III.

UI 계층

강 사 룡

Android 설계에서 가장 중요한 부분은 바로 프리젠테이션 계층을 어떻게 구성할 것인가에 대한 것이라고 해도 과언이 아닙니다. 이번 강의에서는 유지보수성이 높고 테스트가 용이한 구조를 만들기 위한 UI 계층 구현을 위한 아키텍처 패턴인 MVP, MVVM, MVI 등의 패턴에 대해서 깊숙이 살펴봅니다.

the Red.

Fast campus

TABLE OF CONTENT.

MVx의 대원칙.

UI 계층(MVx)의 대원칙

어떤 경우이든 Model은 분리되어야 한다

→ 적어도 데이터 계층(로컬 DB 사용, remote API 접근 등)에서 처리되는
모든 로직은 UI 레이어에서 독립

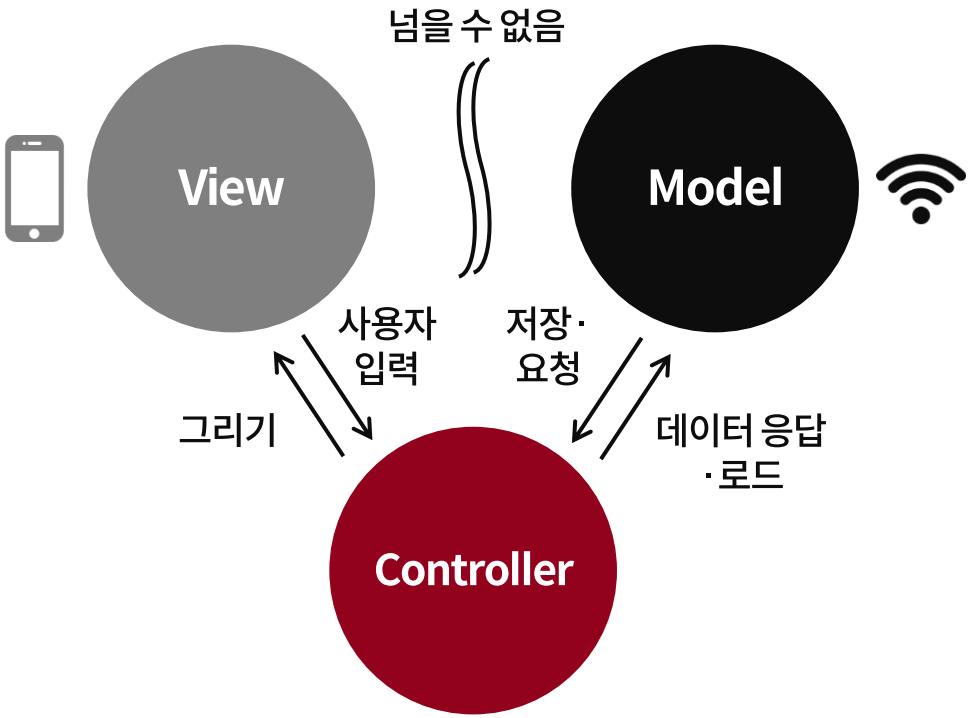
뷰의 역할을 할 수 있는 한 분리시켜야 한다는 공통의 문제 의식

Android에서 발생할 수 있는 특수한 상황들을 잘 처리할 수 있는 체계가 필요

- Context의 처리
- 생명 주기 이벤트(lifecycle event) 처리: 앱의 안정성을 담보할 필요가 있고,
  이벤트 처리 과정에서 사용자에게 기대하지 않은 동작을 보여줘서는 안 됨
- 할 수 있는 한 많은 부분이 테스트 가능하도록 만들어 져야 함

MVC 패턴

- 모델(Model)과 뷰(View)는 일종의 레고 블록
  : 모델 (Data + Domain) 계층에서는 비즈니스 로직을,
  뷰는 UI로직을 제공
- 컨트롤러(Controller)는 레고 블럭의 조립을 담당
- C는 어떤 V를 보여줄 것인가를 결정해서, M에서 받은 데이터를 V로 넘겨줌
- 그리고 에러를 어떻게 보여줄지를 결정해서 V로 넘겨줌
- 플랫폼을 막론하고 유용하게 적용되는 패러다임
  - ✓ 특히 웹 환경에서 잘 동작
    - 뷰와 모델이 각자의 잘 분화된 클래스들을 제공하고,
    컨트롤러가 이들을 잘 조직화 할 수 있음



왜 Android에서 MVC는 잘 동작하지 않는가?

모바일 환경의 문제

- 복잡한 비동기 처리
- 라이프 사이클 처리
- UI 로직 분리의 어려움: 웹의 html에서는 뷰가 컨트롤러와 완전히 독립된 형태로 UI 로직을 구현 가능하지만, 모바일은 그렇지 못함

안드로이드의 문제:
뷰 - 컨트롤러 분리가 애매하다.

- 뷰: Android의 XML은 기본 레이아웃만을 제공. UI 로직이 들어갈 여지가 없음 (특히 리스트 구현)
- 컨트롤러: Activity / Fragment가 컨트롤러 역할을 하는 것이 불가피
- ⇒ 결국 Activity/Fragment가 뷰, 컨트롤러 모두를 담당하게 됨

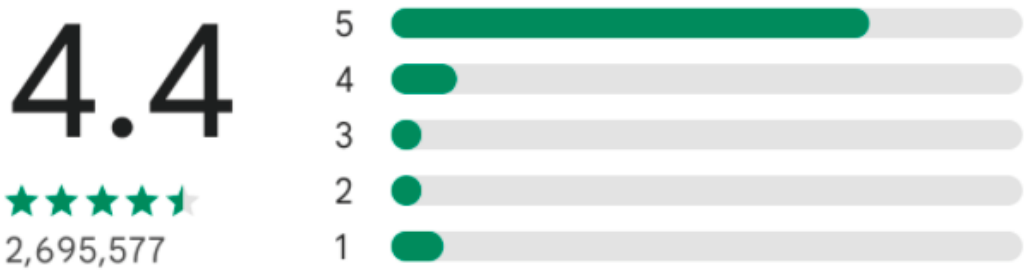
그 결과 - 총체적 난국:
가독성, 유지보수성, 확장성

- Fat Activity: Activity-Fragment에 너무 많은 로직이 들어간다
- Unit Test를 만들기가 매우 까다롭다 - 대부분의 테스트 케이스에 context가 필요

해결책

“
뷰의 분화

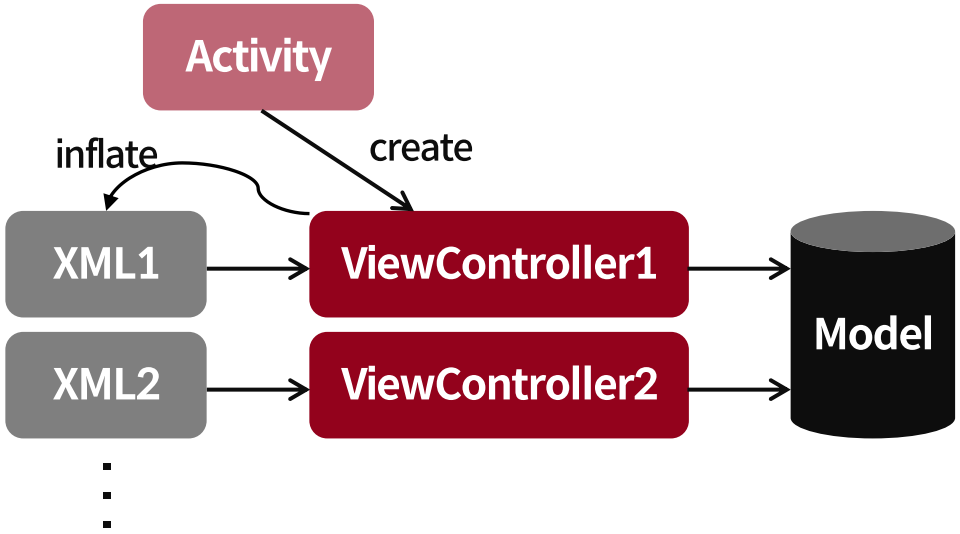
- Android는 <include> 태그를 통해 XML 정의를 여러 개로 분리 가능
- 나눈 View 들은 Activity / Fragment가 아닌 별도의 컨트롤러를 통해 제어



해결책

“ ViewController

- Activity/Fragment는 뷰도 컨트롤러도 아니도록
  → 화면 안 요소들의 생성, 라이프 사이클 처리, 그리고 context에 밀접한 처리를 bridge 해주는 역할만 남겨야
- 각 뷰마다 ViewController를 만들어 뷰의 동작에 관련된 로직 및 컨트롤러 로직을 여기에 구현
- 단, ViewController는 설정 변경(configuration change)으로 인한 라이프사이클 변화에서 살아 남도록 구현해야 함
- 예를 들면, Hilt의 @RetainedActivityScope 지정으로 실현 가능



여전히 남는 문제

사용자 이벤트와 외부 이벤트 등의 효과적인 처리가 여전히 어려움

또 다른 문제는 역시 context

- ViewController의 상당수 동작을 위해 Context가 필요
- Fragment에 연결된 ViewController라면 설정 변경에서 살아남게 만든다는 것이 말처럼 쉽지 않음
- 테스트의 어려움 - 여전히 대부분의 테스트 케이스에 context가 필요하므로 테스트 작성도 까다롭고 테스트 실행 속도도 느림

해법: non-MVC! → 뷰와 컨트롤러를 완전히 분리

- 특히 컨트롤러는 context를 모르고도 대부분의 동작을 수행할 수 있어야 함
- 컨트롤러만으로는 부족: 기본적으로 모바일 환경의 이벤트 처리 문제

Non-MVC의 설계 전략

Non-MVC에서 Activity는 무엇인가?

→ Activity는 컨트롤러! (그것도 아주 제한적인..)

MVP의 (궁극적) 접근법

- Activity에서 뷰와 컨트롤러의 역할을 최대한 빼앗아 뷰와 프리젠터(Presenter)로 넘김
- Activity는 객체 생성 및 순수 흐름 관리 역할 위주

MVVM/MVI 등의 접근법

- 단방향 데이터 흐름(Uni-directional Data Flow): 컨트롤러
  → 뷰 방향의 데이터 흐름을 이벤트를 수신하는 형태로 구현
- 마찬가지로 Activity는 최대한 일부 context 의존 기능만 하도록
- 뷰 logic은 최대한 data binding으로 구현

MVP (Model-View-Presenter).

Model-View-Presenter?

1990년대 초 IBM에서 최초로 구현

2006년 마틴 파울러(Martin Fowler)의 소개로 널리 알려짐

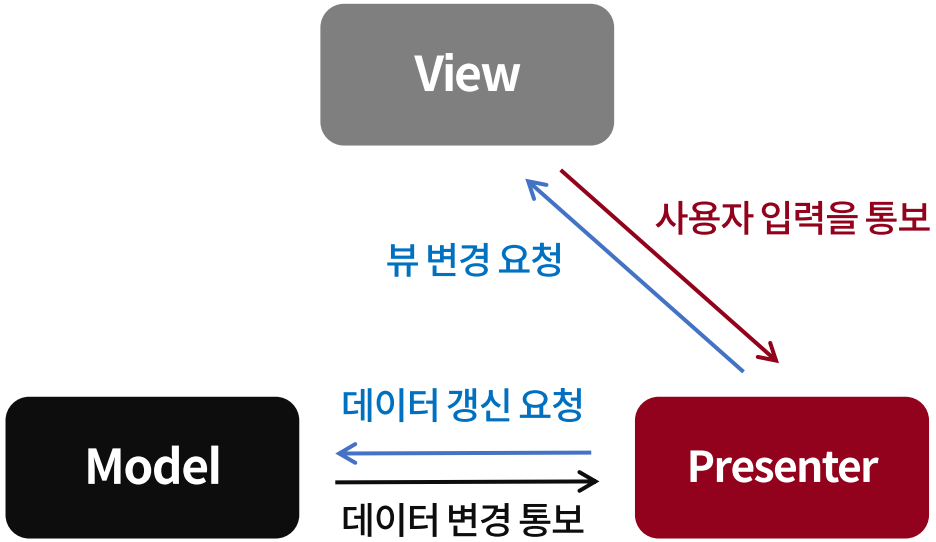
→ 참조: GUI Architectures https://martinfowler.com/eaaDev/uiArchs.html

사실 안드로이드 초기부터 구전으로 전해져 왔던 아키텍처

뷰는 비즈니스 로직에 관련된 부분을 관여하지 못하도록 분리,
Presenter로 넘김

핵심 아이디어

- 모든 UI 관련 비즈니스 로직을 프리젠터에서 처리
- 뷰는 프리젠터 요청에 따라 수동적으로 UI를 처리함
- 개방-폐쇄의 원칙
  : 뷰보다 더 높은 수준의 요소인 프리젠터를 뷰의 변경으로부터 보호함
  - ✓ 어떻게? 뷰는 반드시 interface로만 프리젠터를 참조
  - ✓ 여기에 의존성 역전의 원칙을 적용하면 뷰와 프리젠터 서로가 인터페이스로만 각자를 참조하도록 가능



```
interface TasksContract {
  interface View : BaseView<Presenter> {
    fun setLoadingIndicator(active: Boolean)
    fun showTasks(tasks: List<Task?>)
    fun showAddTask()
    fun showTaskDetailsUi(taskId: String)
  }

  interface Presenter : BasePresenter {
    val filtering: TasksFilterType
    fun loadTasks(forceUpdate: Boolean)
    fun addNewTask()
    fun openTaskDetails(requestedTask: Task)
    fun completeTask(completedTask: Task)
    fun activateTask(activeTask: Task)
    fun clearCompletedTasks()
  }
}
```

- Todo 앱의 각 작업(task)를 보여주고 추가하는 화면을 위한 interface 정의
- Contract 패턴: 뷰와 프리젠터 간의 직접 연결을 끊음

참 좋죠?

```
interface TasksContract {
  interface View : BaseView<Presenter> {
    fun setLoadingIndicator(active: Boolean)
    fun showTasks(tasks: List<Task?>)
    fun showAddTask()
    fun showTaskDetailsUi(taskId: String)
  }

  interface Presenter : BasePresenter {
    val filtering: TasksFilterType
    fun loadTasks(forceUpdate: Boolean)
    fun addNewTask()
    fun openTaskDetails(requestedTask: Task)
    fun completeTask(completedTask: Task)
    fun activateTask(activeTask: Task)
    fun clearCompletedTasks()
  }
}
```

다시 한 번 자세히 봅시다.

- 어라? Presenter의 메소드들을 보니 데이터를 넘겨주는 것들이 있네?
- openTaskDetails()이라는 이름도 왠지 좋지 않은 예감이..

```
/// Display a grid of [Task]s. User can choose to view all, active or
completed tasks.
class TasksFragment : Fragment(), TasksContract.View {
  private lateinit var presenter: TasksContract.Presenter
  ...
  val mItemListener: TaskItemListener = object : TaskItemListener() {
    fun onTaskClick(clickedTask: Task?) {
      presenter.openTaskDetails(clickedTask)
    }
    fun onCompleteTaskClick(completedTask: Task?) {
      presenter.completeTask(completedTask)
    }
    fun onActivateTaskClick(activatedTask: Task?) {
      presenter.activateTask(activatedTask)
    }
  }
  private class TasksAdapter(
    private val tasks: List<Task>?,
    private val itemListener: TaskItemListener
  ) : BaseAdapter() { ... }
}
```

“

잠시 Pause!
뷰 쪽의 구현 부분입니다.
어떤 것이 좋지 않은 지 찾아보세요.

```
/// Display a grid of [Task]s. User can choose to view all, active or
completed tasks.
class TasksFragment : Fragment(), TasksContract.View {
  private lateinit var presenter: TasksContract.Presenter
  ...
  val mItemListener: TaskItemListener = object : TaskItemListener() {
    fun onTaskClick(clickedTask: Task?) {
      presenter.openTaskDetails(clickedTask)
    }
    fun onCompleteTaskClick(completedTask: Task?) {
      presenter.completeTask(completedTask)
    }
    fun onActivateTaskClick(activatedTask: Task?) {
      presenter.activateTask(activatedTask)
    }
  }
  private class TasksAdapter(
    private val tasks: List<Task>?,
    private val itemListener: TaskItemListener
  ) : BaseAdapter() { ... }
}
```

- openTaskDetails() 함수라니.. 프리젠터 로직이 너무 구체적이군요. 이러면 나중에 프리젠터가 변경되면 어쩌려고? → 개방-폐쇄 원칙 위반
- 리스트 아이템의 각 상태에 대해 어떤 비즈니스 로직이 실행되어야 하는지를 뷰의 아주 작은 요소가 이해하고 있군요. → 뷰가 그리 수동적이지 않음
- 그리고, 아까 Fragment는 뷰가 아니라고 하지 않았던가요?

해법은?

수동적인 뷰(Passive View): 뷰는 완전한 험블 객체(Humble Object)여야 함

- 이런 문제 때문에 마틴 파울러는 프리젠터라는 이름을 폐기하고 Supervising Controller라고 부를 것을 제안하기도 함
- 참고: Retirement note for Model View Presenter Pattern
  https://martinfowler.com/eaaDev/ModelViewPresenter.html
- 수동적인 뷰는 (플랫폼 의존적인) 그리기 이외에는 아무것도 알 수 없도록 함

수동적인 뷰는 이벤트 처리를 어떻게 이해해야할까?

- "[완료] 버튼이 눌려졌다" 이상의 의미는 감춰져야 함
- 좋은 대안: 이벤트 핸들러를 프리젠터에서 구현, 그 후에 UI State에 데이터와 함께 이를 람다(lambda) 함수 형태로 전달
  (이는 Jetpack Compose에서도 매우 유용한 기법)

```
data class TaskUiState(
  val title: String,
  val isCompleted: Boolean,
  val onSelectTask: () -> Unit,
  val onCompleteTask: () -> Unit,
  val onActivateTask: () -> Unit
)
class TasksPresenterImpl : TasksContract.Presenter {
  fun loadTasks(forceUpdate: Boolean) {
    ...
    view.updateTasks(
      tasks.map { task ->
        TaskUiState(
          task.title, task.isCompleted,
          onSelectTask = { ... },
          onCompleteTask = { repository.completeTask(task.id) },
          onActivateTask = { repository.activateTask(task.id) }
        )
      }
    )
...
```

> UiState 클래스의 정의는 Contract 인터페이스 안에 함께 공유

```
data class TaskUiState(
  val title: String,
  val isCompleted: Boolean,
  val onSelectTask: () -> Unit,
  val onCompleteTask: () -> Unit,
  val onActivateTask: () -> Unit
)
class TasksPresenterImpl : TasksContract.Presenter {
  fun loadTasks(forceUpdate: Boolean) {
    ...
    view.updateTasks(
      tasks.map { task ->
        TaskUiState(
          task.title, task.isCompleted,
          onSelectTask = { ... },
          onCompleteTask = { repository.completeTask(task.id) },
          onActivateTask = { repository.activateTask(task.id) }
        )
      }
    )
...
```

TaskUiState 안에 이벤트 핸들러가 구현되어 있으므로, 뷰는 각 람다 함수를 연결만 해주면 됨
→ 향후 이벤트 발생 시 비즈니스 로직이 변경된다고 해도 뷰는 전혀 영향을 받지 않음

```
data class TaskUiState(
  val title: String,
  val isCompleted: Boolean,
  val onSelectTask: () -> Unit,
  val onCompleteTask: () -> Unit,
  val onActivateTask: () -> Unit
)
class TasksPresenterImpl : TasksContract.Presenter {
  fun loadTasks(forceUpdate: Boolean) {
    ...
    view.updateTasks(
      tasks.map { task ->
        TaskUiState(
          task.title, task.isCompleted,
          onSelectTask = { ... },
          onCompleteTask = { repository.completeTask(task.id) },
          onActivateTask = { repository.activateTask(task.id) }
        )
      }
    )
...
```

인터페이스 분리의 원칙:
뷰의 리스트 어댑터는 자신이 받은 UiState 이외의 것을 알 필요도, 알 수도 없음

MVP의 장점

Fat Activity / Fragment 방지

수동적인 뷰는 이벤트 처리를 어떻게 이해해야 할까?

→ 뷰는 화면이 액터, 프리젠터는 사용자가 액터

테스트 가능성 증대

→ 플랫폼 의존적인 UI 처리는 뷰에 분리되었기 때문에,
프리젠터는 JVM 내에서 테스트가 가능해짐

non-MVC 구조 중 비교적 완만한 학습 곡선

- 타 구조에 비해 직관적
- 코루틴이나 RxJava 등의 반응형 라이브러리가 반드시 필요하지는 않음

MVP의 단점

프리젠터는 뷰의 기능을 기본적으로 메소드 호출을 통해서 호출한다

즉, 뷰의 기능이 명령형(imperative)일 때 잘 동작함

→ Jetpack Compose와 같은 선언형(declarative) 뷰에는 적용하기 까다로움
(추가 UI State 처리를 위한 중재자가 필요)

기본적으로 뷰가 프리젠터를 참조하고 있음

→ 개념적 상호 참조
→ 잘못된 프리젠터 인터페이스 사용을 완전히 막기는 어려움

완전히 수동적인 뷰를 구현하려면 많은 고민이 필요함

수동적인 부분이 부족한 뷰 인터페이스를 설계할 경우, 리스코프 치환 원칙을 위반하지 않으려면 뷰 구현이 필요 이상으로 많은 것을 생각해야 함 (테스트도 의외로 쉽게 만들기 어려울 수도 있음)
→ 마틴 파울러의 제안에도 이런 배경이 있었음

MVP(그리고 non-MVC)에서의 주의사항

안티패턴: 비동기 처리의 결과를 동기적으로(보통 함수 리턴 값으로) 받는다

- Jank 혹은 ANR을 유발할 수 있음
- 실행 중에 라이프사이클이 변경되면 앱이 크래시 될 수 있다

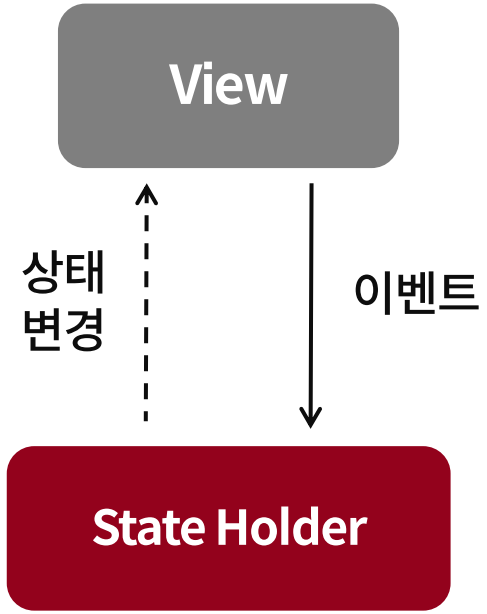
설정의 변경 (Configuration Change)

- 설정 변경은 화면 회전 때만 일어나지 않음
  - 다크모드 변경, 화면 사이즈 변경, 폰트 변경, 언어 설정 변경 등
- 반드시 던져봐야 할 질문: 서버에서 로딩 이벤트를 요청했는데, 도중에 설정 변경이 일어난다면?
- 그리고 프로세스 종료도..
  → SavedInstanceState 등을 이용해 해결 가능

MVVM (Model-View-ViewModel).

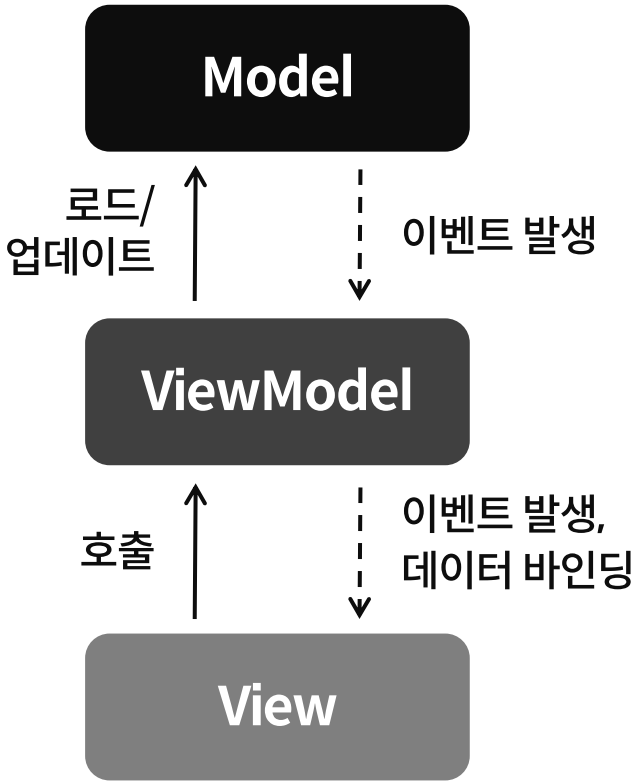
단방향 데이터 흐름 (Unidirectional Data Flow)

- 뷰를 완벽하게 수동적으로 만들기 위한 구조
- UDF의 원칙
  - 상위 객체(Higher scoped objects)는 하위 객체로부터 상태를 읽을 수 없어야 한다
  - 하위 객체(Lower scoped objects)는 상위 객체의 이벤트를 직접 읽을 수 없고, 상위 객체도 하위 객체로 직접 이벤트를 보낼 수도 없다
- 상태를 관리하는 객체(State Holder)는 뷰 방향으로의 의존성을 전혀 갖지 않는다
- 뷰는 자신이 받은 사용자 입력을 전달만 할 뿐 직접적으로 결과를 받지 않는다. 대신 간접적으로 이벤트 형태로 상태 변경을 통보 받는다



M-V-ViewModel?

- 역시 마틴 파울러의 Presentation Domain Separation (PDS)가 시초
  - 2004년 소개된 Presentation Model이 원형
  - 참조: https://martinfowler.com/eaaDev/PresentationModel.html
  - Microsoft에서 처음 구현함
- ViewModel: Uni-directional Mediator
  - 프리젠터와 매우 유사하나, 뷰모델은 뷰를 제어하지 않음
  - 뷰는 뷰모델을 호출하되, 결과는 Callback / Observable 형태로 받음



MVVM의 주요 특징

- **뷰모델은 그 자체로 독립적인 (self-contained) 시스템**
  - 결과를 이벤트로 보낼뿐 수신인의 존재를 모름
- **그리고 중요한 추가 사항**
  - Data binding! - 이벤트를 뷰로 바로 적용할 수 있는 매커니즘을 제공

```
<CheckBox
    android:id="@+id/rememberMeCheckBox"
    android:checked="@{viewmodel.rememberMe}"
    android:onCheckedChanged="@{(v, checked) -> viewmodel.rememberMeChanged(checked)}" />
```

데이터 바인딩

- **데이터 바인딩이 없다면? → 순환 이벤트 흐름**
  - 뷰 이벤트 발생 → 뷰모델이 처리 후, 이벤트로 알려줌 → 뷰가 다시 이를 받음
  - 또 다른 예: 숫자 카운터에서 1을 증가 시켰다면?

```
override fun onViewCreated(...) {
  ...
  viewModel.rememberMe.observe(viewLifeCycleOwner) {
    binding.rememberMeCheckBox.isChecked = it
  }
  viewModel.rememberMeCheckBox.setOnCheckedChangeListener { _, isChecked ->
    viewModel.rememberMeChanged(isChecked)
  }
}
```

MVVM의 단점

학습 곡선

- 이벤트는 기본적으로 반응형(Reactive)로 받아야 하므로, 이 개념에 대한 숙지가 필요
- RxJava 혹은 Coroutines 라이브러리 학습이 필요

Fat ViewModel이 될 위험성

→ 큰 크기의 모델을 피하기 어려움

순환 이벤트 흐름을 완전히 막기 어려움

→ 또 다른 예2: [다음 화면]으로 버튼을 눌렀을 때의 네비게이션 처리?

여기서 의문

“

왜 구글은 (배우기도 어렵고 RxJava / Coroutine이 없으면 제대로 구현도 안 되는) ViewModel만을 AAC(Android Architecture Component)에서 지원하는가?

- 기존 Activity/Fragment의 형태를 깨뜨리지 않으면서 재사용성/생산성 높은 아키텍처를 구현 가능하기 때문
- 안드로이드의 생존주기 처리에 가장 잘 어울리는 구조

AAC ViewModel에서 제공하는 것

“

생명주기 내에서 설정 변경과 프로세스 종료가 일어나도 ViewModel의 내용이 보존되는 구조를 제공

- 일반 변수 (그리고 Flow, LiveData, Compose State)가 보존 될 수 있도록 하는 API 제공 - SavedStateHandle
- Coroutines가 ViewModel의 생명주기 내에서 동작될 수 있도록 하는 환경을 제공

Fat ViewModel의 해결책

“

뷰를 최대한 분리,
그리고 각 뷰마다 뷰모델도 역시 함께 분리

뷰모델의 비즈니스 로직 중 도메인 로직에 해당하는 부분은
분리하여 도메인 계층으로 이동

”

ViewModel Antipatterns

상태가 유실 될 수 있는 공급자를 통해 이벤트를 전달한다

→ Channel, Flowable 등의 API는 데이터 전달을 완전히 보장하지 않음

UI 코드에서 액션까지 구현한다

→ UI는 현재 상태가 무엇이고, 어떻게 반영할지만 결정. 비즈니스 로직, 네비게이션 등은 VM의 영역

1회성 이벤트를 즉시 처리하지 않는다

→ Channel.send(), MutableLiveData.postValue() 등의 API는 낮은 우선순위로 실행될 수 있음

UI 상태 처리에 Compose의 State 대신 StateFlow / SharedFlow를 이용한다

→ 불필요한 오버헤드 발생

퀴즈: 아래 코드에서 문제점을 찾아 보시오

```
@Composable
fun MyComposable(
  viewModel: MyViewModel = hiltViewModel()
) {
  Text(text = viewModel.myState)
}

class MyViewModel : ViewModel() {
  var myState by mutableStateOf("A")
    private set

  init {
    viewModelScope.launch(Dispatchers.IO) {
        myState = "B"
    }
  }
 }
```

ViewModel 생성자에서 초기화 이상의 일을 하는 것은 anti pattern! (리디자인 필요)

- 사실 ViewModel 뿐 아니라 어떤 종류의 클래스라도 동일함
  - 단일 책임 원칙 위반
  - 테스크 구현이 어려움
  - 상속시 자식에게 과도한 정보의 습득을 요구
  - 사용자에게 필요없는 초기화를 수행하지 않게 만들 선택권을 박탈
- 생성자에서 하면 좋지 않은 일: 다른 객체의 생성, 정적 메소드 호출, 필드에 단순 대입 이상의 초기화 로직 구현, if/when을 이용한 분기 로직
- 네?! 그러면 init() 함수 같은 걸 만들라는 건가요? 1장에서 복잡성은 아래로 내리라고 하지 않았나요? → 게으른(lazy) 초기화를 구현하면 됨. 자세한 것은 실습 코너에서..

MVI (Model-View-Intent).

MVI - 출발점

“

모바일 앱에서 가장 큰 난제 중의 하나는
복잡한 상태들을 어떻게 깔끔하고 유지보수성 높게 구현할 것인가의 문제

발상을 바꿔서 앱 전체의 이벤트를
State Machine으로 처리한다면 어떨까?

- Life Cycle 처리로 골머리 앓을 일이 급격히 줄어듦
- 상태에 따라 다르게 동작하는 UI를 보다 직관적으로 관리 가능
- 특정화면에서 수정 된 내용 (예: “좋아요” 상태 변경)이 다른 곳에서 저절로 적용

“

그런데…

State Machine을 직접 구현하다 보면.. 각 로직의 추상화를 어떻게 할 지 많은 고민을 하게 됨

상태	먼저 상태를 저장하는 부분은 보통 enum class 아니면, sealed class로 구현
상태 천이 (transition/flow)	어떤 조건에서는 B로, 또 다른 조건에서는 C로 변경
로직 수행	각 상태가 변경 되었을 때 수행해야할 실제 로직

위 세 요소의 복잡도에 따라 상황에 맞게 구현해야하지만 가장 범용적인 방법 중 하나는
uni-directional한 pipeline 형태 → 바로 MVI 구조!

MVI? Flux? Redux?

MVI	단방향의 이벤트 처리 파이프라인을 가지는 아키텍처를 부르는 넓은 의미의 용어
Flux	Facebook에서, 웹 프론트엔드의 상태 처리를 직관적이고 깔끔하게 하기 위해 고안한 Architectural Pattern. 단방향의 pipelined loop를 통해 이벤트를 처리
Redux	• MVI 패턴의 변형 구현체 중 가장 효율적이라고 알려진 구현체. (단, Java Script로..) – immutable state, single source of truth를 실현한 단순한 구현 • Dan Abramov라는 젊은 개발자가 고안. 100여 줄에 불과한 라이브러리로 모든 MVI 구현체를 평정(했었음)

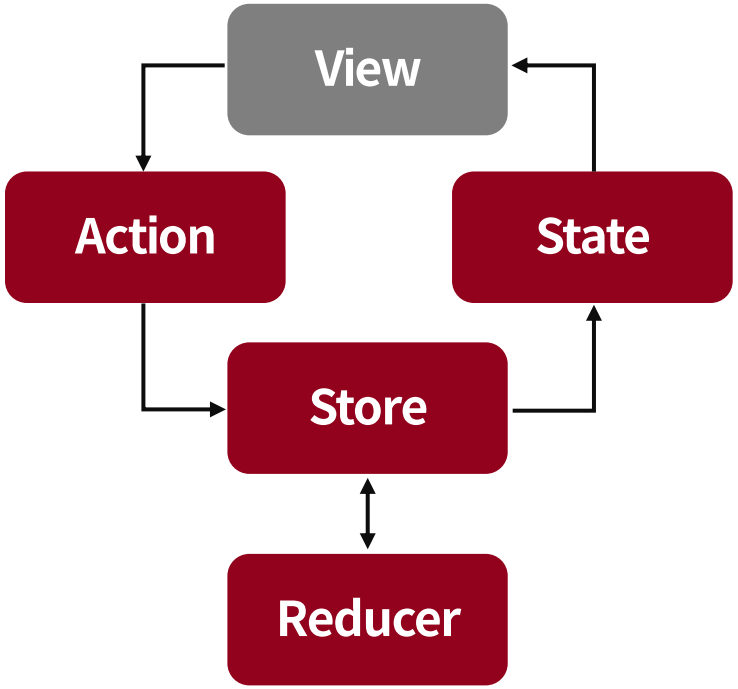
Redux

Action
- 상태를 변경하라는 명령만을 가짐
- 보통은 Action Creator를 통해 만들어짐

State
- 상태의 데이터만을 가짐. no logic
- immutable (flux는 mutable)

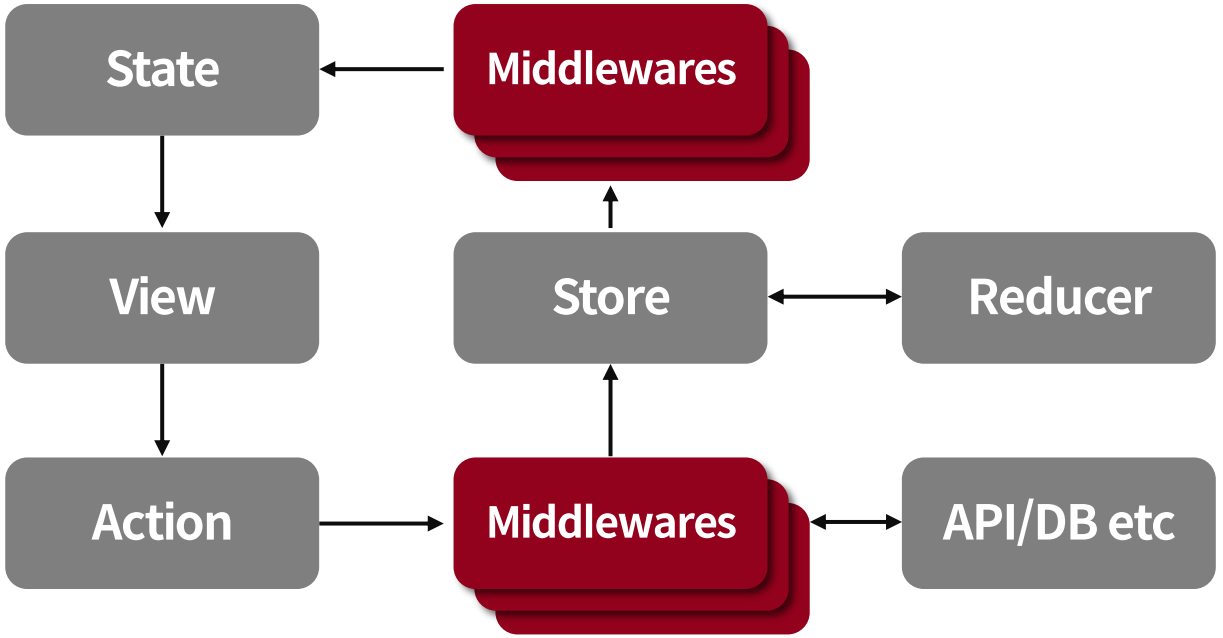
Store
- 상태의 저장소
- 상태를 변경시키고, 통지 하는 역할
- RxJava 혹은 Coroutine으로 손쉽게 구현 가능

Reducer
- 뷰가 바로 사용할 수 있는 형태의 상태로 만들어 줌
- 순수 함수(pure function) 형태로 상태 변경을 구현: (State, Action) -> new State



그럼 side effect 처리는 대체 누가?

Middleware!



Redux의 장단점

장점

- 상태 처리에서의 라이프 사이클의 영향을 어느 정도 배제할 수 있음
- 화면에서 back 버튼 누른 후에 뒤늦게 발생한 API 에러.. 같은 애매하고 난해한 상황도 효과적으로 대처 가능
- 상태의 정의 부분, 저장 부분, 처리 부분이 완벽히 나눠져 있기 때문에 수정/확장이 극도로 편함
- Functional한 구조의 특성상 유닛 테스트를 작성하는 것이 극도로 편리함
- Debugging의 신세계! 현재의 모든 전역적 상태를 하나의 Store에서 확인할 수 있고, 상태 변수 값을 직접 수정해서 전체 상태를 바꿔볼 수도 있음

단점

- 상당한 수준의 학습 비용이 소요
- 코드량 증가, 특히 boiler plate 코드의 반복적 생산은 피하기 어려움

Redux가 (웹 프론트엔드에서) 성공한 이유?

이벤트 타입과 이벤트 핸들러를 같은 타입처럼 쓰는 트릭이 가능
- JavaScript의 특성을 활용

- Time-machine Debugging - 상태를 뒤로 돌리는 등의 조작이 가능
- 마법 같은 테스트 작성 - 각 상태의 처리 파이프라인에 대한 테스트 작성이 매우 쉽고, 타 컴포넌트로부터 독립적임

그러나..

- TypeScript를 포함해 정적 타입 언어에서는 누릴 수 없는 특성
- 웹 프론트엔드에서 다양한 MVI 라이브러리들이 쏟아지고 있지만 js-Redux 만큼 압도적인 사용성을 주지 못하고 있음

MVI 도입 시 유의해야할 점

이벤트의 취소는 어떻게 할 수 있는가?

- 사용자 입력이 대량 발생하는 경우:
  예. pull to refresh를 연달아 시도, 자동 입력 완성으로 검색어 추천
- 이벤트에 대한 처리가 끝나지 않은 상태에서 다른 중요 이벤트 발생
  - 서버에서 리스트 로딩 중에 상세 화면으로 이동

그럼 반대로.. MVVM에선 괜찮나요?

- 예!
  - debounce()
  - switchMap(), flatMapLatest()
- 사실 왼쪽과 같이 Flow (Observable) 의 오퍼레이션 체인으로 MVI의 상태 파이프라인과 똑같은 기능 구현 가능

```
val feedState: StateFlow<ForYouFeedUiState> =
  newsRepository.getNewsResourcesStream()
    .filterNot { it.isEmpty() }
    .map { newsResources ->
      ...
    }
    .onStart { emit(ForYouFeedUiState.Loading) }
    .flatMapLatest { it }
    .stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5_000)
,
        initialValue = ForYouFeedUiState.Loading
    )
```

수고하셨습니다!