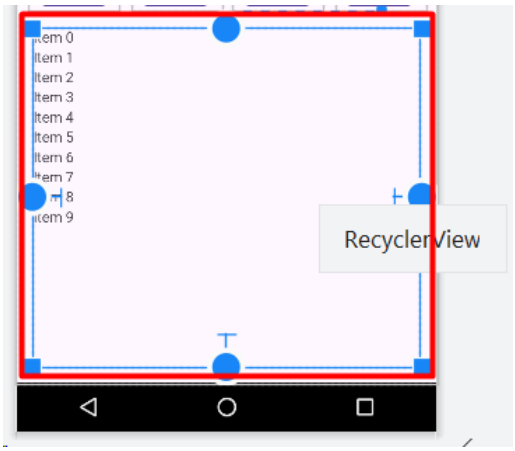
Creamos un nuevo paquete 'adapters' donde almacenaremos las clases `UsuarioAdapter` y `UsuarioViewHolder`, de momento sin código:



Añadir widget RecyclerView al archivo activity_main.xml

Sustituimos el ScrollView y el TextView anterior en el que mostrábamos el contenido de la tabla usuarios por un elemento RecyclerView:

```
    <!-- El resto del archivo activity_main.xml se omite por brevedad -->
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/rvUsuarios"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="16dp" />

</LinearLayout>
```



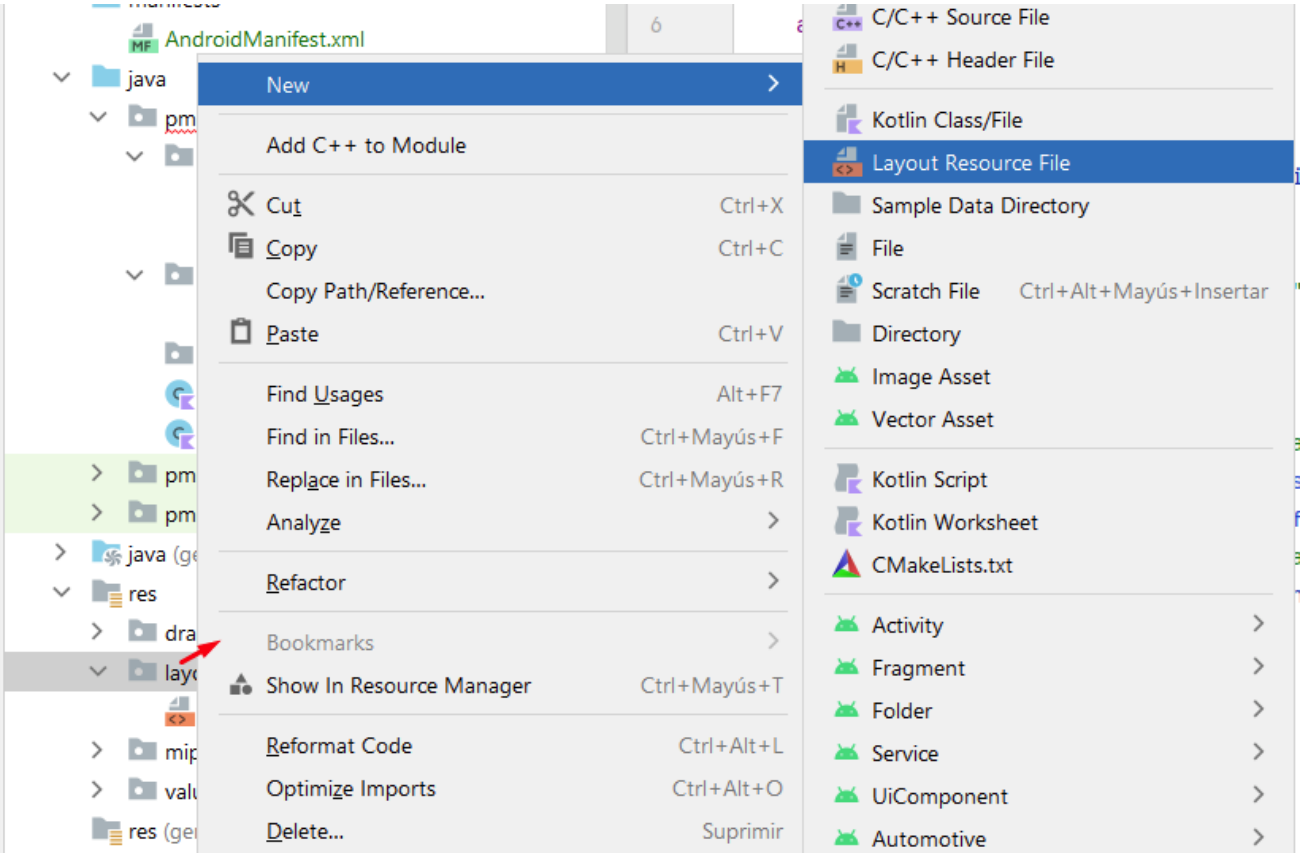
Clase Usuario (paquete data)

Esta clase, que ya creamos en la demo anterior, representa lo que queremos mostrar en la lista (RecyclerView). Sin cambios con respecto a la demo anterior:

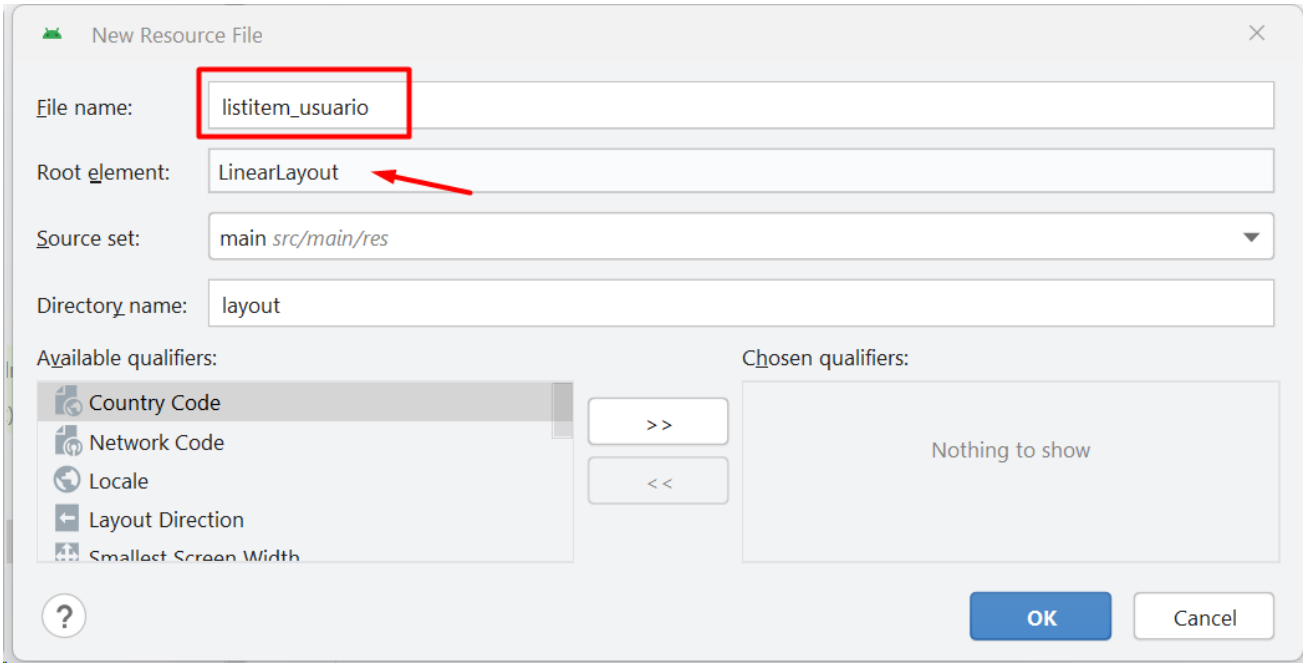
```
package com.example.demo5_3_recycler.data
data class Usuario(
    var id: Int? = null,  // Dejarlo como null permite que SQLite lo genere
                          // automáticamente (autoincrement)
    var nombre: String,
    var email: String
)
```

Creamos layout con el formato que queremos para cada Usuario: listitem_usuario.xml

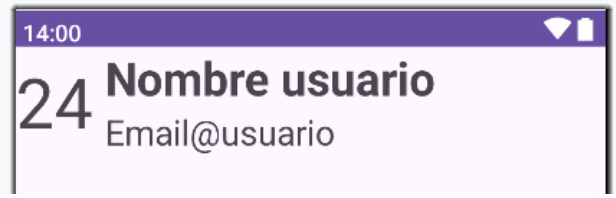
Click botón derecho en la carpeta res/layout, New > Layout Resource File:



Lo llamamos `listitem_usuario.xml` y elegimos como root un LinearLayout en lugar del tradicional `ConstraintLayout` (aunque podría ser cualquiera):



En cada elemento de la lista queremos mostrar el id, nombre y email del usuario. Sería algo así:




```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal">
    <TextView
        android:id="@+id/tvId"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        tools:text="24"
        android:layout_gravity="center"
        android:textSize="48sp"/>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingStart="8dp"
        android:paddingEnd="8dp"
        android:orientation="vertical">
        <TextView
            android:id="@+id/tvNombre"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="32sp"
```

```
            android:textStyle="bold"
            tools:text="Nombre usuario" />
        <TextView
            android:id="@+id/tvEmail"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="24sp"
            tools:text="Email@usuario" />
    </LinearLayout>
</LinearLayout>
```

Código del ViewHolder: UsuarioViewHolder.kt

Comenzamos a rellenar la clase que representa el contenedor de cada usuario: la clase *UsuarioViewHolder.kt*. En primer lugar, esta clase debe heredar de `RecyclerView.ViewHolder`.

Añadimos también por comodidad un método `bindUsuario()` que se encargará de "pintar" los diferentes campos que corresponden a cada elemento de la lista (cada view holder).

```
package com.example.demo5_3_recycler.adapters

import android.view.View
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
import com.example.demo5_3_recycler.data.Usuario
import com.example.demo5_3_recycler.databinding.ListitemUsuarioBinding

/**
* Esta clase nos ayuda a rellenar cada elemento de la lista (ViewHolder de RecyclerView)
* Hacemos que herede de RecyclerView.ViewHolder y le pasamos la vista (itemView)
*/
class UsuarioViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    // Referencias UI
    private val idTextView: TextView
    private val nombreTextView: TextView
    private val emailTextView: TextView

    // Asociación ViewHolder (UsuarioViewHolder) - Layout (listitem_usuario.xml)
    private val binding = ListitemUsuarioBinding.bind(itemView)

    init {
        idTextView = binding.tvId
        nombreTextView = binding.tvNombre
        emailTextView = binding.tvEmail
    }

    /**
     * Función auxiliar que nos va a ayudar a rellenar los campos id, nombre y email
     * de cada fila de la lista de Usuarios
     */
    fun bindUsuario(usuario: Usuario) {
        idTextView.text = usuario.id.toString()
```

```
        nombreTextView.text = usuario.nombre
        emailTextView.text = usuario.email
    }
}
```

Código del Adapter: UsuarioAdapter.kt

A continuación, añadiremos el código de nuestro Adapter ya creado al inicio (*UsuarioAdapter.kt*), que deberá heredar de `RecyclerView.Adapter`, y sobrescribir los siguientes métodos:

- `onCreateViewHolder()`. Se utiliza para crear una nueva vista para cada elemento en la lista. Este método se invoca solo una vez por elemento y se utiliza para inicializar la vista y devolver un objeto ViewHolder que representa la vista creada.
- `onBindViewHolder()`. Su objetivo es mostrar en la pantalla los datos correctos en la vista correcta. Asocia los datos de la fuente de datos (en nuestro caso, SQLite) con una vista (ítem de la lista). ASOCIA LOS DATOS CON LA VISTA (ÍTEM).
- `onItemCount()`. Indica el número de elementos de la colección de datos.

```
package com.example.demo5_3_recycler.adapters

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView
import com.example.demo5_3_recycler.R
import com.example.demo5_3_recycler.data.Usuario

// Recibe como parámetro una lista de objetos Usuario y extiende de RecyclerView.Adapter
class UsuarioAdapter(private val listaUsuarios: List<Usuario>) :
RecyclerView.Adapter<UsuarioViewHolder>() {

    /**
     * Inflamos (construimos) una vista a partir del layout listitem_usuario.xml
     * y creamos y devolvemos un nuevo ViewHolder pasándole dicha vista como parámetro.
     */
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): UsuarioViewHolder {
        // Inflamos la vista con el layout listitem_usuario.xml
        val itemView = LayoutInflater.from(parent.context).inflate(
            R.layout.listitem_usuario,    // id del layout a inflar (el layout de cada
item)
            parent,
            false)
        // Creamos un nuevo ViewHolder pasándole la vista anteriormente creada
        return UsuarioViewHolder(itemView)
    }

    /**
     * Recuperamos el objeto Usuario correspondiente a la posición recibida como parámetro
     * y asignamos sus datos sobre el ViewHolder también recibido como parámetro
     */
    override fun onBindViewHolder(holder: UsuarioViewHolder, position: Int) {
        val usuario = listaUsuarios[position]
```

```
        holder.bindUsuario(usuario)
    }

    override fun getItemCount(): Int {
        return listaUsuarios.size
    }
}
```

Con esto tendríamos terminado el Adapter y el ViewHolder.

TextView tabla vacía

```
<TextView
    android:id="@+id/tvMensajeListaVacia"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="20sp"
    android:gravity="center"
    android:layout_gravity="center"
    android:textStyle="bold"
    android:layout_marginTop="20dp"
    android:visibility="gone"
    android:text="La lista está vacía" />
```

En la función mostrarUsuarios de MainActivity.kt añadiremos el código necesario para hacer visible este TextView, en el caso de que la tabla esté vacía, o el RecyclerView, en el caso de que la tabla tenga algún usuario.

MainActivity.kt

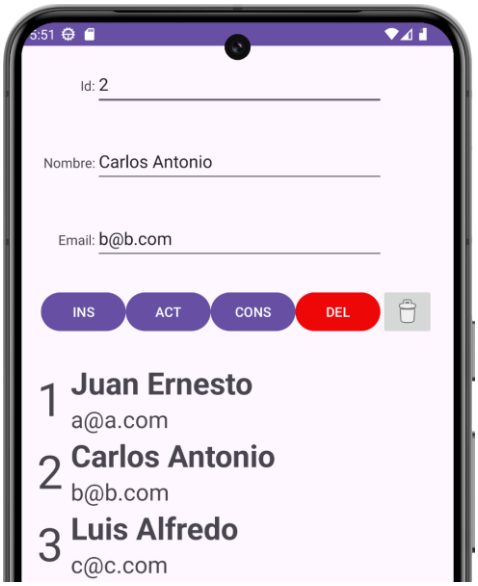
Declaramos los elementos necesarios para trabajar con el RecyclerView:

```
// ------- Elementos necesarios para el RecyclerView -------
private lateinit var listaUsuarios: List<Usuario>
private lateinit var adapter: UsuarioAdapter
private val manager = LinearLayoutManager(this)
// LayoutManager de tipo LinearLayoutManager. Muestra los elementos en una lista vertical


    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        operaciones = UsuarioDAOImpl(binding.root, dbHelper)

        mostrarUsuarios()

        binding.btnInsertar.setOnClickListener {
```

```
            //TODO: Comprobar que los campos no estén vacíos
            val usuario = Usuario(
                nombre = binding.etName.text.toString(),
                email = binding.etEmail.text.toString()
            )
            operaciones.insertarUsuario(usuario)
            // Resetear editText
            resetearEditText()
        }

        binding.btnActualizar.setOnClickListener {
            //TODO: Comprobar que los campos no estén vacíos
            val usuario = Usuario(
                binding.etId.text.toString().toInt(),
                binding.etName.text.toString(),
                binding.etEmail.text.toString()
            )
            operaciones.actualizarUsuario(usuario)
            // Resetear editText
            resetearEditText()
        }

        binding.btnEliminar.setOnClickListener {
            val campoId = binding.etId.text.toString().toInt()
            operaciones.borrarUsuario(campoId)
            resetearEditText()
        }

        binding.btnConsultar.setOnClickListener {
            val campoId = binding.etId.text.toString()
            if (campoId.isEmpty()) {
                resetearEditText()
                mostrarUsuarios()
            } else {
                val usuario = operaciones.leerUsuarioPorId(campoId.toInt())
                // TODO: si es null, mostrar Snackbar
                binding.etName.setText(usuario.nombre)
                binding.etEmail.setText(usuario.email)
            }
        }

        binding.btnVaciar.setOnClickListener {
            operaciones.borrarTodosLosUsuarios()
        }
    } // Fin onCreate

    private fun resetearEditText() {
        binding.etId.setText("")
        binding.etName.setText("")
        binding.etEmail.setText("")
    }

    // Adaptado a RecyclerView
    private fun mostrarUsuarios() {
```

```
        val listaUsuarios = operaciones.leerUsuarios()
        if (listaUsuarios.isEmpty()) {
            binding.tvMensajeListaVacia.visibility = View.VISIBLE
            //binding.tvMensajeListaVacia.text = ""

            binding.rvUsuarios.visibility = View.GONE
        } else {
            binding.tvMensajeListaVacia.visibility = View.GONE
            binding.rvUsuarios.visibility = View.VISIBLE

            binding.rvUsuarios.layoutManager = manager
            binding.rvUsuarios.adapter = UsuarioAdapter(listaUsuarios)
        }
    }


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

Ya sería posible ejecutar la aplicación para ver cómo quedan nuestros datos en pantalla. Ejecutamos, insertamos algunos registros y probamos el botón de consulta (CONS):

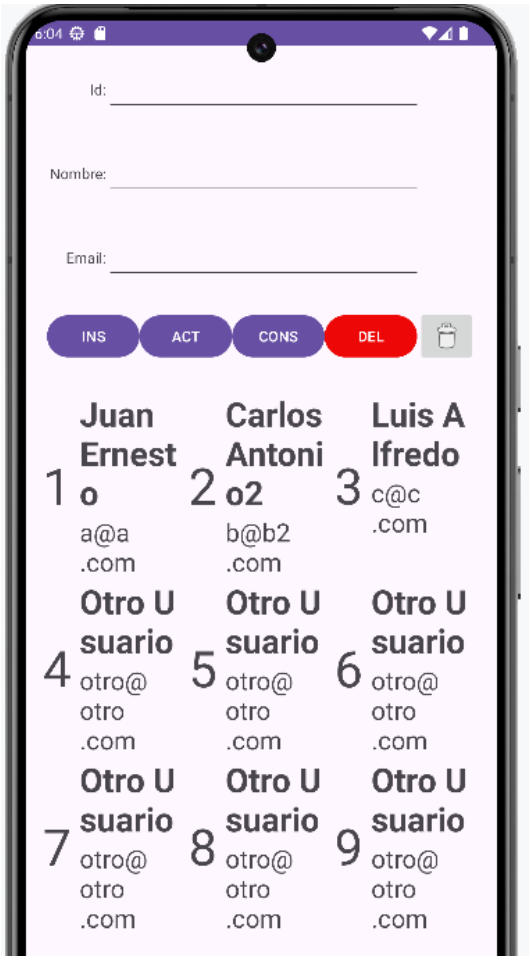


Cambio de LayoutManager

Un cambio sencillo que podríamos realizar sería modificar el LayoutManager que hemos utilizado (LinearLayoutManager) por uno en formato grid (rejilla). Por ejemplo, especificando 3 columnas (fichero MainActivity.kt):

```
24      // ------- Elementos necesarios para el RecyclerView -------
25      private lateinit var listaUsuarios: List<Usuario>
26      private lateinit var adapter: UsuarioAdapter
27      //private val manager = LinearLayoutManager(this)
28      private val manager = GridLayoutManager( context: this,  spanCount: 3) // 3 columnas
```

Resultado:



Añadir un divisor entre los elementos del RecyclerView (ItemDecoration)

Los `ItemDecoration` nos servirán para personalizar el aspecto de un `RecyclerView`. Un ejemplo típico de uso son los **separadores** o **divisores** de una lista.

Crear un `ItemDecoration` personalizado tiene su complejidad. No es algo inmediato, por lo que sólo vamos a ver un pequeño ejemplo de uso de los separadores que nos proporciona Android, disponibles mediante la clase `DividerItemDecoration`.

Para utilizar este componente debemos simplemente crear el objeto y asociarlo a nuestro RecyclerView mediante `addItemDecoration`() en nuestra actividad principal.

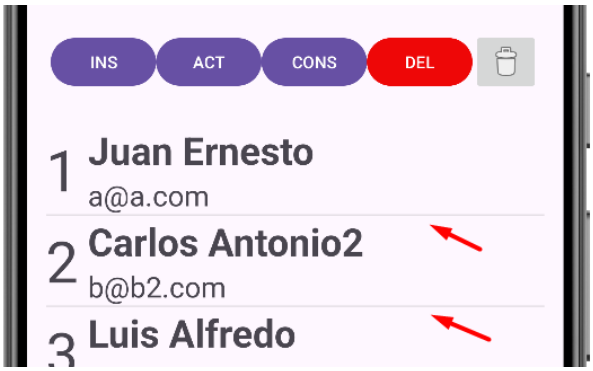
Volvemos a colocar como LayoutManaget un LinearLayoutManager en MainActivity.kt:

```
24      // ------- Elementos necesarios para el RecyclerView -------
25      private lateinit var listaUsuarios: List<Usuario>
26      private lateinit var adapter: UsuarioAdapter
27      private val manager = LinearLayoutManager( context: this)
28      //private val manager = GridLayoutManager(this, 3) // 3 columnas
```

Modificamos el método initRecyclerView():

```
// Añadimos línea divisoria entre elementos del RecyclerView
binding.rvUsuarios.addItemDecoration(DividerItemDecoration(
        context: this,
        DividerItemDecoration.VERTICAL))
```

Ejecutamos:



FIN DEMO 5.3