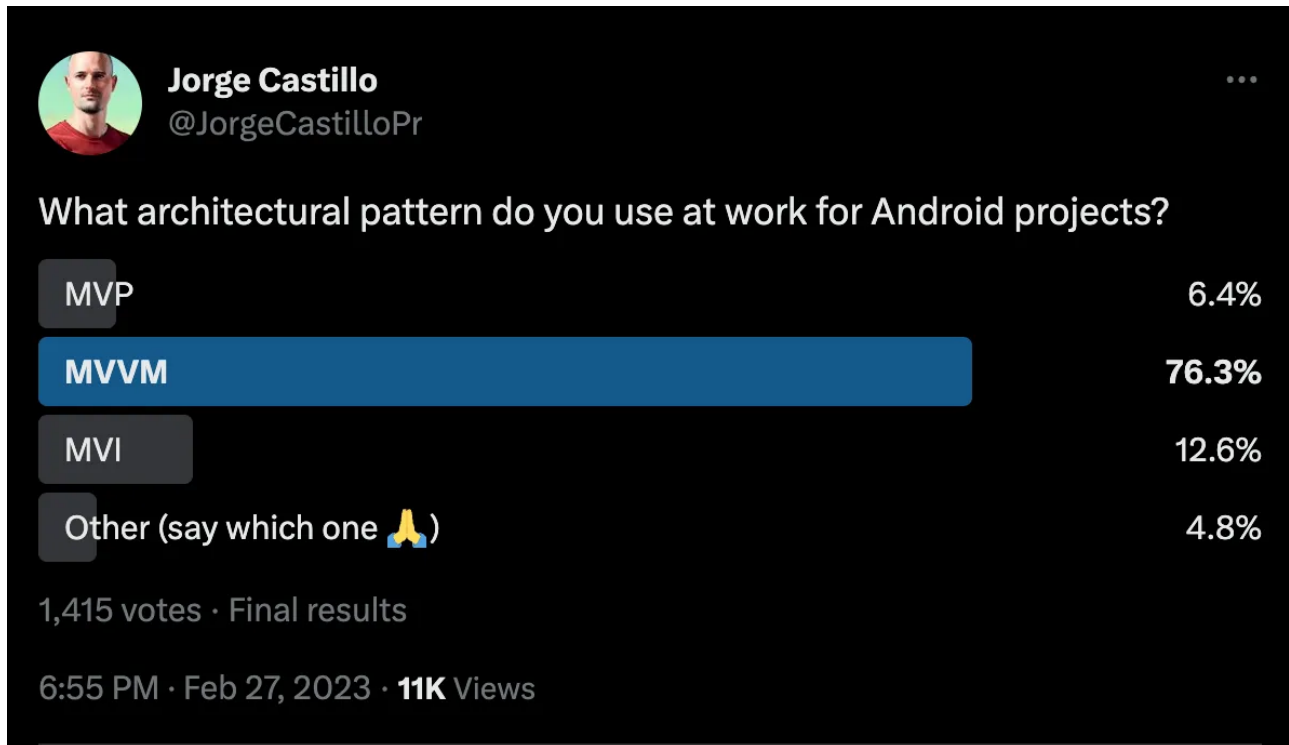


Using Jetpack Compose with MVVM

How to integrate Compose efficiently with an MVVM architecture



Jorge Castillo
Mar 9



I ran [this poll](#) on Twitter some weeks ago in order to know what architectural pattern people was more familiar with. For that reason, I decided to write a short post about **Compose + MVVM**.

First, the solution

If you came here looking for the actual code, here it is (extracted from [NowInAndroid](#)):

```
@Composable
fun BookmarksRoute(
    onTopicClick: (String) -> Unit,
    modifier: Modifier = Modifier,
    viewModel: BookmarksViewModel = hiltViewModel(),
) {
    val feedState by
    viewModel.feedUiState.collectAsStateWithLifecycle()
```

```

BookmarksScreen(
    feedState = feedState,
    removeFromBookmarks = viewModel::removeFromSavedResources,
    onTopicClick = onTopicClick,
    modifier = modifier,
)
}

```

```

class BookmarksViewModel @Inject constructor(
    private val repo: UserDataRepository,
    getSaveableNewsResources: GetUserNewsResourcesUseCase,
) : ViewModel() {

    val feedUiState: StateFlow<NewsFeedUiState> =
        getSaveableNewsResources()
            .filterNot { it.isEmpty() }
            .map { it.filter(UserNewsResource::isSaved) }
            .map(NewsFeedUiState::Success)
            .onStart { emit>Loading }
            .stateIn(
                scope = viewModelScope,
                started = SharingStarted.WhileSubscribed(5_000),
                initialValue = Loading,
            )

    fun removeFromSavedResources(newsResourceId: String) {
        viewModelScope.launch {
            repo.updateNewsResourceBookmark(newsResourceId,
false)
        }
    }
}

```

Benefits of Compose + MVVM

Using Compose in conjunction with a solid battle-tested architecture like MVVM (Model-View-ViewModel) can bring several benefits to the table, including:

- Separation of concerns: UI is separated from business logic, making business and data logic unit-testable in isolation. It also makes it easier to test UI, since we can feed it UI states and assert how the screen looks for each one.

- **Reactive UI:** We can easily integrate compose to work along with reactive data types that are state-of-the-art in the Android dev industry.
- **Familiarity:** developers are very familiar with MVVM at this point, which makes it very easy to onboard new members of the team and iterate quickly.

Collecting state in a lifecycle-aware manner

When collecting state in a Composable function, it's important to ensure that the act of collecting is lifecycle-aware. This means that collection is automatically started and stopped when the Composable is visible or hidden on the screen, respectively. This helps to avoid unnecessary resource usage in different cases, for example when the application is in the background.

For this, we can use `collectAsStateWithLifecycle`: an extension function for the `StateFlow` class. This function starts and stops collection when the host lifecycle owner goes above or below a specific lifecycle state (`Lifecycle.State.STARTED` by default). Read more about this topic [in this post by Manuel Vivo](#).

An example of it (from NowInAndroid):

```
@Composable
internal fun BookmarksRoute(
    onTopicClick: (String) → Unit,
    modifier: Modifier = Modifier,
    viewModel: BookmarksViewModel = hiltViewModel(),
) {
    val feedState by viewModel.feedUiState.collectAsStateWithLifecycle()

    BookmarksScreen(
        feedState = feedState,
        removeFromBookmarks = viewModel::removeFromSavedResources,
        onTopicClick = onTopicClick,
        modifier = modifier,
    )
}
```

The ViewModel side

Here is how the ViewModel could look in an average Android app:

```
class BookmarksViewModel @Inject constructor(
    private val repo: UserDataRepository,
    getSaveableNewsResources: GetUserNewsResourcesUseCase,
) : ViewModel() {

    val feedUiState: StateFlow<NewsFeedUiState> =
        getSaveableNewsResources()
            .filterNot { it.isEmpty() }
            .map { it.filter(UserNewsResource::isSaved) }
            .map(NewsFeedUiState::Success)
            .onStart { emit>Loading }
            .stateIn(
                scope = viewModelScope,
                started = SharingStarted.WhileSubscribed(5_000),
                initialValue = Loading,
            )

    fun removeFromSavedResources(newsResourceId: String) {
        viewModelScope.launch {
            repo.updateNewsResourceBookmark(newsResourceId, false)
        }
    }
}
```

`StateFlow` is used to expose the state that the UI collects. We want to read any data Flows from repositories (or use cases) and use the Flow operators to transform the data based on our needs. In case of having multiple Flows, we can combine them into a single one [using the Flow operators](#).

The goal is to reduce all sources of data into a single flow that can be converted into a **hot** flow with the `stateIn` operator.

Merging all flows into one works well when we represent our entire screen with a single UI state class. That is usually recommended to have a single source of truth for UI state that the UI can collect.

The `stateIn` operator uses the provided `viewModelScope` to automatically cancel the flow when the `ViewModel` is cleared. This helps to avoid leaking any background tasks that could still be emitting from our data layer after that.

The `WhileSubscribed(5_000)` is a somewhat controversial trick used to avoid the flow getting canceled during some configuration changes like rotations (they usually take less than 5 secs). `WhileSubscribed` implies that sharing of the `StateFlow` starts when the first subscriber appears, and immediately stops when the last subscriber disappears (by default), keeping the replay cache forever (by default).

Finally, the initial UI state to be emitted is passed (`Loading`).

And this is pretty much all we need to do. We can reduce almost any average Android app to this approach.

Unidirectional data flow

Any architectures or architectural patterns that expose observable state to the UI can use UDF (unidirectional data flow). The way to do this with MVVM is simple:

- Make any actions to read data from repositories return `Flow`.
- Make `ViewModels` or use cases always read from the database as the single source of truth, so any changes on the application data are reduced and transformed into UI state via `Flow` operators.
- Model one-off actions performed on the data layer (like triggering a network request) as `suspend` functions (e.g: `removeFromSavedResources` in the previous snippet).

This allows data to flow in a single direction, achieving the following cycle 

1. The user clicks a button
2. UI notifies the `ViewModel` about the user interaction
3. `ViewModel` launches a coroutine to call a `suspend` function (using a repository or a use case) to trigger a network request (for example)
4. When the network request completes, data is stored in the database
5. Since `ViewModel` is observing changes in the database by only reading `Flows` from it, it will get fresh data emitted

6. ViewModel can transform and reduce the data as required into UI state
7. UI collects and draws the new state.


Conclusion

The declarative nature of Compose makes it a perfect match for any architectural patterns that expose observable UI state as the source of truth for UI. Compose can leverage that by efficiently diffing the already created UI tree every time a new UI state is emitted, so not all the tree needs to be redrawn. This is a big benefit compared to Android Views.


Master Jetpack Compose with me 🚀

Join me and several other Android devs in the Jetpack Compose and internals online training. [Register for a reduced price now.](#)

Take the online course and join the exclusive community



Master Jetpack Compose and learn how to work efficiently with it by taking this course. Enjoy the perfect mix of theory and exercises with the best trainer ★★★★★




Created and delivered by Jorge Castillo, Google Developer Expert for Android and Kotlin

[Tell me more 🚀](#)

Opinions from previous attendees

Taken by engineers from



Join the Effective Android Discord community

Join the exclusive community with 100+ developers where we discuss about Jetpack Compose in depth and any other Android topics. Use it to make important career connections. **Yearly paid subscriptions** to this newsletter give you full access to it. I'll be waiting for you there! 🙌

<input type="text" value="Type your email..."/>	Subscribe
---	---------------------------

Comments



Write a comment...