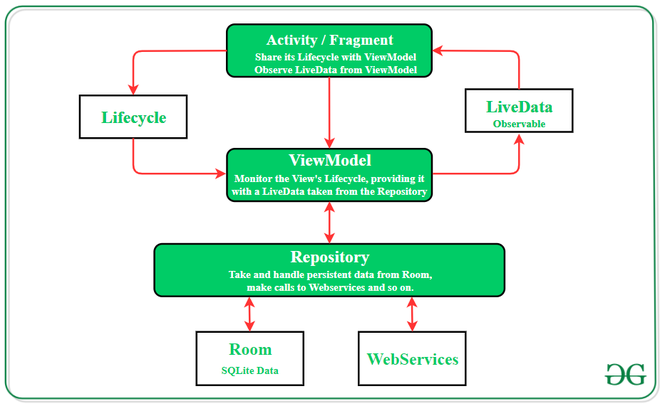
**MVVM – ROOM DB**



1. Room DB

Room Database là một thành phần của Android Jetpack, giúp quản lý cơ sở dữ liệu SQLite trong ứng dụng Android một cách dễ dàng và hiệu quả. Nó cung cấp một lớp trừu tượng trên SQLite, giúp bạn làm việc với cơ sở dữ liệu dễ dàng hơn bằng cách sử dụng các phương thức đối tượng thay vì viết mã SQL thuần túy. Dưới đây là các thành phần chính của Room Database

1. **Entity**: Là một lớp đại diện cho một bảng trong cơ sở dữ liệu. Mỗi trường trong lớp tương ứng với một cột trong bảng

package com.example.contactsmanagerapp;  
  
import androidx.room.ColumnInfo;  
import androidx.room.Entity;  
import androidx.room.PrimaryKey;  
  
@Entity(tableName = "contacts\_table")-> Thông báo đây là Entity  
public class Contacts {  
 @ColumnInfo(name = "contact\_id") -> Tên của trường trong BD nếu k điền sẽ để là tên biến  
 @PrimaryKey(autoGenerate = true) -> xác định khóa chính  
 private int id;  
 @ColumnInfo(name = "contact\_name")  
 private String name;  
 @ColumnInfo(name = "contact\_email")  
 private String email;  
  
 public Contacts(int id, String name, String email) {  
 this.id = id;  
 this.name = name;  
 this.email = email;  
 }  
  
 public Contacts() {  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public void setEmail(String email) {  
 this.email = email;  
 }  
}

1. **DAO (Data Access Object):** Là một giao diện hoặc lớp cung cấp các phương thức để tương tác với dữ liệu trong cơ sở dữ liệu. Các phương thức này có thể bao gồm các thao tác như thêm, sửa, xóa, và truy vấn dữ liệu (Nói cách khác ở đây tạo các phương thức insert delete hoặc query cho database thành các function mà ta sẽ gọi trong code)

package com.example.contactsmanagerapp;  
  
import androidx.room.Dao;  
import androidx.room.Delete;  
import androidx.room.Insert;  
import androidx.room.Query;  
  
import java.util.List;  
  
@Dao  
public interface ContactDAO {  
  
 @Insert  
 void insert(Contacts contact);  
 @Delete  
 void delete(Contacts contact);  
 @Query("SELECT \* FROM contacts\_table")  
 LiveData<List<Contacts>> getAllContact();

}

Các từ khóa @Insert @Delete là mặc định hoặc ta có thể tự custom Query theo ý muốn

1. **Database**: Là một lớp trừu tượng mở rộng từ RoomDatabase, và chứa cơ sở dữ liệu của ứng dụng. Nó cung cấp các phương thức để truy cập DAO.

Quản lý kết nối cơ sở dữ liệu: Lớp Database mở rộng từ RoomDatabase và chịu trách nhiệm thiết lập và duy trì kết nối với cơ sở dữ liệu SQLite. Nó đảm bảo rằng chỉ có một kết nối duy nhất được sử dụng để tương tác với cơ sở dữ liệu, giúp tránh các vấn đề liên quan đến việc truy cập đồng thời từ nhiều luồng.

package com.example.contactsmanagerapp;  
  
import android.content.Context;  
  
import androidx.room.Database;  
import androidx.room.Room;  
import androidx.room.RoomDatabase;  
  
@Database(entities = {Contacts.class},version = 1)  
public abstract class ContactsDatabase extends RoomDatabase {  
 public abstract ContactDAO getContactDAO();  
  
 private static volatile ContactsDatabase *db\_instance*;  
 public static synchronized ContactsDatabase getInstance(Context context){  
 if(*db\_instance* == null)  
 {  
 *db\_instance* = Room.*databaseBuilder*(context.getApplicationContext(),ContactsDatabase.class,"contact\_db")  
 .fallbackToDestructiveMigration()  
 .build();  
 }  
 return *db\_instance*;  
  
 }  
}

Volatile Instance: Sử dụng từ khóa volatile để đảm bảo rằng các thay đổi đối với biến INSTANCE được nhìn thấy ngay lập tức bởi tất cả các luồng. java

Sử dụng khối synchronized để đảm bảo rằng chỉ một luồng có thể truy cập đoạn mã tạo instance tại một thời điểm.(Signle ton pattern)

Cách tiếp cận này đảm bảo rằng việc tạo instance của AppDatabase là thread-safe và chỉ một instance duy nhất của cơ sở dữ liệu được tạo ra trong suốt vòng đời của ứng dụng. Điều này rất quan trọng để tránh các vấn đề liên quan đến việc truy cập đồng thời vào cơ sở dữ liệu từ nhiều luồng khác nhau.

A diagram of a software application

Description automatically generated

1. Để kết hợp **Repository** vào cấu trúc của Room Database, bạn cần tạo một lớp Repository chịu trách nhiệm quản lý dữ liệu từ nhiều nguồn khác nhau (như cơ sở dữ liệu, mạng, hoặc bộ nhớ cache). Repository đóng vai trò như một nguồn duy nhất để lấy dữ liệu và cung cấp một API rõ ràng cho phần còn lại của ứng dụng.

package com.example.contactsmanagerapp;  
  
import android.app.Application;  
import android.os.Handler;  
import android.os.Looper;  
  
import androidx.lifecycle.LiveData;  
  
import java.util.List;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class Repository {  
//Method in DAO is execute through Repository  
 private final ContactDAO contactDAO;  
 ExecutorService executor;  
 Handler handler;  
  
 public Repository(Application application) {  
  
 ContactsDatabase contactsDatabase = ContactsDatabase.*getInstance*(application);  
 this.contactDAO = contactsDatabase.getContactDAO();  
 executor = Executors.*newSingleThreadExecutor*();  
 handler = new Handler(Looper.*getMainLooper*());  
 }  
  
 public void insertContact(Contacts contact)  
 {  
  
 executor.execute(new Runnable() {  
 @Override  
 public void run() {  
 contactDAO.insert(contact);  
 }  
 });  
  
 }  
 public void deleteContact(Contacts contact)  
 {  
 executor.execute(new Runnable() {  
 @Override  
 public void run() {  
 contactDAO.delete(contact);  
 }  
 });  
  
 }  
 public LiveData<List<Contacts>> getAllContact()  
 {  
 return contactDAO.getAllContact();  
  
 }  
}

Để đảm bảo rằng các thao tác cập nhật cơ sở dữ liệu được thực hiện trên luồng nền và không ảnh hưởng đến giao diện người dùng (UI), bạn có thể sử dụng các Executor hoặc Coroutine trong Kotlin. Dưới đây là cách tiếp cận sử dụng Executor trong Java để thực hiện các thao tác cập nhật ở background

**Executor Service**: Sử dụng Executors.newFixedThreadPool(NUMBER\_OF\_THREADS) để tạo một nhóm luồng cố định cho các tác vụ nền.

**Repository**: Thực hiện các thao tác cơ sở dữ liệu như thêm, sửa, xóa trong databaseWriteExecutor để đảm bảo chúng chạy trên luồng nền.

1. **LiveData** là một thành phần của Android Jetpack, được thiết kế để giữ dữ liệu và cho phép giao diện người dùng (UI) quan sát các thay đổi của dữ liệu này. Khi dữ liệu thay đổi, LiveData sẽ tự động cập nhật UI. Việc kết hợp LiveData với Room Database và Repository giúp quản lý và cập nhật dữ liệu một cách hiệu quả và an toàn trên các luồng khác nhau
2. **ViewModel**

package com.mastercoding.contactsmanagerapp;  
  
  
import android.app.Application;  
  
import androidx.annotation.NonNull;  
import androidx.lifecycle.AndroidViewModel;  
import androidx.lifecycle.LiveData;  
import androidx.lifecycle.ViewModel;  
  
import java.util.List;  
  
public class MyViewModel extends AndroidViewModel {  
  
 // If you need to use context inside your Viewmodel  
 // you should use AndroidViewModel (AVM),  
 // because it contains the application context.  
  
 // Repository  
 private Repository myRepository;  
  
 // LiveData  
 private LiveData<List<Contacts>> allContacts;  
  
 public MyViewModel(@NonNull Application application) {  
 super(application);  
 this.myRepository = new Repository(application);  
 }  
  
 public LiveData<List<Contacts>> getAllContacts(){  
 allContacts = myRepository.getAllContacts();  
 return allContacts;  
 }  
  
  
 public void addNewContact(Contacts contact){  
 myRepository.addContact(contact);  
 }  
  
 public void deleteContact(Contacts contact){  
 myRepository.deleteContact(contact);  
 }  
  
  
  
  
 // AndroidViewModel class is a subclass of ViewModel  
 // and similar to them, they are designed to store and  
 // manage UI-related data are responsible to  
 // prepare & provide data for UI and automatically  
 // allow data to survive configuration change.  
}

**Repository và ViewModel** là hai thành phần quan trọng trong kiến trúc MVVM (Model-View-ViewModel) của Android, mỗi thành phần có vai trò và trách nhiệm riêng biệt, giúp duy trì nguyên tắc phân tách trách nhiệm (Separation of Concerns). Dưới đây là lý do vì sao chúng không nên được gộp lại: 1. Phân tách trách nhiệm

**Repository**: Chịu trách nhiệm quản lý nguồn dữ liệu. Nó là một lớp trừu tượng giữa các nguồn dữ liệu khác nhau (cơ sở dữ liệu cục bộ, mạng, cache, v.v.) và ViewModel. Repository cung cấp một API gọn gàng để truy xuất và quản lý dữ liệu, đồng thời đảm bảo rằng ViewModel không phải biết về chi tiết triển khai của các nguồn dữ liệu.

**ViewModel**: Chịu trách nhiệm chuẩn bị và quản lý dữ liệu cho UI. ViewModel giữ dữ liệu trong suốt vòng đời của các Activity và Fragment, và tương tác với Repository để lấy dữ liệu khi cần. ViewModel không nên biết về chi tiết cách dữ liệu được lưu trữ hoặc truy xuất, chỉ cần biết cách sử dụng Repository để lấy dữ liệu.

**Đơn giản hóa kiểm thử Repository**: Dễ dàng kiểm thử với các trường hợp kiểm thử khác nhau liên quan đến việc truy xuất và lưu trữ dữ liệu. Bạn có thể kiểm thử Repository mà không cần phải quan tâm đến UI hoặc vòng đời của Activity/Fragment. ViewModel: Dễ dàng kiểm thử logic nghiệp vụ và cách dữ liệu được chuẩn bị cho UI. Bạn có thể kiểm thử ViewModel bằng cách mock Repository và kiểm tra cách ViewModel xử lý dữ liệu và tương tác với Repository. **Tái sử dụng mã nguồn Repository**: Có thể tái sử dụng với nhiều ViewModel khác nhau hoặc thậm chí trong nhiều ứng dụng khác nhau. Repository cung cấp một lớp trừu tượng để truy xuất dữ liệu, cho phép tái sử dụng logic truy xuất dữ liệu mà không cần thay đổi các ViewModel. ViewModel: Được thiết kế cho một nhiệm vụ cụ thể liên quan đến UI. Mỗi ViewModel tương ứng với một phần cụ thể của UI và sử dụng Repository để lấy dữ liệu cần thiết.

**Quản lý vòng đời** Repository: Không cần biết về vòng đời của Activity/Fragment. Nó chỉ cần biết cách truy xuất và quản lý dữ liệu. ViewModel: Nhận thức về vòng đời của Activity/Fragment và giúp quản lý dữ liệu trong suốt vòng đời của chúng. ViewModel sống lâu hơn Activity/Fragment, đảm bảo dữ liệu không bị mất khi cấu hình thay đổi (như xoay màn hình).

Tạo GridView

<?xml version="1.0" encoding="utf-8"?>  
  
<layout  
 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">  
   
   
 <data>  
 <variable  
 name="clickHandler"  
 type="com.mastercoding.contactsmanagerapp.MainActivityClickHandlers" />  
   
 </data>  
  
  
 <androidx.constraintlayout.widget.ConstraintLayout  
 android:background="@drawable/back2"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 tools:context=".MainActivity">  
  
  
 <androidx.recyclerview.widget.RecyclerView  
 android:id="@+id/recyclerview"  
 android:layout\_width="0dp"  
 android:layout\_height="0dp"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintEnd\_toEndOf="parent"  
 app:layout\_constraintStart\_toStartOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent" />  
  
 <com.google.android.material.floatingactionbutton.FloatingActionButton  
 android:src="@drawable/baseline\_add\_24"  
 android:id="@+id/floatingActionButton"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginEnd="16dp"  
 android:layout\_marginBottom="16dp"  
 android:onClick="@{clickHandler::onFABClicked}"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintEnd\_toEndOf="parent" />  
  
  
 </androidx.constraintlayout.widget.ConstraintLayout>  
  
</layout>

Tạo layout\_Item

<?xml version="1.0" encoding="utf-8"?>  
  
<layout xmlns:tools="http://schemas.android.com/tools"  
 xmlns:app="http://schemas.android.com/apk/res-auto"  
 xmlns:android="http://schemas.android.com/apk/res/android">  
  
 <data>  
 <variable  
 name="contact"  
 type="com.mastercoding.contactsmanagerapp.Contacts" />  
  
 </data>  
  
 <androidx.cardview.widget.CardView  
 android:elevation="20dp"  
 app:cardCornerRadius="20dp"  
 android:layout\_marginLeft="20dp"  
 android:layout\_marginRight="20dp"  
 android:layout\_marginTop="5dp"  
 android:layout\_marginBottom="5dp"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content">  
  
  
  
 <androidx.constraintlayout.widget.ConstraintLayout  
 android:layout\_margin="16dp"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content">  
  
  
 <TextView  
 android:id="@+id/textViewName"  
 android:layout\_width="0dp"  
 android:layout\_height="wrap\_content"  
 android:text="@{contact.name}"  
 android:textSize="32sp"  
 app:layout\_constraintEnd\_toEndOf="parent"  
 app:layout\_constraintStart\_toStartOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent">  
  
 </TextView>  
  
 <TextView  
 android:id="@+id/textView2"  
 android:layout\_width="0dp"  
 android:layout\_height="wrap\_content"  
 android:text="@{contact.email}"  
 android:textSize="24sp"  
 app:layout\_constraintEnd\_toEndOf="parent"  
 app:layout\_constraintStart\_toStartOf="parent"  
 app:layout\_constraintTop\_toBottomOf="@+id/textViewName" />  
  
  
 </androidx.constraintlayout.widget.ConstraintLayout>  
  
  
  
  
 </androidx.cardview.widget.CardView>  
  
  
</layout>