MVVM (Model View ViewModel) Architecture Pattern in Android

Overview

A design pattern called Model View ViewModel (MVVM) aids developers in separating the View, which is the user interface (UI), from the Model, which is the data. The View-Model component of the MVVM is in charge of exposing the data objects from the Model in a way that makes it simple for the View to use them.

Introduction

MVVM architecture android is a method of coding organization. The UI elements of an application can be kept separate from the business logic thanks to MVVM. Additionally, the database activities are kept separate from the business logic itself. Naturally, MVVM makes writing enjoyable and simple to comprehend.

MVVM is also in charge of separating tightly coupled components from one another. It is also crucial to understand that with MVVM architecture, the child does not have a reference to the parent. They use observables as a reference.

What is an Architecture Pattern?

A broad, reusable solution to a frequently occurring problem in software architecture within a specific context is what is known as an architectural pattern. The architectural patterns address a variety of problems in software, including performance restrictions on computer hardware, high availability, and risk minimization.

Even though an architectural pattern presents a system's picture, it is not an architecture. A software architecture's key coherent components are defined and solved by a concept called an architectural pattern. Numerous distinct architectures could use the same pattern and have similar properties. The definition of a pattern is frequently strictly described and widely available.

What is MVVM (Model View ViewModel)?

Model-view-viewmodel (MVVM) architecture android is an architectural design pattern for computer software that makes it easier to separate the creation of the graphical user interface (UI) from the creation of the business logic or back-end logic (the model) so that the view is independent of any particular model platform. The MVVM viewmodel is a value converter, which means it is in charge of making the model's data objects manageable and presentable by converting them. This is where the viewmodel differs from the view in that it controls the

majority, if not all, of the display logic for the view. The viewmodel might utilize a mediator pattern to organize access to the back-end logic according to the categories of use cases that the view supports.

Why Use MVVM in Android Development?

With MVVM architecture android, your view (represented by Activities and Fragments) and business logic are separated. For small projects, MVVM is sufficient, but as your software grows, your viewmodels begin to bloat. Responsibility division becomes challenging. In these situations, MVVM with Clean Architecture works quite well.

MVVM is the best architecture for Android app development:

1. There is low or no dependency on Android APIs.
2. High XML complexity.
3. Great in Unit Testability.

The Separate Code Layers of MVVM

The MVVM architecture android pattern consists of View, ViewModel, and Model. Each of them has different responsibilities in Android developments defined below:

- **View:**

It is in charge of creating the user interfaces that the user sees on the screen. The view is made up of Android components with UI components like TextView, Button, or Jetpack Compose UI. By monitoring data or UI states from the ViewModel, UI elements configure UI screens and send user events to the ViewModel. View does not contain business logic but only contains UI logic that displays the screen and user interactions.

- **ViewModel:**

  This is a stand-alone component that is independent of the View that stores business information or UI states that are propagated into UI elements from the Model. The link between ViewModel and Model is often numerous (one-to-many), and ViewModel alerts View of data changes as domain data or UI states. Google advises utilizing the ViewModel library when creating modern Android applications because it makes it simple for developers to store business data and maintain configuration states.

- **Model:**

  Encapsulates the domain and data model of the application, which often consists of business logic, intricate computations, and validation logic. Model classes are typically employed in repositories that contain data access like a set of executable domain

functions, together with distant services and local databases. The repositories guarantee the immutability of all app data and one source of truth across many data sources.

Ways to Implement MVVM in the Project

a. Using the DataBinding Library Released by Google

1. Set up DataBinding in your project: Add the DataBinding library to your project by including the following lines in your module-level build. gradle file:

```
android {
    ...
    dataBinding {
        enabled = true
    }
}
```

2. Create the necessary components:

- **Model:** Define your data classes or objects that represent the data in your application.
- **View:** Create your XML layout files that define the UI components.
- **ViewModel:** Create your ViewModel classes that handle the logic and provide data to the View.

3. Implement DataBinding in the XML layout:

- Wrap your XML layout with the <layout> tag.
- Use the <data> tag to define the data variables and import the ViewModel class.
- Bind the data variables to the UI elements using the @{} syntax.

4. Set up the ViewModel:

- Create a ViewModel class that extends ViewModel.
- Define the necessary LiveData or Observable fields to hold the data.
- Implement methods to update the data and handle user actions.

5. Connect the View and ViewModel:

- In your activity or fragment, retrieve the ViewModel using ViewModelProviders or the by viewModels() Kotlin property delegate.
- Use the binding object generated by DataBinding to access the UI elements in your layout.
- Bind the View and ViewModel together by setting the ViewModel as the binding variable.

6. Observe data changes:

- Use the observe() method from LiveData or Observable fields to observe data changes in the ViewModel.

- Update the UI elements in the View based on the changes received from the ViewModel.

b. Using Any Tool Like RxJava for DataBinding

1. Set up DataBinding and RxJava in your project: Add the necessary dependencies for DataBinding and RxJava in your module-level build.gradle file.

2. Create the necessary components:

- **Model:** Define your data classes or objects that represent the data in your application.
- **View:** Create your XML layout files that define the UI components.
- **ViewModel:** Create your ViewModel classes that handle the logic and provide data to the View.

3. Implement DataBinding in the XML layout:

- Wrap your XML layout with the <layout> tag.
- Use the <data> tag to define the data variables and import the ViewModel class.
- Bind the data variables to the UI elements using the @{} syntax.

4. Set up the ViewModel:

- Create a ViewModel class that holds the necessary RxJava Observables or Subjects to manage the data

flow.
- Implement methods to update the data and handle user actions using RxJava operators.

5. Connect the View and ViewModel:

- In your activity or fragment, retrieve the ViewModel instance.
- Use the binding object generated by DataBinding to access the UI elements in your layout.
- Bind the View and ViewModel together by setting the ViewModel as the binding variable.

6. Observe data changes using RxJava:

- Use RxJava operators like map(), filter(), or flatMap() to transform or filter the data emitted by Observables.
- Subscribe to the Observables to receive data updates.
- Update the UI elements in the View based on the changes received from the ViewModel.

Example of MVVM Architecture Pattern

Example of the MVVM (Model-View-ViewModel) architecture pattern in an Android application:

a. Model:

User.kt: Represents the data model for a user.

```
data class User(val id: String, val name: String, val email: String)
```

b. View:

MainActivity.kt: Represents the main activity class that displays user information.

```
class MainActivity: AppCompatActivity() {
    private lateinit var viewModel: UserViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        viewModel = ViewModelProvider(this).get(UserViewModel::class.j

        observeUserData()
        loadUserData()
    }

    private fun observeUserData() {
        viewModel.user.observe(this, Observer { user ->
            // Update the UI with the user data
            textViewId.text = user.id
            textViewName.text = user.name
            textViewEmail.text = user.email
        })
    }

    private fun loadUserData() {
        viewModel.loadUser()
```

```
    }
}
```

c. ViewModel:

UserViewModel.kt: Acts as a mediator between the View and the Model, providing data and handling business logic.

```
class UserViewModel : ViewModel() {
    private val userRepository = UserRepository()

    private val _user = MutableLiveData<User>()
    val user: LiveData<User> = _user

    fun loadUser() {
         // Retrieve user data from repository or API
        val user = userRepository.getUser()
        _user.value = user
    }
}
```

d. Repository:

UserRepository.kt: Handles data operations, such as fetching user data from a remote data source or database.

```
class UserRepository {
```

```
    fun getUser(): User {
        // Fetch user data from a remote data source or database
        return User("1", "John Doe", "john.doe@example.com")
    }
}
```

Explanation:

The User class serves as the data model in this illustration. The MainActivity serves as the View and is in charge of projecting the user data into the screen. The UserViewModel performs the role of the ViewModel by giving the View the user data and managing the logic for loading the user data. Data operations, such as obtaining user data from a remote source or database, are the responsibility of the UserRepository.

The MainActivity uses the ViewModelProvider to generate an instance of the UserViewModel and monitors the user LiveData to refresh the UI whenever the user data changes. The UserViewModel updates the user's LiveData and loads the user's data from the UserRepository, which causes the UI update in the MainActivity.

Step by Step Implementation

Example of implementing the MVVM architecture in a user login application:

The following code must be included in the build.gradle(build.gradle ()) file of an Android application to enable DataBinding:

```
Enable DataBinding:
android {
   dataBinding {
       enabled = true
      }
}
Add lifecycle dependency:
implementation 'android.arch.lifecycle:extensions:1.1.1'
```

Step 1: Create a New Project

- Starting with the File Tab, select New => New Project and then click.
- Select a bare activity.
- Decide between Java and Kotlin.
- Choose the bare minimal SDK.

Step 2: Modify String.xml File

All the required strings used in the activity are given below:

```
<resources>
    <string name="app_name">MVVM</string>
    <string name="heading">MVVM Architecture</string>
    <string name="email_hint">Email-ID</string>
```

```
    <string name="password_hint">Password</string>
    <string name="button_text">Sign-In</string>
</resources>
```

Step 3: Creating the Model Class

Make a model class that registers the user's password and email address. To create a good Model class, take into account the code below.

```
class Model(
    var email: String?
    var password: String?
)
```

Step 4: Working with the activity_main.xml File

To receive inputs for Email-ID and Password, open the activity_main.xml file and add EditText. To verify the user's input and display the proper Toast message, a Login Button is also necessary.

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:bind="http://schemas.android.com/tools">

    <!-- binding object of ViewModel to the XML layout -->
```

```
<data>
    <variable
        name="viewModel"
        type="com.example.mvvmarchitecture.AppViewModel" />
</data>

<!-- Provided Linear layout-->
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="center"
    android:layout_margin="18dp"
    android:background="#E5EFC3"
    android:orientation="vertical">

    <!-- TextView for the heading -->
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/heading"
        android:textAlignment="center"
        android:textColor="@android:color/green"
        android:textSize="20sp"
        android:textStyle="bold" />

    <!-- EditText field for Email-ID -->
    <EditText
        android:id="@+id/inEmail"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginStart="12dp"
        android:layout_marginTop="60dp"
        android:layout_marginEnd="9dp"
        android:layout_marginBottom="18dp"
        android:hint="@string/email_hint"
        android:inputType="textEmailAddress"
        android:padding="10dp"
        android:text="@={viewModel.userEmail}" />
```

```
        <!-- EditText field for password -->
        <EditText
            android:id="@+id/inPassword"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="9dp"
            android:layout_marginEnd="10dp"
            android:hint="@string/password_hint"
            android:inputType="textPassword"
            android:padding="10dp"
            android:text="@={viewModel.userPassword}" />

        <!-- Login Button -->
        <Button
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="25dp"
            android:layout_marginTop="64dp"
            android:layout_marginEnd="22dp"
            android:background="#A2D5AA"
            android:fontFamily="@font/roboto"
            android:onClick="@{() -> viewModel.onButtonClicked()}"
            android:text="@string/button_text"
            android:textColor="@android:color/background_light"
            android:textSize="25sp"
            android:textStyle="bold"
            bind:toastMessage="@{viewModel.toastMessage}" />

        <ImageView
            android:id="@+id/imageView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="120dp"
            app:srcCompat="@drawable/banner" />

    </LinearLayout>
</layout>
```

Step 5: Creating the ViewModel Class

All of the methods required by the application layout will be contained in this class. As it transforms data into streams and alerts the view whenever the toast message property changes, the ViewModel class will extend BaseObservable.

```
import android.text.TextUtils
import android.util.Patterns
import androidx.databinding.BaseObservable
import androidx.databinding.Bindable
import androidx.databinding.ObservableField

class AppViewModel : BaseObservable() {

    // This creates an object of the Model class
    private val model: Model = Model("", "")

    // Creates string variables for
    // toast messages
    private val successMessage = "Login success"
    private val errorMessage = "Entered Email-ID or Password is not va

    // Creates ObservableField for toast message
    val toastMessage: ObservableField<String> = ObservableField()

    // Creates getter and setter methods
    // for email variable
    @Bindable
    fun getUserEmail(): String? {
        return model.email
    }
```

```
    fun setUserEmail(email: String) {
        model.email = email
        notifyPropertyChanged(BR.userEmail)
    }

    // Creates getter and setter methods
    // for password variable
    @Bindable
    fun getUserPassword(): String? {
        return model.password
    }

    fun setUserPassword(password: String) {
        model.password = password
        notifyPropertyChanged(BR.userPassword)
    }

    // Login Button Logic
    fun onButtonClicked() {
        if (isValid())
            toastMessage.set(successMessage)
        else
            toastMessage.set(errorMessage)
    }

    // checks if the input fields are valid
    private fun isValid(): Boolean {
        return !TextUtils.isEmpty(getUserEmail()) &&
                Patterns.EMAIL_ADDRESS.matcher(getUserEmail()).matches
                getUserPassword()?.length ?: 0 > 5
    }
}
```

Step 6: Define Functionalities of View in the MainActivity File

The application's User Interface is updated by the View class. The Binding Adapter will activate the View layer depending on modifications made to the toast message provided by ViewModel. The observer will then be informed of the data changes by the Toast message setter. The view will then follow through with the necessary actions.

```
import android.os.Bundle
import android.view.View
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.databinding.BindingAdapter
import androidx.databinding.DataBindingUtil
import com.example.mvvmarchitecture.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // The ViewModel will update the Model
        //After observing the changes in the View

        val activityMainBinding: ActivityMainBinding =
        DataBindingUtil.setContentView(this, R.layout.activity_main)

        activityMainBinding.viewModel = AppViewModel()
        activityMainBinding.executePendingBindings()
    }

    companion object {
        @JvmStatic
        @BindingAdapter("toastMessage")
        fun showToast(view: View, message: String?) {
```

```
            message?.let {
                Toast.makeText(view.context,
                               message, Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

Advantages of MVVM Architecture

- **Simple to create :** Since the View and the logic are kept apart, it is possible for many developer teams to work on various components at once. Designers can concentrate on the UI while programmers implement the logic (ViewModel and Model).
- **Less difficult to test:** The user interface (UI) of an application is one of the most difficult components to test. It helps Developers to build tests for the ViewModel and Model without using the View because they are independent of the View.
- **Easy to Maintain :** The separation of the application's various components makes maintenance easier. The application code is hence significantly simpler to comprehend and maintain. It is simpler to comprehend where new features should be added and how they relate to the current design. Additional architectural patterns (dependency inversion, services, and more) are likewise simpler to create with an MVVM.

Disadvantages of MVVM Architecture

- Data binding and the various MVVM components' communication can be difficult.
- Views and view models are hard to reuse in code.
- In layered views and intricate UIs, managing view models and their state is challenging.
- It can be challenging for beginners to use MVVM.
- Testing for application can be difficult.
- Debugging issue due to XML.
- Very limited options are available in view methods.

Comparison with Other Architectures

a. MVP vs MVVM

MVP	MVVM
By employing Presenter as a route of communication between Model and View, it solves the issue of having a dependent View.	Because it leverages data binding, this architectural style is more event-driven and makes it simple to separate the view from the main business logic.
As a Presenter layer, observables are not required in this.	Because there is no Presenter layer in this, observables are required.
In MVP, UI is used more frequently.	In MVVM, there is no user interface.
The model layer sends the Presenter, who then sends it to the View, the response to the user's input.	The Model layer responds to the user's input by conducting operations and returning the result to the View.

More classes will be included in the project file in addition to the code.	More classes but less code per class will be found in the project file.

b. MVC vs MVVM

MVC	MVVM
MVC typically requires manual updating of the view by the controller.	MVVM utilizes data binding to establish a connection between the view and the ViewModel, allowing automatic synchronization.
In MVC, the view can contain some presentation logic.	In MVVM, the ViewModel handles most of the presentation logic, keeping the view more focused on displaying data.
Less testable in comparison to MVVM.	With MVVM, since the ViewModel contains most of the business logic, it can be easily unit tested without the need for the view or model.
MVC separates concerns by assigning specific roles to the model, view, and controller.	MVVM takes it a step further by introducing the ViewModel as a separate component responsible for presenting data to the view.
MVC, with its simpler structure, may be more suitable for smaller projects or applications with straightforward requirements.	MVVM can introduce additional complexity due to the introduction of the ViewModel layer.

c. MVI vs MVVM

MVI	MVVM
The data flow in MVI is unidirectional, with the view sending intents to the model and the model emitting new states that the view then notices.	MVVM, on the other hand, allows bidirectional data binding between the view and ViewModel.
In MVI, the model's state is typically immutable. Each intent triggers a new state that is calculated by the model.	In MVVM, the model's state can be mutable, and changes in the model are automatically propagated to the view.
MVI places a strong emphasis on capturing user intents as distinct actions.	MVVM focuses more on exposing data and commands from the ViewModel to the view.
MVI is very testable due to its unidirectional flow and focus on pure functions. By supplying specified intents and confirming the resultant states, unit testing the behavior of the model is made simple.	MVVM also allows for effective testing, but the bidirectional data binding and view updates can introduce some complexity in unit testing.

Conclusion

- MVVM separates the concerns of UI, business logic, and data by introducing the ViewModel layer, which acts as a mediator between the View and the Model.
- The View is responsible for creating the user interfaces and displaying data to the user. It interacts with the ViewModel to retrieve data and handle user interactions.
- The ViewModel holds the business logic and exposes

data to the View through observable properties or LiveData. It updates the View with the latest data and handles user actions.

- The Model encapsulates the domain and data model of the application. It includes the business logic, data access, and validation rules. The ViewModel interacts with the Model to retrieve and update data.
- MVVM promotes low coupling and high cohesion between components, making the codebase more modular, maintainable, and scalable.
- There are different ways to implement MVVM in an Android project, such as using the DataBinding library provided by Google or combining MVVM with other tools like RxJava for reactive programming.
- MVVM is especially beneficial for Android app development due to its low dependency on Android APIs, support for complex XML layouts, and improved unit testability.