## Code refactor Retrofit + Kotlin Coroutines = MVVM

**NetworkLayer**

We created the object NetworkLayer to handle moshi, retrofit and movieDbService interface outside of the Main Activity. Decoupling it will give us a higher scalability as we will have to use these components in different parts of the app, and not just in the Main Activity.

Also, we instantiated an apiClient which is a class that handles the movieDbService interface and all the functions.

**ApiClient**

This class has MovieDbService part of the constructor and it holds a suspended function that mimics the functions created at MovieDbService. It needs to be a suspend function as it will be used in **Coroutine**.

suspend fun getMovieById(movie\_ID: Int): Response<GetMovieByIdResponse>{  
 return movieDbService.getMovieById(movie\_ID)  
}

**MVVM**

In order to use the API client created above, we have to use it inside a coroutine and not the Main Activity. Therefore, MVVM model comes into play to make this possible, separating each concern, including the coroutine.

**SharedRepository**

This class handles the requests to the API. Therefore, the getMovieId function is created there and it returns the API’s response body.

suspend fun getMovieById(movie\_ID: Int) : GetMovieByIdResponse? {  
  
 val request = NetworkLayer.apiClient.getMovieById(movie\_ID)  
  
 if(request.*isSuccessful*){  
 return request.body()!!  
 }  
  
 return null  
}

SharedRepository -> ApiClient -> MovieDbService

**SharedViewModel**

The ViewModel has reference of the repository, as it will be responsible to control it.  
Also, it has a live data that will be observed by the View to get the API response.

private val \_getMovieByIdLiveData = MutableLiveData<GetMovieByIdResponse?>()  
val getMovieByIdLiveData: LiveData<GetMovieByIdResponse?> = \_getMovieByIdLiveData

The live data is broken in 2 parts as shown above. The public one is only a LiveData which reflects the other livedata result. With that, whoever is observing that, won’t be able to change the information on it, making it a Read-Only type of variable. In the other hand, we have \_movieByIdLiveData which is a MutableLiveData that will be changing according to the responses from the API request.

In Gradle, we had to add the dependency:

implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.0-beta01"

This will allow us to add coroutine to our view model, following the MVVM architecture.

Back in the SharedViewModel, we create the following coroutine to execute the function getMovieById created in the repository. The result of this function updates the live data variable.

*viewModelScope*.*launch* **{** val response = repository.getMovieById(movie\_ID)  
  
 \_getMovieByIdLiveData.postValue(response)  
**}**

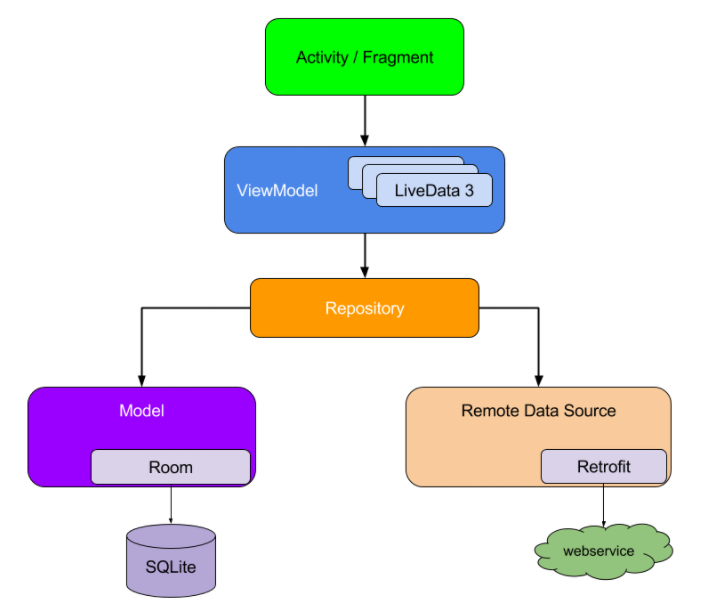
**Main Activity**

Now that we have all the viewModel prepared, we just have to invoke all the functions in the Main Activity, and set the live data observer.

Therefore, we instantiated the viewModel, and added a design patter called observer:

viewModel.getMovieByIdLiveData.observe(this)**{**response **-> … }**

Basically, the Main Activity will be always observing the movieByIdLiveData, so if there’s any change on it, it will get the response and update the view. This is one of main concept of MVVM.

The diagram describes the MVVM architecture.

In our project, we have an activity so far.  
In the next layer, we have our SharedViewModel, which contains getMovieByIdLiveData.  
Down another layer, we also have the SharedRepository.  
Our SharedRepository then contains the getMovieById function, that mimics our ApiClient.  
And finally, we have the ApiClient, which will be the final layer that is using Retrofit to access the webservice and get the desired information.