Marvel App

# Overview: Scope of the work

Original requirements of the projects:

* Get the character provided by the Marvel Api: /v1/public/characters
* Get the character object provided by the Marvel Api: /v1/public/characters{charactersId}
* Show the list and navigate to a selected character( 2 screens)

**Extra work delivered**:

* Additional Marvel endpoints supported:
  + Get the comics provided by the Marvel Api: /v1/public/comics
  + Get the comic object provided by the Marvel Api: /v1/public/comics{comicsId}
* 2 screens for comics and comicDetail presentation
* Onboaring Screen. It’d a very common element of an App.It is purpose is to show initial information/ a quick tutorial. This is meant to be shown only once if the user decides so ticking the “dont’ show again” box”

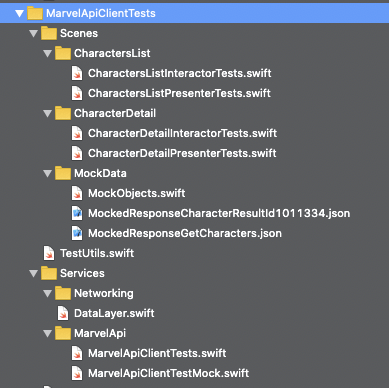
This screen requires the extra storing functionality of the app.

It also requires a state machine logic incorporated into the AppCoordinator that determines the app floor to launch depending on some conditions ( the onboarding has been skipped OR the user has already seen it and doesn’t want to see it again. In perspective, this is very much required when the flow of the App depends on more complex conditions like the user being logged in or not.

* Deep Linking functionality. Scheme supported:
  + marvelapiclient://characters/
  + marvelapiclient://character/{characterId} ( for instance characterId = 1010755)
  + marvelapiclient://comics/
  + marvelapiclient://comic/{comicId}( for instance comicId = 1886)

To test the deep link functionality use the Notes app to edit one of the supported links. The app should start ( if not already active) and lead exactly to the specified page.

* DataServiceLayer:
  + A generic component that persists data on disk. The concrete method is UserDefaults but this is an implementation detail
* Networking Layer: RestApiClient, ApiRequest etc
  + Encapsulates REST related operations in a structured way allowing a good scalability of the solution
* Tests:
* The original requirements ( not the extra ones) have been fully unit tested ( except for UI testing):
  + Presentation layer tests
  + Interactor layer with
  + Api Service Layer:
    - With mocked data
    - And with real end to end connection



# Frameworks and tools used:

[RSwift](https://github.com/mac-cain13/R.swift) ( cocoaPods): allows static typing for name based resources.

[AlamofireImage](https://github.com/Alamofire/AlamofireImage) ( swiftPakage): allows to asynchronously load images

[Siterep](https://github.com/twostraws/Sitrep) ( script with installed program)

Gives statistics about the project( lines of code, nº oc structs/classes etc)

[SwiftLint](https://github.com/realm/SwiftLint) ( script with installed program)

Static analysis tool

[Quicktype](https://app.quicktype.io/)

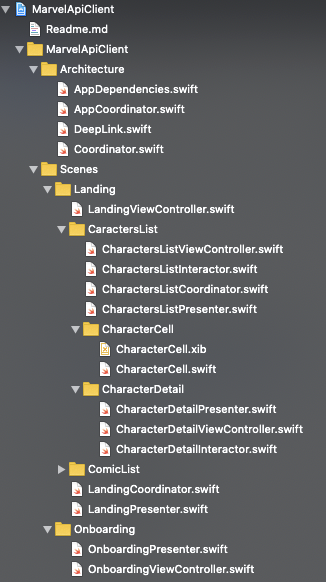
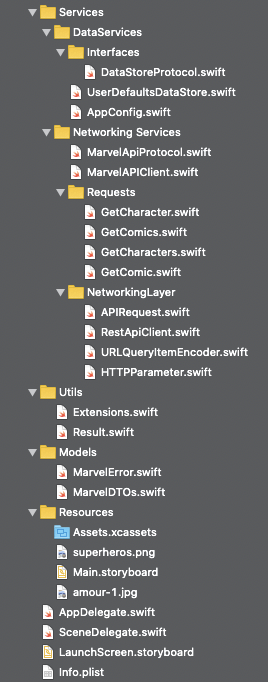
To quickly generate data classes from the json structure provided by Marvel

[Draw.io](https://www.draw.io/)

To draw the diagrams presented here

## 

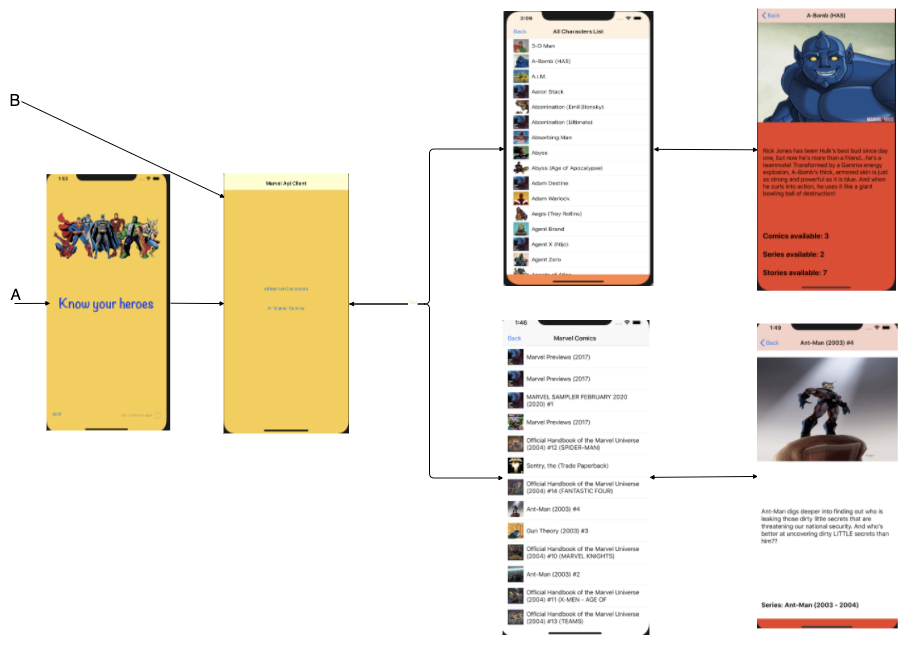
## Xcode Project structure:

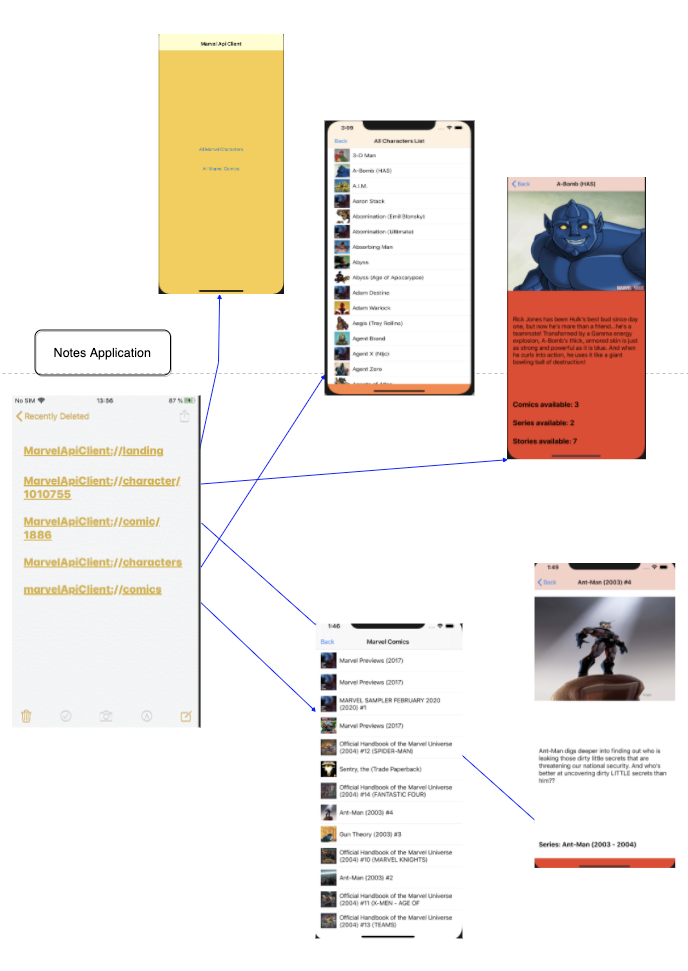
# 

# App Flows

Normal Flow



DeepLink Flows:



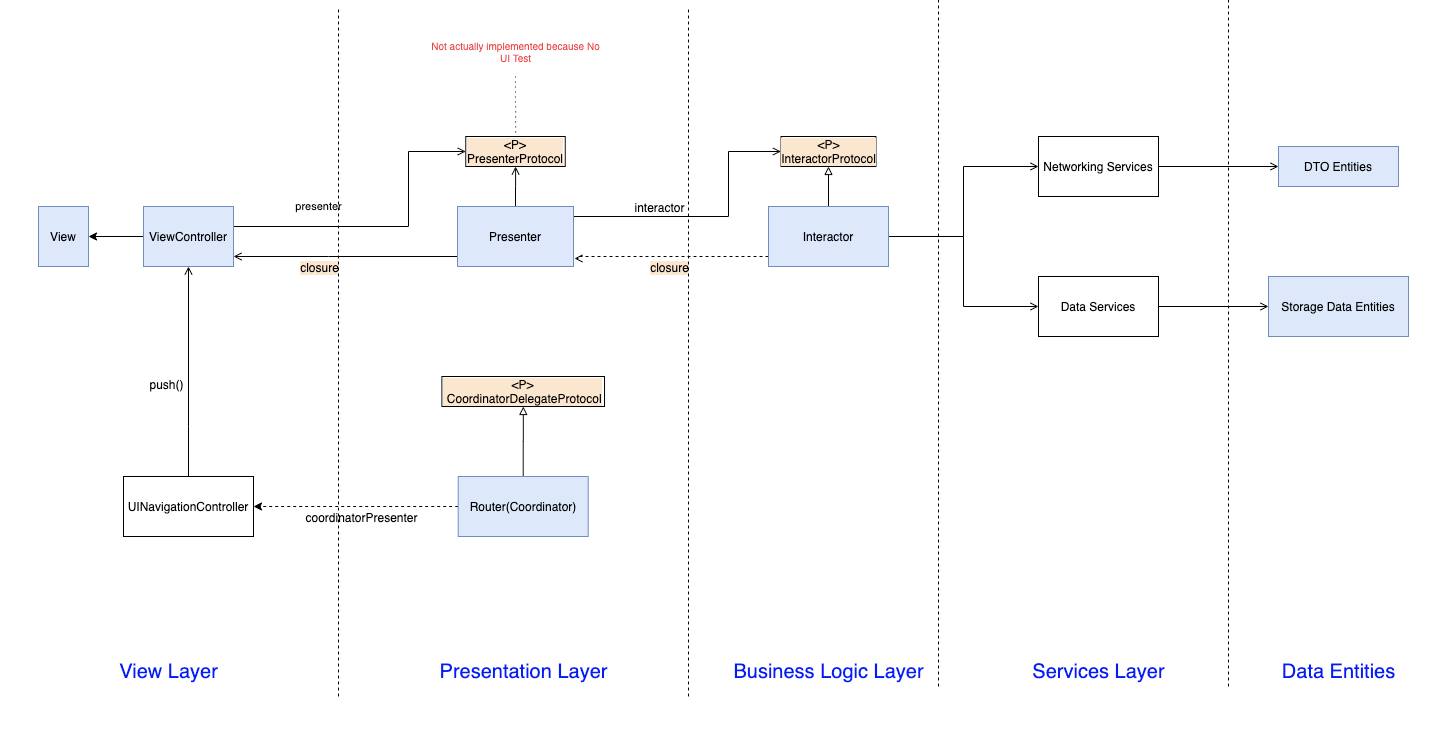
# Architecture

For this exercise I have used a VIPER architecture following the [Clean Architecture](https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html) model.

**Note**: As Router I have used the [Coordinator](https://khanlou.com/2015/01/the-coordinator/) pattern for its flexibility and ability to easily support deep linking.

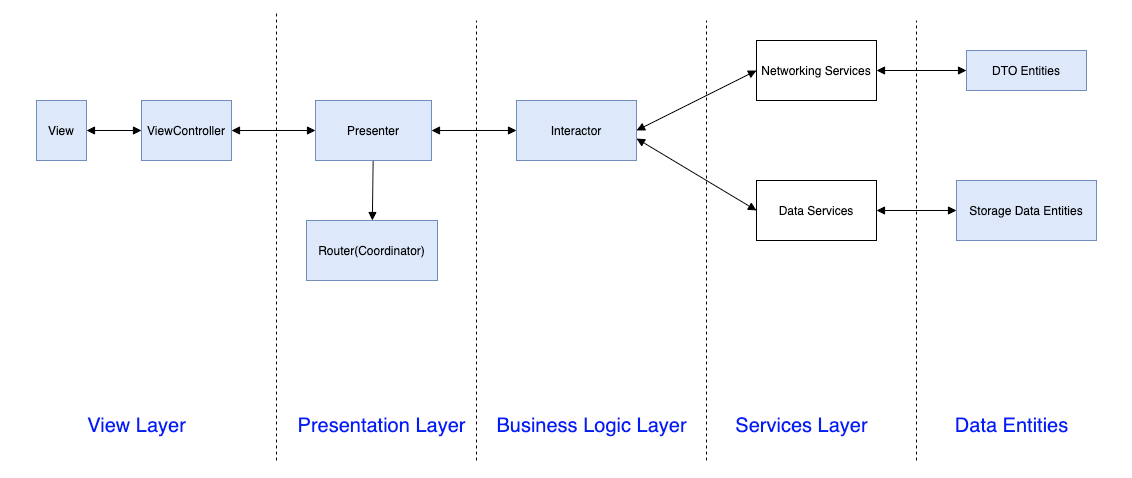
This model allows the identification and definition of responsibility boundaries through **protocols** as we can see from the following diagram:

Dependency diagram:



The use of **Protocols** along with the use of **Dependency Injection** has been used for the keys classes in order to ease the mocking of objects necessary to test in isolation.

Data flow diagram



Quality parameters:

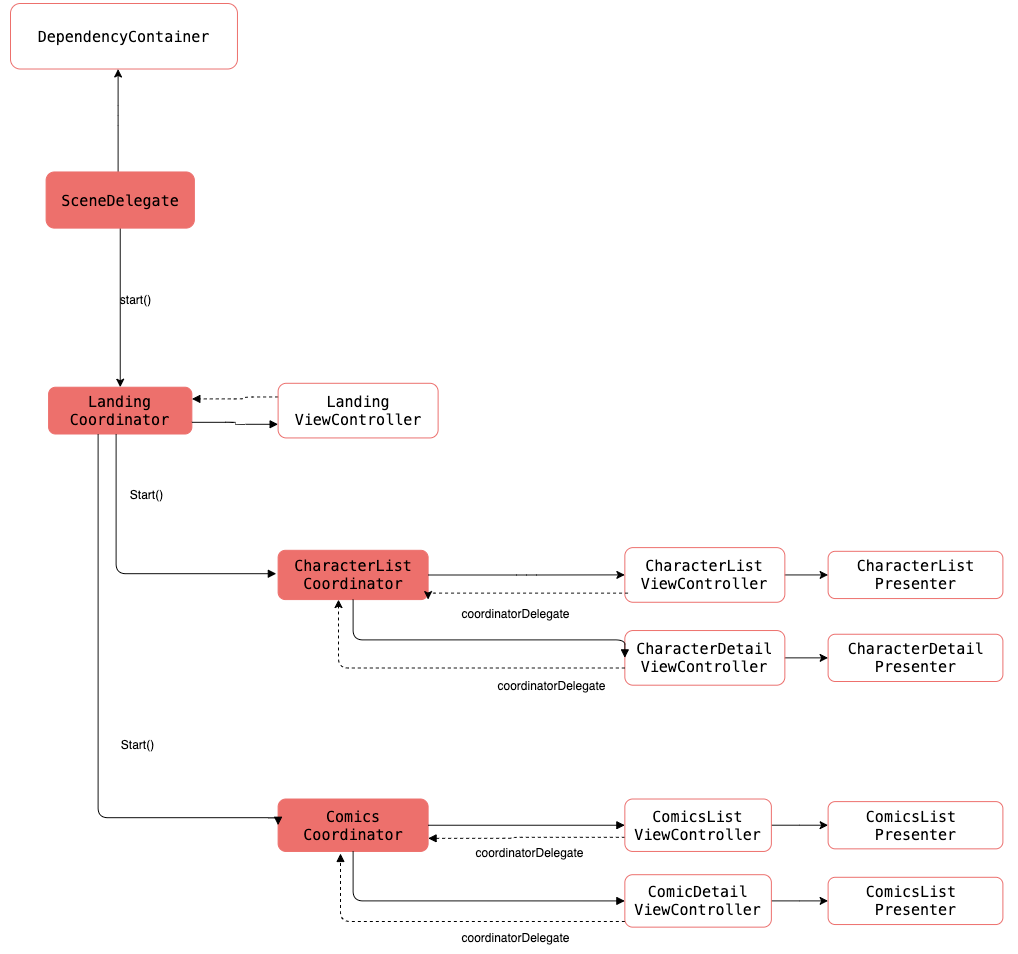
* Followed S.O.L.I.D. principles
* Zero errors/warnings
* Swiftlint functions < 15 lines of code for easy code readability

## Coordinators:

The basic principle is that the viewcontrollers are instantiated programmatically, without segues, by the coordinator. This means that each viewcontroller doesn't need to know anything about the following viewcontroller so that the two things are decoupled.

Moreover a viewcontroller doesn't need to tell its owner, the navigator, to push a new viewController which would be a breach of the the separation of concern objective

So we take the flow control out of the viewcontroller making in fact the viewcontroller reusable in other flows and can be more easily tested.



* One AppCoordinator to encapsulate the display of the initial screen
* I’m using a coordinator for each flow that defines a unity of coordination: LandingCoordinator for the top level flow; CharactersListCoordinator for the characters flow , ComicsListCoordinator, for the Comics flow.
* The coordinator instantiate and configure the relevant Vcs according to the flow logic it defines in its delegateProtocols (one for each VC it coordinates)
* A View uses these delegate methods to pass events notifications to its coordinator which will take the decision on the next route.

## Possible Improvements

* Improve the UI design
* Test edge cases and complete tests also for extra requirements. Test the data layer
* Improve the deep linking mechanism. Now it allows to reach a particular screen in “isolation” that means that it is not possible for the user to navigate back from that screen.
* Create a separate RestApiClient and DataLayer components in as a Swift Package Manager package
* Be able to select a character comic from the character detail screen and vice versa
* Implement Error screens
* Search comics and characters by name or other parameters.