PROGRAMAÇÃO MOBILE – NOVAS TECNOLOGIAS

Flutter

CRIANDO A PRIMEIRA APLICAÇÃO MOBILE

Professor: ***Ronaldo Silva***
Formação: ***Análise e Desenvolvimento de Sistemas***
Especialização: ***Sistemas de Informação***

SOBRE ESTE MATERIAL

O conteúdo abordado a seguir trata-se de um material de apoio para a disciplina de "Programação Mobile - Novas Tecnologias" do curso **TDS** *(Técnico em Desenvolvimento de Sistemas)* fornecido pela **ETE** *(Escola Técnica Estadual)*. Elaborado pelo professor ***Ronaldo Silva***, formado em *Análise e Desenvolvimento de Sistemas* pela ***Universidade Estácio de Sá*** e Pós-Graduado em *Sistemas de Informação* pela ***Faculdade Unyleya***. Esta apostila tem como base principal trazer conhecimento de programação para *Dispositivos Móveis* sob novos métodos e aplicação para os alunos junto ao norteamento dos professores da área destinada.

"O conhecimento é imprescindível para o crescimento pessoal e profissional."

(Prof. Ronaldo Silva)

"Sorte é o que acontece quando a oportunidade encontra alguém preparado."

(Sêneca)

Bons Estudos!"

ÍNDICE

INTRODUÇÃO

A todo momento surgem soluções do tipo “um código, dois aplicativos” entre as comunidades de desenvolvimento mobile. Isso é natural porque a ideia de escrever um único código que pode ser reutilizado entre aplicações e em diversas plataformas é tentadora tanto para quem financia o projeto, pela possibilidade de manter equipes menores, quanto para o desenvolvedor, que muitas vezes pode utilizar suas habilidades como ponto de partida no estudo de uma tecnologia, reduzindo assim a sua curva de aprendizado. Entretanto, muitas dessas soluções, citadas no fim, tendem a sacrificar a performance para alcançar certa onipresença.

Por exemplo, o Ionic, por executar em uma Webview, pode entregar um aplicativo multiplataforma, mas a um custo de desempenho altíssimo. O React Native, embora utilize código nativo de cada plataforma, ainda precisa de “pontes” que façam a ligação entre as mesmas e a lógica JavaScript da sua aplicação, o que também podem ser um causador de gargalos.

E se pudéssemos gerar código nativo de alto desempenho tanto para Android quanto para iOS, utilizando uma linguagem de alto nível? E se existisse uma biblioteca que nos desse todos os elementos visuais necessários para uma aplicação, seja seguindo o Material Design do Android ou o padrão de design do iOS, que permitisse a criação de aplicativos extremamente elegantes, com animações fluídas? Esse mundo existe e se chama Flutter.

CONHECENDO O FLUTTER

Flutter é um kit de desenvolvimento de software de interface de usuário (**toolkit** e **framework**), de código aberto, criado pela empresa Google em 2015, baseado na linguagem de programação ***Dart***, que possibilita a criação de aplicativos para Web (através de **WebAssembly**) e para os sistemas operacionais **Android**, **iOS**, **Windows**, **Linux** e **Fuchsia**.

O Flutter foi apresentado pela primeira vez em meados de 2017 pelo Google, mas foi em dezembro de 2018, que a primeira versão estável foi liberada para o público. Com o Flutter podemos construir aplicativos nativos e de alta qualidade tanto para Android quanto para iOS, respeitando os padrões de cada uma dessas plataformas, a partir de um mesmo código escrito com a linguagem Dart.

Esse material ajudará a criar nosso primeiro aplicativo utilizando o Flutter. Você poderá testar os códigos no **DartPad**, um editor online usado em navegadores modernos. No caso dos testes mais sólidos será necessário realizar a instalação do **Flutter SDK**, um conjunto de ferramentas e componentes que serão utilizados para construir e compilar a aplicação. Começaremos com a configuração do **Visual Studio Code**, de forma a integrá-lo com o SDK e, dessa forma, seremos capazes de utilizar o recurso de **Hot Reload**, bem como o debugger disponibilizado pelo Flutter. Hot Reload é uma tecnologia que agiliza a depuração, diminuindo o tempo necessário para que as alterações feitas no aplicativo sejam enviadas para o dispositivo que executa a aplicação. Para aumentar a opção e o melhor aprendizado você também poderá usar o **Android Studio** tendo assim uma excelente opção para desenvolvimento.

LINGUAGEM DE PROGRAMAÇÃO (DART)

Dart (originalmente denominada ***Dash***) é uma linguagem de script voltada à web desenvolvida pela Google. Ela foi lançada na GOTO Conference 2011, que aconteceu de 10 a 11 de outubro de 2011 em Aarhus, na Dinamarca. O objetivo da linguagem Dart foi inicialmente a de substituir a JavaScript como a linguagem principal embutida nos navegadores.[4] Programas nesta linguagem podem tanto serem executados em uma máquina virtual quanto compilados para JavaScript.

Em novembro de 2013, foi lançada a primeira versão estável, Dart 1.0. Em agosto de 2018 foi lançado o Dart 2.0, um *reboot* da linguagem, otimizado para o desenvolvimento *client-side* para Web e dispositivos móveis.

DARTPAD

DartPad é um editor online gratuito usado em navegadores modernos e de código aberto para ajudar desenvolvedores a aprender sobre Dart e Flutter. Você pode acessá-lo em dartpad.dev .

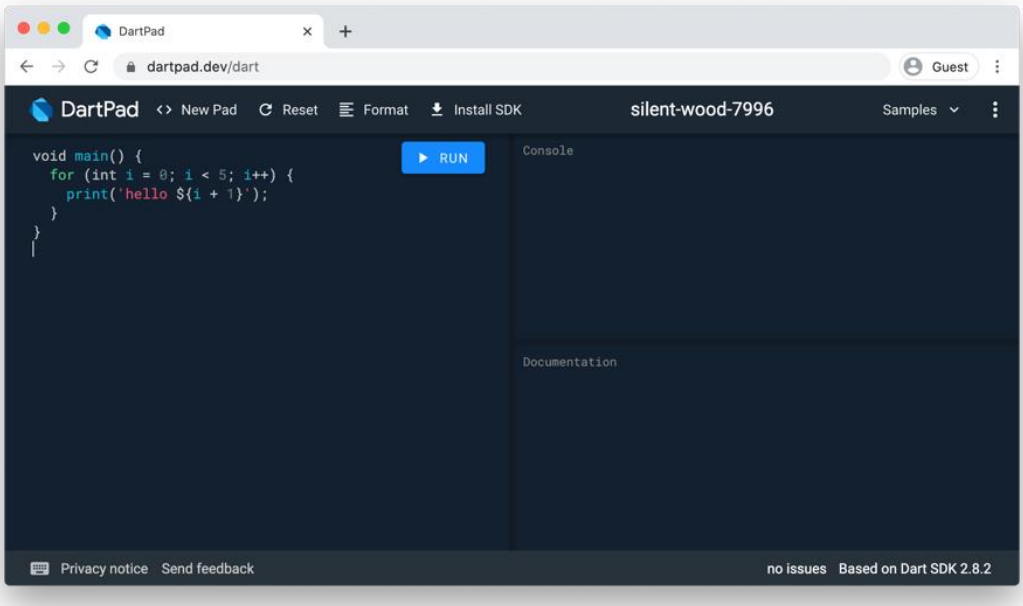

Figura 1. Página do DartPad.

INSTALANDO O FLUTTER SDK

O Flutter SDK pode ser utilizado no Windows, MacOS e Linux. Aqui, abordaremos apenas a instalação em ambiente Windows. Antes de mais nada, a fim de gerarmos um APK para o Android, também precisaremos do Android SDK, que pode ser obtido pela instalação do Android Studio. No DevMedia tem um curso que ensina **como preparar um ambiente de desenvolvimento Android**, que cobre esse passo e pode ser utilizado como consulta, caso você precise de alguma ajuda.

Com o Android SDK instalado, podemos partir para a instalação do Flutter SDK:

Na página de download do **Flutter SDK**, clique no botão "Windows".

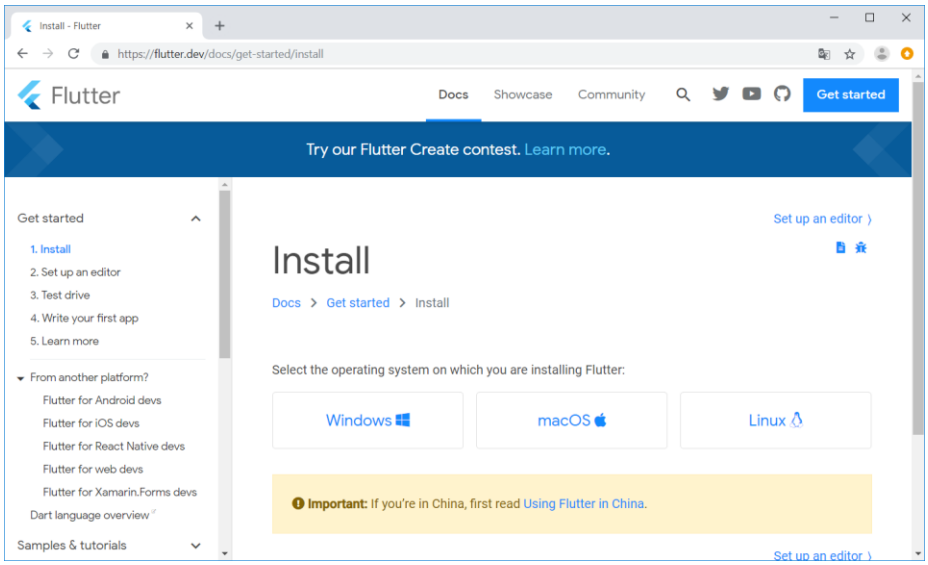

Figura 2. Página com as opções de download do Flutter SDK.

Na sessão "Get the Flutter SDK", logo abaixo dos requerimentos do sistema, clique no botão flutter_windows_v1.0.0-stable.zip.

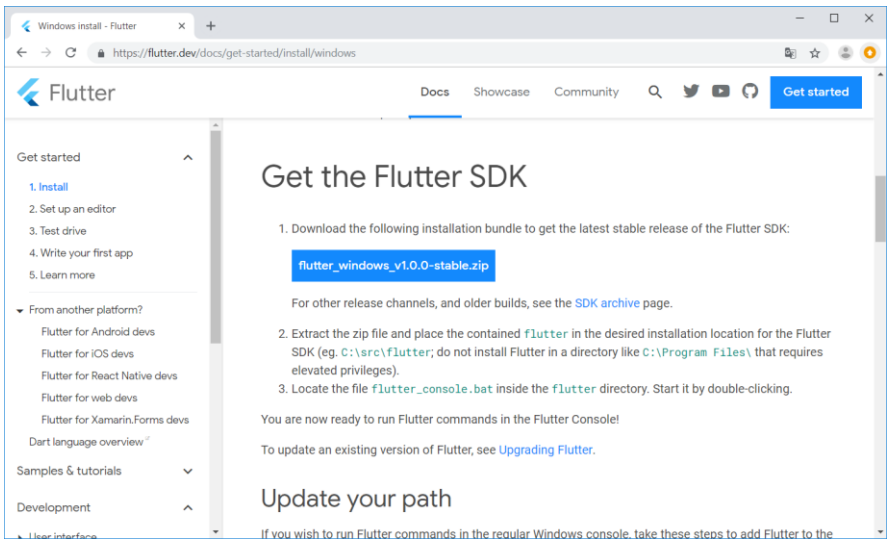

Figura 3. Página de download do Flutter SDK para Windows.

Na partição C:/ do seu HD, crie um diretório chamado src/ e extraia os arquivos do SDK nesse diretório:

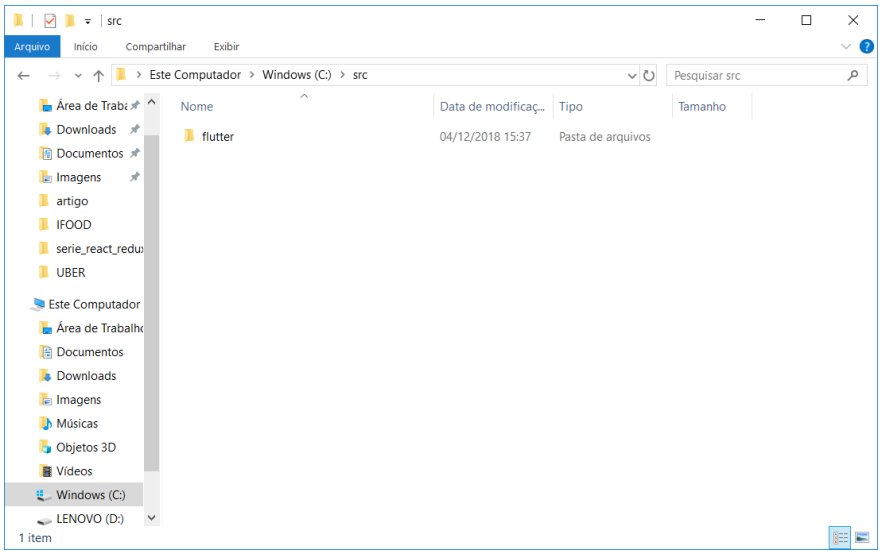

Figura 4. Diretório C:/src/

Agora que realizamos o download do SDK, criaremos uma variável de ambiente do Windows para sermos capazes de utilizar a ferramenta de linha de comando do Flutter através do terminal do Windows:

Na barra de busca do Windows pesquise por "Editar as variáveis de ambiente do sistema" e clique no primeiro resultado:

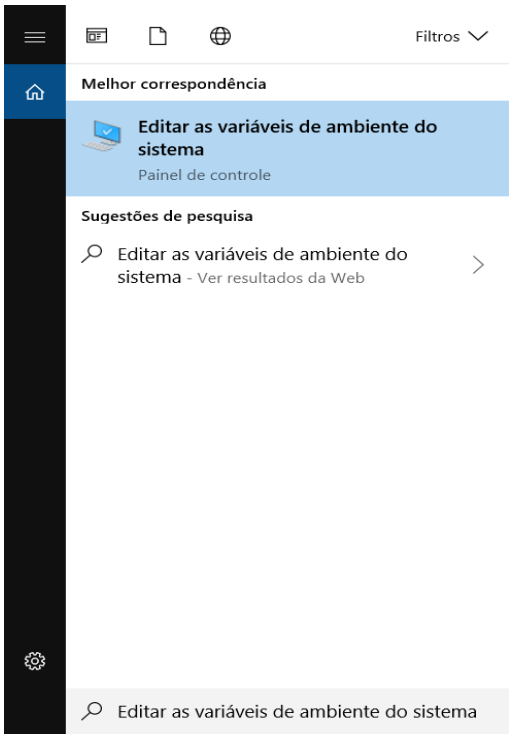

Figura 5. Barra de busca do Windows.

Na janela que abrir, clique no botão "Variáveis de Ambiente…":

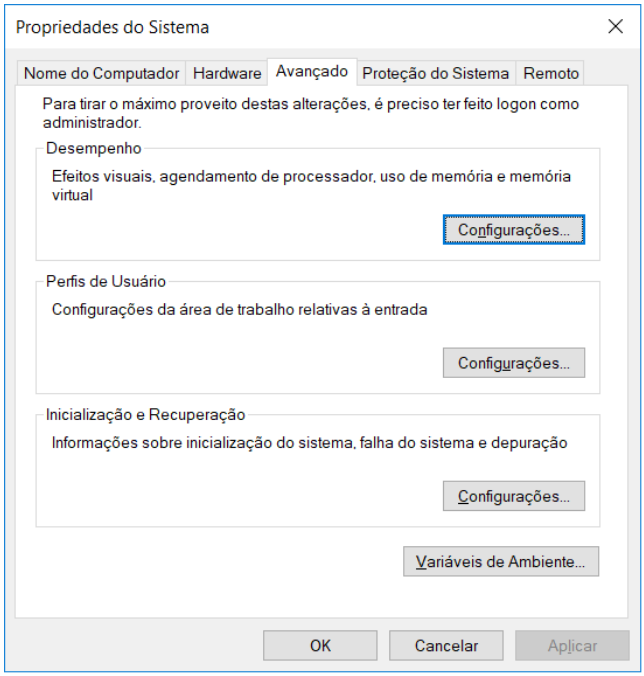

Figura 6. Janela de propriedades do sistema no Windows 10.

No campo superior, chamado "Variáveis de usuário…", clique na variável "Path" e,então, no botão "Editar":

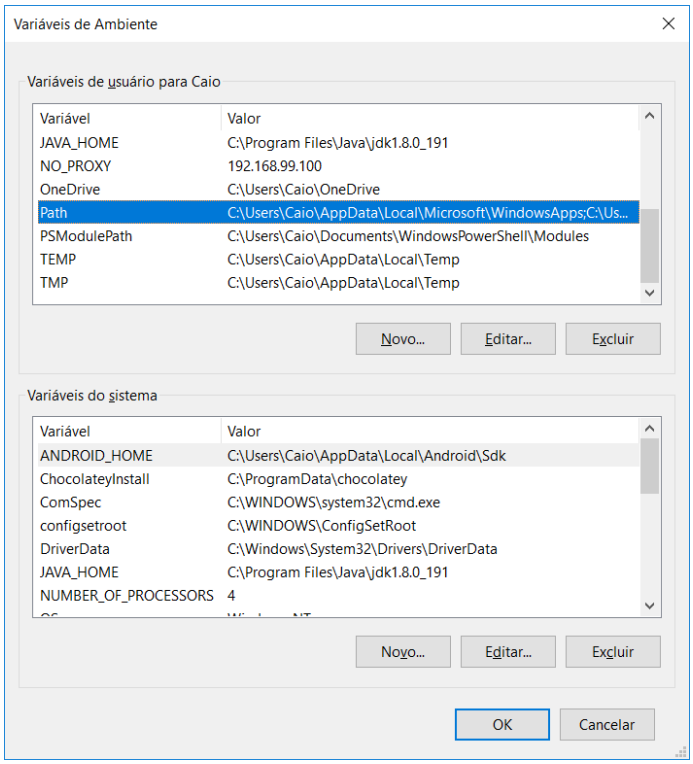

Figura 7. Variáveis de ambiente do sistema.

Na janela que abrir encontraremos uma lista de diretórios. Clique em "Novo" e adicione o endereço do diretório C:/src/flutter/bin do SDKe clique em "OK":

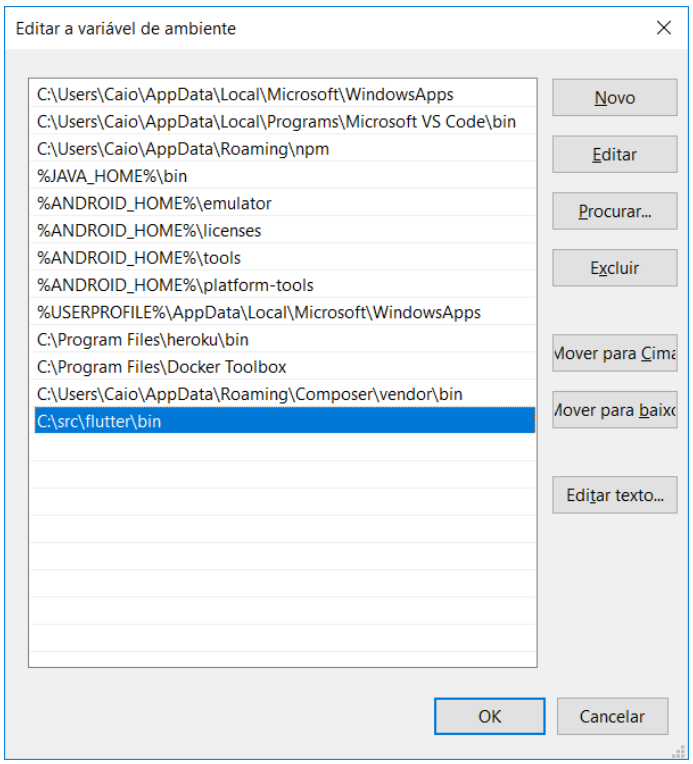

Figura 8. Caminho do Flutter SDK adicionado a variável de ambiente Path.

Para verificar se o SDK foi instalado corretamente, abra uma sessão do CMD e execute o seguinte comando:

```
flutter doctor
```

Se tudo estiver instalado corretamente, deverá receber uma resposta similar a essa:

```
Doctor summary (to see all details, run flutter doctor -v):[√] Flutter
(Channel beta, v1.1.8, on Microsoft Windows
[versão 10.0.17134.590], locale pt-BR)
[!] Android toolchain - develop for Android devices (Android SDK version 28.0.3)
   ! Some Android licenses not accepted.  To resolve this,
   run: flutter doctor --android-licenses
[√] Android Studio (version 3.2)
[√] VS Code (version 1.31.1)
[!] Connected device
   ! No devices available
```

INSTALAÇÃO DO VS CODE

Pronto! Agora que já possuímos o SDK, estamos prontos para configurar nossa IDE. Como exemplo usaremos o Visual Studio Code, você poderá baixar no link https://code.visualstudio.com/download, mas se quiser também pode utilizar o Android Studio que será visto mais a frente (basta ter memória RAM suficiente) em seguida também veremos a instalação do Git:

O **Visual Studio Code** é um editor de código-fonte desenvolvido pela Microsoft para Windows, Linux e macOS. Ele inclui suporte para depuração, controle de versionamento Git incorporado, realce de sintaxe, complementação inteligente de código, *snippets* e refatoração de código. Ele é customizável, permitindo que os usuários possam mudar o tema do editor, teclas de atalho e preferências. Ele é um software livre e de código aberto, apesar do download oficial estar sob uma licença proprietária.

O Visual Studio Code é baseado no Electron, um *framework* que é usado para desenvolver aplicativos Node.js para o desktop rodando no motor de layout Blink. Apesar de usar o Electron como *framework*, o software não usa o Atom e em seu lugar emprega o mesmo componente editor (apelidado "Monaco") usado no Visual Studio Team Services (anteriormente chamado de Visual Studio Online) .

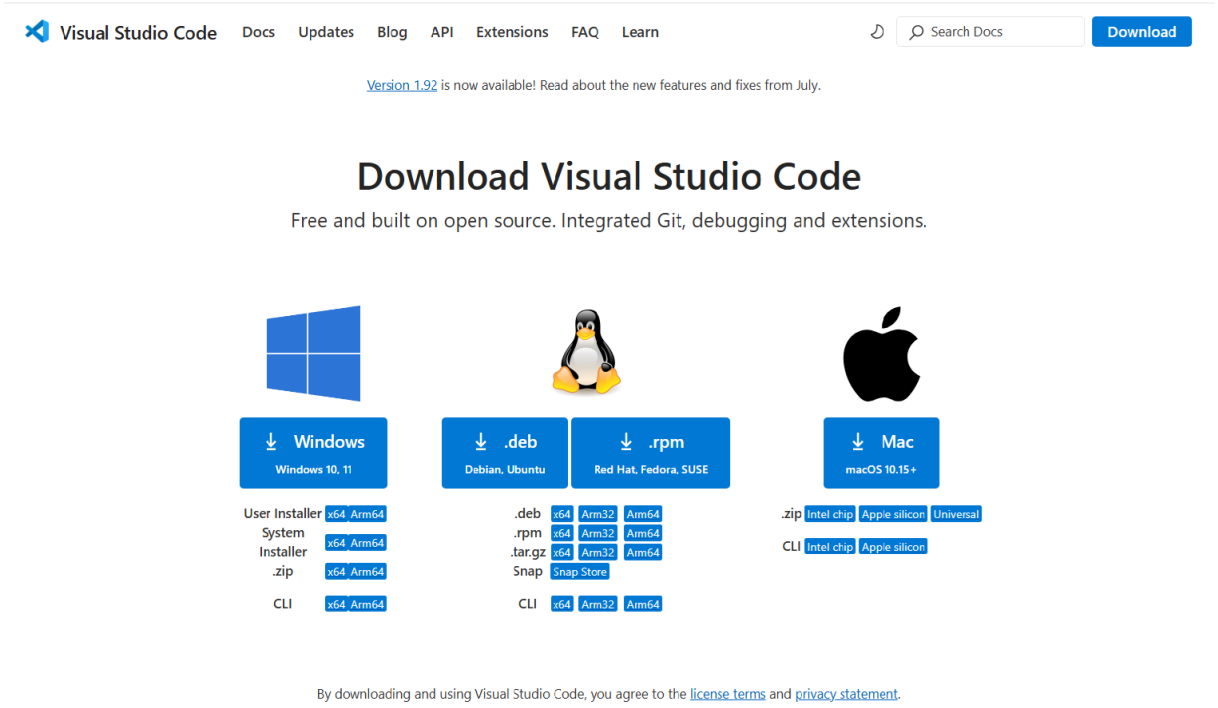

Figura 9. Download do Visual Studio Code

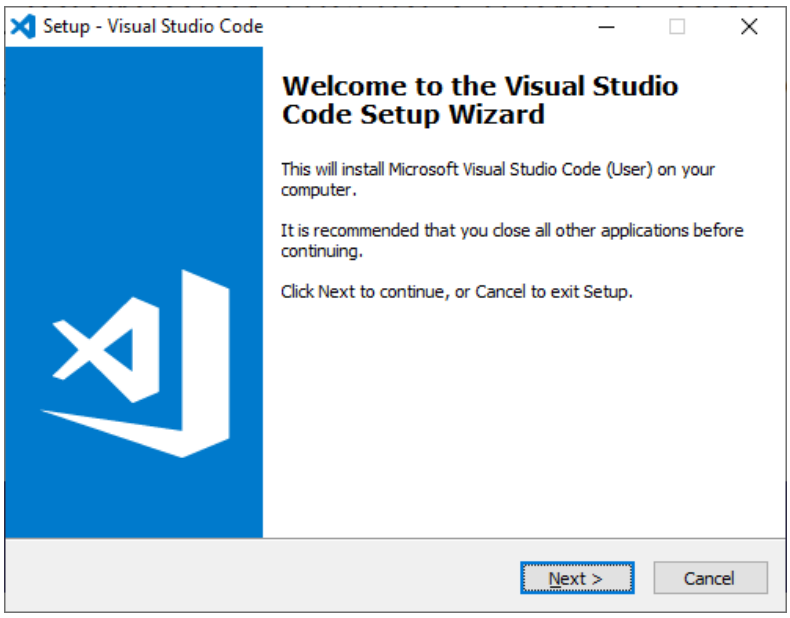

Figura 10. Janela de Instalação do Visual Studio Code

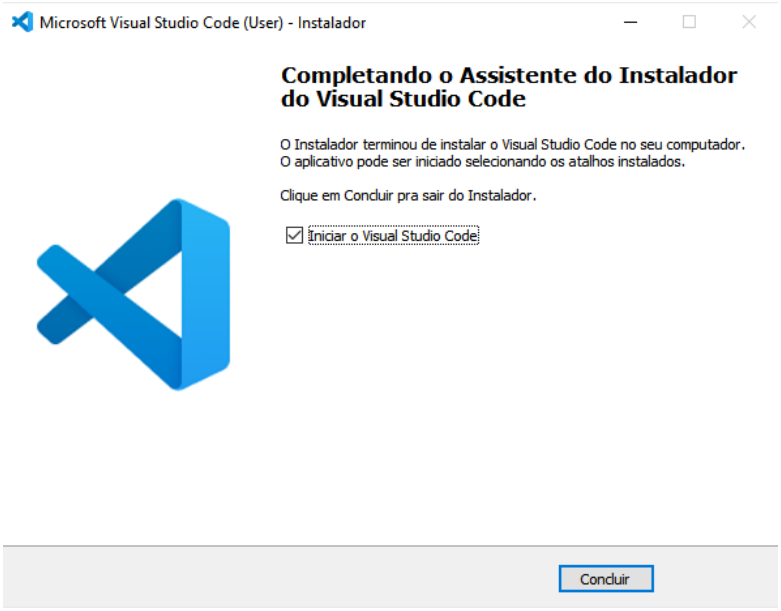

Figura 11. Término da Instalação do Visual Studio Code

Com o Visual Studio Code aberto, no menu lateral, selecione a aba "extensões":

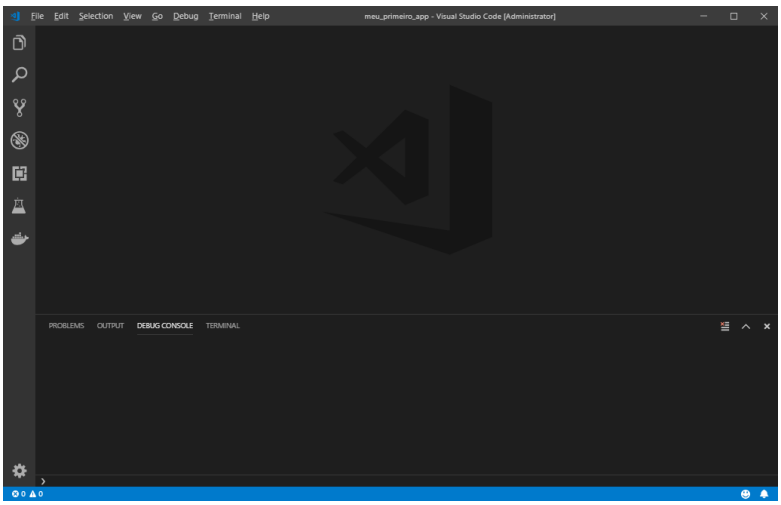

Figura 12. Botão de extensões do Visual Studio Code

No campo de busca, digite "Flutter". O primeiro resultado da busca será o plugin oficial. Clique no pequeno botão "Install" verde. O plugin já contém tudo que precisamos para executar e debugar nosso código Dart.

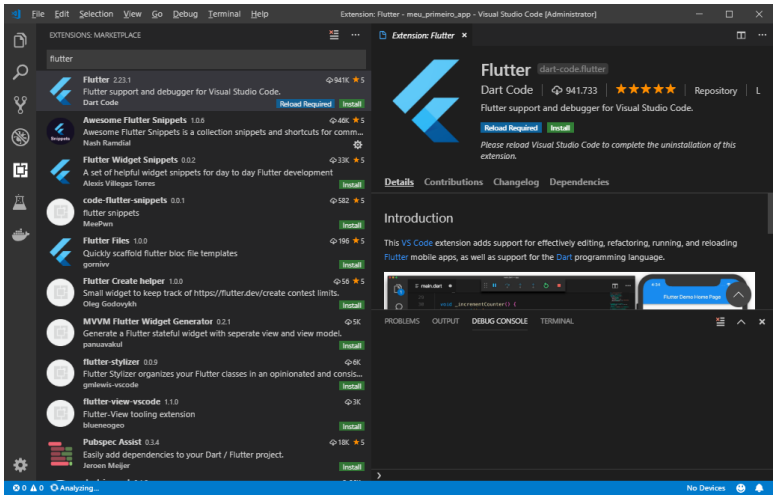

Figura 13. Plugin oficial do Flutter para Visual Studio Code

Pronto! Finalmente podemos criar nosso primeiro aplicativo em Flutter no VS Code!

INSTALAÇÃO DO ANDROID STUDIO

Uma das opções mais usadas é o Android Studio que deixa seu projeto com características ainda mais inovadoras e robustas.

Além do SDK do Flutter, precisamos de uma IDE (ou editor de textos) para escrevermos nossos códigos e, com isso, desenvolvermos nossas aplicações. O Android Studio possui um suporte incrível para construir aplicações Flutter, desde criar um projeto até a compilação.

Para instalação do Android Studio, é necessário baixar o instalador da página oficial, independente da plataforma que esteja utilizando. Você pode baixar a última versão do Android Studio em sua página oficial. Após acessar este link, basta clicar no botão “Download Android Studio” e aceitar os termos de licenciamento para que o download seja iniciado.

Após a conclusão do Download, o processo de instalação dependerá de sua plataforma e é bem simples, seguindo as etapas sugeridas pelo programa.

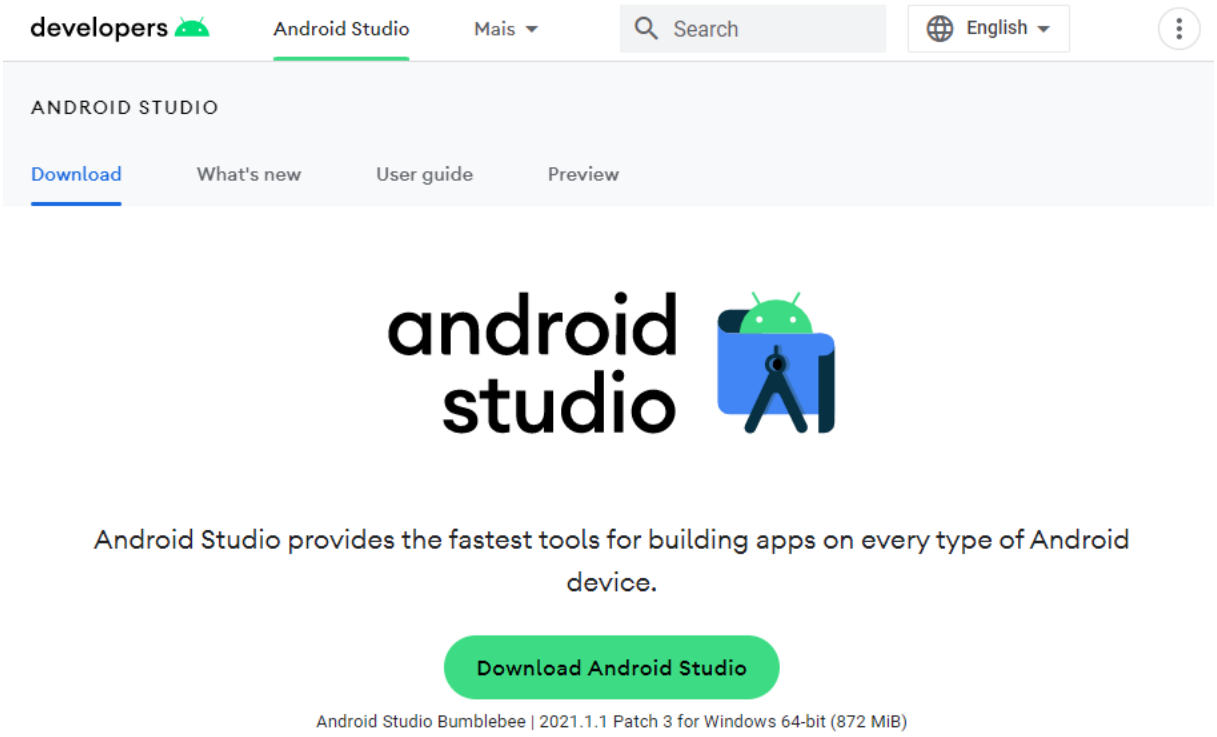

Figura 14. Download do Android Studio

Figura 15. Janela de Instalação do Android Studio

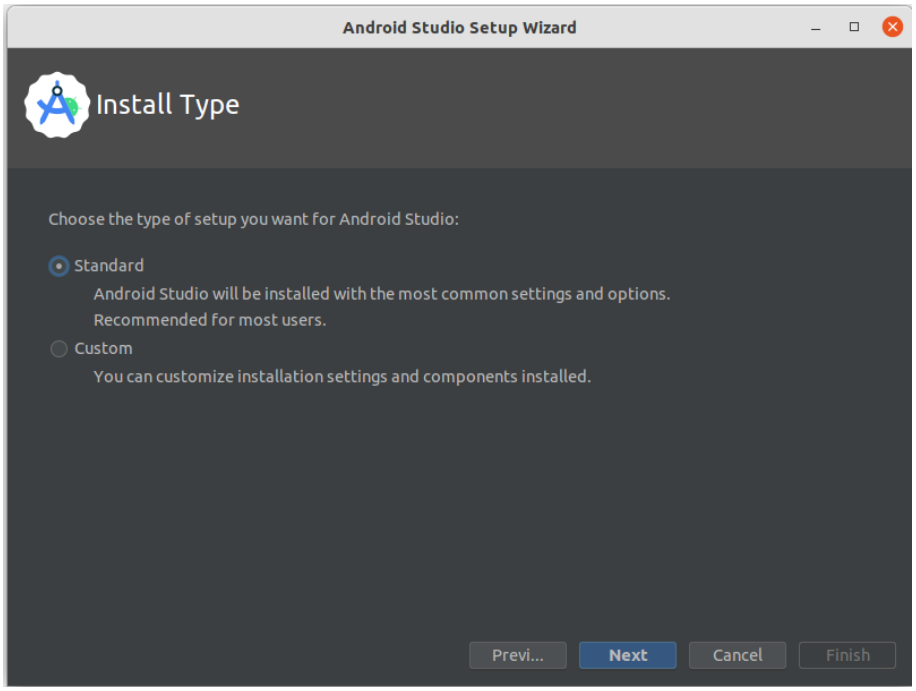

Figura 16. Instalação Padrão do Android Studio

CRIANDO UM EMULADOR NO ANDROID STUDIO

Um emulador facilita (e muito) no desenvolvimento de nossos aplicativos, já que é com ele que conseguiremos testar nosso app e verificar se tudo está funcionando como deveria. Para isso, o Android Studio permite a criação e gerenciamento de emuladores dentro da IDE.

Para criar um emulador no Android Studio, devemos seguir os seguintes passos:

Ao iniciar o Android Studio, clicamos em Configure e selecionamos a opção AVD Mananger, como mostrado na imagem abaixo:

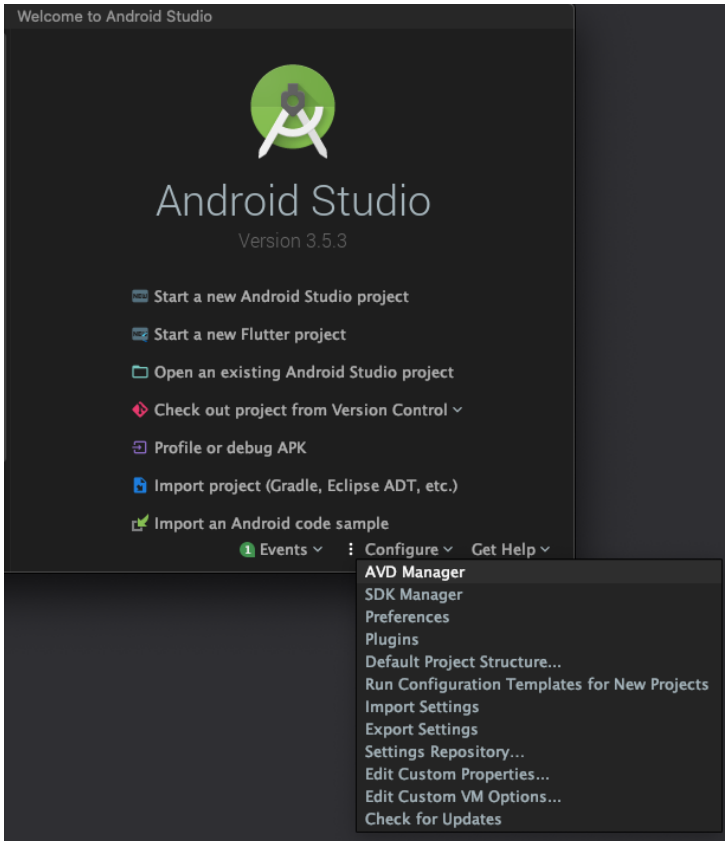

Figura 17. Android Studio já instalado exibindo a etapa de criação do dispositivo virtual.

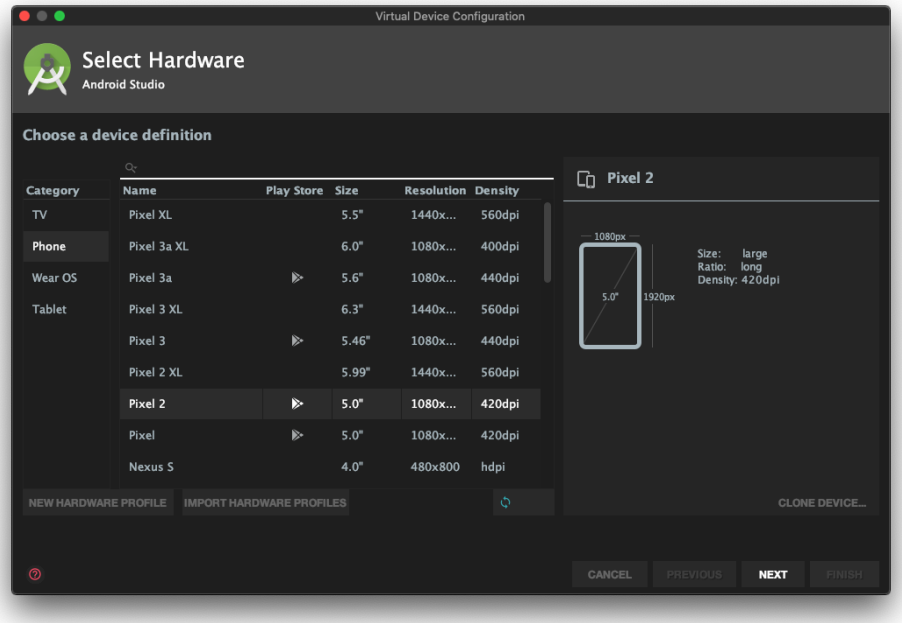

Figura 18. Android Studio - Criando um dispositivo virtual.

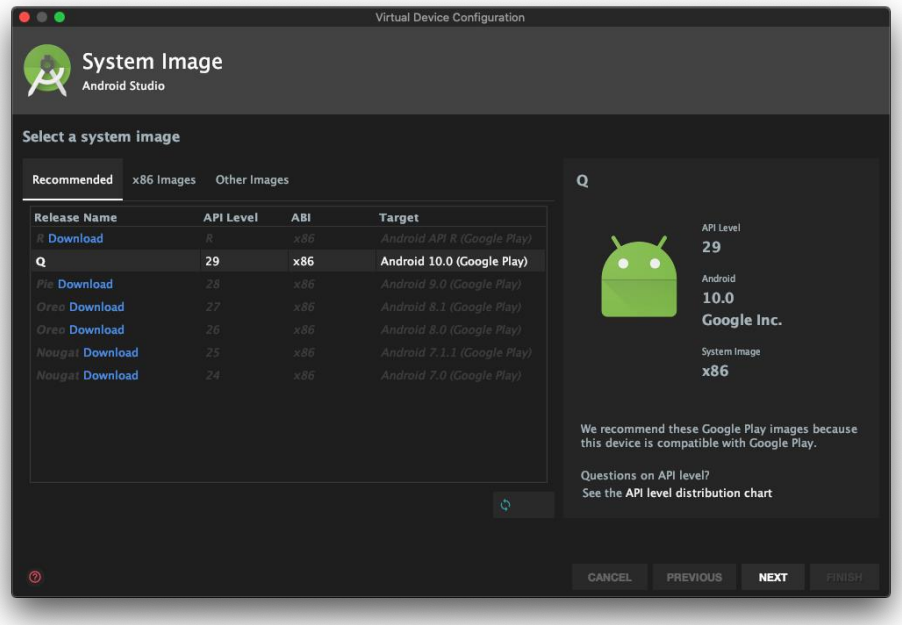

Figura 19. Selecionando a versão do Android para desenvolvimento.

INSTALANDO O GIT

Git é um sistema de controle de versões distribuído, usado principalmente no desenvolvimento de software, mas pode ser usado para registrar o histórico de edições de qualquer tipo de arquivo (Exemplo: alguns livros digitais são disponibilizados no GitHub e escrito aos poucos publicamente). O Git foi inicialmente projetado e desenvolvido por Linus Torvalds para o desenvolvimento do kernel Linux, mas foi adotado por muitos outros projetos.

Cada diretório de trabalho do Git é um repositório com um histórico completo e habilidade total de acompanhamento das revisões, não dependente de acesso a uma rede ou a um servidor central. O Git também facilita a reprodutibilidade científica em uma ampla gama de disciplinas, da ecologia à bioinformática, arqueologia à zoologia.

O Git é um software livre, distribuído sob os termos da versão 2 da GNU General Public License. Sua manutenção é atualmente supervisionada por Junio Hamano. Você poderá baixar o Git para versão Windows pelo link: https://git-scm.com/download/win

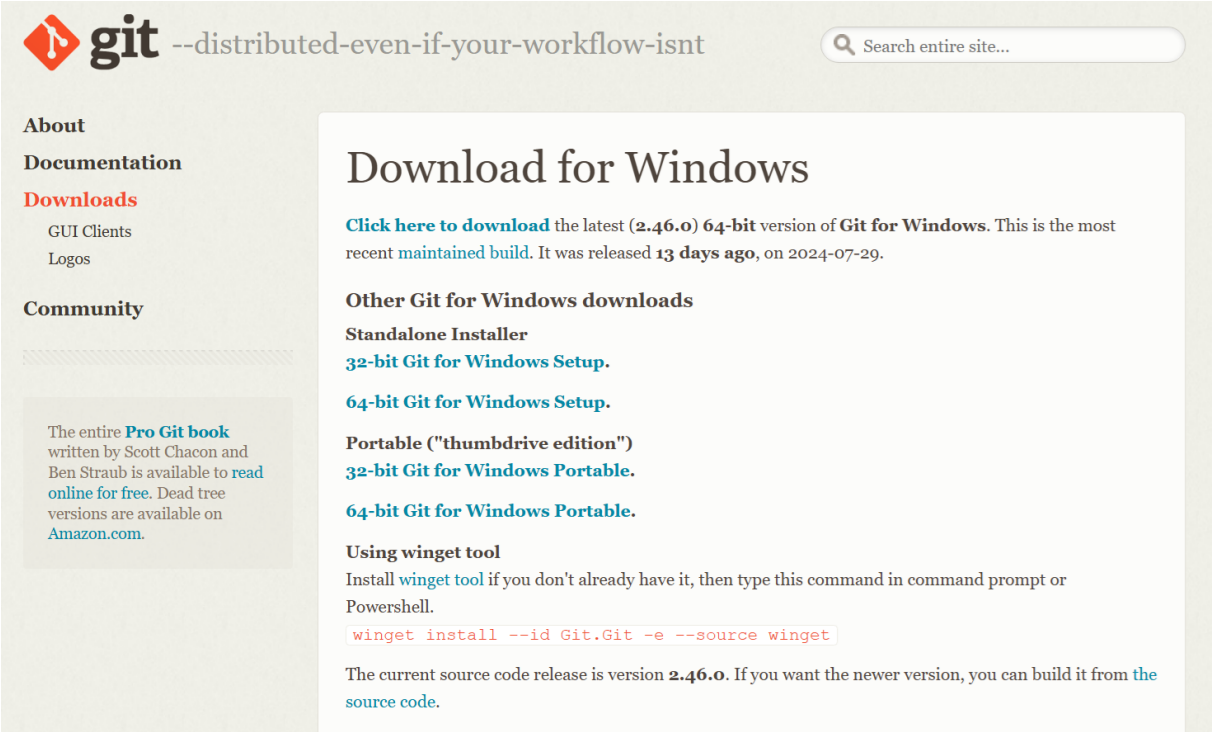

Figura 20. Download do Git.

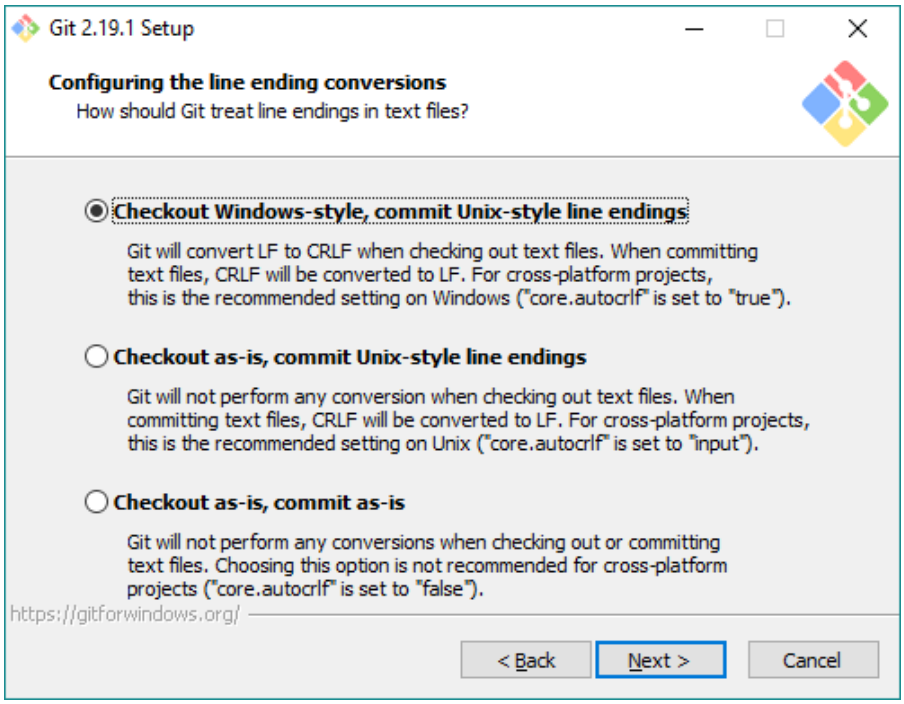

Figura 21. Instalador do Git.

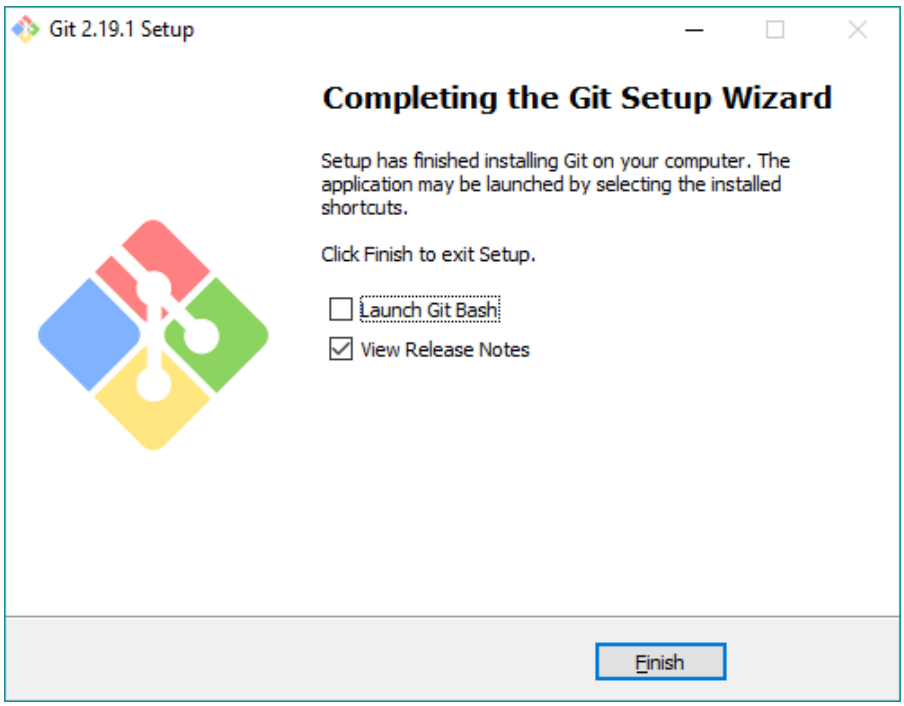

Figura 22. Término da instalação do Git.

DECLARANDO VARIÁVEIS COM A LINGUAGEM DART

Dart é uma linguagem estaticamente tipada, o quer dizer que uma vez que atribuímos um valor a uma variável, valores de outros tipos não poderão ser armazenados por essa mesma variável. Também por esse motivo, Dart é uma linguagem Type Safe e por isso operações estranhas com variáveis, a exemplo de tentar somar caracteres e números, poderão ser alertados pelo compilador e corrigidos antes do programa falhar em tempo de execução.

Em Dart todos os tipos são objetos. Graças a essa característica temos acesso a um amplo conjunto de funções para processamento de dados, mesmo em objetos de tipos mais básicos, fornecidos pelo ambiente de execução do Dart. Esses tipos são:

- **Number**
- **String**
- **Boolean**
- **dynamic**
- **Function**
- **List**
- **Map**

A seguir, falamos um pouco sobre esses tipos, bem como sobre as suas principais funções.

Number

Dart oferece três tipos para armazenar valores numéricos. O primeiro deles é `int`, utilizado para o armazenamento de qualquer número inteiro, seja ele negativo ou positivo. O segundo é `double`, que é utilizado para o armazenamento de números de pontos flutuantes. Ambos, `int` e `double`, são subtipos de num. Ao declarar uma variável como num ela pode ser tanto um inteiro quanto um número de ponto flutuante:

```
num pi = 3;
pi = 3.14;
```

Contudo, ao declarar uma variável como um inteiro, ela não poderá receber um decimal. Vejamos:

```
int pi = 3;
pi = 3.14;

Error: A value of type 'double' can't be assigned to a variable of type 'int'.
```

Quando encontramos um número ou texto no código, como no exemplo acima, esse valor é chamado literal. Um literal é um valor que não precisa de avaliação por parte do compilador para ser atribuído a uma variável. Sabendo disso podemos concluir que de acordo com o literal utilizado, o compilador cria uma instância de num adequada a esse valor. Naturalmente, o mesmo também ocorrerá com o resultado de uma expressão.

Assim como o tipo num, int e double também fornecem diversos métodos e propriedades que podem ser utilizados para a transformação e checagem de dados. Eles também dispõem de capacidades para expressões utilizando os operadores + (adição), - (subtração), * (multiplicação), / (divisão) e outros. O exemplo a seguir demonstra algumas dessas expressões:

```
double pi = 3.14159265359;

int pi_arredondado = pi.floor();

print(pi_arredondado); // 3
```

Quando declaramos uma variável informando o seu tipo não é permitido o uso da palavra reservada var. Podemos, entretanto, utilizando final e const, se necessário.

No código acima inicializamos a variável pi do tipo double com o valor de pi contendo 11 casas decimais. Em seguida utilizamos o método floor() disponibilizado pelo tipo num (do

qual o tipo double deriva) para arredondar o valor de pi para o número inteiro imediatamente menor que pi.

Outros métodos para operações matemáticas estão disponíveis na tabela a seguir:

Método	**Descrição**
abs()	Retorna o valor absoluto do número.
ceil()	Retorna o último inteiro imediatamente superior.
ceilToDouble()	Retorna o último número imediatamente superior com o tipo double.
clamp(num limiteInferior, num limiteSuperior)	Se o número estiver dentro do limite, retorna o número. Se não, retorna o limite o qual ele extrapolou.
compareTo(num outro)	Compara com outro número, retornando 1 quando forem diferentes e 0 quando forem iguais.
floor()	Arredonda o número para o inteiro anterior.
floorToDouble()	Arredonda o número para o número inteiro anterior no tipo double.
remainder(num outro)	Retorna a sobra da divisão com outro número.
round()	Arredonda o número para o inteiro mais próximo.
roundToDouble()	Arredonda o número para o valor inteiro mais próximo no tipo double.
toDouble()	Converte o número para Double.
toInt()	Converte o número para Int.
toString()	Converte o número em uma String.

toStringAsExponential([int digitos])	Converte para string com exponencial.
toStringAsFixed(int decimais)	Converte para String contendo N casas decimais.
toStringAsPrecision(int digitos)	Converte para String contendo N dígitos.
truncate()	Retira as casas decimais, retornando um inteiro.
truncateToDouble()	Retira as casas decimais, retornando um double.

Além dos métodos de transformação, também temos disponíveis diversas propriedades que permitem realizarmos checagens de valores, tais como `isFinite` (retorna `true` quando o número for finito), `isInfinite` (retorna `true` quando o número for infinito) e `isNegative` (retorna `true` quando o número for negativo):

```
double euler = 0.5772156649;

print(euler.isFinite); // true

print(euler.isInfinite); // false

print(euler.isNegative); // false
```

Strings

Strings são cadeias de caracteres que podemos representar com aspas duplas ou aspas simples:

```
String nome_usuario = "Caio";

String sobrenome_usuario = 'Silva';

print(nome_usuario); // Caio

print(sobrenome_usuario); // Silva
```

Assim como os numbers, o objeto String também nos fornece diversos atributos e métodos para a verificação e transformação de cadeias de caracteres, tais como toUpperCase, que transforma os caracteres do texto em maiúsculas, toLowerCase, que transforma todos os caracteres em letras minúsculas, trim, que remove os espaços vazios do início e fim do texto, e muitos outros. Vejamos alguns exemplos:

```
String nome_usuario = "caio";

String sobrenome_usuario = ' SILVA';

print(nome_usuario); // caio

print(sobrenome_usuario); // SILVA

nome_usuario = nome_usuario.toUpperCase();

print(nome_usuario); // CAIO

sobrenome_usuario = sobrenome_usuario.trim();

print(sobrenome_usuario); // SILVA
```

```
nome_usuario = nome_usuario.toLowerCase();

sobrenome_usuario = sobrenome_usuario.toLowerCase();

print(nome_usuario); // caio

print(sobrenome_usuario); // silva
```

Podemos juntar duas strings através da concatenação ou através da interpolação. Com a segunda podemos, inclusive, transformar outros tipos de dados em String. Vejamos um exemplo que utiliza tanto concatenação quanto interpolação de Strings:

```
String nome = "Caio";

String sobrenome = "Silva";

int idade = 22;

// Concatenação de duas Strings

String nome_completo = nome+" "+sobrenome;

//Interpolação com String e valor do tipo inteiro.

String mensagem = "O usuário $nome_completo possui $idade anos de idade.";

print(mensagem); // O usuário Caio Silva possui 22 anos de idade.
```

Diferentemente do Java, por exemplo, que necessitamos utilizar o método equals para realizar a comparação de duas Strings, no Dart podemos compará-las utilizando o operador de igualdade ==:

```
String nome_usuario_1 = "Caio";

String nome_usuario_2 = "Caio";

if(nome_usuario_1 == nome_usuario_2) {

  print("Ambos os usuários possuem o mesmo nome.");

}
```

Boolean

Para representarmos valores booleanos no Dart utilizamos o tipo bool que pode ser representado por dois valores: true (verdadeiro) e false (falso). Quando utilizamos propriedades de checagem vistas anteriormente, como isFinite, isInfinite e isNegative por exemplo, o valor retornado por estas é do tipo bool.

O Dart fornece diversos operadores de comparação com resultados booleanos. Podemos, inclusive, checar por tipos durante a execução do software:

Operador	**Descrição**
>=	Maior ou igual
>	Maior
<=	Menor ou igual
<	Menor

is	É do mesmo tipo
is!	Não é do mesmo tipo
==	Igual
!=	Diferente
&&	E lógico (AND)
||	OU lógico (OR)

Vejamos um exemplo:

```
String papel = "ADMIN";

bool esta_logado = true;

String nome_usuario = "Caio";

if(papel == "ADMIN" && esta_logado) {

  print("O usuário $nome_usuario é um Admin e está logado.");

}
```

dynamic

No Dart existe tipo chamado dynamic, onde podemos atribuir valores de todos os outros tipos, e até mesmo modificar esses valores em tempo de execução. Vejamos:

```
dynamic nome = "Caio Silva";

dynamic idade = 22;

print(nome); // Caio Silva

print(idade); // 22

idade = "22 anos";

print(idade); // 22 anos
```

Function

Uma função é um trecho de código, um processo ou rotina, responsável pela execução de uma tarefa específica, podendo ser executado múltiplas vezes durante a execução do programa. No Dart, funções são um tipo de dado e podemos tratá-las como qualquer outro dado: podemos passá-las como parâmetro, atribuir a variáveis e mais.

Uma função pode receber parâmetros (dados necessários para sua execução) e retornar outros valores gerados a partir de um processamento qualquer. Entretanto, nenhum dos dois é obrigatório: uma função pode não precisar de parâmetros e não retornar nada. Por exemplo:

```
exibirMensagemDeErro(){

  print("Desculpa, encontramos um erro.");

}
```

No trecho de código anterior declaramos nossa função, mas esta não executará de imediato. Para que esta seja executada, precisamos explicitamente ordenar a execução:

```
exibirMensagemDeErro(); // Desculpa, encontramos um erro.
```

Funções também podem ter seu retorno tipado. Se ela não tiver retorno, podemos tipá-la como void:

```
void exibirMensagemDeErro(){

 print("Desculpa, encontramos um erro.");

}
```

Para recebermos parâmetros declararmos cada uma das variáveis que a função pode receber, e seus tipos, dentro dos parênteses:

```
void exibirNomeECargo(String nome, String cargo) {

 print("Nome: $nome ; Cargo: $cargo");

}
exibirNomeECargo("Caio", "Desenvolvedor"); // Nome: Caio ; Cargo: Desenvolvedor
```

No exemplo acima temos um exemplo de passagem de parâmetros obrigatórios, e posicionais (mudar a ordem de passagem dos parâmetros influencia a execução da função). Se um desses parâmetros não for passado na execução da função, um erro será gerado.

No Dart é possível definir parâmetros opcionais posicionais e nomeados, mas não ambos. Vejamos um exemplo de cada.

Parâmetro Opcional Posicional

Para definir parâmetros opcionais posicionais, devemos agrupar esses parâmetros entre colchetes.

Atenção: parâmetros opcionais posicionais precisam ser declarados após os parâmetros obrigatórios:

```
void exibirNomeECargo(String nome, [String cargo]) {

  if(cargo != null) {
    print("Nome: $nome ; Cargo: $cargo");
  } else {
    print("Nome: $nome ;");
  }
}

exibirNomeECargo("Caio", "Desenvolvedor"); // Nome: Caio ; Cargo: Desenvolvedor

exibirNomeECargo("Caio"); // Nome: Caio ;
```

Também temos a opção de declararmos valores padrão para os parâmetros opcionais:

```
void exibirNomeECargo(String nome, [String cargo = "Desconhecido"]) {

  print("Nome: $nome ; Cargo: $cargo");

}

exibirNomeECargo("Caio", "Desenvolvedor"); // Nome: Caio ; Cargo: Desenvolvedor

exibirNomeECargo("Caio"); // Nome: Caio ; Cargo: Desconhecido
```

Parâmetro Opcional Nomeado

A segunda forma de passarmos parâmetros opcionais para uma função se dá através da nomeação destes parâmetros. Nesse caso, no momento em que executamos a função, precisamos conhecer os nomes dos parâmetros que serão recebidos, não sua posição.

Da mesma forma que parâmetros opcionais posicionais, os parâmetros nomeados também precisam ser declarados após a declaração dos parâmetros obrigatórios. Dessa vez, entretanto, utilizamos chaves na declaração:

```
void exibirNomeECargo(String nome, {String cargo}) {

 if(cargo != null) {

  print("Nome: $nome ; Cargo: $cargo");

 } else {

  print("Nome: $nome ;");

 }

}

exibirNomeECargo("Caio", cargo: "Desenvolvedor"); // Nome: Caio ; Cargo: Desenvolvedor

exibirNomeECargo("Caio"); // Nome: Caio ;
```

Para retornar valores de uma função devemos utilizar a palavra reservada return, que retorna o valor gerado e interrompe a execução da função. Neste caso, podemos declarar o retorno da função como o tipo de dado que ela retornará:

```
String gerarMsgNomeECargo(String nome, String cargo) {

```

```
  return "Nome: $nome ; Cargo: $cargo";

}

print(gerarMsgNomeECargo("Caio", "Desenvolvedor")); // Nome: Caio ; Cargo: Desenvolvedor
```

No Dart, podemos tratar uma função como qualquer outro valor. Podemos, por exemplo, atribuir uma função a uma variável e passá-la como parâmetro para outra função, que executará essa variável:

```
Function printarErro = (String erro) {

  print("Encontramos um erro: $erro");

};

void checarIgualdade(String valor1, String valor2, Function callback) {

  if(valor1 != valor2){

    callback("Os valores são diferentes");

  }

}

checarIgualdade("Caio", "Caio", printarErro); // Não printa nada

checarIgualdade("Caio", "Silva", printarErro); // Encontramos um erro: Os valores são diferentes
```

Funções que executam apenas um comando também podem ser escritas da seguinte forma:

```
exibirMsg(String msg) => print(msg);

exibirMsg("Essa é minha mensagem."); // Essa é minha mensagem.
```

List

Quando necessitamos trabalhar com arrays (estrutura de dados que armazena valores identificador a partir de um index), podemos contar com o objeto List. Uma lista pode ser criada de duas formas: através da criação de uma instância do List ou criando uma lista literal:

```
// Criando instância do List

var lista_usuarios = List(); // Também poderia ser new List()

// Usando o método add do objeto para adicionar novos valores a lista

lista_usuarios.add("Caio");

lista_usuarios.add("Aylan");

lista_usuarios.add("Estêvão");

// Lista literal

var lista_usuarios_literal = ["Caio", "Aylan", "Estêvão"];
Os exemplos anteriores possuem resultados idênticos:
print(lista_usuarios[0]); // Caio

print(lista_usuarios[1]); // Aylan

print(lista_usuarios[2]); // Estêvão

print(lista_usuarios_literal[0]); // Caio
```

```
print(lista_usuarios_literal[1]); // Aylan

print(lista_usuarios_literal[2]); // Estêvão
```

Perceba que utilizamos [número] após a variável para recuperar os valores. Esse número, que chamamos de index, é a posição dos valores na lista (que começa em 0).

Assim como outros tipos, também podemos realizar a tipagem de variáveis que recebam uma List. Mais do que isso: também podemos especificar o tipo dos valores que uma lista pode receber:

```
List lista_usuarios = List();

lista_usuarios.add("Caio");

lista_usuarios.add("Aylan");

lista_usuarios.add(10); // Gerará um erro de compilação
```

No Dart, um List é um genérico. Isto é: passamos o tipo do dado que ele armazenará como parâmetro.

O objeto List nos fornece diversos métodos e propriedades que facilitam a interação, manipulação e a análise de arrays. Podemos correr a lista utilizando o método forEach, modificar os dados da lista com o método map, filtrar com o método where, contar o número de valores com o atributo length e muito mais:

```
List lista_numeros = List();

lista_numeros.add(1);

lista_numeros.add(2);
```

```
lista_numeros.add(3);

lista_numeros.add(4);

lista_numeros.add(5);

lista_numeros.forEach((numero) {

  print("Número: $numero");

});



// Resultado

// Número: 1
// Número: 2
// Número: 3
// Número: 4
// Número: 5
```

No exemplo anterior utilizamos o método forEach, que iterará o array e executar a função de callback uma vez para cada valor. Podemos combinar diversos métodos para gerar diferentes resultados:

```
lista_numeros.map((numero) {

  return numero * 10;

}).toList().forEach((numero) {

  print("Número: $numero");

});


// Resultado

// Número: 10
// Número: 20
// Número: 30
// Número: 40
// Número: 50
```

Dessa vez, utilizamos o método map para modificar cada um dos valores da List e, ao final do processo, exibimos o valor de cada um dos números na tela com o método forEach. Podemos utilizar o atributo length para saber a quantidade de valores em uma List:

```
print("Quantidade de valores: ${lista_numeros.length}"); // Quantidade de valores: 5
```

Também é possível iterar um array, como o método forEach faz, utilizando estruturas mais familiares como o for, por exemplo:

```
for(int index = 0; index < lista_numeros.length; index++) {

  print("Linha ${lista_numeros[index]}");

}
```

Map

Um Map é uma estrutura similar à uma List. Entretanto, diferentemente da List onde cada valor tinha um index (número inteiro) correspondente, em um Map cada valor terá uma chave (que pode ser um objeto qualquer) correspondente. Assim como a List, o Map também pode ser criado através da criação de uma instância ou de forma literal.

Atenção: se criarmos um Map de forma literal, a chave só pode ser do tipo String.:

```
var nome_sobrenome = {

  'Caio': 'Silva',

  'Aylan': 'Boscarino',

  'Estêvão': 'Dias',

};


print(nome_sobrenome['Caio']); // Silva

print(nome_sobrenome['Aylan']); // Boscarino

print(nome_sobrenome['Estêvão']); // Dias
```

Se criarmos uma instância, entretanto, qualquer objeto pode ser utilizado como chave:

```
var chave_valor = Map();

chave_valor[10] = 'Valor 1';

chave_valor[true] = 'Valor 2';

chave_valor["Chave"] = 'Valor 3';

print(chave_valor[10]); // Valor 1

print(chave_valor[true]); // Valor 2

print(chave_valor["Chave"]); // Valor 3
```

No Dart tudo é um objeto, e isso abre incríveis possibilidades. Poderíamos, por exemplo, sobrescrever métodos e operadores (como o operador == de igualdade) de forma a adequar o comportamento destes de acordo com as necessidades de nossa aplicação. List e Map são essenciais de qualquer aplicação construída com o Dart, e seus métodos de modificação e filtragem podem transformar algoritmos complexos em algumas linhas de código.

ESTRUTURAS DE CONDIÇÃO NO DART

Estruturas de condição são artifícios das linguagens de programação para determinar qual bloco de código será executado a partir de uma determinada condição. No Dart, assim como em outras linguagens, podemos trabalhar com as estruturas de condição utilizando o if/else e o switch/case como veremos abaixo:

if/else

O **if** e o **else** são comandos que verificam determinada condição na programação. O uso do if em um programa em Dart visa verificar se determinada ação é verdadeira e executar o bloco de código contido em seu escopo. Basicamente é feita da seguinte forma:

Exemplo:

```
void main() {
  var idade = 18;
  if (idade < 18) {
    print("Você é menor de idade");
  }
}
```

No código acima, estamos declarando uma variável chamada idade que recebe a idade de um usuário. Com a estrutura if, podemos verificar se o valor da variável idade é menor que 18. Caso positivo, exibimos a mensagem “Você é menor de idade”.

Já o uso do if/else fará com que uma das ações sejam executadas, já que se a condição dentro do if não for verdadeira, será executado o código contido no else. O if/else irá testar caso a condição seja verdadeira e executar uma determinada ação ou caso a mesma não seja, executar outra.

Exemplo:

```
void main() {
  var idade = 18;
  if (idade < 18) {
    print("Você é menor de idade");
  } else {
    print("Você é maior de idade");
  }
}
```

switch/case

Por sua vez, o switch/case na programação, da mesma forma que o if também tem a função de verificar se determinada ação é verdadeira e executar o bloco de código contido em seu escopo, controlando assim o fluxo do programa. O comando switch compara o valor de uma variável aos valores que serão especificados nos comandos case.

Exemplo:

```
void main() {
  var opcao = 2;
  switch (opcao) {
     case 1:
       print("Cadastrar");
       break;
     case 2:
       print("Listar");
       break;
     default:
       print("Sair");
    }
}
```

No código acima, estamos avaliando o valor da variável opcao dentro do switch. Caso o valor seja 1, executamos o bloco de código contido no case e exibimos a mensagem “Cadastrar”. Caso o valor seja 2, executamos o bloco de código contigo no outro case para exibir a mensagem “Listar”. Caso a opção não esteja definida nos “cases”, executamos o bloco de código contigo no default e exibimos a mensagem “Sair”.

ESTRUTURAS DE REPETIÇÃO NO DART

Estruturas de repetição são artifícios das linguagens de programação para executar um determinado bloco de código por uma certa quantidade de vezes, seja ela definida (utilizando o for) ou a partir de uma condição (utilizando o while).

for

A estrutura de repetição for executará um determinado bloco de código por um número definido de vezes. Esta estrutura é muito útil quando já sabemos a quantidade de vezes que precisamos executar determinado bloco de código.

Exemplo:

```
void main() {
  for (int numero = 1; numero <= 10; numero++){
    if (numero % 2 == 0) {
      print(numero);
    }
  }
}
```

No exemplo acima, estamos criando um intervalo de 1 a 10 e verificando os números dentro deste intervalo que são pares (resto da divisão deste número por 2 é igual a 0). Com o for, basta definir o início e fim deste intervalo e o valor a ser incrementado. No caso do exemplo acima, estamos iniciando o intervalo por 1 e executando-o enquanto o número seja menor ou igual a 10, além de incrementar seu valor em uma unidade a cada execução.

while

O while é uma estrutura de repetição que permite executar um determinado bloco de código enquanto uma condição for verdadeira. É muito similar ao if, com a diferença que o bloco será executado enquanto a condição for verdadeira, e não se a condição for verdadeira.

Para isso, após o comando while precisamos definir uma condição que será testada a cada execução do seu loop. O while só será finalizado quando essa condição não for mais atendida.

Exemplo:

```
void main() {
  var numero = 1;
  while (numero <= 10) {
    if (numero % 2 == 0) {
      print(numero);
    }
  numero++;
  }
}
```

No exemplo acima, estamos declarando uma variável chamada “numero” e utilizamos a estrutura while para executar o bloco contido em seu escopo enquanto a variável seja menor ou igual a 10. Além disso, para cada número percorrido, verificamos se o resto da sua divisão por 2 é igual a 0, caso positivo, este número é par. Por fim, incrementamos o valor desta variável para não cairmos em um “loop infinito”.

do/while

O do/while é uma estrutura de repetição que terá quase o mesmo funcionamento que a estrutura de repetição while, porém, a diferença entre eles é que no do/while o comando será executado ao menos uma única vez.

Exemplo:

```
void main() {
  var numero = 1;

  do {
   if (numero % 2 == 0) {
     print(numero);
   }
   numero++;
  } while (numero <= 10);
}
```

Seguindo o mesmo exemplo das estruturas anteriores, com o do while, a primeira execução sempre é feita, já que o teste só é realizado no final da estrutura. Deste modo, sempre que precisarmos de uma estrutura de repetição que execute, no mínimo, a primeira vez, a melhor escolha é o do while.

CRIANDO UM APLICATIVO COM FLUTTER

Neste exemplo usaremos o *Visual Studio Code*. Com o SDK instalado e adicionado à variável de ambiente "Path", criar um novo projeto Flutter é extremamente simples. Para isso, no terminal do Windows, navegue até o diretório onde deseja iniciar o seu projeto e execute o seguinte comando:

```
flutter create meu_primeiro_app
```

O parâmetro meu_primeiro_app é o nome da nossa aplicação. Sinta-se livre para colocar o nome que quiser.

O Flutter ficará encarregado de criar todos os arquivos da nossa aplicação. Feito isso, ainda no terminal, navegue até o diretório que foi criado pelo CLI:

```
cd meu_primeiro_app
```

Para abrir o projeto no Visual Studio Code basta executar o comando:

```
code .
```

EXECUTANDO O APLICATIVO FLUTTER EM UM EMULADOR ANDROID

O Android Studio, além do Android SDK, também nos fornece um emulador Android que contém todas as suas versões. Dessa forma, podemos testar o comportamento do nosso aplicativo em diversos dispositivos, ao mesmo tempo que elimina a necessidade de conectarmos um dispositivo físico para realizar testes, embora essa opção ainda seja possível.

No Visual Studio Code, abra o arquivo lib/main.dart. Ele será o ponto de entrada do nosso aplicativo, ou seja, toda aplicação será iniciada através da função main no início do arquivo. Todo o código que você vê abaixo dessa função é o aplicativo que o Flutter oferece como exemplo. Por enquanto não realizaremos nenhuma modificação, apenas executaremos este aplicativo. Para isso, com o arquivo aberto, pressione:

```
CRTL + F5
```

No topo superior do Visual Studio Code abrirá uma pequena aba. Se você já possui emuladores configurados, eles devem aparecer nessa aba. Se não, basta selecionar a opção “Create New”:

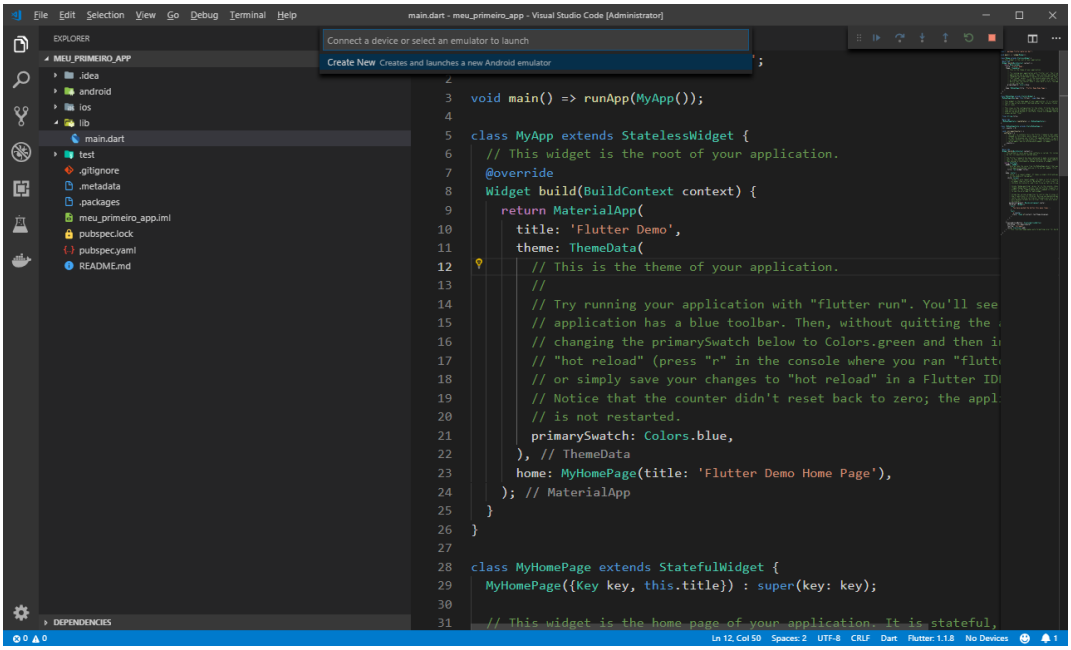

Figura 23. Executando o emulador Android a partir do Visual Studio Code.

O processo deve demorar alguns minutos, e você pode acompanhá-lo através da pequena aba aberta no canto inferior direito:

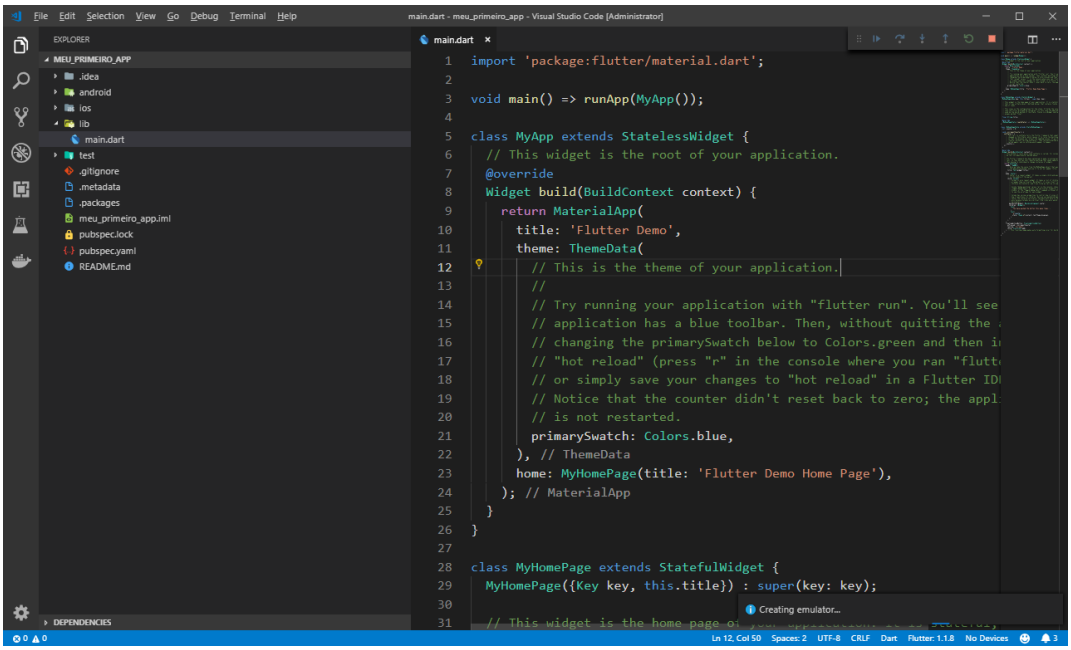

Figura 24. Processo de carregamento do emulador.

Ao final do processo, quando o emulador for automaticamente aberto, pode ser que seu aplicativo não seja executado. Dessa forma, no Visual Studio Code, pressionamos "CRTL + F5" novamente. Entretanto, como o emulador já estará aberto, não será necessário escolher nenhuma opção. O Flutter ficará responsável por iniciar a aplicação no emulador e emitir todos os eventos e erros através da aba "DEBUG CONSOLE" no rodapé do Visual Studio Code:

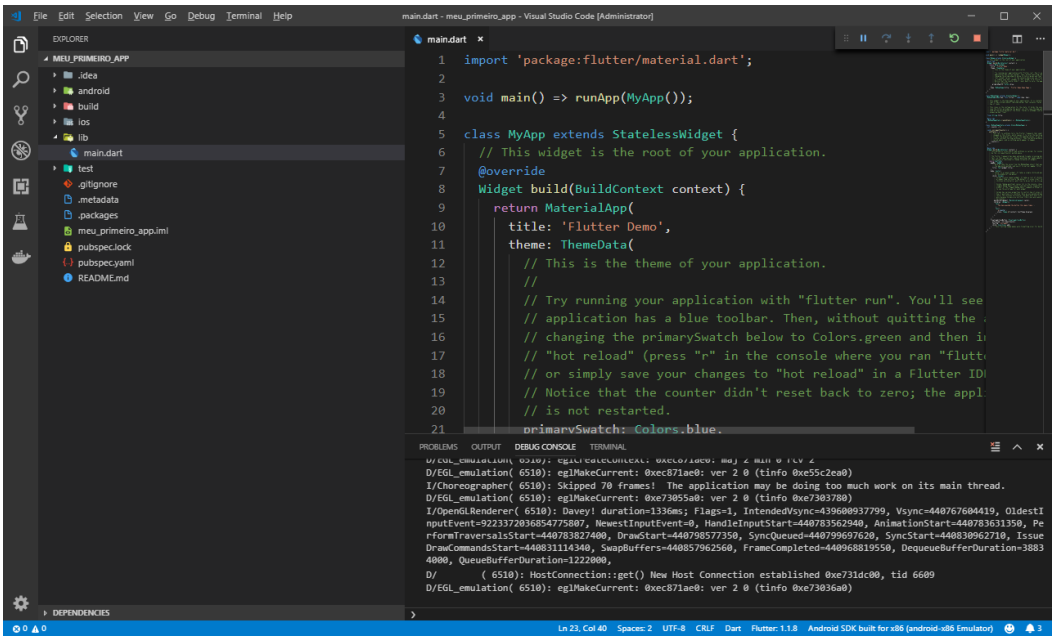

Figura 25. Debug Console do Flutter.

Nesse momento, no emulador, seu aplicativo já será executado:

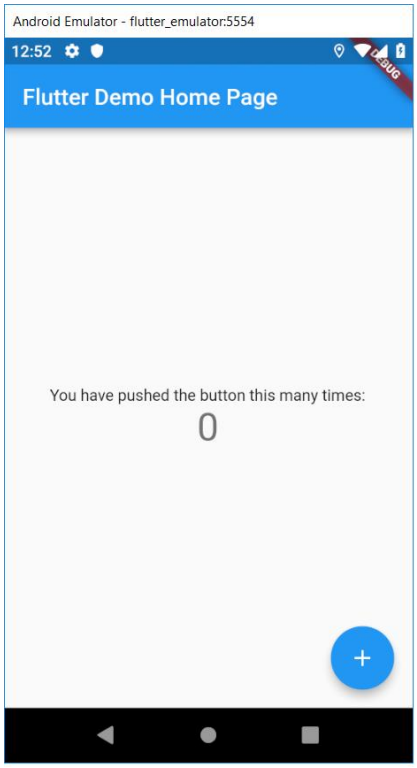

Figura 26. Aplicativo padrão criado automaticamente pelo comando flutter create.

E é isso ai! Você acabou de entrar no mundo do desenvolvimento multiplataforma nativo. Que tal darmos uma explorada no aplicativo de exemplo que foi criado?

WIDGETS! EXPLORANDO UM APLICATIVO FLUTTER

Tudo no Flutter é um Widget. Seja uma página, um botão ou um texto. Sempre que quisermos exibir qualquer tipo de elemento para o usuário, usaremos Widgets.

O Flutter fornece diversos Widgets que são essenciais para o funcionamento da nossa aplicação. Por isso, no arquivo main.dart onde podemos encontrar todos os elementos do nosso aplicativo de exemplo, logo na primeira linha, encontramos a importação dos Widgets do Material Design:

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

Mais abaixo encontramos o Widget superior da aplicação. É nele, chamado de MyApp, que seremos capazes de definir um esquema de cores padrão, bem como o título e a página inicial do nosso aplicativo. Perceba também que a função main, a "porta de entrada" cria uma instância desse Widget, iniciando o aplicativo:

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

context é uma instância de BuildContext que serve para informar ao Widget em que lugar ele se encontra dentro da árvore de Widgets da aplicação.

Repare que na **linha 9**, no parâmetro primarySwatch do construtor da classe ThemeData, passamos o valor Colors.blue. A classe Colors é fornecida pelo Flutter Material, onde encontramos diversas cores comumente utilizadas pelo Material Design. Que tal mudarmos a cor primária do nosso aplicativo para vermelho?

```
ThemeData(
    primarySwatch: Colors.red,
),
```

É só salvar e, em um passe de mágica, o recurso de Hot Reload do Flutter modificará nosso aplicativo executado no emulador:

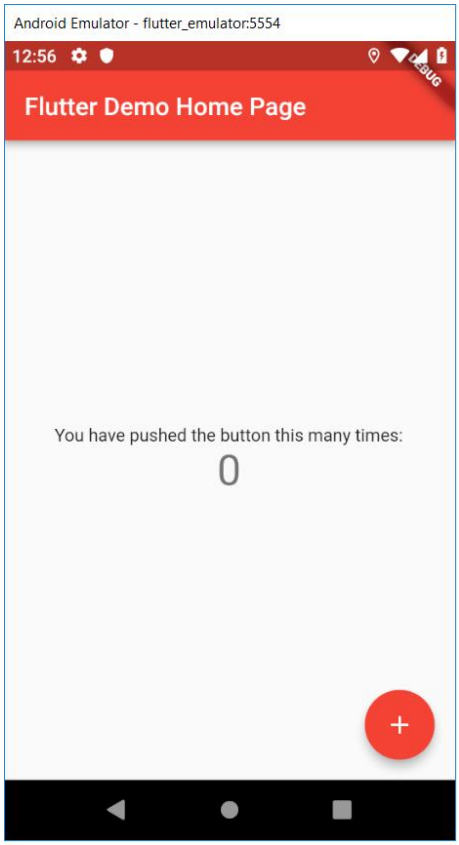

Figura 27. Aplicativo padrão pintado de vermelho.

Mais abaixo encontraremos outras duas classes: MyHomePage e _MyHomePageState. Diferentemente do Widget superior da aplicação, que se tratava de um StatelessWidget (um Widget que não armazena dados), o Widget da nossa página inicial se trata de um StatefulWidget (um Widget que armazena dados e possui Lifecycle). Por isso possuímos duas classes: a primeira, chamada MyHomePage, conterá apenas os dados imutáveis do nosso Widget, ou seja, apenas os dados que forem passados como parâmetro para o Widget. É essa classe que instanciamos dentro do Widget MyApp para abrir a página inicial.

```
class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) : super(key: key);

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}
```

A segunda classe, `_MyHomePageState`, é onde armazenamos os dados mutáveis do componente (estado) e implementamos o método `build` utilizado pelo Flutter para criar o Widget na tela do usuário:

```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {

    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
```

```
          const Text(
            'You have pushed the button this many times:',
          ),
          Text(
            '$_counter',
            style: Theme.of(context).textTheme.headline4,
          ),
        ],
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ),
  );
 }
}
```

Perceba que é nesse Widget que encontramos todo o código da nossa aplicação responsável por exibir o contador e, ao clique do botão de incremento, realizamos a atualização dos dados. Agora que alteramos a cor do nosso aplicativo, que tal implementar mais uma funcionalidade? Apenas um botão de incremento não é suficiente, também precisamos decrementar!

Ainda na classe `_MyHomePageState` criaremos mais um método chamado `_decrementCounter`. Este método ficará responsável por decrementar o atributo `_counter` da nossa aplicação e por executar o método setState disponível em todo Widget do tipo Stateful:

```
...
void _decrementCounter() {
  setState(() {
    _counter--;
  });
 }
...
```

Por fim precisaremos, no atributo floatingActionButton do Widget Scaffold, adicionar mais um botão ao lado do botão de incremento. Faremos isso agrupando ambos os botões no Widget Row, também disponibilizado pelo Flutter:

```
floatingActionButton: Row(
   mainAxisAlignment: MainAxisAlignment.end,
   children: <Widget>[
     FloatingActionButton(
       onPressed: _incrementCounter,
       tooltip: 'Increment',
       child: const Icon(Icons.add),
     ),
     const SizedBox(
       width: 10.0,
     ),
     FloatingActionButton(
       onPressed: _decrementCounter,
       tooltip: 'Decrement',
       child: const Icon(Icons.remove),
     ),
   ],
),
```

Salve as alterações e agora conseguimos mudar a cor de tema da aplicação, adicionando uma nova funcionalidade em alguns poucos passos.

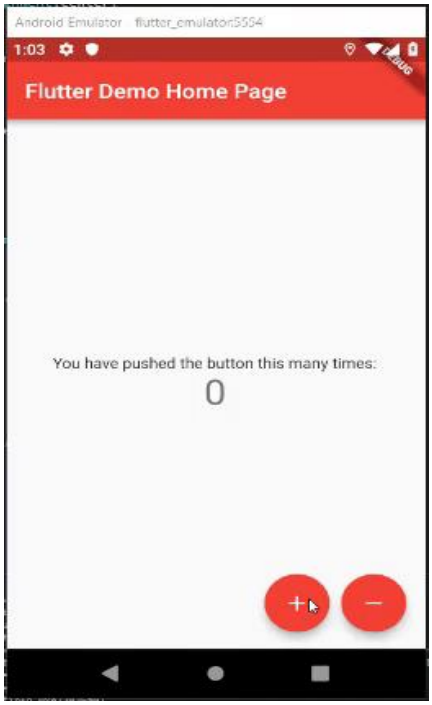

Figura 28. Aplicativo aperfeiçoado com decremento do contador.

Veja abaixo o código completo utilizado neste artigo:

```
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.red,
    ),
    home: const MyHomePage(title: 'Flutter Demo Home Page'),
   );
  }
}

class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) : super(key: key);

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  void _decrementCounter() {
   setState(() {
     _counter--;
   });
  }

  @override
  Widget build(BuildContext context) {

    return Scaffold(
```

```
  appBar: AppBar(
    title: Text(widget.title),
  ),
  body: Center(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        const Text(
          'You have pushed the button this many times:',
        ),
        Text(
          '$_counter',
          style: Theme.of(context).textTheme.headline4,
        ),
      ],
    ),
  ),
  floatingActionButton: Row(
    mainAxisAlignment: MainAxisAlignment.end,
    children: <Widget>[
      FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
      const SizedBox(
        width: 10.0,
      ),
      FloatingActionButton(
        onPressed: _decrementCounter,
        tooltip: 'Decrement',
        child: const Icon(Icons.remove),
      ),
    ],
  ),
```

```
    );
  }
}
```

Além dos recursos apresentados aqui, o Flutter fornece centenas de Widgets, que agilizam muito o processo de desenvolvimento.

REFERÊNCIAS:

https://pt.wikipedia.org/wiki/Dart_(linguagem_de_programa%C3%A7%C3%A3o)

https://dartpad.dev/

https://pt.wikipedia.org/wiki/Git

https://pt.wikipedia.org/wiki/Flutter

https://www.devmedia.com.br/sintaxe-dart-tipos-nao-tao-primitivos/40368

https://www.treinaweb.com.br/blog/estruturas-condicionais-e-de-repeticao-no-dart

https://www.devmedia.com.br/hello-world-com-flutter/40321