

# 1 Introdução

Flutter é um framework criado pelo Google que permite a criação de aplicações “**bonitas**” para múltiplas plataformas. Veja algumas características.

Veja a sua página oficial.

<https://flutter.dev/>

A partir de um único “codebase”, podemos gerar aplicações para múltiplas plataformas como

- Linux
- Windows
- MacOS
- Android
- iOS
- Web

Compilação **nativa** para arquiteturas **ARM** e **Intel**. Também compila para **Javascript**.

Interfaces gráficas construídas utilizando-se **Widgets**. Um Widget é um componente visual como um botão, um campo textual, um gerenciador de layout etc.

Cada **Widget** é implementado como uma classe **Dart**.

Flutter possui um vasto catálogo de Widgets. Ele oferece pode ser encontrado a seguir

<https://docs.flutter.dev/ui/widgets>

É uma boa ideia manter esse link na sua barra de favoritos enquanto desenvolve.

Também podemos criar nossos próprios Widgets.

Neste material, vamos desenvolver uma aplicação que exibe fotos obtidas por meio de requisições HTTP. Ela nos permitirá aprender o uso básico de Widgets Flutter bem como alguns recursos interessantes, tais como o processamento assíncrono funciona neste ambiente.

## 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\**

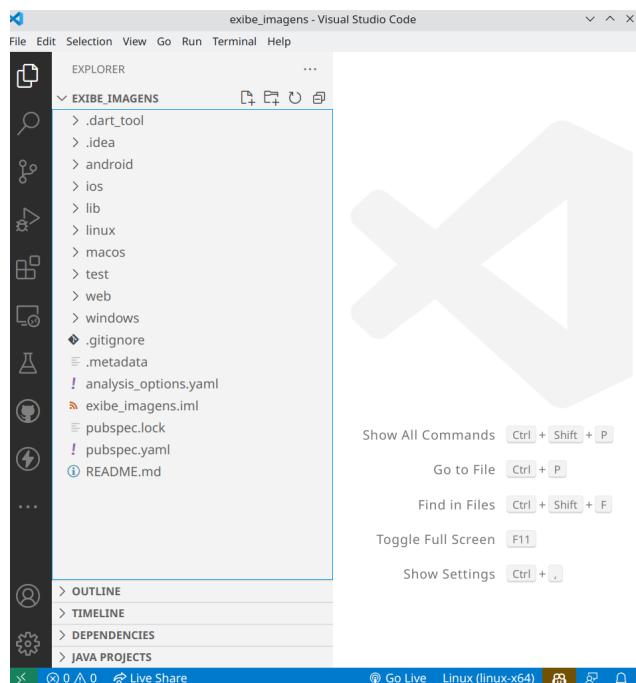
A seguir, abra um terminal e vincule-o ao diretório que acabou de criar com

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

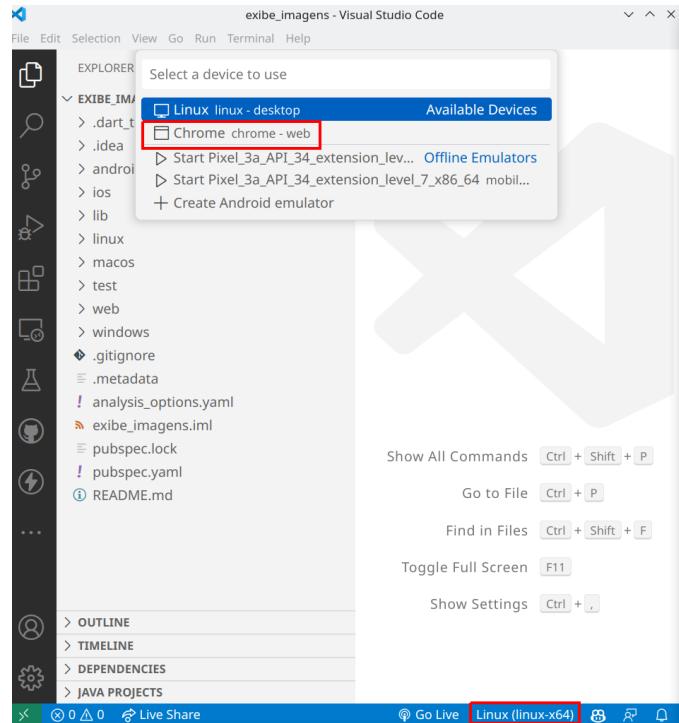
Crie o projeto Flutter com

**flutter create exibe\_imagens**

Abra o VS Code e clique em **File > Open Folder**. Navegue até a pasta recém-criada. Veja o resultado esperado.



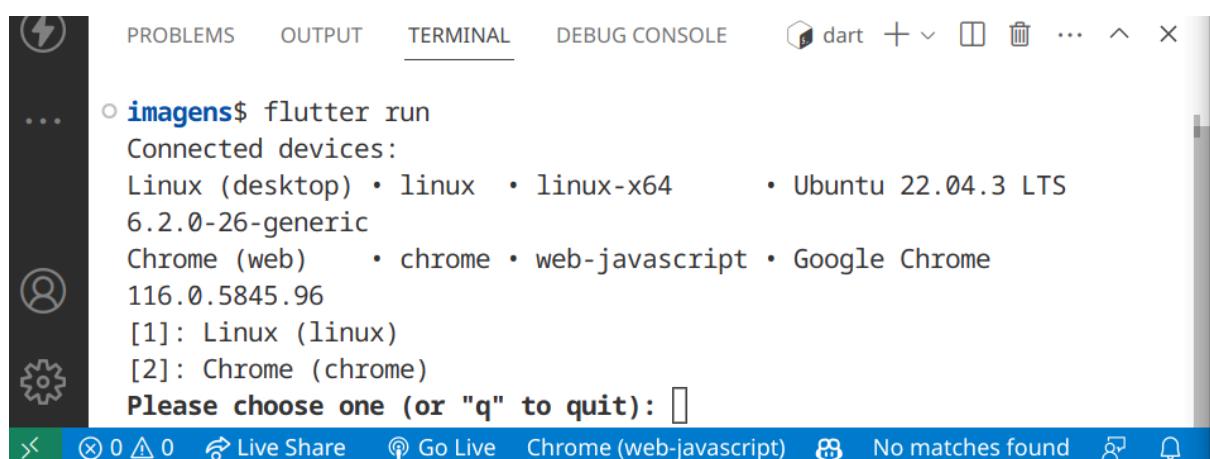
Observe que no canto inferior direito há a possibilidade de escolher a plataforma em que a aplicação será colocada em execução. No exemplo a seguir, o padrão selecionado é Linux. Dependendo de seu ambiente, o padrão pode ser diferente. Clique sobre esta opção e escolha o Chrome para executarmos a aplicação na Web.



No VS Code, clique em **Terminal >> New Terminal**. Use

**flutter run**

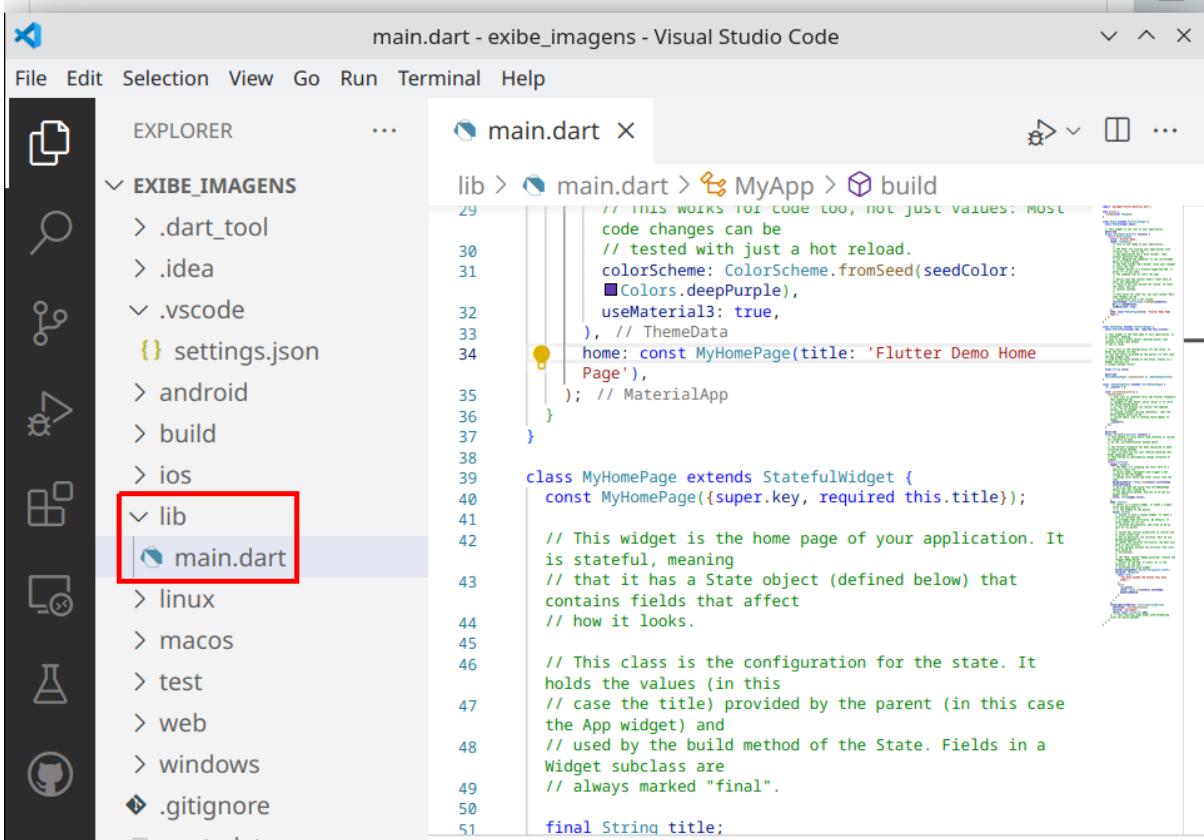
para executar a aplicação. É possível que você tenha de selecionar a plataforma alvo. Digite o número associado ao Google Chrome.



A aplicação resultante apenas mostra um botão que, quando clicado, incrementa um contador exibido na parte central da tela.



Observe que temos uma pasta chamada **lib** e que, dentro dela, há um arquivo chamado **main.dart**. O código Dart da aplicação exemplo se encontra ali. O ponto de partida da aplicação se encontra neste arquivo. Seu nome **deve ser main.dart**.



```

main.dart - exibe_imagens - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
  EXIBE_IMAGENS
    > .dart_tool
    > .idea
    > .vscode
      > settings.json
    > android
    > build
    > ios
    > lib
      > main.dart
    > linux
    > macos
    > test
    > web
    > windows
    > .gitignore
  = metadata
main.dart x
lib > main.dart > MyApp > build
29   // THIS WORKS FOR code too, not just values: MOST
30   // code changes can be
31   // tested with just a hot reload.
32   colorScheme: ColorScheme.fromSeed(seedColor:
33     Colors.deepPurple),
34   useMaterial3: true,
35   ), // ThemeData
36   home: const MyHomePage(title: 'Flutter Demo Home
37   Page'),
38   ); // MaterialApp
39
40 class MyHomePage extends StatefulWidget {
41   const MyHomePage({super.key, required this.title});
42
43   // This widget is the home page of your application. It
44   // is stateful, meaning
45   // that it has a State object (defined below) that
46   // contains fields that affect
47   // how it looks.
48
49   // This class is the configuration for the state. It
50   // holds the values (in this
51   // case the title) provided by the parent (in this case
52   // the App widget) and
53   // used by the build method of the State. Fields in a
54   // Widget subclass are
55   // always marked "final".
56
57   final String title;

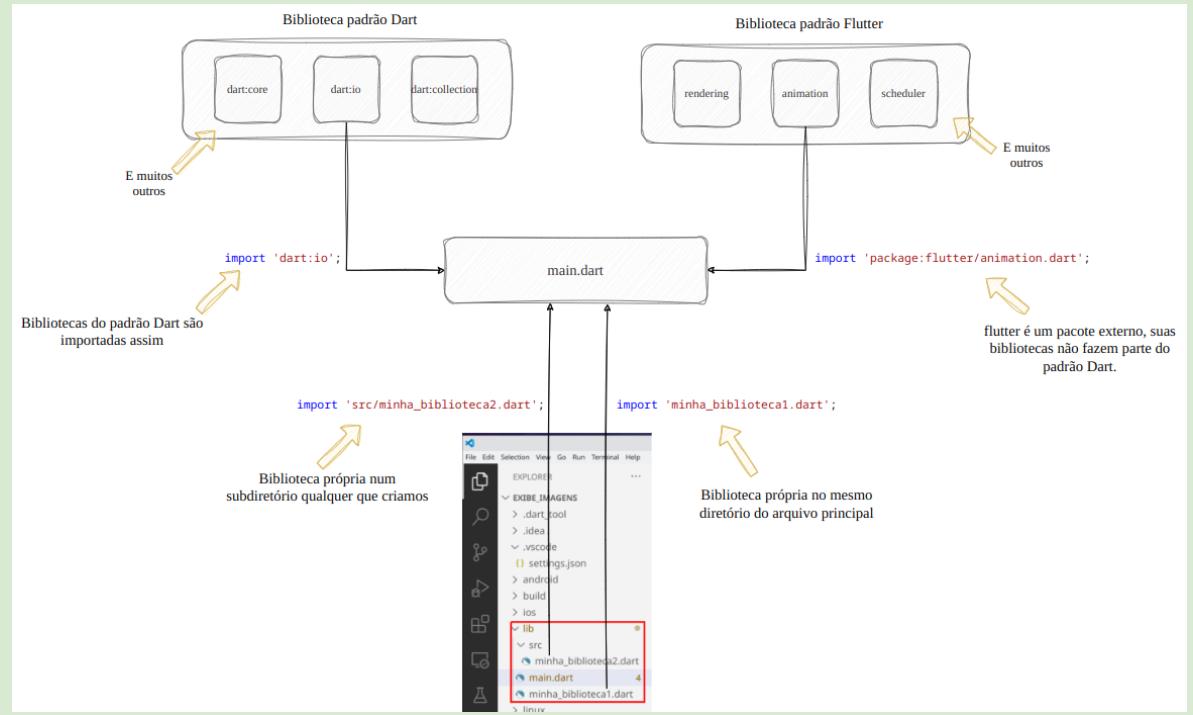
```

Apague todo o conteúdo do arquivo **main.dart** para começarmos a trabalhar em nossa aplicação do zero.

**(Exibindo um componente textual)** Vamos começar apenas exibindo um componente textual. Para isso, precisamos

- importar o pacote que contém o Widget desejado
- escrever uma função main, que é o ponto em que a aplicação começa
- criar o Widget capaz de exibir conteúdo textual
- exibir o Widget na tela

**Nota.** Há três formas de se importar conteúdo num arquivo dart. Para entender cada um, observe a seguinte figura.



Assim, podemos importar bibliotecas do ambiente padrão dart, bibliotecas de pacotes externos - como o flutter - e bibliotecas próprias que criamos em qualquer diretório de nossa aplicação que seja subdiretório de lib.

Para utilizar componentes visuais do Flutter, vamos importar o pacote material.

**Nota.** O pacote material traz a implementação da especificação conhecida como **Material Design**, do Google. Trata-se de um sistema de design com instruções para o desenvolvimento envolvendo UX, a implementação de componentes visuais, combinação de cores, feedback visual mediante interação do usuário etc. Para saber mais, visite

<https://m3.material.io/>

Veja o import. Estamos no arquivo **lib/main.dart**.

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

A seguir, escrevemos a função main, como de costume.

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

void main() {

}
```

No momento, nosso objetivo é apenas exibir um componente textual. Vamos construir um Widget do tipo **MaterialApp**, para começar.

**Nota.** O Widget **MaterialApp** se encarrega de realizar configurações básicas, como a possibilidade de **navegação entre diferentes telas**. Seu uso é bastante comum.

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

void main() {
    var app = MaterialApp();
}
```

Observe que há uma sugestão de aplicar o modificador **const** à chamada do construtor. Isso acontece pois esse é um construtor que foi declarado como const. De maneira simplificada, isso quer dizer que ele constrói um objeto imutável. Quando aplicamos const à chamada de um construtor, o compilador nos entrega código otimizado.

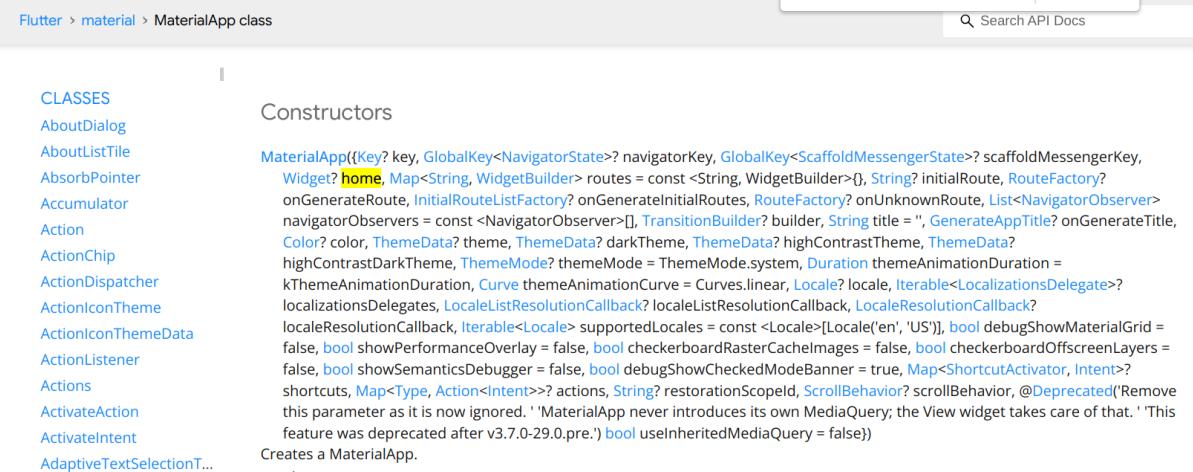
```
import 'package:flutter/material.dart';

void main() {
    var app = const MaterialApp();
}
```

Visite a documentação do Widget **MaterialApp** a seguir.

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

Observe que seu construtor possui um parâmetro nomeado **home**.



CLASSES

AboutDialog

AboutListTile

AbsorbPointer

Accumulator

Action

ActionChip

ActionDispatcher

ActionIconTheme

ActionIconThemeData

ActionListener

Actions

ActivateAction

ActivateIntent

AdaptiveTextSelectionT...

...

Constructors

`MaterialApp({Key? key, GlobalKey<NavigatorState>? navigatorKey, GlobalKey<ScaffoldMessengerState>? scaffoldMessengerKey, Widget? home, Map<String, WidgetBuilder> routes = const <String, WidgetBuilder>{}, String? initialRoute, RouteFactory? onGenerateRoute, InitialRouteListFactory? onGenerateInitialRoutes, RouteFactory? onUnknownRoute, List<NavigatorObserver> navigatorObservers = const <NavigatorObserver>[], TransitionBuilder? builder, String title = '', GenerateAppTitle? onGenerateTitle, Color? color, ThemeData? theme, ThemeData? darkTheme, ThemeData? highContrastTheme, ThemeData? highContrastDarkTheme, ThemeMode? themeMode = ThemeMode.system, Duration themeAnimationDuration = kThemeAnimationDuration, Curve themeAnimationCurve = Curves.linear, Locale? locale, Iterable<LocalizationsDelegate>? localizationsDelegates, LocaleListResolutionCallback? localeListResolutionCallback, LocaleResolutionCallback? localeResolutionCallback, Iterable<Locale> supportedLocales = const <Locale>[Locale('en', 'US')], bool debugShowMaterialGrid = false, bool showPerformanceOverlay = false, bool checkerboardRasterCachelImages = false, bool checkerboardOffscreenLayers = false, bool showSemanticsDebugger = false, bool debugShowCheckedModeBanner = true, Map<ShortcutActivator, Intent>? shortcuts, Map<Type, Action<Intent>>? actions, String? restorationScopeld, ScrollBehavior? scrollBehavior, @Deprecated('Remove this parameter as it is now ignored. ' 'MaterialApp never introduces its own MediaQuery; the View widget takes care of that. ' 'This feature was deprecated after v3.7.0-29.0.pre.') bool useInheritedMediaQuery = false})`

Creates a MaterialApp.

`const`

Desejamos associar um novo Widget a esse parâmetro. Ele será exibido na região principal da tela. Neste momento, vamos apenas utilizar um Widget textual.

**Nota.** Também poderíamos aplicar **const** à chamada do construtor Text. Isso seria redundante pelo fato de ele estar sendo utilizado como parâmetro de MaterialApp e por já termos aplicado const a esse construtor.

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

void main() {
  var app = const MaterialApp(
    home: Text('Hello, Dart')
  );
}
```

**Nota.** Observe a forma como ajustamos o código. Salve o arquivo. É provável que ele seja reorganizado da seguinte forma.

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

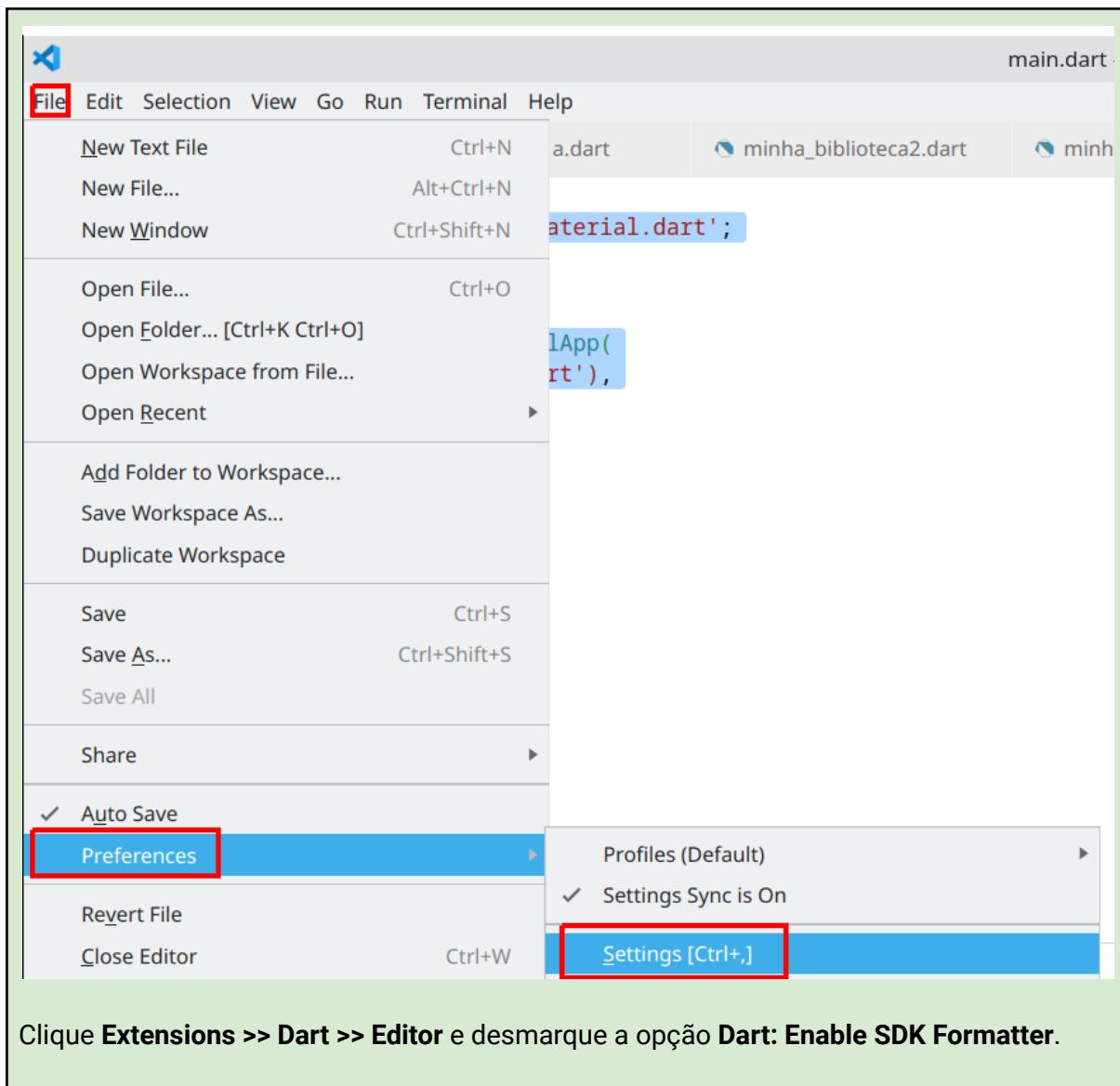
void main() {
  var app = const MaterialApp(home: Text('Hello, Dart'));
}
```

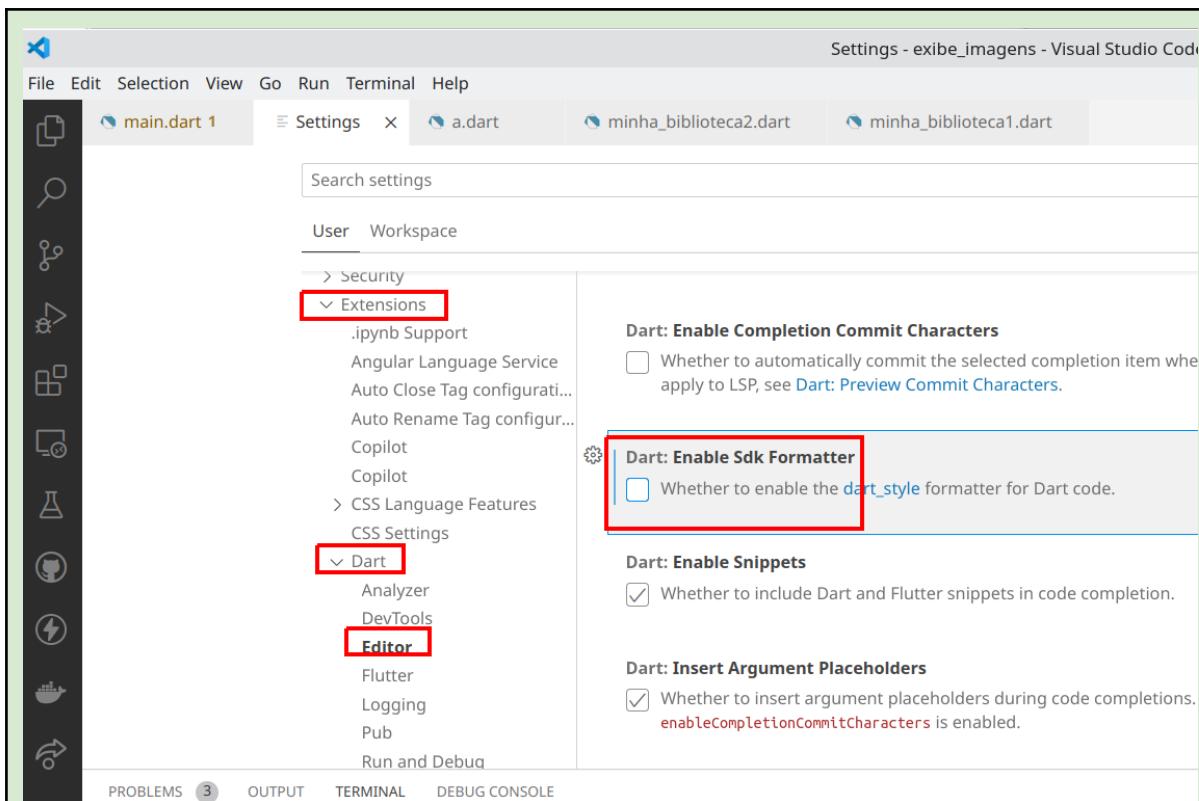
Se desejar evitar esse comportamento, você pode adicionar uma vírgula no final da definição do último componente. Observe.

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

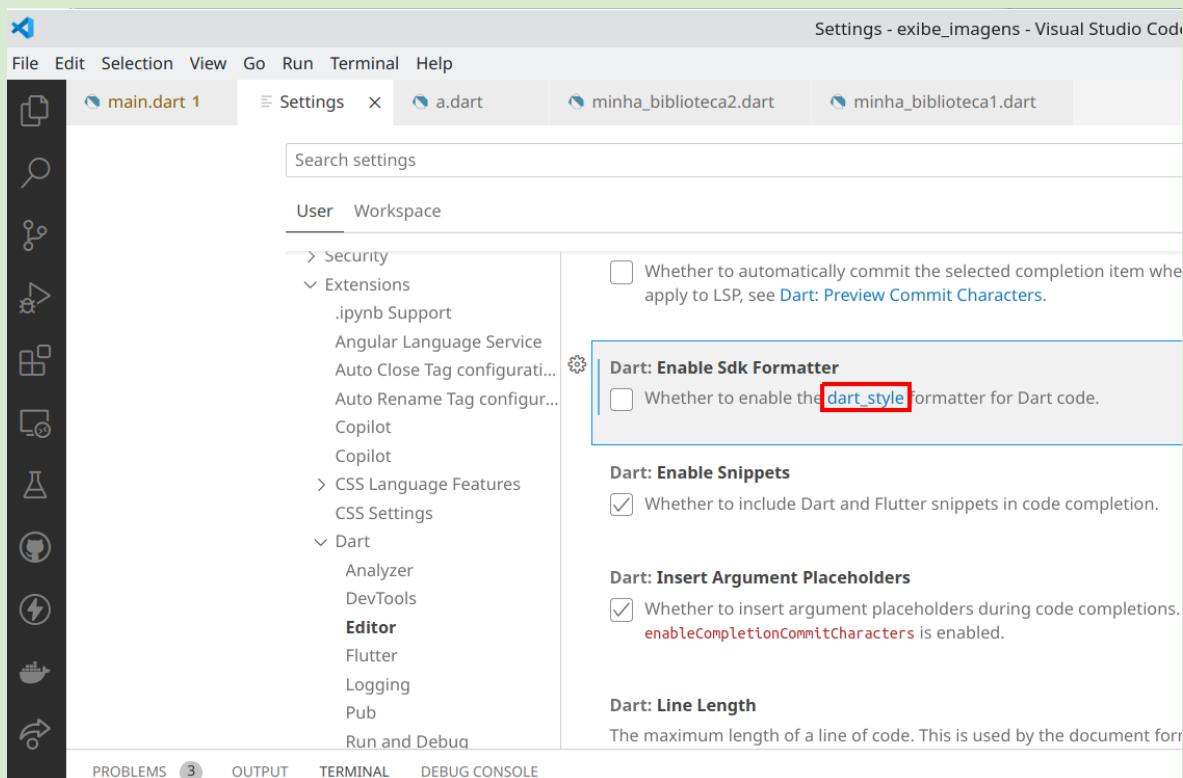
void main() {
  var app = const MaterialApp(
    home: Text('Hello, Dart'),
  );
}
```

Salve o arquivo novamente e repare que o código permanece da forma como você deixou. Se preferir, também é possível desabilitar esse funcionamento utilizando uma opção da extensão Dart, embora ela também envolva outros aspectos. No VS Code, clique **File >> Preferences Settings**.

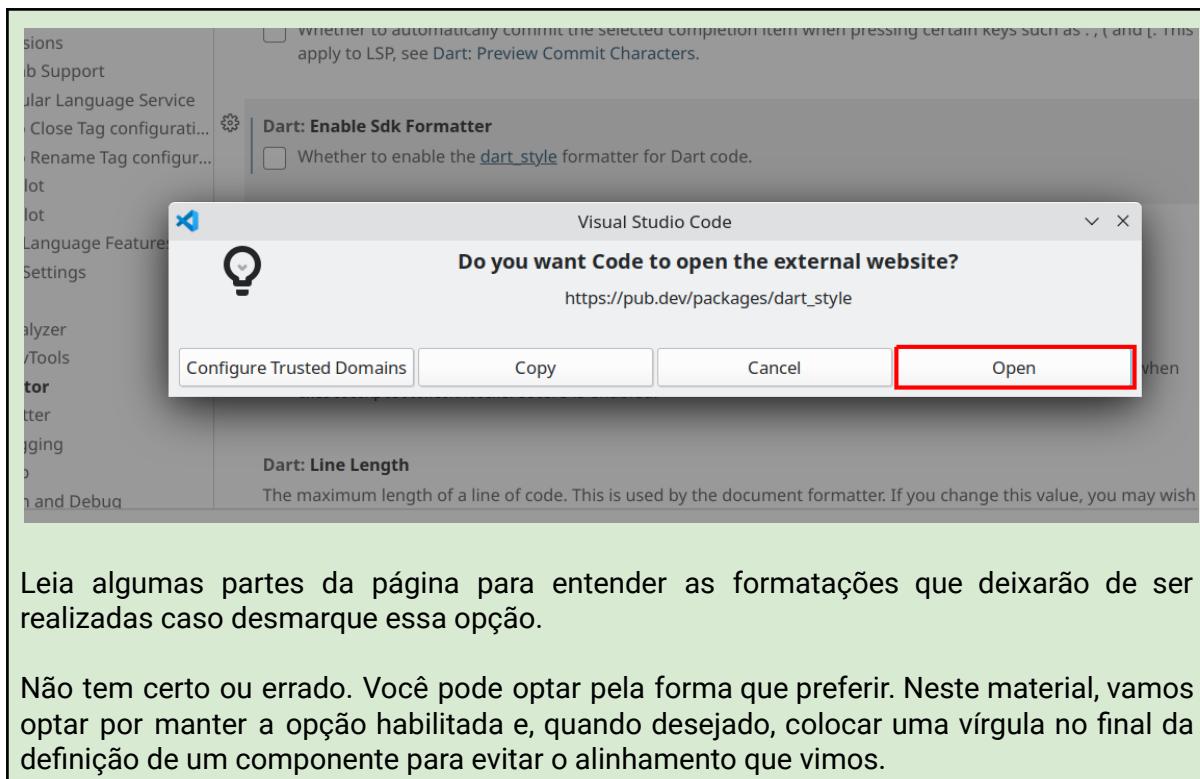




Observe, entretanto, que desmarcar esta opção, envolve diversos outros tipos de ajustes feitos automaticamente e que estão de acordo com as recomendações oficiais Dart. Ainda nesta tela, clique **dart\_style**.



Se necessário, clique em **Open**.



Leia algumas partes da página para entender as formatações que deixarão de ser realizadas caso desmarque essa opção.

Não tem certo ou errado. Você pode optar pela forma que preferir. Neste material, vamos optar por manter a opção habilitada e, quando desejado, colocar uma vírgula no final da definição de um componente para evitar o alinhamento que vimos.

Por enquanto, o código está assim.

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

void main() {
  var app = const MaterialApp(
    home: Text('Hello, Dart'),
  );
}
```

A simples definição da função main não é suficiente para que possamos colocar a aplicação em execução. Aplicações Flutter devem ser colocadas em execução de um jeito especial: **utilizando uma função chamada runApp**. Na função main, vamos chamar a função runApp entregando a ela o Widget "raiz" da aplicação.

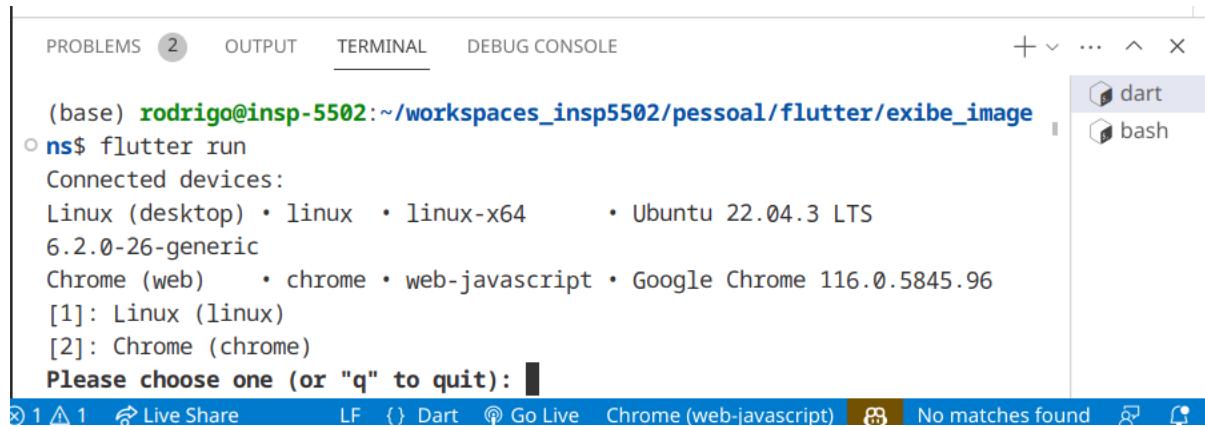
```
import 'package:flutter/material.dart';

void main() {
  var app = const MaterialApp(
    home: Text('Hello, Dart'),
  );
  runApp(app);
}
```

No VS Code, clique em **Terminal >> New Terminal** para abrir um terminal interno, caso ainda não possua um. Use

**flutter run**

para executar a aplicação. No terminal, escolha a plataforma para execução digitando seu número associado. No exemplo a seguir, digitamos **2** para executar a aplicação no Google Chrome.



The screenshot shows the VS Code interface with the 'TERMINAL' tab selected. The terminal window displays the following text:

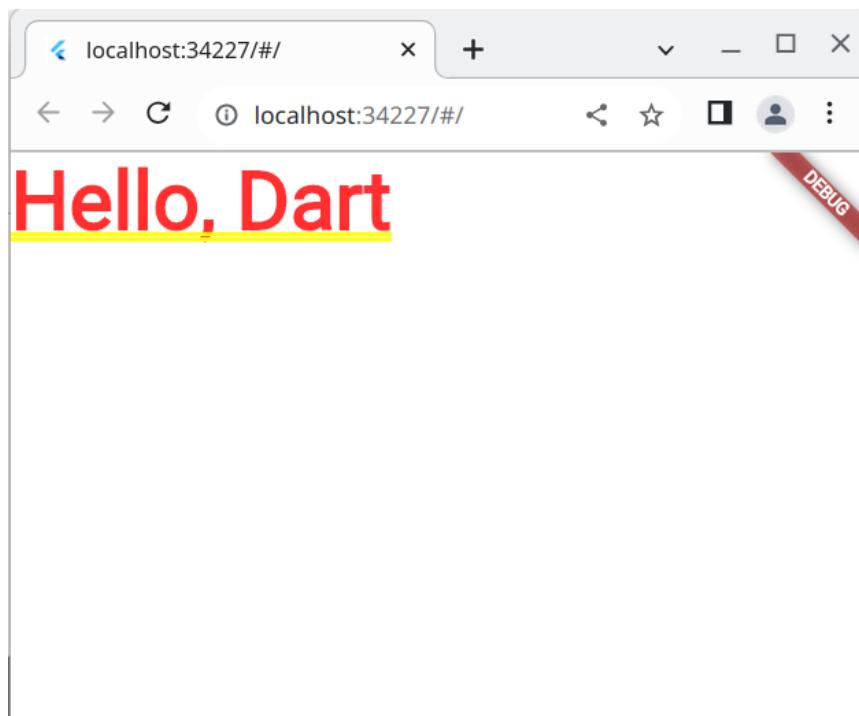
```
(base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/flutter/exibe_image
$ flutter run
Connected devices:
Linux (desktop) • linux • linux-x64      • Ubuntu 22.04.3 LTS
6.2.0-26-generic
Chrome (web)     • chrome • web-javascript • Google Chrome 116.0.5845.96
[1]: Linux (linux)
[2]: Chrome (chrome)

Please choose one (or "q" to quit):
```

The right sidebar shows a dropdown menu with 'dart' and 'bash' options, with 'dart' currently selected. The bottom of the terminal window shows various status icons and text.

**Nota.** É natural que este processo demore algo entre 30 segundos a 1 minuto. Ou talvez um pouco mais.

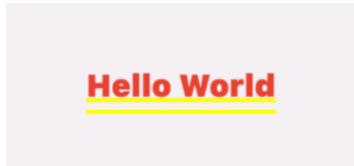
A esperança é que uma aba do navegador seja aberta e que a aplicação seja exibida. Observe.



Percebeu como o estilo do texto, incluindo as cores, não é lá dos melhores? Isso é de propósito. Veja o que a documentação da classe `MaterialApp` fala sobre isso.

Why is my app's text red with yellow underlines?

`Text` widgets that lack a `Material` ancestor will be rendered with an ugly red/yellow text style.



The typical fix is to give the widget a `Scaffold` ancestor. The `Scaffold` creates a `Material` widget that defines its default text style.

```
const MaterialApp(
  title: 'Material App',
  home: Scaffold(
    body: Center(
      child: Text('Hello World'),
    ),
  ),
)
```

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

A ideia é lembrar o desenvolvedor de fazer uso adequado de componentes que apliquem temas do Material Design.

**(Hot reload, hot restart e full restart)** Conforme testamos nossa aplicação, estamos interessados em visualizar as modificações. Para isso, podemos usar as seguintes opções:

- **hot reload**: atualiza o trecho de código alterado direto na Dart VM e reconstrói a árvore de Widgets. **Preserva o estado da aplicação. Não a atualiza executando novamente o método main**. No terminal do VS Code em que você digitou `flutter run`, aperte a tecla **r** para utilizar.

- **hot restart**: atualiza o trecho de código alterado direto na Dart VM e **reinicia a aplicação, o que faz com o que o estado seja perdido**. No terminal do VS Code em que você digitou `flutter run`, aperte **SHIFT + r** para utilizar.

- **full restart**: reinicia a aplicação completamente. Demora mais pois o código também é recompilado. Não há atalho no teclado. Basta encerrar o processo de execução da aplicação no terminal (**CTRL + C**) e executar novamente.

Faça um teste alterando o texto da aplicação e fazendo um hot reload logo a seguir. Testaremos outras situações ao longo do desenvolvimento.

```

import 'package:flutter/material.dart';

void main() {
  var app = const MaterialApp(
    home: Text('Helloooo, Dart'),
  );
  runApp(app);
}

```

Lembre-se de apertar **r** no terminal em que executou o comando **flutter run**.

**Nota.** Há casos em que o **hot reload e/ou o hot restart podem não funcionar**.Leia mais no link a seguir. Veja alguns exemplos da documentação.

## An app is killed

Hot reload can break when the app is killed. For example, if the app was in the background for too long.

## Compilation errors

When a code change introduces a compilation error, hot reload generates an error message similar to:

```

Hot reload was rejected:
'/path/to/project/lib/main.dart': warning: line 16 pos 38: unbalanced '{' open:
  Widget build(BuildContext context) {
  ^
'/path/to/project/lib/main.dart': error: line 33 pos 5: unbalanced ')'
  );
  ^

```

In this situation, simply correct the errors on the specified lines of Dart code to keep using hot reload.

## Enumerated types

Hot reload doesn't work when enumerated types are changed to regular classes or regular classes are changed to enumerated types.

For example:

Before the change:

```
enum Color {
    red,
    green,
    blue,
}
```

After the change:

```
class Color {
    Color(this.i, this.j);
    final int i;
    final int j;
}
```

## Generic types

Hot reload won't work when generic type declarations are modified. For example, the following won't work:

Before the change:

```
class A<T> {
    T? i;
}
```

After the change:

```
class A<T, V> {
    T? i;
    V? v;
}
```

## Native code

If you've changed native code (such as Kotlin, Java, Swift, or Objective-C), you must perform a full restart (stop and restart the app) to see the changes take effect.

Estes exemplos foram tirados da documentação no momento em que este documento está sendo escrito. Para mais detalhes, visite a página a seguir.

<https://docs.flutter.dev/tools/hot-reload>

**(Usando um Widget Scaffold)** A fim de organizar a estrutura da aplicação, vamos fazer uso de um Widget chamado Scaffold. Veja a sua documentação a seguir.

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

**Nota.** "Scaffold" significa algo como **"esqueleto"** ou **"andaime"**.

O código dos exemplos da documentação pode envolver conceitos ainda não vistos neste momento. Não se preocupe com ele. Mas observe que um Scaffold permite especificarmos barras inferiores (**BottomAppBar**), botões flutuantes (**FloatingActionButton**) e assim por diante. Role a página até encontrar a parte que ilustra o construtor. A seguir, ilustramos dois exemplos de Widgets que podem ser entregues ao construtor de Scaffold.

Inheritance  
Object > DiagnosticableTree > Widget > StatefulWidget > Scaffold

### Constructors

```
Scaffold({Key? key, PreferredSizeWidget? appBar, Widget? body, Widget? floatingActionButton, FloatingActionButtonLocation? floatingActionButtonLocation, FloatingActionButtonAnimator? floatingActionButtonAnimator, List<Widget>? persistentFooterButtons, AlignmentDirectional persistentFooterAlignment = AlignmentDirectional.centerEnd, Widget? drawer, DrawerCallback? onDrawerChanged, Widget? endDrawer, DrawerCallback? onEndDrawerChanged, Widget? bottomNavigationBar, Widget? bottomSheet, Color? backgroundColor, bool? resizeToAvoidBottomInset, bool primary = true, DragStartBehavior drawerDragStartBehavior = DragStartBehavior.start, bool extendBody = false, bool extendBodyBehindAppBar = false, Color? drawerScrimColor, double? drawerEdgeDragWidth, bool drawerEnableOpenDragGesture = true, bool endDrawerEnableOpenDragGesture = true, String? restorationId})
```

Creates a visual scaffold for Material Design widgets.

const

**Nota.** O símbolo `?` que aparece colado ao tipo dos parâmetros indica que eles são opcionais.

**Nota.** Os parâmetros que se encontram entre `{ }` na lista de parâmetros do construtor são **parâmetros nomeados**. Eles são especificados na chamada do construtor assim: `ConstrutorQualquer (nomeDoParametro: objetoDesejado)`. Abra o **DartPad** e faça o teste a seguir.

**Não altere seu código no VS Code. Esse é apenas um teste no DartPad.**

<https://dartpad.dev/>

```
//pode ser null
//e implicitamente é null, inicializado
//pelo compilador
void opcional({int? numero}) {
  print(numero);
}

//assim não funciona pois int não pode ser null
//e o valor implícito é null
// void obrigatorio({int numero}) {
//   print(numero);
// }

void obrigatorio({int numero = 5}) {
  print(numero);
}

void comParametrosSemNome(int semNome, {int? comNome}) {
  print(semNome);
  print(comNome);
}

//não funciona
// nomeados (separados por { }) têm de aparecer depois
// de todos os sem nome
// void comParametrosSemNome(int? comNome}, int semNome) {
//   print(semNome);
//   print(comNome);
// }
```

```

void main(){
  //exibe null
  opcional();
  //exibe 2
  opcional(numero: 2);
  //exibe 5
  obrigatorio();
  //exibe 10
  obrigatorio(numero: 10);
  //exibe 2 null
  comParametrosSemNome(2);
  //exibe 2 2
  comParametrosSemNome(2, comNome: 2);
}

```

**Nota.** Não há sobrecarga de métodos/funções em Dart. Lembre-se que Dart é a linguagem de programação que estamos utilizando para programar com o Framework Flutter, que é escrito em Dart.

**Nota.** Observe que o primeiro parâmetro é do tipo **Key**. Esse parâmetro está associado à atualização de estado da aplicação, o que pode envolver a substituição de Widgets. Veja o que a documentação diz.

key

**key** property

`Key? key`  
`final`

Controls how one widget replaces another widget in the tree.

If the `runtimeType` and `key` properties of the two widgets are `operator==`, respectively, then the new widget replaces the old widget by updating the underlying element (i.e., by calling `Element.update` with the new widget). Otherwise, the old element is removed from the tree, the new widget is inflated into an element, and the new element is inserted into the tree.

Leia mais na página a seguir.

<https://api.flutter.dev/flutter/widgets/Widget/key.html>

**Exercício.** Substitua o Widget Text por um Widget Scaffold da seguinte forma:

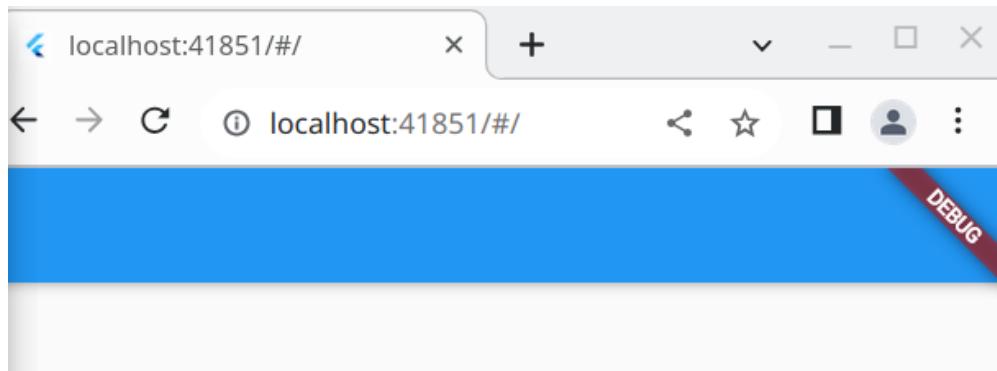
- Deve ser associado à propriedade **home** de MaterialApp.
- Deve ter um parâmetro nomeado **appBar** do tipo **AppBar**
- Observe que o construtor de AppBar não é **const**. Remova const do construtor de MaterialApp.
- Faça hot reload e veja o resultado na tela. Você deverá ver uma barra superior.
- Faça push ao repositório Git.

Veja como fica.

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

void main() {
  var app = MaterialApp(
    home: Scaffold(
      appBar: AppBar(),
    ),
  );
  runApp(app);
}
```

No navegador, o resultado deve ser o seguinte.

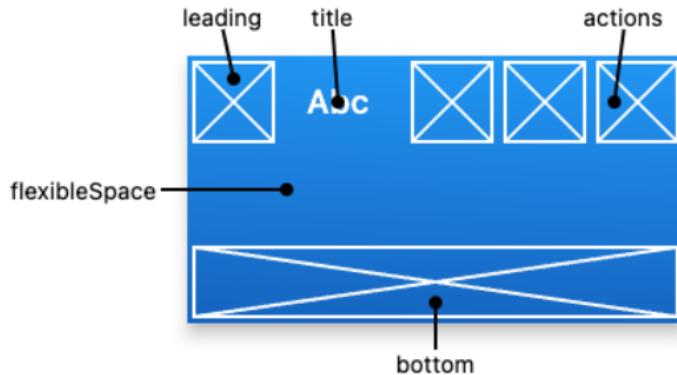


Estudemos um pouco sobre o Widget AppBar. Sua documentação se encontra em

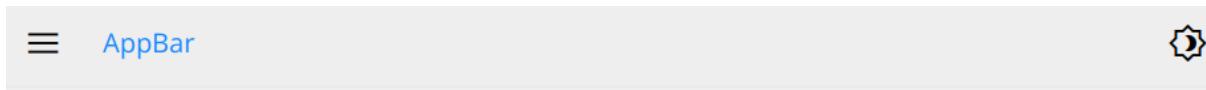
<https://api.flutter.dev/flutter/material/AppBar-class.html>

A figura a seguir, retirada da documentação, mostra o nome de Widgets importantes que podemos encaixar dentro de um AppBar.

The `AppBar` insets its content based on the ambient `MediaQuery`'s padding, to avoid system UI intrusions. It's taken care of by `Scaffold` when used in the `Scaffold.appBar` property. When animating an `AppBar`, unexpected `MediaQuery` changes (as is common in `Hero` animations) may cause the content to suddenly jump. Wrap the `AppBar` in a `MediaQuery` widget, and adjust its padding such that the animation is smooth.



Veja também a lista de parâmetros de seu construtor.



## Constructors

```
AppBar({Key? key, Widget? leading, bool automaticallyImplyLeading = true, Widget? title, List<Widget>? actions, Widget? flexibleSpace, PreferredSizeWidget? bottom, double? elevation, double? scrolledUnderElevation, ScrollNotificationPredicate notificationPredicate = defaultScrollNotificationPredicate, Color? shadowColor, Color? surfaceTintColor, ShapeBorder? shape, Color? backgroundColor, Color? foregroundColor, IconThemeData? iconTheme, IconThemeData? actionsIconTheme, bool primary = true, bool? centerTitle, bool excludeHeaderSemantics = false, double? titleSpacing, double toolbarOpacity = 1.0, double bottomOpacity = 1.0, double? toolbarHeight, double? leadingWidth, TextStyle? toolbarTextStyle, TextStyle? titleTextStyle, SystemUiOverlayStyle? systemOverlayStyle, bool forceMaterialTransparency = false, Clip? clipBehavior})
```

Creates a Material Design app bar.

### Exercício.

- Lendo a documentação de AppBar, descubra o nome do Widget que podemos utilizar para adicionar um título ao Widget AppBar. Adicione o título “Minhas Imagens”.
- Faça hot reload e verifique o resultado no navegador.
- Faça push ao repositório Git.

Veja como fica.

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

void main() {
  var app = MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
    ),
  );
  runApp(app);
}
```

A seguir, vamos lidar com um botão que, quando clicado, deve realizar uma tarefa. Essa tarefa é realizada por uma função. Por isso, vamos estudar um pouco sobre elas em Dart. Faça os seguintes exemplos no DartPad.

```
void main() {
    final f1 = () {
        print('f1');
    };
    final f2 = () {
        print('f2');
        return "f2";
    };
    final f3 = () => print("f3");
    //parece uma função que pode ter várias linhas
    //mas é uma função que devolve um set ou map
    //delimitado por {}
    final f4 = () => {
        //esse print não vale pois {} não é o corpo de
        //uma função
        // print ("E agora?");
    };
    //devolve um Set com 1 e 2
    final f5 = () => {
        1, 2
    }
    //devolve um mapa com nome: 'Ana'
    final f6 = () => {
        'nome': 'Ana'
    };

    //sem inferência de tipo: funções são do tipo Function
    Function f7 = () {
        print('f7');
    };
    //funções podem receber funções
    final f8 = (f) {
        print('f8');
        //chamando a função recebida
        f();
    };
}
```

```

//com parâmetro
final f9 = (int? a){
    print(a);
}
//f1 devolve null
print(f1());
//f2 devolve seu nome
print(f2());
//f3 devolve o que o print que ela chama
//devolve. void neste caso
//por isso, não podemos imprimir o retorno de f3
//print(f3());
//devolve um map vazio
print(f4());
//devolve um set com 1 e 2
print(f5());
//devolve um mapa com nome: Ana
print(f6());
//normal, mas sem inferência de tipo
print(f7());
//f8 recebe f1 e a coloca em execução
print(f8(f1));
//f8 recebe uma arrow function sem nome
print(f8(() => print('oi')));
//recebe 2 como parâmetro
print(f9(2));
}

```

### Exercício.

- Lendo a documentação de **Scaffold**, descubra o nome do Widget que podemos utilizar para adicionar um botão flutuante ao Scaffold.
- Adicione um botão flutuante do tipo **FloatingActionButton**.
- Você terá obtido um erro, indicando que FloatingActionButton requer um parâmetro. Descubra qual é e corrija, lendo a documentação de FloatingActionButton.

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

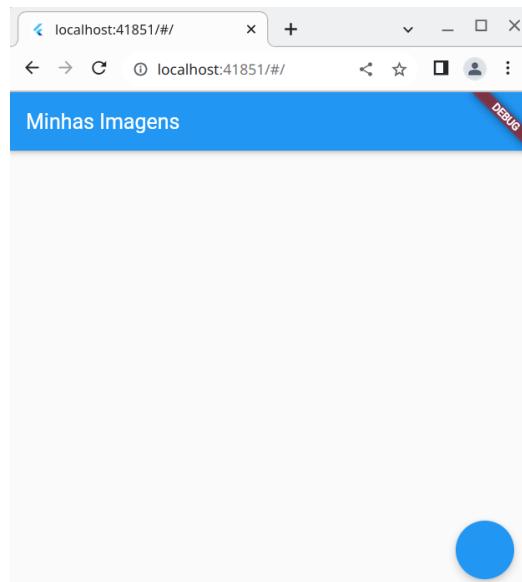
- Faça um hot reload e veja a saída no terminal e no console do navegador (Chrome Dev Tools: CTRL + SHIFT + I).
- Faça push ao repositório Git.

Veja como fica.

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

void main() {
  var app = MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          print("Hello!");
        },
      ),
    ),
  );
  runApp(app);
}
```

No navegador, o resultado esperado é o seguinte.



O botão ainda não exibe conteúdo algum. Precisamos descobrir como fazê-lo mostrar algo. Talvez um símbolo “+” neste caso. Para tal, vamos inspecionar a sua documentação uma vez mais.

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

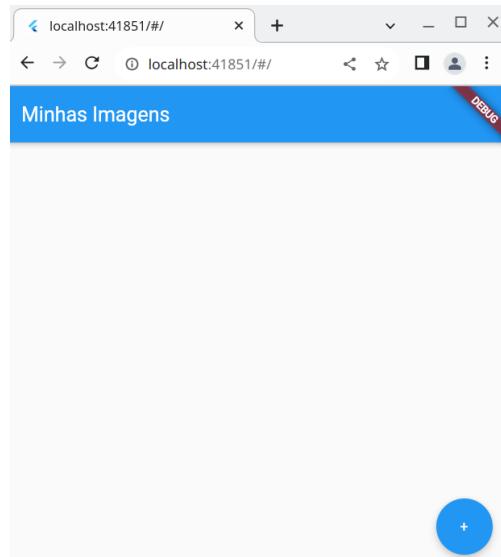
## Constructors

```
FloatingActionButton({Key? key, Widget? child, String? tooltip, Color? foregroundColor, Color? backgroundColor, Color? focusColor, Color? hoverColor, Color? splashColor, Object? heroTag = const _DefaultHeroTag(), double? elevation, double? focusElevation, double? hoverElevation, double? highlightElevation, double? disabledElevation, required VoidCallback? onPressed, MouseCursor? mouseCursor, bool mini = false, ShapeBorder? shape, Clip clipBehavior = Clip.none, FocusNode? focusNode, bool autofocus = false, MaterialTapTargetSize? materialTapTargetSize, bool isExtended = false, bool? enableFeedback})  
Creates a circular floating action button.  
const
```

Para exibir o conteúdo textual, vamos tentar fazer uso do parâmetro nomeado **child**.

```
import 'package:flutter/material.dart';  
  
void main() {  
  var app = MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(  
        title: const Text("Minhas Imagens"),  
      ),  
      floatingActionButton: FloatingActionButton(  
        child: const Text("+"),  
        onPressed: () {  
          print("Hello!");  
        },  
      ),  
    ),  
  );  
  runApp(app);  
}
```

Embora funcione, o resultado visual mostra um símbolo “+” muito pequeno.



Ocorre que não estamos exibindo o ícone “+” corretamente. Observe o destaque. Ele é um ícone, não apenas um texto. Precisamos encontrar o mecanismo apropriado para fazer a sua exibição. Ela pode ser feita usando a classe **Icon**.

### Exercício.

- Estude a documentação da classe **Icon** e descubra como exibir o ícone “+”.

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

#### Constructors

```
Icon(IconData? icon, {Key? key, double? size, double? fill, double? weight, double? grade, double? opticalSize, Color? color, List<Shadow>? shadows, String? semanticLabel, TextDirection? textDirection})  
Creates an icon.  
const
```

**Dica.** Neste caso, vamos fazer uso de um parâmetro não nomeado.

**Dica.** Tente passar uma constante da classe **Icons** ao construtor da classe **Icon**. Para isso, estude também a documentação da classe **Icons**.

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

- Faça hot reload e veja o resultado no navegador.
- Faça push ao repositório Git.

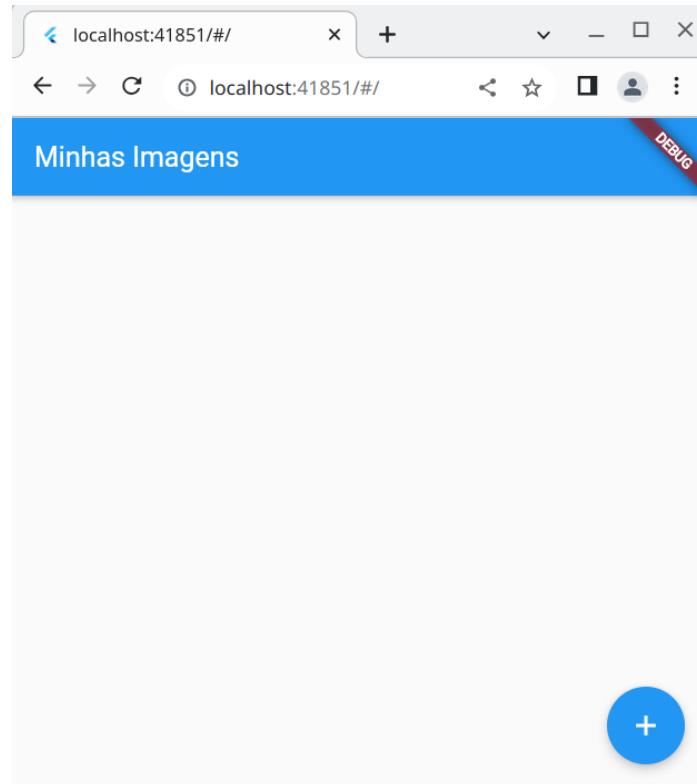
Veja como fica.

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

void main() {
  var app = MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: () {
          print("Hello!");
        },
      ),
    ),
  );
}
```

```
  runApp(app);  
}
```

O resultado no navegador é o seguinte.



**(Criando um Widget Stateless)** Observe que a aplicação inteira está definida em um único arquivo. Claro, esse é um aspecto muito ruim para qualquer ambiente de desenvolvimento de software. Por isso, vamos criar um Widget próprio nosso, com a finalidade de isolar toda a definição que fizemos até então. Este é um bom momento para estudarmos os diferentes tipos de Widgets em Flutter.

**Widget:** Um Widget é uma descrição imutável de parte de uma interface gráfica. Documentação: <https://api.flutter.dev/flutter/widgets/Widget-class.html>

## Widget class abstract



Describes the configuration for an [Element](#).

Widgets are the central class hierarchy in the Flutter framework. A widget is an immutable description of part of a user interface. Widgets can be inflated into elements, which manage the underlying render tree.

Widgets themselves have no mutable state (all their fields must be final). If you wish to associate mutable state with a widget, consider using a  [StatefulWidget](#), which creates a [State](#) object (via  [StatefulWidget.createState](#)) whenever it is inflated into an element and incorporated into the tree.

A given widget can be included in the tree zero or more times. In particular a given widget can be placed in the tree multiple times. Each time a widget is placed in the tree, it is inflated into an [Element](#), which means a widget that is incorporated into the tree multiple times will be inflated multiple times.

The [key](#) property controls how one widget replaces another widget in the tree. If the [runtimeType](#) and [key](#) properties of the two widgets are [operator==](#), respectively, then the new widget replaces the old widget by updating the underlying element (i.e., by calling  [Element.update](#) with the new widget). Otherwise, the old element is removed from the tree, the new widget is inflated into an element, and the new element is inserted into the tree.

**StatelessWidget:** Descreve parte de uma interface gráfica utilizando outros Widgets. "Sem estado" significa que aquilo que ele exibe depende apenas de sua configuração própria e do objeto [BuildContext](#) que recebe no método [build](#). Documentação:

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

A stateless widget is a widget that describes part of the user interface by building a constellation of other widgets that describe the user interface more concretely. The building process continues recursively until the description of the user interface is fully concrete (e.g., consists entirely of  [RenderObjectWidgets](#), which describe concrete  [RenderObjects](#)).



Stateless widget are useful when the part of the user interface you are describing does not depend on anything other than the configuration information in the object itself and the [BuildContext](#) in which the widget is inflated. For compositions that can change dynamically, e.g. due to having an internal clock-driven state, or depending on some system state, consider using  [StatefulWidget](#).

**StatefulWidget:** Descreve parte de uma interface gráfica utilizando outros Widgets. "Com estado" significa que aquilo que ele exibe depende de informações que podem ser obtidas externamente quando ele é construído e que também podem ser alteradas enquanto ele está sendo exibido. Documentação:

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

## StatelessWidget class abstract



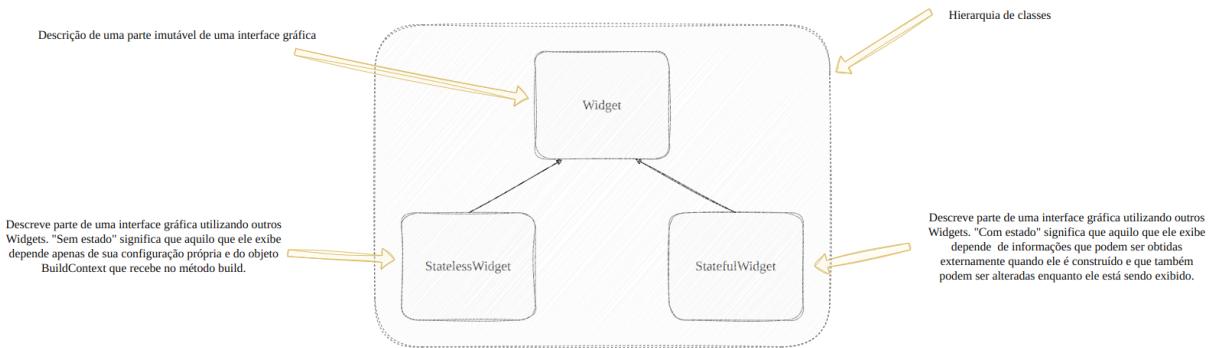
A widget that has mutable state.

State is information that (1) can be read synchronously when the widget is built and (2) might change during the lifetime of the widget. It is the responsibility of the widget implementer to ensure that the [State](#) is promptly notified when such state changes, using [State.setState](#).

A stateful widget is a widget that describes part of the user interface by building a constellation of other widgets that describe the user interface more concretely. The building process continues recursively until the description of the user interface is fully concrete (e.g., consists entirely of [RenderObjectWidgets](#), which describe concrete [RenderObjects](#)).

Stateful widgets are useful when the part of the user interface you are describing can change dynamically, e.g. due to having an internal clock-driven state, or depending on some system state. For compositions that depend only on the configuration information in the object itself and the [BuildContext](#) in which the widget is inflated, consider using  [StatelessWidget](#).

Veja a hierarquia de classes.



### Exercício.

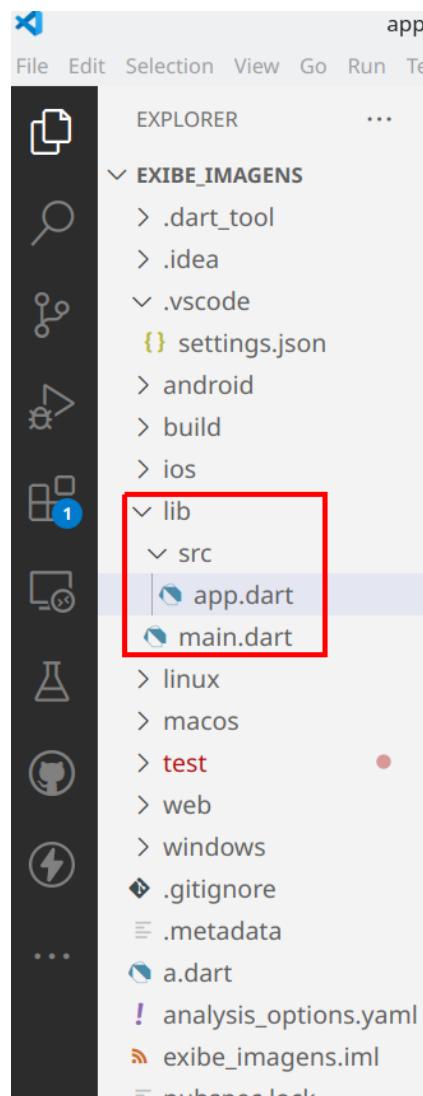
- Crie uma subpasta de **lib** chamada **src**.
- Crie um arquivo chamado **app.dart** na pasta **src**.
- No arquivo **app.dart**, importe a biblioteca material do Flutter.
- No arquivo **app.dart**, crie uma classe chamada **App** e faça com que ela herde de **StatelessWidget**.

**Dica.** O operador de herança em Dart é o **extends**.

- Escreva um método chamado **build** cujo tipo de retorno é **Widget**. Ele deve receber um parâmetro do tipo **BuildContext**.
- Recorte o conteúdo do arquivo **main.dart**, a partir da definição de **MaterialApp** e cole no método **build**, fazendo com que ele devolva o Widget **MaterialApp**. No arquivo **main.dart**, mantenha a declaração da variável **app** sem nada atribuído a ela neste momento.
- Faça o método **print** exibir outro texto, algo como "Estou no arquivo **app.dart**".
- Salve o arquivo **app.dart** para que o código seja indentado automaticamente.

- No arquivo **main.dart**, importe o arquivo **app.dart**.
- Construa uma instância de App no arquivo **main.dart** e faça com que a variável **app** a referencie.
- Faça hot reload e confira o resultado, clicando no FloatingActionButton.
- Faça push ao repositório Git.

Veja como fica.



No arquivo **app.dart**, a definição inicial fica assim.

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

class App extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

Depois disso, o método **build** passa devolver o Widget **MaterialApp** outrora definido no arquivo **main.dart**. Recorte a sua definição do arquivo **main.dart** e cole no arquivo **app.dart**.

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

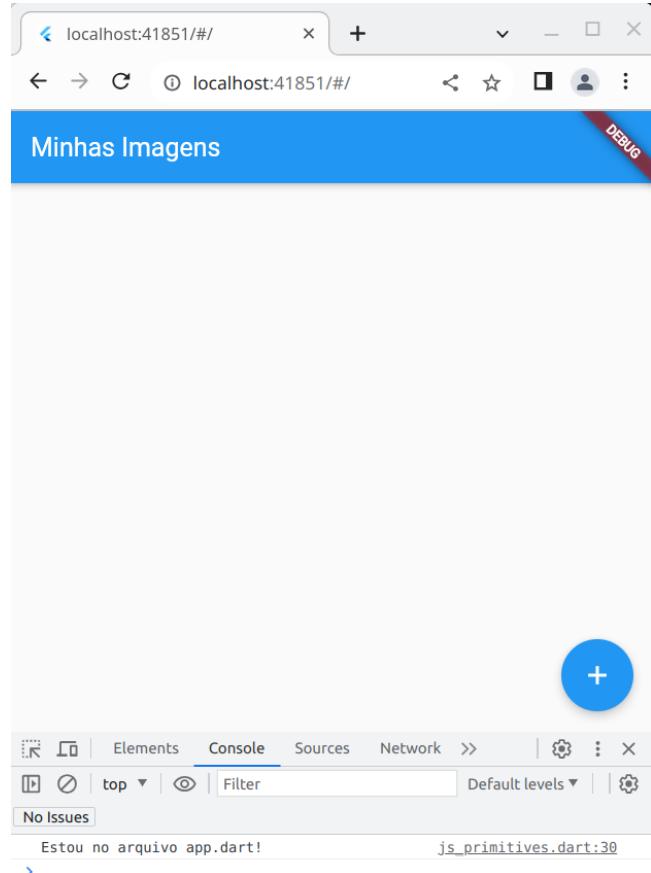
class App extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text("Minhas Imagens"),
        ),
        floatingActionButton: FloatingActionButton(
          child: const Icon(Icons.add),
          onPressed: () {
            print("Estou no arquivo app.dart!");
          },
        ),
        ),
      );
  }
}
```

No arquivo **main.dart**, importamos e passamos a utilizar o novo Widget.

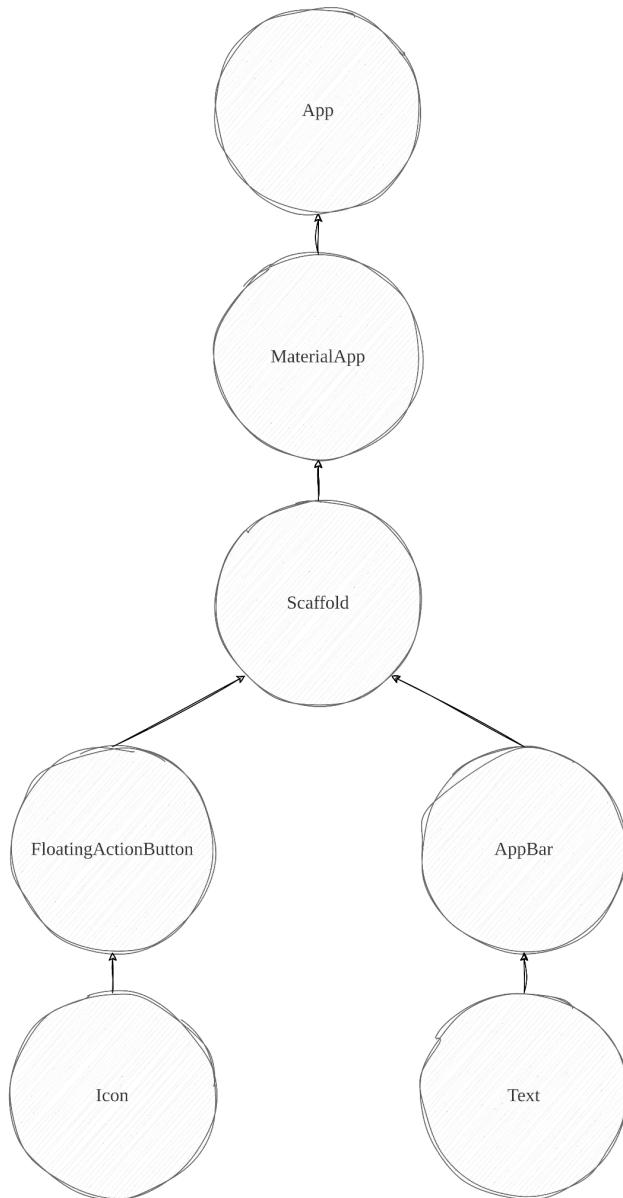
```
import 'package:flutter/material.dart';
import 'src/app.dart';

void main() {
  var app = App();
  runApp(app);
}
```

Aperte SHIFT + R no terminal em que executou flutter run. Clique no FloatingActionButton e veja se funcionou. Veja o resultado esperado.



**(A árvore de Widgets)** A interface gráfica de uma aplicação Flutter é representada por uma árvore de Widgets. Veja a árvore que temos até este momento.



**(Estado. Exibindo o número de imagens: Utilizando um Widget Stateful)** Vamos ajustar a aplicação para que ela exiba o número de imagens exibidas, além das próprias imagens. Essa é uma informação que, evidentemente, será atualizada sempre que o usuário clicar no botão para adicionar nova imagem. Ela faz parte do **estado** do Widget. Por isso, vamos precisar de um Widget com estado. Assim, vamos refatorar a aplicação, deixando de utilizar um **StatelessWidget** e passando a utilizar um **StatefulWidget**.

A refatoração envolve os seguintes passos.

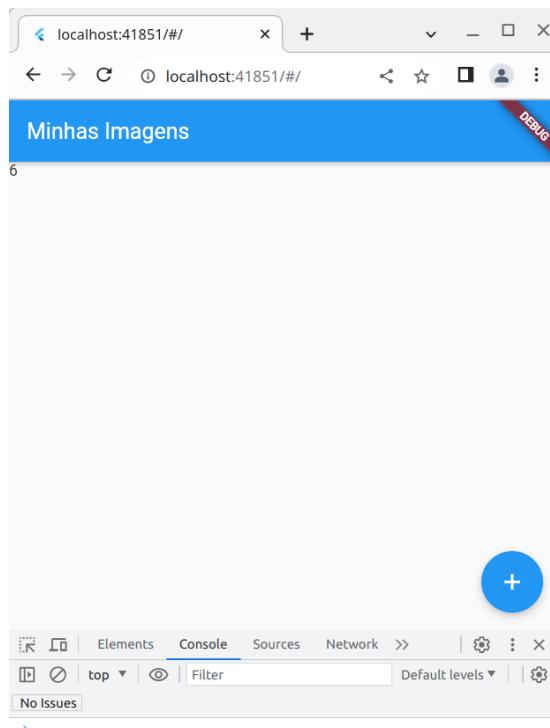
- Escrever duas classes: uma representa o estado da aplicação e a outra representa o Widget.
- A classe que representa o Widget deve possuir um método chamado **createState**. Ele devolve uma instância da classe que representa o estado do Widget.
- A classe que representa o estado da aplicação deve possuir um método chamado **build** que devolve a descrição do Widget de interesse.
- A classe que representa o estado da aplicação deve possuir variáveis de instância. Cada uma representa uma “fatia” do estado de interesse.
- Quando os dados forem alterados, ou seja, quando o usuário clicar no botão, vamos chamar o método **setState**.

#### Exercício.

- No arquivo **app.dart**, renomeie a classe **App** para **AppState**.
- No arquivo **app.dart** crie uma classe chamada **App** que herda de  **StatefulWidget**.
- Faça com que a classe **AppState** herde de **State <App>**. Repare no generics.
- Defina um método chamado **createState** na classe **App**. Seu tipo de retorno deve ser **State <App>**. Ele não recebe coisa alguma como parâmetro. Ele deve devolver uma instância de **AppState**.
- Adicione um inteiro como variável de instância à classe **AppState**. Ela deve se chamar **numerolimagens** e deve ser inicializada com zero.
- Quando o usuário clicar no botão, incremente o contador de imagens.
- Chame o método **setState** depois de incrementar. Passe como parâmetro uma arrow function. Ela deve incrementar o contador. Assim, deixe de incrementá-lo como havia feito anteriormente e apenas incremente na função entregue a **setState**.
- Estude a documentação de Scaffold e descubra como exibir um “corpo”.
 

<https://api.flutter.dev/flutter/material/Scaffold-class.html>
- Use a propriedade encontrada como parâmetro nomeado de Scaffold. Associe a ela um Widget Text. Seu conteúdo deve ser o contador de imagens. Utilize interpolação para exibir seu valor.
- Faça push ao repositório Git.

Veja como fica. Clique algumas vezes no botão para testar.



Primeiro renomeamos a classe **App**.

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

class AppState extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    ...
  }
}
```

Depois disso, criamos uma classe chamada **App** que herda de **StatefulWidget**. Ela possui o método **createState**. Ele devolve uma instância de AppState. Para tal, precisamos fazer com que **AppState** passe a herdar de **State<App>**.

```

import 'package:flutter/material.dart';

class App extends StatefulWidget {
    @override
    State<App> createState() {
        return AppState();
    }
}

class AppState extends State<App> {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            ...
        );
    }
}

```

A seguir, adicionamos o contador de imagens à classe AppState.

```

...
class AppState extends State<App> {
    int numeroImagens = 0;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            ...
        );
    }
}

```

Atualizamos o contador. Num primeiro momento, direto na função que é chamada quando o usuário clica no botão.

```

...
floatingActionButton: FloatingActionButton(
    child: const Icon(Icons.add),
    onPressed: () {
        numeroImagens++;
        print("Estou no arquivo app.dart!");
    },
),
...

```

Chamamos o método `setState` quando o usuário clicar no botão. Ele recebe uma função e incrementa o contador de imagens. O incremento passa a ser feito apenas ali. Também podemos remover a instrução `print`, que estávamos utilizando apenas para um teste inicial.

```
...
floatingActionButton: FloatingActionButton(
  child: const Icon(Icons.add),
  onPressed: () {
    setState(() => numeroImagens++);
  },
),
...

```

Adicionamos o parâmetro `home` `Scaffold` e a ele associamos uma instância de `Text`. Como parâmetro, ela recebe o número de imagens. Repare na interpolação.

```
...
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: () {
          setState(() => numeroImagens++);
        },
      ),
      body: Text('$numeroImagens'), //parâmetro entregue ao construtor de Scaffold
    ),
  );
}
...

```

**(Obtenção de imagens: Pexels)** Vamos obter imagens a partir do site Pexels. Para isso, comece acessando a sua página inicial e criando uma conta para você.

<https://www.pexels.com/>

O acesso ao conteúdo de Pexels é gratuito. Porém, há alguns limites para acessos realizados por meio de requisições direcionadas à sua API.

Segundo a sua documentação, que pode ser acessada a partir do link a seguir, o limite atual é

- 200 requisições por hora
- 20000 requisições por mês

É possível entrar em contato com o suporte e solicitar um aumento gratuito. Basta apresentar razões e exemplos concretos. Se for de interesse da Pexels, você pode até mesmo obter acesso ilimitado.

<https://www.pexels.com/api/documentation/>

## Guidelines

Whenever you are doing an API request make sure to show a **prominent link to Pexels**. You can use a text link (e.g. "Photos provided by Pexels") or a link with our logo.

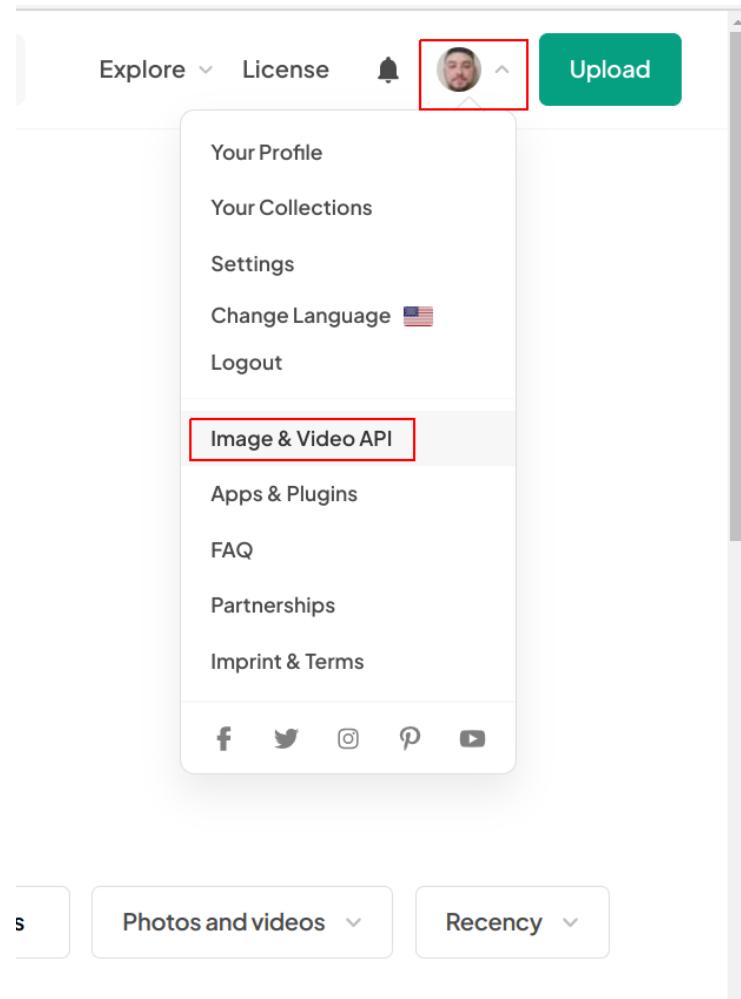
Always credit our photographers when possible (e.g. "Photo by John Doe on Pexels" with a link to the photo page on Pexels).

You may not copy or replicate core functionality of Pexels (including making Pexels content available as a wallpaper app).

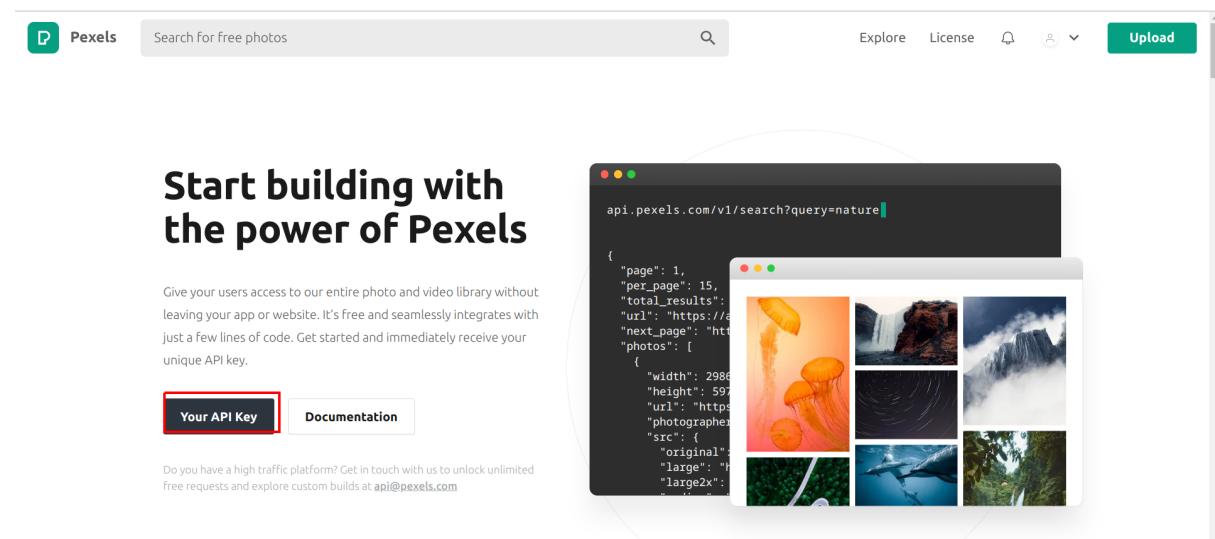
Do not abuse the API. By default, the API is rate-limited to 200 requests per hour and 20,000 requests per month. **You may contact us to request a higher limit**, but please include examples, or be prepared to give a demo, that clearly shows your use of the API with attribution. If you meet our API terms, you can get unlimited requests for free.

Abuse of the Pexels API, including but not limited to attempting to work around the rate limit, will lead to termination of your API access.

Depois de criar a conta e fazer login, clique no seu avatar no canto superior direito e escolha **Image & Video API**.



Clique em **Your API Key** para visualizar a sua chave de API.



**(Trabalhando com objetos JSON)** JSON significa Javascript Object Notation. É uma notação bastante utilizada para a representação de dados que precisam ser enviados e recebidos por aplicações distintas, eventualmente escritas em diferentes linguagens de programação. Veja a sua especificação oficial.

<https://www.json.org/json-en.html>

Visite a página e estude os grafos sintáticos.

Veja alguns exemplos práticos.

Uma pessoa se chama Ana e tem 22 anos.

```
{  
  "nome": "Ana",  
  "idade": 22  
}
```

Uma pessoa se chama João, tem 30 anos e mora na Rua B, número 10, Vila J.

```
{  
  "nome": "João",  
  "idade": 30,  
  "endereco": {  
    "logradouro": "Rua B",  
    "numero": 10,  
    "bairro": "Vila J"  
  }  
}
```

Uma coleção de pessoas. Cada pessoa tem apenas nome.

```
[  
  "Ana",  
  "João",  
  "Pedro"  
]
```

Uma coleção de pessoas. Cada pessoa tem nome e idade.

```
[
  {
    "nome": "Ana",
    "idade": 22
  },
  {
    "nome": "Pedro",
    "idade": 28
  }
]
```

Agora as pessoas são alunos de uma escola. A escola tem um nome.

```
{
  "nome": "Escola A",
  "alunos": [
    {
      "nome": "Ana",
      "idade": 22
    },
    {
      "nome": "Pedro",
      "idade": 28
    }
  ]
}
```

**Exercício.** Represente o seguinte com JSON. Uma concessionária tem um CNPJ e duas filiais. Cada filial tem nome, endereço (logradouro, numero e bairro) e uma coleção de veículos. Cada veículo tem modelo, placa e marca. Cada filial tem dois veículos.

Veja como fica.

```
{  
  "cnpj": "123456789/0001",  
  "filiais": [  
    // filial 1  
    {  
      "nome": "Filial 1",  
      "endereco": {  
        "logradouro": "Rua A",  
        "numero": 100,  
        "bairro": "Vila A"  
      },  
      "veiculos": [  
        {  
          "modelo": "Gol",  
          "marca": "VW",  
          "placa": "ABC-1234"  
        },  
        {  
          "modelo": "Celta",  
          "marca": "Chevrolet",  
          "placa": "DEF-5544"  
        }  
      ]  
    },  
    // filial 2  
    {  
      "nome": "Filial 2",  
      "endereco": {  
        "logradouro": "Rua B",  
        "numero": 100,  
        "bairro": "Vila B"  
      },  
      "veiculos": [  
        {  
          "modelo": "Corsa",  
          "marca": "Chevrolet",  
          "placa": "ERF-1122"  
        },  
        {  
          "modelo": "Fusca",  
          "marca": "VW",  
          "placa": "DEF-1111"  
        }  
      ]  
    }  
  ]  
}
```

**(Manipulando JSON com Dart)** Nesta seção, vamos abrir uma instância do DartPad para aprender como podemos manipular objetos JSON em Dart. Lembre-se que o DartPad está disponível a seguir.

<https://dartpad.dev/>

Comece escrevendo uma função main. Ela define um JSON que representa uma pessoa.

```
import 'dart:convert';

void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';
}
```

Observe que um JSON é uma criatura puramente textual. Se desejarmos extrair alguma informação a respeito da pessoa, como seu nome, teremos de aplicar algoritmos de manipulação de strings, o que pode ser muito repetitivo e trabalhoso. Seria muito mais prático se pudéssemos utilizar uma instrução como “me dê o valor associado à chave “nome””. De fato, podemos. Utilizando mapas. Para isso, basta convertermos o objeto JSON para uma estrutura do tipo mapa. Essa é uma operação muito comum em muitas linguagens de programação. Em Dart, podemos fazê-lo utilizando o pacote **dart:convert**. Ele oferece um objeto chamado json e, por meio dele, acessamos o método **decode**. Ele recebe um JSON e devolve um mapa. Veja.

```
import 'dart:convert';

void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';

  var pessoa = json.decode(pessoaJson);

  print(pessoa);
  // _JsonMap
  print(pessoa.runtimeType);

  print(pessoa['nome']);
  print(pessoa['idade']);

}
```

O mesmo vale para coleções. Observe como podemos iterar sobre elas com um `for each` e também com um `for "regular"`.

```
import 'dart:convert';

void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';

  var pessoa = json.decode(pessoaJson);

  print(pessoa);
  // _JsonMap
  print(pessoa.runtimeType);

  print(pessoa['nome']);
  print(pessoa['idade']);

  var veiculosJson = '''[
    {
      "marca": "VW",
      "modelo": "Gol"
    },
    {
      "marca": "Chevrolet",
      "modelo": "Fusca"
    }
]'''';

  var veiculos = json.decode(veiculosJson);
  print(veiculos);
  // _JSArray<dynamic>
  print(veiculos.runtimeType);
  for (final veiculo in veiculos) {
    print(veiculo);
    print(veiculo['marca']);
    print(veiculo['modelo']);
  }

  for (var i = 0; i < veiculos.length; i++) {
    print(veiculos[i]);
  }
}
```

**Exercício.** Neste exercício, vamos utilizar o objeto “concessionária” previamente ilustrado. Por isso, copie a sua definição para o DartPad, fazendo os ajustes necessários.

- Adicione um preço a cada veículo de cada filial.
- Calcule e mostre o preço médio dos veículos, por filial.
- Calcule e mostre o preço médio dos veículos da concessionária como um todo. Mostre também o total de veículos da concessionária.

Veja como fica.

```
import 'dart:convert';
void main() {
  var concessionariaJSON = '''
  "cnpj": "123456789/0001",
  "filiais": [
    {
      "nome": "Filial 1",
      "endereco": {
        "logradouro": "Rua A",
        "numero": 100,
        "bairro": "Vila A"
      },
      "veiculos": [
        {
          "modelo": "Gol",
          "marca": "VW",
          "placa": "ABC-1234",
          "preco": 20000
        },
        {
          "modelo": "Celta",
          "marca": "Chevrolet",
          "placa": "DEF-5544",
          "preco": 17000
        }
      ]
    },
    {
      "nome": "Filial 2",
      "endereco": {
        "logradouro": "Rua B",
        "numero": 200,
        "bairro": "Vila B"
      },
      "veiculos": [
        {
          "modelo": "Fusca",
          "marca": "VW",
          "placa": "GHI-2345",
          "preco": 30000
        },
        {
          "modelo": "Gol",
          "marca": "VW",
          "placa": "JKL-3456",
          "preco": 22000
        }
      ]
    }
  ]
}'''
```

```

"nome": "Filial 2",
"endereco": {
    "logradouro": "Rua B",
    "numero": 100,
    "bairro": "Vila B"
},
"veiculos": [
    {
        "modelo": "Corsa",
        "marca": "Chevrolet",
        "placa": "ERF-1122",
        "preco": 12000
    },
    {
        "modelo": "Fusca",
        "marca": "VW",
        "placa": "DEF-1111",
        "preco": 15000
    }
]
}
}''';

var concessionaria = json.decode(concessionariaJSON);
print(concessionaria.runtimeType);
var filiais = concessionaria['filiais'];
print(filiais.runtimeType);
double mediaTotal = 0;
int totalVeiculos = 0;
for (final filial in filiais){
    var veiculos = filial['veiculos'];
    double media = 0;
    for (final veiculo in veiculos){
        media += veiculo['preco'];
        mediaTotal += veiculo['preco'];
        totalVeiculos++;
    }
    print('Filial ${filial['nome']}: ${media/veiculos.length}');
}
print('Média total: ${mediaTotal / totalVeiculos}');
print('Total de veículos: ${totalVeiculos}');

```

```
}
```

**(Classes de modelo)** Os valores contidos na estrutura que a função **decode** nos entrega têm tipo conhecido somente em tempo de execução (dynamic). Podemos deixar o código mais seguro e previsível convertendo essa estrutura para uma classe de modelo. Esta classe contém os campos desejados e eles são definidos utilizando-se o sistema de tipos estático da linguagem. No DartPad, volte para o seguinte modelo simples.

```
import 'dart:convert';
void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';
}
```

Depois disso, escrevemos uma classe de Modelo. Ela tem as propriedades de interesse e um construtor.

```
import 'dart:convert';
void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';
}

class PessoaModel{
  String nome;
  int idade;
  //construtor
  PessoaModel(this.nome, this.idade);
}
```

Por fim, fazemos a conversão com o método **decode** e construímos um objeto de modelo. O que ganhamos essencialmente foi o uso do sistema de tipos estático e, portanto, o suporte do compilador.

```

import 'dart:convert';
void main() {
  var pessoaJson = '{"nome": "Ana", "idade": 18}';
  var pessoaDecoded = json.decode(pessoaJson);
  var pessoa = PessoaModel(
    pessoaDecoded['nome'],
    pessoaDecoded['idade']
  );
  //agora podemos acessar assim:
  print(pessoa.nome);
  print(pessoa.idade);
}
class PessoaModel{
  String nome;
  int idade;
  //construtor
  PessoaModel(this.nome, this.idade);
}

```

**(Construtores nomeados)** Nesta seção, vamos definir um construtor **nomeado**. Ele terá as seguintes propriedades:

- receberá o objeto JSON produzido pela função decode.
- devolverá um objeto construído a partir da classe de Modelo, já com os campos preenchidos. O objetivo é isolar e escrever uma única vez o seguinte trecho:

```

//não altere nada, apenas observe
var pessoa = PessoaModel(
  pessoaDecoded['nome'],
  pessoaDecoded['idade']
);

```

O trecho citado incomoda pois pode se tornar muito grande e repetitivo caso o objeto envolvido possua muitas propriedades.

Começamos definindo o construtor nomeado.

```
...
class PessoaModel {
  String nome;
  int idade;

  //construtor
  PessoaModel(this.nome, this.idade);

  //construtor nomeado
  PessoaModel.fromJson(decodedJSON) {
    nome = decodedJSON['nome'];
    idade = decodedJSON['idade'];
  }
}
```

Observe que o código causa um erro.

```
21
22  //construtor nomeado
23  PessoaModel.fromJson(decodedJSON) {
24    nome = decodedJSON['nome'];
25    idade = Non-nullable instance field 'nome'.
26  };
27 }
```

Ocorre que, quando o construtor nomeado entra em execução, o ambiente Dart já espera que os campos tenham sido inicializados. Podemos ajustar marcando-os como opcionais ou “prometendo” que eles terão sido inicializados posteriormente, antes de serem acessados. Escolhamos a segunda opção. Para isso, marcamos as variáveis com **late**.

```

...
class PessoaModel {
    late String nome;
    late int idade;

    //construtor
    PessoaModel(this.nome, this.idade);

    //construtor nomeado
    PessoaModel.fromJSON(decodedJSON) {
        nome = decodedJSON['nome'];
        idade = decodedJSON['idade'];
    }
}

```

Passamos a construir o objeto assim.

```

import 'dart:convert';
void main() {
    var pessoaJson = '{"nome": "Ana", "idade": 18}';
    var pessoaDecoded = json.decode(pessoaJson);
    var pessoa = PessoaModel.fromJSON(pessoaDecoded);
    //agora podemos acessar assim:
    print(pessoa.nome);
    print(pessoa.idade);
}

class PessoaModel{
    late String nome;
    late int idade;
    //construtor
    PessoaModel(this.nome, this.idade);
    //construtor nomeado
    PessoaModel.fromJSON(decodedJSON) {
        nome = decodedJSON['nome'];
        idade = decodedJSON['idade'];
    }
}

```

**(Modelo para a imagem)** De volta ao projeto, vamos criar uma classe de modelo para as imagens que a Pexels nos entregará. Veja um exemplo de JSON que ela nos entrega.

```

2   "page": 1,
3   "per_page": 1,
4   "photos": [
5     {
6       "id": 5637733,
7       "width": 3840,
8       "height": 5760,
9       "url": "https://www.pexels.com/photo/man-in-yellow-jacket-and-black-pants-walking-on-sidewalk-5637733/",
10      "photographer": "RDNE Stock project",
11      "photographer_url": "https://www.pexels.com/@rdne",
12      "photographer_id": 3149039,
13      "avg_color": "#AD9A4A",
14      "src": {
15        "original": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg",
16        "large2x": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&dpr=2&h=650&w=940",
17        "large": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&h=650&w=940",
18        "medium": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&h=350",
19        "small": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&h=130",
20        "portrait": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&fit=crop&h=1200&w=800",
21        "landscape": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&fit=crop&h=627&w=1200",
22        "tiny": "https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&cs=tinysrgb&dpr=1&fit=crop&h=200&w=280"
23      },
24      "liked": false,
25      "alt": "Man in Yellow Jacket and Black Pants Walking on Sidewalk"
26    }
27  ],
28  "total_results": 8000,
29  "next_page": "https://api.pexels.com/v1/search/?page=2&per_page=1&query=people"
-- .
```

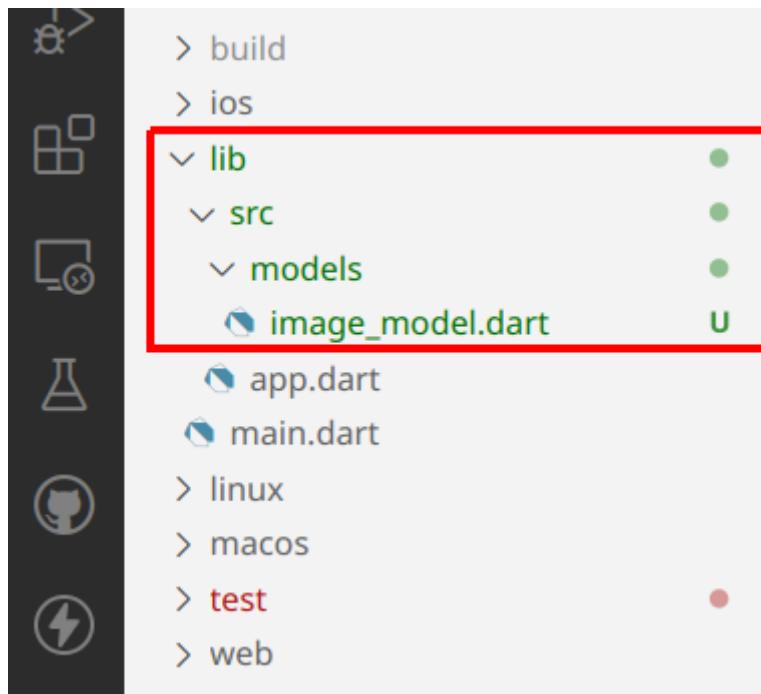
↖ ↘

Observe que o resultado possui uma coleção chamada **photos**. Cada posição contém uma foto. Cada foto tem algumas propriedades, como:

- **url**: endereço dela que nos leva para uma página do site Pexels. Não é esse que desejamos
- **src**: um JSON com propriedades como original, large2x, medium, small etc. É daí que escolheremos a URL a utilizar na aplicação. A foto é sempre a mesma, o que muda é o tamanho.
- **alt**: é uma descrição textual da foto.

Entre outras propriedades. Inspecione o resultado para conhecer mais.

Para criar o modelo de imagem, crie uma pasta chamada **models** na pasta **src** de seu projeto. A seguir, crie um arquivo chamado **image\_model.dart**. Observe.



Digamos que o modelo, a princípio, tem as seguintes propriedades: url e alt.

**Exercício.** Escreva a classe de modelo com url e alt, considerando a estrutura do objeto devolvido pela Pexels.

Veja como fica.

```
class ImageModel {
  late String url;
  late String alt;

  ImageModel(this.url, this.alt);

  ImageModel.fromJSON(Map <String, dynamic> decodedJSON) {
    //o resultado tem uma coleção de fotos
    //vamos pegar sempre a primeira
    //por isso, posição zero
    //dela, pegamos o tamanho médio (propriedade do objeto src)
    url = decodedJSON['photos'][0]['src']['medium'];
    alt = decodedJSON['photos'][0]['alt'];
  }
}
```

**Nota.** Há uma outra forma de definição do corpo de um construtor. Veja.

```
class ImageModel {
  late String url;
  late String alt;

  ImageModel(this.url, this.alt);

  // ImageModel.fromJSON(Map <String, dynamic> decodedJSON) {
  //   // o resultado tem uma coleção de fotos
  //   // vamos pegar sempre a primeira
  //   // por isso, posição zero
  //   // dela, pegamos o tamanho médio (propriedade do objeto src)
  //   url = decodedJSON['photos'][0]['src']['medium'];
  //   alt = decodedJSON['photos'][0]['alt'];
  // }

  ImageModel.fromJSON(Map <String, dynamic> decodedJSON)
    : url = decodedJSON['photos'][0]['src']['medium'],
      alt = decodedJSON['photos'][0]['alt'];
}
```

Neste material, entretanto, vamos manter a primeira forma.

**(Requisição HTTP para obter uma imagem)** Quando o usuário clicar no botão, desejamos disparar uma requisição HTTP direcionada aos servidores da Pexels a fim de obter uma imagem. Para isso, vamos escrever uma função que tem essa responsabilidade. Estamos no arquivo **app.dart**.

```
...
class AppState extends State<App> {
  int numeroImagens = 0;

  void obterImagem() {
    ...
  }

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

A seguir, na função associada ao parâmetro `onPressed` do `FloatingActionButton`, vamos deixar de atualizar o estado e chamar a função criada.

```
...
floatingActionButton: FloatingActionButton(
  child: const Icon(Icons.add),
  onPressed: () {
    obterImagem();
  },
),
...
```

Observe que, agora, temos a especificação de uma função que não faz nada além de chamar outra. Assim, podemos especificar apenas o nome da função interna. Veja.

```
...
floatingActionButton: FloatingActionButton(
  child: const Icon(Icons.add),
  onPressed: obterImagem,
),
...
```

**Cuidado.** Especificar apenas o nome da função significa deixá-la associada ao parâmetro nomeado no sentido de que ela deve ser chamada quando o evento envolvido acontecer. Isso é completamente diferente de chamá-la explicitamente, como a seguir.

```
floatingActionButton: FloatingActionButton(
  child: const Icon(Icons.add),
  onPressed: obterImagem(), //errado, não faça isso
),
```

Em Dart, isso é um erro em tempo de compilação, pois a nossa função devolve “void”. Assim que o método `build` é chamado, ela também é chamada (ao invés de apenas ser chamada quando o evento de clique no botão acontecer) e, assim, estamos tentando associar “void” ao parâmetro `onPressed`. É claro, não podemos fazer isso. Em outros ambientes, como no React, esse é um erro comum que faz com que a função seja chamada assim que o componente aparece na tela. Em geral, isso faz com que a tela seja atualizada e a função seja chamada de novo, causando uma recursão infinita.

Para realizar as requisições HTTP, vamos utilizar o pacote **http**. O primeiro passo é fazer a sua instalação. Num terminal vinculado à raiz da aplicação, execute

```
flutter pub add http
```

Investigue a seção **dependencies** de seu arquivo **pubspec.yaml**. A nova dependência deve ter sido adicionada ali.



```

app.dart M
! pubspec.yaml M x
! pubspec.yaml

27 # dependencies can be manually updated by char
numbers below to
28 # the latest version available on pub.dev. To
dependencies have newer
29 # versions available, run `flutter pub outdated`
30 dependencies:
31   flutter:
32     |   sdk: flutter
33
34
35   # The following adds the Cupertino Icons for
36   # Use with the CupertinoIcons class for iOS
37   cupertino_icons: ^1.0.2
38   http: ^1.1.0
39
40 dev_dependencies:
41   flutter_test:
42     |   ...

```

Visite também a documentação do pacote para conhecer mais sobre ele.

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

O próximo passo é importar o pacote no arquivo **app.dart**.

```

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

```

O **import** que fizemos foi “sem nome”. Assim, as funções do pacote podem ser acessadas diretamente, sem qualificação. Veja como falamos **get** diretamente no código a seguir, sem qualificação.

```
...
void obterImagem() {
  get(Uri.parse(''));
}
...
```

Também podemos dar um nome ao pacote. Assim, o acesso a suas funções precisa ser qualificado. Observe.

```
...
import 'package:http/http.dart' as http;

...
void obterImagem() {
  http.get(Uri.parse(''));
}
...
```

**Nota.** Há também as construções **show** e **hide** que também podem ser envolvidas numa instruções import. A documentação a seguir explica isso.

<https://dart.dev/language/libraries>

#### Importing only part of a library

If you want to use only part of a library, you can selectively import the library. For example:

```
// Import only foo.
import 'package:lib1/lib1.dart' show foo;

// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

Além disso, há um mecanismo conhecido como **lazy loading**. Quando utilizado, a aplicação apenas importa a biblioteca quando ela é utilizada. Observe que ele funciona apenas na **web**.

### Lazily loading a library

*Deferred loading* (also called *lazy loading*) allows a web app to load a library on demand, if and when the library is needed. Here are some cases when you might use deferred loading:

- To reduce a web app's initial startup time.
- To perform A/B testing—trying out alternative implementations of an algorithm, for example.
- To load rarely used functionality, such as optional screens and dialogs.

⚠ Only `dart compile js` supports deferred loading. Flutter and the Dart VM don't support deferred loading. To learn more, see [issue #33118](#) and [issue #27776](#).

To lazily load a library, you must first import it using `deferred as`.

```
import 'package:greetings/hello.dart' deferred as hello;
```

Visite sempre a documentação a fim de obter informações mais atualizadas.

Neste material, vamos manter o import com nome e acesso qualificado.

```
...
import 'package:http/http.dart' as http;
...
void obterImagem() {
  http.get(Uri.parse(' '));
}
....
```

**(URL Pexels e seus parâmetros)** A URL de acesso à Pexels é a seguinte.

<https://api.pexels.com/v1/search>

Para especificar o assunto da foto que desejamos, utilizamos o parâmetro `query`. Fica assim:

<https://api.pexels.com/v1/search?query=people>

Além disso, os resultados que a Pexels nos entrega são **paginados**.

Isso funciona da seguinte forma:

- Uma consulta por people pode resultar numa coleção muito grande de resultados, contendo milhares ou milhões de fotos.
- Na primeira resposta, a Pexels nos entrega apenas uma quantidade pequena de fotos, digamos 15. Além disso, ela nos diz que estamos na “página 1”.
- Na próxima requisição podemos obter mais 15 resultados, instruindo a Pexels a nos entregar os resultados da página 2.

E assim por diante.

Os parâmetros envolvidos se chamam **page** e **per\_page**. Eles indicam, respectivamente, o número da página de interesse e o número de itens por página. A URL final fica assim:

[https://api.pexels.com/v1/search?query=people&per\\_page=1&page=1](https://api.pexels.com/v1/search?query=people&per_page=1&page=1)

Para pedir uma nova foto, podemos acessar a URL

[https://api.pexels.com/v1/search?query=people&per\\_page=1&page=2](https://api.pexels.com/v1/search?query=people&per_page=1&page=2)

Ou seja, incrementamos o número da página. Claro, também podemos aumentar o número de resultados por página. Depende da natureza da aplicação.

Assim, podemos realizar a requisição.

**Exercício.** No método `obterImagem`:

1. Construa um objeto url a partir de `Uri.https`.
  - primeiro parâmetro: host da pexels
  - segundo parâmetro: recurso
  - terceiro parâmetro: mapa de parâmetros da query
2. Construa um objeto `Request` (a partir do objeto `http`)
  - Primeiro parâmetro: `get` (é o nome do método de interesse)
  - segundo parâmetro: o objeto `url`
3. Acesse a propriedade **headers** do objeto `Request` e dela, chame o método **addAll**.
  - Primeiro parâmetro: um mapa com chave `Authorization` associada à sua chave Pexels.
4. Envie a requisição com `send`.

Veja como fica.

**Nota.** O valor de **page** está fixo em 1, neste momento. No futuro, a aplicação será atualizada para que este valor seja atualizado a cada requisição. Assim, teremos uma nova imagem a cada clique. No momento, cada clique nos produz exatamente a mesma imagem.

```
...
void obterImagem() {
    var url = Uri.https(
        'api.pexels.com',
        '/v1/search',
        {'query': 'people', 'page': '1', 'per_page': '1'},
    );
    var req = http.Request('get', url);
    req.headers.addAll(
    {
        'Authorization':
        'chave',
    },
    );
    req.send();
}
...
```

E agora, como verificar o resultado? Veja a documentação da função **get**.

<https://pub.dev/documentation/http/latest/http/get.html>

http package > documentation > http > get function Search AP

http library

**CLASSES**

- BaseClient
- BaseRequest
- BaseResponse
- ByteStream
- Client
- MultipartFile
- MultipartRequest
- Request
- Response
- StreamedRequest
- StreamedResponse

**FUNCTIONS**

- delete
- get

## get function

```
Future<Response> get(
    Uri url,
    {Map<String, String>? headers}
)
```

Sends an HTTP GET request with the given headers to the given URL.

This automatically initializes a new `Client` and closes that client once the request is complete. If you're planning on making multiple requests to the same server, you should use a single `Client` for all of those requests.

For more fine-grained control over the request, use `Request` instead.

### Implementation

```
Future<Response> get(Uri url, {Map<String, String>? headers}) =>
    _withClient((client) => client.get(url, headers: headers));
```

Veja também a documentação da função **send** de `Request`.

<https://pub.dev/documentation/http/latest/http/BaseRequest/send.html>

http package > documentation > http > BaseRequest > send method Search AP

BaseRequest class

**CONSTRUCTORS**

- BaseRequest

**PROPERTIES**

- contentLength
- finalized
- followRedirects
- hashCode
- headers
- maxRedirects
- method
- persistentConnection
- runtimeType
- url

**METHODS**

## send method

```
Future<StreamedResponse> send()
```

Sends this request.

This automatically initializes a new `Client` and closes that client once the request is complete. If you're planning on making multiple requests to the same server, you should use a single `Client` for all of those requests.

### Implementation

```
Future<StreamedResponse> send() async {
    var client = Client();

    try {
        var response = await client.send(this);
        var stream = onDone(response.stream, client.close);
        return StreamedResponse(ByteStream(stream), response.statusCode,
            contentLength: response.contentLength,
            request: response.request
    }
}
```

Ambas devolvem um objeto do tipo **Future**. O que é um objeto do tipo Future?

**Nota.** Um Future é um objeto que fica associado a uma computação assíncrona, não bloqueante, potencialmente demorada. Ele nos permite obter o resultado calculado por esta computação, no futuro, quando ela terminar. Não bloqueante ou assíncrono quer dizer que o fluxo principal de nossa aplicação prossegue normalmente e a função demorada opera em background. Assim, não congelamos a interface com o usuário.

**Nota.** Um Future em Dart é equivalente a uma Promise em Javascript. Não é necessário conhecer Promises para entender o que são Futures. E vice-versa. Essa é apenas uma analogia que pode ser útil.

Já temos o objeto **Future** em mãos. Ele nos foi entregue pela função `send`. Agora precisamos descobrir como podemos acessar o resultado calculado pela função demorada, quando ela terminar. Neste caso, a função demorada é o acesso à Pexels e o resultado que desejamos é o objeto JSON. Podemos fazer uso da função **then** para obter o resultado.

**Nota.** Entenda a função **then** da seguinte forma: “execute a computação demorada associada a este Future e então, execute o código da função entregue por parâmetro à função `then`.

Vejamos um primeiro teste usando a função **then**.

```
...
void obterImagem() {
  var url = Uri.https(
    'api.pexels.com',
    '/v1/search',
    {'query': 'people', 'page': '1', 'per_page': '1'},
  );
  var req = http.Request('get', url);
  req.headers.addAll(
  {
    'Authorization':
    'chave',
  },
  );
  req.send().then((result) {
    print(result);
  });
}
...
```

Veja o resultado esperado.

---



---

Performing hot restart...  
 621ms  
 Restarted application in 621ms.  
 Instance of 'StreamedResponse'  
 □

Ainda não tem muita graça. Temos uma instância de **StreamedResponse** em mãos. Veja a sua definição, dada pela documentação.

<https://pub.dev/documentation/http/latest/http/StreamedResponse-class.html>

StreamedResponse class

An HTTP response where the response body is received asynchronously after the headers have been received.

Inheritance: Object > BaseResponse > StreamedResponse

Implementers: IOStreamedResponse

Temos um objeto cujos dados internos podem ser obtidos pouco a pouco, sob demanda. É um “fluxo(stream) de resposta”. Dele podemos pegar a propriedade **stream**.

```
...
req.send().then((result) {
  print(result.stream);
});
```

Veja o resultado.

409MS  
 Restarted application in 460ms.  
 Instance of 'ByteStream'  
 □

Ainda não tem graça. Podemos verificar o código de status. Se ele for 200 (sucesso, segundo a especificação do protocolo HTTP), fazemos algo com o resultado. Neste caso, vamos converter nosso StreamedResponse para um Response. Um Response nos permite

obter os dados de interesse por meio de propriedades previamente definidas. Como essa conversão também pode ser demorada, temos de lidar com um novo objeto Future. Veja.

```
...
req.send().then((result) {
  if (result.statusCode == 200) {
    http.Response.fromStream(result).then((response) {
      print(response.body);
    });
  } else {
    print('Falhou!');
  }
});
...
...
```

Agora o resultado ficou mais interessante.

```
Performing hot restart...
408ms
Restarted application in 408ms.
{"page":1,"per_page":1,"photos":[{"id":5637733,"width":3840,"height":5760,"url":"https://www.pexels.com/photo/man-in-yellow-jacket-and-black-pants-walking-on-sidewalk-5637733/","photographer":"RDNE Stock project","photographer_url":"https://www.pexels.com/@rdne","photographer_id":3149039,"avg_color": "#ADA9A4","src":{"original":"https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg","large2x":"https://images.pexels.com/photos/5637733/pexels-photo-5637733.jpeg?auto=compress&u0026cs=tinysrgb&u0026dpr=2&u0026h=650\u0026
```

Podemos até decodificar o JSON recebido e converter para um ImageModel, não é mesmo?

```
import 'package:flutter/material.dart';
import 'dart:convert';
import 'models/image_model.dart';
import 'package:http/http.dart' as http;
...
req.send().then((result) {
  if (result.statusCode == 200) {
    http.Response.fromStream(result).then((response) {
      var decodedJSON = json.decode(response.body);
    });
  }
});
```

```

        var imagem = ImageModel.fromJSON(decodedJSON);
        print(imagem);
    });
} else {
    print('Falhou!');
}
);
...

```

Antes de executar, pode ser boa ideia fazer a sobrescrita do método **toString** da classe **ImageModel**. Assim, obtemos uma representação textual mais interessante de seus objetos. Vá até o arquivo **image\_model.dart**.

**Exercício.** Faça uma sobrescrita do método `toString` para a classe `ImageModel`. Ele deve produzir uma `String` contendo ambas as propriedades do objeto.

Veja uma possível sobrescrita. Claro, você pode variar se desejar.

```

class ImageModel {
    late String url;
    late String alt;

    ImageModel(this.url, this.alt);

    ImageModel.fromJSON(Map<String, dynamic> decodedJSON) {
        //o resultado tem uma coleção de fotos
        //vamos pegar sempre a primeira
        //por isso, posição zero
        //dela, pegamos o tamanho médio (propriedade do objeto src)
        url = decodedJSON['photos'][0]['src']['medium'];
        alt = decodedJSON['photos'][0]['alt'];
    }

    @override
    String toString() {
        return 'url:$url, alt:$alt';
    }
}

```

Execute mais uma vez para ver o resultado.

```

Performing hot restart...
  410ms
Restarted application in 410ms.
url:https://images.pexels.com/photos/5637733/pexels-photo-5637733.
jpeg?auto=compress&cs=tinysrgb&h=350, alt:Man in Yellow Jacket and
Black Pants Walking on Sidewalk

```

█

Bem mais interessante, né?

Embora funcione, o código que envolve o método **then** pode ser difícil de entender e de manter. Há uma construção chamada **async/wait** (também semelhante àquela de Javascript) que nos permite escrever código assíncrono num formato “sequencial”, mais fácil de entender, manter e depurar. Começamos marcando a função que fará uso da construção como **async**.

```

...
void obterImagem() async {
  var url = Uri.https(
  ...
}
```

A seguir, deixamos de chamar a função **then** e “aguardamos” pelo resultado da computação demorada, guardando seu resultado de modo aparentemente sequencial. Para isso, marcamos a chamada à função que produz um Future com **await**. Veja.

```

void obterImagem() async {
  var url = Uri.https(
    'api.pexels.com',
    '/v1/search',
    {'query': 'people', 'page': '1', 'per_page': '1'},
  );
  var req = http.Request('get', url);
  req.headers.addAll(
  {
    'Authorization':
    'chave',
  },
  );

  final result = await req.send();
  if (result.statusCode == 200) {
    final response = await http.Response.fromStream(result);
    var decodedJSON = json.decode(response.body);
  }
}
```

```

    var imagem = ImageModel.fromJson(decodedJSON);
    print(imagem);
} else {
    print('Falhou!');
}
}

```

**(Criando uma lista de ImageModel)** Sempre que tivermos uma nova imagem, ela será representada por uma instância de ImageModel e armazenada em uma lista. Essa atualização de dados envolve a atualização da tela. Isso quer dizer que a lista será uma **variável de estado**. Veja.

```

...
class AppState extends State<App> {
    int numeroImagens = 0;
    List<ImageModel> imagens = [];
...

```

A cada clique, adicionamos uma nova instância de ImageModel à lista.

```

...
final result = await req.send();
if (result.statusCode == 200) {
    final response = await http.Response.fromStream(result);
    var decodedJSON = json.decode(response.body);
    var imagem = ImageModel.fromJson(decodedJSON);
    imagens.add(imagem);
} else {
...

```

Lembre-se, entretanto, que estamos lidando com uma variável de estado. Assim, precisamos fazer a atualização de maneira adequada, usando o método **setState**. Ele se encarrega de atualizar a tela quando for a hora.

```

...
final result = await req.send();
if (result.statusCode == 200) {
  final response = await http.Response.fromStream(result);
  var decodedJSON = json.decode(response.body);
  var imagem = ImageModel.fromJSON(decodedJSON);
  setState(() {
    imagens.add(imagem);
  });
} else {
  print('Falhou!');
}
...

```

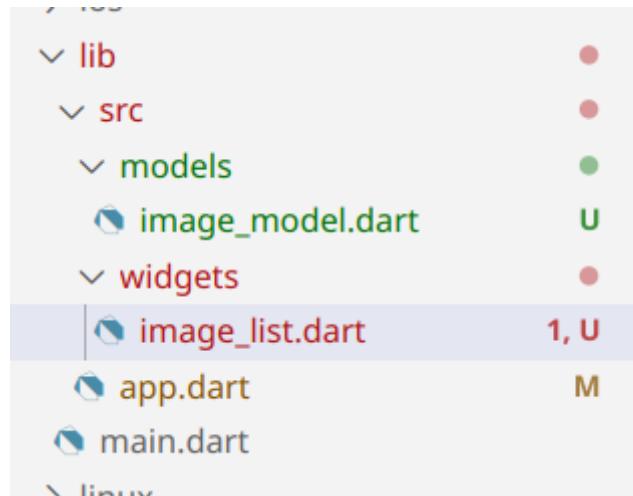
Neste ponto, a nossa aplicação merece nova refatoração. Observe como a classe **AppState** está bem grandinha.

### Exercício.

1. Crie uma pasta **widgets**, subpasta de src.
2. Crie um arquivo chamado **image\_list.dart** na pasta widgets.
3. Importe a biblioteca material de flutter.
4. Importe a classe **ImageModel**.
5. Defina uma classe chamada **ImageList**. Ela deve passar no teste É-UM **StatelessWidget**. Já já vemos por que ele não é um **StatefulWidget**.
6. Defina o método **build**: ele devolve **Widget** e recebe **BuildContext**.
7. No arquivo **app.dart**, importe o arquivo **image\_list.dart**.
8. Ajuste a propriedade **body** de **Scaffold**. Ele deixa de exibir um **Text** e passa a exibir um **ImageList**.
9. Ainda em **app.dart**, ao construir um **ImageList**, passe como parâmetro a lista de **ImageModel**. Isso causará um erro em tempo de compilação.
10. No arquivo **image\_list.dart** defina uma variável de instância chamada **imagens**. Ela deve ser do tipo **List<ImageModel>**. Já que o widget é sem estado, ela deve ser marcada final.
11. No arquivo **image\_list.dart**, escreva um construtor que recebe uma lista de **ImageModel**.

**Nota.** O Widget que criamos é “sem estado” pois uma nova instância dele é construída a cada atualização de tela. Cada nova instância recebe, como parâmetro do construtor, a nova lista a ser exibida, já atualizada. A atualização da lista fica a cargo do Widget principal. Por essa razão, ele é “com estado”

Veja como fica.



```
//arquivo image_list.dart
import 'package:flutter/material.dart';
import '../models/image_model.dart';

class ImageList extends StatelessWidget {
  final List<ImageModel> imagens;
  ImageList(this.imagens);
  @override
  Widget build(BuildContext context) {
  }
}
```

## Arquivo **app.dart**.

```

import 'models/image_model.dart';
import 'widgets/image_list.dart';
import 'package:flutter/material.dart';
import 'dart:convert';
import 'package:http/http.dart' as http;
...
@Override
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: obterImagen,
      ),
      body: ImageList(imagens),
    ) );
}
...

```

**(Exibição de widgets com ListView)** Um Widget do catálogo oficial capaz de exibir uma lista de widgets se chama ListView. Veja a sua documentação.

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

Observe que ListView possui alguns construtores.

CLASSES

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

Constructors

`ListView([Key? key, Axis scrollDirection = Axis.vertical, bool reverse = false, ScrollController? controller, bool? primary, ScrollPhysics? physics, bool shrinkWrap = false, EdgeInsetsGeometry? padding, double? itemExtent, Widget? prototypeItem, bool addAutomaticKeepAlives = true, bool addRepaintBoundaries = true, bool addSemanticIndexes = true, double? cacheExtent, List<Widget> children = const <Widget>[], int? semanticChildCount, DragStartBehavior dragStartBehavior = DragStartBehavior.start, ScrollViewKeyboardDismissBehavior keyboardDismissBehavior = ScrollViewKeyboardDismissBehavior.manual, String? restorationId, Clip clipBehavior = Clip.hardEdge])`  
Creates a scrollable, linear array of widgets from an explicit List.

`ListView.builder([Key? key, Axis scrollDirection = Axis.vertical, bool reverse = false, ScrollController? controller, bool? primary, ScrollPhysics? physics, bool shrinkWrap = false, EdgeInsetsGeometry? padding, double? itemExtent, Widget? prototypeItem, required NullableIndexedWidgetBuilder itemBuilder, ChildIndexGetter? findChildIndexCallback, int? itemCount, bool addAutomaticKeepAlives = true, bool addRepaintBoundaries = true, bool addSemanticIndexes = true, double? cacheExtent, int? semanticChildCount, DragStartBehavior dragStartBehavior = DragStartBehavior.start, ScrollViewKeyboardDismissBehavior keyboardDismissBehavior = ScrollViewKeyboardDismissBehavior.manual, String? restorationId, Clip clipBehavior = Clip.hardEdge])`  
Creates a scrollable, linear array of widgets that are created on demand.

`ListView.custom([Key? key, Axis scrollDirection = Axis.vertical, bool reverse = false, ScrollController? controller, bool? primary, ScrollPhysics? physics, bool shrinkWrap = false, EdgeInsetsGeometry? padding, double? itemExtent, Widget? prototypeItem, required SilverChildDelegate childrenDelegate, double? cacheExtent, int? semanticChildCount, DragStartBehavior dragStartBehavior = DragStartBehavior.start, ScrollViewKeyboardDismissBehavior keyboardDismissBehavior = ScrollViewKeyboardDismissBehavior.manual, String? restorationId, Clip clipBehavior = Clip.hardEdge])`  
Creates a scrollable, linear array of widgets with a custom child model.

Observe a descrição daquele que aparece primeiro na lista.

**Creates a scrollable, linear array of widgets from an explicit List.**

Já o segundo, chamado **builder**, é descrito assim.

**Creates a scrollable, linear array of widgets that are created on demand.**

O texto “on demand” faz toda a diferença. Se temos uma lista pequena, digamos com 5 elementos, o primeiro construtor resolve bem o problema. Ele constrói os 5 Widgets de para exibição de cada elemento de uma vez. Porém, se tivermos 5 milhões de itens para exibir, não podemos construir um Widget para cada item assim que a lista for construída. A melhor abordagem é fazer a construção **sob demanda**, conforme o usuário navega na lista. Por isso, vamos utilizar o construtor chamado **builder**.

No arquivo **image\_list.dart**, vamos construir e devolver um ListView a partir do método build.

Os parâmetros que especificamos são os seguintes:

- **itemCount**: número de elementos na lista.
- **itemBuilder**: uma função que explica a forma como cada item da lista deve ser exibido. Ela recebe um objeto do tipo **BuildContext**, chamado **context**, e um objeto do tipo **int**, chamado **index**. Ela devolve um Widget que exibe o item na posição **index** da lista de imagens.

#### Exercício.

1. Faça o método `build` de `ImageList` devolver um `ListView`, construído pelo construtor chamado `builder`.
2. Entregue os dois parâmetros a ele: `itemCount` e `itemBuilder`.
  - `itemCount` deve ser o número de imagens existentes na lista.
  - `itemBuilder` deve ser uma função que recebe um `BuildContext` e um `int`.
  - Faça a função associada ao parâmetro `itemBuilder` devolver um Widget textual que exibe a representação textual do `ImageModel` contido na posição `index`.

Veja como fica.

```
//arquivo image_list.dart
import 'package:flutter/material.dart';
import '../models/image_model.dart';

class ImageList extends StatelessWidget {
  final List<ImageModel> imagens;

  ImageList(this.imagens);

  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: imagens.length,
      itemBuilder: (BuildContext context, int index) {
        return Text(imagens[index].toString());
      },
    );
  }
}
```

Teste a aplicação clicando algumas vezes no FloatingActionButton. Veja o resultado esperado.



**Exercício.** Observe que, a cada clique, a aplicação exibe o mesmo texto e a mesma URL. Faça o ajuste necessário para que ela exiba um novo item a cada clique. Lembre-se de usar o número de imagens, variável de estado da classe AppState.

Veja como fica.

```

class AppState extends State<App> {
    //começamos de 1 pois não há página 0 na Pexels
    int numeroImagens = 1;
    List<ImageModel> imagens = [];
    void obterImagem() async {
        var url = Uri.https(
            'api.pexels.com',
            '/v1/search',
            //número da página requisitada igual ao valor existente em
            numeroImagens
            {'query': 'people', 'page': '$numeroImagens', 'per_page': '1'},
        );
        var req = http.Request('get', url);
        req.headers.addAll(
        {
            'Authorization':
            'chave',
        },
    );
    final result = await req.send();
    if (result.statusCode == 200) {
        final response = await http.Response.fromStream(result);
        var decodedJSON = json.decode(response.body);
        var imagem = ImageModel.fromJSON(decodedJSON);
        setState(() {
            //incremento a cada clique
            numeroImagens++;
            imagens.add(imagem);
        });
    }
    else {
        print('Falhou!');
    }
}

```

**(Exibindo imagens com o Widget Image)** Há um widget chamado Image capaz de fazer a exibição de imagens. Veja a sua documentação.

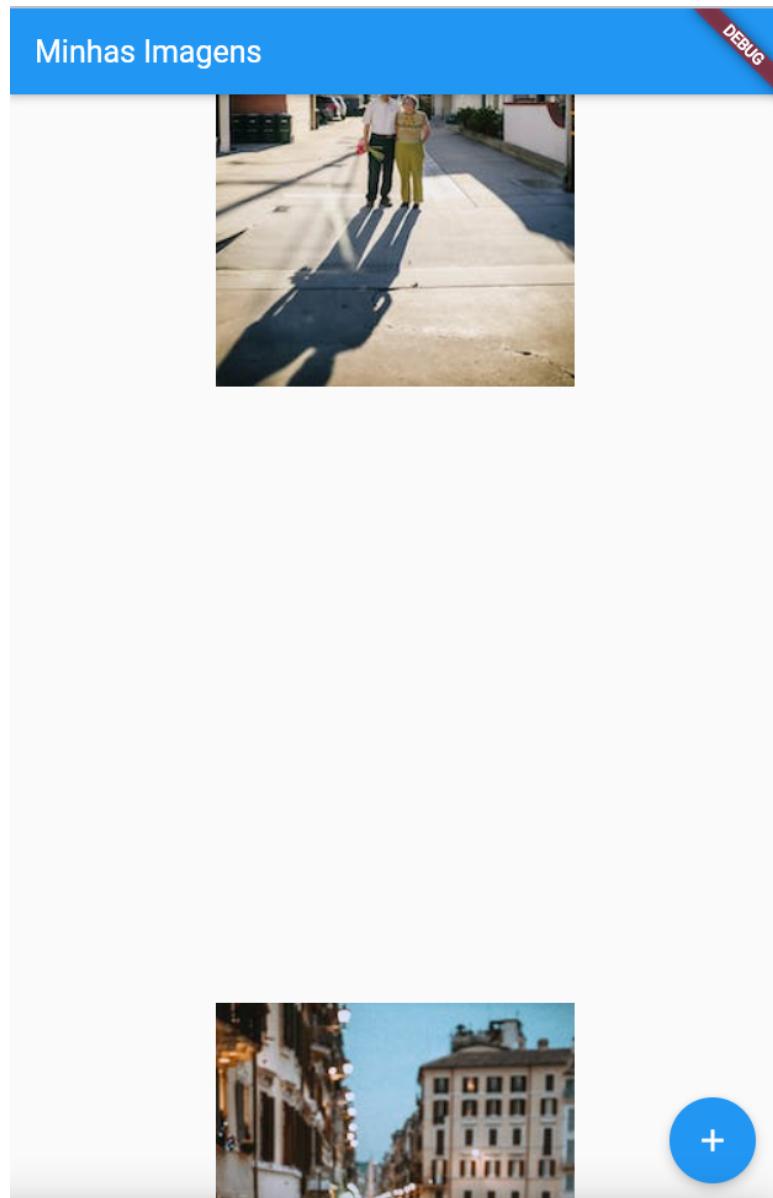
<https://api.flutter.dev/flutter/widgets/Image-class.html>

**Exercício.** Encontre o construtor apropriado para exibir uma imagem vinda da Internet. Lembre-se de utilizar a propriedade url do ImageModel.

Veja como fica. Estamos no arquivo **image\_list.dart**.

```
...
@Override
Widget build(BuildContext context) {
  return ListView.builder(
    itemCount: imagens.length,
    itemBuilder: (BuildContext context, int index) {
      return Image.network(imagens[index].url);
    },
  );
}
...
```

Teste novamente a aplicação e veja o resultado, após clicar algumas vezes no FloatingActionButton.



Observe que as imagens aparecem mas elas estão distantes umas das outras. Há alguns ajustes de posicionamento, borda, margem, padding etc a serem realizados.

**Exercício.** Faça ajustes na aplicação para que as imagens fiquem “próximas” umas das outras. Também não permita que elas fiquem “coladas” às bordas da tela do dispositivo. Para isso, estude sobre leiautes.

<https://docs.flutter.dev/ui/layout>

Estude também sobre bordas.

<https://docs.flutter.dev/ui/layout#container>

Exiba a descrição de cada imagem logo abaixo dela.

Acima da lista de imagens, exiba o texto: Você tem  $n$  imagens, sendo  $n$  o número de imagens atual.

Troque o leiaute da aplicação. Passe a exibir duas imagens por linha.

### ***Referências***

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

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