앱 안정성 및 확장성 강화를 위한

안드로이드 아키텍처

-UI 계층-

[Part 3. UI 계층 2](#_Toc124166406)

[1. MVx의 대원칙 2](#_Toc124166407)

[2. MVC 패턴 3](#_Toc124166408)

[3. MVVM 패턴 4](#_Toc124166409)

[참고 자료 6](#_Toc124166410)

작성자 : 박현준

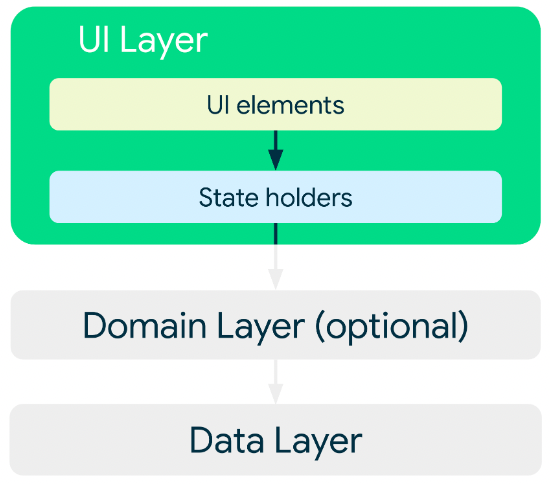
june2ac4dev@gamail.com

22 초안작성

23.1.9 MVP/MVI 내용 삭제 및 오타 수정

# Part 3. UI 계층

## 1. MVx의 대원칙

****

[그림 1.1 일반 아키텍처]

-Model은 분리되어야 한다.

>최소 데이터 레이어에서 처리되는 모든 로직(로컬 DB, remote API 접근 등)은 UI레이어에서 독립되어야 한다.

-뷰의 역할을 할 수 있는 한 분리시켜야 한다.

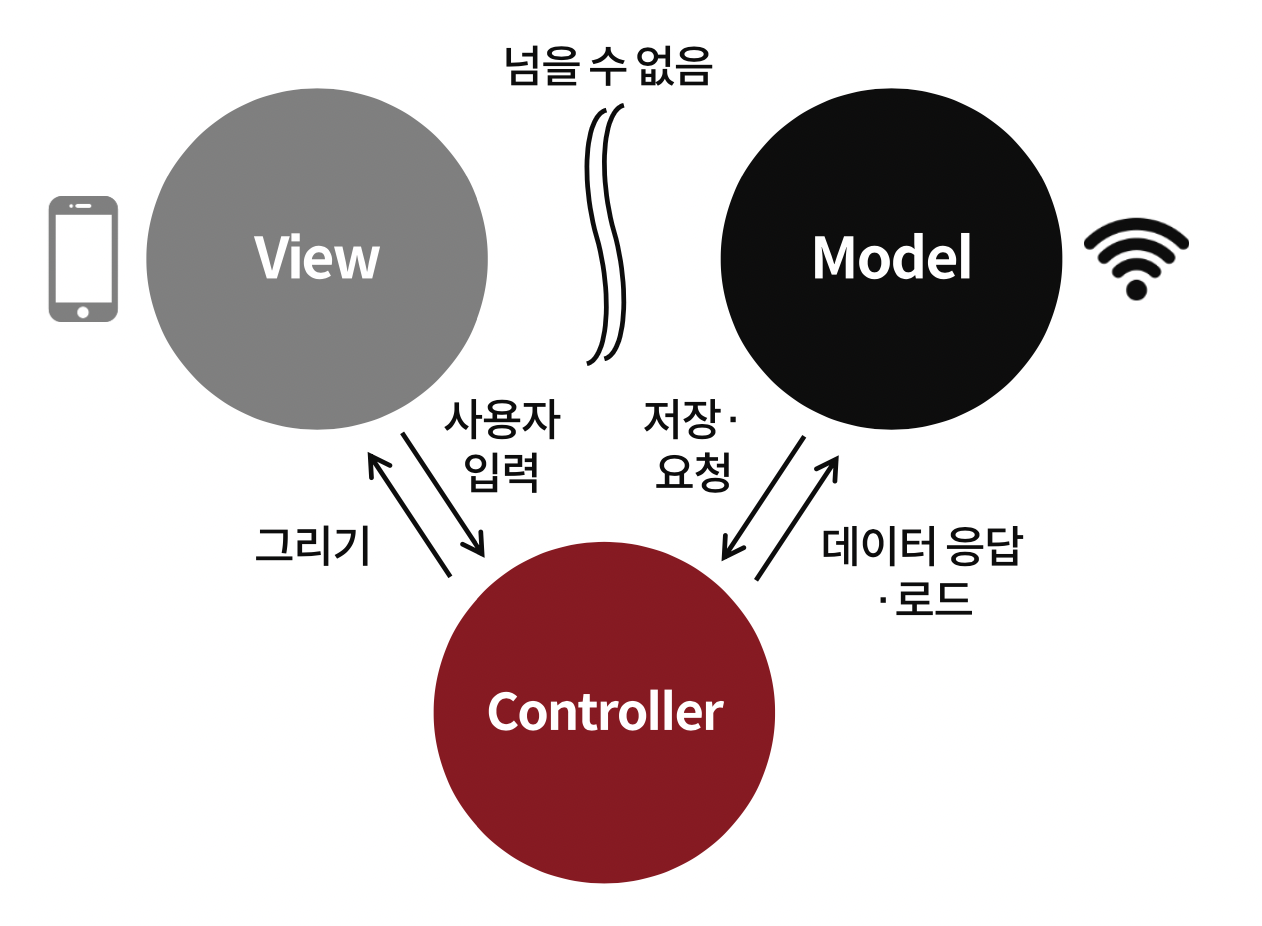
-UI계층에서 안드로이드에서 발생할 수 있는 특수한 상황들을 잘 처리할 수 있는 체계가 필요하다.

>Context처리

>Life Cycle 이벤트 처리: 앱의 안정성을 담보할 필요가 있고, 이벤트 처리 과정에서 사용자에게 기대하지 않는 동작을 보여줘서는 안된다.

>디바이스 종류가 다양하기 때문에 디바이스마다 테스트하는 것이 불가능하지만 가능한 많은 부분이 테스트 가능하도록 만들어 져야한다.

## **2. MVC 패턴**



[그림 2.1 MVC 패턴]

**2.1 MVC 기본 개념**

-모델(M)과 뷰(V)는 일종의 레고 블록. 컨트롤러(C)는 레고블럭의 조율을 담당한다.

-모델 계층(Data Layer + Domain Layer)에서는 비지니스 로직을, 뷰에서는 UI로직을 제공한다.

-컨트롤러는 어떤 뷰를 보여줄 것인가를 결정해 모델에서 받은 데이터/에러를 뷰로 넘겨준다.

-플랫폼을 막론하고 유용하게 적용되는 패러다임. 특히 웹에서 잘 동작한다.

2.2 왜 안드로이드에서 MVC 는 잘 동작하지 않는가?

**2.2.1 모바일 환경의 문제**

-복잡한 비동기 작업과 Life Cycle 처리

-UI 로직 분리의 어려움

cf. 웹의 html에서는 뷰가 컨트롤러와 완전히 독립된 형태로 UI로직 구현 가능하지만, 안드로이드 플랫폼에서는 불가능

**2.2.2 안드로이드 플랫폼 상 문제 : 뷰와 컨트롤러의 분리가 애매하다**

-뷰: 안드로이드의 XML은 기본 레이아웃만을 제공하며, UI로직이 들어갈 여지가 없다.

-컨트롤러: Activity/Fragment가 뷰, 컨트롤러 역할을 모두 담당하게 된다.

>로직이 너무 많이 들어가Fat Activity 발생 -> 가독성, 유지보수성, 확장성 저하 -> Activty 자체가 Context기 때문에 유닛테스트 만들기가 매우 까다로움

2.3 Non-MVC의 설계 전략

**2.3.1 MVP 접근법**

-Activity에서 뷰와 컨트롤러의 역할을 최대한 빼 뷰와 프레젠터로 넘긴다.

-Activity는 객체 생성 및 순수 흐름 관리 위주 역할을 수행한다.

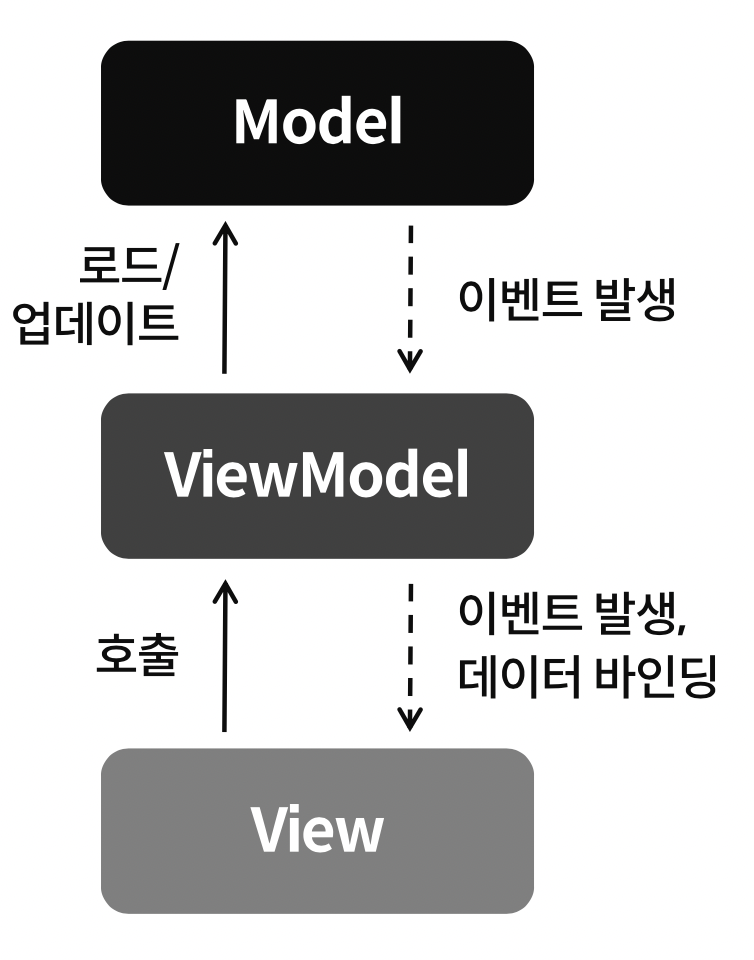
2.3.2 MVVM 접근법

-ViewModel을 추가해 비지니스 로직을 관리한다.

-UI로직을 최대한 dataBinding으로 구현한다.

>단방향 데이터 흐름(UDF)으로 설계해 UI 관련 순환 이벤트 발생을 막는다.

## 3. MVVM 패턴



[그림 3.1 MVVM 패턴]

3.1 MVVM 주요 특징

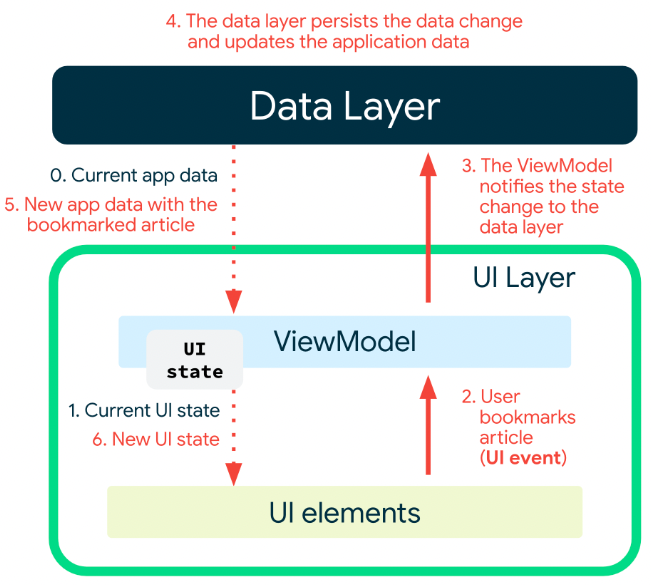
**3.1.1 단방향 데이터 흐름(UDF, Uni-directional Data Flow)**

-뷰는 자신이 받은 사용자 입력을 전달만 할 뿐 직접적으로 결과를 받지 않는다. 대신 간접적인 이벤트 형태로 상태 변경을 통보 받는다.

**3.1.2 VIewModel**

-뷰모델은 뷰를 제어하지 않으며, 뷰는 뷰모델을 호출하되, 결과를 Callback / Observable 형태로 받는다.

-뷰모델은 그 자체로 독립적인 시스템으로, 뷰가 없어도 동작 가능하며 결과를 이벤트로 보낼 뿐 수신인의 존재를 모른다.



[그림 3.2 UDF 예시]

3.1.3 MVVM 단점

-이벤트는 기본적으로 반응형으로 받아야 하므로, 이 개념에 대한 숙지가 필요하며 Rx자바, 코루틴 라이브러리에 대한 학습이 필요하다.

-모든 비지니스로직을 뷰모델로 이전 시켰기 때문에 Fat ViewModel이 될 위험성이 있다.

-순환 이벤트 흐름을 완전히 막기 어렵다. -> Databinding 을 사용해 순환 이벤트 최소화

3.1.4 Fat ViewModel 해결책

-뷰를 최대한 분리, 그리고 뷰마다 뷰모델도 역시 함께 분리한다.

>Activity/Fragment 의 사이즈가 작다면 뷰모델과 1:1 구조가 맞으나, 사이즈가 크다면 일정 기준(ex 통신 서버, 로컬 DB 종류 등)으로 뷰모델을 분리해 1:N 구조로 설계할 수 있다.

>ViewModel간 데이터를 공유할 수 있는 공유 ViewModel을 설계할 수 있다.

-뷰모델의 비지니스 로직 중 도메인 로직에 해당하는 부분은 분리하여 도메인 계층으로 이동시킬 수 있다.

3.2 데이터 바인딩

-이벤트를 뷰로 바로 적용할 수 있는 바인딩

-데이터 바인딩이 없다면 순환 이벤트 흐림이 발생

cf. 순환 이벤트 : 뷰 이벤트 발생 -> 뷰모델이 처리 후, 이벤트로 알려 줌 -> 뷰가 다시 이를 받음

**3.3 AAC ViewModel**

**3.3.1 AAC ViewModel 에서 제공하는 것**

-생명주기 내 설정 변경과 프로세스 종료가 일어나도 뷰모델 내용이 보존되는 구조를 제공한다.

-일반 변수, Flow, LiveData, Compose State 가 보존 될 수 있도록 하는 API 제공 cf. SavedStateHandle

-코루틴이 뷰모델의 생명주기 내에서 동작될 수 있도록 하는 환경을 제공 cf. ViewModel Scope

**3.3.2 구글은 왜 뷰모델만 AAC에서 지원하는가?**

-기존 Activity/Fragment 의 형태를 깨뜨리지 않으면서 재사용성/생산성 높은 아키텍처를 구현 가능하기 때문

-뷰모델은 UI상태 정보(UI State)를 갖고 있기 때문에, 안드로이드 아키텍처에 가장 잘 어울리는 구조다.

ex. 화면이 회전되어 생명 주기가 다시 시작된다고 해도 뷰모델 내부 데이터(UI State)는 보존된다.

**3.4 ViewModel Antipatterns**

-상태가 유실 될 수 있는 공급자(Provider)를 통해 이벤트를 전달한다.

>Coroutine-Channel / RxJava-Flowable 등의 API는 데이터 전달을 완전히 보장하지 않는다.

cf. Channel : 이전 정보는 유실되어도 상관없는 데이터(ex.실시간 위치 정보 등)를 다룰 때 사용

-UI코드에서 액션까지 구현한다.

>UI는 현재 상태가 무엇이고, 어떻게 반영할지 결정한다. 비지니스 로직, 네비게이션 등은 뷰모델의 영역

-일회성 이벤트(ex. Dialog, SnackBar 등)를 즉시 처리하지 않는다.

>Channel.send(), MutableLiveData.postValue() 등의 API는 낮은 우선순위로 실행될 수 있다.

-UI 사태 처리에 Compose의 State 대신 StateFlow/SharedFlow를 이용한다.->불필요한 오버헤드 발생

-뷰모델 생성자에서 초기화 이상의 일을 한다.

>단일 책임 원칙 위반

>테스크 구현이 어려움

>상속 시 자식에게 과도한 정보 습득 요구

>사용자에게 필요 없는 초기화를 수행하지 않게 만들 선택권 박탈

cf. 생성자에서 하면 좋지 않은 작업

-다른 객체의 생성, 정적 메서드 호출, 필드에 단순 대입 이상의 초기화 로직 구현, if/when 분기 로직 등

# 참고 자료

강사룡 안드로이드 아키텍처 Part3 PPT

안드로이드 공식 문서 – UI레이어

<https://developer.android.com/jetpack/guide/ui-layer?hl=ko>