## Aula 1 – Troca de valores entre a variável

**[ Contexto ]**

Para qualquer aprendizado, uma contextualização se faz necessária, sobre ambientes de desenvolvimento, paradigmas de linguagens utilizadas, sintaxes, palavras reservadas (instruções), e é este o objetivo deste primeiro vídeo.

**[ Problema 1 ]**

Teremos duas variáveis, que poderiam ser de qualquer tipo, mas, neste exemplo, utilizaremos o tipo de dados para valores inteiros. Estas variáveis serão inicializadas com valores arbitrários em um primeiro momento. Após isso, trocaremos estes valores entre as variáveis.

**[ Solução 1 ]**

Neste vídeo a ferramenta adotada será o Dartpad, que é uma ferramenta em um estilo comum na atualidade, conhecido como *playground* e pode ser acessado e utilizado de maneira *on-line*, pela URL <https://dartpad.dev/>.

|  |  |
| --- | --- |
|  |  |
| **Código** | **Resultado** |

**[ Teoria ]**

A estrutura de pastas adotada na aplicação é uma hierarquia que adoto seguindo alguns princípios de Clean Architecture (Arquitetura Limpa), buscando separar os artefatos em suas responsabilidades.

Em Flutter, tudo que é renderizado (desenhado), é um Widget. Trabalhamos aqui um tipo especial de widget, o Stateful, que tem a característica de permitir atualização dos widgets renderizados por ele, com base em mudança nos estados (valores) de variáveis ou dos próprios widgets.

Nossa página, por ser um Stateful, possui ligado a ela uma classe State, parametrizada com o tipo de nosso widget. É essa classe que nos possibilita a gerência de estado por meio do setState().

Optei na estrutura básica de nossa página já utilizando um **Scaffold**, com um **AppBar** e um **Column** como **body**, que será o container onde nossos controles serão registrados. Um Scaffold dá a aplicação Flutter a característica ligada a princípios de Material Designer, base para aplicações Flutter.

Criamos os dois controles de entrada de dados para o usuário informar seu email e sua senha. Fizemos uso do TextField(), alterando poucas propriedades, como o tipo de teclado, ícone para prefixo e mensagem de ajuda para o usuário saber o que precisa digitar, algo conhecido como **PlaceHolder**.

Inserimos o botão de acesso, com um Padding, que dará um espaço separando o botão dos controles de acesso.

Utilizamos o **FocusNode**() para possibilitar navegação entre os controles, o que fizemos configurando as propriedades **textInputAction** o **onSubmitted** de nosso controle de email.

Feito isso, precisamos validar os dados informados pelo usuário e seu acesso à área restrita do App e é o que faremos a seguir.

### Vídeo 1.3 - Validação de Acesso

**[ Contexto ]**

Com a página de autenticação pronta, precisamos trabalhar algumas regras em relação aos dados informados e renderização de mensagem de erro para o caso de não cumprimento das regras..

Precisamos saber que não haverá validação se email e senha fazem parte de alguma base de dados, eles apenas devem ser preenchidos e o email deve seguir a regra de um endereço correto em sua formatação.

Trabalharemos também a habilitação ou não do botão de acesso às funcionalidades do aplicativo quando as regras para email e senha forem cumpridas.

**[ Problema 1]**

Precisamos ter em nossa página a possibilidade de exibir ao usuário mensagens de erro para o caso de não cumprimento das regras para email e senha.

**[ Solução 1 ]**

Para resolver este problema, implementamos widgets **Text** abaixo dos **TextFields**, com as mensagens a serem exibidas em caso de não cumprimento das regras. Porém, como estes widgets terão muita coisa em comum, criamos um método privado, que recebe a mensagem de erro e o utilizamos, minimizando redundância de código.

// Método a ser implementado após o build()

*\_mensagemDeErro({String mensagem}) {*

*return Row(*

*mainAxisAlignment: MainAxisAlignment.end,*

*children: [*

*Text(*

*mensagem,*

*style: TextStyle(*

*color: Colors.red,*

*fontSize: 12,*

*fontWeight: FontWeight.bold,*

*),*

*),*

*],*

*);*

*}*

// Código a ser implementado após TextField de email

\_mensagemDeErro*(*mensagem: 'Um email correto é obrigatório'*)*,

// Código a ser implementado após TextField de senha



**[ Problema 2]**

Temos agora a necessidade de fazer com que a mensagem seja exibida apenas enquanto as regras de validação não são cumpridas, pois com a implementação anterior, mesmo que cumpramos estas regras, as mensagens continuam sendo exibidas.

**[ Solução 2 ]**

Para aplicar as regras de validação, precisamos ter os valores informados pelo usuário armazenados em variáveis, para que então as regras possam ser aplicadas nelas.

// Código a ser implementado após as declarações dos FocusNodes

String \_email = '';

String \_senha = '';

Com as variáveis declaradas, aplicamos as regras de validação para decidir se as mensagens são ou não exibidas.

// Métodos a serem implementados após o \_mensagemDeErro

\_oEmailEhValido*() {*

if *(*\_email.trim*()*.length == 0*)* return false;

Pattern pattern =

r'^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)\*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$';

RegExp regex = new RegExp*(*pattern*)*;

return *(*regex.hasMatch*(*\_email*))*;

*}*

\_aSenhaEhValida*() {*

return \_senha.trim*()*.length > 0;

*}*

Encapsulamos o retorno do método **\_mensagemDeErro()** no widget Visibility() para definirmos quando a mensagem deve ser renderizada.

// Adaptação para a mensagem de erro do email

Visibility*(*

visible: !\_oEmailEhValido*()*,

child:

\_mensagemDeErro*(*mensagem: 'Um email correto é obrigatório'*)*,

*)*,

// Adaptação para a mensagem de erro da senha

Visibility*(*

visible: !\_aSenhaEhValida*()*,

child: \_mensagemDeErro*(*mensagem: 'A senha é obrigatória'*)*,

*)*,

Para que a expressão que determina se o retorno para nossa mensagem de erro seja avaliada, precisamos fazer com que, a cada interação do usuário, as variáveis sejam atualizadas e o estado da página seja forçado a uma atualização.

// Código a ser inserido no widget TextField de email

onChanged: *(*value*) {*

setState*(() {*

\_email = value;

*})*;

*}*,

// Código a ser inserido no widget TextField de senha

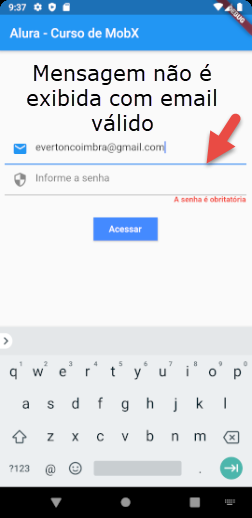
onChanged: *(*value*) {*

setState*(() {*

\_senha = value;

*})*;

*}*,

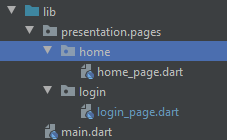


**[ Problema 3]**

Quando o usuário interagir com o botão acessar, ele precisa ser direcionado para a área restrita do App, e não temos ainda uma página que representa isso.

**[ Solução 3]**

Criamos uma pasta chamada **home** em **presentation.pages** e nela um arquivo chamado **home\_page.dart**.



import 'package:flutter/material.dart';

class HomePage extends StatelessWidget *{*

@override

Widget build*(*BuildContext context*) {*

return Scaffold*(*

appBar: AppBar*()*,

body: Center*(*

child: Text*(*'Seja bem-vindo'*)*,

*)*,

*)*;

*}*

*}*

**[ Problema 4]**

Com a home\_page criada, é preciso possibilitar a navegação a ela quando o usuário interagir com o botão de acesso.

**[ Solução 4]**

Implementamos na propriedade **onPressed** de **RaisedButton** a navegabilidade para a **HomePage**.

// Adaptação para a propriedade onPressed no RaisedButton

onPressed: *()* => Navigator.*push(*

context,

MaterialPageRoute*(*

builder: *(*context*)* => HomePage*()*,

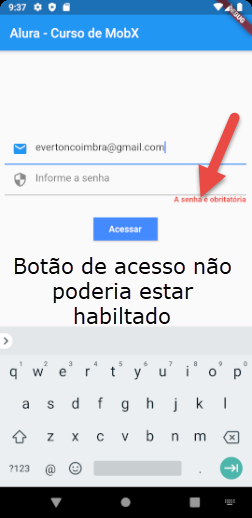
*)*,

*)*,



**[ Problema 5]**

Finalizando esta etapa, tem-se a situação do botão para acessar a área restrita da aplicação, que se encontra sempre habilitado e permite o acesso à área restrita com os valores errados e até sem valores. Mas essa navegabilidade só deve ocorrer quando os dados estiverem corretos.



**[ Solução 5]**

Criamos um método privado que valida os dois dados e delegamos a ele a responsabilidade em definir qual método será atribuído ao RaisedButton.

*// Método a ser implementado após a validação da senha*

*\_onPressedParaBotaoAcessar*

*() {*

*if (\_oEmailEhValido() && \_aSenhaEhValida()) {*

*return () => Navigator.push(*

*context,*

*MaterialPageRoute(*

*builder: (context) => HomePage(),*

*),*

*);*

*} else {*

*return null;*

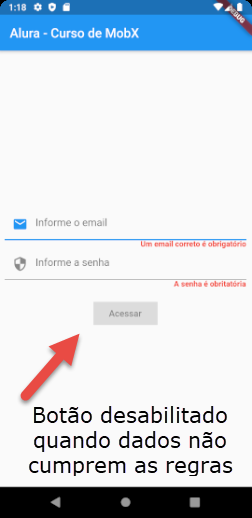
*}*

*}*

Adaptamos a definição da propriedade **onPressed** para o retorno do método **\_onPressedParaBotaoAcessar()**.

// Adaptação para a propriedade onPressed no RaisedButton

onPressed: \_onPressedParaBotaoAcessar*()*,



**[ Teoria ]**

Seguindo alguns princípios de UX (User Experience/Experiência do Usuário), precisamos dar ao usuário um retorno para que ele possa saber se está cumprindo as regras de negócio para a funcionalidade em questão. E este retorno não pode ser só ao final de sua interação, pois muitas vezes a aplicação não saberá quando o usuário terminou de informar o dados.

Implementamos a mensagem de erro em um método privado, pois faz uso de **Text**, com propriedades comuns, o que deixaria o método **build()** poluído. Essa é uma boa prática. Há situações que podemos utilizar mixins ou ainda criarmos widgets específicos para estas situações.

Para validarmos o email para verificarmos se é ou não válido, em sua composição, fazemos uso de expressões regulares. Novamente criamos métodos para abstrairem esta funcionalidade.

O widget **Visibility()** permite uma renderização condicional de outro widget, declarado como seu filho (child), por meio da propriedade **visible**.

Utilizando a propriedade **onChanged()** de um **TextField** podemos, com o valor atual informado, realizar qualquer procedimento de validação/atualização.

Quando se tem uma página com vários controles, pode-se tratar o conjunto de controles como formulário, o que leva a um método que o valide.

A página que representa a área restrita do App é um Stateless widget, pois, a princípio, não haverá necessidade de atualizações de estado de seus controles.

A navegação entre páginas é possível com o uso de **Navigator**, onde o método **push()**, fazendo uso de **MaterialRoutePage**, solicita a nova pagina, trabalhando sempre com o contexto atual e o enviando para a nova página.

Quando a propriedade **onPressed** de um **RaisedButton** tem a ela o valor null atribuído, por definição, o botão fica desabilitado.

**[ Fechamento ]**

Já trabalhamos a validação da entrada de dados, nos possibilitando exibir ou não mensagens de erros e também habilitar ou não o botão de acesso. Mas precisamos fazer uso do setState(), nos levando à necessidade de um Stateful. Não é isso que queremos, pois isso causa a renderização da tela como um todo.

Adiante, teremos a mesma funcionalidade, mas fazendo uso do MobX. Ele que vai gerenciar o estado de nossos controles. Não teremos mais o setState() e poderemos ter nossa página como um Stateless, pois o MobX nos auxiliará em renderizar apenas o widget que teve alteração em seu estado.

## Aula 2 - MobX e Flutter MobX

### Vídeo 2.1 - Introdução ao MobX

**[ Contexto ]**

Com o que trabalhamos anteriormente, já temos os controles de entrada para email e senha sendo validados de acordo às interações do usuário e o botão de acesso fica habilitado apenas quando as regras para os dois controles sejam cumpridas.

Agora precisamos focar na melhoria em nossa gerência do estados dos controles visuais, pois não queremos que nossa página seja atualizada por completo a cada letra digitada para um email e uma senha, pois isso pode degradar o tempo no processo de renderização de nossa página.

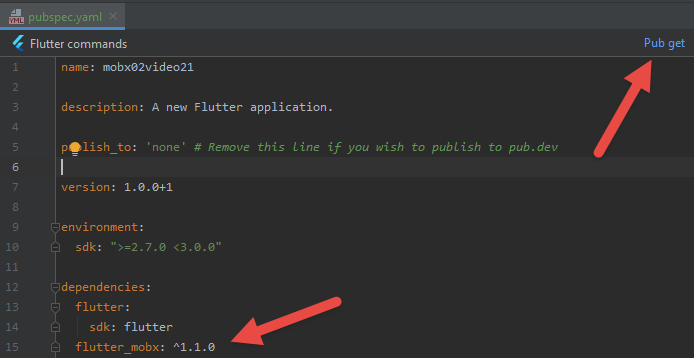
Queremos fazer com que a renderização ocorra apenas onde há necessidade, que é na exibição ou não da mensagem de erro e habilitação ou não do botão de acesso à aplicação. E é exatamente neste ponto que introduziremos o uso do MobX.

**[ Problema 1 ]**

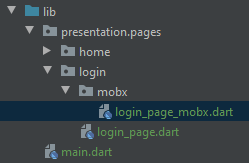
Temos nossas validações sendo realizadas na classe que representa a página de login, pois fazemos uso do setState() para que, após a interação do usuário, as variáveis para os valores digitados sejam atualizadas e as validações possam ocorrer. Agora, para uso do MobX, precisamos tirar desta classe esta responsabilidade e passar para uma classe específica para este objetivo.

**[ Solução 1 ]**

Como nossa solução está utilizando o pacote mobx, instalamos nas dependências de nosso projeto, no **pubspec.yaml**, o componente **flutter\_mobx** e em seguida realizar o **Pub get**.



Com o pacote registrado e obtido, criamos uma pasta chamada **mobx** dentro de **presentation.pages.login** e nela um arquivo chamado **login\_page\_mobx.dart**.



import 'package:mobx/mobx.dart';

class LoginPageMobx *{*

Observable*<*String*>* email = Observable*(*''*)*;

Observable*<*String*>* senha = Observable*(*''*)*;

Action atualizarEmail;

Action atualizarSenha;

LoginPageMobx*() {*

atualizarEmail = Action*(*\_atualizarEmail*)*;

atualizarSenha = Action*(*\_atualizarSenha*)*;

*}*

\_atualizarEmail*(*String novoEmail*) {*

email.value = novoEmail;

*}*

\_atualizarSenha*(*String novaSenha*) {*

senha.value = novaSenha;

*}*

*}*

**[ Problema 2 ]**

Com a atualização das variáveis para email e senha levadas para uma classe específica que utiliza recursos do MobX, precisamos adaptar a página para utilizar esta nova técnica.

**[ Solução 2 ]**

Declarar e inicializar uma variável que representará uma instância da classe **LoginPageMobX** e retirar as declarações para as variáveis **\_email** e **\_senha** e adaptar os métodos de validação.

// Implementar logo no início da classe

final \_loginPageMobx = LoginPageMobx*()*;

// Código a ser removido da classe

String \_email = '';

String \_senha = '';

// Adaptar referência ao valor informado para o email no método \_oEmailEhValido

if *(*\_loginPageMobx.email.value.trim*()*.length == 0*)* return false;

return *(*regex.hasMatch*(*\_loginPageMobx.email.value*))*;

// Adaptar referência ao valor informado para a senha no método \_aSenhaEhValida

return \_loginPageMobx.senha.value.trim*()*.length > 0;

**[ Problema 3 ]**

Já temos toda a lógica de negócio ligada a interface com o usuário levada para uma classe específica, fazendo uso de Observables e Actions, do MobX. Temos também nossa página de login já consumindo este recurso, em substituição ao setState(), mas as mensagens aparecem sempre e o botão está sempre habilitado. O que nos falta? Nos falta um componente que ouça as mudanças dos estados e reflita-as na interface com o usuário e é isso que resolveremos.

**[ Solução 3 ]**

Adaptamos a renderização das mensagens e do RaisedButton para observarem as mudanças de estado gerenciadas pelo MobX.

// Encapsulamos o Visibility para email em um Observer, widget do flutter\_mobx

Observer*(*builder: *(*\_*) {*

return Visibility*(*

visible: !\_oEmailEhValido*()*,

child:

\_mensagemDeErro*(*mensagem: 'Um email correto é obrigatório'*)*,

*)*;

*})*,

// Encapsulamos o Visibility para senha em um Observer, widget do flutter\_mobx

Observer*(*builder: *(*\_*) {*

return Visibility*(*

visible: !\_aSenhaEhValida*()*,

child: \_mensagemDeErro*(*mensagem: 'A senha é obrigatória'*)*,

*)*;

*})*,

// Encapsulamos o RaisedButton em um Observer, widget do flutter\_mobx

child: Observer*(*builder: *(*\_*) {*

return RaisedButton*(*

child: Text*(*'Acessar'*)*,

onPressed: \_onPressedParaBotaoAcessar*()*,

*)*;

*})*,

**[ Problema 4 ]**

Tudo está funcionando, mas, para comprovar que podemos ter nossa página como Stateless, precisamos mudar sua declaração e é isso que faremos.

**[ Solução 4 ]**

Retiramos a declaração da classe de estado e adaptamos a declaração da classe **HomePage**.

// Remoção da linha de definição da classe de estado

class \_LoginPageState extends State*<*LoginPage*> {*

// Remoção do override para createState na classe LoginPage

@override

\_LoginPageState createState*()* => \_LoginPageState*()*;

// Mudança da extensão da classe para StatelessWidget

class LoginPage extends StatelessWidget *{*

// Retirada do fechamento da chave que existe antes da declaração de \_loginPageMobx

}

**[ Problema 5 ]**

Com a mudança que fizemos de Statefull para Stateless o método **\_onPressedParaBotaoAcessar()** apresentou um erro, pois um Stateless não tem um contexto para a aplicação fora do método **build()**, mas resolveremos isso.

**[ Solução 5 ]**

Enviamos ao método **\_onPressedParaBotaoAcessar()** o contexto da aplicação, conhecido pelo método **build()**.

// Adaptação da assinatura do método \_onPressedParaBotaoAcessar

*\_onPressedParaBotaoAcessar({*BuildContext context*}) {*

// Adaptação do onPressed de RaisedButton para enviar o contexto ao método \_onPressedParaBotaoAcessar

onPressed: \_onPressedParaBotaoAcessar*(*context: context*)*,



**[ Teoria ]**

O foco da compreensão no uso do MobX é levarmos para a ele o modelo de negócio para nossa visão. Com isso, temos nele a responsabilidade de atualizar as propriedades que têm mudanças de estados, que causarão nova renderização. Criamos então uma classe MobX para nossa página e nela temos propriedades para email, senha e ações que possibilitarão o registro da atualização dos valores para estas propriedades, conforme o usuário interaja.

Com a lógica de atualização transferida para o MobX, atualizamos nossa página para utilizar essa nova implementação e também adaptamos os métodos de validação para esta nova situação.

Sempre que uma action realiza uma mudança no estado de uma propriedade observável, o MobX realiza uma notificação para a hierarquia de widgets de nossa aplicação, dizendo: mudei isso! Alguém me escuta?

Aí entra o widget **Observer**, do **flutter\_mobx**. Ele fica sempre observando, atento à comunicação de mudança e, quando ela corre, ele possibilita que as propriedades dentro de seu **builder**, que são observáveis sejam novamente renderizadas.

É com a inclusão dos Observers que monitoramos as alterações no email e senha e, com os estados em atualização, poderemos exibir ou não a mensagem de erro e habilitar ou não o botão de acesso.

Com o MobX, apenas os componentes observáveis são renderizados e apenas quando ocorrem mudanças.

Com as mudanças realizadas, transformamos nossa página, de Statefull para Stateless.

Em uma classe Statefull, temos ligada a ela uma classe de estado, que mantém em seu escopo o contexto da aplicação. Quando utilizamos um Stateless, este contexto existe apenas em **build()**, pois não temos a classe de estado.

Já temos nossa aplicação funcionando com o MobX, mas isso nos levou ainda a um pouco mais de código, que pode ser minimizado. Estamos falando de Stores, uma classe específica, que minimiza muito nosso código e é o que veremos na sequência.

### Vídeo 2.2 - Classe Store para Login

**[ Contexto ]**

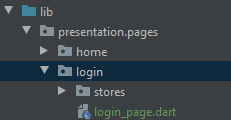
Já vimos o benefício do MobX na gerência de estados de nossos widgets, fazendo uso direto das classes Observable e Action. Mas ainda temos muito código na classe de nossa página. Com o uso de classes Store, o MobX possibilita mais a delegação da responsabilidade de gestão de estado e é o que veremos.

**[ Problema 1 ]**

Temos a atualização dos valores informados para email e senha na classe MobX que criamos anteriormente, mas as validações destes dados ainda estão na página. Como o objetivo no uso do MobX é tirar toda esta gerência da página, passando a ele esta funcionalidade, precisamos criar uma classe do tipo Store, que funcionará como um repositório e possibilitará resolver este problema.

**[ Solução 1 ]**

Criamos uma pasta chamada **stores** dentro de **presentation.pages.login** e nela um arquivo chamado **login\_page\_store.dart**, removendo a pasta **mobx** e o arquivo **login\_page\_store.dart** que está dentro dela.



import 'package:mobx/mobx.dart';

part 'login\_page\_store.g.dart';

class LoginPageStore = \_LoginPageStore with \_$LoginPageStore;

abstract class \_LoginPageStore with Store *{*

@observable

String email = '';

@observable

String senha = '';

@computed

bool get oEmailEhValido *{*

if *(*email.trim*()*.length == 0*)* return false;

Pattern pattern =

r'^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)\*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$';

RegExp regex = new RegExp*(*pattern*)*;

return *(*regex.hasMatch*(*email*))*;

*}*

@computed

bool get aSenhaEhValida => senha.trim*()*.length > 0;

@computed

bool get oFormularioEhValido => *(*oEmailEhValido && aSenhaEhValida*)*;

@action

atualizarEmail*({*String newValue*}) {*

email = newValue;

*}*

@action

atualizarSenha*({*String newValue*}) {*

senha = newValue;

*}*

*}*

**[ Problema 2 ]**

O código do início do arquivo **login\_page\_store.dart** possui dois erros: na declaração da classe **part**, que compõe a classe e na declaração da classe concreta, que utiliza um mixin. É preciso resolver isso.

**[ Solução 2 ]**

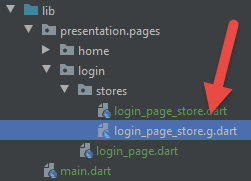
Existe uma necessidade de geração de código, devido aos decorators que usamos na classe Store. Para que sejam gerados, registramos os pacotes **mobx\_generator** e **build\_runner** no **pubspec.yaml**, porém, não em dependencies, mas sim em **dev\_dependencies**, pois estes pacotes serão utilizados em tempo de desenvolvimento e não de execução.

dev\_dependencies:

build\_runner: ^1.9.0

mobx\_codegen: ^1.0.3

Para resolver o problema do mixin a ser criado, executamos uma instrução no terminal, na pasta do projeto. A instrução padrão é **flutter packages pub run build\_runner build**, porém, em alguns casos, pode acontecer um erro relativo à código gerado anteriormente e que precisa ser removido antes de uma nova geração e isso pode ser realizado pela instrução **flutter packages pub run build\_runner build --delete-conflicting-outputs**. Por fim, a instrução **flutter packages pub run build\_runner watch** é executada, pois ela permite a geração dinâmica de código, enquanto estamos codifificando nossas classes Store.



**[ Problema 3 ]**

Com toda estrutura para utilizarmos classes Store do MobX, é preciso adaptar a classe de login para consumir estes novos serviços.

**[ Solução 3 ]**

Declaramos uma variável que receberá a instância da classe Store implementada anteriormente.

// Remover este código da classe

final \_loginPageMobx = LoginPageMobx*()*;

// Implementar este código após a definição dos FocusNodes

final \_loginPageStore = LoginPageStore*()*;

// Atualizar onChanged do TextField de email

onChanged: *(*value*)* => \_loginPageStore.atualizarEmail*(*newValue: value*)*,

// Atualizar onChanged do TextField de senha

onChanged: *(*value*)* => \_loginPageStore.atualizarSenha*(*newValue: value*)*,

Removemos os métodos **\_oEmailEhValido()**, **\_aSenhaEhValida()** e adaptamos o **\_onPressedParaBotaoAcessar()**.

// Atualizar o visible do Visibility para mensagem de erro do email

visible: !\_loginPageStore.oEmailEhValido,

// Atualizar o visible do Visibility para mensagem de erro da senha

visible: !\_loginPageStore.aSenhaEhValida,

// Adaptar a expressão de avaliação para fazer uso do Store

if *(*\_loginPageStore.oFormularioEhValido*)*



**[ Teoria ]**

O MobX possibilita o uso de classes que estendem Store, uma classe específica do pacote, o que pode melhorar bem mais a maneira que podemos gerenciar os estados de widgets, em nosso caso, a página de login.

Implementamos a classe Store, de maneira privada e abstrata (que não permite instanciação) com as propriedades observáveis e métodos actions, todos decorados para serem vistos com estas características.

Para a validação do email, senha e liberação do botão de acesso, implementamos na classe Store propriedades @computed, que trabalham como métodos de apenas leitura (get), tirando da página de login esta responsabilidade.

Declaramos uma classe concreta como sendo uma extensão da classe Store, fazendo uso de um Mixin, criado por um gerador de código do Flutter em conjunto com o MobX.

Mixins são artefatos que podem ser utilizados em classes. Podem ser vistos como um complemento ao processo de herança por comportamento, dando funcionalidades adicionais à classe que o utiliza, por meio da cláusula **with**, ao final da assinatura da classe.

Dart é uma linguagem que aceita decorators, ou anotações, porém, estes recursos são utilizados para geração de código e não para interpretação durante a execução das aplicações, como ocorre em linguagens como Java e C#.

Para a implementação de nossa classe Store, com maiores flexibilidades para a gerência de estado oferecida pelo MobX, registramos dependências para o desenvolvimento de componentes que gerarão códigos necessários para a classe Store e executamos uma instrução no terminal para que o código possa ser gerado.

Com a validação para o formulário levada para a Store, adaptamos a condição avaliada no método que retorna qual método deverá ser atribuído ao **onPressed** do **RaisedButton** e removemos os métodos que validam o email e senha da página.

**[ Fechamento ]**

Tivemos uma boa experiência na transição da gerência de estados com setState() para o uso de Store em nossa aplicação de login. Na sequência começaremos a implementar a parte de visualização de produtos oferecidos pelo Alura Delivery, onde o usuário poderá selecionar os itens que deseja, informando a quantidade para eles, utilizando um number picker.

## Aula 3 - Visualização de Produtos

### Vídeo 3.1 - Itens disponíveis e selecionados

**[ Contexto ]**

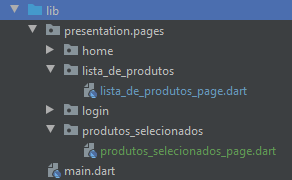
Desenvolvemos anteriormente uma página que é acessada após o usuário realizar o login com seu email e senha. Ocorre que esta página apresenta apenas uma mensagem de boas vindas e nós precisamos que ela possua um recurso de navegação entre duas páginas: uma que apresenta todos os produtos que o Alura Delivery tem disponíveis e uma que exibirá os produtos que o usuário selecionará para seu pedido. Possibilitaremos a navegação entre estas páginas por meio da propriedade **bottomNavigationBar** do Scaffold.

**[ Problema 1 ]**

Para que possa ser oferecida ao usuário a possibilidade de navegar entre duas páginas, precisamos delas criadas e é o que faremos.

**[ Solução 1 ]**

Criamos duas pastas dentro de **presentation.pages**,chamadas **lista\_de\_produtos** e **produtos\_selecionados**. Na primeira criamos um arquivo chamado **lista\_de\_produtos\_page.dart** e na segunda outro arquivo, chamado **produtos\_selecionados\_page.dart**. Estes arquivos terão as classes que precisamos para as páginas de produtos existentes e de produtos selecionados.



.

import 'package:flutter/material.dart';

class ListaDeProdutosPage extends StatelessWidget *{*

@override

Widget build*(*BuildContext context*) {*

return Center*(*

child: Text*(*'Lista de Produtos'*)*,

*)*;

*}*

*}*

import 'package:flutter/material.dart';

class ProdutosSelecionadosPage extends StatelessWidget *{*

@override

Widget build*(*BuildContext context*) {*

return Center*(*

child: Text*(*'Produtos selecionados'*)*,

*)*;

*}*

*}*

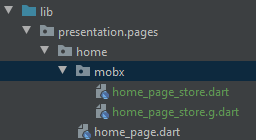
**[ Problema 2 ]**

Com a utilização da propriedade **bottomNavigationBar** do Scaffold para promover a navegação entre as páginas criadas, surge a necessidade de se planejar a atualização de estado, pois as duas páginas serão exibidas na **HomePage**, de acordo à opção do usuário. Ainda, o título exibido na **HomePage**, será atualizado de acordo à essa escolha do usuário. Queremos utilizar o MobX para isso e implementaremos esta solução.

**[ Solução 2 ]**

Criamos uma pasta dentro de **presentation.pages.home**,chamada **stores** e nela criamos um arquivo chamado **home\_page\_store.dart**.

Executamos a instrução **flutter packages pub run build\_runner watch**, caso não esteja em execução, para que a classe necessária para o uso MobX seja criada.



import 'package:mobx/mobx.dart';

part 'home\_page\_store.g.dart';

class HomePageStore = \_HomePageStore with \_$HomePageStore;

abstract class \_HomePageStore with Store *{*

@observable

int paginaAtual = 0;

@computed

String get tituloHomePage =>

*(*paginaAtual == 0*)* ? 'Produtos disponíveis' : 'Produtos selecionados';

@action

alternarPagina*({*int novaPagina*}) {*

paginaAtual = novaPagina;

*}*

*}*

**[ Problema 3 ]**

Com as páginas necessárias para a aplicação e a para gerência de estado criadas, é preciso fornecer ao usuário a possibilidade de navegar entre as duas páginas e é o que faremos.

**[ Solução 3 ]**

Adaptamos a classe HomePage para que possa oferecer ao usuário a possibilidade de escolher entre as duas páginas, atualizando o título, de acordo à seleção realizada.

*// Declaração da store e de uma lista com as duas páginas para navegação*

*final HomePageStore \_homePageStore = HomePageStore();*

*final List<Widget> \_paginas = [*

*ListaDeProdutosPage(),*

*ProdutosSelecionadosPage(),*

*];*

*// Inserção de um título na AppBar de HomePage, com valor dependendo da ação do usuário*

*appBar: AppBar(*

*title: Observer(builder: (\_) {*

*return Text(\_homePageStore.tituloHomePage);*

*}),*

*),*

*// Adaptação do conteúdo de body para que renderize a página selecionada pelo usuário*

*body: Observer(builder: (\_) {*

*return \_paginas[\_homePageStore.paginaAtual];*

*}),*

*// Criação, abaixo do builder, do método que gerará os botões de opções de acesso*

*\_bottomNavigationBarItem({IconData icone, String titulo}) {*

*return BottomNavigationBarItem(*

*icon: new Icon(*

*icone,*

*),*

*title: new Text(titulo),*

*);*

*}*

*// Implementação da propriedade bottomNavigationBar, no Scaffold*

*bottomNavigationBar: Observer(builder: (\_) {*

*return BottomNavigationBar(*

*iconSize: 48,*

*currentIndex: \_homePageStore.paginaAtual,*

*onTap: (index) => \_homePageStore.alternarPagina(novaPagina: index),*

*items: [*

*\_bottomNavigationBarItem(icone: Icons.menu, titulo: 'Produtos'),*

*\_bottomNavigationBarItem(icone: Icons.add\_shopping\_cart, titulo: 'Pedido'),*

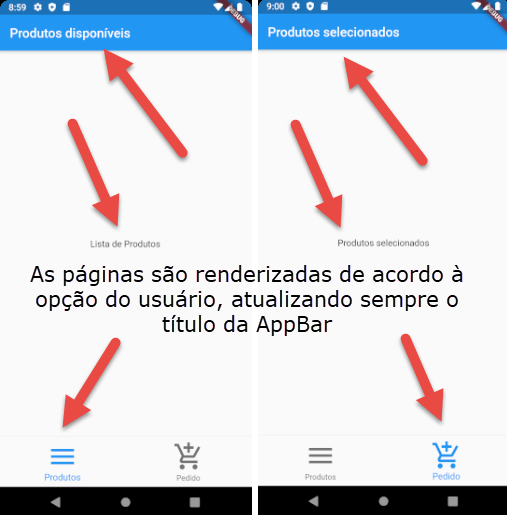
*],*

*);*

*}),*

*// Adaptação da propriedade home de MaterialApp para agilizar os testes*

*home: HomePage(),*

**

**[ Teoria ]**

Seguimos utilizando a estrutura de pastas adotada na aplicação para a criação das páginas de listagem de produtos disponíveis e a de produtos selecionados.

Criamos a página store com propriedade se ações que são consumidas pela HomePage para renderizar corretamente a escolha do usuário e a declaramos na HomePage.

Para facilitar o processo de navegação, criamos uma lista com instâncias para cada página disponível para navegação.

Inserimos um título na **AppBar** da página, fazendo uso de Observer, do MobX, para que o título seja renderizado corretamente.

Adaptamos o **body** de Scaffold, para renderizar a página selecionada pelo usuário.

Criamos um método para gerar as opções que serão disponibilizadas no **bottomNavigationBar**, minimizando redundância de código.

Configuramos a propriedade **bottomNavigationBar** com o widget **BottomNavigationBar**, configurando tamanho padrão para ícones, página atual a ser renderizada e comportamento para quando usuário interagir com as opções. Tudo fazendo uso do MobX.

Definimos a **HomePage** como página de inicialização, ao invés da **LoginPage**, buscando agilizar os testes.

Como próxima etapa é termos os produtos disponíveis pelo Alura Delivery exibidos na sua respectiva página, e trabalharemos isso na sequência.

### Vídeo 3.2 - Listagem de Produtos

**[ Contexto ]**

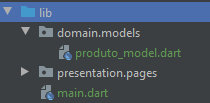
Implementamos as páginas relativas à exibição de produtos ofertados pela Alura Delivery e a que conterá os que forem selecionados pelo usuário/cliente. Entretanto, na página de produtos ofertados, temos apenas um texto que a identifica. Precisamos que os produtos sejam exibidos, com dados que subsidiem o usuário na escolha.

**[ Problema 1 ]**

A aplicação precisará ter acesso aos dados que compõem cada produto que será apresentado ao usuário, o que remete a um modelo de negócio, que é o foco a ser trabalhado agora.

**[ Solução 1 ]**

Seguindo a arquitetura em pastas que estamos utilizando, criamos em **lib**, uma pasta chamada **domain**, nela outra, chamada **models** e dentro desta última um arquivo chamado **produto\_model.dart**.



import 'package:meta/meta.dart';

class ProdutoModel *{*

final int produtoId;

final String nome;

final String descricao;

final double valor;

final String urlImagem;

ProdutoModel*({*

@required this.produtoId,

@required this.nome,

@required this.descricao,

@required this.valor,

@required this.urlImagem,

*})*;

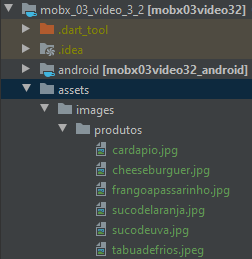
*}*

**[ Problema 2 ]**

Com o modelo de negócio implementado, já é possível obter os dados que popularão a listagem de produtos ofertados pelo Alura Delivery. Como não trabalhamos com base de dados neste curso, nosso problema está em providenciar uma fonte de dados baseada em objetos da classe de modelo.

**[ Solução 2 ]**

Como em nosso modelo temos uma propriedade que representa o endereço de uma imagem, que será exibida em conjunto aos demais dados e, estes serão fornecidos pelo aplicativo. Teremos os arquivos de imagens na estrutura física do projeto.



Para que estas imagens sejam acessíveis pela aplicação, precisamos registrar esta liberação no arquivo **pubspec.yaml**, dentro da categoria **flutter:**.

flutter:

uses-material-design: true

assets:

- assets/images/produtos/

Com as imagens copiadas para o projeto e registrada suas liberações, criamos em **lib** uma pasta chamada **data**, dentro dela uma chamada **datasources** e, nesta pasta, um arquivo chamado **produtos\_datasource.dart**.



import 'package:mobx03video32/domain/models/produto\_model.dart';

List*<*ProdutoModel*>* \_listaDeProdutos = *[*

ProdutoModel*(*

produtoId: 1,

nome: 'Suco de Laranja',

descricao: 'Suco natural (300ml)',

valor: 4,

urlImagem: 'assets/images/produtos/sucodelaranja.jpg'*)*,

ProdutoModel*(*

produtoId: 2,

nome: 'Suco de Uva',

descricao: 'Poupa da fruta (300ml)',

valor: 4,

urlImagem: 'assets/images/produtos/sucodeuva.jpg'*)*,

ProdutoModel*(*

produtoId: 3,

nome: 'Frango a passarinho',

descricao: 'Porção de 800g de frango a passarinho',

valor: 15,

urlImagem: 'assets/images/produtos/frangoapassarinho.jpg'*)*,

ProdutoModel*(*

produtoId: 4,

nome: 'Tábua de frios',

descricao:

'800g distribuídos em queijo, salame, ovo de cordona, presunto, tomate e azeitona',

valor: 20,

urlImagem: 'assets/images/produtos/tabuadefrios.jpeg'*)*,

ProdutoModel*(*

produtoId: 5,

nome: 'Cheese Salad',

descricao:

'Sanduíche com pão baguete, hamburguer de 150g, queijo, alface, tomate 50g de fritas',

valor: 10,

urlImagem: 'assets/images/produtos/cheeseburguer.jpg'*)*,

*]*;

Com os dados a serem consumidos implementados, codificamos um método que retorna estes dados, classificados pelo nome, como uma lista não modificável

List*<*ProdutoModel*>* listaDeProdutos*() {*

\_listaDeProdutos

.sort*((*a, b*)* => a.nome.toLowerCase*()*.compareTo*(*b.nome.toLowerCase*()))*;

List*<*ProdutoModel*>* \_unmodifiabileList = List.unmodifiable*(*\_listaDeProdutos*)*;

return \_unmodifiabileList;

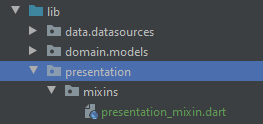
*}*

**[ Problema 3 ]**

Temos os dados que serão exibidos como produtos ofertados pela Alura Delivery. Nos resta agora implementar na página que tem esta funcionalidade, uma visão que os apresente de maneira que pareça uma listagem de produtos.

**[ Solução 3 ]**

Começamos a resolução criando recursos que possibilitam um bom reuso de código e minimização de redundâncias. O primeiro recurso será um método, implementado em um mixin, para possibilitar a renderização de widgets Text. Para isso, começamos criando uma pasta em **presentation**, chamada **mixins** e nela, um arquivo chamado **presentation\_mixin.dart**.



import 'package:flutter/material.dart';

mixin PresentationMixin *{*

gerarText*({*

String texto,

bool negrito = false,

double tamanhoFonte = 20,

Color cor = Colors.*black*,

*}) {*

return Text*(*

texto,

style: TextStyle*(*

fontWeight: negrito ? FontWeight.*bold* : FontWeight.*normal*,

fontSize: tamanhoFonte,

color: cor,

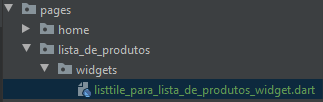
*)*,

*)*;

*}*

*}*

Na sequência, para a exibição dos produtos, implementamos um **ListView**, fazendo uso de **ListTile** para representar cada produto. Porém, buscando sempre uma boa coesão e boa legibilidade dos códigos, criamos na pasta **presentation.pages.lista\_de\_produtos**, uma nova, chamada **widgets** e, dentro dela um arquivo chamado **listtile\_para\_lista\_de\_produtos\_widget.dart**.



import 'package:flutter/material.dart';

import 'package:mobx03video32/domain/models/produto\_model.dart';

import 'package:mobx03video32/presentation/mixins/presentation\_mixin.dart';

class ListTileParaListaDeProdutosWidget extends StatelessWidget

with PresentationMixin *{*

final ProdutoModel produtoModel;

ListTileParaListaDeProdutosWidget*({*

@required this.produtoModel,

*})*;

@override

Widget build*(*BuildContext context*) {*

return ListTile*(*

// Imagem para cada produto

leading: CircleAvatar*(*

radius: 25,

backgroundImage: AssetImage*(*produtoModel.urlImagem*)*,

*)*,

// Linha com dados sobre o produto

title: Row*(*

children: *[*

// Renderização do nome e descrição para o produto

Expanded*(*

child: Column*(*

crossAxisAlignment: CrossAxisAlignment.start,

children: *[*

gerarText*(*texto: produtoModel.nome, negrito: true*)*,

gerarText*(*texto: produtoModel.descricao, tamanhoFonte: 15*)*,

*]*,

*)*,

*)*,

// Valor para o produto

gerarText*(*

texto: produtoModel.valor.toString*()*,

negrito: true,

cor: Colors.*blue)*,

*]*,

*)*,

// Botão que possibilitará o registro do produto no pedido do usuário

trailing: Column*(*

children: *[*

InkWell*(*

child: Icon*(*

Icons.*add*,

size: 48,

*)*,

onTap: *() {})*,

*]*,

*)*,

*)*;

*}*

*}*

Enfim, com os artefatos implementados, adaptamos a página de produtos disponíveis para renderizar a listagem deles.

// Criação de um apelido para a o código importado de produtos\_datasource.dart

import '../../../data/datasources/produtos\_datasource.dart' as datasource;

// Reescrita do método build da classe ListaDeProdutosPage

@override

Widget build*(*BuildContext context*) {*

List*<*ProdutoModel*>* produtos = datasource.listaDeProdutos*()*;

return ListView.builder*(*

padding: EdgeInsets.symmetric*(*vertical: 8*)*,

itemCount: produtos.length,

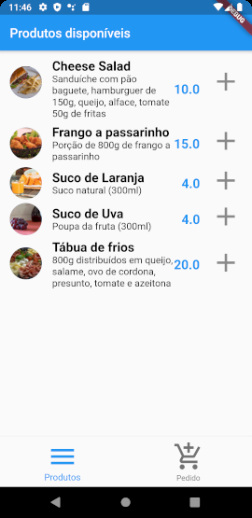
itemBuilder: *(*context, index*) {*

return ListTileParaListaDeProdutosWidget*(*

produtoModel: produtos*[*index*])*;

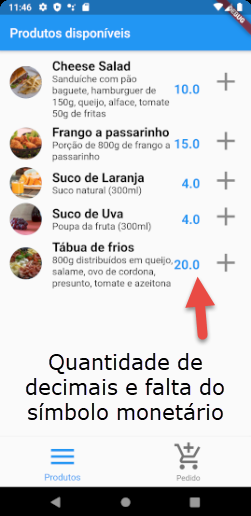
*})*;

*}*



**[ Problema 4 ]**

Temos os produtos devidamente exibidos, porém, os valores, embora sejam facilmente compreensíveis, é interessante que sejam tratados como valores monetários, exibindo o símbolo monetário e formando corretamente os valores.



**[ Solução 4 ]**

Trazemos para a aplicação um componente especializado na formatação de valores e o utilizaremos para formatar corretamente o valor dos produtos, que é o **intl**, registrado no **pubspec.yaml**.

dependencies:

flutter:

sdk: flutter

flutter\_mobx: ^1.1.0

intl: ^0.16.1

// Declaração implementada no início da classe ListTileParaListaDeProdutosWidget

final formatacaoMonetaria = NumberFormat.simpleCurrency*(*locale: 'pt\_BR'*)*;

// Adaptação da propriedade texto na invocação a gerarText para o valor do produto

texto: formatacaoMonetaria.format*(*produtoModel.valor*)*,



**[ Teoria ]**

Sempre que se trabalha com dados, de qualquer fonte que sejam, é preciso que eles sejam modelados para classes da aplicação em que serão consumidos e, para resolver esta situação, criamos o arquivo **produto\_model.dart**, na pasta **/lib/domain/models**, cuja responsabilidade é específica para esta funcionalidade.

Toda imagem que deva ser acessível à aplicação, precisa ter este registro no **pubspec.yaml**, caso contrário, ao utilizá-la, além de não ser exibida, um erro será gerado durante a execução, por isso realizamos o registro.

Implementamos uma classe que servirá como fonte de dados, e nela implementamos um **List**, com vários objetos de **ProdutoModel**.

Para que os dados possam ser consumidos, implementamos um método que retorna uma lista não modificável deles, classificados pelo nome de cada produto.

Temos a necessidade cada vez maior de renderizarmos Texts e, em muitas vezes, esse processo tem valores comuns e implementar diretamente o Text, pode ser que o código fique muito verboso, poluído até. Fazemos uso de Mixin aqui, delegando a ele métodos que sejam comuns em toda a camada de apresentação.

Para minimizar a verbosidade no método **build()** da página, criamos um widget para renderizar cada item a ser exibido no **ListView**.

Implementamos a localização para português brasileiro para que os valores monetários sejam devidamente exibidos, fazendo uso do pacote **intl**.

Com a página de produtos disponíveis implementada, surge agora a necessidade de implementar a seleção de um produto e sua quantidade no pedido do usuário, e é o que faremos na sequência.

### Vídeo 3.3 - Registro de seleção de produto

**[ Contexto ]**

Concluímos a etapa de exibição de produtos disponíveis pelo Alura Delivery, porém, nosso usuário ainda não pode selecioná-lo, pois não implementamos esta funcionalidade, que estará ligada à interação do usuário com o ícone de adição, ao lado do preço de cada produto. Precisaremos possibilitar ao usuário a possibilidade de informar a quantidade para cada item selecionado e, após o registro, que o total para o pedido seja exibido.

**[ Problema 1 ]**

A cada seleção de um produto, teremos a atualização do valor para o pedido e precisaremos exibir este valor ao nosso usuário. Ainda não temos isso em nossa interface, mas implementaremos isso agora.

**[ Solução 1 ]**

// Código a ser implementado na classe \_HomePageStore

@observable

double \_totalPedido = 0;

@computed

double get totalPedido => \_totalPedido;

@action

somarAoTotalDoPedido*({*double valor*}) {*

\_totalPedido += valor;

*}*

Precisamos que o valor total do pedido seja exibido com o símbolo monetário e casas decimais devidamente configuradas, tal qual fizemos para os produtos disponíveis.

// Declaração implementada no início da classe HomePage

final formatacaoMonetaria = NumberFormat.simpleCurrency*(*locale: 'pt\_BR'*)*;

Para que o total seja exibido, implementamos esta funcionalidade nas **actions** do **AppBar** e aqui faremos uso do método **gerarText()**, que implementamos no **PresentationMixin**.

// Declaração para o uso do PresentationMixin em HomePage

class HomePage extends StatelessWidget with PresentationMixin *{*

actions: *[*

Column*(*

mainAxisAlignment: MainAxisAlignment.center,

children: *[*

Text*(*'Total Pedido:'*)*,

Observer*(*

builder: *(*\_*) {*

return gerarText*(*

texto:

formatacaoMonetaria.format*(*\_homePageStore.totalPedido*)*,

negrito: true,

cor: Colors.*yellowAccent*,

*)*;

*}*,

*)*,

*]*,

*)*,

*]*,



**[ Problema 2 ]**

Temos o label DEBUG sobrepondo a informação para o total do pedido, precisamos resolver isso.



**[ Solução 2 ]**

// Configuração de propriedade no MaterialApp, na classe MyApp no arquivo main.dart

debugShowCheckedModeBanner: false,



**[ Problema 3 ]**

Agora que temos o total do pedido sendo exibido, precisamos dar ao usuário a possibilidade de selecionar os produtos desejados e é o que resolveremos agora.

**[ Solução 3 ]**

Adotamos na implementação desta solução a exibição de uma janela de diálogo com quantidades possíveis de serem selecionadas para os itens selecionados. Fazemos aqui o uso de um pacote, o **numberpicker**.

*# registro do pacote numberpicker no pubspec.yaml, em dependencies, seguido do Pub get*

numberpicker: ^1.2.0

Criamos uma pasta chamada **mixins** na pasta **/lib/presentation/pages/lista\_de\_produtos** e, nesta pasta, criamos o arquivo **lista\_de\_produtos\_page\_mixin.dart**.

import 'package:flutter/cupertino.dart';

import 'package:flutter/material.dart';

import 'package:numberpicker/numberpicker.dart';

mixin ListaDeProdutosPageMixin *{*

Future*<*int*>* exibirSelecaoDeQuantidade*(*

*{*BuildContext context, String item*})* async *{*

return await showDialog*<*int*>(*

barrierDismissible: false,

context: context,

builder: *(*BuildContext context*) {*

return NumberPickerDialog.integer*(*

cancelWidget: Text*(*'Cancelar'*)*,

confirmWidget: Text*(*'Confirmar'*)*,

initialIntegerValue: 1,

minValue: 1,

maxValue: 10,

title: new Text*(*

"Quantidade para \n*${*item.toUpperCase*()}*",

textAlign: TextAlign.center,

*)*,

*)*;

*}*,

*)*;

*}*

*}*

**[ Problema 4 ]**

Já temos a funcionalidade para a seleção de produtos e informação de quantidade para eles, mas ainda não temos a invocação ao método implementado e é o que criaremos agora.

**[ Solução 4 ]**

// Declaração de uma nova propriedade para widget ListTileParaListaDeProdutosWidget

final Function funcaoDeCallbackParaInserirProduto;

// Atribuição do argumento à nova propriedade no construtor de ListTileParaListaDeProdutosWidget

@required this.funcaoDeCallbackParaInserirProduto,

// Atribuição da função recebida ao onTap de InkWell no widget ListTileParaListaDeProdutosWidget

onTap: funcaoDeCallbackParaInserirProduto,

// Declaração da classe ListaDeProdutosPage para fazer uso do mixin ListaDeProdutosPageMixin

class ListaDeProdutosPage extends StatelessWidget with ListaDeProdutosPageMixin *{*

// Envio de uma função anônima para o widget ListTileParaListaDeProdutosWidget

funcaoDeCallbackParaInserirProduto: *()* async *{*

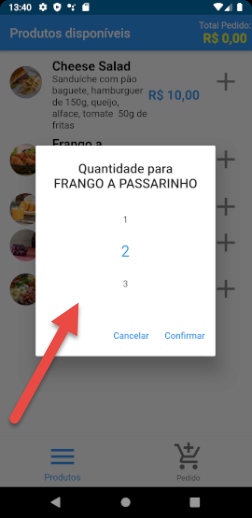
int quantidade = await exibirSelecaoDeQuantidade*(*

context: context, item: produtos*[*index*]*.nome*)*;

if *(*quantidade != null*)*

print*(*'Você selecionou $quantidade *${*produtos*[*index*]*.nome*}*'*)*;

*}*,





**[ Problema 5 ]**

Implementamos a seleção de produto e sua respectiva quantidade, mas precisamos que esta escolha seja utilizada para atualização do total do pedido exibido na **HomePage**. Vamos começar agora a resolver este problema.

**[ Solução 5 ]**

Criamos uma pasta chamada **mobx na** pasta **/lib/presentation/pages/lista\_de\_produtos** e, nesta pasta, criamos o arquivo **lista\_de\_produtos\_page\_store.dart**.

import 'package:mobx/mobx.dart';

import 'package:mobx03video33/domain/models/produto\_model.dart';

part 'lista\_de\_produtos\_page\_store.g.dart';

class ListaDeProdutosPageStore = \_ListaDeProdutosPageStore

with \_$ListaDeProdutosPageStore;

abstract class \_ListaDeProdutosPageStore with Store *{*

@action

registrarProduto*({*ProdutoModel produto, int quantidade*}) {}*

*}*

// Declaração do store para a lista de produtos na página ListaDeProdutosPage

final ListaDeProdutosPageStore \_listaDeProdutosPageStore =

ListaDeProdutosPageStore*()*;

// Adaptação da confirmação de seleção de produto para invocar o método de registro

if *(*quantidade != null*)*

\_listaDeProdutosPageStore.registrarProduto*(*

produto: produtos*[*index*]*, quantidade: quantidade*)*;

**[ Problema 6 ]**

Temos o registro do produto selecionado já implementado, mas o total do pedido ainda não está sendo atualizado. Isso ocorre devido ao fato de esta atualização estar presente às responsabilidades da HomePage. Precisamos invocar estas funcionalidades no Store específico para a listagem do produto. Faremos isso agora, concluindo este problema.

**[ Solução 6 ]**

Precisamos ter acesso à instância de **HomePageStore** em **ListaDeProdutosStore**, o que remete à necessidade de termos esta instância disponibilizada em toda a aplicação. Para isso, registramos o pacote **get\_it** nas dependências de **pubspec.yaml**.

*# registro do pacote get\_it no pubspec.yaml, em dependencies, seguido do Pub get*

get\_it: ^4.0.2

// Registro da instância de HomePageStore no contexto da aplicação fazendo uso de GetIt no método main() de main.dart

GetIt.*I*.registerSingleton*<*HomePageStore*>(*HomePageStore*())*;

// Adaptação da declaração de \_homePageStore em HomePage para recuperar a instância de HomePageStore

final HomePageStore \_homePageStore = GetIt.*instance*.get*<*HomePageStore*>()*;

// Adaptação da action registrarProduto para invocar o método somarAoTotalDoPedido(), de HomePageStore, a seleção do usuário

@action

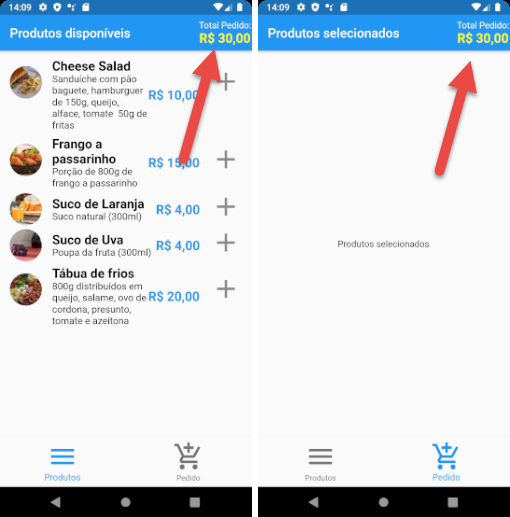
registrarProduto*({*ProdutoModel produto, int quantidade*}) {*

GetIt.*I*

.get*<*HomePageStore*>()*

.somarAoTotalDoPedido*(*valor: produto.valor \* quantidade*)*;

*}*



**[ Teoria ]**

Para que o total do pedido seja exibido tanto na página de produtos disponíveis, como na de produtos selecionados, na classe **\_HomePageStore**, que atenderá a **HomePage**, que é responsável por exibir as duas páginas, implementamos propriedades e actions para esta funcionalidade.

A propriedade **actions** de **AppBar** possibilita a inserção de uma lista de widgets, que normalmente são utilizados para uma interação do usuário. Como qualquer widget é aceito nesta lista, adotamos a exibição do total do pedido, fazendo uso de **Text**.

Como estamos executando a aplicação dentro do Android Studio, normalmente isso ocorre em processo de depuração. O Flutter, neste caso, insere o texto **DEBUG** no topo direito da página. Esta informação não é exibida quando o App é distribuído em sua versão final. Para nos ajudar, forçamos a não exibição deste texto. Essa informação é válida também se estiver utilizando o Visual Studio Code.

Seguindo o padrão de responsabilidades e legibilidade de código, criamos o método **exibirSelecaoDeQuantidade()** em um mixin, onde é feito uso do widget **NumberPickerDialog**, do pacote **numberpicker**.

Para a invocação do método **exibirSelecaoDeQuantidade()**, adotamos a estratégia de enviar uma função anônima ao widget ListTileParaListaDeProdutosWidget, que tratará de executá-la quando o ícone de adição do produto sofrer interação.

O método **exibirSelecaoDeQuantidade()**, retorna a quantidade selecionada, mas se o usuário pressionar o botão Cancelar, um valor nulo é retornado. Validamos isso e exibimos, por enquanto, no console, a seleção completa do usuário.

Como o fato de registrar a seleção de um produto está diretamente ligado à página que exibe os produtos, implementamos um Store específico para esta página e nele, o métod que terá a responsabilidade deste registro.

Tivemos a necessidade de trabalharmos, em classes diferentes, uma mesma instância de uma classe, no caso o **HomePageStore**, responsável pela gerência de estado de HomePage. Isso remeteu ao uso do pacote **get\_it**, que atua como um Service Locator (localizador de serviços). Registramos a dependência do pacote e, com ele disponibilizado, o utilizamos para registrar uma instância de **HomePageStore** no contexto da aplicação. No momento do registro da seleção do produto escolhido pelo usuário, recuperamos o objeto e o utilizamos para atualizar o total do pedido.

**[ Fechamento ]**

Com o registro do produto escolhido pelo usuário, para atualizar o total do pedido implementado. Vamos, na sequência, implementar uma funcionalidade orientativa ao usuário, que é informar a ele onde deverá visualizar o total do pedido.

## Aula 4 - Animação para localização de infos

### Vídeo 4.1 - Balão de Orientação

**[ Contexto ]**

Sempre que temos a situação de utilização de um novo aplicativo, é importante que ele nos forneça orientações que nos auxiliem em seu uso e localização de informações. Desta maneira, desenvolveremos, na sequência, recursos visuais que possibilitem que nosso usuário identifique a localização do total do pedido.

**[ Problema 1 ]**

Uma maneira descontraída para exibir uma orientação ao usuário é tê-la em um balão, do estilo história em quadrinho. Implementaremos a geração deste balão como solução para este problema.

**[ Solução 1 ]**

Para o desenho do balão, fazemos uso de um pacote, o **clippy\_flutter**, que registramos nas dependências de nosso projeto, no arquivo **pubspec.yaml**.

*# registro do pacote clippy\_flutter no pubspec.yaml, em dependencies, seguido do Pub get*

clippy\_flutter: ^1.1.1

Para implementarmos a renderização do balão informativo, desenvolvemos um widget, com essa responsabilidade. Dessa maneira, na pasta **/lib/presentation/pages/home**, criamos outra, chamada **widgets**, e nela, o arquivo **clippy\_widget.dart**.

import 'package:clippy\_flutter/clippy\_flutter.dart';

import 'package:flutter/material.dart';

class ClippyWidget extends StatelessWidget *{*

final double comprimentoCaixaDeMensagem = 200.0;

final double alturaCaixaDeMensagem = 120.0;

@override

Widget build*(*BuildContext context*) {*

return Column*(*

mainAxisAlignment: MainAxisAlignment.center,

children: *<*Widget*>[*

Message*(*

triangleX1: 0.4 \* comprimentoCaixaDeMensagem,

triangleX2: 0.5 \* comprimentoCaixaDeMensagem,

triangleX3: 0.5 \* comprimentoCaixaDeMensagem,

triangleY1: alturaCaixaDeMensagem \* 0.2,

clipShadows: *[*ClipShadow*(*color: Colors.*black)]*,

child: Container*(*

color: Colors.*pink*,

width: comprimentoCaixaDeMensagem,

height: alturaCaixaDeMensagem,

child: Padding*(*

padding: EdgeInsets.only*(*

bottom: alturaCaixaDeMensagem \* 0.2, left: 8, right: 8*)*,

child: Column*(*

mainAxisAlignment: MainAxisAlignment.center,

children: *[*

Text*(*

'Veja sempre o total do pedido, no topo direito da tela.',

textAlign: TextAlign.justify,

style: TextStyle*(*

color: Colors.*white*,

fontWeight: FontWeight.*bold*,

*)*,

*)*,

*]*,

*)*,

*)*,

*)*,

*)*,

*]*,

*)*;

*}*

*}*

**[ Problema 2 ]**

Já temos implementado o widget para renderização do balão, mas ele ainda não está sendo utilizado. Precisamos ajustar esta invocação na **HomePage** e é o que faremos na sequência.

**[ Solução 2 ]**

Adaptamos o **Observer** de **body**, do **Scaffold**, em **HomePage**, para utilizar um **Stack**, que representa uma pilha de widgets sobrepostos, para que o balão possa ser renderizado.

// Adaptação do builder do Observer do body de Scaffold para utilizar o Stack

return Stack*(*

children: *[*

\_paginas*[*\_homePageStore.paginaAtual*]*,

Positioned*(*

right: 0,

width: 250,

top: 10,

child: ClippyWidget*()*,

*)*,

*]*,

*)*;



**[ Problema 3 ]**

Com o balão desenhado, queremos dar a ele a impressão de uma narração, o que nos leva a necessidade de termos um mascote. Vamos implementar isso para este problema.

**[ Solução 3 ]**

Criamos uma nova pasta de imagens, dentro de **assets/images/** chamada **home** e nela copiamos o arquivo para o mascote, e registramos no **pubspec.yaml** o acesso a esta pasta.

*#Registro para acesso a pasta assets/images/home/*

assets:

- assets/images/home/

- assets/images/produtos/

No widget **ClippyWidget** que implementamos, já fazemos uso de **Column** para exibir o balão, o que nos permite a inserção de componentes abaixo dele.

// Inserção de um espaço entre a mensagem e o mascote

SizedBox*(*height: 8.0*)*,

// Inserção do mascote

SizedBox*(*

width: 200.0,

child: Align*(*

alignment: Alignment.*centerRight*,

child: Image.asset*(*

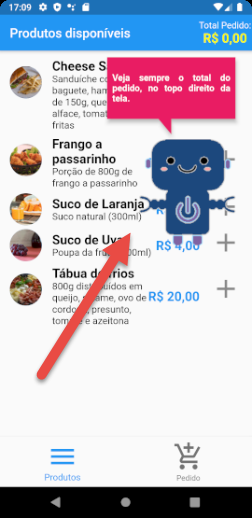
'assets/images/home/clippy.png',

width: 150.0,

*)*,

*)*,

*)*,



**[ Teoria ]**

O pacote **flutter\_clippy** tem diversos widgets que podem ser utilizados para uma interface descontraída e útil para aplicações. Aqui utilizamos o Message, que realiza um corte em Container, que dará a ele o formato de um balão. O uso de sombreamento no balão também é possível por meio de outro widget do pacote, o **ClipShadow**.

O **body** da **HomePage**, agora com mais de um widget a ser exibido nele, mas precisando ser um sobre o outro, precisamos adaptar para o uso de **Stack**, que possibilita esta funcionalidade. Como o balão precisa estar em uma posição específica da página, utilizamos o widget **Positioned**, definindo o ponto onde queremos que o balão seja renderizado.

Para a inserção do mascote, também temos a situação de pilha de widgets, mas agora, um após o outro, por isso o uso de **Column**.

Na sequência trabalharemos uma animação apontando o local do total do pedido, buscando chamar a atenção do usuário para isso.

### Vídeo 4.2 - Flecha animada

**[ Contexto ]**

Temos uma orientação escrita para nosso usuário, dizendo onde ele encontrará sempre o total do pedido. Um apelo visual maior será uma animação, com uma seta apontando onde, visualmente, esta informação está situada.

**[ Problema 1 ]**

Para desenharmos uma animação com uma flecha em nossa HomePage, precisamos antes de tudo ter um widget que representa esta flecha e o implementaremos na sequência.

**[ Solução 1 ]**

Na pasta **/lib/presentation/pages/home/widgets** criamos um arquivo chamado **flecha\_widget.dart**.

import 'package:clippy\_flutter/clippy\_flutter.dart';

import 'package:flutter/material.dart';

class FlechaWidget extends StatelessWidget *{*

final double esquerda;

const FlechaWidget*({*this.esquerda*})*;

@override

Widget build*(*BuildContext context*) {*

return Container*(*

width: double.*infinity*,

height: 50,

child: Stack*(*

children: *[*

Positioned*(*

left: esquerda,

child: Arrow*(*

clipShadows: *[*ClipShadow*(*color: Colors.*black)]*,

triangleHeight: 20.0,

rectangleClipHeight: 10.0,

child: Container*(*

width: 50,

height: 50,

color: Colors.*yellowAccent*,

*)*,

*)*,

*)*,

*]*,

*)*,

*)*;

*}*

*}*

**[ Problema 2 ]**

Temos um widget que renderiza a flecha, mas precisamos exibi-la e é o que faremos na sequência.

**[ Solução 2 ]**

Adaptamos a propriedade title de AppBar do Scaffold de HomePage para, ao invés de exibir o título, exibir provisoriamente a flecha que deverá ser animada.

// Adaptação do title de AppBar, de Scaffold da HomePage para exibição da flecha

title: Observer*(*builder: *(*\_*) {*

return FlechaWidget*(*

esquerda: 10,

*)*;

*})*,

**[ Problema 3 ]**

Já conseguimos ver a imagem da flecha que será utilizada na animação, que apontará o local do total do pedido. Mas ela está estática e nós precisamos fazer com que ela se movimente em direção ao título. É esta funcionalidade que implementaremos agora.

**[ Solução 3 ]**

Transformamos a HomePage em Stateful Widget, pois é um pré-requisito para que animações sejam possíveis.

// Transformação da HomePage para StatefulWidget

class HomePage extends StatefulWidget *{*

@override

\_HomePageState createState*()* => \_HomePageState*()*;

*}*

class \_HomePageState extends State*<*HomePage*>* with PresentationMixin *{*

*// Registramos o uso do Mixin SingleTickerProviderStateMixin na classe State para HomePage*

*class \_HomePageState extends State<HomePage> with PresentationMixin, SingleTickerProviderStateMixin {*

*// Declaração das variáveis que serão utilizadas na animação, dentro da classe \_HomePageState*

*Animation<double> \_animacao;*

*AnimationController \_controladorAnimacao;*

*// Sobrescrita do método initState() para instanciar classes responsáveis pela animação e controle dela*

*@override*

*void initState() {*

*super.initState();*

*\_controladorAnimacao =*

*AnimationController(duration: const Duration(seconds: 3), vsync: this);*

*\_animacao =*

*Tween<double>(begin: -50, end: 300).animate(\_controladorAnimacao)*

*..addStatusListener((status) {*

*if (status == AnimationStatus.completed) {*

*\_controladorAnimacao.reset();*

*\_controladorAnimacao.forward();*

*}*

*});*

*}*

*// Sobrescrita do método dispose() na classe \_HomePageState, para liberar recursos utilizados para o controlador da animação*

*@override*

*void dispose() {*

*\_controladorAnimacao.dispose();*

*super.dispose();*

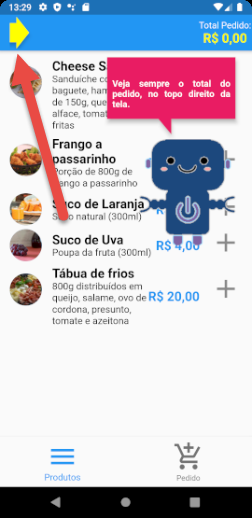
*}*

*// Adaptação do parâmetro enviado para FlechaWidget para que faça uso dos valores gerados para a animação*

*return FlechaWidget(*

*esquerda: \_animacao.value,*

*);*

**

**[ Problema 4 ]**

Temos a animação declarada, instanciada, iniciada e sendo gerenciada, mas não vemos ela ocorrendo na aplicação. Precisamos resolver isso e é o que faremos.

**[ Solução 4 ]**

Na pasta **/lib/presentation/pages/home/widgets**, criamos o arquivo **animacao\_flecha\_widget.dart**, responsável propiciar o efeito da animação.

import 'package:flutter/material.dart';

import 'flecha\_widget.dart';

class AnimacaoFlechaWidget extends AnimatedWidget *{*

AnimacaoFlechaWidget*({*Animation*<*double*>* animacao*})* : super*(*listenable: animacao*)*;

Widget build*(*BuildContext context*) {*

final animacao = listenable as Animation*<*double*>*;

return FlechaWidget*(*

esquerda: animacao.value,

*)*;

*}*

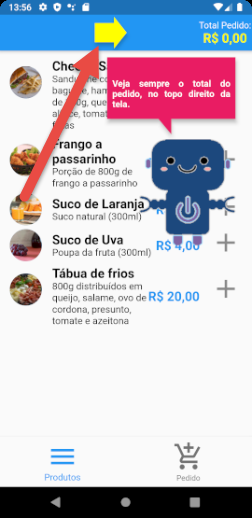
*}*

*// Mudança do widget a ser utilizado para o título*

*return AnimacaoFlechaWidget(*

*animacao: \_animacao,*

*);*

**

**[ Teoria ]**

Ainda fazendo uso do pacote **flutter\_clippy** implementamos um widget que faz uso de **Arrow**, widget do pacote, para renderizar uma flecha.

Para podermos visualizar a flecha gerada pelo widget **FlechaWidget**, adaptamos o **title** do AppBar de **Scaffold** da **HomePage** para exibir a flecha.

Uma animação em uma página exige a instância de algumas classes que precisam ser inicializadas (**Animation** e **AnimationController**) e uso de um Mixin específico para poder tratar responsabilidades para as animações (**SingleTickerProviderStateMixin**). Devido a isso transformamos a página em um Stateful.

Utilizamos a variável **\_animacao**, responsável pela animação a ser realizada e a **\_controladorAnimacao**, que controlará a execução da animação. A animação escolhida foi a **Tween**, que gera valores entre um intervalo, valores estes que serão utilizados no posicionamento da flecha. Um método monitorador (**listener**) é adicionado à a animação para identificar quando ela é completada, para que recomece, dando assim a continuidade dela, até que haja a confirmação de leitura da orientação por parte do usuário.

Ajustamos o valor para a posição da flecha, enviando **\_animacao.value** e iniciamos a animação no método **initState()**.

Utilizamos uma classe específica (um widget) para gerenciar o estado da animação, tirando a responsabilidade da **HomePage**. Criamos então o widget **AnimacaoFlechaWidget** e adaptamos a renderização do título para utilizarmos ele.

**[ Fechamento ]**

Implementamos a exibição da orientação e de uma animação para apontar o local, na interface com o usuário, em que está a informação descrita na orientação. Tivemos que tirar o título para que a animação fosse exibida. Ainda, com a orientação sendo exibida, o usuário ainda pode interagir com o ícone de seleção de produto. Precisamos ter na mensagem um botão para confirmação, proibição de interação e o retorno do título após a confirmação da leitura. Veremos tudo isso na sequência.

## Aula 5 - Confirmação de leitura das infos

### Vídeo 5.1 - Restrição de interação com controles

**[ Contexto ]**

Já temos nossa mensagem orientativa em exibição, dando a impressão de ser uma narração de um mascote e também temos uma flecha animada, apontado para o local do total do pedido. Ocorre, que mesmo utilizando o **Stack**, fazendo com que estes componentes fiquem sobre a listagem de produtos, ainda é possível ao usuário interagir com o ícone de adicionar produtos, o que não é desejado.

**[ Problema 1 ]**

A orientação está sendo exibida conforme pretendíamos, mas temos um problema com isso. Não queremos que o usuário possa registrar um pedido quando ela esteja sendo exibida. Na sequência resolveremos isso.

**[ Solução 1 ]**

Encapsulamos a renderização da página em exibição na **HomePage** para ter uma opacidade, com o widget **Opacity**, e bloqueamos o acesso aos widgets dela por meio do widget **AbsorbPointer**.

// Adaptação da renderização da página em exibição no body de HomePage

AbsorbPointer*(*

child: Opacity*(*

opacity: 0.3,

child: \_paginas*[*\_homePageStore.paginaAtual*]*,

*)*,

*)*,



**[ Problema 2 ]**

Demos ao usuário a sensação de conteúdo desabilitado ao visualizar a mensagem e proibimos seu acesso a este conteúdo. Mas o usuário precisa confirmar a leitura dessa mensagem e com isso precisamos garantir que esta mensagem desapareça. É isso que resolveremos na sequência.

**[ Solução 2 ]**

O texto para a mensagem faz parte de um **Column**, o que nos permite inserir um botão abaixo dele.

// Inserção de um botão abaixo do Text no widget ClippyWidget

FlatButton*(*

onPressed: funcaoParaRegistrarLeituraOrientacao,

child: Text*(*

"Ok. Entendi!",

style: TextStyle*(*color: Colors.*black)*,

*)*,

*)*,

// Com a inserção do botão, precisamos ajustar o padding para a coluna

child: Padding*(*

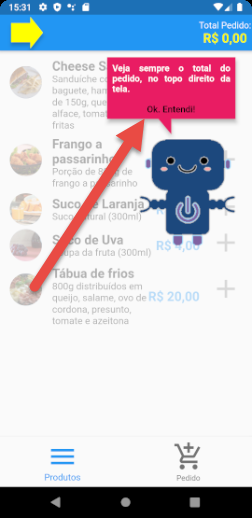
padding: EdgeInsets.only*(*

top: alturaCaixaDeMensagem \* 0.05,

bottom: alturaCaixaDeMensagem \* 0.1,

left: 8,

right: 8*)*,



// Código a ser implementado na classe \_HomePageStore

@observable

bool orientacaoJaLida = false;

@action

registrarLeituraOrientacao*() {*

orientacaoJaLida = true;

*}*

// Criação de um método que retornará o conteúdo a ser renderizado ém relação a página exibida em HomePage a ser implementado após o método \_bottomNavigationBarItem

\_conteudoAbaixoDaMensagemDoMascote*() {*

if *(*\_homePageStore.orientacaoJaLida*)*

return \_paginas*[*\_homePageStore.paginaAtual*]*;

else

return AbsorbPointer*(*

child:

Opacity*(*opacity: 0.3, child: \_paginas*[*\_homePageStore.paginaAtual*])*,

*)*;

*}*

// Adaptação da renderização da página em exibição no body de HomePage para utilizar o método \_conteudoAbaixoDaMensagemDoMascote()

\_conteudoAbaixoDaMensagemDoMascote*()*,

// Widget Positioned, com a mensagem e mascote encapsulado pelo Visibilty, para exibir apenas enquanto a mensagem não for lida

Visibility*(*

visible: !\_homePageStore.orientacaoJaLida,

child: Positioned*(*

right: 0,

width: 250,

top: 10,

child: ClippyWidget*(*

funcaoParaRegistrarLeituraOrientacao: *() {*

\_homePageStore.registrarLeituraOrientacao*()*;

*}*,

*)*,

*)*,

*)*,

// Registro da função anônima a ser enviada para o ClippyWidget executar quando o botão de confirmação for pressionado

child: ClippyWidget*(*

funcaoParaRegistrarLeituraOrientacao: *() {*

\_homePageStore.registrarLeituraOrientacao*()*;

*}*,

*)*,

**[ Problema 3 ]**

Já temos a opacidade sendo retirada quando o usuário confirma a leitura da orientação, mas nossa animação ainda continua sendo exibida e precisamos voltar a exibir o título da página em exibição pela HomePage. Trataremos isso agora.

**[ Solução 2 ]**

*// Criação de um método, após o \_conteudoAbaixoDaMensagemDoMascote(), que retornará o conteúdo a ser renderizado para o título*

*\_tituloAppBar() {*

*return Observer(builder: (\_) {*

*if (\_homePageStore.orientacaoJaLida)*

*return Text(\_homePageStore.tituloHomePage);*

*else*

*return AnimacaoFlechaWidget(*

*animacao: \_animacao,*

*);*

*});*

*}*

// Adaptação da propriedade title para fazer uso do método \_tituloAppBar()

title: \_tituloAppBar*()*,

**[ Teoria ]**

Enquanto o balão é exibido, orientando ao usuário, damos a ele a sensação de que o conteúdo da página está desabilitado, utilizando o widget **Opacity**. Ainda proibimos o acesso ao conteúdo, utilizando o widget **AbsorbPointer**.

Após a confirmação de leitura da orientação pelo usuário, registramos o processo em uma propriedade observável em **HomePageStore**, que servirá de sinalizadora para o que deve ser renderizado na HomePage, em relação ao balão, mascote e conteúdo liberado.

Para que o título da página exibida por **HomePage** torne a ser visível após a confirmação da leitura pelo usuário, criamos um método (**\_tituloAppBar()**) e o utilizamos na propriedade **title**.

Tudo funcionando como se espera, exceto pelo fato de que, se o usuário sair do App e retornar a abri-lo, a mensagem será novamente exibida, mesmo o usuário ter confirmado a leitura. Não é o que desejamos e vamos trabalhar isso na sequência.

### Vídeo 5.2 - Registro de leitura da orientação

**[ Contexto ]**

Temos garantido ao usuário que, durante a execução da aplicação, ele confirmando a leitura da mensagem orientativa, o balão e a flecha desaparecem e o acesso às interações na página de produtos se torna possível. Entretanto, é interessante que essa confirmação tenha validade permanente, mesmo que o usuário saia do App e retorne a usá-lo. Trabalharemos aqui o registro dessa confirmação de maneira física.

**[ Problema 1 ]**

Precisamos tornar permanente a confirmação de leitura da orientação sobre o total de pedidos e é o que resolveremos na sequência.

**[ Solução 1 ]**

Para que a confirmação do usuário possa ser permanente, precisamos registrá-la fisicamente e faremos isso por meio do pacote

*# registro do pacote shared\_preferences no pubspec.yaml, em dependencies, seguido do Pub get*

shared\_preferences: ^0.5.7+3

Criamos, na pasta **/lib/presentation/pages/home**, uma nova, chamada **shared\_preferences**, e nela, um arquivo chamado **orientacao\_total\_pedido\_preferences.dart**.

import 'package:shared\_preferences/shared\_preferences.dart';

class OrientacaoTotalPedidoPreferences *{*

static Future*<*void*> registrarLeituraOrientacao()* async *{*

final SharedPreferences prefs = await SharedPreferences.*getInstance()*;

prefs.setBool*(*'leituraConfirmada', true*)*;

*}*

static Future*<*bool*> verificarSeJaLeuOrientacao()* async *{*

final SharedPreferences prefs = await SharedPreferences.*getInstance()*;

return prefs.getBool*(*'leituraConfirmada'*)* ?? false;

*}*

*}*

// Adaptação da função anônima para que registre fisicamente a leitura da orientação na invocação ao widget ClippyWidget

funcaoParaRegistrarLeituraOrientacao: *()* async *{*

await OrientacaoTotalPedidoPreferences

.*registrarLeituraOrientacao()*;

\_homePageStore.registrarLeituraOrientacao*()*;

*}*,

// Adaptação do método main(), em main.dart para executar um método assíncrono, referente à leitura de ciência sobre o total de pedido, registro dela em HomePageStore e também da instância no contexto da aplicação

WidgetsFlutterBinding.*ensureInitialized()*;

HomePageStore \_homePageStore = HomePageStore*()*;

OrientacaoTotalPedidoPreferences.*verificarSeJaLeuOrientacao()*.then*((*value*) {*

if *(*value*)* \_homePageStore.registrarLeituraOrientacao*()*;

GetIt.*I*.registerSingleton*<*HomePageStore*>(*\_homePageStore*)*;

*})*;

Timer*(*Duration*(*seconds: 1*)*, *()* => runApp*(*MyApp*()))*;

**[ Teoria ]**

Shared Preferences (Preferências do Usuário) é um recurso disponível para aplicativos registrarem pequenos e poucos dados no contexto físico do aplicativo no dispositivo. Estes dados podem ser gravados, recuperados e removidos. Fazemos uso dele no Alura Delivery para registrarmos fisicamente a leitura da orientação sobre o total de pedido.

Para que, no início da aplicação, possamos fazer uma chamada assíncrona, para confirmação de leitura da orientação, por meio de Shared Preferences, fazemos uso de **WidgetsFlutterBinding.ensureInitialized();**, como recomendado por documentação do Flutter. Também não invocamos o método com **await**, por não podermos marcar **main()** como **async**. O fazemos por meio do **then()**, o que pode fazer com que a aplicação seja executada, por meio do **runApp()**, antes da leitura ser realizada. Por isso, fazemos uso de um delay (espera), utilizando o **Timer()**.

**[ Fechamento ]**

Implementamos fisicamente a confirmação dada pelo usuário para a leitura da mensagem orientativa, fazendo assim com que ela não seja mais exibida após isso.