

---

## Library - iOS - MVP + Clean Architecture Demo

### Description

*Library* is an iOS application built to highlight **MVP (Model View Presenter)** and **Clean Architecture** concepts

### Run Requirements

- Xcode 10.2.1
- Swift 5

### High Level Layers

### MVP Concepts

### Presentation Logic

- **View** - delegates user interaction events to the **Presenter** and displays data passed by the **Presenter**
  - All **UIViewController**, **UIView**, **UITableViewCell** subclasses belong to the **View** layer
  - Usually the view is passive / dumb - it shouldn't contain any complex logic and that's why most of the times we don't need write Unit Tests for it
- **Presenter** - contains the presentation logic and tells the **View** what to present
  - Usually we have one **Presenter** per scene (view controller)
  - It doesn't reference the concrete type of the **View**, but rather it references the **View** protocol that is implemented usually by a **UIViewController** subclass
  - It should be a plain **Swift** class and not reference any **iOS** framework classes - this makes it easier to reuse it maybe in a **macOS** application
  - It should be covered by Unit Tests
- **Configurator** - injects the dependency object graph into the scene (view controller)
  - You could very easily use a DI (dependency injection) library. Unfortunately DI libraries are not quite mature yet on **iOS / Swift**
  - Usually it contains very simple logic and we don't need to write Unit Tests for it

- 
- **Router** - contains navigation / flow logic from one scene (view controller) to another
    - In some communities / blog posts it might be referred to as a **FlowController**
    - Writing tests for it is quite difficult because it contains many references to **iOS** framework classes so usually we try to keep it really simple and we don't write Unit Tests for it
    - It is usually referenced only by the **Presenter** but due to the `func prepare(for segue: UIStoryboardSegue, sender: Any?)` method we some times need to reference it in the view controller as well

## Clean Architecture Concepts

### Application Logic

- **UseCase / Interactor** - contains the application / business logic for a specific use case in your application
  - It is referenced by the **Presenter**. The **Presenter** can reference multiple **UseCases** since it's common to have multiple use cases on the same screen
  - It manipulates **Entities** and communicates with **Gateways** to retrieve / persist the entities
  - The **Gateway** protocols should be defined in the **Application Logic** layers and implemented by the **Gateways & Framework Logic**
  - The separation described above ensures that the **Application Logic** depends on abstractions and not on actual frameworks / implementations
  - It should be covered by Unit Tests
- **Entity** - plain **Swift** classes / structs
  - Models objects used by your application such as **Order**, **Product**, **Shopping Cart**, etc

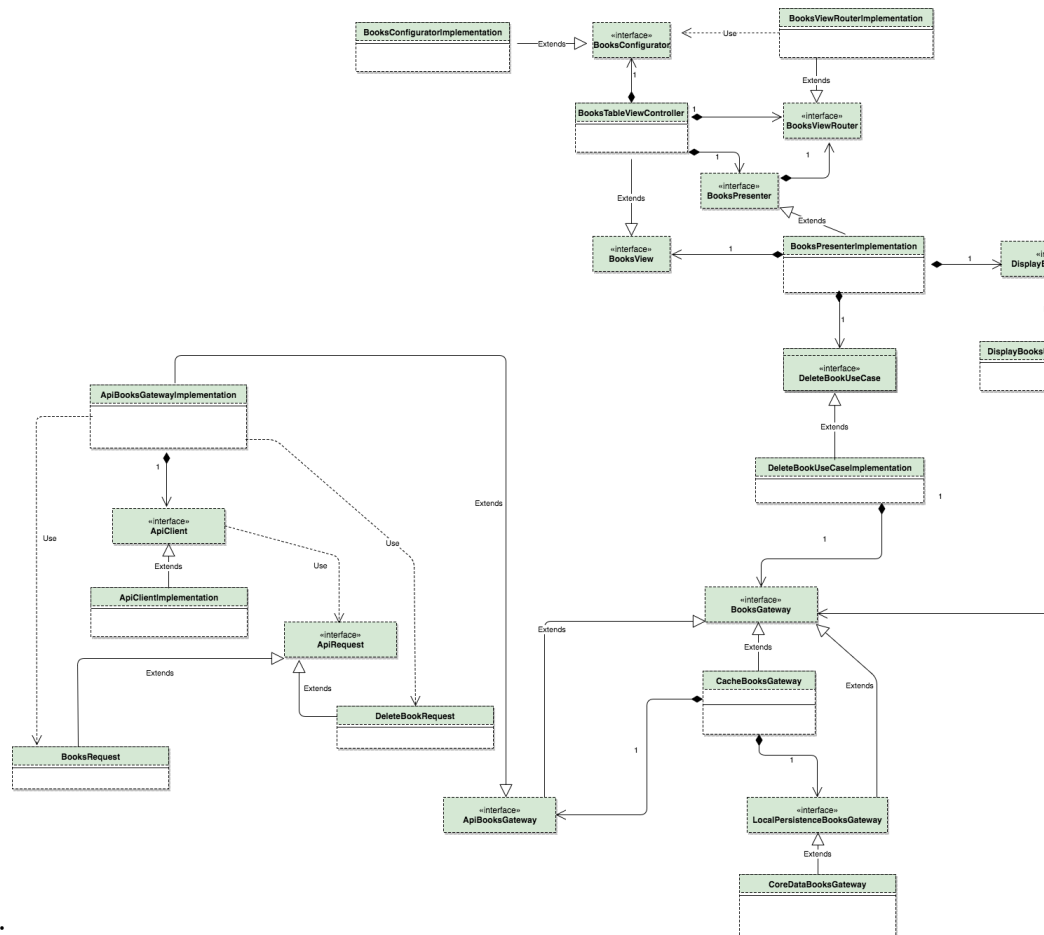
### Gateways & Framework Logic

- **Gateway** - contains actual implementation of the protocols defined in the **Application Logic** layer
  - We can implement for instance a **LocalPersistenceGateway** protocol using **CoreData** or **Realm**
  - We can implement for instance an **ApiGateway** protocol using **URLSession** or **Alamofire**
  - We can implement for instance a **UserSettings** protocol using **UserDefaults**

- 
- It should be covered by Unit Tests
  - **Persistence / API Entities** - contains framework specific representations
    - For instance we could have a **CoreDataOrder** that is a **NSManagedObject** subclass
    - The **CoreDataOrder** would not be passed to the **Application Logic** layer but rather the **Gateways & Framework Logic** layer would have to “transform” it to an **Order** entity defined in the **Application Logic** layer
  - **Framework specific APIs** - contains implementations of **iOS** specific APIs such as sensors / bluetooth / camera

### Demo Application Details

- The demo applications tries to expose a fairly complex set of features that justifies the usage of the concepts presented above
- The following **Unit Tests** have been written:
  - **BooksPresenterTest** - highlights how you can test the presentation logic
  - **DeleteBookUseCaseTest** - highlights how you can test the application / business logic and also how to test async code that uses completion handlers and **NotificationCenter**
  - **CacheBooksGatewayTest** - highlights how you can test a cache policy
  - **CoreDataBooksGatewayTest** - highlights how you can test a **CoreData** gateway
  - **ApiClientTest** - highlights how you can test the API / Networking layer of your application by substituting the **URLSession** stack
- **Code comments** can be found in several classes highlighting different design decisions or referencing followup resources
- The project structure tries to mimic the **Screaming Architecture** concept that can be found in the references section



- High level UML diagram:

## Debatable Design Decisions

Giving that a large majority of mobile apps are a thin client on top of a set of APIs and that most of them contain little business logic (since most of the business logic is found in the APIs) some of the [Clean Architecture](#) concepts can be debatable in the mobile world. Below you can find some:

- Creating a representation for each layer (API, CoreData) might seem like over-engineering. If your application relies heavily on an API that is under your control then it might make sense to model both the entity and the API representation using the same class. You shouldn't however allow the persistence representation (the [NSManagedObject](#) subclass for instance) leak in the other layers (see [Parse](#) example that got discontinued)
- If you find that in most cases your [Use Cases](#) / [Interactors](#) simply delegate the actions to the [Gateway](#) then maybe you don't need the [Use Cases](#) / [Interactors](#) in the first place and you can use the [Gateway](#) directly in the [Presenter](#)
- If you want to enforce the layer separation even more you can consider moving all the layers in

---

their own projects / modules

- Some might consider that creating `display(xyz: String)` methods on a `CellView` protocol is over-engineering and that passing a plain `CellViewModel` object to the `CellView` and have the view configure itself with the view model is more straightforward. If you want to keep the view as passive / dumb as possible then you should probably create the methods, but then again simply reading some strings from a view model and setting some labels is not really complex logic

The list above is definitely not complete, and if you identify other debatable decisions please create an issue and we can discuss about it and include it in the list above.

For the items listed above (and also for other items of your own) it is important that you use **your own judgement** and make an **informed decision**.

Keep in mind that you don't have to make all the design decisions up front and that you can refactor them in as you go.

Discuss about all the design decision with your team members and make sure you are all in agreement.

## Useful Resources

### MVP & Other presentation patterns

- iOS Architecture Patterns
- Architecture Wars - A New Hope
- VIPER to be or not to be?
- Effective Android Architecture - our note here is that you should be careful about coupling your application to Rx\* or any other framework for that matter. Please read Make the Magic go away, by Uncle Bob and think twice before letting a framework take over your application.
- Improve your iOS Architecture with FlowControllers
- GUI Architectures, by Martin Fowler

### Clean Architecture

- The Clean Architecture, by Uncle Bob
- Architecture: The Lost Years, by Uncle Bob
- Clean Architecture, By Uncle Bob
- Uncle Bob's clean architecture - An entity/model class for each layer?

---

## **Unit Tests**

- [xUnit Test Patterns: Refactoring Test Code](#)
- [The Art of Unit Testing: with examples in C#](#)

## **Contributing**

Please feel free to open an issue for any questions or suggestions you have!