DESENVOLVIMENTO DE APLICATIVOS MOBILE

Centro Universitário de Santa Fé do Sul - UNIFUNEC

ANDROID: ACESSANDO UMA API WEB – 14/04/2023

Prof. Marcos Antonio Estremote

1) Configuração inicial do Retrofit

Temos uma API web disponível, então começaremos a integrá-la com nosso aplicativo. Dado que usamos o Android Studio e o Java, uma opção nativa que temos é utilizar a classe `HttpURLConnection`, a partir da qual faremos toda a configuração necessária para a realização de requisições HTTP com o mundo externo.

Esta abordagem, porém, é muito similar ao que vimos no SQLite em baixo nível: ela exige um grande número de configurações, e consequentemente o risco de cometermos erros é maior. Por ser em baixo nível, a própria comunidade do Android disponibiliza outras alternativas, inclusive mais comuns.

Usar o `HttpURLConnection`, portanto, serve para entendermos como se dá o funcionamento em baixo nível, ou para de repente darmos suporte a um aplicativo que acabou sendo legado, ou ainda, se quisermos fazer uma configuração de muito baixo nível, que APIs com alto nível não disponibilizam ou dão conta.

Como alternativas, existem bibliotecas muito famosas, como é o caso do Retrofit, e do Volley. O intuito delas é facilitar o acesso HTTP via requisições.

Durante este curso, optaremos pela biblioteca mais utilizada, que é o **Retrofit**, apesar do Volley ser do próprio Google, e sua manutenção ser feita pela equipe do Android. Além disso, por ser uma biblioteca externa, precisamos adicioná-la como sendo uma dependência do projeto.

Existe um script para isso na própria página com a sua documentação,

```
implementation 'com.squareup.retrofit2:retrofit:2.5.0',
```

do Gradle. Copiaremos o script, acessaremos `build.gradle` com "Ctrl + Shift + N", e no final da parte de `dependencies`, colaremos a instrução. Feito isso, clicaremos em Sync Now para sincronizar o projeto.

Vamos fazer a configuração inicial do Retrofit; segundo a documentação, é necessária primeiramente uma instância do Retrofit, com `Builder()`, como

vimos em outras abordagens, e a configuração de um *Service*. Ele será a instância principal para toda a configuração de qualquer requisição HTTP que fizermos.

O *Service* será a entidade que manterá as possíveis requisições, e é desta forma que identificamos os dois principais componentes do Retrofit. Precisaremos centralizar a sua instância, assim como fazemos ao utilizarmos o "database", em que temos a classe `UnifunecDatabase` com a configuração geral da instância.

Criaremos um pacote chamado "retrofit", e dentro dele criaremos a classe única `AlunoRetrofit`, que centralizará todas as configurações, e onde colaremos a implementação do Retrofit diretamente no construtor, que inseriremos com "Alt + Insert" e selecionando Constructor.

E então precisamos acrescentar o que chamamos de **Base URL**, ou URL base. Todas as vezes em que acessamos nossa API, até mesmo naquele teste que fizemos com o Postman, notem que utilizamos `localhost:8080/`. Assim, independentemente da requisição, seja para buscar, inserir, alterar ou remover produtos, esta URL acaba sendo mantida, e, portanto, é a URL base na nossa API.

```
package br.com.unifunec.aluno.retrofit;

import retrofit2.Retrofit;

public class EstoqueRetrofit {

    public EstoqueRetrofit() {
        new Retrofit.Builder()
            .baseUrl("http://localhost:8080/")
    }
}
```

É importante manter a barra final na URL para disponibilizarmos a configuração de outras entradas, também conhecidas por **endpoints**, *como é o caso de* `/alunos`.

No entanto, ao mantermos esta URL base, tentamos acessar nosso próprio computador nesta porta, e no Android não trabalhamos exatamente com o computador em que estamos desenvolvendo. O nosso aplicativo é executado em outro computador, que no caso é um dispositivo móvel, um celular.

Ao testarmos o acesso pelo navegador do celular, o servidor não será encontrado, porque o `localhost` é o celular em si, e precisamos acessar diretamente pelo IP. Ou seja, em primeiro lugar, é necessário identificarmos o IP do nosso computador. Dependendo do computador e do sistema operacional em uso, pode ser que exista um comando para isso.

No Windows, usamos o comando `ipconfig` no terminal. No Linux e no MAC usamos `ifconfig`, e este comando listará todos os adaptadores de rede disponíveis no computador, identificaremos aquele que está em uso, neste caso um Ethernet via rede cabeada, copiaremos o endereço IPv4, a partir do qual o acesso ao servidor ficará disponível, e ajustaremos a `baseUrl`.

Neste caso, o servidor estará em `192.168.20.249:8080/unifunec/alunos` independentemente do dispositivo em uso. Este é um detalhe que deve ser feito no momento de configuração da URL base. E se você for acessar via celular físico, com cabo USB, os dois computadores, aquele em que você estiver desenvolvendo, e o celular, precisam estar na **mesma rede**.

Em seguida, basta adicionarmos um `build()` que devolverá a instância do Retrofit.

```
public EstoqueRetrofit() {
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://192.168.20.249:8080/")
        .build();
}
```

A seguir começaremos a fazer a configuração e execução do nosso *Service*, responsável em determinar a requisição a ser feita.

2) Adicionando o Retrofit ao projeto

Adicione o Retrofit ao projeto colocando a seguinte dependência:

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
```

Sincronize o projeto. Se tudo estive correto, implemente a classe para gerar a instância do Retrofit com base na URL base.

Você pode obter a URL base via prompt de comando do Windows usando o `ipconfig` ou terminal no Linux ou Mac via `ifconfig`. Lembre-se de pegar o ipv4 do adaptador de rede que o computador está conectado e o dispositivo Android também.

Ao pegar o endereço faça o teste substituindo o `localhost` pelo IP, confira se aparece o resultado esperado pela API. Também faça o teste no dispositivo Android.

3) Configurando o Service

Implementada a instância do Retrofit, começaremos a configurar o *Service* que, por ser uma interface, precisa ser criada para que possamos implementá-la. No diretório "retrofit" de nosso projeto, criaremos um pacotes para *Services*, pois eles são destinados a um recurso da nossa API.

Então, dentro de "retrofit", criaremos o diretório "service", e nele criaremos `AlunoService`, em que colaremos o código referente à definição das requisições por meio de assinaturas, da documentação do Retrofit. A assinatura obrigatoriamente devolverá uma `Call`, que é a entidade que representará nossa requisição e permitirá que ela seja executada.

Importaremos a `Call` de Retrofit, e toda vez que a devolvemos, é obrigatório indicar que tipo de retorno é esperado. Para isso, segundo a

documentação, enviamos via Generics, no caso, uma lista de repositórios. No nosso caso, esperamos uma lista de produtos, e além disso, incluiremos uma anotação para indicar o método HTTP a ser executado para esta requisição em específico, e o *endpoint*.

```
import java.util.List;

import br.com.unifunec.aluno.model.Aluno;
import retrofit2.Call;
import retrofit2.http.GET;

public interface AlunoService {

    @GET("aluno")
    Call<List<Aluno>> buscaTodos();
}
```

Assim, a URL base que configuramos será acrescida da barra final (`/`) e de "produto", como definimos acima.

E para utilizarmos este serviço, a partir da instância de `retrofit` precisamos criar outra, referente à interface. Para isso acrescentaremos mais uma linha em `EstoqueRetrofit()`, usando `create()` e passando a referência do *Service*. Então, é necessário devolvermos como se fosse um atributo, a partir do qual teremos um Getter. Usaremos `.field` para que seja criado o atributo.

```
public AlunoRetrofit() {
    Retrofit retrofit = new Retrofit,Builder()
        .baseUrl("http://192.168.20.249:8080/")
        .build();
    alunoService = retrofit.create(AlunoService.class);
}

public AlunoService getAlunoService() {
    return alunoService;
}
```

Deste modo, temos acesso ao nosso *Service* de qualquer lugar do aplicativo. Passaremos ao teste em que buscaremos por todos os produtos da nossa lista. Pelo nosso aplicativo, sabemos que fazemos esta busca internamente na lista de produto, então podemos acessar `ListaAlunosActivity.java` e,em `buscaAlunos()` adicionaremos o comportamento para que isso seja feito na API.

Comentaremos todo o código que faz a busca internamente, pois primeiro testaremos na API web e, caso tudo funcione bem, o próximo passo será a integração com a busca interna. A implementação será feita com a instância de `AlunoRetrofit()`, como havíamos comentado, pegamos o `getAlunoService()` e o `service`, a partir do qual buscaremos a `Call` com `buscaTodos()`.

Para uma `Call`, existem duas alternativas — a execução síncrona e a assíncrona. Teremos os mesmos problemas vistos no curso de Room, que quando executávamos de maneira assíncrona na *main thread*, a tela poderia travar, resultando na interferência do próprio Room, a não ser que incluíssemos uma permissão.

Neste caso, já que faremos uma requisição web, o próprio sistema operacional Android não permite este tipo de execução síncrona. Ou seja, faremos a execução síncrona também em uma Async Task.

Veremos a abordagem assíncrona via `Call` em outro momento.

Precisaremos de dois Listeners, um para ser executado em background, cujo retorno devolveremos para que seja acessível para o outro. Este, por sua vez, fará uma execução deste retorno na UI thread. Faremos a execução **síncrona** via `Call` a partir do método `execute()`, que não pode ser usado de forma assíncrona, ou teremos uma exceção.

O programa acusa um erro de compilação pois exige que se trate a Exception lançada, então usaremos um bloco Try/Catch. Cada vez que o `execute()` for rodado, teremos o retorno de uma entidade denominada `response`, isto é, a resposta que teremos quando fizermos a requisição, como quando estávamos testando no Postman e tínhamos um retorno.

O `response` mantém um Generics na `Call`, pois ele irá compor todo o conteúdo da resposta, seja Body, Header, entre outros. No caso, acessaremos o corpo, que contém nossos produtos, a serem retornados pelo Listener. Dessa forma, eles são enviados ao `onPostExecute()`, o qual faz o envio para o próximo Listener que estará sendo atualizado na UI thread.

Estamos usando um Try/Catch, e pode ser que este retorno não chegue, por alguma falha de comunicação ou exceção. Para estes casos, é necessário declarar outro retorno, nulo. Feito isso, implementaremos o outro Listener com `resultado`. Tendo por padrão que os produtos novos serão existirão, no caso de haver uma referência nula a recomendação é que sempre verifiquemos isso, evitando um NullPointerException e notificando o usuário.

Por fim, basta executarmos a Async Task, responsável por buscar os produtos da API, enviar o retorno ao Listener, que atualizará o Adapter, e notificar caso haja algum problema.

```
private void buscaAlunos() {
    ProdutoService service = new
EstoqueRetrofit().getProdutoService();
    Call<List<Produto>> call = service.buscaTodos();

    new BaseAsyncTask<>(() -> {
        try {
            Response<List<Aluno>> resposta = call.execute();
            List<Aluno> alunosNovos = resposta.body();
            return alunosNovos;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }, alunosNovos -> {
        if(alunosNovos != null) {
            adapter.atualiza(alunosNovos);
        } else {
            Toast.makeText(context: this,
                           text: "Não foi possível buscar os alunos da
API",
                           Toast.LENGTH_SHORT).show();
        }
```

```
    }).execute();

//    new BaseAsyncTask<>(dao::buscaTodos,
//        resultado -> adapter.atualiza(resultado))
//        .execute();
}
```

Vamos executar o nosso aplicativo, levando em consideração que uma implementação deste tipo pode acabar resultando em várias falhas. Por isso, fica a sugestão de manter o Logcat aberto, configurado para "Error". Pode ser que demore um pouco para o programa ser rodado, pois fizemos as primeiras configurações.

Teremos um erro, e o Logcat indica que houve uma tentativa falha de fazer uma conversão para o tipo que queríamos. Isso porque quando utilizamos o Retrofit apenas com a configuração inicial de `AlunoRetrofit()`, é feita uma tentativa de conversão para o seu tipo genérico, que aparece no Logcat como `ResponseBody`.

Com isso, não temos capacidade de configurar a conversão para tipos específicos, como é o caso de uma lista de produtos, ou de entidades que temos. Ou seja, precisaremos acrescentar um conversor, e a seguir veremos os detalhes de como fazê-lo.

4) Buscando os produtos da API

Consultando a documentação do Retrofit, teremos um tópico chamado "Retrofit Configuration" à direita, que explica sobre a configuração para a conversão da resposta para um objeto desejado. É citado o processo de desserialização, que é quando recebemos uma resposta via HTTP e convertemos para um objeto, o padrão é realmente devolver para um `ResponseBody`.

No entanto, é indicado que existem conversores capazes de fazer a devolução para objetos esperados, e é disso que precisamos. São listados conversores que usam bibliotecas que fazem desserialização e serialização de JSON para objetos, como é o caso de Gson, Jackson, Moshi e outros.

Além disso, podemos fazer nosso próprio conversor, e as orientações se encontram na documentação. Neste caso, dado que o objetivo é fazer uma configuração mais simples possível, utilizaremos o **Gson**.

Copiaremos a dependência `com.squareup.retrofit2:converter-gson`, e a colaremos em nosso build.gradle, usando a **mesma versão do Retrofit**:

```
implementation 'com.squareup.retrofit2:converter-gson:2.5.0'
```

Será feito o download da biblioteca, e dependendo pode ser que isso demore, basta aguardar e sincronizar com o projeto, e assim que a sincronização for feita, podemos adicioná-lo à configuração. Em `EstoqueRetrofit.java`, então, durante o processo de `build()`, adicionaremos um Converter Factory.

```
public AlunoRetrofit() {
    Retrofit retrofit = new Retrofit.Builder()
        .baseUrl("http://192.168.20.249:8080/unifunec/")
        .addConverterFactory(GsonConverterFactory.create())
        .build();
    alunoService = retrofit.create(AlunoService.class);
}
```

Vamos limpar o Logcat, deixar a aba aberta e abrir o emulador, pois é muito comum que ocorram erros a que não estamos atentos durante a configuração. O aplicativo é rodado e é exibida a mensagem de erro que definimos anteriormente. Em Logcat, trocaremos "Error" por "Info", e apesar de nenhum erro ser mostrado, checaremos as informações para tentarmos entender o que houve.

Quando fizemos a requisição, ocorreu um problema que foi apresentado durante a versão do Android 9, envolvendo justamente as requisições HTTP. A partir desta versão, em específico da API 28, começamos a ter algumas mudanças de melhorias para o usuário final, pois quando fazemos uma requisição HTTP, que é uma requisição de um protocolo que trafega textos puros.

Ou seja, sempre que fazemos uma requisição, estamos buscando dados como se fossem textos puros pela rede, e o problema disso é que, se de repente estivermos em uma rede que é interceptada por algum software malicioso que verifica as informações trafegadas, o software acabará capturando um dado sensível, como uma senha, número de cartão de crédito, e assim por diante.

Pensando em evitar este tipo de problema, a própria equipe do Android, a partir da versão 9, da API 28, decidiu não permitir que o padrão seja feito em requisições HTTP, tanto que o Logcat indica que a exceção `UnknownServiceException` diz que "CLEARTEXT communication", que seria a comunicação de texto limpo envolvendo o HTTP para o endereço que configuramos (192.168.20.249) não é permitido via as regras de política de segurança de rede.

Assim, temos duas alternativas: idealmente, poderíamos utilizar um `https` em vez de uma requisição HTTP "pura". Porém, dado que nossa API não foi configurada para um HTTP, ou se estivermos trabalhando com uma API de teste sem certificação para HTTPS, podemos permitir uma requisição deste tipo.

Não é recomendado fazê-lo justamente pelo risco que corremos pelo usuário, mas podemos testar esta abordagem. Para isso, acessamos `AndroidManifest.xml`, e em nosso `application` utilizamos um atributo chamado `usesCleartextTraffic` para indicar que queremos que isso seja permitido, usando o valor `true`, `android:usesCleartextTraffic="true"`.

Ao fazermos isso, o Android Studio inclusive nos orienta a buscar outras opções, informando também que ele passou a ser usado a partir da API 23. Vamos então executar e verificar o que acontece. É exibido

um `SocketException` no Logcat, para indicar que não temos acesso para este tipo de ação.

Isso acontece pois, dependendo da API utilizada no Android, são necessárias permissões de sistemas, como é o caso do acesso à rede, de mapas (GPS), câmeras, entre outros. Para que esta execução de rede seja permitida, precisamos copiar um script de `<uses-permission>` na documentação do Android, que solicitará a permissão ao usuário.

Colaremos o script no `manifest` de `AndroidManifest.xml`, **antes** (fora) de `application`:

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

Neste caso, então, queremos uma permissão para acesso à internet, ou rede, e uma para o estado dela. Feito isso, reexecutaremos o emulador, e desta vez os nossos produtos são carregados com sucesso. Lidamos com toda a complexidade da parte de configurações, cuidados e outros detalhes importantes neste tipo de implementação.

Ainda há o que ser melhorado, claro, como é o caso de passarmos à integração da chamada interna e verificarmos estratégias possíveis para manter um fluxo que faça sentido. Continuaremos a seguir!

5) Adicionando converter e permitindo requisição HTTP

Adicione o converter para que a requisição converta o `ResponseBody` para `List<Produto>`. Para isso adicione um dos conversores do Retrofit ao projeto.

- Gson: `com.squareup.retrofit2:converter-gson`
- Jackson: `com.squareup.retrofit2:converter-jackson`
- Moshi: `com.squareup.retrofit2:converter-moshi`
- Protobuf: `com.squareup.retrofit2:converter-protobuf`
- Wire: `com.squareup.retrofit2:converter-wire`
- Simple XML: `com.squareup.retrofit2:converter-simplexml`
- Scalars (primitives, boxed, and String): `com.squareup.retrofit2:converter-scalars`

No curso foi utilizado o converter do Gson, a versão é a mesma do Retrofit (`2.9.0`). Fique à vontade para usar o converter de sua preferência, porém, alguns deles podem exigir configurações extras.

Adicione o converter no momento de build do Retrofit. Em seguida, adicione as permissões de rede do Android e permita a execução de requisições

http por meio do atributo `usesClearTextTraffic` na application via **AndroidManifest.xml**.

Teste o App e veja se os produtos contidos na API são apresentados.

6) Para saber mais - Sobre alternativas para permitir requisições HTTP

Como mencionado em vídeo, permitir a requisição HTTP não é uma boa prática, pois vai contra a política de segurança exigida a partir da versão 9 do Android.

A abordagem recomendada é por meio da configuração de segurança de rede. Para isso, você precisa criar um arquivo XML de configuração dentro do diretório **res/xml** com o nome `network_security_config`, então indicar na application via **AndroidManifest.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application
android:networkSecurityConfig="@xml/network_security_config">
      <!-- restante do configuração -->
    </application>
</manifest>
```

Dentro dele é possível definir, por exemplo, domínios onde você permite o acesso via HTTP, como é o caso do IP do seu computador.

7) Verificando possíveis erros

Antes de seguirmos com a implementação de nosso app, destacaremos um detalhe muito importante na integração do aplicativo com uma API web: até aqui, graças aos feedbacks precisos, viemos tratando de alguns problemas que foram surgindo, como quando não tínhamos permissão de acesso à internet, ou tentamos fazer uma requisição HTTP e fomos notificados de que no Android 9 não temos esta possibilidade, devido às regras de política de segurança.

Durante uma integração que faça o consumo de uma API web, será comum termos erros não tão facilmente identificáveis. Para exemplificar, faremos uma alteração aparentemente banal — acessaremos `AlunoService` e, em vez de usarmos o endereço URL base com `/aluno`, deixamos `alunos`.

Limparemos o Logcat e executaremos o emulador, que exibe aquela mensagem "Não é possível buscar os produtos da API", que nós colocamos em nossa Activity. Ao consultarmos o Logcat temos um erro que não nos diz muito do que houve, ou seja, não temos um feedback ideal.

Para conseguirmos facilitar esta investigação, existem duas alternativas bem interessantes: no Android Studio, além da execução comum, *Run*, existe outra, denominada *Profile 'app'*, que monitora tudo que acontece em nosso aplicativo, em específico o nosso dispositivo.

Vamos testar executando o aplicativo novamente, desta vez em Profile 'app'. Será aberta uma aba com um perfil criado automaticamente (a chamada **sessão**), que indica o monitoramento de tudo que acontece em nosso aplicativo. Assim, conseguimos pausar a execução para analisar melhor determinada situação em relação a processamento, memória, rede, energia.

Ao clicarmos em rede (*Network*), por exemplo, temos que houve um pico que representa o que foi enviado e recebido, e conseguimos selecionar trechos se clicarmos e arrastarmos com o mouse. E então, se clicarmos na barrinha abaixo de "Timeline", verificamos a requisição realizada em detalhes.

O Retrofit é uma camada acima de `okhttp`, que é o User-Agent exibido na aba Request, que realmente faz a execução, a requisição web.

Como alternativa que não nos faz tão dependentes, existe uma biblioteca bastante utilizada em verificações para cada chamada que fazemos com o Retrofit. O Profile 'app' acaba pegando todas as chamadas de todas as bibliotecas, então se de repente estivermos usando Retrofit junto a uma biblioteca externa que não tem a ver com a comunicação que fazemos com a API direta, isso será exibido na timeline e pode não ser tão interessante.

Esta biblioteca é denominada *Logging interceptor*, cuja proposta é interceptar todas as requisições feitas com o Retrofit com maior precisão, em específico o próprio OkHttp, agente principal responsável pelas requisições. Sendo assim, ele conseguirá fazer o log de tudo que estiver acontecendo.

Para adicioná-lo, basta utilizar a dependência `implementation 'com.squareup.okhttp3:logging-interceptor:(insert latest version)'` via Gradle. Então a copiaremos e colaremos em nosso build.gradle. Antes disso, pausaremos a execução e excluiremos a sessão, para deixar a aba limpa quando formos executar outra. Caso você queira manter por motivos de histórico, fique à vontade.

Não podemos nos esquecer de incluir a versão mais recente na dependência, e para isso acessaremos repositórios que distribuem o Logging interceptor, como o MVN Repository e o JCenter. Ambos disponibilizam, como última versão, a `3.13.1`, portanto será esta que utilizaremos: `implementation 'com.squareup.okhttp3:logging-interceptor:3.13.1'`.

Em seguida, faremos a sincronização, que envolve o download e acesso a esta lib, e sua configuração. Iremos inserir a configuração que vimos no site do OkHttp no momento em que criamos o Retrofit em `EstoqueRetrofit()`. Teremos que importá-lo com "Alt + Enter", e a única alteração que faremos é trocar o `Level.BASIC` por `Level.BODY`, que é o maior nível.

Percebam que criamos o `client` do `OkHttpClient`, o qual precisaremos adicionar e atrelar ao Retrofit, pois ele criará um `client` por padrão, e precisaremos adicionar um com as modificações que queremos, neste

caso, o Logging interceptor. Então, no processo de *build* incluiremos `client()`, que assim ficará acessível para ser adicionado.

```
public AlunoRetrofit() {
    HttpLoggingInterceptor logging = new HttpLoggingInterceptor();
    logging.setLevel(Level.BODY);
    OkHttpClient client = new OkHttpClient.Builder()
                    .addInterceptor(logging)
                    .build();

    Retrofit retrofit = new Retrofit.Builder()
                    .baseUrl("http://192.168.20.249:8080/")
                    .client(client)

.addConverterFactory(GsonConverterFactory.create())
                    .build();
    produtoService = retrofit.create(ProdutoService.class);
}
```

Vamos limpar o Logcat e executar o aplicativo no emulador com "Shift + F10" para verificarmos o que temos como feedback agora que temos um interceptador. Como resultado, obtivemos a mensagem de erro que incluímos na Activity. No Logcat, teremos maiores informações, agora mudando para o modo verboso, ou "Verbose", que é um modo debug.

Teremos que foi feito um GET para um endereço específico, que foi finalizado, a partir do qual obtivemos um 404. É mostrado tudo o que aconteceu, incluindo o que foi transferido, o resultado de Body, entre outros detalhes. Isso acaba sendo bastante similar ao que acontece no modo Profile, com o log sempre disponível para nós.

Com isso, temos flexibilidade e facilidade de entender o que acontece a cada requisição, e isso será muito útil quando não dominamos a API, ou quando fazemos uma configuração extra e acabamos cometendo erros de digitação, por exemplo. Corrigiremos `produto` de `ProdutoService.java`, e reexecutaremos a aplicação para confirmarmos que tudo foi feito corretamente.

Nos próximos passos acabaremos fazendo mais integrações, se de repente tivermos algum problema, conseguimos identificá-lo rapidamente.

8) Adicionando o logging interceptor

Faça com que os detalhes requisições feitas pelo Retrofit sejam apresentadas no logcat. Para isso adicione o logging interceptor:

```
implementation 'com.squareup.okhttp3:logging-interceptor:4.5.0'
```

Então configure-o para apresentar informações de nível `BODY` (mais detalhadas) e atribua o client do OkHttp atrelado ao logging interceptor ao processo de build do Retrofit.

Teste o App e veja se aparece as informações das requisições.