Desenvolvido por [Paulo Salvatore]

Movile Next Program
Formação em Desenvolvimento *Android*
Prof. Paulo Salvatore
Tópicos abordados:

- ***Architecture Patterns* (*MVC, MVP e MVVM*)**
- ***Data Binding***
- ***Reactive Programming* com *ReactiveX: RxJava, RxAndroid* e *RxKotlin***
- ***Dependency Injection* com *Dagger 2***

1. *Movile Next Program*

O *Movile Next* é uma iniciativa criada pelo Grupo *Movile*, em parceria com a *GlobalCode*, para capacitar desenvolvedores *Android*, *iOS* e *Backend* de nível pleno e sênior e possibilitar um avanço em carreira. Durante 4 sábados acontecerão, simultaneamente, as formações de cada especialidade com aulas presenciais ministradas por especialistas do mercado, além de atividades e desafios complementares online durante a semana.

2. Introdução

Nesse material iniciaremos uma abordagem sobre padrões de arquitetura mais modernos e suas evoluções, até chegar no padrão mais recente e atualmente considerado o mais recomendado para aplicações no *framework Android*.

Falaremos sobre *data binding*, uma ferramenta presente no dia-a-dia de todo desenvolvedor avançado de *Android* com diversas funcionalidades interessantes que além de facilitar a vida no desenvolvimento também melhora o fluxo da aplicação, permitindo com que os *designers* de *UX* possam ter mais liberdade na hora de declarar o conteúdo do *layout*, fazendo isso direto pelo *xml*. Além disso também veremos maneiras de criar atributos customizados, permitindo uma dinâmica mais interessante no *xml*.

Entraremos em um dos assuntos mais promissores atualmente, o *ReactiveX*, o pacote de extensões para utilização da programação reativa dentro do nosso universo utilizando as bibliotecas *RxJava* e outras relacionadas. Entenderemos o que é exatamente esse conceito, o que devemos entender e como aplicamos no nosso dia-a-dia de desenvolvimento, desmistificando o uso dessa tecnologia e aproximando mais das nossas aplicações.

Por fim, falaremos sobre a *dependency injection*, ou injeção de dependência, utilizando a biblioteca *Dagger 2* para realização das implementações, entendendo exatamente a importância de realizar esse uso, como essa tecnologia funciona para o desenvolvedor *Android* e o que devemos fazer para implementar.

3. Sumário

4. Architecture Patterns

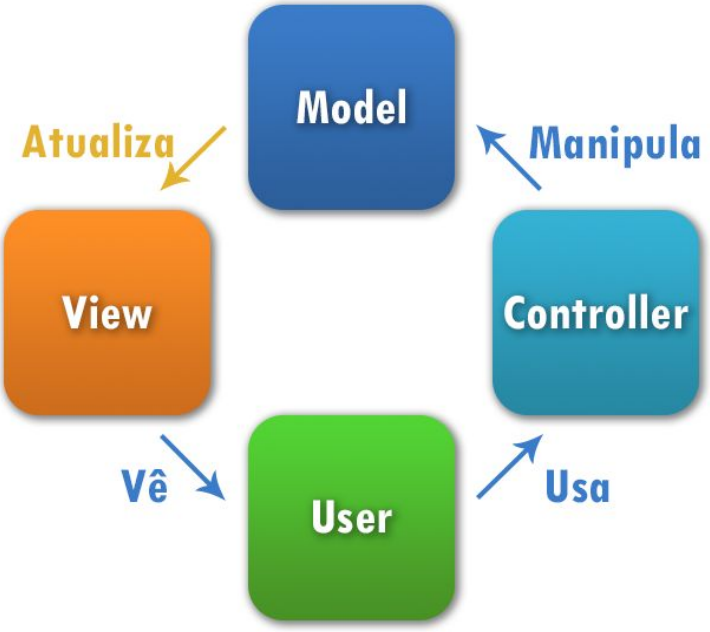
Durante diversos momentos do curso, alguns conceitos importantes de arquitetura, princípios de programação e desenvolvimento ágil serão apresentados. Nessa segunda aula falaremos sobre os padrões de arquitetura conhecidos como MVC, MVP e MVVM, entendendo a evolução e a importância de cada um, as principais características e atrelando o uso deles no desenvolvimento *Android*.

4.1. MVC

O MVC, ***Model-View-Controller***, é um padrão de arquitetura que sintetiza a maneira que o as informações de uma aplicação devem estar relacionadas, colocando o usuário em uma das pontas e ditando exatamente o que cada parte do *software* deve fazer para que o conteúdo chegue até o usuário utilizando boas práticas de programação, promovendo a reutilização de código e visando um trabalho mais eficiente.

Essa abordagem surgiu em 1979, na *Xerox PARC*, e é uma das mais conhecidas sendo utilizada em diversas plataformas de desenvolvimento, contemplando diversas linguagens de programação, principalmente nas aplicações feitas para a *web* [ref 1]. Uma representação pode ser visualizada na figura 1.

Figura 1: Relação MVC



Fonte: autor

Nessa arquitetura, o ***model*** é responsável pela comunicação com os dados, independente de onde eles estão. Ele receberá notificações do *controller* conforme a necessidade desses dados, irá buscá-los, realizar alguma lógica caso haja necessidade e retornar a informação pronta para ser utilizada. Uma frase bem conhecida nesse padrão é "*Fat models and skinny controllers*", que significa *models* gordos e *controllers* magros, porém, eu prefiro dizer a frase "*Skinny models and skinny controllers*", afinal, se alguma coisa está "gorda" é porque cometemos algum erro no planejamento básico. A ideia dessa frase é que a principal lógica dos dados deve estar no *model*, e não no *controller*.

A ***view*** é o componente que o usuário visualiza todas as informações carregadas pelos componentes citados anteriormente. Uma característica da *view* é que ela deve ser "burra", ou seja, não pode conter nenhuma lógica e não deve saber de nada do que está acontecendo, apenas recebe a informação e exibe da melhor maneira possível. Ela é atualizada conforme os dados recebidos do *model* e geralmente essa atualização é feita pelo *controller*.

O ***controller*** por sua vez é responsável por solicitar informações do *model* conforme a necessidade do usuário. Ele também solicitará atualizações no *model*, informando que um cadastro foi atualizado, por exemplo, ou que um registro precisa ser removido. Todos esses gatilhos são realizados a partir da interação do usuário com o sistema, onde o *controller* recebe diretamente essa interação e repassa para o local necessário.

No *Android:* o *model* seria composto de classes de acesso ao banco de dados ou de acesso à *APIs* na *web*; a *view* seria o *layout XML* que exibe os dados diretamente inseridos pela classe da *Activity;* e o *controller* seria nossa classe *Activity* e qualquer classe de comunicação que solicitasse informações ao *model* . A abordagem pode variar em diversos casos mas essa seria uma representação básica, onde não temos uma separação de fato entre *controller* e *view*, onde nossa *Activity* ou *Fragment* atuava muitas vezes em ambas as funções.

Essa solução é considerada um *Monolith*, ou seja, uma maneira antiga de basicamente ter um projeto inteiro utilizando uma única solução de arquitetura contendo todo o código e a lógica dentro dela. Com o crescimento e evolução das aplicações, esses padrões de arquitetura também evoluíram e novas formas bem interessantes foram desenvolvidas e se difundiram entre os desenvolvedores.

4.2. MVP

O MVP, ***Model-View-Presenter***, é um padrão de arquitetura que deriva do *MVC*, visto anteriormente, e é usado na maioria das vezes mas construção de interfaces de usuário, como é o caso do *Android*. Nessa abordagem, temos a

substituição do *controller* pelo *presenter*, que assume a função de "homem-do-meio", onde toda lógica será feita pelo *presenter*, basicamente respondendo às ações da *UI* e controlando a interação entre a *view* e o *model* .

Essa abordagem surgiu no começo dos anos 1990, na *Taligent*, que era uma *joint-venture* da *Apple*, *IBM* e *HP* e começou a ficar bem difundida em 2006 quando a *Microsoft* incorporou em sua documentação incluindo alguns exemplos com o *framework* .NET [ref 2]. Uma representação pode ser visualizada na figura 2.

Figura 2: Relação MVP

Fonte: autor

Nessa arquitetura, o ***model*** é idêntico ao apresentado no *MVC*.

A ***view*** mantém a mesma funcionalidade do *MVC*.

O ***presenter***, que entra para substituir o *controller*, é responsável por intermediar a relação entre a *view* e o *model*, garantindo que qualquer comunicação necessária passe por ele. Ele recebe os dados fornecidos pelo *model*, formata e envia para serem exibidos na *view*. Em algumas implementações é

comum criarmos um novo componente chamado *Supervising Controller*, que irá cuidar de casos onde é necessário recuperar ou persistir o *model* .

No *Android:* o *model* mantém as mesmas classes do *MVC*; a *view* passa a possuir a classe da *Activity*, além dos *layouts* que já existiam no padrão *MVC* e também passa a implementar uma interface *view* que será reconhecida pelo *presenter;* e o *presenter* são novas classes que implementam uma interface *Presenter* que irão atualizar o *model* para solicitar os dados, além de guardar a referência da interface *view* implementada. É preferível que o *presenter* não possua nenhuma referência ou código relacionado à *API* do *Android.*

Um artigo interessante que fala um pouco sobre o *MVC* e o *MVP* fazendo um paralelo entre as duas abordagens no universo *Android* pode ser encontrado no capítulo de leitura adicional.

O *MVP* se tornou muito popular entre os desenvolvedores *Android* com o passar do tempo, porém, para iniciar o desenvolvimento nessa abordagem requer muita escrita de boilerplate e algumas práticas como guardar a referência da *view* em questão dentro do *presenter* dificultam alguns processos dentro dessa arquitetura, principalmente quando temos um *presenter* muito grande e difícil de quebrar em pedaços.

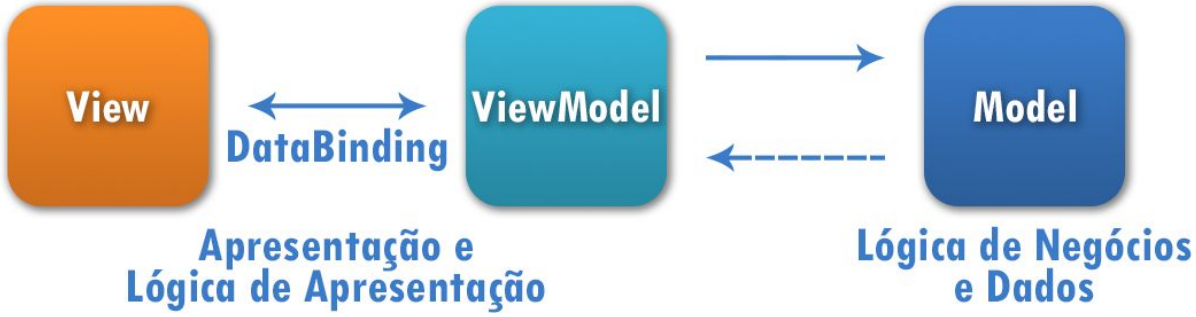
4.3. MVVM

O MVVM, ***Model-View-ViewModel***, também conhecido como *Model-View-Binder*, é uma das abordagens mais atuais em termos de arquitetura de *software*, principalmente na comunidade *Android* recebendo cada vez mais atenção pelos desenvolvedores e pelos adeptos da *Clean Architecture*.

Essa abordagem foi anunciada pela primeira vez em 2005, pela *Microsoft*, e tem como característica facilitar a separação de conceitos no desenvolvimento de uma interface de usuário [ref 3]. Ela foi desenvolvida para utilizar funções conhecidas como *data bindings*, que facilitam uma melhor abstração da *view* do resto do projeto, pois remove boa parte do código de atualização dos elementos da *UI*, eliminando a necessidade dos *designers* de *UX* de escreverem esses códigos, permitindo que utilizem uma linguagem de marcação para isso. A separação dessas tarefas foi um dos pontos pensados durante o desenvolvimento dessa arquitetura visando acelerar o processo de desenvolvimento de uma aplicação tanto em projetos grandes quanto em projetos pequenos. Uma representação pode ser visualizada na figura 3.

Desenvolvido por [Paulo Salvatore]

Figura 3: Relação MVVM



Fonte: autor

Nessa arquitetura, o ***model*** é idêntico ao apresentado no *MVC* e no *MVP*.

A ***view*** ganha uma nova funcionalidade de *bind* em variáveis "observáveis" e ações presentes no *ViewModel* de uma maneira mais flexível.

O ***ViewModel***, que entra para substituir o *controller* e o *presenter*, é responsável por envolver o *model*, preparando variáveis "observáveis" que são necessárias para a *view*. Ele também funciona como uma ponte para a *view* enviar eventos ao *model*. Uma característica importante desse modelo é que o *ViewModel* não possui referência da *view*, permitindo com mais facilidade que um *ViewModel* sirva várias *views*.

No *Android:* o *model* mantém a mesma base do *MVC*; a única diferença da *view* é que ela não possui mais uma interface *view* implementada e passa a ter declarações de observações às variáveis do *ViewModel;* e o *ViewModel* possuirá uma implementação da interface *ViewModel* que está ciente do ciclo de vida da *Activity* ou *Fragment* e ficará responsável pela declaração e atualização das variáveis observáveis que serão observadas pela *view* [ref 4].

Um artigo muito interessante com uma abordagem bem rica de informações e exemplos sobre *MVC*, *MVP* e *MVVM*, que inclusive serviu de principal inspiração para construção desse capítulo, pode ser encontrado na leitura adicional.

O *MVVM* é um padrão de arquitetura que vem ganhando cada vez mais espaço entre os desenvolvedores *Android*, recomendado atualmente pelo *Google* e que ganhou muita força após a *Google I/O* '17 com o anúncio dos *Architecture Components* dentro do *Android Jetpack*, facilitando muito a sua implementação e permitindo com que aplicações de altíssimo nível de arquitetura fossem criadas do zero sem muito esforço inicial, algo que as abordagens anteriores não tinham como principal característica.

Nesse material, utilizaremos diversos exemplos práticos que utilizam esse padrão de arquitetura, utilizando conceitos vistos anteriormente e agregando novas bibliotecas e ferramentas úteis no desenvolvimento de aplicações atuais.

5. Data Binding

Como visto anteriormente, quando utilizamos o padrão de arquitetura *MVVM* uma característica comum é que ele foi desenhado para utilizar funções de *data binding*, independentemente de qual *framework* ou linguagem de programação estamos trabalhando. No *framework Android*, temos a *Data Binding Library* [ref 5], que nos permite utilizar nosso próprio *layout XML* para relacionar os campos de *UI* com a informação que é relevante para cada campo.

Mais detalhes na abordagem do *data binding* utilizando *Kotlin* podem ser vistos no vídeo 1 (ir para vídeo).

5.1. Configurando o projeto

Para esse exemplo, iremos criar um novo projeto com suporte a *Kotlin*, *API* 15 e com uma *EmptyActivity* chamada *"GameInfoActivity"*. Iremos criar uma aplicação simples para exibir as informações de jogos cadastrados. Posteriormente, iremos construir um formulário de cadastro que envia as informações para uma outra tela.

Para iniciar a configuração do projeto com o *data binding* precisamos adicionar algumas linhas no *build.gradle* da aplicação.

```
android {
    // ...
    dataBinding {
        enabled true
    }
}
```

Além disso, como estamos no *Kotlin*, precisamos adicionar suporte ao *Kotlin Annotation Processor*, *kapt*, e adicionar a biblioteca do *data binding*, ainda no *build.gradle* do *app*.

```
// ...
apply plugin: "kotlin-kapt"
// ...
dependencies {
  // ...
  kapt "com.android.databinding:compiler:3.1.3"
```

```
}
```

Com as dependências adicionadas, clique em *'Sync Now'* ou *'Make Project'*. Em algumas versões do *Kotlin* com o *Android Studio*, ao adicionar o *plugin kapt*, sempre que o *sync* é realizado, aparece uma mensagem de erro no *console*.

```
Folder ...\app\build\generated\source\kaptKotlin\debug
Folder ...\app\build\generated\source\kaptKotlin\release
3rd-party Gradle plug-ins may be the cause
```

A princípio esse é um *bug* conhecido e presente no *Android Studio* sempre que o *kapt* é adicionado [ref 6]. Apesar de várias soluções apresentadas [ref 7], a melhor solução temporária é selecionar a opção 'Build | Rebuild Project', porém, sempre que o *sync* for realizado, a mensagem aparecerá novamente, necessitando de um novo *rebuild*. Aparentemente o *bug* foi corrigido a partir da versão 3.2 *Beta* 1 do *Android Studio* [ref 8].

5.2. Configurando o Layout

Com a configuração realizada, basta realizar o *sync* e abrir o arquivo de *layout*, que no caso será o arquivo *'activity_game_info.xml'*. Em todo *layout* que desejamos ativar o *data binding*, precisamos colocá-lo dentro de uma *tag* <*layout*>, isso fará com que o mecanismo de *binding* olhe para os *layouts* apenas quando deixamos isso explícito, evitando chamadas desnecessárias. Com essa modificação, o *layout* ficará assim:

```
<layout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools">
  <data></data>
  <android.support.constraint.ConstraintLayout ...>
     <!-- ... -->
  </android.support.constraint.ConstraintLayout>
</layout>
```

Note que as tags de configuração *xmlns* foram movidas para a *tag <layout>*, realize o mesmo procedimento sempre que atualizar um novo arquivo de *layout* para utilizar com o *data binding*.

5.3. Declarando os dados

Inicialmente iremos construir um *layout* básico, apenas para entender o conceito do *data binding*. Dentro do *ConstraintLayout*, insira um *TextView* que receberá o nome do jogo.

```
<?xml version="1.0" encoding="utf-8"?>
<layout ...>
  <data></data>
  <android.support.constraint.ConstraintLayout ...>
     <TextView ...
        tools:text="Game's Name" />
  </android.support.constraint.ConstraintLayout>
</layout>
```

Em seguida, devemos preparar o *data binding* para entender qual a estrutura da informação que será preenchida na *view*. Para isso, devemos construir nossa classe de dados. Crie uma nova classe *Kotlin* no pacote principal da aplicação chamada *Game*, que possua um campo do tipo *String* chamado *name*:

```
data class Game (val name: String)
```

Com a classe criada, volte para o arquivo de *layout* e dentro da *tag <data>* e adicione uma nova *tag <variable>* que referencie a nova classe criada.

```
<?xml version="1.0" encoding="utf-8"?>
<layout ...>
  <data>
    <variable
       name="game"
       type="app_package.Game" />
  </data>
  <android.support.constraint.ConstraintLayout ...>
```

```
    <TextView ...
        tools:text="Game's Name" />
  </android.support.constraint.ConstraintLayout>
</layout>
```

Com isso, podemos utilizar qualquer campo da classe *Game* em nossas *views*, sempre iniciando com o símbolo @ e realizando a declaração dentro de chaves **{ }**. Dentro das chaves, inserimos o caminho do valor que queremos, ou seja, para acessar o campo *name* da classe *Game*, utilizaríamos a declaração **@{game.name}** direto no atributo *android:text*.

```
<?xml version="1.0" encoding="utf-8"?>
<layout ...>
  <data>
    <variable
       name="game"
       type="app_package.Game" />
  </data>
  <android.support.constraint.ConstraintLayout ...>
     <TextView ...
        android:text="@{game.name}"
        tools:text="Game's Name" />
  </android.support.constraint.ConstraintLayout>
</layout>
```

5.4. Vinculando os dados declarados

Com o *layout* assinalado, precisamos fazer a vinculação dele com o *data binding*, onde devemos declarar alguns códigos na classe da *Activity*. Para que o *data binding* gere os códigos, clique na opção *'Make Project'*, assim que terminar abra a classe da *Activity*.

No método *onCreate()*, temos a chamada do método *setContentView()* que define o arquivo de *layout* que será exibido para essa *Activity*. Quando a compilação foi feita, o *data binding* gerou uma classe automática de *binding*, que usa o nome do *layout xml* como referência. Como no nosso caso o *layout* se chama *'activity_game_info.xml'*, a classe gerada se chama *'ActivityGameInfoBinding'*. Para realizar a vinculação, devemos alterar a chamada do *setContentView()*.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_game_info)
  val binding: ActivityGameInfoBinding =
        DataBindingUtil.setContentView(
              this,
              R.layout.activity_game_info
        )
}
```

A partir de agora, sempre que quisermos definir alguma informação no *layout* utilizando o sistema de *data binding*, devemos utilizar a variável *binding* passando o objeto que ela está esperando. No nosso caso, declaramos uma variável no *xml* chamada *game* que referência a *data class Game* criada. Para ativarmos o *binding*, basta criar um novo objeto dessa classe e passar para o *binding*.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  val binding: ActivityGameInfoBinding =
        DataBindingUtil.setContentView(
              this,
              R.layout.activity_game_info
        )
  val game = Game("Donkey Kong")
  binding.game = game
}
```

Com isso reduzimos a necessidade de pegar a instância de cada *view* através de um *ID* e também de definir individualmente os valores. A princípio pode não parece algo tão útil, porém, imagina em uma aplicação com mais de 30 *views*, pegar a instância de cada uma e definir os valores geralmente se torna algo extremamente complicado e difícil de organizar.

Como podemos perceber, nessa declaração feita a variável *binding* está dentro do escopo do método *onCreate()*. Para que ela esteja acessível em qualquer método da classe, devemos transformá-la em um *field* da classe.

```
var binding: ActivityGameInfoBinding

override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  binding =
        DataBindingUtil.setContentView(
              this,
              R.layout.activity_game_info
        )
  val game = Game("Donkey Kong")
  binding.game = game
}
```

Se tentarmos essa abordagem, receberemos um erro, afinal, o *Kotlin* não aceitará essa declaração pois a variável *binding* deve ser inicializada. Podemos então inicializar como nula, deixando isso explícito no tipo da variável.

```
var binding: ActivityGameInfoBinding? = null

override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  binding =
        DataBindingUtil.setContentView(
              this,
              R.layout.activity_game_info
        )

  val game = Game("Donkey Kong")
  binding?.game = game
}
```

Isso até será aceito, porém, o problema com essa abordagem é que para o *Kotlin* não é interessante que tenhamos variáveis que aceitam valores nulos, afinal, toda vez que fossemos utilizar teríamos que checar com o símbolo de *safe-call* '?', o que não é muito agradável.

Poderíamos então utilizar a propriedade *lateinit*, que deixa explícito para o *Kotlin* que iremos iniciá-la posteriormente em algum lugar do código.

```
lateinit var binding: ActivityGameInfoBinding

override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  binding =
        DataBindingUtil.setContentView(
              this,
              R.layout.activity_game_info
        )
  val game = Game("Donkey Kong")
  binding.game = game
}
```

Parece bom. Não precisaríamos do símbolo de *safe-call* e a variável também não teria valor nulo definido. Porém, ainda temos uma variável *var*, ou seja, que pode sofrer mutação de valor, o que não é ideal para o nosso caso.

Nesse caso, para tornar a variável *val*, podemos inicializá-la utilizando a declaração *lazy*, que fará exatamente o que estamos fazendo com o *lateinit*, porém, permitirá que a variável seja imutável.

```
val binding: ActivityGameInfoBinding by lazy {
  DataBindingUtil.setContentView<ActivityGameInfoBinding>(
        this,
        R.layout.activity_game_info
  )
}

override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  val game = Game("Donkey Kong")
  binding.game = game
}
```

O resultado está como esperado, no entanto, isso ainda sim é um transtorno de ter que escrever esse *boilerplate* em todas as *Activities* que usam o *data binding*. Para melhorar um pouco a declaração, podemos escrever nosso próprio *Delegate*. Vamos iniciar sua criação em um novo arquivo, que chamaremos de *'Delegates.kt'*.

Nossa aplicação está começando a ganhar volume, seria interessante se criássemos alguns pacotes para melhorar a organização. Crie um pacote *model* para a classe *Game*; Um pacote *ui.gameinfo* para a classe *GameInfoActivity* e um pacote *utils* para o arquivo *Delegates*.

No arquivo *Delegates*, criamos uma classe chamada *SetContentView* que receberá o *layout* e retornará o *binding*.

```
class SetContentView<out T : ViewDataBinding>(
  @LayoutRes private val layoutRes: Int) {

  operator fun getValue(thisRef: Activity, property:
KProperty<*>): T {
     return DataBindingUtil.setContentView(thisRef,
layoutRes)
  }
}
```

Para melhorar essa declaração e evitar que uma nova instância no *layout* com o *binding* seja criada sempre, podemos permitir com que a aplicação reutilize a instância tornando-a propriedade da classe.

```
class SetContentView<out T : ViewDataBinding>(
  @LayoutRes private val layoutRes: Int) {

  private var value : T? = null

  operator fun getValue(thisRef: Activity, property:
KProperty<*>): T {
     value = value ?: DataBindingUtil.setContentView(thisRef,
layoutRes)
     return value!!
  }
}
```

E atualizamos também a chamada:

```
val binding: ActivityGameInfoBinding by
SetContentView(R.layout.activity_game_info)
```

Por fim, podemos criar uma função que será responsável por chamar a classe e terminar com uma declaração muito parecida com o *lazy* demonstrado anteriormente.

```
fun <T : ViewDataBinding> contentView(@LayoutRes layoutRes:
Int): SetContentView<T> {
  return SetContentView(layoutRes)
}

class SetContentView<out T : ViewDataBinding>(
  @LayoutRes private val layoutRes: Int) {

  private var value : T? = null

  operator fun getValue(thisRef: Activity, property:
KProperty<*>): T {
     value = value ?: DataBindingUtil.setContentView(thisRef,
layoutRes)
     return value!!
     //...
  }
}
```

```
val binding: ActivityGameInfoBinding by
contentView(R.layout.activity_game_info)
```

5.5. Binding Adapters customizados

Uma outra funcionalidade interessante do *data binding* é a criação de atributos de *xml* customizados, que podemos referenciar como *binding adapters*. Isso significa que podemos programar para um atributo *xml* completamente novo

que realize uma funcionalidade escrita por nós. Prosseguindo com nossa aplicação, vamos adicionar um pouco mais de informação à nossa classe de jogos.

```
data class Game (val name: String,
                 val launchYear: Int,
                 val imageUrl: String)
```

Como podemos observar, um dos atributos recebe a *url* da imagem, que podemos utilizar uma biblioteca de carregamento de imagens para realizar o *load*. Uma maneira muito mágica seria se pudéssemos realizar a declaração direto no *xml* e ele soubesse como lidar com isso, certo? Com o *data binding* isso é possível através dos *binding adapters* customizados.

```
<ImageView
  android:id="@+id/ivImage"
  app:image="@{game.imageUrl}" />
```

Para fazer essa declaração funcionar existem duas maneiras possíveis. A primeira é declarar uma função em qualquer lugar do código e anotá-la com **@BindingApdater("nome_atributo")**.

Crie um arquivo chamado *'BindingAdapters.kt'* no pacote *utils* e declare uma função capaz de receber uma *ImageView* e uma *String* e realizar o carregamento da imagem.

```
fun loadImage(imageView: ImageView, imageUrl: String?) {
  Picasso.get().load(imageUrl).into(imageView)
}
```

Conforme dito anteriormente, para que o *data binding* associe essa função declarada, precisamos deixar isso explícito.

```
@BindingAdapter("image")
fun loadImage(imageView: ImageView, imageUrl: String?) {
  Picasso.get().load(imageUrl).into(imageView)
}
```

Se você quiser tornar essa declaração um pouco mais organizada, é possível colocá-la dentro de uma classe, porém, essa declaração deve ser estática, ou seja, no caso do *Kotlin* precisamos colocar dentro de um *companion object* e também anotar o método com a anotação **@JvmStatic**.

```
class BindingAdapters {
  companion object {
     @JvmStatic @BindingAdapter("image")
     fun loadImage(imageView: ImageView, imageUrl: String?) {
        Picasso.get().load(imageUrl).into(imageView)
     }
  }
}
```

Note que em ambos os casos estamos utilizando a biblioteca de carregamento de imagens *Picasso*, portanto, para que esse exemplo funcione devemos importá-la em nosso *build.gradle* do *app*.

```
implementation "com.squareup.picasso:picasso:2.71828"
```

Como estamos fazendo requisições na internet, também precisamos deixar isso explícito no manifesto da aplicação.

```
<uses-permission
    android:name="android.permission.INTERNET" />
```

5.6. Detalhes de Layout

Existem diversas outras possibilidades para adicionar no conteúdo das propriedades do *layout*. Com o *data binding*, podemos acessar funcionalidades até então presentes apenas no *Java* ou no *Kotlin*. Para incluir o valor de uma variável inteira, por exemplo, devemos fazer a conversão.

```
<TextView ...
```

```
  android:text="@{String.valueOf(game.launchYear)}"
  tools:text="Year" />
```

Também é possível usar expressões para manipular elementos comuns do *layout*. Para demonstrar, criaremos um novo *TextView* que só aparecerá para os jogos que são lançamentos do ano.

```
<TextView ...
  android:visibility="@{game.launchYear < 2000 ? View.VISIBLE
: View.GONE}"
  android:text="@string/classic" />
```

Em alguns casos, alguns símbolos podem causar erros de compilação, portanto, para contornar esse problema, precisamos utilizar a forma 'escapada' desses símbolos. No caso do `<`, essa forma seria `<`.

Além disso, note que estamos utilizando duas constantes da classe *View*, para que isso funcione corretamente, precisamos importar a classe *View* dentro das *tags* <*data*> do *layout*.

```
<data>
  <import type="android.view.View" />
  <!-- ... -->
</data>
```

Apesar de podermos utilizar expressões lógicas de comparação nos atributos de *xml*, muitos desenvolvedores não recomendam essa prática, alegando que devemos deixar essa tarefa para a própria linguagem de programação. Seguindo essa boa prática, podemos transformar essa comparação em um atributo da classe *Game* e apenas verificar seu valor para definir a visibilidade.

```
data class Game (val name: String,
                 val launchYear: Int,
                 val imageUrl: String) {
  val isClassic = launchYear < 2000
}
```

```
android:visibility="@{game.isClassic ? View.VISIBLE :
View.GONE}"
```

Acessar recursos também é possível, facilitando a manipulação de valores pré-definidos, sem precisar ficar criando recursos exclusivos para determinadas situações.

```
android:padding="@{large ? @dimen/largePadding :
@dimen/smallPadding}"

android:text="@{@string/nameFormat(firstName, lastName)}"

android:text="@{@plurals/banana(bananaCount)}"
```

5.7. Objetos observáveis

Agora que possuímos uma noção do que é possível com *data binding* direto no *xml*, chegou a hora de entender o verdadeiro poder dessa ferramenta: os objetos observáveis. Eles basicamente permitem que qualquer alteração realizada no conteúdo da variável seja refletida automaticamente na *view* correspondente.

Para começar, vamos supor que na nossa classe *Game* temos um novo valor para a nota de avaliação do jogo.

```
data class Game (val name: String,
                 val launchYear: Int,
                 val imageUrl: String,
                 var rating: Double) {
  val isClassic = launchYear < 2000
}
```

Note que a variável é mutável, portanto, deve ser marcado como *var*. No entanto, para notificar o *data binding* que essa variável foi alterada e que ele deve alterar sua *view*, precisamos sobrescrever seu *setter*. Para fazer isso no *Kotlin*, precisamos estar fora do construtor, então devemos remover a marcação *var* do construtor, para tornar o recebimento dela local e transformamos em um *field* da classe dentro de seu escopo. Um outro detalhe importante é que para fazer isso,

não podemos mais que a classe seja uma *data class*, portanto, transformaremos ela em uma classe normal. Por fim, na declaração do *setter*, checamos se o valor foi realmente alterado (para evitar gastos de recurso de a toa) e notificamos a alteração através do método *notifyPropertyChanged()* que estará disponível para a classe assim que estendermos a classe *BaseObservable()*. Um outro detalhe importante é que a declaração deve possuir a anotação **@Bindable** e também deve ter a declaração do *get*, mesmo que ele não vá ser redeclarado.

```
data class Game (val name: String,
                 val launchYear: Int,
                 val imageUrl: String,
                 rating: Double) : BaseObservable() {
  val isClassic = launchYear < 2000

  var rating = rating
     @Bindable get
     set(value) {
        if (field != value) {
           field = value
           notifyPropertyChanged(BR.rating)
        }
     }
}
```

Note que o método *notifyPropertyChanged()* precisa de um argumento que no caso é o *BR.rating*. O *BR* é uma classe semelhante ao *R* presente no *Android* que é gerada automaticamente pelo *data binding*. Ele irá perceber as propriedades que podem ser alteradas dentro da classe referenciada e irá criar atributos para elas dentro da classe *BR* em tempo de compilação. Faça o *compile* da aplicação para que a propriedade *BR.rating* seja gerada.

A outra forma de observar alteração de valor em objetos é através dos *ObservableFields* entre os seguintes tipos disponíveis:

- ObservableField
- ObservableBoolean
- ObservableByte
- ObservableChar
- ObservableShort
- ObservableInt

- ObservableLong
- ObservableFloat
- ObservableDouble
- ObservableParcelable

A declaração de uma classe pode ser feitas de várias maneiras. A primeira é recebendo direto os valores do tipo *ObservableSomething*, onde as propriedades podem ser *val*, pois os valores internos delas que serão alterados, e não suas instâncias em si. Além disso, como não precisamos reescrever o *setter*, podemos voltar a ser uma *data class*.

```
data class Game(val name: ObservableField<String>,
                val launchYear: ObservableInt,
                val imageUrl: ObservableField<String>,
                val rating: ObservableDouble)
```

A alteração dos valores da classe seriam feitos individualmente para cada atributo, da seguinte maneira:

```
game.rating.set(4.8)
game.rating.get()
```

Porém, com a classe declarada dessa maneira, a construção de uma instância ficaria um pouco complicada:

```
val game = Game(ObservableField("Donkey Kong"),
                ObservableInt(1981),
                ObservableField("http://..."),
                ObservableDouble(4.5))
```

Para melhorar, podemos abrir mão de ser uma *data class* e receber valores comuns pelo construtor, iniciando as propriedades posteriormente conforme os valores recebidos.

```
class Game(name: String,
           launchYear: Int,
```

```
            imageUrl: String,
            rating: Double) {

  val name = ObservableField<String>(name)
  val launchYear = ObservableInt(launchYear)
  val imageUrl = ObservableField<String>(imageUrl)
  val rating = ObservableDouble(rating)
}
```

Dessa forma, a construção da classe fica muito mais fácil e legível.

```
val game = Game("Donkey Kong",
                1981,
                "http://...",
                4.5)

game.rating.set(4.8)
game.rating.get()
```

Além disso, é possível criar um *Delegate* (assim como fizemos anteriormente) para simplificar ainda mais essa declaração de atualização de campos. Abrindo o arquivo *Delegates.kt*, podemos escrever a seguinte declaração:

```
fun <R : BaseObservable, T : Any> bindable(
     value: T, bindingRes: Int): BindableDelegate<R, T> {
  return BindableDelegate(value, bindingRes)
}

class BindableDelegate<in R : BaseObservable, T : Any>(
     private var value: T, private val bindingEntry: Int) {
  operator fun getValue(thisRef: R, property: KProperty<*>):
T = value

  operator fun setValue(thisRef: R, property: KProperty<*>,
value: T) {
     this.value = value
     thisRef.notifyPropertyChanged(bindingEntry)
```

```
    }
}
```

Dessa forma, podemos declarar as propriedades da classe de uma maneira muito mais fácil, semelhante à declaração de *lazy*.

```
class Game(val name: String,
           val launchYear: Int,
           val imageUrl: String,
           rating: Double) : BaseObservable() {
  // ...
  @get:Bindable
  var rating by bindable(rating, BR.rating)
}
```

5.7.1. Two-way binding

Além das objetos observáveis com alterações feitas diretas pelo código, existe também o *two-way binding*, que basicamente funcionará para os casos contrários, onde a informação de alguma *view* é alterada pelo usuário, por exemplo. Para que o *data binding* saiba que queremos que ele observe alterações nesse campo e chame os métodos de *setter* que definimos, precisamos deixar isso explícito adicionando o símbolo `=` na declaração do *binding*, como por exemplo **@={game.name}**, para simbolizar que qualquer alteração nesse campo irá afetar diretamente o *field* name do objeto *game*.

```
<EditText ...
  android:hint="@string/game_name"
  android:text="@={game.name}" />
```

Lembre-se também de que a variável em questão precisa estar preparada para receber o *binding*, além de ser mutável.

```
class Game(name: String,
           val launchYear: Int,
           val imageUrl: String,
```

```
            rating: Double) : BaseObservable() {
  // ...
  @get:Bindable
  var name by bindable(name, BR.name)
  // ...
}
```

5.8. Tratamento de eventos

Com *data binding* também é possível tratar eventos direto no *xml*, algo que não é tão novidade assim, pois afinal temos a propriedade *onClick* que chama um método declarado na *Activity*, porém, a diferença aqui é a maneira com que isso é executado e também as diferentes possibilidades e atributos existentes nessa abordagem.

Vale ressaltar que essa abordagem possui muitas controvérsias, sendo que alguns desenvolvedores adoram a funcionalidade enquanto outros odeiam. Ao mesmo tempo que ela pode evitar o uso de *boilerplate* na declaração de *listeners* ela também pode complicar um pouco para estruturar de uma maneira organizada as solicitações declaradas.

Primeiro, podemos declarar uma nova maneira de lidar com os eventos de clique chamando diretamente o método de uma classe referenciada.

```
class MyHandlers {
  fun onClickFriend(view: View) {  }
}
```

```
<layout>
  <data>
    <variable
       name="handlers"
       type="package.Handlers" />
  </data>
  // ...
    <TextView ...
       android:onClick="@{handlers::onClickFriend}" />
// ...
```

```
</layout>
```

Também é possível realizar a declaração como *listener*.

```
class MyHandlers {
  fun onClickFriend(view: View) {  }
  fun onSaveClick(task: Task) {  }
}
```

```
<layout>
  <data>
    <variable
       name="handlers"
       type="package.Handlers" />
  </data>
   // ...
   <TextView ...
     android:onClick="@{() -> handlers.onSaveClick(task)}" />
// ...
</layout>
```

Um outro problema nessa abordagem é que o *Android Studio* não consegue indexar o nome dos métodos que você está referenciando, então precisa tomar bastante cuidado durante as declarações para não declarar de maneira errada.

Por fim, existem diversos outros usos interessantes para o *data binding* em diversas maneiras, se tiver interessado em saber mais sobre o tópico recomendo que assista o vídeo 2 (ir para vídeo).

No próximo capítulo, abordaremos o assunto *RxJava* e continuaremos o desenvolvimento desse projeto que começamos no capítulo de *data binding*. Como a declaração dos *layouts* aqui foi feita de maneira encurtada para melhorar o fluxo de entendimento, abaixo segue o *layout* completo da *GameInfoActivity*.

```
<?xml version="1.0" encoding="utf-8"?>
<layout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
```

```
  xmlns:tools="http://schemas.android.com/tools">

  <data>
     <import type="android.view.View" />

     <variable
        name="game"

type="br.com.paulosalvatore.databindingtests.model.Game" />
  </data>

  <android.support.constraint.ConstraintLayout
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     tools:context=".ui.gameinfo.GameInfoActivity">

     <ImageView
        app:image="@{game.imageUrl}"
        android:contentDescription="@string/game_image"
        android:id="@+id/ivImage"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:layout_marginLeft="16dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="8dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

     <TextView
        android:id="@+id/tvName"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginStart="8dp"
        android:text="@{game.name}"
        app:layout_constraintEnd_toEndOf="parent"
```

```
        app:layout_constraintStart_toEndOf="@+id/ivImage"
        app:layout_constraintTop_toTopOf="@+id/ivImage"
        tools:text="Game Name" />

    <TextView
        android:id="@+id/tvYear"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginStart="8dp"
        android:text="@{String.valueOf(game.launchYear)}"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/ivImage"
        app:layout_constraintTop_toBottomOf="@+id/tvName"
        tools:text="Year" />

    <TextView
        android:id="@+id/tvClassic"
        android:textColor="#8e0909"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginStart="8dp"
        android:visibility="@{game.isClassic ? View.VISIBLE :
View.GONE}"
        android:text="@string/classic"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/ivImage"
        app:layout_constraintTop_toBottomOf="@+id/tvYear" />

    <TextView
        android:id="@+id/tvRating"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
```

```
        android:layout_marginEnd="8dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginStart="8dp"
        android:text="@{String.valueOf(game.rating)}"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/ivImage"
        app:layout_constraintTop_toBottomOf="@+id/tvClassic"
        tools:text="Rating" />

    <EditText
        android:id="@+id/etName"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:hint="@string/game_name"
        android:text="@={game.name}"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/ivImage" />
  </android.support.constraint.ConstraintLayout>
</layout>
```

E também o arquivo de *resources strings.xml*.

```
<resources>
  <string name="app_name">Game Info</string>
  <string name="add_game">Add Game</string>
  <string name="name">Name</string>
  <string name="launch_year">Launch Year</string>
  <string name="add">Add</string>
  <string name="clear">Clear</string>
```

```
  <string
name="title_activity_game_info">GameInfoActivity</string>
  <string name="game_information">Game Information</string>
  <string name="game_image">Game Image</string>
  <string name="classic">Classic</string>
  <string name="game_name">Game Name</string>
  <string name="image_url">Image URL</string>
  <string name="rating">Rating</string>
</resources>
```

E por fim, o arquivo *resources* de *styles*:

```
<resources>
  <!-- ... -->
  <style name="LabelTitle">
     <item name="android:gravity">center</item>
     <item
name="android:textColor">@color/colorPrimary</item>
     <item name="android:textSize">20sp</item>
     <item name="android:textStyle">bold</item>
  </style>

  <style name="LabelField">
     <item
name="android:textColor">@color/colorPrimary</item>
     <item name="android:textSize">14sp</item>
     <item name="android:textStyle">bold</item>
  </style>

  <style name="LabelFieldInfo">
     <item name="android:textSize">14sp</item>
  </style>

  <style name="Button">
     <item
name="android:background">@color/colorPrimaryDark</item>
     <item name="android:textColor">#FFFFFF</item>
```

```
  </style>
</resources>
```

6. Reactive Programming com RxJava

Pesquisando o assunto *Reactive Programming* encontramos milhares de descrições e maneiras de exemplificar, todas muito complexas e lotadas de termos técnicos [ref 9], difíceis de entender quando estamos começando a explorar esse tópico. Para sintetizar e tornar algo simples de entender, *reactive programming* é uma forma de programar se preocupando com o fluxo de dados e a propagação de mudanças, ou seja, estamos sempre preocupados com a informação que está vindo e a mudança rápida nessa informação, utilizando uma abordagem com extrema eficiência permitindo um ganho de performance absurdamente superior às abordagens utilizadas anteriormente [ref 10].

Outra grande característica dessa abordagem é que ela facilita a utilização de processos paralelos e de forma assíncrona, que é muito difícil de implementar em um primeiro momento, pois a comunicação desses processos pode se tornar algo bem complexo e o tratamento de eventuais erros dificulta ainda mais o processo, o que torna a curva de aprendizado um pouco íngreme.

Foi então que o pessoal da *Netflix* [ref 11] iniciou a construção de uma biblioteca que permitia qualquer projeto utilizar essa abordagem de maneira mais fácil e simplificada, que contribuiu para o fomento do desenvolvimento RX, fazendo com que o assunto fosse abordado em diversos eventos, incluindo NetflixOSS Meetup, edição de 2014, que falou com muitos detalhes sobre os temas *Reactive* e *Async*, conteúdo que pode ser visto no vídeo 3 (ir para vídeo).

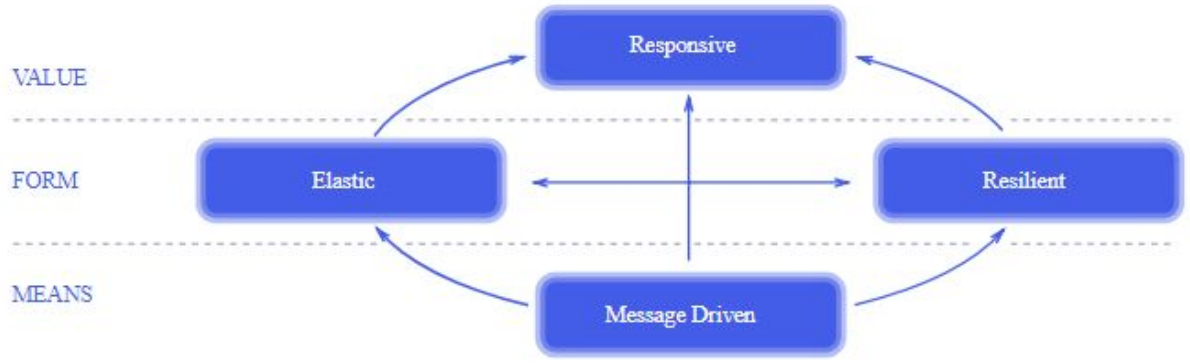
Justamente com esse objetivo, surgiu a *Reactive Extensions*, ou simplesmente *ReactiveX* [ref 12], uma biblioteca para construção de aplicações que funcionem de maneira assíncrona, baseada em eventos utilizando o que chamamos de "sequências observáveis", que irei descrever um pouco mais a frente. Inicialmente a biblioteca estava disponibilizada no repositório da *Netflix* no *Github* e posteriormente, devido a sua importância, foi movida para um repositório exclusivo, levando junto todas as bibliotecas de linguagens de programação específicas, incluindo *RxJava* e *RxAndroid*, que utilizamos durante o desenvolvimento no *framework Android*.

6.1. Pilares da Reactive Programming

De acordo com o "manifesto reativo" [ref 13], publicado em 2014, a programação reativa possui 4 pilares principais, conforme é possível visualizar na figura 4.

1. **Responsivo:** Deve responder o mais rápido possível, onde problemas devem ser detectados e resolvidos com a máxima eficácia;
2. **Resiliente:** Continua respondendo em caso de falha;
3. **Elástico:** Continua responsivo mesmo com aumento da demanda;
4. **Orientado a Mensagens:** Usa passagem de mensagens assíncronas garantindo baixo acoplamento, isolamento e provê meios para tratamento de erros através de mensagens.

Figura 4: Os 4 pilares da *Reactive Programming*



Fonte: *The Reactive Manifesto* [ref 13]

6.2. Fluxo de Dados Assíncrono

Como dito anteriormente, a programação reativa trabalha em cima de um fluxo de dados assíncrono, ou seja, não se sabe ao certo em que ordem nem qual momento as informações virão, assume-se que devemos estar preparados para recebê-las a qualquer momento.

Dados assíncronos acontecem o tempo inteiro no *flow* comum de uma aplicação, por exemplo:

- Eventos de clique
- Ler um arquivo
- Receber uma *Push Notification*

- Acessar um *database*
- Receber um *input* de teclado
- Atualização de um sensor do dispositivo

Pegando a atualização de um sensor do dispositivo como o de localização, por exemplo, quando estamos conectados ao *GPS*. Quando o usuário está se movendo, iremos obter diferentes valores conforme o tempo passa. Inicialmente podemos um valor específico de latitude e de longitude; de repente ele se move e esse valor altera. Um pouco mais de tempo se passa e temos um novo valor. Isso pode ser considerado um dado assíncrono.

Um outro exemplo seria uma resposta de um servidor. Diferente da localização, recebemos a informação de uma única vez, ou seja, uma requisição é feita ao servidor e a resposta é recebida logo em sequência. Mesmo que aconteça dessa maneira, isso também é considerado um dado assíncrono.

Quando enviamos uma requisição para um servidor, estamos sempre esperando uma resposta. Vamos supor que iniciamos uma requisição e recebemos o que estávamos esperando. Tratamos a resposta e prosseguimos com a aplicação. Agora realizamos uma outra requisição idêntica, porém, dessa vez recebemos um erro. Mesmo nesse caso, o erro continua sendo uma resposta válida, devemos tratá-lo e prosseguir com a aplicação.

Esse tratamento da maneira que as coisas são feitas é o mais importante, e o principal é que podemos utilizar a programação reativa para isso, mais especificamente no nosso caso a *ReactiveX* com *RxJava*, *RxAndroid* e *RxKotlin*, fornecendo ferramentas exclusivas que potencializam nosso desenvolvimento.

A primeira pergunta que fazemos é: "Por que usar *RxJava* quando eu já sei como lidar com esses tipos de eventos?". A resposta para essa pergunta é simples, você não precisa usar *RxJava*. Como qualquer outra tecnologia tem seus prós e contras, no entanto, existem diversas razões que agregam valor ao *software* e ao desenvolvedor que utilizam essa abordagem.

6.3. Razões para utilizar Rx Programming

Entre as diversas funcionalidades possíveis com a programação reativa, existem algumas razões para utilizá-la em vez de outras abordagens conhecidas. Como o *Rx* trata tudo de forma assíncrona, realizar as tarefas normais passa a ser um processo muito mais rápido e eficiente.

Neste capítulo listarei algumas funcionalidades interessantes da biblioteca e veremos alguns exemplos aplicados.

6.3.1. Data Transformation

Data Transformation consiste em receber uma informação e transformá-la em outra baseado em nossa implementação, fazendo isso de forma assíncrona. Por exemplo: recebemos uma lista de usuários e obtemos as publicações de cada usuário no *twitter*, transformando enfim uma lista de usuários em uma lista de *tweets*.

6.3.2. Chaining

Chaining consiste em pegar a informação e aplicar um filtro, separando apenas o que é útil e interessante e que desejamos processar, economizando recursos antes mesmo de iniciar os trabalhos.

6.3.3. Abstraction

RxJava atua como uma camada de abstração (*abstraction layer*), ou seja, olhando por baixos do panos, há uma série de códigos *Java* que fazem o trabalho para nós, abstraindo uma densa camada de código e facilitando o processo de utilização dessa lógica de tratamento de eventos assíncronos.

6.3.4. Flexible Threading

Como venho repetindo diversas vezes, a programação reativa resume-se ao fluxo de dados assíncrono. Para que isso seja feito de maneira eficaz é necessário que o sistema possua uma grande flexibilidade para realizar os trabalhos em um excelente sistema de *threading* gerenciável, podendo facilmente decidir de qualquer forma e onde esse processamento será feito e onde o resultado deve ser visualizado.

6.3.4.1. Schedulers

O sistema flexível de *threading* só é possível por conta de um componente chamado *Schedulers*, que basicamente direciona um *Observable* durante a realização do trabalho, tanto durante o processamento da tarefa quanto na conclusão dela, alterando facilmente o local de realização e manipulação da informação.

6.4. Configurando o projeto

Para começarmos a testar algumas implementações com o *ReactiveX* através do *RxJava*, é necessário configurar o projeto para importar as bibliotecas necessárias. Para nosso primeiros testes com a biblioteca iremos utilizar o projeto criado anteriormente no capítulo de *Data Binding*.

Para configurar a biblioteca, abra o arquivo *build.gradle* do *app* e adicione o seguinte conteúdo:

```
// Rx
implementation "io.reactivex.rxjava2:rxjava:2.1.15"
```

Com isso já é possível utilizar a biblioteca, no entanto, existem algumas bibliotecas adicionais que trazem algumas funções específicas para o *framework Android* e também para o *Kotlin*. Adicione também as seguintes dependências:

```
implementation "io.reactivex.rxjava2:rxandroid:2.0.2"
implementation "io.reactivex.rxjava2:rxkotlin:2.2.0"
```

Faça o *sync* do projeto e estaremos prontos para iniciar nossos exemplos práticos. Lembre-se de utilizar a opção *Rebuild Project* caso apresente erros por conta do *kapt*.

6.5. A base do ReactiveX

Para pensar de forma reativa, precisamos entender a base do *ReactiveX*, que basicamente é composta de três componentes principais: os *Observables*, os *Observers* e os *Operators*. Além dessa base, posteriormente é interessante entender o que são os *Schedulers* e qual a sua importância para a realização das tarefas.

6.5.1. Observable

O primeiro item da nossa lista é o *Observable*, onde tudo começa. É nele que definimos o que será realizado, a base da nossa operação. Podemos definir um *Observable* como algo que está produzindo algum fluxo de eventos, podendo ser qualquer coisa que estamos interessados em saber mais sobre.

Um *Observable* pode ser definido entre *Hot* ou *Cold*.

6.5.1.1. Hot Observable

Quando temos um *Hot Observable* significa que esse *observable* começa a produzir eventos assim que ele é criado, sem se importar se alguém está preocupado com sua realização ou não, ele simplesmente produz informação.

Alguns eventos que podem ser considerados *hot observables*, como o clique de um usuário, a entrada de dados de um teclado, uma atualização de um sensor de localização, entre outros.

6.5.1.2. Cold Observable

Já um *Cold Observable* é o oposto, ele basicamente é criado e só iniciará de fato a realização do trabalho quando for solicitado, pacientemente aguardando para que o *Subscriber* solicite suas tarefas.

Alguns eventos que podem ser considerados *cold observables*, como acessar o conteúdo de um arquivo, ou alguma informação presente no banco de dados, entre outros.

6.5.1.3. Declarando um Observable

Para iniciar a programação reativa geralmente partimos de um *Observable*, capaz de fornecer um fluxo de dados inicial que será trabalhado de forma assíncrona. A maneira mais simples de obter um *Observable* é utilizando o *create()*, que implementa a declaração do *subscriber*, responsável por realizar as chamadas de operações longas a serem feitas de maneira assíncrona.

```
Observable.create<String> { subscriber -> }
```

Também é possível utilizar o método *just()* para pegar alguns itens e transformá-los em *observables*.

```
Observable.just(1, 2, 3)
```

O método *interval()* basicamente te fornecerá números a cada determinado período de tempo, onde a unidade de medida pode ser referenciada pela classe *TimeUnit*.

```
Observable.interval(2, TimeUnit.SECONDS)
```

Independente da declaração, para que o *observable* realize suas tarefas, devemos chamar o método *subscribe()*, implementando ou chamando a operação que desejamos realizar.

```
Observable.create<String> { subscriber ->
  try {
     val result = "LongOperationResult"
     subscriber.onNext(result)
     subscriber.onComplete()
  } catch (e: Exception) {
     subscriber.onError(e)
  }
}.subscribe { println("create: $it") }
```

6.5.2. Observer

Através de um *Observer* o *RxJava* permite com que uma operação que está sendo feita pelo *Observable* seja 'observada'. Ele consiste em uma interface básica com quatro métodos que nos permite mapear exatamente o que aconteceu durante a operação.

Se olharmos para a declaração da interface em *Java*, podemos observar exatamente o seguinte:

```
public interface Observer<T> {
   void onSubscribe(@NonNull Disposable d);
   void onNext(@NonNull T t);
   void onError(@NonNull Throwable e);
   void onComplete();
}
```

Inicialmente a interface recebe um tipo **T**, significando o tipo de variável que ele irá observar.

Depois, temos o método *onSubscribe()*, que é chamado assim que o *Observable* começa a ser observado. O objeto da classe *Disposable* é o que é retornado quando declaramos o método *subscribe()*.

O principal entre eles é o *onNext()*, que recebe cada item informado do tipo **T**, e é onde utilizaremos o valor já processado.

Um dos métodos mais interessantes também é o *onError()*, que atua como um centralizador de qualquer erro ocasionado durante a operação realizada, onde podemos declarar a lógica para tratar esses erros recebidos.

Por último temos o *onComplete()*, que basicamente é chamado assim que toda a operação termina.

Sua implementação completa em *Kotlin* ficaria assim:

```
val observer = object : Observer<Int> {
  override fun onError(e: Throwable) {
     Log.e(tag, "onError", e)
  }
  override fun onComplete() {
     Log.i(tag, "onComplete")
  }
  override fun onNext(t: Int) {
     Log.i(tag, "onNext: $t")
  }
  override fun onSubscribe(d: Disposable) {
     Log.i(tag, "onSubscribe")
  }
}
```

Se atrelarmos um *Observable* a esse *Observer* declarado, teremos o resultado esperado.

```
val observable = Observable.just(1, 2, 3)

val tag = "RxJava"
val observer = object : Observer<Int> {
  override fun onError(e: Throwable) {
     Log.e(tag, "onError", e)
  }
  override fun onComplete() {
     Log.i(tag, "onComplete")
  }
```

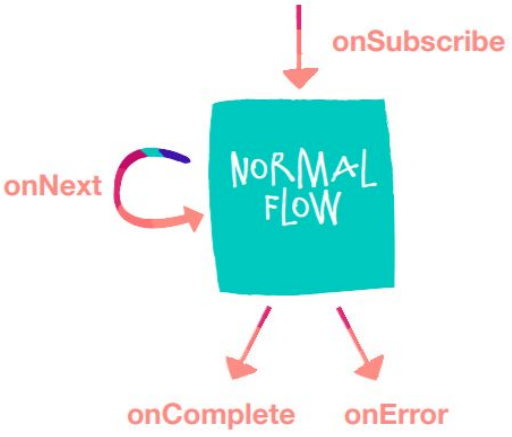
```
    override fun onNext(t: Int) {
        Log.i(tag, "onNext: $t")
    }
    override fun onSubscribe(d: Disposable) {
        Log.i(tag, "onSubscribe")
    }
}

observable.subscribe(observer)
```

6.5.2.1. Observer's Lifecycle

Como é possível perceber pela declaração dos métodos, há um ciclo de vida envolvendo os *Observers*, ou seja, cada método possui uma ordem para acontecer e funcionam de acordo a figura 5.

O ciclo de vida é bem simples: inicialmente temos a chamada do método *onSubscribe()*, que antecipa a execução do *Observable*; depois, entramos no que chamamos de *normal flow*, o fluxo comum da aplicação, chamando o método *onNext()* para cada item recebido; por fim, um dos dois métodos é chamado: em caso de sucesso, o *onComplete()*; e em caso de erro, o *onError()*, encerrando sua execução.

Figura 5: Ciclo de vida de um *Observer*



Fonte: Annyce Davis at *KotlinConf* 2017 [ref 15]

6.5.2.2. Consumer

Também temos a interface *Consumer*, que facilita uma implementação de *subscribe*, onde só implementaremos a passagem no método *onNext()*, que nesse caso é chamado de *accept()*, que receberá o tipo **T** e fará alguma operação. O conteúdo da interface é bem simples, como pode ser observado em sua declaração, em *Java*.

```
public interface Consumer<T> {
    /**
     * Consume the given value.
     * @param t the value
     * @throws Exception on error
     */
    void accept(T t) throws Exception;
}
```

Sua implementação em *Kotlin* ficaria assim:

```
val consumer = object : Consumer<Int> {
  override fun accept(t: Int) {
     Log.i(tag, "accept: $t")
  }
}
```

Se atrelarmos um *Observable* a esse *Consumer* declarado, teremos o resultado esperado.

```
val observable = Observable.just(1, 2, 3)

val tag = "RxJava"
val consumer = object : Consumer<Int> {
  override fun accept(t: Int) {
     Log.i(tag, "onNext: $t")
  }
}
```

```
observable.subscribe(consumer)
```

No *Kotlin*, podemos reduzir essa implementação para uma *lambda function*, aplicando-a direto no método *subscribe()*.

```
val observable = Observable.just(1, 2, 3)
val tag = "RxJava"
observable.subscribe { Log.i(tag, "onNext: $it") }
```

6.5.3. Operators

Entendendo a base do *ReactiveX*, é hora de conhecer um outro atributo muito importante: os operadores. Atualmente existem diversos operadores e cada um com uma funcionalidade específica, uma lista completa deles está disponível na documentação [ref 16], juntamente com o seu funcionamento escrito e também descrito em diagramas.

6.5.3.1. Operator: Map

O funcionamento do operador *map* consiste em transformar uma informação recebida em outra qualquer, de acordo com a nossa declaração.

```
Observable.just(1, 2, 3)
      .map { it * 2 }
      .subscribe { println(it) }
```

Nesse exemplo acima, para cada número recebido, iremos passá-lo pelo *map* e transformá-lo em seu dobro. Como a informação inicial era **"1, 2 e 3"**, o resultado dessa operação será **"2, 4 e 6"**.

Também podemos adicionar a essa declaração um filtro, que irá selecionar apenas os valores interessantes para nós.

```
Observable.just(1, 2, 3)
      .map { it * 2 }
      .filter { it < 6 }
```

```
    .subscribe { println(it) }
```

Nesse caso, o filtro irá permitir apenas números passados pelo *map* que são menores do que 6, retornando então **"2 e 4"** como resultado da operação.

6.5.3.2. Operator: Flatmap

Em um primeiro momento, pode parecer que o operador *Flatmap* realiza a mesma coisa que o *map*, porém, apesar de serem bem parecidos, a diferença é que o *flatmap* transforma um item recebido em um *observable*.

```
val usersObservable // Observable<User>
    val posts: Observable<Post>
    posts = usersObservable.flatMap { it.getUserPosts() }
```

Além do *flatMap*, existe também o *concatMap*, que é muito semelhante em termos de execução com a diferença que esse operador irá preservar a ordem de entrada dos itens, enquanto o *flapMap* retornará o item assim que ele estiver pronto, independente da ordem que entrou.

6.5.4. Exemplo prático: Combine Latest

Agora que estamos entendendo um pouco mais sobre a biblioteca *RxJava* e suas implementações, vamos criar uma funcionalidade muito interessante utilizando o operador *CombineLatest*.

Para iniciar o desenvolvimento do nosso exemplo, vamos configurar um *floating action button* na *GameInfoActivity* que irá abrir uma nova *Activity* de adição de itens. No arquivo de *layout*, adicione o seguinte trecho ao final do arquivo, antes de fechar o *ConstraintLayout*.

```
<android.support.design.widget.FloatingActionButton
  android:id="@+id/fabAdd"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_marginBottom="8dp"
  android:layout_marginEnd="8dp"
  android:layout_marginRight="8dp"
  android:src="@android:drawable/ic_input_add"
```

```
  android:tint="#FFFFFF"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintEnd_toEndOf="parent" />
```

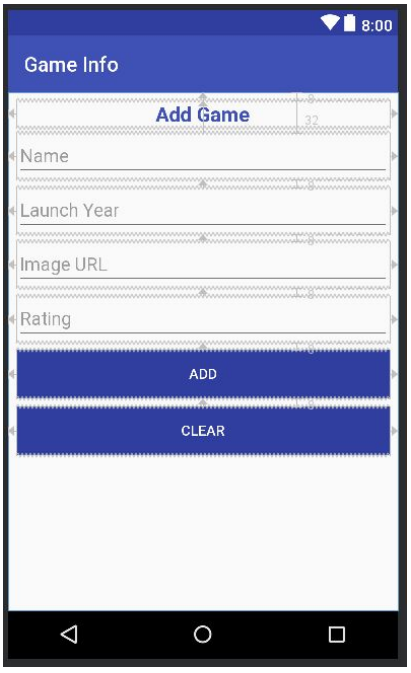
Lembre-se também de configurar o *build.gradle* do *app* com as bibliotecas do *ReactiveX*.

```
// Rx
implementation "io.reactivex.rxjava2:rxjava:2.1.15"
implementation "io.reactivex.rxjava2:rxandroid:2.0.2"
implementation "io.reactivex.rxjava2:rxkotlin:2.2.0"
implementation "com.jakewharton.rxbinding2:rxbinding:2.0.0"
```

Note que também importamos a biblioteca *RxBinding*, fornecida pelo *Jake Wharton*, que fornece algumas funcionalidades interessantes para utilização do *RxJava* detectando eventos de *views*.

Com o projeto configurado, crie um novo pacote chamado *gameadd* dentro do pacote *ui* e dentro dele crie uma nova *EmptyActivity* chamada *GameAddActivity*. Depois, crie um *layout* de acordo com a figura 6.

Figura 6: Referência do *layout* da *GameAddActivity*.



Fonte: autor - *Android Studio*

O *layout* da *Activity* deverá ficar parecido com o seguinte código:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".ui.gameadd.GameAddActivity">

  <TextView
     android:id="@+id/tvTitle"
     android:layout_width="0dp"
     android:layout_height="wrap_content"
     android:layout_marginEnd="8dp"
     android:layout_marginStart="8dp"
     android:layout_marginTop="8dp"
     android:gravity="center"
     android:text="@string/add_game"
     android:textColor="@color/colorPrimary"
     android:textSize="20sp"
     android:textStyle="bold"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintHorizontal_bias="0.0"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toTopOf="parent" />

  <EditText
     android:id="@+id/etName"
     android:layout_width="0dp"
     android:layout_height="wrap_content"
     android:layout_marginEnd="8dp"
     android:layout_marginStart="8dp"
     android:layout_marginTop="32dp"
     android:ems="10"
     android:hint="@string/name"
     android:inputType="textPersonName"
```

```
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="@+id/tvTitle" />

<EditText
    android:id="@+id/etYear"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:ems="10"
    android:hint="@string/launch_year"
    android:inputType="number"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/etName" />

<EditText
    android:id="@+id/etImageUrl"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:ems="10"
    android:hint="@string/image_url"
    android:inputType="textPersonName"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/etYear" />

<EditText
    android:id="@+id/etRating"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
```

```
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:ems="10"
        android:hint="@string/rating"
        android:inputType="numberDecimal"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/etImageUrl" />

    <Button
        android:id="@+id/btAdd"
        style="@style/Button"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:text="@string/add"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/etRating" />

    <Button
        android:id="@+id/btClear"
        style="@style/Button"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:text="@string/clear"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/btAdd" />

</android.support.constraint.ConstraintLayout>
```

Abra o arquivo *GameInfoActivity* e adicione o código para que, ao clicar no *FloatingActionButton* abra a nova *Activity*.

```
fabAdd.setOnClickListener {
  val intent = Intent(this, GameAddActivity::class.java)
  startActivity(intent)
}
```

O funcionamento da nossa tela será bem simples, o *RxJava* atuará como uma validador para o formulário, detectando eventos nas *views* de forma assíncrona e aplicando mensagens de erro conforme a validação dos campos estiver incorreta.

6.5.4.1. Flowable

Um outro elemento do *RxJava* é o *Flowable* [ref 17], que é muito semelhante ao *Observable*. Nas versões anteriores do *RxJava*, o *Observable* era a única classe desse tipo para tratar qualquer tipo de abordagem. Com as atualizações surgiu o *Flowable*, que é uma classe para tratar eventos observáveis utilizando uma estratégia de *Backpressure* personalizada.

6.5.4.2. Backpressure

Backpressure [ref 18] é o nome dado a uma sequência de eventos muito intensa que muitas vezes faz com que um *Observable* receba mais eventos do que consome. Para esses casos, utilizamos uma estratégia de *Backpressure* que eliminará eventos muito próximos, por exemplo, aliviando o sistema de consumo de eventos.

6.5.4.3. Declarando um Flowable

Começaremos construindo um *Flowable* para o *EditText* do nome. Faremos isso utilizando a seguinte declaração:

```
val nameChangeObservable = RxTextView
     .textChanges(etName)
     .skip(1)
     .toFlowable(BackpressureStrategy.LATEST)
```

A classe *RxTextView* pertence a biblioteca *RxBinding*, facilitando a detecção do evento *textChanges()* em um *EditText*. O método *skip()* fará com que a primeira

checagem do evento seja ignorada. Por fim, convertemos o *Observable* para *Flowable* informando a *BackpressureStrategy,* no caso a *LATEST*, que irá pegar o último evento emitido dentre uma sequência de eventos.

Para utilizar o *CombineLatest*, devemos ter pelo menos dois *Observables/Flowables* para unificar, portanto, iremos declarar a mesma coisa para o campo do ano.

```
val yearChangeObservable = RxTextView
     .textChanges(etYear)
     .skip(1)
     .toFlowable(BackpressureStrategy.LATEST)
```

Agora podemos utilizar as duas declarações para uní-las utilizando o *CombineLatest*, que basicamente irá ser ativado quando todos os eventos forem disparados, independente da ordem, portanto, irá esperar até que todos os campos contenham informação para que inicie a sequência ativa de eventos, validando as informações e inserindo os erros caso necessário. Para melhorar a declaração no *Kotlin*, em vez de utilizar a classe *Flowable*, como fazemos normalmente, utilizaremos a classe *Flowables*, que permite a seguinte declaração:

```
val nameChangeObservable = RxTextView
     .textChanges(etName)
     .skip(1)
     .toFlowable(BackpressureStrategy.LATEST)

val yearChangeObservable = RxTextView
     .textChanges(etYear)
     .skip(1)
     .toFlowable(BackpressureStrategy.LATEST)

Flowables.combineLatest(
     nameChangeObservable,
     yearChangeObservable
) { newName: CharSequence,
    newYear: CharSequence ->
}
```

Note que dentro da declaração do *Observer* recebemos a informação contida nas *views*, no formato *CharSequence*. Com isso, podemos declarar a checagem para os dois campos, retornando um *boolean* para caso a validação seja bem sucedida ou não, informação que será utilizada posteriormente pelo *consumer*.

```
Flowables.combineLatest(
     nameChangeObservable,
     yearChangeObservable
) { newName: CharSequence,
    newYear: CharSequence ->
    val nameValid = newName.length > 4
    if (!nameValid) {
       etName.error = "Invalid name"
    }

    val yearValid = newYear.length == 4
    if (!yearValid) {
       etYear.error = "Invalid year"
    }

    nameValid && yearValid
}
```

Com isso, podemos realizar o *subscriber* e alterar o botão de *submit* de acordo com a validação dos campos.

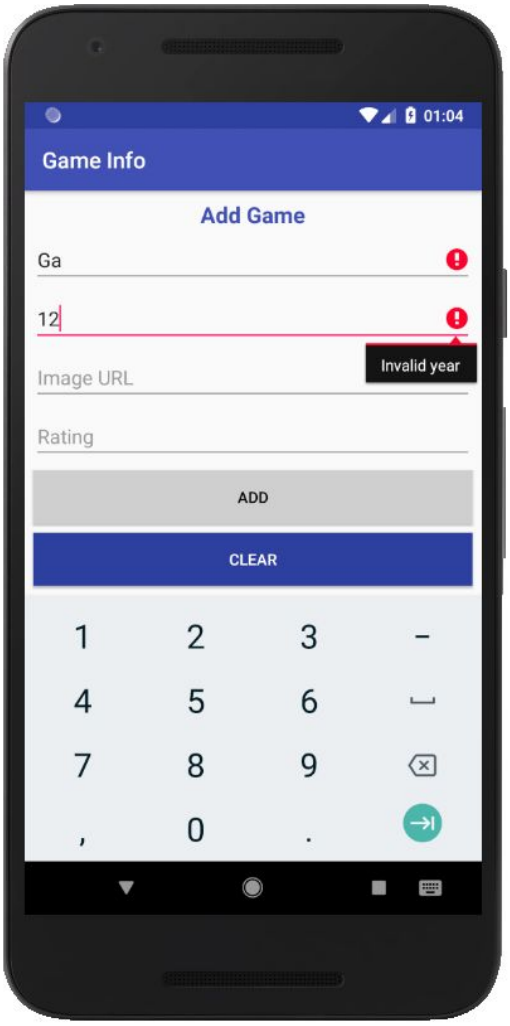
```
Flowables.combineLatest(
     nameChangeObservable,
     yearChangeObservable
) { newName: CharSequence,
   newYear: CharSequence ->
  val nameValid = newName.length >= 3
  if (!nameValid) {
     etName.error = "Invalid name"
  }

  val yearValid = newYear.length == 4
```

```
  if (!yearValid) {
     etYear.error = "Invalid year"
  }

  nameValid && yearValid
}.subscribe { formValid ->
  btAdd.setBackgroundColor(
        ContextCompat.getColor(
              applicationContext,
              if (formValid)
                 R.color.colorPrimaryDark
              else
                 R.color.gray
        )
  )

  btAdd.setTextColor(
        ContextCompat.getColor(
              applicationContext,
              if (formValid)
                 R.color.white
              else
                 R.color.black
        )
  )
}
```

Adicione as seguintes cores ao arquivo de *resources colors.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <!-- ... -->
  <color name="black">#000000</color>
  <color name="white">#FFFFFF</color>
  <color name="gray">#CFCFCF</color>
</resources>
```

Rode a aplicação e veja o resultado:



Note que após digitar a informação do nome, assim que começamos digitar a informação no campo do ano já recebemos o erro de ano inválido. Isso acontece pois o *combine latest* irá apenas detectar qualquer mudança no texto de ambos os campos, assim que todos os textos forem alterados, ele irá executar o *Observer* declarado.

Para melhorar um pouco o fluxo de validação, podemos solicitar ao *RxJava* que aguarde pelo menos alguns milissegundos antes de realizar a validação dos dados. Podemos ter esse resultado utilizando o operador *debounce()*, que fará com que eventos recebidos em um determinado tempo não sejam contabilizados até que esse intervalo de tempo passe. Quando utilizamos esse operador, precisamos utilizar o *Scheduler* para dizer ao *RxJava* que os eventos de alteração da *UI* serão feitos na *MainThread*. Altere o *Flowable* do *name* para possuir as seguintes novas características:

```
val nameChangeObservable = RxTextView
     .textChanges(etName)
     .skip(1)
     .debounce(800, TimeUnit.MILLISECONDS)
     .observeOn(AndroidSchedulers.mainThread())
     .toFlowable(BackpressureStrategy.LATEST)
```

Realize o mesmo procedimento no *Flowable* do *year*.

```
val yearChangeObservable = RxTextView
     .textChanges(etYear)
     .skip(1)
     .debounce(800, TimeUnit.MILLISECONDS)
     .observeOn(AndroidSchedulers.mainThread())
     .toFlowable(BackpressureStrategy.LATEST)
```

É possível se livrar desse *boilerplate* usando uma classe para instanciar o *Flowable*. Abra o arquivo *Delegates.kt* e insira o seguinte conteúdo:

```
fun textChangeObservable(@IdRes viewId: Int):
TextChangeObservableDelegate {
  return TextChangeObservableDelegate(viewId)
}

class TextChangeObservableDelegate(@IdRes private val viewId:
Int) : ReadOnlyProperty<Activity, Flowable<CharSequence>> {
  override fun getValue(thisRef: Activity, property:
KProperty<*>): Flowable<CharSequence> {
     return RxTextView
           .textChanges(thisRef.findViewById(viewId))
           .skip(1)
           .debounce(800, TimeUnit.MILLISECONDS)
           .observeOn(AndroidSchedulers.mainThread())
           .toFlowable(BackpressureStrategy.LATEST)
  }
}
```

E altere a declaração das variáveis na *Activity*.

```
private val nameChangeObservable: Flowable<CharSequence> by
textChangeObservable(R.id.etName)
private val yearChangeObservable: Flowable<CharSequence> by
textChangeObservable(R.id.etYear)
```

Por fim, há um outro problema que não conseguimos percebemos no momento, o *Flowable* irá continuar disparando o evento para sempre, mesmo que a *Activity* seja destruída. Para contornar isso, devemos armazenar o *disposable* em uma propriedade da classe para desativá-lo quando a *Activity* for destruída. Com isso, sobrescrevemos o método *onDestroy()* para realizar a chamada *disposable.dispose()*, que cuidará de desativar o *Flowable*.

```
private lateinit var disposable: Disposable

override fun onDestroy() {
  super.onDestroy()
  disposable.dispose()
}

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

  disposable = Flowables.combineLatest(
// ...
```

Com isso temos a base para o funcionamento do sistema. Adicione a validação para os outros campos da mesma maneira e teste a aplicação.

Grande parte do fluxo presente nesse capítulo foi inspirado no vídeo 4 (ir para vídeo) e no vídeo 5 (ir para vídeo), excelentes em termos de conteúdo e explicação sobre o tema.

Como podemos perceber, o tópico RxJava é muito interessante e muito poderoso, com diversas aplicações e funcionalidades extremamente úteis. Se tiver

interessado em saber mais sobre o assunto, recomendo que assista o vídeo 6 (ir para vídeo), que contém explicações e exemplos muito bons sobre o tema.

7. Dependency Injection com Dagger 2

Assim como a programação reativa, tratada anteriormente, a *Dependency Injection*, ou injeção de dependência, é um outro tópico bem complexo inicialmente e com uma curva de aprendizado lenta. A injeção de dependência é um assunto presente desde o começo dos anos 1990, com abordagens em diversos livros e para diversas linguagens de programação, possuindo uma série de bibliotecas capazes de facilitar a utilização desse conceito.

Esse conceito visa reduzir o nível de acoplamento entre diferentes módulos de um sistema, ou seja, em vez da dependência de um módulo com outro ser escrita diretamente no código de um deles, isso é feito pela configuração do *software*, que irá "injetar" em cada componentes as dependências que foram declaradas. Essa prática está relacionada com o padrão *Inversion of Control* (inversão de controle), que discutiremos mais a frente.

Apesar de ser um assunto antigo no universo *Android*, vem recebendo cada vez mais atenção e está em constante evolução visando melhores práticas tanto de implementação quanto nas abordagens "por baixo dos panos" dos *frameworks*. Com isso em mente, antes de começar a implementação desse padrão no *Android*, precisamos entender o que ele é exatamente e qual a sua real importância para o desenvolvimento de uma aplicação de alto nível.

7.1. O que é Dependency Injection

Dependency Injection é basicamente uma técnica de programação, que pode ser realizada em qualquer linguagem de programação e de várias maneiras. Em qualquer projeto, seja ele grande ou pequeno, mesmo que nós não tenhamos percebidos, existem casos onde objetos dependem de objetos, ou seja, em virtude de construir uma nova informação precisamos que uma outra informação também seja construída.

Visualizar esse conceito sendo aplicado na prática facilita muito o entendimento. Utilizarei como exemplo uma aplicação que realiza posts no *Twitter*. O diagrama inicial da aplicação pode ser observado na figura 7.

Esse exemplo é inspirado em uma apresentação feita por *Jake Wharton*, na conferência *Devoxx*, que aconteceu na Bélgica. Recomendo que assista a apresentação, disponível através do vídeo 7 (ir para vídeo). Além disso, aproveitando a dica, recomendo também como leitura adicional, essa página de

apresentações, falando sobre diversos temas muito interessantes do universo *Java* e *Android*.

Figura 7: Diagrama inicial da aplicação

Fonte: autor

7.2. Exemplo prático

Para iniciar nosso exemplo prático, crie um novo projeto com *Kotlin*, *API* 15 com uma *EmptyActivity* chamada *MainActivity*. Abra o arquivo *build.gradle* do *app* e adicione a biblioteca *OkHttpClient*, que servirá apenas para ilustrar.

```
// OkHttpClient
implementation "com.squareup.okhttp3:okhttp:3.11.0"
```

Iniciando o desenvolvimento da nossa aplicação, precisamos construir nossa classe capaz de receber uma mensagem e executar a *API* do *Twitter*, informando qual o usuário. Chamaremos essa classe de *Tweeter*.

```
class Tweeter {
  fun tweet(tweet: String) {
     val api = TwitterApi()
     api.postTweet("Paulo Salvatore", tweet)
  }
}
```

Para realizar um *tweet* precisamos da classe *TwitterApi*, que está sendo construída dentro da classe *Tweeter*. Perceba que nesse caso temos a nossa primeira dependência, qualquer alteração no construtor da classe *TwitterApi* deverá ser atualizada nesta chamada também. A princípio pode não parecer um problema tão grande, mas imagine que a classe *TwitterApi* é utilizada em 10 outros trechos, a partir disso começamos a ter um problema maior.

Para que esse exemplo funcione precisamos da declaração da classe *TwitterApi*, que deve possuir o membro *postTweet()* e saber como fazer o *post* de um *tweet* recebendo o usuário e a mensagem.

```
class TwitterApi {
  fun postTweet(user: String, tweet: String) {
     val client = OkHttpClient()
     val request = Request.Builder()/*...*/.build()
     client.newCall(request).execute()
  }
}
```

Pronto, agora que temos tudo declarado, podemos realizar a execução dos membros e realizar a publicação de um novo *tweet*. Fazemos isso da seguinte maneira:

```
val tweeter = Tweeter()
tweeter.tweet("Hello, #DependencyInjection")
```

Olhando de volta para o nosso código da classe *TwitterApi*, perceba que toda vez que chamamos o membro *postTweet()*, temos uma nova instância do *OkHttpClient* sendo feita. Esse é nosso primeiro problema, pois não precisamos disso, apenas uma instância dele resolveria todo o nosso trabalho. Imagina ter que declarar informações adicionais em toda construção dessa classe? Inviável. Para resolver isso, a princípio, podemos simplesmente fazer com que a instância seja feita no escopo da classe, fora do membro *postTweet()*, ficando assim:

```
class TwitterApi {
  private val client = OkHttpClient()

  fun postTweet(user: String, tweet: String) {
     // ...
  }
}
```

No entanto, não queremos que a construção do *OkHttpClient* seja feita dessa forma, queremos que uma injeção de dependência seja feita. Para isso,

podemos solicitar que toda vez que a *TwitterApi* for construída, ela receba a dependência do *OkHttpClient*.

```
class TwitterApi(private val client: OkHttpClient) {
  fun postTweet(user: String, tweet: String) {
     val request = Request.Builder()/*...*/.build()
     client.newCall(request).execute()
  }
}
```

Pronto! Com isso nós realizamos nossa primeira injeção de dependência, podendo ver que realmente é basicamente argumentos no construtor, eliminando a responsabilidade da classe realizar a construção de uma de suas dependências.

Voltando para a nossa classe *Tweeter*, agora devemos passar uma instância do *OkHttpClient* durante a construção da *TweeterApi*.

```
class Tweeter {
  fun tweet(tweet: String) {
     val api = TwitterApi(OkHttpClient())
     api.postTweet("Paulo Salvatore", tweet)
  }
}
```

Agora a classe *Tweeter* está no controle de criar a instância do *OkHttpClient* que a *TwitterApi* irá receber para prosseguir com a execução do programa. Entretanto, agora estamos com o mesmo problema, toda vez que realizamos um *tweet*, criamos um novo objeto. Faremos o mesmo para resolver isso, extraímos a declaração. Aproveitando, também não queremos que o nome do usuário esteja declarado dentro do membro, afinal, pretendemos receber diversos usuários que também irão realizar *tweets*. Apesar do usuário ser apenas uma *string*, essa também é uma forma de injeção de dependência, pois remove a necessidade da classe *Tweeter* de saber quem está realizando a ação.

```
class Tweeter(private val user: String) {
  private val api = TwitterApi(OkHttpClient())

  fun tweet(tweet: String) {
```

```
    api.postTweet("Paulo Salvatore", tweet)
  }
}
```

Então, na declaração de execução do programa, devemos atualizar informando o nome do usuário e podemos prosseguir com nossos *tweets*.

```
val tweeter = Tweeter("Paulo Salvatore")
tweeter.tweet("Hello, #DependencyInjection")
tweeter.tweet("Kotlin > Java #KotlinWorld")
tweeter.tweet("#DependencyInjection isn't so hard")
tweeter.tweet("What about #Dagger?")
```

Por enquanto tudo está ótimo, até que decidimos adicionar mais funcionalidades ao nosso programa, como no caso em que queremos adicionar nossa *timeline* para saber o que as outras pessoas estão fazendo.

```
val timeline = Timeline("Paulo Salvatore")
timeline.loadMore(20)
for (tweet in timeline.get()) {
  println(tweet)
}
```

Para que isso funcione, precisamos declarar a classe *Timeline* e implementar os membros solicitados *loadMore()* e *get()*.

```
class Timeline(private val user: String) {
  private val cache: List<Tweet> = emptyList()
  private val api = TwitterApi(OkHttpClient())

  fun get() = cache
  fun loadMore(amount: Int) { /* ... */ }
}
```

Agora temos um novo problema, afinal, estamos utilizando o mesmo mecanismo que necessita da instância da *TwitterApi*. Faremos então o mesmo

processo que realizamos antes, transformamos a dependência da *API* em um argumento vindo do construtor.

```
class Timeline(private val api: TwitterApi,
               private val user: String) {
  private val cache: List<Tweet> = emptyList()

  fun get() = cache
  fun loadMore(amount: Int) { /* ... */ }
}
```

Em seguida, devemos fazer a mesma coisa com o *Tweeter*.

```
class Tweeter(private val api: TwitterApi,
              private val user: String) {
  fun tweet(tweet: String) {
     api.postTweet(user, tweet)
  }
}
```

Com isso, tiramos a responsabilidade de ambas as classes de instanciar ou configurar a classe *TwitterApi*, reutilizando a instância e economizando recursos.

Entretanto, enquanto parece muito bom para a nossa biblioteca, o código que está executando a ação ficará poluído com todo o código necessário para chamar as classes. Se quero apenas criar e ler alguns *tweets*, eu não quero ter que declarar todas as dependências manualmente.

```
val client = OkHttpClient()
val api = TwitterApi(client)
val user = "Paulo Salvatore"

val tweeter = Tweeter(api, user)
tweeter.tweet("Hello, #DependencyInjection")
tweeter.tweet("Kotlin > Java #KotlinWorld")
tweeter.tweet("#DependencyInjection isn't so hard")
tweeter.tweet("What about #Dagger?")
```

```
val timeline = Timeline(api, user)
timeline.loadMore(20)
for (tweet in timeline.get()) {
  println(tweet)
}
```

Perceba a partir desse ponto que nosso *software* está ganhando volume. Temos duas classes principais que realizam funções na *API* do *Twitter* utilizando a conexão estabelecida via *OkHttpClient*. O diagrama atualizado pode ser visualizado na figura 8.

Figura 8: Diagrama atualizado da aplicação

Fonte: autor

A partir desse momento surge a primeira dúvida: como evitar todo esse *boilerplate* manual? É aí que entram as bibliotecas de *dependency injection*.

7.3. Bibliotecas: Evolução

Existem diversas bibliotecas disponíveis para tratar a injeção de dependência, independente da linguagem de programação ou do *framework* que estamos trabalhando. No *Java*, temos algumas principais bibliotecas que evoluíram ao longo do tempo melhorando cada vez mais a utilização e principalmente a performance. As principais bibliotecas que iremos discutir um pouco são: *Guice*; *Dagger v1* e *Dagger v2*.

7.3.1. Guice

Características

- Desenvolvida pelo *Google*;

- *Bob Lee* como desenvolvedor inicial;
- Posteriormente também desenvolvida por *Jesse Wilson* e diversos outros contribuidores;
- Mantida pelo time *Java Core Libraries*, do *Google*;
- Poderosa, dinâmica, muito bem testada e muito bem difundida.

Problemas

- Problemas de configuração apareciam durante o tempo de execução;
- Com novas abordagens, notou-se que possuía uma inicialização lenta, uma injeção também lenta e alguns problemas de memória.

7.3.2. Dagger v1

Características

- Desenvolvida pela *Square*;
- *Jesse WIlson* como desenvolvedor principal, aconselhado por *Bob Lee*;
- Inicialmente direcionada a ambientes com recursos bem restritos, como é o caso do *Android*;
- Possui uma análise estática de todas as dependências e *injection points*, permitindo que os erros de configuração apareçam durante o tempo de compilação, em vez de durante o tempo de execução, como acontecia com o *Guice*;
- Elimina a *reflection* em métodos, *fields* e *annotations*;
    - *Reflection* basicamente serve para determinar métodos e atributos que serão utilizados de determinada classe (que geralmente nem conhecemos) em tempo de execução. Essa abordagem geralmente causa redução de desempenho e possui alguns outros problemas [ref 14].
- Muito bem difundido, obtendo sucesso em grandes aplicações do *Android*.

Problemas

- No entanto, estava causando alguns excessos de carregamento em classes, dependendo da maneira que era utilizada;
- Nem todas as checagens de declarações eram feitas de maneira estática, portanto, nem todos os erros de configuração eram percebidos durante o tempo de compilação;
- Por último, ainda havia um pouco de uso de *reflections*, mesmo que com um sistema forte de *cache*, ainda era algo que consumia recursos importantes.

7.3.3. Dagger v2

Características

- *Fork* do *Dagger v1*;
- Proposta inicial e implementação feita pela equipe *Java Core Libraries*, do *Google*;
- Elimina problemas que eram causados com a biblioteca de *runtime* do *Dagger v1* e também com os padrões utilizados para geração automática de códigos;
- Todos as checagens de configuração eram feitas durante a análise estática, portanto, todos os erros de configuração aconteciam durante o tempo de compilação, sem exceções;
- Implementação da biblioteca no Dagger v1 era feita abaixo do esperado, portanto, um dos objetivos também era melhorar a maneira que as declarações eram feitas em conjunto com a análise estática que permite as checagens de configuração no tempo de compilação.

7.4. Dagger API

Conhecendo a evolução das principais bibliotecas de injeção de dependência fica mais fácil de entender esse cenário e quais os objetivos dos desenvolvedores durante a construção de cada uma delas. A biblioteca *Dagger v2*, do *Google*, é a mais consistente e tem se mostrado cada vez mais poderosa com o uso em aplicações cada vez maiores.

Para entender como ela funciona, precisamos entender seus principais pilares, responsáveis por fornecer as dependências conforme são solicitadas:

- **@Module + @Provides**
  - Mecanismo para fornecer dependências
- **@Inject**
  - Mecanismo para solicitar dependências
- **@Component**
  - Ponte entre *modules* e *injections*

7.4.1. Configurando o projeto

Para adicionar o *Dagger 2* no projeto, abra o arquivo *build.gradle* do *app* e adicione as seguintes dependências:

```
// ...
```

```
apply plugin: "kotlin-kapt"
// ...
dependencies {
  // ...
  // Dagger 2
  kapt "com.google.dagger:dagger-compiler:2.16"
  implementation "com.google.dagger:dagger:2.16"
}
```

7.4.2. Fornecendo dependências

O primeiro assunto que devemos saber é como fornecer as dependências necessárias de uma aplicação. Para isso utilizamos os módulos, que são basicamente classes anotadas com **@Module** e que possuem métodos que fornecem as dependências. Cada um desses métodos deve ser anotado com a classe **@Provides**, permitindo que o *Dagger* encontre eles e saiba o que ele fornece exatamente.

Continuando nosso exemplo de aplicação do *Twitter*, o nosso primeiro módulo seria o responsável pela conexão, que chamaremos de *NetworkModule*. Sua declaração inicial seria essa:

```
class NetworkModule {
  fun provideOkHttpClient(): OkHttpClient = OkHttpClient()

  fun provideTwitterApi(client: OkHttpClient): TwitterApi =
TwitterApi(client)
}
```

Seguindo as necessidades do *Dagger*, a classe deverá ser anotada com **@Module** e cada método que fornece algum tipo de dependência deverá ser anotado com **@Provides**.

```
@Module
class NetworkModule {
  @Provides
  fun provideOkHttpClient(): OkHttpClient = OkHttpClient()
```

```
  @Provides
  fun provideTwitterApi(client: OkHttpClient): TwitterApi =
TwitterApi(client)
}
```

Além disso, para evitar que diversas instâncias desses objetos sejam criadas, podemos anotá-lo utilizando a anotação **@Singleton**, fornecida pelo pacote '*javax.inject*', que fará com que o *Dagger* saiba que essa instância de objeto deve ser reutilizada.

```
@Module
class NetworkModule {
  @Provides @Singleton
  fun provideOkHttpClient(): OkHttpClient = OkHttpClient()

  @Provides @Singleton
  fun provideTwitterApi(client: OkHttpClient): TwitterApi =
TwitterApi(client)
}
```

O *Dagger* basicamente irá olhar para o módulo e os métodos anotados e entenderá exatamente o que você quer fornecer com eles pois, além das anotações, a outra coisa que ele precisa saber é o tipo de retorno dos métodos. Com isso, é possível determinar exatamente qual o tipo do objeto de dependência que ele poderá fornecer. Além disso, ele também olhará para os parâmetros de cada um dos métodos. No nosso caso, o segundo método, *provideTwitterApi()*, necessita receber uma instância do *OkHttpClient*, algo que é percebido pelo *Dagger*.

O próximo módulo da nossa aplicação é o *TwitterModule*, que diferente do anterior, precisa de um usuário para executar suas ações. Esse usuário, por sua vez, deverá ser recebido no construtor da classe. Além disso, como se trata de um módulo, precisamos anotá-lo devidamente.

```
@Module
class TwitterModule(private val user: String)
```

Esse módulo deverá fornecer duas dependências, uma para o *Tweeter* e outra para o *Timeline*.

```
@Module
class TwitterModule(private val user: String) {
  @Provides @Singleton
  fun provideTweeter(api: TwitterApi) = Tweeter(api, user)

  @Provides @Singleton
  fun provideTimeline(api: TwitterApi) = Timeline(api, user)
}
```

Com essas declarações feitas, temos um resultado interessante. Sempre que precisarmos de uma instância da classe *Tweeter*, o *Dagger* irá buscar por um método que pode prover isso, encontrando o *provideTweeter()*. Ele perceberá que esse método possui um parâmetro como dependência e para instanciá-lo é necessário uma instância da classe *TwitterApi*, que pode ser fornecida pelo método *provideTwitterApi()*, do módulo *NetworkModule*. A classe *TwitterApi*, por sua vez, precisa do *OkHttpClient*, como dito anteriormente. O *Dagger* irá fazer todo esse caminho até encontrar e instanciar todos os objetos necessários e retornará um a um até o método que iniciou toda a solicitação, que no nosso caso foi o *provideTweeter()*.

Depois disso, temos a solicitação de uma instância da classe *Timeline*, quem também tem as mesma dependências da classe *Tweeter*. O *Dagger* irá percorrer o mesmo caminho para satisfazer as dependências, porém, quando chegar no método *provideTwitterApi()*, perceberá que já existe uma instância dele criada, afinal, esse método está anotado com a anotação **@Singleton**. Isso fará com o *Dagger* retorne essa instância sem necessariamente percorrer todo o caminho novamente.

7.4.3. Solicitando dependências

Como mencionado anteriormente, para solicitar dependências devemos usar a anotação **@Inject**, do pacote '*javax.inject*'. Essas anotações podem ser feitas em três lugares: No construtor, em algum *field* ou no método que realiza a *injection*.

O primeiro caso é a injeção de dependência declarada no construtor, sendo possível declarar apenas em um construtor, sendo automaticamente válida para todos os parâmetros presentes, sendo possível armazenar qualquer uma das

dependências em variáveis *private* e *val*, representando objetos imutáveis (que não podem mudar de valor).

Prosseguindo na nossa aplicação, que representaremos pela classe *TwitterApplication*, para construí-la temos as dependências das classes *Tweeter* e *Timeline*.

```
class TwitterApplication(private val tweeter: Tweeter,
                         private val timeline: Timeline)
```

Para que essas duas dependências, basta adicionar a anotação **@Inject** antes da declaração do construtor. Como no *Kotlin*, o construtor geralmente é representado apenas pelo parênteses, teremos que utilizar a outra forma de declaração um pouco mais completa.

```
class TwitterApplication @Inject constructor(
                                private val tweeter: Tweeter,
                                private val timeline: Timeline)
```

Outra funcionalidade muito interessante dessa abordagem é que ela permite ao *Dagger* utilizar essa declaração sempre que essa dependência for solicitada, fazendo com que ela fique disponível em outros casos.

Um exemplo mais concreto disso seria com nossa classe *TwitterApi*, que para que sua injeção de dependência seja feita, declaramos o método *provideTwitterApi()*. Essa funcionalidade nos permite, por exemplo, eliminar toda a declaração do método e utilizarmos o *inject* declarado no construtor da classe.

Removemos o método *provideTwitterApi()*.

```
@Module
class NetworkModule {
  @Provides
  @Singleton
  fun provideOkHttpClient(): OkHttpClient = OkHttpClient()

  @Provides
  @Singleton
  fun provideTwitterApi(client: OkHttpClient): TwitterApi =
TwitterApi(client)
```

```
}
```

Adicionamos a anotação **@Inject** no construtor da classe *TwitterApi*.

```
class TwitterApi @Inject constructor(
                          private val client: OkHttpClient) {
  fun postTweet(user: String, tweet: String) {
     val request = Request.Builder()/*...*/.build()
     client.newCall(request).execute()
  }
}
```

Como é possível notar, o método que removemos estava anotado como **@Singleton**, e removendo esse método fará com que o *Dagger* crie uma instância da classe sempre que for solicitado. Para evitar que isso aconteça, simplesmente movemos a anotação para a classe em questão, que no nosso caso é a *TwitterApi*.

```
@Singleton
class TwitterApi @Inject constructor(private val client:
OkHttpClient) {
  fun postTweet(user: String, tweet: String) {
     val request = Request.Builder()/*...*/.build()
     client.newCall(request).execute()
  }
}
```

O segundo caso é a injeção de dependência através de métodos, que é bem semelhante a injeção em construtores, onde os parâmetros do método são as dependências e a injeção acontece apenas depois que o objeto está completamente instanciado. Esse tipo de abordagem só possui um caso real em que é útil e é quando é necessário passar a instância de si mesmo para uma dependência.

Supondo que temos uma classe *Streaming*, que possui um método chamado *register()* que precisa da instância da nossa aplicação. Além disso, nossa aplicação é dependente da instância dessa classe e queremos realizar a injeção de dependência utilizando o *Dagger*. Nesse caso, a injeção de dependência em métodos é extremamente útil.

```
class Streaming {
  fun register(app: TwitterApplication) { /* ... */ }
}
```

```
class TwitterApplication @Inject constructor(
                            private val tweeter: Tweeter,
                            private val timeline: Timeline) {
  @Inject
  fun enableStreaming(streaming: Streaming) {
     streaming.register(this)
  }
}
```

O terceiro e último caso de injeção de dependência é através de *fields*, que basicamente consiste da anotação de injeção inserida nos *fields*, sendo que eles não podem ser *private* e devem ser mutáveis (no *Kotlin*, *var*).

No nosso exemplo, podemos pegar a nossa classe *TwitterApplication* e remover a necessidade de ser construída recebendo os dois parâmetros de dependência. Passamos então a declarar essas dependências como *fields* da classe.

```
class TwitterApplication {
  @Inject lateinit var tweeter: Tweeter
  @Inject lateinit var timeline: Timeline
  // ...
}
```

Note que no *Kotlin*, além da variável precisar se *var*, também devemos adicionar a declaração *lateinit*, para que o *Kotlin* saiba que ela será inicializada depois.

Esse tipo de declaração é muito útil em casos como do *Android*, onde o objeto da *Activity* é criado pelo sistema e não podemos alterar a declaração do construtor.

Assim como a injeção de dependência através de métodos, a injeção através de *fields* só acontece depois que o objeto está completamente instanciado, ocorrendo depois da injeção no construtor, porém, antes da injeção em métodos.

Outra informação importante é nesse caso, o objeto geralmente é responsável ou deve estar consciente da injeção em *fields*, simplesmente pelo fato de que após o construtor ter sido realizado ainda podem haver algumas dependências não resolvidas.

7.4.4. Conectando as duas partes

Finalmente, estamos fornecendo dependências com @Module e solicitando dependências com @Inject, agora precisamos entender como conectar essas duas partes. O mecanismo para realizar isso é chamado de *Component*, que basicamente atua como *injector*.

Um componente é basicamente uma interface com a anotação **@Component**, que possui uma lista os módulos referenciados.

```
@Component(
     modules = [NetworkModule::class,
                TwitterModule::class]
)
interface TwitterComponent
```

Os componentes estão cientes de todas as dependências, então nesse caso como todas as dependências referenciadas são *singleton*, também marcaremos o componentes como *singleton*.

```
@Singleton
@Component(
     modules = [NetworkModule::class,
                TwitterModule::class]
)
interface TwitterComponent
```

A maneira com que expomos os dados em um componente é simplesmente escrevendo métodos abstratos para esses tipos.

```
@Singleton
@Component(
     modules = [NetworkModule::class,
```

```
                TwitterModule::class]
)
interface TwitterComponent {
  fun tweeter(): Tweeter
  fun timeline(): Timeline
}
```

Com isso, o *Dagger* irá gerar a implementação dessa interface, permitindo com que a gente utilize esses métodos em nossa aplicação. Como todos os códigos do *Dagger* são gerados durante o tempo de compilação, devemos executar a opção '*Make Project*' para que a declaração a seguir funcione.

```
val component = DaggerTwitterComponent.builder()
     .networkModule(NetworkModule())
     .twitterModule(TwitterModule("Paulo Salvatore"))
     .build()
```

Simplesmente pelo fato da variável *component* agora possuir a implementação da interface, podemos acessar diretamente seus métodos para acessar as instâncias de nossas dependências completamente construídas e injetadas.

```
val tweeter = component.tweeter()
val timeline = component.timeline()
```

Além disso, podemos notar que nosso *NetworkModule* possui uma construção padrão, que não necessita de argumentos. Nesse caso, podemos deixar com que o *Dagger* realize essa construção, passando apenas os módulos que necessitam de alguma referência externa.

```
val component = DaggerTwitterComponent.builder()
     .networkModule(NetworkModule())
     .twitterModule(TwitterModule("Paulo Salvatore"))
     .build()
```

Agora que temos nossas dependências, podemos realizar nossas atividades normalmente.

```
val component = DaggerTwitterComponent.builder()
     .networkModule(NetworkModule())
     .twitterModule(TwitterModule("Paulo Salvatore"))
     .build()

val tweeter = component.tweeter()
tweeter.tweet("Hello, #Dagger2")
tweeter.tweet("Kotlin > Java #KotlinWorld")
tweeter.tweet("#Dagger2 s2")
tweeter.tweet("#Dagger2 is the best")

val timeline = component.timeline()
timeline.loadMore(20)
for (tweet in timeline.get()) {
  println(tweet)
}
```

Voltando em nossa classe *TwitterApplication* e à injeção de dependência pelo construtor, podemos adicionar uma outra funcionalidade como a implementação da interface *Runnable*, que nos permitirá implementar o método *run()*.

```
class TwitterApplication @Inject constructor(
     private val tweeter: Tweeter,
     private val timeline: Timeline
) : Runnable {
  override fun run() {
     tweeter.tweet("Hello, #Dagger2")

     timeline.loadMore(20)
     for (tweet in timeline.get()) {
        println(tweet)
     }
  }
```

```
  // ...
}
```

Além disso, também podemos expôr essa classe através de um componente, o que geralmente é o padrão que encontramos em aplicações que utilizam o *Dagger*.

```
@Singleton
@Component(
     modules = [NetworkModule::class,
        TwitterModule::class]
)
interface TwitterComponent {
  fun app(): TwitterApplication
}
```

Antes de compilar para conseguir testar, precisamos de uma maneira de injetar a dependência de *Streaming* solicitada pela classe *TwitterApplication*.

```
class Streaming @Inject constructor() {
  fun register(app: TwitterApplication) { /* ... */ }
}
```

Agora podemos compilar utilizando o '*Make Project*' e realizar a execução da aplicação da seguinte forma:

```
component.app().run()
```

Ainda na nossa classe *TwitterApplication*, o que aconteceria se em vez da injeção ser feita no construtor ela fosse feita direto nos campos?

```
class TwitterApplication : Runnable {
  @Inject lateinit var tweeter: Tweeter
  @Inject lateinit var timeline: Timeline
```

```
  override fun run() {
     tweeter.tweet("Hello, #Dagger2")

     timeline.loadMore(20)
     for (tweet in timeline.get()) {
        println(tweet)
     }
  }
  // ...
}
```

Nesse caso, devemos declarar um método abstrato no componente que irá aceitar uma instância desse tipo, conseguindo realizar a injeção de dependência nos campos ou nos métodos desse objeto recebido.

```
@Singleton
@Component(
     modules = [NetworkModule::class,
        TwitterModule::class]
)
interface TwitterComponent {
  fun injectApp(app: TwitterApplication)
}
```

Dessa forma, construímos o componente assim como fizemos antes, porém agora, instanciamos nós mesmos a classe *TwitterApplication*, passamos para o *Dagger* definir os campos desse objeto e por fim podemos executar nossa aplicação.

```
val component = DaggerTwitterComponent.builder()
     .twitterModule(TwitterModule("Paulo Salvatore"))
     .build()

val app = TwitterApplication()
component.injectApp(app)
app.run()
```

Além disso, outra abordagem interessante é que podemos declarar o método no componente para retornar a mesma instância recebida e permitir a você utilizar a declaração do método do componente como geralmente é feito nos padrões de *builder* utilizados em várias declarações.

```
@Singleton
@Component(
     modules = [NetworkModule::class,
                TwitterModule::class]
)
interface TwitterComponent {
  fun injectApp(app: TwitterApplication): TwitterApplication
}
```

```
val app = TwitterApplication()
component.injectApp(app).run()
```

Portanto, podemos realizar o mesmo com objetos criados pelo sistema, como as *Activities*:

```
@Singleton
@Component(
     modules = [NetworkModule::class,
                TwitterModule::class]
)
interface TwitterComponent {
  fun injectActivity(activity: TwitterActivity)
}
```

```
component.injectActivity(this@TwitterActivity)
```

Por último, um outro assunto muito interessante presente nos componentes e que difere o *dagger 2* do *dagger 1*, por exemplo, é a implementação de escopos, que permite um gerenciamento muito mais apurado das instâncias de cada componente e de cada módulo presente na nossa aplicação.

Para demonstrar essa utilização, iremos separar nosso componente principal. Em nossa aplicação temos dois módulos, sendo que cada um possui responsabilidades diferentes. Portanto, criaremos um novo componente chamado *ApiComponent*, que utiliza o módulo *NetworkModule*.

```
@Singleton
@Component(modules = [NetworkModule::class])
interface ApiComponent
```

Com isso, mantemos o *TwitterComponent*, mantendo apenas a declaração do *TwitterModule* e removendo a anotação de *singleton*, por enquanto.

```
@Component(modules = [TwitterModule::class])
interface TwitterComponent {
  fun app(): TwitterApplication
}
```

Depois, adicionamos uma dependência ao *TwitterComponent*, sinalizando que ele é dependente de *ApiComponent*. Isso significa que é impossível criar uma instância do *TwitterComponent* sem que haja uma instância criada do *ApiComponent*.

```
@Component(
     dependencies = [ApiComponent::class],
     modules = [TwitterModule::class]
)
interface TwitterComponent {
  fun app(): TwitterApplication
}
```

Nesse momento, um fato interessante é que caso iniciamos a compilação nesse estado, ela irá falhar. Isso acontecerá pois anteriormente nós declaramos dentro do *TwitterModule* que as classes *Tweeter* e *Timeline* necessitam de uma instância da classe *TwitterApi*. Os componentes não irão expor nenhuma propriedade de seus módulos a não ser que você diga isso explicitamente. Isso significa que teremos que declarar um método dizendo que a instância da

TwitterApi será exposta através do *ApiComponent*, permitindo com que o *TwitterComponent* consiga utilizá-la para criar a *TwitterApplication*.

```
@Singleton
@Component(modules = [NetworkModule::class])
interface ApiComponent {
  fun api(): TwitterApi
}
```

Voltando para o nosso *TwitterComponent*, como feito anteriormente, nós removemos a anotação de *singleton*. Vamos adicioná-la novamente para entender um pouco mais sobre os escopos dos componentes.

```
@Singleton
@Component(
     dependencies = [ApiComponent::class],
     modules = [TwitterModule::class]
)
interface TwitterComponent {
  fun app(): TwitterApplication
}
```

Com a anotação adicionada, caso executarmos o '*Make Project*' iremos receber um erro, pois um componente com escopo *singleton* não pode dependender de outros componentes com esse mesmo escopo.

```
error: This @Singleton component cannot depend on scoped
components
```

Para resolver isso por enquanto, devemos remover todas as declarações de *singleton* presentes tanto no componente quanto no seus módulos, tornando esse componente o que chamamos de *unscoped* (sem escopo).

```
~~@Singleton~~
@Component(
     dependencies = [ApiComponent::class],
```

```
    modules = [TwitterModule::class]
)
interface TwitterComponent {
  fun app(): TwitterApplication
}
```

```
@Module
class TwitterModule(private val user: String) {
  @Provides
  @Singleton
  fun provideTweeter(api: TwitterApi) = Tweeter(api, user)

  @Provides
  @Singleton
  fun provideTimeline(api: TwitterApi) = Timeline(api, user)
}
```

Do jeito que está declarado por enquanto, significa que cada vez que chamarmos o método *app()* irá criar sempre criar uma nova instância da classe *TwitterApplication()*. Se tentarmos compilar o projeto, o *Dagger* irá acusar um novo erro, dessa vez alegando que o componentes sem escopo (*unscoped*) não podem depender de componentes com escopo.

```
error: package.TwitterComponent (unscoped) cannot depend on
scoped components
```

Para resolver isso, precisamos entender como criar escopos customizados, que falarei um pouco mais a frente.

Agora que nossos componentes estão separados, precisamos entender como construí-los. Nosso *ApiComponent* precisa do *NetworkModule*, que por sua vez não precisa de argumentos para ser criado, portanto, podemos deixar esse trabalho para o *Dagger* realizar.

```
val apiComponent = DaggerApiComponent.builder().build()
```

Sempre que temos um componente onde todos os módulos podem ser criados de maneiras implícita, o *Dagger* nos fornece uma maneira mais fácil de declarar sua inicialização, utilizando diretamente o método *create()*.

```
val apiComponent = DaggerApiComponent.create()
```

Para o *TwitterComponent*, em vez de esperar o *NetworkModule*, está esperando agora o *ApiComponent* instanciado, juntamente com o *TwitterModule*, que já estávamos declarando anteriormente.

```
val twitterComponent = DaggerTwitterComponent.builder()
     .apiComponent(apiComponent)
     .twitterModule(TwitterModule("Paulo Salvatore"))
     .build()
```

Com isso, podemos iniciar a execução do componente normalmente como estávamos fazendo anteriormente.

```
twitterComponent.app().run()
```

Para conseguir rodar a aplicação, precisamos corrigir o fato de um componente sem escopo depender de um componente com escopo. Para fazer isso, devemos criar um escopo personalizado, declarando uma *annotation* em qualquer lugar do código.

```
annotation class User
```

Para que essa anotação seja válida como escopo, devemos anotá-la usando **@Scope**, do pacote '*javax.inject*'.

```
@Scope
annotation class User
```

Com isso, devemos adicionar a *annotation* criada no componente e nos métodos anotados com @Provides presentes nos módulos relacionados.

```
@User
@Component(
     dependencies = [ApiComponent::class],
     modules = [TwitterModule::class]
)
interface TwitterComponent {
  fun app(): TwitterApplication
}
```

```
@Module
class TwitterModule(private val user: String) {
  @Provides
  @User
  fun provideTweeter(api: TwitterApi) = Tweeter(api, user)

  @Provides
  @User
  fun provideTimeline(api: TwitterApi) = Timeline(api, user)
}
```

Anteriormente definimos uma injeção direto no construtor da classe *Streaming*. Além disso, também é possível criar um método dentro do *TwitterModule* que ficará responsável por fornecer essa injeção de dependência, removendo a declaração do construtor e construindo o método específico.

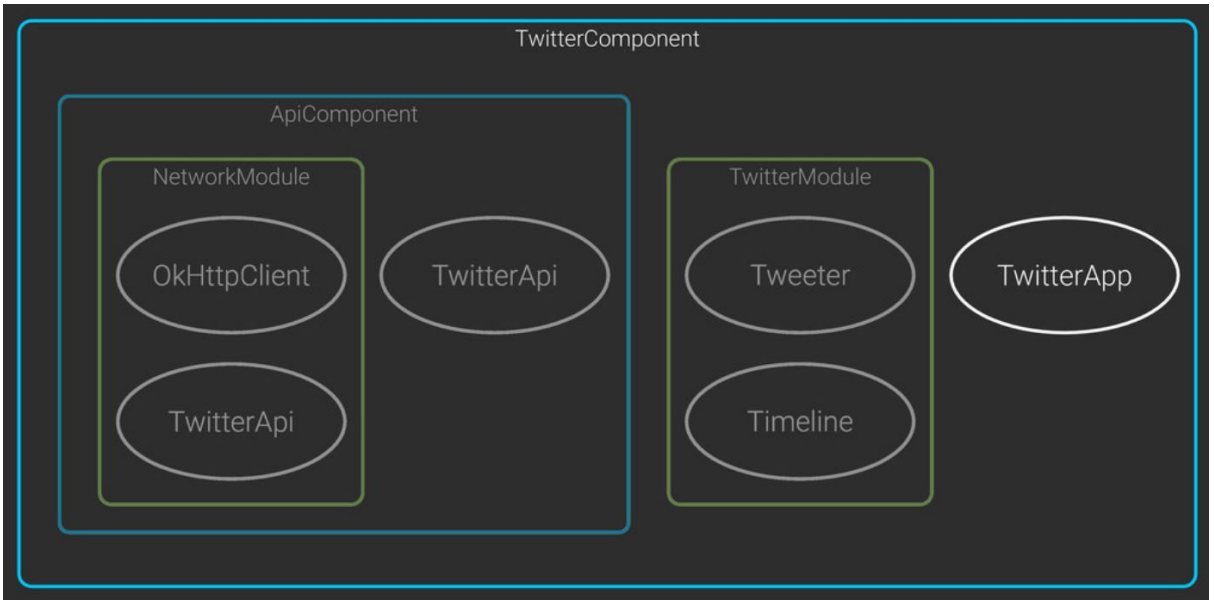
```
class Streaming @Inject constructor() {
  fun register(app: TwitterApplication) { /* ... */
  }
}
```

```
@Module
class TwitterModule(private val user: String) {
  @Provides
  @User
  fun provideTweeter(api: TwitterApi) = Tweeter(api, user)
```

```
  @Provides
  @User
  fun provideTimeline(api: TwitterApi) = Timeline(api, user)

  @Provides
  @User
  fun provideStreaming() = Streaming()
}
```

Por fim, temos nossa primeira aplicação com *Dagger*, contemplando a principal estrutura por trás dessa biblioteca que tem um potencial incrível para construir aplicações mais robustas seguindo os padrões mais atuais em termos de arquitetura de *software*. A estrutura atual dessa aplicação pode ser conferida na figura 9.

Figura 9: Estrutura da aplicação com injeção de dependência feita pelo *Dagger*



Fonte: *Dependency Injection with Dagger 2, by Jake Wharton*

8. Hands-On: MVVM, Architecture Components, RxAndroid e Dagger 2

Para concluir nosso aprendizado, vamos construir uma aplicação que contemple todos os principais itens aprendidos nesta aula, com alguns assuntos vistos na parte de *Architecture Components*, porém, agora em *Kotlin*.

Crie um novo projeto com suporte ao *Kotlin*, *API 15*, sem nenhuma *Activity*.

8.1. Configurando o projeto

Como vamos utilizar diversos componentes, já faremos a configuração toda de uma vez. Abra o *build.gradle* do projeto e do *app* e adicione os seguintes conteúdos, respectivamente:

```
buildscript {
    ext.kotlin_version = "1.2.51"
    ext.lifecycle_version = "1.1.1"
    ext.retrofit_version = "2.4.0"
    ext.dagger2_version = "2.16"
    ext.android_support_version = "27.1.1"
    ext.room_version = "1.1.1"
    // ...
}
```

```
// ...
apply plugin: "kotlin-kapt"

android {
  // ...
  dataBinding {
     enabled = true
  }
}

dependencies {
```

```
implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotli
n_version"
    implementation
"com.android.support:appcompat-v7:$android_support_version"
    testImplementation "junit:junit:4.12"
    androidTestImplementation
"com.android.support.test:runner:1.0.2"
    androidTestImplementation
"com.android.support.test.espresso:espresso-core:3.0.2"

  // Constraint Layout
  implementation
"com.android.support.constraint:constraint-layout:1.1.2"

  // Support Design
  implementation
"com.android.support:design:$android_support_version"

  // RecyclerView
  implementation
"com.android.support:recyclerview-v7:$android_support_version
"

   // LiveData & ViewModel
   implementation
"android.arch.lifecycle:extensions:$lifecycle_version"

  // Data Binding
  kapt "com.android.databinding:compiler:3.1.3"

  // Retrofit
  implementation
"com.squareup.retrofit2:retrofit:$retrofit_version"
  implementation
"com.squareup.retrofit2:adapter-rxjava2:$retrofit_version"
  implementation
"com.squareup.retrofit2:converter-moshi:$retrofit_version"
```

```
  // Dagger 2
  implementation "com.google.dagger:dagger:$dagger2_version"
  kapt "com.google.dagger:dagger-compiler:$dagger2_version"
  compileOnly "org.glassfish:javax.annotation:3.1.1"

  // Rx
  implementation "io.reactivex.rxjava2:rxjava:2.1.15"
  implementation "io.reactivex.rxjava2:rxandroid:2.0.2"

  // Room
  implementation
"android.arch.persistence.room:runtime:$room_version"
  kapt "android.arch.persistence.room:compiler:$room_version"
}

kapt {
  generateStubs = true
}
```

8.2. Bases

Crie um novo pacote chamado *base* e em seguida crie uma nova classe abstrata chamada *BaseViewModel* que estenda a classe *ViewModel* do *Architecture Components*. Deixe-a vazia por enquanto.

```
import android.arch.lifecycle.ViewModel

abstract class BaseViewModel: ViewModel() {
}
```

8.3. Model

Em seguida, crie um novo pacote chamado *model* na raiz da aplicação e crie a classe *Post*, que será nossa *data class*.

```
data class Post(val userId: Int,
                val id: Int,
                val title: String,
                val body: String)
```

8.4. Retrofit

Com o *model* declarado, vamos pegar uma lista de *Posts* da *JSON Placeholder API* [ref 20] utilizando o *Retrofit*. Também utilizaremos os *adapters* e *converters* do *Retrofit* com *RxJava* e para desserialização do *JSON* utilizaremos a biblioteca *Moshi*.

Crie um pacote chamado *network* e dentro cria uma interface chamada *PostApi*, que ficará responsável por obter os *Posts*.

```
interface PostApi {
  @GET("/posts")
  fun getPosts(): Observable<List<Post>>
}
```

Aproveitaremos a oportunidade para criar o arquivo *Constants.kt*, dentro de um pacote chamado *utils*, que conterá algumas constantes presentes para a aplicação. Começaremos inserindo a base da *url* da *API* que utilizaremos para fazer as requisições.

```
const val BASE_URL: String =
"https://jsonplaceholder.typicode.com"
```

8.5. Dagger 2

Iremos utilizar o *Dagger 2* para injetar as dependências do *Retrofit* na *ViewModel*. Crie um pacote chamado *injection* e dentro crie um outro pacote chamado *module*. Depois, crie uma *object* dentro do pacote *module* chamada *NetworkModule*, que será responsável pela injeção do *Retrofit*.

```
@Module
@Suppress("unused")
object NetworkModule {
  @Provides
  @Reusable
  @JvmStatic
  internal fun providePostApi(retrofit: Retrofit): PostApi {
     return retrofit.create(PostApi::class.java)
  }

  @Provides
  @Reusable
  @JvmStatic
  internal fun provideRetrofitInterface(): Retrofit {
     return Retrofit.Builder()
           .baseUrl(BASE_URL)

.addConverterFactory(MoshiConverterFactory.create())

.addCallAdapterFactory(RxJava2CallAdapterFactory.createWithSc
heduler(Schedulers.io()))
           .build()
  }
}
```

Note que nesse caso estamos usando a anotação **@Reusable**, que é semelhante ao *Singleton* e também a anotação **@JvmStatic**, que fará com que esse método seja gerado como estático pelo *Dagger 2*.

8.6. Post MVVM

Adicione um novo pacote *ui* no pacote raiz da aplicação e adicione um novo pacote dentro chamado *post*, onde criaremos todas as *views* e *view models* relacionados aos *posts*.

8.6.1. ViewModel component e injection

Dentro do novo pacote criado, crie uma classe chamada *PostListViewModel*, que será nosso *view model*. Por enquanto, apenas pegaremos resultados da *API* e exibiremos na *view*. Primeiro iremos criar uma implementação da interface *PostApi* para conseguir pegar os resultados da *API*. A instância será injetada pelo *Dagger*.

```
class PostListViewModel : BaseViewModel() {
  @Inject
  lateinit var postApi: PostApi
}
```

Agora crie um novo pacote *component* dentro do pacote *injection* e crie uma interface *ViewModelInjector* dentro dele.

```
@Singleton
@Component(modules = [(NetworkModule::class)])
interface ViewModelInjector {
  fun inject(postListViewModel: PostListViewModel)

  @Component.Builder
  interface Builder {
     fun build(): ViewModelInjector

     fun networkModule(networkModule: NetworkModule): Builder
  }
}
```

Tudo o que devemos fazer agora é adicionar a injeção das dependências necessárias na classe *BaseViewModel*.

```
abstract class BaseViewModel: ViewModel() {
  private val injector: ViewModelInjector =
DaggerViewModelInjector
        .builder()
        .networkModule(NetworkModule)
```

```
        .build()

  init {
     inject()
  }

  private fun inject() {
     when (this) {
        is PostListViewModel -> injector.inject(this)
     }
  }
}
```

Note que a classe *DaggerViewModelInjector* não é possível de ser importada. Isso se deve por conta que ela será gerada automaticamente pelo *Dagger* em tempo de compilação. Clique em *'Make Projec'* para que o *Dagger* gere a classe e você consiga importá-la.

8.6.2. Obtendo dados no ViewModel

Agora que a injeção de dependência está feita podemos pegar dados da *API*. Para isso nós precisaremos executar uma *background thread* enquanto os resultados serão replicados na *main thread*. Para realizar isso de maneira mais eficiente utilizaremos a biblioteca *RxJava*. Abra a classe *PostListViewModel* e adicione as seguintes declarações:

```
class PostListViewModel : BaseViewModel() {
  @Inject
  lateinit var postApi: PostApi

  private lateinit var subscription: Disposable

  init{
     loadPosts()
  }

  private fun loadPosts(){
```

```
    subscription = postApi.getPosts()
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .doOnSubscribe { onRetrievePostListStart() }
            .doOnTerminate { onRetrievePostListFinish() }
            .subscribe(
                    { onRetrievePostListSuccess() },
                    { onRetrievePostListError() }
            )
  }

  private fun onRetrievePostListStart(){
  }

  private fun onRetrievePostListFinish(){
  }

  private fun onRetrievePostListSuccess(){
  }

  private fun onRetrievePostListError(){
  }
}
```

8.6.3. ViewModel onCleared()

É realizar o *dispose()* da *subscription* quando o *view model* não for mais ser usado e está prestes a ser destruído. Para isso o *Android View Model* fornece o método *onCleared()* que é chamado exatamente quando isso ocorre.

```
class PostListViewModel : BaseViewModel() {
  // ...
  private lateinit var subscription: Disposable
  // ...
  override fun onCleared() {
     super.onCleared()
     subscription.dispose()
```

```
    }
    // ...
}
```

8.6.4. LiveData

Agora adicionaremos um *MutableLiveData* para observar a atualização da visibilidade da *ProgressBar* que mostraremos conforme os registros são carregados da *API*.

```
val loadingVisibility: MutableLiveData<Int> =
MutableLiveData()
// ...
private fun onRetrievePostListStart(){
    loadingVisibility.value = View.VISIBLE
}

private fun onRetrievePostListFinish(){
    loadingVisibility.value = View.GONE
}
```

8.6.5. Layout, Data Binding e BindingAdapters

Para essa parte utilizaremos *data binding* e *data binders* para atualizar as informações da nossa *view*, que utilizará um *ConstraintLayout* e uma *RecyclerView*.

Dentro da pasta *res* crie um novo diretório chamado *layout* e crie um novo arquivo de *layout* chamado *activity_post_list.xml*. Insira o conteúdo abaixo sem se esquecer de alterar o nome do pacote.

```
<?xml version="1.0" encoding="utf-8"?>
<layout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <data>
        <variable
            name="viewModel"
```

```
        type="package.PostListViewModel" />
  </data>
  <android.support.constraint.ConstraintLayout
     android:layout_width="match_parent"
     android:layout_height="match_parent">

     <ProgressBar
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"

app:mutableVisibility="@{viewModel.getLoadingVisibility()}"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />

     <android.support.v7.widget.RecyclerView
        android:id="@+id/post_list"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />
  </android.support.constraint.ConstraintLayout>
</layout>
```

Note que temos um atributo personalizado *mutableVisibility* na *ProgressBar*. Para que esse atributo funcione devemos criar um *BindingAdapter*. Para isso, crie um arquivo chamado *BindingAdapters.kt* no pacote *utils*. Dentro do arquivo definiremos um *DataBinder* para o atributo em questão.

```
@BindingAdapter("mutableVisibility")
fun setMutableVisibility(view: View, visibility:
MutableLiveData<Int>?) {
  val parentActivity: AppCompatActivity? =
view.getParentActivity()
```

```
  if (parentActivity != null && visibility != null) {
     visibility.observe(parentActivity, Observer { value ->
view.visibility = value ?: View.VISIBLE })
  }
}
```

Infelizmente não existe um método chamado *getParentActivity()*, portanto, devemos criá-lo. Para isso, adicionaremos uma função de extensão para a classe *View*. Crie um pacote chamado *extension* dentro do pacote *utils* e depois crie um arquivo chamado *ViewExtension.kt*.

```
fun View.getParentActivity(): AppCompatActivity? {
  var context = this.context
  while (context is ContextWrapper) {
     if (context is AppCompatActivity) {
        return context
     }
     context = context.baseContext
  }
  return null
}
```

8.6.6. PostListActivity

Dentro do pacote *ui* criaremos a classe da nossa *Activity*, chamada *PostListActivity*.

```
class PostListActivity : AppCompatActivity() {
  private lateinit var binding: ActivityPostListBinding
  private lateinit var viewModel: PostListViewModel

  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)

     binding = DataBindingUtil.setContentView(this,
R.layout.activity_post_list)
```

```
        binding.postList.layoutManager =
LinearLayoutManager(this, LinearLayoutManager.VERTICAL,
false)

        viewModel =
ViewModelProviders.of(this).get(PostListViewModel::class.java
)
        binding.viewModel = viewModel
    }
}
```

8.6.7. Manifesto

Para que nossa *Activity* funcione corretamente e para que a aplicação consiga se conectar à *internet* devemos registrar essas intenções no manifesto do *app*. Para adicionar a *Activity* no manifesto é possível clicar em cima do nome da *Activity* e selecionar a opção *'Add activity to manifest'*. Não se esqueça também de adicionar o *intent-filter* para que a *Activity* seja definida como a principal da aplicação.

```
<manifest ...>

  <uses-permission android:name="android.permission.INTERNET"
/>

  <application ...>
     <activity android:name=".ui.post.PostListActivity">
        <intent-filter>
           <action android:name="android.intent.action.MAIN"
/>

           <category
android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
     </activity>
  </application>
</manifest>
```

Agora podemos rodar nosso *app*. O resultado esperado nesse momento é a *ProgressBar* aparecendo e desaparecendo. Tudo funcionando? Perfeito! Mas estaria melhor se exibissimos a lista de *Posts*.

8.6.8. Tratando erros

Para exibir os erros quando a conexão não for estabelecida iremos utilizar uma *Snackbar*, permitindo com que o usuário reenvie a requisição de buscar os *Posts*. Antes de exibir o erro, adicione as seguintes *strings* no arquivo de *resources*.

```
<string name="retry">Retry</string>
<string name="post_error">An error occurred while loading the
posts</string>
```

Agora vamos adicionar um *MutableLiveData* para a mensagem do erro e um OnClickListener para ação do erro. Abra o arquivo *PostListViewModel* e adicione as seguintes informações:

```
val errorMessage: MutableLiveData<Int> = MutableLiveData()
val errorClickListener = View.OnClickListener { loadPosts() }
// ...
private fun onRetrievePostListStart() {
  loadingVisibility.value = View.VISIBLE
  errorMessage.value = null
}

private fun onRetrievePostListFinish() {
  loadingVisibility.value = View.GONE
}

private fun onRetrievePostListSuccess() {
}

private fun onRetrievePostListError() {
  errorMessage.value = R.string.post_error
}
```

Agora precisamos observar o valor da mensagem de erro para exibir a *Snackbar*. Abra o arquivo *PostListActivity* e faça as seguintes alterações:

```
class PostListActivity : AppCompatActivity() {
  private lateinit var binding: ActivityPostListBinding
  private lateinit var viewModel: PostListViewModel

  private var errorSnackbar: Snackbar? = null

  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)

     binding = DataBindingUtil.setContentView(this,
R.layout.activity_post_list)
     binding.postList.layoutManager =
LinearLayoutManager(this, LinearLayoutManager.VERTICAL,
false)

     viewModel =
ViewModelProviders.of(this).get(PostListViewModel::class.java
)
     viewModel.errorMessage.observe(this, Observer {
errorMessage ->
        if (errorMessage != null) {
           showError(errorMessage)
        } else {
           hideError()
        }
     })
     binding.viewModel = viewModel
  }

  private fun showError(@StringRes errorMessage: Int) {
     errorSnackbar = Snackbar.make(binding.root,
errorMessage, Snackbar.LENGTH_INDEFINITE)
     errorSnackbar?.setAction(R.string.retry,
viewModel.errorClickListener)
     errorSnackbar?.show()
```

```
    }

    private fun hideError() {
        errorSnackbar?.dismiss()
    }
}
```

Rode a aplicação em modo avião para testar a implementação da *Snackbar*. O resultado deve ser semelhante à figura 10.

Figura 10: Mensagem de erro na *Snackbar*
Fonte: autor - *Android Studio (API 27)*

8.6.9. Exibindo a lista de Posts

Finalmente vamos exibir a lista de *Posts*! Para fazer isso, vamos criar um *View Model* para os itens da lista. Crie uma classe chamada *PostViewModel* dentro do pacote *ui.post*.

```
class PostViewModel : BaseViewModel() {
  private val postTitle = MutableLiveData<String>()
  private val postBody = MutableLiveData<String>()

  fun bind(post: Post) {
     postTitle.value = post.title
     postBody.value = post.body
  }

  fun getPostTitle(): MutableLiveData<String> {
     return postTitle
  }

  fun getPostBody(): MutableLiveData<String> {
     return postBody
  }
}
```

Agora crie o *layout* para os itens, com o nome *item_post.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<layout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">
  <data>
     <variable
        name="viewModel"
        type="package.PostViewModel" />
  </data>

  <android.support.constraint.ConstraintLayout
```

```
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:paddingLeft="16dp"
      android:paddingRight="16dp">

      <TextView
         android:id="@+id/post_title"
         android:layout_width="0dp"
         android:layout_height="wrap_content"
         android:textStyle="bold"
         app:layout_constraintEnd_toEndOf="parent"
         app:layout_constraintStart_toStartOf="parent"
         app:layout_constraintTop_toTopOf="parent"
         app:mutableText="@{viewModel.getPostTitle()}" />

      <TextView
         android:id="@+id/post_body"
         android:layout_width="0dp"
         android:layout_height="wrap_content"
         android:layout_marginTop="8dp"
         app:layout_constraintEnd_toEndOf="parent"
         app:layout_constraintStart_toStartOf="parent"
         app:layout_constraintTop_toBottomOf="@+id/post_title"
         app:mutableText="@{viewModel.getPostBody()}" />
   </android.support.constraint.ConstraintLayout>
</layout>
```

Agora precisamos editar o arquivo *BindingAdapters.kt* para adicionar um *Binding Adapter* para o atributo *app:mutableText*.

```
//...
@BindingAdapter("mutableText")
fun setMutableText(view: TextView, text:
MutableLiveData<String>?) {
   val parentActivity: AppCompatActivity? =
view.getParentActivity()
   if (parentActivity != null && text != null) {
```

```
    text.observe(parentActivity, Observer { value ->
view.text = value ?: "" })
  }
}
```

8.6.10. Adapter e ViewModel

Crie uma nova classe chamada *PostListAdapter* no pacote *ui.post* para adicionarmos o *adapter* da *RecyclerView*.

```
class PostListAdapter :
RecyclerView.Adapter<PostListAdapter.ViewHolder>() {
  private lateinit var postList: List<Post>

  override fun onCreateViewHolder(parent: ViewGroup,
viewType: Int): PostListAdapter.ViewHolder {
    val binding: ItemPostBinding =
DataBindingUtil.inflate(LayoutInflater.from(parent.context),
R.layout.item_post, parent, false)
    return ViewHolder(binding)
  }

  override fun onBindViewHolder(holder:
PostListAdapter.ViewHolder, position: Int) {
    holder.bind(postList[position])
  }

  override fun getItemCount(): Int {
    return if (::postList.isInitialized) {
      postList.size
    } else {
      0
    }
  }

  fun updatePostList(postList: List<Post>) {
    this.postList = postList
```

```
        notifyDataSetChanged()
    }

    class ViewHolder(private val binding: ItemPostBinding) :
RecyclerView.ViewHolder(binding.root) {
        fun bind(post: Post) {
            // ...
        }
    }
}
```

Agora vamos construir a classe *ViewHolder*. Nós sabemos como pegar o *ViewModel* associado com a *Activity* porque adicionamos o método de extensão *View.getParentActivity()* e seria fácil obter a instância do *PostViewModel* utilizando o *ViewModelProviders(...)*. No entanto, o *ViewModel* associado com a *Activity* é *singleton*, portanto, utilizá-lo assim iria fazer com que a lista mostrasse a mesma informação para todas as linhas.

Entretanto, por estarmos usando um *MutableLiveData*, não precisamos instanciar um novo *PostViewModel* para cada item, e sim para cada instância do *ViewHolder*.

```
class ViewHolder(private val binding: ItemPostBinding) :
RecyclerView.ViewHolder(binding.root) {
    private val viewModel = PostViewModel()

    fun bind(post: Post){
        viewModel.bind(post)
        binding.viewModel = viewModel
    }
}
```

8.6.11. Definindo o adapter na RecyclerView

Precisamos definir o *PostListAdapter* na *RecyclerView* e atualizar os dados na medida que forem chegando. Para fazer isso, primeiro nós precisamos editar a classe *PostListViewModel*.

```
val postListAdapter: PostListAdapter = PostListAdapter()

private fun loadPosts() {
  subscription = postApi.getPosts()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .doOnSubscribe { onRetrievePostListStart() }
        .doOnTerminate { onRetrievePostListFinish() }
        .subscribe(
             { result -> onRetrievePostListSuccess(result) },
             { onRetrievePostListError() }
        )
}
// ...
private fun onRetrievePostListSuccess(postList: List<Post>) {
  postListAdapter.updatePostList(postList)
}
// ...
```

Agora, abra o arquivo de *layout activity_post_list.xml* e adicione o adapter diretamente na *RecyclerView*.

```
<android.support.v7.widget.RecyclerView
  android:id="@+id/post_list"
  android:layout_width="0dp"
  android:layout_height="0dp"
  app:adapter="@{viewModel.getPostListAdapter()}"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent" />
```

E depois adicione o *BindingAdapter* correspondente.

```
@BindingAdapter("adapter")
fun setAdapter(view: RecyclerView, adapter:
RecyclerView.Adapter<*>) {
```

```
    view.adapter = adapter
}
```

Rode a aplicação e veja a lista de *Posts* carregada a partir da *API*. O resultado deve ser semelhante ao da figura 11.

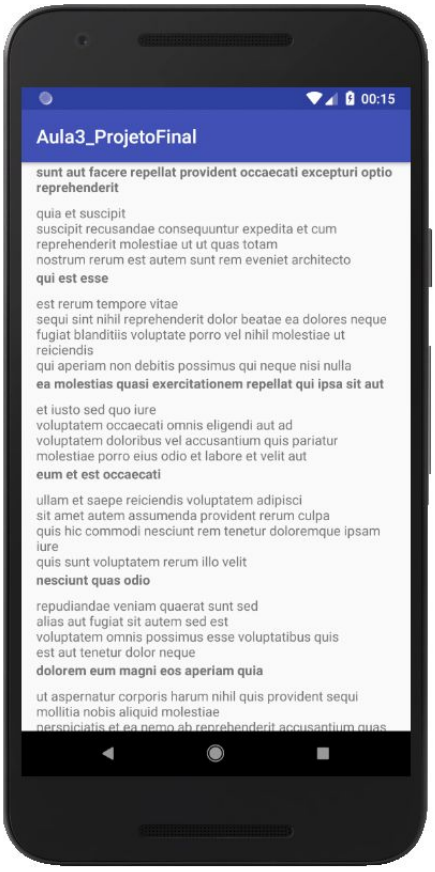


Figura 11: Lista de *Posts* carregada via *API*
Fonte: autor - *Android Emulator (API 27)*

8.7. Room

Uma funcionalidade bem interessante para nossa aplicação seria a utilização do *Room* para armazenar de forma *offline* os registros da *API*. Para isso, adicionaremos e configuraremos o componente *Room*.

8.7.1. Entity

Abra o arquivo da *data class Post*.Devemos anotá-lo com **@Entity**, simbolizando que é uma entidade do *Room*. Também adicionaremos a marcação de chave primária para o campo *id*.

```
import android.arch.persistence.room.Entity
import android.arch.persistence.room.PrimaryKey

@Entity
data class Post(val userId: Int,
                @field:PrimaryKey
                val id: Int,
                val title: String,
                val body: String)
```

8.7.2. Dao

A classe *Dao* servirá para nos permitir inserir e obter *Posts* a partir do *database*. Para fazer isso, precisamos criar uma interface chamada *PostDao* no pacote *model*.

```
import android.arch.persistence.room.Dao
import android.arch.persistence.room.Insert
import android.arch.persistence.room.Query

@Dao
interface PostDao {
  @get:Query("SELECT * FROM post")
  val all: List<Post>

  @Insert
  fun insertAll(vararg posts: Post)
}
```

8.7.3. Database

Para completar, crie um pacote chamado *database* dentro do pacote *model* e adicione uma classe chamada *AppDatabase*.

```
@Database(entities = [(Post::class)], version = 1)
abstract class AppDatabase : RoomDatabase() {
  abstract fun postDao(): PostDao
}
```

8.8. Context Dependent instances no ViewModel

Precisamos de uma instância da interface *PostDao* para obter os *Posts* a partir do database e também para inserí-los. Para fazer isso, vamos adicionar um novo argumento no construtor do *PostListViewModel*. Tudo que devemos fazer depois é chamar os métodos *insert()* e *all()* da instância do *PostDao*.

```
class PostListViewModel(private val postDao: PostDao) :
BaseViewModel() {
  // ...
  private fun loadPosts() {
     subscription = Observable.fromCallable { postDao.all }
           .concatMap { dbPostList ->
              if (dbPostList.isEmpty())
                 postApi.getPosts().concatMap { apiPostList
-> postDao.insertAll(*apiPostList.toTypedArray())
                    Observable.just(apiPostList)
                 }
              else
                 Observable.just(dbPostList)
           }
           .subscribeOn(Schedulers.io())
           .observeOn(AndroidSchedulers.mainThread())
           .doOnSubscribe { onRetrievePostListStart() }
           .doOnTerminate { onRetrievePostListFinish() }
           .subscribe(
```

```
            { result -> onRetrievePostListSuccess(result) },
            { onRetrievePostListError() }
        )
  }
  // ...
```

8.9. ViewModelProvider.Factory

Por último, precisamos criar e declarar o *ViewModelFactory*, para saber como instanciar o *ViewModel*. Dentro do pacote *injection*, crie a classe *ViewModelFactory*.

```
class ViewModelFactory(private val activity:
AppCompatActivity) : ViewModelProvider.Factory {
  override fun <T : ViewModel?> create(modelClass: Class<T>):
T {
     if
(modelClass.isAssignableFrom(PostListViewModel::class.java))
{
        val db =
Room.databaseBuilder(activity.applicationContext,
AppDatabase::class.java, "posts").build()
        @Suppress("UNCHECKED_CAST")
        return PostListViewModel(db.postDao()) as T
     }
     throw IllegalArgumentException("Unknown ViewModel
class")
  }
}
```

Depois, abra o arquivo da *Activity* e atualize a declaração do *ViewModelProvider*.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  // ...
```

```
  viewModel = ViewModelProviders.of(this,
ViewModelFactory(this)).get(PostListViewModel::class.java)
  // ...
}
```

Ufa! Terminamos! Pode testar a aplicação. Uma vez que a lista de *Post* foi carregada, mesmo que na próxima execução não tenha acesso à internet ainda sim poderemos navegar pelo conteúdo carregado.Esse projeto teve como referência o projeto *MVVM Posts*, publicado por *Gahfy* no *blog ProAndroidDev* [ref 19].

9. Referências Bibliográficas

1. Model–view–controller - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller (Acessado em 24 de Julho de 2018)
2. Model–view–presenter - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter (Acessado em 24 de Julho de 2018)
3. Model–view–viewmodel - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel (Acessado em 24 de Julho de 2018)
4. The MVC, MVP, and MVVM Smackdown - https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/ (Acessado em 24 de Julho de 2018)
5. Data Binding Library - https://developer.android.com/topic/libraries/data-binding/ (Acessado em 24 de Julho de 2018)
6. AS 3.1 RC2 report generated folders from kapt as warnings - https://issuetracker.google.com/issues/74537216 (Acessado em 26 de Julho de 2018)
7. 3rd-party Gradle plug-ins may be the cause - https://stackoverflow.com/questions/49518223/3rd-party-gradle-plug-ins-may-be-the-cause (Acessado em 26 de Julho de 2018)
8. Android Studio 3.2 Beta 1 available - https://androidstudio.googleblog.com/2018/06/android-studio-32-beta-1-available.html (Acessado em 26 de Julho de 2018)
9. Reactive programming - https://en.wikipedia.org/wiki/Reactive_programming (Acessado em 24 de Julho de 2018)
10. RxNetty vs Tomcat Performance Results - https://pt.slideshare.net/brendangregg/rxnetty-vs-tomcat-performance-results (Acessado em 24 de Julho de 2018)
11. Reactive Programming in the Netflix API with RxJava - https://medium.com/netflix-techblog/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a (Acessado em 24 de Julho de 2018)
12. ReactiveX - https://reactivex.io/ (Acessado em 24 de Julho de 2018)
13. The Reactive Manifesto - https://www.reactivemanifesto.org/ (Acessado em 24 de Julho de 2018)
14. Conhecendo Java Reflection - https://www.devmedia.com.br/conhecendo-java-reflection/29148 (Acessado em 25 de Julho de 2018)
15. Talk: RxJava in Baby Steps - http://adavis.info/2017/10/talk-rxjava-in-baby-steps.html (Acessado em 27 de Julho de 2018)
16. The Operators of ReactiveX - http://reactivex.io/documentation/operators.html

(Acessado em 27 de Julho de 2018)
17. RxJava 2 – Flowable - http://www.baeldung.com/rxjava-2-flowable (Acessado em 27 de Julho de 2018)
18. Dealing with Backpressure with RxJava - http://www.baeldung.com/rxjava-backpressure (Acessado em 27 de Julho de 2018)
19. MVVM with Kotlin — Android Architecture Components, Dagger 2, Retrofit and RxAndroid - https://proandroiddev.com/mvvm-with-kotlin-android-architecture-components-dagger-2-retrofit-and-rxandroid-1a4ebb38c699 (Acessado em 27 de Julho de 2018)
20. JSON Placeholder API - https://jsonplaceholder.typicode.com/ (Acessado em 27 de Julho de 2018)

10. Vídeos Recomendados

1. Lisa Wray — Data Binding in a Kotlin world - https://youtu.be/TW9dSEgJIa8 (Acessado em 24 de Julho de 2018)
2. Droidcon NYC 2016 - Advanced Data Binding in Practice - https://youtu.be/u8d_zXukB2w (Acessado em 27 de Julho de 2018)
3. NetflixOSS Meetup - Season 2 Episode 2 - Reactive / Async - https://youtu.be/aEuNBk1b5OE (Acessado em 24 de Julho de 2018)
4. KotlinConf 2017 - RX Java with Kotlin in Baby Steps by Annyce Davis - https://youtu.be/YPf6AYDaYf8 (Acessado em 27 de Julho de 2018)
5. Learning RxJava (for Android) by example - https://youtu.be/k3D0cWyNno4 (Acessado em 27 de Julho de 2018)
6. GOTO 2016 • Exploring RxJava 2 for Android • Jake Wharton - https://youtu.be/htIXKI5gOQU (Acessado em 27 de Julho de 2018)
7. The Future of Dependency Injection with Dagger 2 - https://youtu.be/plK0zyRLIP8 (Acessado em 25 de Julho de 2018)

11. Licença e termos de uso

12. Leitura adicional

Além de todas as referências, sugiro que leia alguns materiais adicionais para aumentar o seu conhecimento sobre alguns assuntos específicos.

12.1. Architecture Principles - MVC, MVP e MVVM

- Desmistificando o MVC e MVP no Android - https://medium.com/android-dev-br/desmistificando-o-mvc-e-mvp-no-android-abe927d01df7
- The MVC, MVP, and MVVM Smackdown - https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

12.2. Data Binding

- Android Data Binding Advanced Concepts - https://www.journaldev.com/11950/android-data-binding-advanced-concepts

12.3. Dependency Injection com Dagger 2

- Jake Wharton Presentations - https://jakewharton.com/presentations/
- Introdução ao Dagger 2 - https://medium.com/android-dev-br/introdu%C3%A7%C3%A3o-ao-dagger-2-56d193118a6c
- Tasting Dagger 2 on Android - https://fernandocejas.com/2015/04/11/tasting-dagger-2-on-android/

12.4. ReactiveX

- Programação Reativa (Reactive Programming) - http://blog.andrefaria.com/programacao-reativa-reactive-programming
- Reactive Programming para Desenvolvedores Android — Part I~III
    - https://medium.com/google-developer-experts/reactive-programming-para-desenvolvedores-android-part-i-53788a4fbc9f
    - https://medium.com/google-developer-experts/reactive-programming-para-desenvolvedores-android-part-2-7cda0679b9
    - https://medium.com/google-developer-experts/functional-reactive-programming-para-desenvolvedores-android-part-3-461c2f558565