

Desenvolvimento iOS com Firebase

Paulo Cesar Siecola

Desenvolvimento iOS com Firebase

Paulo Cesar Siecola

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

Essa versão foi publicada em 2020-12-19



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.

© 2020 Paulo Cesar Siecola

Para Matilde!

Conteúdo

1 - Introdução	1
1.1 - A quem se destina esse livro	1
1.2 - O que é necessário para desenvolver aplicativos para iOS	1
1.3 - Estrutura didática do livro	2
1.4 - Capítulos do livro	2
2 - Como desenvolver aplicativos para o iOS	3
2.1 - iOS Simulators	3
2.2 - Xcode preview	5
2.3 - Conclusão	6
3 - Sobre o Google Firebase	7
3.1 - Firebase Cloud Messaging	7
3.2 - Firebase Authentication	7
3.3 - Firebase Analytics	7
3.4 - Firebase Remote Config	8
3.5 - Firebase Cloud Firestore	8
3.6 - Conclusão	8
4 - Conceitos básicos de Swift	10
4.1 - Sobre o Swift	10
4.2 - Documentação sobre Swift	10
4.3 - O Playground do Xcode	11
4.4 - Hello World em Swift	11
4.5 - Criando o primeiro Playground no Xcode	11
4.6 - Declaração e tipos de variáveis	12
4.7 - Declarando constantes	13
4.8 - Trabalhando com String	14
4.9 - Declarando variáveis com valores opcionais	16
4.10 - Arrays e Dictionaries	20
4.11 - Controle de fluxo com if e switch	24
4.12 - Estruturas de repetição	27
4.13 - Funções	30
4.14 - Objetos e classes	33

CONTEÚDO

4.15 - Inicialização da classe	34
4.16 - Herança	37
4.17 - Protocol	37
4.18 - Deinitialization	38
4.19 - Automatic Reference Counting - ARC	39
4.20 - Enum	39
4.21 - Tratamento de erros	40
4.22 - Structures	44
4.23 - Closures	45
4.24 - Conclusão	48
5 - Hello World com SwiftUI	49
5.1 - Criando o projeto	51
5.2 - Entendendo a estrutura do projeto	52
5.3 - Construindo uma interface básica com SwiftUI	57
5.4 - Utilizando @State	59
5.5 - Criando o modelo de produtos	62
5.6 - Criando a tela de cadastro do produto	64
5.7 - Salvando o produto	66
5.8 - Exibindo o produto salvo	69
5.9 - Refatorando as views e utilizando @Binding	70
5.10 - Criando o campo para o preço do produto	74
5.11 - Conclusão	76
6 - Consumindo serviços REST com Alamofire	77
6.1 - O provedor de serviços de vendas	79
6.2 - Criando e preparando o projeto	88
6.3 - Criando as classes de modelo	90
6.4 - Criando o cliente REST com o Alamofire	94
6.5 - Exibindo a lista de produtos	103
6.6 - Conclusão	107
7 - Criando navegação com o NavigationView e consumindo outras operações do serviço REST	108
7.1 - Adicionando navegação ao aplicativo	108
7.2 - Criando um novo produto	112
7.3 - Excluindo um produto	120
7.4 - Visualizando e alterando os detalhes de um produto	124
7.5 - Conclusão	130
8 - Recebendo mensagens via Firebase Cloud Messaging	131
8.1 - O que é Firebase Cloud Messaging	131
8.2 - Arquitetura do projeto iOS com FCM	133
8.3 - Criando o novo projeto no Xcode	134

CONTEÚDO

8.4 - Adicionando o Firebase ao projeto criado	135
8.5 - Configurando o FCM para funcionar com o APNs	138
8.6 - Registrando no FCM para receber notificações	143
8.7 - Recebendo mensagens do FCM	147
8.8 - Conclusão	156
9 - Autenticando usuários com Firebase Authentication	157
9.1 - O que o Firebase Authentication	157
9.2 - Arquitetura do projeto	158
9.3 - Criando o novo projeto no Xcode	160
9.4 - Criando o novo projeto no Firebase	160
9.5 - Adicionando as bibliotecas do Firebase Authentication ao projeto	161
9.6 - Inicializando o Firebase	162
9.7 - Criando o viewModel de autenticação	163
9.8 - Configurando o Firebase Authentication	166
9.9 - Criando a tela de login	166
9.10 - Criando o fluxo de autenticação do app	167
9.11 - Conclusão	170
10 - Persistindo dados com o Firestore	171
10.1 - Entendendo a hierarquia de dados no Firestore	172
10.2 - Arquitetura do projeto	173
10.3 - Criando o modelo de produtos	175
10.4 - Especificando as regras de segurança do Firestore	176
10.5 - Construindo o índice de pesquisa no Firestore	177
10.6 - Criando o viewModel da lista de produtos	179
10.7 - Criando a view da lista de produtos	181
10.8 - Criando o viewModel de criação de produtos	183
10.9 - Construindo a view de criação de produtos	184
10.10 - Exibindo a tela de criação de produtos	187
10.11 - Testando a criação e listagem de produtos	188
10.12 - Excluindo produtos	188
10.13 - Criando o viewModel de edição de produtos	190
10.14 - Construindo a view de edição de produtos	191
10.15 - Testando a sincronização automática da lista de produtos	194
10.16 - Conclusão	195
11 - Entendendo o comportamento da aplicação e dos usuários com Firebase Analytics	196
11.1 - Analisando o dashboard do Firebase Analytics	197
11.2 - Preparando o projeto para o Firebase Analytics	199
11.3 - Gerando eventos do usuário	199
11.4 - Visualizando os eventos no Firebase Analytics	201
11.5 - Conclusão	203

CONTEÚDO

12 - Alterando o comportamento da aplicação com o Firebase Remote Config	204
12.1 - Preparando o app para o Firebase Remote Config	204
12.2 - Buscando as configurações do Remote Config	206
12.3 - Mudando o comportamento do app com o Firebase Remote Config	206
12.4 - Criando o parâmetro no console do Firebase Remote Config	207
12.5 - Conclusão	208
13 - Salvando fotos dos produtos no Firebase Storage	209
13.1 - Estrutura e organização do Firebase Storage	211
13.2 - Criando as regras de segurança de acesso	212
13.3 - Preparando o projeto	213
13.4 - Alterando o modelo de produtos	214
13.5 - Criando o view model para a lista de produtos	214
13.6 - Adaptando a lista de produtos para exibir sua imagem	216
13.7 - Criando a view para escolher a foto com o UIImagePickerController	222
13.8 - Salvando o produto com sua foto	225
13.9 - Alterando o produto com sua foto	230
13.10 - Conclusão	236

1 - Introdução

Bem vindo ao livro **Desenvolvimento iOS com Firebase!** Com ele, o leitor construirá aplicativos para iOS 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, regtos 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 iOS 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 para **iOS** utilizando serviços do **Google Firebase**.

Todos os aplicativos desenvolvidos aqui serão criados utilizando a linguagem **Swift**, criada pela Apple. 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 Swift serão apresentados ao longo do livro, principalmente aqueles mais utilizados para o desenvolvimento de aplicativos para iOS.

Apesar desse livro ser voltado a conceitos e arquiteturas não triviais no âmbito do desenvolvimento de aplicativos para iOS, 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 iOS

Será necessário apenas o Xcode para seguir esse livro. O Xcode é a IDE da Apple para desenvolvimento de aplicativos para iOS, OS X, tvOS e watchOS. Para maiores informações sobre ele, consulte esse [link¹](#)

¹<https://developer.apple.com/xcode>

Obviamente, para se trabalhar com o Xcode, é necessário ter um computador da Apple, rodando o OS X.

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

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

O **capítulo 2** oferece uma visão geral de como o processo de desenvolvimento de um app iOS acontece.

No **capítulo 3**, o leitor terá uma explicação inicial sobre os serviços do Firebase que serão utilizados nesse livro, para integrar nas aplicações a serem desenvolvidas para iOS.

O **capítulo 4** é uma introdução a alguns conceitos da linguagem **Swift**, principalmente aqueles que serão mais utilizados nos exemplos que serão desenvolvidos nesse livro.

No **capítulo 5** é criado o primeiro aplicativo, já utilizando **SwiftUI** e as facilidades do Xcode Preview. Também são introduzidos conceitos como **State** e **Binding**, que facilitam a atualização da interface gráfica através de eventos.

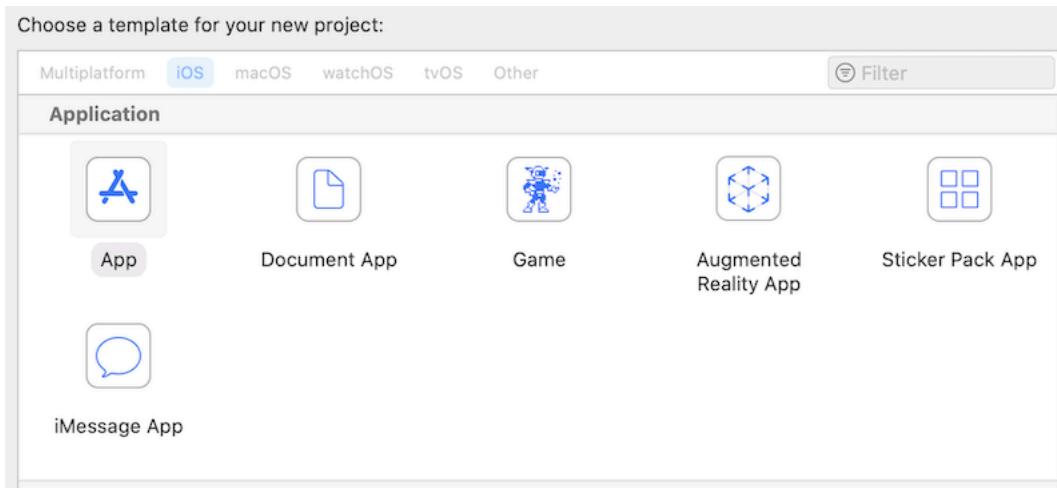
O **capítulo 6** inicia um novo projeto onde são apresentadas técnicas modernas para a criação de um aplicativo para consumir **serviços REST autenticação OAuth** utilizando o Alamofire. Tudo construído com **SwiftUI**, mostrando o que ele tem de melhor, com a utilização de **viewModels** para a representação dos dados a serem atualizados na interface gráfica. Esse mesmo projeto será continuado no **capítulo 7**, onde será introduzida a navegação entre **view** dentro do app, assim como o consumo de outras operações do serviço REST.

O **capítulo 8** apresenta como construir um app para se registrar e receber mensagens do **Firebase Cloud Messaging**, um serviço capaz de enviar notificações a dispositivos móveis do Google.

²<http://firebase.google.com/>

2 - Como desenvolver aplicativos para o iOS

O iOS é o sistema operacional proprietário, criado pela Apple, utilizado no iPhone. Nele é possível desenvolver aplicações para os usuários utilizando a linguagem Swift, através da ferramenta Xcode, como dito no capítulo anterior.



Na figura anterior é possível visualizar os tipos de aplicações que podem ser desenvolvidas para o iOS utilizando o Xcode.

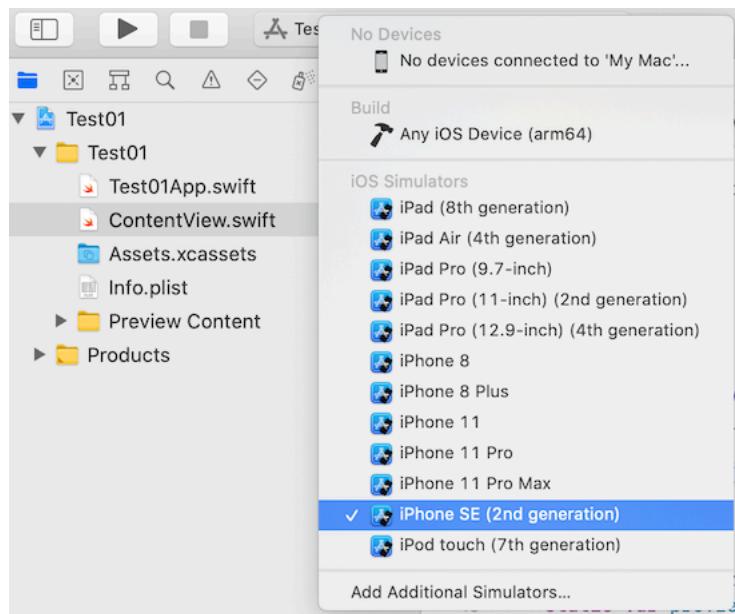
2.1 - iOS Simulators

Para testar as aplicações desenvolvidas para o iOS no Xcode, pode-se utilizar dispositivos reais conectados à máquina onde o Xcode está sendo executado, ou simuladores que já vêm instalados com ele, como pode ser observado na figura a seguir:



Simuladores de dispositivos do Xcode

A vantagem dos simuladores é que é possível testar em vários dispositivos diferentes:



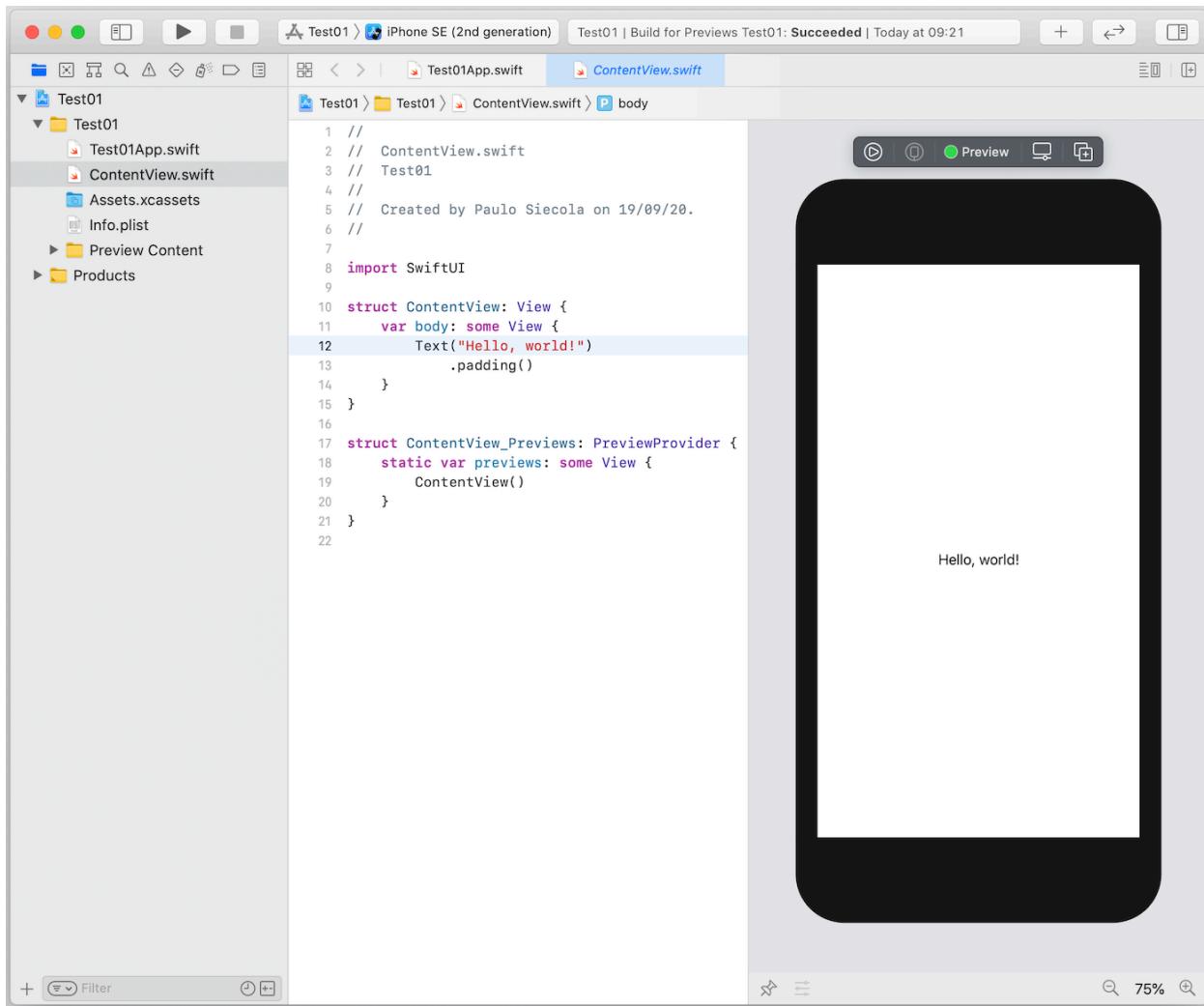
Tipos de dispositivos que podem ser simulados

Além disso, é possível executar mais de um simulador ao mesmo tempo, para verificar a compatibilidade de aplicativo em dispositivos diferentes.

Obviamente os simuladores consumem CPU e memória da máquina, o que em certas configurações inviabiliza seu uso, sendo necessário utilizar um dispositivo real conectado no Xcode.

2.2 - Xcode preview

Construindo o aplicativo utilizando SwiftUI, algo que será visto a partir do capítulo 5 desse livro, é possível visualizar a construção da interface gráfica através do Xcode preview, como pode ser visto na figura a seguir:



Xcode preview

Também é possível simular a execução do aplicativo de acordo com o dispositivo selecionado, acelerando o processo de desenvolvimento.

Essa técnica será largamente utilizada nos projetos que serão desenvolvidos ao longo desse livro.

2.3 - Conclusão

Como pode ser visto, o Xcode já traz tudo o que é necessário para se desenvolver aplicativos para iOS.

No próximo capítulo, será explicado um pouco sobre os serviços do Google que serão utilizados nesse livro para a construção de aplicativos conectados à nuvem, através dos serviços oferecidos pelo Firebase.

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 iOS com o Firebase é muito simples e pode ser feita através de bibliotecas oferecidas pela própria Google, além de alguns 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 iOS 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

Analizar 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 iOS 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 tentativas 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 iOS conectadas à nuvem.

O próximo capítulo irá apresentar alguns conceitos iniciais sobre a linguagem Swift, principalmente aqueles que serão mais utilizados ao longo desse livro.

4 - Conceitos básicos de Swift

Antes de começar a desenvolver uma aplicação para iOS, é interessante conhecer um pouco mais da linguagem da Apple: o Swift.

Neste capítulo, serão apresentadas noções básicas do Swift, principalmente aquelas que serão utilizados ao longo desse livro.

4.1 - Sobre o Swift

O Swift é a linguagem de programação *open source* da Apple que pode ser utilizada para criação de aplicativos para qualquer um dos seus sistemas operacionais, como iOS, iPadOS, macOS, watchOS e tvOS.

Essa linguagem possui muitas características inovadoras, mas também preza características como desempenho de execução e segurança. Esse último ficará evidente ao longo desse capítulo, principalmente no que se refere a definições de tipos de variáveis. Tudo isso ainda, desejando ser uma linguagem agradável para o programador, em sua sintaxe.

Um dos objetivos do Swift é tornar-se uma linguagem para substituir as que são baseadas em C, como C++ e Objective-C e o próprio C. Por isso o compromisso com desempenho para execução de muitas tarefas, em comparação com essas linguagens.

Para saber mais sobre Swift, consulte o site de referência³

4.2 - Documentação sobre Swift

No site citado na seção anterior, há uma excelente [documentação](#)⁴ sobre a linguagem Swift, incluindo as duas principais seções a seguir:

- [Tour](#)⁵ pelo Swift: essa seção apresenta os conceitos iniciais da linguagem, com exemplos simples de se entender;
- [Guia](#)⁶ da linguagem: aqui é possível entender mais profundamente os conceitos chaves da linguagem. Também pode ser utilizado como guia de referência.

Por se tratar de uma linguagem nova, de certa forma, é interessante que o leitor possua um guia de referência, principalmente no início do seu aprendizado.

³<https://swift.org/>

⁴<https://swift.org/documentation/>

⁵<https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html>

⁶<https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html>

4.3 - O Playground do Xcode

O Xcode possui uma ferramenta muito interessante, tanto para desenvolvedores experientes, quanto para novatos: o Playground. Com ele é possível:

- Testar trechos de códigos, sem a necessidade de criar um novo projeto somente para isso;
- Verificar o funcionamento de algoritmos dentro de um projeto;
- Visualizar comportamentos da interface gráfica, baseado em condições controladas;
- Visualizar valores de variáveis ao longo da execução do código;

Nesse capítulo, serão apresentados os conceitos iniciais da linguagem Swift utilizando o Xcode Playground.

4.4 - Hello World em Swift

Em Swift, um simples “Hello World” pode ser escrito da seguinte forma:

```
print("Hello World!!!")
```

Em Swift, isso pode representar um programa completo, pois não é necessário importar bibliotecas tão pouco possuir uma função de entrada, como o `main` em C.

Repare também que não é necessário colocar o ; no final de uma instrução, embora isso seja opcional. O ; só é obrigatório se for necessário colocar mais de uma instrução em uma mesma linha. Nesse caso, o ; deve ser colocado para indicar a finalização de cada instrução.

4.5 - Criando o primeiro Playground no Xcode

Para testar o primeiro programa, criado na seção anterior, no Xcode, basta abrir um novo Playground, sem a necessidade de criar um projeto.

Para isso, vá no menu do Xcode em `File -> New -> Playground`. Em seguida, escolha a opção `iOS -> Blank`, além do nome do seu novo Playground e o local onde ele deverá ser salvo.

Deverá ser criado um novo arquivo com o seguinte trecho de código:

```
import UIKit

var str = "Hello, playground"
```

Apesar da instrução `import UIKit` estar presente, ela não será necessária nos testes que serão feitos nesse capítulo, pois somente códigos simples serão tratados aqui.

4.6 - Declaração e tipos de variáveis

A instrução `var str = "Hello, playground"`, que apareceu no código de exemplo criado pelo Xcode, é uma declaração de uma variável do tipo `String`.

Repare que foi utilizada a instrução `var` para fazer isso, mas nenhum tipo foi definido para a variável. Então como é possível saber que a variável `str` é do tipo `String`? Na verdade o compilador infere o tipo da variável em casos como esse, quando um valor já é atribuído a ela na declaração. Dessa forma, é possível se fazer a declaração de uma variável de duas formas:

- Declaração com tipo implícito: quando o compilador infere o tipo da variável pelo valor inicial atribuído a ela no momento da declaração;
- Declaração com tipo explícito: quando a instrução da declaração da variável informa o tipo da variável.

Declaração de variável tipo implícito

Veja alguns exemplos de declarações implícitas:

```
var idade = 16
var peso = 12.5
var nome = "Matilde"
```

Tipos de variáveis

No exemplo anterior, a variável `idade` é do tipo inteiro, `peso` é do tipo ponto-flutuante e `nome` é do tipo `String`. Os tipos básicos de variáveis existentes em Swift são:

- `Int`: para números inteiros;
- `Double` e `Float`: para números decimais;
- `Bool`: para valores booleanos;
- `String`: para textos.

Ainda existem tipos de variáveis para coleções, que serão vistos mais adiante.

Declaração de variável com tipo explícito

Para declarar uma variável, já informando seu tipo, basta fazer da seguinte forma:

```
var idade: Int
```

Também é possível declarar a variável de forma explícita e, ao mesmo tempo, atribuir um valor inicial a ela:

```
var idade: Int = 11
```

Nada impede que seja feita uma declaração de uma variável com o tipo explícito e atribua um valor inicial que possa dar a entender que ela é de outro tipo, como no exemplo a seguir:

```
var peso: Double = 12
```

Nesse caso, a variável peso é do tipo Double, apesar do valor inicial sugerir que poderia ser um inteiro.

Erros mais comuns na declaração de variáveis

Declarar variáveis sem um valor inicial, para que o compilador possa inferir seu tipo, ou não colocar o tipo explicitamente, gera um erro do compilador, como no exemplo a seguir:

```
var idade
```

Da mesma forma, declarar uma variável com o seu tipo e atribuir um valor incoerente a esse tipo, também gera um erro:

```
var idade: Int = "Matilde"
```

Obviamente, declarar a variável com um tipo e depois atribuir um valor incoerente a esse tipo, também causa erro:

```
var idade: Int
```

```
idade = "Matilde"
```

Acontece o mesmo se a declaração for com o tipo implícito:

```
var idade = 16
```

```
idade = "Matilde"
```

4.7 - Declarando constantes

Na seção anterior foi demonstrado como fazer para declarar variáveis, com a instrução var.

Para declarar constantes, basta utilizar a instrução let, como no exemplo a seguir, em uma declaração com o tipo implícito:

```
let idade = 16
```

E com o tipo explícito:

```
let idade: Int = 16
```

As regras com os tipos, para as variáveis, são as mesmas para as constantes.

Como o próprio nome diz, constantes não podem ter seu valor modificados ao longo da execução do código, mas podem ser inicializadas após a sua declaração, somente uma vez:

```
let idade: Int
```

```
idade = 16
```

```
print (idade)
```

O trecho de código mostrado iria imprimir o valor 16.

Porém, isso não seria permitido de se fazer:

```
let idade:Int
```

```
idade = 16
```

```
print (idade)
```

```
idade = 6
```

```
print (idade)
```

A instrução `idade = 6` seria considerada um erro, informando que `idade` deveria ser alterado para `var`, ao invés de `let`, para que seu valor pudesse ser alterado.



É possível utilizar `let` para se trabalhar com constantes dentro de testes condicionais, como será visto mais adiante.

4.8 - Trabalhando com String

Declarar variáveis ou constantes do tipo `String` em Swift é tão simples quanto os demais tipos. Por exemplo, para declarar uma variável `String`, de forma implícita, basta fazer:

```
var nome = "Matilde"
```

E de forma explícita:

```
var sobrenome: String = "Siécola"
```

Até aí, não há nada de novo. O mesmo pode ser feito para constantes, com a instrução let:

```
let nome = "Matilde"
```

E de forma explícita:

```
let sobrenome: String = "Siécola"
```

Mas como fazer para concatenar duas Strings? Poderia ser feito assim:

```
var nomeCompleto = nome + " " + sobrenome
```

Como as variáveis nome e sobrenome são do tipo String, então o compilador entende, implicitamente, que a nova variável nomeCompleto também é do tipo String.

E se concatenação tiver que ser feita com variáveis de outros tipos, como inteiros? Aí mora um perigo, pois em Swift as variáveis nunca são convertidas implicitamente. De qualquer tipo, para qualquer outro tipo, deve-se fazer isso de forma explícita. Veja no exemplo a seguir, quando um inteiro é concatenado em uma String:

```
var nome = "Matilde"

var sobrenome:String = "Siécola"

var idade:Int = 16

var nomeCompleto = "Nome: " + nome + " " + sobrenome +
" - Idade: " + String(idade) + " anos"
```

Especificamente no caso de conversões de tipos para String, quando deseja-se concatená-los, como feito anteriormente, é possível adotar uma notação mais simples. Veja como ficaria:

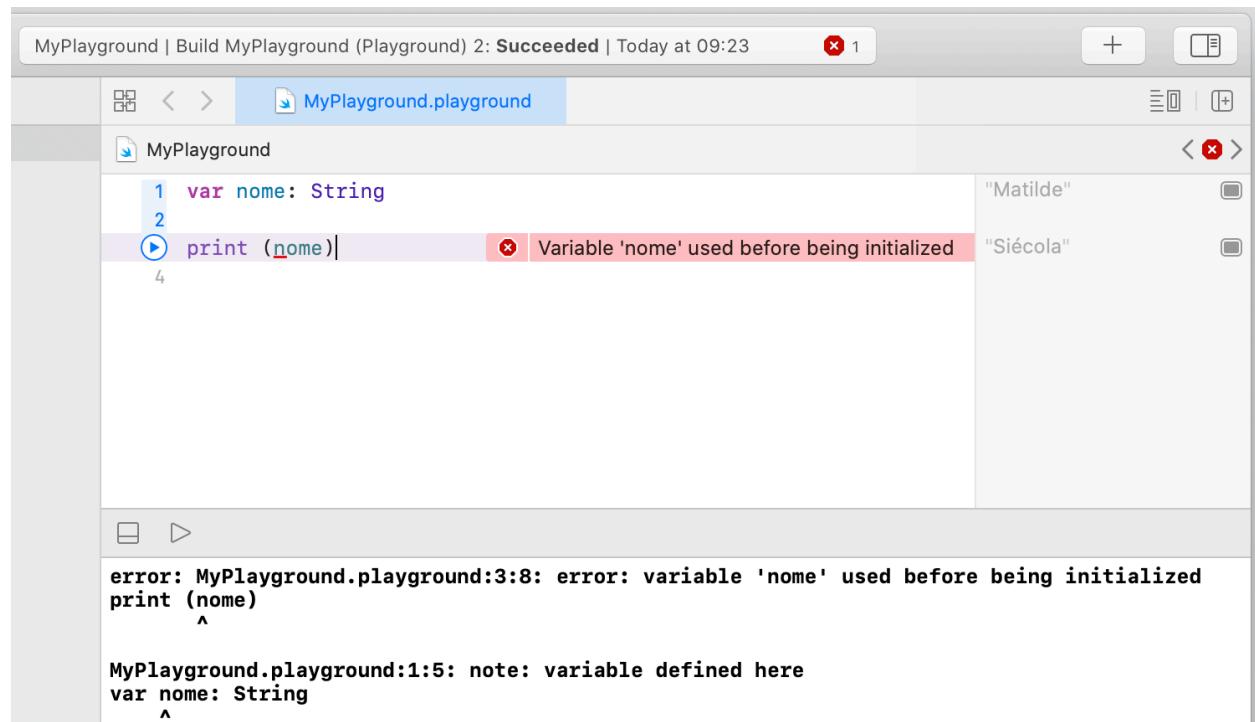
```
var nomeCompleto = "Nome: \(nome) \(sobrenome) - Idade: \(idade) anos"
```

4.9 - Declarando variáveis com valores opcionais

A linguagem Swift possui uma forma de tratar variáveis para evitar problemas com aquelas que não foram inicializadas. É o conceito de **Optional** ou opcional, traduzindo.

Há situações que serão vistas mais adiante nesse livro, que fazem com que uma variável tenha que ter, obrigatoriamente, um valor atribuído a ela. Mas o que fazer, nesse caso, quando o valor não pode ser atribuído no momento da declaração da variável? É aí que entra o uso dos valores opcionais.

Veja um exemplo, em que o Xcode reclama do uso de uma variável que não foi inicializada:



The screenshot shows a Xcode playground window titled "MyPlayground". In the code editor, there is a file named "MyPlayground.playground". The code contains the following:

```

1 var nome: String
2
3 print(nome)
4

```

A red error highlight covers the line "print(nome)". A tooltip next to the error message says "Variable 'nome' used before being initialized". Below the code editor, the output pane shows the error message:

```

error: MyPlayground.playground:3:8: error: variable 'nome' used before being initialized
print ^

MyPlayground.playground:1:5: note: variable defined here
var nome: String ^

```

Variável sem valor inicial

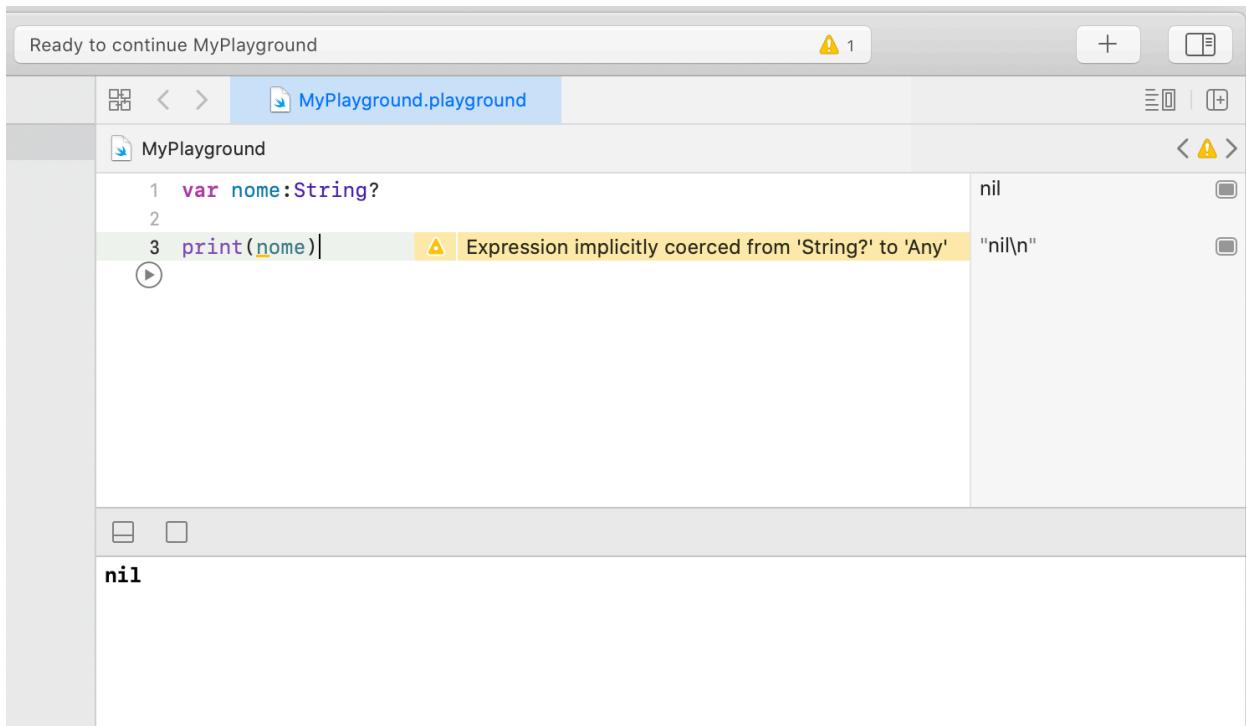
É claro que esse é só um exemplo, mas há diversos casos onde uma variável deve ser declarada, sem que haja um valor inicial para ela. E ainda, há possibilidade dela não assumir nenhum valor.

Esse fato se resolve utilizando o operador ? no momento da declaração da variável. Veja:

```
var nome: String?
```

```
print(nome)
```

Isso diz ao compilador que o valor da variável `nome` é opcional e que, inclusive, pode ser vazio ou nulo. Veja que o compilador agora não reclama:



The screenshot shows a Xcode playground window titled "MyPlayground.playground". In the code editor, there is one line of code:

```
1 var nome:String?  
2  
3 print(nome)|
```

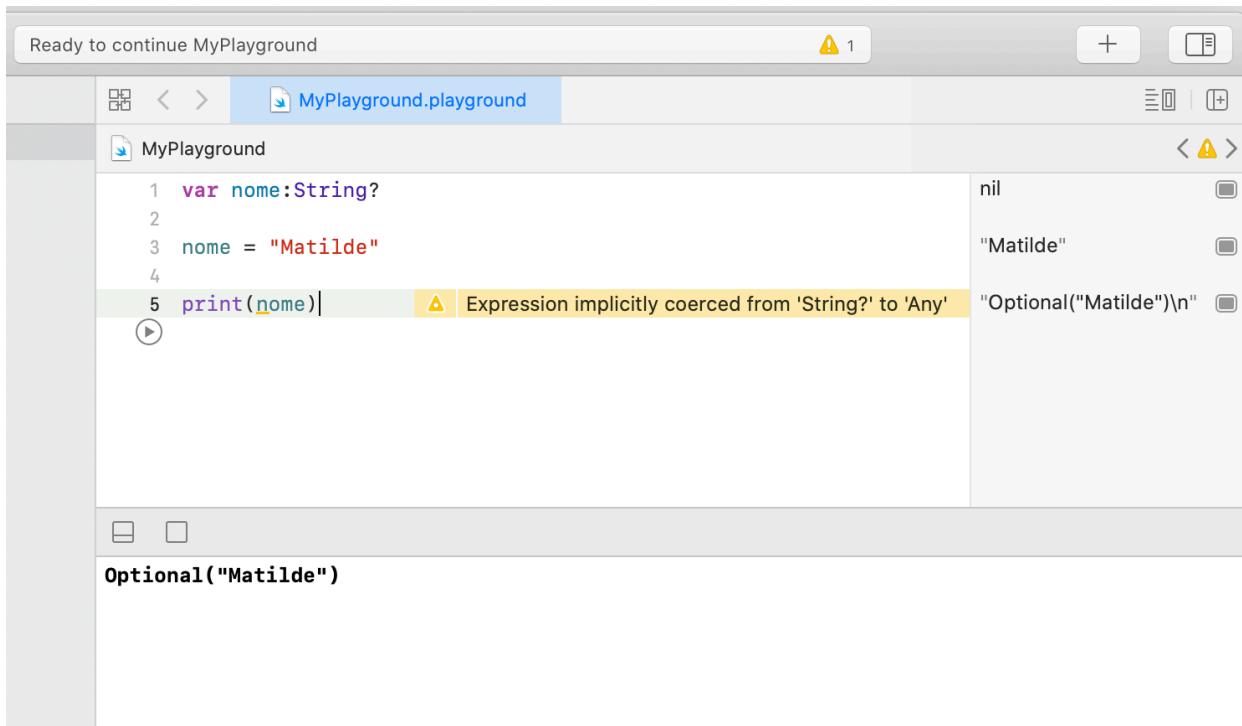
A yellow warning icon appears next to the word "print", indicating a warning: "Expression implicitly coerced from 'String?' to 'Any'". To the right of the code editor, the output pane shows the result of the print statement: "nil\n".

Declarando variável com valor opcional



No Swift, o valor nulo é representado por `nil`.

Obviamente, se a variável `nome` tiver um valor atribuído a ela, a instrução `print` irá mostrá-lo da seguinte forma:

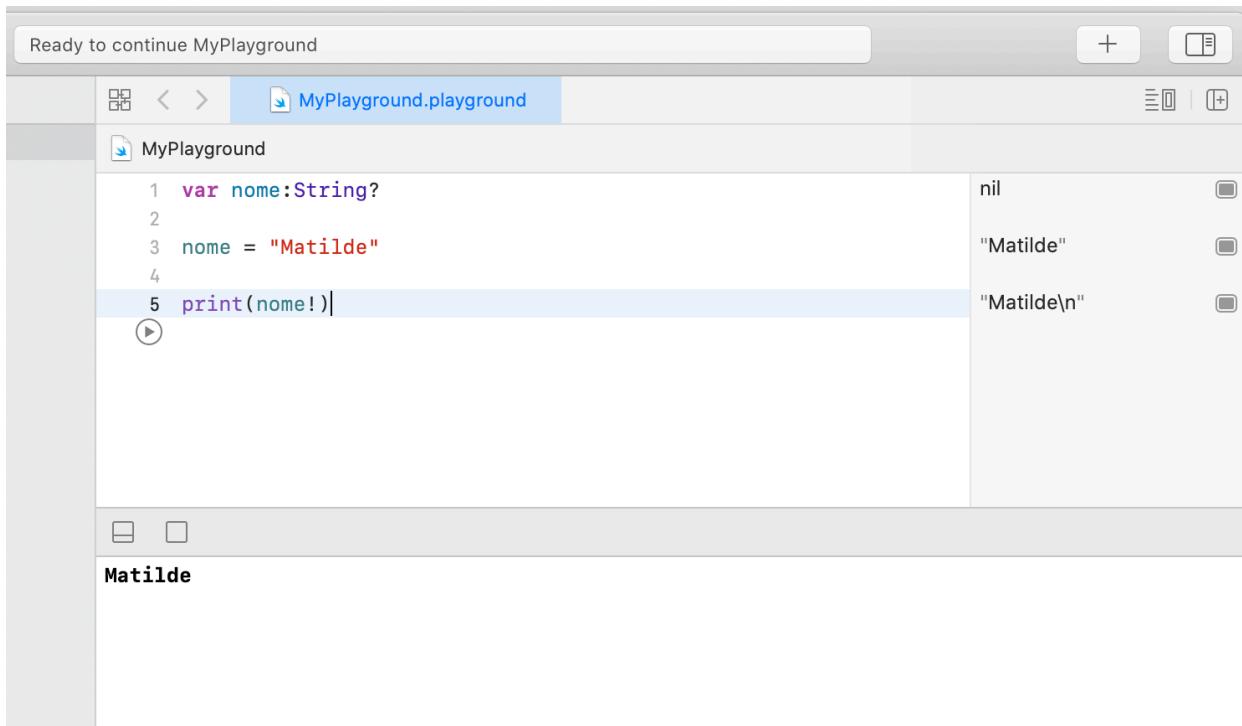


The screenshot shows a Xcode playground window titled "MyPlayground.playground". In the code editor, there is a single line of code: "print(nome)". A warning icon appears next to the line, indicating that the expression is implicitly coerced from 'String?' to 'Any'. The playground's output pane shows three rows of information: "nil", "Matilde", and "Optional("Matilde")\n". Below the output pane, the text "Optional("Matilde")" is displayed.

Print de String to tipo opcional

Mas repare que o valor que foi impresso não era o nome Matilde, mas sim uma indicação de que o que estava dentro da variável nome era um opcional. Isso é feito para deixar claro que o valor que está dentro da variável é um opcional.

Se quiser realmente o valor que está dentro do opcional, deve-se utilizar o operador !, como no exemplo a seguir:



The screenshot shows a Xcode playground window titled "MyPlayground.playground". The code in the playground is:

```
1 var nome:String?  
2  
3 nome = "Matilde"  
4  
5 print(nome!)|
```

The playground output pane shows three rows of results corresponding to the print statements:

Output	Value
nil	
"Matilde"	"Matilde"
"Matilde\n"	"Matilde\n"

The word "Matilde" is also highlighted in the sidebar under "Variables".

Lendo Optional com !



A técnica de utilizar o sinal `!`, para extrair o valor de um opcional, deve ser utilizado com muito cuidado! Isso só deve ser utilizado se o programador tiver certeza do que está fazendo, ou seja, ter certeza de que o opcional está preenchido.

Se a variável `nome` não foi preenchida com um valor e o operador `!` for utilizado para extrair seu valor, a aplicação vai quebrar. Veja o erro a seguir, nessa situação:

The screenshot shows a Xcode playground window titled "MyPlayground". The code in the playground is:

```

1 var nome:String?
2
3 //nome = "Matilde"
4
5 print(nome!) ✘ error: Execution was interrupted, reason: EXC_BAD_INSTRU...
6

```

The line `print(nome!)` is highlighted with a red background and has a red error icon next to it. Below the code, the output pane shows the error message:

**Fatal error: Unexpectedly found nil while unwrapping an Optional value: file
__lldb_expr_18/MyPlayground.playground, line 5**

Erro ao tentar extrair valor de Optional vazio



Para que o leitor não fique com a impressão de que o Optional só serve para atrapalhar, tenha em mente que o Swift quer evitar erros do programador, no caso do acesso às variáveis nulas. Porém, permite que elas possam existir. Nesse caso, ele quer garantir, através do operador `!`, que o programador sabe o que está fazendo. Isso faz com que o código fique mais evidente, com relação aos erros que podem acontecer nesse caso.

4.10 - Arrays e Dictionaries

O Swift possui suporte para se trabalhar com arrays e dicionários.

Arrays

Para trabalhar com arrays em Swift, basta fazer a declaração com a seguinte sintaxe:

```

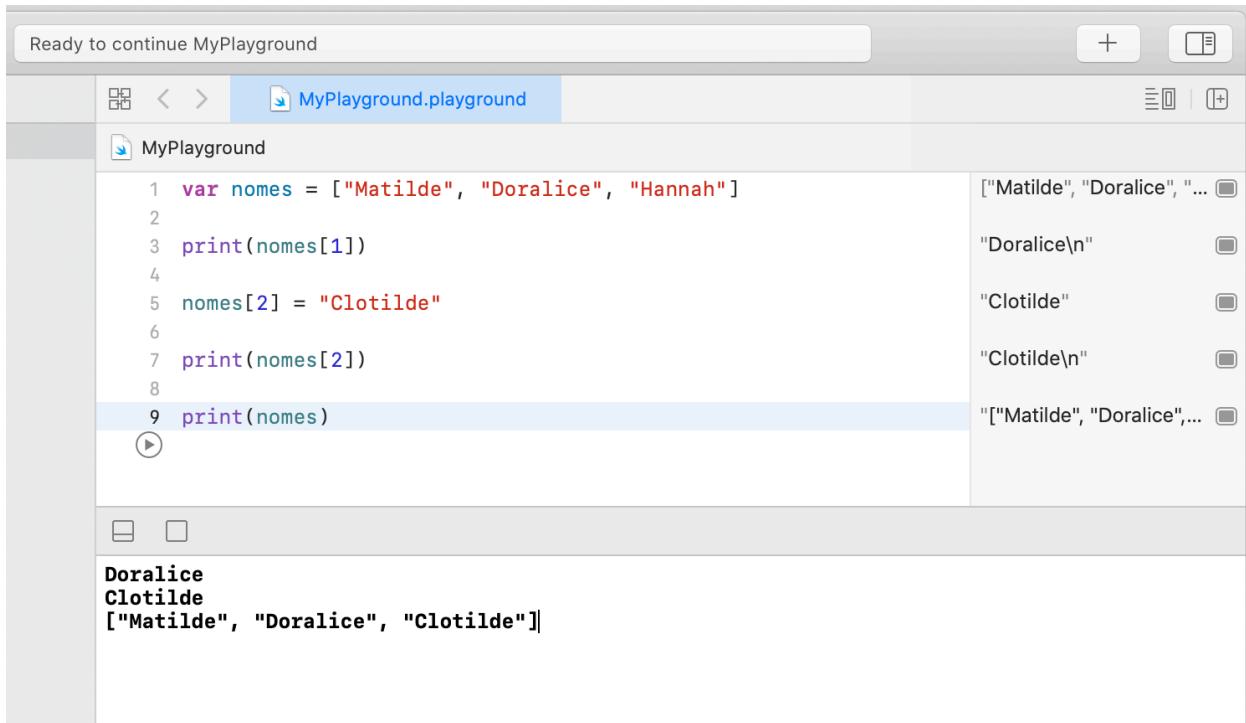
var nomes = ["Matilde", "Doralice", "Hannah"]

print(nomes[1])

```

É possível acessar os elementos do array colocando seu índice, que começa em 0.

Também é possível alterar os valores ao longo da execução:



The screenshot shows a Xcode playground window titled "MyPlayground.playground". The code in the playground is as follows:

```
1 var nomes = ["Matilde", "Doralice", "Hannah"]
2
3 print(nomes[1])
4
5 nomes[2] = "Clotilde"
6
7 print(nomes[2])
8
9 print(nomes)
```

The output pane shows the results of each print statement:

- "Doralice\n"
- "Clotilde"
- "Clotilde\n"
- "[\"Matilde", "Doralice", "Clotilde"]"

The code at line 9 is highlighted with a blue selection bar.

Declarando Arrays

Para adicionar um novo valor ao array, basta acessar seu método `append`, passando como argumento um valor do mesmo tipo do array. Veja no exemplo a seguir:

The screenshot shows a Xcode playground window titled "MyPlayground.playground". In the code editor, the following Swift code is written:

```

1 var nomes = ["Matilde", "Doralice", "Hannah"]
2
3 print(nomes[1])
4
5 nomes[2] = "Clotilde"
6
7 print(nomes[2])
8
9 print(nomes)
10
11 nomes.append("Doralice")
12
13 print(nomes)

```

The output pane displays the results of each print statement:

- "Doralice\n"
- "Clotilde"
- "Clotilde\n"
- "["Matilde", "Doralice", ...]"
- "["Matilde", "Doralice", ...]"
- "["Matilde", "Doralice", ...]"

Below the output, the playground's history shows the changes made to the array:

```

Doralice
Clotilde
["Matilde", "Doralice", "Clotilde"]
["Matilde", "Doralice", "Clotilde", "Doralice"]

```

Adicionando novo elemento ao Array

Perceba que a declaração do array, utilizado no exemplo anterior, foi feita de forma implícita, ou seja, o seu tipo não foi declarado. O Swift, nesse caso, infere o tipo do array pelos valores que foram colocados na sua atribuição inicial.

Veja um outro exemplo onde o Swift infere o tipo do array, pelos elementos da atribuição inicial:

```
var idades = [16, 15, 14]
```

Nesse caso, como todos os elementos da atribuição inicial são inteiros, então será um array de `Int`.

E nesse caso, como ficaria?

```
var pesos = [11.5, 6, 25]
```

Ora, não poderia ser um array de `Int`, pois há pelo menos um valor que não é inteiro. Logo, o Swift irá encarar como sendo um array de `Double`.

Também é possível declarar um Array, informando o tipo dos elementos, mas sem atribuir valores iniciais, utilizando a seguinte sintaxe:

```
var idades = [Int]()
```

E para adicionar valores a esse Array, basta chamar o método `append`:

```
var idades = [Int]()  
  
idades.append(16)  
idades.append(15)  
idades.append(14)  
  
print(idades)
```

Mais adiante, será mostrado como iterar nos elementos de um Array, utilizando estruturas de repetição.

Dictionaries

Uma explicação rudimentar, porém eficiente, do que são dicionários, é compará-los com mecanismos de armazenamento **chave-valor**, onde se tem uma lista de chaves, cada uma associada a um valor.

As mesmas regras da declaração implícita e explícita valem para os dicionários.

Para declarar um dicionário, com Strings como chaves e inteiros como valores, basta utilizar a seguinte sintaxe:

```
var nomesIdades = ["Matilde" : 16,  
                  "Doralice" : 15,  
                  "Hannah" : 14]
```

E para acessar um dos valores, através da chave, basta:

```
print(nomesIdades["Doralice"])
```

Para alterar um valor, através da chave:

```
nomesIdades["Doralice"] = 11
```

Para adicionar um novo valor com uma nova chave:

```
nomesIdades.updateValue(3, forKey: "Clotilde")
```

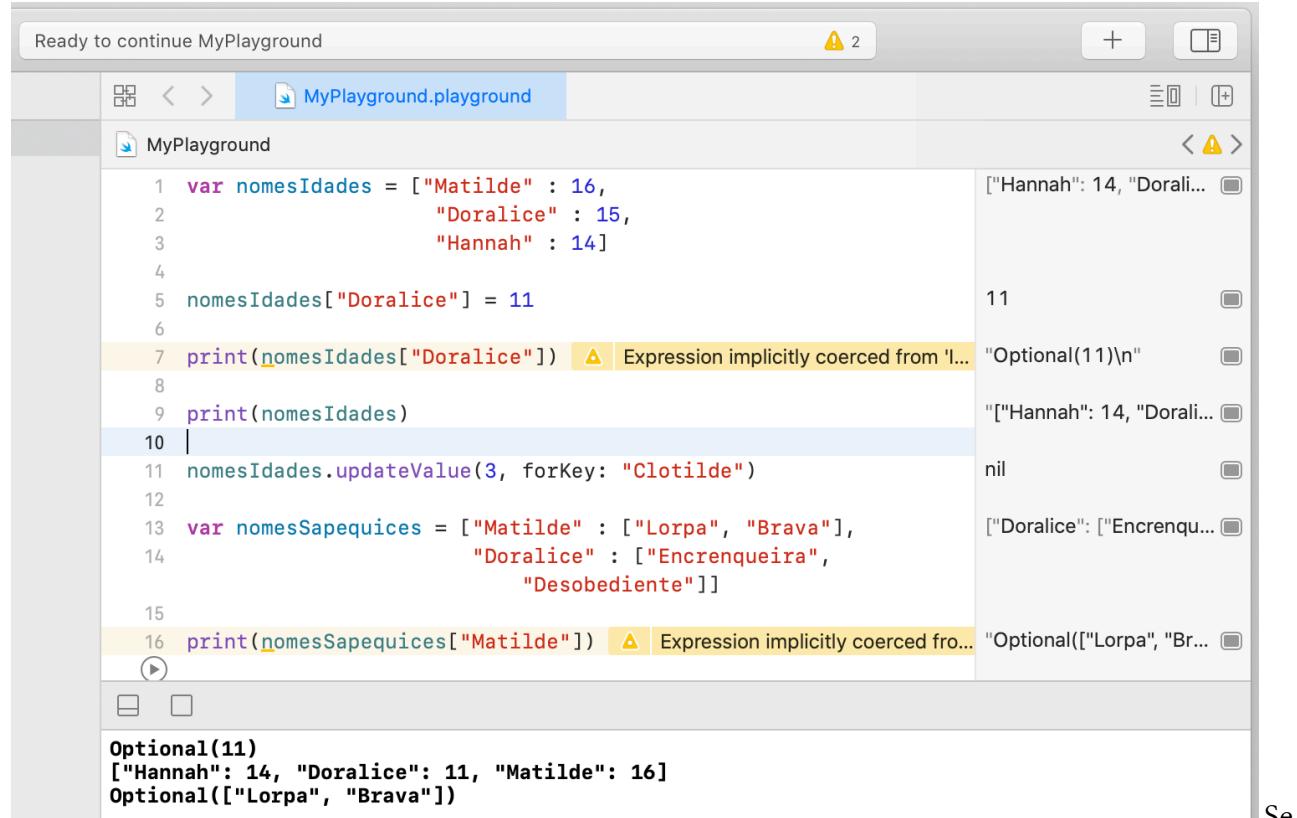
Não se preocupe com essa sintaxe. Isso será melhor explicado no tópico sobre funções.

Também é possível construir dicionários onde o valor pode ser um array, como no exemplo a seguir:

```
var nomesSapequices = ["Matilde" : ["Lorpa", "Brava"],
                      "Doralice" : ["Encrenqueira", "Desobediente"]]
```

Da mesma forma, é possível construir dicionários onde os valores são arrays de outros tipos.

É importante de lembrar que o valor extraído, de um dicionário, através de sua chave, é um Optional:



The screenshot shows an Xcode playground window titled 'MyPlayground.playground'. The code area contains the following Swift code:

```
Ready to continue MyPlayground
MyPlayground.playground
MyPlayground

1 var nomesIdades = ["Matilde" : 16,
2                     "Doralice" : 15,
3                     "Hannah" : 14]
4
5 nomesIdades["Doralice"] = 11
6
7 print(nomesIdades["Doralice"]) // Expression implicitly coerced from 'Optional<Int>' to 'Int'
8
9 print(nomesIdades)
10
11 nomesIdades.updateValue(3, forKey: "Clotilde")
12
13 var nomesSapequices = ["Matilde" : ["Lorpa", "Brava"],
14                         "Doralice" : ["Encrenqueira",
15                                     "Desobediente"]]
16
17 print(nomesSapequices["Matilde"]) // Expression implicitly coerced from 'Optional<[String]>' to '[String]'
```

The right pane shows the results of the print statements:

- Line 7: "Optional(11)\n"
- Line 9: "[Hannah": 14, "Doralice": 11, "Matilde": 16]
- Line 16: "[Lorpa", "Brava"]"

At the bottom of the playground window, there is a status bar with the text "Se".

Quiser acessar somente o valor de uma chave, sem o Optional, basta utilizar o operador `!`, lembrando do erro que isso pode causar, se a chave não existir:

```
print(nomesIdades["Doralice"]!)
```

4.11 - Controle de fluxo com if e switch

As instruções de controle de fluxo com `if` e `switch` em Swift são semelhantes a outras linguagens, mas com algumas regras e características interessantes, em particular com as expressões que podem ser utilizadas com o `switch`.

O if

O `if` no Swift funciona de maneira muito semelhante a outras linguagens (e não poderia ser diferente). Aqui há uma pequena lista de algumas diferenças mais marcantes:

- A condição a ser testada sempre deve ser um Boolean, ou seja, true ou false. Ele não faz comparação com zero (false) ou diferente de zero (true);

The screenshot shows a Xcode playground window titled "MyPlayground". The code editor contains the following Swift code:

```

1 var idade = 16
2
3 if idade {    Type 'Int' cannot be used as a boolean; test for '!=' instead
4     print ("Tem 16 anos")
5 }
6

```

A red error highlight covers the line `if idade {` with the message "Type 'Int' cannot be used as a boolean; test for '!=' instead". Below the code editor, the output pane shows the error message:

```

error: MyPlayground.playground:3:4: error: type 'Int' cannot be used as a boolean; test for
'!= 0' instead
if idade {
^
(      != 0)

```

if deve ser com Boolean

É claro que o teste não faz sentido para o que se propõe... Foi só um exemplo para demonstrar que o condicional deve ser um Boolean. Em outras linguagens, o teste passaria e a mensagem seria impressa.

Para eliminar o erro, o código deveria ser:

```

var idade = 16

if idade == 16 {
    print ("Tem 16 anos")
}

```

- Outra diferença é que os parênteses são opcionais, como pode ser visto no exemplo;
- As chaves são obrigatórios, mesmo que o código a ser executado, caso o teste passe, seja somente uma linha;
- É possível utilizar a instrução let em conjunto com o teste, para validar se o conteúdo de uma variável opcional está diferente de nulo ou não, e já extrair seu valor:

```
var idade:Int?  
  
idade = 16  
  
if let i = idade {  
    if (i == 16) {  
        print ("Tem 16 anos")  
    }  
}
```

Nesse caso, se `idade` fosse nulo, o primeiro teste já falharia, mas como ele possui um valor, então uma constante chamada `i` é criada, já atribuída com o valor de `idade`, e o primeiro teste passa.

Novamente, esse foi só um exemplo para explicar a utilização conjunta do `if` com o `let`, pois não seria necessário extrair o valor da variável `idade` para testar seu valor.

O switch

As operações de testes condicionais do `switch` em Swift suportam comparações com vários tipos de dados. Isso significa que é possível fazer muito mais do que comparações com inteiros e testes de igualdade.

Também é possível utilizar a instrução `let`, para testar e obter o valor de uma variável armazenado em uma constante, para que possa ser utilizado dentro do `case` do `switch`.

Outra diferença do `switch` no Swift em relação a outra linguagens, é que quando uma condição do `case` é satisfeita e seu bloco é executado, nenhum outro bloco com `case` será executado. Com isso, não é necessário colocar a instrução `break` no final de cada `case`. Também não é necessário colocar chaves para delimitar o bloco de código de cada `case`, embora isso possa melhorar a aparência do código.

Veja um exemplo básico de utilização do `switch`:

```
var idade:Int = 16  
  
switch (idade) {  
case 0...2:  
    print ("Bebezão")  
  
case 3..  
    print("Idade difícil")  
  
case 5:  
    print("Flor da idade")
```

```

case 6...10:
    print ("Maturidade")

case let i where idade > 10:
    print ("Idoso")

default:
    print ("Idade inválida")
}

```

Nos testes onde há ..., significa que a condição será verdadeira se o número estiver entre os dois valores, incluindo-os. Já a comparação 3..**<5** significa que será verdadeiro se a idade for 3 ou 4, somente, excluindo o 5. Essa forma de comparação de faixas de valores também pode ser utilizada em condicionais de estrutura de repetição, como será visto mais adiante.

No último case, repare a utilização do let. Isso poderia ser feito com uma String também, como no exemplo a seguir:

```

var nome = "Matilde Siécola"

switch (nome) {
case "Doralice":
    print ("Peste")

case let x where nome.hasPrefix("Matilde"):
    print ("Lorpa: \(x)")

default:
    print ("Nome inválido")
}

```

Veja como o switch pode ser versátil no Swift!

4.12 - Estruturas de repetição

Em Swift, existem as seguintes estrutura de repetição:

- **for**: mesmo uso que em outras linguagens como Java e C, mas com uma sintaxe um pouco diferente;
- **for-in**: como se fosse um **foreach**, para iterar em arrays e dictionaries;
- **while**: mesmo uso que em outras linguagens como Java e C, mas com uma sintaxe um pouco diferente;
- **repeat**: mesmo uso do **do-while** em outras linguagens como Java e C, mas com uma sintaxe um pouco diferente.

for

Veja um exemplo do `for`:

```
for i in 0..<10 {
    print(i)
}
```

Isso faz com que o programa imprima 10 vezes o valor de `i`, de 0 a 9.

Repare que não há parênteses na estrutura de comparação, nem a necessidade do incremento de `i`.

Para alterar o incremento, é necessário utilizar uma sintaxe diferente:

```
for i in stride(from: 0, to: 10, by: 2) {
    print(i)
}
```

for-in

Veja um exemplo do `for-in`, para iterar em um array:

```
var nomes = [String]("Matilde", "Doralice", "Hannah", "Clotilde"))

for (nome) in nomes {
    print(nome)
}
```

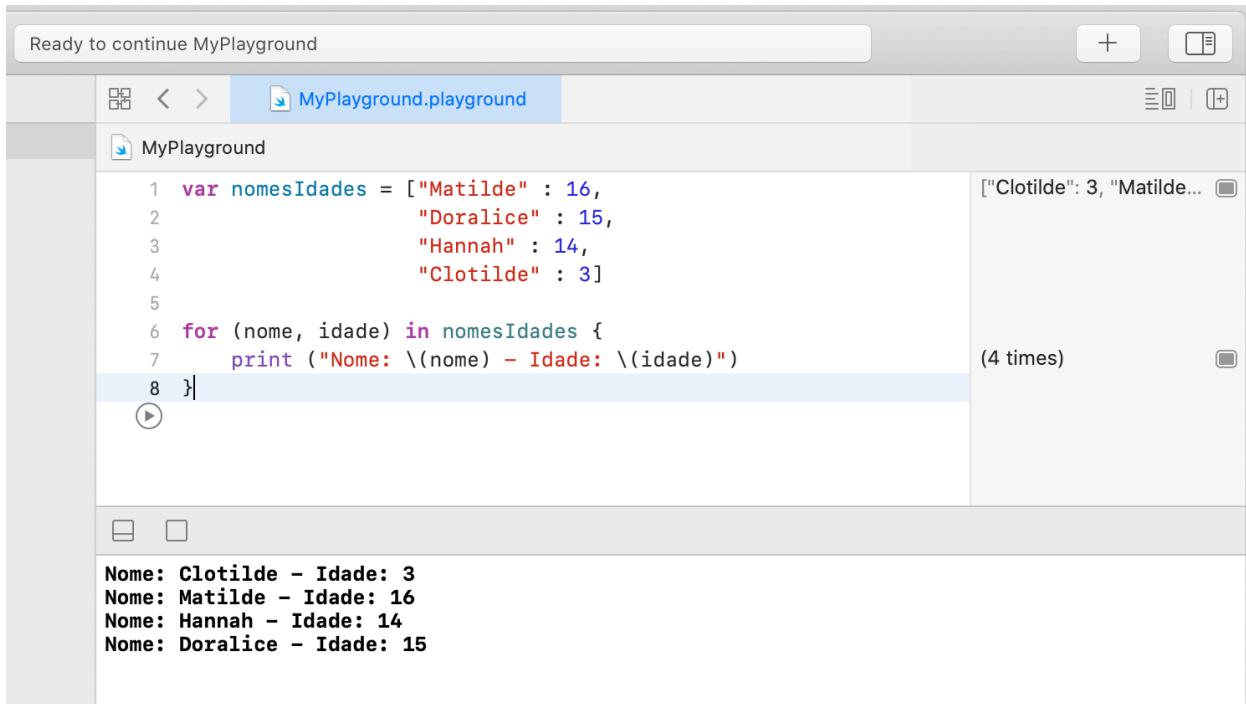
A variável `nome` é criada dentro do `for-in` e vale dentro do seu escopo somente.

Para iterar em um dictionary, basta utilizar a seguinte sintaxe:

```
var nomesIdades = ["Matilde" : 16,
                   "Doralice" : 15,
                   "Hannah" : 14,
                   "Clotilde" : 3]

for (nome, idade) in nomesIdades {
    print("Nome: \(nome) - Idade: \(idade)")
}
```

Nesse caso, veja como ficaria o resultado:



The screenshot shows a Xcode playground window titled "MyPlayground.playground". In the code editor, there is a snippet of Swift code:

```

1 var nomesIdades = ["Matilde" : 16,
2                     "Doralice" : 15,
3                     "Hannah" : 14,
4                     "Clotilde" : 3]
5
6 for (nome, idade) in nomesIdades {
7     print ("Nome: \(nome) - Idade: \(idade)")
8 }

```

The output pane shows the results of the execution:

```

["Clotilde": 3, "Matilde": 16, "Hannah": 14, "Doralice": 15]
(4 times)

```

Below the output, the terminal pane shows the individual printed statements:

```

Nome: Clotilde - Idade: 3
Nome: Matilde - Idade: 16
Nome: Hannah - Idade: 14
Nome: Doralice - Idade: 15

```

Iterando em um Dictionary

while

A estrutura `while` em Swift é bem semelhante a outra linguagens, como Java e C. Veja:

```

var j = 0

while j < 10 {
    print (j)
    j = j + 1
}

```

repeat

O `repeat` em Swift faz a mesma função do `do while` em outras linguagens, como Java e C. Veja:

```
var j = 0

repeat {
    print(j)
    j = j + 1
} while (j < 10)
```

4.13 - Funções

Para declarar uma função em Swift, basta utilizar `func`, seguido pelo nome da função e seus parâmetros entre parênteses. O retorno é definido pelo símbolo `->`. Veja um exemplo:

```
func soma (numA: Int, numB: Int) -> Int {
    var resultado: Int = 0

    resultado = numA + numB

    return resultado
}
```

E para chamar a função, basta utilizar a seguinte sintaxe:

```
let resultadoSoma = soma (numA: 3, numB: 4)
```

Repare que é necessário adicionar um *label* antes dos valores de cada parâmetro.

Caso queira-se escrever uma função para não exigir que os nomes dos parâmetros sejam passados durante sua chamada, basta adicionar o símbolo `_` antes do nome do parâmetro na declaração da função, como mostra o trecho a seguir:

```
func soma (_ numA: Int, numB: Int) -> Int {
    var resultado: Int = 0

    resultado = numA + numB

    return resultado
}

let resultadoSoma = soma (3, numB: 4)
```

Isso pode ser feito para qualquer parâmetro da função.



É importante que o leitor se adapte a essa notação de colocar os nomes dos parâmetros nas chamadas das funções, pois os frameworks do iOS utilizam muito dessa técnica para deixar as chamadas de suas funções mais declarativas e óbvias, no que tange ao que o método ou parâmetro realmente faz.

Também é possível escrever uma função que retorna mais de um parâmetro. Veja o exemplo a seguir:

```
func funcoesMatematicas (numA: Double, numB: Double) ->
(soma: Double, subtracaoAB: Double, multiplicacao: Double,
divisaoAB: Double, divisaoPorZero: Bool) {

    var soma: Double = 0
    var subtracaoAB: Double = 0
    var multiplicacao: Double = 0
    var divisaoAB: Double = 0
    var divisaoPorZero: Bool = false

    soma = numA + numB

    subtracaoAB = numA - numB

    multiplicacao = numA * numB

    if numB == 0 {
        divisaoPorZero = true
    } else {
        divisaoAB = numA / numB
    }

    return (soma, subtracaoAB, multiplicacao, divisaoAB, divisaoPorZero)
}
```

Os valores de retorno podem ser acessados pelo seus nomes, como pode ser visto no trecho a seguir:

```
let resultados = funcoesMatematicas (numA: 3, numB: 4)

print ("Soma: \(resultados.soma)")
print ("Subtração: \(resultados.subtracaoAB)")
print ("Multiplicação: \(resultados.multiplicacao)")
print ("Divisão: \(resultados.divisaoAB)")
print ("Divisão por zero: \(resultados.divisaoPorZero)")
```

Funções que retornam Optional

É possível, para uma função, retornar um valor do tipo Optional, por qualquer que seja a razão. Veja no exemplo a seguir, onde o resultado de uma função que realiza a divisão de dois números, retorna um Optional:

```
func divisao (numA: Double, numB: Double) -> Double? {
    var resultado: Double?

    if (numB != 0) {
        resultado = numA / numB
    }

    return resultado
}
```

O retorno está marcado como Optional, através do sinal ?, após o tipo de retorno.

Essa função poderia ser chamada da seguinte forma:

```
let resultado = divisao(numA: 4, numB: 3)
print ("Resultado: \(resultado)")
```

Porém, o valor impresso seria:

Resultado: Optional(1.333333333333333)

Para testar o valor, se for diferente de nulo e já pegar o resultado, caso diferente de nulo, poderia se utilizar a técnica do if com o let. Veja:

```
if let resultado = divisao(numA: 4, numB: 3) {
    print ("Resultado: \(resultado)")
} else {
    print ("Divisão por zero")
}
```

Nesse caso, o valor impresso seria:

Resultado: 1.333333333333333

4.14 - Objetos e classes

Para declarar uma classe em Swift, basta utilizar `class` antes do nome da classe. Propriedades e métodos da classe devem ser escritos da mesma forma que foram vistos anteriormente. Veja um exemplo:

```
class Cachorro {
    var nome: String?
    var idade: Int?
    var peso: Double?

    func tipoPorIdade() -> String? {
        var tipo: String?

        if let i = self.idade {
            switch (i) {
                case 0...2:
                    tipo = "Bebezão"

                case 3..<5:
                    tipo = "Idade difícil"

                case 5:
                    tipo = "Flor da idade"

                case 6...10:
                    tipo = "Maturidade"

                case _ where i > 10:
                    tipo = "Idoso"

                default:
                    tipo = "Sem idade"
            }
        }

        return tipo
    }
}
```

```
        tipo = "Idade inválida"
    }
}

return tipo
}
}
```

Veja que todas as propriedades da classe estão marcadas como opcionais, com ?. Isso porque, como já dito anteriormente, todas as variáveis devem ser inicializadas no Swift, e o mesmo ocorre para as propriedades de uma classe.

Repare que, para acessar uma propriedade da classe, basta utilizar `self` e o nome da propriedade.

Para criar uma instância da classe `Cachorro`, basta fazer:

```
let matilde = Cachorro()
```

Para alterar uma propriedade dessa nova instância:

```
matilde.idade = 16
```



Repare que a propriedade `idade` da instância de `Cachorro` pode ser alterada, mesmo que a instância da classe tenha sido declarada como constante, com a instrução `let`. Isso porque é a instância que não pode ser alterada, e não suas propriedades.

Para chamar o método da classe que foi criada, basta:

```
if let tipo = matilde.tipoPorIdade() {
    print ("Tipo de cachorro: \(tipo)")
}
```

O `let` e o `if` foram utilizados aqui, porque o método retorna um `Optional`.

4.15 - Inicialização da classe

A classe pode ter um método que é executado quando um objeto é instanciado. Esse método é chamado de inicializador:

```
class Cachorro {
    var nome: String
    var idade: Int?
    var peso: Double?

    init (nome: String) {
        self.nome = nome
    }
}
```

Dessa forma, a propriedade `nome` pode deixar de ser opcional, pois agora ela é inicializada no método `init`. O mesmo poderia ser feito para as demais propriedades.

Agora, para instanciar um objeto da classe `Cachorro`, bastaria:

```
let matilde = Cachorro(nome: "Matilde")
```

Toda propriedade da classe precisa ser inicializada, seja na sua declaração ou no método inicializador. O correto então seria:

```
class Cachorro {
    var nome: String
    var idade: Int
    var peso: Double

    init (nome: String, idade: Int, peso: Double) {
        self.nome = nome
        self.idade = idade
        self.peso = peso
    }
}
```

As propriedades também poderiam ser inicializadas com valores padrões, mas isso não é uma boa prática.

Nesse caso, a criação de um novo objeto `Cachorro`, ficaria:

```
let matilde = Cachorro(nome: "Matilde", idade: 16, peso: 12.5)
```

A classe completa, com as alterações no método `tipoPorIdade`, ficaria:

```
class Cachorro {  
    var nome: String  
    var idade: Int  
    var peso: Double  
  
    init (nome: String, idade: Int, peso: Double) {  
        self.nome = nome  
        self.idade = idade  
        self.peso = peso  
    }  
  
    func tipoPorIdade() -> String? {  
        var tipo: String?  
  
        switch (idade) {  
        case 0...2:  
            tipo = "Bebezão"  
  
        case 3..<5:  
            tipo = "Idade difícil"  
  
        case 5:  
            tipo = "Flor da idade"  
  
        case 6...10:  
            tipo = "Maturidade"  
  
        case _ where idade > 10:  
            tipo = "Idoso"  
  
        default:  
            tipo = "Idade inválida"  
        }  
  
        return tipo  
    }  
}
```



Muito ainda será visto sobre classes, nesse capítulo e até o fim do livro. Eles serão apresentados quando necessários.

4.16 - Herança

Para fazer que uma classe herde de outra, basta fazer como o exemplo a seguir:

```
class CachorroPreto: Cachorro {  
    let cor = "Preto"  
    var raca: String  
  
    init (nome: String, idade: Int, peso: Double, raca: String) {  
        self.raca = raca  
        super.init(nome: nome, idade: idade, peso: peso)  
    }  
}
```

Dessa forma, a nova classe CachorroPreto, herda de Cachorro, com todos as suas propriedades e métodos.

É possível adicionar outras propriedades à nova classe, declarando-os da mesma forma como já era feito.

Caso seja necessário invocar um método da classe pai, basta utilizar `super`.

Se for preciso sobrescrever algum método da classe pai, é só colocar o modificador `override` antes da função de mesmo nome, como no exemplo a seguir:

```
override func tipoPorIdade() -> String? {  
    return "Sempre preto!"  
}
```

4.17 - Protocol

Em outras linguagens, como Java e C#, é possível definir um contrato de implementação para classes, que é chamado de **interface**. Em Swift, um conceito semelhante é chamado de **Protocol**.

O Protocol possui requisitos, que podem ser de propriedades ou métodos. Veja um exemplo a seguir:

```
protocol AtitudesQualidadesDeCachorro {
    var brincalhao: Bool { get set }

    func quantoDormePorDia () -> Int
}
```

Para uma classe adotar um protocolo, basta declará-lo da mesma forma como foi feito para herdar de uma outra classe. Porém, na definição da classe, é necessário colocar os nomes das classes que ela herda, antes dos protocolos.

Nesse caso, se uma classe adotar o protocolo `AtitudesQualidadesDeCachorro`, ela terá os mesmos atributos e funções de tal protocolo. Veja no exemplo a seguir:

```
class CachorroPreto: Cachorro, AtitudesQualidadesDeCachorro {
    let cor = "Preto"
    var raca: String

    var brincalhao: Bool

    init (nome: String, idade: Int, peso: Double, raca: String) {
        self.raca = raca
        self.brincalhao = true
        super.init(nome: nome, idade: idade, peso: peso)
    }

    func quantoDormePorDia() -> Int {
        return 4
    }

    override func tipoPorIdade() -> String? {
        return "Sempre preto!"
    }
}
```

Protocolos são interessantes, principalmente para implementações de listeners.

4.18 - Deinitialization

É possível criar um método, dentro de uma classe, para ser chamado quando a instância dessa classe for desalocada (mais informações como esse processo de desalocação no próximo tópico).

Dessa forma, é possível executar códigos para, por exemplo, fechar arquivos, conexões ou outras tarefas importantes, mas que só devem ser realmente executadas quando nada mais for necessário na instância da classe.

Para isso, basta criar a função `deinit`, sem receber parâmetros, que ela será chamada quando a instância da classe for desalocada. Essa função não pode ser chamada, como qualquer outra função. Veja um exemplo:

```
class Cachorro {
    var nome: String

    init (nome: String) {
        self.nome = nome
        print ("Novo cachorro: \(self.nome)")
    }

   _deinit {
        print ("Apagando cachorro: \(self.nome)")
    }
}
```

4.19 - Automatic Reference Counting - ARC

No Swift, não é necessário se preocupar em desalocar instâncias de classes. Isso é feito automaticamente pelo **ARC - Automatic Reference Counting**.

O ARC trabalha de uma forma semelhante como outros mecanismos de outras linguagens, monitorando as referências de um objeto em relação às variáveis que estão utilizando-as. Quando não houver mais nenhuma referência a uma instância, o ARC então a desaloca, quando possível. Nesse momento, o método `deinit` é chamado, se estiver definido na classe.

Na classe `Cachorro`, criada no tópico anterior, esse comportamento pode ser observado, quando atribuímos `nil` à instância criada:

```
var cachorro1: Cachorro?

cachorro1 = Cachorro(nome: "Doralice", idade: 15, peso: 9)

// O método deinit é chamado nesse momento
cachorro1 = nil
```

4.20 - Enum

Enums são utilizados para criar definições de valores. Veja um exemplo:

```
enum Velocidade: Int {  
    case Lento = 1  
    case Normal = 3  
    case Rapido = 5  
    case Veloz = 7  
    case MaisVeloz, SuperVeloz, VelocidadeDaLuz  
}
```

Por padrão, os valores começam sempre de zero e são incrementados de 1 a cada definição. No exemplo dado, é atribuído 1 ao primeiro valor e outros valores fixos às demais definições. Quando não há mais definição de valor, são atribuídos valores incrementados de um a cada nova definição.

Para utilizar esse enum, basta colocar seu nome e o valor desejado:

```
print ("Velocidade: \(Velocidade.Normal)")
```

Também é possível definir uma constante a partir de um valor do enum:

```
let velocidade = Velocidade.Rapido  
  
print ("Velocidade: \(velocidade)")
```

Se for necessário obter o valor numérico do enum, basta acessar a propriedade rawValue:

```
print ("Velocidade: \(velocidade.rawValue)")
```

4.21 - Tratamento de erros

Em Swift, assim como em outras linguagens como Java e C#, é possível fazer tratamento de erros de execução de código. Também é possível criar seu próprio tipo de erro e lançá-lo durante a execução de uma função, por exemplo.

Para criar o seu próprio tipo de erro, basta definir um enum que implemente o protocolo `ErrorType`, como pode ser visto a seguir:

```
enum ValoresErrados: Error {
    case DivisorZero
    case DividendoZero
    case DividendoMenorQueDivisor
    case DivisorNegativo
    case DividendoNegativo
}
```

E para fazer com que uma função lance uma exceção, utilize a instrução `throws` antes do tipo de retorno dessa função:

```
func dividindoMacas (macas: Int, criancas: Int) throws -> Int
```

Veja que não é necessário especificar o tipo da exceção que é lançada.

Dentro da função, para lançar uma exceção, utilize a instrução `throw`:

```
throw ValoresErrados.DivisorZero
```

Veja como fica uma função de exemplo, que divide maçãs para crianças, mas que faz alguns testes para ver se não vai dar briga...

```
func dividindoMacas (macas: Int, criancas: Int) throws -> Int {

    if criancas == 0 {
        throw ValoresErrados.DivisorZero
    }

    if macas < criancas {
        throw ValoresErrados.DividendoMenorQueDivisor
    }

    return macas / criancas
}
```

Essa função poderia fazer mais testes e lançar outras exceções definidas em `ValoresErrados`.

Para chamar a função `dividindoMacas` e tratar as possíveis exceções que ela pode dar, é só utilizar a instrução `try` antes de chamá-la, dentro de um bloco do `- catch`. Veja a seguir como pode ficar:

```

do {
    let macasParaCadaCrianca = try dividindoMacas(macas: 3, criancas: 2)

    print ("\"(\(macasParaCadaCrianca)) maçã(s) para cada criança")

} catch ValoresErrados.DivisorZero {
    print("Não é possível dividir maçãs para nenhuma criança")
} catch ValoresErrados.DividendoMenorQueDivisor {
    print("Não é possível dividir o número de maçãs para " +
        "essa quantidade de crianças")
}

```

A sequência de `catch` funciona da mesma forma que o `case` em um `switch`, seguindo a ordem em que foi colocado e executando somente um.

Ainda seria possível adicionar um bloco de código para ser executado independentemente do que acontecesse com os `catch`. Esse bloco é o `defer`. Veja como ficaria:

```

do {
    let macasParaCadaCrianca = try dividindoMacas(macas: 3, criancas: 2)

    print ("\"(\(macasParaCadaCrianca)) maçã(s) para cada criança")

} catch ValoresErrados.DivisorZero {
    print("Não é possível dividir maçãs para nenhuma criança")
} catch ValoresErrados.DividendoMenorQueDivisor {
    print("Não é possível dividir o número de maçãs para " +
        "essa quantidade de crianças")
}; defer {
    print ("Nenhuma criança brigou...")
}

```

A mesma instrução que define o bloco `defer` também pode ser utilizada dentro da função que lança a exceção, como forma de garantir que um trecho de código seja executado, mesmo que um erro aconteça:

```
func dividindoMacas (macas: Int, criancas: Int) throws -> Int {

    defer {
        print ("Será que deu certo?")
    }

    print("Calculando maçãs para cada criança...")

    if criancas == 0 {
        throw ValoresErrados.DivisorZero
    }

    if macas < criancas {
        throw ValoresErrados.DividendoMenorQueDivisor
    }

    print("Maças divididas para cada criança!")
    return macas / criancas
}
```

Dentro da função, o código delimitado por `defer` será executado nas seguintes condições:

- Se nenhuma exceção for lançada, ele será executado antes do `return`;
- Se uma exceção for lançada, será executado logo após seu lançamento, antes da função retornar para o ponto em que foi chamada.

Tratando exceções sem `catch`

É possível chamar a função `dividindoMacas` e tratar todas as exceções que ela pode lançar, sem utilizar `catch`. Veja como isso pode ser feito:

```
let macasParaCadaCrianc = try? dividindoMacas(macas: 3, criancas: 2)

print ("\nmacasParaCadaCrianc) maçã(s) para cada criança")
```

Veja que basta colocar o modificador `?` após a instrução `try`. Porém, isso transformará o retorno da função em um `Optional`.

Essa técnica fica melhor se utilizar o `if let`, como já foi aprendido:

```
if let macasParaCadaCrianca = try? dividindoMacas(macas: 3, criancas: 2) {
    print ("\"(\(macasParaCadaCrianca)) maçã(s) para cada criança")
} else {
    print ("Não foi possível dividir as maçãs")
}
```

Utilizando o `if let`, o código do `else` será executado se a função lançou alguma exceção. Já o código do `if` só será executado se tudo deu certo, e ainda, dentro desse bloco a variável `macasParaCadaCrianca` já foi retirada do `Optional` e pode ter o seu valor acessado diretamente.

4.22 - Structures

Structures são muito semelhantes a classes em Swift e em casos mais simples, utilizá-las ao invés de classes é mais interessante.

Veja como deveria ficar a definição de uma `struct` utilizada nos exemplos anteriores nesse capítulo:

```
struct Cachorro {
    var nome: String
    var idade: Int
    var peso: Double
}
```

Veja que não é necessário definir um inicializador padrão para as propriedades, pois isso já é feito automaticamente pelo compilador, o que significa que a criação de uma instância dessa struct ficaria dessa forma:

```
var cachorro1 = Cachorro(nome: "Matilde", idade: 16, peso: 12.5)
```

O acesso às propriedades de uma instância de uma `struct` é da mesma forma que é feito em instâncias de classes.

Da mesma forma, uma `struct` pode ter funções, como é feito em classes e foi apresentado nesse capítulo.

Uma diferença importante entre classe e `struct` é que uma instância de uma `struct` é sempre passada como valor, ao contrário de uma instância de uma classe que é passada como referência. Isso significa que uma cópia da instância da `struct` é passada. Veja um exemplo de uma situação mais simples:

```

struct Cachorro {
    var nome: String
    var idade: Int
    var peso: Double
}

var cachorro1 = Cachorro(nome: "Matilde", idade: 16, peso: 12.5)

var cachorro2 = cachorro1
cachorro2.idade = 10

print ("Idade do cachorro 1: \(cachorro1.idade)")
print ("Idade do cachorro 2: \(cachorro2.idade)")

```

As duas mensagens a serem impressas nesse caso seriam:

```

Idade do cachorro 1: 16
Idade do cachorro 2: 10

```

Repare que as idades são diferentes entre as duas instâncias da struct Cachorro, pois quando a variável cachorro1 foi atribuída à variável cachorro2, na verdade foi criada uma **cópia da instância**. Com isso, a alteração em uma instância não interfere na outra. O mesmo aconteceria se ela fosse passada como **parâmetro em uma função**.



Em SwiftUI, structures serão largamente utilizadas para definir as *views*, ou seja, os componentes gráficos que serão utilizados para construir as interfaces gráficas.

Porém, se a ideia é utilizar heranças nas definições de modelos, para que uma classe herde comportamentos de outra classe, então utilize classes ao invés de structures.



Para maiores informações sobre como decidir entre o uso de classe ou struct, consulte esse link⁷.

4.23 - Closures

Em Swift, Closures são blocos de códigos auto-contidos para realizar uma determinada tarefa, e que podem ser passados como parâmetros para funções. Eles são, de certa forma, semelhantes ao conceito de funções Lambda em outras linguagens. A ideia de apresentar esse conceito nesse capítulo

⁷https://developer.apple.com/documentation/swift/choosing_between_structures_and_classes

introdutório se dá pelo fato que um aplicativo para iOS, construído com **SwiftUI**, utiliza muito o conceito de **Closures** para construção de funções que devem ser executadas mediante a eventos, como os gerados pelas ações do usuário, por exemplo.

Um Closure em Swift pode capturar variáveis e constantes dentro do contexto onde ele foi definido, o que facilita sua escrita, sem a necessidade de passar tais valores como parâmetros.

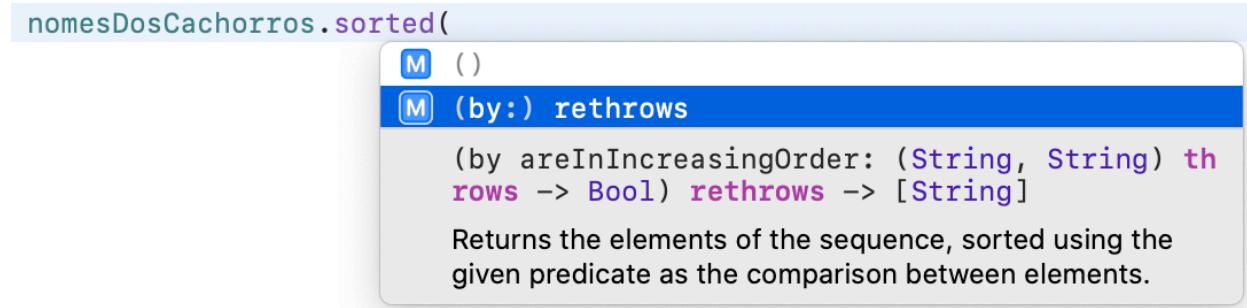
Com Closures é possível tornar o código mais enxuto, utilizando-se de suas características como:

- Inferir parâmetros e tipos de retornos;
- Deixar retornos implícitos quando tem-se uma única expressão;
- Criar nomes de argumentos mais curtos;
- Utilizar a sintaxe `trailing closure`, quando um Closure é o último parâmetro de uma função a ser chamada.

Para exemplificar alguns desses conceitos, tome um *array* de Strings, como no exemplo a seguir:

```
let nomesDosCachorros = ["Clotilde", "Matilde", "Doralice", "Hannah"]
```

Um *array* como esse possui uma função chamada `sorted`, que recebe uma função como parâmetro, contendo dois argumentos do tipo String e um Bool, como retorno, para dizer se uma String deve ser ordenada antes ou depois da outra. Veja a seguir sua sintaxe:



Função `sorted by`

O propósito da função `sorted` é permitir que o desenvolvedor crie seu próprio mecanismo para dizer como o *array* deve ser ordenado.

Então, imagine que a ideia é ordenar as Strings em ordem alfabética reversa. Logo, ela poderia ser da seguinte forma:

```
func inverterNomes(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
```

Veja que a função `inverterNomes` possui a mesma sintaxe exigida pela função `sorted`. Dessa forma, ela poderia ser passada como parâmetro, como no trecho a seguir:

```
let nomesDosCachorros = ["Clotilde", "Matilde", "Doralice", "Hannah"]

func inverterNomes(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}

let nomesInvertidos = nomesDosCachorros.sorted(by: inverterNomes)

print(nomesInvertidos)
```

O que produziria o seguinte resultado:

```
["Matilde", "Hannah", "Doralice", "Clotilde"]
```

É claro que a ideia da função `inverterNomes` aqui é bem simples. Ela apenas diz se uma `String` é maior do que a outra, em termos de ordenação alfabética.

Utilizando a sintaxe do `Closure`, seria possível fazer o mesmo trabalho, sem a necessidade de criar uma função:

```
{ (parameters) -> return type in
    statements
}
```

Seguindo esse princípio, o código ficaria da seguinte forma:

```
let nomesDosCachorros = ["Clotilde", "Matilde", "Doralice", "Hannah"]

let nomesInvertidos = nomesDosCachorros.sorted(by: { (s1: String, s2: String) -> Bool
    in
        return s1 > s2
})

print(nomesInvertidos)
```

Nesse exemplo, o `Closure` foi criado e passado como parâmetro de nome `by` da função `sorted`, com a mesma sintaxe exigida por ele, ou seja, dois parâmetros do tipo `String` e um retorno do tipo `Bool`. Mas esse código pode ficar um pouco mais curto, valendo-se da característica do `Closure` de inferir tipos de parâmetros e de retorno:

```
let nomesDosCachorros = ["Clotilde", "Matilde", "Doralice", "Hannah"]

let nomesInvertidos = nomesDosCachorros.sorted() { s1, s2 in
    s1 > s2
}

print(nomesInvertidos)
```

Aqui já é possível ver o conceito de trailing closure, onde um Closure deve ser passado como último parâmetro de um função, fazendo que ele na verdade possa ser escrito do lado de fora dos parênteses da chamada da função. Essa sintaxe será muito utilizada na construção das *views* com SwiftUI.

Nesse trecho de código, `s1` e `s2` são do tipo String, que foi inferido pela sintaxe da função exigida pelo parâmetro `by` da função `sorted`, assim como o tipo do retorno, que deve ser `Bool`. Ainda, a instrução `return` pôde ser omitida, pois o Closure possui apenas uma expressão, que é a comparação `s1 > s2`. Existe ainda uma forma de deixar o código ainda mais enxuto, que é a omissão dos parâmetros de entrada do Closure:

```
let nomesDosCachorros = ["Clotilde", "Matilde", "Doralice", "Hannah"]

let nomesInvertidos = nomesDosCachorros.sorted() {
    $0 > $1
}
```

Nesse caso, `$0` e `$1` representam os dois parâmetros de entrada do Closure.

É importante ter em mente essa sintaxe e o conceito de Closures, pois como dito, isso será largamente utilizado na construção das interfaces gráficas com SwiftUI.



Para maiores informações sobre Closures, consulte a documentação oficial através desse [link](#)⁸.

4.24 - Conclusão

Esse capítulo mostrou alguns conceitos chaves do Swift, principalmente aqueles que serão mais utilizados ao longo desse livro.

Não deixe de consultar a documentação oficial do Swift, apresentado no início do capítulo, caso tenha alguma dúvida ou queira se aprofundar mais em algum conceito.

O próximo capítulo começará a criação de um aplicativo simples, apresentando alguns conceitos introdutórios do SwiftUI, assim como a funcionalidade de *preview* do Xcode.

⁸<https://docs.swift.org/swift-book/LanguageGuide/Closures.html>

5 - Hello World com SwiftUI

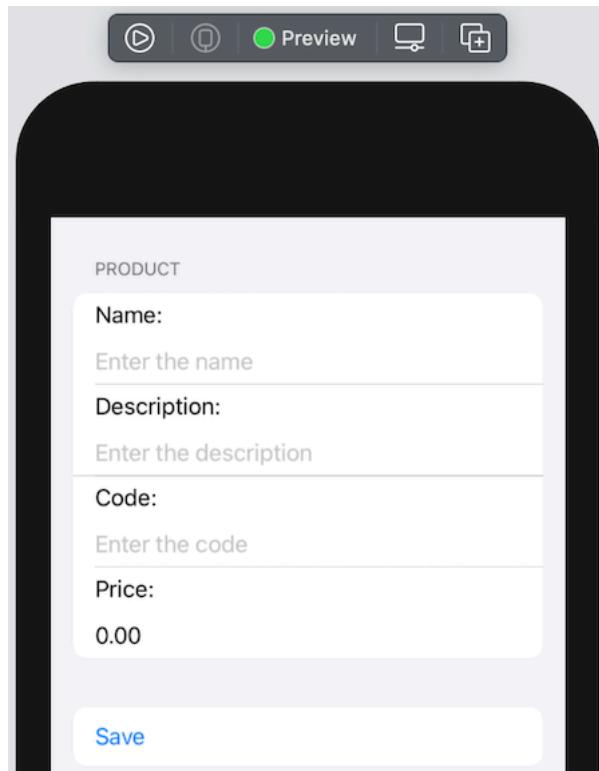
Agora que alguns conceitos de Swift foram apresentados no capítulo anterior, é possível iniciar o desenvolvimento de um aplicativo simples para iOS, utilizando o **SwiftUI**⁹, uma forma declarativa de construir interfaces gráficas para qualquer dispositivo da Apple, totalmente integrado do Xcode.

A ideia desse capítulo é construir um aplicativo para cadastrar um produto e mostrá-lo ao usuário. Na verdade o produto em si não será salvo em nenhum lugar, mas muitos conceitos já poderão ser ensaiados, como:

- Como construir uma interface gráfica com SwiftUI;
- Reagir a eventos do usuários, como em cliques de botões;
- Fazer validações de informações de entrada;
- Entender conceitos como State e Binding;
- Utilizar o Xcode Preview em vários dispositivos diferentes;
- Refatorar a tela para compô-la em várias *views*.

A figura a seguir mostra como será parte da interface gráfica do projeto proposto nesse capítulo:

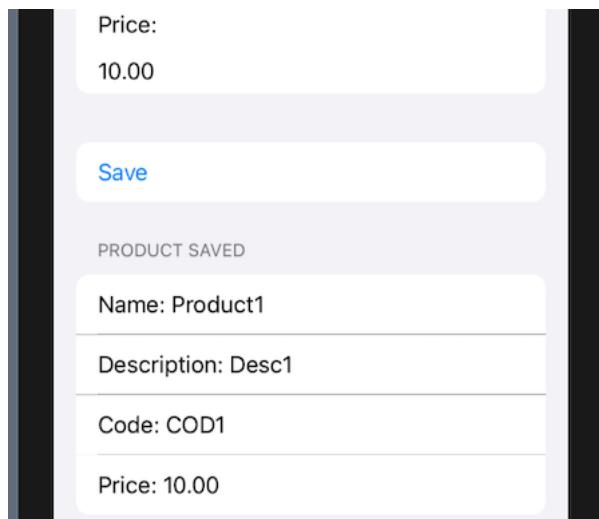
⁹<https://developer.apple.com/xcode/swiftui/>



Cadastro do produto

Perceba que essa aplicação está sendo executada no Xcode Preview, que permite que o aplicativo possa ser testado enquanto o desenvolvedor cria seu código.

A figura a seguir mostra o restante da interface gráfica, exibindo o produto “salvo” ao usuário:



Produto salvo

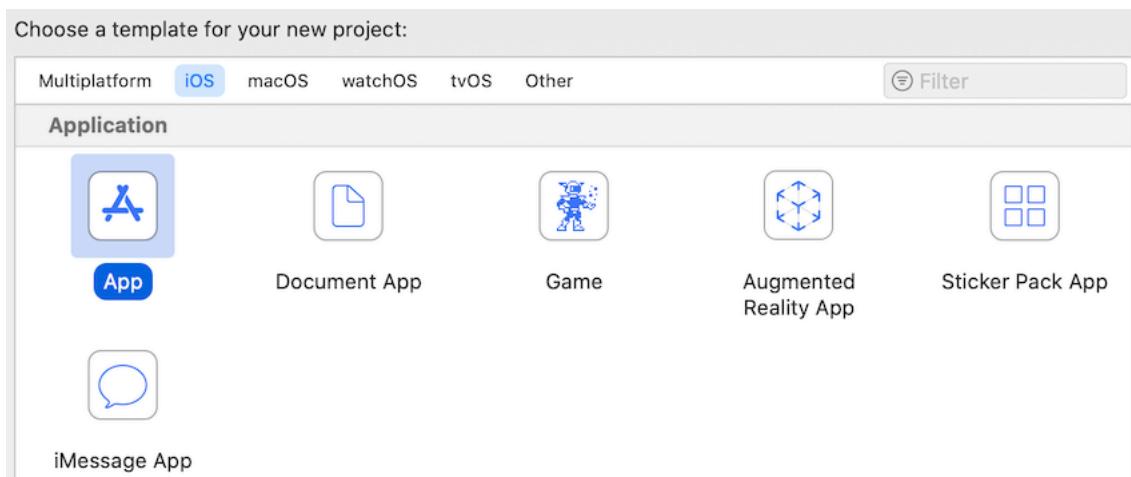
À medida em que esse capítulo evolui, outros conceitos serão mostrados com mais detalhes, princi-

palmente no que se refere a construção da interface gráfica e como utilizar o Xcode Preview para visualizá-la.

Ainda serão mostrados os conceitos de State e Binding, que permitem controlar o estado da interface do usuário de forma simples e criando muito menos código para mantê-la sempre atualizada às alterações nos dados que a compõem.

5.1 - Criando o projeto

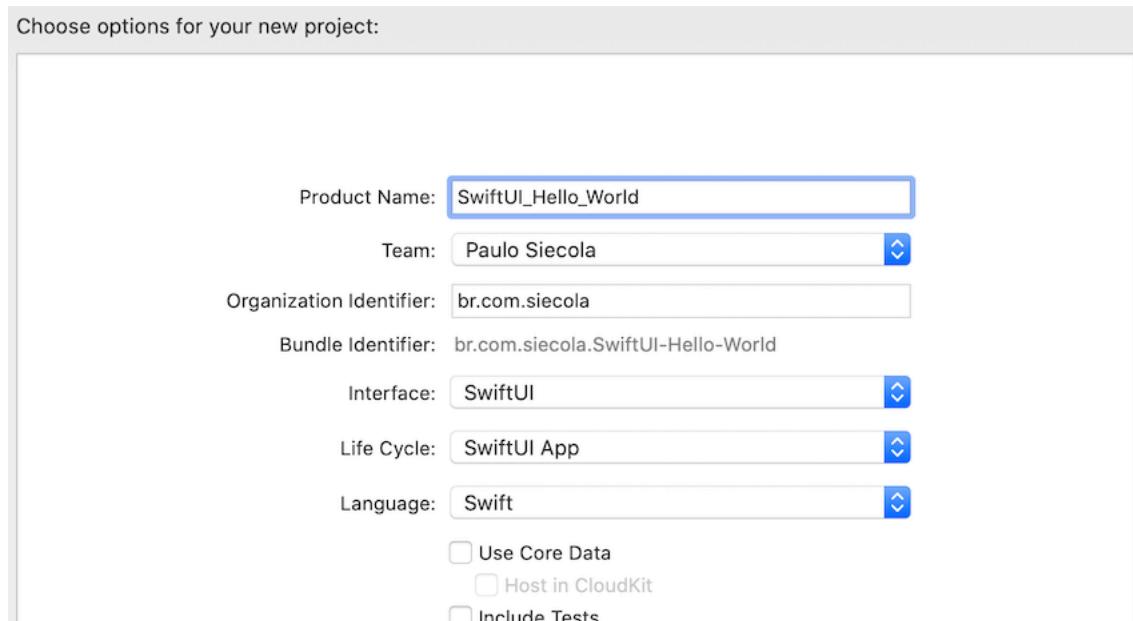
Para começar a criar o projeto no Xcode, escolha a opção `File -> New -> Project` e escolha a opção `iOS -> App`, como mostra a figura a seguir:



Escolhendo o template para o novo projeto

Esse *template* escolhido faz com que o Xcode crie o projeto com tudo o que é necessário para desenvolver um aplicativo para iOS.

Clique em `Next` para passar para a próxima tela de criação do projeto, como mostra a figura a seguir:

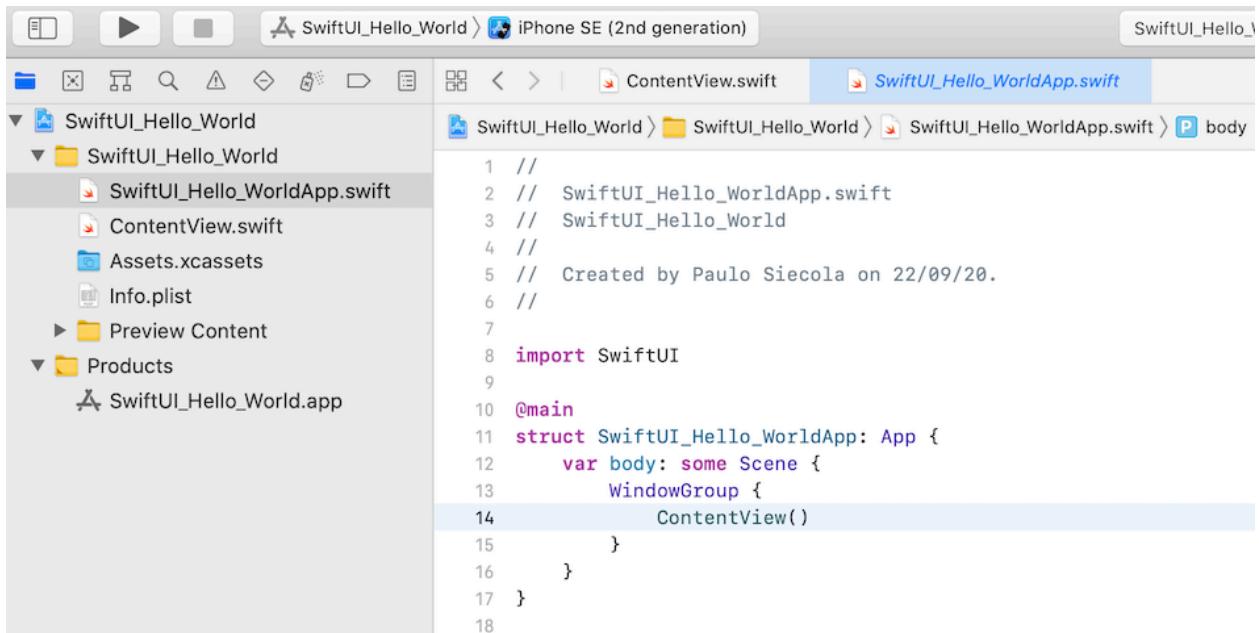


Configurações do novo projeto

Digite o nome do projeto no campo Product Name e deixe as configurações Interface, Life Cycle e Language como mostram a figura anterior. Para continuar, clique em Next e escolha o local onde o projeto será salvo na máquina.

5.2 - Entendendo a estrutura do projeto

Depois que o Xcode finaliza a criação do projeto, é possível visualizar sua estrutura na seção Project Navigator na aba Navigator do Xcode, como mostra a figura a seguir:



Estrutura do projeto

A estrutura do projeto contém dois arquivos mais importantes nesse momento:

- **SwiftUILHelloWorldApp.swift**: esse é o ponto de entrada da aplicação. Nele há uma *struct* que implementa o protocolo *App*, para exibir a primeira tela da aplicação;
- **ContentView.swift**: é aqui onde está a definição da primeira tela a ser exibida.

Veja como o arquivo ContentView.swift está organizado:

```

import SwiftUI

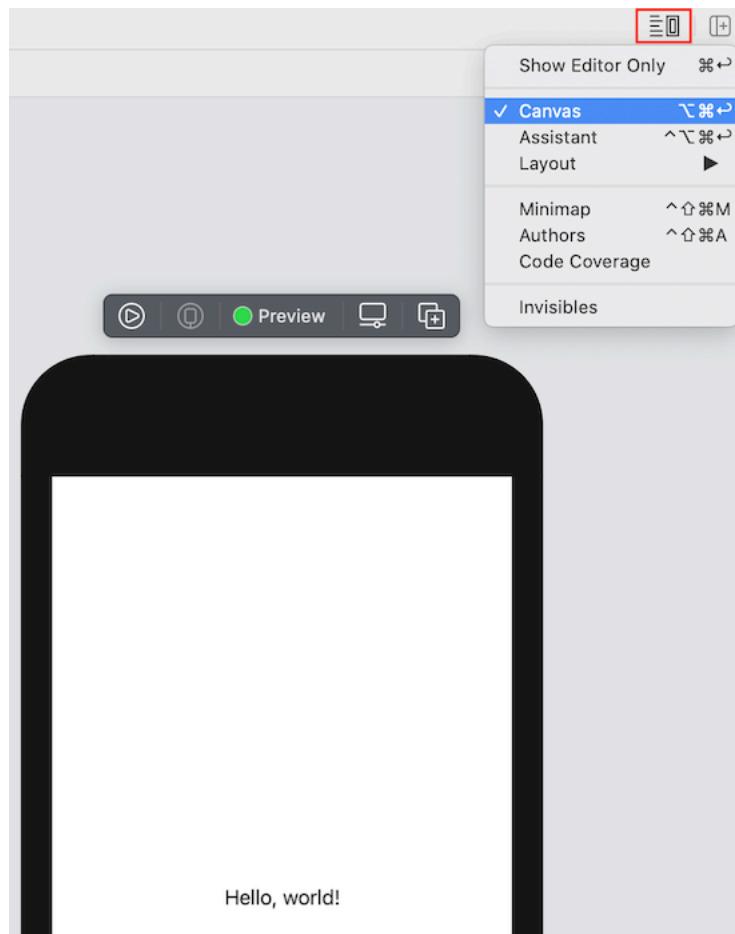
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

Ele contém uma *struct* chamada *ContentView*, que implementa o protocolo *View*. Tal protocolo define que a *struct* deve ter um atributo do tipo *View* com o nome de *body*. É nesse atributo que a tela será construída com **SwiftUI**.

No *template* criado há apenas uma caixa de texto exibindo a mensagem `Hello, world!`, o que pode ser visto no Xcode Preview, como mostra a figura a seguir:



Primeira tela da aplicação



Caso não esteja visualizando o Xcode Preview, selecione a opção `Adjust Editor Options`, localizado no canto superior direito do Xcode, como ressaltado na figura anterior, e selecione a opção `Canvas`.

Essa tela foi então construída com o conteúdo da variável `body` da `struct ContentView`. Mais adiante outros componentes serão adicionados nele para compor a tela desejada, como mencionado no início desse capítulo.

Ainda nesse arquivo há uma outra `struct` de nome `ContentView_Previews` que implementa o protocolo `PreviewProvider`, como pode ser visto no trecho a seguir:

```
struct ContentView_Previews: PreviewProvider {  
    static var previews: some View {  
        ContentView()  
    }  
}
```

Esse trecho é o responsável por exibir a tela criada em ContentView no Xcode Preview, como pode ser observado na figura anterior. Cada arquivo criado com o template SwiftUI terá um Preview associado a ele para que possa também ser exibido no Xcode Preview.

Também é possível definir algumas configurações do dispositivo que será exibido no Xcode Preview, como seu tipo:

```
struct ContentView_Previews: PreviewProvider {  
    static var previews: some View {  
        ContentView()  
            .previewDevice(PreviewDevice(rawValue: "iPhone 11 Pro Max"))  
            .previewDisplayName("iPhone 11 Pro Max")  
    }  
}
```

Nesse exemplo, o dispositivo iPhone 11 Pro Max foi selecionado.

Da mesma forma, é possível adicionar mais de um dispositivo para ser exibido no Xcode Preview. Para isso, basta adicionar mais uma seção com o ContentView, como mostra o trecho a seguir:

```
struct ContentView_Previews: PreviewProvider {  
    static var previews: some View {  
        ContentView()  
            .previewDevice(PreviewDevice(rawValue: "iPhone 11 Pro Max"))  
            .previewDisplayName("iPhone 11 Pro Max")  
  
        ContentView()  
            .previewDevice(PreviewDevice(rawValue: "iPhone SE"))  
            .previewDisplayName("iPhone SE")  
    }  
}
```

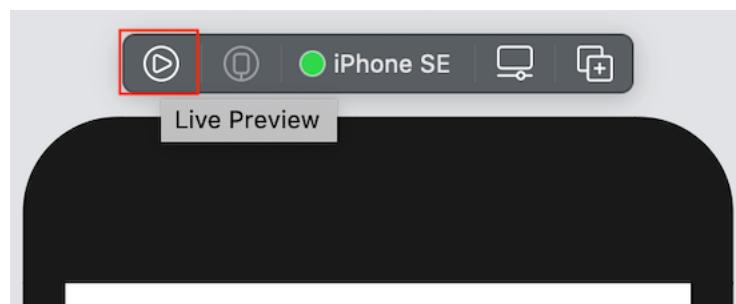
Isso fará com que os dois dispositivos apareçam no Xcode Preview, como pode ser visto na figura a seguir:



Xcode Preview com múltiplos dispositivos

Essa técnica pode ser utilizada para visualizar a tela em mais de um dispositivo ao mesmo tempo, enquanto ela é construída no código em Swift.

Para fazer com que a aplicação entre em execução no Xcode Preview, pressione o botão `Live Preview`, o primeiro da esquerda pra direita na barra de comandos logo acima do dispositivo, como pode ser observado na figura a seguir:



Executando o *live preview*

Essa ação colocará a aplicação em execução no *live preview*. Isso pode ser feito sempre que necessário para testar a aplicação, sem a necessidade de se executar um simulador de um dispositivo.

5.3 - Construindo uma interface básica com SwiftUI

Antes de efetivamente começar a criar a tela da aplicação proposta nesse capítulo, é conveniente apresentar alguns conceitos introdutórios do SwiftUI. Para isso, abra o arquivo `ContentView.swift` e vá até a definição da variável `body` da `struct ContentView`. Veja que ele define uma caixa de texto com a mensagem `Hello, world!`. A cor desse texto pode ser modificada adicionando a instrução `.foregroundColor()` logo abaixo da instrução que adiciona um *padding* a ela, como pode ser visto no trecho a seguir:

```
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
            .foregroundColor(.red)
    }
}
```

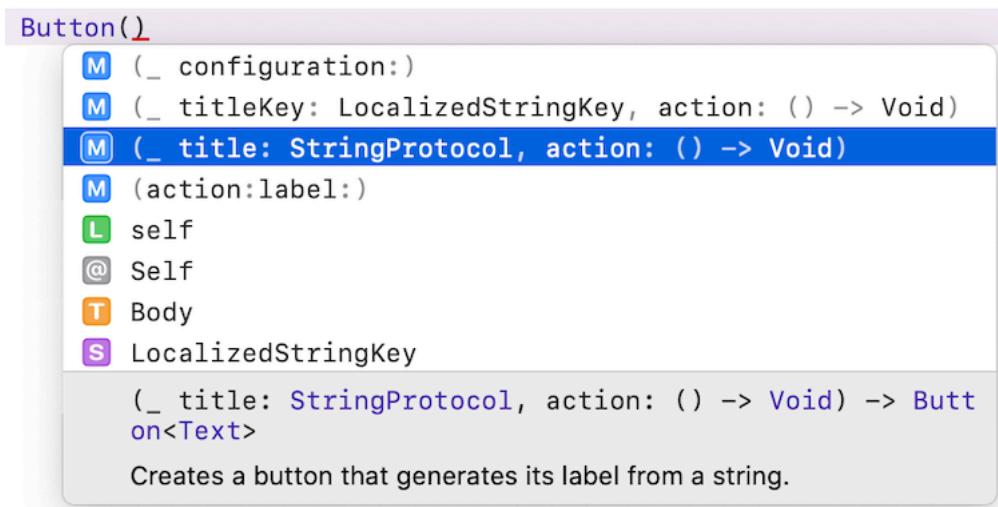
Nesse exemplo, a cor do texto foi alterada para vermelho, que pode ser observada imediatamente na tela sendo exibida no Xcode Preview.

Agora imagine que um botão deva ser adicionado logo abaixo da caixa de texto. Pois bem, a *view* que é definida pela variável `body` só pode ter um componente dentro dela, logo é necessário adicionar um outro componente, que irá gerenciar o layout da tela, responsável por distribuir a caixa de texto e o botão na vertical. Tal componente é o `VStack` ou *vertical stack*. Veja como ele pode ser criado:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, world!")
                .padding()
                .foregroundColor(.red)
        }
    }
}
```

Dessa forma a variável `body` continua tendo apenas um componente definido em sua raiz, que é o `VStack`.

Agora basta criar o botão logo após a definição da caixa de texto, dentro do componente `VStack`. Veja como é a sintaxe para a sua criação:



Criando um botão

Veja que o construtor de `Button` pode receber uma `String`, na verdade um `StringProtocol`, que será o nome aplicado ao título do botão que o usuário irá ver na tela, e uma ação, que será invocada para realizar a tarefa necessária quando o usuário clicar no botão. Essa ação pode ser definida em um *closure* e passada como parâmetro na criação do botão, ou melhor ainda, ser escrita na forma de um *trailling closure*, pois ele é o último parâmetro, como foi explicado no fim do capítulo anterior. Veja como pode ficar:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, world!")
                .padding()
                .foregroundColor(.red)

            Button("Click me") {
                print("Button clicked...")
            }
        }
    }
}
```

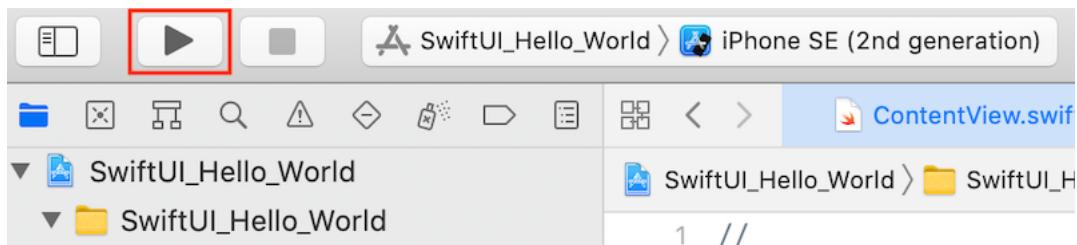
Perceba que a instrução a ser executada na ação de clique do botão está definida em um *closure*, criado logo após a definição desse botão.

Veja como esse código fica representado no Xcode Preview:



O Xcode também possui ferramentas onde os componentes podem ser criados e adicionados ao código através de ações em sua interface gráfica, além de comandos para encapsular componentes em outros componentes e adicionar modificadores.

Para testar essa aplicação, execute-a em um simulador do iOS, clicando no botão *play* localizado no canto superior esquerdo do Xcode:



Executando a aplicacão em um simulador

Como a ideia aqui é imprimir uma mensagem de log, é necessário executar a aplicação em um simulador do iOS, porém os demais testes a serem feitas nesse capítulo poderão ser realizados utilizando o Xcode Preview.

5.4 - Utilizando @State

Indo um passo a frente dessa simples interface com uma caixa de texto e um botão, imagine agora que a ação a ser realizada por esse botão fosse alterar o texto contido nessa mesma caixa onde está escrito `Hello, world!`. A maneira mais simples de se fazer isso é criando uma propriedade privada da `struct ContentView`, no qual a caixa de texto fique observando por mudanças em seu conteúdo. Dessa forma, quando algo mudasse o valor dessa propriedade, automaticamente essa caixa de texto seria atualizada com o novo valor.

Essa é uma técnica moderna do **SwiftUI**, permitindo a criação de um código conciso e fácil de se entender. Para isso, basta criar tal atributo privado decorado com `@State`. Veja como deve ficar sua declaração no início da `struct ContentView`:

```
struct ContentView: View {  
    @State private var text: String = "Hello, world!"
```

Perceba que o valor inicial da variável já é definido com o texto que deverá ser mostrado quando a *view* aparecer na primeira vez.

Dessa forma, pode-se criar a caixa de texto para exibir o conteúdo desse atributo, como mostra o trecho a seguir:

```
Text(text)  
    .padding()  
    .foregroundColor(.red)
```

Veja que ao invés de atribuir um texto estático à caixa de texto, é colocada a variável *text*. Como esse atributo está decorado com `@State`, o SwiftUI irá atualizar essa caixa de texto sempre que o valor desse atributo se alterar.

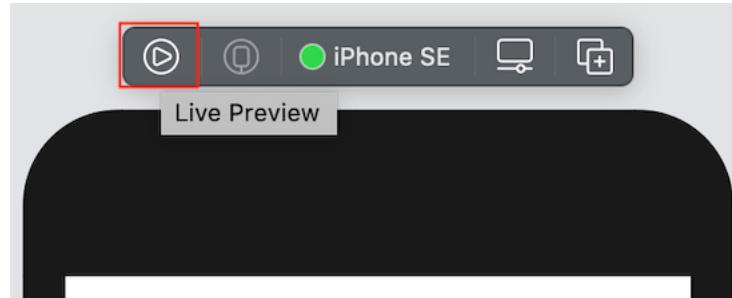
E para alterar seu valor quando o botão for clicado pelo usuário, simplesmente altere o valor do atributo *text* dentro do *closure action* desse botão, como mostra o trecho a seguir:

```
Button("Click me") {  
    text = "Button clicked!"  
}
```

Veja como deve ficar o código completo:

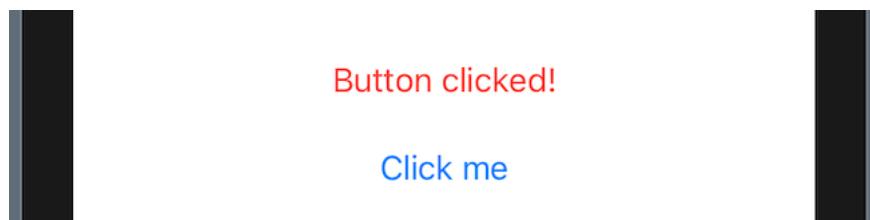
```
struct ContentView: View {  
    @State private var text: String = "Hello, world!"  
  
    var body: some View {  
        VStack {  
            Text(text)  
                .padding()  
                .foregroundColor(.red)  
  
            Button("Click me") {  
                text = "Button clicked!"  
            }  
        }  
    }  
}
```

Para testar essa implementação, pressione o botão `Live Preview`, como mostra a figura a seguir:



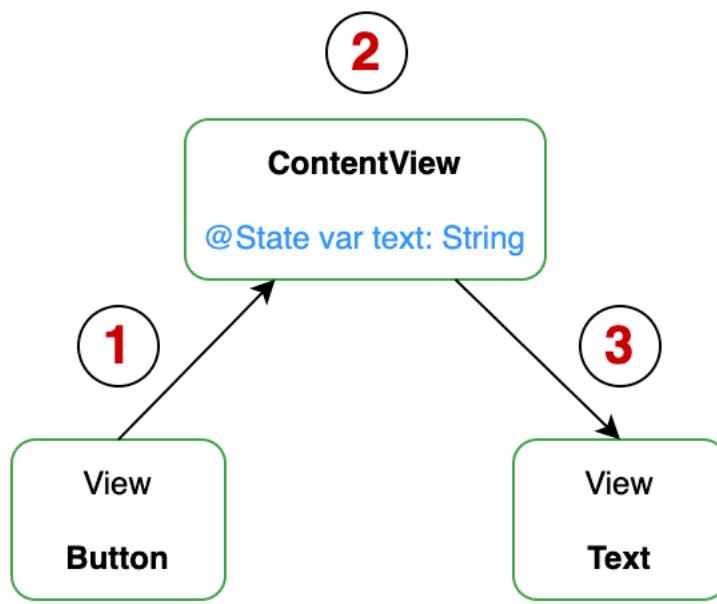
Testando no *live preview*

Depois que a aplicação estiver rodando no `Live Preview`, clique no botão e veja que o texto será alterado para o valor `Button clicked!`, como mostra a figura a seguir:



Testando o comportamento do botão

Veja que apenas alterando o valor da variável `text` foi possível alterar o conteúdo da caixa de texto. Isso porque o atributo `text` está decorada com `@State`, o que faz com essa mesma caixa de texto seja redesenhada quando há alguma alteração no valor por ela **observada**, como mostra o diagrama a seguir:



Atualização da view através de State

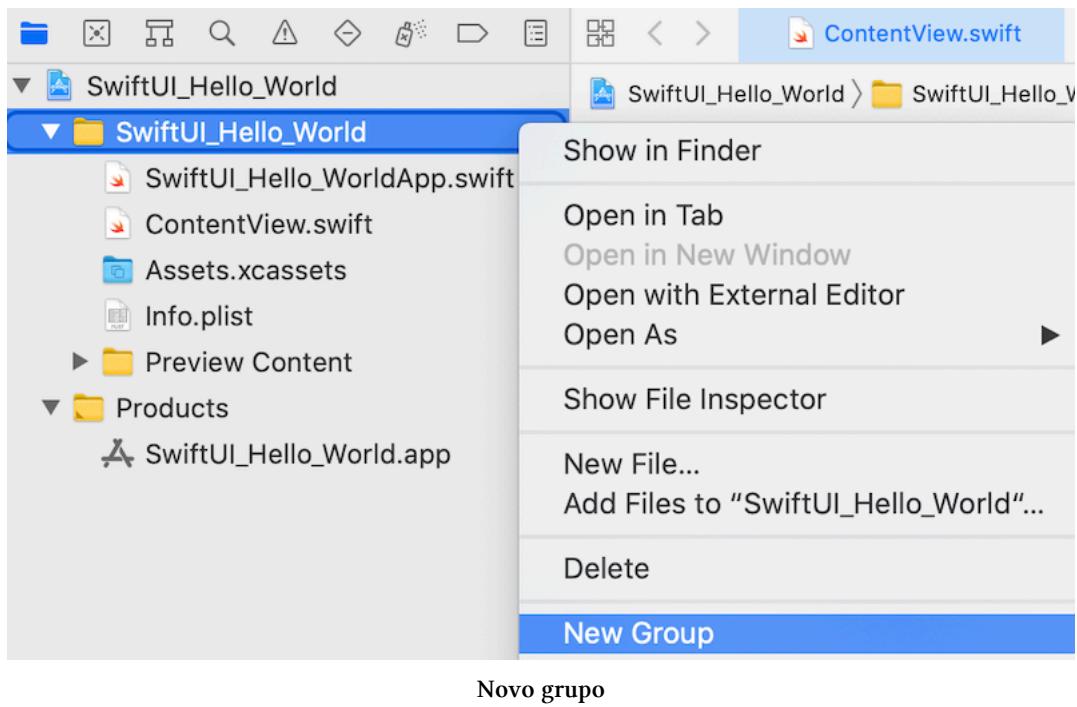
A tela que é exibida ao usuário é uma *view*, construída pela *struct ContentView*, assim como os componentes que estão nela também são *views*, como o botão e a caixa de texto. Esse último observa qualquer alteração no valor do atributo *text* que está decorado com *@State*.

Na figura anterior, no passo 1, quando o botão é clicado pelo usuário, o valor do atributo *text* é alterado para *Button clicked!*, o que faz com que *ContentView* perceba, em 2, que seu atributo *text* teve seu valor alterado. Esse por sua vez redesenha a caixa de texto em 3, alterando o valor exibido ao usuário.

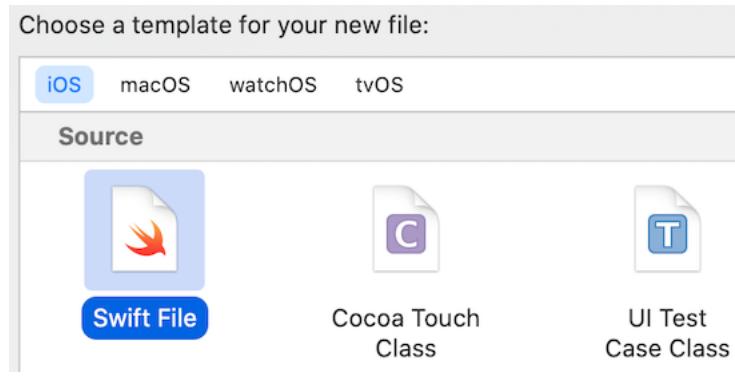
Mais adiante, ainda nesse capítulo, será detalhado um outro conceito adjacente ao explicado nessa sessão, quando um atributo pode ser anotado com *@Binding*, que permite que *views* “filhas” possam compartilhar o controle de atributos decorados com *@State* de sua *view* “mãe”.

5.5 - Criando o modelo de produtos

Para continuar com a proposta desse capítulo, que é criar uma tela para um certo tipo de cadastro de um produto apenas, é necessário criar um modelo que o represente. Para isso, crie um novo grupo chamado *Models* dentro da pasta principal do projeto:



Dentro dele, crie um novo arquivo chamado Product, seguindo o *template* iOS -> Swift File:



Dentro desse novo arquivo, crie uma *struct* chamada Product, contendo os atributos necessários para representar o produto, como no trecho a seguir:

```
struct Product {
    var name: String
    var description: String
    var code: String
    var price: Double
}
```



Como a ideia é apenas criar um modelo que representa o produto, sem a necessidade de se fazer herança com qualquer outro modelo, o ideia aqui é utilizar *struct* para representá-lo.

Veja que não é necessário criar um inicializador para os atributos de *Product*. O fato do modelo ter sido construído utilizando *struct* já permite que o inicializador padrão exista para atribuir os valores a todos os atributos, como será visto mais adiante na sua primeira utilização.

Como o modelo de *Product* não terá nenhuma função adicional em seu comportamento, ele está finalizado para ser utilizado.

5.6 - Criando a tela de cadastro do produto

Para começar a criar a tela de cadastro de produto, volte ao arquivo *ContentView.swift* e apague o atributo privado de nome *text*, assim como o conteúdo de *body*, como mostra o trecho a seguir:

```
struct ContentView: View {  
  
    var body: some View {  
  
    }  
}
```

Não se preocupe com os erros nesse momento. À medida em que a tela for sendo reconstruída eles desaparecerão.

Agora crie um atributo privado chamado *product*, do mesmo tipo da *struct* criada na sessão anterior, como no trecho a seguir:

```
struct ContentView: View {  
    @State private var product = Product(name: "", description: "", code: "", price: 0.0)  
  
    var body: some View {  
  
    }  
}
```

Veja que o atributo deve estar decorado com *@State*, pois ele será utilizado como **fonte de dados** para as caixas de texto que coletaram e exibirão seu conteúdo, a serem criadas mais adiante.

Como a ideia é criar um formulário para o usuário digitar as informações do produto, é conveniente então utilizar o componente *Form* para começar a montar tal tela. Dentro do *Form* deve-se criar seções,

que servem para dividir a tela de acordo com algum contexto. Cada seção pode ou não ter um título em seu cabeçalho.

Nessa tela serão criadas 3 seções, como descritas a seguir:

- A primeira seção a ser criada é a que abrigará os campos para o usuário digitar as informações do produto;
- A segunda conterá apenas o botão para que o usuário salve o produto;
- A terceira e última seção deverá conter as informações do produto salvo, que só deverá aparecer depois do processo de validação das informações digitadas.

Veja como deve ficar a primeira seção:

```
var body: some View {
    Form {
        Section(header: Text("Product")) {
            VStack(alignment: .leading) {
                Text("Name:")
                TextField("Enter the name", text: $product.name)
            }

            VStack(alignment: .leading) {
                Text("Description:")
                TextField("Enter the description", text: $product.description)
            }

            VStack(alignment: .leading) {
                Text("Code:")
                TextField("Enter the code", text: $product.code)
            }
        }
    }
}
```

Perceba que a *view* criada dentro da variável *body* continua com apenas um elemento raiz, que é o *Form*. Dentro dele, a primeira seção foi criada com o título *Product*, já com os componentes para o usuário digitar o nome, descrição e código do produto, agrupados com o *VStack*, cada um deles contendo uma caixa de texto (*Text*) e um campo editável (*TextField*).



O campo para o usuário digitar o preço do produto possui algumas peculiaridades e será criado mais adiante.

Cada *TextField* é criado com dois argumentos:

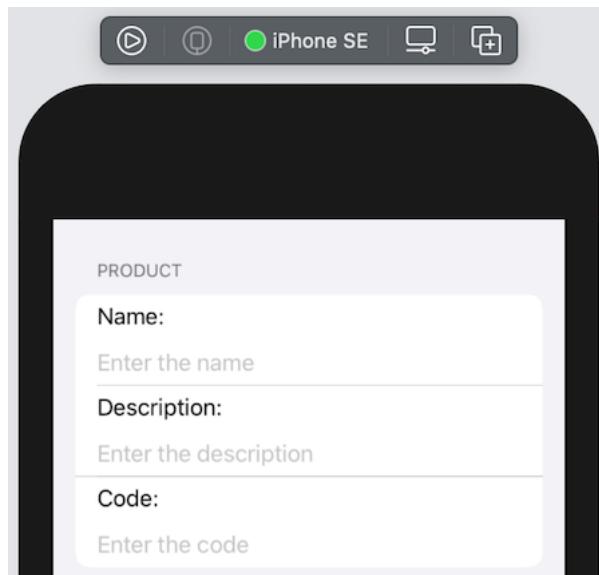
- Texto a ser apresentado ao usuário, até que ele preencha o campo, ou seja, um *placeholder*;
- Uma variável que permita uma **ligação em duplo sentido**, ou seja, que possa ler sua informação, assim como possa escrever nela.

Perceba que o segundo parâmetro passado em cada `TextField` é o campo equivalente à informação que ele representa, proveniente do atributo `product` criado em `ContentView`, com um sinal `$` antecedendo-o. Isso indica ao SwiftUI que tal informação representa, na verdade, uma variável de estado, ou seja, anotada com `@State`, em que é possível escrever informações nela. Isso significa que, quando o usuário digitar algum valor no `TextField`, ele será alterado também dentro do campo correspondente da variável `product`.



O conceito de *Binding*, que permite essa ligação em duplo sentido, ainda será melhor explicado mais adiante ainda nesse capítulo.

Veja como a tela deve ficar, até o momento:



Tela parcial de cadastro de produtos

Repare que o formulário de cadastro de produto já está ganhando forma com pouco código escrito até o momento.

5.7 - Salvando o produto

A ideia de salvar o produto na verdade é apenas fazer uma validação simples e exibi-lo nessa mesma tela ao usuário, em uma outra sessão. Para isso, é necessário criar uma função capaz de fazer tal verificação simplória e dizer se o produto informado pelo usuário é válido ou não para ser salvo. Além

disso, seria interessante fornecer alguma indicação sobre qual campo não está com a informação correta. A ideia é fazer uma validação da quantidade de caracteres do campo código.

Dessa forma, é necessário criar duas variáveis do tipo `Bool`: uma para dizer se o produto pode ou não ser salvo e outra para dizer se o campo do código do produto está válido. Mais adiante, ambos serão utilizados para controlar o comportamento da interface gráfica, logo é interessante que eles sejam criados já decorados com `@State`, pois na verdade eles vão representar estados da `view`. Veja como deve ficar no trecho a seguir:

```
struct ContentView: View {
    @State private var product = Product(name: "", description: "", code: "", price: 0.0)
    @State private var isValidCode = true
    @State private var isProductSaved = false
```

Veja que tais variáveis, `isValidCode` e `isProductSaved` já foram inicializadas com valores que fazem sentido para o momento em que a `view` aparecer pela primeira vez.

Agora é necessário criar uma função que será invocada pelo clique do botão `Save`, que ainda será criado. Ela deve ser responsável pelas validações das informações do produto e dizer se ele pode ou não ser salvo. Para isso, crie essa função logo abaixo da declarações das variáveis:

```
private func saveProduct() {
    isValidCode = product.code.count >= 5
    isProductSaved = isValidCode
}
```

Perceba que na primeira linha é feita uma validação para saber se o código do produto possui 5 caracteres ou mais. Caso positivo, a segunda linha dirá que o produto poderá ser salvo. Mais adiante a validação de preço também será feita, então esse código ainda será modificado, deixando mais claro a necessidade de mais de uma variável de controle.

Para criar o botão de `Save` e invocar essa função, crie uma nova seção dentro do `Form`, logo abaixo da seção já existente, como no trecho a seguir:

```
Section {
    Button(action: saveProduct) {
        Text("Save")
    }
}
```

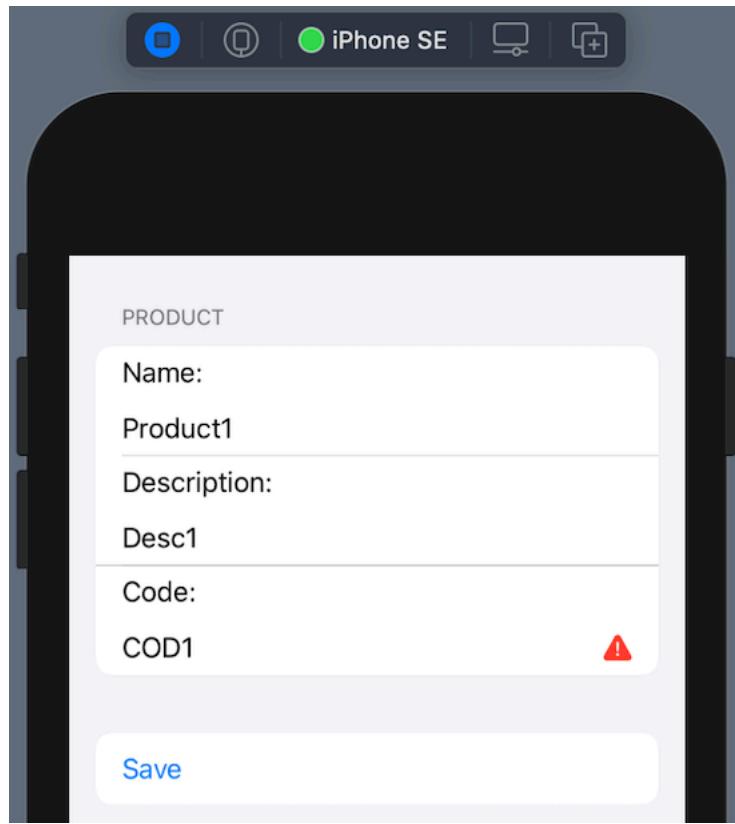
Como a execução dessa função altera a variável de estado `isValidCode`, caso o código do produto não seja válido, obviamente, ela então pode ser utilizada para informar ao usuário desse fato. Isso

pode ser feito adicionando uma imagem de erro do lado direito do campo onde o código do produto deve ser digitado, condicionado ao valor da variável `isValidCode`, ou seja, se estiver com valor `false`, a imagem é exibida.

Para conseguir tal efeito, primeiramente é necessário envolver o `TextField` do código do produto em um `HStack`, que permite que componentes fiquem dispostos na horizontal, além de adicionar a imagem, condicionada ao valor de `isValidCode`, como mostra o trecho a seguir:

```
HStack {  
    TextField("Enter the code", text: $product.code)  
    if (self.isValidCode == false) {  
        Image(systemName: "exclamationmark.triangle.fill").foregroundColor(Color.red)  
    }  
}
```

Para testar essa implementação, execute a aplicação no `Live Preview`, digite um código do produto com menos de 5 caracteres e pressione o botão `Save`. Isso vai fazer com que a função `saveProduct` seja executada, que por sua vez valida a quantidade de caracteres do código do produto, que nesse caso estará inválida. Nesse momento a variável de estado `isValidCode`, que foi inicializada com `true`, passará a conter o valor `false`. Como essa variável teve seu valor alterado, a `view` será redesenhada, fazendo com que o sinal de erro apareça do lado do campo de código, como pode ser observado na figura a seguir:



Código do produto inválido

Para complementar e reforçar a ideia de que a `view` é atualizada em toda mudança de valor em uma variável de estado, preencha o campo código do produto com um valor com mais de 5 caracteres e pressione o botão `Save` novamente. A função `saveProduct` será chamada e como agora o código será válido, a variável `isValidCode` será alterado para `true`, fazendo com que a imagem de erro desapareça.

Essa mesma estratégia será utilizada para exibir o produto salvo, quando ele estiver válido, como será detalhado na próxima seção.

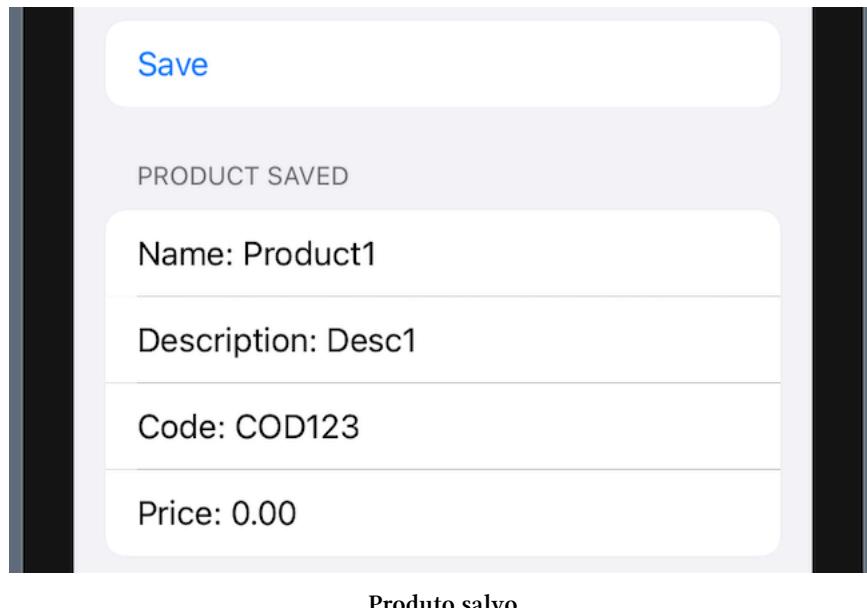
5.8 - Exibindo o produto salvo

A ideia para exibir o produto salvo, depois que ele for validado pela função `saveProduct`, é construir uma nova seção nessa `view` que terá sua exibição inteira condicionada ao valor da variável de estado `isProductSaved`. A ideia é criar uma caixa de texto para cada campo do produto e organizá-la dentro do novo componente `Section`, como pode ser visto no trecho a seguir. Esse trecho deve ser colocado logo abaixo do componente `Section` onde o botão de `Save` foi criado, ainda dentro do componente `Form`:

```
if (self.isProductSaved) {  
    Section(header: Text("Product Saved")) {  
        Text("Name: \(product.name)")  
        Text("Description: \(product.description)")  
        Text("Code: \(product.code)")  
        Text("Price: \(product.price, specifier: "%.02f")")  
    }  
}
```

Veja que o campo do preço do produto está formatado para aparecer com apenas 2 casas decimais.

Para testar essa implementação, execute novamente a aplicação no Live Preview, preencha todos os campos do produto e clique no botão Save. Se o código do produto for válido, a nova seção deverá aparecer como a figura a seguir:



Apesar do campo para o cadastro do preço do produto ainda não ter sido construído, ele já está presente no modelo Product e pode ser utilizado.

5.9 - Refatorando as views e utilizando @Binding

O código está funcional, mas ele pode ficar um pouco mais bem organizado. Uma coisa que poderia ser feito é separar algumas porções da tela em *subviews*, como por exemplo as seções de cadastro do produto e a que exibe o produto já validado e cadastrado. Isso deixaria a *struct ContentView* mais limpa e mais fácil de ser entendida. Essa técnica de subdividir uma *view* em outras menores é incentivado, pois não gera custo de processamento adicional.

Para começar, crie uma nova *struct*, dentro do mesmo arquivo `ContentView.swift`, chamada `ProductSavedView`, implementando o protocolo `View`. Dentro dela, crie um atributo do tipo `Product`, que receberá o produto salvo a ser exibido. Na implementação da variável `body`, transporte a seção de título `Product Saved` para dentro da construção dessa variável. Veja como deve ficar no trecho a seguir:

```
struct ProductSavedView: View {
    var product: Product

    var body: some View {
        Section(header: Text("Product Saved")) {
            Text("Name: \(product.name)")
            Text("Description: \(product.description)")
            Text("Code: \(product.code)")
            Text("Price: \(product.price, specifier: "%.02f")")
        }
    }
}
```



Essa *struct* pode ser criada no mesmo arquivo `ContentView.swift`, desde que seja fora de qualquer outra *struct*, obviamente.

Essa nova *view* implementada por `ProductSavedView` agora pode ser instanciada dentro do bloco `if` que testa se o produto foi salvo ou não, dentro do componente `Form` de `ContentView`, como mostra o trecho a seguir:

```
if (self.isProductSaved) {
    ProductSavedView(product: self.product)
}
```

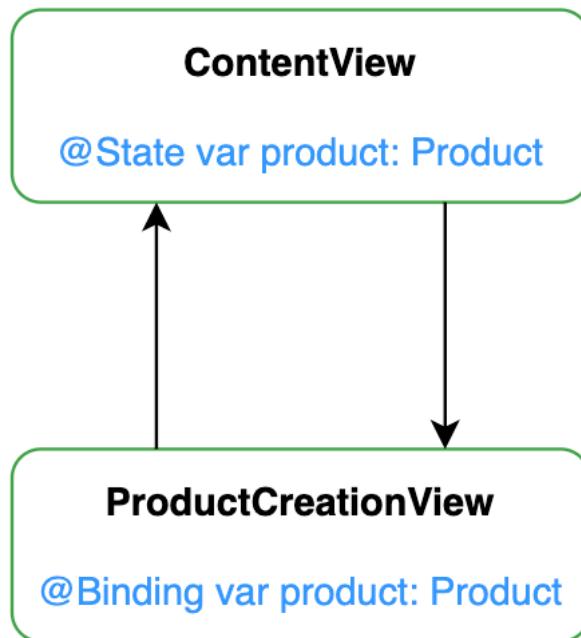
Perceba que como a *struct* `ProductSavedView` possui um atributo do tipo `Product`, é possível então passar o **valor do atributo** `product` que foi preenchido em `ContentView`. Como dito no capítulo anterior, *structures* são passadas como valor, logo o atributo da *struct* `ProductSavedView` é uma cópia da instância `product` de `ContentView`.

Ainda é possível fazer outra melhoria em `ContentView`, que é transformar a seção que cadastra o produto em uma outra *view*, seguindo o mesmo exemplo que foi feito anteriormente. Porém, aqui existem algumas peculiaridades interessantes que essa nova *view* precisa, que se chamará `ProductCreationView`:

- A variável de estado que diz que o código do produto é válido ou não precisa ser passada para controlar a exibição da imagem de erro;

- O atributo `product` de `ContentView` precisa ser fornecido como parâmetro para `ProductCreationView`, com a possibilidade que ela escreva na mesma instância, através dos componentes `TextField`.

Como o atributo `product` de `ContentView` é uma variável de estado e `ProductCreationView` precisa alterar seu valor, é necessário criar uma ligação de **duplo sentido** entre entre os dois atributos do tipo `Product` dessas *views*. Isso pode ser conseguido decorando o atributo do tipo `Product` de `ProductCreationView` com `@Binding`:



Criando a associação com `@Binding`

Se `ProductCreationView` alterar, por exemplo, o valor da descrição do produto de dentro dela mesma, essa alteração deverá ser refletida na instância `product` de `ContentView`, criando assim a ligação de duplo sentido entre esse atributo das duas *views*.

Veja como deve ficar a *struct* `ProductCreationView`, que pode ser criada dentro do mesmo arquivo `ContentView.swift`, assim como foi feito com `ProductSavedView`:

```

struct ProductCreationView: View {
    @Binding var product: Product
    var isValidCode: Bool

    var body: some View {
        Section(header: Text("Product")) {
            VStack(alignment: .leading) {
                Text("Name:")
                TextField("Enter the name", text: $product.name)
            }
        }
    }
}

```

```
        }
    }

    VStack(alignment: .leading) {
        Text("Description:")
        TextField("Enter the description", text: $product.description)
    }

    VStack(alignment: .leading) {
        Text("Code:")
        HStack {
            TextField("Enter the code", text: $product.code)
            if (self.isValidCode == false) {
                Image(systemName:"exclamationmark.triangle.fill").foreground\\
Color(Color.red)
            }
        }
    }
}
```

Perceba que a novidade aqui é a criação do atributo product decorado com @Binding. Isso trará uma diferença na passagem desse parâmetro quando uma instância de ProductCreationView tiver que ser criada.

Agora substitua o primeiro componente `Section` de `Form` de `ContentView` pela criação da instância de `ProductCreationView`, como no trecho a seguir, que mostra todo o conteúdo da variável `body` de `ContentView`:

```
var body: some View {
    Form {
        ProductCreationView(product: $product, isValidCode: isValidCode)

        Section {
            Button(action: saveProduct) {
                Text("Save")
            }
        }

        if (self.isProductSaved) {
            ProductSavedView(product: self.product)
        }
    }
}
```

Veja que é necessário passar o primeiro atributo `product` para `ProductCreationView` com o sinal de `$`, pois seu construtor na verdade espera um parâmetro do tipo `Binding<Product>`, como pode ser visto na figura a seguir:

```
var body: some View {
    Form {
        ProductCreationView(product: $product, isValidCode: isValidCode)
            .product(isValidCode:)
```

Passando um atributo do tipo Binding

Teste novamente a aplicação e veja que tudo continua funcionando como antes.

Obviamente a criação de `ProductCreationView` foi intencional para mostrar como uma variável de estado, decorada com `@State` pode ser passada como parâmetro para uma outra `view`, com poderes de alterar seu valor, desde que decorada com `@Binding`, criando assim a chamada ligação de duplo sentido, onde uma alteração em uma `view` reflete na outra `view`.

5.10 - Criando o campo para o preço do produto

Como dito anteriormente, a criação do campo para a inserção do preço do produto possui alguns desafios, mas antes disso, é necessário preparar o restante do código para poder fazer a validação de seu conteúdo, por exemplo, verificando se o valor é maior do que zero. Para isso, volte até a definição dos atributos de `ContentView` e adicione uma nova variável de estado, como no trecho a seguir:

```
@State private var isValidPrice = true
```

Agora altere a função `saveProduct` e acrescente a validação do preço, como no trecho a seguir:

```
private func saveProduct() {
    isValidCode = product.code.count >= 5
    isValidPrice = product.price >= 0

    isProductSaved = isValidCode && isValidPrice
}
```

Isso fará com que o produto seja salvo somente se seu código e preço forem válidos.

Para o campo de preço, é necessário fazer a conversão da `String` digitada pelo usuário em um valor do tipo `Double`, mantendo a ligação de duplo sentido com a variável de estado `product`. Esse fato

associado traz algumas complicações na implementação, fazendo com que na verdade tenha que ser criada uma *view* especial para tratar esse cenário conjugado. O código para isso é consideravelmente avançado para esse ponto do livro, mas a essência do que deve ser feito é o que foi explicado nesse parágrafo. Veja como deve ficar o código, que deve ser inserido nesse arquivo:

```
import Combine
struct PriceField : View {
    @State private var enteredValue: String = ""
    @Binding var value: Double
    var isValid: Bool

    var body: some View {
        HStack {
            TextField("", text: $enteredValue)
                .onReceive(Just(enteredValue)) { typedValue in
                    if let newValue = Double(typedValue) {
                        self.value = newValue
                    }
                }.onAppear(perform: { self.enteredValue = String(format: "%.02f", self.value) })
                .keyboardType(.decimalPad)

            if (self.isValid == false) {
                Image(systemName: "exclamationmark.triangle.fill").foregroundColor(Color.red)
            }
        }
    }
}
```

Alguns conceitos ainda precisariam ser introduzidos para o completo entendimento desse código, mas esse ainda não é o momento para isso.

Com isso, *PriceField* pode ser utilizado da mesma forma como um *TextField*, por isso volte à *struct ProductCreationView* e acrescente mais um componente *VStack* ao final de *Section*, como no trecho a seguir:

```
VStack(alignment: .leading) {
    Text("Price:")
    PriceField(value: $product.price, isValid: isValidPrice)
}
```

Veja que agora é necessário declarar um novo atributo para dizer se o preço é válido ou não, de nome

`isValidPrice`. Faça isso adicionando-o juntamente com os demais atributos de `ProductCreationView` como no trecho a seguir:

```
struct ProductCreationView: View {
    @Binding var product: Product
    var isValidPrice: Bool
    var isValidCode: Bool
```

Com isso, será necessário adicionar a variável de estado `isValidPrice` na criação de `ProductCreationView`, da mesma forma que foi feito com `isValidCode`. Veja como deve ficar o código:

```
var body: some View {
    Form {
        ProductCreationView(product: $product,
                             isValidPrice: isValidPrice,
                             isValidCode: isValidCode)
```

Com essas alterações é possível testar novamente a aplicação, agora com o campo de preço. Perceba que se o código ou o preço estiverem inválidos, a imagem com o erro aparecerá ao lado do campo correspondente, caso contrário, o produto será salvo e exibido na última seção dessa *view*.

E isso finaliza a implementação desse projeto!

5.11 - Conclusão

Esse capítulo introduziu alguns conceitos de SwiftUI, na construção de uma interface gráfica simples, já demonstrando conceitos como `@State` e `@Binding` para controlar seu comportamento.

Muito ainda será visto de SwiftUI nesse livro, como construção de listas e navegação entre telas, como será visto no próximo capítulo, onde também será mostrado como consumir um Web service REST com autenticação OAuth.

6 - Consumindo serviços REST com Alamofire

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.

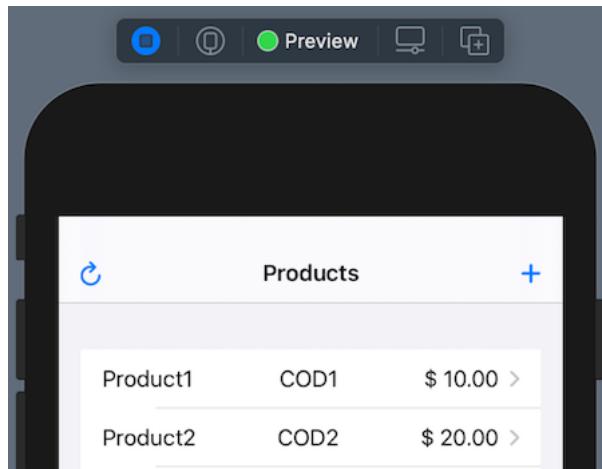
É muito comum que aplicações iOS necessitem consumir serviços REST para exibir informações ao usuário, seja como parte principal de suas funcionalidades, como em aplicativos de redes sociais, ou seja como funcionalidade adicional, como em apps de investimento onde alguma taxa ou cotação de ações deve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

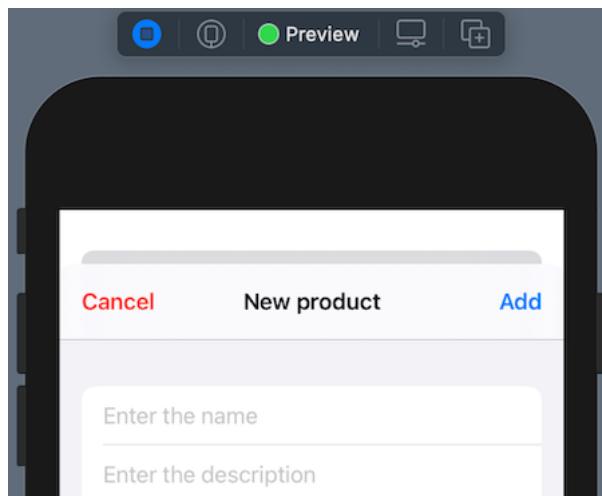
O objetivo desse capítulo é construir uma aplicação iOS capaz de consumir serviços REST de um provedor de serviços de vendas, detalhado na próxima seção. A ideia é **exibir uma lista de produtos previamente cadastrados**, além de seus detalhes.

A seguir algumas imagens do que será desenvolvido ao longo desse capítulo:



Lista de produtos

Nessa figura é possível ver como ficará a tela principal, com a lista dos produtos e barra de navegação. A figura a seguir mostra a tela de criação de um novo produto:



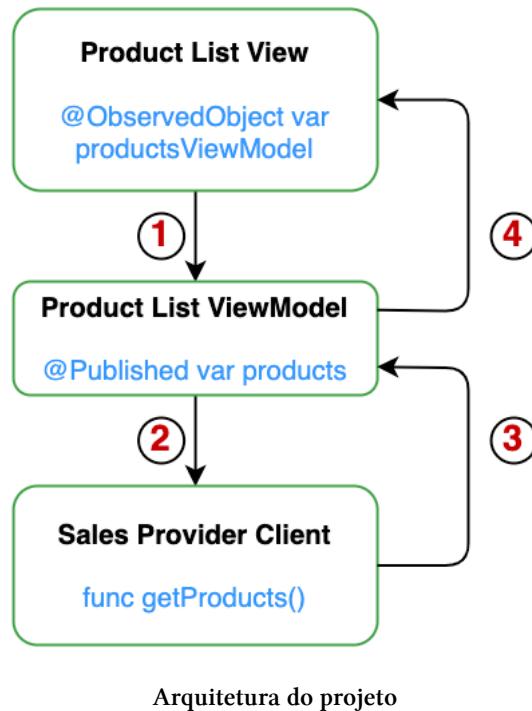
View para criação de um novo produto

Para consumir os serviços REST que serão apresentados na próxima seção, a aplicativo que será desenvolvido utilizará o [Alamofire¹⁰](#), uma biblioteca construída sobre o [URLSession¹¹](#) da Apple. Com o Alamofire será possível fazer requisições assíncronas e validações de forma simples e com um código elegante, como será mostrado ainda nesse capítulo.

A figura a seguir ilustra a arquitetura que será criada no projeto desse capítulo:

¹⁰<https://github.com/Alamofire/Alamofire>

¹¹https://developer.apple.com/documentation/foundation/url_loading_system



A ideia é que cada *view* observe seu *view model* por mudanças para atualizar a interface gráfica do usuário. Essa estratégia é particularmente interessante no cenário desse aplicativo, onde será feita uma requisição a um serviço REST, que dará sua resposta de forma assíncrona, sem bloquear a interface do usuário.

A seção 6.5 desse capítulo irá detalhar a construção dessa arquitetura e como as camadas irão interagir entre si.

6.1 - 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 iOS 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](#)¹². Seu código-fonte encontra-se nesse [repositório](#)¹³.



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:

¹²<https://www.casadocodigo.com.br/products/livro-gae>

¹³<https://github.com/siecola/GAESalesProvider>

[https://sales-provider.appspot.com¹⁴](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 [Postman¹⁵](#).

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 [repositório¹⁶](#).

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

6.1.1 - Diagrama do provedor de serviços de vendas

O provedor de serviços de vendas é uma aplicação Web construída em Java utilizando o *framework* [Spring Boot¹⁷](#) para ser hospedado na plataforma [Google App Engine¹⁸](#).

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;
- **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:

¹⁴<https://sales-provider.appspot.com/>

¹⁵<https://www.postman.com>

¹⁶https://github.com/siecola/GAESalesProvider/blob/master/postman/GAE_Sales.postman_collection.json

¹⁷<https://spring.io/projects/spring-boot>

¹⁸<https://cloud.google.com/appengine>

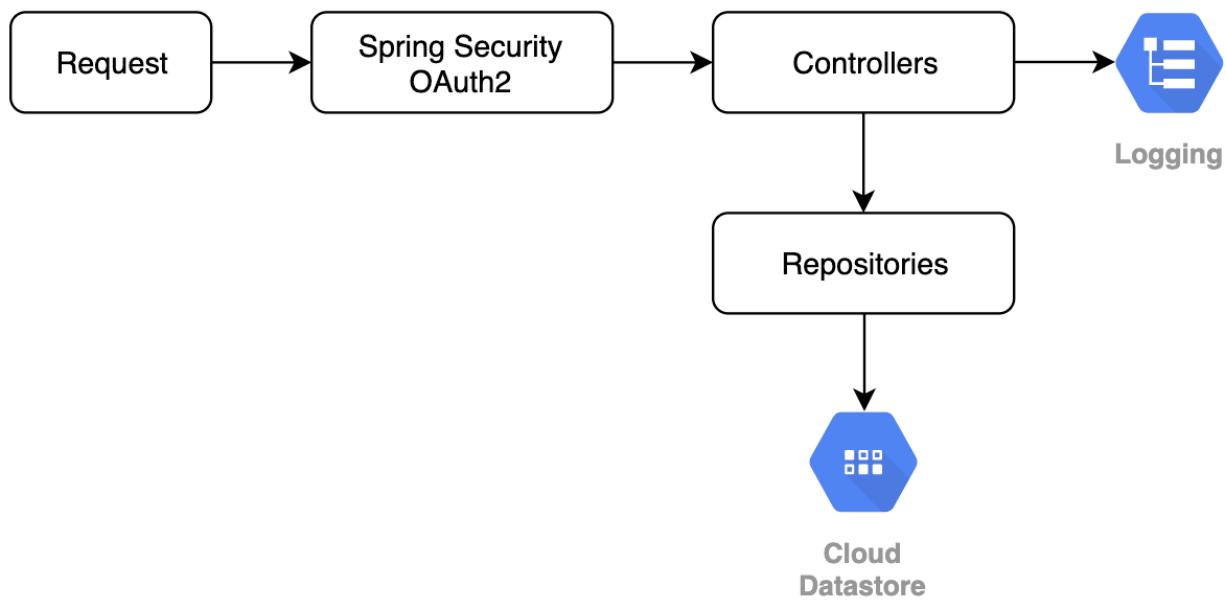


Diagrama do provedor de serviços de vendas

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.

6.1.2 - 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.

6.1.3 - 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:

Get Token

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

Params	Authorization	Headers (3)	Body	Pre-request Script	Tests	Settings						
Headers (3) <table border="1"> <thead> <tr> <th>KEY</th> <th>VALUE</th> </tr> </thead> <tbody> <tr> <td><input checked="" type="checkbox"/> Content-Type</td> <td>application/x-www-form-urlencoded</td> </tr> <tr> <td><input checked="" type="checkbox"/> Authorization</td> <td>Basic c2llY29sYTptYXRpbGRI</td> </tr> </tbody> </table>							KEY	VALUE	<input checked="" type="checkbox"/> Content-Type	application/x-www-form-urlencoded	<input checked="" type="checkbox"/> Authorization	Basic c2llY29sYTptYXRpbGRI
KEY	VALUE											
<input checked="" type="checkbox"/> Content-Type	application/x-www-form-urlencoded											
<input checked="" type="checkbox"/> Authorization	Basic c2llY29sYTptYXRpbGRI											

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.

The screenshot shows the Postman interface for a POST request to `https://sales-provider.appspot.com/oauth/token`. The 'Body' tab is selected, showing the raw JSON body:

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

The response body is displayed in Pretty mode:

```
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 }
```

The response is labeled **Token requisitado**.



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

6.1.4 - 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:

▶ Create User

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](#)¹⁹.



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



Para as operações de criação, alteração e exclusão, que serão implementadas no próximo capítulo, será necessário um usuário com papel ADMIN.

6.1.5 - 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 iOS, que será construída nesse capítulo. Com ela será possível obter

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

todos os produtos que estão cadastrados e exibí-los em uma lista na tela para o usuário.

A seguir, as informações necessárias para acessar essa operação:

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 [repositório²⁰](#).

6.2 - Criando e preparando o projeto

Para começar a implementação do app para o cliente do provedor de serviços de vendas, crie um novo projeto no Xcode de nome `SwiftUI_REST_Client`, seguindo os mesmos passos descritos no capítulo anterior.

Depois de criado o projeto, será necessário adicionar a biblioteca `Alamofire`, que irá facilitar a construção das requisições REST que serão feitas. Para isso, feche o Xcode e abra um terminal no OS X para instalar o Cocoapods:

```
sudo gem install cocoapods
```

Cocoapods é um gerenciamento de pacotes que pode ser utilizado para acrescentar bibliotecas a um projeto do Xcode.

Agora, no diretório raiz do projeto, onde está localizado o arquivo `SwiftUI_REST_Client.xcodeproj`, execute o seguinte comando para preparar o projeto:

²⁰<https://github.com/siecola/GAESalesProvider>

```
pod init
```

Esse passo irá criar um arquivo chamado `Podfile`, na raiz o projeto do Xcode. Nele é possível adicionar as bibliotecas desejadas para serem adicionadas ao projeto. Abra esse arquivo e substitua o conteúdo pelo trecho a seguir:

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, '14.0'

target 'SwiftUI_REST_Client' do
  use_frameworks!

  # Pods for SwiftUI_REST_Client
  pod 'Alamofire', '~> 5.2'

end
```

Repare na linha a seguir:

```
pod 'Alamofire', '~> 5.2'
```

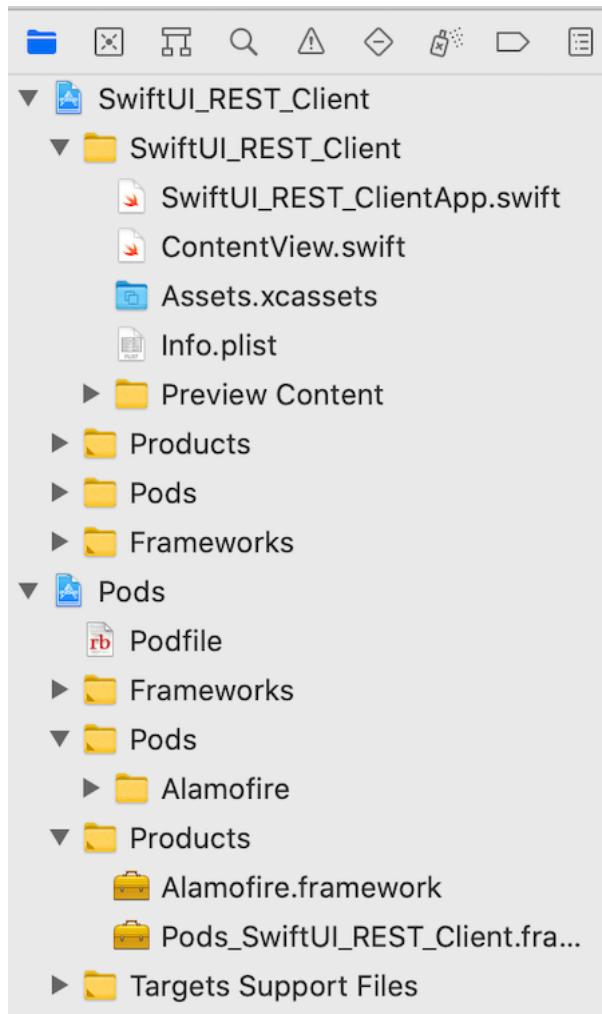
Ela é responsável por adicionar a dependência da biblioteca Alamofire ao projeto.

Salve e feche o arquivo `Podfile`. Voltando ao diretório raiz do projeto, execute o comando a seguir no terminal para efetivamente instalar o `Alamofire` no projeto :

```
pod install
```

Finalmente, para abrir o projeto configurado com a dependência instalada, é necessário abrir o `workspace` formado pelo projeto `SwiftUI_REST_Client` e as bibliotecas. Para isso, dê um duplo clique no arquivo `SwiftUI_REST_Client.xcworkspace`.

Veja como deve ficar a estrutura do projeto:



Estrutura do projeto



Nesse momento o projeto deverá compilar sem nenhum erro. Caso algum problema aconteça, verifique os passos anteriores para verificar se tudo foi feito corretamente.

6.3 - Criando as classes de modelo

Como visto nas seções anteriores, o aplicativo precisará buscar o token de acesso do usuário, para que ele possa ser utilizado em todas as requisições para ser autenticado pelo provedor de serviços de vendas. Para isso, será necessário criar modelos para:

- Representar a resposta da operação do provedor de serviços de vendas de requisição do token de acesso;
- Modelar uma abstração do mecanismo de gerenciamento do token de acesso, depois de obtido;

- Representar as credenciais de acesso do usuário;
- Representar a resposta da operação de listar os produtos.

Nesse exemplo, para manter a simplicidade do código, não serão mostrados os seguintes mecanismos:

- Persistência do token de acesso dentro do aplicativo, para que ele possa ser utilizado posteriormente. Ao invés disso, o token de acesso será armazenado em memória e estará válido enquanto o app estiver aberto, sendo necessário requisitá-lo sempre que for aberto;
- Criação de uma tela para o usuário informar suas credenciais de acesso. As informações ficarão fixas dentro do código fonte do aplicativo.

Os dois pontos citados acima não fazem parte do escopo desse capítulo, para tornar a explicação mais direta ao que o capítulo se propõe.

Para começar com a criação dos modelos citados acima, dentro do projeto do Xcode, crie um novo grupo/pasta chamado `WebServices`. Dentro dessa nova pasta ficarão toda a implementação relacionada ao cliente do provedor de serviços de vendas. Ainda dentro desse novo grupo, crie um outro chamado `Models`.

Dentro do grupo `Models`, comece um novo arquivo chamado `AccessTokenResponse` para representar a resposta do token de acesso que será fornecida pela operação citada na seção 6.1.3 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"
}
```

Como a ideia aqui é obter os valores dos campos `access_token`, `refresh_token` e `expires_in`, será então necessário representá-los no modelo `AccessTokenResponse`. Para isso, crie uma struct dentro desse novo arquivo, como o trecho a seguir:

```
import Foundation

struct AccessTokenResponse: Decodable {
    let access_token: String
    let refresh_token: String
    let expires_in: Int
}
```

Veja que cada propriedade da struct possui o mesmo nome e tipo de dado do campo que irá representar da resposta do token de acesso. Perceba também que a struct implementa o protocolo Decodable, o que facilita a interpretação de um JSON com esses campos e a criação de uma instância desse tipo para representar o token em si, algo que será feito na próxima seção.

O próximo modelo a ser criado é o que irá representar esse token do ponto de vista do mecanismo de controle. Basicamente ele precisa saber o valor do token em si e quando ele expira. Para isso crie um novo arquivo na pasta Models chamado AccessTokenStorage. Dentro desse novo arquivo, crie a classe de mesmo nome, como no trecho a seguir:

```
import Foundation

class AccessTokenStorage {
    var accesstoken: String
    var expiresDate: NSDate

    init (accesstoken: String, expiresIn: Int) {
        self.accesstoken = accesstoken
        self.expiresDate = NSDate().addingTimeInterval(
            TimeInterval(expiresIn))
    }
}
```



Essa é apenas uma abstração do modelo que poderia ser utilizado em um mecanismo para persistir o token de acesso, que, como já foi comentado anteriormente, não será implementado aqui.

Veja que agora foi utilizado uma classe ao invés de uma struct para criação desse último modelo. A razão para isso é permitir que uma instância desse tipo possa ser passada como referência, para que possa ser alterada por uma função que a receba, algo que será visto na seção seguinte.

Como o token de acesso recebido do provedor de vendas possui o parâmetro `expires_in`, que na verdade é a quantidade de segundos que o token é válido depois de ser requisitado, é necessário converter esse valor de tempo relativo em um valor absoluto, para que possa ser comparado a qualquer momento, algo que é feito na última linha do inicializador da classe `AccessTokenStorage`.

Ainda dentro dessa classe, crie uma função para dizer se o token ainda está válido ou não, ou seja, se o valor do campo `expiresDate` ainda aponta para um valor no futuro em relação ao momento dessa consulta:

```
func isValidToken() -> Bool {
    if NSDate().timeIntervalSinceReferenceDate <
        self.expiresDate.timeIntervalSinceReferenceDate {
        return true
    } else {
        return false
    }
}
```

O token será válido se o valor de `expiresDate` for maior do que o momento da consulta. Essa função será utilizada na seção seguinte pelos interceptadores de requisições do Alamofire, para saber se devem ou não utilizar um token existente.

O próximo modelo a ser criado é o que será utilizado para representar as credenciais do usuário. Para isso crie um novo arquivo dentro da pasta `Models` com o nome de `UserCredential`. Dentro dele, crie uma struct conforme o trecho a seguir:

```
import Foundation

struct UserCredential {
    let username: String
    let password: String
}
```

Essas informações serão utilizadas para a aquisição do token de acesso, como será mostrado na seção seguinte.

O último modelo a ser criado é o que vai representar o produto, presente na resposta do provedor de serviços de vendas quando eles forem consultados pelo aplicativo sendo desenvolvido aqui. Dentro da mesma pasta `Models`, crie um arquivo chamado `Product`, com o trecho a seguir:

```
import Foundation

struct Product: Codable {
    let id: UInt
    let name: String
    let description: String
    let code: String
    let price: Double
}
```

Veja que a struct `Product` também implementa o protocolo `Codable`, afinal, ela será utilizada para interpretar a resposta da operação de listar todos os produtos do provedor de serviços de vendas, que terá o seguinte formato:

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

Perceba que os campos do modelo representado pela struct Product possui os campos com os mesmos nomes e tipos do JSON mostrado anteriormente. Isso fará com que tal JSON possa ser deserializado facilmente.

6.4 - Criando o cliente REST com o Alamofire

Agora que todos os modelos já foram criados é possível construir a camada do cliente REST que efetivamente irá realizar as operações de buscar o token de acesso e a lista e produtos do provedor de serviços de vendas.

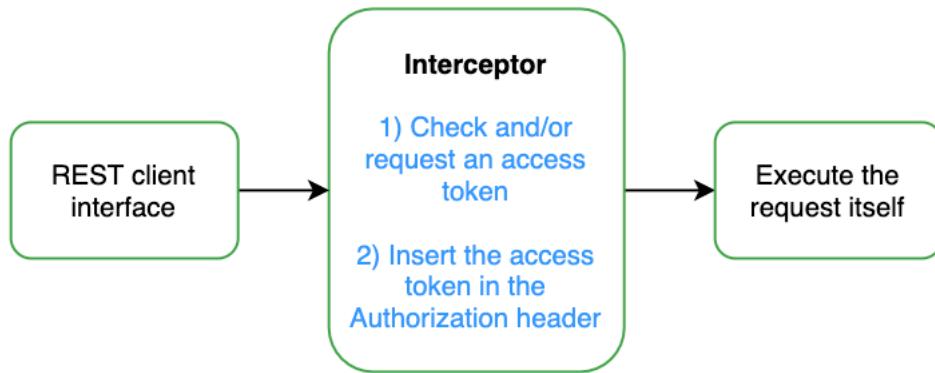
Esse cliente ainda irá evoluir um pouco mais no próximo capítulo, com a construção de outras operações, como a de criação de um novo produto.

6.4.1 - Detalhando a estratégia dos interceptadores de requisição com o Alamofire

Como explicado anteriormente, o provedor de serviços de vendas possui um mecanismo de autenticação OAuth 2.0. Isso significa que é necessário obter um token de acesso com as credenciais de um usuário previamente criado em sua base de dados para ter acesso às operações do serviço de produtos, utilizando o token de acesso, que deverá ser inserido no cabeçalho de nome Authorization em todas as requisições.

Esse token de acesso, uma vez requisitado ao provedor de serviços de vendas, possui uma validade máxima de 3600 segundos, ou seja, se um cliente requisitar o token, este deverá fazê-lo novamente após esse tempo, para que continue tendo suas requisições autenticadas e executadas pelo provedor de serviços de vendas.

Do ponto de vista da camada que implementa o cliente REST da aplicação, esse trabalho de verificar se um token de acesso válido existe e caso não exista, solicitar um novo, deveria ser feito toda vez que uma requisição ao provedor de serviços de vendas tivesse que ser realizada, independente de qual seja. Esse é um excelente argumento para implementar um mecanismo que intercepta todas as requisições e realiza esse trabalho, como mostra a figura a seguir:



Estratégia com interceptadores de requisições

Dessa forma, quando alguma parte da aplicação necessita executar alguma operação, como consultar todos os produtos, no provedor de serviços de vendas, a interface do cliente REST abstrai o mecanismo para verificar se existe um token válido ou peça um novo, caso ele não esteja válido, cuidando de inseri-lo no cabeçalho `Authorization` da requisição.

Felizmente, o Alamofire possui uma forma de criar tal mecanismo com interceptadores, fazendo com que todas as requisições o utilizem. Para começar, crie um novo arquivo chamado `AuthenticationInterceptor` dentro do grupo `WebServices` no projeto do Xcode. Dentro desse arquivo, crie uma classe de mesmo nome, implementando o protocolo `Alamofire.RequestInterceptor`, exigido pelo Alamofire para que a classe seja reconhecida com uma implementação válida de um interceptador. Veja como sua implementação inicial deve ficar:

```

import Foundation
import Alamofire

final class AuthenticationInterceptor: Alamofire.RequestInterceptor {
    private var accessTokenStorage: AccessTokenStorage?
    private let userCredential: UserCredential
    private let baseUrl: String

    init(accessTokenStorage: AccessTokenStorage?, userCredential: UserCredential,
         baseUrl: String) {
        self.accessTokenStorage = accessTokenStorage
        self.userCredential = userCredential
        self.baseUrl = baseUrl
    }
}

```

Veja que ela deve receber uma instância de `AccessTokenStorage`, para que possa salvar informações do token quando um novo for requisitado, assim como as credenciais do usuário, através do tipo

UserCredential, ambos criados na seção anterior. Essa classe também deve receber o endereço base do provedor de serviço de vendas, para poder acessar a operação de requisitar o token de acesso.

Seu construtor inicializa todos os atributos da classe, para que fiquem disponíveis nas função que serão implementadas a seguir.

6.4.2 - Criando a função para requisitar o token de acesso

A primeira função a ser implementada na classe AuthenticationInterceptor é a que verifica se o token existe e ainda é válido, como mostra o trecho a seguir:

```
private func isValidToken(accessTokenStorage: AccessTokenStorage?) -> Bool {
    if let accessToken = accessTokenStorage {
        return accessToken.isValidToken()
    } else {
        return false
    }
}
```

Essa função simplesmente verifica se o atributo accessTokenStorage é diferente de nulo e chama sua função para verificar a data de expiração do token, que deve ser maior do que a data/hora atual dessa operação, para que o token seja considerado válido.

A próxima função a ser implementada é que faz a requisição de um novo token, caso o existente já não seja válido. Comece-a criando sua assinatura, bem como algumas variáveis que serão utilizadas dentro dela, como no trecho a seguir:

```
private func getAccessToken(completion: @escaping (RetryResult) -> Void) {
    let tokenEndpoint = baseUrl + "oauth/token"

    let params : Parameters = ["grant_type": "password",
                               "username": userCredential.username,
                               "password": userCredential.password]
    let tokenHeaders: HTTPHeaders =
        ["Content-Type": "application/x-www-form-urlencoded",
         "Authorization": "Basic c211Y29sYTptYXRpbGR1"]

}
```

Perceba que a função getAccessToken não retorna nenhum valor. Isso porque a operação a ser realizada dentro dela com o Alamofire, para buscar tal token no provedor de serviços de vendas, é assíncrona. Por isso tal função recebe um closure como parâmetro, que deverá ser invocado quando essa operação assíncrona for concluída. Inclusive, essa é a razão para o closure estar anotado com @escaping:

```
private func getAccessToken(completion: @escaping (RetryResult) -> Void)
```

O parâmetro do tipo `RetryResult`, que será passado pelo closure quando for invocado, vem do mecanismo de retentativa do interceptador do Alamofire, que será explicado mais adiante.

A função `getAccessToken` também possui algumas variáveis, que são detalhadas a seguir:

- `tokenEndpoint`: esse é o endereço completo da operação para requisitar um token de acesso no provedor de serviços de vendas;
- `params`: aqui são colocadas as credenciais de acesso do usuário, para a obtenção de seu token de acesso;
- `tokenHeaders`: a implementação do provedor de serviços de vendas exige uma autenticação do cliente que solicita o token, por isso é necessário passar tal informação no campo `Authorization` quando essa operação for realizada.

Agora dentro dessa função, logo a pós a variável `tokenHeaders`, crie o código para efetivamente requisitar o token, como no trecho a seguir:

```
AF.request(tokenEndpoint, method: .post, parameters: params,
           encoding: URLEncoding.httpBody, headers: tokenHeaders)
    .validate(statusCode: 200..<300)
    .validate(contentType: ["application/json"])
    .responseDecodable(of: AccessTokenResponse.self) { response in
        switch response.result {
        case .success:
            if let accessTokenResponse = response.value {
                print("The token is: \(accessTokenResponse.access_token)")
                print("It expires in: \(accessTokenResponse.expires_in)")

                self.accessTokenStorage = AccessTokenStorage(
                    accessToken: accessTokenResponse.access_token,
                    expiresIn: accessTokenResponse.expires_in)

            }
            return completion(.retry)
        }
        case let .failure(error):
            print(error)
        }
    }
}
```

Na primeira linha do trecho anterior a função `request` do Alamofire é invocada, tendo como parâmetros o endereço da operação, o método, os parâmetros e os cabeçalhos definidos no início da implementação da função.

Logo em seguida, as função `validate` são chamadas para indicar ao Alamofire deve validar o código de retorno, que deve estar entre 200 e 300, além do tipo da resposta, que deve ser `application/json`. Essa estratégica de deixar que o Alamofire faça validações da resposta é interessante para deixar o código mais sucinto.

Ainda no processo da construção da requisição, a função `responseDecodable` é chamada para dizer ao Alamofire que o *payload* de resposta esperado é do tipo representado pelo modelo `AccessTokenResponse`. Isso traz uma grande vantagem ao restante do código, pois a resposta já estará deserializada dentro do parâmetro `response`.



O código dentro das chaves é executado somente quando a resposta é recebida do provedor de serviços de vendas, ou seja, é uma execução assíncrona.

Quando a resposta da requisição é obtida, é possível saber se ela foi ou não bem sucedida através do parâmetro `response.result`. E caso ela tenha sido realizada corretamente e passado pelas validações configuradas anteriormente, o conteúdo da resposta estará dentro de `response.value`, já deserializado para um objeto do tipo `AccessTokenStorage`, como pode ser visto na instrução `if let accessTokenResponse = response.value` que verifica se realmente existe algo nesse atributo.

Se o token foi obtido corretamente, ele então será salvo em `accessTokenStorage`, para ser reutilizado em uma requisição posterior. Nesse mesmo momento o `closure` recebido como parâmetro da função `getAccessToken` é chamado, informando ao Alamofire que a requisição que originou esse processo pode ser tentada novamente, agora com um token válido.

Caso o token não seja obtido, por exemplo por credenciais inválidas, o `closure` não será invocado e o Alamofire irá interromper a requisição em andamento.

6.4.3 - Criando as funções do interceptador de requisições

Agora as funções do interceptador, exigidas pelo protocolo `Alamofire.RequestInterceptor`, podem de fato serem construídas na classe `AuthenticationInterceptor`. A primeira delas é a função `adapt`. Veja sua assinatura no trecho a seguir:

```
func adapt(_ urlRequest: URLRequest, for session: Session,  
          completion: @escaping (Result<URLRequest, Error>) -> Void)
```

Essa função sempre será chamada pelo Alamofire antes da execução de qualquer requisição, sendo um bom momento para a inserção do token de acesso no cabeçalho HTTP, caso ele exista dentro da aplicação, para que a requisição possa ser autorizada pelo provedor de serviços de vendas.

Dessa forma, veja como deve ficar a implementação completa dessa função:

```
func adapt(_ urlRequest: URLRequest, for session: Session,
          completion: @escaping (Result<URLRequest, Error>) -> Void) {
    var urlRequest = urlRequest

    if isValidToken(accessTokenStorage: accessTokenStorage) {
        urlRequest.setValue("Bearer " + accessTokenStorage!.accesstoken,
                            forHTTPHeaderField: "Authorization")
    }

    completion(.success(urlRequest))
}
```

Veja que, dentre os parâmetros que ela recebe, dois são importantes para a análise de seu funcionamento. O primeiro deles é o parâmetro do tipo `URLRequest`. Ele representa a requisição que de fato será feita, e onde o cabeçalho `Authorization` deve ser inserido.

O outro parâmetro de relevância é o closure de nome `completion`, que será invocado depois que a requisição estiver adaptada ter sua execução continuada.

No corpo da função, é feita a validação do token de acesso existente. Se ele realmente existir e ainda for válido, então o cabeçalho `Authorization` é preenchido com essa informação para que a requisição possa então ser aceita pelo provedor de serviços de vendas.

Caso não exista um token de acesso válido, a requisição continuará do mesmo jeito, mas será negada pelo provedor de serviços de vendas, fazendo com que a função `retry` seja chamada pelo Alamofire. Veja sua implementação completa no trecho a seguir:

```
func retry(_ request: Request, for session: Session, dueTo error: Error,
          completion: @escaping (RetryResult) -> Void) {
    guard let response = request.task?.response as? HTTPURLResponse,
          response.statusCode == 401 else {
        return completion(.doNotRetryWithError(error))
    }

    getAccessToken(completion: completion)
}
```

Perceba que essa função também recebe um closure para ser invocado caso o Alamofire tenha que tentar refazer a requisição ou desistir. Esse closure será passado para a função `getAccessToken`, detalhada anteriormente. Caso o token de acesso seja adquirido, esse mesmo closure será chamado com o parâmetro indicando que a requisição pode ser feita novamente, como pode ser visto no trecho abaixo, dentro do case `.success` da função `getAccessToken`:

```
return completion(.retry)
```

Nesse caso, a função `adapt` será chamada novamente, mas agora com um token válido para ser inserido no cabeçalho `Authorization`.

Veja que o início da função `retry` verifica se o código da resposta da requisição a ser tentada novamente. Se for `HTTP 401 Unauthorized`, significa que a requisição não foi autorizada, nesse caso o processo de retentativa continua. Caso contrário, o closure `completion` é chamado informando o Alamofire para não tentar novamente, como pode ser visto no trecho dentro do código `guard`:

```
return completion(.doNotRetryWithError(error))
```

E isso finaliza a implementação do interceptador de requisições, que será utilizado para configurar a instância do Alamofire a ser utilizado em todas as requisições, como será detalhado na seção a seguir.

6.4.4 - Criando o cliente do provedor de serviços de vendas

Agora que o interceptador de requisições já foi criado, para lidar com a questão da autenticação das requisições do cliente REST, é possível começar a construir a classe responsável por efetivamente fazer as requisições ao provedor de serviços de vendas, que será utilizado pelas outras partes da aplicação para acessarem seus serviços. Para isso, comece criando um arquivo chamado `SalesProviderClient`, dentro da pasta `WebServices` e nele a classe de mesmo nome, como no trecho a seguir:

```
import Foundation
import Alamofire

class SalesProviderClient {
    static let sharedInstance = SalesProviderClient()
    private let baseUrl = "https://sales-provider.appspot.com/"

    private var accessTokenStorage: AccessTokenStorage?
    private let session: Session

    private init() {

    }
}
```

Essa classe implementará um *singleton*, com a finalidade de haver apenas uma instância dela em toda a aplicação, por isso a necessidade de um construtor privado e um atributo estático com a sua instância.

Além da URL base do provedor de serviços de vendas, também existe um outro atributo para armazenar o token de acesso, que foi obtido na classe criada na seção anterior.

Por último existe o atributo `session`, que será utilizado para efetivamente realizar as requisições REST. Ele será configurado para utilizar o interceptador criado na seção anterior. Para isso, continue a criação da função `init`, como no trecho a seguir:

```
private init() {
    let configuration = URLSessionConfiguration.default
    configuration.timeoutIntervalForRequest = 10
    configuration.timeoutIntervalForResource = 10

    //TODO - fetch credentials from somewhere
    let userCredential = UserCredential(username: "matilde@siecola.com.br",
                                         password: "matilde")

    session = Session(configuration: configuration,
                       interceptor: AuthenticationInterceptor(
                           accessTokenStorage: accessTokenStorage,
                           userCredential: userCredential, baseUrl: baseUrl))
}
```

A primeira linha dessa função cria um objeto para a configuração de `session`, que basicamente define alguns *timeouts* nas requisições. Em seguida é criada uma instância de `UserCredential`, com dados fixos.



Em uma aplicação real as credenciais de usuário viriam de algum local seguro de armazenamento do sistema iOS, depois que o usuário os digitasse em um uma tela de login, por exemplo.



Utilize as credenciais de acesso do usuário criado na seção 6.1.4 desse capítulo.

Na última linha da função `init` a instância de `Session` é criada, com a configuração dos *timeouts* e também com o interceptador criado na sessão anterior. Isso significa que toda a requisição feita pelo Alamofire utilizando essa instância irá utilizar tal interceptador, fazendo com que todas as requisições tenha um token de acesso para serem autenticadas no provedor de serviços de vendas.

6.4.5 - Criando a operação para buscar a lista de produtos

Agora que a camada do cliente REST do projeto já possui toda a sua base montada, é hora de criar a primeira operação, que será para buscar todos os produtos cadastrados no provedor de serviços de vendas. Para isso, crie uma nova função na classe `SalesProviderClient`, como o trecho a seguir:

```

func getProducts(completion: @escaping ([Product]?) -> Void) {
    session.request(baseUrl + "api/products")
        .validate(statusCode: 200..<300)
        .validate(contentType: ["application/json"])
        .responseDecodable(of: [Product].self) { response in

            switch response.result {
            case .success:
                DispatchQueue.main.async {
                    return completion(response.value)
                }
            case let .failure(error):
                print(error)
                DispatchQueue.main.async {
                    completion(nil)
                }
            }
        }
}

```

Perceba na assinatura dessa nova função que ela não recebe nenhum parâmetro a ser utilizado na requisição em si, uma vez que a operação equivalente do provedor de serviços de vendas também não recebe. Mas, veja que a função também não retorna nenhum valor, isso porque sua execução é assíncrona. Na verdade, o resultado da operação, que é a lista de produtos, será passada no closure fornecido no parâmetro `completion`.

Aqui a instância `session` é utilizada, já configurada para usufruir do interceptador de requisições criado na seção anterior. Ela apenas é configurada, para essa requisição específica, a utilizar a URL equivalente à operação, que é `api/products`, conforme a [documentação²¹](#) do provedor de serviços de vendas.

A resposta esperada deve estar entre os códigos HTTP 200 e 299, estar no formato `application/json` e ser uma lista que possa ser deserializada em objetos do tipo `Product`. Se isso tudo estiver correto, a lista então é devolvida na invocação do closure `completion`, extraída do atributo `response.value`. Caso contrário, um *optional* nulo será retornado indicando que algo deu errado.

As chamadas ao closure `completion` estão envolvidas na instrução `DispatchQueue.main.async` pois elas irão atualizar a interface gráfica do usuário.

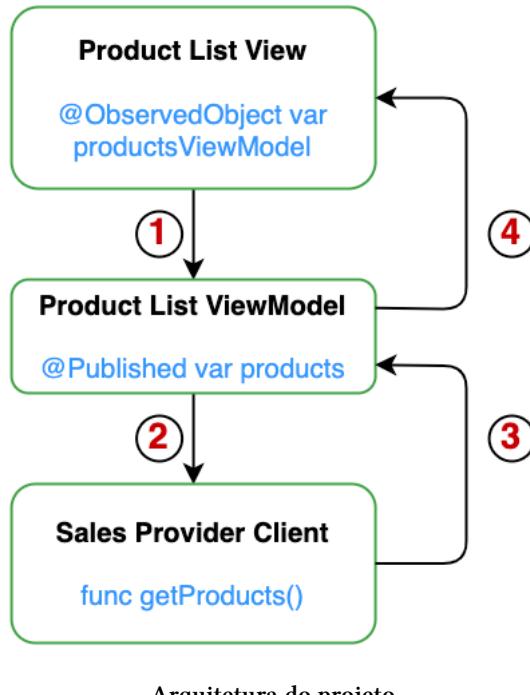
A utilização dessa função para consultar a lista de produtos será vista na seção a seguir.

Outras funções serão adicionadas a essa camada do cliente REST, como será visto no capítulo seguinte.

²¹<https://github.com/siecola/GAESalesProvider>

6.5 - Exibindo a lista de produtos

Agora que a camada de serviço já foi construída, com a sua primeira operação para buscar todos os produtos do provedor de serviços de vendas, é hora de construir as demais camadas da arquitetura que foi proposta no início desse capítulo para esse projeto, como mostra a figura a seguir:



Arquitetura do projeto

Como a função `getProducts` responde com a lista de produtos de forma assíncrona, como já foi explicado na sessão anterior, é necessário construir uma arquitetura que se comporte adequadamente a esse tipo de estratégia, de tal forma que o usuário tenha a informação solicitada na *view*, sem a necessidade de bloqueá-lo.

A camada **Product List View**, é a responsável por efetivamente exibir a interface com a lista de produtos ao usuário. Quando ela é instanciada, seu primeiro passo (1) é criar uma instância de **Product List ViewModel**, observando por alterações que possam acontecer para atualizar a lista de produtos ao usuário.

A camada **Product List ViewModel**, por sua vez possui um atributo de nome `products`, que é a lista e produtos em si a ser exibida. Quando essa lista é alterada, todos os que a observam são notificados. Quando essa camada é instanciada, ela chama a operação `getProducts` de **Sales Provider Client** no passo (2). Nesse momento um closure é fornecido para ser invocado quando a lista de produtos for obtida.

Quando a lista de produtos é adquirida, a camada **Sales Provider Client** chama o closure recebido como parâmetro para informar a lista de produtos no passo (3). Nesse momento a variável `products` de **Product List ViewModel** é alterada, fazendo com que todos os que o observam sejam notificados,

como mostra o passo (4).

Quando `productsViewModel` de `Product List View` é notificado de sua alteração, o SwiftUI então se encarrega de redesenhar para o usuário, já com a lista de produtos a ser exibida.

6.5.1 - Criando o view model da lista de produtos

Antes de começar a criação do *view model* para a lista de produtos, é preciso criar um modelo para ser aproveitado aqui e quando os detalhes do produto também forem exibidos. Para isso, crie um nova pasta/grupo no projeto com o nome de `ViewModels`. Dentro dele, crie um novo arquivo de nome `ProductModel`, conforme o trecho a seguir:

```
import Foundation

class ProductModel {
    var product: Product

    init(product: Product) {
        self.product = product
    }

    var id: UInt {
        return self.product.id
    }

    var name: String {
        return self.product.name
    }

    var description: String {
        return self.product.description
    }

    var code: String {
        return self.product.code
    }

    var price: Double {
        return self.product.price
    }
}
```

Com `ProductModel` será possível exibir as informações do produto, seja em uma lista ou em uma tela com seus detalhes. Sua utilização ficará mais clara na construção do *view model* da lista de produtos.

Para isso, crie um novo arquivo (nessa mesma pasta) com o nome de `ProductListViewModel`, como o trecho a seguir:

```
import Foundation

class ProductListViewModel: ObservableObject {
    @Published var products = [ProductModel]()

}
```

Perceba que `ProductListViewModel` implementa o protocolo `ObservableObject`, com que fará que quem o utilizar seja notificado sobre alguma mudança, que nesse caso será feita pelo seu atributo `products`, que é uma lista de `ProductModel` anotada com `@Published`. Com essa anotação, qualquer mudança que aconteça nesse atributo fará com que os observadores de `ProductListViewModel` sejam notificados.

Para fazer com que a operação para buscar os produtos seja invocada é necessário criá-la para utilizar a classe `SalesProviderClient` e além disso chamá-la na inicialização de `ProductListViewModel`, como mostra o trecho a seguir:

```
init() {
    fetchProducts()
}

func fetchProducts() {
    SalesProviderClient.sharedInstance.getProducts { products in
        if let products = products {
            self.products = products.map(ProductModel.init)
        }
    }
}
```

Veja que dentro da função `fetchProducts`, a instância de `SalesProvider` está sendo utilizada para invocar a função `getProducts`, que nesse caso está implementando um *trailing closure* para receber a resposta dessa operação assíncrona. Se os produtos realmente vierem na resposta, o atributo `products` com a lista de `ProductModel` é mapeada através de seu inicializador.

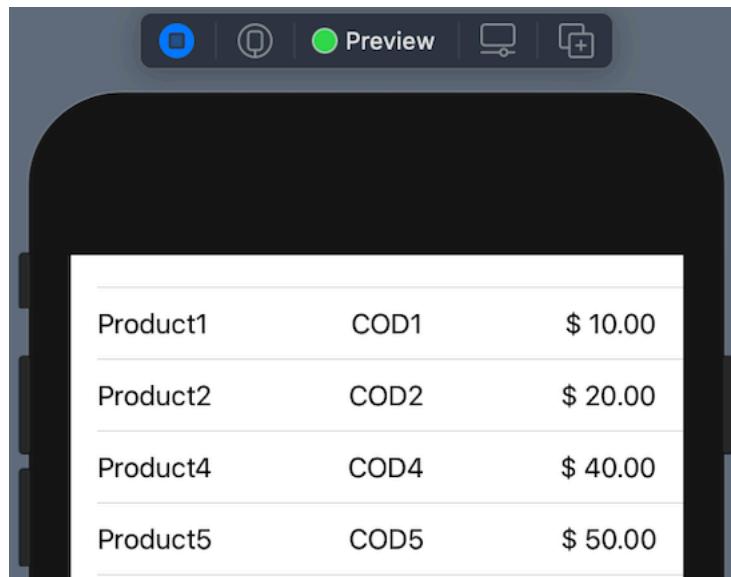
Nesse momento os observadores de `ProductListViewModel` serão notificados, como explicado anteriormente, o que ficará evidente na próxima seção.

6.5.2 - Criando a view da lista de produtos

Agora que as demais camadas do projeto já foram construídas, é possível criar a interface gráfica para exibir a lista de produtos, que por enquanto não será navegável ao usuário. Para isso, crie uma nova

pasta/grupo no projeto do Xcode de nome `Views`. Comece arrasando o arquivo `ContentView.swift` para dentro dele, assim todas as `views` ficarão organizadas nessa pasta.

A ideia é exibir apenas a lista e produtos requisitada no provedor de serviços de vendas, mostrando algumas informações como seu nome, código e preço, como ilustra a figura a seguir:



Lista de produtos

Para implementar essa tela, refatore o nome da struct `ContentView`, alterando-a para `ProductListView`. Agora acrescente o atributo que irá observar as alterações da lista de produtos de `ProductListViewModel`, como no trecho a seguir:

```
@ObservedObject private var productListViewModel = ProductListViewModel()
```

E para implementar a tela como mostrada na figura anterior, basta criar um componente do tipo `List` que itera na lista `products` de `ProductListViewModel`, exibindo o nome, código e preço do produto em caixas de texto, como no trecho a seguir, que substitui a implementação antiga do atributo `body` da antiga classe `ContentView`:

```
var body: some View {
    List {
        ForEach(self.productListViewModel.products, id: \.id) { product in
            HStack {
                Text(product.name)
                Spacer()
                Text(product.code)
                Spacer()
                Text(String(format: "$ %.2f", product.price))
            }
        }
    }
}
```

```
    }  
}  
}
```

Veja que a construção de `body` ainda continua tendo somente uma `view` como raiz, que é o componente `List`. Dentro dele é utilizado o `ForEach` que itera na lista de produtos de `ProductListViewModel`, tendo como identificador o atributo `id` do produto.

A montagem da célula para exibir cada produto é feita com o componente `HStack`, com caixas de texto para cada campo e um espaçador entre eles, para que eles ocupem adequadamente todo o espaço disponível na tela.

Para testar toda a implementação até agora, execute a aplicação no Xcode Preview ou em um dispositivo simulado. A lista de produtos deverá aparecer, como na figura anterior.

6.6 - Conclusão

Até o momento, nesse capítulo, foi construída a base para o projeto cliente do provedor de serviços de vendas, para consumir as operações do serviço de produtos. Nesse processo já foi visto como criar um cliente REST para responder a consultas assíncronas, além de utilizar uma arquitetura que permite que um `view model` notifique uma `view` para que a interface do usuário seja redesenhada, caso alguma informação se altere, como foi o caso da lista de produtos aqui construída.

No próximo capítulo, esse projeto continuará a ser desenvolvido, adicionando navegação para a exibição dos detalhes do produto, bem como outras operações que poderão ser feitas no provedor de serviços de vendas.

7 - Criando navegação com o NavigationView e consumindo outras operações do serviço REST

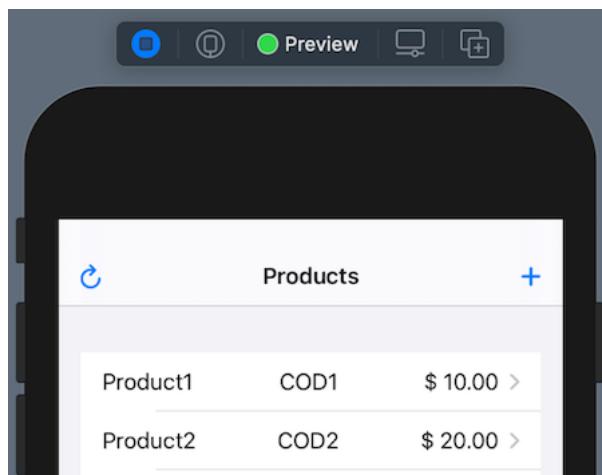
No capítulo anterior foi construída a lista que exibe os produtos, trazidos do provedor de serviços de vendas. Agora é possível consumir outras operações do provedor, assim como construir a navegação dentro do aplicativo para que o usuário possa realizar outras tarefas como criar, editar e excluir produtos.



Para as operações de criação, alteração e exclusão, que serão implementadas nesse capítulo, será necessário um usuário com papel ADMIN cadastrado no provedor de serviços de vendas.

7.1 - Adicionando navegação ao aplicativo

A ideia dessa sessão é adicionar um mecanismo de navegação no aplicativo, para que, nas próximas seções, o usuário possa abrir a tela de detalhes do produto, quando clicar em um item da lista. Além disso também será possível adicionar botões para carregar todos os produtos novamente e para adicionar um novo, como pode ser visto na figura a seguir:



Navegação do aplicativo

Veja que também será adicionado um título principal a essa tela, com o nome de `Products`.

Para começar essa implementação, abra o arquivo `ProductListView.swift` e envolva o componente `List` com o componente `NavigationView`, como mostra o trecho a seguir:

```

var body: some View {
    NavigationView {
        List {
            ForEach(self.productListViewModel.products, id: \.id) { product in
                //TODO 1 - add the navigation link
                HStack {
                    Text(product.name)
                    Spacer()
                    Text(product.code)
                    Spacer()
                    Text(String(format: "$ %.2f", product.price))
                }
            }
            //TODO 3 - add the delete action function
        }
        //TODO 2 - navigation view configuration
    }
}

```

As configurações do componente `NavigationView`, como os novos botões a serem adicionados, deverão ser colocados onde há o comentário TODO - 2.

Também deverá ser adicionado um componente chamado `NavLink`, envolvendo o `HStack` dentro do `ForEach`, no lugar onde há o comentário TODO - 1. Esse componente será responsável instruir o SwiftUI para onde a navegação deverá acontecer quando o usuário clicar em um produto da lista, que nesse caso levará para a tela de detalhes do produto.

Então, para começar, adicione os componentes `NavigationView` e `NavLink`, nos locais indicados, como mostra o trecho a seguir:

```

var body: some View {
    NavigationView {
        List {
            ForEach(self.productListViewModel.products, id: \.id) { product in
                NavLink(
                    destination: Text("Destination")) {
                    HStack {
                        Text(product.name)
                        Spacer()
                        Text(product.code)
                        Spacer()
                        Text(String(format: "$ %.2f", product.price))
                    }
                }
            }
        }
    }
}

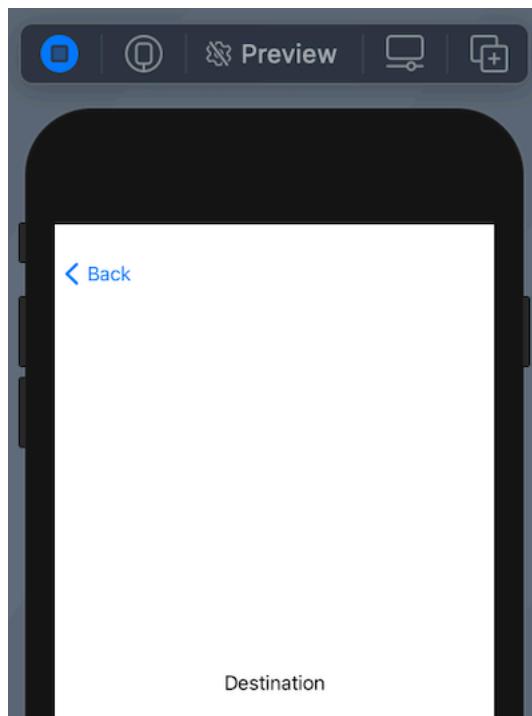
```

```
        }
        //TODO 3 - add the delete action function
    }
    //TODO 2 - navigation view configuration
}
}
```

Por enquanto o componente `NavigationView` ainda não tem nenhuma configuração, que será feita logo a seguir.

Veja que o componente `NavLink` possui um atributo chamado `destination`. Nele é que será colocada a `view`, que ainda será criada, para exibir os detalhes do produto. No momento, somente uma caixa de texto foi criada.

Executando o aplicativo do jeito que ele está, já é possível clicar em qualquer produto e ver que será exibida uma tela, já com o botão de `Back`, que exibe a caixa de texto, como pode ser visto na figura a seguir:



Iniciando a navegação do aplicativo

Essa tela será construída na próxima seção.

Antes de adicionar os botões de navegação na tela principal, crie um novo atributo para a `struct ProductListView`, como o trecho a seguir:

```
@State private var showModal = false
```

Esse atributo de estado será utilizado para controlar o comportamento da tela de criação de um novo produto, como será visto em algumas seções mais adiante.

Os dois botões que serão criados na barra de navegação do app deverão invocar duas novas funções: recarregar os produtos e abrir a tela para a criação de um novo. Elas devem ser criadas dentro da *struct* ProductListView, como mostra o trecho a seguir:

```
private func reloadProducts() {
    productListViewModel.fetchProducts()
}

private func showNewProductView() {
    self.showModal = true
}
```

A função `reloadProducts` invoca a função `fetchProducts` de `ProductListViewModel`, que se encarrega de buscar os produtos novamente e atualizar a tela, através do atributo de mesmo nome anotado com `@ObservedObject`.

Já a função `showNewProductView` apenas altera o valor do atributo de estado `showModal` para que a tela de criação de um novo produto apareça.

Agora que tudo já está preparado, crie as configurações do `NavigationView` para exibir seu título e adicionar os botões na barra superior, como no trecho a seguir, que deve ser colocado onde está o comentário `TODO - 2:`:

```
.navigationBarTitle("Products", displayMode: .inline)
.navigationBarItems(leading: Button(action: reloadProducts) {
    Image(systemName: "arrow.clockwise")
        .foregroundColor(Color.blue)
}, trailing: Button(action: showNewProductView) {
    Image(systemName: "plus")
        .foregroundColor(Color.blue)
})
.sheet(isPresented: $showModal) {
    VStack {
        Text("Destination")
        Button("Close") {
            self.showModal = false
        }
    }
}
```

Esse trecho de código está dividido em 3 porções:

- **navigationBarTitle**: configura o título da barra de navegação;
- **navigationBarItems**: adiciona os dois botões da barra de navegação;
- **sheet**: responsável por exibir um modal, controlado pela variável de estado `showModal1`.

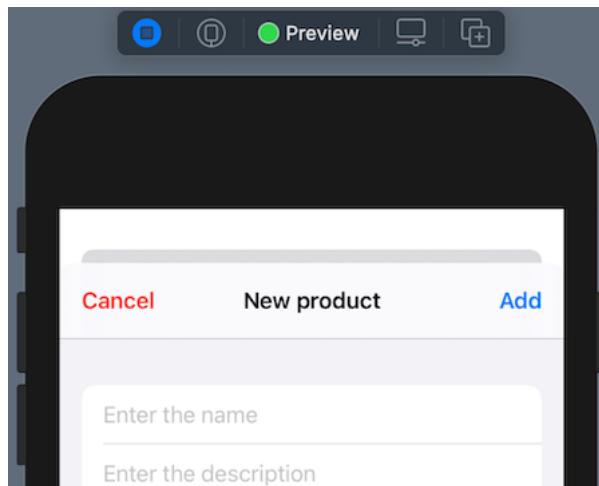
Os dois botões adicionados possuem as ações em si que serão invocadas quando o usuário clicar sobre eles, que nesse caso são as funções `reloadProducts` e `showNewProductView`, além dos ícones a serem exibidos na interface gráfica.

O *closure* dentro da função `sheet` configura a *view* a ser exibida quando a variável de estado `showModal1` tiver o valor `true`, algo que será alterado quando o botão para adicionar um novo produto for clicado. Esse trecho de código ainda será alterado em algumas seções adiante. Por enquanto ele apenas exibe uma caixa de texto com um botão para fechar o modal, exemplificando o mecanismo em si.

Executando a aplicação é possível verificar o título na barra de navegação, assim como os dois botões, que recarrega os produtos do provedor de serviços de vendas e abre o modal.

7.2 - Criando um novo produto

Agora que o projeto já foi preparado para ter um mecanismo de navegação, é hora de implementar mais uma operação nele, que é a criação de um novo produto, que será feita através de um modal com os dados do produto a serem cadastrados pelo usuário, como mostra a figura a seguir:



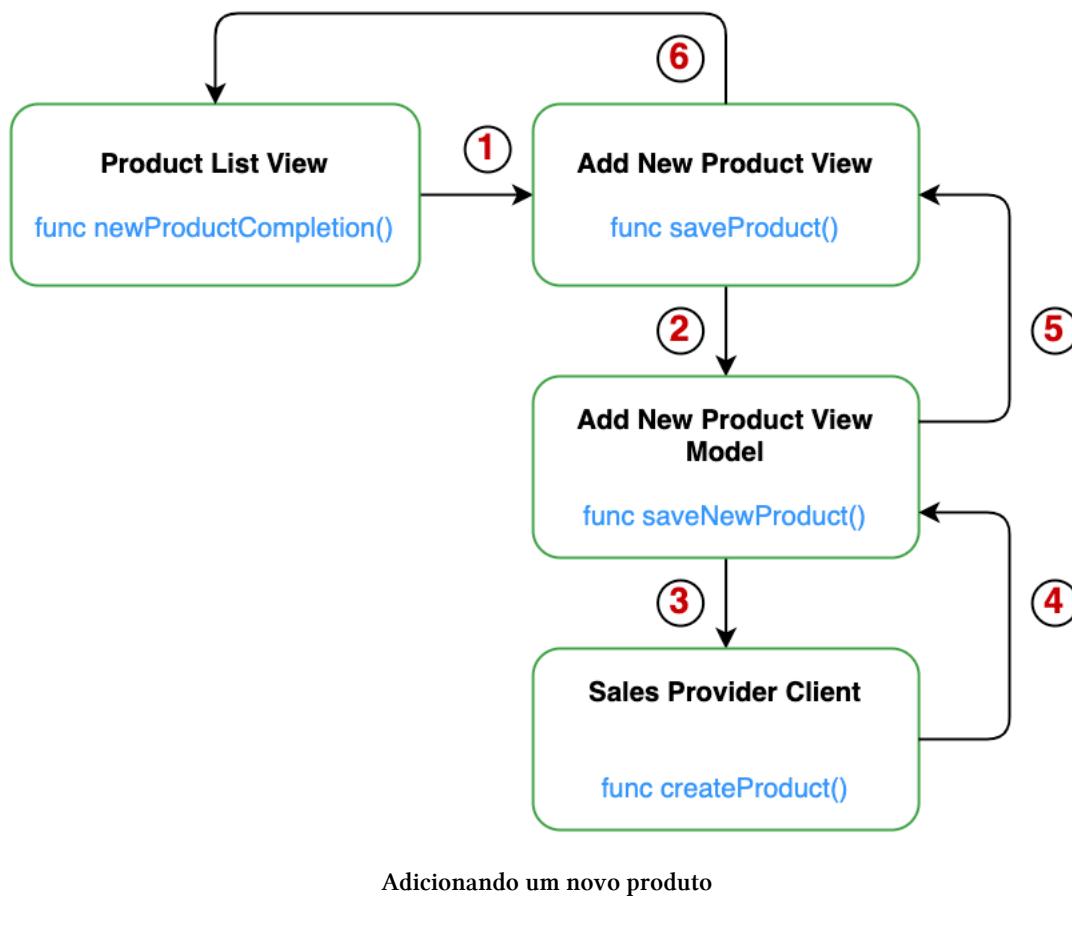
View para criação de um novo produto

Caso o usuário pressione o botão `Cancel`, a criação do produto é cancelada e ele é redirecionado de volta para a tela com a lista. Ao pressionar o botão `Add`, a operação de criação de um novo produto será invocada no provedor de serviços de vendas, com os dados informados pelo usuário. Aqui será

demonstrada uma forma simples para tratamento de erros que podem acontecer, como o cadastro de mais de um produto com o mesmo código.

A lista de produtos da tela principal será atualizada, caso o produto seja cadastrado com sucesso, através de um *callback* que irá informar o novo produto, sem a necessidade do usuário ter que atualizar a tela novamente.

A seguir um diagrama mostrando como essa nova tela irá funcionar:



Quando o usuário clicar no botão para adicionar um novo produto, na tela principal, uma nova *view*, que será criada com o nome de `AddNewProductView` aparecerá como um modal (1). Depois que o usuário preencher todos os dados do produto e clicar em Add, a função `saveNewProduct` do novo *view Model* de nome `AddNewProductViewModel` será invocada (2), para preparar o produto para ser enviado ao cliente `SalesProviderClient` em (3), que também recebe um closure para ser chamado quando a resposta do provedor de serviços de vendas chegar, em (4). Depois de fazer sua validação, o *viewModel* então avisa a *view* que o produto foi salvo, em (5), que por sua vez fecha o modal, atualizando a lista de produtos em (6).

As seções a seguir irão criar e detalhar todos esse novos componentes, bem como seus passos.

7.2.1 - Criando a operação para criar um novo produto

A camada do cliente REST do aplicativo precisa de uma nova função para criar um produto novo no provedor de serviços de vendas. Para isso, abra o arquivo `SalesProviderClient.swift` e crie-a, como mostra o trecho a seguir:

```
func createProduct(product: Product,
                    completion: @escaping (Product?) -> Void) {
    session.request(baseUrl + "api/products", method: .post,
                    parameters: product, encoder: JSONParameterEncoder.default)
    .validate(statusCode: 200..<300)
    .validate(contentType: ["application/json"])
    .responseDecodable(of: Product.self) { response in

        switch response.result {
        case .success:
            DispatchQueue.main.async {
                return completion(response.value)
            }
        case let .failure(error):
            print(error)
            DispatchQueue.main.async {
                completion(nil)
            }
        }
    }
}
```

Veja que o princípio e a utilização da sessão configurada do Alamofire são muito semelhantes ao que foi criado no capítulo anterior para buscar todos os produtos, mas aqui a operação a ser feita é um HTTP POST, contendo o produto a ser criado no corpo da requisição, que é passada no parâmetro de nome `parameters` da chamada `session.request`. O parâmetro `encoder` diz que o produto, a ser inserido no corpo da requisição, deve estar no formato JSON.

A resposta esperada do provedor de serviços de vendas deve conter o código de resposta HTTP `Created 201` com o produto criado em formato JSON em seu corpo, o que é verificado nas funções `validate` e `responseDecodable`. Se tudo tiver certo, o closure de nome `completion` é chamado, passando o produto criado.

7.2.2 - Criando o view model para adicionar um novo produto

Seguindo a estratégia apresentada na figura anterior, crie um novo arquivo chamado `AddNewProductViewModel` na pasta `ViewModels` para representar a classe de mesmo nome que será utilizada como o `viewModel`

do modal para a criação de um novo produto. Ele deverá conter os atributos do produto que serão digitados pelo usuário, mais uma função para invocar o cliente REST. Veja como ele deverá ficar, no trecho a seguir:

```
import Foundation

class AddNewProductViewModel {
    var name: String = ""
    var description: String = ""
    var code: String = ""
    var price: Double = 0.0

    func saveNewProduct(completion: @escaping (Product?) -> Void) {
        let product = Product(id: 0, name: self.name, description:
            self.description, code: self.code,
            price: self.price)

        SalesProviderClient.sharedInstance
            .createProduct(product: product) { product in
                completion(product)
            }
    }
}
```

Perceba que a função `saveNewProduct` recebe um closure para ser invocado quando a resposta do cliente REST for gerada. O produto a ser criado é formado pelos atributos de `AddNewProductViewModel`, que serão associados aos componentes da interface do usuário, como será visto na sessão seguinte.

7.2.3 - Criando a view para adicionar um novo produto

A estrutura da `view` para criar o novo produto será semelhante à que foi feita no capítulo 5, com um componente do tipo `Form` e seus campos. Para começar, crie um novo arquivo com o *template* SwiftUI View, com o nome de `AddNewProductView` na pasta `Views`. Veja como deve ficar sua estrutura principal, já com os atributos que serão necessários para seu funcionamento:

```
import SwiftUI

struct AddNewProductView: View {
    @Binding var isPresented: Bool
    @State private var addNewProductViewModel = AddNewProductViewModel()
    @State private var showAlert = false
    var completion: (Product) -> Void

    var body: some View {
        ...
    }
}

struct AddEditProductView_Previews: PreviewProvider {
    static var previews: some View {
        AddNewProductView(isPresented: .constant(false)) { product in
            ...
        }
    }
}
```

Os atributos que foram criados nessa struct são os seguintes:

- **isPresented**: essa é uma variável de estado que será passada por `ProductListView`, para dizer se o modal da criação do produto deve ou não aparecer;
- **addNewProductViewModel**: esse é o `viewModel` criado para essa tela;
- **showAlert**: essa variável de estado irá controlar se o alerta de falha de criação do produto deverá ou não aparecer;
- **completion**: esse closure será invocado assim que um produto for criado com sucesso. Ele deverá ser passado como parâmetro por `ProductListView` no momento da criação de `AddEditProductView`.

Perceba que a struct `AddEditProductView_Previews`, que foi criada pelo *template* do SwiftUI, também teve que ser alterada, para que o Xcode Preview funcione adequadamente para essa tela. Os dois atributos passados como parâmetros na criação de `AddEditProductView` nesse caso são apenas para ficar em conformidade com os atributos que ela deve receber.

Será necessário criar duas funções dentro de `AddNewProductView`. Uma para cada botão disponível ao usuário, ou seja, adicionar o produto ou cancelar a ação. Por isso, comece com a função a seguir:

```
private func dismiss() {
    self.isPresented = false
}
```

Ela será chamada quando o usuário clicar no botão Cancel. Veja que ela apenas altera o valor da variável de estado `isPresented` para falso. Como ela está atrelada a `ProductListView`, o modal de criação de produto será fechado, como será visto na seção seguinte.

A outra função deverá ser criada para tratar o evento de clique no botão Add, como pode ser visto no trecho a seguir:

```
private func saveProduct() {
    self.showAlert = false
    self.addNewProductViewModel.saveNewProduct { product in
        if let product = product {
            completion(product)
            self.isPresented = false
        } else {
            self.showAlert = true
        }
    }
}
```

Antes de mais nada a função deve ter certeza de que o alerta, que ainda será criado no body de `AddNewProductView` não apareça. Por isso o atributo `showAlert` é mantido com falso. Em seguida, a função `saveNewProduct` é invocada para a criação do novo produto. Repare que as informações para construção do novo produto estarão associadas aos componentes gráficos que ainda serão criados no body da `view`.

Caso o produto seja criado com sucesso, o closure é invocado, tendo o produto passado como parâmetro e a variável de estado `isPresented` alterada para `false`, para que o modal seja fechado. O tratamento da invocação do closure `completion` será visto na seção seguinte.

Para implementar a interface gráfica da tela de criação do novo produto, será necessário ter uma caixa de texto especial para o preço, de forma semelhante ao que foi feito no capítulo 5. Como isso também será necessário na tela de edição, crie um novo arquivo no *template* SwiftUI View com o nome de `PriceField` para construir tal componente, como no trecho a seguir:

```

import SwiftUI
import Combine

struct PriceField : View {
    @State private var enteredValue: String = ""
    @Binding var value: Double

    var body: some View {
        TextField("Enter the price", text: $enteredValue)
            .onReceive(Just(enteredValue)) { typedValue in
                if let newValue = Double(typedValue) {
                    self.value = newValue
                }
            }
            .onAppear(perform:{self.enteredValue = String(format: "%.02f", self.value)})
            .keyboardType(.decimalPad)
    }
}

struct PriceField_Previews: PreviewProvider {
    static var previews: some View {
        PriceField(value: .constant(0))
    }
}

```

O código é muito semelhante ao que foi visto no capítulo 5, com exceção que aqui não está sendo feita nenhuma validação do preço em si, para deixar o código um pouco mais simples.

De volta à struct AddNewProductView, prossiga com a implementação da interface gráfica para a criação do produto, em seu no atributo body. Ela deverá conter um NavigationView, onde os botões Cancel e Add serão posicionados. Também deverá ter um Form com uma Section onde os componentes do produto serão criados. Além disso, haverá um trecho para a exibição de um alerta ao usuário, caso a criação do produto dê errado. Veja como deve ficar o código no trecho a seguir:

```

var body: some View {
    NavigationView {
        Form {
            Section() {
                TextField("Enter the name",
                          text:$self.$addNewProductViewModel.name)

                TextField("Enter the description",
                          text:$self.$addNewProductViewModel.description)
            }
        }
    }
}

```

```

        TextField("Enter the code",
            text:self.$addNewProductViewModel.code)

        PriceField(value: self.$addNewProductViewModel.price)
    }
}

.navigationBarTitle("New product", displayMode: .inline)
.navigationBarItems(leading: Button(action: dismiss) {
    Text("Cancel").foregroundColor(.red)
}, trailing: Button(action: saveProduct) {
    Text("Add").foregroundColor(.blue)
})
.alert(isPresented: $showAlert) {
    Alert(title: Text("Error"),
        message: Text("The product couldn't be saved"), dismissButton: .default(
        Text("Dismiss")))
}
}

```

Perceba que os componentes de `Section` estão associados aos atributos do `viewModel AddNewProductViewModel`. Isso significa que quando o usuário preencher os dados do produto, seus atributos receberão os valores, para a criação da instância de `Product` a ser enviada para o cliente REST.

A instrução `navigationBarItems` de `NavigationView` posiciona os dois botões no topo da tela, que invocam as funções criadas na seção anterior: cancelar e criar o produto.

Perceba a seção `alert`, controlada pela variável de estado `showAlert`. Se o valor dela alterar para `true`, dentro da função `saveProduct`, um alerta será mostrado o usuário.

7.2.4 - Alterando a view principal para adicionar um novo produto

De volta ao arquivo `ProductListView`, crie a função que será passada como parâmetro na criação de `AddNewProductView`, como no trecho a seguir:

```

private func newProductCompletion(product: Product) {
    self.productListViewModel.products
        .append(ProductModel(product: product))
}

```

Ela será o closure que será invocado assim que o produto for criado com sucesso dentro de `AddNewProductView`.

Por fim, para vá até a sheet de ProductListView e altere todo o seu conteúdo para exibir a tela de criação de um novo produto, como no trecho a seguir:

```
.sheet(isPresented: $showModal) {  
    AddNewProductView(isPresented: self.$showModal,  
                      completion: newProductCompletion)  
}
```

Veja que a variável de estado showModal é utilizada para dizer se o modal de criação de novo produto deve ou não aparecer.

Perceba também que a função newProductCompletion, que foi criada há pouco, é passada para ser invocada quando o produto for efetivamente criado.

Para testar toda a implementação, execute aplicação em um simulador de iOS ou no Xcode Preview, partindo de ProductListView, para que a lista de produtos seja exibida.

Para criar um novo produto, clique no ícone + da tela principal. O modal para sua criação deverá aparecer. Tente criar um novo produto, com um código que ainda não exista no provedor de serviços de vendas, pois ele não deixará mais de um produto com o mesmo código. Aliás, esse é uma boa estratégia para testar a falha na criação de um produto, para visualizar o alerta ao usuário.

7.3 - Excluindo um produto

A próxima operação a ser implementada será para apagar um produto. A estratégia será semelhante à utilizada na seção anterior:

- Criar a operação correspondente na camada do cliente REST;
- Criar a função no *viewModel* para ser invocada pela *view*;
- Disponibilizar a operação na interface do usuários.

Na interface gráfica, será disponibilizado o gesto de clicar sobre um item da lista e arrastar para a esquerda, fazendo que a opção Delete apareça ao usuário.

7.3.1 - Criando a operação para apagar um produto pelo seu código

Para começar, abra o arquivo SalesProviderClient.swift e crie a nova função para acessar a operação para apagar um produto no provedor de serviços de vendas, seguindo sua [documentação](#)²², que deve receber o código do produto a ser apagado:

²²<https://github.com/siecola/GAESalesProvider>

```
func deleteProduct(productCode: String,
                   completion: @escaping (Product?) -> Void) {
    session.request(baseUrl + "api/products/\\" + productCode
        .addingPercentEncoding(withAllowedCharacters: .urlHostAllowed)!) ,
        method: .delete)
    .validate(statusCode: 200..<300)
    .validate(contentType: ["application/json"])
    .responseDecodable(of: Product.self) { response in

        switch response.result {
        case .success:
            DispatchQueue.main.async {
                return completion(response.value)
            }
        case let .failure(error):
            print(error)
            DispatchQueue.main.async {
                completion(nil)
            }
        }
    }
}
```

Veja que a função recebe, além do código do produto a ser apagado, o closure para ser invocado quando a operação for concluída. O código do produto é então passado na URL, na chamada da função `request` de `session`, assim como a operação HTTP `DELETE` é informada no parâmetro `method`. As validações também são informadas nas chamadas a `validate`, assim como a resposta esperada no corpo da mensagem é o produto apagado em si, como pode ser visto em `responseDecodable`. O closure `completion` é chamado com o produto apagado, que foi deserializado pelo `Alamofire`.

7.3.2 - Alterando o view model para excluir o produto

Como a ideia é notificar o usuário, caso aconteça algum problema e o produto não seja apagado do provedor de serviços de vendas, será necessário fazer com que a função a ser criada em `ProductListViewModel` receba um closure para ser invocado sobre o resultado dessa operação, além obviamente do código do produto a ser excluído. Para isso, abra o arquivo `ProductListViewModel.swift` e crie a nova função, como mostra o trecho a seguir:

```
func deleteProduct(product: ProductModel,
                    completion: @escaping (Product?) -> Void) {
    SalesProviderClient.sharedInstance
        .deleteProduct(productCode: product.code) { product in
            completion(product)
        }
}
```

Veja que a operação criada em `SalesProviderClient` está sendo invocada, tendo o retorno repassado para o closure `completion`.

7.3.3 - Alterando a view com a opção para excluir o produto

Tendo a função para apagar o produto pronta, é hora de adaptar a *view* da lista de produtos para oferecer essa opção ao usuário. Como dito anteriormente, essa opção estará disponível no gesto de clicar em um produto da lista e arrastar para a esquerda. Isso pode ser feito simplesmente adicionando o *callback* `onDelete` no componente `List`, onde está o comentário `TODO 3`, como no trecho a seguir:

```
.onDelete(perform: self.deleteProduct)
```

Obviamente a função `deleteProduct` ainda não existe, mas antes de escrevê-la, crie um atributo de estado para controlar a exibição do alerta ao usuário, caso aconteça algum erro nessa operação. Isso pode ser feito adicionando o trecho a seguir, logo após a declaração de `showModal`:

```
@State private var showAlert = false
```

Agora essa variável pode ser utilizada para balizar o comportamento do alerta, que deve ser declarado logo após o fechamento da chave do componente `NavigationView`, como no trecho a seguir:

```
.alert(isPresented: $showAlert) {
    Alert(title: Text("Error"),
          message: Text("The product couldn't be deleted"), dismissButton: .default(\n
Text("Dismiss")))
}
```

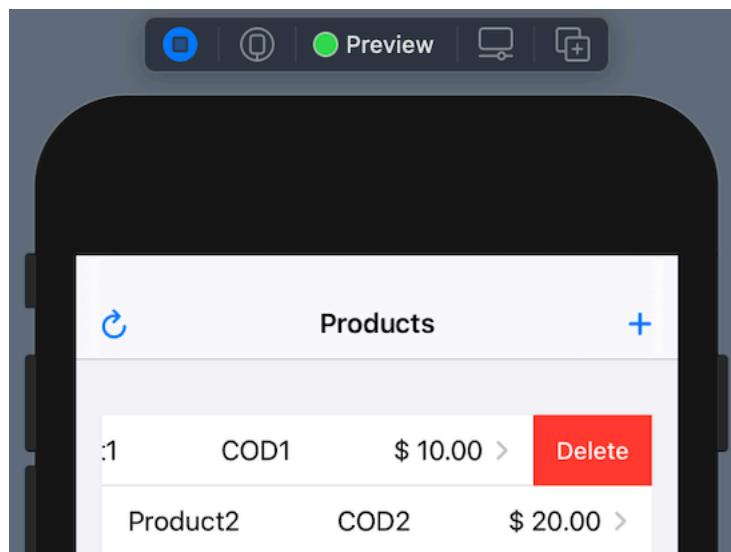
Veja que esse trecho é bem parecido com o que foi utilizado na *view* para criação de um novo produto. Por fim, a função `deleteProduct` pode ser criada dentro da *struct* `ProductListView`, como no trecho a seguir:

```
private func deleteProduct(indexSet: IndexSet) {
    showAlert = false
    let productToBeDeleted = self.productListViewModel
        .products[indexSet.first!]

    productListViewModel
        .deleteProduct(product: productToBeDeleted) { product in
            if let _ = product {
                self.productListViewModel.products.remove(at: indexSet.first!)
            } else {
                showAlert = true
            }
        }
}
```

A razão pela qual essa função recebe um parâmetro do tipo `IndexSet` se deve ao fato do `callback onDelete` poder passar vários itens a serem deletados ao mesmo tempo, que não é caso aqui, mas que ainda assim poderia ser utilizado em outras ocasiões. De posse desse índice do produto a ser apagado da lista, é necessário localizar o produto em si. Depois, é necessário chamar a função `deleteProduct` de `ProductListViewModel`. Se tudo deu certo e o produto foi efetivamente apagado do provedor de serviços de vendas, então ele também pode ser removido da lista de produtos de `ProductListViewModel`, fazendo com que a tela seja atualizada de tal operação. Caso contrário, um alerta é exibido ao usuário que a operação falhou.

Para testar essa implementação, execute a aplicação em um simulador ou no Xcode Preview, clicando sobre um item da lista de produtos e arrastando para a esquerda. A seguinte opção deverá aparecer:



Excluindo um produto

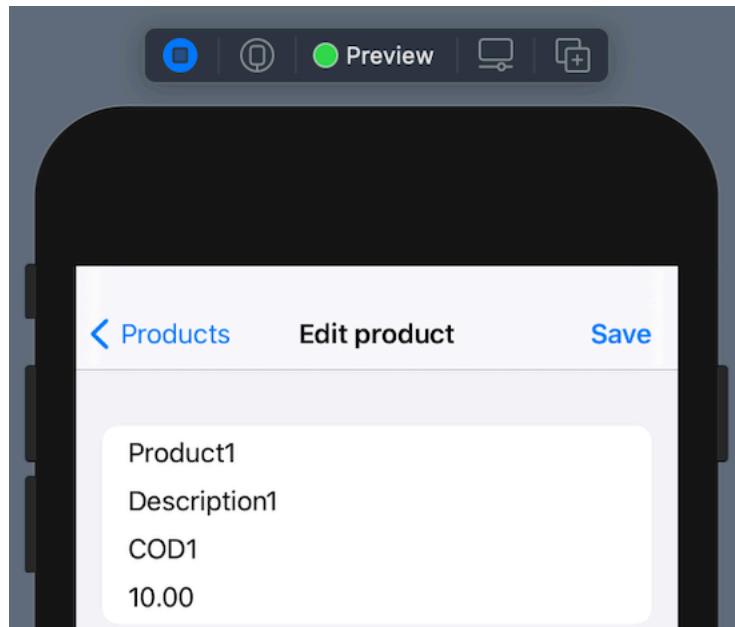
Ao clicar-se no botão `Delete`, o mecanismo desenvolvido nessa seção será invocado, dando início ao processo de exclusão do produto no provedor de serviços de vendas.

7.4 - Visualizando e alterando os detalhes de um produto

A última operação a ser implementada será para visualizar e editar os detalhes de um produto. A estratégia será semelhante à utilizada na seção anterior:

- Criar a operação correspondente na camada do cliente REST;
- Criar a função no `viewModel` para ser invocada pela `view`;
- Chamar a nova `view` da tela com a lista de produtos.

Veja como deverá ficar a tela para a edição do produto:



Editando um produto



Para deixar o código mais simples, nada de especial foi utilizado na aparência dessa tela.

Veja que o único botão a ser adicionado na barra superior é o `Save`, que deverá invocar a função para salvar o produto existente. Como essa tela será chamada da principal, com a lista de produtos, que já possui o `NavigationView`, esta também estará inserida nesse mecanismo, por isso a navegação já está “pronta”.

7.4.1 - Criando a operação para alterar um produto pelo seu código

Para criar a operação para alterar um produto existente, abra o arquivo `SalesProviderClient.swift` e acrescente uma nova função para esse trabalho, que deverá receber o código do produto a ser alterado, o produto em si e um closure para ser invocado quando a operação for concluída.

De acordo com a [documentação²³](#) do provedor de serviços de vendas, o verbo HTTP deverá ser o PUT. Veja como deve ficar o código seguindo essas diretrizes:

```
func updateProduct(productCode: String, product: Product,
                    completion: @escaping (Product?) -> Void) {
    session.request(baseUrl + "api/products/\\" + productCode
        .addingPercentEncoding(withAllowedCharacters: .urlHostAllowed)!) ,
        method: .put,
        parameters: product, encoder: JSONParameterEncoder.default)
    .validate(statusCode: 200..<300)
    .validate(contentType: ["application/json"])
    .responseDecodable(of: Product.self) { response in

        switch response.result {
        case .success:
            DispatchQueue.main.async {
                return completion(response.value)
            }
        case let .failure(error):
            print(error)
            DispatchQueue.main.async {
                completion(nil)
            }
        }
    }
}
```

Perceba que o formato dessa função é muito parecido com as outras que foram implementadas nesse capítulo.

7.4.2 - Criando o view model para alterar um produto

Para construir o `viewModel` que irá cuidar da edição do produto, crie um novo arquivo chamado `EditProductViewModel` na pasta `ViewModels`. Uma das diferenças desse novo `viewModel` é que ele

²³<https://github.com/siecola/GAESalesProvider>

deverá guardar o código original do produto, assim como seu id, para que possam ser fornecidos ao provedor de serviços de vendas, uma vez que o próprio código pode ser alterado pelo usuário. Além disso, os dados do produto deverão ser passados por uma função a ser chamada no inicializador da *view* que for utilizá-lo.

```
import Foundation

class EditProductViewModel {
    private var id: UInt = 0
    private var originalCode: String = ""
    var name: String = ""
    var description: String = ""
    var code: String = ""
    var price: Double = 0.0

    func setProduct(product: Product) {
        self.id = product.id
        self.name = product.name
        self.description = product.description
        self.originalCode = product.code
        self.code = product.code
        self.price = product.price
    }
}
```

Por fim, será necessário criar a função para chamar a camada do serviço REST para de fato alterar o produto:

```
func updateProduct(completion: @escaping (Product?) -> Void) {
    let product = Product(id: self.id, name: self.name, description:
        self.description, code: self.code,
        price: self.price)

    SalesProviderClient.sharedInstance
        .updateProduct(productCode: self.originalCode,
                       product: product) { product in
            completion(product)
        }
}
```

Veja que o produto a ser construído preserva seu id original, tendo os demais parâmetros obtidos do *viewModel*, que foram alterados pelo usuário. Repare também que a chamada ao provedor de

serviços de vendas utiliza o código original do produto, para que ele possa de fato ser encontrado para ser alterado.

7.4.3 - Criando a view para alterar um produto

Para começar a construir a *view* para a edição do produto, crie um novo arquivo com o *template* do SwiftUI View na pasta Views com o nome de EditProductView. Essa *struct* deverá receber o produto a ser editado, assim como um *closure* que será chamado quando a operação for concluída com sucesso. Como atributos privados, ela deverá ter uma variável de estado para controlar a exibição de uma mensagem de alerta ao usuário, além do *viewModel* construído na seção anterior.

Um atributo especial que será necessário aqui é a variável de ambiente *PresentationMode* para controlar a navegação, quando a operação de alteração do produto for concluída.

Também será necessário um inicializador para inserir o produto a ser alterado no *viewModel*. Veja como deverá ficar até o momento:

```
import SwiftUI

struct EditProductView: View {
    @Environment(\.presentationMode) var presentationMode: Binding<PresentationMode>

    var product: Product
    var completion: (Product) -> Void

    @State private var showAlert = false
    @State private var editProductViewModel = EditProductViewModel()

    init(product: Product, completion: @escaping (Product) -> Void) {
        self.product = product
        self.completion = completion
        self.editProductViewModel.setProduct(product: product)
    }

    var body: some View {
    }
}

struct EditProductView_Previews: PreviewProvider {
    static var previews: some View {
        EditProductView(product: Product(id: 0, name: "product",
                                         description: "desc", code: "cod", price: 0.0)) { product in
    }
}
```

```

    }
}
}
```

Da mesma forma como foi feito na tela de criação do produto, é necessário fazer com que o *preview* crie uma instância de `EditProductView`, obedecendo os parâmetros exigidos pelo seu construtor, como pode ser visto no fim do trecho anterior.

A função para efetivamente invocar a operação para alterar o produto deve ser criada dentro da `struct EditProductView`, de forma parecida com a que foi feita para a criação do produto, como pode ser visto no trecho a seguir:

```

private func updateProduct() {
    showAlert = false
    editProductViewModel.updateProduct { product in
        if let product = product {
            completion(product)
            self.presentationMode.wrappedValue.dismiss()
        } else {
            showAlert = true
        }
    }
}
```

Veja que a única diferença aqui é a instrução `self.presentationMode.wrappedValue.dismiss()` que comanda o `NavigationView` a voltar para a tela principal, quando o produto é alterado com sucesso no provedor de serviços de vendas.

O que falta agora é construir a interface gráfica, preenchendo a variável `body` dessa `struct`. Como dito anteriormente, para manter a simplicidade do código, não houve nenhum tratamento da interface gráfica para que ficasse mais bonita ao usuário.

O princípio é muito parecido com o que foi feito na tela de criação do produto, ou seja, um `Form` com `TextField` para cada campo. Além disso é necessário criar o botão para salvar o botão, que deverá chamar a função `updateProduct`. Veja como deve ficar no trecho a seguir:

```

var body: some View {
    Form {
        Section() {
            TextField("Enter the name",
                      text:self.$editProductViewModel.name)

            TextField("Enter the description",
                      text:self.$editProductViewModel.description)
    }
}
```

```

        TextField("Enter the code",
            text:$editProductViewModel.code)

        PriceField(value: $editProductViewModel.price)
    }

    .navigationBarTitle("Edit product", displayMode: .inline)
    .navigationBarItems(trailing: Button(action: updateProduct) {
        Text("Save").foregroundColor(.blue)
    })

    .alert(isPresented: $showAlert) {
        Alert(title: Text("Error"),
            message: Text("The product couldn't be updated"),
            dismissButton: .default(Text("Dismiss")))
    }
}
}
}

```

Repare também que há sessão `alert`, que será exibida se a alteração do produto falhar, algo que pode acontecer se o usuário alterar para um código de um outro produto que tenha o mesmo valor.

7.4.4 - Adaptando a view principal

Na *view* principal, é necessário criar uma nova função dentro da *struct ProductListView* para ser chamada quando a alteração do produto for concluída. Dessa forma, será possível alterar a lista de produtos sem que o usuário tenha que apertar o botão de *refresh*. Veja como deve ficar essa nova função:

```

private func updateProduct(product: Product) {
    if let productToBeUpdatedIndex = self.productViewModel
        .products.firstIndex(where: { $0.id == product.id }) {
        self.productViewModel
            .products[productToBeUpdatedIndex] = ProductModel(product: product)
    }
}

```

Perceba que primeiramente é obtido o índice da lista de produtos com o mesmo id do produto que foi alterado, para que ele possa então ser atualizado na mesma lista.

A última alteração a ser feita é substituir o conteúdo do parâmetro *destination* do componente *NavigationLink*, dentro do *ForEach* da construção da lista no *body* para que ele possa chamar a nova *view* de edição do produto. Veja como deve ficar:

```
NavigationLink(  
    destination:  
        EditProductView(product: product.product) {  
            product in  
            updateProduct(product: product)  
        }) {
```

Veja que onde havia um `Text` no parâmetro `destination` agora há a criação de `EditProductView`, criado na seção anterior. Essa nova `view` recebe o produto a ser alterado e o closure para ser invocado quando a operação for concluída com sucesso, que nesse caso é a função `updateProduct` criada recentemente.

Para testar essa implementação, execute a aplicação em um simulador iOS ou no Xcode Preview. Quando estiver rodando, clique sobre um produto para ver a tela de detalhes do produto, que também permite sua alteração. Para salvar as alterações, pressione o botão `Save`, para voltar à tela principal, que já deverá ser atualizada com o que foi feito na tela de edição.

7.5 - Conclusão

E esse capítulo conclui esse projeto que mostrou vários conceitos, como navegação através de listas, consumo de serviços REST, tratamento de erros e atualização da tela de acordo com a resposta de tais serviços.

Todo o projeto foi construído em camadas, que puderam ser reutilizadas para cada operação que foi feita, totalmente aderente aos conceitos do SwiftUI.

O próximo capítulo irá começar a introduzir conceitos da construção de aplicativos integrados com o Google Firebase, a começar pelo recebimento de notificações pelo Firebase Cloud Messaging.

8 - Recebendo mensagens via Firebase Cloud Messaging

Esse livro introduz, a partir desse capítulo, conceitos sobre como criar aplicativos iOS que estejam conectados a algum serviço de *cloud computing*, a começar pelo **Firebase Cloud Messaging (FCM)**, 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 iOS 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 receber a notificação, poderá clicá-la para navegar para dentro da aplicação;
- De dentro da aplicação, a mensagem será interpretada e exibida em uma tela ao usuário;

Os conceitos de *viewModel* e objetos que podem ser observados pela *view* construída com SwiftUI ainda continuarão a ser utilizados. A ideia é mostrar como o mecanismo de registro e recebimento de notificação do cliente FCM da aplicação pode facilmente se integrar em um projeto feito com SwiftUI.

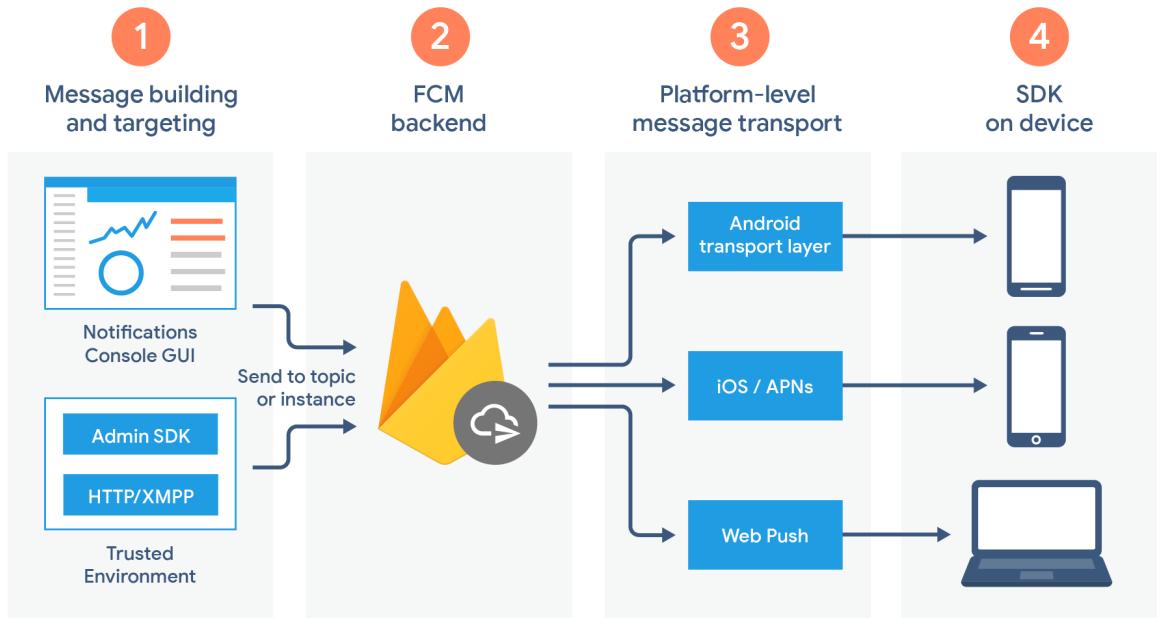
Embora possa parecer bem simples, a ideia é apenas criar uma base para que o leitor possa desenvolver aplicações mais complexas utilizando esse mecanismo.

8.1 - 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 do envio da 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 iOS, mas se tiver interesse em aprender a criar uma aplicação de backend para **envio de notificações pelo FCM**, consulte o seguinte [livro²⁴](#).

Do ponto de vista da construção do sistema de envio de notificações para aplicativos iOS, 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 iOS 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 iOS**: esse processo é bem simples e consiste em adicionar um arquivo de configuração gerado no passo anterior. Com esse arquivo a aplicação iOS fica toda

²⁴<https://www.casadocodigo.com.br/products/livro-gae>

configurada e preparada para acessar os serviços do Firebase sobre o projeto criado no primeiro passo.

Analisando a figura anterior, o processo de envio de notificações funciona da seguinte forma:

- 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.

8.2 - Arquitetura do projeto iOS com FCM

Como dito no início desse capítulo, o intuito desse projeto é apenas exibir a notificação ao usuário, quando ela for recebida do FCM, e exibir seu conteúdo na tela. A ideia é deixar a implementação o mais simples possível, inclusive em termos visuais, para manter o foco na criação do mecanismo de registro e recebimento de mensagens do FCM.

A figura a seguir exibe a arquitetura de como projeto será organizado:



Arquitetura do projeto



A mensagem a ser enviada ao app é a mesma utilizada na implementação do **provedor de envio de mensagens**, criado como exemplo de aplicação de backend para envio de notificações pelo FCM. Sua construção pode ser vista nesse [livro²⁵](#).

Na figura anterior, a classe App Delegate é uma implementação que atende a eventos do sistema iOS, como notificações e ações do usuário. Ela também será utilizada para se registrar no FCM, quando a aplicação inicia, bem como receber a notificação, no **passo 1**.

A *struct Content View* será utilizada para exibição da mensagem recebida do FCM, já interpretada dentro de App Delegate. A ideia é utilizar um *ObservedObject* para que o SwiftUI possa redesenhar a tela, no **passo 2**, assim que a mensagem interpretada for publicada no **passo 1**.

As seções a seguir tratam da criação do projeto no Xcode, bem como o que será necessário de ser feito no console do Firebase.

8.3 - Criando o novo projeto no Xcode

Para esse aplicativo será criado um novo projeto no Xcode, da mesma forma como foi feito para os outros dois criados anteriormente nesse livro. Para isso, abra o Xcode e selecione a opção para criar um novo projeto. Configure-o como mostra a figura a seguir:

²⁵<https://www.casadocodigo.com.br/products/livro-gae>

Product Name:

Team: ^ ▼

Organization Identifier:

Bundle Identifier:

Interface: ^ ▼

Life Cycle: ^ ▼

Language: ^ ▼

Criando o novo projeto

O valor do campo `Bundle Identifier` será utilizado na próxima seção, durante a criação do projeto no Firebase, onde esse aplicativo será registrado.

8.4 - Adicionando o Firebase ao projeto criado

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.

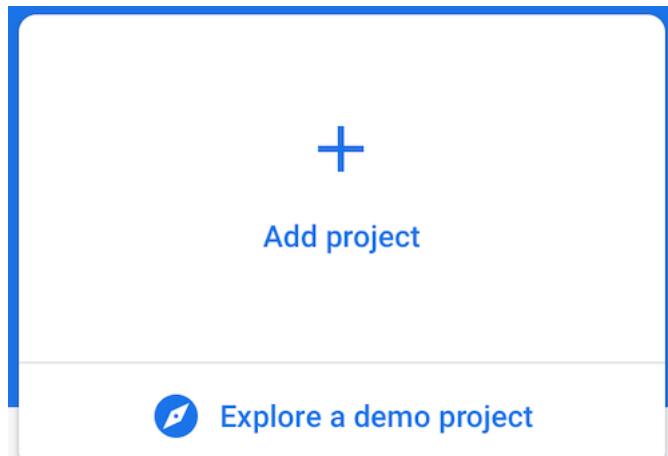
Para começar a criar um projeto no Firebase, acesse o seguinte [link²⁶](#) e clique no opção `Go to console`, localizado no canto superior direito da página. É necessário estar logado com uma conta do Google para começar esse processo.

8.4.1 - Criando o projeto no Firebase

Para criar um novo projeto no Firebase, vá até seu [console²⁷](#), efetue o login com a sua conta ou crie uma nova, caso não tenha. Dentro do console, selecione a opção para adicionar um novo projeto, na tela principal, como mostra a figura a seguir:

²⁶<http://firebase.google.com/>

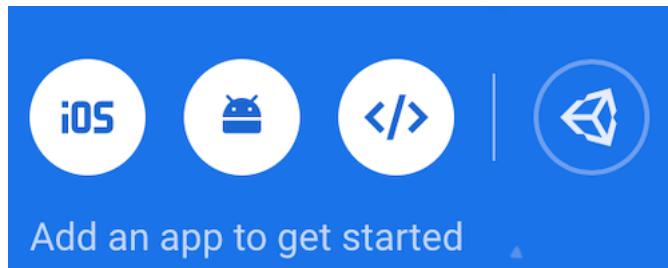
²⁷<https://console.firebaseio.google.com>



Novo projeto

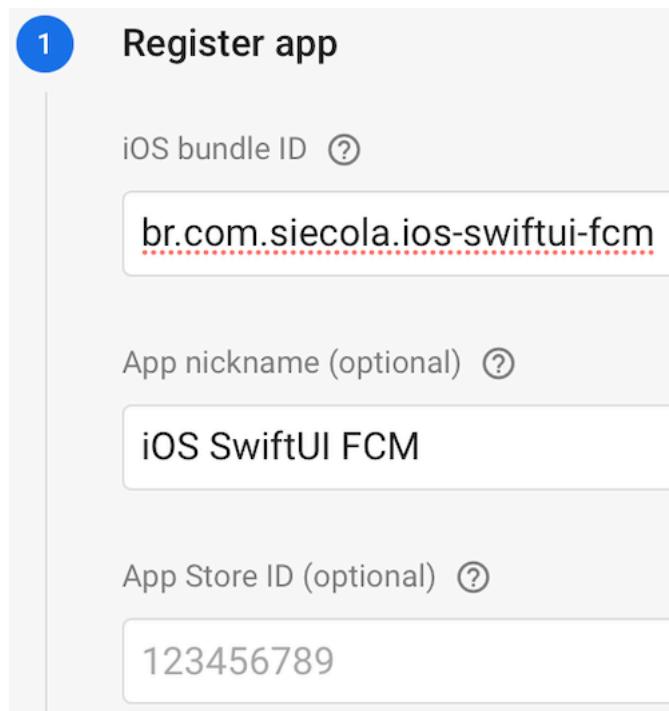
Na primeira tela, escolha o nome que desejar para o novo projeto e clique em Continue. Em sua segunda tela, desmarque a opção Enable Google Analytics for this project, pois essa opção não será necessária para a esse projeto. Em seguida clique em Create Project.

Depois que o projeto for criado, clique em Continue para ser redirecionando ao seu console. Na tela principal do projeto, clique na opção iOS para adicionar um novo aplicativo a esse projeto do Firebase:

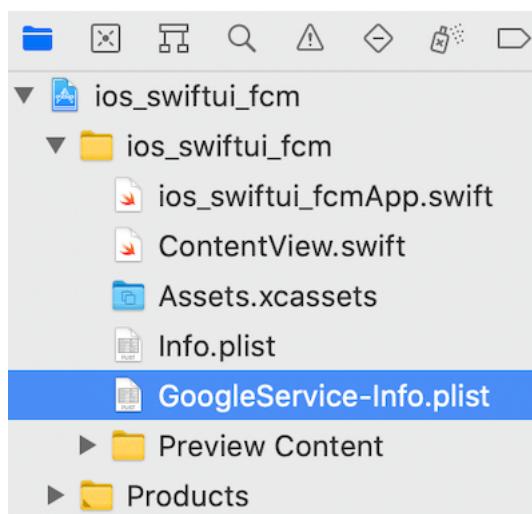


Adicionando um novo app iOS

Na tela de registro do novo app, preencha o campo iOS bundle ID com a identificação do projeto que foi criado no Xcode, que nesse caso foi br.com.siecola.ios-swiftui-fcm, como pode ser visto na figura a seguir:



Para continuar, clique em Register app. Na segunda tela de registro do app, o Firebase oferece o arquivo GoogleService-Info.plist para ser baixado. Baixe-o e copie para dentro da pasta do projeto, no mesmo lugar onde está o arquivo Info.plist, como pode ser visto na figura a seguir:



Copiando o arquivo GoogleService-Info.plist

Para prosseguir com o registro do aplicativo no Firebase, clique em Next no console do Firebase. As próximas seções desse processo no console do Firebase são apenas instruções de como prosseguir com o projeto do Xcode, que serão explicadas a seguir. Por isso, continue clicando em Next até finalizar por completo o processo de registro do app no Firebase.

8.4.2 - Configurando o projeto para trabalhar com o Firebase

Agora é hora de adicionar as bibliotecas do Firebase ao projeto do Xcode. Para isso, feche-o e abra um terminal de comandos na pasta raiz do projeto. Dentro dessa pasta, execute o seguinte comando para criar o arquivo `Podfile`, onde as bibliotecas do Firebase serão adicionadas:

```
pod init
```

Agora abra o arquivo `Podfile` com um editor de texto e substitua seu conteúdo pelo trecho a seguir:

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, '14.0'

target 'ios_swiftui_fcm' do
    use_frameworks!

    # Pods for ios_swiftui_fcm
    pod 'Firebase/Messaging'

end
```

Repare que foi adicionada a biblioteca `Firebase/Messaging` ao arquivo `Podfile` para que o projeto possa trabalhar com esse serviço do Firebase.

Salve e feche o arquivo. De volta ao terminal de comandos, digite o comando a seguir para instalar a biblioteca `Firebase/Messaging` ao projeto:

```
pod install
```

Depois que a biblioteca for baixada e instalada no projeto, abra o arquivo com extensão `.xcworkspace` utilizando o Xcode, pois agora o projeto criado anteriormente pertence a um *workspace*, juntamente com as bibliotecas do Firebase que foram adicionadas.

8.5 - Configurando o FCM para funcionar com o APNs

Para que o FCM consiga enviar notificações a dispositivos iOS utilizando o APNs, é necessário gerar os seguintes itens no [console²⁸](#) do desenvolvedor da Apple:

- Chave de autenticação;
- Registro do App ID;

²⁸<https://developer.apple.com>

- Perfil de provisionamento.

Para isso, faça o login da sua conta de desenvolvedor da Apple no [console²⁹](#) e siga os passos das sessões adiante.

8.5.1 - Gerando a chave de autenticação

Dentro do console do desenvolvedor da Apple, vá até a opção Certificates, IDs & Profiles. Nessa página, selecione a opção Keys e em seguida no botão para adicionar uma nova chave, localizado ao lado do título da seção, como mostra a figura a seguir:



Gerando a chave de autenticação

Na tela de registro da nova chave, digite um nome para ela e selecione a opção Apple Push Notifications service (APNs), como mostra a figura a seguir:

Register a New Key

Key Name

iOS SwiftUI FCM

You cannot use special characters such as @, &, *, !, ", -, .

ENABLE

NAME

SERVICE



Apple Push Notifications service (APNs)

Establish conn
Notification se

Registrando a nova chave

Clique em Continue e em seguida em Register. Depois da chave ter sido registrada, clique no botão Download para baixá-la.

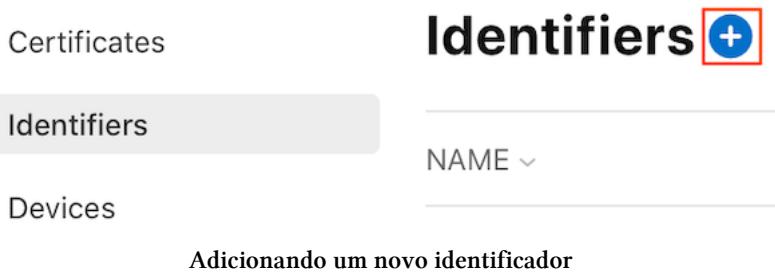
²⁹<https://developer.apple.com>



Salve a chave em um local seguro para depois ser utilizada na configuração do projeto criado no FCM.

8.5.2 - Registrando o App ID

Dentro do console do desenvolvedor da Apple, vá até a opção Certificates, IDs & Profiles. Nessa página, selecione a opção Identifiers e em seguida no botão para adicionar um novo identificador, localizado ao lado do título da seção, como mostra a figura a seguir:



Na tela que aparecer, selecione a opção App IDs e clique em Continue:

[< All Identifiers](#)

Register a new identifier

• App IDs

Register an App ID to enable your app, app your app in a provisioning profile. You can settings later.

[Novo App ID](#)

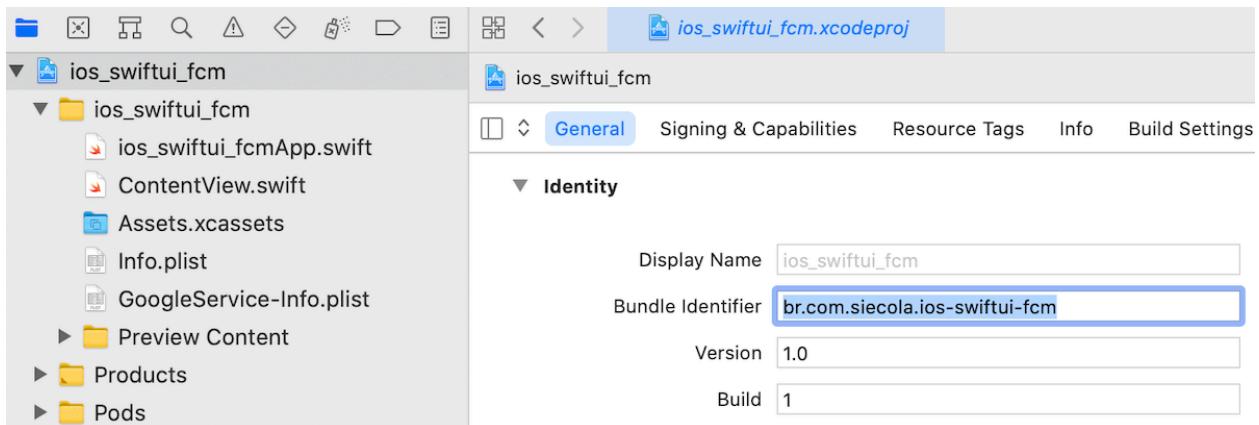
No próximo passo, selecione a opção App e clique em Continue. Nessa tela, preencha a descrição do novo App ID, bem como o Bundle ID, como mostra a figura a seguir:

Register an App ID

Platform iOS, macOS, tvOS, watchOS	App ID Prefix Q7A92P5WPW (Team ID)
Description <input type="text" value="iOS SwiftUI FCM"/>	
You cannot use special characters such as @, &, *, :, ", -, .	
Bundle ID <input checked="" type="radio"/> Explicit <input type="radio"/> Wildcard <input type="text" value="br.com.siecola.ios-swiftui-fcm"/>	
We recommend using a reverse-domain name (com.domainname.appname). It cannot contain special characters.	

59_apns_03

O valor do campo Bundle ID deve ser o mesmo configurado no projeto e pode ser verificado na aba General das configurações do projeto, dentro do Xcode, como mostra a figura a seguir:



Visualizando o Bundle Identifier

Ainda na tela de registro do App ID, selecione a opção Push Notifications, que é o serviço que será utilizado pelo FCM.

Para finalizar, clique em Continue e em seguida em Register. O novo App ID deverá aparecer na lista Identifiers.

8.5.3 - Criando o perfil de provisionamento

Para que o aplicativo possa ser testado em tempo de desenvolvimento, ou seja, sem ainda ter sido publicado na loja da Apple, é necessário criar um perfil de provisionamento.

Dentro do console do desenvolvedor da Apple, vá até a opção Certificates, IDs & Profiles. Nessa página, selecione a opção Profiles e em seguida no botão para adicionar um novo perfil, localizado ao lado do título da seção, como mostra a figura a seguir:



Na primeira tela, selecione a opção iOS App Development e em seguida em Continue.

Na tela seguinte, selecione o App ID registrado na seção anterior e clique em Continue. Depois, selecione a máquina de desenvolvimento registrada com o Xcode que deseja deixar esse perfil de provisionamento habilitado e clique em Continue.

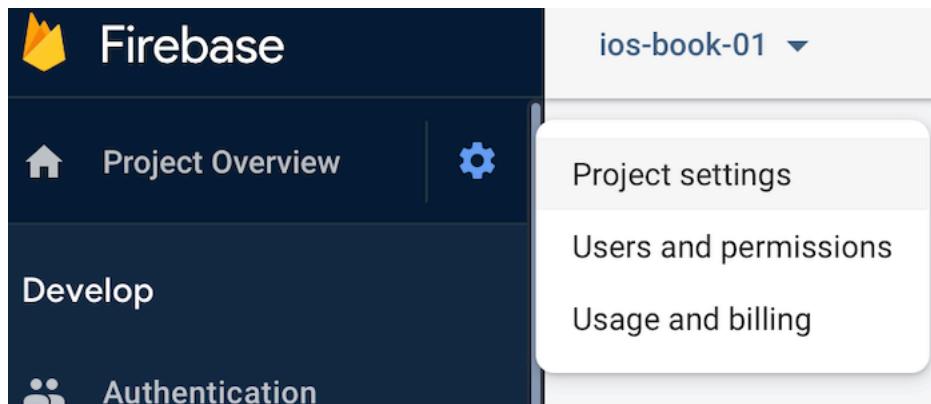
Da mesma forma, selecione o dispositivo em que deseja fazer o teste do envio de notificações pelo APNs e clique em Continue.

Na última tela, digite um nome para o perfil de provisionamento e clique em Generate. Depois que o perfil for gerado, clique em Download para baixá-lo.

Para finalizar, vá até o local onde o arquivo foi baixado e dê um duplo clique para instalá-lo em sua máquina.

8.5.4 - Fazendo o upload da chave de autenticação no Firebase

Agora que tudo já foi criado na conta de desenvolvedor da Apple, é possível fazer o *upload* do arquivo com a chave de autenticação no FCM, para que ele possa se comunicar com o APNs e enviar notificações. Para isso, volte até o console do projeto criado no Firebase e selecione a opção Project Settings no menu de configuração do projeto:



Configurações do projeto Firebase

Na tela que abrir, selecione a aba Cloud Messaging. Nessa tela haverá uma seção com o app que foi adicionado ao projeto do Firebase. Clique no botão Upload da opção APNs Authentication Key e localize o arquivo com a chave de autenticação gerado na seção anterior, de extensão .p8.

Na tela de *upload* da chave de autenticação, também será necessária a identificação da chave que foi gerada. Essa informação pode ser obtida no console do desenvolvedor da Apple, na seção Certificates, Identifiers & Profiles -> Keys. Nessa tela, clique na chave que foi gerada e copie o conteúdo do campo Key ID, colando no campo de mesmo nome no console do Firebase.

Outra informação necessária ainda na tela de *upload* da chave de autenticação no Firebase é o Team ID. Isso pode ser obtido no console do desenvolvedor da Apple, no canto superior direito, logo após o nome do proprietário da conta. Cole esse valor no console do Firebase, no campo de mesmo nome.

Para finalizar, no console do Firebase, clique em Upload para inserir o arquivo com a chave de autenticação no Firebase.

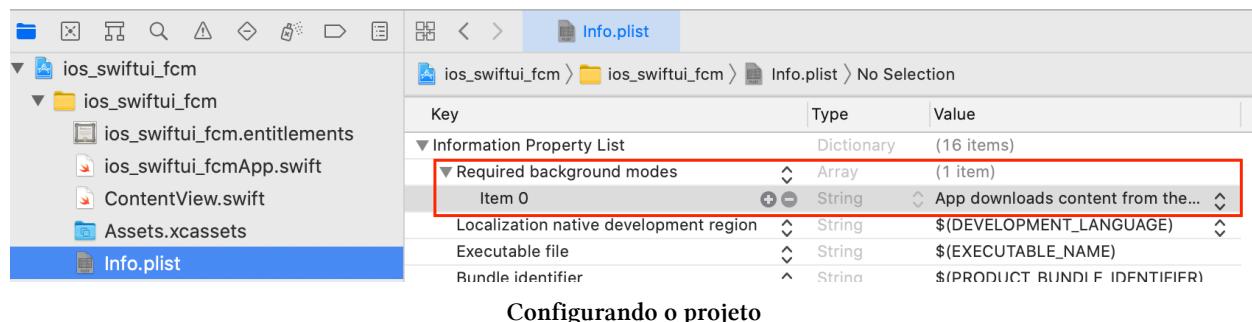
Agora o projeto do Firebase está totalmente configurado para enviar mensagens pelo APNs.

8.6 - Registrando no FCM para receber notificações

Antes de começar o código para o app se registrar no FCM e receber notificações dele, são necessárias algumas configurações no projeto para isso, como descrito nas seções a seguir.

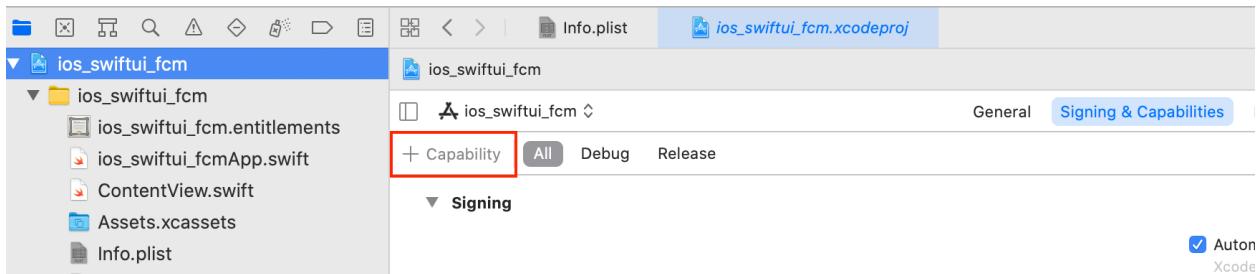
8.6.1 - Configurando o projeto no Xcode

Dentro do projeto do Xcode, selecione o arquivo Info.plist e adicione a seguinte configuração a ele:



Nesse arquivo, adicione a seção Required Background modes com o valor App downloads content from the network.

Ainda no projeto do Xcode, selecione a aba Signing & Capabilities. Nessa aba, adicione a configuração Push Notifications, clicando no botão + Capability, como pode ser visto na figura a seguir:



Adicionando a config de push notification

Agora já é possível começar o código para registrar no FCM, como será visto na seção a seguir.

8.6.2 - Registrando o app para receber notificações

Como mostrado no início da seção 8.2 desse capítulo, o projeto precisará de uma classe que irá tratar eventos do sistema iOS, bem como cuidar do processo de registro no FCM e tratar seus eventos quando uma notificação chegar ao dispositivo. Essa classe será registrada de uma forma especial na inicialização do app, dentro da *struct ios_swiftui_fcmApp*, que implementa o protocolo App.

Para começar, crie uma nova classe na raiz do projeto, com o nome de *AppDelegate*, como mostra o trecho a seguir:

```
import Foundation
import Firebase

//This is implementation is based on https://github.com/firebase/quickstart-ios

class AppDelegate: UIResponder, UIApplicationDelegate {
    let gcmMessageIDKey = "gcm.message_id"
    let orderInfoKey = "gcm.notification.orderInfo"

}
```

Veja que ela estende de *UIResponder* e está em conformidade com o protocolo *UIApplicationDelegate* para atender a eventos do sistema e ser invocado na inicialização do app.

As duas constantes privadas serão utilizadas para buscar as informações nas notificações recebidas do FCM, como a identificação única da mensagem e a mensagem em si que foi enviada. Ambas serão utilizadas a seguir.

A primeira função a ser criada nessa classe é a que será invocada na inicialização do app. Nela será possível dar início ao processo de registro do dispositivo no FCM, assim como a configuração do mecanismo de notificação do usuário, quando uma mensagem chegar e a atribuição de um *delegate* para tal função. Veja como ela deve ficar:

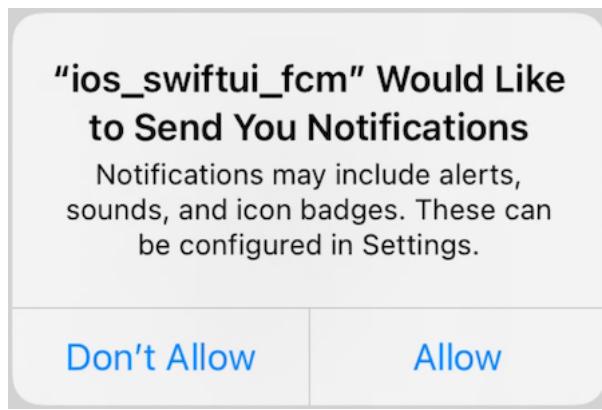
```
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    FirebaseApp.configure()

    Messaging.messaging().delegate = self
    UNUserNotificationCenter.current().delegate = self

    let authOptions: UNAuthorizationOptions = [.alert, .badge, .sound]
    UNUserNotificationCenter.current().requestAuthorization(
        options: authOptions,
        completionHandler: {_, _ in })
    application.registerForRemoteNotifications()

    return true
}
```

Veja que a primeira linha da função inicializa o Firebase dentro da aplicação. As linhas a seguir solicitam, na primeira vez que a aplicação é executada, a permissão do usuário para enviar notificações a ele, como pode ser visto na figura a seguir:



Solicitação de permissão do app para notificações

Uma vez que o usuário aceita a solicitação, o app está apto a enviar notificações ao usuário, como será visto na próxima seção.

Ainda serão criadas nessa classe *extensions* de `Messaging` e `UNUserNotificationCenter` para lidarem com eventos de recebimento de notificação e registro de token.

As demais funções a serem criadas nessa classe devem existir para que ela esteja em conformidade com o protocolo assumido. São elas:

```
func application(_ application: UIApplication,
                 didReceiveRemoteNotification userInfo: [AnyHashable: Any]) {
    if let messageID = userInfo[gcmMessageIDKey] {
        print("Message ID: \(messageID)")
    }
}

func application(_ application: UIApplication,
                 didReceiveRemoteNotification userInfo: [AnyHashable: Any],
                 fetchCompletionHandler completionHandler:
                     @escaping (UIBackgroundFetchResult) -> Void) {
    if let messageID = userInfo[gcmMessageIDKey] {
        print("Message ID: \(messageID)")
    }
    completionHandler(UIBackgroundFetchResult.newData)
}

func application(_ application: UIApplication,
                 didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {
    print("APNs token retrieved: \(deviceToken)")
}

func application(_ application: UIApplication,
                 didFailToRegisterForRemoteNotificationsWithError error: Error) {
    print("Unable to register for remote notifications: \(error.localizedDescription")
}
```

Como dito anteriormente, serão criadas extensões para tratar desses eventos a seguir, onde seus detalhes serão explicados.

Para que a aplicação seja notificada quando ela se registrar no FCM, crie a extensão AppDelegate dentro do arquivo AppDelegate.swift, mas fora da classe de mesmo nome. Ela deverá estar em conformidade com o protocolo MessagingDelegate, como no trecho a seguir:

```
extension AppDelegate : MessagingDelegate {  
    func messaging(_ messaging: Messaging,  
                  didReceiveRegistrationToken fcmToken: String?) {  
        print("FCM registration token: \(String(describing: fcmToken))")  
  
        //Enviar a informação para o servidor  
    }  
}
```

Veja que nessa função o token de registro do FCM é recebido. Com ele será possível enviar mensagens diretamente ao dispositivo, como será visto na próxima seção.



A utilização desse token pelo aplicativo depende da implementação do sistema sendo construído. Ele poderia ser enviado a um servidor, para que esse pudesse associar um usuário ao seu registro no FCM, e dessa forma enviar notificações a esse usuário especificamente.

A ideia aqui é apenas imprimir o token de registro do FCM no console do Xcode, para que ele possa ser utilizado na seção seguinte.

8.7 - Recebendo mensagens do FCM

Para tratar a mensagem recebida, interpretando-a e publicando-a para as demais partes do app, são necessários os seguintes passos:

- Criar uma *struct* modelo para representar o conteúdo da notificação;
- Criar um *viewModel* para ser observado quando uma nova mensagem for publicada nele;
- Fazer com que a *view* principal observe as mudanças do *viewModel* para redesenhar a tela com a informação da mensagem recebida.

Essa seção detalha esses passos, a começar pelo modelo da notificação.

8.7.1 - Criando o modelo da notificação

O conteúdo da notificação a ser recebida na mensagem enviado pelo FCM terá o seguinte formato:

```
{  
    "infoReason": "Status de pedido",  
    "orderID": 3,  
    "userEmail": "matilde@siecola.com.br",  
    "newOrderStatus": "Novo pedido"  
}
```

Na verdade, como dito anteriormente, o formato do conteúdo da mensagem pode ser qualquer um. O exemplo anterior é o que será utilizado nos testes dessa seção. Ele representa a informação de status de um pedido de um *e-commerce* fictício.

Para representar esse modelo, crie uma nova *struct* de nome `SalesNotification` dentro de uma nova pasta chamada `Models`, como no trecho a seguir:

```
import Foundation  
  
struct SalesNotification: Decodable {  
    let newOrderStatus: String  
    let userEmail: String  
    let orderID: Int  
    let infoReason: String  
}
```

A ideia é interpretar o JSON recebido e criar uma instância de `SalesNotification` para representá-lo, por isso a *struct* implementa o protocolo `Decodable`.

8.7.2 - Criando o view model para a mensagem recebida

Para atualizar a tela principal com a informação recebida na notificação do FCM, é necessário criar um *viewModel* que possa ser observado pela *view* principal. Esse *viewModel* então poderá ser atualizado a cada mensagem recebida, fazendo com que a *view* seja notificada dessa mudança e exiba a cada mensagem.

Para isso, crie uma classe chamada `SalesMessageViewModel`, em conformidade com o protocolo `ObservableObject`, em uma nova pasta chamada `ViewModels`, como pode ser visto no trecho a seguir:

```
import Foundation

class SalesMessageViewModel: ObservableObject {

    private init() {}

    static let shared = SalesMessageViewModel()

    @Published var newOrderStatus: String = ""
    @Published var userEmail: String = ""
    @Published var orderID: Int = 0
    @Published var infoReason: String = ""
    @Published var messageId: String = ""

}
```

Veja que a estratégia aqui é utilizar um *singleton* dessa classe, uma vez que ele deverá ser utilizado dentro de `AppDelegate`. Por isso, volte até o arquivo `AppDelegate.swift` e crie um novo atributo privado no início da declaração da classe `AppDelegate`, como o trecho a seguir:

```
private var salesMessageViewModel: SalesMessageViewModel = .shared
```

Esse atributo agora poderá ser preenchido quando uma mensagem for recebida e interpretada.

8.7.3 - Recebendo e interpretando a mensagem

Dentro do arquivo `AppDelegate.swift`, e fora da classe de mesmo nome, crie a extensão `AppDelegate`, implementando o protocolo `UNUserNotificationCenterDelegate`, como no trecho a seguir:

```
extension AppDelegate : UNUserNotificationCenterDelegate {

}
```

A primeira função a ser criada deve ser a que será chamada quando uma mensagem for recebida quando o app **estiver aberto**. Basicamente ela deverá receber a mensagem, publicar para outras partes do app que estiverem interessadas e invocar o *handler* que a tarefa foi finalizada. Veja como deve ficar no trecho a seguir:

```
func userNotificationCenter(_ center: UNUserNotificationCenter,
                           willPresent notification: UNNotification,
                           withCompletionHandler completionHandler:
                           @escaping (UNNotificationPresentationOptions) -> Void) {
    publishMessage(userInfo: notification.request.content.userInfo)

    completionHandler([ [.alert, .sound] ])
}
```

A mensagem recebida do FCM está contida dentro de `notification.request.content.userInfo`, como pode ser visto na primeira linha da função. Mais adiante esse objeto será aberto e interpretado.

A função será executada através da instrução `completionHandler([[.alert, .sound]])` gerando uma notificação ao usuário de que a mensagem foi recebida.

Veja que existe a chamada a uma outra função de nome `publishMessage`. Ela será criada na próxima seção e será responsável por publicar a mensagem às demais partes do app.

A próxima função a ser criada nessa extensão é a que será invocada quando o **app estiver fechado ou em background**, como pode ser visto no trecho a seguir:

```
func userNotificationCenter(_ center: UNUserNotificationCenter,
                           didReceive response: UNNotificationResponse,
                           withCompletionHandler completionHandler:
                           @escaping () -> Void) {
    publishMessage(userInfo: response.notification.request.content.userInfo)

    completionHandler()
}
```

Para finalizar, a função `publishMessage` deve ser criada dentro da
`extension AppDelegate : UNUserNotificationCenterDelegate`.

Elá deverá interpretar a mensagem recebida, transformando-a em um objeto do tipo `SalesNotification` e publicar sua informação através de `SalesMessageViewModel`, como mostra o trecho a seguir:

```
private func publishMessage(userInfo: [AnyHashable: Any]) {
    print(userInfo)

    if let messageID = userInfo[gcmMessageIDKey] {
        print("Message ID: \(messageID)")
        salesMessageViewModel.messageId = "\(messageID)"
    }

    if let notification = userInfo[orderInfoKey] {
        let data = "\(notification)".data(using: .utf8)!

        if let salesNotification = try? JSONDecoder()
            .decode(SalesNotification.self, from: data) {
            salesMessageViewModel.newOrderStatus = salesNotification.newOrderStatus
            salesMessageViewModel.userEmail = salesNotification.userEmail
            salesMessageViewModel.orderID = salesNotification.orderID
            salesMessageViewModel.infoReason = salesNotification.infoReason
        }
    }
}
```

A primeira linha imprime a mensagem recebida no log do Xcode, o que serve apenas para que o desenvolvedor entenda melhor seu formato, que possui valores que podem ser resgatados através de chaves.

O primeiro valor a ser lido da mensagem é o que contém a chave `gcm.message_id`. Isso é feito através da constante `gcmMessageIDKey`. O campo `messageID` serve para identificar unicamente a mensagem dentro do FCM, facilitando o *tracing* dela ao longo de todo o processo, desde a publicação até seu recebimento, como poderá ser observado a seguir.

Logo a seguir o conteúdo da notificação que foi publicado no FCM é retirado do *payload* através da chave `gcm.notification.orderInfo`, contido na constante `orderInfoKey`. Tendo esse conteúdo resgatado, ele é interpretado e escrito em uma instância de `SalesNotification`.

O último passo dessa função é preencher os atributos de `salesMessageViewModel` para que a tela que irá observá-lo seja notificada para exibir seu conteúdo.

8.7.4 - Registrando a classe AppDelegate

Agora que a classe `AppDelegate` está pronta, ela deve ser registrada na aplicação, para que ela realmente atue recebendo os eventos que nela foram tratados. Para isso, abra o arquivo principal da aplicação, com o nome do projeto, que no caso desse exemplo é `ios_swiftui_fcmApp`. Dentro dela há uma *struct* que está em conformidade com o protocolo `App`. Dentro dela, registre a classe `AppDelegate`, anotando-a com `@UIApplicationDelegateAdaptor`, como pode ser visto no trecho a seguir:

```
import SwiftUI

@main
struct ios_swiftui_fcmApp: App {

    @UIApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Isso fará com que a função de AppDelegate a seguir seja executada toda vez que a aplicação for iniciada:

```
func application(_ application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions:
                     [UIApplication.LaunchOptionsKey: Any]?) -> Bool
```

Lembrando, essa é a função que registra o dispositivo no FCM, como foi explicado na seção anterior.

8.7.5 - Construindo a view para exibir a notificação

Todo mecanismo para o registro e recebimento de notificações pelo FCM já foi implementado. Agora é hora de construir a interface gráfica que irá exibir a mensagem quando ela for recebida pelo dispositivo. Para isso, abra o arquivo `ContentView.swift` e crie o atributo do tipo `SalesMessageViewModel`, anotando-o com `@ObservedObject`, como no trecho a seguir:

```
struct ContentView: View {
    @ObservedObject private var salesMessageViewModel:
        SalesMessageViewModel = .shared
```

Isso fará com que o SwiftUI atualize a tela toda vez que alguma informação for alterada dentro de `SalesMessageViewModel`, algo que acontecerá dentro da função `publishMessage` de `AppDelegate`.

Ainda dentro da `struct ContentView`, crie os campos para exibir os atributos da mensagem:

```
var body: some View {
    VStack(alignment: .leading) {
        Text("Message Id: \(salesMessageViewModel.messageId)")
        Text("Reason: \(salesMessageViewModel.infoReason)")
        Text("Order Id: \(salesMessageViewModel.orderID)")
        Text("Status: \(salesMessageViewModel.newOrderStatus)")
        Text("Email: \(salesMessageViewModel.userEmail)")
    }
}
```

Veja que todos os campos da mensagem recebida são exibidos, inclusive a identificação única da mensagem.

A sessão a seguir irá detalhar os passos para que toda a implementação possa ser testada, incluindo o passo para o envio de uma notificação pelo FCM, utilizando o Postman.

8.7.6 - Testando o recebimento de notificações do FCM

O teste da implementação do projeto criado nesse capítulo se divide em duas ações:

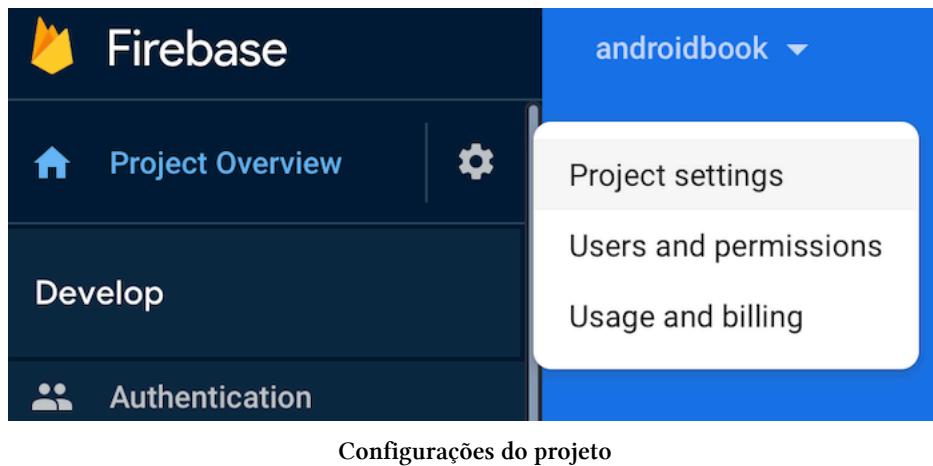
- **Registro no FCM:** quando o aplicativo se registrar no FCM, o token de registro deverá aparecer no log do Xcode;
- **Recebimento de mensagens enviadas pela API do FCM:** através da API do FCM, será possível enviar uma mensagem ao dispositivo executando o aplicativo, utilizando o token de registro citado anteriormente.

Para começar os testes, execute a aplicação em um dispositivo real. Após alguns segundos o token de registro no FCM deverá aparecer no log do Xcode, em uma mensagem semelhante a seguir:

```
FCM registration token: Optional("eeDh2Sd5HknG1zdC1w1UBD:APA91bFKGr99Cpj7q1Es0oexCeo\x
Xf9qinT6GuI-pvCouR8AKueLR1zNc_QZVv_U7t068XGEbPheXZwLF5fvUNZ8c018jv6vMP-1tAQojY6Q4tDV\x
3ySXN")
```

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:



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 mostra 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",
  "notification": {
    "body": "Status do pedido",
    "title": "Provedor de Mensagens",
    "orderInfo": {
      "infoReason": "Status de pedido",
      "orderID": 3,
      "userEmail": "matilde@siecola.com.br",
      "newOrderStatus": "Novo pedido"
    }
  },
}
```

```
    "priority": "high",
    "content_available": true
}
```

O valor do token de registro do FCM deve ser inserido no campo `to` do JSON anterior.

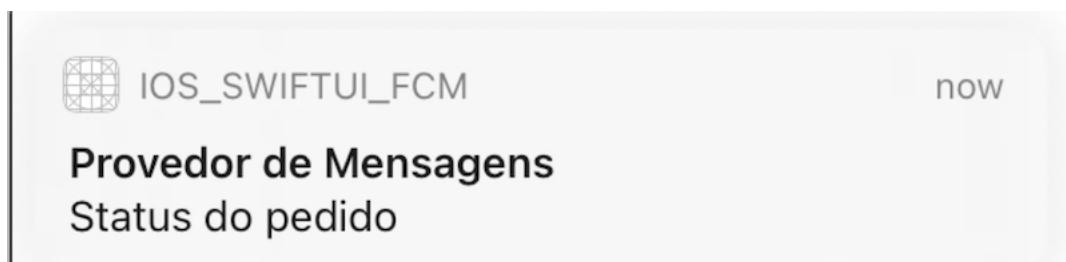
Depois de configurar o Postman com as informações apresentadas aqui, clique no botão `Send`. Caso a mensagem tenha sido aceita pelo Firebase, a resposta deverá ser algo no seguinte formato:

```
{
  "multicast_id": 3529387488852107593,
  "success": 1,
  "failure": 0,
  "canonical_ids": 0,
  "results": [
    {
      "message_id": "1603214613006763"
    }
  ]
}
```

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. Ele terá o mesmo valor do campo `messageID` recebido pelo app, na classe `AppDelegate`, o que facilita rastrear a mensagem, desde seu envio até seu recebimento.

Quando a mensagem for recebida pelo dispositivo, uma notificação será gerada no sistema iOS, semelhante a da figura a seguir:



Veja que as informações dos campos `body` e `title` da mensagem que foi publicada aparecem na notificação do sistema.

No log do Xcode, é possível ver a notificação completa recebida do FCM, como no trecho a seguir:

```
[AnyHashable("gcm.notification.orderInfo"): {"newOrderStatus": "Novo pedido", "orderId": 3, "userEmail": "matilde@siecola.com.br", "infoReason": "Status de pedido"}, AnyHashable("google.c.a.e"): 1, AnyHashable("gcm.message_id"): 1603214613006763, AnyHashable("google.c.sender.id"): 19283323559, AnyHashable("aps"):  
    alert = {  
        body = "Status do pedido";  
        title = "Provedor de Mensagens";  
    };  
    "content-available" = 1;  
}  
Message ID: 1603214613006763
```

O campo `message_id` da resposta obtida pelo Postman ao acessar a API do FCM aparece na notificação recebida pelo dispositivo.

Veja também que a notificação completa recebida do FCM contém algumas chaves, dentre elas `gcm.notification.orderInfo`, que contém o objeto `orderInfo` enviado pela API do FCM, no corpo da requisição, assim como a chave `gcm.message_id` com a identificação única da mensagem enviada.

Dessa forma, a mensagem pode ser interpretada e exibida ao usuário na tela principal do app, como mostra a figura seguir:

Message Id: 1603214613006763
Reason: Status de pedido
Order Id: 3
Status: Novo pedido
Email: matilde@siecola.com.br

Mensagem exibida ao usuário

Dessa forma, toda a implementação pode ser testada!

8.8 - Conclusão

Esse capítulo mostrou como é simples construir um aplicativo que se integra com o Firebase Cloud Messaging para o recebimento de notificações por essa plataforma.

A arquitetura do projeto foi construída seguindo o que já foi visto nos capítulos anteriores, ou seja, utilizando o que o SwiftUI tem de melhor a oferecer para a construção da interface gráfica.

O próximo capítulo continua com um outro aplicativo, que será utilizado para se integrar com outros serviços como Firebase Authentication, para autenticação de usuários de uma forma fácil e sem a necessidade de construção de um backend para isso.

9 - Autenticando usuários com Firebase Authentication

Construir um aplicativo iOS 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 iOS, 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 serem liberados para os usuários que são donos dele, ou àqueles que compartilham do acesso a ele. Isso significa que os usuários da aplicação iOS 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.

Esse e o próximo capítulo introduzem os dois próximos serviços do Firebase que serão trabalhados: **Firebase Authentication** e **Firestore**.

A ideia desses dois próximos capítulos é utilizar esses dois serviços do Firebase para construir uma aplicação iOS 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.

9.1 - 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, Google e Apple, tudo isso com uma interface de login própria ou utilizando uma já pronta do Google, que pode ser personalizada.

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.

9.2 - Arquitetura do projeto

A aplicação que começará a ser construída aqui será muito semelhante a que foi finalizada no capítulo anterior, ou seja, uma lista de produtos com a possibilidade de criar, alterar e apagar itens dessa lista. As principais diferenças aqui serão:

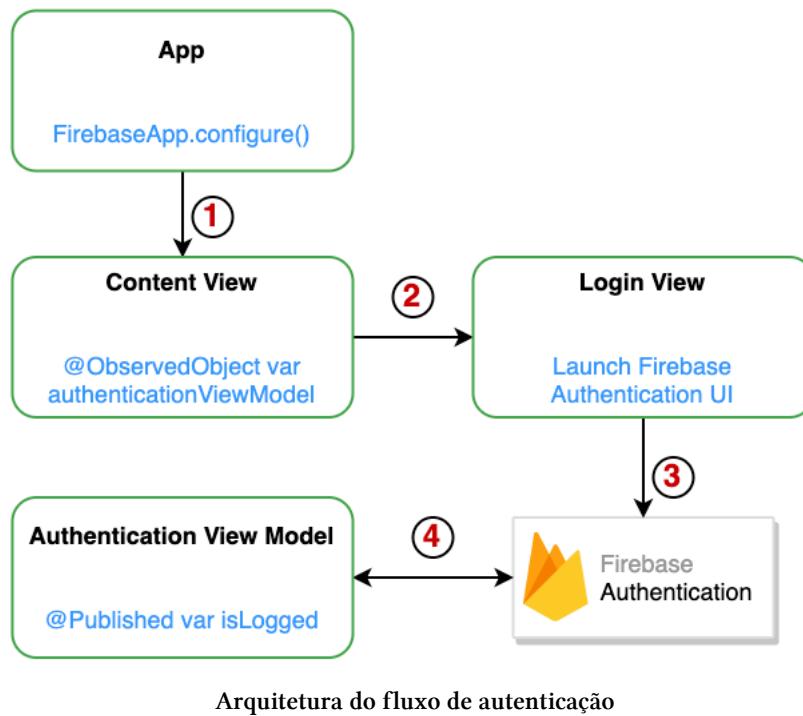
- O usuário deverá efetuar login utilizando o Firebase Authentication com sua conta do Google;
- Os produtos serão armazenados no Firebase Firestore, como será visto no próximo capítulo;
- O usuário poderá ver somente os produtos que ele mesmo criou.

Em se tratando do fluxo de autenticação do usuário, ele terá uma tela simples para selecionar sua conta e efetuar o login. Esse fluxo será iniciado assim que o usuário entrar na aplicação. Ele também terá a opção de efetuar o logout da sua conta, clicando em um ícone no canto superior esquerdo da tela.

Caso o usuário já esteja logado na sua conta dentro do app, ele será redirecionado automaticamente para a tela principal, que será a lista de produtos, semelhante ao que já foi feito no projeto passado.

Para isso, será necessário intervir no fluxo de inicialização do app, para poder verificar se o usuário já está logado ou não, e caso não esteja, redirecioná-lo para a tela de login do FirebaseUI.

Para facilitar esse fluxo, será criado um *viewModel* capaz de cuidar completamente desse processo, como mostra a figura a seguir com a arquitetura proposta para esse projeto, no que se refere ao processo de login, somente:



Quando o app é lançado, a função `init` da `struct` que implementa o protocolo `App`, que é o ponto de entrada da aplicação, é executada. Esse é um bom momento para inicializar o Firebase em todo o projeto, como pode ser visto no **passo 1**. Logo depois a tela principal da aplicação, aqui representada pela `struct ContentView`, é exibida ao usuário. Antes de exibir essa tela de fato ao usuário, ela consulta o `AuthenticationViewModel`, que irá receber os eventos de autenticação e efetivamente saber se um usuário está ou não autenticado, através das API do Firebase Authentication.

Se o usuário não está autenticado, a tela de login implementada por `LoginView` é exibida, como pode ser visto no **passo 2**, dando caminho para o usuário escolher o provedor de sua escolha, que nesse exemplo será somente através da conta do Google. Sendo assim o processo de autenticação começa a acontecer através do Firebase Authentication, como mostra o **passo 3**.

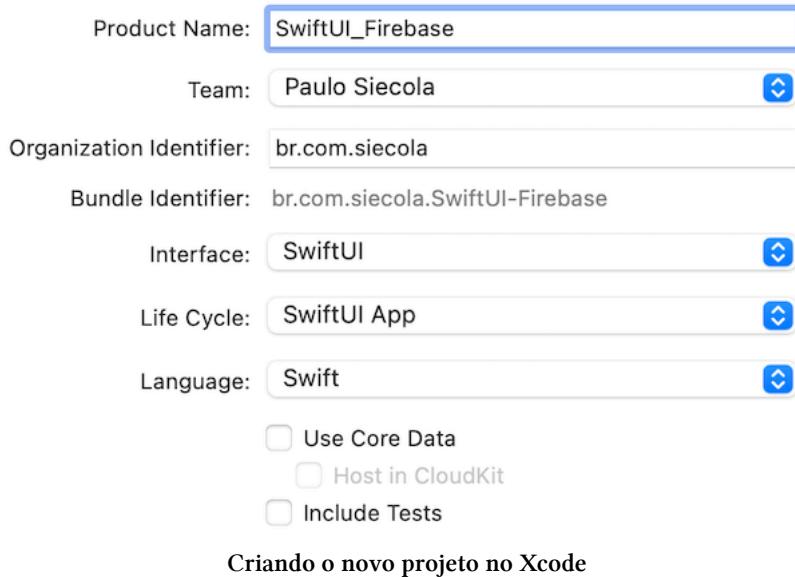
Após a conclusão do processo, caso ele termine com sucesso, o `viewModel AuthenticationViewModel` publica tal status, fazendo com que `ContentView`, que o observa, passe a exibir a tela principal da aplicação, como mostra o **passo 4**.

A classe `AuthenticationViewModel` também cuida do processo de logout do usuário, chamando a função específica para isso no Firebase Authentication.

As próximas seções irão detalhar o processo de criação e configuração do projeto para que ele funcione com o Firebase Authentication, seguindo a arquitetura apresentada aqui.

9.3 - Criando o novo projeto no Xcode

Para começar a criar o projeto, abra o Xcode e selecione o menu `File -> New -> Project` e escolha a opção `iOS -> App`. Na tela seguinte, preencha todos os dados, da mesma forma como foi feito com o projeto passado, escolhendo um nome de seu interesse para esse novo app, como mostra a figura a seguir:



Um ponto importante de se notar aqui é que o valor do campo `Bundle Identifier` será utilizado na configuração do projeto do Firebase.

Depois dessas configurações, salve o projeto onde desejar.

9.4 - 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 projeto do iOS a ele, da mesma forma que foi feito no capítulo anterior. Lembre-se de digitar o conteúdo do campo `Bundle Identifier`, gerado no processo de criação do projeto, como mostra a figura da seção anterior, no campo `iOS bundle ID` da página do Firebase, durante o processo de associação do projeto iOS.

Da mesma forma que foi feito no projeto anterior, baixe o arquivo `GoogleService-Info.plist` para a pasta raiz do projeto. Esse é o arquivo que contém as configurações para que o app ao projeto do Firebase que foi criado aqui.

9.5 - Adicionando as bibliotecas do Firebase Authentication ao projeto

Agora é a hora de adicionar as bibliotecas do Firebase ao projeto. Esse passo já irá incluir todas as bibliotecas que serão utilizadas até o fim desse livro, abrangendo outros serviços do Firebase que serão vistos. Para isso, fecho o projeto do Xcode e execute o comando `pod init` a partir de um terminal do MacOS, de dentro da pasta raiz do projeto.

Em seguida, altere o arquivo `Podfile` com o seguinte conteúdo:

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, '14.0'

target 'SwiftUI_Firebase' do
    use_frameworks!

    pod 'Firebase/Auth'
    pod 'FirebaseUI/Auth'
    pod 'FirebaseUI/Google'
    pod 'Firestore/Firestore'
    pod 'FirebaseFirestoreSwift'
    pod 'Analytics/Analytics'
    pod 'RemoteConfig/RemoteConfig'
end
```

Nesse arquivo já estão adicionadas as bibliotecas para se trabalhar com os seguintes serviços:

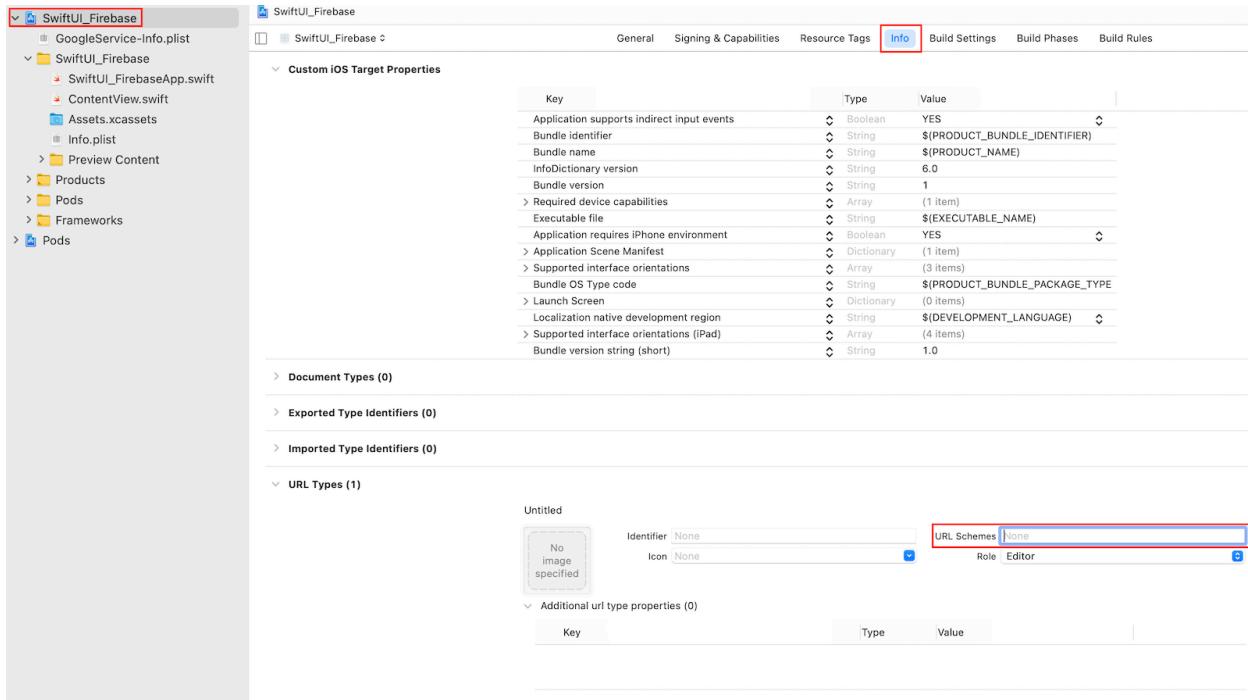
- Firebase Authentication;
- FirebaseUI;
- Firestore;
- Analytics;
- Remote Config.

Os capítulos remanescentes desse livro irão tratar de todos esses serviços.

Após salvar o arquivo, execute o comando `pod install` para que as bibliotecas sejam adicionadas ao projeto. Agora abra o workspace do projeto no Xcode.

O exemplo que será feito nesse capítulo irá autenticar o usuário através de sua conta do Google. Para isso, é necessário realizar uma pequena configuração no projeto, que é a adição do valor do campo `REVERSED_CLIENT_ID` do arquivo `GoogleService-Info.plist` na configuração `URL Types -> URL Schemes` da seção `Info` do projeto do Xcode.

Para realizar a configuração mencionada no parágrafo anterior, abra o arquivo `GoogleService-Info.plist`, que foi adicionado ao projeto, e copie o valor do campo `REVERSED_CLIENT_ID`. Agora clique sobre o nome do projeto no Xcode, para abrir suas configurações. Nessa tela, navegue até a aba Info e adicione o valor copiado do `REVERSED_CLIENT_ID` na configuração URL Schemes dentro da seção URL Types, como pode ser visto na figura a seguir:



Configurando uma nova URL ao projeto

Agora que todas as configurações já foram realizadas no projeto, é hora de partir para o código.

9.6 - Inicializando o Firebase

As bibliotecas do Firebase precisam ser inicializadas assim que o app é executado. Um bom lugar para se fazer isso é dentro da *struct* principal da projeto, a que leva seu nome e que implementa o protocolo App. Até o momento, essa *struct* possui apenas a declaração da variável body, que lança a tela principal da aplicação, como pode ser visto no trecho a seguir:

```
import SwiftUI

@main
struct SwiftUI_FirebaseApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Para que o Firebase possa ser inicializado aqui, primeiramente importe a sua biblioteca base com a instrução `import Firebase`, logo abaixo do importe do `SwiftUI`.

Em seguida, crie a função `init`, que será executada assim que a aplicação for lançada. Dentro dessa função, inicialize o Firebase com o comando `FirebaseApp.configure()`. Veja como deve ficar o código todo desse arquivo:

```
import SwiftUI
import Firebase

@main
struct SwiftUI_FirebaseApp: App {

    init() {
        FirebaseApp.configure()
    }

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Isso fará com que os demais serviços do Firebase que serão demonstrados nesse projeto possam ser utilizados.

9.7 - Criando o `viewModel` de autenticação

Como explicado anteriormente na seção que descreve a arquitetura do fluxo de autenticação desse app, será necessário criar um `viewModel` que irá concentrar esse fluxo, bem como publicar a

informação se o usuário está ou não logado em sua conta do Google. Esse *viewModel* também fará a configuração do Firebase Authentication, para que ele então funcione como provedor desejado, bem como saber onde deverá entregar os eventos de autenticação, através de um *delegate*, que será esse próprio *viewModel*.

Para começar, crie uma nova pasta no projeto, chamada `ViewModels` e dentro dela uma nova classe chamada `AuthenticationViewModel`, como pode ser vista no trecho a seguir:

```
import Foundation
import FirebaseUI

class AuthenticationViewModel: NSObject, ObservableObject, FUIAuthDelegate {
    @Published var isLoggedIn = false

}
```

Veja que aqui está sendo importada a biblioteca do `FirebaseUI`, para que a classe esteja em conformidade com o protocolo `FUIAuthDelegate`. Dessa forma ela poderá receber os eventos de autenticações feitos pelo usuário.

Essa classe também está em conformidade com o protocolo `ObservableObject`, para que sua instância possa ser observada por quem desejar obter eventos sobre o status da autenticação do usuário, que será publicada através da variável `isLoggedIn`, declarada no começo da classe.

A primeira função a ser criada nesse classe é a que será invocada sobre os eventos de autenticação do usuário. Com ela será possível saber quando e se o processo de login foi feito com sucesso. Essa função faz parte do protocolo `FUIAuthDelegate`, que a classe está em conformidade. Veja como ela deve ficar:

```
func authUI(_ authUI: FUIAuth, didSignInWith authDataResult: AuthDataResult?,  
           error: Error?) {  
    print("Signed in as " +  
         "\u{201c}(String(describing: Auth.auth().currentUser?.displayName))\u{201d}")  
    if let _ = error {  
        isLoggedIn = false  
    }  
    else {  
        isLoggedIn = true  
    }  
}
```

Se o parâmetro `error` estiver diferente de nulo, isso significa que o login do usuário não foi feito com sucesso. Caso contrário, ele está autenticado. Perceba que nesse teste a variável `isLoggedIn` está

sendo alterado, fazendo com que seu status seja publicado, pois ela está anotada com @Published, como pode ser visto em sua declaração no início da classe.

Veja que a informação do usuário pode ser obtida através da chamada a Auth.auth().currentUser. Essa informação será utilizada em vários pontos desse projeto para obter a identificação única do usuário, que será importante para o acesso ao Firestore no próximo capítulo. Esse atributo traz informações do usuário, como seu username.

Para que o mecanismo de autenticação seja configurado, é necessário implementar na função init dessa classe as instruções para isso. Basicamente é necessário definir os provedores de autenticação que serão utilizados e o receptor de eventos de autenticação, que nesse caso será a própria classe AuthenticationViewModel. Dessa forma, crie essa função nesse classe, como no trecho a seguir:

```
override init() {
    super.init()

    let authUI = FUIAuth.defaultAuthUI()!

    let providers: [FUIAuthProvider] = [
        FUIGoogleAuth(),
    ]

    authUI.providers = providers
    authUI.delegate = self

    if Auth.auth().currentUser != nil {
        isLoggedIn = true
    } else {
        isLoggedIn = false
    }
}
```

Veja que a segunda instrução da função init busca a configuração da interface padrão de autenticação do Firebase Authentication. Logo em seguida é feita a configuração da lista de provedores a serem utilizados para que o usuário possa se autenticar. Como dito anteriormente, é possível se autenticar com vários provedores, como Facebook, Google e Apple. Dessa forma o usuário pode escolher qual conta utilizar. Nesse exemplo será utilizado somente o provedor do Google para autenticar o usuário.

Logo em seguida é definido que o delegate que irá receber os eventos de autenticação é a própria classe AuthenticationViewModel, pois ela está em conformidade com o protocolo FUIAuthDelegate.

No fim da função init, é feita a validação para verificar se o usuário já não estava logado no app, caso ele já tenha feito isso em outra interação com ele. Isso significa que com o Firebase Authentication não é necessário que o usuário se autentique toda vez que for abrir o app, pois o Firebase guarda a sessão da última autenticação feita pelo usuário.

Veja que a variável `isLoggedIn` está sendo escrita nessa verificação, o que faz com que seu valor seja publicado para que estiver observando-a, como será visto em algumas seções a seguir.

A última função a ser criada aqui é logout do usuário, como pode ser visto no trecho a seguir:

```
func signOut() {  
    try? Auth.auth().signOut()  
    isLoggedIn = false  
}
```

Aqui é necessário chamar a função `signOut` de `Auth.auth()`. Caso tudo dê certo nesse processo, a variável `isLoggedIn` é alterada para `false` para indicar que o usuário não está mais logado, para os que estiverem observando-a.

9.8 - Configurando o Firebase Authentication

Como explicado na seção anterior, é possível definir os provedores de autenticação para que o usuário possa escolher em qual deseja se autenticação no app. Mas para isso é necessário habilitá-los no console do Firebase, na seção Build -> Authentication.

Nessa tela, escolha a aba `Sign-in method` e habilite a opção `Google`, que é o provedor de autenticação escolhido para essa demonstração. Dessa forma o usuário poderá se autenticar com sua conta do Google.

9.9 - Criando a tela de login

Como dito anteriormente, a tela de login a ser utilizada será a padrão oferecida pelo FirebaseUI. Ela possui basicamente os botões com os provedores de autenticação escolhidos na configuração feita na classe `AuthenticationViewModel`.

Para exibir essa tela em uma aplicação feita com SwiftUI, é necessário criar uma `view` que esteja em conformidade com o protocolo `UIViewControllerRepresentable`, como será visto mais adiante. Dessa forma a tela de login poderá ser exibida ao usuário.

A ideia é que essa tela seja exibida ao usuário quando a aplicação perceber que ele não está autenticado na aplicação, sobrepondo-a à tela principal da aplicação.

Para começar, crie um novo grupo no projeto do Xcode, de nome `Views` e dentro dele um novo arquivo chamado `LoginView`, como no trecho a seguir:

```
import SwiftUI
import FirebaseUI

struct LoginView: UIViewControllerRepresentable {
```

}

Como dito, essa *struct* deve estar em conformidade com o protocolo `UIViewControllerRepresentable`. Isso faz com que ela tenha que implementar duas funções que serão chamadas para construção e atualização da tela, como pode ser visto no trecho a seguir:

```
func makeUIViewController(context: Context) -> UIViewController {
    return FUIAuth.defaultAuthUI()!.authViewController()
}

func updateUIViewController(_ uiViewController: UIViewController,
                           context: Context) {

}
```

Na verdade, a única função útil para o processo de autenticação é a construção da *view* de fato, que é feito na função `makeUIViewController`. Nessa função então é criado um *viewController* através da chamada `FUIAuth.defaultAuthUI()!.authViewController()`. Isso faz com que a tela de autenticação padrão do `FirebaseUI` seja exibida ao usuário para que ele possa iniciar o fluxo de autenticação.

Essa *view* será chamada no fluxo de autenticação, que será explicado na próxima seção.

9.10 - Criando o fluxo de autenticação do app

O fluxo de autenticação do usuário dentro do app será gerenciado na tela principal, que está implementada no arquivo `ContentView.swift`. Dentro de sua *struct*, será feita a validação para ver se já existe um usuário autenticado, antes da exibição dessa tela principal. Caso não exista nenhum usuário autenticado, então a *view* criada na seção anterior será exibida.

Para isso, abra o arquivo `ContentView.swift` e crie uma variável do tipo `AuthenticationViewModel` para observar o status de login do usuário, como pode ser visto no trecho a seguir:

```
struct ContentView: View {
@ObservedObject private var authenticationViewModel = AuthenticationViewModel()
```

Quando o status da variável `isLogged` de `AuthenticationViewModel` sofrer alguma alteração, essa *view* poderá ser atualizada a contento.

O próximo passo é criar uma função dentro da *struct ContentView* que será chamada quando o usuário quiser fazer o logout da aplicação, como mostra o trecho a seguir:

```
func signOut() {  
    authenticationViewModel.signOut()  
}
```

Veja que aqui está sendo chamada a função `signOut` de `AuthenticationViewModel`, para fazer todo esse trabalho, além de atualizar a variável `isLoggedIn` com seu novo status.

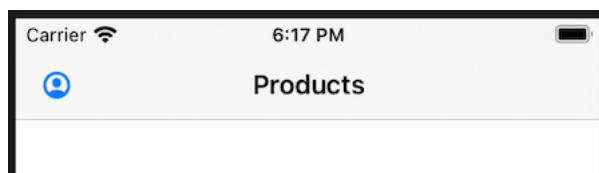
Como dito anteriormente, a ideia desse app é mostrar uma lista de produtos, que serão persistidos no Firebase Firestore, como será visto no próximo capítulo. Dessa forma, será necessário implementar um mecanismo de navegação, da mesma forma como foi feito no projeto do capítulo 7, para que o usuário possa clicar em um produto da lista e ver seus detalhes.

A criação desse mecanismo de navegação abre a possibilidade da configuração de botões de ações na barra superior do app, que nesse momento poderá ser aproveitada para a inclusão do botão de logout a ser acessado pelo usuário quando ele quiser sair da aplicação.

Para isso, vá no arquivo `ContentView.swift` e adicione o componente `NavigationView`, envolvendo a caixa de texto que lá já existe, como no trecho a seguir:

```
NavigationView {  
    Text("Hello, world!")  
    .padding()  
  
    .navigationBarTitle("Products", displayMode: .inline)  
    .navigationBarItems(leading: Button(action: signOut) {  
        Image(systemName: "person.crop.circle")  
        .foregroundColor(Color.blue)  
    })  
}
```

Veja que aqui já está sendo definido o título da barra de navegação e também está sendo criado um botão no canto superior esquerdo para que o usuário possa clicar e fazer o logout da aplicação. A ação desse botão irá chamar a função `signOut` criada anteriormente. Veja como a tela deve ficar, até o momento:



Bara superior do app

O botão localizado no canto superior esquerdo será utilizado para fazer o logout da aplicação.

Porém, veja que ao executar a aplicação, a tela inicial ainda é exibida, ao invés da que inicia o fluxo de autenticação. Isso porque é necessário verificar se o usuário está de fato logado no app, e caso não esteja, exibir a tela criada em `LoginView`. Para isso, utilize o `AuthenticationViewModel` para saber se existe um usuário logado. Esse teste deve envolver todo o componente `NavigationView`, como no trecho a seguir:

```
var body: some View {
    if authenticationViewModel.isLoggedIn {
        NavigationView {
            Text("Hello, world!")
                .padding()

                .navigationBarTitle("Products", displayMode: .inline)
                .navigationBarItems(leading: Button(action: signOut) {
                    Image(systemName: "person.crop.circle")
                        .foregroundColor(Color.blue)
                })
        }
    } else {
        LoginView()
    }
}
```

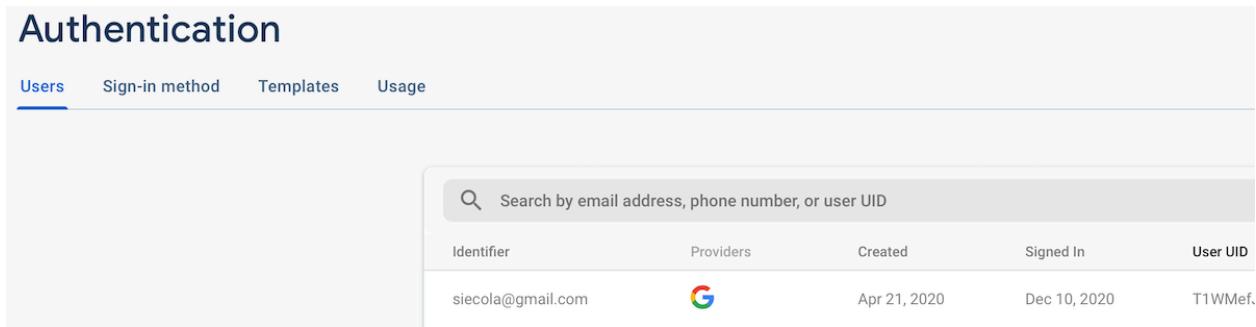
Veja que a tela principal só é exibida se o valor da variável `isLoggedIn` for `true`. Caso contrário a tela implementada em `LoginView` é exibida, dando início ao fluxo de autenticação do usuário através de sua conta do Google.

Quando o fluxo for concluído, o valor da variável `isLoggedIn` de `AuthenticationViewModel` será publicado, fazendo com que a tela implementada em `ContentView` seja redesenhada, mas agora com o usuário já autenticado, permitindo que ele veja a tela principal.

Caso o usuário queira fazer o logout da aplicação, basta clicar no botão no canto superior esquerdo da barra superior, fazendo com que a tela de login apareça novamente, escondendo a tela principal da aplicação.

Faça o teste executando a aplicação e perceba que depois do login efetuado, o usuário permanece logado, mesmo que a aplicação seja fechada e aberta novamente.

No console do Firebase, dentro da seção `Authentication`, é possível observar os usuários que estão autenticados, como mostra a figura a seguir:



The screenshot shows the Firebase Authentication console under the 'Users' tab. A search bar at the top allows searching by email address, phone number, or user UID. Below the search bar is a table with columns: Identifier, Providers, Created, Signed In, and User UID. One user entry is listed: siecola@gmail.com, Google (G), Apr 21, 2020, Dec 10, 2020, T1WMef.

Identifier	Providers	Created	Signed In	User UID
siecola@gmail.com		Apr 21, 2020	Dec 10, 2020	T1WMef.

Usuários autenticados no Firebase

9.11 - Conclusão

Esse capítulo mostrou como pode ser fácil implementar o processo de autenticação do usuário utilizando o Firebase Authentication, para que ele possa se autenticar utilizando sua conta do Google.

Ter o usuário autenticado na aplicação pode trazer vários benefícios, como será explicado no próximo capítulo, onde será possível criar regras de segurança e acesso aos produtos criados, de acordo com o usuário autenticado na aplicação.

10 - Persistindo dados com o 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.

The screenshot shows the Google Cloud Platform Firestore console. At the top, there's a navigation bar with a home icon, followed by 'products' and a document ID '0otLwJumWEus...'. Below the navigation is a header row with three columns: 'products' (with a 'Start collection' button), 'products' (with an 'Add document' button), and '0otLwJumWEus0cVp0pN6' (with a 'Start collection' button). The main area displays a collection named 'products' with one document listed. The document has the ID '0otLwJumWEus0cVp0pN6' and contains the following fields:

code	: "COD2X"
description	: "Desc2"
name	: "Product2"
price	: 15.5
userId	: "T1WMefJ1S6PqI"

Console do Firestore

Nesse console o desenvolvedor pode criar coleções, documentos ou alterar dados já existentes.

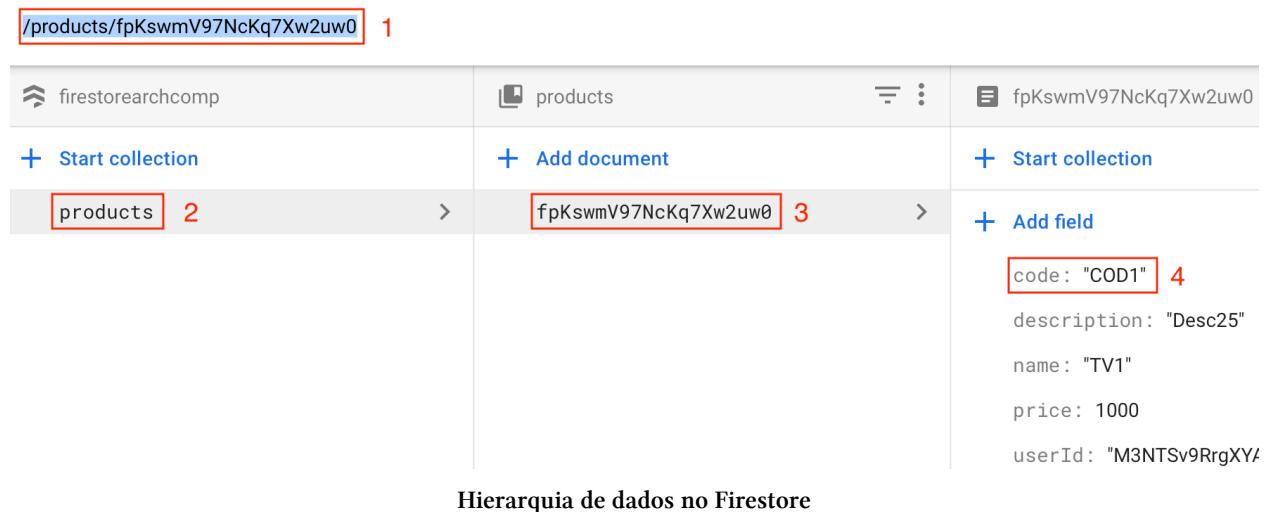
A seção seguinte detalha a hierarquia de dados no Firestore.

10.1 - 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.

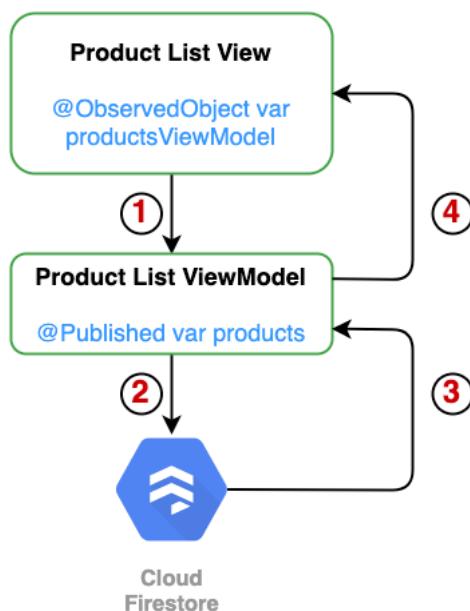
Na figura anterior observe que o documento possui um campo chamado `userId`. Ele guarda a identificação do usuário dono do produto, ou seja, quem foi que o criou. Essa informação será utilizada para definir as regras de acesso, nas próximas seções.

10.2 - Arquitetura do projeto

A arquitetura desse novo projeto será parecida com o que foi construído nos capítulos 6 e 7. Isso significa que muito código que foi construído lá poderá ser aproveitado nesse capítulo.

A estratégia será a mesma: utilizar `viewModels` para cada `view`.

A tela que irá listar todos os produtos irá observar seu `viewModel` para obter publicações de alterações na lista de produtos. Isso significa que a lista de produtos poderá ser alterada mesmo sem a interação do usuário do app naquele dispositivo, caso ele faça alguma alteração em outro dispositivo que esteja logado com a mesma conta, algo que irá demonstrar a funcionalidade *realtime* do Firestore. Veja como deve ficar a arquitetura dessa parte do projeto:

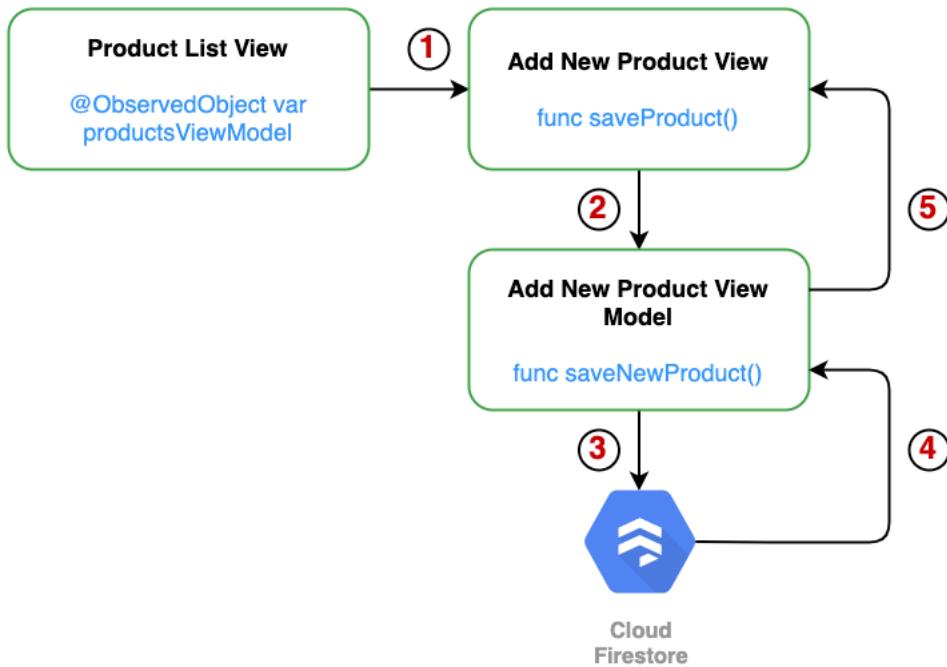


Arquitetura da tela com a lista de produtos

Quando a `view` solicitar a lista e produtos para o `viewModel`, no **passo 1**, esse irá acessar o Firestore diretamente, solicitando a consulta desejada, que deverá filtrar todos os produtos do usuário autenticado, ordenando-os por nome, como mostra o **passo 2**. Como a consulta ao Firestore é assíncrona, ele irá notificar o `viewModel` quando os dados estiverem prontos para serem lidos, como pode ser

visto no **passo 3**. O *viewModel* por sua vez irá publicar a lista de produtos que está sendo observada pela *view*, como mostra o **passo 4**.

O fluxo de criação de um novo produto também segue um princípio semelhante, como mostra a figura a seguir:



Arquitetura do fluxo de criação de um novo produto

Veja que o diagrama é muito parecido com o que foi mostrado no capítulo 7.

Quando o usuário clicar no botão da tela principal para adicionar um novo produto, ele será redirecionado para a *view* de criação da nova entidade, como mostra o **passo 1**. Nela o usuário poderá digitar os dados do produto e clicar no botão de salvar, que irá invocar a função específica de seu *viewModel*, como pode ser visto no **passo 2**, que por sua vez acessa o Firestore para salvá-lo de fato, como mostra o **passo 3**. Como esse processo é síncrono, a resposta pode ser informada ao *viewModel* se o produto foi salvo ou não, como pode ser visto no **passo 4**. Esse por sua vez retorna para a *view* no **passo 5**, que pode navegar de volta para a tela principal, caso o processo tenha sido feito com sucesso, ou caso contrário, exibir uma mensagem de erro ao usuário.

Veja que nesse caso a tela principal com a lista de produtos não está sendo avisada que um novo produto foi incluído, para que ela possa atualizar essa lista. Isso já será feito automaticamente pelo processo que observa a lista de produtos filtrados pelo `userId`. Isso significa que ao navegar de volta para a tela principal, o produto recém criado já aparecerá na tela, ordenado pelo seu nome.

10.3 - Criando o modelo de produtos

Para representar os dados exibidos no console do Firestore da figura da seção passada, é necessário definir um modelo contendo atributos que representem cada campo. Esse modelo então será utilizado tanto para persistir e ler os produtos do Firestore, quanto para exibi-los na tela ao usuário. Da mesma forma, quando um usuário digitar os dados para a criação de um novo produto, uma instância desse modelo será criada para ser salva no Firestore.

Para começar, crie uma nova pasta no projeto do Xcode chamada `Models` e dentro dela um novo arquivo com o nome de `Product`, como no trecho a seguir:

```
import Foundation
import FirebaseFirestoreSwift

struct Product: Codable {

    static let COLLECTION = "products"
    static let USER_ID = "userId"
    static let NAME = "name"

    @DocumentID var id: String? = UUID().uuidString
    var userId: String
    var name: String
    var description: String
    var code: String
    var price: Double
}
```

Primeiramente, veja que a `struct Product` está em conformidade com o protocolo `Codable`. Isso será útil para converter uma instância dela em um dado a ser salvo no Firestore, bem como fazer o contrário. Isso será possível com a utilização do `FirebaseFirestoreSwift`, como será visto mais adiante.

Logo no início da `struct` há a declaração das constantes `COLLECTION`, `USER_ID` e `NAME`, que serão utilizadas para definir o nome da coleção de produtos no Firestore e seus campos `userId` e `name`, respectivamente. Essas constantes serão utilizadas mais adiante nas funções de consulta de produtos.

O primeiro atributo a ser definido no modelo de produtos é o seu identificador único. Como pode ser observado na figura da seção anterior, ele não é necessariamente um campo do documento `product`, mas sim a referência para chegar até um produto específico. Declarar esse campo no modelo com a anotação `DocumentID` faz com que o campo exista somente no modelo, para uso interno do app e não apareça como campo no documento do Firestore.

Os demais atributos representam os campos do documento de produto do Firestore, com especial atenção ao campo `userId`, que conterá a identificação única do usuário logado no Firebase

Authentication. Esse campo será utilizado para compor as regras de segurança de acesso aos dados armazenados no Firestore, como será visto na seção seguinte.

10.4 - Especificando as regras de segurança do Firestore

A coleção de produtos no Firestore irá armazenar todos os produtos de todos os usuários do app. Porém, a ideia é que apenas os produtos criados por cada usuário possam ser acessados por eles. Isso significa que deve haver uma regra de segurança que garanta o acesso aos produtos somente aos usuários que os criaram, evitando que um usuário possa acessar um documento que não o pertença.

Outra diretiva de segurança é que somente usuários que tenha efetuado o login no app utilizando o Firebase Authentication possam acessar a tabela para fazer qualquer operação.

Essas duas regras de segurança podem ser facilmente criadas no console do Firestore. Porém, como dito na seção anterior, a regra que garante o acesso do usuário somente aos dados que ele criou, depende do campo `userId` criado no modelo de produtos na seção anterior.

Para começar a especificar essas regras, vá até o console do Firebase e acesse a opção Cloud Firestore. Dentro da página inicial do Firestore, clique no botão Create database para começar o processo de provisionamento do novo banco de dados. O popup que aparecer irá questionar sobre como as regras de segurança deverão ser criadas para esse banco de dados. Selecione a opção Start in production mode para que as regras de segurança tenham que ser definidas manualmente para permitir qualquer acesso ao banco. Para finalizar, clique em Next, e em seguida escolha a região onde desejar criar o banco e clique em Enable.

A tela que aparecer é o console do Firestore, que por enquanto não possui nenhuma coleção, mas não é necessário criar uma, pois o primeiro acesso do app irá criar a coleção de produtos.

Para de fato criar as regras de segurança, acesse a aba Rules no topo da página. Essa página permite configurar, testar e monitorar as regras de segurança e para criar as regras necessárias, cole o seguinte texto no editor de regras dessa página:

```
rules_version = '2';
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.

10.5 - Construindo o índice de pesquisa no Firestore

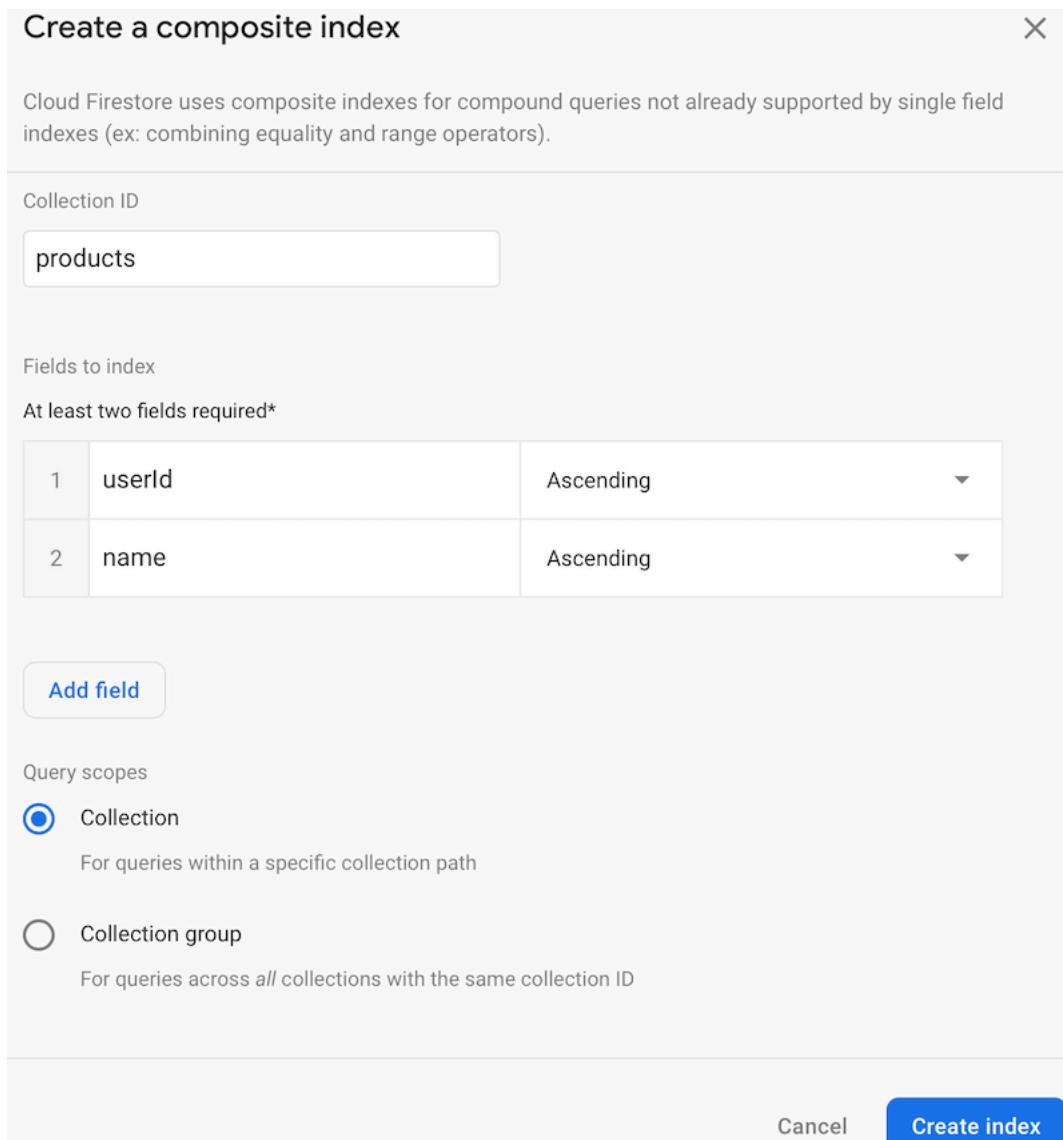
Como é comum banco de dados não relacionais, é necessário criar índices para pesquisas na coleção de documentos. No exemplo que será construído será necessário criar um índice para a pesquisa dos produtos ordenados por nome e que pertençam a um `userId`. Para a criação desse índice, existem duas formas de se proceder:

- Executar o Xcode e procurar nos logs por uma mensagem que diz que o índice deve ser criado, que já contém o link para a criação automática dele;
- Utilizar o console do Firestore para a criação manual do índice.

As duas opções são válidas, principalmente a primeira, onde o link para o console do Firestore já cria o índice automaticamente, baseado na consulta realizada pelo app, porém pode ser um pouco difícil de localizar a mensagem de log específica no Xcode. Mas essa estratégia pode ser útil quando não se tem certeza de como o índice deve ser criado de fato, em uma consulta mais complexa. Veja um exemplo de mensagem de log no trecho a seguir:

Error fetching documents: Error Domain=FIRFirestoreErrorDomain Code=9 "The query requires an index. You can create it here: https://console.firebaseio.google.com/v1/r/project/iosbook/firestore/indexes?create_composite=C1Nwcm9qZWN0cy9hbmRyb2lkYm9vazItOWUz\N2Q

Para criar o índice que será utilizado nesse exemplo utilizando o console do Firestore, acesse a aba Indexes e clique no link Create index no centro da página. No popup que aparecer, preencha com as seguintes informações, conforme a figura a seguir:



Criando um índice de pesquisa

- **Collection ID:** aqui deve ser informado o nome da coleção em que o índice se refere;

- **Fields to index:** preencha com os nomes dos campos que farão parte da pesquisa. Nesse caso devem ser colocados os campos `name` e `userId`.
- **Query scopes:** deixe a opção `Collection`, pois a estrutura de dados desse exemplo não possui coleções aninhadas.

Para concluir a criação do índice, clique no botão `Create index`, que pode demorar alguns minutos até a sua finalização.

Depois que o índice for criado, será possível continuar a implementação do código do projeto do app, como será detalhado nas próximas seções.

10.6 - Criando o `viewModel` da lista de produtos

Para começar com a criação do `viewModel` da lista de produtos, vá até a pasta `ViewModels` e crie um novo arquivo de nome `ProductListViewModel`, como no trecho a seguir:

```
import Foundation
import Firebase
import FirebaseFirestoreSwift

class ProductListViewModel: ObservableObject {
    @Published var products = [Product]()

    private let db = Firestore.firestore()

}
```

Veja que ele também está em conformidade com o protocolo `ObservableObject`, para que sua instância possa ser observada pela `view` com a lista de produtos. Da mesma forma, ele possui o atributo `products`, que irá publicar eventos de alteração na lista de produtos.

Repare também que há um atributo privado com a instância do `Firestore`. Com ela será possível realizar a pesquisa dos produtos diretamente nele.

A primeira função a ser criada na classe `ProductListViewModel` é a de busca dos produtos, como pode ser vista no trecho a seguir:

```

func fetchProducts() {
    db.collection(Product.COLLECTION).whereField(Product.USER_ID,
        isEqualTo: Auth.auth().currentUser!.uid).order(by: Product.NAME)
    .addSnapshotListener { querySnapshot, error in
        guard let documents = querySnapshot?.documents else {
            print("Error fetching documents: \(error!)")
            return
        }

        self.products = documents.compactMap {
            queryDocumentSnapshot -> Product? in
            return try? queryDocumentSnapshot.data(as: Product.self)
        }
    }
}

```

A primeira parte da instrução dessa função cuida de acessar a coleção de produtos, filtrando-os pelo campo `userId` sendo igual ao do usuário autenticado e ordenando-os pelo seu nome:

```

db.collection(Product.COLLECTION).whereField(Product.USER_ID,
    isEqualTo: Auth.auth().currentUser!.uid).order(by: Product.NAME)

```

Veja que isso pode ser feito utilizando a instância do Firestore, aqui representado pelo atributo privado `db`. Com ele basta a acessar a coleção com a sua função `collection` passando seu nome. A filtragem fica por conta da função `whereField`, que recebe o nome do campo e seu valor, que nesse caso recebe a identificação única do usuário autenticado através do objeto `currentUser` de `Auth`. Por último a função `order` é chamada para ordenar por um campo específico, que nesse caso é o nome do produto.

Como dito anteriormente, essa chamada é assíncrona, fazendo com que o resultado tenha que ser tratado posteriormente. Porém aqui o desejo é que essa mesma chamada também fique observando por mudanças no resultado da pesquisa, para poder atualizar a tela com a lista de produtos, caso ela tenha alguma alteração, por um outro app em outro dispositivo logado com a mesma conta do usuário. Isso pode ser obtido com a chamada a `addSnapshotListener` que recebe um *closure* com os parâmetros `querySnapshot`, com o resultado da pesquisa, e um outro atributo para indicar se houve algum erro.

Dentro desse *closure* pode ser verificado se a pesquisa retornou algum documento, como pode ser observado na instrução `guard` que verifica se existem documentos resultantes da pesquisa.

Caso haja algum produto no resultado da pesquisa, eles serão transformados em uma lista `products`, como pode ser visto nas instruções finais da função:

```
self.products = documents.compactMap {
    queryDocumentSnapshot -> Product? in
    return try? queryDocumentSnapshot.data(as: Product.self)
}
```

Veja que aqui cada resultado da pesquisa é transformado diretamente para uma instância de `Product`, o modelo que foi criado algumas sessões atrás para representar o produto no Firestore. Isso foi possível graças à biblioteca `FirebaseFirestoreSwift`.

O *closure* presente dentro da função `addSnapshotListener` será chamado sempre que houver alguma mudança no resultado da pesquisa, seja pela adição, remoção ou alteração de algum elemento. Logo, a escrita no atributo `products` do `viewModel` fará que um evento seja publicado para quem estiver observando-o posso atualizar a tela, que é o que será feito na próxima seção.

Uma outra importante função a ser criada nesse `viewModel` é para limpar a lista de produtos. Isso será necessário quando o usuário autenticado for trocado por outro. Logo essa função deverá ser chamada durante o processo de logout. Veja como deve ficar no trecho a seguir:

```
func clearProducts() {
    products = [Product]()
}
```

Dessa forma, acrescente a chamada a essa função dentro da `struct ContentView`, na função `signOut`, como no trecho a seguir:

```
func signOut() {
    productListViewModel.clearProducts()
    authenticationViewModel.signOut()
}
```

Como essa função será chamada quando o usuário pressionar o botão de logout, seus produtos serão limpos da tela logo quando o processo de logout for concluído, evitando que o próximo usuário veja seus produtos.

10.7 - Criando a view da lista de produtos

A lista de produtos será exibida na `struct` já existente de nome `ContentView`. Primeiramente é necessário criar a instância de `ProductListViewModel`, para que ela tenha acesso à função criada na seção anterior e possa observar as mudanças de sua lista de produtos. Para isso, crie o seguinte atributo nessa `struct`:

```
@ObservedObject private var productListViewModel = ProductListViewModel()
```

Agora substitua o componente Text pela implementação da lista de produtos, utilizando o productListViewModel, como no trecho a seguir:

```
List {
    ForEach(self.productListViewModel.products, id: \.id) { product in
        HStack {
            Text(product.name)
            Spacer()
            Text(product.code)
            Spacer()
            Text(String(format: "$ %.2f", product.price))
        }
    }
}
```

Repare que esse código é muito semelhante ao que foi utilizado no capítulo 6 para a construção da lista de produtos.

Quando o atributo products de productListViewModel publicar alguma alteração, essa lista será redesenhada para mostrar tal alteração, fazendo com que a tela sempre fique em sincronia com o resultado da pesquisa realizada pela função fetchProducts, que deve ser invocada nessa view, na função onAppear do componente NavigationView, como no trecho a seguir:

```
NavigationView {
    List {
        // outros componentes
    }
}.onAppear() {
    productListViewModel.fetchProducts()
}
```

A função onAppear de NavigationView será invocada somente se o usuário estiver se autenticado no Firebase Authentication, pois esse componente só aparece nesse caso, que está sendo controlado pelo teste realizado no início da declaração da variável body:

```
var body: some View {
    if authenticationViewModel.isLoggedIn {
        NavigationView {
```

Essa implementação já poderia ser testada, se houvesse algum produto cadastrado no Firestore. Isso será feito nas próximas duas seções, com a tela de criação dessa entidade, logo toda a implementação será testada a seguir.

10.8 - Criando o viewModel de criação de produtos

Para iniciar o fluxo de criação de um novo produto, é necessário começar pelo *viewModel* dessa tela. Ele será muito semelhante ao que foi feito no capítulo 7. Para isso, crie a classe `AddNewProductViewModel` na pasta `ViewModels` do projeto, como no trecho a seguir:

```
import Foundation
import Firebase
import FirebaseFirestoreSwift

class AddNewProductViewModel {
    var name: String = ""
    var description: String = ""
    var code: String = ""
    var price: Double = 0.0

    private let db = Firestore.firestore()

}
```

Veja que os atributos dessa classe representam os campos que o usuário irá preencher na tela de criação de produto.

A função para salvar o produto irá então construir uma instância de `Product` e deverá adicionar um novo documento na coleção `products` do Firestore, que indicará se ele foi salvo ou não, devolvendo seu novo Id, como pode ser visto na implementação na função a seguir, que deve ser adicionada na nova classe:

```
func saveNewProduct() -> Bool {
    let product = Product(userID: Auth.auth().currentUser!.uid,
                          name: self.name,
                          description: self.description,
                          code: self.code,
                          price: self.price)

    let newProductRef = db.collection(Product.COLLECTION).document()

    do {
        try newProductRef.setData(from: product)
        print("The new product has been saved - ID: \(newProductRef.documentID)")
        return true
    } catch {
        print("Error saving product: \(error.localizedDescription)")
        return false
    }
}
```

```
    } catch let error {
        print("Error while saving new product: \(error)")
        return false
    }
}
```

Veja que na criação da instância de `Product`, logo na primeira linha da função, o campo `userId` está sendo preenchido com a identificação única do usuário que está logado no app. Isso fará com que esse produto só possa ser acessado por ele novamente, protegendo os dados que ele criar dentro da coleção `products` no Firestore.

A instrução seguinte solicita que um novo documento seja criado na coleção `products` do Firestore, que é preenchido com a função `setData` dentro do bloco `try`. Veja que a instância de `product` é convertida para um documento do Firestore, graças a biblioteca `FirebaseFirestoreSwift` e pelo modelo `Product` estar em conformidade com o protocolo `Codable`.

Caso o produto seja criado com sucesso, o valor `true` é retornado pela função, que será utilizado pela `view` que será construída na próxima seção.

10.9 - Construindo a view de criação de produtos

A ideia da tela de criação de um novo produto é muito semelhante à que foi implementada no capítulo 7. Ela deverá conter duas variáveis de estado:

- `addNewProductViewModel`, que a instância do `viewModel` criado na seção anterior;
- `showAlert`, para exibir um alerta ao usuário, caso a criação do produto falhe.

Essa `view` também deverá ter uma variável para controlar sua exibição, da mesma forma como foi feito no capítulo 7. Variável essa que é uma cópia da variável de estado que deverá ser criada na `view` de lista de produtos.

Para começar, crie o arquivo `AddNewProductView`, seguindo o *template* do SwiftUI, na pasta `Views` do projeto, como no trecho a seguir:

```
import SwiftUI

struct AddNewProductView: View {
    @Binding var isPresented: Bool
    @State private var addNewProductViewModel = AddNewProductViewModel()
    @State private var showAlert = false

    var body: some View {
        Text("Hello, World!")
    }
}

struct AddNewProductView_Previews: PreviewProvider {
    static var previews: some View {
        AddNewProductView(isPresented: .constant(false))
    }
}
```

Perceba que a `struct AddNewProductView_Previews` deve conter o atributo `isPresented` em sua declaração, pois ele será passado pela tela de listagem de produtos.

Agora crie a função privada para ser invocada quando for necessário voltar para a tela principal do app, como no trecho a seguir:

```
private func dismiss() {
    self.isPresented = false
}
```

Como já feito anteriormente, isso pode ser feito facilmente apenas alterando a variável de estado `isPresented` para `false`.

A próxima função privada a ser construída é aquela para chamar o `viewModel` para salvar o produto, como pode ser visto no trecho a seguir:

```
private func saveProduct() {
    self.showAlert = false
    if self.addNewProductViewModel.saveNewProduct() {
        self.isPresented = false
    } else {
        self.showAlert = true
    }
}
```

Veja que ela chama a função `saveNewProduct` do `viewModel`, que retorna `true`, caso o processo tenha sido concluído com sucesso. Isso então pode ser utilizado para controlar o alerta a ser exibido ao usuário, através da variável de estado `showAlert`.

Agora crie o `NavigationView` para a exibição dos controles dessa nova tela, como mostra o trecho a seguir:

```
var body: some View {
    NavigationView {
        Text("Hello, World!")

        .navigationBarTitle("New product", displayMode: .inline)
        .navigationBarItems(leading: Button(action: dismiss) {
            Text("Cancel").foregroundColor(.red)
        }, trailing: Button(action: saveProduct) {
            Text("Add").foregroundColor(.blue)
        })
        .alert(isPresented: $showAlert) {
            Alert(title: Text("Error"),
                  message: Text("The product couldn't be saved"),
                  dismissButton: .default(Text("Dismiss")))
        }
    }
}
```

Veja que o título da tela e seus controles na barra superior já foram criados, com os botões de cancelar e salvar o produto, que chamam as duas funções criadas anteriormente. Também foi criado o alerta que será exibido caso o produto não seja salvo no Firestore.

Por fim, substitua o componente `Text` pela criação do formulário para o usuário digitar os dados do produto, como no trecho a seguir:

```
Form {
    Section() {
        TextField("Enter the name",
                  text: self.$addNewProductViewModel.name)

        TextField("Enter the description",
                  text: self.$addNewProductViewModel.description)

        TextField("Enter the code",
                  text: self.$addNewProductViewModel.code)

        PriceField(value: self.$addNewProductViewModel.price)
```

```
    }  
}
```

Note que o componente `PriceField` não existe nesse projeto, mas ele é o mesmo que foi criado no capítulo 7 e por isso pode ser copiado de lá.

E isso conclui a implementação dessa tela, que será chamada na seção seguinte.

10.10 - Exibindo a tela de criação de produtos

Para exibir a tela de criação de produtos criada na seção anterior, primeiramente é necessário definir a variável de estado que vai controlar sua exibição, por isso volte na `struct ContentView` e crie com no trecho a seguir:

```
@State private var showModal = false
```

Da mesma forma, é necessário criar a função que será chamada quando o usuário clicar no botão para adicionar um novo produto, que deve alterar o valor dessa variável de estado, como mostra o trecho a seguir:

```
private func showNewProductView() {  
    self.showModal = true  
}
```

Para que essa função seja chamada, é necessário exibir o botão para adição do novo produto. Isso pode ser feito na construção dos itens da barra de navegação em `navigationBarItems`, como no trecho a seguir:

```
.navigationBarItems(leading: Button(action: signOut) {  
    Image(systemName: "person.crop.circle")  
        .foregroundColor(Color.blue)  
, trailing: Button(action: showNewProductView) {  
    Image(systemName: "plus")  
        .foregroundColor(Color.blue)  
})
```

Por fim, para que a tela de criação de produtos seja de fato exibida, crie-a dentro da função `sheet` do `NavigationView`, como no trecho a seguir:

```
.sheet(isPresented: $showModal) {  
    AddNewProductView(isPresented: self.$showModal)  
}
```

Veja que ela é controlada pela variável de estado `showModal`.



A próxima seção irá detalhar os passos para testar a implementação construída até o momento.

10.11 - Testando a criação e listagem de produtos

Para testar a implementação que exibe a lista de produtos, execute a aplicação e clique no botão + localizado no canto superior da barra de navegação. Isso fará com que a tela de criação de um novo produto apareça. Nessa tela, preencha as informações do produto a ser criado e pressione o botão Salvar no canto superior direito.

Após o produto ser salvo, a janela é fechada e é exibida a tela principal da aplicação, que já deverá conter o produto criado. Esse produto também já deverá aparecer no console do Firestore, com todos seus campos preenchidos, inclusive a identificação única do usuário logado que criou tal produto.

Experimente executar a aplicação, mas agora com uma outra conta do Google. O produto criado com a primeira conta não deverá aparecer, pois não pertence a esse usuário. Repita o passo de criação de um novo produto para essa segunda conta e observe no console do Firestore que agora existem dois produtos, mas com os valores do campo `userId` diferentes, confirmando que cada produto criado por usuário só pode ser acessado por ele.

10.12 - Excluindo produtos

Para excluir um produto da lista é necessário criar uma nova função em `ProductListViewModel`, que basicamente deve receber o produto a ser excluído e um *closure* para informar sobre o resultado da operação, como pode ser visto na função a seguir:

```
func deleteProduct(product: Product, completion: @escaping (Bool) -> Void) {
    db.collection(Product.COLLECTION).document(product.id!).delete() { error in
        if let error = error {
            print("Error while removing the product: \(error)")
            completion(false)
        } else {
            print("Product successfully removed!")
            completion(true)
        }
    }
}
```

Veja que a função `delete` do documento obtido da coleção `products` através do seu id é invocada para excluir o produto. Caso o produto seja e fato apagado, então o valor `true` é informado para quem chamou a função `deleteProduct`.

De volta na `struct ContentView`, é necessário criar uma variável de estado para controlar a exibição de um alerta ao usuário, caso a operação de excluir o produto falhe, como mostra o trecho a seguir:

```
@State private var showAlert = false
```

Logo abaixo da declaração do sheet, crie o alerta que será controlado por essa variável:

```
.alert(isPresented: $showAlert) {
    Alert(title: Text("Error"),
          message: Text("The product couldn't be deleted"), dismissButton: .default(
    Text("Dismiss")))
}
```

E para de fato chamar a função de exclusão do produto do `viewModel`, cria a função privada para fazer tal trabalho e controlar a variável de estado `showAlert` com o resultado dessa operação, como mostra o trecho a seguir:

```
private func deleteProduct(indexSet: IndexSet) {
    let productToDelete = self.productListViewModel
        .products[indexSet.first!]

    productListViewModel
        .deleteProduct(product: productToDelete) { result in
            showAlert = !result
        }
}
```

O princípio aqui é o mesmo utilizado no capítulo 7, ou seja, o produto deve ser localizado na lista e então passado ao *viewModel*,

Essa função então deverá ser chamada no método `onDelete`, que deve ser criado logo após o fechamento da chave do `ForEach`:

```
.onDelete(perform: self.deleteProduct)
```

Para testar essa implementação, execute a aplicação e faça o gesto de clicar em um item da lista e arrastar para esquerda. A opção de excluir o produto deverá aparecer. Quando o produto for apagado do dispositivo, ele também será apagado do Firestore.

10.13 - Criando o *viewModel* de edição de produtos

O *viewModel* para a edição de um produto seguirá um princípio muito parecido com o que foi desenvolvido no capítulo 7, por isso crie sua nova classe de nome `EditProductViewModel` na pasta `ViewModels`, como mostra o trecho a seguir:

```
import Foundation
import Firebase
import FirebaseFirestoreSwift

class EditProductViewModel {
    private var id: String?
    var name: String = ""
    var description: String = ""
    var code: String = ""
    var price: Double = 0.0

    private let db = Firestore.firestore()

    func setProduct(product: Product) {
        self.id = product.id
        self.name = product.name
        self.description = product.description
        self.code = product.code
        self.price = product.price
    }
}
```

Até o momento, não existe nada de especial. Apenas estão sendo declarados os atributos que serão editados pelo usuário, a instância do Firestore e a função para receber o produto a ser editado.

A seguir, a função para fazer a edição de fato deve ser criada, como no trecho a seguir:

```
func updateProduct() -> Bool {
    let product = Product(userID: Auth.auth().currentUser!.uid,
                          name: self.name,
                          description: self.description,
                          code: self.code,
                          price: self.price)

    let productRef = db.collection(Product.COLLECTION).document(self.id!)

    do {
        try productRef.setData(from: product)
        print("The product has been updated - ID: \(productRef.documentID)")
        return true
    } catch let error {
        print("Error while updating the product: \(error)")
        return false
    }
}
```

A primeira instrução dessa função cria uma instância de `Product`, que já contém tudo o que foi editado pelo usuário, bem como a identificação única do usuário autenticado.

Em seguida o documento que representa esse documento no Firestore é obtido através e sua identificação única, através dessa instrução:

```
let productRef = db.collection(Product.COLLECTION).document(self.id!)
```

Desse posse desse documento, é possível fazer a alteração necessária nele, chamando sua função `setData`, passando a entidade `product` criada anteriormente. O Firestore então se encarrega de fazer a alteração no documento.

Caso tudo dê certo, a função `updateProduct` retorna `true` para indicar que a operação foi realizada com sucesso.

Agora é possível construir a *view* para utilizar essa função, como será visto na próxima sessão.

10.14 - Construindo a view de edição de produtos

A *view* de edição de um produto também será muito parecido com a que foi criada no capítulo 7. Por isso comece criando uma nova *struct* na pasta `Views`, de nome `EditProductView`, como mostra o trecho a seguir:

```
import SwiftUI

struct EditProductView: View {
    @Environment(\.presentationMode)
    var presentationMode: Binding<PresentationMode>

    var product: Product

    @State private var showAlert = false
    @State private var editProductViewModel = EditProductViewModel()

    var body: some View {
        Text("Hello, World!")
    }
}

struct EditProductView_Previews: PreviewProvider {
    static var previews: some View {
        EditProductView(product: Product(userId: "1", name: "Product1",
                                         description: "desc", code: "cod", price: 0.0))
    }
}
```

Até o momento, não existe nada de novo nesse trecho de código para começar a nova tela.

Agora crie a função `init`, que será invocada quando a tela aparecer, para configurar o produto a ser editado no `viewModel`, como mostra o trecho a seguir:

```
init(product: Product) {
    self.product = product
    self.editProductViewModel.setProduct(product: product)
}
```

Também deve ser criada a função que será invocada quando o usuário concluir a alteração do produto:

```
private func updateProduct() {
    showAlert = false
    if editProductViewModel.updateProduct() {
        self.presentationMode.wrappedValue.dismiss()
    } else {
        showAlert = true
    }
}
```

Veja que o retorno da função `updateProduct`, que diz se a operação foi de fato concluída com sucesso, é utilizada para definir se o usuário deve voltar à tela inicial ou se um alerta deve ser exibido a ele.

Agora altere a implementação da variável `body` para começar a construir de fato a tela de edição. Comece definindo o título da barra de navegação, seus botões e o alerta a ser exibido ao usuário caso a operação falhe, como mostra o trecho a seguir:

```
var body: some View {
    Form {
        .navigationBarTitle("Edit product", displayMode: .inline)
        .navigationBarItems(trailing: Button(action: updateProduct) {
            Text("Save").foregroundColor(.blue)
        })
        .alert(isPresented: $showAlert) {
            Alert(title: Text("Error"),
                  message: Text("The product couldn't be updated"),
                  dismissButton: .default(Text("Dismiss")))
        }
    }
}
```

Por fim, dentro do componente `Form`, antes da chamada a `navigationBarTitle`, crie a seção do formulário de edição do produto:

```
Section() {
    TextField("Enter the name",
        text:$editProductViewModel.name)

    TextField("Enter the description",
        text:$editProductViewModel.description)

    TextField("Enter the code",
        text:$editProductViewModel.code)

    PriceField(value: $editProductViewModel.price)
}
```

Veja que toda a implementação dessa tela ficou bem parecida com a do capítulo 7.

Perceba também que a tela que exibe a lista de produtos não é avisada da alteração do produto em si. Isso porque ela já está observando tal lista e será notificada dessa alteração.

Para que essa tela agora possa ser exibida, volte na *struct ContentView* e envolva o componente `HStack` com o `NavLink`, para que a navegação possa ser feita quando o usuário clicar sobre um item da lista de produtos:

```
NavLink(destination: EditProductView(product: product)) {
    HStack {
        Text(product.name)
        Spacer()
        Text(product.code)
        Spacer()
        Text(String(format: "$ %.2f", product.price))
    }
}
```

Para testar essa implementação, execute a aplicação e clique sobre um dos produtos da lista. Faça alguma alteração, como por exemplo seu código, e clique em `Save`. Perceba que o usuário é navegado de volta para a tela inicial, que já exibe a alteração feita.

10.15 - Testando a sincronização automática da lista de produtos

Um dos recursos importantes do Firestore é fazer com os clientes estejam sempre sincronizados com as alterações feitas nas coleções que eles observam. Isso significa que se alguma alteração for feita no Firestore, todos os clientes receberão essa atualização.

Para poder testar essa funcionalidade, mantenha a tela do app com a lista de produtos e vá até o console do Firestore. Nele, altere, por exemplo, o código de um dos produtos criados pelo usuário que está logado no app nesse momento. Esse produto será alterado imediatamente na tela do app, pois ele está observando esse produto, bem como todos os que esse usuário criou.

Esse efeito também pode ser observado se um outro dispositivo logado com a mesma conta criar um novo produto. Esse novo produto deverá aparecer em todos os dispositivos em que esse usuário está logado.

10.16 - Conclusão

Nesse capítulo foi tratado conceitos importantes do Firestore, um banco de dados não-relacional em formato de documentos com atualizações em tempo real.

Tudo foi feito com base no usuário que foi logado com o Firebase Authentication, mostrando a integração perfeita entre esses dois serviços.

Esse projeto será aproveitado para o próximo capítulo, onde as ações do usuário poderão ser monitoradas utilizando o Analytics.

11 - 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 iOS, 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.

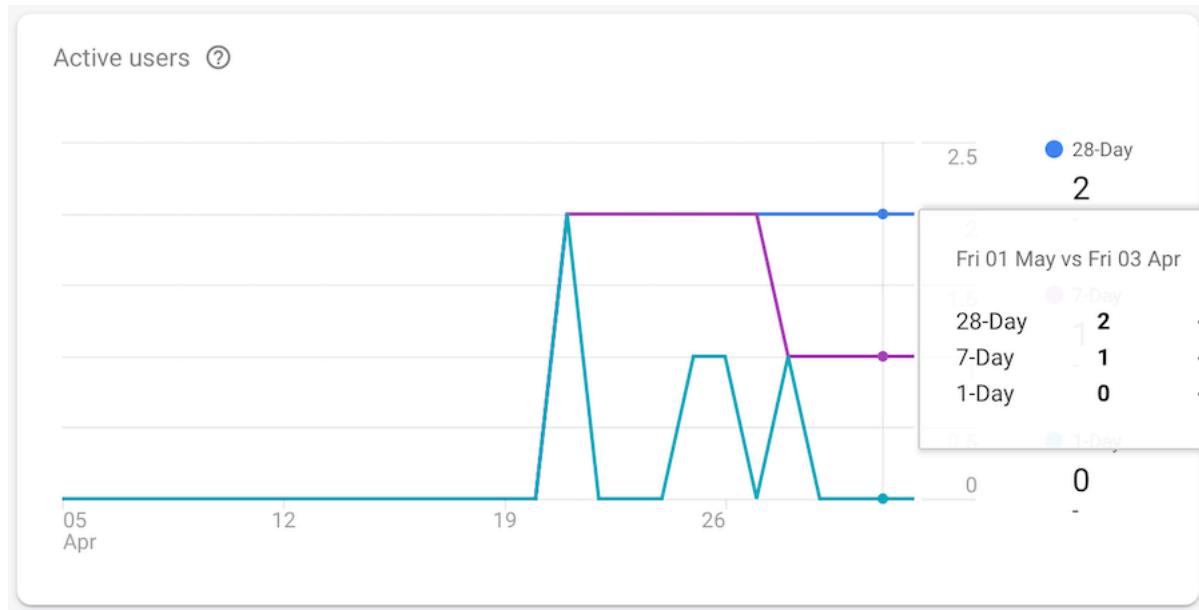
Também é possível criar uma audiência com usuários específicos e alterar o comportamento do aplicativo através do Firebase Remote Config, como será visto no próximo capítulo.

A ideia desse capítulo é utilizar o mesmo projeto do capítulo passado, adicionando os seguintes eventos:

- Seleção de um produto para ser visualizado;
- Intenção de criação de um novo produto;
- Exclusão de um produto.

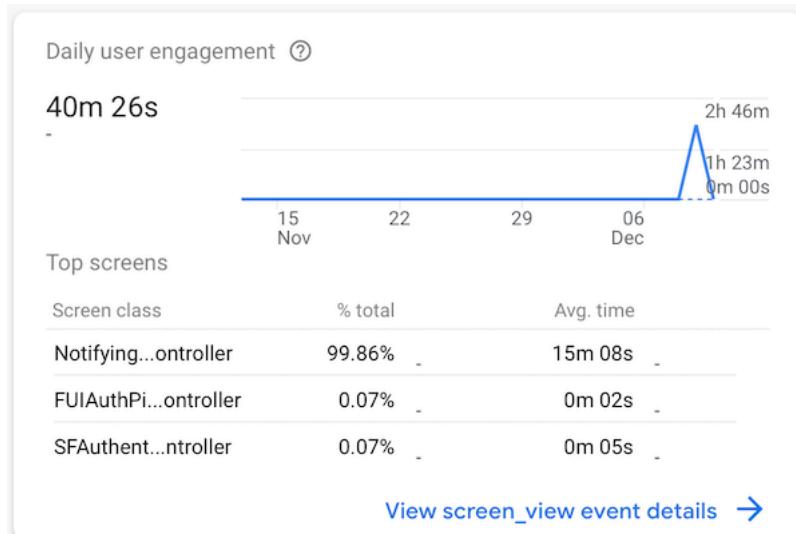
11.1 - Analisando o dashboard do Firebase Analytics

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 iOS, 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.

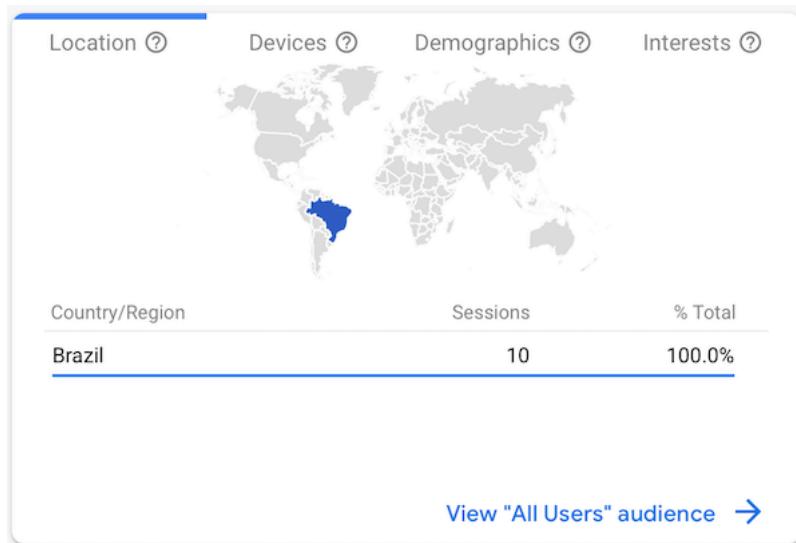
Também é possível ter uma estimativa mais detalhada do engajamento dos usuários, como pode ser visto na figura a seguir:



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:



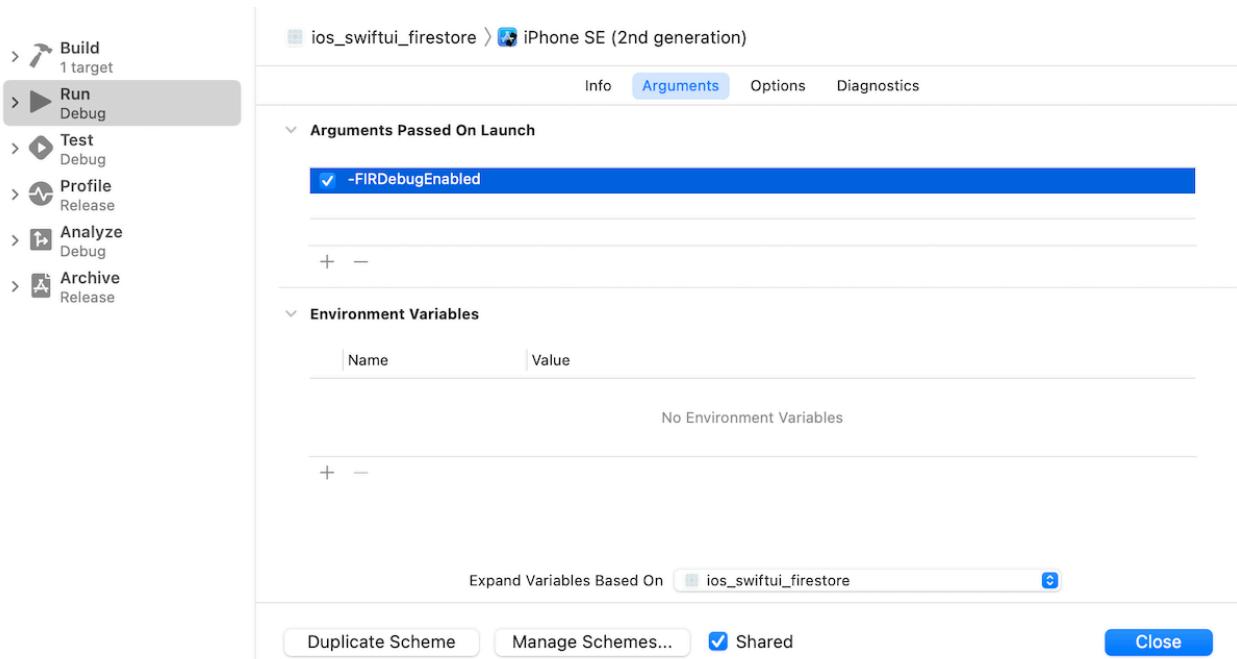
Localização dos usuários

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.

11.2 - Preparando o projeto para o Firebase Analytics

Uma função interessante do Firebase Analytics é o DebugView, que permite que os eventos gerados em um dispositivo de teste possa ser visto em seu console. Essa funcionalidade é muito interessante para poder acompanhar todos os eventos gerados por esse dispositivo, em tempo de desenvolvimento. Isso será demonstrado mais adiante nesse capítulo, mas para isso é necessário habilitar essa função no dispositivo de teste. Para isso, no Xcode, acesse o menu Product -> Scheme -> Edit Scheme....

Nessa tela, adicione o argumento `-FIRDebugEnabled`, como pode ser visto na figura a seguir:



Habilitando o DebugView no dispositivo de teste

Agora os eventos, que serão gerados na seção seguinte, poderão ser vistos no DebugView do console do Firebase Analytics.

11.3 - Gerando eventos do usuário

A geração de eventos com o Firebase Analytics pode conter ou não parâmetros, que ajudam no entendimento da ação do usuário. Esses eventos, assim como seus parâmetros podem ser customizados, ou seja, o desenvolvedor pode criar seus eventos e parâmetros, bem como pode utilizar de alguns já pré-definidos pelo Firebase.

Para começar com o evento gerado quando o usuário visualiza um produto, vá até o arquivo `EditProductView` e importe a biblioteca do Firebase nele:

```
import Firebase
```

Isso permitirá a geração do evento, que deve ser colocado dentro da função `init` desse mesmo arquivo, como no exemplo a seguir:

```
Analytics.logEvent(FirebaseAnalytics.AnalyticsEventSelectItem,  
    parameters: [AnalyticsParameterItemID: product.code  
])
```

Veja que a função `logEvent` de `Analytics` é chamada para gerar um evento do tipo `AnalyticsEventSelectItem` com o parâmetro `AnalyticsParameterItemID` que contém o código do produto que está sendo visualizado pelo usuário. Nesse caso, estão sendo utilizados eventos e parâmetros já definidos pelo Analytics.

O segundo evento que pode ser gerado é quando o usuário abre a janela para a criação de um novo produto, como no trecho a seguir, que deve ser colocado na `struct AddNewProductView`, logo após a definição do `alert` exibido ao usuário:

```
.onAppear() {  
    Analytics.logEvent("new_item", parameters: nil)  
}
```



Lembre-se de importar o Firebase nesse arquivo.

Veja que aqui o evento `new_item` está sendo fornecido, ao invés de um já padronizado pelo Analytics. O último evento a ser gerado como exemplo é da tentativa de exclusão de um produto pelo usuário. Isso pode ser feito no arquivo `ContentView`, dentro da função `deleteProduct`, como mostra o trecho a seguir:

```
Analytics.logEvent("attempt_delete_product", parameters: [  
    AnalyticsParameterItemID: productToDelete.code  
])
```



Lembre-se de importar o Firebase nesse arquivo.

Nesse caso, o evento é customizado e está sendo utilizado um parâmetro padrão do Analytics.

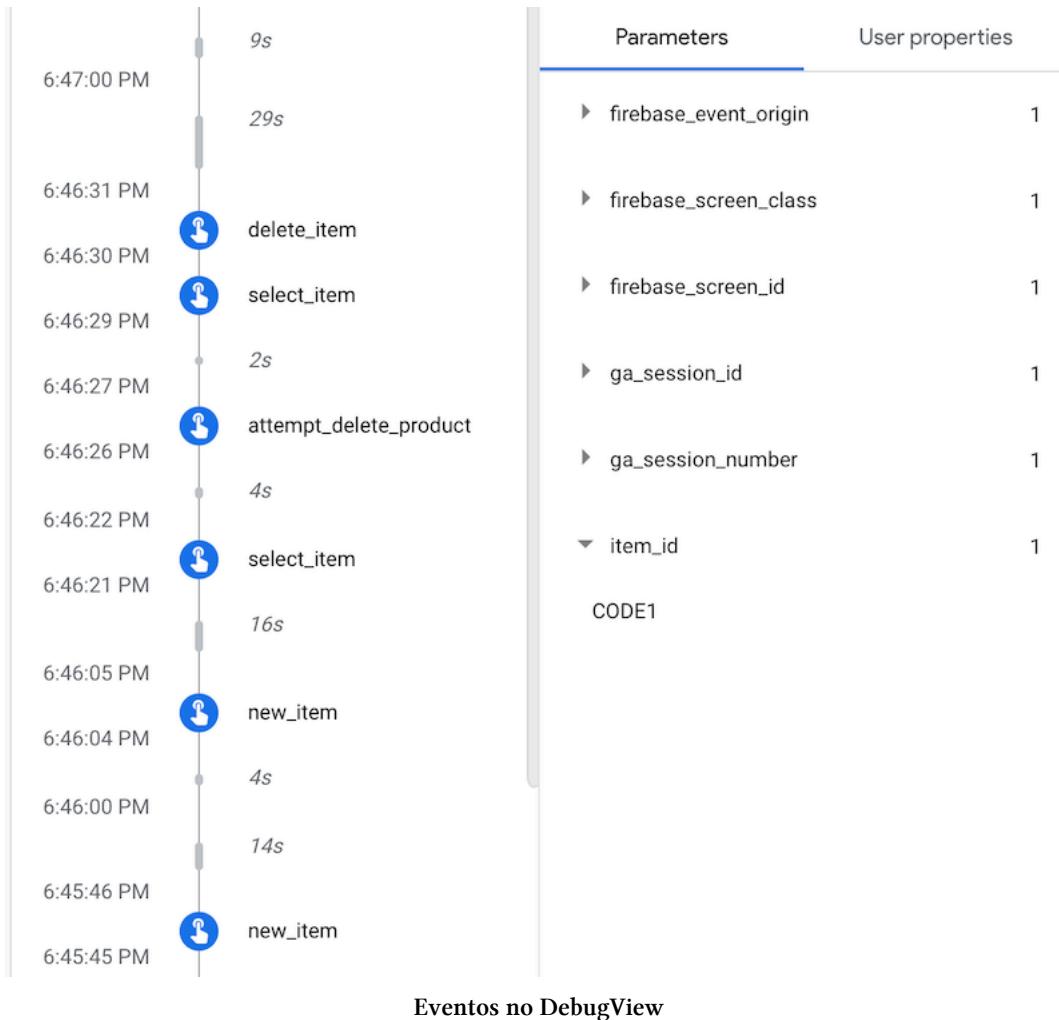
Agora é possível executar a aplicação e visualizar os eventos aparecendo no `DebugView` do console do Analytics, quando usuário fizer as ações citadas acima, como será visto na seção seguinte.

11.4 - 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 accese 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:

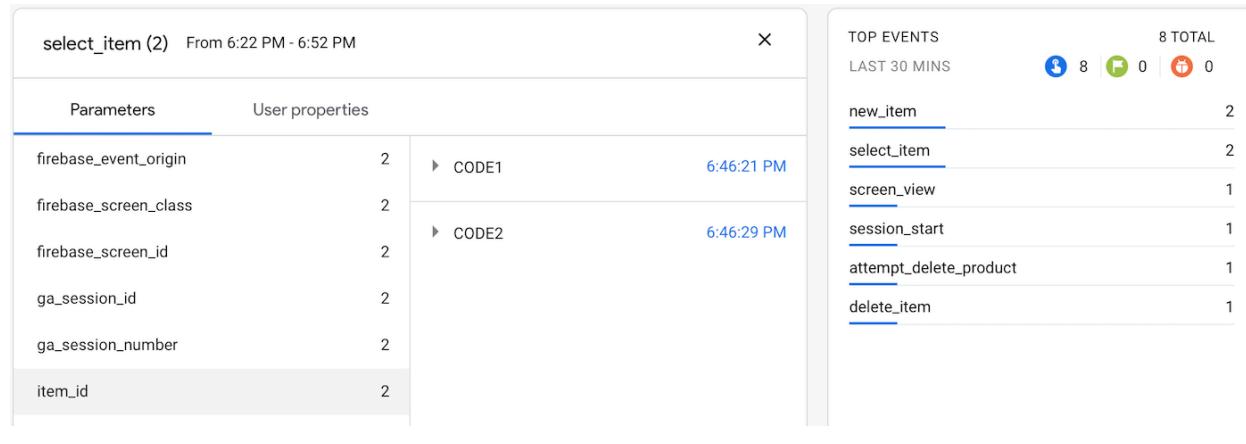


Nesse exemplo os seguintes eventos foram gerados:

- 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.

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		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 iOS, 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.

11.5 - 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 iOS 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.

12 - 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 10, possa ser habilitada ou não, através de uma configuração que será criada no Firebase Remote Config.

12.1 - Preparando o app para o Firebase Remote Config

Preparar o app para o Firebase Remote Config é bem simples. Basicamente devem ser feitos os seguintes passos na inicialização do app:

- Configurar o intervalo mínimo entre as buscas que o app deve fazer por novas configurações;
- Configurar os valores padrões das propriedades de configuração, que deverão ser utilizadas se o app ainda tiver buscado as configurações feitas no Firebase Remote Config.

Esses dois passos devem, obviamente, ser feitos logo após a inicialização do Firebase, ou seja, após a instrução `FirebaseApp.configure()` da função `init` da classe principal do projeto, como mostra o trecho a seguir:

```
init() {
    FirebaseApp.configure()

    let remoteConfig = RemoteConfig.remoteConfig()
    let settings = RemoteConfigSettings()
    settings.minimumFetchInterval = 60
    remoteConfig.configSettings = settings

    let defaults: [String: Any?] = [
        "delete_list_view" : true
    ]

    remoteConfig.setDefaults(defaults as? [String: NSObject])
}
```

Veja que a configuração do `RemoteConfig` é obtida através das instrução `RemoteConfig.remoteConfig()`.

Depois disso é criada a configuração para o intervalo mínimo de busca da configuração no Firebase Remote Config. Esse valor na verdade deve ser em torno de algumas horas, para evitar que os dispositivos fiquem consultando a todo instante o Firebase. O valor configurado aqui de 60 segundos é apenas para ser utilizado durante os testes do app.

Logo em seguida essa configuração é aplicada ao objeto `RemoteConfig` através das instrução `remoteConfig.configSettings = settings`.

O segundo passo começa pela criação de um dicionário com os valores padrões a serem assumidos pelo app, caso ele não ainda não tenha obtido os valores configurados no console do Firebase Remote Config. Isso é obtido pela criação da variável `defaults` com a chave `delete_list_view` com o valor `true`. Posteriormente, esse valor será configurado no console do Firebase Remote Config, para controlar se o usuário poderá ou não apagar um produto da lista. Por enquanto, essa chave terá o valor `true` para permitir que o usuário possa realizar essa ação.



A estratégia de adicionar valores padrões à configuração local do Remote Config faz com que o app fique apto a trabalhar com configurações que ainda não foram feitas no console do Firebase, por exemplo em caso de uma versão ainda não ter sido lançada.

Esse dicionário então é aplicado à configuração padrão da instância do Remote Config do app, para que possa ser utilizado por ele.

Quando o app conseguir fazer o carregamento das configurações feitas no console do Remote Config, como será visto a seguir, esse valor será substituído pelo que estiver configurado no console do Firebase.

12.2 - Buscando as configurações do Remote Config

Agora é necessário fazer com que o app busque suas configurações no Remote Config, quando ele inicializar. Essas configurações então poderão ser utilizadas para balizar seu comportamento, como é o caso do exemplo desse capítulo, que irá controlar se o usuário poderá ou não ter acesso à exclusão do produto na tela principal.

Isso pode ser feito na mesma função `init` trabalhada na seção passada, logo após a configuração dos valores padrões das configurações, como pode ser visto no trecho a seguir:

```
remoteConfig.fetchAndActivate { (status, error) -> Void in
    if status == .successFetchedFromRemote {
        print("The RemoteConfig has been fetched!")
    } else {
        print("RemoteConfig not fetched")
    }
}
```

Esse trecho irá buscar todas as configurações que se aplicam àquela instância do aplicativo, no que diz respeito a fatores como idioma, localização e outros, como será visto na próxima seção.

Uma vez obtida a configuração que se aplica à instância desse app, ela estará disponível para ser utilizada em qualquer ponto do código, como no exemplo a seguir:

```
let value = RemoteConfig.remoteConfig()
    .configValue(forKey: "delete_list_view").boolValue
```

Esse seria a instrução para obter a configuração para a chave `delete_list_view`, que será utilizada para configurar a ação de delete do produto na tela principal. Como dito anteriormente, quando essa instrução for utilizada em qualquer ponto do código, ela trará o valor da configuração realizada no Remote Config para essa chave em específico. Tal valor foi buscado na inicialização do app, como visto no início dessa seção.

Caso o valor seja alterado no Firebase Remote Config, a próxima vez que o app for inicializado, depois do intervalo mínimo configurado na seção passada, que foi de 60 segundos, ele irá tentar buscar as configurações novamente, trazendo os valores atualizados do Remote Config. Isso significa que o comportamento do app pode ser alterado apenas modificando essa variável, como será visto na seção seguinte.

12.3 - Mudando o comportamento do app com o Firebase Remote Config

Para utilizar o valor `delete_list_view` para controlar a ação de delete de produto da lista principal, vá até a `struct ContentView` e crie a seguinte função para trazer esse valor do que foi buscado do

Remote Config:

```
func isDeleteEnabled() -> Bool {
    return RemoteConfig.remoteConfig()
        .configValue(forKey: "delete_list_view").boolValue
}
```

Quando essa função for chamada, irá dizer se o usuário poderá ou não apagar um produto da lista, mas perceba que essa informação vem do que foi carregado do Remote Config, quando o app foi aberto, através da chave `delete_list_view`.

Para aplicar de fato o uso dessa função, procure pela definição do evento `onDelete` e altere para o trecho a seguir:

```
.onDelete(perform: isConfigEnabled() ? self.deleteProduct : nil)
```

Veja que, caso o valor retornado pela função seja `true`, então a função `deleteProduct` é oferecida como parâmetro da função `onDelete`, o que faz com que ela fique habilitada ao usuário. Caso contrário, é passado o valor `nil`, que automaticamente desabilita tal ação disponível ao usuário.

12.4 - Criando o parâmetro no console do Firebase Remote Config

Para criar o parâmetro de configuração, vá até o console do Firebase, na seção Engage e clique na opção `Remote Config`. A seguinte tela deve aparecer, caso nenhum parâmetro tenha sido criado ainda:

Add a parameter

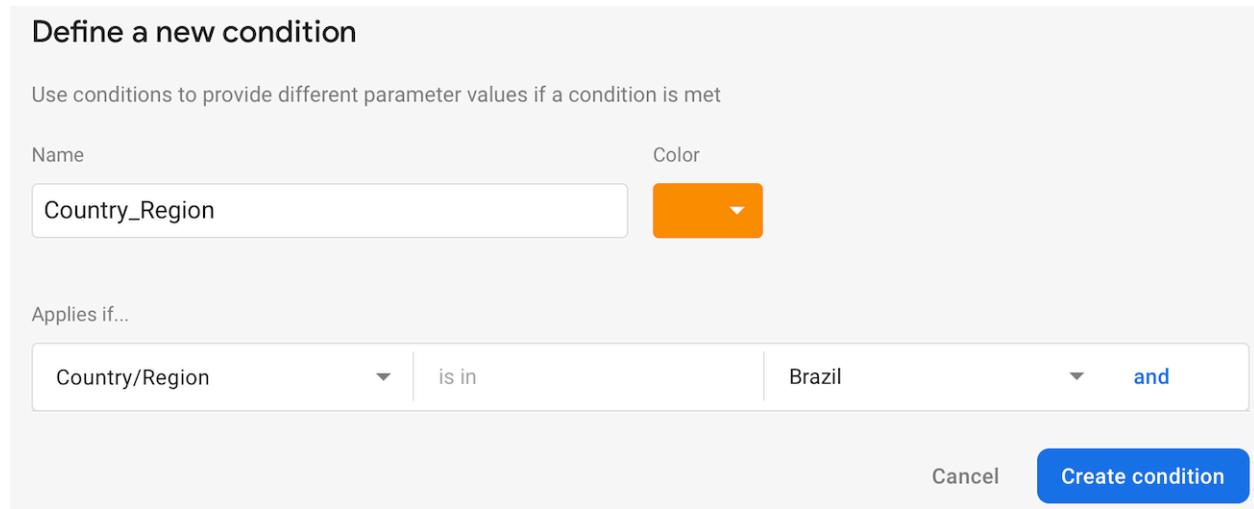
Parameter key	Default value	Add value for condition
Example: holiday_promo_enabled	(empty string)	Other empty values { }
Add description		Add parameter

Tela inicial do Firebase Remote Config

Para começar a criação do parâmetro de configuração, digite o valor `delete_list_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:



Criando a condição do parâmetro de configuração

Em seguida, clique em Create condition. Para finalizar, clique no botão Add parameter. Isso fará com que o parâmetro seja 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 Contry_Region, aguarde em torno de 60 segundos e execute a aplicação novamente. Se ele buscou uma nova configuração, a seguinte mensagem deverá aparecer na aba de logs do Xcode:

The RemoteConfig has been fetched!

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.

12.5 - 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.

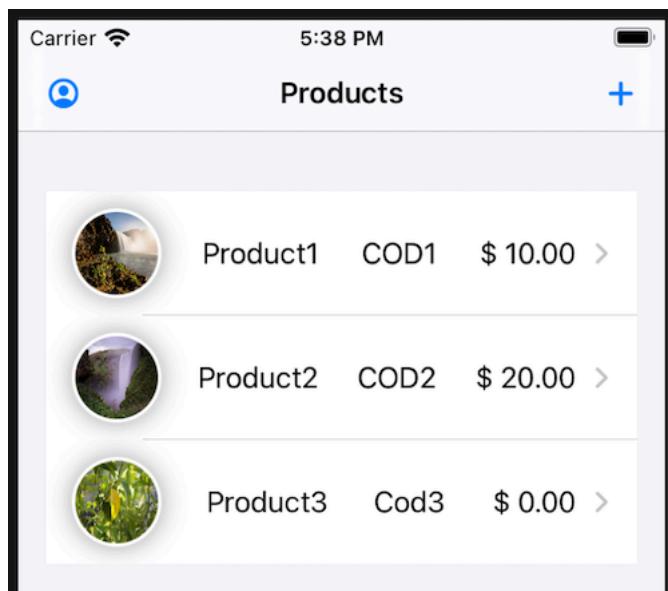
13 - Salvando fotos dos produtos no Firebase Storage

O Firebase Storage é uma excelente opção para armazenamento de objetos, como imagens, áudio e vídeos, de forma segura e descomplicada, através de aplicativos Android, iOS ou Web. Com ele é possível definir regras de segurança baseados nos usuários que estão autenticados no app, assim como foi feito com os dados armazenados no Firestore.

A ideia desse capítulo é adicionar uma foto aos produtos que estão sendo salvos no app que está sendo construído. Essa foto será armazenada no Firebase Storage, podendo ser acessada somente pelo usuário que é dono do produto criado, garantindo a segurança do arquivo, seguindo os mesmos critérios criados para a proteção dos dados salvos no Firestore.

Também será utilizado um mecanismo de *cache* local, para que o usuário ainda consiga ver as fotos dos produtos, mesmo que o aparelho não tenha conexão com a Internet. Com esse *cache* também será possível evitar o consumo desnecessário de rede do aparelho, baixando as imagens somente quando não estiverem presentes nesse *cache*.

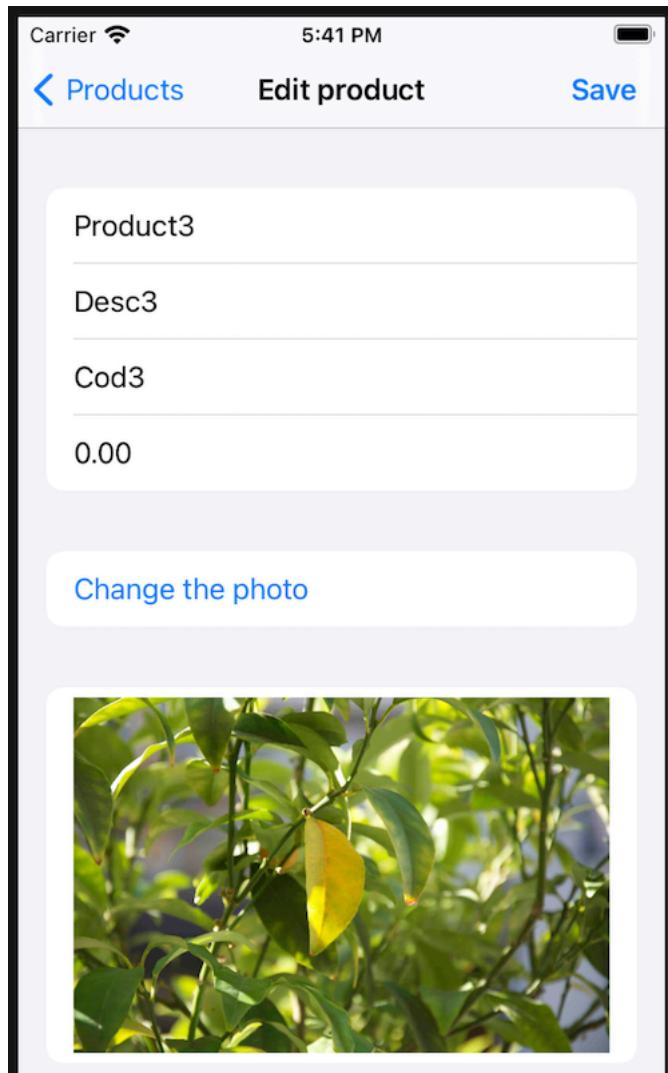
A seguir, veja como deverá ficar a tela principal do app:



Lista de produtos com fotos

As fotos de cada produto, nessa lista, serão carregadas do Firebase Storage, mas com o auxílio do *cache* local.

A tela de edição de produtos deverá ficar como mostra a figura a seguir:



Edição do produto com foto

Quando o usuário entrar nessa tela, ele poderá visualizar a foto do produto, bem como alterá-la.

A tela de criação de produto também terá uma opção para que o usuário selecione a foto. Em ambas as *views*, será utilizado o componente `UIImagePickerController` para abrir a galeria de fotos do aparelho, permitindo que o usuário selecione a que ele desejar.

A seguir, os passos que serão necessários para desenvolver essa funcionalidade de salvar fotos para os produtos:

- Criar as regras de segurança de acesso no Firebase Storage;
- Adicionar as bibliotecas específicas disso ao projeto do app no Xcode;
- Alterar o modelo de produtos que é salvo no Firestore;
- Criar um modelo específico para ser utilizado pela tela de listagem de produtos;
- Criar a *view* para escolha das fotos que está na galeria do dispositivo;

- Alterar os *viewModels* para acessarem o Firebase Storage quando estiverem manipulando um produto no Firestore;
- Alterar a *view* de listagem de produtos para exibir uma foto pequena de cada produto;
- Modificar a *view* de inclusão de um novo produto, para que o usuário possa escolher a foto;
- Alterar a *view* de edição do produto, para que o usuário veja a foto do produto e possa alterá-la, se desejar.

Esses passos serão detalhados nas próximas seções desse capítulo.

13.1 - Estrutura e organização do Firebase Storage

No Firebase Storage é possível criar *buckets* com uma organização de pastas e seus arquivos. Os *buckets* podem ser criados para separar objetos de diferentes apps ou mesmo para propósitos distintos dentro do projeto do Firebase.

Com a criação do projeto no Firebase, um *bucket* padrão é criado e dentro dele podem ser criadas pastas e sub-pastas.

A organização a ser adotada nesse exemplo é criar um pasta de nome `users` dentro do *bucket* padrão. Por sua vez, cada usuário terá sua pasta dentro da pasta `users`, tendo como o nome a identificação única do usuário autenticado no Firebase Authentication.

Em cada pasta do usuário, serão colocados os arquivos das fotos de seus produtos. Veja na figura a seguir como essa organização deve ficar:

Name	Size	Type	Last modified
5B4C780E-6451-4DFD-8E92-93C9D26B2F47	239.58 KB	application/octet-stream	Dec 17, 2020
86FAFCFE-F3A6-4B9B-8E4E-FB8366DF3040	384.89 KB	application/octet-stream	Dec 17, 2020
AE07F0C2-E133-4193-98F3-0B0CF4BAB774	302.53 KB	application/octet-stream	Dec 17, 2020
B1B5487E-353C-4814-A6D4-2ECA2306ABB3	304.13 KB	application/octet-stream	Dec 17, 2020

Estrutura de pastas e arquivos



Essa visualização pode ser obtida no console do Firebase Storage, na seção Storage, na aba Files.

Na figura anterior, podem ser observados os seguintes pontos principais:

- Em 1, está o nome do *bucket* utilizado para esse exemplo, criado dentro do projeto do Firebase;
- A pasta utilizada para armazenar as fotos dos produtos dos usuários está na pasta de nome `users`, como mostra o ponto 2;
- Dentro da pasta `users`, há uma pasta para cada usuário, com o nome contendo a identificação única desse usuário no Firebase Authentication, como pode ser visto em 3;
- Por último, está o nome o arquivo, como mostra o ponto 4, que será um UUID gerado para cada upload realizado.

O nome do arquivo, que é uma String contendo um UUID, será armazenado no documento com as informações do produto, como pode ser visto na figura a seguir:

The screenshot shows a document in the Firestore database with the following fields:

- `code: "COD1"`
- `description: "Desc1"`
- `imageId: "86FAFCFE-F3A6-4B9B-8E4E-FB8366DF3040"` (This field is highlighted with a red rectangle.)
- `name: "Product1"`
- `price: 10`
- `userId: "T1WMe"`

Below the document, the text "Identificação da imagem no documento de produto" is displayed.

O novo campo que será adicionado no documento de produto, de nome `imageId`, será o nome do arquivo que será armazenado dentro da pasta do usuário no Firebase Storage, dessa forma o app poderá saber qual arquivo está associado a cada produto.

13.2 - Criando as regras de segurança de acesso

Para garantir que somente os usuários possam acessar seus arquivos dentro do Firebase Storage, é necessário criar regras de segurança de acesso dentro dele, assim como foi feito no Firestore. Essas

regras utilizam o mesmo princípio do que já foi feito no capítulo 9, ou seja, a informação do usuário autenticado no Firebase Authentication será utilizada aqui também.

Para começar a criar essas regras, vá até o console do Firebase Storage e clique na aba Rules. Essa página, que é muito parecida com a do Firestore, é possível definir as regras de segurança, seguindo o mesmo princípio e formato.

O padrão da regra parte da estrutura dos arquivos, como foi mostrado anteriormente, que seria como o trecho a seguir:

```
/users/{userId}/{imageId}
```

Dentro do *bucket*, existe a pasta `users` e a sub-pasta com a identificação única do usuário, que conterá todas as fotos dos seus produtos.

As operações que devem ser controladas podem ser limitadas a **escrita e leitura**. Logo, a regra deverá ficar como no trecho a seguir:

```
rules_version = '2';
service firebase.storage {
  match /b/{bucket}/o {
    match /users/{userId}/{imageId} {
      allow read, write: if request.auth.uid == userId;
    }
  }
}
```

Veja que as operações de leitura e escrita, que são as únicas liberadas, só podem ser realizadas de fato se o id do usuário autenticado for o mesmo da pasta que ele está tentando acessar. Isso significa que somente o usuários podem acessar essa sub-pasta e seu conteúdo.

Depois de colocar essas regras, clique em `Publish`, na parte superior da tela, para torná-las efetivas.

Obviamente, o app terá que ser desenvolvido para que ele grave e leia arquivos da pasta do usuário que está autenticado, como será visto adiante, para que ele esteja de acordo com as regras de segurança de acesso que foram criadas.

13.3 - Preparando o projeto

Para poder trabalhar com o Firebase Storage e com seus componentes gráficos, é necessário adicionar mais duas bibliotecas ao projeto, por isso abra o arquivo `Podfile`, que foi gerado no processo de criação do projeto, e acrescente as duas linhas a seguir na lista de `pods`:

```
pod 'Firebase/Storage'  
pod 'FirebaseUI/Storage'
```

Em seguida, feche o Xcode e execute o comando `pod install` de um terminal, de dentro da pasta raiz do projeto.

A primeira biblioteca mencionada anteriormente diz respeito ao SDK do Firestore Storage. A segunda é uma biblioteca que auxilia na exibição de imagens, em conjunto com um sistema de *cache*.

13.4 - Alterando o modelo de produtos

Para que o produto persistido no Firestore passe a guardar também a identificação única de sua foto salva no Firebase Storage, é necessário adicionar mais um campo a ele. Porém, é necessário tomar um cuidado adicional, pois podem existir produtos que já foram criados, e que obviamente não possuem tal informação.

Para isso, abra o arquivo `Product.swift` da pasta `Models` do projeto do app no Xcode e adicione o seguinte atributo à *struct* `Product`:

```
var imageUrl: String?
```

Perceba que o atributo é do tipo `Optional`, justamente para permitir entidades que não tenham esse valor cadastrado no Firestore.

Apesar da alteração ser significativa para a base do projeto, ela não causa nenhum erro ao compilar e rodar, ou seja, tudo continua funcionando perfeitamente.

A próxima vez que um novo produto for salvo, com uma foto associada a ele, esse atributo será preenchido com a identificação única do arquivo de imagem, que será gerada no momento de fazer seu upload, e será salvo no Firestore.

13.5 - Criando o view model para a lista de produtos

Até o momento não foi necessário criar um modelo específico a ser utilizado pela tela de listagem de produtos, como foi feito na seção 6.5.1 do capítulo 6. Essa estratégia pode ser interessante em certos casos, como será agora.

Como a ideia é exibir a foto do produto na tela que exibe sua lista, e ela não está no modelo criado para representar os documentos do Firestore, é necessário então criar um outro modelo que represente totalmente essa entidade, para facilitar a criação dessa nova tela com uma pequena imagem do produto, como foi mostrado no início desse capítulo.

Para isso, vá na pasta `ViewModels` e crie uma nova *struct* chamada `ProductModel`, como no trecho a seguir:

```
import Foundation
import SwiftUI

class ProductModel {
    init(product: Product) {
        self.product = product
    }
}
```

Da mesma forma como foi feito no capítulo 6, é necessário ter um atributo que representa o produto do modelo da camada de persistência, mas também é necessário ter um outro atributo para representar a *view* para exibir sua foto. Por isso, adicione-os, como no trecho a seguir:

```
var product: Product
var image = UIImageView(image: UIImage(systemName: "photo"))
```

Veja que o atributo `image` é do tipo `UIImageView`, que será responsável por exibir a foto do produto, e inclusive já é inicializado com uma foto padrão, para o caso do produto não tiver nenhuma ainda cadastrada.



A ideia de colocar um atributo do tipo `UIImageView` nesse modelo será bem interessante para facilitar o carregamento da imagem quando a lista de produtos for buscada no Firestore, tarefa essa realizada na classe `ProductListViewModel`.

Quando a *view* de listagem de produtos for alterada, ela poderá exibir esse componente, já com a foto do produto.

Para completar esse modelo, crie os *getters* de cada atributo do produto, como no trecho a seguir:

```
var id: String? {
    return self.product.id
}

var name: String {
    return self.product.name
}

var code: String {
    return self.product.code
}

var price: Double {
    return self.product.price
}
```

Esses *getters* serão utilizados pela tela de listagem de produtos para exibir as informações na tela principal.

13.6 - Adaptando a lista de produtos para exibir sua imagem

A primeira alteração a ser feita para adaptar a lista de produtos para que ela exiba suas fotos é trabalhar na classe `ProductListViewModel` para que ela agora passe a ter uma lista de `ProductModel`, o *view model* que foi criado no capítulo anterior, ao invés de uma lista de `Product`. Para isso, abra essa classe e altere seu único atributo para ficar como o trecho a seguir:

```
class ProductListViewModel: ObservableObject {
    @Published var products = [ProductModel]()
```

Agora, na função `fetchProducts` dessa mesma classe, que busca os produtos do Firestore e os converte para a lista representada por esse atributo, faça o mapeamento das entidades `Product` para `ProductModel`, como no trecho a seguir:

```
self.products = documents.compactMap {
    queryDocumentSnapshot -> Product? in
    return try? queryDocumentSnapshot.data(as: Product.self)
}.map(ProductModel.init)
```

Veja que o que foi alterado aqui é apenas a inclusão da instrução `.map(ProductModel.init)`, que faz justamente esse mapeamento para a entidade `ProductModel` de cada elemento da lista de `Product`.



Dessa forma, o modelo que representa os documentos no Firestore está isolado do modelo que representa o produto que será exibido na tela.

Para completar essa primeira parte das alterações, vá até a *struct ContentView*, que desenha a tela principal, e altere a instrução a seguir para passar a entidade `product`, que agora está dentro do `ProductModel`:

```
NavigationLink(destination: EditProductView(product: product)) {
```

Obviamente, o objeto `product` é um elemento da lista `products` devolvida por `ProductListViewModel`, que é do tipo `ProductModel` agora. Para fazer essa alteração, basta acessar o seu atributo `product`, como no trecho a seguir:

```
NavigationLink(destination: EditProductView(product: product.product)) {
```

Algo semelhante deve ser feito na instrução que passa o produto a ser apagado, dentro da função `deleteProduct` da *struct ContentView*:

```
.deleteProduct(product: productToBeDeleted) { result in
```

O atributo a ser passado agora na verdade é o `product` que está dentro de `ProductModel`, como no trecho a seguir:

```
.deleteProduct(product: productToBeDeleted.product) { result in
```

Ainda dentro da *struct ContentView*, volte para onde cada produto é exibido, dentro do componente `HStack` do `NavLink` que fica dentro do `ForEach`. Essa é a estrutura para representar cada linha que traz os produtos da lista. Para inserir a imagem do produto, basta acrescentar um componente de imagem para exibir o atributo `image` de `ProductModel`, como no trecho a seguir:

```
Image(uiImage: product.image.image!)
    .resizable()
    .clipShape(Circle())
    .overlay(Circle().stroke(Color.white, lineWidth: 2))
    .shadow(radius: 7)
    .frame(width: 50, height: 50)
Spacer()
```

Esse trecho deve ficar logo após a abertura da chave do `HStack`, acima do componente `Text` que exibe o nome do produto.

Veja que aqui está sendo exibida a imagem do produto que está em `ProductModel`, com alguns adereços como um círculo a envolvendo-a e alguns efeitos de sombra, como poderá ser visto mais adiante na figura de exemplo.

Executando a aplicação nesse ponto, já é possível ver que a lista exibe uma imagem padrão do produto, que foi inserida na *struct ProductModel*:

```
var image = UIImageView(image: UIImage(systemName: "photo"))
```

A ideia é fazer com que essa foto seja exibida, caso o produto ainda não tenha nenhuma. Veja como deve ficar essa interface gráfica até o momento:

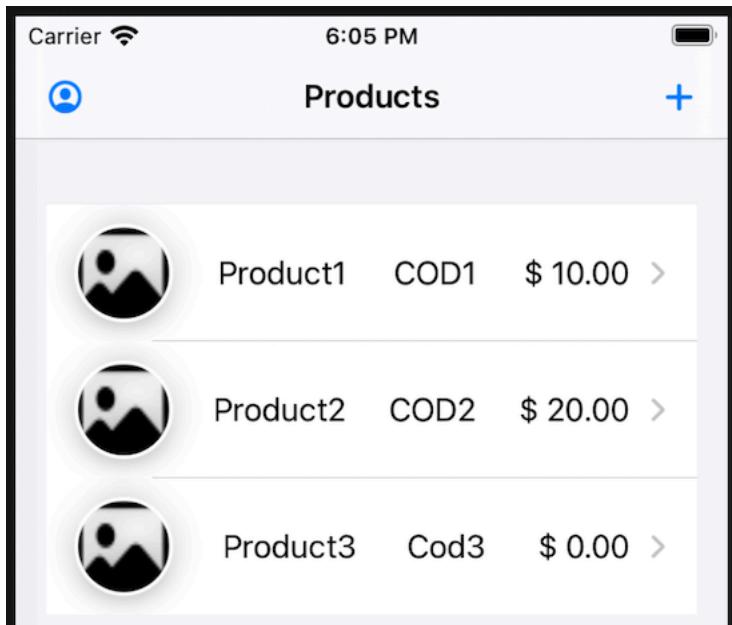


Imagen padrão do produto

Quando um produto tiver uma foto associada a ele, ela aparecerá nesse círculo que fica ao lado do nome do produto, como será visto mais adiante.

E isso finaliza a implementação na tela que exibe a lista de produtos, porém ainda é necessário trabalhar em seu *viewModel* para poder buscar a foto de cada produto, que estarão armazenadas no Firebase Storage, quando eles forem carregados ou quando alguma alteração nessa lista for feita. Da mesma forma, será necessário apagar a imagem associada ao produto quando ele for apagado. Para fazer essas duas implementações, vá até a classe `ProductListViewModel` para criar outras duas funções nela para esses propósitos.



As operações executadas no Firebase Storage, como leitura, upload e exclusão, são assíncronas, logo a implementação deve sempre considerar esse fato.

A primeira operação a ser feita é a de exclusão de uma imagem, que está no Firebase Storage e está associada a um produto. Tanto para essa quanto para as operações de leitura e upload de arquivos, é sempre necessário criar uma **referência** para o arquivo, composto basicamente das seguintes informações:

- *Bucket* a ser acessado, que nesse exemplo será o padrão que já foi criado com a criação do projeto no Firebase;
- Caminho contendo todas as pastas e subpastas, que nesse exemplo seguirá o padrão `/users/{userId}`;
- Nome do arquivo, que no exemplo será um UUID gerado no momento de seu upload.

Tendo essa referência, basta invocar a operação `delete` e passar um *closure* para obter sua resposta, como pode ser observado no trecho a seguir que deve ser inserido na classe `ProductListViewModel`:

```

private func deleteImage(imageId: String) {
    let storage = Storage.storage()
    let storageRef = storage.reference()

    let imageRef = storageRef
        .child("/users/\(Auth.auth().currentUser!.uid)/\(#imageId)")

    imageRef.delete { error in
        if error != nil {
            print("Failed to delete the product image")
        } else {
            print("The product image has been deleted")
        }
    }
}

```

Veja que as 3 primeiras instruções criam a referência ao arquivo, que é formada pelo caminho até a imagem do produto, composta pela identificação única do usuário autenticado no Firebase Authentication e o id da imagem do produto que é recebida como parâmetro na função.

A instrução seguinte invoca a função `delete` para de fato apagar o arquivo a partir dessa referência. O *closure* será invocado quando o arquivo for apagado com sucesso ou algum erro acontecer, que nesse caso está apenas imprimindo mensagens informativas a fim de simplificar o código.

Essa função então deverá ser chamada logo no início de `deleteProduct`, nessa mesma classe, caso o produto tenha uma imagem associada a ele, como pode ser visto no trecho a seguir:

```

if let imageId = product.imageId {
    self.deleteImage(imageId: imageId)
}

```

Caso o produto não possua uma imagem associada a ele, a função de exclusão de imagens do Firebase Storage nem chega a ser chamada.

Repare que o processo de exclusão da imagem e do produto em si ocorrem em paralelo, pois ambos são assíncronos.

Para baixar a imagem do produto, podem ser utilizadas algumas abordagens distintas. Uma delas envolve fazer esse trabalho manual utilizando o SDK do Firebase Storage diretamente, de forma semelhante com o que foi feito para exclusão do produto. Essa **abordagem³⁰** para baixar o arquivo do Storage, apesar de simples, não possui nenhum tipo de mecanismo de *cache*, o que significaria que isso teria que ser feito para evitar baixar o mesmo arquivo várias vezes durante a navegação do usuário pelo app.

³⁰<https://firebase.google.com/docs/storage/ios/download-files>

A ideia aqui é utilizar uma abordagem mais sofisticada, já incluindo um mecanismo de *cache*. Felizmente o conjunto de ferramentas do SDK do FirebaseUI já disponibiliza componentes e mecanismos para baixar um arquivo de imagem, exibi-lo em uma *view* de imagem do iOS e ainda implementar o mecanismo de *cache*, com poucas configurações adicionais. Isso tudo é feito em cima de um componente chamado [SDWebImage³¹](#). Essa será a estratégia adotada para exibição das imagens dos produtos.

Para começar, o mecanismo de *cache* deve ser configurado na inicialização do app, na classe principal do projeto, aquela com seu nome e anotada com `@main`. Basicamente deve ser criada uma configuração para o seu tamanho e então isso deve ser adicionado ao componente SDWebImage, no qual os componentes do FirebaseUI para isso foram construídos, como mencionado anteriormente.

Veja então como deve ficar a configuração desse *cache*, no trecho a seguir, que deve ser adicionado no fim da função `init`, da classe principal do projeto, logo após a configuração do `RemoteConfig`:

```
let cache = SDImageCache(namespace: "tiny")
cache.config.maxMemoryCost = 100 * 1024 * 1024 // 100MB memory
cache.config.maxDiskSize = 50 * 1024 * 1024 // 50MB disk
SDImageCachesManager.shared.addCache(cache)
SDWebImageManager.defaultImageCache = SDImageCachesManager.shared
```

Obviamente, para esse código compile, é necessário importar o `FirebaseUI`:

```
import FirebaseUI
```

Veja que o *cache* em memória está sendo configurado para 100 MB e o em disco para 50 MB. Obviamente esses valores podem ser configurados de forma diferente, dependendo dos tipos de arquivos e da utilização do app.

Agora que o *cache* já foi configurado, é possível partir para a implementação do mecanismo de download dos arquivos na classe `ProductListViewModel`, que como dito anteriormente, será feito com a utilização dos componentes do `FirebaseUI`. Para isso, importe-o nessa classe:

```
import FirebaseUI
```

Para começar a implementar esse mecanismo, é necessário lembrar que o objeto manipulado e publicado pela classe `ProductListViewModel` é uma lista de `ProductModel`. Dentro de cada item dessa lista está a representação do documento que guarda o produto, bem como o componente gráfico responsável por exibir sua imagem. Logo, a ideia é criar uma função que varre essa lista, olhando para identificação única da imagem que está armazenada no produto, como mostra o trecho a seguir:

³¹<https://github.com/SDWebImage/SDWebImage>

```

private func loadImages() {
    let storage = Storage.storage()
    let storageRef = storage.reference()
    let userid = Auth.auth().currentUser!.uid

    for product in products {
        if let imageUrl = product.product.imageId {

            let imageRef = storageRef
                .child("/users/\(userid)/\("\(imageUrl)")

            print("Loading imageUrl: \(imageUrl)")
            //TODO - carregar a imagem
        }
    }
}

```

Veja que no processo de iteração da lista de ProductModel, é verificado se o produto possui uma identificação única que aponta para uma imagem. Essa informação então é utilizada para criar a referência ao arquivo no Firebase Storage, juntamente com o nome da pasta composta pela identificação única do usuário autenticado no Firebase Authentication.

Cada referência dessa será passada para o componente UIImageView dentro do ProductModel de representa cada produto a ser exibido na lista da tela principal, como mostra o trecho a seguir, que deve ser inserido no lugar do TODO do trecho anterior:

```

product.image.sd_setImage(with: imageRef,
    placeholderImage: UIImage(systemName: "photo")) {
    (image, error, cacheType, storageRef) in

    if let error = error {
        print("Error loading image: \(error)")
    } else {
        print("Image loaded")
        DispatchQueue.main.async {
            product.image = UIImageView(image: image)
            self.objectWillChange.send()
        }
    }
}

```

A função sd_setImage de ImageView foi desenvolvida especialmente para o propósito de baixar uma imagem do Firebase Storage, a partir de uma referência ao arquivo. Veja também que pode ser

fornecida uma imagem padrão, até que a imagem seja baixada completamente, para que o usuário tenha uma melhor experiência visual.

Dentro do *closure* que é invocado quando a operação é concluída, existe a possibilidade de verificar se houve algum erro. Caso nenhum erro aconteça, o componente `UIImageView` é preenchido com a imagem, que eventualmente pode vir do *cache*, caso o arquivo exista nele.

Perceba que logo após a alteração do atributo `image` de `product`, é invocada a função `objectWillChange.send()` do `viewModel`. Isso é necessário para que o SwiftUI redesenhe a tela, a partir de uma alteração que foi feita internamente em um elemento da lista que é publicada por esse `viewModel`.

Veja também que essas duas instruções comentadas anteriormente estão envolvidas por `DispatchQueue.main.async`, para que elas possam ser executadas dentro da *thread* da UI do app, uma vez que essas alterações iriam modificá-la.

A função `loadImages` recém criada deverá ser invocada no final da execução do `listener` criado quando a lista de produtos é buscada do Firestore, na função `fetchProducts` dessa classe, como pode ser visto no trecho a seguir, um pouco mais expandido para ter um contexto do local da alteração:

```
.addSnapshotListener { querySnapshot, error in
    guard let documents = querySnapshot?.documents else {
        print("Error fetching documents: \(error!)")
        return
    }

    self.products = documents.compactMap {
        queryDocumentSnapshot -> Product? in
        return try? queryDocumentSnapshot.data(as: Product.self)
    }.map(ProductModel.init)

    self.loadImages()
}
```

Essa implementação ainda não pode ser testada, pois nenhum produto possui uma imagem associada a ele, mas isso será feito nas seções seguintes.

13.7 - Criando a view para escolher a foto com o `PHPickerViewController`

Para que o usuário possa escolher a foto que deseja associar a um produto, ele deverá ter acesso à galeria de imagens do dispositivo, para que ele possa escolher a figura que desejar. A melhor forma de fazer isso é utilizando o componente `PHPickerViewController`. Para isso, é necessário criar uma `view`, de forma muito semelhante como foi feito no capítulo 9 com a tela de login do usuário.

Essa nova *view* será implementada por uma *struct* que deverá estar em conformidade com o protocolo `UIViewControllerRepresentable`, como mostra o trecho a seguir, que deve ser criado em um arquivo de nome `ProductPhotoPicker` na pasta `Views`:

```
import Foundation
import SwiftUI
import PhotosUI

struct ProductPhotoPicker: UIViewControllerRepresentable {

    @Binding var showPhotoPicker: Bool
    var completion: (UIImage) -> Void

}
```

Veja que a biblioteca `PhotosUI` é importada aqui, para a criação do *controller* que será exibido para o usuário escolher a foto.

O primeiro atributo criado nessa *struct* é um *binding* para uma variável de estado que será utilizada para fechar a tela de escolha das fotos. Essa variável será utilizada pela *view* que invocar a galeria de imagens do dispositivo, que acontecerá na tela de criação e edição do produto.

O segundo atributo é um *closure* que será passado como parâmetro na criação de `ProductPhotoPicker`. Ele será invocado quando o usuário concluir a escolha da imagem e essa for transformada em um componente `UIImage`, para ser exibida na *view* que o invocou.

O *controller* que será criado aqui precisará receber uma classe que seja capaz de receber os evento que ela pode gerar, será quando o usuário escolher uma foto da galeria do dispositivo. para isso, crie uma classe interna à *struct* `ProductPhotoPicker` para implementar o receptor desse evento, como no trecho a seguir:

```
class Coordinator: PHPickerViewControllerDelegate {
    private let parent: ProductPhotoPicker

    init(_ parent: ProductPhotoPicker) {
        self.parent = parent
    }

}
```

Veja que a nova classe `Coordinator` está em conformidade com o protocolo `PHPickerViewControllerDelegate`, que solicita a criação de uma função chamada `picker`, que será criada mais adiante.

O atributo da classe `Coordinator` é a referência a instância de `ProductPhotoPicker`. Ela precisará dessa referência para alterar a variável de estado `showPhotoPicker`, bem como o *closure* guardado na variável `completion`.

O trabalho da função `picker` da classe `Coordinator` deve ser basicamente varrer o vetor de imagens que pode vir como resultado da escolha do usuário, apesar que será configurado para o usuário pegar somente uma, e carregar o arquivo, gerando um componente do tipo `UIImage` com a foto escolhida pelo usuário, como mostra o trecho a seguir:

```
func picker(_ picker: PHPickerViewController,
            didFinishPicking results: [PHPickerResult]) {
    if results.count > 0 {
        if (results[0].itemProvider.canLoadObject(ofClass: UIImage.self)) {
            results[0].itemProvider.loadObject(ofClass: UIImage.self) {
                image, error in

                    if let error = error {
                        print(error.localizedDescription)
                    } else {
                        self.parent.completion (image as! UIImage)
                    }
                }
            }
        }

        parent.showPhotoPicker = false
    }
}
```

Repare que o parâmetro `results` dessa função é uma lista de resultados, que contém as imagens que podem ter sido selecionadas pelo usuário, mas aqui o interesse é apenas uma imagem, e por isso será carregada somente uma, transformando em um componente do tipo `UIImage`, passado como parâmetro do *closure* de `ProductPhotoPicker`. Logo em seguida, é atribuído o valor `false` à variável de estado `showPhotoPicker`, para que a janela na qual ele foi exibido seja fechada.

A instância da classe `Coordinator` pode ser criada dentro da função `makeCoordinator` de `ProductPhotoPicker`, fazendo parte do protocolo `UIViewControllerRepresentable` assumido por ele, como pode ser visto no trecho a seguir:

```
func makeCoordinator() -> Coordinator {
    Coordinator(self)
}
```

Perceba que o parâmetro `self` é passado no momento da criação da instância de `Coordinator`, que a instância de `ProductPhotoPicker`.

Continuando com as exigências do protocolo `UIViewControllerRepresentable`, é necessário criar a função `updateUIViewController` na *struct* `ProductPhotoPicker`, que nesse caso não será utilizada:

```
func updateUIViewController(_ uiViewController: UIViewControllerType,  
                           context: Context) {  
  
}
```

Por último, a função `makeUIViewController` deve ser construída para de fato construir o *controller* responsável por exibir a tela que permitirá que o usuário escolha uma imagem. Além disso, esse *controller* será configurado para exibir somente imagens, uma vez que a galeria pode ter vídeos e *live photos*. Outra configuração importante a ser feita é permitir que o usuário selecione apenas uma foto na galeria, como pode ser visto no trecho a seguir:

```
func makeUIViewController(context: Context) -> PHPickerViewController {  
    var config = PHPickerConfiguration(photoLibrary: PHPPhotoLibrary.shared())  
    config.filter = .images  
    config.selectionLimit = 1  
  
    let controller = PHPickerViewController(configuration: config)  
    controller.delegate = context.coordinator  
  
    return controller  
}
```

Perceba que a instância de `PHPickerViewController` é criada com a configuração realizada no início da função, bem como tem seu *delegate* apontado para a instância da classe *Coordinator* criada aqui, para que receba o evento quando o usuário escolher a imagem ou cancelar a escolha.

Essa *view* será invocada nas telas de criação e edição do produto, como será visto nas seções seguintes.

13.8 - Salvando o produto com sua foto

A estratégia para salvar o produto com sua foto será composta pelos seguintes passos:

- Criar um botão na tela de criação do produto para o usuário abrir a galeria de fotos do dispositivo, através da *struct ProductPhotoPicker* criada no capítulo anterior;
- Criar um local na tela de criação do produto para que o usuário veja a foto que escolheu para o produto;
- Alterar o *viewModel* de criação de produto para receber a foto selecionada pelo usuário e informar, através de um *callback*, quando as operações de upload da foto no Firebase Storage e gravação do documento no Firestore forem concluídas, para que o usuário possa ser navegado de volta para a tela de listagem de produtos.

É interessante começar pelo *viewModel*, pois ele irá balizar as modificações a serem feitas na *view* de criação do produto, para isso, abra o arquivo `AddNewProductViewModel` e comece criando uma nova função privada para salvar o produto no Firestore, que basicamente é a função que já havia sido implementada, com as seguintes modificações:

- Ao invés de retornar uma `String`, ela não retornará nenhum parâmetro;
- Receber um parâmetro, que é a identificação única da imagem que irá representar a foto do produto no Firebase Storage;
- Receber um segundo parâmetro que é um *closure* para indicar o fim da operação de persistir o documento no Firestore;
- Passar a identificação única da imagem, recebida como parâmetro, na criação da entidade `Product`.

Veja como deve ficar essa nova função, com as alterações citadas acima:

```
private func saveProduct(imageId: String?,
                        completion: @escaping (Bool) -> Void) {

    let product = Product(userId: Auth.auth().currentUser!.uid,
                          name: self.name,
                          description: self.description,
                          code: self.code,
                          price: self.price,
                          imageId: imageId)

    let newProductRef = db.collection(Product.COLLECTION).document()

    do {
        try newProductRef.setData(from: product)
        print("Product saved - ID: \(newProductRef.documentID)")
        completion(true)
    } catch let error {
        print("Error while saving new product: \(error)")
        completion(false)
    }
}
```

O restante do código é muito parecido com a função que já existia nessa classe, e que será modificada mais adiante para invocar essa nova e também a que irá fazer o upload do arquivo no Firebase Storage.

Veja que no final dessa função, ao invés de retornar a identificação única do documento criado no Firestore, é invocado um *closure* com o status dessa operação.

A seguir deverá ser criada uma outra função privada na classe `AddNewProductViewModel`, que deverá receber os seguintes parâmetros:

- Uma representação da imagem escolhida, no formato `Data`, que será utilizada para fazer o upload no Firebase Storage;
- A identificação única da imagem, para compor a referência do arquivo no Firebase Storage;
- O *closure* para informar o status da operação, quando ela for concluída.

De posse dessas informações, a função deverá criar a referência do objeto a ser salvo no Firebase Storage, seguindo o padrão já citado, que é `/users/{userId}/imageId`, como pode ser visto no trecho a seguir:

```
private func saveProductImage(imageData: Data, imageId: String,
                               completion: @escaping (Bool) -> Void) {
    let storage = Storage.storage()
    let storageRef = storage.reference()

    let imageRef = storageRef
        .child("/users/\(Auth.auth().currentUser!.uid)/\(imageId)")

    imageRef.putData(imageData, metadata: nil) { (metadata, error) in
        if metadata == nil {
            print("Failed to upload the image")
            completion(false)
        }

        print("The image has been uploaded")
        completion(true)
    }
}
```

Durante o processo de criação do arquivo no Firebase Storage, um metadata é criado, com algumas informações como a data de criação do objeto. Por isso é útil verificar se tudo deu certo no processo de upload verificando se esse metadata foi criado.

Agora a função `saveNewProduct` pode ser totalmente reformulada para implementar uma lógica de primeiro fazer o upload da imagem no Firebase Storage para depois salvar o documento no Firestore, como pode ser visto no trecho a seguir:

```

func saveNewProduct(imageData: Data?,
                    completion: @escaping (Bool) -> Void) {

    if let imageData = imageData {
        let imageId = UUID().uuidString

        saveProductImage(imageData: imageData, imageId: imageId) {
            success in

                if success {
                    self.saveProduct(imageId: imageId) { result in
                        completion(result)
                    }
                } else {
                    completion(false)
                }
            }
        } else {
            saveProduct(imageId: nil) { result in
                completion(result)
            }
        }
    }
}

```

Veja que primeiramente é verificado se a imagem é de fato passada como parâmetro para a função. Caso não seja, a função que salva o documento no Firestore já é invocada, sem a identificação da imagem. Caso contrário um UUID é gerado e passado como parâmetro para a função que faz o upload da imagem.

Somente depois que o upload da imagem é feito é que a função para salvar o documento no Firestore é invocada, agora com a associação do arquivo da imagem desse produto. Isso é feito para que a função `saveNewProduct` somente informe que seu trabalho concluiu quando ambas as operações foram finalizadas, com sucesso ou não.

Agora as alterações `AddNewProductView` pode ser feitas, conforme descrito no início dessa seção. Para começar, crie novas variáveis de estado para controlar essa tela, conforme o trecho a seguir:

```

@State private var isSaving = false
@State private var image = UIImage(systemName: "photo")!
@State private var showPhotoPicker = false

```

A primeira diz respeito ao controle de um alerta a ser exibido ao usuário, até que o produto como um todo seja salvo. A segunda vai cuidar de guardar a imagem a ser selecionada pelo usuário.

A terceira variável de estado irá controlar a exibição da tela para o usuário acessar a galeria de imagens do dispositivo.

Agora crie mais duas seções no componente `Form` para exibir o botão para o usuário escolher a foto e o local onde ela será exibida, como no trecho a seguir:

```
Section() {
    Button("Select the photo") {
        showPhotoPicker.toggle()
    }
}

Section() {
    Image.init(uiImage: image)
        .resizable()
        .aspectRatio(contentMode: .fit)
}
```

Veja que o trabalho do botão é apenas inverter o valor da variável de estado `showPhotoPicker`, que irá controlar um `sheet` que será criado mais adiante.

A segunda sessão exibe uma imagem, que terá ser o conteúdo controlado pela variável de estado `image`.

Agora crie o `sheet` que exibirá a galeria de imagens para o usuário escolher a foto, logo abaixo do `alert` criado nessa `view`:

```
.sheet(isPresented: $showPhotoPicker) {
    ProductPhotoPicker(showPhotoPicker: self.$showPhotoPicker) { image in
        DispatchQueue.main.async {
            self.image = image
        }
    }
}
```

Veja que função desse `sheet` é exibir `ProductPhotoPicker`, passando o *binding* da variável de estado `showPhotoPicker`, para controlar sua exibição, além do *closure*, que será invocado quando a imagem for selecionada, que por sua vez altera a variável de estado `image`, para exibir a imagem selecionada na tela de criação do produto.

Para que o usuário receba uma mensagem enquanto o produto está sendo salvo, crie um novo `alert`, logo abaixo do existente nessa `view`:

```
.alert(isPresented: $isSaving) {
    Alert(title: Text("New product"),
          message: Text("Saving product..."))
}
```

Como dito, ele será controlado pela variável de estado `isSaving`, que terá seu valor alterado pela função `saveProduct`, que deverá ser reformulada para se adaptar às alterações feitas no `viewModel`, como mostra o trecho a seguir:

```
private func saveProduct() {
    self.showAlert = false
    self.isSaving = true
    self.addNewProductViewModel
        .saveNewProduct(imageData: image.jpegData(compressionQuality: 0.0)) {
            success in

                self.isSaving = false
                if success {
                    self.isPresented = false
                } else {
                    self.showAlert = true
                }
        }
}
```

Além das modificações para controlar as variáveis de estado mediante ao retorno assíncrono da chamada a função `saveNewProduct` do `viewModel`, é importante ressaltar aqui que a imagem está sendo comprimida antes de ser enviada para ser salva no Firebase Storage, como mostra a instrução `image.jpegData(compressionQuality: 0.0)`. Esse processo de compressão é necessário para diminuir ao máximo o tamanho do arquivo.

Agora a implementação até aqui pode ser testada! Para isso execute a aplicação, clique no botão + da tela principal, preencha os dados do produto e escolha uma foto para ele. Ao clicar-se no botão para salvá-lo um alerta deverá aparecer, até que todo o processo termine e o usuário seja conduzido de volta para a tela principal, que já deverá exibir o novo produto com sua foto em miniatura à esquerda.

13.9 - Alterando o produto com sua foto

As tarefas a serem realizadas para alterar o produto com sua foto são bem parecidas com as que foram feitas na seção passada, com as seguintes diferenças:

- Ao alterar o produto, deve-se verificar se a foto foi de fato trocada, para fazer o upload da nova imagem;
- Antes de fazer o upload da nova foto, é necessário excluir o antigo arquivo e criar uma identificação única para a nova imagem;
- O componente para exibir a imagem na tela de edição deve ser capaz de mostrar a foto existente do produto, bem como a nova escolhida pelo usuário.

Dito isso, abra a classe `EditProductViewModel` e comece criando um novo atributo privado para representar a identificação única do produto, como no trecho a seguir:

```
private var imageUrl: String?
```

E obviamente, inicialize esse atributo no fim da função `setProduct`:

```
self.imageUrl = product.imageUrl
```

A seguir, crie a função para excluir a imagem do Firebase Storage:

```
private func deleteImage(imageId: String) {
    let storage = Storage.storage()
    let storageRef = storage.reference()

    let imageRef = storageRef
        .child("/users/\(Auth.auth().currentUser!.uid)/\("\(imageId)")

    imageRef.delete { error in
        if error != nil {
            print("Failed to delete the product image")
        } else {
            print("The product image has been deleted")
        }
    }
}
```

De posse da identificação única da imagem, o arquivo pode ser excluído do Firebase Storage a partir da referência montada a partir dela.

Agora crie a função para fazer o upload da imagem, como mostra o trecho a seguir:

```

private func saveProductImage(imageData: Data, imageId: String,
                             completion: @escaping (Bool) -> Void) {
    let storage = Storage.storage()
    let storageRef = storage.reference()

    let imageRef = storageRef
        .child("/users/\(Auth.auth().currentUser!.uid)/\(#imageId)")

    imageRef.putData(imageData, metadata: nil) { (metadata, error) in
        if metadata == nil {
            print("Failed to upload the image")
            completion(false)
        }

        print("The image has been uploaded")
        completion(true)
    }
}

```

Veja que o processo é semelhante ao que foi construído na seção passada para fazer o upload da imagem.

Da mesma forma como foi feito na seção passada, a função para salvar o documento no Firestore deverá ser reformulada para incluir um mecanismo de *callback* para informar quando o processo foi concluído, por isso crie a nova função a seguir para estar em conformidade com essa estratégia:

```

private func update(imageId: String?,
                   completion: @escaping (Bool) -> Void) {
    let product = Product(userId: Auth.auth().currentUser!.uid,
                           name: self.name,
                           description: self.description,
                           code: self.code,
                           price: self.price,
                           imageId: self.imageId)

    let productRef = db.collection(Product.COLLECTION).document(self.id!)

    do {
        try productRef.setData(from: product)
        print("The product has been updated - ID: \(productRef.documentID)")
        completion(true)
    } catch let error {
        print("Error while updating the product: \(error)")
    }
}

```

```
        completion(false)
    }
}
```

Aqui as alterações se restringem à inclusão da imagem do produto no documento a ser salvo e a chamada do *closure* para indicar o fim do processo com seu status.

Por fim, refaça a função `updateProduct` para primeiro fazer o upload da imagem no Firebase Storage, antes de salvar o documento no Firestore, como mostra o trecho a seguir:

```
func updateProduct(newPhoto: Bool, imageData: Data?,
                  completion: @escaping (Bool) -> Void) {
    if newPhoto, let imageData = imageData {
        if let imageId = self.imageId {
            deleteImage(imageId: imageId)
        }
        self.imageId = UUID().uuidString
        saveProductImage(imageData: imageData, imageId: self.imageId!) {
            success in

            if success {
                self.update(imageId: self.imageId) { result in
                    completion(result)
                }
            } else {
                completion(false)
            }
        }
    } else {
        update(imageId: self.imageId) { result in
            completion(result)
        }
    }
}
```

A estratégia básica permanece a mesma utilizada na função para salvar o produto da seção passada, mas aqui só é feito o upload da imagem se o parâmetro `newPhoto` indicar que de fato uma nova imagem foi escolhida pelo usuário.

Agora é necessário realizar as alterações na `view` de edição do produto, por isso abra a `struct EditProductView` e comece adicionando as mesmas variáveis de estado detalhadas na seção anterior:

```
@State private var isSaving = false
@State private var image = UIImageView(image: UIImage(systemName: "photo"))
@State private var showPhotoPicker = false
@State private var newPhoto = false
```

A diferença aqui é que o atributo `image` agora é do tipo `UIImageView`, que é o componente capaz de carregar uma imagem do Firebase Storage, como será visto adiante.

Veja também que foi adicionada uma nova variável de estado para controlar se o usuário escolheu uma nova foto ou não. Isso é necessário para otimizar o mecanismo de gravação do novo produto, fazendo com que ele somente faça o upload se realmente existe um novo arquivo selecionado pelo usuário.

A seguir, adicione uma nova seção no componente `Form`, para exibir o botão para o usuário alterar a foto do produto:

```
Section() {
    Button("Change the photo") {
        showPhotoPicker.toggle()
    }
}
```

Agora acrescente a seção para exibir a imagem do produto, se ele já existir no Firebase Storage, como mostra o trecho a seguir:

```
Section() {
    Image(uiImage: image.image!)
        .resizable()
        .aspectRatio(contentMode: .fit)
        .onAppear() {
            if let imgId = self.product.imageId {
                let storage = Storage.storage()
                let storageRef = storage.reference()

                let imageRef = storageRef
                    .child("/users/\(Auth.auth().currentUser!.uid)/\("\(imgId)"))

                self.image.sd_setImage(with: imageRef,
                    placeholderImage: UIImage(systemName: "photo")) {
                    (image, error, cacheType, storageRef) in

                if let error = error {
                    print("Error loading image: \(error)")
                }
            }
        }
}
```

```
        }
    }
}
}
```

A grande diferença aqui é que, quando esse componente `Image` aparecer, ele deverá buscar a imagem do produto, através de sua identificação única, criando a referência para ser passada para a função `sd_setImage`, que cuidará de buscar o arquivo, primeiro no *cache*, se existir, e em seguida no próprio Firebase Storage, cuidando para exibir um imagem de fundo até que a foto seja efetivamente baixada.

Agora crie o alerta e o sheet para mostrar o `ProductPhotoPicker` para que o usuário selecione uma nova foto, da mesma forma como foi feito na tela de criação do produto na sessão passada:

```
.alert(isPresented: $isSaving) {
    Alert(title: Text("Edit product"),
          message: Text("Saving product..."))
}

.sheet(isPresented: $showPhotoPicker) {
    ProductPhotoPicker(showPhotoPicker: self.$showPhotoPicker) { image in
        self.newPhoto = true
        DispatchQueue.main.async {
            self.image = UIImageView(image: image)
        }
    }
}
```

Por fim, adapte a função `updateProduct` já existente para cuidar das novas variáveis de estado, além de receber o evento informando o fim do processo de upload da foto no Firebase Storage e gravação do documento no Firestore, como mostra o trecho a seguir:

```
private func updateProduct() {
    showAlert = false
    self.isSaving = true
    editProductViewModel.updateProduct(newPhoto: self.newPhoto,
                                         imageData: image.image!.jpegData(compressionQuality: 0.0)) {
        success in

        self.isSaving = false
        if success {
            self.presentationMode.wrappedValue.dismiss()
        } else {
            showAlert = true
        }
    }
}
```

```
    }  
}
```

Agora toda a aplicação pode ser testada! Para isso execute a aplicação e clique sobre um produto existente. A tela de edição deverá mostrar a foto do produto que já foi cadastrado. Agora altere essa e salve o produto. O usuário deverá ser navegado de volta para a tela principal, após o upload da nova foto. Por sua vez, a tela principal já deverá a foto alterada para o produto que foi editado.

Um teste interessante de ser feito é executar a aplicação em um dispositivo real, cadastrar alguns produtos com fotos e em seguida desligar a Internet desse dispositivo. Ao abri-se a aplicação novamente, será possível continuar vendo as fotos dos produtos, pois elas estão no *cache* que foi configurado justamente para isso.

13.10 - Conclusão

Essa capítulo mostrou como pode ser fácil integrar o Firebase Storage em uma aplicação com SwiftUI para agregar fotos aos produtos cadastrados no Firestore, mostrando como é possível integrar múltiplos serviços do Firebase, como o Firebase Authentication, que foi utilizado para definir as regras de acesso ao Storage.

Também foi mostrado como a biblioteca do FirebaseUI auxilia e trabalha de forma integrada com o SwiftUI com componentes que buscam fotos no Firebase Storage, utilizando um mecanismo de *cache* de fácil utilização e configuração.