Migrating to MVVM Architecture using KOTLIN

Introduction

Model-View-ViewModel (MVVM) is a design pattern that helps separate concerns within an application, making it easier to manage and test. This guide will walk you through the process of migrating an existing Android project to the MVVM architecture.

Steps to Migrate to MVVM

1. Project Structure

   First, update your project structure to reflect the MVVM pattern. Here is an example structure;

```
com.yourapp
├── data
│   ├── model
│   │   └── User.kt
│   ├── repository
│   │   └── UserRepository.kt
│   └── source
│       ├── local
│       │   └── LocalDataSource.kt
│       └── remote
│           └── RemoteDataSource.kt
├── di
│   └── AppModule.kt
├── ui
│   ├── base
│   │   └── BaseActivity.kt
│   │   └── BaseFragment.kt
│   ├── user
│   │   ├── UserActivity.kt
│   │   ├── UserFragment.kt
│   │   ├── UserViewModel.kt
│   │   └── UserAdapter.kt
├── util
│   └── Extensions.kt
└── App.kt
```

2. Data Layer

a. Create Models

Move your existing data models into the data/model package.

```kotlin
package com.mvvmTest.data.model

import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable

new *
@Serializable
data class AvailableCabsResponse(
    @SerialName("AvailableCabList")
    val availableCabList: List<AvailableCab>,
    @SerialName("PassengerLat")
    val passengerLat: String,
    @SerialName("PassengerLon")
    val passengerLon: String,
    @SerialName("selectedDrivers")
    val selectedDrivers: List<String>,
    @SerialName("selectedRiders")
    val selectedRiders: List<String>,
    @SerialName("VehicleTypes")
    val vehicleTypes: List<VehicleType>
)
```

- **@Serializable**: indicates that instances of this class can be converted into a stream of bytes for serialization
- **@SerialName**: can be used to specify the name of a property during serialization/deserialization. This can be helpful when dealing with data coming from external sources that might have different property names.

b. Create Repositories

Repositories handle data operations and abstract the data sources.
Create repository classes in the data/repository package.
Most of the code written in **ExecuteWebServerUrl.java** class may have to be replaced with this repository as ExecuteWebServerUrl.java contains the functions related to API calls.

```
class Repository @Inject constructor(private val generalFunc: GeneralFunctions) {

    new *
    fun loadAvailableCabs(data: HashMap<String, String>) {
        execute(data)
    }
    new *
    fun execute(params: HashMap<String, String>){
        performPostCall()
    }
    new *
    private fun performPostCall() {

    }
}
```

- **@Inject**: This annotation is used to mark fields or constructor parameters in your classes that require dependency injection. When you apply @Inject to a field or constructor parameter, you signal to Dagger Hilt that this member needs to be provided with an appropriate instance during object creation.

3. ViewModel Layer

   Create ViewModel classes to manage UI-related data and handle business logic. This class will be used as a SharedViewModel which will help us share the data between the activity and fragments.

```
@HiltViewModel
class MainViewModel @Inject constructor(private val repository: Repository): ViewModel() {

    private val _parameters = MutableLiveData<HashMap<String, String>>()
    new *
    val parameters: LiveData<HashMap<String, String>> get() = _parameters
    new *
    init {
        getUserName()
    }
    new *
    fun getParameterData(parameters: HashMap<String, String>) {
        _parameters.value = parameters
    }

    new *
    fun loadAvailableCabs(parameters: HashMap<String, String>) {
        repository.loadAvailableCabs(parameters)
    }

    new *
    fun getUserName(){

    }
}
```

- **@HiltViewModel**: This annotation is used to simplify the creation and dependency injection of ViewModels in your Android application. You don't need to manually create or manage the lifecycle of the ViewModel component for your ViewModels. Hilt takes care of these aspects.

4. View Layer

Update your activities and fragments to interact with ViewModel.

```
@AndroidEntryPoint
class MainFragment : Fragment() {

    private lateinit var binding: FragmentMainHeaderNewBinding
    private val viewModel: MainActivityNewViewModel by activityViewModels()
    new *
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        binding = FragmentMainHeaderNewBinding.inflate(inflater, container, attachToParent: false)
        return binding.root
    }
    new *
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewModel.parameters.value?.let { it: HashMap<String, String>
            viewModel.setParameterData(it)
        }
    }
}
```

- In Fragments, ViewModel is initialized using "**activityViewModels()"** which allows you to create a single instance of a ViewModel that can be accessed and shared by all fragments within the same host activity. This is particularly useful for data that needs to be consistent across fragments.
- **"viewmodel.parameters.value"** will help us in the getting the data from the ViewModel

5. Dependency Injection

Dependency Injection (DI) is a design pattern used in software development. In simpler terms, DI allows you to inject dependencies into a class, rather than the class creating or finding those dependencies itself. We are using the Dagger Hilt library to manage dependencies.

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
    new *
    @Provides
    @Singleton
    fun provideMainRepository(
        generalFunctions: GeneralFunctions
    ): MainRepository {
        return MainRepository(generalFunctions)
    }
    new *
    @Provides
    @Singleton
    fun provideGeneralFunctions(@ApplicationContext context: Context): GeneralFunctions {
        return GeneralFunctions(context)
    }
}
```

- **@Module**: Marks a class as a Dagger Hilt module.
- **@InstallIn**: This annotation plays a crucial role in specifying the Hilt component where a module should be installed.
- **@Provides:** This annotation helps in defining how dependencies are created or obtained within the application
- **@Singleton**: This annotation is used to mark a dependency or a class that provides a dependency as a singleton

6. Use ViewBinding

Enable ViewBinding in your build.gradle file.

```
android {

    buildFeatures{ ApplicationBuildFeatures it ->
        viewBinding = true
    }
}
```

7. Common Utils

Move utility functions and classes into the utils package.

```
package com.mvvmTest.utils

import android.view.View

new *
fun View.show() {
    this.visibility = View.VISIBLE
}

new *
fun View.hide() {
    this.visibility = View.GONE
}
```

8. Coroutines

   We have been using Runnable with Handler in the Activities and Fragments to schedule tasks on the main thread from a background thread. This is useful for updating the UI after completing a background operation. Here is an example;

But with the help of MVVM and Kotlin, we can make use of ViewModels and Coroutines to achieve the same. Here is an example;

In ViewModel:

```
    private val _panelHeight = MutableLiveData<Int>()
    new *
    val panelHeight: LiveData<Int> get() = _panelHeight
    new *
    fun startBackgroundTask() {
        viewModelScope.launch { this: CoroutineScope
            try {
                withContext(Dispatchers.Main) { this: CoroutineScope
                    _panelHeight.value = 280
                }
            } catch (e: Exception) {
                delayAndRetry()
            }
        }
    }
    new *
    private fun delayAndRetry() {
        viewModelScope.launch { this: CoroutineScope
            delay( timeMillis: 20)
            startBackgroundTask()
        }
    }
}
```

a. Private MutableLiveData: **_panelHeight** is a MutableLiveData instance that holds the result of the background task.
b. Public LiveData: **panelHeight** is a publicly exposed, read-only LiveData that can be observed by the UI layer to get updates.
c. **ViewModelScope** is a property that is tied to the lifecycle of the ViewModel. This scope is designed to launch coroutines that will be automatically canceled when the ViewModel is cleared, preventing memory leaks and ensuring that background work is appropriately cleaned up when the ViewModel is no longer in use.

d. **Coroutine Contex**t: It uses **Dispatchers.Main** by default, meaning coroutines launched in viewModelScope will run on the main thread unless another dispatcher is specified.

In Fragment:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    viewModel.startBackgroundTask()

    viewModel.panelHeight.observe(viewLifecycleOwner) { height ->
        Log.d( tag: "PanelHeight", height.toString())
    }
}
```

a. **Observe LiveData:** The fragment observes panelHeight LiveData from the ViewModel using viewLifecycleOwner, ensuring the observer is lifecycle-aware. When panelHeight changes, the lambda function is executed to update the UI.

b. Start Background Task: The background task is started by calling viewModel.startBackgroundTask().

9. ConfigPassagerTripStatus

- Without MVVM

```
updateReconnectionTask = new UpdateFrequentTask( mInterval: 19000);
updateReconnectionTask.setTaskRunListener(() -> {});
```

UpdateFrequentTask is an inbuilt class from the general file that takes an argument “minInterval”.

```
@Override
public void onTaskRun() {
    configTripStatus();
}

1 usage  new *
private void configTripStatus(){
    Context mContext = MyApp.getInstance().getCurrentAct();
    HashMap<String, String> parameters = new HashMap<~>();
    parameters.put("type", "configPassengerTripStatus");
    ExecuteWebServerUrl exeWebServer = new ExecuteWebServerUrl(mContext, parameters);
    exeWebServer.execute();
}
```

When the listener gets triggered, the onTaskRun will be called continuously after the given interval.

- With MVVM

```
private val _updateMarker = MutableLiveData( value: 0)
new *
val updateMarker: LiveData<Int> get() = _updateMarker

new *
fun getConfigPassengerUpdateStatus(
) {
    val updateDestMarkerTask = UpdateFrequentTask( mInterval: 19000)
    updateDestMarkerTask.setTaskRunListener {
        _updateMarker.value = _updateMarker.value!! + 1
    }
    updateDestMarkerTask.startRepeatingTask()
}
```

The UpdateFrequentTask class can be called from the ViewModel and the live data variable can observed from the Fragment/Activity.

Or

Coroutines can used instead of the UpdateFrequentTask class for setting up the interval

```
fun getConfigPassengerUpdateStatus(){
    viewModelScope.launch { this: CoroutineScope
        while (_updateMarker.value!! >= -1){
            delay( timeMillis: 19000)
            _updateMarker.value = _updateMarker.value!! + 1
        }
    }
}
```