ЛЕКЦИЯ I

Android:
Android architectures

Теодор Костадинов

Съдържание

1. Откъде е дошла нуждата за архитектури?
2. MVP - Model View Presenter
3. MVVM - Model View ViewModel
4. Demo

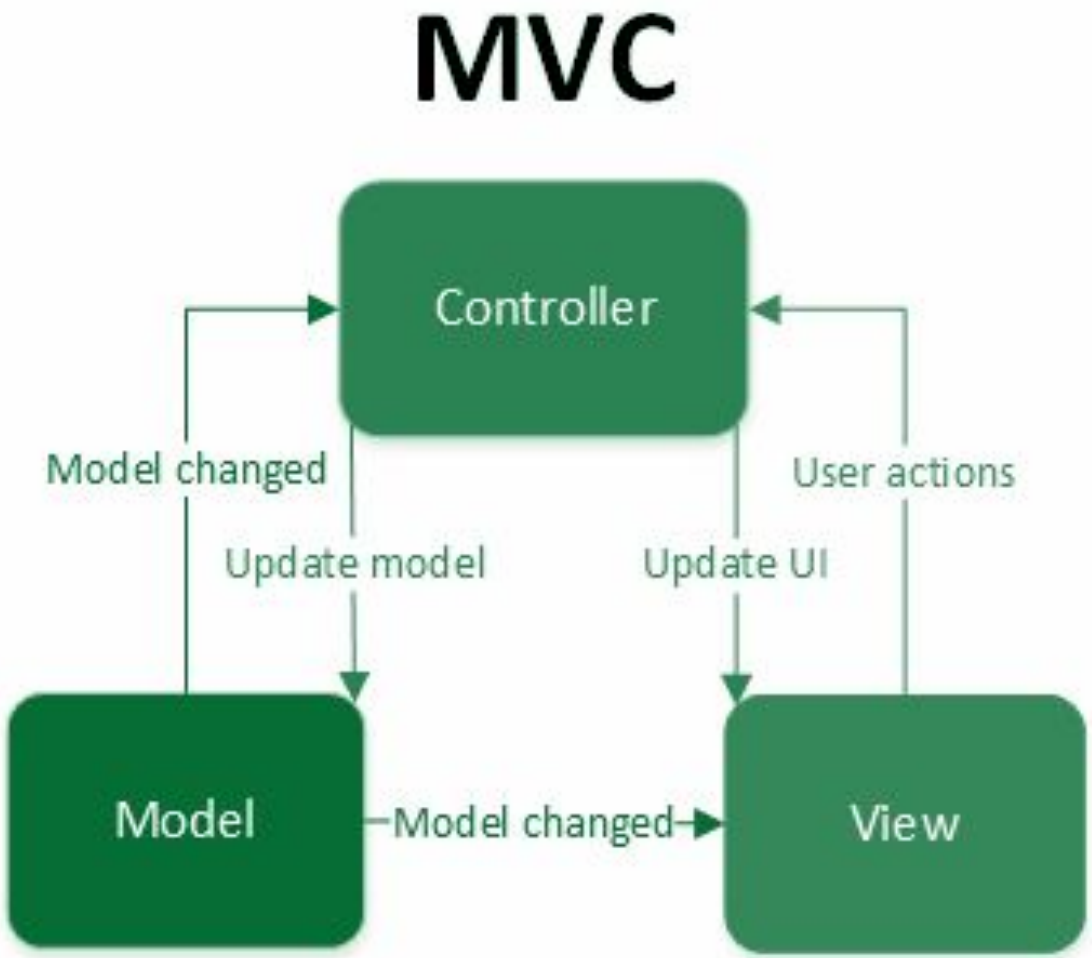
Защо са важни архитектурите?

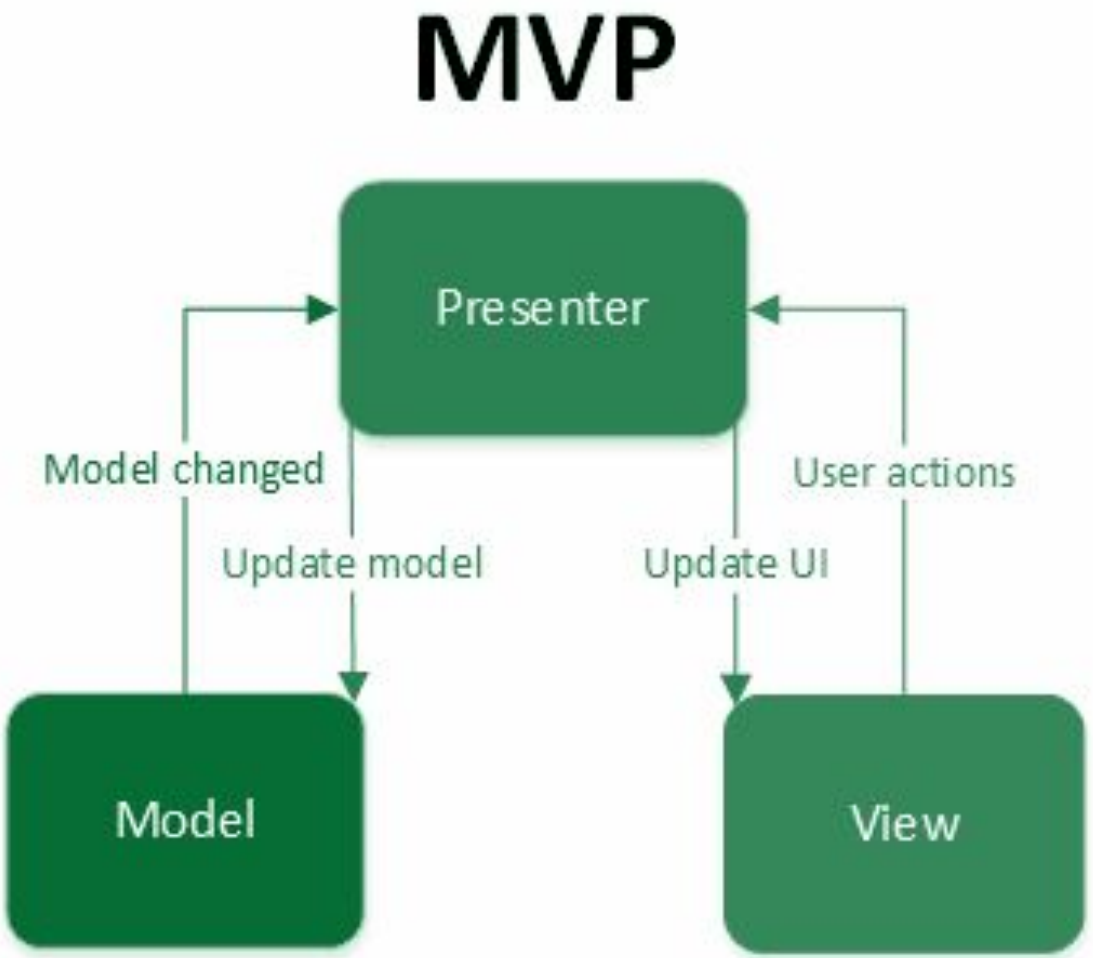
- Приложенията за Android, за разлика от настолните, имат много по-сложна структура. Типично приложение за Android съдържа множество компоненти - **activity**-та, **fragment**-и, **service**-и и други, които декларираме в нашия **manifest** файл.
- Едно правилно написаното приложение за Android съдържа набор от компоненти.
- Потребителите често взаимодействат с множество приложения едновременно за кратък период от време, затова вашите приложения трябва да се **адаптират**.

Нуждата от архитектури

- Преди да се изградят общи правила, всеки разработчик е имал свободата да изгради приложението си по удобен за него начин, което е създавало объркване при потребителите и техния **user experience**.
- **Дискусиите** по темата са започнали да се появяват **едва през 2014 г.** въпреки, че Android сам по себе си съществува от 2009 година.
- По-ранното Андроид разработване не се е справяло добре със задачата, но в момента има **инструменти и насоки**, които служат за ориентир, за да се гарантира качество.

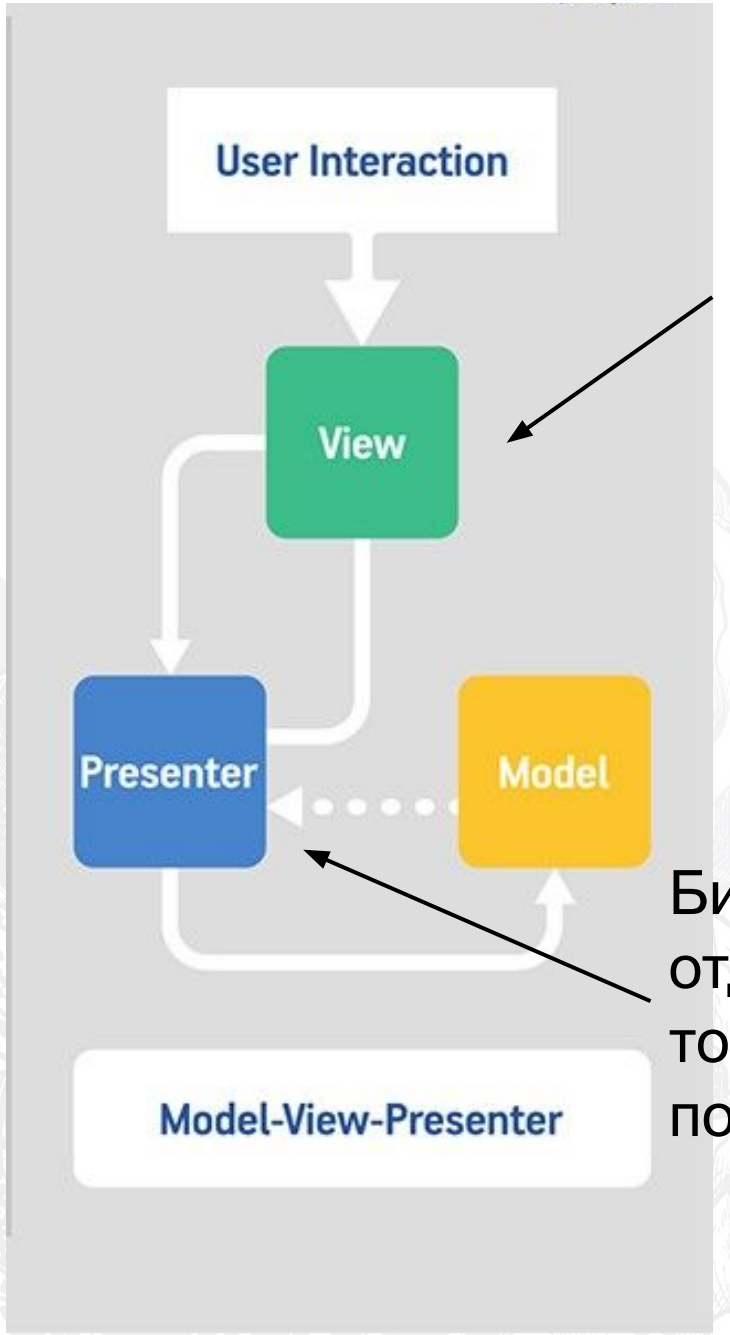
MVP


MVC
Controller
Model changed
Update model
Update UI
User actions
Model
Model changed
View


MVP
Presenter
Model changed
Update model
Update UI
User actions
Model
View

Model	View	Presenter
В приложение с добра архитектура този слой би бил само **обвивка на бизнес логиката на приложението**. Той е своеобразен доставчик на данните, които искаме да покажем в изгледа. **Отговорностите** на модела включват **използване на API, кеширане на данни, управление на бази данни** и т.н.	Обикновено е **имплементиран от Activity** и ще съдържа **референция към Presenter-а**. Единственото нещо, за което view-то се грижи, е да вика **метод от Presenter-а** при дадено взаимодействие с интерфейса от страна на потребителя.	Той е отговорен да действа като **посредник между View-то и Model-а**. Той извлича данни от Model-а и ги връща **правилно форматирани** на view-то. Но за разлика от типичния MVC, той също **решава какво ще се случи**, когато взаимодействате с view-то - тоест какво да се случи при натискането на даден бутон например.

- **Presenter-ът** съдържа **логиката**, която **управлява View-то**, то не знае нищо и **само се грижи да показва** текущото състояние
- **Всички извиквания**, отправени към View-то, **се препращат към Presenter-а**, който взима решенията
- Всяко **View има** отделен **Presenter**



Всяко view си има presenter

Бизнес логиката е отделена от view-то, а presenter-ът е посредникът

MVVM

Model	View	ViewModel
Това се отнася до **данните** (или модела на домейна), които се използват като и**зточник на информация, която ViewModel предоставя** на **View**-то.	Това е представянето на **графичния интерфейс** на кода - представени от XML файлове. Обикновено има някаква **форма на bind-ване** на данните, която **свързва View-то с ViewModel**.	**ViewModel-ът** е отговорен за **wrap-ването на Model-а** и подготовката на **данните**, които ще наблюдаваме, за чиито **промени ще следим**, които са необходими на View-то. Той също така **осигурява начин на View-то да предава данни на Model-а,** като играе ролята на посредник.

Съхранява и управлява данните по **lifecycle conscious** начин

Грижи се да **следи за промяна на данните** в бизнес модела и да променя View-то, **част от Android Architecture Components**

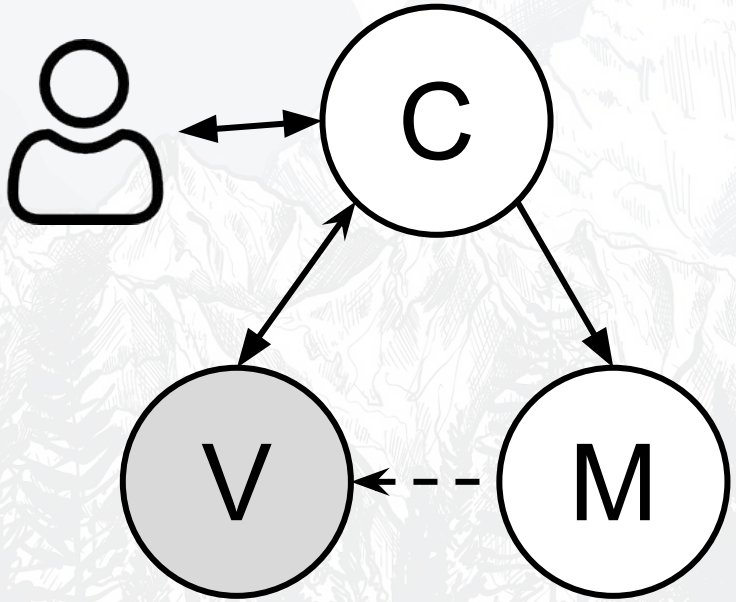
ViewModel-ите **НЕ** трябва да имат **референция** към:

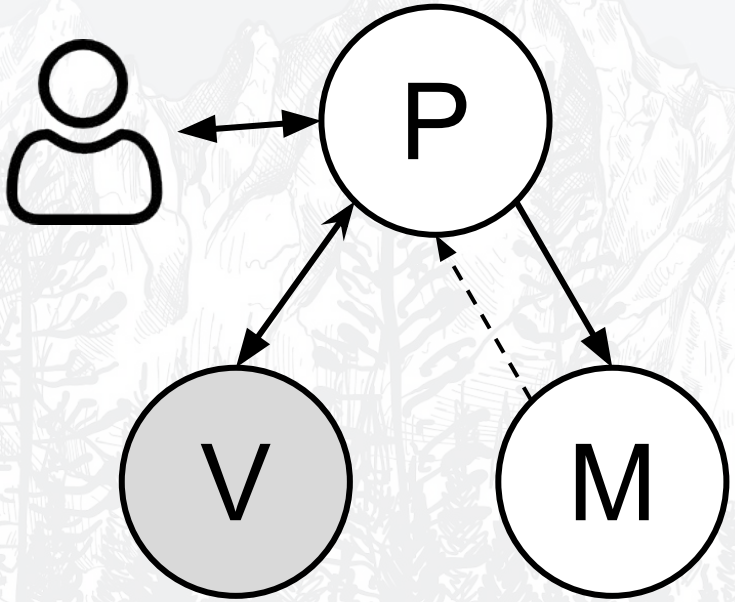
~~**Activity**~~

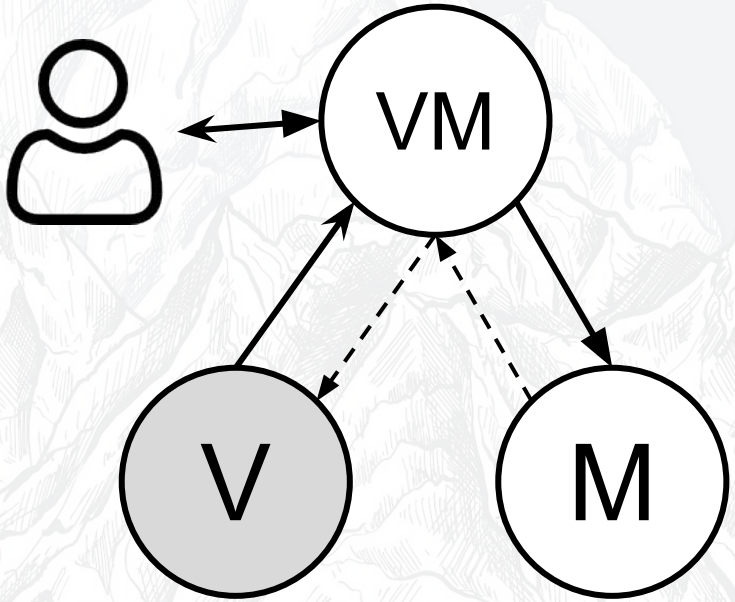
~~**Fragment**~~

~~**Context**~~

MVC	MVP	MVVM
Входната точка на приложението е **Controller**-ът	**Входната точка** на приложението е **View-то**	**Входната точка** на приложението е **View-то**
View-то няма референция към **Controller**-а	**View-то има** референция към **Presenter**-а	**View-то има** референция към **ViewModel**-а (bind-ва се)
View-то е наясно какъв е точно е **Model-ът**	**View-то не е** наясно какъв е точно е **Model-ът**	**View-то не е** наясно какъв е точно е **Model-ът**





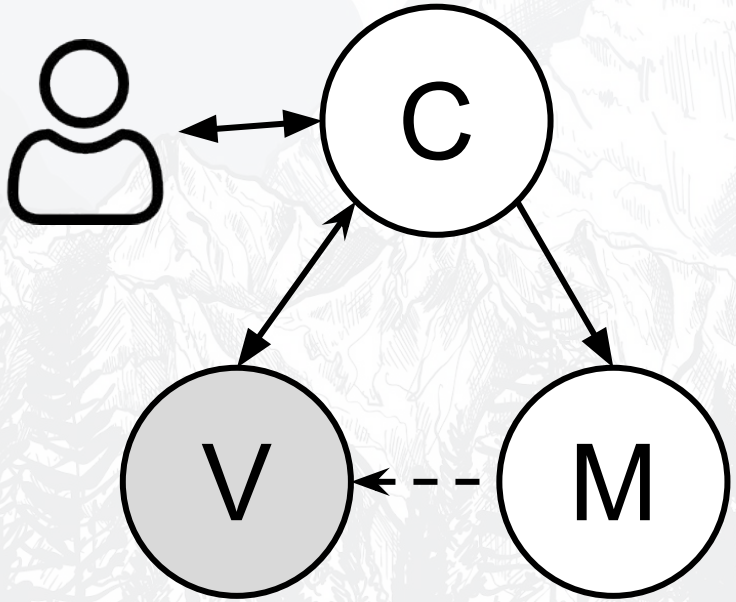


MVC

Входната точка на приложението е **Controller**-ът

View-то няма референция към **Controller**-а

View-то е наясно какъв е точно е **Model-ът**



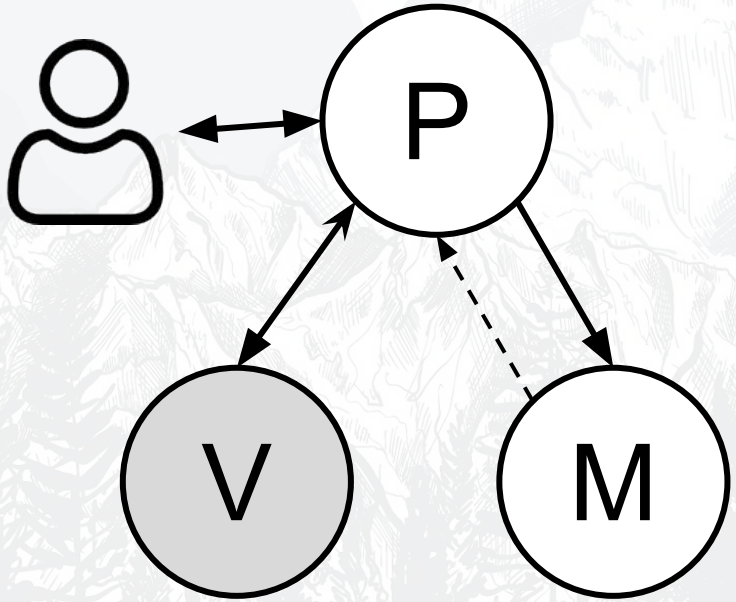
- В MVC **View-то и Controller-а** наравно се намират **на върха на нашата архитектура.**
- Model-ите се намират под Controller-а, тоест **нашите View-та знаят за нашите контролери, а пък контролерите знаят за Model-ите**.
- Тук нашите **View-та** имат **директен достъп до Model-ите**, тоест могат да си взаимодействат.
- Недостатък е, че излагането на цялата логика на приложението директно на View-то може да има **неблагоприятно влияние върху сигурността и производителността**.

MVP

Входната точка на приложението е **View-то**

View-то има референция към **Presenter**-а

View-то не е наясно какъв е точно е **Model-ът**



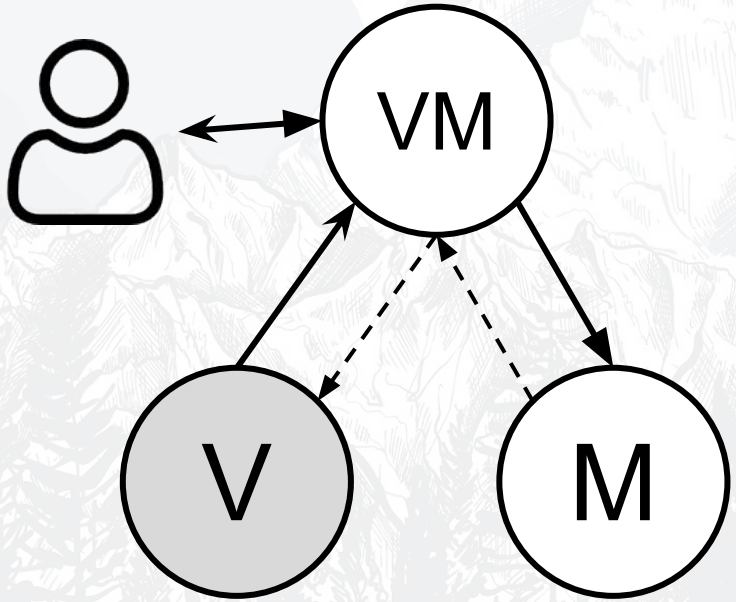
- В MVP ролята на Controller-а се заменя с Presenter.
- **Presenter-ите слушат за събития** както от потребителския интерфейс - **View-то, с което са свързани,** така и **от Model-а**.
- Разчитаме на това, че всяко View ще **имплементира интерфейс**(или няколко), позволяващ на неговия Presenter да си взаимодейства с него - тоест да използваме callbacks.

MVVM

Входната точка на приложението е **View-то**

View-то има референция към **ViewModel**-а (bind-ва се)

View-то не е наясно какъв е точно е **Model-ът**



- Последно MVVM ни позволява да създаваме **специфични** за всеки Model **ViewModel-и** (по един или повече на Model), които могат да съдържат **информация за състоянието и логиката**, **избягвайки** необходимостта да се **разкрива целият модел на View-то**.
- За разлика от Presenter-ът на MVP, **ViewModel-ът не е задължително да има референция към конкретно View**. **View-то може да се bind-ва даден ViewModel**, които от своя страна предоставя на View-то данните, съдържащи се в Model-ите. Този подход е най-труден за имплементиране и обикновено се предпочита за по-големи проекти.

View Model

View Model

ViewModel-ите съхраняват и манипулират данни свързани с UI-я на приложението.
Lifecycle aware е.
Спомага на данните да се запазят, ако сменим ориентацията на екрана (обичайно тогава се губят данните).

View Model

```
class FragmentViewModel extends {
   private LiveData<Data> data;

   public LiveData<Data> getData() { … }
ViewModel }
```

View Model

Как създаваме инстанция на ViewModel
(в активити или фрагмент):

```
private FragmentViewModel viewModel;

   ...

viewModel = new ViewModelProvider(this).get(FragmentViewModel.class);
```

Полезни материали

https://www.educba.com/mvc-vs-mvp-vs-mvvm/ - Разликите между MVC, MVP И MVVM в дълбочина

https://www.youtube.com/watch?v=orH4K6qBzvE - ViewModel explained

https://www.youtube.com/watch?v=suC0OM5gGAA - LiveData explained

https://developer.android.com/jetpack/guide?gclid=Cj0KCQjw1PSDBhDbARIsAPeTqrcbyA6yYZbhuzW5P3YZ_rPtxOQ2lLUcOVEtz96U2eXBZcycA8ui4HAaAu45EALw_wcB&gclsrc=aw.ds - Android Docs

ВЪПРОСИ?

БЛАГОДАРЯ
ЗА ВНИМАНИЕТО!