Mô hình MVVM trong Swift

Có thể bạn cũng đã biết về mô hình MVC(Model-View-Controller) nhưng trong iOS có thể nó sẽ thành Massive-View-Controller vì nếu bạn đã làm một dự án lớn thì có thể các view controllers có thể sẽ hơn 1000 dòng code, đấy chính là vấn đề tại sao nó biến các view controllers ở lên hỗ độn và khi đọc rất khó chịu.

Dưới đấy là cấu trúc theo lý thuyết MVC:

i)

Nhưng trong thực tế đôi khi nó sẽ khác:

i)

Như các bạn thấy thành phần Controllers sẽ phình to ra khi dự án phát triển.

Trong bài này hôm nay chúng ta sẽ tìm hiểu về MVVM(Model-View-ViewModel). Cấu trúc nó sẽ như sau:

i)

Bây giờ thì thành phần Controller có nhiện vụ hiển thị những gì mà View Model muốn hiển thị và nó không quan tâm và cũng không biết về Model. Các thành phần như UILabel, UITextField,… nó chỉ hiển thị dữ liệu mà ViewModel cung cấp.

Vậy lợi ích ở đây là gì?

Ví dụ: Controller sẽ không phình to nữa. Nếu muốn chỉnh sửa lại Model, chúng ta đơn giản chỉ cần làm ở View Model. Ví dụ khi thay đổi và hiển thị các định dạng khác nhau với NSDate. Thường thì chúng ta sẽ chỉnh sửa nó ngay trong ViewController, có nghĩa là ViewController sẽ phình to ra. Thay vì thế chúng ta sẽ xử lý định dạng để hiển thị ngày tháng ở trong ViewModel nó sẽ chỉ rõ ra class/file nào chịu trách nhiệm cho việc chuyển đổi từ NSData sang String.

Lợi ích nữa khi sử dụng MVVM là ứng dụng sẽ dễ kiểm thử hơn, bạn có thể dễ dàng viết các tests cho các ViewModels vì các thành phần liên kết dưới loose coupling.

Ví dụ:

Model, ViewModel, Unit Test

Tôi sẽ tạo class Car

Swift

|  |
| --- |
| class Car {    var model: String    var make: String    var kilowatts: Int    var photoURL: String      init(model: String, make: String, kilowatts: Int, photoURL: String) {      self.model = model      self.make = make      self.kilowatts = kilowatts      self.photoURL = photoURL    }  } |

Và tiếp theo là ViewModel tôi đặt tên là CarViewModel:

|  |  |
| --- | --- |
| 2  3  4  5  6  7  8  9  10  11  12 | class CarViewModel {    private var car: Car    static let horsepowerPerKilowatt = 1.34102209      var modelText: String {      return car.model    }      var makeText: String {      return car.make    }      var horsepowerText: String {      let horsepower = Int(round(Double(car.kilowatts) \* CarViewModel.horsepowerPerKilowatt))      return "\(horsepower) HP"    }      var titleText: String {      return "\(car.make) \(car.model)"    }      var photoURL: NSURL? {      return NSURL(string: car.photoURL)    }      init(car: Car) {      self.car = car    }  } |

Tiếp theo tôi sẽ viết TDD test(test driven development)

i)

Chúng ta sẽ khởi tạo đối tượng Ferrari F12 như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | func testCarViewModelWithFerrariF12() {    let ferrariF12 = Car(model: "F12", make: "Ferrari", horsepower: 730, photoURL: "http://auto.ferrari.com/en\_EN/wp-content/uploads/sites/5/2013/07/Ferrari-F12berlinetta.jpg")    let ferrariViewModel = CarViewModel(car: ferrariF12)    XCTAssertEqual(ferrariViewModel.modelText, "F12")    XCTAssertEqual(ferrariViewModel.makeText, "Ferrari")    XCTAssertEqual(ferrariViewModel.horsepowerText, "730 HP")    XCTAssertEqual(ferrariViewModel.photoURL, NSURL(string: ferrariF12.photoURL!))    XCTAssertEqual(ferrariViewModel.titleText, "Ferrari F12")  } |

Bây giờ bạn có thể ấn command+U để chạy thì nó sẽ báo lỗi, nhưng không sao chúng ta sẽ sửa nó.

class CarViewModel {

  private var car: Car

  static let horsepowerPerKilowatt = 1.34102209

  var modelText: String {

    return car.model

  }

  var makeText: String {

    return car.make

  }

  var horsepowerText: String {

    let horsepower = Int(round(Double(car.kilowatts) \* CarViewModel.horsepowerPerKilowatt))

    return "\(horsepower) HP"

  }

  var titleText: String {

    return "\(car.make) \(car.model)"

  }

  var photoURL: NSURL? {

    return NSURL(string: car.photoURL)

  }

  init(car: Car) {

    self.car = car

  }

}

Bây giờ chúng ta có thể thấy lợi ích của việc sử dụng MVVM. Class Car khá đơn giản, nhưng CarViewModel lại thêm một số các phần getter của các thuộc tính. Ví dụ với biến titleText, nó liên kết với thuộc tính của Car tương tự với horsepowerText,… Bây giờ bạn có thể ấn command+U để chạy test này.

User interface(UITableView)

Để giải thích chi tiết hơn về MVVM, chúng ta sẽ tạo một mảng cars và hiển thị nó lên UITableView.

Đầu tiên tôi sẽ tạo một class mới có tên TableViewController mà nó là subclass của UITableViewController

i)

Tôi xoá ViewController hiện tại ở trong Main.storyboard vào kéo một TableViewController vào.

i)

Sau đó tôi set định danh cho cell là CarCell và chỉnh style thành Right Detail(ở mỗi cell chúng ta sẽ hiển thị photo, title, .. của Car).

i)

Bây giờ chúng ta sẽ viết các đoạn mã cho class TableViewController.

Nếu chúng ta đang làm theo mô hình MVVM thì ở thành phần hiển thị là các Views sẽ không cần biết về model Car.

Trong ứng dụng thực thế thì ứng dụng sẽ lấy dữ liệu từ bên ngoài app nhưng trong ví dụ này chúng ta sẽ tạo dữ liệu cứng và để ở AppDelegate.

let cars: [CarViewModel] = {

  let ferrariF12 = Car(model: "F12", make: "Ferrari", horsepower: 730, photoURL: "http://auto.ferrari.com/en\_EN/wp-content/uploads/sites/5/2013/07/Ferrari-F12berlinetta.jpg")

  let zondaF = Car(model: "Zonda F", make: "Pagani", horsepower: 602, photoURL: "http://storage.pagani.com/view/1024/BIG\_zg-4-def.jpg")

  let lamboAventador = Car(model: "Aventador", make: "Lamborghini", horsepower: 700, photoURL: "http://cdn.lamborghini.com/content/models/aventador\_lp700-4\_roadster/gallery\_2013/roadster\_21.jpg")

  return [CarViewModel(car: ferrariF12), CarViewModel(car: zondaF), CarViewModel(car: lamboAventador)]

}()

Và ở TableViewController tôi sẽ lấy dữ liệu từ Appdelegate:

|  |
| --- |
| let cars: [CarViewModel] = (UIApplication.sharedApplication().delegate as! AppDelegate).cars |

Chúng ta tiếp tục khai báo 2 hàm sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | override func numberOfSectionsInTableView(tableView: UITableView) -> Int {    return 1  }    override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {    return cars.count  } |

Tiếp theo đến phần hiển thị dữ liệu:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {    let cell = tableView.dequeueReusableCellWithIdentifier("CarCell", forIndexPath: indexPath)    let carViewModel = cars[indexPath.row]      cell.textLabel?.text = carViewModel.titleText    cell.detailTextLabel?.text = carViewModel.horsepowerText    loadImage(cell, photoURL: carViewModel.photoURL)      return cell  } |

Các bạn có thể thấy có function loadImage, dĩ nhiên chúng ta sẽ không muốn block main thread nên lúc tải ảnh tôi sẽ chon nó chạy ở background và khi tải xong thì hiển thị ở main thread.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | func loadImage(cell: UITableViewCell, photoURL: NSURL?) {    dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0)) {      guard let imageURL = photoURL, imageData = NSData(contentsOfURL: imageURL) else {        return      }      dispatch\_async(dispatch\_get\_main\_queue()) {        cell.imageView?.image = UIImage(data: imageData)        cell.setNeedsLayout()      }    }  } |

Từ iOS 9 trở đi khi truy vấn đến một URL thì cần khai báo như sau ở plist:

nfo.plist

Swift

<key>NSAppTransportSecurity</key>

<dict>

<key>NSAllowsArbitraryLoads</key>

<true/>

</dict>

Bây giờ khi chạy ứng dụng thì nó sẽ hiển thị dữ liệu như sau:

i)

Dưới đây tôi sẽ viết một đoạn mã đơn giản về UI test:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | class MVVMUITests: XCTestCase {      override func setUp() {      super.setUp()        // Put setup code here. This method is called before the invocation of each test method in the class.        // In UI tests it is usually best to stop immediately when a failure occurs.      continueAfterFailure = false      // UI tests must launch the application that they test. Doing this in setup will make sure it happens for each test method.      XCUIApplication().launch()        // In UI tests it’s important to set the initial state - such as interface orientation - required for your tests before they run. The setUp method is a good place to do this.    }      override func tearDown() {      // Put teardown code here. This method is called after the invocation of each test method in the class.      super.tearDown()    }      func testFerrariF12DataDisplayed() {      let app = XCUIApplication()      let table = app.tables.elementBoundByIndex(0)        let ferrariCell = table.cells.elementBoundByIndex(0)      XCTAssert(ferrariCell.staticTexts["Ferrari F12"].exists)      XCTAssert(ferrariCell.staticTexts["730 HP"].exists)        let zondaCell = table.cells.elementBoundByIndex(1)      XCTAssert(zondaCell.staticTexts["Pagani Zonda F"].exists)      XCTAssert(zondaCell.staticTexts["602 HP"].exists)        let lamboCell = table.cells.elementBoundByIndex(2)      XCTAssert(lamboCell.staticTexts["Lamborghini Aventador"].exists)      XCTAssert(lamboCell.staticTexts["700 HP"].exists)    }    } |

Khi dữ liệu thay đổi

Giả sử Car model cần thay đổi, dữ liệu hay đổi khi kilowatts của Car thay đổi, vậy chúng ta sẽ chỉ cần kiểm tra xem kilowatts có giá trị hay không trước khi tính toán

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Car {    var model: String?    var make: String?    var kilowatts: Int?    var photoURL: String?      init(model: String, make: String, kilowatts: Int, photoURL: String) {      self.model = model      self.make = make      self.kilowatts = kilowatts      self.photoURL = photoURL    }  } |

Và thay đổi CarViewModel như sau:

Swift

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // class CarViewModel {  // ...  static let horsepowerPerKilowatt = 1.34102209  var horsepowerText: String? {    guard let kilowatts = car?.kilowatts else {      return nil    }    let horsepower = Int(round(Double(kilowatts) \* CarViewModel.horsepowerPerKilowatt))    return "\(horsepower) HP"  }  // ...  // } |

Dĩ nhiên ở AppDelegate chúng ta cũng cần thay đổi phần khởi tạo dữ liệu:

// AppDelegate.swift

let cars: [CarViewModel] = {

let ferrariF12 = Car(model: "F12", make: "Ferrari", kilowatts: 544, photoURL: "http://auto.ferrari.com/en\_EN/wp-content/uploads/sites/5/2013/07/Ferrari-F12berlinetta.jpg")

let zondaF = Car(model: "Zonda F", make: "Pagani", kilowatts: 449, photoURL: "http://storage.pagani.com/view/1024/BIG\_zg-4-def.jpg")

let lamboAventador = Car(model: "Aventador", make: "Lamborghini", kilowatts: 522, photoURL: "http://cdn.lamborghini.com/content/models/aventador\_lp700-4\_roadster/gallery\_2013/roadster\_21.jpg")

return [CarViewModel(car: ferrariF12), CarViewModel(car: zondaF), CarViewModel(car: lamboAventador)]

}()

// MVVMTests.swift

let ferrariF12 = Car(model: "F12", make: "Ferrari", kilowatts: 544, photoURL: "http://auto.ferrari.com/en\_EN/wp-content/uploads/sites/5/2013/07/Ferrari-F12berlinetta.jpg")

Bây giờ các bạn có thể chạy ứng dụng mà không gặp lỗi gì. Như các bạn biết thì hầu hết các ứng dụng sẽ có phần tương tác người dung, nó có nghĩa là Controller sẽ chịu trách nghiệm việc cập nhật View Model và nó sẽ tự động cập nhật Model của chính ViewModel đó, chúng ta sẽ sử dụng framework RxSwift để thực hiện việc bindling dữ liệu ở phần tiếp theo.

**Phần 2**

Ở phần 2 này chúng ta sẽ sử dụng RxSwift để thực hiện binding dữ liệu:

Chúng ta đã thấy những lợi ích của mô hình MVVM ở phần đầu tiên nhưng về phần hiển thị dữ liệu và việc tự cập nhật giao diện khi dữ liệu thay đổi thì chúng ta sẽ cùng tìm hiểu ở bài này.

Nên dùng gì?

Trong iOS thì điều đâu tiên tôi nghĩ đến là KVO(Key-Value Observing), nhưng chúng ta sẽ phải viết rất nhiều các khuôn mẫu với nhiều trường hợp vậy phải làm như nao?

Vì dụ tôi muốn sử dụng khái niệm lập trình Functional reactive programming. Nếu bạn không biết về nó thì bạn đơn giản bạn chỉ cần google cái là ra rất nhiều kết quả.

Có rất nhiều framework sử dụng tư tưởng FRP và trong đó có ReactiveCocoa và RxSwift.

Trước đây với nhưng dự án sử dụng Objective-C, tôi cũng có biết về ReactiveCocoa, nhưng với dự án về Swift thì tôi khuyên bạn nên sử dụng RxSwift và trong bài này chúng ta sẽ sử dụng RxSwift. Để giải thích việc nó hoạt động thì chúng ta sẽ đi thẳng vào ví dụ.

Cài đặt RxSwift

Chắc bạn đã quen thuộc với việc cài đặt một framework sử dụng pod file rồi:

Chạy pod install và ở file MVVM.xcworkspace.

Reactive với CarViewModel

Dĩ nhiên với vì dụ này chúng ta sẽ cho class CarViewModel reactive. Car model sẽ không thay đổi, nó sẽ không giao tiếp trực tiếp với các đối tượng UI, nó chỉ có tác dụng lưu dữ liệu.

Tiếp theo chúng ta sẽ làm việc với signals, observes, và observables, chúng ta cần chắc chắc cần clean các đối tượng khi không sử dụng nữa tránh trường hợp, một đối tượng không sử dụng nữa mà vẫn tồn tại nó sẽ là nguyên nhân gây ra hiện tượng memory leaks.

Điều đầu tiên chúng ta sẽ them vào một dispose bag.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import RxSwift  import RxCocoa    // class CarViewModel {  // ...      let disposeBag = DisposeBag()    // ...  // } |