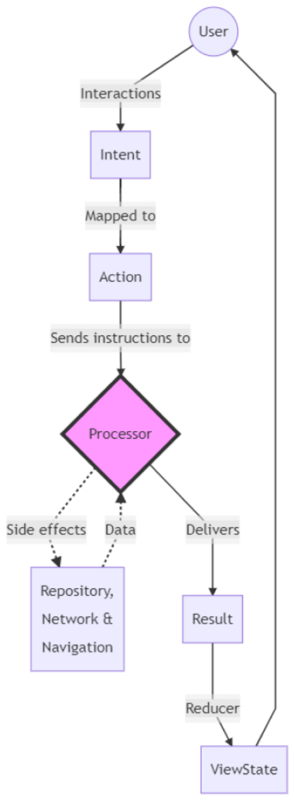
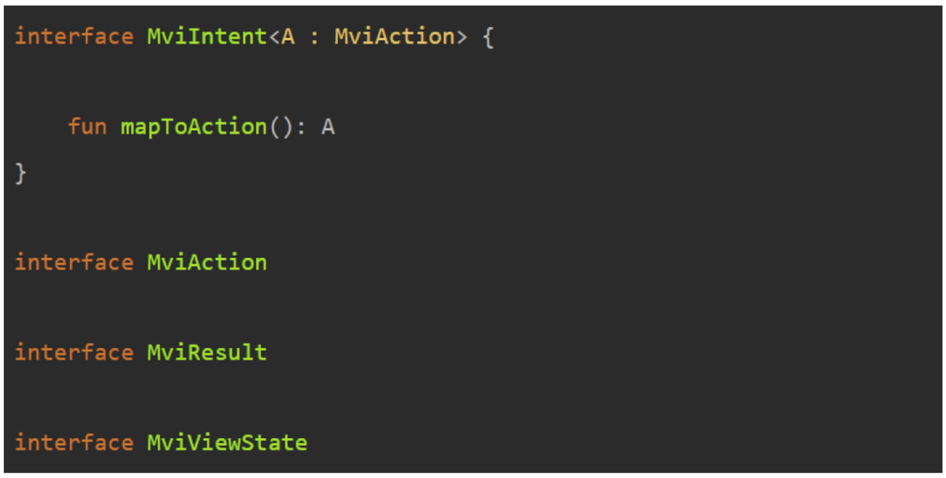
MVI - The Good, the Bad, and the Ugly

A Gentle Introduction

Fundamentally, MVI imagines the user as a function, and models a unidirectional data flow based on that. The user creates new events at the top of the loop, which are interpreted through various steps until rendered in the UI. This lends itself very well to modern functional reactive programming techniques, and attempts to keep side effects to a minimum and only in one place.

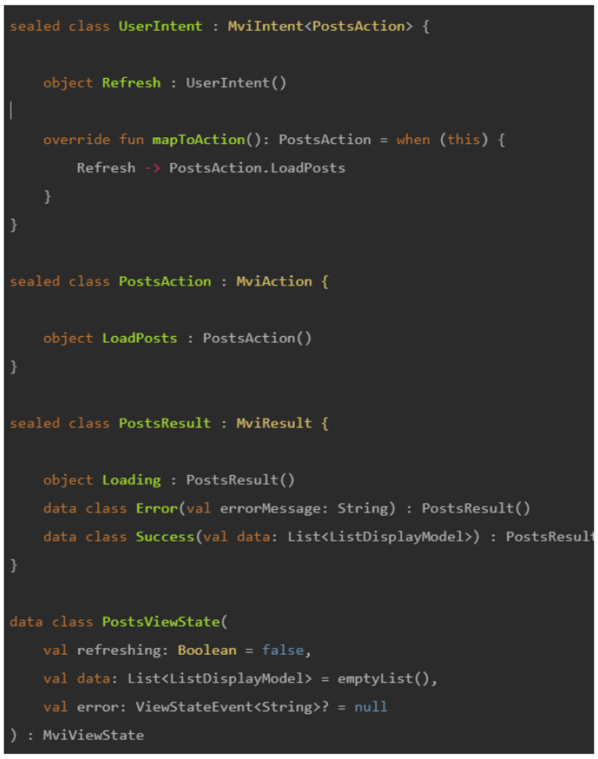


To model this data flow fully, we'll use 4 marker interfaces: Intent, Action, Result and ViewState:



```
interface MviIntent<A : MviAction> {

    fun mapToAction(): A
}

interface MviAction

interface MviResult

interface MviViewState
```

These are then subclassed by sealed classes, except for ViewState which is subclassed by a data class. It’s this modelling at every step of the data flow which is so beneficial - it forces you to think about every possible input to the model and how it affects other parts of the system. Here’s a contrived example for a view that allows users to load a list of posts:

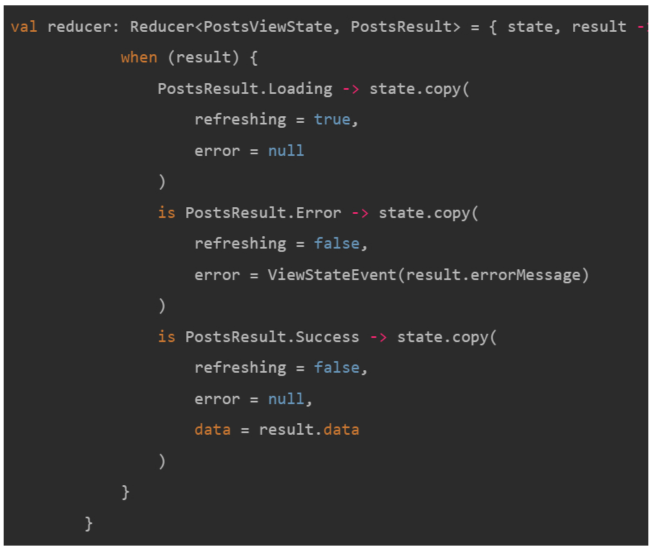


```
sealed class UserIntent : MviIntent<PostsAction> {

    object Refresh : UserIntent()

    override fun mapToAction(): PostsAction = when (this) {
        Refresh -> PostsAction.LoadPosts
    }
}

sealed class PostsAction : MviAction {

    object LoadPosts : PostsAction()
}

sealed class PostsResult : MviResult {

    object Loading : PostsResult()
    data class Error(val errorMessage: String) : PostsResult()
    data class Success(val data: List<ListDisplayModel>) : PostsResul
}

data class PostsViewState(
    val refreshing: Boolean = false,
    val data: List<ListDisplayModel> = emptyList(),
    val error: ViewStateEvent<String>? = null
) : MviViewState
```

`Intent` objects are mapped to an `Action`, and there may well be some crossover between the two. For a very contrived example, a `UserClickedRefresh` type `Intent` may very well map to the same kind of `Action` as `UserClickedRetry`. The `Processor` which handles the `Action` objects doesn't care what the user intended: it only cares about what it needs to do in response - in this case load the Posts. You could very easily argue that `Action` classes are somewhat redundant, but these are important (in my opinion) to help separate the view from the underlying logic.

These `Action` objects are then sent to a `Processor`, and this is where the meat of the work happens. Within this class, `Action` objects are filtered by type and then split into individual streams where some data may be fetched from the network, or perhaps some navigation occurs. Once the data is fetched, this is then returned as a `Result` type, and all of these streams are merged into a single Observable which is fed back into the `Model`.

It's at this point that the `Result` is reduced: if a `Result.Loading` object is returned, you might update the current ViewState by copying it and setting `loading` to `true`. If your result returns some data, you might set `loading` to false and copy the data into the `ViewState`. Here, we're making use of Kotlin's `copy` constructor for data classes.

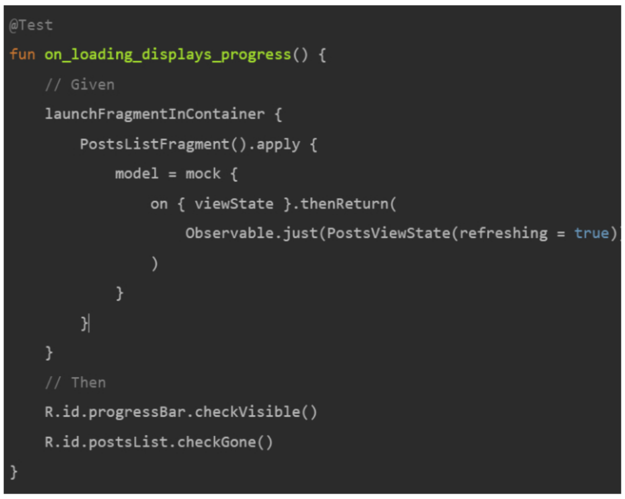


```
val reducer: Reducer<PostsViewState, PostsResult> = { state, result -
            when (result) {
                PostsResult.Loading -> state.copy(
                    refreshing = true,
                    error = null
                )
                is PostsResult.Error -> state.copy(
                    refreshing = false,
                    error = ViewStateEvent(result.errorMessage)
                )
                is PostsResult.Success -> state.copy(
                    refreshing = false,
                    error = null,
                    data = result.data
                )
            }
        }
```

Finally, this new ViewState is sent to the View, where it's up to the Activity or Fragment to render this information.

The Good

This architecture really forces you to think about how you model the users interactions and the possible states of the View, and this can be highly beneficial. In working out which Intent objects are required, I often realise that I'm missing some key piece somewhere and that alone can be really valuable. I'm sure people feel the same about MVVM or MVP, but in my experience I find that MVI has the strongest "planning" effect on me. This may be because it's so critical to model failure states in MVI otherwise the stream gets broken - you can't just log an error in onError and be done with it.



```
@Test
fun on_loading_displays_progress() {
    // Given
    launchFragmentInContainer {
        PostsListFragment().apply {
            model = mock {
                on { viewState }.thenReturn(
                    Observable.just(PostsViewState(refreshing = true)
                )
            }
        }
    }
    // Then
    R.id.progressBar.checkVisible()
    R.id.postsList.checkGone()
}
```

The Bad and the Ugly