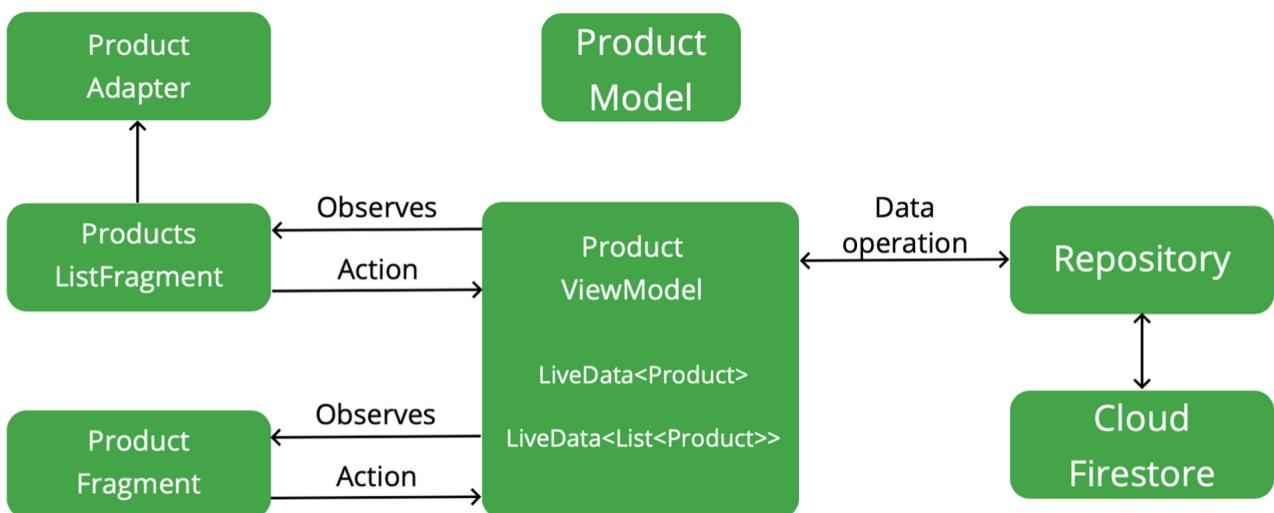




siecola.com.br

# Desenvolvimento Android com Firebase



# Desenvolvimento Android com Firebase

Desenvolva apps Android com o Firebase e serviços como Authentication, Firestore, Analytics e Remote Config

Paulo Cesar Siecola

Esse livro está à venda em <http://leanpub.com/androidcloud>

Essa versão foi publicada em 2020-05-26



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2018 - 2020 Paulo Cesar Siecola

*Para Matilde, Doralice, Hannah e Clotilde, minhas adoráveis cachorras loucas!*

# Conteúdo

<b>1 - Introdução</b>	2
1.1 - A quem se destina esse livro	2
1.2 - O que é necessário para desenvolver aplicativos para Android	2
1.3 - Estrutura didática do livro	3
1.4 - Capítulos do livro	3
<b>2 - Sobre o sistema operacional Android</b>	10
2.1 - Um sistema feito em camadas	10
2.2 - Android virtual device	12
2.3 - Como desenvolver aplicativos para o sistema Android	12
2.4 - Conclusão	16
<b>3 - Sobre o Google Firebase</b>	17
3.1 - Firebase Cloud Messaging	17
3.2 - Firebase Authentication	17
3.3 - Firebase Analytics	17
3.4 - Firebase Remote Config	18
3.5 - Firebase Cloud Firestore	18
3.6 - Conclusão	18
<b>4 - Preparando o ambiente de desenvolvimento</b>	20
4.1 - Instalação do Android Studio	20
4.2 - Seleção de pacotes adicionais	20
4.3 - Criação e configuração do AVD	22
4.5 - Configuração de um dispositivo real para ser utilizado durante o desenvolvimento	24
4.5 - Criação da conta do Firebase	24
4.6 - Conclusão	24
<b>5 - Um pouco sobre Kotlin</b>	25
5.1 - O básico de Kotlin	25
5.2 - Classes e objetos	31
5.3 - Conceitos avançados	34
5.4 - Conclusão	34
<b>6 - Criação do primeiro projeto</b>	35

## CONTEÚDO

6.1 - Criando o primeiro projeto . . . . .	37
6.2 - Estrutura do projeto . . . . .	40
6.3 - Arquivo build.gradle . . . . .	42
6.4 - Criando a primeira interface gráfica . . . . .	44
6.5 - Criando o comportamento da interface gráfica . . . . .	54
6.6 - Conclusão . . . . .	61
<b>7 - Activity e seu ciclo de vida . . . . .</b>	<b>62</b>
7.1 - O que é uma Activity . . . . .	62
7.2 - O ciclo e vida de uma activity . . . . .	63
7.3 - Salvando o estado de uma Activity . . . . .	69
7.4 - Conclusão . . . . .	72
<b>8 - Consumindo serviços REST com Retrofit e coroutines . . . . .</b>	<b>73</b>
8.1 - O provedor de serviços de vendas . . . . .	77
8.2 - Criando o novo projeto para consumir serviços REST . . . . .	86
8.3 - Armazenando o token de autenticação nas SharedPreferences . . . . .	89
8.4 - Criando o cliente REST com Retrofit . . . . .	93
8.5 - Requisitando o token OAuth . . . . .	99
8.6 - Interceptando requisições para inserir o token OAuth . . . . .	102
8.7 - Criando o ViewModel de produtos . . . . .	104
8.8 - Requisitando a lista de produtos com coroutines . . . . .	105
8.9 - Analisando o comportamento da aplicação através de logs . . . . .	109
8.10 - Conclusão . . . . .	111
<b>9 - Utilizando Fragments e RecyclerView . . . . .</b>	<b>112</b>
9.1 - Adicionando novas dependências ao projeto . . . . .	115
9.2 - Criando o fragmento para a lista de produtos . . . . .	116
9.3 - Adicionando o mecanismo de navegação entre telas . . . . .	131
9.4 - Analisando a integração entre o Fragment e o ViewModel . . . . .	134
9.5 - Criando o fragmento para os detalhes do produto . . . . .	137
9.6 - Atualizando a lista de produtos . . . . .	154
9.7 - Conclusão . . . . .	156
<b>10 - Recebendo mensagens pelo Firebase Cloud Messaging . . . . .</b>	<b>157</b>
10.1 - O que é Firebase Cloud Messaging . . . . .	157
10.2 - Arquitetura do projeto Android com FCM . . . . .	159
10.3 - Criando o novo projeto no Android Studio . . . . .	160
10.4 - Criando o projeto no Firebase . . . . .	162
10.5 - Criando a classe de serviço do FCM . . . . .	165
10.6 - Criando o ViewModel . . . . .	170
10.7 - Criando o binding adapter . . . . .	171
10.8 - Recebendo as notificações na MainActivity . . . . .	172
10.9 - Criando o fragmento para exibir as mensagens . . . . .	175

## CONTEÚDO

10.10 - Testando o recebimento de mensagens . . . . .	184
10.11 - Conclusão . . . . .	187
<b>11 - Autenticando usuários com Firebase Authentication e persistindo dados com o Firestore . . . . .</b>	<b>189</b>
11.1 - O que o Firebase Authentication . . . . .	189
11.2 - O que é Firestore . . . . .	190
11.3 - Arquitetura do projeto . . . . .	191
11.4 - Criando o novo projeto no Android Studio . . . . .	193
11.5 - Criando o novo projeto no Firebase . . . . .	194
11.6 - Autenticando o usuário com Firebase Authentication . . . . .	196
11.7 - Criando a estrutura base do projeto . . . . .	201
11.8 - Entendendo a hierarquia de dados no Firestore . . . . .	204
11.9 - Criando as regras de acesso ao Firestore . . . . .	205
11.10 - Criando o repositório de produtos . . . . .	208
11.11 - Criando e visualizando os dados do Firestore com Two-way data binding . . . . .	213
11.12 - Simulando a atualização em tempo real . . . . .	222
11.13 - Conclusão . . . . .	224
<b>12 - Entendendo o comportamento da aplicação e dos usuários com Firebase Analytics . . . . .</b>	<b>225</b>
12.1 - Analisando o dashboard do Firebase Analytics . . . . .	226
12.2 - Habilitando o dispositivo para o modo debug do Firebase Analytics . . . . .	228
12.3 - Construindo a funcionalidade de apagar um produto . . . . .	229
12.4 - Gerando eventos com ações do usuário . . . . .	231
12.5 - Visualizando os eventos no Firebase Analytics . . . . .	234
12.6 - Conclusão . . . . .	237
<b>13 - Alterando o comportamento da aplicação com o Firebase Remote Config . . . . .</b>	<b>238</b>
13.1 - Adicionando a biblioteca do Firebase Remote Config . . . . .	238
13.2 - Lendo a configuração do Firebase Remote Config . . . . .	239
13.3 - Mudando o comportamento da aplicação com o parâmetro do Firebase Remote Config	241
13.4 - Criando o parâmetro no console do Firebase Remote Config . . . . .	241
13.5 - Conclusão . . . . .	243



# 1 - Introdução

Bem vindo ao livro **Desenvolvimento Android com Firebase!** Com ele, o leitor construirá aplicativos para Android utilizando as mais modernas técnicas, tecnologias e arquiteturas existentes. Além disso, utilizará o **Google Firebase**, uma poderosa plataforma de *cloud computing* que oferece, dentre outras coisas, mecanismos de autenticação, banco de dados, registos de eventos e muito mais!

O leitor que decidiu por esse livro já deve saber da importância que aplicativos para dispositivos móveis representam na vida das pessoas, bem como da demanda pelas empresas por desenvolvedores com habilidades nessa área.

Seja por *hobby* ou para alcançar uma vaga de emprego tão sonhada, aprender a desenvolver aplicativos para Android pode ser **desafiador e divertido!**

## 1.1 - A quem se destina esse livro

O público alvo desse livro são desenvolvedores com conhecimento em programação orientada a objetos, que desejam conhecer e desenvolver aplicativos **Android** utilizando serviços do **Google Firebase**.

Todos os aplicativos desenvolvidos aqui serão criados utilizando a linguagem **Kotlin**, criada pela JetBrains. Ter conhecimento nessa linguagem é desejável, mas a estrutura didática do livro considera que o leitor não tem nenhuma experiência prévia. Por isso, alguns conceitos importantes sobre Kotlin serão apresentados ao longo do livro, principalmente aqueles mais utilizados para o desenvolvimento de aplicativos para Android.

Apesar desse livro ser voltado a conceitos e arquiteturas não triviais no âmbito do desenvolvimento de aplicativos para Android, ele não considera como requisito um conhecimento prévio do leitor nesse assunto. O mesmo se aplica ao Google Firebase. Por isso, seu conteúdo passará por assuntos básicos até chegar ao seu propósito final, como pode ser visto em detalhes na seção 1.4 desse capítulo.

## 1.2 - O que é necessário para desenvolver aplicativos para Android

Para aproveitar esse livro em sua totalidade, acompanhar os exercícios práticos e desenvolver os exercícios propostos, será necessário utilizar a IDE Android Studio. Nesse link<sup>1</sup> é possível visualizar os requisitos mínimos necessários da máquina de desenvolvimento.

---

<sup>1</sup><https://developer.android.com/studio>

Também será necessário um dispositivo Android com versão 4.4 - Kitkat ou superior. Caso não possua um aparelho com Android, é possível criar um dispositivo emulado, porém, os requisitos da máquina de desenvolvimento deverão ser maiores, principalmente em termos de memória RAM.

Também será necessário criar uma conta no [Firebase<sup>2</sup>](#), para criação dos recursos que serão utilizados nele.



O capítulo 3 irá detalhar o processo de preparação da máquina de desenvolvimento, instalando as ferramentas e configurando-as.

Todas as ferramentas e contas necessárias para o acompanhamento desse livro podem ser obtidas ou criadas de forma gratuita.

## 1.3 - Estrutura didática do livro

A estrutura didática desse livro permeia um conceito conhecido como **aprendizado baseado em problemas**, onde o leitor é apresentado e conduzido aos conceitos chaves através de problemas que devem ser resolvidos utilizando tais conhecimentos.

Obviamente, nem todos os conceitos podem ser apresentados dessa forma, mas para aqueles que permitem, será utilizado uma abordagem mais prática de aprendizado, fazendo com que o leitor sempre tenha em mente um problema a ser resolvido utilizando a tecnologia que é detalhadamente apresentada. Dessa forma, tendo-se sempre em mente o problema alvo a ser resolvido, o leitor pode absorver os conceitos que são apresentados de forma mais eficiente.

## 1.4 - Capítulos do livro

Os capítulos serão apresentados da seguinte forma:

O **capítulo 2** apresenta um pouco sobre o sistema operacional Android e o **capítulo 3** mostra conceitos por trás da plataforma Firebase que será utilizada pelos aplicativos desenvolvidos nesse livro.

Apesar desse ser um livro totalmente voltado à prática, é importante apresentar alguns conceitos iniciais sobre as plataformas que serão utilizadas, para que o leitor tome conhecimento sobre tudo o que será utilizado, antes de mergulhar no código.

O **capítulo 4** instrui o leitor a preparar seu ambiente de desenvolvimento, no processo de instalação das ferramentas e bibliotecas necessárias, bem como suas configurações.



O processo de preparação do ambiente de desenvolvimento pode ser bem demorado. Se desejar, pule para o **capítulo 4** e comece esse trabalho, antes de continuar sua leitura.

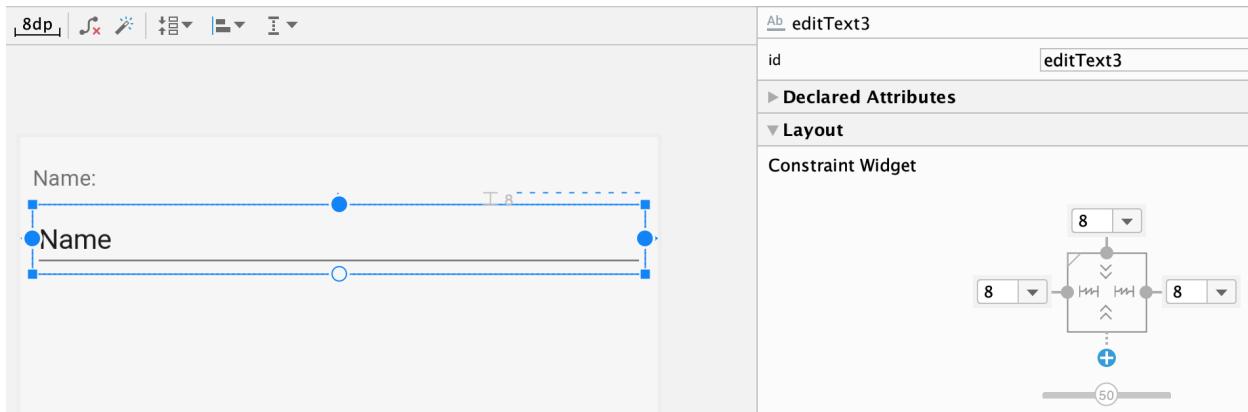
---

<sup>2</sup><http://firebase.google.com/>

No **capítulo 5** são apresentados alguns conceitos da linguagem Kotlin, focado no que será utilizado para o desenvolvimento dos aplicativos nesse livro.

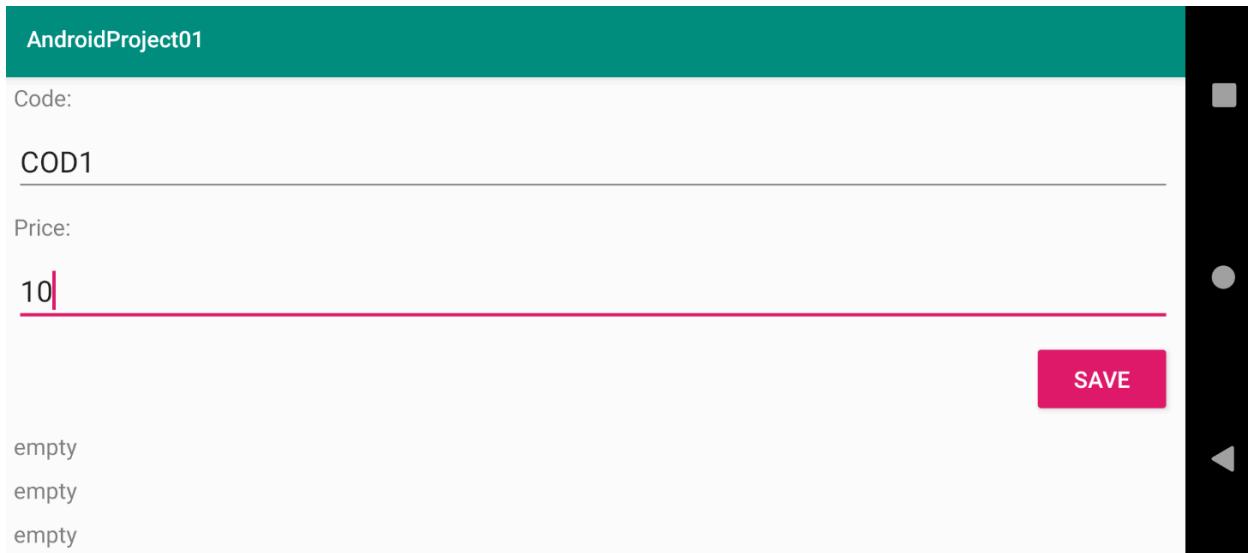
```
var dogs = listOf("Matilde", "Doralice", "Hannah", "Clotilde")
for (dog in dogs) {
    println("New dog is: ${dog}")
}
```

No **capítulo 6** começa o desenvolvimento de um aplicativo bem simples, somente com o intuito de demonstrar a ferramenta Android Studio, a linguagem Kotlin e a estrutura de um projeto inicial, utilizando ConstraintLayout para a construção da interface gráfica:



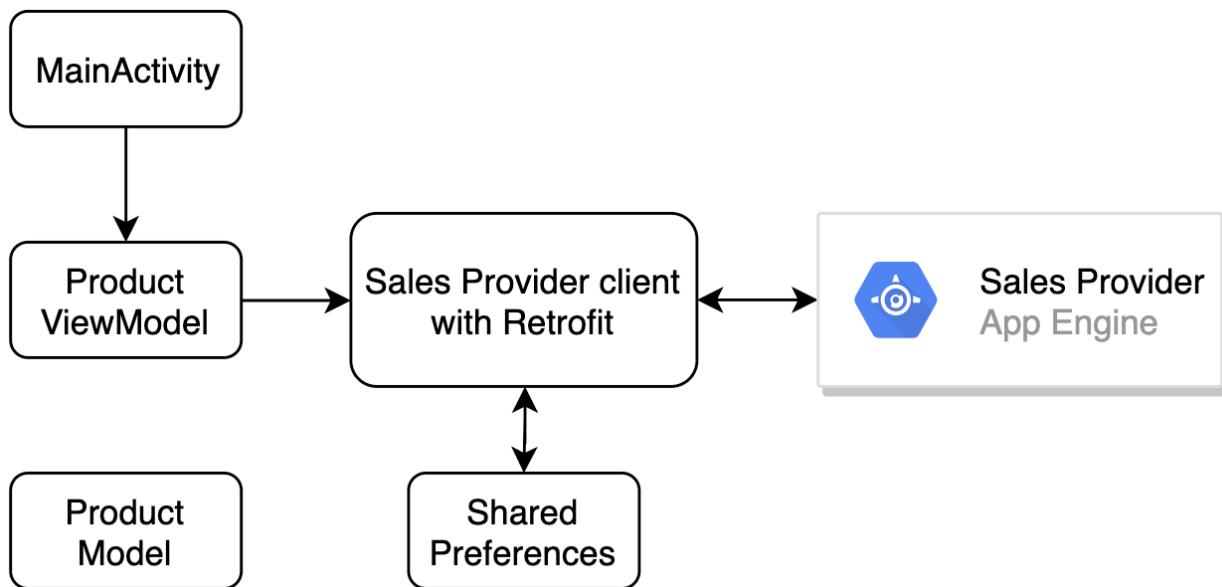
Constraints do EditText

Ainda com o mesmo aplicativo desenvolvido, o **capítulo 7** apresenta alguns desafios do mundo de dispositivos móveis e como resolvê-los com técnicas simples. Aqui, serão apresentados conceitos como `Activity`, seu ciclo de vida e como lidar com seus estados.



Alteração de configuração do dispositivo

O capítulo 8 inicia um novo aplicativo para consumir serviços REST com OAuth utilizando o framework Retrofit.



Arquitetura do projeto

Nele também serão utilizados conceitos como ViewModel e Kotlin coroutines.

O capítulo 9 continua o segundo aplicativo, introduzindo conceitos de listas criadas com o componente RecyclerView, assim como a navegação entre telas utilizando o NavigationController, que permite a utilização de *safe args* entre fragmentos.

AndroidProject02		
Product1	COD1	\$ 10.00
Product2	COD2	\$ 20.00
Product3	COD3	\$ 30.00

Tela inicial com a lista de produtos

No capítulo 10 um novo aplicativo será criado para receber mensagens através do **Firebase Cloud Messaging**, dando início a parte do livro que trata sobre a interação de aplicações Android com a plataforma Firebase.

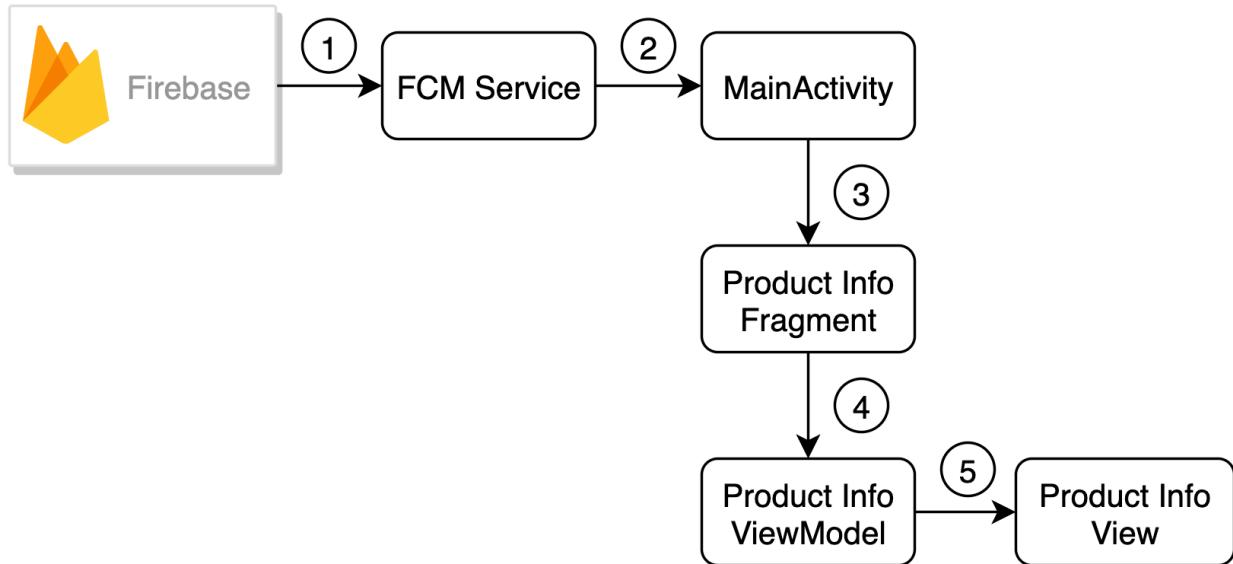
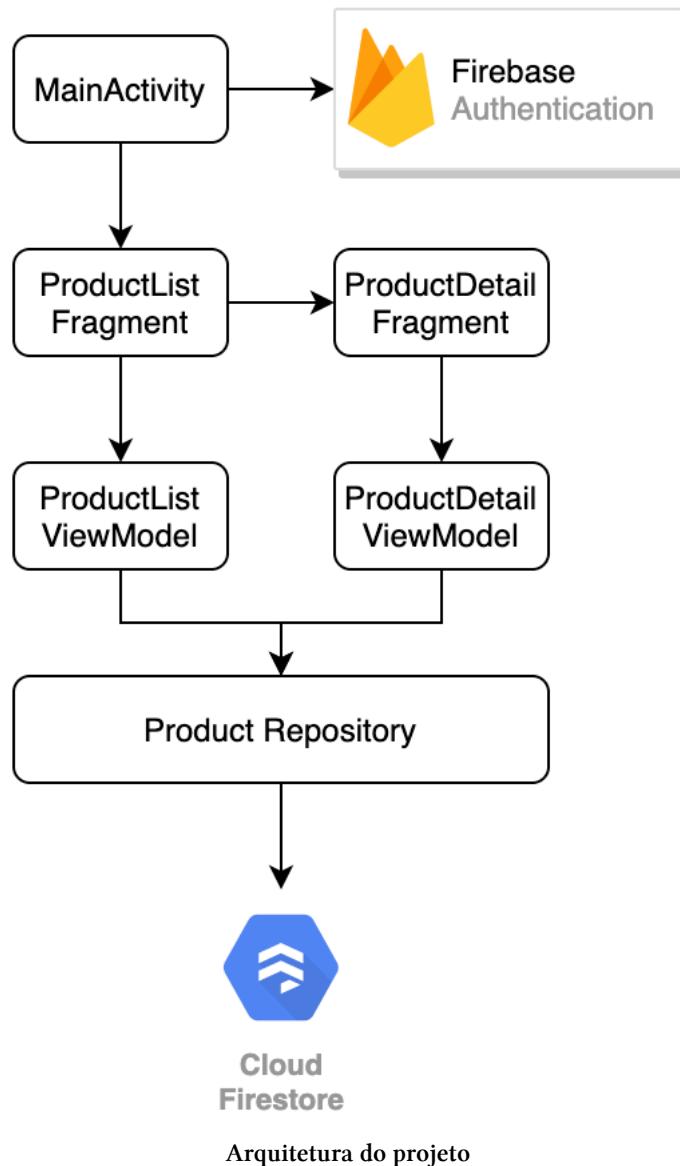


Diagrama do projeto

Com o Firebase Cloud Messaging é possível enviar notificações aos dispositivos, utilizando essa plataforma do Google.

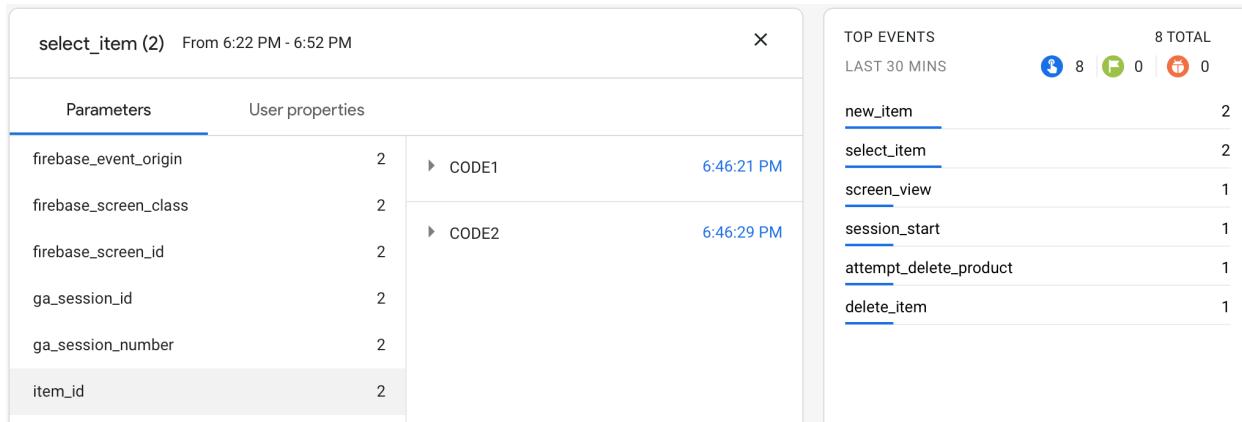
O capítulo 11 inicia um novo aplicativo para utilizar o **Firebase Authentication**, para autenticação de usuários utilizando a conta do Google, sem a necessidade da criação de infraestruturas complexas.



Arquitetura do projeto

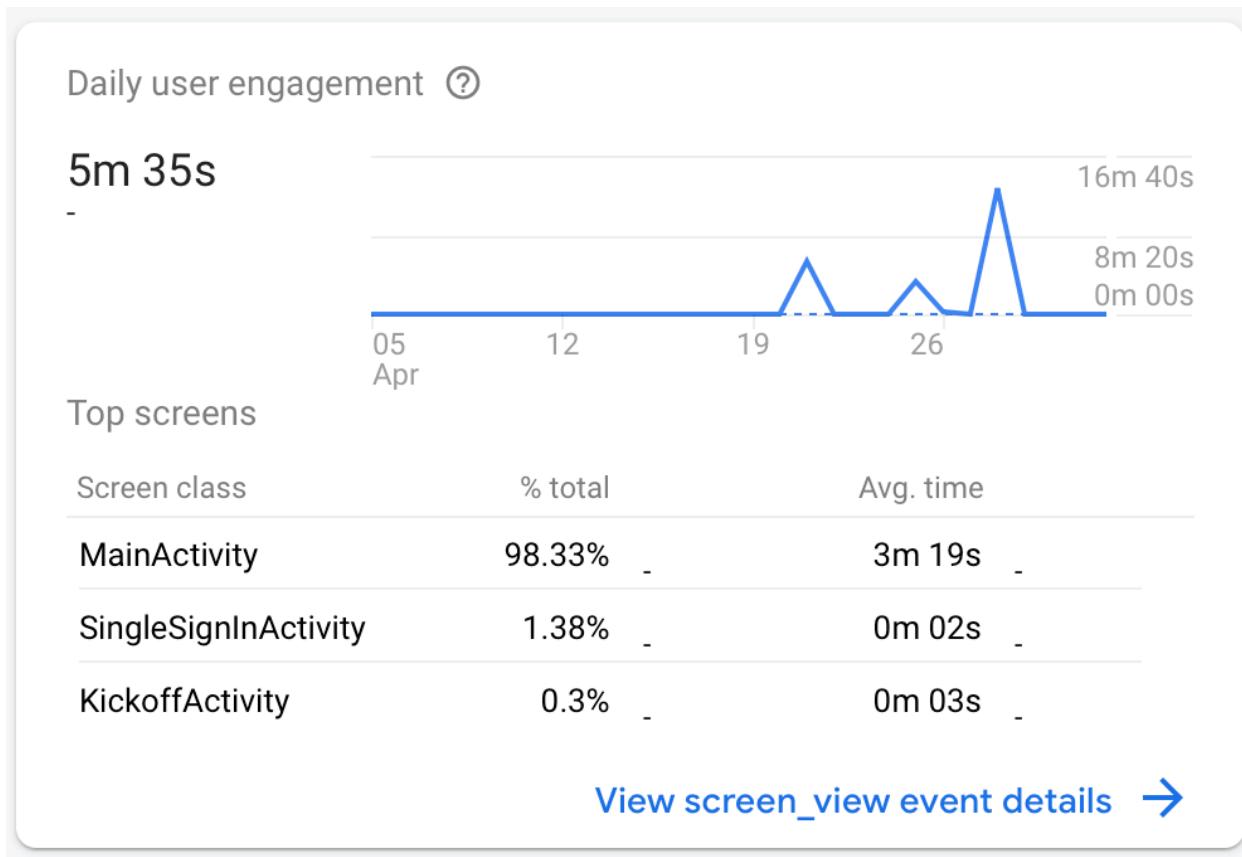
Nessa aplicação também será utilizado o **Firestore**, que é um banco de dados não-relacional na nuvem, que permite o armazenamento de coleções de documentos, com sincronismo em tempo real entre dispositivos.

O capítulo 12 utiliza uma ferramenta extremamente poderosa para o entendimento do comportamento do usuário com um aplicativo: **Firebase Analytics**. Com ele é possível gerar eventos da utilização de cada parte da aplicação, concentrando em um dashboard analítico.



### Relatório de eventos

Com o Firebase Analytics também é possível obter informações do engajamento dos usuários, bem como informações sobre localização, modelos de dispositivos e versões de sistema operacional.



### Engajamento de usuários

Finalmente o **capítulo 13** introduz o **Firebase Remote Config**, que possibilita a alteração de partes do aplicativo mediante variáveis de configuração que podem ser alteradas no console do Firebase. Isso faz com que uma série de possibilidades possam ser criadas, desde a liberação de funcionalidades

a um pequeno grupo de usuários, testes A/B, gratificações para usuários VIPs ou até mesmo a mudança do comportamento da aplicação. Tudo isso sem que haja a necessidade da publicação de uma nova versão do aplicativo.

Está pronto para embarcar nessa jornada pelo mundo do Android? Então vamos lá que o livro está repleto de novidades a serem descobertas por você!

# **2 - Sobre o sistema operacional Android**

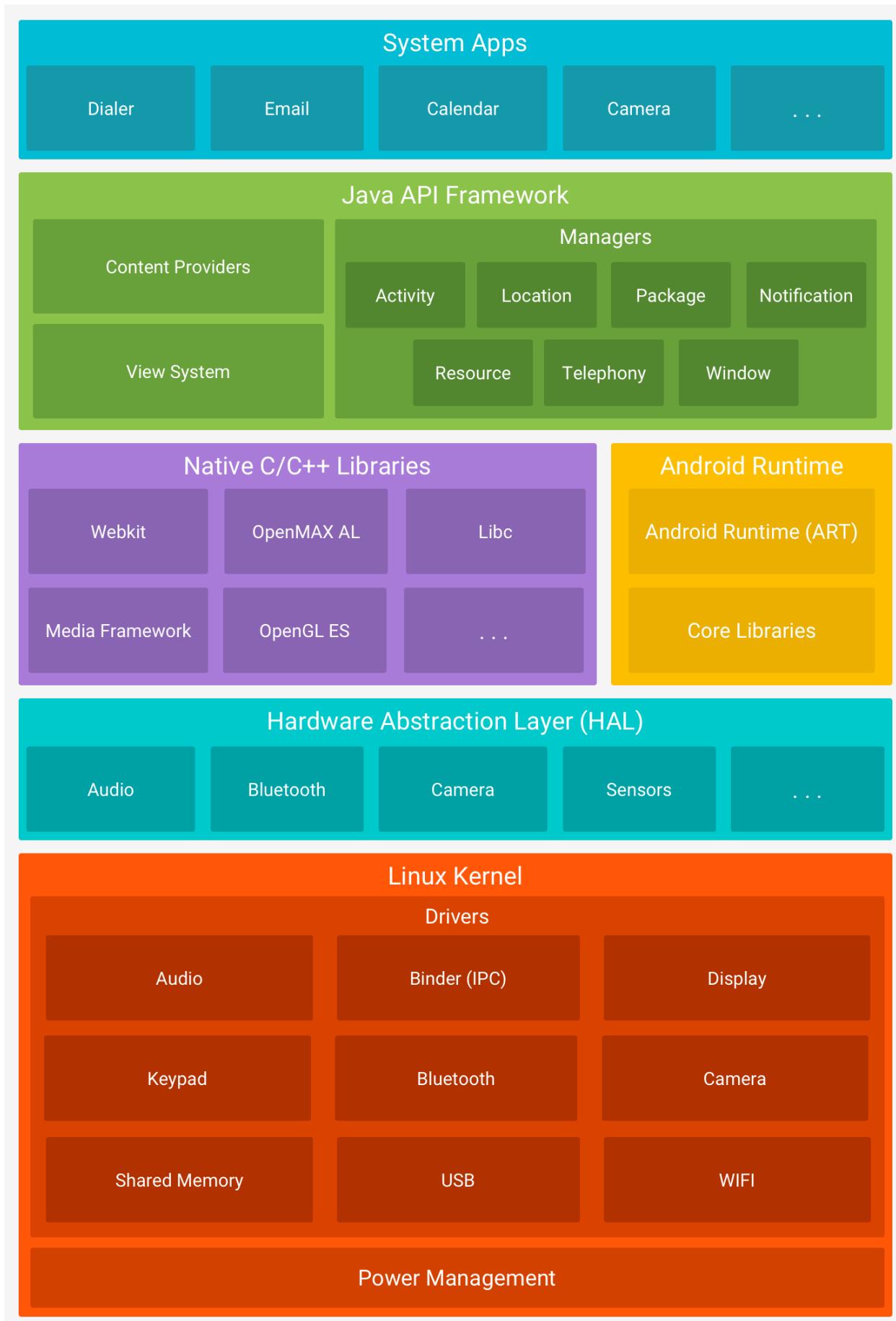
O Android é um sistema baseado em Linux, de código fonte aberto e capaz de ser utilizado em uma variedade de dispositivos, como *smartphones*, *tablets*, relógios, aparelhos de TV e de entretenimento em carros.

Ao longos dos anos o sistema Android tem se tornado complexo e cheio de funcionalidades, tanto para desenvolvedores como para usuários.

Tal complexidade traz um desafio em particular para aqueles que se aventuram no mundo do desenvolvimento de aplicativos para esse ecossistema.

## **2.1 - Um sistema feito em camadas**

O sistema Android é feito em camadas, como é exemplificado pela figura a seguir:



Felizmente, não é necessário conhecer todas as camadas, tão pouco todas as suas partes em profundidade, porém, é interessante ao desenvolvedor saber que elas existem e como estão conectadas entre si. Para aqueles que desejam se aprofundar nessa arquitetura, existe uma [documentação<sup>3</sup>](#) interessante que aborda esse assunto.

Basicamente, o conteúdo desse livro se concentra na primeira camada, chamada de **System Apps**, onde os aplicativos do próprio Android estão, como calendário, câmera e outros, e também onde estão os que o próprio usuário instala.

Para o desenvolvedor que vai criar aplicativos para o usuário final de Android, é importante ter um bom conhecimento sobre os recursos utilizados da camada seguinte, chamada **Java API Framework**, uma vez que é ela quem provê toda a interface de interação com o sistema Android. Obviamente não é necessário conhecer todas as partes dessa camada, mas sim estar a par dos recursos que serão utilizados dela.

À medida em que os conteúdos desse livro forem sendo apresentados, as partes significantes da API (Application Programming Interface) do Android serão detalhadas quando necessário.

## 2.2 - Android virtual device

Como parte das ferramentas necessárias para o desenvolvimento de aplicativos Android, que serão detalhadas no capítulo 4, está o Android virtual device, ou **AVD**, que é um dispositivo que pode ser emulado em um computador com Windows, Linux ou MacOS. Com ele é possível depurar e testar aplicativos em tempo de desenvolvimento, simulando várias situações e ambientes, sem a necessidade de um dispositivo real.

Embora essa abordagem exija um máquina de desenvolvimento mais potente, ela pode ser interessante para evitar a necessidade de vários dispositivos para testes em tempo de codificação.

## 2.3 - Como desenvolver aplicativos para o sistema Android

Existem algumas possibilidades para o desenvolvimento de aplicativos para Android, mas elas podem ser concentradas em duas grandes áreas:

a) **Desenvolvimento híbrido:** nessa modalidade é possível reaproveitar partes do projeto para criar aplicativos para outros sistemas, como o iOS. Isso traz uma agilidade maior no processo de desenvolvimento de produtos que almejam alcançar um número maior de usuários em várias plataformas. Porém, essa abordagem pode trazer complicações em aplicativos complexos ou que necessitem de interações profundas com o sistema operacional.

b) **Desenvolvimento nativo:** essa abordagem utiliza as ferramentas, linguagens e frameworks oficiais do Android, oferecidos pela própria Google. O código fonte e os projetos criados nessa

---

<sup>3</sup><https://developer.android.com/guide/platform>

modalidade não pode ser reaproveitados para outras plataformas. Essa abordagem acaba fazendo com que empresas tenham que ter equipes multidisciplinares ou até mesmo distintas para a criação de produtos para as plataformas de diferentes sistemas operacionais.



A ideia aqui não é estabelecer uma comparação entre as duas modalidades, tão pouco apontar o melhor caminho, mas sim esclarecer ao leitor as opções disponíveis.

Esse livro concentra-se na abordagem do desenvolvimento nativo, utilizando a linguagem **Kotlin** e a ferramenta **Android Studio**.

O processo de criação de uma aplicação Android é bem diferente, por exemplo, do desenvolvimento de uma aplicação Web, por isso é importante reconhecer que conhecimentos prévios da linguagem escolhida são importantes, mas, não unicamente suficientes. A razão disso é que o framework para o desenvolvimento de aplicativos Android é complexo e vasto, e muitas das vezes, diferente de outras plataformas.

### 2.3.1 - A criação da interface gráfica

Em um projeto de um aplicativo Android, a interface gráfica do usuário é criada através de arquivos XML, que podem ser editados diretamente ou com a ajuda de ferramentas gráficas no Android Studio. Versões recentes dessa IDE têm tornado esse trabalho cada vez mais eficiente, graças aos lançamentos de novos componentes e funcionalidades.

Como exemplo, segue um arquivo XML de uma tela simples de um aplicativo Android:

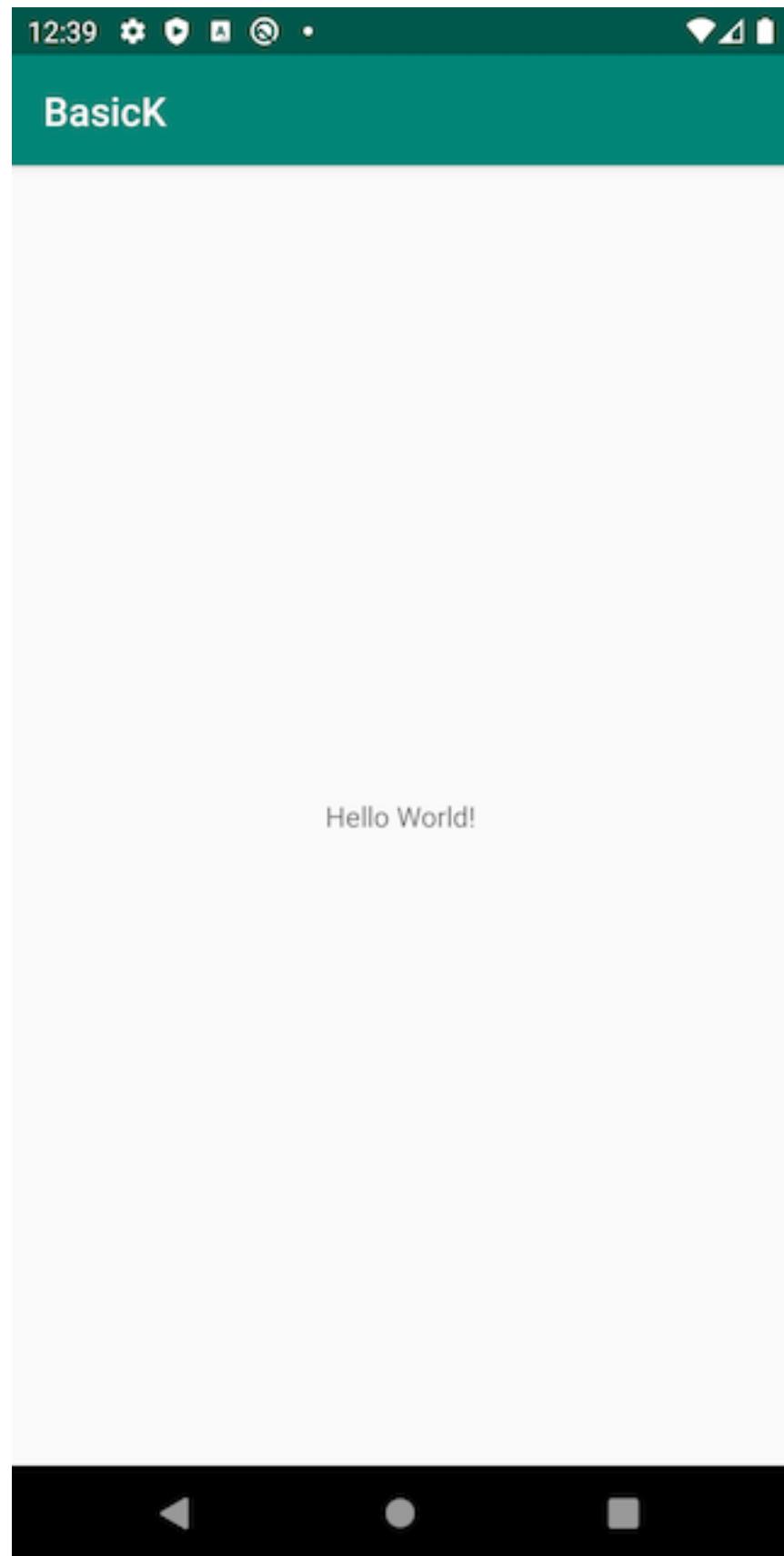
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Como pode ser observado, essa tela contém somente dois componentes: um gerenciador de layout e uma caixa de texto, com algumas propriedades pré-definidas.

Esse arquivo é responsável por criar uma tela como a da figura a seguir:



Exemplo de tela de um aplicativo

Embora simples de se construir, essa tela já está preparada para se adaptar a vários tipos de dispositivos, com diferentes tamanhos e resoluções de tela, algo que, para telas mais complexas, pode ser considerado como uma arte!

Porém, essa tela não está adaptada a dispositivos com idiomas diferentes do Inglês, mas isso será detalhado mais adiante.

No fim das contas, essa tela será representada como um **recurso de layout** dentro do projeto Android.

### 2.3.2 - A criação do comportamento

Para que a tela anterior apareça dentro de uma aplicação Android, é necessário escrever um código em Kotlin capaz de exibi-la, como no trecho a seguir:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
    }  
}
```

Essa classe, diante de algumas configurações feitas no projeto, pode ser considerada o ponto de entrada da aplicação. Ela é responsável por exibir um **recurso de layout**, que é o arquivo XML criado com a caixa de texto **Hello World!** da seção anterior.



Tecnicamente o ponto de entrada de uma aplicação Android não é esse, mas até esse momento do livro, é apropriado dizer isso do ponto de vista didático.

Aqui já é possível perceber que existem particularidades únicas do Android para a construção de aplicativos.

A partir desses dois arquivos, é possível construir o projeto para a aplicação Android, obviamente contendo outros arquivos de configuração e com uma estrutura singular.

Pelas razões apresentadas, é importante conhecer bem o framework de desenvolvimento do sistema Android, independente da abordagem escolhida: nativo ou híbrido.

## 2.4 - Conclusão

Esse capítulo deu uma brevíssima introdução ao universo do sistema Android, que como já foi dito, é vasto e complexo.

O capítulo seguinte fará algo semelhante, introduzir o leitor ao **Google Firebase**, peça essencial dos objetivos desse livro.

# 3 - Sobre o Google Firebase

O Firebase é a plataforma do Google que oferece serviços para aplicações web e para dispositivos móveis, como **Android** e **iOS**, permitindo a rápida criação de sistemas baseados em *cloud computing* sem a necessidade de se gerenciar infraestrutura de servidores ou sistemas computacionais.

A integração de uma aplicação Android com o Firebase é muito simples e pode ser feita através de bibliotecas oferecidas pela própria Google, além de algumas passos e configurações que serão detalhados no momento oportuno nesse livro.

Como um dos objetivos principais desse livro é fazer com que o leitor possa construir aplicativos Android utilizando o Firebase, é interessante conhecer alguns de seus serviços oferecidos por essa plataforma, como descritos a seguir:

## 3.1 - Firebase Cloud Messaging

Talvez esse seja um dos serviços mais conhecidos do Firebase. Com ele é possível enviar notificações para uma aplicação em um dispositivo móvel, permitindo que o usuário possa ser notificado sobre algum evento.

Esse mecanismo pode ser invocado por uma aplicação de backend ou até mesmo através de eventos automáticos ou agendados.

O Firebase Cloud Messaging se encarrega totalmente de entregar a mensagem, cuidando da garantia de entrega, bem como o armazenamento da mesma, caso o dispositivo não esteja conectado à Internet no momento da entrega da mensagem.

## 3.2 - Firebase Authentication

Em uma aplicação onde é necessário autenticar o usuário, normalmente é necessário manter uma base desses usuários e também oferecer um mecanismo onde o dispositivo móvel possa verificar as suas credenciais. Com o Firebase Authentication é possível fazer isso de forma simples e direta, tanto para o desenvolvedor, como para o usuário da aplicação.

## 3.3 - Firebase Analytics

Analisar eventos, logs e entender comportamentos de aplicativos para dispositivos móveis pode ser um desafio, uma vez que eles não estão concentrados, como em um serviço na nuvem. Para isso

existe o Firebase Analytics, um poderoso coletor, concentrador e analisador de eventos das mais diversas naturezas.

Com o Firebase Analytics, é possível gerar logs e eventos de dentro de uma aplicação Android e enviá-los para o Firebase, onde os desenvolvedores poderão analisá-los. Tal mecanismo também pode ser utilizado para enviar exceções durante a execução do código, permitindo a geração de evidências de defeitos na aplicação.

Uma das coisas interessantes desse serviço, do ponto de vista da aplicação no dispositivo móvel, é que mesmo que ele não esteja conectado à Internet no momento da geração do evento, ele será enviado para o Firebase, quando o dispositivo voltar a ficar conectado. Tudo sem a necessidade do desenvolvedor criar mecanismos complexos de armazenamento local de eventos e retentativas de entrega, ou seja, esse trabalho é totalmente feito pelo SDK do Firebase Analytics.

## 3.4 - Firebase Remote Config

Com o Firebase Remote Config é possível criar configurações para as aplicações, de tal forma que possam ser aplicadas sem a necessidade do usuário baixar uma nova versão. Tais configurações podem ser aplicadas a todos os usuários ou para segmentos que atendam algum critério de seleção, como país, gênero do usuário ou idioma.

Utilizando a interface do Firebase Remote Config, é possível alterar os valores das configurações existentes a cada aplicação. Essas são imediatamente enviadas a todos os dispositivos selecionados. Isso faz com que o comportamento da aplicação seja alterado, mediante o valor de cada configuração.

Esse recurso também pode ser utilizado para testes A/B ou mesmo para liberação de novas funcionalidades para grupos restritos de usuários.

## 3.5 - Firebase Cloud Firestore

Talvez esse seja o mais interessante e poderoso recurso do Firebase. Com ele é possível criar um banco de dados NoSQL para as aplicações, sem a necessidade de se manter um backend para prover o acesso pelos dispositivos móveis.

Com ele é possível restringir o acesso aos dados do banco através de políticas de segurança baseadas na autenticação do usuário.

Outra característica que torna esse serviço interessante, é o fato de ser considerado um banco de dados *realtime*, ou seja, caso um valor seja alterado, ele é imediatamente refletido em todos os dispositivos que estejam interessados nele.

## 3.6 - Conclusão

Esses, são apenas alguns dos serviços oferecidos pelo Firebase e que serão utilizados ao longo desse livro para criar aplicações para dispositivos Android conectadas à nuvem.

O próximo capítulo irá conduzir o leitor a preparar seu ambiente de desenvolvimento e tudo o que for necessário para começar a desenvolver sua primeira aplicação Android.

# 4 - Preparando o ambiente de desenvolvimento

Esse capítulo é dedicado à preparação do ambiente de desenvolvimento necessário para se trabalhar com Android, bem como a criação da conta do Firebase.

Esse processo pode demorar bastante, dependendo da velocidade de conexão com a Internet que a máquina de desenvolvimento possui.



Nesse [link<sup>4</sup>](#) é possível visualizar os requisitos mínimos necessários da máquina de desenvolvimento.

## 4.1 - Instalação do Android Studio

O Android Studio é a IDE oficial para o desenvolvimento de aplicações para Android. Ele pode ser baixado nesse [link<sup>5</sup>](#), de acordo com o sistema operacional, que pode ser Windows, Linux ou MacOS.

Após baixar o Android Studio, siga as instruções do processo de instalação.



Esse livro foi desenvolvido utilizando o Android Studio versão 3.5.3. Algumas telas podem ser diferentes em versões superiores.

## 4.2 - Seleção de pacotes adicionais

Depois de baixar e instalar o Android Studio, confira se os seguintes pacotes estão instalados em sua máquina, acessando o menu Tools -> SDK Manager e marcando a opção Show Package Details no canto inferior direito da janela que abrir:

<sup>4</sup><https://developer.android.com/studio>

<sup>5</sup><https://developer.android.com/studio>

	Name	API Level	Revision	Status
▼	Android 10.0 (Q)			
<input checked="" type="checkbox"/>	Android SDK Platform 29	29	4	Installed
<input checked="" type="checkbox"/>	Sources for Android 29	29	1	Installed
<input type="checkbox"/>	Intel x86 Atom System Image	29	7	Not installed
<input type="checkbox"/>	Intel x86 Atom_64 System Image	29	7	Not installed
<input type="checkbox"/>	Google APIs Intel x86 Atom System Image	29	9	Not installed
<input type="checkbox"/>	Google APIs Intel x86 Atom_64 System Image	29	9	Not installed
<input checked="" type="checkbox"/>	Google Play Intel x86 Atom System Image	29	8	Installed
<input type="checkbox"/>	Google Play Intel x86 Atom_64 System Image	29	8	Not installed

#### Pacotes necessários do SDK

A revisão pode ser diferente da imagem anterior, mas tenha certeza de que os pacotes marcados estejam instalados em sua máquina de desenvolvimento.

Ainda nessa janela, selecione a opção **SDK Tools** e instale a última versão disponível dos seguintes pacotes marcados, como mostra a figura a seguir:

	Name	Version
<input checked="" type="checkbox"/>	Android SDK Build-Tools	
<input type="checkbox"/>	GPU Debugging tools	
<input type="checkbox"/>	LLDB	
<input type="checkbox"/>	NDK (Side by side)	
<input type="checkbox"/>	CMake	
<input type="checkbox"/>	Android Auto API Simulators	1
<input type="checkbox"/>	Android Auto Desktop Head Unit emulator	1.1
<input checked="" type="checkbox"/>	Android Emulator	29.3.2
<input checked="" type="checkbox"/>	Android SDK Platform-Tools	29.0.5
<input checked="" type="checkbox"/>	Android SDK Tools	26.1.1
<input checked="" type="checkbox"/>	Documentation for Android SDK	1
<input checked="" type="checkbox"/>	Google Play APK Expansion library	1
<input checked="" type="checkbox"/>	Google Play Instant Development SDK	1.9.0
<input checked="" type="checkbox"/>	Google Play Licensing Library	1
<input checked="" type="checkbox"/>	Google Play services	49
<input checked="" type="checkbox"/>	Google Web Driver	2
<input checked="" type="checkbox"/>	Intel x86 Emulator Accelerator (HAXM installer)	7.5.1

#### Ferramentas

Novamente, a versão dos pacotes podem estar diferentes.

O Google libera atualizações constantes dos pacotes e ferramentas e o Android Studio freqüentemente sugere atualizações dos itens que estão instalados.



Caso tenha problemas em instalar o pacote Intel x86 Emulator Accelerator no Windows, consulte o seguinte [link<sup>6</sup>](#).

## 4.3 - Criação e configuração do AVD

É necessário criar um **Android Virtual Device** ou AVD para executar um dispositivo emulado do Android na máquina de desenvolvimento, para testar e executar os aplicativos que serão desenvolvidos ao longo desse livro.



Esse passo é opcional, pois também é possível utilizar um dispositivo real para testar os aplicativos.

Para criar um novo AVD, dentro do Android Studio, acesse o menu Tools -> AVD Manager. Isso fará com que uma tela apareça, listando todos os dispositivos Android virtuais presentes em sua máquina de desenvolvimento.

Nessa tela, clique no botão + Create Virtual Device.... Na tela que aparecer, selecione a opção Phone e escolha a opção **Pixel 3** ou um modelo de dispositivo mais novo, mas que tenha a opção Play Store marcada, como mostra a figura a seguir:

**Choose a device definition**

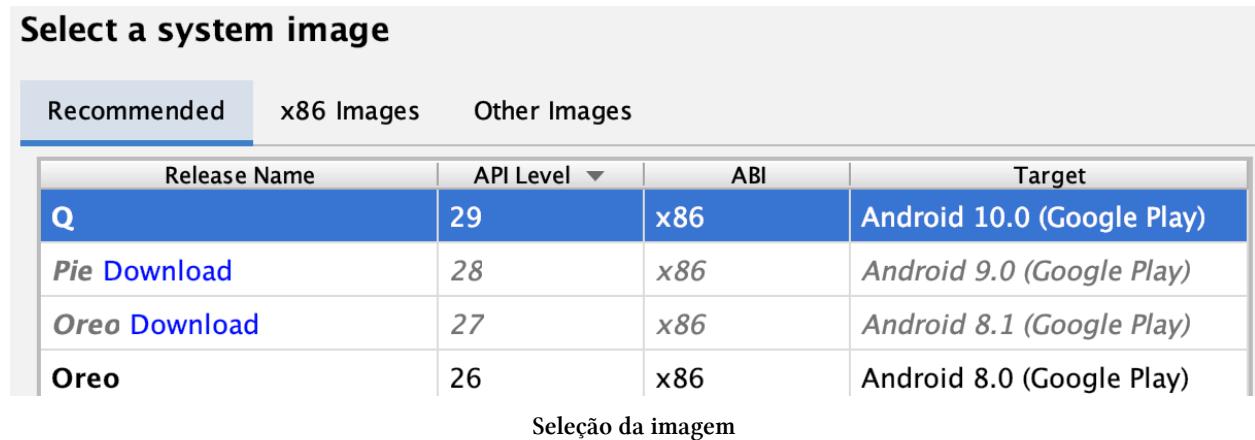
Category	Name	Play Store	Size	Resolution	Density
TV	Pixel XL		5.5"	1440x2...	560dpi
Phone	Pixel 3a XL		6.0"	1080x2...	400dpi
Wear OS	Pixel 3a	►	5.6"	1080x2...	440dpi
Tablet	Pixel 3 XL		6.3"	1440x2...	560dpi
	<b>Pixel 3</b>	►	<b>5.46"</b>	<b>1080x2...</b>	<b>440dpi</b>
	Pixel 2 XL		5.99"	1440x2...	560dpi

Escolha do dispositivo

<sup>6</sup><https://developer.android.com/studio/run/emulator-acceleration#vm-windows>

Para prosseguir, clique no botão Next.

Na tela para a escolha das imagens, selecione uma que tenha a coluna API Level com o valor 29, ou superior. Caso a imagem a ser selecionada não esteja disponível em sua máquina, clique no link para fazer download:



O nível da API se refere à versão do Android. Isso será detalhado no capítulo 6, mas a API 29 se refere ao Android 10.

Em seguida, clique em Next e configure um nome desejado para novo AVD, na última tela de configuração.



É aconselhável deixar as opções avançadas dessa tela nas configurações padrão. Altere algo apenas se tiver certeza do que está fazendo.

Para finalizar, clique no botão Finish para a conclusão da criação do novo AVD.

Ao final do processo de criação do AVD, o Android Studio voltará à tela inicial do AVD Manager e o novo dispositivo estará disponível para ser iniciado ou alterado, através dos botões localizados no canto direito dessa tela.

Execute o novo AVD para ter certeza de que ele está funcionando na sua máquina de desenvolvimento. Caso tenha problemas para executar o AVD, consulte esse [link<sup>7</sup>](#) para obter informações e respostas para seu possível problema.

É possível executar mais de um AVD ao mesmo tempo, dependendo do poder de processamento da máquina de desenvolvimento.

<sup>7</sup><https://developer.android.com/studio/run/emulator-troubleshooting>

## 4.5 - Configuração de um dispositivo real para ser utilizado durante o desenvolvimento

Também é possível habilitar um dispositivo real Android para que seja utilizado durante o processo de desenvolvimento de aplicações, ou seja, para que ele sirva como dispositivos de testes. Para isso, é necessário habilitar o **modo desenvolvedor** do aparelho. Esse [link<sup>8</sup>](#) fornece informações úteis para isso, porém, é importante frisar que as opções variam de acordo com o fabricante, por isso consulte a documentação oficial do seu aparelho antes.

Depois que o dispositivo real for configurado, ele ficará disponível para o Android Studio, quando ele for executar alguma aplicação, como será visto no capítulo 6.

## 4.5 - Criação da conta do Firebase

Para acompanhar todos os exercícios propostos nesse livro, será necessário criar uma conta no Firebase. Para isso, acesse esse [link<sup>9</sup>](#) para poder prosseguir com o processo. Você também pode utilizar sua conta do Google, caso já possua alguma.

## 4.6 - Conclusão

Esse capítulo mostrou alguns caminhos iniciais que devem ser seguidos para preparar a máquina de desenvolvimento para trabalhar com Android.

O capítulo seguinte oferece uma introdução útil à linguagem Kotlin, para servir de base teórica ao capítulo 6, onde o primeiro projeto Android será criado.

---

<sup>8</sup><https://developer.android.com/studio/debug/dev-options>

<sup>9</sup><http://console.firebaseio.google.com/>

# 5 - Um pouco sobre Kotlin

O objetivo desse capítulo não é transformar o leitor em um especialista em Kotlin, mas sim apresentar alguns conceitos importantes que serão necessários ao longo desse livro.

A linguagem Kotlin é semelhante, em alguns aspectos, à linguagem Java. Além disso, Kotlin é interoperável com Java, ou seja, em uma mesma aplicação é possível ter as duas linguagens funcionando juntas.

A documentação oficial de Kotlin é muito boa e está disponível nesse [link<sup>10</sup>](#). Além disso, também existe um curso online gratuito que pode ser acessado [aqui<sup>11</sup>](#).

Outro recurso para aprender Kotlin e praticar é esse [playground<sup>12</sup>](#) oferecido pelo seu site. Se desejar, utilize esse site para testar trechos de código desse capítulo.

Os tópicos a seguir cobrem alguns pontos básicos de Kotlin, apenas demonstrando conceitos, principalmente os que se diferem de alguma forma de outras linguagens como o Java.

## 5.1 - O básico de Kotlin

Essa seção é dedicada aos pontos mais básicos de Kotlin, como declaração de variáveis, funções e expressões interessantes com a cláusula `when`.

### 5.1.1 - Declaração de variáveis

Em Kotlin, as variáveis possuem tipos definidos, mas podem ser declaradas essencialmente de duas formas:

- **implícita:** quando o tipo é inferido pelo valor atribuído durante sua declaração

```
var x = 7  
var name = "Matilde"
```

- **explícita:** quando o tipo é definido explicitamente durante sua declaração

---

<sup>10</sup><https://kotlinlang.org/docs/reference/>

<sup>11</sup><https://www.udacity.com/course/kotlin-bootcamp-for-programmers--ud9011>

<sup>12</sup><https://play.kotlinlang.org>

```
var x: Int = 8
var name: String = "Doralice"
```

Em Kotlin, as variáveis devem possuir um valor quando são declaradas ou terem um valor atribuído antes de serem utilizadas. Obviamente, existem contornos na linguagem para isso.

Nos dois casos apresentados anteriormente, as variáveis podem ter seus valores alterados, mas também é possível declará-las de tal forma que seus valores não possam ser alterados, com a instrução `val`:

```
val y = 7
val name = "Clotilde"
```

Dessa forma, essas duas variáveis não poderão sofrer alterações dentro do escopo que pertencem.

## 5.1.2 - String templates

Em Kotlin é possível montar strings contendo valores de outras variáveis de forma muito mais intuitiva, como no exemplo a seguir:

```
var z = 3
print("The value is ${z}")
```

Nesse caso, a expressão  `${z}` será substituída no resultado final com o valor da variável `z`:

```
The value is 3
```

## 5.1.3 - Condicionais

As expressões condicionais em Kotlin possuem semelhanças com outras linguagens, como em Java:

```
val x = 3
val y = 5
if (x > y) {
    print("x is greater than y")
} else {
    print("y is greater than x")
}
```

Porém, elas podem possuir condições para intervalos, como no trecho a seguir:

```
var x = 4
if (x in 1..10) {
    print("It's in the range")
}
```

Os operadores de intervalos também podem ser utilizados em loops e em cláusulas when.

## 5.1.4 - Loops

A seguir um exemplo de uma iteração, utilizando a estrutura for para varrer uma lista de strings:

```
var dogs = listOf("Matilde", "Doralice", "Hannah", "Clotilde")
for (dog in dogs) {
    println("New dog is: ${dog}")
}
```

E aqui um outro exemplo simples utilizando a estrutura while:

```
var x = 0
while (x < 10) {
    println("Value is: ${x}")
    x++
}
```

## 5.1.5 - Funções

Funções em Kotlin possuem uma sintaxe bem diferente de Java, além de algumas vantagens interessantes, como:

- valores padrões;
- argumentos com nomes.

A seguir um exemplo de uma função simples para somar dois inteiros e retornar seu resultado:

```
fun sum(a: Int, b: Int): Int {
    val result = a + b
    return result
}
```

Ela poderia ser chamada da seguinte forma:

```
fun main() {  
    val result = sum(3, 4)  
    print("Result: ${result}")  
}
```

Na segunda linha, a função `sum` é chamada com seus parâmetros, mas ela também poderia ser invocada da seguinte forma:

```
fun main() {  
    val result = sum(a = 3, b = 4)  
    print("Result: ${result}")  
}
```

Repare que agora os parâmetros são nomeados, fazendo com que o código fique mais legível, principalmente quando se tem argumentos de tipos iguais, como é o caso.

Uma outra forma de definir os parâmetros de uma função é criando parâmetros com valores padrões, como no trecho a seguir:

```
fun sum(a: Int, b: Int = 1): Int {  
    val result = a + b  
    return result  
}
```

Dessa forma, se o segundo parâmetro for omitido em sua invocação, ela assumirá o valor `1`, nesse caso:

```
fun main() {  
    val result = sum(a = 3)  
    print("Result: ${result}")  
}
```

## 5.1.6 - Expressão When

A expressão `when` em Kotlin substitui o `switch`, presente em outras linguagens, porém, com muitas vantagens, pois ela pode ter condições variadas de testes. Além disso, seu resultado pode ser utilizado em retorno de funções e atribuições de variáveis.

```
fun testNumber(number: Int) {  
    when (number) {  
        in 1..10 -> print("Between 1 and 10")  
        11 -> print("Exactly 11")  
        else -> print("Unknown number")  
    }  
}
```

As várias formas de fazer condicionais em Kotlin podem ser utilizadas dentro da expressão when.

Como dito, o resultado da expressão when pode ser utilizado para retorno e funções ou atribuições de variáveis:

```
fun main() {  
    print(testNumber(15))  
}  
  
fun testNumber(number: Int): String {  
    return when (number) {  
        in 1..10 -> "Between 1 and 10"  
        11 -> "Exactly 11"  
        else -> "Unknown number"  
    }  
}
```

Na verdade também é possível chamar outras funções de dentro do when, como no trecho a seguir:

```
fun main() {  
    testNumber(7)  
}  
  
fun testNumber(number: Int) {  
    when (number) {  
        in 1..10 -> numberCondition1()  
        11 -> numberCondition2()  
        else -> numberCondition2()  
    }  
}  
  
fun numberCondition1() {  
    print("Between 1 and 10")  
}
```

```
fun numberCondition2() {
    print("Exactly 11")
}

fun numberCondition3() {
    print("Unknown number")
}
```

### 5.1.7 - Valores nulos e checks

Em Kotlin há uma abordagem diferente para condições para quando uma variável pode ter um valor nulo ou não. Por exemplo, o seguinte trecho de código não compila:

```
fun main() {
    var name: String = "Matilde"
    name = null //compilation error

    print("Name: ${name}")
}
```

Isso quer dizer que variáveis declaradas dessa forma não podem receber valores nulos. Porém, se houver a necessidade de uma variável receber um valor nulo, durante a execução do programa, ela deve ser declarada com essa questão de forma explícita, da seguinte forma:

```
fun main() {
    var name: String? = "Matilde"
    name = null

    print("Name: ${name}")
}
```

Repare que na declaração da variável há um sinal de ? indicando que ela pode receber nulo durante a execução do programa.

E para acessar métodos dessa variável, é necessário fazer um *null check* antes:

```
fun main() {
    var name: String? = "Matilde"

    if (name != null && name.length > 10) {
        print("Long name...")
    } else {
        print("Short name...")
    }
}
```

Uma vez que sem isso, há um erro de compilação:

```
fun main() {
    var name: String? = "Matilde"

    if (name.length > 10) { //compilation error
        print("Long name...")
    } else {
        print("Short name...")
    }
}
```

Ainda existem outras vantagens dessa abordagem, principalmente quando é necessário atribuir valores em propriedades de objetos, que podem ser nulos.

## 5.2 - Classes e objetos

Classes em Kotlin são declaradas através da palavra-chave `class`:

```
class Dog {
```

E um objeto dessa classe pode ser criado e atribuído em uma variável, como no trecho à seguir:

```
fun main() {
    var dog = Dog()
}
```

A declaração da classe também pode ser feita já com suas propriedades, como no trecho a seguir:

```
class Dog (var name: String, var age: Int, var color: String = "black") {  
}
```

Nesse caso o **construtor primário** está presente na declaração da classe, como pode ser visto logo após seu nome, com as propriedade sendo definidas.

Perceba que também é possível utilizar valores padrões para as propriedades da classe durante a criação da sua instância:

```
fun main() {  
    var dog = Dog("Matilde", 15)  
}
```

Nesse caso as propriedades `name` e `age` receberão os valores `Matilde` e `15`, que foram passados como parâmetros na criação. Já a propriedade `color` receberá o valor `black`, uma vez que ele foi omitido nessa declaração, mas que poderia ter um outro valor se ele fosse passado:

```
fun main() {  
    var dog = Dog("Clotilde", 15, "multicolored")  
}
```

A seguir serão mostrados alguns outros conceitos úteis e interessantes sobre classes em Kotlin.

## 5.2.1 - Bloco inicializador

É possível criar um código para ser executado no momento da criação de um objeto de uma classe, através do bloco `init`:

```
class Dog (var name: String, var age: Int, var color: String = "black") {  
    init {  
        println("Dog name: ${name}")  
    }  
}
```

Isso é útil uma vez que o construtor primário não pode ter nenhum código para se executado.

## 5.2.2 - Funções de classes

Obviamente uma classe em Kotlin também pode ter funções, como o exemplo a seguir:

```

class Dog (var name: String, var age: Int, var color: String = "black") {
    init {
        print("Dog name: ${name}")
    }

    fun dogType(): String {
        return when (color) {
            "black" -> "The real black dog"
            "blond" -> "A special blond dog"
            else -> "Maybe it's colored"
        }
    }
}

```

Perceba que a declaração de uma função de uma classe segue os mesmos padrões de funções comuns.

Para invocar essa função, basta fazer como no trecho a seguir:

```

fun main() {
    var dog = Dog("Clotilde", 15, "multicolored")
    println ("Dog type: ${dog.dogType()}")
}

```

Como a função retorna uma string, ela já pode ser utilizada diretamente na impressão de uma mensagem de texto.

### 5.2.3 - Herança

Para declarar uma nova classe que herda de Dog, primeiramente é necessário deixar claro que ela pode ser herdada, através do modificador open, como mostra o trecho a seguir:

```

open class Dog (var name: String, var age: Int, var color: String = "black") {

}

```

Dessa forma, é possível criar uma outra classe (BlackDog) que herda dela:

```

class BlackDog (name: String, age: Int): Dog(name, age, "black") {

}

```

Perceba que em sua declaração, os parâmetros são passados para a classe pai (Dog).

## 5.2.4 - Data classes

Para classes onde a única razão é armazenar dados, é possível utilizar *data classes* em Kotlin, como no trecho a seguir:

```
data class Cat(val name: String, val color: String)
```

Sua criação e utilização são semelhantes a classes comuns, como pode ser visto no trecho a seguir:

```
fun main() {
    val cat = Cat("Eek", "purple")
    print("Cat name: ${cat.name} - color: ${cat.color}")
}
```

## 5.3 - Conceitos avançados

Ainda existem outros conceitos de Kotlin que serão utilizados ao longo desse livro, principalmente porque muitas bibliotecas do Android fazem uso desses conceitos. Porém, eles serão introduzidos à medida em que forem necessários. Dessa forma o leitor consegue ter uma apresentação mais prática e eficiente. A seguir, alguns desses conceitos:

- inline function;
- lambda expression;
- lateinit.

Quando alguns desses recursos de Kotlin forem necessários pela primeira vez, haverá uma breve explicação no livro para que o leitor possa ter um entendimento mais amplo.

## 5.4 - Conclusão

Esse capítulo introduziu alguns conceitos básicos de Kotlin, necessários para que, a partir do próximo capítulo, o leitor possa começar a construir o primeiro aplicativo para Android utilizando essa linguagem.

# 6 - Criação do primeiro projeto

Esse capítulo é dedicado a descrever os passos para a criação do primeiro projeto em Android utilizando Kotlin, além de começar a explorar alguns conceitos básicos, como:

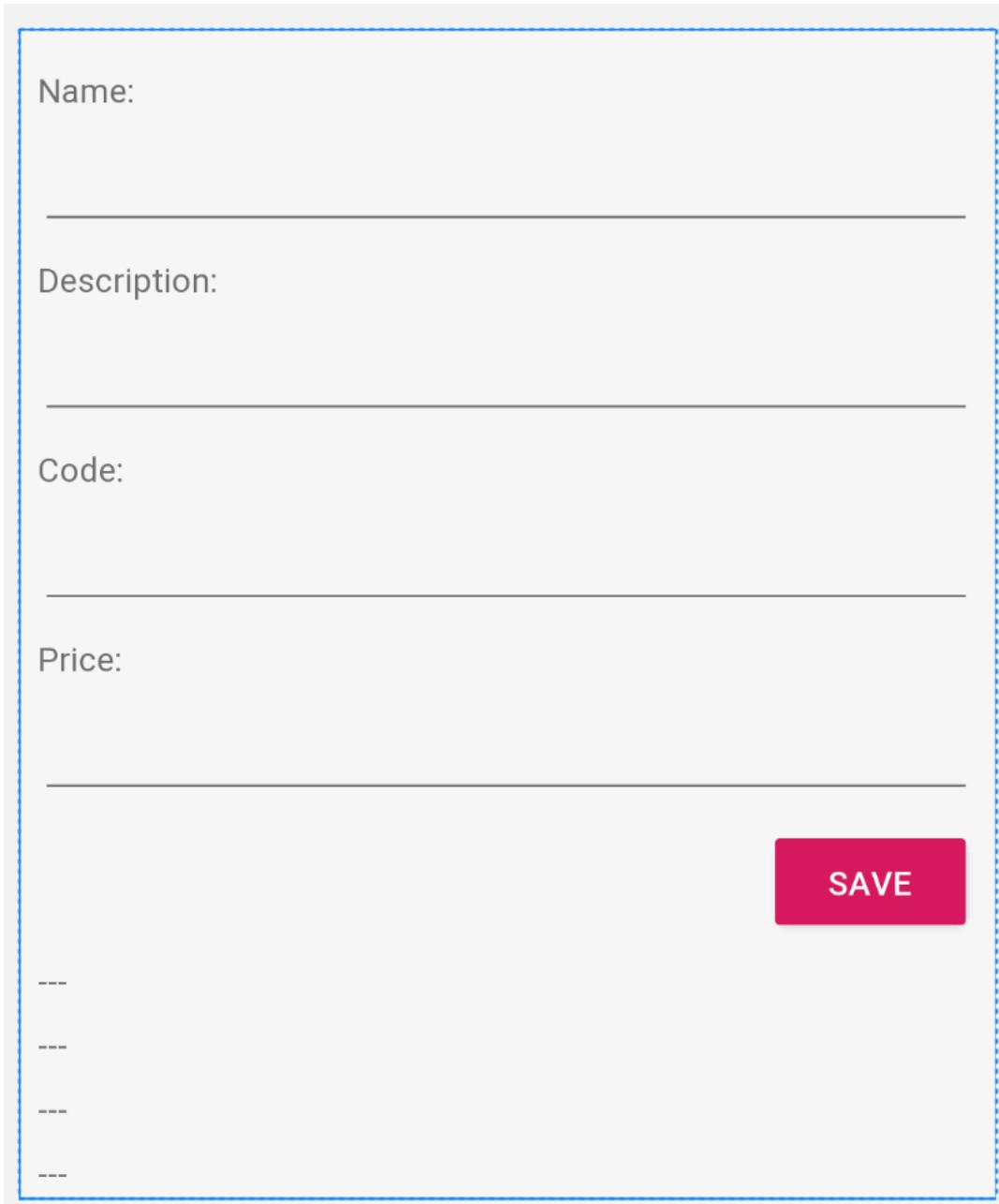
- Estrutura do projeto Android;
- Criação e conceitos iniciais da interface gráfica;
- Definição do comportamento da interface gráfica através do código em Kotlin;
- View binding.



É importante lembrar que o código do projeto completo está disponível nos extras que pode ser baixado junto com o livro.

A ideia desse capítulo é fazer com que o leitor tenha uma introdução desses conceitos para prosseguir com assuntos mais avançados. Por isso, guarde esse projeto, pois ele ainda será bastante utilizado em capítulos adiante.

O aplicativo a ser criado aqui será uma simples tela onde o usuário poderá digitar informações de um produto e, quando desejar, poderá “salvar” (que na verdade não vai salvar em nenhum lugar), exibindo suas informações em caixas de texto na mesma tela:

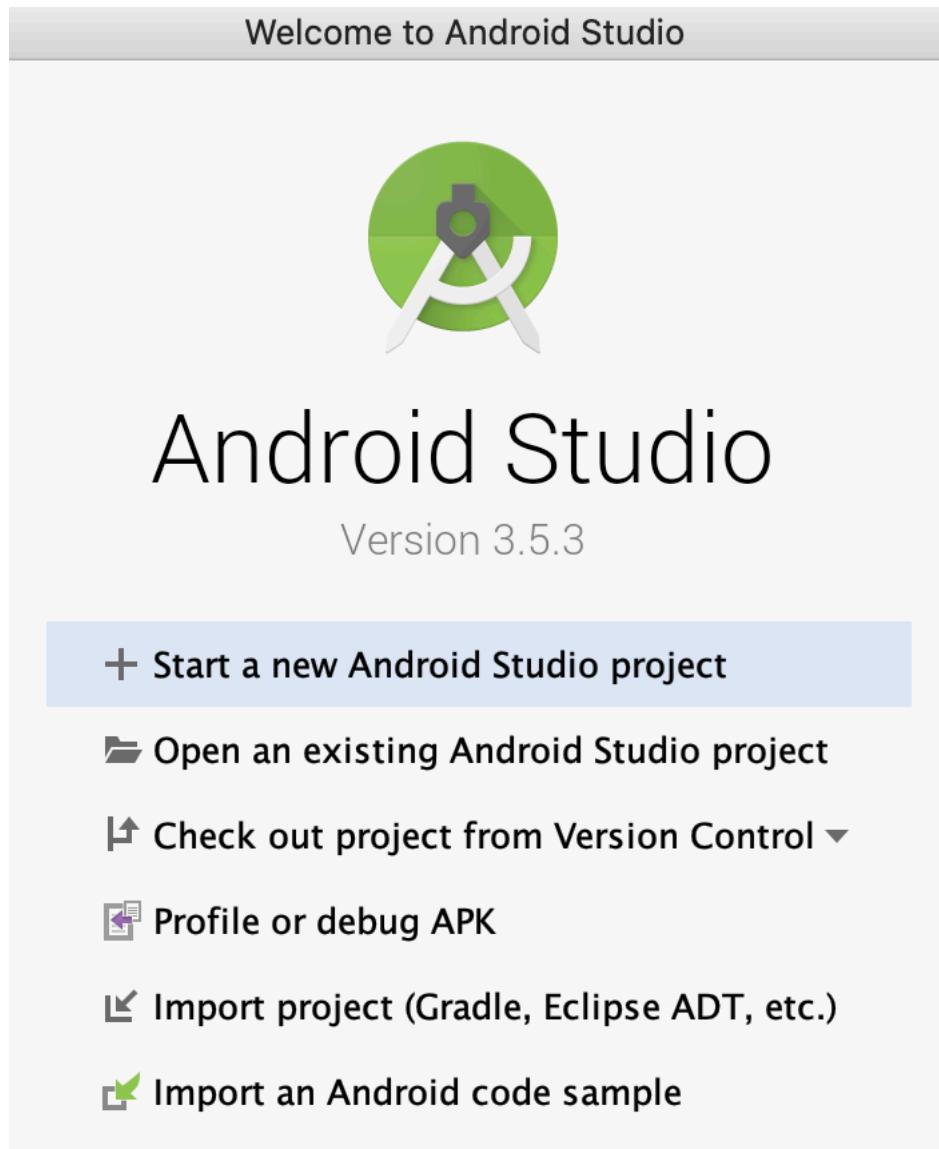


Tela do aplicativo

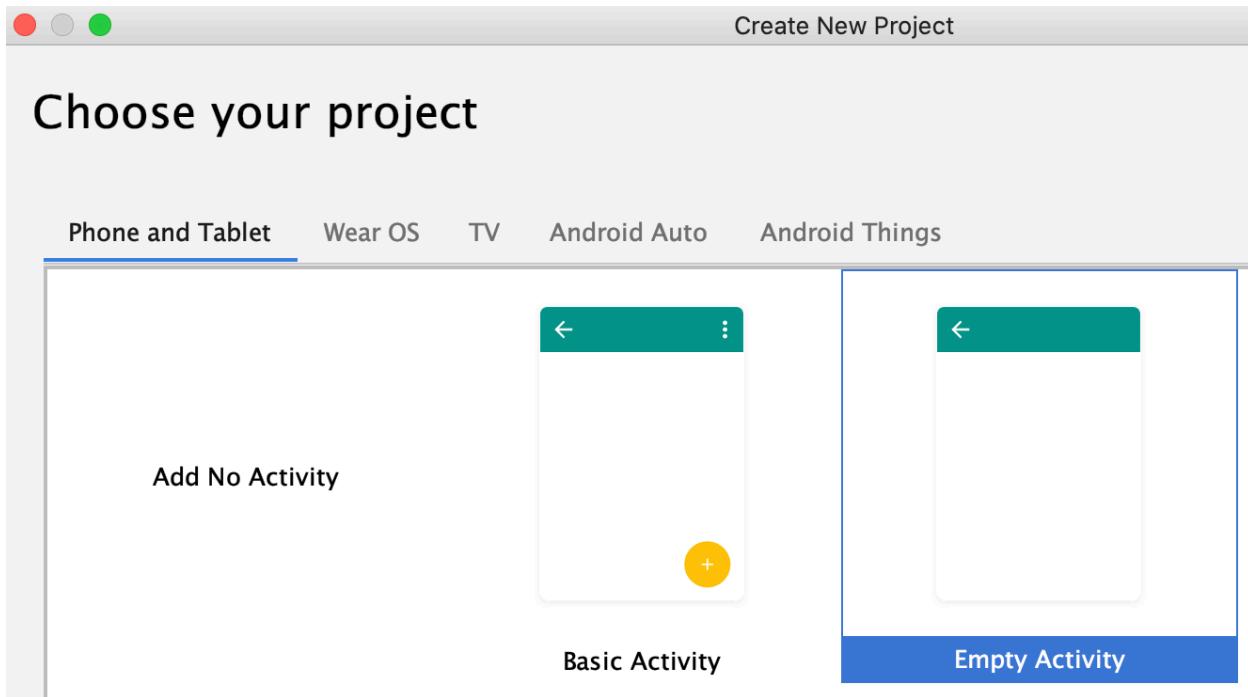
A ideia é fazer um aplicativo de lista de compras, que só pode ter um produto na lista. Obviamente, isso não tem nenhuma utilidade prática, mas tem muitos detalhes que serão utilizados para ensinar os primeiros conceitos.

## 6.1 - Criando o primeiro projeto

Para criar o primeiro projeto, abra o Android Studio e selecione a opção + Start a new Android Studio project:



Como a ideia desse projeto é apenas dar os passos iniciais, sua estrutura será a mais simples possível, por isso selecione a opção Empty Activity na tela seguinte, dentro da seção Phone and Tablet:



Selecionando a estrutura do projeto

Repare que nessa tela existem outras opções para criação de projetos mais complexos, o que facilita o trabalho inicial na construção de aplicativos com opções de navegação mais sofisticados. Porém, esse não é o caso agora, então a opção **Empty Activity** é suficiente. Em seguida, clique em **Next** para passar para a próxima tela:

**Name**

AndroidProject01

**Package name**

br.com.siecola.androidproject01

**Save location**

[Empty field] 

**Language**

Kotlin

**Minimum API level** API 21: Android 5.0 (Lollipop)

**Info** Your app will run on approximately **85.0%** of devices.  
[Help me choose](#)

This project will support instant apps

Use androidx.\* artifacts

Configuração do projeto

Nessa última tela existem configurações importantes a serem feitas:

- **Nome do projeto:** Esse é o nome que vai aparecer para o usuário quando ele instalar o aplicativo em seu dispositivo;
- **Nome do pacote:** O nome do pacote deve ser único para ser publicado na loja do Google, por isso é interessante colocar seu site em notação reversa, seguido do nome do projeto. Porém, para facilitar a condução dos exercícios nesse livro, mantenha o mesmo nome que foi configurado na figura anterior;
- **Local a ser salvo:** Escolha um local para salvar o projeto na sua máquina de desenvolvimento;
- **Linguagem:** Escolha a linguagem **Kotlin**;

- **Nível mínimo da API:** Essa é uma configuração importante, que define a versão mínima do Android em que o aplicativo irá rodar. Para esse exemplo, escolha a mesma opção da figura anterior.



Esse livro foi desenvolvido utilizando o Android Studio versão 3.5.3. Algumas telas podem ser diferentes em versões superiores.

O nível mínimo da API pode ser alterado posteriormente dentro do projeto, porém é importante saber escolher esse nível, pois, ela definirá a versão mínima do sistema operacional que o aplicativo poderá ser executado. O Android Studio oferece uma estatística da porcentagem de dispositivos atuais capazes de rodar o aplicativo em cada nível de API. Para saber mais sobre qual API corresponde a cada versão do Android, consulte esse [link<sup>13</sup>](#).

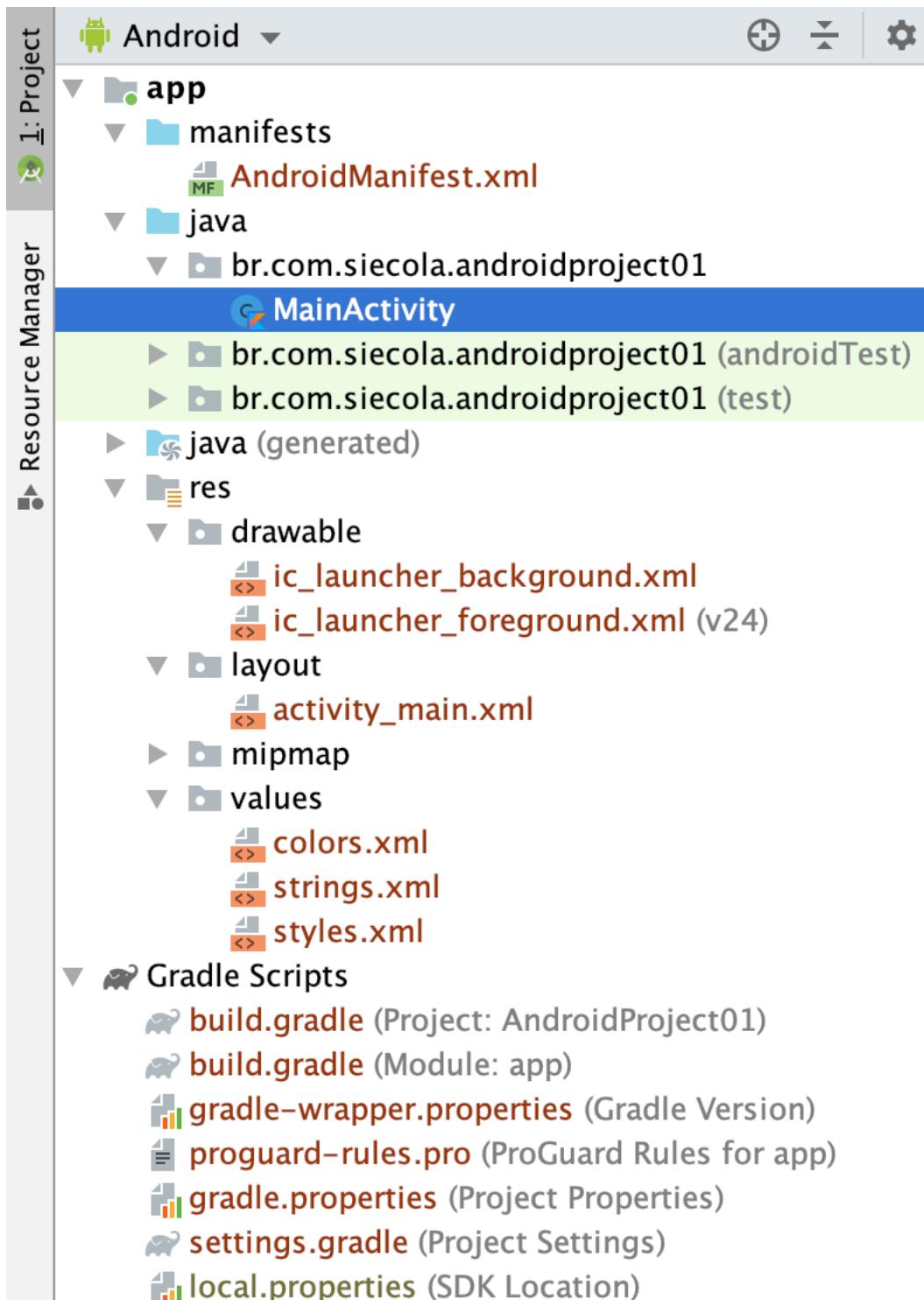
Finalize a configuração e peça ao Android Studio para criar o projeto. Após algum tempo, ele estará aberto e pronto para ser trabalhado!

## 6.2 - Estrutura do projeto

Depois que o projeto é criado, é possível observar sua estrutura no Android Studio, na aba lateral esquerda, chamada Project:

---

<sup>13</sup><https://source.android.com/setup/start/build-numbers>



Estrutura do projeto

Essa é a estrutura básica de um projeto Android. A seguir, algumas informações sobre essa estrutura.

a) Arquivo `AndroidManifest.xml`: esse arquivo define permissões que o aplicativo necessita para

ser executado além de instruções ao sistema Android. Nesse primeiro projeto não será necessário alterá-lo, mas no próximo projeto sim, pois, será necessário solicitar permissão de acesso à Internet;

**b) Classe MainActivity:** essa é a classe em Kotlin onde será criado o comportamento do aplicativo;

**c) Pasta de recursos:** imagens, strings, layouts e valores são considerados recursos em projeto Android, e ficam nessa pasta;

**d) Scripts do Gradle:** aqui estão definidos os arquivos para que o projeto seja compilado utilizando Gradle. A seção seguinte traz mais detalhes sobre esse tópico.

A pasta `res` está dividida pelo tipo do recurso e a o arquivo `activity_main.xml` é o que define a interface gráfica do usuário. Ele é um arquivo no formato XML onde os componentes são criados, configurados e ajustados no layout da tela. A seção 6.4 adiante traz mais detalhes sobre esse arquivo.

## 6.3 - Arquivo build.gradle

O projeto Android utiliza o Gradle como gerenciador de dependências, que são as bibliotecas que podem ser adicionadas para desempenhar diversas funções.

Na pasta `Gradle scripts` existem dois arquivos `build.gradle`: um para todo o projeto e outro para o módulo `app`. Isso faz com que o projeto esteja preparado para múltiplos módulos, que não é alvo desse livro. Para saber mais sobre módulos, consulte esse [link<sup>14</sup>](#).

O arquivo `build.gradle` (`Module: app`) é o que define as configurações do módulo `app`, que nesse caso concentra todo o aplicativo a ser desenvolvido. Nele é possível distinguir 3 seções até o momento:

**a) Plugins:**

Nessa seção são definidos os plugins necessários para a compilação do projeto. A seguir estão os que foram criados pelo Android Studio:

```
apply plugin: 'com.android.application'  
apply plugin: 'kotlin-android'  
apply plugin: 'kotlin-android-extensions'
```

Repare que existem dois plugins para o Kotlin, para que o projeto tenha suporte a essa linguagem.

**b) Configurações de compilação:**

Essa seção foi parcialmente definida durante a criação do projeto, quando a versão mínima da API foi definida:

---

<sup>14</sup><https://developer.android.com/studio/projects>

```
android {  
    compileSdkVersion 29  
    buildToolsVersion "29.0.3"  
    defaultConfig {  
        applicationId "br.com.siecola.androidproject01"  
        minSdkVersion 21  
        targetSdkVersion 29  
        versionCode 1  
        versionName "1.0"  
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
    }  
    buildTypes {  
        release {  
            minifyEnabled false  
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), '\\"  
            proguard-rules.pro'  
        }  
    }  
}
```

Aqui vale a pena ressaltar as seguintes configurações:

- **applicationId**: essa é a identificação única do aplicativo. Esse valor será utilizado caso ele seja publicado na loja do Google;
- **compileSdkVersion**: esse valor especifica o nível da API em que o projeto será compilado;
- **minSdkVersion**: define o nível mínimo da API que o aplicativo poderá ser executado, que consequentemente define a versão mínima do Android;
- **targetSdkVersion**: esse é o valor que indica a versão da API que o aplicativo será testado.

Algumas configurações ainda serão feitas nessa seção para adicionar funcionalidades ao processo de compilação e desenvolvimento do projeto.

### c) Dependências:

Nessa seção são colocadas as dependências de bibliotecas que o projeto necessita:

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.1.0'  
    implementation 'androidx.core:core-ktx:1.1.0'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'  
}
```

As versões devem ser muito bem escolhidas para não haver incompatibilidade entre elas. Uma dependência que vale a pena ser ressaltada nessa configuração inicial é a biblioteca ConstraintLayout, que facilita muito o desenvolvimento de interfaces gráficas.



De preferência, utilize as versões listadas no trecho anterior.

Quando novas bibliotecas forem necessárias ao projeto, sejam do próprio Google ou de terceiros, elas deverão ser adicionadas aqui para serem utilizadas pelo projeto.

Para informações mais detalhadas sobre o arquivo `build.gradle`, consulte a documentação oficial nesse [link<sup>15</sup>](#).

## 6.4 - Criando a primeira interface gráfica

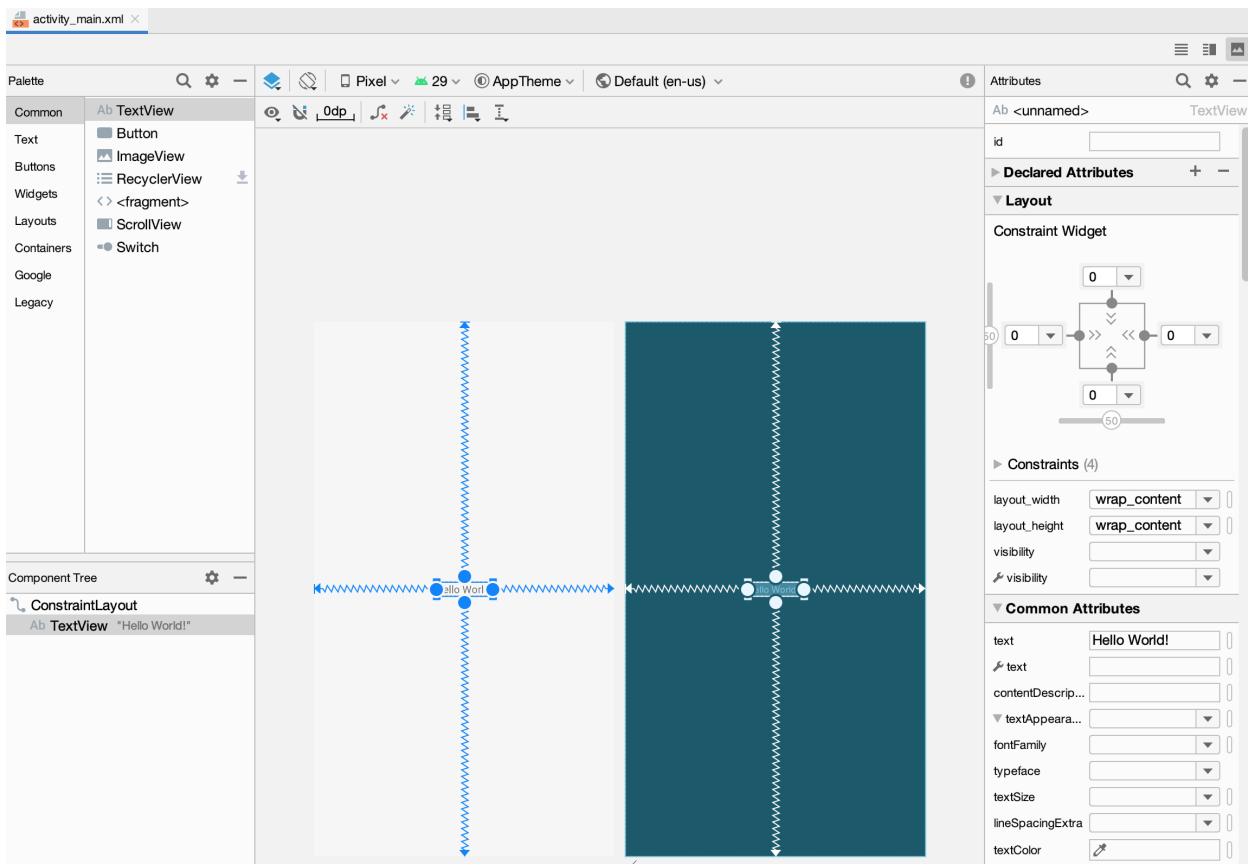
Como dito anteriormente, as interfaces gráficas em projeto Android são definidas em arquivos XML. O projeto criado possui apenas uma tela, que está no arquivo `res\layout\activity_main.xml`.

Ao abrir esse arquivo é possível ter três visões, controladas por abas que estão localizadas na parte superior direita da tela:

- **Code:** nessa visão é possível editar o arquivo XML e ter uma pré-visualização ao mesmo tempo;
- **Split:** aqui é possível visualizar o código e o design da tela ao mesmo tempo;
- **Design:** nessa aba é possível visualizar e editar a interface gráfica utilizando os recursos do Android Studio.

Ao selecionar a aba Design, é possível ter uma visualização como a da figura a seguir:

<sup>15</sup><https://developer.android.com/studio/build>



Design da tela

À esquerda estão a paleta de componentes gráficos, que podem ser arrastados para dentro da tela e sua árvore de componentes. No centro, está a visualização da tela, bem como um *blue print*, onde configurações de layout e espaçamento podem ser observados. Por fim, do lado direito são exibidos os atributos do componente selecionado, bem como as suas configurações de posicionamento na tela.

No Android, é possível criar telas constituídas de vários tipos de layouts, que permitem a organização dos componentes em forma de listas horizontais, listas verticais, grades com colunas e linhas e também com posicionamento relativo entre os componentes.

A tela criada pelo Android Studio possui um layout chamado `ConstraintLayout`, como pode ser observado dentro do arquivo XML, na aba Text da figura anterior:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```



Não se preocupe se existem muitas coisas novas nesse arquivo. Os conceitos ainda serão apresentados.

Esse tipo de layout permite que os componentes fiquem organizados na tela de forma relativa a outros componentes.

Nesse caso, há apenas um componente do tipo `TextView`, que está posicionado no centro de seu pai, que é a tela inteira. Isso permite que ele sempre fique nessa posição, independente do tamanho da tela ou de sua orientação durante a execução da aplicação. Os atributos que definem esse comportamento são os que estão definidos com os nomes iniciando com `app:layout_constraint`.

No Android, todos os componentes gráficos são considerados algum tipo de `View`. Nesse caso, `TextView` é uma `View` especializada em exibição de textos.

Ainda no componente `TextView`, existem 3 outras propriedades importantes:

- **`layout_width`**: isso define a largura máxima que o componente irá utilizar da tela. A opção `wrap_content` diz que ele não deve ocupar mais do que o necessário para exibir o conteúdo;
- **`layout_height`**: essa propriedade define a altura máxima que o componente deve ocupar na tela. A opção `wrap_content` diz que tal altura deve ser o suficiente para exibir o conteúdo;
- **`text`**: esse é o valor que será exibido na tela para o usuário.

Ainda existem outras propriedades que um `TextView` pode assumir, mas que não fazem parte dessa primeira `View`.

Todas as propriedades dos componentes, bem como sua organização e criação, podem ser feitas tanto diretamente no arquivo XML, quanto na interface gráfica, na aba Design. O desenvolvedor tem a liberdade de escolher a forma que preferir.

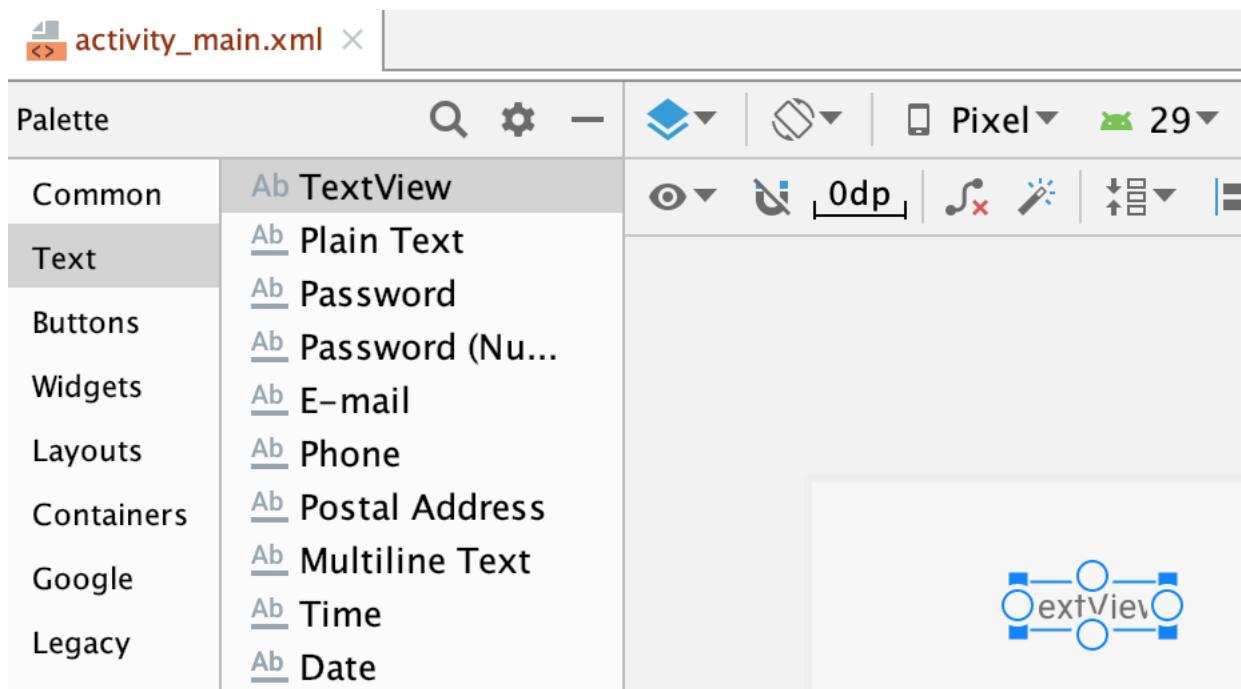
Como o objetivo aqui é desenvolver uma tela, como a do início do capítulo, serão necessários os seguintes componentes:

- 8 TextView;
- 4 EditText;
- 1 Button.

Esses componentes devem ser agrupados de tal forma a criar uma interface gráfica capaz de se adaptar a qualquer tamanho de tela e orientação. E é aí que o ConstraintLayout pode ser utilizado para facilitar esse trabalho.

Para começar, mude para a aba Design e apague o TextView que está na tela.

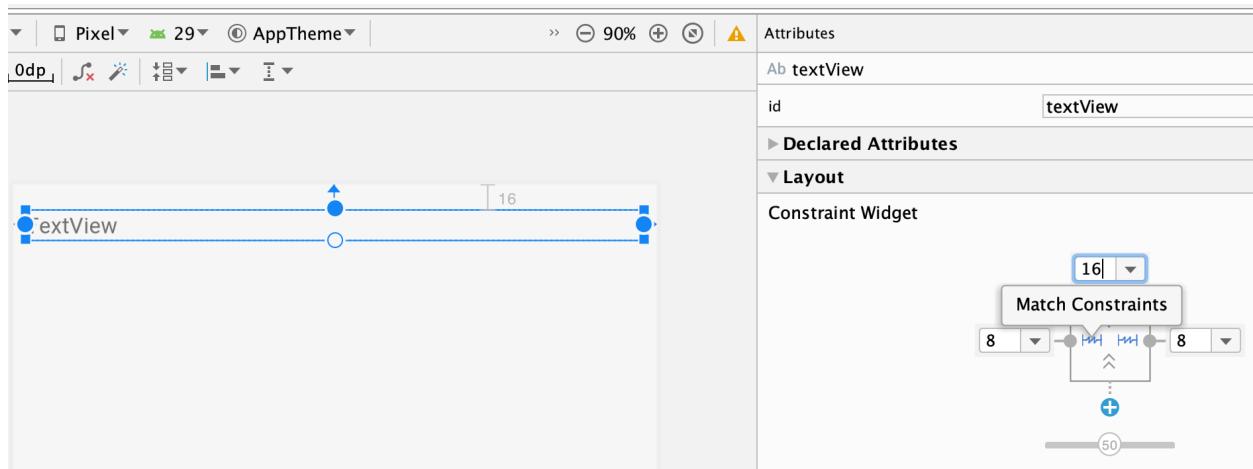
Utilizando a paleta de componentes do lado esquerdo, escolha o componente TextView e coloque próximo ao topo da tela. Repare que quando o componente é selecionado, aparecem 4 círculos, como na figura a seguir:



Selecionando o TextView

Esses círculos são utilizados para posicionar o componente em relação a outros componentes ou às bordas da tela, quando se constroi uma interface com base no ConstraintLayout.

Conecte os círculos superior, direito e esquerdo às bordas da tela:



Constraints do TextView

Porém, quando as ligações são feitas com as bordas da tela, o Android assume valores padrões para os atributos dessas ligações, que não são as desejadas para esse componente. Para isso, mantendo o TextView selecionado, vá até a aba Attributes, no canto direito e configure a seção Layout com as seguintes características:

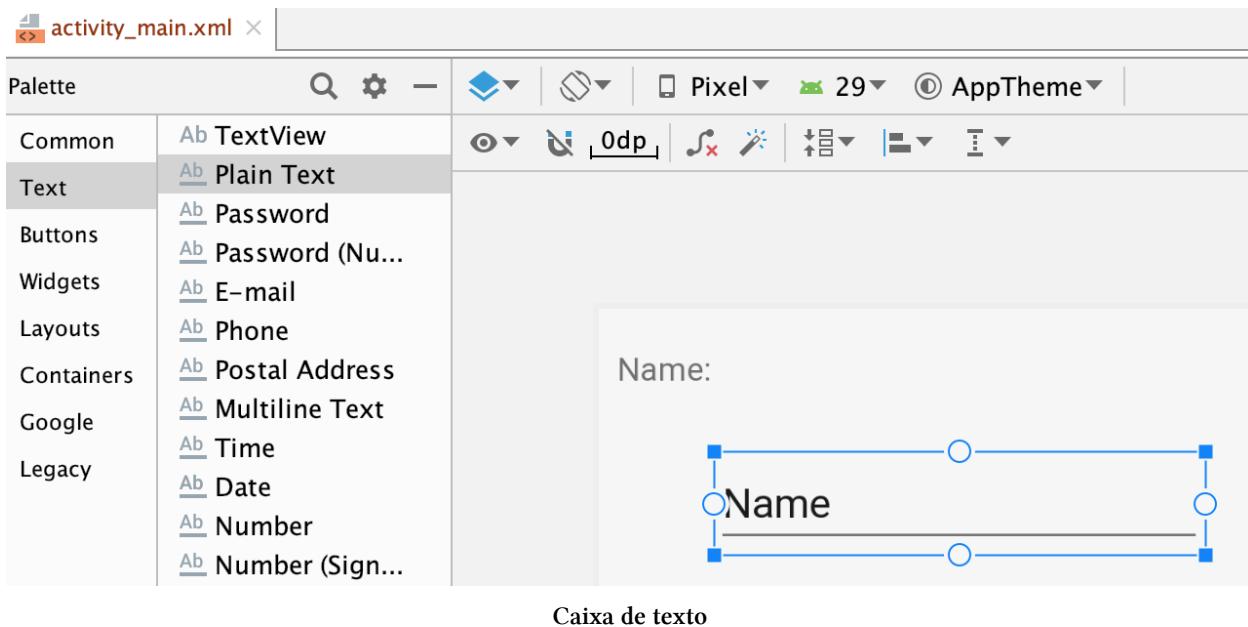
- Espaçamento superior com o valor 16;
- Espaçamentos laterais com valor 8;
- Mude o tipo do espaço lateral para Match Constraints. Isso pode ser feito clicando-se no desenho marcado em azul na figura anterior, entre os dois *combo boxes*.

Essas configurações farão com que o componente fique afastado de 16 dp do topo, bem como 8 dp das margens laterais. Além disso, o componente irá se adaptar às variações de largura, de acordo com o tamanho da tela ou sua orientação.

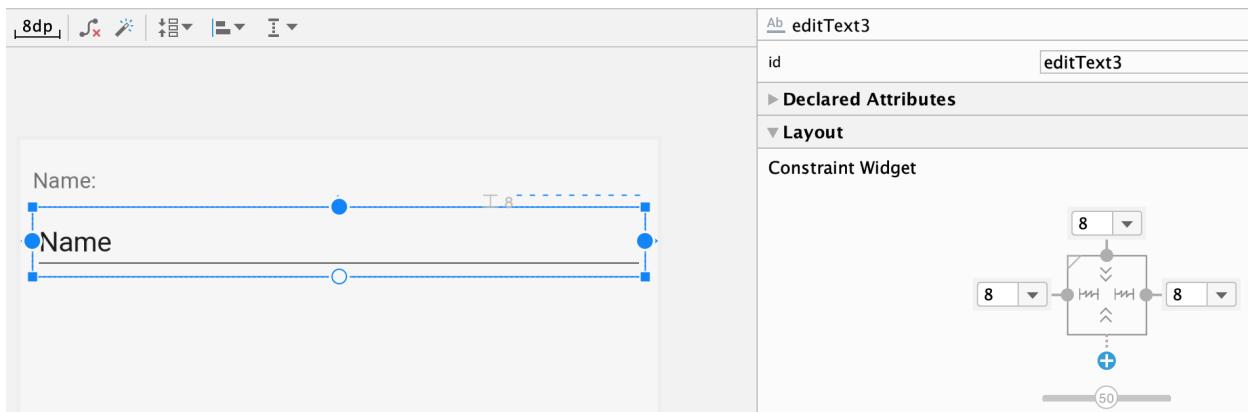
Ainda com o TextView selecionado, altere o atributo text, localizado na seção Common Attributes na aba Attributes para o valor Name: . Esse é o valor que será exibido para o usuário na interface gráfica.

Como esse componente não será referenciado pelo código Kotlin, não é necessário definir um identificação única para ele, além da que o próprio Android Studio já definiu.

O próximo componente a ser posicionado na tela é uma caixa de texto que poderá ser editada pelo usuário. Esse componente chama-se EditText , mas aparece com o nome Plain Text na seção Text da paleta de componentes. Selecione-o e arraste-o para logo abaixo do TextView:



Conekte os círculos laterais às bordas da tela, da mesma forma como foi feito com o TextView. Em seguida, coneke o círculo de sua parte superior ao círculo da parte inferior do TextView:



Da mesma forma como foi feito com o TextView, o novo EditText terá sua largura adaptável ao tamanho da tela e sua orientação. Porém, ele sempre estará próximo ao TextView, fazendo que a interface gráfica se mantenha coerente independentemente dessas mudanças.

Existem apenas mais duas configurações a serem feitas nesse EditText:

- Remova o valor do atributo text, que está o valor Name. Esse é o local onde o usuário irá digitar o nome do produto, por isso deve estar vazio desde o início;
- Configure o atributo id, localizado no topo da aba Attributes, para o valor edtName. Esse será o valor no qual esse componente poderá ser referenciado de dentro do código Kotlin.

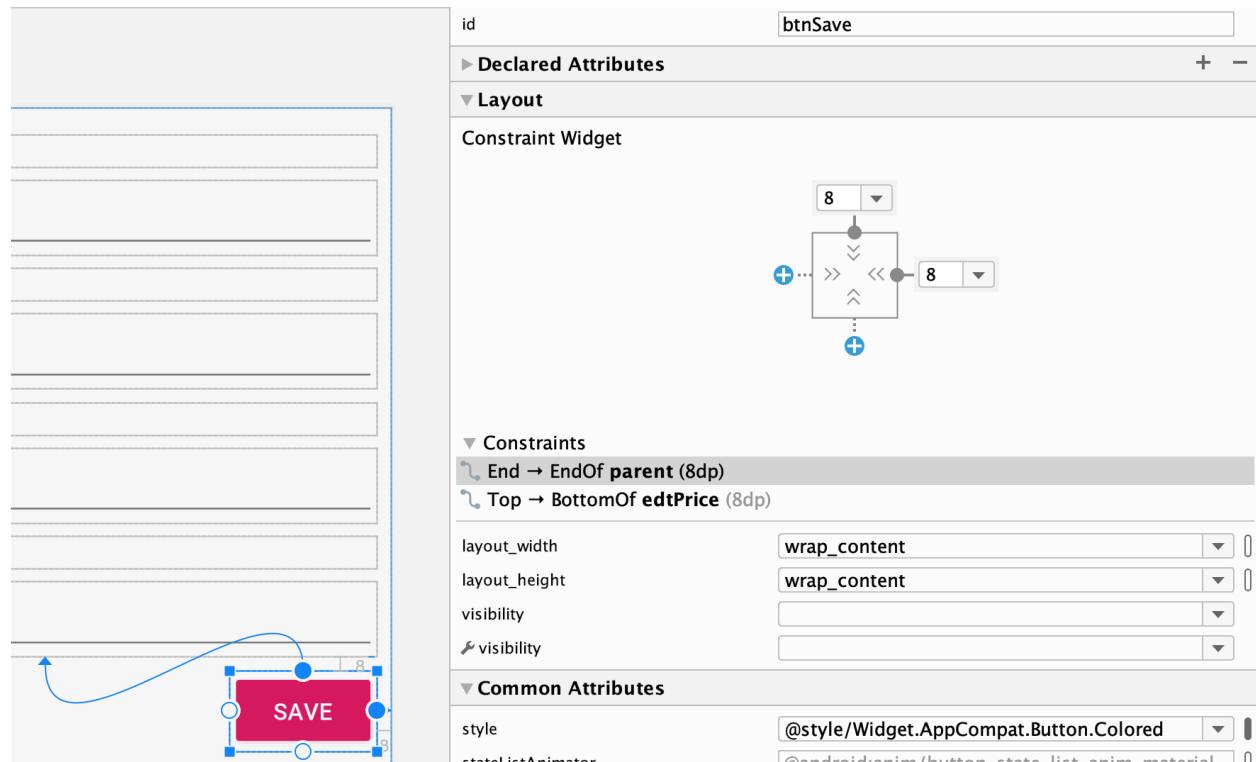
A ideia é ler o valor que o usuário digitou nesse campo, por isso ele deve possuir uma identificação única para ser localizado de dentro do código Kotlin.

Acrescente mais 3 conjuntos de `TextView` e `EditText` para representar os campos de descrição, código e preço do produto. Configure o atributo `id` dos componentes `EditText` seguintes com os seguintes valores:

- Descrição do produto: `edtDescription`
- Código do produto: `edtCode`
- Preço do produto: `edtPrice`

O componente para edição do preço do produto merece uma atenção especial, afinal, seria interessante exibir um teclado diferente ao usuário, uma vez que ele irá digitar um valor decimal. Isso pode ser feito configurando o atributo `inputType` para o valor `numberDecimal`. Isso fará com que o usuário possa digitar apenas números decimais para o preço, ao invés de letras, o que faz todo o sentido.

Logo abaixo do último `EditText`, que é do preço, posicione um botão, como na figura a seguir:



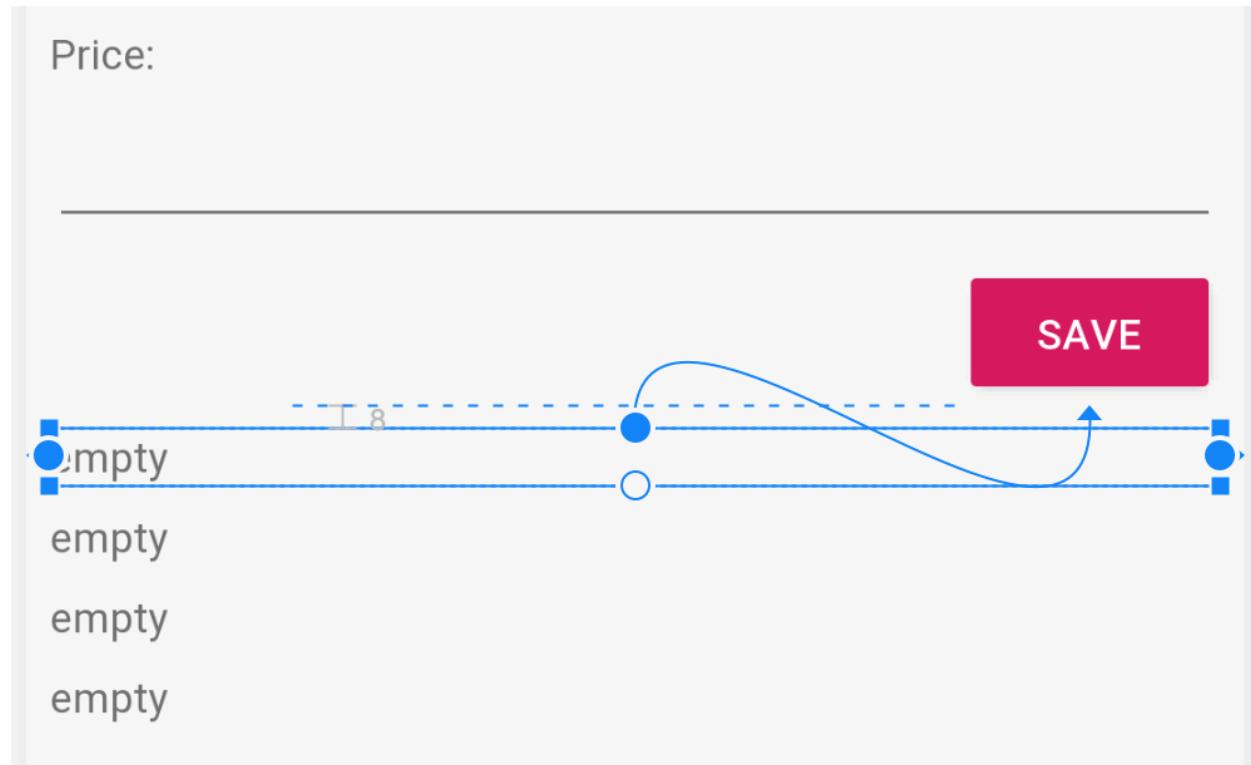
Posicionando o botão

Repare que o botão está conectado à última caixa de texto e à borda da tela, como pode ser observado na seção `Layout` da seção `Attributes`. Veja também seu `id` foi definido para `btnSave`, pois será necessário referenciá-lo no código Kotlin.

Para deixar o botão com esse estilo, basta alterar o atributo `style` para o valor `@style/Widget.AppCompat.Button.Colored`. Esse é um dos estilos pré-definidos do Android.

Além disso, configure também seu texto para Save.

Agora posicione mais 4 `TextView` logo após o botão. Eles servirão para exibir os detalhes do produto, depois que o usuário pressionar o botão Save. Veja como deve ficar:



Coloque um valor fixo para o atributo `text` desses novos `TextView`. Isso dará um efeito didático para a demonstração do funcionamento do aplicativo, da forma como ele será construído.

Defina um `id` para cada desses novos 4 `TextView`:

- Nome do produto: `txtName`
- Descrição do produto: `txtDescription`
- Código do produto: `txtCode`
- Preço do produto: `txtPrice`

Definir uma identificação única para esses novos 4 `TextView` é essencial, pois eles terão seus valores modificados pelo código Kotlin, quando o usuário clicar no botão Save.

Ainda existem algumas configurações necessárias para essa tela, que são:

- Colocar todos os componentes dentro de um outro componente capaz de fazer a tela rolar, caso ela seja exibida em um dispositivo pequeno ou quando simplesmente não houver espaço para mostrar todos os componentes;
- Transformar as strings que aparecem na tela, em recursos capazes para facilitar a tradução.



Essa última ação é algo que deve ser feito desde o início do desenvolvimento de um aplicativo, ainda que a ideia de torná-lo disponível em outros idiomas esteja distante.

O componente capaz de fazer a tela rolar chama-se ScrollView, e deve englobar todos os elementos da tela, inclusive o componente do tipo ConstraintLayout. Para fazer isso, estando com o arquivo `activity_main.xml` aberto, mude para a aba Text, localizada na parte inferior da tela, para poder alterar o arquivo XML manualmente.

Agora vá até o início do arquivo e altere-o para ficar como o trecho a seguir:

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

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        tools:context=".MainActivity">
```

Da mesma forma, altere o fim do arquivo para finalizar as tags XML corretamente para esses dois componentes:

```
</androidx.constraintlayout.widget.ConstraintLayout>
</ScrollView>
```

Lembre-se de ajustar a tabulação do arquivo o final das alterações.

Essa modificação faz com que todos os componentes da tela, incluindo o gerenciador de layouts ConstraintLayout fique dentro de ScrollView, tornando a tela capaz de rolar caso não haja espaço para mostrar todos os componentes.

A segunda alteração que deve ser feita no arquivo XML, é a extração das strings que estão fixas no código, tornando-as recursos que podem ser traduzidos para outros idiomas. Perceba que o Android Studio ressalta essas strings no arquivo `activity_main.xml`, como mostra a figura a seguir:

```

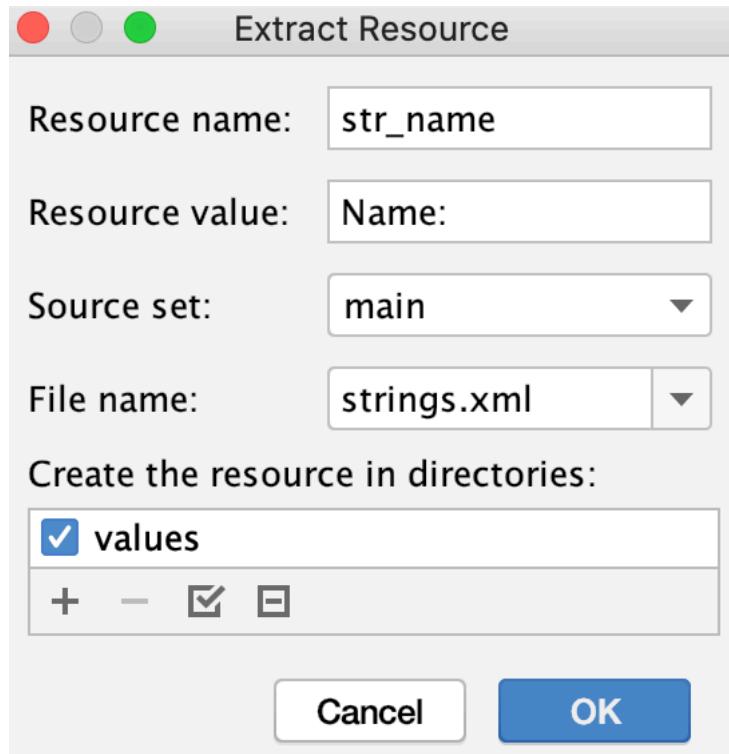
13
14    <TextView
15        android:id="@+id/textView"
16        android:layout_width="0dp"
17        android:layout_height="wrap_content"
18        android:layout_marginStart="8dp"
19        android:layout_marginTop="16dp"
20        android:layout_marginEnd="8dp"
21        android:text="Name:" █
22        app:layout_constraintEnd_toEndOf="parent"
23        app:layout_constraintStart_toStartOf="parent"
24        app:layout_constraintTop_toTopOf="parent" />
25

```

Strings com valores fixos

Da forma como está, não é possível traduzir o aplicativo para outros idiomas. Então, para extrair essa e as demais strings para a pasta de recursos, clique na lâmpada amarela localizada à esquerda e escolha a opção Extract string resource.

Na tela que aparecer, digite o nome do novo recurso, por exemplo str\_name. Perceba que o arquivo com o nome onde recurso será salvo é strings.xml, localizado na pasta res\values do projeto:



Criando o recurso de string

Se o projeto já tiver mais que um idioma, ele aparecerá na lista na parte inferior dessa tela. Clique em OK e repita o processo para os demais atributos text desse arquivo, até que todas as strings que estão fixas nesse arquivo sejam extraídas.

Em seguida, vá até o arquivo `res\values\strings.xml` e perceba que as strings que foram extraídas agora estão lá:

```
<resources>
    <string name="app_name">AndroidProject01</string>
    <string name="str_name">Name:</string>
    <string name="str_description">Description:</string>
    <string name="str_code">Code:</string>
    <string name="str_price">Price:</string>
    <string name="str_save">Save</string>
    <string name="str_empty">empty</string>
</resources>
```

Esse então é o arquivo padrão para os recursos de string. Cada idioma que for acrescentado ao projeto, terá um arquivo igual a esse, traduzido para esse idioma.



O processo descrito aqui para a criação de interfaces gráficas será repetido novamente em outros capítulos. Por isso, caso tenha alguma dúvida posterior de como isso deve ser feito, volte a essa seção para relembrar.

Para verificar como a tela ficou, execute a aplicação no emulador ou em um dispositivo real. Gire o aparelho e verifique que é possível rolar a tela, para exibir os componentes que possivelmente não cabem na tela.



Para visualizar como o arquivo `activity_main.xml` deve ficar, caso tenha dúvidas, consulte o projeto nos extras que podem ser baixados junto com o livro.

## 6.5 - Criando o comportamento da interface gráfica

Agora que a interface gráfica já foi construída, é hora de adicionar algum comportamento a ela. A ideia por enquanto é simples, para seguir com a didática proposta no capítulo:

- O usuário irá digitar os detalhes do produto desejado nos 4 campos superiores da tela;
- Quando finalizar, ele deverá clicar no botão Save;
- Essa ação fará com que a aplicação leia os dados digitados pelo usuário e exiba-os adequadamente nas caixas de texto abaixo do botão Save.

Não se preocupe se esse aplicativo não tem uma função prática. O objetivo é mostrar como criar um código em Kotlin capaz de interagir com a tela e as ações do usuário.

Para começar, vá até o arquivo `MainActivity`. Ele deve estar semelhante ao trecho de código a seguir:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
    }  
}
```

Essa classe representa uma `Activity` no Android, que é responsável por exibir e definir os comportamentos de uma tela da aplicação. Porém, ela não é uma classe qualquer, pois ela estende de `AppCompatActivity`, como pode ser visto em sua declaração. Isso faz com que ela tenha métodos especiais que serão invocados pelo sistema Android.

Um dos métodos especiais dessa classe é o `onCreate`. Ele é chamado pelo sistema Android toda vez que a `Activity` deve ser criada, que pode acontecer em algumas situações como:

- Quando o usuário deseja executar a aplicação, clicando no seu ícone;
- Quando o usuário gira o aparelho, colocando-o em modo paisagem;
- Quando o usuário altera a configuração de idioma do aparelho e a tela precisa ser redesenhada.

As duas últimas condições são chamadas de **alteração de configuração da tela**. Quando isso acontece, a tela precisa ser redesenhada totalmente, pois houve uma configuração em que não é possível alterar sua aparência sem destruí-la e recriá-la novamente. O capítulo seguinte irá tratar esse assunto com um pouco mais de detalhe. O importante agora é saber que esse método é chamado quando a tela precisa ser criada.

Veja também que o método `onCreate` recebe um parâmetro chamado `savedInstanceState` do tipo `Bundle`. Ele também será detalhado no capítulo seguinte.

Dentro desse método existe a instrução `setContentView`, que recebe como parâmetro um recurso de layout para ser exibido, que é justamente a tela `activity_main` que estava sendo construída na seção anterior desse capítulo. Essa instrução é a responsável por, finalmente, exibir a tela dentro da `Activity` para o usuário.

## 6.5.1 - View binding

Agora que a tela já está sendo exibida para o usuário, pode-se começar a criar o código para interagir com ela, mas para isso é necessário criar as referências dos componentes gráficos dentro da classe `ActivityMain`. Dessa forma, o código em Kotlin poderá manipulá-los.

Existem algumas formas diferentes para a criação das referências dos componentes gráficos dentro do código Kotlin. Uma das formas mais modernas até o momento é a utilização de uma técnica chamada **View binding**. Para implementar essa técnica é necessário realizar os seguintes passos:

**a) Indicar ao Android Studio que o projeto deve utilizar a técnica:**

Para isso, abra o arquivo `build.gradle`, vá até a seção `android` e acrescente a subseção `dataBinding` no final, como no trecho a seguir:

```
dataBinding {  
    enabled = true  
}  
}
```

O Android Studio solicitará que o projeto seja sincronizado novamente, através de uma mensagem no canto superior direito da tela. Execute esse passo para que o projeto seja preparado para se trabalhar com **View Binding**.

**b) Envolver o componente ScrollView de `activity_main` em um componente do tipo layout:**

Para isso, abra o arquivo `activity_main.xml` e altere seu início para ficar com o trecho a seguir:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    xmlns:tools="http://schemas.android.com/tools">  
  
    <ScrollView  
        android:layout_width="match_parent"  
        android:layout_height="match_parent">
```

Da mesma forma, altere seu fim para que as tags do XML fechem corretamente:

```
    </ScrollView>  
</layout>
```

Isso fará com que o Android Studio gere as classes necessárias para representar todos os componentes dessa tela, facilitando muito o trabalho a ser desenvolvido na classe `MainActivity`.

**c) Criar o atributo `ActivityMainBinding` na classe `MainActivity`:**

Depois que os passos anteriores foram realizados, o Android Studio gera automaticamente uma classe do tipo `ActivityMainBinding`, que contém a referência para todos os componentes gráficos da tela definida em `activity_main.xml`. Os identificadores únicos que foram definidos nesse arquivo são utilizados para definir os nomes das referências dentro de `ActivityMainBinding`.

Agora, basta criar um atributo desse tipo dentro da classe `MainActivity`, como no trecho a seguir:

```
import br.com.siecola.androidproject01.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
```

A última linha do trecho anterior é a criação do atributo do tipo `ActivityMainBinding`, da classe `MainActivity`. Repare que ele está com o modificador `lateinit`, para garantir o Kotlin que ele será inicializado antes de ser utilizado. Sem isso, o compilador geraria um erro.

Finalizando, para que esse atributo seja inicializado, é necessário associá-lo com o recurso de layout de tela que será exibido. Isso deve ser feito dentro do método `onCreate`, logo após a instrução `super.onCreate(savedInstanceState)`, como no trecho a seguir:

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

Essa instrução deve substituir a que estava em seu lugar:

```
setContentView(R.layout.activity_main)
```

Ou seja, essa linha deve ser removida dessa função.

A partir desse ponto, o atributo `binding` possui todas as referências dos componentes criados em `activity_main.xml`, e podem ser utilizados para a definição do comportamento dessa tela.

## 6.5.2 - Criando o comportamento da tela

As seguintes ações podem ser seguidas para definir o comportamento da tela de uma aplicação Android:

- Reagir ao evento de clique em um botão;
- Ler uma caixa de texto editável com o valor digitado pelo usuário;
- Atualizar um conteúdo em uma caixa de texto;
- Alterar uma imagem em um `ImageView`.
- E vários outros.

Cada evento pode ser tratado e implementado de forma diferente, dependendo do componente e do evento, mas a ideia principal é indicar ao componente que existe uma função especial que deve ser invocada quando o evento acontecer.

Para exemplificar, pegue a referência do botão `btnSave` dentro do atributo `binding` da classe `ActivityMain`, criado em `activity_main.xml` e configure-o para executar uma função que imprime uma mensagem de log, como no trecho a seguir, que deve ser colocado no final da função `onCreate`:

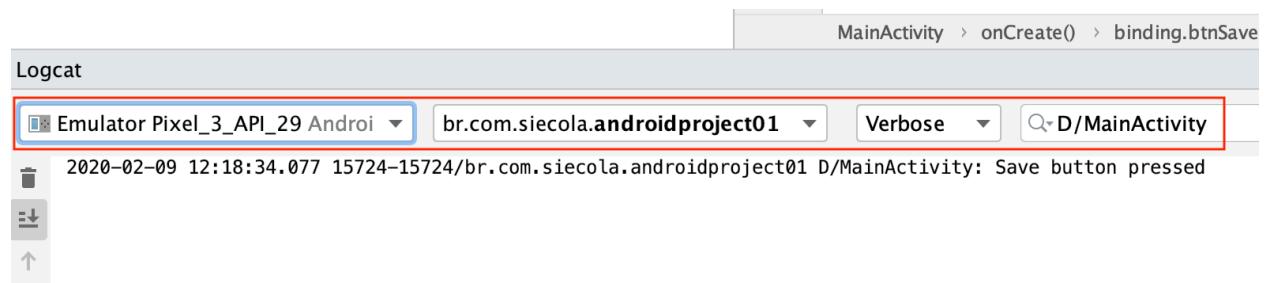
```
binding.btnSave.setOnClickListener {
    Log.d("MainActivity", "Save button pressed")
}
```

Basicamente, esse trecho executa os seguintes passos:

- Pega o atributo `binding`, que possui todas as referências dos componentes gráficos criados em `activity_main.xml`;
- Acessa o componente de identificação `btnSave`;
- Define o comportamento para o *listener* invocado quando o botão é clicado para executar a função anônima delimitada por `{ }`;
- Dentro dessa função anônima, a instrução `Log.d("MainActivity", "Save button pressed")` imprime uma mensagem de log.

A classe `Log` é a responsável por imprimir mensagens no Logcat do Android Studio. O primeiro parâmetro é chamado de *tag*, que é utilizado para diferenciar mensagens de contextos diferentes. Normalmente esse parâmetro leva o nome da classe em que está sendo executado. O segundo parâmetro é a mensagem em si.

Para testar essa implementação, execute a aplicação novamente. Assim que ela estiver rodando, clique no botão Save e veja que o log foi impresso no Logcat, localizado nessa aba na parte inferior do Android Studio:



Exibindo uma mensagem de log

Tenha certeza de selecionar o dispositivo correto, bem como o nome do aplicativo e configurar o filtro de mensagens para `D/MainActivity`, que é o nível de log em debug e o nome da classe que o gerou.

Agora que o evento de clique no botão `btnSave` está funcionando, é possível finalizar a implementação desse aplicativo, exibindo os detalhes do produto, que foram digitados pelo usuário, nas caixas de texto localizadas após o botão de salvar. Para isso, acrescente o seguinte trecho de código na função que trata o evento de clique do botão `btnSave`:

```
binding.txtName.text = binding.edtName.text
binding.txtDescription.text = binding.edtDescription.text
binding.txtCode.text = binding.edtCode.text
binding.txtPrice.text = binding.edtPrice.text
```

Como o atributo da classe `binding` possui a referência de todos os componentes da tela, basta acessar cada uma das caixas de texto editáveis, os `EditText`, buscando pela propriedade `text` e atribuir seu valor ao `TextView` correspondente. Isso fará com que a informação digitada pelo usuário seja exibida no final da tela.

Veja como deve ficar toda a classe `MainActivity`:

```
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.util.Log
import androidx.databinding.DataBindingUtil
import br.com.siecola.androidproject01.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

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

        binding.btnSave.setOnClickListener {
            Log.d("MainActivity", "Save button pressed")

            binding.txtName.text = binding.edtName.text
            binding.txtDescription.text = binding.edtDescription.text
            binding.txtCode.text = binding.edtCode.text
            binding.txtPrice.text = binding.edtPrice.text
        }
    }
}
```

Para testar a implementação, execute novamente a aplicação, preencha os campos com os detalhes de um produto qualquer e clique no botão Save. Os campos de texto deverão ficar preenchidos com tais informações, como mostra a figura a seguir:

Code:

COD1

---

Price:

10

---

SAVE

Product1  
Desc1  
COD1  
10

**Testando a aplicação**

Essa aplicação poderia ficar melhor. Pelo menos poderia tratar a condição quando o usuário não digitar um valor para o nome do produto, considerando que somente ele fosse necessário. Nesse caso, uma mensagem de erro poderia ser lançada na tela. Para isso, acrescente uma condição para validar esse teste:

```
if (!binding.edtName.text.isEmpty()) {  
    binding.txtName.text = binding.edtName.text  
    binding.txtDescription.text = binding.edtDescription.text  
    binding.txtCode.text = binding.edtCode.text  
    binding.txtPrice.text = binding.edtPrice.text  
} else {  
    Toast.makeText(this, "Please, enter the name.", Toast.LENGTH_SHORT).show()  
}
```

Na primeira linha desse trecho, foi verificado se o atributo text do componente edtName não está vazio. Se ele não estiver, significa que o usuário digitou algo nesse campo e então os detalhes do produto podem ser lidos.

Caso não haja informação dentro do atributo text de edtName, uma mensagem é lançada na tela, através da instrução `Toast.makeText`. Essa instrução recebe os seguintes parâmetros:

- Uma referência da Activity na qual a mensagem deve aparecer;
- A mensagem de texto em si;
- A duração em que a mensagem deve ficar exibida na tela.

Por último, o método `.show()` é chamado para exibir a mensagem na tela ao usuário.

Por fim, para manter a estratégia coerente de possibilitar que esse aplicativo seja traduzido para outros idiomas, é necessário adicionar a string `Please, enter the name` como um recurso do projeto. Isso pode ser feito colocando o cursor sobre esse texto e acessando a opção `Extract string resource` da lâmpada amarela que aparecer do lado esquerdo. Para finalizar, basta digitar um nome para o recurso, por exemplo `str_enter_name`.

Com essa última modificação, o Android Studio se encarrega de alterar o código Kotlin para buscar a string do recurso definido, como pode ser visto no trecho a seguir:

```
Toast.makeText(this, getString(R.string.str_enter_name), Toast.LENGTH_SHORT).show()
```

Quando esse aplicativo for traduzido para outros idiomas, o Android se encarregará de pegar o texto no idioma correto para ser exibido.



É importante lembrar que o código do projeto está no GitHub e pode ser acessado [aqui<sup>16</sup>](#).

## 6.6 - Conclusão

Esse capítulo apresentou alguns conceitos iniciais de um projeto Android, bem como os passos para a construção de interfaces gráficas utilizando o `ConstraintLayout`, o que faz com que a tela fique adaptável a diversos tamanhos de dispositivos.

Também apresentou o conceito de `View binding`, uma técnica moderna que facilita a associação e utilização das referências dos componentes gráficos dentro do código em Kotlin.

Por fim, iniciou o leitor na construção de comportamentos da aplicação, com interação direta com a interface gráfica do usuário.

O próximo capítulo traz conceitos mais aprofundados sobre `Activity`, introduzindo conceitos de ciclo de vida e gerenciamento de seus estados.

<sup>16</sup>[https://github.com/siecola/android\\_project01](https://github.com/siecola/android_project01)

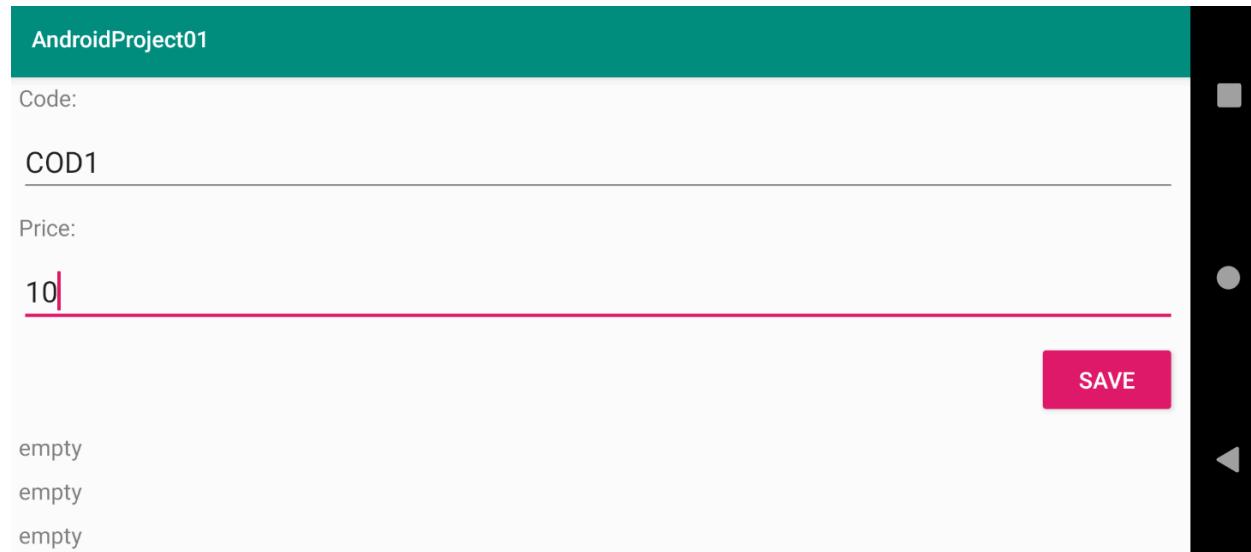
# 7 - Activity e seu ciclo de vida

No capítulo anterior, o comportamento da tela foi criado dentro do método `onCreate` da classe `MainActivity`, bem como a exibição do layout através da seguinte instrução dentro desse método:

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

Tudo funciona muito bem: o usuário digita os detalhes do produto, clica no botão para salvar e os mesmos detalhes são exibidos nas caixas de texto na parte inferior da tela, exatamente como foi planejado.

Porém, caso o usuário gire o dispositivo, fazendo com que aconteça uma mudança de configuração da tela, ela acaba sendo redesenhada e a informação que havia dentro das caixas de texto na parte inferior desaparecem, voltando com o texto original, ou seja, “empty”:



Alteração de configuração do dispositivo

Esse fato acontece devido a essa mudança de configuração, que faz com que a tela tenha que ser destruída e recriada novamente. Isso faz parte do ciclo de vida de uma Activity, que será tratado nesse capítulo.

## 7.1 - O que é uma Activity

A Activity é uma parte essencial de uma aplicação Android, pois é nela que a interface do usuário é construída e exibida.

Uma aplicação pode ter mais de uma Activity, para exibir mais de uma tela, embora esse modelo tenha sido substituído por uma arquitetura com múltiplos fragmentos, como será visto no capítulo seguinte.

Para que uma Activity seja reconhecida pelo sistema Android, é necessário declará-la no arquivo `AndroidManifest.xml`, como já foi feito no projeto `android_project01`:

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

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

Essa configuração faz com que o sistema Android reconheça a `MainActivity` e além disso, lance-a em primeiro lugar, quando a aplicação é iniciada.

Caso a aplicação tenha mais de uma Activity, é necessário declarar todas nesse arquivo, mas somente uma pode ser considerada principal para ser iniciada junto à aplicação.

Em capítulos mais adiante, será desenvolvida uma aplicação com mais de uma Activity, para fazer com que o usuário se autentique.

Basicamente, toda aplicação possui uma Activity, inclusive as próprias aplicações nativas do Android, como a tela de realizar ligações, contatos, câmera e assim por diante.

## 7.2 - O ciclo e vida de uma activity

Uma Activity possui um ciclo de vida dentro do sistema Android que pode mudar de acordo com várias situações, como por exemplo:

- Se o usuário pressionar o botão de voltar do aparelho;
- Se o usuário pressionar o botão *home*;
- Se uma ligação chegar e o usuário atender;
- Se o usuário mudar para outra aplicação

Em todas essas situações, a Activity da aplicação, que estava sendo exibida ao usuário, passa do estado em que estava em exibição em primeiro plano (*foreground*), para um estado onde não está mais sendo exibida, ou seja, em segundo plano (*background*). Porém, a aplicação ainda está em execução, mas ela é notificada que não está mais em exibição.

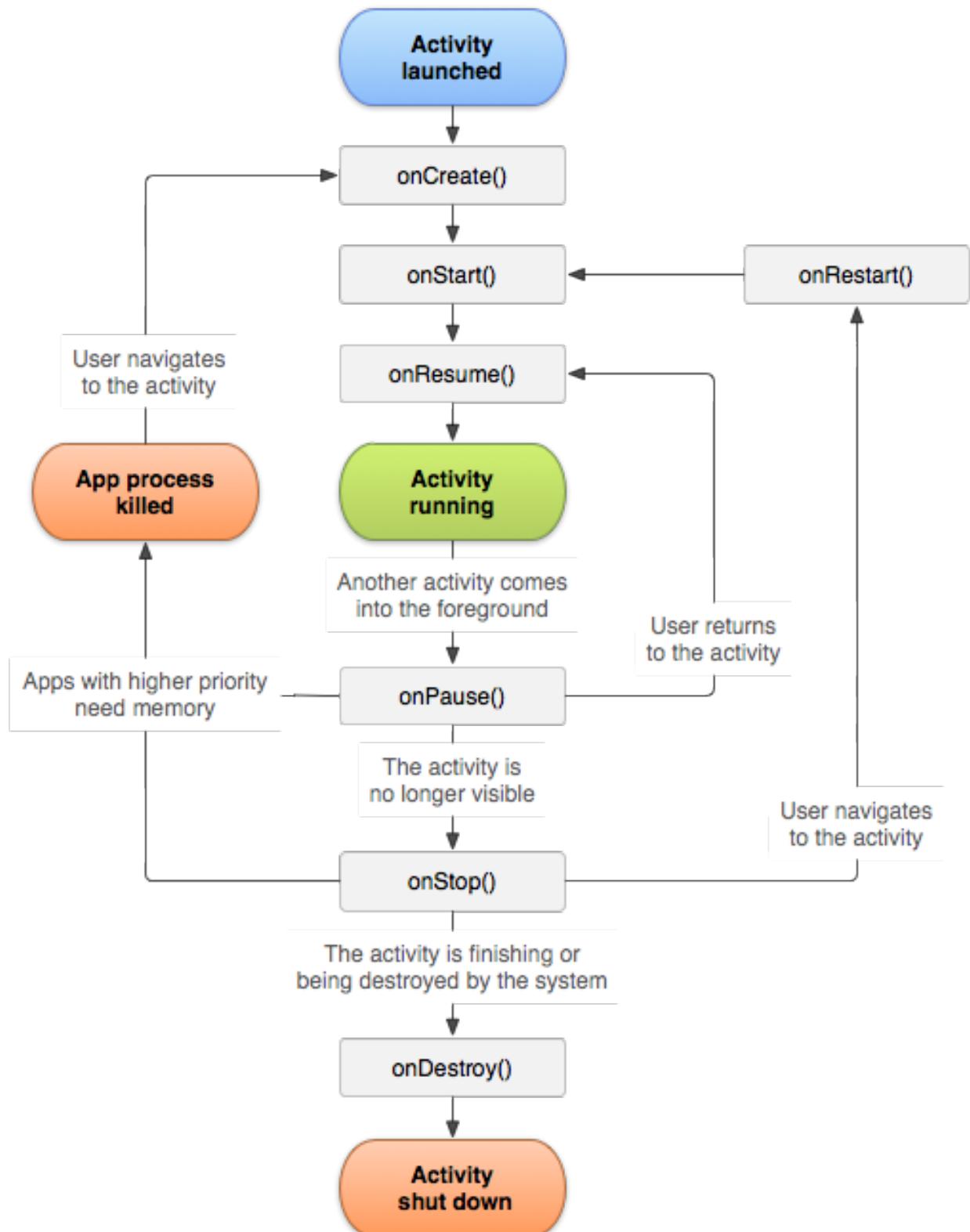
É necessário avisar a Activity dessas mudanças de estado, pois, ela pode precisar interromper ou retomar algum processo em que a interface gráfica do usuário está sendo atualizada. O que obviamente só é necessário de ser feito quando tal interface está em primeiro plano.

O sistema Android notifica para a Activity dessas mudanças através de métodos, como o `onCreate`, avisada-a que a tela precisa ser recriada, pelas razões já mencionadas.

A seguir, uma breve descrição de alguns métodos que a Activity possui para indicar suas transições de estado:

- **onStart**: ocorre quando a Activity está visível para o usuário, ou seja, em *foreground*. Aqui é onde o código para inicializar informações da tela deve ficar;
- **onResume**: ocorre quando a Activity está apta a receber eventos do usuário pela interface gráfica;
- **onPause**: o sistema Android utiliza esse método para indicar que a Activity não ficará mais visível ao usuário;
- **onStop**: ocorre quando a Activity já não está em *background*, ou seja, foi substituída por outra;
- **onRestart**: ocorre quando o usuário navega de volta para a Activity e ela está prestes a se tornar visível novamente;
- **onDestroy**: ocorre quando o sistema Android destrói a Activity completamente.

A seguir, um diagrama que ilustra a transição entre esses estados.



Ciclo de vida de uma Activity - Fonte: <https://developer.android.com/guide/components/activities/activity-lifecycle>

Um bom teste para entender alguns cenários e quando os métodos são invocados pelo sistema Android, é implementá-los e colocar uma linha de log para indicar que a execução passou por ali, como no trecho a seguir:

```
Log.d("MainActivity", "onCreate")
```

Essa linha pode ser adicionada no fim do método `onCreate`. E para os demais métodos, implemente-os como no trecho a seguir:

```
override fun onStart() {
    super.onStart()
    Log.d("MainActivity", "onStart")
}

override fun onResume() {
    super.onResume()
    Log.d("MainActivity", "onResume")
}

override fun onPause() {
    super.onPause()
    Log.d("MainActivity", "onPause")
}

override fun onStop() {
    super.onStop()
    Log.d("MainActivity", "onStop")
}

override fun onRestart() {
    super.onRestart()
    Log.d("MainActivity", "onRestart")
}

override fun onDestroy() {
    super.onDestroy()
    Log.d("MainActivity", "onDestroy")
}
```

Agora execute a aplicação novamente, veja que os logs que apareceram no Logcat correspondem aos eventos de criação, *start* e *resume*:

The screenshot shows the Android Logcat interface. At the top, it displays the device as "Emulator Pixel\_3\_API\_29 Androi" and the package name as "br.com.siecola.androidproject01". The filter is set to "Verbose" and the search term is "D/MainActivity". The log output shows the following entries:

```

2020-02-09 19:06:34.321 20213-20213/br.com.siecola.androidproject01 D/MainActivity: onCreate
2020-02-09 19:06:34.334 20213-20213/br.com.siecola.androidproject01 D/MainActivity: onStart
2020-02-09 19:06:34.338 20213-20213/br.com.siecola.androidproject01 D/MainActivity: onResume

```

Below the log, there are navigation buttons: up, down, and double arrows, followed by tabs for "5: Debug", "6: Logcat" (which is selected), "TODO", "Terminal", "Version Control", "Build", and "Profiler". The title "Logs do ciclo de vida da Activity" is centered at the bottom.

Se o botão de *home* do dispositivo for pressionado, a Activity irá para *background* e os seguintes eventos serão chamados:

The screenshot shows the Android Logcat interface. The setup is identical to the previous one. The log output shows the following entries:

```

2020-02-09 19:24:02.046 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onCreate
2020-02-09 19:24:02.068 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onStart
2020-02-09 19:24:02.071 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onResume

```

Following these, two additional entries are shown, both highlighted with a red box:

```

2020-02-09 19:24:08.296 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onPause
2020-02-09 19:24:09.206 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onStop

```

Below the log, there are navigation buttons: up, down, and double arrows, followed by tabs for "5: Debug", "6: Logcat" (selected), "TODO", "Terminal", "Version Control", "Build", and "Profiler". The title "Logs do ciclo de vida da Activity" is centered at the bottom.

Isso significa que a Activity foi colocada em *background*, ou seja, não está mais visível para o usuário, tão pouco está recebendo eventos dele.

Se o usuário navegar de volta para o aplicativo, utilizando os controles do aparelho, a Activity ficará visível novamente e os seguintes estados serão acionados pelo sistema Android:

The screenshot shows the Android Logcat interface. The setup is identical to the previous ones. The log output shows the following entries:

```

2020-02-09 19:24:08.296 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onPause
2020-02-09 19:24:09.206 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onStop

```

Following these, three additional entries are shown, all highlighted with a red box:

```

2020-02-09 19:26:08.858 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onRestart
2020-02-09 19:26:08.859 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onStart
2020-02-09 19:26:08.869 20825-20825/br.com.siecola.androidproject01 D/MainActivity: onResume

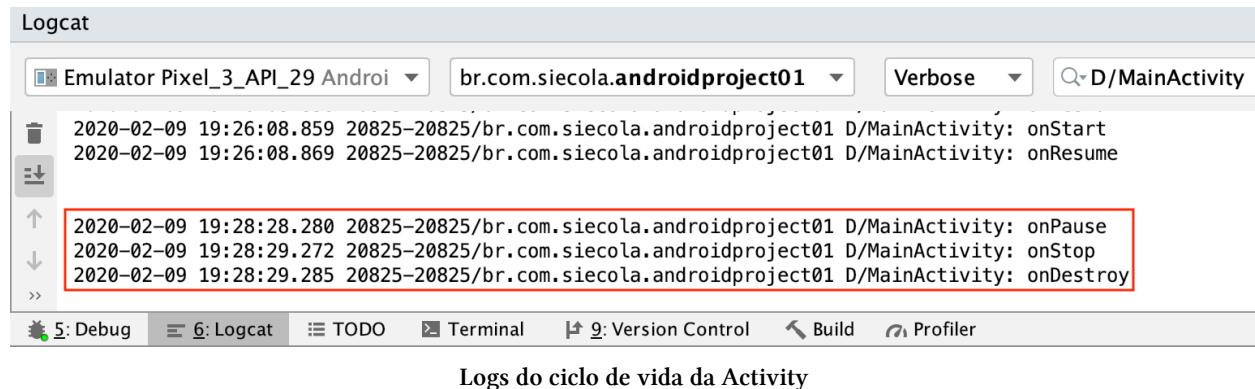
```

Below the log, there are navigation buttons: up, down, and double arrows, followed by tabs for "5: Debug", "6: Logcat" (selected), "TODO", "Terminal", "Version Control", "Build", and "Profiler". The title "Logs do ciclo de vida da Activity" is centered at the bottom.

Nesse momento, a Activity está novamente visível ao usuário, ou seja em *foreground* e apta a receber eventos do usuário.

Para finalizar, se o usuário pressionar o botão de voltar do aparelho, indicando que não deseja mais

interagir com a aplicação, o sistema Android irá destruir a Activity, porém irá invocar métodos onPause, onStop e por fim o onDestroy, indicando que ela será destruída:



#### Logs do ciclo de vida da Activity

Com esses conceitos agora é possível entender o que acontece quando o aparelho é girado e os dados do produto salvo são perdidos, como foi explicado no início desse capítulo. Para isso, limpe os logs do Logcat e execute a aplicação novamente. Os seguintes métodos serão chamados, como já foi demonstrado:

```
2020-02-09 19:40:15.297 21081-21081/br.com.siecola.androidproject01 D/MainActivity: \
onCreate
2020-02-09 19:40:15.312 21081-21081/br.com.siecola.androidproject01 D/MainActivity: \
onStart
2020-02-09 19:40:15.317 21081-21081/br.com.siecola.androidproject01 D/MainActivity: \
onResume
```

Agora preencha as informações do produto e salve:

```
2020-02-09 19:43:45.195 21081-21081/br.com.siecola.androidproject01 D/MainActivity: \
Save button pressed
```

Agora gire o aparelho, para colocá-lo em modo paisagem. Veja que os seguintes métodos são invocados:

```
2020-02-09 19:44:41.911 21081-21081;br.com.siecola.androidproject01 D/MainActivity: \
onPause
2020-02-09 19:44:41.936 21081-21081;br.com.siecola.androidproject01 D/MainActivity: \
onStop
2020-02-09 19:44:41.945 21081-21081;br.com.siecola.androidproject01 D/MainActivity: \
onDestroy
2020-02-09 19:44:42.312 21081-21081;br.com.siecola.androidproject01 D/MainActivity: \
onCreate
2020-02-09 19:44:42.327 21081-21081;br.com.siecola.androidproject01 D/MainActivity: \
onStart
2020-02-09 19:44:42.354 21081-21081;br.com.siecola.androidproject01 D/MainActivity: \
onResume
```

Perceba que o método `onDestroy` foi chamado, indicando que a Activity foi destruída para ser reconstruída novamente, pois agora ela possui um novo layout. Nesse momento, os dados que foram digitados e salvos pelo usuário foram perdidos, pois a Activity que os controlava foi destruída. Logo em seguida o método `onCreate` foi acionado novamente, para que a Activity seja criada novamente, mas agora sem os dados que foram salvos pelo usuário.

Esse é um cenário muito comum e deve ser tratado adequadamente para evitar que o usuário perca dados enquanto está utilizando a aplicação.

## 7.3 - Salvando o estado de uma Activity

Para corrigir o problema levantado no fim da seção anterior, é necessário entender outros dois métodos que cuidam do ciclo de vida de uma Activity:

- **onSaveInstanceState**: quando existe uma mudança de configuração da Activity, esse método é chamado antes da Activity ser destruída, permitindo que a aplicação salve informações e estados para serem utilizados posteriormente;
- **onRestoreInstanceState**: esse método é chamado momentos antes da Activity tornar a ser visível novamente, depois da mudança de configuração, quando a tela foi reconstruída novamente. Nesse momento é possível recuperar as informações salvas no método `onSaveInstanceState` e permitir que o usuário continue utilizando a aplicação sem perder nenhuma informação.

Novamente, para entender melhor quando esses métodos são chamados, implemente-os como no trecho a seguir, colocando uma linha de log em cada um:

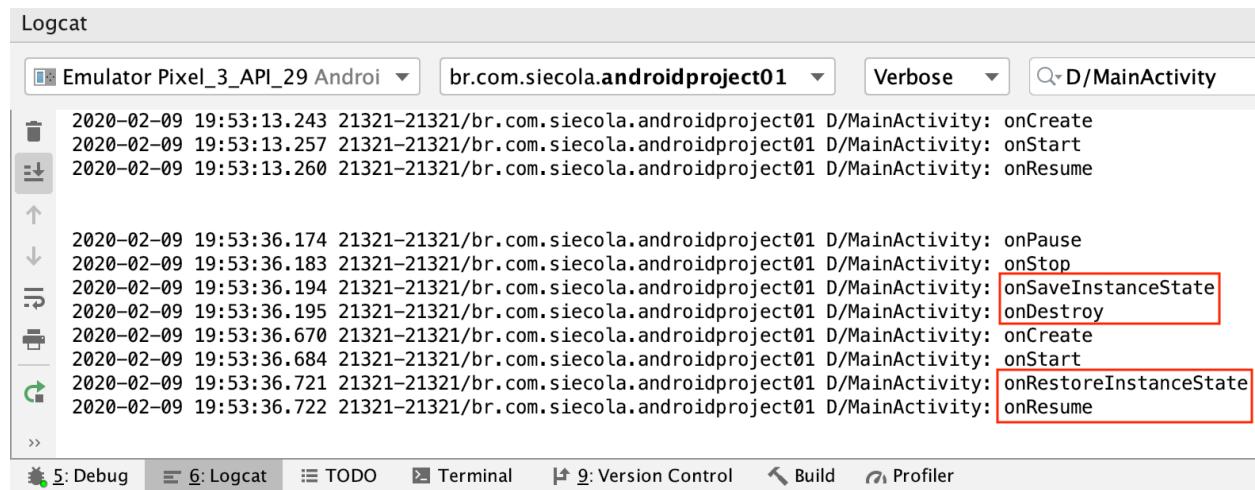
```

override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    Log.d("MainActivity", "onSaveInstanceState")
}

override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    Log.d("MainActivity", "onRestoreInstanceState")
}

```

Agora execute novamente a aplicação, gire o aparelho, visualizando os logs no Logcat:



#### Salvando o estado da Activity

Veja que o método `onSaveInstanceState` foi chamado antes da Activity ser destruída, logo ele é um bom lugar para salvar informações do produto. Isso pode ser feito utilizando o objeto passado como parâmetro a essa função, chamado `outState` e do tipo `Bundle`.

Da mesma forma, o método `onRestoreInstanceState` foi chamado antes da tela tornar-se ativa novamente para o usuário, ou seja, é um bom lugar para recuperar os dados salvos do produto no objeto do tipo `Bundle` dentro da função `onSaveInstanceState`, que também é passado para a função `onRestoreInstanceState`.

O objeto do tipo `Bundle` presente nessas duas funções permanece intacto durante uma mudança de configuração da Activity, como quando a tela do dispositivo muda para o estado paisagem ou retrato. Com ele, é possível salvar e recuperar informações através de uma chave do tipo `String`, inclusive objetos complexos.

Para demonstrar como esse mecanismo pode ser utilizado para salvar as informações do produto digitadas pelo usuário, crie uma nova classe chamada `Product`, no mesmo pacote da classe `MainActivity`, para armazenar tais informações:

```
import java.io.Serializable

data class Product(var name: String, var description: String, var code: String, var \
price: Double) :
    Serializable
```

Repare que essa classe é do tipo data, que é o que permite que alguns métodos já sejam criados automaticamente sem a necessidade de escrevê-los. Veja também que ela estende de Serializable, para que objetos desse tipo possam ser salvos no Bundle.

Agora volte ao método onSaveInstanceState e crie um novo objeto do tipo Product, preenchido com as informações digitadas pelo usuário, para que ele possa ser salvo no Bundle, como mostra o trecho a seguir:

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    Log.d("MainActivity", "onSaveInstanceState")

    val product = Product(
        name = binding.edtName.text.toString(),
        description = binding.edtDescription.text.toString(),
        code = binding.edtCode.text.toString(),
        price = binding.edtPrice.text.toString().toDouble()

    outState.putSerializable("product", product)
}
```

Depois que o objeto do tipo Product é criado, ele então é salvo no Bundle através da instrução outState.putSerializable("product", product), que recebe a chave como primeiro parâmetro e o produto em si, como segundo parâmetro.

Agora, para recuperar o produto que foi salvo no Bundle, altere o método onRestoreInstanceState de acordo com o trecho a seguir:

```
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    Log.d("MainActivity", "onRestoreInstanceState")

    val product = savedInstanceState.getSerializable("product") as Product

    binding.txtName.text = product.name
    binding.txtDescription.text = product.description
    binding.txtCode.text = product.code
    binding.txtPrice.text = product.price.toString()
}
```

Veja que o produto é restaurado do Bundle utilizando a mesma chave passada para salvá-lo: a string product.

Depois que o produto é recuperado, a tela então pode ser restaurada com essas informações.

Teste novamente a aplicação, salvando um produto e girando o aparelho. Agora as informações não deverão mais desaparecer quando esse evento acontece.



É importante lembrar que o código do projeto completo está disponível nos extras que pode ser baixado junto com o livro.

## 7.4 - Conclusão

Esse capítulo tratou de um fato importante no desenvolvimento de aplicativos para Android, que são os estados e o ciclo de vida de uma Activity, apresentando uma das possíveis técnicas para salvar tais estados e deixar que o usuário continue utilizando a aplicação em caso de mudança de configuração.

O mesmo conceito pode ser aplicado para fragmentos, pois eles sempre estão dentro de uma Activity, o que acaba passando por estados semelhantes.

Os próximos capítulos avançam para um novo projeto, onde serão apresentados conceitos mais avançados, como:

- Fragmentos: onde a aplicação terá mais de uma tela;
- Listas para exibição de vários ítems;
- Consumo de um serviço REST utilizando Retrofit;
- ViewModel e LiveData: conceitos avançados para lidar de forma elegante com ciclo de vida de fragmentos.

# 8 - Consumindo serviços REST com Retrofit e coroutines

REST é um acrônimo para REpresentational State Transfer. É uma forma de definir comunicação entre sistemas, principalmente através da Internet, utilizando requisições HTTP.

Basicamente essa arquitetura define alguns conceitos:

- **Provedor:** a aplicação que provê os recursos para serem consumidos por clientes interessados em seus serviços;
- **Serviços:** conjunto de operações contextuais que o provedor oferece, como por exemplo um serviço de produtos, usuários ou pedidos;
- **Operações:** dentro de cada serviço, representa uma ação que pode ser solicitada pelo consumidor, como cadastrar um usuário, alterar um produto ou cancelar um pedido;
- **Consumidor:** é a entidade que solicita operações de cada serviço oferecido pelo provedor.

É muito comum que aplicações Android necessitem consumir serviços REST para exibir informações ao usuário, seja como parte principal de suas funcionalidades, como em aplicativos de redes sociais, ou seja como funcionalidade adicional, como em apps de investimento onde alguma taxa ou cotação de ações deve ser buscada para a realização de algum cálculo.

Apesar de ser uma tarefa comum, consumir serviços REST em aplicações apresenta vários desafios, como:

- Executar tarefas assíncronas e atualizar a interface do usuário;
- Garantir o mínimo de consumo de bateria e de dados de rede do aparelho;
- Lidar com exceções inerentes do processo de comunicação através da Internet.

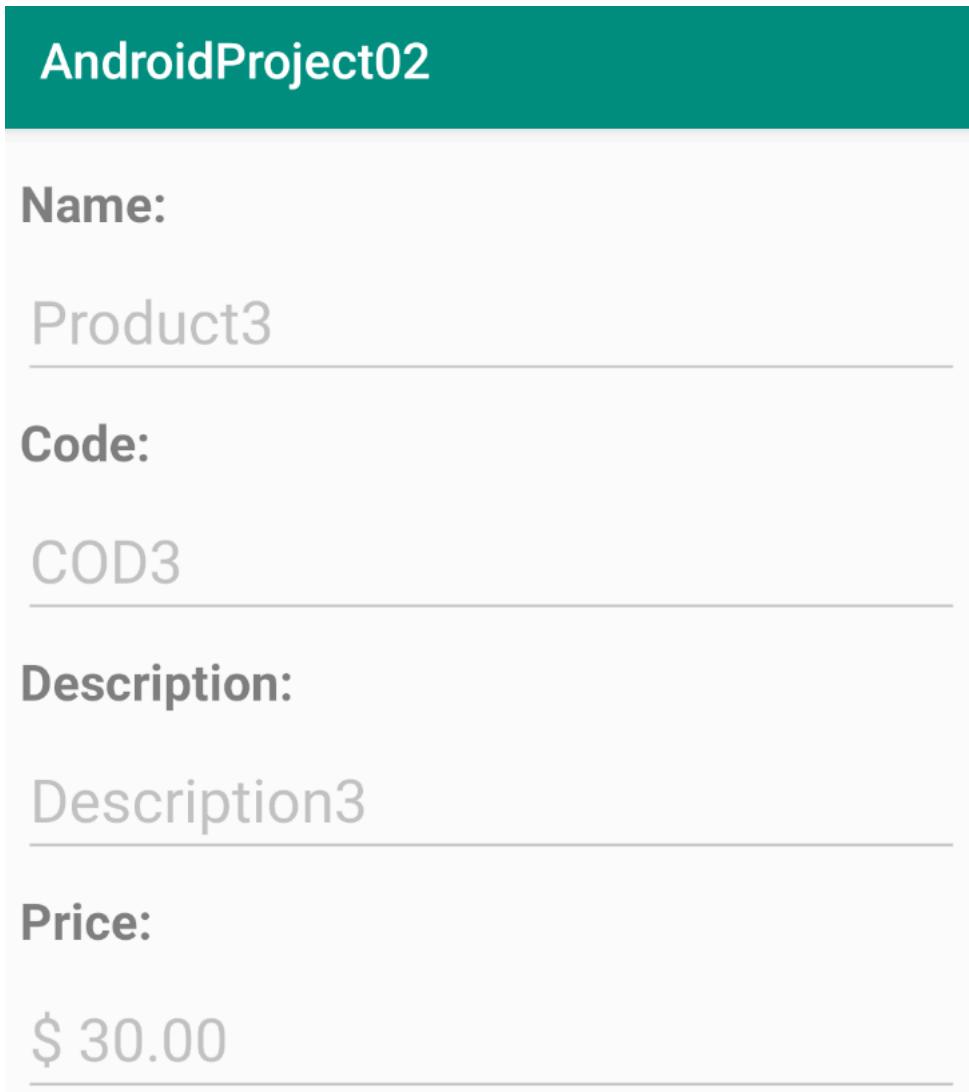
O objetivo desse e dos próximos capítulos é construir uma aplicação Android capaz de consumir serviços REST de um provedor de serviços de vendas, detalhado na próxima seção. A ideia é **exibir uma lista de produtos previamente cadastrados**, além de seus detalhes.

A seguir algumas imagens do que será desenvolvido ao final desses capítulos:

AndroidProject02		
Product1	COD1	\$ 10.00
Product2	COD2	\$ 20.00
Product3	COD3	\$ 30.00

Tela com a lista de produtos

Essa é a tela com a lista de produtos. Quando o usuário clicar em um dos produtos exibidos aqui, ele será redirecionado para a tela com os detalhes do produto selecionado:



Tela com os detalhes do produto selecionado

Durante o processo de criação dessa nova aplicação Android, muitos conceitos serão apresentados. A seguir um diagrama ilustrativo do que será desenvolvido:

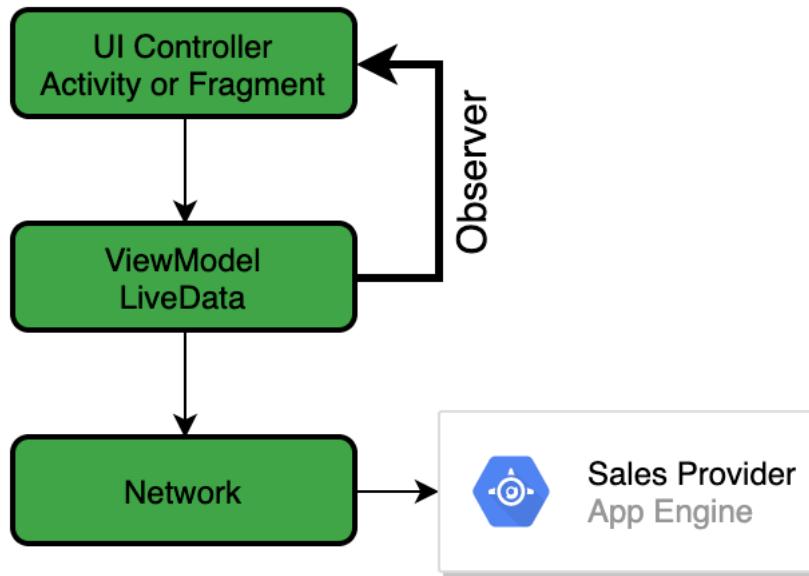


Diagrama da aplicação

As camadas apresentadas nessa figura ilustram as partes que serão desenvolvidas, que começará de baixo para cima, iniciando na camada de rede (Network), depois de visualização (ViewModel), até chegar na interface de usuário (UI Controller), que ainda será desenvolvida nos capítulos seguintes.

Esse capítulo se concentra em introduzir e detalhar os seguintes tópicos:

- Introduzir o conceito de permissões no Android;
- Criar uma camada para salvar informações nas SharedPreferences, um local de fácil acesso para persistência de dados não voláteis que representam estados ou preferências do usuário;
- Criar o cliente de serviços REST utilizando Retrofit;
- Requisitar e manipular token OAuth para autenticação no provedor de serviços de vendas;
- Criar interceptadores de requisições no Retrofit para injeção de cabeçalhos na requisição HTTP;
- Iniciar a criação da camada responsável pela exibição dos produtos para o usuário, que é chamada de ViewModel;
- Consumir o serviço de produtos do provedor de serviços de vendas utilizando Kotlin coroutines.

Os próximos capítulos continuarão com o processo de implementação dessa aplicação, introduzindo novos conceitos importantes como fragmentos e navegação entre diferentes telas da aplicação.

Esse capítulo se limitará a criação da camada de serviço responsável por estabelecer a comunicação com o provedor de serviços de vendas, mas para isso, será necessário iniciar a construção de algumas peças importantes que serão finalizadas nos próximos capítulos.



O código-fonte do projeto completo encontra-se nesse [repositório<sup>17</sup>](https://github.com/siecola/android_project02).

<sup>17</sup>[https://github.com/siecola/android\\_project02](https://github.com/siecola/android_project02)

## 8.1 - O provedor de serviços de vendas

O provedor de serviços de vendas, citado anteriormente, será responsável por prover os serviços que serão consumidos por essa nova aplicação Android que será iniciada nesse capítulo. Esse provedor foi construído utilizando Spring Boot e está hospedado no Google App Engine, uma plataforma de cloud computing do Google. Caso haja interesse do leitor em saber como ele foi construído, consulte o seguinte [livro<sup>18</sup>](#). Seu código-fonte encontra-se nesse [repositório<sup>19</sup>](#).



Felizmente, não será necessário que o leitor construa ou hospede seu próprio provedor de vendas, pois um já está disponível para ser utilizado.

O provedor de serviços de vendas a ser utilizado nesse projeto está hospedado no Google App Engine e sua URL base de acesso é:

[https://sales-provider.appspot.com<sup>20</sup>](https://sales-provider.appspot.com)

As seções a seguir detalham as operações do provedor de serviços de vendas que serão utilizados pela aplicação, bem como seu diagrama de mecanismo de autenticação. Caso o leitor deseje acessar os serviços do provedor através de um cliente REST, a sugestão é utilizar o [Postman<sup>21</sup>](#).

E para aqueles que possuem familiaridade com essa aplicação, existe uma *collection* com as principais requisições ao provedor de serviço de vendas preparada para o Postman nesse [repositório<sup>22</sup>](#).

A ideia desse capítulo é detalhar ao leitor as operações que serão utilizadas pela aplicação Android, antes de construí-las em seu código-fonte.

### 8.1.1 - Diagrama do provedor de serviços de vendas

O provedor de serviços de vendas é uma aplicação Web construída em Java utilizando o *framework* [Spring Boot<sup>23</sup>](#) para ser hospedado na plataforma [Google App Engine<sup>24</sup>](#).

Basicamente a aplicação possui os seguintes serviços:

- **Gerenciamento de usuários:** permite a criação, alteração e exclusão de usuários que podem ter acesso aos demais serviços do provedor;
- **Gerenciamento de produtos:** permite a criação, alteração e exclusão de produtos disponíveis no provedor de serviços de vendas, para serem associados aos pedidos a serem feitos pelos usuários;

<sup>18</sup><https://www.casadocodigo.com.br/products/livro-gae>

<sup>19</sup><https://github.com/siecola/GAESalesProvider>

<sup>20</sup><https://sales-provider.appspot.com/>

<sup>21</sup><https://www.postman.com>

<sup>22</sup>[https://github.com/siecola/GAESalesProvider/blob/master/postman/GAE\\_Sales.postman\\_collection.json](https://github.com/siecola/GAESalesProvider/blob/master/postman/GAE_Sales.postman_collection.json)

<sup>23</sup><https://spring.io/projects/spring-boot>

<sup>24</sup><https://cloud.google.com/appengine>

- **Gerenciamento de pedidos:** permite a criação e exclusão de pedidos, como em uma loja virtual, que podem ser feitos pelo usuários, associando produtos existentes nessa loja;
- **Autenticação de acesso:** todos os serviços necessitam de autenticação para serem acessados. Isso é feito através do mecanismo OAuth2.

Veja seu diagrama simplificado a seguir:

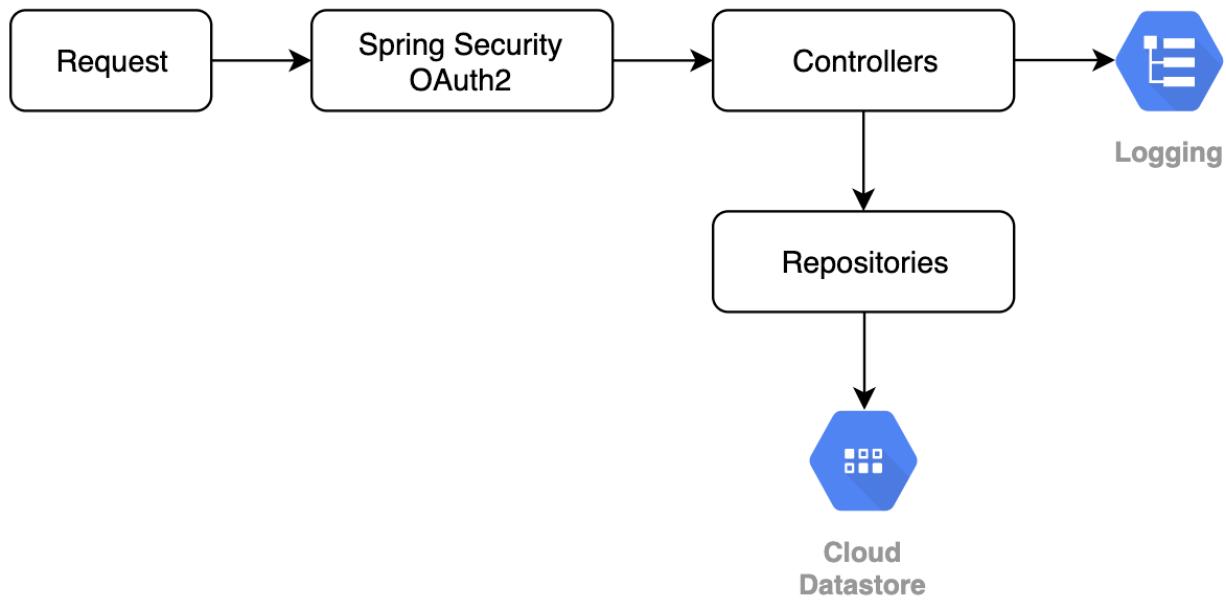


Diagrama do provedor de serviços de vendas

Toda a requisição ao provedor de serviço de vendas deve ser autenticado, por isso a camada **Sprint Security** cuida de interceptar todas as chamadas aos **Controllers** que fornecem os serviços citados anteriormente pela aplicação.

Para cada serviço, representado pelo seu *controller* existe um repositório de dados que acessa o **Google Datastore**, responsável por armazenar as entidades de usuários, produtos e serviços.

A aplicação também utiliza o serviço **Logging** do Google Cloud Platform para geração de logs de operação da aplicação.

### 8.1.2 - Mecanismo de autenticação OAuth2

O mecanismo de autenticação utilizado pela aplicação é o OAuth2, que basicamente oferece um serviço para requisição de um token de acesso para um usuário pré-existente, a partir de suas credenciais. Com esse token então é possível acessar todos as operações dos serviços que exigem esse tipo de autenticação.

### 8.1.3 - Requisitando acesso de administrador

Como dito anteriormente, todos os serviços dessa aplicação utilizam o mecanismo de autenticação OAuth 2.0.

Para obter o token de acesso, a aplicação cliente deve fazer uma requisição ao servidor com as seguintes informações:

**Método:** POST

**URL:** <http://sales-provider.appspot.com/oauth/token>

**Cabeçalhos:**

- Content-Type: application/x-www-form-urlencoded
- Authorization: Basic c2lly29sYTptYXRpbGRI

**Corpo da mensagem:**

```
grant_type=password&username=matilde@siecola.com.br&password=matilde
```

Os valores dos campos **username** e **password** devem ser trocados para as credenciais de acesso do usuário que deseja obter o token.

A resposta a essa autenticação é o token de acesso no seguinte formato:

```
{
    "access_token": "13968c9d-e3ef-4b32-b119-b6d93f59963f",
    "token_type": "bearer",
    "refresh_token": "e567d8b8-def6-4f37-8b6d-d532a47a08ac",
    "expires_in": 3599,
    "scope": "read write"
}
```

O campo **access\_token** na mensagem de resposta será utilizado para acessar as demais operações do provedor que exigem um usuário autenticado.



O provedor de serviço de vendas já possui um usuário com papel de administrador, que não pode ser alterado nem apagado. Seu login é `matilde@siecola.com.br` e sua senha é `matilde`.

A seguir, um exemplo de como o Postman deve ser configurado para acessar essa operação e requisitar o token de acesso ao usuário ADMIN:

The screenshot shows the Postman interface with a request titled 'Get Token'. The method is set to 'POST' and the URL is 'https://sales-provider.appspot.com/oauth/token'. The 'Headers' tab is selected, showing three configured headers: 'Content-Type' with value 'application/x-www-form-urlencoded' and 'Authorization' with value 'Basic c2IY29sYTptYXRpbGRI'. Other tabs like 'Params', 'Authorization', 'Body', 'Pre-request Script', 'Tests', and 'Settings' are also visible.

#### Configurando o Postman para requisitar um token de acesso

Veja que o método POST e o endereço da operação para requisitar o token devem ser configurados na barra superior, bem como os *headers* necessários, na aba de mesmo nome.

O corpo da mensagem deve possuir as credenciais de acesso do usuário que se deseja obter o token, como mostra a figura a seguir. Depois de tudo configurado, pressione o botão **Send** para obter o token de acesso do usuário desejado.

▶ Get Token

POST https://sales-provider.appspot.com/oauth/token

Params Authorization Headers (11) **Body** Pre-request Script Tests

none form-data x-www-form-urlencoded raw binary GraphQL

1 grant\_type=password&username=matilde@siecola.com.br&password=matilde

Body Cookies Headers (15) Test Results

Pretty Raw Preview Visualize **BETA** JSON

```
1 {  
2     "access_token": "93805723-2950-42e1-821d-ec912b1775a5",  
3     "token_type": "bearer",  
4     "refresh_token": "bec986e7-b041-49d2-b51e-bf7c55921b40",  
5     "expires_in": 3599,  
6     "scope": "read write"  
7 }
```

Token requisitado



Essa operação será criada dentro da aplicação Android que será construída nesse capítulo, por isso é importante que o leitor entenda os passos que devem ser feitos, antes de codificá-los realmente.

### 8.1.4 - Criando um novo usuário

Para criar um novo usuário na aplicação, é necessário fazer uma requisição ao serviço de gerenciamento de usuários com as seguintes informações:

**Método:** POST

**URL:** <https://sales-provider.appspot.com/api/users>

**Permissão de acesso:** somente usuário com papel ADMIN. Isso significa que é necessário requisitar um token de acesso do usuário com papel ADMIN (Matilde, por exemplo), como descrito na seção anterior.

**Exemplo de corpo de requisição:**

```
{  
    "email": "doralice@siecola.com.br",  
    "password": "doralice",  
    "gcmRegId": null,  
    "lastLogin": null,  
    "lastGCMRegister": null,  
    "role": "USER",  
    "enabled": true  
}
```

**Exemplo de mensagem de resposta:**

```
{  
    "id": 5629499534213120,  
    "email": "doralice@siecola.com.br",  
    "password": "doralice",  
    "gcmRegId": null,  
    "lastLogin": null,  
    "lastGCMRegister": null,  
    "role": "USER",  
    "enabled": true  
}
```

A seguir, veja como o Postman deve ser configurado para a realização dessa operação:

▶ Create User

POST https://sales-provider.appspot.com/api/users

Params Authorization Headers (10) Body Pre-request Script

▼ Headers (2)

	KEY	VALUE
<input checked="" type="checkbox"/>	Content-Type	application/json
<input checked="" type="checkbox"/>	Authorization	Bearer f6e755dc-39e5-436c-8b2c-26edb647b8f1 33_postman_create_user_01

Veja que o token solicitado na seção anterior deve vir no cabeçalho Authorization.

E para finalizar a configuração do Postman para a criação de um usuário, basta preencher a aba Body, como no exemplo a seguir, sem o parâmetro id:

POST https://sales-provider.appspot.com/api/users

Params Authorization Headers (10) Body Pre-request Script

none form-data x-www-form-urlencoded raw binary

```

1 {  

2     "id": 5629499534213120,  

3     "email": "doralice@siecola.com.br",  

4     "password": "doralice",  

5     "gcmRegId": null,  

6     "lastLogin": null,  

7     "lastGCMRegister": null,  

8     "role": "USER",  

9     "enabled": true  

10 }

```

`34_postman_create_user_02`

Para demais operações do serviço de gerenciamento de usuários do provedor de serviços de vendas, consulte a documentação em seu [repositório](#)<sup>25</sup>.



Não será necessário implementar o cliente dessa operação na aplicação Android que será criada nesse capítulo. Ao invés disso, é necessário que o leitor crie seu próprio usuário para ser utilizado dentro do código Android. Isso facilita a implementação, concentrando os esforços no objetivo principal do capítulo.

### 8.1.5 - Requisitando todos os produtos

A operação para listar todos os usuários será a mais importante do provedor de serviços de vendas a ser utilizada pela aplicação Android, que será construída nesse capítulo. Com ela será possível obter todos os produtos que estão cadastrados e exibi-los em uma lista na tela para o usuário.

A seguir, as informações necessárias para acessar essa operação:

<sup>25</sup><https://github.com/siecola/GAESalesProvider>

**Método: GET**

URL: <https://sales-provider.appspot.com/api/products>

**Permissão de acesso:** qualquer usuário autenticado. Isso significa que é possível acessar essa operação com o token previamente requisitado, de qualquer usuário que esteja cadastrado no provedor de serviços de vendas.

**Exemplo de resposta:**

```
[  
  {  
    "id": 5707702298738688,  
    "name": "product1",  
    "description": "description1",  
    "code": "COD1",  
    "price": 10  
  },  
  {  
    "id": 5668600916475904,  
    "name": "product2",  
    "description": "description2",  
    "code": "COD2",  
    "price": 20  
  }  
]
```

Veja como Postman deve ser configurado para acessar essa operação:

▶ Get Products

GET https://sales-provider.appspot.com/api/products

Params Authorization Headers (8) Body Pre-request Script

▼ Headers (1)

	KEY	VALUE
<input checked="" type="checkbox"/>	Authorization	Bearer f6e755dc-39e5-436c-8b2c-26edb647b8f1
	Key	Value

Listando todos os produtos com o Postman

Veja que o mesmo cabeçalho `Authorization` deve ser preenchido com o token OAuth2 requisitado nas seções anteriores.

Para demais operações do serviço de gerenciamento de produtos do provedor de serviços de vendas, consulte a documentação em seu [repositório<sup>26</sup>](#).

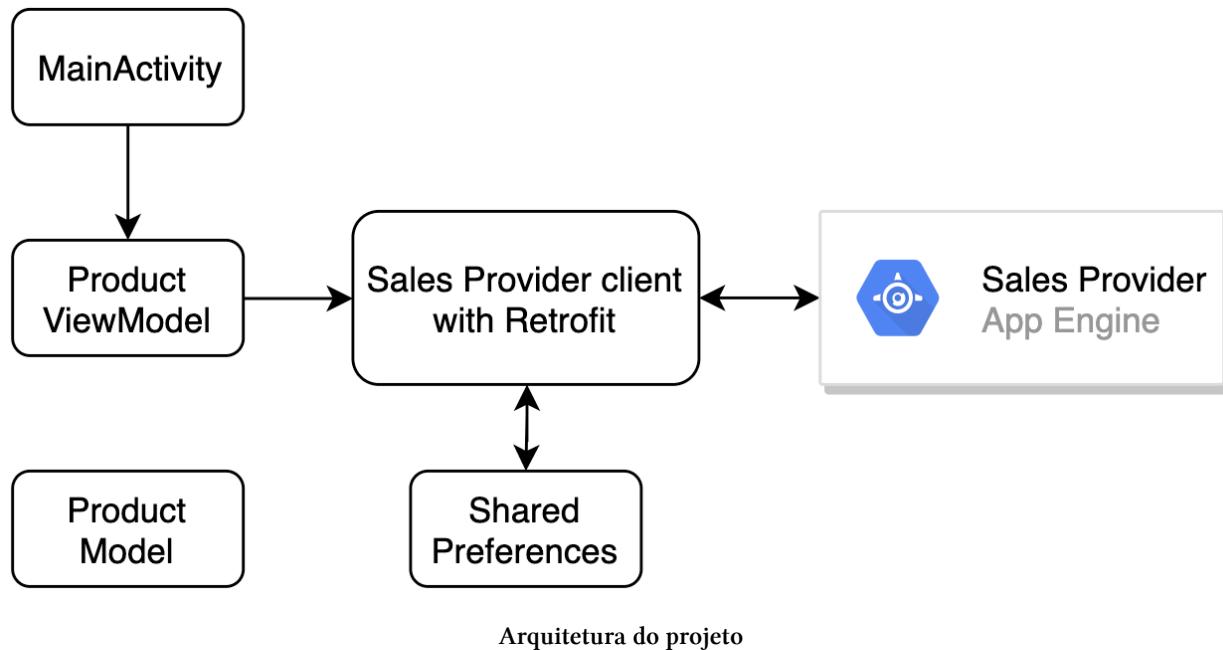
## 8.2 - Criando o novo projeto para consumir serviços REST

Para começar o desenvolvimento da aplicação cliente do provedor de serviços de vendas, crie um novo projeto no Android Studio com o nome de `AndroidProject02`, seguindo exatamente os mesmos passos descritos na seção 6.1 do capítulo 6.

### 8.2.1 - Diagrama do projeto

O projeto que começará a ser desenvolvido possui uma arquitetura em camadas, tendo cada uma sua responsabilidade bem definida. A seguir um esboço inicial do que será construído ainda nesse capítulo:

<sup>26</sup><https://github.com/siecola/GAESalesProvider>



Os detalhes de cada uma serão apresentados nas próximas seções, mas a ideia principal é:

- **SharedPreferences**: esse é um espaço que o sistema Android oferece para que as aplicações guardem informações não-voláteis, como estados e pequenas configurações. Ele será utilizado para armazenar o token de autenticação para que ele possa ser utilizado nas requisições a serem feitas ao provedor de serviços de vendas;
- **Sales Provider client**: essa camada será responsável por comunicar-se com o provedor de serviços de vendas utilizando requisições REST. Ela será implementada utilizando-se a biblioteca Retrofit;
- **Product ViewModel**: essa camada ainda evoluirá um pouco no próximo capítulo, mas ela começará a ser construída aqui para criar uma ligação entre o cliente REST do provedor de vendas e a camada de visualização de dados pelo usuário;
- **MainActivity**: por enquanto ela será utilizada apenas para invocar o *ViewModel* de produtos para disparar as requisição ao provedor de vendas. Obviamente, ela é a responsável pela interface gráfica com o usuário.
- **Product Model**: esse é o modelo que define a entidade que será buscada no provedor de serviços de vendas. Ela será utilizada para interpretar a resposta dele e transformar produtos em objetos dentro da aplicação.

## 8.2.2 - Configurando as dependências

Tendo o projeto criado, é hora de adicionar as bibliotecas que serão utilizadas para criar o que foi proposto nesse capítulo: **o cliente REST para acessar o provedor de serviços de vendas**.

As bibliotecas que serão adicionadas nessa seção são as que serão utilizadas nesse capítulo. Outras bibliotecas terão que ser adicionadas nos capítulos futuros.

Para começar, abra o arquivo `build.gradle` da aplicação e vá no final da seção `android`. Ainda dentro dessa seção, acrescente o seguinte trecho:

```
dataBinding {  
    enabled = true  
}  
androidExtensions {  
    experimental = true  
}
```

A primeira configuração se refere a habilitação do DataBind e ViewBinding para que seja possível referenciar componentes gráficos diretamente do código Kotlin, assim como foi feito no primeiro projeto.

A segunda configuração é para habilitar a utilização de extensões do Kotlin para Android, com funções auxiliares de classes existentes, por exemplo.

Na seção `dependencies` do mesmo arquivo, altere e adicione às seguintes dependências:

```
//Kotlin  
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
implementation 'androidx.core:core-ktx:1.2.0'  
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.0"  
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.0"
```

Essas são as dependências para adicionar as extensões do Kotlin ao projeto, bem como para acrescentar as bibliotecas de coroutines, que serão utilizadas no cliente REST, como será explicado mais adiante.

Por fim, acrescente as dependências finais que serão utilizadas nesse capítulo:

```
// ViewModel and LiveData  
implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"  
  
//Retrofit  
implementation "com.squareup.retrofit2:retrofit:2.5.0"  
implementation "com.squareup.retrofit2:converter-moshi:2.5.0"  
implementation "com.jakewharton.retrofit:retrofit2-kotlin-coroutines-adapter:0.9.2"  
  
//Moshi  
implementation "com.squareup.moshi:moshi:1.8.0"  
implementation "com.squareup.moshi:moshi-kotlin:1.8.0"
```

A dependência `lifecycle-extensions` é responsável por adicionar a capacidade do projeto trabalhar com `ViewModel` e `LiveData`, que serão detalhados mais adiante ainda nesse capítulo.

Os dois últimos conjuntos de bibliotecas são para adicionar o `Retrofit` e o `Moshi`, que serão utilizados para a construção do cliente REST do provedor de serviços de vendas.

Após ter adicionado todas as dependências, sincronize o projeto para que elas possam ser efetivamente adicionadas ao processo de compilação do projeto.

### 8.2.3 - Configurando as permissões no Android

O sistema Android possui um mecanismo bem elaborado e permissões as quais os aplicativos podem solicitar para ter acessos a recursos do sistema, como a comunicação via rede de dados, que é o caso do projeto que está sendo construído. Afinal, para acessar o provedor de serviços de vendas é necessário trafegar dados pela Internet.

A forma de solicitar permissões ao sistema Android varia de acordo com cada permissão e tem evoluído muito nas últimas versões do Android de forma a possibilitar maior segurança ao usuário. Existem algumas permissões que devem ser solicitadas ao usuário durante a execução do aplicativo, permitindo inclusive que ele possa desabilitar uma autorização prévia.

Para estabelecer uma comunicação através da rede de dados do aparelho, basta solicitar tal permissão no arquivo `AndroidManifest.xml` localizado no pacote `manifests` do projeto. Dentro desse arquivo, localize a tag `manifest` e adicione a seguinte linha dentro dela:

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

Isso dará permissão para que a aplicação possa trafegar dados pela Internet utilizando a rede de dados do aparelho. Sem essa permissão não será possível estabelecer comunicação com o provedor de serviços e vendas ou com qualquer outro *host* na Internet.

## 8.3 - Armazenando o token de autenticação nas Shared Preferences

Na próxima seção será construído o cliente REST para acessar o provedor de serviço de vendas. Durante o processo de comunicação com ele será necessário solicitar o token de autenticação de cada usuário, como foi demonstrado na seção 8.1 desse capítulo.

Esse token deverá ser armazenado pela aplicação Android, para que possa ser utilizado sempre que for necessário realizar requisições ao provedor de serviços de vendas. Isso evita que um novo token tenha que ser solicitado a todo momento, uma vez que ele possui validade pré-determinada e pode ser reutilizado se ainda não tiver expirado.

Para armazenar esse token, uma boa solução seria utilizar o `SharedPreference`, um espaço do sistema Android disponível para que aplicações possam persistir dados em forma de chave-valor e recuperar mesmo que a aplicação tenha sido fechada ou o aparelho tenha sido reiniciado.

### 8.3.1 - Expondo o contexto geral da aplicação

Para preparar a aplicação para utilizar o SharedPreferences, será necessário criar um *singleton* capaz de retornar o contexto global da aplicação. Esse contexto será utilizado pela classe que será criada na seção seguinte ao acessar o SharedPreferences do Android.

Uma das formas de se fazer tal tarefa é criar uma classe que estende de Application, sendo iniciada assim que a aplicação entra em execução e criando o *singleton* que possui a referência do contexto da aplicação.

Para começar, crie uma nova classe dentro do pacote principal do projeto, chamada MainApplication:

```
import android.app.Application  
import android.content.Context  
  
class MainApplication : Application() {  
  
}
```

Perceba que a nova classe MainApplication estende de Application, o que possibilita a requisição do contexto geral da aplicação.

Dentro dessa classe, crie um companion object, que é a forma de se criar um *singleton* em Kotlin, para conter a instância dessa classe:

```
companion object {  
    private var instance: MainApplication? = null  
  
}
```

Ainda dentro da classe, antes da declaração do companion object , crie o inicializar da classe para atribuir o valor inicial ao atributo instance:

```
init {  
    instance = this  
}
```

Dessa forma, quando essa classe for criada, durante a inicialização da aplicação, o atributo instance possuirá sua referência.

Para finalizar, crie o método getApplicationContext para retornar o contexto da aplicação dentro do companion object:

```

companion object {
    private var instance: MainApplication? = null

    fun getApplicationContext(): Context {
        return instance!!.applicationContext
    }
}

```

Esse método pode ser utilizado em qualquer parte da aplicação para requisição do contexto global da aplicação, o que será feito na seção seguinte para utilização do SharedPreferences.

Para que essa classe seja efetivamente criada e ter seu bloco inicializador executado quando a aplicação iniciar, é necessário alterar o arquivo `AndroidManifest.xml` com instruções específicas para tal. Isso pode ser feito adicionando-se a seguinte configuração na `tag application`:

```
android:name=".MainApplication"
```

### 8.3.2 - Criando a classe de utilidades

A classe de utilidades aqui referida é apenas uma forma de criar métodos para deixar o código mais organizado, evitando repetições de trechos espalhados pelo projeto. A ideia é fazer com que as funções para salvar e resgatar as informações do token de acesso do SharedPreferences possam ficar concentradas em único lugar.

Para começar, crie um novo pacote dentro do pacote principal da aplicação, chamado `util`. Dentro desse pacote, crie uma nova classe chamada `SharedPreferencesUtils`.

```

import android.content.Context
import android.content.SharedPreferences

private const val ACCESS_TOKEN = "access_token"
private const val EXPIRES_IN = "expires_in"

object SharedPreferencesUtils {
}

```



Lembre-se que a declaração é feita com `object` e não com `class`.

Repare que ao invés de utilizar o modificador `class`, antes do nome da classe em sua definição, foi utilizado o modificador `object`. Essa é a forma de se criar uma espécie de classe estática em Kotlin.

As duas constantes declaradas fora da classe servem para definir os nomes das chaves dos valores que serão salvos no Shared Preferences.

Agora crie o método `getSharedPreferences` dentro da classe `SharedPreferencesUtils`:

```
private fun getSharedPreferences(): SharedPreferences {
    val context = MainApplication.getApplicationContext()
    return context.getSharedPreferences("main", Context.MODE_PRIVATE)
}
```

Esse método é o responsável por retornar a instância do `SharedPreferences`, resgatado através do contexto global da aplicação utilizando o método `getApplicationContext` da recém-criada classe `MainApplication`. Essa instância será utilizada para ler e salvar os valores no `SharedPreferences`.

O segundo atributo na chamada `context.getSharedPreferences` define que somente essa aplicação tem acesso aos dados que serão salvos por ela, ou seja, esse conteúdo será privado.

Continuando, agora crie o método `saveToken`, que será chamado toda vez que um novo token de acesso for solicitado:

```
fun saveToken(accessToken: String, expiresIn: Int) {
    with(getSharedPreferences().edit()) {
        putString(ACCESS_TOKEN, accessToken)
        putInt(EXPIRES_IN, ((System.currentTimeMillis() / 1000) + expiresIn).toInt())
        commit()
    }
}
```

Repare que esse método recebe a `String` do token de acesso e também o número de segundos em que ele é válido. Essas informações serão salvas através dos métodos `putString` e `putInt` da referência do editor do `SharedPreferences` requisitada na primeira linha do método.

O tempo em que o token de acesso expira é calculado para refletir o momento atual no tempo, acrescido do valor em segundos recebido pelo método como parâmetro. Dessa forma será possível saber se um token está expirado ou não, quando ele for resgatado do `SharedPreferences`.

Na última instrução do método, é executado o comando `commit` do editor do `SharedPreference`, que efetivamente persiste as informações nele.

Para finalizar, crie o método `getAccessToken`, que deve buscar no `SharedPreferences` a existência de um token de acesso que ainda não tenha expirado:

```
fun getAccessToken(): String? {
    var accessToken: String? = null

    with (getSharedPreferences()) {
        if (contains(ACCESS_TOKEN) && contains(EXPIRES_IN)) {
            val expiresIn = getInt(EXPIRES_IN, 0)
            if (expiresIn > (System.currentTimeMillis() / 1000)) {
                accessToken = getString(ACCESS_TOKEN, "")
            }
        }
    }

    return accessToken
}
```

Veja que o que o método faz é buscar os valores do token de acesso e seu tempo de expiração pelas chaves em que eles foram salvos no método criado anteriormente. Dessa forma é possível saber se os valores existem e principalmente se o token ainda não expirou, comparando o tempo atual com o valor salvo no Shared Preferences.

Caso tudo esteja correto, o token de acesso é retornado para ser utilizado pela camada de serviço REST, que será criada na próxima seção.

## 8.4 - Criando o cliente REST com Retrofit

O Retrofit é uma biblioteca que facilita a criação de clientes REST em aplicações em Java e em Kotlin, que também pode ser utilizado em aplicações Android. Com ele é possível definir as requisições que devem ser feitas, incluindo parâmetros de URL, cabeçalhos HTTP, *payload* de requisição e resposta.

Com o Retrofit também é possível criar interceptadores de tal forma que um certo trecho de código seja executado antes de todas as requisições da aplicação, para, por exemplo, inserir cabeçalhos HTTP de forma dinâmica nas requisições.

### 8.4.1 - Definindo os modelos de dados

Antes de começar a criação do cliente REST com Retrofit, crie os modelos que serão utilizados nas requisições a serem realizadas pela camada de serviço: produtos e o token de acesso. Eles são necessários para representar os dados que foram mostrados na seção 8.1:

```
{
    "id": 5707702298738688,
    "name": "product1",
    "description": "description1",
    "code": "COD1",
    "price": 10
}
```

Essa é a resposta que o serviço de gerenciamento de produtos retorna para um produto. Por isso, é necessário definir um modelo no projeto que o represente. Para isso, crie um novo pacote chamado `network` dentro do pacote principal, para acomodar todas as classes referentes a essa camada.

Dentro desse novo pacote, crie uma classe chamada `Product`:

```
data class Product(
    var id: Long = 0,
    var name: String,
    var description: String,
    var code: String,
    var price: Double
)
```

Perceba que os atributos dessa classe possuem o mesmo nome e tipo dos atributos do JSON que representa um produto. Isso é feito para facilitar sua interpretação e conversão desse mesmo JSON para uma instância dessa classe.

Para o token de acesso o processo é o mesmo. Deve ser criado um modelo que represente seu modelo JSON:

```
{
    "access_token": "13968c9d-e3ef-4b32-b119-b6d93f59963f",
    "token_type": "bearer",
    "refresh_token": "e567d8b8-def6-4f37-8b6d-d532a47a08ac",
    "expires_in": 3599,
    "scope": "read write"
}
```

Dentro do mesmo pacote `network`, crie uma classe chamada `OauthTokenResponse`:

```

import com.squareup.moshi.Json

data class OAuthTokenResponse (
    @Json(name = "access_token")
    val accessToken: String,

    @Json(name = "expires_in")
    val expiresIn: Int
)

```

Nesse caso as anotações `@Json`, da biblioteca Moshi que foi adicionada ao projeto, são necessárias para definir os nomes dos campos desejados, para mantê-los dentro do padrão de nomenclatura de atributos. O Moshi também será utilizado como o interpretador de JSON pelo Retrofit, sendo responsável converter essas duas classes modelo de e para JSON.

## 8.4.2 - Criando a camada de serviço

Agora que os modelos que definem a comunicação com o provedor de vendas foram criados, é possível começar a implementação de sua camada de serviço. Para isso, crie uma nova classe no pacote network chamada `SalesApi`:

```

import com.jakewharton.retrofit2.adapter.kotlin.coroutines.CoroutineCallAdapterFactory
import com.squareup.moshi.Moshi
import com.squareup.moshi.kotlin.reflect.KotlinJsonAdapterFactory
import kotlinx.coroutines.Deferred
import okhttp3.OkHttpClient
import retrofit2.Call
import retrofit2.Retrofit
import retrofit2.converter.moshi.MoshiConverterFactory
import retrofit2.http.*
import java.util.concurrent.TimeUnit

private const val BASE_URL = "https://sales-provider.appspot.com"

object SalesApi {
}

```

O modificador `object` permite que seus métodos ou atributos possam ser acessados sem a necessidade da criação de uma instância dessa classe.

Repare que também foi criada uma constante para definir o endereço base do provedor de serviços de vendas.

Nesse mesmo arquivo, antes da definição da classe SalesApi, crie um objeto do interpretador JSON a ser utilizado, o Moshi:

```
private val moshi = Moshi.Builder()
    .add(KotlinJsonAdapterFactory())
    .build()
```

Ele será utilizado para definir a criação do cliente Retrofit.

Em seguida, crie um objeto para definir as configurações do cliente HTTP a ser utilizado pelo Retrofit:

```
private val okHttpClient = OkHttpClient.Builder()
    .connectTimeout(60, TimeUnit.SECONDS)
    .readTimeout(60, TimeUnit.SECONDS)
    .addInterceptor(OAuthTokenInterceptor())
    .authenticator(OAuthTokenAuthenticator())
    .build()
```

A ideia de utilizar um cliente customizado no Retrofit é justamente para ter a possibilidade de definir as configurações realizadas no trecho anterior, como tempos de *timeout* e interceptadores de requisições, que serão criados e detalhados a seguir.

Agora crie a instância do Retrofit, de acordo com o seguinte trecho:

```
private val retrofit = Retrofit.Builder()
    .baseUrl(BASE_URL)
    .addConverterFactory(MoshiConverterFactory.create(moshi))
    .addCallAdapterFactory(CoroutineCallAdapterFactory())
    .client(okHttpClient)
    .build()
```

Veja que aqui são definidas algumas configurações, como:

- **Endereço base:** definido pela constante BASE\_URL;
- **Interpretador JSON:** aqui será utilizado o Moshi com as configurações realizadas anteriormente;
- **Adaptador para chamadas de retorno:** nesse momento é definido que será utilizado coroutines para implementação das chamadas de retorno, que será detalhado mais adiante.
- **Cliente HTTP:** aqui será utilizado a instância do OkHttpClient, criada anteriormente com suas configurações de *timeout* e interceptadores.

Agora que todas as configurações já foram estabelecidas, pode-se criar a interface que define as operações da camada de serviço. Por isso, ainda dentro do arquivo `Sales ApiService` e antes da definição da classe `SalesApi`, crie a interface `Sales ApiService`, como no trecho a seguir:

```
interface Sales ApiService {  
}
```



Caso tenha dúvidas sobre como o código deve ficar, consulte o projeto nos extras que podem ser baixados juntos com o livro.

Dentro dessa interface, defina seu primeiro método, para consultar a lista de produtos do provedor de serviços de vendas:

```
@GET("api/products")  
fun getProducts(): Deferred<List<Product>>
```

A primeira linha, com a anotação `@GET`, define qual deve ser o verbo HTTP da operação. Ela recebe como parâmetro o endereço da operação, que será concatenado pelo endereço base que foi definido no início do arquivo `Sales ApiService`.

Aqui está implícito que o formato dos dados trafegados serão JSON.

A segunda linha define o método `getProducts`, que não recebe nenhum parâmetro, o que é óbvio visto que a intenção é buscar todos os produtos sem nenhum filtro. Porém, o tipo de retorno é explícito pela seguinte definição:

```
Deferred<List<Product>>
```

Analizando de dentro pra fora, essa declaração define primeiramente que a operação retornará uma lista de objetos do tipo `Product`, modelo esse que foi definido na seção anterior, ou seja, o retorno representará todos os objetos retornados pela operação correspondente do provedor de serviços de vendas.

O tipo `Deferred` na declaração do tipo de retorno faz parte das `coroutines` em `Kotlin`, que serão detalhadas na seção 8.8 desse capítulo. Porém, é interessante ressaltar aqui que chamadas a serviços REST com o `Retrofit` podem ter dois comportamentos:

- **Blocking:** essa forma faz com que o processo que invocou a execução de um método para realizar uma chamada REST fique bloqueado até que a requisição se complete totalmente;
- **Non-blocking:** essa maneira faz com que o processo que fez a chamada REST seja liberado imediatamente, deixando a possibilidade do resultado da requisição ser consultado posteriormente.

No Android, nenhuma requisição de rede deve ser feita dentro da *thread* que cuida da atualização da interface gráfica do usuário. Isso faz com que mecanismos assíncronos tenham que ser utilizados para a realização de requisições REST, como será visto mais adiante nas demais seções. Nesse sentido, a junção do Retrofit com coroutines em Kotlin é perfeita para atingir esse critério de execução de processos assíncronos, sem interferir na experiência do usuário enquanto ele interage com a tela.

O segundo método a ser criado dentro da interface Sales ApiService é a definição da operação para buscar o token de acesso do usuário, como no trecho a seguir:

```
@POST("oauth/token")
@FormUrlEncoded
fun getToken(
    @Header("Authorization") basicAuthentication: String,
    @Field("grant_type") grantType: String,
    @Field("username") username: String,
    @Field("password") password: String
): Call<OauthTokenResponse>
```

A primeira linha define que o verbo HTTP POST deve ser utilizado, com a URL oauth/token, como foi demonstrado na seção 8.1 desse capítulo.

A segunda linha define o formato do dado a ser trafegado no corpo da mensagem, que segue esse padrão:

```
grant_type=password&username=<username>&password=<password>
```

Repare também que esse método possui parâmetros de entrada, bem como anotações em cada um deles:

- **@Header("Authorization")**: essa anotação define que o valor desse parâmetro de entrada será colocado no cabeçalho HTTP com o nome Authorization, que é necessário para obtenção do token de acesso;
- **@Field("grant\_type")**: assim como os demais, essa anotação define que o valor do parâmetro de entrada será colocado no atributo de mesmo nome no corpo da requisição;
- **@Field("username")**: define o valor do atributo username do corpo da requisição;
- **@Field("password")**: define o valor do atributo password do corpo da requisição.

Por último, veja que o tipo de retorno do método é definido como:

```
Call<OauthTokenResponse>
```

OauthTokenResponse é o modelo criado na seção anterior para definir o formato da resposta à solicitação de um novo token de acesso, como foi visto na seção 8.1 desse capítulo:

```
{
    "access_token": "13968c9d-e3ef-4b32-b119-b6d93f59963f",
    "token_type": "bearer",
    "refresh_token": "e567d8b8-def6-4f37-8b6d-d532a47a08ac",
    "expires_in": 3599,
    "scope": "read write"
}
```

Porém, ainda há a definição `Call` no tipo de retorno desse método. Essa é a forma de especificar ao Retrofit que a chamada deve ser *blocking*, ou seja a execução é síncrona e o processo que o invocar deverá aguardar até que a requisição seja feita para obter seu resultado.

Utilizar a maneira síncrona para requisitar o token de acesso é conveniente dado o ponto que ele será chamado, que será dentro de interceptador de requisições do Retrofit, como será detalhado na próxima seção.

Para finalizar, defina o atributo `retrofitService` dentro da classe `SalesApi`, no final do arquivo, de acordo com o seguinte trecho:

```
val retrofitService: Sales ApiService by lazy {
    retrofit.create(Sales ApiService::class.java)
}
```

Repare que ele é do tipo `Sales ApiService`, a mesma interface que foi criada nesse arquivo com as definições das operações REST do provedor de serviços de vendas. Dessa forma, de qualquer ponto do projeto, bastará fazer uma chamada como a seguir, para consultar todos os produtos:

```
SalesApi.retrofitService.getProducts()
```

A forma com que a classe `SalesApi` foi criada, com o modificador `object`, não é necessário criar uma instância sua a todo o momento de sua utilização. Da mesma forma, o atributo `retrofitService` será inicializado na primeira vez que ele for acessado, criando uma instância de `Sales ApiService`.

Para finalizar a camada de serviço REST do provedor de serviços, é necessário criar os interceptadores de todas as requisições, como será visto nas próximas duas seções desse capítulo.

## 8.5 - Requisitando o token OAuth

Na definição do cliente HTTP para o Retrofit, foi criada uma instância de `OkHttpClient`. Um dos parâmetros na construção dessa instância é um interceptador para requisitar o token de acesso:

```
.authenticator(OauthTokenAuthenticator())
```

A ideia é que esse interceptador seja invocado toda vez que alguma requisição ao provedor de serviços de vendas retorne HTTP 401 Unauthorized. Isso pode acontecer se a requisição não possuir o cabeçalho HTTP Authorization, como descrito na seção 8.1.5 desse capítulo. Obviamente, isso acontecerá na primeira vez em que a aplicação for executada, pois nenhum token de acesso ainda terá sido requisitado.

Porém, quando um token de acesso requisitado já tiver expirado, o provedor de serviços de vendas também retornará HTTP 401 Unauthorized, fazendo com que esse mesmo interceptador seja invocado novamente.

Portanto, a ideia por trás desse interceptador é fazer com que ele solicite um token de acesso, sempre que qualquer condição faça com a que uma requisição prévia não tenha sido autorizada pelo provedor de serviços de vendas. Para isso, crie uma nova classe chamada OauthTokenAuthenticator dentro do pacote network, estendendo de Authenticator, uma interface do OkHttp apropriada à estratégia definida aqui:

```
import android.util.Log
import br.com.siecola.androidproject02.util.SharedPreferencesUtils
import okhttp3.Authenticator
import okhttp3.Request
import okhttp3.Response
import okhttp3.Route

private const val TAG = "OauthTokenAuthenticator"

class OauthTokenAuthenticator() : Authenticator {
```

```
}
```

A constante definida com o nome de TAG será utilizada na geração de logs por essa classe.

Repare que o Android Studio acusa que há um erro nessa classe. Isso porque a interface Authenticator obriga a declaração da função authenticate:

```
override fun authenticate(route: Route?, response: Response): Request? {
```

```
}
```

Essa é a função que será chamada pelo Retrofit toda vez que uma requisição prévia for não autorizada pelo provedor de serviços de vendas, fazendo com que a mesma requisição possa ser refeita novamente, agora com o token de acesso. Logo, os passos a serem realizados dentro dessa função são os seguintes:

- Buscar por um token de acesso previamente salvo no SharedPreferences;
- Caso não haja nenhum token de acesso salvo no SharedPreferences, um novo deve ser requisitado ao provedor de serviços de vendas;
- Se um novo token de acesso foi requisitado com sucesso, ele deverá ser salvo no SharedPreferences para ser utilizado em uma próxima requisição;
- De posse do token de acesso, é necessário inseri-lo na requisição atual, através do cabeçalho HTTP Authorization, na tentativa de que ele agora seja autorizada pelo provedor de serviços de vendas.

Para implementar esse passos, comece com a criação da função para solicitar um novo token, dentro da classe:

```
private fun retrieveNewToken(): OAuthTokenResponse {  
    Log.i(TAG, "Retrieving new token")  
    return SalesApi.retrofitService.getToken(  
        "Basic c211Y29sYTptYXRpbGR1",  
        "password",  
        "matilde@siecola.com.br",  
        "matilde"  
    ).execute().body()!!  
}
```

Essa função deverá retornar um objeto do tipo OAuthTokenResponse, que é a representação do token de acesso fornecido pelo provedor de serviços de vendas.

A primeira linha dessa função é uma chamada ao mecanismo de logs do Android, que será utilizado para acompanhar o funcionamento dessa camada através do Logcat, um painel que mostra todos os logs do sistema, inclusive das aplicações em execução.

A segunda linha é essencialmente a chamada ao serviço criado na seção anterior, através de sua operação para requisitar um novo token, com os seguintes parâmetros, definidos na ordem em que aparecem na chamada da função:

- **Autenticação para requisição do token:** o provedor de serviços de vendas requer uma autenticação do cliente (não do usuário), para fornecer um token. Aqui está sendo utilizado uma credencial fixa em Base64;
- **Tipo da credencial:** a credencial a ser enviada é a senha do usuário, por isso aqui a String password está sendo fornecida;
- **Username:** username do usuário que foi criado. Aqui substitua pelo username do usuário que foi criado com o Postman, conforme instruções da seção 8.1.4 desse capítulo;
- **Password:** senha do usuário que foi criado. Aqui substitua pela senha do usuário que foi criado com o Postman, conforme instruções da seção 8.1.4 desse capítulo.



As credencias do usuário, que aqui estão colocadas de forma fixa, poderiam ser buscadas no Shared Preferences e ser cadastradas utilizando uma tela de login, por exemplo. Porém a ideia aqui foi deixar essa parte de lado para deixar a explicação mais direta ao ponto proposto no capítulo.

Agora é possível implementar a função authenticate para realizar os passos definidos acima:

```
override fun authenticate(route: Route?, response: Response): Request? {
    val accessToken: String?

    val token = retrieveNewToken()
    accessToken = token.accessToken
    SharedPreferencesUtils.saveToken(token.accessToken, token.expiresIn)

    return response.request().newBuilder()
        .header("Authorization", "Bearer ${accessToken}")
        .build()
}
```

Essa função utiliza o SharedPreferencesUtils, classe criada anteriormente, para buscar um token pré-existente no Shared Preferences. Porém, caso ele não exista ou esteja expirado, um novo é solicitado através da função retrieveNewToken dessa classe.

De posse do token de acesso, ele é inserido na requisição através do cabeçalho HTTP Authorization, na última linha da função.

## 8.6 - Interceptando requisições para inserir o token OAuth

Ainda durante a definição do cliente OkHttpClient no arquivo Sales ApiService, foi adicionado um outro interceptador:

```
.addInterceptor(OauthTokenInterceptor())
```

Ao contrário do que foi criado na seção anterior, que só entrava em execução caso a resposta do provedor de serviço de vendas fosse HTTP 401 Unauthorized, esse por sua vez intercepta todas as requisições para inserir o token de acesso, caso ele exista e não esteja expirado.

Com essa implementação, tendo os dois interceptadores, é possível requisitar um token de acesso assim que a aplicação é executada pela primeira vez e utilizá-lo nas requisições futuras, até que ele expire e um novo possa ser solicitado. Isso evita que a aplicação Android fique requisitando um novo token toda vez que precisa acessar o provedor de serviços de vendas.

Para começar a implementação desse novo interceptador, crie uma nova classe chamada OauthTokenInterceptor dentro do pacote network, estendendo da interface Interceptor do OkHttpClient:

```
import android.util.Log
import br.com.siecola.androidproject02.util.SharedPreferencesUtils
import okhttp3.Interceptor
import okhttp3.Response

private const val TAG = "OauthTokenInterceptor"

class OauthTokenInterceptor() : Interceptor {
```

```
}
```

A interface Interceptor obriga a implementação da seguinte função:

```
override fun intercept(chain: Interceptor.Chain): Response {
```

```
}
```

Ela é a que será invocada quando o interceptador entrar em ação, logo é aqui que o token de acesso deve ser resgatado do SharedPreferences, caso ele exista e ainda esteja válido, e inserido na requisição para que ela possa ser autorizada pelo provedor de serviços de vendas:

```
override fun intercept(chain: Interceptor.Chain): Response {
    var request = chain.request()
    val accessToken = SharedPreferencesUtils.getAccessToken()

    if (accessToken != null) {
        Log.i(TAG, "Using the existing token")
        request = request.newBuilder()
            .addHeader("Authorization", "Bearer ${accessToken}")
            .build()
    }

    return chain.proceed(request)
}
```

Veja que o token é lido do SharedPreferences e caso exista, o que é feito através da instrução accessToken.let, é inserido no cabeçalho Authorization.

E isso então finaliza a construção de toda a camada de serviço da aplicação para acessar o provedor de serviços de vendas.

## 8.7 - Criando o ViewModel de produtos

Agora que a camada de rede teve suas fundações construídas, pode-se passar para a próxima camada, que é a de visualização, ou **ViewModel**, como no diagrama a seguir:

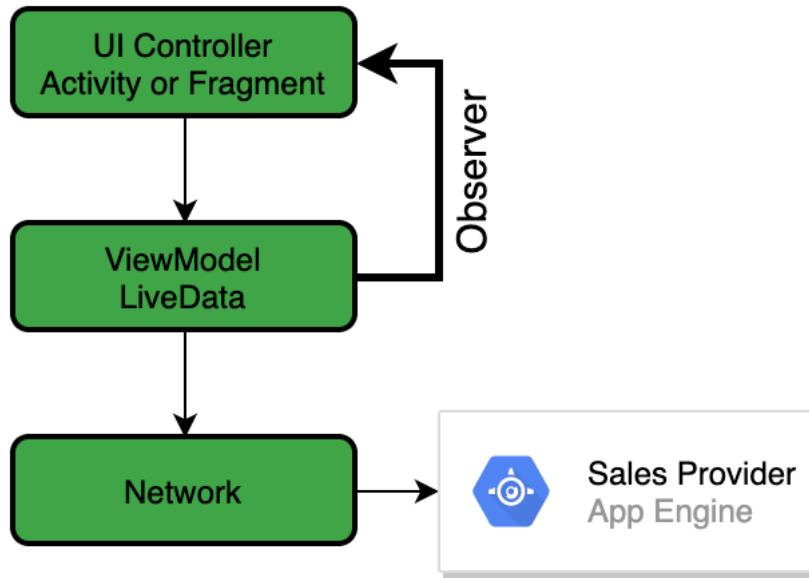


Diagrama da aplicação com ViewModel

Essa arquitetura permite que as camadas de interface de usuário e de rede fiquem isoladas, sem nenhum tipo de dependência. O que significa que a implementação da camada de rede, seja pela utilização do Retrofit ou de outra biblioteca, não interfere na exposição dos dados ao usuário.

Isso significa que as operações assíncronas realizadas pela camada de rede podem acontecer sem impactar a *thread* principal da interface gráfica, um requisito mais que necessário para manter a experiência do usuário o mais fluída possível.

Outra grande vantagem da utilização desse tipo de arquitetura é o desacoplamento entre a camada da interface gráfica (**UI Controller**) e a de visualização (**ViewModel**), fazendo que essa segunda não contenha nenhuma referência de seu *controller*.

A mecânica entre as duas camadas superiores se baseia em um *listener* criado pela camada UI Controller, que fica observando as mudanças nas entidades desejadas, ou seja, ela é notificada pela camada ViewModel sobre alguma alteração de dados ou estados. Uma vez notificada, a camada UI Controller exibe os dados ou toma alguma ação frente a alguma mudança de estado.

A técnica utilizada pela camada ViewModel para permitir que seus dados e estados possam ser observados utiliza o conceito de `LiveData`, que encapsula objetos e emite sinais quando eles sofrem algum tipo de alteração. Dessa forma, uma outra camada pode observar a emissão de tais sinais e reagir de acordo com um deles de forma separada.

Com a camada UI Controller isolada, ela fica apenas responsável pela criação e manutenção da

interface gráfica ao usuário. Essa camada será mais trabalhada nos próximos capítulos. A ideia dessa e das próximas seções nesse capítulo é construir as fundações da camada `ViewModel`, de modo que ela fique funcional para pelo menos requisitar a lista de produtos ao provedor de serviços de vendas.

Para começar, crie um novo pacote chamado `product` e uma nova classe dentro dele chamada `ProductListViewModel`, que será a implementação da camada `ViewModel`, até o momento:

```
import android.util.Log
import androidx.lifecycle.ViewModel
import br.com.siecola.androidproject02.network.SalesApi
import kotlinx.coroutines CoroutineScope
import kotlinx.coroutines Dispatchers
import kotlinx.coroutines Job
import kotlinx.coroutines launch

private const val TAG = "ProductListViewModel"

class ProductListViewModel : ViewModel() {

}
```

Veja que a nova classe estende de `ViewModel`, fazendo com que ela possa ter dados e estados observáveis por uma `Activity` ou `Fragment`.

No momento, essa classe não terá nenhum dado ou estado a ser observado pela `MainActivity`, pois a ideia é apenas dar início ao processo de requisição da lista de produtos, a ser construído na próxima seção. Nos próximos capítulos ela ganhará dados e eventos que serão observados pela camada `UI Controller`.



Os próximos capítulos trarão mais detalhes sobre o conceito de `ViewModel`.

## 8.8 - Requisitando a lista de produtos com coroutines

Como dito anteriormente, o processo para acessar uma operação de um serviço REST no Android deve ser feito de forma assíncrona, pois existem vários fatores que podem contribuir para que tal requisição demore mais tempo que o esperado, como:

- Atrasos na rede de comunicação;
- Perda de pacotes e retentativas de conexão;
- Carga elevada de requisições no servidor.

Por esses motivos que no Android toda e qualquer comunicação via rede deve ser feita fora da *thread* da interface gráfica, pois nenhuma operação bloqueante deve afetar a experiência do usuário.

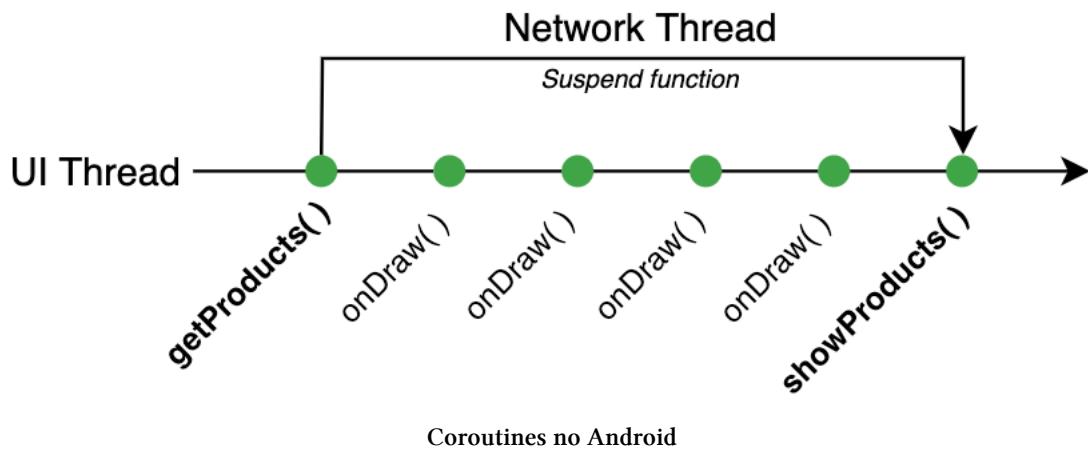
Mas, como fazer com que uma outra *thread* execute de forma assíncrona e ainda consiga atualizar a interface do usuário com o resultado da requisição? Ao longo dos anos muitas técnicas foram sendo utilizadas para atender a esses requisitos, incluindo tarefas assíncronas, que embora funcionem muito bem e têm bibliotecas interessantes, talvez não possuam um código tão simples de ser entendido e mantido.

Com a introdução dos componentes arquiteturais do sistema Android, chamados agora de **Android Architecture Components**, sob a *suite* **Android Jetpack**, como os já citados **ViewModel** e **LiveData**, ficou muito mais elegante trabalhar com operações assíncronas, de forma geral, embora tais componentes não sejam exclusivos para isso.

E para facilitar ainda mais o trabalho de se trabalhar com tarefas assíncronas no Android, é possível utilizar um recurso muito interessante, embora não exclusivo, do Kotlin: **coroutines**.

Coroutines é uma forma de se trabalhar com operações assíncronas ou não-bloqueantes, através do suporte de bibliotecas do Kotlin, delegando toda a implementação trabalhosa de controle de processos e contextos a essas bibliotecas. Isso obviamente deixa o código dos projetos que os usam muito mais fáceis de se entender e de se manter.

Basicamente a ideia é que operações assíncronas sejam executadas fora da *thread* principal de execução, que no caso do Android essa é a *thread* da interface do usuário, mas que rodem dentro de um escopo pré-determinado, permitindo que as funções que executam tais operações bloqueantes possam ficar **suspensas** até que essa operação acabe e tenha seu resultado. Veja a seguir um diagrama resumindo isso:



Do ponto de vista da UI Thread, a requisição é iniciada através da chamada da função `getProducts()`, que por sua vez chama uma função para efetivamente fazer a requisição em uma *thread* separada, e que pode ficar suspensa até que toda a operação seja concluída. Ao mesmo tempo, a UI Thread pode continuar atualizando a interface gráfica e reagindo a eventos do usuário normalmente.

Quando a requisição é completada, a função suspensa entra em execução novamente, com possibilidade de atualizar a interface gráfica do usuário, aqui representada pela função `showProducts()`.



O próximo capítulo irá implementar a atualização da interface gráfica do usuário com a lista de produtos, utilizando o conceito de `LiveData`.

Além disso, também é desejável que a função suspensa, que executa operação bloqueante, possa ser cancelada em algum momento, seja por ação do usuário ou do próprio sistema.

Para continuar a implementação da classe `ProductListViewModel` com os conceitos mencionados acima, crie os dois atributos privados a seguir:

```
private var viewModelJob = Job()
private val coroutineScope = CoroutineScope(viewModelJob + Dispatchers.Main)
```

O primeiro é a representação da tarefa que será executada pela `coroutine`, dentro do escopo definido pelo segundo atributo privado. Nesse caso o escopo faz parte da `UI Thread`, especificado por `Dispatchers.Main`, pois é daí que a requisição assíncrona será iniciada.

A tarefa a ser executada, definida pelo primeiro atributo, pode ser cancelada a qualquer momento, seja pelo sistema Android ou pelo próprio usuário. A melhor forma de fazer isso de forma segura e principalmente no momento exato, é quando o próprio `ViewModel` é destruído, o que é informado pela execução da função `onCleared()`:

```
override fun onCleared() {
    super.onCleared()
    viewModelJob.cancel()
}
```

Quando essa função for chamada, significa que a instância desse `ViewModel` já não é mais necessária e será destruída, logo é um bom momento para cancelar a tarefa que possivelmente está em execução.

Para efetivamente implementar a que dará início à tarefa assíncrona, crie o método `getProducts()`, como no trecho a seguir:

```
private fun getProducts() {
    Log.i(TAG, "Preparing to request products list")
    coroutineScope.launch {
        var getProductsDeferred = SalesApi.retrofitService.getProducts()
        try {
            Log.i(TAG, "Loading products")

            var productsList = getProductsDeferred.await()

            Log.i(TAG, "Number of products ${productsList.size}")
        } catch (e: Exception) {
```

```
        Log.i(TAG, "Error: ${e.message}")
    }
}
Log.i(TAG, "Products list requested")
}
```

Veja que o atributo `coroutineScope` é utilizado para lançar a  `coroutine` através da instrução `launch`. O código entre as chaves é então executado em uma *thread* separada, possibilitando que a UI Thread, que chamou essa função, possa continuar sua execução sem ser bloqueada.

Repare que logo na primeira linha da execução da  `coroutine` há a seguinte instrução:

```
var getProductsDeferred = SalesApi.retrofitService.getProducts()
```

Que é a chamada da camada de serviço para acessar o provedor de serviços de vendas, para requisitar a lista de produtos. Veja que o objeto retornado é do tipo `Deferred`, como foi definido na interface com o Retrofit:

```
fun getProducts(): Deferred<List<Product>>
```

Esse objeto permite a chamada à função suspensa `await()`, que aguarda a sua execução terminar para retornar o resultado, que é a lista de produtos:

```
var productsList = getProductsDeferred.await()
```

Dessa forma, `productsList` é a a lista propriamente dita de produtos, do tipo `List<Product>`.

Aqui é apenas gerado um log com a quantidade de produtos, para demonstrar que tudo ocorreu corretamente, mas no próximo capítulo essa lista será exibida ao usuário.



As mensagens de log nessa função foram colocadas para facilitar o entendimento de seu funcionamento, como será explicado na próxima seção desse capítulo.

Porém, para fazer com que a função `getProducts()` seja efetivamente invocada, é necessário chamá-la no inicializador da classe `ProductListViewModel`, por isso crie esse bloco logo após a declaração dos dois atributos privados:

```
init {
    getProducts()
}
```

Isso faz com que os produtos sejam requisitados logo quando a instância de `ProductListViewModel` for criada.

A criação de `ProductListViewModel` deve estar atrelada a uma tela, porém essa aplicação possui apenas uma até o momento, que é a `MainActivity`. Isso será alterado no próximo capítulo, com a inserção de um fragmento para exibir a lista de produtos.

Para efeitos de demonstração do que foi criado, vá até a classe `MainActivity` e adicione o seguinte trecho de código no final da função `onCreate`:

```
val viewModel = ViewModelProviders.of(this).get(ProductListViewModel::class.java)
```

Essa instrução cria uma instância de `ProductListViewModel` ou reaproveita uma já previamente criada.

Como dito anteriormente, a criação de uma instância de `ProductListViewModel` faz com que seu bloco inicializador seja executado:

```
init {
    getProducts()
}
```

Logo a `coroutine` para buscar todos os produtos do provedor de serviços é executada.

A seção seguinte irá analisar a execução do que foi feito até o momento através dos logs que foram espalhados pelo código do projeto.

## 8.9 - Analisando o comportamento da aplicação através de logs

Para testar a aplicação, execute-a em um emulador ou dispositivo real que tenha acesso à Internet. O que se espera é que o log com a quantidade de produtos existentes no provedor de serviços de vendas apareça no Logcat, aba localizada no parte inferior esquerda do Android Studio:

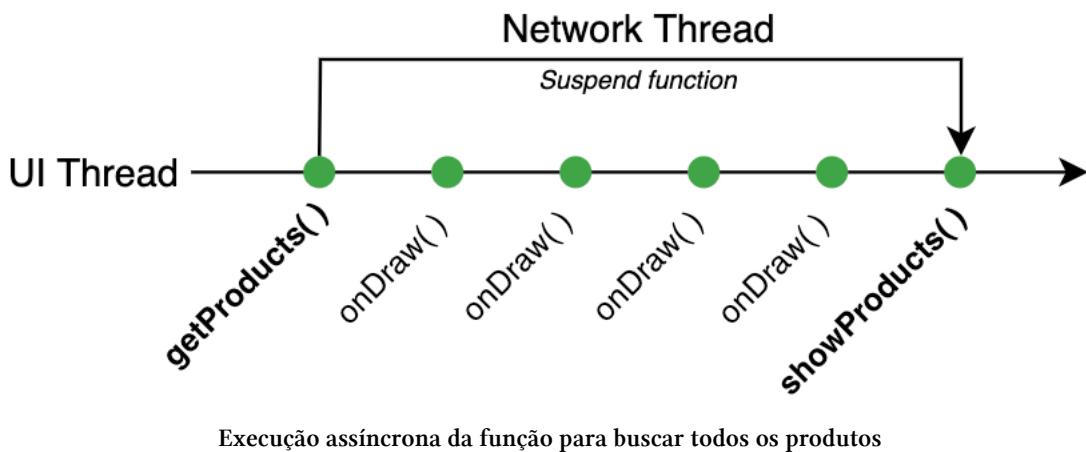


Para facilitar a visualização das mensagens de log, filtре-as utilizando o nível `Info` e selecione apenas as que tiverem a `tag` `I/ProductListViewModel`, como mostra a figura anterior.

Veja que o primeiro log que aparece é o que foi colocado logo no início da função `getProducts()`, antes da coroutine ser iniciada. Logo a seguir, a mensagem que aparece é o log que foi colocado no fim dessa mesma função, provando que a UI Thread não foi bloqueada com a execução da coroutine.

O terceiro log se refere ao momento anterior à chamada da função suspensa `await`, que tem sua execução bloqueada por alguns segundos até obter a resposta do provedor de serviços de vendas com a lista de produtos, expressa na última mensagem desse trecho de logs.

Essas mensagens de log comprovam o diagrama a seguir, que já foi citado anteriormente, sobre a execução assíncrona da função responsável por requisitar os produtos, bem como a liberação imediata da UI Thread.



Também é possível observar, na primeira execução da aplicação, os logs relativos a requisição do token de acesso, através dos interceptadores de requisição que foram construídos:

```
br.com.siecola.androidproject02 ▾ Info ▾ I/Oauth
269/br.com.siecola.androidproject02 I/OauthTokenInterceptor: Using the existing token
269/br.com.siecola.androidproject02 I/OauthTokenAuthenticator: Retrieving new token
269/br.com.siecola.androidproject02 I/OauthTokenInterceptor: Using the existing token
Requisitando o token de acesso
```

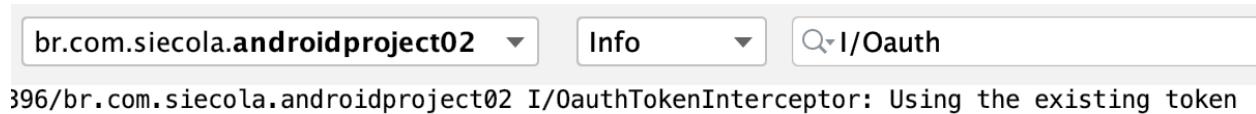
Da mesma forma, para facilitar a visualização somente dos logs relativos a essa operação, configure o Logcat para exibir somente mensagens do tipo Info, filtrando pela tag I/Oauth.

Veja que primeiro a aplicação tenta utilizar um token já existente em `OauthTokenInterceptor`, mas como é a primeira vez que ela entra em execução, obviamente não existe nenhum. Então ele é requisitado ao provedor de serviços de vendas, através de `OauthTokenAuthenticator`, que o salva no `SharedPreferences` para ser utilizado posteriormente.

Numa nova tentativa realizada automaticamente pelo Retrofit, a mesma chamada para buscar todos os produtos é refeita, mas agora o token de acesso existe no `SharedPreferences` e pode ser utilizado para autenticar a requisição.

Se a aplicação for fechada e executada novamente, o token de acesso já existirá no `SharedPreferences`,

logo a primeira tentativa de fazer a requisição ao provedor de serviços de vendas já o conterá em seu cabeçalho `Authorization`, logo não é necessário refazê-la, tampouco solicitar outro token:



#### Reaproveitando o token de acesso

É possível ensaiar a condição inicial da aplicação, para que ela possa solicitar um novo token, apagando o aplicativo do aparelho ou limpando seus dados na seção de configurações do Android. E isso conclui a implementação desse capítulo!



É importante lembrar que o código do projeto completo está disponível nos extras que pode ser baixado junto com o livro.

## 8.10 - Conclusão

Nesse longo capítulo foi possível iniciar alguns conceitos avançados sobre Android, como:

- Retrofit para realizar requisições HTTP;
- SharedPreferences para armazenamento do token de acesso;
- ViewModel e LiveData, partes do Android Architecture Components;
- Coroutines para execução de tarefas assíncronas.

No próximo capítulo, esse projeto será continuado para apresentar a lista de produtos do provedor de serviços de vendas na interface gráfica do usuário. Será utilizando LiveData para popular tal lista, utilizando o componente Recyclerview.

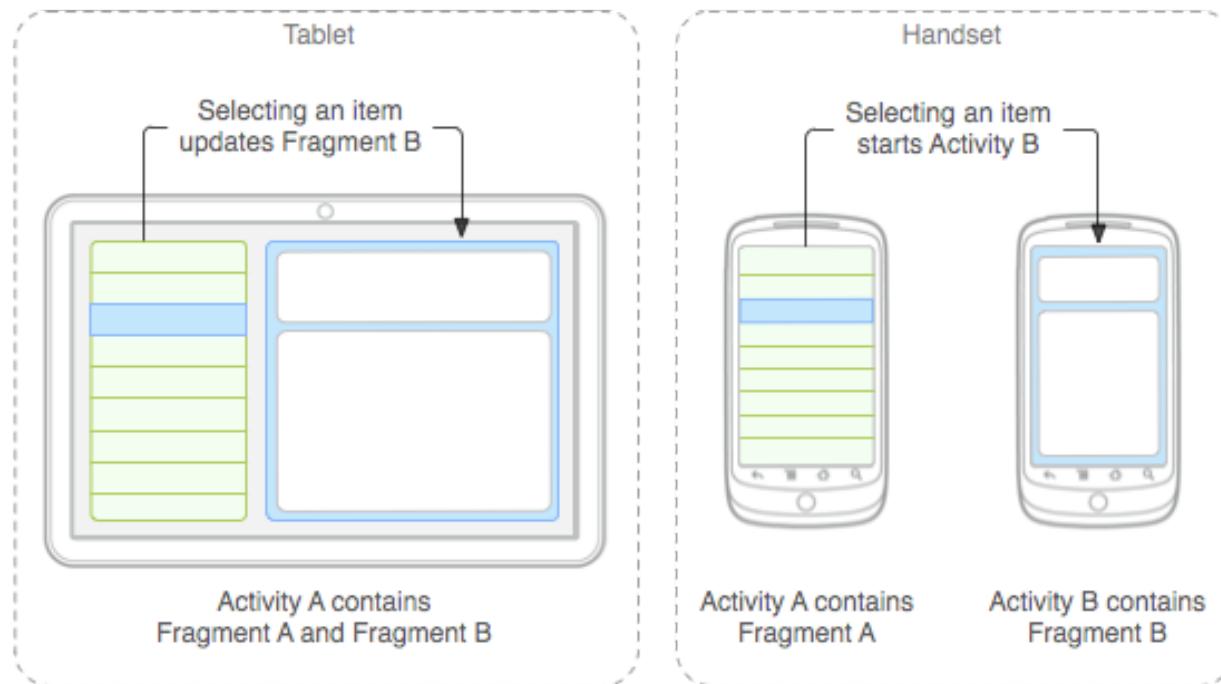
# 9 - Utilizando Fragments e RecyclerView

A lista de produtos já pode ser requisitada através do consumo da operação construída no provedor de serviços de vendas. Agora é hora de exibi-la ao usuário, mas para isso é necessário realizar algumas mudanças na estrutura da camada de visualização, aquela que se encarrega de exibir os dados ao usuário e atender seus eventos.

A ideia, como dito anteriormente, é exibir os produtos em uma lista com algumas informações básicas sobre cada um, como:

- Nome;
- Código;
- Preço.

Para isso será criado uma tela específica na aplicação, com um componente de lista chamado RecyclerView. Essa tela não será uma nova Activity, mas sim um Fragment, que é a forma ideal no Android para construção de aplicativos que possuem mais de uma tela, como será o caso dessa:



Activities e Fragments - Fonte: <https://developer.android.com/guide/components/fragments>

Dessa forma é possível exibir os detalhes de cada produto em uma outra tela, ou Fragment, quando o usuário clicar em um dos produtos, sem a necessidade de instanciar outra Activity. Isso também permite que o usuário navegue da tela de detalhes de volta para a lista, sem muitas implementações no código do projeto.

Para construir a navegação entre as telas será utilizado o componente `NavHostFragment`, que facilita a passagem de parâmetros entre `Fragments` e auxilia na exibição deles.

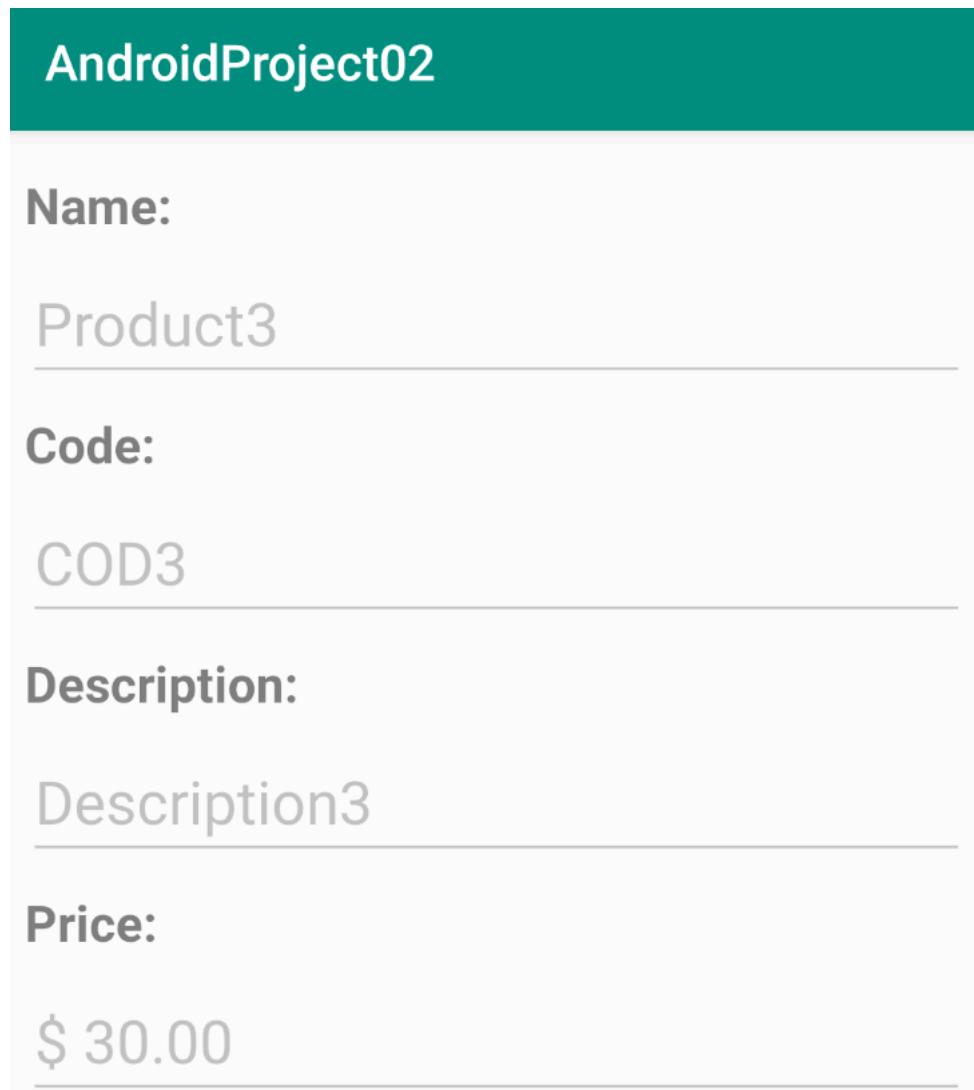
O componente `RecyclerView`, que será utilizado para exibição da lista de produtos, é ideal para conteúdos dinâmicos e com um grande volume de itens. Ele também facilita a alteração de elementos específicos na lista, sem a necessidade de redesenhar todos eles. Além disso, ele torna o uso do data binding fácil de implementar, que ajuda na construção dos layouts, em seu arquivos XML, associando-os diretamente a objetos que representam os conteúdos a serem exibidos.

A seguir, as telas que serão construídas ao final desse capítulo:

AndroidProject02		
Product1	COD1	\$ 10.00
Product2	COD2	\$ 20.00
Product3	COD3	\$ 30.00

Tela com a lista de produtos

Essa é a tela com a lista de produtos. Quando o usuário clicar em um dos produtos exibidos aqui, ele será redirecionado para a tela com os detalhes do produto selecionado:



Tela com os detalhes do produto selecionado

Também serão mostrados mais conceitos sobre `ViewModel`, em conjunto com os `Fragments` e seu ciclo de vida, reforçando ainda mais a necessidade de se adotar uma arquitetura como a iniciada no capítulo anterior.

A seguir, os passos que serão demonstrados ao longo desse capítulo:

- Configurar as dependências do projeto;
- Criar o layout do item da lista de produtos;
- Criar o adaptador da entidade produto para a lista;
- Criar os *binders* extras para auxiliar na construção das telas;
- Incrementar o `ViewModel` da lista de produtos;
- Criar o layout do fragmento da lista de produtos;
- Criar o fragmento para exibir a lista e produtos;

- Adequar o layout da `MainActivity`;
- Criar o mecanismo de navegação com o componente `Navigation`;
- Criar o `ViewModel` dos detalhes do produto;
- Criar o layout do fragmento dos detalhes do produto;
- Criar o fragmento para exibir os detalhes do produto;
- Criar a navegação entre as telas de lista e detalhes dos produtos.

Os passos foram organizados de forma a tornar a explicação mais didática, tentando construir tudo o que é necessário em cada passo.

## 9.1 - Adicionando novas dependências ao projeto

Essa seção dedica-se a preparar o projeto `AndroidProject02` para incluir novas funcionalidades, como navegação entre telas. Para isso, abra o arquivo `build.gradle` do projeto e vá até a seção a seção `dependencies`, onde estão as definições de *plugins* utilizados pelo projeto. Acrescente mais o seguinte:

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.2.1"
```

Esse *plugin* adiciona ao projeto a possibilidade de utilização de passagem de argumentos entre fragmentos de forma segura, o que será visto mais adiante ainda desse capítulo.

Agora, vá no outro arquivo `build.gradle`, que define as dependências do módulo `app`, e acrescente os dois *plugins* a seguir, no topo do arquivo:

```
apply plugin: 'kotlin-kapt'  
apply plugin: "androidx.navigation.safeargs"
```

A primeira biblioteca permite a utilização de anotações no projeto e a segunda se refere aos argumentos a serem passados entre os fragmentos, como citado anteriormente.

Ainda nesse arquivo, na seção `dependencies`, acrescente as seguintes bibliotecas:

```
//Recyclerview  
implementation "androidx.recyclerview:recyclerview:1.1.0"  
  
// Navigation  
implementation "androidx.navigation:navigation-fragment-ktx:2.2.1"  
implementation "androidx.navigation:navigation-ui-ktx:2.2.1"
```

A primeira biblioteca se refere ao componente `RecyclerView`, que será utilizado para exibir a lista de produtos.

O segundo conjunto são extensões do Kotlin para utilização de fragmentos e navegação entre eles.



Lembre-se de sincronizar o projeto, através da opção `Sync Now`, localizado no canto superior direito do Android Studio, quando um dos dois arquivos `build.gradle` está aberto.

Depois de sincronizar o projeto, compile-o para verificar se tudo está correto até agora.

## 9.2 - Criando o fragmento para a lista de produtos

No capítulo anterior a lista de produtos foi requisita ao provedor de serviços de vendas, através da seguinte operação da camada de serviços implementada com o Retrofit:

```
fun getProducts(): Deferred<List<Product>>
```

Agora é a hora de exibir essa lista ao usuário de uma maneira elegante, utilizando o componente `RecyclerView`, com algumas informações sobre cada um deles:

- Nome;
- Código;
- Preço.

Para isso são necessários alguns passos, como já descritos no início desse capítulo e aqui mais detalhados:

- **Criar o layout do item:** para exibir uma lista no Android é necessário criar um layout que represente cada item dentro da lista. Dessa forma, cada item da lista será desenhado segundo esse layout;
- **Criar o adaptador ao item:** o componente `RecyclerView` que será utilizado para construir a lista não conhece a entidade `Product`, por isso é necessário criar um adaptador que saiba converter um produto para um item a ser exibido na lista;
- **Criar o layout da lista:** a lista em si deve estar contida em um layout de um fragmento de tela, posicionada de tal forma a ocupar todo o espaço disponível.

### 9.2.1 - Criando o layout do item da lista de produtos

O layout do item da lista e produtos pode ser entendido como uma linha em uma tabela, ou seja, caixas de textos dispostas horizontalmente, uma para cada informação. Dessa forma ele pode ser criado utilizando o componente `LinearLayout`, que permite a disposição de elementos na horizontal.

Para isso, crie um novo arquivo chamado `item_product.xml` dentro da pasta `res\layout`, que é onde devem ficar os recursos de layout em um projeto Android. Sua estrutura base deve ser como o trecho a seguir:

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

    <data>
        <variable
            name="product"
            type="br.com.siecola.androidproject02.network.Product" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
        </LinearLayout>
    </layout>
```

A tag raiz `layout` é utilizada aqui para fazer com que seja possível utilizar o *data binding* com a entidade `Product`, facilitando o preenchimento de seus detalhes no item da lista.

Perceba que o gerenciador de layout `LinearLayout` é configurado para a orientação `horizontal`. Isso fará com que todos os componentes dentro dele sejam exibidos lado a lado um do outro.

Agora é necessário colocar caixas de texto dentro do `LinearLayout`, para representar os detalhes dos produtos. Por isso, logo antes do fechamento dessa tag, adicione a caixa de texto para representar o nome do produto:

```
<TextView
    android:id="@+id/txtProductListitemName"
    android:layout_width="0dp"
    android:layout_height="46dp"
    android:layout_marginStart="8dp"
    android:layout_marginEnd="8dp"
    android:layout_weight="0.40"
    android:paddingStart="8dip"
    android:paddingTop="12dip"
    android:paddingBottom="12dip"
    android:text="@{product.name}"
    tools:text="Product 1"
    android:textAlignment="viewStart"
    android:textSize="14sp"
    android:textStyle="normal"
    app:layout_constraintEnd_toStartOf="@+id/txtProductListitemCode"
```

```
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Veja que a propriedade `android:text` apronta para a variável `product` definida no início do arquivo. Isso fará com que esse campo seja preenchido pelo nome do produto que for atribuído a cada item da lista.

Logo abaixo, está a propriedade `tools:text`, que é apenas utilizada para exibir uma informação na tela no Android Studio, em tempo de desenvolvimento.

As outras propriedades cuidam da disposição do componente em relação aos demais na tela.

Após esse componente, adicione um outro para representar o código do produto:

```
<TextView
    android:id="@+id/txtProductListItemCode"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_weight="0.30"
    android:minWidth="70dp"
    android:paddingStart="8dip"
    android:paddingTop="12dip"
    android:paddingEnd="8dip"
    android:paddingBottom="12dip"
    android:text="@{product.code}"
    tools:text="COD123"
    android:textAlignment="center"
    android:textSize="14sp"
    android:textStyle="normal"
    app:layout_constraintEnd_toStartOf="@+id/txtProductListItemPrice"
    app:layout_constraintTop_toTopOf="parent" />
```

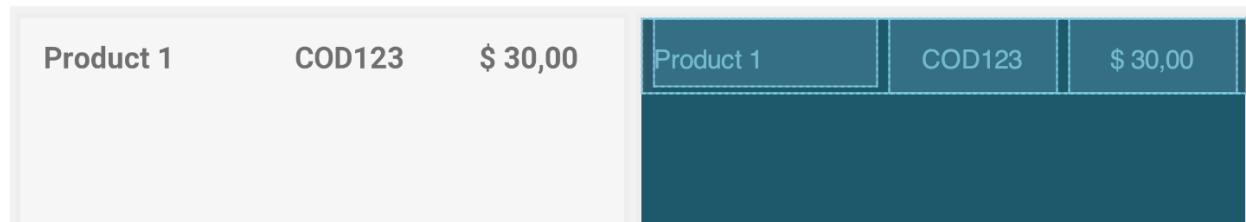
Da mesma forma, o atributo `code` de `product` é utilizado para exibir essa informação na propriedade `android:text` do componente `TextView`.

Para finalizar, logo após esse componente, adicione mais um para representar o preço do produto:

```
<TextView  
    android:id="@+id/txtProductListItemPrice"  
    android:layout_width="0dp"  
    android:layout_height="wrap_content"  
    android:layout_marginEnd="8dp"  
    android:layout_weight="0.30"  
    android:minWidth="40dp"  
    android:paddingStart="8dip"  
    android:paddingTop="12dip"  
    android:paddingEnd="8dip"  
    android:paddingBottom="12dip"  
    android:text="---"  
    tools:text="$ 30,00"  
    android:textAlignment="center"  
        android:textSize="14sp"  
        android:textStyle="normal"  
    app:layout_constraintTop_toTopOf="parent" />
```

O valor a ser exibido na propriedade `android:text` desse último componente do preço do produto deve utilizar um adaptador para exibir seu valor, que ainda será desenvolvido.

A representação gráfica desse layout pode ser vista na aba Design desse arquivo:



Item da lista de produtos

Esse layout agora pode ser utilizado para construção da lista de produtos.

Compile todo o projeto com a opção Rebuild Project no menu Build, para ter certeza de que não há nenhum erro.

## 9.2.2 - Criando o adaptador de produto para a lista

O objetivo dessa seção é criar um adaptador para que o componente RecyclerView saiba como transformar uma lista de entidades do tipo Product em elementos a serem exibidos em uma lista de itens, conforme o layout que foi criado na seção anterior.

Esse adaptador será instanciado no fragmento responsável por exibir a lista de produtos e receberá tal lista durante sua criação. Além disso, ele terá um evento que será invocado quando um item da lista for clicado pelo usuário. Dessa forma será possível direcioná-lo à tela de detalhes do produto.

Como dito anteriormente, o componente RecyclerView é capaz de atualizar cada item da lista individualmente, sem a necessidade de redesenhá-la totalmente. Porém, para isso é necessário criar, dentro do adaptador que será construído nesse capítulo, um mecanismo capaz de fazer com que o componente RecyclerView possa identificar elementos diferentes.

Para começar, crie uma nova classe chamada ProductAdapter no pacote product já existente no projeto, como no trecho a seguir:

```
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter
import androidx.recyclerview.widget.RecyclerView
import br.com.siecola.androidproject02.network.Product
import br.com.siecola.androidproject02.databinding.ItemProductBinding

class ProductAdapter(val onProductClickListener: ProductClickListener) :
    ListAdapter<Product, ProductAdapter.ProductViewHolder>(ProductDiff) {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): ProductAdapter.ProductViewHolder {
    }

    override fun onBindViewHolder(holder: ProductAdapter.ProductViewHolder, position\:
    Int) {
    }
}
```

As duas funções criadas dentro dessa classe serão implementadas a seguir.

Na declaração da classe ProductAdapter , perceba que ela recebe um parâmetro do tipo ProductClickListener , que ainda será construído nesse arquivo. Ele é o *listener* que será invocado quando um produto da lista for clicado pelo usuário, que permitirá a navegação para a tela de detalhes.

Ainda em sua declaração, há a definição de três outras classes, detalhadas a seguir:

- **ProductViewHolder**: essa classe receberá a instância de cada item do layout da lista de produtos e poderá atribuir o produto à variável product declarada dentro da tag data dela;
- **ProductDiff**: ela será responsável por fazer a comparação entre dois elementos da lista, possibilitando o RecyclerView reconhecer quando dois itens são iguais ou não;
- **ProductClickListener**: ela será responsável por tratar o evento de clique do usuário em um item da lista.

Elas ainda não existem e devem ser criadas como classes internas de ProductAdapter . Por isso comece com a ProductViewHolder , colocando sua declaração logo abaixo da função onBindViewHolder :

```
class ProductViewHolder(private var binding: ItemProductBinding):
    RecyclerView.ViewHolder(binding.root) {
    fun bind(product: Product) {
        binding.product = product
        binding.executePendingBindings()
    }
}
```

Veja que a função `bind` recebe uma instância de `Product`, que é atribuída à variável `product`. Essa variável foi declarada dentro do layout de `item_product.xml`, como pode ser visto no trecho a seguir:

```
<data>
    <variable
        name="product"
        type="br.com.siecola.androidproject02.network.Product" />
</data>
```

Dessa forma, os valores da variável `product` podem ser utilizados dentro dos componentes desse layout, como é o caso do nome do produto em sua primeira caixa de texto:

```
android:text="@{product.name}"
```

Voltando à classe `ProductAdapter`, logo abaixo de `ProductViewHolder` e ainda dentro de `ProductAdapter`, crie a classe `ProductDiff`:

```
companion object ProductDiff : DiffUtil.ItemCallback<Product>() {
    override fun areItemsTheSame(oldItem: Product, newItem: Product): Boolean {
        return oldItem === newItem
    }

    override fun areContentsTheSame(oldItem: Product, newItem: Product): Boolean {
        return ((oldItem.id == newItem.id)
            && (oldItem.name.equals(newItem.name))
            && (oldItem.code.equals(newItem.code)))
            && (oldItem.price == newItem.price))
    }
}
```

Perceba que existem duas funções herdadas da classe abstrata `ItemCallback`:

- **`areItemsTheSame`**: essa função será chamada pelo `RecyclerView` para verificar se duas instâncias de `Product` representam o mesmo produto. Graças ao operador `==` do Kotlin isso é possível de ser feito em apenas uma comparação simples;

- **areContentsTheSame**: essa função é chamada para verificar se o conteúdo a ser exibido de duas instâncias são os mesmos.

Com essas duas funções o RecyclerView pode ser mais eficiente e redesenhar um item da lista somente quando ele realmente mudar.

A terceira classe interna de ProductAdapter a ser criada é a responsável por tratar o evento de clique do usuário em um item da lista de produtos:

```
class ProductClickListener(val clickListener: (product: Product) -> Unit) {  
    fun onClick(product: Product) = clickListener(product)  
}
```

Ela deve receber como parâmetro uma função que será invocada quando tal evento acontecer. Essa função será criada dentro do fragmento responsável por exibir a lista de produtos, que ainda será construído.

O que falta para concluir a implementação da classe ProductAdapter é finalizar as funções onCreateViewHolder e onBindViewHolder:

```
override fun onCreateViewHolder(  
    parent: ViewGroup,  
    viewType: Int  
): ProductAdapter.ProductViewHolder {  
    return ProductViewHolder(ItemProductBinding.inflate(LayoutInflater.from(parent.c\\  
ontext)))  
}
```

Perceba que o único trabalho dessa função é criar uma instância de ProductViewHolder inflando um layout de um item da lista de produtos.

Essa instância é então utilizada na chamada à função onBindViewHolder para que o produto de uma posição específica da lista possa ser atribuído à variável do layout do item da lista, através da função bind:

```
override fun onBindViewHolder(holder: ProductAdapter.ProductViewHolder, position: Int) {
    val product = getItem(position)
    holder.bind(product)
    holder.itemView.setOnClickListener {
        onProductClickListener.onClick(product)
    }
}
```

Também dentro dessa função o evento de clique no item da lista é atribuído a esse produto.

E isso finaliza a implementação do adaptador da entidade `Product` para que ela possa ser exibida em uma lista no `RecyclerView`.

Uma instância desse adaptador será criada e utilizada na construção do objeto de lista durante a inicialização do fragmento responsável por exibir a lista de produtos.

### 9.2.3 - Criando os *binders extras* para as telas

No arquivo `item_product.xml`, onde o layout do item da lista foi criado, é necessário fazer a associação do preço do produto à caixa de texto, como foi feito nas demais:

```
android:text="@{product.code}"
```

Porém, o atributo `text` só pode receber um conteúdo que seja `String`, por isso é necessário criar um adaptador a essa associação, ou um *binding adapter*. Dessa forma é possível fazer a associação direta do preço do produto à caixa de texto. Além disso, também é possível formatar como esse valor aparecerá ao usuário.

Para isso, crie um novo arquivo chamado `ProductBindingAdapters` dentro do pacote `product`, com os seguintes *imports*:

```
import android.widget.TextView
import androidx.databinding.BindingAdapter
import androidx.recyclerview.widget.RecyclerView
import br.com.siecola.androidproject02.network.Product
```

O objetivo desse arquivo é utilizar a anotação `@BindingAdapter` para criar novas funções que possam ser utilizadas como novos atributos na associação de valores aos componentes gráficos criados nos layout dos arquivos XML.

Para começar, crie uma nova função para fazer com que a lista de produtos possa ser associada a um componente do tipo `RecyclerView`:

```
@BindingAdapter("productsList")
fun bindProductsList(recyclerView: RecyclerView, products: List<Product>?) {
    val adapter = recyclerView.adapter as ProductAdapter
    adapter.submitList(products)
}
```

Todo o segredo por traz dessas funções está na anotação `@BindingAdapter` e na definição da assinatura da função. Veja que a anotação recebe um nome como parâmetro. Esse nome é o que será utilizado no componente `RecyclerView` da lista de produtos.

A função então recebe o componente em si em que a lista deve ser associada e a lista e produtos. Dessa forma, o adaptador de produtos, criado na seção anterior, é resgatado e tem sua lista de produtos submetida para ser exibida. E isso tudo é feito de forma automática durante a exibição do fragmento da lista e produtos.

A segunda função a ser criada nesse arquivo será utilizada para a exibição do preço do produto na caixa de texto que foi criada para o layout do item de cada produto:

```
@BindingAdapter("productPrice")
fun bindProductPrice(txtProductPrice: TextView, productPrice: Double?) {
    productPrice?.let {
        val price = "$ " + "%.2f".format(productPrice)
        txtProductPrice.text = price
    }
}
```

O nome do atributo a ser utilizado na caixa de texto é o definido no parâmetro da anotação, ou seja, `productPrice`. A função em si recebe o componente da caixa de texto e o preço, que pode ser nulo, logo há um teste para saber se ele existe ou não. Caso exista, o preço é convertido para `String` e atribuído à caixa de texto numa formatação adequada.

Com esse novo atributo, volte ao arquivo `item_product.xml` e acrescente a seguinte instrução na última caixa de texto definida nesse arquivo, a que foi criada para exibir o preço do produto. Lembre-se de remover o atributo `android:text` que estava com valor fixo:

```
app:productPrice="@{product.price}"
```

Veja que a nova propriedade `productPrice` aparece disponível para ser utilizada, pois ela foi definida através da anotação `@BindingAdapter("productPrice")`.

## 9.2.4 - Incrementando o ViewModel da lista de produtos

A classe `ProductListViewModel` foi criada para cuidar da exibição da lista de produtos, fazendo a ponte entre o fragmento que a exibe e a camada de serviços que utiliza o Retrofit para fazer a consulta ao provedor de serviços de vendas.

Até o momento ela está apenas responsável por fazer a consulta da lista de produtos propriamente dita. Agora já é possível fazer com que essa classe ganhe mais responsabilidade e expor essa lista de produtos para o fragmento que for utilizá-la, através de um `LiveData` para que ele possa observar atualizações nessa lista. Afinal, o adaptador para o componente `RecyclerView` foi preparado para isso.

Para começar, vá até a classe `ProductListViewModel` e crie os seguintes atributos privados:

```
private val _products = MutableLiveData<List<Product>>()
val products: LiveData<List<Product>>
    get() = _products
```

O primeiro atributo é privado e é o que será efetivamente atualizado quando a operação de consulta dos produtos for concluída. Já o segundo é o que será exposto para o fragmento, que poderá observar as alterações nesse atributo.

Perceba que o atributo `products` possui apenas a operação `get` definida, que expõe a lista de produtos através do primeiro atributo `_products`, o que faz com que ela apenas possa ser observada e não ter nenhum valor alterado.

Agora, para fazer com que a lista de produtos seja alterada, faça com que o atributo `_products` receba esse valor dentro da função `getProducts`, logo após a instrução de log que exibe a quantidade de produtos que foi recebida:

```
Log.i(TAG, "Number of products ${productsList.size}")
```

```
_products.value = productsList
```

Isso fará com que quem estiver observando o atributo `products` seja notificado da alteração. Isso será feito no fragmento que será criado em algumas seções adiante nesse capítulo. Isso é possível graças a utilização de `LiveData`, como pode ser observado em sua declaração.

## 9.2.5 - Criando o layout da lista de produtos

Agora que o `ViewModel` da lista de produtos está apto a expor seus dados, é possível criar o seu layout. Para isso, crie um novo arquivo de recurso na pasta `res\layout`, chamado `fragment_products_list.xml`:

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

    <data>
        <variable
            name="productListViewModel"
            type="br.com.siecola.androidproject02.product.ProductListViewModel" />
    </data>
</layout>
```

Atente-se para substituir todo seu conteúdo pelo trecho anterior, que contém como componente base o layout, para ser possível adicionar a tag data e criar a variável productListViewModel, que vai representar todo do ViewModel da lista de produtos.

Agora dentro da tag layout, após a tag data, crie o componente RecyclerView dentro de um ConstraintLayout, como no trecho a seguir:

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/rcvProducts"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="8dp"
        app:layoutManager="androidx.recyclerview.widget.GridLayoutManager"
        app:spanCount="1"
        android:orientation="vertical"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:productsList="@{productListViewModel.products}"
        tools:listitem="@layout/item_product" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Aqui o componente RecyclerView está posicionado para ocupar todo o espaço disponível na tela, com algumas margens.

Repare que o último atributo é o `app:productsList`, que foi criado no arquivo `ProductBindingAdapters.kt` para receber a lista de produtos, que nesse caso está sendo passada pela variável `productListViewModel` criada nesse arquivo de layout.

### 9.2.6 - Criando o fragmento da lista de produtos

Agora que o layout da tela com a lista de produtos foi construída, é possível criar o fragmento responsável por exibi-la e tratar os eventos do usuário, como o clique do usuário em um elemento da lista.

Um fragmento sempre está atrelado a uma `Activity` e da mesma forma ao seu ciclo de vida, como será mostrado na figura a seguir.

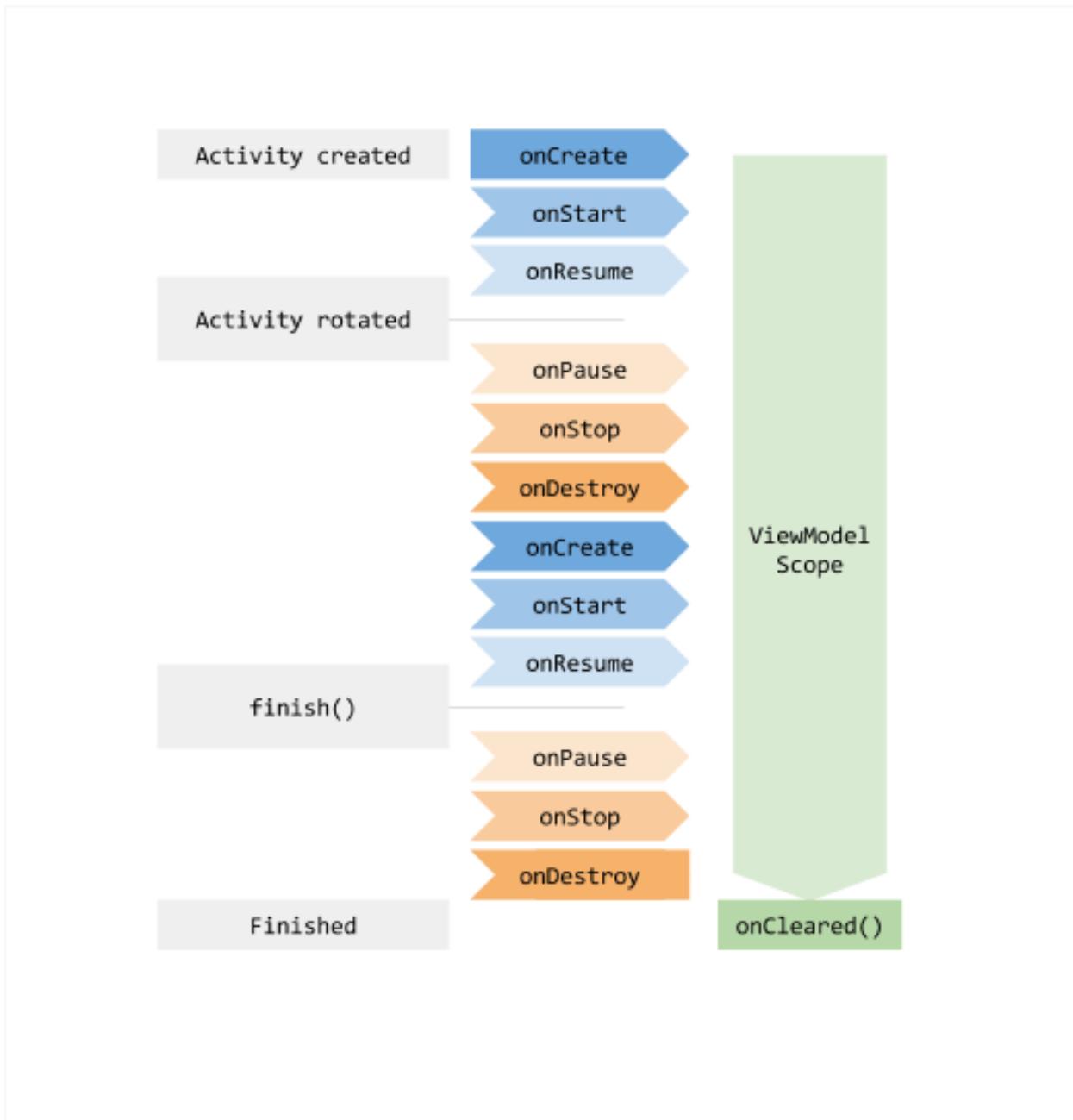
A arquitetura atual para desenvolvimento para Android recomenda que aplicativos com múltiplas telas sejam desenvolvidos utilizando `Fragments`, como será feito a partir desse capítulo. Isso significa que, na maioria das vezes, a aplicação possui apenas uma `Activity` e todos os `Fragments` são exibidos através dela e atrelados a ela.

Da mesma forma, em uma arquitetura onde `ViewModels` são escolhidos para fazer a ponte entre a camada de UI e as demais da aplicação, significa que cada fragmento deve ter o seu próprio `ViewModel`.

Esse fragmento que será construído utilizará o `ViewModel` criado no capítulo anterior, e incrementado nesse, para observar as mudanças de dados (lista de produtos) e estados (ações do usuário). Tudo isso atrelado ao seu ciclo de vida, muito semelhante ao que foi demonstrado com relação a uma `Activity`. Isso significa que o ciclo de vida de `ProductListViewModel` estará de certa forma ligado ao fragmento que aqui será criado.

Com isso, a instância de `ProductListViewModel` será criada quando esse novo fragmento for criado e também será destruído quando ele não for mais necessário de ser exibido, por exemplo quando outro fragmento o substituir na tela.

Observe a figura a seguir, que associa o escopo de vida de um `ViewModel` a uma `Activity`. A mesma ideia pode ser aplicada se comparada com um ciclo de vida de um `Fragment`:



Escopo do ViewModel - Fonte: <https://developer.android.com/topic/libraries/architecture/viewmodel>

Perceba que o ViewModel “resiste” à mudanças de configuração, como a rotação da tela do dispositivo, que faz com que a Activity ou até mesmo um Fragment, seja destruído. Essa é uma das grandes razões da utilização dessa técnica com o ViewModel, pois sua instância se mantém ativa até que a Activity ou o Fragment que são “donos” dele seja substituído por outro dentro do escopo da aplicação.

Repare também na figura anterior, ao final do ciclo de vida do ViewModel, que a função `onCleared()` é chamada antes dele ser completamente destruído. Em `ProductListViewModel`, essa função está sendo

utilizada para cancelar alguma tarefa pendente e suas coroutines que talvez possam estar ativas, consultando o provedor de serviços de vendas. Isso é uma forma segura de cancelar trabalhos, fechar conexões e liberar recursos de forma limpa e eficiente.

Como dito anteriormente, uma Activity ou um Fragment são donos do ciclo de vida de um ViewModel, ou como se pode encontrar em documentações: *Lifecycle owners*.

Para começar a criação do novo fragmento para a lista de produtos, crie no pacote product, uma nova classe chamada ProductsListFragment, como no trecho a seguir:

```
import android.os.Bundle
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.lifecycle.ViewModelProviders
import androidx.navigation.fragment.findNavController
import br.com.siecola.androidproject02.databinding.FragmentProductsListBinding
import androidx.recyclerview.widget.RecyclerView.VERTICAL

private const val TAG = "ProductsListFragment"

class ProductsListFragment : Fragment() {

}
```

Perceba que ela é uma classe especial, pois herda de Fragment.

Dentro da classe, crie o atributo productListViewModel para representar o ViewModel da lista de produtos criado no capítulo anterior:

```
private val productListViewModel: ProductListViewModel by lazy {
    ViewModelProviders.of(this).get(ProductListViewModel::class.java)
}
```

Agora crie a função onCreateView, que equivale à função onCreate em uma Activity, ou seja, é chamada quando o fragmento deve ser criado, seja em sua inicialização ou em sua recriação após uma mudança de configuração, como a rotação da tela do dispositivo:

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?): View? {
}
```

O layout a ser exibido deve ser “inflado” dentro dessa função. Nesse momento é possível pegar a referência ao objeto que representa o *binding* da tela e atribuir as demais informações a ele, como o *ViewModel*:

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?): View? {
    val binding = FragmentProductsListBinding.inflate(inflater)

    binding.setLifecycleOwner(this)

    binding.productListViewModel= productListViewModel

    val itemDecor = DividerItemDecoration(getContext(), VERTICAL);
    binding.rcvProducts.addItemDecoration(itemDecor);

    binding.rcvProducts.adapter = ProductAdapter(ProductAdapter.ProductClickListener\
    {
        Log.i(TAG, "Product selected: ${it.name}")
    })
}

return binding.root
}
```

Veja que com a referência do layout a ser exibido nesse fragmento é possível ter acesso à referência do *RecyclerView* que foi criado no arquivo `fragment_products_list.xml` para exibir a lista de produtos:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/rcvProducts"
```

Com essa referência o adaptador *ProductAdapter* pode ser criado e atribuído a ele, ao mesmo tempo em que o *listener* para escutar o evento de clique em um produto é criado e atribuído, como pode ser visto na instrução criada no trecho anterior:

```
binding.rcvProducts.adapter = ProductAdapter(ProductAdapter.ProductClickListener {
    Log.i(TAG, "Product selected: ${it.name}")
})
```

Por enquanto esse *listerner* vai apenas imprimir uma mensagem de log dizendo qual produto foi criado, mas em algumas seções à frente nesse capítulo ele irá implementar a navegação para a tela de detalhes do produto.

A criação do objeto do tipo `DividerItemDecoration`, e depois sua atribuição na instância `rcvProducts`, serve para adicionar uma linha entre os elementos da lista de produtos, facilitando sua visualização.

Finalmente, na última instrução da função `onCreateView`, a `View` que representa a tela com a lista de produtos é devolvida para ser exibida.

As novas implementações feitas até agora nesse capítulo poderiam ser testadas, para a visualização da lista de produtos, porém ainda faltam alguns passos por conta da adaptação da arquitetura do projeto para utilizar fragmentos e a navegação entre eles. Tais passos serão feitos na próxima seção.

## 9.3 - Adicionando o mecanismo de navegação entre telas

Como dito anteriormente, a ideia é fazer com que o aplicativo tenha duas telas: uma para exibir a lista de produtos e outra para seus detalhes quando o usuário clicar em um item dessa lista.

Na seção anterior a tela com a lista de produtos foi criada, mas ainda não pôde ser exibida, pois seu fragmento não fazia parte da estrutura base do projeto.

Essa seção irá começar a criação do mecanismo de navegação entre essas duas telas utilizando o `NavHostFragment`, ideal quando um aplicativo possui uma `Activity` e vários `Fragments`, como é o caso do que está sendo construído nesse capítulo.

Com esse mecanismo também é possível realizar a navegação entre fragmentos passando argumentos entre eles, de uma forma segura através do conceito de `SafeArgs`, como será visto na próxima seção.

Ao final dessa preparação, o fragmento com a lista de produtos será visível ao usuário, quando o aplicativo abrir.

### 9.3.1 - Adequando o layout da MainActivity

O layout do arquivo `activity_main.xml` foi criado pelo *template* do Android Studio no momento da criação do projeto. Ele apenas exibia um `TextView`, mas agora ele deve ser alterado para propiciar o mecanismo de navegação entre `Fragments`. Para isso, abra-o e altere seu conteúdo de acordo com o seguinte trecho:

```
<?xml version="1.0" encoding="utf-8"?>

<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/navigation_graph" />
```

Esse simples trecho cria a base para o mecanismo de navegação entre os fragmentos de um aplicativo, mantendo-se apenas uma Activity. Ele também adiciona um novo componente chamado NavGraph, que será criado e detalhado na seção seguinte.

Com esse mecanismo será possível exibir o fragmento da lista de produtos e substituí-lo por outro fragmento, como o de detalhes do produto, quando um usuário clicar em um item da lista.

Outra adaptação que deve ser feita é na classe `MainActivity`. Na verdade ela deve voltar a ser o que era quando foi criada pelo *template* do Android Studio, por isso abra-a e remova a seguinte linha de código:

```
val viewModel = ViewModelProviders.of(this).get(ProductListViewModel::class.java)
```

Essa linha foi adicionada para criar a instância de `ProductListViewModel` para que o cliente do provedor de serviços de vendas buscassem todos os produtos nele. Porém isso agora faz parte do ciclo de vida do fragmento `ProductsListFragment`.

### 9.3.2 - Criando o mecanismo de navegação entre as telas

Para finalizar a base do mecanismo de navegação entre as telas, é necessário criar o arquivo `navigation/navigation_graph.xml` mencionado na seção anterior. Ele será o responsável por criar um grafo de todos os fragmentos da aplicação e seus relacionamentos, que são as navegações possíveis de serem feitas entre eles.

Como o projeto ainda possui apenas um fragmento, o de lista de produtos, ainda não será possível construir um verdadeiro grafo, mas na seção seguinte isso será resolvido.

Para começar, crie um diretório novo na pasta `res` chamado `navigation`. Dentro dele crie um novo arquivo chamado `navigation_graph.xml`, como no trecho a seguir:

```
<?xml version="1.0" encoding="utf-8"?>

<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/navigation_graph"
    app:startDestination="@+id/fragmentProductsList">

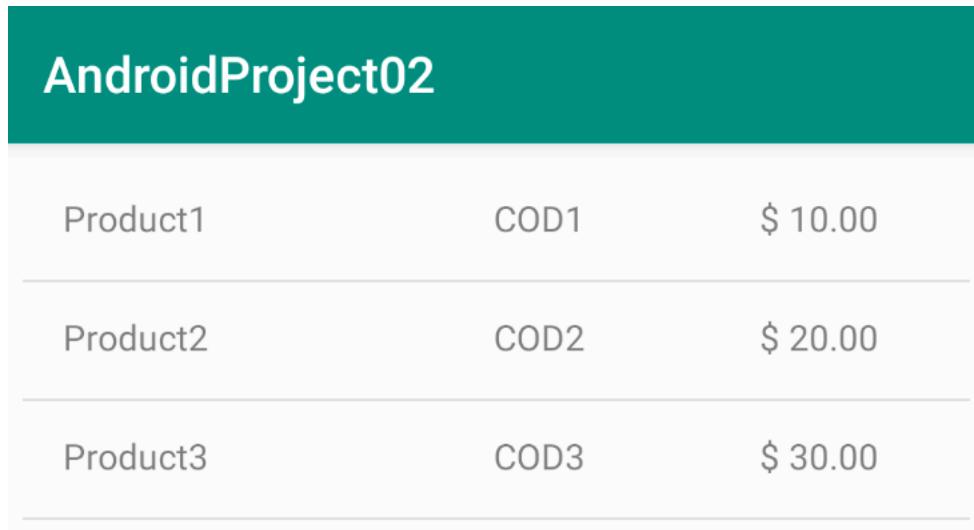
    <fragment
        android:id="@+id/fragmentProductsList"
        android:name="br.com.siecola.androidproject02.product.ProductsListFragment"
        android:label="fragment_products_list"
        tools:layout="@layout/fragment_products_list" >
    </fragment>
</navigation>
```

O componente raiz desse arquivo é o `navigation`, que possui o id `navigation_graph`, mencionado na seção anterior, dentro do arquivo `activity_main.xml`. Esse componente é o responsável por listar todos os fragmentos que serão exibidos.

Perceba que ele possui o atributo `app:startDestination`, que é responsável por definir qual é o fragmento que será exibido quando a aplicação abrir pela primeira vez. Ele está configurado com o valor `@+id/fragmentProductsList`, que é a identificação do fragmento responsável por exibir a lista de produtos, criado nesse capítulo. Isso significa que ele será a primeira tela que o usuário verá.

A identificação desse fragmento está definida dentro da tag `fragment`, dentro do componente `navigation`. Ele por sua vez aponta para a classe `br.com.siecola.androidproject02.product.ProductsListFragment` e o layout `fragment_products_list`, criados nesse capítulo para exibir a lista de produtos.

Com apenas essa configuração já é possível executar a aplicação e verificar que a tela com a lista de produtos é exibida ao usuário:



Tela inicial com a lista de produtos

Dessa forma, tudo o que foi feito até então nesse capítulo pôde ser testado:

- O layout do item da lista de produtos;
- O layout da lista de produtos;
- O fragmento que exibe o layout da lista de produtos;
- O adaptador que transforma a entidade Product em um item a ser exibido pelo RecyclerView;
- A integração entre o fragmento ProductsListFragment e o ViewModel ProductListViewModel.

As próximas seções se encarregarão de analisar o comportamento do aplicativo, até o momento, e incluir a tela que exibirá os detalhes do produto quando ele foi clicado pelo usuário.

## 9.4 - Analisando a integração entre o Fragment e o ViewModel

Agora que a lista de produtos já está sendo exibida na tela inicial da aplicação é possível entender um pouco mais sobre a integração entre o fragmento que a exibe e o ViewModel que a requisita e a detém.

Para isso, acrescente mais uma mensagem de log no início da função onCreateView da classe ProductsListFragment:

```
Log.i(TAG, "Creating ProductsListFragment")
```

Essa mensagem irá aparecer sempre que essa função for chamada pelo Android, ou seja, quando a tela com a lista de produtos precisar ser criada pela primeira vez ou quando houver uma mudança de configuração e ela precisar ser recriada, como em uma rotação do dispositivo.

O interessante de se analisar aqui é o comportamento de `ProductListViewModel` e a requisição dos produtos ao provedor de serviços de vendas. O ideal seria que ele tivesse as seguinte ações em relação ao fragmento que detém seu ciclo de vida, ou seja, `ProductsListFragment`:

- Quando o fragmento for criado, uma instância de `ProductListViewModel` deve ser criada também;
- No momento em que a instância de `ProductListViewModel` é criada, a lista e produtos deve ser requisitada e armazenada em seu atributo privado;
- A lista então deve ser exibida pelo fragmento `ProductsListFragment`;
- Caso aconteça alguma mudança de configuração, como a rotação do dispositivo, uma nova instância de `ProductsListFragment` deve ser criada;
- Nesse caso da recriação de `ProductsListFragment` por uma mudança de configuração, a mesma instância de `ProductListViewModel` deve ser utilizada.

Como a mesma instância `ProductListViewModel` é utilizada quando uma mudança de configuração acontece, nenhuma requisição deve ser feita ao provedor de serviços de vendas, pois isso só é feito quando sua instância é criada, economizando o consumo de dados do aparelho.

Esse comportamento pode ser analisado através das mensagens de logs da aplicação, ajustando o filtro do Logcat para INFO e I/Product:



The screenshot shows the Android Logcat interface. At the top, there are three dropdown menus: the first is set to 'Info', the second to 'Logcat', and the third has a magnifying glass icon and the text 'I/Product'. Below these, a list of log messages is displayed in a monospaced font. The messages are as follows:

```
I/ProductsListFragment: Creating ProductsListFragment
I/ProductListViewModel: Preparing to request products list
I/ProductListViewModel: Products list requested
I/ProductListViewModel: Loading products
I/ProductListViewModel: Number of products 3
I/ProductsListFragment: Creating ProductsListFragment
I/ProductsListFragment: Creating ProductsListFragment
```

Logs com mudanças de configuração

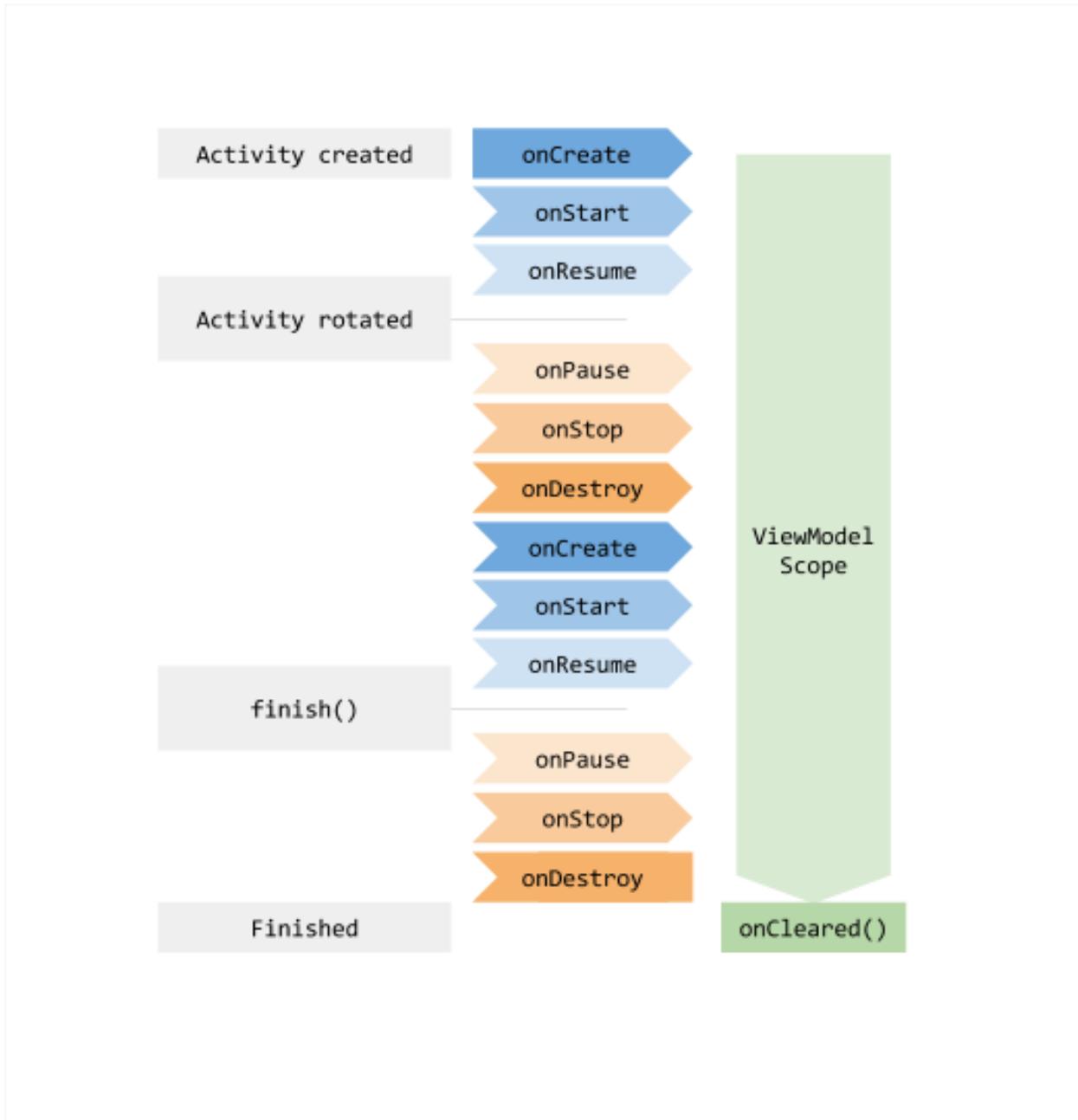
Veja que o primeiro log com a mensagem `Creating ProductsListFragment` acontece quando a aplicação é iniciada. Logo em seguida aparecem as mensagens de `ProductListViewModel`, realizando a requisição ao provedor de vendas para buscar a lista de produtos.

Depois que os produtos são trazidos eles são exibidos ao usuário.

Os dois últimos logs se referem a duas rotações do dispositivo, fazendo com que o fragmento `ProductsListFragment` seja recriado e tendo sua função `onCreateView` invocada pelo sistema Android. Porém, perceba que nenhuma mensagem de log foi gerada por `ProductListViewModel`, pois a mesma instância criada na inicialização da aplicação foi utilizada, logo nenhuma requisição ao provedor de vendas foi feita.

Esse é o comportamento desejado, pois evita que várias requisições sejam feitas desnecessariamente, evitando o consumo de dados do dispositivo.

Esse experimento também reforça o que foi detalhado anteriormente nesse capítulo sobre o ciclo de vida de um Fragment e seu ViewModel:



Escopo do ViewModel - Fonte: <https://developer.android.com/topic/libraries/architecture/viewmodel>

Veja que a figura anterior ilustra exatamente o que aconteceu: a instância do ViewModel manteve-se ativa mesmo com uma mudança de configuração e recriação do Fragment que é dono de seu ciclo

de vida.

## 9.5 - Criando o fragmento para os detalhes do produto

Agora que a lista de produtos já está sendo exibida para o usuário e a base do mecanismo de navegação entre fragmentos já foi criada, é hora de criar a tela que exibirá os detalhes do produto, quando ele for clicado pelo usuário na lista.

Algumas das atividades necessárias para isso já começaram a ser preparadas no código, embora ainda não tenham recebido a devida explicação por não ter sido o momento apropriado. Para tanto, essa seção cuidará de juntar as peças restantes.

A ideia, como dito anteriormente, é exibir os detalhes do produto mediante ao clique do usuário em um item da lista. Para isso será necessário:

- Criar uma nova operação para buscar o produto pelo seu código no provedor de serviços de vendas;
- Criar um novo ViewModel para a tela de detalhes do produto, que irá consultar o produto pelo seu código;
- Criar o layout da tela de detalhes do produto, atrelada ao ViewModel citado anteriormente;
- Criar o fragmento para exibir a tela de detalhes do produto;
- Criar a navegação entre as telas de listagem de produtos e seus detalhes, passando o código do produto como parâmetro.

O passo para buscar um produto pelo seu código não precisaria ser feito, a rigor, pois a entidade Product contida na lista já possui todos os detalhes a serem exibidos. A ideia aqui é apenas trazer desafios didáticos, incrementando a camada de serviço da aplicação e criando um outro ViewModel que também a utiliza.

### 9.5.1 - Buscando o produto pelo seu código

O provedor de serviços de vendas possui uma operação para buscar um produto pelo seu código, como pode ser visto na seção 4.b de sua [documentação<sup>27</sup>](#).

Método: GET

URL: <https://sales-provider.appspot.com/api/products/{code}>

Para buscar um produto pelo seu código, basta passá-lo como argumento na URL e utilizar o método GET , como descrito no trecho anterior.

Como dito na seção anterior, essa operação será utilizada pela tela de detalhes do produto, como forma de incrementar sua complexidade técnica, para fins didáticos.

<sup>27</sup><https://github.com/siecola/GAESalesProvider>

Dessa forma, é necessário criar uma nova função na camada de serviços da aplicação para que essa operação possa ser realizada. Para isso, abra o arquivo `SalesApi.kt`, dentro do pacote `network`. Dentro da interface `Sales ApiService` já existem as seguinte operações definidas:

```
interface Sales ApiService {  
    @GET("api/products")  
    fun getProducts(): Deferred<List<Product>>  
  
    @POST("oauth/token")  
    @FormUrlEncoded  
    fun getToken(  
        @Header("Authorization") basicAuthentication: String,  
        @Field("grant_type") grantType: String,  
        @Field("username") username: String,  
        @Field("password") password: String  
    ): Call<OauthTokenResponse>  
}
```

A operação `getProducts()` já está sendo utilizada pela tela que lista todos os produtos e agora é necessário criar uma outra operação para buscar o produto pelo seu código, dentro dessa interface, como no trecho a seguir:

```
@GET("api/products/{code}")  
fun getProductByCode(@Path("code") code: String): Deferred<Product>
```

De acordo com a documentação do provedor de vendas, o código do produto deve ser passado como parâmetro na URL, por isso a anotação `@GET` recebe o parâmetro `"api/products/{code}"`, com o código na URL.

Da mesma forma, a assinatura da função recebe o código do produto como parâmetro, anotado com `@Path("code")`.

Perceba também que o retorno da função é do tipo `Deferred<Product>`, que primeiramente define que apenas um produto será devolvido e também que o retorno poderá ser utilizado como função suspensa dentro de uma `Coroutine`.

Agora essa função poderá ser utilizada pelo `ViewModel` do fragmento de detalhes do produto, a ser criado na próxima seção.

## 9.5.2 - Criando o ViewModel dos detalhes do produto

O `ViewModel` a ser criado para a tela de detalhes do produto deve possuir as seguintes características:

- Receber o código do produto como parâmetro de sua criação;

- Guardar o código do produto em um atributo privado;
- Buscar o produto pelo seu código, no provedor de serviços de vendas, durante sua inicialização;
- Guardar o produto buscado em um atributo do tipo LiveData para que possa ser observado pelo fragmento da tela de detalhes.



Novamente, fazer com que esse novo ViewModel receba o código ao invés do produto todo é apenas um recurso didático para explorar mais afundo essa técnica.

O primeiro requisito para esse novo ViewModel traz um desafio adicional, pois é necessário utilizar-se de um padrão conhecido como factory para poder criar um ViewModel que receba um parâmetro no momento de sua criação.

Para começar, crie um novo pacote no projeto chamado productdetail. Nele, crie uma nova classe chamada ProductDetailViewModelFactory, como no trecho a seguir:

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider

class ProductDetailViewModelFactory(private val code: String)
    : ViewModelProvider.Factory {

}
```

Veja que essa classe recebe o código do produto como parâmetro, bem como implementa a interface ViewModelProvider.Factory, que a obriga a ter a função create, que será chamada no momento de sua criação, como no trecho a seguir, que deve ser inserido na classe:

```
override fun <T : ViewModel?> create(modelClass: Class<T>): T {
    if (modelClass.isAssignableFrom(ProductDetailViewModel::class.java)) {
        return ProductDetailViewModel(code) as T
    }
    throw IllegalArgumentException("The ProductDetailViewModel class is required")
}
```

A responsabilidade dessa função é apenas criar uma instância de ProductDetailViewModel passando o código do produto como parâmetro. Por enquanto, o Android Studio irá acusar alguns erros nessa classe, pois ProductDetailViewModel ainda não existe, o que será resolvido algumas seções à diante.

Quando uma instância de ProductDetailViewModel tiver que ser criada, o factory ProductDetailViewModelFactory deverá ser utilizado.

Agora o ViewModel dos detalhes do produto já pode ser criado. Para isso, dentro do pacote productdetail, crie uma nova classe chamada ProductDetailViewModel, como no trecho a seguir:

```
import android.util.Log
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import br.com.siecola.androidproject02.network.Product
import br.com.siecola.androidproject02.network.SalesApi
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.Job
import kotlinx.coroutines.launch

private const val TAG = "ProductDetailViewModel"

class ProductDetailViewModel(private val code: String): ViewModel() {
    private var viewModelJob = Job()
    private val coroutineScope = CoroutineScope(viewModelJob + Dispatchers.Main)

    private val _product = MutableLiveData<Product>()

    val product: LiveData<Product>
        get() = _product

    init {
        getProduct()
    }
}
```

Veja que essa nova classe recebe o código do produto como parâmetro em seu construtor, como já foi dito durante a criação de seu *factory*.

Os dois primeiros atributos privados foram criados com o propósito de controlar a execução da coroutine que será criada para acessar o provedor de vendas, assim como foi feito no ViewModel da lista de produtos.

Da mesma forma, os atributos `_product` e `product` foram criados para guardarem o produto buscado no provedor de vendas, lembrando que somente o segundo pode ser observado de fora do `ViewModel`.

A função `init`, que será chamada no momento da criação desse `ViewModel`, irá invocar a função `getProduct` para buscar o produto pelo seu código, como no trecho a seguir, que deve ser colocado na classe:

```

private fun getProduct() {
    Log.i(TAG, "Preparing to request a product by its code")
    coroutineScope.launch {
        var getProductDeferred = SalesApi.retrofitService.getProductByCode(code)
        try {
            Log.i(TAG, "Loading product by its code")

            var productByCode = getProductDeferred.await()

            Log.i(TAG, "Name of the product ${productByCode.name}")

            _product.value = productByCode
        } catch (e: Exception) {
            Log.i(TAG, "Error: ${e.message}")
        }
    }
    Log.i(TAG, "Product requested by code")
}

```

Veja que essa função tem o mesmo princípio de funcionamento de `getProducts` de `ProductListViewModel`. Basicamente o que muda é o fato dela requisitar apenas um produto, através da operação `SalesApi.retrofitService`. Os demais conceitos de utilização de `coroutine` são os mesmos já apresentados.

Para finalizar esse `ViewModel`, crie a função `onCleared()`, que será automaticamente invocada quando esse `ViewModel` não for mais necessário e for destruído:

```

override fun onCleared() {
    Log.i(TAG, "onCleared")
    super.onCleared()
    viewModelJob.cancel()
}

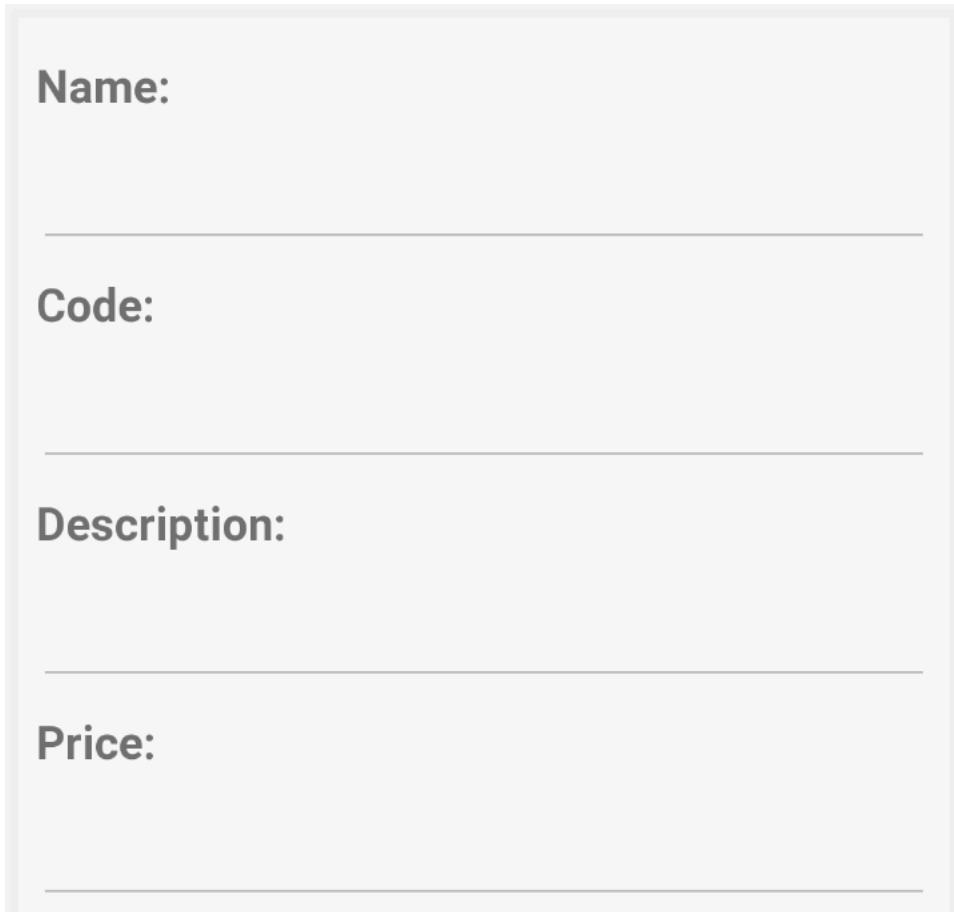
```

Isso fará com que qualquer tarefa em execução seja finalizada.

Esse `ViewModel` agora poderá ser utilizado dentro do fragmento para exibir os detalhes do produto, a ser criado nas próximas seções desse capítulo.

### 9.5.3 - Criando o layout do fragmento dos detalhes do produto

O layout do fragmento para exibir os detalhes do produto é bem simples, uma vez que não possui interação com o usuário, pois ele não irá editar nada nessa tela. Por isso ela deve conter campos apenas para exibir as informações do produto. Porém, é importante considerar que essa tela pode ser exibida em dispositivos pequenos ou em modo paisagem, por isso é importante permitir que o usuário possa rolar a tela para exibir todo o seu conteúdo. Veja a seguir como ela deve ficar:



Layout com os detalhes do produto

Para começar, crie um novo arquivo na pasta `res\layout` , chamado `fragment_product_detail.xml`:

```
<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

        <variable
            name="productDetailViewViewModel"
            type="br.com.siecola.androidproject02.productdetail.ProductDetailViewMod\
el" />
    </data>

    <ScrollView
        android:layout_width="match_parent"
```

```
    android:layout_height="match_parent">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/layout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <!--Visual components here-->

    </androidx.constraintlayout.widget.ConstraintLayout>
</ScrollView>
</layout>
```

A estrutura desse novo layout contém a declaração da variável `productDetailViewModel`, bem como o componente `ScrollView`, que possibilita o usuário rolar a tela em dispositivos menores. Dentro dele há o `ConstraintLayout`, onde os componentes para a exibição dos detalhes do produto devem ser distribuídos e alinhados entre si.

Para começar, crie os dois componentes para exibirem o nome do produto, dentro do componente `ConstraintLayout`:

```
<TextView
    android:id="@+id/textView2"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="8dp"
    android:labelFor="@id/edtName"
    android:text="Name:"
    android:textSize="20sp"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

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

```
    android:ems="10"
    android:inputType="textCapWords"
    android:textSize="24sp"
    android:enabled="false"
    android:text="@{productDetailViewModel.product.name}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView2" />
```

Apesar do segundo componente também poder ser uma caixa de texto não editável, fica mais agradável visualmente deixá-la como uma editável, porém, com essa opção desabilitada como pode ser visto no atributo `android:enabled="false"`.

Perceba que o atributo `android:text="@{productDetailViewModel.product.name}"` recebe o nome do produto através do `ViewModel` declarado no início do arquivo XML.

Agora crie outros dois componentes para o código do produto:

```
<TextView
    android:id="@+id/textView3"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:labelFor="@+id/editName"
    android:text="Code:"
    android:textSize="20sp"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/editName" />
```

```
<EditText
    android:id="@+id/editCode"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:ems="10"
    android:inputType="textCapCharacters"
    android:textSize="24sp"
    android:enabled="false"
```

```
    android:text="@{productDetailViewModel.product.code}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView3" />
```

Da mesma forma, o código do produto é preenchido no segundo componente através do ViewModel.

Para a descrição do produto, proceda da mesma forma:

```
<TextView
    android:id="@+id/textView5"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:labelFor="@+id/edtDescription"
    android:text="Description:"
    android:textSize="20sp"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/edtCode" />

<EditText
    android:id="@+id/edtDescription"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:ems="10"
    android:inputType="textCapSentences"
    android:textSize="24sp"
    android:enabled="false"
    android:text="@{productDetailViewModel.product.description}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView5" />
```

E finalmente, para o preço do produto:

```
<TextView  
    android:id="@+id/textView4"  
    android:layout_width="0dp"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="8dp"  
    android:layout_marginTop="8dp"  
    android:layout_marginEnd="8dp"  
    android:labelFor="@+id/editPrice"  
    android:text="Price:"  
    android:textSize="20sp"  
    android:textStyle="bold"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintHorizontal_bias="0.0"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/editDescription" />  
  
<EditText  
    android:id="@+id/editPrice"  
    android:layout_width="0dp"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="8dp"  
    android:layout_marginTop="8dp"  
    android:layout_marginEnd="8dp"  
    android:ems="10"  
    android:inputType="numberSigned|numberDecimal"  
    android:textSize="24sp"  
    android:enabled="false"  
    app:productPrice="@{productDetailViewModel.product.price}"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintHorizontal_bias="0.0"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/textView4" />
```

A diferença aqui está no momento de mostrar o preço, onde o *binder* que foi criado no arquivo ProductBindingAdapters é utilizado para converter o preço através da instrução app:productPrice="@{productDetailViewModel.product.price}" como foi feito no layout que exibia o item do produto na lista.

Esse layout está pronto para ser exibido em seu fragmento, o que será criado na próxima seção.

#### 9.5.4 - Criando o fragmento para exibir os detalhes do produto

Para o fragmento que exibirá os detalhes do produto, comece criando, dentro do pacote productdetail, uma nova classe chamada ProductDetailFragment, como no trecho a seguir:

```
import android.os.Bundle
import android.util.Log
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import br.com.siecola.androidproject02.databinding.FragmentProductDetailBinding

private const val TAG = "ProductDetailFragment"

class ProductDetailFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?): View? {
        Log.i(TAG, "Creating ProductDetailFragment")

        val binding = FragmentProductDetailBinding.inflate(inflater)

        binding.setLifecycleOwner(this)

        //Fetch the product code and create the ViewModel here

        return binding.root
    }
}
```

Até o momento essa classe não possui nada de novo, pois é ela é muito semelhante à que foi criada para a lista de produtos. Porém, ela deve receber o código do produto através de um argumento para poder criar o ViewModel, mas para isso é necessário trabalhar no mecanismo de navegação que foi criado em algumas seções anteriores, no arquivo `navigation_graph.xml`.

### 9.5.5 - Criando a navegação entre os fragmentos

A base do mecanismo de navegação da aplicação já havia sido criada nesse capítulo, como mostra o trecho a seguir do arquivo `navigation_graph.xml`:

```
<?xml version="1.0" encoding="utf-8"?>

<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/navigation_graph"
    app:startDestination="@+id/fragmentProductsList">

    <fragment
        android:id="@+id/fragmentProductsList"
        android:name="br.com.siecola.androidproject02.product.ProductsListFragment"
        android:label="fragment_products_list"
        tools:layout="@layout/fragment_products_list" >
    </fragment>
</navigation>
```

O trecho anterior mostra o que foi criado até o momento, que apenas mostra o fragmento com a lista de produtos. Agora são necessários os seguintes passos:

- Acrescentar o segundo fragmento que mostra os detalhes do produto;
- Indicar no primeiro fragmento que ele possui um ação de navegação para que mostre os detalhes do produto;
- Criar o argumento que será passado ao fragmento de detalhes do produto e que deverá carregar seu código quando for chamado.

Para isso, crie a representação do segundo fragmento, logo após o fechamento da tag `fragment`, como no trecho a seguir:

```
<fragment
    android:id="@+id/fragmentProductDetail"
    android:name="br.com.siecola.androidproject02.productdetail.ProductDetailFragmen\
t"
    android:label="fragment_product_detail"
    tools:layout="@layout/fragment_product_detail" >
</fragment>
```

O princípio é o mesmo utilizado para criar a representação do primeiro fragmento, ou seja, o layout e a classe do fragmento que mostram os detalhes do produto são utilizados, além da definição e uma identificação para o novo fragmento.

Agora volte à definição do primeiro fragmento e acrescente a ação que permitirá a navegação da tela com a lista de produtos para a de detalhes:

```
<fragment
    android:id="@+id/fragmentProductsList"
    android:name="br.com.siecola.androidproject02.product.ProductsListFragment"
    android:label="fragment_products_list"
    tools:layout="@layout/fragment_products_list" >
    <action
        android:id="@+id/action_showProductDetail"
        app:destination="@+id/fragmentProductDetail" />
</fragment>
```



O trecho que deve ser acrescentado é apenas o que está dentro da tag `action` no trecho anterior, pois o restante do conteúdo já existia nesse arquivo.

A tag `action` define que existe uma ação de navegação do fragmento com a lista de produtos para o fragmento que está definido com a identificação `fragmentProductDetail`.

A última coisa que falta a ser feira nesse arquivo é a definição do argumento que `fragmentProductDetail` deve receber ao ser invocado. Para isso acrescente o seguinte trecho antes do fechamento da tag que o define:

```
<argument
    android:name="productCode"
    app:argType="string" />
```

Isso obrigará a passagem de um parâmetro do tipo `String`, com o nome de `productCode`, toda vez que esse fragmento for invocado.

Veja como deve ficar o arquivo completo:

```
<?xml version="1.0" encoding="utf-8"?>

<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/navigation_graph"
    app:startDestination="@+id/fragmentProductsList">

    <fragment
        android:id="@+id/fragmentProductsList"
        android:name="br.com.siecola.androidproject02.product.ProductsListFragment"
        android:label="fragment_products_list"
        tools:layout="@layout/fragment_products_list" >
```

```

<action
    android:id="@+id/action_showProductDetail"
    app:destination="@+id/fragmentProductDetail" />
</fragment>

<fragment
    android:id="@+id/fragmentProductDetail"
    android:name="br.com.siecola.androidproject02.productdetail.ProductDetailFra\mment"
    android:label="fragment_product_detail"
    tools:layout="@layout/fragment_product_detail" >
    <argument
        android:name="productCode"
        app:argType="string" />
</fragment>
</navigation>

```

A aba Design do Android Studio mostra uma representação gráfica desse arquivo, que pode ser muito interessante para o entendimento da navegação do aplicativo, principalmente se muitas telas existirem:

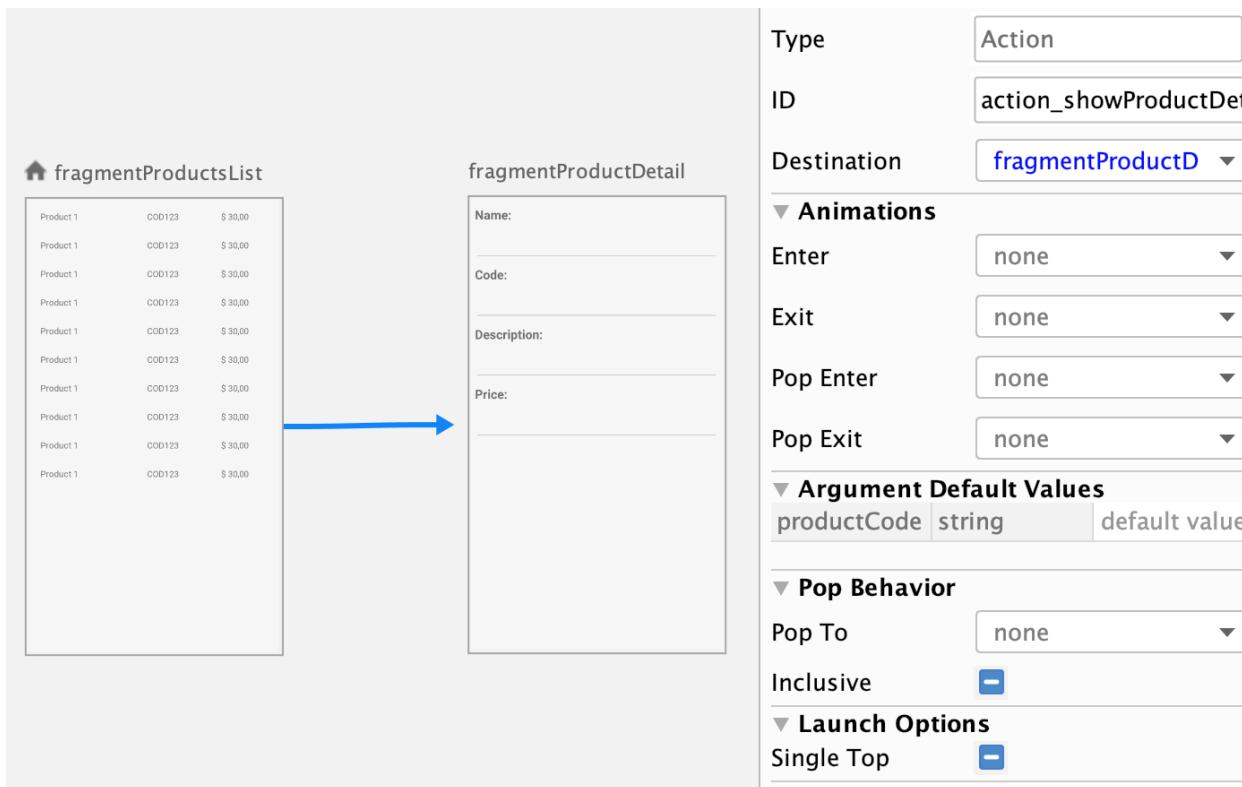


Gráfico de navegação do aplicativo

Nessa aba é possível visualizar a interação entre as telas, além das ações entre elas e seus argumentos.



Compile todo o projeto para que as alterações feitas nesse arquivo passem a valer.

Agora que todo o mecanismo de navegação está concluído, é possível utilizá-lo nos dois fragmentos da aplicação. A ideia é realizar os seguintes procedimentos:

- Tratar o evento de clique do usuário em um item da lista de produtos na primeira tela;
- Navegar para o fragmento de detalhes do produto, passando seu código como argumento;
- Resgatar o argumento com o código do produto quando o fragmento de detalhes for invocado.

Para começar, vá até a classe `ProductsListFragment`, na função `onCreateView` e localize o seguinte trecho de código:

```
binding.rcvProducts.adapter = ProductAdapter(ProductAdapter.ProductClickListener {  
    Log.i(TAG, "Product selected: ${it.name}")  
})
```

Esse código captura o evento de clique do usuário em um item da lista de produtos na primeira tela do aplicativo e imprime uma mensagem no Logcat. A variável `it` dentro desse `listener` é a instância de `Product` que representa o produto que foi clicado pelo usuário.

O que falta agora é inserir dentro desse `listener`, logo após a mensagem de log que está sendo gerada, a navegação para o o fragmento de detalhes, passando o código do produto, como no trecho a seguir:

```
binding.rcvProducts.adapter = ProductAdapter(ProductAdapter.ProductClickListener {  
    Log.i(TAG, "Product selected: ${it.name}")  
    this.findNavController()  
        .navigate(ProductsListFragmentDirections.actionShowProductDetail(it.code))  
})
```

Aqui está sendo utilizado um componente chamado `NavController` para fazer a navegação entre os fragmentos. Repare que a classe `ProductsListFragmentDirections`, foi gerada automaticamente pelo Android Studio a partir do que foi feito em `navigation_graph.xml`. Essa classe contém as possíveis ações de navegação que podem ser realizadas pelo fragmento `ProductsListFragment`.

Dentro da ação `actionShowProductDetail` está definido que ela deve navegar ao fragmento `fragmentProductDetail`:

```
<action
    android:id="@+id/action_showProductDetail"
    app:destination="@+id/fragmentProductDetail" />
```

Que por sua vez possui exige um argumento do tipo `String`:

```
<argument
    android:name="productCode"
    app:argType="string" />
```

Essa técnica, chamada de `SafeArgs`, garante a troca de argumentos entre fragmentos.

Para concluir, volte à classe `ProductDetailFragment` para pegar o argumento passado e criar a instância do `ViewModel` responsável por exibir os detalhes do produto. Isso pode ser feito com o trecho a seguir, que deve ser inserido dentro da função `onCreateView`, antes da instrução `return binding.root`:

```
val productCode = ProductDetailFragmentArgs.fromBundle(arguments!!).productCode

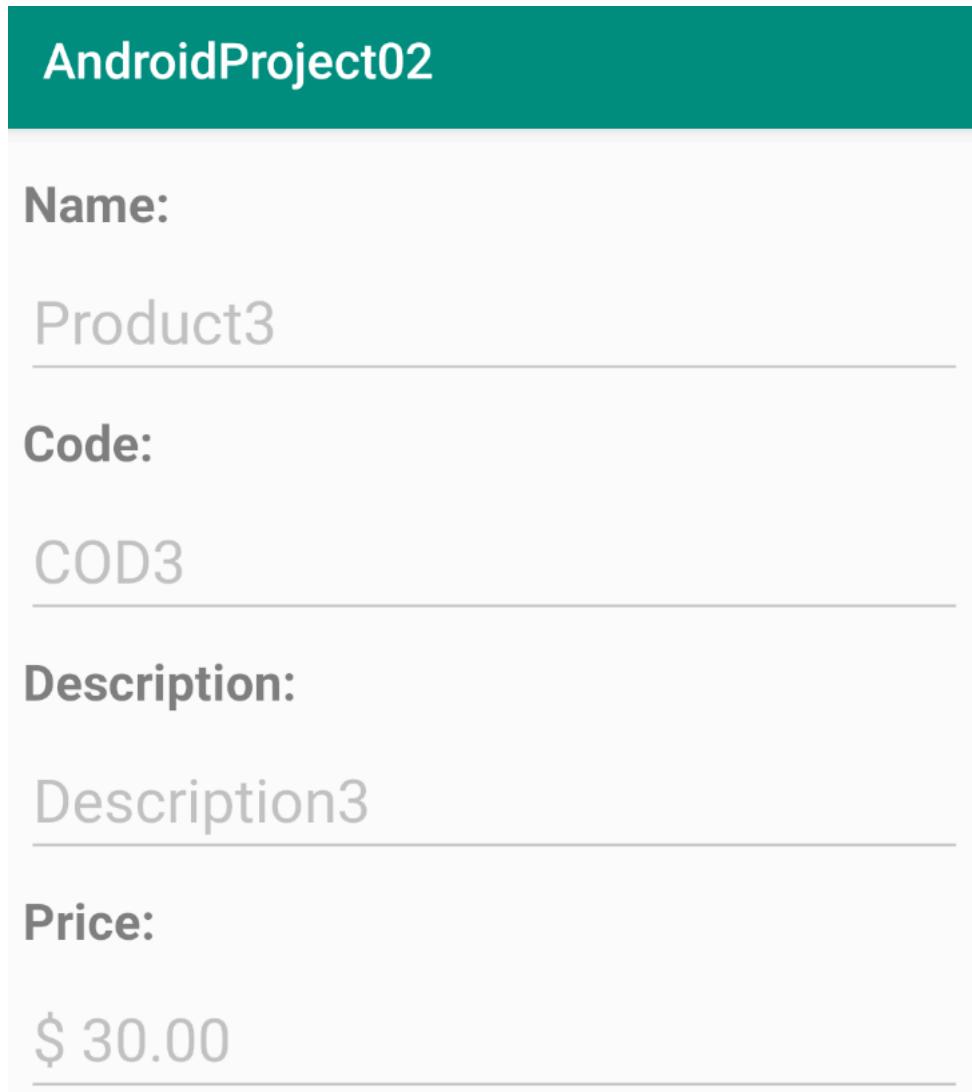
val productDetailViewModelFactory = ProductDetailViewModelFactory(productCode)

binding.productDetailViewModel = ViewModelProviders.of(
    this, productDetailViewModelFactory).get(ProductDetailViewModel::class.java)
```

A primeira linha desse trecho se encarrega de retirar o argumento, que é o código do produto, que o fragmento exige que seja passado. A segunda linha cria a instância do `ViewModelFactory`, passando o código do produto em seu construtor.

A última linha se encarrega de criar efetivamente a instância do `ViewModel` de detalhes de produto, utilizando o `factory` criado anteriormente. Nesse instante ele faz a solicitação do produto ao provedor de serviços de vendas, utilizando seu código como parâmetro.

Para testar a implementação já com a navegação, execute a aplicação novamente. Quando a lista aparecer, clique em um item. Essa ação deverá mostrar os detalhes do produto na tela criada recentemente, como mostra a figura a seguir:



Tela com os detalhes do produto

Para voltar à lista, clique no botão de voltar.

Limpe o Logcat e execute a aplicação novamente. Clique em um produto, depois gire o dispositivo algumas vezes e volte para a tela com a lista de produtos. Os logs deverão ser semelhantes aos da figura a seguir:

I/ProductsListFragment: Creating ProductsListFragment  
I/ProductListViewModel: Preparing to request products list  
I/ProductListViewModel: Products list requested  
I/ProductListViewModel: Loading products  
I/ProductListViewModel: Number of products 3  
I/ProductsListFragment: Product selected: Product2  
I/ProductDetailFragment: Creating ProductDetailFragment  
I/ProductDetailViewModel: Preparing to request a product by its code  
I/ProductDetailViewModel: Product requested by code  
I/ProductDetailViewModel: Loading product by its code  
I/ProductDetailViewModel: Name of the product Product2  
I/ProductDetailFragment: Creating ProductDetailFragment  
I/ProductDetailFragment: Creating ProductDetailFragment  
I/ProductsListFragment: Creating ProductsListFragment

#### Logs da aplicação

Repare que embora os dois fragmentos estejam sendo criados e recriados algumas vezes, por conta da rotação do dispositivo ou pela navegação em si, é feita apenas uma requisição ao provedor de vendas para buscar a lista e uma para carregar o produto pelo seu código. Isso se deve ao fato dos dois ViewModels manterem-se ativos durante o ciclo de vida dos fragmentos.

## 9.6 - Atualizando a lista de produtos

Existe ainda uma melhoria que pode ser feita na lista de produtos, que é disponibilizar um gesto para que o usuário possa solicitar a atualização dessa lista. Isso pode ser feito com o componente SwipeRefreshLayout. Para isso, abra o arquivo `fragment_products_list.xml` e localize o seguinte trecho:

```
<androidx.recyclerview.widget.RecyclerView  
    android:id="@+id/rvProducts"  
    android:layout_width="0dp"
```

Essa é a declaração do RecyclerView, que exibe a lista de produtos. O que deve ser feito é envolver-lo com o SwipeRefreshLayout:

```
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout  
    android:id="@+id/productsRefresh"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
<androidx.recyclerview.widget.RecyclerView  
    android:id="@+id/rcvProducts"  
    android:layout_width="0dp"  
    android:layout_height="0dp"  
    android:layout_marginStart="8dp"  
    android:layout_marginTop="8dp"  
    android:layout_marginEnd="8dp"  
    android:layout_marginBottom="8dp"  
    android:orientation="vertical"  
    app:layoutManager="androidx.recyclerview.widget.GridLayoutManager"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    app:productsList="@{productListViewModel.products}"  
    app:spanCount="1"  
    tools:listitem="@layout/item_product" />  
</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

Repare na segunda linha desse trecho que a identificação para o componente foi criada com o valor productsRefresh.

Agora volte à classe ProductListViewModel e crie a seguinte função:

```
fun refreshProducts() {  
    _products.value = null  
    getProducts()  
}
```

Ela será invocada quando o usuário decidir realizar o sincronismo da lista de produtos. E para isso, é necessário criar um *listener* no final do método onCreateView da classe ProductsListFragment para tratar tal evento:

```
binding.productsRefresh.setOnRefreshListener {  
    Log.i(TAG, "Refreshing products list")  
    productListViewModel.refreshProducts()  
  
    binding.productsRefresh.isRefreshing = false  
}
```

Dessa forma quando o usuário clicar na lista de produtos e arrastar pra baixo, será invocado esse *listener*, que por sua vez faz com que o ProductListViewModel recarregue a lista de produtos.

## 9.7 - Conclusão

Esse capítulo apresentou os seguintes conceitos:

- Criação de lista com o componente RecyclerView;
- Criação de adaptadores para listas com o RecyclerView;
- Criação de *binders* customizados;
- Fragmentos para exibição de porções de tela;
- Navegação entre fragmentos utilizando SafeArgs.

Também foi aprofundado o conceito de ViewModel, analisando seu comportamento principalmente com relação a navegação entre fragmentos.

O próximo capítulo inicia um novo desafio: criar uma aplicação para se integrar com o Google Firebase.

# 10 - Recebendo mensagens pelo Firebase Cloud Messaging

Esse livro introduz, a partir desse capítulo, conceitos sobre como criar aplicativos Android que estejam conectados a algum serviço de *cloud computing*, a começar pelo **Firebase Cloud Messaging**, do Google, que permite o envio de mensagens através de notificações aos dispositivos dos usuários.

O Firebase, como já foi dito em capítulos anteriores, é a plataforma do Google que oferece serviços para aplicações web e para dispositivos móveis, como **Android** e **iOS**, permitindo a rápida criação de sistemas baseados em *cloud computing* sem a necessidade de se gerenciar infraestrutura de servidores ou sistemas computacionais.

Os capítulos seguintes abordarão outros serviços do Firebase e como construir aplicativos Android para desfrutarem de suas vantagens.

O objetivo desse capítulo é criar um aplicativo que possa receber notificações através do Firebase Cloud Messaging e exibir seu conteúdo na tela para o usuário. Seu funcionamento será da seguinte forma:

- Quando a mensagem for recebida pelo aplicativo, ele deverá gerar uma notificação ao usuário;
- O usuário, quando ver a notificação, poderá clicar nela para navegar para dentro da aplicação;
- De dentro da aplicação, a mensagem será interpretada e exibida em uma tela ao usuário;
- Tal tela será construída com um **Fragment** e seu arquivo de layout **XML**, utilizando o mesmo conceito de **ViewModel** já explicado em capítulos anteriores.

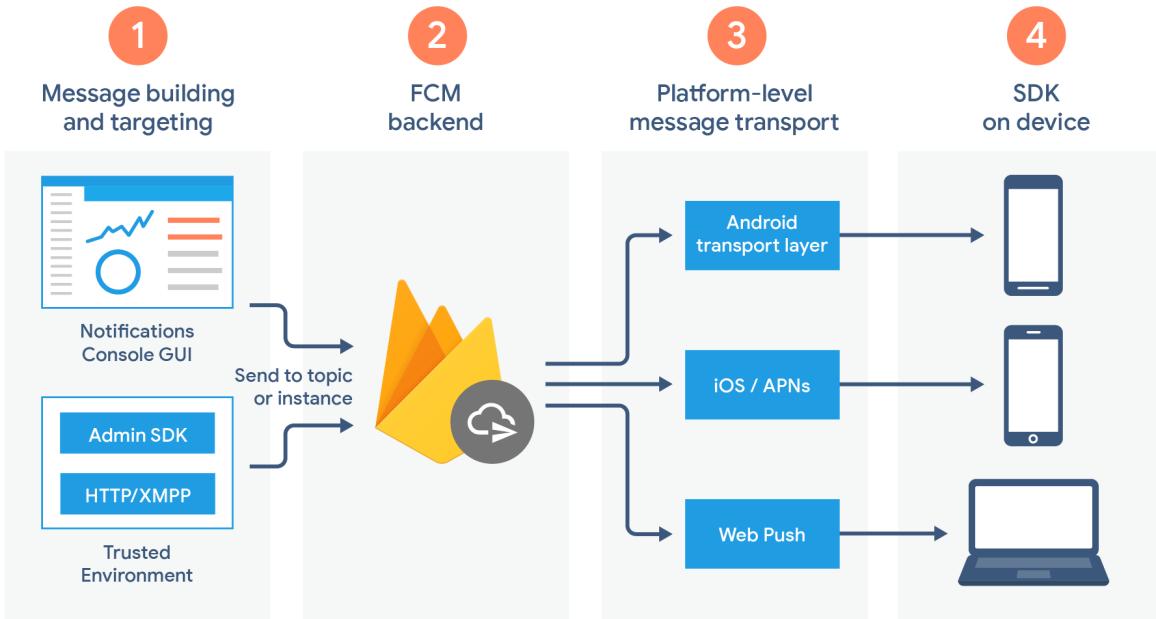
Embora possa parecer bem simples, a ideia é apenas criar uma base para que o leitor possa desenvolver aplicações mais complexas utilizando esse mecanismo.

## 10.1 - O que é Firebase Cloud Messaging

Talvez esse seja um dos serviços mais conhecidos do Firebase. Com ele é possível enviar notificações para uma aplicação em um dispositivo móvel, permitindo que o usuário possa ser notificado sobre algum evento.

Esse mecanismo pode ser invocado por uma aplicação de backend ou até mesmo através de eventos automáticos ou agendados.

O Firebase Cloud Messaging se encarrega totalmente de entregar a mensagem, cuidando da garantia de entrega, bem como o armazenamento da mesma, caso o dispositivo não esteja conectado à Internet no momento da entrega da mensagem.



Arquitetura do Firebase Fonte: <https://firebase.google.com/docs/cloud-messaging/fcm-architecture>

Através do Firebase Cloud Messaging, ou FCM, é possível enviar mensagens de forma individual ou através de tópicos, para aplicações iOS, Android ou Web. Porém, para que essas aplicações possam receber essas notificações, é necessário que elas se registrem em um projeto do Firebase. Dessa forma uma aplicação de backend pode utilizar o FCM para enviar mensagens aos dispositivos registrados.

Esse capítulo se restringe aos detalhes da construção da aplicação Android, mas se tiver interesse em aprender a criar uma aplicação de backend para **envio de notificações pelo FCM**, consulte o seguinte [livro<sup>28</sup>](#).

Do ponto de vista da construção do sistema de envio de notificações para aplicativos Android, os seguintes passos devem ser realizados pelo desenvolvedor:

- **Criação do projeto no Firebase:** esse projeto representa a instância do Firebase para todo o sistema. O mesmo projeto criado aqui será útil para os demais capítulos, onde outros serviços do Firebase serão explorados;
- **Registro do projeto Android no Firebase:** esse processo habilita a aplicação a se registrar no Firebase para ter acesso aos serviços habilitados ou configurados no projeto do Firebase;
- **Configuração do projeto Android:** esse processo é bem simples e consiste em adicionar um arquivo de configuração gerado no passo anterior. Com esse arquivo a aplicação Android fica toda configurada e preparada para acessar os serviços do Firebase sobre o projeto criado no primeiro passo.

Analizando a figura anterior, o processo de envio de notificações funciona da seguinte forma:

<sup>28</sup><https://www.casadocodigo.com.br/products/livro-gae>

- A aplicação se registra no FCM, assim que abre pela primeira vez. Esse processo gera um token único que identifica essa aplicação no FCM;
- No **passo 1** da figura, uma aplicação Web pode enviar uma mensagem para um dispositivo identificado pelo token de registro no FCM, como descrito anteriormente;
- O FCM então localiza o dispositivo no **passo 2** e descobre qual é o seu tipo (iOS, Android);
- Sabendo qual é o tipo do dispositivo, o FCM então despacha a mensagem no **passo 3**;
- O dispositivo então recebe a mensagem quando ele estiver conectado na Internet, como mostra o **passo 4**.

É importante ressaltar que o FCM cuida de gerenciar filas de entrega de mensagens, assim como as tentativas de entrega das mensagens caso o dispositivo não esteja conectado à Internet no momento do envio das notificações. Ele também cuida do registro de todas as aplicações que estão interessadas em receber as notificações de um determinado projeto criado no Firebase.

Dessa forma, a aplicação de backend apenas instrui o FCM a enviar a mensagem a um dispositivo específico, ou a um grupo deles. O FCM então se encarrega de todo o trabalho de fazer a entrega das notificações.

## 10.2 - Arquitetura do projeto Android com FCM

O novo projeto desse capítulo será composto pelas seguintes partes:

- **FCM Service**: esse serviço será responsável por receber as notificações do FCM, assim como a informação de que um novo token de registro do FCM foi gerado para o dispositivo;
- **MainActivity**: será responsável por criar a navegação dentro da aplicação;
- **Product Info Fragment**: responsável por exibir o fragmento de tela;
- **Product Info View Model**: irá armazenar as informações a serem exibidas ao usuário;
- **Product Info View**: irá efetivamente exibir as informações armazenadas no **ViewModel** aos usuários.

O diagrama a seguir mostra a interação entre essas partes e o Firebase Cloud Messaging:

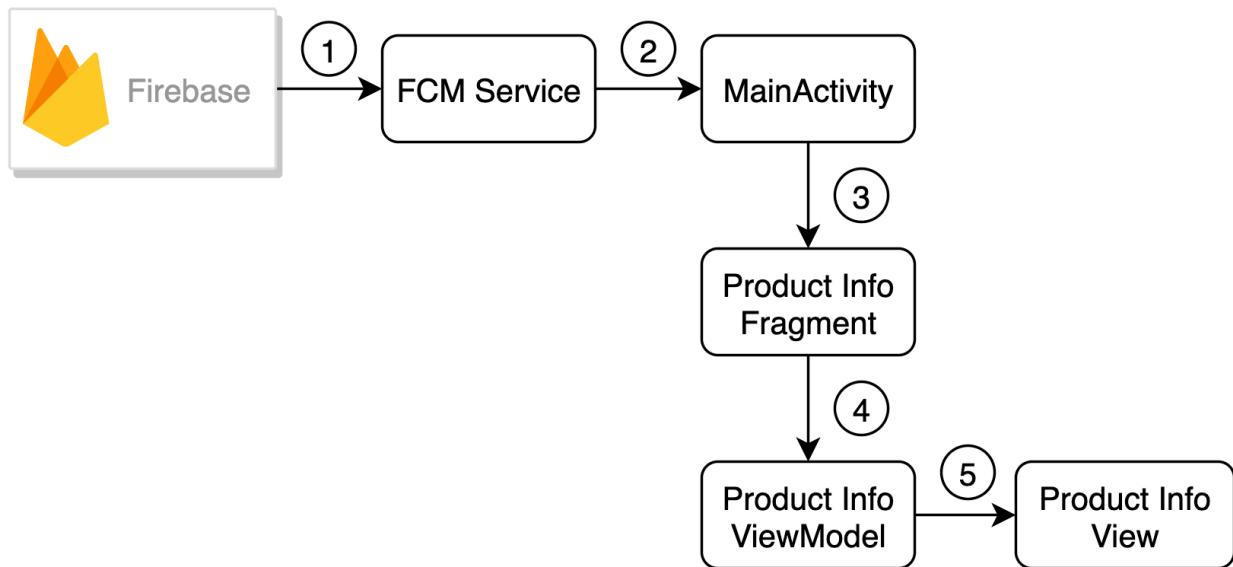


Diagrama do projeto

A seguir, a descrição do fluxo de execução da aplicação, de acordo com os passos informados no diagrama anterior:

- 1) O **FCM Service** é a parte da aplicação que receberá as mensagens do Firebase Cloud Messaging, bem como o token de registro que o identifica unicamente nele;
- 2) Quando a mensagem for recebida pelo **FCM Service**, ele será encarregado de gerar uma notificação a ser exibida ao usuário, com algumas informações customizadas e sobre a mensagem recebida. Dessa forma a **MainActivity** será executada quando o usuário clicar na notificação;
- 3) Quando a **MainActivity** receber a notificação, irá invocar o fragmento **ProductInfo**, passando a mensagem recebida;
- 4) De posse da mensagem recebida **ProductInfoFragment** irá interpretá-la e salvá-la em **ProductInfoViewModel** para que possa ser exibida;
- 5) Finalmente **ProductInfoView** exibirá a mensagem recebida ao usuário, expondo seus dados ao usuário.

Os capítulos a seguir detalham os passos para a criação do projeto no Android Studio e no Firebase.

## 10.3 - Criando o novo projeto no Android Studio

No Android Studio, crie o novo projeto, seguindo o que foi feito nos dois últimos projetos desse livro, com as seguintes informações:

- **Tipo:** Empty Activity
- **Nome do projeto:** AndroidProject03

- **Pacote base:** br.com.siecola
- **Linguagem:** Kotlin
- **Versão mínima do SDK:** 21

No arquivo `build.gradle` do projeto, adicione as seguintes dependências:

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.2.1"
classpath 'com.google.gms:google-services:4.3.3'
```

O primeiro já é conhecido, que trata da funcionalidade de se utilizar argumentos seguros na chamada de fragmentos. O segundo é para permitir que a aplicação acesse serviços do Google, como o Firebase.

Agora no arquivo `build.gradle` do módulo `app`, tenha certeza de aplicar todos os seguinte plugins:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'
apply plugin: "androidx.navigation.safeargs"
apply plugin: 'com.google.gms.google-services'
```

O novo plugin que aparece aqui é o do Google Services.

Nesse mesmo arquivo, na seção `android`, adicione as seguintes configurações, que já foram utilizadas no projeto anterior:

```
dataBinding {
    enabled = true
}
androidExtensions {
    experimental = true
}
```

Para finalizar, ainda nesse arquivo, configure a seção `dependencies` para ficar com o trecho a seguir:

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.1.0'  
    implementation 'androidx.core:core-ktx:1.2.0'  
  
    //Constraint Layout  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
  
    // ViewModel and LiveData  
    implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"  
  
    //Navigation  
    implementation 'androidx.navigation:navigation-fragment-ktx:2.2.1'  
    implementation 'androidx.navigation:navigation-ui-ktx:2.2.1'  
  
    //Firebase  
    implementation 'com.google.firebaseio:firebase-messaging:20.1.0'  
  
    //Moshi  
    implementation "com.squareup.moshi:moshi:1.8.0"  
    implementation "com.squareup.moshi:moshi-kotlin:1.8.0"  
  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'  
}
```

Em relação ao projeto anterior, foi adicionada a biblioteca do Firebase Messaging, que permite que a aplicação se registre no FCM e receba mensagens através dele.



Depois de adicionar as dependências ao projeto, lembre-se de sincronizá-lo para baixar as novas bibliotecas.

A base do projeto ainda não está pronta. Inclusive ele nem compila, pois depende de um arquivo de configurações que será gerado pelas instruções da próxima seção desse capítulo.

## 10.4 - Criando o projeto no Firebase

O Firebase oferece planos gratuitos de avaliação de seus serviços, por isso a criação do projeto para esse e os próximos capítulos pode ser feita sem nenhum custo.

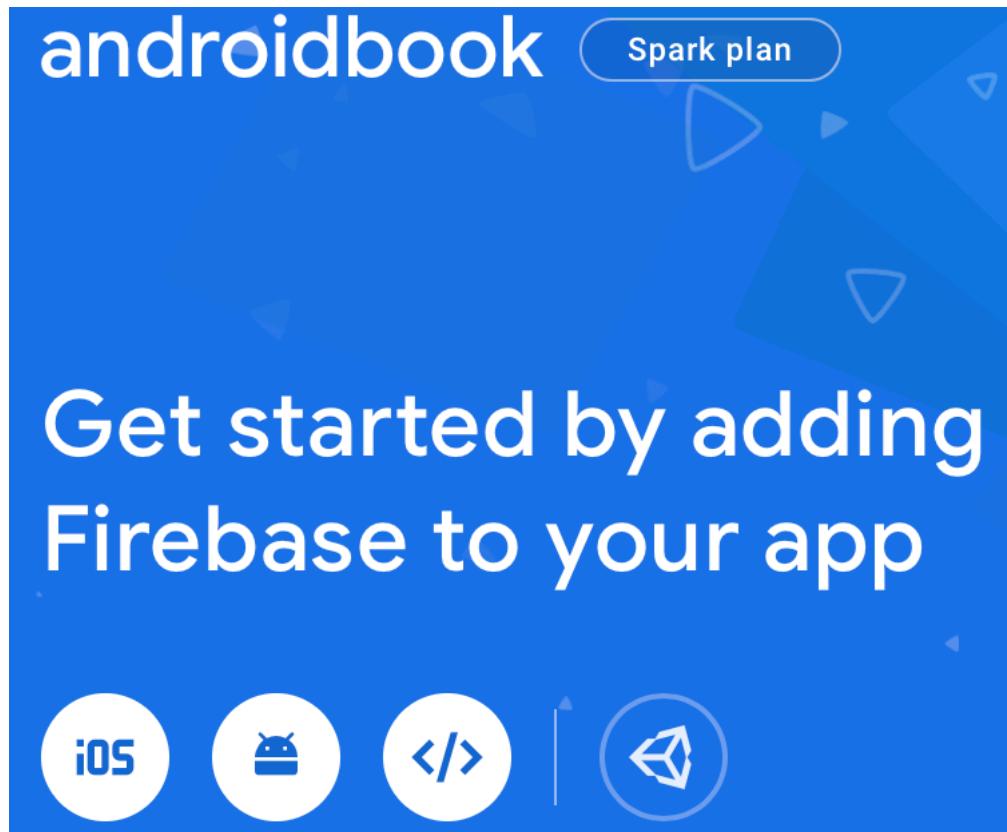
Para começar a criar um projeto no Firebase, acesse o seguinte link<sup>29</sup> e clique no opção Go to console, localizado no canto superior direito da página. É necessário estar logado com uma conta do Google para começar esse processo.

Ao entrar no console do Firebase, clique no botão + Add project e digite o nome que desejar. Aguarde o processo de criação do projeto no Firebase.



Para esse novo projeto no Firebase, não é necessário habilitar a opção Google Analytics.

Quando o projeto estiver criado, entre nele. Perceba que na primeira página existe uma opção para adicionar aplicativos a esse projeto, como pode ser observado na figura a seguir:

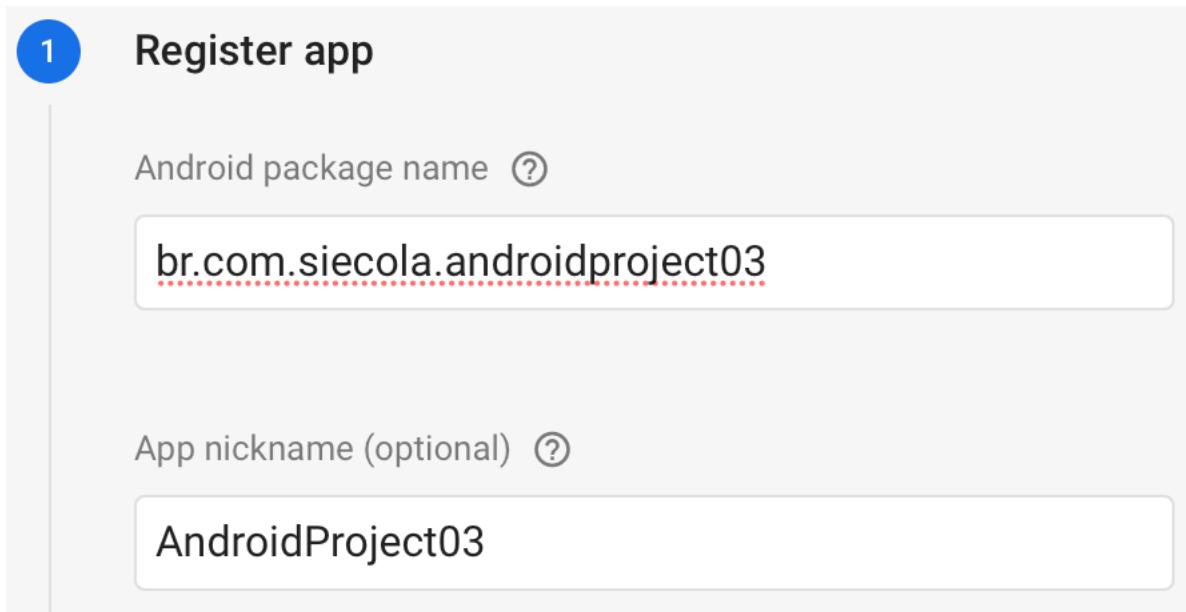


Adicionar aplicativos ao projeto

Selecione a opção com o símbolo do Android para começar o processo para adicionar o aplicativo que será desenvolvido nesse capítulo a esse projeto do Firebase.

Na próxima tela, preencha o campo `Android package name` com o pacote base do projeto criado na seção anterior, bem como seu nome no campo `App nickname`:

<sup>29</sup><http://firebase.google.com/>



O terceiro campo, de nome `Debug signing certificate SHA-1`, não necessita ser configurada para esse projeto, mas será utilizado para os capítulos seguintes.

Para passar para o próximo passo, clique no botão `Register app`. Nesse momento o Firebase irá gerar um arquivo de nome `google-services.json` que deve ser baixado e colocado dentro da pasta `app` do projeto no Android Studio, como informa a figura da página do Firebase desse passo.

De posse desse arquivo, é possível compilar e executar o novo projeto no Android Studio.

O arquivo `google-services.json` não deve ser colocado em um repositório público, pois ele possui as credenciais de acesso ao projeto do Firebase que está sendo criado.

Clique no botão `Next` do Firebase para continuar para o próximo passo. O terceiro passo consiste em instruções que devem ser feitas para configurar o projeto Android para se trabalhar com o Firebase, algo que já foi feito na seção passada desse capítulo.

Agora volte ao Android Studio e execute a aplicação em um dispositivo real ou em um emulador com o Google Play Services instalado. Caso ela não compile, verifique as dependências e principalmente se o arquivo `google-services.json` está no local adequado, ou seja, dentro da pasta `app`.

De volta do Firebase, clique no botão `Next` para ir ao último passo de criação do projeto. O Firebase irá aguardar até que o aplicativo正在被 Android Studio executado e comunique com ele, como mostra a figura a seguir:

**4****Run your app to verify installation**

- Checking if the app has communicated with our servers. You may need to uninstall and reinstall your app.

Aguardando o aplicativo se comunicar com o Firebase

Embora esse seja um passo adicional, é interessante para verificar que tudo foi configurado corretamente.

Se por alguma razão o aplicativo não conseguir se conectar com o Firebase nesse momento, não se preocupe. A próxima seção irá executar outros passos importantes para verificar se tudo está funcionando corretamente.

Finalize a configuração do novo aplicativo no Firebase para voltar à página de seu console principal.

## 10.5 - Criando a classe de serviço do FCM

Para que uma aplicação Android receba notificações pelo Firebase Cloud Messaging, é necessário construir uma classe que estenda de uma classe especial, chamada `FirebaseMessagingService`, que possui duas funções importantes:

- `onNewToken`: essa função é chamada assim que o dispositivo se registra no FCM, para receber notificações do projeto que foi criado no Firebase, configurado na aplicação através do arquivo `google-services.json`;
- `onMessageReceived`: nessa função é possível receber a mensagem enviada ao dispositivo.

Com apenas essas duas funções é possível preparar o projeto para receber notificações do FCM, bem como ser avisado quando ele se registra, para a obtenção do token que permite o envio de mensagens específicas a um dispositivo.

Para começar, crie um novo pacote chamado `messaging` e dentro dele a nova classe chamada `FCMService`, como no trecho a seguir:

```
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.util.Log
import androidx.core.app.NotificationCompat
import br.com.siecola.androidproject03.MainActivity
import br.com.siecola.androidproject03.R
import com.google.firebase.messaging.FirebaseMessagingService
import com.google.firebase.messaging.RemoteMessage

private const val TAG = "FCMService"

class FCMService : FirebaseMessagingService() {

}
```

A ideia dessa classe é, dentre outras coisas, receber o token obtido durante o processo de registro no FCM. Essencialmente esse token deve ser informado à aplicação responsável pelo envio das mensagens para o FCM. Dessa forma ela pode despachar uma mensagem a cada usuário, através de seu token de registro único. Para isso, construa a função `onNewToken`, como no trecho a seguir:

```
override fun onNewToken(token: String) {
    Log.d(TAG, "FCM token: $token")
}
```

Aqui o token está apenas sendo impresso no Logcat. Mas, esse seria um ponto ideal para fazer uma chamada REST para a aplicação que envia as mensagens através do FCM, informando que o usuário dessa aplicação está registrado no FCM com esse token único.

Essa função será chamada na primeira vez que a aplicação for executada e conseguir se registrar no FCM. Da mesma forma, será chamada toda vez que o token precisar ser reciclado.

A segunda função a ser criada nessa classe é a `onMessageReceived`, como no trecho a seguir:

```
override fun onMessageReceived(remoteMessage: RemoteMessage) {
    remoteMessage.data.isNotEmpty().let {
        Log.d(TAG, "Payload: " + remoteMessage.data)

    }
}
```

O objeto do tipo `RemoteMessage` passado como parâmetro contém a mensagem recebida dentro de seu atributo `data`, que é um `Map` com chaves e valores. As chaves, bem como o formato do dado, é definido pela aplicação que publica a mensagem no FCM.



Essa função será invocada pelo sistema Android independente do estado da aplicação, ou seja ela pode estar em segundo plano ou até mesmo fechada.

Como dito no início desse capítulo, existe um exemplo de uma aplicação construída para ser hospedada no Google App Engine nesse [repositório<sup>30</sup>](#), além de um [livro<sup>31</sup>](#) que ensina como criá-la. Essa aplicação está preparada para enviar uma mensagem com a chave `product`, contendo informações de um produto cadastrado em sua base de dados.



Para testar esse projeto Android não será necessário criar e hospedar a aplicação servidora no GAE. Ao invés disso serão feitas requisições REST diretamente ao FCM para que ele envie as mensagens ao dispositivo através do token de registro obtido no método `onNewToken`.

A ideia é fazer com que esse método receba essa mensagem e gere uma notificação ao usuário, que ao ser clicada, exiba os detalhes desse produto na tela. Por isso, complete a função `onMessageReceived` com o tratamento para extrair a chave `product` do atributo `data` da mensagem recebida, como no trecho a seguir:

```
override fun onMessageReceived(remoteMessage: RemoteMessage) {
    remoteMessage.data.isNotEmpty().let {
        Log.d(TAG, "Payload: " + remoteMessage.data)
        if (remoteMessage.data.containsKey("product")) {
            sendProductNotification(remoteMessage.data.get("product")!!)
        }
    }
}
```

A mensagem a ser recebida no *payload* será um um produto em formato JSON, como no trecho a seguir:

---

<sup>30</sup><https://github.com/siecola/GAEBookV3Exemplo1>

<sup>31</sup><https://www.casadocodigo.com.br/products/livro-gae>

```
{
    "productID": "3",
    "name": "Nome 3",
    "model": "Model 3",
    "code": 3,
    "price": 30
}
```

Esse conteúdo será interpretado e transformado em um objeto Kotlin no fragmento que for responsável por exibir a informação. Por isso ele não será transformado dentro dessa classe.

A função `sendProductNotification` ainda não existe nessa classe, por isso comece criando-a como no trecho a seguir:

```
private fun sendProductNotification(productInfo: String) {
    val intent = Intent(this, MainActivity::class.java)
    intent.putExtra("product", productInfo)
    sendNotification(intent)
}
```

Perceba que ela recebe a string com a informação do produto e começa a preparar uma Intent, que nada mais é do que uma declaração de uma intenção de fazer algo, que nesse caso é chamar a Activity `MainActivity`, passando a informação recebida.

Essa função por sua vez invoca uma outra chamada `sendNotification`, passando a Intent criada. Ela deve ser responsável por criar a notificação a ser exibida ao usuário, fornecendo a Intent criada na função anterior, que é a de chamar `MainActivity`. Dessa forma, quando o usuário clicar na notificação, ele será redirecionado para a `MainActivity`.

Veja como essa função deve ficar:

```
private fun sendNotification(intent: Intent) {
    val pendingIntent = PendingIntent.getActivity(
        this, 0, intent,
        PendingIntent.FLAG_UPDATE_CURRENT
    )

    val channelId = "1"
    val notificationBuilder = NotificationCompat.Builder(this, channelId)
        .setSmallIcon(R.drawable.ic_cloud_queue_black_24dp)
        .setContentTitle("Sales Message")
        .setAutoCancel(true)
        .setContentIntent(pendingIntent)
```

```

val notificationManager =
    getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    val channel = NotificationChannel(
        channelId,
        "Sales provider",
        NotificationManager.IMPORTANCE_DEFAULT
    )
    notificationManager.createNotificationChannel(channel)
}

notificationManager.notify(0, notificationBuilder.build())
}

```

O processo de criação da notificação não é o escopo desse capítulo, mas em linha gerais o trecho anterior define algumas informações como título, ícone e a Intent criada na função `sendProductNotification`. Além disso configura que a notificação deve ser fechada assim que o usuário interage com ela, através da instrução `.setAutoCancel(true)`.

É importante ressaltar que o ícone é definido pelo recurso `ic_cloud_queue_black_24dp`, que deve ser adicionado na pasta `drawable` do projeto. Isso pode ser feito clicando-se com o botão direito sobre essa pasta, escolhendo a opção `New -> Vector Asset` e selecionando a figura dentro da opção `Clipart`, na tela que abrir.



Como dito anteriormente, quando usuário clicar na notificação ele será redirecionado para dentro da aplicação. Esse tratamento será feito na `MainActivity` em algumas seções à frente.

O último passo para fazer com que a classe `FCMService` funcione adequadamente é informar ao sistema Android que ela é a responsável por tratar os eventos do FCM. Isso deve ser feito no arquivo `AndroidManifest.xml`. Para isso, abra-o e acrescente o seguinte trecho dentro da tag `application`:

```

<service
    android:name="br.com.siecola.androidproject03.messaging.FCMService"
    android:exported="false">
    <intent-filter>
        <action android:name="com.google.firebaseio.MESSAGING_EVENT" />
    </intent-filter>
</service>

```

Perceba que o primeiro parâmetro dessa configuração é o caminho da classe `FCMService`. Da mesma forma, o parâmetro `intent-filter` está configurado para associar essa classe ao evento de recebimento de uma mensagem do FCM.

Para finalizar a configuração nesse arquivo, acrescente também dentro da tag `application` a configuração do canal padrão para geração da notificação dentro da aplicação:

```
<meta-data  
    android:name="com.google.firebaseio.messaging.default_notification_channel_id"  
    android:value="1" />
```

Essa é uma configuração que faz parte do mecanismo de notificação do Android, que possibilita a criação de canais onde o usuário pode configurar suas preferências, como ser notificado ou não.

Com essas configurações, já é possível com que a aplicação se registre no FCM e receba mensagens através dele.

As próximas seções continuam a criação da estrutura para lidar com a mensagem dentro da aplicação, bem como sua apresentação ao usuário.

## 10.6 - Criando o ViewModel

Como dito na seção anterior, a mensagem a ser recebida pela classe `FCMService` é um *payload* em formato JSON, como no trecho a seguir:

```
{  
    "productID": "3",  
    "name": "Nome 3",  
    "model": "Model 3",  
    "code": 3,  
    "price": 30  
}
```

A ideia de transportar uma informação em formato JSON pelo FCM é interessante, pois permite que o receptor possa interpretá-la através de seus campos, decidindo a forma de tratamento e exibição ao usuário.

Dessa forma, para fazer com que a aplicação saiba interpretar essa mensagem, é necessário criar um modelo para representá-la. Para isso, crie um novo pacote chamado `product` e dentro dele, uma classe de nome `Product`:

```
data class Product(  
    var productID: String,  
    var name: String,  
    var model: String,  
    var code: Int,  
    var price: Float  
)
```

Como já demonstrado algumas vezes, uma classe do tipo `data` em Kotlin possui tudo o que é necessário para representar um modelo de dados.

Essa classe `Product` será utilizada para transformar a mensagem recebida do FCM em um objeto que possa ser manipulado pela aplicação, porém isso só será feito no fragmento que for exibir tais informações.

De posse dessa classe, é possível agora criar o `ViewModel` que guardará a instância de `Product`, depois que a mensagem for interpretada, para ser exibida ao usuário. Por isso, crie a classe `ProductInfoViewModel` dentro do pacote `product`, como no trecho a seguir:

```
class ProductInfoViewModel : ViewModel() {  
  
    val fcmRegistrationId = MutableLiveData<String>()  
  
    val product = MutableLiveData<Product>()  
}
```

Esse `ViewModel` não terá nada de especial, pois, seu único objetivo é guardar as informações do token de registro no FCM e a mensagem interpretada do produto. Dessa forma, eles poderão ser exibidos pelo fragmento de tela através do *binding* realizado em seu arquivo XML.

## 10.7 - Criando o binding adapter

Da mesma forma como foi feito na aplicação anterior, é necessário criar um adaptador para exibir certas tipos de informações do produto, como código e preço. Esses adaptadores serão utilizados no fragmento que exibe as informações ao usuário. Para isso, crie um novo arquivo chamado `ProductBindingAdapters` no pacote `product`, como no trecho a seguir:

```
import android.widget.TextView
import androidx.databinding.BindingAdapter

@BindingAdapter("productPrice")
fun bindProductPrice(txtProductPrice: TextView, productPrice: Float?) {
    productPrice?.let {
        val price = "$ " + "%.2f".format(productPrice)
        txtProductPrice.text = price
    }
}

@BindingAdapter("productCode")
fun bindProductCode(txtProductCode: TextView, productCode: Int?) {
    productCode?.let {
        txtProductCode.text = "$productCode"
    }
}
```

Perceba que o princípio é o mesmo utilizado no projeto anterior.

## 10.8 - Recebendo as notificações na MainActivity

Como dito anteriormente nesse capítulo, a ideia é fazer com que a notificação seja gerada pela classe FCMService e emitida para a visualização do usuário. No momento da criação dessa notificação, a Intent foi configurada para invocar a MainActivity, como pode ser visto no trecho a seguir:

```
val intent = Intent(this, MainActivity::class.java)
```

Isso quer dizer que quando o usuário clicar na notificação, essa será a ação realizada pelo sistema Android: chamar a classe MainActivity da aplicação.

A MainActivity por sua vez será responsável por receber a notificação e chamar o fragmento de tela, que ainda será criado, para exibir a informação do produto contido dentro da mensagem. A ideia para implementar essa navegação é utilizar o NavigationController, como no projeto anterior.

Com isso, é necessário adaptar o layout da MainActivity definido no arquivo res\layout\activity\_main.xml para que fique da seguinte forma:

```
<?xml version="1.0" encoding="utf-8"?>

<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/navigation_graph" />
```

Repare que é o mesmo formato utilizado no projeto anterior. Veja também que o componente de identificação `navigation_graph` ainda não foi criado nesse projeto, por isso crie um novo pacote chamado `navigation`, na pasta `res` com um novo arquivo chamado `navigation_graph.xml`:

```
<?xml version="1.0" encoding="utf-8"?>

<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/navigation_graph"
    app:startDestination="@+id/fragmentProductInfo">

    <!--Add new actions here -->

    <fragment
        android:id="@+id/fragmentProductInfo"
        android:name="br.com.siecola.androidproject03.product.ProductInfoFragment"
        android:label="fragment_products_list"
        tools:layout="@layout/fragment_product_info" >
    </fragment>
</navigation>
```

Aqui o componente `navigation` aponta para a primeira tela que deverá ser exibida na aplicação, que ainda será definida no arquivo `fragment_product_info.xml`.

No local onde está o comentário no arquivo `navigation_graph.xml`, dentro da tag `navigation`, é necessário definir a ação que será utilizada por `MainActivity` para invocar o fragmento `ProductInfo`, passando como argumento a mensagem recebida. Para isso, substitua o comentário pelo seguinte trecho:

```

<action
    android:id="@+id/action_showProductInfo"
    app:destination="@+id/fragmentProductInfo" >
    <argument
        app:nullable="true"
        android:name="productInfo"
        app:argType="string"/>
</action>

```



Compile todo o projeto após a criação do arquivo `navigation_graph.xml`.

Essa ação invocará o fragmento para exibir o conteúdo da notificação que será passado como argumento.

A ação criada deverá ser invocada de dentro de `MainActivity`, quando o usuário clicar na notificação. Por isso, implemente a função `showProductInfo`, como no trecho a seguir:

```

private fun showProductInfo(productInfo: String) {
    this.findNavController(R.id.nav_host_fragment)
        .navigate(ProductInfoFragmentDirections.actionShowProductInfo(productInfo))
}

```

Essa função invoca a ação criada no arquivo `navigation_graph.xml`, passando o argumento `productInfo` para ser exibido pelo fragmento a ser criado na próxima seção. A invocação dessa função deverá acontecer em dois lugares diferentes de `MainActivity`:

- Na função `onCreate`, se nenhuma instância de `MainActivity` existir;
- Na função `onNewIntent`, caso não exista nenhuma instância de `MainActivity` em execução.

É necessário implementar os dois pontos para que a notificação seja sempre tratada por `MainActivity`, independente do estado em que ela tiver.

Começando com a função `onCreate`, logo após a instrução `setContentView(R.layout.activity_main)`, adicione o trecho a seguir:

```

if (this.intent.hasExtra("product")) {
    showProductInfo(intent.getStringExtra("product")!!)
}

```

Veja que a informação presente na notificação clicada pelo usuário virá através do atributo `intent` de `MainActivity`, que por sua vez carrega um atributo extra com a chave `product`, inserido de dentro da classe `FCMService`, como pode ser observado no trecho a seguir na função `sendProductNotification`:

```
private fun sendProductNotification(productInfo: String) {  
    val intent = Intent(this, MainActivity::class.java)  
    intent.putExtra("product", productInfo)  
    sendNotification(intent)  
}
```

Perceba que na instrução `intent.putExtra("product", productInfo)` a variável `intent` recebe um novo valor extra com a chave `product`, a mesma que agora está sendo extraída em `MainActivity`.

E esse cobre o primeiro caso onde `MainActivity` ainda não existe e será criada com o valor extra em seu atributo `intent`.

O segundo ponto que deve ser construído em `MainActivity` é na função `onNewIntent`, como no trecho a seguir:

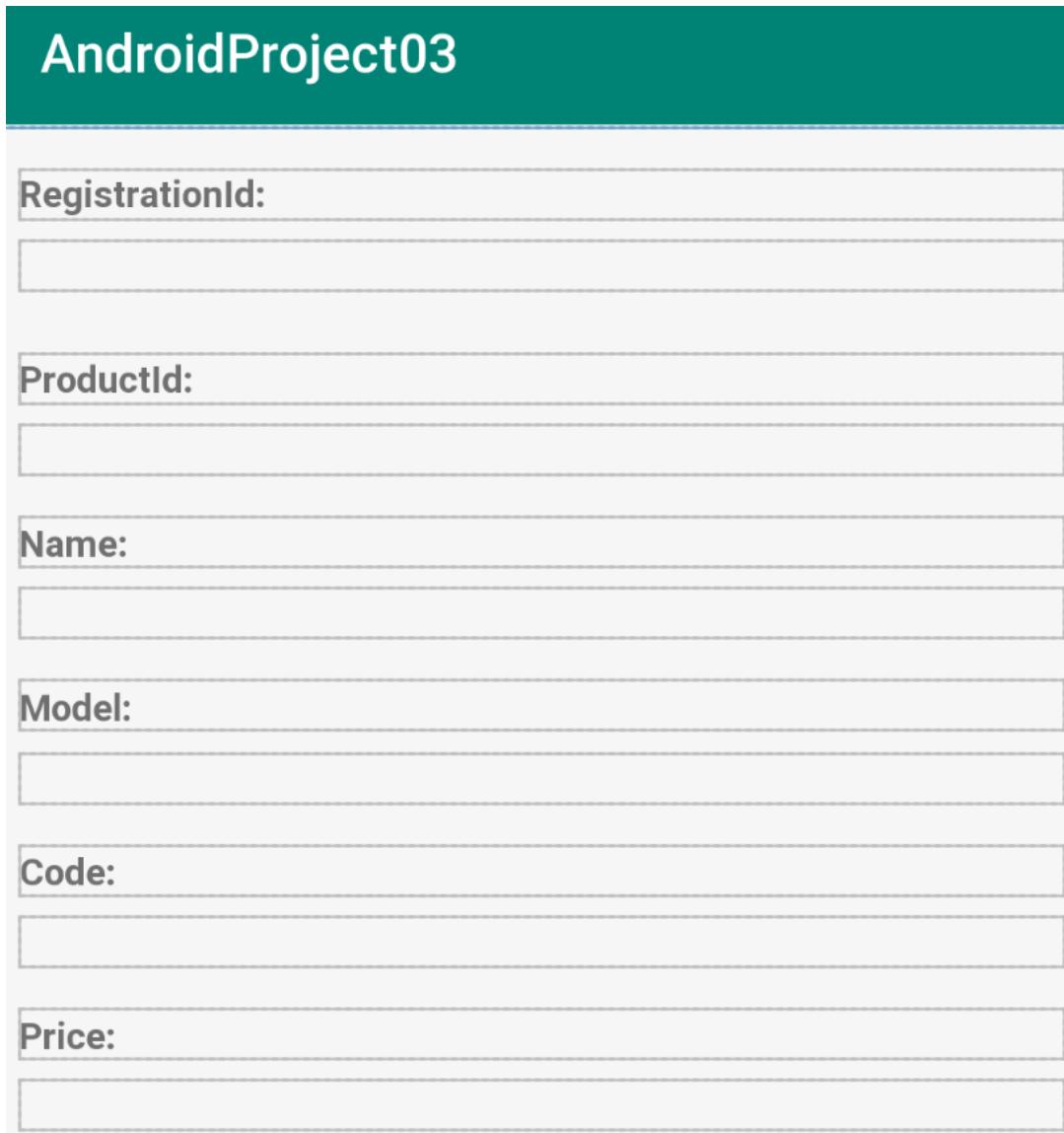
```
override fun onNewIntent(intent: Intent) {  
    if (intent.getStringExtra("product") != null) {  
        showProductInfo(intent.getStringExtra("product"))  
    }  
    super.onNewIntent(intent)  
}
```

Esse é o caso onde `MainActivity` já existe, portanto ela apenas será atualizada com a informação extra através do atributo `intent` passado pela função.

E isso finaliza a implementação na classe `MainActivity`.

## 10.9 - Criando o fragmento para exibir as mensagens

A proposta para a tela que vai exibir as informações do produto e do registro do aplicativo no Firebase Cloud Messaging é a seguinte:



Modelo da tela a ser construída

O primeiro campo é destinado a exibir o token obtido pelo aplicativo ao se registrar no FCM. É interessante que esse campo possa ter seu conteúdo copiado para facilitar seu uso, como será mostrado na próxima seção desse capítulo.

Os demais campos são para exibir os detalhes do produto recebido na mensagem de notificação vinda pelo FCM. Perceba que tecnicamente não há nada de novo nessa tela. Por isso, comece criando um novo arquivo na pasta `res/layout` chamado `fragment_product_info.xml`, como no trecho a seguir:

```
<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

        <variable
            name="productInfoViewModel"
            type="br.com.siecola.androidproject03.product.ProductInfoViewModel" />
    </data>

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <androidx.constraintlayout.widget.ConstraintLayout
            android:id="@+id/layout"
            android:layout_width="match_parent"
            android:layout_height="wrap_content">

            <!-- New components here -->

        </androidx.constraintlayout.widget.ConstraintLayout>
    </ScrollView>
</layout>
```

Sua estrutura é basicamente a mesma vista no projeto anterior, com a tag `data` para conter a variável que aponta para o `ViewModel` criado e os componentes `ScrollView` e `ConstraintLayout` para acomodar os demais componentes.

Para continuar a construir a tela, no lugar onde está o comentário dentro da tag `ConstraintLayout`, crie o par de componentes para o campo `RegistrationId` :

```
<TextView
    android:id="@+id/textView2"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="8dp"
    android:text="RegistrationId:"
    android:textStyle="bold"
```

```
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/textView3"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:text="@{productInfoViewModel.fcmRegistrationId}"
    android:textIsSelectable="true"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView2" />
```

Veja que a instrução `android:textIsSelectable="true"` do componente de identificação `textView3` permite que seu conteúdo possa ser selecionável. Isso será útil para copiar seu valor, que é o token de registro no FCM.

Continuando, crie agora o par de componentes para o campo `ProductId`:

```
<TextView
    android:id="@+id/textView4"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="24dp"
    android:layout_marginEnd="8dp"
    android:text="ProductId:"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView3" />

<TextView
    android:id="@+id/textView5"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
```

```
    android:text="@{productInfoViewModel.product.productID}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView4" />
```

Perceba que ProductInfoViewModel está sendo utilizado em cada campo para exibir as informações que estão armazenadas nele.

Agora para o campo Name, crie o seguinte par de componentes:

```
<TextView
    android:id="@+id/textView6"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="8dp"
    android:text="Name:"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView5" />
```

```
<TextView
    android:id="@+id/textView7"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:text="@{productInfoViewModel.product.name}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView6" />
```

E para o campo Model:

```
<TextView
    android:id="@+id/textView8"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="8dp"
    android:text="Model:"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView7" />

<TextView
    android:id="@+id/textView9"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:text="@{productInfoViewModel.product.model}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView8" />
```

Agora para o campo Code:

```
<TextView
    android:id="@+id/textView10"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="8dp"
    android:text="Code:"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView9" />

<TextView
    android:id="@+id/textView11"
    android:layout_width="0dp"
```

```
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    app:productCode="@{productInfoViewModel.product.code}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView10" />
```

Perceba que a instrução para associar o valor do código do produto utiliza o *binder* que foi construído no arquivo ProductBindingAdapters.kt:

```
app:productCode="@{productInfoViewModel.product.code}"
```

Essa associação deve ser feita dessa maneira para converter o código do produto em uma String.

Isso também deve ser feito para o último par de componentes para o campo Price:

```
<TextView
    android:id="@+id/textView12"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="8dp"
    android:text="Price:"
    android:textStyle="bold"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView11" />
```

```
<TextView
    android:id="@+id/textView13"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    app:productPrice="@{productInfoViewModel.product.price}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView12" />
```

E isso finaliza a construção da tela para exibição das informações do token de registro obtido do FCM e o produto vindo da notificação através dele.

Para exibir essa tela é necessário construir um fragmento, que fará a junção de tudo o que foi feito até o momento nesse projeto. Para isso, crie, no pacote `product` uma nova classe chamada `ProductInfoFragment`, como no trecho a seguir:

```
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.lifecycle.ViewModelProviders
import br.com.siecola.androidproject03.databinding.FragmentProductInfoBinding
import com.google.firebase.iid.FirebaseInstanceId
import com.squareup.moshi.JsonAdapter
import com.squareup.moshi.Moshi

class ProductInfoFragment : Fragment() {

    private val productInfoViewModel: ProductInfoViewModel by lazy {
        ViewModelProviders.of(this).get(ProductInfoViewModel::class.java)
    }

}
```

Veja que o atributo privado do tipo `ProductInfoViewModel` foi criado para receber a referência desse `ViewModel`, que já foi construído para guardar as informações do token de registro no FCM e o produto a ser exibido.

Para continuar essa classe, crie a função `onCreateView`, como no trecho a seguir:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val binding = FragmentProductInfoBinding.inflate(inflater)

    binding.setLifecycleOwner(this)

    binding.productInfoViewModel = productInfoViewModel

    //Add the rest of the code here
```

```

    return binding.root
}

```

O trecho de código acima é semelhante ao que já foi criado no projeto anterior, que apenas faz as associações necessárias entre o fragmento de tela e o `ViewModel`.

No local onde está o comentário, é necessário implementar a lógica para exibir o token de registro no FCM, como pode ser visto no trecho a seguir:

```

FirebaseInstanceId.getInstance().instanceId
    .addOnCompleteListener { task ->
        if (task.isSuccessful) {
            productInfoViewModel.fcmRegistrationId.value = task.result?.token
            Log.i("ProductInfoFragment", "FCM Token: ${task.result?.token}")
        }
    }
}

```

Esse trecho cria um *listener* para receber o token de registro do FCM, quando essa operação for concluída. Dessa forma seu valor pode ser enviado à tela através de `ProductInfoViewModel` e de seu atributo `fcmRegistrationId`.

Para concluir a implementação dessa classe, acrescente logo abaixo do trecho anterior o código responsável por receber a informação do produto através do argumento enviado ao fragmento:

```

if (this.arguments != null) {
    if (this.arguments!!.containsKey("productInfo")) {
        val moshi = Moshi.Builder().build()
        val jsonAdapter: JsonAdapter<Product> =
            moshi.adapter<Product>(Product::class.java)

        jsonAdapter.fromJson(this.arguments!!.getString("productInfo")!!).let {
            productInfoViewModel.product.value = it
        }
    }
}

```

Esse argumento foi enviado pela classe `MainActivity` e está sendo retirado aqui como `String`, por isso ele deve ser convertido para o modelo `Product` presente no projeto. Esse trabalho está sendo feito pela biblioteca `Moshi`, que facilita essa conversão.

Depois que a `String` contendo o payload em formato JSON é convertido para uma entidade do tipo `Product`, ela pode ser atribuída ao `ProductInfoViewModel` para ser exibida na tela através do *binder* criado para isso no arquivo `fragment_product_info.xml`.

E isso conclui a implementação desse projeto. A seção seguinte contém instruções de como testá-la, enviando mensagens através da API de envio de mensagens do Firebase Cloud Messaging.

## 10.10 - Testando o recebimento de mensagens

O teste da implementação do projeto criado nesse capítulo se divide em duas ações:

- **Registro no FCM:** quando o aplicativo se registrar no FCM, o token de registro deverá aparecer no primeiro campo da tela, chamado `RegistrationId`;
- **Recebimento de mensagens enviadas pela API do FCM:** através da API do FCM, será possível enviar uma mensagem ao dispositivo executando o aplicativo, utilizando o token de registro citado anteriormente.

Para começar os testes, execute a aplicação em um dispositivo real ou em um emulador que tenha o Google Play instalado. Após alguns segundos o campo `RegistrationId` deve mostrar o valor do token de registro no FCM. Caso a aplicação não compile, verifique se o arquivo `google-services.json` está no local correto como indicado nos passos descritos na seção 10.4 desse capítulo.

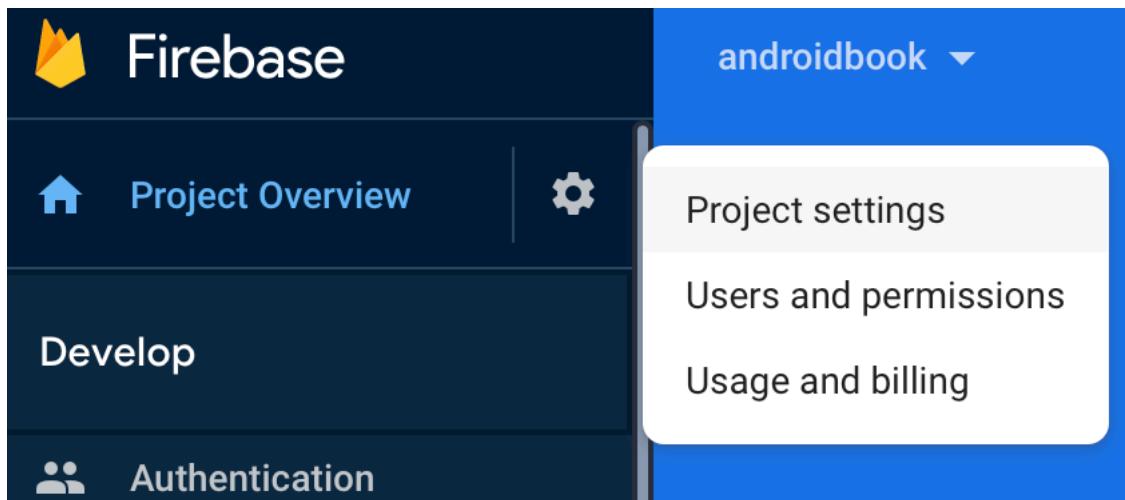
Caso o token de registro não apareça na tela, verifique os logs da aplicação no Logcat do Android Studio, bem como se o dispositivo está com conexão à Internet. Uma outra alternativa pode ser fechar e abrir o aplicativo novamente.



É possível copiar o conteúdo desse campo clicando e segurando, até que o Android permita que ele seja selecionado.

O valor do token é necessário para utilizar a API do FCM para enviar mensagens ao dispositivo. Esse token o identifica unicamente na plataforma, dessa forma ele pode ser alcançado pelo FCM.

Porém, ainda é necessário obter a chave de acesso do projeto criado no Firebase. Para isso, vá até o console do Firebase e acesse a opção `Project Settings`, como mostra a figura a seguir:



Configurações do projeto

Na tela que aparecer, clique na aba Cloud Messaging. Nessa tela, há uma sessão chamada Project credentials, com um campo chamado Server key. O valor desse campo deve ser utilizado nas instruções a seguir.

De posse do **token de registro** e **Server key**, é possível enviar uma mensagem ao dispositivo utilizando o Postman, como as seguintes configurações:

- **URL de acesso:** <https://fcm.googleapis.com/fcm/send>
- **Método HTTP:** POST
- **Cabeçalhos HTTP:**
  - Content-Type: application/json
  - Authorization: valor do campo **Server key** obtido no console do Firebase, como explicado anteriormente. O conteúdo desse cabeçalho deve ser no seguinte formato:  
key=<Server key>
- **Payload:** informação contendo o token de registro do FCM e a informação a ser enviada, como no exemplo a seguir.

```
{
  "to" : "token de registro no FCM",
  "data" : {
    "product": "{\"productID\": \"3\", \"name\": \"Nome 3\", \"model\": \"Model 3\", \"code\": 3, \"price\": 30.0}"
  }
}
```

O valor do token de registro do FCM deve ser inserido no campo **to** do JSON anterior.

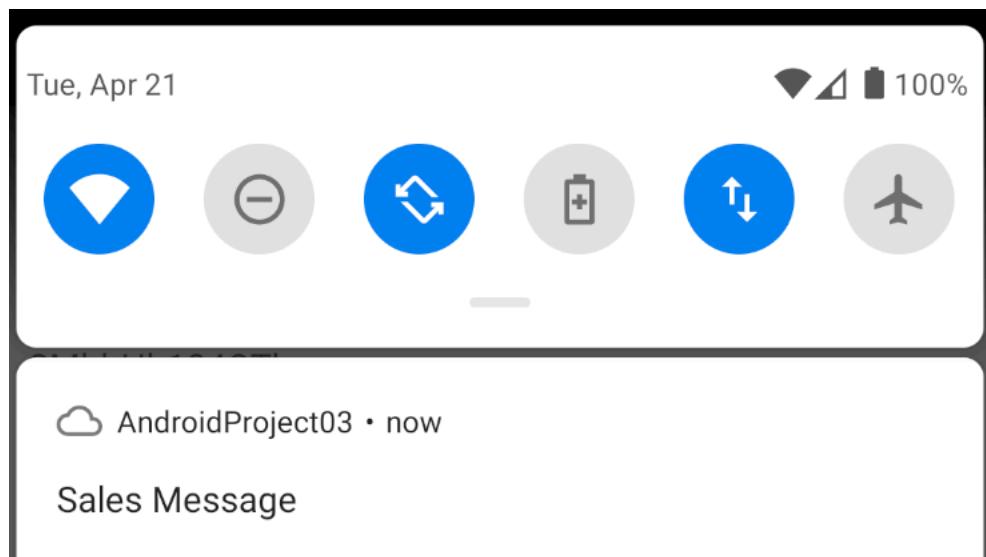
Depois de configurar o Postman com as informações apresentadas aqui, clique no botão **Send**. Caso a mensagem tenha sido aceita pelo Firebase, a resposta deverá ser algo no seguinte formato:

```
{  
    "multicast_id": 8661942880170194994,  
    "success": 1,  
    "failure": 0,  
    "canonical_ids": 0,  
    "results": [  
        {  
            "message_id": "0:1587484575523841%26135ff6f9fd7ecd"  
        }  
    ]  
}
```

O campo `success` indica que a mensagem foi aceita para ser entregue pelo FCM e não que ele realmente foi entregue.

O campo `message_id` é um identificador único da mensagem.

O dispositivo deve receber a mensagem recebida, no painel de notificações do sistema Android, como mostra a figura a seguir:



Notificação recebida pelo FCM

Quando o usuário clicar na notificação com o nome `Sales Message`, ele será redirecionado para dentro da aplicação, que exibirá o conteúdo do produto que foi enviado dentro da notificação, como pode ser observado na figura a seguir:

<b>ProductId:</b>
3
<b>Name:</b>
Nome 3
<b>Model:</b>
Model 3
<b>Code:</b>
3
<b>Price:</b>
\$ 30.00

Produto recebido pelo FCM

E isso finaliza o teste de recebimento de notificações pelo FCM em uma aplicação Android!



Se tiver interesse em aprender a criar uma aplicação de backend para **envio de notificações pelo FCM**, consulte o seguinte [livro](#)<sup>32</sup>.

## 10.11 - Conclusão

Esse capítulo introduziu um dos mais famosos serviços do Firebase, o Cloud Messaging. Com ele é possível desenvolver facilmente aplicações para o recebimento de notificações através dessa plataforma, algo muito interessante para notificar usuários sobre algum evento de seu interesse ou mesmo promover o maior engajamento deles.

O próximo capítulo apresentará dois novos serviços do Firebase:

- **Authentication:** utilizado para permitir que os usuários se autentiquem na aplicação, usando uma conta do Google, por exemplo.
- **Firestore:** um banco de dados NoSQL *realtime*, capaz de realizar sincronismos dos dados entre diversas aplicações diferentes.

<sup>32</sup><https://www.casadocodigo.com.br/products/livro-gae>

O novo projeto que será criado também contará com outros serviços do Firebase, que serão introduzidos nos capítulos seguintes.

# 11 - Autenticando usuários com Firebase Authentication e persistindo dados com o Firestore

Construir um aplicativo Android para persistir dados gerados pelo usuário já apresenta seus desafios, mas fazer com que tais dados estejam disponíveis em todos os dispositivos do usuário, incluindo uma aplicação Web, pode ser bastante complexo.

Caso a ideia também seja compartilhar alguns dados com outros usuários, como uma lista de compras entre membros de uma família, por exemplo, a dificuldade de implementação pode ser ainda maior.

Porém, essa dificuldade de implementação não se restringe somente ao projeto Android, mas também à aplicação de back-end que deve suportar o sincronismo e compartilhamento de dados entre todo os dispositivos móveis envolvidos ou interessados nesses dados.

Em se tratando segurança, esses dados também devem ser protegidos de acessos não autorizados e somente ser liberados para os usuários que são donos dele, ou àqueles que compartilham do acesso a ele. Isso significa que os usuários da aplicação Android devem se autenticar e também deve existir uma aplicação de back-end que faça a validação das credenciais do usuário.

Criar toda essa implementação é tecnicamente possível, porém o Firebase oferece dois serviços que são ideais para esse tipo de situação, deixando os desenvolvedores focados na implementação do negócio e obviamente agilizando entregas de valores do produto.

As duas seções seguintes introduzem os dois serviços que serão trabalhados nesse capítulo: **Firebase Authentication** e **Firestore**.

A ideia desse capítulo é utilizar esses dois serviços do Firebase para construir uma aplicação Android onde os usuários podem se autenticar e cadastrar produtos, como uma lista de compras. Esses produtos ficarão armazenados no Firebase e poderão ser acessados de qualquer dispositivo em que o usuário se autenticar. Da mesma forma, quando algum produto for alterado em um dos dispositivos do usuário, esse mesmo dado será atualizado nos seus demais dispositivos.

Para construção dessa proposta de aplicativo, será utilizado muito do que já foi visto aqui nesse livro, como *data binding*, *ViewModel*, *data adapters* e navegação com *safe args*.

## 11.1 - O que o Firebase Authentication

Muitas aplicações precisam saber a identidade do usuário para promover regras de seguranças eficientes e oferecer experiências personalizadas. Com Firebase Authentication é possível autenticar

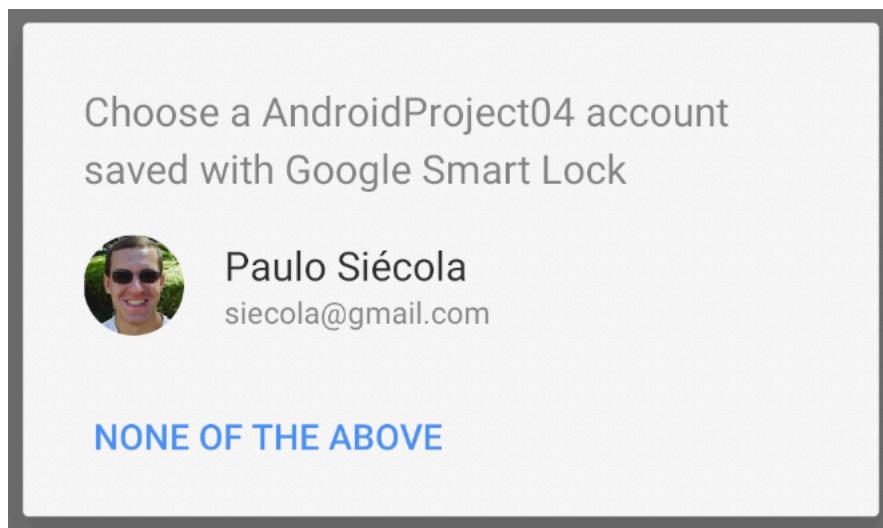
o usuário com vários provedores, como Facebook, Twitter, GitHub e Google, tudo isso com uma interface de login própria ou utilizando uma já pronta do Google, que pode ser customizada.

Quando o usuário é autenticado, é possível obter informações sobre ele, sendo possível criar experiências personalizadas de acordo com o perfil do usuário. Além disso, o Firebase gerencia e mantém uma sessão do usuário autenticado, assim é possível fazer com o aplicativo seja reiniciado sem a necessidade de solicitar o login novamente.

O Firebase Authentication funciona com *call backs*, o que significa que a aplicação é informada quando um login foi realizado com sucesso ou não, da mesma forma em que é notificada quando um usuário realiza logout. Isso permite que a aplicação carregue as informações do usuário que se autenticou no momento adequado, assim como pode excluir tudo quando o usuário faz um logout.

A informação de qual usuário está autenticado pode ser utilizada no planejamento das entidades que são persistidas no Firestore, fazendo com que as regras de segurança sejam construídas em cima dessa informação. Isso significa que somente o usuário que é dono de uma informação no Firestore é quem poderá acessá-la.

A seguir, o exemplo de uma tela de login da aplicação Android que será construída nesse capítulo, utilizando a interface padrão do FirebaseUI.



Tela de login do FirebaseUI

Essa interface também pode ser customizada em sua aparência e comportamento, mas sua utilização na forma padrão já traz uma experiência agradável ao usuário, além de ser bem direta de se integrar no código do projeto.

## 11.2 - O que é Firestore

O Firestore é um banco de dados não-relacional, que permite que os dados sejam armazenados na nuvem, com atualização *realtime*, o que significa que quando um dado é alterado, os clientes

observadores desse dado recebem a atualização sobre sua mudança. Isso permite que os dados possam ser acessados de outros dispositivos do usuário ou mesmo sejam compartilhados entre outros usuários.

Os dados também podem ser salvos no dispositivo mesmo se ele estiver *offline*, pois eles serão salvos no Firebase quando o dispositivo conseguir se conectar à Internet, sem nenhuma implementação adicional a ser feita pelo desenvolvedor, como mecanismos de sincronização.

Com o Firestore, não é necessário se preocupar com a escalabilidade do banco de dados do aplicativo, não importa quantos usuários ou acessos simultâneos ele tiver, pois ele é servido com a mesma infraestrutura dos demais serviços do Google.

Com a integração com o Firebase Authentication, é possível criar regras de segurança, fazendo com que os dados somente sejam acessados pelos usuários que possuem as permissões corretas pra tal.

A estruturação dos dados no Firestore é através de documentos que possuem chaves e valores. Também é possível utilizar coleções de documentos dentro de um documento.

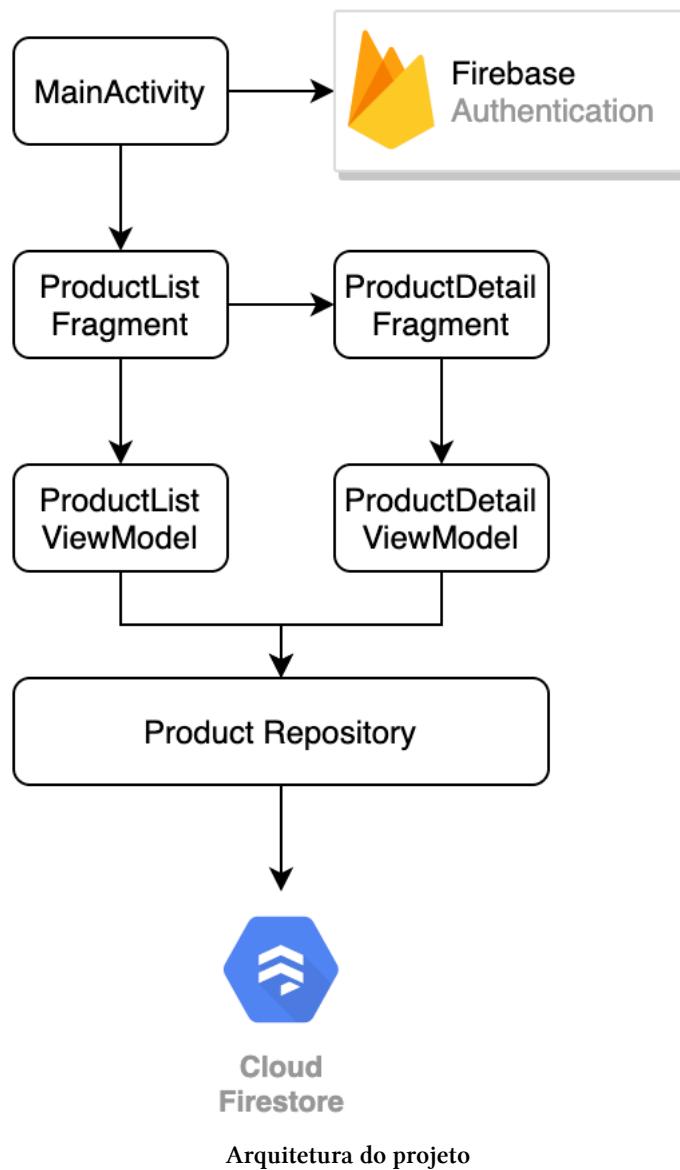
Para o desenvolvedor, o Firestore possui uma interface Web onde é possível observar e editar os dados de forma simples e direta.

## 11.3 - Arquitetura do projeto

A arquitetura do projeto a ser construído nesse capítulo é semelhante à utilizada em *AndroidProject02*, com uma lista de produtos e uma tela para exibir seus detalhes. Por isso, muito do que foi feito naquele projeto será reutilizado aqui, para que a explicação foque no que há de novo, como o Firebase Authentication e o Firestore.

A grande diferença aqui é que o usuário terá que se autenticar com sua conta do Google e os produtos poderão ser salvos e editados utilizando o Firestore como banco de dados.

Veja na figura a seguir, a arquitetura proposta para esse projeto:



A implementação começará pela autenticação do usuário utilizando o Firebase Authentication, depois passará para a construção da base do projeto, como fragmentos, layouts, ViewModels e navegação, aproveitando ao máximo do que foi feito em `AndroidProject02`.

Depois será feita a construção do repositório de dados responsável por gerenciar o acesso ao Firestore, onde os produtos serão armazenados.

Esse mesmo projeto que será criado no Android Studio será utilizado para os demais capítulos a frente nesse livro, para a demonstração da utilização de outros recursos do Firebase.

## 11.4 - Criando o novo projeto no Android Studio

No Android Studio, crie o novo projeto, seguindo o que foi feito nos três últimos projetos desse livro, com as seguintes informações:

- **Tipo:** Empty Activity
- **Nome do projeto:** AndroidProject04
- **Pacote base:** br.com.siecola
- **Linguagem:** Kotlin
- **Versão mínima do SDK:** 21

No arquivo `build.gradle` do projeto, adicione as seguintes dependências:

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.2.1"
classpath 'com.google.gms:google-services:4.3.3'
```

O primeiro já é conhecido, que trata da funcionalidade de se utilizar argumentos seguros na chamada de fragmentos. O segundo é para permitir que a aplicação acesse serviços do Google, como o Firebase.

Agora no arquivo `build.gradle` do módulo `app`, tenha certeza de aplicar todos os seguinte plugins:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'
apply plugin: "androidx.navigation.safeargs"
apply plugin: 'com.google.gms.google-services'
```

Nesse mesmo arquivo, na seção `android`, adicione as seguintes configurações, que já foram utilizadas no projeto anterior:

```
dataBinding {
    enabled = true
}
androidExtensions {
    experimental = true
}
```

Para finalizar, ainda nesse arquivo, configure a seção `dependencies` para ficar com o trecho a seguir:

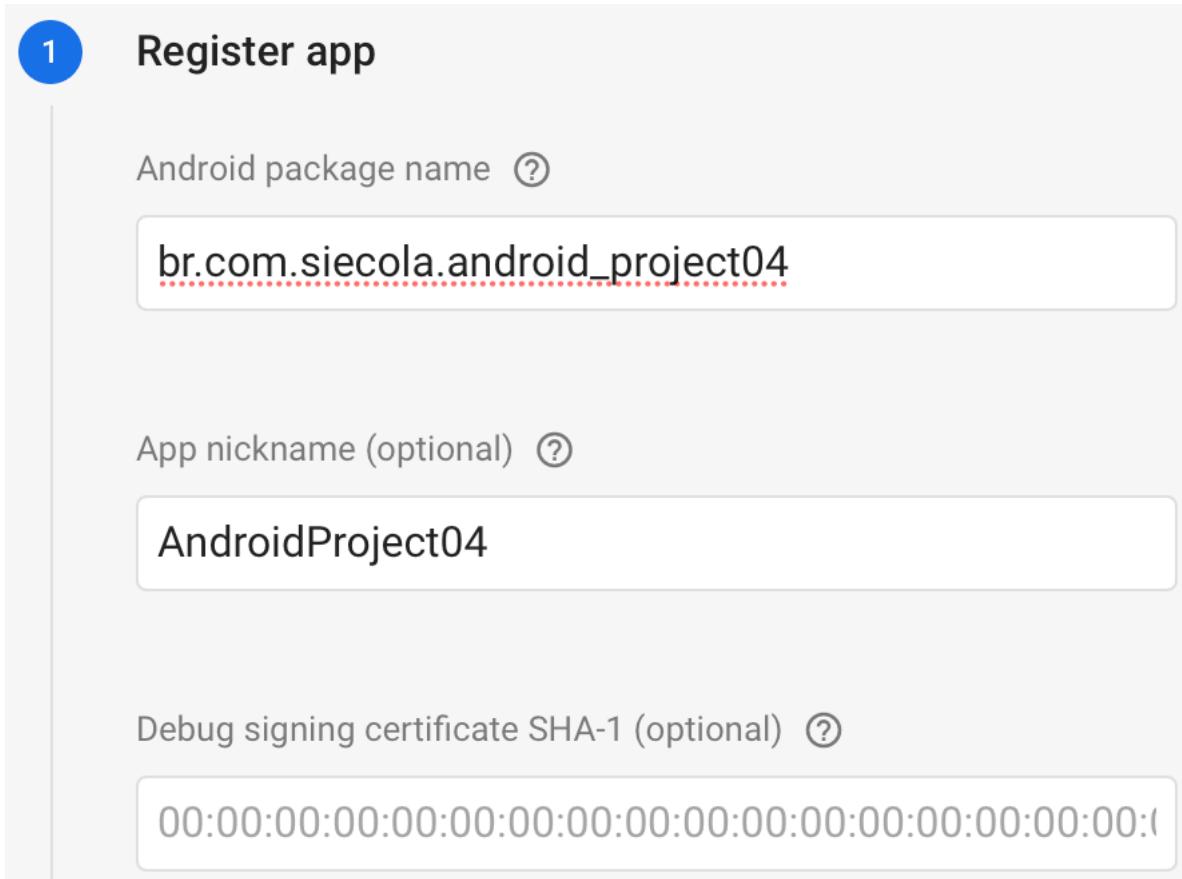
```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.1.0'  
    implementation 'androidx.core:core-ktx:1.2.0'  
  
    //Constraint Layout  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
  
    // ViewModel and LiveData  
    implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"  
  
    //Recyclerview  
    implementation "androidx.recyclerview:recyclerview:1.1.0"  
  
    //Navigation  
    implementation 'androidx.navigation:navigation-fragment-ktx:2.2.1'  
    implementation 'androidx.navigation:navigation-ui-ktx:2.2.1'  
  
    // ViewModel and LiveData  
    implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"  
  
    //Firebase  
    implementation 'com.google.firebaseio:firebase-core:17.3.0'  
    implementation 'com.google.firebaseio:firebase-auth:19.3.0'  
    implementation 'com.firebaseioui:firebase-ui-auth:6.2.0'  
        implementation 'com.google.firebaseio:firebase-firebase-ktx:21.4.3'  
    implementation 'com.google.firebaseio:firebase-analytics:17.2.2'  
  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'  
}
```

As novas bibliotecas que foram adicionadas aqui são do Firebase, que serão utilizadas para incluir o mecanismo de autenticação utilizando a interface gráfica padrão disponibilizada por ele.

## 11.5 - Criando o novo projeto no Firebase

O novo projeto do Firebase é semelhante ao anterior, porém com algumas opções a mais. Para começar, vá ao seu console e comece o processo de criação de um novo projeto.

Na primeira página de criação, habilite a opção do Google Analytics para o novo projeto. Configure com a sua conta padrão do Firebase. Depois que o projeto for criado, adicione o aplicativo Android a ele, da mesma forma que foi feito no capítulo anterior, porém com um detalhe a mais:



## Adicionando app Android ao projeto

Preencha o campo Debug signing certificate SHA-1. Esse passo é necessário para utilizar o Firebase Authentication em modo de depuração. Siga as instruções desse [link<sup>33</sup>](#) para gerar a informação solicitada, de acordo com o sistema operacional da máquina de desenvolvimento utilizada.

Durante o processo para adicionar o projeto Android, baixe o arquivo `google-services.json` e adicione ao projeto, dentro da pasta `app`, como foi feito no capítulo anterior.

Depois que adicionar o arquivo `google-services.json` ao projeto Android, compile-o e execute-o, para ter certeza de que tudo está funcionando, até o momento.

<sup>33</sup><https://developers.google.com/android/guides/client-auth>

## 11.6 - Autenticando o usuário com Firebase Authentication

Como dito anteriormente nesse capítulo, os usuários dessa nova aplicação Android deverão se autenticar para poderem usá-la. Esse tipo de mecanismo pode ser utilizado para vários propósitos, como:

- Definir regras de validação de acesso a dados restritos ou somente de propriedade do usuário;
- Fornecer experiências customizadas a cada usuário, de acordo com suas características;
- Permitir segmentação de usuários baseados em suas características e localidade.

A ideia de se utilizar o Firebase Authentication nesse projeto é definir regras de acesso aos dados que serão persistidos no Firestore. Dessa forma, cada usuário poderá ter um acesso seguro aos dados que foram criados por ele.

Para começar, é necessário habilitar essa funcionalidade no Firebase, por isso, dentro do console do projeto que acabou de ser criado, vá até o menu Authentication, localizado na lista de serviços do Firebase no canto esquerdo e depois na aba Sign-in method. Em seguida habilite a opção para deixar o usuário se logar com a conta do Google e acrescente um e-mail de suporte.

Veja que nessa área existem muitos outros provedores que podem ser utilizados para permitir que o usuário se autentique na aplicação.

Com apenas essa configuração no Firebase, já é possível implementar a parte da aplicação Android que irá autenticar os usuários. A ideia é ser simples para poder demonstrar o conceito de como o Authentication pode ser integrado ao projeto, por isso as seguintes características serão criadas no projeto Android:

- Autenticar o usuário com a sua conta do Google;
- Utilizar a interface gráfica padrão FirebaseUI, sem customizações;
- Criar um botão para que o usuário faça logout;
- Quando o usuário fizer login, as informações da aplicação deverão ser exibidas (por enquanto somente um texto de boas-vindas);
- Quando o usuário fizer logout, a tela de login deverá ser exibida novamente.

Para começar essa implementação, abra o projeto `AndroidProject04` no Android Studio e vá até a classe `MainActivity`, na função `onCreate`, que deve estar como no trecho a seguir:

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

A ideia aqui é inicializar o cliente do Firebase e verificar se já um usuário autenticado na aplicação, lembrando que o Firebase Authentication mantém a sessão ativa, caso usuário já tenha se logado antes.

Caso exista um usuário logado, a tela de boas-vindas é exibida, que por enquanto apenas carrega um texto simples, gerado pelo *template* do Android Studio.

Caso não exista informação de um usuário logado na aplicação, essa função deverá invocar a tela de login para o provedor Google. Veja como essa função deverá ficar com essa implementação:

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

    FirebaseApp.initializeApp(this)
    val user = FirebaseAuth.getInstance().currentUser

    if (user != null) {
        val name = user.displayName
        val email = user.email
        setContentView(R.layout.activity_main)
    } else {
        val providers = arrayListOf(AuthUI.IdpConfig.GoogleBuilder().build())

        startActivityForResult(
            AuthUI.getInstance()
                .createSignInIntentBuilder()
                .setAvailableProviders(providers)
                .build(), 1)
    }
}
```

Veja que a instrução `FirebaseApp.initializeApp(this)` é responsável por inicializar o cliente do Firebase na aplicação.

Depois de inicializado, é possível verificar se existe um usuário logado na aplicação através da instrução `val user = FirebaseAuth.getInstance().currentUser`.

Se a variável `user` for diferente de `null`, significa que existe um usuário autenticado na aplicação, e com isso é possível ler certas informações sobre ele, como nome e e-mail. Nesse instante, a tela de

boas-vindas pode ser exibida, aqui realizado pela instrução `setContentView(R.layout.activity_main)`.

Caso não existe nenhum usuário logado, o restante do código anterior cuida de solicitar ao sistema Android que exiba a tela de login, para que o usuário se autentique com sua conta do Google. Nesse momento uma nova Activity é lançada e seu resultado poderá ser acompanhado através da seguinte função:

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

    if (requestCode == 1) {
        val response = IdpResponse.fromResultIntent(data)

        if (resultCode == Activity.RESULT_OK) {
            val user = FirebaseAuth.getInstance().currentUser
            setContentView(R.layout.activity_main)

        } else {
            Toast.makeText(this, "Sign in failed", Toast.LENGTH_SHORT).show()
        }
    }
}
```

Veja que o primeiro teste dentro dessa função verifica se o código da requisição é igual ao mesmo valor solicitado na instrução `startActivityForResult` dentro da função `onCreate`. Isso deve ser feito pois essa função é genérica para indicar resultados retornados por outras ações dentro da Activity.

Ainda dentro dessa função, é verificado se o usuário realmente conseguiu se autenticar. Caso positivo, novamente a tela de boas-vindas é exibida.

Nesse momento, tendo um usuário logado na aplicação, é necessário oferecer um mecanismo para que ele faça um logout. Para isso, será criado um botão na barra superior da aplicação para que ele possa executar essa ação.

Para começar, adicione um novo ícone à pasta `res\drawable` para ser utilizado na barra superior. A sugestão é utilizar o ícone `ic_exit_to_app_black_24dp`.

Agora é necessário criar um menu para ele ser exibido. Para isso crie o arquivo `main_menu.xml` na pasta `res\menu`, como no trecho a seguir:

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

    <item
        android:id="@+id/nav_sign_out"
        android:icon="@drawable/ic_exit_to_app_black_24dp"
        android:title="Sign out"
        app:showAsAction="always" />
</menu>
```

Veja que o ícone que foi adicionado na pasta `res\drawable` deve ser utilizado na propriedade `android:icon`.

Agora que o menu existe com o ícone de logout, volte à `MainActivity` e crie a função responsável por exibi-lo, como no trecho a seguir:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.main_menu, menu)
    return true
}
```

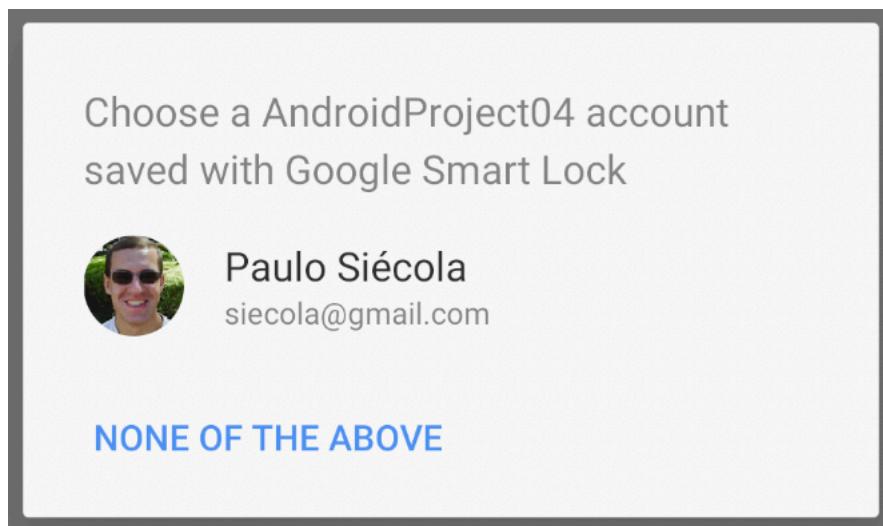
Perceba que o recurso de menu criado é utilizado na instrução `inflater.inflate(R.menu.main_menu, menu)`.

Isso já é suficiente para exibir o menu na tela ao usuário, porém ainda é necessário criar uma função capaz de tratar o evento de clique nos botões que ele exibe, e para isso, crie a função a seguir:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.nav_sign_out -> {
            AuthUI.getInstance()
                .signOut(this)
                .addOnCompleteListener {
                    this.recreate()
                }
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

Nesse trecho, é verificado qual item do menu foi clicado. Até o momento só existe o que realiza o logout da aplicação. Então a única ação necessária é invocar a instrução `signOut`, que possui um *listener* que é chamado quando o processo de logout é finalizado com sucesso. Aqui a solução foi simplesmente recriar a `MainActivity` para que ela peça para que um novo usuário se autentique, porém seria possível realizar outras operações de limpeza ou até mesmo redirecionar o usuário para uma outra tela.

Para testar a implementação até o momento, execute a aplicação em um dispositivo real ou um emulador. Na primeira vez deverá aparecer uma tela como a seguinte:



Tela de login do FirebaseUI

Escolha a conta do usuário para se autenticar e finalize sua autenticação. A tela de boas-vindas deve aparecer, assim como o botão de logout no canto superior direito da tela.

E é apenas isso para realizar o login do usuário com o Firebase Authentication!

No console do Firebase, é possível verificar os usuários que estão autenticados em todas as instâncias da aplicação, através do menu `Authentication`, na aba `Users`, como pode ser observado na figura a seguir:

Search by email address, phone number, or user UID			
Identifier	Providers	Created	Signed In
siecola@gmail.com	G	Apr 21, 2020	Apr 26, 2020

#### Lista de usuários autenticados

A próxima seção se encarregará de criar a estrutura base do projeto, aproveitando muito do que foi criado em `AndroiProject02`.

## 11.7 - Criando a estrutura base do projeto

A estrutura base do projeto será composta pelos seguintes itens principais:

- Layout da lista de produtos e seus itens;
- Layout da tela de detalhes do produto;
- Mecanismo de navegação entre as telas de lista e detalhes de produtos;
- Fragmento de tela, adaptador e ViewModel para a tela de lista de produtos;
- Fragmento de tela, adaptador e ViewModel para a tela de detalhe de produto.

Essa base poderá ser copiada de `AndroidProject02`, para isso abra esse projeto no Android Studio, mantendo o `AndroidProject04` também aberto, para copiar algumas partes de um para o outro.

Antes de começar o processo de cópia, crie o modelo que representará o produto no projeto. Para isso crie um novo pacote chamado `persistence` e dentro dele uma nova classe chamada `Product` , como no trecho a seguir:

```
import com.google.firebaseio.firebaseio.Exclude
import com.google.firebaseio.IgnoreExtraProperties

@IgnoreExtraProperties
data class Product(
    @Exclude var id: String? = null,
    var userId: String? = null,
    var name: String? = null,
    var description: String? = null,
    var code: String? = null,
    var price: Double? = null
)
```

Ele também representará o modelo que será salvo no Firestore, porém, sua utilização ainda será detalhada em seções mais adiante.

As instruções a seguir devem ser executadas sempre copiando algo de `AndroidProject02` para o projeto `AndroidProject04`. Durante a cópia dos itens e até mesmo ao final dela, não será possível compilar o projeto, pois algumas adaptações deverão ser feitas.

Depois que tudo for copiado e ajustado, será possível adicionar a camada de persistência dos produtos, através do repositório que acessará o Firestore.



Esse processo pode ser um pouco trabalhoso, mas vale a pena em relação a construir tudo do zero novamente, repetindo passos. Fique atento às correções que devem ser feitas, principalmente com o nome do pacote base, que é diferente entre os dois projetos.

Para começar, copie a pasta `res\navigation`, que contém as configurações de navegação da aplicação. Nas linhas 11 e 21 do arquivo `res\navigation\navigation_graph.xml`, ajuste o nome do pacote do projeto para `android_project04`. Veja que os pacotes `product` e `productdetail` ainda serão criados, assim como os layouts `fragment_products_list` e `fragment_product_detail`, por isso não se incomode com os erros nesse arquivo.

Agora copie todos os arquivos da pasta `res\layout`, sobrescrevendo o arquivo `res\layout\activity_main.xml`. Com isso, as alterações a seguir devem ser feitas nos seguintes arquivos:

- **`item_product.xml`**: altere o pacote base do projeto para `br.com.siecola.android_project04.persistence` na linha 9;
- **`fragment_products_list.xml`**: altere o pacote base do projeto para `android_project04` na linha 10;
- **`fragment_product_detail.xml`**: altere o pacote base do projeto para `android_project04` na linha 10 e remova todas as propriedades `android:enabled="false"` dos componentes `EditText`.

Agora copie o pacote `product` inteiro. Atente-se para corrigir o pacote base das classes dentro dessa pacote, pois o Android Studio pode não fazer isso sozinho:

De:

```
package br.com.siecola.androidproject02.product
```

Para:

```
package br.com.siecola.android_project04.product
```

Para todas as classes copiadas desse pacote, ajuste também os *imports* para apontarem para pacotes do novo projeto, com a base `br.com.siecola.android_project04`.

A classe `ProductListViewModel` merece uma atenção especial, pois ela está utilizando `coroutines` e o cliente REST que foi construído no projeto `AndroidProject02`, que não serão utilizados aqui. Por isso, remova as duas linhas a seguir dessa classe:

```
private var viewModelJob = Job()
private val coroutineScope = CoroutineScope(viewModelJob + Dispatchers.Main)
```

Também remova a seguinte instrução da função `onCleared`:

```
viewModelJob.cancel()
```

Já a função `getProducts` será toda refeita, por isso apague todo o seu conteúdo, deixando apenas a declaração da função em si.

Agora é a hora do pacote `productdetail`. Copie para projeto `AndroidProject04` e repita os mesmos ajustes realizados no pacote `product`, começando pela correção do pacote base das classes:

De:

```
package br.com.siecola.androidproject02.product
```

Para:

```
package br.com.siecola.android_project04.product
```

Depois ajuste os *imports* para apontarem para pacotes do novo projeto, com a base `br.com.siecola.android_project04`.

A classe `ProductDetailViewModel` também merece uma atenção especial, pois ela também estava utilizando coroutines e o cliente REST. Por isso remova as seguintes instruções:

```
private var viewModelJob = Job()
private val coroutineScope = CoroutineScope(viewModelJob + Dispatchers.Main)
```

Também remova a instrução a seguir da função `onCleared`:

```
viewModelJob.cancel()
```

A função `getProducts` também será toda refeita, por isso apague todo o seu conteúdo, deixando apenas sua declaração.

Para finalizar os ajustes nessa classe, altere o tipo dos atributos de acordo com o trecho de código a seguir:

```
private lateinit var _products: MutableLiveData<List<Product>>
val products: LiveData<List<Product>>
    get() = _products
```

Esses ajustes finais farão com que essa classe fique preparada para as alterações que ainda serão realizadas nela.

Nesse momento deve compilar sem nenhum erro. Caso ainda existam algum, volte aos passos descritos nessa seção, se atentando principalmente às modificações relacionadas à alteração dos nomes dos pacotes.

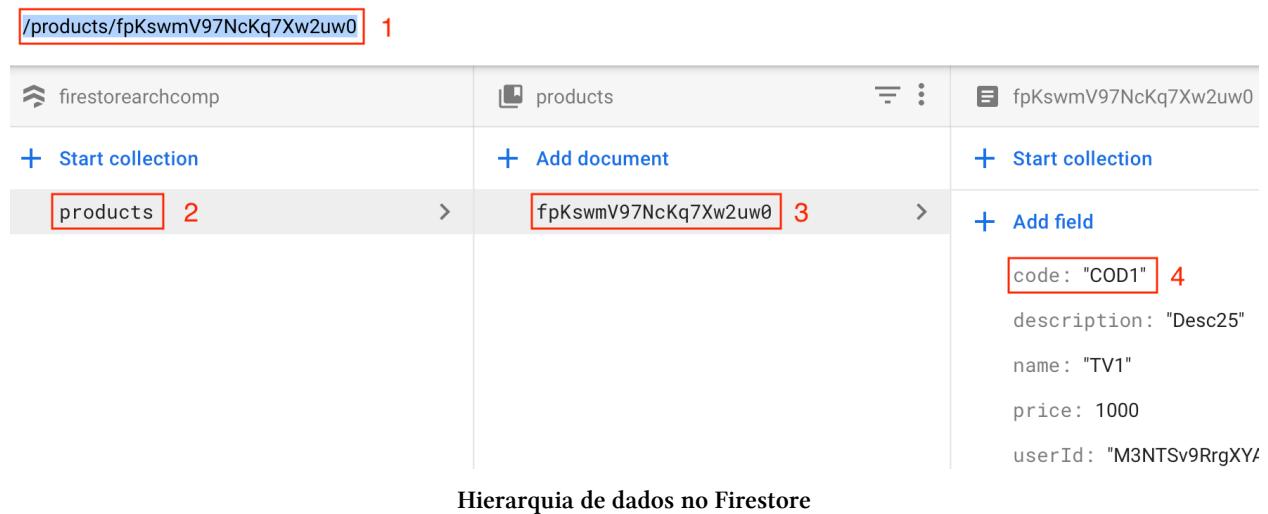
Agora a estrutura base do projeto está preparada para começar a implementação da camada que irá persistir os dados no Firestore.

## 11.8 - Entendendo a hierarquia de dados no Firestore

No Firestore, a hierarquia de dados funciona da seguinte forma:

- **Documento:** essa é a unidade básica de informação que representa um registro, que pode conter campos. Por exemplo, um produto salvo será representado por um documento, com uma identificação única. Os atributos do produto serão campos desse documento;
- **Coleção:** a coleção agrupa documentos. No Firestore, sempre deve haver pelo menos uma coleção, ou seja, nenhum documento é persistido fora de uma coleção;
- **Campo:** como explicado, o campo é um atributo dentro de um documento;
- **Subcoleção:** uma subcoleção pode existir dentro de um documento, que já pertence a uma coleção. Pode ser entendido como um campo de um documento que possui uma lista de outros documentos.
- **Referência:** é como um endereço que permite apontar para uma coleção, para um documento específico dentro de uma coleção representado por um identificador único, para uma subcoleção dentro de um documento em uma coleção e assim por diante.

A figura a seguir ilustra um exemplo de uma coleção chamada `products`, com um documento apenas e seus campos:



Em (1) é possível observar a referência do documento (3) de identificação `fpKswmV97NcKq7Xw2uw0` que foi selecionado na coleção `product` em (2). Veja que o documento possui alguns campos (4), cada um com seu tipo e nome.

Obviamente a coleção `products` pode conter mais documentos, assim como qualquer documento poderia conter subcoleções.

Também é possível que o projeto no Firestore contenha outras coleções em sua raiz, como por exemplo uma que contivesse `orders`, com documentos que representassem pedidos, como em uma loja.

A aplicação que está sendo desenvolvida nessa capítulo terá apenas uma coleção de produtos, de nome products. Cada documento dessa coleção será um produto e os campos serão os atributos de cada um deles.

Na figura anterior observe que o documento possui um campo chamado userId. Ele guarda a identificação do usuário dono do produto, ou seja, quem foi que o criou. Essa informação será utilizada para definir as regras de acesso, na próxima seção.

## 11.9 - Criando as regras de acesso ao Firestore

Como dito anteriormente nesse capítulo, com o Firestore é possível definir regras de acesso, como por exemplo:

- Permitir o acesso somente a usuários que estejam autenticados com o Firebase Authentication;
- Restringir o acesso a recursos que somente o usuário criou.

Obviamente tais restrições devem ser definidas de acordo com as regras de negócio do sistema sendo desenvolvido.

Na entidade Product que foi criada, existe um campo chamado userId, do tipo String:

```
import com.google.firebaseio.firebaseio.Exclude
import com.google.firebaseio.IgnoreExtraProperties

@IgnoreExtraProperties
data class Product(
    @Exclude var id: String? = null,
    var userId: String? = null,
    var name: String? = null,
    var description: String? = null,
    var code: String? = null,
    var price: Double? = null
)
```

A ideia é armazenar a identificação única do usuário autenticado no Firebase dentro desse campo. Com isso será possível fazer com que o usuário tenha acesso somente aos produtos criados por ele. A anotação @Exclude diz que esse campo não deve fazer parte como campo do documento Product, mas sim existir apenas no modelo do projeto AndroidProject04.

Para complementar e fazer com que essa primeira regra funcione, é necessário restringir a operação de criação de um novo documento somente a usuário autenticados. Pode parecer redundante, mas a regra anterior se refere ao acesso a uma entidade já existente e essa à criação de uma nova.

Tais regras devem ser criadas no console do Firebase. Para isso, acesse o menu Database na seção Develop. A página inicial oferece a opção para a criação de um novo banco de dados, por isso clique no botão Create Database. Essa opção deve exibir a seguinte tela:

The screenshot shows the first step of a 'Create database' wizard. The title 'Create database' is at the top. Below it, two numbered steps are shown: '1 Secure rules for Cloud Firestore' (with a blue circle) and '2 Set Cloud Firestore location' (with a grey circle). A note below the steps reads: 'After you define your data structure, you will need to write rules to secure your data.' followed by a 'Learn more' link with a help icon.

**Start in production mode**

Your data will be private by default. Client read/write access will only be granted as specified by your security rules.

**Start in test mode**

Your data will be open by default to enable quick setup. Client read/write access will be denied after 30 days if security rules are not updated.

On the right side, there is a code snippet for Cloud Firestore security rules:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
  }
}
```

**i All third party reads and writes will be denied**

### Criando um banco de dados

Como o intuito dessa seção é detalhar o processo de criação das regras de segurança para um banco de dados no Firestore, selecione a opção Start in production mode, o que faz com que todos os acessos aos dados desse banco sejam submetidos às regras que serão criadas aqui.

Para prosseguir, clique no botão Next. Na próxima tela, escolha a localidade desejada para o banco de dados e em seguida clique em Done.

Depois que o banco for criado, deverá aparecer o seguinte console de administração:

The screenshot shows the Firebase Database console interface. At the top, there's a navigation bar with the word "Database" on the left and a "Cloud Firestore" dropdown on the right. Below the navigation bar, there are four tabs: "Data" (which is underlined in blue, indicating it's the active tab), "Rules", "Indexes", and "Usage". In the main content area, there's a list of collections. The first collection is named "androidbook2-9e37d" and has a small icon next to it. Below this collection, there's a large blue button with a plus sign and the text "Start collection".

Console Firestore

Os dados, quando eles forem criados, deverão aparecer na seção onde está escrito Start collection. Em breve a coleção de produtos será criada e deverá aparecer nesse local.

Na parte superior dessa tela, clique na aba Rules para configurar as regras de acesso a esse banco de dados, que deverão ser as seguintes:

- Permitir o acesso somente a usuários que estejam autenticados com o Firebase Authentication;
- Restringir o acesso a recursos que somente o usuário criou, ou seja, operação de escrita e leitura só poderão ser feitas pelo usuário que criou o produto.

Para isso, a seguinte regra deve ser criada:

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /products/{product} {
      allow create: if request.auth.uid != null;
      allow read, update, delete: if request.auth.uid != null
          && resource.data.userId == request.auth.uid;
    }
  }
}
```

Perceba que a seção `match /products/{product}` define que as regras dentro dela se referem à coleção `products` e seus documentos. Dentro dessa seção existem duas regras:

- `allow create: if request.auth.uid != null`: essa regra define que somente usuários autenticados poderão acessar essa coleção, por isso o aplicativo aqui desenvolvido requer que o usuário se autentique com o **Firebase Authentication**;
- `allow read, update, delete: if request.auth.uid != null && resource.data.userId == request.auth.uid`: essa regra define que operações de leitura, alteração e exclusão só poderão ser feitas se os recursos acessados, no caso os produtos, tiverem o campo `userId` igual à identificação do usuário autenticado tentando realizar a operação. Isso garante que somente o usuário que criou o recurso acesse-o.

Perceba que a operação de criação está condicionada apenas ao usuário estar autenticado. Mais adiante será mostrado que durante esse processo, a identificação única do usuário será salva no campo `userId` do produto, fazendo com que somente ele acesso-o novamente.

Veja também que, apesar desse banco possuir apenas uma coleção, as regras podem ser diferentes para cada coleção, permitindo, por exemplo, que alguns dados fiquem abertos completamente, ou compartilhados com apenas alguns usuários escolhidos.

Para concluir a criação dessas regras, clique no botão `Publish`. Dentro de alguns minutos as regras criadas aqui terão efeito.

Agora que a regra de controle de acesso à coleção de produtos foi criada no Firestore, é possível partir para a implementação final do lado do aplicativo Android.

## 11.10 - Criando o repositório de produtos

O repositório de produtos é a camada do projeto que será responsável por acessar a coleção de produtos no Firestore. A seguir, as operações que essa camada deverá realizar:

- Criar ou alterar um produto;
- Apagar um produto existente através da sua identificação única;

- Localizar todos os produtos de um determinado usuário;
- Localizar um produto pelo seu código.

Para começar, criar uma nova classe no pacote `persistence` chamada `ProductRepository`, como no trecho a seguir:

```
import android.util.Log
import androidx.lifecycle.MutableLiveData
import com.google.firebase.auth.FirebaseAuth
import com.google.firebaseio.firebaseio.FirebaseFirestore
import com.google.firebaseio.firebaseio.Query
import com.google.firebaseio.ktx.toObject

private const val TAG = "ProductRepository"

private const val COLLECTION = "products"
private const val FIELD_USER_ID = "userId"
private const val FIELD_NAME = "name"
private const val FIELD_DESCRIPTION = "description"
private const val FIELD_CODE = "code"
private const val FIELD_PRICE = "price"

object ProductRepository {
```

```
}
```

As constantes aqui declaradas irão ajudar na construção das funções para realizar as operações listadas anteriormente. Dessa forma ficará mais fácil se referir ao nome da coleção e aos nomes dos campos do documento.

Para fazer com que o repositório de produtos acesse a identificação única do usuário que está autenticado na aplicação, crie um atributo privado do tipo `FirebaseAuth`:

```
private val firebaseAuth: FirebaseAuth by lazy {
    FirebaseAuth.getInstance()
}
```

Dessa forma, quando algum produto precisar ser salvo ou localizado, a identificação única do usuário será retirada desse atributo.

E para fazer com que o repositório acesse efetivamente o `Firestore`, crie um outro atributo privado do tipo `FirebaseFirestore`, como no trecho a seguir:

```
private val firebaseFirestore: FirebaseFirestore by lazy {
    FirebaseFirestore.getInstance()
}
```

Com esse atributo será possível realizar as operações na coleção de produtos.

Agora prossiga com a criação da função para criar ou salvar um novo produto, como no trecho a seguir:

```
fun saveProduct(product: Product): String {
    val document = if (product.id != null) {
        firebaseFirestore.collection(COLLECTION).document(product.id!!)
    } else {
        product.userId = FirebaseAuth.getInstance().uid
        firebaseFirestore.collection(COLLECTION).document()
    }
    document.set(product)

    return document.id
}
```

Veja que a função recebe como parâmetro um objeto do tipo `Product`, que deverá ser construído a partir das informações preenchidas pelo usuário na tela de edição e produtos. O retorno dessa função é a identificação única do produto, caso alguma parte do projeto necessite.

Repare que a primeira instrução é uma verificação para ver se o produto recebido já possui uma identificação única. Caso possua, significa que é um produto que já foi cadastrado e ele deve estar sendo alterado.

Nesse caso a instrução `firebaseFirestore.collection(COLLECTION).document(product.id!!)` busca na coleção `products` o documento com essa identificação única fornecida dentro do atributo `id` de `product`. O que essa instrução na verdade faz, é buscar no Firestore o documento com o seguinte endereço: `/products/{identificação do produto}`.

Caso o produto recebido pela função não tenha uma identificação única, é possível que a intenção seja criar um novo. Com isso é necessário buscar a identificação única do usuário para poder guardar dentro do produto a ser criado, pois esse campo será utilizado pelas regras de segurança criadas na seção anterior.

Veja que aqui um novo documento é criado dentro da coleção `products`, através da instrução `firebaseFirestore.collection(COLLECTION).document()`.

Em ambos os casos, a variável `document` conterá o produto a ser persistido no Firestore. Por isso a instrução `document.set(product)` atualiza esse documento com as alterações do produto recebidas pela função.

Para finalizar, a identificação única do produto é retornada pela função.

Agora crie a função para excluir um produto pela sua identificação, como no trecho a seguir:

```
fun deleteProduct(productId: String) {
    val document = firebaseFirestore.collection(COLLECTION).document(productId)
    document.delete()
}
```

Veja que novamente o documento com a identificação única é buscado na coleção products, para que seja excluído através da instrução `document.delete()`.

A função para listar todos os produtos possui um requisito a mais que a torna mais complexa de ser implementada. A ideia é utilizar a possibilidade de receber atualizações dos produtos, caso eles sejam alterados no Firestore por outra aplicação. Para que isso possa ser alcançado, é necessário fazer com que essa função não retorne simplesmente uma lista de produto, mas sim uma lista de produtos dentro de um `MutableLiveData`, que é uma classe que permite a notificação de *listeners* sobre a mudança de dados em seu conteúdo.

Fazer com que a função de listar todos produtos retorne um `MutableLiveData` fará com que sua utilização fique totalmente aderente ao conceito utilizado de `ViewModel`, que permitirá que a tela com a lista de produtos receba atualizações sobre a lista exibida.

Essa função também deverá trazer somente os produtos que foram criados pelo usuário que está autenticado na aplicação, pois é o que as regras criadas para a coleção de produtos diz. Para isso, será necessário criar uma consulta que filtre pelo campo `userId` do produto.

Veja como deve ficar essa função com todos os requisitos listados acima:

```
fun getProducts(): MutableLiveData<List<Product>> {
    val liveProducts = MutableLiveData<List<Product>>()

    firebaseFirestore.collection(COLLECTION)
        .whereEqualTo(FIELD_USER_ID, firebaseAuth.uid)
        .orderBy(FIELD_NAME, Query.Direction.ASCENDING)
        .addSnapshotListener { querySnapshot, firebaseFirestoreException ->
            if (firebaseFirestoreException != null) {
                Log.w(TAG, "Listen failed.", firebaseFirestoreException)
                return@addSnapshotListener
            }

            if (querySnapshot != null && !querySnapshot.isEmpty) {
                val products = ArrayList<Product>()
                querySnapshot.forEach {
                    val product = it.toObject<Product>()
                    product.id = it.id
                    products.add(product)
                }
                liveProducts.value = products
            }
        }
}
```

```

        }
        liveProducts.postValue(products)
    } else {
        Log.d(TAG, "No product has been found")
    }
}

return liveProducts
}

```

A primeira instrução cria uma variável do tipo `MutableLiveData<List<Product>>` para ser devolvida pela função, contendo a lista de produtos pesquisada.

A segunda instrução cria a pesquisa na coleção `products`, filtrando aqueles que possuem o campo `userId` iguais à identificação única do usuário que está autenticado na aplicação nesse momento, assim como ordena o resultado pelo campo `name` em ordem crescente.

A instrução `addSnapshotListener` é o ponto chave para fazer com que essa função atualize o `MutableLiveData<List<Product>>` com alterações na lista solicitada, que por sua vez irá notificar os *listeners* interessados nessas mudanças.

O resultado dessa chamada é retornado em uma variável do tipo `QuerySnapshot`, que contém o resultado da pesquisa em lista. As instruções dentro do *listener* então varrem essa lista, convertendo cada item em um produto e atribuindo a identificação única a cada um deles. Esses produtos então são adicionados na variável `liveProducts` para serem devolvidos pela função.

Veja que a consulta utilizando a instrução `addSnapshotListener` é assíncrona, o que faz com que inicialmente a função retorne uma lista vazia, que será atualizada depois que a consulta for realizada. Da mesma forma, caso exista alguma alteração, o *listener* será novamente invocado para atualizar a lista de produtos, consequentemente, a tela será atualizada com a alteração. Isso será simulado em algumas seções adiante.

Por último, implemente a função para buscar um produto pelo seu código. A implementação é bem semelhante a anterior, com apenas algumas modificações, como pode ser observado no trecho a seguir:

```

fun getProductByCode(code: String): MutableLiveData<Product> {
    val liveProduct: MutableLiveData<Product> = MutableLiveData()

    firebaseFirestore.collection(COLLECTION)
        .whereEqualTo(FIELD_CODE, code)
        .whereEqualTo(FIELD_USER_ID, firebaseAuth.uid)
        .addSnapshotListener { querySnapshot, firebaseFirestoreException ->
            if (firebaseFirestoreException != null) {
                Log.w(TAG, "Listen failed.", firebaseFirestoreException)
            }
            return@addSnapshotListener
        }
}

```

```
    }

    if (querySnapshot != null && !querySnapshot.isEmpty) {
        val products = ArrayList<Product>()
        querySnapshot.forEach {
            val product = it.toObject<Product>()
            product.id = it.id
            products.add(product)
        }
        liveProduct.postValue(products[0])
    } else {
        Log.d(TAG, "No product has been found")
    }
}

return liveProduct
}
```

Aqui o esperado é que apenas um produto seja retornado, e não uma lista. Por isso o tipo do retorno da função é um `MutableLiveData<Product>`.

Repare que agora a consulta é montada com uma condição a mais: pesquisar um produto onde o campo `code` é igual ao código do produto recebido pela função.



Buscando uma simplicidade didática nessa implementação, apenas um produto é retornado, uma vez que não há um mecanismo para evitar que um mesmo código de produto seja cadastrado para um mesmo usuário.

Quem invocar essa função também terá a oportunidade de observar as mudanças do produto buscado.

## 11.11 - Criando e visualizando os dados do Firestore com Two-way data binding

Para utilizar o repositório de produtos criado na seção anterior, criar e visualizar produtos do Firestore, é necessário continuar com as modificações no projeto, aproveitando em partes de `AndroidProject02`. Também serão incluídas as opções para criação e exclusão de produtos, algo que não havia no antigo projeto.

Para começar, abra a classe `ProductListViewModel`. Essa classe já foi previamente preparada com algumas alterações, mas agora que o repositório de produtos foi concluído, é possível utilizá-los na função `getProducts`, como no trecho a seguir:

```
private fun getProducts() {
    _products = ProductRepository.getProducts()
}
```

Veja que o que deve ser feito aqui é simplesmente chamar a função `getProducts` do repositório de produtos. O retorno dessa função é um objeto do tipo `MutableLiveData<List<Product>>`, o mesmo tipo do atributo `_products` desse `ViewModel`. Com isso, quando um produto da lista for alterado, o fragmento que exibe esses produtos também será alterado, se ele estiver sendo exibido. Isso tudo é feito de forma automática pela arquitetura construída, utilizando `RecyclerView` em conjunto com o `ViewModel`.

O próximo passo é ajustar a navegação entre as telas de lista e detalhes de produto, pois agora é necessário criar uma nova forma de navegação para a adição de um novo produto. Com isso, o argumento da ação para chamar `fragmentProductDetail`, criado no arquivo `res\-navigation\navigation_graph.xml`, deve ser alterado para permitir valores nulos, como no trecho a seguir:

```
<fragment
    android:id="@+id/fragmentProductDetail"
    android:name="br.com.siecola.android_project04.productdetail.ProductDetailFragme\
nt"
    android:label="fragment_product_detail"
    tools:layout="@layout/fragment_product_detail" >
    <argument
        android:name="productCode"
        app:nullable="true"
        app:argType="string" />
</fragment>
```

Repare que foi adicionado o atributo `app:nullable="true"`, que faz com que o atributo `productCode` do tipo `String` possa ser passado com valor nulo. Isso é necessário para a ação de criar um novo produto, que obviamente não possui nenhum código para fornecer para a tela de edição do produto.

A ação que irá disparar a criação de um novo produto começa com o clique em um botão na tela principal. Para esse botão será criado um *Float Action Button*, que nada mais é, como o próprio nome diz, um botão que flutua por cima da lista de produtos. Para isso, abra o arquivo `res\layout\fragment_products_list.xml` e acrescente o seguinte trecho de código logo após o fechamento da tag `</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>`:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16dp"
    android:layout_marginEnd="16dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:srcCompat="@drawable/ic_add_black_24dp" />
```

Perceba que ele utiliza um recurso de imagem. Aqui foi escolhido o `ic_add_black_24dp`, logo ele deve ser adicionado na pasta `res\drawable`, da mesma forma como foi feito com outros recursos que foram adicionados em outros projetos anteriores.

A declaração do *Floating Action Button* define sua identificação única e características de seu posicionamento. O tratamento do clique nesse botão deve ficar na classe `ProductsListFragment`, antes do retorno da função `onCreateView`, como no trecho a seguir:

```
binding.fab.setOnClickListener { view -
    this.findNavController()
        .navigate(ProductsListDirections.actionShowProductDetail(null))
}
```

Veja que, quando esse botão for clicado, o usuário será direcionado para a tela de detalhes de produto, mas o código informado é nulo, o que fará com que o `ViewModel` de detalhes de produto entenda que na verdade um novo produto deverá ser criado. Por isso foi necessário alterar o argumento da navegação no arquivo `res\navigation\navigation_graph.xml`.

Como agora o código do produto pode ser passado como nulo, para o caso da criação de um novo, é necessário fazer com que `ProductDetailViewModel` e seu `ProductDetailViewModelFactory` se adaptem. Para isso, abra a classe `ProductDetailViewModelFactory` e altere o tipo do atributo `code` para `String?`, como no trecho a seguir:

```
class ProductDetailViewModelFactory(private val code: String?)
```

O mesmo deve ser feito para na classe `ProductDetailViewModel`:

```
class ProductDetailViewModel(code: String?) : ViewModel() {
```

Essa é a classe que terá mais adaptações nesse trabalho, primeiramente, comece removendo todos os atributos que ela ainda possui, deixando apenas o seguinte:

```
lateinit var product: MutableLiveData<Product>
```

Ele irá representar o produto que essa tela está alterando ou criando, dependendo da ação iniciada pelo usuário. A grande diferença aqui é que a tela poderá ler e escrever nesse atributo, para ambas ações mencionadas.

A função `init` também deve ser adaptada a esse contexto. Veja como ela deve ficar:

```
init {
    if (code != null) {
        getProduct(code)
    } else {
        product = MutableLiveData<Product>()
        product.value = Product()
    }
}
```

Perceba que, caso o código do produto seja diferente de nulo, ele será exibido através da chamada da função `getProduct`, passando seu código. Caso contrário, um novo produto será criado e atribuído ao `MutableLiveData` da classe.

Já a função `getProduct` apenas fará a ponte para a chamada ao repositório de produtos, como no trecho a seguir:

```
private fun getProduct(productCode: String) {
    product = ProductRepository.getProductByCode(productCode)
}
```

Da mesma forma como foi feito com a lista de produtos, o retorno da função `getProductbyCode` do repositório de dados retorna o tipo `MutableLiveData<Product>`, que faz com que a tela continue observando caso ele seja alterado.

E finalmente, a função `onCleared` pode ser utilizada para persistir as alterações feitas no produto em edição ou criar um novo, caso essa tenha sido a ação iniciada pelo usuário. Essa função é ideal para isso, pois ela será chamada imediatamente antes da tela deixar de ser exibida ao usuário, quando ele pressionar o botão de voltar do dispositivo, ou seja, um bom momento para salvar o que ele fizer nessa tela. Veja como essa função deve ficar:

```
override fun onCleared() {
    if (product.value != null) {
        ProductRepository.saveProduct(product.value!!)
    }
    super.onCleared()
}
```

Porém, a forma como esse ViewModel está sendo utilizado, através do *binding* com a View que exibe o layout fragment\_product\_detail, só é possível fazer com que a View leia as informações do produto, que era o objetivo quando AndroidProject02 foi construído.

Agora é necessário fazer com que a View também escreva na variável productDetailViewModel.product declarada no arquivo res\layout\fragment\_product\_detail.xml:

```
<data>

<variable
    name="productDetailViewModel"
    type="br.com.siecola.android_project04.productdetail.ProductDetailViewModel"\>
/>
</data>
```

O nome da técnica para fazer com que uma View leia e escreva em um *data binding* é conhecido como Two-way data binding. Dessa forma, a View, além de ser responsável por ler o conteúdo que está no ViewModel, também pode reescrever as alterações feitas pelo usuário.

A alteração necessária para habilitar o Two-way data binding é bem simples e pode ser feita no arquivo res\layout\fragment\_product\_detail.xml. Basta trocar a notação do *binding* realizado nas caixas de edição de texto dos campos nome, código e descrição do produto. Veja como deve ficar:

- De:

```
    android:text="@{productDetailViewModel.product.name}"
```

- Para:

```
    android:text="@=productDetailViewModel.product.name"
```

Repare que o que mudou foi apenas a adição do sinal = logo após o @. Faça isso para os campos código e descrição para habilitar o Two-way data binding para esses campos também.

Para o campo com o preço do produto a estratégia é ligeiramente diferente, pois ele possui um conversor para a exibição do dado. Por isso crie um novo arquivo na pasta product chamado ProductConverter:

```

import androidx.databinding.InverseMethod

object PriceConverter {
    @InverseMethod("stringToPrice")
    @JvmStatic
    fun priceToString(newValue: Double): String {
        return "$ " + "%.2f".format(newValue)
    }

    @JvmStatic
    fun stringToPrice(newValue: String): Double {
        var price = 0.0
        try {
            price = newValue.removePrefix("$").trim().toDouble()
        } catch (e: NumberFormatException) {
            Log.e("PriceConverter", " Failed to convert price")
        }
        return price
    }
}

```

Essas funções serão utilizadas para converter o preço do produto em String, com o caracter \$ na frente, e fazer o processo reverso, transformando uma String em um valor Double.

Esse conversor agora será utilizado para exibir o valor do produto e também para converter o que o usuário digitar, para ser salvo no Firestore. Por isso, substitua o atributo app:productPrice="@{productDetailViewModel.product.price}" pelo seguinte:

```
android:text="@={PriceConverter.priceToString(productDetailViewModel.product.price)}"
```

Para esse conversor ser entendido pelo projeto, importe-o no começo do arquivo, logo após a abertura da tag data:

```

<data>
    <import type="br.com.siecola.android_project04.product.PriceConverter"/>

    <variable
        name="productDetailViewModel"
        type="br.com.siecola.android_project04.productdetail.ProductDetailViewModel"\/>
</data>

```

Com essas implementações já é possível testar o aplicativo, mas ainda é necessário criar indexes no Firestore para que as pesquisas que serão realizadas sejam efetivamente concluídas. Lembre-se que atualmente o aplicativo realiza duas pesquisas:

- Listar todos os produtos criados por um determinado usuário, ordenando-os pelo nome do produto;
- Buscar produtos pelo seu código, sendo criado por um determinado usuário.

Sem a criação desses índices não é possível que o aplicativo realize tais pesquisas. O Firestore possui uma interface amigável para a criação desses índices, mas o Android Studio também fornece uma ajuda interessante para esse processo, que é o fornecimento de um link já com todas as instruções ao Firestore para que ele crie o índice.

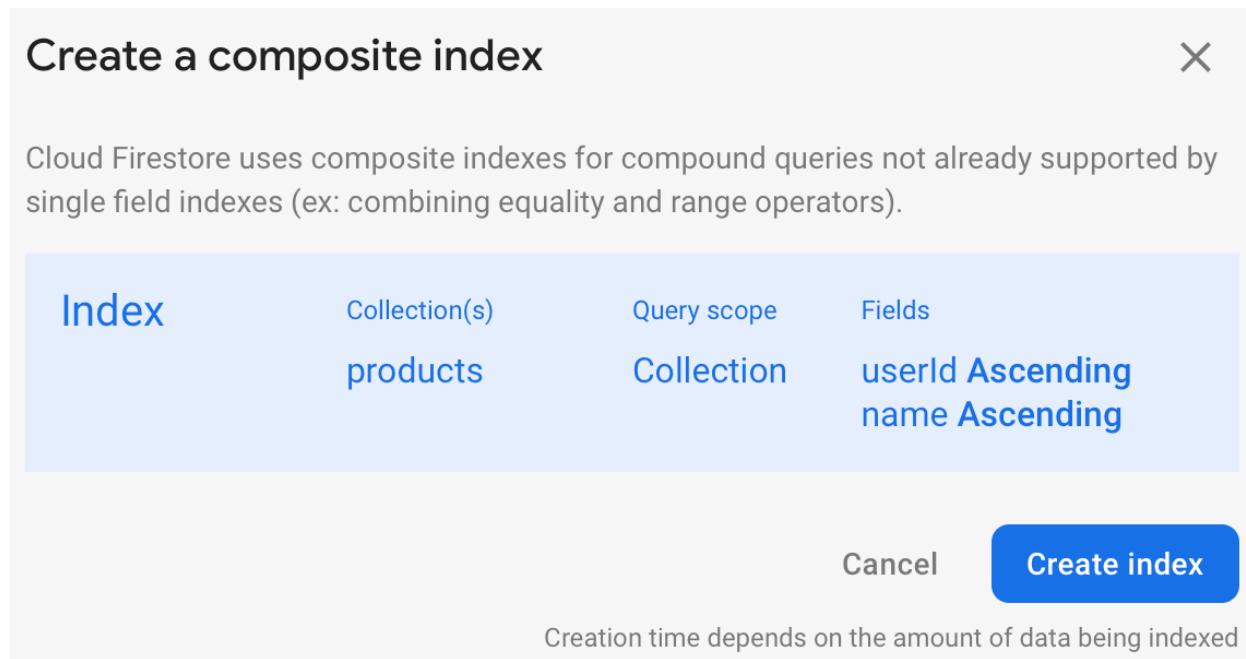
Para obter esse link no Android Studio, execute a aplicação em um emulador ou dispositivo real e faça o login do usuário. Depois do login do usuário, a aplicação tenta buscar todos os produtos que ele criou, que obviamente não haverá nenhum, porém o Android Studio imprime uma mensagem de erro no Logcat dizendo que é necessário criar o índice para essa pesquisa. Veja um exemplo na figura a seguir:

```
0: JAVAFX-1.3.1.1@21.4.3:115 [1]: https://console.firebaseio.google.com/v1/r/project/androidbook2-9e37d.firebaseioindexes
at com.google.firebaseio.util.AsyncQueue$SyncronizedShutdownAwareExecutors$DelayedStartFactory.run(com.google.firebaseio.firebaseio@21.4.3:229)
at java.lang.Thread.run(Thread.java:919)
Caused by: io.grpc.StatusException: FAILED_PRECONDITION: The query requires an index. You can create it here: https://console.firebaseio.google.com/v1/r/project/androidbook2-9e37d.firebaseioindexes
at io.grpc.Status.asException(Status.java:541)
at com.google.firebaseio.util.Util.exceptionFromStatus(com.google.firebaseio.firebaseio@21.4.3:115) <27 more...>
2020-05-02 17:44:41.711 21967-22051/com.siecola.android_project04 W/Firestore: Listen for Query(target=Query(products where userId == # com.google.firebaseio.v1.Value@78844ebd
integer_value: 0
```

#### Link para criação do primeiro índice de pesquisa

Clique nesse link que aparece no Android Studio para ser redirecionado para a tela de criação de índices no Firestore.

Quando o link for aberto, deverá aparecer uma tela como a figura a seguir:



Criando o primeiro índice no Firestore

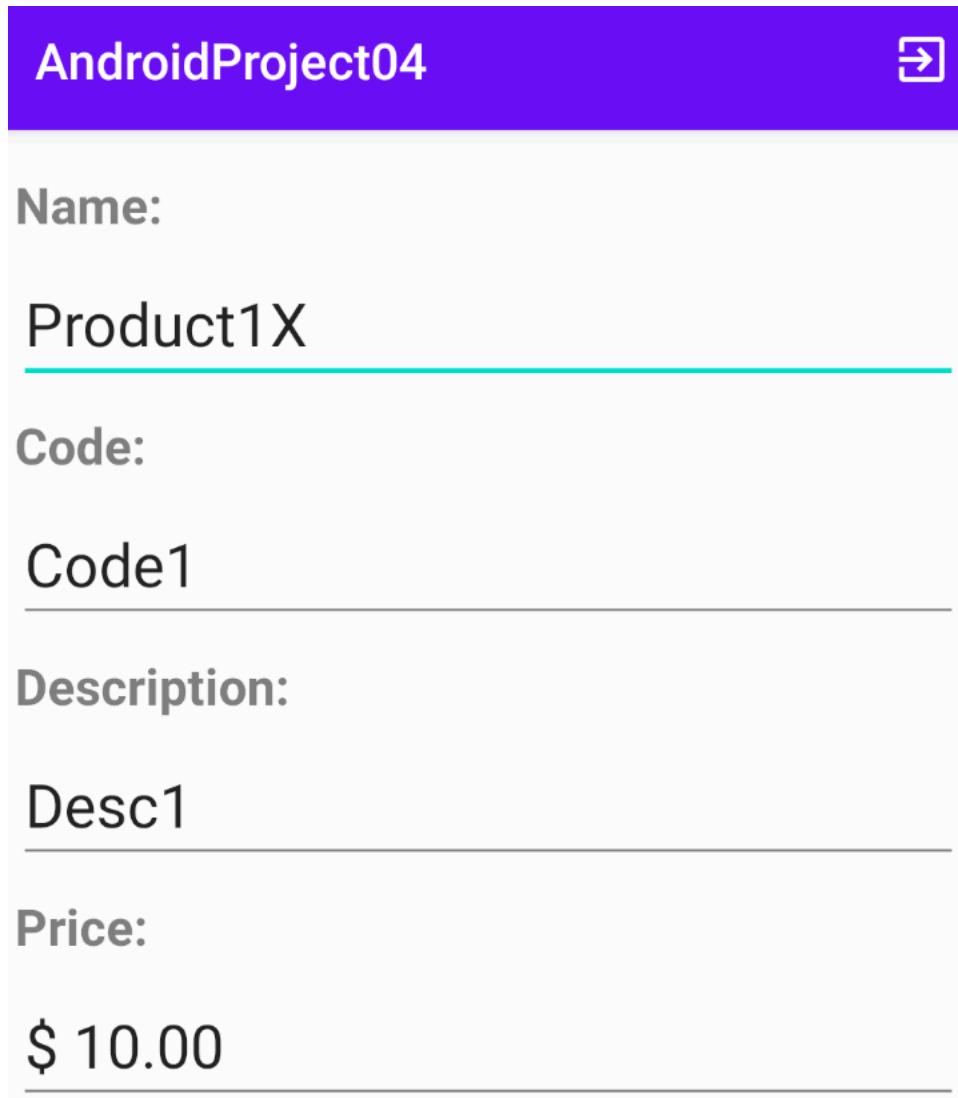
Repare que as informações desse primeiro índice são exatamente as mencionadas anteriormente. Para prosseguir, clique no botão `Create index`. Após alguns segundos, o índice será criado e o aplicativo poderá realizar essa pesquisa.

Depois que o índice for criado, execute a aplicação novamente. Perceba que haverá um botão localizado no canto inferior esquerdo da tela, com um sinal de +. Clique nele para ir para a tela de criação de um novo produto.

Preencha os dados do produto que deseja criar e clique no botão de voltar do dispositivo. O produto deverá aparecer na tela de listagem e produtos, como na figura a seguir:

AndroidProject04		
Product1	Code1	\$ 10.00
Product2	Code2	\$ 20.00
Criando um novo produto		

E para editar um produto criado, clique sobre ele para ser redirecionado à tela de edição, como mostra a figura a seguir:



#### Editando um produto

Nessa tela, altere qualquer campo e clique no botão de voltar do dispositivo. O produto alterado deverá aparecer atualizado na lista dos produtos na primeira tela.

Esses produtos também podem ser vistos no console de administração do Firestore, na aba Data, como pode ser visto na figura a seguir:

Produtos no console do Firestore

products	M1ZySTcPOABNRb6788AX	WVK8PXv91gmtaDZPbUmz
		code: "Code1" description: "Desc1" id: "WVK8PXv91gmtaDZPbUmz" name: "Product1" price: 10 userId: "T1WMefJ1S6Pql"

Veja que os produtos contêm a identificação única do usuário que os criou, permitindo que as regras de segurança avaliem esse campo, restringindo o acesso aos usuários donos dos dados.

## 11.12 - Simulando a atualização em tempo real

Agora que os produtos já estão sendo cadastrados no Firestore, é possível simular uma funcionalidade muito interessante dele, que é a atualização dos dados em tempo real, em todos os dispositivos que estiverem observando-os.

Lembre-se que a função que lista todos os produtos no repositório que foi construído executa a seguinte chamada:

```
fun getProducts(): MutableLiveData<List<Product>> {
    val liveProducts = MutableLiveData<List<Product>>()

    firebaseFirestore.collection(COLLECTION)
        .whereEqualTo(FIELD_USER_ID, firebaseAuth.uid)
        .orderBy(FIELD_NAME, Query.Direction.ASCENDING)
        .addSnapshotListener { querySnapshot, firebaseFirestoreException ->
            if (firebaseFirestoreException != null) {
                Log.w(TAG, "Listen failed.", firebaseFirestoreException)
                return@addSnapshotListener
            }

            if (querySnapshot != null && !querySnapshot.isEmpty) {
                val products = ArrayList<Product>()
                querySnapshot.forEach {

```

```

        val product = it.toObject<Product>()
        product.id = it.id
        products.add(product)
    }
    liveProducts.postValue(products)
} else {
    Log.d(TAG, "No product has been found")
}
}

return liveProducts
}

```

A instrução `addSnapshotListener` cria um *listener* para ouvir alterações nas entidades resultantes da pesquisa, que por sua vez altera a coleção de produtos dentro de um `MutableLiveData`. Isso faz com que quem o chamou seja notificado das alterações em qualquer dessas entidades.

Nesse caso, se a tela de listagem de produtos estiver aberta, ela será notificada de qualquer alteração nos produtos em que exibe, através da ligação entre `ProductListViewModel`, que chamou o repositório de dados e portanto ouve as alterações em `MutableLiveData<List<Product>>`, e o componente `RecyclerView` responsável por efetivamente exibir a lista.

É possível testar esse comportamento de duas formas diferentes:

- Executar a aplicação com o mesmo usuário autenticado em dois dispositivos diferentes;
- Alterar os dados diretamente no console do Firestore, clicando em um documento e alterando um de seus campos.

Em qualquer uma das situações acima, você perceberá que qualquer alteração nos dados, será refletida imediatamente em todos os clientes que estiverem executando a aplicação com o mesmo usuário autenticado.

Também é possível criar um novo produto em um dispositivo, ou no console do Firestore, e ver que ele aparece imediatamente no outro dispositivo.

Essa funcionalidade do Firestore elimina a necessidade da criação de um mecanismo de sincronismo entre os dados locais de um dispositivo e os armazenados no Firestore, assim como facilita muito o compartilhamento de informações entre vários dispositivos executando a aplicação e observando a mesma porção de dados.

Para ver uma demonstração desse mecanismo, consulte esse [vídeo<sup>34</sup>](#).

---

<sup>34</sup><https://www.youtube.com/watch?v=6qpRgD-yki0>

## 11.13 - Conclusão

Esse capítulo demonstrou como construir uma aplicação Android utilizando o Firestore e o Firebase Authentication para autenticar usuários e persistir dados em um banco de dados não-relacional.

O Firestore é muito poderoso recurso e possui outras funcionalidades, como sub-coleções e pesquisas avançadas. A base criada aqui permite a criação de aplicações elaboradas, aplicando conceitos como o `ViewModel` e o `Two-way data binding`.

A autenticação do usuário com o Firebase Authentication faz com que a aplicação tenha um robusto mecanismo de autenticação, com uma interface padronizada pelo Google, sem que o desenvolvedor tenha que criar ou se integrar em complexos mecanismos para isso.

O próximo capítulo demonstra como é possível monitorar o uso de funcionalidades pelos usuários e entender seu comportamento, através de envio de eventos para o Firebase.

# 12 - Entendendo o comportamento da aplicação e dos usuários com Firebase Analytics

Entender o comportamento dos usuários em um aplicativo para dispositivos móveis pode ser um importante fator para decisões de marketing, desenvolvimento de novas funcionalidades e criação de campanhas com promoções e divulgações de opções que podem trazer recursos financeiros para o desenvolvedor.

Além disso, entender como o aplicativo está se comportando, nos diversos dispositivos que está sendo executado, localidades e idiomas, também pode ser essencial para avaliar sua adoção num mercado mundial.

Quando o projeto foi criado no Firebase, foi ativado o Google Analytics. Somente com essa opção e adicionado o SDK do Firebase no projeto Android, já é possível obter uma série de informações como: localização dos usuários, número de usuários ativos, engajamento diário, relatório de *crashes*, relatório de retenção de usuários, modelos de dispositivos e versões de sistema operacional e outras informações do usuário e do dispositivo que estão executando a aplicação.

O Firebase possui eventos predeterminados que já são gerados automaticamente, e que possibilitam o entendimento de algumas características dos usuários e do funcionamento do aplicativo, como será visto na próxima seção. Porém, também é possível **gerar eventos customizados**, para entender o que os usuários estão fazendo dentro do aplicativo, por exemplo para verificar se uma funcionalidade está sendo utilizada ou não e com que frequência.

Essas informações podem ser utilizadas para definir campanhas com promoções ou divulgações de novas funcionalidades pagas ou disponíveis em assinaturas, através do Firebase Notifications.

Também é possível criar uma audiência com usuários específicos e alterar o comportamento do aplicativo através do Firebase Remote Config, como será visto no próximo capítulo.

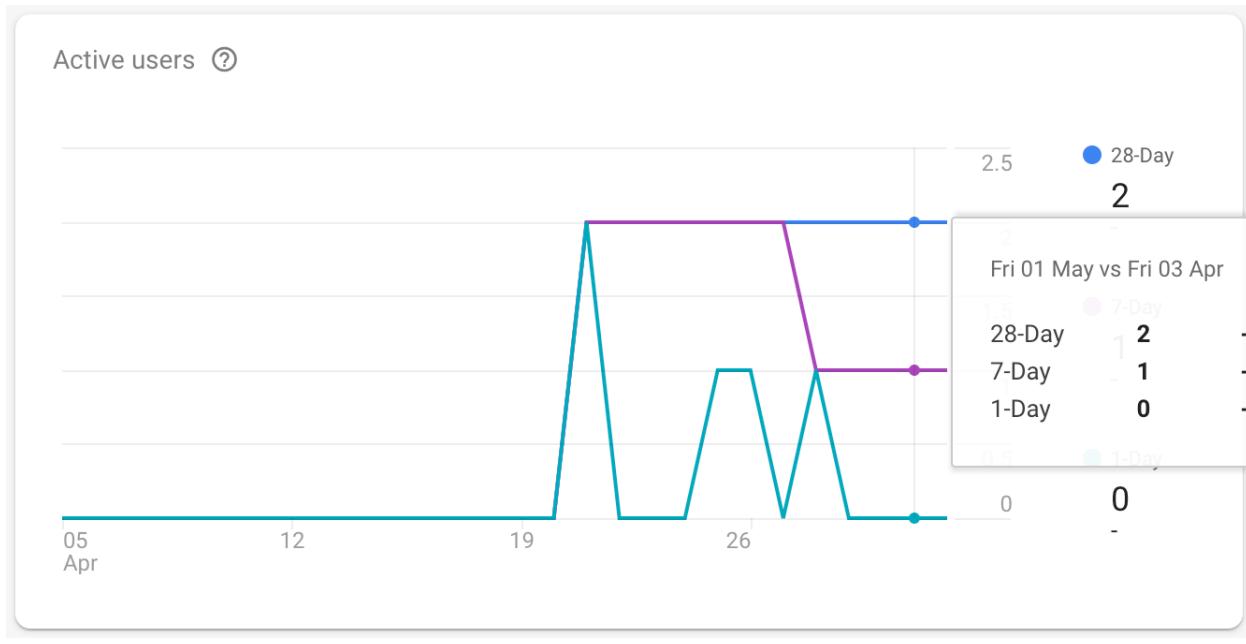
A ideia desse capítulo é utilizar o projeto `AndroidProject04`, adicionando os seguintes eventos:

- Seleção de um produto para ser visualizado;
- Intenção de criação de um novo produto;
- Exclusão de um produto, através de uma nova funcionalidade que será adicionada ao aplicativo.

Sobre esse último evento, será demonstrado como é possível utilizar o Firebase Analytics para analisar o comportamento do usuário diante de uma funcionalidade que pode não ter sido desenvolvida como ele esperava.

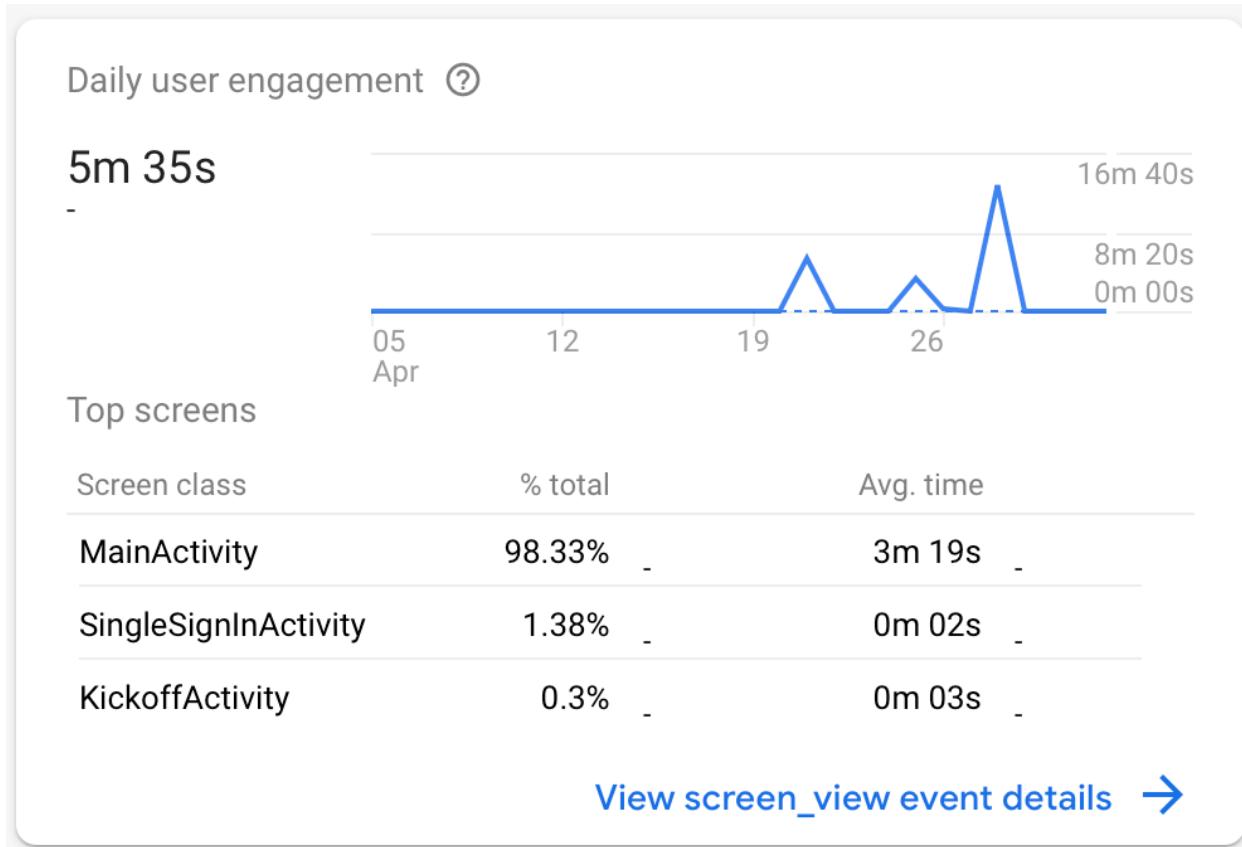
## 12.1 - Analisando o dashboard do Firebase Analytics

Como dito no início desse capítulo, apenas habilitando o Google Analytics no início da criação do projeto do Firebase e adicionando o SDK do Firebase ao projeto do Android, já é possível obter algumas informações sobre a utilização do aplicativo, como pode ser visto no console do Firebase, na seção Analytics, no menu Dashboard, como pode ser visto na figura a seguir:



Esse gráfico mostra os usuários ativos ao longo dos dias. Com ele é possível observar o quanto eles estão efetivamente utilizando o aplicativo e com que frequência.

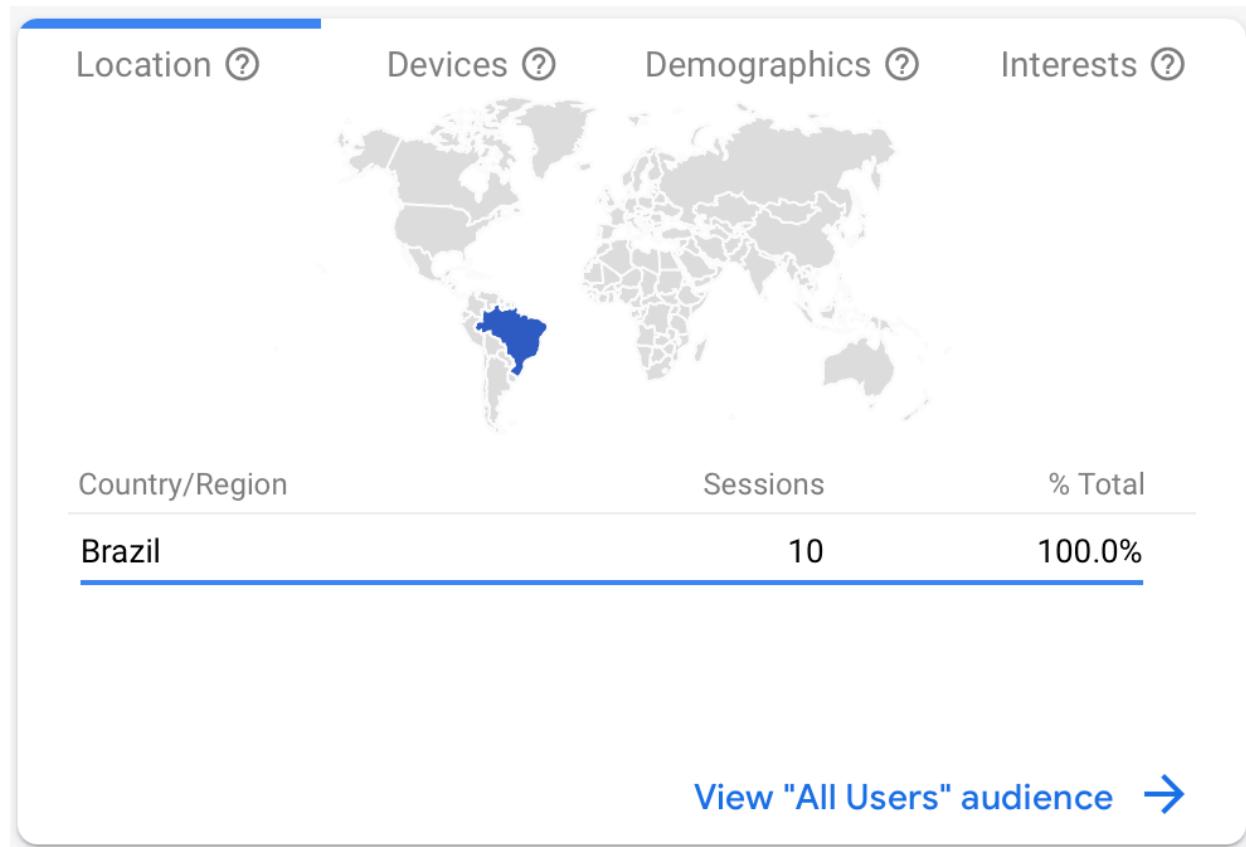
Também é possível ter uma estimativa mais detalhada do engajamento dos usuários, como pode ser visto na figura a seguir:



#### Engajamento de usuários

Obviamente, quanto mais usuários utilizarem o aplicativo, mais esses dados serão consistentes e informativos.

Ainda nesse dashboard, é possível observar a localização dos usuários, como pode ser visto na figura a seguir:



#### Localização dos usuários

Perceba que ainda existem outros gráficos nesse dashboard, com muito mais informação para poder entender dados como dispositivos sendo utilizados, informações de interesse sobre os usuários, versões de sistema operacional utilizado, etc.

## 12.2 - Habilitando o dispositivo para o modo debug do Firebase Analytics

Os eventos gerados pelo aplicativo, sejam eles customizados ou não, demoram a aparecer no dashboard de eventos no Firebase. Por esse motivo, é interessante habilitar o modo de depuração, que permite que os eventos seja, observados em uma *timeline* no console do Firebase. Para isso, abra o terminal do computador de desenvolvimento e execute o seguinte comando:

```
adb shell setprop debug.firebaseio.analytics.app br.com.siecola.android_project04
```

Caso o comando adb não seja reconhecido pelo sistema, navegue até o local onde ele está instalado. Esse local pode ser encontrado no Android Studio, no menu Tools -> SDK Manager, dentro do campo Android SDK Location. Dentro dessa pasta, vá até o local sdk/platform-tools/, que é onde o adb está instalado.

Lembre-se que o último parâmetro do comando citado anteriormente representa o pacote base da aplicação, por isso se ele estiver diferente, ajuste-o durante a execução do comando.

Com esse passo será possível monitorar os eventos que serão gerados pelo aplicativo, sem a necessidade de aguardar até que o Firebase os exiba em seu dashboard.

## 12.3 - Construindo a funcionalidade de apagar um produto

Como dito anteriormente, um dos eventos que será gerado com o Firebase Analytics é quando o usuário apagar um produto, mas para isso essa funcionalidade ainda tem que ser construída. E com um **propósito didático**, que será apresentado na próxima seção, a localização desse comando não ficará no local mais adequado, que seria quando o usuário fizesse a ação de clique logo sobre um produto. Ao invés disso, ele será colocado na tela de detalhes do produto, que não é um local de todo ruim, mas fica um pouco escondido e foge da ação esperada e mais comum do usuário, pois ele primeiro deve entrar em um produto para depois apagá-lo.



O intuito aqui é ser simples e direto no propósito de criar a função de apagar o produto. Outras questões de experiência do usuário serão deixadas de lado, como a confirmação que um produto foi excluído.

Para começar, crie um novo arquivo XML de menu, na pasta `\res\menu`, com o nome de `product_details_menu.xml`, como no trecho a seguir:

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

    <item
        android:id="@+id/delete_product"
        android:icon="@drawable/ic_delete_black_24dp"
        android:title="Delete product"
        app:showAsAction="always" />
</menu>
```

Aqui está sendo utilizado um novo recurso de imagem, com o nome `ic_delete_black_24dp`. Ele deve ser adicionado na pasta `res\drawable`, da mesma forma como foi feito com outros recursos do mesmo tipo em outras ocasiões.

Esse menu será “inflado” quando a tela de detalhes de produto for exibida. Ele aparecerá no canto superior da tela, na barra superior do aplicativo.

A exibição desse menu e o tratamento do evento de clique desse botão será feito no fragmento que exibe os detalhes do produto, por isso abra a classe `ProductDetailFragment` e comece transformando a variável `binding`, criada dentro da função `onCreateView`, em um atributo da classe, para que ela possa ser acessada de fora dessa função também:

```
private lateinit var binding: FragmentProductDetailBinding
```

Em seguida, adicione a instrução para informar o Android que esse fragmento agora possui um menu a ser exibido. Esse comando deve ser adicionado antes do retorno da função `onCreateView`:

```
setHasOptionsMenu(true)
```

Com essa instrução, é necessário implementar a função que realmente infla o menu, como no trecho a seguir:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.product_details_menu, menu)
}
```

E finalmente, quando o usuário clicar no botão para apagar o produto, a seguinte função será chamada:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.delete_product -> {
            binding.productDetailViewModel?.deleteProduct()
            findNavController().popBackStack()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

Veja que dentro dela, a função `deleteProduct` de `ProductDetailViewModel` está sendo invocada para apagar o produto. Como essa função ainda não existe, abra a classe `ProductDetailViewModel` e crie-a, como no trecho a seguir:

```
fun deleteProduct() {
    if (product.value?.id != null) {
        ProductRepository.deleteProduct(product.value!! .id!!)
        product.value = null
    }
}
```

Dessa forma, se algum produto existir dentro do contexto desse `ViewModel`, o repositório de produtos será invocado para excluí-lo.

Para testar essa funcionalidade, execute a aplicação novamente, clique em um dos produtos da lista e em seguida clique no ícone que foi adicionado nessa seção, que estará localizado no canto superior direito da tela. A aplicação voltará para a primeira tela, já com o produto apagado.

Essa ação também apaga o produto do banco de dados do Firestore.

## 12.4 - Gerando eventos com ações do usuário

Agora, imagine o seguinte cenário onde o desenvolvedor tem dúvidas sobre como a funcionalidade de apagar o produto deveria aparecer na tela. Seguindo os princípios de experiência do usuário, a opção para apagar um item em uma lista deveria estar disponível quando o usuário clicasse e segurasse sobre o item. Porém, essa opção requer um pouco mais de trabalho.

Como o desenvolvedor tem dúvidas de como deve implementar, imagine que ele decide colocar a opção no menu da barra superior, na tela de detalhes do produto, pois é mais fácil de ser criada, embora não esteja totalmente errado. Na verdade, ambas as opções deveriam estar disponíveis.

Visando saber se os usuários realmente ficam confusos e procurando pela funcionalidade de apagar um item da lista clicando e segurando-o, o desenvolvedor então decide gerar um evento com o Firebase Analytics. Dessa forma ele poderá entender o comportamento dos usuários. Porém, também é importante saber se os usuários estão conseguindo apagar produto, por isso o desenvolvedor também decide colocar um outro evento quando o botão no menu na tela de detalhes é clicado, evidenciando que o usuário encontrou a opção dentro do aplicativo.

Resumindo, os eventos que devem ser gerados são:

- Seleção de um produto para ser visualizado;
- Intenção de criação de um novo produto;
- Exclusão de um produto, gerado na tela de detalhes;
- Tentativa de exclusão através do clique longo na lista de produtos.

O último evento tem por objetivo, como dito anteriormente, observar se os usuários da aplicação estão confusos sobre como um produto pode ser apagado. Esse tipo de análise é extremamente importante em um aplicativo, pois pode mostrar que uma funcionalidade não foi desenvolvida

corretamente, seja pela experiência que a interface gráfica proporciona ao usuário, seja pela clareza como as informações são exibidas.

Da mesma forma, também é importante saber se uma funcionalidade está realmente sendo utilizada, pois tempo e recursos foram gastos em sua construção e testes. Essa é uma métrica que gestores e analistas de negócio podem utilizar para definir escopos melhores no futuro.

Logo, para gerar o primeiro evento, abra a classe `ProductAdapter` e acrescente o seguinte atributo:

```
private lateinit var firebaseAnalytics: FirebaseAnalytics
```

E para inicializá-lo, acrescente a seguinte instrução dentro da função `onCreateViewHolder`:

```
firebaseAnalytics = FirebaseAnalytics.getInstance(parent.context)
```

Com isso será possível utilizar a instância do Firebase Analytics dentro dessa classe.

A geração do evento em si deve ser feita dentro da função `onBindViewHolder`, no tratamento do clique de um item, como pode ser visto no trecho a seguir:

```
holder.itemView.setOnClickListener {
    val bundle = Bundle()
    bundle.putString(FirebaseAnalytics.Param.ITEM_ID, product.code)
    firebaseAnalytics.logEvent(FirebaseAnalytics.Event.SELECT_ITEM, bundle)

    onProductClickListener.onClick(product)
}
```

Veja que o evento padrão `FirebaseAnalytics.Event.SELECT_ITEM` está sendo utilizado para informar que um item foi selecionado. Como parâmetro, o código do produto está sendo informado. Na verdade vários parâmetros poderiam ser passados em um mesmo evento. Eles possuem o objetivo de informar mais detalhes sobre o evento e devem ser utilizados de acordo com o propósito da geração do evento em si.

Esse simples trecho de código faz com que a aplicação gere um evento no Firebase Analytics, que será enviado a ele quando o dispositivo puder se conectar à Internet, ou seja, sua geração não será perdida se o dispositivo estiver offline.

Ainda nessa função, acrescente o seguinte trecho:

```
holder.itemView.setOnLongClickListener {
    val bundle = Bundle()
    bundle.putString(FirebaseAnalytics.Param.ITEM_ID, product.code)
    firebaseAnalytics.logEvent("attempt_delete_product", bundle)
    true
}
```

Ele será responsável por gerar o evento quando o usuário tentar apagar um produto, dando um clique longo sobre um produto. O intuito desse evento é evidenciar que usuários estão buscando essa opção de apagar o produto dando esse comando de clique longo sobre ele.

Repare que aqui um evento customizado é gerado, de nome `attempt_delete_product`, pois o Firebase Analytics não possui um evento específico para isso.

Para o evento de intenção de criar um novo produto, vá até a classe `ProductsListFragment` e localize o seguinte trecho dentro da função `onCreateView`:

```
binding.fab.setOnClickListener { view ->
    this.findNavController()
        .navigate(ProductsListFragmentDirections.actionShowProductDetail(null))
}
```

Esse é o trecho responsável por tratar o evento de clique no botão para adicionar um novo produto. Então, para gerar o evento relativo a essa ação, acrescente o seguinte trecho:

```
val firebaseAnalytics = FirebaseAnalytics.getInstance(this.context!!)
firebaseAnalytics.logEvent("new_item", null)
```

Como para esse evento não há nenhuma informação adicionar a ser colocada como parâmetro, somente o nome do evento customizado em si é passado.

Por último, para o evento de exclusão de um produto, abra a classe `ProductDetailFragment` e transforme a variável `productCode`, utilizada na função `onCreateView` em um atributo privado da classe, como no trecho a seguir:

```
private var productCode: String? = null
```

Agora, localize o tratamento do evento de clique no botão de apagar o produto, localizado na função `onOptionsItemSelected`. Altere-a para contemplar a geração do evento de exclusão do produto em questão:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.delete_product -> {
            binding.productDetailViewModel?.deleteProduct()

            val firebaseAnalytics = FirebaseAnalytics.getInstance(this.context!!)
            val bundle = Bundle()
            bundle.putString(FirebaseAnalytics.Param.ITEM_ID, productCode)
            firebaseAnalytics.logEvent("delete_item", bundle)

            findNavController().popBackStack()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

Novamente um evento customizado é passado, informando o código do produto que foi selecionado para ser excluído.

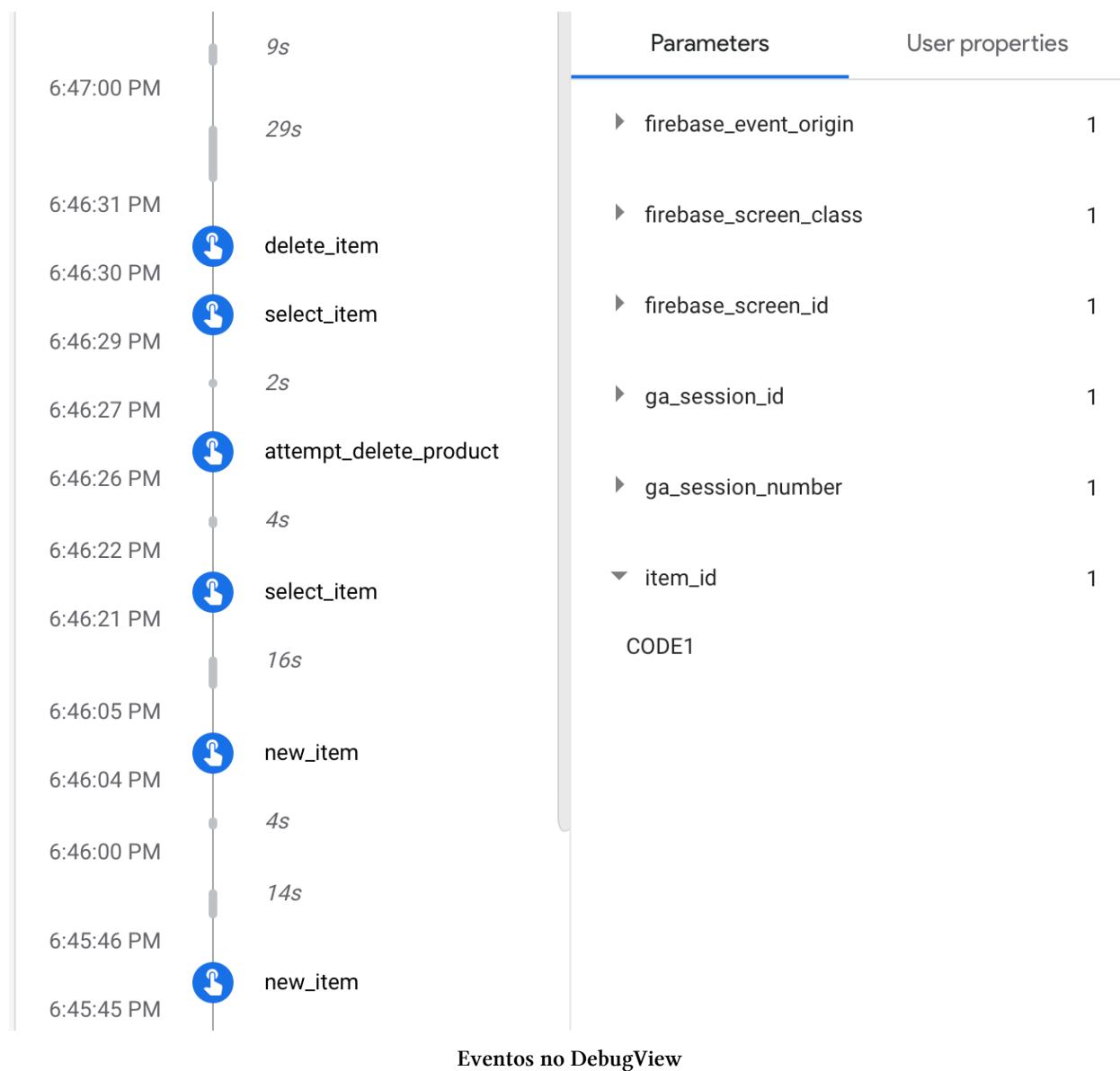
A próxima seção irá mostrar como esses eventos podem ser visualizados no Firebase Analytics.

## 12.5 - Visualizando os eventos no Firebase Analytics

Para visualizar os eventos que foram criados na seção anterior, vá até o console do Firebase, na seção Analytics e accesse a opção DebugView. Nesse momento, execute a aplicação em um emulador ou dispositivo real.

Execute as operações que geram eventos, como criação de um novo produto, visualização de um existente e exclusão.

Veja na figura a seguir um exemplo de como o *timeline* de eventos pode aparecer:

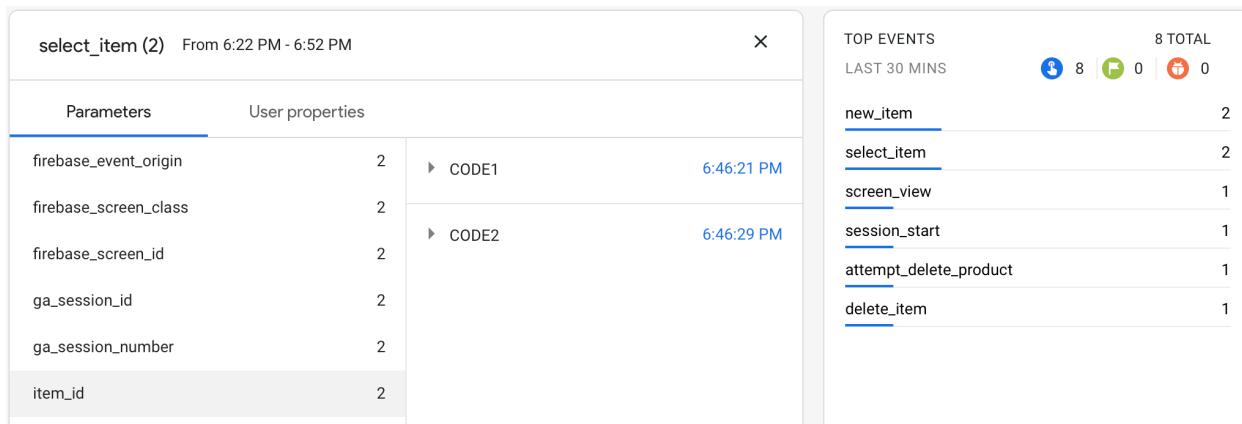


Nesse exemplo os seguintes eventos foram gerados:

- Dois produtos foram criados;
- Um foi selecionado para ser exibido;
- Houve uma tentativa de exclusão de um produto com a ação de clique longo em um item;
- E um produto foi apagado.

Também é possível ver na figura anterior, os detalhes de um dos eventos. Veja que o parâmetro `item_id` traz o código do produto em questão, além de outras informações adicionais.

À esquerda dessa tela é possível observar um relatório dos eventos que foram capturados durante a janela em que o DebugView está aberta, como pode ser visto na figura a seguir:



### Relatório de eventos

Selecionando um desses eventos do relatório, é possível visualizar seus detalhes aglutinados, como por exemplo o de seleção de um item, que traz quais foram selecionados pelo usuário.

Os eventos também aparecem no menu **Events**, na seção **Analytics** do console do Firebase, após algumas horas em que eles foram gerados:

Existing events					
Event name ↑	Count	% change	Users	% change	
app_remove	1	-	1	-	
attempt_delete_product	1	-	1	-	
delete_item	1	-	1	-	
first_open	2	-	2	-	
new_item	2	-	1	-	
screen_view	117	-	2	-	
select_item	15	-	1	-	
session_start	16	-	2	-	

### Dashboard de eventos

Com essa funcionalidade do Firebase, é possível gerar eventos facilmente em uma aplicação Android, de forma tão simples como se estivesse gerando mensagens de log. E com o SDK do Firebase, os eventos gerados quando o dispositivo está offline são armazenados e enviados assim que ele se conecta na Internet, não deixando nenhum evento importante ser perdido.

## 12.6 - Conclusão

Esse capítulo introduziu o conceito de eventos gerados com o Firebase Analytics, uma forma simples de gerar informação valiosa para a análise do funcionamento de um aplicativo, bem como o comportamento de seus usuários.

Além disso, também é possível obter informações importantes como localização dos usuários, versões do sistema Android e modelos dos dispositivos.

Esses eventos também podem ser utilizados para criação de campanhas para divulgação de ofertas e novas funcionalidades.

O próximo capítulo introduz o conceito de configuração remota, em que o comportamento do aplicativo pode ser alterado sem que haja a necessidade da publicação de uma nova versão.

# **13 - Alterando o comportamento da aplicação com o Firebase Remote Config**

Com o Firebase Remote Config é possível habilitar uma funcionalidade para um pequeno grupo de pessoas, antes de liberá-la para toda a base de usuários. Isso é uma ótima ferramenta para verificar se a funcionalidade foi bem desenvolvida e está sendo bem aceita pelos usuários que a estão utilizando.

Também é possível construir aplicativos dinâmicos e adaptáveis aos usuários, baseados em suas necessidades ou críticas.

O Firebase Remote Config é um mecanismo de armazenamento simples baseado em chaves e valores que ficam armazenados na nuvem, ou seja, para alterá-los basta acessar o seu console no Firebase. Com isso é possível alterar o funcionamento de um aplicativo sem ter que lançar uma nova versão dele.

Isso significa que os aplicativos podem ler essas configurações do Remote Config e adaptar-se de acordo com a informação que está armazenada lá.

Em conjunto com a segmentação de usuários do aplicativo, através da criação de audiências baseadas em propriedades ou comportamentos desse usuários, é possível criar um mecanismo de testes A/B com o Firebase Remote Config, ajustando variáveis para os diferentes grupos ou audiências.

Para utilizar o Remote Config é necessário adicionar uma biblioteca específica do Firebase, que é responsável por estabelecer a comunicação com ele e verificar as configurações desejadas pelo aplicativo.

A ideia desse capítulo é fazer com que a funcionalidade de exclusão do produto, criada no capítulo anterior, possa ser habilitada ou não, através de uma configuração que será criada no Firebase Remote Config. Também poderia escolher onde essa opção poderia estar: na tela de detalhes ou na listagem do produto. Dessa forma, com o uso do Firebase Analytics e os eventos gerados pelo aplicativo, seria possível determinar qual a opção que mais foi utilizada pelos usuários.

## **13.1 - Adicionando a biblioteca do Firebase Remote Config**

Para utilizar o Firebase Remote Config no projeto `AndroidProject04` basta adicionar a seguinte biblioteca no arquivo `build.gradle` do app, na seção `dependencies`:

```
implementation 'com.google.firebaseio:firebase-config-ktx:19.1.4'
```

Com isso já será possível ler os parâmetros configurados no projeto do Firebase em que a aplicação está associada, como será visto na próxima seção.

## 13.2 - Lendo a configuração do Firebase Remote Config

Como dito anteriormente, é possível criar parâmetros de configuração no Firebase Remote Config, que podem ser lidos através de chaves que os identificam. Esses parâmetros devem ser configurados no console do Firebase, como será mostrado mais adiante nesse capítulo.

Para a leitura desses parâmetros, é necessário realizar algumas configurações, como:

- Intervalo mínimo de consultas que a aplicação deve fazer para buscar os valores das configurações;
- Valores padrões para cada parâmetro de configuração desejado.

Feito essas configurações, deve-se comandar a instância do Firebase Remote Config na aplicação para que busque os parâmetros com o intervalo configurado. Dessa forma, em qualquer ponto da aplicação é possível resgatar a instância do Firebase Remote Config para ler os parâmetros de configuração.

Essa configuração inicial deve ser feita logo que a aplicação é iniciada, por isso abra a classe `MainActivity` e crie uma nova função chamada `setFirebaseRemoteConfig`, como no trecho a seguir:

```
fun setFirebaseRemoteConfig() {  
    val remoteConfig = Firebase.remoteConfig  
    val configSettings = remoteConfigSettings {  
        minimumFetchIntervalInSeconds = 60  
    }  
    remoteConfig.setConfigSettingsAsync(configSettings)  
  
    val defaultConfigMap: MutableMap<String, Any> = HashMap()  
    defaultConfigMap["delete_detail_view"] = true  
    defaultConfigMap["delete_list_view"] = false  
  
    remoteConfig.setDefaultsAsync(defaultConfigMap)  
  
    remoteConfig.fetchAndActivate()  
        .addOnCompleteListener(this) { task ->  
            if (task.isSuccessful) {  
                val updated = task.result
```

```
        Log.d("MainActivity", "Remote config updated: $updated")
    } else {
        Log.d("MainActivity", "Failed to load remote config")
    }
}
}
```

Veja que a primeira metade do código dessa função é responsável por criar as configurações citadas no início dessa seção. Aqui o valor de 60 segundos está sendo utilizado apenas como teste, pois é interessante que ele fique em torno de algumas horas, para evitar que a aplicação fique consultando o Firebase a todo o momento, o que não faz sentido, visto que essas configurações não devem mudar a todo instante.

Ainda nesse primeiro trecho da função, os valores padrões foram criados, para o caso da consulta ao Firebase ainda não tiver sido concluída e eles precisarem ser utilizados:

```
val defaultConfigMap: MutableMap<String, Any> = HashMap()
defaultConfigMap["delete_detail_view"] = true
defaultConfigMap["delete_list_view"] = false
```

O valor `delete_detail_view` é o que realmente será utilizado nesse exemplo. Ele servirá para definir se o botão de apagar um produto deve aparecer na tela de detalhes do mesmo. Veja que o valor padrão está como verdadeiro, o que significa que ele irá aparecer, mesmo que a consulta ao Firebase ainda não tenha sido concluída, como por exemplo se o dispositivo estiver sem conexão com a Internet.

A segunda metade da função efetivamente realiza a consulta ao Firebase Remote Config, buscando pelos parâmetros existentes. Lembre-se que esses parâmetros devem ser criados no Firebase, como será explicado mais adiante nesse capítulo.

Veja que a consulta ao Firebase é assíncrona e o `listener` criado aqui apenas imprime uma mensagem de log com sucesso ou falha da consulta.



A leitura efetiva dos valores de configuração será mostrada na próxima seção.

A função `setFirebaseRemoteConfig` deve ser chamada em dois pontos na classe `MainActivity`. A primeira delas é dentro da função `onCreate`, logo após a instrução `setContentView(R.layout.activity_main)`.

O segundo ponto é dentro da função `onActivityResult`, também logo após instrução `setContentView(R.layout.activity_main)`.

Com essas duas alterações, será possível invocar a função para configurar a instância do Firebase Remote Config e comandar a busca pelos parâmetros de configuração. Somente isso é necessário para que a aplicação esteja sempre atualizada com as últimas configurações a serem realizadas no console do Firebase.

## 13.3 - Mudando o comportamento da aplicação com o parâmetro do Firebase Remote Config

Como dito anteriormente, o exemplo com a utilização de um parâmetro de configuração será bem simples: mudar o comportamento da aplicação, exibindo ou não a opção de apagar um produto na tela de detalhes.

Isso pode ser feito com uma configuração *booleana*, onde o valor verdadeiro poderia exibir a opção e o valor falso a esconderia.

Como a tela de detalhes possui apenas uma opção no menu, é mais simples apenas exibir ou não o menu inteiro, ou seja, basta passar true ou false para a instrução `setHasOptionsMenu` no final da função `onCreateView` da classe `ProductDetailFragment`.

Com o Firebase Remote Config, basta ler o parâmetro `delete_detail_view` e passar seu valor para a função `setHasOptionsMenu`, como pode ser visto no trecho a seguir que deve ser colocado no fim da função `onCreateView` da classe `ProductDetailFragment`:

```
val remoteConfig = Firebase.remoteConfig  
setHasOptionsMenu(remoteConfig.getBoolean("delete_detail_view"))
```

Tendo a instância do Firebase Remote Config, basta ler o valor do parâmetro, passando sua chave. O valor retornado será aquele configurado no console do Firebase e será atualizado pela aplicação em até 60 segundos, como foi configurado na seção anterior.

Agora o parâmetro deve ser criado no console do Firebase, como será explicado na seção adiante.

## 13.4 - Criando o parâmetro no console do Firebase Remote Config

Para criar o parâmetro de configuração, vá até o console do Firebase, na seção `Grow` e clique na opção `Remote Config`. A seguinte tela deve aparecer, caso nenhum parâmetro tenha sido criado ainda:

Add a parameter

Parameter key <small>(?)</small>	Default value	Add value for condition ▾
Example: <code>holiday_promo_enabled</code>	(empty string)	Other empty values ▾ { }
<small>Add description</small>		
<small>Add parameter</small>		

Tela inicial do Firebase Remote Config

Para começar a criação do parâmetro de configuração, digite o valor `delete_detail_view` no campo `Parameter key` e configure a opção `Default value` para `true`.

Veja que há a opção `Add value for condition`, localizada na parte superior desse menu. Isso pode ser utilizado para criar um valor para essa configuração baseado em audiências de usuários, versões do aplicativo, regiões ou países e até um percentual aleatório dos usuários.

Para exercitar a criação de uma condição, escolha na opção `Define new condition` e crie uma nova condição para a região do Brasil, como mostra a figura a seguir:

**Define a new condition**

Use conditions to provide different parameter values if a condition is met

Name	Color
Country_Region	Orange

Applies if...

Country/Region	is in	Brazil	and
----------------	-------	--------	-----

Cancel      Create condition

### Criando a condição do parâmetro de configuração

Em seguida, clique em `Create condition`. O parâmetro criado deve ficar como o da figura seguir:

#### Add a parameter

Parameter key	Value for <code>Country_Region</code>	Add value for condition
<code>delete_detail_view</code>	false	{ } X

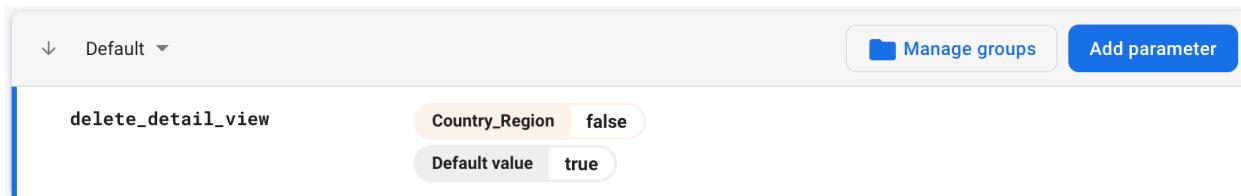
Add description

Default value	true	{ }
---------------	------	-----

Add parameter

### Finalizando a criação do parâmetro de configuração

Para finalizar, clique no botão `Add parameter`. Isso fará com que o parâmetro seja criado, como mostra a figura a seguir:



Parâmetro de configuração criado

Para que essa configuração passe a valer nas condições em que o parâmetro foi criado, clique no botão Publish changes, localizado no canto superior da página.

Isso fará que esse parâmetro fique disponível para todas as instâncias da aplicação e aqueles que estiverem na região do Brasil, terão seu valor configurado para `false`, o que deve fazer a opção de apagar o produto na tela de detalhes desaparecer.

Faça testes alterando esse valor da condição `Country_Region`, aguarde em torno de 60 segundos e execute a aplicação novamente. Se ele buscou uma nova configuração, a seguinte mensagem deverá aparecer no Logcat:

```
Remote config updated: true
```

Isso significa que o que foi publicado no Firebase Remote Config foi atualizado no dispositivo, fazendo com que a aplicação obedeça a configuração criada no console do Firebase.

## 13.5 - Conclusão

Esse capítulo apresentou uma opção simples e direta para a configuração remota do aplicativo, baseado em condições como região, versão, plataforma. Com o Firebase Remote Config é possível liberar novas funcionalidades para pequenos grupos de usuários, sem que uma nova versão tenha que ser lançada.