Ciclo de Vida Android ViewModel, LiveData e Coroutines

Pedro Henrique Silva

Senior Software Engineer

Sobre Mim

Objetivo Geral

O objetivo principal desse curso é explorar as bibliotecas de arquitetura ViewModel e LiveData juntamente com coroutines.

Pré-requisitos

- Conhecimento básico em Android
- Conhecimento básico em Kotlin

Ciclo de Vida Android ViewModel, LiveData e Coroutines

Etapa 1	Introdução ao MVVM
Etapa 2	Mantendo dados com ViewModel
Etapa 3	Observando eventos com LiveData
Etapa 4	O que é LifecycleOwner

Ciclo de Vida Android ViewModel, LiveData e Coroutines

Etapa 5	Compartilhando ViewModel entre Fragments
Etapa 6	Usando dependências na ViewModel
Etapa 7	Chamadas assíncronas
Etapa 8	Retornando múltiplos valores

Etapa 1

Introdução ao MVVM

Ciclo de Vida Android ViewModel, LiveData e Coroutines

Etapa 1	**Introdução ao MVVM**
Etapa 2	Mantendo dados com ViewModel
Etapa 3	Observando eventos com LiveData
Etapa 4	O que é LifecycleOwner

O que é MVVM

Model-View-ViewModel (MVVM) é um padrão de projeto criado por arquitetos da *Microsoft*, que é utilizado para separar a lógica do software e os controles de interface do usuário.

Model

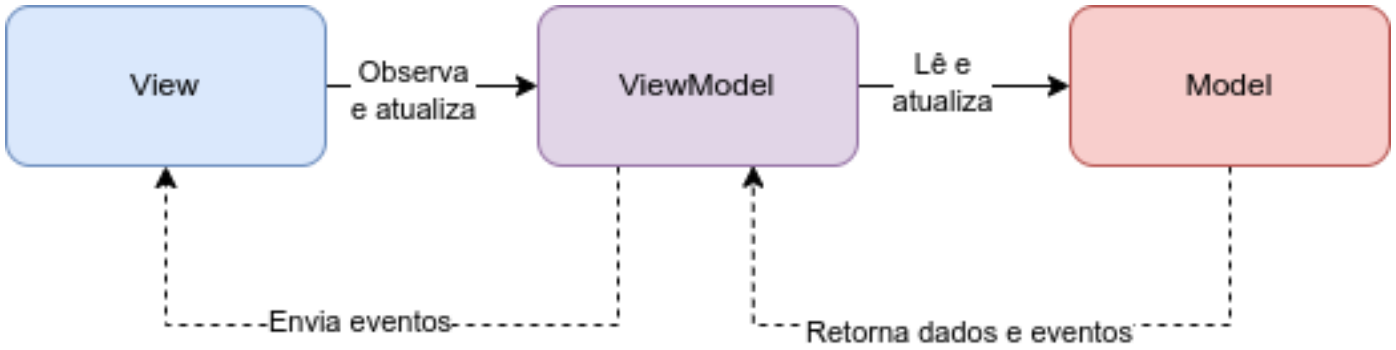
É onde fica a lógica do software, é a dependência da ViewModel que recebe e envia dados para que ela trate e interaja com a View e entradas do usuário.

View

View são os elementos visíveis e que podem receber interação com as entradas do usuário. No Android geralmente nos referimos à *Activity*, *Fragments* e os componentes que estendem a classe *android.view.View* (exemplo: *TextView*, *EditText*, *ImageView*).

ViewModel

Fica entre as camadas View e Model, é onde estão os controles para interagir com a View e expõe controles para que a View possa interagir com ela também.


View
Observa
e atualiza
ViewModel
Lê e
atualiza
Model
Envia eventos
Retorna dados e eventos

Por que devo utilizar?

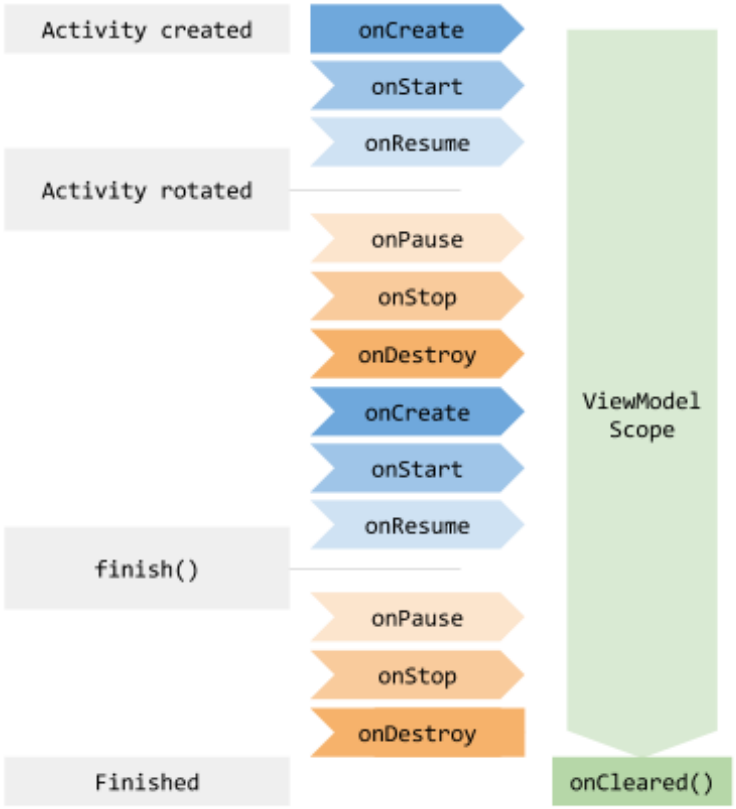
Assim como diversos outros padrões de projeto e arquiteturas, MVVM ajuda organizar o código, quebrando o software em camadas, fazendo com que a manutenção e o reuso de código sejam mais fácil e rápido.

Etapa 2

Mantendo dados com ViewModel

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 1~~	~~Introdução ao MVVM~~
Etapa 2	**Mantendo dados com ViewModel**
Etapa 3	Observando eventos com LiveData
Etapa 4	O que é LifecycleOwner


Activity created
onCreate
onStart
onResume
Activity rotated
onPause
onStop
onDestroy
onCreate
onStart
onResume
finish()
onPause
onStop
onDestroy
Finished
ViewModel Scope
onCleared()

Etapa 3

Observando eventos com LiveData

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 1~~	~~Introdução ao MVVM~~
~~Etapa 2~~	~~Mantendo dados com ViewModel~~
Etapa 3	**Observando eventos com LiveData**
Etapa 4	O que é LifecycleOwner

O que é o LiveData?

LiveData é uma classe armazenadora de dados observável. Diferente de um observável comum, o LiveData conta com reconhecimento de ciclo de vida, ou seja, ele respeita o ciclo de vida de outros componentes do app, como atividades, fragmentos ou serviços. Esse reconhecimento garante que o LiveData atualize apenas os observadores de componente do app que estão em um estado ativo no ciclo de vida.

https://developer.android.com/topic/libraries/architecture/livedata

Vantagens

- Garantia de que a UI corresponde ao estado dos dados
- Sem vazamentos de memória
- Sem falhas causadas por Activities e Fragments interrompidos
- Sem gerenciamento manual do ciclo de vida
- Dados sempre atualizados
- Mudanças de configuração apropriadas
- Compartilhamento de recursos

https://developer.android.com/topic/libraries/architecture/livedata

Etapa 4

O que é LifecycleOwner

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 1~~	~~Introdução ao MVVM~~
~~Etapa 2~~	~~Mantendo dados com ViewModel~~
~~Etapa 3~~	~~Observando eventos com LiveData~~
Etapa 4	**O que é LifecycleOwner**

Etapa 5

Compartilhando ViewModel entre Fragments

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 1~~	~~Introdução ao MVVM~~
~~Etapa 2~~	~~Mantendo dados com ViewModel~~
~~Etapa 3~~	~~Observando eventos com LiveData~~
~~Etapa 4~~	~~O que é LifecycleOwner~~

Ciclo de Vida Android ViewModel, LiveData e Coroutines

Etapa 5	**Compartilhando ViewModel entre Fragments**
Etapa 6	Usando dependências na ViewModel
Etapa 7	Chamadas assíncronas
Etapa 8	Retornando múltiplos valores

Etapa 6

Usando dependências na ViewModel

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 1~~	~~Introdução ao MVVM~~
~~Etapa 2~~	~~Mantendo dados com ViewModel~~
~~Etapa 3~~	~~Observando eventos com LiveData~~
~~Etapa 4~~	~~O que é LifecycleOwner~~

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 5~~	~~Compartilhando ViewModel entre Fragments~~
Etapa 6	**Usando dependências na ViewModel**
Etapa 7	Chamadas assíncronas
Etapa 8	Retornando múltiplos valores

Etapa 7

Chamadas assíncronas - Parte I

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 1~~ ~~Introdução ao MVVM~~

~~Etapa 2~~ ~~Mantendo dados com ViewModel~~

~~Etapa 3~~ ~~Observando eventos com LiveData~~

~~Etapa 4~~ ~~O que é LifecycleOwner~~

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 5~~	~~Compartilhando ViewModel entre Fragments~~
~~Etapa 6~~	~~Usando dependências na ViewModel~~
Etapa 7	**Chamadas assíncronas**
Etapa 8	Retornando múltiplos valores

```kotlin
fun main() {
    val startTime = System.currentTimeMillis()
    println("started")
    method1(startTime)
    method2(startTime)
    println("completed")
}

private fun method1(startTime: Long) {
    val seconds = 5
    Thread.sleep( millis: seconds * 1000L)
    println("method1 ends in ${(System.currentTimeMillis() - startTime) / 1000} seconds")
}

private fun method2(startTime: Long) {
    val seconds = 2
    Thread.sleep( millis: seconds * 1000L)
    println("method2 ends in ${(System.currentTimeMillis() - startTime) / 1000} seconds")
}
```

```
/opt/android-studio/jre/bin/java ...
started
method1 ends in 5 seconds
method2 ends in 7 seconds
completed

Process finished with exit code 0
```

Coroutines (corrotina)

Coroutines são threads leves que permitem escrever código assíncrono sem bloqueio. Kotlin fornece a biblioteca kotlinx.coroutines com uma série de primitivos habilitados para coroutines de alto nível.

https://kotlinlang.org/docs/multiplatform-mobile-concurrency-and-coroutines.html#coroutines

```kotlin
fun main(): Unit = runBlocking { this: CoroutineScope
    val startTime = System.currentTimeMillis()
    println("started")
    method1(startTime)
    method2(startTime)
    println("completed")
}

private fun CoroutineScope.method1(startTime: Long) = launch { this: CoroutineScope
    val seconds = 5
    delay( timeMillis: seconds * 1000L)
    println("method1 ends in ${(System.currentTimeMillis() - startTime) / 1000} seconds")
}

private fun CoroutineScope.method2(startTime: Long) = launch { this: CoroutineScope
    val seconds = 2
    delay( timeMillis: seconds * 1000L)
    println("method2 ends in ${(System.currentTimeMillis() - startTime) / 1000} seconds")
}
```

```
/opt/android-studio/jre/bin/java ...
started
completed
method2 ends in 2 seconds
method1 ends in 5 seconds

Process finished with exit code 0
```

Suspend function

A palavra-chave suspend significa que esta função pode ser bloqueante. Funções suspensas podem ser criadas como funções Kotlin padrão, mas precisamos estar cientes de que só podemos chamá-las de dentro de uma coroutine. Caso contrário, obteremos um erro do compilador.

https://www.baeldung.com/kotlin/coroutines

Context

Coroutines sempre executam em algum contexto representado por um valor do tipo *CoroutineContext*, definido na biblioteca padrão Kotlin. O contexto da corrotina é um conjunto de vários elementos. Os principais elementos são o Job da corrotina, e seu dispatcher, que é abordado nesta seção.

https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html

Dispatchers e threads

O contexto de corrotina inclui um dispatcher de corrotina que determina qual thread ou threads a corrotina correspondente usa para sua execução. O despachante de corrotina pode confinar a execução de corrotina a um encadeamento específico, despachá-lo para um pool de encadeamentos ou deixá-lo ser executado sem confinamento.

https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html#dispatchers-and-threads

```kotlin
launch(Dispatchers.Main) { // context of the parent, main runBlocking coroutine
    println("main runBlocking      : I'm working in thread ${Thread.currentThread().name}")
}
launch(Dispatchers.Unconfined) { // not confined -- will work with main thread
    println("Unconfined            : I'm working in thread ${Thread.currentThread().name}")
}
launch(Dispatchers.IO) { // will get dispatched to DefaultIoScheduler
    println("IO                    : I'm working in thread ${Thread.currentThread().name}")
}
launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher
    println("Default               : I'm working in thread ${Thread.currentThread().name}")
}
launch(newSingleThreadContext( name: "MyOwnThread")) { // will get its own new thread
    println("newSingleThreadContext: I'm working in thread ${Thread.currentThread().name}")
}
```

```
/opt/android-studio/jre/bin/java ...
Unconfined            : I'm working in thread main
Default               : I'm working in thread DefaultDispatcher-worker-2
IO                    : I'm working in thread DefaultDispatcher-worker-1
main runBlocking      : I'm working in thread main
newSingleThreadContext: I'm working in thread MyOwnThread

Process finished with exit code 0
```

Etapa 7

Chamadas assíncronas - Parte II

Etapa 8

Retornando múltiplos valores

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 1~~	~~Introdução ao MVVM~~
~~Etapa 2~~	~~Mantendo dados com ViewModel~~
~~Etapa 3~~	~~Observando eventos com LiveData~~
~~Etapa 4~~	~~O que é LifecycleOwner~~

Ciclo de Vida Android ViewModel, LiveData e Coroutines

~~Etapa 5~~ ~~Compartilhando ViewModel entre Fragments~~

~~Etapa 6~~ ~~Usando dependências na ViewModel~~

~~Etapa 7~~ ~~Chamadas assíncronas~~

Etapa 8 **Retornando múltiplos valores**

Suspend vs Flow

```
suspend fun singleResult(): Int {
    delay( timeMillis: 1_000)
    return 1
}

fun main(): Unit = runBlocking { this: C
    val result = singleResult()
    println(result)
}
```

```
fun multipleResults(): Flow<Int> {
    return flow { this: FlowCollector<Int>
        for (number in 0 ≤ .. ≤ 10) {
            delay( timeMillis: 1_000)
            emit(number)
        }
    }
}

fun main(): Unit = runBlocking { this: Coroutin
    multipleResults().collect { result ->
        println(result)
    }
}
```

Para saber mais

- **ViewModel:** https://developer.android.com/topic/libraries/architecture/viewmodel
- **LiveData:** https://developer.android.com/topic/libraries/architecture/livedata
- **Coroutines:** https://kotlinlang.org/docs/coroutines-overview.html
- **Flow:** https://kotlinlang.org/docs/flow.html
- **Codelabs:** https://developer.android.com/codelabs/advanced-kotlin-coroutines

Dúvidas?

> Fórum/Artigos

> Comunidade Online (Discord)