Android architecture components : MVVM

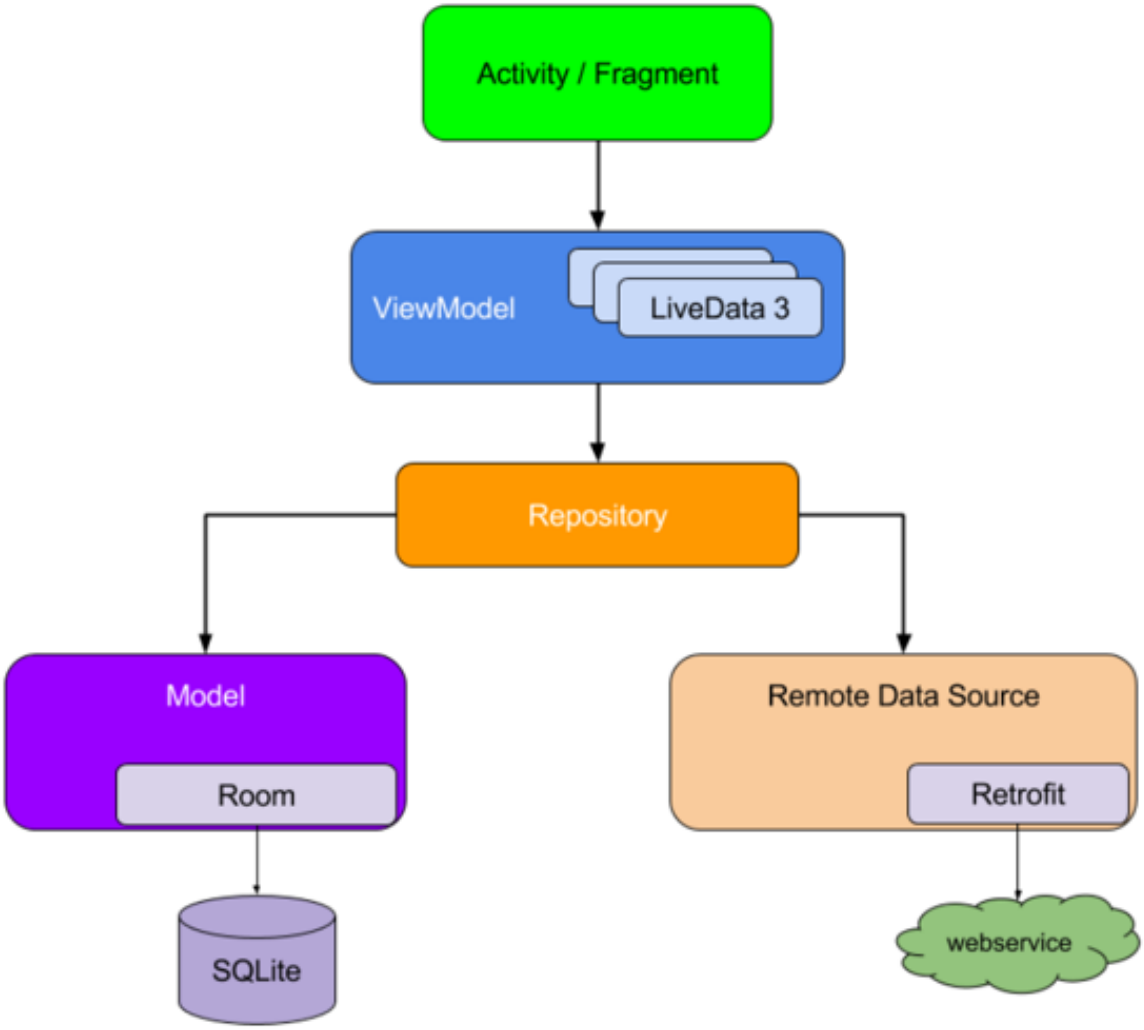
Clara Gros
Babacar Toure

February 2020

Contents

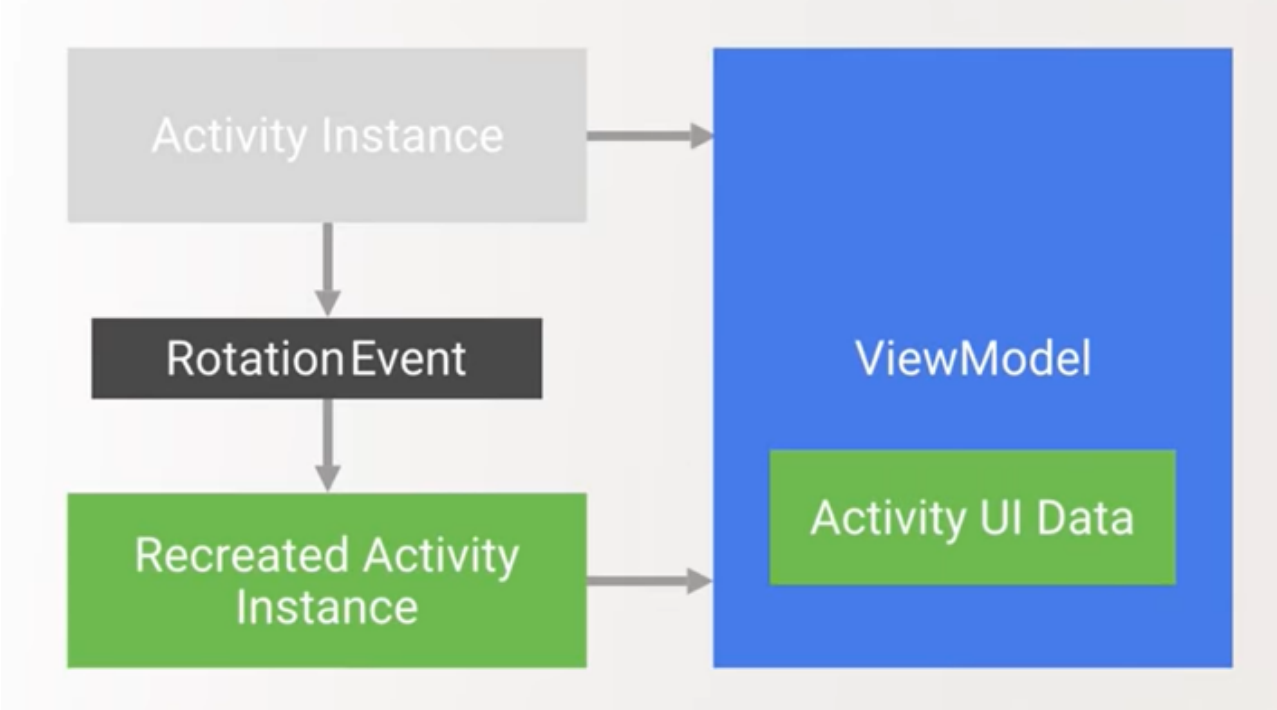
1 Introduction

This document will give insights on how to build an Android app with architecture components. There are bunch of different librairies that are supposed to help us build more robust and maintainable apps [2]. What does that mean ?
As beginners we often just put almost all the codes in our activities/fragments (no **separation of concerns** e.g. store and process data, start and stop different long running tasks...). For smaller projects, that's not really a problem but when an app gets bigger, this type of code can turn into a problem because it gets harder to make changes to it or even completely swap implementations without always having a huge wip effect through the rest of the codebase and managing the activity and fragments lifecycles become harder the most stuff we put in there. Also the spaghetti code is hard to test because many different parts depend on each other [1].
Architectural patterns are there to help us create more separated modular components where each part of the program has a well defined responsibility and can ideally be modified or replaced without touching any other component. Android provides us the framework and the core classes that interact with the Android system but developpers have to decide on how to organize their app and which architecture they use. We won't use all of the architecure components at our disposal but only the ones that make sense to us.The approach Android recommands consists in an MVVM architecture (short for Model ViewModel Model) where the data sources and repositories are the model, the activities and fragments are the views or the UI in other words and the viewModel class is obviously the ViewModel. If implemented correctly, we would have a nice clean architecture where all layers are modular and decoupled from each other and every part has well defined responsibilities : every layer only knows about the components directly below it. The viewModel gets the data from the repository but it doesn't have to know how the data is retrieved from the different sources and the UI controller trust the data from the viewModel onto the screen but it doesn't have to store the data itself or initiate any database operations directly.



2 ViewModel [3]

The viewModel component basically has the job of holding and preparing all the data for the UI so we don't have to put any of it directly into our activity/fragment. Instead, the activity or fragment connects to this viewModel, gets all the necessary data from there and then only has the job of drawing it onto the screen and reporting user interactions to the viewModel. The viewModel then forwards these user interactions to the underlying layers of the app either to load new data or to make changes to the data set. So the viewModel basically works as the gateway for the UI controller to the rest of the app and we don't initiate any database operations from our controller directly so the controller itself doesn't know what is going on in the underlying layers. This way we keep our activity and fragment classes clean.



The best thing about viewModel class is it survives configuration changes. When we rotate an Android device or make any other runtime configuration change, the activity on the screen gets destroyed and recreated for the purpose of providing an alternative layout file or other resources. That also means we lose the state of this activity and its member variables and we basically start with a completely new one. With viewModel, the new activity receives the same viewModel instance that still contains all the data. We also don't need to add any logic to handle configuration changes, such as the user rotating the device's screen. The viewModel is automatically restored when the configuration changes, so as soon as the new fragment is created, it receives the same instance of ViewModel, and the callback is invoked immediately using the current data.

3 LiveData

But that's not all, there's another architecture component that will play a major role here and this is liveData. LiveData is a wrapper that can hold any type of data and it can be observed by the UI controller which means whenever the data in this liveData object changes, the observer automatically gets notified with the new data and can refresh the UI in reaction to it. Another great thing about liveData is that it is lifecycle aware which mmeans that it knows when the activity or fragment that observes it is in the background and automatically stops updating it until it come back into the foreground so we don't have to manually stop and resume observations in our controllers lifecycle methods and it also cleans up any references when the associated controller is destroyed.

To summarize, the viewModel survives configuration changes and in it we have observable liveData that automatically does the right thing at the right time in the lifecycle of our controllers and together they saves us from a lot of lifecycle related issues and potential bugs and memory leaks.

4 Repository

A first idea for implementing the ViewModel might involve directly calling the Webservice to fetch the data and assign this data to our LiveData object. This design works, but by using it, our app becomes more and more difficult to maintain as it grows. It gives too much responsibility to the viewModel class, which violates the separation of concerns principle. Additionally, the scope of a ViewModel is tied to an Activity or Fragment lifecycle, which means that the data from the Webservice is lost when the associated UI object's lifecycle ends. This behavior creates an undesirable user experience. Instead, our ViewModel delegates the data-fetching process to a new module, a repository.
This is an ordinary Java class and not an architecture component but it's a recommended approach because in the repository we can mediate between different data sources (webservers and local SQLlite dbs, such as persistent models and caches). This way, the viewModel doesn't have to interact with the different data sources directly : it doesn't have to care where the data comes from or how it is fetched and instead repository modules handle data operations. They provide a clean API so that the rest of the app can retrieve this data easily. They know where to get the data from and what API calls to make whenever data is updated.

References

[1] *Android architecture app components example*. URL: https://www.youtube.com/watch?v=ARpn-1FPNE4.

[2] *Guide to app architecture*. URL: https://developer.android.com/jetpack/docs/guide.

[3] *ViewModel*. URL: https://www.youtube.com/watch?v=5qlIPTDE274.