# 의존성 주입 (Dependency Injection)

# 의존성 주입

## Dependency Injection

- **의존성:** 서로 다른 객체 사이에 의존 관계가 있다는 것
  **주입**: 외부에서 객체(또는 데이터)를 생성해서 넣는 것 (생성자를 통해)

- **의존성 주입**
  프로그램 디자인이 결합도를 느슨하게 되도록하고 **의존관계 역전 원칙**과 **단일 책임 원칙**을 따르도록 클라이언트 생성에 대한 의존성을 클라이언트의 행위로 부터 분리하는 것

- 기존의 의존성을 개선하여 **"(개선된) 의존성을 외부에서 주입할 수 있는 방식"**으로 바꾸는 것

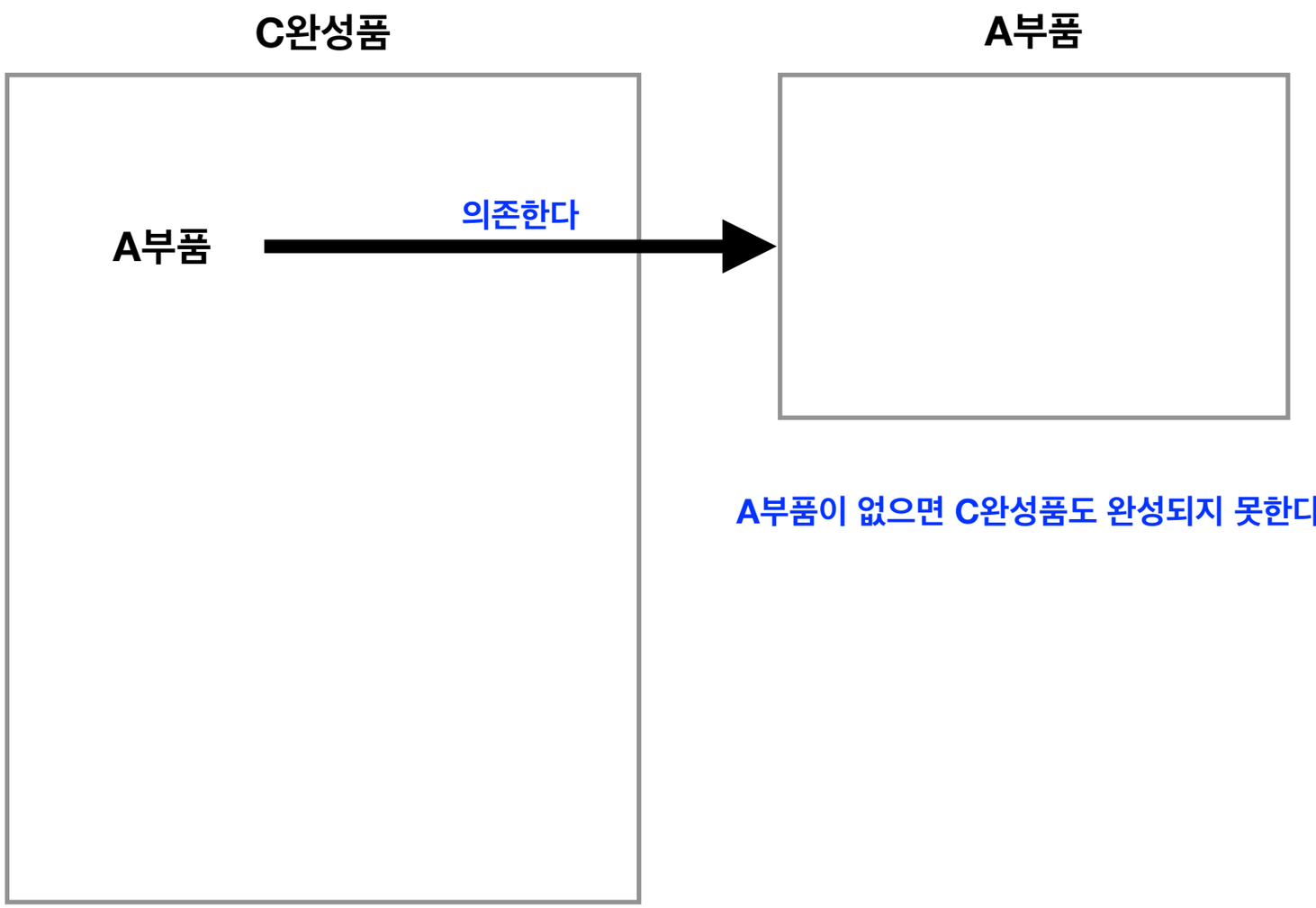
- **(개선된) 의존성:** 프로토콜을 사용해서 의존성을 분리시키고
  의존관계를 역전(Inversion Of Control) 시킴

- **주입:** 생성자를 통해서 외부에서 값을 주입한다.
  (생성시 값 할당 가능 / 언제든지 교체 가능해져 확장성이 늘어남)

# 의존성 주입

## Dependency Injection

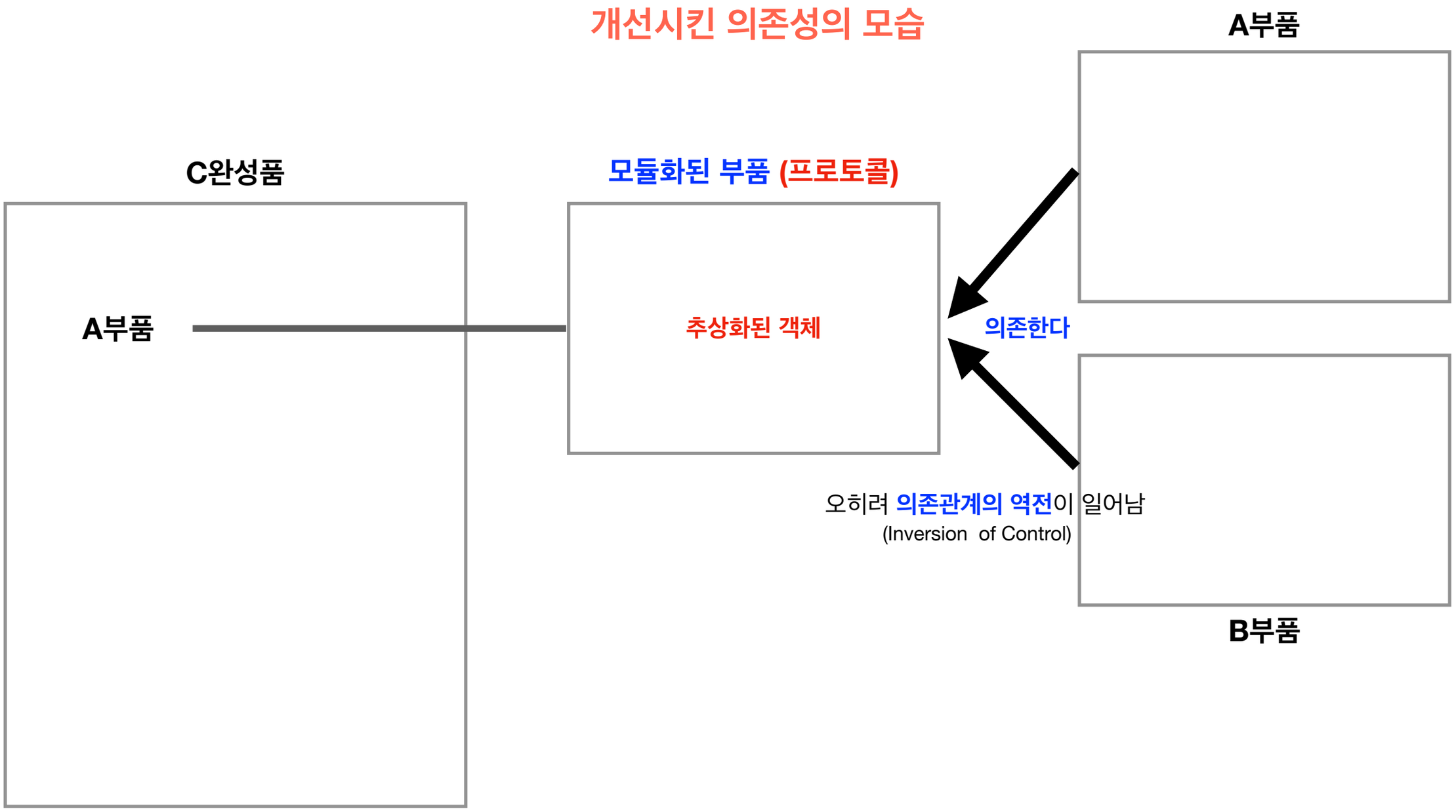
개선되기 전의 의존성의 모습




**A부품이 없으면 C완성품도 완성되지 못한다.**

# 의존성 주입

## Dependency Injection



# 의존성 주입

## Dependency Injection

- **의존성 주입**
  기존의 의존성을 개선하여 **"(개선된) 의존성을 외부에서 주입할 수 있는 방식"**으로 바꾸는 것

- **의존성 주입의 장점 / 사용하는 이유**
  - 객체 간의 의존성을 줄여서 코드의 재활용성 / 확장성이 높아짐
  - 객체 간의 결합도가 낮아져 유연한 코드/유연한 프로그램을 작성 가능
  - 유지 보수 쉬워짐
  - Unit Test가능해짐 (특정 객체에 대한 의존성 없애고 Test객체 주입가능)

- **객체지향 프로그래밍(OOP)의 5대 원칙(SOLID) 중**
  하나가 의존 관계 역전 원칙(DIP: Dependency Inversion Principle) (의존 관계의 분리)

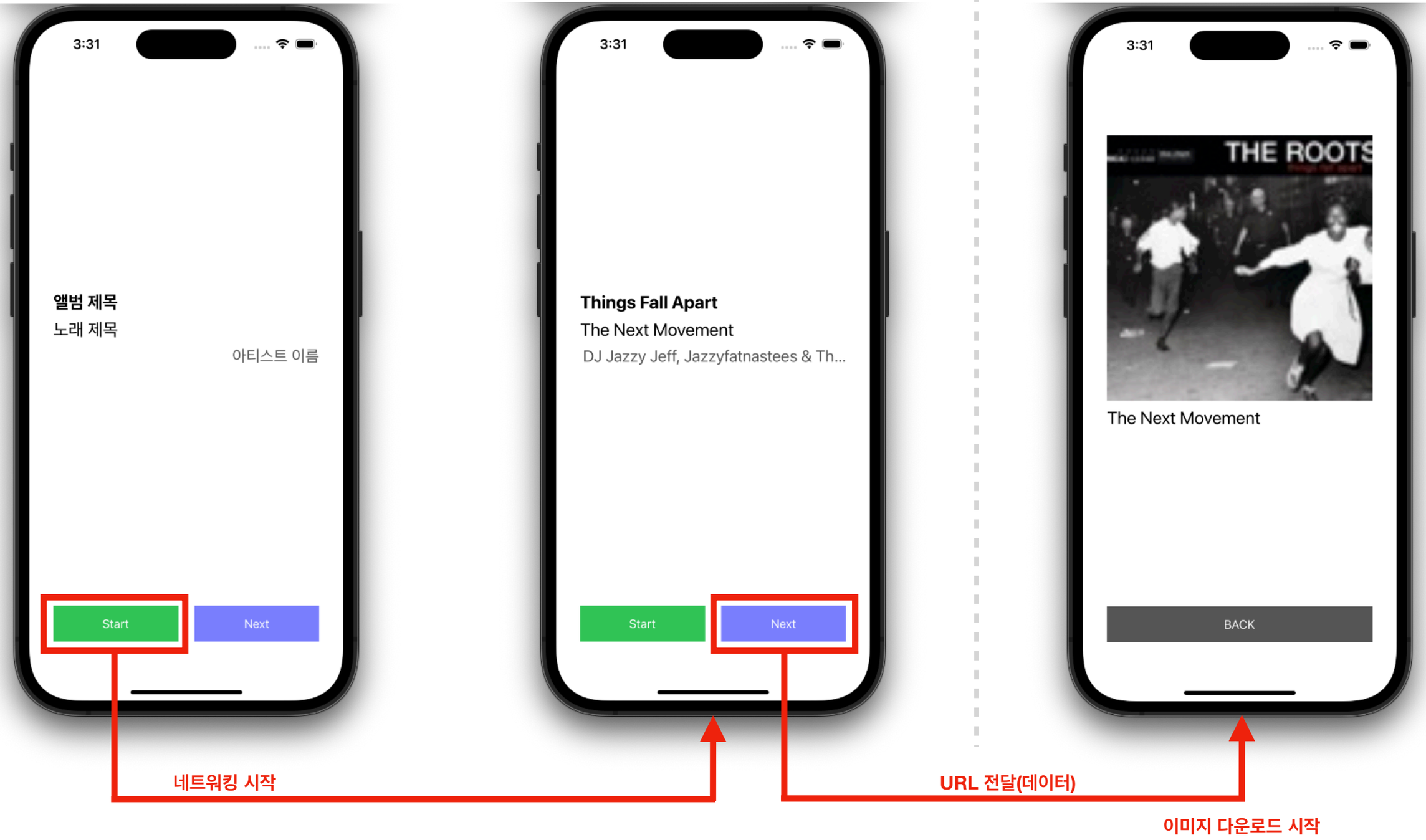
  - 추상화된 것은 구체적인 것에 의존하면 안되고, 구체적인 것이 추상화된 것에 의존해야 한다.
    즉, 구체적인 객체는 **추상화된 객체(프로토콜)**에 의존해야 한다.

# 예제 (간단한 앱으로 이해하기)

## MVC 아키텍처

# 아키텍처 패턴

## MVC패턴



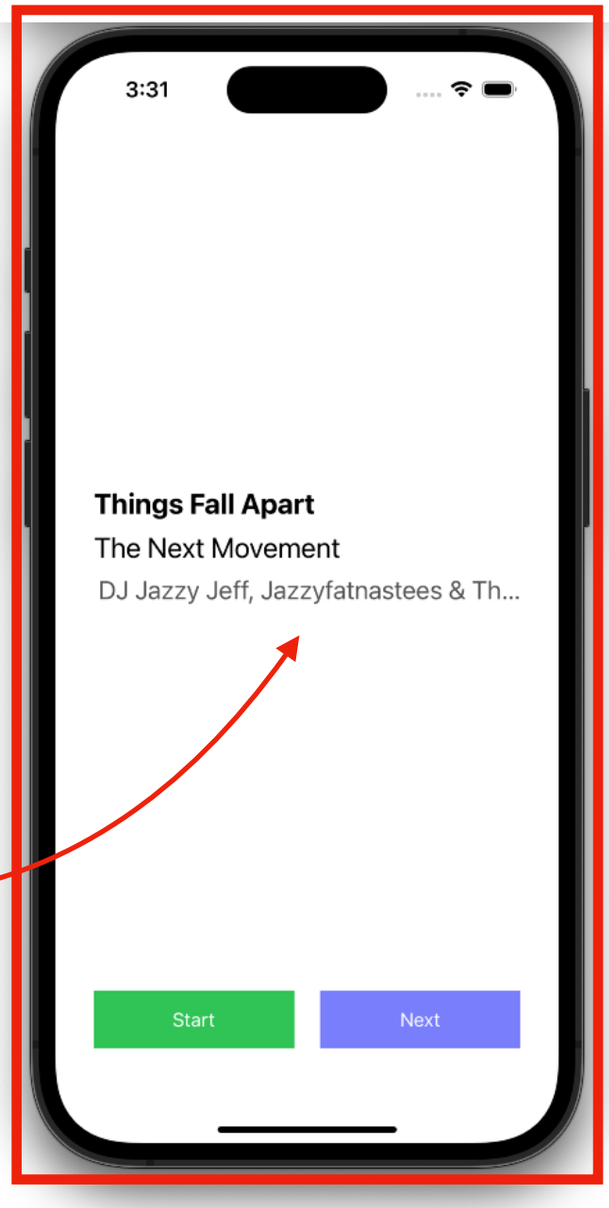
# 아키텍처 패턴

## MVC패턴

모델 (데이터)

```
struct Music: Codable {
    let songName: String?
    let artistName: String?
    let albumName: String?
    let imageUrl: String?
}
```

화면 (뷰)



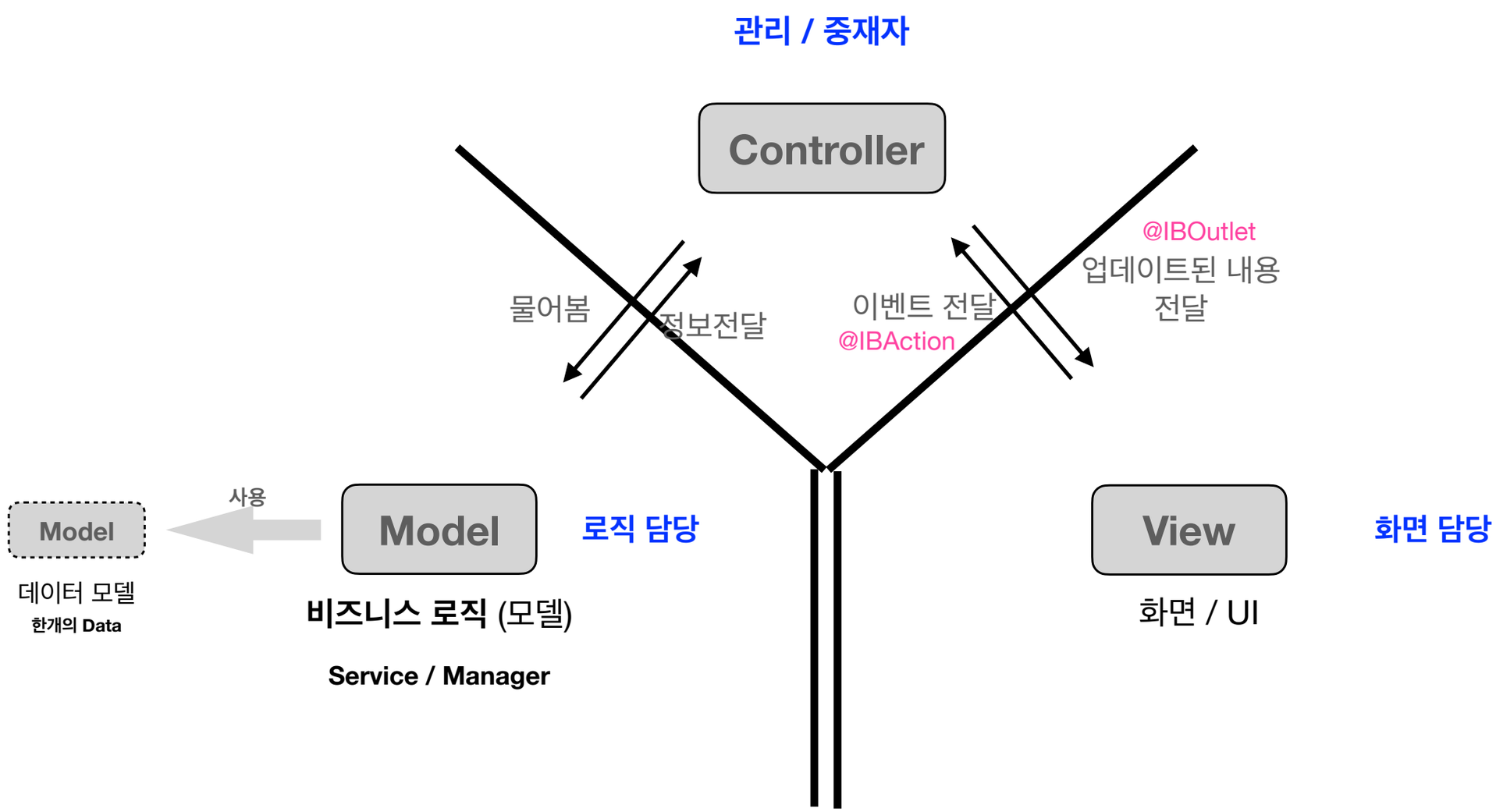

**모델이 가지고 있는 데이터를 가져다가 화면(뷰)에 표시**

모든 앱에서 일어나는 일.
결국 어떤 앱이든, 앱에 표시하게
되는 데이터는 모델에서 가져다가..

# MVC

## Model - View - Controller

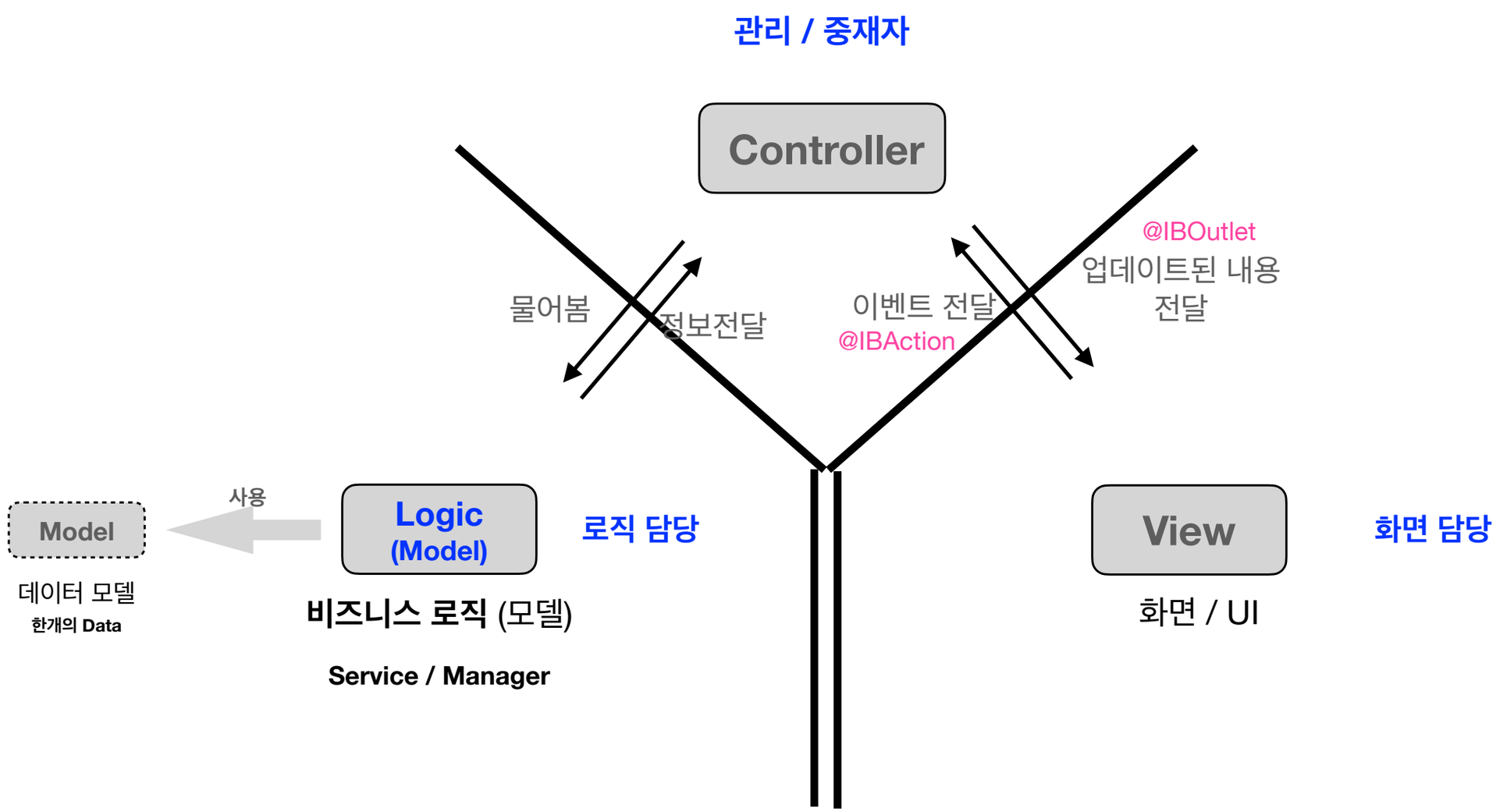
(단순하게) 각자의 역할과 책임을 나누는 것이 목적



# MVC

## Model - View - Controller

(단순하게) 각자의 역할과 책임을 나누는 것이 목적



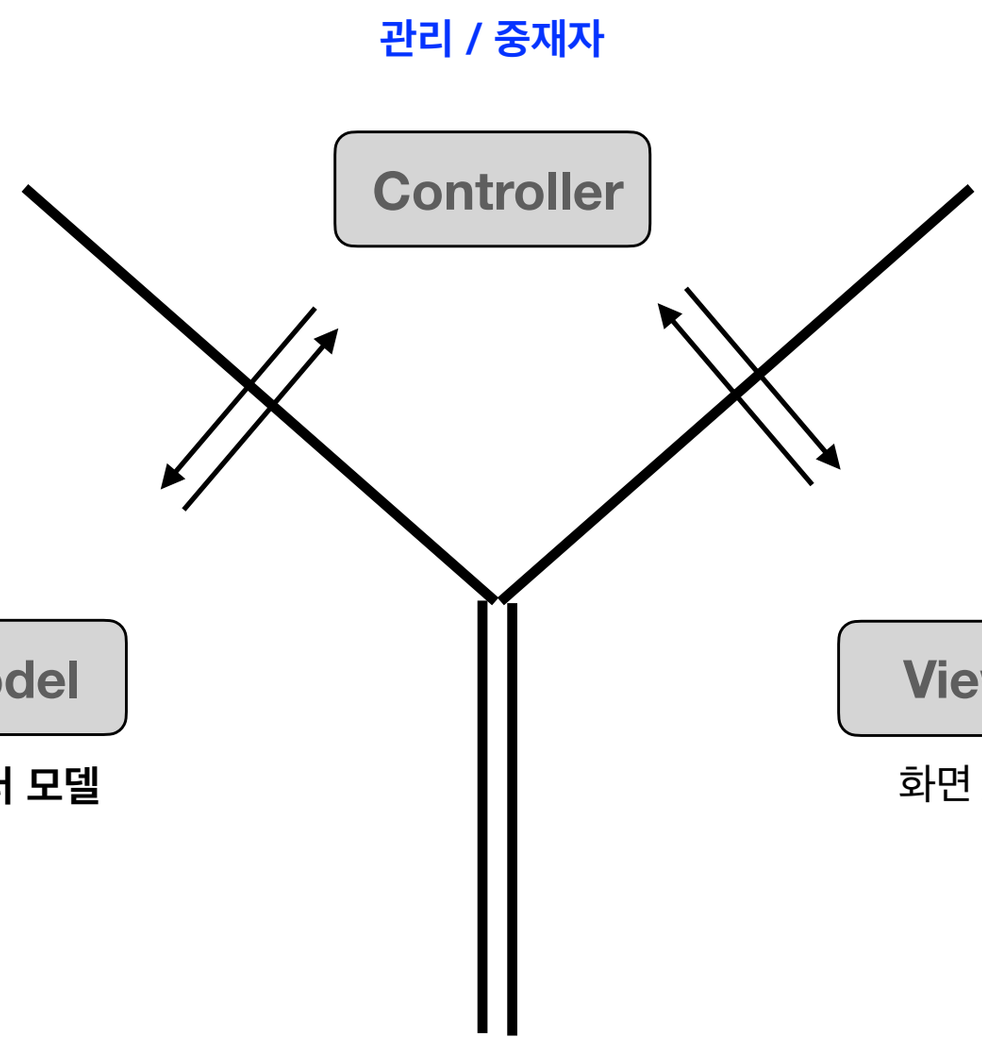
# MVC

## Model - View - Controller

(단순하게) 각자의 역할과 책임을 나누는 것이 목적

MVC에서 말하는
이론적인

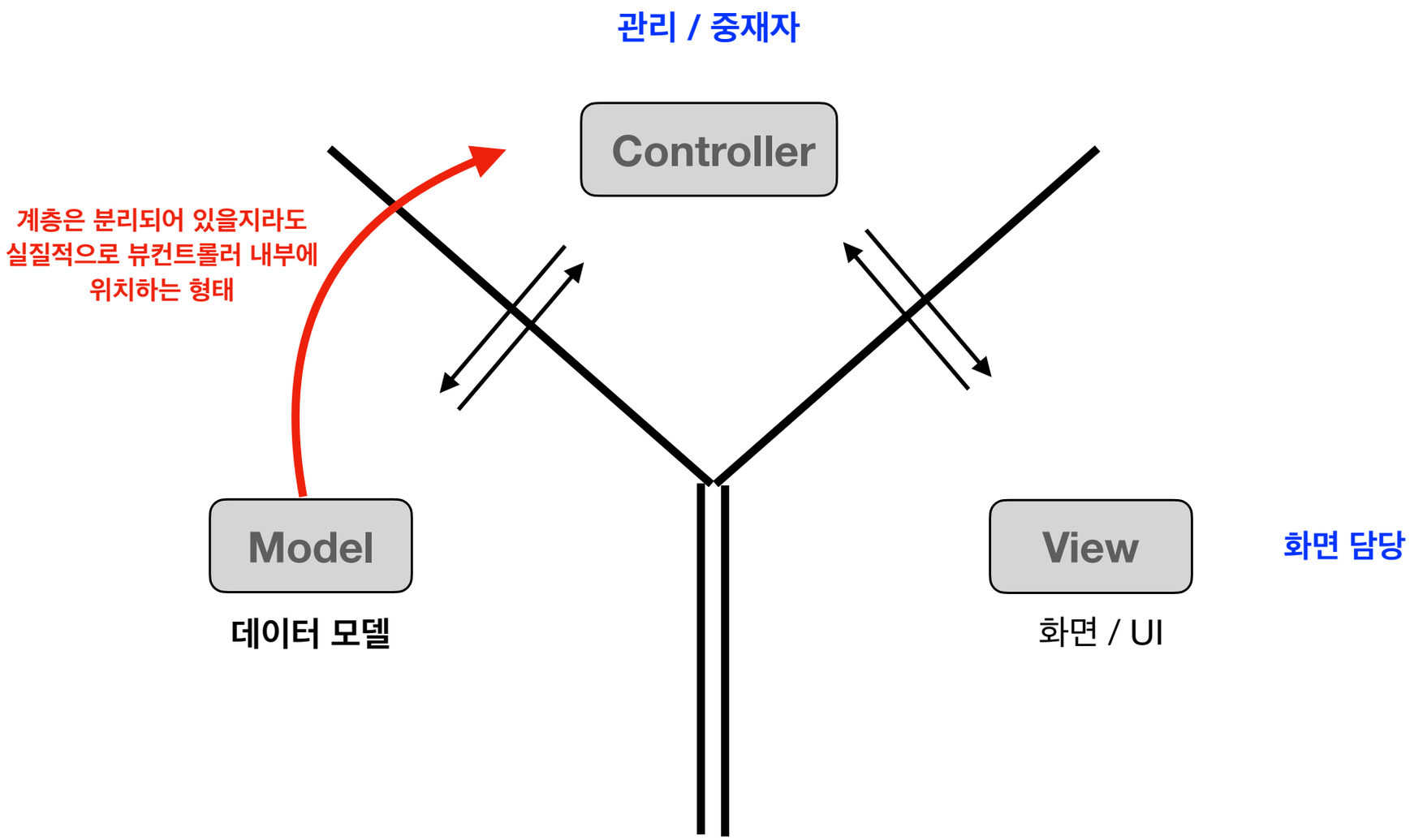
**Model - 데이터 모델**



# MVC

## Model - View - Controller

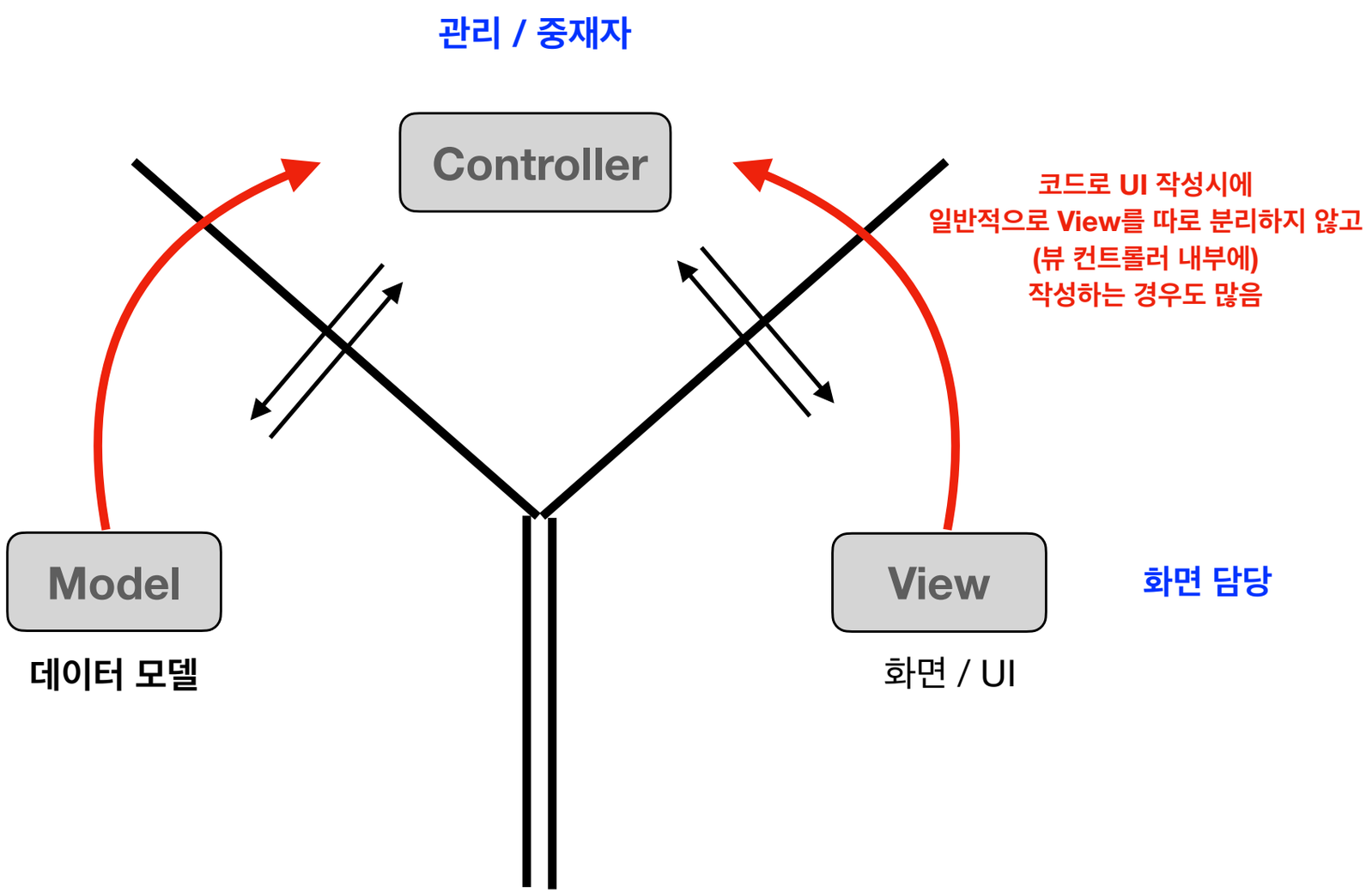
(단순하게) 각자의 역할과 책임을 나누는 것이 목적



# MVC

## Model - View - Controller

(단순하게) 각자의 역할과 책임을 나누는 것이 목적



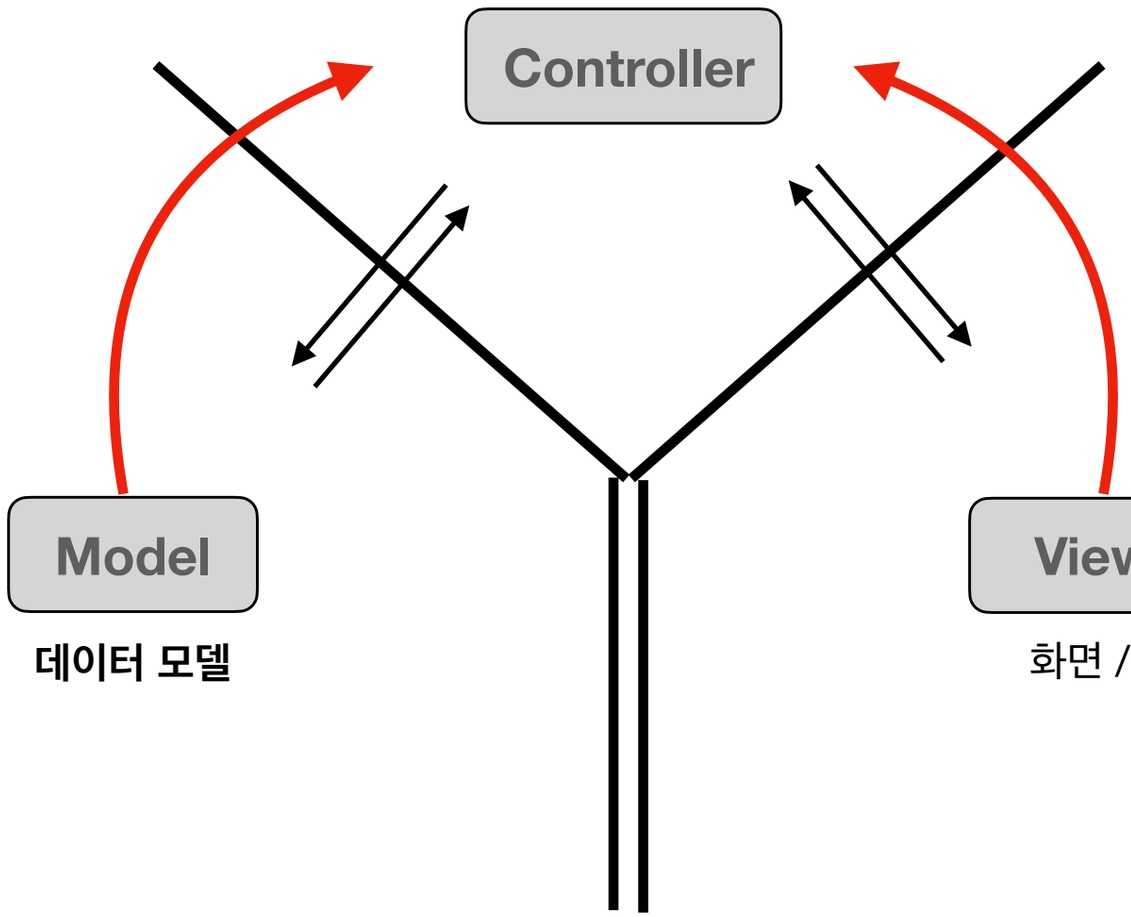
# MVC

## Model - View - Controller

(단순하게) 각자의 역할과 책임을 나누는 것이 목적

비대해짐 / 너무 역할이 커짐 / 테스트 코드 작성 불가능

관리 / 중재자




화면 담당

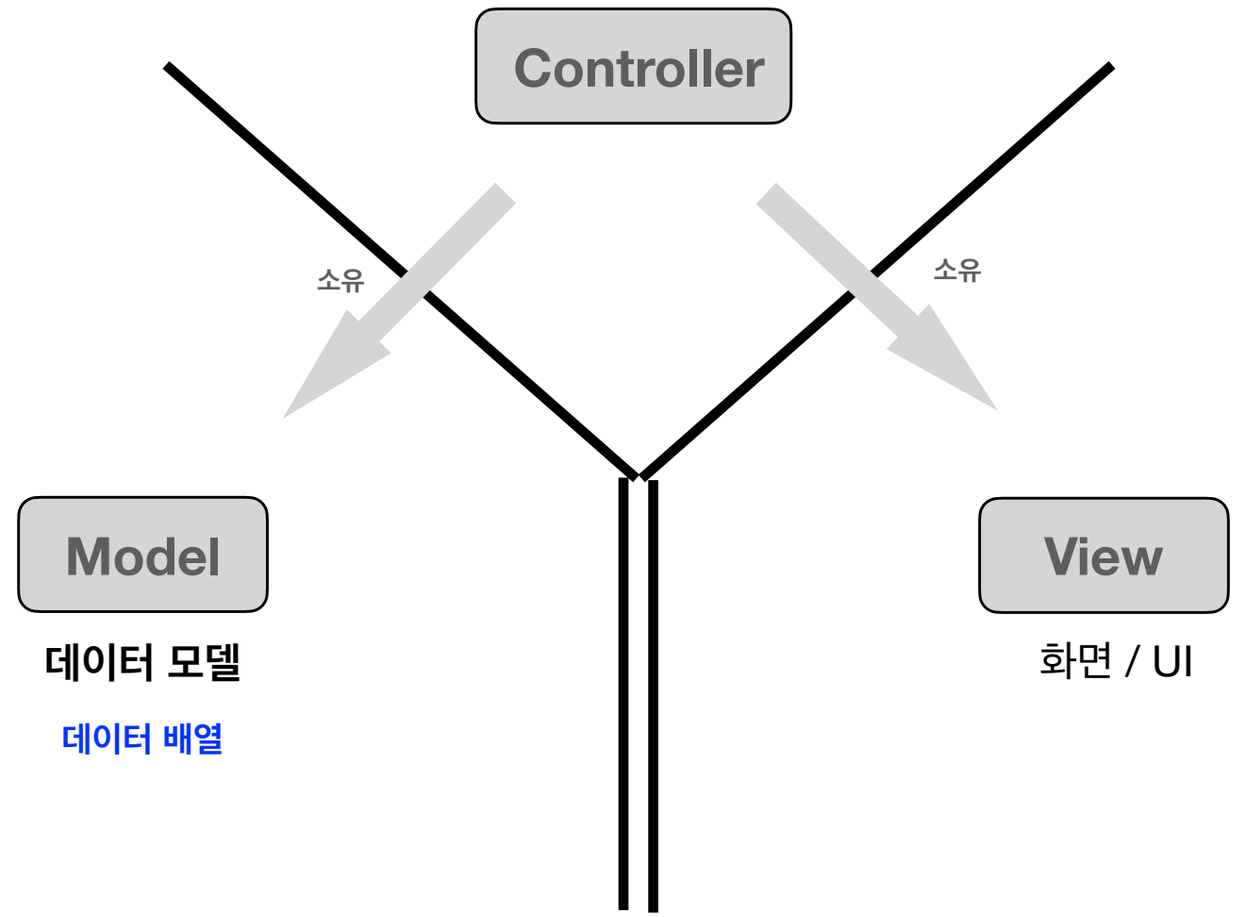
# MVC

## Model - View - Controller

(단순하게) 각자의 역할과 책임을 나누는 것이 목적

비대해짐 / 너무 역할이 커짐 / 테스트 코드 작성 불가능

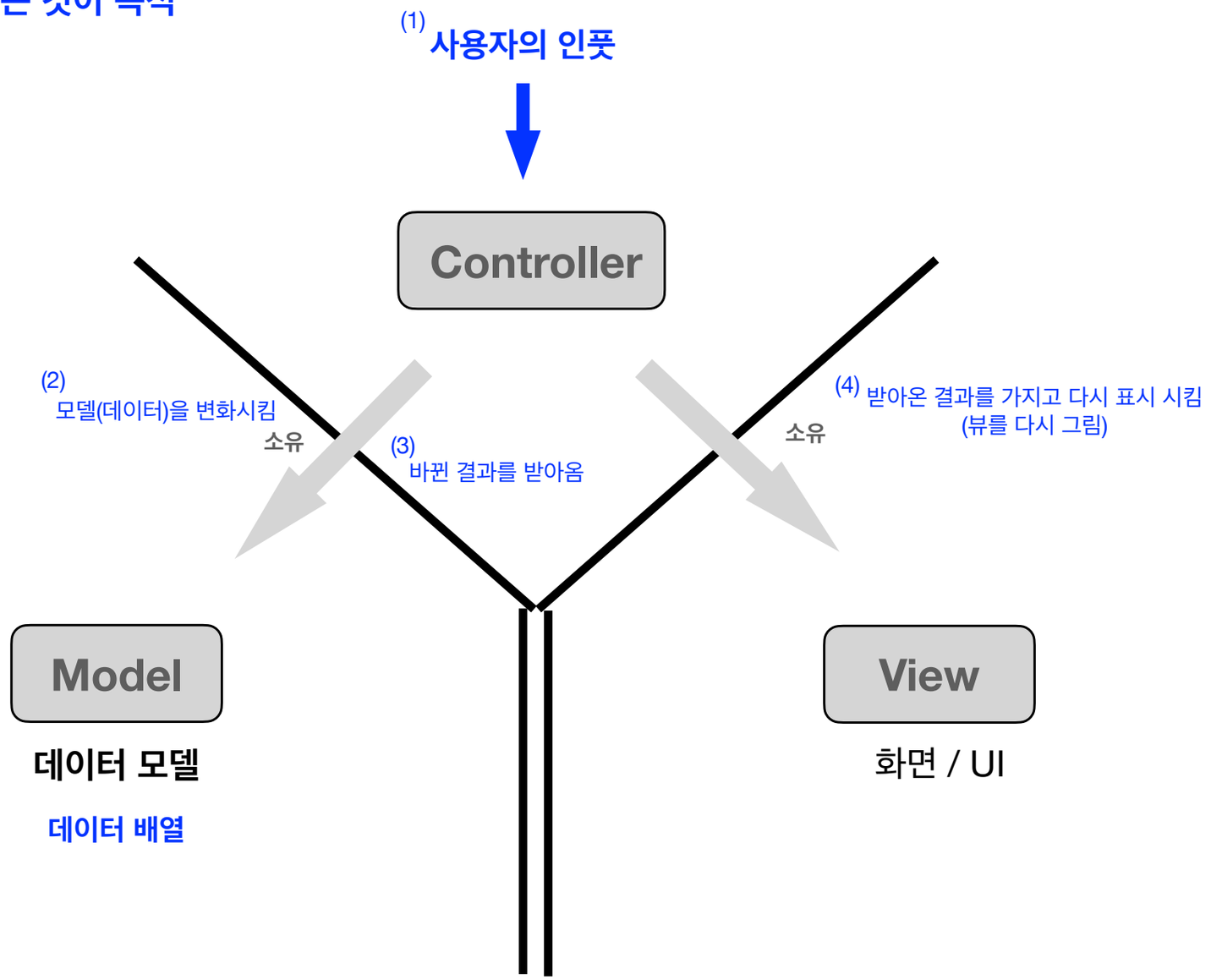
관리 / 중재자



# MVC

## Model - View - Controller

(단순하게) 각자의 역할과 책임을 나누는 것이 목적



# 왜 아키텍처 패턴이 필요할까?

왜 코드를 나누려고 하는가

# 아키텍처 패턴

**왜 코드를 나누어서 설계하려고 하는가?**

- (실무환경) **여러명이서 하나의 거대한 앱을 만들게 됨 (수만줄의 코드)**
  (누가봐도 명확한.. 약속된 기준과 엄격한 룰에 따라) 코드를 분리해서, **유지보수 편해야 함**

- **좋은 아키텍처의 특징들**
  단단하고/안정적, 모듈화 가능(재사용), 확장 용이(향후 확장성)
  테스트 가능(UI와 비즈니스 로직에서 강력/안정적), Decoupling (느슨한 결합) 등..

- **올바른 아키텍처 선택의 기준**
  - 유지보수 용이해야 한다.
  - 명확한 분리 기준: 단일 책임 원칙
  - 테스트 가능: TDD(테스트 주도 개발) / 유닛테스트 가능해야 (더 안전한 코드 작성 가능)

- **완벽한 / 절대적인 아키텍처는 없다**
  앱의 규모와 앱을 운영하는 팀 환경에 맞는 선택을 해야.. (정답이 없다)

# 아키텍처 패턴

**왜 코드를 나누어서 설계하려고 하는가?**

- **MVC (Model - View - Controller)**
  - 뷰컨 - 이벤트도 일어나고, 비즈니스로직도 존재하고..(VC 역할 비대)
  - 거대한 뷰 컨트롤러의 덩어리 / 로직과 뷰가 엉켜서 테스트 불가능 (UI + 로직)
  - 뷰컨(및 뷰)라이프 사이클과 데이터(모델)이 밀접한 연관성(밀접한 상호 작용)을 가져 각자 분리시켜 테스트가 안됨
  - 개발이 용이한 장점 (이해 쉬움) / 개발 속도가 빠를 수 있음

- **MVVM (Model - View - ViewModel)**
  - 뷰모델(ViewModel)의 도입 / 뷰모델이 로직 소유
  - 복잡성 감소 - VC비대해짐을 해소 가능 (VC가 간단해짐) / 비즈니스 로직이 더 잘 표현됨
  - 비즈니스 로직과 뷰의 분리가 가능 (테스트 코드 작성 가능)
  - 바인딩까지 활용한다면
    데이터가 변했을때 뷰가 자동으로 갱신되는 것까지도 쉽게 구현 가능 (직관적)

# 예제 (간단한 앱으로 이해하기)

**MVVM 아키텍처**

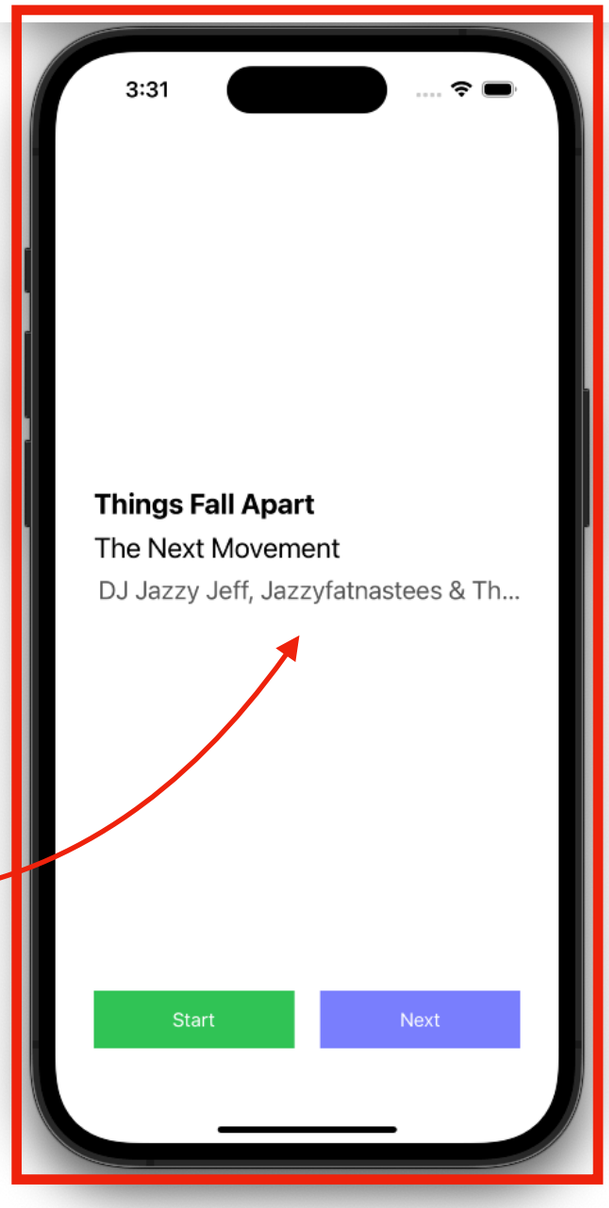
# 아키텍처 패턴

## MVC패턴

모델 (데이터)

```
struct Music: Codable {
    let songName: String?
    let artistName: String?
    let albumName: String?
    let imageUrl: String?
}
```

화면 (뷰)




**모델이 가지고 있는 데이터를 가져다가 화면(뷰)에 표시**

모든 앱에서 일어나는 일.
결국 어떤 앱이든, 앱에 표시하게
되는 데이터는 모델에서 가져다가..

# 아키텍처 패턴

## MVVM패턴

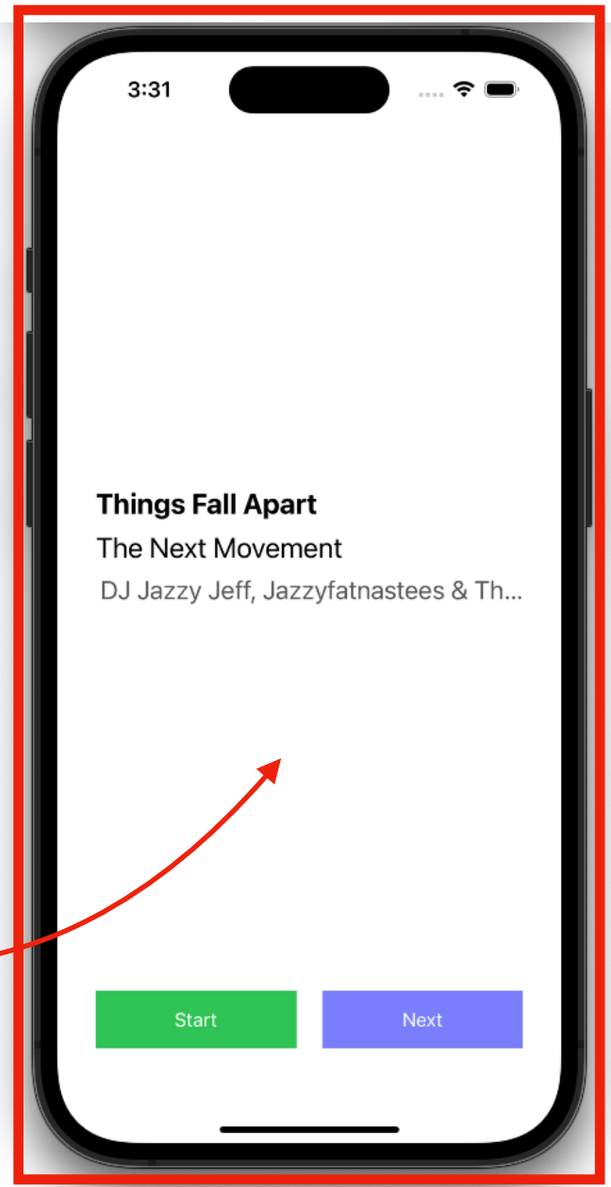
모델 (데이터)

```
struct Music: Codable {
    let songName: String?
    let artistName: String?
    let albumName: String?
    let imageUrl: String?
}
```

뷰모델 (뷰를 위한 데이터)

```
class MusicViewModel {

}
```

화면 (뷰)




모델이 가지고 있는 데이터를 가지고

모든 앱에서 일어나는 일.
결국 어떤 앱이든, 앱에 표시하게
되는 데이터는 모델에서 가져다가..

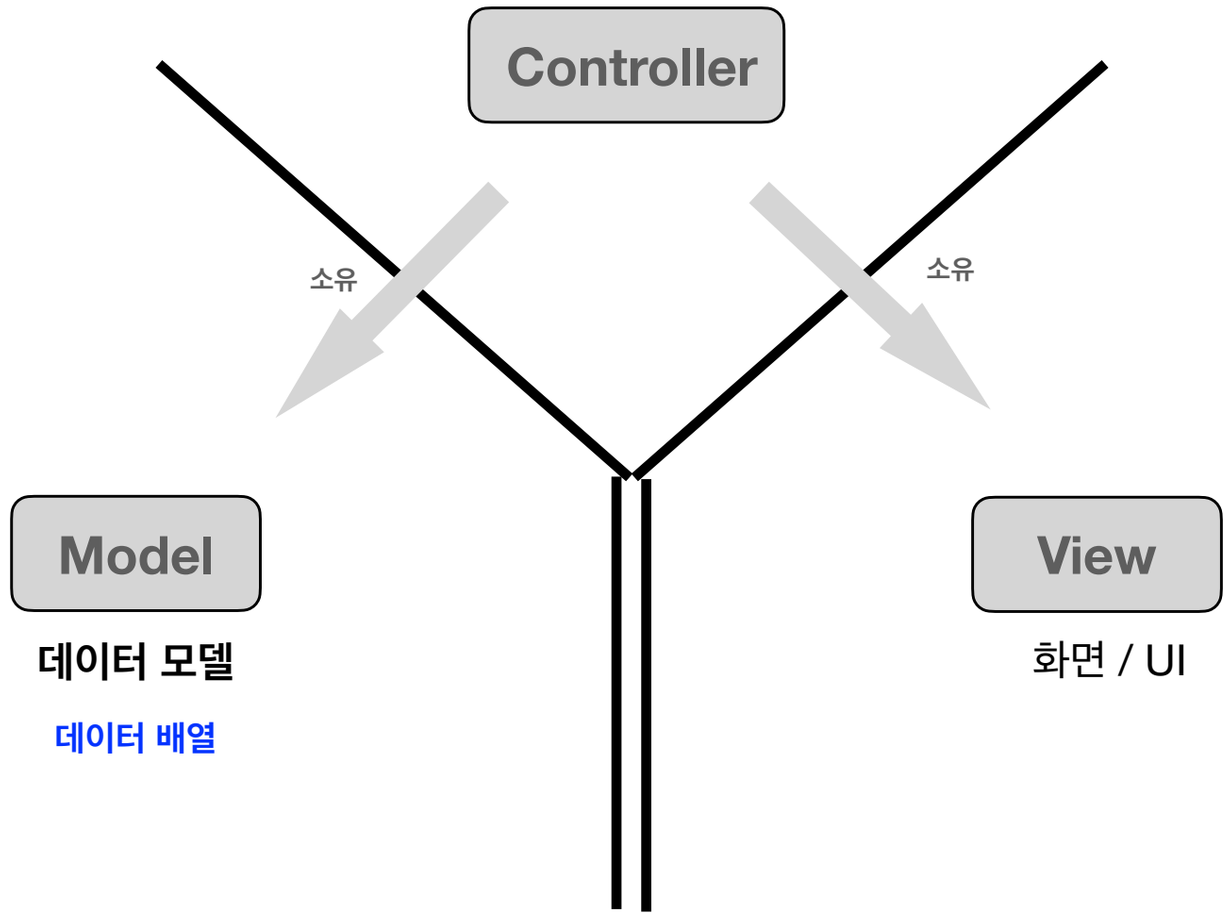
완벽하게 뷰를 위한
데이터의 형태와

뷰를 위한 로직

# MVC

## Model - View - Controller

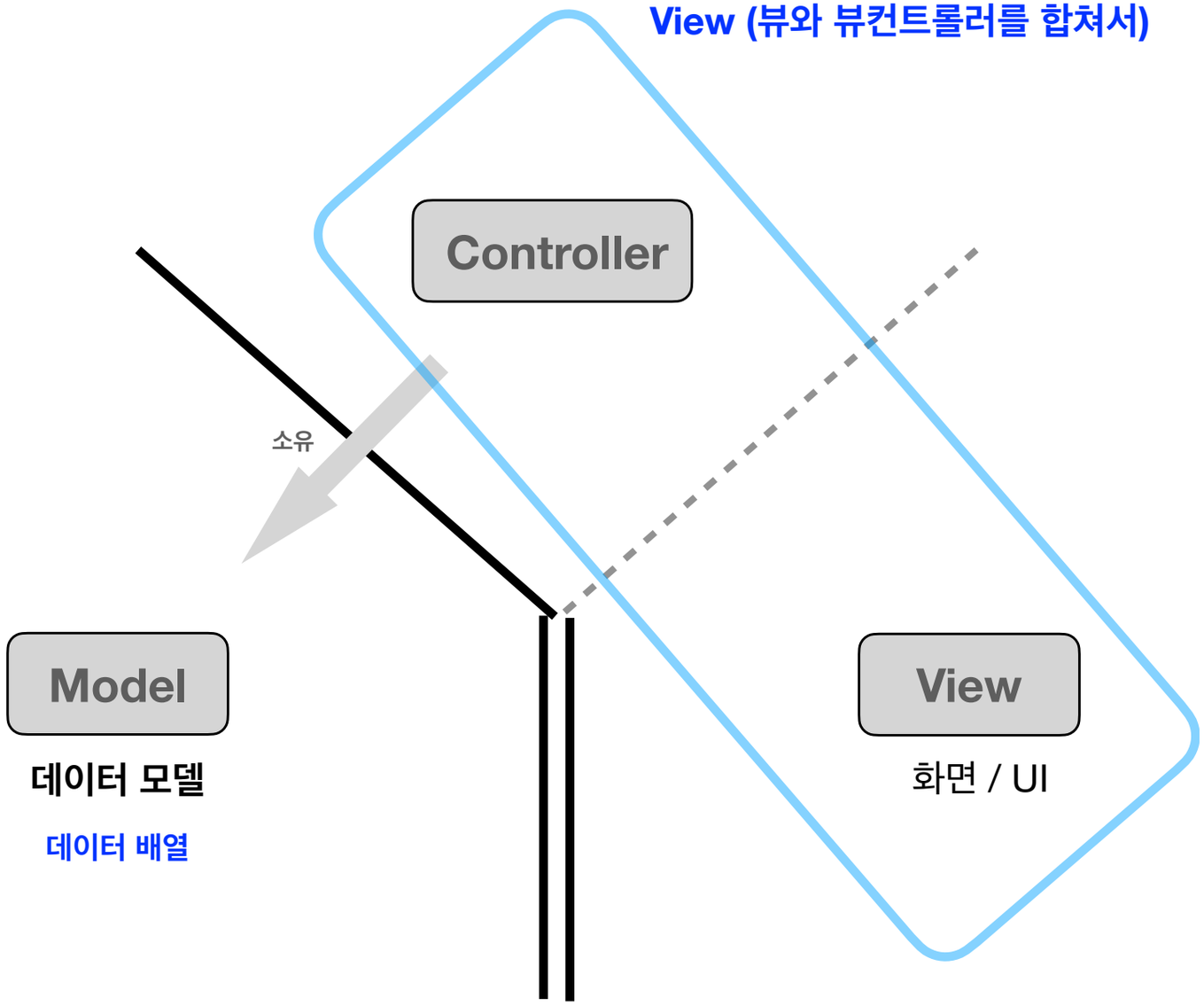
(단순하게) 각자의 역할과 책임을 나누는 것이 목적



# MVVM

## Model - View - ViewModel

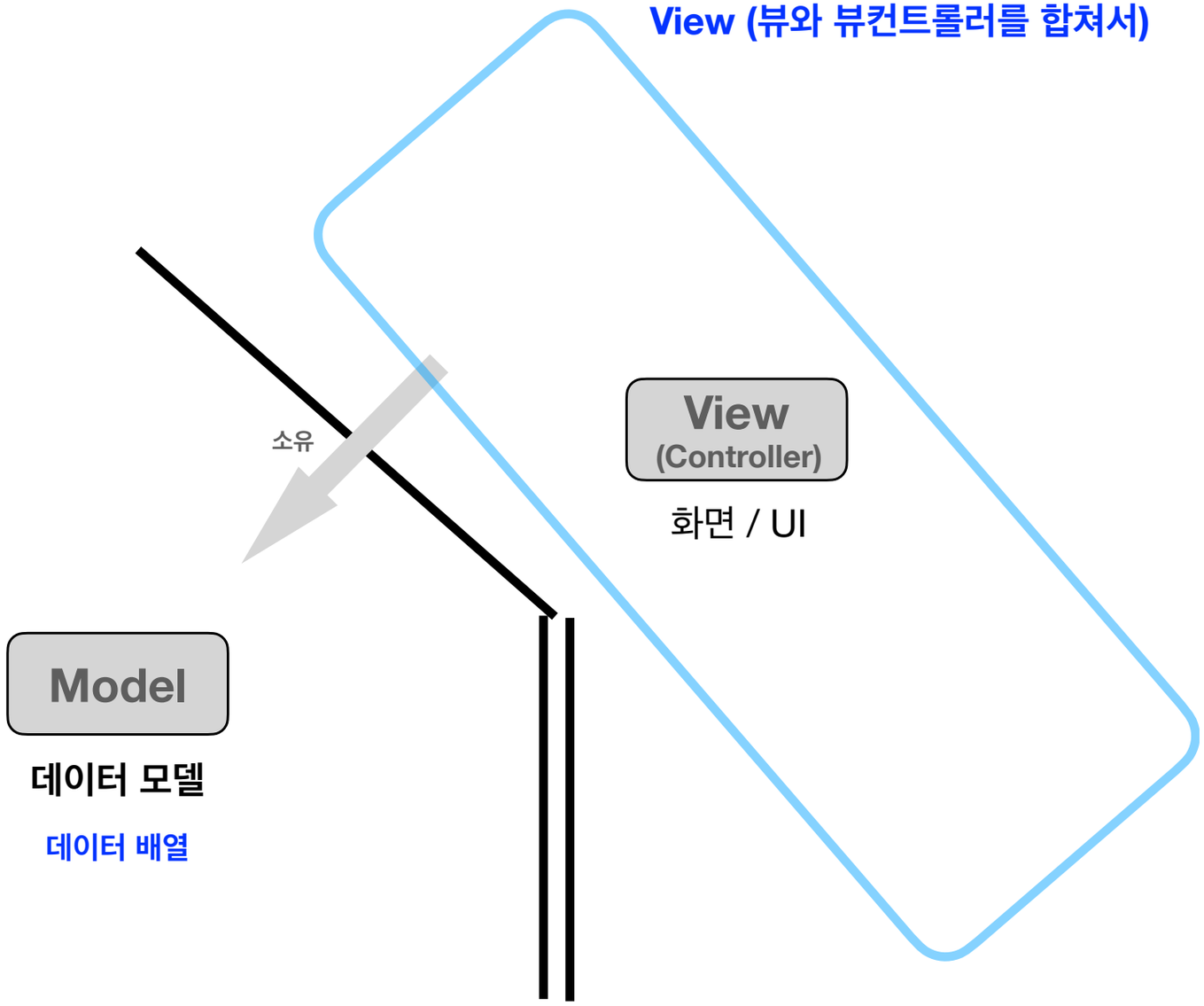
뷰를 위한 모델(뷰모델)을 만들자



# MVVM

## Model - View - ViewModel

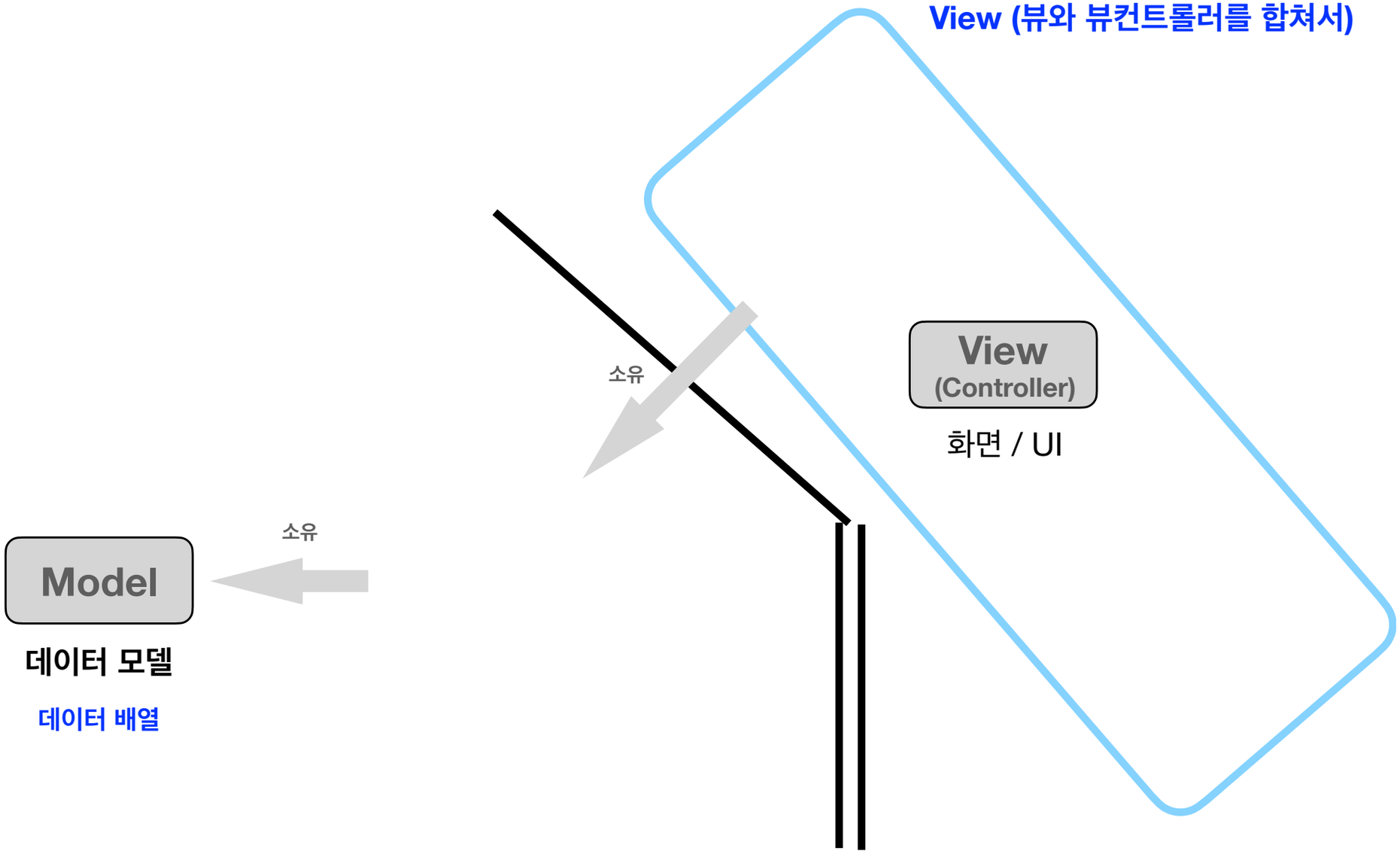
뷰를 위한 모델(뷰모델)을 만들자



# MVVM

## Model - View - ViewModel

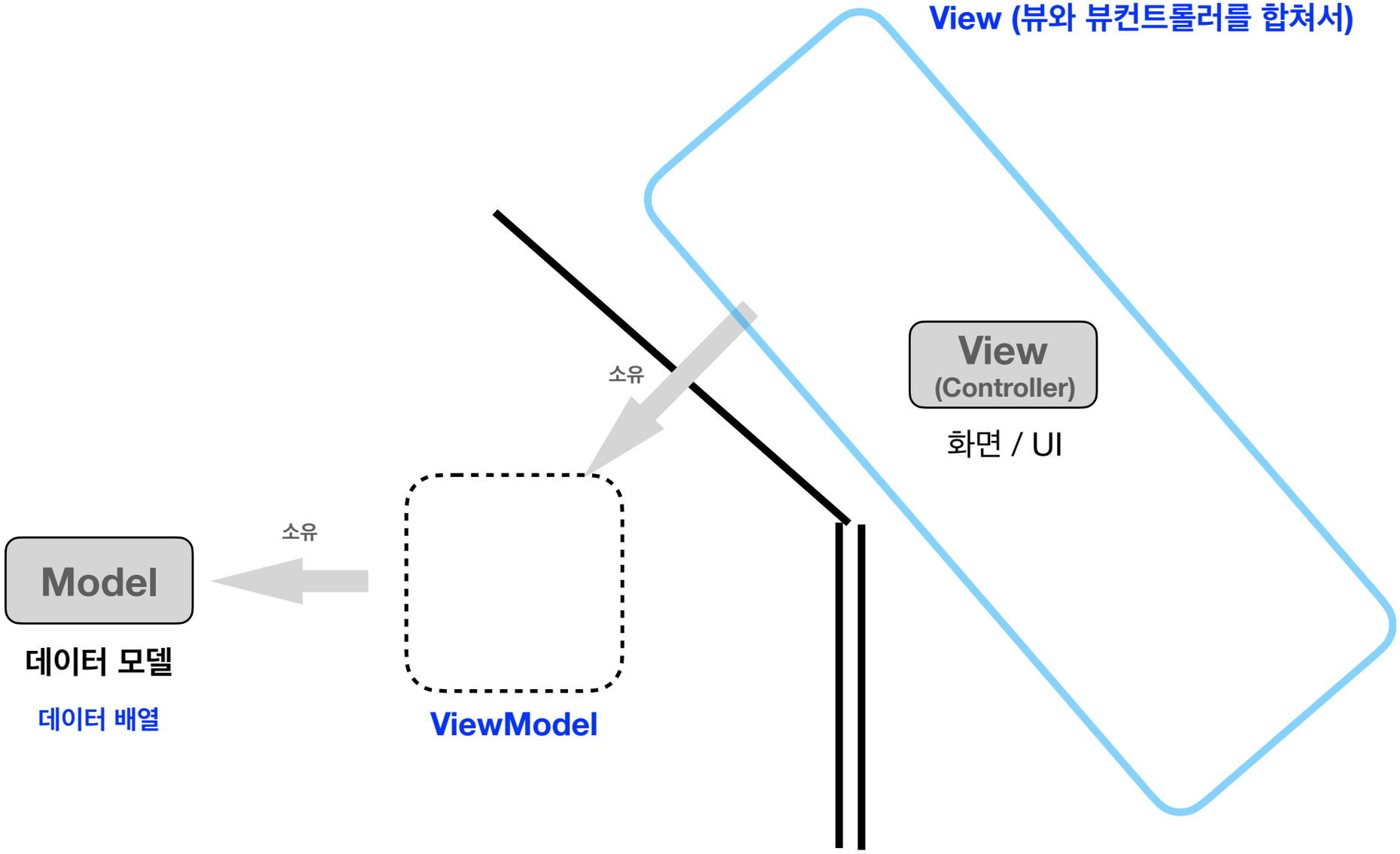
뷰를 위한 모델(뷰모델)을 만들자



# MVVM

## Model - View - ViewModel

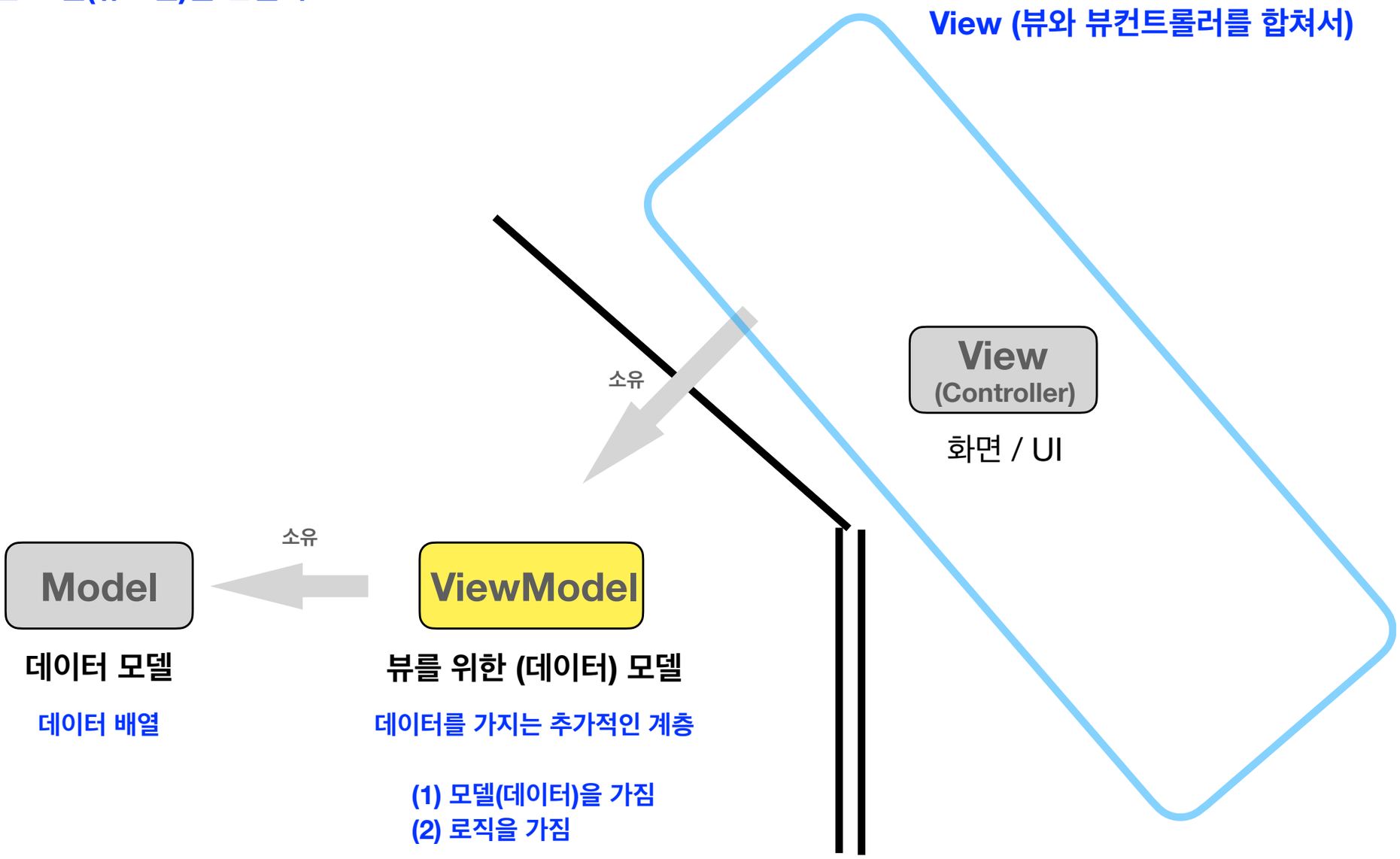
뷰를 위한 모델(뷰모델)을 만들자



# MVVM

## Model - View - ViewModel

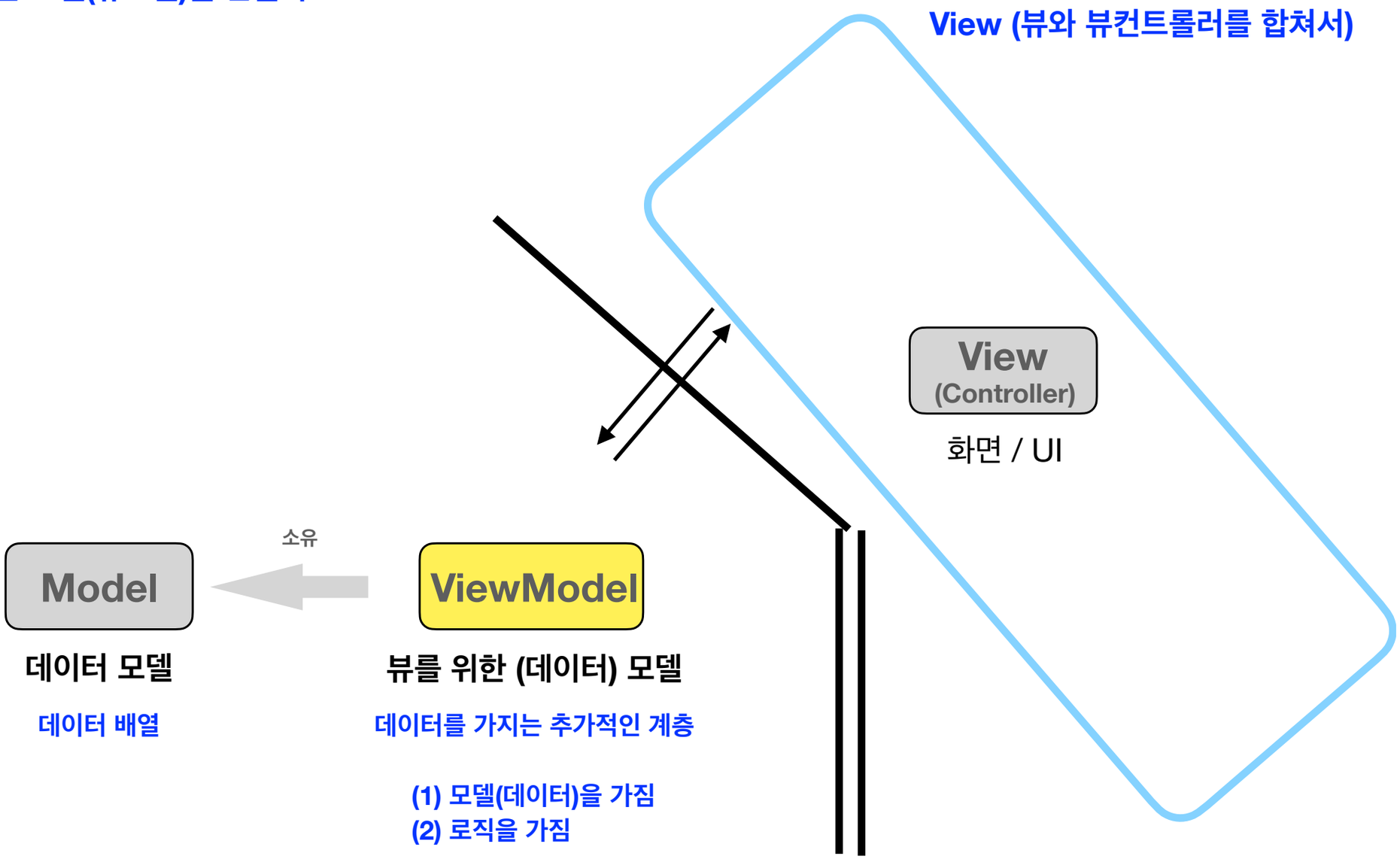
뷰를 위한 모델(뷰모델)을 만들자



# MVVM

## Model - View - ViewModel

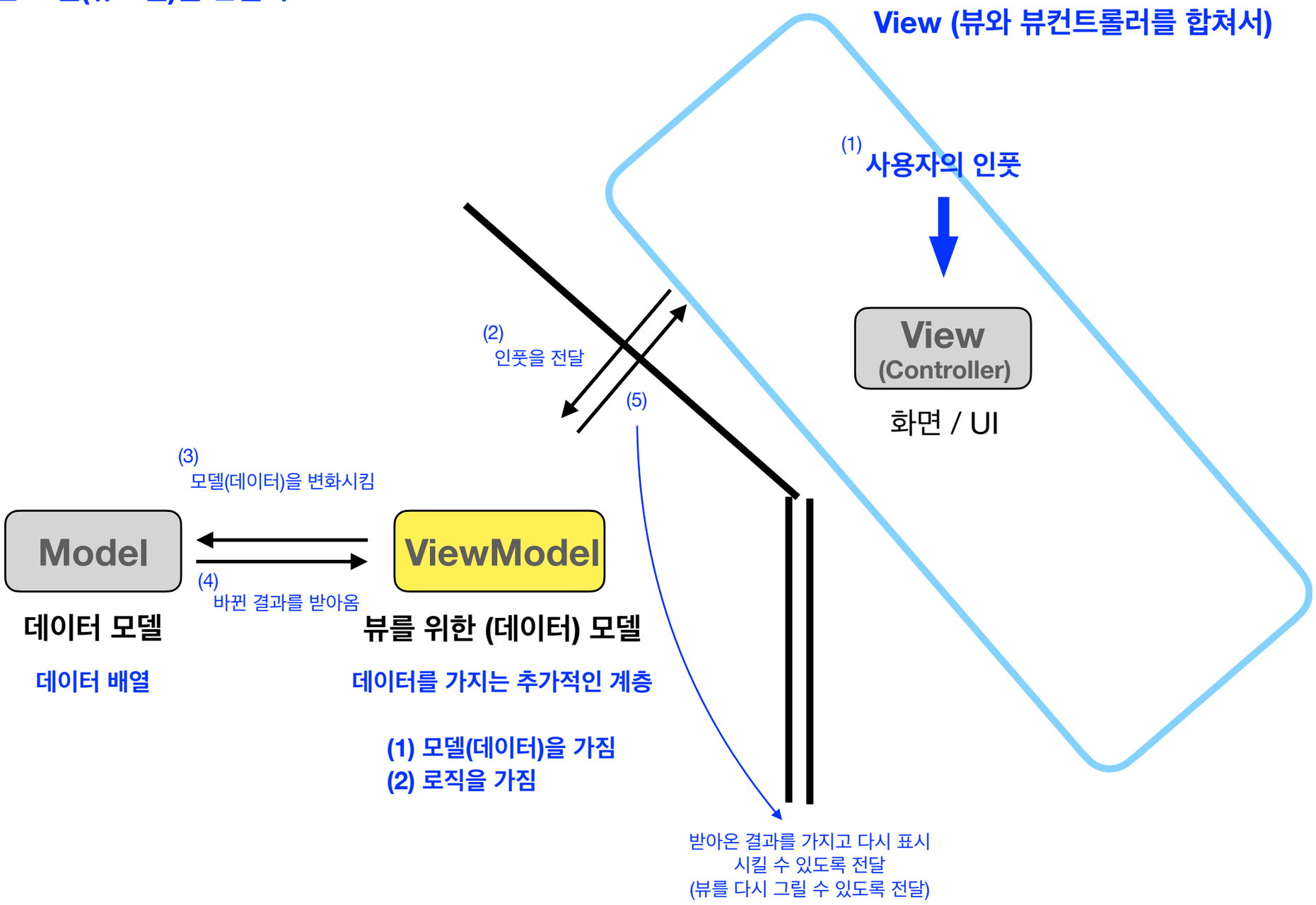
뷰를 위한 모델(뷰모델)을 만들자



# MVVM

## Model - View - ViewModel

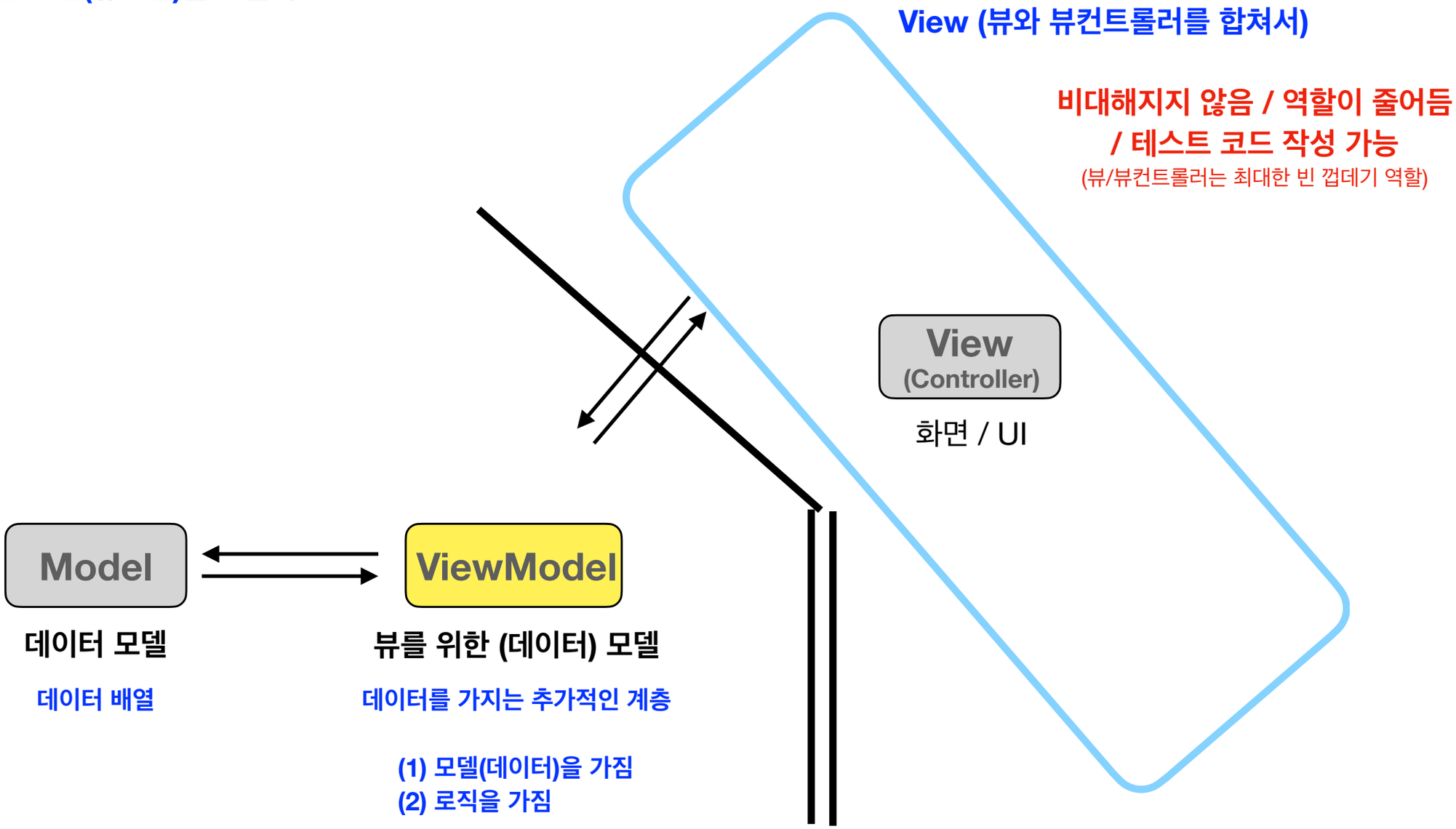
뷰를 위한 모델(뷰모델)을 만들자



# MVVM

## Model - View - ViewModel

뷰를 위한 모델(뷰모델)을 만들자



# 아키텍처 패턴

## MVVM패턴

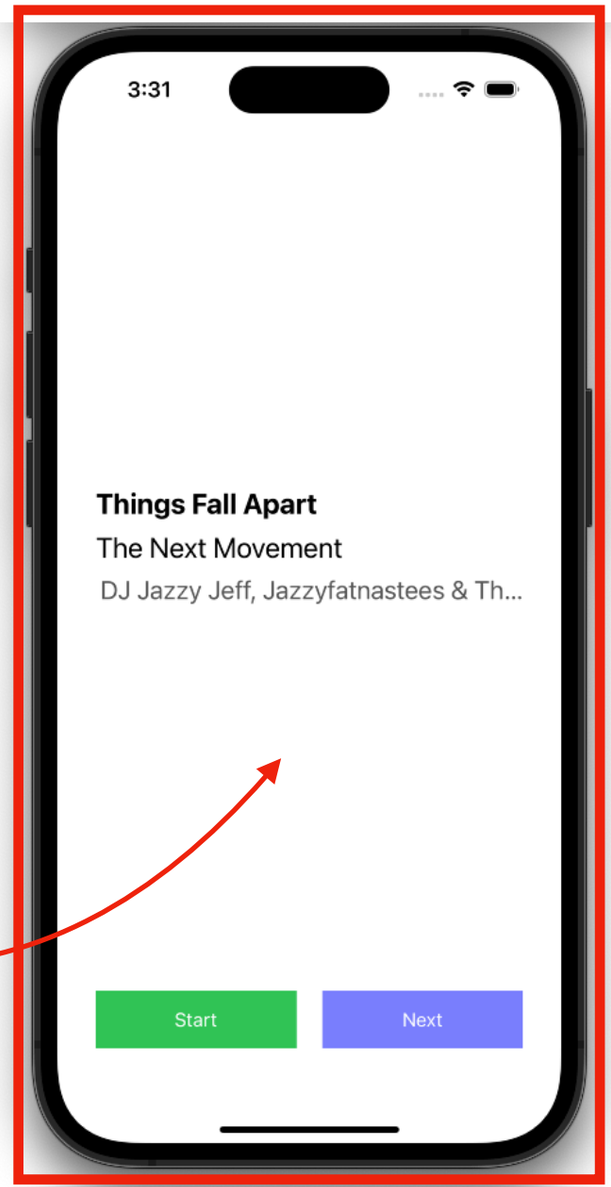
모델 (데이터)

```
struct Music: Codable {
    let songName: String?
    let artistName: String?
    let albumName: String?
    let imageUrl: String?
}
```

뷰모델 (뷰를 위한 데이터)

```
class MusicViewModel {

}
```

화면 (뷰)




모델이 가지고 있는 데이터를 가지고

모든 앱에서 일어나는 일.
결국 어떤 앱이든, 앱에 표시하게
되는 데이터는 모델에서 가져다가..

완벽하게 뷰를 위한
데이터의 형태와

뷰를 위한 로직

# 아키텍처 패턴

## MVVM패턴

모델 (데이터)

```
struct Music: Codable {
    let songName: String?
    let artistName: String?
    let albumName: String?
    let imageUrl: String?
}
```

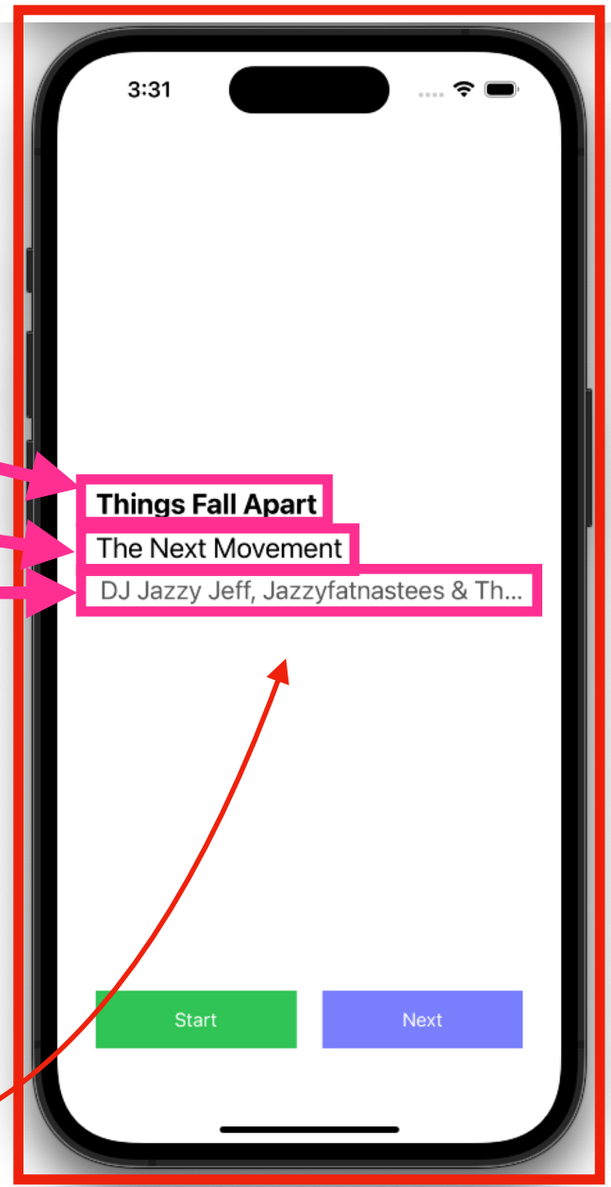
모델이 가지고 있는 데이터를 가지고
뷰를 위한 형태로 가공됨

모든 앱에서 일어나는 일.
결국 어떤 앱이든, 앱에 표시하게
되는 데이터는 모델에서 가져다가..

뷰모델 (뷰를 위한 데이터)

```
class MusicViewModel {

    // 데이터

    var albumName: String?

    var songName: String?

    var artistName: String?


    // 로직

    func handleSomeLogic() {

    }

}
```

화면 (뷰)




완벽하게 뷰를 위한
데이터의 형태와
뷰를 위한 로직

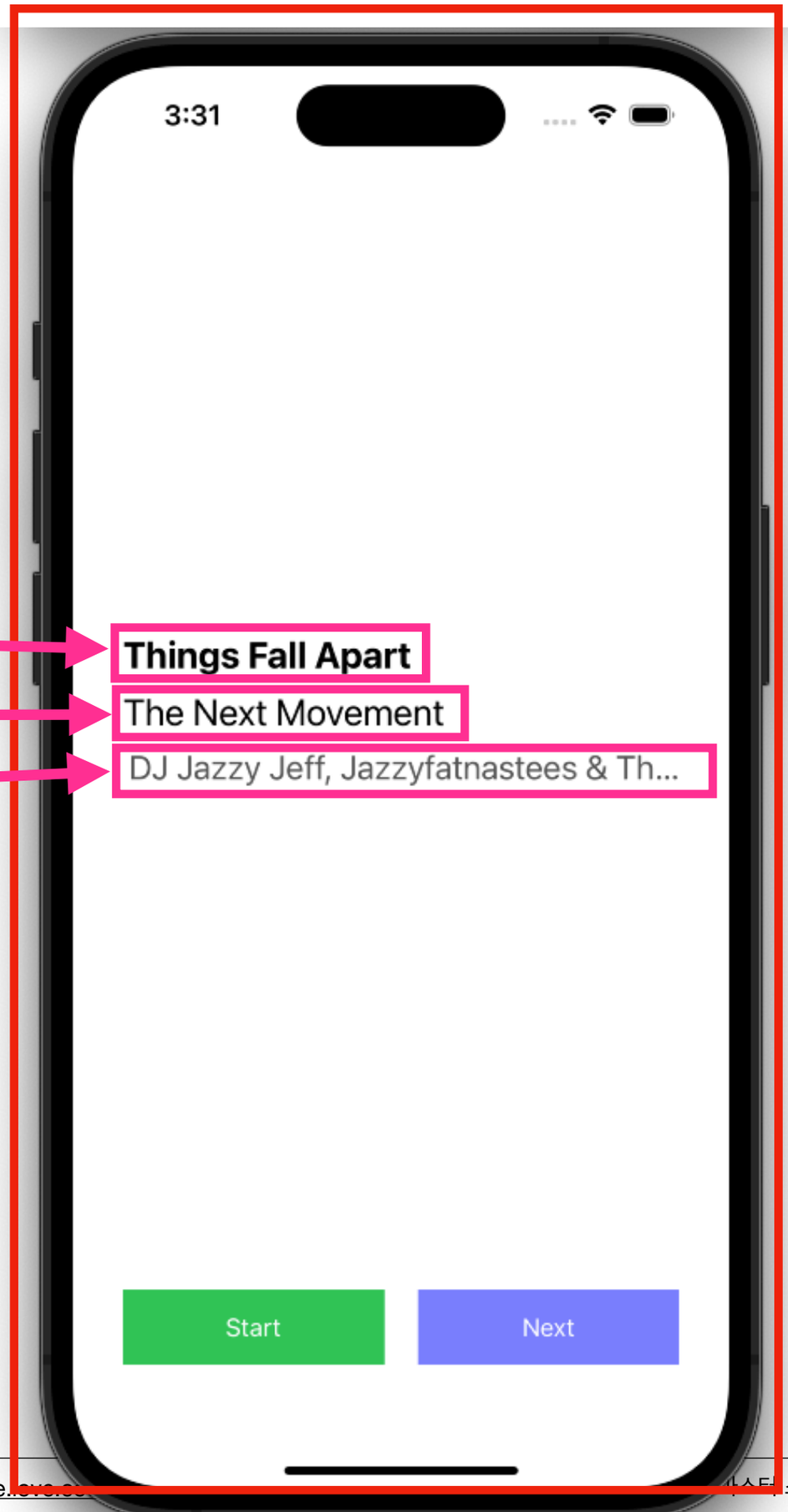
# 아키텍처 패턴

## MVVM패턴

뷰모델 (뷰를 위한 데이터)

```
class MusicViewModel {

    // 데이터

    var albumName: String?

    var songName: String?

    var artistName: String?


    // 로직

    func handleSomeLogic() {

    }

}
```

화면 (뷰)



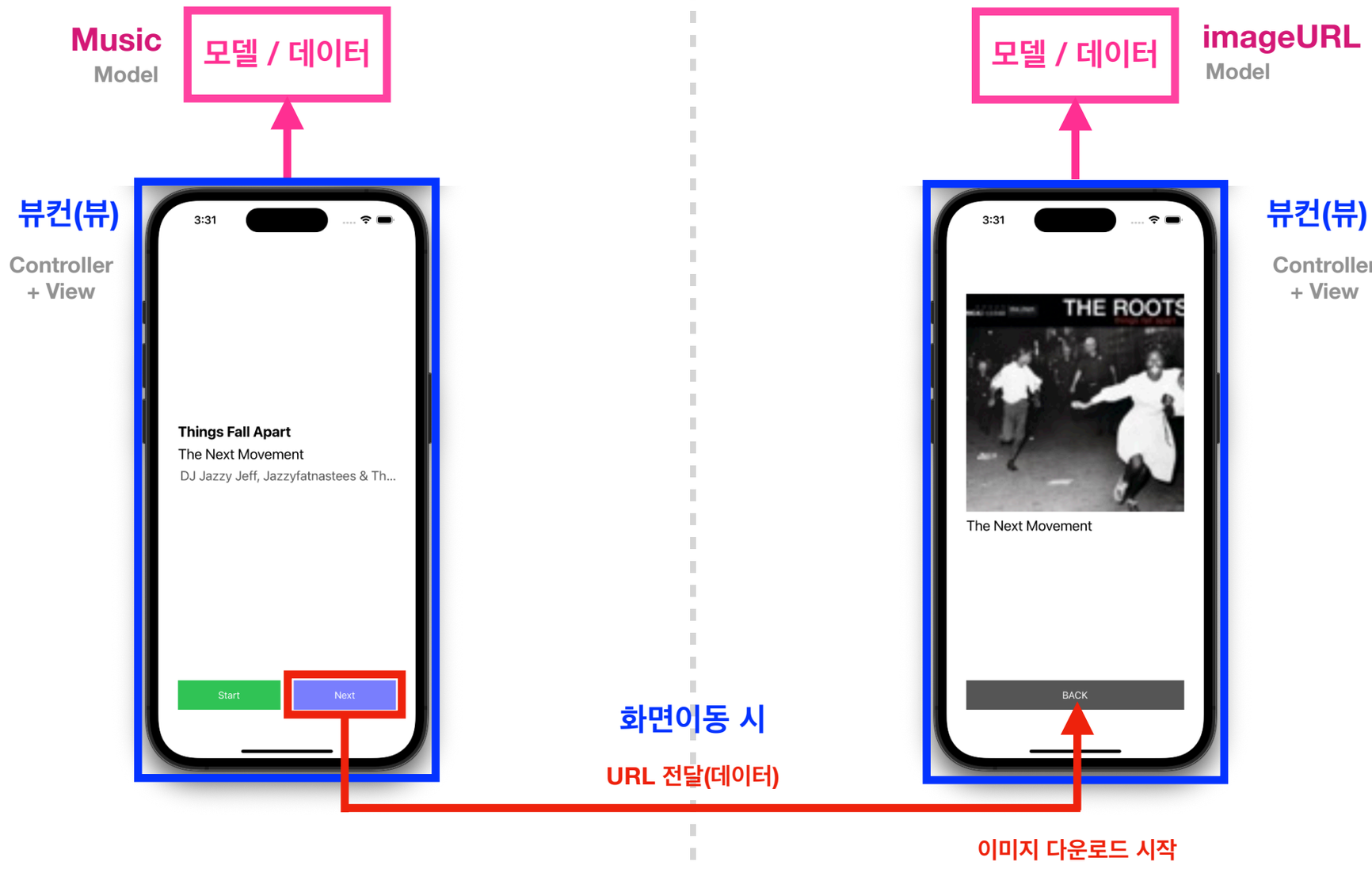

 

# 화면이동과 데이터 전달

**MVVM 아키텍처에서의 화면 이동과 데이터 전달**

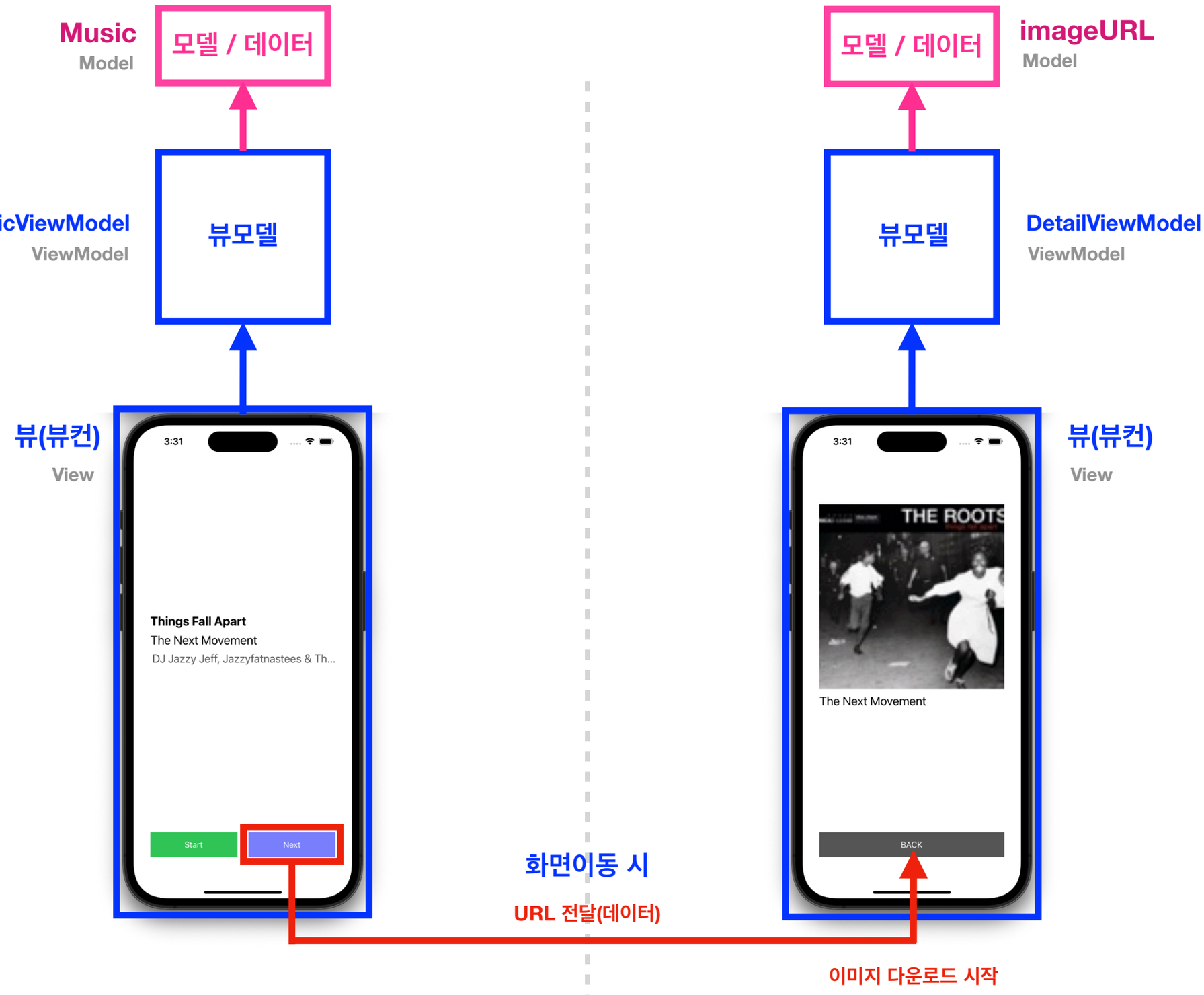
# 아키텍처 패턴

## MVC에서의 데이터 전달



# 아키텍처 패턴

## MVVM에서의 데이터 전달



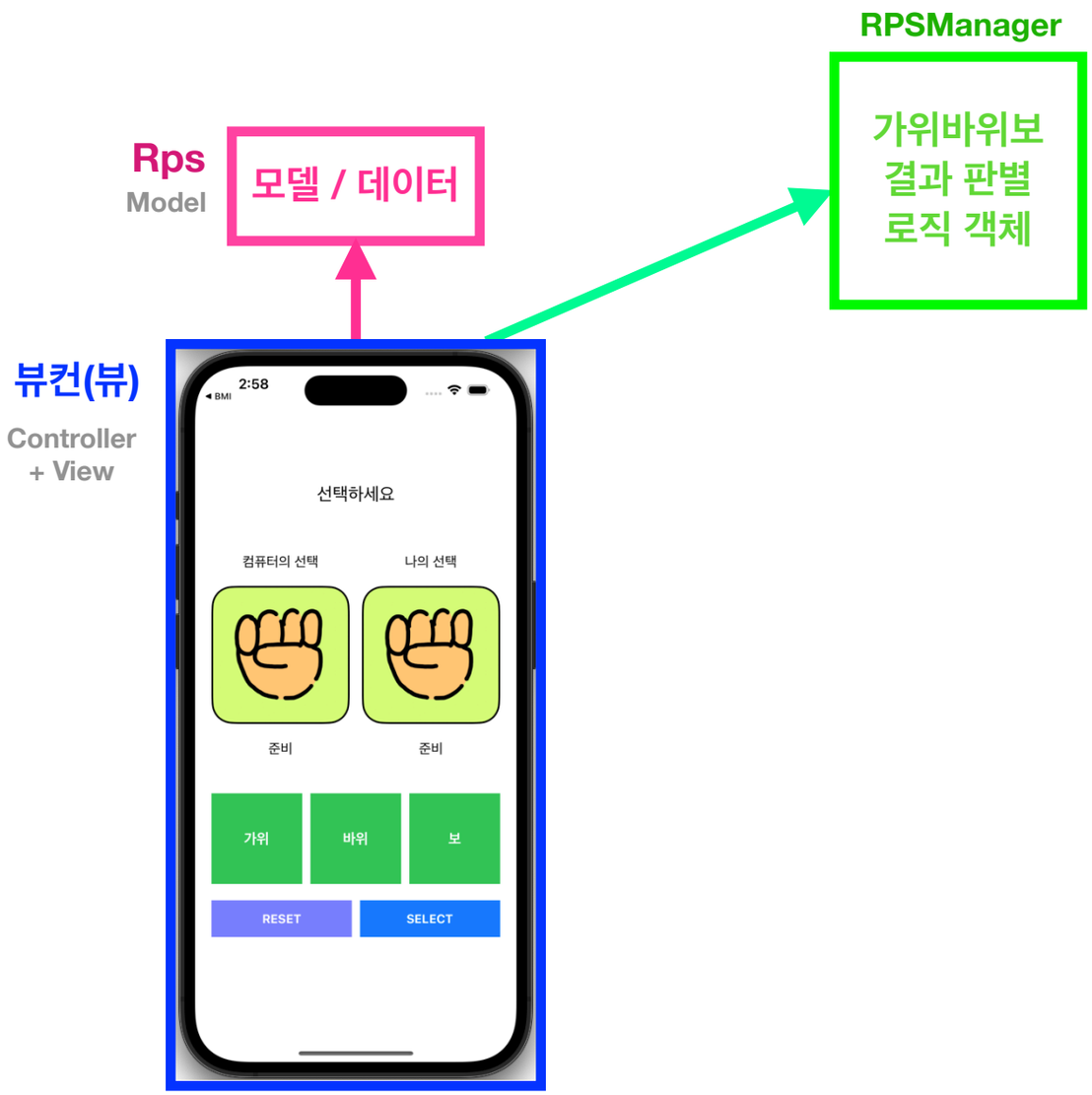
기초 3

# RPS Game 프로젝트 구조

## MVC와 MVVM의 비교

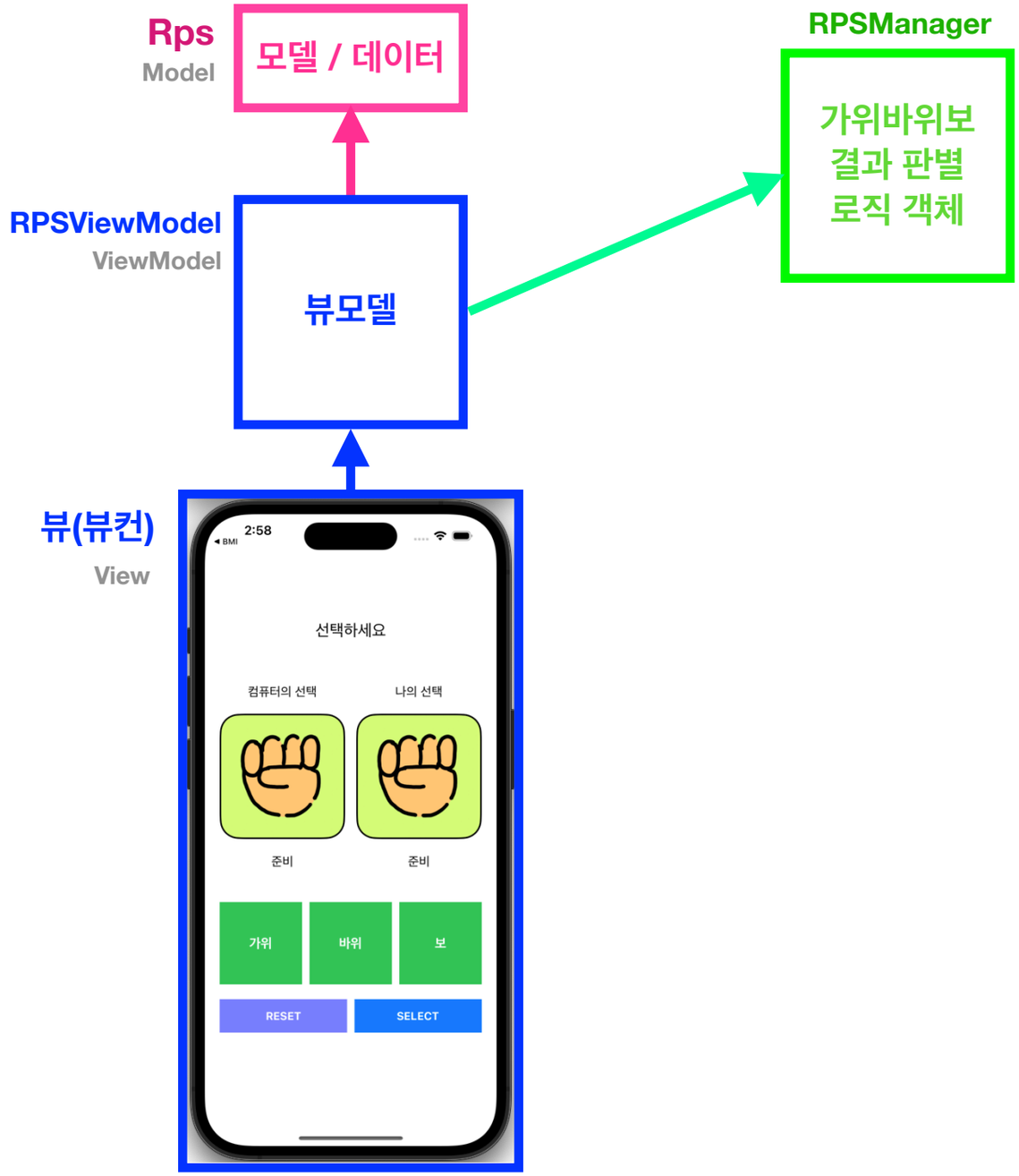
# RPS Game

## MVC 아키텍처



# RPS Game

## MVVM 아키텍처



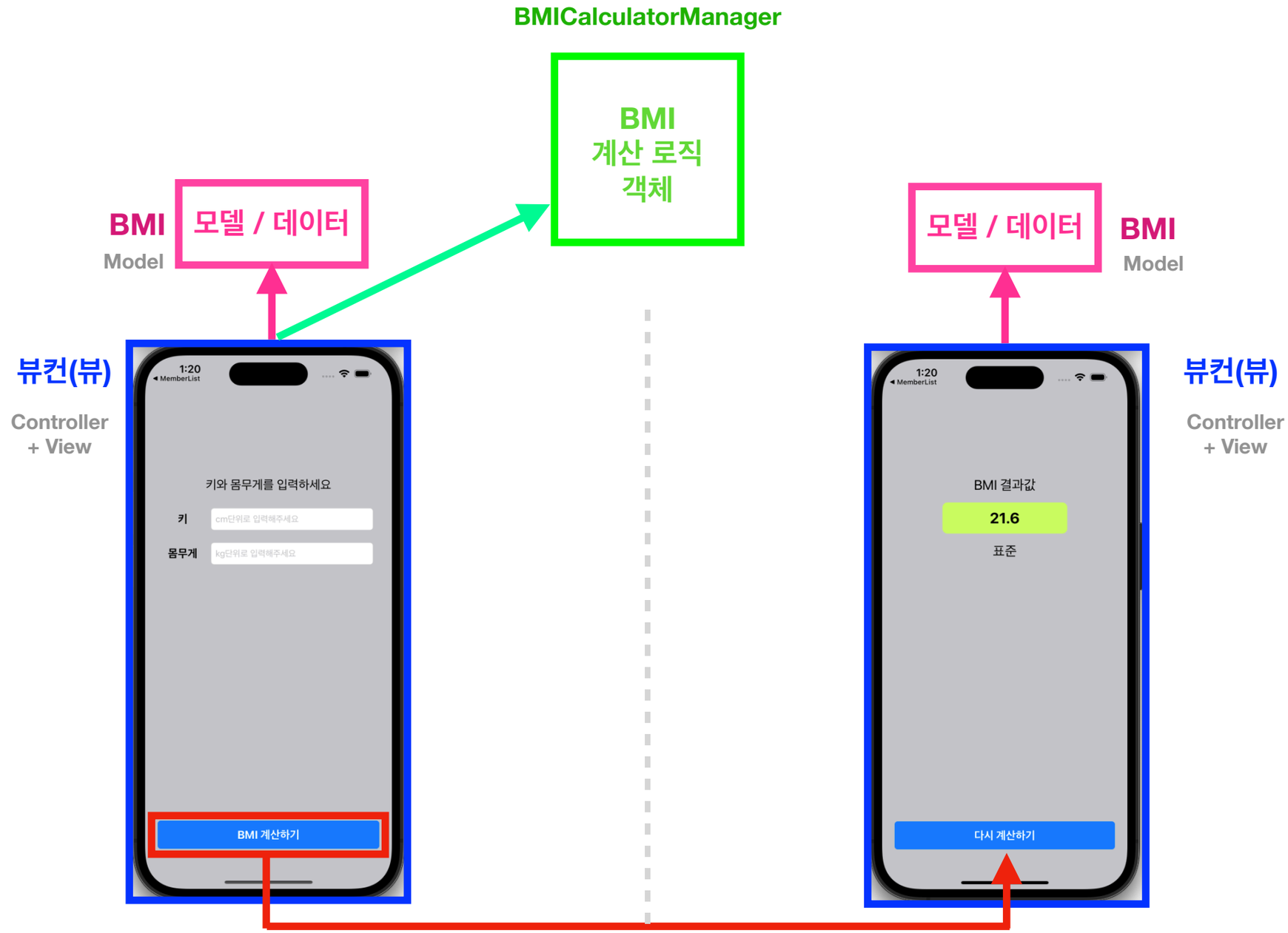
활용 4

# BMI 프로젝트 구조

## MVC와 MVVM의 비교

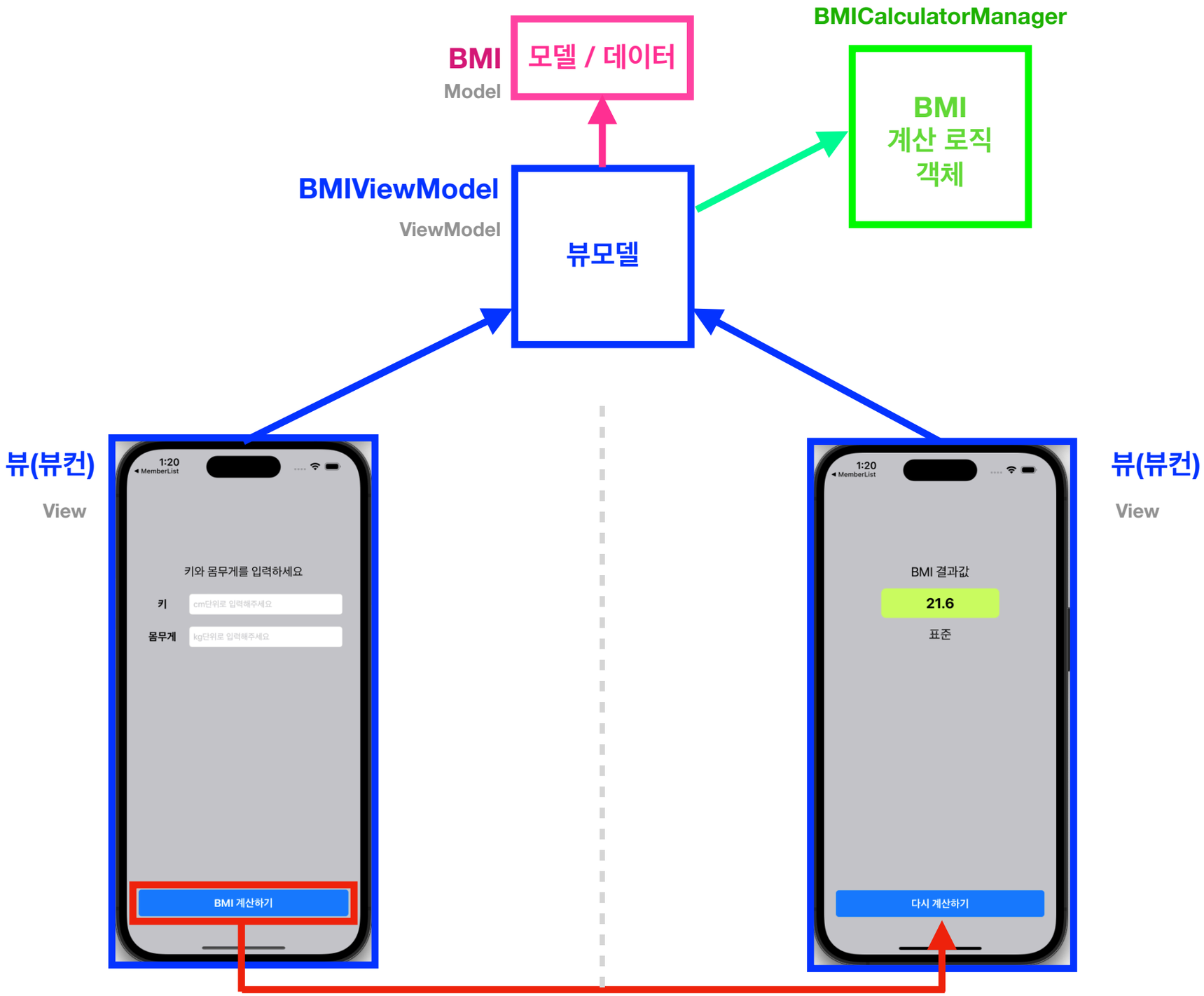
# BMI

## MVC 아키텍처



# BMI

## MVVM 아키텍처



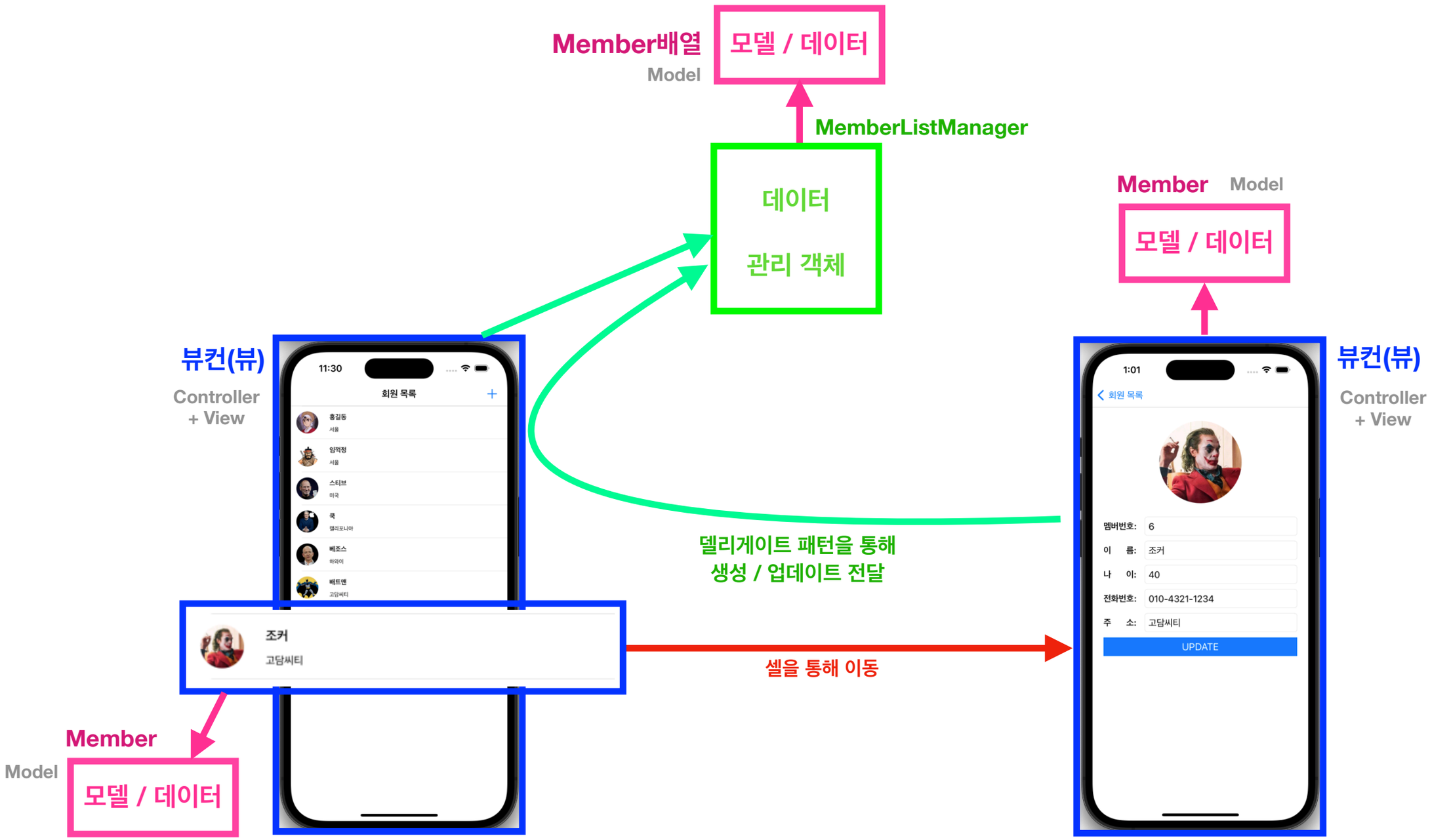
활용 8

# 테이블뷰 프로젝트 구조

## MVC와 MVVM의 비교

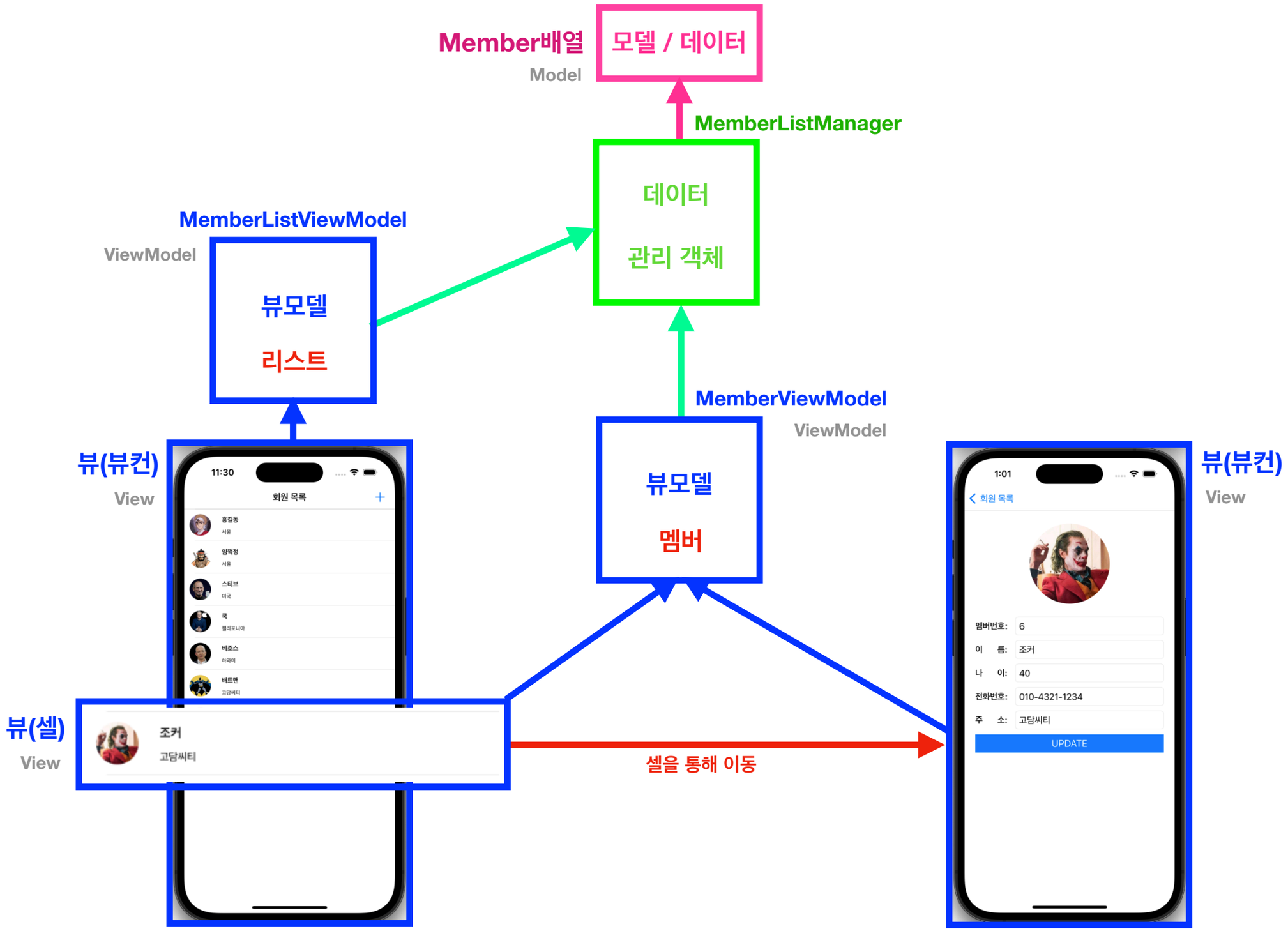
# 테이블뷰(커스텀델리게이트)

## MVC



# 테이블뷰

## MVVM



참고)

# 바인딩 개념의 이해

데이터가 변할때 뷰가 같이 변하면 어떨까?

# 뷰와 뷰모델

## MVVM패턴

뷰모델 (뷰를 위한 데이터)

```
class MusicViewModel {
    var music: Music?

}
```

화면 (뷰)


앨범 제목

노래 제목

가수 이름

**반응형 프로그래밍의 기본 컨셉**

데이터가 변했을때 자동으로 뷰가 변한다면?

# 뷰와 뷰모델

## MVVM패턴

뷰모델 (뷰를 위한 데이터)

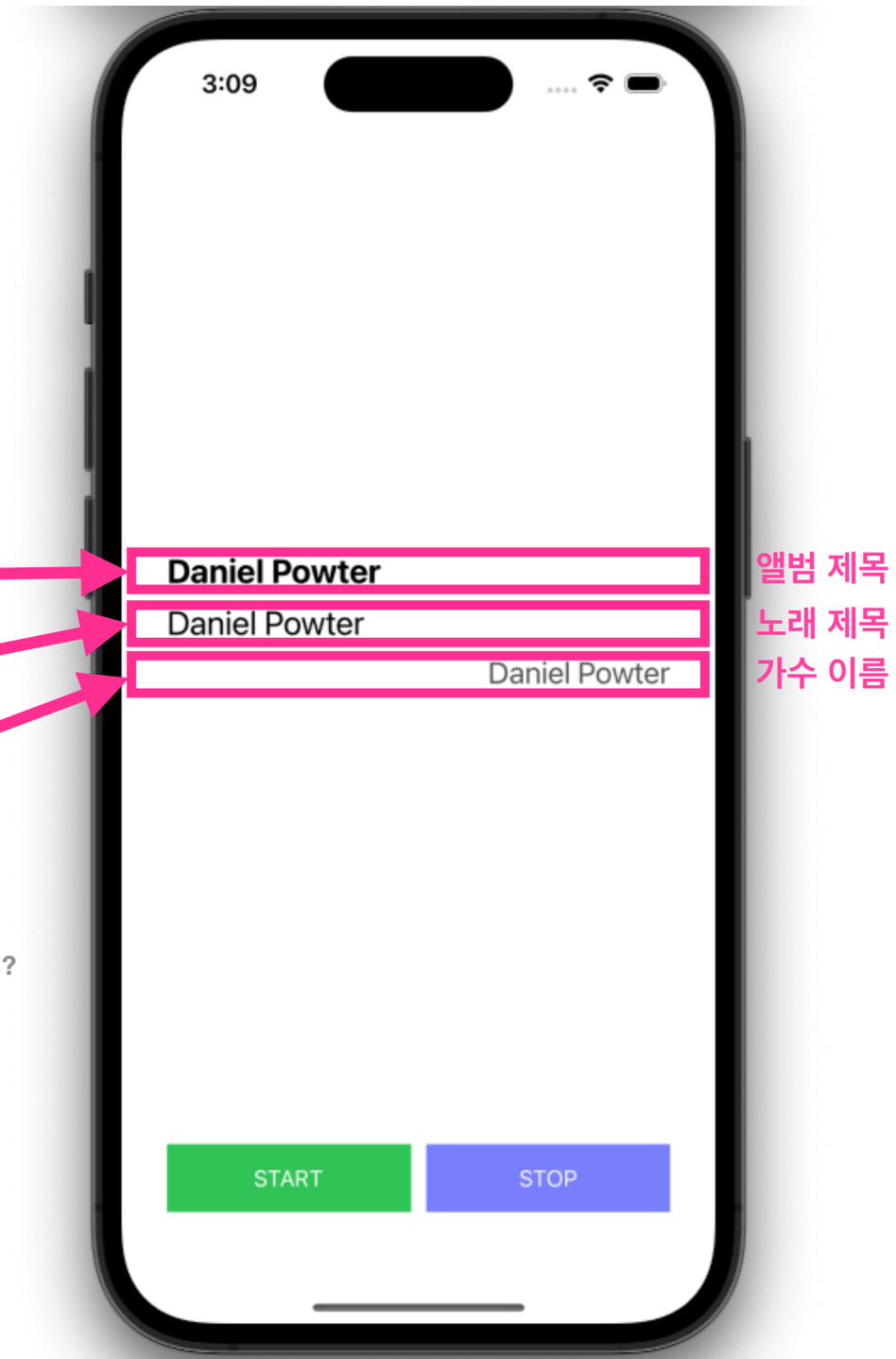
```
class MusicViewModel {

    var music: Music?

    var albumNameString: String? {
        return music?.albumName
    }

    var songNameString: String? {
        return music?.albumName
    }

    var artistNameString: String? {
        return music?.albumName
    }

}
```

1:1 매칭의 형태

**반응형 프로그래밍의 기본 컨셉**

데이터가 변했을때 자동으로 뷰가 변한다면?

화면 (뷰)




앨범 제목

노래 제목

가수 이름

# 바인딩의 구현 방법

**데이터가 변하면 - 뷰도 따라서 변하도록 구현**

- **델리게이트 패턴 (프로토콜 사용)**
- **Notification**
- **클로저 (함수호출)**
- **속성 감시자 (Box 방식)** - 클래스로감싸진데이터 (RxSwift의 기본 컨셉)
- **반응형 프로그래밍**(함수형 프로그래밍) **(RxSwift, Combine)**

# 뷰와 뷰모델

## MVVM패턴

뷰모델 (뷰를 위한 데이터)

```
class MusicViewModel {
    var music: Music?

}
```

화면 (뷰)


앨범 제목

노래 제목

가수 이름

**반응형 프로그래밍의 기본 컨셉**

데이터가 변했을때 자동으로 뷰가 변한다면?

# 뷰와 뷰모델

## MVVM패턴

화면 (뷰)

뷰모델 (뷰를 위한 데이터)

```
class MusicViewModel {
    var music: Music?

}
```

Box

클래스로감싸진데이터

Music 구조체

클래스는 메서드도 가질 수 있고..
여러 기능도 가질 수 있다.

데이터가 변했을때 자동으로 뷰가 변하게
만들기 쉽다.


앨범 제목
노래 제목
가수 이름

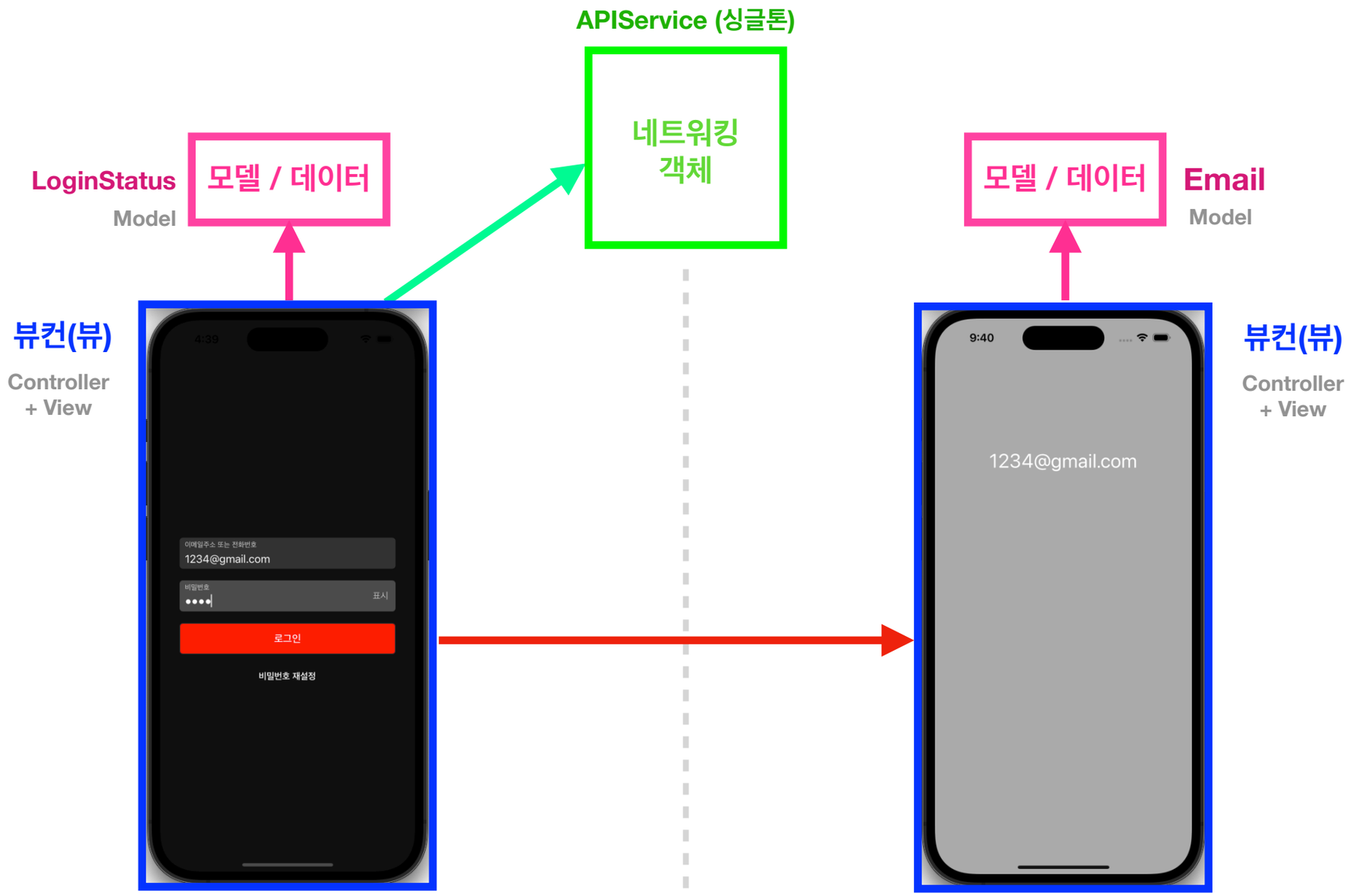
활용 3

# Login 프로젝트 구조

## MVC와 MVVM의 비교

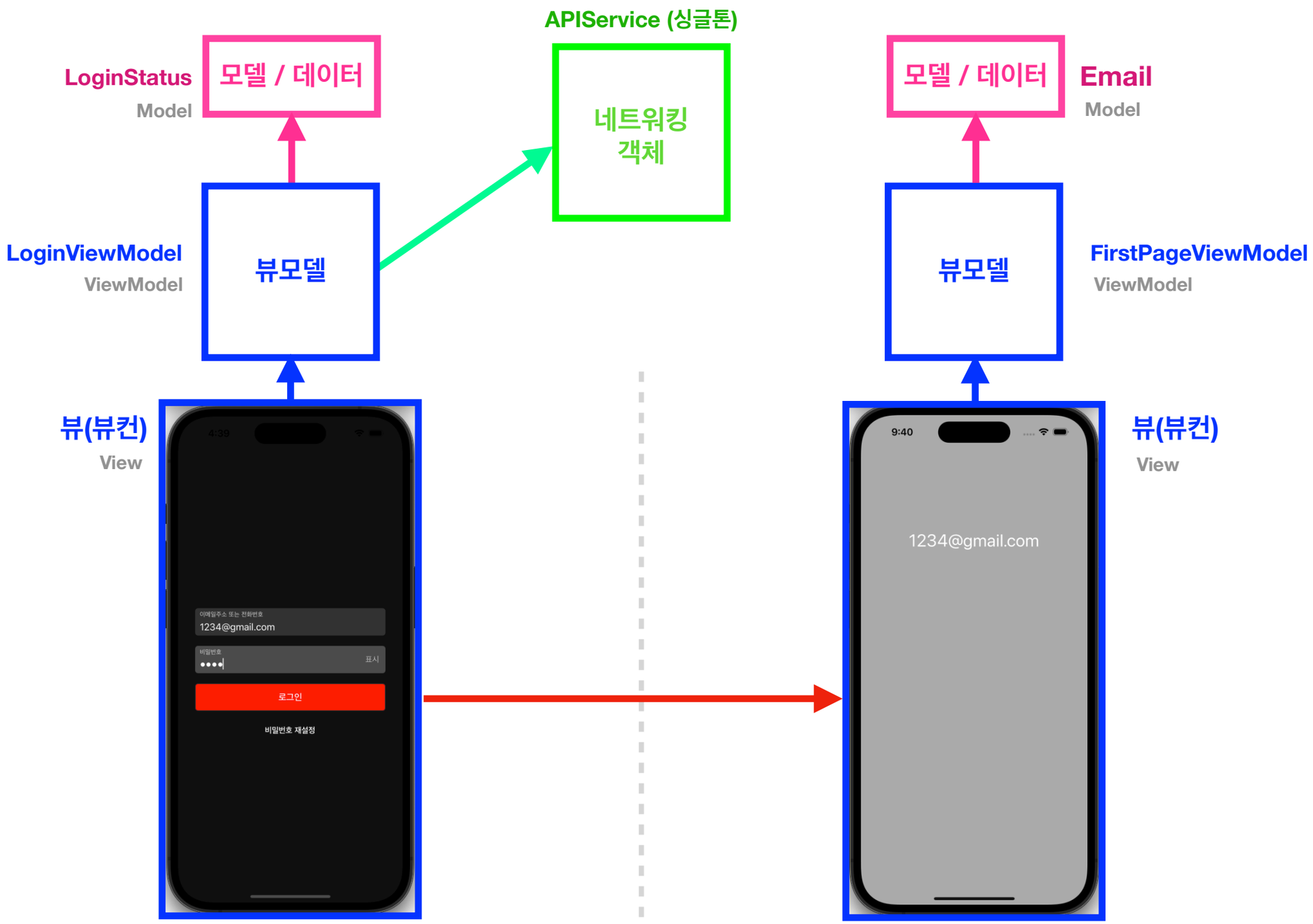
# Login

## MVC 아키텍처



# Login

## MVVM 아키텍처



# 뷰와 뷰모델

## MVVM패턴

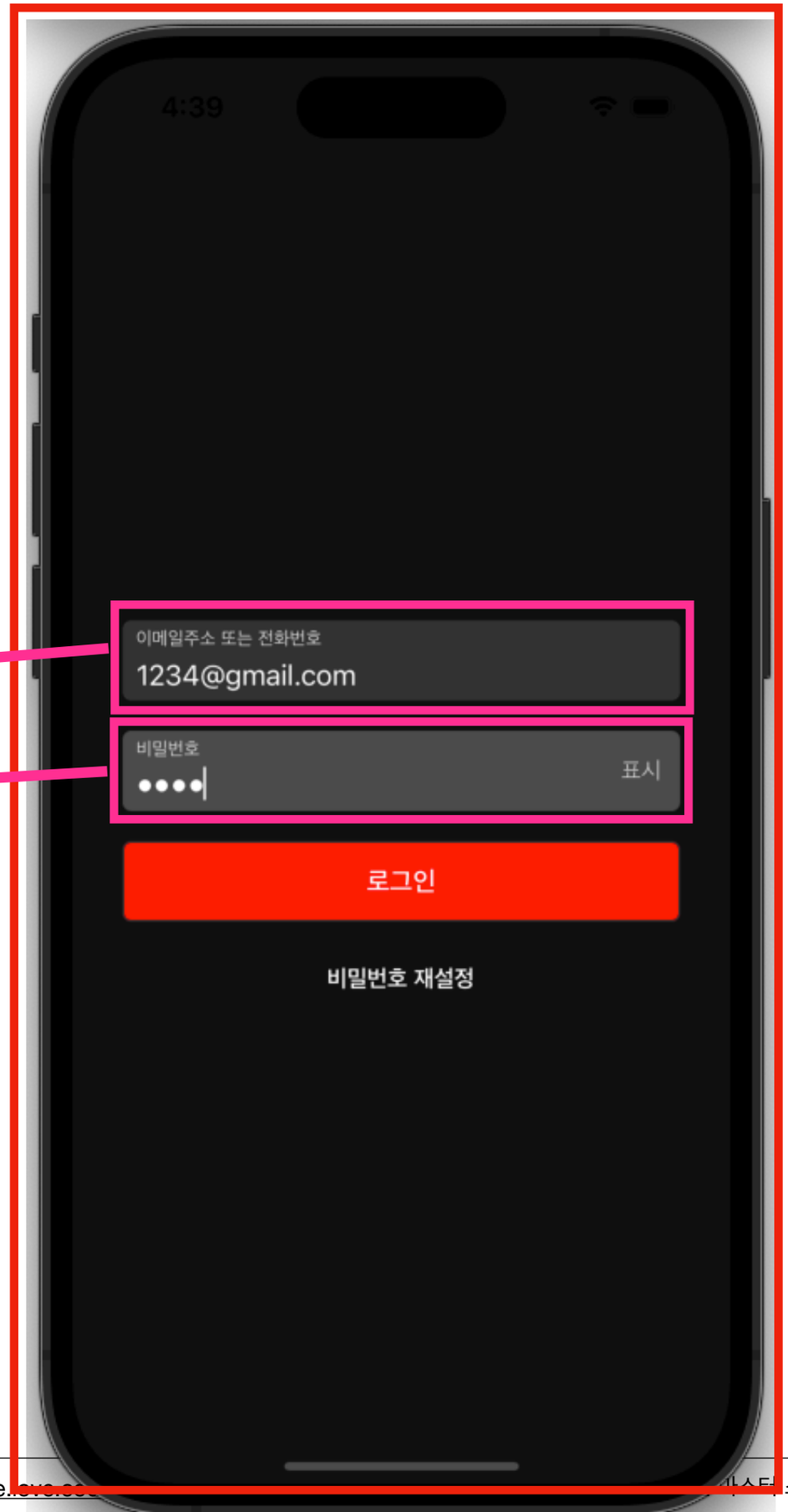
뷰모델 (뷰를 위한 데이터)

```
class LoginViewModel {

    var emailString: String = ""

    var passwordString: String = ""

    var loginStatus: LoginStatus = .none

}
```

setEmailText(...)

setPasswordText(...)

화면 (뷰)



# 뷰와 뷰모델

## MVVM패턴

뷰모델 (뷰를 위한 데이터)

```
class LoginViewModel {

    var emailString: String = ""

    var passwordString: String = ""

    var loginStatus: LoginStatus = .none

}
```

데이터와 일대일 매칭

데이터와 일대일 매칭

(데이터가 변해도 뷰가 변하고, 반대로 뷰에서 변해도 데이터를 변하게 만듦)

화면 (뷰)