Criando um app com Flutter

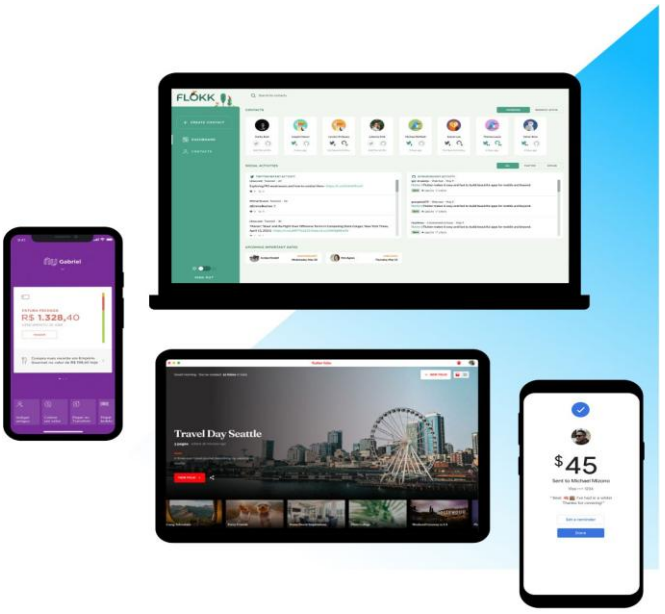
Sumário

Capítulo I

O que é Flutter? e para que serve?

Flutter é um framework de desenvolvimento de interface de usuário, de código aberto, criado pela empresa Google em 2015, baseado na linguagem de programação Dart, que possibilita a criação de aplicativos compilados nativamente, para os sistemas operacionais Android, iOS, Windows, Mac, Linux e, Fuchsia e Web.

O Flutter é um framework[1] voltado inicialmente para o desenvolvimento de aplicativos Android e IOs, porém com um tempo ele foi se avançando para o desenvolvimento Web, Linux, Windows e Mac. Você consegue fazer aplicativos para iOS e Android usando uma mesma base de código onde, na hora de compilar, ele transforma o código em uma versão nativa de cada plataforma, o que agiliza a abertura e o desempenho do aplicativo. O código Flutter é compilado para código de máquina ARM ou Intel, bem como JavaScript, para desempenho rápido em qualquer dispositivo.



Google, acessado em 29-08-2022, disponível em: https://flutter.dev/

O Flutter utiliza uma linguagem também criada pelo próprio Google, chamada Dart., sendo essa linguagem compatível com a orientação a objetos e programação funcional (o que diminui a curva de aprendizado). Apesar de ser nova para muitos, o Dart é uma linguagem simples e fácil de se aprender. Mesmo que você tenha somente o mínimo conhecimento de programação, verá que, com pouco tempo de estudo, você já se sentirá confortável com a linguagem. O Dart é uma linguagem de programação fortemente tipada inicialmente criada pela Google em 2011. A missão inicial do Dart era substituir o JavaScript para desenvolvimento de scripts em páginas web. Porém, com a evolução da linguagem e com o passar dos anos, ela hoje pode ser considerada uma linguagem multi-paradigma, embora a linguagem apresente fortes estruturas típicas de linguagens orientadas a objetos. O Dart possui uma sintaxe com estilo baseado no C. Isso faz com que sua sintaxe seja muito

[1] Framework é um facilitador no desenvolvimento de diversas aplicações e, sem dúvida, sua utilização poupa tempo e custos para quem utiliza, pois de forma mais básica, é um conjunto de bibliotecas utilizadas para criar uma base, onde as aplicações são construídas, um otimizador de recursos. Possui como principal objetivo resolver problemas recorrentes com uma abordagem mais genérica. Ele permite ao desenvolvedor focar nos “problemas” da aplicação, não na arquitetura e configurações.

similar à linguagens atualmente populares, como Java e C#. Porém, o Dart tenta reduzir um pouco os ruídos característicos de linguagens baseadas no C.
Um simples *"Hello World"* em Dart poderia ser escrito da seguinte forma:

```
main() {
  print('Hello World!');
}
```

Dart também possui algumas características de linguagens funcionais. Isso fica claro no exemplo abaixo, onde uma sequência de Fibonacci é gerada.

```
int fib(int n) => (n > 2) ? (fib(n - 1) + fib(n - 2)) : 1;

void main() {
  print('fib(20) = ${fib(20)}');
}
```

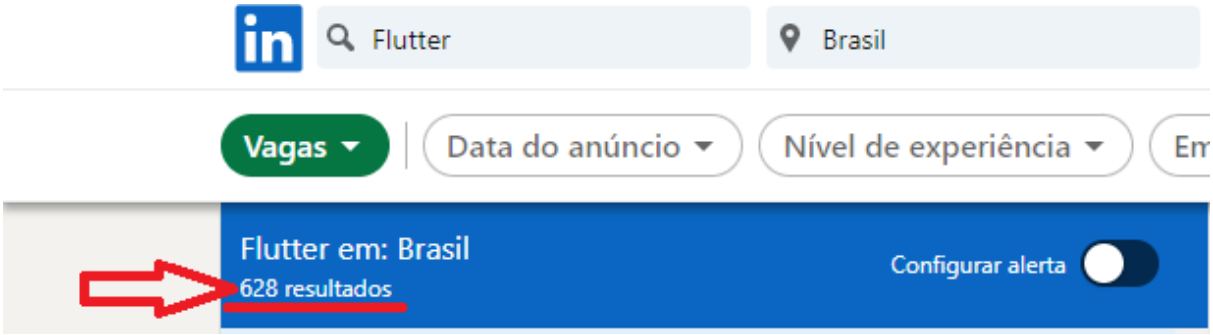
É importante saber que todos os componentes do Flutter são widgets. Sendo assim, um label, um campo de entrada de texto e até mesmo o processo de detecção de um gesto na interface são tratados como widgets dentro do Flutter. Uma aplicação Flutter, no final, é uma árvore hierárquica e coordenada destes widgets. Escreveremos nosso código na linguagem **Dart**, própria do Flutter, e aprenderemos que ele costuma trabalhar, em sua estrutura, com **Widgets** - sejam eles **Stateful Widgets** ou **Stateless Widgets**, conceitos que abordaremos em detalhes no futuro, mas por enquanto podemos entender Widgets como um componente visual para definir a interface de um aplicativo. Existem diversos Widgets já prontos que nos permitem adicionar componentes visuais à aplicação, e aprenderemos a utilizá-los de modo a evitar que tenhamos que fazer isso manualmente. Na documentação, podemos encontrar até mesmo um catálogo para conferir esses componentes visuais, utilizando a princípio o *Material Components*, regras de design comuns para aplicativos Android e que nos ajdarão a entregar um projeto com visual interessante, incluindo animações e transições.

Mercado de trabalho com Flutter

As vagas no Flutter se encontram em altos picos de contratação de profissionais que dominam o Framework, pois o mesmo está virando tendência das grandes empresas no momento de escolher a tecnologia utilizada no desenvolvimento de suas aplicações. As grandes empresas já conhecidas por todos da área tech utilizam o Flutter em suas organizações como Google, Nubank, Ebay, BMW e outras gigantes do mercado.

Google, acessado em 29/08/2022, disponivel em: https://flutter.dev/

Em uma pesquisa feita no Linkedin no dia 29-08-2022 foi encontrado de forma rápida cerca de 628 vagas postadas na plataforma que foram catalogadas com a label de Flutter.



Linkedin, acesso em 29/08/2022, disponivel em: https://www.linkedin.com/

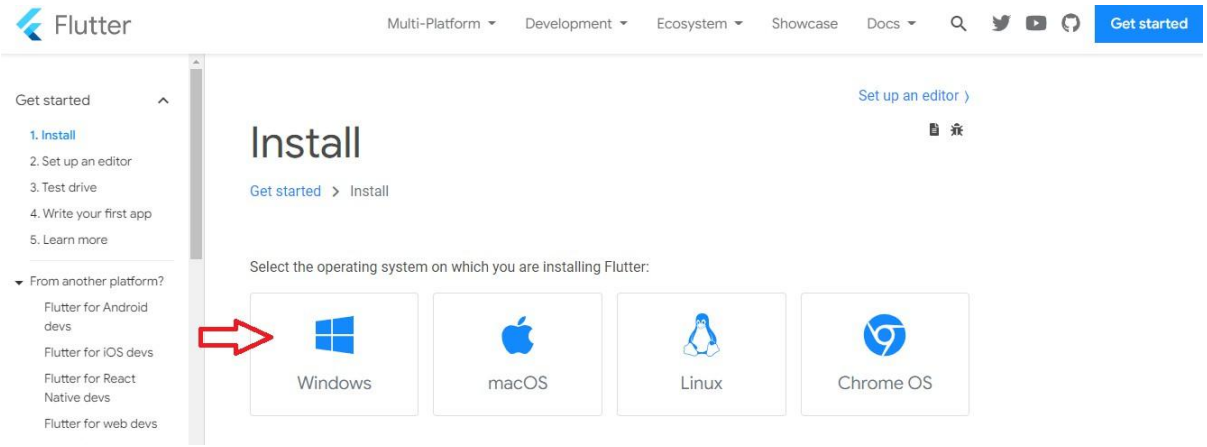
O Framework tem tudo para se transformar em uma excelente opção de estudo para desenvolvedores que buscam uma alternativa para suas aplicações.
Como foi possível perceber, o Flutter é uma tecnologia que traz uma série de features importantes que conseguem diminuir o tempo de trabalho, promover mais agilidade para os Devs bem como reduzir custos, além de tantas e tantas outras vantagens.

Agora que sabemos um pouco mais sobre o Framework Flutter, vamos ver como ele funciona na prática em um ambiente Windows.

Preparando o ambiente de trabalho para o Flutter

Baixando e instalando o Flutter

Para começar a desenvolver com Flutter são necessários alguns passos. Inicialmente será preciso fazer a instalação do Flutter, basta acessar o link: https://docs.flutter.dev/get-started/install e fazer o download dos arquivos zipados do Flutter.

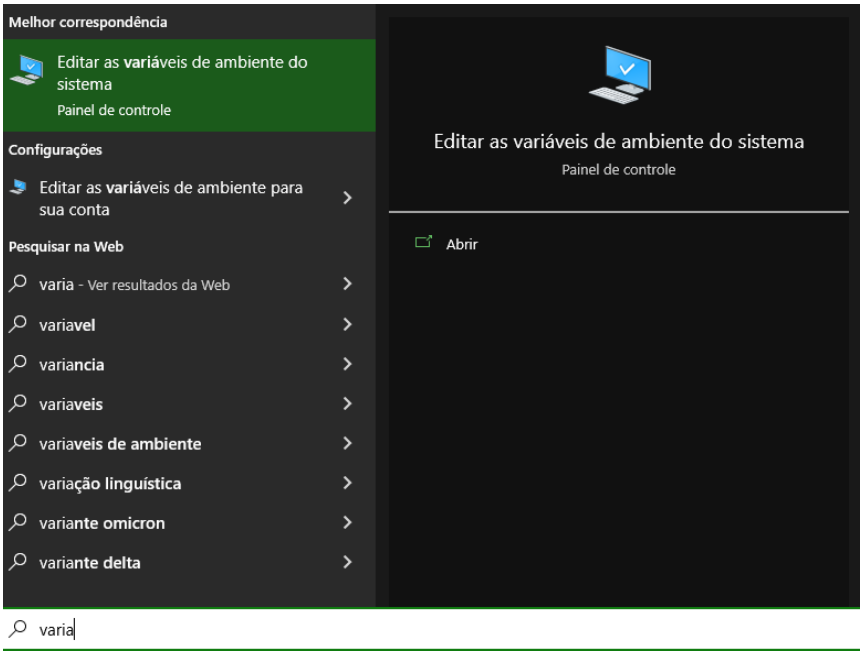


Flutter, disponivel em: https://docs.flutter.dev

Após feito o download do arquivo Flutter zipado, é necessário fazer a configuração das variáveis de ambiente da sua máquina, é bem simples e rápido.
Primeiro passo é criar uma nova pasta com nome “src” no seu disco C:/ da sua máquina, e em seguida podemos extrair os arquivos baixados para essa pasta criada.
O segundo passo é configurar nossas variáveis de ambiente.

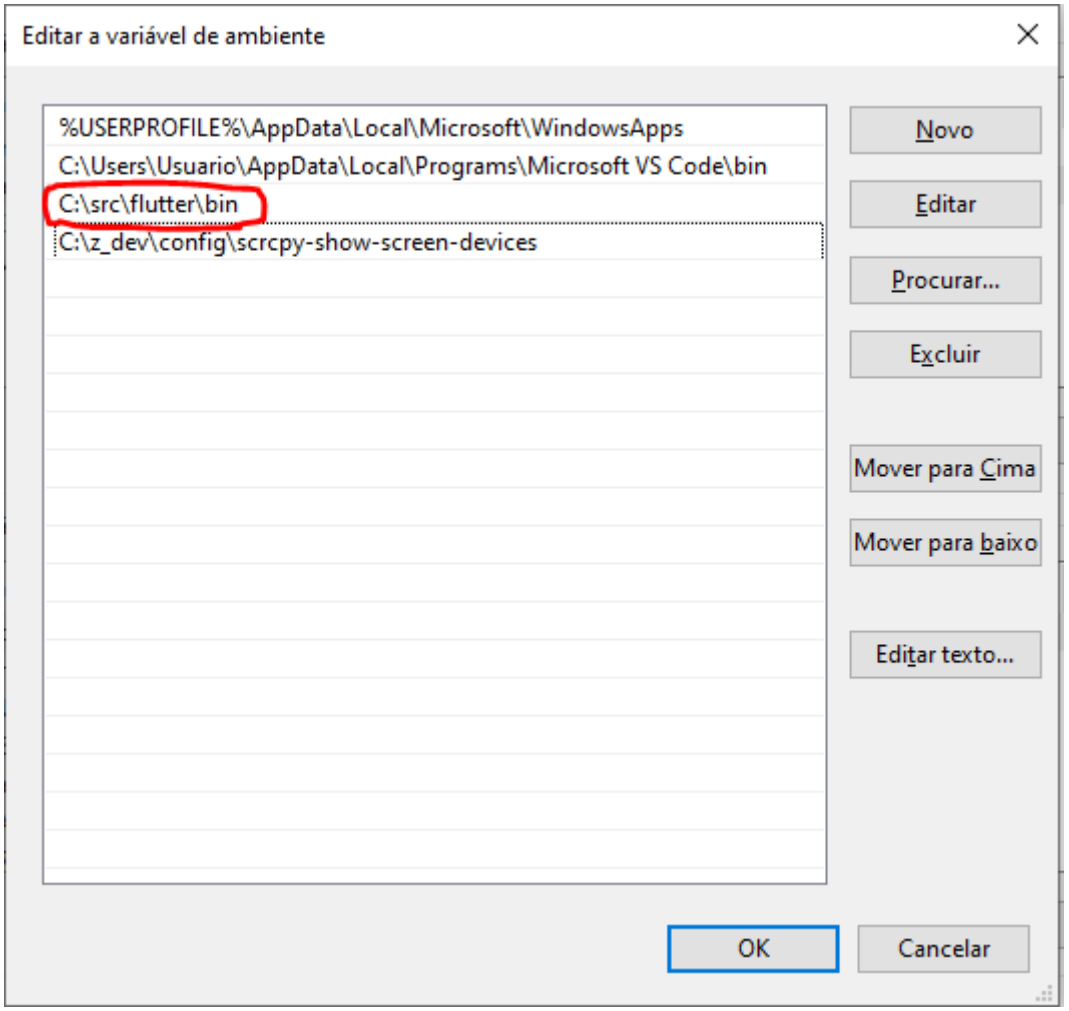
Configurando as variáveis de ambiente

para isso vamos entrar na pasta criada e encontrar nosso diretório **/bin**. Podemos copiar esse path do diretório **C:\src\flutter\bin** e copiá-lo. Após copiar esse caminho, vamos colá-lo nas variáveis de ambiente, para isso vamos clicar na tecla Win do teclado e digitar **Variáveis de Ambiente.**



E vamos clicar no botão "Variáveis de ambiente”, após isso vai abrir uma nova janela e então na aba “Variáveis de usuário” vamos clicar em **Path** e vamos clicar no botão “Editar”, feito isso, irá abrir uma nova

janela, clicamos no botão “Novo” e em seguidas vamos colar o caminho da pasta bin/ que copiamos.



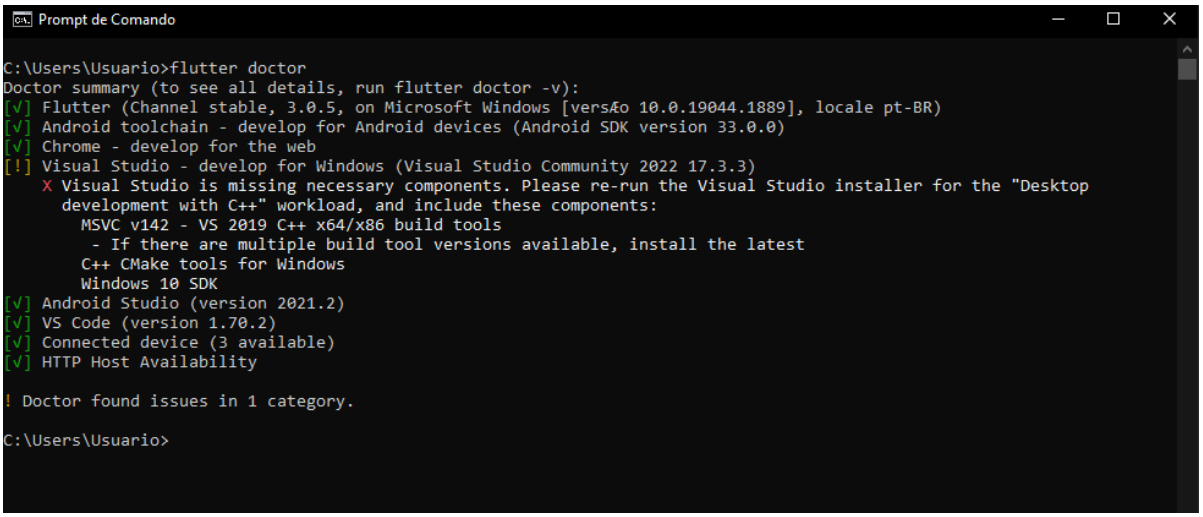
Por fim, podemos clicar em “OK” e continuar e fechar as janelas. Pronto, finalizamos nossa instalação do Flutter.

É preciso, também, de um dispositivo mobile onde seja possível executar o aplicativo (pode ser o emulador do Android e/ou iOS ou um dispositivo físico) e o SDK da plataforma que estejamos desenvolvendo (o SDK do Android pode ser obtido através da instalação do Android Studio).

Verificando dependências do Flutter

Para verificar se está tudo ok, em seu CMD podemos fazer o seguinte comando “flutter doctor” e em seguida pressionar a tecla ENTER. O “flutter doctor” é o comando responsável por verificar se existem dependências do Flutter a serem instaladas. Além disso, ele retorna um relatório sobre o status da instalação contendo as dependências que faltam, como instalá-las, problemas encontrados e como resolvê-los.

Após esse comando será exibida uma mensagem de aviso, que por hora podemos ignorá-la e seguir com o desenvolvimento de nosso aplicativo, mas é recomendado que todas as dependências do Flutter sejam instaladas, elas são fáceis de resolver e o próprio Flutter nos ensina como resolvê-las com o comando “flutter doctor”.



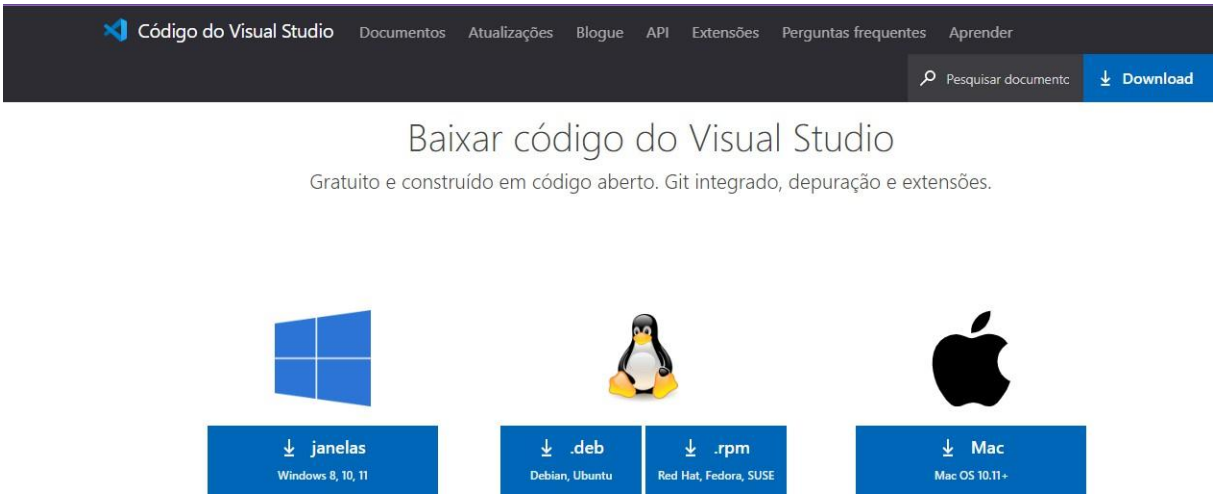
Para o desenvolvimento do nosso aplicativo, é recomendável utilizar algum editor de código (IDE[2]) de sua preferência.

Os editores de código mais conhecidos para o desenvolvimento mobile são o Android Studio e o Visual Studio Code, ambos tem suas vantagens e suas desvantagens, logo, o Android Studio possui mais recursos e por sua consome um pouco mais de recursos da máquina, já o Visual Studio Code é mais leve e podemos configurá-lo de forma rápida para o desenvolvimento com Flutter, logo, utilizaremos o Visual Studio Code por ser mais leve e otimizado para quem possui poucos recursos de hardware.

[2] Uma IDE é um ambiente de desenvolvimento integrado (Integrated Development Environment). É um software que vai nos auxiliar no desenvolvimento de nossos aplicativos.

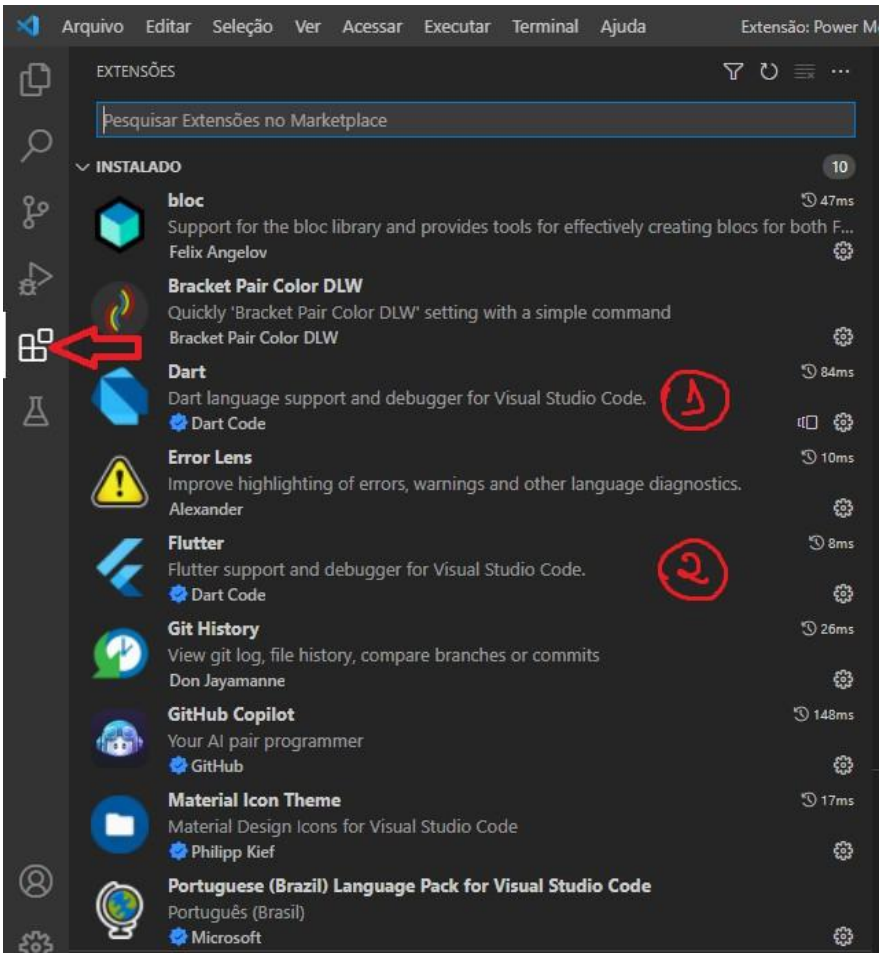
Instalando a IDE Visual Studio Code

Para fazer a instalação do Visual Studio Code, basta acessar o link https://code.visualstudio.com/download e fazer o download do executável do seu sistema operacional Windows, como na imagem abaixo:



Código do Visual Studio, disponivel em: https://code.visualstudio.com/download

Após a instalação do Visual Studio é necessário instalar algumas extensões neste programa. Com ele aberto, podemos clicar no menu lateral “Extensões” e pesquisar por essas duas extensões exibidas na imagem abaixo, Dart e Flutter.



Finalizada a instalação dessas extensões, estamos prontos para iniciar de fato o desenvolvimento do nosso primeiro aplicativo feito com Flutter.

Criando nosso primeiro aplicativo com Flutter

Para a criação do novo projeto Flutter no Visual Studio Code, podemos utilizar o comando "flutter create <nome do app>".

```
TERMINAL   CONSOLE DE DEPURAÇÃO   PROBLEMAS   SAÍDA

PS C:\z_dev> flutter create my_app
Creating project my_app...
Running "flutter pub get" in my_app...                          3,0s
Wrote 127 files.

All done!
In order to run your application, type:

  $ cd my_app
  $ flutter run

Your application code is in my_app\lib\main.dart.
```

Pronto! Criamos nosso primeiro app com Flutter.

Após a criação do novo projeto, podemos ver que foi criado um novo aplicativo de demonstração disponibilizado pelo próprio Flutter, que se trata de um contador de cliques, para executar esse aplicativo, podemos digitar no terminal o comando “flutter run” que será gerado o build do projeto que será executado no emulador ou no seu dispositivo físico.

Entendendo a estrutura do projeto

Uma parte fundamental do desenvolvimento de software é entender a estrutura de pastas do seu projeto, podemos observar que foi criado várias pastas em nosso aplicativo e uma delas é a **lib/. É n**a nossa pasta lib/ onde vão ficar os arquivos que terão nossos códigos.

Nas pastas android/, ios/, linux/, windows/, macos/ e web, podemos ver todos os nossos arquivos que são relacionados ao Android

Nativo, IOs Nativo e etc.. Podemos ver também que tem alguns arquivos fora das pastas, como por exemplo o pubspec.yaml que é muito importante para o desenvolvimento com Flutter.

O arquivo pubspec.yaml

O arquivo pubspec.yaml especifica as dependências que o projeto requer, como pacotes específicos (e suas versões), fontes ou arquivos de imagem. Ele também especifica outros requisitos, como dependências de pacotes do desenvolvedor (como pacotes de teste ou simulação) ou restrições específicas na versão do SDK do Flutter. É nele que vamos deixar nossas dependências de packages, mas o que seria packages no Flutter? Packages podemos ter em mente que são trechos de códigos reutilizáveis que podemos utilizar em nossos projetos para facilitar e nos auxiliar durante o desenvolvimento. A plataforma mais conhecida onde podemos encontrar os packages é a pub.dev. Logo mais vamos utilizá-la em nosso projeto, mas por enquanto é importante apenas sabermos que ela existe.
Em nosso arquivo pubspec.yaml, temos um trecho de dependencies e dev_dependencies, em dependencies serão as dependências em que o projeto irá utilizar durante a execução, e nossas dev_dependencies, são as dependências do projeto somente durante o desenvolvimento do nosso aplicativo.

```
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.2

dev_dependencies:
  flutter_test:
    sdk: flutter
  flutter_lints: ^2.0.0
```

Agora que entendemos um pouco mais sobre nosso arquivo pubspec.yaml, podemos voltar para nosso arquivo main.dart.

O arquivo main.dart

Quando abrimos a nossa pasta lib/ podemos ver que nela existe um arquivo chamado **main.dart**, é nesse arquivo que temos o código principal do nosso aplicativo.
A função **main()** é a primeira função que é chamada quando executamos nosso aplicativo, e podemos ver que ela faz a chamada de outra função, a runApp, ela é a responsável por construir nossos Widgets/Telas que no caso é o nosso **MyApp**();

```
void main() {
  runApp(const MyApp());
}
```

Em nosso MyApp podemos ver que ele é uma classe que herda os comportamentos de um widget StatelessWidget, mas antes temos que saber o que é um widget **Stateless e Stateful.**

Stateless e StateFul

A tradução seca de stateless é “sem estado”, e stateful é “com estado”, mas afinal, o que é um estado? Basicamente, um estado é uma informação ou grupo de informações que são alteradas durante o tempo de execução do aplicativo. Imagine que você está entrando no site do Youtube, podemos ver que ele inicialmente dá um efeito de que está carregando os vídeos e depois os vídeos aparecem magicamente, certo? Podemos fazer essa relação que o momento em que está carregando os vídeos é um estado de loading (carregando) e quando os vídeos são carregados é um estado de loaded (carregado). Ou seja, estados são “momentos” que nosso aplicativo pode possuir. Então podemos relacionar os widgets stateless como widgets que não possuem estado, ou seja, eles não trocam de estado na tela. E os widget stateful? São os widgets que possuem estado, que podem trocar de estado durante a execução do aplicativo.

O Widget mais importante - MaterialApp

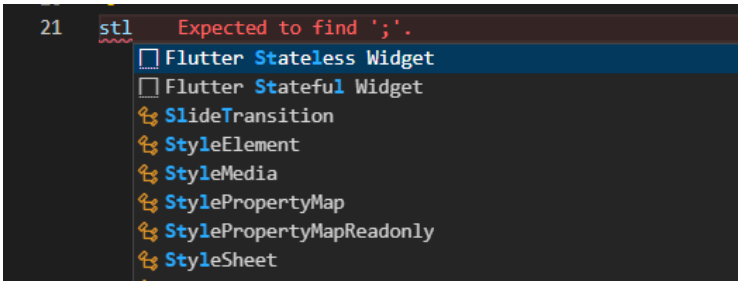
Em nosso MyApp() temos um método build() com uma anotação de @override e com o retorno temos o nosso MaterialApp que é um dos widgets mais importantes da nossa árvore de widgets. Em nosso MaterialApp podemos notar que ele possui 2 parâmetros importantes que são o **title, theme** e o **home**. O title é basicamente o título do nosso aplicativo principal, nele podemos definir como queremos que o usuário veja o nome do nosso aplicativo. E o theme é o tema do nosso aplicativo, nele podemos definir as cores primárias, secundárias, cores de botões, e input de textos e diversas outras coisas. O home é o widget que vai ser desenhado na tela do nosso dispositivo, nele vamos colocar o widget que foi criado MyHomePage().

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

Para fins didáticos vamos excluir todo o widget MyHomePage(). Agora vamos criar o nosso primeiro widget!

Criando nossa primeira tela - Contador

Após apagarmos o widget **MyHomePage()** vamos criar nosso widget, podemos chamá-lo de HomePage(). Para isso podemos utilizar o atalho da nossa IDE digitando apenas **stf** e clicando no autocomplete fornecido. E então definimos o nome do nosso widget como **HomePage().**



Pronto, nosso widget foi criado, porém está aparecendo alguns erros em nossa IDE, como apagamos o widget MyHomePage, ele não existe mais então temos que substituí-lo pelo widget que criamos o HomePage e retirar tudo de dentro dos parênteses, e então nossa class MyApp ficará desta forma:

```
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const HomePage(),
    );
  }
}
```

Voltando para nosso HomePage, vamos recriar a tela de um contador de cliques, então inicialmente vamos criar uma variável do tipo **int,** podemos chamá-la de contador e iniciá-la com o valor 0. `int contador = 0;` Essa variável vai ser responsável por armazenar os valores dos cliques que vamos fazer na tela.

Após criar nossa variável contador, vamos dar início a criação da nossa tela. O primeiro passo para criar uma tela em Flutter é imaginar quais serão os widgets que terão na tela, para nosso contador vamos ter os widgets de texto para exibir a quantidade de cliques, e um botão para que possamos clicar nele e aumentar a quantidade de cliques.

Então no **return** do build(), iremos utilizar o widget **Scaffold()** que é responsável por fazer a estrutura da nossa tela, ele funciona como o esqueleto para nossos widgets.
Em nosso Scaffold temos um atributo chamado **body**, o body é responsável por armazenar todo o corpo da nossa tela, e neles iremos definir o widget **Center(),** esse widget tem como principal função centralizar todo o seu conteúdo no centro da tela, no widget Center temos acesso a um parâmetro child: que vamos defini-lo como um **Column().**

O widget Column funciona como uma coluna onde ele vai estruturar os widgets filhos dele um em cima do outro como uma pilha, para isso vamos utilizar o atributo do Column chamado **children: [];** que funciona como um array de widgets. Ainda em cima do atributo children:[] vamos definir um comportamento para nossa coluna que é a de centralizar seus filho no centro da coluna, para isso vamos utilizar o parâmetro mainAxisAlignment: e vamos definir o valor MainAxisAlignment.center.

E dentro no nosso children vamos colocar nossos widgets que foram definidos, que será um texto e um botão, para o texto vamos utilizar o widget **Text** e definir qual vai ser nossa mensagem e usaremos o $(cifrão) para referenciar nossa variável.

```
return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text('Foi clicado $contador vezes'),
```

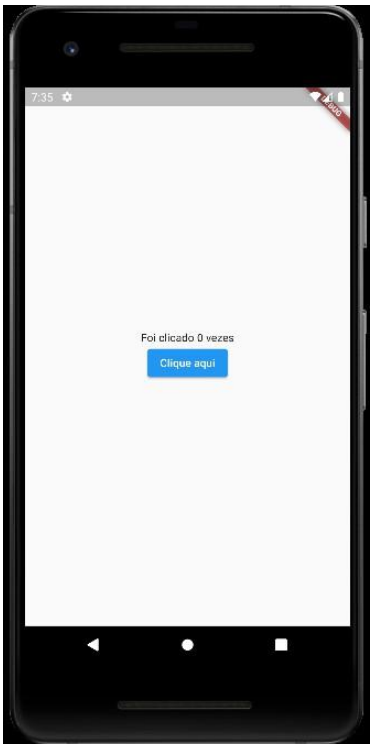
Agora vamos adicionar nosso botão, podemos utilizar o widget **ElevatedButton,** esse widget tem 2 atributos um **child** e um **onPressed**, para nosso child vamos definir um Text, pois queremos que dentro do botão tenha um texto escrito “Clique aqui” e dentro do nosso onPressed vamos definir uma função que será executada quando alguém clicar no botão.

Voltando a nossa lógica do botão, queremos que o valor da variável contador seja incrementada ao clicar no botão, então vamos fazer nossa variável incremento da seguinte forma contador++; Porém para que o valor do nosso contador seja atualizado na tela, temos que utilizar a função **setState(){}** que vai atualizar o valor da nossa tela e ser feito o build do widget novamente. Após isso, finalizamos a nossa tela e nosso app de contar cliques.

Você pode verificar o nosso widget como ficou abaixo:

```dart
class _HomePageState extends State<HomePage> {
  int contador = 0;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text('Foi clicado $contador vezes'),
            ElevatedButton(
              onPressed: () {
                setState(() {
                  contador++;
                });
              },
              child: const Text('Clique aqui'),
            ),
          ],
        ),
      ),
    );
  }
}
```

Podemos ir em nosso terminal e testar nosso widget, no terminal digitamos o seguinte comando: **flutter run** ou então clicar na tecla F5, que será executada no nosso aplicativo.
E ele ficará parecido com o seguinte app.



É muito importante que você conheça alguns widgets básicos como Column, Row, Text, ElevatedButton e Image, pois esses widgets são os mais utilizados durante o desenvolvimento de qualquer aplicativo.

Capítulo II

Consumindo uma API

Neste capítulo vamos criar um aplicativo que vai consumir a api de cep disponibilizada pela ViaCEP que pode ser encontrada no site https://viacep.com.br/ws/01001000/json/
Primeiramente vamos entender o que é uma **API**, a definição de API é uma interface de programação de aplicação (API) é um código que permite que dois programas de softwares se comuniquem entre si. A API determina a maneira correta para que um desenvolvedor escreva um programa que solicite os serviços de um sistema operacional (**SO**) ou de outra aplicação.
Uma API normalmente tem 4 funções básicas que são: **GET**, **POST**, **PUT** e **DELETE.**

GET:

O método HTTP GET é usado para **ler** (ou recuperar) uma representação de um recurso. No caminho "feliz" (ou sem erro), GET retorna uma representação em XML ou **JSON** e um código de resposta HTTP de 200 (OK). Em um caso de erro, na maioria das vezes ele retorna um 404 (NÃO ENCONTRADO) ou 400 (PEDIDO RUIM).

POST:

O verbo POST é usado com mais frequência para **criar** novos recursos. Em particular, é usado para criar objetos no nosso banco de dados.

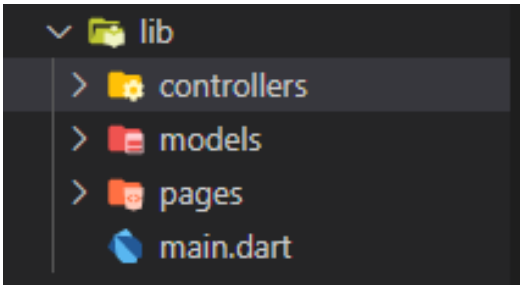
PUT:

PUT é usado com mais frequência para recursos de **atualização**, PUT para um URI de recurso conhecido com o corpo da solicitação contendo a representação recém-atualizada do recurso original.

DELETE:

O DELETE é muito fácil de entender. Ele é usado para **excluir** um recurso identificado por um URI.Na exclusão bem-sucedida, retorne o status HTTP 200 (OK).

Agora que entendemos melhor o que é uma API, vamos dar início ao desenvolvimento do nosso app. Para isso podemos reutilizar nosso app existente ou criar outro com o comando flutter create cep_api. Após feito esse passo, vamos criar três pastas em nosso projeto, todas dentro da pasta lib/, serão elas: pages/, models/, controllers/, vamos seguir uma arquitetura simples, a **MVC**. Essa arquitetura é dividida em 3 camadas, uma camada para os **Models**, uma camada para nossas **Views** e a última camada de **Controllers**. Após criar essas pastas, teremos uma estrutura de pastas como na imagem.



Criando a camada Models

O primeiro passo do nosso desenvolvimento é implementarmos a camada de Models. Vamos criar um arquivo dentro da pasta models/ com nome de cep_model.dart. E então vamos entrar no site que nos disponibiliza a API e vamos copiar todo o json disponibilizado.

```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "complemento": "lado ímpar",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
  "ibge": "3550308",
  "gia": "1004",
  "ddd": "11",
  "siafi": "7107"
}
```

Agora para agilizar nosso processo vamos acessar um site que nos auxilia bastante na criação de models, é o JsonToDart, podemos acessá-lo seguindo este link:https://javiercbk.github.io/json_to_dart/, e em seguida vamos colar o json que copiamos anteriormente.

Vamos seguir alguns passos nesse site:

1. Vamos colar nosso json dentro do campo respectivo.
2. Vamos escolher o nome do nosso model, no exemplo a seguir o nome escolhido foi CepModel.
3. Vamos clicar em "**Generate Dart**"
4. E por último vamos clicar em **"Copy Dart code to clipboard"**

Por fim, vamos colar todo esse código gerado dentro do nosso arquivo criado, **cep_model.dart.** Pronto, assim finalizamos nossa camada do model.

O segundo passo é criar a camada de **Controllers**.

Criando a camada de Controllers

Para a criação da camada de controllers, primeiramente temos que instalar um package que vai nos ajudar a fazer o consumo da api, o package que vamos utilizar é o **http**, para isso, vamos na plataforma já explicada no anterioriormente a pub.dev e pesquisar por **http**. Caso queira, pode acessar o link diretamente https://pub.dev/packages/http, e então vamos na aba **Installing** e copiamos o comando: **flutter pub add http** e então colocamos eles em nosso terminal e apertamos a tecla ENTER, pronto, o Flutter vai ficar responsável por adicionar esse package no nosso arquivo **pubspec.yaml** automaticamente, caso queira verificar se foi instalado corretamente podemos ir no nosso arquivo e procura-lo em dependencies.

```
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.2
  http: ^0.13.5 #<<<<<<<<<< AQUI ESTÁ NOSSO PACKAGE BAIXADO
```

Voltando para a implementação da camada de Controllers, vamos criar um arquivo chamado **cep_controller.dart** e em seguida vamos criar nossa classe de controller e criar uma variável do tipo model, e fazer a importação do nosso package http, como no exemplo:

```
import 'package:http/http.dart' as http;
import 'package:my_app/models/cep_model.dart';

class CepController {
  final _httpClient = http.Client();
  CepModel cep = CepModel();
```

Após isso vamos criar nossa função que vai ser responsável por fazer o chamado da nossa API e retornar nosso objeto do tipo CepModel();.
Podemos chamá-la de carregarCEP e vamos receber por parâmetro o CEP que será digitado pelo usuário na tela que ainda vamos criar. Então ficará assim:

```
carregarCEP(String input) async {
}
```

Podemos notar a palavra reservada **async**, ela serve para nos informar que essa função vai fazer alguma operação que poderá ter ou não um retorno instantâneo pois ela irá depender da nossa API que está na internet e pode ou não responder rapidamente.
Em seguida vamos criar uma variável que vai receber todo o retorno da nossa api, lembrando que nossa api vai nos retornar um json, na mesma estrutura do json já mostrado anteriormente, podemos colocar o nome desta variável de response, pois ela vai ter a resposta da api. E então vamos fazer nossa requisição da api, para isso vamos fazer a chamada de uma função do nosso package http, que é o **GET,** este método espera como parâmetro uma URI que é o link da nossa API dada no site, após a criação da variável a mesma seria assim:

```
final response = await _httpClient.get(
      Uri.parse('https://viacep.com.br/ws/$input/json/'),
    );
```

Podemos observar uma nova palavra chave que é a **await** que significa “esperar”, ou seja, nosso aplicativo vai esperar que a requisição seja feita e que nos retorne. Dessa forma nossa variável response quando obtiver um retorno da API terá alguns atributos que serão interessantes como a **request** que vai ter nosso link da requisição, nosso **statusCode** que terá o status da nossa API (A API retorna de uma requisição alguns códigos que poderão ser analisados de forma mais completa nesse link:https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status) e também tem o **body** que será a resposta da API.
Prosseguindo com o desenvolvimento, vamos fazer uma verificação no statusCode da

nossa resposta, é interessante para nosso app atualmente só o statusCode que seja igual 200, pois é esse status que diz que tudo está “OK”. Vamos fazer a verificação com um if:

```
if (response.statusCode == 200) {
}
```

Como dito anteriormente nosso **response.body** contém toda a resposta da api, ou seja, dentro dela estará nosso json como resposta, se faz necessário a decodificação desse json e para isso poderemos utilizar uma função nativa do Flutter, a jsonDecode(), e passar como parâmetro o nosso response.body. Em seguida vamos passar esse json para ser transformado em nosso model, podemos utilizar nossa função criada no model que é a CepModel.fromJson(); e então retornar nosso model.

```
if (response.statusCode == 200) {
      final json = jsonDecode(response.body);
      cep = CepModel.fromJson(json);
      return cep;
    }
```

Pronto, temos aqui nosso caminho feliz, quando o statusCode for igual a 200, mas, e se for diferente de 200? É necessário fazer o tratamento deste erro. Então no else desse if podemos lançar uma Exception com uma mensagem para o usuario

```
else {
      throw Exception('Erro ao carregar o CEP');
      }
```

O código completo do nosso controller irá ficar exatamente assim:

```
import 'dart:convert';

import 'package:http/http.dart' as http;
import 'package:my_app/models/cep_model.dart';

class CepController {
  final _httpClient = http.Client();
  CepModel cep = CepModel();

  carregarCEP(String input) async {
    final response = await _httpClient.get(
      Uri.parse('https://viacep.com.br/ws/$input/json/'),
    );
    if (response.statusCode == 200) {
      final json = jsonDecode(response.body);
      cep = CepModel.fromJson(json);
      return cep;
    } else {
      throw Exception('Erro ao carregar o CEP');
    }
  }
}
```

Agora só falta a última parte, criar nossa camada que é a criação da nossa tela

Criação da camada View

Para a criação da nossa tela, vamos criar um arquivo dentro da pasta pages/, podemos chamá-lo de home_page.dart. Vamos criar como no último exercício um widget stateful, podemos utilizar o atalho **stf,** podemos renomear nosso widget como HomePage.

Novamente, vamos utilizar o widget Column() para que os widgets fiquem empilhados, e também utilizaremos novos widgets como o Padding e o TextFormField.
O **Padding** é responsável por adicionar um espaço em branco em nossa coluna, para que a mesma não fique tão encostada no final da tela.
O **TextFormField** é o widget que será o input da nossa tela que receberá o CEP que o usuário vai digitar. Nele temos acesso a um atributo que é o decoration, que podemos definir vários estilos para nosso input, mas nesse exemplo só iremos adicionar um **hintText**.
O **SizedBox** podemos utilizá-lo para que ele faça espaçamentos entre os widgets da nossa tela que também será utilizado para definir o tamanho da largura do nosso botão.
Se criarmos seguindo o exercício anterior e adicionando os novos widgets teremos nossa tela dessa forma:

```
return Scaffold(
      body: Center(
        child: Padding(
          padding: const EdgeInsets.all(20),
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              TextFormField(
                decoration: const InputDecoration(
                  hintText: 'Digite um cep',
                ),
              ),
              const SizedBox(height: 20),
              SizedBox(
                width: 300,
                child: ElevatedButton(
                  onPressed: () {}
                  child: const Text('Buscar CEP'),
                ),
              ),
              const SizedBox(height: 40),
```

Após isso vamos criar 3 variáveis que são importantes para nosso app, a primeira é uma variável do tipo bool, uma variável do tipo do nosso modelo que foi criado, e uma variável que vai armazenar o valor digitado do usuário no input.

```
bool isLoading = true;
CepModel? cepModel;
final textController = TextEditingController();
```

E então no nosso widget TextFormField vamos adicionar um atributo chamado controller: e adicionar nossa variável textController. Nosso TextFormField ficará assim:

```
TextFormField(
   controller: textController,
   decoration: const InputDecoration(
     hintText: 'Digite um cep',
   ),
),
```

E no onPressed do nosso ElevatedButton iremos vamos a chamada da nossa função criada no controller, ficando assim:

```
SizedBox(
  width: MediaQuery.of(context).size.width,
  child: ElevatedButton(
    onPressed: () async {
      final CepModel cepModel =
          await CepController().carregarCEP(textController.text);
      setState(() {
        this.cepModel = cepModel;
        isLoading = false;
      });
    },
    child: const Text('Buscar CEP'),
  ), // ElevatedButton
), // SizedBox
```

Em seguida vamos fazer os widgets que vão exibir nosso CEP que vem da API, vamos criar uma verificação na nossa variável isLoading, se ela for verdadeira iremos retornar apenas um Container() vazio, e se ela for falsa vamos retornar alguns widgets e passar como parâmetro nossa variável model.

```
const SizedBox(height: 40),
if (isLoading)
  Container()
else
  Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      Text('CEP: ${cepModel!.cep}'),
      Text('Logradouro: ${cepModel!.logradouro}'),
      Text('Complemento: ${cepModel!.complemento}'),
      Text('Bairro: ${cepModel!.bairro}'),
      Text('Localidade: ${cepModel!.localidade}'),
      Text('UF: ${cepModel!.uf}'),
      Text('IBGE: ${cepModel!.ibge}'),
      Text('GIA: ${cepModel!.gia}'),
      Text('DDD: ${cepModel!.ddd}'),
      Text('SIAFI: ${cepModel!.siafi}'),
    ],
  ), // Column
```

Pronto, para executar o código do nosso aplicativo, podemos ir no terminal e digitar o seguinte comando “flutter run” ou simplesmente apertar F5. E assim finalizamos nosso aplicativo que vai consumir a API de CEP disponibilizada pela ViaCEP. Caso queira baixar esse projeto pronto para estudos, será disponibilizado no link abaixo: Google Drive.