

MVC, MVVM, ReactorKit, VIPER를 거쳐 RIB 정착기

Architecture 이야기

안정민 한국카카오은행

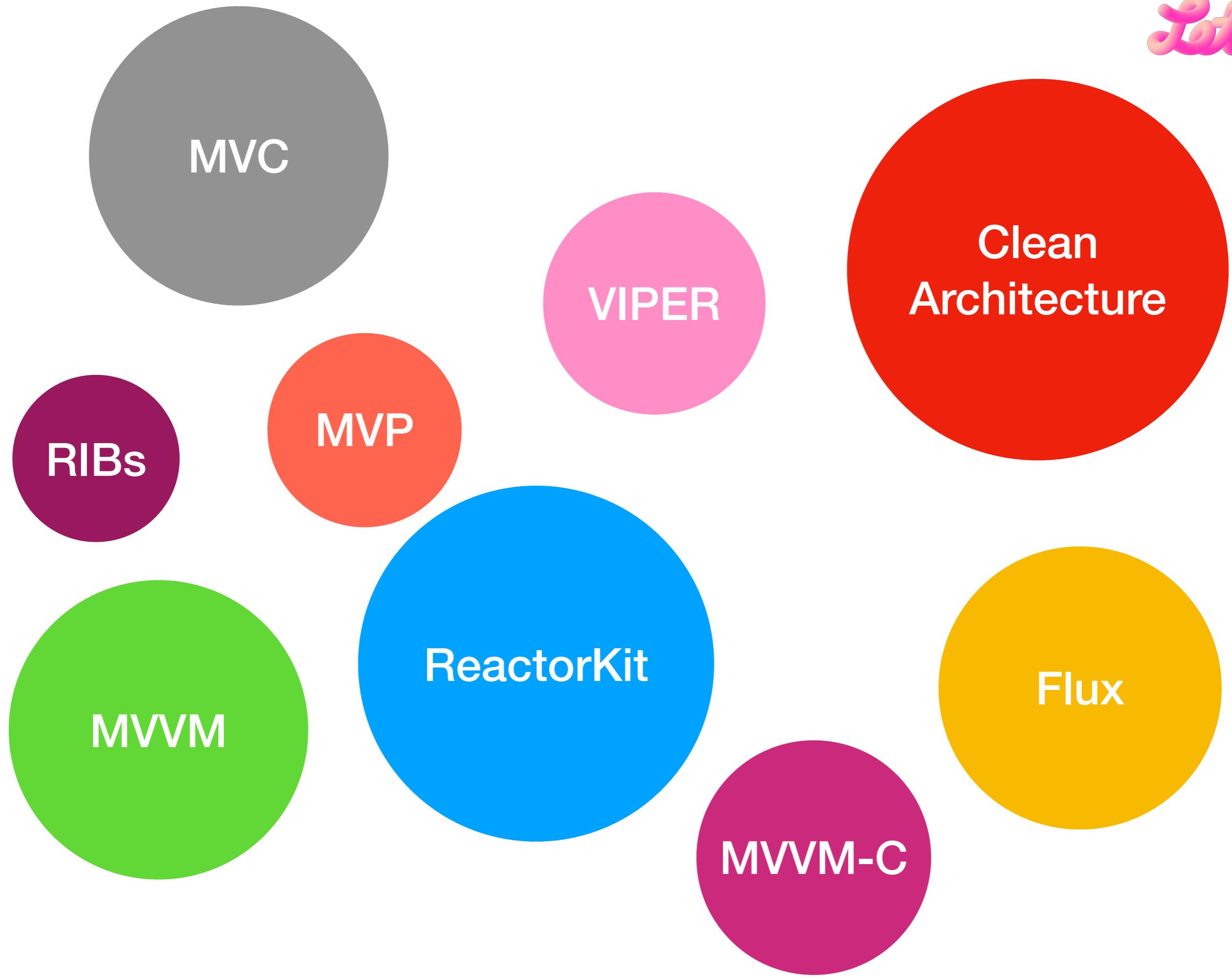
소개

- 現 한국카카오은행 iOS 개발자 (16.10 ~)
- 블로그 운영 : minsone.github.io

목차



기존 아키텍처에
왜 만족 못했는가?



재사용성

격리화

수정 용이함

코드 표준화

비즈니스
로직 단순화

테스트코드

명확한
코드 위치

기존 아키텍처에 왜 만족 못했는가?

- 기존 기능 수정 및 확장 어려움
- 화면 단위가 아닌 프로세스 단위로 유연한 개발 필요
- 더 확실한 안정화 필요

기존 아키텍처에 왜 만족 못했는가?

- 기존 기능 수정 및 확장 어려움
 - Swift 2버전부터 개발로 당시 Swift 이해도 낮음
 - 당시 아키텍처의 표준화된 틀이 없음.
 - 유연하지 않은 설계로 인한 기능 확장의 어려움.
- 16년 10월 : 3명 -> 19년 10월 16명

기존 아키텍처에 왜 만족 못했는가?

- 기존 기능 수정 및 확장 어려움
- 화면 단위가 아닌 프로세스 단위로 유연한 개발 필요
- 더 확실한 안정화 필요

기존 아키텍처에 왜 만족 못했는가?

- 화면 단위가 아닌 프로세스 단위로 유연한 개발 필요
 - Viewless 아키텍처가 필요.
 - 자체 아키텍처 제작
 - ReactorKit, Redux, Coordinator 등 아키텍처 참고
 - 한 곳에서 많은 로직을 처리하게 코드가 변질됨.
 - 재사용 어렵고, 새로운 비즈니스 로직을 넣기 어려움.
 - 자체 제작 아키텍처의 유지보수 어려움.

기존 아키텍처에 왜 만족 못했는가?

- 기존 기능 수정 및 확장 어려움
- 화면 단위가 아닌 프로세스 단위로 유연한 개발 필요
- 더 확실한 안정화 필요

기존 아키텍처에 왜 만족 못했는가?

- 더 확실한 안정화 필요
 - 앞의 이유로 인한 테스트 코드 작성 시늉도 어려움.
 - 바쁘다는 핑계로 테스트를 제대로 못했던 문제
 - 테스트 코드 템플릿 또는 가이드가 있는 아키텍처가 거의 없음.
 - 체계화된 테스트 코드 작성 방식이 필요

Let



RIBs 도입까지

아키텍처 여정

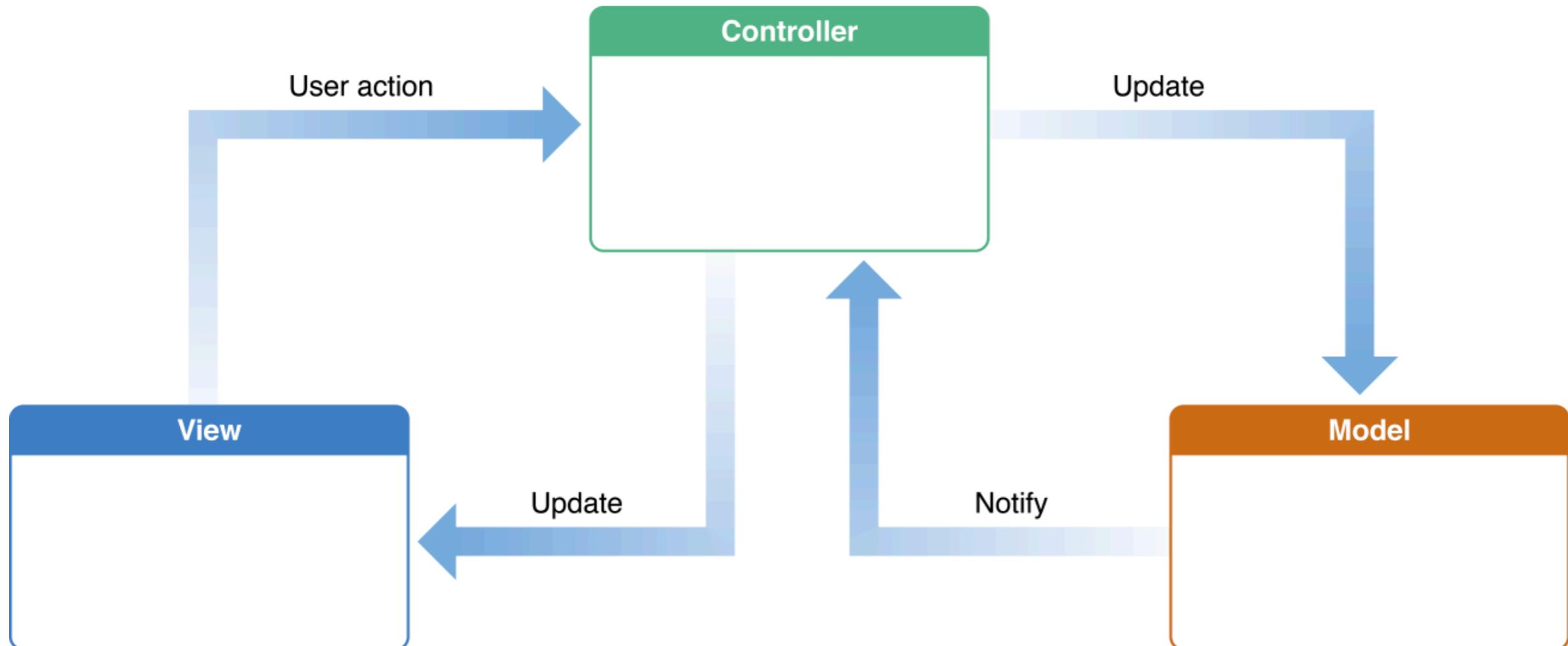
적용했던 아키텍처 예정

- MV(C)
- MVVM
- ReactorKit
- VIPER

적용했던 아키텍처 예정

- MV(C)
- MVVM
- ReactorKit
- VIPER

적용했던 아키텍처 예정 - MV(C)



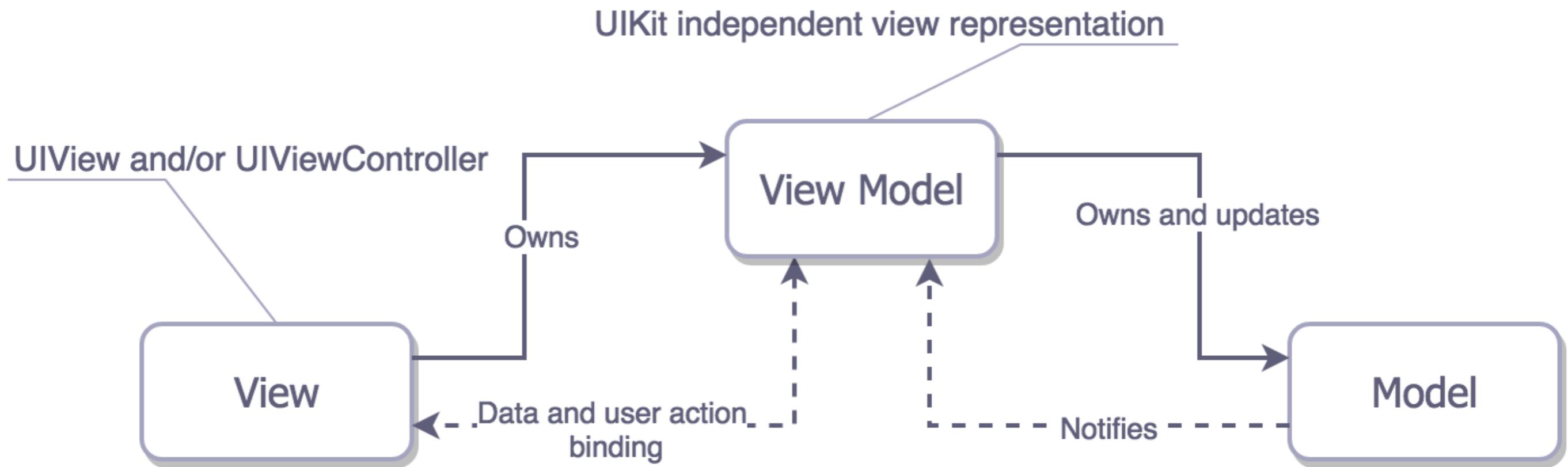
적용했던 아키텍처 예정 - MV(C)

- 장점
 - 기존에 익숙하던 구조
 - 비즈니스 로직이 거의 없는 단순한 화면에만 적용
- 단점
 - 화면의 기능이 확장되면 코드가 잘 정리되지 않고 깔끔하게 작성하기 쉽지 않음.

적용했던 아키텍처 예정

- MV(C)
- MVVM
- ReactorKit
- VIPER

적용했던 아키텍처 예정 - MVVM



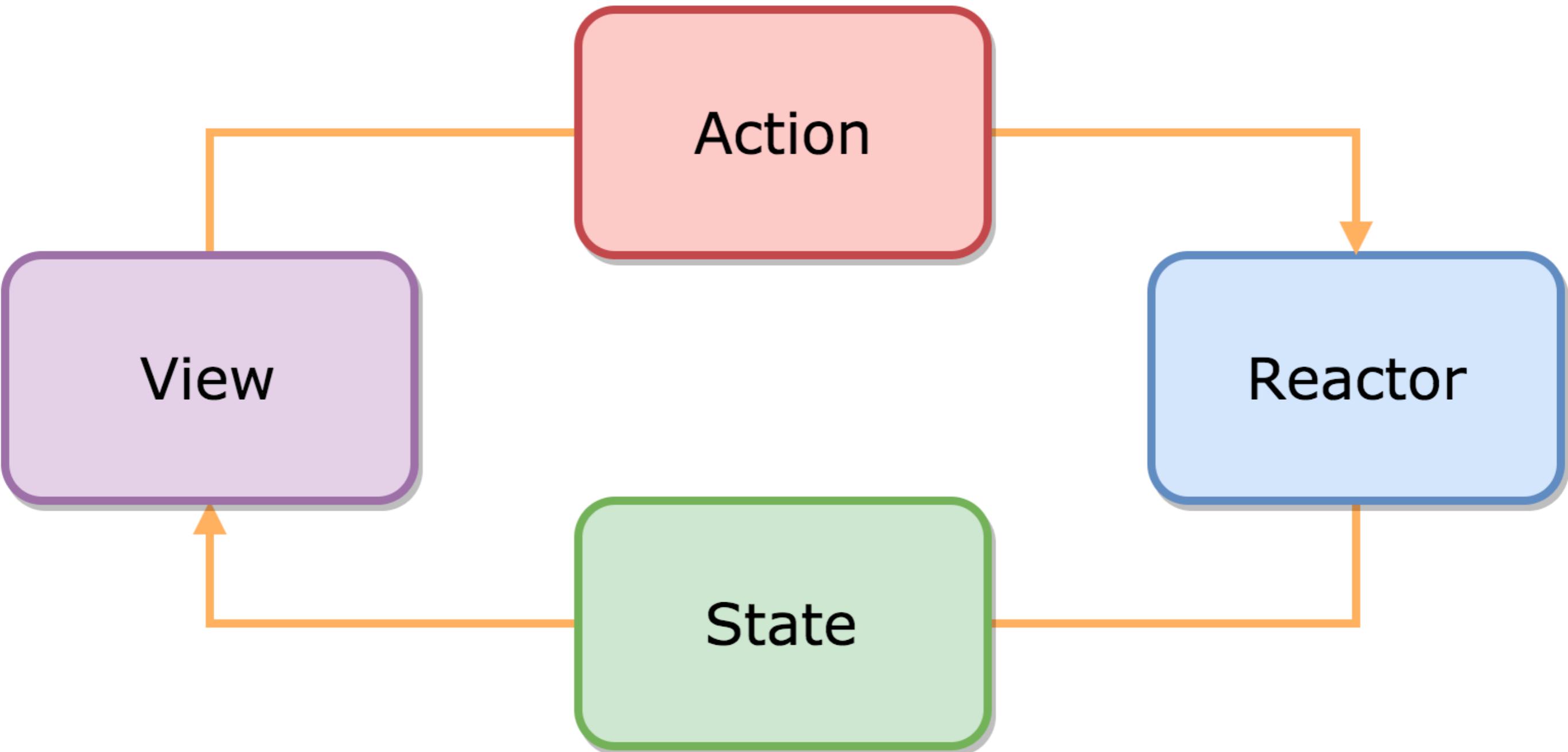
적용했던 아키텍처 예정 - MVVM

- 장점
 - MVC보다는 코드가 정리되는 느낌적인 느낌
- 단점
 - 복잡해질수록 ViewModel이 빠르게 비대해짐.
 - 표준화된 틀이 존재하지 않아 사람마다 이해가 다름.
 - 테스트 코드 작성이 가능하면서도 어려움.
 - Rx의 허들 및 디버깅

적용했던 아키텍처 예정

- MV(C)
- MVVM
- ReactorKit
- VIPER

적용했던 아키텍처 예정 - ReactorKit



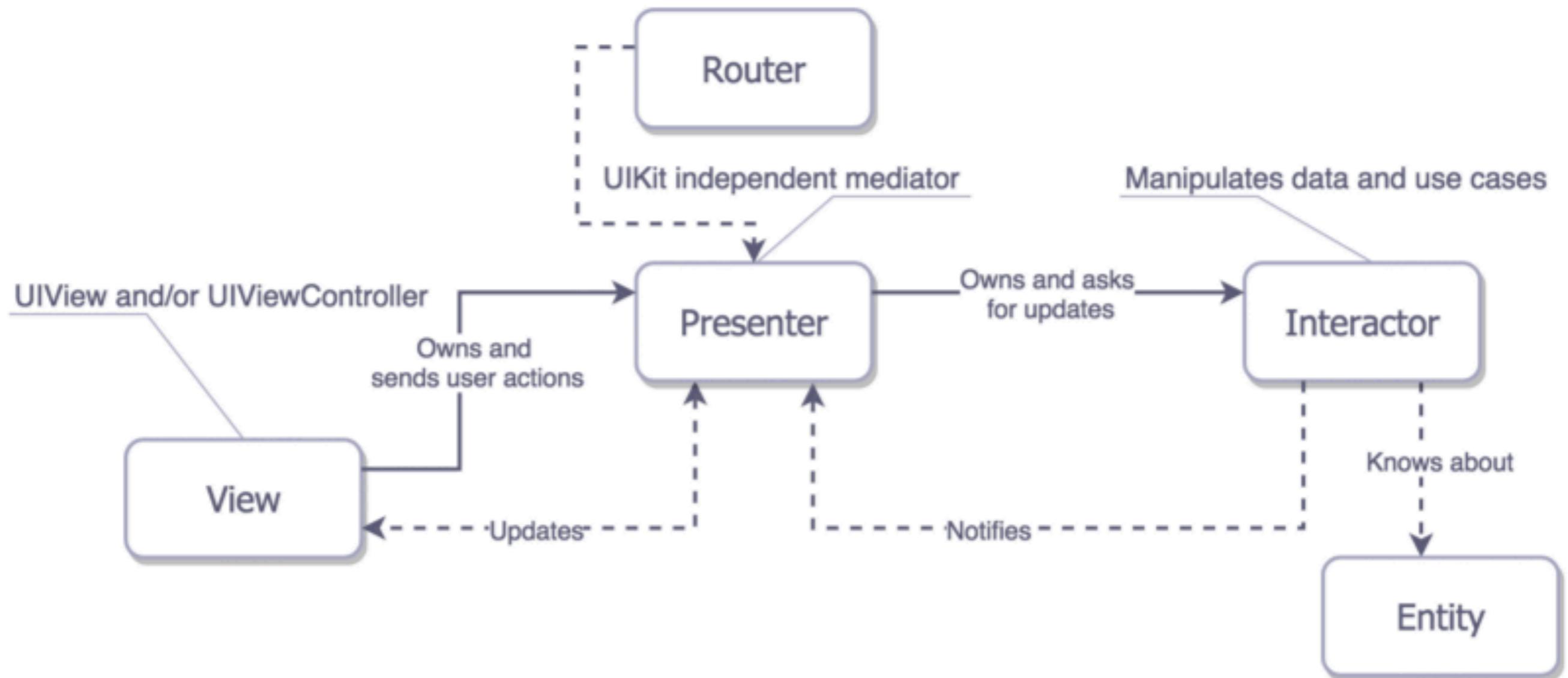
적용했던 아키텍처 예정 - ReactorKit

- 장점
 - 상태가 단방향으로 정리되어 View와 관계된 로직이 깔끔 함.
 - 저자가 전수열님 - 언제든지 한국어로 물어볼 수 있음.
- 단점
 - 프로세스 단위의 아키텍처가 아니라서 아쉬움.

적용했던 아키텍처 예정

- MV(C)
- MVVM
- ReactorKit
- VIPER

적용했던 아키텍처 예정 - VIPER



적용했던 아키텍처 예정 - VIPER

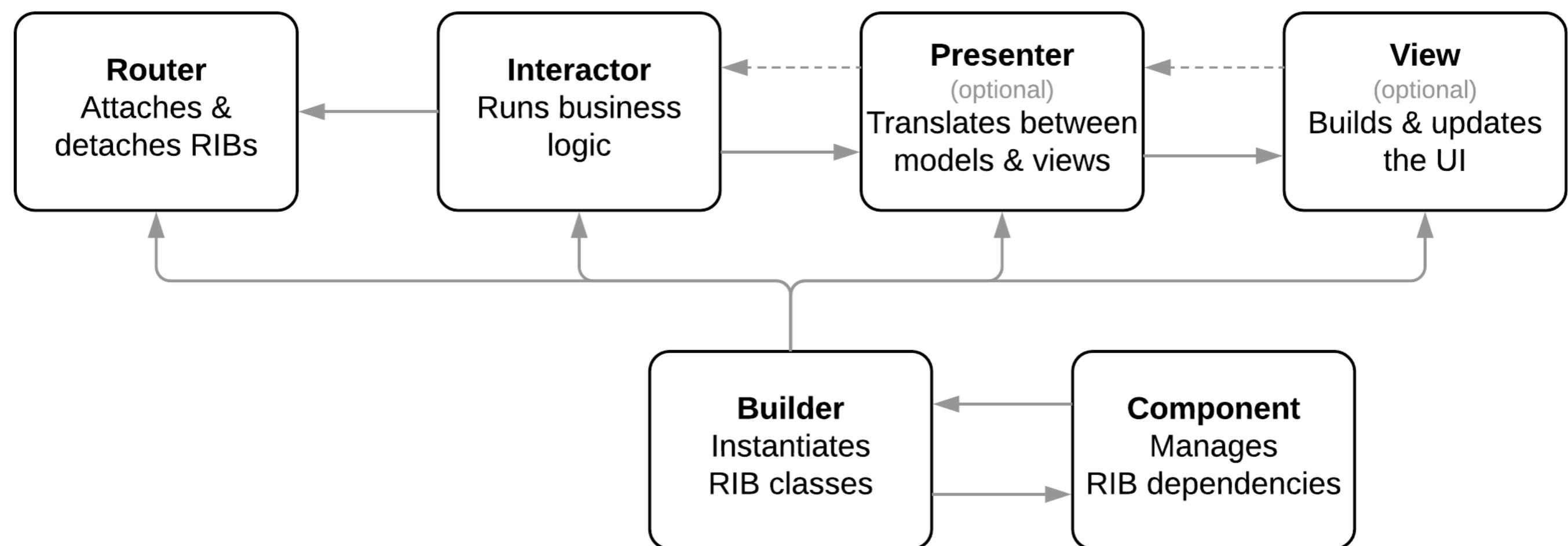
- 장점
 - V, I, P, E, R 역할이 명확하게 구분됨.
- 단점
 - VIPER 아키텍처 설계가 여러 곳으로 난립함.
 - 명확한 가이드, 테스트 코드 템플릿이 제대로 갖춰지고 유지보수 되는 곳이 없음.

Let



RIBs 아키텍처를 선택할 수 밖에 없었던 이유

RIBs Architecture



Related Sessions

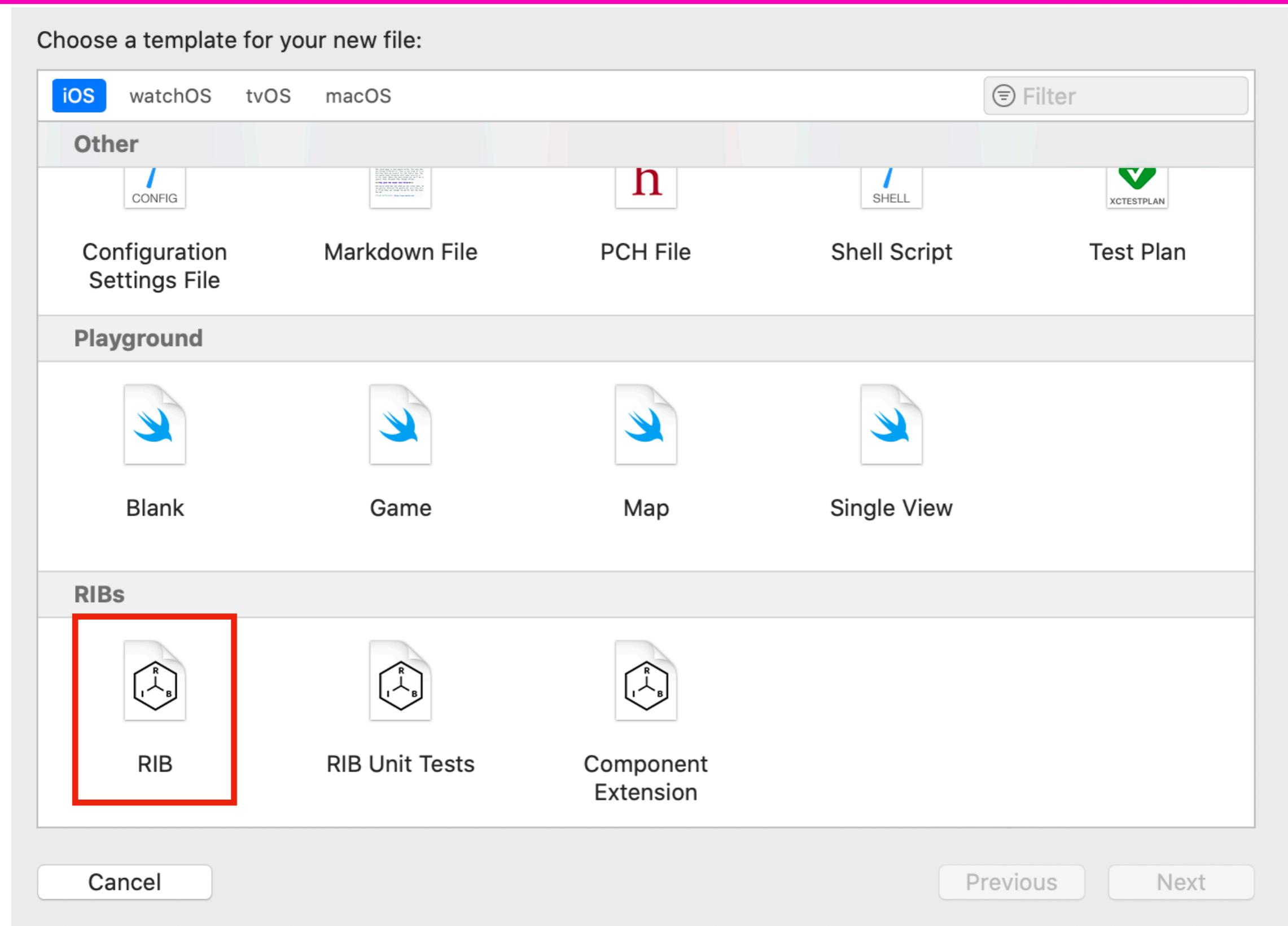
- RxRIBs, Multiplatform architecture with Rx

Session Let'Swift 2018

RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- 템플릿화된 코드 및 테스트 작성
- (강제)프로토콜 지향 프로그래밍
- 의존성 주입 DI
- Viewless RIB을 통한 비즈니스 로직 정리

RIBs 아키텍처를 선택할 수 밖에 없었던 이유



RIBs 아키텍처를 선택할 수 밖에 없었던 이유

Choose options for your new file:

RIB name: LoggedIn

Owns corresponding view

Adds XIB file

Adds Storyboard file

Cancel

Previous

Next

▼	 LoggedIn
	 LoggedInRouter.swift
	 LoggedInViewController.storyboard
	 LoggedInViewController.swift
	 LoggedInBuilder.swift
	 LoggedInInteractor.swift
▼	 LoggedIn
	 LoggedInRouter.swift
	 LoggedInViewController.swift
	 LoggedInBuilder.swift
	 LoggedInInteractor.swift
▼	 LoggedIn
	 LoggedInRouter.swift
	 LoggedInBuilder.swift
	 LoggedInInteractor.swift

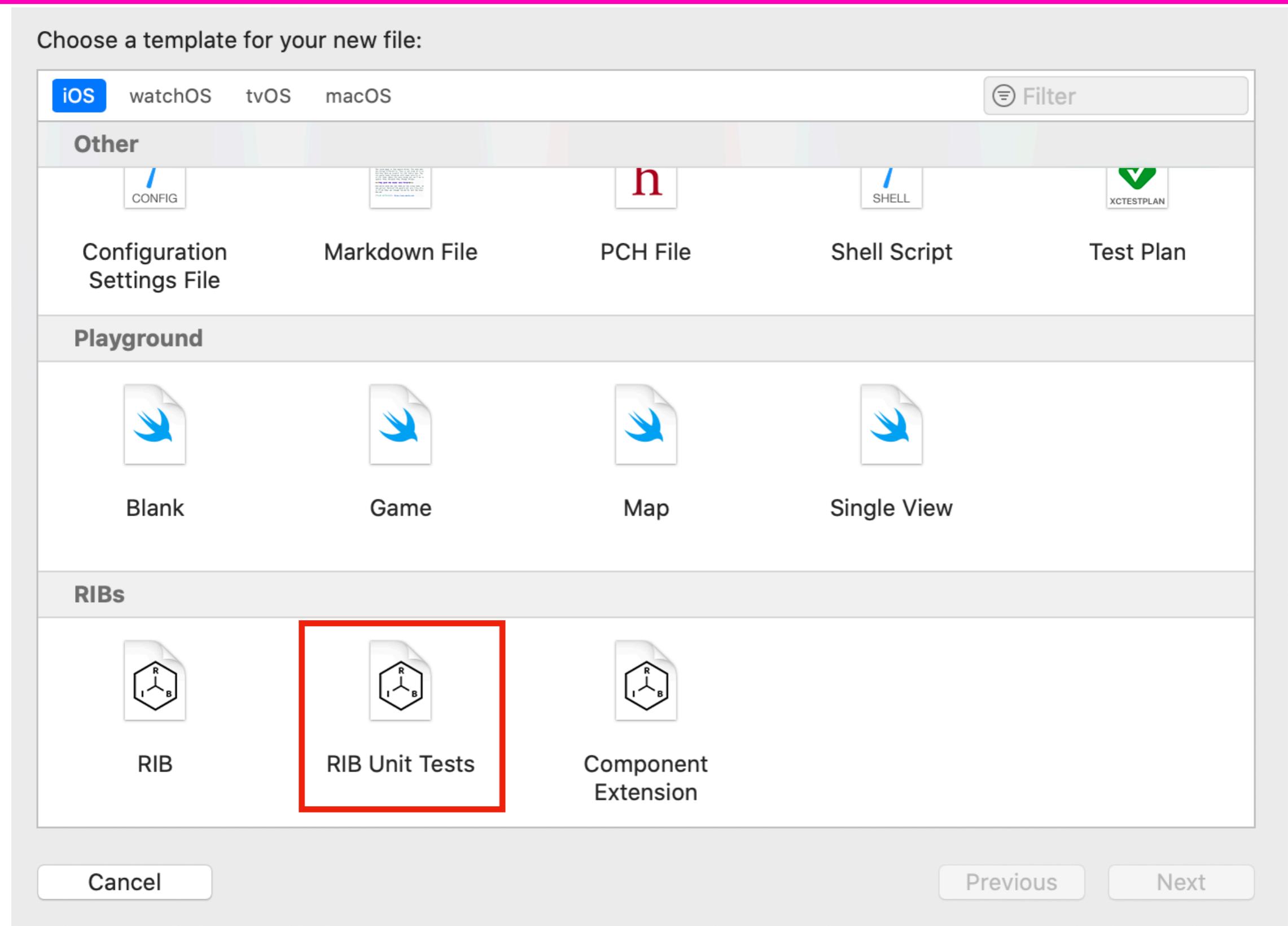
- Own corresponding view
 - Adds Storyboard file
-
- Own corresponding view
 - ✗ Adds Storyboard file
-
- ✗ Own corresponding view
 - ✗ Adds Storyboard file



▼	LoggedIn
	LoggedInRouter.swift
	LoggedInBuilder.swift
	LoggedInInteractor.swift
▼	LoggedIn
	LoggedInRouter.swift
	LoggedInBuilder.swift
	LoggedInInteractor.swift
▼	LoggedIn
	LoggedInRouter.swift
	LoggedInBuilder.swift
	LoggedInInteractor.swift

- Own corresponding view
 - Adds Storyboard file
-
- Own corresponding view
 - ✗ Adds Storyboard file
-
- ✗ Own corresponding view
 - ✗ Adds Storyboard file

RIBs 아키텍처를 선택할 수 밖에 없었던 이유



RIBs 아키텍처를 선택할 수 밖에 없었던 이유

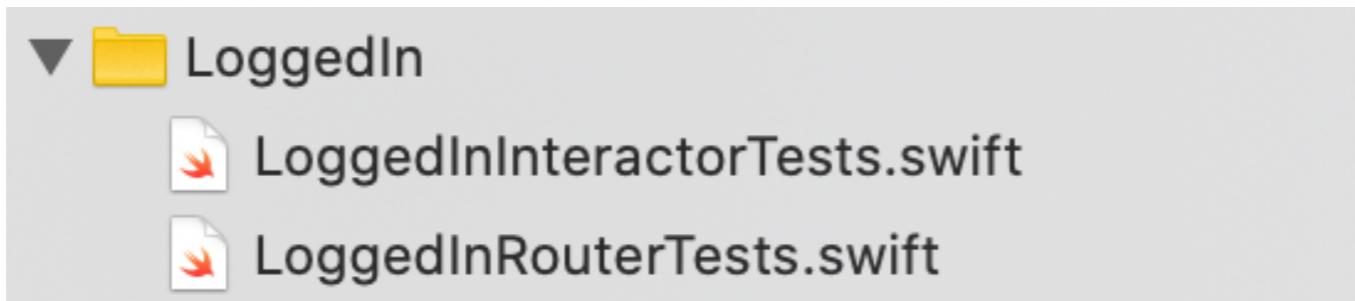
Choose options for your new file:

Name of the RIB to test:

[Cancel](#) [Previous](#) [Next](#)

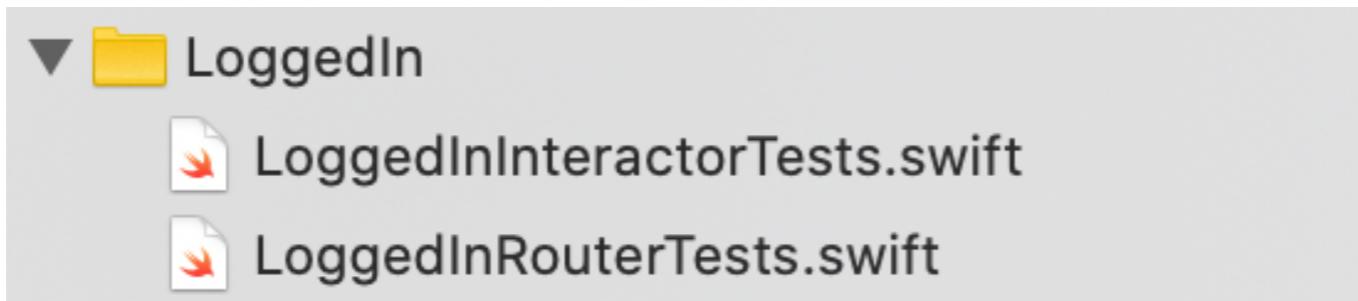
RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- 템플릿화된 코드 및 테스트 작성
 - 기본적인 테스트 코드 추가



RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- 템플릿화된 코드 및 테스트 작성
 - 기본적인 테스트 코드 추가



- 테스트 관련 Mock 샘플 파일이 Tutorial2에서 제공



```
@testable import TicTacToe
import RIBs
import RxSwift
import UIKit

// MARK: - LoggedInBuildableMock class

/// A LoggedInBuildableMock class used for testing.
class LoggedInBuildableMock: LoggedInBuildable {

    // Function Handlers
    var buildHandler: ((_ listener: LoggedInListener) -> LoggedInRouting)?
    var buildCallCount: Int = 0

    init() {
    }

    func build(withListener listener: LoggedInListener) -> LoggedInRouting {
        buildCallCount += 1
        if let buildHandler = buildHandler {
            return buildHandler(listener)
        }
        fatalError("Function build returns a value that can't be handled with a default value and its handler must be set")
    }
}

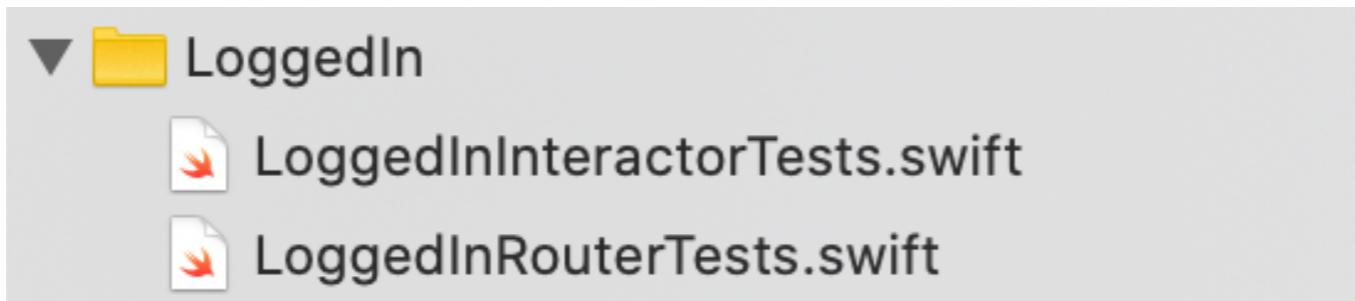
// MARK: - LoggedInInteractableMock class

/// A LoggedInInteractableMock class used for testing.
class LoggedInInteractableMock: LoggedInInteractable {
    // Variables
    var router: LoggedInRouting? { didSet { routerSetCallCount += 1 } }
    var routerSetCallCount = 0
    var listener: LoggedInListener? { didSet { listenerSetCallCount += 1 } }
    var listenerSetCallCount = 0
}
```

RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- 템플릿화된 코드 및 테스트 작성

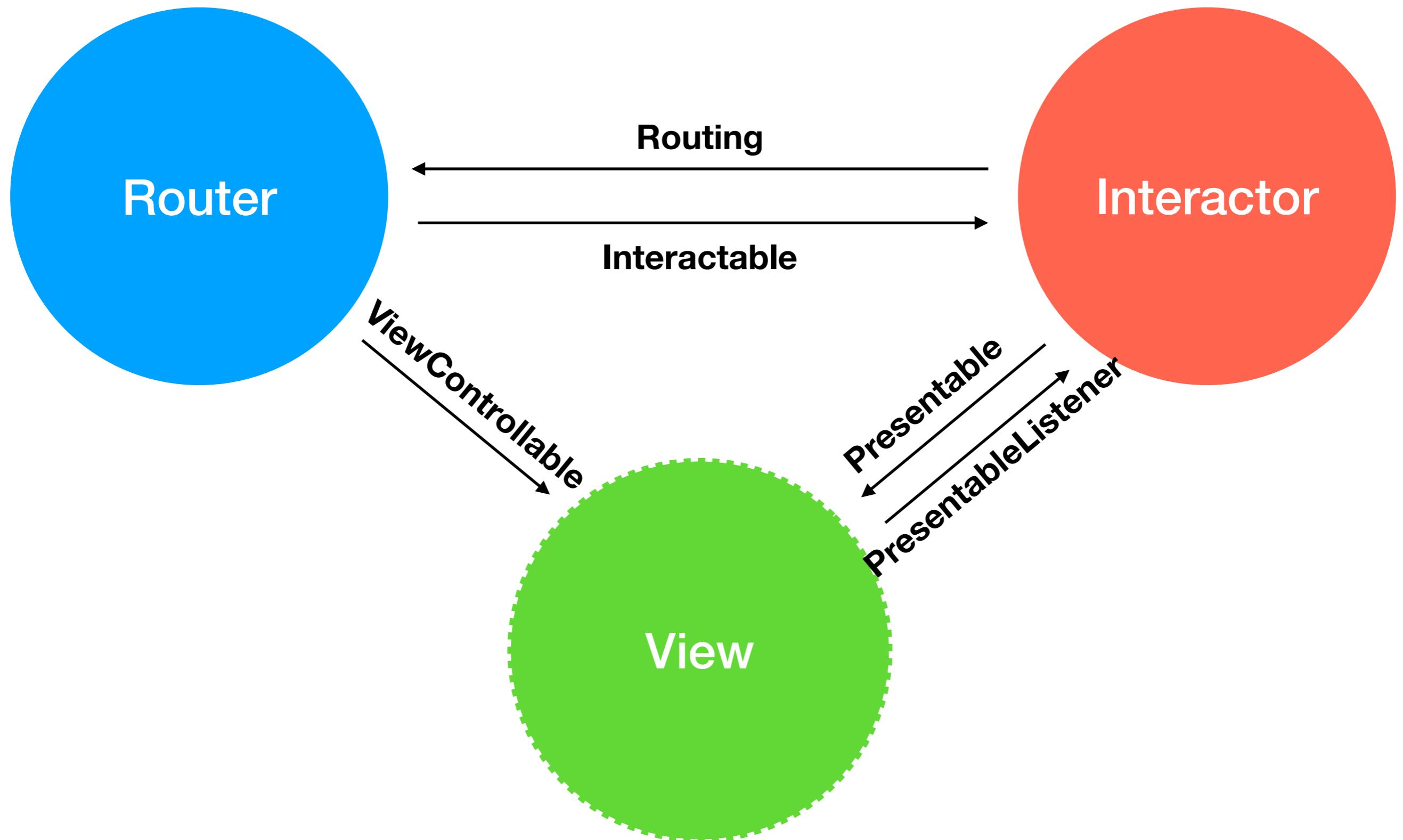
- 기본적인 테스트 코드 추가



- 테스트 관련 Mock 샘플 파일이 Tutorial2에서 제공
 - RIB 파일을 SourceKitten을 이용해 분석하여 Mock 코드 생성 스크립트 작성

RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- 템플릿화된 코드 및 테스트 작성
- (강제)프로토콜 지향 프로그래밍
- 의존성 주입 DI
- Viewless RIB을 통한 비즈니스 로직 정리



```
import RIBs
import RxSwift

protocol LoggedInRouting: ViewableRouting {
}
```

```
protocol LoggedInPresentable: Presentable {
    var listener: LoggedInPresentableListener? { get set }
}
```

```
protocol LoggedInListener: class {

}
```

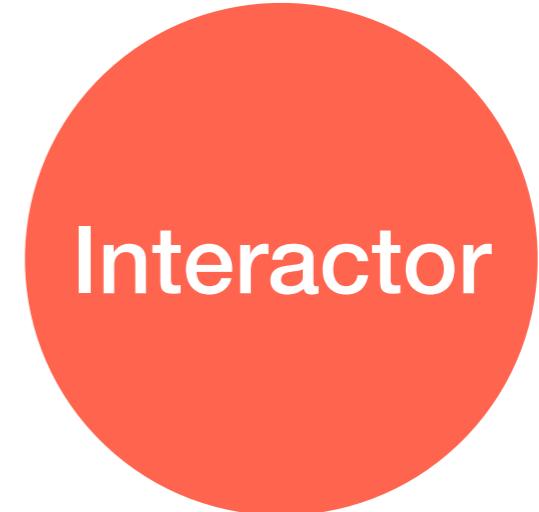
```
final class LoggedInInteractor: PresentableInteractor<LoggedInPresentable>, LoggedInInteractable,
LoggedInPresentableListener {
```

```
weak var router: LoggedInRouting?
weak var listener: LoggedInListener?
```

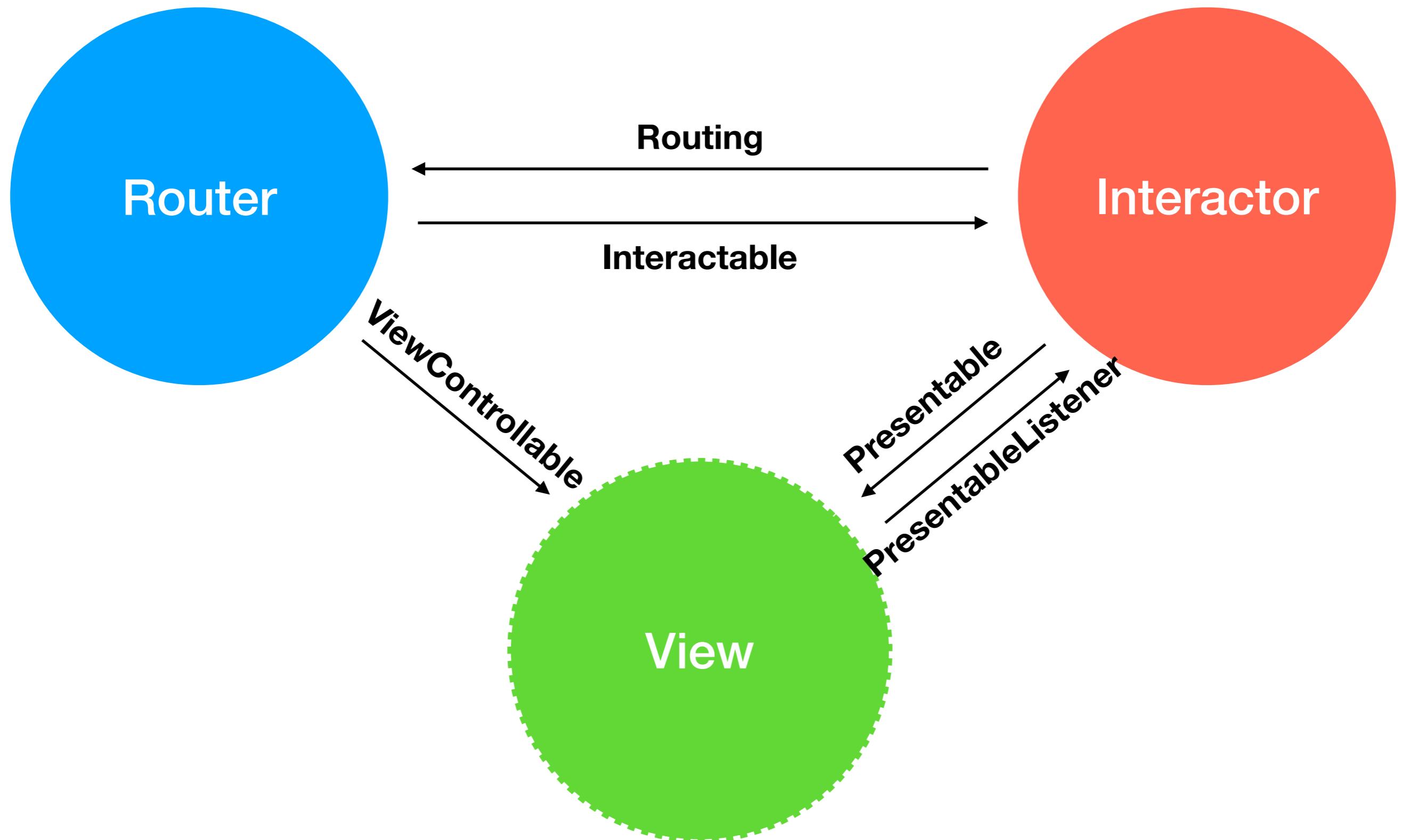
```
override init(presenter: LoggedInPresentable) {
    super.init(presenter: presenter)
    presenter.listener = self
}
```

```
override func didBecomeActive() {
    super.didBecomeActive()
    // TODO: Implement business logic here.
}
```

```
override func willResignActive() {
    super.willResignActive()
    // TODO: Pause any business logic.
}
```



Interactor



Router

```
import RIBs

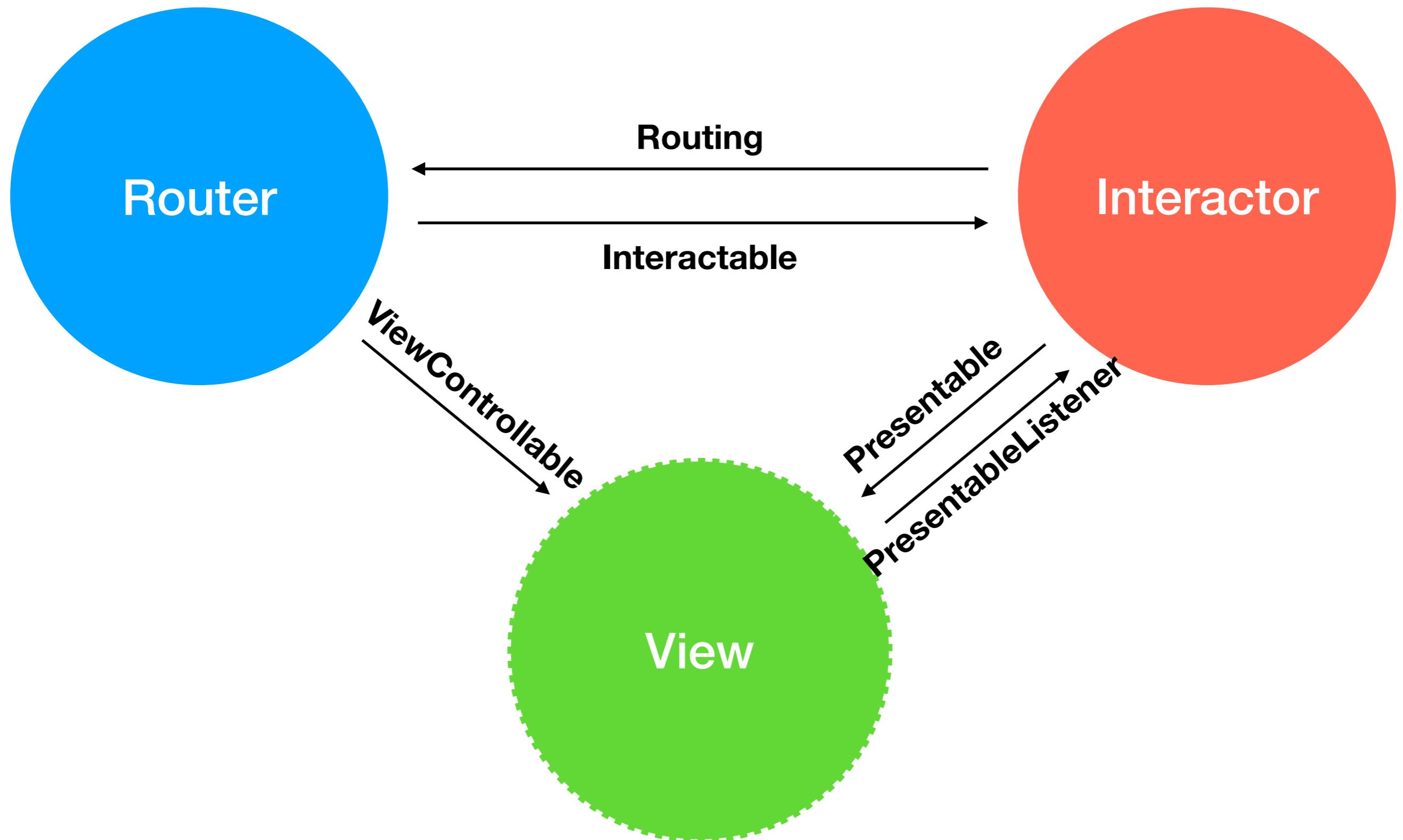
protocol LoggedInInteractable: Interactable {
    var router: LoggedInRouting? { get set }
    var listener: LoggedInListener? { get set }
}

protocol LoggedInViewControllable: ViewControllable {

}

final class LoggedInRouter: ViewableRouter<LoggedInInteractable, LoggedInViewControllable>, LoggedInRouting {

    override init(interactor: LoggedInInteractable, viewController: LoggedInViewControllable) {
        super.init(interactor: interactor, viewController: viewController)
        interactor.router = self
    }
}
```





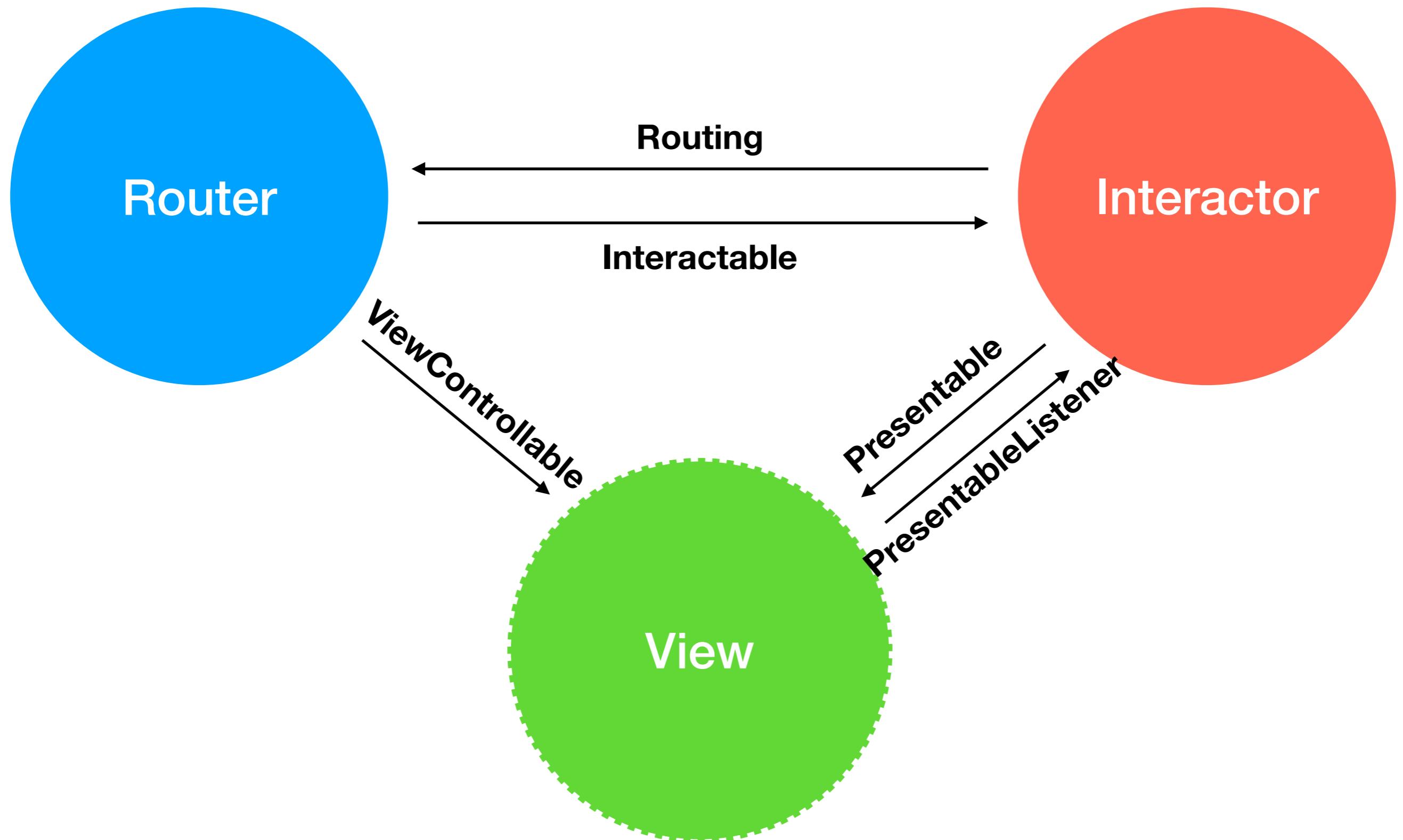
```
import RIBs
import RxSwift
import UIKit

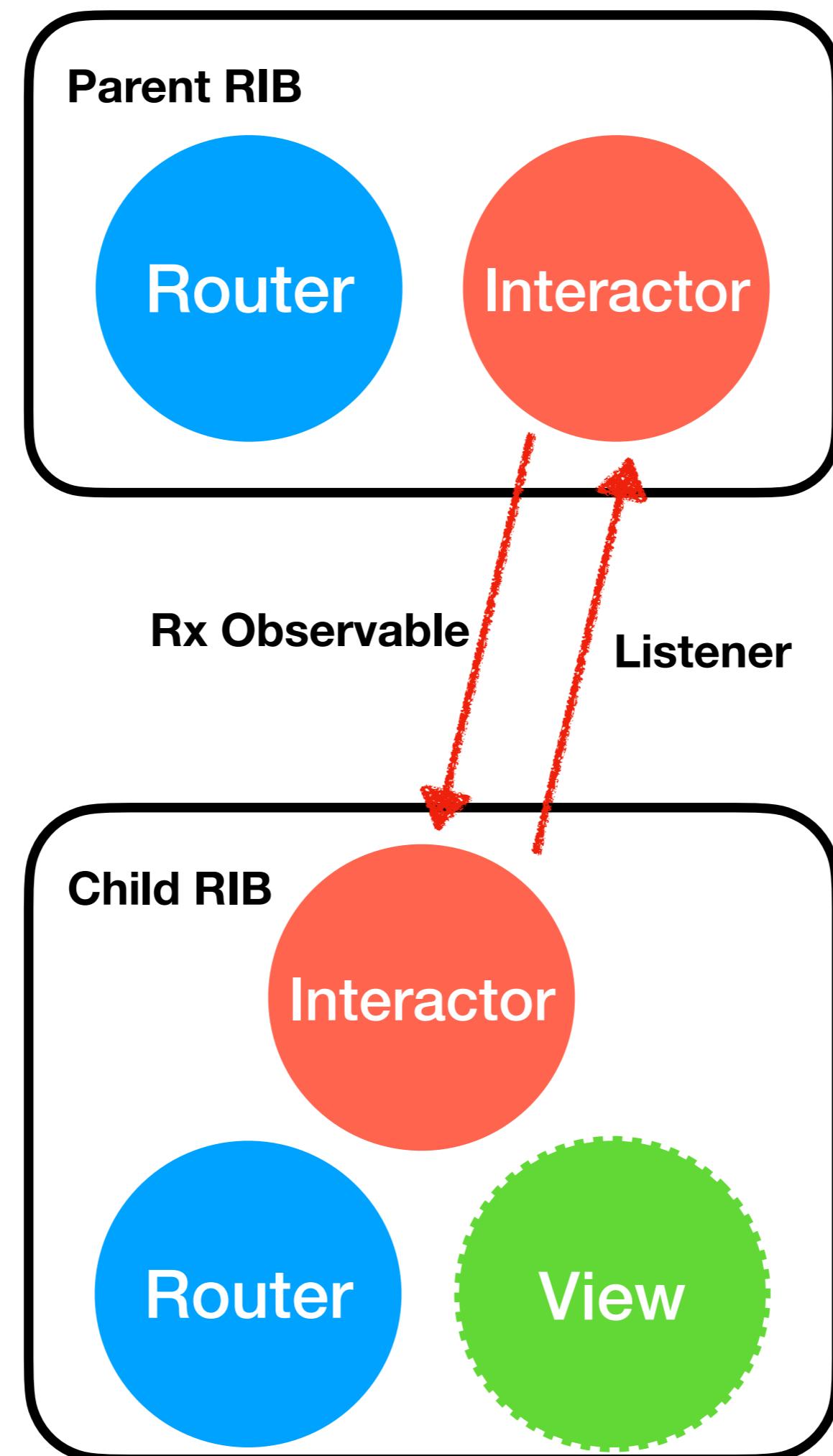
protocol LoggedInPresentableListener: class {
```

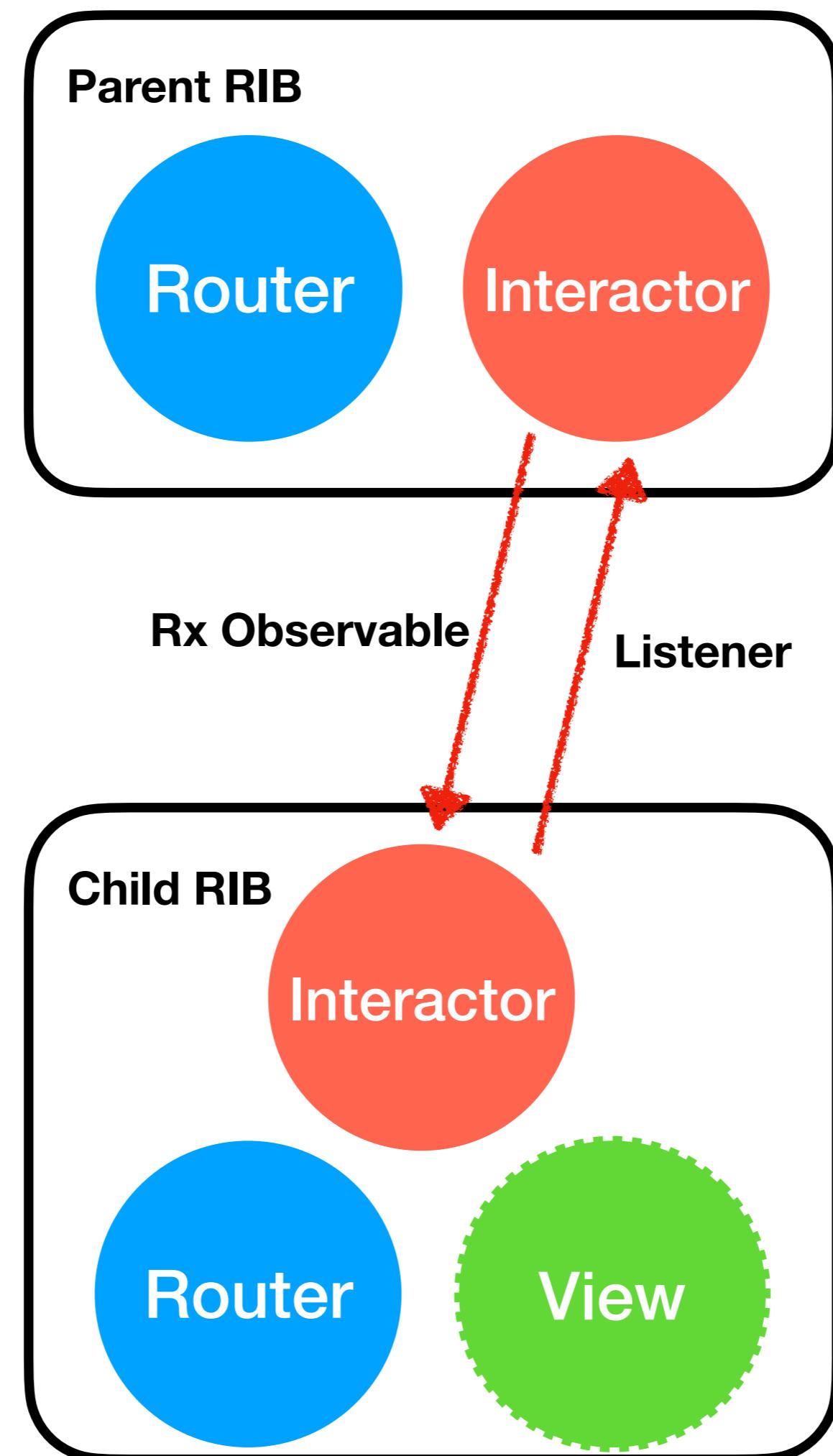
```
}
```

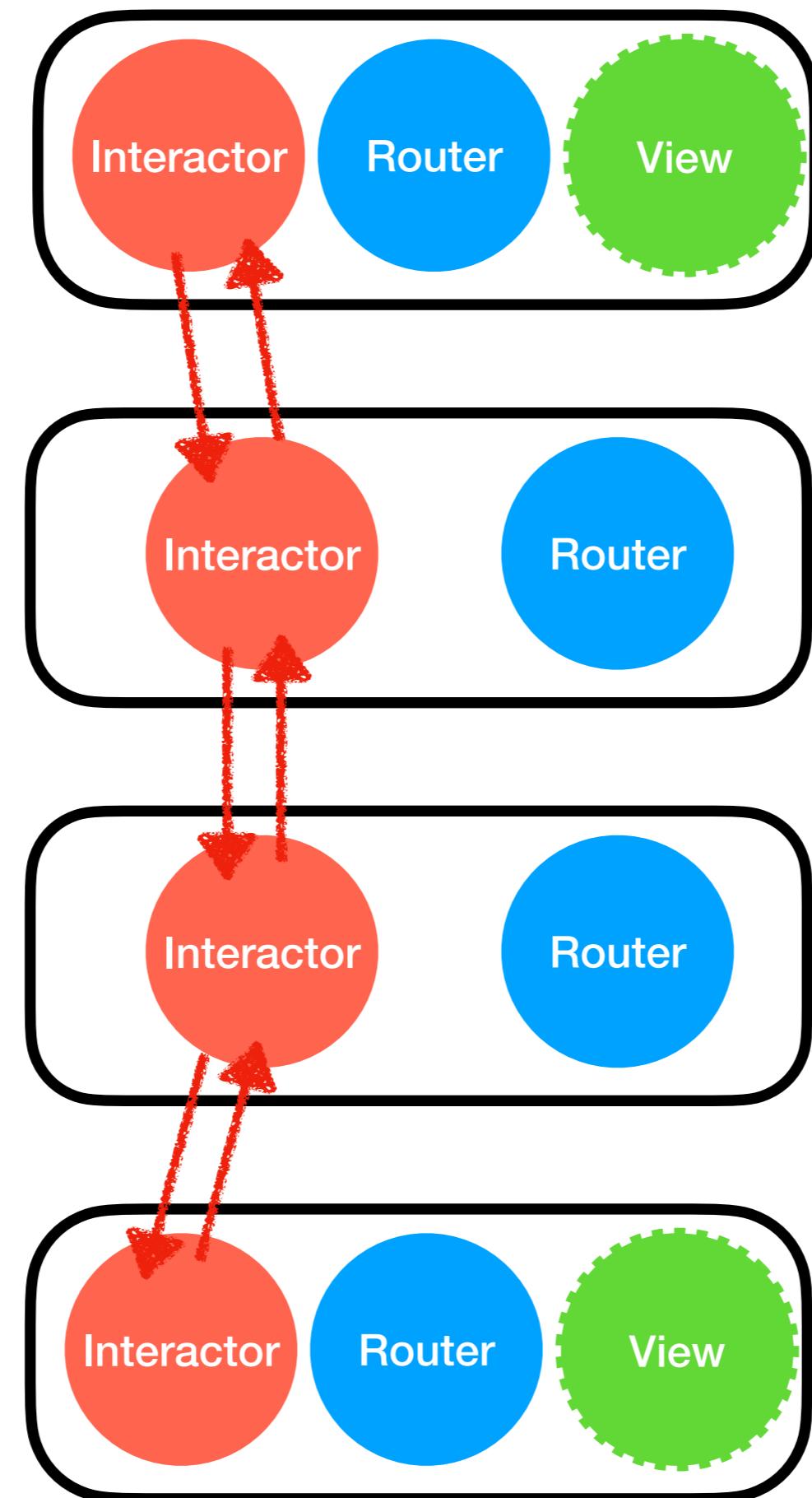
```
final class LoggedInViewController: UIViewController, LoggedInPresentable,
LoggedInViewControllable {

    weak var listener: LoggedInPresentableListener?
}
```









RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- 템플릿화된 코드 및 테스트 작성
- (강제)프로토콜 지향 프로그래밍
- 의존성 주입 DI
- Viewless RIB을 통한 비즈니스 로직 정리

RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- 의존성 주입 DI
 - 해당 RIB에서 필요한 속성을 Dependency에 정의.

The logo consists of the word "Let" in a stylized, handwritten font. The letters are primarily pink with some yellow and white highlights, giving it a vibrant, modern appearance.

```
import RIBs

protocol LoggedInDependency: Dependency {

}

final class LoggedInComponent: Component<LoggedInDependency> {
```



```
import RIBs

protocol LoggedInDependency: Dependency {
    var name: String { get }
    var nickName: String { get }
}

final class LoggedInComponent: Component<LoggedInDependency> {
    fileprivate var name: String {
        return dependency.name
    }
    fileprivate var nickName: String {
        return dependency.name
    }
}
```

**Dependency 확장
name, nickName 필요**

```
import RIBs

protocol LoggedInDependency: Dependency {

}

final class LoggedInComponent: Component<LoggedInDependency> {

}
```

RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- 의존성 주입 DI
 - 해당 RIB에서 필요한 속성을 Dependency에 정의.
 - Dependency에 정의된 요소들을 구현 필요.
 - Dependency가 확장되면 컴파일 에러 발생
 - 해당 RIB을 사용하는 곳은 모든 곳에서 다 작성해야 함.

```
import RIBs

final class UserAccountInputComponent: Component<UserAccountInputDependency>, LoggedInDependency {
    let name: String
    let nickName: String

    init(dependency: UserAccountInputDependency, name: String, nickName: String) {
        self.name = name
        self.nickName = nickName
        super.init(dependency: dependency)
    }
}
```

```
import RIBs

protocol LoggedInDependency: Dependency {
    var name: String { get }
    var nickName: String { get }
}

final class LoggedInComponent: Component<LoggedInDependency> {
    fileprivate var name: String {
        return dependency.name
    }
    fileprivate var nickName: String {
        return dependency.name
    }
}
```

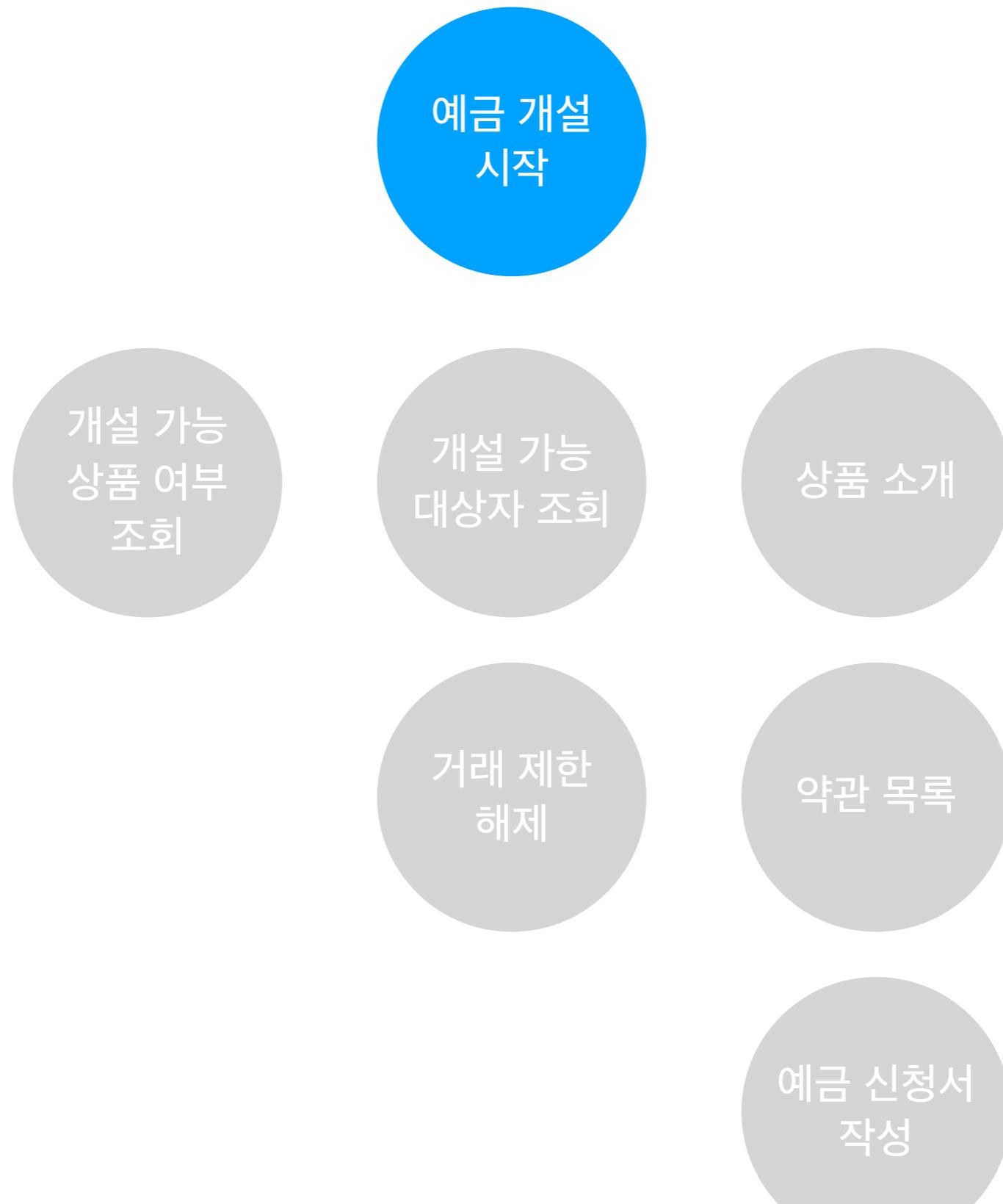
RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- 템플릿화된 코드 및 테스트 작성
- (강제)프로토콜 지향 프로그래밍
- 의존성 주입 DI
- Viewless RIB을 통한 비즈니스 로직 정리

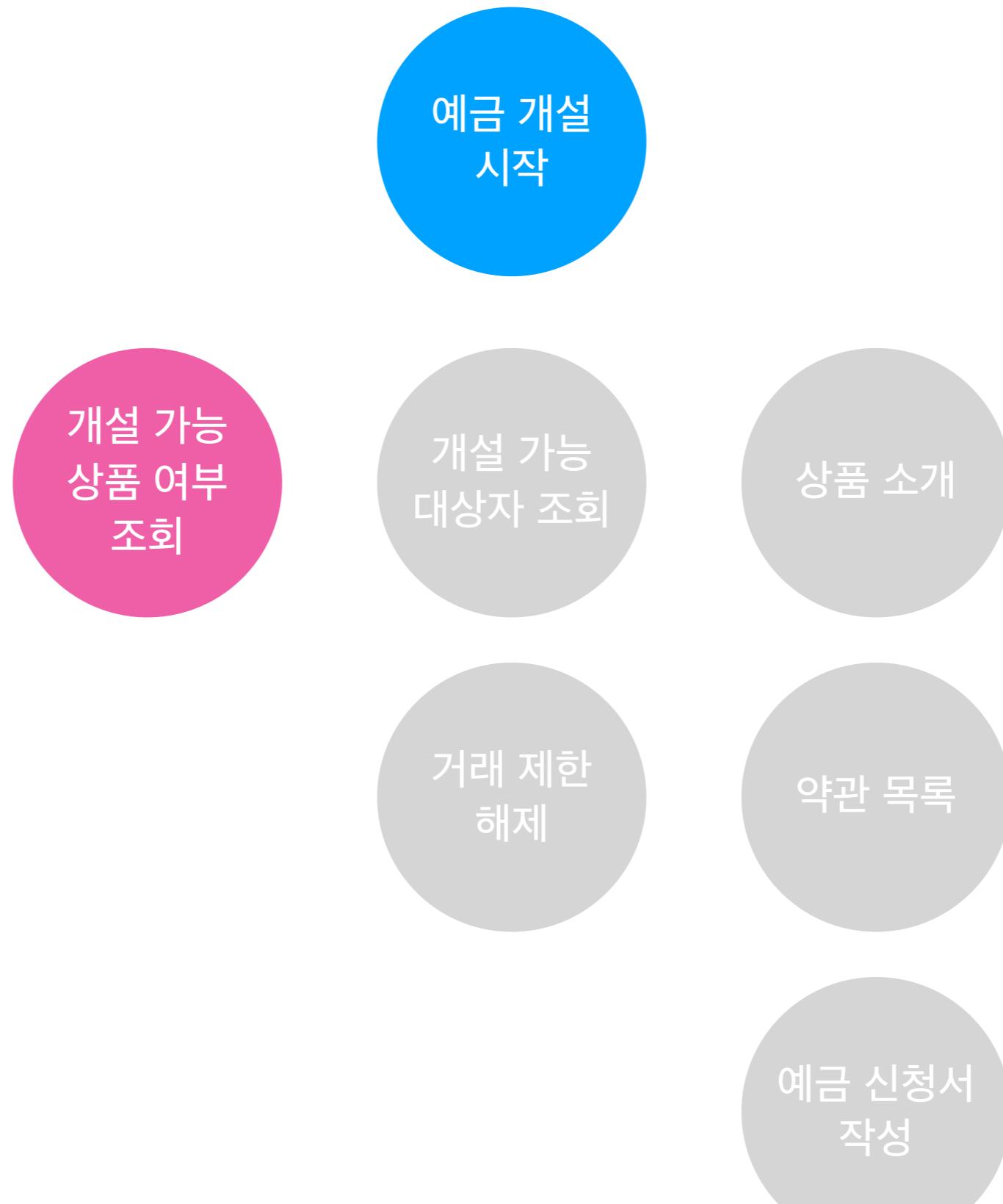
RIBs 아키텍처를 선택할 수 밖에 없었던 이유

- Viewless RIB을 통한 비즈니스 로직 정리
 - 복잡한 로직을 작은 단위의 RIB을 엮어 만듬.

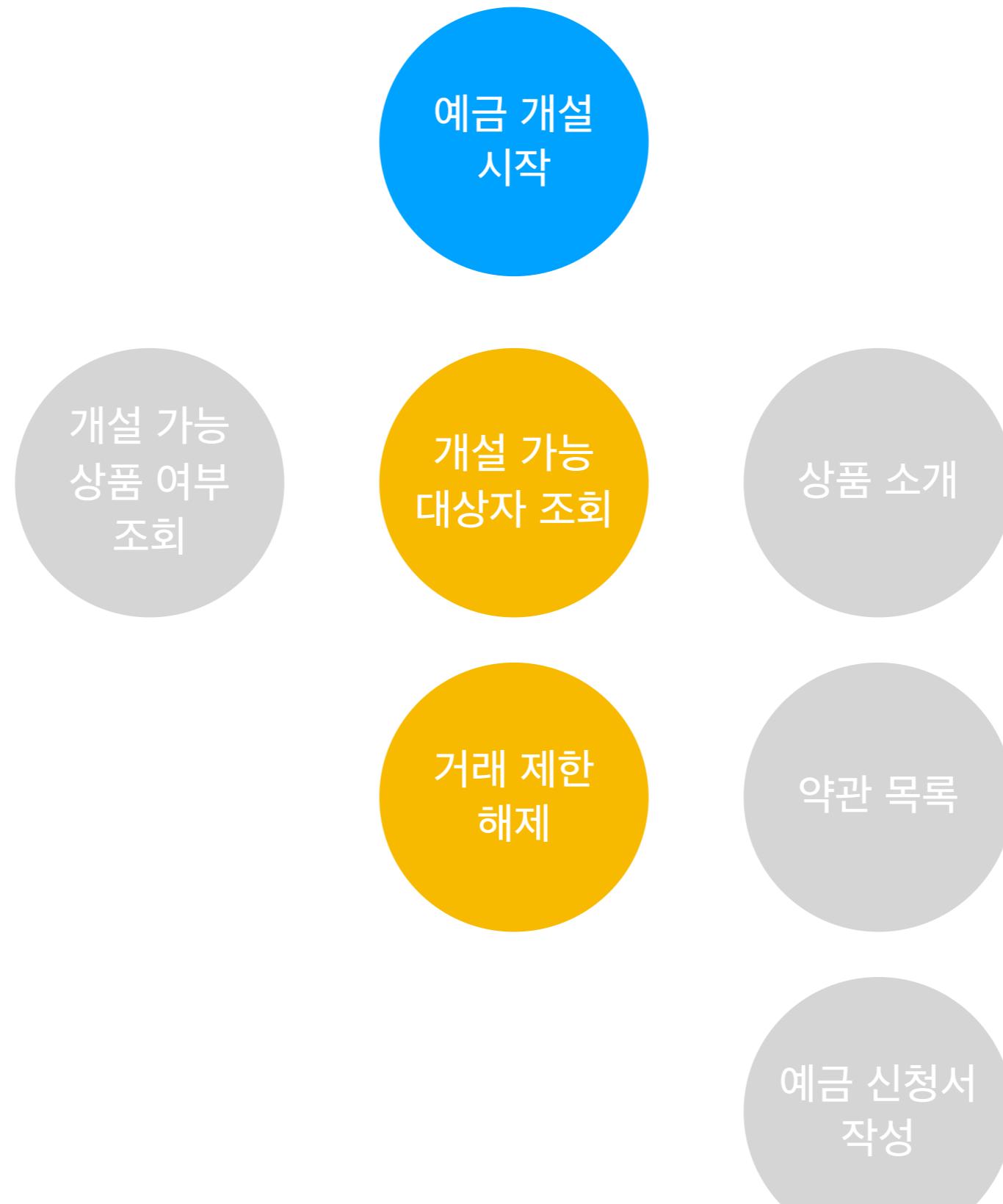
RIBs 아키텍처를 선택할 수 밖에 없었던 이유



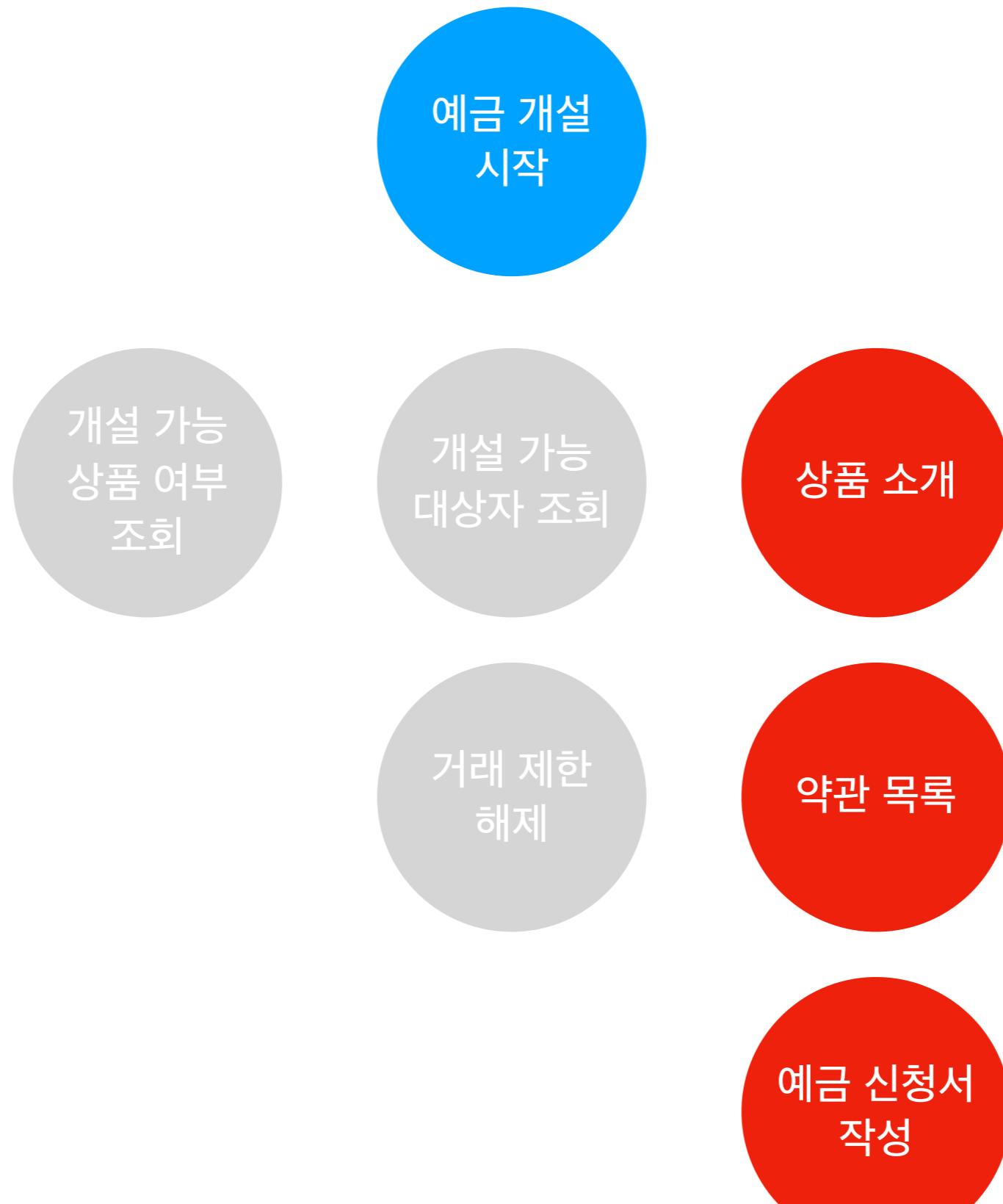
RIBs 아키텍처를 선택할 수 밖에 없었던 이유



RIBs 아키텍처를 선택할 수 밖에 없었던 이유



RIBs 아키텍처를 선택할 수 밖에 없었던 이유



Let



**기존 프로젝트를 RIBs로
바꿀 수 있을까?**

Let



가능합니다.

지금 바꾸고 있으니까요.

어떻게 기존 프로젝트에 RIBs을 적용할까?

- 신규 화면 또는 비즈니스 로직을 RIB으로 변경
- Packaging

어떻게 기존 프로젝트에 RIBs을 적용할까?

- 신규 화면 또는 비즈니스 로직을 RIB으로 변경
 - 비즈니스 로직만 처리하는 RIB을 호출하는 경우

어떻게 기존 프로젝트에 RIBs을 적용할까?

- 신규 화면 또는 비즈니스 로직을 RIB으로 변경
 - 비즈니스 로직만 처리하는 RIB을 호출하는 경우

```
import RIBs

var router: BusinessLogicRouting?

let router = BusinessLogicBuilder(dependency: self).build()
router.interactable.activate()
router.load()
self.router = router
```

어떻게 기존 프로젝트에 RIBs을 적용할까?

- 신규 화면 또는 비즈니스 로직을 RIB으로 변경
 - 비즈니스 로직만 처리하는 RIB을 호출하는 경우
 - 화면 모듈을 RIB으로 연결하는 경우

```
import RIBs

var router: ARSAuthenticationViewableRouting?

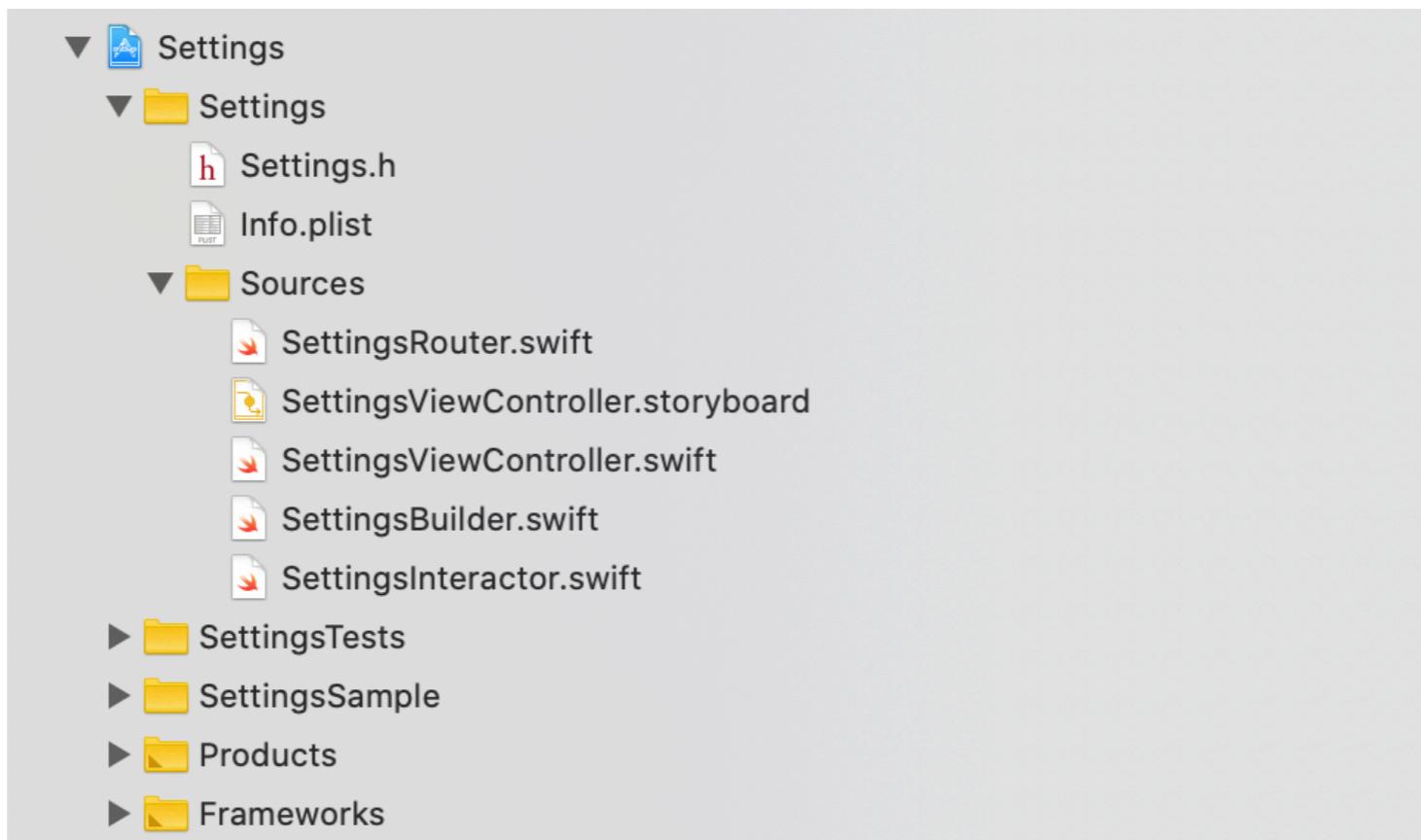
let router = ARSAuthenticationBuilder(dependency: self).build()
router.interactable.activate()
router.load()
self.router = router
self.present(router.uiviewController, animated: true, completion: nil)
```

어떻게 기존 프로젝트에 RIBs을 적용할까?

- 신규 화면 또는 비즈니스 로직을 RIB으로 변경
- Packaging

어떻게 기존 프로젝트에 RIBs을 적용할까?

- Packaging
 - 프로젝트를 별도로 만들어 패키징하고 RIB을 작게 시작.
 - 코드 작성 및 테스트 코드 작성이 쉬워짐.



Related Sessions

- Static, Dynamic Framework 그리고 Encapsulation Session

Letusgo
2019 Fall

- 프레임워크 주도 개발

Session

Letusgo
2019 Spring

어떻게 기존 프로젝트에 RIBs을 적용할까?

- 신규 화면 또는 비즈니스 로직을 RIB으로 변경
- 격리화 & 패키징
- ~~찾아가는 서비스(고기 필요)~~

Let



요약

요약 - RIB Architecture

- 장점
 - 좀 더 템플릿화가 잘 되어 있음.
 - 프로토콜 지향 프로그래밍을 할 수 밖에 없는 구조.
 - 복잡한 로직 흐름을 모듈화 및 재사용에 강함.
- 단점
 - 프로토콜 지향 프로그래밍으로 꼭 해야하지 하는 이해의 문제
 - 파일이 많이 생성되는 문제.

Reference

- LetsSwift 2018 - VCNC RIBs
- Github : Uber/RIBs
- Uber Engineering Blog

Let



QnA

Let

