

SUMÁRIO

1. O QUE É FLUTTER	1
2. FUNDAMENTOS DO FLUTTER.....	1
2.1. A DOCUMENTAÇÃO DO FLUTTER.....	1
2.2. CONFIGURAÇÃO DO AMBIENTE FLUTTER	2
2.2.1. INSTALAÇÃO DO SDK DO FLUTTER NO WINDOWS	2
2.2.2. A ESCOLHA DA IDE PARA DESENVOLVIMENTO	2
2.2.3. A ESCOLHA DO SOFTWARE PARA EMULAÇÃO DO DISPOSITIVO	3
2.2.4. EMULAÇÃO DO DISPOSITIVO ANDROID: CRIANDO UM AVD	3
2.3. FLUTTER É ORIENTADO A COMPONENT..WIDGET!.....	5
2.3.1. WIDGETS DE LAYOUT	6
2.3.2. WIDGETS DE INTERFACE E PACOTES READY-TO-USE	6
2.4. ÁRVORE OU HIERARQUIA DE WIDGETS	6
2.5. ESTADOS DOS WIDGETS	6
2.5.1. STATELESS WIDGET	7
2.5.2. STATEFUL WIDGET	7
2.5.3. ESTADO LOCAL E GLOBAL.....	7
2.6. NAVEGANDO ENTRE PÁGINAS: PILHAS DE TELAS.....	7
3. COMEÇANDO UM PROJETO FLUTTER	8
3.1. COMANDOS BÁSICOS – CMD/CLI	8
3.1.1. CRIAÇÃO DO PROJETO FLUTTER	8
3.1.2. INSTALAÇÃO DE PACOTES	8
3.1.3. ATUALIZAÇÃO DOS PACOTES	9
3.1.4. OUTROS COMANDOS	9
3.2. ARQUITETURA DE UM PROJETO FLUTTER	9
3.3. CÓDIGO BÁSICO DE UM PROJETO FLUTTER	10
3.4. EXECUTANDO O CÓDIGO NO ANDROID EMULADO	11
3.4.1. PRIMEIRA EXECUÇÃO	11
3.4.2. EXECUTANDO O CÓDIGO NO INTELLIJ IDEA.....	12
3.5. EXECUTANDO O CÓDIGO NO VISUAL STUDIO CODE	13
3.5.1. HOT RELOAD E HOT RESTART	13
FONTES	15

1. O QUE É FLUTTER

Criado pelo Google, o Flutter é um Framework para o desenvolvimento frontend de aplicativos mobile para Android e IOS. Além disso, ele utiliza a linguagem de programação Dart e um modelo declarativo de codificação.

Ao criar um aplicativo com o Flutter, seu código é compilado para a linguagem base do dispositivo, ou seja, as aplicações são realmente nativas e por isso conseguem acessar recursos do dispositivo sem a “ajuda” de terceiros e com o desempenho maior.

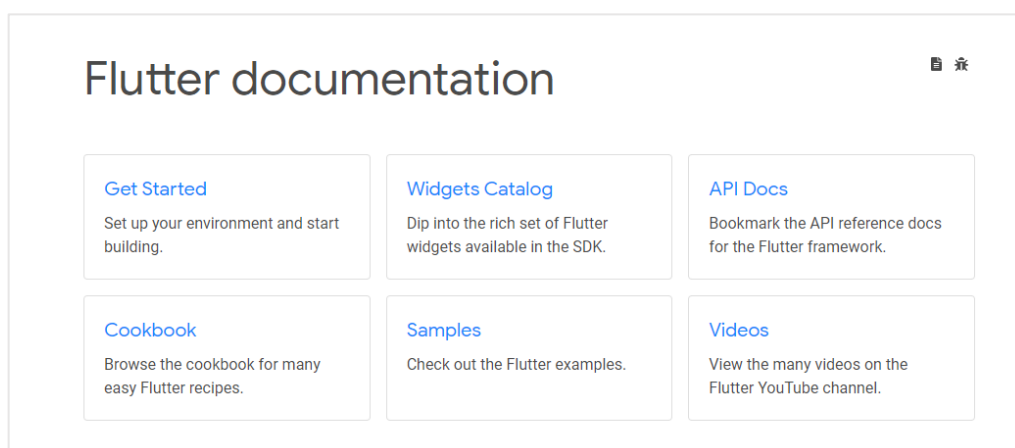
2. FUNDAMENTOS DO FLUTTER

2.1. A DOCUMENTAÇÃO DO FLUTTER

Para o desenvolvimento de aplicativos, a orientação da documentação das tecnologias é imprescindível, e neste caso, obviamente não é diferente.

Na verdade, dada a forma de fazer do Flutter, de uso constante de componentes visuais pré-moldados, a documentação, neste caso, é até mesmo mais frequentemente necessária que em outras situações de desenvolvimento, porque é nela que se encontram as classificações, as orientações, as descrições e a base do código desses componentes.

Pois bem: clicando [neste link](#), a página inicial da documentação será acessada. Tudo é relevante, mas existem algumas seções que auxiliam diretamente a prática de desenvolvimento. Por exemplo, o [Widget Catalog](#) é, como o próprio nome já diz, um **catalogo de Widgets** pré-moldados, disponíveis e prontos para uso. Também, o [Cookbook](#) é uma seção de “livro de receitas” para criar aplicativos, funcionalidades e mecanismos comuns. A seção [API Docs](#) tem a listagem de todas as bibliotecas nativas do SDK do Flutter.



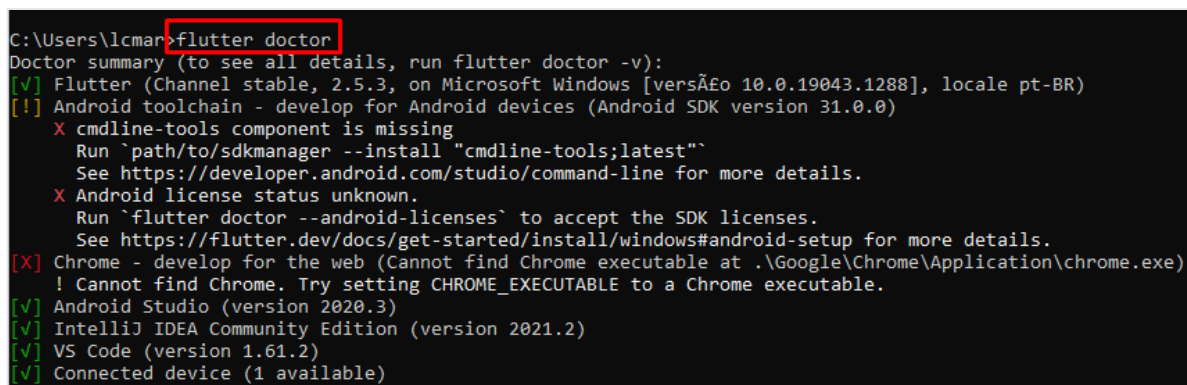
2.2. CONFIGURAÇÃO DO AMBIENTE FLUTTER

2.2.1. INSTALAÇÃO DO SDK DO FLUTTER NO WINDOWS

Para o desenvolvimento em Flutter ser possível, assim como com outras tecnologias, uma série de configurações prévias são necessárias. O primeiro passo é realizar o download e a instalação do “Software Development Kit” (SDK) do Flutter.

Para instalar no Windows, esses passos devem ser seguidos:

- A) Baixar e instalar o Git – clique [aqui](#);
- B) Baixar o SDK do Flutter, que é um arquivo compactado – clique [aqui](#);
- C) Extrair o SDK compactado para C:\flutter;
- D) Adicionar à variável de ambiente “path” o caminho C:\flutter\bin, para que seja possível o uso dos recursos CLI;
- E) Testar o Flutter CLI abrindo o console de comando e digitando, sem aspas, “flutter”;
- F) Digitar, também no console de comando, e sem aspas, “flutter doctor”, para diagnosticar a instalação do SDK.



```
C:\Users\lcmar>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[v] Flutter (Channel stable, 2.5.3, on Microsoft Windows [vers o 10.0.19043.1288], locale pt-BR)
[!] Android toolchain - develop for Android devices (Android SDK version 31.0.0)
    X cmdline-tools component is missing
      Run `path/to/sdkmanager --install "cmdline-tools;latest"`
      See https://developer.android.com/studio/command-line for more details.
    X Android license status unknown.
      Run `flutter doctor --android-licenses` to accept the SDK licenses.
      See https://flutter.dev/docs/get-started/install/windows#android-setup for more details.
[X] Chrome - develop for the web (Cannot find Chrome executable at .\Google\Chrome\Application\chrome.exe)
    ! Cannot find Chrome. Try setting CHROME_EXECUTABLE to a Chrome executable.
[v] Android Studio (version 2020.3)
[v] IntelliJ IDEA Community Edition (version 2021.2)
[v] VS Code (version 1.61.2)
[v] Connected device (1 available)
```

2.2.2. A ESCOLHA DA IDE PARA DESENVOLVIMENTO

Para desenvolver com Flutter é necessário a utilização de um ambiente integrado de desenvolvimento. Uma boa escolha de IDE é o [IntelliJ IDEA](#) da JetBrains, embora também seja possível com o [Visual Studio Code](#).

A diferença entre ambos, no entanto, é nítida com a experiência. A primeira ferramenta oferece mais recursos, e alguns muito relevantes, como o gerenciamento da árvore de componentes.

Embora citado por último, e já sendo assunto do próximo tópico, que é sobre as tecnologias possíveis para emulação do Android, o [Android Studio](#) é sempre citado como o **melhor ambiente de desenvolvimento** do mercado para Flutter.

2.2.3. A ESCOLHA DO SOFTWARE PARA EMULAÇÃO DO DISPOSITIVO

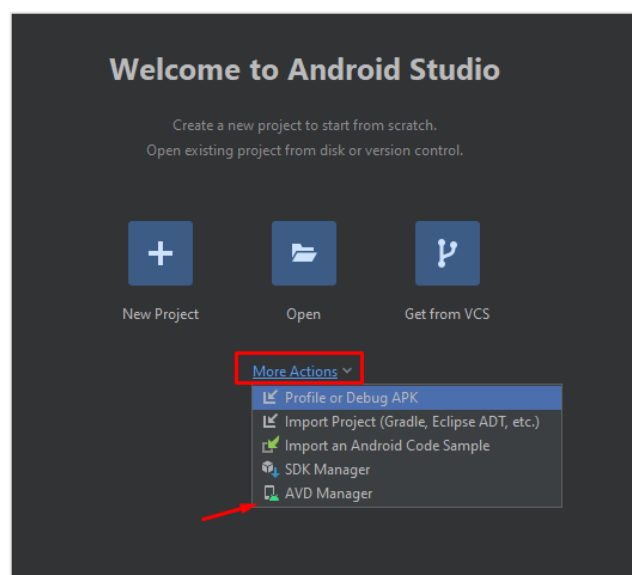
Para testar o código de um aplicativo Flutter, isto é, ver o seu resultado como programa, é necessário, para uma melhor experiência de aprendizado, ter um **emulador de dispositivo Android ou iOS**. Na ausência de um, o aplicativo é executado no navegador, ou até mesmo em um dispositivo físico conectado à máquina de desenvolvimento.

Pois bem: para este caso, de estudo da tecnologia, **será utilizado um emulador Android**. Para emular o sistema, existem duas opções alternativas de tecnologia, e que são utilizadas para esse fim: a IDE do [Android Studio](#), que será a alternativa escolhida, e o [Genymotion](#), que também é bastante utilizado.

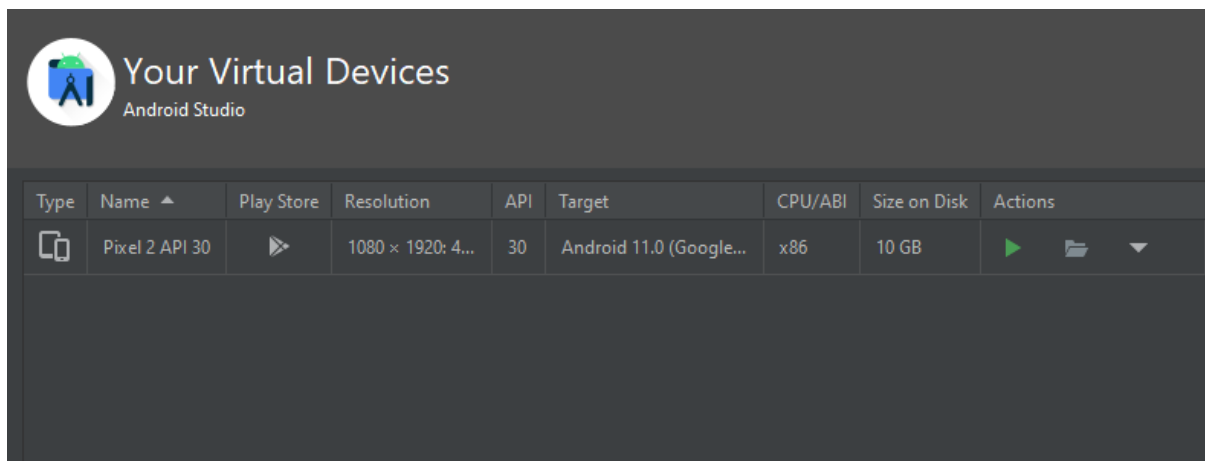
2.2.4. EMULAÇÃO DO DISPOSITIVO ANDROID: CRIANDO UM AVD

Um Dispositivo virtual Android (AVD, na sigla em inglês) é uma configuração que define as características de um smartphone ou tablet Android, Wear OS, Android TV ou um dispositivo Automotive OS que você queira simular no Android Emulator. Os AVDs contêm perfil de hardware, imagem do sistema, área de armazenamento, skin e outras propriedades.

Para criar um AVD no Android Studio, basta seguir os seguintes passos: abrir a IDE, clicar em “More Actions”, e depois em “AVD Manager”.



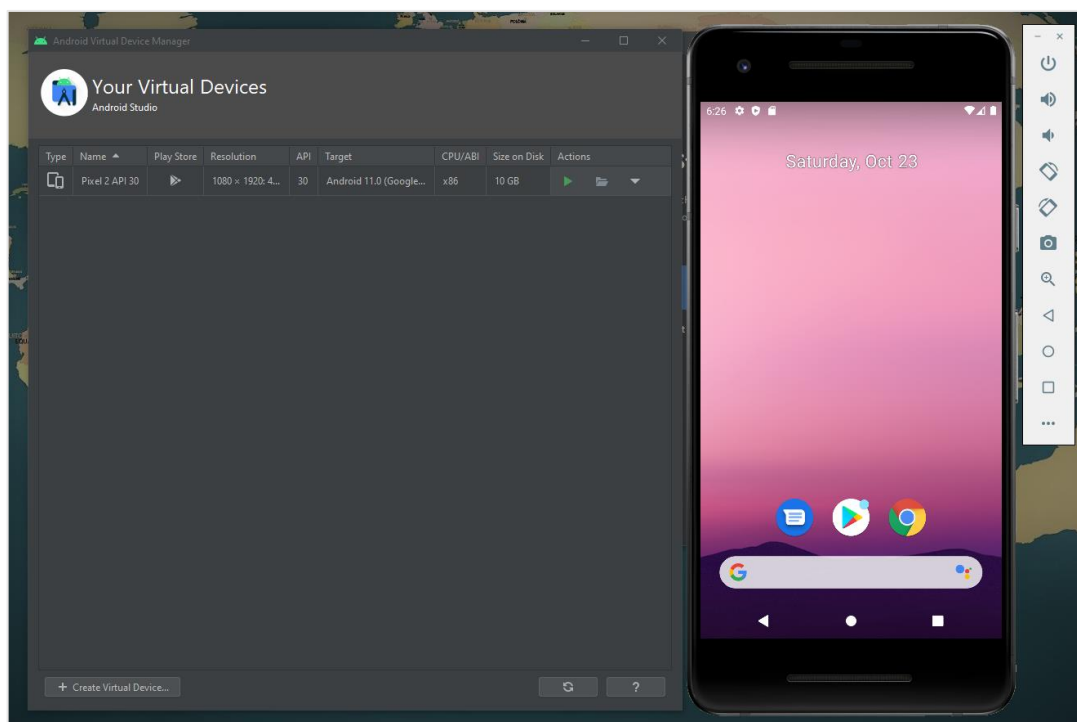
Após isso, será aberta a seção de gerenciamento de AVD's, e que, na primeira vez, não terá nenhum item disponível para uso. Na imagem abaixo, no entanto, existe um AVD disponível, porque já havia sido criado anteriormente.



Um AVD possui propriedades básicas, como o seu nome, a loja de aplicativos do mercado que é compatível, a sua resolução, [level da API](#), sua versão do Android, espaço em disco, entre outros. Veja mais sobre [aqui](#).

Agora é necessário criar um virtual device para executar o aplicativo Flutter que será criado. Para criar um, basta clicar em “+ Create Virtual Device”, selecionar a categoria e o dispositivo na tela de seleção que se apresentará, clicar em “Next”, e selecionar a imagem do sistema, que pode ser a imagem “R”, que é para as versões mais recentes do Android. Pronto! AVD foi criado!

Para testar a emulação, basta clicar no botão verde de “play” na coluna “Actions” do AVD criado na tela de gerenciamento de AVD's. O dispositivo irá ser carregado, lentamente, mas apenas na primeira vez. O dispositivo simulado, que será inicializado após esse processo, é tão funcional quanto um físico:



Para o desenvolvimento de um aplicativo que poderá ser executado em mais de um dispositivo, é importante criar os AVD's correspondentes aos ambientes de execução possíveis. Ou seja, para este teste, um AVD foi criado, mas no futuro será necessário, com certeza, mais de um.

Quando o código for criado, e a execução for necessária, será explicado como conectar o ambiente de desenvolvimento do aplicativo com um AVD criado.

2.3. FLUTTER É ORIENTADO A COMPONENT...WIDGET!

Todo aplicativo Flutter é um widget composto de outras centenas de widgets, em uma árvore ou hierarquia, que parte de um widget root, e progride para diversos outros fluxos parente-child.

Uma coisa interessante, e que é realmente nítida para aqueles com experiência em React, é que na própria documentação do Flutter, eles deixam claro que a inspiração para os widgets vieram dele. Assim como no React, em que todo aplicativo é baseado em componentes funcionais, no Flutter todo aplicativo é baseado em classes de widgets – o nome é outro, um utiliza funções, o outro classes, mas a lógica é a mesma.

Como diz na documentação, pense no aplicativo como um LEGO, onde cada pequeno widget representa uma peça e ao final, várias peças compõem um brinquedo. A seguir será explicada a classificação básica de widgets, embora existam outras: widgets de layout e de interface.

2.3.1. WIDGETS DE LAYOUT

Widgets de layout, são aqueles que se preocupam apenas em posicionar outros widgets.

Alguns dos principais:

- A) **Column Widget**: um widget que exibe seus filhos em uma matriz vertical. Ver [aqui](#).
- B) **Row Widget**: um widget que exibe seus filhos em uma matriz horizontal. Ver [aqui](#).
- C) **Scaffold Widget**: este widget, muito utilizado, provê uma **estrutura padrão** para os aplicativos, suportando o uso de componentes do [Material Design](#). Ver [aqui](#).
- D) **Stack Widget**: um widget que posiciona seus filhos em relação às bordas de sua caixa. Ver [aqui](#).

2.3.2. WIDGETS DE INTERFACE E PACOTES READY-TO-USE

Widgets de interface, são aqueles que efetivamente estão visíveis ao usuário, como: [Text](#), [RaisedButton](#) e [Switch](#).

Precisamos dar destaque para 2 conjuntos de widgets de interface amplamente utilizados: o primeiro e mais popular é o [pacote Material](#), que segue as definições de layout do Material Design, e o [pacote Cupertino](#), que segue as definições de design do iOS.

Ambos os pacotes fornecem **widgets ready-to-use**, isto é, “prontos para uso”, bastando importá-los no projeto e utilizá-los.

2.4. ÁRVORE OU HIERARQUIA DE WIDGETS

Widget tree é a estrutura que representa como nossos widgets estão organizados. Sendo assim, conforme vamos construindo nosso aplicativo, compondo widgets uns aos outros, esta estrutura pode (e com certeza vai) crescendo.

Veja mais sobre isso [aqui](#).

2.5. ESTADOS DOS WIDGETS

Basicamente, um estado é uma informação ou grupo de informações que são alteradas durante o tempo de execução do aplicativo. Novamente, para quem conhece React, a lógica dos states é fácil e clara, uma vez que lá os estados também são fundamentais.

Essa alteração das informações em tempo de execução é chamada de mudança de estado. Podemos entender o estado como uma variável que persiste os dados atribuídos durante todo o tempo de execução, e enquanto ela não sofrer uma alteração.

2.5.1. STATELESS WIDGET

Vamos ver agora o que é um Stateless widget da forma mais simples possível: na prática, é um widget sem o controle de estado. Este tipo de widget não possibilita alterações dinâmicas, entenda-o como algo completamente estático.

Imagine-os como tijolos em uma construção: são a base da construção, mas não fazem nada além disso.

2.5.2. STATEFUL WIDGET

Os widgets Stateful são praticamente o oposto dos Stateless. Eles contêm estado e isso os torna mutáveis. É por meio deles que construiremos boa parte das aplicações e componentes ao trabalhar com Flutter.

Os widgets com controle de estado são os elementos-chave para o desenvolvimento móvel da forma interativa que conhecemos.

2.5.3. ESTADO LOCAL E GLOBAL

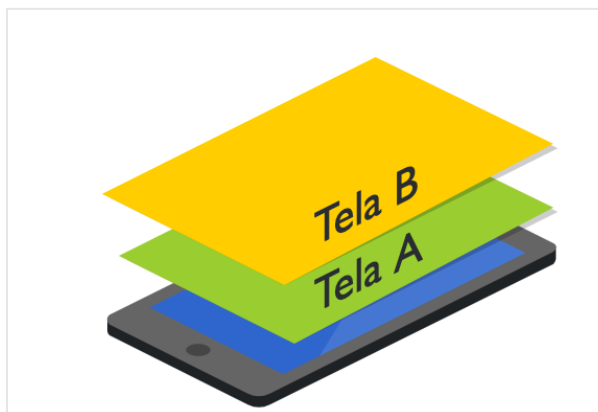
Aprofundando um pouco mais sobre os conceitos do state (estado), temos dois tipos quanto a sua abrangência no aplicativo:

A) Estado Local: é o estado isolado de um único widget.

B) Estado Global: é um estado compartilhado por diversos widgets, ou cujo acesso, como valor, está disponível em toda a aplicação.

2.6. NAVEGANDO ENTRE PÁGINAS: PILHAS DE TELAS

A navegação no Flutter utiliza o conceito de "pilha" (stack). Ou seja, se estamos na Tela A e vamos para a Tela B, internamente, estamos colocando a Tela B acima da Tela A. Assim, a Tela B está no "topo" da pilha.



Para manipular a "pilha" de telas no Flutter, utilizamos o [widget Navigator](#). Com o princípio de "o último que entra é o primeiro que sai", temos as operações de push e pop, para adicionarmos e removermos telas da nossa "pilha" de navegação.

3. COMEÇANDO UM PROJETO FLUTTER

Uma aplicação Flutter é criada a partir de um projeto, que é, objetivamente, assim como um projeto Dart, uma estrutura de pastas e arquivos gerados por um comando de terminal.

3.1. COMANDOS BÁSICOS – CMD/CLI

A ferramenta de linha de comando é o meio pelo qual os desenvolvedores (ou IDEs em nome dos desenvolvedores) interagem com o Flutter para criar, analisar, testar e executar projetos.

3.1.1. CRIAÇÃO DO PROJETO FLUTTER

O comando para criar um projeto Flutter é:

flutter create my_app

É importante que o caminho do local onde o projeto é gerado não contenha caracteres como traços, acentos e espaços, porque alguns comandos, como para executar o projeto, acusam erros relacionados à codificação de nomes do caminho.

3.1.2. INSTALAÇÃO DE PACOTES

O comando para instalar pacotes em uma aplicação Flutter é:

flutter pub add nome_pacote

Também é possível escrever diretamente a dependência no arquivo de gerenciamento de dependências, e em seguida realizar o comando “**flutter pub get**”.

3.1.3. ATUALIZAÇÃO DOS PACOTES

Para atualizar as dependências o comando é esse:

flutter pub upgrade

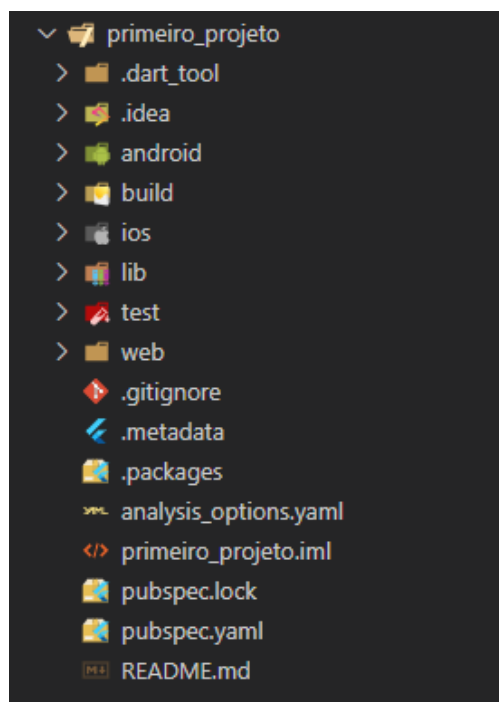
3.1.4. OUTROS COMANDOS

Para ver outros comandos do flutter CLI, e entender melhor sobre isso, clique [aqui](#).

3.2. ARQUITETURA DE UM PROJETO FLUTTER

Executando o comando para criar um projeto Flutter, por exemplo, “**flutter create primeiro_projeto**”, uma pasta com esse mesmo nome será criada no local, e dentro dela terão os recursos que compõem um projeto.

Veja a estrutura de pastas e arquivos que é gerado pela execução deste comando, e com esse mesmo nome utilizado de exemplo:



A primeira pasta é a “**dart_tool**”, que também existe no template de projetos Dart. Ela é utilizada pelo gerenciador de pacotes PUB, e por outras ferramentas, e contém o importante arquivo `package_config.json` que possui as especificações dos pacotes utilizados no projeto.

A segunda pasta é a “**idea**”, que contém configurações para algumas IDEs, como as da JetBrains.

A **terceira pasta é a “android”**, que contém arquivos e pastas necessários para executar o aplicativo em um sistema operacional Android. Esses arquivos e pastas são gerados automaticamente durante a criação do projeto de flutter.

A **quarta pasta é a “build”**, que contém o código compilado de seu aplicativo Flutter. O conteúdo desta pasta é gerado automaticamente como parte do processo de construção do Flutter, para que você não precise alterar nada manualmente.

A **quinta pasta é a “ios”**, que contém arquivos e pastas necessários para executar o aplicativo em um sistema operacional iOS.

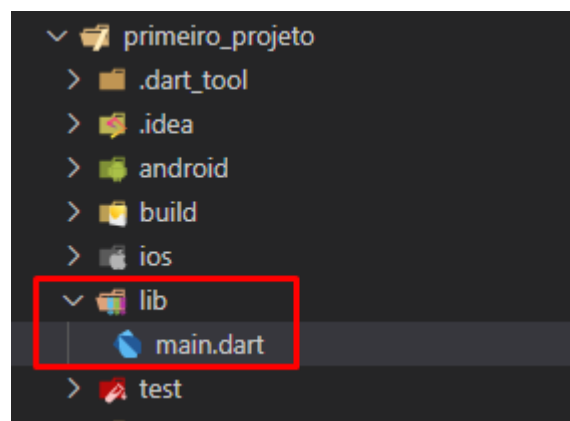
A **sexta pasta é a “lib”**, que conterá todos os arquivos de extensão dart relacionados ao projeto flutter. É nessa pasta onde existirão os códigos que darão vida ao aplicativo.

A **sétima pasta é a “test”**, que, assim como em projetos Dart, terá os scripts de teste dos códigos do aplicativo.

A **oitava e última pasta é a “web”**, e ela contém recursos para a criação de uma versão web do aplicativo. Inclusive, nessa pasta, existe um arquivo index.html com algumas configurações iniciais.

3.3. CÓDIGO BÁSICO DE UM PROJETO FLUTTER

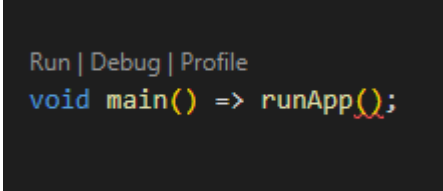
Como já sabemos, os scripts que dão vida ao código existem na pasta “lib”, e de fato quando o projeto é criado, também é gerado um arquivo de nome main e extensão dart nessa mesma pasta.



Nesse arquivo serão escritos os códigos para dar forma e rosto ao aplicativo. Pois bem: lembrando da seção sobre widgets, o aplicativo Flutter é como um lego, em que as peças prontas são os widgets, e a lógica de encaixes é uma orientada a uma hierarquia de relações parent-child.

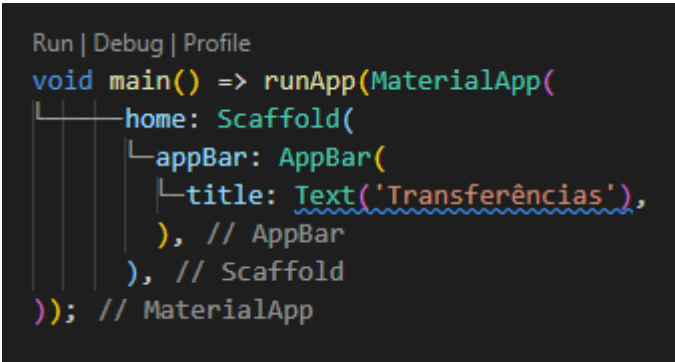
Também, lembrando, o começo dessa hierarquia de widgets se dá também em um widget, que pode ser chamado de Widget “root”, porque ele contém todos os outros.

Traduzindo isso para o código, o arquivo main possui, como todo outro programa Dart, uma função de alto nível chamada “main()”, mas, aqui no Flutter, ela possui uma outra dentro de si, padrão, que não pode ser alterada, chamada “runApp()”.



```
Run | Debug | Profile
void main() => runApp();
```

Essa função, “runApp()” espera como argumento um widget.., e é aqui que a aplicação tem o seu início, porque esse widget esperado deve ser o root! O começo dos encaixes de widgets se dá, então, nessa função, e depois prossegue indefinidamente. Veja o exemplo abaixo:



```
Run | Debug | Profile
void main() => runApp(MaterialApp(
  | home: Scaffold(
    | | appBar: AppBar(
      | | | title: Text('Transferências'),
      | | | ), // AppBar
    | | ), // Scaffold
  )); // MaterialApp
```

O widget root é o MaterialApp, que possui encaixes pré-definidos para outros widgets do pacote Material. Veja que o encaixe “home” utilizado é dele, e esse encaixe recebe o widget “Scaffold”, que, como já explicado anteriormente, é do tipo layout. Então o “Scaffold”, como um widget, possui outros encaixes, e um deles é utilizado, o “AppBar”, que recebe o widget de interface “AppBar”. Por último, esse widget, o “AppBar” tem o seu encaixe “title” utilizado, recebendo um outro widget, de nome “Text” que serve justamente para receber um texto.

E assim segue uma aplicação Flutter. É um amaranhado de widgets que possuem encaixes para outros widgets, que possuem encaixes para outros, e assim em diante.

3.4. EXECUTANDO O CÓDIGO NO ANDROID EMULADO

3.4.1. PRIMEIRA EXECUÇÃO

Tendo escrito aquele código de exemplo, no tópico anterior, agora você pode acessar o local do projeto, via linha de comando, e digitar o comando “**flutter emulators**” para verificar

os emuladores criados, e em seguida “**flutter emulators –launch ID**” para executar o código no dispositivo, sendo ID igual ao identificador do dispositivo.

Na escrita desse texto, o dispositivo criado e que será utilizado é Pixel_2_API_30. Assim, digitando o comando com esse ID, o resultado é esse:

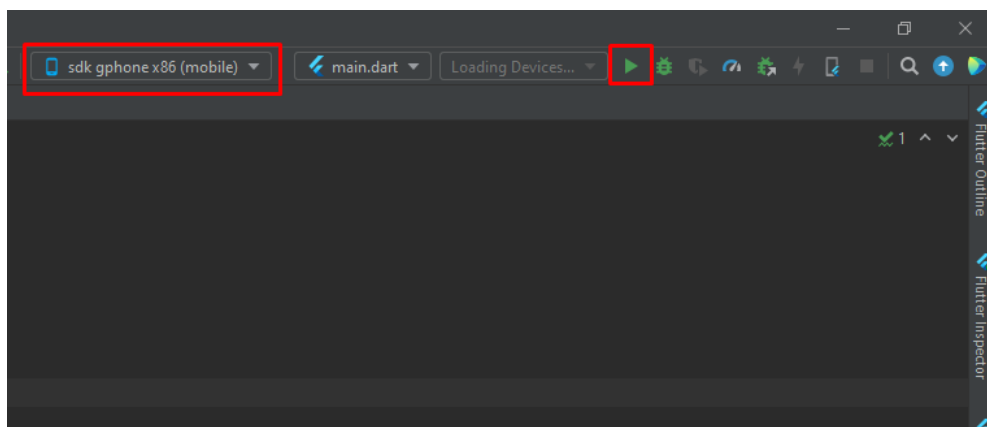


É possível que, em algumas ocasiões, o aplicativo não atualize o seu estado quando o emulador é inicializado. Nestes casos, uma alternativa é executar o comando “**flutter clean**”, e depois “**flutter run**”.

Além dessas soluções possíveis, utilizar as IDEs “especializadas” também pode evitar que esse problema ocorra.

3.4.2. EXECUTANDO O CÓDIGO NO INTELLIJ IDEA

Estando com o projeto aberto no IntelliJ Idea, basta selecionar o dispositivo emulado no menu de seleção de dispositivo, que por padrão estará como “<no devices>”, e após isso clicar no botão “play” de cor verde (Shift + F10).



A execução pelo mecanismo da IDE é essencial porque executa o aplicativo com o código no seu estado presente. Pode ocorrer, de vez em quando, de o dispositivo ser inicializado com o aplicativo já em execução, mas em um estado anterior às atualizações do código.

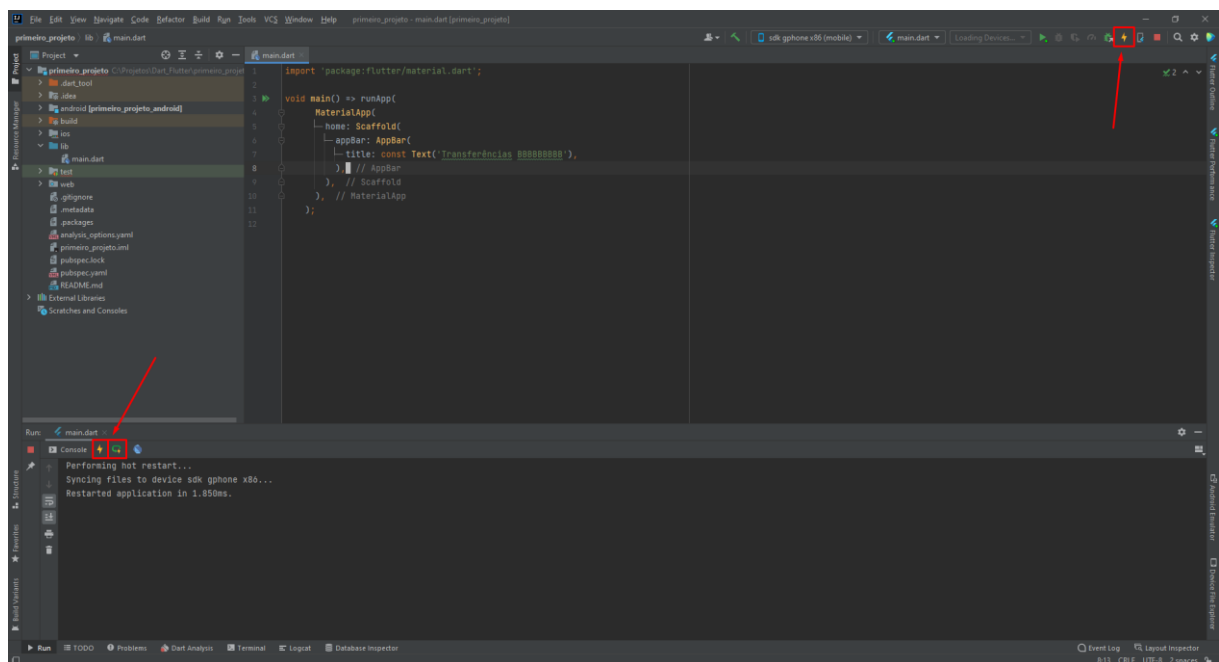
3.5. EXECUTANDO O CÓDIGO NO VISUAL STUDIO CODE

3.5.1. HOT RELOAD E HOT RESTART

Quando construímos qualquer aplicativo Flutter, é preciso esperar alguns segundos para que ele seja executado, ou mais, dependendo da capacidade da máquina.

Para resolver isso temos dois recursos em um flutter: o Hot Reload, e o Hot restart. Eles ajudam a diminuir o tempo de execução de nosso aplicativo depois de executá-lo. Eles são muito melhores e mais rápidos do que o reinício padrão, mas só podem ser utilizados após a primeira execução do sistema.

No IntelliJ Ideal esses recursos podem ser encontrados facilmente. Veja na imagem seguir sua localização na interface da IDE:



Próximo do canto superior direito, o símbolo de raio é o botão para o Hot Reload, que também existe próximo do canto inferior esquerdo, e ao lado de um outro botão, o de Hot Restart.

Existem algumas importantes diferenças entre os dois processos, embora ambos possam servir para atualizar rapidamente o aplicativo. A principal diferença entre eles, é que o **Hot Reload só funciona com widgets de classe que implementam o método construtor**. Isso

ocorre porque o que ele faz é forçar a reconstrução dos widgets de classe a partir de seus construtores. Além disso, o **Hot Reload preserva os states do aplicativo**.

Em contraste, o Hot Restart, que é um pouco mais demorado, não preserva quaisquer estados do aplicativo. Ele o atualiza completamente. Qualquer mudança, em qualquer parte da aplicação é considerada na reinicialização com essa funcionalidade.

FONTES

<https://www.flutterparainiciantes.com.br/>

<https://flutter.dev/docs>

<https://www.alura.com.br/artigos/flutter-diferenca-entre-stateless-e-statefull-widget>

