**Section 29 – MVVM + Room DB – Contacts Manager App**

**Key Concepts Learned**

1. **Room Database**
   * ORM library for SQLite in Android.
   * Provides compile-time SQL verification.
   * Reduces boilerplate for object mapping.
   * Integrates with **LiveData** for reactive updates.
2. **Room Components**
   * **Entity** → Defines DB tables.
   * **DAO (Data Access Object)** → Defines SQL operations (Insert, Delete, Query, Update).
   * **Database** → Main DB holder class (extends RoomDatabase).
3. **MVVM Architecture**
   * **Model** → Data layer (Room, Repository).
   * **ViewModel** → Holds UI data & mediates between Model and View.
   * **View** → UI layer observing LiveData.
4. **Repository**
   * Abstracts data sources from ViewModel.
   * Manages threading for DB operations.
   * Uses ExecutorService + Handler in Java for background tasks.
5. **Data Binding**
   * Connects XML layouts directly to data objects.
   * Supports **two-way binding** (@={}) for EditText.
6. **RecyclerView with Data Binding**
   * Binding item layout directly to data objects in Adapter.
   * No need to manually bind TextViews.
7. **LiveData**
   * Reactive data container.
   * Auto-updates UI when data changes.
8. **Navigation & Click Handlers**
   * Using separate click handler classes.
   * Passing Context or ViewModel to handlers.
9. **Swipe-to-Delete**
   * Implemented via ItemTouchHelper.SimpleCallback.

**Step-by-Step Implementation Guide**

**1. Add Dependencies**

// Room Database

implementation "androidx.room:room-runtime:2.x.x"

annotationProcessor "androidx.room:room-compiler:2.x.x" // For Java

// LiveData & ViewModel

implementation "androidx.lifecycle:lifecycle-livedata:2.x.x"

implementation "androidx.lifecycle:lifecycle-viewmodel:2.x.x"

// Data Binding

android {

buildFeatures {

dataBinding true

}

}

**2. Create Entity**

@Entity(tableName = "contacts\_table")

public class Contacts {

@PrimaryKey(autoGenerate = true)

@ColumnInfo(name = "contact\_id") // Optional: customize column name

private int id;

@ColumnInfo(name = "contact\_name")

private String name;

@ColumnInfo(name = "contact\_email")

private String email;

// Constructor (no ID - auto-generated)

public Contacts(String name, String email) {

this.name = name;

this.email = email;

}

// Getters and Setters

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; }

}

**3. Create DAO**

@Dao

public interface ContactDao {

@Insert

void insert(Contacts contact);

@Delete

void delete(Contacts contact);

@Query("SELECT \* FROM contacts\_table")

LiveData<List<Contacts>> getAllContacts();

}

**4. Create Database Class (Singleton)**

@Database(entities = {Contacts.class}, version = 1)

public abstract class ContactDatabase extends RoomDatabase {

private static ContactDatabase instance;

public abstract ContactDao getContactDao();

// Singleton pattern

public static synchronized ContactDatabase getInstance(Context context) {

if (instance == null) {

instance = Room.databaseBuilder(context.getApplicationContext(),

ContactDatabase.class, "contacts\_DB")

.fallbackToDestructiveMigration()

.build();

}

return instance;

}

}

**5. Create Repository**

public class Repository {

private final ContactDao contactDao;

private final ExecutorService executor;

private final Handler handler;

public Repository(Application application) {

ContactDatabase db = ContactDatabase.getInstance(application);

contactDao = db.getContactDao();

executor = Executors.newSingleThreadExecutor();

handler = new Handler(Looper.getMainLooper());

}

// Insert (background thread)

public void insertContact(Contacts contact) {

executor.execute(() -> contactDao.insert(contact));

}

// Delete (background thread)

public void deleteContact(Contacts contact) {

executor.execute(() -> contactDao.delete(contact));

}

// Get all contacts (LiveData - auto-updates UI)

public LiveData<List<Contacts>> getAllContacts() {

return contactDao.getAllContacts();

}

}

**6. Create ViewModel**

public class MyViewModel extends AndroidViewModel {

private final Repository repository;

private final LiveData<List<Contacts>> allContacts;

public MyViewModel(@NonNull Application application) {

super(application);

repository = new Repository(application);

allContacts = repository.getAllContacts();

}

public LiveData<List<Contacts>> getAllContacts() {

return allContacts;

}

public void insert(Contacts contact) {

repository.insertContact(contact);

}

public void delete(Contacts contact) {

repository.deleteContact(contact);

}

}

**7. RecyclerView Item Layout (contact\_list\_item.xml)**

<layout xmlns:android="http://schemas.android.com/apk/res/android">

<data>

<variable

name="contact"

type="com.example.Contacts" />

</data>

<androidx.cardview.widget.CardView

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:layout\_margin="8dp"

android:elevation="6dp">

<LinearLayout

android:orientation="vertical"

android:padding="16dp"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content">

<TextView

android:text="@{contact.name}"

android:textSize="18sp"

android:textStyle="bold" />

<TextView

android:text="@{contact.email}"

android:textSize="14sp" />

</LinearLayout>

</androidx.cardview.widget.CardView>

</layout>

**8. RecyclerView Adapter with Data Binding**

public class MyAdapter extends RecyclerView.Adapter<MyAdapter.ViewHolder> {

private List<Contacts> contactList = new ArrayList<>();

static class ViewHolder extends RecyclerView.ViewHolder {

private final ContactListItemBinding binding;

public ViewHolder(ContactListItemBinding binding) {

super(binding.getRoot());

this.binding = binding;

}

}

@NonNull

@Override

public ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {

ContactListItemBinding binding = DataBindingUtil.inflate(

LayoutInflater.from(parent.getContext()),

R.layout.contact\_list\_item,

parent,

false

);

return new ViewHolder(binding);

}

@Override

public void onBindViewHolder(@NonNull ViewHolder holder, int position) {

Contacts current = contactList.get(position);

holder.binding.setContact(current);

}

@Override

public int getItemCount() {

return contactList.size();

}

public void setContacts(List<Contacts> contacts) {

this.contactList = contacts;

notifyDataSetChanged();

}

public Contacts getContactAt(int position) {

return contactList.get(position);

}

}

**9. Main Activity**

public class MainActivity extends AppCompatActivity {

private MyViewModel viewModel;

private MyAdapter adapter;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activity\_main);

adapter = new MyAdapter();

binding.recyclerView.setLayoutManager(new LinearLayoutManager(this));

binding.recyclerView.setHasFixedSize(true);

binding.recyclerView.setAdapter(adapter);

viewModel = new ViewModelProvider(this).get(MyViewModel.class);

viewModel.getAllContacts().observe(this, contacts -> adapter.setContacts(contacts));

// Swipe to delete

new ItemTouchHelper(new ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.LEFT) {

@Override public boolean onMove(@NonNull RecyclerView rv, @NonNull RecyclerView.ViewHolder vh, @NonNull RecyclerView.ViewHolder target) { return false; }

@Override public void onSwiped(@NonNull RecyclerView.ViewHolder vh, int dir) {

viewModel.delete(adapter.getContactAt(vh.getAdapterPosition()));

}

}).attachToRecyclerView(binding.recyclerView);

}

}

**10. Add New Contact Activity**

public class AddContactActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

ActivityAddContactBinding binding = DataBindingUtil.setContentView(this, R.layout.activity\_add\_contact);

MyViewModel viewModel = new ViewModelProvider(this).get(MyViewModel.class);

Contacts contact = new Contacts("", "");

binding.setContact(contact);

binding.saveButton.setOnClickListener(v -> {

if (!contact.getName().isEmpty() && !contact.getEmail().isEmpty()) {

viewModel.insert(new Contacts(contact.getName(), contact.getEmail()));

finish();

} else {

Toast.makeText(this, "Fields cannot be empty", Toast.LENGTH\_SHORT).show();

}

});

}

}

**Tools, Libraries, and APIs**

* **Room** (androidx.room.\*)
* **LiveData, ViewModel** (androidx.lifecycle.\*)
* **Data Binding** (androidx.databinding.\*)
* **RecyclerView** (androidx.recyclerview.widget.RecyclerView)
* **ItemTouchHelper** for swipe gestures.

**Best Practices & Latest Alternatives**

* **Coroutines + Flow (Kotlin)** instead of ExecutorService for background work.
* Use **DiffUtil** in RecyclerView instead of notifyDataSetChanged() for better performance.
* Implement **Migrations** instead of fallbackToDestructiveMigration() in production.
* Use **sealed UI states** or StateFlow instead of just LiveData for more control.
* Use **View Binding** if you don’t need full data binding.

**Part B – Additional Important Points**

1. **Testing**
   * Unit test DAO methods with Room.inMemoryDatabaseBuilder for fast, isolated DB tests.
2. **Error Handling**
   * Wrap DB calls in try-catch or return Result objects.
3. **Pagination**
   * Use **Paging 3** library for large datasets.
4. **Security**
   * Encrypt database using **SQLCipher** if sensitive data is stored.
5. **UI/UX**
   * Use Snackbar on delete with “Undo” option for better UX.
6. **Offline + Sync**
   * Combine Room with a network layer for offline-first apps.