Aprofundando em Flutter

Desenvolva aplicações Dart com Widgets

Casa do Código

EVERTON COIMBRA DE ARAÚJO

Sumário

ISBN

Impresso: 978-65-86110-75-3

Digital: 978-65-86110-76-0

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Este é um espaço que considero muito importante em meus livros. É o momento para reconhecer o apoio direto ou indireto de pessoas fundamentais para mim durante o processo de pesquisa e escrita do livro. Se você já leu outros livros meus, certamente achará esta introdução parecida, mas aqui, nesta seção, sou assim.

Não posso jamais deixar de agradecer ao apoio de meus filhos, pessoas-chave em minha vida. No momento em que me encontro, pessoal e profissionalmente, duas pessoinhas são muito importantes para os momentos de relaxamento: Maria Clara e Vicente Dirceu. Meus caçulas, minhas joias mais preciosas. Meu filho Gabriel, conversamos quase todos os dias e, na maioria das vezes, é você quem me dá conselhos. Comecei a escrever este livro depois de um desafio do Biel, que está sempre me motivando a desbravar. Obrigado, meus amados filhos.

Um agradecimento muito especial, mais como uma dedicatória, é para meu pai, Élio Pedroso Araújo, e minha irmã, Alba Lucínia Coimbra de Araújo. Que Deus ilumine muito vocês.

Um hiato pessoal, me permitam. Conheci a dança no período de escrita deste livro. Vocês não sabem como essa atividade faz bem para a vida, para a mente e o poder que ela tem de nos trazer felicidade. Se não sabem dançar, aprendam. Se sabem dançar, dancem mais. Bolero, tango, samba, forró ou o gênero que você gostar. Na dança, no meu caso, conheci pessoas e grupos maravilhosos, do bem, todos dispostos a ajudar e a ensinar. Além de vivermos momentos espetaculares que são perpetuados com essa atividade, nos desenvolvemos, pois nos pontos de dificuldades, a dança relaxa a mente e as soluções chegam. Dance!

Voltando, em relação à revisão técnica, uma vez mais meu sincero e grande agradecimento à excelente pessoa, competentíssimo profissional: Leonan Fraga Leonardo, que mantém sempre seu LinkedIn atualizado. O Leonan tem sido um grande amigo, parceiro

de testes, que em muitos momentos tira minhas dúvidas sobre problemas que ele já resolveu. Hoje o cara está muito bem colocado em uma grande multinacional de TI. Focado em aplicações web e dispositivos móveis, ele está sempre disponível para trocar informações. No início da escrita do livro, comecei a pensar em quem convidar para escrever o prefácio. Comigo em várias atividades de revisão, escolhi o Leonan já antes de iniciar a escrita. A satisfação de ter meu livro prefaciado por esse cara, é imensurável. Obrigado, Leonan. Carinho, gratidão e respeito.

Embora este texto seja repetitivo, sua força só aumenta cada vez que o escrevo. Agradeço muito à Casa do Código, principalmente à Vivian Matsui, um anjo que sempre me acompanha nos trabalhos que desenvolvo, sempre paciente e procurando tempo para ler e contribuir com as revisões dos livros. Obrigado, Vivian e este agradecimento se estende a toda a equipe da editora.

De maneira geral, agradeço sempre aos meus colegas de trabalho do Departamento de Computação da UTFPR, campus Medianeira, meu paraíso profissional, pelo coleguismo e pelos ouvidos.

Não poderia concluir os agradecimentos sem fazê-los também aos meus alunos e alunas. Queridas pessoinhas que estão aos poucos se soltando mais e vendo que a cara feia não é de bravo, mas sim de alguém que deseja que tenham um futuro bom, respeitoso e com muito orgulho de ser programador.

Sobre o autor

Everton Coimbra de Araújo atua na área de ensino de linguagens de programação e desenvolvimento. É tecnólogo em processamento de dados pelo Centro de Ensino Superior de Foz do Iguaçu, possui mestrado em Ciência da Computação pela UFSC e doutorado pela UNIOESTE em Engenharia Agrícola, na área de Estatística Espacial. É professor da Universidade Tecnológica Federal do Paraná (UTFPR), campus Medianeira, onde leciona disciplinas no curso de Ciência da Computação e de mestrados. Já ministrou aulas de Algoritmos, Técnicas de Programação, Estrutura de Dados, Linguagens de Programação, Orientação a Objetos, Análise de Sistemas, UML, Java para Web, Java EE, Banco de Dados e .NET e nos últimos dois anos tem se dedicado de corpo e alma ao desenvolvimento mobile.

Possui experiência na área de Ciência da Computação, com ênfase em Análise e Desenvolvimento de Sistemas, e atua principalmente nos seguintes temas: Desenvolvimento Web com Java e .NET, Persistência de Objetos e agora em Dispositivos Móveis. O autor é palestrante em seminários de informática voltados para o meio acadêmico e empresarial.

Prefácio

O Dart e o Flutter podem parecer estranhos no começo para alguns iniciantes, principalmente por suas sintaxes. Porém, uma vez que a estranheza inicial se esvai, mesmo conteúdos complexos tornam-se simples. Criar componentes reutilizáveis pode ser tão simples quanto utilizar um componente já existente. Esse fator gera uma sensação de liberdade, que facilita o sentimento entusiástico ao trabalhar com tais tecnologias.

Quanto à obra em questão, a abstração do conhecimento fica mais evidente ao ler os conteúdos apresentados pelo professor Everton. Isso mostra que mesmo conteúdos complexos são simples de serem entendidos, pois o livro traz uma abordagem pragmática e foca realmente no que agrega valor, facilitando a abstração de conteúdos complexos. Além disso, a metodologia utilizada para explicar os conceitos, apresentando códigos incrementados gradativamente e explicando o conteúdo em pequenas partes, faz com que o aprendizado seja tão natural que dificilmente será necessário fazer uma releitura para entender algum conceito.

À medida que os capítulos são explorados, maior a vontade em dar continuidade na leitura e facilmente haverá dedicação e entrega para conhecer o próximo conteúdo. Os primeiros capítulos são introdutórios e fornecem uma base sólida para a sequência dos próximos conteúdos. Ainda no excelente trabalho feito pelo professor Everton, é possível notar o comprometimento dele em abordar conteúdos aprofundados e, ao mesmo tempo, relevantes para o mercado de trabalho.

É fascinante consumir o conteúdo abordado no livro e perceber que criar um componente para dispositivos móveis utilizando Flutter é tão simples a ponto de caber em menos de 30 páginas de conteúdo, explicados e exemplificados por meio de trechos de código e imagens.

Ademais, ao ler o livro, é possível notar a empatia do professor Everton ao abordar os conteúdos, pois parece que estamos em um diálogo com ele. Isso é notável quando são identificados pelo próprio autor momentos que possam gerar incertezas e, na linha seguinte, o eventual ponto de dúvida é esclarecido.

Por fim, ao final do livro o leitor ou leitora terá uma compreensão sobre boas práticas de desenvolvimento, aprenderá uma nova linguagem de programação que está em voga e com muito potencial, também se sentirá capaz de desenvolver aplicações móveis, tanto para Android quanto para iOS, e ainda poderá, com o conteúdo aprendido, criar componentes próprios.

Leonan Fraga Leonardo

Sobre o livro

Quando conheci o Flutter, depois de ter trabalhado com o Xamarin e Ionic, logo pensei: isso é algo diferente!

O Flutter fez com que eu voltasse a me apaixonar pela programação. Tive vontade de voltar a desenvolver aplicativos, não mais apenas no meio acadêmico, e, ao terminar este livro, já estava com diversos projetos para novos, que logo se concretizarão.

Este livro traz, na prática, o desenvolvimento de aplicações *cross-platform* com o Flutter, um framework de desenvolvimento de aplicativos para dispositivos móveis, com versões para desenvolvimento web, desktop e PWA. Desenvolver um aplicativo para ser publicado em dispositivos com plataformas diferentes (iOS e Android) é uma tarefa muito simples com o Flutter.

Os aplicativos, nesse apaixonante framework, são criados por meio de widgets, o kernel do Flutter. Em conjunto, temos a linguagem Dart, que possibilitou a criação do Flutter e nos permite utilizar, customizar e criar nossos widgets e aplicativos.

O livro é desenvolvido em 13 capítulos. O primeiro é apenas teórico, mas não menos importante, pois trago nele contextualizações sobre dispositivos móveis e as ferramentas usadas no livro por meio de um resgate histórico de linguagens e ferramentas utilizadas no desenvolvimento de aplicativos. Além disso, é nesse capítulo inicial que aponto a preparação para o ambiente que vamos utilizar.

No capítulo 2, é apresentado o Android Studio, ambiente que utilizaremos para o desenvolvimento do aplicativo proposto no livro, um jogo da forca. Utilizaremos o Android Studio, mas você pode ficar à vontade para utilizar o Visual Studio Code, outro IDE muito utilizado. Neste capítulo, esmiuçaremos o aplicativo criado como base pelo template do Flutter no Android Studio.

A aplicação implementada durante a leitura do livro refere-se, como mencionado, a um famoso jogo, o jogo da forca, ou *hangman game*. Registraremos palavras que serão utilizadas no jogo, teremos implementações de visões de abertura da aplicação, menus para navegação, entre outras opções disponíveis ao usuário. Utilizaremos recursos de persistência local e trabalharemos com gerência de estado, com setState, BLoC e MobX.

A prática começa no capítulo 3, no qual criaremos uma visão de splash screen e nosso primeiro widget. Aprenderemos também a inserir assets e imagens no nosso projeto.

No capítulo 4, criaremos uma tela com uma mensagem de boas-vindas, que o usuário poderá marcar como lida para que ela não apareça mais na abertura do aplicativo. Essa funcionalidade será trabalhada por meio da persistência de dados ditos pequenos com uma técnica chamada *Shared Preferences*. Veremos aqui novos componentes, uma prática que será comum em todos os capítulos.

No capítulo 5, já começaremos a dar ao app uma aparência legal criando um menu com opções para navegação entre as visões criadas no app. Usaremos um `Drawer`, componente comum em aplicações móveis clássicas, faremos várias customizações de componentes e conheceremos novos e importantes widgets.

O capítulo 6 começa a apresentar conhecimentos mais avançados. Customizaremos o `Drawer` padrão oferecido pelo Flutter e traremos uma animação que será controlada por BLoC, um excelente recurso para gerência de estado, o que evitará o `setState()`.

O uso de rotas, que auxiliará nossa navegação entre as visões de maneira nomeada, será apresentado no capítulo 7. Nele criaremos um formulário para que o usuário possa informar as palavras que devem ser registradas para o uso no jogo. Neste capítulo, não teremos a persistência dessas palavras, mas teremos a validação dos controles toda realizada por meio do BLoC. Conheceremos também a extensão simulando herança múltipla de comportamento

pelo uso de Mixin, um recurso interessante em algumas linguagens também trazido pelo Dart. Extensão, em Orientação a Objetos, é a herança aplicada a uma classe que estende comportamentos de outra. Um Mixin é um recurso do Dart, existente também em outras linguagens, que propicia que parte do comportamento de uma classe seja implementado nele, simulando assim uma herança múltipla por extensão, o que não é comum em linguagens Orientadas a Objetos.

A persistência das palavras informadas no capítulo anterior em banco é trabalhada nos capítulos 8 e 9. Neles faremos uso do SQLite, um mecanismo local para a persistência dos dados. Trabalharemos todas as operações do CRUD, registrando, atualizando, removendo e visualizando as palavras registradas. Veremos recursos bem interessantes relacionados à rolagem infinita de dados, com carga paginada da recuperação deles na base de dados, dando subsídio para um consumo de um serviço web, por exemplo. Veremos também a rolagem de um *ListView* em busca de um item específico do conjunto de dados utilizado e a marcação dele como selecionado para o caso de uma alteração. Tudo isso usando BLoC.

O capítulo 10 e 11, os mais esperados e muito divertidos, tratam do desenvolvimento do jogo da forca. Desenharemos do início a interface com o usuário e discutiremos regras do jogo para uma boa implementação. Traremos animações criadas em Flare e consumidas em apps Flutter. Conheceremos um novo componente e técnica para gestão de estado, o MobX, que trabalharemos integrado com o GetIt, uma implementação de Service Locator. Você certamente gostará deste capítulo.

O capítulo 12 traz observações e recursos para que nossa aplicação possa ficar ainda melhor e mais próxima do entregável de um cliente.

O livro termina no capítulo 13 com uma conclusão sobre o trabalho desenvolvido e um apontamento para estudos futuros.

Certamente, este livro pode ser usado como ferramenta em disciplinas que trabalham o desenvolvimento de dispositivos móveis, quer seja por acadêmicos ou professores. Ele é o resultado das experiências que venho acumulando ao ministrar aulas dessa disciplina. O que trago aqui são os anseios e as dúvidas dos meus alunos, para os quais já aplico esse conteúdo e do qual eles tanto gostam.

É importante que o leitor ou leitora tenha conhecimento de Orientação a Objetos, de alguma linguagem de programação, conhecimentos básicos sobre banco de dados e seria interessante também saber o conceito básico do Flutter, pois este não é um livro introdutório. Contudo, não ter esses conhecimentos não é um fator impeditivo. O repositório com todos os códigos-fontes usados no livro pode ser encontrado em https://github.com/evertonfoz/implementacoes-de-livros/tree/master/flutter.

Os arquivos disponibilizados no GitHub estão de acordo com as versões apontadas no livro. Recomendo que você implemente inicialmente os exemplos nessas versões e, após o sucesso, tente utilizar a versão mais recente, já que é possível que haja essa atualização quando você estiver lendo o livro. A equipe do Flutter é muito dinâmica e sempre está disponibilizando atualizações evolutivas e corretivas. Reforçando, fique atento às atualizações dos plugins e componentes dos projetos, pois essa tecnologia é dinâmica e atualizações estão sempre ocorrendo. Coloco-me sempre à disposição para esses casos via e-mail direto, evertoncoimbra@gmail.com.

Que a leitura deste livro seja para você tão prazerosa quanto foi para mim escrevê-lo. Desfrute sem moderação e espero que ao final você também esteja apaixonado pelo desenvolvimento mobile com Flutter. Sucesso.

Capítulo 1
Introdução

Tenho três livros sobre dispositivos móveis publicados pela Casa do Código. Nos três, comecei falando sobre a ascendência dessa tecnologia em nossas vidas, quer seja no campo pessoal, profissional ou acadêmico, em suas diversas ramificações.

Minha fraqueza sempre foi "desenhar" a interface com o usuário. Tudo era simples quando existia o Clipper (saudosismo). Depois, surgiram diversas linguagens e ambientes, onde a Microsoft tentou conquistar os desenvolvedores com o Visual Basic, mas quem ganhou a parada foi a saudosa Borland com o Delphi, que está ressurgindo com a Embarcadero. Foi uma época maravilhosa também.

Nesse ínterim, a web (www) ia crescendo, saindo apenas das páginas estáticas com HTML. O CSS ia evoluindo para embelezar as páginas e o JavaScript (fantástico) ia gerenciando o comportamento dessas páginas.

Assim como na migração de aplicações voltadas para console para o ambiente gráfico, o então Windows, a migração de aplicações em janelas para o ambiente web teve uma grande mudança de modelo, não só de desenvolvimento, mas envolvendo arquitetura, metodologias e ferramentas. Algo realmente grande.

Eu nunca me dei bem com CSS, não sou designer, mas o básico eu conseguia fazer. Mas ficou clara para mim a necessidade de outras formações de profissionais para a área de desenvolvimento.

A Orientação a Objetos já não era novidade nessa época, mas o procedural ainda era forte, mesmo com o Delphi que possibilitava o desenvolvimento com OO.

O Java começou a ganhar mercado! Mas o desenvolvimento nele, por mais que parecesse simples, não era para os que começavam, pois ele era OO e muito amplo. Tínhamos uma necessidade de plataformas disponíveis para dispositivos móveis, que foi um dos motivos para o surgimento do Java.

Eu me apeguei à plataforma e fiquei no back-end, era mais minha praia. Muitos frameworks surgiram para minimizar o esforço do desenvolvimento com Java. Todos tinham, no início, certos complicadores, mas a plataforma e a linguagem ganharam o mundo.

A Microsoft (lembra dela?) não estava morta na área de desenvolvimento. Até tentou emplacar sua versão do Java, mas não se deu bem. E eis que surgiu o .NET, vinha modesto, mas prometendo. Criou para seu ambiente de desenvolvimento toda a facilidade do arrastar e soltar das janelas. Hoje o .NET Core é uma coisa certa, estável, aberta e confiável. Tudo isso sem falar da linguagem C#, que também é muito brilhante.

Os dispositivos móveis vinham ganhando adeptos, usuários famintos por aplicativos. O Google chegou com o Android, que é Java, no qual todo mundo apostou e que continua aí até hoje, alguns dizem que com os dias contados. O Google tem mudado a linguagem do Android de Java para o Kotlin por ser uma linguagem moderna e estaticamente tipada, que possibilita um desenvolvimento mais rápido e que, segundo dados da própria Google, já é utilizada por 60% de profissionais que desenvolvem para Android.

Sempre foi muito difícil ou trabalhoso (para mim) desenvolver com Android. Não pela complexidade, mas por ter ferramentas de desenvolvimento pesadas, como o Android Studio, que graças ao IntelliJ, ficou mais leve e muito produtivo. Você comprovará isso neste livro, caso ainda não tenha utilizado esse IDE.

Muitos frameworks de desenvolvimento de aplicativos móveis surgiram, como o Xamarin, hoje da Microsoft, o Ionic, que começou

se baseando no Angular do Google e o React Native do Facebook. Alguns desses geravam aplicativos nativos nas plataformas em que seriam executados, o iOS e o Android. Outros utilizam o termo "híbrido", por se basearem em executores web, como se fossem uma máquina virtual onde as aplicações seriam executadas.

Alguns desses frameworks possibilitam inclusive que a aplicação gerada seja executada em navegadores ou ainda em ambientes desktop, pois algumas ferramentas de desenvolvimento são as velhas conhecidas HTML, CSS e JavaScript, também conhecidos em alguns ambientes como "Tecnologia Web".

No lugar do JavaScript, surgiu o maravilhoso TypeScript também da Microsoft. Ele permite que nosso código, que será convertido em JavaScript, seja compilado em tempo de implementação, com validações sintáticas, minimizando muitos problemas que o JavaScript trazia em relação à depuração.

Mas o livro não é de Flutter? Por que tudo isso? Pois é. O Flutter é maravilhoso. Você verá, se já não viu, que a curva de aprendizado dele é muito curta. Se temos conhecimento de Orientação a Objetos, é ainda mais fácil.

Para desenvolvermos aplicativos Flutter, precisamos utilizar uma nova linguagem, a Dart, que é simples, muito boa e não tão nova assim. Ela ficou modesta, na surdina. Ela lembra um pouco o Java, o C#, sem segredos para trabalharmos. Alguns entusiastas comentam nas redes que o Flutter e a Dart vieram para desbancar o Android e o Java, mas não aposto nisso. O mercado está aberto e as tecnologias são muito dinâmicas.

Você leu sobre o Flutter desbancar o Android? Só especulação, ok? Mas você sabia que as duas tecnologias são do Google? Os caras são bons.

Lembra que falei que sou péssimo no design visual de aplicações? Pois é. O Flutter me fez sentir que sou um pouco bom. De maneira bem simples, ele trouxe para seus widgets todo o poder do CSS que

é utilizado no HTML. E a interação e comportamento da aplicação? Tudo em Dart. O Flutter foi implementado em Dart, e é tudo aberto. Já falaremos sobre widgets.

Uma dica sobre o Flutter: ele está com versões estáveis para Desktop e Web e embora ainda não estejam liberadas, vale a pena conhecer.

1.1 O que são os widgets?

Como comentado anteriormente, os aplicativos desenvolvidos em Flutter são todos baseados em widgets. Mas o que são os widgets? Vamos às analogias.

Você consegue lembrar dos conceitos iniciais de OO, nos quais tudo, em uma análise de problema, é um objeto? Em Flutter, tudo o que você puder pensar que comporá sua interface com o usuário é um widget.

Esse conceito não é novo se compararmos um widget a um componente visual. Acredito que estejamos mais acostumados com aplicações baseadas em janelas. Nessas aplicações, temos a situação de que cada componente visual é um controle.

E o que esses controles, visuais ou não, devem possuir para estarem adequadamente alocados em nossas aplicações? Propriedades são a resposta. Características que definirão como serão exibidos. E alguns desses controles ainda possuem eventos, que capturam a interação do usuário com eles.

Conseguiu abstrair o que estou dizendo? Então, você já chegou à resposta. Os widgets são componentes, controles, visuais ou não, com propriedades e eventos, ou seja, a ideia não é nova. Mas por que estou dizendo, com tanto alvoroço, que o Flutter é maravilhoso?

O motivo da minha empolgação é que a configuração para a apresentação visual desses controles é muito simples. Não exigirá de você, programador ou programadora, uma experiência avançada de design para criar aplicações bonitas e eficazes. Note que não estou dizendo que um designer pode deixar sua aplicação mais bonita, ok? Mas certamente ele pode. :-)

1.2 O que veremos neste livro?

O objetivo principal deste livro não é ser uma ferramenta básica de aprendizado, mas sim um recurso de nível intermediário ou avançado para quem desenvolve em Flutter.

Entretanto, não quis escrever um livro que tivesse, como pré-requisito, a obrigatoriedade de você já conhecer o Flutter. Com isso, teremos alguns capítulos iniciais, que darão a quem está começando a oportunidade de conseguir executar todas as implementações que serão trabalhadas neste livro. Além disso, nos exemplos, procurei sempre comentar o que pode ser um pré-requisito.

Como minha expertise é voltada para aplicações comerciais, nossos projetos serão voltados principalmente para este fim. Não tenho a pretensão de desenvolver uma aplicação completa que resolverá um problema específico por completo, mas tenha certeza de que o que aprender aqui poderá ser aplicado, sem sombra de dúvidas, às suas aplicações.

Embora meu foco seja apps comerciais, a ideia de jogos básicos como ferramentas para o aprendizado me cativa e procurei trazer para os capítulos iniciais, que introduzirão o Flutter e Dart, o desenvolvimento de um jogo para adivinhar palavras, que talvez você conheça como jogo da forca.

1.3 Preparação de nosso ambiente de trabalho

Apesar de gostar muito do Visual Studio Code, por ser possível, por meio de plugins, desenvolvermos aplicações Flutter, utilizarei como ferramenta para nosso desenvolvimento o Android Studio, que está muito bom, bem produtivo e mais leve.

No momento da escrita deste livro, o Android Studio está em sua versão 3.6.2, o Flutter está na versão 1.12.13-hotfix.9 e a Dart está na 2.7.2.

No período de desenvolvimento deste material, o Flutter teve alguns updates, mas não quebrou o funcionamento de nada, o que me fez perceber a estabilidade da plataforma e seriedade da equipe responsável por ele.

Entretanto, é importante saber que mudanças podem ocorrer e, se suas ferramentas forem diferentes desta versão, e problemas ocorrerem, ficarei feliz em auxiliar você e procurar manter o livro sempre atualizado.

Como dito, o nível deste livro não é básico, dessa maneira, não trouxe tutoriais de como instalar e preparar nosso ambiente, mas comentarei sobre os passos e recomendarei fontes oficiais que o auxiliarão nesse processo.

Toda a instalação no ambiente Windows foi tranquila. No Mac tive alguns problemas, mas bastou atualizar algumas ferramentas dos pré-requisitos e tudo foi resolvido.

Para a instalação no ambiente Windows, acesse https://flutter.dev/docs/get-started/install/windows e leia com atenção toda a orientação desse site. A recomendação, ao baixar o arquivo compactado do Flutter, é a instalação em uma pasta chamada `/src/flutter`, na raiz de seu disco. Eu recomendo que seja apenas `/flutter`. Para mim, isso ficou mais semântico e fácil de seguir. Essa mesma recomendação eu faço para o Mac.

Fique atento à atualização das variáveis de ambiente, necessárias para o perfeito funcionamento do Flutter, integrado ou não ao IDE. Se você encontrar dificuldades nesse procedimento para Windows, pode ler rapidamente o texto: https://professor-falken.com/pt/windows/como-configurar-la-ruta-y-las-variables-de-entorno-en-windows-10/.

Na sequência, é preciso executar uma ferramenta do Flutter que nos auxilia na identificação de problemas, inconsistências ou problemas que possam impedir o desenvolvimento de nossos apps. Essa ferramenta é o Flutter Doctor. Com exceção de ter o Visual Studio Code instalado e configurado em sua máquina, todos os demais requisitos deverão ser cumpridos. Ele é um bom orientador em casos de identificação de problemas.

Após a instalação correta do Flutter, a documentação apresenta a instalação do Android Studio, que recomendo fortemente. Também é orientado na instalação de dispositivos virtuais, os emuladores. A dica que dou é que você instale um emulador em que tenha o Play Store habilitado. O link https://developer.android.com/studio/run/managing-avds.html?hl=pt-br poderá lhe orientar caso tenha dúvidas sobre esse ponto. Recomendo a criação de emuladores com as três últimas versões do Android.

Caso você utilize um Mac, o link para orientações é https://flutter.dev/docs/get-started/install/macos. Nele, você encontrará os mesmos pontos trabalhados anteriormente, com alguns requisitos a mais, como o XCode. Você verá também neste documento, como executar uma aplicação em um dispositivo físico, que, para o iOS, é um pouco mais rigoroso o processo, em relação ao que temos para o Android.

Com a leitura dos dois documentos oficiais apontados anteriormente, já temos nosso ambiente preparado e é possível que você já tenha criado sua primeira aplicação e a executado se seguiu as orientações até o final.

Conclusão

Fechamos nosso primeiro capítulo. Tivemos nele um resgate sobre ferramentas de desenvolvimento e uma breve introdução ao Flutter. Concluímos com orientações para a preparação do nosso ambiente de trabalho. Foi um capítulo leve com o objetivo de aguçar seu interesse para o que virá. Nem tome água, vá direto para o segundo capítulo. Fortes emoções nos aguardam.

CAPÍTULO 2
Ambientando-se com o Flutter

Com o intuito de ser um capítulo nivelador para aqueles que ainda não implementaram em Flutter, veremos o essencial sobre o framework e o desenvolvimento com o Android Studio. Se você é um leitor ou leitora com pouca bagagem em Flutter, aprenderá alguns conceitos básicos, caso contrário, utilize este capítulo para relembrar alguns conceitos.

Criaremos aqui um app desde o início e para isso trabalharemos com contêineres e outros widgets de conteúdo.

2.1 Instalação e configuração do Flutter e do Dart no Android Studio

O Android Studio suporta o desenvolvimento de aplicações Dart, que é a linguagem utilizada para desenvolvermos aplicações Flutter. Entretanto, o IDE não vem por definição com esse recurso habilitado. Precisamos instalar alguns plugins.

Se você estiver com a janela de boas-vindas do Android Studio aberta, na parte inferior dessa janela, em `Configure`, escolha `Settings`. Se o IDE já estiver aberto com algum projeto, você pode acessar `Settings` diretamente no menu `File`.

Com a janela de `Settings` aberta, clique em `Plugins` ao lado esquerdo dela. Você verá uma relação de plugins. No topo dessa janela, tem uma opção chamada `Marketplace`. Clique nela. Na caixa de digitação de busca por plugins, digite `Flutter`. Quando o plugin for exibido, instale-o. Por padrão, ele já solicitará a instalação do Dart, mas se isso não ocorrer, procure-o e instale-o.

Após a instalação, vamos fazer algumas configurações para o uso do Flutter e Dart no Android Studio. Ainda na janela de `Settings`, clique em `Languages & Frameworks`, depois em `Flutter` e verifique as configurações em destaque na figura a seguir.

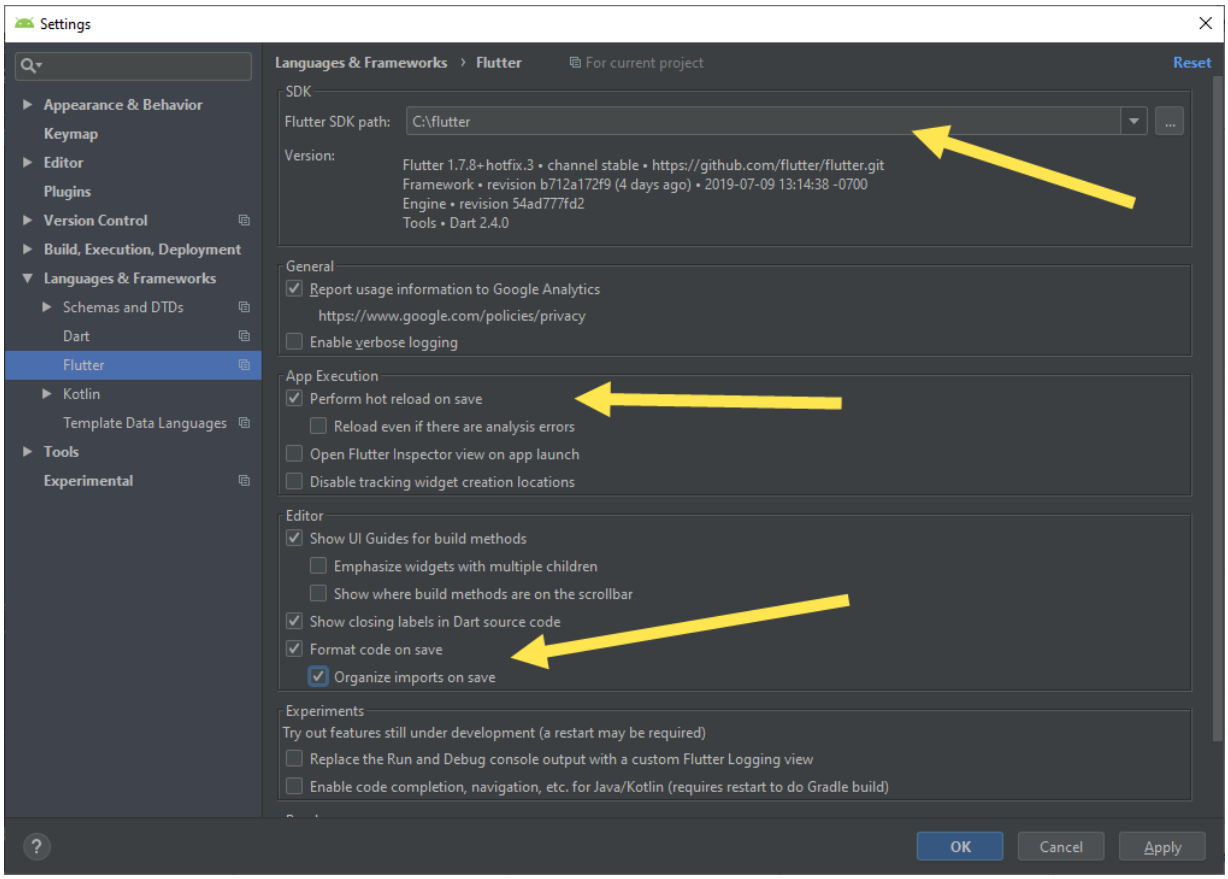

Figura 2.1: Configuração do plugin do Flutter

Os pontos destacados na figura anterior permitem a configuração de todo nosso código ao salvarmos o arquivo em edição. Esse é um recurso maravilhoso e você verá isso. Também configuramos o Hot Reload automático após a gravação. Isso faz com que mudanças realizadas nos widgets sejam visíveis, na maioria das vezes, de imediato.

Também temos a informação da localização de nosso Flutter SDK. Se não estiver correto, configure-o indicando o caminho para ele. O Flutter SDK é aquele que você baixou e descompactou em sua

máquina. Se seguiu as orientações, ele está em `\flutter` no root de seu disco.

2.2 O App criado pelo template do Android Studio

Com o Android Studio aberto em sua tela de boas-vindas, selecione a opção `Start a new Flutter project`. Caso você esteja com o IDE já aberto, selecione o menu `File -> New -> New Flutter Project`. Na janela que se abre, escolha o template `Flutter Application` e pressione o botão `Next`.

Informe o nome de seu projeto, eu utilizei `cc_02`. Confirme a localização do Flutter SDK, que instalamos no capítulo anterior. Selecione um caminho para seu projeto. Caso você queira, coloque uma descrição para ele. Com tudo informado, clique no botão `Next`.

Na nova etapa, informe o domínio para a companhia. Eu coloquei `casadocodigoflutter.com.br`, o que levou ao nome `br.com.casadocodigoflutter.cc02` para o package. Essa é uma regra para identificação das aplicações em dispositivos móveis, conhecida como domínio invertido. Mantenha checadas as opções sobre `androidx`, `kotlin` e `swift`. Com tudo isso feito, pressione o botão `Finish` e nosso projeto será criado.

A figura a seguir traz o IDE com nosso projeto criado e, em seguida, há explicações sobre os destaques dessa figura. Note que o arquivo aberto chama `main.dart` e se encontra na pasta `lib`. Logo falaremos bastante sobre isso.

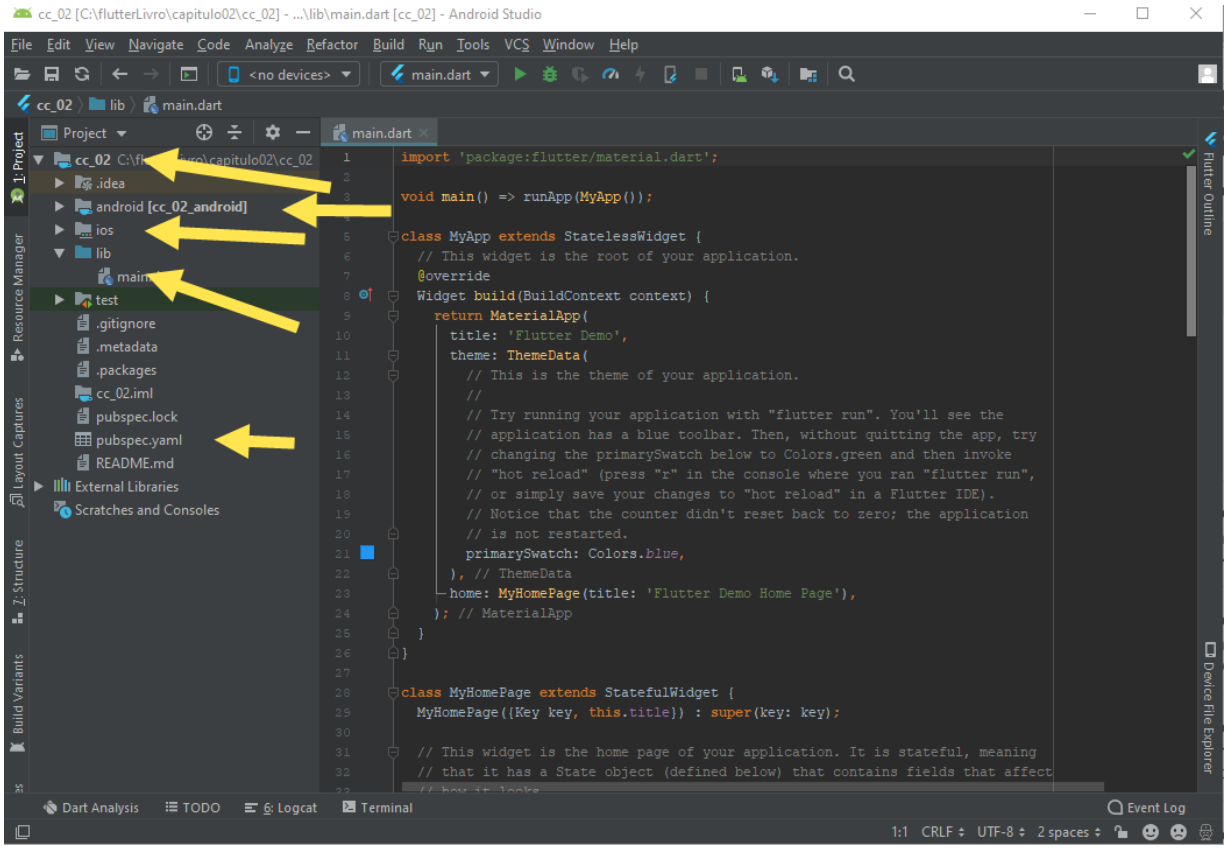

Figura 2.2: Ambiente com nosso primeiro projeto criado

Na área à direita da figura, ou área central, temos o editor de códigos do Android Studio. É nesse editor que codificaremos nosso app. Já do lado esquerdo, temos um browser para os artefatos do nosso projeto, pastas onde estarão as aplicações em Android e iOS, a `lib`, que é onde codificaremos nossos arquivos em Dart, e um arquivo especial para a configuração do nosso projeto, o `pubspec.yaml`. O template criará uma pasta chamada `test`. Podemos removê-la, pois não trabalharemos com testes neste livro.

Muito bem, com nosso projeto criado, podemos executá-lo, mas antes precisamos ter um emulador iniciado ou uma conexão com um dispositivo físico, via USB. Vamos dar uma olhadinha na figura a seguir.

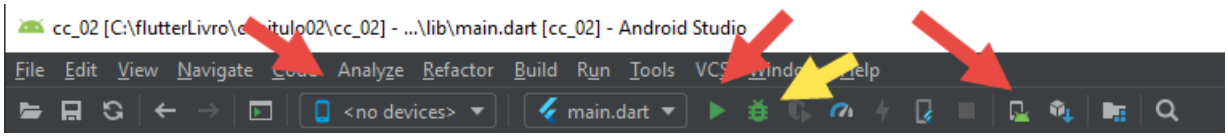

Figura 2.3: Passos para a execução da aplicação

Se você estiver em um Mac e quiser um emulador de um dispositivo iOS, basta clicar na primeira opção sinalizada na figura anterior e então escolher a opção de iOS Simulator. Outra maneira para isso é acessar o Xcode e, nele, selecionar o emulador desejado. Vimos um pouco disso no capítulo anterior em um link oficial para essa atividade, mas o processo é iniciar o Xcode e então o menu `Xcode -> Open Developer Tool -> Simulator`, e escolher seu simulador. Uma vez iniciado, o simulador será exibido como opção no primeiro destaque da figura anterior.

Se seu objetivo for executar a aplicação em um emulador Android, o processo de seleção do emulador é o mesmo do iOS. O que temos de diferente é a inicialização do emulador. Ela deve ser realizada por meio do AVD Manager, acessível pela quarta indicação da figura anterior. Nessa janela, aparecerão os emuladores disponíveis em sua máquina. Note que aqui falamos de Mac ou Windows (em nosso caso aqui no livro). Se você ainda não tem emuladores, pode retomar o link da documentação oficial, que vimos no capítulo anterior, mas que trago aqui para facilitar: https://developer.android.com/studio/run/managing-avds.html?hl=pt-br.

Com os emuladores ou dispositivos selecionados na barra de tarefas exibida na figura anterior, já podemos executar nossa aplicação clicando no segundo botão destacado na figura anterior. Também podemos rodar em modo de `debug`, ou seja, depuração, onde poderemos inserir *breakpoints* e executar nossa aplicação passo a passo para identificar pontos com erros. Sempre digo aos meus alunos que um programador não é bom o suficiente se ele não souber depurar. Então, saiba que precisamos dominar essa técnica

e os recursos do Android Studio, que realmente são bons e nos auxiliam muito nesse processo.

Para facilitar, caso não esteja na frente de seu equipamento, trago o código gerado para o arquivo `main.dart` na sequência. Logo começaremos a conversar sobre ele. Tirei os comentários gerados pelo template, para que o código fique mais limpo. A ideia é aguçar a curiosidade de quem está começando e, por meio desse app, trazer o jogo à memória daqueles que já experimentaram a brincadeira.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;
```

```
  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
    );
  }
}
```

Muito bem, vamos então verificar essa aplicação em execução. Escolha executar ou depurar, neste momento não importa. Recomendo até o executar, pois será mais rápido.

Por falar em velocidade, você pode sentir que a primeira execução é demorada, mas não se preocupe com isso, pois existe todo um

processo de tradução do código Dart para as plataformas, como compilação, implantação e execução no dispositivo ou emulador. Depois, veremos, caso você ainda desconheça, que o tempo de desenvolvimento e de testes é muito pequeno com Flutter. Veja a figura a seguir, que traz a aplicação criada em execução. Logo a comentaremos.

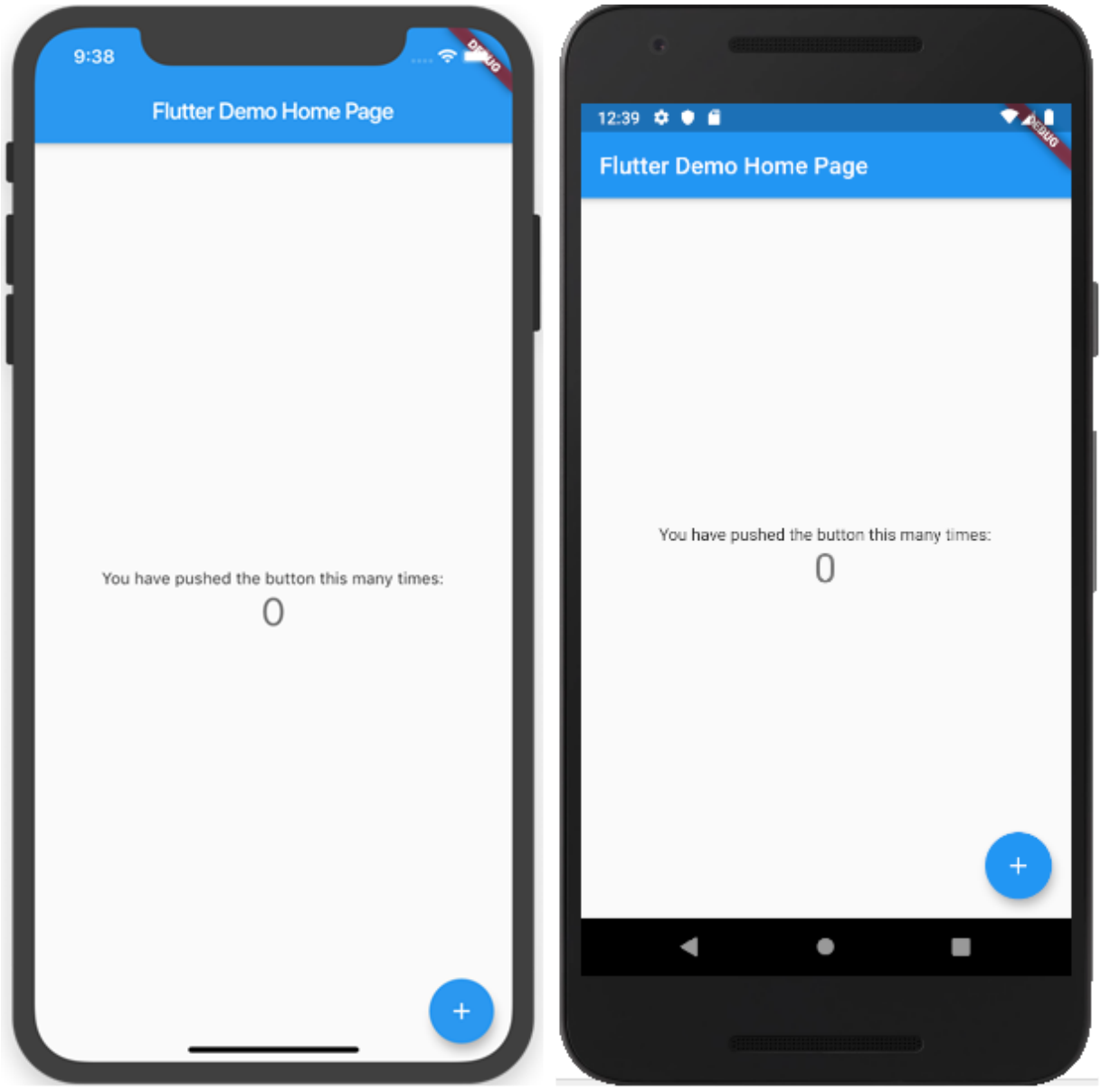

Figura 2.4: A aplicação do template em execução

O app que vemos na figura anterior é bem simples, mas poderemos ver várias características interessantes do que é desenvolver em Flutter. Caso você já tenha conhecimento sobre isso, por favor, entenda que estamos fazendo um flashback para os iniciantes.

Temos na `tela` uma String, que, com exceção ao valor exibido, é constante. Temos um botão, conhecido como `FloatingActionButton`, que, ao ser clicado, incrementará o valor exibido como quantidade de vezes em que foi clicado. Isso parece muito simples, muito óbvio e até não merecedor de nossa atenção. Entretanto, acredite. Isso tem sua complexidade.

Vamos a uma explanação, sem antes tocar no código que foi criado. Temos nossa tela com todos os componentes de um app para dispositivo móvel. Temos a barra de título, conhecida como `AppBar`, temos nossa área principal, que chamaremos de `body` e nosso `FloatingActionButton`. Tudo isso de acordo com os princípios do *Material Design*, princípios e componentes idealizados pelo Google. Se quiser, você pode dar uma olhada nisso em https://material.io/design/. O assunto é extenso, merecedor de um livro, por isso, não nos estenderemos nele, pois não é nosso foco principal.

O importante agora é sabermos que os componentes apontados anteriormente fazem parte de outro componente, maior na hierarquia, chamado `Scaffold`. O `Scaffold` é um componente que implementa a estrutura de layout básica do Material Design. Com o Flutter, isso fica transparente no iOS e Android.

Essa estrutura visual é toda criada pelo método `build()`, implementado na classe de estado `_MyHomePageState`, que é parte componente de `MyHomePage`, que é nosso `Widget`, ou seja, que representa o desenho de nossa tela. Isso mesmo, construímos widgets com outros widgets. Tudo pode ser visto como o processo de desenhar. Veremos tudo isso com calma nesta introdução.

Para que possamos ter um widget desenhado em uma tela do nosso dispositivo, precisamos que ele seja invocado e inserido em determinado ponto, normalmente, dentro de outro widget. Mas, temos apenas uma aplicação simples, com um único arquivo de código, o `main.dart`.

Verifique, logo no início desse arquivo, que temos a instrução `void main() => runApp(MyApp());`, que é a declaração de um método com a execução de uma única instrução. Essa grafia é conhecida como `Arrow Function`, mas há outras nomenclaturas que você poderá encontrar. O importante é saber que nossa aplicação será disparada pelo método `main()`, que está no arquivo `main.dart`. Quando esse método for invocado, ele executará o método `runApp`, que receberá como argumento, uma instância de `MyApp`, que é em si nossa aplicação.

Notou alguns termos, como `classe`, `método`, `instância`? Pois é. Isso é linguajar orientado a objetos, um pré-requisito importante para quem vai trabalhar com desenvolvimento Flutter. Embora eu tenha dito "pré-requisito", você pode continuar com a leitura e a prática mesmo que não tenha essa condição satisfeita, mas é muito importante que você estude isso, ok?

Eu ainda não falei o que essa aplicação faz, mas você ficou curioso em testar? A interface fluente, preocupada com a experiência do usuário, torna a interação tão fácil, que descobrimos sozinhos as funcionalidades e objetivos, não é mesmo? E o Flutter é perfeito para aplicar princípios de User Experience.

2.3 Vamos aprofundar no código de exemplo

Nesta seção, trabalharemos com detalhes todo o código gerado e apresentado na seção anterior. Sei que é clichê, mas vamos começar pelo início. Vou sempre apresentar o código e, em seguida,

as explicações e discussões sobre ele. Isso possibilitará que sua curiosidade o leve a ler e buscar interpretação do código apresentado, o que poderá nos ajudar na explanação. Veja então o código inicial na sequência.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());
```

Toda linguagem, mas toda mesmo, precisa de artefatos que chamamos de bibliotecas. Essas bibliotecas contêm códigos que minimizam nossos esforços na resolução dos problemas que temos. Isso promove uma reutilização muito boa.

Em Dart isso não é diferente. Isso mesmo: Dart. Lembre-se de que o Flutter é todo criado em Dart. Então, além de trabalharmos Flutter, trabalharemos Dart neste livro.

Sempre que formos trabalhar em uma aplicação Flutter, é muito provável que precisaremos ter incluída em nossos arquivos a biblioteca básica para nosso desenvolvimento, que é `material.dart`, importada para o escopo do arquivo `main.dart` na primeira instrução do arquivo. A título de curiosidade, você pode implementar tudo para iOS usando `cupertino.dart` e os componentes específicos para iOS.

Você pode fazer um teste agora, por curiosidade. Comente essa instrução, colocando `//` no início da linha. O Android Studio oferece um atalho legal para isso, que no Windows é a combinação de `CTRL + /` e no Mac é `Command + /`. Comentando a instrução, veja os erros que aparecem no editor, normalmente com riscos vermelhos abaixo de palavras (instruções) desconhecidas.

Esses erros também podem ser verificados na barra de rolagem vertical à direita do editor, um risquinho vermelho. Outra maneira de visualizar os erros e dicas de melhoria em código é clicando na guia `Dart Analysis`, que aparece bem na base esquerda da janela, mas

você pode visualizá-la pelo menu `View -> Tool Windows -> Dart Analysis`.

A segunda instrução da listagem anterior declara um método chamado `main()`, que é a porta de entrada para nossa aplicação. Temos uma função sendo invocada, a `runApp()`. Se você fez a atividade de comentar a primeira instrução, viu que ela ficou marcada como erro. Isso mesmo, ela está definida na biblioteca importada.

Podemos brincar de investigar um pouco as coisas. Coloque o cursor do mouse sobre a instrução `runApp()`, pressione `CTRL` e clique nela. Você será conduzido à implementação dessa função. Isso é muito bom. Notou que você tem acesso ao código-fonte de tudo que usa? Teste o mesmo com o `material.dart`. Investigue! Você verá que a investigação é muito importante para o nosso aprendizado.

Agora, voltaremos ao nosso código. Veja a implementação a seguir:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

No código da listagem anterior, podemos ver a definição de uma classe chamada `MyApp`. Utilizamos essa classe na execução do método `runApp()`, apresentado anteriormente. Ou seja, nossa aplicação é uma instância dessa classe que comentaremos agora.

O primeiro ponto é saber que essa classe é uma extensão de `StatelessWidget`, pois ela herda as características (métodos, propriedades) dessa classe. É a herança da OO. Mas como isso realmente afeta `MyApp`?

É preciso saber que essa extensão faz com que `MyApp` se transforme em um `Widget` que, como já dissemos, é algo que pode ser exibido na interface com o usuário. Mas o que é o `Stateless`?

Traduzindo literalmente do inglês, temos `sem estado`. No contexto do Flutter, significa que tudo que for exibido por esse widget não sofrerá alterações, que costumam ocorrer normalmente em decorrência da interação com o usuário. Dessa maneira, sempre que tivermos um widget que não sofrerá mudanças em seu estado, optamos pelo `StatelessWidget`. Veremos bem isso, em detalhes.

Como implementação da classe, temos apenas um método, o `build()`, que recebe `BuildContext`, que é injetado automaticamente. Primeira observação: veja o `@override` antes do método. Isso garante que estamos sobrescrevendo um método que existe na superclasse, em nosso caso, a `StatelessWidget`.

É no método `build()` que a mágica acontece. Ele `desenha` o widget e o retorna para que seja inserido em outro widget.

Neste primeiro momento, nosso widget utiliza a `MaterialApp()`, que representa uma aplicação que faz uso do `Material Design`. Essa classe renderizará o início de nossa aplicação. Observe que no `build()`, há o `return` da instância dessa classe. Todo widget é uma classe, isto é, um widget utilizado é um objeto.

É passado para o construtor da classe, nesse exemplo, valores para `title`, `theme` e `home`. O primeiro argumento, é claro, representa qual o título de nossa aplicação, que aparecerá quando você navegar nas aplicações abertas no dispositivo. O `theme` representa configurações de temas para a aplicação. O terceiro argumento, `home`, receberá o widget que será renderizado como página inicial

da nossa aplicação. Veja que é uma classe que também foi criada pelo template e que recebe um título.

O que acha de ser curioso e investigar o código de `MaterialApp` como fizemos anteriormente? Assim será possível conhecer e aprender todos os parâmetros possíveis de enviarmos para a classe. No Android Studio, se, ao abrirmos parênteses de uma classe ou método, pressionarmos `CTRL + Espaço`, podemos ver também a relação de parâmetros.

Em Dart, podemos codificar nossas classes para que nossos parâmetros sejam nomeados, o que facilita muito a implementação e o aproveitamento delas por desenvolvedores que venham a utilizar seu código.

2.4 Enfim, o widget do template

O que vimos até aqui foi a implementação e a execução do app e do widget Stateless que representará nossa aplicação. Mas o que realmente será desenhado no dispositivo?

Se você lembrar, no último código, atribuímos uma instância de `MyHomePage` para o parâmetro `home` de nosso `MaterialApp`. Vamos então rever esse código na sequência, agora em detalhe, mas já sabemos que `home` receberá um Widget (um objeto de `MyHomePage`).

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

O template criado pelo Android Studio traz alguns comentários sobre o que está sendo implementado, então procurarei trazer aqui esses detalhes. O primeiro ponto é notarmos que nosso widget, `MyHomePage`, estende `StatefulWidget`, o que significa que ele possuirá um objeto que manterá seu estado, um `State`.

A primeira instrução dentro da classe representa o construtor para nossa classe. Veja que ele possui dois argumentos opcionais, pois estão entre chaves. Um argumento opcional também é visto como nomeado, tal como comentado anteriormente. Quando instanciamos nossa classe em `home`, enviamos o valor para `title`, mas não para `key`, bem comum em todas as classes para identificar objetos registrados na árvore de widgets. É importante saber que remetemos esse argumento ao construtor da nossa superclasse por meio de `super(key: key)`.

A instrução seguinte define a variável `title` como sendo do tipo `String` e também como `final`. Uma variável definida como `final` permite apenas uma atribuição de valor e, sendo redundante, esse valor não poderá mais mudar após essa atribuição.

É comum uma confusão com variáveis do tipo `const`, conhecidas como constantes. A diferença básica, e muito importante, é que uma constante precisa que o valor a ser atribuído à variável seja conhecido, seja constante, definido. O que não ocorre com `final`, pois não sabemos que valor será atribuído a ela. Lembre-se disso!

É importante saber uma regra convencionada: campos (variáveis), na subclasse de um widget, são sempre `final`. Caso você desrespeite isso, mensagens de alerta e erros aparecerão em seu código e na janela `Dart Analysis`, apresentada anteriormente. Mas lembre-se de que é convenção.

Por fim, como última instrução, temos a invocação do método sobrescrito, `createState()`, que instancia a classe `_MyHomePageState`, que veremos na sequência.

Muito bem! Agora vamos à classe que será responsável pelo nosso widget, que será nossa primeira visão de interação com o usuário. Veja o código dela na sequência. Já o vimos anteriormente de forma rápida apenas para aguçar seu instinto de aprendizado, agora vamos detalhá-lo.

```
class _MyHomePageState extends State<MyHomePage> {
 int _counter = 0;

 void _incrementCounter() {
   setState(() {
     _counter++;
   });
 }

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           Text(
             'You have pushed the button this many times:',
           ),
           Text(
             '$_counter',
             style: Theme.of(context).textTheme.display1,
           ),
         ],
       ),
     ),
     floatingActionButton: FloatingActionButton(
       onPressed: _incrementCounter,
       tooltip: 'Increment',
       child: Icon(Icons.add),
     ),
```

```
    );
  }
}
```

Observe, logo na primeira instrução da listagem anterior, a definição da nossa classe, que estende `State`, especificando de maneira genérica a classe do nosso widget Stateful. Verifique que, como temos o Stateful e nele criamos o objeto que controlará o estado de nosso widget, precisamos dessa classe `State`, que possuirá os widgets que terão alteração em seus dados, nesse caso com base em uma interação com o usuário.

Logo após a definição da classe, temos a declaração de uma variável inteira nela, privada, chamada `_counter`, e sendo inicializada com `0`. Como sabemos que esta variável é privada? Pelo underscore (`_`). É essa a convenção adotada pelo Dart. Sendo assim, saberemos que essa variável é privada da classe e deve ter seu código alterado apenas por ela. É possível implementar métodos de leitura (`get`) e escrita (`set`).

Na sequência, temos um método também privado chamado `_incrementCounter()`. Ele não recebe nada e não devolve nada. Sua responsabilidade é apenas incrementar o valor de `_counter` em 1 a cada vez que for invocado. Ocorre que esse incremento está dentro de uma função anônima, atribuída como argumento ao método `setState()`.

Mas o que é uma função anônima? Essa ideia surgiu há muito tempo, e eu a conheci quando comecei com JavaScript, também há muito tempo. Temos funções nomeadas, que são as funções que podem ser chamadas por diversas partes do código, por isso precisam de um nome. Uma função anônima representa um conjunto de instruções que serão realizadas apenas em um determinado momento. Não haverá reutilização para essa situação.

Nós podemos declarar uma função anônima como `() {}`. Os parênteses são a função, sem nome. As chaves representam o conteúdo da função. No caso do `setState()`, apresentado

anteriormente e que trago na sequência para auxiliar, existe apenas o incremento de `_counter`.

```
setState(() {
  _counter++;
});
```

Mas o que é o `setState()`? Lembra que trabalhamos com um Stateful? E que temos essa classe como `State`? Então, teremos a mudança de valor em uma variável que é utilizada no desenho do widget e queremos que, quando esse valor for alterado, nosso widget seja comunicado de que precisa exibir essa mudança, que chamamos de alteração de estado. O `setState()` causa a reinvocação do método `build()`, que é o método que constrói nosso widget e o retorna para que seja desenhado.

Neste momento, você pode estar pensando: nossa, isso é lento, causará o redesenho de toda a interface. Calma, é tudo tranquilo. O redesenho ocorre, mas não é demorado e não pesa tanto. Só precisamos saber usar com parcimônia, é claro. No livro, veremos alternativas para isso, que inclusive deixarão nosso código mais limpo.

Até o momento, vimos que se formos alterar o conteúdo de alguma variável que precise ter seu valor atualizado na interface, precisamos fazer isso dentro do `setState()`, pois só ele causará a reinvocação de `build()`.

Já comentamos algumas vezes o que é o `build()` e para que ele serve, mas, como estamos trabalhando um nivelamento, vamos esmiuçá-lo um pouco mais aqui. Na assinatura do método, note que o retorno dele deve ser um `Widget`. Ou seja, tudo que possa ser desenhado na tela do dispositivo. Nosso retorno será um `Scaffold, também já mencionado e ao final do capítulo alguns links orientativos serão disponibilizados.

É importante sabermos que `Scaffold` é uma classe que gera um widget. Sendo assim, ao invocarmos `return Scaffold()`, estamos

instanciando essa classe e enviando a ela parâmetros para a criação e configuração do objeto de que precisamos.

Verificando atentamente nossa aplicação, temos uma área superior da interface, em azul, que é nossa `AppBar`. Na base, do lado direito, temos nosso `FloatingActionButton`, que poderia estar em outras posições por ser configurável. No centro, temos uma área maior toda em branco, que é o corpo, `body`, de nosso `Scaffold`. Podemos dizer que é a área de conteúdo do nosso app. Tudo isso é enviado como parâmetro para `Scaffold()`.

Agora, com o olhar mais focado, veja o código a seguir para o `build()` e perceba que estes parâmetros são, em várias situações, instâncias de outras classes, objetos, widgets. Neste pequeno código você verá diversos widgets que detalharemos mais à frente.

```
@override
Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
```

```
      ),
    );
}
```

2.5 Sobre os widgets do exemplo do template

Como dito na seção anterior, nosso widget representante da interface que será visualizada pelo usuário possui outros widgets. Alguns deles até já foram comentados por nós, mas os trarei aqui com um olhar mais detalhado.

AppBar

O primeiro é o `AppBar`. O `AppBar` é um widget que se posiciona no topo da página de um app e é registrado em um `Scaffold`. A figura a seguir, oficial do Flutter, retrata melhor do que em palavras o que é esse widget. Olhe a imagem atentamente, já comentaremos sobre ela.

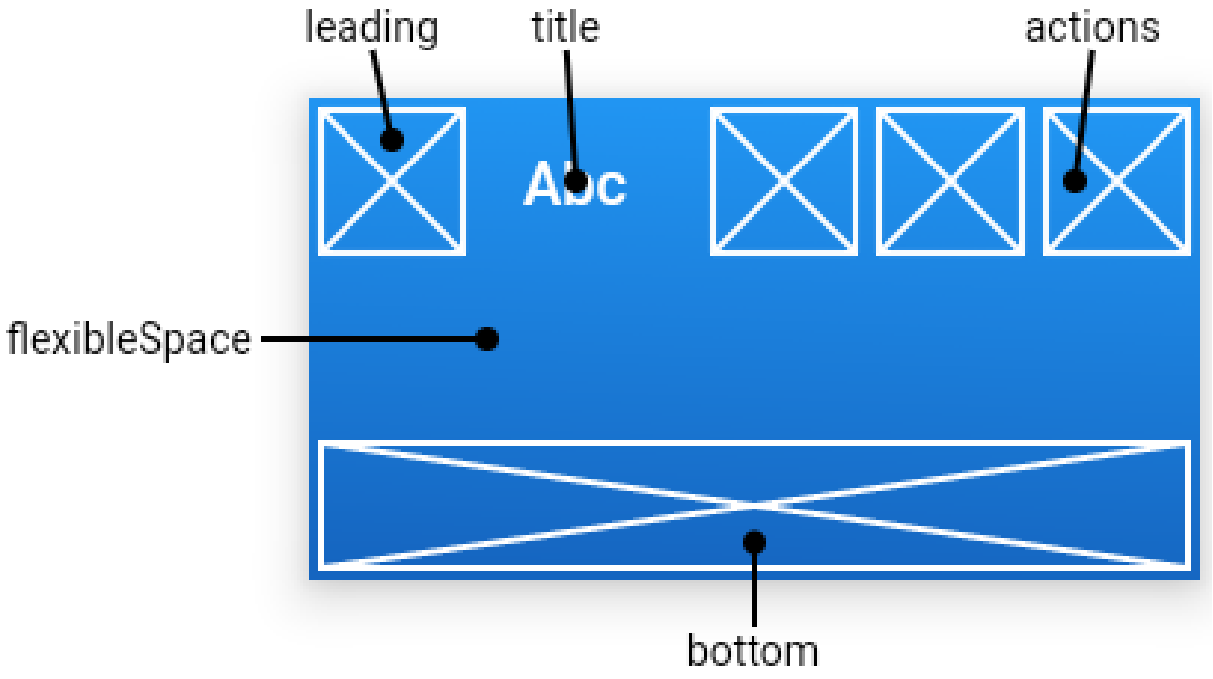

Figura 2.5: Layout do AppBar

No exemplo que desenvolvemos, nossa `AppBar` tem apenas o título (title) e, pela figura anterior, podemos ver uma área chamada `leading`. Essa área é reservada para a exibição de botões característicos de Apps Mobile. Em uma página que é chamada por outra, normalmente há o botão de retornar à tela anterior, conhecido como `Back button`. Outro botão comum é aquele para ativação de um menu e opções, conhecido como `Hamburger Menu`. Em aplicações Flutter, esse menu é um widget chamado `Drawer`.

Já à direita da figura anterior, temos uma área reservada a `actions`, que podem ser botões, textos, imagens, qualquer widget que realizará alguma funcionalidade para o nosso app.

A figura ainda traz uma área na base, chamada `bottom`. Ela estará ao final do `AppBar`, podendo ser qualquer widget, mas o comum é um `TabBar`.

Finalizando a explicação desse widget, temos uma área entre a `bottom` e a área superior, chamada de `flexibleSpace`. O widget

exibido nessa área ficará empilhado atrás do `bottom` e do `toolbar`.

Text

O widget `Text`, cuja função foi apresentada no exemplo, parece ser muito simples, muito básico, mas ele é muito poderoso, como todos os widgets. No decorrer do livro veremos várias de suas facetas. Aqui, no momento, falarei apenas o básico, deixando os detalhes para implementações que ainda realizaremos.

Vou sempre apontar a necessidade de pesquisa, não só para o Flutter, mas para tudo na vida. Dessa maneira, ao final do capítulo, teremos links para os recursos aqui trabalhados e, verificando a documentação para o construtor de todos os widgets, você poderá identificar tudo o que é possível enviar como argumento para recebermos um objeto configurado conforme nossa necessidade.

De início, podemos citar que podemos ter estilos específicos, alinhamentos, direção para a escrita, localização e diversas outras propriedades para cada `Text`.

Em nosso exemplo, temos as três situações a seguir de uso para o widget:

```
title: Text(widget.title)

Text('You have pushed the button this many times:',)

Text('$_counter',style: Theme.of(context).textTheme.display1,)
```

No primeiro uso, enviamos a variável `title` para o construtor na instanciação de nosso widget Stateful. Essa propriedade não existe em nossa classe `_MyHomePageState`, apenas na `MyHomePage`, por isso o uso de `widget.` antes do nome da variável, para indicar que essa variável pertence ao nosso widget. O exemplo é para deixar claro que podemos enviar uma variável que contenha o que deve ser exibido pelo widget.

O segundo exemplo é mais clássico, enviamos uma String constante para o construtor. Entretanto, veja que após o texto existe uma vírgula antes do fechamento dos parênteses. Sintaticamente, isso não é nada, pois não temos outro parâmetro sendo enviado. Porém, para o Android Studio, isso diz muito, pois fará com que ele formate nosso código de uma maneira mais legível. Esse é um recurso fantástico do IDE, mas para isso você deve ter realizado a configuração apontada no início do capítulo.

O terceiro exemplo traz um uso um pouco mais complexo, mas você verá que é tranquilo. Temos a exibição do conteúdo da variável `_counter`. Até aí tudo bem, mas o conteúdo dessa variável está dentro de uma expressão literal, uma String. Aí entra o caractere especial de interpolação de Strings, o `$`. Ainda, é mais comum e seguro que você faça o uso de chaves para a variável a ser interpolada, como `${_counter}`. O não uso, para uma variável simples, funciona tranquilamente, mas, se temos propriedades nas variáveis, precisamos das chaves.

Ainda no terceiro exemplo, tempos o envio de um `style`. Esse envio poderia ser feito pela configuração e um widget `TextStyle`, mas aqui estamos fazendo uso de `Themes`, algo um pouco mais abrangente e que comentarei mais adiante. Apenas saiba que capturamos o tema do contexto atual da aplicação, obtemos o `textTheme`, que representa uma configuração específica para `texts`, e então fazemos uso do estilo `display1`.

Se você quiser se aventurar em temas, um referencial oficial de rápida leitura pode ser obtido em https://flutter.dev/docs/cookbook/design/themes, mas trabalharemos isso em nossos exemplos.

Center

O Flutter oferece uma categoria de widgets chamada `Layouts`, que possui widgets que organizam outros widgets dentro de si. São vários esses widgets de Layout e, caso você esteja curioso, pode

acessar o endereço https://flutter.dev/docs/development/ui/widgets/layout e ver todos eles. Agora, neste ponto, nosso foco é no `Center`.

No código apresentado anteriormente, nosso `Center` contém outro widget, o `Column`, que também é um widget de Layout. Falaremos sobre ele na próxima subseção. O que precisamos saber agora é que o widget que for informado como `child` de `Center` será centralizado tanto verticalmente como horizontalmente dentro da área em que se encontra, que, em nosso caso, é o `body` de nosso `Scaffold`. Ou seja, toda a área de conteúdo da nossa tela.

Além de `child`, o construtor de `Center` aceita opcionalmente mais dois parâmetros: `widthFactor` e `heightFactor`. Quando valores são enviados para esses argumentos, significa que nosso widget `Center` terá sua altura e/ou largura influenciada pelo valor enviado. Como exemplo, se atribuímos 10 a uma (ou as duas) propriedades, significa que nosso `Center` terá 10 vezes a largura e/ou altura de nosso widget `child`.

Caso você queira se aprofundar no `Center`, um artigo pode ser lido em https://medium.com/@meysam.mahfouzi/center-widget-the-story-of-a-logo-8c0380bcdc45, mas, novamente, trabalharemos isso no livro.

Column

Como dito anteriormente, o `Column` é um widget de Layout, como o `Center`. Entretanto, uma característica básica que temos de diferença e que gerará alguma contextualização é que um `Center` possui apenas um `child` (filho), enquanto o `Column` possui vários, atribuídos à propriedade `children`.

Isso tem relação com algo que não comentamos na subseção anterior. A categoria de widgets de Layout se divide em dois tipos: os widgets de Layout com um único filho e os com muitos filhos, onde nesse caso se enquadra o `Column`. Veremos diversos outros

componentes de Layout das duas subcategorias durante o livro e saiba que são muitos widgets, o que é maravilhoso. Mas vamos ao `Column`.

Pense em como é uma coluna, sem se preocupar com Flutter ou tecnologia. Poderíamos dizer que é uma pilha de objetos, uns em cima dos outros. Tente abstrair isso. Vamos a alguns exemplos acadêmicos: cartas de baralho, caixas empilhadas. Agora, traga isso para uma tela de um dispositivo móvel, em um app. Visualize os componentes de uma aplicação verticalmente empilhados.

É fácil, não é? Um exemplo que temos na própria documentação do Flutter é uma série de widgets `Text` que aparecem empilhados. Agora, veja o código a seguir, que é um trecho específico do `Column` da aplicação que estamos desenvolvendo. Note a propriedade `children`, que recebe uma matriz de `Widget`, com a declaração `<Widget>[]`. Viu o uso de Generics com `<>`? Notou que temos dois `Text` dentro de `Column`? Já vimos a aplicação em execução e conseguimos identificar que temos um `Text` embaixo (ou em cima) de outro.

```
child: Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    Text(
      'You have pushed the button this many times:',
    ),
    Text(
      '$_counter',
      style: Theme.of(context).textTheme.display1,
    ),
  ],
),
```

Veremos na sequência a imagem do código anterior, que já conhecemos, mas com destaque para o `Column`. Você sabe que o conteúdo está centralizado, porque temos esse widget dentro de `Center`, correto? Mas essa centralização ocorre apenas na

horizontal e queremos que o conteúdo de `Column` fique centralizado verticalmente também, como vemos na figura a seguir. Podemos fazer isso configurando a propriedade `mainAxisAlignment` com o valor `MainAxisAlignment.center`.

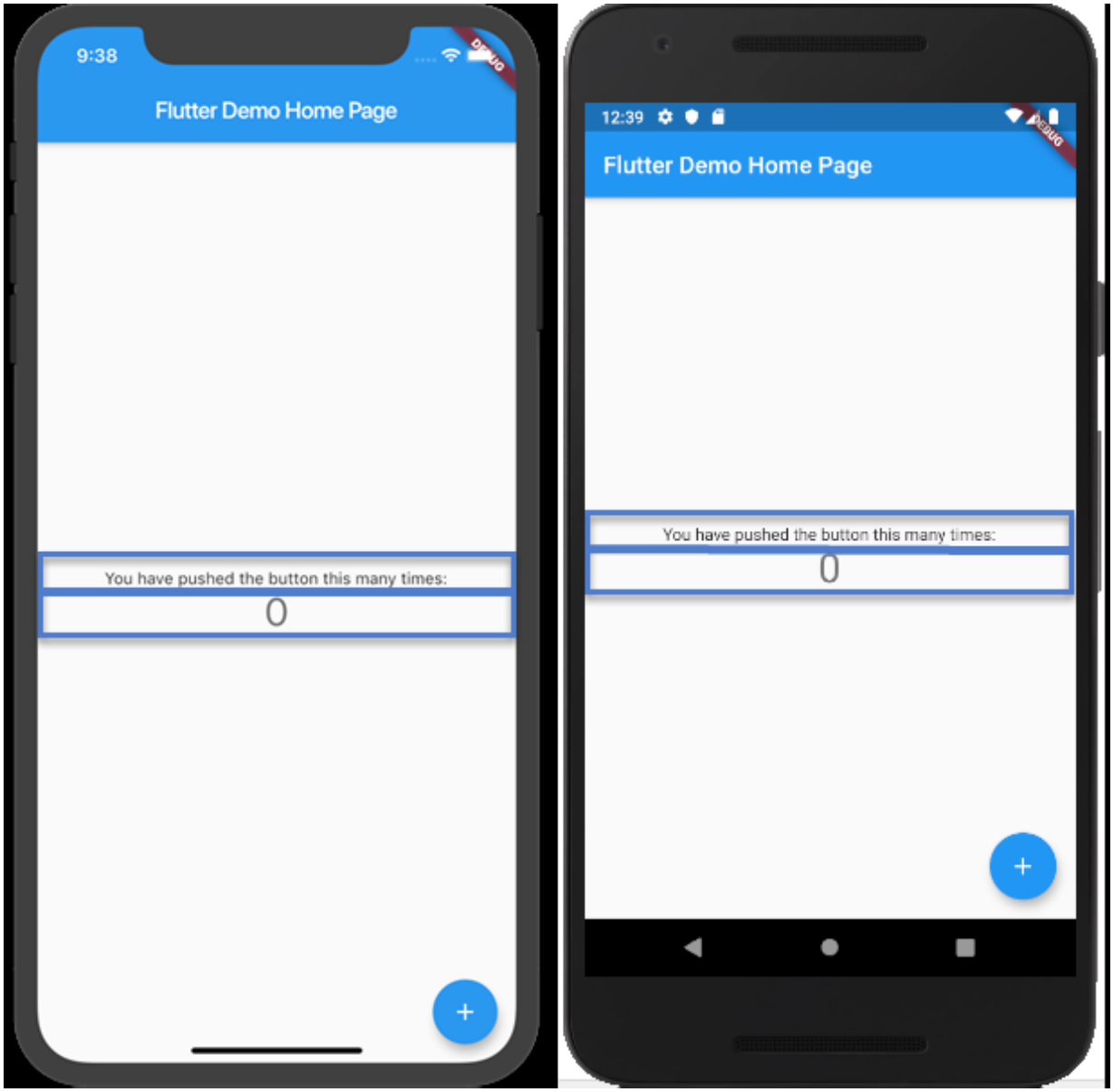

Figura 2.6: Column com children destacados

O que acha de testarmos o comentado no parágrafo anterior? Comente a declaração da propriedade e veja a mudança visual. Existem diversas outras propriedades e diversos outros valores para

essa propriedade em questão. Podemos testar essas alterações. No Android Studio, apague o valor `center`, pressione `CTRL+ESPAÇO` e veja outras opções. Procure testá-las, essa verificação visual é importante.

2.6 Hot Reload e Flutter Hot Restart

No início do capítulo, executamos nossa aplicação e vimos que o processo demora um pouco, por motivos justificados, pois existe toda uma tradução de código e preparação para a execução na plataforma selecionada. Mas para toda modificação que fizermos, precisaremos fazer tudo isso para testarmos? Não será lento e improdutivo?

Para essas situações, temos o *Hot Reload*, que processa as alterações e as aplica em nosso dispositivo ou emulador, mantendo o estado (State) dos objetos, ou seja, nossas implementações são aplicadas e as características visuais e os valores são mantidos. Isso é bem rápido.

Em casos em que precisamos descartar o estado de nossos objetos e começar nossa aplicação desde o início, temos o *Flutter Hot Restart*. Veja na figura a seguir onde temos um atalho para essas funcionalidades, mas elas também são acessíveis pelo menu `Run`. Temos a escolha da guia `Run` e, no topo, a primeira opção é a `Hot Reload` e a segunda, a `Flutter Hot Restart`.

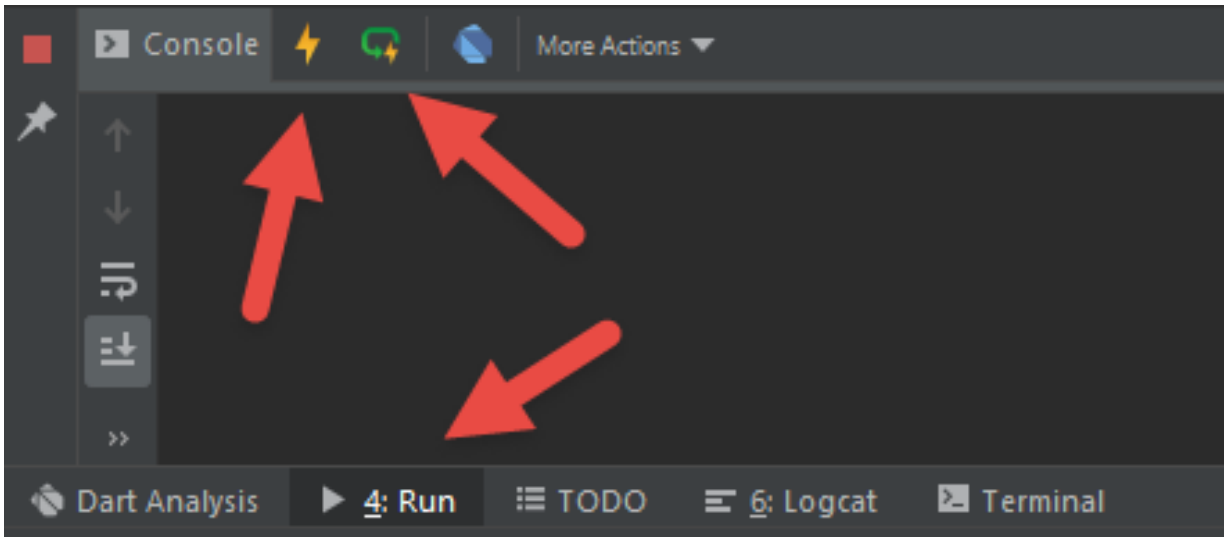

Figura 2.7: Acesso ao Hot Reload e Flutter Hot Restart

O que acha de testar as sugestões anteriores e utilizar o Hot Reload e Flutter Hot Restart para que as alterações sugeridas, que você certamente implementou, possam ser verificadas? Execute a aplicação, pressione o botão para aumentar a contagem e use uma das opções de execução da figura anterior.

FloatingActionButton

Já comentamos algumas vezes o widget `FloatingActionButton`. Ele é um componente definido inicialmente no `Material Design` e que, com o Flutter, podemos utilizar também em aplicações iOS.

Esse widget, conhecido também como `FAB`, é o que temos em azul (na imagem Column com children destacados vista anteriormente) com um símbolo de adição na base direita de nossa tela. Ele normalmente fica em uma camada acima dos widgets da visão em que se encontra. Para facilitar, vamos trazer o código específico dele na sequência.

```
floatingActionButton: FloatingActionButton(
    onPressed: _incrementCounter,
    tooltip: 'Increment',
```

```
    child: Icon(Icons.add),
),
```

Como todos os widgets, temos algumas propriedades que podemos personalizar. Em nosso exemplo, a primeira é a `onPressed`, que representará uma função que será invocada quando o usuário pressionar o FAB. Note que informamos apenas o nome do método e não sua invocação, que teria o abre e fecha parênteses. Temos essa função no código da nossa classe apresentada anteriormente.

A segunda propriedade é a `tooltip`, que contém um texto que será exibido ao usuário quando ele mantiver o widget pressionado por um período maior de tempo. A documentação diz que esse valor também é utilizado para acessibilidade e o Flutter é todo preparado para essa inclusão.

Nossa terceira propriedade é a `child`, que já pudemos verificar que é comum em praticamente todos os widgets, seja no singular ou plural. Para nosso `child` do FAB, temos um `Icon`, outro widget.

Relembrando, o `floatingActionButton` é uma propriedade de `Scaffold` que define uma área específica para os `FAB`. Isso mesmo. Podemos ter mais de um `FAB` disponível para o usuário em uma mesma interface.

Comentamos anteriormente a possibilidade de ter o FAB em outras posições da interface fora a padrão, e para o `Scaffold` temos a propriedade `floatingActionButtonLocation`, que pode ser utilizada para isso. Tente adicionar o código a seguir após a declaração do `floatingActionButton` e veja o resultado em seu dispositivo. Lembre-se do Hot Reload.

```
floatingActionButtonLocation : FloatingActionButtonLocation.centerFloat
```

Conclusão

Chegamos ao final deste primeiro capítulo prático, o segundo do livro. Caso você queira pesquisar futuramente de forma mais rápida

o que vimos aqui, deixarei a relação de cada ponto trabalhado (que foram apenas widgets): `MaterialApp`, `Scaffold`, `AppBar`, `Text`, `TextStyle`, `Center`, `Column`, `FloatingActionButton` e `FloatingActionButtonLocation`.

Fizemos um rápido nivelamento sobre aplicações em Flutter. Não criamos nada novo, mas revisamos bastante coisa com base no template criado pelo Android Studio. Passamos por alguns widgets, algumas características e recursos do Android Studio e terminamos com uma relação dos recursos vistos no capítulo.

O Flutter possui um enorme conjunto de widgets disponíveis para o desenvolvimento de aplicações para dispositivos móveis e veremos uma grande parte deles neste livro.

Dê uma relaxada agora, tome uma água e se prepare para o próximo capítulo, onde começaremos o processo para a criação de um app para um jogo bem tradicional, a forca.

CAPÍTULO 3
A Splash Screen na inicialização da aplicação

É comum verificarmos que os apps trazem uma tela de abertura, apresentando um logo da empresa ou da aplicação, talvez algum texto e possivelmente uma imagem animada, dando a ideia de que algo está sendo processado e preparado para que a aplicação seja executada. É nessa página, normalmente chamada de *splash screen*, que trabalharemos neste capítulo.

Importante deixar claro que um app possui dois momentos de inicialização. O primeiro, conhecido como *launch screen*, ocorre nativamente pela plataforma onde o app está instalado (veremos isso no capítulo 11). Aqui, a *splash screen*, segundo momento de inicialização, refere-se à sua aplicação já em execução, momento em que você pode implementar regras ou recursos que serão necessários durante a execução da aplicação.

O projeto que trabalharemos, a partir deste capítulo, será um só, mas você pode, se quiser, criar um aplicativo a cada capítulo, ou dar sequência, como eu farei. Para facilitar, os capítulos estarão no repositório do livro sempre com a implementação final deles.

3.1 Inicialização da aplicação

De acordo com o que vimos no capítulo anterior na criação do projeto básico oferecido pelo Flutter, vamos agora criar nosso projeto e chamá-lo de `forca`. Com ele criado, precisamos começar com implementações iniciais para o app antes de chegarmos à splash screen. A primeira intervenção que faremos é remover a pasta `test`, que se refere a testes unitários, pois não a utilizaremos em nosso livro. Se mantivermos essa pasta, mensagens

indesejadas podem aparecer na guia `Dart Analysis`, e não queremos isso desviando nossa atenção.

Em seguida, precisamos ajustar nosso `main.dart` para a necessidade que teremos em nosso projeto. Neste momento, teremos apenas uma visão com a cor verde sendo exibida. O código da listagem a seguir é simples, mas peço uma atenção para a definição do tema com uma cor de `background` e o uso desta cor no `Container` da página. Neste momento, não trarei imagens por julgar desnecessário.

```
import 'package:flutter/material.dart';

void main() => runApp(ForcaApp());

class ForcaApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Forca da UTFPR',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        backgroundColor: Colors.green,
      ),
      home: ForcaHomePage(),
    );
  }
}

class ForcaHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        color: Theme.of(context).backgroundColor,
      ),
    );
  }
}
```

Observou que temos dois widgets no `main.dart`? O primeiro refere-se à nossa aplicação em si e o segundo, à página inicial para a aplicação. O importante aqui é notarmos que os dois são Stateless (devido à herança/extensão de `StatelessWidget`), ou seja, nenhum controle dessa classe sofrerá alterações com base em interações com o usuário ou algum processamento.

É também comum a exibição de uma imagem com fundo transparente na splash screen, ou, ainda, o que está em moda, uma imagem em forma circular. Adotaremos aqui essa segunda opção, pois a aproveitaremos para criarmos nosso primeiro componente, que será responsável por renderizar uma imagem circular. Vamos componentizar essa funcionalidade prevendo uma possível reutilização.

3.2 Customização de um widget para uma imagem circular

Crie uma pasta chamada `widgets` na pasta `lib`. Se você é iniciante, isso pode ser feito clicando com o botão direito do mouse sobre a pasta `lib` e então escolhendo `New->Package`. Dê à pasta o nome `widgets`.

Nessa nova pasta criada, vamos implementar uma classe, que representará nosso primeiro widget customizado. A criação de uma classe se dá da mesma maneira que fizemos para a pasta, escolhendo apenas a opção `Dart File`. Atribua `circular_image_widget.dart` ao nome do arquivo. Não precisa informar a extensão, ok? Veja a implementação de nossa classe na listagem a seguir. Após a listagem teremos alguns comentários sobre ela.

```
import 'package:flutter/material.dart';

class CircularImageWidget extends StatelessWidget {
  final ImageProvider imageProvider;
```

```
  final double width;
  final double heigth;

  CircularImageWidget({
    @required this.imageProvider,
    this.width = 300,
    this.heigth = 300,
  });

  @override
  Widget build(BuildContext context) {
    return Container(
      height: this.heigth,
      width: this.width,
      decoration: BoxDecoration(
        border: Border.all(
          color: Colors.black,
          width: 5.0,
        ),
        shape: BoxShape.circle,
        image: DecorationImage(
          fit: BoxFit.cover,
          image: this.imageProvider,
        ),
      ),
    );
  }
}
```

O componente é simples, um `Container` decorado como um círculo, que possui uma imagem que será ajustada de maneira a cobrir toda a área do contêiner.

Relembrando, um `Container` é um componente que tem, dentro de si, outro componente. O Flutter traz uma série de controles desse tipo, que representam, como o nome diz, contêineres. No caso, `Container` é um contêiner de um único filho, mas esse filho pode ser outro contêiner.

Recomendo, caso queira, uma investigação sobre as opções para `BoxShape` e `BoxFit`. Mas, de maneira simplista, `BoxShape` definirá a forma como o contêiner decorado (`decoration: BoxDecoration()`) será renderizado, e `BoxFit` é como a imagem terá ocupação no contêiner.

Ainda no código anterior, verifique que temos três propriedades definidas para o nosso widget. Todas elas apontadas como opcionais no construtor para facilitar a implementação que apontará o nome do argumento a ser enviado ao construtor.

A `imageProvider`, embora esteja como opcional, é obrigatória e as referências ao tamanho da imagem, caso não sejam informadas, terão valores padrões atribuídos a elas.

Nesse componente, você poderia também pensar em deixar customizável pelo construtor a cor e a espessura da borda a serem atribuídas à imagem. Procure sempre pensar que valores fixos podem não ser interessantes em alguns casos.

O Flutter oferece componentes prontos para imagem em um círculo, como o `CircleAvatar`, que trabalharemos mais à frente. O objetivo aqui foi trazer para você a possibilidade de criar um componente desde o início.

3.3 O widget para a Splash Screen

Com o nosso widget para imagem circular implementado, vamos ao que representará nossa splash screen, o foco deste capítulo. Dessa maneira, na pasta `lib`, crie outra, chamada `routes` e, dentro dela, um arquivo `Dart` chamado `splash_screen_route.dart`. Veja a implementação para ele na sequência.

```
import 'package:flutter/material.dart';
import '../widgets/circular_image_widget.dart';

class SplashScreenRoute extends StatefulWidget {
```

```
  @override
  _SplashScreenRouteState createState() => _SplashScreenRouteState();
}

class _SplashScreenRouteState extends State<SplashScreenRoute> {
  @override
  void initState() {
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return Center(
      child: CircularImageWidget(
        imageProvider: AssetImage(
          'assets/images/splashscreen.png',
        ),
      ),
    );
  }
}
```

Observou inicialmente que agora temos um Stateful por estendermos a classe `StatefulWidget` por meio do `extends`? Por ser uma visão que pode ter seus componentes alterados por alguma interação com o usuário, ou em base por algum processamento interno, precisamos ter a possibilidade de controlar o `estado` desse nosso widget. Veja que temos uma segunda classe no código que estende `State`, tipificada por generics para `SplashScreenRoute`, que é nossa classe de widget.

Quando implementamos um `StatefulWidget`, precisamos ter controle de seu estado para que, em caso de alterações relacionadas à renderização de componentes pertencentes ao widget principal, elas possam ser refletidas na interface visualizada pelo usuário.

A princípio, temos a sobrescrita (`@override`) do método `initState()`, que faz parte do ciclo de vida de um `StatefulWidget`, mas não implementamos nada específico nele no momento. Temos também

a sobrescrita ao `build()`, responsável pela renderização do widget principal do `StatefulWidget` em questão.

Nesse método, estamos retornando o widget `Center`, responsável por centralizar, vertical e horizontalmente, o seu `child` em seu contêiner ancestral. Assim como `Container`, o `Center` é visto como um widget contêiner de outro widget, um único filho.

Como `child` de `Center`, temos nosso `CircularImageWidget`, que implementamos anteriormente. Observe que enviamos a ele, por meio do argumento `imageProvider`, um `AssetImage`, que se refere a um arquivo de imagem que devemos ter em nossa aplicação.

Aqui, em relação ao `AssetImage`, cabe um momento para algumas explicações. O Flutter traz um widget chamado `Image`, que representa uma imagem, não importando se ela é de um `asset` da aplicação, se ela é um arquivo físico armazenado em seu dispositivo, se ela tem a internet (rede) como origem, ou se vem da memória.

É comum utilizarmos construtores nomeados (ou de fábrica como a documentação diz) para instanciarmos uma imagem, como o caso de `Image.asset()`, que criará o `Image`, com uma imagem de nossa aplicação representando-o. Já o `AssetImage` é um provider, que é o responsável por obter uma imagem, tendo como base seu endereço (url).

3.4 Adicionando assets ao nosso aplicativo

Creio que você verificou que estamos utilizando uma figura como `asset` para ser renderizada em nossa splash screen, a `assets/images/splashscreen.png`, porém ainda não a temos em nosso projeto. Mas antes de resolvermos isso, o que é `Asset`?

A tradução literal para asset é `Ativo`. Algo que possuímos e que tem valor agregado ao contexto. Nossas aplicações podem ter assets, quer sejam imagens, ícones, fontes ou vídeos, que normalmente são distribuídos em conjunto com o app, pois são ativos necessários para a perfeita execução delas.

O Flutter trata os assets de maneira burocrática. Precisamos tê-los registrados em nossa aplicação para podermos utilizá-los.

Uma primeira burocracia, que não é regra, mas convenção, é termos uma pasta (directory) chamada `assets` na raiz de nosso projeto e, dentro dela, organizarmos os assets em categorias, como o caso de `images` e então, nessas pastas, gravarmos nossos arquivos.

O primeiro que teremos é o `splashscreen.png`, que para seu projeto pode ser qualquer imagem que você possua, a seleção fica a seu critério.

Com as pastas criadas e os arquivos devidamente armazenados, precisamos configurar a burocracia que é regra para o Flutter e isso é feito no arquivo `pubspec.yaml`. Este arquivo tem a responsabilidade de registrar algumas configurações para nossa aplicação Flutter, dê uma lida nele na sequência. Observe as últimas instruções referentes a `assets`.

```
name: capitulo04_splashscreen
description: A new Flutter application.

version: 1.0.0+1

environment:
  sdk: ">=2.1.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter

  cupertino_icons: ^0.1.2
```

```
dev_dependencies:
  flutter_test:
    sdk: flutter

flutter:
  uses-material-design: true

  assets:
    - assets/images/splashscreen.png
```

Na configuração dos assets, o espaçamento em branco, que serve como indentação, é uma sintaxe. Tem que ser dessa maneira, senão receberemos erros no processo de preparação para compilação do código.

A escrita de `assets:` começa com dois espaços em branco em relação à escrita de `flutter:`. Para cada `assets` e cada imagem, eles devem estar dois espaços à frente de `assets:`, começando sempre com hífen (`-`), seguido do caminho físico até a imagem. É possível registrar toda uma pasta em vez de um arquivo por vez. Essa é uma situação comum e produtiva, mas fica a seu critério como trabalhar em seus projetos. Apenas saiba que se optar por essa estratégia, é preciso terminar a linha com uma barra (`/`).

Para toda alteração realizada no `pubspec.yaml`, é preciso forçarmos o `Pub get`, que aparece no topo do editor quando estamos com esse arquivo aberto.

3.5 A Splash Screen em execução

Podemos pensar em executar nossa aplicação e verificar o funcionamento inicial dela, entretanto precisamos realizar uma mudança em nossa `ForcaHomePage`, que tem como `body` em seu `Scaffold` um `Container`. Queremos que agora seja exibida nossa

splash screen. É simples. Basta alterar o código para o que temos na sequência, ficando sempre atento aos imports necessários.

```
class ForcaHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: SplashScreenRoute(),
    );
  }
}
```

Com isso, podemos executar nossa aplicação e você pode ver sua imagem na figura a seguir. Exibirei as execuções principalmente em Android e, quando for necessário, trarei algo em iOS também.

Figura 3.1: Splash Screen sendo exibida

Você concorda e lembra que eu comentei que poderíamos, em conjunto com a imagem, colocar uma animação para que o usuário pudesse ter o sentimento de que a aplicação não está travada? Veja na sequência o novo código para o método `build` de `SplashScreenRoute`.

```
Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        CircularImageWidget(
          imageProvider: AssetImage(
            'assets/images/splashscreen.png',
```

```
          ),
        ),
        Padding(
          padding: const EdgeInsets.only(top: 25.0, bottom: 25),
          child: Text(
            'Aguarde....',
            style: TextStyle(
              fontSize: 40,
              fontWeight: FontWeight.bold,
            ),
          ),
        ),
        Padding(
          padding: const EdgeInsets.only(
              left: 100, right: 100),
          child: LinearProgressIndicator(
            backgroundColor: Colors.blue[200],
            valueColor: AlwaysStoppedAnimation<Color>(Colors.blue[900]),
          ),
        ),
      ],
    );
}
```

Verifique que retiramos o `Center()`, pois com o `Column()` que estamos utilizando trabalhamos a propriedade `mainAxisAlignment: MainAxisAlignment.center` para garantir que os widgets inseridos nele sejam dispostos sempre na parte central do contêiner em que estão, ajustando a distribuição dos demais para cima e para baixo, buscando sempre centralizar seus filhos.

Nosso primeiro componente, em `Column()`, é nosso `CircularImageWidget()`, que já utilizamos. Depois, dentro de um `Padding()`, temos um `Text()` solicitando ao usuário que aguarde e, por fim, um widget padrão do Flutter que reflete uma imagem em constante movimento, o `LinearProgressIndicator()`, que também está em um `Padding()`.

O `Padding()` reserva espaços na área interna do contêiner onde o widget será exibido. Existem outros *factories* para o `EdgetInsets()`, que vamos utilizar no livro, mas vale a pena uma investigação. Agora estamos usando o `only()`, que nos permite dizer quais lados e valores serão atribuídos. Vale a pena uma brincadeira com isso, para você experimentar.

O `LinearProgressIndicator()` poderia ser utilizado sem nenhum argumento. Ele seria renderizado com as configurações padrões, mas optei por customizá-lo com cores escolhidas. O `background` é a cor fixa do widget e `valueColor` é a cor que ficará em constante movimento e, por isso, precisamos utilizar o `AlwaysStoppedAnimation()` para informar essa cor. Esse widget é bem interessante, cabe uma investigação para outros usos também.

Terminamos essa etapa básica de nossa splash screen. Veja na figura a seguir sua nova representação.

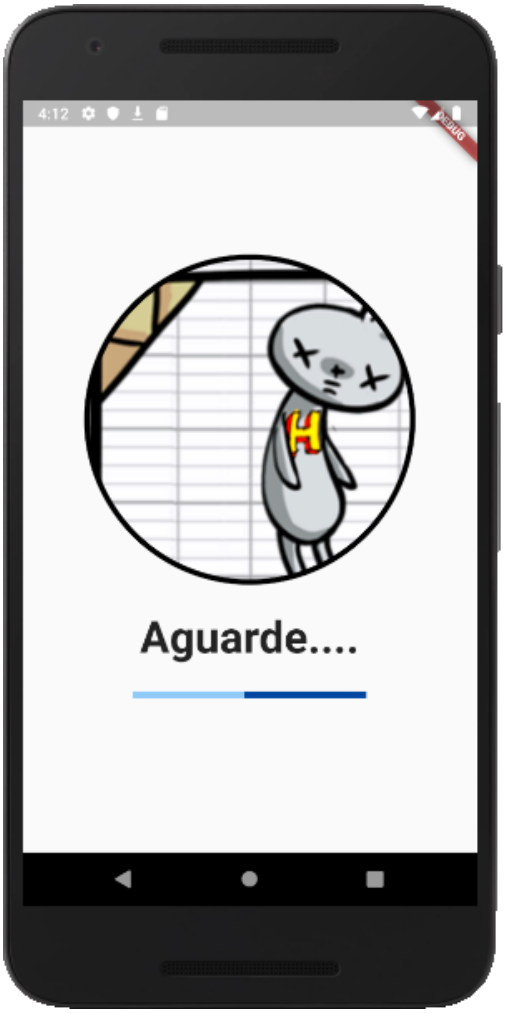

Figura 3.2: Splash Screen refatorada

3.6 Recursos vistos no capítulo

Apenas para enumerar o que vimos neste capítulo, que neste momento foram apenas widgets, trago aqui uma relação que pode auxiliar, caso queira, em uma pesquisa futura, específica para cada ponto trabalhado. São eles: `AlwaysStoppedAnimation`, `AssetImage`, `BoxDecoration`, `BoxShape`, `DecorationImage`, `EdgeInsets`, `ImageProvider`, `Theme`, `ThemeData`, `Padding` e `LinearProgressIndicator`.

Conclusão

Este capítulo foi curto e objetivo. Além de vermos e aplicarmos novos recursos do Flutter, vimos nossa aplicação em execução e exibindo uma janela de splash screen. Nada produtivo ainda para interação com o usuário, mas vamos chegar lá. O objetivo é mostrar recursos que você poderá utilizar em qualquer aplicação que venha a desenvolver.

No próximo capítulo, continuaremos na splash screen, mas veremos um recurso legal, conhecido como *Shared Preferences*, para persistência de "pequenos" dados em nosso objetivo de simples e fácil recuperação.

Capítulo 4
Persistência de dados com Shared Preferences

É raro uma aplicação não ter dados que precisam estar disponíveis entre sessões de sua execução. Precisamos de algo que grave algumas preferências para o uso de nosso app e que, mesmo após fecharmos a aplicação ou até reiniciarmos nosso dispositivo, elas estejam lá para serem lidas e aplicadas novamente.

Uma técnica para persistir dados, vinda lá dos confins da programação, é termos os dados gravados em arquivos, textos ou binários. Depois, os bancos de dados foram utilizados para a persistência de maneira que a manipulação desses dados fosse mais simplificada, mas oferecendo maiores recursos, como seleção e classificação. Como consequência da evolução da tecnologia, surgiu a necessidade de transmissão de dados entre aplicações, o que trouxe, como ferramentas para auxiliar nesse processo, o XML e depois o JSON.

Com o passar dos anos, das tecnologias e dos dispositivos onde os apps são executados, veio o conceito de Shared Preferences e isso é o que veremos neste capítulo.

Sobre Shared Preferences

A tradução literal de Shared Preferences pode ser "preferências compartilhadas", mas para nosso contexto ela pode não trazer o real significado. Podemos melhorá-la se complementarmos a tradução com "... no dispositivo para o aplicativo". Na realidade, o significado é bem simples. É um recurso oferecido pelos dispositivos de armazenamento de pequenas informações, que podem ser recuperadas a qualquer momento pelo aplicativo, sem a necessidade explícita de uma base de dados.

Vamos a um exemplo: quando você instala um app, na maioria das vezes uma apresentação sobre termos de uso é exibida e, depois que você confirma e volta a usar o app, essa informação não aparece mais, pois você já a respondeu uma vez e não há necessidade de exibição a cada execução do app.

No momento em que confirmamos a leitura, o app registra esse comportamento certamente por meio de Shared Preferences, pois é menos custoso e, como dito, não demanda acesso a banco de dados.

É importante termos o conhecimento de que o uso de Shared Preferences pode se dar quando há a necessidade de mantermos dados fisicamente, em pequeno número, não justificando termos um banco de dados. Os tipos de dados que persistimos com Shared Preferences são limitados e veremos isso mais à frente, o que é diferente de um banco de dados, que pode tê-los em forma de tabelas.

Outro tipo de dado que podemos manter, relacionado diretamente a preferências, é a cor de fundo padrão utilizada para sua aplicação. Com isso, seu usuário pode trocar seu tema pessoal para o app.

> Em hipótese alguma grave senhas de usuário por meio deste mecanismo. Para isso, existe um pacote diferente oferecido ao Flutter para gravar preferências de maneira criptografada.

4.1 Tela de boas-vindas com Shared Preferences

Para utilizarmos Shared Preferences, precisaremos de um plugin específico para isso. Um plugin é um recurso que nos garante alguma funcionalidade não fornecida de maneira nativa pela tecnologia que estamos utilizando, no caso, o Flutter.

O Flutter faz uso de Dart, a linguagem em que foi escrita e a que utilizamos para nossas implementações. Dessa maneira, existe um repositório de plugins Dart oficial acessível pelo endereço https://pub.dev/. Você pode "fuçar" este repositório em busca de componentes que possam lhe ser úteis, além de procurar por componentes no GitHub.

O plugin que utilizaremos é o `shared_preferences`, que está na versão `0.5.6+3` no momento da escrita deste livro. Sua página oficial é https://pub.dev/packages/shared_preferences.

Todo plugin que formos utilizar em nossas aplicações passa a ser uma dependência, o que nos leva a registrar essa necessidade em nosso arquivo `pubspec.yaml`, tal como podemos ver na sequência. Observe a indentação correta para o registro do plugin.

```
dependencies:
  flutter:
    sdk: flutter

  cupertino_icons: ^0.1.2
  shared_preferences: ^0.5.6+3
```

Reforçando, sempre que registrarmos uma nova dependência em nossa aplicação, precisamos pará-la e executar o `flutter pub get` no terminal, na pasta de nosso projeto. Entretanto, como já comentado, o Android Studio nos oferece esse recurso diretamente na IDE. Basta estar com o arquivo `yaml` aberto e clicar no link que podemos ver na figura a seguir.

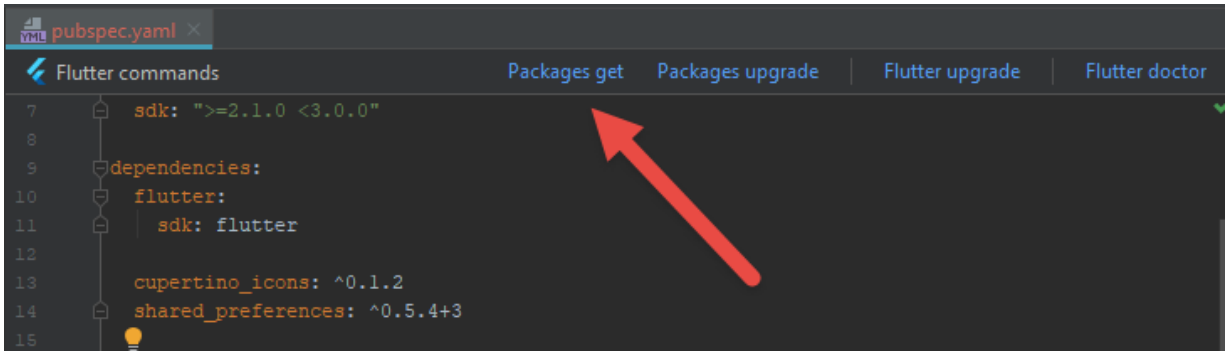

Figura 4.1: Packages get no Android Studio

Criaremos uma tela de boas-vindas que será exibida uma única vez ao usuário caso ele confirme a leitura, por exemplo, do termo de uso ou de alguma orientação. Na pasta `routes`, crie um arquivo dart chamado `welcome_route.dar` e implemente o código a seguir nele. Embora um pouco grande, procure lê-lo e notar novos recursos que estamos utilizando. Após ele, teremos algumas explanações.

```
import 'package:flutter/material.dart';

class WelcomeRoute extends StatefulWidget {
  @override
  _WelcomeRouteState createState() => _WelcomeRouteState();
}

class _WelcomeRouteState extends State<WelcomeRoute> {
    @override
    Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        width: double.infinity,
        child: Stack(
          children: <Widget>[
            Align(
              alignment: Alignment.center,
              child: Text(
                'Bem-vindo',
                style: TextStyle(
                  fontSize: 40,
                  fontWeight: FontWeight.bold,
```

```
                ),
              ),
            ),
            Column(
              mainAxisAlignment: MainAxisAlignment.end,
              children: <Widget>[
                Row(
                  mainAxisAlignment: MainAxisAlignment.end,
                  children: <Widget>[
                    Text(
                      'Marcar como lido',
                      style: TextStyle(
                        fontSize: 20,
                        fontWeight: FontWeight.bold,
                      ),
                    ),
                    SizedBox(
                      width: 5,
                    ),
                    Checkbox(
                      value: true,
                    ),
                  ],
                ),
                SizedBox(
                  width: 10,
                ),
                const RaisedButton(
                  onPressed: null,
                  child:
                      Text('Disabled Button', style: TextStyle(fontSize:
20)),
                )
              ],
            ),
          ],
        ),
      ),
    );
  }
}
```

Nós já conhecemos o `Container`, mas você verificou agora que o tamanho/largura dele está configurado como `double.infinity`? Isso quer dizer que o espaço será a largura do dispositivo, o que é interessante em alguns casos. Existe outra maneira de obter esse valor que veremos mais à frente. Também é preciso saber que esse uso, em alguns casos, pode acarretar problemas, pois existem alguns widgets que, quando filhos de outros, podem precisar de um tamanho fixo, conhecido. Isso pode ocorrer em widgets que permitem scroll (rolagem).

Em seguida, verificamos que o filho do `Container` é um `Stack` e, se você conferir, ele possui uma propriedade chamada `children`, igual ao `Column` e também ao `Row`, que estamos utilizando.

O `Stack`, em sua tradução literal, significa pilha, ou, em uma tradução computacional, empilhamento. Em outras palavras, podemos dizer que os widgets serão empilhados como se estivessem em camadas sobrepostas, diferente de `Column` e `Row`, que têm seus filhos na mesma camada de exibição.

Como filhos do `Stack`, temos um `Align` ao centro e uma `Column`. O `Align` é um widget que nos possibilita "dizer", explicitamente, onde queremos que seu widget filho seja exibido. Em nosso caso, o `Text`, que no momento tem uma mensagem simples, poderia exibir todo um contexto de boas-vindas com um conjunto de widgets, como uma imagem para o período do dia, uma mensagem de bom dia, boa tarde ou boa noite, ou seja, o que for interessante para seu app. O `Align` tem seu uso orientado mais a `Stacks`.

Nesse exemplo, poderíamos ter dois `Columns` ou dois `Align` porque um está sobre o outro como duas folhas de papel transparentes, onde veríamos a imagem do primeiro papel atrás da imagem do segundo, que está por cima.

No `Column`, estamos alinhando a posição inicial de renderização dos componentes ao final da visão por meio de `mainAxisAlignment: MainAxisAlignment.end`.

Nosso `Column` tem três componentes como filhos: `Row`, `SizedBox` e um `RaisedButton`. O `Row` é um contêiner de muitos filhos, assim como o `Column` e `Stack`. Entretanto, seus filhos são inseridos sempre na vertical, um ao lado do outro e, nele (`Row`), temos dois filhos: um `Text` e um `CheckBox`, onde o usuário verá um texto orientativo e marcará a leitura para que, em uma próxima execução, essa janela de boas-vindas não seja exibida.

Como separador dos componentes que ficarão na base da tela do dispositivo, utilizamos um controle chamado `SizedBox`, uma caixa com um tamanho específico. Podemos dimensionar esse controle pelas propriedades `width/largura` ou `height/altura`. Em nosso caso, como ele está dentro de uma `Column`, dimensionamos sua largura.

Há ainda outros controles que podem trabalhar como separadores de widgets, como o `Divider`, `VerticalDivider` e `Spacer`, mas aqui, pela simplicidade, optamos por `SizedBox`.

Ao final, temos um `RaisedButton`, um dos muitos widgets com características de botão que o Flutter oferece, e você verá que nós mesmos podemos criar nossos botões. Veja que utilizamos `const` para esse controle. É uma prática comum quando nossos controles não terão criação dinâmica com base em argumentos. Isso economiza recursos computacionais na execução do app. Ao final, temos um `RaisedButton`, um dos muitos widgets com características de botão que o Flutter oferece, e você verá que nós mesmos podemos criar nossos botões.

A interface que desenhamos no código apresentado e discutido pode ser verificada na figura a seguir, mas sua aplicação ainda não a exibirá.

Figura 4.2: Rota Welcome sendo exibida

4.2 A execução para exibir as boas-vindas

O primeiro passo é identificar quando e onde realizaremos a invocação dessa nova rota. A invocação será feita na splash screen e depois de um tempo de exibição dela (da splash screen). Veja o código a seguir, que deve ser implementado em nossa classe de `SplashScreen`.

```
@override
void initState() {
```

```
    super.initState();
    Timer(Duration(seconds: 3), () {
      Navigator.push(
        context,
        MaterialPageRoute(builder: (context) => WelcomeRoute()),
      );
    });
}
```

Anteriormente, tínhamos a sobrescrita ao `initState()`, mas sem implementação nossa. Agora temos um temporizador que após três segundos promoverá a navegação para uma nova rota, nossa `WelcomeRoute`. Você deverá importá-la para o início deste arquivo, assim como `dart:async` para o `Timer`.

Trouxemos para o código anterior alguns novos recursos: `Timer`, `Duration`, `Navigator` e `MaterialPageRoute`. Vamos a algumas explicações sobre eles. O `Timer` é um componente com características de um contador de um tempo determinado, como segundos, minutos, milissegundos, horas e dias, por exemplo. Esse tempo é estipulado por `Duration`, que pode definir o período em milissegundos, segundos, minutos ou horas. É possível ainda que esse temporizar ocorra de maneira repetida. O segundo argumento para `Timer` é uma função `VoidCallback` ou `Function`, que conterá o comportamento a ser executado quando o `Duration` for concluído.

Em nosso caso, quando o `Timer` for concluído, temos a invocação do `Navigator` por meio do método `push`, que traz uma nova página para a pilha. Lembra do conceito de pilha que comentamos há pouco? As páginas exibidas têm a mesma ideia.

O `MaterialPageRoute` tem a característica de propiciar uma navegação onde haverá a substituição de uma tela inteira. Estamos utilizando-o com um argumento, o `builder`, que tem como comportamento, em nosso exemplo, a renderização de `WelcomeRoute`. Veja que estamos utilizando a técnica de `arrow function` por termos uma única instrução como comportamento da função enviada ao widget.

Tente agora executar sua aplicação. Veja em seu dispositivo que não conseguimos mudar o estado do `CheckBox` e tampouco interagir com nosso `RaisedButton`. Isso se dá pelo fato de, no `RaisedButton`, na propriedade `onPressed`, termos atribuído `null` em vez de uma função com algum comportamento, o que desabilita o botão. Já para o `CheckBox` estamos atribuindo um valor constante para `value` e não estamos capturando a interação do usuário com o controle.

4.3 Interação com o CheckBox e RaisedButton

Como comentado, por mais que você interaja com o `CheckBox`, ele não altera seu estado, pois o valor atribuído a `value` é sempre `true`. Precisamos atrelar essa atribuição a uma variável, que alterará seu valor sempre que o usuário interagir com o `CheckBox`. Isso é relativamente simples.

Logo após a declaração de nossa classe de estado, vamos declarar e inicializar nossa variável de controle. Veja o código a seguir. Mantive a declaração da classe para auxiliar na localização do código.

```
class _WelcomeRouteState extends State<WelcomeRoute> {
  bool _checkBoxIsChecked = false;

  // Demais código
}
```

Em seguida, é necessário configurarmos nosso `CheckBox` para utilizar a variável para definir seu estado e um comportamento para quando o usuário interagir com ele. Veja o código a seguir, específico para este widget.

```
Checkbox(
  value: this._checkBoxIsChecked,
  onChanged: (status) {
    setState(() {
```

```
      this._checkBoxIsChecked = status;
    });
  },
),
```

Note que a propriedade `onChanged` representa uma função, que recebe como argumento o novo estado para o `CheckBox`, o qual precisamos atribuir à nossa variável de controle. Entretanto, após a atribuição, precisamos que o novo valor seja refletido visualmente, o que nos leva a utilizar o `setState()`, que já vimos no capítulo 2. Caso você já conheça o Flutter, sabe de sua importância no controle de estados dos widgets.

Teste sua aplicação. Veja que o `CheckBox` já tem um visual diferente e sua interação manipulará seu estado de checado ou não. Agora, com o estado do `CheckBox` gerenciável, podemos pensar em implementar a captura de interação com o `RaisedButton`. Mas para onde vamos levar o usuário quando ele interagir com esse controle?

Antes de implementarmos o método para o `RaisedButton`, vamos criar esse destino para o usuário. Na pasta `routes`, crie um arquivo Dart chamado `home_route.dart` e implemente o código a seguir nele.

```
import 'package:flutter/material.dart';

class HomeRoute extends StatefulWidget {
  @override
  _HomeRouteState createState() => _HomeRouteState();
}

class _HomeRouteState extends State<HomeRoute> {
  @override
  void initState() {
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
```

```
        width: double.infinity,
        child: Stack(
          children: <Widget>[
            Align(
              alignment: Alignment.center,
              child: Text(
                'Vamos Jogar?',
                style: TextStyle(
                  fontSize: 40,
                  fontWeight: FontWeight.bold,
                ),
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

Precisamos agora voltar à nossa `welcome_route` e configurar a navegação quando o `RaisedButton` for pressionado. Veja a seguir o código específico para isso. Você terá um erro nessa implementação. Realize o import para a nova página e leia o comentário após a listagem.

```
RaisedButton(
  onPressed: () => Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => HomeRoute()),
  ),
  child: Text('Avançar', style: TextStyle(fontSize: 20)),
),
```

Veja que retiramos o `const` que antecedia o `RaisedButton()`, pois agora, no `onPressed`, temos o uso de uma variável, `context`, o que torna a implementação dinâmica, podendo variar de uma invocação para outra. Na implementação de `OnPressed`, em forma de `array function`, temos uma navegação entre páginas que irá para a que acabamos de implementar e podemos vê-la na figura a seguir.

Figura 4.3: Rota da aplicação após boas-vindas

Agora sim, vamos à implementação com o `Shared Preferences`. Nosso código deverá ocorrer no momento em que o usuário navegar da rota de boas-vindas para a de início da aplicação. Vamos trabalhar sempre de maneira organizada em nosso app, separando as necessidades em pastas, buscando uma arquitetura em camadas. Em `lib`, vamos criar uma pasta chamada `shared_preferences`. Crie nela um arquivo Dart chamado `app_preferences.dart` e, nele, a classe a seguir.

```
import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';
```

```
class AppPreferences {
  static setWelcomeRead({@required bool status}) async {
    final SharedPreferences prefs = await SharedPreferences.getInstance();
    prefs.setBool('BoasVindasLida', status);
  }
}
```

Observe a importação do plugin logo no início. Veja que nosso método é estático, o que nos possibilita invocá-lo sem termos a classe instanciada. Atenção à assincronicidade na declaração (`async`) e na obtenção de uma instância (`await`).

Tudo bem, é funcional, mas vamos analisar um pouco as boas práticas. Não é recomendado termos literais ou valores constantes sendo utilizados como parâmetros ou operadores em nosso código. Se em algum momento precisarmos alterar esse dado e ele for utilizado em várias partes da aplicação, temos que sair alterando e correndo riscos.

Vamos então reduzir esse trabalho fazendo uso de constantes. Na pasta `lib` , crie outra chamada `app_constants` e, na nova pasta, um arquivo Dart chamado `shared_preferences_constants.dart` . Nele, inicialmente, insira a instrução a seguir. Em Dart, a convenção para nomear constantes é termos a letra `k` em minúsculo como prefixo.

```
const kWelcomeRead = 'BoasVindasLida';
```

Voltando à `AppPreferences` , substitua a literal `'BoasVindasLida'` por `kWelcomeRead` . Um erro surgirá, mas basta importar o arquivo indicado pelo Android Studio e tudo se ajustará.

4.4 Registro e leitura da Shared Preference

Já temos o recurso de gravação da preferência implementado. Precisamos agora invocá-lo e faremos isso no `onPressed` do `RaisedButton` , que está em nosso widget `WelcomeRoute` . Veja na

sequência a nova implementação. Note a mudança do método de `arrow function` para `body function`. Atente ao import para `AppPreferences`.

```
RaisedButton(
  onPressed: () async {
    AppPreferences.setWelcomeRead(
        status: this._checkBoxIsChecked);

    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => HomeRoute()),
    );
  },
  child: Text('Avançar', style: TextStyle(fontSize: 20)),
),
```

Observe algumas mudanças. Veja novamente a assincronicidade no método anônimo para a propriedade `onPressed` e o `await` na invocação de nosso serviço.

Vamos agora validar a existência, ou não, dessa preferência e tomar decisões com base em seu valor persistido. Nosso problema está em, a partir da splash screen, chamar a Welcome ou a Home, dependendo sempre da preferência do usuário em relação à mensagem de boas-vindas. Vamos então inserir o método da listagem a seguir em nosso `app_preferences.dart`:

```
static getWelcomeRead() async {
    final SharedPreferences prefs = await SharedPreferences.getInstance();
    return prefs.getBool(kWelcomeRead)  ?? false;
}
```

Observe que estamos realizando uma leitura de um `bool`, que tem como chave o valor de nossa constante `kWelcomeRead`. Precisamos agora consumir este serviço. Ainda, no retum, veja `?? false`. Isso garante que, caso não exista a preferência registrada para o app, o valor a ser retomado seja `false`, o que evita o retorno de `null`.

Para isso, vamos invocar esse método na `SplashScreen` antes de navegarmos para a página alvo, a Home, e isso está sendo feito no `initState()`. Mas se colocarmos uma lógica de verificação para a rota (página) que a aplicação deve seguir, estamos indo contra os princípios de coesão e acoplamento, pois um método deve sempre realizar apenas uma funcionalidade. Precisamos, assim, delegar. Vamos criar um método específico para redirecionamento, tal qual o código a seguir, em nossa `SplashScreen`. Atente-se ao import.

```
_whereToNavigate({@required bool welcomeRead}) {
    if (welcomeRead)
      Navigator.push(
          context, MaterialPageRoute(builder: (context) => HomeRoute()));
    else
      Navigator.push(
        context,
        MaterialPageRoute(builder: (context) => WelcomeRoute()),
      );
}
```

Vamos alterar nosso `initState()` ainda na `SplashScreen`, como podemos ver na sequência. Verifique que estamos tratando a chamada ao método assíncrono de outra maneira, fazendo uso do `then()`. O valor de `status` será atribuído pelo método invocado e então consumimos o novo método criado anteriormente. Novamente, lembre-se do import.

```
void initState() {
    super.initState();
    Timer(Duration(seconds: 3), () {
      AppPreferences.getWelcomeRead().then((status) {
        _whereToNavigate(welcomeRead: status);
      });
    });
}
```

Precisamos aqui de um momento para reflexão. O `initState()` faz parte do ciclo de vida do `StatefulWidget` e só pode ser executado uma única vez quando o widget é inicializado. É preciso muita

atenção ao fazer uso de chamadas assíncronas no `initState()`. A recomendação é não fazer, principalmente se precisarmos desse valor para renderizar algo por meio do método `build()`. E o que fizemos vai funcionar.

Ainda assim, há um meio oferecido pela plataforma de invocarmos assincronamente um método usando o `await` e isso se dá pelo uso da instrução `WidgetsBinding.instance.addPostFrameCallback()`, que trago na sequência aplicada ao nosso mesmo problema.

```
void initState() {
    super.initState();
    Timer(Duration(seconds: 3), () {
        WidgetsBinding.instance.addPostFrameCallback((_) {
          AppPreferences.getWelcomeRead().then((status) async {
            await _whereToNavigate(welcomeRead: status);
          });
        });
    });
}
```

Mas e se realmente precisarmos do resultado de uma chamada assíncrona que interferirá em nossa interface com o usuário? Aí precisamos utilizar widgets específicos, como `FutureBuilder` e `StreamBuilder`.

Por ora, execute a aplicação, confirme leitura na página de boas-vindas, avance, depois execute novamente sua aplicação e veja se ela vai direto para a página principal. Os dados persistidos em `SharedPreferences` serão descartados na desinstalação do app do dispositivo. Caso você queira ter isso de maneira forçada, pode invocar o método `clear()` na instância obtida pelo plugin, algo como:

```
final SharedPreferences prefs = await SharedPreferences.getInstance();
await prefs.clear();
```

Conclusão

Vimos um recurso legal para persistência de pequenos dados que pode nos auxiliar em alguma tomada de decisão em nossa aplicação. Aplicamos esse recurso na exibição de uma tela de boas-vindas e registro de leitura, evitando uma nova exibição para o usuário que não mais quer ler o conteúdo.

Trabalhamos novos widgets e tivemos nosso primeiro plugin externo ao Flutter. Vimos que é um processo simples e tivemos acesso ao repositório oficial de plugins para investigarmos e identificarmos componentes que nos possam ser úteis.

Apenas para enumerar o que vimos neste capítulo, que neste momento foram apenas widgets, trago aqui a relação deles, o que pode lhe auxiliar, caso queira, em uma pesquisa futura específica para cada ponto trabalhado. São eles: `Align`, `CheckBox`, `Duration`, `Row`, `SizedBox`, `Stack`, `Timer`, `RaisedButton` e `SharedPreferences`.

No próximo capítulo, trabalharemos com um componente de navegação, o `Drawer`, que nos exibirá uma série de opções, como `Registrar Palavras` e `Jogar`, para que possamos escolher a que quisermos.

Capítulo 5
Um menu com opções de acesso e início com animações

O Flutter tem alguns widgets que nos permitem oferecer ao usuário da aplicação algumas possibilidades de navegabilidade. Na figura a seguir, podemos ver dois exemplos com imagens tiradas da documentação do Flutter. A imagem da esquerda representa o `BottomNavigationBar`, um componente comum em apps mobile, onde as opções são disponibilizadas em uma barra normalmente na base do dispositivo e, ao selecionar uma, o usuário visualiza a página desejada. Já a figura da direita traz um `Drawer`, que nada mais é que uma página com opções também, mas dispostas em forma de um menu, algo também comum em apps mobile e em aplicações web. Essas opções normalmente são acessíveis pela interação com um widget conhecido como `Hamburger Menu`, que também veremos, e que fica normalmente disponível no topo esquerdo das páginas da aplicação. Caso você queira dar uma olhada na documentação desses widgets, acesse: https://api.flutter.dev/flutter/material/BottomNavigationBar-class.html e https://flutter.dev/docs/cookbook/design/drawer.

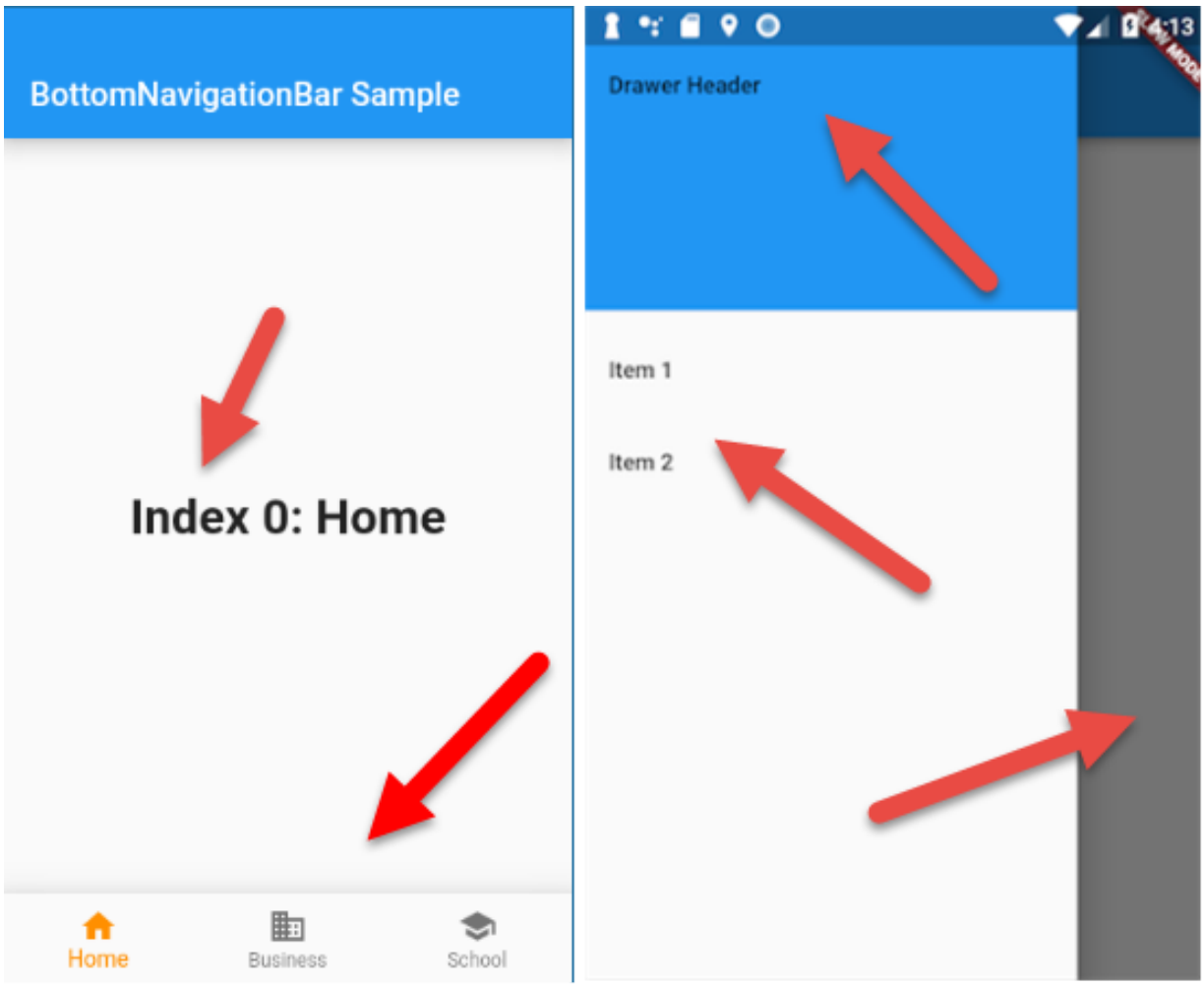

Figura 5.1: Widgets para navegação entre páginas

Neste capítulo, trabalharemos com o `Drawer` como aplicação para essas navegações, mas saiba que, além das duas opções comentadas, nós podemos criar as nossas, personalizadas para nossas necessidades e gostos.

5.1 Criação de nosso Drawer

Estamos trabalhando de maneira organizada em nossa aplicação e vamos manter isso. Em nossa pasta `lib`, crie outra, chamada `drawer` e, dentro dela, um arquivo Dart chamado `drawer_route.dart`.

Vamos criar nele um widget Stateless, como apresentado na sequência. Veja que temos um novo parâmetro do `Scaffold`, o `drawer`, que recebe um `Drawer`.

```
import 'package:flutter/material.dart';

class DrawerRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(
          "Jogo da Forca",
        ),
        centerTitle: true,
      ),
      body: Container(),
      drawer: Drawer(),
    );
  }
}
```

Com esse código já podemos testar nossa aplicação, mas precisamos alterar os pontos onde invocamos a `HomeRoute`, substituindo por `DrawerRoute`. Isso está na `SplashScreenRoute` e na `WelcomeRoute`. Realize essas alterações, execute sua aplicação e compare com a figura:

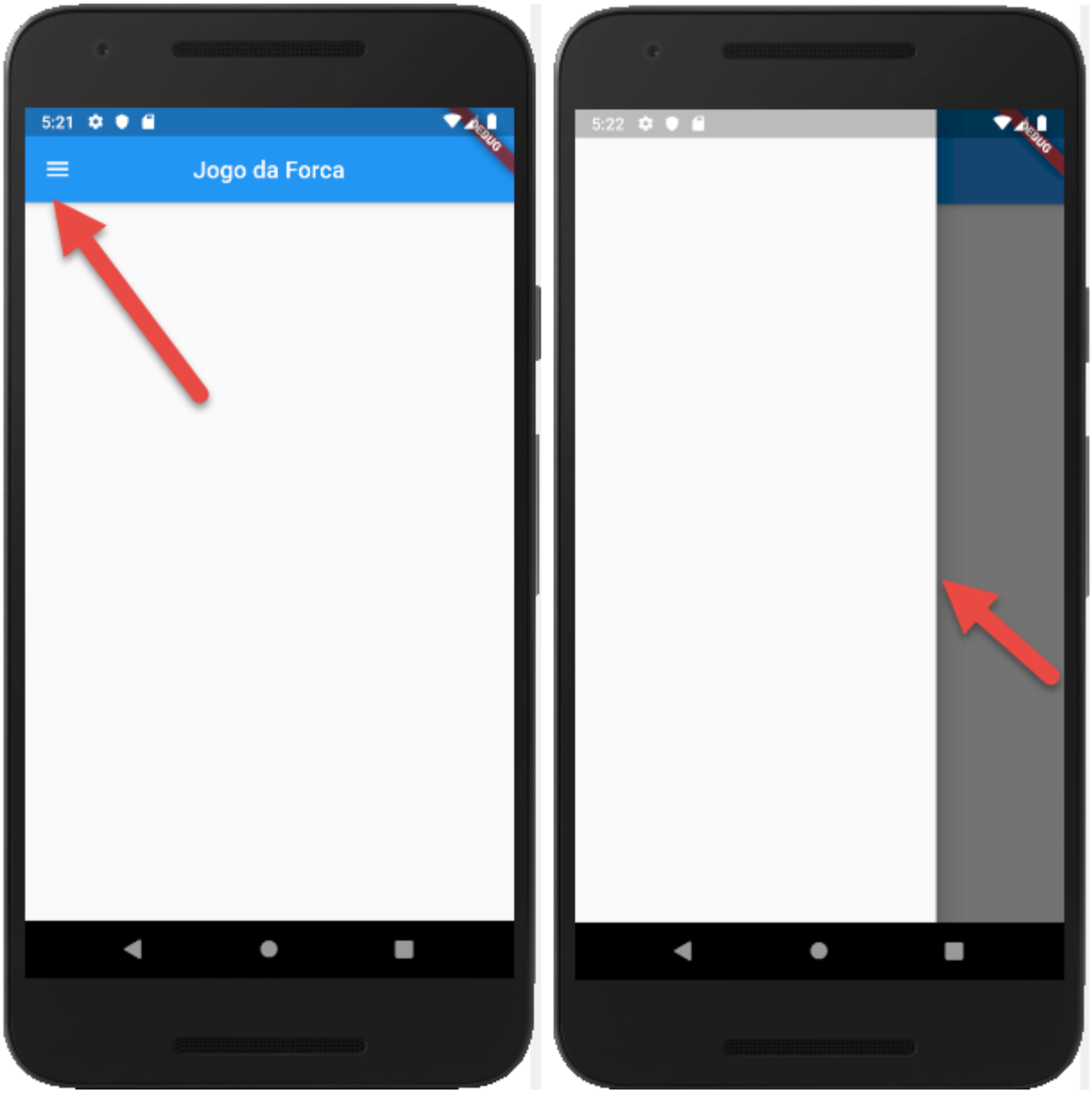

Figura 5.2: O Drawer para nossa aplicação

Observou, no topo esquerdo, na `AppBar`, a existência de um ícone? Ele foi exibido automaticamente, pois foi identificada a existência de um Drawer no `Scaffold`. Ao interagirmos com esse ícone, temos a visualização de nosso `Drawer`, ainda vazio.

Podemos customizar esse ícone por meio da propriedade `leading` do `AppBar`. Só que, realizando essa customização, precisaremos

abrir o `Drawer` via código, o que não é tão difícil, mas vamos usar a praticidade aqui. Mais à frente veremos isso.

5.2 O cabeçalho de nosso Drawer

O `Drawer()`, como a maioria dos widgets, possui um parâmetro chamado `child` e nele podemos inserir diversos tipos de widgets e então formatar nosso Drawer como desejarmos. Para este momento, faremos uso de recursos já oferecidos pelo Flutter e inseriremos em nosso Drawer um cabeçalho padrão. Veja na sequência um novo código para esta adaptação. O código traz apenas a nova implementação para o `Drawer()`.

```
Drawer(
    child: Column(
      children: <Widget>[
        DrawerHeader(
          padding: EdgeInsets.zero,
          margin: EdgeInsets.zero,
          decoration: BoxDecoration(
            image: DecorationImage(
              fit: BoxFit.fill,
               image:
AssetImage('assets/images/drawer/drawer_header.png'),
            ),
          ),
          child: Container(),
        ),
      ],
    ),
),
```

Percebeu o uso de uma imagem, que você deve inserir em seu projeto, e a configuração do `pubspec.yaml` para esse novo asset? É importante também lembrar que podemos utilizar a ideia da pasta toda como asset inserindo `- assets/images/drawer/` em vez de `-`

`assets/images/drawer/drawer_header.png` . Mas deixo a seu critério o que utilizar neste momento.

Realize as alterações apresentadas no código anterior, execute sua aplicação e a compare com a figura a seguir.

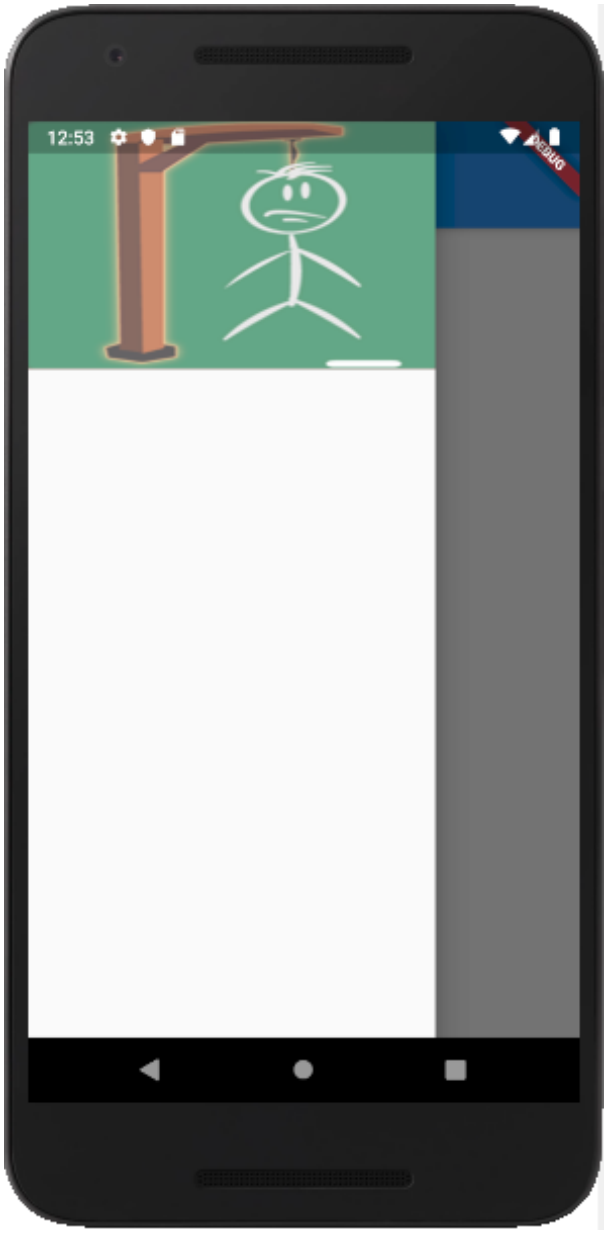

Figura 5.3: O Drawer com um cabeçalho

É importante sabermos que podemos desenhar no Drawer da maneira que quisermos, inserindo nele contêineres e então os

decorando, porém a ideia aqui é mostrar produtividade para que possamos utilizar um Drawer no padrão oferecido pelo Material Design.

Vamos substituir o `Container()`, que está como `child` do `DrawHeader`, pelo código a seguir. Note as imagens que precisam ser adicionadas em seu projeto. Após o código, temos alguns comentários sobre ele.

```
UserAccountsDrawerHeader(
    decoration: BoxDecoration(color: Colors.transparent),
    accountName: Text(
      "Everton Coimbra de Araújo",
      style: TextStyle(
        color: Colors.black,
      ),
    ),
    accountEmail: Text(
      "evertoncoimbra@gmail.com",
      style: TextStyle(
        color: Colors.black,
      ),
    ),
    currentAccountPicture: CircleAvatar(
      backgroundImage:
          AssetImage('assets/images/drawer/avatar_picture.jpg'),
    ),
    otherAccountsPictures: <Widget>[
      CircleAvatar(
        backgroundImage: AssetImage(
            'assets/images/drawer/avatar_picture_03.png'),
      ),
    ],
),
```

`UserAccountsDrawerHeader()` é um widget que desenha o modelo padrão de um cabeçalho para um Drawer nas normas e padrões do Material Design. Colocamos uma cor transparente para o `decoration`, pois o padrão é a cor azul, mas isso pode ser configurado ao se trabalhar com temas. Comente essa linha e veja o resultado, como curiosidade.

Esse widget ainda tem uma série de propriedades (argumentos), tais como: `accountName`, `accountEmail`, `currentAccountPicture` e `otherAccountPictures`. As duas primeiras são bem claras, deverão receber o nome do usuário conectado na aplicação e seu e-mail. Já a segunda e a terceira referem-se a imagens do usuário que serão exibidas no cabeçalho.

Pelo formato como estamos trabalhando nosso Drawer, deixei apenas uma dessas outras imagens. A que seria a segunda imagem, eu deixei comentada para que você possa ver em sua aplicação e então decidir se a mantém ou não.

Para a exibição das imagens, estamos optando por utilizar um widget já existente, o `CircleAvatar`, que, de certa maneira, desempenha o mesmo comportamento que implementamos em nosso widget `CircularImageWidget` anteriormente. O Flutter traz vários widgets prontos, mas nada nos impede de criarmos os nossos. Realize essas alterações, execute sua aplicação e a compare com a figura a seguir.

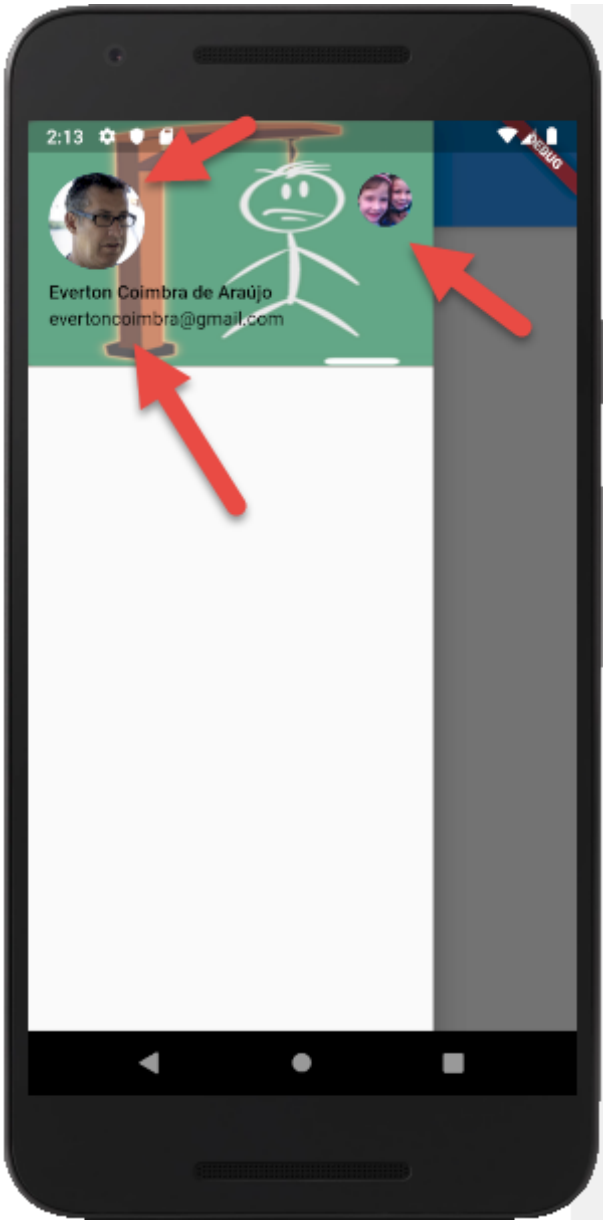

Figura 5.4: O Drawer com um cabeçalho

Se você observar nosso código, ele já está ficando grande e pode ficar maior ainda. Precisamos cuidar disso para garantir uma boa legibilidade e primarmos em coesão e acoplamento. Vamos então tirar tudo o que se refere ao cabeçalho de nosso Drawer dessa rota e implementar em um widget específico o cabeçalho para o Drawer. Essa deverá sempre ser sua prática.

Na pasta `drawer`, crie outra chamada `widgets`. Voltando ao código, selecione todo o widget `DrawerHeader()`, clique com o botão direito sobre a seleção, depois em `Refactor`, em `Extract` e, então, `Flutter Widget`. Dê a ele o nome `DrawerHeaderApp`. Veja que o widget foi criado no mesmo arquivo, mas vamos tirá-lo daí.

Na pasta `widgets`, recentemente criada, crie um arquivo chamado `drawerheader_app.dart` e recorte este código extraído para o novo arquivo. Lembre-se de inserir a importação `import 'package:flutter/material.dart';` no início do arquivo.

Voltando ao nosso Drawer, há agora um erro na chamada do widget recém-criado. Clique sobre ele e, na lâmpada vermelha que aparece com o erro, aceite a importação do arquivo Dart recém-criado.

Pode testar sua aplicação novamente. Ela está funcionando normalmente, mas agora temos responsabilidades mais coesas.

5.3 As opções oferecidas ao usuário pelo Drawer

Notando a figura anterior, nosso Drawer traz o seu título (header) já configurado com cor e imagem, mas o corpo abaixo do header ainda está branco. Vamos mudar isso. Faremos com que a cor do cabeçalho varie, de maneira gradiente, de cima para baixo na mesma cor mas em tonalidades diferentes.

Como estamos criando widgets para utilizarmos, faremos o mesmo agora para o corpo do Drawer. Na pasta `widgets`, que criamos dentro de `drawer`, crie agora outro arquivo Dart, chamado `drawerbody_app.dart` e, dentro dele, insira o código a seguir.

```
import 'package:flutter/material.dart';

class DrawerBodyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
    return Expanded(
      child: Container(
        decoration: new BoxDecoration(
          gradient: new LinearGradient(
            colors: [Colors.green[100], Colors.green[400]],
            begin: Alignment.topLeft,
            end: Alignment.bottomRight,
            stops: [0.0, 1.0],
          ),
        ),
      ),
    );
  }
}
```

Voltando para nosso `DrawerRoute` como novo filho da coluna do Drawer, insira `DrawerBodyApp()` abaixo do `DrawerHeaderApp()`. Execute novamente sua aplicação, lembrando que, se você está com sua aplicação em execução, o Hot Reload já atualiza a interface com uma velocidade maravilhosa a cada gravação. Veja a figura a seguir, agora com o corpo do Drawer.

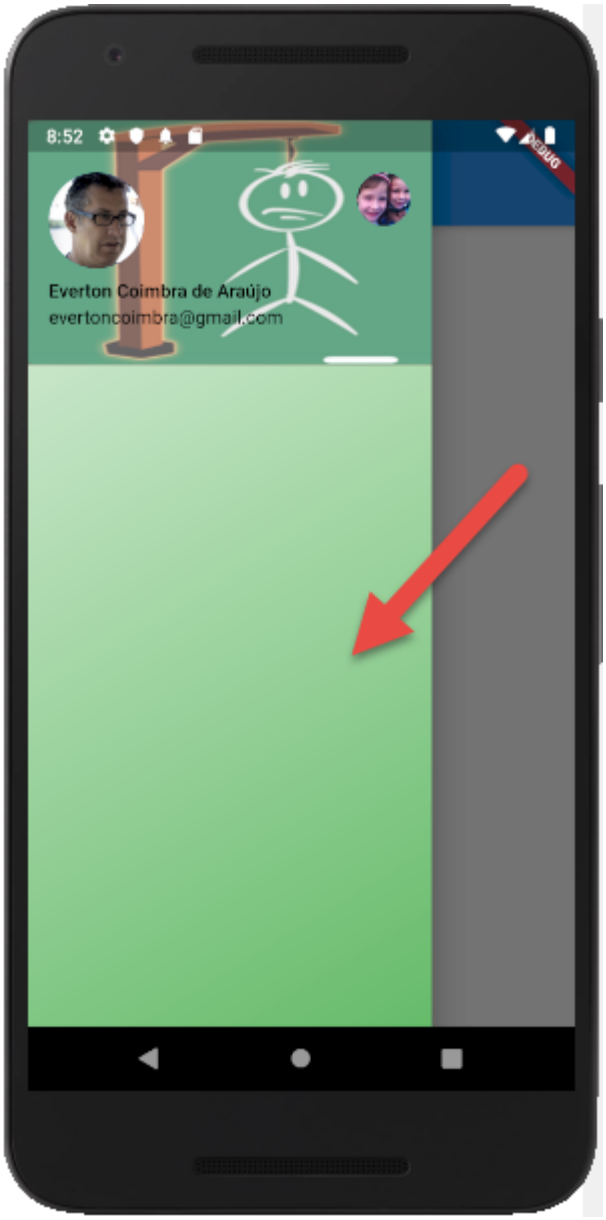

Figura 5.5: O Drawer com um corpo

Agora precisamos exibir no corpo do Drawer as opções que nosso usuário terá no aplicativo. Antes de começarmos a codificar, farei algo diferente, trarei a figura apresentando o resultado que precisamos alcançar. Veja-a na sequência.

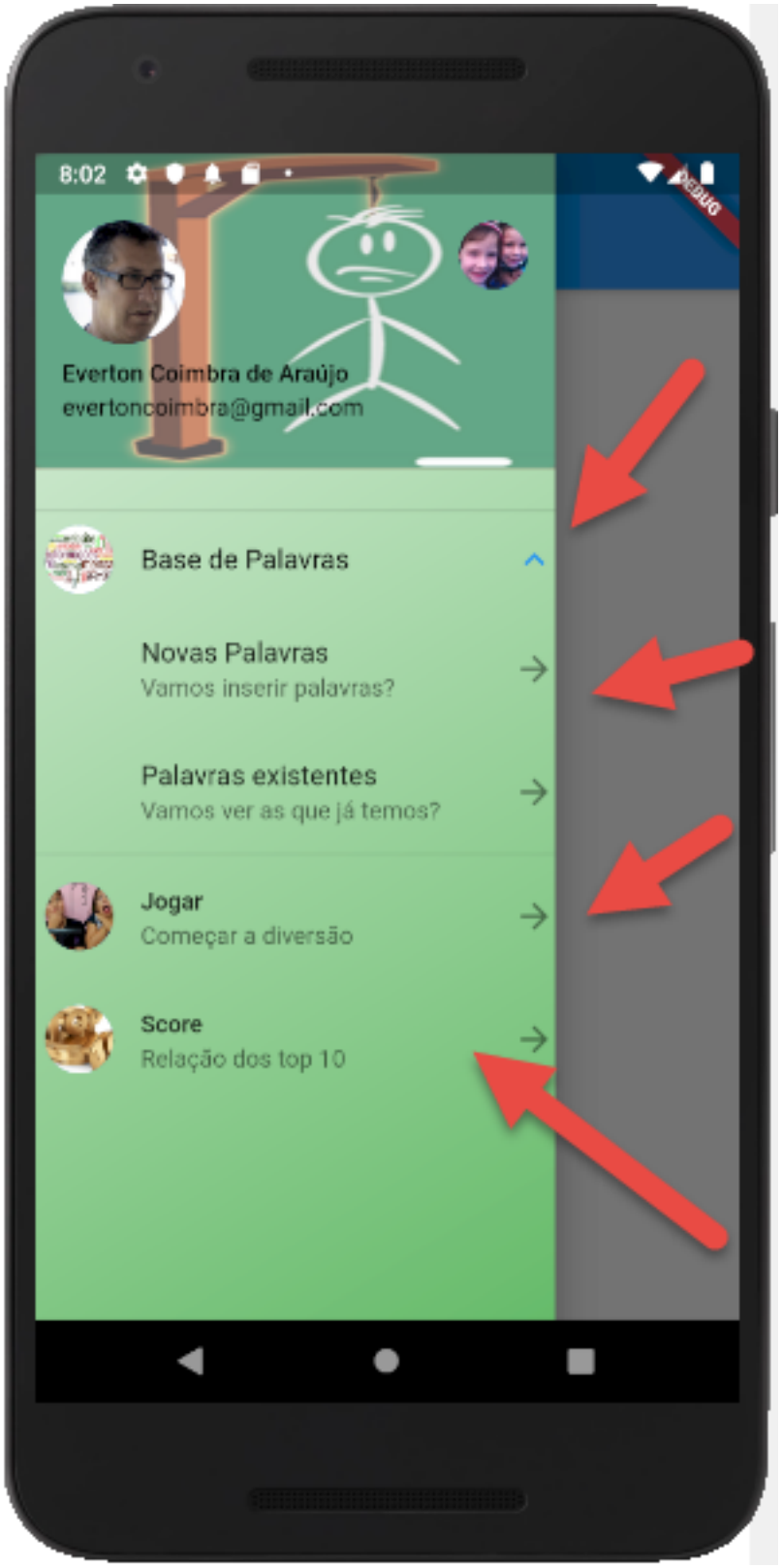

Figura 5.6: O Drawer com as opções ao usuário

Notou na figura que temos três opções principais e que a primeira possui subitens, que são exibidos quando o usuário a seleciona? Muito bom, vamos para uma explicação teórica sobre essas opções.

Teremos basicamente três opções para o usuário, porém como pode ser visto na figura anterior, quando a primeira opção for selecionada, ela se expandirá, exibindo duas subopções. Veremos nessa implementação alguns widgets novos.

O primeiro é o `ListView`, um widget responsável por gerar uma listagem. Esse widget é inteligente o bastante para saber que, se seus filhos ultrapassarem a altura ou a largura (se sua orientação for horizontal), ele deve fazer o scroll automaticamente.

Qualquer widget pode ser filho de `ListView`, entretanto o mais comum é que seus filhos sejam `ListTile`, que contém uma estrutura própria para ser utilizada nele. Na figura anterior, as subopções da primeira opção e as duas últimas são resultados do `ListTile`.

Em situações nas quais temos uma opção que deverá exibir outras opções, como vimos na figura anterior, o recomendado é utilizarmos o `ExpansionTile`, que, ao ser selecionado, se expande e exibe outros widgets registrados como seus filhos.

Na pasta `drawer/widgets`, vamos criar um novo arquivo Dart chamado `drawerbodycontent_app.dart` e, nele, vamos inserir o conteúdo a seguir. Trabalharemos este widget em partes por termos vários códigos nele.

```
import 'package:flutter/material.dart';

class DrawerBodyContentApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ListView(
      children: <Widget>[
        ListTileTheme(
          contentPadding: EdgeInsets.only(left: 6.0),
          child: ExpansionTile(
```

```
            leading: CircleAvatar(
              backgroundImage:
                  AssetImage('assets/images/drawer/base_de_palavras.png'),
            ),
            title: Text(
              "Base de Palavras",
              style: TextStyle(
                color: Colors.black,
                fontSize: 16.0,
              ),
            ),
            onExpansionChanged: null,
          ),
        ),
      ],
    );
  }
}
```

No código anterior, temos como retorno do widget um `ListView` como antecipado, e como primeiro filho temos um `ListTileTheme`, que terá então o `ExpansionTile`. Aqui, se quiser, você pode realizar um teste deixando o `ExpansionTile` diretamente como filho do `ListView`, mas você notará que ele será renderizado mais adentro dos demais filhos, que logo implementaremos. O `ListTileTheme` permite que trabalhemos o tema (do Material Design) para os `ListTile`, o que nos possibilita definir um `contentPadding`, como pode ser visto no código anterior.

Para o `ExpansionTile`, temos o `leading`, que define o que será exibido no início do item. Para este caso escolhemos o `CircleAvatar`, mas poderia ser qualquer widget. Como `title`, colocamos um `Text` formatado, mas também poderia ser qualquer widget. Veja que não definimos o `trailing`, que é a pequena seta que aparece ao lado direito do item. Ela é definida automaticamente pelo `ExpansionTitle`.

Deixei como nula a propriedade `onExpansionChanged`, que recebe uma função a ser executada quando ocorre a expansão e contração do

item, apenas para que você saiba que há essa possibilidade de controle.

Agora, precisamos definir os itens que serão exibidos quando o `ExpansionTile` for expandido. Você deverá inserir o código a seguir uma linha após o `onExpansionChanged`, da listagem anterior.

```
children: <Widget>[
      ListTile(
        contentPadding: EdgeInsets.only(left: 62.0),
        trailing: Icon(Icons.arrow_forward),
        title: Text('Novas Palavras'),
        subtitle: Text('Vamos inserir palavras?'),
        onTap: () {},
      ),
      ListTile(
        contentPadding: EdgeInsets.only(left: 62.0),
        trailing: Icon(Icons.arrow_forward),
        title: Text('Palavras existentes'),
        subtitle: Text('Vamos ver as que já temos?'),
        onTap: () {},
      ),
],
```

Observe que definimos os filhos para o `ExpansionTile` e optamos por dois `ListTile`, que, como dito anteriormente, é o mais recomendado, mas poderia ser qualquer widget.

Veja que agora não temos o `leading` por opção, mas temos o `subtitle`, que exibe uma linha com fonte menor abaixo do `title`, e agora colocamos o `trailing` como ícone.

Para auxiliar e saber como capturar a interação do usuário, deixei as propriedades `onTap` com funções vazias, apenas para você saber onde deverá implementar o comportamento necessário para cada opção.

Com esses códigos implementados, se você quiser, já pode verificar sua aplicação em execução, mas precisamos realizar uma

adaptação em nosso `DrawerBodyApp`, pois agora mandaremos para ele o conteúdo que ele deverá exibir.

Dessa maneira, antes do método `build`, insira o código a seguir, onde declaramos uma propriedade para a classe e a inicializamos no construtor, como um parâmetro opcional, mas requerido. Essa técnica auxilia na hora da implementação do consumo da classe.

```
final Widget child;

const DrawerBodyApp({@required this.child});
```

Depois, precisamos inserir o código a seguir após o fechamento do `BoxDecoration` no método `build()`.

```
child: this.child,
```

Enfim, precisamos enviar nosso `DrawerBodyContentApp` para o `DrawerBodyApp`. Veja a alteração a ser realizada no `DrawerRoute`.

```
DrawerBodyApp(child: DrawerBodyContentApp(),),
```

Agora sim, suas implementações podem ser vistas em execução, mas continuaremos com o código que falta para as duas últimas opções. Veja na sequência que eles devem ser inseridos como filhos do `ListView` logo após o `ListTileTheme`, na classe `DrawerBodyContentApp`.

```
ListTile(
  contentPadding: EdgeInsets.only(left: 6.0),
  leading: CircleAvatar(
    backgroundImage: AssetImage('assets/images/drawer/jogar.png'),
  ),
  trailing: Icon(Icons.arrow_forward),
  title: Text('Jogar'),
  subtitle: Text('Começar a diversão'),
  onTap: () {},
),
ListTile(
  contentPadding: EdgeInsets.only(left: 6.0),
  leading: CircleAvatar(
```

```
    backgroundImage: AssetImage('assets/images/drawer/top10.png'),
  ),
  trailing: Icon(Icons.arrow_forward),
  title: Text('Score'),
  subtitle: Text('Relação dos top 10'),
  onTap: () {},
),
```

Se você executar sua aplicação, as três opções são exibidas e, se selecionar a primeira, verá que as duas internas também são exibidas e que, ao contrair, elas se ocultam.

5.4 Refatoração para os ListTiles

Nos quatro `ListTiles` vistos anteriormente, temos muita coisa em comum, ou seja, temos redundância de código, o que não é algo bonito e tampouco produtivo. Vamos minimizar esse problema criando uma função na classe `DrawerBodyContentApp`, com o corpo apresentado na sequência, logo após o fechamento de `build()`.

```
ListTile _createListTile({
    @required EdgeInsets contentPadding,
    ImageProvider avatarImage,
    @required String titleText,
    @required String subtitleText,
    }) {
    return ListTile(
      contentPadding: contentPadding,
      leading: avatarImage != null
          ? CircleAvatar(backgroundImage: avatarImage)
          : null,
      trailing: Icon(Icons.arrow_forward),
      title: Text(titleText),
      subtitle: Text(subtitleText),
      onTap: () {},
    );
}
```

Observe que o método `_createListTile()` retoma um `ListTile`. Nesse método, recebemos tudo o que um `ListTile` precisa saber para ser renderizado: três parâmetros obrigatórios e um opcional e a imagem para o avatar, pois temos os internos ao `ExpansionTile`, que não apresentam o `leading`. Temos ainda a função para o `onTap` não implementada, mas logo veremos isso.

Precisamos agora consumir esse método e faremos isso nos lugares em que temos a indicação dos `ListTile`. Apresentarei aqui essas chamadas para você poder alterar seu código. Lembre-se que as duas primeiras são do `ExpansionTile`.

```
_createListTile(
    contentPadding: EdgeInsets.only(left: 62.0),
    titleText: 'Novas Palavras',
    subtitleText: 'Vamos inserir palavras?',
),
_createListTile(
    contentPadding: EdgeInsets.only(left: 62.0),
    titleText: 'Palavras existentes',
    subtitleText: 'Vamos ver as que já temos?',
),

_createListTile(
    contentPadding: EdgeInsets.only(left: 6.0),
    titleText: 'Jogar',
    subtitleText: 'Começar a diversão',
    avatarImage: AssetImage('assets/images/drawer/jogar.png'),
),
_createListTile(
    contentPadding: EdgeInsets.only(left: 6.0),
    titleText: 'Score',
    subtitleText: 'Relação dos top 10',
    avatarImage: AssetImage('assets/images/drawer/top10.png'),
),
```

Talvez, lendo o código da função e as invocações anteriores, você pare e pense se, pela quantidade de código, isso vale a pena. Acredite, vale sim. É Orientação a Objetos, coesão, acoplamento.

Se formos mudar algo em relação a como os `ListTiles` devem ser exibidos, mudamos apenas na função.

Pode executar sua aplicação e ver que o resultado é o mesmo, mas estamos melhores agora. Da mesma maneira que criamos um método, poderíamos pensar em criar um widget específico para isso. Veremos algo semelhante na sequência.

5.5 Customização para os ListTiles internos

Se você notar bem em sua interface, ou na figura anterior expondo nosso Drawer e seus itens, os itens externos possuem um espaço muito grande entre eles e talvez isso não seja tão necessário. Isso é um padrão para o `ListTile`. Mesmo se o colocássemos em um `Padding` com `top: 0` e `bottom: 0`, esse espaço não diminuiria.

Vamos criar um `ListTile` personalizado para que possamos ter total controle sobre o espaço entre eles. Veja a figura a seguir para entender aonde chegaremos. À direita está nossa nova implementação.

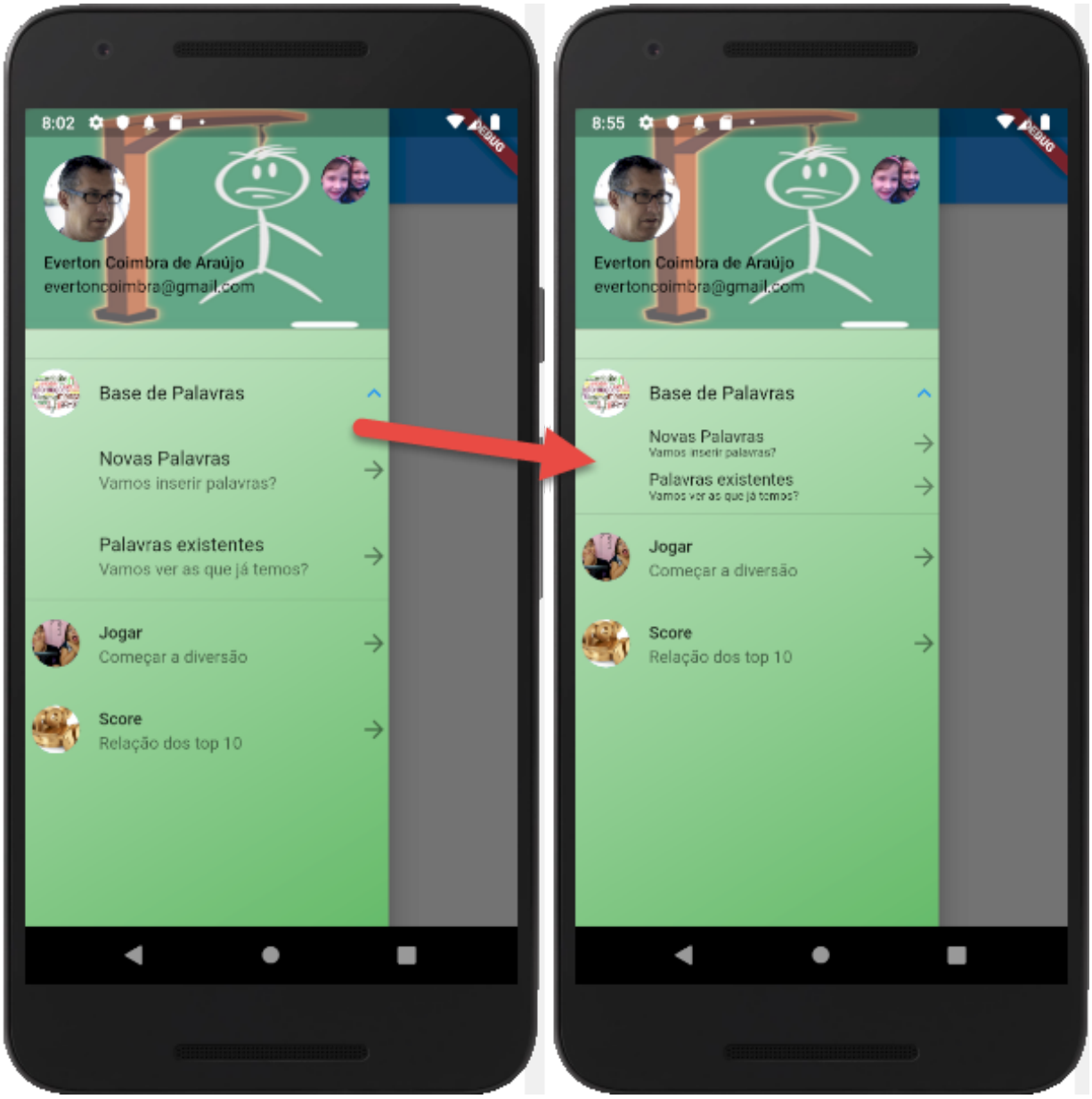

Figura 5.7: Espaço entre os ListTiles personalizado

Para que possamos criar o widget com nossa personalização, vamos criar um arquivo chamado `listtile_app_widget.dart` na pasta `widgets`, logo abaixo de `lib`. Vou trazer a implementação por partes para que possamos criá-lo aos poucos, tendo controle do que fazemos. Na sequência, o primeiro trecho da classe:

```
import 'package:flutter/material.dart';
```

```
class ListTileAppWidget extends StatelessWidget {
  final EdgeInsets contentPadding;
  final ImageProvider avatarImage;
  final String titleText;
  final String subtitleText;

  const ListTileAppWidget({
    this.contentPadding =
        const EdgeInsets.only(left: 54.0, top: 0.0, bottom: 8.0),
    this.avatarImage,
    @required this.titleText,
    @required this.subtitleText,
  });

  @override
  Widget build(BuildContext context) {
  }
}
```

Veja no código anterior que temos praticamente as mesmas propriedades utilizadas na criação do método `_createListTile()`, que refatoramos para a exibição de nossos `ListTile`. Temos algo diferente no construtor: a definição de um valor padrão, que definimos para `contentPadding`, que é opcional.

Vamos agora à primeira implementação em nosso método `build`. Veja-a na sequência e verifique que estamos utilizando o `Padding` para definir os espaços internos para nosso `ListTile` personalizado. Surpreso por termos nele um `Row`? Acredito que não, pois vendo o `ListTile` padrão, temos a abstração de uma linha.

```
return Padding(
  padding: contentPadding,
  child: Row(
    children: <Widget>[
    ],
  ),
);
```

Nosso `ListTile` padrão possui quatro propriedades básicas: `leading`, `title`, `subtitle` e `trailing`. Então, precisamos ter em nossa linha esses quatro componentes. Entretanto, no caso de `title` e `subtitle`, o segundo está abaixo do primeiro, o que nos remete a um `Column`. Veja a implementação do que temos para o `leading` na sequência, que deve ser o primeiro filho de `Row`.

```
(avatarImage == null)
  ? Container(
      width: 0,
    )
  : Container(
      width: 30.0,
      height: 30.0,
      decoration: BoxDecoration(
        shape: BoxShape.circle,
        image: DecorationImage(
          image: avatarImage,
          fit: BoxFit.cover,
        ),
      ),
    ),
```

Como fizemos para o método responsável pelo `ListTile`, verificamos se recebemos ou não um `ImageProvider`. Caso não o tenhamos recebido, o `leading` será um `Container` vazio, caso contrário, temos o `Container` com um tamanho de `30x30` e com um `decoration` em forma de círculo. É claro que poderíamos utilizar o `CircleAvatar` aqui, mas já que estamos personalizando, vamos ver também esse belíssimo recurso do `decoration`.

Nosso segundo filho para `Row` deveria ser um `Column` para o `title` e `subtitle`, mas precisamos de um comportamento diferente para esse widget. Precisamos que ele ocupe todo o espaço possível a partir de onde será renderizado. Para isso, utilizaremos o `Expanded` e ele, sim, terá o `Column` como filho. Veja o código a seguir.

```
Expanded(
    child: Column(
```

```
      crossAxisAlignment: CrossAxisAlignment.start,
      children: <Widget>[
        Text(titleText),
        Text(
          subtitleText,
          style: TextStyle(
            fontSize: 10.0,
          ),
        ),
      ],
    ),
),
```

Agora, finalizando nossa customização, precisamos implementar o que será o `trailing`. Veja o código na sequência. Note que é um ícone simples como filho de um `GestureDetector`, que dá a qualquer widget a habilidade de ser interativo com o usuário. É claro que, com a implementação que estamos fazendo, apenas quando o usuário interagir com o ícone haverá a navegabilidade para uma nova rota. Mas isso é fácil de resolver, basta colocarmos o `GestureDetector` como pai do widget `Row`, do widget que estamos implementando. Veja o que é melhor para você e altere se for necessário.

```
GestureDetector(
    child: Icon(
      Icons.arrow_forward,
      color: Colors.black38,
    ),
    onTap: () {},
)
```

Precisamos agora consumir nosso widget. Vamos fazer isso no `DrawerBodyContentApp` substituindo a invocação ao método que implementamos pelo widget que acabamos de criar. Veja na sequência essas implementações. Lembre-se de que são filhos de `ExpansionTile` e que será necessário importar o arquivo de nosso widget.

```
ListTileAppWidget(
    titleText: 'Novas Palavras',
    subtitleText: 'Vamos inserir palavras?',
),
ListTileAppWidget(
    titleText: 'Palavras existentes',
    subtitleText: 'Vamos ver as que já temos?',
),
```

Conclusão

Este capítulo foi relativamente tranquilo, aprendemos a utilizar um widget para navegação entre rotas, embora ainda não tenhamos trabalhado as rotas em si. Customizamos um widget para nossas necessidades em relação à maneira como as opções podem ser exibidas ao usuário. Também aplicamos um widget padrão para o cabeçalho do menu de opções, o que pode ser também personalizado, tal qual fizemos para nosso `ListTile`.

Criamos um widget com base em um widget oficial. Foi um processo simples, onde criamos um `ListTile` específico para o projeto, que poderia inclusive ser disponibilizado para outros programadores.

Em Flutter, alterar o comportamento de um widget já existente criando um novo é muito simples, enquanto em outras tecnologias essa situação não é tão trivial.

Apenas para enumerar o que vimos neste capítulo, que neste momento foram apenas widgets, trago aqui a relação deles, o que pode lhe auxiliar, caso queira, em uma pesquisa futura específica para cada ponto trabalhado. São eles: `Alignment`, `BoxDecoration`, `BoxFit`, `CircleAvatar`, `Drawer`, `DrawerHeader`, `Expanded`, `ExpansionTile`, `GestureDetector`, `LinearGradient`, `ListView`, `ListTile`, `ListTileTheme` e `UserAccountsDrawerHeader`.

No próximo capítulo, continuaremos no Drawer, mas vamos personalizá-lo e inserir nele uma animação bem simples mas divertida.

Capítulo 6
Abrindo o Drawer via código e uma animação com BLoC

Vimos que não precisamos realizar nenhuma implementação especial para que o `Hamburger Menu Button` fosse exibido em nossa visão. Apenas a existência de um valor na propriedade `drawer` do `Scaffold` fez tudo para nós. Mas vamos trabalhar a situação em que precisaríamos customizar isso.

Para gerar e solucionar esse problema, vamos trazer para nossa página principal uma imagem, o logo de nossa aplicação. Vamos exibi-lo na base direita da página e ver como abrir o `Drawer` da direita para a esquerda. Quando isso ocorrer, nossa imagem deverá se mover para a base esquerda para não ficar oculta e, quando o `Drawer` for recolhido, a imagem retoma para a posição inicial. Utilizaremos aqui o `BLoC Pattern` .

6.1 Contextualização sobre o problema e a solução proposta

O `Drawer` pronto, que utilizamos no capítulo anterior, não nos oferece um mecanismo de controle e identificação de quando ele é exibido/aberto e ocultado/fechado. Dessa maneira, não temos como identificar a transição de estado necessária para podermos mover a imagem do logo do app e isso é nosso problema.

Para identificar e implementar a solução, precisamos realizar uma abstração sobre o Flutter. Tudo nele é widget, tudo é componente e o `Drawer` não é diferente. O que precisamos fazer? Criar nosso próprio Drawer, e isso é tranquilo, você verá.

O nosso `DrawerRoute` segue o princípio do Material Design, pois tem um `Scaffold`, que tem um `AppBar` e um `body`. Quando o `Scaffold` possui um valor para `Drawer`, um botão é disponibilizado na `AppBar` e, ao interagir com ele, ele abre ou fecha o `Drawer`. Entretanto, se pressionarmos qualquer área fora do `Drawer`, ele também se fecha e precisamos capturar esse evento. Não podemos obrigar apenas o fechamento pela interação com o botão.

Falando ainda de nosso `DrawerRoute`, ele é um Stateless, o que impede uma atualização de renderização em seu estado, o que ocorrerá quando formos reposicionar nosso logo e exibir e ocultar o Drawer. Sendo assim, nosso primeiro passo é transformarmos nossa rota em um Stateful.

Essa transformação é relativamente simples com o Android Studio. Basta clicar sobre a palavra `StatelessWidget` e, quando for exibida a lâmpada auxiliadora da IDE, clique nela e pressione a opção de transformação para `StatefulWidget`.

6.2 Personalização do Drawer

Nosso Drawer personalizado será exatamente igual ao `Scaffold`, porém nós não faremos uso das propriedades oferecidas por ele. Ou seja, nossos `appbar`, `body` e `drawer` serão "desenhados" por nosso componente.

Lembra do `Stack`? Ele permite que diversos widgets sejam inseridos um sobre o outro em um contêiner. Verificando o comportamento do `Drawer`, ele é exibido acima do `Scaffold` e seus componentes. Vamos iniciar a criação de nosso componente.

Na nossa pasta `lib\drawer\widgets`, crie um arquivo Dart chamado `drawer_controller_widget.dart` e coloque o código inicial a seguir nele.

```
import 'package:flutter/material.dart';

class DrawerControllerWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) => Scaffold(
        body: Stack(
          children: [
            Positioned(
              top: 0.0,
              left: 0.0,
              right: 0.0,
              child: AppBar(),
            ),
          ],
        ),
      );
}
```

Temos um widget novo na listagem anterior, o `Positioned`. Esse widget, quando utilizado em um `Stack`, nos possibilita explicitar o local exato onde queremos que o widget filho (`child`) seja desenhado, com base na orientação do dispositivo. Em nosso caso, é no topo esquerdo, ou seja, `top: 0` e `left: 0`. O `right: 0` precisa ser especificado para que possamos inserir os componentes da `AppBar`. O widget filho de `Positioned` neste momento para nosso Drawer é um `AppBar`, apenas para podermos ver a rota renderizada na aplicação em execução.

Observou que como temos apenas uma instrução a ser executada pelo `build`, que é o `return` para o `Scaffold`, adotei o uso de `arrow function` (`=>`)? É um recurso cada vez mais utilizado em diversas linguagens.

Agora precisamos mudar nosso `build` em nosso `DrawerRoute` como veremos na sequência e, é claro, precisamos importar o novo controle para nosso arquivo. Você pode digitar `import 'widgets/drawer_controller_widget.dart';` logo no início ou usar os recursos do IDE para essa função. Sei que estamos trocando todo o conteúdo por um único widget. Está tudo certo.

```
@override
Widget build(BuildContext context) {
    return DrawerControllerWidget();
```

Prefiro economizar espaço aqui e não exibir a figura, mas teste sua aplicação e veja que é renderizada uma rota em branco no dispositivo, com o `AppBar` em azul.

Queremos que quem consuma nosso widget possa personalizar sua `AppBar`, o que nos leva a não definir valores para os parâmetros desse widget em nossa personalização. Ofereceremos então a quem consumir nosso widget a possibilidade de enviar um `AppBar` pelo construtor. Sendo assim, insira antes do `build()` do `DrawerControllerWidget` as instruções a seguir.

```
final AppBar appBar;

const DrawerControllerWidget({this.appBar});
```

Na listagem, estamos definindo uma propriedade para nossa propriedade chamada `appBar` do tipo `AppBar` e criando um construtor para que essa propriedade possa ter esse valor inicializado, uma vez que a propriedade é `final`. Precisamos conseguir utilizar essa propriedade em nosso código. Para isso, no `child` do `Positioned`, insira o código a seguir:

```
child: (appBar == null) ? AppBar() : appBar,
```

Quando o `appBar` for invocado, caso seja nulo, uma instância de `AppBar` é retomada. Fica a seu critério o que utilizar.

Veja o operador ternário no código anterior. Caso nenhum `AppBar` seja recebido, utilizamos um padrão. Notou as `{}` no construtor? Elas tornam este parâmetro opcional, certo?

O Dart nos traz um elegante operador para uma situação como a anterior, que pode ser utilizado no lugar do operador ternário. É o `coallesce operator` ou `null-aware`. Veja o código a seguir:

```
child: appBar ?? AppBar()
```

Vamos agora adaptar a invocação de nosso widget no `DrawerRoute`. Veja a nova implementação para o `build()` na sequência. Após a implementação, teste novamente sua aplicação e veja o `AppBar` com o visual que desejamos pronto.

```
@override
Widget build(BuildContext context) {
    return DrawerControllerWidget(
      appBar: AppBar(
        automaticallyImplyLeading: false,
        title: Text('Jogo da Forca'),
        centerTitle: true,
        actions: <Widget>[
          Icon(
            Icons.menu,
            size: 40,
          ),
        ],
      ),
    );
}
```

No código anterior, usamos uma propriedade ainda não vista, a `automaticallyImplyLeading`. Ela é responsável por desenhar ou não o ícone/botão de retornar à rota chamadora. No capítulo anterior, não a configuramos, pois estávamos utilizando a propriedade `drawer`, que desenhava automaticamente o botão de menu. Se agora deixarmos o padrão, que é `true`, o botão de retornar será desenhado e não queremos isso.

Nossa próxima etapa é desenhar o corpo (`body`) que será renderizado pelo nosso controle, então, precisamos parametrizar nosso controle para receber esse widget. Veja a seguir a implementação da nova propriedade em `DrawerControllerWidget` e a adaptação do construtor para recebê-la.

```
final AppBar appBar;
final Widget body;
```

```
const DrawerControllerWidget({this.appBar, this.body});
```

Com a propriedade recebida por nosso widget, precisamos agora poder renderizá-la e, para isso, adicionaremos o código a seguir logo após o `Positioned` do `AppBar`.

```
Positioned(
  top: MediaQuery.of(context).size.height - 105,
  left: MediaQuery.of(context).size.width - 105,
  child: (body == null) ? Container() : body,
),
```

Temos algo novo no código anterior, o `MediaQuery`, que é uma classe que nos retorna informações relativas ao dispositivo em que a aplicação está sendo executada. Verifique que estamos obtendo a altura (`height`) e tamanho (`width`) de nossa interface. Com essas informações em mãos, retiramos o tamanho dedicado a nosso widget e o atribuímos ao topo e à esquerda, o que fará com que o logo do app seja exibido na base da tela ao lado direito, que é o nosso objetivo. No `child` anterior, poderíamos também utilizar o `coallesce operator`.

Na sequência, para podermos ver nossa aplicação em execução com o comportamento comentado, precisamos adaptar nosso `DrawerRoute` e inserir o código a seguir após o envio do `AppBar`. Lembre-se de importar o `CircularImageWidget`.

```
body: CircularImageWidget(
    imageProvider: AssetImage('assets/images/splashscreen.png'),
    width: 100,
    heigth: 100,
),
```

Com a implementação realizada, execute sua aplicação. O resultado deve ser semelhante ao apresentado na figura a seguir.

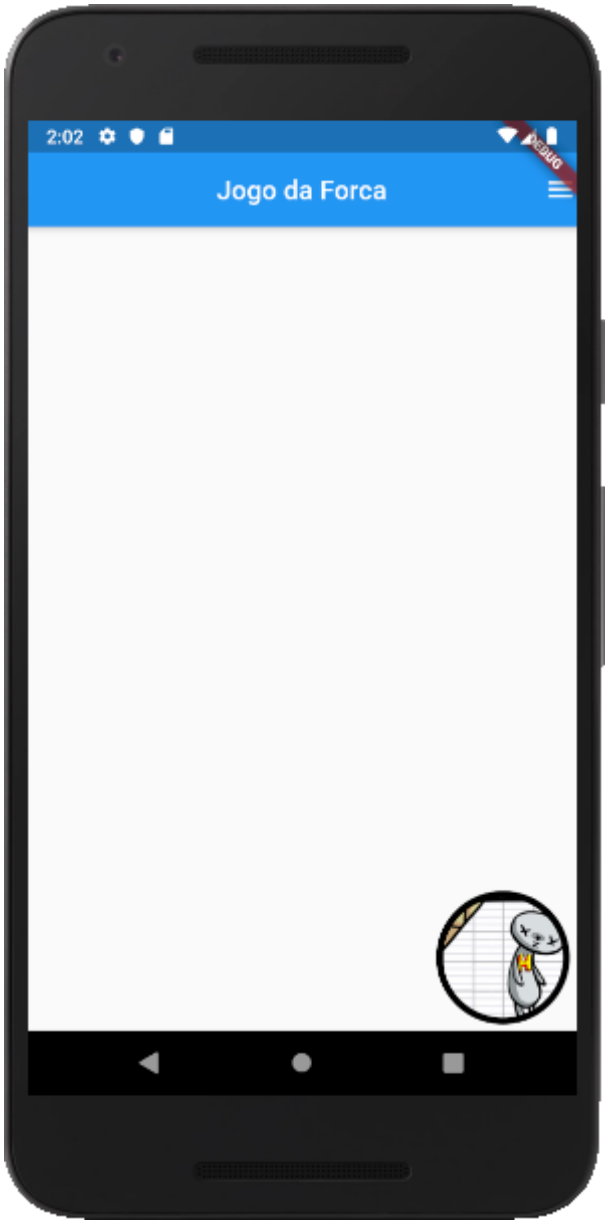

Figura 6.1: Drawer personalizado exibindo o logo do app

Temos duas situações para pensar com esse resultado. O primeiro está visivelmente claro em nossa execução, que é o banner de `debug`, característico de aplicações Flutter em execução em modo de depuração. Isso podemos resolver facilmente, basta inserirmos o parâmetro a seguir em nosso arquivo `main.dart`, quando invocarmos o `MaterialApp()`. Veja o resultado em sua aplicação. Quando fizer o deploy de sua aplicação, automaticamente esse banner desaparece, sem necessidade deste código.

```
child: MaterialApp(
    debugShowCheckedModeBanner: false,
    // código omitido
}
```

A segunda situação que temos e que deve realmente gerar preocupação é a dimensão e posicionamento de nosso widget que representam o corpo da rota renderizada. Da maneira que fizemos, estamos praticamente obrigando aqueles que consumirão nossa aplicação a utilizar um objeto posicionado na pilha na posição que queremos. Ou seja, estamos engessando nosso controle. E se aqueles que utilizarem nosso widget quiserem exibir um objeto ocupando toda a tela? Precisamos então fazer igual ao que fizemos para o `AppBar`.

Entretanto, nosso `DrawerRoute` não tem um `Stack`, e o `Positioned` só pode ser utilizado em um `Stack`, um problema para refletir, mas fácil de resolver. Vamos mandar o posicionamento para o nosso Drawer personalizado e, com base nele, tomaremos a decisão de como renderizar o corpo do controle. Veja na sequência nossas novas propriedades e o novo construtor para `DrawerControllerWidget`.

```
final double topBody;
final double leftBody;

const DrawerControllerWidget(
  {this.appBar, this.body, this.topBody, this.leftBody});
```

Agora vamos à readaptação para a renderização de nosso `body`. Observe no código que utilizaremos o `Positioned` apenas se um dos valores de posicionamento for recebido, caso contrário, renderizamos o componente com as características que forem enviadas. Estamos substituindo o código de `body`, ok?

```
(this.topBody != null || this.leftBody != null)
    ? Positioned(
        top: this.topBody,
        left: this.leftBody,
        child: body ?? Container(),
```

```
      )
    : body,
```

Vamos então adaptar o `DrawerRoute` para esse novo comportamento, tal qual vemos a seguir. Substitua o atual `body` pelas instruções do código.

```
topBody: MediaQuery.of(context).size.height - 105,
leftBody: MediaQuery.of(context).size.width - 105,
body: CircularImageWidget(
    imageProvider: AssetImage('assets/images/splashscreen.png'),
    width: 100,
    heigth: 100,
),
```

Você pode realizar alguns testes no código anterior e ver que o comportamento é o mesmo apresentado na figura anteriormente exibida. Comente as propriedades de posicionamento. Veja o que ocorre com o logo do app. Envolva o `body` em um `Center` e veja o que também ocorre. Você pode também tirar o comentário de um dos parâmetros do posicionamento e retirar o `Center` . Assim deixamos nosso widget menos acoplado e mais coeso. Uma modificação que sugiro a você é deixar o tamanho do logo em uma variável para utilizá-lo de maneira coerente no posicionamento dele para nosso widget.

A última parte de nossa customização é inserir o Drawer em nosso widget. Nós já o temos todo pronto do capítulo anterior, resta-nos apenas utilizá-lo.

Novamente, quem utilizar nosso widget poderá ter a liberdade de enviar o Drawer como quer que ele seja exibido. Seguindo o que já estamos acostumados, vamos parametrizar nossa personalização. Veja a nova propriedade para `DrawerControllerWidget` na sequência, assim como o novo construtor.

```
final Drawer drawer;
```

```
const DrawerControllerWidget(
  {this.appBar, this.body, this.topBody, this.leftBody, this.drawer});
```

Com a possibilidade de recebermos o Drawer, já podemos configurar sua renderização. Veja o código a seguir e já vamos comentá-lo. É preciso inseri-lo após nosso `body`, em `DrawerControllerWidget`.

```
DrawerController(
  alignment: DrawerAlignment.end,
  child: drawer != null ? drawer : Container(),
),
```

Estamos utilizando um novo widget, o `DrawerController`. O Drawer que utilizamos no capítulo anterior é baseado neste, mas ele não nos oferece mecanismos para identificar sua mudança de estado e, conforme contextualizado no início do capítulo, foi isso o que nos trouxe a essa personalização que estamos fazendo.

Na sequência, vamos enviar nosso Drawer pelo `DrawerRoute` para nosso widget. Veja o código. Não temos nada de novo, é tudo o que vimos no capítulo anterior. Insira este código após o `body`.

```
drawer: Drawer(
    child: Column(
      children: <Widget>[
        DrawerHeaderApp(),
        DrawerBodyApp(
          child: DrawerBodyContentApp(),
        ),
      ],
    ),
),
```

Ao executarmos nossa aplicação, não vemos ainda nosso Drawer sendo exibido. Isso tem explicação. Nosso `Drawer Menu` não está interceptando os gestos do usuário com o dispositivo. No capítulo anterior, isso era implícito. O Flutter gerenciava tudo isso diretamente pelo `Scaffold` e pela propriedade `drawer`. Agora nós chamamos a responsabilidade para nós.

Nós precisaremos realizar algumas alterações em nosso componente, pois a abertura do Drawer se dará dentro dele. Da maneira que estamos enviando o `AppBar`, a captura de interação deveria ser configurada no `DrawerRoute`, mas ele não conhece o comportamento de nosso componente. Veja na sequência a nova configuração para renderização do `AppBar` recebido em `DrawerControllerWidget`.

```
child: (appBar == null)
  ? AppBar()
  : AppBar(
      automaticallyImplyLeading:
          appBar.automaticallyImplyLeading,
      title: appBar.title,
      centerTitle: appBar.centerTitle,
      actions: <Widget>[
        appBar.actions[0],
      ],
    ),
```

Observe que configuramos o `AppBar` do nosso widget com as propriedades que recebemos do `AppBar` enviado pelo `DrawerRoute`. Precisamos então configurar agora nossa `action` para capturar gestos do usuário e faremos isso com a adaptação a seguir no código.

```
actions: <Widget>[
    GestureDetector(
      child: appBar.actions[0],
    ),
],
```

Aqui precisamos ter uma atenção especial com quem consumirá nosso controle. Caso o `GestureDetector` seja implementado no envio do `AppBar`, é o `onTap` dele que será executado, e não o definido em nosso controle. Ou seja, é preciso documentar bem seu componente para isso.

Mas o que é o `GestureDetector`? É um widget que dá a qualquer outro a possibilidade de receber interações do usuário por meio de seus gestos. Nós estamos utilizando aqui apenas o `onTap`, mas existem vários, que você pode e deve dar uma investigada.

Tudo bem, mas e a abertura do Drawer? Precisamos realizar ainda algumas configurações em nosso componente para podermos interagir com o `DrawerController`.

6.3 Abertura, fechamento e estado no Drawer

Temos nosso componente pronto para renderizar um Drawer personalizado. Falta-nos apenas fazê-lo ser exibido. É importante sabermos que o processo de abertura e fechamento (ou exibição e ocultação) do Drawer que utilizamos no capítulo anterior é controlado por uma classe chamada `DrawerControllerState`. Nós não nos preocupamos com ela antes, pois tudo foi gerenciado pelo Flutter. Agora é diferente.

Para podermos controlar o estado de um componente em específico, como o caso de nosso Drawer personalizado, precisamos ter uma maneira de resgatar esse controle na árvore de widgets (*Widgets Tree*) e uma das maneiras mais simples é associarmos ao controle uma chave, conhecida por `GlobalKey`. A documentação do Flutter diz que o uso em excesso de chaves desse tipo pode ser custoso para a aplicação, pois será garantida uma chave única para cada componente.

Existem outras maneiras de identificar o widget que buscamos, como percorrer a árvore de widgets em todo o contexto até identificá-lo, mas a implementação é trabalhosa e, neste momento, adotaremos a `Global Key`. Veja o código a seguir que, por convenção, vamos implementar após o construtor. É uma técnica que utilizo para propriedades que não sejam recebidas pelo

construtor. Insira o código após o construtor de `DrawerControllerWidget`.

```
GlobalKey<DrawerControllerState> drawerKey =
GlobalKey<DrawerControllerState>();
```

A implementação anterior, que define uma chave global específica para um `DrawerControllerState`, causará o surgimento de um erro na linha de nosso construtor, que não poderá ser mais constante, pois o objeto a ser renderizado pode ter uma de suas propriedades alteradas a qualquer tempo do ciclo de vida do objeto. Podemos resolver retirando `const`, que precede o nome do construtor.

Agora que temos uma chave única para nosso `DrawerController`, precisamos atribuí-la a ele. Veja isso na sequência. A única alteração está na definição de `key`.

```
DrawerController(
  key: drawerKey,
  alignment: DrawerAlignment.end,
  child: drawer != null ? drawer : Container(),
),
```

Com isso, podemos controlar o estado de nosso Drawer, o que nos leva então a implementar um método capaz de exibi-lo. Veja-o na sequência, ele pode ser implementado antes do `build()`.

```
void _openDrawer() {
    drawerKey.currentState.open();
}
```

No código anterior, obtemos o estado atual do `DrawerController` por meio de nossa `Global Key` e então invocamos o método `open()` para que o Drawer seja exibido.

Precisamos agora invocar esse método no `onTap` de nosso `GestureDetector`. Veja essa implementação na sequência. Novamente, uma única mudança foi realizada no código, a que registra o `onTap`.

```
actions: <Widget>[
    GestureDetector(
      child: appBar.actions[0],
      onTap: () => _openDrawer(),
    ),
],
```

Podemos testar nossa aplicação agora e ver o `Drawer` sendo exibido no momento em que interagimos com o `Hamburger Menu` e vê-lo fechar quando clicamos fora, o que é o padrão que vimos no exemplo anterior. Veja na figura a seguir o Drawer em exibição.

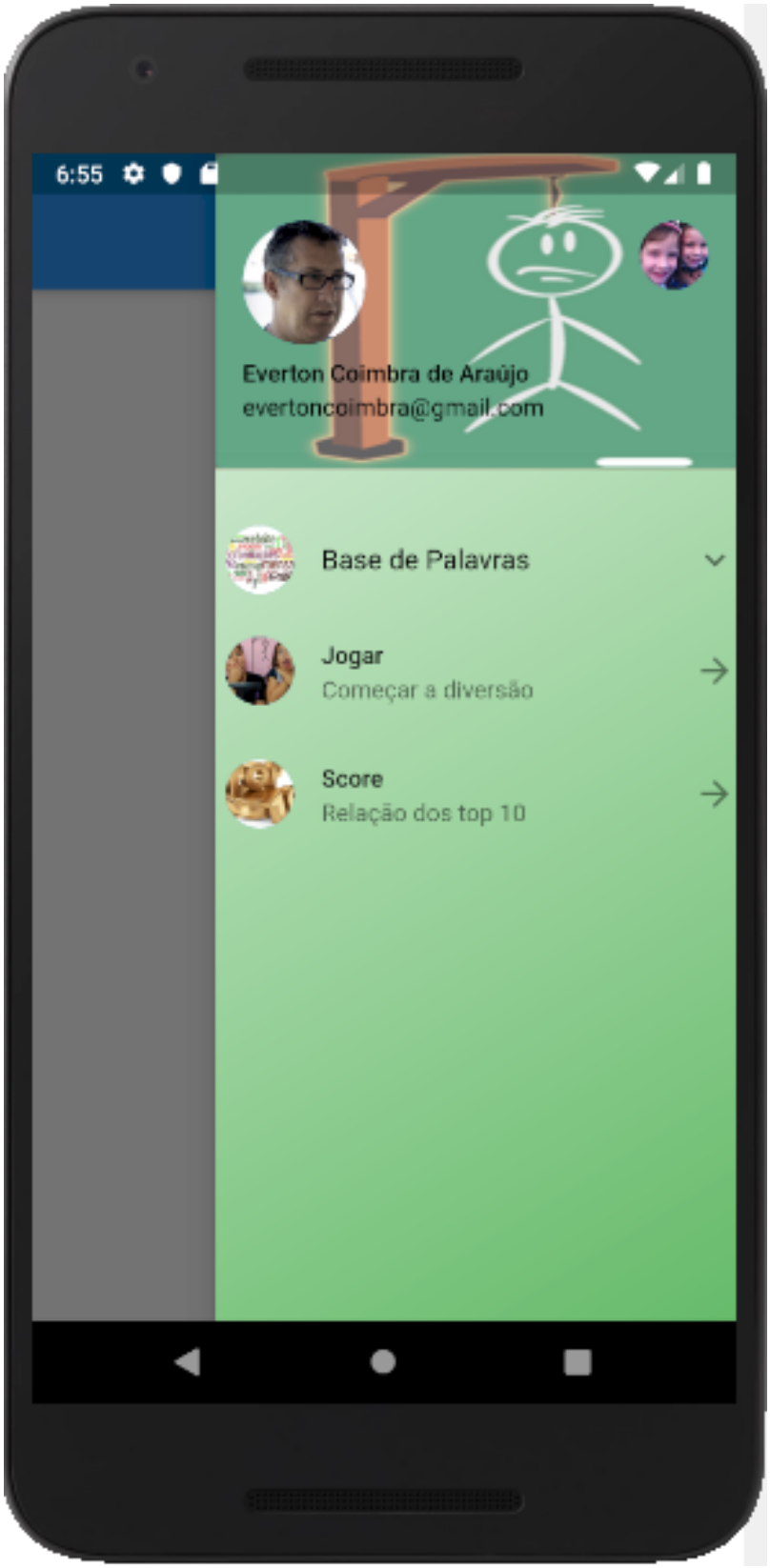
6:55
Everton Coimbra de Araújo
evertoncoimbra@gmail.com
Base de Palavras
Jogar
Começar a diversão
Score
Relação dos top 10

Figura 6.2: Drawer personalizado sendo exibido

Tudo perfeito, funcionando direitinho, mas e a animação para o logo do app que comentamos? Veremos na próxima seção.

6.4 Animação do logo do app

Contextualizando nosso problema de animação, podemos identificar uma coisa bem simples: precisamos mudar a posição em que nosso logo é exibido, horizontalmente, o que é apontado pelo parâmetro `leftBody`, que enviamos ao construtor. Esse valor será um quando o Drawer estiver oculto e outro quando o Drawer estiver em exibição.

Com isso, levantamos outro problema: como saber quando o Drawer está sendo exibido ou fechado? Exibido pode ser fácil, pois poderíamos interceder no método `_openDrawer()`, que implementamos na seção anterior, mas e quando ele for fechado? Lembra que isso foi o real motivo de personalizarmos nosso componente? Veja a nova implementação para o `DrawerController` na sequência.

```
DrawerController(
  key: drawerKey,
  alignment: DrawerAlignment.end,
  child: drawer != null ? drawer : Container(),
  drawerCallback: (status) => _drawerCallback(status),
),
```

Observou a inserção da propriedade `_drawerCallback`? O nome dela é semântico. Podemos implementar uma função que vai ser executada quando o estado do Drawer se alterar. É disso que precisamos. Veja que a função tem um valor como entrada, que é um `bool`, nomeado aqui como `status`, o qual enviamos para uma função que ainda precisamos implementar antes de `build()`, a `_drawerCallback()`, que está na sequência.

```
void _drawerCallback(bool status) {
}
```

Reparou que ainda não temos nenhum comportamento implementado na função anterior? Pois bem, o que acontece é que, quando ocorrer uma mudança de estado, precisamos mudar a posição do nosso logo, que está definida no widget que está utilizando nosso Drawer personalizado. Então é no widget cliente que precisamos implementar essa função, mas não podemos com a estratégia que estamos utilizando. O que fazemos? Implementamos a função no cliente, a enviamos para o `DrawerPersonalizado` e a invocamos nele. Vamos começar esta implementação.

Nossas primeiras adaptações estão relacionadas às posições enviadas ao nosso Drawer. Elas não podem ser estáticas, pois seus valores vão variar de acordo com o estado do Drawer, o que nos levará a implementar duas funções antes do `build()` de `DrawerRoute`, veja-as na sequência.

```
bool _drawerIsOpen = false;

double _topBody() {
    return MediaQuery.of(context).size.height - 105;
}

double _leftBody() {
    if (!_drawerIsOpen)
      return MediaQuery.of(context).size.width - 105;
    else
      return 5;
}
```

Verifique que estamos utilizando uma variável que será o *flag* do valor retomado para a `leftBody`. Isso é provisório, apenas para fins didáticos do que estamos implementando. Com essas adaptações, precisamos alterar o envio das propriedades para o Drawer personalizado em nosso `DrawerRoute`. Veja a atualização na sequência.

```
topBody: _topBody(),
leftBody: _leftBody(),
```

Precisamos agora de uma função que seja executada no `DrawerRoute` e que altere o valor de `_drawerIsOpen`, de acordo com o estado do Drawer que estamos implementando. Veja esta função a seguir também no `DrawerRoute`, ela pode ser implementada antes de `build()`.

```
_handleDrawer(bool drawerIsOpen) {
    setState(() {
      this._drawerIsOpen = drawerIsOpen;
    });
}
```

Essa função deve agora ser enviada ao nosso widget para que ela seja invocada no momento de execução do método de `callback`. Para isso, precisamos alterar as propriedades de nosso componente `DrawerControllerWidget` e também seu construtor, como no código a seguir.

```
final Function callbackFunction;

DrawerControllerWidget({this.appBar, this.body, this.topBody,
this.leftBody, this.drawer, this.callbackFunction});
```

Em nosso `DrawerRoute`, precisamos enviar nossa função `_handleDrawer()` com a instrução a seguir após o envio de `drawer` para nosso `DrawerControllerWidget`.

```
return DrawerControllerWidget(
    // Demais argumentos
    callbackFunction: _handleDrawer,
);
```

Agora nos resta implementar o comportamento para a função de `callback`, que está em `DrawerControllerWidget`, e que podemos ver isso na sequência.

```
void _drawerCallback(bool status) {
    callbackFunction(status);
}
```

Execute agora sua aplicação e observe que o logo foi reposicionado, mas não é uma animação e, o pior, nosso Drawer não foi exibido. A segunda situação ocorreu porque estamos chamando o `setState()`, que vai renderizar toda a árvore de widgets e gerará novamente nosso componente, que, por padrão, não exibe o Drawer. Se você verificar o console, verá que erros foram exibidos. Logo resolveremos isso.

Vamos nos concentrar agora na animação que queremos. O Flutter é fantástico para animações, pois tem um componente próprio para isso, o `AnimatedPositioned`. Existem diversas animações, mas a que queremos agora é bem simples. A mudança é simples. Em nosso `DrawerControllerWidget`, substitua o `Positioned` do `body` pelo código a seguir.

```
(this.topBody != null || this.leftBody != null)
    ? AnimatedPositioned(
        duration: Duration(seconds: 1),
        top: this.topBody != null ? this.topBody : null,
        left: this.leftBody != null ? this.leftBody : null,
        child: (body == null) ? Container() : body,
      )
    : body,
```

Veja que, além de mudarmos o widget, inserimos o `duration`, especificando um segundo para o período final de animação. Vamos conversar um pouco sobre isso.

Uma animação é composta por dois estados, o inicial e o final. O que ocorre entre esses dois estados é chamado de interpolação e essa interpolação ocorrerá em um segundo pelo nosso código. Você pode brincar com esses valores e com os argumentos de `Duration`, fique à vontade. Voltando à interpolação, quem a controlará com esse componente é o Flutter. Bem simples, não é?

Teste sua aplicação, tente exibir o Drawer e veja a animação. Na próxima seção, trabalharemos a exibição de nosso Drawer, que ainda não está aparecendo.

6.5 Atualização de estado com BLoC

Temos uma situação em que precisamos renderizar nossa visão com base em uma mudança de estado. Sabemos que isso pode ser facilmente resolvido por meio da invocação do método `setState()`, já conhecido e comumente utilizado nesses casos. Ocorre que, neste problema em particular, nosso Drawer, desenhado por um componente que é parte daquele que terá a renderização realizada, deixará de ser exibido pela invocação do `setState()`.

Existem diversas alternativas para gestão do estado de widgets além do `setState()` e algumas são mais eficientes que outras. Já trabalhei com `Scoped Model`, sobre o qual você pode ler em https://medium.com/flutter-community/flutter-architecture-scopedmodel-a-complete-guide-to-real-world-architecture-205a24674964, e também com o BLoC. Recentemente o MobX tem sido bem comentado, e um bom material sobre ele pode ser acessado em https://medium.com/flutterando/gerenciamento-de-estado-no-flutter-o-uso-do-mobx-a71c5dc3b6ca. No livro, veremos o BLoC e o MobX.

Seguido de perto pelo MobX, o mais utilizado até o momento é o BLoC e é ele que utilizaremos agora para resolvermos o problema com o qual nos deparamos para uso de nosso Custom Drawer.

BLoC é a abreviação de *Business Logic Component*, que pode ser semanticamente traduzido para a separação da regra de negócio e interface do usuário em nossos componentes.

Um conceito, uma técnica ou um recurso por trás do BLoC é *Streams*, que nada mais são do que fluxos. Cada vez mais, as

aplicações têm se tornado reativas, adaptativas à forma como o usuário interage com ela. Podemos mapear isso para Streams, onde temos um fluxo de entrada e um fluxo de saída. Uma mudança de estado, por exemplo, pode ser nosso fluxo de entrada e o novo estado pode ser o fluxo de saída. Basicamente, BLoC se encaixa com o problema que precisamos solucionar.

Não implementaremos Streams diretamente aqui no livro. Nosso foco é mais o consumo de recursos que os implementem, como o uso de BLoC. Dessa maneira, não implementaremos BLoC "na raça", usaremos componentes prontos, que nos possibilitam esse recurso de uma maneira menos custosa para a implementação.

O primeiro componente é o `bloc`, que está disponível em https://pub.dev/packages/bloc, com informações e exemplos. Você verá que sua utilização é muito simples. Esse componente é uma biblioteca Dart para implementação do `BLoC Pattern`. Entretanto, como implementaremos o BLoC na renderização de widgets, podemos usar direto o componente `flutter_bloc`, que já traz o `bloc` consigo e está disponível em https://pub.dev/packages/flutter_bloc. Vamos começar aqui a implementação de nosso BLoC.

Seguindo nossa organização de pastas e pacotes, vamos criar uma nova pasta chamada `blocs` na pasta `lib\drawer`. Dentro dela, criaremos dois arquivos: um chamado `drawer_bloc_enums.dart` e outro chamado `drawer_bloc.dart`. Veja na sequência o código para o primeiro arquivo.

```
enum DrawerControllerEvent { open, close }
```

É apenas a declaração de um `enum` com nossas possíveis mudanças de estado. Poderíamos ter isso no mesmo arquivo do BLoC, mas, para um baixo acoplamento, é melhor separar.

Antes de termos a implementação de nosso segundo arquivo com o BLoC, precisamos instalar o componente em nosso projeto. Para isso, no `pubspec.yaml`, insira `flutter_bloc: ^4.0.0` abaixo de `dependencies`, onde `^4.0.0` é a versão disponível no momento da

escrita deste livro. Em seguida, clique em `Packages get` no topo da janela do editor para que o componente possa ser instalado. Pode ser bom parar a execução total do aplicativo neste momento, caso esteja em execução.

Agora sim, vamos para a implementação de nosso segundo arquivo. Veja a implementação inicial na sequência. Após o código, temos algumas explicações.

```
import 'package:bloc/bloc.dart';
import 'drawer_bloc_enums.dart';

class DrawerOpenStateBloc extends Bloc<DrawerControllerEvent, bool> {
  @override
  bool get initialState => false;

  @override
  Stream<bool> mapEventToState(DrawerControllerEvent event) async* {
    switch (event) {
      case DrawerControllerEvent.open:
        yield true;
        break;
      case DrawerControllerEvent.close:
        yield false;
        break;
    }
  }
}
```

Vemos uma classe normal no código anterior, que estende `Bloc`, fazendo uso de *generics* para dizer que o argumento para nosso fluxo de entrada deve ser um valor do tipo `DrawerControllerEvent`, e o tipo de dado para o fluxo de saída um `bool`.

Ao estendermos `Bloc`, precisamos sobrescrever dois métodos. Um é um `get` para o `initialState`, que na realidade é uma propriedade que definirá o valor inicial que nosso BloC terá quando for instanciado. Em nosso caso é `false`, pois inicialmente nosso Drawer estará fechado.

O segundo método que sobrescrevemos no código anterior é o `mapEventToState`, que recebe o evento que está acontecendo, e, com base nele, o fluxo de saída ocorre. Traduzindo, quando abrirmos o Drawer, um `true` será retornado e, quando fecharmos, um `false`.

Observe o tipo de dado retornado pelo segundo método. É um `Stream` tipificado para `bool`. Após o fechamento dos parênteses para o método, temos `async*`, reforçando que o retorno será dado por `yield` e não por `return`, por estarmos trabalhando com Streams. Caso queira se aprofundar em Streams, uma boa leitura pode ser feita em https://medium.com/flutter-community/understanding-streams-in-flutter-dart-827340437da6.

Precisaremos realizar algumas mudanças em nosso código para que a implementação com BLoC funcione perfeitamente. A primeira delas será registrar a mudança de estado em nosso Custom Drawer. Neste momento, você deve estar se perguntando: "Mas nosso widget é Stateless, como vou registrar mudanças de estado?" Esta é a mágica do BLoC.

Nossa primeira mudança será alterar os métodos que retornam as posições para `top` e `left`. Como estamos trabalhando apenas com `left`, vou me preocupar apenas com ela, mas é interessante você realizar o mesmo para `top`, o que deixaria seu widget mais independente. Veja o novo código da declaração dos métodos na sequência em `DrawerRoute`.

```
double _leftBodyOpen() {
    return 5;
}

double _leftBodyClose() {
    return MediaQuery.of(context).size.width - 105;
}
```

Podemos remover o `_leftBody()` e o `_handleDrawer()`, pois todos esses registros farão parte do nosso controle. Precisamos também ajustar nossas propriedades do controle e o construtor para receber

os dois novos valores. Veja na sequência apenas a declaração deles em nosso controle `DrawerControllerWidget` e ajuste o construtor.

```
final double leftBodyOpen;
final double leftBodyClose;
```

Retire a propriedade `_leftBody` também do construtor e insira as duas novas propriedades nele. Será preciso alterar a nossa nova implementação em `DrawerRoute`. Retire também o parâmetro `_leftBody`, pois não teremos mais a propriedade, e insira os dois novos parâmetros e também a `callbackFunction`.

Vamos ajustar nosso método `drawerCallback()`, como apresentado na sequência. Você precisará inserir imports para o `flutter_bloc` e as classes de `bloc` criadas anteriormente.

```
void _drawerCallback(bool status) {
    BlocProvider.of<DrawerOpenStateBloc>(this.context)
        .add(status ? DrawerControllerEvent.open :
DrawerControllerEvent.close);
}
```

Observou que temos o `this.context`? Pois é, precisamos declarar essa propriedade abaixo de nosso `drawerKey`, como apresentado na sequência.

```
BuildContext context;
```

Precisamos atualizar esse `context` durante a renderização do nosso controle. Sendo assim, insira logo no início de `build` a instrução a seguir. Se seu método `build` estiver em `arrow function`, transforme-o em `body function`.

```
this.context = context;
```

Já temos nosso registro de transição entre estados realizado. Resta-nos agora capturar essa transição e renderizar nossa animação corretamente. Para isso, precisamos encapsular nosso `AnimatedPositioned` com um `BlocBuilder`. Veja esta alteração na sequência.

```
(this.topBody != null)
  ? BlocBuilder<DrawerOpenStateBloc, bool>(
      builder: (context, isDrawerOpen) {
      double left =
          isDrawerOpen ? this.leftBodyOpen : this.leftBodyClose;
      return AnimatedPositioned(
        duration: Duration(seconds: 1),
        top: this.topBody,
        left: left,
        child: (body ?? Container()),
      );
    })
  : body,
```

Veja que recebemos do BLoC o seu estado atual, que foi registrado pelo `drawerCallback`. Criamos uma variável para `left`, que terá seu valor dependente do estado recebido para o Drawer, e ajustamos a propriedade `left` do `AnimatedPositioned`.

O `BlocBuilder` é um widget que buscará pelo BLoC informado como generic no contexto da aplicação. Mas nós ainda não o temos. Esse registro precisa ser sempre realizado antes do uso do `BlocBuilder`. Eu tenho como hábito realizar isso diretamente no `main`, ou, quando a aplicação tem muitos níveis, em um nível anterior ao seu consumo. Veja na sequência a implementação realizada no `main`.

```
void main() => runApp(
      MultiBlocProvider(providers: [
        BlocProvider<DrawerOpenStateBloc>(
          create: (BuildContext context) => DrawerOpenStateBloc(),
        ),
      ], child: ForcaApp()),
    );
```

Estamos utilizando o `MultiBlocProvider` para registrar nosso BLoC pelo `BlocProvider`. Se tivermos certeza de que apenas um BLoC será registrado, não precisamos do primeiro, mas, para facilitar novos registros, já prefiro utilizá-lo. Já podemos testar nossa aplicação. Verifique o comportamento do nosso logo ao abrir e

fechar o Drawer, sem o erro que tínhamos antes em relação ao `setState()`.

Para finalizar, em relação à propriedade `callbackFunction` do nosso componente, que retiramos, você pode pensar em deixá-la como opcional, pois pode haver necessidade de executar algo no cliente quando a transição de estado do Drawer ocorrer. Eu optei por retirar a função do parâmetro de construção do widget em `DrawerRoute`, assim como eliminar essa função como você viu nas leituras anteriores.

Caso você queira, é possível tornar nosso `DrawerRoute`, que está como Stateful, em Stateless, pois não há mais a necessidade de gerir o estado dele. Mas isso fica contigo.

Conclusão

Trabalhamos bastantes técnicas neste capítulo. Ele foi um pouquinho mais extenso que os anteriores, mas tenho certeza de que valeu a pena. Criamos um componente com certa complexidade, inserimos animação em nossa aplicação e ainda vimos o gerenciamento de estado por meio de BLoC. Reforço ainda o comentário da conclusão do capítulo anterior: poderíamos pensar em disponibilizar esse componente para outros programadores. No próximo capítulo, trabalharemos a navegação entre rotas para as opções escolhidas em nosso Drawer.

Apenas para enumerar o que vimos neste capítulo, que neste momento foram apenas widgets, trago aqui a relação deles, o que pode auxiliar, caso queira, em uma pesquisa futura específica para cada ponto trabalhado. São eles: `AnimatedPositioned`, `BLoC`, `DrawerController`, `DrawerControllerState`, `Duration`, `MediaQuery` e `Positioned`.

CAPÍTULO 7
Rotas, transições entre elas e o formulário para palavras

Nós já implementamos navegação entre rotas em nossa aplicação, da splash screen para a de boas-vindas e depois para nossa home, representada pelo nosso Drawer. Até aqui tudo bem, mas por meio de recursos oferecidos pelo Flutter podemos gerenciar a navegação entre rotas em um único local e usar rotas nomeadas, o que pode facilitar muito nossa arquitetura e a aplicação de técnicas.

Para aplicação desses conteúdos comentados, criaremos a rota responsável pelo registro das palavras que serão utilizadas em nosso jogo e implementaremos um formulário nessa rota usando novamente o BLoC. Esse formulário possuirá um modelo, uma classe, e nela usaremos anotações JSON, obtenção e geração de objetos a partir e de um mapa de dados no formato JSON. Teremos novos componentes para isso. Também veremos `Mixin`, um recurso utilizado na herança, que possibilita uma pseudo-herança múltipla por extensão. Há muitas coisas para aprendermos, então vamos lá.

7.1 A classe modelo para o formulário

Sempre que implementamos a solução para um problema, identificamos o modelo de negócio necessário e o mapeamos para as classes que o descrevem. Nesta seção, trabalharemos o registro de palavras, então precisaremos de uma classe que mapeie as propriedades (e comportamentos, em alguns casos) para as palavras que utilizaremos no jogo.

Como este será nosso primeiro modelo a ser implementado, crie outra pasta chamada `models` na pasta `lib` do projeto e, dentro dela,

um arquivo chamado `palavra_model.dart` com o código apresentado na sequência. Observe que temos três propriedades e um construtor com parâmetros opcionais.

```
class PalavraModel {
  final String palavraID;
  final String palavra;
  final String ajuda;

  PalavraModel({this.palavraID, this.palavra, this.ajuda});
}
```

Quando trabalhamos com instâncias de classes, isto é, objetos, podem ocorrer situações nas quais haja a necessidade de realizarmos comparações entre eles. Entretanto, quando utilizamos o operador `==` na comparação, ela ocorre como verificador de instâncias (objetos) iguais e não de identidade (valores) dos objetos.

Em OO, a maioria das classes proporciona a implementação da identidade de objetos por meio da sobrescrita de dois métodos. Em Dart, são utilizados o operador `==` e a propriedade `hashCode`. Em outras linguagens, como o Java, temos o `equals()` e o `hashcode()` e no C# o `Equals()` e `GetHashCode()`.

Esse é um processo extremamente importante, mas muito trabalhoso, principalmente quando podem ocorrer mudanças na definição das propriedades que compõem a identidade do objeto. Por isso, podemos ter comodidade ao utilizar um componente que faz esse trabalho para nós sem muito esforço. Estou falando do `Equatable`, sobre o qual você pode obter maiores informações em https://pub.dev/packages/equatable.

No momento da escrita deste livro, o plugin estava na versão 1.1.1 e a instrução a seguir realiza seu registro em nosso arquivo `pubspec.yaml` nas dependências do projeto. Lembre-se de clicar em `Packages get` após o registro.

```
equatable: ^1.1.1
```

Com o plugin devidamente instalado em nosso projeto, precisamos alterar a implementação de nossa classe. Para facilitar, repetirei todo o código da classe na listagem a seguir. Observe a extensão de `Equatable` e a sobrescrita de `props`, que define nossa propriedade para a identidade do objeto.

```
import 'package:equatable/equatable.dart';

class PalavraModel extends Equatable {
  final String palavraID;
  final String palavra;
  final String ajuda;

  @override
  List<Object> get props => [palavraID];

  PalavraModel({this.palavraID, this.palavra, this.ajuda});
}
```

7.2 Implementações para uso de BLoC no formulário

Existem diversas técnicas para implementar um formulário no Flutter, mas eu gosto de uma apontada inclusive como exemplo de uso do plugin `flutter_bloc`. Ela possui uma certa complexidade inicial, mas nada que possa assustar. Vamos implementá-la aos poucos e com explicações que visam auxiliar na compreensão.

A classe do estado do formulário

Em nossa pasta `lib`, em `routes`, crie outra pasta chamada `palavras` e, dentro dela, uma chamada `bloc`. Dentro desta pasta, crie outra chamada `crud` e, nela, um arquivo chamado `palavras_crud_form_state.dart`. Esse arquivo conterá uma classe que mapeará o estado do formulário. Ela pode ser vista como uma

aplicação do padrão MVVM que faz uso de classes para o `Model`, `View` e `View-Model`. Em nosso caso, a classe que implementaremos representa a de `View-Model`. Pelo fato de a classe ser um pouco grande, vou apresentá-la por partes. A primeira está na sequência.

```
class PalavrasCrudFormState {
  final String palavra;
  final bool aPalavraEhValida;
  final String ajuda;
  final bool aAjudaEhValida;
  final bool formularioEnviadoComSucesso;

  bool get isFormValid => aPalavraEhValida && aAjudaEhValida;

  const PalavrasCrudFormState(
      {this.palavra,
      this.aPalavraEhValida,
      this.ajuda,
      this.aAjudaEhValida,
      this.formularioEnviadoComSucesso});
}
```

O código anterior pode parecer redundante com nosso modelo de negócio, pois traz novamente a definição de `palavra` e `ajuda`. Entretanto, é preciso que seja abstraído que essas propriedades estarão ligadas ao formulário e não ao nosso modelo de negócio. Essa classe será o modelo de negócio para a visão (o `View-Model`).

Temos também a definição de propriedades do tipo `bool`, que serão nossos flags para validar os estados das propriedades de negócio e de nosso formulário. O código termina com a implementação do nosso construtor com os parâmetros opcionais e define o construtor como constante.

Na sequência, após o código citado acima, implementaremos um construtor de fábrica, ou um `factory constructor`, que nos fornecerá um objeto inicializado de nossa classe, uma prática muito comum em Dart. Você pode optar por não ter esse método, o que levaria ao

uso do construtor padrão, que não tem seus parâmetros obrigatórios, então seriam nulos.

```
factory PalavrasCrudFormState.initial() {
    return PalavrasCrudFormState(
        palavra: '',
        aPalavraEhValida: false,
        ajuda: '',
        aAjudaEhValida: false,
        formularioEnviadoComSucesso: false);
}
```

Terminando nossa implementação para a classe, vamos codificar o que segue na listagem após o método anterior. O método `copyWith()` existe em muitas classes oferecidas pelo Dart e Flutter. Em essência, ele retorna uma nova instância da classe com alguns valores do objeto que está sendo copiado e alguns valores novos. Veja que, no construtor do `return`, temos o operador `??`. No caso de o primeiro operando ser nulo, o valor do segundo é utilizado. Logo veremos o uso desse método.

```
PalavrasCrudFormState copyWith(
  {String palavra,
  bool aPalavraEhValida,
  String ajuda,
  bool aAjudaEhValida,
  bool formularioEnviadoComSucesso}) {
    return PalavrasCrudFormState(
        palavra: palavra ?? this.palavra,
        aPalavraEhValida: aPalavraEhValida ?? this.aPalavraEhValida,
        ajuda: ajuda ?? this.ajuda,
        aAjudaEhValida: aAjudaEhValida ?? this.aAjudaEhValida,
        formularioEnviadoComSucesso:
            formularioEnviadoComSucesso ??
this.formularioEnviadoComSucesso);
}
```

As classes dos eventos de transição para o BLoC

Como comentado no capítulo anterior, o pattern BLoC trabalha com Streams (fluxos), onde temos um evento que representa o fluxo de entrada e um estado que representa o fluxo de saída. Naquele exemplo, utilizamos enumeradores, porque a situação era mais simples, mas agora teremos eventos que representam a transição de estados de controles em nosso formulário, mais precisamente os controles que representam a entrada dos valores para `palavra` e `ajuda`. Isso nos leva a implementar uma hierarquia de classes. Veja o código a seguir, que deve ser codificado em um arquivo chamado `palavras_crud_form_event.dart` na mesma pasta onde criamos o arquivo anterior. Observe que importamos `meta.dart` para não termos dependência de plataformas.

```
import 'package:meta/meta.dart';

abstract class PalavrasCrudFormEvent {
  const PalavrasCrudFormEvent();
}

class PalavraChanged extends PalavrasCrudFormEvent {
  final String palavra;

  const PalavraChanged({@required this.palavra});
}

class AjudaChanged extends PalavrasCrudFormEvent {
  final String ajuda;

  const AjudaChanged({@required this.ajuda});
}

class FormSuccessSubmitted extends PalavrasCrudFormEvent {}

class FormReset extends PalavrasCrudFormEvent {}
```

Notou que nossa primeira classe é `abstract`? Isso nos garante que essa classe não pode ser instanciada. Afinal, ela será a superclasse de nossa hierarquia. Depois, temos duas classes que auxiliarão na transição de valores para a palavra a ser registrada e sua ajuda. Ao

final, temos duas classes que representarão ações do usuário na submissão e *reset* de nossos controles do formulário.

A classe que implementa o BLoC

Chegamos ao terceiro arquivo de nossa arquitetura, o BLoC em si. É nessa classe que implementaremos toda a regra de negócio ligada ao nosso formulário de registro de palavras, pois não queremos deixar essa responsabilidade em nosso widget da rota, que deve conter apenas o desenho da interface com o usuário.

Vamos criar o arquivo `palavras_crud_form_bloc.dart` na mesma pasta utilizada nas duas últimas implementações. Com o arquivo criado, implemente nele o código da listagem a seguir. Vamos focar na explicação no método `mapEventToState()`. Veja nele que verificamos o tipo do evento gerador da invocação e retornamos o estado por meio de `state`, que o plugin inteligentemente identifica, com valores novos para propriedades específicas, de acordo com o evento em questão. Ao final do código, temos duas funções implementadas que validarão os valores de `word` e `help`, que, em nosso caso, têm apenas a necessidade de possuir valor.

```
import 'package:flutter_bloc/flutter_bloc.dart';
import 'palavras_crud_form_event.dart';
import 'palavras_crud_form_state.dart';

class PalavrasCrudFormBloc
    extends Bloc<PalavrasCrudFormEvent, PalavrasCrudFormState> {
  @override
  PalavrasCrudFormState get initialState =>
PalavrasCrudFormState.initial();

  @override
  Stream<PalavrasCrudFormState> mapEventToState(
    PalavrasCrudFormEvent event,
  ) async* {
    if (event is PalavraChanged) {
      yield state.copyWith(
        palavra: event.palavra,
```

```
        aPalavraEhValida: _aPalavraEhValida(event.palavra),
      );
    } else if (event is AjudaChanged) {
      yield state.copyWith(
        ajuda: event.ajuda,
        aAjudaEhValida: _aAjudaEhValida(event.ajuda),
      );
    } else if (event is FormSuccessSubmitted) {
      yield state.copyWith(formularioEnviadoComSucesso: true);
    } else if (event is FormReset) {
      yield PalavrasCrudFormState.initial();
    }
  }

  bool _aPalavraEhValida(String palavra) {
    return palavra.isNotEmpty;
  }

  bool _aAjudaEhValida(String ajuda) {
    return ajuda.isNotEmpty;
  }
}
```

Organizando os pacotes para importação

Criamos três arquivos e toda vez que formos utilizar os recursos deles teremos que importar os três. Isso é um pouco verboso e trabalhoso. Dart traz um recurso de exportação de pacotes, que nos permite criar um arquivo e nele ter a exportação de diversos outros pacotes, sendo possível importarmos um único arquivo no uso dos recursos implementados em três. Na mesma pasta dos arquivos anteriores, crie o arquivo `palavras_crud_bloc.dart` e nele insira o código da sequência.

```
export 'palavras_crud_form_bloc.dart';
export 'palavras_crud_form_event.dart';
export 'palavras_crud_form_state.dart';
```

7.3 Mixin no Dart

Quando trabalhamos herança de classes em qualquer linguagem orientada a objetos, nós nos deparamos com herança por extensão e por implementação. Na extensão, a maioria das linguagens permite apenas que seja de uma classe (normalmente de uma classe abstrata), não havendo limites para a implementação (por interfaces). Mas e se tivermos necessidade de que uma classe tenha competências já implementadas em outras classes, que não sejam a que ela estende?

Usaremos aqui um exemplo que poderíamos abstrair e criar um widget em vez de um Mixin, mas o objetivo será apenas didático. Informações maiores sobre Mixin podem ser obtidas em https://medium.com/flutter-community/dart-what-are-mixins-3a72344011f3.

Na pasta `lib`, crie outra chamada `mixins` e, dentro dela, um arquivo `widgets_mixin.dart`, com o código apresentado na sequência. Nosso objetivo é sempre minimizar código na classe de nossos widgets que representam uma visão com o usuário. Com a implementação do código a seguir, teremos um método que retornará um `TextFormField`, que será configurado de acordo com parâmetros enviados para o método.

```
import 'package:flutter/material.dart';

mixin TextFormFieldMixin {
  textFormField({
    maxLines,
    focusNode,
    controller,
    labelText,
    textInputAction,
    onFieldSubmitted,
    validator,
  }) {
    return TextFormField(
```

```
      autovalidate: true,
      maxLines: maxLines ?? 1,
      focusNode: focusNode,
      controller: controller,
      decoration: InputDecoration(
        labelText: labelText ?? 'Informe o labelText',
      ),
      textInputAction: textInputAction,
      onFieldSubmitted: onFieldSubmitted,
      validator: validator,
    );
  }
}
```

7.4 A rota para o formulário de registro de palavras

Agora que temos nossa arquitetura toda implementada, precisamos começar a desenhar nossa rota com o formulário para registro de palavras. Na pasta `\lib\routes\palavras`, crie um arquivo chamado `palavras_crud_route.dart` com a implementação inicial apresentada a seguir. Observe o `with` na declaração da classe. Poderíamos ter vários separados por vírgula. Não se incomode com um erro na declaração da classe `State`.

```
import 'package:flutter/material.dart';
import 'package:cc04/mixins/widgets_mixin.dart';

class PalavrasCRUDRoute extends StatefulWidget {
  @override
  _PalavrasCRUDRouteState createState() => _PalavrasCRUDRouteState();
}

class _PalavrasCRUDRouteState extends State<PalavrasCRUDRoute>
    with TextFormFieldMixin {
}
```

Vamos à declaração de variáveis que precisaremos utilizar em nosso código. Logo após a declaração da classe `State`, implemente o código a seguir, onde temos dois pares de controles que serão utilizados em nossos `TextFormField`. Para cada propriedade do nosso modelo de negócio, teremos um `TextEditingController` para ouvirmos e controlarmos as alterações nos controles quando o usuário for informar os valores para esses controles e um `FocusNode`, responsável por gerenciar o foco e navegação entre os `TextFormField`. Finalizamos a declaração das variáveis com uma para nosso BLoC e outra para nosso contexto de renderização. Fique atento aos imports. Para o `bloc`, importe o último arquivo que criamos, que tem referência para todos os demais.

```
final _palavraController = TextEditingController();
final _ajudaController = TextEditingController();
final FocusNode _palavraFocus = FocusNode();
final FocusNode _ajudaFocus = FocusNode();

PalavrasCrudFormBloc _palavrasCrudFormBloc;
BuildContext _buildContext;
```

Com a declaração das variáveis concluídas, precisamos inicializá-las e finalizá-las, liberando os recursos consumidos. Para isso, sobrescreveremos dois métodos, o `initState()` e o `dispose()`, como pode ser visto na sequência. Cuide da importação para `BlocProvider` e não se preocupe com os erros nos `addListeners`.

```
@override
void initState() {
    super.initState();
    this._palavrasCrudFormBloc = BlocProvider.of<PalavrasCrudFormBloc>
(context);
    this._palavraController.addListener(_onPalavraChanged);
    this._ajudaController.addListener(_onAjudaChanged);
}

@override
void dispose() {
    this._palavraController.dispose();
```

```
    this._ajudaController.dispose();
    super.dispose();
}
```

Observe que, para os `TextEditingController`, adicionamos um método como *listener* deles, o que nos possibilitará controlar as interações do usuário. Veja estes métodos na sequência. Observe que registramos uma transição em nosso `BLoC` em cada um dos métodos, enviando sempre o respectivo valor que deverá ser informado como alterado. Se você voltar ao código de nosso BLoC, verá que esses eventos atualizam os valores recebidos e alteram os flags de validação de nossos dados. Atente ao import e veja que o erro anterior desaparece.

```
void _onPalavraChanged() {
    _palavrasCrudFormBloc.add(PalavraChanged(palavra:
this._palavraController.text));
}

void _onAjudaChanged() {
    _palavrasCrudFormBloc.add(AjudaChanged(ajuda:
this._ajudaController.text));
}
```

Para já podermos ter alguma coisa renderizada por nossa rota, vamos começar a configurar nosso método `build()` com o código apresentado na sequência. Após a implementação, o erro na definição da classe de `State` desaparece.

```
@override
Widget build(BuildContext context) {
    this._buildContext = context;
    return Scaffold(
      backgroundColor: Colors.grey[400],
      appBar: AppBar(
        backgroundColor: Colors.grey[600],
        title: Text(
          'Registro de Palavras',
        ),
      ),
```

```
      body: SafeArea(
        child: Center(
          child: SingleChildScrollView(
            child: Container(),
          ),
        ),
      ),
    );
}
```

Podemos ver que, logo no início do método, atribuímos nosso contexto à nossa propriedade e depois retomamos um `Scaffold` com uma cor específica de background, um `AppBar` e um `body`. Neste último, temos dois widgets novos, o `SafeArea` e o `SingleChildScrollView`. O primeiro nos garantirá que não teremos nada renderizado nas áreas inacessíveis dos dispositivos, como o `Notch`, introduzido pelo iPhone X, e hoje comum em dispositivos de muitas marcas. Já o `SingleChildScrollView` faz com que nossa visão seja rolável. Isso é importante em rotas que solicitarão dados ao usuário pelo teclado virtual. Caso não utilizemos esse widget, teremos erro na renderização dele. Recomendo que, caso não conheça esse problema, tire esse widget do exemplo para testar.

7.5 O controle de rotas

Com as implementações que temos, já podemos testar a execução e navegação de nossa aplicação para a rota de registro de palavras, mas usaremos os recursos do Flutter para isso. Criaremos inicialmente dois arquivos para implementar essa solução.

Na pasta `lib\app_constants`, crie um arquivo chamado `router_constants.dart` com o código mostrado na sequência. Veja que apenas definimos uma constante literal.

```
const String kPalavrasCRUDRoute = '/palavrasCRUD';
```

Para o segundo arquivo, ainda na pasta `lib`, crie outra pasta, agora chamada `apphelpers` e, nela, o arquivo `app_router.dart` com o código a seguir. Observe que na classe temos apenas um método estático que pode ser invocado sem termos que instanciar a classe. Esse método receberá algumas configurações relacionadas à rota que será aplicada e uma delas é a `name`. Com este valor atribuído a `settings`, fazemos um processo de seleção de qual rota deve ser seguida. A princípio teremos apenas a que acabamos de criar.

```
import 'package:cc04/app_constants/router_constants.dart';
import 'package:cc04/routes/palavras/palavras_crud_route.dart';
import 'package:flutter/material.dart';

class AppRouter {
  static Route<dynamic> generateRoute(RouteSettings settings) {
    switch (settings.name) {
      case kPalavrasCRUDRoute:
        return MaterialPageRoute(builder: (_) => PalavrasCRUDRoute());
      default:
        return MaterialPageRoute(
            builder: (_) => Scaffold(
                  body: Center(
                      child: Text('No route defined for
${settings.name}')),
                ));
    }
  }
}
```

Notou que não estamos utilizando o `Navigator`, mas sim o `MaterialPageRoute`? Com esse widget temos total controle do processo de transição entre rotas, mas no momento queremos apenas navegar para uma.

Para utilizarmos esse nosso controlador de rotas, precisamos registrá-lo em nosso `MaterialApp`, que está no `main.dart`, e faremos isso adicionando a seguinte configuração logo no início do construtor da classe. Para facilitar, trouxe parte do código. Veja-o na sequência e lembre-se do import.

```
class ForcaApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      onGenerateRoute: AppRouter.generateRoute,
      debugShowCheckedModeBanner: false,
// Código omitido
}
```

Já temos tudo implementado e configurado. Precisamos agora realmente codificar o momento da navegação do Drawer para a rota de registro de palavras. Isso deve ser feito na classe `DrawerBodyContentApp`, que está no arquivo `drawerbodycontent_app.dart`. Precisamos enviar para nosso `ListTileAppWidget` o que fazer quando houver a interação com o usuário. Veja o código na sequência.

```
ListTileAppWidget(
    titleText: 'Novas Palavras',
    subtitleText: 'Vamos inserir palavras?',
    onTap: () {
      Navigator.of(context).pop();
      Navigator.of(context).pushNamed(kPalavrasCRUDRoute);
    },
),
```

Certamente, você terá um erro ao implementar o código anterior, pois nosso `ListTileAppWidget` não tem ainda a propriedade `onTap` configurada. A propósito, viu que no envio para ela estamos realizando um `pop()`? Isso faz com que o Drawer seja fechado antes de realizarmos a navegação por meio de `pushNamed()` para nossa rota. Veja no código a seguir a implementação dessa propriedade e a declaração dela no construtor. Isso é no arquivo `listtile_app_widget.dart`.

```
// antes do construtor
final Function onTap;

// no construtor, como último argumento
@required this.onTap
```

Quando criamos nosso primeiro Drawer, no capítulo 6, no qual criamos o widget `ListTileAppWidget`, colocamos a captura de interação apenas no ícone referente ao `trailing`. Lá comentei que isso talvez não fosse a melhor opção e deixei a seu cargo capturar o gesto em todo o widget. Vou trazer aqui a implementação para você comparar com a sua, veja-a na sequência. Trago apenas o início do método `build()`. Lembre-se de remover o `GestureDetector` do `Icon`, ok?

```
@override
Widget build(BuildContext context) {
    return Padding(
      padding: contentPadding,
      child: GestureDetector(
        onTap: this.onTap,
        child: Row(
// código omitido
```

Falta pouco, precisamos registrar nosso `BlocProvider` lá no `main()`. Lembra que comentamos isso quando implementamos o BLoC para o Drawer? Para facilitar, trago todo o código, mas a mudança é só para o `BlocProvider<PalavrasCrudFormBloc>`.

```
void main() => runApp(
      MultiBlocProvider(
        providers: [
          BlocProvider<DrawerOpenStateBloc>(
            create: (BuildContext context) => DrawerOpenStateBloc(),
          ),
          BlocProvider<PalavrasCrudFormBloc>(
            create: (BuildContext context) => PalavrasCrudFormBloc(),
          ),
        ],
        child: ForcaApp(),
      ),
    );
```

Agora sim você pode testar sua aplicação. Veja que tivemos bastante conteúdo a aprender e implementação a fazer para podermos executar a aplicação. Mas, ao selecionar a opção de

registrar palavras, a navegação levará para uma rota só com o `AppBar` e um `Container` vazio, ok? Lembre-se disso.

7.6 O formulário para registro das palavras

Nosso formulário terá, no total, três controles com os quais o usuário poderá interagir: 2 `TextFormFields`, sendo um para a palavra e outro para a ajuda, e 1 botão para o usuário confirmar o registro. Implementamos a regra para os `TextFormFields`, que não podem estar vazios, e o botão só poderá estar habilitado se os dois `TextFormFields` estiverem validados, o que validará também o formulário como preenchido. Tudo isso é regra de negócio de nossa visão que implementamos no BLoC.

Entretanto, antes de termos esses controles, vamos criar um novo widget onde esses componentes serão inseridos. Será um controle legal, como um card, com um sombreamento que dará a ele a sensação de estar iluminado. Este componente será chamado de `ContainerIluminadoWidget`, que deverá estar em um arquivo chamado `container_iluminado_widget.dart` na pasta `widgets`, com o código apresentado na sequência.

```
import 'package:flutter/material.dart';

class ContainerIluminadoWidget extends StatelessWidget {
  final double height;
  final Color backgroundColor;
  final Color shadowColor;
  final Widget child;

  ContainerIluminadoWidget(
      {this.height = 125,
      this.backgroundColor = Colors.white,
      this.shadowColor = Colors.grey,
      this.child});
```

```
  @override
  Widget build(BuildContext context) {
    return Container(
      margin: EdgeInsets.only(bottom: 16.0),
      child: Container(
          height: this.height,
          decoration: BoxDecoration(
              color: this.backgroundColor,
              borderRadius: BorderRadius.all(Radius.circular(15)),
              boxShadow: [
                BoxShadow(
                  blurRadius: 5,
                  color: this.shadowColor,
                  spreadRadius: 5,
                  offset: Offset(0.1, 0.1),
                )
              ]),
          child: child),
    );
  }
}
```

Verificou que o código é simples? Um `Container` com um `BoxDecoration`, que terá uma sombra fornecida pelo `BoxShadow` e terá um certo arredondamento nas bordas propiciado por `BorderRadius`. Nosso construtor recebe quatro argumentos, que possibilitam uma configuração mínima para nosso contêiner. É possível flexibilizar mais com a adição de novos argumentos para valores constantes no código, caso você prefira.

Na sequência, vamos criar nosso método `_form()`, responsável pelos componentes de interação que comentamos há pouco e faremos isso na classe `_PalavrasCRUDRouteState`. Este código é simples, mas com pontos interessantes. Estamos usando o `textFormField()`, que implementamos em nosso `Mixin`. Temos o `SizedBox` separando os controles de interação e temos um `RaisedButton`, que o usuário utilizará para confirmar os dados. Teremos um erro, mas não se preocupe. Atente-se ao import para `PalavrasCrudFormState`.

```
Widget _form(PalavrasCrudFormState formState) {
    return Form(
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.stretch,
        children: <Widget>[
          textFormField(
              focusNode: this._palavraFocus,
              controller: this._palavraController,
              labelText: 'Palavra',
              onFieldSubmitted: (_) => FocusScope.of(this._buildContext)
                  .requestFocus(this._ajudaFocus),
              textInputAction: TextInputAction.next,
              validator: (_) {
                return formState.aPalavraEhValida ? null : 'Informe a
palavra';
              }),
          SizedBox(
            height: 20,
          ),
          textFormField(
              maxLines: 5,
              focusNode: this._ajudaFocus,
              controller: this._ajudaController,
              labelText: 'Ajuda',
              validator: (_) {
                return formState.aAjudaEhValida ? null : 'Informe a
ajuda';
              }),
          SizedBox(
            height: 20,
          ),
          RaisedButton(
            onPressed: formState.isFormValid ? _onSubmitPressed : null,
            child: Text('Gravar'),
          ),
        ],
      ),
    );
}
```

Verifique o envio para `validator` nos `textFormField()`. Temos uma verificação em nosso estado, encaminhado pelo BLoC para cada controle. Caso não seja válido, uma mensagem de erro é exibida abaixo do `TextFormField`. Temos também um flag em nosso `RaisedButton`, que só terá a atribuição de um método, o `_onSubmitPressed()`, caso o formulário esteja validado como corretamente preenchido. Caso isso não ocorra, `null` é atribuído ao `onPressed`, o que deixará o botão desabilitado.

Ainda não temos o método `_onSubmitPressed()` implementado, mas poderemos fazer isso com o código a seguir. Veja como ele é simples. Ele apenas registra um evento em nosso BLoC e a regra para este evento está na implementação do BLoC.

```
void _onSubmitPressed() {
    _palavrasCrudFormBloc.add(FormSuccessSubmitted());
}
```

Com o objetivo de retirar do método `build()` a criação de todos os controles, criaremos um método em nossa classe `_PalavrasCRUDRouteState` que será responsável por configurar nossa coluna para o formulário e este código pode ser visto na sequência. Observe que o retorno é um `Column`, que tem dois filhos, um `Container` e `SizedBox`. No primeiro, temos um `Padding`, que tem como filho o `ContainerIluminadoWidget`, que criamos há pouco. Como filho deste contêiner, temos um `Padding` que armazenará nosso formulário, que pertence ao `BlocBuilder<PalavrasCrudFormBloc, PalavrasCrudFormState>`. Veja que invocamos o método chamado `_form()`, também criado há pouco, enviando a ele o estado retomado pelo BLoC. Lembre-se do import.

```
Widget _mainColumn() {
    return Column(
      children: <Widget>[
        Container(
          child: Padding(
            padding: const EdgeInsets.symmetric(horizontal: 10.0,
vertical: 10),
```

```
            child: ContainerIluminadoWidget(
              backgroundColor: Colors.white,
              shadowColor: Colors.white70,
              height: 350,
              child: Padding(
                padding: const EdgeInsets.only(left: 10, top: 10, right:
10),
                child: BlocBuilder<PalavrasCrudFormBloc,
PalavrasCrudFormState>(
                    builder: (context, formState) {
                  return _form(formState);
                }),
              ),
            ),
          ),
        ),
        SizedBox(
          height: 10,
        ),
      ],
    );
}
```

Para podermos testar nossa implementação realizada até aqui, basta, em nosso método `build()`, substituir o `Container`, que está como `child` de `SingleChildScrollView`, pelo nosso método `_mainColumn()`. Você já verá o formulário exibido e poderá testar todo o comportamento comentado durante as explicações. A figura a seguir apresenta como deverá estar seu formulário com e sem o preenchimento dos controles, liberando o botão para interação.

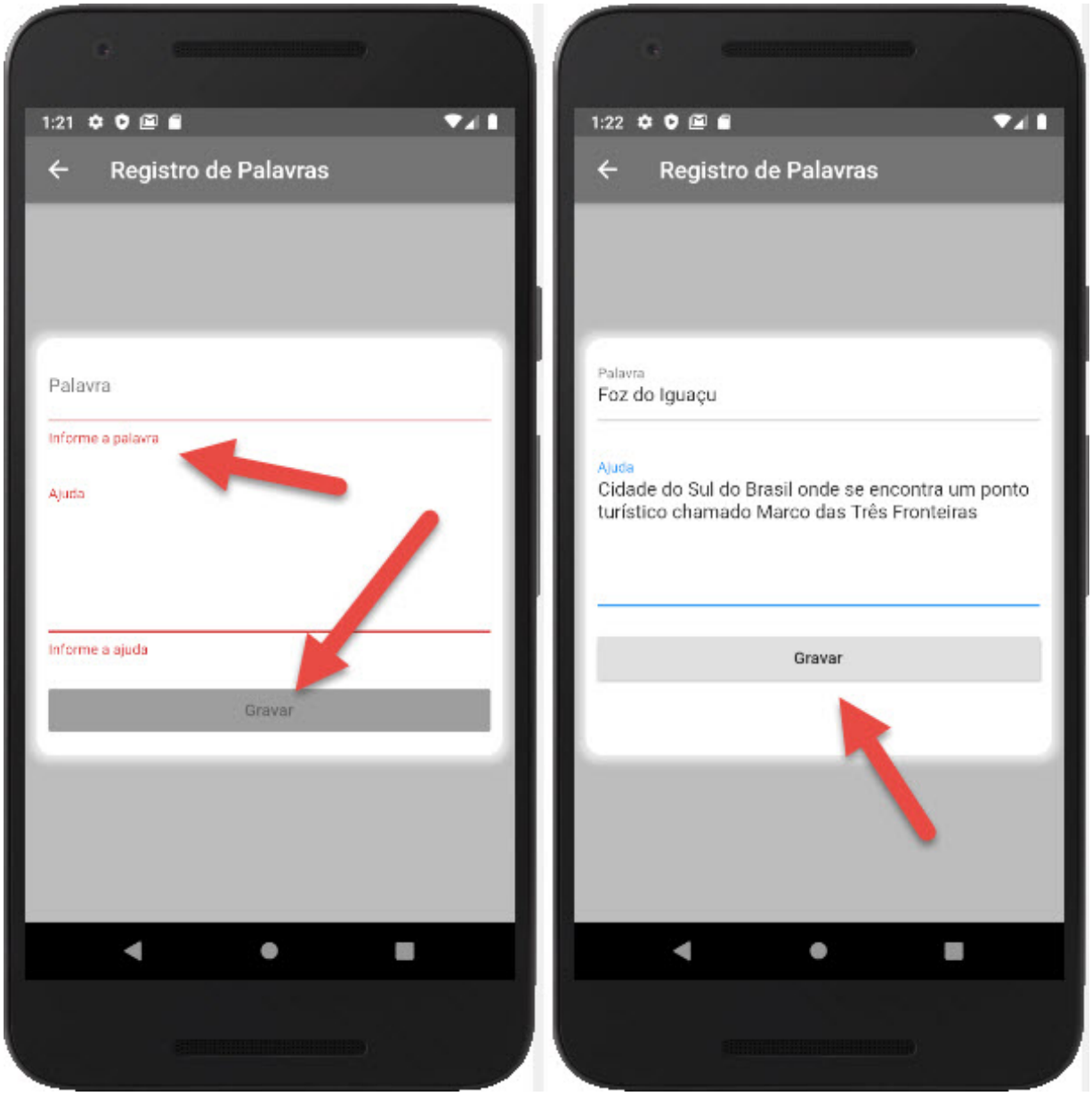

Figura 7.1: Formulário para registro de palavras

7.7 Confirmação dos dados informados

Já temos o `RaisedButton` habilitado quando os dados para o formulário estiverem informados. Precisamos agora trabalhar a informação ao usuário sobre esse processo e veremos nesta seção

algumas das possibilidades. Uma das técnicas será verificar o estado retomado para o formulário no método `_mainColumn()`, em nosso `BlocBuilder`, e com base nele definir o que fazer. Veja que, no momento, estamos apenas retomando `_form()` sem nenhuma verificação.

Um widget genérico para a confirmação

Vamos criar um widget genérico para a confirmação de validade dos dados do formulário. Algo simples, mas que possa ser reutilizável em qualquer parte do projeto ou por outros projetos. Na pasta `widgets`, crie outra chamada `dialogs` e, dentro dela, um arquivo chamado `success_dialog_widget.dart` com o código apresentado na sequência. Veja que temos um `Padding` como retorno com um `Column`, que tem dois filhos, sendo um deles um `Row`, que exibe um `Icon` e um `Text`, e um `RaisedButton`.

```
import 'package:flutter/material.dart';

class SuccessDialogWidget extends StatelessWidget {
  final VoidCallback onDismissed;

  SuccessDialogWidget({@required this.onDismissed});

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: EdgeInsets.all(20),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Row(
            mainAxisSize: MainAxisSize.max,
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Icon(Icons.info),
              Flexible(
                child: Padding(
                  padding: EdgeInsets.all(10),
```

```
                child: Text(
                  'Dados informados com sucesso!',
                  softWrap: true,
                ),
              ),
            ),
          ],
        ),
        RaisedButton(
          child: Text('OK'),
          onPressed: onDismissed,
        ),
      ],
    ),
  );
 }
}
```

Da maneira que implementamos o widget anterior, ele está engessado no texto e no ícone informado, mas você poderia colocar esses widgets com seus valores sendo recebidos pelo construtor, tal qual estamos fazendo para `onDismissed`, um `VoidCallback`, que é o padrão para uma função que não retoma valores. Poderíamos ter utilizado o `Function` aqui também.

Agora, para consumirmos esse widget, vamos adaptar nossa implementação no `_mainColumn()`, como pode ser visto no código a seguir, que traz apenas a parte envolvida para essa implementação. Note que estamos falando do arquivo `palavras_crud_route.dart` e que vamos substituir o `child:` de nosso `Padding`, que já tem um `BlocBuilder`. Atente-se ao import.

```
child: BlocBuilder<PalavrasCrudFormBloc, PalavrasCrudFormState>(
    builder: (context, formState) {
      if (formState.formularioEnviadoComSucesso)
        return SuccessDialogWidget(
          onDismissed: () {
            _palavraController.clear();
            _ajudaController.clear();
            this._palavrasCrudFormBloc.add(FormReset());
```

```
          },
        );
      return _form(formState);
}),
```

Observe que o método anônimo que passamos para `onDismissed` está limpando os `TextFormFields` por meio de seus controladores e está gerando um novo evento em nosso BLoC. Podemos ver na figura a seguir nossa aplicação com a confirmação dos dados exibida.

Figura 7.2: Confirmação dos dados digitados

Um widget com características da plataforma do dispositivo para a confirmação

Vamos para uma segunda técnica para exibição da mensagem de confirmação dos dados para o formulário. Utilizaremos agora um componente `modal`, que ficará sobre o formulário.

Cada plataforma tem suas características visuais. Um diálogo de alerta no Android é visualmente diferente de um no iOS com a mesma funcionalidade, embora possamos criar um widget totalmente customizado para ser idêntico em ambas as plataformas.

Criaremos aqui um widget um pouco mais complexo que invocará dois outros widgets, de acordo com plataforma de execução, e esses dois widgets ainda serão compostos por outro. Mas veremos tudo com calma e explicação.

Em nossa pasta recém-criada chamada `dialogs`, vamos criar um arquivo chamado `information_cupertino_alert_dialog_widget.dart`. Nele, insira o código da sequência. Na leitura do código, identifique as propriedades que serão populadas pelo construtor. Temos um título, uma mensagem e um `List<Widget>`, que conterá os botões com possíveis opções para o usuário escolher.

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';

class InformationCupertinoAlertDialogWidget extends StatelessWidget {
  final String title;
  final String message;
  final List<Widget> actions;

  const InformationCupertinoAlertDialogWidget({
    @required this.title,
    @required this.message,
    @required this.actions,
  });

  @override
```

```
  Widget build(BuildContext context) {
    return CupertinoAlertDialog(
      title: Column(
        children: <Widget>[
          Text(this.title),
          SizedBox(height: 20,),
        ],
      ),
      content: Text(this.message),
      actions: actions,
    );
  }
}
```

Veja que, no método `build()`, estamos utilizando o `CupertinoAlertDialog`, que é característico de aplicações iOS. Como título para este *dialog*, temos um `Column` que faz uso do argumento recebido pelo construtor para desenhar o título (`title`), seguido de um `SizedBox` para que possa haver uma separação entre o título e a mensagem. Na sequência, em `content`, temos a mensagem recebida. Por fim, temos as `actions` para o dialog, que em nosso caso serão os botões com possibilidades de interação para o usuário.

Vamos agora criar uma classe semelhante à anterior, mas para o Android, usando `Material`. Na pasta `dialogs`, crie um arquivo chamado `information_material_alert_dialog_widget.dart` e, nele, o código apresentado na sequência. Não vamos comentá-lo muito, ele só muda o retorno do `build`, que é `AlertDialog`, de `Material` do Android. Poderíamos melhorar este código criando um widget para cada componente dos dialogs (`title`, `content` e `actions`), pois é tudo igual, ou ainda melhorá-lo com uma hierarquia de classes, mas isso ficará para você abstrair, ok?

```
import 'package:flutter/material.dart';

class InformationMaterialAlertDialogWidget extends StatelessWidget {
  final String title;
  final String message;
```

```
  final List<Widget> actions;

  const InformationMaterialAlertDialogWidget({
    @required this.title,
    @required this.message,
    @required this.actions,
  });

  @override
  Widget build(BuildContext context) {
    return AlertDialog(
      title: Column(
        children: <Widget>[
          Text(this.title),
          SizedBox(
            height: 20,
          ),
        ],
      ),
      content: Text(
        this.message,
      ),
      actions: actions,
    );
  }
}
```

Nas duas classes anteriores, temos `actions` recebendo `actions` e criaremos um novo widget para cada action que será exibida no dialog. Ainda na pasta `dialogs`, crie um novo arquivo chamado `actions_flatbutton_to_alertdialog_widget.dart` e, nele, implemente o código a seguir. Observe as propriedades definidas e o construtor para a classe no código. Temos aqui a previsão para aplicações iOS nos parâmetros `isDefaultAction` e `isDestructiveAction`, que semanticamente são compreendidas. Elas não são relevantes para o Android.

Atente-se ao `import` para `dart:io`, pois estamos usando `Platform` em nosso `return`. Verificou que retomamos o widget de acordo com

a plataforma em execução do aplicativo? Se for o Android em execução, retornamos um `FlatButton` e, se for iOS, um `CupertinoDialogAction`, ambos configurados com o que exibirão e o que realizarão se forem pressionados.

```
import 'dart:io';

import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';

class ActionsFlatButtonToAlertDialogWidget extends StatelessWidget {
  final String messageButton;
  final bool isDefaultAction;
  final bool isDestructiveAction;

  const ActionsFlatButtonToAlertDialogWidget({
    @required this.messageButton,
    this.isDefaultAction = false,
    this.isDestructiveAction = false,
  });

  @override
  Widget build(BuildContext context) {
    return (Platform.isAndroid)
        ? FlatButton(
            onPressed: () =>
Navigator.of(context).pop(this.messageButton),
            child: Text(
              this.messageButton,
            ),
          )
        : CupertinoDialogAction(
            isDefaultAction: this.isDefaultAction,
            isDestructiveAction: this.isDestructiveAction,
            child: Text(this.messageButton),
            onPressed: () =>
Navigator.of(context).pop(this.messageButton),
          );
  }
}
```

Em nosso `build`, no código anterior, o `onPressed` faz uso do `pop()`, retomando o texto do botão pressionado. O `child` está fácil de entender, apenas exibe o texto recebido. No caso do iOS, ainda encaminhamos os dois argumentos específicos.

Com o que temos implementado, já poderíamos verificar a plataforma de execução de nossa aplicação e escolher por instanciar uma ou outra classe para exibir um dialog, mas queremos que isso seja transparente para o consumidor de nossos widgets. Vamos criar um widget específico que deverá ser invocado e este widget decide qual dos dois que implementamos deve ser renderizado. Crie um arquivo chamado `information_alert_dialog_widget.dart` em nossa pasta `dialogs` e codifique-o de acordo com o código a seguir.

```
import 'dart:io';
import 'package:flutter/material.dart';
import 'information_cupertino_alert_dialog_widget.dart';
import 'information_material_alert_dialog_widget.dart';

class InformationAlertDialogWidget extends StatelessWidget {
  final String title;
  final String message;
  final List<Widget> actions;

  const InformationAlertDialogWidget({
    @required this.title,
    @required this.message,
    @required this.actions,
  });
  @override
  Widget build(BuildContext context) {
    return (Platform.isAndroid)
        ? InformationMaterialAlertDialogWidget(
            title: title,
            message: message,
            actions: actions,
          )
        : InformationCupertinoAlertDialogWidget(
```

```
            title: title,
            message: message,
            actions: actions,
          );
  }
}
```

Atente-se ao `import` para `dart:io` novamente, pois estamos usando aqui também o `Platform` em nosso `return`. Verificou que retomamos o widget de acordo com a plataforma em execução do aplicativo? E que o construtor encaminha para os widgets específicos apenas os valores recebidos?

Vamos agora enfim consumir nosso widget e, no código a seguir, trago a nova implementação para o `RaisedButton` de nosso formulário. Observe que agora alteramos a invocação ao `_onSubmitPressed`. Em vez de termos o `arrow function`, criamos um corpo maior para a invocação, pois exibiremos nosso *dialog*, e também separamos a responsabilidade de *reset* do formulário. Isso está no `palavras_crud_crud.dart` no método `_form()`.

```
RaisedButton(
    onPressed: formState.isFormValid
        ? () async {
            _onSubmitPressed();
            await _successDialog();
            _resetForm();
          }
        : null,
    child: Text('Gravar'),
),
```

Nós ainda não temos dois métodos apresentados na listagem anterior, mas vamos trazê-los no código a seguir, que começa com a implementação para `_successDialog()`, no mesmo arquivo. Observe a lista atribuída a `buttons`. Poderíamos ter aí quantos *action buttons* fossem necessários. Apenas para ilustrar, estamos enviando `true` para o `isDefaultAction`. Isso seria mais necessário se tivéssemos dois ou mais *action buttons* no *dialog*. Atente-se aos imports.

```
_successDialog() async {
    await showDialog(
      barrierDismissible: false,
      context: context,
      child: InformationAlertDialogWidget(
        title: 'Tudo certo',
        message: 'Os dados informados foram registrados com sucesso.',
        actions: [
          ActionsFlatButtonToAlertDialogWidget(
            messageButton: 'OK',
            isDefaultAction: true,
          ),
        ],
      ),
    );
}
```

Na primeira técnica que aplicamos para a confirmação dos dados, enviamos ao método todo o corpo da função que resetava o formulário após o sucesso no registro. Agora, refatoramos para uma função específica, a `_resetForm()`, que tem seu código apresentado na sequência.

```
_resetForm() {
    _palavraController.clear();
    _ajudaController.clear();
    this._palavrasCrudFormBloc.add(FormReset());
}
```

Agora sim, já podemos testar nossa aplicação. Execute-a, informe os dados e pressione o botão de gravar. A figura a seguir traz o resultado da execução em um Android e em um iOS. Note a diferença no visual. Como dito, poderíamos customizar um *dialog* comum às duas plataformas, mas aí poderíamos privar o usuário de sua experiência em seu dispositivo, então é uma situação que cabe a você e sua equipe definir.

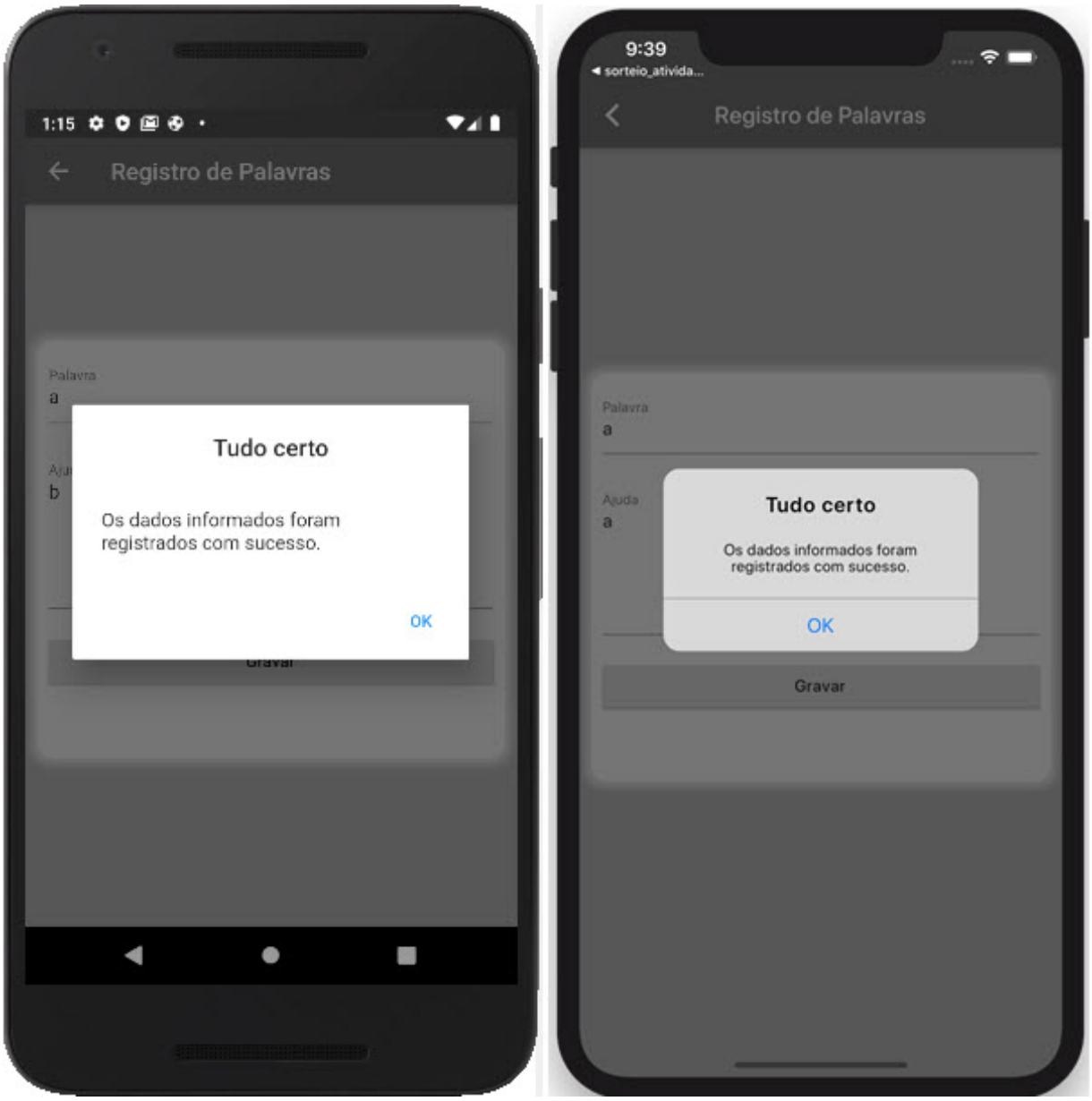

Figura 7.3: Dialog de confirmação dos dados digitados

Um widget de mensagem que se oculta automaticamente

No início desta seção, falamos em três técnicas para avisar o usuário do sucesso da operação de confirmação dos dados. Já vimos duas, vamos então à terceira que fará uso de um recurso chamado `SnackBar`, responsável por exibir uma mensagem na base

da tela do dispositivo, que fica exibida por um determinado período de tempo e depois desaparece.

Existem algumas maneiras para se exibir um `SnackBar`, mas a mais recomendada é aquela em que encapsulamos em um widget sua invocação. Em nosso caso, ele será invocado quando nosso `RaisedButton` for pressionado, então criaremos um widget que terá o `RaisedButton` e a invocação do `SnackBar` nele. Esse é um processo que isola os contextos da árvore de widgets e evita problemas durante a execução.

Em nossa pasta `\lib\widgets`, crie um arquivo chamado `raisedbutton_with_snackbar_widget.dart` e implemente o código a seguir nele. Temos esse widget customizado para identificar quando o botão estará habilitado (`onPressedVisible`), o texto a ser exibido no `RaisedButton` (`buttonText`), o texto a ser exibido no `SnackBar` (`textToSnackBar`) e as funções a serem executadas quando o botão for pressionado e após o `SnackBar` ser fechado (`onButtonPressed` e `onStackBarClosed`, respectivamente).

```
import 'package:flutter/material.dart';

class RaisedButtonWithSnackbarWidget extends StatelessWidget {
  final bool onPressedVisible;
  final String buttonText;
  final String textToSnackBar;
  final Function onButtonPressed;
  final Function onStackBarClosed;

  RaisedButtonWithSnackbarWidget({
    @required this.onPressedVisible,
    @required this.buttonText,
    @required this.textToSnackBar,
    @required this.onButtonPressed,
    @required this.onStackBarClosed,
  });

  @override
  Widget build(BuildContext context) {
```

```
    return RaisedButton(
      child: Text(this.buttonText),
      onPressed: this.onPressedVisible
          ? () async {
              FocusScope.of(context).unfocus();
              this.onButtonPressed();

              Scaffold.of(context)
                  .showSnackBar(
                    SnackBar(
                      backgroundColor: Colors.indigo,
                      content: Text(
                        this.textToSnackBar,
                        textAlign: TextAlign.center,
                        style: TextStyle(
                          fontWeight: FontWeight.bold,
                        ),
                      ),
                      duration: Duration(seconds: 3),
                    ),
                  )
                  .closed
                  .then((_) => this.onStackBarClosed());
            }
          : null,
    );
  }
}
```

Observe o uso de `FocusScope.of(context).unfocus();`, o que garante que, caso haja algum `TextFormField` com o foco, ele o perca, deixando a rota totalmente visível ao usuário. Depois, nossa função responsável pelo `onPressed` do `RaisedButton` é executada e, em seguida, invocamos nosso `showSnackBar`, encaminhando a ele um `SnackBar` com algumas customizações facilmente identificadas. Temos a propriedade `duration` configurada para `3` segundos. Finalizando, quando o `SnackBar` é fechado, executamos a função para este momento, que em nosso caso será para resetar o formulário.

Como tudo implementado, já podemos consumir esta última técnica. Vamos substituir nosso `RaisedButton` no formulário pelo código a seguir.

```
RaisedButtonWithSnackbarWidget(
    onPressedVisible: formState.isFormValid,
    buttonText: 'Gravar',
    textToSnackBar: 'Os dados informados foram registrados com sucesso.',
    onButtonPressed: _onSubmitPressed,
    onStackBarClosed: _resetForm,
),
```

Como estamos mudando a maneira como o feedback é dado ao usuário, podemos retirar o *dialog* que temos utilizando nosso `SuccessDialogWidget`. Em nosso método `_mainColumn()`, vamos ajustar nosso `BlocBuilder` para o código a seguir, já removendo o código comentado. Note a inserção de um `Timer`. Optei por essa estratégia para que os controles sejam limpos após o `SnackBar` desaparecer. Aqui poderia ser pensado uma estratégia de bloqueio do formulário para evitar que o usuário tente inserir dados antes de o processo ser concluído.

```
child: BlocBuilder<PalavrasCrudFormBloc, PalavrasCrudFormState>(
        builder: (context, formState) {
  if (formState.formularioEnviadoComSucesso) {
    Timer(Duration(seconds: 4), () {
      _palavraController.clear();
      _ajudaController.clear();
      this._palavrasCrudFormBloc.add(FormReset());
    });
  }
  return _form(formState);
}),
```

Agora sim, vamos executar a aplicação e ver o resultado na figura a seguir.

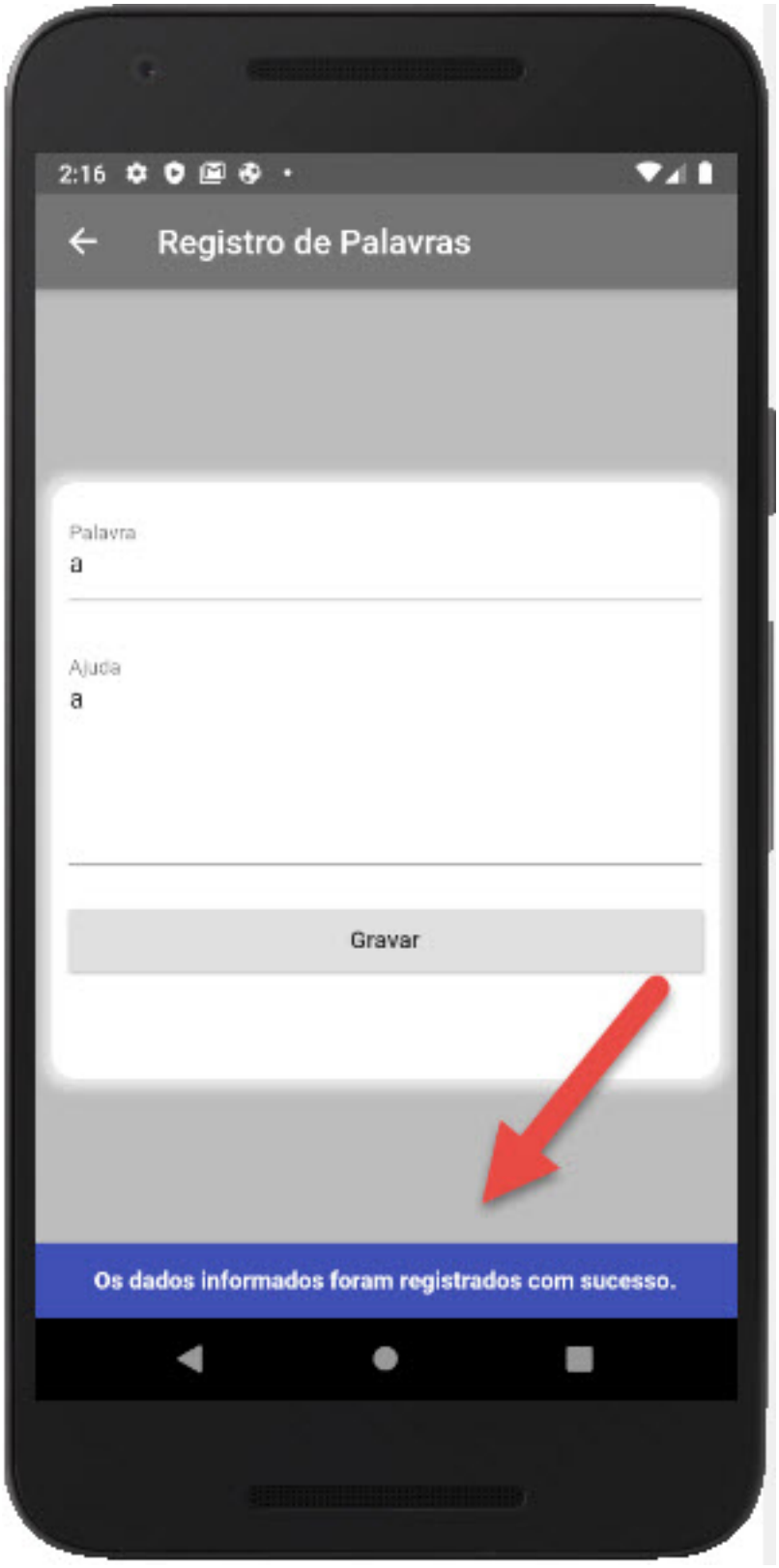

Figura 7.4: SnackBar de confirmação dos dados digitados

Conclusão

Este foi até agora o maior capítulo que tivemos neste livro, mas vimos muitas coisas boas e importantes, que auxiliarão você no desenvolvimento de suas aplicações. Trabalhamos uma gerência de navegação entre rotas nomeadas, criamos um formulário com toda lógica de navegação sendo realizada por meio de BLoC e ainda aprendemos sobre Mixins no Dart. Também criamos alguns widgets nossos para renderização de componentes de acordo com a plataforma em que estamos executando nossa aplicação.

Apenas para enumerar o que vimos neste capítulo, que neste momento foram apenas widgets, trago aqui a relação deles, o que pode lhe auxiliar em uma pesquisa futura específica para cada ponto trabalhado. São eles: `AlertDialog`, `CupertinoAlertDialog`, `FocusNode`, `SafeArea`, `SingleChildScrollView`, `TextEditingController` e `Route`.

No próximo capítulo, trabalharemos a persistência dos dados informados no formulário em uma base de dados, tudo com BLoC. Até daqui a pouco.

Capítulo 8
Persistência na inserção e BLoC na recuperação de dados e animação na transição de rotas

No capítulo anterior, trabalhamos muita coisa interessante. Dentre elas, o controle centralizado para navegação entre rotas nomeadas. Utilizamos novamente o BLoC, desta vez integrado com um formulário de entrada, buscando apresentar um pouco mais os recursos dos quais podemos usufruir desse sistema de gerenciamento de estado.

Criamos uma Rota (visão, página, formulário) responsável pelo registro de palavras que usaremos em nosso jogo, utilizamos plugins que se responsabilizam por criar os serviços necessários para mapeamento de objetos em JSON e também o caminho contrário. A manipulação de JSON é uma característica muito necessária quando consumimos objetos que venham de uma base de dados ou de um serviço web. Buscamos dividir responsabilidades ainda de maneira tímida fazendo uso de Mixins.

Os dados informados no formulário gerado no capítulo anterior não são persistidos, ou seja, não estão sendo armazenados em local algum, o que é inviável para uma aplicação que precisará desses dados para que o jogo possa ser executado. É com isso que trabalharemos neste e no próximo capítulo, onde também criaremos outra rota, agora para exibição dos dados registrados por nós.

Estes dois capítulos, juntos, são extensos e com um pouco de complexidade. Peço a você muita concentração e poder de abstração na leitura, além de grande atenção aos códigos, pois passaremos por vários processos de alteração. Teremos um pouquinho de trabalho, mas, ao final, tudo será compensado, pois veremos diversos novos widgets, recursos e técnicas. Prepare-se!

8.1 O SQLite como base de dados local

Já vimos, no capítulo 5, que podemos persistir dados em nosso dispositivo por meio de `Shared Preferences`, entretanto o domínio de dados é limitado, além de não ser viável persistirmos com esse recurso um número maior de dados que estejam integrados, como em uma base de dados relacional.

Felizmente, existe o SQLite, um banco de dados relacional com muitos recursos e que até os dias de hoje é unanimidade quando o tema é aplicativos para dispositivos móveis. Mas o SQLite não se limita a isso, é possível utilizá-lo em outras plataformas. Ocorre que bancos robustos, como Oracle, PostgreSQL e SQL Server não são possíveis (ainda, quem sabe) em dispositivos móveis.

Cada framework costuma ter componentes que realizam a interação de seu aplicativo em sua plataforma com o SQLite em seu dispositivo. E para o Flutter isso não é diferente, pois temos nele um pacote chamado `SQFLite` e é o que utilizaremos neste capítulo.

Preparação do ambiente de nosso projeto

Nosso primeiro passo para o uso do SQFLite é trazermos para nosso projeto as dependências necessárias para nossa aplicação. Dessa maneira, em seu `pubspec.yaml`, em `dependecies`, insira os dois pacotes a seguir. O `path_provider` será necessário para termos um local físico no dispositivo onde armazenar e recuperar nossa base de dados.

```
sqflite: ^1.3.0
path_provider: ^1.6.7
```

Criação da base de dados no SQLite

Em nossa pasta `lib`, vamos criar outra chamada `local_persistence`. Nela, implementaremos o que for necessário para nosso aplicativo persistir e recuperar dados.

Precisaremos informar alguns valores na configuração de nossa base de dados e, em vez de utilizarmos valores específicos onde eles serão necessários, vamos concentrar tudo em constantes, facilitando o reúso posterior e também correções que possam ser necessárias. Essa é uma boa prática a se seguir.

Em nossa pasta recém-criada, vamos criar um arquivo Dart chamado `lp_constants.dart` e inserir nele inicialmente o código a seguir.

```
const kDatabaseName = "jogodaforca.db";
const kDatabaseVersion = 1;
```

Ainda na nova pasta, vamos criar outro arquivo Dart chamado `database.dart` e, nele, implementaremos o código a seguir, que será o básico necessário para que nossa aplicação consiga se integrar ao `SQLite`.

```
import 'dart:io';

import 'package:path/path.dart';
import 'package:path_provider/path_provider.dart';
import 'package:sqflite/sqflite.dart';

import 'lp_constants.dart';

class SQFLiteDataBase {
  /// Objeto SQFLite para nossa base de dados
  static Database _database;

  /// Um construtor privado, atuando como um singleton para termos sempre
a mesma
  /// conexão em toda a aplicação
  SQFLiteDataBase._privateConstructor();
  static final SQFLiteDataBase instance =
SQFLiteDataBase._privateConstructor();

  /// Acesso à base de dados
  Future<Database> get database async {
    if (_database != null) return _database;
```

```
    _database = await _initDatabase();
    return _database;
  }

  /// Método que inicializará a base de dados, caso ainda não exista
  _initDatabase() async {
    Directory documentsDirectory = await
getApplicationDocumentsDirectory();
    String path = join(documentsDirectory.path, kDatabaseName);
    return await openDatabase(path,
        version: kDatabaseVersion,
        onCreate: _onCreateDb,
        onUpgrade: _nUpgradeDb,
        onDowngrade: _onDowngradeDb);
  }

  /// Métodos que serão executados de acordo com o estado identificado na
  /// inicialização.
  Future _onCreateDb(Database database, int version) async {}
  Future _onUpgradeDb(
      Database database, int previousVersion, int newVersion) async {}
  Future _onDowngradeDb(
      Database database, int previousVersion, int newVersion) async {}
}
```

Para o consumo dessa classe, não vamos querer que ela seja instanciada sempre que for necessário, por isso definimos os métodos públicos como `static`, o que nos leva também a definir uma propriedade `static`, que será utilizada nesses métodos. É um princípio básico de OO.

Definimos um método privado que atuará como construtor para nossa classe e, após ele, temos outra propriedade `static`, agora se referenciando ao nosso construtor privado. Em Dart, para criar um construtor privado, basta termos o nome da classe separado por um ponto do nome de um método, que por sua vez tem o nome iniciado com `_`, como em `SQFLiteDataBase._privateConstructor();`, extraído do código anterior.

Na sequência, você deve ter notado uma propriedade pública e estática chamada `database`. Ela é responsável por manter em `_database` a nossa base inicializada, e essa inicialização ocorrerá apenas uma vez durante o ciclo de execução da aplicação. Nosso método `_initDatabase()` é responsável pela criação física de nossa base e faz uso de recursos do package `path_provider`, que importamos há pouco. Nesse método, recuperamos o caminho padrão do dispositivo para a pasta de documentos da aplicação, depois unimos esse caminho ao nome de nossa base e a abrimos.

Note que, na abertura da base de dados, na invocação a `openDatabase()`, configuramos a versão da base e delegamos métodos para a criação inicial, atualização na estrutura da tabela para um `upgrade` ou um `downgrade`.

A tabela de palavras no SQLite

O código que implementamos no tópico anterior nos subsidiarão na criação e no acesso à base de dados para nossa aplicação em nosso dispositivo. Entretanto, uma base de dados sem tabelas é algo incomum, vamos então à implementação necessária para isso. Na mesma classe, substitua o método `_onCreateDb()` pela implementação a seguir.

```
Future _onCreateDb(Database database, int version) async {
    await database.execute("CREATE TABLE palavras ("
        "palavraID TEXT PRIMARY KEY,"
        "palavra TEXT,"
        "ajuda TEXT"
        ")");
}
```

Se você estiver habituado com SQL, o código anterior é facilmente compreendido. Estamos usando DDL para criação de uma tabela chamada `palavras` com dois campos, sendo o `palavraID` a chave primária.

É possível que você pense que a chave primária pudesse ser um campo do tipo numérico inteiro e autoincrementado, porém, como estamos trabalhando com dispositivos móveis e a aplicação pode ter sua base de dados depois conectada e sincronizada com a internet e diversos outros dispositivos, manteremos o tipo `TEXT` e usaremos, nas classes, um valor UUID (GUID). Logo chegaremos a isso.

Muito bem, já temos nossa estrutura da base de dados configurada para o início de nossas atividades. Agora, em nossa interface implementada no capítulo anterior, precisaremos implementar o código para que os dados informados sejam então persistidos.

8.2 A manipulação de dados no SQLite

Em nossa pasta `local_persistence`, vamos criar uma nova chamada `daos`, onde `DAO` significa *Data Access Object*, um padrão de acesso e manipulação de dados.

Como estamos trabalhando com o modelo de negócio da classe `Palavra`, criaremos um DAO específico para ela. Aqui, poderíamos aplicar princípios da OO e criar uma classe abstrata (Dart ainda não tem interface) e então implementar o contrato em nossos DAOs específicos, mas esse não é o foco aqui.

Como dito na seção anterior, nossa chave primária será baseada em um código UUID, ou GUID em algumas literaturas. Um package oferecido para o Flutter que utilizaremos é o `uuid`. Em nosso exemplo, usaremos a versão do código a seguir, que deve ser inserida nas dependências do `pubspec.yaml`.

```
dependencies:
  ... outras dependências
  uuid: ^2.0.4
```

Quando formos implementar nossas requisições no SQLite, haverá momentos em que precisaremos informar os nomes e colunas das tabelas envolvidas. Isso nos levaria ao uso de literais, textos entre aspas, o que pode ser ruim, não pela possível mudança de valores, pois isso, embora seja possível, não ocorre com frequência, mas sim pela possibilidade de precisarmos escrever várias vezes esses valores e em algum local errarmos o que estamos escrevendo.

Retomando à introdução de constantes, temos duas possibilidades para declaração de variáveis constantes em Dart, utilizando o `const` e o `final`. Para usar o `const`, precisamos ter o valor já sabido na declaração da variável. No caso do `final`, podemos declarar a variável e inicializá-la depois, mas a inicialização só poderá ocorrer uma vez, ou seja, não será possível alterar o valor. No primeiro caso, o valor também não pode ser alterado. Caso você queira compreender melhor, o link https://news.dartlang.org/2012/06/const-static-final-oh-my.html traz um excelente post cuja leitura recomendo.

Em nosso arquivo `lp_constants.dart`, implemente a constante a seguir. Em Dart, é uma convenção que as constantes comecem com a letra `k`.

```
const kPalavrasTableName = "palavras";
```

Como dito anteriormente, precisaremos ter a funcionalidade de transformação de nossos objetos relacionados a palavras em JSON e de JSON em objeto. Nós poderíamos tranquilamente implementar toda essa lógica, mas vamos usar componentes que nos possibilitam isso. Vamos ao registro dessas dependências. Mantenha os que já temos no `pubspec.yaml`, apenas inserindo os que estão a seguir.

```
dependencies:
  json_annotation: ^3.0.1

dev_dependencies:
```

```
  build_runner: ^1.9.0
  json_serializable: ^3.3.0
```

Por meio desses componentes, serão injetados códigos em nosso projeto que serão responsáveis por toda essa transformação de objetos em Json e vice-versa. O `build_runner` será o responsável pela execução de atualizações via terminal. Precisamos adaptar um pouco nossa classe `PalavraModel`. Veja a nova implementação na sequência, seguida por comentários. Alguns erros surgirão, mas não se preocupe.

```
import 'package:equatable/equatable.dart';
import 'package:json_annotation/json_annotation.dart';

part 'palavra_model.g.dart';

@JsonSerializable()
class PalavraModel extends Equatable {
  String palavraID;
  final String palavra;
  final String ajuda;

  @override
  List<Object> get props => [palavraID];

  PalavraModel({this.palavraID, this.palavra, this.ajuda});

  factory PalavraModel.fromJson(Map<String, dynamic> json) =>
      _$PalavraModelFromJson(json);
  Map<String, dynamic> toJson() => _$PalavraModelToJson(this);
}
```

Logo no início do código anterior, temos uma instrução `part 'palavra_model.g.dart';`, onde determinamos que nosso arquivo atual terá parte de seu código em outro arquivo com o nome parecido, seguido apenas de `.g`, pois essa é uma convenção e o `g` vem de `generated`.

Em seguida, temos uma anotação `decorator`, o `@JsonSerializable()`, que é parte do `json_annotation` que registramos como dependência.

Esse componente possui diversos outros recursos, mas deixo essa investigação a seu cargo.

Também retiramos o `final` de `palavraID`, pois o valor dela será atribuído no momento da inserção, pois já receberá um objeto de `PalavraModel`.

Precisamos agora gerar o código que corrigirá o erro que temos em nossa classe. Para isso, precisamos ir ao terminal na pasta de nosso projeto e inserir a instrução a seguir. É importante que você realize um `pub get` antes de executar este código.

```
// código normal
flutter packages pub run build_runner build

// código para caso apareçam erros na execução
flutter packages pub run build_runner build --delete-conflicting-outputs
```

Atenção: é importante que você saiba que a cada alteração nas propriedades da classe, a execução dessa instrução se faz necessária.

Após a execução bem-sucedida, o novo arquivo fará parte do seu projeto e os erros anteriores terão sido corrigidos.

Agora sim, já temos os pré-requisitos para criar nossa primeira classe DAO. Na pasta `daos`, vamos criar o arquivo `palavra_dao.dart` e, nele, o código a seguir. Após sua leitura do código, teremos algumas considerações.

```
import 'package:meta/meta.dart';
import 'package:sqflite/sqlite_api.dart';
import 'package:uuid/uuid.dart';
import 'package:cc04/models/palavra_model.dart';

import '../database.dart';
import '../lp_constants.dart';

class PalavraDAO {
  Future<String> insert({@required PalavraModel palavraModel}) async {
```

```
    String result;
    try {
      Database lpDatabase = await SQFLiteDataBase.instance.database;

      palavraModel.palavraID  = Uuid().v1();
      result = palavraModel.palavraID;

      var recordsAffected =
          await lpDatabase.insert(kPalavrasTableName,
palavraModel.toJson());
      if (recordsAffected == 0) result = null;
    } catch (exception) {
      rethrow;
    }
    return result;
  }
}
```

Nosso método retornará um `Future` tipificado em uma String, pois, em caso de sucesso, queremos que quem o invoque receba o valor gerado para o `ID`. Toda a execução do método está em um bloco `try...catch`, que, no segundo caso, dispara novamente a exceção para quem o invocou, possibilitando um tratamento de erros diretamente na interface com o usuário. Há técnicas mais apuradas para esse tipo de tratamento de erros, mas esse não é o nosso foco aqui.

Já dentro do `try`, temos a recuperação da base de dados fornecida por nosso `Singleton`. Após isso, geramos nosso `UUID` com a invocação ao `v1()`. O pacote oferece diversos outros métodos cuja documentação creio ser interessante que você leia, mas o `v1()` supre as necessidades para nosso projeto.

O método `insert()` de `Database` precisa receber dois parâmetros, o nome da tabela, que está na constante enviada como primeiro parâmetro, e um mapa para as colunas e seus valores, e isso é criado pelo nosso método `toJson()`, que já temos implementado em nossa classe de modelo. Caso a inserção ocorra bem, um valor inteiro será retornado, representando a quantidade de registros que

foram inseridos. Em nosso caso, é para recebermos o valor 1, mas se o valor 0 for retornado, algum erro ocorreu e será preciso verificar.

Essa última verificação é algo desnecessário para nosso método `insert`, mas é algo extremamente necessário para a atualização e remoção que veremos mais adiante. Para manter o padrão, eu preferi deixar também nesse método. Veja que, em caso de exceção, chamamos `rethrow` para que ela possa ser novamente disparada e capturada por nós na interface com o usuário.

8.3 Utilização do DAO em nossa visão de inserção

Temos toda a estrutura de criação e o acesso à base de dados implementados, assim como o método responsável pela inserção de um registro referente aos dados para a palavra informada em nossa visão. Precisamos agora consumir esse método.

Inicialmente precisamos realizar uma alteração em nosso widget `RaisedButtonWithSnackbarWidget`, pois agora temos uma situação possível de erro na invocação do evento `onPressed` do `RaisedButton` que temos. Com isso, precisamos ter uma mensagem caso esse erro ocorra. Vamos então inserir a seguinte propriedade no widget, lembrando também de configurá-la no construtor.

```
final String failTextToSnackBar;
```

Também, por semântica, mudaremos o nome da propriedade `textToSnackBar` para `successTextToSnackBar`.

No capítulo anterior, implementamos todo o comportamento destinado ao evento `onPressed` no próprio método `build()`. Isso não é recomendado. Não faz parte das responsabilidades desse método fazer isso. Vamos então criar um método privado na classe após o

`build()` com o código a seguir, que possui alterações necessárias para nossa nova realidade.

```
_onPressedRaisedButton(BuildContext buildContext) async {
    String textToSnackBar = this.successTextToSnackBar;
    bool success = true;
    FocusScope.of(buildContext).unfocus();
    try {
      await this.onButtonPressed();
    } catch (e) {
      textToSnackBar = this.failTextToSnackBar + ': ' + e.toString();
      success = false;
    }

    Scaffold.of(buildContext)
        .showSnackBar(
          SnackBar(
            backgroundColor: (success) ? Colors.indigo : Colors.red,
            content: Text(
              textToSnackBar,
              textAlign: TextAlign.center,
              style: TextStyle(
                fontWeight: FontWeight.bold,
                fontSize: (success) ? 14 : 16,
              ),
            ),
            duration: Duration(seconds: (success) ? 3 : 5),
          ),
        )
        .closed
        .then((_) => (success) ? this.onSnackBarClosed() : () {});
}
```

Verifique que temos duas novas variáveis, `textToSnackBar` e `success`. A primeira, com base no sucesso ou não da invocação do método `onButtonPressed()` que vem de nosso CRUD, terá uma mensagem de sucesso ou de falha. Observe isso no `try...catch`. A mesma condição realiza a atribuição negativa à segunda variável, que tem seu valor inicial atribuído com `true`.

Com essas variáveis tendo seus devidos valores atribuídos, realizamos testes com o operador temário `?` para atribuir o `backgroundColor`, `fontSize`, `duration` e o que fazer no fechamento do `SnackBar`. Veja também que o `Text` para `content` recebe o valor de `textToSnackBar`.

Muito bem, com nossas devidas melhorias realizadas, podemos trabalhar na nova invocação ao nosso widget que, por didática, trago toda na sequência. Ela está em nosso `palavra_crud_route.dart` no método `_form()`.

```
RaisedButtonWithSnackbarWidget(
    onPressedVisible: formState.isFormValid,
    buttonText: 'Gravar',
    successTextToSnackBar:
        'Os dados informados foram registrados com sucesso.',
    failTextToSnackBar: 'Erro na inserção',
    onButtonPressed: _onSubmitPressed,
    onSnackBarClosed: _resetForm,
),
```

Agora, precisamos implementar a real funcionalidade no método `_onSubmitPressed()`, que implementamos no capítulo anterior. Veja o código a seguir. Verifique a instanciação de nosso `DAO` de nosso `PalavraModel` e o bloco `try...catch` para a inserção da palavra que teve seus dados informados em nosso CRUD. Atente-se aos imports.

```
void _onSubmitPressed() async {
    PalavraDAO palavraDAO = PalavraDAO();
    PalavraModel palavraModel = PalavraModel(
        palavra: this._palavraController.text,
        ajuda: this._ajudaController.text);

    try {
      await palavraDAO.insert(palavraModel: palavraModel);
      _palavrasCrudFormBloc.add(FormSuccessSubmitted());
    } catch (e) {
      rethrow;
```

```
    }
}
```

Verificou o uso de `rethrow` em dois de nossos códigos recentes? No método `insert()` de nosso `DAO`, em caso de algum erro na inserção do registro, precisamos capturar a exceção no `_onSubmitPressed()`, no `try...catch` deste método. Neste momento, precisamos que essa exceção seja capturada em nosso `RaisedButtonWithSnackbarWidget`, o que fica relativamente simples com o `rethrow`. Existem diversas técnicas para a captura e o tratamento de erros, mas não nos estenderemos nesse tema.

Vamos testar nossa aplicação? A figura a seguir traz dois momentos no processo de registro da palavra. Notou o `async` na assinatura do método?

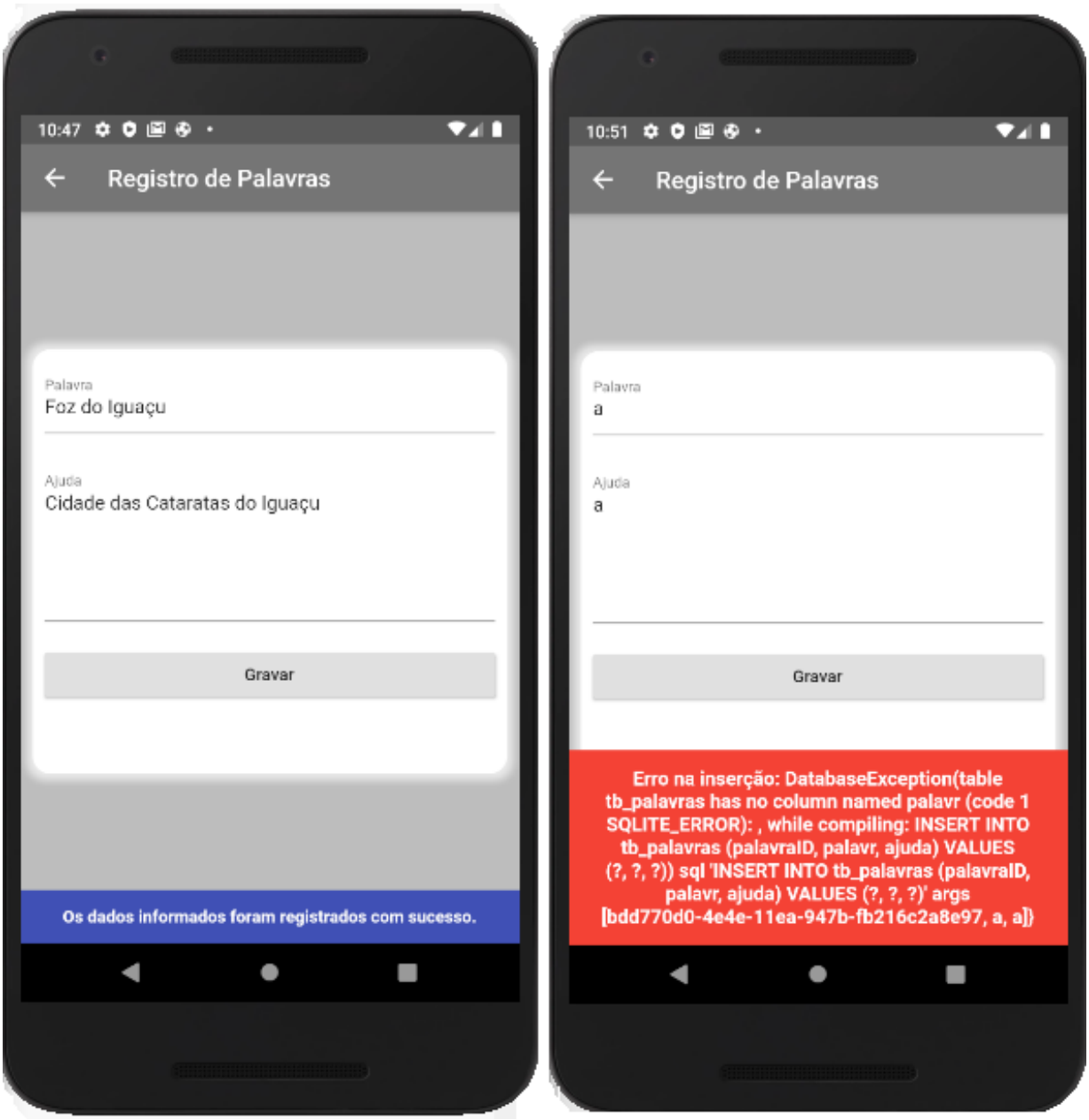

Figura 8.1: Registro de palavras no SQLite

Apenas para ilustrar, a figura anterior trouxe a informação que será exibida no sucesso da inserção e a informação que será exibida caso algo de errado aconteça nesse processo. No caso, eu forcei o erro tentando usar o nome de uma coluna inexistente na tabela.

8.4 A visualização de todas as palavras já registradas no SQLite

Em nosso `Drawer`, temos a opção de vermos todas as palavras já registradas e é essa a funcionalidade que implementaremos nesta seção. Para isso, começaremos criando nosso arquivo que conterá nossa rota para a listagem de palavras. Sendo assim, na pasta `palavras`, dentro de `routes`, crie o arquivo `palavras_listview_route.dart`, que deverá possuir inicialmente o código a seguir. Optamos por um `Stateful`, pois teremos atualização de estados de widgets nesta visão.

```
import 'package:flutter/material.dart';

class PalavrasListViewRoute extends StatefulWidget {
  @override
  _PalavrasListViewRouteState createState() =>
_PalavrasListViewRouteState();
}

class _PalavrasListViewRouteState extends State<PalavrasListViewRoute> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

O objetivo aqui é que tenhamos uma listagem das palavras registradas, mas que não sejam trazidas todas de uma vez da base de dados. Traremos um determinado número de registro e, conforme o usuário vá rolando a listagem, novas palavras serão trazidas.

Faremos novamente o uso de BloC, agora para a recuperação de registros da base com paginação, tendo sempre o primeiro registro a ser recuperado e a quantidade máxima a partir dele. Mas vamos por etapas em um processo de construção.

Nosso primeiro passo será a implementação do método que recuperará as palavras na base de dados e veremos isso a seguir.

Recuperação de dados em nosso DAO

Antes de implementarmos o método de recuperação das palavras, precisamos criar algumas constantes para nos referirmos às colunas de nossa tabela de palavras. Faremos isso em nosso arquivo `lp_constants.dart`, tal qual podemos ver no código a seguir.

```
const kPalavraPalavraID = 'palavraID';
const kPalavraPalavra = 'palavra';
const kPalavraAjuda = 'ajuda';
```

Nós ainda temos uma situação em relação às letras acentuadas em nossas palavras em português e, como inicialmente traremos nossos registros classificados em ordem alfabética pela palavra registrada, precisamos nos preocupar com isso. Faremos as implementações necessárias, mas, se depois você quiser comentar o que faremos aqui para ver o resultado sem esse recurso, fique à vontade.

Este workaround foi necessário devido ao fato de o resultado da consulta realizada não respeitar caracteres acentuados e letras maiúsculas e minúsculas ao mesmo tempo. Recorri à documentação que orientou o uso de `COLLATE` tanto na criação da tabela como na consulta. Utilizei recursos do plugin também para inserir o SQL puro, sem fazer uso de seus recursos, mas não consegui o resultado desejado. Talvez seja algum bug da versão. Se, ao ler esta seção, você ver esse problema resolvido, ficarei feliz em saber.

Vamos lá então. Crie uma nova pasta chamada `functions` em `lib` e, nela, crie um arquivo chamado `string_functions.dart` com o conteúdo a seguir. Observe que criamos um `Map`, onde a chave são as letras acentuadas e os valores, as letras sem acento.

Nossa função `removerAcentos` receberá uma palavra, um texto, uma letra, que pode ou não ter acentos. Após a declaração do `Map`, transformamos o argumento recebido em um `List`. Em seguida, varremos essa lista buscando nela caracteres acentuados de acordo com nosso `Map` e, em caso positivo, substituímos o valor acentuado pelo valor sem acento. Após esse processo, a função retorna o argumento recebido com as letras não acentuadas. Isso nos auxiliará na classificação alfabética.

```
String removerAcentos(String s) {
  Map<String, String> letrasAcentuadas = {
    "â": "a",
    "Â": "A",
    "à": "a",
    "À": "A",
    "á": "a",
    "Á": "A",
    "ã": "a",
    "Ã": "A",
    "ê": "e",
    "Ê": "E",
    "è": "e",
    "È": "E",
    "é": "e",
    "É": "E",
    "î": "i",
    "Î": "I",
    "ì": "i",
    "Ì": "I",
    "í": "i",
    "Í": "I",
    "õ": "o",
    "Õ": "O",
    "ô": "o",
    "Ô": "O",
    "ò": "o",
    "Ò": "O",
    "ó": "o",
    "Ó": "O",
    "ü": "u",
```

```
    "Ü": "U",
    "û": "u",
    "Û": "U",
    "ú": "u",
    "Ú": "U",
    "ù": "u",
    "Ù": "U",
    "ç": "c",
    "Ç": "C"
  };

  List<String> origem = s.split('');
  for (int i = 0; i < origem.length; i++) {
    origem[i] = (letrasAcentuadas[origem[i]] != null)
        ? letrasAcentuadas[origem[i]]
        : origem[i];
  }
  return origem.join();
}
```

Precisamos importar esse nosso novo arquivo no `palavra_dao.dart`. Basta inserir a instrução a seguir nos imports.

```
import '../../functions/strings_functions.dart' as StringFunctions;
```

Agora que já temos as constantes que precisaremos em nossa classe `PalavraDAO`, vamos implementar o seguinte método. Observe que declaramos um `List` de um `Map` nele, onde teremos uma `String` como chave e um `dynamic` para o valor armazenado. Cada chave representará um campo da tabela e cada `Map` um registro dela.

```
Future<List> getAll({int startIndex, int limit}) async {
    List<Map<String, dynamic>> dataList = List();
    try {
      Database lpDatabase = await SQFLiteDataBase.instance.database;

      var result = await lpDatabase.query(
        kPalavrasTableName,
        columns: [kPalavraPalavraID, kPalavraPalavra, kPalavraAjuda],
        offset: startIndex ?? null,
```

```
        limit: limit ?? null,
        orderBy: '$kPalavraPalavra COLLATE LOCALIZED',
      );

      dataList = result.toList();
      dataList.sort((a, b) {
        return
StringFunctions.removerAcentos(a[kPalavraPalavra].toLowerCase())
            .compareTo(StringFunctions.removerAcentos(
                b[kPalavraPalavra].toLowerCase()));
      });

      return dataList;
    } catch (exception) {
      rethrow;
    }
}
```

Ainda em relação ao código, temos a obtenção da instância de nossa base de dados e com ela executamos o método `query()`, que recebe o nome da tabela onde a consulta será executada e valores para os parâmetros `columns`, `offset`, `limite` e `orderby`. Vamos às explicações sobre o que é cada um.

`columns`, semanticamente, fica fácil de notar que se refere às colunas que queremos que estejam no `select` da consulta SQL; `offset` terá opcionalmente o índice do primeiro registro que deve ser recuperado; `limit` se refere à quantidade máxima de registros que devem ser recuperados na consulta. Observe que as propriedades `offset` e `limit` são recebidas, opcionalmente, pelo método `getAll()`. Mais à frente comento sobre o `orderBy`.

Com o resultado da consulta na variável `result`, precisamos atribuir seu valor para `dataList`, pois `result` será apenas leitura, e precisamos classificar o resultado levando em conta as palavras com letras acentuadas. Se a classificação não precisasse passar por esse processo, poderíamos, na chamada de `query()`, fazer uso da propriedade `orderBy`.

Observe a chamada à `sort()` de nosso `List`. Como argumento, temos uma função que receberá dois argumentos, os quais serão comparados pelo método `compareTo()` do valor de `a`, que é o que queremos comparar com `b`. Veja que invocamos a função `removerAcentos()` para os dois valores. Note também que transformamos, apenas para a comparação, todos em letras minúsculas com `toLowerCase()`. O método `compareTo()` retornará a informação de quem é `menor` entre os dois valores e então retornará isso à `sort` para que a classificação possa ser executada.

8.5 BLoC para a recuperação dos dados

Poderíamos pensar em ter na nossa visão todos os dados sendo trazidos de uma única vez da base de dados e atribuindo-os a um `ListView` que apenas trate a rolagem desses dados. Isso seria fantástico se tivéssemos apenas um número pequeno de registros, mas podemos pensar em dezenas, centenas e milhares, o que resultaria em um enorme custo para nossa aplicação, além de deixar nosso usuário esperando, provavelmente pensando que a aplicação travou.

Já implementamos nosso método `getAll()` prevendo a possibilidade de `paginação`, ou seja, enviamos a ele a posição inicial em que queremos que a consulta comece e a quantidade de registros que devem ser recuperadas a cada requisição ao `query()`.

Com esse comportamento, precisamos pensar então em dar ao nosso `ListView` a possibilidade de carregar mais ao chegar ao último registro procurado. Isso pode ser nomeado de `ListView infinito`, embora o fim dele chegue quando não houver mais registros. Não queremos fazer isso com o `setState()`, pois já sabemos da existência do `BLoC` e o usaremos aqui.

A classe do estado do ListView

Em nossa pasta `lib\routes\palavras\bloc`, crie uma nova, chamada `listview` e, nela, um arquivo chamado `palavras_listview_state.dart`. Esse arquivo conterá uma classe que mapeará o estado do `ListView` em relação aos dados que o popularão. Pode ser vista como uma aplicação do padrão MVVM, que faz uso de classes para o `Model`, `View` e `View-Model`, sendo a classe que implementaremos a de `View-Model`, como comentamos quando criamos o BLoC para o registro de palavras. Pelo fato de a classe ser um pouco grande, vou apresentá-la por partes. A primeira está na sequência.

```
abstract class PalavrasListViewBlocState {
  const PalavrasListViewBlocState();
}

class PalavrasListViewBlocUninitialized extends PalavrasListViewBlocState
{}

class PalavrasListViewBlocError extends PalavrasListViewBlocState {
  final errorMessage;

  PalavrasListViewBlocError({this.errorMessage});
}
```

Iniciamos a definição da classe base como sendo abstrata e possuindo um construtor padrão. Teremos diferentes estados em nossa visão em relação à recuperação de nossos dados, neste caso, as palavras. Poderíamos pensar em termos enumeradores para isso, mas a documentação oficial do `package` de BLoC que estamos utilizando recomenda o uso de classes, e você verá na sequência que isso é extremamente importante, pois podemos ter estados que implementam comportamentos (métodos) e características (propriedades).

Nossos dois primeiros estados são: `PalavrasListViewBlocUninitialized` e `PalavrasListViewBlocError`. Os nomes são por si só autoexplicativos. O primeiro estado caracteriza que os dados ainda não foram inicializados e o segundo, alguma situação de erro, tendo inclusive a possibilidade de receber uma mensagem de erro.

Precisamos agora trabalhar o cenário real, quando as palavras, nossos dados, tiverem efetivamente sido carregadas e estejam prontas para serem disponibilizadas ao nosso `ListView`. Veja essa implementação na sequência, logo após o código anterior e, após ela, algumas considerações. Importe o modelo.

```
class PalavrasListViewLoaded extends PalavrasListViewBlocState {
  final List<PalavraModel> palavras;
  final bool hasReachedMax;

  const PalavrasListViewLoaded({
    this.palavras,
    this.hasReachedMax,
  });

  PalavrasListViewLoaded copyWith({
    List<PalavraModel> palavras,
    bool hasReachedMax,
  }) {
    return PalavrasListViewLoaded(
      palavras: palavras ?? this.palavras,
      hasReachedMax: hasReachedMax ?? this.hasReachedMax,
    );
  }

  @override
  String toString() =>
      'PalavrasListViewLoaded { palavras: ${palavras.length},
hasReachedMax: $hasReachedMax }';
}
```

Possuímos duas propriedades públicas na classe anterior, `palavras` e `hasReachedMax`. A primeira conterá as palavras recuperadas e que serão utilizadas pelo `ListView`. A segunda, booleana, indica quando o `ListView` exibirá a última palavra carregada na primeira propriedade. Temos também um construtor, que receberá, opcionalmente, valores para essas propriedades.

Na sequência, temos um método comum em boa parte das classes e widgets do Dart e Flutter, o `copyWith()`, que possibilita uma

clonagem de nosso objeto com alteração de valores particulares e necessários a um novo objeto. Veja que todos os argumentos são opcionais. Já tivemos esse método implementado no BLoC do capítulo anterior.

Ao final, apenas como padrão de OO, sobrescrevemos o `toString()` caso queiramos, principalmente, depurar com o uso de console o estado de objetos dessa classe.

As classes dos eventos de transição para o BLoC

Precisamos agora implementar os métodos que capturarão os eventos responsáveis pela transição de estado de nossa visão com o ListView. Veja o código a seguir, que deve ser codificado em um arquivo chamado `palavras_listview_event.dart`, na mesma pasta onde criamos o arquivo anterior. Procure observar que seguimos com o padrão de uma classe abstrata, que permite especialização agora para os eventos.

```
abstract class PalavrasListViewBlocEvent {}

class PalavrasListViewBlocEventFetch extends PalavrasListViewBlocEvent {}

class PalavrasListViewBlocEventResetFetch extends
PalavrasListViewBlocEvent {}
```

Estamos atribuindo nomes específicos para o modelo que estamos trabalhando, `Palavra`, mas poderíamos pensar em algo mais generalizado em alguns casos, como este, que poderia ser utilizado por demais modelos da aplicação.

A primeira classe representa os momentos em que os dados serão carregados e a segunda terá como indicação o momento em que os dados recuperados devem ser resetados para uma nova carga, ou seja, um novo `Fetch`. Lembra que no capítulo anterior comentei que eventos representam momentos de interação com o usuário?

A classe que implementa o BLoC

Chegamos ao terceiro arquivo de nossa arquitetura, o BLoC em si, tal qual fizemos no capítulo anterior. É nesta classe que implementaremos toda a regra de negócio ligada ao nosso `ListView`, que exibirá as palavras registradas pelo usuário. Lembre-se de que o objetivo é sempre retirar a lógica de negócio da visão.

Vamos criar o arquivo `palavras_listview_business_bloc.dart` na mesma pasta utilizada nas duas últimas implementações. Com o arquivo criado, implemente nele o código da listagem a seguir. Como o código para essa classe é um pouco extenso, apresentei por partes, sempre seguidas de explicações. Veja a primeira parte a seguir. Erros surgirão, mas fique tranquilo.

```
import 'package:cc04/local_persistence/daos/palavra_dao.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:meta/meta.dart';

import 'palavras_listview_event.dart';
import 'palavras_listview_state.dart';

class PalavrasListViewBloc
    extends Bloc<PalavrasListViewBlocEvent, PalavrasListViewBlocState> {
  final PalavraDAO palavraDAO;

  bool _hasReachedMax(PalavrasListViewBlocState state) => state is
PalavrasListViewLoaded && state.hasReachedMax;

  PalavrasListViewBloc({@required this.palavraDAO}) : assert (palavraDAO
!= null);

  @override
  get initialState => PalavrasListViewBlocUninitialized();

}
```

Na declaração da classe, temos a extensão de `Bloc`, já parametrizando nossas classes de eventos e de estados - algo que

já fizemos anteriormente em outras aplicações.

Nosso construtor trivial receberá o nosso DAO, que está como opcional, para facilitar na codificação, mas deverá ser informado. Veja o `assert` por segurança e o `@required`. Sabemos que o DAO recebido será automaticamente atribuído à propriedade `palavraDAO` pelo uso de `this`.

Ainda antes do construtor, temos um método booleano, o `_hasReachedMax()`, privado, que retornará verdadeiro caso o estado que ele receba como argumento seja o `PalavrasListViewLoaded` e já tenha sido alcançado o máximo de registros estipulados pela paginação. Lembre que temos a propriedade `hasReachedMax` definida em nossas classes de estado.

Terminando o código anterior, temos a definição para o estado inicial de nosso BLoC.

Na sequência, precisamos abstrair que o usuário pode rolar rapidamente nosso `ListView` e disparar uma chamada repetitiva e excessiva ao nosso BLoC. Precisamos impedir isso, dando, a cada requisição, um tempo mínimo de espera. É aqui que entra o `RxDart` com o método `debounceTime()`. Precisamos então registrar nossa dependência para o `RxDart` em nosso `pubspec.yaml`. Veja isso na sequência.

```
dependencies:
  rxdart: ^0.24.0
```

Após o `Pub get` é preciso importar o pacote em nosso BloC. Veja a importação:

```
import 'package:rxdart/rxdart.dart';
```

Agora, veja a implementação para o método comentado anteriormente, que deve estar abaixo do código anterior, que sobrescreve o método `transformEvents()` de `Bloc`.

```
  @override
  Stream<Transition<PalavrasListViewBlocEvent, PalavrasListViewBlocState>>
      transformEvents(
    Stream<PalavrasListViewBlocEvent> events,
    TransitionFunction<PalavrasListViewBlocEvent,
PalavrasListViewBlocState>
        next,
  ) {
    return super.transformEvents(
      events.debounceTime(
        Duration(milliseconds: 500),
      ),
      next,
    );
  }
}
```

Note que temos o recebimento dos eventos pelo BLoC e uma função representando um estado que terá o evento a ser executado como argumento armazenado em `next`. O retorno para esse método é simples. Ele aguarda meio segundo e então invoca o próximo evento.

Vamos aproveitar as implementações de que necessitaremos para codificar o método privado, que será responsável pela invocação de nosso `getAll()` no DAO que já temos implementado. Este código está na sequência. Veja que ele recebe a posição inicial e limite para leitura, valores que serão delegados ao `getAll()`. Com o `List` que recebemos, mapeamos cada elemento para um objeto de `PalavraModel` e, então, após todos os objetos serem recuperados, são mapeados para um `List` tipificado já na assinatura do método como `List<PalavraModel>`, sendo também o generics para `Future`, pois esse método é assíncrono. Veja que trabalhamos com previsão de exceção e a disparamos novamente. É preciso importar o model.

```
Future<List<PalavraModel>> _fetchPalavras(int startIndex, int limit) async
{
    try {
      final List data =
```

```
          await this.palavraDAO.getAll(startIndex: startIndex, limit:
limit);
      return data.map((palavra) {
        return PalavraModel.fromJson(palavra);
      }).toList();
    } catch (exception) {
      rethrow;
    }
}
```

Agora trabalharemos a lógica relacionada efetivamente à captura de eventos e mapeamento deles para novos estados, que estão diretamente relacionados a como nossa visão se comportará. O método todo é apresentado na sequência. Faça uma leitura e depois veja os comentários e esclarecimentos.

```
@override
Stream<PalavrasListViewBlocState>
mapEventToState(PalavrasListViewBlocEvent event) async* {
    final currentState = state;

    if (event is PalavrasListViewBlocEventResetFetch) {
      yield PalavrasListViewBlocUninitialized();
      return;
    }

    if (event is PalavrasListViewBlocEventFetch &&
!_hasReachedMax(currentState)) {
      try {
        if (currentState is PalavrasListViewBlocUninitialized) {
          final palavras = await _fetchPalavras(0, 20);
          yield PalavrasListViewLoaded(
              palavras: palavras, hasReachedMax: (palavras.length >= 20) ?
false : true);
          return;
        }
        if (currentState is PalavrasListViewLoaded) {
          final palavras =
              await _fetchPalavras(currentState.palavras.length, 20);
          yield palavras.isEmpty
```

```
              ? currentState.copyWith(hasReachedMax: true)
              : PalavrasListViewLoaded(
                  palavras: currentState.palavras + palavras,
                  hasReachedMax: false,
                );
        }
      } catch (exception) {
        yield PalavrasListViewBlocError(errorMessage: exception);
      }
    }
}
```

Começamos o método declarando e inicializando `currentState` com o estado atual para o BLoC. Poderíamos não ter essa variável, mas fica mais fácil para sabermos com qual estado estamos trabalhando.

Em seguida, temos um `if()`, que verifica se o evento recebido é para resetar o estado do BLoC para ainda não inicializado (`PalavrasListViewBlocUninitialized`), o que forçará uma recuperação a partir do início do select de nosso `getAll()`.

Depois, em um novo `if()`, temos uma dupla verificação, onde se espera um evento de `fetch` (`PalavrasListViewBlocEventFetch`) e que o máximo de registros para o estado atual ainda não tenha sido alcançado. Com essa dupla condição sendo avaliada como verdadeira, verificamos inicialmente se o estado atual é de não inicializado e, se for, então realizamos a recuperação inicial.

Na sequência do código, verificamos se o evento é o de visão já inicializada, carregada (`PalavrasListViewLoaded`). Se for, começamos a recuperação tendo como posição inicial a quantidade de palavras já recuperadas para o estado atual. Note que definimos o tamanho para cada paginação em 20, nos dois casos comentados. Poderíamos ter esse valor em uma constante, mas isso fica com você.

Com base no valor devolvido por `_fetchPalavras()`, se foi ou não retomado algo, mudamos o valor para `hasReachedMax`, ou retornamos um novo estado, já com a concatenação das palavras anteriormente

recuperadas com as novas. Em caso de algum tipo de erro, retornamos como estado um objeto de `PalavrasListViewBlocError`. Observe que, como a cada `if()` temos um retorno, optei por não fazer uso do `else`.

Organizando os pacotes para importação

Criamos três arquivos e, toda vez que formos utilizar os recursos deles, teremos que importar os três. Isso é um pouco verboso e trabalhoso. O Dart traz um recurso de exportação de pacotes que nos permite criar um arquivo e nele ter a exportação de diversos outros, o que nos permite importar um único arquivo no uso dos recursos implementados em três. Na mesma pasta, crie o arquivo `palavras_listview_bloc.dart` e insira nele o código da sequência.

```
export 'palavras_listview_event.dart';
export 'palavras_listview_state.dart';
export 'palavras_listview_business_bloc.dart';
```

8.6 A rota para a visualização de palavras registradas

Agora que temos nossa arquitetura toda implementada, precisamos começar a desenhar nossa rota com o formulário para registro de palavras. Teremos alguns comportamentos que poderão ser implementados em um `Mixin`, tal qual fizemos no capítulo anterior.

Dessa maneira, antes de começarmos com nossa rota, na pasta `\lib\routes\palavras`, vamos criar outra, chamada `mixin` e, nela, um arquivo chamado `palavras_listview_mixin.dart` com o código apresentado na sequência.

```
import 'package:flutter/material.dart';

mixin PalavrasListViewMixin {
  centerText({String text}) {
    return Center(child: Text(text));
  }
}
```

Vamos começar nossa visão de listagem de palavras. Vamos trabalhar com o arquivo `palavras_listview_route.dart`, que já temos criado, retomando um `Container()`. Agora vamos mudar para o código a seguir. Este código terá evoluções e complexidades, por isso optei por exibi-lo por partes. Poderemos ver neste código um simples `Scaffold`, com um `Container` como `body`. Observe o `with` para nosso mixin.

```
import 'package:flutter/material.dart';
import
'package:capitulo09_persistencia_e_anim/routes/palavras/mixin/palavras_lis
tview_mixin.dart';

class PalavrasListViewRoute extends StatefulWidget {
  @override
  _PalavrasListViewRouteState createState() =>
_PalavrasListViewRouteState();
}

class _PalavrasListViewRouteState extends State<PalavrasListViewRoute>
with PalavrasListViewMixin {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        centerTitle: true,
        title: Text('Palavras registradas'),
        elevation: 10,
      ),
      body: Container(),
    );
```

```
  }
}
```

Não temos nada funcional ainda em relação à listagem de palavras registradas, mas vamos implementar a navegação de nosso `Drawer` até o que temos implementado nesse momento. Iniciamos inserindo uma nova constante em nosso arquivo `/lib/appconstants/router_constants.dart`, como mostro a seguir.

```
const String kPalavrasAllRoute = '/palavrasAll';
```

Agora, em nossa classe `AppRouter`, no arquivo `/lib/apphelpers/app_router.dart`, vamos inserir um novo `case` para a nova rota. Veja a seguir. Lembre-se do import.

```
case kPalavrasAllRoute:
    return MaterialPageRoute(builder: (_) => PalavrasListViewRoute());
```

Por fim, em nosso arquivo `/lib/drawer/widgets/drawerbodycontent_app.dart`, na classe `DrawerBodyContentApp`, configuramos o evento `onTap` para acessar nossa visão. Veja o trecho completo dessa inserção na sequência.

```
ListTileAppWidget(
    titleText: 'Palavras existentes',
    subtitleText: 'Vamos ver as que já temos?',
    onTap: () {
      Navigator.of(context).pop();
      Navigator.of(context).pushNamed(kPalavrasAllRoute);
    },
),
```

Agora podemos executar nossa aplicação e, no `Drawer`, selecionar a opção `Palavras existentes`. Com isso, teremos uma visão básica, exibindo apenas a `AppBar`.

Criação de um widget para as palavras

Já sabemos que usaremos um `ListView` e, como filhos dele, podemos utilizar qualquer widget, mas faremos uso básico do

`ListTile`. Porém, como se trata de um widget que pode ter especializações para o referido uso, vamos criar um widget específico para nossa visão.

Na pasta `/lib/routes/palavras`, crie outra, chamada `widgets` e, nela, um arquivo chamado `palavras_listtile_widget.dart`. Vamos implementar nessa classe o conteúdo a seguir.

```
import 'package:flutter/material.dart';

class PalavrasListTileWidget extends StatelessWidget {
  final String title;
  final Widget trailing;

  const PalavrasListTileWidget({this.title, this.trailing});

  @override
  Widget build(BuildContext context) {
    return Container(
      child: ListTile(
        contentPadding: EdgeInsets.only(left: 5, bottom: 5, top: 3),
        title: Text(
          title,
        ),
        trailing: trailing,
      ),
    );
  }
}
```

Você certamente verificou que nosso widget é bem simples. Para o momento, nosso `ListTile` sequer faz uso do `leading` e do `subtitle`. Poderíamos usar o `leading` se nossas `palavras` possuíssem uma foto, por exemplo, e poderíamos até pensar em ter no `subtitle` a `ajuda` para nossa palavra. Mas essas evoluções deixo a seu critério.

Recuperação do conteúdo com o BLoC

Agora que temos tudo pronto para que possamos gerar nossa visão com a exibição das palavras registradas, vamos à implementação

desse processo. Vamos substituir o `body: Container()` de nosso `build()` pelo código a seguir, que explicarei após sua exibição. Atente-se aos imports e lembre-se de declarar `PalavrasListViewBloc _palavrasListViewBloc;` no início de `_PalavrasListViewRouteState`.

```
body: BlocBuilder<PalavrasListViewBloc, PalavrasListViewBlocState>
(builder: (context, state) {
    if (state is PalavrasListViewBlocError) {
      return centerText(text: 'Falha ao recuperar palavras:
${state.errorMessage}');
    }

    if (state is PalavrasListViewLoaded) {
      if (state.palavras.isEmpty) {
        return centerText(text: 'Nenhuma palavra registrada ainda.');
      }

      return ListView.builder(
        padding: EdgeInsets.only(top: 10),
        itemCount: state.palavras.length,
        itemBuilder: (BuildContext context, int index) {
          return PalavrasListTileWidget(
            title: state.palavras[index].palavra,
            trailing: Icon(Icons.keyboard_arrow_right),
          );
        },
      );
    }

    if (state is PalavrasListViewBlocUninitialized)
      _palavrasListViewBloc.add(PalavrasListViewBlocEventFetch());

    return Center(
      child: CircularProgressIndicator(),
    );
}),
```

O `body` agora será o corpo de `builder` de nosso `BlocBuilder`. Já no primeiro `if()`, verificamos se o estado recebido se trata de um erro

e, em caso positivo, retornamos o widget que implementamos em nosso `mixin` anteriormente.

No segundo `if()`, caso as palavras tenham sido carregadas corretamente, verificamos, em um `if()` interno, se alguma palavra foi recuperada. Caso o `List palavras` esteja vazio, também retornamos uma mensagem para informar ao usuário. Caso a recuperação tenha trazido resultados, ou seja, palavras, retornaremos um `ListView`, onde consumiremos o widget para o `ListTile` implementado anteriormente.

Na última condição, temos a verificação de carga ainda não inicializada, já pensando no usuário ao entrar e sair dessa rota. Mais à frente comentamos um pouco mais isso, mas é importante saber que caso o estado do BLoC seja este, o fetch é solicitado da mesma maneira que fizemos no registro do BLoC no `main()`. Caso nenhuma condição anterior tenha sido satisfeita, significa que nossa aplicação está processando a recuperação dos dados, então retornamos um `CircularProgressIndicator`.

Só nos falta registrar nosso BLoC para que tudo possa funcionar. Estamos concentrando isso em nosso `main.dart`, lembra? Muito bem, vamos então inserir o código a seguir na sequência dos BLoCs que já temos lá. Observe que estamos utilizando o operador `..`, que nos possibilita o encadeamento do método com uma instância, neste caso, de `PalavrasListViewBloc`. Isso causará a primeira carga de dados para serem consumidos quando acessarmos a visão de listagem. Novamente, preste atenção aos imports.

```
BlocProvider<PalavrasListViewBloc>(
  create: (BuildContext context) =>
      PalavrasListViewBloc(palavraDAO: PalavraDAO())
        ..add(PalavrasListViewBlocEventFetch()),
),
```

Em relação ao `orderBy`, que implementamos no `getAll()` e que eu disse que comentaria mais à frente, chegou o momento. Lá na propriedade, temos o acréscimo de `COLLATE LOCALIZED` para que isso

possa ser levado em consideração nas consultas realizadas no banco. Neste ponto, se tivermos palavras acentuadas, a paginação pode não ser bem-sucedida. Para testar, recomendo que você insira letras simples como palavras, na seguinte ordem: `a, A, à, À, á e Á`. Já retomaremos isso.

Vamos testar nossa aplicação. Se tudo der certo, como já temos palavras registradas devido aos testes que realizamos, já teremos dados sendo exibidos em nossa visão. Eu ainda não tinha mais de 20 palavras registradas, então, para simularmos isso, inseri o seguinte código logo no início do método `build()` da classe `ForcaApp`, em `main.dart`. Realize os imports. Lembre-se de realizar um Hot Reload para que a aplicação reinicie e registre o BloC.

```
PalavraDAO palavraDAO = PalavraDAO();
for (int i = 0; i < 30; i++) {
  var random = Random();
  var palavra = random.nextInt(1000).toString();
  palavraDAO.insert(
      palavraModel: PalavraModel(
          palavra: 'Palavra $palavra',
          ajuda: 'Ajuda para palavra $palavra'));
}
```

Execute sua aplicação e as palavras serão inseridas. Depois, comente o código anterior, execute novamente sua aplicação e vá até a rota de palavras registradas. Veja que apenas 20 palavras serão exibidas, pois nosso `Fetch` está ocorrendo apenas no início da aplicação. Não estamos ainda capturando o momento de rolagem, em que o usuário tentará ver mais palavras do que as que temos recuperadas.

Volte ao `getAll()` e retire o `COLLATE LOCALIZED` do `orderBy`. Viu que não apareceram no início todas as letras que inserimos antes? Este é um problema que precisa ser pensado e contornado. É uma característica de localização no SQLite, mas resolver isso aqui demandaria mais conteúdo, fora do escopo, que é a rolagem infinita com BLoC. Você ainda poderá verificar que as letras minúsculas,

dependendo da inserção, não aparecem todas de uma vez, sendo intercaladas por maiúsculas. Tentei utilizar o `LOCALIZED` em conjunto com `NOCASE`, mas sem sucesso. Uma possibilidade para ajustar isso seria adaptar a função de retirada de acentos, ou criar outra para classificar primeiro as minúsculas e depois as maiúsculas.

Uma atenção especial aqui. Se você acessar nossa rota de inserção de palavras e for para a listagem, essas palavras não aparecerão. Caso você queira que isso aconteça, neste momento deixarei você refletir o que é e como corrigir, mas na seção *A alteração de uma palavra já registrada*, no próximo capítulo, trabalharemos isso.

8.7 A rolagem infinita dos dados

Como explicado anteriormente, nossa recuperação será sempre disparada quando houver uma rolagem no `ListView` para além dos lados recuperados. Para isso, precisamos realizar uma preparação em nossa rota. Vamos começar com a declaração de algumas propriedades, tal qual segue o código, no `palavras_listview_route.dart`.

```
final _scrollController = ScrollController();
final _scrollThreshold = 200.0;
PalavrasListViewBloc _palavrasListViewBloc;
```

A primeira propriedade nos dará um controlador para o processo de rolagem do `ListView`, pois precisamos inferir quando isso ocorrer. A segunda define um valor que será utilizado para que, quando chegarmos ao final da listagem e uma nova busca por `palavras` ocorrer, tenhamos um espaço entre o `ListView` e o final da tela do dispositivo, onde mostraremos ao usuário que a aplicação está realizando uma nova carga de dados. A terceira propriedade é um auxiliador para utilizarmos nosso BLoC. Ela não é final por não termos ainda o `context` para inicializá-la, e faremos isso no `initState()`.

Antes de implementarmos o `initState()`, precisamos do método que será disparado quando ocorrer a rolagem no `ListView`. Veja este método na sequência, que deve ser inserido em nosso `PalavrasListViewMixim`.

```
onScroll(
  {PalavrasListViewBloc palavrasListViewBloc,
  ScrollController scrollController,
  double scrollThreshold}) {
    final maxScroll = scrollController.position.maxScrollExtent;
    final currentScroll = scrollController.position.pixels;
    if (maxScroll - currentScroll <= scrollThreshold) {
      palavrasListViewBloc.add(PalavrasListViewBlocEventFetch());
    }
}
```

O método anterior recebe os argumentos necessários para sua execução e obtém valores relacionados ao `ScrollController` para que possa verificar se há necessidade de realizar uma nova recuperação de dados. Com a implementação anterior realizada, vamos então implementar nosso `initState()`, como no código apresentado a seguir.

```
  @override
void initState() {
    super.initState();
    _palavrasListViewBloc = BlocProvider.of<PalavrasListViewBloc>
(context);
    _scrollController.addListener(
      () => onScroll(
          palavrasListViewBloc: _palavrasListViewBloc,
          scrollController: _scrollController,
          scrollThreshold: _scrollThreshold),
    );
}
```

Como estamos usando objetos que consomem recursos de nosso dispositivo durante a execução dessa rota, precisamos finalizá-los quando ela for encerrada. Para isso, sobrescrevemos o método `dispose()`, tal qual apresentado na sequência. Não feche o BLoC,

pois com isso não poderíamos mais disparar eventos. É importante saber que precisamos preparar o BLoC para reiniciar quando sair dessa rota, pois, caso contrário, seu estado continuará a ser o mesmo da última interação do usuário. Para isso, invocamos o evento para reiniciar a carga também no `dispose()`. Lembra que nosso primeiro `Fetch` foi dado no `main()`, no registro do BLoC? Isso resolve o problema comentado anteriormente, sobre sair da listagem, inserir e retomar para ela.

```
@override
void dispose() {
    _scrollController.dispose();
    _palavrasListViewBloc.add(PalavrasListViewBlocEventResetFetch());
    super.dispose();
}
```

Precisamos também, em nosso `ListView.builder()`, atribuir nosso `ScrollController` a ele e o fazemos com o código a seguir. É uma propriedade para o `ListView`.

```
controller: _scrollController,
```

Muito bem, execute novamente sua aplicação. Role os itens para cima, note que agora todos os registrados, que são mais de 20, são exibidos. Mas como podemos dar ao usuário uma resposta para que, caso a recuperação de novos dados leve um tempo, ele não pense que a aplicação travou?

Vamos recorrer a uma simples artimanha. Veja no código a seguir uma nova declaração para `itemCount` de nosso `ListView.builder()`. Observe a condição e os valores para o operador ternário. Caso não tenhamos chegado ao final da listagem, ou seja, caso haja ainda dados a serem recuperados, aumentamos em 1 a quantidade de itens.

```
itemCount: state.hasReachedMax
    ? state.palavras.length
    : state.palavras.length + 1,
```

Com essa nova situação, vamos agora criar um widget que dará ao usuário a informação de que uma nova requisição de dados está sendo realizada. Na pasta `widgets` de `palavras`, insira um arquivo chamado `bottom_loader_widget`, com o código apresentado a seguir.

```
import 'package:flutter/material.dart';

class BottomLoaderWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.indigo,
      child: Padding(
        padding: const EdgeInsets.symmetric(vertical: 10),
        child: Center(
          child: Text(
            'Carregando mais registros',
            textAlign: TextAlign.center,
            style: TextStyle(
              color: Colors.white,
              fontSize: 25,
            ),
          ),
        ),
      ),
    );
  }
}
```

Para que nosso novo widget seja consumido, precisamos agora realizar uma adaptação em nosso `itemBuilder` do `ListView.builder()`, como podemos ver na sequência. Veja que fazemos uma verificação em cima do `index` e, caso tenhamos atingido o mínimo da quantidade de palavras, exibimos nosso novo widget, caso contrário, exibimos o elemento recuperado. Lembre-se de que, com nossa implementação anterior para `itemCount`, nosso `index` pode ir até um elemento a mais que a quantidade de palavras. Não se esqueça do import.

```
itemBuilder: (BuildContext context, int index) {
  return (index >= state.palavras.length)
      ? BottomLoaderWidget()
      : PalavrasListTileWidget(
          title: state.palavras[index].palavra,
          trailing: Icon(Icons.keyboard_arrow_right),
        );
},
```

Vamos testar? Execute sua aplicação, realize a rolagem e veja se em seu dispositivo a mensagem destacada na figura a seguir é exibida.

Figura 8.2: Mensagem de carregamento de registros

Conclusão

Terminamos a primeira parte relacionada ao objetivo apresentado no início do capítulo. Sugiro que dê uma respirada, tome uma água e após o relaxamento, retorne para o próximo capítulo, onde finalizaremos o processo de persistência e animação.

CAPÍTULO 9
Remoção de dados e atualização do ListView com destaque para alterações realizadas

No capítulo anterior, começamos nossas implementações para a persistência de dados e animações na transição de rotas. Precisamos agora concluir essa atividade.

9.1 A remoção da palavra do ListView

Já temos a inserção e recuperação de todos os dados de nossa tabela de palavras. Falta-nos a alteração, que precisa antes de uma recuperação de dados em específico e da remoção de uma palavra da base de dados.

Precisamos começar a resolução dessas pendências com o código que nos subsidiará a interface com o usuário e execução correta da atividade. Nós faremos uso de um widget novo, o `Dismissible`. Esse widget possibilita que um item de um `ListView`, ao ser deslizado, seja retirado da listagem. A princípio, essa operação pode ser realizada em qualquer direção, mas é possível controlar isso.

O `Dismissible` nos permite capturar o momento em que o usuário realiza o deslize, onde podemos inferir e perguntar ao usuário se ele confirma ou não a remoção. Depois, caso a operação seja confirmada, temos condições de realizar também alguma funcionalidade para isso.

O que faremos é executar um `Dialog`, que exibirá uma pergunta e solicitará a resposta ao usuário por meio de botões. Para minimizarmos o código aqui, vamos utilizar um componente que disponibilizei no `http://pub.dev`, chamado `dialog_information_to_specific_platform`, e que vamos declarar em

nosso `pubspec.yaml`, como podemos ver na sequência. Este pacote renderiza um `Dialog` seguindo os padrões de cada plataforma, conforme o que vimos anteriormente sobre componentes renderizados de acordo com a plataforma em execução.

```
dialog_information_to_specific_platform: ^0.0.4
```

Veja na listagem a seguir a implementação do método que será executado para confirmar a remoção da palavra da listagem. Ele deve ser implementado em nosso arquivo de `Mixin`, o `palavras_listview_mixin.dart`. Antes do código, temos as importações necessárias para o pacote que estamos utilizando.

```
import
'package:dialog_information_to_specific_platform/dialog_information_to_spe
cific_platform.dart';
import
'package:dialog_information_to_specific_platform/flat_buttons/actions_flat
button_to_alert_dialog.dart';

Future<String> confirmDismiss({BuildContext context, String palavra,
String palavraID}) async {
    return await showDialog(
      barrierDismissible: false,
      context: context,
      child: InformationAlertDialog(
        iconTitle: Icon(
          Icons.message,
          color: Colors.red,
        ),
        title: 'Oops...Quer remover?',
        message: 'Confirma a remoção da palavra ${palavra.toUpperCase()}',
        buttons: [
          ActionsFlatButtonToAlertDialog(
            messageButton: 'Não',
            isEnabled: true,
          ),
          //   InformationAlertDialog.createFlatButton(),
          ActionsFlatButtonToAlertDialog(
            messageButton: 'Sim',
```

```
            isEnabled: true,
          ),
          //   InformationAlertDialog.createFlatButton(),
        ],
      ),
    );
}
```

O método anterior recebe o contexto e a palavra que se deseja remover. Como retorno, temos o `showDialog()`, tendo como filho o componente importado `InformationAlertDialog`. As propriedades dele são autoexplicativas, tendo em `buttons` um list de widget, que aqui é outro widget do pacote, o `ActionsFlatButtonToAlertDialog`, também com propriedades autoexplicativas.

Como é possível verificar, o método seguindo o padrão de `showDialog()` retornará o texto do botão pressionado para quem o invocou e com isso conseguiremos tomar uma decisão sobre o que fazer.

Agora, precisamos implementar o método que realizará o processo de remoção do item da lista, e podemos vê-lo na sequência em nosso mesmo arquivo de `Mixin`. Observe que, até o momento, estamos apenas exibindo uma mensagem, via `SnackBar`, de que o item foi removido. Podemos trabalhar com qualquer widget em `content`, mas, como o foco não é design, estou trazendo apenas um `Text` informativo.

```
Future<void> dismissedComplete(
  {BuildContext context, String palavraID, String palavra}) async {
    Scaffold.of(context).showSnackBar(SnackBar(
      backgroundColor: Colors.indigo,
      content: Text(
        'Palavra ${palavra.toUpperCase()} foi removida',
      ),
    ));
}
```

Na sequência, vamos implementar o `Dismissible` e faremos isso em nosso `itemBuilder` do `ListView`. Já temos o método implementado, mas agora o adaptaremos. Veja o código a seguir. Este código está no `palavras_listview_route.dart`, em nosso `ListView`.

```
itemBuilder: (BuildContext context, int index) {
  return (index >= state.palavras.length)
      ? BottomLoaderWidget()
      : Dismissible(
          key: Key(state.palavras[index].palavraID),
          confirmDismiss: (direction) async {
            var oQueFazer = await confirmDismiss(
                context: context,
                palavra: state.palavras[index].palavra);
            return oQueFazer == 'Sim';
          },
          onDismissed: (direction) async {
            await dismissedComplete(
                context: context,
                palavraID: state.palavras[index].palavraID,
                palavra: state.palavras[index].palavra);
            return;
          },
          background: Container(
            color: Colors.red,
          ),
        // Aqui é o que temos já no método
          child: PalavrasListTileWidget(...),
        );
},
```

Verifique que temos inicialmente a definição de um valor para `key`, pois cada `Dismissible` precisa de um `key` diferente. Depois, temos a definição das ações para `confirmDismiss` e `onDismissed`, que são as propriedades respectivas aos métodos que implementamos anteriormente. Quase terminando, temos a `background`, que pode ter qualquer widget, mas, por simplicidade, utilizei apenas um `Container`.

Vamos ao teste? Execute sua aplicação e, na listagem de palavras, deslize uma delas para qualquer lado. Veja que um componente vermelho será exibido em seu lugar e, quando terminar o deslize, uma dialog com solicitação de confirmação aparecerá. Ao confirmar a remoção, você verá que a palavra foi removida e uma mensagem será exibida na base da tela confirmando a operação. Veja essas situações na figura a seguir.

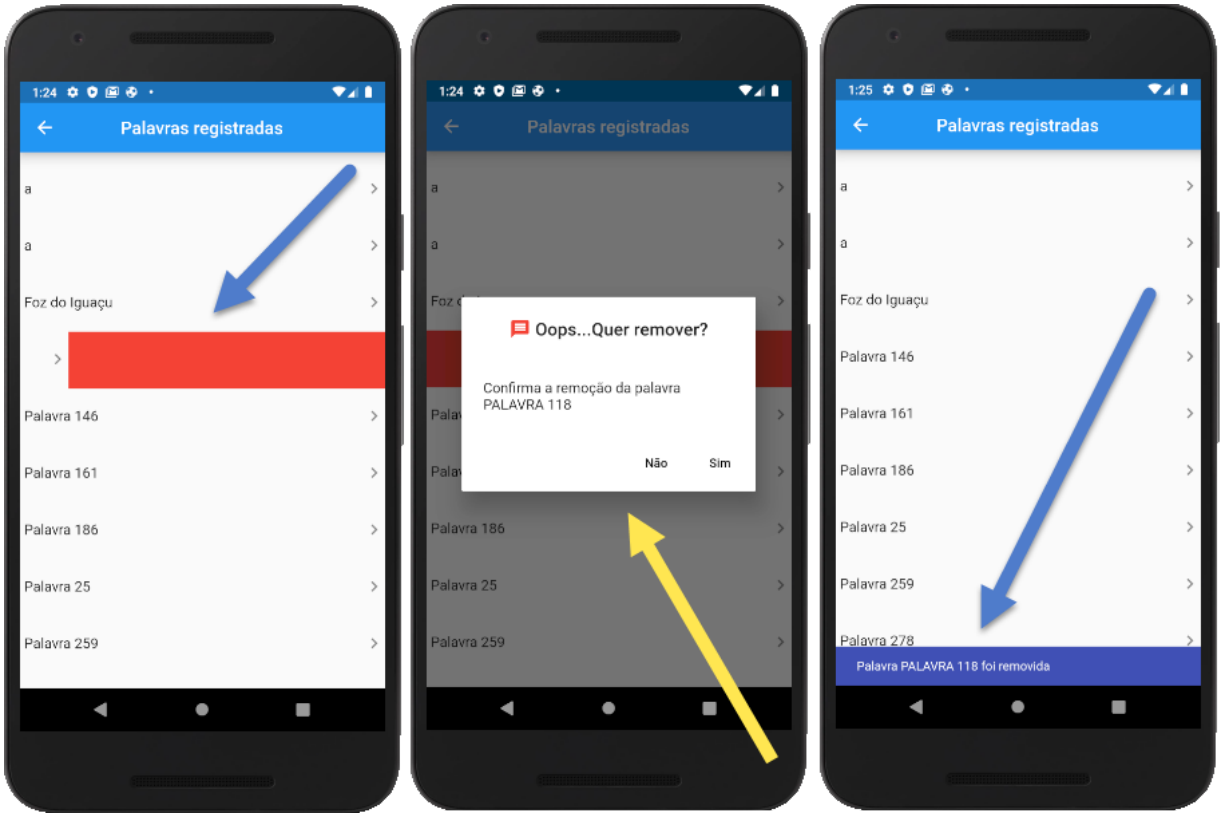

Figura 9.1: Retirada de uma palavra da listagem

9.2 A remoção da palavra da tabela de dados

Se você realizou o teste anterior, viu que a retirada da palavra da lista ocorre perfeitamente, mas, se voltarmos a exibir a listagem, a palavra estará lá, pois não a removemos da nossa base de dados.

Precisamos fazer isso com a implementação de um método específico em nosso `palavra_dao`. Veja-o na sequência.

```
Future<int> deleteByID(String palavraID) async {
    try {
      Database lpDatabase = await SQFLiteDataBase.instance.database;

      var result = await lpDatabase.delete(kPalavrasTableName,
          where: '$kPalavraPalavraID = ?', whereArgs: [palavraID]);

      return result;
    } catch (exception) {
      rethrow;
    }
}
```

Observe que o processo de obtenção da base de dados é semelhante aos métodos já implementados, alteramos apenas o método a ser executado. Agora, precisamos invocar esse método no momento em que a remoção for confirmada. Para isso, adaptaremos nosso método `confirmDismiss()` em nosso `Mixin`. Inicialmente veja a nova invocação ao `showDialg()` na sequência e observe que substituímos o `return` por uma atribuição do resultado a `oQueFazer`. Mantenha o código representado por `...`.

```
String oQueFazer = await showDialog(...);
```

Após a resposta do usuário ao `Dialog` exibido, precisamos verificar se a palavra será ou não removida. Dessa maneira, veja no trecho a seguir, logo após o fechamento da invocação anterior, a confirmação ou não do desejo de remoção. Precisaremos trocar o genérico `String` na assinatura do método para `bool`.

```
if (oQueFazer == 'Não') return false;
```

Neste momento, teremos uma situação em que precisaremos exibir um `SnackBar` para o sucesso da remoção e um para o caso de algum erro ocorrer na invocação de nosso método `deleteByID()`. Sendo

assim, vamos criar um método específico para exibição de nosso `SnackBar` em nosso `Mixin`. Veja-o na sequência.

```
Future showSnackBarMessage(
  {BuildContext context, String message, Color backgroundColor}) async {
    Scaffold.of(context)
        .showSnackBar(SnackBar(
          backgroundColor: backgroundColor,
          content: Text(
            message,
          ),
        ))
        .closed
        .then((_) {
      return;
    });
}
```

Para a remoção, vamos criar um método privado em nosso `Mixin`, que efetivamente realizará a remoção. Veja-o na sequência e note que invocamos nosso método de DAO e, caso haja sucesso, retomamos `true`, caso contrário, exibimos um `SnackBar` alertando o erro. Veja a invocação ao método anterior. Será preciso um import.

```
Future<bool> _removePalavra(
    String palavraID, BuildContext context, String palavra) async {
  try {
    PalavraDAO palavraDAO = PalavraDAO();
    await palavraDAO.deleteByID(palavraID);
    return true;
  } catch (exception) {
    showSnackBarMessage(
        context: context,
        message:
            'Erro ao remover a Palavra ${palavra.toUpperCase()}:
$exception',
        backgroundColor: Colors.red);
    return false;
  }
}
```

Novamente, com as implementações necessárias prontas, precisamos adaptar nosso método `confirmDismiss()` para que, caso o usuário realmente confirme a remoção, ela possa ser disparada. Ao final do método `confirmDismiss()`, após o `return` que implementamos há pouco, insira o código a seguir.

```
return await _removePalavra(palavraID, context, palavra);
```

Precisamos adaptar a definição do `confirmDismiss` em nosso `Dismissible`. Veja-a a seguir. Note que já retornamos diretamente o próprio resultado do método `confirmDismiss()`. Isso em nosso `ListView`.

```
return await confirmDismiss(
    context: context,
    palavra: state.palavras[index].palavra,
    palavraID: state.palavras[index].palavraID);
```

Você pode executar sua aplicação agora, acessar a listagem, remover uma palavra, retornar para o `Drawer`, acessar novamente e comprovar que a palavra foi removida não só da listagem, mas também da tabela. Mas falta algo que veremos na próxima seção.

9.3 A remoção da palavra da coleção que popula o ListView

No exemplo que fizemos funcionar anteriormente, a palavra era removida do componente de visualização, o `ListView`, mas ela continuava em nossa coleção, que é retornada pelo nosso BLoC, mesmo tendo sido removida da tabela.

Nós poderíamos - e seria recomendado - definir e tratar toda essa lógica de remoção em nosso BLoC, mas, para que pudéssemos ver algumas técnicas e recursos diferentes, optei por continuar com a estratégia que estamos seguindo. Então, utilizaremos BLoC para

remover nossa palavra da coleção. Nosso primeiro passo é criarmos um novo evento em `palavras_listview_event.dart`, tal qual apresento na sequência.

```
class PalavrasListViewBlocEventConfirmDismiss extends
PalavrasListViewBlocEvent {
  final int indexOfDismissible;

  PalavrasListViewBlocEventConfirmDismiss({
    this.indexOfDismissible,
  });
}
```

Precisamos preparar nossa lógica de BLoC para manipular este evento. O código a seguir pode ser inserido logo após o primeiro `if()` de `mapEventToState()` de nosso arquivo `palavras_listview_business_bloc.dart`. No código, verificamos o evento recebido e o estado atual do BLoC. Caso a condição seja válida, removemos da coleção o item no índice recebido no evento e retomamos nosso estado `PalavrasListViewLoaded` normalmente.

```
if (event is PalavrasListViewBlocEventConfirmDismiss &&
    currentState is PalavrasListViewLoaded) {
  currentState.palavras.removeAt(event.indexOfDismissible);
  yield PalavrasListViewLoaded(
      palavras: currentState.palavras,
      hasReachedMax: currentState.hasReachedMax);
  return;
}
```

Precisamos agora invocar o evento que implementamos e faremos isso na definição `onDismissed` de nosso `Dismissible`. Para facilitar, trago toda esta definição na sequência.

```
onDismissed: (direction) async {
    await dismissedComplete(
        context: context,
        palavraID: state.palavras[index].palavraID,
        palavra: state.palavras[index].palavra);
```

```
    _palavrasListViewBloc.add(
      PalavrasListViewBlocEventConfirmDismiss(
          indexOfDismissible: index),
);
```

Agora podemos testar nossa aplicação, a remoção está completa. As possíveis imagens já foram apresentadas anteriormente.

9.4 A alteração de uma palavra já registrada

Precisamos implementar o que seria a última funcionalidade de nosso CRUD, a alteração nos dados de uma palavra já registrada. Além dessa, há ainda a recuperação de um registro com base em seu ID, mas, devido à técnica que estamos seguindo, precisaremos implementar apenas a alteração.

Para alterarmos uma palavra, usaremos a rota já existente que criamos para a inserção. Os componentes de interação são os mesmos, mas agora trabalharemos apenas com os dados já existentes, que deverão ser exibidos quando acessarmos a rota.

Poderíamos aqui seguir o uso de BLoC e ter no BLoC de CRUD a definição de uma propriedade que mantivesse o objeto de referente a uma palavra, que seria a palavra selecionada na listagem, mas vamos trabalhar com a injeção por meio do construtor, o que nos possibilitará também trabalhar com argumentos em nosso helper de rotas.

Nosso primeiro passo será definir um parâmetro opcional em `PalavrasCRUDRoute`, que receberá a palavra a ser alterada. Veja essa modificação no código a seguir.

```
class PalavrasCRUDRoute extends StatefulWidget {
  final PalavraModel palavraModel;

  const PalavrasCRUDRoute({this.palavraModel});
```

```
  @override
  _PalavrasCRUDRouteState createState() => _PalavrasCRUDRouteState();
}
```

Precisamos agora pensar em como, em nosso `ListView`, capturar um gesto de pressão prolongada em um item, conhecido como `longPress`. Já conhecemos o `GestureDetector`, que desempenha bem esta funcionalidade, mas vamos utilizar o `InkWell`, que nos dá um efeito visual legal na interação com o usuário.

Mudaremos o `child` de `Dismissible`, que é `PalavrasListTileWidget`, para `InkWell`. E, como `child` de `InkWell`, teremos `PalavrasListTileWidget`. A implementação a seguir traz esta alteração, já com o código que teremos para exibir nossa rota de alteração de dados da palavra selecionada. Fique atento ao import.

```
child: InkWell(
    onLongPress: () {
      Navigator.of(context).pushNamed(kPalavrasCRUDRoute,
          arguments: state.palavras[index]);
    },
    child: PalavrasListTileWidget(
      title: state.palavras[index].palavra,
      trailing: Icon(Icons.keyboard_arrow_right),
    ),
),
```

Observou que, no `pushNamed`, temos agora `arguments`, que receberá o objeto atual na criação do widget? Agora, precisamos trabalhar em nosso `app_router.dart` a verificação da chegada de argumentos para poder remetê-lo à nossa rota. Veja na sequência.

```
case kPalavrasCRUDRoute:
    return MaterialPageRoute(
      builder: (_) => PalavrasCRUDRoute(
        palavraModel:
            settings.arguments != null ? settings.arguments : null,
      ),
    );
```

Basta trabalharmos em nossa rota para utilizar com o argumento recebido quando formos realizar uma alteração.

Em nosso exemplo, temos apenas dois controles para receber dados do usuário, mas precisamos pensar que poderíamos ter vários e que a atualização deles poderia ocorrer em mais de uma situação. Desta maneira, criaremos em nossa `PalavrasCRUDRoute` o seguinte método privado.

```
_initializeTextControllers() {
    this._palavraController.text = widget.palavraModel.palavra;
    this._ajudaController.text = widget.palavraModel.ajuda;
}
```

Agora, em nosso `initState()`, invocamos este método, caso tenha sido recebido um objeto para alteração. Veja o código que deve ser inserido no `initState()`, ao final, na sequência.

```
if (widget.palavraModel != null) {
  _initializeTextControllers();
}
```

Com essa implementação já é possível testarmos a funcionalidade. Acesse o `ListView` e pressione de maneira prolongada um item qualquer. Veja que você será redirecionado à visão que temos para inserção, mas com os dados já exibidos.

Podemos também alterar o `title` do `AppBar` para mostrar que operação está sendo realizada. Veja o código na sequência.

```
title: Text(
  widget.palavraModel == null
      ? 'Registro de Palavras'
      : 'Alteração de uma palavra',
),
```

9.5 Atualização da listagem com palavras inseridas e alteradas pela rota de CRUD

Algumas páginas atrás resolvemos um problema relacionado à invocação do `getAll()` quando acessamos nossa listagem. Nós fizemos isso trabalhando no BLoC, em nosso `ListView` e no `dispose`. Isso foi necessário porque, como dito, ao registrarmos o BLoC, já executamos o método `getAll()` e depois não tínhamos nada para que este método fosse executado novamente, até que ajustamos isso.

Como comentei na seção *Recuperação do conteúdo com o BLoC*, se utilizarmos nossa rota de CRUD para inserir ou alterar uma palavra, esta inserção e alteração não serão refletidas em nossa listagem. Vamos resolver este problema agora, para vermos outro que surgiu com a alteração.

Vamos retirar o `..add()` do `main` e inseri-lo no `initState()` de nossa rota de listagem. Veja a nova instrução na sequência, que deve estar logo após a chamada ao `super.initState()`. Essa execução no `main()` não é tão interessante, pois estamos executando um serviço que só será necessário em uma rota que o usuário pode inclusive nem acessar.

```
_palavrasListViewBloc = BlocProvider.of<PalavrasListViewBloc>
(context)..add(PalavrasListViewBlocEventFetch());
```

A inserção já estava funcionando, pois tínhamos ajustado, mas ainda temos o problema da alteração. Se alterarmos uma palavra e a gravarmos, ela será duplicada em nossa base. Isso é normal, pois estamos invocando o método `insert()` de nosso `DAO`. Precisamos corrigir isso e chamar o `update()` e, caso a palavra ainda esteja sem `ID`, o método deve delegar para o `insert()`. Vamos lá, veja o método `update()` a ser inserido no `palavra_dao.dart` na sequência.

```
Future<String> update({@required PalavraModel palavraModel}) async {
    String result;
```

```
    try {
      if (palavraModel.palavraID == null) {
        String result = await insert(palavraModel: palavraModel);
        return result;
      }

      Database lpDatabase = await SQFLiteDataBase.instance.database;

      var recordsAffected = await lpDatabase.update(
          kPalavrasTableName, palavraModel.toJson(),
          where: "$kPalavraPalavraID = ?", whereArgs:
[palavraModel.palavraID]);
      if (recordsAffected == 0)
        result = null;
      else
        result = recordsAffected.toString();
    } catch (exception) {
      rethrow;
    }
    return result;
}
```

Veja que recebemos o objeto que deve ser atualizado, verificamos a existência ou não do valor de `ID` e, caso não exista, delegamos para o método que já temos, como dito antes. Caso seja um objeto já existente, preparamos a atualização. Observe o `where`, que faz a seleção do objeto que deverá ser atualizado na base.

Agora precisamos adaptar o momento em que o usuário quer gravar a palavra, seja para inserção ou alteração. Para facilitar, trouxe novamente todo o método, mas me aterei às mudanças realizadas. Veja o código na sequência. Observe que agora estamos atribuindo um valor para `palavraID`, pois, caso estejamos alterando, esta propriedade já tem valor. Depois, trocamos a invocação ao `insert()` para o `update()`.

```
void _onSubmitPressed() async {
    PalavraDAO palavraDAO = PalavraDAO();
    PalavraModel palavraModel = PalavraModel(
        palavraID: (widget.palavraModel == null)
```

```
            ? null
            : widget.palavraModel.palavraID,
        palavra: this._palavraController.text,
        ajuda: this._ajudaController.text);

    try {
      await palavraDAO.update(palavraModel: palavraModel);
      _palavrasCrudFormBloc.add(FormSuccessSubmitted());
    } catch (e) {
      rethrow;
    }
}
```

Podemos executar nossa aplicação alterar um dado e retomar para a listagem. Nova surpresa: a alteração ainda não aparece. É preciso sair da listagem e retomar a ela, mas não é isso que queremos. Vamos adaptar isso. Na definição do `onLongPress` de `InkWell`, adapte a implementação para o código a seguir. Veja que agora ele é `async`.

```
onLongPress: () async {
  await Navigator.of(context).pushNamed(
      kPalavrasCRUDRoute,
      arguments: state.palavras[index]);

      Timer(Duration(seconds: 1), () {
        _palavrasListViewBloc
            .add(PalavrasListViewBlocEventResetFetch());
      });
},
```

Observou que, após o retomo da navegação, temos uma nova transição de estado, agora para o de carga de dados? Viu o `await` no `Navigator`? Pois é. Precisamos aguardar o retomo para que os dados da listagem possam novamente ser recuperados. Ainda sobre a nova transição de estados, ela está em um `Timer()`. Isso é um workaround, pois sem este artifício o Flutter não estava realizando o `refresh` no `ListView`.

Teste agora sua aplicação e veja que a alteração realizada, ao retomar para a listagem, já está refletida.

Mas temos ainda uma situação que seria interessante trabalharmos. Se alterarmos uma palavra que está mais abaixo da listagem exibida no dispositivo, ao retomarmos, não a vemos, sendo necessário realizar o *scroll* em busca da palavra. Eu julgo que o correto é que, quando a palavra tiver sido alterada, ao retomar para a listagem, esta palavra esteja em foco para ficar fácil a visualização da alteração. Vamos trabalhar nisso.

9.6 Destaque no ListView para a palavra alterada

Muito bem, vamos realizar estas alterações aqui, com vistas já ao final do capítulo. Inicialmente, precisaremos de duas variáveis auxiliares, e as declararemos ao final das já declaradas. Veja o código na sequência. A primeira será um flag para manter o ID da palavra que será alterada e a segunda será utilizada como flag para definir uma cor diferente para a palavra alterada, para que apareça em destaque na listagem. Tudo isso no `palavras_listview_route`.

```
String _palavraIDSelected;
String _palavraIDOfTileToHighlight;
```

Após a implementação anterior, no código do `onLongPress`, insira, logo no início, a instrução a seguir. Isso garantirá que teremos sempre o ID da palavra que será alterada.

```
_palavraIDSelected = state.palavras[index].palavraID;
```

Precisamos processar a identificação da palavra que foi alterada nas palavras que são exibidas na listagem e a movimentar para o topo. O processo de movimentação será forçar uma rolagem e precisamos saber a altura que teremos que rolar. Para isso, criamos

uma nova variável, que deve ser implementada em conjunto com as anteriores, da maneira exibida a seguir.

```
final double _listTileHeight = 70;
```

Em nosso `PalavrasListTileWidget`, precisamos definir uma propriedade que receberá este valor e o utilizará na definição de seu contêiner. Veja o novo código na sequência.

```
// Propriedade
final double listTileHeight;

// No construtor
this.listTileHeight,
```

Ao invocarmos nosso widget `PalavrasListTileWidget`, precisamos encaminhar o valor para a nova propriedade, e o código a seguir demonstra isso. Insira-o ao final de `PalavrasListTileWidget()`, em nosso `ListView`, no `palavras_listview_route`.

```
listTileHeight: _listTileHeight,
```

Agora sim, já podemos implementar o código que será responsável pela rolagem das palavras para que a alterada seja exibida. Veja-o na sequência, que deve estar antes do `ListView.builder()`.

```
Future.delayed(Duration(milliseconds: 500)).then((onValue) {
    if (this._scrollController.hasClients) {
      for (int i = 0; i < state.palavras.length; i++) {
        if (state.palavras[i].palavraID == this._palavraIDSelected) {
          _scrollController.animateTo(i * _listTileHeight,
              duration: new Duration(seconds: 2), curve: Curves.ease);
          setState(() {
            _palavraIDOfTileToHighlight = this._palavraIDSelected;
          });
          this._palavraIDSelected = null;
        }
      }
      if (this._palavraIDSelected != null)
        _palavrasListViewBloc.add(PalavrasListViewBlocEventFetch());
```

```
    }
});
```

O código será executado apenas após ter passado meio segundo. Nele, há uma condição que verifica se o `ListView` já é visto como cliente do `ScrollController`. Caso isso seja verdadeiro, um laço que percorrerá todas as palavras disponíveis no estado atual será executado. Palavra por palavra é verificada, até que seja encontrada a que possui o ID da que foi alterada. Caso haja uma avaliação correta, o scroll é realizado pelo App e atualizamos a variável que funcionará como flag para mudarmos a cor do `TileList` alterado.

Caso a palavra não tenha sido encontrada na coleção dos dados exibidos, uma nova recuperação será realizada, o que leva a uma atualização do `ListView` e, consequentemente, a uma nova busca pelo item alterado.

Para finalizar, precisamos informar na criação do `ListTile` que ele pode ter uma cor diferente, com base em seu estado. Para isso, em `PalavrasListTileWidget`, insira uma nova propriedade, tal qual o código a seguir.

```
// Propriedade
final Color color;

// Construtor
this.color,
```

Só nos resta configurar a cor em nosso `Container`, do `PalavrasListTileWidget`, que pode ser realizada com o código a seguir.

```
color: this.color,
```

Para finalizar esta etapa, precisamos adaptar a invocação de nosso widget para que ele saiba que precisa ajustar a cor do item selecionado. Veja essa alteração na sequência, que deve ser inserida ao final da invocação de `PalavrasListTileWidget`.

```
color: (_palavraIDOfTileToHighlight == state.palavras[index].palavraID)
  ? Colors.grey[300]
  : Colors.transparent,
```

Tudo pronto agora, podemos testar nossa aplicação. Altere um item qualquer, retorne para a listagem e veja que ele é encontrado e marcado com cinza. Estes efeitos visuais ficam a seu critério. Procure sempre lembrar de cuidar da altura do widget que representará os itens, coloque este valor em uma constante. Isso garantirá uma rolagem mais bonita.

9.7 Finalizações em nosso CRUD

Temos alguns pontos a serem repensados em nossa visão do CRUD. O usuário tem apenas o botão de gravar, não tem um para cancelar os dados informados e retornar ao estado original dos dados. Precisamos resolver isso. Outro ponto de que trataremos é o fato de o usuário poder retornar para a rota anterior, estando no CRUD, mesmo com os dados tendo sido alterados. Isso precisa ser evitado. Vamos também implementar isso.

Começaremos com o novo botão. Veja o código a seguir, que deve ser inserido no local de `RaisedButtonWithSnackbarWidget`, no método `_form()` de `PalavrasCRUDRoute`.

```
Row(
    mainAxisAlignment: MainAxisAlignment.end,
    children: <Widget>[
      RaisedButton(
        onPressed: formState.isFormValid ? () {} : null,
        child: Text('Cancelar'),
      ),
      SizedBox(
        width: 20,
      ),
      RaisedButtonWithSnackbarWidget(
```

```
        onPressedVisible: formState.isFormValid,
        buttonText: 'Gravar',
        successTextToSnackBar:
            'Os dados informados foram registrados com sucesso.',
        failTextToSnackBar: 'Erro na inserção',
        onButtonPressed: _onSubmitPressed,
        onSnackBarClosed: _resetForm,
      ),
    ],
),
```

Verifique que onde tínhamos um botão como último elemento de nossa `Column` temos agora uma `Row`, com um botão e um espaço, antecedendo o botão que já existia. Para facilitarmos aqui, eu optei em colocar a mesma regra para habilitar o botão de cancelar, mas ele poderia estar habilitado quando apenas um campo fosse preenchido. Fica a ideia para você implementar. Vamos ao código para quando o usuário pressionar o botão de cancelamento.

Antes de codificar o `RaisedButton`, precisamos ter dois métodos que serão responsáveis por reiniciar os dados quando o usuário solicitar o cancelamento do que já informou. Vamos criar estes novos métodos em nosso `palavras_crud_route.dart`. Observe que o comportamento está diretamente ligado a ser uma alteração ou inserção.

```
_clearTexts() {
    _palavraController.clear();
    _ajudaController.clear();
}

_restoreOriginalDataToTexts() {
    if (widget.palavraModel == null) {
      _clearTexts();
    } else {
      _palavraController.text = widget.palavraModel.palavra;
      _ajudaController.text = widget.palavraModel.ajuda;
    }
}
```

Agora vamos adaptar o `onPressed` de nosso botão de cancelar, tal qual eu apresento na sequência.

```
onPressed: formState.isFormValid ? _restoreOriginalDataToTexts : null,
```

Com essa implementação já podemos testar nossa alteração. Acesse o CRUD, para inserir e para alterar, modifique os valores e não grave, pressione "cancelar" e veja se o comportamento é desejado.

Falta-nos agora apenas uma situação: bloquear o retorno para a visão anterior à do CRUD, quando houver dados alterados pelo usuário. Você verá que é algo relativamente simples, faremos uso de um widget chamado `WillPopScope`, que captura o momento em que o usuário realizará o `pop`, para retornar à visão anterior à atual.

Em nossa `PalavrasCRUDRoute` temos um `Scaffold` como widget principal para retorno do `build`. Precisaremos mudar isso. Vamos inserir o `WillPopScope` e o `Scaffold` será atribuído com `child` para ele. Veja a implementação inicial para esta solução na sequência.

```
return WillPopScope(
  onWillPop: () async {
    return true;
  },
  child: Scaffold(...)
```

Pelo código, é possível verificar que a interação com o usuário, para confirmar ou não o fechamento, deve ser realizada no método atribuído ao evento `onWillPop`, mas criaremos um método que será responsável por isso e o consumiremos aqui. Vamos criar este método em um `Mixin` e, depois, você pode adaptar todo o código que está no widget para estar nele, caso queira, é claro.

Na pasta `mixin`, crie um arquivo chamado `palavras_crud_mixin.dart` e, nele, implemente o código a seguir. Observe que existe alguma similaridade com o que implementamos para a listagem de quando vamos remover um item. Pode ser verificada aqui uma questão de

redundância e criação de um único método, mas fica para você este desafio.

```
import 'package:cc04/models/palavra_model.dart';
import
'package:dialog_information_to_specific_platform/dialog_information_to_spe
cific_platform.dart';
import
'package:dialog_information_to_specific_platform/flat_buttons/actions_flat
button_to_alert_dialog.dart';
import 'package:flutter/material.dart';

mixin PalavrasCRUDMixin {
  Future<bool> onWillPop({BuildContext context, PalavraModel palavraModel,
String palavra, String ajuda}) async {
    if (palavraModel == null) {
      if (palavra.isEmpty && ajuda.isEmpty) return true;
    }

    if (palavraModel != null) {
      if (palavraModel.palavra == palavra && palavraModel.ajuda == ajuda)
        return true;
      if (palavra.isEmpty && ajuda.isEmpty) return true;
    }

    String oQueFazer = await showDialog(
      barrierDismissible: false,
      context: context,
      child: InformationAlertDialog(
        iconTitle: Icon(
          Icons.error,
          color: Colors.red,
        ),
        title: 'Quer sair?',
        message: 'Olha, os dados foram alterados, você vai descartá-los?',
        buttons: [
          ActionsFlatButtonToAlertDialog(
            messageButton: 'Não',
            isEnabled: true,
          ),
          //   InformationAlertDialog.createFlatButton(),
```

```
          ActionsFlatButtonToAlertDialog(
            messageButton: 'Sim',
            isEnabled: true,
          ),
          //   InformationAlertDialog.createFlatButton(),
        ],
      ),
    );
    return oQueFazer == 'Sim';
  }
}
```

Agora nos resta pouco, apenas adaptar nosso método para o evento `onWillPop` do `WillPopScope()`, que consumirá o método anterior e já podemos vê-lo na sequência. Precisaremos adicionar o `PalavrasCRUDMixin` ao `with` que já temos na declaração de nossa classe de estado do CRUD.

```
onWillPop: () async => await onWillPop(
  context: context,
  palavraModel: widget.palavraModel,
  palavra: _palavraController.text,
  ajuda: _ajudaController.text),
```

Temos uma última preocupação, que é na gravação de uma alteração. Quando o usuário realizar a gravação dos novos dados, ele precisará retornar para a listagem de maneira automática. Vamos então adaptar nosso método responsável por esta funcionalidade, tal qual vemos na sequência.

```
_resetForm() {
    _clearTexts();
    if (widget.palavraModel != null)
      Navigator.of(context).pop();
    else
      this._palavrasCrudFormBloc.add(FormReset());
}
```

Terminamos! Pode testar sua aplicação. Altere algum dado, seja em inserção ou alteração e tente retornar para a visão anterior, veja que

um dialog será exibido, cabendo ao usuário a decisão. Depois, realize a alteração completa de uma palavra, gravando-a. Veja se retornou automaticamente para a listagem.

Conclusão

Terminamos nossa implementação, proposta no capítulo anterior.

Estes dois capítulos foram intensos, de verdade. Foi uma longa etapa e com certa complexidade, mas tivemos muita coisa boa. Trabalhamos intensamente com BLoC, desenvolvemos e aplicamos técnicas importantes para integração em formulários de entrada de dados e visões que trazem uma lista de dados que possam ser exibidos ao usuário e com o qual ele possa interagir.

Concluímos a atividade iniciada no capítulo 7, persistindo em uma base de dados o que o usuário informar no aplicativo. Aprendemos a recuperar estes dados, atualizá-los e removê-los. Pudemos fazer uso de controle de exceções para identificar momentos onde erros possam ocorrer e então informar de maneira correta o usuário.

Apenas para enumerar o que vimos neste capítulo, que neste momento foram apenas widgets, trago aqui a relação deles, o que pode lhe auxiliar, caso queira, em uma pesquisa futura, específica para cada ponto trabalhado. São eles: `BlocBuilder`, `Dismissible`, `Exceptions`, `Future`, `Path Provider`, `ScrollController`, `SQLite`, `WillPopScope` e `UUID`.

Espero que estes dois capítulos tenham valido a pena para você. No próximo, começaremos a trabalhar a execução de nosso jogo. Quero pensar que será, além de interessante, divertido, pois veremos muita coisa nova. Vamos lá?

Capítulo 10
Funcionamento e interação do usuário com o jogo e o teclado para as letras

Neste capítulo e no próximo, vamos trabalhar a implementação e a execução do nosso jogo. Quando implementei essa funcionalidade me diverti muito e vi que podemos trabalhar diversos conceitos, técnicas e alguns outros recursos que ainda não vimos.

Na implementação que veremos, utilizaremos uma ferramenta muito interessante, cheia de recursos e com poucas limitações, se é que existem. É perfeita para animações 2D, não só para o Flutter, mas em um contexto geral de aplicação, principalmente a web. Estou falando do **Rive**, conhecido anteriormente como *Flare*. Este nome antigo ainda é utilizado e conhecido.

Você pode acessar o Rive em https://rive.app/ e buscar por recursos que lhe auxiliem no aprendizado no próprio portal ou na internet. Não temos espaço neste livro para trabalhar toda a extensão da ferramenta, que merece por si só um livro para ela. Aqui apenas utilizaremos o arquivo que gerei e que você pode acessar no GitHub do livro, ou diretamente em https://flare.rive.app/a/evertonfoz/files/flare/forca-casa-do-codigo/preview. No link anterior, você pode abrir o projeto no Rive e então exportar o arquivo para que ele seja gravado em seu equipamento.

Como gosto sempre de inovar, trarei aqui mais uma técnica para controle de estado dos widgets. Já conhecemos o `setState()`, vimos o BloC, que é supereficiente, mas veremos aqui o MobX, que tem uma implementação bem interessante para a gestão do estado de widgets.

De quebra, para trabalharmos com o MobX, veremos o GetIt, um componente de localização de serviços (*Service Locator*) que nos

auxiliará muito na recuperação de objetos inicializados e mantidos no contexto de nossa aplicação.

Veremos ainda um componente bem legal para exibir alertas de maneira diferente, mais bonitos. Prepare-se, este é um capítulo cheio de emoções.

10.1 Contextualização do jogo

O jogo da forca, ou *hangman game*, consiste em o jogador identificar uma palavra que tem suas letras exibidas como linhas tracejadas. Essas linhas são substituídas por seus respectivos valores, de acordo com os acertos do usuário.

E se/quando o usuário errar? Aí entra o nome do jogo. A cada erro, uma imagem de uma parte do corpo de um boneco simples é exibida, até que uma forca apareça e então o boneco seja enforcado. Trágico, não é? Mas é só um jogo, como outro qualquer.

O jogo termina quando o jogador causa, por erros de adivinhação, o enforcamento do boneco, ou quando toda a palavra é descoberta. Várias são as versões desse tipo de jogo, com uma gama enorme de cenários, que, em alguns casos, você pode replicar ou utilizar como base para criar o seu próprio.

Como o Rive permite o uso de imagens SVG (*Scalable Vector Graphic*), ou Gráficos Vetoriais Escalonáveis, eu não criei nada do zero. Utilizei um ícone disponível em: https://www.flaticon.com/free-icon/man-on-a-chair-before-suicide-with-a-hanging-rope_44003.

10.2 O esboço de layout para nosso jogo

Traremos recursos novos também. Vamos começar desenhando os espaços da tela de nosso dispositivo que precisaremos reservar para nossa interface com o usuário. Utilizaremos um widget chamado `PlaceHolder`. Esse controle é clássico quando se dimensiona uma interface com o usuário, deixando claro como os espaços serão aproveitados.

Você verá que neste capítulo trabalharemos de maneira muito passo a passo, mais do que vimos até agora, pois teremos toda uma lógica que precisa ser observada, além das interações com o usuário já conhecidas.

Começaremos com a criação de uma pasta chamada `jogo` dentro da pasta `routes` e, dentro dessa nova pasta, um arquivo chamado `jogo_route.dart`. Nele, criaremos uma classe básica de acordo com o código apresentado na sequência. Teremos alguns comentários após a listagem.

```
import 'package:flutter/material.dart';

class JogoRoute extends StatefulWidget {
  @override
  _JogoRouteState createState() => _JogoRouteState();
}

class _JogoRouteState extends State<JogoRoute> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: SafeArea(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Placeholder(fallbackHeight: 50, color: Colors.red[900]),
            Placeholder(fallbackHeight: 50, color: Colors.blue),
            Placeholder(fallbackHeight: 100, color: Colors.green),
            Placeholder(fallbackHeight: 350, color: Colors.yellow),
            Placeholder(fallbackHeight: 100, color: Colors.black),
          ],
```

```
        ),
      ),
    );
  }
}
```

Notou que no código anterior temos a estrutura clássica de uma aplicação que faz uso do Material Design? Temos o `Scaffold`, nele, o `SafeArea` e então um `Column` com 5 `Placeholders`, cada um com uma cor e altura máxima específicas. É possível não utilizar essas configurações, mas aí não teremos a dimensão que queremos para os componentes definitivos. Veja a figura a seguir, que apresenta o resultado da listagem anterior.

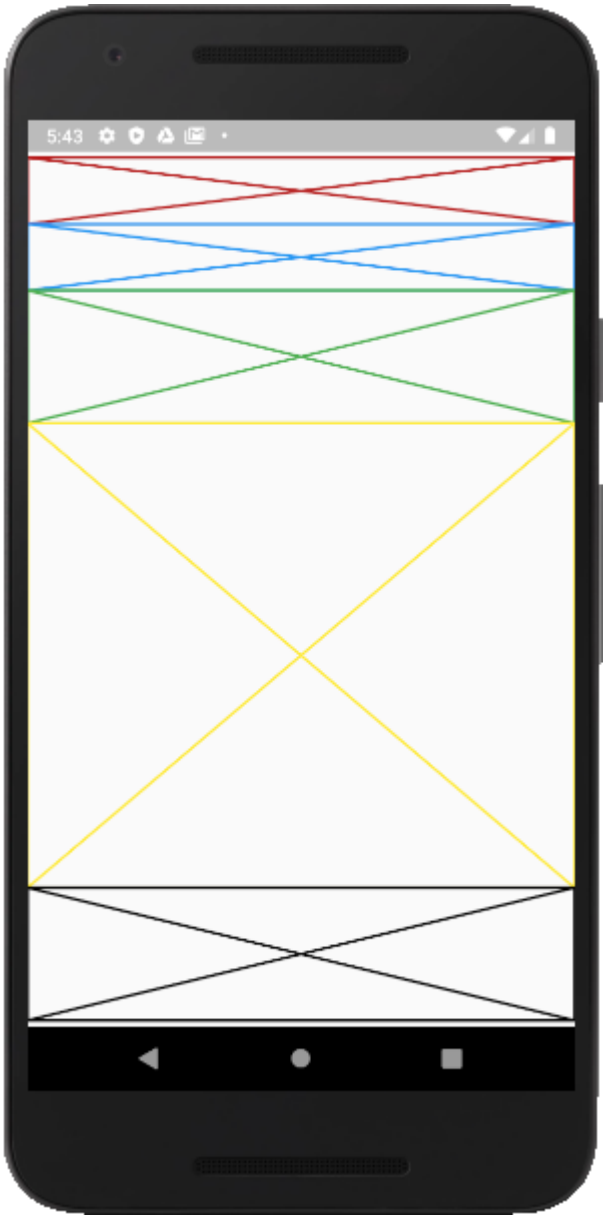

Figura 10.1: Protótipo com Placeholders

Vamos adaptar nosso aplicativo para você conseguir chegar a essa imagem? Em nosso arquivo `/appconstants/router_constants.dart`, insira a instrução:

```
const String kJogoRoute = '/jogo';
```

No arquivo `/apphelpers/app_router.dart`, insira a instrução a seguir antes do `default:`, lembrando do import para `JogoRoute()`:

```
case kJogoRoute:
    return MaterialPageRoute(builder: (_) => JogoRoute());
```

Como teremos uma interação pelo widget gerado por meio de nosso `_createListTile()`, precisamos inserir nele uma propriedade específica para capturar o `onTap`. Veja-a na sequência e lembre-se de ajustá-la também no `ListTile`. Os dois códigos estão a seguir.

```
// Parâmetro para o método
@required Function onTap

// Atribuição no ListTile
onTap: onTap,
```

Por fim, no arquivo `/drawer/widgets/drawerbodycontent_app.dart`, em nosso `_createListTile()`, para acessar o jogo, vamos configurar o `onTap` de seu `Tile`, de acordo com o seguinte código.

```
onTap: () {
    Navigator.of(context).pop();
    Navigator.of(context).pushNamed(kJogoRoute);
},
```

Com isso pronto, basta executar a aplicação e acessar a opção de jogar no `Drawer`.

Eu poderia explicar o que queremos inserir em cada placeholder da imagem anterior, mas a própria documentação do Flutter traz esse componente como algo a ser utilizado em última instância na prototipação, pois dá trabalho para configurar e não diz nada ao futuro usuário. Eu apenas quis trazê-lo para que você pudesse saber de sua existência.

Vamos trabalhar em um protótipo um pouco mais semântico. Veja a nova listagem na sequência, substituindo apenas o que existe no `children` de `Column`. Logo após, teremos a figura atualizada para este código e algumas explicações.

```
Container(height: 50, color: Colors.red[900]),
Container(height: 50, color: Colors.blue),
```

```
Container(height: 100, color: Colors.green),
Container(height: 350, color: Colors.yellow),
Container(height: 100, color: Colors.grey),
```

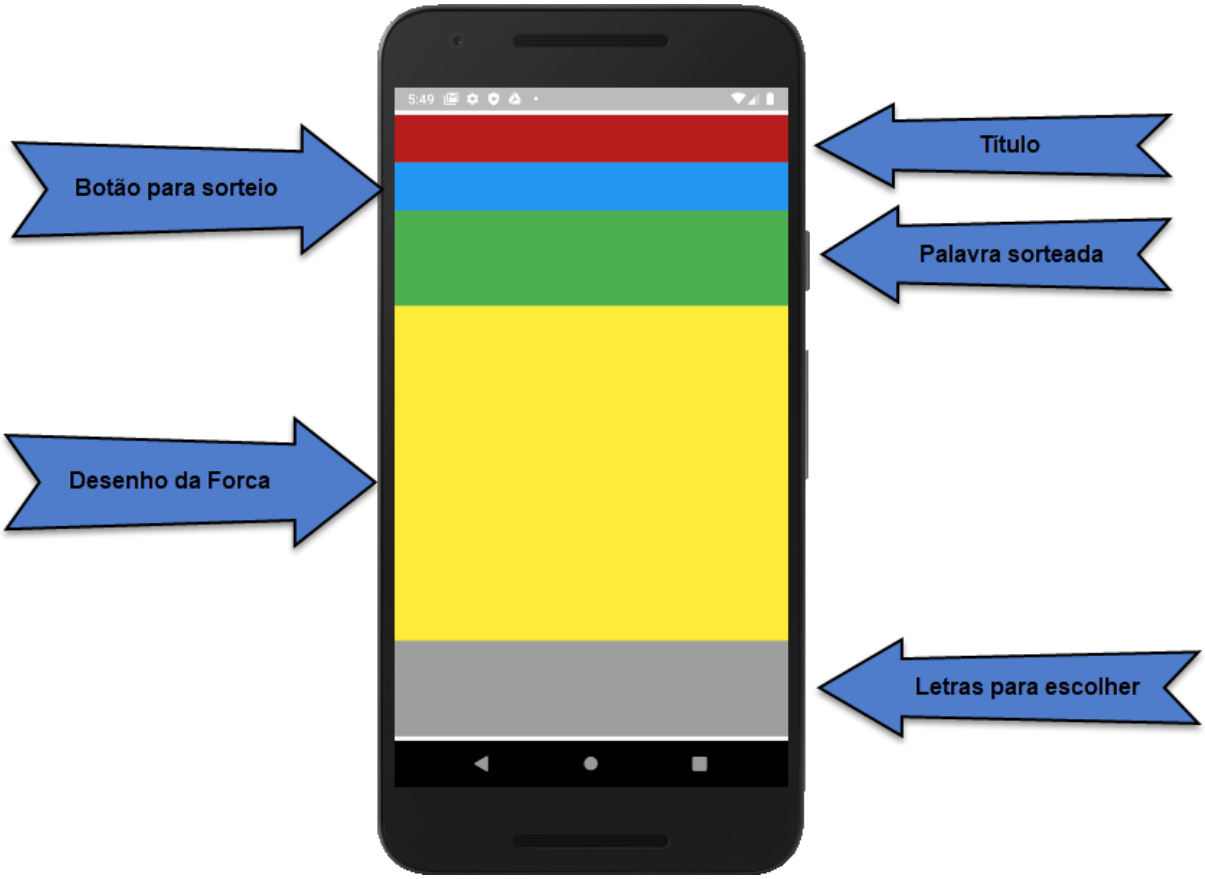

Figura 10.2: Protótipo com Contêineres

Verificando e comparando as duas imagens, não temos muita diferença de compreensão sem o auxílio dos textos que vemos nessa segunda figura, mas são maneiras de prototipação que você pode pensar para sua aplicação.

Quando formos desenvolver aplicações, é importante termos `mockups`, que são desenhos de telas normalmente não funcionais, mas que levam ao usuário um protótipo de como sua aplicação ficará. A minha recomendação, se você gostar de design, é se dedicar ao Adobe XD, que tem uma versão gratuita e há promessas de integração com o Flutter. Fica a dica.

10.3 Layout real do jogo

Agora, com base na imagem anterior, começaremos a desenvolver o layout real do nosso jogo. Começaremos de cima para baixo, ou seja, vamos começar com o título da visão e teremos uma subseção para cada componente que implementaremos.

O título do jogo

Vamos aplicar algumas técnicas que vimos durante o livro. Sempre que criamos visões, a recomendação é que apenas `build()` a renderize e que não tenha muito código nele, buscando deixar o código mais limpo e fácil de compreender.

Trabalharemos com mixins e com a criação de widgets para o desenho de nossa visão do jogo. Sendo assim, na pasta `/routes/jogo`, crie outra pasta chamada `mixins`. Nela, crie um arquivo chamado `jogo_mixin.dart` com o código da listagem a seguir. Todo o código trazido já é conhecido por nós, dispensando comentários. Também é comum trabalharmos herança e agregação na definição de responsabilidades.

```
import 'package:flutter/material.dart';

mixin JogoMixin {
  titulo() {
    return Padding(
      padding: const EdgeInsets.only(top: 10.0, bottom: 15),
      child: Text(
        'Vamos jogar a Forca?',
        style: TextStyle(
          fontSize: 30,
        ),
      ),
    );
  }
}
```

Precisamos indicar que utilizaremos esse mixin em nossa visão. Dessa maneira, adicione o `with` na declaração da classe de nossa rota para o jogo, conforme a listagem a seguir. Lembre-se do `import .`

```
class _JogoRouteState extends State<JogoRoute> with JogoMixin {
```

Seguindo com a implementação, vamos substituir o primeiro `Container()` de nossa visão pela invocação ao método criado no mixin. Veja a listagem a seguir e note como o nosso `build()` fica simples.

```
titulo(),
```

Agora você pode verificar a execução da aplicação em seu dispositivo/emulador. Como essa mudança é simples, não trarei aqui a imagem do app em execução, deixarei para mostrar na próxima implementação.

O botão para o sorteio da palavra

Nós temos já implementada a persistência de palavras que serão utilizadas em nosso jogo e vamos utilizá-las aqui. O momento em que usaremos essas palavras será no início do jogo, que dependerá de uma interação do usuário ao pressionar um botão para que a palavra seja selecionada.

Vamos então criar este componente em nosso `JogoMixin` com o código apresentado na sequência. Após ele, temos alguns comentários.

```
botaoParaSorteioDePalavra() {
    return Container(
      padding: const EdgeInsets.only(bottom: 5.0),
      height: 50,
      decoration: new BoxDecoration(
        boxShadow: [
          BoxShadow(
            color: Colors.indigo,
```

```
            blurRadius: 20.0,
            spreadRadius: 1.0,
            offset: Offset(
              5.0,
              5.0,
            ),
          )
        ],
      ),
      child: FlatButton(
        child: Text('Pressione para sortear uma palavra'),
        color: Colors.blue[200],
        onPressed: () {},
      ),
    );
}
```

Em relação ao código anterior, já conhecemos o `Container` e o `BoxDecoration`, mas ainda não tínhamos trabalhado o `BoxShadow`, que será responsável por desenhar uma sombra em torno do nosso contêiner. Além disso, nosso contêiner conterá nosso `FlatButton`, também novo para nós no livro.

Observe que, além de `color`, que é fácil entendermos, nosso `BoxShadow` traz o `blurRadius`, que permite suavizar a sombra desenhada, o `spreadRadius`, que estende a sombra, e o `offSet`, que utiliza `Offset()` com dois parâmetros, sendo que o primeiro desloca a sombra horizontalmente e o segundo, verticalmente.

Para consumirmos este novo método em nossa visão, substitua o segundo contêiner pela invocação a esse método. Para auxiliar, trago na sequência o código que temos até agora para `children` de `Column`.

```
children: <Widget>[
    titulo(),
    botaoParaSorteioDePalavra(),
    Container(height: 100, color: Colors.green),
    Container(height: 350, color: Colors.yellow),
```

```
    Container(height: 100, color: Colors.grey),
],
```

Agora sim, com duas alterações podemos trazer uma figura que representará a visão até o momento. Procure brincar com os valores que utilizamos para ver as mudanças, pode ser legal.

Figura 10.3: Título e botão desenhados para o jogo

A palavra que deve ser adivinhada pelo jogador

Vamos trabalhar agora em algo que é extremamente simples, o desenho com linhas tracejadas da palavra que deve ser adivinhada pelo jogador. Vamos seguir o que estamos fazendo e criar um novo método em nosso mixin com o código a seguir. Algumas reflexões estão após ele.

```
palavraParaAdivinhar({String palavra}) {
    return Padding(
      padding: const EdgeInsets.only(top: 20.0, bottom: 10),
      child: Text(
        palavra,
        style: TextStyle(
          fontSize: 30,
        ),
      ),
    );
}
```

Você notou similaridade entre esse método e o que fizemos para o título? Pois é. A mudança é que esse método recebe o valor a ser exibido e o `top` para o `padding`. Isso nos remete ao reúso de código, algo bem trabalhado em Orientação a Objetos. Vamos refatorar isso. Veja um novo método a ser implementado em nosso mixin na sequência. Note que o `_` define o método como privado.

```
_text({String text, EdgeInsets edgeInsets}) {
    return Padding(
      padding: edgeInsets,
      child: Text(
        text,
        style: TextStyle(
          fontSize: 30,
        ),
      ),
);
```

Recebemos o texto a ser desenhado e o objeto que deverá ser atribuído ao nosso `padding`. Isso nos dá liberdade para reutilizar esse método. É claro que poderíamos customizar mais, recebendo nele o tamanho da fonte, cor, mas isso fica para você, ok? Nós

agora vamos consumir esse método nos dois que são redundantes. Veja a nova implementação deles na sequência.

```
titulo() {
    return _text(
      text: 'Vamos jogar a Forca?',
      edgeInsets: const EdgeInsets.only(
        top: 10.0,
        bottom: 15,
      ),
    );
}

palavraParaAdivinhar({String palavra}) {
    return _text(
      text: palavra,
      edgeInsets: const EdgeInsets.only(
        top: 20.0,
        bottom: 10,
      ),
    );
}
```

Precisamos adaptar nossa rota para exibir a palavra a ser adivinhada. Veja-a na sequência, apenas como ilustração.

```
palavraParaAdivinhar(palavra: '_ _ _ _ _ _ _ _ _ _'),
```

Aqui cabe uma análise que implementaremos no momento certo, mas que julgo importante comentar agora. O botão só será exibido quando o jogo não tiver iniciado e a palavra só será exibida quando o jogo estiver em andamento. Vá pensando nisso. Deixarei a figura da visão para mais tarde.

A animação da forca

Chegamos ao momento em que traremos para nossa aplicação a animação criada no `Rive` que comentamos no início do capítulo, o que nos leva à necessidade de instalar um componente que leia o

arquivo de animação e o exiba em nossa visão. Este componente é o `flare_flutter`.

Já sabemos de capítulos anteriores como instalar em nosso projeto as dependências. Então, vamos inserir esta em nosso arquivo `pubspec.yaml`, logo abaixo das que já possuímos. Veja o código a ser inserido na sequência.

```
flare_flutter: ^2.0.3
```

Nós criaremos um novo método em nosso `jogo_mixin.dart`, que será responsável por renderizar nossa animação. Dessa maneira, precisamos importar o pacote necessário para isso. No início do arquivo, insira o código a seguir.

```
import 'package:flare_flutter/flare_actor.dart';
```

Agora sim, podemos implementar nosso novo método. Ele está na sequência para que você possa lê-lo, com comentários após a listagem.

```
animacaoDaForca({String animacao}) {
    return Expanded(
      child: FlareActor(
        "assets/flare/forca_casa_do_codigo.flr",
        alignment: Alignment.center,
        fit: BoxFit.contain,
        animation: animacao,
      ),
    );
}
```

Antes de tudo, observe que o método recebe um argumento que será um nome relativo à animação que deverá ser renderizada. Se você abriu a animação no Rive, deve ter notado que lá temos uma guia com a relação das animações criadas.

Também é importante que você copie o arquivo para a pasta chamada `flare`, dentro de `assets`, que precisa ser criada, ou altere o código anterior para o local que está usando.

Como eu criei a pasta `flare` para as animações, é preciso liberar o acesso para ela no `pubspec.yaml` da mesma forma que fizemos para as imagens. Veja a instrução a seguir que deve ser implementada abaixo das liberações já existentes.

```
- assets/flare/
```

Quase lá. Vamos substituir, em nossa visão, o quarto contêiner pela invocação deste novo método. Veja a instrução a seguir.

```
animacaoDaForca(animacao: 'idle'),
```

`idle` é o nome de uma animação criada em nosso arquivo e temos outras para cada fase do jogo que logo veremos. Agora sim, vamos apresentar uma figura com o estado atual de nossa visão. Veja a figura a seguir.

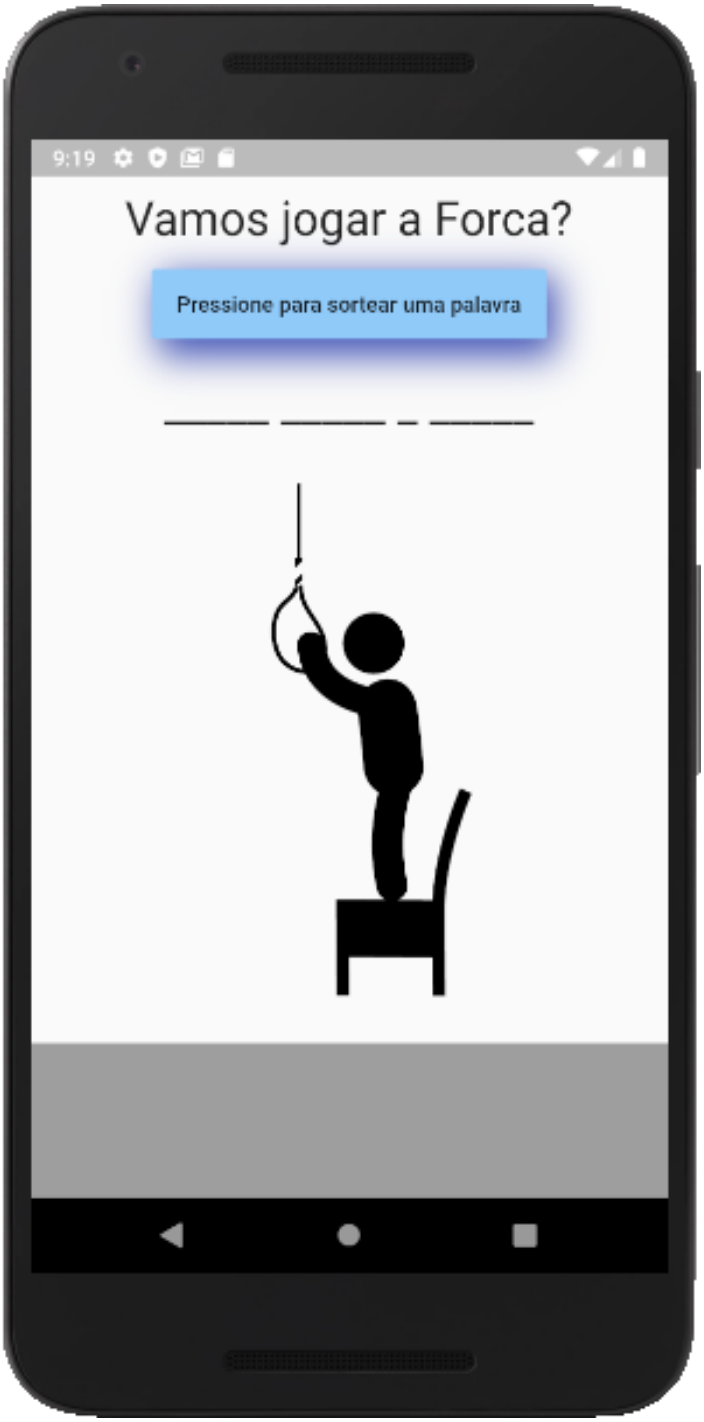

Figura 10.4: A palavra para adivinhar e animação

Uma atenção especial é que a figura pode demorar um pouquinho para aparecer da primeira vez, pois há todo o processo de carregamento e transformação do arquivo.

O teclado com as letras para seleção

Estamos quase terminando nossa visão. Só nos falta uma área onde o usuário possa indicar as letras que ele acredita que estejam na palavra sorteada e é essa área que implementaremos agora. Inicialmente apenas a desenharemos, pois ela terá uma certa complexidade e ajustaremos isso conforme a lógica do jogo for sendo apresentada.

Vamos então implementar um novo método em nosso `jogo_mixin.dart`, que é bem interessante. Ele está na sequência e, após ele, os comentários.

```
letrasParaSelecao({String letras}) {
    List<Widget> textsParaLetras = List<Widget>();

    for (int i = 0; i < letras.length; i++) {
      textsParaLetras.add(Text(
        letras[i],
        style: TextStyle(
          fontSize: 40,
        ),
      ));
    }

    return Padding(
      padding: const EdgeInsets.only(left: 10, right: 10, bottom: 10.0),
      child: Wrap(
        alignment: WrapAlignment.center,
        spacing: 20,
        runSpacing: 5,
        children: textsParaLetras,
      ),
    );
}
```

Notou que recebemos uma `String` chamada `letras`? Essa variável terá todas as letras que devem ser desenhadas em nosso teclado virtual. Quando formos invocar esse método, você verá como será.

Logo de início, criamos um `List<Widget>`, pois para cada letra precisaremos de um controle visual com o qual o usuário poderá interagir e, em nosso caso, serão `Texts`. Verifique o `for()` e observe que percorremos a string recebida como se fosse uma matriz e, a cada elemento, adicionamos um `Text` em nosso `list`.

Terminado o laço, precisamos retornar esses componentes, o que faremos com um widget novo aqui no livro, o `Wrap`. Este componente pode funcionar como um `Column` ou um `Row`, dependendo sempre das configurações de seus parâmetros. Em nosso caso, estamos usando o padrão que é comportamento do `Row`. Estamos alinhando seu conteúdo ao centro (`alignment`) com espaços de `20` entre cada componente (`spacing`) e um espaço entre cada linha de `20` (`runSpacing`). Terminamos a invocação ao `Wrap` enviando nosso `list` para `children`.

Para finalizar a explicação, envolvemos o `Wrap` em um `Padding` para termos espaços ao lado e na base.

Agora vamos ao consumo deste novo método. Em nossa visão do jogo, substitua o último `Container` pela instrução a seguir.

```
letrasParaSelecao(letras: 'ABCDEFGHIJKLMNOPQRSTUWXYZ'),
```

Veja as letras que estamos enviando para composição do teclado, estamos idealizando apenas letras sem acentos ou cedilha. O controle para isso envolveria uma complexidade maior e, para o momento, isso nos atenderá. Ficará para você o desafio, podendo optar por inserir na string essas letras especiais ou trabalhar uma lógica de semelhança, como pelo `A` verificar a existência de `Á`, `À` ou `Ã`. Lembra da função de retirada de acentos que vimos no capítulo anterior? Fica a dica.

Muito bem. Vamos ver nossa visão do jogo. Veja a figura a seguir.

Figura 10.5: A visão completa para o jogo

10.4 A implementação para o funcionamento do jogo

Já temos toda a interface construída, mas ela ainda não é funcional. Precisamos implementar a funcionalidade necessária e vamos por partes, tal qual fizemos para o desenho dela.

Configuração do MobX e GetIt

Para essa implementação, faremos uso do MobX e GetIt como comentado no início do capítulo. Precisamos preparar nossa estrutura para isso. Sendo assim, vamos inserir as dependências a seguir em nosso `pubspec.yaml` .

```
mobx: ^1.1.1
get_it: ^4.0.2
```

O `MobX` trabalha com a geração de código com base em `decorators` e este código, para ser gerado, precisa ter a execução de comandos externos via terminal. Vamos usar um componente que nos auxiliará nisso e ele é configurado também no `pubspec.yaml` , mas como `dev_dependencies` , como fizemos para a geração dos códigos `Json` na criação de nossos modelos. Veja na sequência a instrução a ser inserida.

```
mobx_codegen: ^1.0.3
```

Após essas configurações, lembre-se de executar o `Packages get` , que o `Android Studio` oferece quando o `pubspec.yaml` está aberto.

O início com MobX

O MobX é um componente que leva para si parte da responsabilidade da manutenção de estado de dados que são consumidos por sua aplicação. Não cabe aqui um aprofundamento no componente, mas veremos o que será necessário para nossa implementação.

Três são os conceitos básicos implementados pelo MobX: `Observables`, `Actions` e `Reactions`. Os `Observables` são, em sua essência, "variáveis" que sofrem alteração em algum momento da aplicação e, quando essa alteração ocorre, é preciso que quem a esteja utilizando seja notificado e tenha seu dado atualizado. `Actions` são métodos responsáveis por registrar as mudanças de valores nas "variáveis". Mais tecnicamente, as variáveis referem-se a um estado de determinado objeto. Dessa maneira, as `Actions` registram essa transição de estado. Por fim, as `Reactions` são "eventos" que capturam o exato momento da transição, levando ao desenvolvedor a possibilidade de interagir com a informação e realizar algo relacionado à lógica do negócio.

Para implementarmos as características comentadas, precisamos abstrair uma hierarquia chamada `widget-store-service`, na qual `widget` representa o estado que trabalharemos em nossa interface com o usuário, `store` contém as informações sobre o estado que temos no `widget`, e `service` é utilizado para a recuperação de dados que desejamos manter no `store`. Existem algumas arquiteturas para essa hierarquia, mas aqui trabalharemos com um cenário em que cada visão (rota/página) estará ligada a um `store` e os `services` podem ser utilizados por vários stores. Parece complicado? Não é. Logo veremos na prática.

Você verá similaridade com a funcionalidade do BLoC. Eu particularmente gosto mais do MobX para gestão de estado, mas é você quem definirá o que é melhor para você e seu projeto.

Implementação de stores para o MobX

Vamos começar nossa implementação passo a passo. Inicialmente vamos criar uma pasta chamada `mobx_stores` dentro de `/routes/jogo/` e, nela, um arquivo chamado `jogo_store.dart` com o código apresentado na sequência.

```
import 'package:mobx/mobx.dart';
```

```
abstract class _JogoStore with Store {
  @observable
  String palavraParaAdivinhar;

  @observable
  String ajudaPalavraParaAdivinhar;
}
```

Observe a importação do pacote logo no início. Em seguida, veja que estamos declarando uma classe abstrata e privada, o que impossibilita sua instanciação e o acesso externo a esse arquivo. Ainda, estamos usando um mixin para `Store` e, a princípio, não estamos usando nada dele. Dentro da classe temos duas propriedades, ambas decoradas com `@observable`. Isso significa que, quando tudo estiver certo e essas propriedades forem alteradas, elas notificarão quem as estiver usando.

Precisamos agora de uma classe concreta e faremos isso com a seguinte implementação, que, como costume, é feita antes da classe abstrata exibida anteriormente.

```
class JogoStore = _JogoStore with _$JogoStore;
```

Deu um erro em `_$JogoStore`, certo? É esperado, pois não temos esse mixin implementado e nem seremos nós que o implementaremos, será o `mobx_codegen` e, se você lembra de quando criamos o arquivo para JSON, no capítulo 8, precisamos ter a declaração de `part` para o arquivo a ser gerado. Insira a instrução a seguir antes da listagem anterior.

```
part 'jogo_store.g.dart';
```

Também teremos erro nessa implementação, pois o arquivo não existe. Precisamos criá-lo. Vamos acessar o terminal e, na pasta de nosso projeto, vamos executar a instrução a seguir, já conhecida nossa. Mas, por garantia, execute antes o `pub get`.

```
flutter packages pub run build_runner build
```

Caso o erro `Conflicting outputs were detected and the build is unable to prompt for permission to remove them` apareça, há instruções de execução com um flag para correção, mas, para facilitar, é só executar a instrução a seguir.

```
flutter packages pub run build_runner build --delete-conflicting-outputs
```

Com a execução da instrução anterior bem-sucedida, nossos erros desaparecem e o arquivo parcial é criado. Se você quiser, pode estudá-lo, mas não o altere.

É preciso saber que, a cada alteração em nosso store, esse processo precisa ser executado, o que é trabalhoso mas necessário já que, se esquecermos, podemos ficar um tempo buscando por erros que na realidade não existem na lógica. Dessa maneira, vamos executar no terminal a instrução a seguir, que fica monitorando alterações e, caso ocorram, o processo é executado. Lembre-se de garantir que o `watch` esteja em execução durante o desenvolvimento.

```
flutter packages pub run build_runner watch
```

Vamos dar sequência a nossa implementação. Já temos duas propriedades observáveis, agora precisamos de uma ação que registre alteração nelas. Veja o código a seguir, que deve ser implementado após as propriedades. O código é simples e dispensa explicações. Para nossa implementação, não faremos uso de *reactions* agora, mas daqui a pouco verificaremos.

```
@action
registrarPalavraParaAdivinhar({String palavra, String ajuda}) {
    this.palavraParaAdivinhar = palavra;
    this.ajudaPalavraParaAdivinhar = ajuda;
}
```

Disponibilização do store para a aplicação

Já temos nosso store criado, precisamos utilizá-lo. A recomendação da documentação é a criação do store diretamente no `main.dart`

para que ele possa estar disponível para toda a aplicação e assim não tenhamos execução redundante desse processo. Porém, embora eu siga aqui essa recomendação, acho mais interessante termos o store sempre inicializado no widget ao qual ele estará ligado. Mas, para que possamos conhecer outro componente, aumentando nosso conhecimento, vamos seguir a estratégia recomendada.

Com o exposto, já definimos que a instanciação de nosso `JogoStore()` gerará o objeto `Store` de que precisaremos no início da aplicação. Mas como armazená-lo e recuperá-lo em qualquer lugar da aplicação? Aí entra o GetIt.

Vamos criar uma variável que representará um objeto do `GetIt` para acessarmos em toda a aplicação. Assim, na pasta `functions`, crie um arquivo chamado `getit_function.dart` e nele coloque o código a seguir. O código é bem simples, obtém apenas uma instância de `GetIt`.

```
import 'package:get_it/get_it.dart';

GetIt getIt = GetIt.instance;
```

Agora, em nosso `main.dart`, vamos utilizar a variável anterior e o registro de nosso store. Veja o código a seguir. Note que o `main()` não está como arrow function, mas sim como bloco. Atente-se aos imports.

```
void main() {
  getIt.registerSingleton<JogoStore>(JogoStore());
...
}
```

Caso você prefira não utilizar variáveis globais, é possível utilizar `GetIt.I.registerSingleton<JogoStore>(JogoStore());` em seu código.

10.5 A interação do usuário com o jogo

Já temos toda nossa infraestrutura preparada. Podemos agora começar a trabalhar com a interação do usuário. Nesta seção, daremos continuidade ao trabalho com MobX e GetIt, pois são peças fundamentais para o funcionamento do jogo.

O início do jogo

Sabemos que, para o jogo começar, é preciso termos uma palavra sorteada dentre as que temos registradas. Para que o sorteio ocorra, precisamos aguardar a interação do usuário com o botão que temos em nossa visão e alguns passos devem ser seguidos em nossa lógica quando isso ocorrer. Vamos comentando-os aqui e implementando conforme surgirem.

Como trabalharemos com nosso store registrado no método `main()`, precisamos agora recuperá-lo em nossa visão, pois é ele que será responsável por cuidar dos dados consumidos por nossa visão. Portanto, precisamos ter uma variável para nosso store em nossa visão. Veja o código a seguir com a declaração dessa variável logo no início da classe. Lembre-se do import.

```
class _JogoRouteState extends State<JogoRoute> with JogoMixin {
  JogoStore _jogoStore;
  ...
}
```

Na sequência, sobrescreveremos o método `initState()` para nele inicializarmos essa variável com nosso store já registrado e faremos isso também pelo GetIt. Aqui eu utilizo, com mais frequência, a instanciação do store, como comentei anteriormente. Veja o código a seguir, será preciso o import.

```
@override
void initState() {
    super.initState();
```

```
    _jogoStore = getIt.get<JogoStore>();
}
```

Precisamos invocar nossa action para registrar a palavra sorteada, o que ocorrerá na interação com o botão da página. Precisamos também saber que temos este comportamento implementado no método `botaoParaSorteioDePalavra()` em nosso mixin, responsável por sua geração. Lá em nosso código, temos o evento `onPressed` do `FlatButton`, que conterá a função a ser executada no momento da interação.

Não podemos ter essa função no mixin, pois ele não conhece nosso store e não queremos enviá-lo como parâmetro. O que fazer? Vamos enviar a função para o `onPressed` como parâmetro, um `Callback function`. Veja no código a seguir a nova assinatura para o método.

```
botaoParaSorteioDePalavra({@required Function onPressed}) {
    ...
}
```

Com isso, ao `onPressed` de nosso `FlatButton`, vamos atribuir a função recebida. Veja o novo código na sequência. Para facilitar, trouxe todo o código para o botão.

```
child: FlatButton(
    child: Text('Pressione para sortear uma palavra'),
    color: Colors.blue[200],
    onPressed: onPressed,
),
```

Agora vamos voltar para nossa visão e mudar a invocação para o método. Veja o código a seguir. Note que estamos enviando dados constantes apenas para teste.

```
botaoParaSorteioDePalavra(
  onPressed: () => this._jogoStore.registrarPalavraParaAdivinhar(
      palavra: 'teste', ajuda: 'ajuda para teste'),
),
```

Vamos testar os *reactions* agora? Criaremos um uso apenas para visualizarmos seu funcionamento. Não será uma implementação final. É importante sabermos que o objetivo de reactions é dar à aplicação o conhecimento de quando uma propriedade observável sofre alterações. Começaremos com a declaração de uma variável logo após a declaração de nosso store. Veja o código a seguir, que traz também uma variável que usaremos como flag para nosso teste e uma string com a ajuda para o jogador.

```
class _JogoRouteState extends State<JogoRoute> with JogoMixin {
  JogoStore _jogoStore;
  List<ReactionDisposer> _reactionDisposers;
  bool _jogoIniciado = false;
  String _ajudaParaPalavra = '';
  ...
}
```

Para que possamos capturar a mudança de nossas propriedades observáveis, precisamos sobrescrever um método chamado `didChangeDependencies()`, que é invocado sempre que uma mudança nas dependências do estado atual do widget ocorre, que, em nosso caso, se dá nas propriedades observáveis de nosso store. Veja o código a seguir para nosso `jogo_route.dart`. Temos algumas explicações após o código.

```
@override
void didChangeDependencies() {
    super.didChangeDependencies();
    _reactionDisposers ??= [
      reaction(
        (_) => _jogoStore.palavraParaAdivinhar,
        (String palavra) => print('nova palavra: $palavra'),
      ),
      reaction(
        (_) => _jogoStore.ajudaPalavraParaAdivinhar,
        (String ajuda) {
          print('nova ajuda: $ajuda');
          setState(() {
            this._jogoIniciado = !this._jogoIniciado;
```

```
            this._ajudaParaPalavra = ajuda;
          });
        },
      ),
    ];
}
```

Veja o operador `??` na atribuição para `_reactionDisposers`. Ele garante que a atribuição ocorra apenas caso a variável que receberá a atribuição seja nula. Isso garante a não reatribuição a cada invocação do método.

Como `_reactionDisposers` é um `List`, tratamos como uma matriz e atribuímos dois elementos `reaction()` a ele. Este método recebe duas funções como argumento. A primeira retorna qual propriedade observável terá a reação capturada e a segunda recebe o novo valor para a propriedade. Em nosso exemplo, para `palavra`, estamos apenas imprimindo na console seu valor.

Na segunda `reaction`, atualizamos as variáveis `_jogoIniciado` e `_ajudaParaPalavra` dentro de um `setState()` para que o `build()` possa novamente ser executado e possamos trabalhar com esse novo valor. Mas é só para testarmos a ideia com MobX, que é evitar o uso do `setState()`, como feito com BLoC.

Sempre que trabalharmos com reactions, precisamos liberar os recursos consumidos e isso deve ser feito na sobrescrita do método `dispose()`, como é apresentado na sequência.

```
@override
void dispose() {
    _reactionDisposers.forEach((d) => d());
    super.dispose();
}
```

Para ver o funcionamento de nossos reactions, vamos alterar a invocação para o método que renderiza o botão de início de jogo. Mude para o código a seguir. Note que trouxemos um novo widget,

o `Visibility()`, que avalia uma condição para `visible`, caso o valor seja verdadeiro, renderiza o `child`.

```
Visibility(
  visible: !this._jogoIniciado,
  child: botaoParaSorteioDePalavra(
    onPressed: () => this._jogoStore.registrarPalavraParaAdivinhar(
        palavra: 'teste', ajuda: 'ajuda para teste'),
  ),
),
```

Podemos aproveitar o momento e inserir a ajuda para a palavra logo abaixo dela, apenas como teste, sem muita configuração, mas fique à vontade para isso. Veja o código a seguir, que deve ser inserido logo após a invocação a `palavraParaAdivinhar(palavra: '_ _ _ _ _ _ _ _ _ _');`. Veja que nosso flag agora não é a negação da variável. Este é um novo controle, não desenhado em nossos modelos anteriores.

```
Visibility(
  visible: this._jogoIniciado,
  child: Text(
    this._ajudaParaPalavra,
    textAlign: TextAlign.center,
  ),
),
```

Reinicie a aplicação, vá ao jogo. Veja que o botão aparece. Pressione o botão, veja que ele desaparece e a ajuda para a palavra é exibida. Legal, não é? Fizemos isso para usar o reaction, mas faremos isso tudo ser mais legal ainda. Chegou a ver o console com os resultados dos reactions?

O `Visibility` tem outras propriedades e talvez seja interessante você dar uma estudada, mas fica a seu critério.

A atualização com Observers

Foi muito legal o que vimos anteriormente, mas não queremos utilizar o `setState()`. Vamos então realizar algumas alterações em

nosso código e remover o que implementamos para testar os reactions. Sendo assim, comente tudo relacionado a reactions que fizemos anteriormente, tal qual as orientações a seguir.

1. Deixe declarada apenas a variável `_jogoStore` em nosso widget. Você pode comentar as demais em vez de retirá-las.
2. Faça o mesmo para a atribuição de `_reactionDisposers` no `didChangeDependencies()`.
3. Mesmo procedimento no `dispose()` para `_reactionDisposers`.

Isso gerará alguns erros, mas no momento não se preocupe com isso. Agora trabalharemos na adaptação de nossos widgets `Visibility`. Vamos mudar de widget. Utilizaremos um específico para o uso de MobX e precisamos registrá-lo em nossas dependências no `pubspec.yaml`. Veja isso na sequência. Lembre-se do `Packages get`, ok?

```
flutter_mobx: ^1.1.0
```

Vamos começar a implementação inserindo em nosso `jogo_mixin.dart` o método a seguir, que será o responsável por exibir a ajuda para a palavra sorteada.

```
ajudaParaAdivinharAPalavra({String ajuda}) {
    return (ajuda != null)
        ? _text(
            text: ajuda,
            edgeInsets: const EdgeInsets.only(
              top: 10.0,
              bottom: 15,
            ),
          )
        : Container();
}
```

Vamos deixar as partes relacionadas ao botão, palavra e ajuda, em nosso `jogo_route.dart` da seguinte maneira. Observe que mudamos o valor de `ajuda` em `ajudaParaAdivinharAPalavra()` para `null`. Os erros anteriores deixarão de existir agora.

```
botaoParaSorteioDePalavra(
  onPressed: () => this._jogoStore.registrarPalavraParaAdivinhar(
      palavra: 'teste', ajuda: 'ajuda para teste'),
),
palavraParaAdivinhar(palavra: '_____ _____ _ _____'),
ajudaParaAdivinharAPalavra(ajuda: null),
```

Faremos uma mudança em nosso código que requer uma atenção. Vamos substituir o `child` de nosso `SafeArea` para o código a seguir. Observe que antes o `child` era `Column`. Este `Column` agora é o retorno para o `builder()` de `Observer`. Leia o código e as observações após ele.

```
child: Observer(
  builder: (_) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        titulo(),
        botaoParaSorteioDePalavra(
          onPressed: () => this
              ._jogoStore
              .registrarPalavraParaAdivinhar(
                  palavra: 'teste', ajuda: 'ajuda para teste'),
        ),
        palavraParaAdivinhar(palavra: '_____ _____ _ _____'),
        ajudaParaAdivinharAPalavra(ajuda: ''),
        animacaoDaForca(animacao: 'idle'),
        letrasParaSelecao(letras: 'ABCDEFGHIJKLMNOPQRSTUWXYZ'),
      ],
    );
  },
),
```

O primeiro ponto é o `Observer`, que precisará de uma importação para o `flutter_mobx` registrado há pouco. Este widget possui um parâmetro, `builder`, que é o responsável pela renderização de um widget que tenha propriedades observáveis. Nós ainda não temos nenhuma propriedade observável no código, por isso um erro pode aparecer em seu console durante a execução. Não se preocupe.

Vamos ver a mágica? Altere o parâmetro `ajuda` na invocação do método `ajudaParaAdivinharAPalavra()` para o que vemos na sequência. Reinicie sua aplicação, inicie o jogo e veja como a atualização funciona. Bem melhor, não é?

```
ajudaParaAdivinharAPalavra(
    ajuda: this._jogoStore.ajudaPalavraParaAdivinhar),
```

Seleção da palavra na base de dados

Já estamos exibindo a ajuda para a palavra selecionada, mas ainda não estamos escolhendo uma palavra que esteja na coleção registrada pelo jogador. Vamos fazer isso agora. Você verá que é bem simples e que já temos muito do necessário implementado, mas teremos algumas mudanças e novas implementações. A primeira delas será ter uma lista para as palavras registradas em nosso store. Veja a declaração na sequência, que deve ser feita no `jogo_store.dart`. Será preciso um import.

ATENÇÃO: é pré-requisito que você tenha registrado palavras em sua base de dados quando implementamos a aplicação no capítulo 8, ok?

```
abstract class _JogoStore with Store {
  List<PalavraModel> _palavrasRegistradas= [];
  ...
}
```

Essa variável terá nela registradas todas as palavras que serão recuperadas da base de dados. Teremos algumas operações, mas as veremos aos poucos. Vamos ao método que realizará a carga das palavras persistidas na base de dados. Veja-o na sequência, ainda em nosso store. Precisaremos importar o DAO.

```
Future<List<PalavraModel>> _carregarPalavras() async {
    try {
      PalavraDAO palavraDAO = PalavraDAO();
      final List data = await palavraDAO.getAll();
      return data.map((palavra) {
```

```
        return PalavraModel.fromJson(palavra);
      }).toList();
    } catch (exception) {
      rethrow;
    }
}
```

Observe que é um método privado e assíncrono que faz uso de nosso `PalavraDAO` e, com o retorno obtido de `getAll()`, um mapeamento dos dados para um `List` ocorre e então é retornado para quem consumir esse método. Essa implementação já não é novidade para nós.

Com essa nova estratégia que estamos implementando, precisamos alterar nosso método `registrarPalavraParaAdivinhar()`, que é público, para privado. Para isso, basta alterar seu nome para ter como prefixo o `_`, ou seja, o novo método será `_registrarPalavraParaAdivinhar()`. Lembre-se de checar se nosso `watch` está sempre em execução. Isso nos causará um erro na rota, mas não se preocupe.

Precisamos agora implementar o método que selecionará a palavra dentre as recuperadas e que fará uso do método `_registrarPalavraParaAdivinhar()`. Temos este método na sequência e observações após a listagem. Ele deve também ser implementado em nosso store.

```
selecionarPalavraParaAdivinhar() async {
    if (this._palavrasRegistradas.length == 0)
      this._palavrasRegistradas = await _carregarPalavras();

    var random = new Random();
    int indiceSorteado = random.nextInt(this._palavrasRegistradas.length);
    PalavraModel palavraSelecionada =
this._palavrasRegistradas[indiceSorteado];

    _registrarPalavraParaAdivinhar(
        palavra: palavraSelecionada.palavra, ajuda:
palavraSelecionada.ajuda);
```

```
    this._palavrasRegistradas.removeAt(indiceSorteado);
}
```

Esse método é também assíncrono, porém público, e será ele que invocaremos em nosso jogo, pois mudaremos o que temos atualmente para isso. Veja que verificamos se existem ainda palavras registradas em nossa variável e, caso não exista, uma nova carga é realizada. Isso nos dá a possibilidade de o jogador brincar com todas as palavras registradas, assim repetições só ocorrem após todas serem usadas.

Na sequência, utilizamos `Random()` para sortear um número no intervalo de zero ao tamanho máximo de nossa lista. Precisamos importar `math` aqui. Com o índice retornado, podemos selecionar a palavra, como fazemos na linha seguinte.

Com a palavra registrada, invocamos o método de registro de palavras que já temos, mas o tornamos privado. Após o registro, tiramos de nossa variável de palavras o índice selecionado, aplicando a lógica comentada de utilizarmos todas as palavras da base de dados.

Precisamos fazer apenas uma mudança em nosso método `_registrarPalavraParaAdivinhar()`, que é adicionar `.toUpperCase()` à `palavra` atribuída à variável `this.palavraParaAdivinhar`. Veja esta mudança na sequência.

```
_registrarPalavraParaAdivinhar({String palavra, String ajuda}) {
    this.palavraParaAdivinhar = palavra.toUpperCase();
    ...
}
```

Agora, em nosso route, para que o erro comentado anteriormente possa ser corrigido e nossa aplicação executada corretamente, vamos adaptar o `onPressed` de nosso botão para o código a seguir.

```
onPressed: () => this._jogoStore.selecionarPalavraParaAdivinhar(),
```

Transformação da palavra para adivinhar

Já temos a ajuda para o jogador adivinhar a palavra que está sendo exibida, mas não temos a palavra a ser adivinhada devidamente transformada, ou seja, com os sublinhados aparecendo de acordo com a palavra. Vamos trabalhar nisso agora. Em nosso `jogo_store.dart`, precisamos criar uma propriedade que terá nossa palavra a ser adivinhada formatada com os sublinhados. Veja-a na sequência. Note que a inicializamos com um literal vazio.

```
String palavraAdivinhada = '';
```

Vamos agora criar o método, que será privado, responsável pela geração de espaços sublinhados para cada letra da palavra a ser adivinhada. Veja-o na sequência, que deve também estar em nosso store, com observações após a listagem.

```
_transformarPalavraParaAdivinhar() {
    String palavraFormatada = '';
    for (int i = 0; i < this.palavraParaAdivinhar.length; i++) {
      if (this.palavraParaAdivinhar[i] != ' ')
        palavraFormatada = palavraFormatada + '_';
      else
        palavraFormatada = palavraFormatada + ' ';
    }
    return palavraFormatada;
}
```

Logo no início do método temos a inicialização de uma variável string vazia. Após isso, temos um laço. Veja que esse laço percorre o tamanho da palavra a ser adivinhada. Ele serve para construir o valor de nossa `palavraFormatada`. Caso haja em cada posição da `palavraParaAdivinhar` um valor diferente de espaço em branco, um sublinhado será utilizado e, caso haja o espaço em branco, apenas repetimos o espaço.

Se você quiser, há ainda uma maneira menos verbosa para o método anterior, que seria fazendo o uso de expressões regulares adaptando o corpo do método para:

```
return palavra.replaceAll(RegExp('[A-Za-zÀ-ÖØ-öø-ÿ0-9]'), '_');
```

Precisamos utilizar esse método para transformar a palavra adivinhada e atribuí-la à propriedade `palavraAdivinhada`. O primeiro momento para fazermos isso é no registro da palavra selecionada, ou seja, em nosso método `_registrarPalavraParaAdivinhar()`. Sendo assim, antes do término desse método, insira a instrução a seguir.

```
this.palavraAdivinhada = _transformarPalavraParaAdivinhar();
```

Se você optou pela sugestão anterior para expressões regulares, poderá também substituir a instrução anterior pela seguinte, deixando de existir o método `_transformarPalavraParaAdivinhar()`.

```
this.palavraAdivinhada = this.palavraParaAdivinhar.replaceAll(RegExp('[A-Za-zÀ-ÖØ-öø-ÿ0-9]'), '_');
```

Temos nossa palavra para adivinhar devidamente formatada, mas seria interessante que tivéssemos espaços entre cada letra e que, no caso de espaços em branco, que tivéssemos espaçamento duplo.

Vamos fazer isso, mas precisaremos de uma nova propriedade, agora observável, que terá nossa palavra a ser adivinhada formatada da maneira que deve ser exibida ao jogador. Um novo método precisa ser implementado para isso. Veja as duas situações na sequência, precedidas pela propriedade observável `palavraAdivinhadaFormatada` em nosso store.

```
@observable
String palavraAdivinhadaFormatada = '';

_palavraAdivinhadaFormatada() {
    String palavraFormatada = '';
    for (int i = 0; i < this.palavraAdivinhada.length; i++) {
      palavraFormatada = palavraFormatada + this.palavraAdivinhada[i] + '
';
    }
    return palavraFormatada;
}
```

Uma vez mais, precisamos retomar ao método `_registrarPalavraParaAdivinhar()` para que o registro da nossa palavra formatada seja exibida no jogo para o usuário. Implemente a instrução a seguir ao final do método em questão.

```
this.palavraAdivinhadaFormatada = _palavraAdivinhadaFormatada();
```

Agora precisamos adaptar nosso `jogo_route.dart` para que ele renderize corretamente nossa palavra a ser adivinhada. Veja a adaptação na sequência e, após a implementação, teste o jogo novamente. Pressione várias vezes o botão. Lembre-se de que a variação das palavras depende da quantidade que você registrou.

```
palavraParaAdivinhar(palavra: this._jogoStore.palavraAdivinhadaFormatada),
```

10.6 O teclado do jogo com as letras

Já temos nosso teclado desenhado em nossa visão, mas precisamos trabalhar a interação do usuário com ele. Quando essa interação ocorrer, precisaremos verificar se a letra pressionada existe na palavra adivinhada e, se não existir, precisamos trabalhar a etapa do jogo, mudando a animação. Ainda, caso a letra exista na palavra, precisamos atualizar a palavra com sublinhados para mostrar todas as ocorrências da letra nela. Também precisaremos mudar o estilo da letra para ficar visível ao usuário quais ele já pressionou.

O widget que representará cada letra

Como esta parte de nossa visão terá uma certa complexidade, tanto na interação com ela como na manutenção de seu estado, vamos criar widgets com essas responsabilidades. A primeira situação nos remete a cada letra exibida, que, além de seu símbolo visual, precisamos saber se já foi ou não pressionada. Sendo assim, na pasta `/routes/jogo/`, vamos criar uma nova chamada `widgets`. Nela,

criaremos o arquivo `letra_teclado_jogo_widget.dart`. Veja na listagem a seguir o código para ele.

```
import 'package:flutter/material.dart';

class LetraTecladoJogoWidget extends StatefulWidget {
  final String letra;
  final bool foiUtilizada;

  const LetraTecladoJogoWidget({this.letra, this.foiUtilizada = false});

  @override
  _LetraTecladoJogoWidgetState createState() =>
_LetraTecladoJogoWidgetState();
}

class _LetraTecladoJogoWidgetState extends State<LetraTecladoJogoWidget> {
  @override
  Widget build(BuildContext context) {
    return Text(
      widget.letra,
      style: TextStyle(
        color: widget.foiUtilizada ? Colors.red : Colors.black,
        fontSize: 40,
      ),
    );
  }
}
```

Podemos notar que não temos nada de muito complexo. Apenas um Stateful com duas propriedades que são utilizadas em sua renderização.

Para implementarmos essa nova funcionalidade, algumas alterações serão necessárias em nosso código e vamos fazer isso agora. A primeira é declararmos algumas variáveis em nosso widget anterior na classe `_TecladoJogoWidgetState`. Veja-as a seguir. Atente-se ao import.

```
String letrasParaTeclado = 'ABCDEFGHIJKLMNOPQRSTUWXYZ';
List<LetraTecladoJogoWidget> widgetsDeLetrasDoTeclado =
List<LetraTecladoJogoWidget>();
```

A primeira variável, `letrasParaTeclado`, será responsável por guardar os símbolos a serem desenhados no teclado. Se você lembrar, estávamos enviando diretamente esse valor para um método. Isso agora mudará.

Já a segunda, `widgetsDeLetrasDoTeclado`, será responsável por armazenar os widgets de nossas teclas, os quais serão renderizados no teclado. Precisaremos trabalhar os estados deles, então precisamos tê-los no contexto.

Com as declarações realizadas, vamos trabalhar em nosso `initState()`, populando nossa matriz de widgets. Veja o código a seguir e o implemente antes do final do método. Veja que não temos nada de outro mundo, apenas adicionamos widgets em nossa variável, um para cada letra a ser representada por teclas.

```
for (int i = 0; i < letrasParaTeclado.length; i++) {
  widgetsDeLetrasDoTeclado.add(
    LetraTecladoJogoWidget(
      letra: this.letrasParaTeclado[i],
    ),
  );
}
```

Agora precisamos ajustar nosso método `letrasParaSelecao()` em nosso `jogo_mixin.dart`, pois ele precisará se ajustar à nova situação. Veja-o na sequência. A parte referente ao `Wrap` continua a mesma, por isso a omiti do código. Note que mudamos o argumento que chega no método. Temos algumas observações após a listagem. Você precisará de um import.

```
letrasParaSelecao({List<LetraTecladoJogoWidget> letras}) {
    List<Widget> textsParaLetras = List<Widget>();

    for (int i = 0; i < letras.length; i++) {
```

```
      textsParaLetras.add(
        InkWell(
          child: letras[i],
          onTap: () => print('Letra ${letras[i].letra} foi pressionada'),
        ),
      );
    }

    ...
}
```

Em sua leitura, você observou que estamos utilizando o `InkWell`? Pois é. Ele é outro componente para capturar gestos do usuário em seu app. Porém, ao contrário do `GestureDetector`, ele aplica recursos visuais definidos no Material Design. Fica a dica para o uso de animações e efeitos.

Retomando ao código, no evento `onTap`, para cada widget inserido em nossa variável `textsParaLetras` temos apenas a exibição em console de um texto. Logo trabalharemos isso melhor.

Finalizando esses ajustes, vamos melhorar o código que exibe nossas teclas em nosso `jogo_route.dart`. Veja a nova invocação na sequência. Aqui, recomendo até uma refatoração no nome desse método para `exibirTecladoParaJogo`, o que acha? Fica a seu critério.

```
letrasParaSelecao(letras: this.widgetsDeLetrasDoTeclado),
```

O widget do teclado virtual

A maneira como temos a exibição do teclado está funcionando. Para o problema que temos até aqui, não precisaríamos mexer. Entretanto, com vistas a possibilidades de personalização para o teclado e buscando coesão e acoplamento corretos, vamos também criar um widget para ele. Na pasta `/routes/jogo/widgets`, vamos criar um arquivo de nome `teclado_jogo_widget.dart`. Veja-o na sequência.

```
import 'package:flutter/material.dart';
```

```
class TecladoJogoWidget extends StatefulWidget {
  final List<Widget> textsParaLetras;

  const TecladoJogoWidget({this.textsParaLetras});

  @override
  _TecladoJogoWidgetState createState() => _TecladoJogoWidgetState();
}

class _TecladoJogoWidgetState extends State<TecladoJogoWidget> {
  @override
  Widget build(BuildContext context) {
    return Wrap(
      alignment: WrapAlignment.center,
      spacing: 20,
      runSpacing: 5,
      children: widget.textsParaLetras,
    );
  }
}
```

Observou que não temos nada que necessite de muita explicação? Temos uma propriedade a ser recebida pelo widget e então a renderizamos, tal qual fazemos em nosso mixin, que inclusive precisamos alterar. Veja o código a seguir, já com o nome do método refatorado, como dito anteriormente. Alterei apenas a parte do `Wrap` para nosso novo widget. Atente-se ao import.

```
exibirTecladoParaJogo({List<LetraTecladoJogoWidget> letras}) {
    ...

    return Padding(
      padding: const EdgeInsets.only(left: 10, right: 10, bottom: 10.0),
      child: TecladoJogoWidget(
        textsParaLetras: textsParaLetras,
      ),
    );
}
```

Nós teremos aqui um processo de refatoração muito maior e veremos tudo isso na próxima seção, pois precisaremos nos preocupar com a interação do usuário com as letras e a atualização dos estados delas, o que mudará suas características visuais e comportamentais. Trabalharemos com o MobX novamente, mas precisamos preparar nossos artefatos para isso.

Refatoração do widget de teclado

Nosso primeiro passo nesse processo será retirar de nosso `jogo_route.dart` o que é de responsabilidade do teclado. Começaremos com as variáveis `letrasParaTeclado` e `widgetsDeLetrasDoTeclado`. Levaremos a primeira como está para nosso widget específico do teclado. Apenas a recorte de um arquivo e cole no outro. Não se preocupe com os erros que forem surgindo, vamos ajustar todos eles.

Moveremos a segunda variável, que é a `widgetsDeLetrasDoTeclado`, para um store que criaremos na sequência, pois precisaremos monitorar as alterações realizadas em objetos que estarão no `list` dessa variável. Sendo assim, na pasta `/routes/jogo/mobx_stores`, crie o arquivo `teclado_store.dart` e nele insira o código inicial a seguir.

```
import 'package:cc04/routes/jogo/widgets/letra_teclado_jogo_widget.dart';
import 'package:mobx/mobx.dart';

part 'teclado_store.g.dart';

class TecladoStore = _TecladoStore with _$TecladoStore;

abstract class _TecladoStore with Store {
  @observable
  ObservableList<LetraTecladoJogoWidget> widgetsDeLetrasDoTeclado =
ObservableList<LetraTecladoJogoWidget>();
}
```

Notou que nossa propriedade observável agora é um `ObservableList`? Isso é muito legal, pois podemos ter toda uma coleção em observação. Logo veremos isso funcionando.

Na sequência, retiraremos do `initState()` de nosso `jogo_route.dart` o código referente ao laço que popula a coleção `widgetsDeLetrasDoTeclado`, já retirada. Nós utilizaremos essa lógica em nosso store, sendo assim, no código anterior, insira o que está na sequência.

```
@action
inicializarTeclado({String letrasParaTeclado}) {
    for (int i = 0; i < letrasParaTeclado.length; i++) {
      widgetsDeLetrasDoTeclado.add(
        LetraTecladoJogoWidget(
          letra: letrasParaTeclado[i],
        ),
      );
}
```

Essa action será invocada sempre que nossa visão do jogo for inicializada, o que nos leva a algumas implementações em `jogo_route.dart`. A primeira é a declaração de nosso store em nosso widget do teclado, que, como pode ser vista na sequência, deve ser inserida logo após a declaração das letras. Aqui trabalharemos daquela maneira que comentei quando criamos o primeiro store. Você pode avaliar o que for mais interessante para seu projeto. Atente-se ao import.

```
TecladoStore _tecladoStore;
```

A segunda implementação refere-se ao `initState()` de nosso widget de teclado. Veja o código para ele na sequência.

```
@override
void initState() {
    super.initState();
    _tecladoStore = TecladoStore();
    _tecladoStore.inicializarTeclado(letrasParaTeclado:
```

```
letrasParaTeclado);
}
```

Com isso tudo implementado, precisamos adaptar a renderização de nosso teclado. Em nosso widget, vamos criar um método que será responsável por essa renderização. Veja-o a seguir e alguns comentários após a listagem.

```
_gerarTeclado() {
    var teclado = List<Widget>();
    for (int i = 0; i < _tecladoStore.widgetsDeLetrasDoTeclado.length;
i++) {
      teclado.add(InkWell(
        onTap: () {
          print('Letra Pressionada');
        },
        child: _tecladoStore.widgetsDeLetrasDoTeclado[i],
      ));
    }
    return teclado;
}
```

O método começa com uma declaração para um `List`. Em seguida, um laço é implementado para percorrer a coleção de letras que temos no store. Para cada letra, adicionamos um `InkWell` que a encapsula, pois precisaremos trabalhar a interação do usuário com essas letras. Observe que, caso haja a interação, no momento estamos apenas exibindo uma mensagem na console e, como filho do `InkWell`, temos a letra para a posição de nosso `List`.

Precisamos agora adaptar nosso `Wrap` para que ele tenha em sua propriedade `children` o resultado do método que acabamos de implementar. Veja essa alteração na sequência.

```
children: _gerarTeclado(_tecladoStore.widgetsDeLetrasDoTeclado),
```

Para finalizarmos e testarmos nosso app, precisamos trocar a invocação ao método `exibirTecladoParaJogo()` em nosso `jogo_route`, que pode ser retirado de nosso `jogo_mixin` pela invocação a nosso widget. Veja isso a seguir. Lembre-se do import para ele.

```
TecladoJogoWidget(),
```

Com tudo isso implementado, podemos executar novamente nosso app e ver que o teclado está sendo renderizado sem nenhum problema.

Registro de letra já utilizada

Uma das etapas após a interação de nosso usuário com o teclado é registrar que uma letra já foi utilizada e não permitir que ele interaja novamente com ela. Você verá que isso é simples. Teremos novamente algumas alterações e novas implementações, mas todas bem básicas com o conhecimento que já temos.

Dessa maneira, em nosso store (do teclado), precisamos criar um método que registre o pressionamento da letra que sofreu a interação do usuário. Vamos criá-lo.

```
@action
letraPressionada({int indiceDaLetra}) {
    widgetsDeLetrasDoTeclado[indiceDaLetra] = LetraTecladoJogoWidget(
      letra: widgetsDeLetrasDoTeclado[indiceDaLetra].letra,
      foiUtilizada: true,
    );
}
```

Você notou que estamos substituindo o objeto que temos na posição da letra pressionada por uma nova instância de `LetraTecladoJogoWidget` agora com o flag `foiUtilizada` em `true`? Pois é. Para que nosso `ObservableList` possa entender que um elemento seu foi manipulado, é preciso trocar o valor dele. Não bastaria aqui apenas trocar o valor da propriedade em específico de cada elemento do `List`.

Quando precisarmos que um objeto que já temos tenha variações apenas em algumas propriedades, mantendo os valores das demais, podemos usar uma convenção de Dart. Isso é interessante, pois você lembra que as propriedades de nossos objetos

`LetraTecladoJogoWidget` são `final`, certo? Isso não nos permite alterar seu valor. Foi uma escolha nossa, poderia ser diferente.

Vamos então adaptar nossa classe `LetraTecladoJogoWidget` para que possua um método chamado `copyWith()`, que é padrão para o problema que temos. Veja-o na sequência. Ele é implementado na classe `StatefulWidget` e não na de `State`, ok?

```
LetraTecladoJogoWidget copyWith({String letra, bool foiUtilizada}) {
    return LetraTecladoJogoWidget(
      letra: letra ?? this.letra,
      foiUtilizada: foiUtilizada ?? this.foiUtilizada,
    );
}
```

No código anterior, verifique que o método recebe como argumento valores opcionais, tal como nossa classe, e o retorno é uma instância da própria classe. Essa nova instância usará os novos valores quando eles existirem nos argumentos de chegada, caso contrário, utilizará o atual da classe. É uma estratégia bem legal para clonagem de objetos e é um padrão em Dart/Flutter.

Precisamos agora atualizar nossa action `letraPressionada()` do store do teclado para usarmos essa nova estratégia e podemos ver isso no código a seguir.

```
@action
letraPressionada({int indiceDaLetra}) {
    widgetsDeLetrasDoTeclado[indiceDaLetra] =
        widgetsDeLetrasDoTeclado[indiceDaLetra].copyWith(
      letra: widgetsDeLetrasDoTeclado[indiceDaLetra].letra,
      foiUtilizada: true,
    );
}
```

Muito bem, com este método implementado, precisamos consumi-lo e faremos isso lá no método `_gerarTeclado()` em nosso widget do teclado. No `onTap` do `InkWell`, substitua o `print` que deixamos pelo código a seguir.

```
onTap: () => _tecladoStore.letraPressionada(indiceDaLetra: i),
```

Vamos agora executar nosso jogo e ver o que acontece com as letras que sofrem interação com o usuário? Elas ficam vermelhas, não é? E, além disso, não permitem mais interação. Não mudamos nada no visual para isso, mas semanticamente, o jogador saberá que já utilizou essa letra destacada. Veja a figura a seguir com esse destaque. Não se preocupe se a letra ainda não tiver a cor alterada. Logo veremos isso.

Figura 10.6: Teclado com letras já pressionadas

Fica aqui uma dica muito simples para resolver esse problema, o que evitaria a criação um novo objeto para associá-lo novamente à lista observável. Poderíamos criar um store do item, que será o

genérico da lista, e manipular alterações nesse elemento. Dá um pouco mais de trabalho, mas é o mais recomendado. Vou deixar essa dica para você fazer, pois o livro já está extenso. Se precisar de ajuda, terei o maior prazer em ajudar.

Conclusão

Terminamos a primeira parte relacionada ao objetivo apresentado no início do capítulo.

O que você tem a dizer em relação à gestão de estados comparando o BLoC e o MobX?

Apenas para enumerar o que vimos neste capítulo, que neste momento foram apenas widgets, trago aqui a relação deles, o que pode lhe auxiliar, caso queira, em uma pesquisa futura, específica para cada ponto trabalhado. São eles: `BorderRadius`, `BoxShadow`, `Flare`, `GetIt`, `InkWell`, `MobX`, `PlaceHolder`, `FlatButton`, `Offset`, `Random`, `Wrap` e `Visibility`.

Sugiro que dê uma respirada, tome uma água e, após o relaxamento, retorne para o próximo capítulo, no qual finalizaremos o nosso jogo.

CAPÍTULO 11
Validação da letra escolhida e verificação de vitória e derrota

No capítulo anterior, começamos nossas implementações para a finalização do nosso jogo. Precisamos agora concluir essa atividade, começando pelo registro de acerto ou erro na letra selecionada.

11.1 Verificação da existência da letra

Estamos próximos de concluir nosso jogo, resta pouco. Uma pendência é verificar se a letra pressionada faz parte da palavra a ser descoberta. Caso a letra exista, precisamos atualizar sua exibição no jogo. Caso contrário, a animação da forca precisa mudar.

Vamos começar esta etapa com a implementação de uma action em nosso `jogo_store`, que será responsável pela verificação de existência da letra escolhida na palavra a ser adivinhada. Veja o código dela na sequência e, após a listagem, algumas explicações.

```
@action
verificarExistenciaDaLetraNaPalavraParaAdivinhar({String letra}) {
    int indexOfWord = this.palavraParaAdivinhar.indexOf(letra, 0);
    if (indexOfWord < 0) {
      return;
    }

    while (indexOfWord >= 0) {
      this.palavraAdivinhada =
          this.palavraAdivinhada.replaceFirst('_', letra, indexOfWord);

      indexOfWord = this.palavraParaAdivinhar.indexOf(letra, (indexOfWord
+ 1));
```

```
    }

  this.palavraAdivinhadaFormatada = _palavraAdivinhadaFormatada();
}
```

Nosso método começa com a declaração e inicialização da variável `indexOfWord` com uma possível posição da letra dentro da palavra. Em seguida, verificamos o valor para essa variável. Se for negativo, a letra não existe e, para este momento, apenas encerramos o método com `return`.

Caso a letra exista, iniciamos um laço com `while()` e, dentro dele, substituímos o `_` pela letra na posição identificada. Após essa operação, realizamos uma nova pesquisa pela letra na palavra, mas apenas nas letras após a posição que já utilizamos. Com isso, o `while()` avalia novamente a continuidade ou não dessa lógica.

Finalizando o método, temos a atribuição da palavra adivinhada à variável `palavraAdivinhadaFormatada` já com os espaços entre ela.

Precisamos agora invocar esse método e o faremos no método `_gerarTeclado()` de nosso `teclado_jogo_widget`. Basta inserir a invocação a ele após o registro da letra pressionada no `onTap` de `InkWell`. Veja a adaptação na sequência, que também muda como o corpo do método é implementado. Isso resolverá o problema da atualização de cor para as letras comentado anteriormente.

```
onTap: () {
  _tecladoStore.letraPressionada(indiceDaLetra: i);
  _jogoStore.verificarExistenciaDaLetraNaPalavraParaAdivinhar(
      letra: _tecladoStore.widgetsDeLetrasDoTeclado[i].letra);
},
```

Deu erro na invocação do método, não foi? Pois é. Não temos o `_jogoStore` em nosso widget. Mas isso é fácil de resolver. Vamos declarar o código a seguir, junto com as outras variáveis.

```
JogoStore _jogoStore;
```

Agora, precisamos recuperar nosso store no contexto da aplicação como fizemos no `jogo_route`. Implemente a instrução a seguir no `initState()` de nosso widget. Realize os imports necessários.

```
_jogoStore = getIt.get<JogoStore>();
```

Vamos testar? Pressione o botão para uma palavra ser sorteada e escolha as letras. Elas ficam vermelhas e a palavra adivinhada começa a exibir a letra se você acertou, correto?

Ainda temos um problema em relação ao `onTap`. Se você colocar o `print()` com qualquer conteúdo dentro dele, verá que mesmo com a letra em vermelho ele é processado. Precisamos garantir que isso não ocorra. Vamos então adaptá-lo tal qual o código a seguir.

```
onTap: (!_tecladoStore.widgetsDeLetrasDoTeclado[i].foiUtilizada)
    ? () {
        _tecladoStore.letraPressionada(indiceDaLetra: i);
        _jogoStore.verificarExistenciaDaLetraNaPalavraParaAdivinhar(
            letra: _tecladoStore.widgetsDeLetrasDoTeclado[i].letra);
      }
    : null,
```

Identificou o operador ternário atuando sobre a propriedade `foiUtilizada`? Muito bem. Com isso, o `onTap` só será executado uma vez para cada letra.

11.2 Errou, começa a animação da forca

Chegamos à segunda pendência para nosso jogo, o momento em que o usuário errou a letra pressionada, ou seja, ela não existe na palavra. Nós já temos implementado o local onde isso é verificado, lembra? Resta-nos apenas trabalhar a animação. Você verá que isso é bem simples. Vamos começar com duas variáveis a serem declaradas em nosso `jogo_store`. Veja-as na sequência.

```
int quantidadeErros = 0;

@observable
String animacaoFlare = 'idle';
```

Notou o nome semântico para as variáveis? É uma recomendação `Clean Code`. Fica a dica, pois vale a pena estudar isso. Mas vamos trabalhar agora a lógica dos erros.

Com base na quantidade de erros, precisamos trabalhar qual animação deverá ser renderizada. Vamos criar um método específico para isso em nosso store para o jogo. Veja-o na sequência.

```
@action
registrarErro() {
    quantidadeErros++;
    if (this.quantidadeErros == 1)
      this.animacaoFlare = 'cadeira';
    else if (this.quantidadeErros == 2)
      this.animacaoFlare = 'corpo';
    else if (this.quantidadeErros == 3)
      this.animacaoFlare = 'cabeca';
    else if (this.quantidadeErros == 4)
      this.animacaoFlare = 'balanco';
    else if (this.quantidadeErros == 5)
      this.animacaoFlare = 'enforcamento';
}
```

Agora, lá no método `verificarExistenciaDaLetraNaPalavraParaAdivinhar()`, ainda no store do jogo, vamos ajustar o código para quando houver o erro. Trago todo o `if` na sequência para auxiliar, mas de novo só tem o incremento a invocação para `registrarErro()`, que acabamos de implementar.

```
if (indexOfWord < 0) {
  registrarErro();
  return;
}
```

Falta-nos ajustar a animação do início do jogo, que é o momento de seleção da palavra por meio da interação com o botão. Então, no `jogo_store.dart`, ao final do método `selecionarPalavraParaAdivinhar()`, vamos inserir a seguinte declaração.

```
this.animacaoFlare = 'inicio';
```

Só nos resta adaptar nosso `jogo_route` para fazer uso dessa nossa propriedade. Veja a alteração na sequência.

```
animacaoDaForca(animacao: this._jogoStore.animacaoFlare),
```

Vamos ver isso em funcionamento? Jogue, ganhe e perca. Ao perder, verifique que toda a animação é atualizada. Legal, não é? Mas ainda temos alguns retoques a fazer em nosso jogo e os faremos na próxima seção.

11.3 Fechamentos para concluir o jogo

Nosso jogo está funcionando, está bonito, mas temos alguns pontos que precisamos ajustar em nossa visão e vamos vê-los todos nesta seção.

O primeiro deles é que, se você notar a figura a seguir, estamos com espaços reservados para a palavra adivinhada e para a ajuda para a palavra, o que deixa menos espaço para a animação da forca no início do jogo. Vamos trabalhar um pouco isso.

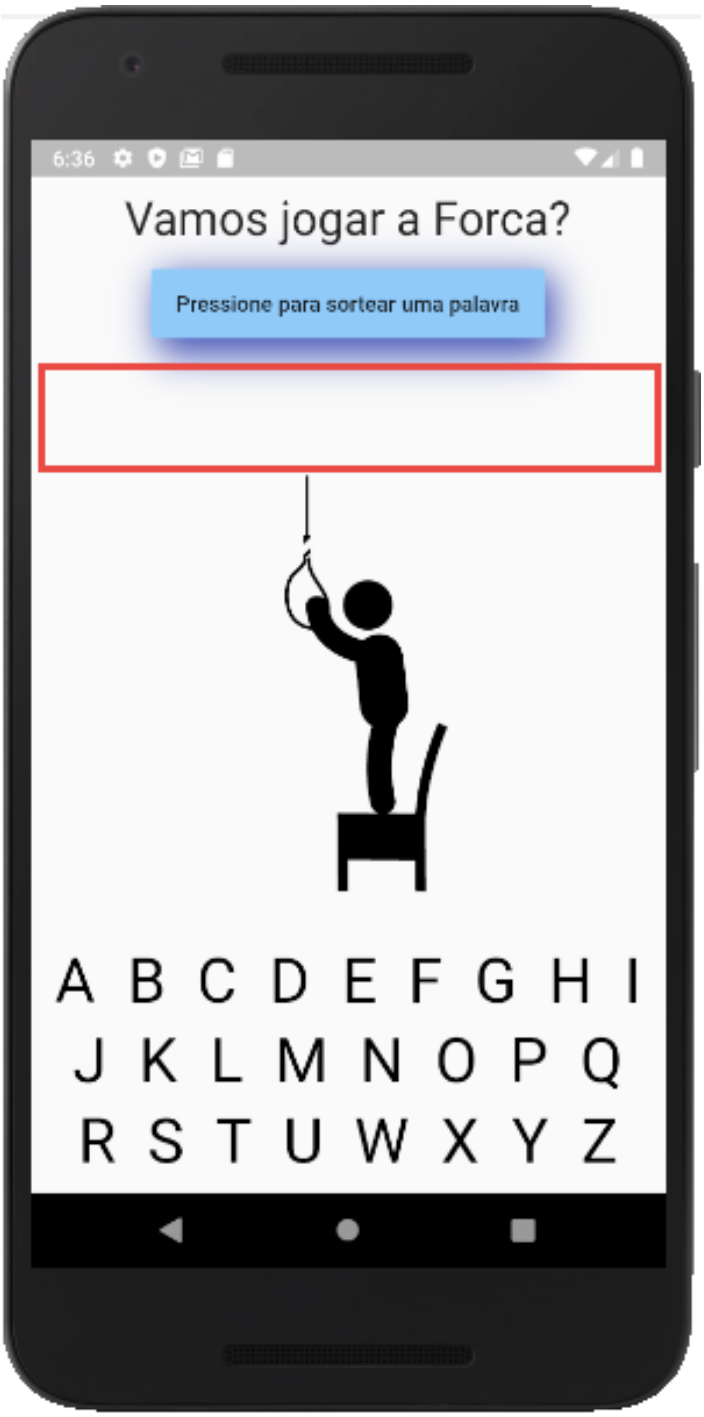

Figura 11.1: Espaços reservados no início do jogo

Lembra do `Visibility`? Pois é, vamos usá-lo. Adapte nosso `jogo_route.dart` na renderização da palavra adivinhada e da ajuda de acordo com o apresentado na sequência. Não temos nada em

especial ou dificultoso que precise de maiores explicações no código.

```
Visibility(
  visible:
      this._jogoStore.palavraAdivinhadaFormatada.isNotEmpty,
  child: palavraParaAdivinhar(
      palavra: this._jogoStore.palavraAdivinhadaFormatada),
),
Visibility(
  visible: this._jogoStore.palavraAdivinhadaFormatada.isNotEmpty,
  child: ajudaParaAdivinharAPalavra(
      ajuda: this._jogoStore.ajudaPalavraParaAdivinhar),
),
```

Execute o app e veja que com isso resolvemos o problema. Agora, temos outro: o momento em que o usuário pressiona o botão para iniciar o jogo e o botão continua aparecendo. Precisamos ocultar o botão e o título antes do botão, pois já começamos o jogo neste momento e não precisamos dele após o início do jogo. Vamos adaptar o código de sua renderização para o que temos na sequência. Uma vez mais, execute o app. Veja a nova renderização com essas mudanças.

```
Visibility(
  visible: this._jogoStore.palavraAdivinhadaFormatada.isEmpty,
  child: titulo(),
),
Visibility(
  visible: this._jogoStore.palavraAdivinhadaFormatada.isEmpty,
  child: botaoParaSorteioDePalavra(
    onPressed: () =>
        this._jogoStore.selecionarPalavraParaAdivinhar(),
  ),
),
```

11.4 O jogador ganhou

Outra situação que temos que resolver é o momento da vitória do jogador, pois nada acontece ainda. Precisamos dar a ele o sentimento de vitória. Poderíamos pensar em animação aqui também, mas para minimizar nosso esforço, vamos apenas exibir uma mensagem de vitória, ficando para você a sugestão de usar animação.

Em meu caso, na pasta `/assets/images`, criei outra, chamada `jogo` e, dentro dela, coloquei um arquivo de imagem qualquer, que chamei de `vitoria.jpg`. Para lembrar, tive que ajustar o `pubspec.yaml` na parte de `assets` para aceitar a nova pasta.

Em seguida, em nosso `jogo_store`, adicionei outra propriedade observável, que especificará quando o jogador ganhou. Ela está na sequência.

```
@observable
bool ganhou = false;
```

Como você pôde notar, a variável terá, por padrão, o valor `false`, mas precisamos ajustá-lo para `true` quando o jogador tiver adivinhado toda a palavra. Isso ocorre em nosso método `verificarExistenciaDaLetraNaPalavraParaAdivinhar()`. Veja na sequência a implementação necessária ao final dele.

```
if (this.palavraAdivinhada.indexOf('_', 0) < 0)
    this.ganhou = true;
```

Viu que, caso não tenhamos mais sublinhados, significa que o o jogador ganhou? Pois bem, precisamos agora trabalhar isso em nosso `jogo_route` no momento da vitória do jogador. Mas, antes disso, vamos criar um widget que será responsável por exibir a imagem de vitória quando ela ocorrer.

Na pasta `/routes/jogo/widgets`, crie um arquivo chamado `vitoria_widget.dart` e nele coloque o código a seguir. Note que o que trazemos é praticamente tudo o que já trabalhamos, mas, para efeito de resultado, quando este widget for renderizado, ele exibirá

uma imagem em toda a tela do dispositivo e, na base, uma mensagem em um contêiner colorido com sombras alertando que o jogo logo recomeçará.

```
import 'package:flutter/material.dart';

class VitoriaWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Stack(
      children: <Widget>[
        Container(
          decoration: BoxDecoration(
            image: DecorationImage(
                image: AssetImage("assets/images/jogo/vitoria.jpg"),
                fit: BoxFit.cover),
          ),
        ),
        Align(
          alignment: Alignment.bottomCenter,
          child: Padding(
            padding: const EdgeInsets.all(8.0),
            child: Container(
              height: 100,
              decoration: BoxDecoration(
                  color: Colors.green,
                  borderRadius: BorderRadius.all(Radius.circular(5)),
                  boxShadow: [
                    BoxShadow(
                      blurRadius: 5,
                      color: Colors.white,
                      spreadRadius: 5,
                      offset: Offset(0.1, 0.1),
                    )
                  ]),
              child: Center(
                child: Text(
                  'Parabéns pela vitória. Já retornaremos ao jogo.',
                  textAlign: TextAlign.center,
                  style: TextStyle(fontSize: 30),
```

```
                ),
              ),
            ),
          ),
        )
      ],
    );
  }
}
```

11.5 Reinício após a vitória

Precisamos agora pensar no que devemos fazer para que, ao retomar para o jogo, ele seja reiniciado. Nosso primeiro registro deve ser atribuir `false` à propriedade `ganhou`. Em seguida, precisamos também limpar nossa `palavraAdivinhadaFormatada`, pois ela tem sido nosso flag para exibir ou não certos componentes de nossa visão de jogo. Um último ajuste é reiniciarmos a animação. Tudo isso precisa ser feito após um intervalo de tempo para que a imagem renderizada seja omitida e o jogo retorne ao início. Veja que o código a seguir deve ser inserido no início de nosso `build()`, antes do `return`, nesse novo componente que criamos. Atente-se aos imports.

```
Future.delayed(Duration(seconds: 5)).then((_) {
  getIt.get<JogoStore>().ganhou = false;
  getIt.get<JogoStore>().palavraAdivinhadaFormatada = '';
  getIt.get<JogoStore>().animacaoFlare = 'idle';
  getIt.get<JogoStore>().quantidadeErros = 0;
});
```

Note que estamos usando um temporizador, que, após a passagem de 5 segundos, executará as instruções em seu corpo. Vamos ver a visão da vitória? Veja-a na sequência, mas ainda não a teremos se formos testar.

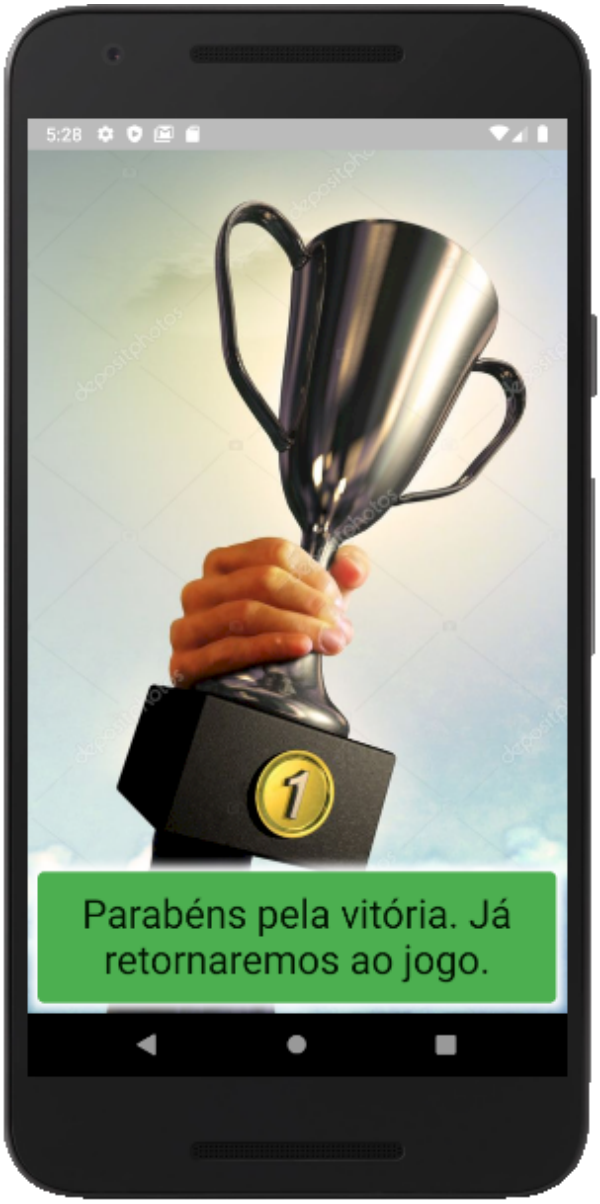

Figura 11.2: Jogador ganhou

Vamos então fazer com que nosso widget de vitória seja exibido. No início de nosso `Observer()`, no `build()` dele, no `jogo_route`, precisamos inserir o código a seguir. Observe que, caso haja vitória, em vez de renderizarmos a visão do jogo, renderizamos a vitória, que vimos há pouco. Atente-se aos imports.

```
if (this._jogoStore.ganhou) {
  return VitoriaWidget();
}
```

11.6 O jogador perdeu

Precisamos gerar algo semelhante para quando o jogador perder e seu personagem for enforcado. Poderíamos pensar em criar um novo widget para derrota, mas o comportamento que desejamos é o mesmo, então vamos refatorar o que criamos e reutilizá-lo. Vamos começar declarando um flag para a derrota, logo depois do que implementamos para a vitória, no `jogo_store`. Veja o código na sequência.

```
@observable
bool perdeu = false;
```

Agora, precisamos alterar o valor dessa propriedade observável e faremos isso quando atualizarmos a animação para a última fase dela no método `registrarErro()`, também no `jogo_store`. Veja na sequência o trecho que precisamos adaptar. Nele, alteramos a animação e, cinco segundos depois, registramos a derrota.

```
else if (this.quantidadeErros == 5) {
  this.animacaoFlare = 'enforcamento';
  Future.delayed(Duration(seconds: 5)).then((_) {
    this.perdeu = true;
  });
}
```

Vamos começar nossa refatoração e você verá a reutilização em prática. Vamos renomear o arquivo `vitoria_widget.dart` para `jogo_terminou_widget.dart` e a classe dele `VitoriaWidget` para `JogoTerminouWidget`. Lembre-se de usar recursos do Android Studio para renomear os artefatos, é mais seguro e rápido.

Precisamos agora injetar no widget a URL da imagem e o texto a ser exibido. Veja na sequência a mudança na declaração das propriedades e do construtor para a classe que o implementa.

```
final String urlImagem;
final String mensagem;
```

```
const JogoTerminouWidget({this.urlImagem, this.mensagem});
```

Nesse widget, temos a reinicialização do flag para o caso de o jogador ganhar, mas não temos para o caso de ele perder. Dessa maneira, em nosso `Future.delayed()`, logo no início de `build()`, insira a instrução a seguir abaixo do `acertou`.

```
getIt.get<JogoStore>().perdeu = false;
```

Para concluirmos essa refatoração, precisamos apenas usar essas propriedades nos lugares em que tínhamos o valor constante. Veja na sequência os trechos desses códigos.

```
decoration: BoxDecoration(
    image: DecorationImage(
        image: AssetImage(urlImagem), fit: BoxFit.cover),
),
```

e

```
Text(
  mensagem,
  textAlign: TextAlign.center,
  style: TextStyle(fontSize: 30),
),
```

Muito bem. Nosso widget está pronto, vamos agora ajustar a invocação dele em nosso `jogo_route.dart` e temos isso na sequência, substituindo o que tínhamos antes para a vitória. Você certamente verá que esse código poderia estar mais customizado, genérico, mas vou deixar a tarefa para você caso queira. Após o ajuste, o que acha de testar novamente o jogo? Perca e ganhe. Veja que legal.

Como melhoria para a implementação deste código, poderíamos também pensar em um operador ternário, ou ainda isolar em um método exclusivo a responsabilidade de verificar o status da partida, fica essa sugestão para sua verificação.

```
if (this._jogoStore.ganhou) {
  return JogoTerminouWidget(
    urlImagem: "assets/images/jogo/vitoria.jpg",
    mensagem: 'Parabéns pela vitória. Já retornaremos ao jogo.',
  );
} else if (this._jogoStore.perdeu) {
  return JogoTerminouWidget(
    urlImagem: "assets/images/jogo/derrota.jpg",
    mensagem: 'Que pena, você perdeu, mas já retornaremos ao jogo.',
  );
}
```

11.7 Ajuste final para começar o jogo

Nosso jogo começa bem legal, mas porque temos as palavras já persistidas do capítulo anterior. Mas e se o jogador baixar o app da store e já for direto para o jogo? Como não temos nossa base na nuvem, precisamos orientá-lo a registrar essas palavras antes de começar. Faremos isso de uma maneira divertida utilizando um plugin para alertas mais bonitos e que podem ser interessantes. Em nosso `pubspec.yaml`, registre a dependência a seguir e atualize-as com o `pub get`.

```
dependencies:
  giffy_dialog: ^1.7.0
```

Este componente nos permite utilizar gifs animados ou animações Flare. Aqui, faremos uso de um gif, mas a documentação dele é clara no `pub.dev` para usar Flare.

Escolha um gif que queira utilizar e grave-o em seus `assets`. Se for criar uma pasta diferente para ele, lembre-se de registrá-la no `pubspec.yaml`.

Agora vamos adaptar, em nosso arquivo `drawerbodycontent_app.dart`, o método para o `onTap` da opção de jogo. Veja a nova

implementação na sequência. Atenção aos imports que serão necessários. Alguns comentários estão após a listagem. Poderíamos também pensar em uma isolação do código, mas essa melhoria fica a seu critério.

```
onTap: () async {
    try {
      PalavraDAO palavraDAO = PalavraDAO();
      final List data = await palavraDAO.getAll();
      if (data.length == 0) {
        await showDialog(
          context: context,
          builder: (_) => AssetGiffyDialog(
            image: Image.asset(
              "assets/gifs/error.gif",
              fit: BoxFit.cover,
            ),
            title: Text(
              'Não existem palavras registradas',
              textAlign: TextAlign.center,
              style: TextStyle(
                  fontSize: 22.0, fontWeight: FontWeight.w600),
            ),
            description: Text(
              'Para você poder jogar a forca, você precisa antes registrar
algumas palavras',
              textAlign: TextAlign.center,
            ),
            buttonOkText: Text('Ok'),
            onlyOkButton: true,
            entryAnimation: EntryAnimation.BOTTOM_RIGHT,
            onOkButtonPressed: () => Navigator.of(context).pop(),
          ),
        );
      } else {
        Navigator.of(context).pop();
        Navigator.of(context).pushNamed(kJogoRoute);
      }
    } catch (exception) {
      rethrow;
```

```
    }
},
```

A primeira alteração que fizemos foi adicionar `async` à declaração do método, pois teremos uma chamada com `await` nele. Na sequência, para vermos se há ou não palavras registradas, precisamos invocar nosso `DAO`. Você pode notar que poderíamos criar um método específico para essa funcionalidade, mas como este capítulo já está longo, preferi implementar dessa maneira.

Caso não haja palavras, invocamos `showDialog()`, tendo nosso `AssetGiffyDialog()` no `builder`. Existem widgets para `Network` também, assim como para o Flare, como já comentado.

Procure dar uma lida nas propriedades codificadas, elas são autoexplicativas. Temos a imagem, um título, uma descrição, um botão, o comportamento para ele e a animação que o widget utilizará para ser exibido. Este componente é bem divertido. Veja o resultado visual na figura a seguir.

Figura 11.3: Alerta para jogar sem palavras

11.8 Correção de bugs que ficaram

Durante a implementação do jogo, não nos preocupamos com alguns detalhes. Até imagino e espero que você já tenha verificado um deles. Estou falando da visão inicial do jogo, em que o jogador precisa interagir com o botão para que o jogo possa começar. Se você tentar utilizar o teclado, verá que ele aceita suas interações, mas não deveria. Isso só poderia ser possível após o jogo ter iniciado, mas é algo fácil de corrigir. Basta adaptarmos a invocação ao teclado com a implementação a seguir.

```
Visibility(
  visible:
      this._jogoStore.palavraAdivinhadaFormatada.isNotEmpty,
  child: TecladoJogoWidget(),
),
```

Outra situação que também é um bug é quando o jogador perde. Se você tentar, o teclado continua aceitando interação, permitindo inclusive que você acerte a palavra, o que precisamos impedir. Para a implementação que temos, precisamos verificar qual animação está em execução antes de executar as instruções na interação. Não podemos usar o flag `perdeu` que temos, pois ele só será atualizado cinco segundos após o jogador perder para que ele possa ver a animação completa. Sendo assim, no método `_gerarTeclado()`, dentro de nosso widget do teclado, coloque a instrução a seguir logo no início do `onTap` quando a letra não tiver sido ainda utilizada.

```
if (_jogoStore.animacaoFlare == 'enforcamento') return;
```

E para o jogador sair do jogo e retornar para a visão que apresenta o nosso Drawer? Precisamos disso também. Por simplicidade, faremos isso com um `FloatActionButton`. Vamos adicionar então o código a seguir no corpo do `Scaffold` do `JogoRoute`. Veja que estamos fazendo uso de `Observer` e `Visibility`. Usamos aqui um

`Observer` pelo fato de o nosso anterior estar dentro do `body` de `Scaffold`.

```
floatingActionButton: Observer(builder: (_) {
    return Visibility(
      visible: this._jogoStore.palavraAdivinhadaFormatada.isEmpty,
      child: FloatingActionButton(
        onPressed: () => Navigator.of(context).pop(),
        child: Icon(Icons.arrow_back),
      ),
    );
}),
```

Para concluirmos, vamos ajustar nossa animação do Flare. Logo que acessamos a visão do jogo, há uma demora na exibição da animação e isso pode dar uma impressão não muito boa para o usuário. Vamos mostrar a ele alguma atividade, enquanto a imagem é carregada. Faremos o uso de cacheamento, oferecido pelo próprio componente. Vamos adaptar todo o método `animacaoDaForca()` de nosso `jogo_mixin` para o que podemos ver na sequência. Leia os comentários após o código e lembre-se dos imports.

```
animacaoDaForca({String animacao}) {
    return Expanded(
      child: FlareCacheBuilder([
        AssetFlare(
            bundle: rootBundle, name:
'assets/flare/forca_casa_do_codigo.flr')
      ], builder: (BuildContext context, bool isWarm) {
        return !isWarm
            ? Center(
                child: Container(
                  child: Text(
                    "Carregando animação...",
                    textAlign: TextAlign.center,
                    style: TextStyle(
                      fontSize: 40,
                    ),
                  ),
                ),
```

```
              )
            : FlareActor(
                "assets/flare/forca_casa_do_codigo.flr",
                alignment: Alignment.center,
                fit: BoxFit.contain,
                animation: animacao,
              );
      }),
    );
}
```

Veja que agora temos o widget `FlareCacheBuilder`, que faz uso de um `AssetFlare`, que aponta para a animação que queremos cachear. No `builder` de nosso `FlareCacheBuilder`, temos um argumento, `isWarm`, que informará quando a animação estiver carregada. Em nosso caso, enquanto ela não estiver, exibimos uma mensagem para o usuário. Aqui você pode usar o recurso que for melhor ou julgar mais atrativo. Fica a seu critério.

Pronto, tudo certo agora. Vamos jogar?

Conclusão

Estes dois capítulos também foram longos, mas com muito conteúdo legal. Aplicamos aqui muito conhecimento, técnica, recursos e componentes. Conseguimos fazer nosso jogo funcionar. Espero que você tenha se divertido e aprendido bastante coisa.

Estamos quase terminando o livro. Deixei para o próximo capítulo o fechamento, já que o conteúdo proposto não está diretamente ligado com o que vimos aqui.

No próximo capítulo, veremos opções para criar uma visão de splash, conhecidas como launch screen, específicas para as plataformas. Ajustaremos a proibição de rotação em nosso app e também criaremos o ícone que queremos que nossa aplicação tenha quando for instalada.

Dê uma respirada e volte para concluirmos.

CAPÍTULO 12
Fechamento para o app

Este capítulo será curto mas importante. Veremos a criação da tela de abertura da aplicação e a configuração do ícone a ser exibido para o app nos dispositivos. Vamos a isso já.

12.1 Launch Screen

Ao executar nossa aplicação, você deve ter notado que demora um pouco até que a visão do app seja exibida. Isso se deve ao fato de o app ter uma configuração nativa para a exibição de sua inicialização em suas plataformas, iOS ou Android.

Fazendo uma rápida busca na internet, você verá que existem vários tutoriais que ensinam a realizar essa configuração trabalhando no projeto Android, no próprio Android Studio, e no X-Code para o iOS.

Isso é um pouco chato e trabalhoso. Mas felizmente temos um componente para nos auxiliar e que é bem fácil de utilizar, o `flutter_native_splash`.

Ele não tem dependência para a codificação e execução do app, mas tem para a etapa de desenvolvimento. Ou seja, registramos a dependência em um local diferente do `pubspec.yaml`, logo após o `dependencies`, como você pode ver na sequência. Certamente você já sabe, pois já utilizamos isso durante o livro.

```
dev_dependencies:
  flutter_native_splash: ^0.1.9
```

Além dessa dependência, precisamos registrar uma área específica para esse componente logo após o `dev_dependencies`. Veja isso no

código a seguir, onde estamos definindo a imagem e a cor de fundo para a launch screen. Fique à vontade para fazer suas escolhas.

```
flutter_native_splash:
  image: assets/images/launchscreen.png
  color: "336699"
```

É interessante que você leia a documentação das plataformas para saber detalhes em relação à resolução de imagens. Também é importante que você identifique a diferença entre *launch screen* e *splash screen*. A primeira é a visão de inicialização do app na plataforma, e a segunda pode ser algo que ficará visível enquanto algumas configurações e preparações são realizadas em seu app, antes de ele ser devidamente disponibilizado para o usuário.

Agora, precisamos executar uma instrução que será responsável pela geração e atualização dos projetos nativos para o iOS e Android. Sendo assim, no Terminal, na pasta de seu projeto, após realizar o `pub get` para atualizar as dependências, execute a instrução a seguir.

```
flutter pub run flutter_native_splash:create
```

Após a execução com sucesso da instrução anterior, pare o aplicativo em seu IDE e o execute novamente no emulador ou dispositivo, fica a seu critério. Veja a figura a seguir, que apresenta o resultado da inicialização do app.

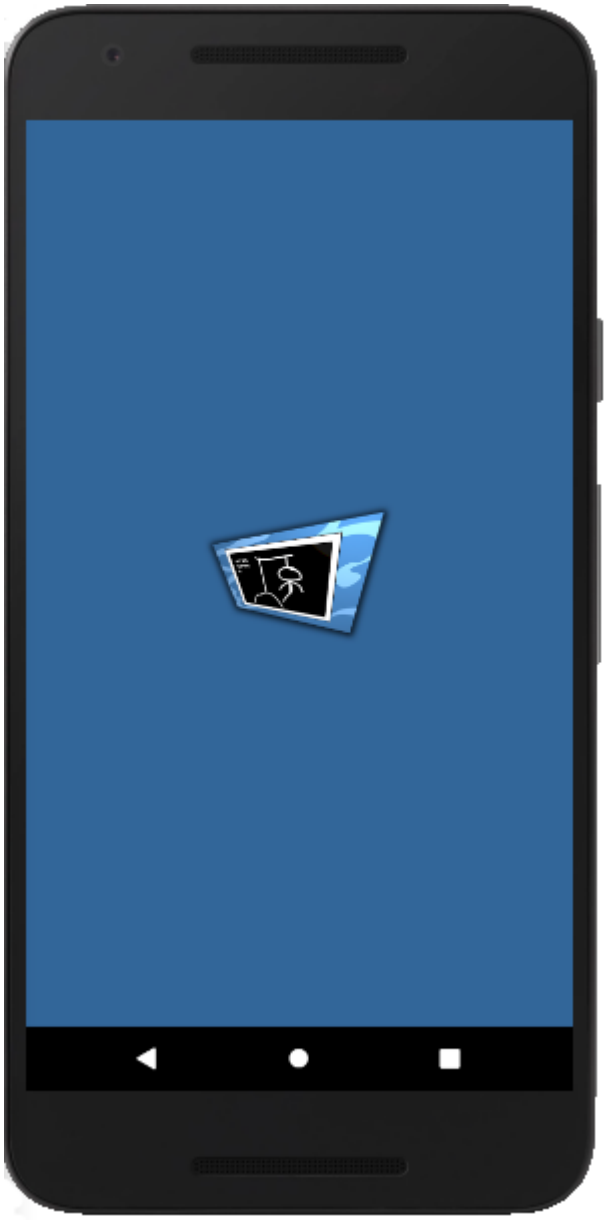

Figura 12.1: Launch Screen em execução

12.2 Ícone e nome para o app no dispositivo

Assim como a launch screen possui recomendações de uso, pois depende exclusivamente da plataforma em que será executada, há também recomendações para a inserção do ícone da aplicação que

devem ser seguidas para cada plataforma e isso deve ser verificado em suas respectivas documentações.

Entretanto, a exemplo do que fizemos na seção anterior, usaremos componentes de dependência em tempo de execução, que ajustarão os arquivos necessários para que o app tenha um ícone próprio e não o do Flutter. Este componente é o `flutter_launcher_icons`, que devemos adicionar às dependências tal como apresentado na sequência.

```
dev_dependencies:
  flutter_launcher_icons: ^0.7.5
```

Também precisamos de uma seção específica no `pubspec.yaml` para ele após a seção `flutter_native_splash`, que inserimos anteriormente. Ela está exemplificada na sequência, na qual definimos uma imagem padrão para as duas plataformas informadas como alvo para a geração do ícone. Outras opções estão disponíveis e podem ser verificadas na documentação do componente.

```
flutter_icons:
  image_path: "assets/images/launchscreen.png"
  android: true
  ios: true
```

Agora, precisamos executar o `pub get` e a instrução a seguir na pasta do projeto no terminal.

```
flutter pub run flutter_launcher_icons:main
```

Com a execução anterior, já podemos ver o resultado, mas vamos antes alterar também o nome do app, o que precisaremos fazer manualmente.

Para o Android, precisamos alterar o arquivo `AndroidManifest.xml`, que está em `android\app\src\main`. O que alteraremos é a propriedade `android:label` de `<application>`. Nela, coloque o nome desejado para o seu app.

Finalizando, para o iOS, o arquivo a ser alterado é o `Info.plist`, que está em `ios\Runner`, e o que alteraremos é a chave `CFBundleName` para o nome desejado também do nosso app.

Agora sim, pare a execução de seu aplicativo e o execute novamente. Verifique o ícone e o nome em seu dispositivo ou emulador, tal qual apresenta a figura a seguir, com o ícone e nomes que optei para mim.

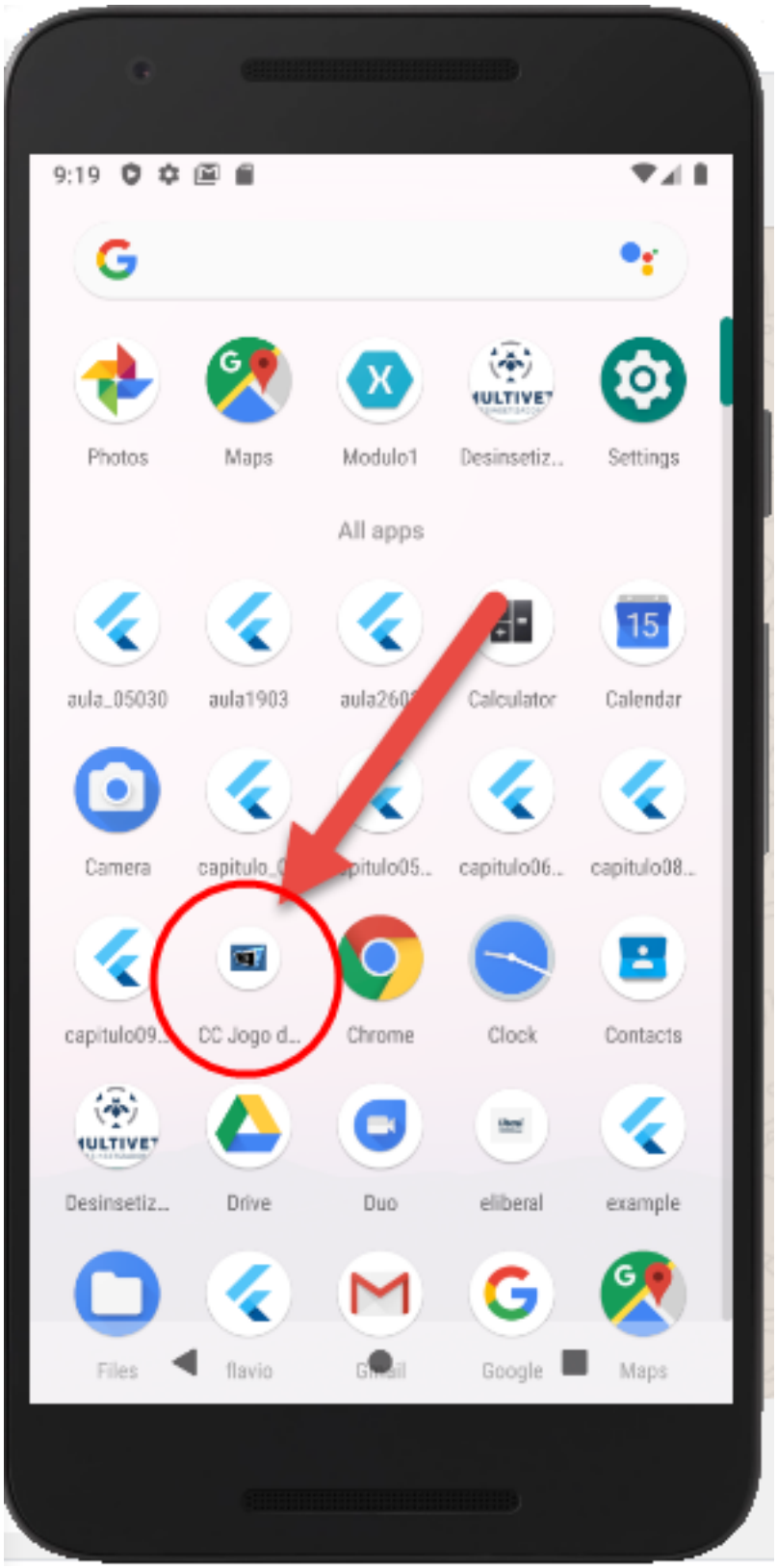

Figura 12.2: Ícone e nome para o app

12.3 Responsividade

Você já ouviu este termo? Podemos dizer, a grosso modo, que se trata de uma aplicação responder de maneira adequada em diferentes dispositivos, principalmente com tamanhos diferentes de telas. Ou seja, ela "responder" conforme as situações e locais onde é executada.

Isso pode ser o calcanhar de Aquiles para os desenvolvedores. O Flutter oferece alguns widgets interessantes que podem auxiliar nessa empreitada. Não os utilizamos no livro por não fazerem parte do contexto em que trabalhamos. Mas certamente veremos isso em um novo trabalho.

Ainda sobre responsividade, temos a situação de rotatividade dos dispositivos. Aqui precisamos pensar se nossa aplicação deverá responder a esses movimentos ou não. Para nosso app, como não trabalhamos responsividade, a decisão é que ele não aceite a orientação em modo de paisagem, *landscape*. Mas, se virarmos o dispositivo com a aplicação em execução, ele buscará se ajustar e não ficará legal. Então vamos proibir.

Para isso, em nosso `main.dart`, vamos inserir as instruções a seguir logo no início de nosso método `main()`. A primeira instrução é que garante que possamos executar a restrição de orientação. Sem ela, teremos o disparo de exceções que acusarão a tentativa de manipular recursos relacionados ao dispositivo e o processo de *binding* de nossa aplicação com os componentes que serão renderizados antes de a aplicação estar totalmente inicializada. As outras duas são semânticas e habilitam as duas possíveis orientações.

```
WidgetsFlutterBinding.ensureInitialized();
SystemChrome.setPreferredOrientations(
  [DeviceOrientation.portraitDown, DeviceOrientation.portraitUp]);
```

Conclusão

Fechamos nosso aplicativo do jogo da forca com este último capítulo de conteúdo prático. No próximo capítulo, há apenas um fechamento sobre o que vimos.

Deixei para você a ideia de criar um score para o jogo, pois temos essa opção no menu. Pense em como pontuar e classifique as tentativas do usuário. Ainda, se você quiser, dá para pensar em ter os dados na nuvem em uma autenticação e ter o score de todos os usuários. Pense em desafios. É possível colocar muita diversão neste jogo.

Capítulo 13
Os estudos não param por aqui

Os dispositivos móveis já fazem parte do dia a dia da maioria da população. Você, como programador(a) ou desenvolvedor(a) não pode perder essa onda.

São duas as maiores plataformas móveis atualmente disponíveis (iOS e Android) e várias são as ferramentas para desenvolvimento de aplicações para elas, quer seja de forma nativa ou híbrida. Neste livro, você teve acesso à metodologia de implementação de aplicações que podem ser executadas nas plataformas descritas por meio do Flutter, um apaixonante framework que, no momento de escrita deste livro, já tem versões quase liberadas para desenvolvimento web e desktop.

Fizemos um passeio por grande parte dos controles disponibilizados pelo Flutter, além de diversos componentes que fomos identificando e utilizando.

Trabalhamos a gestão do estado de aplicações usando três técnicas, todas recomendadas. A decisão de qual opção usar para seu projeto é sua. E lembre-se de que essas três não são as únicas que existem, tem bem mais.

Como o foco da aplicação desenvolvida no livro foi um jogo, não podíamos deixar de trabalhar com o acesso à base de dados, o que fizemos por meio do SQLite, chamado no Flutter de SQFLite. Fizemos uso também de um componente específico para esta finalidade.

O livro não esgotou os temas trabalhados. É preciso dedicação para investigação e descoberta de novas tecnologias e recursos. Tenho certeza de que ele foi mais do que um pontapé inicial para o seu desenvolvimento no que diz respeito a aplicações móveis,

considerando que este livro está para um nível intermediário de conhecimento.

Espero que o conteúdo que trabalhei tenha despertado em você uma grande curiosidade e que o livro tenha desmitificado o desenvolvimento de aplicativos para dispositivos móveis. Agora é bola para a frente na evolução.

Obrigado pela companhia e sucesso.