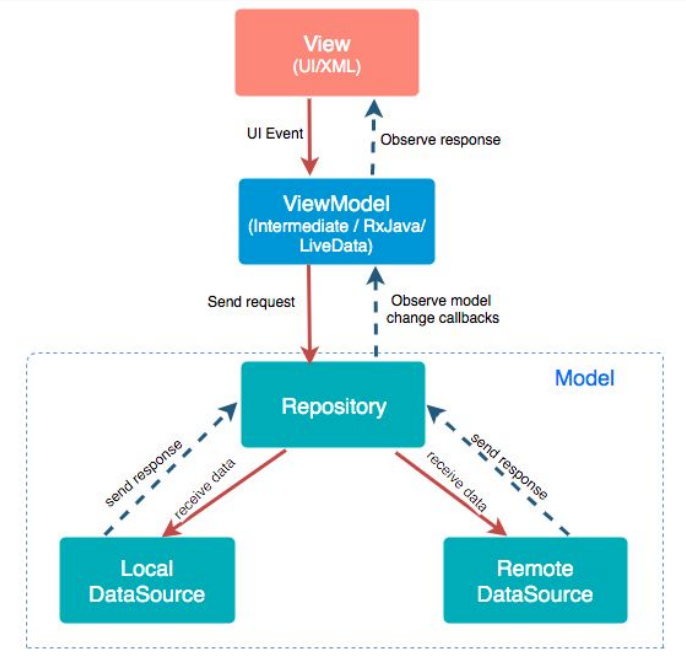
MVVM

MVVM??

MVVM

MVVM 이란?



View - ViewModel - Model 구조를 가지고 있다.

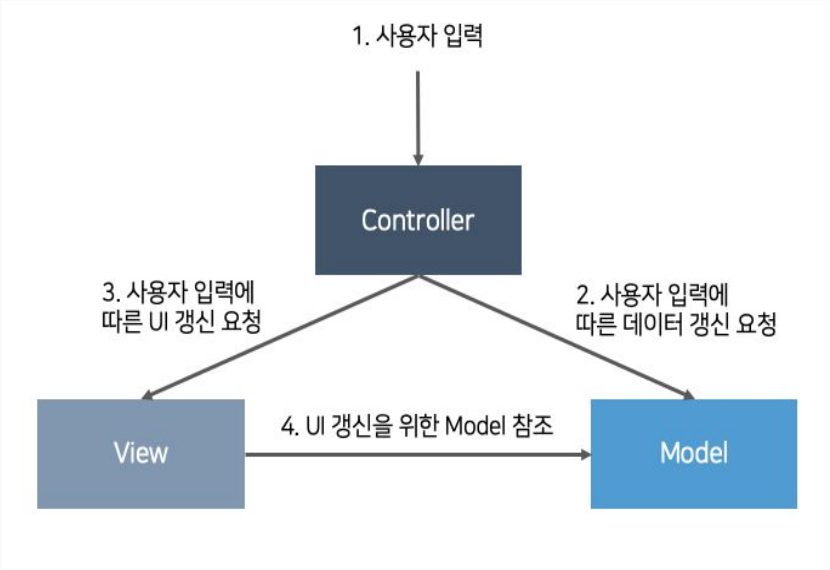
1. View (Activity, Fragment, XML, Compose)
   a. View의 Listener(Event)를 받는다.
   b. ViewModel의 데이터를 관찰하여 View 갱신
2. ViewModel
   a. View의 상태 값, 이벤트 등을 관찰 가능하게 세팅
   b. View에서 요청한 데이터를 Model에 요청
3. Model (DB : Room, Realm | API : Retrofit2)
   a. ViewModel에서 요청한 값을 반환

※ Model을 Data로 바라보는 관점과 Data + Logic으로 바라보는 관점이 있다.

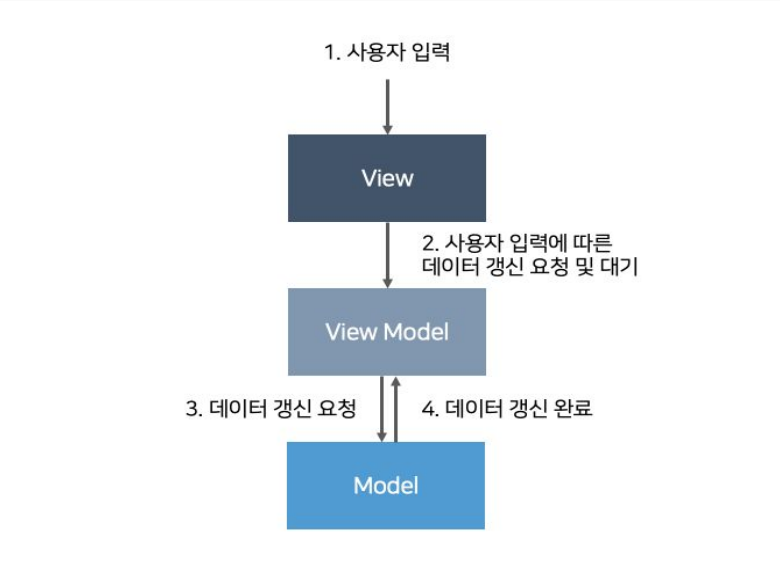
※ N:M의 관계

MVC vs MVVM

- MVC



- MVVM



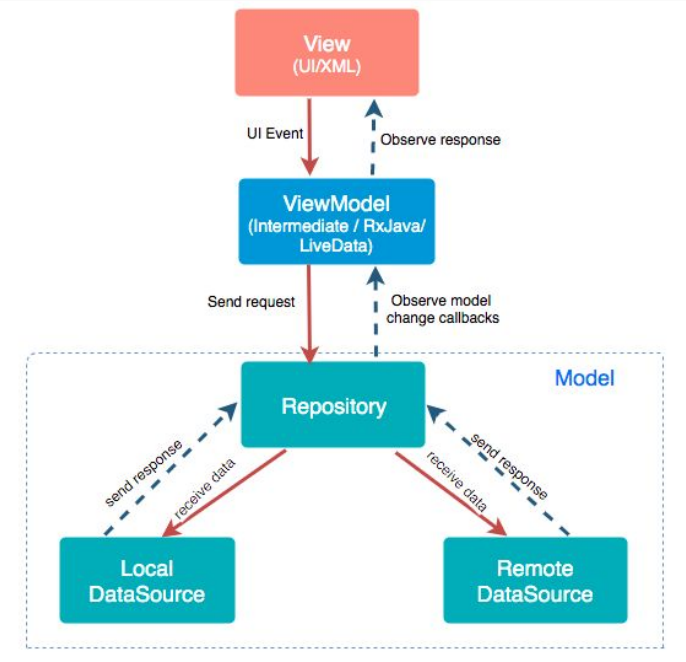
MVP vs MVVM

- MVP

```
interface ViewInterface {
    fun onBtnClicked()

    fun invalidateRecords()
}

class View: ViewInterface {
    val presenter: PresenterInterface = Presenter( view: this)

    override fun onBtnClicked() {
        presenter.getRecords()
    }
    override fun invalidateRecords() {
        // ui 꾸미기
    }
}
```

```
interface PresenterInterface {
    fun getRecords()
}

class Presenter(val view: ViewInterface): PresenterInterface {
    override fun getRecords() {
        // Records 가지고 오고
        view.invalidateRecords()
    }
}
```

MVVM 장점, 단점

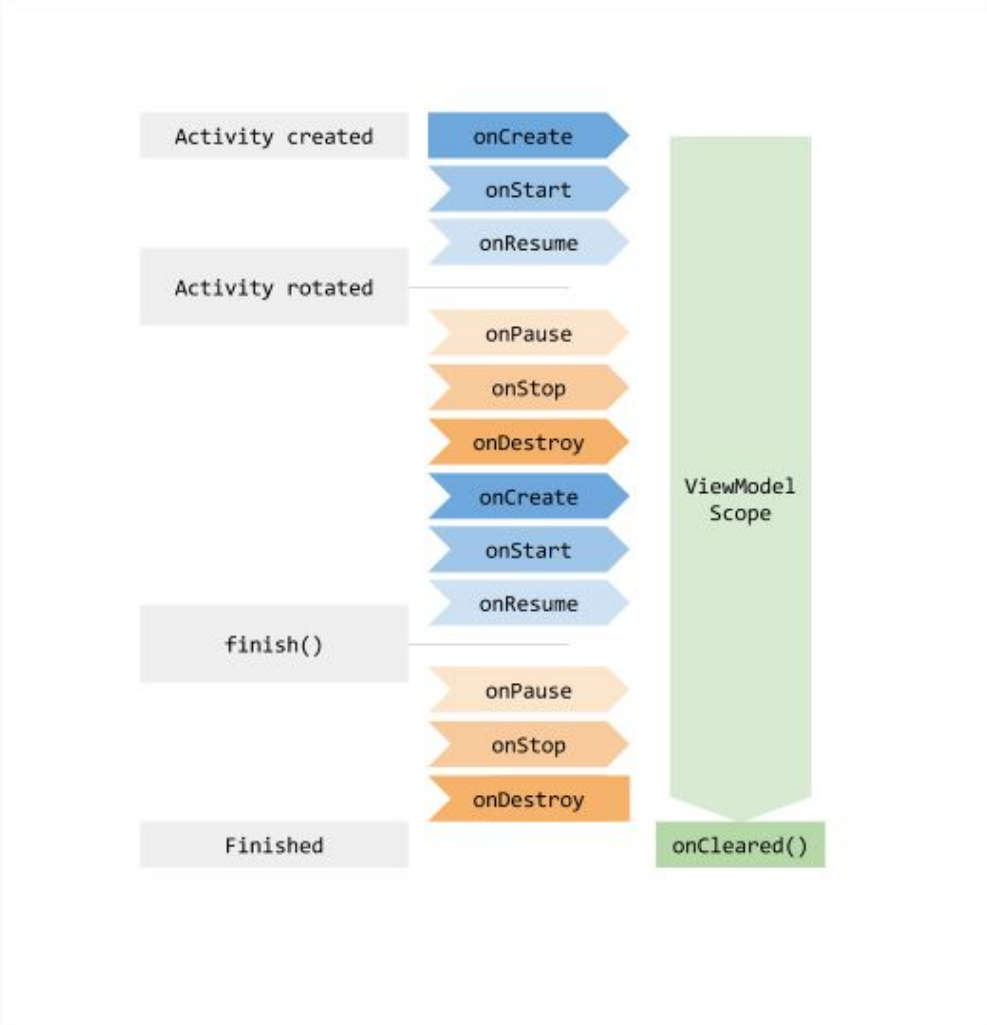


1. 장점
   a. View - ViewModel - Model 간의 의존성이 없음
   b. 중복 코드를 모듈화 할 수 있음
   c. View가 ViewModel의 Data를 관찰하고 있으므로 View 업데이트가 간편하다
2. 단점
   a. ViewModel을 잘 설계해야 한다. (쉽지 않음)
   b. UI Event, Observer pattern으로 잘 설계해야 한다.

AAC ViewModel

MVVM의 ViewModel과의 차이

1. MVVM ViewModel : View에서 필요한 데이터와 비즈니스 로직을 담당
2. AAC ViewModel : 생명주기를 고려해 데이터를 저장하는 용도



UI(View)

View Binding

- findById를 대체한다

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.textView.text = "Test"
```

Data Binding

- XML 파일에서 다뤄보자

1. ViewModel

```
private val _testText = MutableStateFlow( value: "Test")
val testText = _testText.asStateFlow()
```

2. XML

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@={vm.testText}"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Data Binding

- XML 파일에서 다뤄보자

1. ViewModel

```
fun testFun() {
    viewModelScope.launch { this: CoroutineScope
        _testText.emit( value: "testFunClicked")
    }
}
```

2. XML

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@={vm.testText}"
    android:onClick="@{() -> vm.testFun()}"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Data Binding

- XML 파일에서 다뤄보자

```
<data>
    <import type="android.view.View" />
    <variable name="vm" type="dev.sincere.todoapp.MainViewModel" />
</data>
```

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@={vm.testText}"
    android:visibility="@{vm.testText == `Test` ? View.VISIBLE : View.GONE}"
    android:onClick="@{() -> vm.testFun()}"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Data Binding

- BindingAdapter에 대해 알아보자

```
@BindingAdapter( ...value: "customVisible")
fun View.visible(isVisible: Boolean) {
    visibility = if (isVisible) View.VISIBLE
    else View.GONE
}
```

```
customVisible="@{vm.testText == `Test`}"
```

View Binding
And
Data Binding

1. 공통점
   a. FindById를 대신한다.
2. DataBinding에서 추가된 기능
   a. xml 문법을 통해 binding을 할 수 있다
   b. two-way 방식을 지원한다.
   c. BindingAdapter를 통해 속성값을 생성해 binding을 할 수 있다.

DataBinding을 좀 더 고급지게 사용한다면 BindingAdapter를 통해 two-way 방식도 만들 수 있다.

이 두가지는 동시에 활용할 수 있다.

StateFlow and SharedFlow

1. Hot Stream vs Cold Stream
    a. Hot Stream (StateFlow, SharedFlow, Channel)
        i. Lazy 발행(데이터 발행 시 발행, 구독 시 데이터 발행 X)
        ii. 구독자와 상관없이 데이터를 발행한다.
        iii. 구독자가 다수
        iv. State, Event 값을 저장할 때 활용
    b. Cold Stream (Flow, 1회성?)
        i. 정해진 데이터를 바로 발행(구독 시 발행)
        ii. 구독자가 구독을 했을 경우 데이터를 발행한다
        iii. 구독자가 하나
        iv. DB, 서버 통신할 때 활용
2. 이 둘의 차이는 동작 방식에 대한 차이가 있을 뿐 Hot Stream이다.
3. StateFlow는 SharedFlow를 활용해 만든 하나의 구현체
4. 차이점(이 있지만SharedFlow를 활용해서 다양한 활용법이 존재)
    a. StateFlow
        i. 구독 시 최근 데이터도 받을 수 있음
        ii. 기존 데이터와 같은 데이터 발행 X
    b. SharedFlow
        i. 다양하게 활용할 수 있음
        ii. 구독 시 최근 데이터 받을 수 없음 (Default)

StateFlow (상태값 저장)

```
private val _stateFlow = MutableStateFlow( value: "Test")
val stateFlow = _stateFlow.asStateFlow()
```

```
lifecycleScope.launchWhenStarted { this: CoroutineScope
    viewModel.stateFlow.collect { it: String
        Log.e( tag: "State Flow : ", it)
    }
}
```

SharedFlow (Event)

```
sealed class Event {
    object RefreshEvent: Event()
    data class DeleteEvent(val data: String): Event()
}
```

```
private val _sharedFlow = MutableSharedFlow<Event>()
val sharedFlow = _sharedFlow.asSharedFlow()
```

```
lifecycleScope.launchWhenStarted {  this: CoroutineScope
    viewModel.sharedFlow.collect {  it: Event
        when (it) {
            is Event.RefreshEvent -> {

            }
            is Event.DeleteEvent -> {
                val item = it.data

            }
        }
    }
}
```

LiveData

```
private val _livedataState = MutableLiveData( value: "Test")
val livedataState: LiveData<String> = _livedataState
```

```
viewModel.livedataState.observe( owner: this) {  it: String!
    Log.e( tag: "Live Data : ", it)
}
```

LiveData vs State(Shared) Flow

1. LiveData와 Flow의 차이점
    a. LiveData
        i. Lifecycle에 의존성이 있음
        ii. 구독 시 최근 데이터도 받을 수 있음
        iii. 기존 데이터와 같은 데이터 발행 O
    b. StateFlow
        i. 구독 시 최근 데이터도 받을 수 있음
        ii. 기존 데이터와 같은 데이터 발행 X
    c. SharedFlow
        i. 구독 시 최근 데이터 받을 수 없음 (Default)
        ii. 기존 데이터와 같은 데이터 발행 O (Default)

※ LiveData보다는 Flow를 활용하자

1. MVVM패턴이 View - ViewModel - Model간의 의존성을 낮추기 위함인데 Lifecycle에 의존성이 생기므로 피해야함

2. 상태값을 저장하는 경우에는 구독 시 최근 데이터를 받을 수 있어야 하지만 그 후에 들어오는 데이터는 기존 데이터와 같은 경우 발행하지 않는게 더 좋아보인다.

3. 이벤트의 경우는 기존데이터와 같더라도 발행되어야 하지만 구독시에 받는 데이터는 없는게 좋아 보인다.

시연

질문 받도록
하겠습니다.

감사합니다.

eunmin.sincere@gmail.com