

Desenvolvido por [Paulo Salvatore](#)

Movile Next Program

Formação em Desenvolvimento *Android*

Prof. Paulo Salvatore

Tópicos abordados:

- *Architecture Principles (KISS, YAGNI, DRY e The Boy Scout Rule)*
- *Push Notification com FCM*
- *Image Loading*
- *Networking*

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 princípios importantíssimos aplicados em diversos aspectos na vida e que são fundamentais para um bom desenvolvimento inclusive durante o desenvolvimento de aplicações e códigos de uma maneira geral.

Prosseguindo no desenvolvimento *Android*, iniciaremos o desenvolvimento com as *Push Notifications* presentes o tempo inteiro no sistema *Android*. Faremos a integração com o *Firebase*, que ficará responsável pelo envio de mensagens através da sua *API*, onde construiremos um código capaz de receber as mensagens e construir notificações personalizadas para os usuários.

Depois, discutiremos diversos aspectos envolvendo o *image loading*, discutindo problemas comuns na implementação, abordagens possíveis e as principais bibliotecas disponíveis para realizar o carregamento de imagens da melhor maneira possível. Também mostraremos algumas maneiras de selecionar imagens armazenadas no dispositivo ou solicitando para o usuário tirar uma nova, utilizando abordagens recomendadas pelo *Google* na documentação *Android*.

Por fim, falaremos sobre *networking*, ou seja, abordagens para comunicação com o protocolo *web*, também um assunto com diversos aspectos e abordagens possíveis, com várias complexidades e diversas bibliotecas disponíveis que trabalham na resolução de problemas específicos envolvendo algumas

Desenvolvido por [Paulo Salvatore](#)

implementações que vão desde obtendo de dados da *web* até a serialização desses dados em formatos característicos.

No final desse material há também um projeto extra, que deixo como desafio proposto para resolução que contemplará todos os assuntos abordados e fará com que você pratique o desenvolvimento com a linguagem de programação *Kotlin*, que eu garanto antecipadamente: se você não conhece, irá amar!

Esse material foi desenvolvido com muito carinho e cuidado, espero que ele possa contribuir com a sua evolução!

3. Sumário

1. Mobile Next Program	1
2. Introdução	1
3. Sumário	3
4. Architecture Principles	5
4.1. KISS	5
4.2. YAGNI	6
4.3. DRY	7
4.4. The Boy Scout Rule	7
5. Push Notification com FCM	9
5.1. Configurando o ambiente Android	9
5.1.1. Configuração do build.gradle	10
5.1.2. Configuração do Android Manifest	13
5.2. Implementação do FirebaseMessagingService	14
5.3. Criação da classe NotificationCreation	16
6. Image Loading	22
6.1. Problemas comuns e possíveis soluções	22
6.2. Bibliotecas para carregamento de imagens	23
6.2.1. Qual delas usar? Qual é a melhor?	23
6.2.2. Glide implementation	24
6.2.3. Picasso implementation	24
6.3. Carregar imagem da galeria ou tirando uma nova	25
7. Networking	34
7.1. REST	34
7.2. JSON	35
7.2.1. JSON Visualizer	37
7.3. Serialization	37
7.4. Bibliotecas existentes: Networking	38
7.4.1. Retrofit	38
7.4.2. Volley	39
7.4.3. Considerações adicionais	40
7.5. Bibliotecas existentes: Serialization	40
7.6. Exemplo prático	41

7.7. Network Profiling	61
8. Projeto extra	63
9. Referências Bibliográficas	68
10. Vídeos Recomendados	71
11. Licença e termos de uso	72
12. Leitura adicional	73
12.1. Architecture Principles	73
12.1.1. KISS	73
12.1.2. YAGNI	73
12.1.3. DRY	73
12.1.4. The Boy Scout Rule	73
12.2. Kotlin	73
12.3. Networking	74
12.4. Assuntos diversos	74

4. Architecture Principles

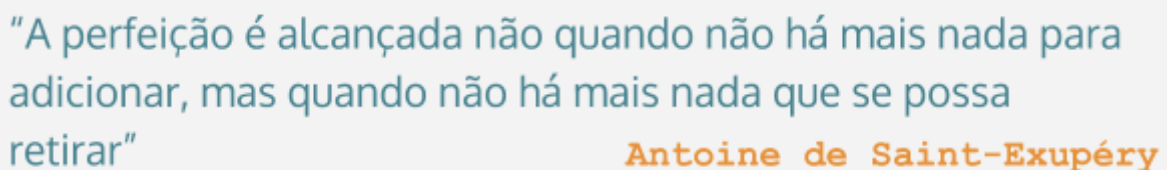
Durante diversos momentos do curso, alguns conceitos importantes de arquitetura, princípios de programação e desenvolvimento ágil serão apresentados. Nessa segunda aula falaremos sobre alguns conceitos bem simples mas que são bem importantes de saber e levar em consideração em vários aspectos do desenvolvimento de *softwares*.

Neste capítulo iremos falar sobre os princípios *KISS*, *YAGNI*, *DRY* e *The Boy Scout Rule*. Diversas referências de leitura sobre eles podem ser encontradas no capítulo de [leitura adicional](#) no final desse material.

4.1. KISS

KISS, conhecido como princípio do beijo, acrônimo para "*Keep It Simple, Stupid*", significa "Mantenha simples, estúpido". Esse princípio valoriza a simplicidade do objeto, muito conhecido na área de *design* como "menos é mais". Como o *Stefan Mischook* cita no seu vídeo 6 ([ir para vídeo](#)) sobre o *KISS*, a grande questão que devemos pensar é: o que deixar de fora? Qual parte do código é desnecessária?

Um grande pensamento que gosto de ter enquanto estou programando é qual parte realmente do código que estou escrevendo é necessária. Em alguns casos, quando estamos construindo um método muito grande, talvez quebrá-lo em duas etapas pode ser a abordagem mais interessante em termos de organização e clareza no que estamos tentando dizer.



"A perfeição é alcançada não quando não há mais nada para adicionar, mas quando não há mais nada que se possa retirar"

Antoine de Saint-Exupéry

Figura 1: "A perfeição é alcançada não quando não há mais nada para adicionar, mas quando não há mais nada que se possa retirar"

Fonte: *Terre des Hommes* (1939), por *Antoine de Saint-Exupéry*

Esse princípio é um dos mais importantes porque dita a simplicidade de algo. Um caso em particular com um grande amigo que foi meu professor na faculdade e inclusive tenho o prazer de lecionar ao lado dele em diversas oportunidades, quando estávamos desenvolvendo nossos projetos ou buscando soluções de

problemas e encontrávamos uma maneira de resolver aquele problema de maneira extremamente simples ele me dizia a frase: "*The simplest explanation tends to be the right one*". A explicação mais simples tende a ser a correta. Isso é fundamental para quando resolvemos um problema de uma maneira extremamente simples e tendemos a complicá-lo buscando resolver algo que já está resolvido.

4.2. YAGNI

YAGNI, acrônimo para "*You Ain't Gonna Need It*", é um princípio muito importante no que diz respeito à adição de funcionalidades. Ele nos diz para implementar funcionalidades apenas quando elas são extremamente necessárias. Sempre que estamos prevendo o que pode ser necessário é sinônimo de tempo gasto a toa. Mesmo que consigamos prever um problema, nunca conseguiremos solucioná-lo de fato antes que ele aconteça.

Usage of Features and Functions in Typical System

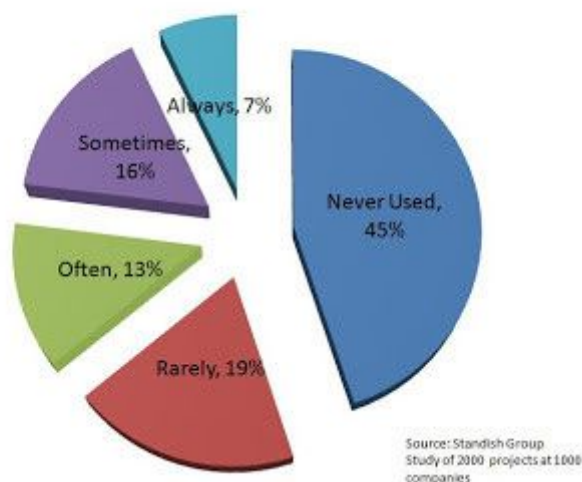


Figura 2: Uso de funcionalidades disponíveis em um sistema comum

Fonte: *StandishGroup, 2010*, acessado em [[ref 31](#)]

Outra fase importante é: faça a coisa mais simples possível que possa funcionar; evite implementações complexas antes que elas sejam necessárias. Quando a necessidade real surgir, podemos focar um tempo e pensar na melhor maneira de resolvê-la. Essa dica é fundamental e pode te poupar tempo, cabelo e diminuir consideravelmente o número de *bugs* que podem ocorrer tanto durante o desenvolvimento da aplicação quanto depois que ela está em uso.



Figura 3: Charge sobre a complexidade das funcionalidades implementadas

Fonte: *Dilbert.com*, acessado em [ref 31]

4.3. DRY

.*DRY*, acrônimo para "*Don't Repeat Yourself*", propõe a redução da repetição de código durante o desenvolvimento. No livro "*The Pragmatic Programmer*", escrito por *Andy Hunt* e *Dave Thomas*, o conceito foi sintetizado como: "Cada parte de conceito deve possuir uma representação única, autoritária e livre de ambiguidades dentro de um sistema.

Uma estratégia básica para reduzir a complexidade em unidades gerenciáveis com intuito de dividir o sistema em partes. A ideia é que um componente muito grande deve ser dividido em sub componentes que não mais fáceis de manter e possui uma funcionalidade mais específica, melhorando o entendimento e tornando mais simples sua implementação.

É recomendado que você pense nesse princípio sempre que perceber que está escrevendo um código que é bem similar ou igual a algo que já escreveu antes. Quando isso ocorrer, recomenda-se que pare um momento para refletir no que está fazendo e evitar a repetição.

Em oposto a essa solução, existe uma prática conhecida com *WET*, acrônimo para "*Write Everything Twice*", "*We Enjoy Typing*" ou até mesmo "*Waste Everyone's Time*", que basicamente diz completamente o oposto e geralmente é aplicada em algumas arquiteturas com múltiplas camadas como alguns componentes da *web*, onde uma mesma declaração de um nome para uma *label* aparece em outros lugares do *Form*, nomenclatura de função, variável ou no banco de dados.

4.4. The Boy Scout Rule

Finalizando esse capítulo de princípios de arquitetura com chave de ouro temos uma famosa regra escrita pelo *Robert Cecil Martin*, conhecido na comunidade como *Uncle Bob*, famoso escritor de livros sobre boas práticas em

Desenvolvido por [Paulo Salvatore](#)

termos de código, como "Código Limpo", "O Codificador Limpo" e "Clean Architecture: A Craftsman's Guide to Software Structure and Design", livros de cabeceira para qualquer programador.



Figura 4: "Deixa as coisas MELHORES do que quando você encontrou elas"

Fonte: *Robert Baden Powell*

O próprio nome "*The Boy Scout Rule*" [ref 32], algo como "a regra do bom escoteiro" já nos desperta esse sentimento de pacificadores em busca de um ambiente mais agradável. A regra muito comum que não ouvimos isso de hoje, é uma boa prática de boa convivência em sociedade, mas que se aplica muito no código, apesar de não aplicarmos quase nunca. A partir de hoje, quando estiver algum pedaço de *software*, mesmo que não seja você que tenha escrito, caso encontre algo que caiba uma melhora (lembre-se: levando todas as boas práticas em consideração), não hesite e faça! Deixe comentários caso necessário, mas busque sempre pela melhora!

Não importa o quão experiente é o programador, sempre estamos limitados ao prazo de desenvolvimento da aplicação, quase sempre apertado, deixando muito suscetível a inserção de trechos de código que nos trarão problemas no futuro ou que podem apresentar falhas visíveis, também conhecidos como *software rot* [ref 33].

Desenvolver software não é apenas aplicar o conhecimento que aprendemos. É como ter um Bonsai, que deve ser podada constantemente e cuidada. Cuidar do próprio código é trabalho dos desenvolvedores, mas cuidar e se comportar como dono do código da equipe é bem diferente.



Figura 5: "*Kaizen*: As simples e pequenas mudanças que resultam em uma vida inteira de melhorias"

Fonte: Filosofia Japonesa

5. Push Notification com FCM

Como sabemos, as notificações no *Android* [ref 18] estão presentes o tempo inteiro e basicamente são a base de entrada para interações com diversos aplicativos, promovendo engajamento e contribuindo muito com a experiência do usuário. Muito por conta disso, do lado *development*, elas recebem atenção constante passando por diversas melhorias em *APIs* recentes e sendo possível criar diversas funcionalidades interessantes.

Por muito tempo o *Google* utilizou a tecnologia conhecida como *GCM* (*Google Cloud Messaging*) para o envio de mensagens remotas para aplicações, não se limitando ao escopo *Android*, funcionando também para ambiente *iOS* e *Chrome*. Em 2018, o *Google* depreciou a biblioteca *GCM* [ref 10], anunciando sua remoção completa em 2019. Isso aconteceu pois, na *Google I/O 16'*, foi anunciado que serviço de mensagem do *Firebase*, o *FCM* (*Firebase Cloud Messaging*) [ref 11], substituiria esse tipo de implementação. Atualmente o *FCM* está disponível para *Android*, *iOS*, *Web* (implementação customizada), *C++* e *Unity*.

Desde então utilizamos toda a base do projeto *Firebase Messaging* como biblioteca de envio de mensagens remotas para o *Android*. Essas mensagens são recebidas e formatadas em notificações, conhecidas como *Push Notifications*, que atualmente são feitas através da classe *NotificationCompat*, que permite uma série de customizações, todas com a finalidade de melhorar a interação do usuário com a aplicação que está sendo desenvolvida.

5.1. Configurando o ambiente Android

A configuração do ambiente *Android* para utilização do *FCM* é muito simples e o próprio site possui um tutorial extremamente completo para realização dessa etapa. Para começar, crie um novo projeto no *Android Studio* com suporte ao *Kotlin*, *API 15* e uma *EmptyActivity* chamada *MainActivity*.

Com o projeto criado, certifique-se de possui a última versão do *Google Play Services* instalada. Para isso, vá nas configurações de *SDK*, selecione a aba *SDK Tools* e procure pela ferramenta. Caso não tenha ou precise atualizar, marque a caixinha e clique em *Apply*. Caso seja necessário, reinicie o *Android Studio*. Após isso, precisamos configurar o ambiente do *Firebase* e fazer o *download* do arquivo *json* de configuração.

Acesse o *Console* do *Firebase* através do link abaixo e siga o passo-a-passo.

Link: <https://console.firebase.google.com>

Desenvolvido por [Paulo Salvatore](#)

1. Adicione um novo projeto;
2. Após criar, na tela aberta, clique em 'Adicionar o Firebase ao app para o Android';
3. Informe o pacote do *app* (exatamente como está lá) - os outros campos são todos opcionais - e clique em 'Registrar app';
4. Faça o *download* do arquivo 'google-services.json' e:
 - a. Altere a visão do projeto no *Android Studio* para *Project*;
 - b. Arraste o arquivo para a pasta 'app'.
5. Abra os arquivos 'build.gradle' e insira o conteúdo informado (também estão referenciados abaixo);
6. Sincronize, abra o *app* e espere o *Firebase* encontrar o seu projeto;
7. Apareceu a mensagem de conectado? Pronto! O *Firebase* está configurado. A mensagem deve estar de acordo com a figura 10;
8. Procure pelo serviço *Notification*, clique em 'Começar' e aguarde! - em breve testaremos o envio de notificações pelo *Firebase*.

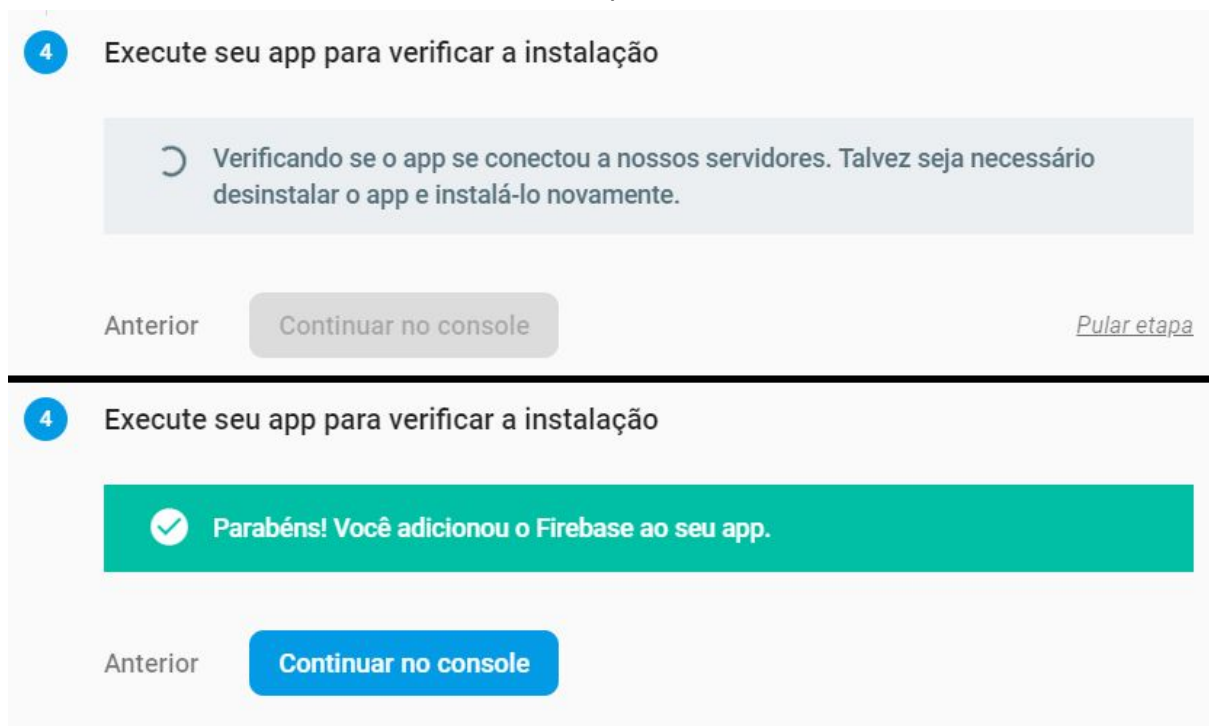


Figura 10: Firebase conectado à aplicação

Fonte: autor - *Firebase Console*

5.1.1. Configuração do build.gradle

Ainda no arquivo *build.gradle* do *app*, além do *firebase-core*, devemos importar o *firebase-messaging*, conforme o trecho abaixo:

```
implementation  
"com.google.firebase:firebase-messaging:17.1.0"
```

Além do suporte ao firebase, certifique-se de atualizar o suporte ao *Kotlin* *JDK7*, atualizar sua versão e importar a biblioteca *anko-commons*. Também utilizarei algumas versões e implementações provavelmente diferentes das suas. Se quiser estar de acordo com o meu projeto, coloquei aqui configurações iniciais para o *build.gradle* do projeto e do *app*.

Arquivo *build.gradle* do projeto:

```
buildscript {  
    ext {  
        // Gradle  
        gradleVersion = "3.1.3"  
  
        // Kotlin  
        kotlinVersion = "1.2.51"  
        ankoVersion = "0.10.5"  
  
        // Support  
        supportLibVersion = "27.1.1"  
        constraintVersion = "1.1.2"  
  
        // Testing  
        junitVersion = "4.12"  
        runnerVersion = "1.0.2"  
        espressoVersion = "3.0.2"  
  
        // Firebase  
        firebaseCoreVersion = "16.0.1"  
        firebaseMessagingVersion = "17.1.0"  
  
        // Google Play Services  
        googleServicesVersion = "4.0.0"  
        googlePlayVersion = "15.0.1"  
    }  
}
```

```
repositories {
    google()
    jcenter()
}
dependencies {
    classpath
"com.android.tools.build:gradle:$gradleVersion"
    classpath
"org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlinVersion"
    classpath
"com.google.gms:google-services:$googleServicesVersion"
}
}
```

O *build.gradle* do *app* está com *API* mínima 15 e *target* 27, além das configurações:

```
dependencies {
    implementation fileTree(dir: "libs", include: ["*.jar"])
    implementation
"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlinVersion"
    implementation
"com.android.support:appcompat-v7:$supportLibVersion"
    implementation
"com.android.support.constraint:constraint-layout:$constraint
Version"
    testImplementation "junit:junit:$junitVersion"
    androidTestImplementation
"com.android.support.test:runner:$runnerVersion"
    androidTestImplementation
"com.android.support.test.espresso:espresso-core:$espressoVer
sion"

    // Anko
    implementation
"org.jetbrains.anko:anko-commons:$ankoVersion"
    implementation
"org.jetbrains.anko:anko-design:$ankoVersion"
```

```
// FCM Messaging
implementation
"com.google.firebase:firebase-core:$firebaseCoreVersion"
implementation
"com.google.firebase:firebase-messaging:$firebaseMessagingVer
sion"
}

configurations.all {
    resolutionStrategy.eachDependency { details ->
        def requested = details.requested
        if (requested.group == "com.android.support") {
            if (!requested.name.startsWith("multidex")) {
                details.useVersion "${supportLibVersion}"
            }
        }
    }
}

apply plugin: "com.google.gms.google-services"
```

Nota: as linhas de *'configuration.all'* servem para corrigir alguns conflitos de versão entre as *libs* de *support*.

5.1.2. Configuração do Android Manifest

Com o *gradle* devidamente configurado vamos iniciar a configuração do *AndroidManifest* e posteriormente a criação do principal *Service* responsável por se comunicar com o *Firebase*.

Para isso, adicione as seguintes linhas ao arquivo *AndroidManifest.xml*, dentro da tag *<application>*.

```
<application>
    <service
        android:name=".MyFirebaseMessagingService"
        android:stopWithTask="false">
        <intent-filter>
```

```
<action
android:name="com.google.firebase.MESSAGING_EVENT" />
</intent-filter>
</service>
</application>
```

Perceba que o nome do *Service* irá aparecer em vermelho, clique em qualquer lugar do nome, pressione *alt + enter* e selecione a opção *Create class*. Isso deverá criar uma nova classe pacote principal da aplicação. Antes de iniciar a implementação da classe, vamos organizar nossos pacotes iniciais. Crie um pacote chamado *fcm* e coloque a classe *Service* nele. Crie um outro pacote chamado *view* e coloque a *MainActivity* nele, ficando como na figura 11.

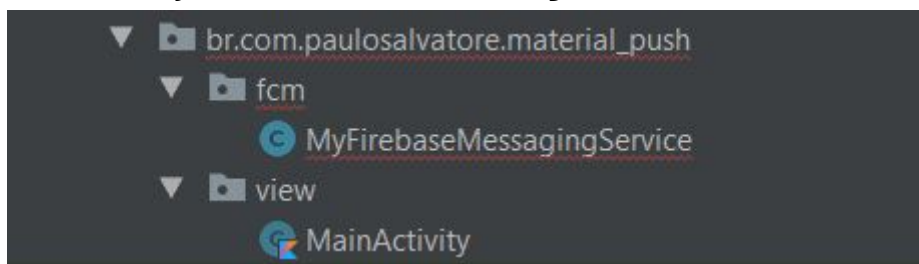


Figura 11: Configuração atual dos pacotes

Fonte: autor - *Android Studio*

5.2. Implementação do FirebaseMessagingService

Antes de implementar a classe, devemos realizar algumas alterações. Perceba que o *Android Studio* automaticamente criou uma classe *Java*. Além de perceptível pelo conteúdo, podemos ver que o próprio ícone das classes são diferentes. Para alterar, basta acessar o menu "Code | Convert Java File to Kotlin File" ou acessar pelo atalho *Ctrl + Alt + Shift + K*.

Perceba que a alteração foi realizada. Agora devemos alterar a interface a ser implementada, em vez de classe padrão *Service*, devemos implementar a classe *FirebaseMessagingService*. Faça a alteração e atualize os *imports* do projeto.

Dica: Você pode usar o atalho *Ctrl + Alt + O* para otimizar os *imports* do projeto automaticamente.

Feito isso, clique em algum lugar da declaração da classe e utilize o atalho *Ctrl + O* e sobrescreva o membro *onNewToken()*, responsável pela geração de novos *tokens*, o *onNewToken()*. Esse membro substitui o então depreciado *onTokenRefresh()* e serve como *callback* para cada vez um novo *token* é gerado

para aquele dispositivo. O conteúdo dessa função basicamente exibirá o *token* no *Logcat* e também irá assinalar a instância do *Firebase* no tópico "MAIN" do *app*:

```
override fun onNewToken(token: String?) {  
    Log.e("NEW_TOKEN", token)  
  
    FirebaseMessaging.getInstance().subscribeToTopic("MAIN")  
}
```

O próximo membro a ser implementado é o *onMessageReceived()*, realize o mesmo processo feito anteriormente para iniciar o *override* e selecione-o na lista. Ele será chamado sempre que uma nova mensagem chegar do *Firebase*, enviando um objeto *RemoteMessage* junto, com os dados informados no *console* do *Firebase* durante a construção de uma nova notificação. Iremos implementá-lo para pegar os valores recebidos e posteriormente gerar uma notificação para eles.

```
private val sTAG = "FMService"  
  
override fun onMessageReceived(remoteMessage: RemoteMessage)  
{  
    val notification = remoteMessage.notification  
  
    Log.d(sTAG, "FCM Message ID: ${remoteMessage.messageId}")  
    Log.d(sTAG, "FCM Data Message: ${remoteMessage.data}")  
    Log.d(sTAG, "FCM Notification Message: $notification")  
  
    if (notification != null) {  
        val title = notification.title ?: ""  
        val body = notification.body ?: ""  
        val dados = remoteMessage.data  
  
        Log.d(sTAG, "FCM Notification Title: $title")  
        Log.d(sTAG, "FCM Notification Body: $body")  
        Log.d(sTAG, "FCM Notification Data: $dados")  
  
        // Criar a notificação  
    }  
}
```

5.3. Criação da classe NotificationCreation

Agora, para criar a notificação, vamos criar um novo pacote chamado *notification* e dentro criaremos a classe *NotificationCreation*. Nessa classe, todos os membros dela terão comportamentos semelhantes ao *static* do *Java*, portanto, inicializaremos eles dentro de um objeto do *Kotlin* chamado *companion*.

```
class NotificationCreation {  
    companion object {  
    }  
}
```

A seguir, declararemos as propriedades padrões para a construção de uma notificação, instância do *NotificationManager*, *ID* da Notificação, padrão de vibração e informações do *channel*. Nosso *companion* ficará assim por enquanto:

```
companion object {  
    private var notificationManager: NotificationManager? =  
    null  
  
    const val sNOTIFY_ID = 1000  
    private val sVIBRATION = longArrayOf(300, 400, 500, 400,  
    300)  
  
    // Channel Information  
    private const val sCHANNEL_ID = "MovileNext_1"  
    private const val sCHANNEL_NAME = "MovileNext - Push  
    Channel 1"  
    private const val sCHANNEL_DESCRIPTION = "MovileNext - Push  
    Channel - Used for main notifications"  
}
```

Com as variáveis definidas, vamos iniciar a construção do membro básico de criação de uma função, o *create()*, que basicamente receberá o *context* que está executando, um título e um corpo para a notificação. Detalhe: tudo isso dentro do *companion*, para que o membro funcione também sem precisar de uma instância.

Desenvolvido por [Paulo Salvatore](#)

```
fun create(context: Context, title: String, body: String) {  
    // Criação da notificação  
}
```

Primeiro, pegaremos o *NotificationManager*, armazenaremos na variável destinada a ele e também iniciaremos a declaração do *channel* para as notificações.

```
if (notificationManager == null)  
    notificationManager =  
    context.getSystemService(Context.NOTIFICATION_SERVICE) as  
    NotificationManager  
  
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
    var channel =  
    notificationManager?.getNotificationChannel(sCHANNEL_ID)  
  
    if (channel == null) {  
        val importance = NotificationManager.IMPORTANCE_HIGH  
  
        channel = NotificationChannel(sCHANNEL_ID,  
sCHANNEL_NAME, importance)  
        channel.description = sCHANNEL_DESCRIPTION  
        channel.enableVibration(true)  
        channel.enableLights(true)  
        channel.vibrationPattern = sVIBRATION  
  
        notificationManager?.createNotificationChannel(channel)  
    }  
}
```

Depois, declararemos a *Intent* e a *PendingIntent* que serão executadas quando o usuário clicar na notificação.

```
val intent = Intent(context, MainActivity::class.java)  
intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TOP or  
Intent.FLAG_ACTIVITY_SINGLE_TOP
```

Desenvolvido por [Paulo Salvatore](#)

```
val pendingIntent = PendingIntent.getActivity(context, 0,
intent, 0)
```

Agora basta usar a classe *NotificationCompat.Builder* para iniciar a construção da notificação, passando todos os valores em sequência.

```
val builder =
    NotificationCompat.Builder(context, sCHANNEL_ID)
        .setContentTitle(title)
        .setSmallIcon(R.drawable.ic_notification)
        .setContentText(body)
        .setDefaults(Notification.DEFAULT_ALL)
        .setAutoCancel(true)
        .setContentIntent(pendingIntent)
        .setTicker(title)
        .setVibrate(sVIBRATION)
        .setOnlyAlertOnce(true)
        .setStyle(NotificationCompat
            .BigTextStyle()
            .bigText(body))
```

Note que no final adicionamos um *setStyle(Big)*, responsável por permitir grandes textos na notificação. Lembre-se de criar um *drawable* para o ícone da notificação, você pode substituir na declaração *setSmallIcon()*, caso queira.

Para finalizar a exibir a notificação, basta aplicar o membro *build()* e passá-la para o *NotificationManager* através do *notify()*, informando também a *ID* da notificação.

```
val notificationApp = builder.build()
notificationManager?.notify(sNOTIFY_ID, notificationApp)
```

Nesse momento, o código deve estar assim:

```
import android.app.Notification
import android.app.NotificationChannel
import android.app.NotificationManager
```

```
import android.app.PendingIntent
import android.content.Context
import android.content.Intent
import android.os.Build
import android.support.v4.app.NotificationCompat
import br.com.paulosalvatore.push.R
import br.com.paulosalvatore.push.view.MainActivity

class NotificationCreation {
    companion object {
        private var notificationManager: NotificationManager? =
null

        const val sNOTIFY_ID = 1000
        private val sVIBRATION = longArrayOf(300, 400, 500, 400,
300)

        // Channel Information
        private const val sCHANNEL_ID = "MovileNext_1"
        private const val sCHANNEL_NAME = "MovileNext - Push
Channel 1"
        private const val sCHANNEL_DESCRIPTION = "MovileNext -
Push Channel - Used for main notifications"

        fun create(context: Context, title: String, body:
String) {
            if (notificationManager == null)
                notificationManager =
context.getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager

            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
                var channel =
notificationManager?.getNotificationChannel(sCHANNEL_ID)

                if (channel == null) {
                    val importance =
NotificationManager.IMPORTANCE_HIGH
```

```
        channel = NotificationChannel(sCHANNEL_ID,
sCHANNEL_NAME, importance)
        channel.description = sCHANNEL_DESCRIPTION
        channel.enableVibration(true)
        channel.enableLights(true)
        channel.vibrationPattern = sVIBRATION

notificationManager?.createNotificationChannel(channel)
    }
}

    val intent = Intent(context,
MainActivity::class.java)
    intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TOP or
Intent.FLAG_ACTIVITY_SINGLE_TOP

    val pendingIntent =
PendingIntent.getActivity(context, 0, intent, 0)

    val builder =
        NotificationCompat.Builder(context,
sCHANNEL_ID)

        .setContentTitle(title)
        .setSmallIcon(R.drawable.ic_notification)
        .setContentText(body)
        .setDefaults(Notification.DEFAULT_ALL)
        .setAutoCancel(true)
        .setContentIntent(pendingIntent)
        .setTicker(title)
        .setVibrate(sVIBRATION)
        .setOnlyAlertOnce(true)
        .setStyle(NotificationCompat
            .BigTextStyle()
            .bigText(body))

    val notificationApp = builder.build()
```

Desenvolvido por [Paulo Salvatore](#)

```
notificationManager?.notify(sNOTIFY_ID,  
notificationApp)  
}  
}  
}
```

Para poder testar se está funcionando, volte para o arquivo que contém o *Service* do *Firebase* e procure pela linha '// Criar a notificação'. Basta inserir a chamada do membro que acabamos declarar.

```
// Criar a notificação  
NotificationCreation.create(this, title, body)
```

Pronto! Agora podemos testar, volte ao site do *Firebase* e clique em 'Enviar sua primeira mensagem'.

No *Firebase Notifications*, apenas o texto e o destino da mensagem são obrigatórios, preencha as opções que desejar (clique nas opções avançadas para explorar) e clique em 'Enviar mensagem'. Veja o resultado na figura 12.

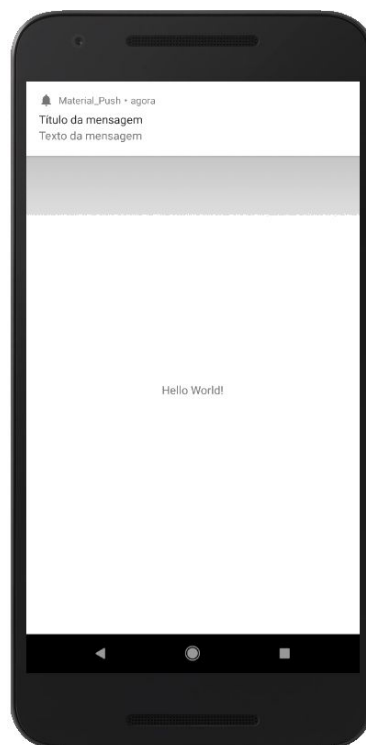


Figura 12: Notificação recebida através do *FirebaseMessagingService*

Com isso temos a base do envio de notificações com o *FMS*, em breve abordaremos mais alguns aspectos interessantes que podem ser incorporados às notificações, como *actions* e imagens.

6. Image Loading

Trabalhar com o carregamento de imagens no *Android* a princípio parece uma tarefa bem simples, afinal, é apenas pegar uma imagem e exibí-la, o que poderia dar errado nisso? Geralmente é o que todos pensamos de início. A grande e dolorida verdade é que carregar imagens não é tão simples assim, independente se a imagem está armazenada na memória do dispositivo ou em algum site da *web*.

6.1. Problemas comuns e possíveis soluções

O principal fator é o tamanho da imagem, afinal, temos dispositivos hoje com câmeras cada vez mais poderosas que tiram fotos mais resoluções altíssimas (mesmo a tela do celular não reproduza a quantidade de pixels), onde o tamanho do arquivo é tão alto quanto. Como o dispositivo precisa armazenar essa informação na memória do celular (que é muito escassa), muita vezes acaba resultando em uma *exception* conhecida como *OutOfMemory*.

Outro fator além do tamanho é quando a imagem está em um servidor da *web*, onde a comunicação dos dados deve ser feita via rede que em muitos casos é extremamente lenta e pode apresentar interrupções e instabilidade.

Somando tudo isso temos um fator inconsistente tornando o carregamento de imagem uma tarefa complicada e demorada (em termos de execução). É por isso que o carregamento de imagens deve ser feito em processos separados, visando não impactar diretamente o processo principal da aplicação conhecido como *Main Thread* ou *UI Thread*.

Carregar grandes *bitmaps* de forma eficiente virou tópico de discussão na comunidade *Android* [[ref](#) 12] e pode ser feito basicamente prevendo o tamanho da imagem antes de carregá-la e associando com o tamanho da *imageView* que receberá a informação, fazendo com que a imagem seja carregada em um tamanho menor, acelerando o processo e diminuindo o consumo de recursos, atrelando também diferentes tipos de configuração de *bitmap* (como *ARGB_8888*, por exemplo) que alteram diretamente o consumo de *bytes* por *pixel* carregado.

6.2. Bibliotecas para carregamento de imagens

Uma outra maneira (a mais utilizada) é basicamente externalizar a resolução do problema para uma biblioteca capaz de realizar tal ação. Atualmente existem três principais bibliotecas que realizam essa função: *Glide*, *Picasso* e *Fresco*. Se mais tiver interesse em saber como essas bibliotecas funcionam dê uma olhada na [referência](#) 13.

6.2.1. Qual delas usar? Qual é a melhor?

Aquela dúvida sempre surge quando estamos decidindo qual biblioteca usar para uma mesma finalidade. A boa notícia é que essa dúvida apareceu para mais desenvolvedores ao longo do processo, alguns preparando bons artigos buscando responder essa questão: afinal, qual a melhor biblioteca para usar no meu projeto?

Dois artigos em especial apresentam pontos bem interessantes comparando algumas bibliotecas. O primeiro artigo [\[ref 14\]](#) faz uma comparação entre *Picasso*, *Glide* e *Fresco*, comparando tanto em termos de sintaxe, funcionalidade padrão (carregamento de imagem) e funcionalidade extras. Já o segundo artigo [\[ref 15\]](#) faz uma comparação apenas entre *Glide* e *Picasso*, trazendo aspectos técnicos muito mais apurados mostrando resultados bem interessantes e visuais dos testes feitos para cada biblioteca. Vale ressaltar que os artigos com o tempo ficam desatualizados pois as bibliotecas pois conforme o tempo passa elas estão em constante atualização e mudanças, apesar das versões testadas ainda estarem disponíveis para uso.

Independente do teste, a resposta que tanta buscamos por enquanto continuará a mesma: a melhor biblioteca a ser utilizada depende do projeto; as considerações a serem levadas: quantidade de imagens carregadas ao mesmo tempo; qualidade desejada para a imagem final; velocidade desejada para carregamento da imagem; tamanho impactado no *APK*; *features* exclusivas; entre diversos outros fatores.

Antes de implementarmos as bibliotecas, quando estamos utilizando a *internet* em uma aplicação *Android* é necessário sinalizar esse uso no manifesto da aplicação:

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

6.2.2. Glide implementation

A biblioteca de carregamento de imagens *Glide* foi desenvolvida e é mantida pela *bumptech* e atualmente encontra na sua versão 4.x [\[ref 16\]](#). Atualmente ela é a recomendada pelo *Google* na própria documentação do *Android* [\[ref 12\]](#).

Para implementar a solução do *Glide* no projeto primeiro precisamos importar a biblioteca no *build.gradle* do *app*:

```
implementation "com.github.bumptech.glide:glide:4.7.1"
annotationProcessor
"com.github.bumptech.glide:compiler:4.7.1"
```

Depois, basta utilizar o seguinte trecho de código para carregar uma imagem a partir de uma *URL* e atribuí-la direto para uma *ImageView*.

```
Glide.with(this@MainActivity)
    .load(url)
    .into(imageView)
```

6.2.3. Picasso implementation

A biblioteca de carregamento de imagens *Picasso*, desenvolvida e mantida pela *square* (empresa bem conhecida no meio *Android* por desenvolver grandes ferramentas) e atualmente se encontra na versão 2.x [\[ref 17\]](#).

Para implementar a solução do *Picasso* no projeto primeiro precisamos importar a biblioteca no *build.gradle* do *app*:

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

Depois, basta utilizar o seguinte trecho de código para carregar uma imagem a partir de uma *URL* e atribuí-la direto para uma *ImageView*.

```
Picasso.get()
    .load(url)
    .into(imageView)
```


6.3. Carregar imagem da galeria ou tirando uma nova

Além de carregar imagens de *URLs* (o que fica muito tranquilo com as bibliotecas) em alguns casos também queremos carregar imagens armazenadas no dispositivo. Para isso, existem algumas *Intents* pré-definidas que buscam arquivos de imagem na galeria, nos documentos ou até mesmo solicitam para tirar uma nova foto.

Para iniciar a solicitação de acessar conteúdo nos documentos podemos usar uma *Intent* com uma ação *ACTION_GET_CONTENT*:

```
// Pick Intent
val pickIntent = Intent(Intent.ACTION_GET_CONTENT)
pickIntent.type = "image/*"
```

Já uma intenção que permite o usuário tirar uma foto nova deve ser solicitada com a ação *ACTION_IMAGE_CAPTURE*:

```
// Take Picture Intent
val takePictureIntent =
Intent(MediaStore.ACTION_IMAGE_CAPTURE)
```

Com a solicitação da câmera, precisamos declarar o uso desse *hardware* no manifesto da aplicação:

```
<uses-feature
    android:name="android.hardware.camera"
    android:required="true" />
```

E por último, uma intenção de acessar conteúdos direto na galeria é uma *Intent* com uma ação *ACTION_PICK*, passando também a *Uri* correspondente à galeria:

```
// Pick Gallery Image
val pickGalleryImageIntent = Intent(Intent.ACTION_PICK,
    MediaStore.Images.Media.EXTERNAL_CONTENT_URI)
```

Desenvolvido por [Paulo Salvatore](#)

Para exibir o menu de opções de *Intents* devemos usar uma *ChooserIntent*, unindo todas as informações declaradas:

```
// Choose Intent
val pickTitle = "Select or take a new Picture"
val chooserIntent = Intent.createChooser(pickIntent,
pickTitle)

chooserIntent.putExtra(
    Intent.EXTRA_INITIAL_INTENTS,
    arrayOf(takePictureIntent, pickGalleryImageIntent)
)
```

Por último, checamos se o dispositivo possui aplicativos capazes de executar as ações desejadas e iniciamos a seleção de uma imagem:

```
// Send Intent
if (chooserIntent.resolveActivity(packageManager) != null) {
    startActivityForResult(chooserIntent, SELECT_PICTURE)
}
```

Colocando todos esses métodos dentro de uma função devemos ter um membro pronto para pegar uma imagem no aparelho a partir da escolha do usuário, ficando assim:

```
fun pickImage() {
    // Pick Intent
    val pickIntent = Intent(Intent.ACTION_GET_CONTENT)
    pickIntent.type = "image/*"

    // Take Picture Intent
    val takePictureIntent =
Intent(MediaStore.ACTION_IMAGE_CAPTURE)

    // Pick Gallery Image
    val pickGalleryImageIntent = Intent(Intent.ACTION_PICK,
        MediaStore.Images.Media.EXTERNAL_CONTENT_URI)
```

```
// Choose Intent
val pickTitle = "Select or take a new Picture"
val chooserIntent = Intent.createChooser(pickIntent,
pickTitle)

chooserIntent.putExtra(
    Intent.EXTRA_INITIAL_INTENTS,
    arrayOf(takePictureIntent, pickGalleryImageIntent)
)

// Send Intent
if (chooserIntent.resolveActivity(packageManager) != null)
{
    startActivityForResult(chooserIntent, SELECT_PICTURE)
}
}
```

Com isso, falta apenas declarar o membro responsável por receber o resultado da seleção da imagem, portanto, sobrescreva o *onActivityResult()* e insira o seguinte código:

```
override fun onActivityResult(requestCode: Int,
                                resultCode: Int,
                                data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)

    if (resultCode == RESULT_OK) {
        if (requestCode == SELECT_PICTURE) {
            if (data?.extras != null) {
                imageView.setImageBitmap(
                    data.extras.get("data") as Bitmap
                )
            } else if (data?.data != null) {
                Picasso.get()
                    .load(data.data)
                    .into(imageView)
            }
        }
    }
}
```

```
    }  
    }  
    }  
}
```

Com isso devemos ter o resultado esperado, solicitamos que o usuário selecione uma imagem armazenada em seu dispositivo, acessamos a imagem selecionada pelos dados extras recebidos da *Intent* e carregamos em uma *ImageView*.

Algo que vale a pena ficar atento é a qualidade da foto tirada. Por padrão, o *Android* irá fornecer apenas um *thumbnail* da foto tirada, caso queiramos a foto em tamanho real, a implementação é um pouquinho mais complicada [ref 19], pois devemos utilizar os *Providers* para acessar os diretórios disponíveis, criar um novo arquivo disponível para foto e passar isso como conteúdo extra da *Intent*. Isso pode ser feito da seguinte maneira.

Dentro do método *pickImage()*, logo após a *takePictureIntent*, adicione o trecho abaixo, responsável por solicitar e preparar a *Uri* para armazenamento da imagem.

```
// Best Quality Picture  
try {  
    val photoFile = createImageFile()  
  
    val photoURI = FileProvider.getUriForFile(  
        this,  
        applicationContext.packageName + ".provider",  
        photoFile)  
  
    takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,  
        photoURI)  
} catch (ex: IOException) {  
    Log.e("IMAGE_LOADING", "Error occurred while creating the  
        File.")  
}
```

Com isso, precisamos preparar uma outra função:

Desenvolvido por [Paulo Salvatore](#)

```
val SELECT_PICTURE = 1
var photoPath: String = ""

@Throws(IOException::class)
private fun createImageFile(): File {
    val timeStamp = SimpleDateFormat("yyyyMMdd_HHmmss",
    Locale.getDefault())
    val imageFileName = "JPEG_" + timeStamp + "_"
    val storageDir =
    getExternalFilesDir(Environment.DIRECTORY_PICTURES)
    val image = File.createTempFile(
        imageFileName,
        ".jpg",
        storageDir
    )

    photoPath = image.absolutePath
    return image
}
```

Além disso, adicione uma nova condição ao membro *onActivityResult()*, logo após o primeiro *else if ()*:

```
else if (photoPath.isNotEmpty()) {
    val photoFile = File(photoPath)

    Picasso.get()
        .load(photoFile)
        .into(imageView)

    photoPath = ""
}
```

No final, o código da *Activity* deverá estar assim:

```
package
br.com.paulosalvatore.pushnotification_images_kotlin.view
```

```
import android.content.Intent
import android.graphics.Bitmap
import android.os.Bundle
import android.os.Environment
import android.provider.MediaStore
import android.support.v4.content.FileProvider
import android.support.v7.app.AppCompatActivity
import android.util.Log
import br.com.paulosalvatore.pushnotification_images_kotlin.R
import com.squareup.picasso.Picasso
import kotlinx.android.synthetic.main.activity_push.*
import java.io.File
import java.io.IOException
import java.text.SimpleDateFormat
import java.util.*

class MainActivity :
    AppCompatActivity() {

    val SELECT_PICTURE = 1
    var photoPath: String = ""

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

        pickImage()
    }

    @Throws(IOException::class)
    private fun createImageFile(): File {
        val timeStamp = SimpleDateFormat("yyyyMMdd_HHmmss",
Locale.getDefault())
        val imageFileName = "JPEG_" + timeStamp + "_"
        val storageDir =
getExternalFilesDir(Environment.DIRECTORY_PICTURES)
        val image = File.createTempFile(
```

```
        imageFileName,
        ".jpg",
        storageDir
    )

    photoPath = image.absolutePath
    return image
}

fun pickImage() {
    // Pick Intent
    val pickIntent = Intent(Intent.ACTION_GET_CONTENT)
    pickIntent.type = "image/*"

    // Take Picture Intent
    val takePictureIntent =
Intent(MediaStore.ACTION_IMAGE_CAPTURE)

    // Best Quality Picture
    try {
        val photoFile = createImageFile()

        val photoURI = FileProvider.getUriForFile(
            this,
            applicationContext.packageName + ".provider",
            photoFile)

        takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
photoURI)
    } catch (ex: IOException) {
        Log.e("IMAGE_LOADING", "Error occurred while creating
the File.")
    }

    // Pick Gallery Image
    val pickGalleryImageIntent = Intent(Intent.ACTION_PICK,
        MediaStore.Images.Media.EXTERNAL_CONTENT_URI)
```

```
// Choose Intent
val pickTitle = "Select or take a new Picture"
val chooserIntent = Intent.createChooser(pickIntent,
pickTitle)

chooserIntent.putExtra(
    Intent.EXTRA_INITIAL_INTENTS,
    arrayOf(takePictureIntent, pickGalleryImageIntent)
)

// Send Intent
if (chooserIntent.resolveActivity(packageManager) !=
null) {
    startActivityForResult(chooserIntent, SELECT_PICTURE)
}

override fun onActivityResult(requestCode: Int,
                                resultCode: Int,
                                data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)

    if (resultCode == RESULT_OK) {
        if (requestCode == SELECT_PICTURE) {
            if (data?.extras != null) {
                imageView.setImageBitmap(
                    data.extras.get("data") as Bitmap
                )
            } else if (data?.data != null) {
                Picasso.get()
                    .load(data.data)
                    .into(imageView)
            } else if (photoPath.isNotEmpty()) {
                val photoFile = File(photoPath)

                Picasso.get()
                    .load(photoFile)
                    .into(imageView)
            }
        }
    }
}
```



```
        photoPath = ""
    }
}
}
```

Por último, precisamos declarar algumas configurações no manifesto e em alguns recursos *xml*:

```
<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"
    android:maxSdkVersion="18" />

<application>
    <!--...-->
    <provider
        android:name="android.support.v4.content.FileProvider"
        android:authorities="${applicationId}.provider"
        android:exported="false"
        android:grantUriPermissions="true">
        <meta-data
            android:name="android.support.FILE_PROVIDER_PATHS"
            android:resource="@xml/file_paths"></meta-data>
        </provider>
    </application>
```

Crie um novo *resource directory* chamado *xml*, e crie um novo arquivo *xml* chamado 'file_paths.xml' e insira o seguinte conteúdo:

Nota: lembre-se de mudar o conteúdo grifado pelo nome do seu pacote.

```
<?xml version="1.0" encoding="utf-8"?>
<paths
    xmlns:android="http://schemas.android.com/apk/res/android">
    <external-path name="my_images"
```

```
path="Android/data/br.com.paulosalvatore.pushnotification_images_kotlin/files/Pictures" />
</paths>
```

Ufa! Depois de todas essas implementações é possível pegar a foto tirada na resolução real. Mais alguns detalhes sobre essa implementação é possível verificar na [referência](#) 19.

7. Networking

Networking no *Android* basicamente é a capacidade de se comunicar com algum serviço disponível na *web* que fornecerá algum dado em algum formato entre os vários existentes. Assim como o carregamento de imagens da *web* recebe uma quantidade de dados estruturados correspondentes aos dados da imagem, o carregamento de informações puramente estruturadas em texto também pode ter as mesmas complicações, afinal, estamos em um protocolo onde não temos controle e que pode se tornar uma operação longa e que até mesmo não complete, seja por queda de sinal, muito tempo demorado, falta de recurso para processar a execução e diversos outros motivos.

Um detalhamento mais completo dessa abordagem no *Android* está disponível na própria documentação [[ref](#) 20], com assuntos de altíssimo nível como: maneiras de utilizar essa abordagem economizando bateria do dispositivo; utilizando abordagens diferentes para transferência de dados; entre diversos outros.

Neste capítulo entenderemos a principal forma de comunicação pela *web* hoje que é pelo protocolo *REST* utilizando a estrutura de dados *JSON*. Veremos também os principais meios de transformar texto *JSON* em objetos com informações estruturadas e prontas para serem usadas, além de ver quais bibliotecas temos disponíveis para melhorar tais resultados.

Vale ressaltar que toda requisição na *web* deve ser feita em um processo separado, a fim de evitar erros que podem travar a aplicação.

7.1. REST

Um termo muito falado entre os desenvolvedores é o *REST*. Apesar das várias explicações complexas e cheias de termos técnicos sobre esse termo eu

gosto de deixar tudo isso de lado e simplificar que *REST* é apenas a maneira mais elegante atualmente de solicitar informações que estão disponíveis na *web*.

Para acessar essas informações utilizamos *web services* (que também irei referir como *API*) que são basicamente aplicações *RESTful*, ou seja, que estão aptas a se comunicar utilizando a diretrizes do *REST*. Isso significa que o *web service* irá falar a língua do *REST*, que é toda construída em cima do protocolo *HTTP* e composta por alguns verbos principais: *GET*, *POST*, *PUT* e *DELETE* e alguns outros um pouco mais específicos.

Quando enviamos uma requisição de solicitação, estamos esperando receber alguma informação em troca, seja os dados de cadastro de algum usuário ou até mesmo o sinal de que o novo usuário foi cadastrado com sucesso. Sendo assim, essa informação precisa estar estruturada de alguma maneira dentre as disponíveis.

7.2. JSON

As duas formas mais comuns de estruturação de dados são *JSON* e *XML*, onde hoje em dia é extremamente mais comum as informações estarem estruturadas em *JSON*.

O *JSON* (*Java Script Object Notation*) é uma forma de estruturação de dados muito comum para trocar informações entre diversas aplicações, independente da linguagem de programação em que ela é feita. Sendo inspirada na declaração de objetos da linguagem *JavaScript*, é considerada uma das formas mais leves de estruturar uma informação, levando grande vantagem em relação ao *XML* pois no *XML* cada *tag* que foi aberta deve ser fechada repetindo o nome usado na abertura, gastando muito mais *bytes* apenas para abrir e fechar conteúdos, o que não acontece no *JSON*, já que a estruturação é organizada utilizando símbolos como chaves e colchetes, como por exemplo:

```
{
  "colors": [
    {
      "color": "red",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [
          255,
```

```
        0,  
        0,  
        1  
    ],  
    "hex": "#FF0"  
  }  
},  
{  
  "color": "white",  
  "category": "value",  
  "code": {  
    "rgba": [  
      0,  
      0,  
      0,  
      1  
    ],  
    "hex": "#FFF"  
  }  
},  
{  
  "color": "green",  
  "category": "hue",  
  "type": "secondary",  
  "code": {  
    "rgba": [  
      0,  
      255,  
      0,  
      1  
    ],  
    "hex": "#0F0"  
  }  
}  
]  
}
```

7.2.1. JSON Visualizer

Muitas vezes os dados em *JSON* estão otimizados, sem os espaços e as quebras de linha que facilitam a visualização das informações por humanos. Para isso, existem algumas ferramentas disponíveis para melhorar essa visualização nos facilitando o trabalho de entendimento da estrutura da informação.

A primeira é uma versão online mantida pelo *Code Beautify* chamada *JSON Viewer*, disponível em <https://codebeautify.org/jsonviewer> e que nos permite customizar a organização de várias maneiras. Ela é muito útil para exibir trechos rápidos de *JSON* que estão minimizados e permite navegar entre o conteúdo de uma maneira interessante.

A segunda é uma extensão do *Chrome* também chamada *JSON Viewer*, que basicamente entende qualquer página na *web* que contenha conteúdo *JSON* e automaticamente faz com que o navegador estruture os dados presentes de uma maneira navegável e mais organizada, exibindo o número de cada linha e permitindo expandir ou ocultar diferentes níveis da hierarquia.

O projeto e o link para instalação do *JSON Viewer* estão disponíveis em <https://github.com/tulios/json-viewer>.

7.3. Serialization

Um outro capítulo bem importante quando o assunto é *networking* e transmissão de dados é a *serialization*, ou simplesmente serialização. Os dados recebidos chegam em um *feed* de texto puro mas que estão estruturados da maneira que esperamos, com isso, essa técnica consiste em pegar essa informação e passá-la por um interpretador que irá transformar o texto em um objeto nativo da linguagem da programação.

Dependendo da linguagem de programação, esse processo pode ser mais simples ou mais complicado. Como nas linguagens presentes no ambiente *Android* temos algumas características mais robustas de tipo de variáveis e estruturação de conteúdo, devemos utilizar esse interpretador avisando exatamente qual estrutura de dados que estamos esperando, e não apenas a técnica de estruturação.

Para isso, tanto no *Java* quanto no *Kotlin*, criamos classes que estão estruturadas da maneira exata que aquilo é feito na *API*. Dessa maneira, o interpretador sabe exatamente como estruturar os dados recebidos e inclusive se alguma coisa não está de acordo com o esperado, acusando alguns erros ou tratando de acordo com o que foi programado.

A implementação mais básica no *Android* para transformar *JSON* em objeto da linguagem é utilizando a classe *JSONParser*, por exemplo:

```
val reader = JSONObject(jsonData)
val main = reader.getJSONObject("main")
val temperature = main.getString("temp")
```

7.4. Bibliotecas existentes: Networking

Assim como no carregamento de imagens, a obtenção e transmissão também é um assunto no *Android* muito discutido pelos desenvolvedores e com diversas soluções disponíveis através de bibliotecas. As principais bibliotecas de comunicação com o protocolo *HTTP* para obtenção de dados são: *Retrofit*, *Volley* e *Fast Android Networking*.

7.4.1. Retrofit

A biblioteca *Retrofit* [[ref](#) 21] é desenvolvida e mantida pela *square* e é uma das mais populares entre os desenvolvedores *Android*, estando presente na maioria dos projetos que precisam fazer requisições na *web*. Ela possui um uso extensivo de *annotations* que são capazes de sintetizar rapidamente a informação recebida e auxiliar muito no processo de serialização dos dados, que geralmente é feito por uma biblioteca externa.

A implementação dela é bem simples, inicialmente precisamos importá-la no *build.gradle* do *app*:

```
implementation 'com.squareup.retrofit2:retrofit:2.4.0'
```

Depois, precisamos construir um *Service* responsável pela declaração dos métodos que saberão quais acessos da *API* realizar e quais informações são necessárias, além de saber exatamente o tipo de retorno esperado:

```
interface GitHubService {
    @GET("users/{user}/repos")
    fun listRepos(@Path("user") user: String): Call<List<Repo>>
}
```

Com a interface declarada, iniciamos a declaração do código que irá construir o *Retrofit* e inserir a *URL* desejada:

```
internal var retrofit = Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build()

internal var service =
retrofit.create(GitHubService::class.java)
```

O *Retrofit* irá permitir uma série de customizações durante a implementação, veremos mais alguns detalhes sobre ela durante o nosso exemplo prático de utilização da biblioteca.

7.4.2. Volley

A biblioteca *Volley* [[ref](#) 22] já é bem antiga no universo *Android* e é desenvolvida e mantida pelo *Google*. Um diferencial dessa biblioteca é que ela possui suporte não apenas ao carregamento de dados como também tratamento para imagens e algumas *features* que permitem mais controle em relação a quais ações tomar durante o manuseio do *networking* no *Android*.

Uma sessão de *training* para o *Volley* está disponível na própria documentação do *Android*, acessando pelo link presente na [referência](#) 23.

A implementação dela é bem simples, inicialmente precisamos importá-la no *build.gradle* do *app*:

```
implementation 'com.android.volley:volley:1.1.1'
```

Uma característica do *Volley* é o seu sistema de filas (*queues*), portanto, para iniciar a utilização, precisamos pegar sua instância e depois inserir uma nova *request* na fila de processamento:

```
// Instantiate the RequestQueue.
val queue = Volley.newRequestQueue(this)
val url = "http://www.google.com"

// Request a string response from the provided URL.
val stringRequest = StringRequest(Request.Method.GET, url,
    Response.Listener<String> { response ->
```

```
// Display the first 500 characters of the response
string.
textView.text = "Response is: ${response.substring(0,
500)}"
},
Response.ErrorListener { textView.text = "That didn't
work!" })

// Add the request to the RequestQueue.
queue.add(stringRequest)
```

7.4.3. Considerações adicionais

Uma outra biblioteca que não citarei neste material mas que vale dar uma olhada é a *Fast Android Networking*, com o projeto disponível no *GitHub* [\[ref 24\]](#). Se tiver interessado em conhecer mais sobre essa biblioteca recomendo dar uma olhada no artigo disponível na [referência 25](#).

Um artigo interessante faz a comparação entre *Retrofit* e *Volley*, apresentando diversos pontos entre as duas bibliotecas e também realizando alguns testes de performance. Assim como a comparação que mostrei das bibliotecas de carregamento de imagem, não existe uma resposta concreta de qual biblioteca usar, depende de uma série de fatores a serem considerados. O artigo pode ser visualizado na [referência 26](#).

Antes de começar, gostaria também de fazer a recomendação de um *software* muito útil para trabalhar com *APIs*, o *Postman*, que torna muito fácil o controle de requisições, resultados e diversas outras funcionalidades que aceleram o processo de desenvolvimento, disponível em <https://www.getpostman.com/>.

7.5. Bibliotecas existentes: Serialization

Separado das bibliotecas de *HTTP*, temos uma grande quantidade de bibliotecas para serialização dos dados em *JSON*, onde as principais são: *GSON* (Google), *Jackson* (FasterXML) e *Moshi* (square). Um estudo [\[ref 27\]](#) fez uma comparação entre as bibliotecas e revelou resultados semelhantes entre as três em termos de desempenho, apontando a biblioteca *GSON* como a mais popular atualmente.

Um caso bem interessante envolve as bibliotecas *GSON* e *Moshi*, cujo um dos principais desenvolvedores do *GSON* iniciou o projeto *Moshi* posteriormente.

Uma dúvida levantada no *site Stackoverflow* [ref 28] levantou a questão entre qual das duas bibliotecas é melhor. O desenvolvedor respondeu com alguns aspectos interessantes presentes no *Moshi* que não são possíveis no *Gson* e concluindo referenciando um outro artigo no *site Medium* [ref 29] que apresenta vários pontos para considerarmos a utilização do *Moshi* como principal biblioteca para processar *JSON*, acredito que vale a leitura.

7.6. Exemplo prático

Agora que conhecemos todos os pontos principais sobre *Networking*, vamos fazer uma rápida implementação de um aplicativo que fará requisições a uma *API*, trabalhará com os resultados retornados e exibirá para o usuário. Nesse exemplo utilizaremos o *Retrofit* para acesso à *API* e o *GSON* para desserialização dos dados em formato *JSON*.

Como base para o nosso projeto, utilizaremos o *app* desenvolvido no capítulo de *Kotlin*, que possuía uma *RecyclerView* com uma lista de linguagens de programação. Resumidamente teremos a *RecyclerView* construída no exercício anterior, sendo que cada item possuirá um *listener* que quando ativado irá buscar na *API* do *GitHub* os repositórios que estão escritos na linguagem clicada. Com isso, carregaremos uma nova *RecyclerView* com os resultados da busca. Mãos à obra?

Iniciando o projeto, iremos fazer algumas configurações. Como utilizaremos a *internet*, precisamos sinalizar isso no manifesto da aplicação. Além disso, a fim de melhor a experiência de usuário, também iremos checar se temos conexão ou não e exibir uma alerta caso não tenha, por tanto, mais uma declaração de permissão precisa ser feita:

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

Agora, abra a classe *MainActivity* para iniciar alguns testes de requisição na web. A *API* que utilizaremos é a do *GitHub* e pegaremos o repositório da linguagem *Kotlin* através da url: <https://api.github.com/search/repositories?q=kotlin>. Experimente abrí-la no navegador ou com alguma das maneiras citadas no capítulo [JSON Visualizer](#). Sinta-se à vontade para olhar na documentação da *API* [ref 30] para entendê-la melhor.

Vamos iniciar agora fazendo uma requisição simples para obter todo o *feed* de dados disponível no endereço em questão. Para isso, utilizaremos a classe *URL*, informando a *url* e executando o método *readText()* dentro de uma *thread* separada, que no *Kotlin* com a biblioteca *Anko* iniciaremos apenas digitando *doAsync { }*. Após a requisição concluir, voltaremos para a *mainThread* para exibir que ela foi concluída e também adicionar a *resultString* recebida no *console*:

```
val url =
    "https://api.github.com/search/repositories?q=kotlin"

override fun onCreate(savedInstanceState: Bundle?) {
    //...
    doAsync {
        val resultString = URL(url).readText()
        Log.d("URL_CONTENT", resultString)
        uiThread { longToast("Request performed") }
    }
    //...
}
```

Rode a aplicação e veja o resultado. Caso algum erro ocorra no *console*, tente rotacionar o dispositivo para que o código seja executado novamente e faça uma nova requisição. Em caso de erros na requisição a aplicação não travará mas também não será bem sucedida. O resultado deve ser algo semelhante ao da figura 13.

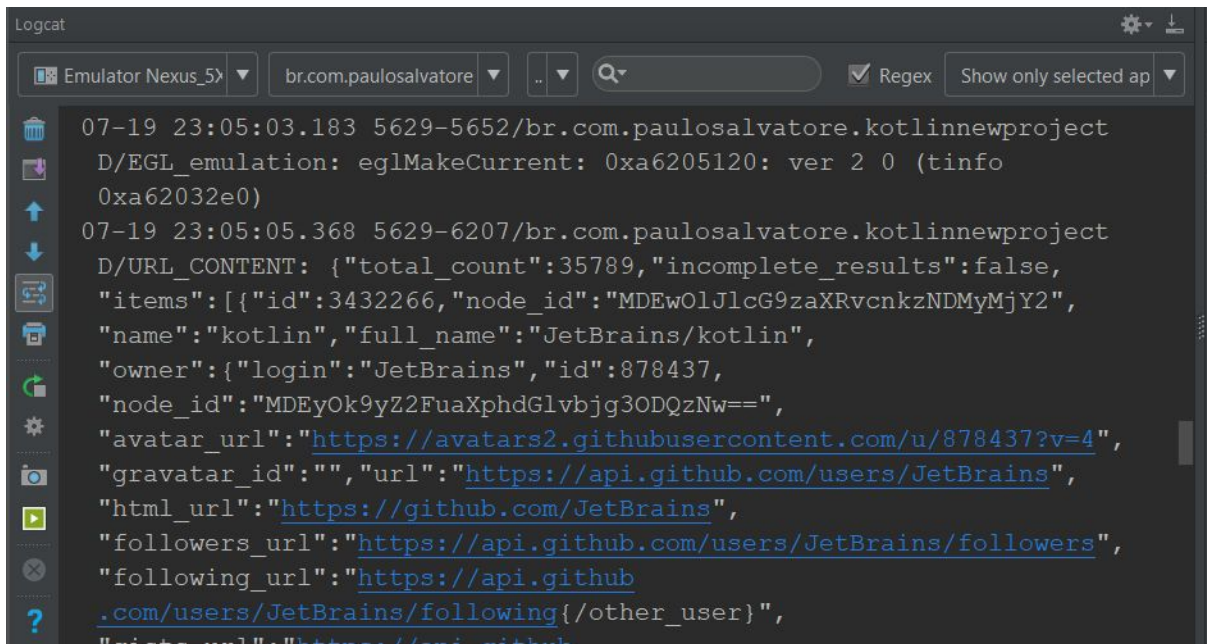


Figura 13: Resultado da requisição no *Logcat*

Fonte: autor - *Android Studio*

Existem algumas maneiras de prosseguir com essa implementação, porém, todas são bem trabalhosas visto que teríamos que tratar diversas condições inesperadas. Por tanto, a partir deste momento utilizaremos o *Retrofit* e o *GSON* para realizar a requisição e formatar os dados corretamente. Para configurá-los, abra o arquivo *build.gradle* do projeto e do *app* e insira os conteúdos respectivamente:

```
buildscript {  
    //...  
    ext.retrofit_version = "2.4.0"  
    //...  
}
```

```
dependencies {  
    //...  
    // Retrofit  
    implementation  
    "com.squareup.retrofit2:retrofit:$retrofit_version"  
    implementation  
    "com.squareup.retrofit2:converter-gson:$retrofit_version"
```

```
}
```

Crie um novo pacote para a aplicação chamado *api* e crie uma nova interface *Kotlin* chamada *GithubService*.

```
interface GithubService {  
}
```

Dentro dessa interface é onde inserimos os membros responsáveis por receber argumentos e construir a *URL* da requisição. No nosso caso, teremos apenas o *searchRepositories()*:

```
interface GithubService {  
    @GET("/search/repositories")  
    fun searchRepositories(  
        @Query("q") query: String,  
        @Query("sort") sort: String = "stars",  
        @Query("order") order: String = "desc"  
    ): Call<GithubRepositoriesResult>  
}
```

Não esqueça de adicionar os *imports* corretamente, no nosso caso deverá ficar assim:

```
import retrofit2.Call  
import retrofit2.http.GET  
import retrofit2.http.Query
```

Dentro do pacote *model*, crie um novo arquivo *Kotlin* chamado *GithubRepositoriesResult*, conforme a figura 14.

Desenvolvido por [Paulo Salvatore](#)

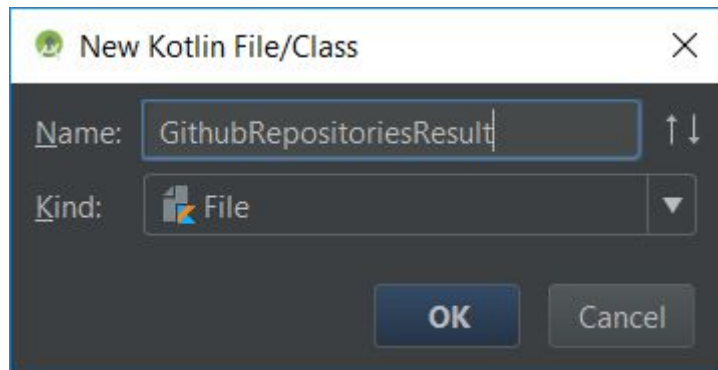


Figura 14: Criação do arquivo *GithubRepositoriesResult*
Fonte: autor - *Android Studio*

Dentro do arquivo, iremos criar as três *data classes* responsáveis pela serialização dos dados. A estrutura das classes deve estar de acordo com os dados esperados da *API*.

```
import com.google.gson.annotations.SerializedName

data class GithubRepositoriesResult(
    @SerializedName(value = "items")
    val repositories: List<Repository>
)

data class Repository(val id: Long?,
    val name: String?,
    val full_name: String?,
    val owner: Owner,
    val private: Boolean,
    val html_url: String?,
    val description: String?)

data class Owner(val login: String?,
    val id: Long?,
    val avatar_url: String?)
```

Vamos criar os arquivos da nossa nova *RecyclerView* dos repositórios, para facilitar esse processo, vamos copiar e colar os arquivos e fazer as devidas alterações. Copie e cole o arquivo do *layout* (*Ctrl + C* e *Ctrl + V*) do item chamado

programming_language_item.xml: na caixa de diálogo que apareceu altere o nome do arquivo para *repository_item.xml*. Dentro do novo arquivo apenas mude a *View* até então de *ID tvLaunchYear* colocando o conteúdo abaixo no lugar. O restante manteremos o mesmo.

```
<TextView
    android:id="@+id/tvOwner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textColor="#AAA"
    android:textSize="12sp"
    tools:text="Owner" />
```

Copie e cole o arquivo do *Adapter* (*Ctrl + C* e *Ctrl + V*): na caixa de diálogo que apareceu altere o nome do arquivo para *RepositoryAdapter* e pressione *enter*. Dê dois cliques no nome *ProgrammingLanguage* em qualquer uma das ocorrências dentro do arquivo, Aperte *Ctrl + R* e digite *Repository*. Dentro de *onCreateViewHolder()* altera o *inflate* do *layout* para o que criamos anteriormente.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): ViewHolder {
    val view =
LayoutInflater.from(context).inflate(R.layout.repository_item
, parent, false)
    return ViewHolder(view)
}
```

Vamos iniciar a configuração do *bindView()* da classe *ViewHolder*, que pegará as informações de cada *Repository* e atualizar no *layout* do item. Por enquanto deixaremos a imagem sem exibição, apenas preencheremos as informações de texto.

```
class ViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
    fun bindView(item: Repository,
                listener: (Repository) -> Unit) =
with(itemView) {
```

```
        tvTitle.text = item.name
        tvOwner.text = item.owner.login
        tvDescription.text = item.description

        setOnClickListener { listener(item) }
    }
}
```

Caso o elemento *tvTitle* dê algum conflito, verifique os *imports*, certificando de que está importando os elementos de *view* do *layout* correto. O conteúdo completo do arquivo deve estar assim:

```
import android.content.Context
import android.support.v7.widget.RecyclerView
import android.support.v7.widget.RecyclerView.Adapter
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import br.com.paulosalvatore.kotlinnewproject.R
import
br.com.paulosalvatore.kotlinnewproject.model.Repository
import kotlinx.android.synthetic.main.repository_item.view.*

class RepositoryAdapter(
    private val items: List<Repository>,
    private val context: Context,
    private val listener: (Repository) -> Unit
) : Adapter<RepositoryAdapter.ViewHolder>() {
    override fun onCreateViewHolder(parent: ViewGroup,
viewType: Int): ViewHolder {
        val view =
LayoutInflater.from(context).inflate(R.layout.repository_item
, parent, false)
        return ViewHolder(view)
    }

    override fun getItemCount(): Int {
```

```
        return items.size
    }

    override fun onBindViewHolder(holder: ViewHolder, position:
Int) {
        val item = items[position]
        holder.bindView(item, listener)
    }

    class ViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        fun bindView(item: Repository,
                    listener: (Repository) -> Unit) =
with(itemView) {
            tvTitle.text = item.name
            tvOwner.text = item.owner.login
            tvDescription.text = item.description

            setOnClickListener { listener(item) }
        }
    }
}
```

Antes de prosseguir, remova ou comente os trechos de códigos inseridos no início deste capítulo na *MainActivity*, tanto a declaração da variável *url* quanto a declaração inteira do *doAsync* {}.

Selecione todo o trecho de código responsável pela ativação da *RecyclerView*, tanto do seu conteúdo quanto do seu *layout* e pressione o atalho *Ctrl + Alt + M*. Na janela que apareceu digite *loadDefaultRecyclerView* e pressione *OK*. Se tiver interessado em mais atalhos de produtividade como esse, recomendo a leitura adicional do [ReferenceCard da IntelliJ IDEA](#), com uma lista completa de funcionalidade presentes em todas as *IDEs* da *Jetbrains*.

Dentro do pacote *api* crie uma nova classe chamada *RepositoryRetriever*, utilizaremos ela para centralizar todas as chamadas que serão feitas através do *Retrofit*. Inicialmente iremos declarar a propriedade *GithubService* que guarda a instância do *service* construído anteriormente, além da *URL* base da *API*:


```
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

class RepositoryRetriever {
    private val service: GithubService

    companion object {
        const val BASE_URL = "https://api.github.com/"
    }

    init {
        val retrofit = Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(
                GsonConverterFactory.create()
            )
            .build()

        service = retrofit.create(GithubService::class.java)
    }
}
```

Com isso a base do *retriever* já está definida, agora podemos declarar todos os *callbacks* que utilizaremos para fazer as requisições na *API* e que serão responsáveis por enfileirá-los no *Retrofit*. Nesse exemplo teremos apenas o membro *getLanguageRepositories()*.

```
fun getLanguageRepositories(callback:
    Callback<GithubRepositoriesResult>,
                           query: String) {
    val call = service.searchRepositories("language:$query")
    call.enqueue(callback)
}
```

Com isso nosso *retriever* está pronto. Prosseguiremos construindo a declaração do *callback* na *MainActivity* que será enviado para o *retriever* e conterá as implementações para o caso de requisição bem ou mal sucedida.

```
private val repositoryRetriever = RepositoryRetriever()

private val callback = object :
    Callback<GithubRepositoriesResult> {
        override fun onFailure(
            call: Call<GithubRepositoriesResult>?,
            t: Throwable?) {

            // Implementação em caso de falha
            longToast("Fail loading repositories.")

            Log.e("MainActivity", "Problem calling Github API", t)
            Log.d("MainActivity", "Fail on URL:
            ${call?.request()?.url()}")
        }

        override fun onResponse(
            call: Call<GithubRepositoriesResult>?,
            response: Response<GithubRepositoriesResult>?) {

            // Implementação em caso de sucesso
            longToast("Load finished.")
        }
    }
}
```

Note que a implementação em caso de falha possui alguns *logs* importantes: a *exception* ocorrida durante a falha e a *URL* onde houve falha na requisição. A importância de exibir a *URL* construída é que você pode usar o *link* que o *Logcat* cria automaticamente para abri-lo para verificar quaisquer problemas eventuais.

Com o *callback* definido, vamos começar a implementação em caso de requisição bem sucedida. Ela consistirá em receber a resposta, acessar o *body()* e a partir disso pegar os objetos que já foram desserializados pelo *GSON*. Lembrando que objetos *nullables* precisam do marcador de *safe call* '?. ' para serem acessados com segurança. Com a lista de repositórios retornadas, passamos para o *RepositoryAdapter* que irá construir a *RecyclerView* e atualizar sua exibição. Logo após o *longToast* que exibe a finalização do *load* insira:

```
response?.isSuccessful.let {
    response?.body()?.repositories?.let {
        val resultList = response.body()?.repositories ?:
emptyList()
        recyclerView.adapter =
            RepositoryAdapter(
                resultList,
                this@MainActivity) {
                    longToast("Clicked item: ${it.full_name}")
                }
    }
}
```

Voltando para a *MainActivity*, dentro do membro *loadDefaultRecyclerView()* precisamos atualizar o *listener* do *Adapter* para realizar a chamada no *RepositoryRetriever* passando os valores que declaramos.

```
recyclerView.adapter =
    ProgrammingLanguageAdapter(
        recyclerViewItems(),
        this) {
        longToast("Loading ${it.title} repositories...")
        repositoryRetriever.getLanguageRepositories(
            callback,
            it.title
        )
    }
}
```

Vamos testar! Rode o projeto e veja o resultado parcial. O fluxo de funcionamento esperado da aplicação pode ser visualizado na figura 15.

Desenvolvido por [Paulo Salvatore](#)

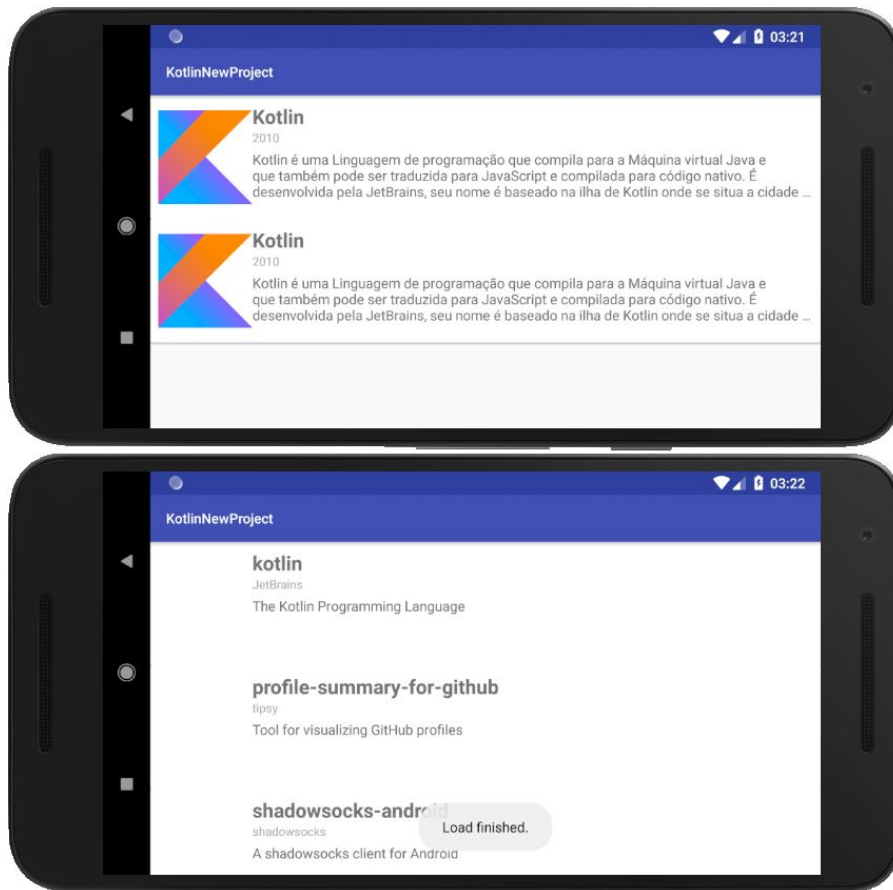


Figura 15: Fluxo atual da aplicação
Fonte: autor - *Android Emulator (API 27)*

Vamos agora realizar alguns aprimoramentos no nosso *app*: adicionar um botão para voltar para a *RecyclerView* inicial; carregar algumas imagens na *RecyclerView* dos *Repositories*; e adicionar um tratamento para quando o celular está sem conexão.

Para adicionar o botão vamos para o *layout* da *MainActivity*. Utilizaremos a janela de *Design* para criá-lo. Na aba *Pallete*, selecione a categoria *Common* ou *Buttons*, pegue o *Button* padrão e arraste-o para a aba *Component Tree*, soltando-o logo abaixo da *RecyclerView*, porém, dentro do *ConstraintLayout*. Atenção para não colocá-lo dentro da *RecyclerView*. Faça a ligação do *constraint left* e *right* com as laterais do *layout* e o *constraint bottom* com a parte de baixo do *layout*. Mude o texto do botão para '*Reload RecyclerView*', o atributo *layout_width* para *match_parent* e adicione a *ID* '*btReload*'. A visualização do *layout* deverá ficar como na figura 16.



Figura 16: Visualização do *layout* da *MainActivity*
Fonte: autor - *Android Studio*

Com o botão construído, vamos adicionar um *listener* para que ele funcione devidamente, chamando o membro de carregamento da *RecyclerView* que já está pronto. Faça a declaração dentro do *onCreate()* da classe *MainActivity*.

```
btReload.setOnClickListener { _ ->
    loadDefaultRecyclerView()
}
```

Crie um novo membro chamado *isNetworkConnected()* que retornará um *Boolean*.

```
private fun isNetworkConnected(): Boolean {
```

```
val connectivityManager =  
getSystemService(Context.CONNECTIVITY_SERVICE) as  
ConnectivityManager  
val networkInfo = connectivityManager.activeNetworkInfo  
return networkInfo != null && networkInfo.isConnected  
}
```

Agora dentro do membro *onCreate()*, em vez de chamar direto o carregamento da *RecyclerView*, iremos encapsulá-lo com uma checagem que chama o membro que acabamos de declarar. Caso o serviço de conexão esteja indisponível iremos exibir um alerta para o usuário informando uma mensagem personalizada.

```
if (isNetworkConnected()) {  
    loadDefaultRecyclerView()  
} else {  
    alert ("Please check your internet connection and try  
again.",  
        "No internet connection") {  
        this.iconResource = android.R.drawable.ic_dialog_alert  
        yesButton { }  
    }.show()  
}
```

O código da *MainActivity* a essa altura deve se parecer com isso:

```
import android.content.Context  
import android.net.ConnectivityManager  
import android.os.Bundle  
import android.support.v7.app.AppCompatActivity  
import android.support.v7.widget.LinearLayoutManager  
import android.util.Log  
import br.com.paulosalvatore.kotlinnewproject.R  
import  
br.com.paulosalvatore.kotlinnewproject.adapter.ProgrammingLanguageAdapter
```

```
import
br.com.paulosalvatore.kotlinnewproject.adapter.RepositoryAdap
ter
import
br.com.paulosalvatore.kotlinnewproject.api.RepositoryRetrieve
r
import
br.com.paulosalvatore.kotlinnewproject.model.GithubRepositori
esResult
import
br.com.paulosalvatore.kotlinnewproject.model.ProgrammingLangu
age
import kotlinx.android.synthetic.main.activity_main.*
import org.jetbrains.anko.alert
import org.jetbrains.anko.longToast
import org.jetbrains.anko.yesButton
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Response

class MainActivity : AppCompatActivity() {
    private val repositoryRetriever = RepositoryRetriever()

    private val callback = object :
Callback<GithubRepositoriesResult> {
        override fun onFailure(call:
Call<GithubRepositoriesResult>?,
                                t: Throwable?) {
            // Implementação em caso de falha
            longToast("Fail loading repositories.")

            Log.e("MainActivity", "Problem calling Github API",
t)

            Log.d("MainActivity", "Fail on URL:
${call?.request()?.url()}")

        }
    }
}
```

```
        override fun onResponse(call:
Call<GithubRepositoriesResult>?,
                                response:
Response<GithubRepositoriesResult>?) {
            // Implementação em caso de sucesso
            longToast("Load finished.")

            response?.isSuccessful.let {
                response?.body()?.repositories?.let {
                    val resultList = response.body()?.repositories
?: emptyList()
                    recyclerView.adapter =
                        RepositoryAdapter(
                            resultList,
                            this@MainActivity) {
                                longToast("Clicked item:
${it.full_name}")
                            }
                        }
                }
            }
        }

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

            if (isNetworkConnected()) {
                loadDefaultRecyclerView()
            } else {
                alert ("Please check your internet connection and try
again.",
                    "No internet connection") {
                    this.iconResource =
android.R.drawable.ic_dialog_alert
                    yesButton { }
                }.show()
            }
        }
    }
}
```



```
        btReload.setOnClickListener { _ ->
            loadDefaultRecyclerView()
        }
    }

    private fun isConnected(): Boolean {
        val connectivityManager =
            getSystemService(Context.CONNECTIVITY_SERVICE) as
            ConnectivityManager
        val networkInfo = connectivityManager.activeNetworkInfo
        return networkInfo != null && networkInfo.isConnected
    }

    private fun loadDefaultRecyclerView() {
        // Lista na Vertical
        val layoutManager = LinearLayoutManager(this)
        recyclerView.layoutManager = layoutManager

        recyclerView.adapter =
            ProgrammingLanguageAdapter(
                recyclerViewItems(),
                this) {
                    longToast("Loading ${it.title}
repositories...")
                    repositoryRetriever.getLanguageRepositories(
                        callback,
                        it.title
                    )
                }
    }

    private fun recyclerViewItems(): List<ProgrammingLanguage>
    {

        val kotlin = ProgrammingLanguage(R.drawable.kotlin,
            "Kotlin",
            2010,
```

```
"Kotlin é uma Linguagem de programação que compila para a Máquina virtual Java e que também pode ser traduzida para JavaScript e compilada para código nativo. É desenvolvida pela JetBrains, seu nome é baseado na ilha de Kotlin onde se situa a cidade russa de Kronstadt, próximo à São Petersburgo. Apesar de a sintaxe de Kotlin diferir da de Java, Kotlin é projetada para ter uma interoperabilidade total com código Java. Foi considerada pelo público a 2ª linguagem 'mais amada', de acordo com uma pesquisa conduzida pelo site Stack Overflow em 2018.")
```

```
    return listOf(kotlin, kotlin)
}
```

Para testar essa funcionalidade experimente colocar o dispositivo em modo avião e forçar a *Activity* a ser recriada. Deverá aparecer a caixa de alerta conforme mostrado na figura 17. Fique a vontade para adicionar os textos no arquivo *resources* de *strings*, para facilitar posteriormente a internacionalização do *app*.

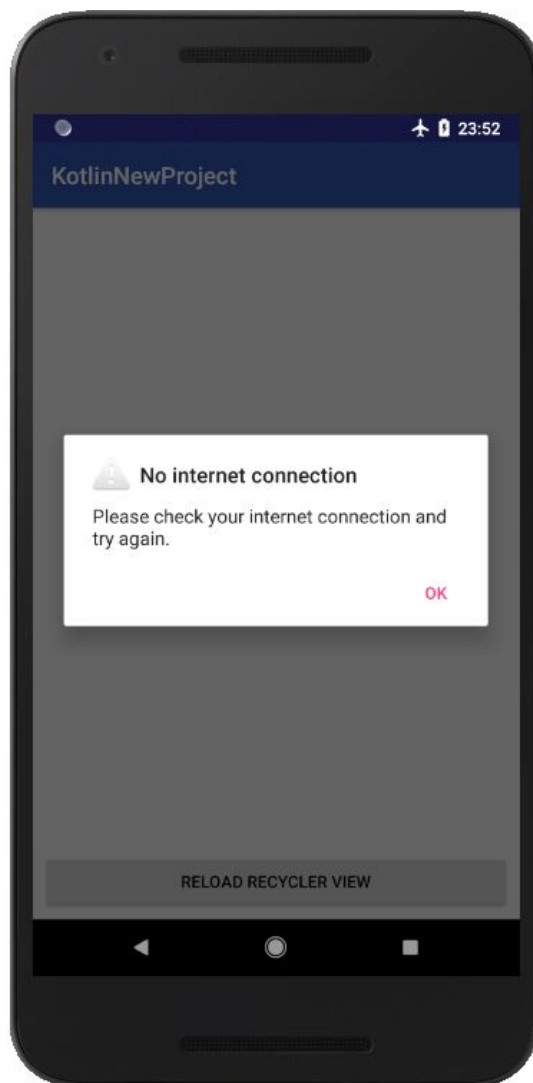


Figura 17: Caixa de alerta para falta de conexão.

Fonte: autor - *Android Emulator (API 27)*

Por último mas não menos importante, vamos carregar as imagens na *RecyclerView* dos *Repositories* utilizando a biblioteca de carregamento de imagens *Glide*. Primeiro devemos importá-la: abra o *build.gradle* do projeto e do *app* e adicione as seguintes configurações, respectivamente:

```
buildscript {  
    //...  
    ext.glide_version = "4.7.1"  
    //...  
}
```

```
dependencies {  
    //...  
    // Glide  
    implementation  
    "com.github.bumptech.glide:glide:$glide_version"  
    annotationProcessor  
    "com.github.bumptech.glide:compiler:$glide_version"  
}
```

Abra a classe *RepositoryAdapter* e adicione o carregamento da imagem dentro do membro *bindView()*.

```
class ViewHolder(itemView: View) :  
    RecyclerView.ViewHolder(itemView) {  
    fun bindView(item: Repository,  
        listener: (Repository) -> Unit) =  
        with(itemView) {  
            Glide.with(context)  
                .load(item.owner.avatar_url)  
                .into(ivMain)  
  
            tvTitle.text = item.name  
            tvOwner.text = item.owner.login  
            tvDescription.text = item.description  
  
            setOnClickListener { listener(item) }  
        }  
    }  
}
```

Para melhorar um pouco a experiência do usuário, podemos adicionar *snackbars* no lugar de alguns *toasts* de *feedback*. Para isso, adicione a *ID* 'root' no *ConstraintLayout* da *MainActivity*. Depois, vá no membro *loadDefaultRecyclerView()* da *MainActivity* e no lugar do *toast* de carregamento de repositórios, adicione a seguinte linha:

```
longSnackbar(root, "Loading ${it.title} repositories...")
```

Com isso terminamos a nossa aplicação trabalhando com *networking*. Fique à vontade para melhorar a aplicação e explorar diferentes funcionalidades para praticar um pouco o desenvolvimento com o *Kotlin* e com as novas bibliotecas aprendidas.

Algumas partes desse projeto foram inspiradas no fluxo presente em um tutorial muito bom criado por *Eunice Obugyei* e publicado no site *raywenderlich.com*, disponível na [referência](#) 34.

7.7. Network Profiling

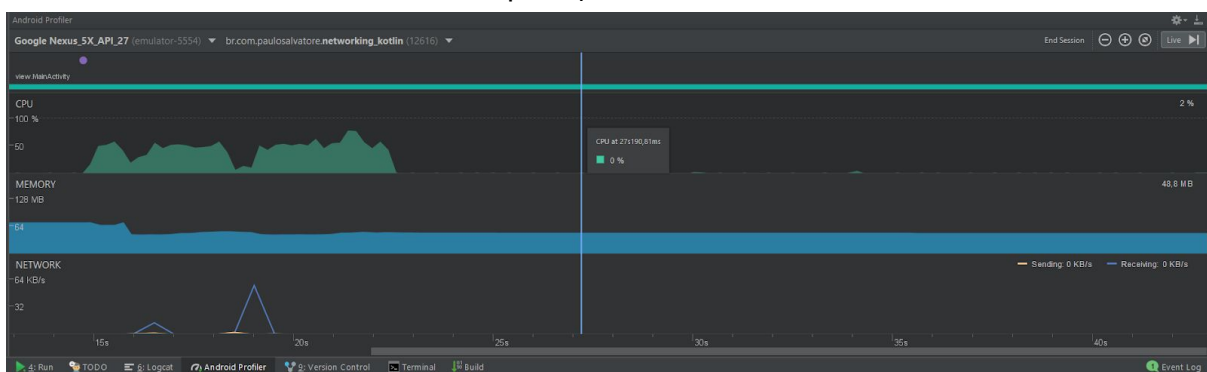
O *Network Profiling* é um assunto de extrema importância, pois nos permite entender exatamente o que está acontecendo com nossa aplicação em termos de comunicação externa com a *web*, que no nosso caso, está sendo feita durante a requisição de dados dos repositórios e também durante o carregamento das respectivas imagens.

A biblioteca *OkHttp* possui o *Interceptor* que nos permite registrar no console as requisições feitas com o *Retrofit*, o que pode ajudar a depurar nossas utilizações da *web*. Entretanto, a partir do *Android Studio* 3.0 foi inserido o *Android Network Profiler*, que substitui a necessidade do *Interceptor*.

Em vez de rodar normalmente nosso *app*, iniciamos a execução clicando no ícone do *Android Profiler* (quando passar o mouse aparecerá o nome *Profile 'app'*), conforme mostrado na figura abaixo.



Na janela que apareceu, selecione o dispositivo que deseja rodar e aguarde a execução do *app*. Note que, diferente de quando executamos normalmente, o *Android Profiler* estará atrelado à aplicação.

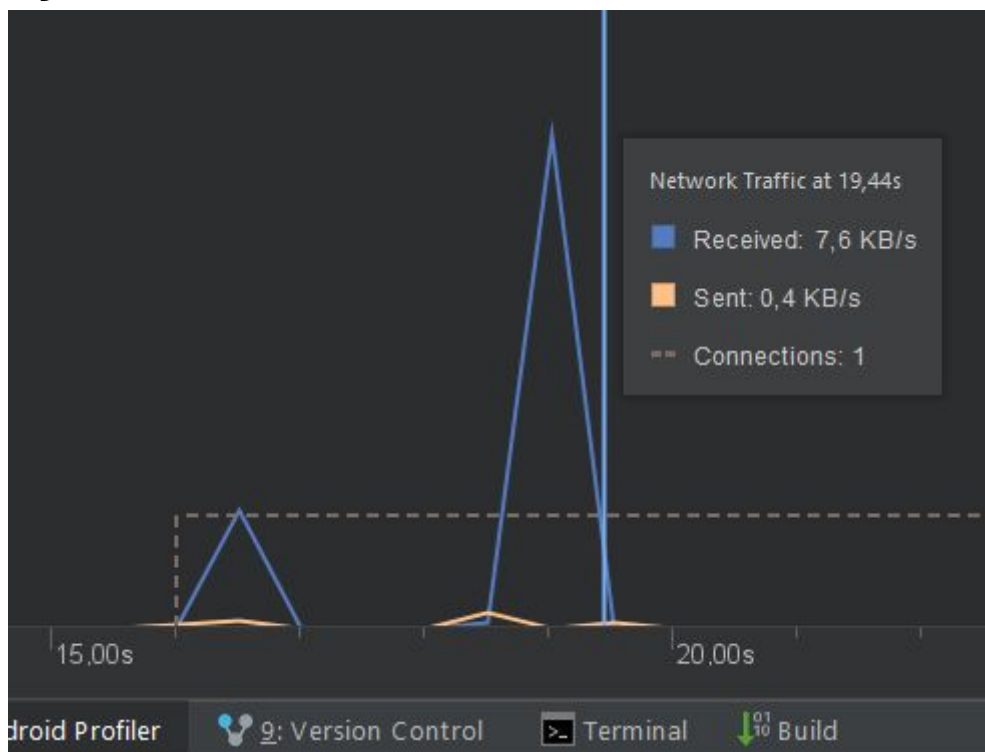


Desenvolvido por [Paulo Salvatore](#)

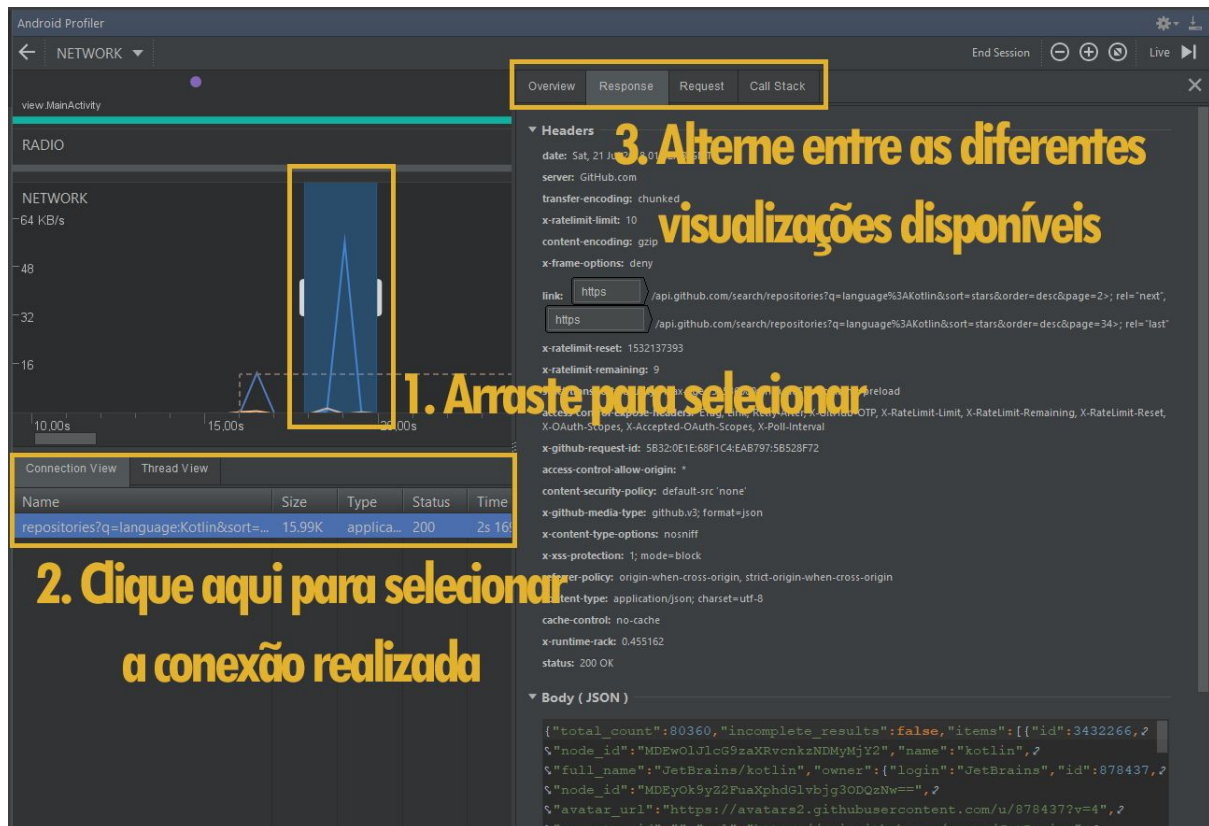
O *profiler* irá exibir os dados em tempo real da performance do seu aplicativo. É possível ativar/desativar a visualização em tempo real dos dados clicando o botão *Live*. Movimentar a barra de rolagem horizontal também irá desativar a visualização em tempo real.



Para acessar as ferramentas com maior nível de detalhes, assim como o *Network profiler*, clique no gráfico correspondente. Note que existem alguns picos de consumo da rede, provavelmente quando o *app* buscou pelos repositórios ou pelas imagens deles.



Para aumentar ainda mais o nível de detalhes, podemos selecionar um trecho que desejamos visualizar, podendo verificar exatamente qual requisição externa foi feita e todas as suas informações completas presentes no cabeçalho e no corpo.



Depurar a comunicação *Network* do nosso projeto nunca foi tão fácil! Agora fica muito mais tranquilo de tratar requisições que não deram certo e entendê-las, assim como também facilitará o processo de entender requisições bem sucedidas e todo o conteúdo recebido, conseguindo visualizar falhas na estrutura dos dados ou outros tipos de abordagens que otimizarão nosso tempo durante em realizações desse tipo.

8. Projeto extra

Além dos projetos desenvolvidos nesse material, durante essa aula desenvolvi uma aplicação que contemplava todos os aspectos aprendidos e explorava um pouco mais o universo das *push notifications* atrelando com as outras bibliotecas que vimos.

Esse projeto está integralmente disponível no *GitHub* para consultas, porém, antes de visualizar o código fonte, gostaria de ter convidar a tentar desenvolver as funcionalidade presentes no *app* da sua maneira, apenas pensando no comportamento de cada componente.

O link atualizado do projeto está em minha conta do *GitHub*, disponível através do seguinte endereço:

https://github.com/paulosalvatore/Push_Images_Networking_Kotlin

No que diz respeito à funcionalidade, temos uma *MainActivity* que concentra todas as ações do *app* e uma *PushActivity* que é adicionada quando o usuário clica em uma notificação que a aplicação criou. Basicamente o foco da aplicação é a construção e personalização de *push notifications*, com a possibilidade de customizar as notificações geradas em diversos aspectos, como é possível observar no fluxo descrito na figura x.

Faça o *download* do projeto no *GitHub* e execute-o para testar o aplicativo. Você também pode instalar o *apk* disponível no seu celular caso prefira.



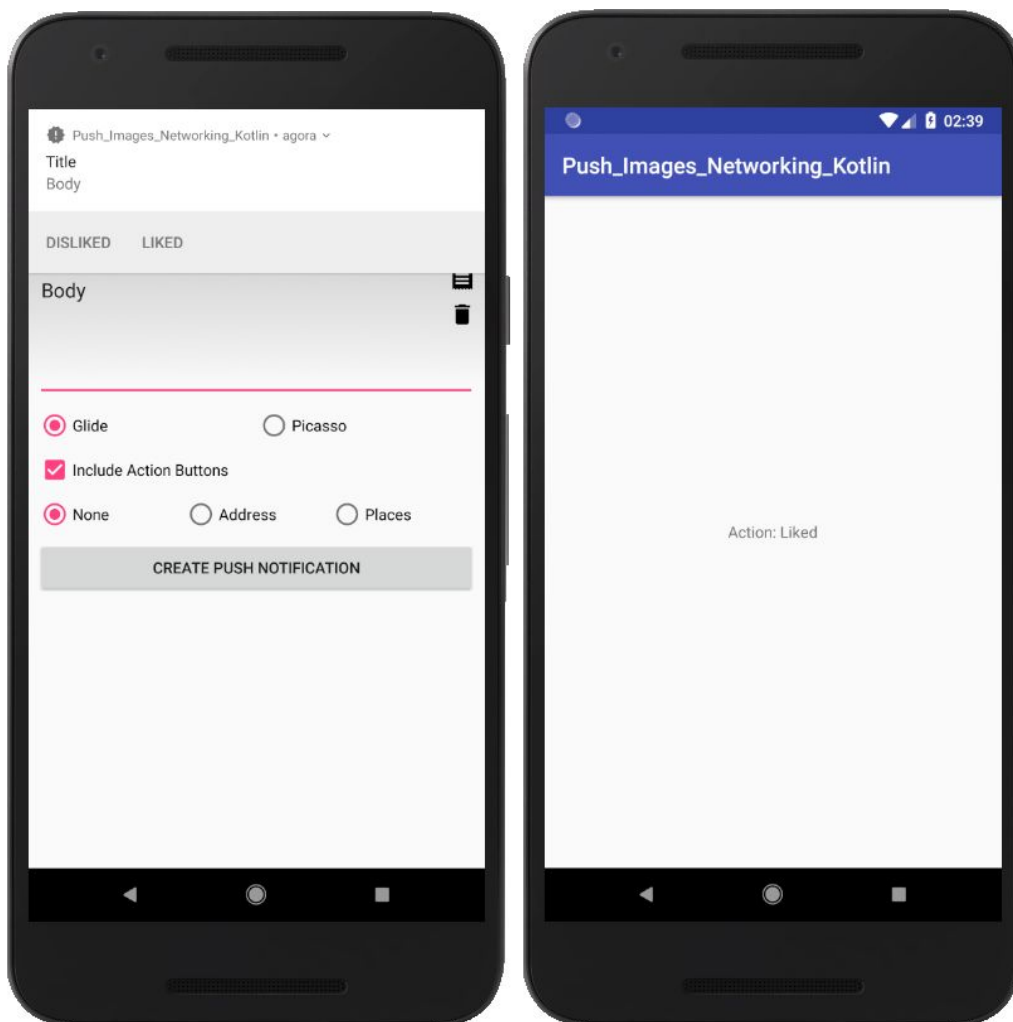
1. *EditText* com Título da notificação
2. *EditText* com Corpo da notificação - Aceita textos longos de múltiplas linhas
3. Botão para gerar texto *placeholder* nos *EditTexts* do título e no corpo da notificação - exibe uma janela de confirmação
4. Botão para limpar os textos inseridos nos *EditTexts* do título e no corpo da notificação - exibe uma janela de confirmação
5. *RadioGroup* de seleção entre tipos de carregamento de imagem, definindo qual biblioteca usar
6. *Checkbox* para inclusão das ações na notificação
7. Espaço reservado para inserção do local a ser pesquisado, seja o endereço ou o lugar (*GooglePlaces*)
8. *RadioGroup* para seleção de qual local será inserido
9. Botão para criar notificação

Desenvolvido por [Paulo Salvatore](#)

Explicando rapidamente o fluxo presente na aplicação, temos uma interface para criação de notificações locais customizadas. Além dos campos autoexplicativos (ou explicados na referência acima), temos algumas particularidades em termos de funcionamento que não estão claras visualmente.

Vale ressaltar que a aplicação está integrada com a *API* do *Firebase* para recebimento de notificações, então é possível enviar novas notificações a partir da nuvem e não apenas utilizando a interface local.

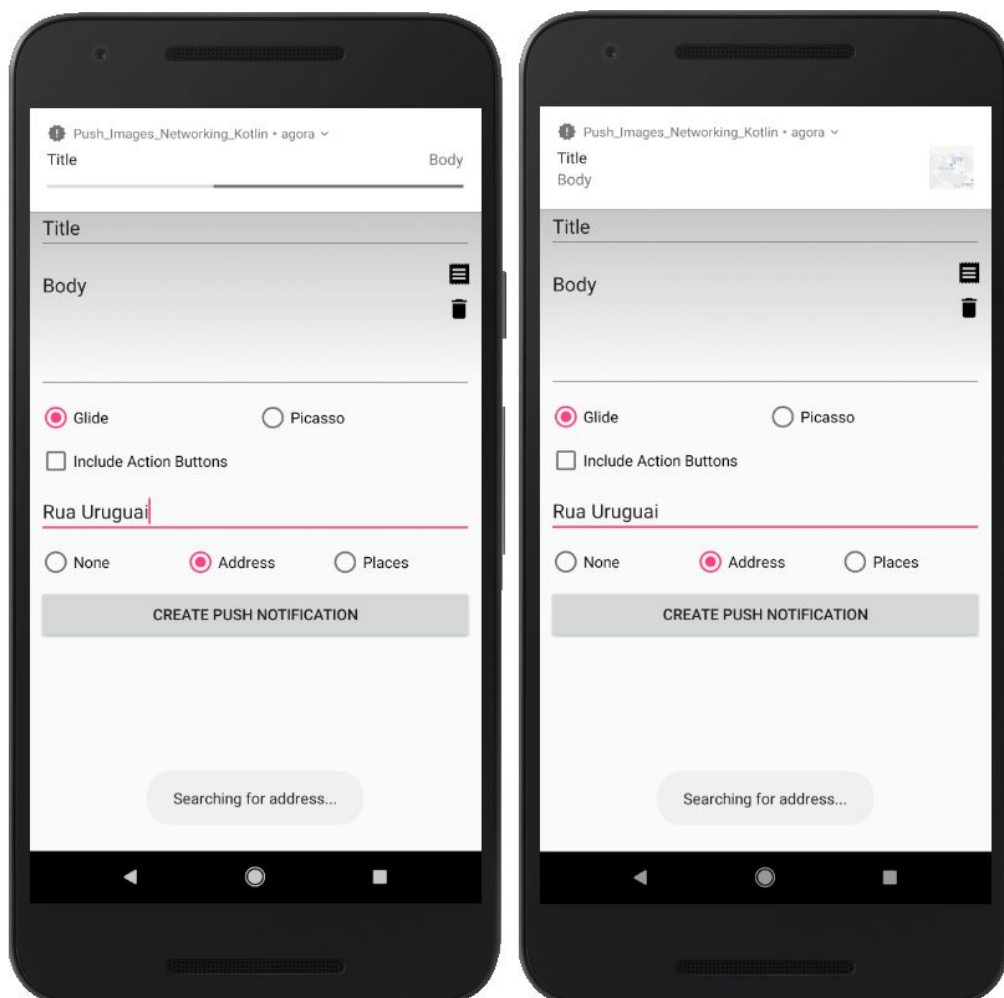
A primeira é a opção de incluir botões de ação, que adiciona dois botões, 'disliked' e 'liked', que quando clicados abre uma nova *Activity* enviando informações que identificam qual deles recebeu o evento. Clicar na notificação também abre a mesma *Activity* dos botões de ação, porém, exibindo 'No action'.



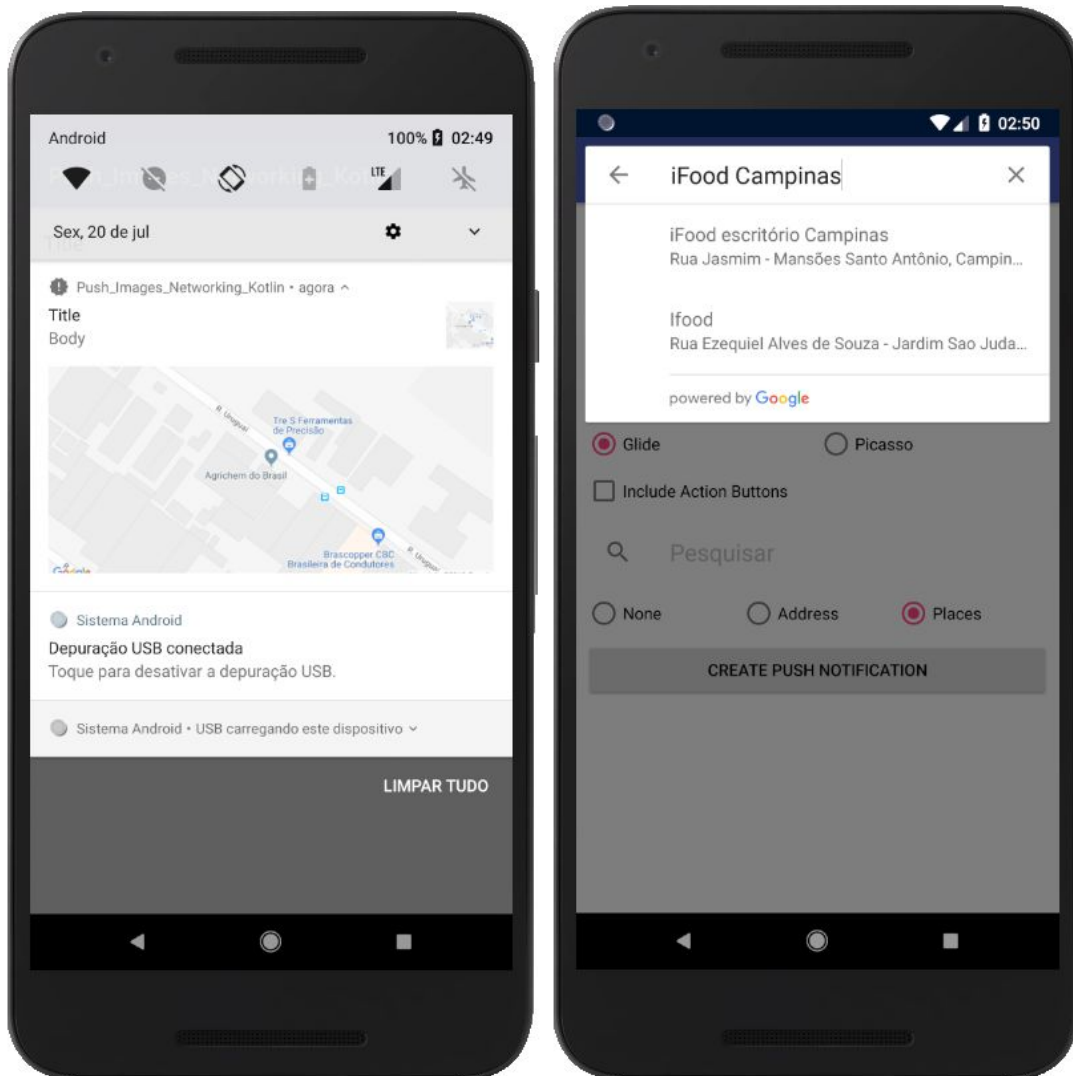
A próxima opção é a de busca de endereço. Quando a opção 'Address' é selecionada, um *EditText* aparece no lugar reservado (ícone 7 da imagem de referência), onde um endereço qualquer deve ser digitado. Ao clicar para criar a notificação, o sistema irá na *API* do *GoogleMaps* buscar esse endereço e retornar o primeiro resultado que encontrar, obtendo sua latitude e longitude.

O mesmo acontece para o 'Places', com a diferença que utilizamos a biblioteca de *Places* do *GoogleMaps* para buscar um lugar, sendo que essa busca é exibida utilizando os recursos prontos de visualização que são fornecidos pelo próprio *Google*.

A seleção de um endereço ou de um lugar é sempre obrigatória e após feita, a notificação criada usará a latitude e a longitude para buscar uma imagem do local em questão e adicionar na notificação, exibindo uma notificação inicial com uma barra de progresso com *loading* indeterminado e quando a imagem for carregada, atualiza a notificação inserindo-a da melhor forma.



Desenvolvido por [Paulo Salvatore](#)



A aplicação apesar de simples irá treinar diversos aspectos abordados nessa aula e deve servir como um bom objeto de estudo de *Kotlin*, *Push Notifications*, *Image Loading*, *Networking* e ainda brincar um pouco com a *API* de *Places* do *Google*.

Qualquer dúvida durante o desenvolvimento não hesite em me mandar!



9. Referências Bibliográficas

1. Kotlin - Wikipedia - <https://pt.wikipedia.org/wiki/Kotlin> (Acessado em 17 de Julho de 2018)
2. Kotlin's GitHub Project - <https://github.com/JetBrains/kotlin> (Acessado em 17 de Julho de 2018)
3. Kotlin: A nova linguagem oficial para desenvolvimento Android - <https://www.treinaweb.com.br/blog/kotlin-a-nova-linguagem-oficial-para-desenvolvimento-android/> (Acessado em 17 de Julho de 2018)
4. Kotlin Documentation - <https://kotlinlang.org/docs/reference/kotlin-doc.html> (Acessado em 18 de Julho de 2018)
5. Kotlin Basic Syntax - <https://kotlinlang.org/docs/reference/basic-syntax.html> (Acessado em 18 de Julho de 2018)
6. Kotlin - Coding Conventions - <https://kotlinlang.org/docs/reference/coding-conventions.html> (Acessado em 17 de Julho de 2018)
7. Kotlin Android Extensions - <https://kotlinlang.org/docs/tutorials/android-plugin.html> (Acessado em 18 de Julho de 2018)
8. Kotlin Anko - <https://github.com/Kotlin/anko> (Acessado em 18 de Julho de 2018)
9. Kotlin Anko Commons - <https://github.com/Kotlin/anko#anko-commons> (Acessado em 18 de Julho de 2018)
10. Time to Upgrade from GCM to FCM - <https://android-developers.googleblog.com/2018/04/time-to-upgrade-from-gcm-to-fcm.html> (Acessado em 18 de Julho de 2018)
11. Firebase Cloud Messaging - <https://firebase.google.com/docs/cloud-messaging/> (Acessado em 18 de Julho de 2018)
12. Loading Large Bitmaps Efficiently - <https://developer.android.com/topic/performance/graphics/load-bitmap> (Acessado em 18 de Julho de 2018)
13. How The Android Image Loading Library Glide and Fresco Works? - <https://blog.mindorks.com/how-the-android-image-loading-library-glide-and-fresco-works-962bc9d1cc40> (Acessado em 18 de Julho de 2018)
14. Solving the Android Image Loading Problem: An Updated Guide - <https://www.bignerdranch.com/blog/solving-the-android-image-loading-problem-an-updated-guide/> (Acessado em 18 de Julho de 2018)
15. Introduction to Glide, Image Loader Library for Android, recommended by Google - <https://inthecheesefactory.com/blog/get-to-know-glide-recommended-by-google/en> (Acessado em 18 de Julho de 2018)
16. Glide Documentation - <https://bumptech.github.io/glide/> (Acessado em 18 de Julho de 2018)
17. Picasso Documentation - <http://square.github.io/picasso/> (Acessado em 18 de Julho de 2018)

18. Notifications

<https://developer.android.com/guide/topics/ui/notifiers/notifications> (Acessado em 19 de Julho de 2018)

19. Take photos - <https://developer.android.com/training/camera/photobasics> (Acessado em 19 de Julho de 2018)

20. Perform network operations overview -

<https://developer.android.com/training/basics/network-ops/> (Acessado em 19 de Julho de 2018)

21. Retrofit Documentation - <http://square.github.io/retrofit/> (Acessado em 19 de Julho de 2018)

22. Volley Page on GitHub - <https://github.com/google/volley> (Acessado em 19 de Julho de 2018)

23. Volley overview - <https://developer.android.com/training/volley/> (Acessado em 19 de Julho de 2018)

24. Fast Android Networking Page on GitHub -

<https://github.com/amitshekharitbhu/Fast-Android-Networking> (Acessado em 19 de Julho de 2018)

25. The Best Android Networking Library for Fast and Easy Networking -

<https://medium.com/mindorks/simple-and-fast-android-networking-19ed860d1455> (Acessado em 19 de Julho de 2018)

26. Is Retrofit faster than Volley? The answer may surprise you! -

<https://medium.com/@ali.muzaffar/is-retrofit-faster-than-volley-the-answer-may-surprise-you-4379bc589d7c> (Acessado em 19 de Julho de 2018)

27. Android JSON Parsers Comparison -

<https://medium.com/@IlyaEremin/android-json-parsers-comparison-2017-8b5221721e31> (Acessado em 19 de Julho de 2018)

28. Moshi vs Gson in android -

<https://stackoverflow.com/questions/43577623/moshi-vs-gson-in-android> (Acessado em 19 de Julho de 2018)

29. Moshi, another JSON Processor -

<https://medium.com/square-corner-blog/moshi-another-json-processor-624f8741f703> (Acessado em 19 de Julho de 2018)

30. GitHub API - Search Documentation -

<https://developer.github.com/v3/search/#search-repositories> (Acessado em 19 de Julho de 2018)

31. Why 45% of all software features in production are NEVER Used. -

<https://www.linkedin.com/pulse/why-45-all-software-features-production-never-used-david-rice/> (Acessado em 20 de Julho de 2018)

32. Step 8: The Boy Scout Rule ~Robert C. Martin (Uncle Bob) -

<https://medium.com/@biratkirat/step-8-the-boy-scout-rule-robert-c-martin-uncle-bob-9ac839778385>

33. Software rot - https://en.wikipedia.org/wiki/Software_rot (Acessado em 20 de Julho de 2018)

34. Android Networking Tutorial: Getting Started -

<https://www.raywenderlich.com/184995/android-networking-tutorial-getting-start>

Desenvolvido por [Paulo Salvatore](#)

[ed-2](#) (Acessado em 20 de Julho de 2017)

10. Vídeos Recomendados

1. Google announces Kotlin for Android | Google I/O 2017 - <https://youtu.be/d8ALcQiuPWs> (Acessado em 17 de Julho de 2018)
2. Introduction to Kotlin (Google I/O '17) - <https://youtu.be/X1RVYt2QKQE> (Acessado em 18 de Julho de 2018)
3. How to Kotlin - from the Lead Kotlin Language Designer (Google I/O '18) - <https://youtu.be/6P20npkvcb8> (Acessado em 19 de Julho de 2018)
4. Modern Android development: Android Jetpack, Kotlin, and more (Google I/O 2018) - <https://youtu.be/lrMw7MEgADk> (Acessado em 19 de Julho de 2018)
5. Volley: why not other networking libraries? - https://youtu.be/GeQ_5x3jS98 (Acessado em 19 de Julho de 2018)
6. KISS Principle in Code - <https://youtu.be/RBiRjXY36mg> (Acessado em 20 de Julho de 2018)

11. Licença e termos de uso

Todos os direitos são reservados. É expressamente proibida a distribuição desse material sem a permissão, por escrito, do **autor** ou da **GlobalCode Treinamentos Ltda - ME**. Mais informações sobre *copyright*:

<https://choosealicense.com/no-permission/>

Conteúdos que foram baseados em declarações providas da documentação do *Google Developers* estão licenciados sob a *Creative Commons License 3.0* (<https://creativecommons.org/licenses/by/3.0/>).

Códigos providos do *Google* estão licenciados sob a *Apache License 2.0* (<https://www.apache.org/licenses/LICENSE-2.0>).

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

- 3 Key Software Principles You Must Understand - <https://code.tutsplus.com/tutorials/3-key-software-principles-you-must-understand-net-25161>

12.1.1. KISS

- KISS (Keep it Simple, Stupid) - A Design Principle - <https://www.interaction-design.org/literature/article/kiss-keep-it-simple-stupid-a-design-principle>

12.1.2. YAGNI

- You're NOT gonna need it! - <https://ronjeffries.com/xprog/articles/practices/pracnotneed/>
- Why 45% of all software features in production are NEVER Used - <https://www.linkedin.com/pulse/why-45-all-software-features-production-never-used-david-rice/>

12.1.3. DRY

- Is Your Code DRY or WET? - <https://dzone.com/articles/is-your-code-dry-or-wet>

12.1.4. The Boy Scout Rule

- Step 8: The Boy Scout Rule ~Robert C. Martin (Uncle Bob) - <https://medium.com/@biratkirat/step-8-the-boy-scout-rule-robert-c-martin-uncle-bob-9ac839778385>

12.2. Kotlin

- What We Learned About Kotlin's Growth This Year - <https://thenewstack.io/what-we-learned-about-kotlins-growth-this-year/>
- Hack - From Java to Kotlin - <https://github.com/MindorksOpenSource/from-java-to-kotlin/>

Desenvolvido por [Paulo Salvatore](#)

- Lessons learned while converting to Kotlin with Android Studio - <https://medium.com/google-developers/lessons-learned-while-converting-to-kotlin-with-android-studio-f0a3cb41669>

12.3. Networking

- Retrofit vs Volley (as of May 2018) - https://www.reddit.com/r/androiddev/comments/8j639v/retrofit_vs_volley_as_of_may_2018/

12.4. Assuntos diversos

- IntelliJ IDEA - ReferenceCard https://resources.jetbrains.com/storage/products/intellij-idea/docs/IntelliJ_IDEA_ReferenceCard.pdf