**✅ Section 27: LiveData + ViewModel + Data Binding App**

**📚 Key Concepts Covered**

1. **ViewModel** – Lifecycle-aware data holder that survives configuration changes.
2. **LiveData** – Observable and lifecycle-aware data container.
3. **MutableLiveData** – Variant of LiveData used for updating values.
4. **Data Binding Integration** with ViewModel and LiveData.
5. **UI Persistence** across screen rotations without saving state manually.
6. **Loose coupling** between UI and data logic using MVVM architecture.

**Key Concepts Taught**

1. **ViewModel**
   * Manages UI-related data for activities/fragments.
   * Survives configuration changes (e.g., screen rotation).
   * Separates business logic from UI components.
2. **LiveData**
   * Lifecycle-aware observable data holder.
   * Automatically updates UI only when observers are active (e.g., STARTED/RESUMED).
   * Prevents memory leaks by auto-unregistering observers.
3. **Data Binding**
   * Binds UI components directly to data sources in XML.
   * Reduces boilerplate code (e.g., findViewById).
4. **Problem Solved**
   * Without ViewModel/LiveData: Data resets on configuration changes (e.g., screen rotation).
   * Solution: ViewModel persists data; LiveData auto-updates UI.

**🛠️ Tools, Libraries, and APIs Used**

* Android **Data Binding**
* AndroidX **Lifecycle Library**
  + ViewModel
  + LiveData / MutableLiveData
* MVVM Architecture Pattern
* ViewModelProvider
* DataBindingUtil

**🧩 Step-by-Step Implementation (with Code and Comments)**

**1. ✅ Enable Data Binding in build.gradle**

android {

...

buildFeatures {

dataBinding = true

}

}

Also, add **Lifecycle dependencies**:

dependencies {

def lifecycle\_version = "2.6.2" // latest stable at time of section

implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle\_version"

implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle\_version"

}

**2. ✅ Create Layout activity\_main.xml**

Use <layout> block and data binding variable:

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

<data>

<variable

name="myViewModel"

type="com.yourpackage.MyViewModel" />

</data>

<LinearLayout

android:orientation="vertical"

android:gravity="center"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView

android:text="Counter App"

android:textSize="20sp" />

<TextView

android:id="@+id/textViewCounter"

android:text="0"

android:textSize="30sp" />

<Button

android:onClick="@{() -> myViewModel.increaseCounter()}"

android:text="Increase" />

</LinearLayout>

</layout>

**3. ✅ Create MyViewModel.java**

public class MyViewModel extends ViewModel {

private final MutableLiveData<Integer> counter = new MutableLiveData<>();

public MyViewModel() {

counter.setValue(0); // Initialize the counter

}

public void increaseCounter() {

Integer currentValue = counter.getValue() != null ? counter.getValue() : 0;

counter.setValue(currentValue + 1);

}

public LiveData<Integer> getCounter() {

return counter;

}

}

🔍 **Explanation**:

* MutableLiveData allows updating values.
* getCounter() exposes it as LiveData, which is observable by UI.
* increaseCounter() updates the value.

**4. ✅ Connect ViewModel and DataBinding in MainActivity.java**

public class MainActivity extends AppCompatActivity {

private ActivityMainBinding binding;

private MyViewModel viewModel;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// Initialize data binding

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

binding.setLifecycleOwner(this); // Required for LiveData binding

// Initialize ViewModel

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

binding.setMyViewModel(viewModel);

// Observe LiveData changes to update the UI

viewModel.getCounter().observe(this, new Observer<Integer>() {

@Override

public void onChanged(Integer value) {

binding.textViewCounter.setText(String.valueOf(value));

}

});

}

}

🧠 **What’s Happening Here:**

* ViewModelProvider(this) ensures ViewModel survives rotation.
* observe(this, observer) watches LiveData.
* onChanged(...) gets called whenever counter changes.

**🌀 Configuration Change Survival (like screen rotation)**

* Without ViewModel: Counter resets to 0 after rotation.
* With ViewModel: Counter persists after rotation automatically.

**🧠 LiveData Safety Check in increaseCounter()**

Integer currentValue = counter.getValue() != null ? counter.getValue() : 0;

This prevents **NullPointerException** when LiveData hasn't been initialized.

**🔄 Lifecycle-Aware Behavior**

| **Component** | **Behavior After Rotation** |
| --- | --- |
| ViewModel | Retained |
| LiveData | Updates new observers |
| Activity | Recreated |
| TextView | Automatically updated via observer |

**✅ Best Practices & Modern Tips**

| **Area** | **Best Practice** |
| --- | --- |
| **Data Management** | Use ViewModel for all UI-related logic |
| **State Updates** | Use LiveData or StateFlow instead of manually updating views |
| **Architecture** | Use MVVM (Model-View-ViewModel) |
| **Null Safety** | Check .getValue() != null or use Objects.requireNonNull() |
| **ViewModel Sharing** | Use activityViewModels() for shared state in fragments |
| **Separation of Concerns** | Keep logic in ViewModel, not in Activity/Fragment |

**Best Practices & Industry Approaches**

1. **ViewModel**
   * **Do**: Store all UI data in ViewModels.
   * **Don’t**: Pass context (causes memory leaks).
2. **LiveData**
   * Use MutableLiveData internally; expose LiveData publicly.
   * Use Transformations for data manipulation (e.g., map, switchMap).
3. **Data Binding**
   * Bind click handlers in XML to reduce Java/Kotlin code.
   * Use binding adapters for custom logic (e.g., formatting text).
4. **Null Safety**
   * Always handle null in LiveData (e.g., counter.getValue() ?? 0).

**✅ Summary Checklist (What You Learned to Build Future Projects)**

* Use ViewModel to persist data across configuration changes.
* Use LiveData to observe and automatically update UI.
* Use DataBinding to remove boilerplate code (findViewById()).
* Always handle nulls when working with LiveData.
* Avoid tight coupling of UI and logic.
* Use proper AndroidX dependencies (ViewModel, LiveData).

**🔍 Part B – Important Topics Not Covered but Essential**

**1. ⚙️ Two-way Data Binding with LiveData**

Binding EditText directly to LiveData like this:

<EditText

android:text="@={myViewModel.name}" />

Requires MutableLiveData<String> in ViewModel.

**2. 🧪 Unit Testing ViewModel**

ViewModels are easy to test because they are lifecycle-independent:

@Test

public void testCounterIncrement() {

MyViewModel vm = new MyViewModel();

vm.increaseCounter();

assertEquals(1, vm.getCounter().getValue().intValue());

}

**3. ⚠️ Avoid Business Logic in Activities**

All logic should be pushed to ViewModel or Repository.

**4. 🔄 StateFlow vs LiveData (Modern Alternative)**

| **Feature** | **LiveData** | **StateFlow** |
| --- | --- | --- |
| Lifecycle-aware | ✅ Yes | ❌ No (manual handling) |
| Coroutine support | ❌ Limited | ✅ Full support |
| Default values | ❌ No | ✅ Yes |
| Jetpack Compose | 🚫 Legacy | ✅ Recommended |

📌 In **Jetpack Compose**, use **StateFlow or MutableState** instead of LiveData.

**5. 💬 Jetpack Compose Alternative**

This whole logic can be replaced in Compose using:

@Composable

fun CounterUI(viewModel: MyViewModel = viewModel()) {

val counter by viewModel.counter.observeAsState(0)

Text("Counter = $counter")

Button(onClick = { viewModel.increaseCounter() }) { Text("Increase") }

}

**Important Topics Not Covered**

1. **ViewModel with SavedStateHandle**
   * Persist data across process death (not just configuration changes).
   * Use SavedStateHandle to store IDs, keys, or small data.

java

public class MyViewModel extends ViewModel {

private SavedStateHandle savedStateHandle;

public MyViewModel(SavedStateHandle savedStateHandle) {

this.savedStateHandle = savedStateHandle;

} }

1. **Repository Pattern**
   * Separate data sources (API, DB) from ViewModel.
   * ViewModel calls a repository, which manages data operations.
2. **Kotlin Coroutines + LiveData**
   * Use liveData{} builder for asynchronous operations.

kotlin

val data = liveData {

emit(repository.fetchData())

}

1. **Unit Testing**
   * Test ViewModel logic with TestCoroutineDispatcher.
   * Use InstantTaskExecutorRule for LiveData testing.
2. **StateFlow as a LiveData Alternative**
   * Modern, Kotlin-only alternative for reactive streams.
   * Better integration with coroutines.

**✅ Conclusion**

You now understand and can apply:

* ViewModel to preserve UI data
* LiveData for observing data changes
* DataBinding to cleanly bind UI and ViewModel
* Full MVVM architecture in a clean and maintainable way