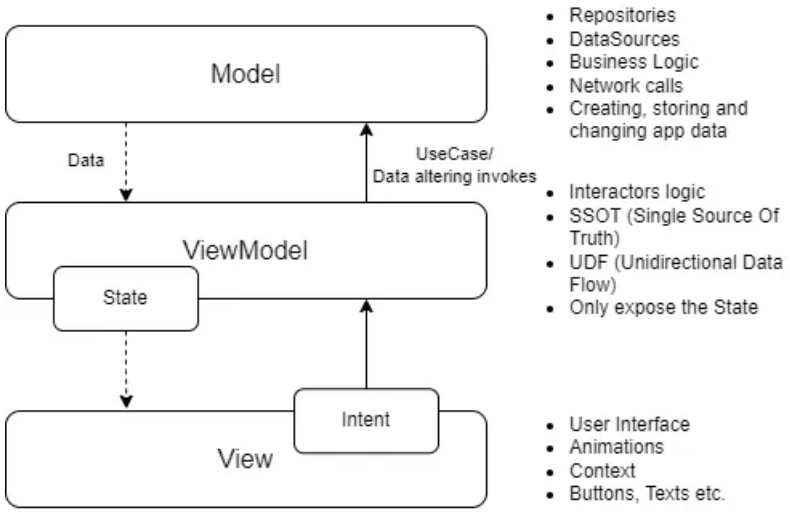
MVI Architecture Explained On Android

Michal Ankiersztajn · Follow
Published in Stackademic · 4 min read · Jan 10

What is MVI?

You can implement **MVI** using **ViewModel** or **Presenter**, but on *Android*, we're accustomed to using **ViewModel**, so it's what I'm going to show You:



MVI Architecture Diagram

M — Model

- Data-altering operations (networking, local storage, etc.)
- Business logic (UseCases, Clean Model classes)

V — View

- It's everything the user can see
- Observes the **State** exposed by the **ViewModel**
- Sends **Intents** to **ViewModel** through **lambda function**

- It doesn't need to reference **ViewModel**! **ViewModel** creation will usually be placed in something responsible for showing the screen, like Navigation and from there, it'll pass the **State** and **Intent** handling to the **View**.

I — Intent

- Represents **View Intentions**. It doesn't have to be a **User Intention** (but it usually is)
- Used to communicate with **ViewModel**

This pattern is very similar to **MVVM**, the main difference is you communicate with **ViewModel** through **Intents** instead of *invoking methods directly*. This means your **View** doesn't need to know anything about the object handling his **Intentions**.

Create Example App

Let's build a simplified Message sending screen to understand everything in practice!

What will you build?

Example app using MVI

For those that want to jump straight into completed code, check this MVIExample project:

1. Create the State

The State needs to represent:

- Typed in text
- Whether we're still sending the message
- The result of sending the operation

```
data class ExampleState(
    val text: String = "",
    val isSending: Boolean = false,
    val message: String? = null,
)
```

2. Create the Events (Intents)

In Android, we call it **Event** because **Intent** stands for something different, and we want to avoid problems with naming things.

The screen we'll create allows users to:

- Change the message
- Send the message

But we also show a success message that needs to be cleared after it's delivered, so our Events class will look like this:

```
sealed class ExampleEvent {
    data class TextChanged(val text: String) : ExampleEvent()
    data object SendClicked : ExampleEvent()
    data object MessageShown : ExampleEvent()
}
```

Note that it needs to be a sealed class so every Event is stored in one file and can differ in its arguments.

3. Create the Screen

One of the main advantages of this is that Screens are independent of your architecture. All they know is the State and events.

```
@Composable
fun ExampleScreen(
    state: ExampleState,
    sendEvent: (ExampleEvent) -> Unit,
) {
    val context = LocalContext.current
    LaunchedEffect(state.result) {
        if (state.result != null) {
            Toast.makeText(context, state.result, Toast.LENGTH_SHORT).show()
            sendEvent(ExampleEvent.MessageShown)
        }
    }
    Column(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center,
    ) {
        if (state.isSending) {
            CircularProgressIndicator()
        } else {
            OutlinedTextField(
                value = state.text,
                onValueChange = { text ->
                    sendEvent(ExampleEvent.TextChanged(text))
                },
                label = { Text(text = "Message") }
            )

            Button(onClick = { sendEvent(ExampleEvent.SendClicked) }) {
                Text(text = "SEND")
            }
        }
    }
}
```

4. Create the ViewModel

ViewModel works best in this scenario because it *exposes the State* to a **View**.

- It can only have a single public method called **handleEvent** that takes in the **Event.**
- All handling functions are private and are *products of* ***Events***
- It exposes a single-state

```
class ExampleViewModel(
    private val repository: ExampleRepository = ExampleRepository(),
) : ViewModel() {
```

```
    private val _state = MutableStateFlow(ExampleState())
    val state = _state.asStateFlow()

    fun handleEvent(event: ExampleEvent) {
        when (event) {
            ExampleEvent.SendClicked -> sendClicked()
            is ExampleEvent.TextChanged -> textChanged(event)
            ExampleEvent.MessageShown -> messageShown()
        }
    }

    private fun sendClicked() {
        _state.update { state -> state.copy(isSending = true) }
        viewModelScope.launch {
            repository.sendMessage(state.value.text)
            _state.update { state ->
                state.copy(
                    text = "",
                    isSending = false,
                    result = "Message sent successfully",
                )
            }
        }
    }

    private fun textChanged(data: ExampleEvent.TextChanged) {
        _state.update { state -> state.copy(text = data.text) }
    }

    private fun messageShown() {
        _state.update { state -> state.copy(result = null) }
    }
}
```

If you had some trouble setting it all up, check how it works in my github repo:

GitHub - AndroBrain/MVIExample

github.com

In conclusion:

Advantages:

- The **View** depends on **Intents** instead of **concrete objects** that handle them.
- It's easy to swap implementations that handle **Intents**.
- It's easy to build abstractions on top of **Intents**. You can replace state classes with interfaces and add mappers for **Intents** to build more generic **ViewModels** that can handle multiple **Intents** and expose generic **State**. Be careful here, as it may lead to bloated and unreadable **ViewModels**.

- Works well with **Compose** because the **View** doesn't need to reference the **Intent handler**.

Disadvantages:

- You've to write a lot of **boilerplate code.** You'll have to repeat yourself a lot. You need to define **Intents** and then write a function for each. You can write more generic functions to handle multiple events, but it rarely happens.
- **ViewModel** often ends up being tightly coupled to single **View Intents**.
- When used with **XML Layouts,** it's hard not to reference **ViewModel** that handles **Intents**, which makes the whole pattern a bit useless since our View depends on **ViewModel** anyway. We could make the intent handling functions public and skip creating the **Event** class. Hence, I recommend using **MVVM** instead of **MVI** for XML Layouts.

If you're interested in the MVVM pattern, please check out my previous

Many Android Developers don't know what MVVM is or how to use it. Learn how to use ViewModels properly on Android and...

medium.com

If you found this guide valuable, it would help others find this article if you click the 👏 icon and follow me for more! Thank You 🙏

Stackademic

Thank you for reading until the end. Before you go:

- Please consider **clapping** and **following** the writer! 👏
- Follow us X | **LinkedIn** | **YouTube** | **Discord**
- Visit our other platforms: **In Plain English** | **CoFeed** | **Venture**

Mvi Kotlin Android App Development AndroidDev Mvvm