2.6. Архитектурные компоненты ViewModel и LiveData

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 2.6. Архитектурные компоненты ViewModel и LiveData

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 17:46

Оглавление

2.6.1 Введение в архитектурные компоненты

Архитектурные компоненты в Android - набор библиотек, которые помогают разрабатывать легкие в поддержке, надежные и тестируемые приложения.

Архитектурные компоненты включают множество новых классов, таких как: LifecycleObserver, LiveData, ViewModel, LifecycleOwner, а также библиотеку Room для работы с базой данных приложения.

Эти компоненты помогают разработчикам следовать надлежащему архитектурному шаблону проектирования, а также позволяют легко и безболезненно вносить изменения, избегая утечек памяти (memory leak) и обновлять пользовательский интерфейс при смене данных.

В рамках данной темы мы рассмотрим два ключевых архитектурных компонента ViewModel и LiveData.

LiveData — это наблюдаемый объект для хранения данных, он уведомляет наблюдателей, когда данные изменяются. Этот компонент также является связанным с жизненным циклом LifecycleOwner (Activity или Fragment), что помогает избежать утечек памяти и других неприятностей. Компонент LiveData — предназначен для хранения объекта и разрешает подписаться на его изменения.

ViewModel — предназначен для хранения и управления данными, связанными с пользовательским интерфейсом, с учетом жизненного цикла. Класс ViewModel позволяет данным сохраняться при изменении конфигурации, например при повороте экрана.

Перед тем как рассмотреть названные архитектурные компоненты рассмотрим шаблон шаблон проектирования MVVM.

2.6.2. Шаблон проектирования MVVM

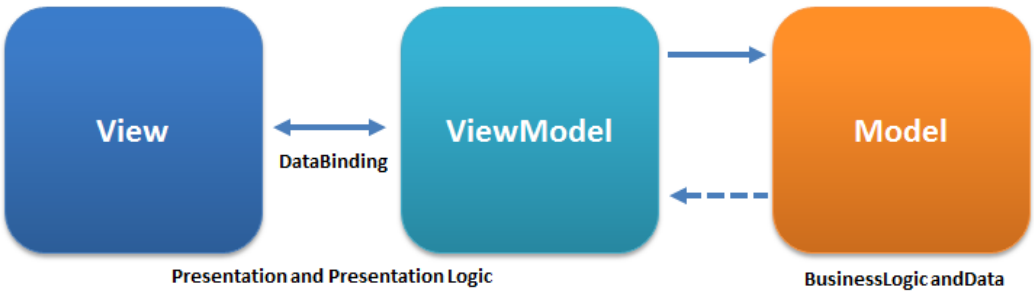
Model-View-ViewModel (MVVM) — шаблон проектирования архитектуры приложения. Предложен в 2005 году Джоном Госсманом как модификация шаблона Presentation Model. Шаблон ориентирован на современные платформы разработки, в том числе Android.

Итак, MVVM - это шаблон архитектуры клиентских приложений, который был предложен как альтернатива шаблонам MVC и MVP при использовании связывания данных (Data Binding). Его суть состоит в отделении логики представления данных от логики приложения путем вынесения последней в отдельный класс для большего разграничения.

Теперь давайте разберемся, что же значит каждая из трех частей в названии:

- **Model** - это логика, которая связанная с данными приложения. Другими словами - это POJO, классы для работы с API, базами данных и др.
- **View** - это разметка экрана (layout), в которой расположены необходимые представления для отображения данных.
- **ViewModel** - объект, в котором описана логика поведения **View** в зависимости от результата работы **Model**. В некоторой литературе этот объект называют моделью поведения **View**. Это может быть логика управления видимостью представлений, форматирование текста, отображение разнообразных состояний, таких как ошибки, загрузка и т.д. Также в ней описано поведение пользователей (свайпы, нажатия клавиш, касания и т.д.)

Логика модели представлена на следующем рисунке:



В конечном итоге мы имеем гибкость разработки и рост эффективности работы в команде, т.к. пока один разработчик описывает логику получения данных и их обработки, другой в это время работает над стилизацией экрана. Кроме этого такая структура значительно упрощает написание тестов и процесс создания mock-объектов. В большинстве случаев исчезает потребность в автоматическом UI-тестировании, поскольку можно снабдить unit-тестами сам **ViewModel**. Используя архитектуру MVVM, код становится более простым, читабельным и легким в сопровождении, поскольку каждый программный модуль имеет свое конкретное назначение.

Кроме вышеперечисленных плюсов имеются и недостатки. Например, для небольших проектов этот подход может быть неоправданным. Если логика привязки данных слишком сложная - отлаживать приложение будет труднее.

Если MVVM реализован грамотно то это хороший способ разбить код и сделать его более тестируемым, это помогает следовать принципам SOLID, поэтому код легче поддерживать на всех этапах жизненного цикла приложения.

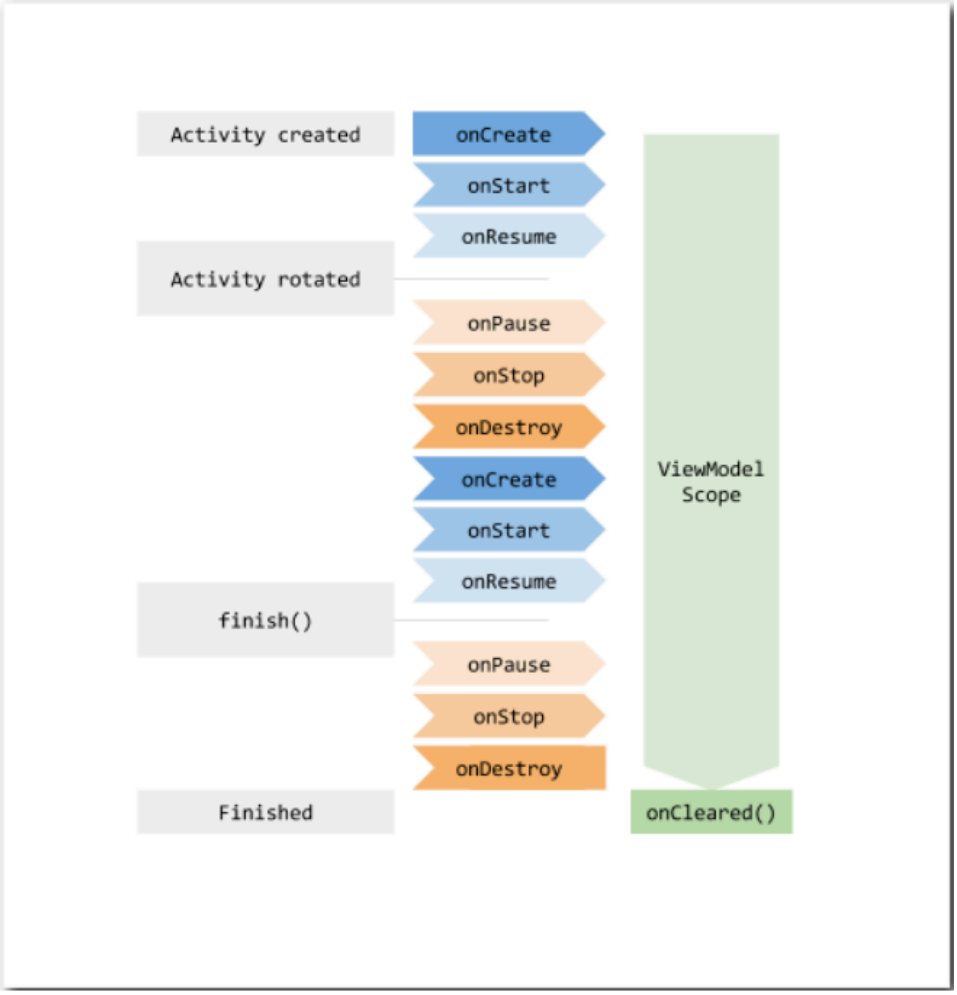
2.6.3. Класс ViewModel

Одним из ключевых компонентов шаблона проектирования MVVM выступает класс ViewModel. Его основное предназначение - хранение и управление данными, связанными с интерфейсом пользователя, с учетом жизненного цикла Активности или Фрагмента. Класс `ViewModel` позволяет данным отслеживать и адаптироваться к изменениям конфигурации устройств, таких например, как смена ориентации экрана.

> Чтобы импортировать `ViewModel` в проект Android, см. инструкции по объявлению зависимостей в заметках о выпуске Lifecycle.

Таким образом класс `ViewModel` - класс, который был создан для возможности Activity и фрагментам сохранять необходимые им объекты при повороте экрана.

Ниже приведен жизненный цикл объекта `ViewModel`



Из картинки видно, что `ViewModel` жизнеспособен, пока `Activity` окончательно не закроется .

При новом запуске `Activity` класс `ViewModel` все еще живет и задействован во вновь созданном `Activity`.

Далее в упражнении мы разберем пример с созданием счетчика и использованием класса `ViewModel`.

2.6.4. Класс LiveData

Класс LiveData - хранилище информации, работающий по принципу шаблона проектирования Observer (наблюдатель). Данное хранилище играет две ключевые роли:

- в него можно поместить какой-либо объект;
- на него можно подписаться и получать объекты, которые в него помещают.

То есть во-первых пользователь может поместить объект в хранилище, во-вторых подписанные стороны могут получить этот объект .

В качестве аналога можно привести, например, каналы в социальных сетях. Автор пишет сообщение, отправляет его , а все его подписчики получают это сообщение.

В таком виде хранения есть один большой плюс. Класс `LiveData` умеет определять активен подписчик или нет, и отправлять информацию будет только активным. Предполагается, что рассылка `LiveData` будет проводиться `Activity` и фрагментам. А их состояние активности будет определяться с помощью их жизненного цикла.

LiveData имеет ряд интересных характеристик:

- предотвращает утечку памяти, когда наблюдатель привязан к жизненному циклу;
- предотвращает сбои из-за остановки активности;
- автоматически обрабатывает жизненный цикл;
- при использовании `LiveData` код становится значительно проще для тестирования.

Приведем пример

```
private val fullname = MutableLiveData<String>()

// Called on app launch
fun initNetworkRequest() {
    // expensive operation, e.g. network request
    fullname.value = "Andriiginting"
}
fun getFullname(): LiveData<String> {
    return fullname
}
// Called on Activity creation
getFullname().observe(this, Observer { user -> Log.d(TAG, user) })
```

Давайте обсудим фрагмент кода

```
private val fullname = MutableLiveData<String>()
```

Как уже отмечалось, Kotlin строго типизирован, то есть каждая переменная имеет явный тип. Вы также можете написать:

```
val fullname: MutableLiveData<String> = MutableLiveData<String>
```

Класс `MutableLiveData` расширяет `LiveData`, с отличием в том что это не абстрактный класс и методы setValue(T) и postValue(T) публичны.

Далее рассмотрим пример совместного использования классов `ViewModel` и `LiveData`

Упражнение 2.6.

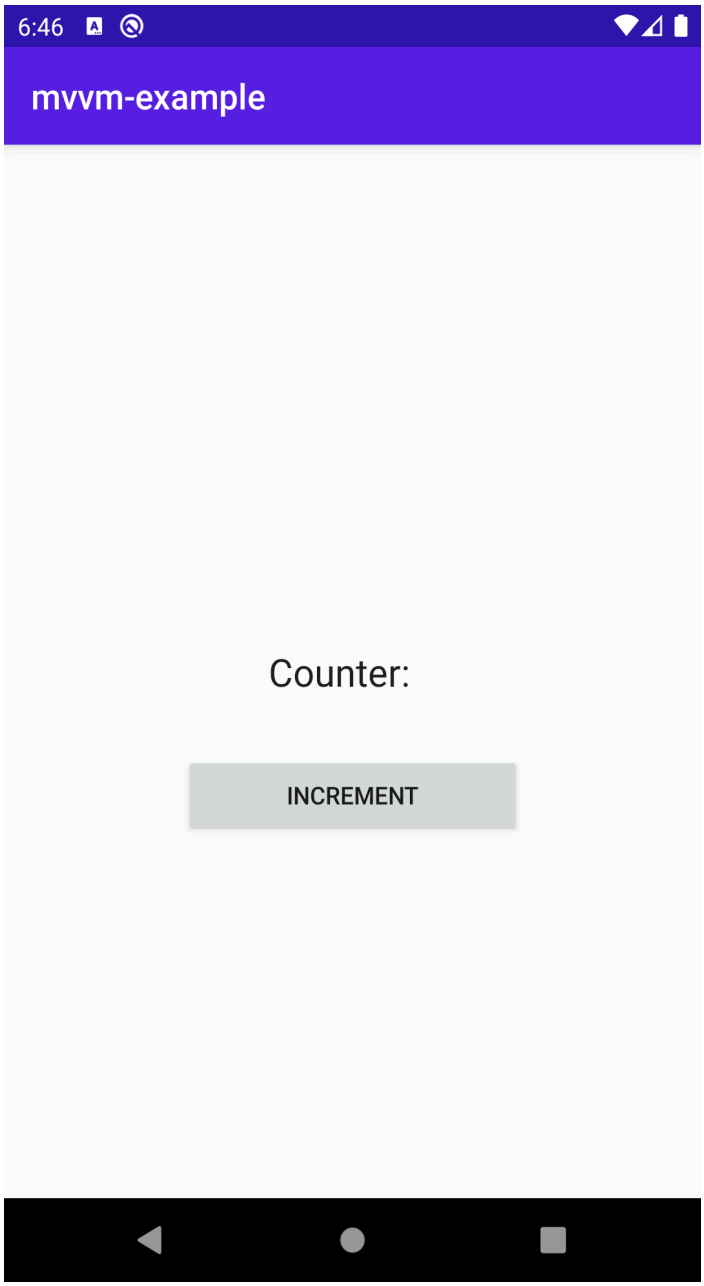
Разработаем приложение Приложение "Счетчик" с использованием архитектуры MVVM. Функционально приложение будет очень простым - по нажатию кнопки счетчик будет прибавляться.

Для начала добавим необходимые зависимости lifecycle-extensions в файл build.gradle

```
dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
    implementation "androidx.activity:activity-ktx:1.1.0"
    implementation "androidx.fragment:fragment-ktx:1.2.2"
}
```

Разметка будет состоять из одного `TextView` - ссчетчика нажатия кнопки `Button`.

```
<?xml version="1.0" encoding="UTF-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
tools:context="ru.samsung.itacademy.mdev.MainActivity"
android:layout_height="match_parent"
android:layout_width="match_parent"
xmlns:tools="http://schemas.android.com/tools"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:android="http://schemas.android.com/apk/res/android">

<TextView
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        app:layout_constraintHorizontal_chainStyle="packed"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toStartOf="@id/text_counter"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        android:layout_marginEnd="8dp"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        android:text="@string/counter" android:id="@+id/label_counter"/>

<TextView
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        tools:text="123"
        android:id="@+id/text_counter"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@id/label_counter"
        android:layout_marginStart="8dp"/>

<Button
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        app:layout_constraintStart_toStartOf="parent"
        android:text="@string/increment"
        android:id="@+id/btn_increment"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@id/label_counter"
        android:layout_marginTop="32dp" android:minWidth="200dp"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```



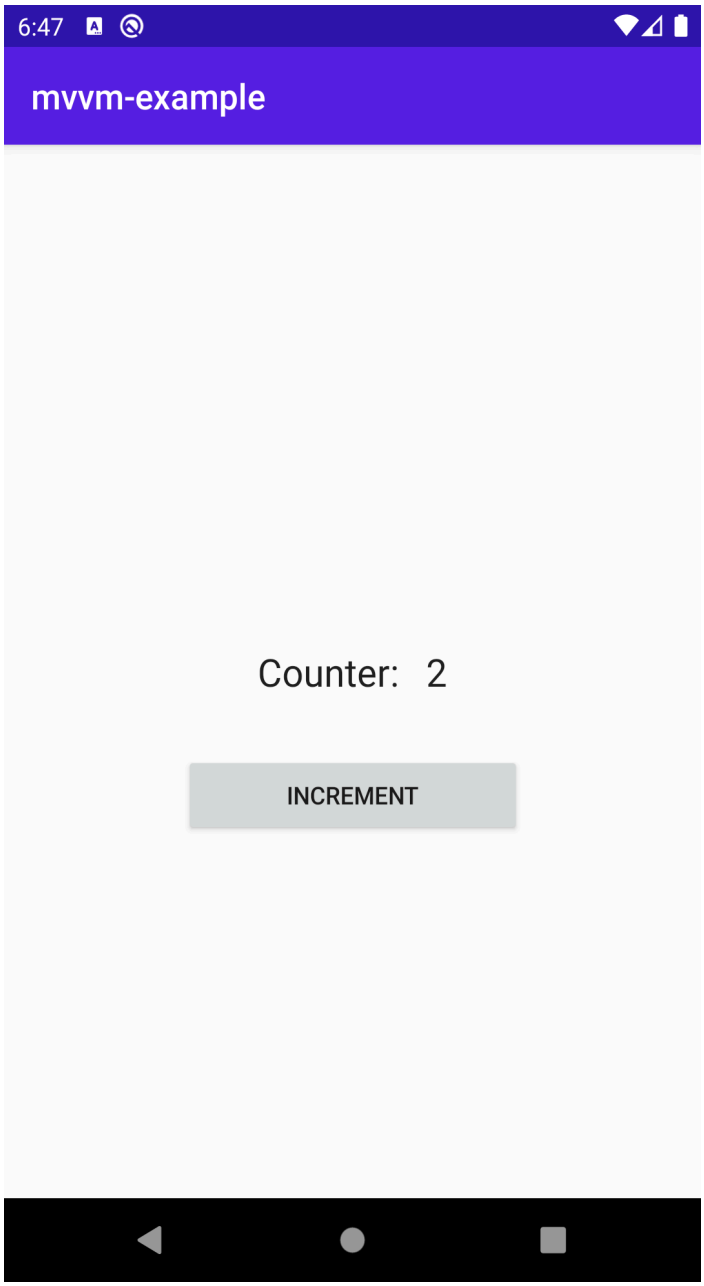
По нажатию на кнопку счетчик увеличивается на 1. Состояние счетчика хранится в классе `MainViewModel` - наследник `ViewModel`. Класс выглядит следующим образом:

```
package ru.samsung.itacademy.mdev

import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel

class MainViewModel : ViewModel() {
    val counter = MutableLiveData<Int>()
    // Can also be written as:
    // val counter: LiveData<Int> = MutableLiveData<Int>()

    fun onIncrementClicked() {
        counter.value = (counter.value ?: 0) + 1
    }
}
```

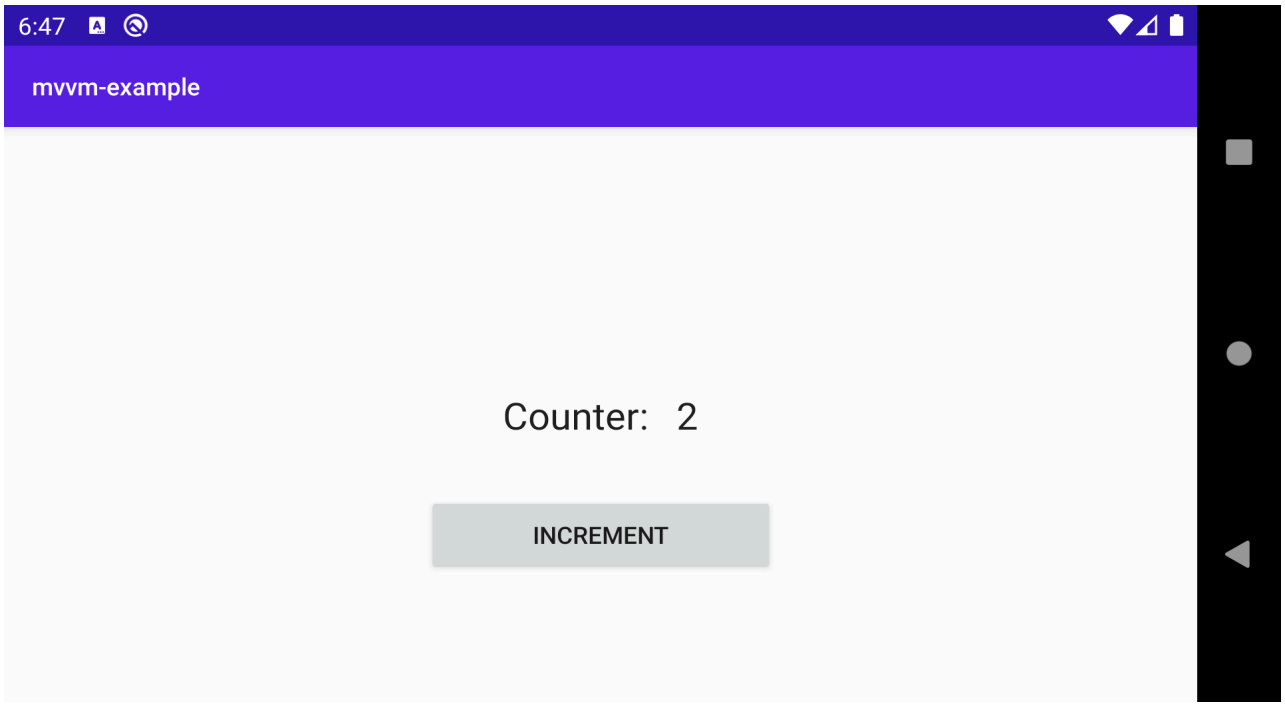
Для счетчика используется объект класса `MutableLiveData` — это подкласс `LiveData`, который является частью Архитектурных компонентов, и следует паттерну `Observer` (наблюдатель). Таким образом при смене ориентации экрана, состояние будет сохранятся. Класс `MainActivity` будет следующим:

```
package ru.samsung.itacademy.mdev

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.Observer
import androidx.lifecycle.ViewModelProvider
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    lateinit var viewModel: MainViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val provider = ViewModelProvider(this)
        viewModel = provider.get(MainViewModel::class.java)

        observeViewModel()

        initView()
    }

    fun observeViewModel() {
        viewModel.counter.observe(this, Observer {
            text_counter.text = it.toString()
        })
    }

    fun initView() {
        btn_increment.setOnClickListener {
            viewModel.onIncrementClicked()
        }
    }
}
```

Таким образом в результате работы приложения получится следующий результат:



После поворота экрана счетчик сохранится:



Полный код приложения можно посмотреть здесь.

Начать тур для пользователя на этой странице