

Introdução ao Flutter



Definições iniciais

- Flutter: é o kit de ferramentas de interface do usuário do Google para criar aplicativos compilados nativamente para dispositivos móveis, web e desktop a partir de uma única base de código.
- É gratuito e possui código fonte aberto.
- Possui o chamado “hot reload” que permite a atualização visual da aplicação sem a necessidade de uma recompilação completa da aplicação. No “hot reload” ocorre a injeção de arquivos de código-fonte atualizados na Dart Virtual Machine (VM) em tempo de execução. Depois que a VM atualiza as classes com as novas versões de campos e funções, a estrutura do Flutter reconstrói automaticamente a árvore de widgets, permitindo visualizar rapidamente os efeitos de suas alterações.



Linguagem de Programação

- Flutter utiliza a linguagem de programação Dart. A linguagem Dart é uma linguagem C-style, ou seja, é parecida com C, C++, Java, Javascript ou C#.
- A linguagem Dart é tipada, mas a definição de tipos é opcional.
- Possui generics, mixins, funções de alta ordem entre os recursos.
- Pode ser compilada e interpretada.
- Ela é uma linguagem lançada em 2011. Seu objetivo inicial era substituir a linguagem Javascript, que é uma linguagem que continua evoluindo mas tem sua origem no século passado. Entretanto naquele momento não obteve sucesso. Com o Flutter, Dart volta como uma linguagem multi paradigma apesar de seu cerne ser uma linguagem orientada a objetos.
- Acredita-se que Dart não evoluiu em 2011 por conta da visão da comunidade de desenvolvimento. A ideia de fragmentar plataformas web não foi bem vista, além disso, a Google também tinha a fama de abandonar produtores de software de maneira abrupta o que gerou certa desconfiança.



Instalação

- A instalação do Flutter é relativamente simples e pode ser encontrada em:
<https://flutter.dev/docs/get-started/install>
- Uma das vantagens na instalação do Flutter é utilização do Flutter Doctor. Essa ferramenta faz uma varredura na plataforma de desenvolvimento verificando se há alguma pendência para o desenvolvimento de aplicações em Flutter.

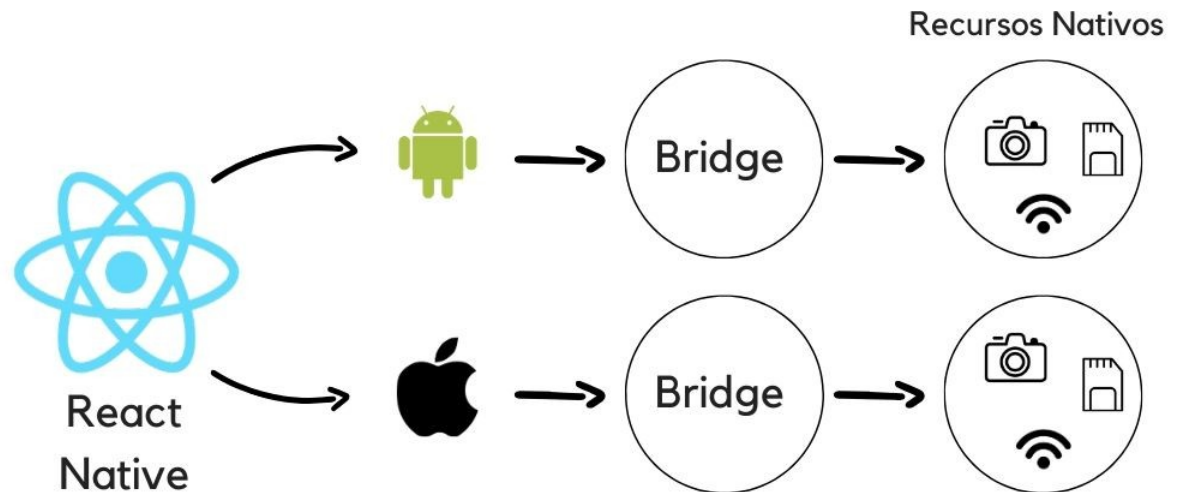
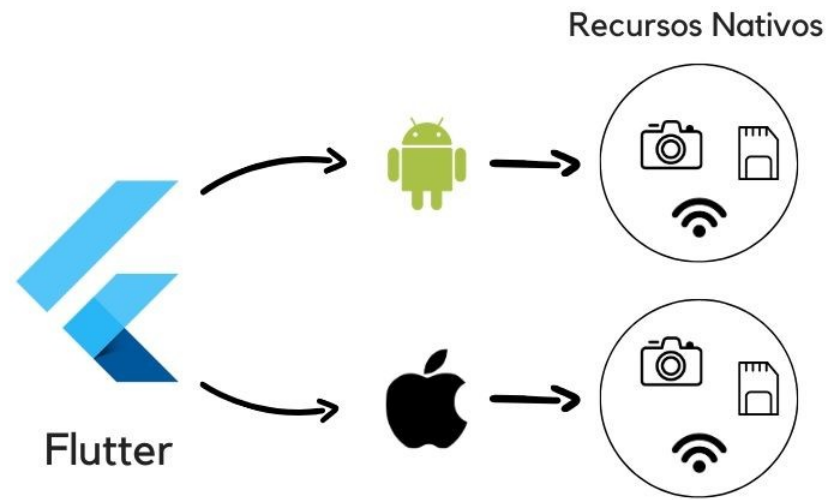


Ionic x React Native x Flutter

- **Ionic:** é um excelente framework para desenvolvimento de aplicações híbridas. Seu foco é “usar a plataforma”, ou seja, usar padrões abertos (HTML, CSS, Javascript) da web sempre que possível. O problema é que, na prática, ele é uma WebView com grande dependência de plugins do Cordova. Como resultado final temos uma aplicação cuja performance é inferior a aplicações nativas.
- **React Native:** desenvolvido pela equipe do Facebook, também é um ótimo framework. Aqui temos uma melhoria do desempenho, visto que o React Native não usa WebViews. Possui mais tempo de mercado que o Flutter e é bastante utilizado, mas o desempenho é inferior ao do Flutter por conta da necessidade de uma “Bridge” (ponte) entre o aplicativo e os recursos nativos. A figura no próximo slide, obtida no site <https://www.treinaweb.com.br/blog/o-que-e-flutter/> em 01/06/2020 mostra essa diferença:



React Native X Flutter



Xamarin x Flutter

- A Xamarin foi originalmente fundada em 2011 por engenheiros que criaram o Mono, uma implementação multiplataforma de Xamarin.Android e Xamarin.iOS. O Xamarin foi a primeira estrutura de desenvolvimento de aplicativos móveis para várias plataformas, na qual as empresas podem criar aplicativos para Android e iOS que parecem quase nativos. Mais tarde, em 2016, a Microsoft adquiriu a Xamarin e tornou-se parte do Microsoft Visual Studio.
- O Xamarin utiliza a linguagem C# e é um framework bastante moderno/maduro e com muitos recursos. Possui forte vínculo com o Visual Studio, sendo praticamente a única IDE de desenvolvimento em Xamarin.
- O Xamarin.Forms é uma ferramenta do Xamarin que permite o reaproveitamento de componentes visuais para várias arquiteturas (iOS, Android, Windows Phone, etc).
- Os componentes visuais do Flutter são mais ricos em recursos e animações que a maioria dos concorrentes, incluindo Xamarin.
- Em termos de desempenho, nas pesquisas que fiz, não é possível afirmar quem é mais rápido. De fato, ambos possuem um desempenho praticamente de código totalmente nativo (Android – Java/Kotlin e iOS – Objective C e Swift).



Delphi x Flutter

- O Delphi é mais uma das opções para quem deseja desenvolver cross-platform. O framework Firemonkey promete uma experiência de usuário tão boa quanto a de aplicativos nativos.
- Normalmente opta por essa opção é quem já desenvolve em Delphi e deseja navegar pela programação para dispositivos móveis.
- A quantidade de material e desenvolvedores é bem menor que a de tecnologias como React Native ou Xamarin.
- Enquanto a Microsoft está por trás do Xamarin, o Facebook do React Native e o Google do Flutter, a Embarcadero é responsável pelo Delphi. Sem dúvida nenhuma uma empresa muito menos conhecida que esses três gigantes.
- Enfim, se o programador já utiliza Delphi em seu trabalho pode ser interessante usar essa ferramenta para desenvolver mobile, mas é importante ficar atento ao tamanho desse mercado e das possíveis limitações dessa tecnologia.



Visão pessoal

- O melhor desempenho possível será sempre o código 100% nativo. Quanto mais próximo disso melhor o desempenho.
- Aprender várias tecnologias ao mesmo tempo é trabalhoso e difícil, nesse contexto, o desenvolvimento híbrido ou cross-plataform possibilitarão maior facilidade de se manter atualizado.
- Utilizar ferramentas ligadas a tecnologias web sempre darão a vantagem de aprender “uma coisa só”.
- Já programei em Android Java, Android Kotlin, iOS Objective C, Kivy (cross-plataform em Python), Xamarin e Flutter. Sempre me senti mais a vontade em tecnologias da Google.
- O Objective C é uma linguagem muito “desagradável de programar” e o XCode mistura muita configuração visual com codificação. Na prática é fácil esquecer algo e demorar a encontrar o erro.



Visão pessoal

- Kivy é uma ferramenta de baixa maturidade e pouca integração. Na prática falta muito ainda para ser uma tecnologia realmente prática para desenvolver para dispositivos móveis.
- Xamarin é uma tecnologia muito boa e é muito interessante desenvolver aplicativos nela (foi minha primeira opção para sair do Objective C). Consegui até fazer um curso da própria Microsoft para uso da tecnologia, o problema, para mim, é que eu senti que mais estavam tentando me vender serviços da Microsoft Azure que desenvolver mesmo nessa tecnologia. Também tive alguns problemas com atualizações de pacotes NuGet e não fiquei tão entusiasmado em continuar investindo nessa tecnologia.
- Android Java sempre foi meu “carro chefe” pela meu conhecimento em Java e pelos cursos que pude fazer. Me sinto bem a vontade em trabalhar nessa tecnologia.
- Android Kotlin foi a última tecnologia que estudei antes do Flutter. Realmente Kotlin é uma linguagem fantástica e sua interoperabilidade com Java permite a evolução de forma gradativa. Mas ou menos como aconteceu com C e C++.



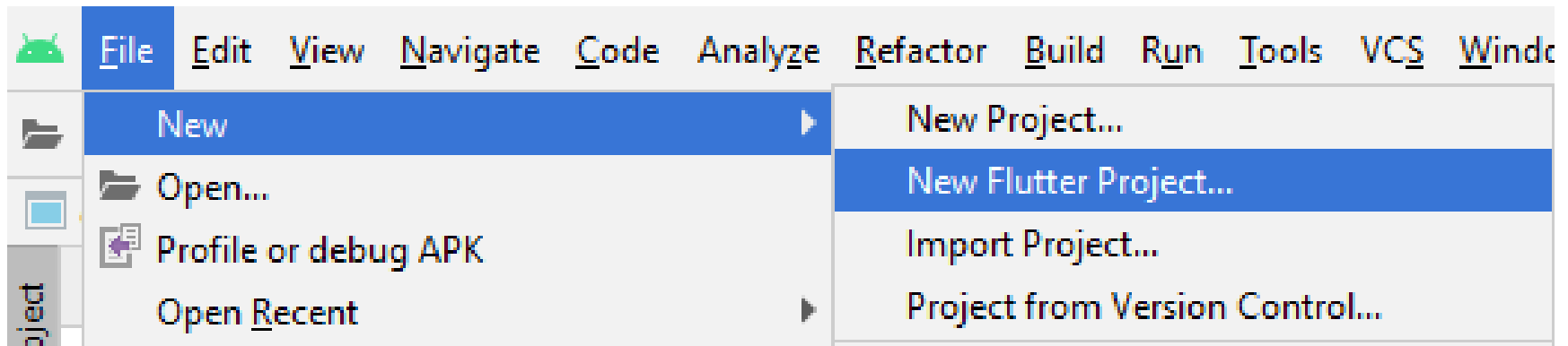
Visão pessoal

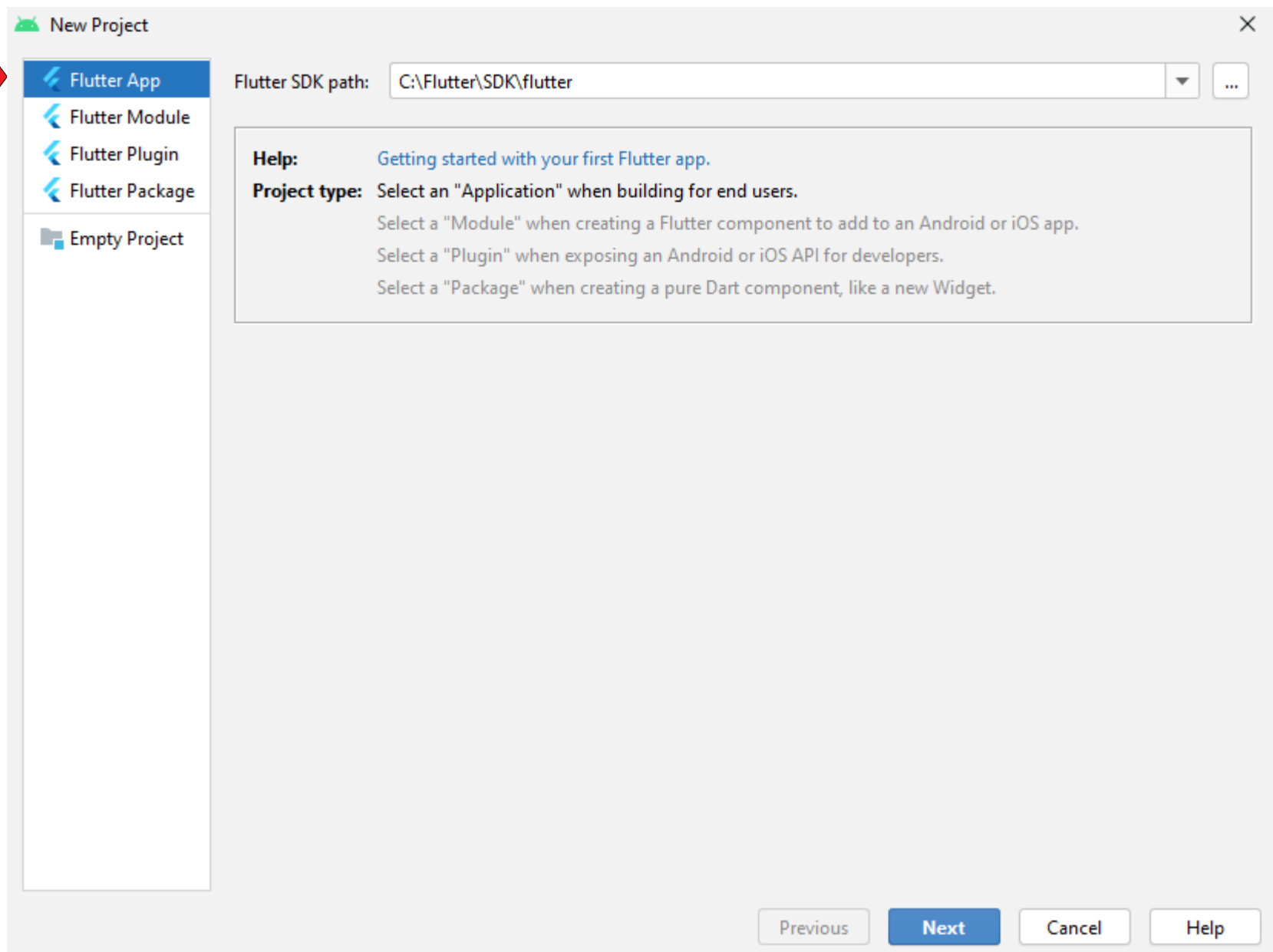
- Aproveitei a quarentena, por conta do Corona Virus, para aprender Flutter. Achei essa tecnologia muito simples de se compreender, pois tudo são widgets. Dart também é bem parecido com Kotlin e Java o que não gerou qualquer dificuldade.
- Em Flutter o encadeamento de objetos em sua criação acabam dificultando um pouco a visualização e o acerto do código (confesso que senti um pouco de falta dos XML do Java/Kotlin, algo que também existe no Xamarim).
- Por outro lado os widgets normalmente são fáceis de configurar e muito bonitos, visualmente falando. Além de serem leves (o desempenho do uso de mapas, por exemplo, foi similar ao Android Java).
- Os componentes já estarem prontos para o Material Design é um facilitador e tanto. O menu lateral (NavigationDrawer no Android Java) é muito fácil de fazer em Flutter.
- Pretendo analisar ainda o uso do Flutter na Web. Para cross-plataform é a minha opção preferida pois possui ótimo desempenho, recursos modernos e em expansão, além da bela aparência dos widgets.

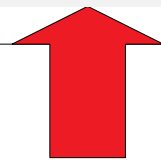
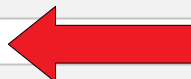
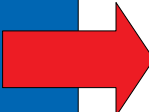


Um “Olá mundo” no Flutter

- Vamos criar um projeto inicial em Flutter e buscar entender seu código.
- Tudo começa em File→New→New Flutter Project ...







New Project

Project name: primeiro_programa

Project location: C:\Users\Giovany\AndroidStudioProjects\primeiro_programa

Description: Meu primeiro programa

Organization: com.minha.empresa

Android language: ☐ Java ☒ Kotlin

iOS language: ☐ Objective-C ☒ Swift

Platforms: ☒ Android ☒ iOS ☐ Linux ☐ MacOS ☐ Web ☐ Windows

Platform availability might depend on your Flutter SDK channel, and which desktop platforms have been enabled.

Additional desktop platforms can be enabled by, for example, running "flutter config --enable-linux-desktop" on the command line.

When created, the new project will run on the selected platforms (others can be added later).

☐ Create project offline

More Settings

Previous Finish Cancel Help



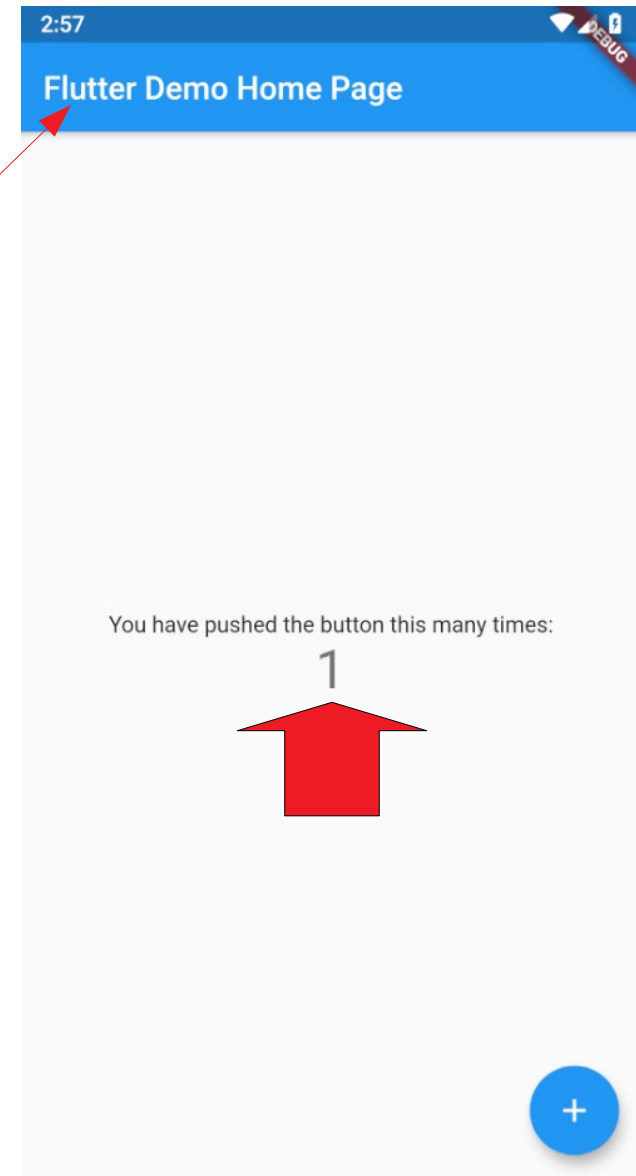
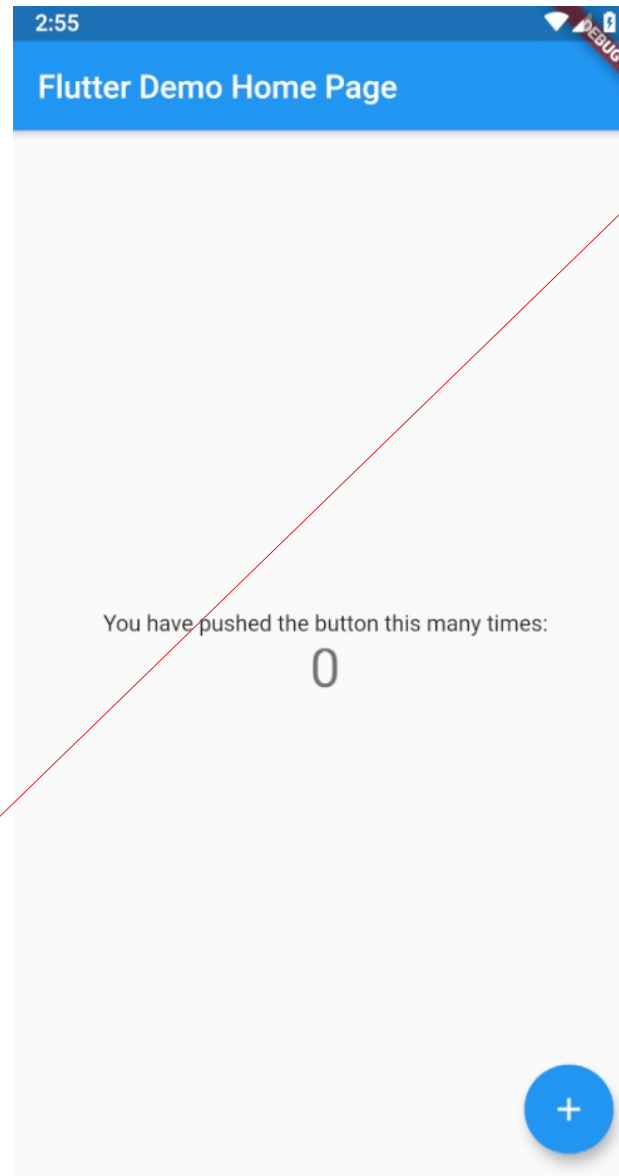
O exemplo

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

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

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,
      ), // ThemeData
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    ); // MaterialApp
  }
}
```



Entendendo o início do exemplo

- Tudo começa no **main**, assim como em outras linguagens.
- **MyApp** é a classe que começa a execução. Ela é **StatelessWidget**, ou seja, não mudará de estado (nenhum atributo ou objeto seu alterará seu valor, por exemplo).
- A palavra **const** serve para criar “objetos constantes”, em outras palavras, ela indica ao compilador que esse objeto NUNCA será alterado, logo se for criado outro objeto igual, ele pode fazer apenas referência ao criado anteriormente. Imagine uma parede de Quadrados exatamente iguais, se criarmos esse Quadrado com const, podemos apenas referenciar o mesmo objeto quantas vezes forem necessárias ao invés de criarmos muitas instâncias de Quadrado para preencher a parede.
- O **MaterialApp** é um widget que é criado uma vez por aplicação e serve para setar configurações iniciais e chamar a primeira tela em si. Nesse exemplo ele setou **primarySwatch** para **Colors.blue**. Experimente mudar para **Colors.green** para ver o efeito na aplicação (não é necessário rodar tudo novamente, basta salvar que o Flutter fará o “hot reload”).
- **MyHomePage** é a primeira tela criada e recebeu o texto *'Flutter Demo Home Page'* no parâmetro title.

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

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

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,
      ), // ThemeData
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    ); // MaterialApp
  }
}
```



Entendendo o inicio do exemplo

- **MyHomePage** é um **StatefulWidget** isso significa que ele suporta mudança de estado.
- Na sequência vemos a definição do construtor e o atributo **title** é inicializado (title é um parâmetro nomeado para torná-lo obrigatório utiliza-se required, já a '?' indica que key pode ser nulo) .
- Por fim é criado um objeto do tipo **_MyHomePageState**. Esse objeto trabalhará com o **MyHomePage** para tratar a questão da mudança de estado.

```
class MyHomePage extends StatefulWidget {  
  const MyHomePage({Key? key, required this.title}) : super(key: key);  
  
  final String title;  
  
  @override  
  State<MyHomePage> createState() => _MyHomePageState();  
}
```



setState()

```
class _MyHomePageState extends State<MyHomePage> {  
  int _counter = 0;  
  
  void _incrementCounter() {  
    setState(() {  
      _counter++;  
    });  
  }  
}
```

- A classe **_MyHomePageState** possui um atributo interno **_counter** que servirá para acumular os cliques no botão flutuante na base da tela.
- O método **_incrementCounter()** serve para essa finalidade. Esse método chama o **setState()** que incrementará o valor do **_counter**. Só é possível utilizar o **setState()** porque a **MyHomePage** é um **StatefulWidget**.
- Quando o **setState()** é chamado o Widget da tela é redesenhado no método **build** (vamos vê-lo no próximo slide).
- É importante reforçar que o atributo interno **_counter** é utilizado no momento do desenho da tela para mostrar o texto da quantidade de cliques.



build

- O método **build** é o responsável por desenhar a tela.
- Logo no início temos um **Scaffold**. Esse Widget é usado como base para telas no Flutter, ou seja, normalmente toda tela começa de um **Scaffold**.
- A **AppBar** é a barra superior que aparece na aplicação. Ela tem como título o valor passado no construtor de **MyHomePage**.
- Como estamos em **_MyHomePageState** para acessar o atributo **title** precisamos fazer referência a palavra **widget**.
- O **body** é o corpo da tela. Nele colocarmos o widget **Center** para centralizar a coluna. E dentro da coluna colocamos dois textos: Um fixo e um que mostra o valor de **_counter**.
- Por fim, temos o **FloatingActionButton** que quando clicado irá chamar o método **_incrementCounter** (aquele que irá mudar o valor de **_counter**) e acionará novamente o **build** para reconstruir toda a tela.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
    ), // AppBar
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          const Text(
            'You have pushed the button this many times:',
          ), // Text
          Text(
            '$_counter',
            style: Theme.of(context).textTheme.headline4,
          ), // Text
        ], // <Widget>[]
      ), // Column
    ), // Center
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ), // FloatingActionButton
  ); // Scaffold
}
```



Dúvidas?

