Best architecture for SwiftUI + Combine

Kyrylo Triskalo

The choice of architecture for the Combine + SwiftUI combination has always raised many questions for me, since there were no decent examples of how to solve such problem as:

•Dependancy Injection

After all, when we talk about MVVM, we almost always mean that we will use the Coordinator, since it helps to solve the problems listed above.

If the problem of routing in SwiftUI can be missed, since the capabilities of SwiftUI in this aspect are much ahead of UIKit, then the problem of dependency injection remains. Actually, I want to consider it in my presentation using examples of 3 architectures.

```swift
protocol AssociatedObject: class {
    func associatedObject<T>(for key: UnsafeRawPointer) -> T?
    func setAssociatedObject<T>(
        _ object: T,
        for key: UnsafeRawPointer,
        policy: AssociationPolicy
    )
}
extension AssociatedObject {
    func associatedObject<T>(for key: UnsafeRawPointer) -> T? {
        return objc_getAssociatedObject(self, key) as? T
    }

    func setAssociatedObject<T>(
        _ object: T,
        for key: UnsafeRawPointer,
        policy: AssociationPolicy = .strong
    ) {
        return objc_setAssociatedObject(
            self,
            key,
            object,
            policy.objcPolicy
        )
    }
}
enum AssociationPolicy {
    case strong
    case copy
    case weak

    var objcPolicy: objc_AssociationPolicy {
        switch self {
            case .strong:
                return .OBJC_ASSOCIATION_RETAIN_NONATOMIC
            case .copy:
                return .OBJC_ASSOCIATION_COPY_NONATOMIC
            case .weak:
                return .OBJC_ASSOCIATION_ASSIGN
        }
    }
}
```

This example shows a Swift wrapper for the Obj-C library

First architecture - MVP + C

For this architecture, we will need to use the Obj-C Runtime library to get and set the associated objects for our coordinators.

Instead of subclassing, within this architecture, polymorphism is achieved through composition, the capabilities of which are extended using mixins.

A few words about what mixin is

Mixin - is an approach that involves the use of composition instead of subclassing in order to avoid potential problems that may arise as a result of multiple inheritance (within the framework of swift, the problem of excessive functionality of parent classes is implied, which can be acquired due to the succession of class inheritance).

Benefits of using Mixins:

Code reuse. Since mixin within Swift is implemented through stored properties in protocols, this allows you to define default functional behavior for completely different classes without overloading them with unnecessary functionality of parent classes.

How are the coordinators arranged

```swift
import SwiftUI

protocol BaseCoordinator: AssociatedObject {
    func stop()
}

extension BaseCoordinator {
    fileprivate(set) var identifier: UUID {
        get {
            guard let identifier: UUID = associatedObject(for: &identifierKey) else {
                self.identifier = UUID()
                return self.identifier
            }
            return identifier
        }
        set { setAssociatedObject(newValue, for: &identifierKey) }
    }

    fileprivate(set) var children: [UUID: BaseCoordinator] {
        get {
            guard let children: [UUID: BaseCoordinator] = associatedObject(for: &childrenKey) else {
                self.children = [UUID: BaseCoordinator]()
                return self.children
            }
            return children
        }
        set { setAssociatedObject(newValue, for: &childrenKey) }
    }

    fileprivate func store<T: Coordinator>(child coordinator: T) {
        children[coordinator.identifier] = coordinator
    }

    fileprivate func free<T: Coordinator>(child coordinator: T) {
        children.removeValue(forKey: coordinator.identifier)
    }
}
```

So, the BaseCoordinator protocol contains 2 fields: **identifier** and **children**.

Thus, each coordinator who subscribes to the protocol will receive:

- the **identifier** field, which stores its identifier

- a field, **children**, which stores an array of its child coordinators.

And also get 2 methods that will set and release child coordinators

```swift
protocol Coordinator: BaseCoordinator {
    associatedtype U: View
    associatedtype P: Coordinator
    func start() -> U
    func shouldStop()
}

extension Coordinator {
    private(set) weak var parent: P? {
        get { associatedObject(for: &childrenKey) }
        set { setAssociatedObject(newValue, for: &childrenKey, policy: .weak) }
    }

    func coordinate<T: Coordinator>(to coordinator: T) -> some View {
        store(child: coordinator)
        coordinator.parent = self as? T.P
        return coordinator.start()
    }

    func stop() {
        children.removeAll()
        parent?.free(child: self)
    }

    func shouldStop() {
        stop()
    }
}

private var identifierKey: UInt8 = 0
private var childrenKey: UInt8 = 0
private var parentKey: UInt8 = 0
```

The Coordinator protocol, in turn, contains:

the **parent** field with a weak reference to the parent coordinator

the coordinate method that creates a link between the parent and child coordinator by calling the **store** method, which sets the child coordinator to the **associated object** for the parent coordinator. Also, the method sets the **parent** field and calls the **start** method of the child coordinator.

Thus, to implement a bundle of modules, 2 steps are required:
create a new module
- call the coordinate method of the parent coordinator and pass the child coordinator to it

```swift
final class AppCoordinator: Coordinator {
    typealias P = AppCoordinator

    weak var window: UIWindow?

    private let services: Services = Services()

    init(window: UIWindow) {
        self.window = window
    }

    deinit {
        print("⚠️ deinit AppCoordinator \(identifier)")
    }

    @discardableResult
    func start() -> some View {
        let coordinator = LoginViewCoordinator<AppCoordinator>(window: window, services: services)
        return coordinate(to: coordinator)
    }
}
```

As we remember, the coordinate method, in addition to creating a link, will also call the start () method of the child coordinator.

```swift
    func coordinate<T: Coordinator>(to coordinator: T) -> some View {
        store(child: coordinator)
        coordinator.parent = self as? T.P
        return coordinator.start()
    }
```

Which initialises itself and using NavigationLinkWrapper will call the method of transition to the view, which it received as an argument

```swift
    @discardableResult
    func start() -> some View {
        let view = DetailFactory.make(with: viewModel, coordinator: self)
        return NavigationLinkWrapper(destination: view, isPresented: isPresented)
    }
}
```

How are the presenters arranged

The architectural task of the presenter is that he is responsible for removing the coordinator from memory. That is, after the view and presenter are freed from memory, the **shouldstop** method is called, which, by the identifier, finds the child coordinator and removes it.

```
4
5  import Foundation
6
7  class Presenter<C: Coordinator> {
8      private(set) weak var coordinator: C?
9
10     init(coordinator: C) {
11         self.coordinator = coordinator
12     }
13
14     deinit {
15         coordinator?.shouldStop()
16         print("\(coordinator?.identifier.description ?? "nil") deinit \(Self.self)")
17     }
18 }
```

Presenter looks like this

Through the generic, we specify the coordinator that will control the module and thus set up all the connections.

```
14  final class LoginPresenter<C: LoginCoordinator>: Presenter<C>, LoginPresenting {
15
16      private let services: Services
17      @Published var viewModel: LoginViewModel
18
19      init(viewModel: LoginViewModel, coordinator: C, services: Services) {
20          self.viewModel = viewModel
21          self.services = services
22          super.init(coordinator: coordinator)
23          setupPublishers()
24      }
25
```

How are the views arranged

```swift
import SwiftUI

struct LoginView<T: LoginPresenting>: View {

    @ObservedObject private var presenter: T

    init(presenter: T) {
        self.presenter = presenter
    }

    var body: some View {
        ZStack {
            VStack {
                UserImage()
                LogInText()
                UsernameTextField(presenter: presenter)
                PasswordSecureField(presenter: presenter)
                LogInButton(presenter: presenter)
                    .hidden(!presenter.viewModel.isValidForm)
            }
            .padding()
        }.padding(.bottom, 50)
    }
}
```

There is no specific underlying implementation for the view layer.

We simply indicate through the generic protocol of our presenter

```swift
protocol LoginPresenting: ObservableObject {
    associatedtype U1: View
    var viewModel: LoginViewModel { get set }
    func openApp(isPresented: Binding<Bool>) -> U1
}
```

What a module factory looks like

```
@discardableResult
func start() -> some View {
    let coordinator = LoginViewCoordinator<AppCoordinator>(window: window, services: services)
    return coordinate(to: coordinator)
}
```

```
enum LoginFactory {
    static func make<T: LoginCoordinator>(with viewModel: LoginViewModel, coordinator: T, services: Services) -> some View {
        let presenter = LoginPresenter(viewModel: viewModel, coordinator: coordinator, services: services)
        let view = LoginView(presenter: presenter)
        return view
    }
}
```

```
@discardableResult
func start() -> some View {
    let view = LoginFactory.make(with: LoginViewModel(), coordinator: self, services: services)
    let navigation = NavigationView { view }
    let hosting = UIHostingController(rootView: navigation)
    window?.rootViewController = hosting
    window?.makeKeyAndVisible()
    return EmptyView() // we have to return something
}
```

Peculiarities of routing between modules

I must say right away that in SwiftUI applications, the UI is the state from the application.

```
struct PaymentView: View {
    @Binding var isOpen: Bool
    @State var isBiometricsOpen = false

    @ObservedObject var store: PaymentStore

    var body: some View {
        switch store.stage {
        case .scanCode:
            QRCodeView(isOpen: $isOpen, onScan: store.fetchPayment)

        case .fetchingDetails:
            loadingView

        case .failedToFetch:
            Color.clear.alert(isPresented: $isOpen) {
                Alert(title: Text(ModeStrings.Payment.FailedToFetch.title),
                      message: Text(ModeStrings.Payment.FailedToFetch.description),
                      dismissButton: .default(Text(ModeStrings.Buttons.ok)))
            }

        case .confirm(let payment):
            NavigationView {
                recap(for: payment)
            }

        case .authorizing:
            loadingView

        case .biometricAuthorizing:
            Color.clear.sheet(isPresented: $isBiometricsOpen) {
                BiometricsAuthorizationView(authenticationContext: .transaction) { isAuthorized in
                    if isAuthorized {
                        store.didAuthorizeBiometric()
                    } else {
                        isOpen = false
                    }
                }
            }
            .onChange(of: isBiometricsOpen) { value in
                if case .biometricAuthorizing = store.stage, !isBiometricsOpen {
                    isOpen = false
                }
            }
```

In practice, this means that the View will have exactly the same appearance as in the example: depending on the change of the state, the View will be redrawn.

Thus, all transitions to other modules will also be inside the View itself. Like in .case confirm.

That is, even if you have completed the network request and want to make a transition to another screen, then you cannot do it directly from the presenter in any way. You will need to change the state of the View, and inside it, call NavigationLink.

In our architecture, there will be no direct transitions using NavigationLink directly from the View, but they will be implemented by means of successive calls from the View to the Coordinator.

In order not to call NavigationLink directly, we use a special View NavigationButton, which, depending on the isPresented state (it changes after clicking), will call completion)

```swift
extension View {
    func withNavigation<T: View>(to destination: T) -> some View {
        background(destination)
    }
}

struct NavigationButton<CV: View, NV: View>: View {
    @State private var isPresented = false

    var contentView: CV
    var navigationView: (Binding<Bool>) -> NV

    var body: some View {
        Button(action: {
            self.isPresented = true
        }) {
            contentView
                .withNavigation(to:
                    navigationView($isPresented)
                )
        }
    }
}
```

```swift
struct DetailView<T: DetailPresenting>: View {
    @ObservedObject var presenter: T

    private var viewModel: DetailViewModel? {
        return presenter.viewModel
    }

    var body: some View {
        Group {
            if viewModel != nil {
                NavigationButton(contentView: Text("\(viewModel!.date, formatter: dateFormatter)"),
                                 navigationView: { isPresented in
                                    self.presenter.buttonPressed(isActive: isPresented)
                })
                .foregroundColor(Color.blue)
            } else {
                Text("Please select a date")
            }
        }.navigationBarTitle(Text("Detail"))
    }
}
```

***** In the case when you just made a network request and you need to make a transition, then you use the same NavigationButton, but only with the marked field isPresented = true**

```
NavigationButton(contentView: Text("\(viewModel!.date, formatter: dateFormatter)"),
                 navigationView: { isPresented in
                    self.presenter.buttonPressed(isActive: isPresented)
})
```

```
 func buttonPressed(isActive: Binding<Bool>) -> some View {
     return coordinator?.presentNextView(isPresented: isActive)
 }
```

```
extension DetailCoordinator {
    func presentNextView(isPresented: Binding<Bool>) -> some View {
        let coordinator = NavigationMasterCoordinator<Self>(viewModel: nil, isPresented: isPresented)
        return coordinate(to: coordinator)
    }
}
```

```
@discardableResult
func start() -> some View {
    let view = DetailFactory.make(with: viewModel, coordinator: self)
    return NavigationLinkWrapper(destination: view, isPresented: isPresented)
}
```

```
struct NavigationLinkWrapper<T: View>: View {
    typealias DestinationView = T

    var destination: T
    @Binding var isPresented: Bool
    var isDetailLink: Bool = true

    var body: some View {
        NavigationLink(destination: destination, isActive: $isPresented) {
            EmptyView()
        }.isDetailLink(isDetailLink)
    }
}
```

And then the next series of calls follows:

By clicking on the NavigationButton in the View, the button changes its state and calls its completion, in which it passes the changed argument isPresented

In the Presenter, the isActive argument is passed to the coordinator method

In the coordinator method, the isPresented argument is written to the child coordinator.

In the child coordinator, in the start () method (it starts after the call to coordinate in the parent coordinator), the initialized View, as well as the View and the isPresented argument are passed to the View wrapper - NavigationLinkWrapper, which contains the same NavigationLink.

Therefore, the argument isPresented = true, so the NavigationLink argument changes and the navigation occurs.

```swift
extension DetailCoordinator {
    func presentNextView(isPresented: Binding<Bool>) -> some View {
        let coordinator = NavigationMasterCoordinator<Self>(viewModel: nil, isPresented: isPresented)
        return coordinate(to: coordinator)
    }

    func presentPreviousView(isPresented: Binding<Bool>) -> some View {
        return openPreviousCoordinator(to: self, isPresented: isPresented)
    }
}
```

```swift
extension Coordinator {
    private(set) weak var parent: P? {
        get { associatedObject(for: &childrenKey) }
        set { setAssociatedObject(newValue, for: &childrenKey, policy: .weak) }
    }

    func coordinate<T: Coordinator>(to coordinator: T) -> some View {
        store(child: coordinator)
        coordinator.parent = self as? T.P
        return coordinator.start()
    }

    func openPreviousCoordinator<T: Coordinator>(to coordinator: T, isPresented: Binding<Bool>) -> some View {
        let parent = self as? T.P
        return parent?.start(isPresented: isPresented)
    }
```

```swift
protocol Coordinator: BaseCoordinator {
    associatedtype U: View
    associatedtype U1: View

    associatedtype P: Coordinator
    func start() -> U
    func shouldStop()
    func start(isPresented: Binding<Bool>) -> U1

}
```

However, going back one screen will be a big problem.

Over the course of hours of trials and attempts, I did not manage to figure out how to do this. What can we say about deep-links ... The author himself stucked on these problems. Since if they are achievable, then they require excessive complexity.

Total:

Despite the goal of the implementation of DI was achieved, nevertheless it brought huge problems with routing.

In addition, we also have redundant complexity in the underlying architecture components.

Even if we solve the problem with deeplinks, the architecture still remains difficult to understand, modifications / extensions + new approaches introduced by SwiftUI just limiting its architecture

Second architecture - MVVM + C

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

@available(iOS 13.0, *)
func makeMoviesQueryListViewModelWrapper(didSelect: @escaping MoviesQueryListViewModelDidSelectAction) ->
    MoviesQueryListViewModelWrapper {
    return MoviesQueryListViewModelWrapper(viewModel: makeMoviesQueryListViewModel(didSelect: didSelect))
}
```

There is an alternative - to use the usual MVVM + C architecture, and inside it, wrap SwiftUI Views in UIHostingViewControllers and operate with ordinary ViewController armies in all flows. This way we can manage SwiftUI modules in the UIKit NavigationStack. This eliminates the complexity of using the some View protocol, which requires associatedType-s for its protocols + is not supported by the UINavigationController. I think this architecture will be justified in cases where it will be necessary to quickly integrate a SwiftUI module into a UIKit application.

Total:

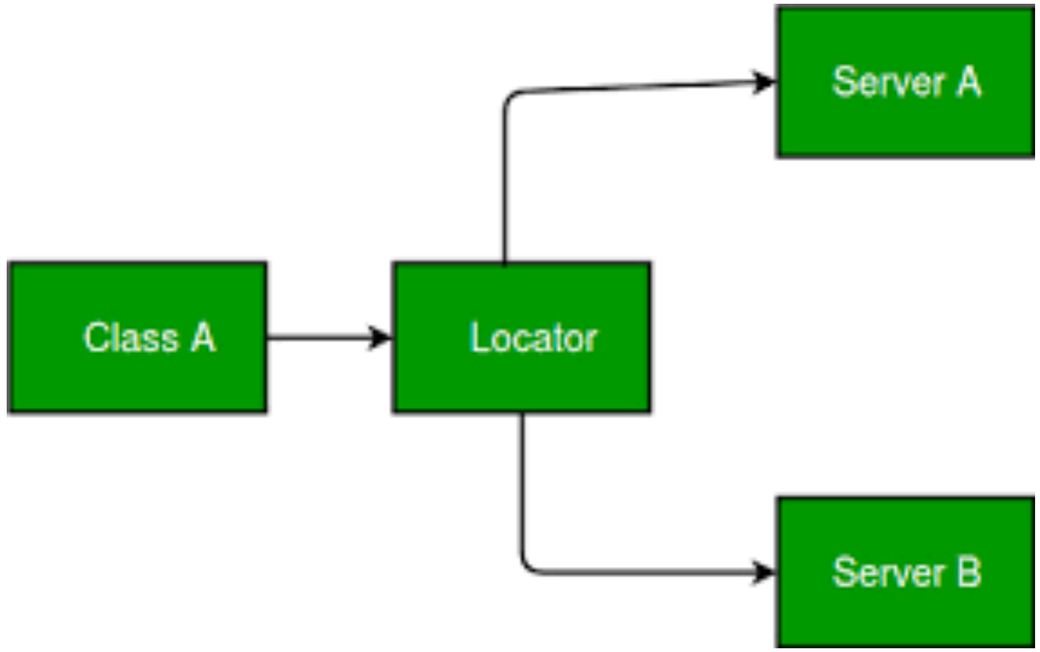
The architecture can be used during the transition period of the application, when certain application modules can be translated to SwitfUI. This approach allows you to solve all the problems associated with routing and di, however, it still remains a hybrid approach to a certain extent limiting the capabilities of modern frameworks

Third architecture - MVVM and pattern ???

Service Locator

Service Locator is a pattern, the main idea of which is that instead of directly instantiating dependencies, we should use a special locator object responsible for finding each dependency (that is, a service, hence the name of the template). The locator allows you to register dependencies and manage their life cycles.

Service Locator has two common implementations:
A globally accessible locator, usually as a singleton or a type with static methods.
Locator embedded as a dependency



The implementation of this pattern is achieved using a powerful tool introduced by Swift 5.1 - propertyWrapper.

Let's look at the implementation using the Resolver framework as an example, which we will use for our 3rd architecture.

```
extension Resolver: ResolverRegistering {
  public static func registerAllServices() {
    defaultScope = .graph
    register { HTTPClient(session: URLSession(configuration: .default)) }
    register { UserService() }
    register { ChatService() }
  }
}
```

```
class ChatListPresenter: ChatListPresenterProtocol {

    @Injected private var chatService: ChatService
    @Published var chatList: ChatList = ChatList(chats: [])
    private var cancellableSet: Set<AnyCancellable> = []
```

How does it all work?

1) first, we register all our services

2) we declare our services in our ViewModels

3) we are happy, because it is extremely simple and convenient

```swift
extension Resolver: ResolverRegistering {
  public static func registerAllServices() {
    defaultScope = .graph
    register { HTTPClient(session: URLSession(configuration: .default)) }
    register { UserService() }
    register { ChatService() }
  }
}
```

```swift
@discardableResult
public final func register<Service>(_ type: Service.Type = Service.self, name: Resolver.Name? = nil,
                                    factory: @escaping ResolverFactory<Service>) -> ResolverOptions<Service> {
  lock.lock()
  defer { lock.unlock() }
  let key = ObjectIdentifier(Service.self).hashValue
  let registration = ResolverRegistrationOnly(resolver: self, key: key, name: name, factory: factory)
  add(registration: registration, with: key, name: name)
  return registration
}
```

```swift
// ResolverRegistration stores a service definition and its factory closure.
public final class ResolverRegistrationOnly<Service>: ResolverRegistration<Service> {

  public var factory: ResolverFactory<Service>

  public init(resolver: Resolver, key: Int, name: Resolver.Name?, factory: @escaping ResolverFactory<Service>) {
      self.factory = factory
      super.init(resolver: resolver, key: key, name: name)
  }

  public final override func resolve(resolver: Resolver, args: Any?) -> Service? {
      guard let service = factory() else {
          return nil
      }
      mutate(service, resolver: resolver, args: args)
      return service
  }
```

```swift
/// Internal function adds a new registration to the proper container.
private final func add<Service>(registration: ResolverRegistration<Service>, with key: Int, name: Resolver.Name?) {
    if var container = registrations[key] {
        container[name?.rawValue ?? NONAME] = registration
        registrations[key] = container
    } else {
        registrations[key] = [name?.rawValue ?? NONAME : registration]
    }
}

private let NONAME = "*"
private let lock = Resolver.lock
private var childContainers: [Resolver] = []
private var registrations = [Int : [String : Any]]()
```

1) Services are registered both by the hash code and name and

2) Initialised using the ResolverRegistrationOnly class

3) Added to the registrations array

```swift
@propertyWrapper public struct Injected<Service> {
    private var service: Service
    public init() {
        self.service = Resolver.resolve(Service.self)
    }
    public init(name: Resolver.Name? = nil, container: Resolver? = nil) {
        self.service = container?.resolve(Service.self, name: name) ?? Resolver.resolve(Service.self, name: name)
    }
    public var wrappedValue: Service {
        get { return service }
        mutating set { service = newValue }
    }
    public var projectedValue: Injected<Service> {
        get { return self }
        mutating set { self = newValue }
    }
}
```

```swift
public static func resolve<Service>(_ type: Service.Type = Service.self, name: Resolver.Name? = nil, args: Any? = nil) -> Service {
    lock.lock()
    defer { lock.unlock() }
    registrationCheck()
    if let registration = root.lookup(type, name: name),
        let service = registration.scope.resolve(resolver: root, registration: registration, args: args) {
        return service
    }
    fatalError("RESOLVER: '\(Service.self):\(name?.rawValue ?? "NONAME")' not resolved. To disambiguate optionals use
        resolver.optional().")
}
```

```swift
private final func lookup<Service>(_ type: Service.Type, name: Resolver.Name?) -> ResolverRegistration<Service>? {
    let key = ObjectIdentifier(Service.self).hashValue
    let containerName = name?.rawValue ?? NONAME
    if let container = registrations[key], let registration = container[containerName] {
        return registration as? ResolverRegistration<Service>
    }
    for child in childContainers {
        if let registration = child.lookup(type, name: name) {
            return registration
        }
    }
    return nil
}
```

1) In the propertyWrapper itself, during initialization, the resolve method is called

2) The resolve method runs the lookUp method, which finds the required service by the cache code and name and returns it.

3) Accordingly, if the service was found, then the resolve method is called for the corresponding scoupe

```swift
public class ResolverScope: ResolverScopeType {

    // Moved definitions to ResolverScope to allow for dot notation access

    /// All application scoped services exist for lifetime of the app. (e.g Singletons)
    public static let application = ResolverScopeCache()
    /// Cached services exist for lifetime of the app or until their cache is reset.
    public static let cached = ResolverScopeCache()
    /// Graph services are initialized once and only once during a given resolution cycle. This is the default scope.
    public static let graph = ResolverScopeGraph()
    /// Shared services persist while strong references to them exist. They're then deallocated until the next resolve.
    public static let shared = ResolverScopeShare()
    /// Unique services are created and initialized each and every time they're resolved.
    public static let unique = ResolverScopeUnique()

    // abstract base for class never called
    public func resolve<Service>(resolver: Resolver, registration: ResolverRegistration<Service>, args: Any?) -> Service? {
        fatalError("abstract")
    }
}
```

```swift
/// Graph services are initialized once and only once during a given resolution cycle. This is the default scope.
public final class ResolverScopeGraph: ResolverScope {

    public override init() {}

    public final override func resolve<Service>(resolver: Resolver, registration: ResolverRegistration<Service>, args: Any?) -> Service? {
        if let service = graph[registration.cacheKey] as? Service {
            return service
        }
        resolutionDepth = resolutionDepth + 1
        let service = registration.resolve(resolver: resolver, args: args)
        resolutionDepth = resolutionDepth - 1
        if resolutionDepth == 0 {
            graph.removeAll()
        } else if let service = service, type(of: service as Any) is AnyClass {
            graph[registration.cacheKey] = service
        }
        return service
    }

    private var graph = [String : Any?](minimumCapacity: 32)
    private var resolutionDepth: Int = 0
}
```

4) In our case, scoupe is graph, since we specified it at the very beginning when registering all services

5) As a result, the function, if there is a registered service, returns it to us and sets the field to @Injected property

Is it possible to do without a framework?

1) The Resolver framework is very lightweight. It only contains 1 file for 800 lines. Which you can add to your project without using Swift Package Manager or CocoaPods.

2) You can always write your own implementation. The amount of Resolver code is equivalent to all the necessary base classes, protocols and launch instructors that will be needed to describe the coordinator for the application.

Summary of the pattern:

I believe that modern problems require modern solutions. Therefore, if old patterns become unnecessarily difficult to apply, then it is time to look for new solutions. It may still change, but for now I consider the Service Locator pattern to be a new Coordinator that will gain widespread acceptance due to its lightness, flexibility, and ease of use.

How does the new architecture work?

Sign in to the application + set up dependencies

Yes, it is so simple that there is really nothing to show ...

Logging into the application takes several lines because it does not require:
- no Launch Instructor
- no implementation of Dependancy Injection into the Application Coordinator and all other coordinators involved in the deployment of the initial application modules (usually there are 2 or 3) - neither the module factories for the coordinators, nor for the modules themselves

```
final class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?

    func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options connectionOptions: UIScene.ConnectionOptions) {

        let contentView = LoginView(presenter: LoginPresenter())

        if let windowScene = scene as? UIWindowScene {
            let window = UIWindow(windowScene: windowScene)
            window.rootViewController = UIHostingController(rootView: contentView)
            self.window = window
            window.makeKeyAndVisible()
        }
    }
}
```

```
extension Resolver: ResolverRegistering {
  public static func registerAllServices() {
    defaultScope = .graph
    register { HTTPClient(session: URLSession(configuration: .default)) }
    register { UserService() }
    register { ChatService() }
  }
}
```

Routing

If we compare it with the classic use of routing in the coordinator (used in the 2nd solution architecture), then the difference is enormous both in ease of use and in the amount of required code.

```
public func getMenuVC () -> MenuViewController {

    let routes = MenuVCRoutes(onMain: {
        self.showMainPharmacyVC()
    }, onFavourites: {
        self.showWishList()
    }, onClinics: {
        self.showClinicsList()
    }, onShares: {
        self.showSuperDeals()
    }, onService: {

    }, onArticles: {
        self.showArticles()
    }, onForum: {
        self.showForum()
    })

    let vc = self.factory.instantiateMenuViewController(routes: routes, services: services)
    return vc
}
```

```
extension DependencyProvider: MenuViewControllerFactory {

   func instantiateMenuViewController(routes: MenuVCRoutes, services: Services) -> MenuViewController {
       let vc = MenuViewController.instantiate(from: .Menu)
       vc.presenter = MenuPresenter(routes: routes, userService: services.userService)
       return vc
   }
```

```
func onMain() {
    routes.onMain?()
}

func onFavourites() {
    routes.onFavourites?()
}

func onClinics() {
    routes.onClinics?()
}

func onShares() {
    routes.onShares?()
}

func onService() {
    routes.onService?()
}

func onArticles() {
    routes.onArticles?()
}

func onForum() {
    routes.onForum?()
}
```

1)We need to create a structure where the properties will contain the functions

2) In the coordinator, initialize each transition

3) Using the module factory (optional) create our module

4) In the ViewModel, call the required .sink method for a specific Publisher

```
    var body: some View {
        NavigationView {
            ZStack {
                VStack {
                    HelloText()
                    UserImage()
                    UsernameTextField(username: $presenter.email)
                    PasswordSecureField(password: $presenter.password)
                    NavigationLink(destination: presenter.buildTabBarView()) { Text("Log In") }
                    .hidden(presenter.logInState != .enable)
                }
                .padding()
            }
        }
    }
}
```

```
extension LoginViewModel: LoginViewModelProtocol {

    func buildTabBarView() -> TabBarView {
        let profile = ProfileView(presenter: ProfilePresenter())
        let chatsView: ChatListView = ChatListView(presenter: ChatListViewModel())
        return TabBarView(profileView: profile, chatsView: chatsView)
    }

}
```

1) Easily access the NavigationLink and transfer what we need to it.

And don't worry about any dependencies, as they are accessible from anywhere in the application.

ViewModel

There will be less code in the ViewModels since you do not need to set each one.

And if suddenly at some point in time you need to add a new service to your module, you can do this with just 1 line, and not a queue of changes in the coordinator.

```
    // MARK: - Vars & Lets

    // Services
    weak var delegate: DoctorVCDelegate?
    private let userService: UserService
    private let consultationService: ConsultationService
    private let routes: DoctorRoutes
    private let loader: Loader = Loader()

    // Data Sources
    var mainDSPresenter: DoctorDSPresenter?
    var mainDS: DoctorDS?
    var reviewDSPresenter: DoctorReviewDSPresenter?
    var reviewDS: DoctorReviewDS?
    var responseDSPresenter: DoctorAnswersDSPresenter?
    var responseDS: DoctorAnswersDS?
    let user: UserProfile

    // MARK: - Init

    init(routes: DoctorRoutes, service: Services, mainDSPresenter: DoctorDSPresenter,
         reviewDSPresenter: DoctorReviewDSPresenter, responseDSPresenter: DoctorAnswersDSPresenter, user: UserProfile) {
        self.routes = routes
        self.userService = service.userService
        self.consultationService = service.consulatationService
        self.mainDSPresenter = mainDSPresenter
        self.mainDS = DoctorDS(presenter: mainDSPresenter)
        self.reviewDSPresenter = reviewDSPresenter
        self.reviewDS = DoctorReviewDS(presenter: reviewDSPresenter)
        self.responseDSPresenter = responseDSPresenter
        self.responseDS = DoctorAnswersDS(presenter: responseDSPresenter)
        self.user = user
        self.mainDS?.presenter.onFold = self.onFold()
    }
```

```
class ChatListViewModel: ChatListViewModelProtocol {

    @Injected private var chatService: ChatService
    @Published var chatList: ChatList = ChatList(chats: [])
    private var cancellableSet: Set<AnyCancellable> = []

    init() {}
}
```

	Comparative table of architectures	
	Hybrid architecture using Coordinator + Combine + SwiftUI	New architecture using ServiceLocator + Combine + SwiftUI
Entrance to the application	Requires a lot of effort to implement, since many components must be involved, such as Dependency Manager, Lauch Instructor and others	Colossally simple, since it requires practically nothing, except for the transfer of the most important View
Dependency injection	Sequential transfer of dependencies from coordinator to coordinator, from flow to flow is required.	Dependencies are available anywhere
Routing	The description of the logic occurs in 3 components of the module to implement the transition logic	Routing can be done directly from View
Initialising modules	It is necessary to carry out dependency injection + routing requires additional data structures, this increases the complexity of initialization of modules	Extremely simple, since dependencies are available anywhere, and there is no need to deal with the routing problem
Number of code	Big	Minimum

Conclusions

In my comparison table, I wanted to show that there is no alternative to the new approach, which consists in abandoning the Coordinator and completely switching to the Service Locator for architectures using the Combine + SwiftUI bundle.

Resources

MVP+C architecture

https://lascorbe.com/posts/2020-04-27-MVPCoordinators-SwiftUI-part1/ - 1 part
https://lascorbe.com/posts/2020-04-28-MVPCoordinators-SwiftUI-part2/ - 2 part
https://lascorbe.com/posts/2020-04-29-MVPCoordinators-SwiftUI-part3/ - 3 part
https://github.com/Lascorbe/SwiftUI-MVP-Coordinator - git

MVVM + C + Swift architecture (wrappers for SwiftUI modules)
https://tech.olx.com/clean-architecture-and-mvvm-on-ios-c9d167d9f5b3

Resolver:
https://www.raywenderlich.com/22203552-resolver-for-ios-dependency-injection-getting-started - tutorial

https://github.com/hmlongco/Resolver - git

Fake REST API:
ihttps://www.youtube.com/watch?v=7vx0RIwHVzg&t=928s
https://github.com/typicode/json-server

THANK YOU FOR YOUR ATTENTION