Architecting Android...Reloaded

After a long time I decided to write again about **Architecture on Android Applications**. The reason? Mainly **feedaback** from the community and **lessons learned**. But even though a lot has been said since the early days when **Clean Architecture** became popular in Mobile Development, there is always room for improvement and evolution.

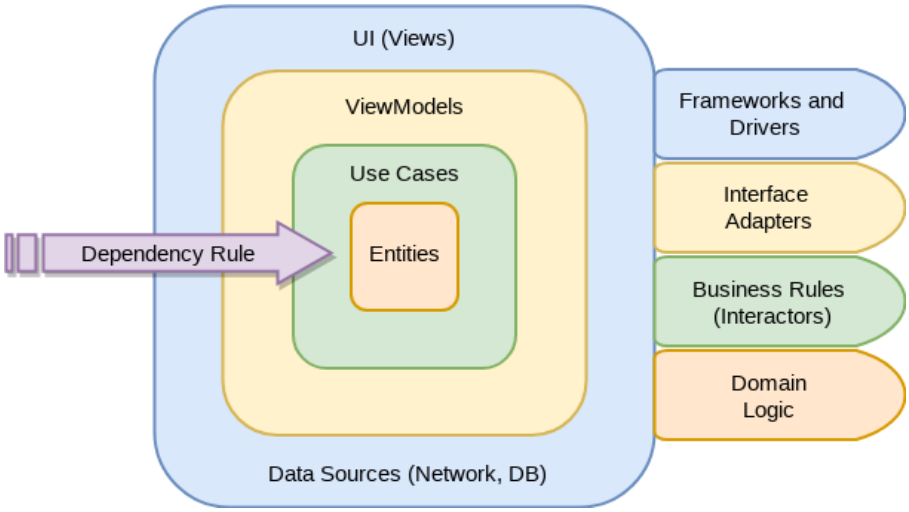
In order to get started and get things easier, I will assume you have already read these old but still valid blog posts:

- Architecting Android…The clean way?
- Architecting Android…The evolution.

Based on the above articles clean architecture example, there is a clear evolution in the codebase, especially because nowadays with applications being key at a business level, more than ever, there is a need to **scale**, **modularize** and **organize teams** around Mobile Development (mainly due to its complexity).

Thus, the idea is to come up with an (elegant?) solution which will make our life easier in terms of:

- **Problem solving.**
- **Scalability.**
- **Modularization.**
- **Testability.**
- **Independence of frameworks, UI and Databases.**



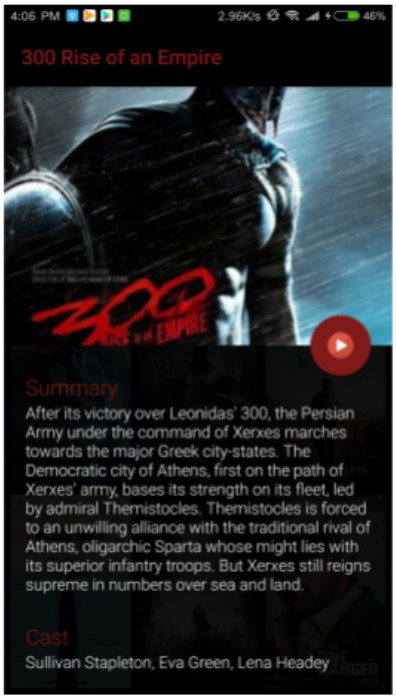
This is the big picture, which should look familiar if you have been using **Clean Architecture** in your Android Applications.

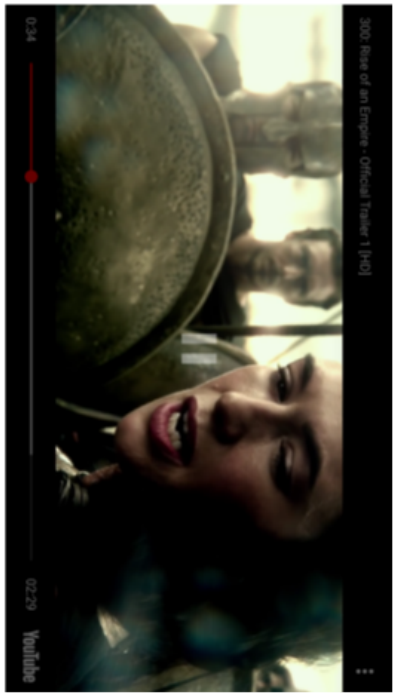
Our scenario

A simple Movies Android Application (any similarities with reality is a mere coincidence).

Written in **Kotlin**: not much to say other than that we want to leverage a modern language's features like **inmutability, conciseness, functional programming,** etc.

With the following flow:





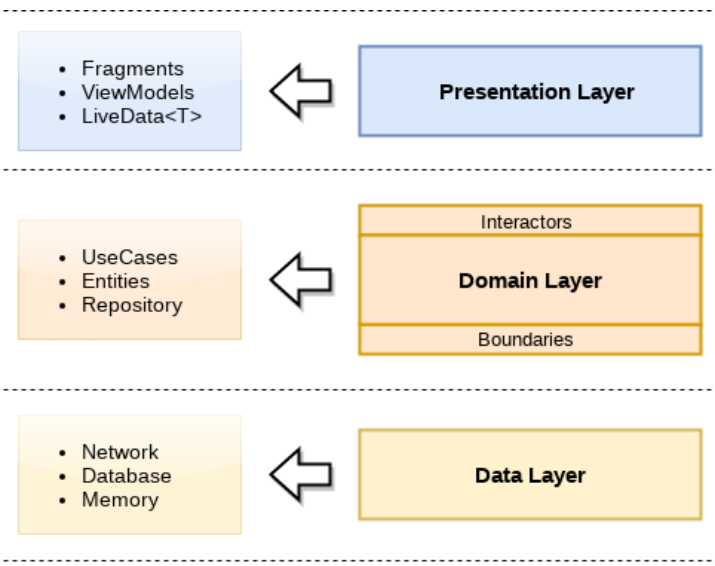
Where we have 3 main use cases:

- Get a list of movies.
- Show details for an specific clicked movie.
- Play a movie.

As usual, the source code is available on github.

General Architecture

The general principle is to use a basic **3 tiers architecture**. The good thing about it, is that it is very easy to understand and many people are familir with it. So we will break down our solution into layers in order to respect the dependency rule (where dependencies flow in one direction: check above the rounded clean architecture graph):



Nothing new here if we keep in mind my previous posts, but do not stop reading yet. Let's dive deeper and go piece by piece for a better understanding.

Domain Layer: Functional Use Cases

A use case is an intention, in other words, something we want to do in our application, one of our main players. **And its main responsibility is to orchestrate our domain logic and its connection with both UI and Data layers.**

By using the power of Kotlin and its treatment of functions as first class citizens (more coming up shortly), we have in our framework a UseCase abstraction, which acts as a contract for all the use cases in our application.

```
abstract class UseCase<out Type, in Params> where Type : Any {

    abstract suspend fun run(params: Params): Either<Failure, Type>

    fun execute(onResult: (Either<Failure, Type>) -> Unit, params: Params) {
        val job = async(CommonPool) { run(params) }
        launch(UI) { onResult.invoke(job.await()) }
    }
}
```

What is going on here?

We have an abstract class which takes 2 generic parameters:

- `<out Type>`: a return type which is the result of the executed use case.
- `<in Params>`: a parameters class which will be consumed inside the run() function in case we need extra data for our use case.

The **execute()** function is where all the magic happens:

- We pass a "**onResult**" function as a parameter which takes an **Either<Failure, Type>** and returns **Unit** (in the Error Handling section I will extend the explanation for **Either<L, R>**, so be patient please :)). The good thing is that the caller of the UseCase is actually establishing the desired behaviour by passing this inmutable function (onResult), thus, avoiding any internal exposure or side effects if we were passing objects (one of the benefits of FP, more coming up).
- Also, by using **Kotlin coroutines** we invoke the passed "onResult" function in a **different thread,** so from this point on, we are safe to write our code in a synchronous fashion. The result will be posted on the **Android Main UI Thread.**

`abstract suspend fun run(params: Params)` is what we have to override when extending the UseCase<out Type, in Params> abstraction, so for instance, this is how our GetMovies use case looks like:

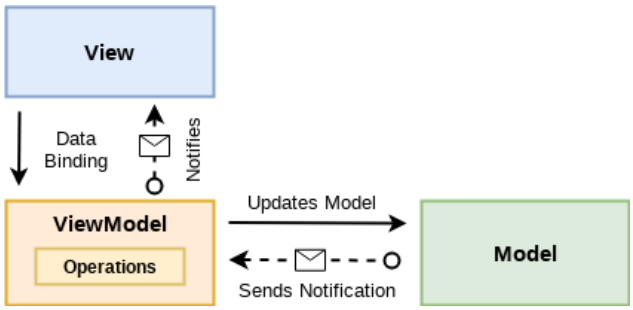
```
class GetMovies
@Inject constructor(private val moviesRepository: MoviesRepository) :
    UseCase<List<Movie>, None>() {

    override suspend fun run(params: None) = moviesRepository.movies()
}
```

In this example we are delegating movies retrieval to a **Repository.** Easy right?

UI Layer: From MVP to MVVM

The **Model-View-ViewModel** Pattern (MVVM) provides a clean separation of concerns between user interface and domain logic.

It has 3 main components: the **model**, the **view**, and the **view model**. There are relationships between them, although each serves a distinct and separate role:



At the highest level, the view "knows about" the view model, and the view model "knows about" the model, but the model is unaware of the view model, and the view model is unaware of the view. **The view model isolates the view from the model classes and allows the model to evolve independently of the view.**

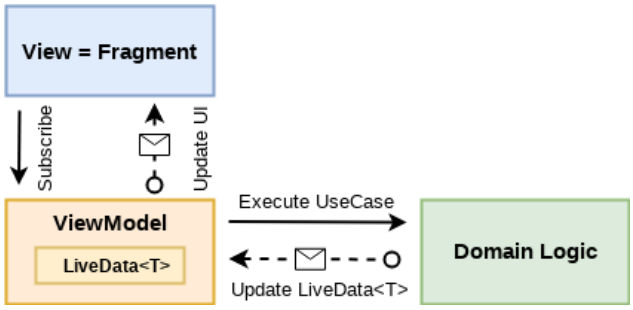
At implementation level, in our example, **MVVM** is accomplished by the usage of Architecture Components, which its main advantage is to handle configuration changes when the screen rotates, something that has given us many headaches as android developers (I guess you know what I'm talking about).

Disclaimer: that does not mean we have to no longer care about lifecycles, but it is way easier.

A comment on MVP (Model View Presenter) from previous example: I found tricky to avoid leaks due to activities and fragments being recreated so I used a **poor man solution: retain fragments.**

However, I ran into these sort of situations anyway. This is the reason why I decided to go and give a try to MVVM.

Let's see what changed with MVVM from previous sample and how it works:



- **Fragments act as views**, where all the logic related to displaying data on the screen happens.
- **Fragments also know about ViewModels**, they actually subscribe to ViewModels.
- **ViewModels contain LiveData** objects and references to UseCases.
- **UseCases update LiveData** which react to those changes and send notifications to ViewModels.
- **ViewModels talk to subscribed Fragments** in order to update the UI.

In order to see all these pieces working together, let's see some code.

ViewModel containing **LiveData** and updating by calling **UseCase.execute()** function:

```
class MoviesViewModel
@Inject constructor(private val getMovies: GetMovies) : BaseViewModel() {

    var movies: MutableLiveData<List<MovieView>> = MutableLiveData()

    fun loadMovies() =
        getMovies.execute({ it.either(::handleFailure, ::handleMovieList) }, None())

    private fun handleMovieList(movies: List<Movie>) {
        this.movies.value = movies.map { MovieView(it.id, it.poster) }
    }
}
```

Fragment subscribing to the above **ViewModel** on **onCreate()**.

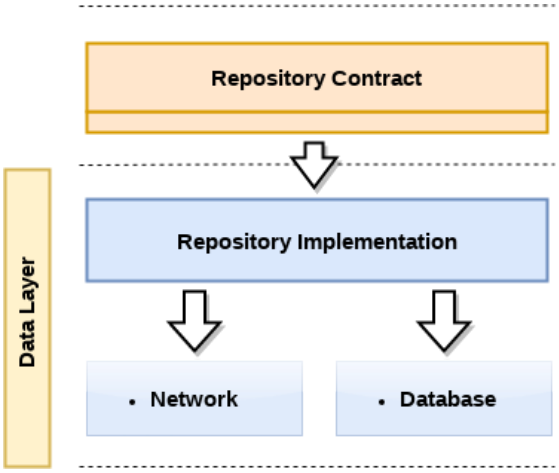
I used a couple of tricks with extension functions to get rid of some verbosity and boilerplate.

```
class MoviesFragment : BaseFragment() {

    @Inject lateinit var navigator: Navigator
    @Inject lateinit var moviesAdapter: MoviesAdapter

    private lateinit var moviesViewModel: MoviesViewModel

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

        //subscribtion to LiveData in MoviesViewModel
        moviesViewModel = viewModel(viewModelFactory) {
            observe(movies, ::renderMoviesList)
            failure(failure, ::handleFailure)
        }
    }
        ...
}
```

Data Layer: Repository Pattern to the rescue

Anything new here in comparison with the previous example since I have had really good results with it.

To keep in mind: **At its core the Repository pattern is a simple interface.** It exists as a layer between our domain and our data so that our logic doesn't need to be concerned with the implementation of different data sources: Network, Database or Memory.



In the following chunk of code we can see our MoviesRepositoy contract:

```
interface MoviesRepository {
    fun movies(): Either<Failure, List<Movie>>
    fun movieDetails(movieId: Int): Either<Failure, MovieDetails>
}
```

In our example we usually inject a Repositoy as a collaborator for our **UseCases** implementations.

Functional Error Handling

Overall **Error/Exception Handling** should be taken care at design and not at implementention level, and in my opinion, one of the biggest mistakes we make as developers (lesson learned). That is why it is important to have a framework in place for this purpose.

What happens with traditional Error Handling?

Observing exceptions (try/catch blocks) and making decisions based on it that changes the control flow is a bad practice: it creates unprediction, affects our resilience and debugging becomes difficult, especially in concurrent environments. Plus going back to C-style error handling, using error codes which need to be checked by convention could be a nightmare.

With that being said, we have seen we are using **Either<L, R>** as a return type in our UseCase abstraction:

```
abstract suspend fun run(params: Params): Either<Failure, Type>
```

So let me introduce Either

Either<L, R> is referred as a disjoint function, which means that this structure is designed to hold either a **Left<T>** or **Right<T>** value but never both. It is a **funcional programming monadic type** not yet existent in the Kotlin Standard Library.

Here is a simple implementation which perfectly fulfills my needs and it is easy to understand and use (ideas taken from Alex Hart):

```
/**
 * Represents a value of one of two possible types (a disjoint union).
 * Instances of [Either] are either an instance of [Left] or [Right].
 * FP Convention dictates that:
 *      [Left] is used for "failure".
 *      [Right] is used for "success".
 *
 * @see Left
 * @see Right
 */
sealed class Either<out L, out R> {
    /** * Represents the left side of [Either] class which by convention is a "Failure". */
    data class Left<out L>(val a: L) : Either<L, Nothing>()
    /** * Represents the right side of [Either] class which by convention is a "Success". */
    data class Right<out R>(val b: R) : Either<Nothing, R>()

    val isRight get() = this is Right<R>

    val isLeft get() = this is Left<L>

    fun either(fnL: (L) -> Any, fnR: (R) -> Any): Any =
            when (this) {
                is Either.Left -> fnL(a)
                is Either.Right -> fnR(b)
            }

    fun <T> flatMap(fn: (R) -> Either<L, T>): Either<L, T> {...}
    fun <T> map(fn: (R) -> (T)): Either<L, T> {...}
}
```

Let me also quote Daniel Westheide (a Scala guru and good person, whom I bumped into at SoundCloud) in one of his amazing blog posts:

> *There is nothing in the semantics of the **Either<L, R>** type that specifies one or the other sub type to represent an error or a success, respectively. In fact, **Either** is a general-purpose type for use whenever you need to deal with situations where the result can be of one of two possible types.*

> *Nevertheless, error handling is a popular use case for it, and by convention, when using it that way, the **Left<T>**represents the error case, whereas the **Right<T>** contains the success value.*

And please do not forget to read his entire serie of scala posts to expand your horizons even more (getting ideas from other languages is always a +1):

- Error Handling With Try.
- The Either Type.

What about our code sample then?

In the GetMovies use case, at implementation level, we always return an **Either<Failure, List<Movie>>** up starting from the data layer, all the way up to our **MoviesViewModel** which updates “either” the failure **LiveData<Failure>** (in case of failure, Left<T>) or the movies **LiveData<List<MovieView>>**(success, Right<T>):

```
class MoviesViewModel
@Inject constructor(private val getMovies: GetMovies) {

    var movies: MutableLiveData<List<MovieView>> = MutableLiveData()
    var failure: MutableLiveData<Failure> = MutableLiveData()

    fun loadMovies() =
        getMovies.execute({ it.either(::handleFailure, ::handleMovieList) }, None())

    private fun handleMovieList(movies: List<Movie>) {
        this.movies.value = movies.map { MovieView(it.id, it.poster) }
    }

    private fun handleFailure(failure: Failure) {
        this.failure.value = failure
    }
}
```

At a View level (our MoviesFragment), we subscribe to the updates coming from the view model:

```
moviesViewModel = viewModel(viewModelFactory) {
    observe(movies, ::renderMoviesList)
    failure(failure, ::handleFailure)
}
```

And this is how **handleFailure()** looks like for Failure treatment:

```
private fun handleFailure(failure: Failure?) {
    when (failure) {
        is NetworkConnection -> renderFailure(R.string.failure_network_connection)
        is ServerError -> renderFailure(R.string.failure_server_error)
        is ListNotAvailable -> renderFailure(R.string.failure_movies_list_unavailable)
    }
}
```

By the way, Failure is a sealed class which offers global default Failures:

```
/**
 * Base Class for handling errors/failures/exceptions.
 * Every feature specific failure should extend [FeatureFailure] class.
 */
sealed class Failure {
    class NetworkConnection: Failure()
    class ServerError: Failure()

    /** * Extend this class for feature specific failures.*/
    abstract class FeatureFailure: Failure()
}
```

I hope now the usage of **Either<L, R>** is clearer and you understand the reason and benefits of this applied technique.

First steps to Modularization

First, in my defense, this post is NOT about this specific topic (which by the way, is huge) but I want to write down some hints based in past experiences, in order to get started. From my perspective, sooner or later, this is the way to go, and **a good architecture should help out to achieve this goal.**

What is Modularization?

Modularization is the process of separating and creating clear boundaries between logical components of code.

If you did your homework and checked my previous posts about clean architecture, you might have seen that I used android modules for representing each layer involved in the architecture.

A recurring question in discussions was: Why? The answer is simple… Wrong technical decision: I relied on different modules in order to be more strict with the dependency rule by establishing borders and, thus, make it more difficult to break it.

But power comes with big responsibilities, and even though this worked pretty well in the beginning, the sample was still a **MONOLITH** and it would bring problems when scaling up:

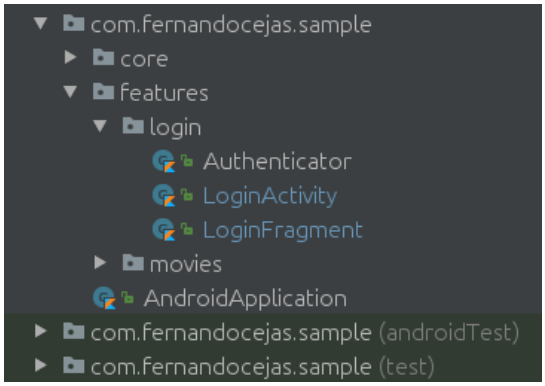
- when modifying or adding a new functionality: we had to touch every single module/layer (strong dependencies/coupling between modules).
- conflicts when developers working in the codebase (the bigger the team, the more conflicts, especially with PRs and git).

Embrace Application Modularization

My first tip to favor modularization is to organize packages by features, this way we accomplish:

- **Higher Modularity.**
- **High Cohesion.**
- **Easier Code Navigation.**
- **Minimizes Scope.**
- **Isolation and Encapsulation.**

Code/package organization is one of the key factors of a good architecture: package structure is the very first thing encountered by a programmer when browsing source code. Everything flows from it. Everything depends on it.



My second tip is to have a core module which will have these main responsitilities:

- **Handle global dependency injection.**
- **Contain extension functions.**
- **Contain the main framework abstractions.**

- **Initiate in the main application common 3rd party libraries like Analytics, Crash Reporting, etc.**

My third tip is not at codebase level, but it might be helpful to add code ownership if we are working with feature teams, which is a win in growing organizations where many developers are working on the same codebase.

These are the main benefits of **Modularization**:

- **Faster Build Time.**
- **Package cohesion.**
- **Re-usability of common functionality.**
- **Conflicts reduction (especially when working with git flows).**
- **Feature encapsulation.**
- **More controlled dependendencies.**
- **Team work: collaboration between teams.**

I know all this sounds good on paper and although modularizing your android codebase is tricky and challenging (due to all the moving parts involved), the advantages are huge.

Other Implementation Details

- RxJava: This one was one of the biggest changes in comparison with the previous example. I got rid of RxJava because I do not need it here. By the way, I bumped into many people ONLY using it because of threading. Of course it facilitates our life in that aspect but brings overhead and complexity in other areas. So make sure threading is not the only reason to introduce it in your codebase.
- Dependency Injection: Simple one using Dagger 2, and it is out of scope of this article. I wrote about it in the past.
- Unit and integration tests are in place.
- Acceptance Tests: Still TODO (PRs are more than welcome).

Source code and discussions

Source code can be found here:

- https://github.com/android10/Android-CleanArchitecture-Kotlin

For discussions, open a new issue on Github, so we can continue the conversation over there.

Conclusion

We have seen the theoretical approach and implementation details. There is much more to explore but that is homework for you :).

Also, it does not matter which architecture you pick for your projects as soon as it fulfills your needs and solves your problems.

Keep in mind that there are NO silver bullets and of course there is always room for improvement, although this should be useful as a starting point.

And some advice:

- **Do not over think (do not do over engineering).**
- **Be pragmatic.**
- **Minimize framework (android) dependencies in your project as much as you can.**
- **Continuous improvement through refactor.**
- **Do not write code without tests (I should not be saying this in 2018 :)).**

Something else to add since I have seen discussions around FP vs OOP: Functional Programming is not new and has been there for a long time but nowadays is gaining more and more adopters. And the fact that Kotlin treats functions as first class citizens give us even more power and tools to solve our problems.

Use whatever works for you, here **I'm conbining what I consider the best of both worlds…**