MVI Architecture with Android

Rim Gazzah · Follow
Published in The Startup
6 min read · Nov 2, 2020

Listen Share More

Photo by Petri Heiskanen on Unsplash

The application lifespan is tied to its flexibility to scale for that it needs a solid base that's why for every project the most important step is to create the app architecture, after a good long discussion with the technical team about defining the elements included in the system, the functionality of each element and how they will be

communicating with each other, we have to put a clear design of the overall architecture.

There are different architectures for android applications, from what I've experienced the last years the MVVM and the MVI architecture are the most common architectures used for large scale applications, even for each of them there is no one way to be implemented and it depends on the application needs, also the developers' styles that worked on it, cause I believe that independently of the Android framework every developer has their unique experience in software development and what they bring to the table is not only their knowledge but also their special way of thinking, problem-solving and designing code.

Why MVI?

With no clear state management, view rendering along with the business logic can get a little bit messy as the application grows or adding functionalities or a feature that was not planned beforehand, and let's be honest that can happen often, it's rare to have all the features clearly and fully defined from the start of the project specifications, the more the app codebase is scalable the more it's flexible to embrace new ideas and updates.

The business logic and the UI rendering get tangled as a result of the state crises, how that happens? well:

- The Presenter/ViewModel has his own state
- The business logic produces its own state
- Trying to Synchronising both of the above states
- If there are no clear management of the inputs to the Presenter/ViewModel -> results of tangles processed outputs-> messy business logic and View rendering -> code smells like the Fragment/Activity become a black hole class (by attracting more responsibilities) or having divergent classes (when you want to make an update but you have to change a class in many different ways for many different reasons) and that results from a poor separate of concern.

What's the benefit of adapting the MVI architecture pattern :

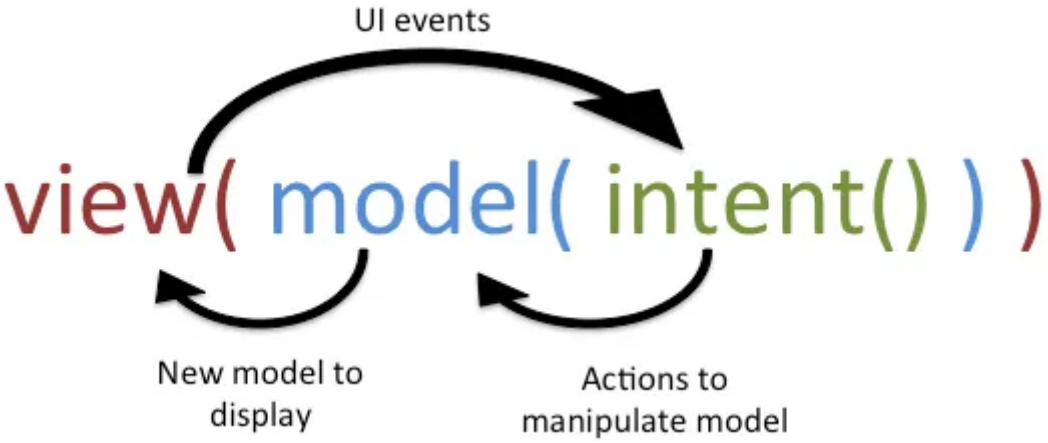
- **State** management using **immutability** to have a **single source of truth.**
- **Unidirectional** data flow

- reproducible state -> simple code reuse
- better separation of concern -> maintainability

Breaking down the layers

MVI stands for **M**odel-**V**iew-**I**ntent :

- **M**odel: instead of having a separate state for the View, ViewModel, and the Data layer, the Model will be the single source of truth of the overall state, and it's immutable to ensure that it will be updated from only one place.
- **V**iew: represent the UI layer, the activity, or the fragment, it defines the user actions and renders the state emitted by the model.
- **I**ntent: do not confuse it with the Android Intent, it's the intention to perform an action either by the user or the app itself.



http://hannesdorfmann.com/android/mosby3-mvi-2

Relying on functional programming each layer get input and feeds the next layer with its output, mathematically we expect the function f(x) to have the same resulted output for the same x input, applying this logic using immutable data and represent it with sealed classes in Kotlin(yes sealed classes will become your best friend in MVI architecture)

So let's have a closer look at each layer and talk in more details about their interaction with each other:

Model

The definition of Model in other architecture like MVP or MVVM is different, the Model in previous patterns is it the data layer and the domain layer that represents

the bridge between the app and the remote data source.

In MVI pattern, the model is also the data but represented as an **immutable state**, it means that the state will be updated in only one place in the app, which is the business logic, and that will ensure that the state won't be changed in no other place in the app, therefore the business logic is the single source of truth that create the immutable Model.

That's how the state will be represented, a sealed class that holds the data:

```
1  sealed class HomeState : ViewState{
```


Now we have a unique state, we don't have to manage the loading, success, or exceptions separately in different calls or operations, with a **render** method in the view that **observes** the state changes and updates the UI or executes a logic according to it.

I did an abstract definition of the view, this how I applied my logic:

Using reactive programming observing the state returned by the ViewModel and applying the new state within the render() function that will be implemented in the BaseActivity child (you can do the same with a BaseFragment)

PS: especially with a shared ViewModel between different Fragments it's better to use state management to not end up with messy "if-else" statements trying to manage the logic for each state of each fragment in the ViewModel and the Views 😨

```kotlin
abstract class BaseActivity<INTENT : ViewIntent, ACTION : ViewAction, STATE : ViewState
        VM : BaseViewModel<INTENT, ACTION, STATE>>(private val modelClass: Class<VM>) :
    RootBaseActivity(), IViewRenderer<STATE> {

    /* viewState will be locked on edit for child classes
     * it will only be updated on state change
     * for child classes they can access it on read only by the mState attribute
     */
    private lateinit var viewState: STATE
    val mState get() = viewState

    private val viewModel: VM by lazy {
        viewModelProvider(
            this.viewModelFactory,
            modelClass.kotlin
        )
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

         //....
        viewModel.state.observe(this, {
            viewState = it
            render(viewState)
        })
      //....
    }

// ...
}
```

BaseActivity.kt hosted with ❤️ by GitHub view raw

```kotlin
interface IViewRenderer<STATE> {
    fun render(state: STATE)
}
```

IViewRenderer.kt hosted with ❤️ by GitHub view raw

Implementing state logic is cleaner, centralized, and more structured, besides it's like you declare all your use cases in the ViewState and define them in the View.

```kotlin
class HomeActivity :
    BaseActivity<HomeIntent, HomeAction, HomeState, HomeViewModel>(HomeViewModel::class

      // ....
    override fun render(state: HomeState) {
        homeProgress.isVisible = state is HomeState.Loading
        homeMessage.isVisible = state is HomeState.Exception
        homeListCharacters.isVisible =
            state is HomeState.ResultSearch || state is HomeState.ResultAllPersona

        when (state) {
            is HomeState.ResultAllPersona -> {
                mAdapter.updateList(state.data)
            }
            is HomeState.ResultSearch -> {
                mAdapter.updateList(state.data)
                // other logic ...
            }
            is HomeState.Exception -> {
                homeMessage.text = state.callErrors.getMessage(this)
            }
        }
    }
}
```

HomeActivity.kt hosted with ❤️ by GitHub view raw

View and Intent

So we know the story from when the view observes on the State and render each time there is a new one, but now we will unfold the process that triggers the state to change from the View up :

Starting by creating the View (layout file and the Fragment /Activity), then defining all the operation on the View either click action or other action generated by the app itself that's called the **Intent,** to put it more simply the intention to do something.

For instance: the user types a word in the search bar then click on the button search, the click action will send to the ViewModel the **Intent** of SearchChatacter with the name as the text entered by the user

```kotlin
sealed class HomeIntent : ViewIntent {
    object LoadAllCharacters : HomeIntent()
    data class SearchCharacter(val name: String) : HomeIntent()
    object ClearSearch : HomeIntent()
}
```

HomeIntent.kt hosted with ❤️ by GitHub view raw

when an **Intent** is stimulated, it'll be dispatched to the ViewModel where it will be interpreted to the corresponding **Action,** here you can ask why we have to interpret the Intent? can't we simply work with it without using Actions? well, that's the trick you can have different Intents for the same Action as you can see in the example below, furthermore, the Intents can be emitted by different Views because you can have multiple Fragment using the same ViewModel. So the ViewModel maps the **Intent** to the appropriate **Action.**

```kotlin
sealed class HomeAction : ViewAction {
    data class SearchCharacters(val name: String) : HomeAction()
    object AllCharacters : HomeAction()
}
```

HomeAction.kt hosted with ❤️ by GitHub view raw

```kotlin
class HomeViewModel @Inject constructor(private val dataManager: CharactersManager) :
    BaseViewModel<HomeIntent, HomeAction, HomeState>() {

    fun intentToAction(intent: HomeIntent): HomeAction {
        return when (intent) {
            is HomeIntent.LoadAllCharacters -> HomeAction.AllCharacters
            is HomeIntent.ClearSearch -> HomeAction.AllCharacters
            is HomeIntent.SearchCharacter -> HomeAction.SearchCharacters(intent.name)
        }
    }

    //...

}
```

HomeViewModel.kt hosted with ❤️ by GitHub view raw

Then the **Action** is handled by the ViewModel, based on the result that is passed to the **Reducer** to define the new **State** that will be sent to the View.

```kotlin
fun Result<List<Persona>>.reduce(isSearchMode: Boolean = false): HomeState {
    return when (this) {
        is Result.Success -> if (isSearchMode) HomeState.ResultSearch(data) else HomeSta
        is Result.Error -> HomeState.Exception(exception)
        is Result.Loading -> HomeState.Loading
    }
}
```

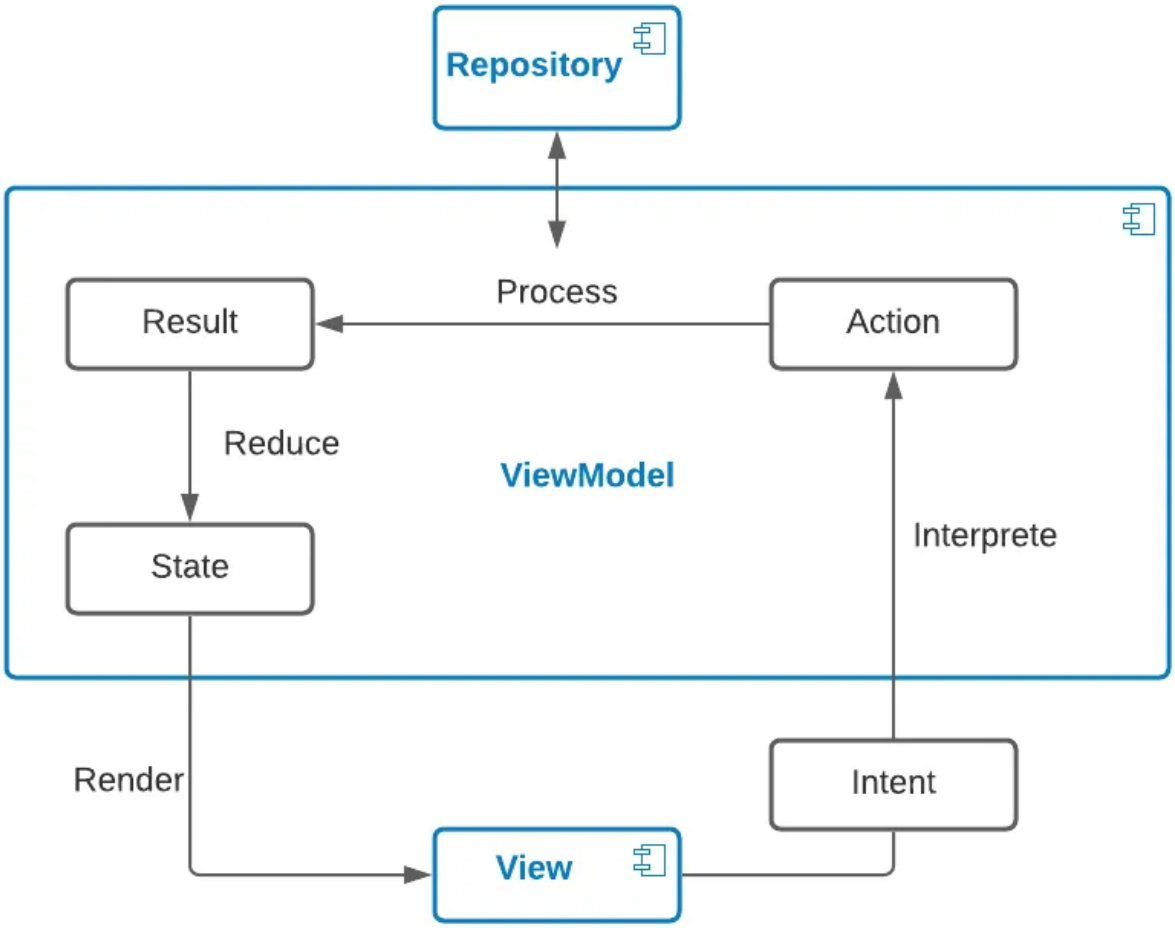
HomeReducers.kt hosted with ❤️ by GitHub view raw

```kotlin
class HomeViewModel @Inject constructor(private val dataManager: CharactersManager) :
    BaseViewModel<HomeIntent, HomeAction, HomeState>() {

    //...

    fun handleAction(action: HomeAction) {
        viewModelScope.launch {
            when (action) {
                is HomeAction.AllCharacters -> {
                    dataManager.getAllCharacters().collect {
                        mState.postValue(it.reduce())
                    }
                }
                is HomeAction.SearchCharacters -> {
                    dataManager.searchCharacters(action.name).collect {
                        mState.postValue(it.reduce(true))
                    }
                }
            }
        }
    }
}
```

HomeViewModel.kt hosted with ❤️ by GitHub view raw

For me, I implemented the reducer as an extension function that will be applied to the result returned by the business logic; the **result** will be mapped to its appropriate **State** by the **reducer,** then the View that **observes** the sate will eventually be updated according to it.

And finally we sum-up with this diagram that gives a clear overview of the circular and unidirectional data flow in MVI architecture:



MVI data flow

Wrapping up

MVI is basically your favorite architecture (MVVM or MVP) with state management where there is only one state for all the app layers, a single-source of truth. Although there is no only one best way to structure your code, it depends on the project, the team, or the client requirement, as long as it follows the principle of clean code with clear separation of concern and scalable codebase, but let's say that the MVI is a great architecture pattern that let you stick to those principles.

You can find the complete sample code about MVI clean architecture in my Github repo using Dagger2, Coroutines, and Android Jetpack components:

RimGazzeh/MVI-cleanArch

simple code for MVI architecture Dismiss GitHub is home to over 50 million developers working together to host and...

github.com