soluções mobile

prof. Thyerri Mezzari

Android / Revisão
Layouts, Componentes Visuais, Activity e Intents

Android: ViewModel, Threads, AsyncTask, Service e AlarmManager

ViewModel

```
class DataViewModel : ViewModel()
```

Atualização de Dados na UI

Uma das atividades principais de um app é atualizar componentes dispostos em tela para o usuário. A ideia de abastecer telas com dados vindos de um ponto de conexão on-line, de um banco de dados local ou restaurar o estado de uma tela com base em preferências do usuário, sempre será um desafio para o desenvolvedor de aplicativos.

Desde as primeiras versões do Android SDK o conceito mais simples de atualizar a UI de um aplicativo passa por obter o campo através de ***findViewById*** e executar os métodos necessários para atualização do componente em questão.

Atualização de Dados na UI

Por exemplo um label de texto disposto em tela, sendo este componente do tipo *TextView* com o ID "count" podemos acessá-lo assim:

```kotlin
// kotlin
val count = findViewById<TextView>(R.id.count)
count.setText("10 cliques")
```

```java
// java
TextView count = this.findViewById(R.id.count);
count.setText("10 cliques");
```

O código acima executa uma ação simples, obtém a referência de um TextView em tela e "dinamicamente" (via código) alterar o valor que será exibido para o usuário do app.

Atualização de Dados na UI

Podemos estender a ideia do exemplo anterior para diversas outras situações, por exemplo obter uma lista de tarefas de um servidor on-line e criar uma linha em tela para cada tarefa encontrada.

Neste exemplo nós teríamos que preparar uma requisição HTTP a ser executada no momento que nossa *Activity* ou *Fragment* é criado em tela, e após a finalização da requisição HTTP montar em tela tudo que precisamos exibir ao usuário.

Este seria um típico exemplo de UI dinâmica em que no layout XML teríamos apenas uma base "vazia" e que o código de programação do app iria realmente finalizar a montagem de tela.

Atualização de Dados na UI

O maior problema de gerenciar dados dinâmicos e de como eles são atualizados em tela está diretamente relacionado ao ciclo de vida de uma *Activity* ou *Fragment*.

Não há garantias que o Android OS em determinada situação não irá destruir e recriar sua *Activity* (e com ela todos os seus fragmentos). Um aplicativo que fique alguns minutos minimizado pode voltar do "background" com sua UI completamente "resetada". Um aplicativo que possui uma navegação baseada em *Activity* (modelo clássico) pode ter sua primeira tela "morta" quando o usuário tiver navegado o suficiente app a dentro.

E por fim, o exemplo mais "recorrente": sempre que a orientação de seu app mudar (exemplo ao deitar seu celular) e o layout de seu aplicativo for "redesenhado" para ocupar o novo "espaço horizontal" sua Activity será reiniciada e todos os seus dados dinâmicos serão "perdidos".

ViewModel

A classe ***ViewModel*** foi projetada para armazenar e gerenciar dados relacionados à UI considerando o ciclo de vida. Ela permite que os dados sobrevivam às mudanças de configuração, como por exemplo a rotação da tela e também ajudará nos outros problemas citados nos slides anteriores.

O framework do Android gerencia os ciclos de vida dos controladores de UI, como atividades e fragmentos. O framework pode decidir destruir ou recriar um controlador de UI em resposta a determinadas ações do usuário ou eventos do dispositivo que estão completamente fora do seu controle.

Se o sistema destruir ou recriar um controlador de IU, todos os dados temporários relacionados à UI armazenados nele serão perdidos.

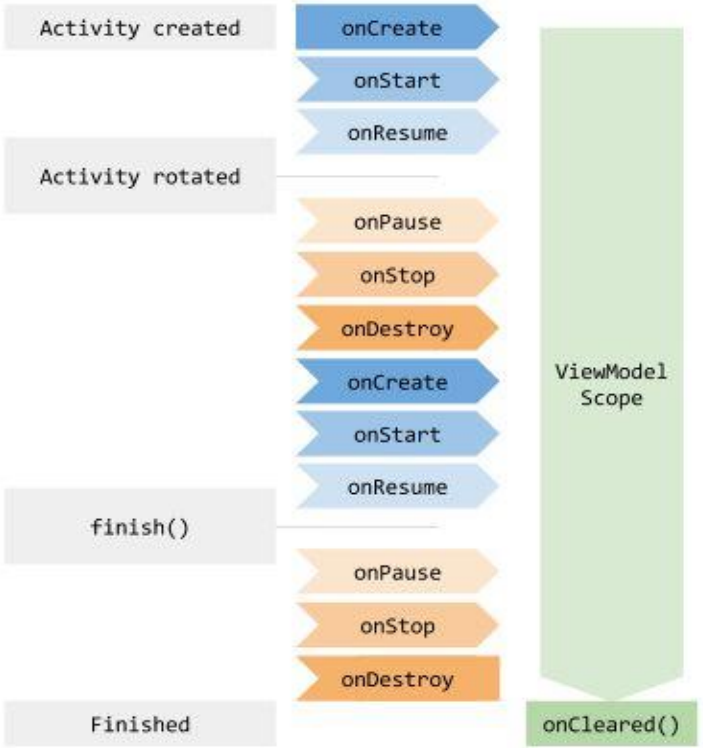
ViewModel

Antigamente para dados simples, a atividade poderia usar o método ***onSaveInstanceState()*** para salvar os dados em memória e restaurar os próprios dados a partir do método em ***onCreate()***.

Com o tempo a equipe do Android chegou à conclusão de que é mais fácil e eficiente separar a propriedade de dados de visualização da lógica do controlador de UI, criando uma forma de "independência" entre as partes.

Logo os objetos ***ViewModel*** são retidos automaticamente durante as mudanças de configuração, de modo que os dados retidos estejam imediatamente disponíveis para a próxima atividade ou instância de fragmento.

LifeCycle de um ViewModel



Configurando o projeto para uso de ViewModel

Ao criar um projeto, nem sempre o mesmo estará preparado para o uso de ***ViewModel***, principalmente se o mesmo estiver baseado em Kotlin.

Um dos requisitos básicos é que seu projeto seja compilado usando a **JVM (Java Virtual Machine) mínima na versão 1.8**.

E se você estiver trabalhando com Kotlin, pacotes adicionais deverão ser instalados em seu projeto

Em seu arquivo **app/build.gradle:**

```
...
dependencies {
    // variável que define a versão do pacote ViewModel/LiveData
    def lifecycle_version = "2.2.0"

    ...

    // esse provavelmente já vai estar no projeto
    implementation 'androidx.core:core-ktx:1.3.1'

    // adicionado o básico de uso ViewModel
    implementation "androidx.activity:activity-ktx:1.1.0"
    implementation "androidx.fragment:fragment-ktx:1.2.5"

    // ViewModel em si
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"

    // LiveData
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"

    // Lifecycles only (without ViewModel or LiveData)
    implementation "androidx.lifecycle:lifecycle-runtime-ktx:$lifecycle_version"

    // Saved state module for ViewModel
    implementation "androidx.lifecycle:lifecycle-viewmodel-savedstate:$lifecycle_version"

    // if using Java8, use the following instead of lifecycle-compiler
    implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"

    ...
}
```

Criando uma classe ViewModel

A ideia de uso de uma classe ***ViewModel*** é simples, ao invés de declarar variáveis soltas (por exemplo um contador) em sua classe *Activity* ou *Fragment*, você irá criar um nova classe para armazenar dados e também manipulá-los caso necessário.

Essa classe deve obrigatoriamente estender/herdar métodos da *classe padrão ViewModel* e não poderá interagir diretamente com a tela do usuário (ex: usar findViewById).

```kotlin
// kotlin
class ContadorViewModel : ViewModel() {
    var count:Int = 0
}
```

Usando uma classe ViewModel

Após criarmos a classe, basta instanciarmos a mesma em nossa *Activity* ou *Fragment* para podermos usar seus dados livremente.

Na Activity:

```kotlin
// kotlin
val correctModel:CorrectViewModel by viewModels()
```

Em um Fragment:

```kotlin
// kotlin
val correctModel:CorrectViewModel by activityViewModels()
```

Exemplo rodando no ***Android Studio***.

Threads

Main Thread, Threads de Trabalho e AsyncTask

A Thread “principal”

Quando um aplicativo é iniciado, e não há outro componente deste aplicativo em execução, o sistema Android inicia um novo processo no Linux (base do Android OS) para o app, este novo processo ou ***Thread*** irá se tornar o veículo principal de processamento de sua interface e também será conhecido como ***Main Thread*** (Thread “principal”).

Essa ***Thread*** é muito importante, sendo ela encarregada de despachar eventos para os componentes da interface do usuário. Quase sempre, é também a ***Main Thread*** que interage com componentes do kit de ferramentas da UI do Android (componentes dos pacotes android.widget e android.view). Portanto, a thread principal também pode se chamar thread de UI.

Quando o aplicativo realiza trabalho intenso em resposta à interação do usuário, esse modelo de uma thread principal pode produzir desempenho inferior, a não ser que você "desenhe" o aplicativo adequadamente.

A Thread “principal”

Quando a ***Main Thread*** é bloqueada, devido ao um alto processamento (cálculos, downloads, processamento de imagem), nenhum evento extra pode ser despachado, incluindo eventos de interação (como clicar em outro botão, por ex).

Da perspectiva do usuário, o aplicativo parece estar travado. Pior ainda, se a thread de UI for bloqueada por mais do que alguns segundos (cerca de 5 segundos atualmente), o usuário receberá a vergonhosa mensagem “aplicativo não respondendo” (ANR).

Sendo assim, o usuário talvez decida fechar o aplicativo e desinstalá-lo se estiver descontente.

Threads de "trabalho"

Por causa deste modelo de thread única descrito anteriormente, é essencial para a capacidade de resposta da UI do aplicativo que você não bloqueie a ***Main Thread***. Caso você queira realizar operações que não sejam instantâneas, faça isso em threads separadas (threads de “segundo plano” ou “de trabalho”).

Apesar da possibilidade de trabalhar com ***Threads*** secundárias a fim de "destravar" o app, já fica o aviso que no entanto, você não pode atualizar a interface do usuário de qualquer thread diferente da thread “principal”. Iremos ver isso melhor adiante.

Travando seu app

Talvez ainda não tenha ficado evidente o que é um ANR, por isso vamos simular um problema deste tipo:

```kotlin
// kotlin
fun buttonClick(view: View) {
    var i = 0
    while (i <= 20) {
        try {
            Thread.sleep(1000)
            i++
        } catch (e: Exception) {
          // error
        }

    }
    myTextView.text = "Button Pressed"
}
```

A ideia deste código seria "travar" a tela de seu aplicativo em um processamento contínuo por 20 segundos (20 pausas de 1 segundo), após este processamento concluído teríamos o final da operação que é mudar o texto de um TextView em tela para "Button Pressed".

Destravando seu app

O jeito mais simples de "destravar" sua operação é iniciar uma nova ***Thread*** e executar o trabalho dentro dela:

```kotlin
// kotlin
fun buttonClick(view: View) {
  Thread(Runnable {
    var i = 0
    while (i <= 20) {
        try {
            Thread.sleep(1000)
            i++
        } catch (e: Exception) {
          // error
        }

    }
    myTextView.text = "Button Pressed"
  }).start()
}
```

Algo de errado não está certo...

Destravando seu app

Lembram quando eu disse "você não pode atualizar a interface do usuário de qualquer thread diferente da main thread"? Pois é.

Após terminar seus cálculos / processamento de atividade mais complexo precisaremos "sair" da Thread alternativa e retornar para a Thread "principal" a fim de atualizar algum componente em tela. Para isto temos algumas alternativas:

- **Activity.runOnUiThread(Runnable)**
- **View.post(Runnable)**
- **View.postDelayed(Runnable, long)**

Destravando seu app

O erro do exemplo anterior poderia ser resolvido deste jeito:

```
// kotlin
fun buttonClick(view: View) {
  Thread(Runnable {
    var i = 0
    ...

    myTextView.post {
      myTextView.text = "Button Pressed"
    }
  }).start()
}
```

Destravando seu app

O erro do exemplo anterior poderia ser resolvido deste jeito:

```
// java
public void buttonClick(View v) {
  new Thread(new Runnable() {
    public void run() {
      int i = 0;
      ...

      myTextView.post(new Runnable() {
        public void run() {
          myTextView.text = "Button Pressed";
        }
      });
    }
  }).start();
}
```

AsyncTask

Atividades de processamento moderado e sem grande complexidade podem ser executadas através de novas *Thread* sem problemas. Porém quando a atividade necessita de um controle mais refinado e que por vezes irá durar mais tempo do que o normal, podemos precisar de uma solução mais robusta, como as classes ***AsyncTask*** por exemplo.

A ***AsyncTask*** permite que você trabalhe de forma assíncrona na interface do usuário. Ela realiza operações de bloqueio em uma thread alternativa de trabalho e, em seguida, publica os resultados na Main Thread, sem que você precise lidar com os métodos alternativos de acesso a thread "principal".

Outra grande vantagem de uma ***AsyncTask*** é poder acompanhar o progresso de sua tarefa, podendo até implementar uma barra de progresso.

O jeito mais comum de criar uma ***AsyncTask*** é definindo a mesma como uma subclasse em sua *Activity* ou *Fragment*:

```kotlin
// kotlin
class AsyncDemoActivity : AppCompatActivity() {

    private inner class MyTask : AsyncTask<String, Int, String>() {

        override fun onPreExecute() {
          // método executado antes da tarefa iniciar
        }

        override fun doInBackground(vararg params: String): String {
          // lógica de processamento pesada / complexa
        }

        override fun onProgressUpdate(vararg values: Int?) {
          // atualização de progresso
        }

        override fun onPostExecute(result: String) {
          // tarefa concluida / finalizada
        }

    }

}
```

AsyncTask

Cada declaração de uma nova AsyncTask deve ser acompanhada pela definição de 3 tipos:

```
AsyncTask<[Params], [Progress], [Result]>
```

- **Params**, o tipo dos parâmetros de entrada dentro da AsyncTask.
- **Progress**, o tipo de parâmetro de progresso publicado ao longo da execução da tarefa, normalmente um inteiro.
- **Result**, o tipo do parâmetro de resultado após o processamento complexo.

Caso você não utilize um ou mais parâmetros de entrada / retorno, basta marcá-los com o tipo ***Void***.

AsyncTask

Depois que sua subclasse ***AsyncTask*** já esteja devidamente declarada em seu código, para executar a mesma basta usar o método **.execute()**:

```
// kotlin
fun buttonClick(view: View) {
  // caso a AsyncTask recebesse parametros de entrada
  // era só rodar .execute(x, y, z, b, d)
  val task = ContagemTask().execute()
}
```

Cancelando a Execução de uma AsyncTask

Digamos que você criei uma ***AsyncTask*** que irá baixar um arquivo de vídeo do servidor para o celular do usuário, é um arquivo grande de uns 20 mb por exemplo.

Você evidentemente irá mostrar uma barra de progresso do download para o usuário acompanhar quanto ainda falta baixar, e possivelmente também irá adicionar algum tipo de botão "cancelar" caso o usuário desista de fazer o download do arquivo.

Sendo assim, se você precisar cancelar alguma ***AsyncTask*** basta rodar o comando **.cancel()**:

```
// kotlin ou java
task.cancel()
```

Services

Auxiliares invisíveis

Serviços

Um ***Service*** é uma parte do aplicativo que pode realizar operações longas e que não possua uma interface do usuário.

Qualquer componente, *Activity* ou *Fragment* de um aplicativo pode iniciar um serviço e ele continuará em execução em segundo plano (background) mesmo que o usuário alterne para outro aplicativo de seu smartphone ou tablet.

Além disso, um componente poderá se vincular a um serviço para interagir com ele e até estabelecer comunicação entre processos.

Por exemplo, um serviço pode lidar com transações de rede, reproduzir música ou interagir com um provedor de conteúdo, tudo a partir do segundo plano.

Service x Thread x AsyncTask

Um serviço é simplesmente um componente de um app que pode ser executado em segundo plano mesmo quando o usuário não esteja interagindo com o aplicativo. Sendo assim, crie um serviço somente se é isso que você quer fazer (rodar algo invisível em background).

Caso precise realizar trabalhos fora do thread principal, mas somente enquanto o usuário estiver interagindo com o aplicativo, crie um novo thread ou trabalhe com *AsyncTask*, e não um serviço.

Criando um Service

Para criar um serviço, você precisa criar uma classe que herde/estenda uma das classes ***Service*** da Android SDK.

Na implementação, será necessário modificar alguns métodos de *callback* que lidem com aspectos essenciais do ciclo de vida do serviço e forneçam um mecanismo para vincular componentes ao serviço, se apropriado.

Criando um Service

Veja os métodos de callback mais importantes a modificar:

- **onStartCommand**() - O sistema invoca esse método quando outro componente, como uma atividade, solicita que o serviço seja iniciado, chamando startService.
- **onBind**() - O sistema invoca esse método quando outro componente quer se vincular ao serviço (como para realizações de RPC) chamando bindService.
- **onCreate**() - O sistema invoca esse método para realizar procedimentos únicos de configuração (antes de chamar onStartCommand ou onBind). Se o serviço já estiver em execução, esse método não é chamado.
- **onDestroy**() - O sistema invoca esse método quando o serviço não é mais usado e está sendo eliminado.

Registrando um Serviço

Assim como uma *Activity*, um ***Service*** customizado de seu aplicativo também precisará ser registrado em seu arquivo *AndroidManifest.xml*.

Para declarar o serviço, adicione um elemento <service> como filho do elemento <application>:

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".MeuTesteService" />
      ...
  </application>
</manifest>
```

O único atributo necessário é ***android:name*** — ele especifica o nome da classe do serviço.

Como executar um service?

Basicamente existem dois modos de iniciar um Service a partir de sua activity, usando os métodos **startService**() ou **bindService**().

Diferença básica entre os dois métodos:

startService() - inicia e executa independentemente do processo que o criou - sempre do zero.

bindService() - inicia (caso ainda não esteja em execução) e/ou simplesmente realiza o bind (conecta a um serviço já existente / rodando).

Como parar um service?

Supondo que você tenha um service em background tocando uma música, o usuário poderá "minimizar" seu app e a música continuar tocando tranquilamente. Porém você como desenvolvedor deseja que no momento que o usuário encerre definitivamente a execução de seu app, que o serviço também seja encerrado, assim parando de tocar a música. Como fazemos isso?

Temos 3 comandos a disposição para fazer isso:

stopSelf() ou **stopService**() - para quando o serviço tenha sido iniciado via startService.

unbindService() - para quando o serviço tenha sido iniciado/vinculado com bindService

O ideal é mantermos esse tipo de "proteção" nos métodos ***onStop*** ou ***onDestroy*** da *Activity* relacionada a fim de encerrar o service junto com o app.

Ciclo de vida de um Service

O ciclo de vida do serviço — desde a criação à eliminação — pode seguir um destes dois caminhos:

Um serviço iniciado

O serviço é criado quando outro componente chama **startService**(). Depois, o serviço permanece em execução indefinidamente e precisa chamar **stopSelf**() para ser interrompido. Outro componente também pode interromper o serviço chamando **stopService**(). Quando o serviço é interrompido, o sistema o elimina.

Ciclo de vida de um Service

O ciclo de vida do serviço — desde a criação à eliminação — pode seguir um destes dois caminhos:

Um serviço vinculado

O serviço é criado quando outro componente (um cliente) chama **bindService**(). O cliente se comunica com o serviço pela interface ***IBinder***. O cliente pode escolher encerrar a conexão chamando **unbindService**(). Vários clientes podem ser vinculados ao mesmo serviço e, quando todos os vínculos terminam, o sistema elimina o serviço. O serviço não precisa ser interrompido.

Ciclo de vida de um Service

Por fim, esses dois caminhos não são inteiramente separados. É possível vincular (***bindService***) um serviço que já foi iniciado com ***startService***().

Por exemplo, você pode iniciar um serviço de música em segundo plano iniciando ***startService***() com um Intent que identifica a música a ser reproduzida. Depois, possivelmente quando o usuário quiser exercer mais controle sobre o reprodutor ou ter informações sobre a música em reprodução, uma atividade pode chamar ***bindService***() para acessar o serviço que já está rodando.

Nesses casos, ***stopService***() ou ***stopSelf***() não interrompem o serviço até todos os clientes serem desvinculados (ninguém mais usando).

AlarmManager

Execução agendada...

AlarmManager

Qual a maneira mais correta para disparar uma intent/ação de um app exatamente as 19 horas, todos os dias?

Com ***AlarmManager*** podemos agendar o disparo/execução de uma ação relacionada ao seu app na data e hora desejada, ou até mesmo programar um intervalo para repetição de uma mesma ação.

Os *Alarms* são gerenciados pela aplicação através do serviço ***AlarmManager***, disponibilizado pelo sistema operacional Android. Ao sair da aplicação o sistema não destruirá um alarme configurado. Porém vale observar que ao reiniciar o aparelho seu alarme agendado poderá ser descartado.

AlarmManager

Os alarmes (com base na classe ***AlarmManager***) oferecem uma maneira de realizar operações baseadas em tempo fora da vida útil do seu app. Você pode usar um alarme para iniciar uma operação de longa duração, por exemplo, para iniciar um serviço uma vez por dia para fazer o download de uma previsão do tempo.

Os alarmes têm estas características:

- Permitem que você dispare intents (ações) em horários e/ou intervalos definidos
- Operam fora do seu app, de modo que você pode usá-los para acionar eventos ou ações mesmo quando o app não está em execução e até mesmo se o próprio dispositivo está inativo
- Ajudam a minimizar os requisitos de recursos do app, sendo que você pode programar operações sem depender de timers ou da execução contínua de serviços em segundo plano

Definir um alarme recorrente

Para "setarmos" um *Alarm* junto ao ***AlarmManager*** de maneira recorrente precisamos das seguintes definições:

- Um tipo de alarme.
- Um tempo de acionamento.
- O intervalo do alarme. Por exemplo: uma vez por dia, a cada hora, a cada cinco minutos e assim por diante.
- Um intent (ação) pendente que é acionado quando o alarme é disparado.

Escolher um tipo de alarme

Uma das primeiras considerações sobre o uso de um alarme recorrente é o tipo dele.

Há dois tipos de relógio geral para alarmes:

- O relógio "tempo real decorrido" que usa o "tempo desde a inicialização do sistema" como referência
- E o "relógio em tempo real" que se adequa ao fuso horário local

Escolher um tipo de alarme

Se você simplesmente precisa que seu alarme dispare em um intervalo específico (por exemplo, a cada meia hora), use um dos tipos de "tempo real decorrido". Em geral, essa é a melhor escolha.

Veja a lista de tipos mais comuns:

ELAPSED_REALTIME_WAKEUP: ativa o dispositivo e dispara o intent pendente após o tempo especificado desde a inicialização do dispositivo.

RTC_WAKEUP: ativa o dispositivo para disparar o intent pendente no horário especificado.

Definindo um alarme

Depois de escolher o tipo de seu alarme, você deverá acionar o AlarmManager do SO e definir um método de execução:

alarmMgr.set - para uma execução única

alarmMgr.setInexactRepeating - para uma execução repetitiva não exata

alarmMgr.setRepeating - para um execução repetitiva exata

Definindo um alarme

Para maiores detalhes sobre como definir alarmes sugiro dar uma olhada na documentação:

https://developer.android.com/training/scheduling/alarms

obrigado 🚀