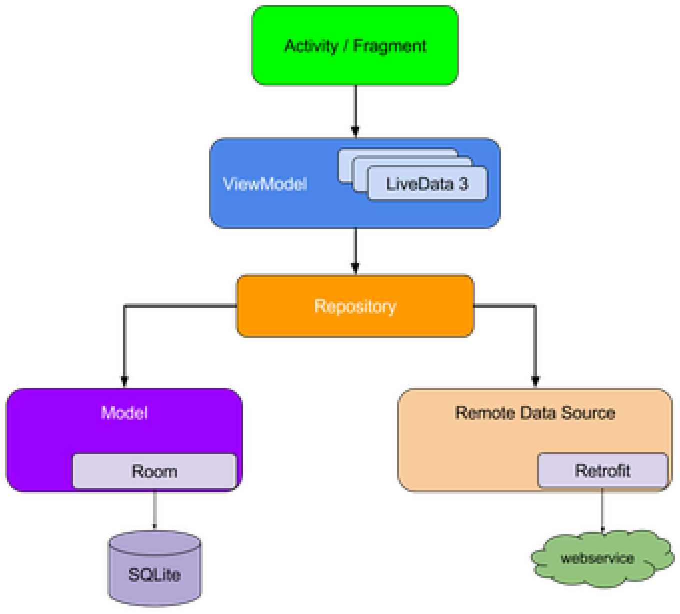
Android MVVM architecture

So far most of the code we have written in our apps has been written in activities. A better practice is to only have code that relates to UI on the activities. This results in a separation of concerns where the UI is only responsible for displaying the UI. Fetching and storing data is handled by other independent components.

This is the recommended architecture that we will implement



In this pattern the activity is only responsible for displaying the UI.

The data is accessed from the repository. The repository can get the data either from a web service (REST API) or from a local database.

The ViewModel is the middleman between the repository and the UI. When the UI requests for data, the ViewModel is responsible for channeling the request to the repository and also sending the data back to the activity.

Implementing Android MVVM

1. Add the following dependencies to your app level build.gradle file. Be sure to sync afterwards

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:
1.5.0'
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.0-
alpha02'
implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.4.0-
alpha02'
implementation 'androidx.activity:activity-ktx:1.2.4'
```

2. You apiClient and apiInterface files should look like this

```
object ApiClient {
  var retrofit = Retrofit.Builder()
    .baseUrl("http://13.244.243.129")
    .addConverterFactory(GsonConverterFactory.create())
    .build()

  fun <T> buildApiClient(apiInterface: Class<T>): T {
    return retrofit.create(apiInterface)
  }
}
```

```
interface ApiInterface {
  @POST("/students/register")
   suspend fun registerStudent(@Body registrationRequest:
RegistrationRequest): Response<RegistrationResponse>
}
```

These files should be inside the `api` package. With these you should also create your RegistationRequest and RegistrationResponse data classes in the models package.

3. Create a new package called `repository` . In this package create a Kotlin file called UserRepository

```
class UserRepository {
  val apiInterface = ApiClient.buildApiClient(ApiInterface::class.java)
  suspend fun registerUser(registrationRequest: RegistrationRequest):
Response<RegistrationResponse> =
    withContext(Dispatchers.IO) {
      var resp = apiInterface.registerStudent(registrationRequest)
      return@withContext resp
    }
}
```

Instead of retrofit callbacks we are using kotlin coroutines to make the network request. A *coroutine* is an instance of suspendable computation. It is similar to a thread in that it takes a block of code and runs it concurrently with the rest of the code.

4. Create a new package called `viewmodel`. In it create a .kt file called UserViewModel

```
class UserViewModel: ViewModel() {
  var registrationLiveData = MutableLiveData<RegistrationResponse>()
  var regError = MutableLiveData<String>()
  val userRepository = UserRepository()

  fun registerUser(registrationRequest: RegistrationRequest){
    viewModelScope.launch {
      val response = userRepository.registerUser(registrationRequest)
      if (response.isSuccessful){
        registrationLiveData.postValue(response.body())
      }
      else{
        regError.postValue(response.errorBody()?.string())
      }
    }
  }
}
```

A `ViewModel` object provides the data for a specific UI component, such as a fragment or activity. It contains data-handling business logic to communicate with the model.(repository) For example, the `ViewModel` can call other components to load the data, and it can forward user requests to modify the data.

`viewModelScope` is a predefined `CoroutineScope` that is included with the `ViewModel` KTX extensions. Note that all coroutines must run in a scope. A `CoroutineScope` manages one or more related coroutines.

`launch` is a function that creates a coroutine and dispatches the execution of its function body to the corresponding dispatcher.
We have some livedata objects here that are responsible for updating the activity with the data that is obtained from the repository. LiveData is an observable data holder class. The obtained data is posted to the livedata objects which are observed by the UI. Any change to the livedata is immediately updated on the UI.

5. Finally in the activity you can have the following:

```
class MainActivity : AppCompatActivity() {
  val userViewModel: UserViewModel by viewModels()
  lateinit var btnRegister: Button

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(layout.activity_main)
    btnRegister = findViewById(R.id.btnRegister)
    btnRegister.setOnClickListener {
      var regRequest = RegistrationRequest("Emma Mutiso", "0700000221",
"emmake@gmail.com", "1921-05-07", "KENYAN",
        "emmie21!")
      userViewModel.registerUser(regRequest)
    }

  }

  override fun onResume() {
    super.onResume()
    userViewModel.registrationLiveData.observe(this, Observer {
response->
      Toast.makeText(baseContext, "Registration Success", Toast.
LENGTH_LONG).show()
    })

    userViewModel.regError.observe(this, Observer { str->
      Toast.makeText(baseContext, str, Toast.LENGTH_LONG).show()
    })
  }
}
```

When the button is clicked, you should obtain the details from the form and validate them, then package them into a RegistrationRequest object then invoke the register user method on your viewmodel. The viewmodel will then invoke the registerUser method on the repository which makes the network call and returns a response that is sent back to the viewmodel. The viewmodel populates posts the data to the livedata objects. In your onresume function, you are observing the livedata objects and updating the UI accordingly.