iOS Development: Architecture

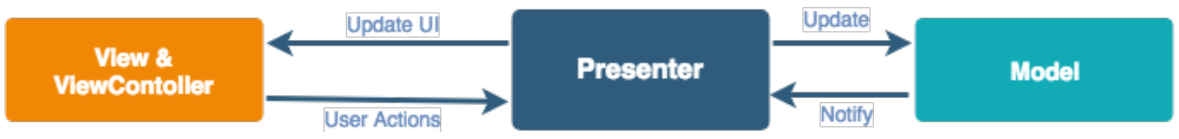
Fundamental Arhitectural Design Patterns

MVC is a software development pattern made up of three main objects:

- The **Model** is where your data resides. Things like persistence, model objects, parsers, managers, and networking code live there.
- The **View** layer is the face of your app. Its classes are often reusable as they don't contain any domain-specific logic. For example, a UILabel is a view that presents text on the screen, and it's reusable and extensible.
- The **Controller** mediates between the view and the model via the delegation pattern. In an ideal scenario, the controller entity won't know the concrete view it's dealing with. Instead, it will communicate with an abstraction via a protocol. A classic example is the way a UITableView communicates with its data source via the UITableViewDataSource protocol.

MVP As an alternative :

Then MVP architecture comes to improve this situation. by adding the main component which is **Presenter**.



Then components description becomes as the following :

- **View** : The view now consists of both views and view controllers, with all UI setup and events.
- **Presenter** : The presenter will be in charge of all the logic , including responding to user actions and updating the UI (via delegate). and the most important is that our **presenter will not be UIKit dependent**. which means well isolated, hence easily testable ;)
- **Model** : the model role will be exactly the same

It's important to note that **MVP uses passive View pattern**. it means all the actions will be forwarded to the presenter. Which will trigger the ui updates using delegates. so the view will only passe actions and listen to the presenter updates.

MVVM Design Pattern

Components Overview and their roles

- **View Controlle**r: It only performs things related to UI — Show/get information. Part of the view layer
- **View Model**: It receives information from VC, handles all this information, and sends it

back to VC.
Model: This is only your model, nothing much here. It’s the same model as in MVC. It is used by VM and updates whenever VM sends new updates

Application flow will be like this:

- View controller will get called and the view will have a reference to the ViewModel
- The View will get some user action and the view will call ViewModel
- ViewModel will request APIService and APIService sends a response back to the ViewModel
- Once we receive a response, ViewModel notifies the view through binding
- The View will update the UI with data

The SOLID Principles:

SOLID represents 5 principles of object-oriented programming:

- **S**ingle Responsibility Principle -> It states that every module should have only one responsibility and reason to change
- **O**pen/Closed Principle - > Open for extension but closed for modification.
- **L**iskov Substitution Principle -> Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

- **I**nterface Segregation -> It states that clients should not be forced to implement interfaces they don’t use.
- **D**ependency Inversion -> High-level modules should not depend on low-level modules both should depend on Abstractions. (Abstractions should not depend upon details. Details should depend upon abstractions)

Building Responsive Apps

The main thread is the one that starts our program, and it’s also the one where all our UI work must happen. However, there is also a main queue, and although sometimes we use the terms “main thread” and “main queue” interchangeably, they aren’t quite the same thing.

That's so important it bears repeating twice: **it's never OK to do user interface work on the background thread**.

So, if you’re on the main queue then you’re definitely on the main thread, but being on the main thread doesn’t automatically mean you’re on the main queue – a different queue could temporarily be running on the main thread.

If you're on a background thread and want to execute code on the main thread, you need to call **async()** again. This time, however, you do it on **DispatchQueue.main**, which is the main thread, rather than one of the global quality of service queues.