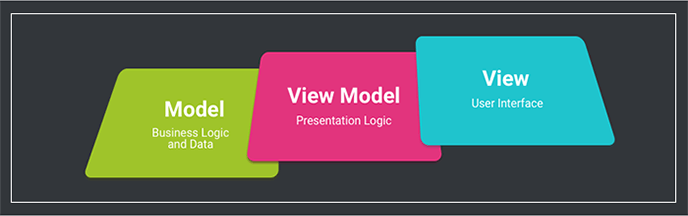
#### Technical Architecture & Blue print

#### C:\Users\msd3kor\Downloads\Untitled Diagram.png

#### MVVM: Model View ViewModel+ Data binding

The key points of MVVM is:



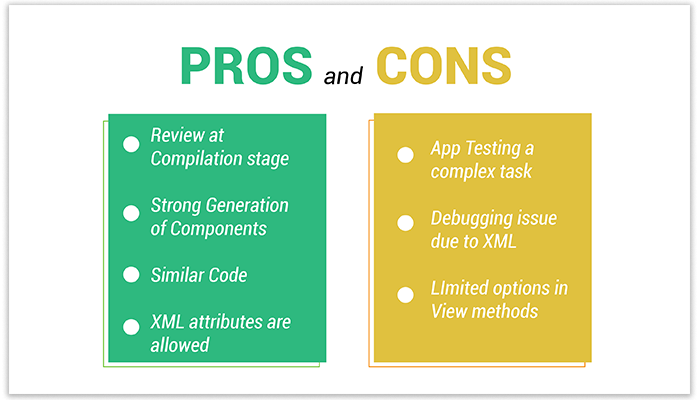
* The user’s interactions*call methods*on the ViewModel
* The ViewModel stores state, manages state, and*exposes change events*when the state changes
* The Model is the data mapped into the observable fields (or any other form of event emission)
* The View *subscribes for change events exposed by the viewmodel* to update itself

The supposed benefit is that View can be “dumb” as all it does is display what’s in the ViewModel, and then what is in the ViewModel can be verified to have “the right values under the right conditions”.

The downside is:

* All state must be moved to the ViewModel (and be somewhat “duplicated”, which is why ViewModel and View must be kept in sync — this is what databinding frameworks help with)
* the ViewModel stores the state, so we must be able to persist/restore the ViewModel state when the process is re-created (which is typically Android-specific)
* Method calls to View are replaced with event emission, so some form of the Observer pattern needs to be implemented — typically combination of PublishSubject/BehaviorSubject, but people try to hack LiveData to work like a PublishSubject and it doesn’t

In this case, the learning curve is the possible overhead, and there aren’t real inherent problems/limitations with the actual *idea*behind it (other than that it binds a ViewModel to a single view, similarly to MVP). Go MVVM!



Pros:

1. more reusability compare to presenter.
2. reactive pattern can be easy by subscribe to the ViewModel which is hard in MVP

Cons:

1. More boilerplate code even with syntax sugar with Kotlin(data class/sealed class/ when etc)
2. You need some smart way to avoid rebind the views from the same data every time you receive a new ViewModel

# Used Android Architecture Components Part of [Android Jetpack](https://developer.android.com/jetpack).

Android architecture components are a collection of libraries that help you design robust, testable, and maintainable apps. Start with classes for managing your UI component lifecycle and handling data persistence.

* Manage your app's lifecycle with ease. New [lifecycle-aware components](https://developer.android.com/topic/libraries/architecture/lifecycle) help you manage your activity and fragment lifecycles. Survive configuration changes, avoid memory leaks and easily load data into your UI.
* Use [LiveData](https://developer.android.com/topic/libraries/architecture/livedata) to build data objects that notify views when the underlying database changes.
* [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel) Stores UI-related data that isn't destroyed on app rotations.
* [Room](https://developer.android.com/topic/libraries/architecture/room) is an a SQLite object mapping library. Use it to Avoid boilerplate code and easily convert SQLite table data to Java objects. Room provides compile time checks of SQLite statements and can return RxJava, Flowable and LiveData observables.