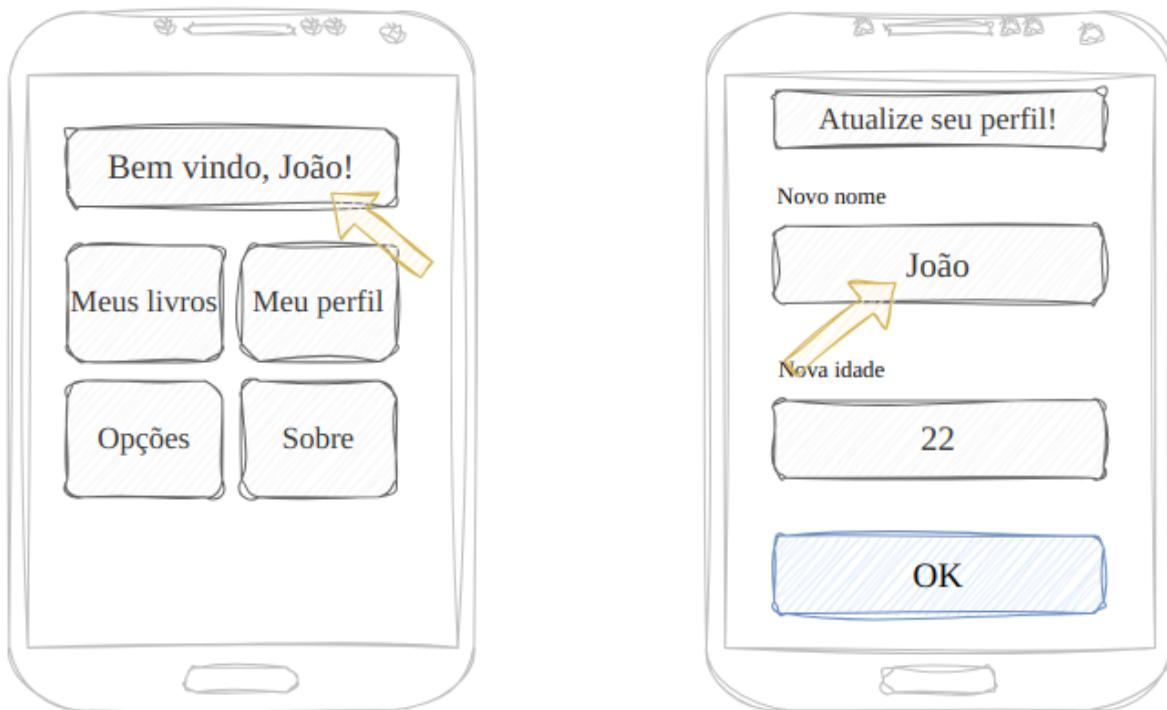


# 1 Introdução

Vimos que o **estado** de um Widget é caracterizado pela **coleção de variáveis cujos valores podem ser alterados enquanto o Widget tem vida útil** e que, em geral, são **utilizados na renderização de partes da tela**.

O Flutter oferece os conhecidos **Stateful Widgets**. Eles permitem o gerenciamento e manipulação de estado. Porém, seu funcionamento é muito simples e pode dar origem a aplicações difíceis de se manter, especialmente aplicações de nível de complexidade razoável.

Considere uma aplicação que tem apenas duas telas.



Observe que o nome do usuário é necessário para a exibição de ambas as telas. Potencialmente, ele faz parte do estado da aplicação. Uma pergunta a ser feita é a seguinte:

“Onde definir essa variável de estado?”

Uma resposta natural seria:

“Podemos criar um Widget com estado (**StatefulWidget**) em que fazemos a definição desta variável. Em função da navegação do usuário, construímos um Widget diferente (um para o Dashboard, outro para a atualização de perfil) e a ele entregamos, via construtor, o valor da variável de estado que lhe interessa.”

Essa abordagem funciona, mas ela se mostra demasiada complicada e difícil de manter, especialmente para aplicações mais complexas. Assim, neste material, vamos estudar sobre diversas opções para realizar o gerenciamento de estado em aplicações Flutter.

**Nota. Não há verdade absoluta a respeito do gerenciamento de estado de aplicações Flutter** (isso também vale para outros ambientes, como aplicações React). Há diferentes formas de se resolver o problema, cada qual com suas vantagens e desvantagens. Algumas das principais são as seguintes

- **método setState**: usado em conjunto com um StatelessWidget. É simples, provido pelo próprio framework Flutter, mas tende a comprometer a manutenibilidade de aplicações mais complexas.

- **pacote Provider**: É uma das bibliotecas mais populares para o gerenciamento de estado de aplicações Flutter. O **framework Flutter oferece** um Widget do tipo

**InheritedWidget**

<https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html>

cuja finalidade é permitir que **o estado de uma aplicação seja propagado para Widgets mais abaixo na árvore**. Um Provider é um “wrapper” sobre este tipo de Widget e traz um nível mais alto de abstração.

Visite-o no pub.dev

<https://pub.dev/packages/provider>

- **Riverpod** (observe que é um **anagrama** de provider): este pacote é baseado no provider e, segundo a sua documentação, oferece mais alto nível de abstração, mais flexibilidade e resolve problemas encontrados no uso do provider mais facilmente.

Visite-o no pub.dev

<https://pub.dev/packages/riverpod>

- **Bloc (Business Logic Component)** Esse é um padrão baseado em **eventos**. Regras de negócio e estado são separados da visualização. Eles se comunicam por meio de **streams**. Embora **possa ser implementado sem pacotes extras**, há também a possibilidade de fazê-lo utilizando pacotes, o que, em geral, promove a produtividade do desenvolvedor. Veja o site a seguir.

<https://bloclibrary.dev>

- **Redux**: Esse pacote é baseado no pacote de mesmo nome do ambiente Javascript. Ele define um estado global único a que todos os componentes interessados têm acesso e uma forma padronizada de atualizá-lo.

Visite-o no pub.dev

<https://pub.dev/packages/redux>

- **Mobx**: Este pacote oferece uma abordagem **reativa** para o gerenciamento de estado. A ideia é “observarmos” potenciais mudanças de estado e “reagirmos” a elas. Apesar do nome “impactante”, a programação reativa se baseia num padrão de projetos muito antigo e elementar: o padrão **Observer**. Tudo se baseia na existência de objetos “observáveis” e objetos “observadores”. Quando um observável sofre um evento de interesse, a coleção de observadores interessados neste evento é notificada e cada um reage da forma que acharpropriada.

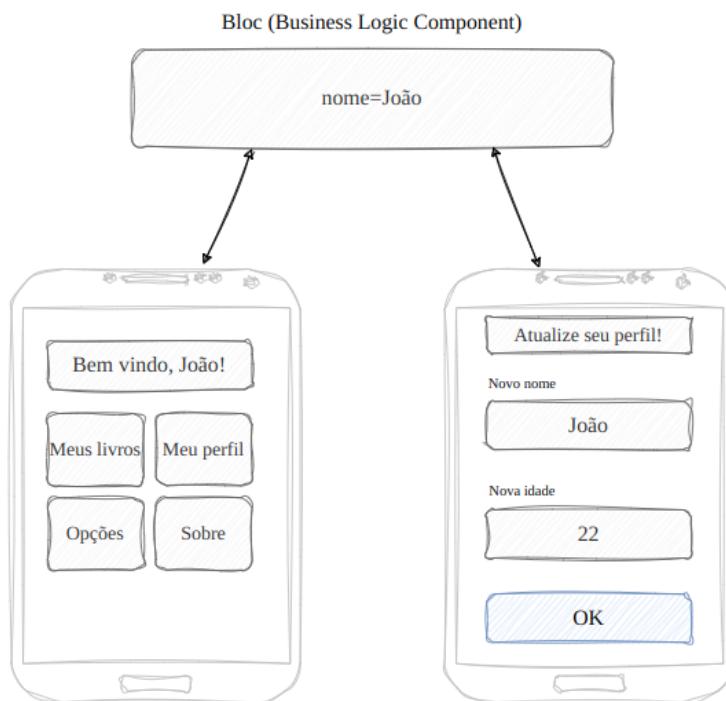
No contexto de interfaces gráficas incluindo variáveis de estado

- os observáveis são as variáveis de estado
- cada observador é uma função que executa automaticamente quando uma variável de estado é atualizada, alterando a interface gráfica

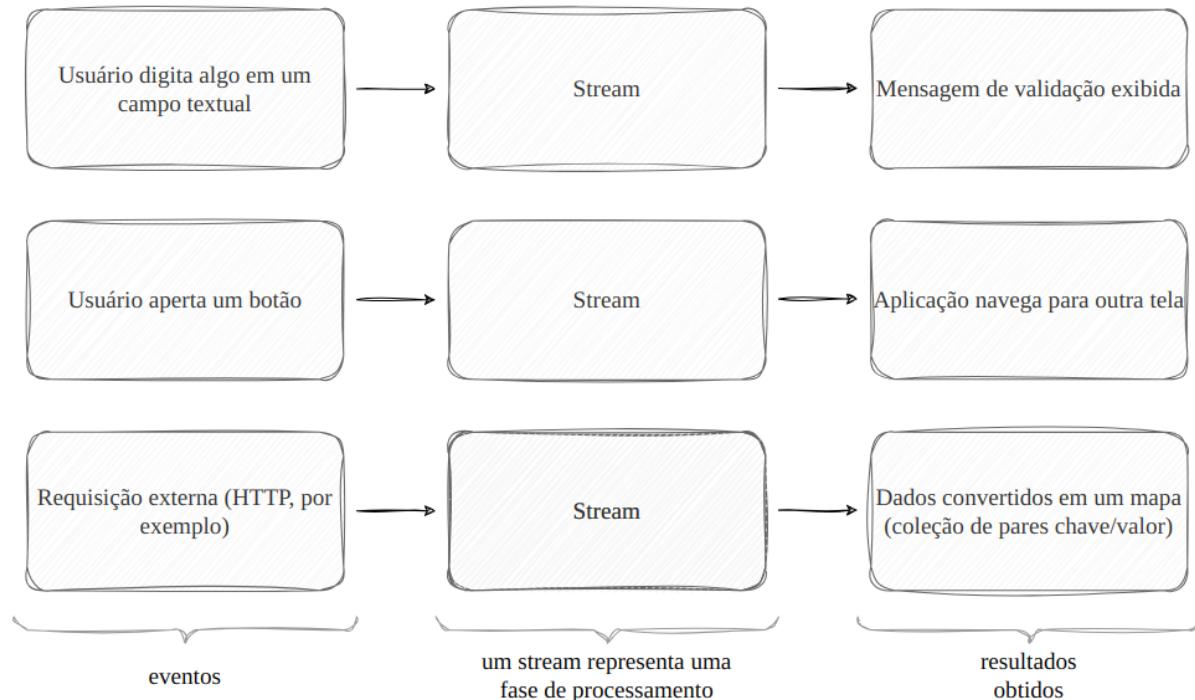
Visite-o no pub.dev

<https://pub.dev/packages/mobx>

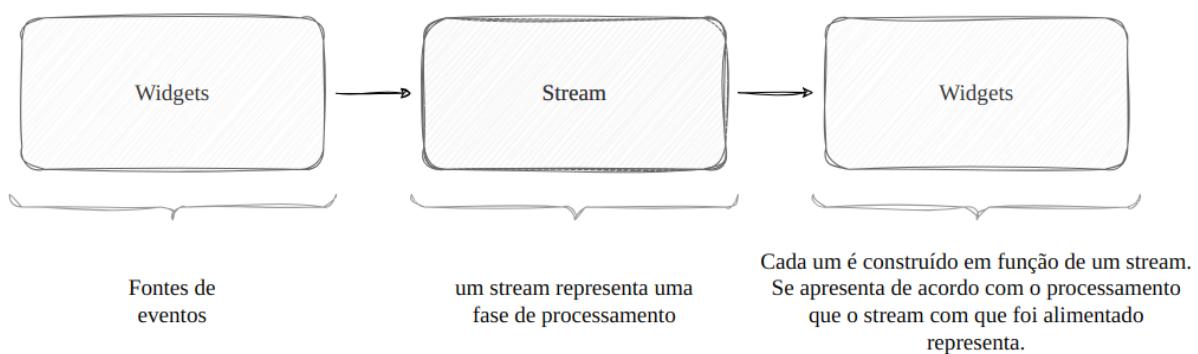
Neste material, vamos implementar o gerenciamento de estado de uma aplicação Flutter utilizando o padrão **Bloc**. A figura a seguir ilustra. Observe a existência de uma espécie de estado centralizado a que todas as telas têm acesso.



Como mencionamos, Bloc é um padrão baseado em **eventos** e **streams**. Eventos são originados a partir de “fontes de eventos”. Um Widget qualquer pode ser uma fonte de evento. Um stream representa um processamento aplicado ao evento de interesse a fim de que o resultado esperado possa ser obtido. Veja.



No contexto do Flutter, teremos Widgets como fontes de eventos. Teremos também Widgets como “**alvos**”. Um Widget alvo é construído em função de um Stream e ele se encarrega, internamente, de se apresentar de acordo com o processamento representado. Observe.



Seguindo a ideia que o padrão Bloc propõe, temos algo assim.



O padrão Bloc pode ser implementado “manualmente” ou utilizando-se uma biblioteca específica, o que pode ser mais simples. Veja uma **comparação entre o padrão Bloc e os Widgets com estado**.

| StatefulWidget                                | Padrão Bloc  |
|---|--|
| Uso muito simples                             | Uso pode ser complexo e a curva de aprendizado pode ser significativa                          |
| Não escala bem para aplicações de maior porte | Quando bem aplicado, atende às necessidades de aplicações de quaisquer níveis de complexidade. |
|   | Recomendado pelo Google  |

## 2 Desenvolvimento

### Workspace, novo projeto Flutter e VS Code

Comece criando uma pasta para desempenhar o papel de Workspace. Ou seja, uma pasta que abriga subpastas, cada qual representando um projeto Flutter. No Windows, você pode usar algo assim:

**C:\Users\seuUsuario\Documents\dev\**

A seguir, abra um terminal e vincule-o ao diretório que acabou de criar com

**C:\Users\seuUsuario\Documents\dev**

Crie o projeto Flutter com

**flutter create gerenciamento\_de\_estado**

A seguir, no terminal, use

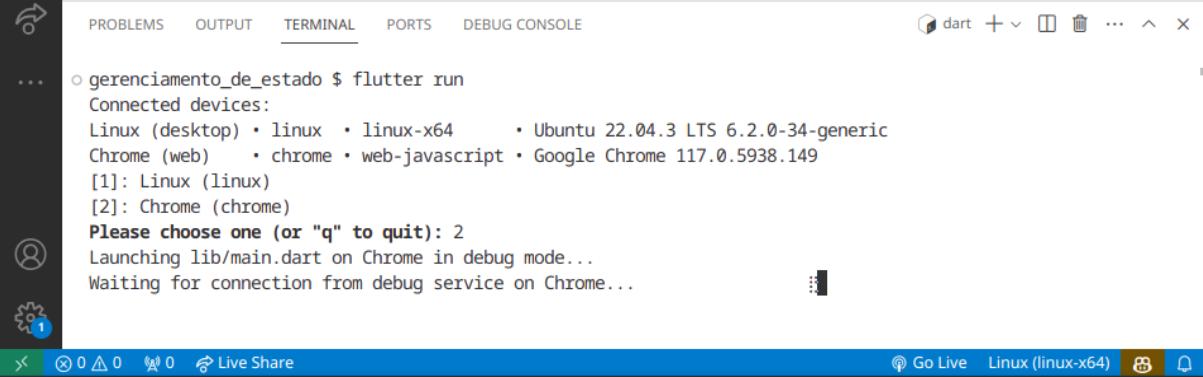
**code gerenciamento\_de\_estado**

para abrir uma instância do VS Code vinculada à pasta recém-criada.

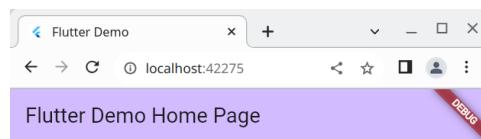
No VS Code, clique Terminal >> New Terminal para abrir um terminal interno do VS Code.  
Use

flutter run

para colocar a aplicação em execução.



```
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
...
o gerenciamento_de_estado $ flutter run
Connected devices:
Linux (desktop) • linux • linux-x64    • Ubuntu 22.04.3 LTS 6.2.0-34-generic
Chrome (web)    • chrome • web-javascript • Google Chrome 117.0.5938.149
[1]: Linux (linux)
[2]: Chrome (chrome)
Please choose one (or "q" to quit): 2
Launching lib/main.dart on Chrome in debug mode...
Waiting for connection from debug service on Chrome...
```



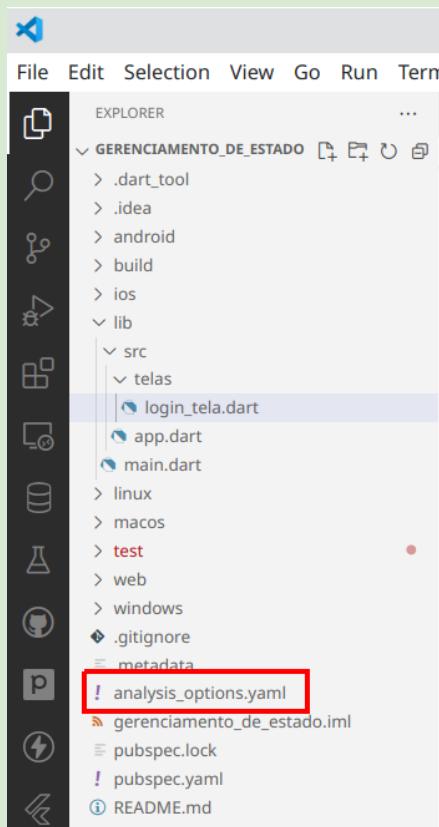
You have pushed the button this many times:  
0

+

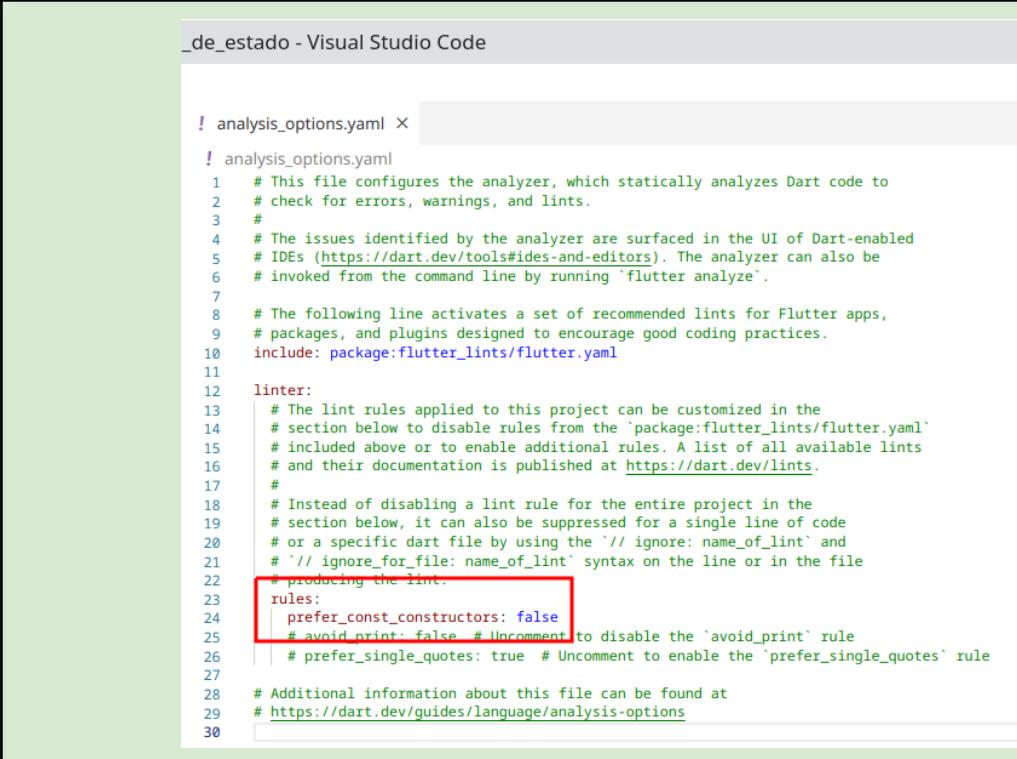
**Nota.** Ao longo do desenvolvimento, você verá mensagens sugerindo que você aplique **const** às chamadas de seus construtores, a fim de que o compilador possa produzir código otimizado. Você pode optar por adicionar **const** sempre que a dica surgir (e depois ter de remover, quando não for mais possível, o que acontece quando você chama um construtor **não const** num construtor que foi marcado como **const**) ou desabilitar essa dica no VS Code. Veja como fica.

```
        ); // ElevatedButton
    }
}
Widget subm
return El
onPress View Problem (Alt+F8) Quick Fix... (Ctrl+.
    child: Text('Login')
);
// ElevatedButton
```

Observe que esse comportamento é causado por uma verificação do linter chamada **prefer\_const\_constructors**. Se desejar, você pode desabilitá-la. Para tal, abra o arquivo **analysis\_options.yaml**.



Encontre a seção **rules** e desabilite a opção da seguinte forma.



```

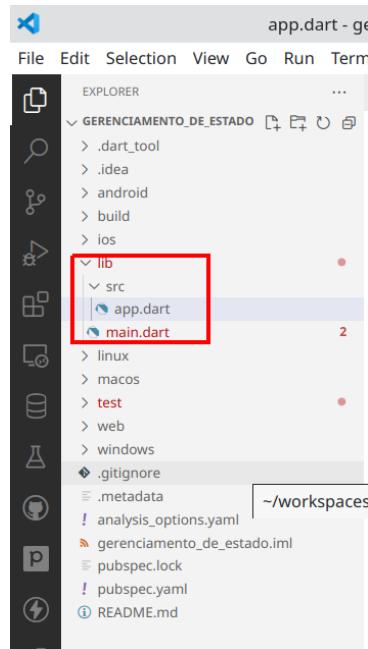
analysis_options.yaml
1  # This file configures the analyzer, which statically analyzes Dart code to
2  # check for errors, warnings, and lints.
3  #
4  # The issues identified by the analyzer are surfaced in the UI of Dart-enabled
5  # IDEs (https://dart.dev/tools#ides-and-editors). The analyzer can also be
6  # invoked from the command line by running 'flutter analyze'.
7  #
8  # The following line activates a set of recommended lints for Flutter apps,
9  # packages, and plugins designed to encourage good coding practices.
10 include: package:flutter_lints/flutter.yaml
11
12 linter:
13   # The lint rules applied to this project can be customized in the
14   # section below to disable rules from the 'package:flutter_lints/flutter.yaml'
15   # included above or to enable additional rules. A list of all available lints
16   # and their documentation is published at https://dart.dev/lints.
17   #
18   # Instead of disabling a lint rule for the entire project in the
19   # section below, it can also be suppressed for a single line of code
20   # or a specific dart file by using the '// ignore: name_of_lint' and
21   # '// ignore_for_file: name_of_lint' syntax on the line or in the file
22   # producing the lint.
23   rules:
24     prefer_const_constructors: false
25     # avoid_print: false # Uncomment to disable the 'avoid_print' rule
26     # prefer_single_quotes: true # Uncomment to enable the 'prefer_single_quotes' rule
27
28   # Additional information about this file can be found at
29   # https://dart.dev/guides/language/analysis-options
30

```

## Estrutura inicial da aplicação

Caracterize a estrutura inicial da aplicação da seguinte forma.

- Crie uma pasta chamada src, filha de lib.
- Crie um arquivo chamado app.dart na pasta src.
- Abra o arquivo **lib/main.dart** e apague todo o seu conteúdo.



No arquivo **app.dart**, crie um Widget sem estado chamado App. Ele usa

- **MaterialApp** - se encarrega de construir o mecanismo de navegação de telas
- **Scaffold** - esqueleto da aplicação, implementa a estrutura básica da especificação Material Design e viabiliza a definição das partes principais da tela, como uma barra inferior, a parte central, um botão de ação (FAB), um menu de “gaveta” etc.

No momento, ele apenas exibe um texto de teste.

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

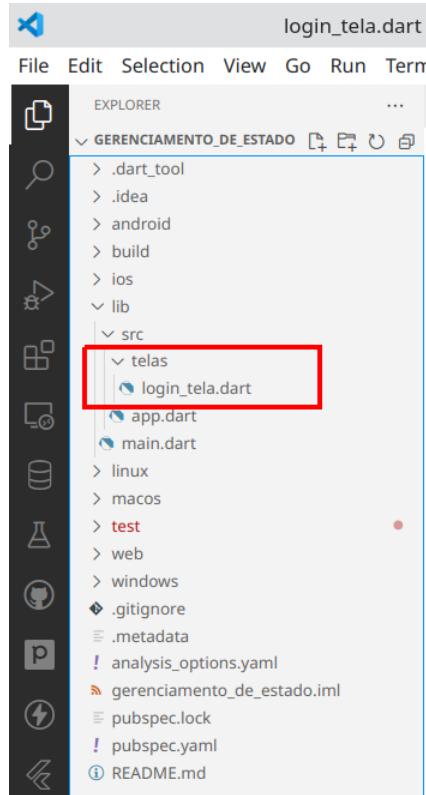
class App extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: 'Login',
      home: Scaffold(
        body: Text("Começando..."),
      )
    );
  }
}
```

Passe a utilizar o Widget App no arquivo **main.dart**, iniciando a aplicação como de costume, usando a função `runApp`.

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

void main() {
  runApp(App());
}
```

**(Uma tela para login)** A primeira tela da aplicação permitirá que o usuário faça login. Para criá-la, crie uma pasta chamada **telas**, subpasta de **src**. Nela, crie um arquivo chamado **login\_tela.dart**.



Veja seu código inicial.

```
import 'package:flutter/material.dart';
class LoginTela extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    ...
  }
}
```

Na raiz, ele possui um **Container**. Veja a sua documentação.

<https://api.flutter.dev/flutter/widgets/Container-class.html>

Ele será utilizado para que possamos definir detalhes como uma margem separando o conteúdo principal das bordas do dispositivo.

```

...
Widget build(BuildContext context) {
  return Container(
    //20 pixels de margem esquerda, direita, em cima e embaixo
    margin: EdgeInsets.all(20.0),
  );
}

```

O conteúdo principal terá Widgets **empilhados**. Para isso, utilizaremos um gerenciador de layouts do tipo **Column**.

```

...
Widget build(BuildContext context) {
  return Container(
    //20 pixels de margem esquerda, direita, em cima e embaixo
    margin: EdgeInsets.all(20.0),
    child: Column(
      ),
  );
}

```

Um Column pode ter vários filhos. Por isso, ele tem uma propriedade chamada **children** a que podemos associar uma coleção de Widgets.

```

...
Widget build(BuildContext context) {
  return Container(
    //20 pixels de margem esquerda, direita, em cima e embaixo
    margin: EdgeInsets.all(20.0),
    child: Column(
      children: [
        ],
    ),
  );
}

```

Passemos a utilizar o Widget recém criado, o que pode ser feito no arquivo **app.dart**.

```

import 'package:flutter/material.dart';
import '../src/telas/login_tela.dart';
class App extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Login',
      home: Scaffold(
        body: LoginTela(),
      )
    );
  }
}

```

**Nota.** O Flutter inclui diversos Widgets próprios para a construção de Forms. Por exemplo, poderíamos usar um **Form** como pai de alguns **TextFormField**. Porém, estes são Widgets com estado, ou seja, são StatefulWidget. Veja a sua documentação.

<https://api.flutter.dev/flutter/widgets/Form-class.html>

<https://api.flutter.dev/flutter/material/TextFormField-class.html>

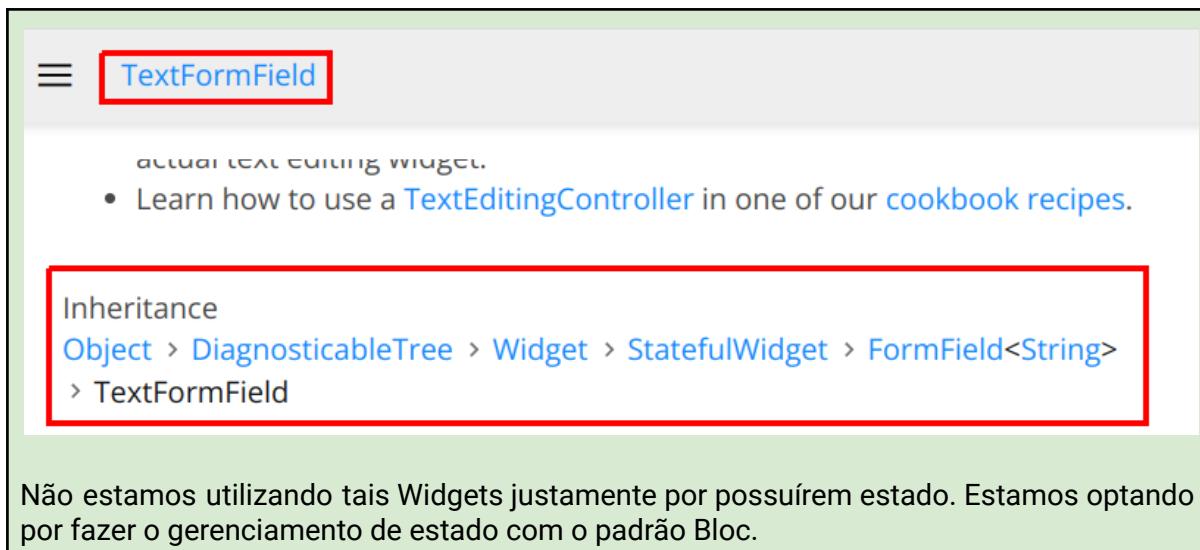
Na documentação, você pode ver a hierarquia de classes da qual eles pertencem. Observe que são subclasses de StatefulWidget.

≡ Form

- **FormField**, a single form field widget that maintains the current state.
- **TextFormField**, a convenience widget that wraps a **TextField** widget in a **FormF**

Inheritance

Object > DiagnosticableTree > Widget > StatefulWidget > Form



TextFormField

• Learn how to use a [TextEditingController](#) in one of our [cookbook recipes](#).

Inheritance

Object > DiagnosticableTree > Widget > StatefulWidget > FormField<String> > TextFormField

Não estamos utilizando tais Widgets justamente por possuírem estado. Estamos optando por fazer o gerenciamento de estado com o padrão Bloc.

**Nota.** Lembre-se sempre de visitar o

<https://pub.dev>

em busca de pacotes de qualidade que lhe permitam evitar reinventar a roda. Quando estiver trabalhando com forms, talvez você queira considerar o seguinte pacote

[https://pub.dev/packages/flutter\\_login](https://pub.dev/packages/flutter_login)

Observe como ele faz uso de recursos diversos, incluindo animações. Neste material, o nosso foco é o gerenciamento de estado. Não utilizaremos este Widget.

Utilizaremos Widgets do tipo **TextField** para os campos de e-mail e senha. Eles serão construídos por métodos auxiliares, assim, a manutenção do código pode se tornar mais simples. No arquivo **login\_tela.dart**, faça a definição do seguinte método.

```
import 'package:flutter/material.dart';
class LoginTela extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Container(
      //20 pixels de margem esquerda, direita, em cima e embaixo
      margin: EdgeInsets.all(20.0),
      child: Column(
        children: [
          ],
        ),
      );
  }
  Widget emailField(){
    return TextField();
  }
}
```

Utilizaremos as seguintes propriedades do TextField.

**keyboardType**: permite escolher o tipo de teclado. Neste caso, estamos escolhendo um apropriado para a digitação de endereços de e-mail.

**decoration**: para exibir texto explicando ao usuário o que ele deseja fazer

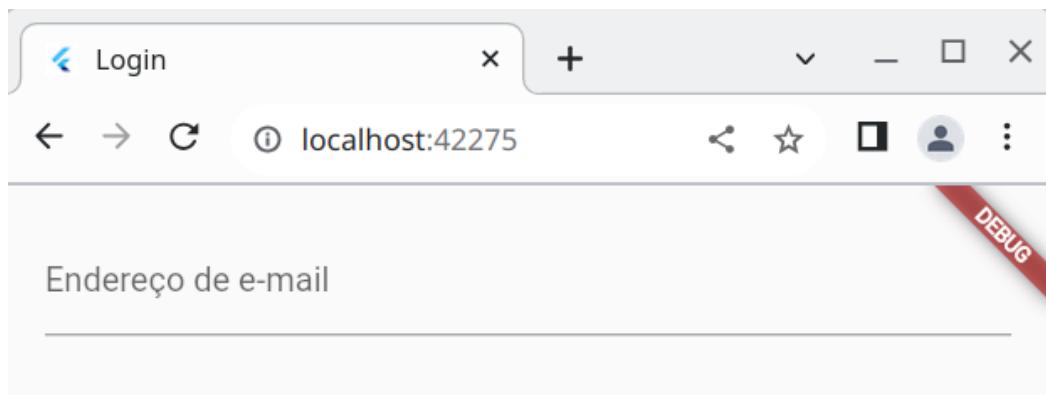
Como o método devolve um Widget, vamos fazer a sua chamada na primeira posição da lista que associamos à propriedade children do gerenciador de layouts Column.

```

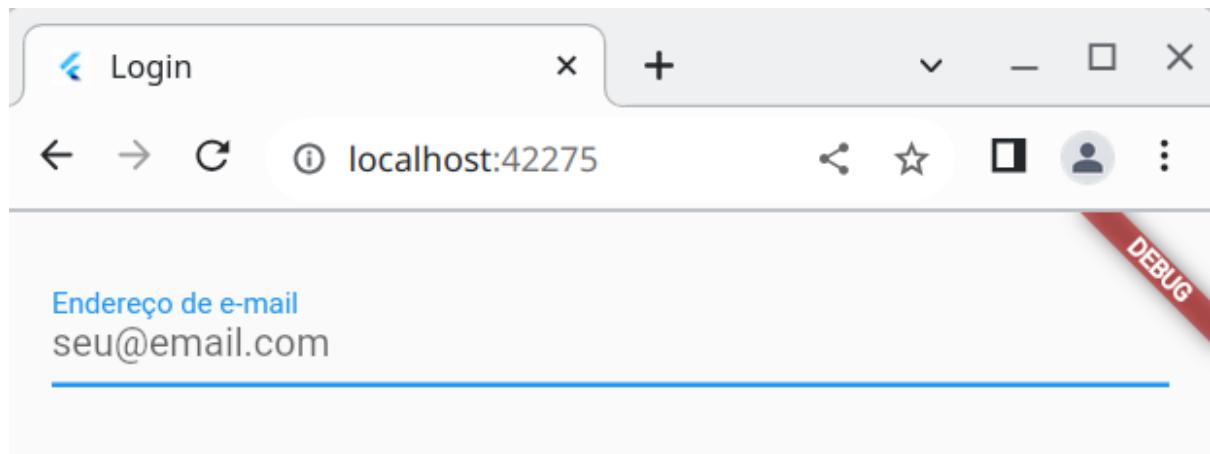
...
Widget build(BuildContext context) {
  return Container(
    //20 pixels de margem esquerda, direita, em cima e embaixo
    margin: EdgeInsets.all(20.0),
    child: Column(
      children: [
        emailField(),
      ],
    ),
  );
}
Widget emailField(){
  return TextField(
    keyboardType: TextInputType.emailAddress,
    decoration: InputDecoration(
      //dica que aparece quando o usuário clica
      hintText: 'seu@email.com',
      //rótulo flutuante: usuário clica, ele "sobe"
      labelText: 'Endereço de e-mail'
    ),
  );
}
...

```

No terminal em que a aplicação está em execução, aperte SHIFT + R para fazer um hot restart e ver o resultado.



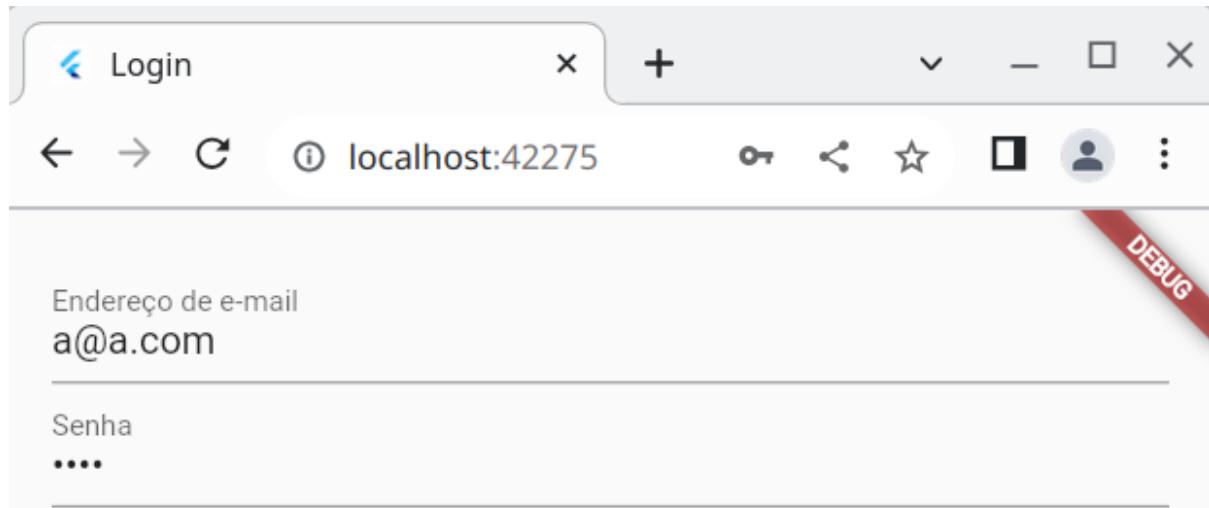
Clique no campo para ver o efeito.



Faremos o mesmo para o campo de senha. Por ser um campo de senha, vamos usar a propriedade **obscureText** para ocultar o texto nele digitado. Defina o novo método e chame-o para que produza o segundo filho da coleção `children` do gerenciador de layouts `Column`.

```
...
Widget build(BuildContext context) {
  return Container(
    //20 pixels de margem esquerda, direita, em cima e embaixo
    margin: EdgeInsets.all(20.0),
    child: Column(
      children: [
        emailField(),
        passwordField()
      ],
    ),
  );
}
Widget emailField(){
  return TextField(
    keyboardType: TextInputType.emailAddress,
    decoration: InputDecoration(
      //dica que aparece quando o usuário clica
      hintText: 'seu@email.com',
      //rótulo flutuante: usuário clica, ele "sobe"
      labelText: 'Endereço de e-mail'
    ),
  );
}
Widget passwordField(){
  return TextField(
    obscureText: true,
    decoration: InputDecoration(
      hintText: "Senha",
      labelText: "Senha"
    ),
  );
}
...
```

Faça novo hot restart (SHIFT + R) para ver o resultado.



Também vamos criar um botão para o envio do form. Ele será um **ElevatedButton**. Veja a sua documentação.

<https://api.flutter.dev/flutter/material/ElevatedButton-class.html>

A definição será feita tal qual fizemos com os campos textuais: criaremos um método que devolve uma instância de interesse e o chamaremos para que produza o terceiro filho da lista children do gerenciador de leiautes Column. Veja.

```
...
child: Column(
  children: [
    emailField(),
    passwordField(),
    submitButton()
  ],
),
);

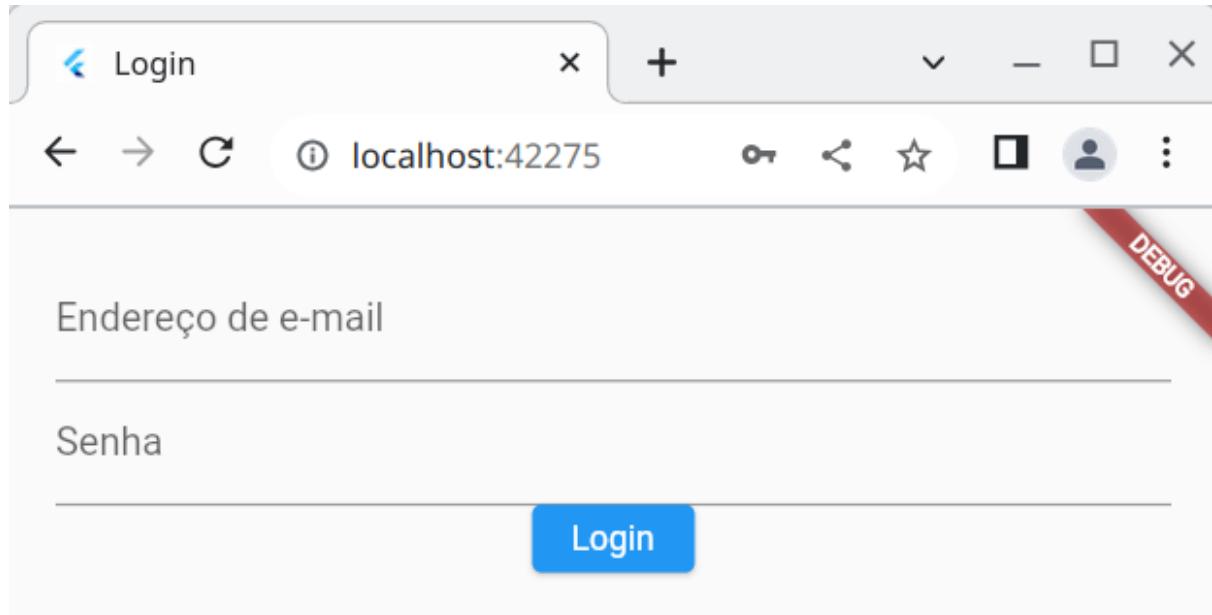
...

Widget passwordField(){
  ...
}

Widget submitButton(){
  return ElevatedButton(
    onPressed: (){},//ainda não temos o que fazer, função vazia
    child: Text('Login')
  );
}

...
```

Faça novo Hot Restart (SHIFT + R) e veja que o resultado pode ser melhor.



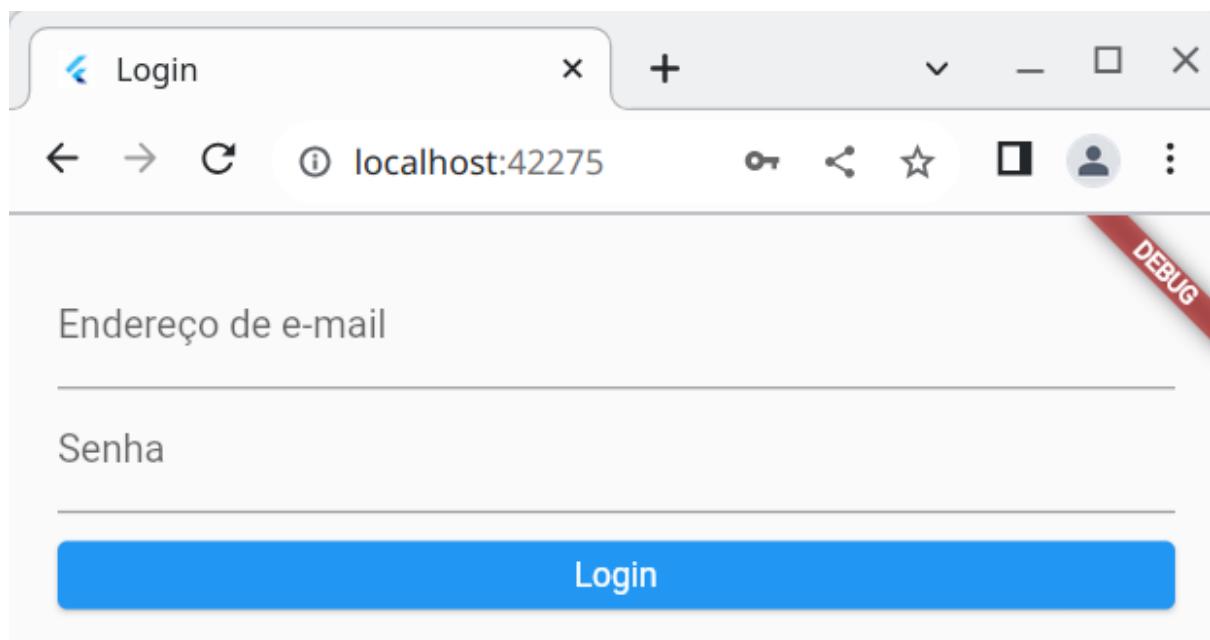
Vamos usar um novo Widget Container para adicionar um pouco de margem entre o campo de senha e o botão.

```
...
Widget build(BuildContext context) {
  return Container(
    //20 pixels de margem esquerda, direita, em cima e embaixo
    margin: EdgeInsets.all(20.0),
    child: Column(
      children: [
        emailField(),
        passwordField(),
        Container(
          margin: EdgeInsets.only(top: 12.0),
          child: submitButton(),
        )
      ],
    ),
  );
}
...
```

Além disso, vamos instruir o botão a tomar todo o espaço que ele pode com um Widget **Expanded**. Observe que, para que o botão faça a expansão na horizontal, será necessário fazer com que ele seja filho de um gerenciador de layout do tipo Row.

```
...
Widget build(BuildContext context) {
  return Container(
    //20 pixels de margem esquerda, direita, em cima e embaixo
    margin: EdgeInsets.all(20.0),
    child: Column(
      children: [
        emailField(),
        passwordField(),
        Container(
          margin: EdgeInsets.only(top: 12.0),
          child: Row(
            children: [
              Expanded(
                child: submitButton()
              )
            ],
          ),
        ),
      ],
    ),
  );
}
...
```

Veja o resultado esperado. Lembre-se de fazer Hot Restart (SHIFT + R).



## Validação dos campos: usando o padrão Bloc

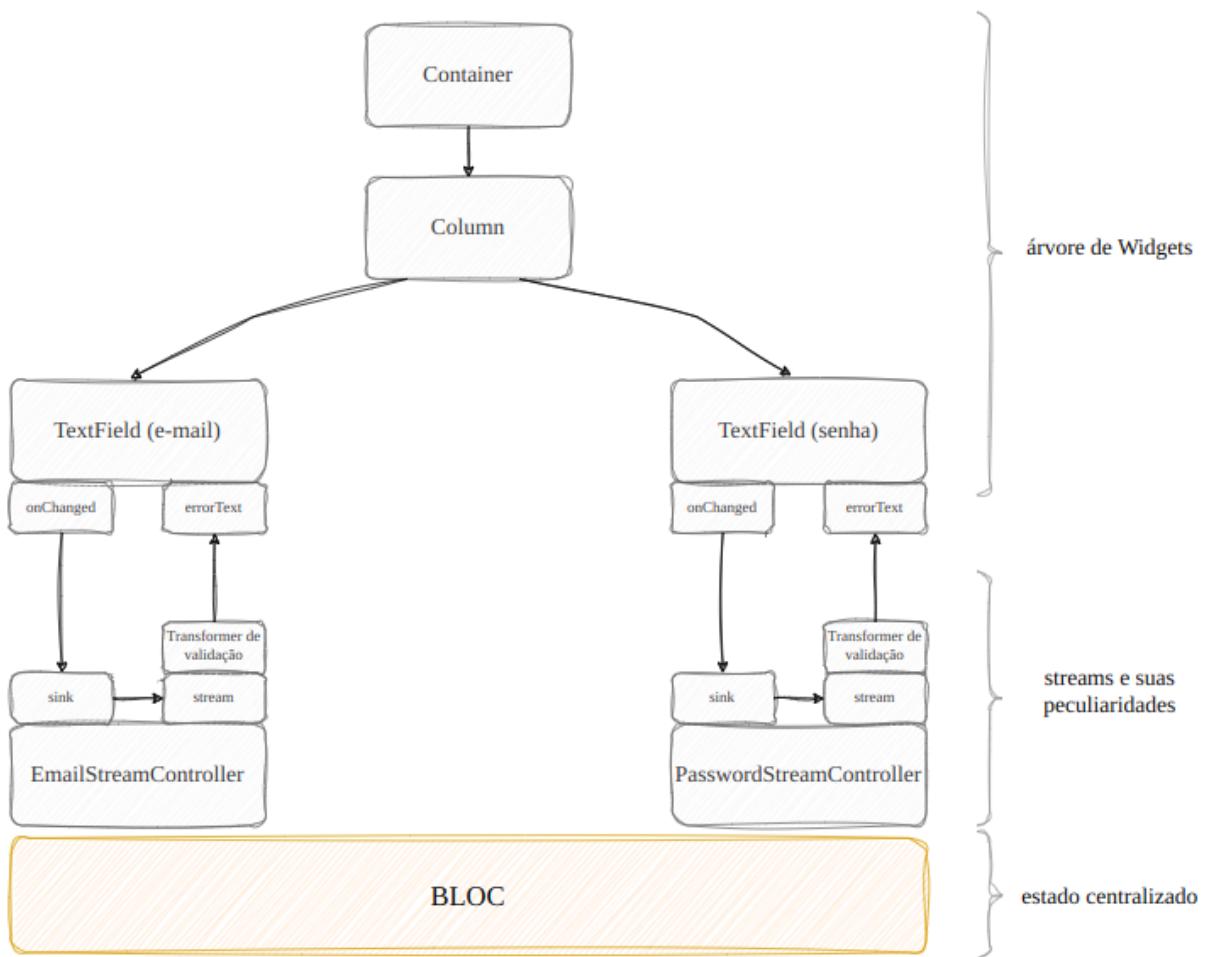
Quando o usuário digitar seu e-mail, desejamos verificar se está no formato correto. Também desejamos aplicar eventuais validações ao campo de senha. Para isso, vamos usar as seguintes propriedades.

- onChanged: propriedade do TextField a que podemos associar uma função a ser executada quando o usuário alterar o seu conteúdo
- errorText: propriedade do InputDecoration (filho do TextField) a que podemos associar uma mensagem que será exibida caso o valor digitado esteja incondizente com o formato desejado.

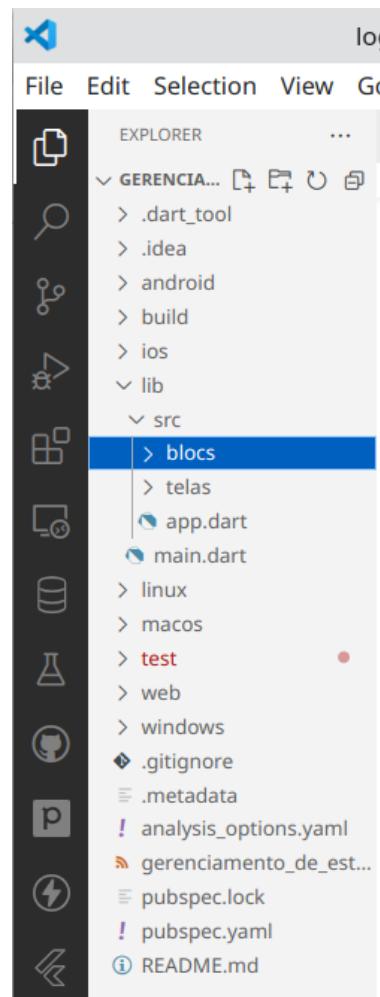
Elas precisam ser **vinculadas** de alguma forma. Para tal, vamos

- criar uma classe para operar como nosso Bloc
- A classe Bloc tem duas propriedades: EmailStreamController e PasswordStreamController. Ou seja, um stream para cada campo.
- Cada controller receberá um evento de interesse por meio de sua propriedade “**sink**”.
- Para cada um, vamos criar um “**transformer**” de validação.
- Atualizar os campos de exibição de mensagens de erro em função dos resultados obtidos pelos transformers.

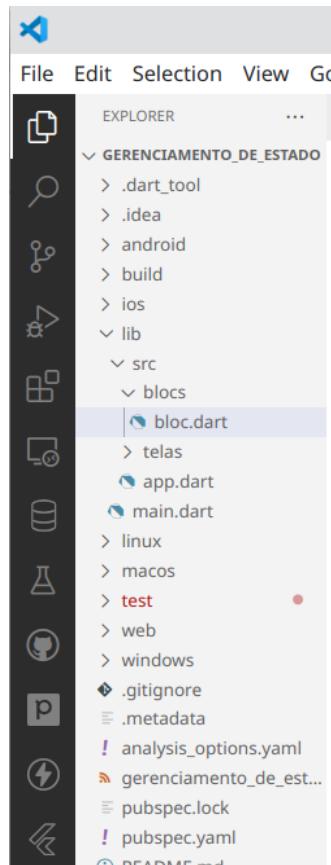
Veja a figura a seguir.



Para começar a codificação, vamos criar uma pasta chamada **blocs**, subpasta de **src**.



Depois, criamos um arquivo chamado **bloc.dart**.



Veja seu código inicial. Observe que as operações realizadas pelos streams são assíncronas, daí a necessidade do módulo.

```
import 'dart:async';

class Bloc{}
```

A seguir, definimos os dois streams.

```
import 'dart:async';

class Bloc{
    //StreamController vem do pacote dart:async
    final emailController = StreamController();
    final passwordController = StreamController();
}
```

A classe StreamController é genérica: podemos especificar qual o tipo do dado envolvido em suas operações. Em ambos os casos, temos strings (e-mail e senha). Vamos aplicar o tipo paramétrico a fim de fazer uso do sistema de tipos estático de Dart. Nossos controllers manipulariam "dynamic" (ou seja, tipos quaisquer) caso não o fizéssemos.

```
import 'dart:async';

class Bloc{
    //StreamController vem do pacote dart:async
    final emailController = StreamController <String> ();
    final passwordController = StreamController <String> ();
}
```

Para tornar o código cliente (aquele que a utiliza) mais simples, a class Bloc oferecerá métodos para a manipulação de seus streams, ocultando seus detalhes de implementação. Para alimentar o stream de email, por exemplo, teríamos de escrever algo assim

- emailController.sink.add

Para não deixar esse detalhe de implementação espalhado pelo código cliente, vamos escrever um método **getter**, simplificando a expressão. O método getter em questão dará acesso ao método add (sim, um getter que devolve um método).

- Para obter o conteúdo do stream, ou seja, o email sendo validado, escreveríamos algo assim

- emailController.stream.listen

Também escreveremos um getter para o stream, permitindo a chamada à função listen.

**Nota.** O método getter que escrevemos em Dart é semelhante àquele que escreveríamos em Java. O conceito é o mesmo. Entretanto, dizemos que Dart tem suporte “nativo” aos métodos getters/setters, tornando a escrita mais simples. Veja uma comparação a seguir. Primeiro, escrevemos uma classe clássica em Java. Depois, uma classe equivalente em Dart.

```
//Java
public class Pessoa {
    private String nome;
    private int idade;

    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }

    public static void main(String[] args) {
        Pessoa pessoa = new Pessoa("Joao", 25);
        System.out.println(pessoa.getNome());
        System.out.println(pessoa.getIdade());
    }
}
```

```

//Dart
class Pessoa {
  // _ como prefixo significa private
  String _nome;
  int _idade;
  //Construtor
  Pessoa(this._nome, this._idade);

  //arrow function
  String get nome => _nome;
  //arrow function
  set nome(String value) => _nome = value;
  //também pode com function "comum"
  //sem lista de parâmetros mesmo
  int get idade {return _idade;}
  //também daria para fazer "getIdade", mas ficaria estranho
  //int get getIdade {return _idade;}
  //setter precisa de lista de parâmetros
  set idade(int value){_idade = value;}
}

//Função main está fora da classe. Em Dart pode
void main() {
  var pessoa = Pessoa('Joao', 25);
  print(pessoa.nome);
  print(pessoa.idade);
  pessoa.nome = 'Maria';
  print(pessoa.nome);
}

```

Já que vamos oferecer acesso simplificado a detalhes internos de implementação, é razoável que ocluirmos os detalhes de implementação que não precisarão mais ser acessados externamente. Para tal, basta preceder seus nomes com um símbolo `_`.

```
import 'dart:async';

class Bloc{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController <String> ();
  final _passwordController = StreamController <String> ();
}
```

A seguir, escrevemos os getters que dão acesso aos streams, viabilizando a chamada ao método **listen** (veremos em breve) e, portanto, tornando possível adicionarmos algo aos streams.

```
import 'dart:async';

class Bloc{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController <String> ();
  final _passwordController = StreamController <String> ();

  get email => _emailController.stream;

  get password => _passwordController.stream;
}
```

Para cada um, podemos escrever o tipo devolvido explicitamente. Neste caso, ambos devolvem `Stream<String>`.

```
import 'dart:async';

class Bloc{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController <String> ();
  final _passwordController = StreamController <String> ();

  Stream<String> get email => _emailController.stream;

  Stream<String> get password => _passwordController.stream;
}
```

A seguir, vamos escrever os getters que dão acesso ao método add de ambos os streams. Ele permite alterarmos os dados existentes.

```
import 'dart:async';

class Bloc{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController <String> ();
  final _passwordController = StreamController <String> ();

  Stream<String> get email => _emailController.stream;

  Stream<String> get password => _passwordController.stream;

  get changeEmail => _emailController.sink.add;

  get changePassword => _passwordController.sink.add;
}
```

Ambos devolvem Function(String). Ou seja, uma função que recebe uma String. Podemos utilizar o sistema de tipos estático também neste caso.

```
import 'dart:async';

class Bloc{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController <String> ();
  final _passwordController = StreamController <String> ();

  Stream<String> get email => _emailController.stream;

  Stream<String> get password => _passwordController.stream;

  Function(String) get changeEmail => _emailController.sink.add;

  Function(String) get changePassword => _passwordController.sink.add;
}
```

O próximo passo será escrever os transformers de validação. Para cada um, vamos

- construir um objeto usando o método `fromHandlers` de `StreamTransformer`
- sobrescrever o método `handleData`, personalizando o seu funcionamento de acordo com as nossas necessidades.

`StreamTransformer` é uma classe genérica. Ela tem dois tipos genéricos. O primeiro é o tipo manipulado pelo Stream, o tipo do dado de entrada. O segundo é o tipo do dado que será produzido. Se nosso transformer fosse converter um inteiro para uma String, por exemplo, a classe seria `StreamTransformer <int, String>`. Como vamos validar uma String e devolver a string validada, a classe será `StreamTransformer <String, String>`.

Para começar a implementação, na pasta **blocs**, crie um arquivo chamado **validators.dart**. Veja seu código inicial. Observe que optamos por escrever um mixin.

```
mixin Validators{  
  
}
```

A seguir, escrevemos o `StreamTransformer` para validação de e-mail. A validação pode ser feita de diferentes formas.

- Usando expressões regulares, com as classes `RegExp`.
- Usando uma biblioteca já pronta, como a `email_validator`, disponível no `pub.dev`

[https://pub.dev/packages/email\\_validator](https://pub.dev/packages/email_validator)

Utilizaremos a segunda.

Para instalar a biblioteca, abra o seu arquivo **pubspec.yaml** e adicione seu nome e versão sob a seção **dependencies**.

```
...  
  
dependencies:  
  flutter:  
    sdk: flutter  
  email_validator: '^2.1.16'  
...
```

A seguir, no terminal, execute

flutter pub get

para que ela seja obtida do pub.dev.

De volta ao arquivo **validators.dart**, importe a biblioteca.

```
import 'package:email_validator/email_validator.dart';
mixin Validators{



}
```

Seguindo o passo a passo descrito, veja como fica o validador de e-mail.

```
import 'dart:async';
import 'package:email_validator/email_validator.dart';
mixin Validators{
    //O primeiro String é o tipo do primeiro parâmetro do handleData
    //O segundo String é o tipo do segundo parâmetro do handleData
    final validateEmail = StreamTransformer<String, String>.fromHandlers(
        handleData: (email, sink){
            if (EmailValidator.validate(email)){
                //adicionamos ao sink, permitindo o "fluxo" do e-mail adiante
                sink.add(email);
            }
            else{
                //caso contrário, adicionamos um erro
                sink.addError("E-mail inválido");
            }
        }
    );
}
```

**Exercício.** Escreva um validador para senhas. Para ser válida, uma senha deve ter, pelo menos, 4 caracteres. Se desejar, pesquise sobre a classe RegExp e faça validações mais interessantes.

<https://api.flutter.dev/flutter/dart-core/RegExp-class.html>

Veja como fica.

```

import 'dart:async';

import 'package:email_validator/email_validator.dart';
mixin Validators{
    //O primeiro String é o tipo do primeiro parâmetro do handleData
    //O segundo String é o tipo do valor manipulado pelo segundo parâmetro do
    handleData (o segundo parâmetro é um EventSink<String>)
    final validateEmail = StreamTransformer<String, String>.fromHandlers(
        handleData: (email, sink){
            if (EmailValidator.validate(email)){
                //adicionamos ao sink, permitindo o "fluxo" do e-mail adiante
                sink.add(email);
            }
            else{
                //caso contrário, adicionamos um erro
                sink.addError("E-mail inválido");
            }
        }
    );
    final validatePassword = StreamTransformer<String, String>.fromHandlers(
        handleData: (email, sink){
            if (email.length > 3){
                sink.add(email);
            }
            else{
                sink.addError("Senha deve ter, pelo menos, 4 caracteres");
            }
        }
    );
}

```

A seguir, no arquivo **bloc.dart**, podemos fazer com que a classe Bloc reutilize as funcionalidades providas pelo mixin.

```

import 'dart:async';
import 'validators.dart';
class Bloc with Validators{
    ...
}

```

O uso dos validadores será feito da seguinte forma: sempre que um stream for solicitado, ele passará por uma transformação feita por um dos transformers de validação. Observe.

```
import 'dart:async';
import 'validators.dart';
class Bloc with Validators{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController <String> ();
  final _passwordController = StreamController <String> ();

  Stream<String> get email =>
    _emailController.stream.transform(validateEmail);

  Stream<String> get password =>
    _passwordController.stream.transform(validatePassword);

  Function(String) get changeEmail => _emailController.sink.add;

  Function(String) get changePassword => _passwordController.sink.add;
}
```

## Fechando o Bloc

Nosso Bloc é composto por recursos que podem - e devem - ser fechados. São os Streams. Eles representam recursos alocados e, quando não forem mais utilizados, devem ser liberados. Como de costume, isso pode ser feito utilizando-se o seu método **close**. Como o Bloc tem potencialmente vários streams, vamos escrever um método que agrupa as chamadas ao método close sobre cada um deles.

**Nota.** “dispose” significa algo como “descartar”.

```

import 'dart:async';
import 'validators.dart';
class Bloc with Validators{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController <String> ();
  final _passwordController = StreamController <String> ();

  Stream<String> get email =>
    _emailController.stream.transform(validateEmail);

  Stream<String> get password =>
    _passwordController.stream.transform(validatePassword);

  Function(String) get changeEmail => _emailController.sink.add;

  Function(String) get changePassword => _passwordController.sink.add;

  void dispose(){
    _emailController.close();
    _passwordController.close();
  }
}

```

Escolhendo o escopo do Bloc: global ou de escopo restrito

Podemos fazer uso do padrão Bloc de algumas formas diferentes. Veja duas delas.

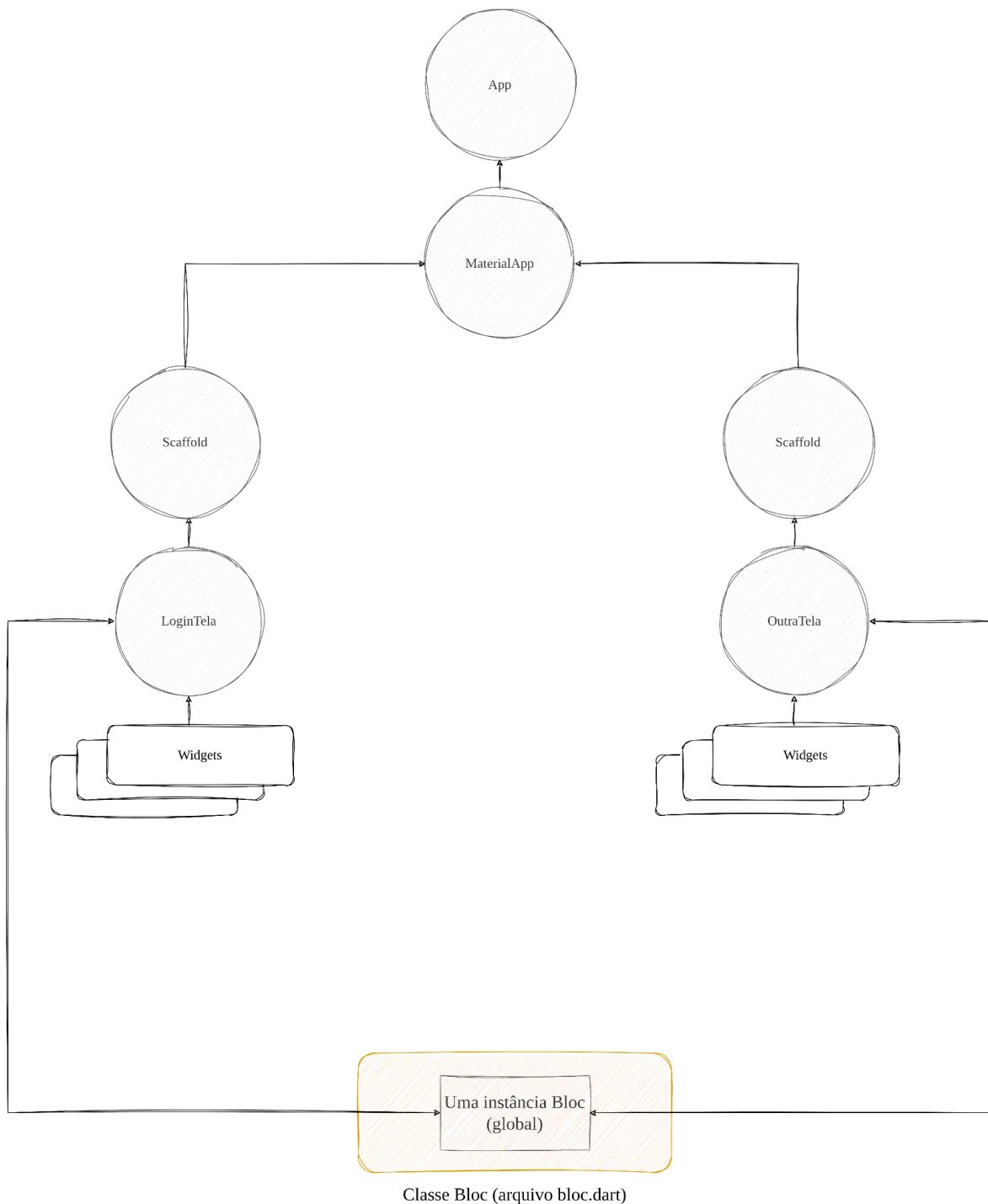
- **Única instância global.** Neste cenário, temos uma única instância Bloc e ela é responsável por manipular o estado da aplicação inteira.

- **Múltiplas instâncias com escopo restrito.** Neste cenário, temos múltiplas instâncias Bloc, cada uma responsável por lidar com uma pequena parte do estado da aplicação.

**Nota.** O primeiro cenário, com uma instância global, tende a ser mais simples e apropriado para aplicações mais simples. O segundo, com múltiplas instâncias, requer mais programação mas pode se mostrar vantajoso para aplicações mais sofisticadas, maiores, com mais Widgets e mais variáveis de estado.

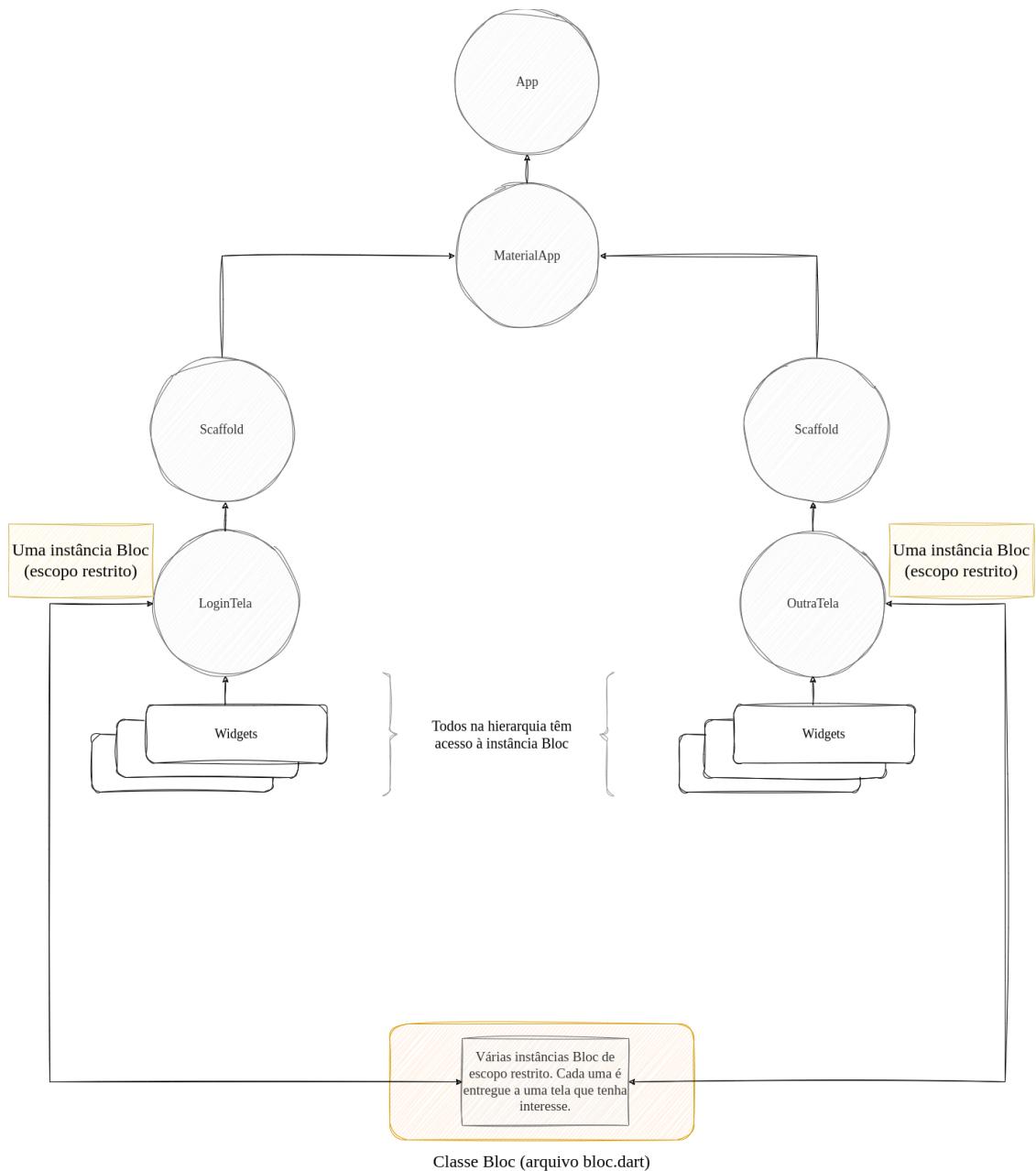
Entretanto, é sempre importante lembrar que **não há verdade absoluta** e que duas aplicações diferentes, ainda que com complexidade semelhante, podem ser implementadas com estratégias diferentes e com resultados semelhantes.

O cenário com **uma única instância global** fica da seguinte forma.



Veja como fica o cenário com **múltiplas instâncias com escopo restrito**.

**Nota.** Há um detalhe técnico sobre a forma como os Widgets “filhos” terão acesso ao bloco de seu antecessor, que envolve o uso de um **InheritedWidget**. Trataremos deste assunto posteriormente.



Utilizando uma única instância Bloc global

Nesta seção, **vamos utilizar a estratégia com uma única instância global**. Ela será construída no arquivo **bloc.dart**. Ou seja, quaisquer arquivos .dart que o importem terão acesso a ela. Observe como ela é construída **fora do corpo da classe**.

```
import 'dart:async';
import 'validators.dart';
class Bloc with Validators{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController <String> ();
  final _passwordController = StreamController <String> ();

  Stream<String> get email =>
    _emailController.stream.transform(validateEmail);

  Stream<String> get password =>
    _passwordController.stream.transform(validatePassword);

  Function(String) get changeEmail => _emailController.sink.add;

  Function(String) get changePassword =>
    _passwordController.sink.add;

  void dispose() {
    _emailController.close();
    _passwordController.close();
  }
}
final bloc = Bloc();
```

A seguir, no arquivo **login\_tela.dart**, importamos o arquivo **bloc.dart**.

```
import 'package:flutter/material.dart';
import '../blocs/bloc.dart';
class LoginTela extends StatelessWidget{
  ...
}
```

No próximo passo, vamos “englobar” o `TextField` de e-mail utilizando um **StreamBuilder**. Visite a sua documentação.

<https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html>

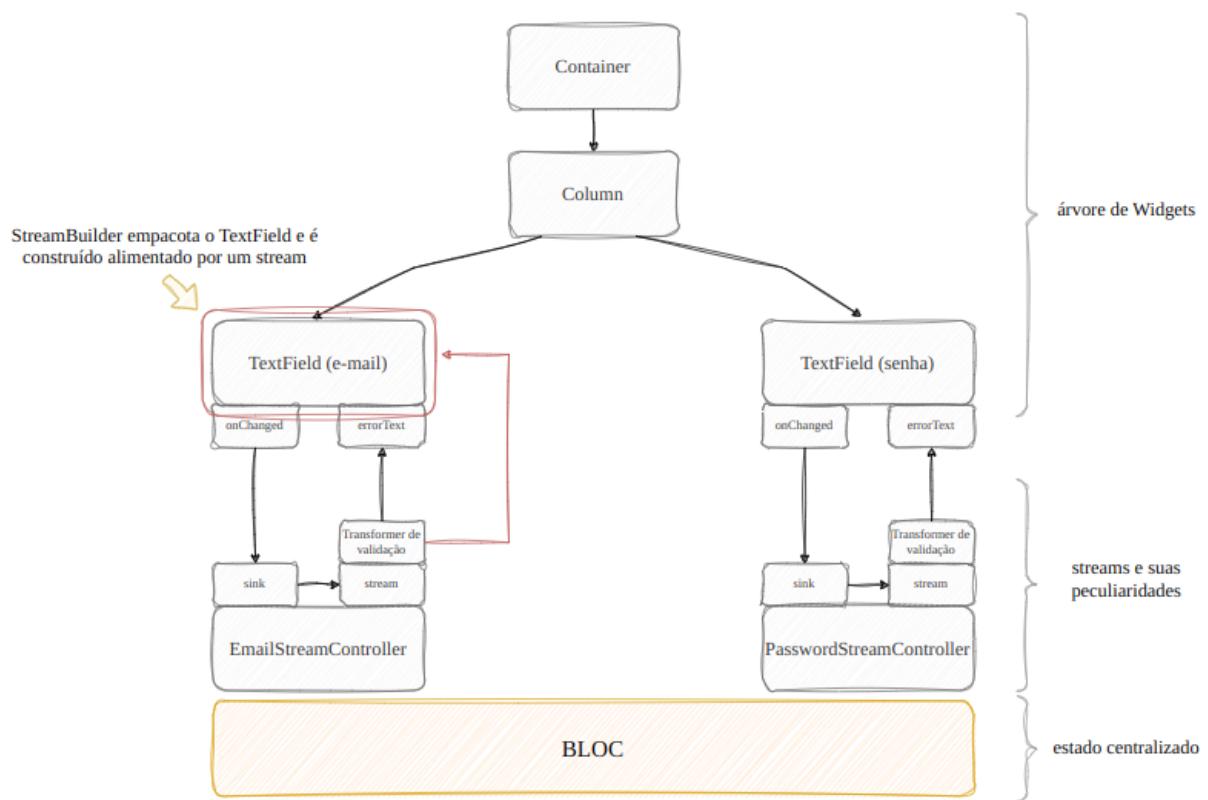
Ele opera da seguinte forma:

- É construído em função de um stream e de um Widget.
- Uma nova atualização do stream gera um “snapshot”.
- Se reconstrói (ou seja, atualiza graficamente o Widget que ele empacota) sempre que um novo snapshot estiver disponível.

Veja.

```
...
Widget emailField(){
    return StreamBuilder(
        //stream que, quando atualizado, produz um snapshot
        //observe como usamos o stream definido no bloc
        stream: bloc.email,
        //função que, quando chamada, causa a atualização do Widget (TextField,
        //neste caso) empacotado pelo StreamBuilder
        builder: ((context, snapshot) {
            return TextField(
                keyboardType: TextInputType.emailAddress,
                decoration: InputDecoration(
                    //dica que aparece quando o usuário clica
                    hintText: 'seu@email.com',
                    //rótulo flutuante: usuário clica, ele "sobe"
                    labelText: 'Endereço de e-mail'
                ),
            );
        })),
    );
}
...
```

Neste ponto, o que implementamos é destacado a seguir.



O próximo passo é responder a seguinte pergunta:

Quando um snapshot novo vai ser gerado?

A resposta é simples:

Um novo snapshot é gerado sempre que o usuário atualiza o campo de e-mail.

Em outras palavras, precisamos vincular a propriedade onChanged do TextField ao método add do stream correspondente. Sim, aquele mesmo que definimos no Bloc. Observe.

```

...
builder: ((context, snapshot) {
  return TextField(
    onChanged: (newValue) {
      bloc.changeEmail(newValue);
    },
    keyboardType: TextInputType.emailAddress,
    decoration: InputDecoration(
      //dica que aparece quando o usuário clica
      hintText: 'seu@email.com',
      //rótulo flutuante: usuário clica, ele "sobe"
      labelText: 'Endereço de e-mail'
    ),
  );
}),
...

```

**Nota.** Também poderíamos especificar apenas o nome da função a ser associada à propriedade onChanged, ao invés de especificar uma função que a chama explicitamente. Ficaria assim.

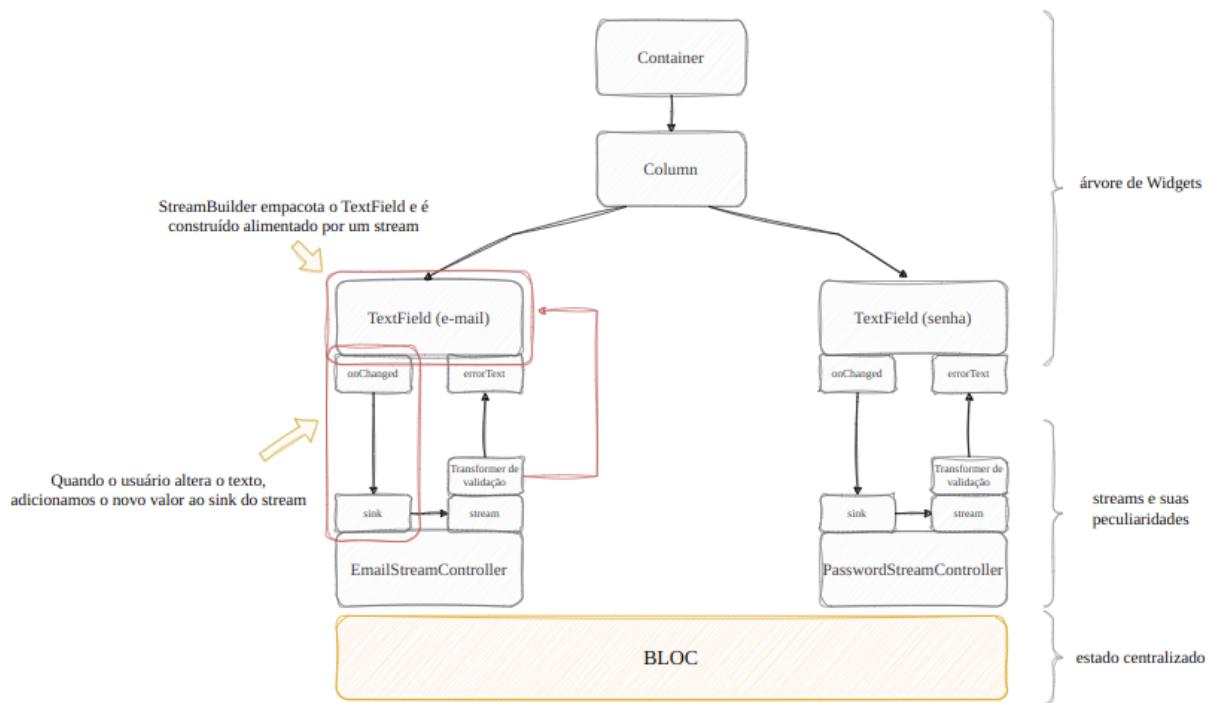
```

...
builder: ((context, snapshot) {
  return TextField(
    onChanged: bloc.changeEmail,
    keyboardType: TextInputType.emailAddress,
    decoration: InputDecoration(
      //dica que aparece quando o usuário clica
      hintText: 'seu@email.com',
      //rótulo flutuante: usuário clica, ele "sobe"
      labelText: 'Endereço de e-mail'
    ),
  );
}),
...

```

A chamada e a passagem do novo valor estão implícitos. O resultado é o mesmo. Mantenha a que você preferir. De preferência, teste as duas. :)

Agora, o que temos implementado é ilustrado a seguir.



Observe que o campo “`errorText`” ainda não está sabendo a respeito das interações do usuário. Ele não sabe ainda que deve se atualizar. Aliás, para fazê-lo, ele precisa saber qual texto exibir, que é disponibilizado pelo `snapshot`! O `snapshot` é do tipo `AsyncSnapshot<String>`. Veja a sua documentação.

<https://api.flutter.dev/flutter/widgets/AsyncSnapshot-class.html>

Em particular, veja o seguinte trecho.

```

(final)
data → T?
The latest data received by the asynchronous computation.

(final)
error → Object?
The latest error object received by the asynchronous computation.

(final)
hasData → bool
Returns whether this snapshot contains a non-null data value.

(read-only)
hasError → bool
Returns whether this snapshot contains a non-null error value.

(read-only)
hashCode → int
The hash code for this object.
  
```

Os campos **data** e **error** permitem a obtenção de dados “comuns” ou de erros recebidos na última atualização do snapshot. Lembra que adicionamos conteúdo ao sink usando os métodos **add** e **addError**? Agora está clara a diferença! Além disso, os métodos **hasData** e **hasError** permitem verificar se há dados comuns ou se há erros.

Aplique o tipo estático do snapshot. Embora não seja obrigatório, utilizar o sistema de tipos estático é sempre uma boa prática.

```

...
return StreamBuilder(
  //stream que, quando atualizado, produz um snapshot
  //observe como usamos o stream definido no bloc
  stream: bloc.email,
  //função que, quando chamada, causa a atualização do Widget (TextField,
  neste caso) empacotado pelo StreamBuilder
  builder: ((context, AsyncSnapshot<String> snapshot) {
    return TextField(
      onChanged: bloc.changeEmail,
      keyboardType: TextInputType.emailAddress,
      decoration: InputDecoration(
        //dica que aparece quando o usuário clica
        hintText: 'seu@email.com',
        //rótulo flutuante: usuário clica, ele "sobe"
        labelText: 'Endereço de e-mail'
      ),
    );
  )),
...

```

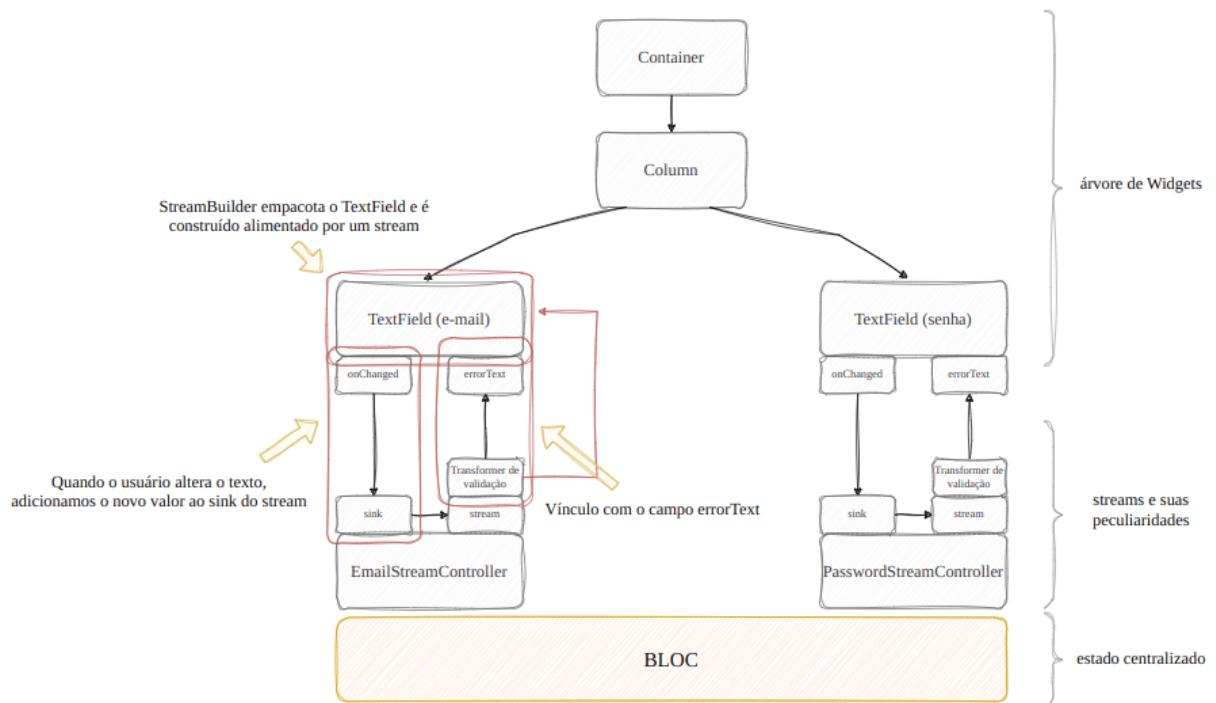
Assim, basta associar a propriedade **error** do snapshot à propriedade **errorText** do **InputDecoration** associado ao **TextField**, filho do **StreamBuilder**.

```

...
builder: ((BuildContext context, AsyncSnapshot <String> snapshot) {
  return TextField(
    onChanged: bloc.changeEmail,
    keyboardType: TextInputType.emailAddress,
    decoration: InputDecoration(
      //dica que aparece quando o usuário clica
      hintText: 'seu@email.com',
      //rótulo flutuante: usuário clica, ele "sobe"
      labelText: 'Endereço de e-mail',
      //o erro não necessariamente é String, por isso seu tipo é Object?,
      dai
      o uso do toString()
      errorText: snapshot.error.toString()
    ),
  );
}),
...

```

Agora a nossa implementação ficou assim.

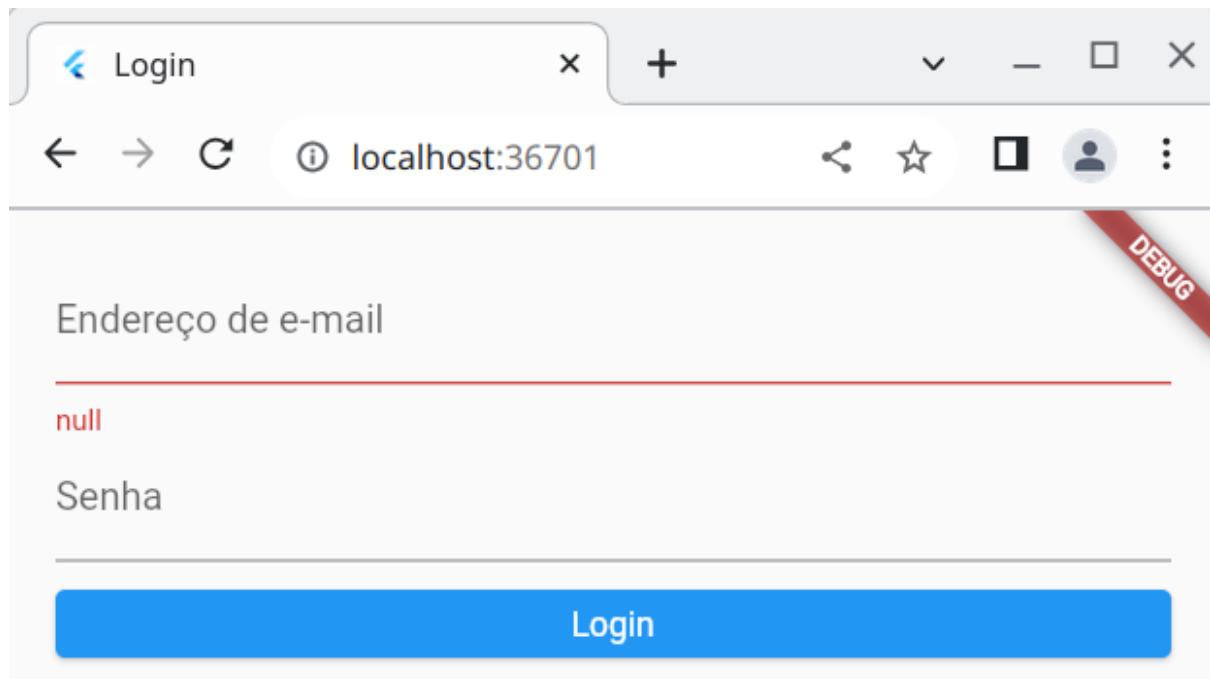


Tudo pronto para o campo de e-mail. Façamos um teste. Primeiro, certifique-se de que a aplicação está em execução. Se necessário, use

flutter run

para executá-la, claro, num terminal do VS Code, para facilitar.

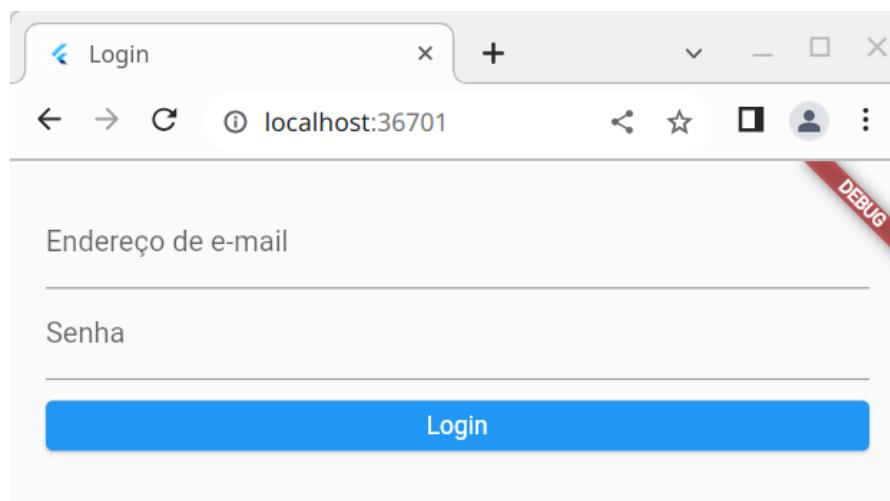
Observe que, assim que a aplicação inicia, o campo de e-mail já está vermelho e exibindo uma mensagem "null" abaixo.



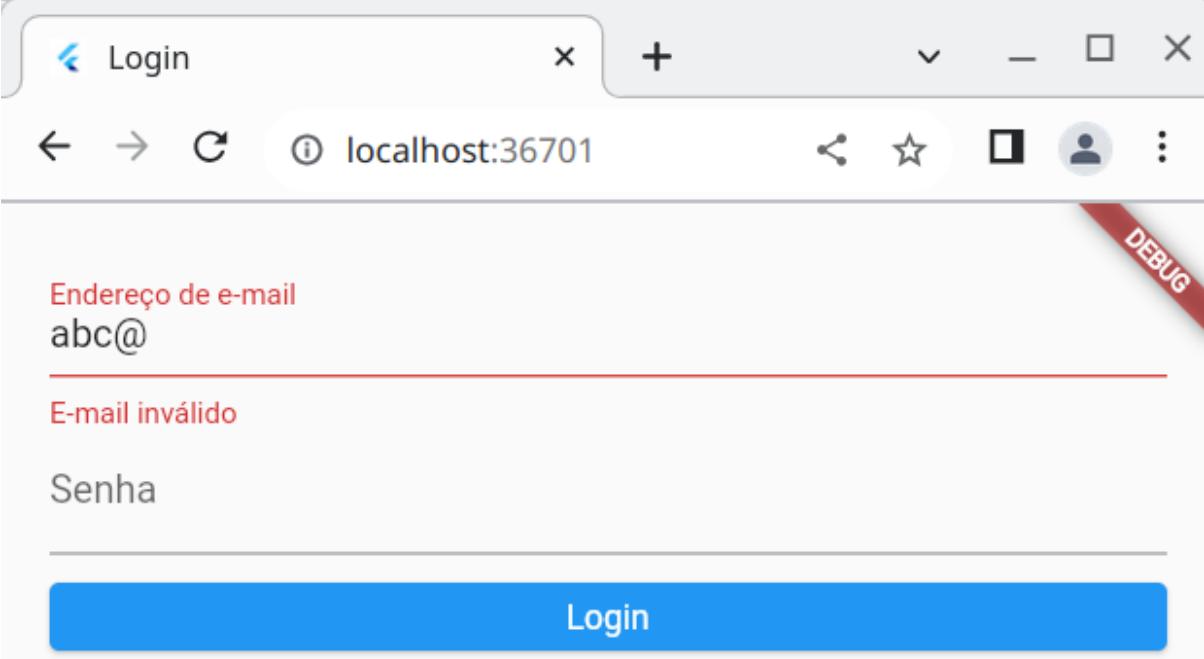
Ocorre que estamos pegando a representação textual de `snapshot.error` (que está valendo `null` e que, portanto, tem representação textual - aquela calculada pelo `toString` - igual a `null`) e associando à propriedade `errorText` de maneira incondicional. Façamos uso do operador ternário, em conjunto com o método `hasError` para consertar.

```
...
errorText: snapshot.hasError ? snapshot.error.toString() : null
...
```

Faça shift + R no terminal e teste novamente. Veja o resultado esperado.

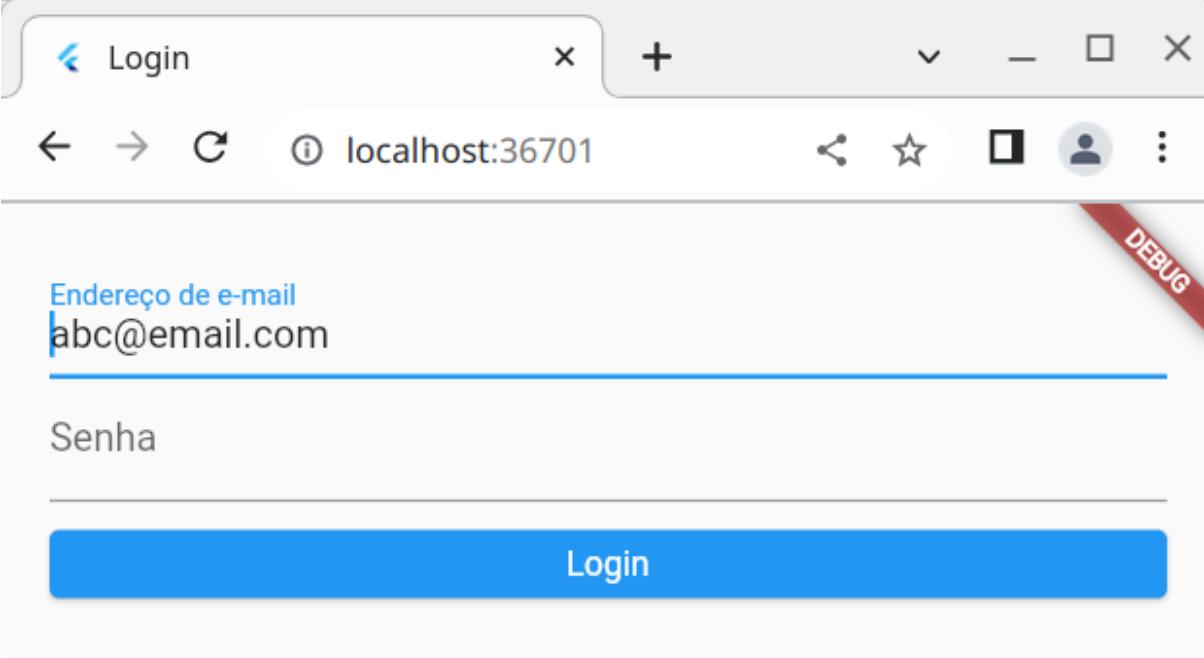


Comece a digitar e verifique o resultado. Esse é um exemplo em que digitamos algo que ainda não é um e-mail válido.



A screenshot of a web browser window titled "Login" at "localhost:36701". The address bar shows the URL. A red "DEBUG" ribbon is visible in the top right corner. The page contains a form with two text input fields. The first field is labeled "Endereço de e-mail" and contains the text "abc@". The second field is labeled "E-mail inválido". Below the inputs is a "Senha" label and a large blue "Login" button.

Quando completamos o texto caracterizando um e-mail válido, a coisa fica assim.



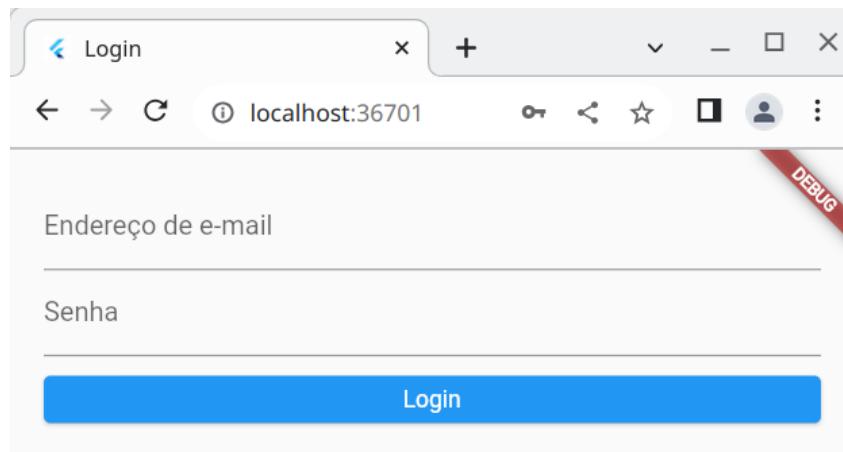
A screenshot of a web browser window titled "Login" at "localhost:36701". The address bar shows the URL. A red "DEBUG" ribbon is visible in the top right corner. The page contains a form with two text input fields. The first field is labeled "Endereço de e-mail" and contains the text "abc@email.com". The second field is labeled "Senha". Below the inputs is a "Login" button.

A seguir, precisamos fazer a mesma implementação para o stream de password. A alteração é idêntica, apenas envolve o outro stream. Veja.

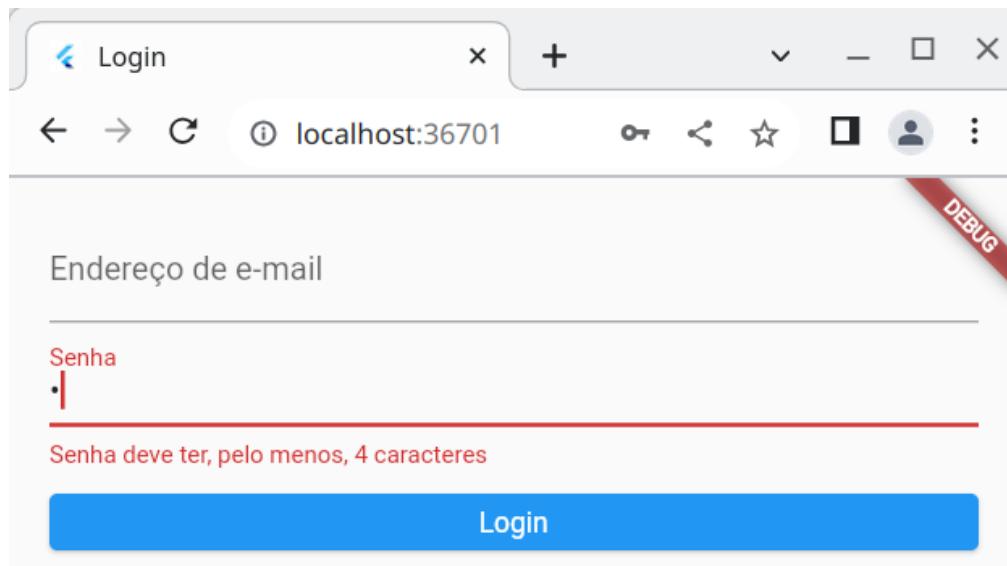
```
...
Widget passwordField() {
    return StreamBuilder(
        stream: bloc.password,
        builder: (context, AsyncSnapshot<String> snapshot) {
            return TextField(
                onChanged: bloc.changePassword,
                obscureText: true,
                decoration: InputDecoration(
                    hintText: "Senha",
                    labelText: "Senha",
                    errorText: snapshot.hasError ? snapshot.error.toString() : null
                ),
            );
        },
    );
}
...
```

Aperte SHIFT + R no terminal novamente e faça testes. Quando a aplicação começa, ela não mostra erro algum.

**Nota.** Este é um form que acabou de nascer. Ele jamais foi tocado. Muitas vezes utilizamos a expressão “**pristine**” para nos referirmos a forms assim. Dizemos que esse form está “pristine”. Pristine significa algo como “imaculado”, “em bom estado”, “impecável”, “puro de qualquer nódoa moral”, “inocente”.

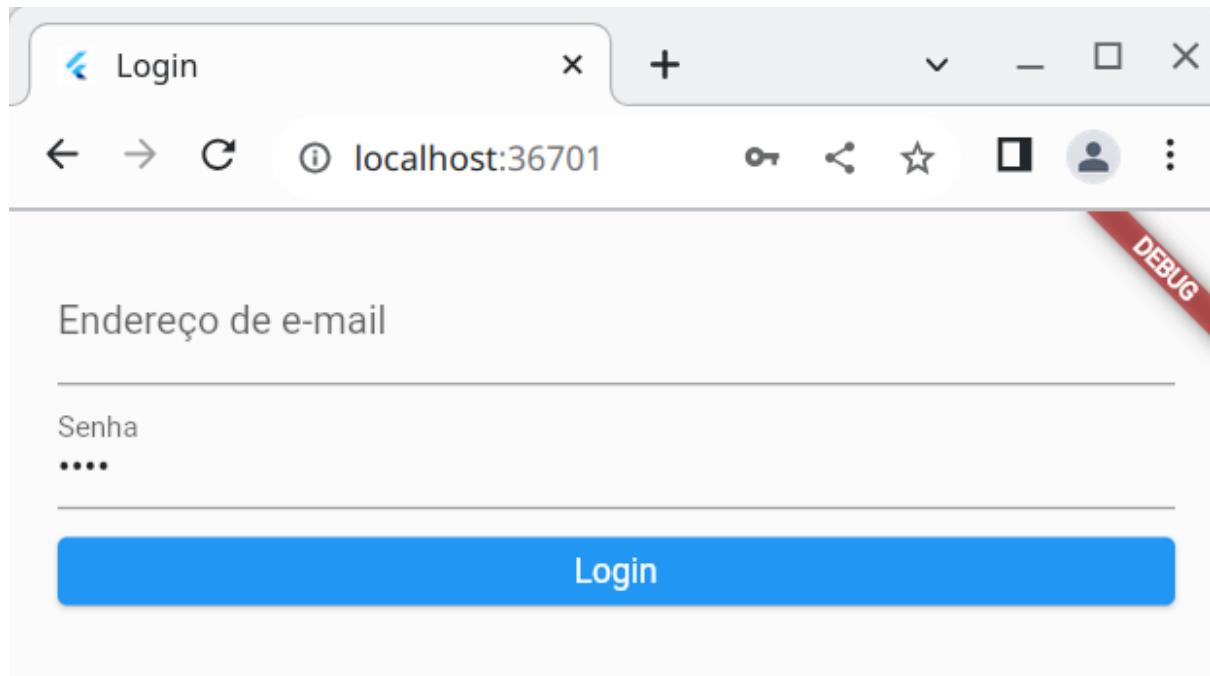


Faça uma atualização digitando apenas uma letra no campo de senha. Veja a mensagem de erro. Observe que, agora, o form, já foi tocado. Ele já não é mais puro (pristine).



A screenshot of a web browser window titled "Login". The address bar shows "localhost:36701". The page contains a form with two fields: "Endereço de e-mail" and "Senha". The "Senha" field contains a single character, a dot. A red error message below the field says "Senha deve ter, pelo menos, 4 caracteres". A blue "Login" button is at the bottom. A red ribbon in the top right corner of the page says "DEBUG".

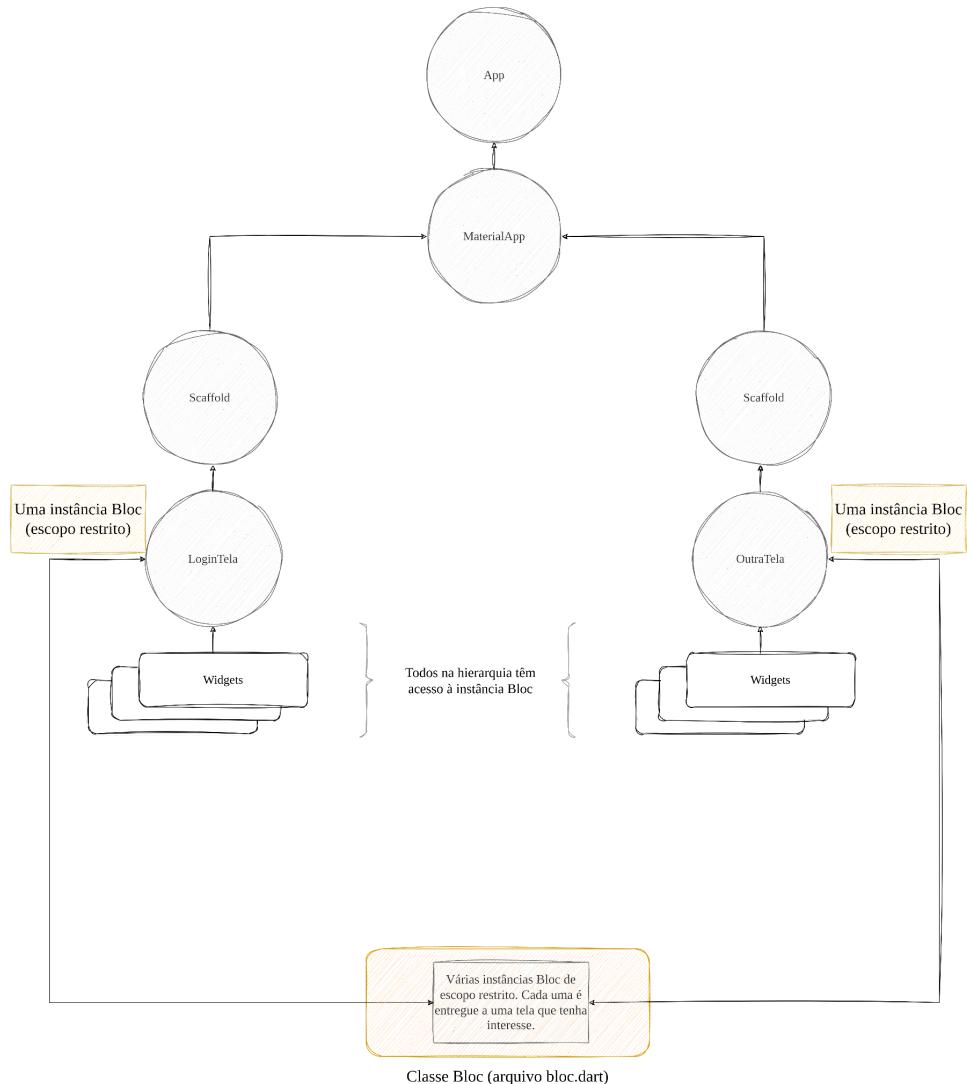
Prossiga atualizando o campo de senha, atualizando-o para que contenha, pelo menos, quatro caracteres. Veja que a mensagem de erro desaparece.



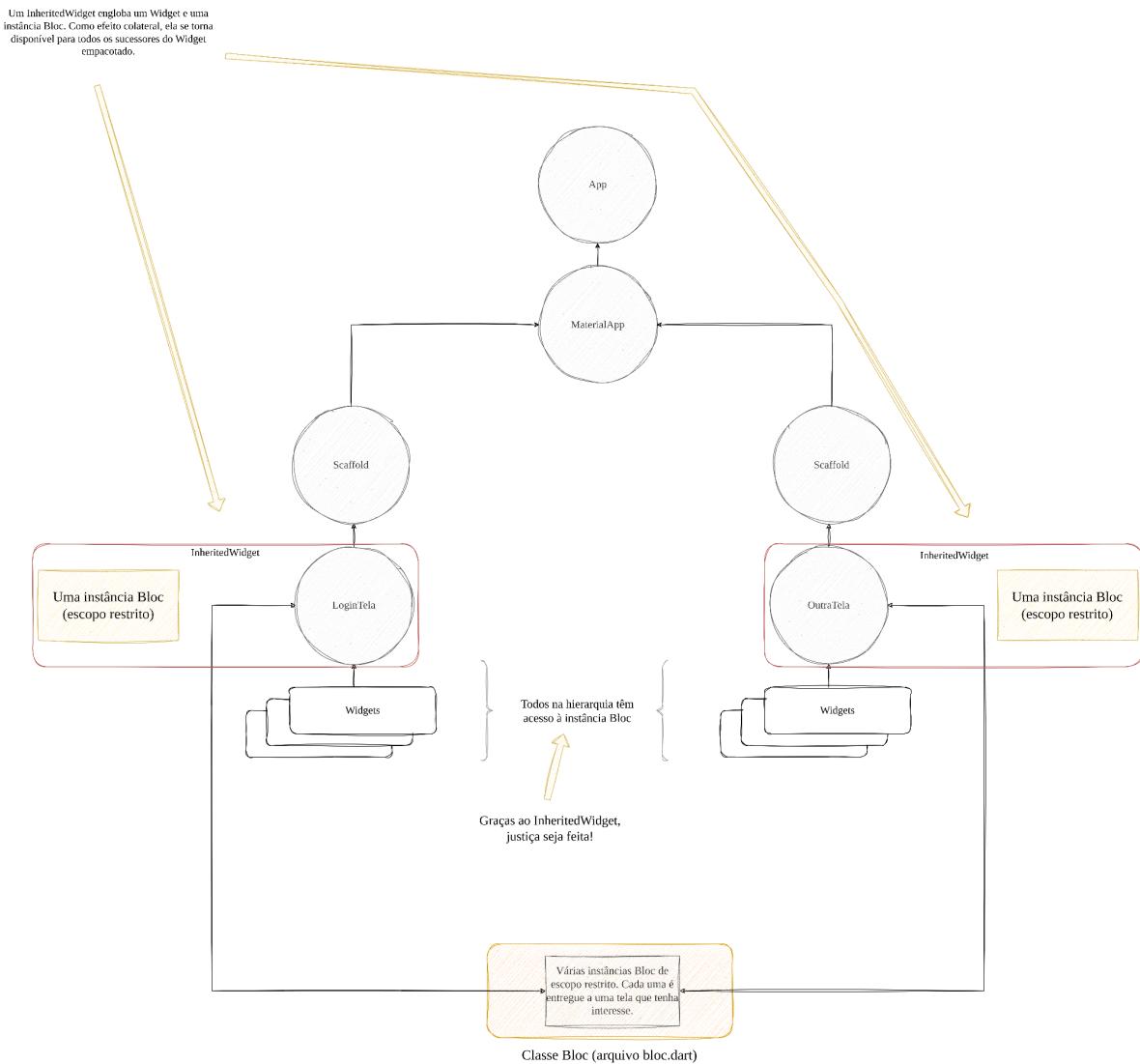
A screenshot of a web browser window titled "Login". The address bar shows "localhost:36701". The page contains a form with two fields: "Endereço de e-mail" and "Senha". The "Senha" field contains four dots. The error message from the previous screenshot is no longer present. A blue "Login" button is at the bottom. A red ribbon in the top right corner of the page says "DEBUG".

## Utilizando múltiplas instâncias Bloc de escopo restrito

Nesta seção, vamos refatorar a aplicação, ilustrando o uso da abordagem em que temos múltiplas instâncias Bloc de escopo restrito. Veja novamente a figura que ilustra esta estratégia, para relembrar.

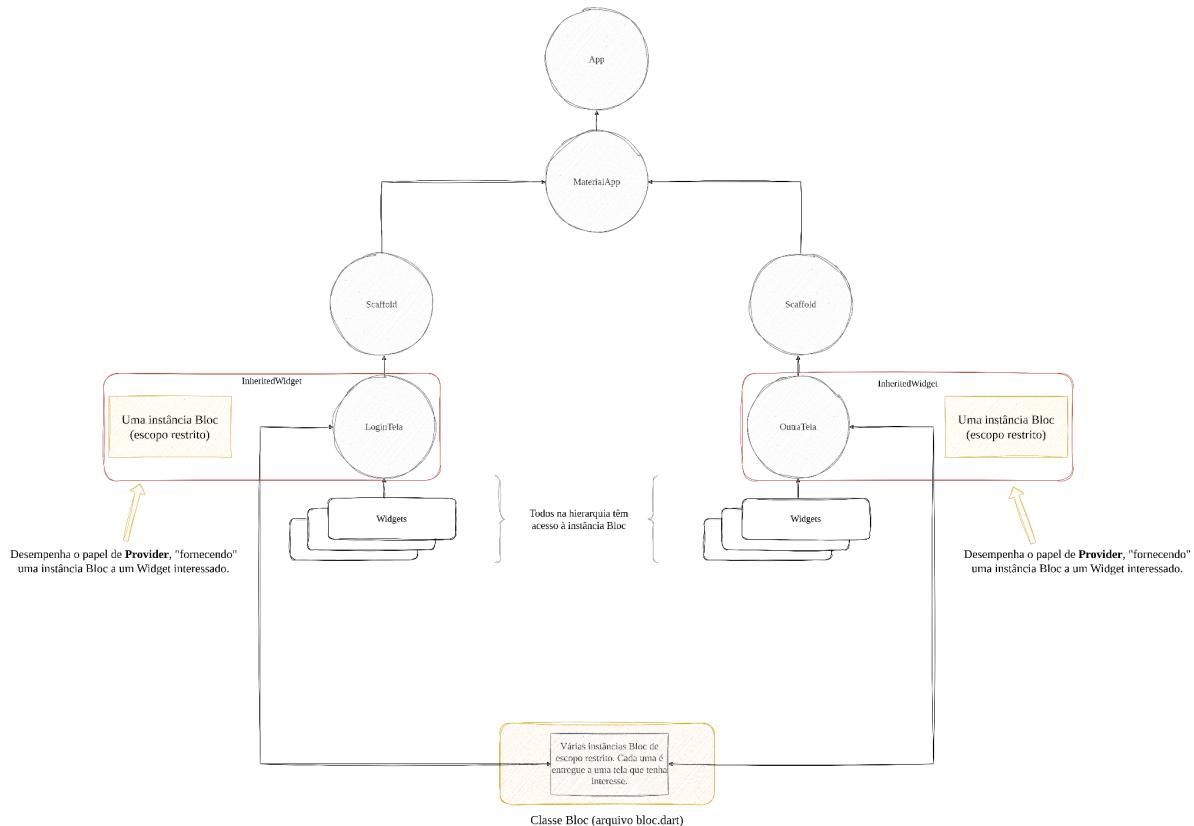


Há um detalhe que esta figura não revela: Os Widgets filhos têm acesso ao bloc de escopo restrito que foi entregue a seu antecessor graças ao uso de um **InheritedWidget**. Veja uma nova versão da figura mais completa tecnicamente.



## Implementando um Widget Provider

Para implementar a solução baseada em múltiplas instâncias Bloc de escopo restrito, vamos escrever uma classe chamada **Provider**. Ou seja, será uma classe responsável por desempenhar o papel de “fornecedor”. Ela “fornecerá” uma instância Bloc a um Widget interessado. Daí o nome. Veja a figura.



Comece criando um arquivo chamado **provider.dart** na pasta **blocs**. Inicialmente, importamos o Flutter e o Bloc.

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

A classe Provider que vamos escrever é subclasse de InheritedWidget.

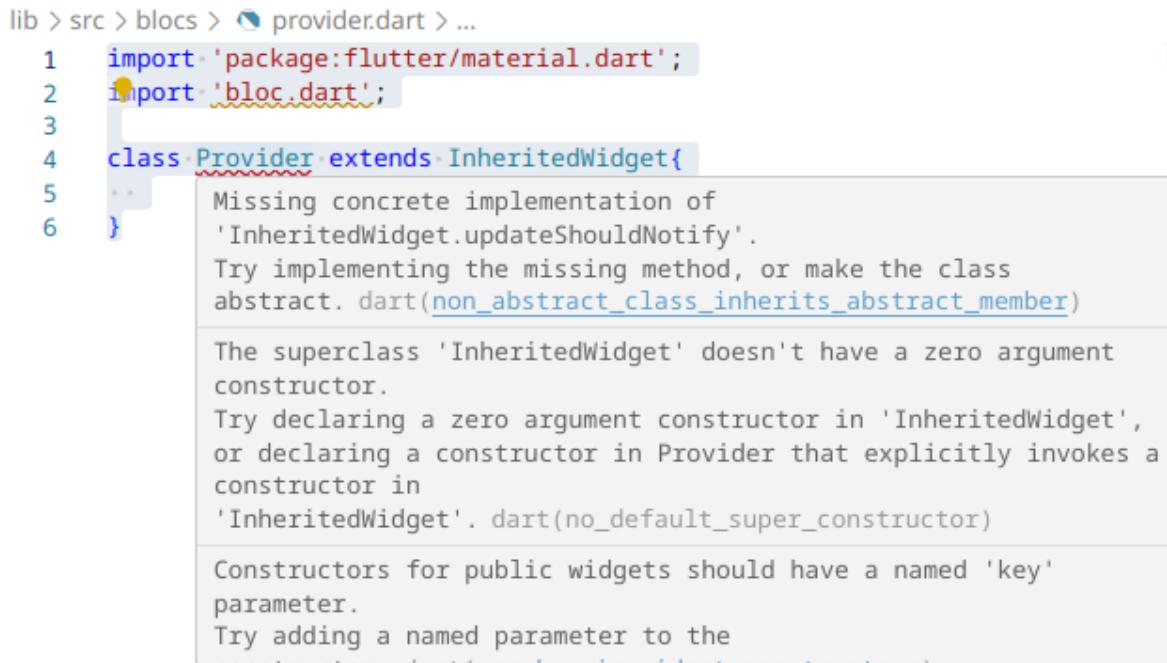
```
import 'package:flutter/material.dart';
import 'bloc.dart';

class Provider extends InheritedWidget{



}
```

Como ainda não compila, você passa o mouse sobre o “vermelho” (sem clicar) e descobre a razão.



```

lib > src > blocs > provider.dart > ...
1 import 'package:flutter/material.dart';
2 import 'bloc.dart';
3
4 class Provider extends InheritedWidget{
5
6 }

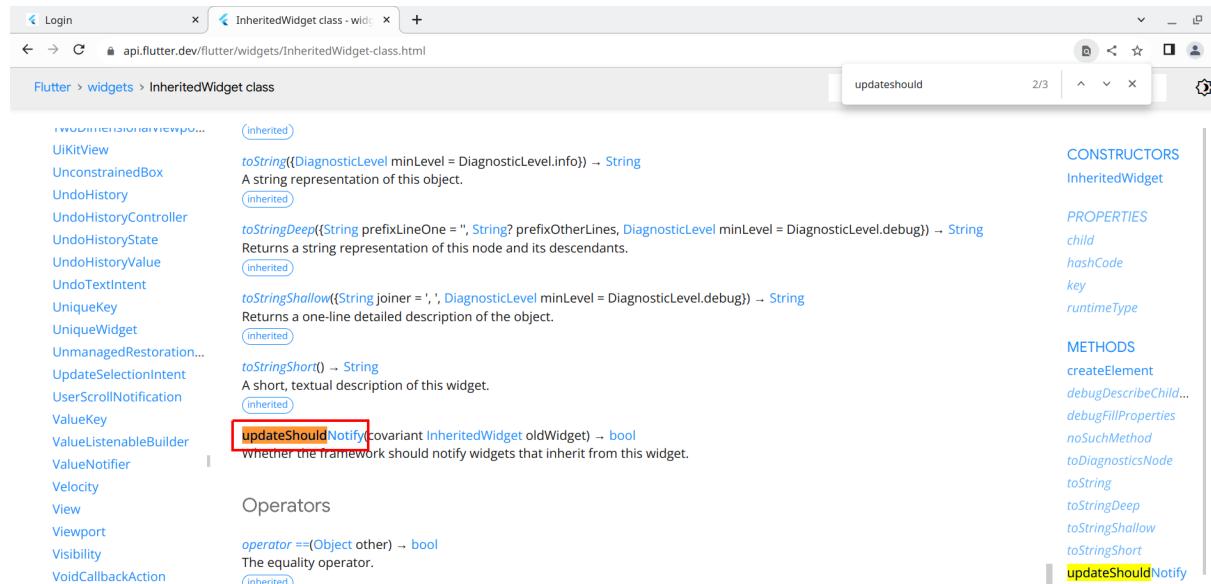
Missing concrete implementation of
'InheritedWidget.updateShouldNotify'.
Try implementing the missing method, or make the class
abstract. dart(non\_abstract\_class\_inherits\_abstract\_member)
The superclass 'InheritedWidget' doesn't have a zero argument
constructor.
Try declaring a zero argument constructor in 'InheritedWidget',
or declaring a constructor in Provider that explicitly invokes a
constructor in
'InheritedWidget'. dart(no\_default\_super\_constructor)
Constructors for public widgets should have a named 'key'
parameter.
Try adding a named parameter to the

```

A mensagem diz que estamos lidando com uma classe concreta (Provider) que não implementa um método abstrato chamado **updateShouldNotify**. Precisamos ler a sua documentação para entender como fazê-lo, bem como o seu propósito. Visite a documentação de `InheritedWidget`.

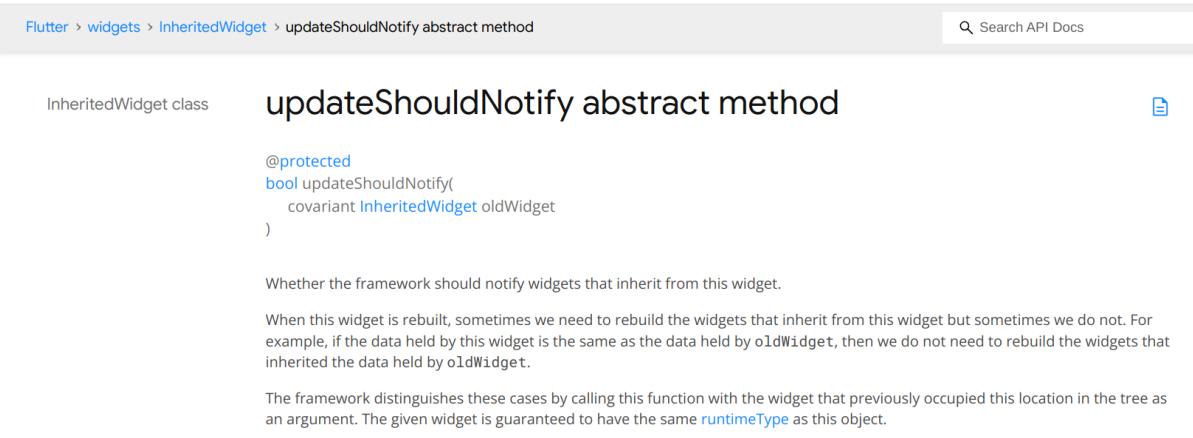
<https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html>

Encontre o método `updateShouldNotify`.



`updateShouldNotify` (covariant `InheritedWidget` oldWidget) → `bool`  
Whether the framework should notify widgets that inherit from this widget.

Clique sobre o seu nome e leia a sua documentação.



Flutter > widgets > InheritedWidget > updateShouldNotify abstract method

Search API Docs

InheritedWidget class **updateShouldNotify** abstract method

```
@protected
bool updateShouldNotify(
    covariant InheritedWidget oldWidget
)
```

Whether the framework should notify widgets that inherit from this widget.

When this widget is rebuilt, sometimes we need to rebuild the widgets that inherit from this widget but sometimes we do not. For example, if the data held by this widget is the same as the data held by oldWidget, then we do not need to rebuild the widgets that inherited the data held by oldWidget.

The framework distinguishes these cases by calling this function with the widget that previously occupied this location in the tree as an argument. The given widget is guaranteed to have the same `runtimeType` as this object.

Para entender o que a documentação diz, precisamos lembrar que os Widgets são imutáveis. Quando a tela precisa ser atualizada, o Widget nela existente é descartado e outro é construído, atualizado a fim de exibir os novos “dados”. Observe esse trechinho da documentação:

*“For example, if the data held by this widget is the same as the data held by oldWidget, then we do not need to rebuild the widgets that inherited the data held by oldWidget.”*

Intuitivo, não? A atualização da tela depende de dados (variáveis de estado). O método `updateShouldNotify` responde (ele devolve boolean) se a tela deve ser atualizada ou não. Enquanto prosseguimos com o passo a passo, vá pensando sobre qual condição faz sentido envolver no teste realizado pelo método `updateShouldNotify` para a aplicação que estamos desenvolvendo. Enquanto isso, vamos apenas escrever uma versão que devolve `true` de maneira incondicional, deixando o compilador feliz.

```
class Provider extends InheritedWidget{
    bool updateShouldNotify(covariant InheritedWidget oldWidget) => true;
}
```

**Nota.** Utilizamos a palavra reservada “**covariant**” do Dart. Como o nome sugere, ela serve para implementarmos o conceito de covariância. Ele está relacionado à maneira como os tipos de dados podem mudar (ou “variar”) em relação uns aos outros através de hierarquias de herança ou de implementação de interfaces.

Como exemplo, considere o seguinte cenário.

1. Há animais dos tipos Cachorro, Gato e Papagaio.
2. Cachorros e Papagaios podem fazer amizade com quaisquer tipos de animais.
3. Gatos somente fazem amizade com gatos.

Começamos escrevendo uma superclasse para representar todos os tipos.

Visite o DartPad para fazer o teste a seguir.

<https://dartpad.dev/>

```
abstract class Animal{
  //lista de amigos, _ para encapsular (tipo o private)
  final _amigos = [];
  //animais têm um nome
  final nome;
  //construtor que recebe nome e atribui à variável de instância
  Animal(this.nome);
  //método sem corpo é abstrato
  void fazerAmizade(Animal a);
  //para as subclasses chamarem
  void adicionar(Animal a){
    _amigos.add(a);
  }
  //representação textual
  String toString(){
    return nome;
  }
  //exibimos os amigos para testar
  void exibir (){
    //a exibição da lista causa a exibição de cada bicho, o que causa a
    //execução do método toString
    print(_amigos);
  }
}
```

Depois, escrevemos as classes Papagaio e Cachorro. Testamos, mostrando que eles cachorros topam ser amigos de todos, assim como os papagaios.

```

abstract class Animal{
  ...
}

class Cachorro extends Animal{
  Cachorro(super.nome);
  void fazerAmizade(Animal a){
    adicionar(a);
  }
}

class Papagaio extends Animal{
  Papagaio(super.nome);
  void fazerAmizade(Animal a){
    adicionar(a);
  }
}

void main(){
  var c1 = Cachorro('Odie');
  var c2 = Cachorro('Luna');
  var p1 = Papagaio('Zazu');
  var p2 = Papagaio('Lola');
  //c1 é amigo de c2, e p2;
  c1.fazerAmizade(c2);
  c1.fazerAmizade(p2);
  //p1 também é amigo de c2 e p2
  p1.fazerAmizade(c2);
  p1.fazerAmizade(p2);
  //amizades duradouras
  c1.exibir();
  p1.exibir();
}

```

Veja o resultado, ao executar.



```

23 //exibimos os amigos para testar
24 void exibir (){
25   //a exibição da lista causa a exibição de cada bicho, o que causa a execu
26   //do método toString
27   print(_amigos);
28 }

```

chilly-flora-1833 local edits

Run

Console

[Luna, Lola]  
[Luna, Lola]

Para representar Gatos, escrevemos a seguinte classe. Observe que **ela não atende às regras de negócio**, pois admite que gatos façam amizade com outros tipos de bichos.

```

abstract class Animal{
    ...
}

class Cachorro extends Animal{
    ...
}

class Papagaio extends Animal{
    ...
}

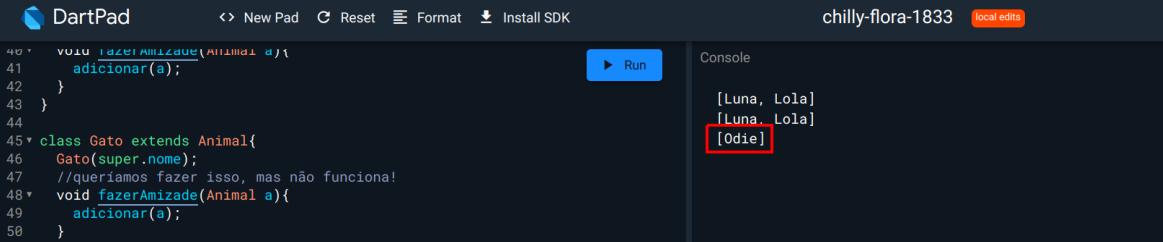
class Gato extends Animal{
    Gato(super.nome);
    //aceita qualquer tipo de animal, não pode!
    void fazerAmizade(Animal a){
        adicionar(a);
    }
}

void main(){
    var c1 = Cachorro('Odie');
    ...
    //amizades duradouras
    c1.exibir();
    p1.exibir();

    //Garfield amigo do Odie!!!!
    var g1 = Gato('Garfield');
    g1.fazerAmizade(c1);
    g1.exibir();
}

```

Execute e veja o resultado.



DartPad

chilly-flora-1833 local edits

```

40    ...
41    adicionar(a);
42 }
43 }
44
45 class Gato extends Animal{
46     Gato(super.nome);
47     //queríamos fazer isso, mas não funciona!
48     void fazerAmizade(Animal a){
49         adicionar(a);
50     }
51 }

```

Run

Console

[Luna, Lola]  
[Luna, Lola]  
[Odie]

Para corrigir, tentamos ajustar a assinatura da sobrescrita do método `fazerAmizade` da classe `Animal`, trocando o tipo do bicho recebido como parâmetro. Agora somente gatos são aceitos.

```

class Gato extends Animal{
  Gato(super.nome);
  //queríamos fazer isso, mas não funciona!
  void fazerAmizade(Gato a){
    adicionar(a);
  }
}

```

A coisa não funciona, pois não é mais uma sobrescrita válida. Veja o resultado quando tentamos executar.

DartPad

Console

```

Error: The parameter 'a' of the method 'Gato.fazerAmizade' has type 'Gato', wh
- 'Gato' is from 'package:dartpad_sample/main.dart' ('lib/main.dart').
- 'Animal' is from 'package:dartpad_sample/main.dart' ('lib/main.dart').
void fazerAmizade(Gato a){
```

lib/main.dart:11:8:

Info: This is the overridden method ('fazerAmizade').

void fazerAmizade(Animal a);

lib/main.dart:71:19:

Error: The argument type 'Cac
- 'Cachorro' is from 'package:dartpad\_sample/main.dart' ('lib/main.dart').

Documentation

line 48 • 'Gato.fazerAmizade' (void Function(Gato)) isn't a valid override of 'Animal.fazerAmizade' (void Function(Animal)). (view docs)

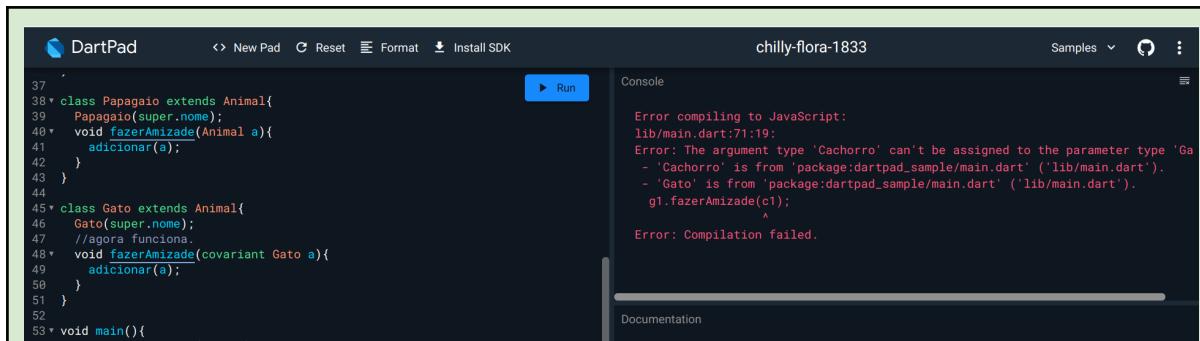
Ou seja, a covariância de tipo não acontece por padrão. Se um método recebe Animal, uma sobrescrita sua deve receber Animal também. É aí que entra a palavra covariant. Quando a utilizamos, estamos dizendo que este fenômeno é desejado neste caso. Ajuste como a seguir e teste novamente.

```

...
class Gato extends Animal{
  Gato(super.nome);
  //agora funciona, por causa do covariant
  void fazerAmizade(covariant Gato a){
    adicionar(a);
  }
}
...

```

Ao executar, observe que o erro é outro. É justamente o que desejamos: o compilador não admite que tentemos fazer com que Gatos tenham amizade com outros bichos.



```

37
38* class Papagaio extends Animal{
39  Papagaio(super.nome);
40* void fazerAmizade(Animal a){
41  adiciona(a);
42  }
43  }
44
45* class Gato extends Animal{
46  Gato(super.nome);
47  //agora funciona.
48* void fazerAmizade(covariant Gato a){
49  adiciona(a);
50  }
51  }
52
53* void main(){}

```

Console

```

Error compiling to JavaScript:
lib/main.dart:71:19:
Error: The argument type 'Cachorro' can't be assigned to the parameter type 'Animal'.
- 'Cachorro' is from package:dartpad_sample/main.dart ('lib/main.dart').
- 'Gato' is from package:dartpad_sample/main.dart ('lib/main.dart').
  g1.fazerAmizade(c1);
                                         ^
Error: Compilation failed.

```

Para consertar, é preciso remover a linha a seguir.

```

...
//Garfield não pode mais ser amigo do Odie
var g1 = Gato('Garfield');
//g1.fazerAmizade(c1);
g1.exibir();
...

```

A classe Provider ainda não compila, pois a classe InheritedWidget não possui um construtor de lista de parâmetros vazia, e ele é chamado por padrão pelo construtor padrão de Provider. Por isso, vamos escrever um construtor que recebe dois parâmetros nomeados e os repassa para o construtor da superclasse (InheritedWidget):

- O primeiro é do tipo Key e é opcional (ainda não estamos usando esse recurso)
- O segundo é um Widget. É o Widget que será empacotado pelo Provider.

**Nota.** Um objeto Key é um identificador de Widgets (e outros elementos). O Flutter pode utilizá-lo para decidir entre apenas atualizar parte da tela ou construir um novo Widget para ela, o que pode ter impacto no desempenho da aplicação. Se desejar, leia mais aqui:

<https://api.flutter.dev/flutter/foundation/Key-class.html>

Observe.

```

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

class Provider extends InheritedWidget{

  Provider({Key? key, required Widget child}): super(key: key , child: child);

  bool updateShouldNotify(covariant InheritedWidget oldWidget) => true;

}

```

Um Provider precisa fornecer acesso a um Bloc ao Widget que ele pretende empacotar. Assim, nossa classe Provider terá uma instância de Bloc.

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

class Provider extends InheritedWidget{

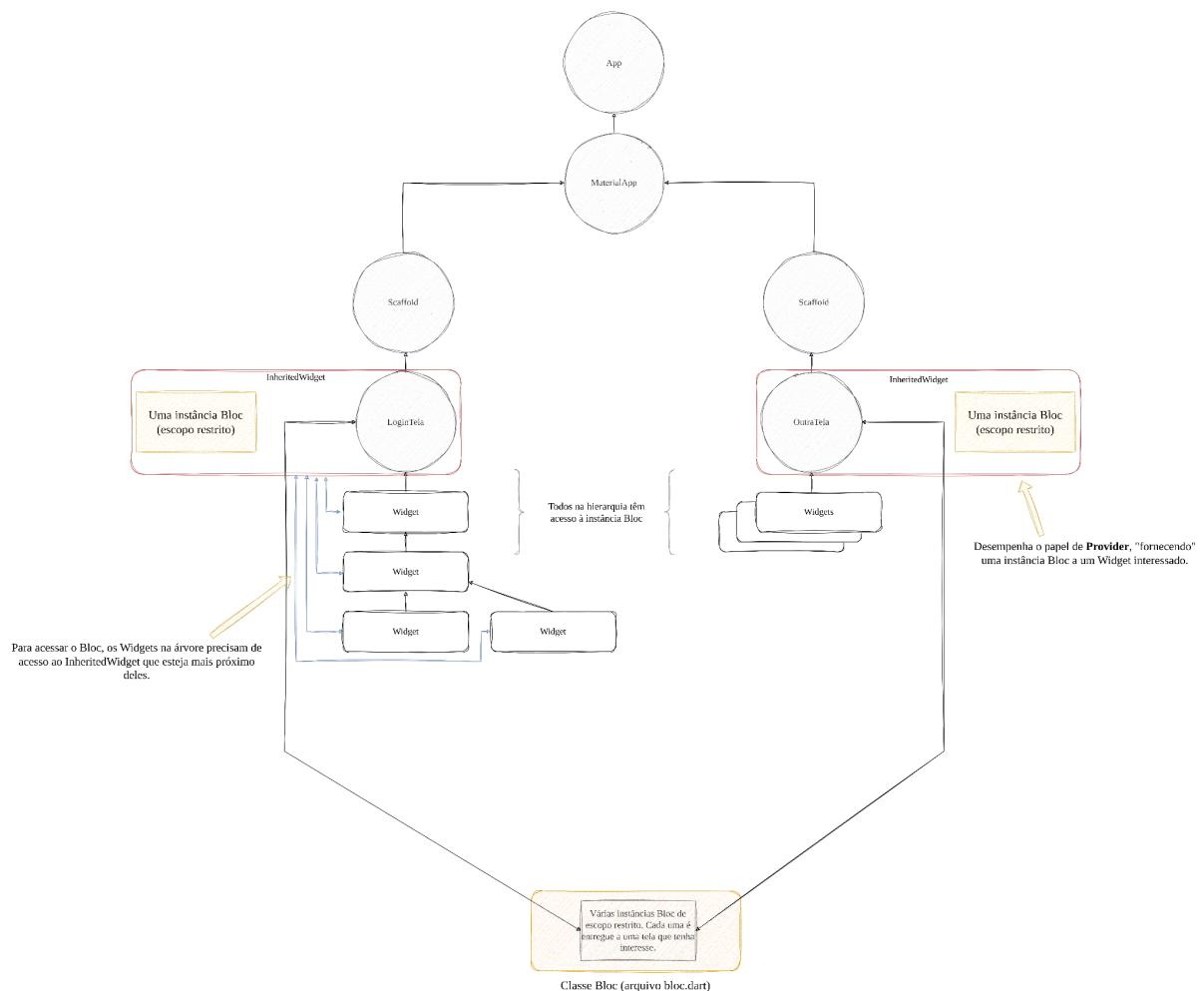
  Provider({Key? key, required Widget child}): super(key: key , child: child);

  final bloc = Bloc();

  bool updateShouldNotify(covariant InheritedWidget oldWidget) => true;

}
```

Eventualmente, os Widgets na árvore precisarão acessar o Bloc. Para isso, precisamos viabilizar que eles encontrem o InheritedWidget que possui o Bloc. De baixo para cima, é o primeiro InheritedWidget na árvore que eles possam encontrar.



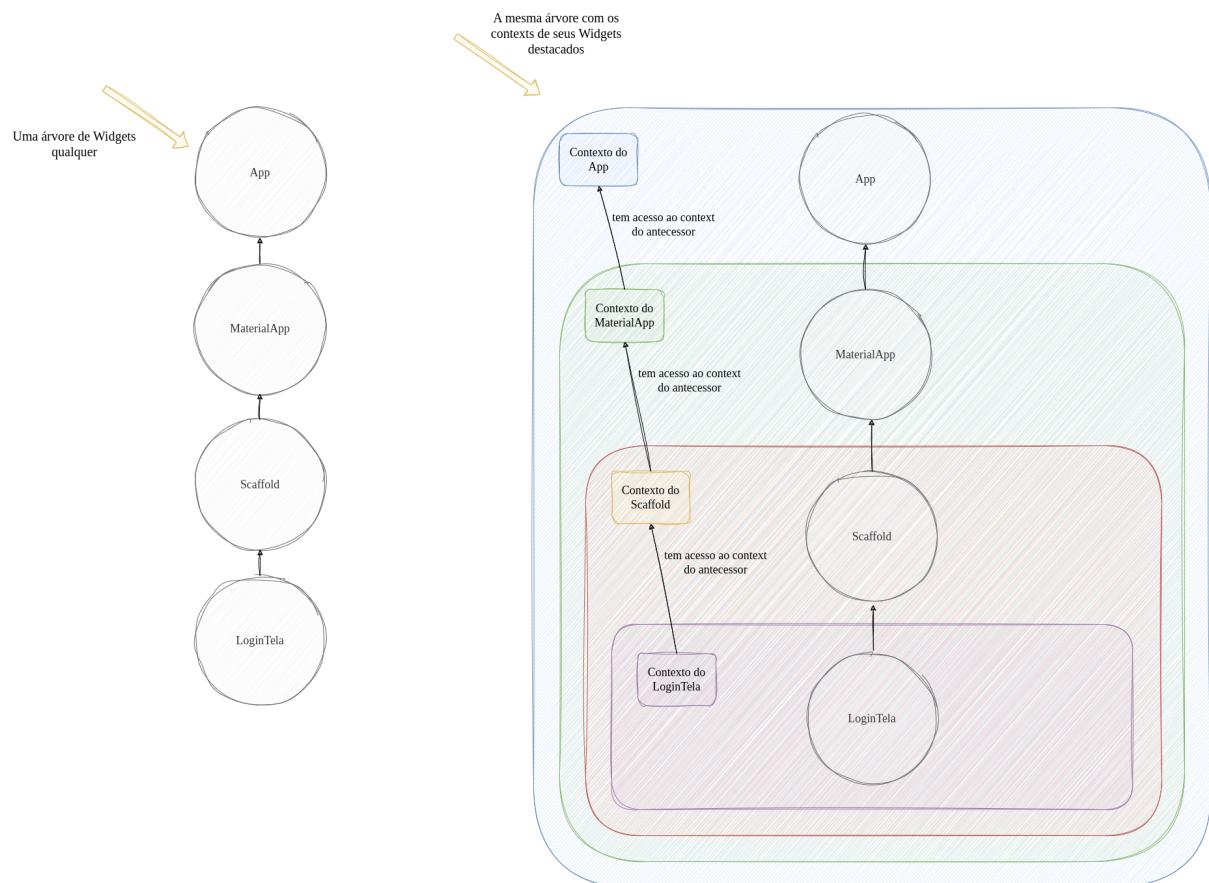
Para resolver este problema, aplicamos uma convenção bastante comum.

- Escrevemos um método chamado “of”.
  - Ele recebe um BuildContext (pois ele dá acesso ao método que realiza a obtenção do InheritedWidget de interesse, na árvore)
  - Ele devolve o dado de interesse (o Bloc inteiro, neste caso)
  - Ele é estático, para que não seja necessário construir uma instância para chamá-lo.
- O método of usa o método **dependOnInheritedWidgetOfExactType** de BuidContext para obter o primeiro InheritedWidget da árvore e dele acessar o Bloc, devolvendo-o a seguir.

**Nota.** O nome **of** é apenas uma convenção. Esse nome não é obrigatório. Seu uso está associado à funcionalidade de busca de uma instância **de(of)** algum tipo de dado ou objeto na árvore.

- Quando olhamos para um Widget e, a partir dele, tentamos encontrar um InheritedWidget “subindo” na árvore, é possível que não encontremos nenhum. Assim, o método **dependOnInheritedWidgetOfExactType** pode devolver null. Utilizamos o operador “!” dizendo ao Dart que garantimos que a expressão será diferente de null. Em outras palavras, saltamos do bote que não afunda e topamos nos molhar, garantindo ao Dart que sabemos nadar. Se não soubermos (se a expressão for null), o Dart lava as mãos (nós mesmos dissemos que ele pode fazer isso quando saltamos do bote) e a gente se afoga (tomamos uma NullPointerException). Ou seja, falamos explicitamente a ele que estamos dispostos a lidar com null.

A figura a seguir mostra uma propriedade importante a respeito do objeto `BuildContext`. Cada Widget tem o seu próprio contexto e, a partir de seu contexto, um Widget pode obter o contexto do seu antecessor. É o que viabiliza a busca de baixo para cima, na árvore.



Veja como fica o código.

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

class Provider extends InheritedWidget{

  Provider({Key? key, required Widget child}): super(key: key, child: child);

  final bloc = Bloc();

  bool updateShouldNotify(covariant InheritedWidget oldWidget) => true;

  static Bloc of(BuildContext context){
    //o operador ! garante que a expressão que o antecede
    // (context.dependOnInheritedWidgetOfExactType<Provider>(), neste caso) é diferente
    // de null podemos acessar a propriedade bloc sem casting pois a classe é genérica
    // (informamos o tipo <Provider>)
    return context.dependOnInheritedWidgetOfExactType<Provider>()!.bloc;
  }
}
```

## Substituindo a instância Bloc global por instâncias de escopo restrito

Em nossa primeira implementação, utilizamos uma única instância Bloc de escopo global. Agora que terminamos de implementar um Provider, vamos substituir esta implementação por aquela em que fazemos uso de múltiplas instâncias Bloc de escopo restrito.

O primeiro passo é empacotar o Widget **MaterialApp** com um Provider. Isso pode ser feito no arquivo em que ele foi definido: **app.dart**.

```
import 'package:flutter/material.dart';
import '../src/telas/login_tela.dart';
//importamos o arquivo provider
import 'blocs/provider.dart';
class App extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    //empacotamos o MaterialApp com um Provider
    return Provider(
      child: MaterialApp(
        title: 'Login',
        home: Scaffold(
          body: LoginTela(),
        )
      )
    );
  }
}
```

Não vamos mais utilizar a instância Bloc global. Sempre que precisarmos de uma instância Bloc, vamos construir um Provider. Cada Provider já nasce com a instância Bloc que promete fornecer. Por isso, no arquivo **bloc.dart**, vamos apagar a instância global.

```

import 'dart:async';
import 'validators.dart';
class Bloc with Validators{
  ...
}

//essa é a instância global e ela não será mais usada, pode apagar ou
comentar
//final bloc = Bloc();

```

Abra o arquivo **login\_tela.dart** para perceber que o bloc global não está mais disponível.



Para ajustar, começamos importando o arquivo **provider.dart** no arquivo **login\_tela.dart**. Embora não mais utilizemos a instância global (até porque ela nem existe mais), vamos manter a importação do arquivo **bloc.dart**. Daqui a pouco vamos aplicar o tipo **Bloc** em variáveis recebidas por métodos internos.

```

import 'package:flutter/material.dart';
import '../blocs/bloc.dart';
import '../blocs/provider.dart';
class LoginTela extends StatelessWidget{
  ...
}

```

Depois disso, construímos uma instância Bloc utilizando o método of do Provider. Ele recebe um BuildContext e, felizmente, estamos num método (build) que tem acesso a um.

```
...
class LoginTela extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    final bloc = Provider.of(context);
    return Container(
      //20 pixels de margem esquerda, direita, em cima e embaixo
      margin: EdgeInsets.all(20.0),
    ...
  }
}
```

Os Widgets que precisam acessar a instância Bloc são construídos por métodos separados, altamente coesos. Eles não têm acesso à instância Bloc construída, já que ela foi definida pelo método build. Dado que o método build os chama, ele pode entregar a instância Bloc como argumento. Claro, a lista de parâmetros de cada um deve ser ajustada de acordo.

```
...
Widget build(BuildContext context) {
  final bloc = Provider.of(context);
  return Container(
    //20 pixels de margem esquerda, direita, em cima e embaixo
    margin: EdgeInsets.all(20.0),
    child: Column(
      children: [
        emailField(bloc),
        passwordField(bloc),
        Container(
          margin: EdgeInsets.only(top: 12.0),
          child: Row(
            ...
          )
        )
      ]
    )
  Widget emailField(Bloc bloc) {
    return StreamBuilder(
      ...
    )
  }

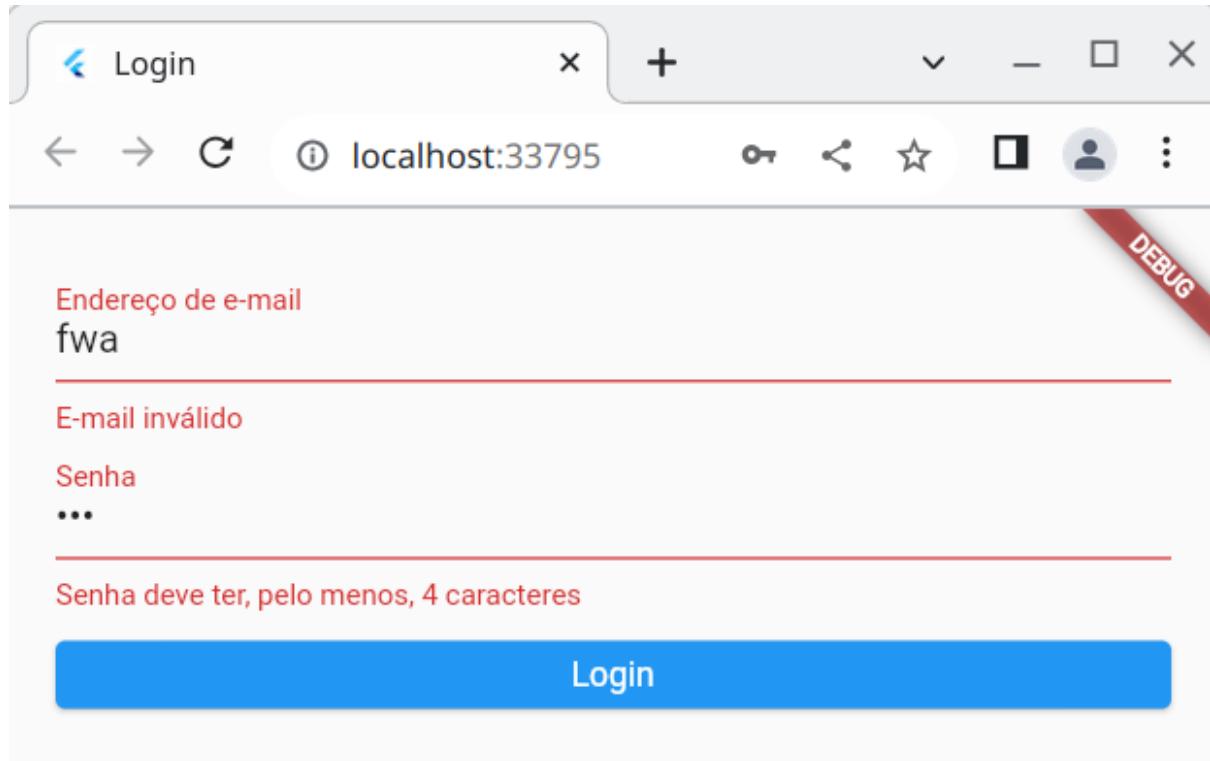
  Widget passwordField(Bloc bloc) {
    return StreamBuilder(
      ...
    )
  }

  Widget submitButton() {
    ...
  }
}
```

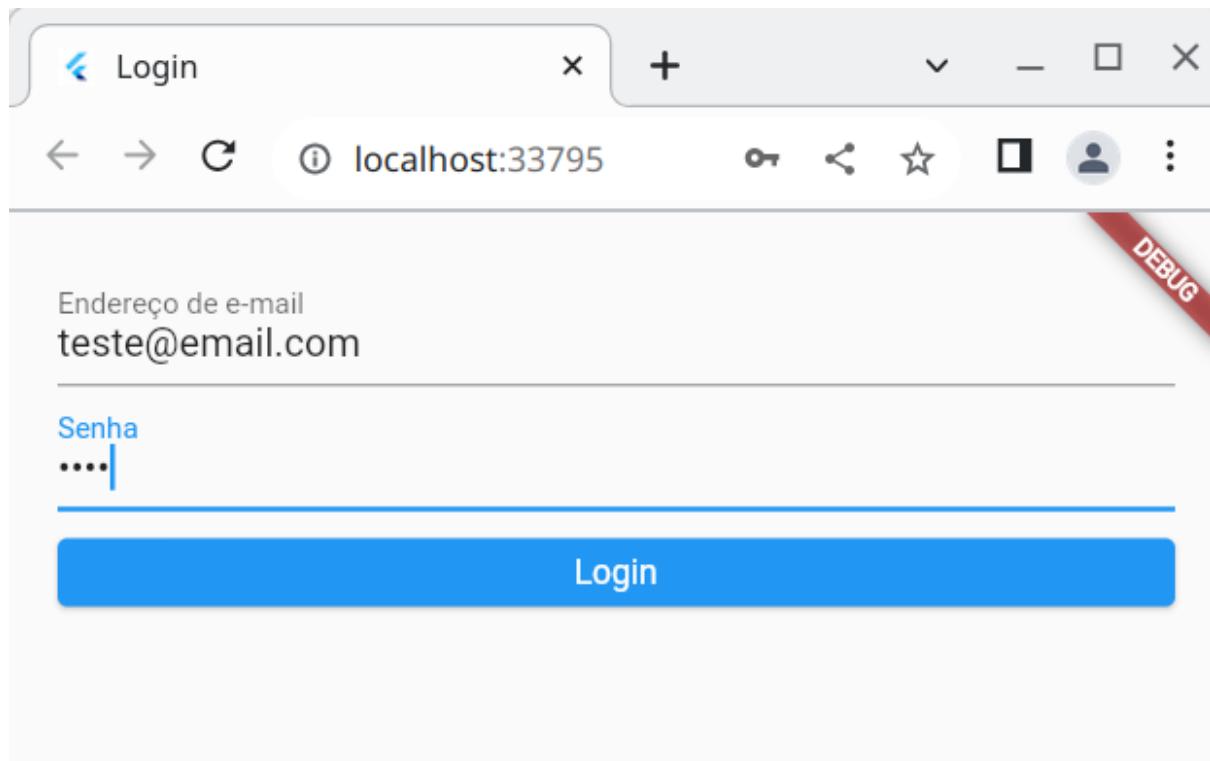
Certifique-se de que a aplicação está em execução para fazer novos testes.

```
flutter run
```

Veja como deve ficar com e-mail e senha inválidos.



E agora com e-mail e senha válidos.



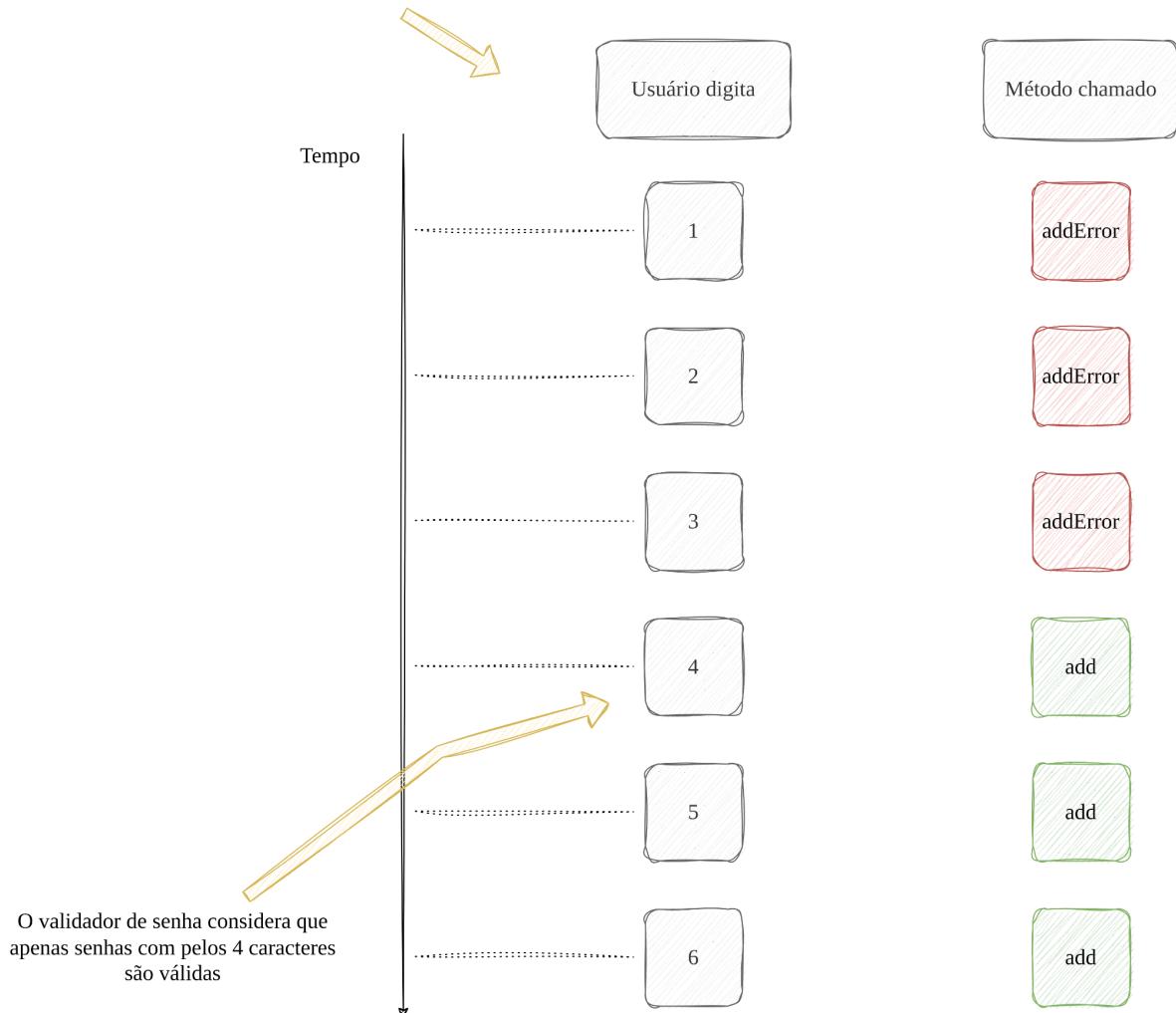
## Submissão do form

Desejamos permitir que o form seja enviado apenas quando o form estiver num estado válido. Ou seja, quando seus dois campos contiverem valores condizentes com as validações realizadas. Para isso, precisamos olhar para ambos os streams do Bloc. Lembre-se que eles funcionam assim:

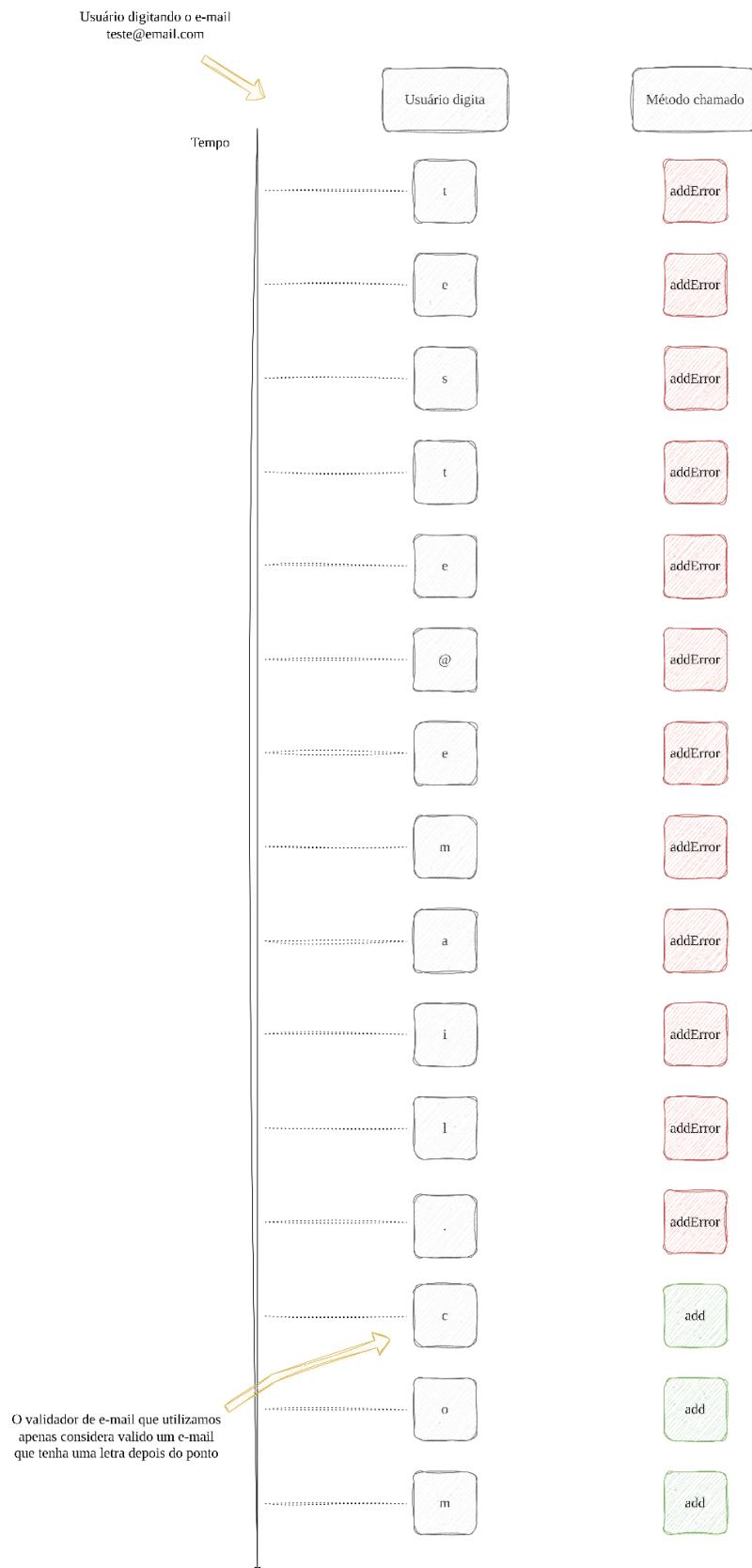
- Quando há um erro, o método addError é chamado
- Quando não há erro algum, o método add é chamado

Veja um exemplo conforme o tempo passa, quando o usuário digita uma senha.

Usuário digitando a senha 123456



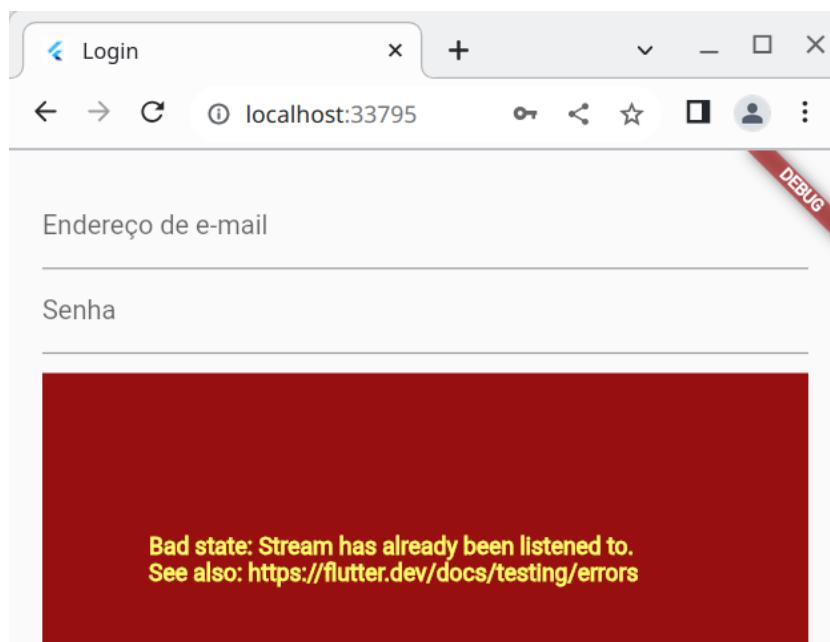
O mesmo vale para o validador de e-mail. Veja.



Precisamos encontrar uma forma de manipular os dois streams simultaneamente. Talvez você pense em construir o botão empacotado por um StreamBuilder. Seu Stream seria um dos dois, de e-mail ou de senha. Para desabilitar o botão, bastaria associar null à sua propriedade onPressed. Ou seja, quando o método hasError devolvesse true, associaríamos null a ela. Quando ele devolvesse false, associaríamos uma função. Depois disso, poderíamos empacotar o StreamBuilder utilizando outro StreamBuilder, a fim de condicionar a exibição do botão à validação do outro campo. Veja como **ficaria**. É uma ideia razoável, verdade. Estamos no arquivo **login\_tela.dart**.

```
...
Widget submitButton(Bloc bloc) {
  return StreamBuilder(
    //tentando usar o stream de email
    stream: bloc.email,
    builder: (context, AsyncSnapshot<String> snapshot) {
      return ElevatedButton(
        //se tiver erro, associamos null à propriedade onPressed,
        //desabilitando o botão
        onPressed: snapshot.hasError ? null : (){},
        child: Text('Login'),
      );
    }
  );
}
...
```

Embora razoável, essa ideia não funciona. Veja a mensagem de erro.



Ela revela que somente podemos “escutar” o resultado de um stream uma única vez. Está tudo bem. Você foi um tanto criativo com essa ideia!

Precisamos de uma alternativa. Por isso, volte a implementação ao que tínhamos antes.

```
...
Widget submitButton(Bloc bloc) {
  return ElevatedButton(
    onPressed: () {},
    child: Text('Login'),
  );
}
...
```

## O Pacote rxDart

O pacote rxDart é uma espécie de extensão à API de Streams e de processamento assíncrono do Dart. Veja a sua página no pub.dev.

<https://pub.dev/packages/rxdart>

E um trecho que o descreve sucintamente.

*“RxDart extends the capabilities of Dart Streams and StreamControllers.*

*Dart comes with a very decent Streams API out-of-the-box; rather than attempting to provide an alternative to this API, RxDart adds functionality from the reactive extensions specification on top of it.*

*RxDart does not provide its Observable class as a replacement for Dart Streams. Instead, it offers several additional Stream classes, operators (extension methods on the Stream class), and Subjects.”*

Observe, também, que RxDart é uma implementação Dart da API **ReactiveX**. Trata-se de uma API para programação assíncrona utilizando streams “observáveis”. Ou seja, a sua implementação se baseia no clássico padrão de projeto **Observer**.

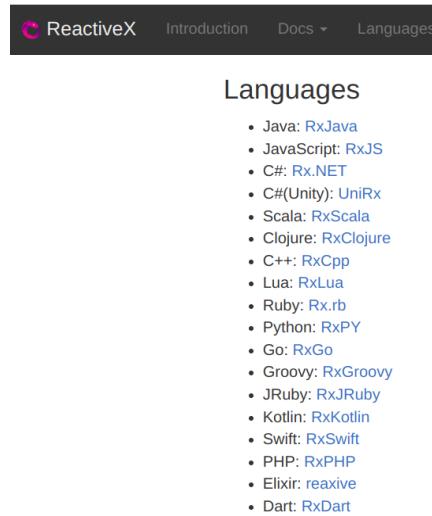
Veja a sua página oficial.

<https://reactivex.io/>

Há implementações de ReactiveX para diversas linguagens de programação. Veja a lista de linguagens para as quais há uma implementação de ReactiveX.

<https://reactivex.io/languages.html>

No momento em que este documento foi escrito, as linguagens para as quais havia uma implementação de ReactiveX eram as seguintes.

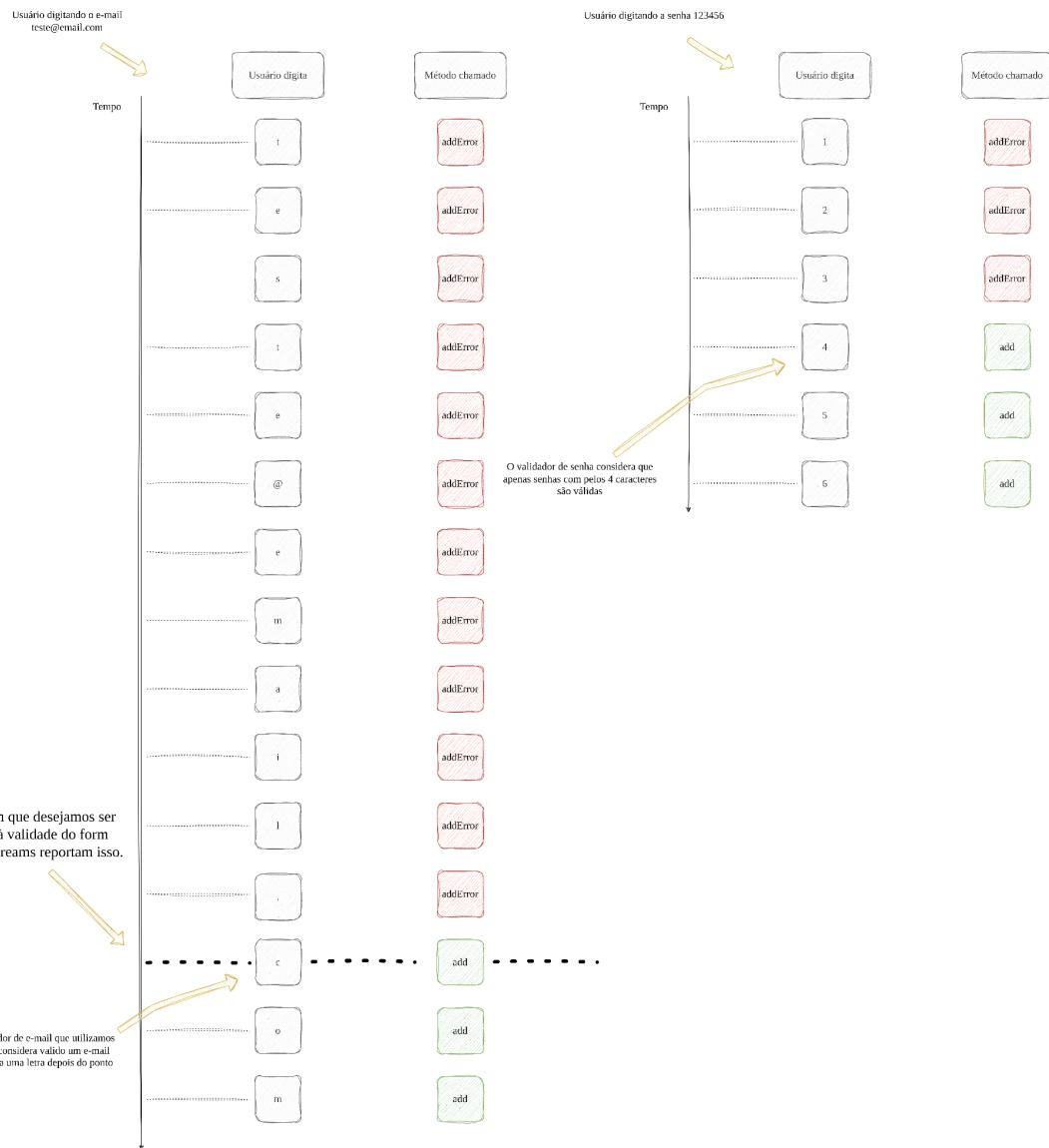


The screenshot shows a dark-themed web page for ReactiveX. At the top, there is a navigation bar with the ReactiveX logo, 'Introduction', 'Docs', and 'Languages'. The 'Languages' section is currently selected and expanded, showing a list of supported languages. The list includes:

- Java: RxJava
- JavaScript: RxJS
- C#: Rx.NET
- C#(Unity): UniRx
- Scala: RxScala
- Clojure: RxClojure
- C++: RxCpp
- Lua: RxLua
- Ruby: Rx.rb
- Python: RxPY
- Go: RxGo
- Groovy: RxGroovy
- JRuby: RxJRuby
- Kotlin: RxKotlin
- Swift: RxSwift
- PHP: RxPHP
- Elixir: reaxive
- Dart: RxDart

Vamos utilizar o pacote RxDart, implementação de ReactiveX em Dart, para resolver o problema de habilitar/desabilitar o botão em função dos streams que já possuímos.

Para isso, temos a intenção de, de alguma forma, “combinar” ou “mesclar” os streams a fim de que saibamos sobre o estado deles a partir de um único objeto. Veja a figura a seguir.



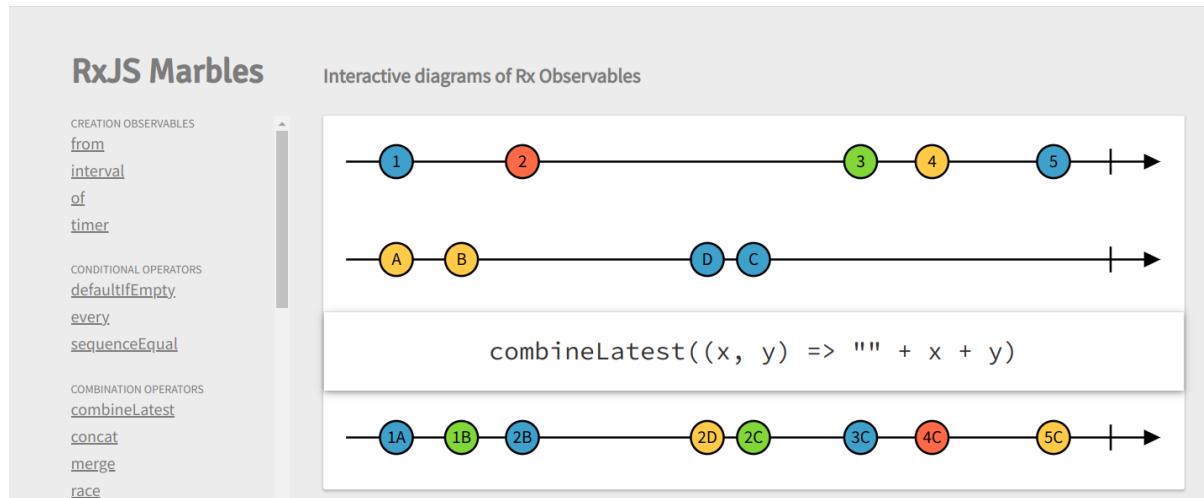
## Combinação de Streams

Para combinar os streams, vamos usar a classe `CombineLatestStream`. Veja a sua documentação.

<https://pub.dev/documentation/rxdart/latest/rx/CombineLatestStream-class.html>

O link a seguir mostra um gráfico interativo que ilustra a combinação de dois streams. Ajuste os valores numéricos e letras e veja o resultado na última linha.

<https://rxmarbles.com/#combineLatest>



Da classe `CombineLatestStream`, queremos usar o método `combine2`, já que temos 2 streams para combinar. Veja o exemplo da documentação.

rxdart package > documentation > rx > `CombineLatestStream<T, R> class` Search API Docs

**CLASSES**

- AbstractConnectableStr...
- BehaviorSubject
- BufferCountStreamTrans...
- BufferStreamTransformer
- BufferTestStreamTransfo...
- CombineLatestStream
- CompositeSubscription

**Example with a specific number of Streams**

If you wish to combine a specific number of Streams together with proper types information for the value of each Stream, use the `combine2` - `combine9` operators.

```
CombineLatestStream.combine2(
  Stream.fromIterable([1]),
  Stream.fromIterable([2, 3]),
  (a, b) => a + b,
)
.listen(print); // prints 3, 4
```

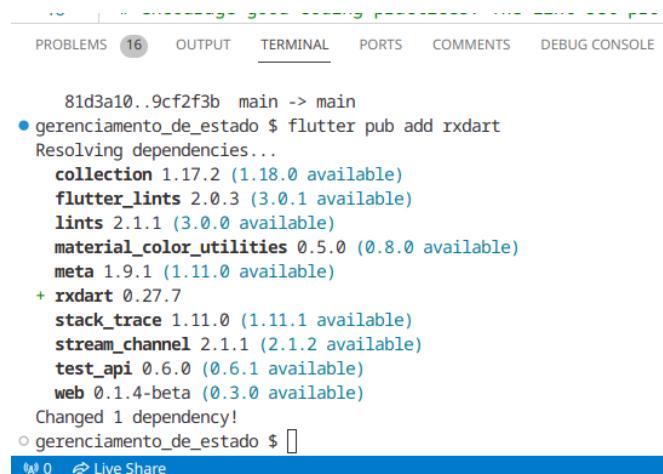
O exemplo mostra que a função `combine2` espera três argumentos:

- o primeiro Stream
- o segundo Stream
- uma função que explica como se dá a combinação dos valores produzidos pelos streams

Para instalar o pacote, use

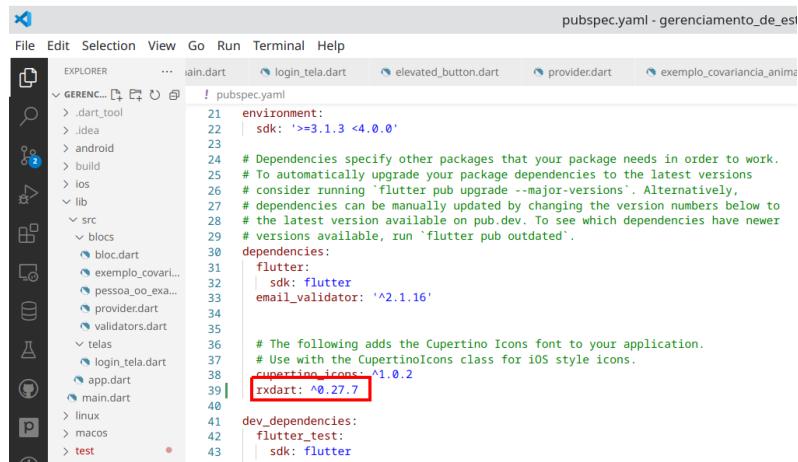
`flutter pub add rxdart`

Veja a saída esperada.



```
81d3a10..9cf2f3b main -> main
● gerenciamento_de_estado $ flutter pub add rxdart
Resolving dependencies...
  collection 1.17.2 (1.18.0 available)
    flutter_lints 2.0.3 (3.0.1 available)
    lints 2.1.1 (3.0.0 available)
    material_color_utilities 0.5.0 (0.8.0 available)
    meta 1.9.1 (1.11.0 available)
  + rxdart 0.27.7
    stack_trace 1.11.0 (1.11.1 available)
    stream_channel 2.1.1 (2.1.2 available)
    test_api 0.6.0 (0.6.1 available)
    web 0.1.4-beta (0.3.0 available)
Changed 1 dependency!
○ gerenciamento_de_estado $
```

Abra o seu arquivo **pubspec.yaml** e repare que a dependência foi adicionada por ali.



```
File Edit Selection View Go Run Terminal Help
File Explorer View Editor Terminal Help
GERENC... 16 PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE
! pubspec.yaml - gerenciamento_de_estado
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.2
  rxdart: ^0.27.7
```

**Nota.** A instalação também poderia ter sido realizada adicionando a dependência manualmente ao arquivo **pubspec.yaml** e, então, executando-se o comando **flutter pub get**.

Para usar as funcionalidades do pacote, precisamos importá-lo no arquivo **bloc.dart**.

```
import 'dart:async';
import 'validators.dart';
import 'package:rxdart/rxdart.dart';
class Bloc with Validators{
```

A seguir, escrevemos um novo método getter. Ele devolve um stream que representa a combinação dos já existentes (e-mail e password), operando da seguinte forma.

- Ele responde se o botão submit pode ser habilitado
- Aquilo que ele produz não será realmente de interesse: o conteúdo combinado dos dois streams originais não é de interesse para a decisão sobre habilitar ou não o botão. Assim, vamos apenas devolver algo como true, indicando que “sim, o form pode ser enviado”. Isso faz sentido, pois a emissão do stream combinado somente vai acontecer quando os demais fizerem uma emissão.

Ainda no arquivo **bloc.dart**, escrevemos o novo stream.

```
...
class Bloc with Validators{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController <String> ();
  final _passwordController = StreamController <String> ();

  Stream<String> get email => _emailController.stream.transform(validateEmail);

  Stream<String> get password => _passwordController.stream.transform(validatePassword);

  Stream<bool> get emailPasswordAreOk => CombineLatestStream.combine2(email, password, (e, p) => true);

  Function(String) get changeEmail => _emailController.sink.add;

  Function(String) get changePassword => _passwordController.sink.add;

  void dispose(){
    _emailController.close();
    _passwordController.close();
  }
}

...
```

De volta ao arquivo **login\_tela.dart**, passamos a utilizar o novo stream. O primeiro passo é “empacotar” o botão utilizando um StreamBuilder. Claro, entregando o stream combinado como parâmetro a ele. Observe.

```

...
Widget submitButton(Bloc bloc) {
    return StreamBuilder(
        stream: bloc.emailPasswordAreOk,
        builder: (context, snapshot) {
            return ElevatedButton(
                onPressed: () {},
                child: Text('Login'),
            );
        },
    );
}
...

```

Mantendo o padrão dos demais, vamos utilizar o sistema de tipos estático com o snapshot. Ele é um `AsyncSnapshot<bool>`.

```

...
Widget submitButton(Bloc bloc) {
    return StreamBuilder(
        stream: bloc.emailPasswordAreOk,
        builder: (context, AsyncSnapshot<bool> snapshot) {
            return ElevatedButton(
                onPressed: () {},
                child: Text('Login'),
            );
        },
    );
}
...

```

Para desabilitar o botão, lembre-se mais uma vez das propriedades que um `AsyncSnapshot` possui, visitando a sua documentação, se necessário (ou consulte a imagem a seguir).

<https://api.flutter.dev/flutter/widgets/AsyncSnapshot-class.html>

Flutter > widgets > AsyncSnapshot<T> class

const

**CLASSES**

AbsorbPointer  
Accumulator  
Action  
ActionDispatcher  
ActionListener  
Actions  
ActivateAction  
ActivateIntent  
Align  
Alignment  
AlignmentDirectional  
AlignmentGeometry  
AlignmentGeometryTwe...  
AlignmentTween  
AlignTransition  
AlwaysScrollableScrollP...  
AlwaysStoppedAnimation  
AndroidView  
AndroidViewSurface  
Animatable  
AnimatedAlign  
AnimatedBuilder

**Properties**

**AsyncSnapshot.withError**(`ConnectionState` state, `Object` error, [`StackTrace` stackTrace = `StackTrace.empty`])  
Creates an `AsyncSnapshot` in the specified state with the specified `error` and a `stackTrace`.  
`const`

**connectionState** → `ConnectionState`  
Current state of connection to the asynchronous computation.  
`[final]`

**data** → `T?`  
The latest data received by the asynchronous computation.  
`[final]`

**error** → `Object?`  
The latest error object received by the asynchronous computation.  
`[final]`

**hasData** → `bool`  
Returns whether this snapshot contains a non-null `data` value.  
`[read-only]`

**hasError** → `bool`  
Returns whether this snapshot contains a non-null `error` value.  
`[read-only]`

**hashCode** → `int`  
The hash code for this object.

Um método que resolve o nosso problema é o `hasError`. Ele resulta em

- true, se um dos dois streams tiver erro
- false, caso contrário

Com esse valor, decidimos entre

- associar null à propriedade `onPressed` do botão, o que faz com que ele seja desabilitado
- associar uma função, tornando-o clicável

```
Widget submitButton(Bloc bloc) {
  return StreamBuilder(
    stream: bloc.emailPasswordAreOk,
    builder: (context, AsyncSnapshot <bool> snapshot) {
      return ElevatedButton(
        onPressed: snapshot.hasError ? null : () {},
        child: Text('Login'),
      );
    },
  );
}
```

## Execute a aplicação

flutter run

para ver que isso não funciona.

```
login_tela.dart - gerenciamento_de_estado.dart
File Edit Selection View Go Run Terminal Help
PROBLEMS 16 OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE
lib > src > telas > login_tela.dart M > LoginTela > submitButton
61     errorText: snapshot.hasError ? snapshot.error.toSt
62     ),
63     // InputDecoration
64   ),
gerenciamento_de_estado.dart flutter run
Connected devices:
Linux (desktop) * linux * linux-x64 * Ubuntu 22.04.3 LTS 6.2.0-36-generic
Chrome (web) * chrome * web-javascript * Google Chrome 117.0.5938.149
[1]: Linux (linux)
[2]: Chrome (chrome)
Please choose one (or "q" to quit): 2
Launching lib/main.dart on Chrome in debug mode...
Waiting for connection from debug service on Chrome... 34.9s
This app is linked to the debug service: ws://127.0.0.1:43087/SMuiPZqEAUy=ws
Debug service listening on ws://127.0.0.1:43087/SMuiPZqEAUy=ws
To hot restart changes while running, press "r" or "R".
For a more detailed help message, press "h". To quit, press "q".
Error: Bad state: Stream has already been listened to.
dart-sdk/lib/internal/jsedev_runtime/private/_dev_runtime/errors.dart 294:49
    thru
dart-sdk/lib/async/stream_controller.dart 686:7
[...]
```

## Broadcast streams

Embora estejamos combinando streams, estamos de volta ao ponto em que - dessa vez, indiretamente - estamos tentando nos registrar junto a um stream mais de uma vez. Isso não é possível, a menos que tenhamos um stream capaz de fazer **broadcast**.

A documentação da classe Stream explica o fenômeno.

<https://api.flutter.dev/flutter/dart-async/Stream-class.html>

Flutter > dartasync > Stream<T> class Search API Docs

**CLASSES**

- Completer
- EventSink
- Future
- FutureOr
- MultiStreamController
- Stream
- StreamConsumer
- StreamController
- StreamIterator
- StreamSink
- StreamSubscription
- StreamTransformer
- StreamTransformerBase
- StreamView
- SynchronousStreamCon...
- Timer
- Zone
- ZoneDelegate
- ZoneSpecification

EXTENSIONS

On an error event from the source stream, the `await` for re-throws that error, which breaks the loop. The error then reaches the end of the `optionalMap` function body, since it's not caught. That makes the error be emitted on the returned stream, which then closes.

The `Stream` class also provides functionality which allows you to manually listen for events from a stream, or to convert a stream into another stream or into a future.

The `forEach` function corresponds to the `await for` loop, just as `Iterable.forEach` corresponds to a normal `for/in` loop. Like the loop, it will call a function for each data event and break on an error.

The more low-level `listen` method is what every other method is based on. You call `listen` on a stream to tell it that you want to receive events, and to register the callbacks which will receive those events. When you call `listen`, you receive a `StreamSubscription` object which is the active object providing the events, and which can be used to stop listening again, or to temporarily pause events from the subscription.

There are two kinds of streams: "Single-subscription" streams and "broadcast" streams.

A *single-subscription stream* allows only a single listener during the whole lifetime of the stream. It doesn't start generating events until it has a listener, and it stops sending events when the listener is unsubscribed, even if the source of events could still provide more. The stream created by an `async*` function is a single-subscription stream, but each call to the function creates a new such stream.

Listening twice on a single-subscription stream is not allowed, even after the first subscription has been canceled.

Single-subscription streams are generally used for streaming chunks of larger contiguous data, like file I/O.

A *broadcast stream* allows any number of listeners, and it fires its events when they are ready, whether there are listeners or not.

Broadcast streams are used for independent events/observers.

If several listeners want to listen to a single-subscription stream, use `asBroadcastStream` to create a broadcast stream on top of the non-broadcast stream.

On either kind of stream, stream transformations, such as `where` and `skip`, return the same type of stream as the one the method was called on, unless

Felizmente, podemos converter um “single-subscription stream” em um “broadcast stream” de maneira muito simples. Vá até o arquivo **bloc.dart** e chame o método **broadcast** na construção dos objetos StreamController.

```
class Bloc with Validators{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController<String>.broadcast();
  final _passwordController = StreamController <String>.broadcast();

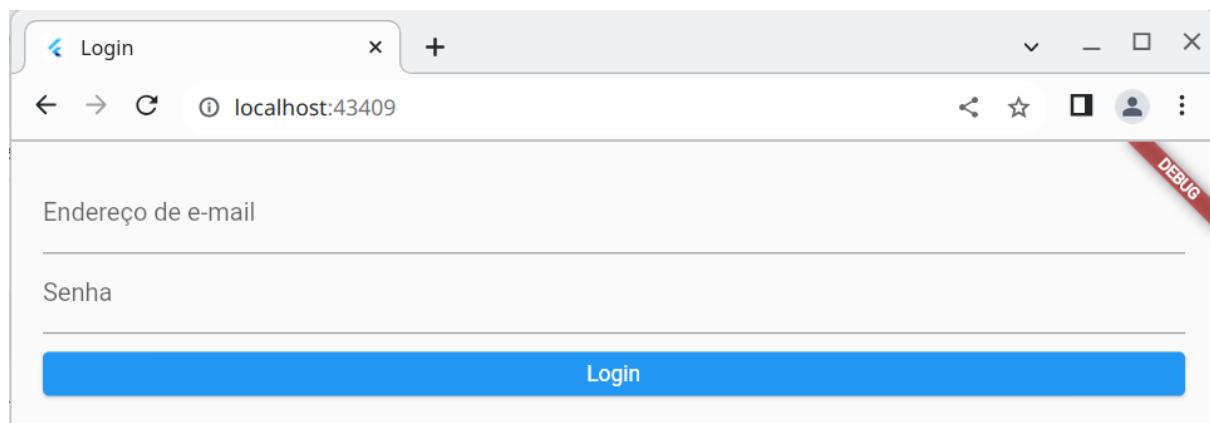
  Stream<String> get email => _emailController.stream.transform(validateEmail);

  Stream<String> get password => _passwordController.stream.transform(validatePassword);

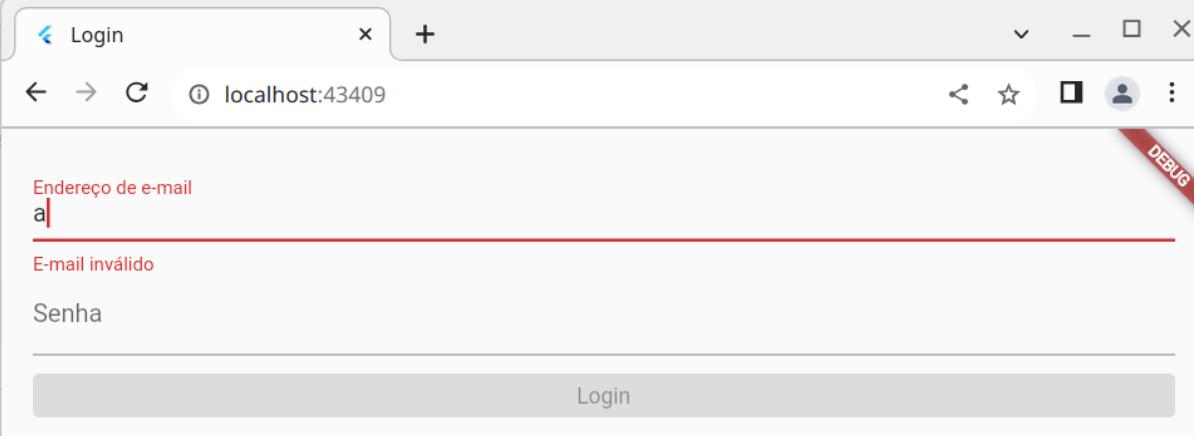
  Stream<bool> get emailPasswordAreOk => CombineLatestStream.combine2(email, password, (e, p) =>
  true);
  ...
}
```

Faça novo hot restart (SHIFT + R) e veja o resultado. O botão nasce habilitado. Porém, faça algumas edições nos campos, fazendo com que ele seja desabilitado. Por fim, digite valores válidos nos dois campos, tornando o botão habilitado novamente.

Botão nasce habilitado (isso acontece pois, quando a aplicação inicia, nenhum stream emitiu erro algum):

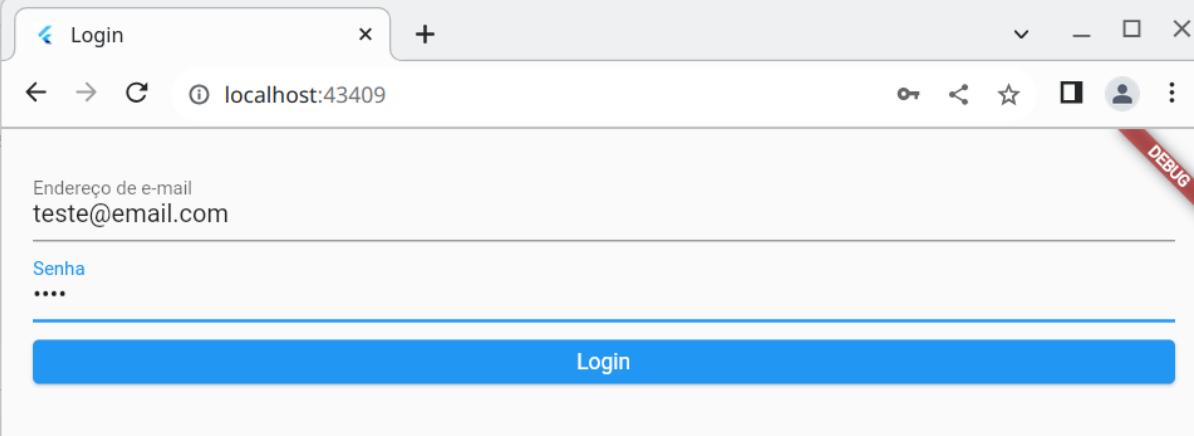


Se um dos campos estiver inválido, isso desabilita o botão.



A screenshot of a web browser window titled "Login". The address bar shows "localhost:43409". The page contains a form with two fields: "Endereço de e-mail" (Email address) and "Senha" (Password). The "Endereço de e-mail" field has the value "a" and is highlighted with a red border, with the error message "E-mail inválido" (Invalid email) displayed below it. The "Senha" field has the value "\*\*\*\*" and is also highlighted with a red border. A grey "Login" button is at the bottom of the form. A red ribbon in the top right corner of the page has the word "DEBUG" written on it.

Digitar valores válidos nos dois campos faz com que o botão seja habilitado novamente.



A screenshot of a web browser window titled "Login". The address bar shows "localhost:43409". The page contains a form with two fields: "Endereço de e-mail" (Email address) and "Senha" (Password). The "Endereço de e-mail" field has the value "teste@email.com" and the "Senha" field has the value "\*\*\*\*". Both fields are now highlighted with a blue border, indicating they are valid. The "Login" button is now a solid blue color, indicating it is enabled. A red ribbon in the top right corner of the page has the word "DEBUG" written on it.

Claro, queremos que o botão nasça desabilitado, afinal, não desejamos que o usuário seja capaz de enviar um form com campos vazios. Para isso, **vamos trocar o método hasError pelo método hasData**, pois

- quando a aplicação inicia, nenhum stream emitiu valor algum, o método hasData devolve false e não renderizamos o botão
- quando um campo está inválido, o método hasError devolve true e, portanto, o método hasData devolve false
- quando ambos os campos estão válidos, o método hasError devolve false e, portanto, o método hasData devolve true.

O funcionamento do método hasData é exatamente o que precisamos para decidir sobre a renderização do botão, incluindo o momento de nascimento da aplicação. Claro, precisamos "inverter" a função e o valor null no ternário.

```
...
Widget submitButton(Bloc bloc) {
  return StreamBuilder(
    stream: bloc.emailPasswordAreOk,
    builder: (context, AsyncSnapshot <bool> snapshot) {
      return ElevatedButton(
        onPressed: snapshot.hasData ? () {} : null,
        child: Text('Login'),
      );
    },
  );
}

...
```

Faça novo SHIFT + R e novos testes, especialmente observando a aplicação quando ela nasce, com algum campo com valor inválido e com os dois campos com valores válidos.

Acessando os valores de e-mail e senha

Quando o usuário clicar no botão e ele se encontrar num estado válido, desejamos acessar os valores digitados para enviar a um servidor. Mas como fazê-lo?

Utilizando o Bloc, vamos fazer a definição da função - seu processamento não tem muito a ver com interface gráfica, certo? - de maneira isolada.

Começamos escrevendo um novo método no arquivo **bloc.dart**.

```

import 'dart:async';
import 'validators.dart';
import 'package:rxdart/rxdart.dart';
class Bloc with Validators{
  //StreamController vem do pacote dart:async
  final _emailController = StreamController<String>.broadcast();
  final _passwordController = StreamController <String>.broadcast();

  Stream<String> get email => _emailController.stream.transform(validateEmail);

  Stream<String> get password => _passwordController.stream.transform(validatePassword);

  Stream<bool> get emailPasswordAreOk => CombineLatestStream.combine2(email, password, (e, p) =>
  true);

  Function(String) get changeEmail => _emailController.sink.add;

  Function(String) get changePassword => _passwordController.sink.add;

  void submitForm(){

  }

  void dispose(){
    _emailController.close();
    _passwordController.close();
  }
}

```

A seguir, no arquivo **login\_tela.dart**, podemos chamar a função quando o botão for clicado.

```

...
Widget submitButton(Bloc bloc) {
  return StreamBuilder(
    stream: bloc.emailPasswordAreOk,
    builder: (context, AsyncSnapshot <bool> snapshot) {
      return ElevatedButton(
        onPressed: snapshot.hasData ? bloc.submitForm : null,
        child: Text('Login'),
      );
    },
  );
}
...

```

Entretanto, os valores digitados pelo usuário são emitidos pelos streams e somente podem ser obtidos por meio dos objetos de tipo `AsyncSnapshot`. E eles se encontram justamente no arquivo (`login_tela.dart`) de onde acabamos de remover a definição da função. Para obtê-los, vamos utilizar um novo recurso da API `rxDart`. Veja a sua documentação.

<https://pub.dev/documentation/rxdart/latest/rx/BehaviorSubject-class.html>

rx documentation | learn | Search API

CLASSES

AbstractConnectableStr...

BehaviorSubject

BufferCountStreamTrans...

BufferStreamTransformer

BufferTestStreamTransfo...

CombineLatestStream

CompositeSubscription

ConcatEagerStream

ConcatStream

ConnectableStream

ConnectableStreamSub...

## BehaviorSubject<T> class

A special StreamController that captures the latest item that has been added to the controller, and emits that as the first item to any new listener.

This subject allows sending data, error and done events to the listener. The latest item that has been added to the subject will be sent to any new listeners of the subject. After that, any new events will be appropriately sent to the listeners. It is possible to provide a seed value that will be emitted if no items have been added to the subject.

BehaviorSubject is, by default, a broadcast (aka hot) controller, in order to fulfill the Rx Subject contract. This means the Subject's `stream` can be listened to multiple times.

Example

A primeira frase diz coisas muito importantes.

- Um `BehaviorSubject` nada mais é do que um tipo específico de `StreamController`
- Diferente do que acontece com os `StreamControllers` “comuns” que temos utilizado, eles têm a habilidade de capturar um item já emitido e entregar a um novo objeto que esteja interessado nele.

Observe, ainda na documentação, que um `BehaviorSubject` é um controller “broadcast”. Fundamental para que ele possa funcionar da forma como promete.

Lembra-se que havíamos convertido nosso “single-subscription stream” em um “broadcast stream”. Dada a natureza do `BehaviorSubject` descrita na sua documentação, isso não será mais necessário.

O pequeno exemplo que a documentação mostra é um tanto intuitivo. Observe.

rxdart package > documentation > rx > BehaviorSubject<T> class

**CLASSES**

- AbstractConnectableStr...
- BehaviorSubject
- BufferCountStreamTrans...
- BufferStreamTransformer
- BufferTestStreamTransfo...
- CombineLatestStream
- CompositeSubscription
- ConcatEagerStream
- ConcatStream
- ConnectableStream
- ConnectableStreamSub...
- DebounceStreamTransf...
- DefaultIfEmptyStreamTr...
- DeferStream
- DelayStreamTransformer
- DelayWhenStreamTrans...
- DematerializeStreamTra...
- DistinctUniqueStreamTr...
- DoStreamTransformer

## BehaviorSubject<T> class

A special StreamController that captures the latest item that has been added to the controller, and emits that as the first item to any new listener.

This subject allows sending data, error and done events to the listener. The latest item that has been added to the subject will be sent to any new listeners of the subject. After that, any new events will be appropriately sent to the listeners. It is possible to provide a seed value that will be emitted if no items have been added to the subject.

BehaviorSubject is, by default, a broadcast (aka hot) controller, in order to fulfil the Rx Subject contract. This means the Subject's `stream` can be listened to multiple times.

**Example**

```
final subject = BehaviorSubject<int>();
subject.add(1);
subject.add(2);
subject.add(3);

subject.stream.listen(print); // prints 3
subject.stream.listen(print); // prints 3
subject.stream.listen(print); // prints 3
```

A alteração é, portanto, um tanto simples. No arquivo **bloc.dart**, troque o tipo dos controllers: eles deixam de ser do tipo StreamController e passam a ser do tipo BehaviorSubject. Lembre-se de que não é mais necessário chamar o método broadcast.

```
import 'dart:async';
import 'validators.dart';
import 'package:rxdart/rxdart.dart';
class Bloc with Validators{
  //StreamController vem do pacote dart:async
  //BehaviorSubject vem do pacote rxdart
  final _emailController = BehaviorSubject<String>();
  final _passwordController = BehaviorSubject <String>();
  ...
}
```

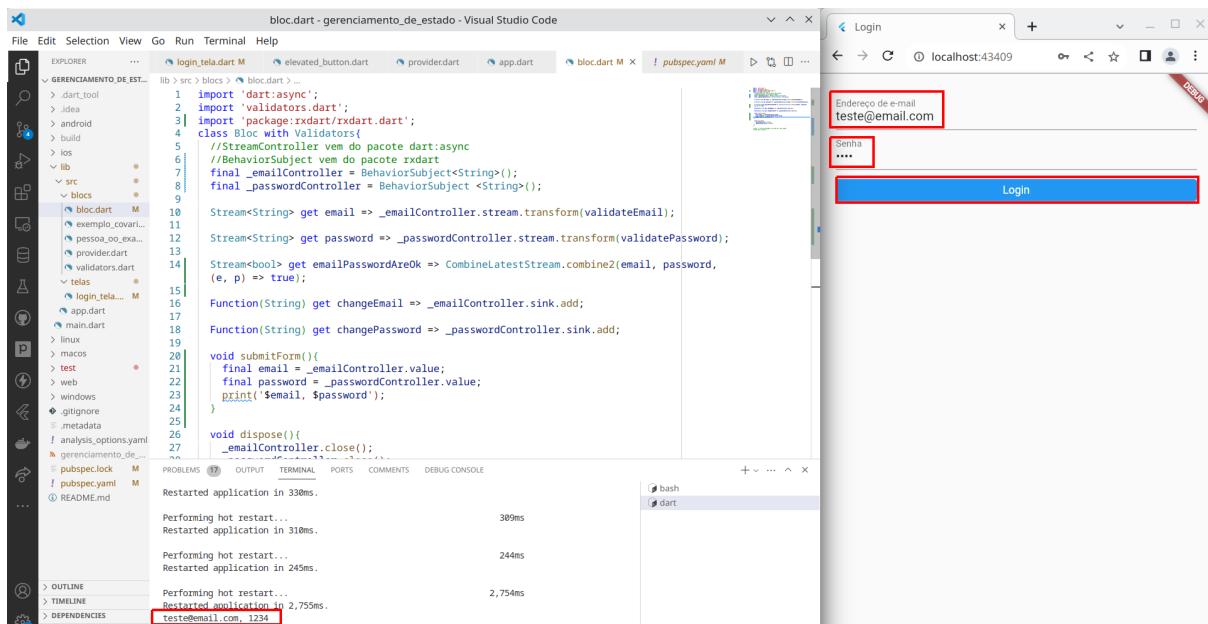
Ainda no arquivo **bloc.dart**, agora podemos implementar o método de envio fictício do form, graças à funcionalidade provida pelo BehaviorSubject. Para encontrar o último valor emitido por ele, basta usar a sua propriedade `value`.

```

class Bloc with Validators{
  ...
  void submitForm() {
    final email = _emailController.value;
    final password = _passwordController.value;
    print('$_email, $password');
  }
  ...
}

```

Faça novo hot restart (SHIFT + R), digite valores válidos nos dois campos, clique no botão e veja o resultado no terminal.

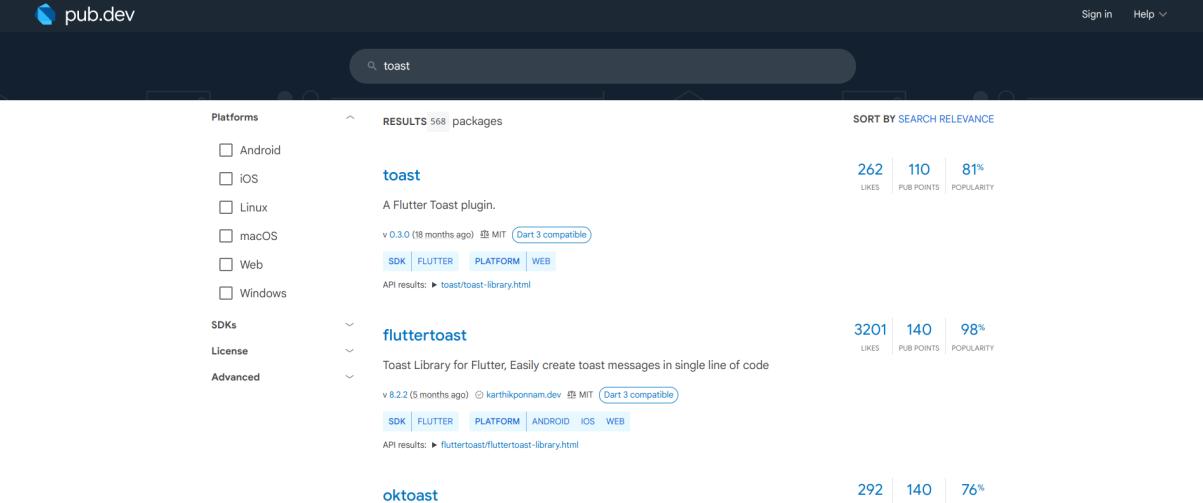


Talvez queiramos exibir os valores num Toast.

**Nota.** Toast vem de “torrada”. É uma mensagem temporária que “sobe”, aparecendo na tela, e depois de pouco tempo “desce”. Como se fosse uma torrada pulando de uma torradeira.

A fim de evitarmos reinventar a roda, façamos uma busca no pub.dev tentando encontrar algum pacote capaz de produzir um toast.

Há inúmeras opções. Veja os resultados obtidos no momento em que esse documento foi escrito.



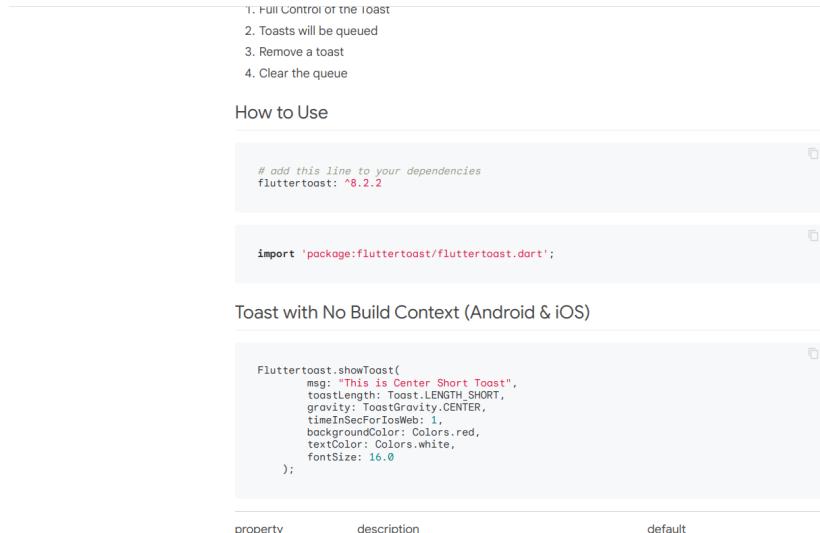
The screenshot shows the pub.dev search interface with the query 'toast'. The results page displays three packages: 'toast', 'flutertoast', and 'oktoast'. The 'toast' package is the top result, with a description of 'A Flutter Toast plugin.', version v0.3.0 (18 months ago), and Dart 3 compatible. It has 262 likes, 110 pub points, and 81% popularity. The 'flutertoast' package is the second result, with a description of 'Toast Library for Flutter, Easily create toast messages in single line of code.', version v8.2.2 (5 months ago), and Dart 3 compatible. It has 3201 likes, 140 pub points, and 98% popularity. The 'oktoast' package is the third result, with a description of 'A simple toast library for Flutter.', version 2.9.2 (10 months ago), and Dart 3 compatible. It has 292 likes, 140 pub points, and 76% popularity.

O pacote “flutertoast” parece ser o mais popular no momento. Vejamos se a sua página oficial oferece um exemplo de uso.

<https://pub.dev/packages/flutertoast>

**Cuidado.** O teste a seguir foi realizado num emulador. Com o código de exemplo, esse Toast somente funciona em emuladores ou dispositivos reais. Não funciona na Web. Vale olhar os exemplos na documentação e também buscar por outros pacotes para a web.

Sim! Há um exemplo um tanto intuitivo. Observe que a documentação também mostra como fazer a instalação do pacote e como fazer o import no código.



The screenshot shows the documentation for the 'flutertoast' package. It includes a list of steps: 1. Full Control or the toast, 2. Toasts will be queued, 3. Remove a toast, 4. Clear the queue. Below this is a 'How to Use' section with two code snippets. The first snippet shows how to add the package to dependencies in the pubspec.yaml file: `# add this line to your dependencies` `flutertoast: ^8.2.2`. The second snippet shows how to import the package in a Dart file: `import 'package:flutertoast/flutertoast.dart';`. Below these snippets is a 'Toast with No Build Context (Android & iOS)' section, which contains a code example for showing a toast message: `Fluttertoast.showToast(` `msg: "This is Center Short Toast",` `toastLength: Toast.LENGTH_SHORT,` `gravity: ToastGravity.CENTER,` `timeInSecForIosWeb: 1,` `backgroundColor: Colors.red,` `textColor: Colors.white,` `fontSize: 16.0` `);`. The table below the code lists the properties, descriptions, and defaults for each column.

| property | description | default |
|----------|-------------|---------|
|          |             |         |

Num terminal do VS Code, use

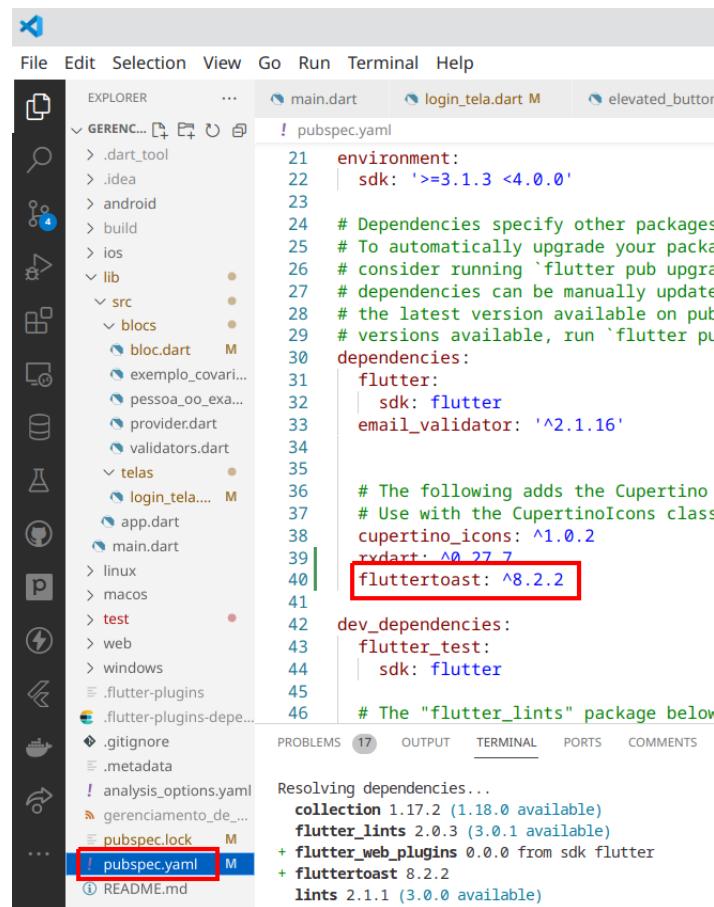
```
flutter pub add fluttertoast
```

Veja o resultado esperado.



```
● gerenciamento_de_estado $ flutter pub add fluttertoast
Resolving dependencies...
  collection 1.17.2 (1.18.0 available)
  flutter_lints 2.0.3 (3.0.1 available)
+ flutter_web_plugins 0.0.0 from sdk flutter
+ fluttertoast 8.2.2
  lint 2.1.1 (3.0.0 available)
  material_color_utilities 0.5.0 (0.8.0 available)
  meta 1.9.1 (1.11.0 available)
  stack_trace 1.11.0 (1.11.1 available)
  stream_channel 2.1.1 (2.1.2 available)
  test_api 0.6.0 (0.6.1 available)
  web 0.1.4-beta (0.3.0 available)
Changed 2 dependencies!
○ gerenciamento_de_estado $
```

Investigue, também, seu arquivo **pubspec.yaml** e certifique-se de que a dependência apareceu por lá (Por razões didáticas apenas. Não há motivo algum para ela não estar. É o comportamento padrão da ferramenta pub.).



```
! pubspec.yaml
21 environment:
22   | sdk: '>=3.1.3 <4.0.0'
23
24 # Dependencies specify other packages
25 # To automatically upgrade your packa
26 # consider running `flutter pub upgra
27 # dependencies can be manually update
28 # the latest version available on pub
29 # versions available, run `flutter pu
30 dependencies:
31   flutter:
32     | sdk: flutter
33     email_validator: '^2.1.16'
34
35 # The following adds the Cupertino
36 # Use with the CupertinoIcons class
37 cupertino_icons: ^1.0.2
38 rxdart: ^0.27.7
39 fluttertoast: ^8.2.2
40
41 dev_dependencies:
42   flutter_test:
43     | sdk: flutter
44
45 # The "flutter_lints" package below
46
```

Resolving dependencies...

```
collection 1.17.2 (1.18.0 available)
flutter_lints 2.0.3 (3.0.1 available)
+ flutter_web_plugins 0.0.0 from sdk flutter
+ fluttertoast 8.2.2
  lint 2.1.1 (3.0.0 available)
```

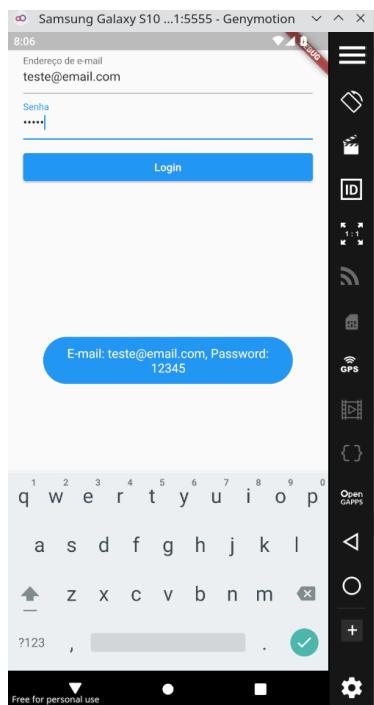
No arquivo **bloc.dart**, importamos o pacote e utilizamos o exemplo da documentação. Observe que vamos utilizar a classe **Colors** do Flutter. Por isso, importamos o Flutter também.

```
import 'dart:async';
import 'validators.dart';
import 'package:rxdart/rxdart.dart';
import 'package:fluttertoast/fluttertoast.dart';
import 'package:flutter/material.dart';
class Bloc with Validators{
  ...
  void submitForm(){
    final email = _emailController.value;
    final password = _passwordController.value;
    //print('$_email, $password');
    Fluttertoast.showToast(
      msg: 'E-mail: $_email, Password: $_password',
      toastLength: Toast.LENGTH_SHORT,
      gravity: ToastGravity.CENTER,
      timeInSecForIosWeb: 1,
      backgroundColor: Colors.blue,
      textColor: Colors.white,
      fontSize: 16.0
    );
  }
  void dispose(){
    _emailController.close();
    _passwordController.close();
  }
}
```

Os nomes das propriedades do **Toast** são auto-explicativos. Inspecione a documentação e troque os valores das propriedades!

Faça novo teste.

- SHIFT + R no terminal
- Digite valores válidos nos dois campos
- Clique no botão
- Veja o Toast



### ***Referências***

**Dart programming language | Dart.** Google, 2023. Disponível em <<https://dart.dev/>>. Acesso em novembro de 2023.

**Flutter - Build apps for any screen.** Google, 2023. Disponível em <<https://flutter.dev/>>. Acesso em novembro de 2023.