Using MVVM with RxSwift

MVVM rather than MVC? Let’s go!

[](https://medium.com/@stevenpcurtis.sc?source=post_page-----baa38325b750----------------------)

[Steven Curtis](https://medium.com/@stevenpcurtis.sc?source=post_page-----baa38325b750----------------------)

[Dec 4](https://medium.com/swlh/using-mvvm-with-rxswift-baa38325b750?source=post_page-----baa38325b750----------------------) · 3 min read

This is a rather basic implementation of MVVM with RXSwift as the bindings, but is great for learning about those bindings. Fantastic!

Difficulty: Beginner | Easy| **Normal** | Challenging

**Prerequisites:**

* Some understanding of OO terminology and practices
* Some understanding of MVVM (or study [HERE](https://medium.com/swlh/what-goes-where-in-the-mvvm-architecture-for-ios-82d0febcddac))

**MVVM in Swift**

**MVVM rather than MVC? Let’s go!**

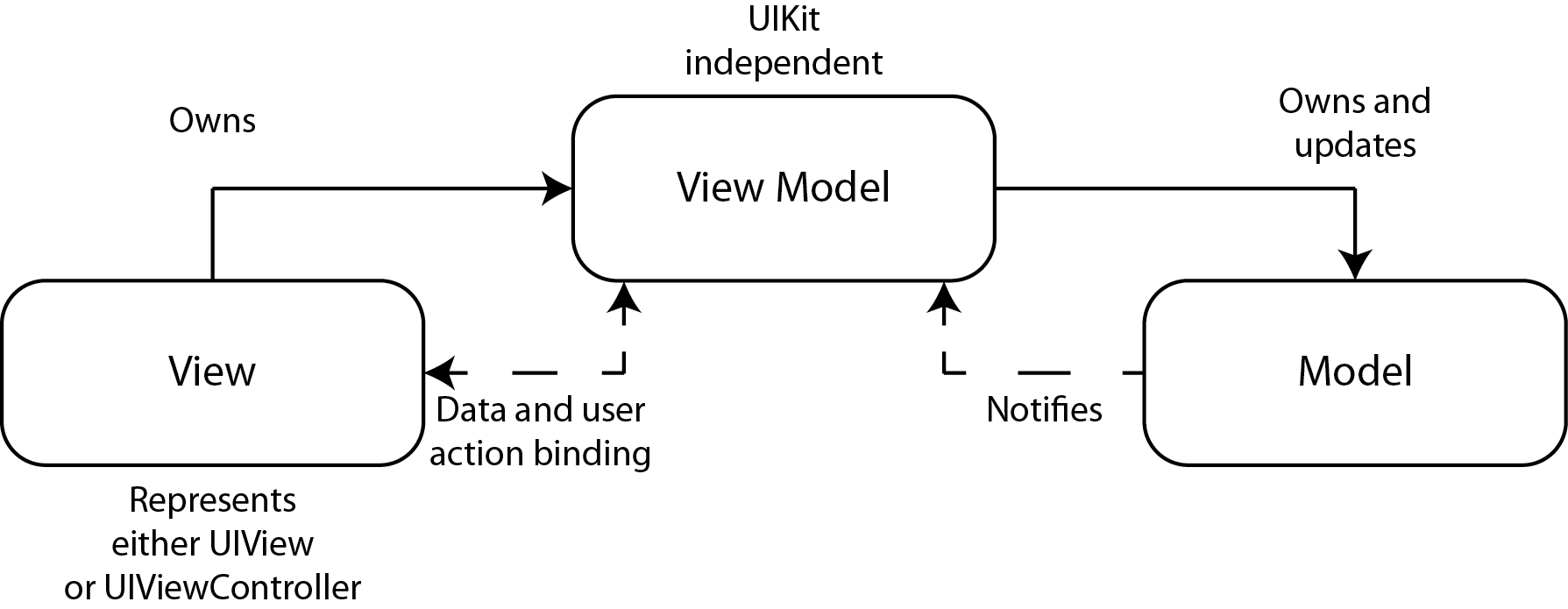
MVVM was proposed by John Gossman in 2005. Interestingly the view should consist only of visual elements.

All architectures have advantages and disadvantages, but MVVM has become increasingly popular in implementations.

**Prerequisites:**

* Some understanding of OO terminology and practices
* Swift’s Result type is used later in the post





**Terminology**

Binding: The mapping of one thing to another

Controller: Sits between the view and the model, tying them together (usually using the delegate pattern). The controller is not tightly bound to a concrete view, and communicates via a protocol to an abstraction. An example of this is the way that a UITableView communicates with its data source through the UITableViewDataSource protocol. Think of it as the **how** of the App. The primary job of the controller is to format the data from the model for the view to display

DisposeBag: Observable objects should be added to the DisposeBag when set up, so when deinit() is called each disposable observer is automatically unsubscribed. This prevents a retain cycle, or a crash from being deallocated while still being subscribed

Model: Where data, and logic that manipulates the data is stored. Perhaps model objects, or networking code is stored here.Think of this as the **what** of the App

Observable: An emitter of notifications of a change

Oberver: Subscribes to an observable to get notified when an observable has changed

Reactive Programming: A programming paradigm that is concerned with data streams and the propagation of change

Subscribe: The action of requesting to be notified when an observable has changed

View: Present information to the user. Views are, well, UIViews and their subclasses. Think of it as the UI components that have to be controlled by the controller

ViewModel: Contains fields that are to be displayed in the view

**The base Implementation**

<https://github.com/stevencurtis/SimpleMVVMNetwork> has a quite simple implementation of MVVM. However, it could be accused of actually being an MVP implementation as the bindings have been created with closures (and not even KVO! The following implementation uses the following API call: https://haveibeenpwned.com/api/v2/breaches

**HTTP Manager**

This makes the requests, and returns the data from a closure.

|  |  |
| --- | --- |
|  | class HTTPManager { |
|  | static let shared: HTTPManager = HTTPManager() |
|  |  |
|  | enum HTTPError: Error { |
|  | case invalidURL |
|  | case invalidResponse(Data?, URLResponse?) |
|  | } |
|  |  |
|  | public func get(urlString: String, completionBlock: @escaping (Result<Data, Error>) -> Void) { |
|  | guard let url = URL(string: urlString) else { |
|  | completionBlock(.failure(HTTPError.invalidURL)) |
|  | return |
|  | } |
|  |  |
|  | let task = URLSession.shared.dataTask(with: url) { data, response, error in |
|  | guard error == nil else { |
|  | completionBlock(.failure(error!)) |
|  | return |
|  | } |
|  |  |
|  | guard |
|  | let responseData = data, |
|  | let httpResponse = response as? HTTPURLResponse, |
|  | 200 ..< 300 ~= httpResponse.statusCode else { |
|  | completionBlock(.failure(HTTPError.invalidResponse(data, response))) |
|  | return |
|  | } |
|  |  |
|  | completionBlock(.success(responseData)) |
|  | } |
|  | task.resume() |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/stevencurtis/bf3c671f724fa27da63e7b1b5f8b5f4b/raw/038bce6e019b8a28006da8d031d1c67653530737/HTTPManager)

[**HTTPManager**](https://gist.github.com/stevencurtis/bf3c671f724fa27da63e7b1b5f8b5f4b#file-httpmanager) hosted with ❤ by [**GitHub**](https://github.com/)

**ViewModel**

The ViewModel fetches information and uses the model to which is then presented in the View Controller

|  |  |
| --- | --- |
|  | func fetchBreaches(completion: @escaping (Result<[BreachModel], Error>) -> Void) { |
|  | HTTPManager.shared.get(urlString: baseUrl + breachesExtensionURL, completionBlock: { [weak self] result in |
|  | guard let self = self else {return} |
|  | switch result { |
|  | case .failure(let error): |
|  | print ("failure", error) |
|  | case .success(let dta) : |
|  | let decoder = JSONDecoder() |
|  | do |
|  | { |
|  | self.breaches = try decoder.decode([BreachModel].self, from: dta) |
|  | completion(.success(try decoder.decode([BreachModel].self, from: dta))) |
|  | } catch { |
|  | // deal with error from JSON decoding if used in production |
|  | } |
|  | } |
|  | }) |
|  | } |

[**view raw**](https://gist.github.com/stevencurtis/ecdee78a2bd67ead0e2e1888aee45dab/raw/23b74bcc8a265aaf29444fbaa0b0c13bfd852058/fetchBreachesInViewModel)

[**fetchBreachesInViewModel**](https://gist.github.com/stevencurtis/ecdee78a2bd67ead0e2e1888aee45dab#file-fetchbreachesinviewmodel) hosted with ❤ by [**GitHub**](https://github.com/)

**View Controller**

The View Controller uses a closure

|  |  |
| --- | --- |
|  | breachesViewModel.fetchBreaches{ [weak self] breaches in |
|  | DispatchQueue.main.async { |
|  | self?.updateUI() |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/stevencurtis/03cd35a32078c90ffe2dad30a3df78ff/raw/1f07fed48c85193099f322efe463179d512d1875/fetchBreaches)

[**fetchBreaches**](https://gist.github.com/stevencurtis/03cd35a32078c90ffe2dad30a3df78ff#file-fetchbreaches) hosted with ❤ by [**GitHub**](https://github.com/)

**The RxSwift implementation**

I installed RxSwift using CocoaPods, although Carthage equally could have been used.

**Connect the View Controller to the View Model**

Rather than using a closure to retrieve the information from the view model, we can subscribe to changes that are made in the view model

|  |  |
| --- | --- |
|  | breachesViewModel.fetchBreachesRX() |
|  | .debug() |
|  | .subscribe({event in |
|  | switch event { |
|  | // next not used in this implementation |
|  | case .next: |
|  | print ("value provided") |
|  | case .error(let error): |
|  | print ("error", error) |
|  | case .completed: |
|  | DispatchQueue.main.async { |
|  | self.updateUI() |
|  | } |
|  | } |
|  | }).disposed(by: disposeBag) |

[**view raw**](https://gist.github.com/stevencurtis/3480ee4ab86dc5d057c3ea4cddda8c4e/raw/a7d36a9530314d78bfffd7fed3f178b070c2648e/fetchBreachesRX)

[**fetchBreachesRX**](https://gist.github.com/stevencurtis/3480ee4ab86dc5d057c3ea4cddda8c4e#file-fetchbreachesrx) hosted with ❤ by [**GitHub**](https://github.com/)

**Create an observable in the View Model**

Which is obviously matched with the observable in the view model

|  |  |
| --- | --- |
|  | func fetchBreachesRX() -> Observable<[BreachModel]> { |
|  | return Observable.create{ |
|  | observer -> Disposable in |
|  | HTTPManager.shared.get(urlString: baseUrl + breachesExtensionURL, completionBlock: { [weak self] result in |
|  | guard let self = self else {return} |
|  | switch result { |
|  | case .failure(let error): |
|  | print ("failure", error) |
|  | observer.onError(error) |
|  | case .success(let dta) : |
|  | let decoder = JSONDecoder() |
|  | do |
|  | { |
|  | self.breaches = try decoder.decode([BreachModel].self, from: dta) |
|  | observer.onNext(try decoder.decode([BreachModel].self, from: dta)) |
|  | observer.onCompleted() |
|  | } catch { |
|  | // deal with error from JSON decoding if used in production |
|  | } |
|  | } |
|  | }) |
|  | return Disposables.create() |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/stevencurtis/5262b0fef8bc1ee3ab9a480218dd5602/raw/4375f2ebfcd9e04988060d73148157c2793afd68/fetchBreachesRXViewModel)

[**fetchBreachesRXViewModel**](https://gist.github.com/stevencurtis/5262b0fef8bc1ee3ab9a480218dd5602#file-fetchbreachesrxviewmodel) hosted with ❤ by [**GitHub**](https://github.com/)

**Dispose bag**

There is one last thing that needs to be created here is the disposeBag in the view controller.

This ensures that the memory is properly deallocated (and is paired with Disposables.create() when the variable is set up)

***var*** *disposeBag = DisposeBag()*

**Conclusion**

The idea behind this particular tutorial is to give an *example* of how RxSwift can be used with MVVM. It does not promise to be perfect, but rather to give an idea for people trying to get ahead in the implementation of their apps.

That said, the best way to look at this example is to download the full code. That code is in the link just below!

The full code is available here:

[**stevencurtis**](https://github.com/stevencurtis/MVVMWithRXSwift?source=post_page-----baa38325b750----------------------)