SQLITE con MVVM

1. Crea un proyecto: File→New Project → Empty Views Activity , lo llamaremos, por ejemplo, **SQLiteMVVM**.

Construye una app que implemente las operaciones CRUD (crear, leer, actualizar y eliminar) almacenados de forma persistente en una base de datos SQLite usando el patrón MVVM y la librería de persistencia **room**. La funcionalidad de la app será la siguiente:

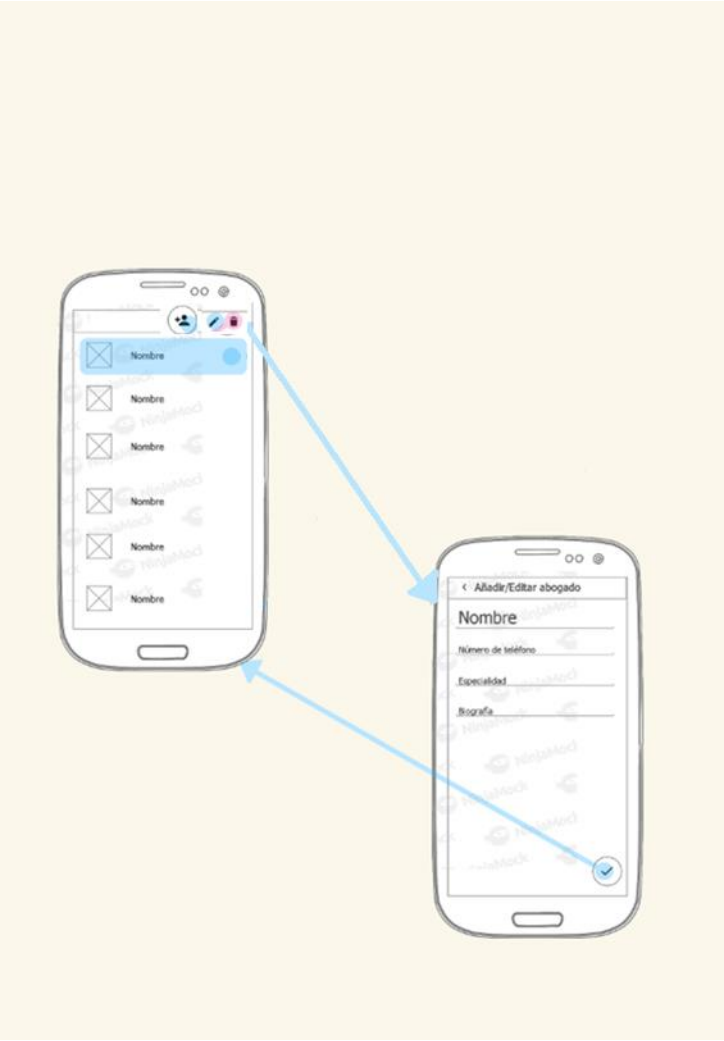
La página principal, tendrá un RecyclerView con la lista de los abogados existentes en la base de datos.

Un abogado tendrá un campo que lo identifica (id_abogado), un nombre, teléfono, especialidad y biografía.

Cada elemento de la lista mostrará el nombre del abogado

Además, tendrá una barra de tareas, con las opciones visibles como iconos de: añadir (se abre la pantalla que permite añadir un abogado), detalle (se muestran los datos del abogado seleccionado), editar (permite editar los datos del abogado seleccionado) y eliminar (elimina el abogado seleccionado).

La pantalla de Alta, Detalle y Modificación deben usar el mismo Fragment. Al usar esta pantalla, usa un parámetro que funcione como modo de funcionamiento, que te indique si se abre para alta, detalle o modificación y en función del modo funcione de una manera u otra.



2. Vamos a usar la libreria **Room Persistence** para SQLITE, la cual actúa como una capa de abstracción sobre SQLite, lo que permite un acceso fluido a la base de datos y al mismo tiempo aprovecha toda la potencia de SQLite.

Puedes encontrar más información sobre el uso de la librería Room Persistence en https://developer.android.com/training/data-storage/room?hl=es-419

Para usar Room en tu app, agrega el siguiente plugin y dependencias al archivo **build.gradle** de la app:

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    id("org.jetbrains.kotlin.kapt")
}
```

```
implementation(libs.androidx.room.common)
implementation(libs.androidx.room.ktx)
implementation("androidx.room:room-runtime:2.6.1")
kapt("androidx.room:room-compiler:2.6.1")
implementation("androidx.fragment:fragment-ktx:1.8.5")
```

3. Crear la clase que representa la tabla de la base de datos de tu app. Mas información sobre como definir entidades en https://developer.android.com/training/data-storage/room/defining-data?hl=es-419

```
package model

import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "abogado")
data class Abogado(
    @PrimaryKey(autoGenerate = true) @ColumnInfo(name = "id_abogado") var
idAbogado: Int = 0,
    @ColumnInfo(name = "nombre") var nombre:String,
    @ColumnInfo(name = "telefono") var telefono:String,
    @ColumnInfo(name = "especialidad") var especialidad:String,
    @ColumnInfo(name = "biografia") var biografia: String): java.io.Serializable
```

4. Igualmente, por cada tabla de base de datos, debemos crear una interfaz para definir las operaciones de base de datos usando el patrón DAO (información en: https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419)

```
package conexion

import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update
import model.Abogado

@Dao
interface AbogadoDAO {
    @Insert
    suspend fun insert(abogado: Abogado)

    @Query("SELECT * FROM abogado")
    fun getAllAbogados(): LiveData<List<Abogado>>

    @Query("SELECT * FROM abogado WHERE id_abogado = :id")
    fun getAbogadoById(id: Int): LiveData<Abogado>

    @Update
    suspend fun update(abogado: Abogado)

    @Delete
    suspend fun delete(abogado: Abogado)

}
```

Para evitar que las sentencias sobre la base de datos bloqueen la IU, Room no permite el acceso a la base de datos en el subproceso principal, por lo que **para acceder a base de datos debemos hacer llamadas asíncronas**.

LiveData es una clase de Jetpack que está diseñado para almacenar y administrar datos que pueden ser observados por componentes de la interfaz de usuario, como actividades y fragmentos, lo cual nos permite crear **consultas observables asíncronas**.

Marcar una función como **suspend** en Kotlin, hace que la función se pueda pausar y reanudar en puntos específicos, sin bloquear el hilo en el que se está ejecutando. Es un componente clave de las corrutinas de Kotlin, que **permite la programación asincróna** en un estilo más secuencial y legible.

5. Crear una clase abstracta que representa la base de datos.

En este ejemplo, **Room.databaseBuilder** crea una instancia de la clase AppDatabase, que representa su base de datos de Room. El archivo de la base de datos se llamará "abogado_database" y se almacenará en el directorio de datos de la aplicación. En tiempo de compilación se creará la clase **AppDatabase_Impl** que implementa la clase abstracta definida como AppDatabase.

Por otra parte, voy a tener acceso a través de la función **abogadoDAO** a una instancia de la clase **AbogadoDAO_Impl** que se creará automáticamente al compilar.

```
package conexion

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import model.Abogado

@Database(entities = [Abogado::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun abogadoDAO(): AbogadoDAO

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "abogado.db3"
                ).build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

Nota: La palabra clave **volatil** es un modificador que se puede aplicar a una variable. Se utiliza para garantizar que los cambios realizados en una variable por un subproceso sean inmediatamente visibles para otros subprocesos, esto evita situaciones en las que un subproceso modifica una variable, pero otro subproceso continúa viendo el valor anterior porque está almacenado en caché.

6. Crear la capa lógica:

Esta clase **AbogadoRepository** sirve como intermediario entre ViewModel y la fuente de datos (en este caso, la base de datos de Room)**:**

- **Abstracción de acceso a datos**: el repositorio oculta los detalles de implementación del acceso a datos desde ViewModel. Proporciona una interfaz de nivel superior para acceder a los datos, independientemente de si provienen de una base de datos local, una API remota u otras fuentes.
- **Operaciones de datos**: Encapsula la lógica para realizar operaciones de datos, como insertar, actualizar, eliminar y consultar datos de la base de datos de Room.
- **Transformación de datos**: Podríamos transformar datos a un formato adecuado para ViewModel y UI. Esto podría implicar combinar datos de múltiples fuentes o aplicar una lógica de negocio específica.
- **Almacenamiento en caché de datos**: se podrían implementar mecanismos de almacenamiento en caché para mejorar el rendimiento y reducir las solicitudes de red.

```
package conexion

import androidx.lifecycle.LiveData
import model.Abogado

class AbogadoRepository(private val abogadoDAO: AbogadoDAO) {
    fun getAllAbogados(): LiveData<List<Abogado>> {
        return abogadoDAO.getAllAbogados()
    }

    suspend fun insert(abogado: Abogado) {
        abogadoDAO.insert(abogado)
    }

    suspend fun update(abogado: Abogado) {
        abogadoDAO.update(abogado)
    }

    suspend fun delete(abogado: Abogado) {
        abogadoDAO.delete(abogado)
    }

    fun getAbogadoById(id: Int): LiveData<Abogado> {
        return abogadoDAO.getAbogadoById(id)
    }
}
```

Clase **AbogadoViewModel**: Actúa como intermediario entre el Modelo y la Vista, se encarga de recuperar/persistir los datos y hacer los datos accesibles a la interfaz de usuario.

```
package viewmodel

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.viewModelScope
import conexion.AbogadoRepository
import conexion.AppDatabase
import kotlinx.coroutines.launch
import model.Abogado

class AbogadoViewModel(application: Application) : AndroidViewModel(application)
{
    private val repository: AbogadoRepository
    val data: LiveData<List<Abogado>>

    init {
        val abogadoDAO =
AppDatabase.getDatabase(application.applicationContext).abogadoDAO()
        data = abogadoDAO.getAllAbogados()
        repository = AbogadoRepository(abogadoDAO)
    }

    private fun getAllAbogados(): LiveData<List<Abogado>> {
        return repository.getAllAbogados()
    }

    fun getAbogadoById(id:Int):LiveData<Abogado> {
        return repository.getAbogadoById(id)
    }

    fun insert(abogado: Abogado) = viewModelScope.launch {
        repository.insert(abogado)
    }

    fun update(abogado: Abogado) = viewModelScope.launch{
        repository.update(abogado)
    }

    fun delete(abogado: Abogado) = viewModelScope.launch{
        repository.delete(abogado)
    }
}
```

Como se indicaba anteriormente las corrutinas en Kotlin, permiten operaciones asincrónicas sin bloquear el hilo principal, manteniendo la interfaz de usuario receptiva. En nuestra clase, vemos que usamos la corrutina: viewModelScope.launch, la cual **nos permite realizar llamadas asincronzas** a las funciones repository.insert/update/delete que al estar definidas como funciones **suspend**, pueden pausarse/reanudarse, teniendo en cuenta el ciclo de vida de ViewModel, lo que permite que por ejemplo se cancelen automáticamente cuando el ViewModel ya no sea necesario, evitando posibles pérdidas de memoria y errores.

7. Crear la parte de la vista o capa de Interfaz de Usuario:

Como último punto se debe crear la parte de Interfaz de Usuario como Activity/Fragment desde donde se deberá llamar a AbogadoViewModel siempre que se quiera realizar una llamada a base de datos.