Mash-Up
iOS Team

Modular Architecture
(with Tuist, RIBs, Clean Arthitecture)

Created by 김찬수

목차

RIBs와

RIBs

저희가 RIBs를 선택한 이유는요…!!

병렬 프로그래밍을 위해서 인터페이스를 잘 구축할 수 있는 아키텍처라고 생각해서!

RIBs

그래서 RIBs가 뭐냐면요

Router + Interactor + Builder

(Presenter?, View?)

RIBs의 특징

- iOS와 Andriod간에 모두 사용가능한 크로스 플랫폼
- global state 최소화와 OCP에 적합 (open-closed principle)
- testability, isolation
- **code template**(generation)과 **memory leak** 탐지 👍

RIBs

등장 배경

MVC 패턴의 한계를 느끼고 **VIPER** 패턴을 개선시켜 만듦

RIBs

등장 배경

MVC 패턴의 한계를 느끼고 **VIPER** 패턴을 개선시켜 만듦

⟶ **그럼 VIPER의 단점은 뭔데?**

RIBs

등장 배경

MVC 패턴의 한계를 느끼고 **VIPER** 패턴을 개선시켜 만듦

⟶ **그럼 VIPER의 단점은 뭔데?**

- view기반 로직으로 작동한다는 것
  → 전체적인 앱의 트리가 뷰를 기준으로 만들어졌기 때문

- view와 logic이 강하게 결합돼있어서 view less한 구조를 만들 수 없음

RIBs

등장 배경

MVC 패턴의 한계를 느끼고 **VIPER** 패턴을 개선시켜 만듦

⟶ **그럼 VIPER의 단점은 뭔데?**

- view기반 로직으로 작동한다는 것
  → 전체적인 앱의 트리가 뷰를 기준으로 만들어졌기 때문

- view와 logic이 강하게 결합돼있어서 view less한 구조를 만들 수 없음

따라서, 리블렛의 핵심은 view logic이 아닌,
비지니스 로직에 따라 routing이 이루어져야한다는 것이다.

RIBs

흐름~ (⭐핵심만)

상위 (Topup) 에서

```
public final class TopupBuilder: Builder<TopupDependency>, TopupBuildable {

    public override init(dependency: TopupDependency) {
        super.init(dependency: dependency)
    }

    public func build(withListener listener: TopupListener) -> Routing {

        let paymentMethodStream = BehaviorRelay<PaymentMethod>(
            value: PaymentMethod(
                id: "",
                name: "",
                digits: "",
                color: "",
                isPrimary: false
            )
        )
        let component = TopupComponent(dependency: dependency, paymentMethodStream: paymentMethodStream)
        let interactor = TopupInteractor(dependency: component)
        interactor.listener = listener

        let enterAmountBuilder = EnterAmountBuilder(dependency: component)

        let cardOnFileBuilder = CardOnFileBuilder(dependency: component)

        return TopupRouter(
            interactor: interactor,
            viewController: component.topupBaseViewController,
            addpaymentbuildable: component.addPaymentMethodBuildable,
            enterAmountBuildable: enterAmountBuilder,
            cardOnFileBuildable: cardOnFileBuilder
        )
    }
}
```

자식 (cardOnFile)RIB의 builder객체 인스턴스화

RIBs 구성요소: Builder

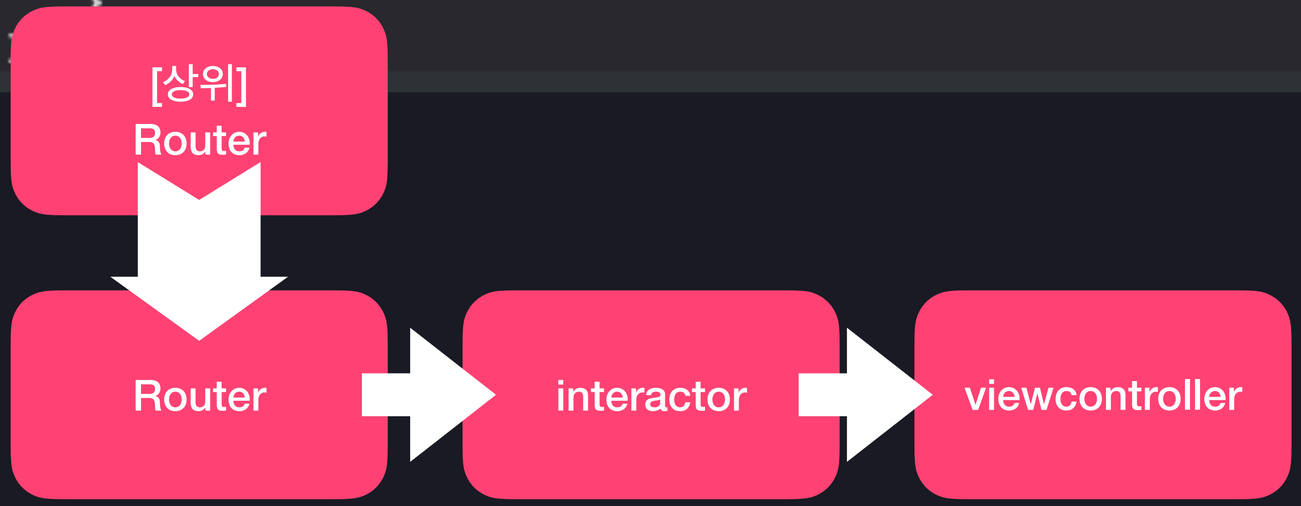
- 본인의 모든 RIB의 구성 클래스 (Interactor, Router, Viewcontroller, Component)
  각 RIB의 자식에 대한 빌더를 인스턴스화함
  like Factory Pattern 😎

RIBs

흐름~ (⭐핵심만)

```swift
final class CardOnFileBuilder: Builder<CardOnFileDependency>, CardOnFileBuildable {

    override init(dependency: CardOnFileDependency) {
        super.init(dependency: dependency)
    }

    func build(withListener listener: CardOnFileListener, paymentMethods: [PaymentMethod]) -> ViewableRouting {
        let _ = CardOnFileComponent(dependency: dependency)
        let viewController = CardOnFileViewController()
        let interactor = CardOnFileInteractor(presenter: viewController, paymentMethods: paymentMethods)
        interactor.listener = listener
        return CardOnFileRouter(interactor: interactor, viewController: viewController)
    }
```

build함수에서는 component, viewcontroller, interactor, router 객체 인스턴스화



RIBs

흐름~ (⭐핵심만)

어떤 상황에 interactor에서 router에게 attach를 알리면 ~

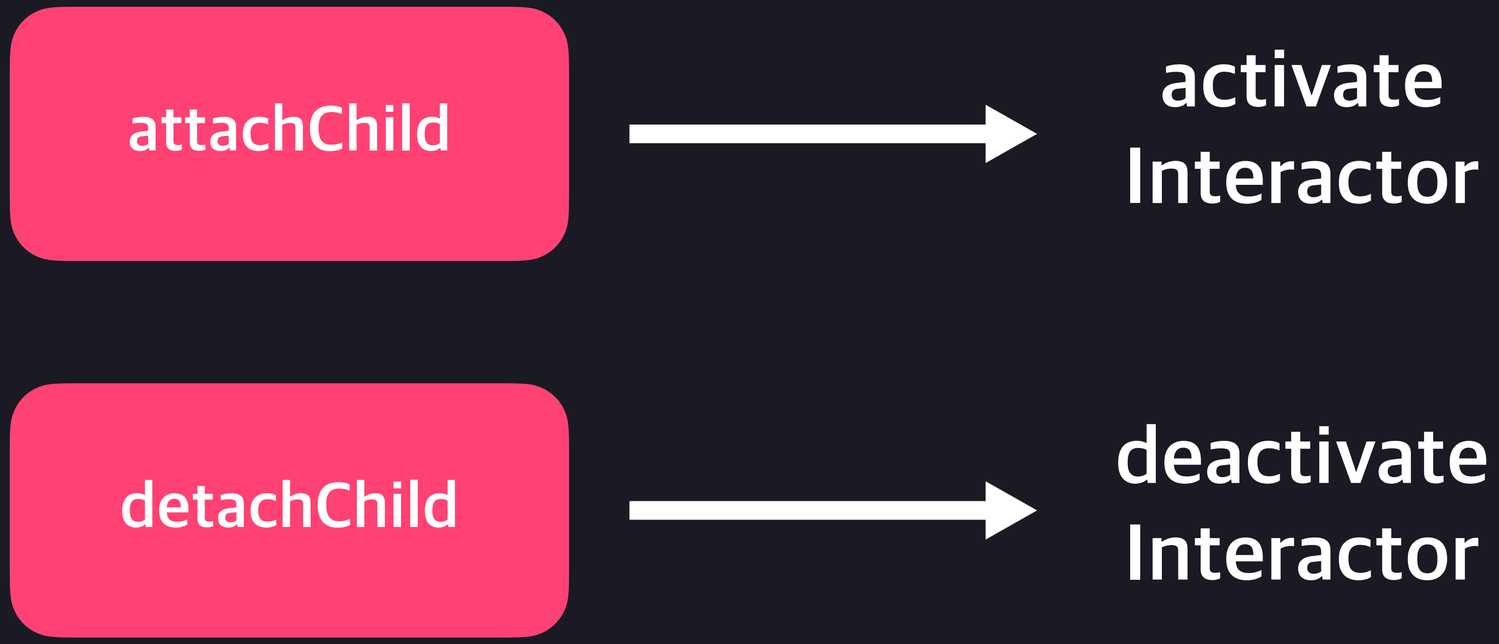
router에서 자식(cardOnFile)을 attach합니다

```
final class TopupRouter: Router<TopupInteractable>, TopupRouting {

    func attachCardOnFile(paymentMethod: [PaymentMethod]) {
        if cardOnFileRouting != nil { return }

        let router = cardOnFileBuildable.build(withListener: interactor, paymentMethods: paymentMethod)
        navigationControllable?.pushViewController(router.viewControllable, animated: true)
        cardOnFileRouting = router
        attachChild(router)
    }
```

자식RIB의 bulider의 build함수를 호출하여
return받은 router를

RIBs 구성요소: Router

- Interactor를 listen해서 자식 RIB을 만들어 붙이거나 떼어내며 tree를 그림



RIBs

흐름~ (⭐핵심만)

어떤 상황에 interactor에서 router에게 attach를 알리면 ~

router에서 자식(cardOnFile)을 attach합니다

```
final class TopupRouter: Router<TopupInteractable>, TopupRouting {

    func attachCardOnFile(paymentMethod: [PaymentMethod]) {
        if cardOnFileRouting != nil { return }

        let router = cardOnFileBuildable.build(withListener: interactor, paymentMethods: paymentMethod)
        navigationControllable?.pushViewController(router.viewControllable, animated: true)
        cardOnFileRouting = router
        attachChild(router)
    }
```

자식RIB의 bulider의 build함수를 호출하여 return받은 router를

attachChild를 합니다.

RIBs

흐름~ (⭐핵심만)

child == CardOnRouting

```
/// - parameter child: The child router to attach.
public final func attachChild(_ child: Routing) {
    assert(!children.contains(where: { $0 === child }), "Attempt to attach child: \(child), which is already attached to \(self).")
```

children 배열에 추가

```
    children.append(child)
```

interactor의 activate 시킨다 (didBecomeActivate에서 콜백받을 수 있음)

```
    // We need to make sure the RIB is activated before letting it attach immutable children.
    child.interactable.activate()
    child.load()
}
```

router를 load시킨다 (didLoad에서 콜밷받을 수 있음)

```
private func setupViewControllerLeakDetection() {
    let disposable = interactable.isActiveStream
        // Do not retain self here to guarantee execution. Retaining self will cause the dispose bag to never be
        // disposed, thus self is never deallocated. Also cannot just store the disposable and call dispose(),
        // since we want to keep the subscription alive until deallocation, in case the router is re-attached.
        // Using weak does require the router to be retained until its interactor is deactivated.
        .subscribe(onNext: { [weak self] (isActive: Bool) in
            guard let strongSelf = self else {
                return
            }

            strongSelf.viewControllerDisappearExpectation?.cancel()
            strongSelf.viewControllerDisappearExpectation = nil

            if !isActive {
                let viewController = strongSelf.viewControllable.uiviewController
                strongSelf.viewControllerDisappearExpectation = LeakDetector.instance.expectViewControllerDisappear(viewController: viewControll
            }
        })
    = deinitDisposable.insert(disposable)
```

viewController의 leak detector가 작동됨

RIBs 구성요소: Interactor

- 비지니스 로직 처리. 😲

흐름~ (⭐핵심만)

어떤 상황에 interactor에서 router에게 detach를 알리면 ~

```
final class TopupRouter: Router<TopupInteractable>, TopupRouting {
    func detachCardOnFile() {
        guard let router = cardOnFileRouting else { return }

        navigationControllable?.popViewController(animated: true)
        detachChild(router)
        cardOnFileRouting = nil
    }
```

router에서 자식(cardOnFile)을 detach합니다

RIBs

흐름~ (⭐핵심만)

```
open class Router<InteractorType>: Routing {

    /// Detaches the given `Router` from the tree.
    ///
    /// - parameter child: The child `Router` to detach.
    public final func detachChild(_ child: Routing) {
        child.interactable.deactivate()
        children.removeElementByReference(child)
    }
```

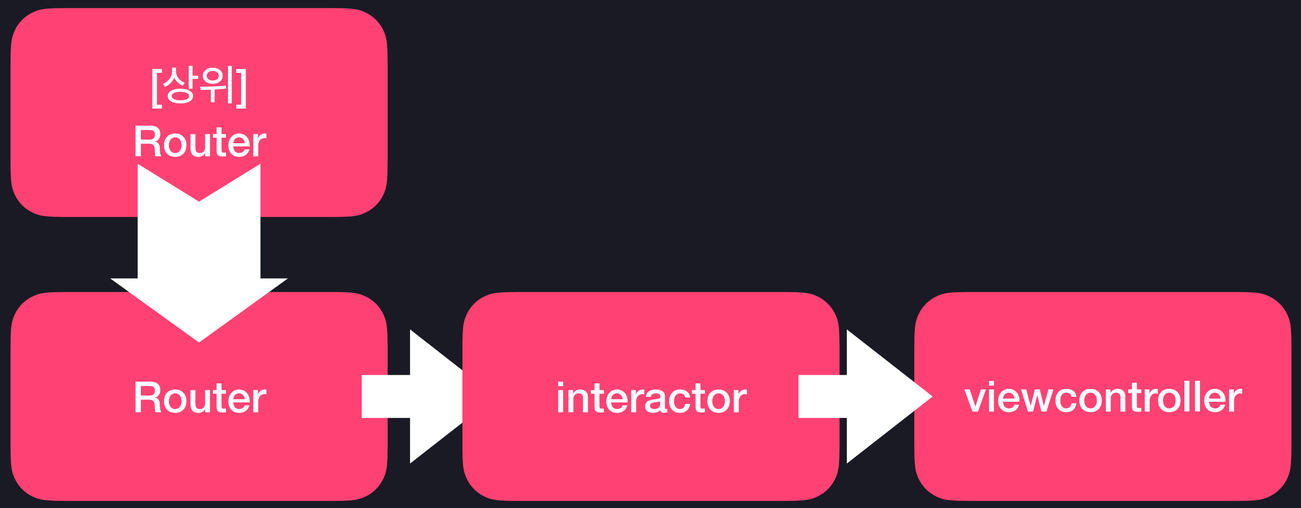
interactor를 deactivate한다.
(willResignActive에서 콜백)

attach할 때 넣은 children 배열에서 remove

-> router가 deinit

RIBs

흐름~ (⭐핵심만)



~~귀찮아서~~
더 자세한 로직은 여러분 숙제입니다

router의 reference count가 없어지니!
router가 deinit되고
interactor가 deinit되고
viewcontroller가 deinit되고

RIBs 구성요소: Presenter (optional)

- Interactor와 View 사이의 통신 관리 역할
- Business Model → View Model 변환하는 sate less class
- 작은 화면에서는 불필요 할 수도 있음
  → 이런 경우에는 이 역할을 View 혹은 Interactor에서 담당

RIBs 구성요소: View(Controller) (optional)

우리가 아는 view

Clean Architecture를

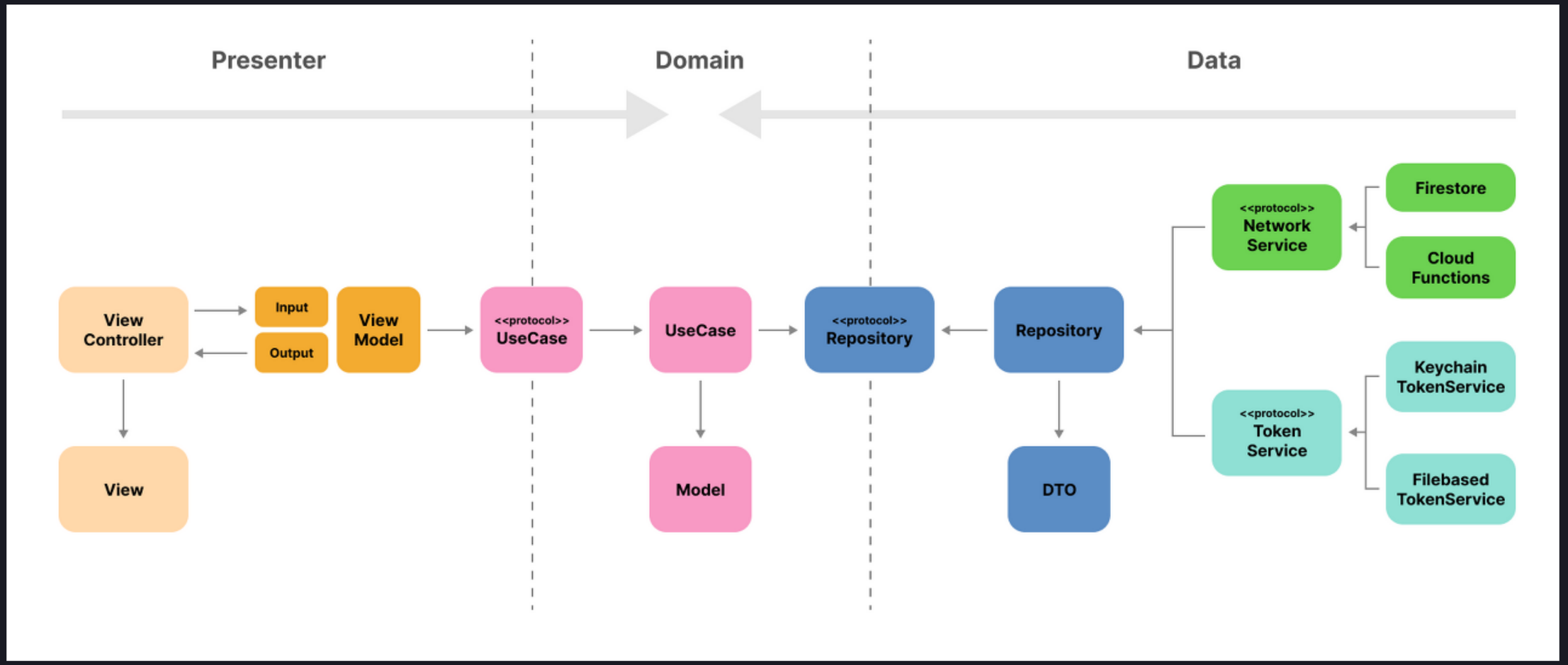
Clean Architecture

ios-chansoo “MVVM-C + Clean Architecture 도입기”

본문은 여기서 확인하세요 ^^

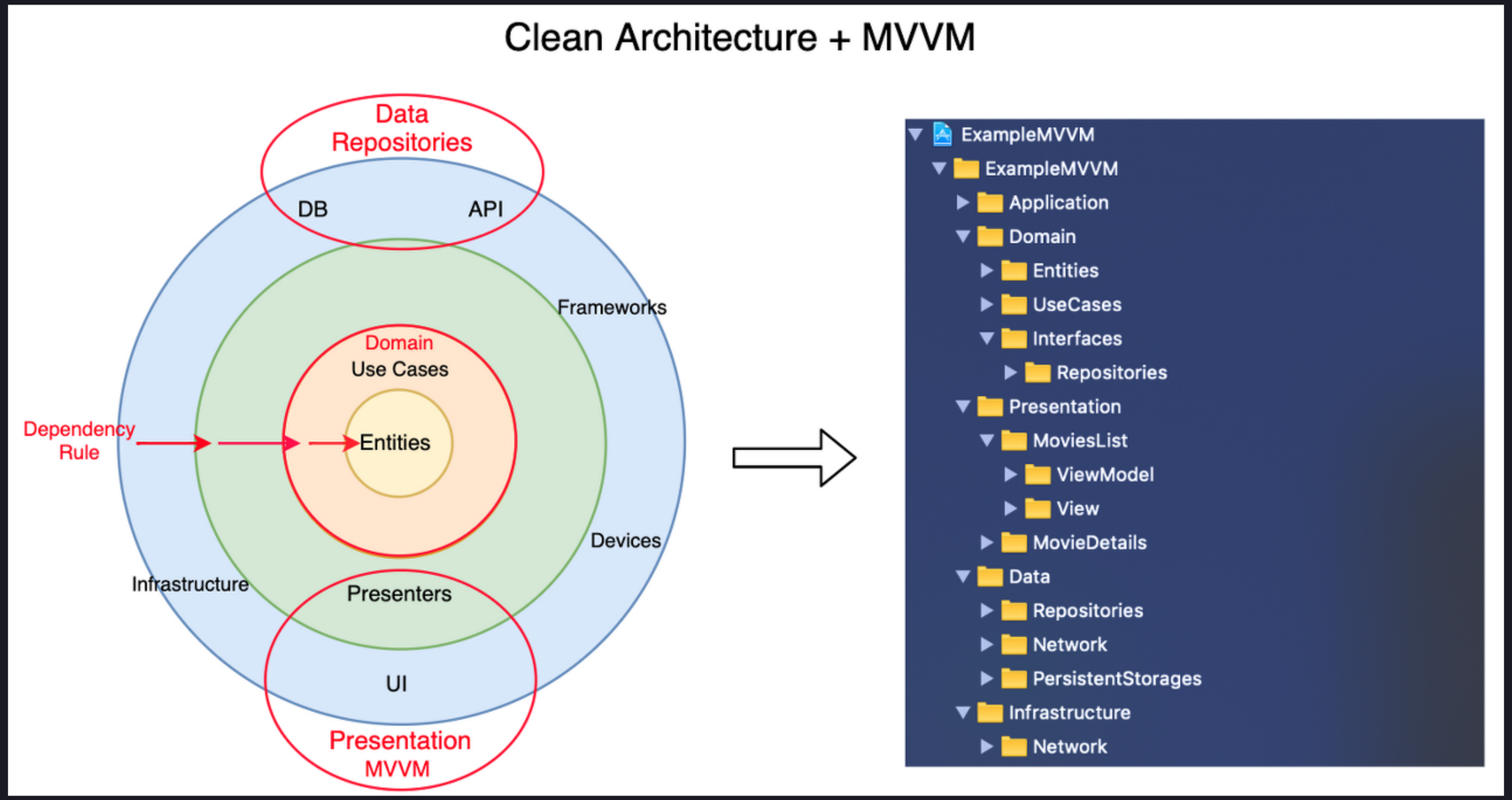
Clean Architecture

ios-chansoo “MVVM-C + Clean Architecture 도입기”



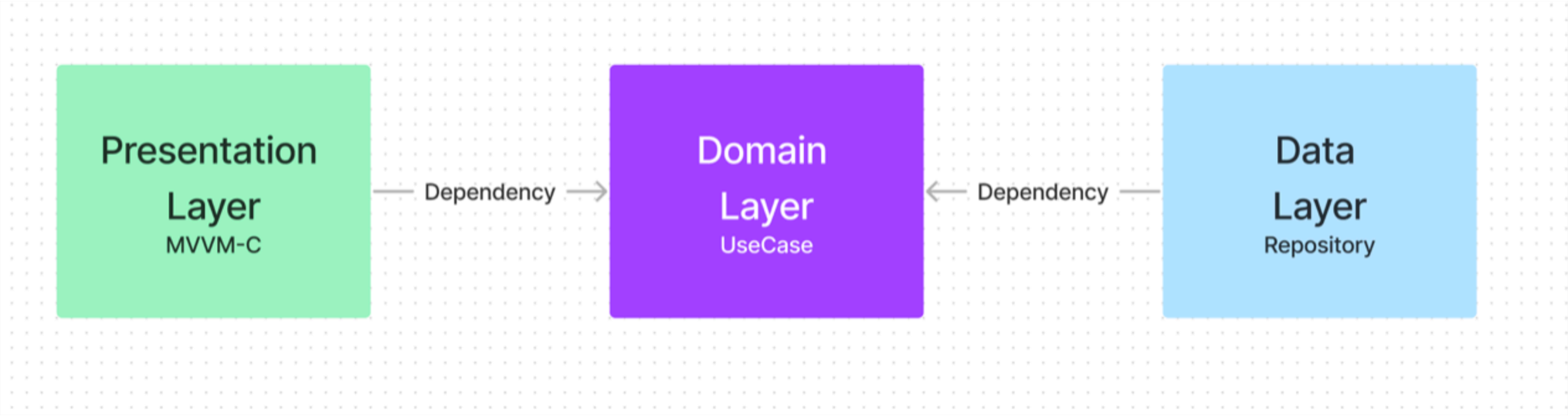
Clean Architecture

ios-chansoo “MVVM-C + Clean Architecture 도입기”



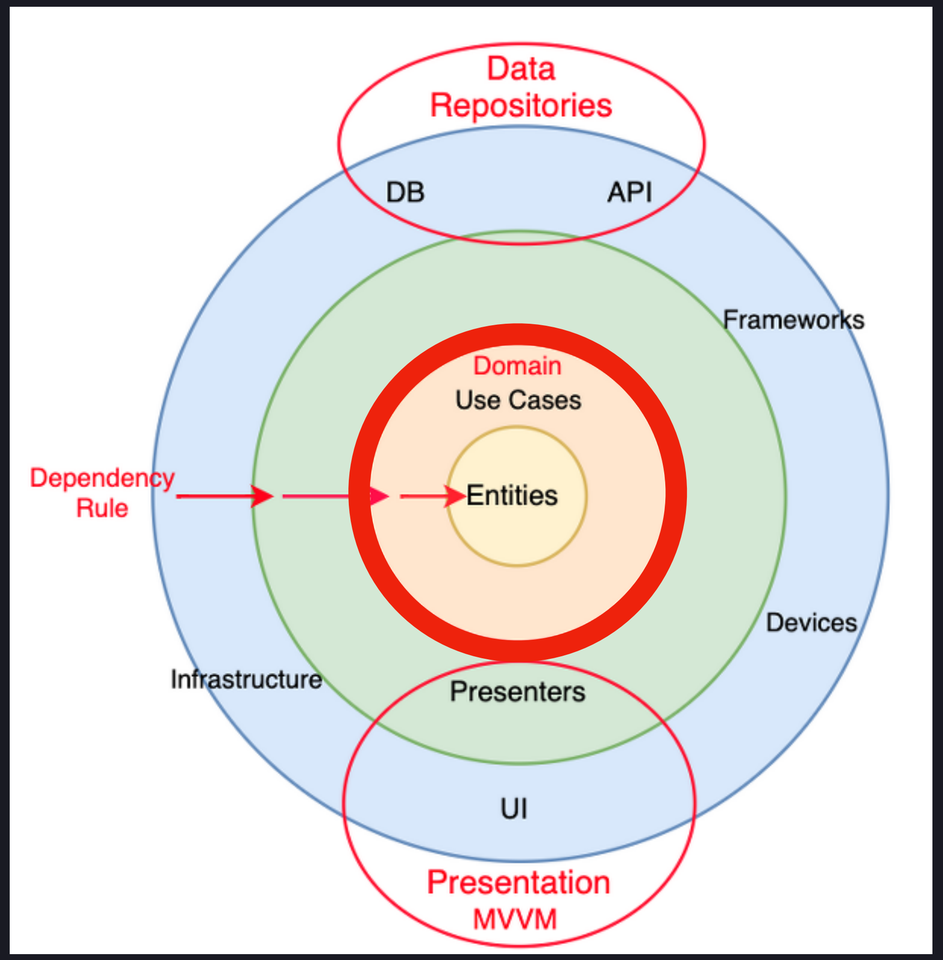
Clean Architecture

ios-chansoo “MVVM-C + Clean Architecture 도입기”



Clean Architecture

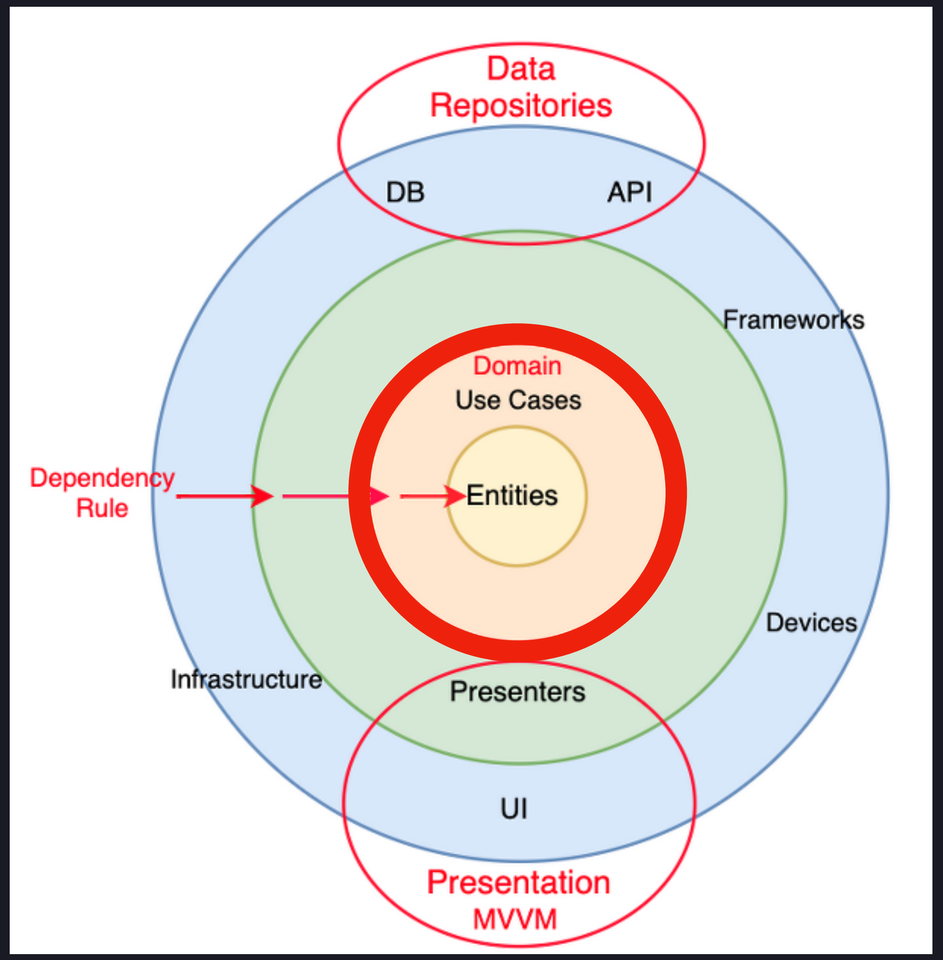
Domain Layer



- 제일 안쪽에 위치
- Entity(Model), Use Case

Clean Architecture

Domain Layer

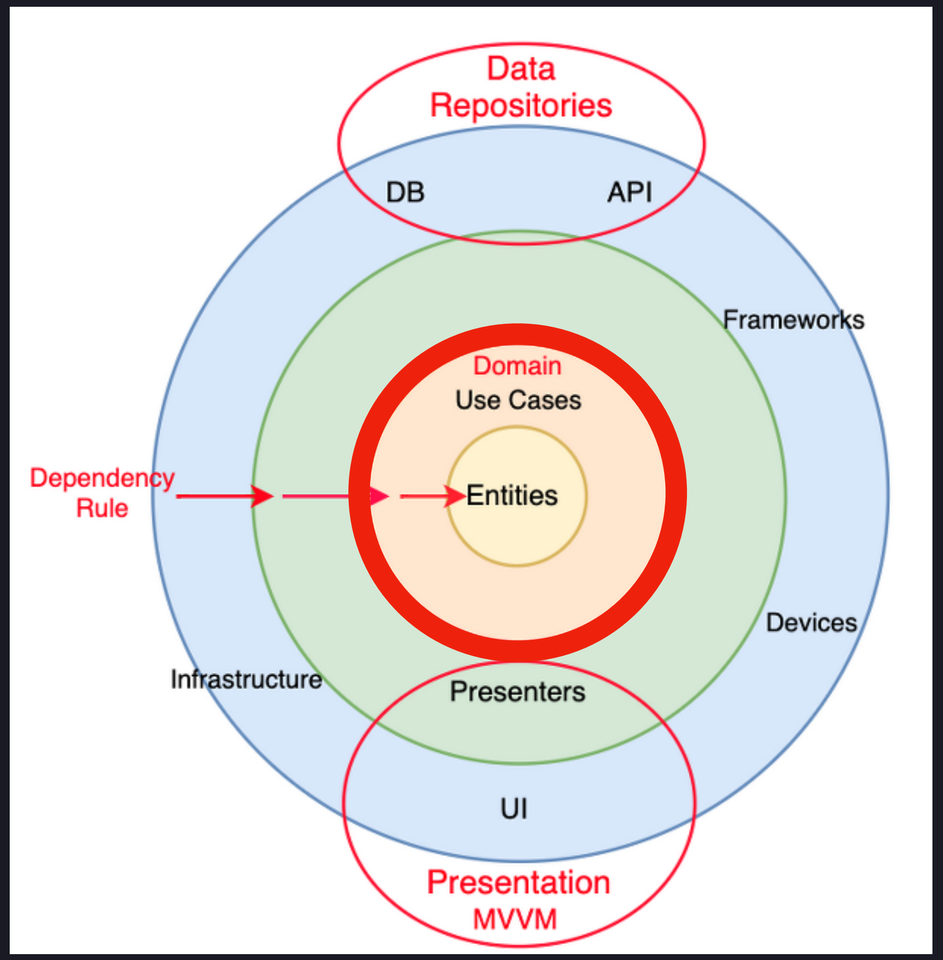


- Entity(Model)

Domain 레이어에서 사용하는 정보

Clean Architecture

Domain Layer

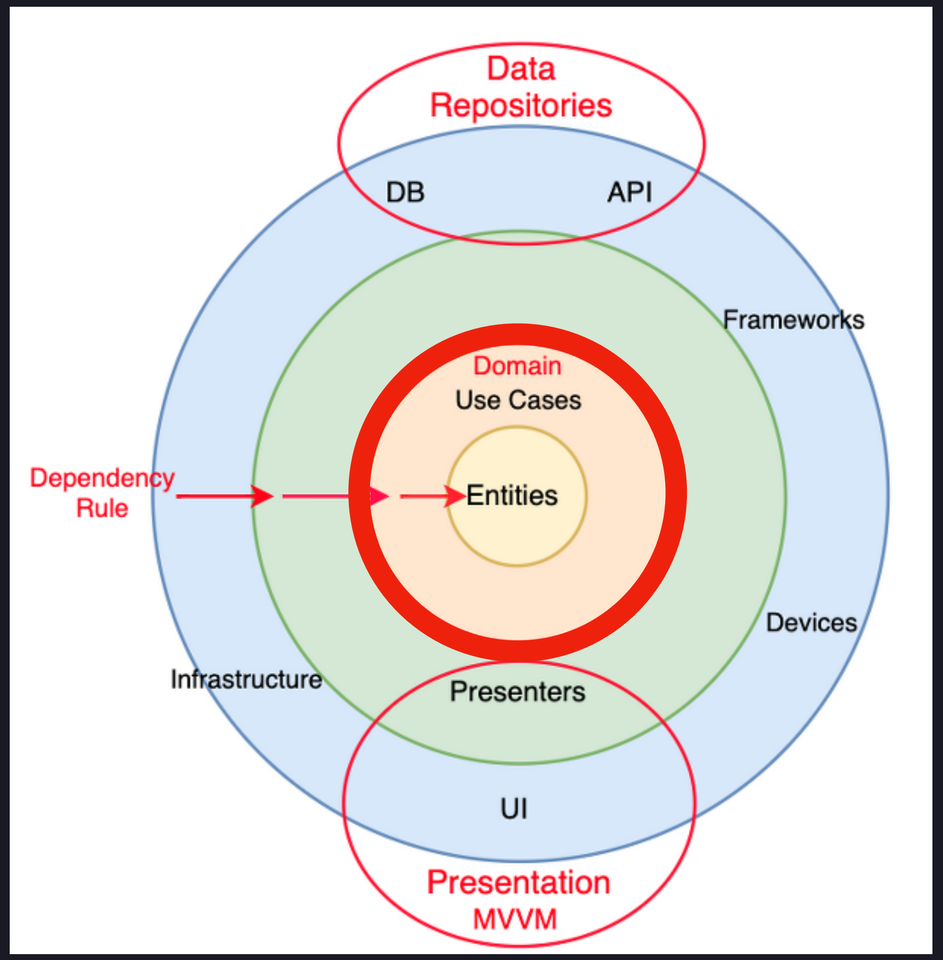


- UseCase

앱의 로직에서 하나의 행동으로 볼 수 있는 역할

Clean Architecture

Domain Layer



- UseCase

앱의 로직에서 하나의 행동으로 볼 수 있는 역할

다른 layer에 대한 의존성을 가지지 않게 하기 위해서 Protocol을 사용

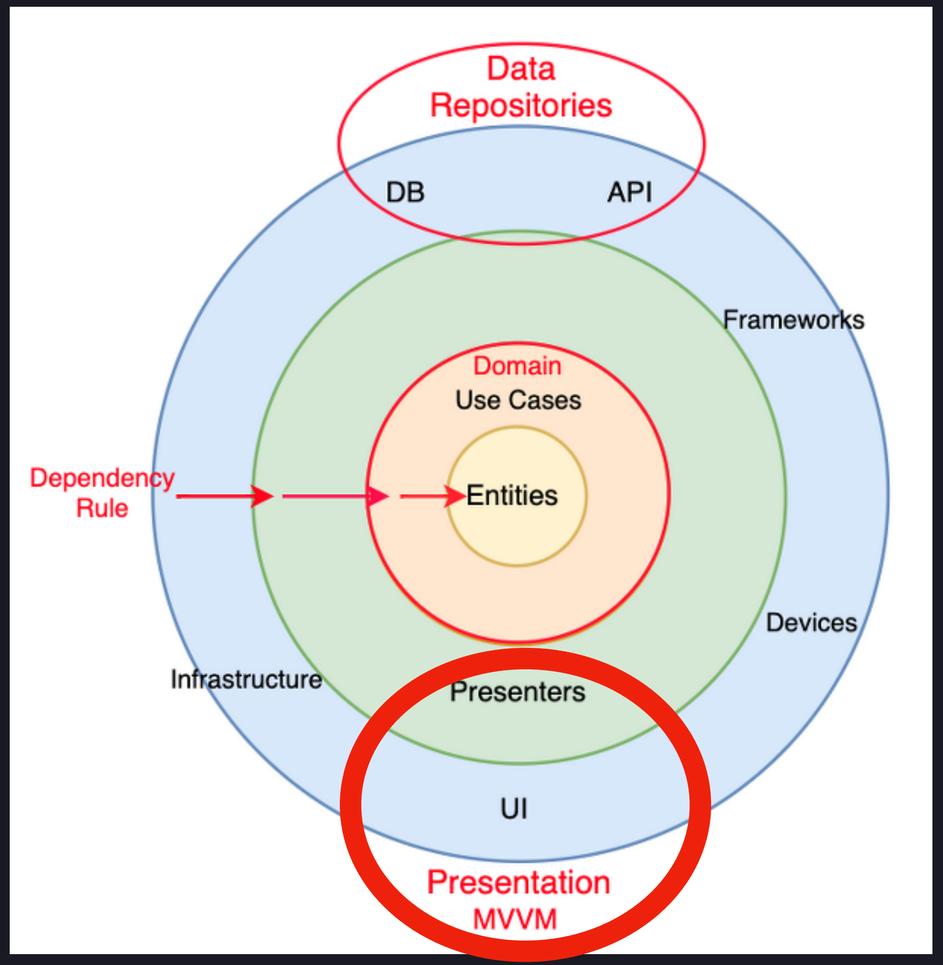
Clean Architecture

Domain Layer

```
// DefaultUseCaseA: 프로토콜인 UseCaseA를 채택해서 구현한 객체
  final class DefaultUseCaseA: UseCaseA {

    // MARK: Properties
    private let repositoryA: RepositoryA
    // MARK: Initializers
    init(repositoryA: RepositoryA) {
      self.repositoryA = repositoryA
    }
  }

  // UseCase 생성부분
  //DefaultRepositoryA: 프로토콜인 RespositoryA를 채택해서 구현한 객체
  let usecase = DefaultUseCaseA(repositoryA: DefaultRepotioryA)
```

Clean Architecture

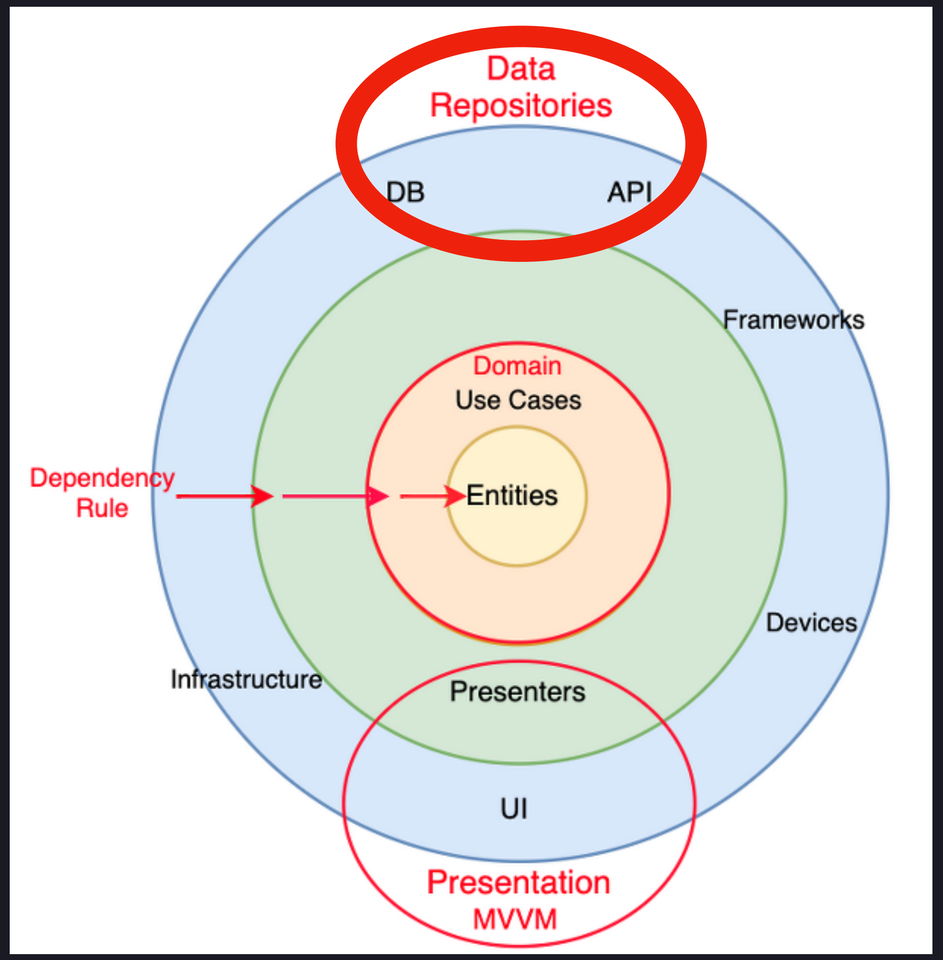
Presentation Layer



- 화면에 관련된 영역을 담당하는 Layer
- MVVM, Coordinator 또한 이 부분에 해당
- Domain layer에서 처리된 데이터들(Entity)를 알맞은 형태로 보여주는 역할 담당

Clean Architecture

Data Layer



- 데이터를 받아오는 계층
- API를 통해서 혹은 CoreData, Realm이 포함
  → DTO(외부에서 받아오는 데이터), Repository(해당 DTO를 받아옴)

Clean Architecture

Data Layer

Repository가 반환하는 Object는

DTO, Domain Layer의 Entity 중 어떤 것이 돼야할까?

Clean Architecture

Data Layer

DTO로 전달할 경우

→ UseCase가 DTO를 알게 됨 → 적절하지 않음!

Clean Architecture

Data Layer

DTO로 전달할 경우

→ UseCase가 DTO를 알게 됨 → 적절하지 않음!

DTO를 Entity로 변환하는 것까지 Repository의 역할

Clean Architecture

Data Layer

근데 **DTO**랑 **Entity**는 뭐가 다르지?

Clean Architecture

Data Layer

DTO는 Data Layer에서 사용, Entity는 Domain Layer에서 사용

Clean Architecture

Data Layer

DTO

```
struct PointHistoryItemDTO: Codable {
  let eventType: Int
  let pointType: Int
  let useType: Int
  let regDate: String
  let isDetail: String
}
```

Clean Architecture

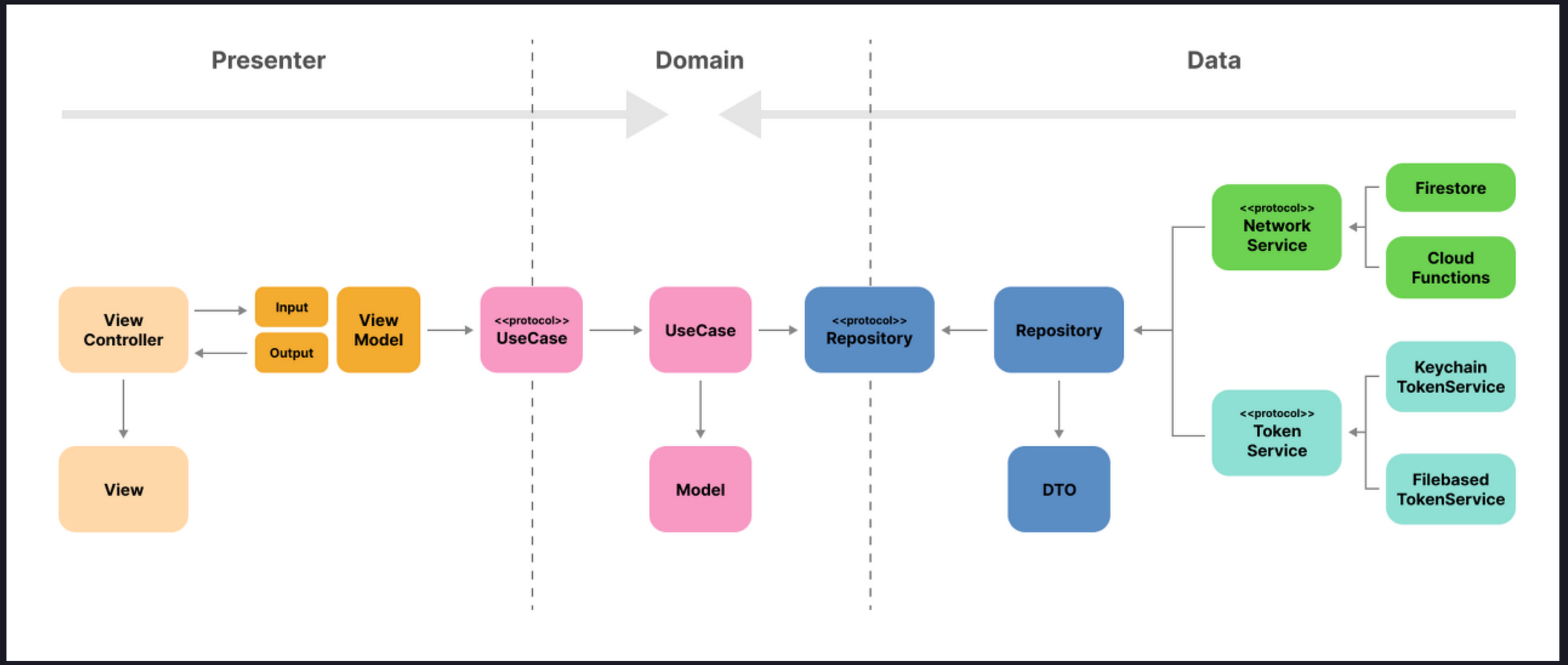
Data Layer

Entity

```
struct PointHistoryItem {
  let yearAndMonth: String
  let date: String
  let eventType: PointEventType
  let pointType: PointType
  let useType: PointUseType
  let isDetail: Bool
}
```

Clean Architecture

ios-chansoo “MVVM-C + Clean Architecture 도입기”



Clean Architecture의 이점

- 각각의 비즈니스 로직을 따로 테스트할 수 있음
- 각 객체의 역할을 프로토콜로 정의하고, Mock 객체를 구현하여 단위 테스트하기에 용이하도록 구현할 수 있음
- 프로토콜로 해당 클래스의 역할과 형태를 명시해서, 협업을 할 때 각 객체가 어떤 역할을 하는지 쉽게 파악할 수 있음
- 여러 뷰에서 동일한 로직이 사용될 때 UseCase를 재사용 할 수 있음

Clean Architecture의 이점

- 각각의 비즈니스 로직을 따로 테스트할 수 있음
- **각 객체의 역할을 프로토콜로 정의하고, Mock 객체를 구현하여 단위 테스트하기에 용이하도록 구현할 수 있음**
- 프로토콜로 해당 클래스의 역할과 형태를 명시해서, 협업을 할 때 각 객체가 어떤 역할을 하는지 쉽게 파악할 수 있음
- 여러 뷰에서 동일한 로직이 사용될 때 UseCase를 재사용 할 수 있음

Clean Architecture의 이점

- 각각의 비즈니스 로직을 따로 테스트할 수 있음
- 각 객체의 역할을 프로토콜로 정의하고, Mock 객체를 구현하여 단위 테스트하기에 용이하도록 구현할 수 있음
- **프로토콜로 해당 클래스의 역할과 형태를 명시해서, 협업을 할 때 각 객체가 어떤 역할을 하는지 쉽게 파악할 수 있음**
- 여러 뷰에서 동일한 로직이 사용될 때 UseCase를 재사용 할 수 있음

Clean Architecture의 이점

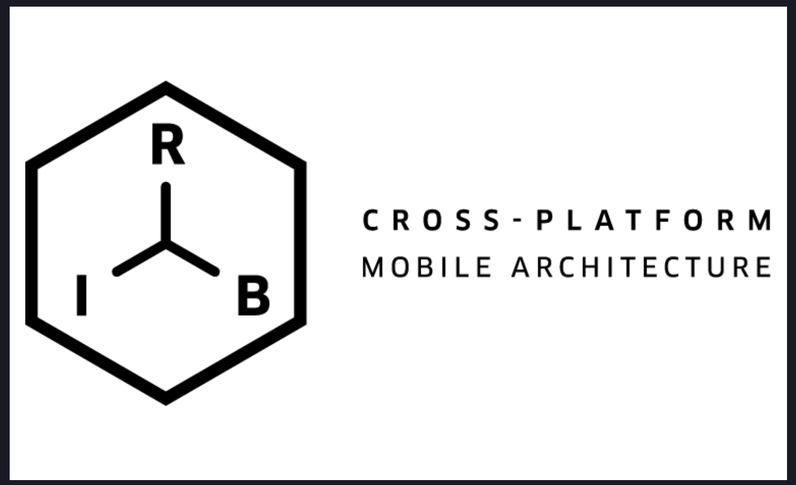
- 각각의 비즈니스 로직을 따로 테스트할 수 있음
- 각 객체의 역할을 프로토콜로 정의하고, Mock 객체를 구현하여 단위 테스트하기에 용이하도록 구현할 수 있음
- 프로토콜로 해당 클래스의 역할과 형태를 명시해서, 협업을 할 때 각 객체가 어떤 역할을 하는지 쉽게 파악할 수 있음
- 여러 뷰에서 동일한 로직이 사용될 때 UseCase를 재사용 할 수 있음

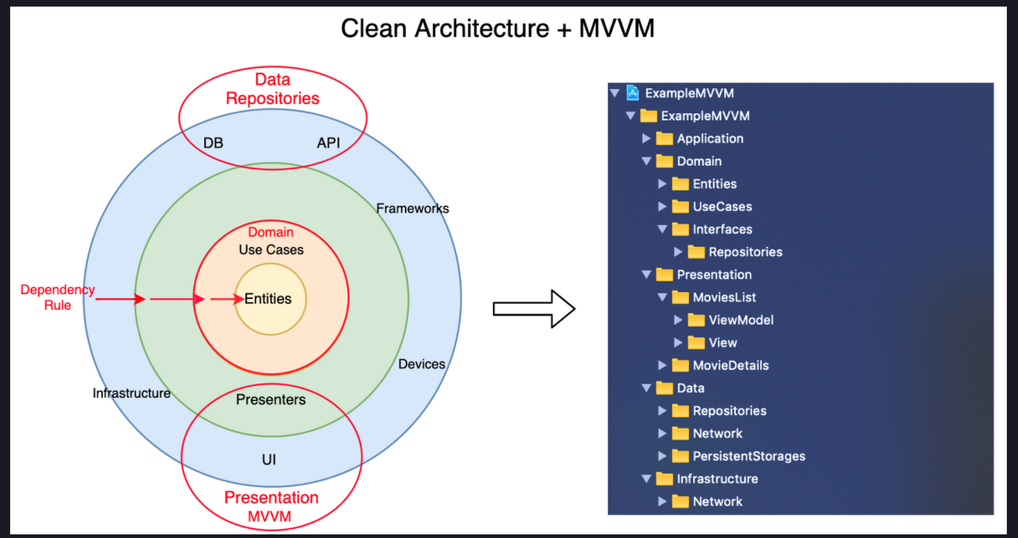
Clean Architecture의 이점

layer를 나누고, 단일 방향으로 흘러가며, 관심사를 분리하는 것

합쳐보았습니다 

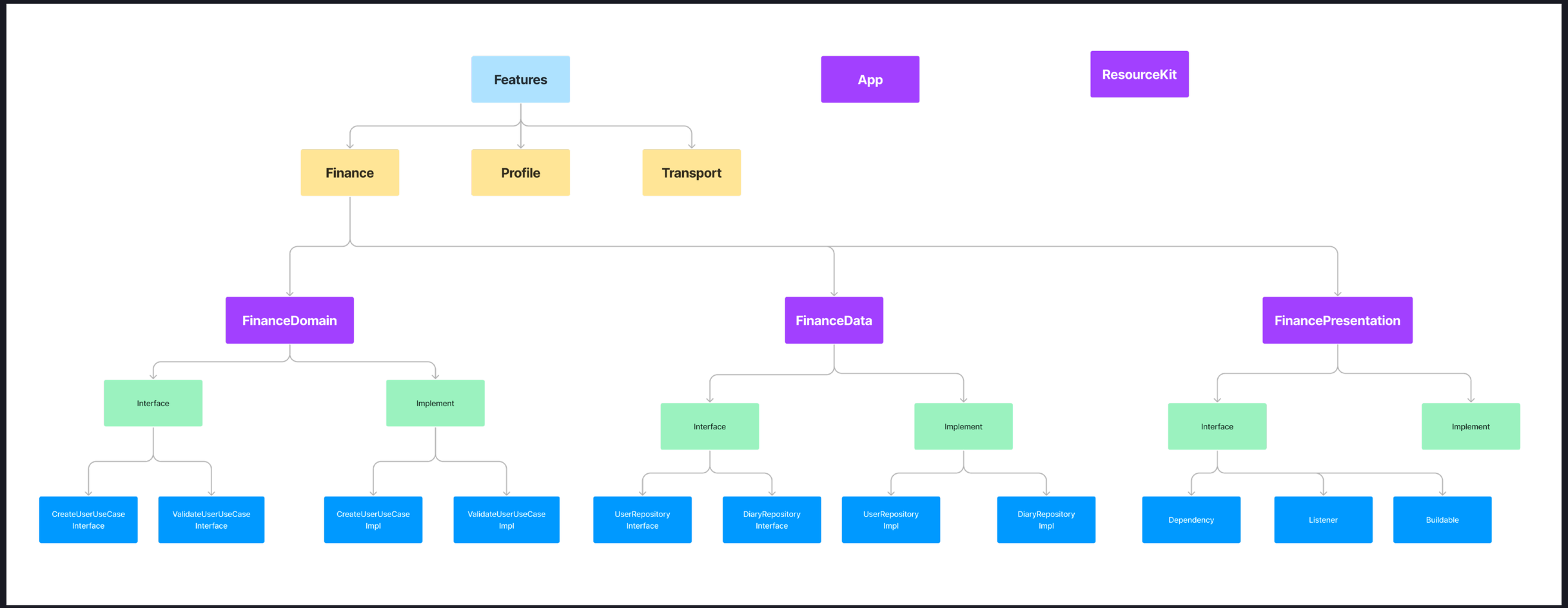
그 전에..,.


R
I
B
CROSS-PLATFORM
MOBILE ARCHITECTURE


Clean Architecture + MVVM
Data
Repositories
DB
API
Frameworks
Domain
Use Cases
Entities
Dependency
Rule
Infrastructure
Presenters
Devices
UI
Presentation
MVVM
ExampleMVVM
ExampleMVVM
Application
Domain
Entities
UseCases
Interfaces
Repositories
Presentation
MoviesList
ViewModel
View
MovieDetails
Data
Repositories
Network
PersistentStorages
Infrastructure
Network

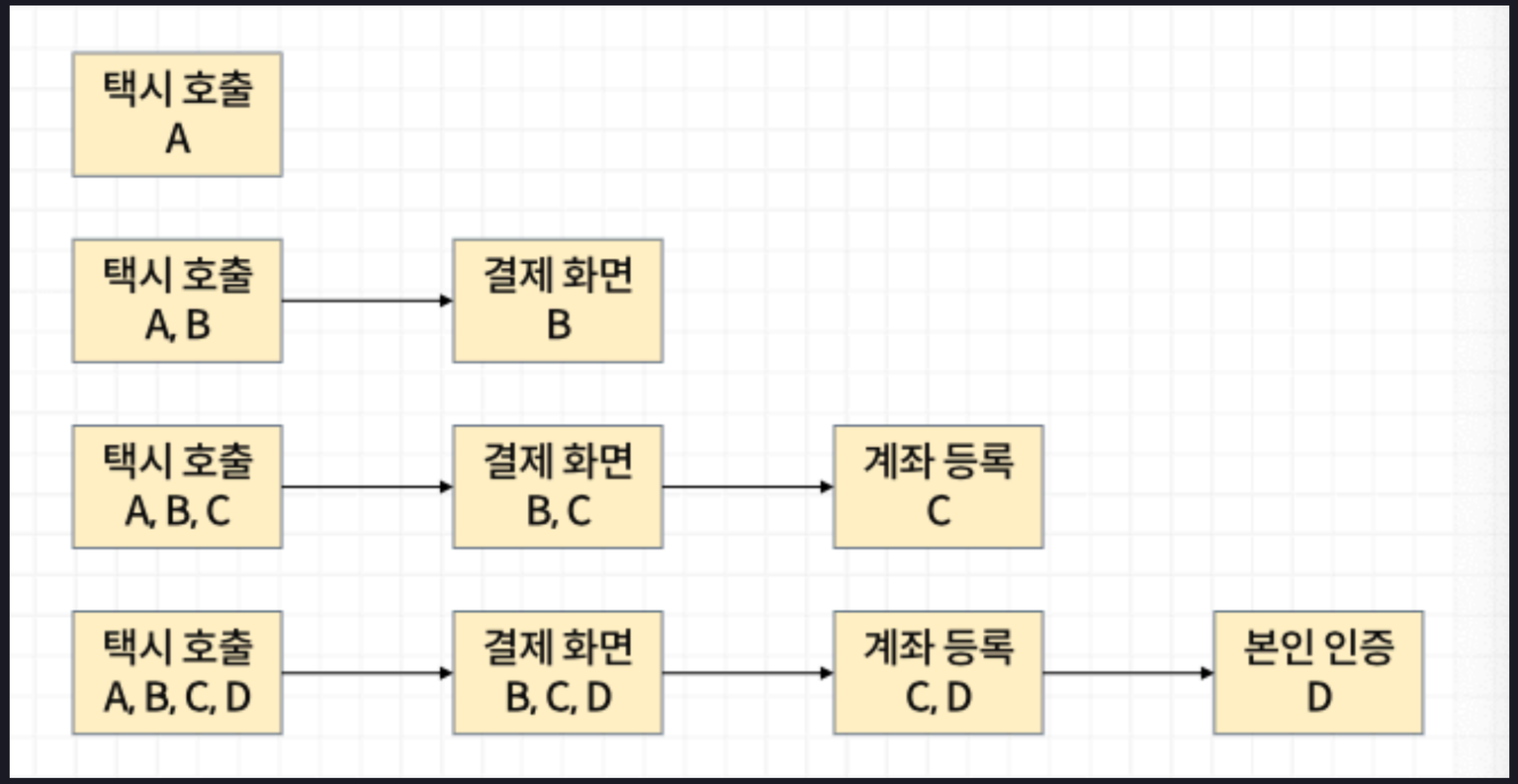
이 둘을 합쳐보았습니다 🌪️

이것이 샘플앱의 뼈대입니다



저희의 고민거리

고민거리



저희의 고민거리(였던 것)

(a.k.a. 트라블슈팅)

저희의 고민거리였던 것

고민거리였던 것

대충 Framework

Static Library