Clean Architecture and MVVM on iOS

Oleh Kudinov · Follow
Published in OLX Engineering
10 min read · Aug 22, 2019

Listen Share

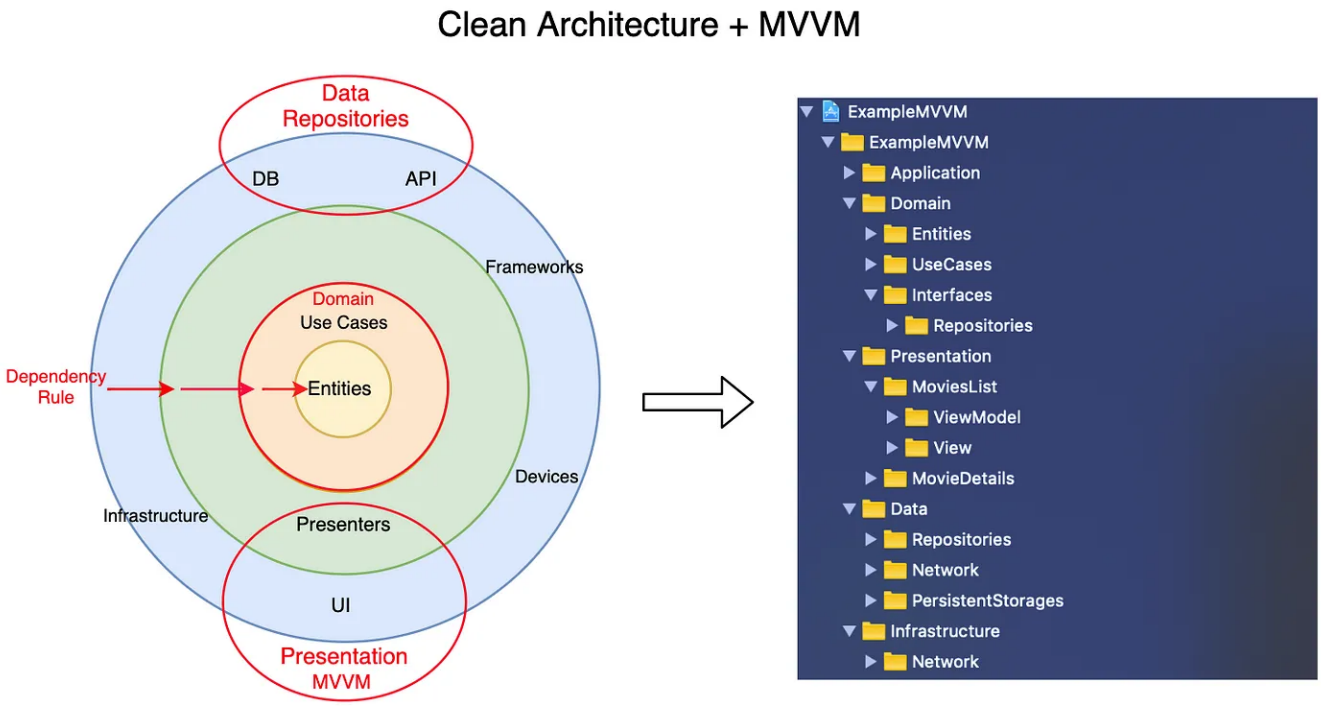
Photo by Joel Filipe on Unsplash

When we develop software it is important to not only use design patterns, but also architectural patterns. There are many different architectural patterns in Software Engineering. In mobile software engineering, the most widely used are MVVM, Clean Architecture and Redux patterns.

In this article, we will show on a **working example project** how architectural patterns **MVVM** and **Clean Architecture** can be applied in an iOS app.

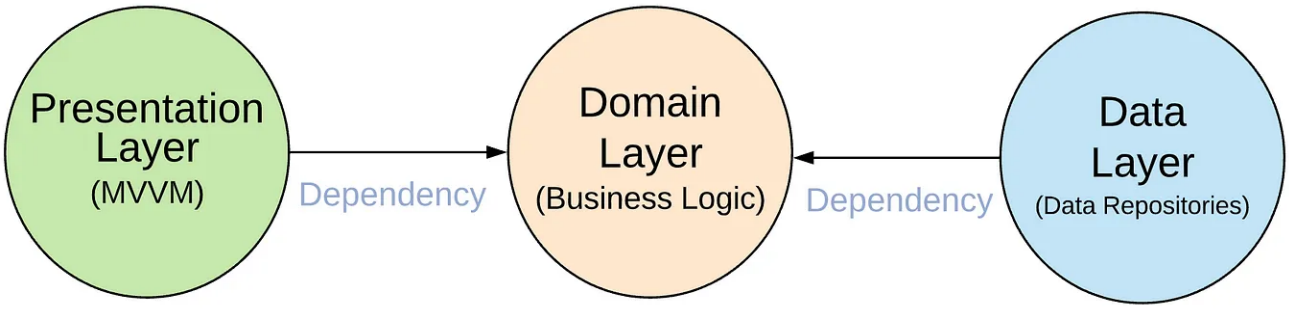
If you are also interested in learning about Redux, check out this book: **Advanced iOS App Architecture**.

More information about **Clean Architecture**.



As we can see in **Clean Architecture** graph, we have different layers in the application. The main rule is *not to have dependencies from inner layers to outers layers*. The arrows pointing from outside to inside is the **Dependency rule**. There can only be dependencies from outer layer inward.

After grouping all layers we have: **Presentation, Domain and Data layers.**



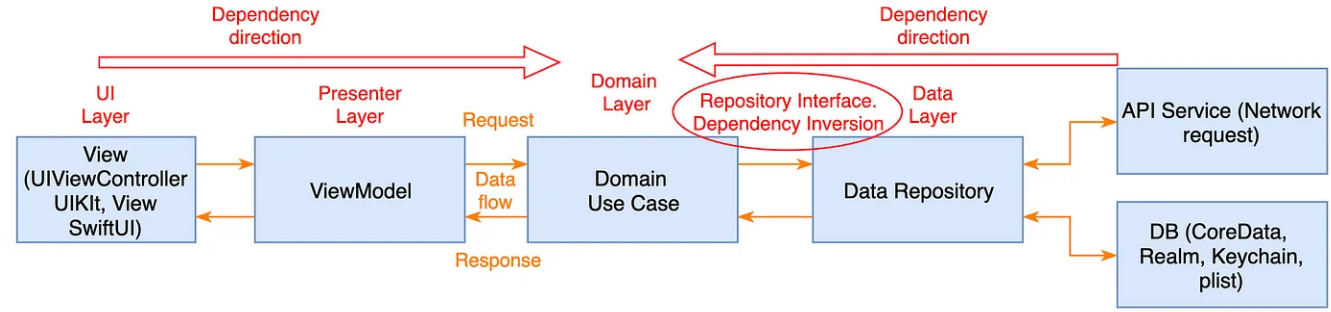
Domain Layer (Business logic) is the inner-most part of the onion (without dependencies to other layers, it is totally isolated). It contains *Entities(Business Models), Use Cases, and Repository Interfaces.* This layer could be potentially reused within different projects. Such separation allows for not using the host app within the test target because **no dependencies (also 3rd party) are needed** — this makes the Domain Use Cases tests take just a few seconds. **Note:** Domain Layer **should not include** anything from other layers(e.g *Presentation — UIKit or SwiftUI* or Data Layer — Mapping *Codable*)

The reason that good architecture is centered around *Use Cases* is so that architects can safely describe the structures that support those *Use cases* without committing to frameworks, tools, and environment. It is called **Screaming Architecture**.

Presentation Layer contains *UI (UIViewControllers or SwiftUI Views). Views* are coordinated by *ViewModels (Presenters) which execute one or many Use Cases.* Presentation Layer **depends only** on the **Domain Layer**.

Data Layer contains *Repository Implementations and one or many Data Sources.* Repositories are responsible for coordinating data from different Data Sources. Data Source can be Remote or Local (for example persistent database). Data Layer **depends only on** the **Domain Layer**. In this layer, we can also add mapping of Network JSON Data (e.g. **Decodable** conformance) to Domain Models.

On the graph below every component from each layer is represented along with **Dependency Direction** and also the **Data flow** (Request/Response). We can see the **Dependency Inversion** point where we use Repository interfaces(protocols). Each layer's explanation will be based on the example project mentioned at the beginning of the article.



Data Flow

*1. **View**(UI) calls method from **ViewModel** (Presenter).*

*2. **ViewModel** executes **Use Case**.*

*3. **Use Case** combines data from **User** and **Repositories**.*

*4. Each **Repository** returns data from a **Remote Data** (Network), **Persistent DB** Storage Source or In-memory Data (Remote or Cached).*

*5. Information flows back to the **View**(UI) where we display the list of items.*

Dependency Direction

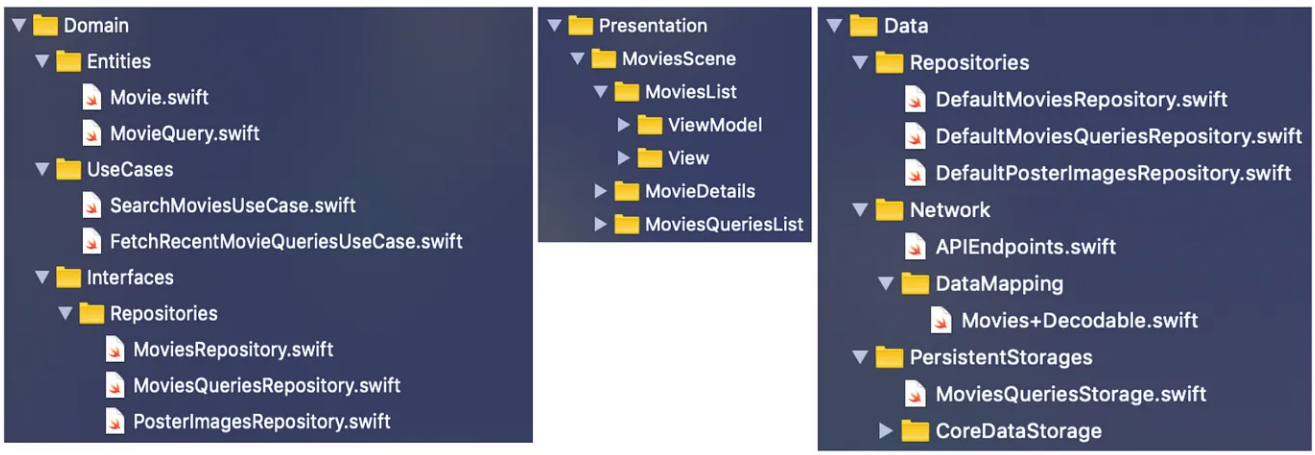
Presentation Layer -> Domain Layer <- Data Repositories Layer

***Presentation Layer (MVVM)** = ViewModels(Presenters) + Views(UI)*

***Domain Layer** = Entities + Use Cases + Repositories Interfaces*

***Data Repositories Layer** = Repositories Implementations + API(Network) + Persistence DB*

Example Project: "Movies App"



Domain Layer

Inside the example project you can find Domain Layer. It contains Entities, SearchMovies**UseCase** which searches movies and stores recent successful queries. Also, it contains *Data Repositories Interfaces* which are needed for **Dependency Inversion.**

```
protocol SearchMoviesUseCase {
    func execute(requestValue: SearchMoviesUseCaseRequestValue,
                 completion: @escaping (Result<MoviesPage, Error>) -> Void) -> Cancella
}

final class DefaultSearchMoviesUseCase: SearchMoviesUseCase {

    private let moviesRepository: MoviesRepository
    private let moviesQueriesRepository: MoviesQueriesRepository

    init(moviesRepository: MoviesRepository, moviesQueriesRepository: MoviesQueriesRepo
        self.moviesRepository = moviesRepository
        self.moviesQueriesRepository = moviesQueriesRepository
    }

    func execute(requestValue: SearchMoviesUseCaseRequestValue,
                 completion: @escaping (Result<MoviesPage, Error>) -> Void) -> Cancella
        return moviesRepository.fetchMoviesList(query: requestValue.query, page: reques

            if case .success = result {
                self.moviesQueriesRepository.saveRecentQuery(query: requestValue.query)
            }

            completion(result)
        }
    }
}

// Repository Interfaces
protocol MoviesRepository {
    func fetchMoviesList(query: MovieQuery, page: Int, completion: @escaping (Result<Mo
}

protocol MoviesQueriesRepository {
    func fetchRecentsQueries(maxCount: Int, completion: @escaping (Result<[MovieQuery],
    func saveRecentQuery(query: MovieQuery, completion: @escaping (Result<MovieQuery, E
}
```

UseCaseExample.swift hosted with ❤️ by GitHub view raw

Note: Another way to create Use Cases is to use *UseCase* protocol with *start()* function and all use cases implementations will conform to this protocol. One of use cases in the example project follows this approach: FetchRecentMovieQueriesUseCase. Use Cases are also called **Interactors**

Note: A *UseCase* can depend on other *UseCases*

Presentation Layer

Presentation Layer contains MoviesList**ViewModel** with items that are observed from the MoviesList**View**. MoviesList**ViewModel** does not import UIKit. Because keeping the ViewModel clean from any UI frameworks like UIKit, SwiftUI or WatchKit will allow for easy reuse and refactor. For example in future the **Views** refactor from UIKit to SwiftUI will be much easier, because the **ViewModel** will not need to change.

```swift
// Note: We cannot have any UI frameworks(like UIKit or SwiftUI) imports here.

protocol MoviesListViewModelInput {
    func didSearch(query: String)
    func didSelect(at indexPath: IndexPath)
}

protocol MoviesListViewModelOutput {
    var items: Observable<[MoviesListItemViewModel]> { get }
    var error: Observable<String> { get }
}

protocol MoviesListViewModel: MoviesListViewModelInput, MoviesListViewModelOutput { }

struct MoviesListViewModelActions {
    // Note: if you would need to edit movie inside Details screen and update this
    // MoviesList screen with Updated movie then you would need this closure:
    //  showMovieDetails: (Movie, @escaping (_ updated: Movie) -> Void) -> Void
    let showMovieDetails: (Movie) -> Void
}

final class DefaultMoviesListViewModel: MoviesListViewModel {

    private let searchMoviesUseCase: SearchMoviesUseCase
    private let actions: MoviesListViewModelActions?

    private var movies: [Movie] = []

    // MARK: - OUTPUT
    let items: Observable<[MoviesListItemViewModel]> = Observable([])
    let error: Observable<String> = Observable("")

    init(searchMoviesUseCase: SearchMoviesUseCase,
         actions: MoviesListViewModelActions) {
        self.searchMoviesUseCase = searchMoviesUseCase
        self.actions = actions
    }

    private func load(movieQuery: MovieQuery) {

        searchMoviesUseCase.execute(movieQuery: movieQuery) { result in
            switch result {
            case .success(let moviesPage):
                // Note: We must map here from Domain Entities into Item View Models. S
                self.items.value += moviesPage.movies.map(MoviesListItemViewModel.init)
                self.movies += moviesPage.movies
            case .failure:
                self.error.value = NSLocalizedString("Failed loading movies", comment:
```

```
                self.error.value = NSLocalizedString("Failed loading movies", comment:
            }
        }
    }
}

// MARK: - INPUT. View event methods
extension MoviesListViewModel {

    func didSearch(query: String) {
        load(movieQuery: MovieQuery(query: query))
    }

    func didSelect(at indexPath: IndexPath) {
        actions?.showMovieDetails(movies[indexPath.row])
    }
}

// Note: This item view model is to display data and does not contain any domain model
struct MoviesListItemViewModel: Equatable {
    let title: String
}

extension MoviesListItemViewModel {
    init(movie: Movie) {
        self.title = movie.title ?? ""
    }
}
```

Note: We use interfaces MoviesList**ViewModelInput** and MoviesList**ViewModelOutput** to make MoviesList**View**Controller testable, by mocking **ViewModel** easily(*example*). Also, we have MoviesListViewModel**Actions** closures, which tells to MoviesSearch**FlowCoordinator** when to present another views. When action closure is called coordinator will present movie details screen. We use a struct to group actions because we can add later easily more actions if needed.

Presentation Layer also contains MoviesList**View**Controller which is **bound to data**(items) of MoviesList**ViewModel**.

UI cannot have access to business logic or application logic (Business Models and UseCases), only ViewModels can do it. This is the **separation of concerns**. We cannot

pass business models directly to the View (UI). This why we are mapping Business Models into ViewModel inside ViewModel and pass them to the View.

We also add a search event call from the View to ViewModel to start searching movies:

```swift
import UIKit

final class MoviesListViewController: UIViewController, StoryboardInstantiable, UISearc

    private var viewModel: MoviesListViewModel!

    final class func create(with viewModel: MoviesListViewModel) -> MoviesListViewContr
        let vc = MoviesListViewController.instantiateViewController()
        vc.viewModel = viewModel
        return vc
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        bind(to: viewModel)
    }

    private func bind(to viewModel: MoviesListViewModel) {
        viewModel.items.observe(on: self) { [weak self] items in
            self?.moviesTableViewController?.items = items
        }
        viewModel.error.observe(on: self) { [weak self] error in
            self?.showError(error)
        }
    }

    func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
        guard let searchText = searchBar.text, !searchText.isEmpty else { return }
        viewModel.didSearch(query: searchText)
    }
}
```

ViewExample.swift hosted with ❤️ by GitHub view raw

Note: We observe items and reload view when they change. We use here a simple *Observable*, which is explained in MVVM section below.

We also assign function *showMovieDetails*(movie:) **to Actions of** MoviesList**ViewModel** inside MoviesSearch**FlowCoordinator**, to present movie details screens from flow coordinator:

```
protocol MoviesSearchFlowCoordinatorDependencies  {
    func makeMoviesListViewController() -> UIViewController
    func makeMoviesDetailsViewController(movie: Movie) -> UIViewController
}

final class MoviesSearchFlowCoordinator {

    private weak var navigationController: UINavigationController?
    private let dependencies: MoviesSearchFlowCoordinatorDependencies

    init(navigationController: UINavigationController,
         dependencies: MoviesSearchFlowCoordinatorDependencies) {
        self.navigationController = navigationController
        self.dependencies = dependencies
    }

    func start() {
        // Note: here we keep strong reference with actions closures, this way this flo
        let actions = MoviesListViewModelActions(showMovieDetails: showMovieDetails)
        let vc = dependencies.makeMoviesListViewController(actions: actions)

        navigationController?.pushViewController(vc, animated: false)
    }

    private func showMovieDetails(movie: Movie) {
        let vc = dependencies.makeMoviesDetailsViewController(movie: movie)
        navigationController?.pushViewController(vc, animated: true)
    }
}
```

FlowCoordinator.swift hosted with ❤️ by GitHub view raw

Note: We use **Flow Coordinator** for presentation logic, to reduce View Controllers' size and **responsibility**. We have **strong** reference to **Flow** (with action closures, self functions) to keep **Flow** not deallocated while is needed.

With this approach, we easily can use different views with the same ViewModel without modifying it. We could just check if iOS 13.0 is available and then create a SwiftUI View instead of UIKit and bind it to the same ViewModel otherwise we

create UIKit View. In the example project I also added SwiftUI example for MoviesQueriesSuggestionsList. *A*t least Xcode 11 Beta is required.

```
// MARK: - Movies Queries Suggestions List
func makeMoviesQueriesSuggestionsListViewController(didSelect: @escaping MoviesQueryList
    if #available(iOS 13.0, *) { // SwiftUI
        let view = MoviesQueryListView(viewModelWrapper: makeMoviesQueryListViewModelWrap
        return UIHostingController(rootView: view)
    } else { // UIKit
        return MoviesQueriesTableViewController.create(with: makeMoviesQueryListViewMode
    }
}
```

MoviesSceneDIContainer.swift hosted with ❤️ by GitHub view raw

Data Layer

Data Layer contains DefaultMovies**Repository.** It conforms to interfaces defined inside Domain Layer (**Dependency Inversion**). We also add here the mapping of JSON data(Decodable conformance) and CoreData Entities to Domain Models.

```
final class DefaultMoviesRepository {

    private let dataTransferService: DataTransfer

    init(dataTransferService: DataTransfer) {
        self.dataTransferService = dataTransferService
    }
}

extension DefaultMoviesRepository: MoviesRepository {

    public func fetchMoviesList(query: MovieQuery, page: Int, completion: @escaping (Re

        let endpoint = APIEndpoints.getMovies(with: MoviesRequestDTO(query: query.query
                                                                     page: page))
        return dataTransferService.request(with: endpoint) { (response: Result<MoviesRe
            switch response {
            case .success(let moviesResponseDTO):
                completion(.success(moviesResponseDTO.toDomain()))
            case .failure(let error):
                completion(.failure(error))
            }
        }
    }
}

// MARK: - Data Transfer Object (DTO)
// It is used as intermediate object to encode/decode JSON response into domain, inside
struct MoviesRequestDTO: Encodable {
    let query: String
    let page: Int
}

struct MoviesResponseDTO: Decodable {
    private enum CodingKeys: String, CodingKey {
        case page
        case totalPages = "total_pages"
        case movies = "results"
    }
    let page: Int
    let totalPages: Int
    let movies: [MovieDTO]
}
...
// MARK: - Mappings to Domain

extension MoviesResponseDTO {
    func toDomain() -> MoviesPage {
```

```
49          return .init(page: page,
50                       totalPages: totalPages,
51                       movies: movies.map { $0.toDomain() })
52      }
53  }
54  ...
```

DataRepository.swift hosted with ❤ by GitHub view raw

Note: Data Transfer Objects DTO is used as intermediate object for mapping from JSON response into Domain. Also if we want to cache endpoint response we would store Data Transfer Objects in persistent storage by mapping them into Persistent objects(e.g. DTO -> NSManagedObject).

In general Data Repositories can be injected with API Data Service and with Persistent Data Storage. Data Repository works with these two dependencies to return data. The rule is to first ask persistent storage for cached data output (NSManagedObject are mapped into Domain via DTO object, and retrieved in *cached data closure*). Then to call API Data Service which will return the latest updated data. Then Persistent Storage is updated with this latest data (DTOs are mapped into Persistent Objects and saved). And then DTO is mapped into Domain and retrieved in *updated data/completion closure*. This way users will see the data instantaneously. Even if there is no internet connection, users still will see the latest data from Persistent Storage. *example*

The storage and API can be replaced by totally different implementations (from CoreData to Realm for example). While all the rest layers of the app will not be affected by this change, this is because DB is a detail.

Infrastructure Layer (Network)

It is a wrapper around network framework, it can be Alamofire (or another framework). It can be configured with network parameters (for example base URL). It also supports defining endpoints and contains data mapping methods (using Decodable).

```swift
struct APIEndpoints {

    static func getMovies(with moviesRequestDTO: MoviesRequestDTO) -> Endpoint<MoviesRe

        return Endpoint(path: "search/movie/",
                        method: .get,
                        queryParametersEncodable: moviesRequestDTO)
    }
}


let config = ApiDataNetworkConfig(baseURL: URL(string: appConfigurations.apiBaseURL)!,
                                  queryParameters: ["api_key": appConfigurations.apiKey
let apiDataNetwork = DefaultNetworkService(session: URLSession.shared,
                                           config: config)

let endpoint = APIEndpoints.getMovies(with: MoviesRequestDTO(query: query.query,
                                                             page: page))
dataTransferService.request(with: endpoint) { (response: Result<MoviesResponseDTO, Erro
    let moviesPage = try? response.get()
}
```

EndpointExample.swift hosted with ❤️ by GitHub view raw

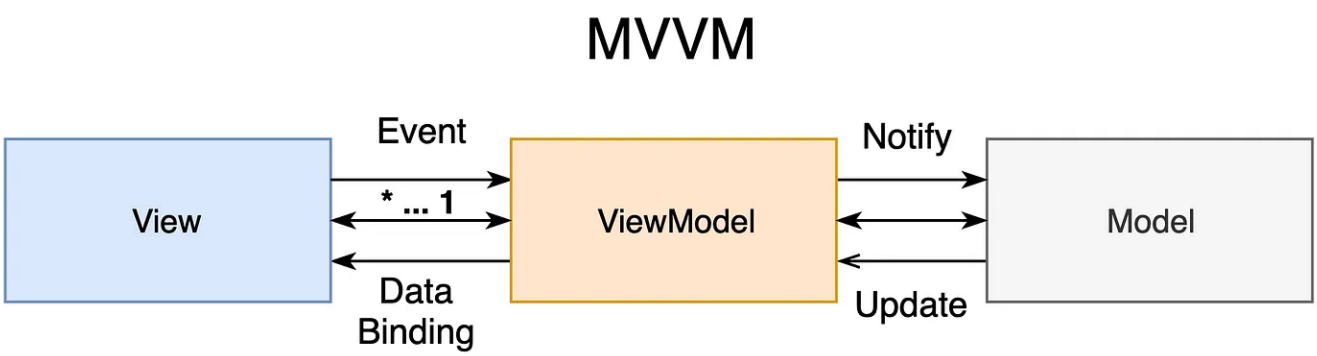
Note: You can read more here: https://github.com/kudoleh/SENetworking

MVVM

The **Model-View-ViewModel** pattern (MVVM) provides a clean separation of concerns between the UI and Domain.

When used together with Clean Architecture it can help to separate concerns between Presentation and UI Layers.

Different view implementations can be used with the same ViewModel. For example, you can use CarsAroundList**View** and CarsAroundMap**View** and use CarsAround**ViewModel** for both. You can also implement one View UIKit and another View with SwiftUI. It is important to remember to not import UIKit, WatchKit and SwiftUI inside your ViewModel. This way it could be easily reused in other platforms if needed.



Data Binding between **View** and **ViewModel** can be done for example with closures, delegates or observables (e.g. RxSwift). Combine and SwiftUI also can be used but only if your minimum supported iOS system is 13. The **View** has a direct relationship to **ViewModel** and notifies it whenever an event inside View happens. From ViewModel, there is no direct reference to View (only Data Binding)

In this example, we will use a simple combination of Closure and didSet to avoid third-party dependencies:

```swift
public final class Observable<Value> {

    private var closure: ((Value) -> ())?

    public var value: Value {
        didSet { closure?(value) }
    }

    public init(_ value: Value) {
        self.value = value
    }

    public func observe(_ closure: @escaping (Value) -> Void) {
        self.closure = closure
        closure(value)
    }
}
```

Observable.swift hosted with ❤️ by GitHub view raw

Note: This is a very simplified version of Observable, to see the full implementation with multiple observers and observer removal: **Observable**.

An example of data binding from ViewController:

```swift
final class ExampleViewController: UIViewController {

    private var viewModel: MoviesListViewModel!

    private func bind(to viewModel: ViewModel) {
        self.viewModel = viewModel
        viewModel.items.observe(on: self) { [weak self] items in
            self?.tableViewController?.items = items
            // Important: You cannot use viewModel inside this closure, it will cause r
            // self?.tableViewController?.items = viewModel.items.value // This would b
        }
        // Or in one line
        viewModel.items.observe(on: self) { [weak self] in self?.tableViewController?.i
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        bind(to: viewModel)
        viewModel.viewDidLoad()
    }
}


protocol ViewModelInput {
    func viewDidLoad()
}

protocol ViewModelOutput {
    var items: Observable<[ItemViewModel]> { get }
}

protocol ViewModel: ViewModelInput, ViewModelOutput {}
```

ViewModelBindingExample.swift hosted with ❤️ by GitHub view raw

Note: Accessing viewModel from observing closure is not allowed, it causes a retain cycle(memory leak). You can access viewModel only with self: self?.viewModel.

An example of data binding on TableViewCell (Reusable Cell):

```swift
final class MoviesListItemCell: UITableViewCell {

    private var viewModel: MoviesListItemViewModel! { didSet { unbind(from: oldValue) }

    func fill(with viewModel: MoviesListItemViewModel) {
        self.viewModel = viewModel
        bind(to: viewModel)
    }

    private func bind(to viewModel: MoviesListItemViewModel) {
        viewModel.posterImage.observe(on: self) { [weak self] in self?.imageView.image
    }

    private func unbind(from item: MoviesListItemViewModel?) {
        item?.posterImage.remove(observer: self)
    }
}
```

Unbinding.swift hosted with ❤️ by GitHub view raw

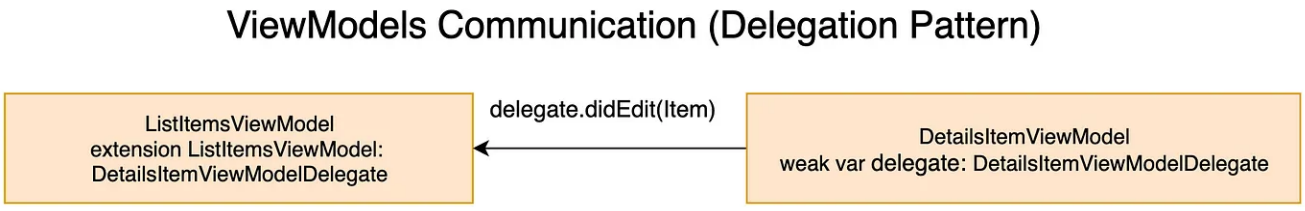
Note: We have to unbind if the view is reusable (e.g. UITableViewCell)

MVVM Templates can be found here

MVVMs Communication

Delegation

ViewModel of one MVVM(screen) communicates with another ViewModel of another MVVM(screen) using delegation pattern:



For example, we have ItemsList**ViewModel** and ItemEdit**ViewModel.** Then we create a protocol ItemEdit**ViewModelDelegate** with method ItemEditViewModelDidEditItem(item). And we make it conform to this protocol: extension ListItemsViewModel: ItemEditViewModel**Delegate**

```swift
// Step 1: Define delegate and add it to first ViewModel as weak property
protocol MoviesQueryListViewModelDelegate: class {
    func moviesQueriesListDidSelect(movieQuery: MovieQuery)
}
...
final class DefaultMoviesQueryListViewModel: MoviesListViewModel {
    private weak var delegate: MoviesQueryListViewModelDelegate?

    func didSelect(item: MoviesQueryListViewItemModel) {
        // Note: We have to map here from View Item Model to Domain Enity
        delegate?.moviesQueriesListDidSelect(movieQuery: MovieQuery(query: item.query))
    }
}

// Step 2:  Make second ViewModel to conform to this delegate
extension MoviesListViewModel: MoviesQueryListViewModelDelegate {
    func moviesQueriesListDidSelect(movieQuery: MovieQuery) {
        update(movieQuery: movieQuery)
    }
}
```

ViewModelsCommunication.swift hosted with ❤️ by GitHub view raw

Note: We can also name Delegates in this case as Responders: ItemEditViewModel**Responder**

Closures

Another way to communicate is by using closures which are assigned or injected by FlowCoordinator. In the example project we can see how MoviesListViewModel uses action **closure** *showMovieQueriesSuggestions* to show the MoviesQueriesSuggestions**View**. It also passes parameter *(_ didSelect: MovieQuery) -> Void* so it can be called back from that **View**. The communication is connected inside MoviesSearch**FlowCoordinator**:

```swift
// MoviesQueryList.swift

// Step 1: Define action closure to communicate to another ViewModel, e.g. here we noti
typealias MoviesQueryListViewModelDidSelectAction = (MovieQuery) -> Void

// Step 2: Call action closure when needed
class MoviesQueryListViewModel {
    init(didSelect: MoviesQueryListViewModelDidSelectAction? = nil) {
        self.didSelect = didSelect
    }
    func didSelect(item: MoviesQueryListItemViewModel) {
        didSelect?(MovieQuery(query: item.query))
    }
}

// MoviesQueryList.swift

// Step 3: When presenting MoviesQueryListView we need to pass this action closure as p
struct MoviesListViewModelActions {
    let showMovieQueriesSuggestions: (@escaping (_ didSelect: MovieQuery) -> Void) -> V
}

class MoviesListViewModel {
    var actions: MoviesListViewModelActions?

    func showQueriesSuggestions() {
        actions?.showMovieQueriesSuggestions { self.update(movieQuery: $0) }
        //or simpler actions?.showMovieQueriesSuggestions(update)
    }
}

// FlowCoordinator.swift

// Step 4: Inside FlowCoordinator we connect communication of two viewModels, by inject
class MoviesSearchFlowCoordinator {
    func start() {
        let actions = MoviesListViewModelActions(showMovieQueriesSuggestions: self.show
        let vc = dependencies.makeMoviesListViewController(actions: actions)
        present(vc)
    }

    private func showMovieQueriesSuggestions(didSelect: @escaping (MovieQuery) -> Void)
        let vc = dependencies.makeMoviesQueriesSuggestionsListViewController(didSelect:
        present(vc)
    }
}
```

ViewModelsCommunicationsActionsClosures.swift hosted with ❤ by GitHub view raw

Layer Separation into frameworks (Modules)

Now each layer (Domain, Presentation, UI, Data, Infrastructure Network) of the example app can be easily separated into separate frameworks.

```
New Project -> Create Project… -> Cocoa Touch Framework
```

Then you can include these frameworks into your main app by using CocoaPods. You can see this working example here. **Note:** you will need to delete **ExampleMVVM.xcworkspace** and run **pod install** to generate a new one, because of a permission issue.

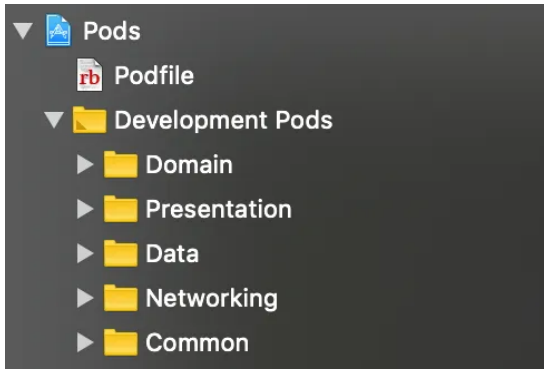
```
platform :ios, '10.0'

target 'ExampleMVVM' do
  use_frameworks!

  pod 'Domain', :path => 'Modules/Domain', :testspecs => ['Tests']
  pod 'Presentation', :path => 'Modules/Presentation', :testspecs => ['Tests']
  pod 'Data', :path => 'Modules/Data'
  pod 'Networking', :path => 'Modules/Networking', :testspecs => ['Tests']
  pod 'Common', :path => 'Modules/Common'

  target 'ExampleMVVMUITests' do
  end

end
```



Dependency Injection Container

Dependency injection is a technique whereby one object supplies the dependencies of another object. DIContainer in your application is the central unit of all injections.

Using dependencies factory protocols

One of the options is to declare a dependencies protocol that delegates the creation of dependency to DIContainer. To do this we need to define MoviesSearchFlowCoordinator**Dependencies** protocol and make your MoviesScene**DIContainer** to conform to this protocol, and then inject it into the MoviesSearch**FlowCoordinator** that needs this injection to create and present MoviesList**View**Controller. Here are the steps:

```swift
// Define Dependencies protocol for class or struct that needs it
protocol MoviesSearchFlowCoordinatorDependencies  {
    func makeMoviesListViewController() -> MoviesListViewController
}

class MoviesSearchFlowCoordinator {

    private let dependencies: MoviesSearchFlowCoordinatorDependencies

    init(dependencies: MoviesSearchFlowCoordinatorDependencies) {
        self.dependencies = dependencies
    }
...
}

// Make the DIContainer to conform to this protocol
extension MoviesSceneDIContainer: MoviesSearchFlowCoordinatorDependencies {}

// And inject MoviesSceneDIContainer `self` into class that needs it
final class MoviesSceneDIContainer {
    ...
    // MARK: - Flow Coordinators
    func makeMoviesSearchFlowCoordinator(navigationController: UINavigationController)
        return MoviesSearchFlowCoordinator(navigationController: navigationController,
                                           dependencies: self)
    }
}
```

DIContainerWithFactoryProtocols.swift hosted with ❤️ by GitHub view raw

Using closures

Another option is to use closures. To do it we need to declare closure inside the class that needs an injection and then we inject this closure. For example:

```swift
// Define makeMoviesListViewController closure that returns MoviesListViewController
class MoviesSearchFlowCoordinator {

    private var makeMoviesListViewController: () -> MoviesListViewController

    init(navigationController: UINavigationController,
         makeMoviesListViewController: @escaping () -> MoviesListViewController) {
        ...
        self.makeMoviesListViewController = makeMoviesListViewController
    }
    ...
}

// And inject MoviesSceneDIContainer's `self`.makeMoviesListViewController function int
final class MoviesSceneDIContainer {
    ...
    // MARK: - Flow Coordinators
    func makeMoviesSearchFlowCoordinator(navigationController: UINavigationController)
        return MoviesSearchFlowCoordinator(navigationController: navigationController,
                                           makeMoviesListViewController: self.makeMovie
    }

    // MARK: - Movies List
    func makeMoviesListViewController() -> MoviesListViewController {
        ...
    }
}
```

DIContainerWithClosures.swift hosted with ❤️ by GitHub view raw

Source code

kudoleh/iOS-Clean-Architecture-MVVM

iOS Project implemented with Clean Layered Architecture and MVVM. (Can be used as Template project by replacing item...

github.com

Companies with many iOS Engineers

Clean Architecture + MVVM is successfully used at fintech company **Revolut** with >70 iOS engineers.

Resources

Advanced iOS App Architecture

The Clean Architecture

The Clean Code

Conclusion

MVVM and Clean Architecture can be used separately of course, but MVVM provides separation of concerns only inside the Presentation Layer, whereas Clean Architecture splits your code into modular layers that can be easily **tested, reused**, and **understood**.

It is important to not skip creation of a Use Case, even if the Use Case does nothing else besides calling Repository. This way, your architecture will be self-explanatory when a new developer sees your Use cases.

Although this should be useful as a starting point, **there are no silver bullets.** You pick the architecture that fulfills your needs in the project.

Clean Architecture works really good with (Test Driven Development) TDD. This architecture makes the project testable and layers can be replaced easily (UI and Data).

Domain-Driven Design (DDD) also works very well with Clean Architecture (CA).

In software development there are more different architectures interesting to know: The 5 Patterns You Need to Know

More software engineering best practices:

- *Do not write code without tests (try TDD)*
- *Do continuous refactoring*
- *Do not over-engineer and be pragmatic*
- *Avoid using third-party framework dependencies in your project as much as you can*

More on Mobile Architecture topic

Modular Architecture

How can you improve your project by decoupling your app into totally isolated modules (e.g. **NetworkingService, TrackingService, ChatFeature, PaymentFeature…**)? And how can all teams work with these modules rapidly and independently 🤔?

Check out the tech article about Modular Architecture: modularisation of the app

Modular Architecture in iOS

In the previous article, we have seen how to create an app using Clean Architecture + MVVM. Here we show how to improve…

tech.olx.com

IOS Mvvm Swift Clean Architecture Software Architecture

Follow

Written by Oleh Kudinov

663 Followers · Writer for OLX Engineering

More from Oleh Kudinov and OLX Engineering

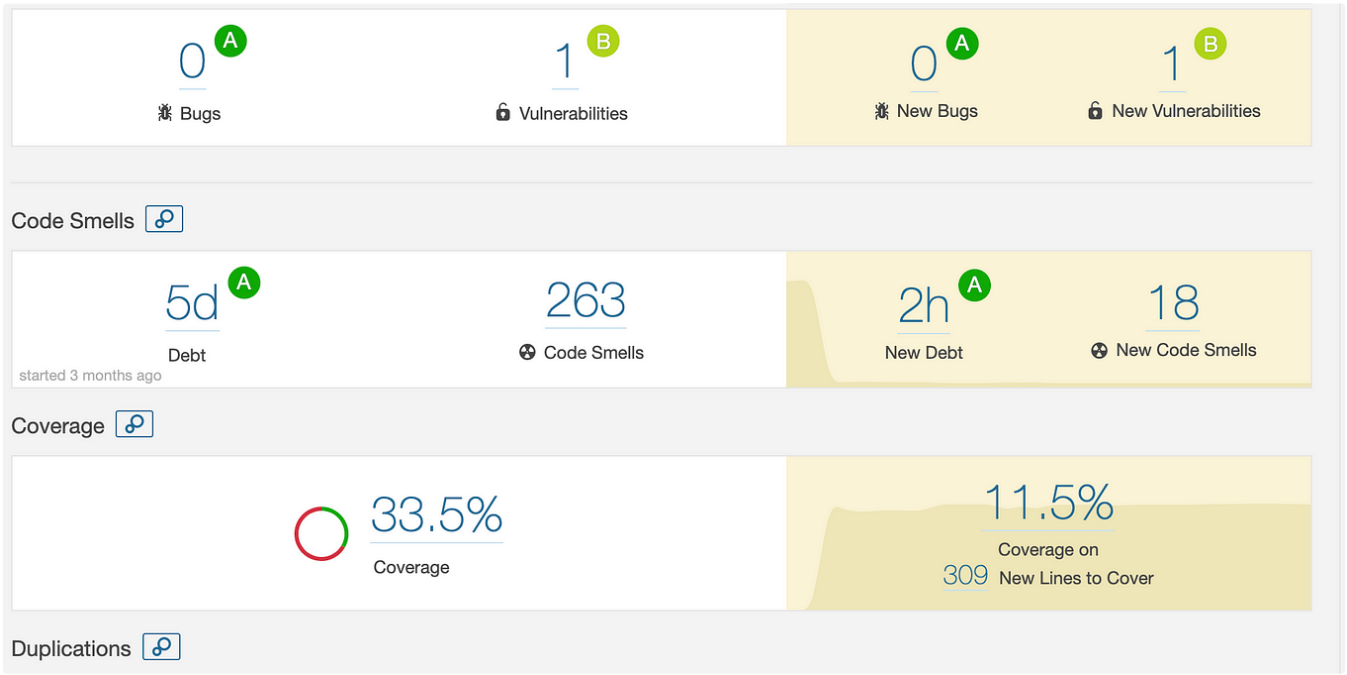
Oleh Kudinov in OLX Engineering

Modular Architecture in iOS

In the previous article, we have seen how to create an app using Clean Architecture + MVVM. Here we show how to improve your project by...

12 min read · Dec 18, 2019

1.4K 15



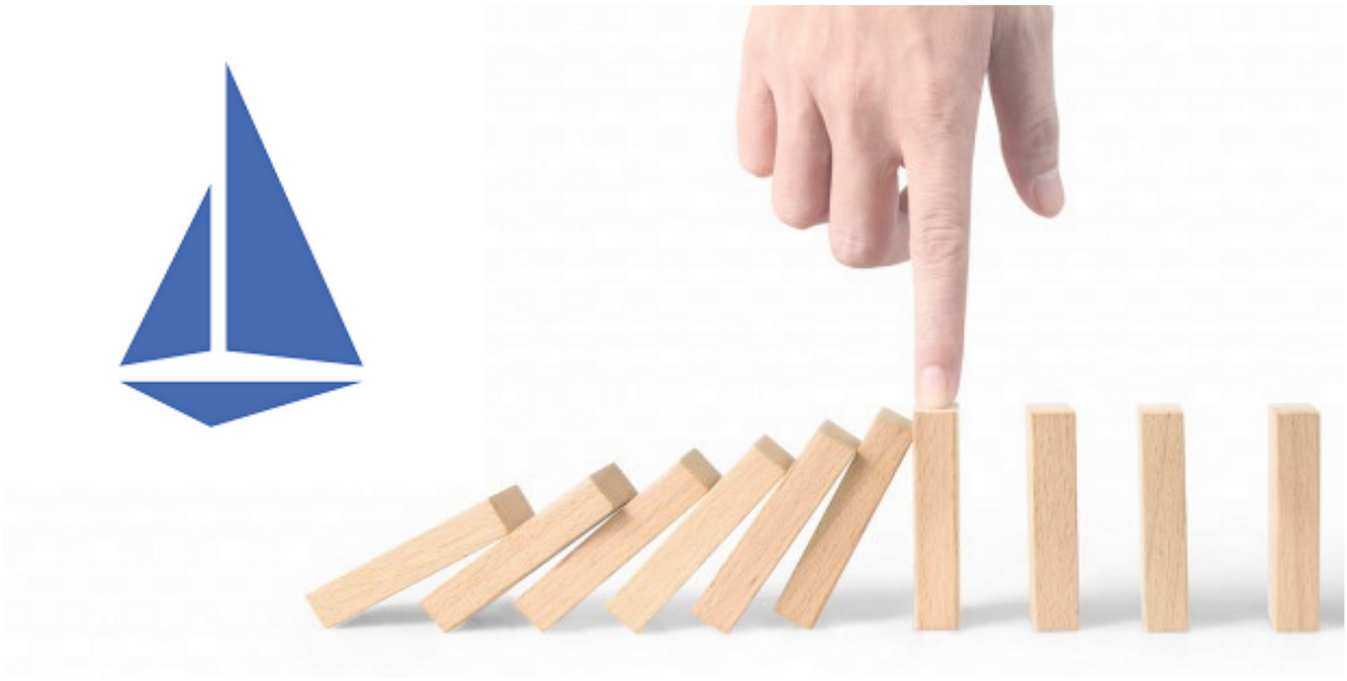
Jatin Juneja in OLX Engineering

SonarQube Integration in Android Application (Part -1)

Series Pit Stops

4 min read · Oct 19, 2019

434 5



Vikas Kumar in OLX Engineering

Demystifying Istio Circuit Breaking

At OLX Autos, we have started adopting Istio service mesh for our EKS cluster. There are a number of features we are excited to use —...

11 min read · Oct 12, 2020

201 2

Abhinav Rohatgi in OLX Engineering

JVM Profiling in Kubernetes with Java Flight Recorder

A walkthrough of the approach followed in OLX Autos for JVM Profiling in a Kubernetes (K8S) Environment using Java Flight Recorder (JFR)

9 min read · Apr 15, 2021


314

See all from Oleh Kudinov

See all from OLX Engineering

Recommended from Medium

Zeba

MVVM in iOS Swift

Model-View-ViewModel (MVVM) is a design pattern widely used in iOS app development to create clean, maintainable, and testable code. MVVM...

4 min read · Sep 6, 2023

15 1

Wahyu Alfandi

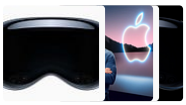
A Beginner's Guide to Clean Architecture in SwiftUI: Building Better Apps Step by Step

Photo by Lala Azizli on Unsplash

9 min read · Aug 24, 2023

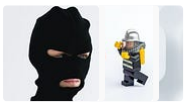
370 7

Lists



Apple's Vision Pro

7 stories · 51 saves

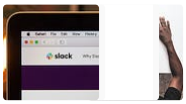


Tech & Tools

16 stories · 139 saves

Icon Design

36 stories · 216 saves



Productivity

237 stories · 317 saves

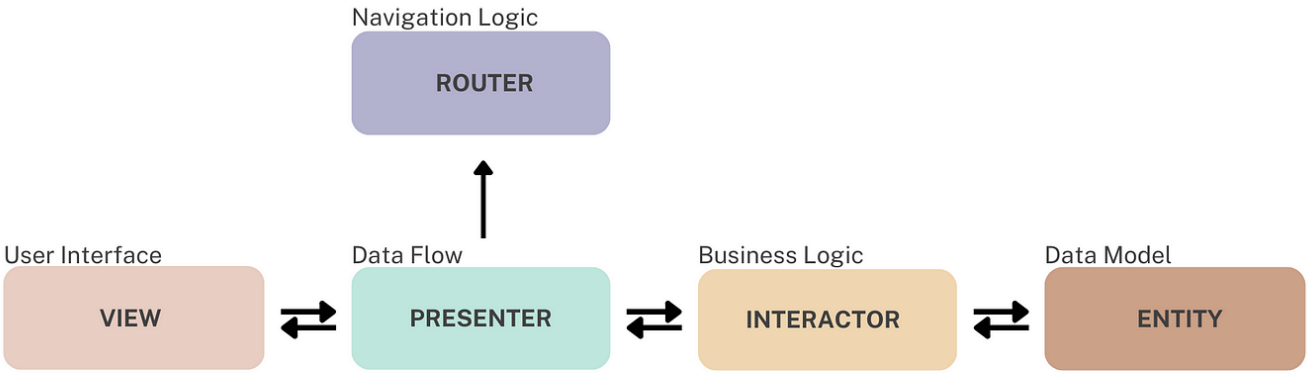
Roli Bernanda

Clean Architecture: Use-Case Centered Design in SwiftUI— MVVM

The idea is straightforward, Our design should not rely frameworks. Frameworks are tools to be used, they should not dictate how we build...

5 min read · Aug 25, 2023

187 2



Aslihan Gurkan in Dev Genius

VIPER Design Pattern in Swift

When it comes to building robust and maintainable iOS applications, having a clear and organized architecture is crucial. This is where...

4 min read · Aug 30, 2023

17

Alba Torres

Development with CI/CD and GitHub Actions on iOS project

I have started learning iOS development using SwiftUI and as part of my learning, I decided to set up a workflow to run Xcode tests using...

3 min read · Sep 27, 2023

106

Abdul Karim

How to be top 1% of iOS Developers in 2024

REMEMBER

3 min read · Jan 5

256 5