MVVM with Kotlin Coroutines and Retrofit [Example]

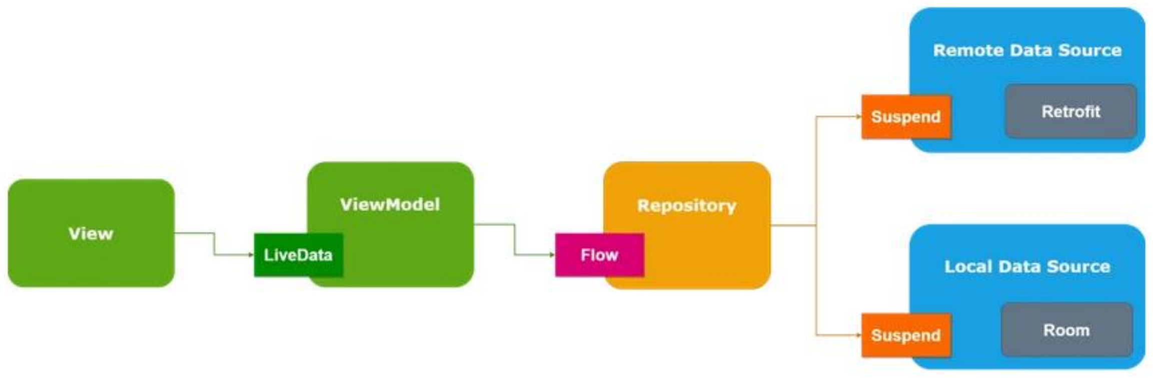
Link - https://medium.com/android-beginners/mvvm-with-kotlin-coroutines-and-retrofit-example-d3f5f3b09050

Coroutines are a neat new feature of the Kotlin language that allow us to write asynchronous code in a more idiomatic way. — This also means you can write asynchronous code the same way you would normally write synchronous code in your project.

Already, I have explained about MVVM in detail in my another post. Please check that for better understanding of MVVM. in this post, I am focusing on coroutines and retrofit working together.

MVVM With Retrofit and Recyclerview in Kotlin [Example]

The flow diagram for the coroutines with retrofit in viewModel.



Coroutines are helpful in two main problems,

1. A long-running task that can block the main thread
2. Main safety allows you to ensure that any suspend function can be called from the main thread

According to the Kotlin docs it is stated that coroutines are a lightweight alternative to threads.

“Coroutines provide a way to avoid blocking a thread and replace it with a cheaper and more controllable operation”

Before we begin I would like to briefly address the concept and the commonly used functions in Coroutine.

Coroutines build upon regular functions by adding two new operations. In addition to **invoke** (or call) and **return**, coroutines add **suspend** and **resume**.

- **suspend** — pause the execution of the current coroutine, saving all local variables
- **resume** — continue a suspended coroutine from the place it was paused

Suspend Function

A **suspending function** is just a regular **Kotlin function** with an additional **suspend** modifier which indicates that the **function** can **suspend** the execution of a coroutine.

```
suspend fun getAllMovies() : Response<List<Movie>>
```

You can only call suspend functions from other suspend functions, or by using a coroutine builder like launch to start a new coroutine.

We use call back functions when we get response from our Async task. Suspend and resume work together to replace callbacks.

To understand suspend functions, we should also know about provided dispatchers by Kotlin.

To specify where the coroutines should run, Kotlin provides three dispatchers that you can use:

- **Dispatchers.Main** — Use this dispatcher to run a coroutine on the main Android thread. This should be used only for interacting with the UI and performing quick work. Examples include calling suspend functions, running Android UI framework operations, and updating LiveData objects.
- **Dispatchers.IO** — This dispatcher is optimized to perform disk or network I/O outside of the main thread. Examples include using the Room component, reading from or writing to files, and running any network operations.
- **Dispatchers.Default** — This dispatcher is optimized to perform CPU-intensive work outside of the main thread. Example use cases include sorting a list and parsing JSON.

Lets, see this with an example -> We are calling our api through coroutines. So, we use **Dispatchers.IO.**

```
CoroutineScope(Dispatchers.IO + exceptionHandler).launch {
        val response = mainRepository.getAllMovies()
    }
```

When we call getAllMovies() suspend method, then it suspends our coroutine. The coroutine on the main thread will be resumed with the result as soon as the **withContext** block is complete.

Note: Using **suspend** doesn't tell Kotlin to run a function on a background thread. It's normal for **suspend** functions to operate on the main thread.

Launch and Async

launch and async are the most commonly used **Coroutine builder.**

launch– *Launches new coroutine without blocking current thread and returns a reference to the coroutine as a Job. The coroutine is canceled when the resulting job is cancelled.*

async– *Creates new coroutine and returns its future result as an implementation of Deferred. The running coroutine is canceled when the resulting object is cancelled.*

Take a look at this piece of code as an example.

```
launch {
  delay(2000)
  println("launch block")
}

val result: Deferred<String> = async {
  delay(3000)
  "async block"
}

println(result.await())
```

From the example, the difference between **launch** and **async** is that **async** can return the future result which has a type of **Deferred<T>**, and we can call **await()** function to the **Deferred** variable to get the result of the Coroutine while launch only executes the code in the block without returning the result.

Coroutine Scope

Coroutine Scope defines a scope for coroutines. Every **coroutine builder** (like launch, async, etc) is an extension on CoroutineScope. When the scope dies, the Coroutines inside will be out of the picture too. Fortunately, the Android **lifecycle-viewmodel-ktx** provides a really easy way to get a **Coroutine Scope** in the ViewModel. I will show you how to do so later.

Coroutines in your Android Project

To begin add the following library to your build.gradle file dependencies:

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.4.1"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.2"
```

Note*: You'll also need to be on kotlin version 1.3 or better.*

Making it work with Retrofit?

Retrofit is a type-safe HTTP client for Android and Java.

Starting from Retrofit 2.6.0 you no longer require the Call Adapter as Retrofit now includes built-in support for Kotlin suspend modifier on functions.

In order to begin, let's add the retrofit dependencies into our app level build.gradle file:

```
// Networking
   implementation "com.squareup.retrofit2:retrofit:2.9.0"
   implementation "com.squareup.okhttp3:okhttp:4.7.2"
   implementation "com.squareup.okhttp3:logging-interceptor:4.7.2"
   implementation "com.squareup.retrofit2:converter-gson:2.9.0"
```

Declaring our interface.

For this example I am using *https://howtodoandroid.com/movielist.json* api to get list of movies.

Observe the below snippet for our interface:

```
interface RetrofitService {

   @GET("movielist.json")
   suspend fun getAllMovies() : Response<List<Movie>>
}
```

You may notice that instead of Call<T>, we now have a function with the suspend modifier defined in our interface function.

According to Retrofit documentation this function will, behind the scenes behave as a normal Call.enqueue operation.

Also we wrap our response in a Response object to get metadata about our request response e.g. information like response code.

We no longer have to await() anymore as this is handled automatically! As with all networking on Android its done on the background. And this is a very clean way of doing so!

Building Retrofit Service

Our Retrofit instance will look like the following code snippet:

```
interface RetrofitService {

   @GET("movielist.json")
   suspend fun getAllMovies() : Response<List<Movie>>
```

```
  companion object {
    var retrofitService: RetrofitService? = null
    fun getInstance() : RetrofitService {
      if (retrofitService == null) {
        val retrofit = Retrofit.Builder()
          .baseUrl("https://howtodoandroid.com/")
          .addConverterFactory(GsonConverterFactory.create())
          .build()
        retrofitService = retrofit.create(RetrofitService::class.java)
      }
      return retrofitService!!
    }

  }
}
```

ViewModel with Coroutines

A CoroutineScope keeps track of all coroutines it creates. Therefore, if you cancel a scope, you cancel all coroutines it created. This is particularly important if you're running coroutines in a ViewModel.

If your ViewModel is getting destroyed, all the asynchronous work that it might be doing must be stopped. Otherwise, you'll waste resources and potentially leaking memory. If you consider that certain asynchronous work should persist after ViewModel destruction, it is because it should be done in a lower layer of your app's architecture.

Add a CoroutineScope to your ViewModel by creating a new scope with a SupervisorJob that you cancel in **onCleared()** method. The coroutines created with that scope will live as long as the ViewModel is being used.

Coroutines and LiveData

LiveData is an observable value holder for UI and we are expected to be able to access the value from the main thread. With the release of livedata-2.1.0-alpha1, google provided the interoperability between LiveData and Coroutines.

```
class MainViewModel constructor(private val mainRepository: MainRepository) : ViewModel() {

  val errorMessage = MutableLiveData<String>()
  val movieList = MutableLiveData<List<Movie>>()
  var job: Job? = null

  val loading = MutableLiveData<Boolean>()

  fun getAllMovies() {
        job = CoroutineScope(Dispatchers.IO).launch {
            val response = mainRepository.getAllMovies()
            withContext(Dispatchers.Main) {
                if (response.isSuccessful) {
                    movieList.postValue(response.body())
                    loading.value = false
                } else {
                    onError("Error : ${response.message()} ")
                }
            }
        }

    }

    private fun onError(message: String) {
        errorMessage.value = message
        loading.value = false
    }

    override fun onCleared() {
        super.onCleared()
        job?.cancel()
    }

}
```

Exception Handling in Kotlin Coroutines

If you consider the above example, you can see we are wrapping our code inside a try-catch exception. But, when we are working with coroutines we can handle an exception using a global coroutine exception handler called **CoroutineExceptionHandler**.

To use it, first, we create an exception handler in our ViewModel,

```
val exceptionHandler = CoroutineExceptionHandler { _, throwable ->
        onError("Exception handled: ${throwable.localizedMessage}")
    }
```

and then we attach the handler to the ViewModelScope.

So, our code looks like,

```
class MainViewModel constructor(private val mainRepository: MainRepository) : ViewModel() {
    val errorMessage = MutableLiveData<String>()
    val movieList = MutableLiveData<List<Movie>>()
    var job: Job? = null
    val exceptionHandler = CoroutineExceptionHandler { _, throwable ->
```

```
        onError("Exception handled: ${throwable.localizedMessage}")
    }
    val loading = MutableLiveData<Boolean>()
    fun getAllMovies() {
        job = CoroutineScope(Dispatchers.IO + exceptionHandler).launch {
            val response = mainRepository.getAllMovies()
            withContext(Dispatchers.Main) {
                if (response.isSuccessful) {
                    movieList.postValue(response.body())
                    loading.value = false
                } else {
                    onError("Error : ${response.message()} ")
                }
            }
        }
    }

    private fun onError(message: String) {
        errorMessage.value = message
        loading.value = false
    }

    override fun onCleared() {
        super.onCleared()
        job?.cancel()
    }
}
```

Kotlin Coroutines With Retrofit Example

Now, lets see the example of list movies using kotlin coroutines and retrofit.

Required Dependencies

Here are the things you need to add to your **build.gradle**

```
// Networking
    implementation "com.squareup.retrofit2:retrofit:2.9.0"
    implementation "com.squareup.okhttp3:okhttp:4.7.2"
    implementation "com.squareup.okhttp3:logging-interceptor:4.7.2"
    implementation "com.squareup.retrofit2:converter-gson:2.9.0"

    //Coroutine
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.4.1"
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.2"
    implementation 'com.google.code.gson:gson:2.8.6'

    //viewModel
    implementation "android.arch.lifecycle:extensions:1.1.1"

    //Glide
    implementation 'com.github.bumptech.glide:glide:4.12.0'
    kapt 'com.github.bumptech.glide:compiler:4.12.0'
```

First, setup the retrofit service.

Model.kt

```
data class Movie(val name: String, val imageUrl: String, val category: String)
```

RetrofitService.kt

```
interface RetrofitService {

    @GET("movielist.json")
    suspend fun getAllMovies() : Response<List<Movie>>

    companion object {
        var retrofitService: RetrofitService? = null
        fun getInstance() : RetrofitService {
            if (retrofitService == null) {
                val retrofit = Retrofit.Builder()
                    .baseUrl("https://howtodoandroid.com/")
                    .addConverterFactory(GsonConverterFactory.create())
                    .build()
                retrofitService = retrofit.create(RetrofitService::class.java)
            }
            return retrofitService!!
        }

    }
}
```

Next step is to setup the repository.

MainRepository.kt

```
class MainRepository constructor(private val retrofitService: RetrofitService) {

    suspend fun getAllMovies() = retrofitService.getAllMovies()

}
```

Setup the ViewModel,

MyViewModelFactory.kt

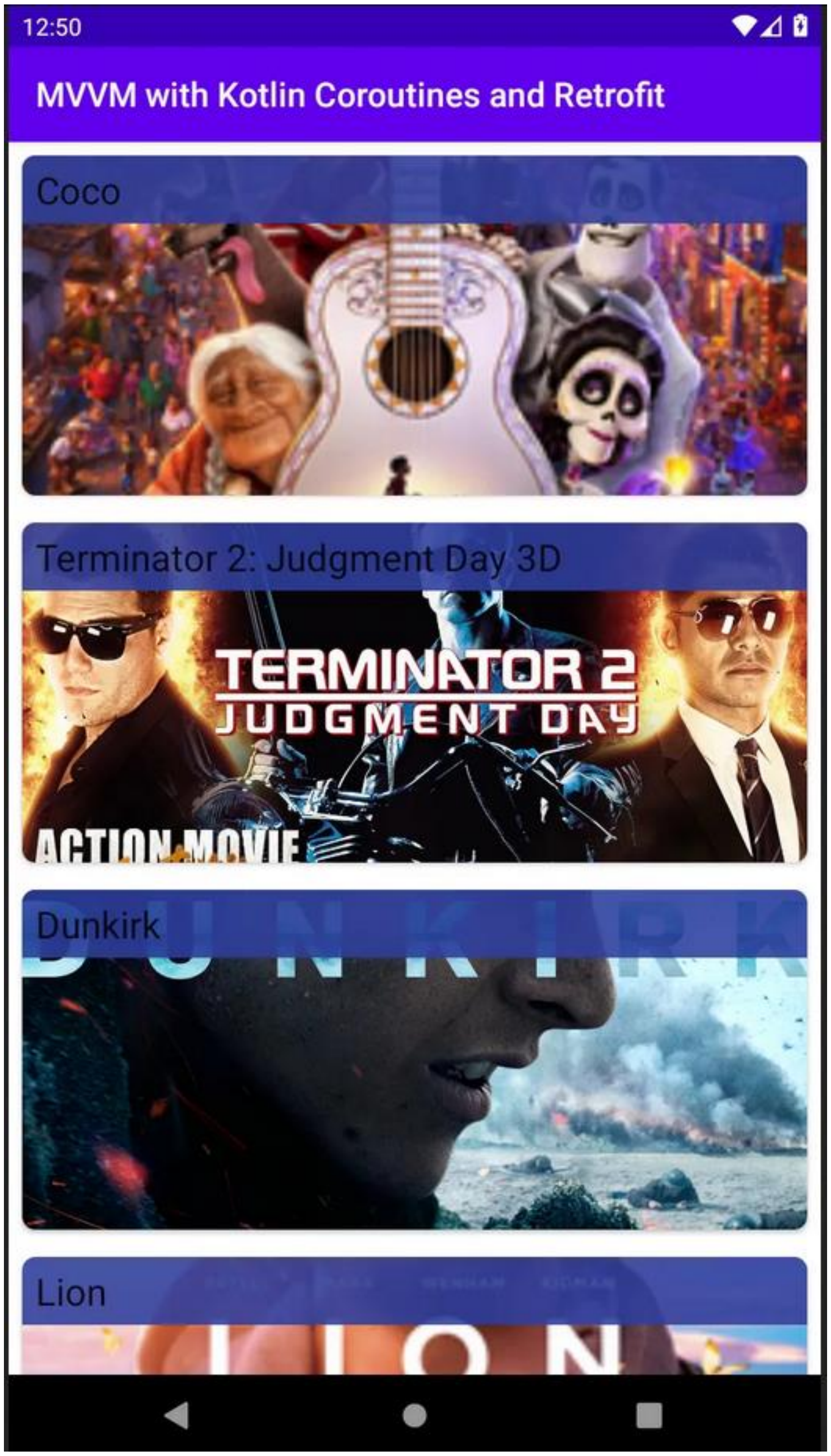
```
class MyViewModelFactory constructor(private val repository: MainRepository):
ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return if (modelClass.isAssignableFrom(MainViewModel::class.java)) {
            MainViewModel(this.repository) as T
        } else {
            throw IllegalArgumentException("ViewModel Not Found")
        }
    }
}
```

MainViewModel.kt

```
class MainViewModel constructor(private val mainRepository: MainRepository) : ViewModel() {

    val errorMessage = MutableLiveData<String>()
    val movieList = MutableLiveData<List<Movie>>()
    var job: Job? = null
    val exceptionHandler = CoroutineExceptionHandler { _, throwable ->
        onError("Exception handled: ${throwable.localizedMessage}")
    }
    val loading = MutableLiveData<Boolean>()

    fun getAllMovies() {
        job = CoroutineScope(Dispatchers.IO + exceptionHandler).launch {
            val response = mainRepository.getAllMovies()
            withContext(Dispatchers.Main) {
                if (response.isSuccessful) {
                    movieList.postValue(response.body())
                    loading.value = false
                } else {
                    onError("Error : ${response.message()} ")
                }
            }
        }

    }

    private fun onError(message: String) {
        errorMessage.value = message
        loading.value = false
    }

    override fun onCleared() {
        super.onCleared()
        job?.cancel()
    }

}
```

Finally, in our MainActivity setup the viewmodel and call the getAllMovies() method of the viewModel.

```
class MainActivity : AppCompatActivity() {
    lateinit var viewModel: MainViewModel
    private val adapter = MovieAdapter()
    lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        val retrofitService = RetrofitService.getInstance()
        val mainRepository = MainRepository(retrofitService)
        binding.recyclerview.adapter = adapter

        viewModel = ViewModelProvider(this,
MyViewModelFactory(mainRepository)).get(MainViewModel::class.java)


        viewModel.movieList.observe(this, {
            adapter.setMovies(it)
        })

        viewModel.errorMessage.observe(this, {
            Toast.makeText(this, it, Toast.LENGTH_SHORT).show()
        })

        viewModel.loading.observe(this, Observer {
            if (it) {
                binding.progressDialog.visibility = View.VISIBLE
            } else {
                binding.progressDialog.visibility = View.GONE
            }
        })

        viewModel.getAllMovies()

    }
}
```


12:50
MVVM with Kotlin Coroutines and Retrofit
Coco
Terminator 2: Judgment Day 3D
TERMINATOR 2
JUDGMENT DAY
ACTION MOVIE
Dunkirk
DUNKIRK
Lion
LION