

Bài viết này, mình sẽ trình bày về MVP.

Content

- Tổng quan về MVC
- Những vấn đề của MVC
- Ý tưởng của MVP
- Áp dụng MVP

Tổng quan về MVC:

Chắc hẳn các bạn đã quen với MVC: 1 trong những architecture pattern phổ biến nhất.

MVC bao gồm 3 phần chính:

Model

Model Layer là nơi lưu trữ data của bạn.

Model layer bao gồm:

- Model Objects (hiển nhiên rồi).
- Ngoài ra còn có 1 vài class và object khác có thể trong Model như các class xử lí Network code, Persistence code(Lưu trữ data đến database, CoreData,...), Parsing Code(parsing network response to model,...), các class Helper, Extensions, ...

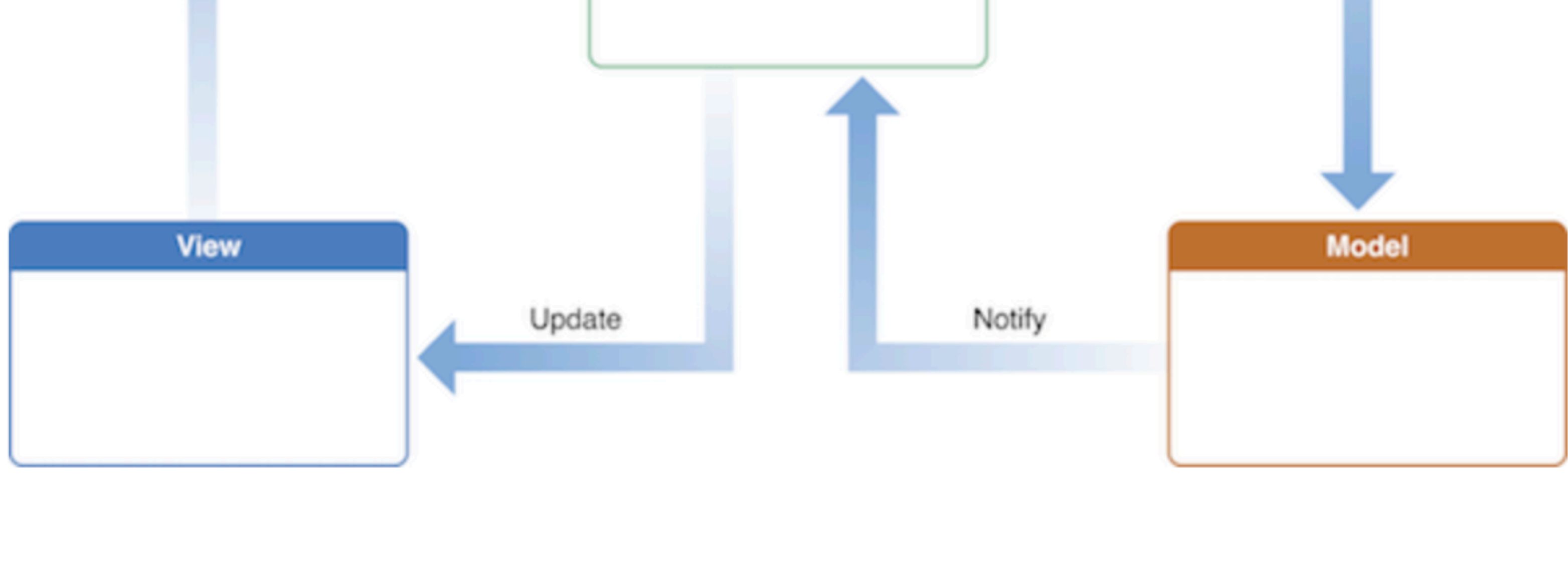
View

Là nơi hiển thị giao diện cho người dùng.

VD: Xib file, UIView class, Core Animation, ... Nói chung là những thứ liên quan đến UIKit

Controller

Là phần tương tác với View và Model. Controller nhận action/events từ view để update Model và View.



Những điều cần chú í ở MVC:

- View chỉ dùng để update UI, không chứa các logic.
- View không được tương tác trực tiếp với Model.
- View không được làm bất cứ điều gì mà không liên quan đến chính nó.

Vấn đề của MVC?:

- Controller chứa rất nhiều logic như update UI, xử lí các logic user action, logic networking, ... tuy điều này có thể được giảm thiểu bởi tách các logic như Network, Lưu database,... sang các class khác.
- Khó test các logic của controller bởi controller liên kết chặt chẽ với View.
- Dự án càng lớn, code controller càng nhiều, dẫn đến khó maintain, bảo trì.

Ý tưởng của MVP:



MVP gồm 3 phần:

View:

Bao gồm Views và View Controller, dùng để tương tác, xử lí UI và nhận các event của user.

Presenter:

Chịu trách nhiệm xử lí logic, gồm các logic xử lí user action, logic networking, tương tác database...

Các event của user sẽ được gửi đến Presenter để Presenter xử lí logic tương ứng, sau đó sẽ tương tác, yêu cầu View update UI bằng cách sử dụng cách sử dụng delegate.

Model:

Tương tự như MVC.

“Note: Tầng Presenter phải không phụ thuộc vào UIKit, để tách biệt với View. Qua đó giúp dễ viết test cho phần logic hơn.

Giờ hãy bắt tay vào demo:

Demo:

Model Entity:

```
import Foundation

struct Category {
    let id: Int
    let name: String
}
```

Class Service chịu trách nhiệm tương tác với Model:

```
import Foundation

protocol ICategoryService {
    func getCategoryById(id: Int) -> Category?
}

class CategoryService: ICategoryService {
    func getCategoryById(id: Int) -> Category? {
        let categories = [Category(id: 1, name: "Movies"),
                           Category(id: 2, name: "Books"),
                           Category(id: 3, name: "Computer")]

        let category = categories.filter({
            $0.id == id
        }).first

        return category
    }
}
```

Class Presenter: Chịu trách nhiệm xử lí logic

```
import Foundation

protocol IHomeViewDelegate: NSObjectProtocol {
    func updateUI(_ categoryName: String)
}

class HomeViewPresenter {
    // 1
    weak var delegate: IHomeViewDelegate?
    let categoryService: ICategoryService!

    init(view: IHomeViewDelegate, service: ICategoryService) {
        delegate = view
        categoryService = service
    }

    func searchCategoryById(id: Int) {
        let result = categoryService.getCategoryById(id: id)
        if let categoryName = result?.name {
            // 2
            delegate?.updateUI(categoryName)
        }
    }
}
```

1. Khai báo delegate để presenter có thể dùng để tương tác với View.
Ở đây, delegate và category được khai báo kiểu protocol và được khởi tạo trong hàm init để tránh bị dependency.
2. Sau khi thực hiện xong việc truy vấn database, presenter dùng delegate để yêu cầu View update UI.

Class View:

```
import UIKit

class HomeView: UIViewController {
    @IBOutlet private weak var categoryNameLabel: UILabel!

    private var presenter: HomeViewPresenter!

    override func viewDidLoad() {
        super.viewDidLoad()
        presenter = HomeViewPresenter(view: self, service: CategoryService())
    }

    // 1
    @IBAction func didTapButtonSearch(_ sender: Any) {
        presenter.searchCategoryById(id: 1)
    }
}

extension HomeView: IHomeViewDelegate {
    // 2
    func updateUI(_ categoryName: String) {
        categoryNameLabel.text = categoryName
    }
}
```

1. Khi user tap button, View sẽ chuyển action đến presenter và yêu cầu presenter thực hiện logic.
2. Update UI khi được presenter yêu cầu.

Lợi ích của MVP:

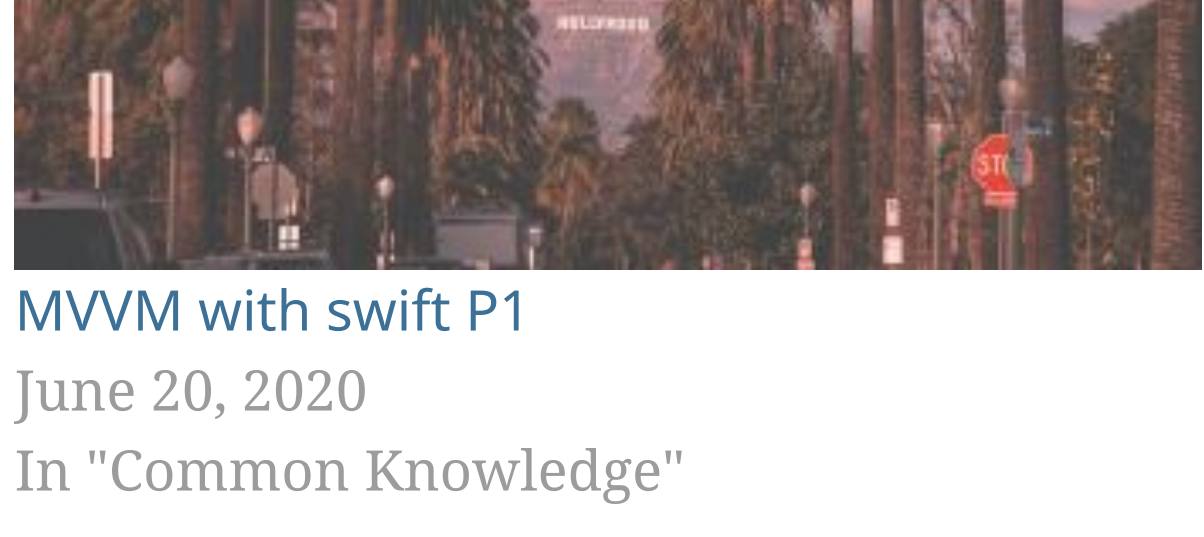
MVP đã giúp tách biệt logic khỏi View, từ đó mang đến các lợi ích:

- Dễ dàng đọc hiểu code.
- Dễ dàng viết Unit test cho logic.
- Dễ maintain, bảo trì.

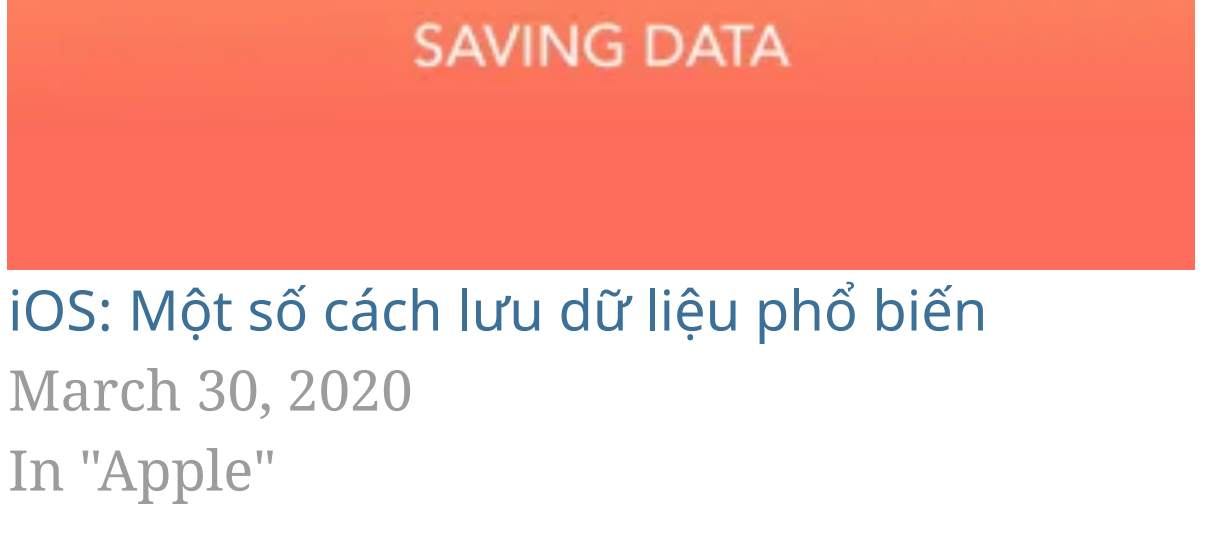
Kết luận:

- MVP chỉ giải quyết việc tách biệt logic khỏi View. Tuy nhiên, những phần logic như truy vấn database, networking, ... thì chưa được xử lí. Những logic như vậy có thể được đặt ở presenter, hoặc tạo 1 class riêng ở Model tùy theo bản thân bạn. **Tuy nhiên, nên tách biệt các phần logic Database, networking, ... ra các class riêng để tuân thủ nguyên tắc Single Responsibility Principle của SOLID.**
- Theo mô hình MVP, Presenter không được import UIKit để tránh logic bị liên quan đến View.

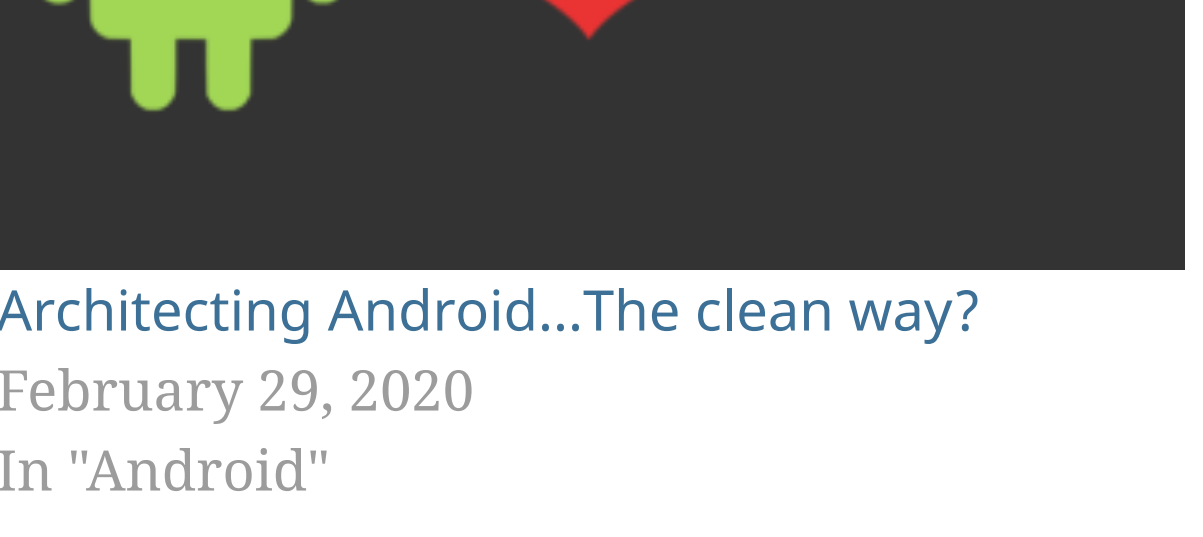
Related



MVM with swift P1
June 20, 2020
In "Common Knowledge"



iOS: Một số cách lưu dữ liệu phổ biến
March 30, 2020
In "Apple"



Architecting Android...The clean way?
February 29, 2020
In "Android"

LEAVE A COMMENT

Your Comment

Name*

Email*

Website

☐ Save my name, email, and website in this browser for the next time I comment.

☐ NOTIFY ME OF FOLLOW-UP COMMENTS BY EMAIL.

☐ NOTIFY ME OF NEW POSTS BY EMAIL.

* By using this form you agree with the storage and handling of your data by this website.

SUBMIT

