Flutter zero to hero edition2023

short line

SUBHASH CHANDRA SHUKLA  
08th Feb, 2023

# 

[Preface](#_iktib7zbq90l)

[About author](#_61jslgd7ke9y)

[Introduction](#_4lqp25cx7kth)

[Dart basics](#_ovawxjoi07ia)

[About dart:](#_d61wgislxrex)

[History of dart:](#_oshtr9pq1a8i)

[Features of dart:](#_26t3s6dbft4c)

[Dartpad:](#_q49nk0zc2exb)

[Variables in Dart:](#_ipq22pqr67fg)

[built-in Dart:](#_5w8v8edk8lmr)

[Functions in dart:](#_x5342gw1o569)

[Types of Functions in dart:](#_amjvrkfj6jw4)

[Operators in dart:](#_w6yvvnn6wpy8)

[Control flow statements in dart:](#_yavx3o2obfz)

[String in dart](#_mnozb9g95h92)

[Environment setup](#_lzr2si3oshbv)

[Flutter CLI -](#_kir2i8gnxr6j)

[Widgets](#_4xv296155sml)

[Stateless widgets((Non-Mutable Data):](#_r8saa7ijv7n3)

[Stateful widgets(Mutable Data):](#_wnaabukas7f9)

[Navigation in flutter:](#_8y6twbwzu5ch)

[1. Navigator widget:](#_lztttsplm6mc)

[2. Named routes:](#_gtjnio3cxd8z)

[3. BottomNavigationBar widget:](#_4ot8crx8hi5l)

[4. TabBar widget:](#_3d8fi5b8sog3)

[Working with assets:](#_ap9rnjv3cs5j)

[Flutter design-principles:](#_spsnln3xsr1r)

[Dependency - Injection:](#_85eocuvna6og)

[Flutter design-patterns-](#_7ma60a4aodkn)

[The Model-View-Controller (MVC) pattern :](#_c11xhfupzc9t)

[● The Model-View-ViewModel (MVVM) pattern:](#_nzxuc5do4g01)

[● The Redux pattern:](#_3gb808nk8nnj)

[● The Scoped Model pattern:](#_rygsa3mx25et)

[SOLID PRINCIPLES](#_n399igrcjt91)

[OOP’S in Dart:](#_yna09nzi0i9z)

[Classes and Objects:](#_xssvtjisp1uh)

[Constructor in dart:](#_9fhei2fxi1pd)

[this and super keyword in dart:-](#_gexile3nao45)

[Encapsulation:-](#_3nmnuc8rz63i)

[Inheritance:](#_be4ufcpsjxqs)

[Polymorphism:](#_11usosvp5xht)

[Abstraction:](#_5lba0qswe0j0)

[Package Manager in flutter:](#_kso0qe9g8572)

[Storage in flutter:](#_78z0aff60h95)

[Shared Preferences (Key-Value) Storage:](#_y763mrjj31o8)

[SQFLite Storage:](#_mm8nqt8a8an8)

[Firebase Storage:](#_b92b6m8zpejn)

[Working with API’s -](#_mms96k6353nr)

[Flutter JSON Parsing:](#_nr6i729t1oec)

[Flutter http package:](#_nf4upwafdvf9)

[Connect to a WebSocket Server:](#_o3dryx4gi57k)

[Listen to messages from the server:](#_ufp9x6wz64ws)

[Send Data to the Server:](#_iz1vebqmyv8y)

[Close the Connection:](#_edd572v73ec)

[Advance dart:](#_5p4nmb3sgqnp)

[Core libraries:](#_q1mzntm1f20j)

[Streams in dart:](#_l1g3wf36zmxn)

[Future in dart:](#_giordxak3l2o)

[Collections in dart:](#_ljc5yr18hy12)

[List in dart:](#_u4mkd0z3zrs8)

[Map in dart:](#_dtfl06inqvgu)

[Sets in dart:](#_g8nlrmk0046j)

[Lambdas in dart:](#_w7td9x1gfvgm)

[Functional programing in dart:](#_2pq81supjvvi)

[Immutability:](#_qj6wv0efq9s0)

[Recursion:](#_7e9qlk750b9f)

[Isolates in dart:](#_f3q7l4corqdh)

[async() and await in dart():](#_2dztc9xyt03a)

[Flutter State Management:](#_7g1d6jrcoso)

[What is the State?](#_ix5jmq609iha)

[Provider-](#_qd15gcqu7ebn)

[Cubit-](#_p0km0kvs4bbu)

[Riverpod-](#_60g3chh8e9cw)

[Animation in flutter:](#_huh5y6yczzil)

[Tween Animation](#_ldok0zcq0018)

[Physics-based Animation](#_9yac548lmv5s)

[Ticker:](#_8i92lg7epymx)

[Animation:](#_7d3v8my8ohgq)

[Animation Controller:](#_qrfrck1qapag)

[Curved Animation:](#_nc3dorge897)

[Syntax:](#_lftvrvdfbcsh)

[Hero Animation:](#_b834046q177v)

[Opacity Animation:](#_dv5yuf5cbirp)

[Testing in flutter:](#_g896wt4meqzb)

[● Unit tests](#_5dr1erw9stjn)

[● Widget tests](#_ma82cifu0r8w)

[● Integration tests](#_mk37cygjhu0f)

[Reactive programing in dart:](#_1bhobbxn5quf)

[RxDart:](#_g08k1x9y531c)

[Flutter internal](#_nfk7v1gcfrkk)

[Renderer engine in flutter:](#_5cgb0z9vgm1n)

[Flutter app deployment](#_34ctye71lj01)

[Deploying to Android:](#_avm5xqjpjrbp)

[Deploying to iOS:](#_jnl07gumqp2g)

# Preface

**Flutter** is a mobile app development framework that was created by Google to help developers build beautiful and performant native apps for iOS and Android from a single codebase. With Flutter, developers can create high-quality user interfaces, fast and easy app development, expressive and flexible designs, and a great development experience.

Flutter uses the Dart programming language, which was also created by Google, and includes a wide range of pre-built widgets and tools that make app development faster and more efficient. With its hot-reload feature, developers can make changes to their code and see the results in real-time, which helps to speed up the development process and improve productivity.

This book is designed to provide an introduction to Flutter and teach developers how to build their own mobile apps using this powerful framework. We will cover the basics of Flutter, including how to set up a development environment, how to use pre-built widgets, and how to create custom widgets. We will also explore more advanced topics, such as how to handle user input, how to manage state in your app, and how to create responsive designs.

Whether you are an experienced mobile app developer or a beginner just getting started, this book is designed to provide you with the knowledge and skills you need to build beautiful and performant native apps using Flutter. So let's get started!

# About author

Subhash Chandra Shukla is a Flutter developer from India who has experience in building mobile applications using Flutter. Based on his LinkedIn profile, he has been working as a Flutter developer for over 3 years and has built several apps using Flutter. He has also contributed to open-source Flutter packages and libraries.

If you have any specific questions about Subhash Chandra Shukla's experience or his work in Flutter, you can refer to his LinkedIn profile or reach out to him directly for more information.

<https://www.linkedin.com/in/subhashcs/>

# Introduction

Welcome to **"Flutter Zero to Hero"!** This book is designed to take you from a beginner to a proficient **Flutter developer in 2023**. Whether you're new to programming or an experienced developer, this book will provide you with the knowledge and skills to build beautiful, responsive, and robust mobile applications using Flutter.

Flutter is a powerful, open-source, cross-platform development framework developed by Google. It allows you to build high-performance, native mobile applications for iOS and Android from a single codebase. With its modern and flexible architecture, Flutter has gained significant popularity in the mobile development industry, making it a valuable tool for any developer seeking to build mobile applications in 2023 and beyond.

In this book, you'll learn the fundamental concepts of Flutter, such as how to set up your development environment, how to build user interfaces using widgets, how to work with stateful and stateless widgets, and how to handle user input. You'll also learn about the core concepts of building a mobile application, such as working with APIs, using local storage, and implementing authentication and authorization.

Throughout the book, you'll build several applications from scratch, each with a unique set of features and challenges. By the end of the book, you'll have the knowledge and experience to build a variety of mobile applications, from simple to complex, with confidence.

If you're ready to become a proficient Flutter developer in 2023, then let's get started!

# 

# 

# Dart basics

## About dart:

Dart is a client-optimized programming language for building fast and high-performing web, server, and mobile apps. It was developed by Google and first released in 2011. Dart is statically typed, object-oriented and supports both imperative and functional programming paradigms. Dart can be used for both front-end (web) and back-end (server-side) development, and it can be compiled to JavaScript for better performance on the web. It has a modern syntax that is easy to learn and provides features such as async-await, generics, and mixins, which make it well suited for complex applications. Dart also comes with a rich set of libraries and tools, such as the Flutter UI framework, which allows developers to build beautiful and high-performing native apps for iOS and Android. Additionally, Dart has a large and growing community of developers who contribute to the language and its ecosystem. In summary, Dart is a versatile and efficient language that is well-suited for developing a wide range of applications, from simple scripts to complex systems.



## History of dart:

Dart is a programming language developed by Google and first released in 2011. It was created as a replacement for JavaScript and aimed to address some of the challenges that Google faced when developing large-scale, complex web applications. Dart was designed to be fast, scalable, and flexible, with a syntax that is easy to learn for developers who already have experience with other programming languages. It also provides features that are not available in JavaScript, such as strong typing, asynchronous programming, and a comprehensive standard library. In 2013, Google released Flutter, a UI framework for building high-performance, natively compiled apps for iOS and Android, which uses Dart as its primary programming language. Since then, Dart has become increasingly popular among developers, especially for building cross-platform apps with Flutter. Today, Dart is used by many companies and organizations around the world, including Google, Alibaba, and Capital One, and has a growing community of developers and contributors. In conclusion, Dart has a rich history of innovation and evolution, and continues to be a powerful and versatile programming language that is well-suited for modern app development.

## Features of dart:

Dart is a feature-rich programming language with several important features that make it well-suited for modern app development:

* **Statically typed:** Dart is statically typed, which means that type-related information is checked at compile-time, helping to prevent common errors and making code easier to maintain.
* **Object-Oriented:** Dart is an object-oriented programming language, which means that it has support for classes, objects, inheritance, and other OOP concepts. Supports functional programming: Dart also has features that support functional programming, such as first-class functions and higher-order functions.
* **Asynchronous programming:** Dart has built-in support for asynchronous programming with the async and await keywords, making it easy to write fast and efficient code that runs concurrently.
* **Interoperable with JavaScript:** Dart code can be compiled to JavaScript, making it possible to run Dart code in any browser that supports JavaScript.
* **Strong type inference:** Dart has strong type inference, which means that the language can often infer the types of variables and objects automatically, reducing the amount of type information that needs to be explicitly declared. Rich
* **libraries and tools:** Dart comes with a rich set of libraries and tools, including the Flutter UI framework, which makes it easy to build beautiful, high-performing apps for multiple platforms.
* **Fast and efficient:** Dart is designed to be fast and efficient, making it well-suited for both client-side and server-side development. Overall, these features make Dart a versatile and powerful programming language that is well-suited for developing modern apps and systems.

## Dartpad:

DartPad is a free, online tool for writing and running Dart code. It allows developers to try out and experiment with Dart programming language in their web browser, without the need to install any software or tools. DartPad provides a simple and convenient platform for learning, exploring and testing Dart language features, libraries, and APIs.



[Link - https://dartpad.dev/](https://dartpad.dev/)

## Variables in Dart:

Variables are used to store values. They are declared using keywords such as var, final, or const.

**var:** Declares a variable whose type can be inferred from the value it is assigned. The type can be changed later in the code.

**final:** Declares a variable whose value can be assigned only once. The type of the variable can be inferred from the value it is assigned.

**const:** Declares a constant variable whose value must be known at compile-time. The type of the variable can be inferred from the value it is assigned.

Here's an example of variable declaration in Dart:

var name = "Subhash";

final dateOfBirth = "1990-01-01";

const num = 42;

## built-in Dart:

Dart has the following built-in data types:

In Dart, there are several built-in types that can be used to represent data:

1. **Numbers:** Dart has two types of numbers, int and double, used to represent integers and floating-point numbers, respectively.

int age = 30;

double height = 1.78;

1. **Strings:** A string is a sequence of characters and is used to represent text.

String name = "Subhash Chandra";

1. **Booleans:** A boolean value is either true or false and is used to represent a logical value.

bool isValid = true;

1. **Lists:** A list is an ordered collection of values and is written as a comma-separated sequence of values surrounded by square brackets.

List<int> numbers = [1, 2, 3, 4, 5];

1. **Maps:** A map is a collection of key-value pairs and is written as a set of key-value pairs surrounded by curly braces.

Map<String, int> marks = {

'English': 80,

'Math': 90,

'Science': 95

};

1. **Runes:** A rune is a Unicode code point and is used to represent characters in a string.

var smiley = '\u{1F600}';

1. **Symbols:** A symbol is a unique identifier that is used to represent an operator or identifier.

Symbol operator = #+;

These data types provide a way to store and manipulate values in Dart programs.

## Functions in dart:

In Dart, a function is a group of statements that perform a specific task. Functions can take input parameters, process them, and return a result. Functions are declared using the **"void"** or **"return type"** keywords, followed by the ***function name, input parameters (if any), and the function body***.

Here is an example of a simple function that takes two numbers as input and returns their sum:

int addNumbers(int a, int b) {

return a + b;

}

You can call this function like this:

int result = addNumbers(10, 20);

print(result); // Output: 30

In Dart, you can also create functions with optional parameters. These parameters can be marked with square brackets "[]" and are given default values if they are not provided during the function call. For example

String sayHello({String name = "World"}) {

return "Hello, $name";

}

## Types of Functions in dart:

In Dart, there are several types of functions:

* **Named functions:** These are functions that have a name and are declared using the ***"function"*** keyword followed by the function name, input parameters, and the function body.

int addNumbers(int a, int b) {

return a + b;

}

///call the above function like -

final addendum = addNumbers(10, 05);

* **Anonymous functions:** These are functions that do not have a name and are declared using the ***"function"*** keyword followed by the input parameters, and the function body. Anonymous functions are often used as input to other functions.

var sum = (int a, int b) => a + b;

* **Arrow functions:** These are a shorthand syntax for anonymous functions and are written using the ***"=>"*** operator

var sum = (int a, int b) => a + b;

* **Static functions:** These are functions that are associated with a class and can be called using the class name. Static functions do not have access to the instance variables and methods of the class.

class Math {

static int add(int a, int b) {

return a + b;

}

}

* **Instance functions:** These are functions that are associated with an object and can be called using the object reference. Instance functions have access to the instance variables and methods of the class.

class Calculator {

int add(int a, int b) {

return a + b;

}

}

var calculator = Calculator();

var result = calculator.add(10, 20);

These are some of the most commonly used types of functions in Dart, and you can choose the type of function that best fits your needs depending on your requirements.

## Operators in dart:

In Dart, operators are symbols that perform specific operations on one or more operands (values or expressions). Dart supports the following types of operators:

1. **Arithmetic Operators:** These are the basic arithmetic operators for performing addition, subtraction, multiplication, and division. For example:

int a = 10;

int b = 20;

print(a + b); // Output: 30

print(b - a); // Output: 10

print(a \* b); // Output: 200

print(b / a); // Output: 2.0

1. **Comparison Operators:** These operators compare two values and return a Boolean value indicating whether the comparison is true or false. For example:

int a = 10;

int b = 20;

print(a == b); // Output: false

print(a != b); // Output: true

print(a < b); // Output: true

print(a > b); // Output: false

1. **Logical Operators:** These operators are used to combine multiple conditions and return a Boolean value indicating whether the combined conditions are true or false. For example:

bool isRaining = true;

bool isCold = false;

print(isRaining && isCold); // Output: false

print(isRaining || isCold); // Output: true

print(!isRaining); // Output: false

1. **Assignment Operators:** These operators are used to assign values to variables. For example:

int a = 10;

a = 20;

print(a); // Output: 20

a += 10;

print(a); // Output: 30

a -= 5;

print(a); // Output: 25

1. **Conditional Operators:** The conditional operator (also known as the ternary operator) is a shorthand way to write an if-else statement. For example:

int a = 10;

int b = 20;

String result = (a > b) ? "a is greater" : "b is greater";

print(result); // Output: b is greater

These are some of the most commonly used operators in Dart, and you can use them to perform various operations on values and expressions.

## Control flow statements in dart:

In Dart, control-flow statements are used to control the flow of execution in a program based on certain conditions. The following are the types of control-flow statements in Dart:

1. **if-else statement:** The if-else statement is used to conditionally execute a block of code based on a Boolean expression. For example:

int a = 10;

int b = 20;

if (a > b) {

print("a is greater than b");

} else {

print("b is greater than or equal to a");

}

**2. for loop**: The for loop is used to repeat a block of code a specified number of times. For example:

for (int i = 0; i < 5; i++) {

print(i);

}

**3. while loop:** The while loop is used to repeatedly execute a block of code as long as a certain condition is true. For example:

int i = 0;

while (i < 5) {

print(i);

i++;

}

**4. do-while loop:** The do-while loop is similar to the while loop, but it guarantees that the loop body will be executed at least once. For example:

int i = 0;

do {

print(i);

i++;

} while (i < 5);

**5. switch-case statement:** The switch-case statement is used to choose between several alternatives based on the value of an expression. For example:

int num = 2;

switch (num) {

case 1:

print("One");

break;

case 2:

print("Two");

break;

default:

print("Invalid");

}

These control-flow statements are used to write programs that can make decisions and execute different actions based on different conditions. They are essential for creating complex and dynamic applications.

## String in dart

A string is a sequence of characters, enclosed in either single quotes (‘…’) or double quotes (“…”). Strings are an essential data type in any programming language, and Dart provides several methods and properties for working with strings. Here are some examples of how strings are used in Dart:

1. **Creating a String:** You can create a string by enclosing a sequence of characters in either single quotes or double quotes. For example:

String name = "Subhash Chandra";

String greeting = 'Hello, $name!';

**2. Concatenating Strings:** You can concatenate two or more strings using the + operator.

**For example:**

**String firstName = "Subhash";**

**String lastName = "Chandra";**

**String fullName = firstName + " " + lastName;**

**3. String Interpolation:** You can insert the value of an expression into a string using string interpolation, which is represented by $ followed by the expression. For example:

int age = 30;

String message = "I am $age years old.";

**4. Length of a String:** You can find the length of a string using the length property. For example:

String name = "Subhash Chandra";

int length = name.length;

**5. SubStrings:** You can extract a portion of a string using the substring method. For example:

String name = "Subhash Chandra";

String firstName = name.substring(0, 4);

**6. Comparing Strings:** You can compare two strings using the equality operator == or the compareTo method. For example:

String name1 = "Subhash";

String name2 = "Any";

print(name1 == name2); // Output: false

print(name1.compareTo(name2)); // Output: -1

**7. Replacing Substrings:** You can replace a portion of a string with another string using the replace method. For example:

String sentence = "The quick brown fox jumps over the lazy dog.";

String newSentence = sentence.replaceAll("dog", "cat");

**8. Converting to Upper- or Lowercase:** You can convert a string to uppercase or lowercase using the toUpperCase and toLowerCase methods. For example:

String name = "Subhash Chandra";

String upperCaseName = name.toUpperCase();

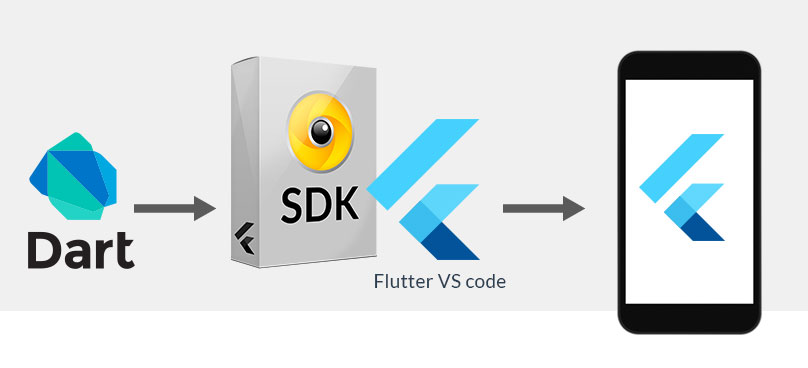
String lowerCaseName = name.toLowerCase();

These are just a few of the many ways you can use strings in Dart. Strings are used extensively in web and mobile applications, so it’s important to have a good understanding of how they work.

# 

# Environment setup

Flutter SDK: Flutter SDK is a software development kit for building high-performance, high-fidelity, and beautiful applications for mobile, web, and desktop from a single codebase. It's an open-source framework developed by Google and is used for creating cross-platform applications for iOS and Android. The Flutter SDK includes a modern reactive framework, a rich set of customizable widgets, and tools for building natively compiled applications for mobile, web, and desktop. With Flutter, developers can create applications with smooth animations, beautiful designs, and native performance on both iOS and Android platforms. The Flutter SDK also provides a fast development cycle with **"hot reload,"** allowing for quicker bug fixing and feature addition. This makes it a popular choice for developers who want to build high-quality apps with a fast turnaround time.



**Flutter Environment setup:**

To set up a development environment for Flutter, you’ll need to install the following software:

1. **Flutter SDK:** The Flutter SDK is the core component of the development environment. You can download it from the official Flutter website.
2. **Dart SDK:** Flutter uses Dart as its programming language, so you’ll also need to install the Dart SDK. This is included in the Flutter SDK, so you don’t need to download it separately.
3. **IDE (Integrated Development Environment):** You can use any popular IDE, such as Visual Studio Code, Android Studio, or IntelliJ IDEA, to write, run, and debug your Flutter applications.
4. **Android Studio (Optional):** If you want to develop for Android, you’ll need to install Android Studio and set up an Android virtual device (AVD) to run your app.
5. **Xcode (Optional):** If you want to develop for iOS, you’ll need to install Xcode, which is only available on MacOS.

Once you’ve installed these components, you’re ready to start developing your first Flutter app. ***To verify that your development environment is set up correctly, you can run the following command in the terminal:***

flutter doctor

This command will check for any missing dependencies and give you a report on the state of your development environment. If everything is set up correctly, you should see a message that says ***“Doctor found no issues.”***

For more knowledge please follow this link to install latest SDK:

1. <https://docs.flutter.dev/get-started/install/windows>
2. <https://docs.flutter.dev/get-started/install/macos>
3. <https://docs.flutter.dev/get-started/install/linux>
4. <https://docs.flutter.dev/get-started/install/chromeos>

**IDE (Integrated Development Environment):** You can use any popular IDE, such as Visual Studio Code, Android Studio, or IntelliJ IDEA, to write, run, and debug your Flutter applications.

# Flutter CLI -

Flutter is a mobile application development framework that enables developers to create high-quality, cross-platform apps for Android and iOS. Here are some of the most commonly used CLI (command line interface) commands for Flutter:

1. **flutter create [project\_name]**: Creates a new Flutter project with the given project name.
2. **flutter run**: Builds and runs the Flutter app on an attached device or emulator.
3. **flutter build**: Builds the Flutter app for release to the App Store or Google Play.
4. **flutter doctor**: Checks your environment and displays a report of the status of your Flutter installation.
5. **flutter pub get**: Fetches the dependencies listed in the pubspec.yaml file.
6. **flutter pub upgrade**: Upgrades the dependencies listed in the pubspec.yaml file to their latest versions.
7. **flutter clean**: Deletes the build/ directory and any other generated files.
8. **flutter packages get**: Fetches the dependencies listed in the pubspec.yaml file.
9. **flutter analyze**: Analyzes your code and displays any issues or errors.
10. **flutter test**: Run the tests in your Flutter project.

These are just a few of the many commands available in the Flutter CLI. For a full list of commands, you can run flutter help in your terminal.

# Widgets

Widgets in Flutter are the basic building blocks of the user interface. They define how the UI looks and behaves. Widgets can be combined to create complex user interfaces and can be easily customized.

*Flutter provides a wide variety of widgets that developers can use to build their applications. Some of the most common types of widgets include:*

## Stateless widgets((Non-Mutable Data):

**A StatelessWidget** is a type of widget in Flutter that is immutable and cannot change dynamically during the lifetime of the widget. StatelessWidgets are typically used to represent static parts of the user interface that don't need to change dynamically, such as text labels, images, and containers. Here's an example of a simple StatelessWidget.

import 'package:flutter/material.dart';

class MyTextLabel extends StatelessWidget {

final String text;

MyTextLabel({required this.text});

@override

Widget build(BuildContext context) {

return Text(

text,

style: TextStyle(fontSize: 24),

);

}

}

In this example, the MyTextLabel widget is a **StatelessWidget** that takes a text parameter and displays it as a Text widget with a **font size of 24**. ***The StatelessWidget is created once and its build method is called only once when it is first rendered. The benefits of using StatelessWidgets include improved performance and reduced memory usage, since they are created only once and their state doesn't change dynamically.***

## Stateful widgets(Mutable Data):

A StatefulWidget is a type of widget in Flutter that can change dynamically during the lifetime of the widget. StatefulWidgets are typically used to represent parts of the user interface that need to change dynamically, such as buttons, text fields, and sliders. In Flutter, a StatefulWidget is combined with a separate State object, which holds the mutable state associated with the widget. When the state of the widget changes, the StatefulWidget is rebuilt using the updated state.

*Here's an example of a simple StatefulWidget in Flutter:*

import 'package:flutter/material.dart';

class MyCounter extends StatefulWidget {

const MyCounter({super.key});

@override

\_MyCounterState createState() => \_MyCounterState();

}

class \_MyCounterState extends State<MyCounter> {

int \_counter = 0;

void \_incrementCounter() {

setState(() {

\_counter++;

});

}

@override

Widget build(BuildContext context) {

return Column(

children: [

Text('Count: $\_counter'),

TextButton(

onPressed: \_incrementCounter,

child: const Text('Increment'),

),

],

);

}

}

**Layout widgets**: These are widgets that control the arrangement and sizing of other widgets. Examples of layout widgets include Container, Row, Column, and Expanded. Text widgets: These are widgets that display text. Examples of text widgets include Text, RichText, and TextField.

**Animated widgets:** These are widgets that provide animation capabilities. Examples of animated widgets include AnimatedContainer, AnimatedCrossFade, and AnimatedOpacity.

**Interactive widgets:** These are widgets that respond to user input. Examples of interactive widgets include Button, Checkbox, and Slider.

**Material widgets:** These are widgets that follow the Material Design guidelines, which are the guidelines for designing Android applications. Examples of Material widgets include Scaffold, AppBar, RaisedButton, and TextField. For more detail — <https://medium.com/@subhashchandrashukla/flutter-material-widgets-883f9946982b>

**Cupertino widgets:** These are widgets that follow the iOS Human Interface Guidelines, which are the guidelines for designing iOS applications. Examples of Cupertino widgets include CupertinoNavigationBar, CupertinoButton, and CupertinoSlider. Examples:

* **CupertinoNavigationBar:** A navigation bar at the top of the screen that includes a back button and a title. For more details please open the link -<https://medium.com/@subhashchandrashukla/flutter-cupertinonavigationbar-27095457c6d2>
* **CupertinoTabBar:** A tab bar that can be used for navigation within an app. For more example go through the link — <https://medium.com/@subhashchandrashukla/flutter-cupertinotabbar-d7fbd0aaf350>
* **CupertinoPageScaffold:** A scaffold that provides a layout structure for an iOS app. For more detail please go through the link — <https://medium.com/@subhashchandrashukla/flutter-cupertinopagescaffold-733c32ae27b0>
* **CupertinoAlertDialog:** A dialog that displays an alert message.
* **CupertinoButton:** A button with a filled background and a text label. These are just a few examples of the types of widgets that are available in Flutter. By using these widgets, developers can build complex and dynamic user interfaces that are performance and visually appealing.

# 

# Navigation in flutter:

Navigation in Flutter is the process of moving between different screens or pages within an app. Flutter provides a variety of built-in navigation widgets and methods to help developers implement navigation in their apps.

Here are some common ways to implement navigation in Flutter:

## **Navigator widget:**

Flutter's Navigator widget provides a stack-based navigation model for managing routes in an app. A route is a screen or page in an app, and each route is pushed onto a stack when it is displayed. The topmost route on the stack is the one that is currently visible.

To use the Navigator widget, you can create a MaterialApp or CupertinoApp widget as the root widget of your app and define a MaterialApp.navigatorKey or CupertinoApp.navigatorKey, respectively. Then, use the Navigator widget to manage the stack of routes by pushing and popping them onto the stack. You can also use the Navigator to pass data between screens using the Navigator.pushNamed() method.

For example -

import 'package:flutter/material.dart';

main() {

runApp(const MaterialApp(

home: FirstPage(),

));

}

class FirstPage extends StatelessWidget {

const FirstPage({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) {

return Scaffold(

body: Center(

child: ElevatedButton(

child: const Text('Navigate to second'),

onPressed: () {

Navigator.push(

context,

MaterialPageRoute(

builder: (\_) => const SecondPage(),

),

);

},

),

),

);

}

}

class SecondPage extends StatelessWidget {

const SecondPage({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('Back to previous'),

),

body: Center(

child: TextButton(

onPressed: () {

Navigator.of(context).pop();

},

child: const Text('Back to previous')),

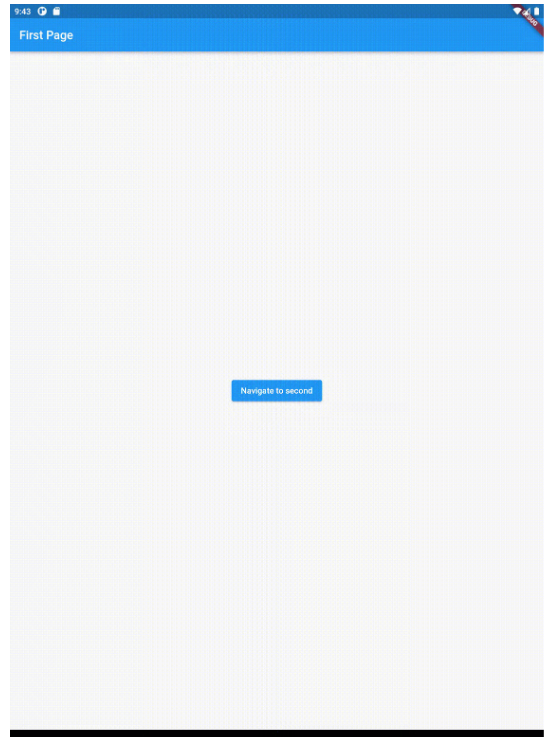
),

);

}

}

**Output:**

****

The first part of this code imports the Material library from Flutter which includes pre-built classes and components, such as Buttons, ApBars and Widgets to help you quickly design visual interfaces for your app. The **main()** function is the entry point for any Flutter application. It instantiates a **MaterialApp** class that sets the main page for the **app (FirstPage())**. The FirstPage is a StatelessWidgetdefined in the code. It renders a Scaffold, which is a basic layout structure in Flutter that usually consists of an AppBar, Drawer (optional) and a body. The build() method creates a Center Widget with an **ElevatedButton** inside which will navigate to another defined widget **(SecondPage())** when pressed.

The SecondPage class is also a StatelessWidget that renders a Scaffold and displays a **TextButton**. When pressed this button will return the user to the previous page using the Navigator.

## **Named routes:**

Named routes are a way to navigate to a specific screen using a unique name. You can define named routes in the MaterialApp or CupertinoApp widget using the routes property. Each named route is associated with a particular widget, and you can navigate to the route using the Navigator.pushNamed() method.

For example -

import 'package:flutter/material.dart';

const kSecondPageRoute = '/second';

void main() {

runApp(MaterialApp(

home: const FirstPage(),

routes: {

kSecondPageRoute: (context) => const SecondPage(),

}

));

}

class FirstPage extends StatelessWidget {

const FirstPage({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('First Page'),

),

body: Center(

child: ElevatedButton(

child: const Text('Navigate to second'),

onPressed: () {

Navigator.pushNamed(context, kSecondPageRoute);

},

),

),

);

}

}

class SecondPage extends StatelessWidget {

const SecondPage({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('Back to previous'),

),

body: Center(

child: TextButton(

onPressed: () {

Navigator.pop(context);

},

child: const Text('Back to previous')),

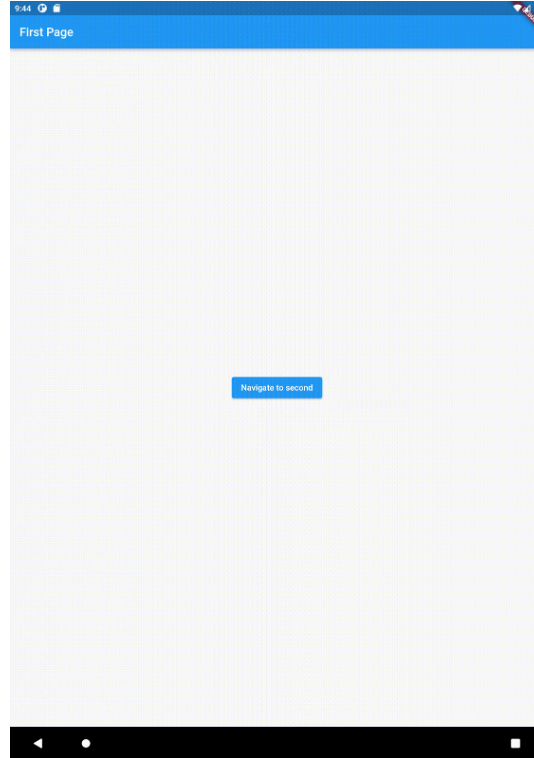
),

);

}

}

**Output:**

****

In this example , First, the **MaterialApp** constructor now has a **named route parameter that points to the SecondPage**. This allows you to more easily refer to it from other parts of the code. Instead of using **Navigator.push** to **navigate** between the **pages**, you can use **Navigator.pushNamed**, which takes a string argument for the route name. This helps keep your navigation flows explicit and makes them easier to maintain. Finally, instead of using **Navigator.of(context).pop()** to go back to the previous page, you can use **Navigator.pop(context)**, which takes the current context as an argument and simplifies things by avoiding the need to get an instance of the Navigator.

## **BottomNavigationBar widget**:

The BottomNavigationBar widget provides a way to display a menu of icons or text labels at the bottom of the screen, allowing users to switch between different pages or screens in the app. You can define the items in the BottomNavigationBar using the BottomNavigationBarItem widget and associate each item with a route.

For example -

import 'package:flutter/material.dart';

// This is the main function

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {

const MyApp({super.key});

@override

Widget build(BuildContext context) {

return const MaterialApp(

title:"BottomNavigationBar",

home: HomePage(),

);

}

}

class HomePage extends StatefulWidget {

const HomePage({super.key});

@override

// ignore: library\_private\_types\_in\_public\_api

\_HomePageState createState() => \_HomePageState();

}

class \_HomePageState extends State<HomePage> {

int \_currentIndex = 0;

final \_pages = const[

HomeFregView(),

ProfileFregView()

];

@override

Widget build(BuildContext context) {

return Scaffold(

backgroundColor: Colors.teal,

body: \_pages[\_currentIndex],

bottomNavigationBar: BottomNavigationBar(

items: const <BottomNavigationBarItem>[

BottomNavigationBarItem(

icon : Icon(Icons.home),

label: 'Home',

),

BottomNavigationBarItem(

icon: Icon(Icons.account\_circle),

label: 'Profile'

)

],

currentIndex: \_currentIndex,

onTap: (int index) {

setState((){

\_currentIndex = index;

});

}

),

);

}

}

class HomeFregView extends StatelessWidget {

const HomeFregView({super.key});

@override

Widget build(BuildContext context) {

return Container(

color: Colors.green.shade100,

child: const Center(

child: Text('Welcome to the Home Page'),

),

);

}

}

class ProfileFregView extends StatelessWidget {

const ProfileFregView({super.key});

@override

Widget build(BuildContext context) {

return Container(

color: Colors.red.shade100,

child: const Center(

child: Text('Welcome to the Profile Page'),

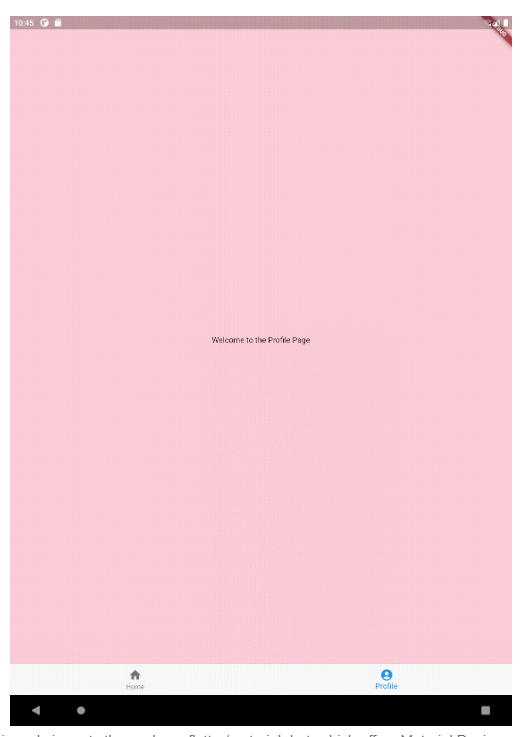
),

);

}

}

**Output:**



In this code imports the package flutter/material.dart, which offers Material Design components to use in creating user interfaces. The main function defines the starting point of a Flutter app by using **runApp()**, which takes an instance of type MyApp as its argument. The MyApp class is a stateless widget, which means it does not contain any state. Everytime this widget is built it contains the same properties and returns the same result. In this case the **MaterialApp()** widget. The **HomePage** class is defined as a statefulWidget, which means it does contain state information (in this case it is an integer). Everytime this widget is built, it contains the state from the last build, so that the properties and results can differ. There is a **\_HomePageState** class, which holds some information. In this case the **currentIndex** will filter out which page to show. The body of the Scaffold widget will be either the **HomeFregView** or **ProfileFregView** based on the **currentIndex** value. The bottom navigation bar contains two items with icons and labels. Setting one of these becomes the currentIndex (the **\_HomePageState** property) which filters out which page is needed to display.

## TabBar widget:

The TabBar widget is similar to the BottomNavigationBar widget but displays tabs at the top of the screen. Each tab can be associated with a route and provides a way for users to navigate between different screens or pages in the app.

For example -

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(

home: DefaultTabController(

length: 2,

child: Scaffold(

appBar: AppBar(

bottom: const TabBar(

tabs: <Widget>[

Tab(icon: Icon(Icons.home)),

Tab(icon: Icon(Icons.account\_circle))

],

),

title: const Text('Tab Bar Sample'),

),

body: const TabBarView(children: [

HomeFregView(),

ProfileFregView()

]),

),

),

);

}

}

class HomeFregView extends StatelessWidget {

const HomeFregView({super.key});

@override

Widget build(BuildContext context) {

return Container(

color: Colors.green.shade100,

child: const Center(

child: Text('Welcome to the Home Page'),

),

);

}

}

class ProfileFregView extends StatelessWidget {

const ProfileFregView({super.key});

@override

Widget build(BuildContext context) {

return Container(

color: Colors.red.shade100,

child: const Center(

child: Text('Welcome to the Profile Page'),

),

);

}

}

**Output:**

# 

# 

Overall, Flutter provides many options for implementing navigation in an app, including the Navigator widget, named routes, and various navigation widgets. By leveraging these tools, developers can create intuitive and user-friendly navigation experiences that enhance the overall usability and functionality of their Flutter apps.

# 

# Working with assets:

To use assets in Flutter, you need to declare the assets in the pubspec.yaml file and then reference them in your code. Here is an example of how you could declare assets in the pubspec.yaml file:

flutter:

assets:

- assets/images/

In this example, all image files located in the **assets/images/** directory will be included as assets in your application. To reference an asset in your code, you can use the **AssetImage widget**. For example:

AssetImage("assets/images/your\_image.png")

**Note:** *that the path specified in the AssetImage constructor should match the path declared in the pubspec.yaml file. It's also worth mentioning that you can use the Image widget to display an image. You can pass the AssetImage to the Image widget's image property to display the image.* For example:

Image(

image: AssetImage("assets/images/your\_image.png"),

)

# 

# Flutter design-principles:

Here are some common design principles that are often followed in Flutter development:

**Material Design:** A design system developed by Google that provides a consistent look and feel across all platforms and devices.

**Responsiveness:** Designing the user interface to work seamlessly on different screen sizes and aspect ratios.

**User-centered Design:** Designing the application with the user's needs and goals in mind.

**Simplicity:** Designing the user interface to be simple and intuitive, minimizing the number of steps needed to complete a task.

**Consistency:** Maintaining a consistent look and feel throughout the application, using consistent colors, typography, and layout.

**Accessibility:** Making the application usable by people with disabilities, including screen reader support, keyboard accessibility, and high-contrast mode.

Performance: Designing the application to be fast and responsive, even on low-end devices. By following these design principles, you can create a user-friendly and visually appealing application that provides a great user experience.

## Dependency - Injection:

Dependency injection is a software design pattern that allows for the removal of hard-coded dependencies and makes it possible to change them, whether at run-time or compile-time. In the context of Flutter, dependency injection is useful for separating the concerns of different parts of an application and promoting modular, testable code.

There are several popular libraries available for implementing dependency injection in Flutter, including:

1. **get\_it:** A simple and efficient service locator for Dart and Flutter applications.

*Here's a simple example of how you can use get\_it to implement dependency*

*injection in a Flutter application:*

* First, you'll need to add the get\_it package to your pubspec.yaml file:

dependencies:

get\_it: ^7.2.0

* Next, you'll create a service class that you want to make available throughout your application. For example, let's say you have a class that provides a user authentication service:

class AuthService {

Future<bool> login(String username, String password) async {

// Perform authentication logic here

return true;

}

}

* Then, you'll register the service on root(main.dart) of your project file with get\_it by calling getIt.registerSingleton:

void setup() {

GetIt getIt = GetIt.instance;

getIt.registerSingleton<AuthService>(AuthService());

}

* Finally, you can access the service from anywhere in your application by calling getIt.get:

In this example, the AuthService is made available throughout the entire application, and it can be easily accessed from any widget. Additionally, if you need to change the implementation of the AuthService, you can do so without having to change the code that depends on it, which makes the code more maintainable and testable.

*For more details please go through link -* <https://github.com/subhashDev11/flutter-boilerplate>

In this example, the AuthService is made available throughout the entire application, and it can be easily accessed from any widget. Additionally, if you need to change the implementation of the AuthService, you can do so without having to change the code that depends on it, which makes the code more maintainable and testable.

1. **Provider:** A Flutter package that provides an easy way to manage the state of your applications.
2. **Riverpod:** A modern and flexible state management library for Flutter.
3. **Flutter Inject:** A library that provides support for dependency injection in Flutter applications.
4. **Dagger:** A fast dependency injector for Dart and Flutter.

These libraries provide different levels of abstraction and can be used depending on the complexity of the application and the personal preference of the developer. Whichever library you choose, the basic idea behind dependency injection is the same: to decouple the implementation of a component from its dependencies and make it easier to test and maintain the code.

## Flutter design-patterns-

What is architecture -

**In** software development, architecture refers to the overall structure and organization of a codebase. In software development, architecture refers to the overall structure and organization of a codebase. In the context of a Flutter app, the architecture refers to the way in which the different parts of the app are organized and the patterns and principles that are used to structure the code.

In Flutter, there are several popular architectural patterns that can be used to structure a Flutter app, each with its own strengths and weaknesses. Some of the most commonly used architectural patterns for Flutter are:

### The Model-View-Controller (MVC) pattern :

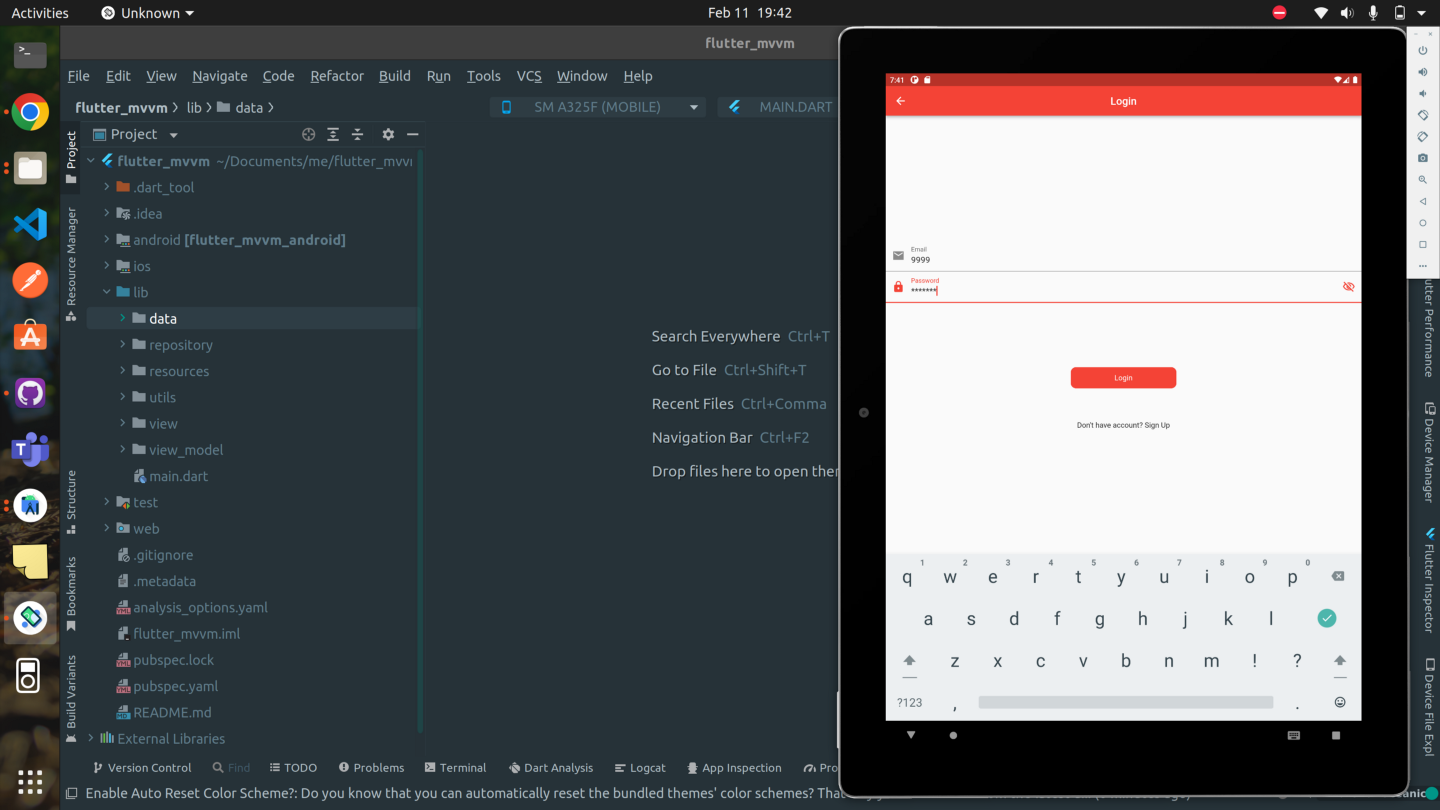
MVC is not built into Flutter, but it can be implemented using various architectural patterns. Here’s an example of how you can implement the MVC pattern in Flutter:

* **Model:** This represents the data and business logic of your app. You can create classes that define your data and perform operations on that data.
* **View:** This represents the user interface of your app. You can create widgets that display your data and respond to user input.
* **Controller:** This acts as the intermediary between the Model and the View. It receives input from the View, manipulates the data in the Model, and updates the View accordingly.

*You can implement the MVC pattern in Flutter using various architectural patterns, such as BLoC (Business Logic Component), Provider, or MobX. These patterns allow you to separate the presentation layer (view) from the business logic (model) and provide a clear separation of concerns, making your code more modular, testable, and maintainable.*

For example purpose go through the link - <https://github.com/subhashDev11/flutter_mvc>

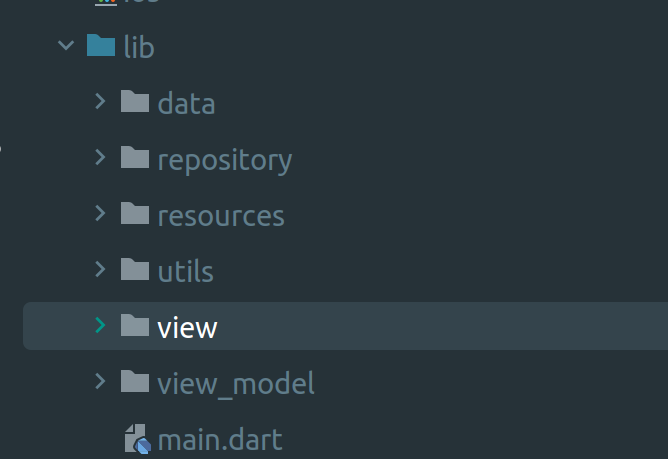
### The Model-View-ViewModel (MVVM) pattern:

****

Flutter MVVM (Model-View-ViewModel): MVVM is an architectural pattern that is used to separate the user interface (View) from the business logic and data (Model) of an application, while also providing a clear separation of concerns. Here’s how you can implement the MVVM pattern in Flutter:

* **Model:** This represents the data and business logic of your app. You can create classes that define your data and perform operations on that data.
* **View:** This represents the user interface of your app. You can create widgets that display your data and respond to user input.
* **ViewModel:** This acts as an intermediary between the Model and the View. It exposes data and operations from the Model to the View, and it also receives input from the View and updates the Model accordingly.

The ViewModel is responsible for maintaining the state of the View and communicating changes to the View. It also provides a way for the View to interact with the Model without directly accessing it.



*You can implement the MVVM pattern in Flutter using various architectural patterns, such as Provider, or BLoC (Business Logic Component) with StreamBuilder. These patterns allow you to separate the presentation layer (View) from the business logic and data (Model) and provide a clear separation of concerns, making your code more modular, testable, and maintainable.*

For example go through the link — <https://github.com/subhashDev11/flutter_mvvm>

* The BLoC (Business Logic Component) pattern:

This pattern is similar to MVVM, but it places more emphasis on separating the business logic from the view and view model layers. It introduces a BLoC (Business Logic Component) layer, which is responsible for implementing the business logic of the app.

For more knowledge in deep you can go through the link - <https://github.com/bw7432/flutter_bloc_starter>

### The Redux pattern:

This is a popular architectural pattern that is particularly well-suited to building large-scale, complex apps with complex state management. It introduces a centralized store that contains the state of the app, and a set of actions and reducers that are responsible for modifying the state.

### The Scoped Model pattern:

This is another architectural pattern that is particularly well-suited to managing state in a Flutter app. It introduces a “Scoped Model” class that holds the state of the app and provides a set of methods for modifying the state.

*All of these architectural patterns can be used to build a Flutter app, and the best choice for your app will depend on the specific requirements of the app, such as the complexity of the app and the amount of state that needs to be managed.*

*Additionally, there are also some libraries that help developers implement different architectures in a flutter such as f****lutter\_clean\_architecture, flutter\_bloc, provider and flutter\_redux*** *they could help you to structure your project and follow the best practices.*

## SOLID PRINCIPLES

The term SOLID is an acronym for five famous design principles, Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion, which we will examine below.

DART follows SOLID principles, which are a set of five design principles that help developers write more maintainable and scalable code.

Here's a brief overview of each SOLID principle and how it can be applied in Dart:

* **Single Responsibility Principle (SRP):** A class should have only one reason to change. In Dart, this means each class should have a single responsibility or purpose.
* **Open/Closed Principle (OCP):** A class should be open for extension but closed for modification. In Dart, this means that you should be able to add new functionality to a class without modifying its existing code.
* **Liskov Substitution Principle (LSP):** Subtypes must be substitutable for their base types. In Dart, this means that you should be able to use a subtype in place of its parent type without affecting the correctness of your code.
* **Interface Segregation Principle (ISP):** Clients should not be forced to depend on interfaces they do not use. In Dart, this means that you should create small and focused interfaces that only contain the methods that are relevant to their clients.
* **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions. In Dart, this means that you should depend on abstractions (interfaces or abstract classes) rather than concrete implementations to decouple your code and make it easier to change and maintain.

By following these principles, you can write code that is more modular, extensible, and maintainable in Dart.

## OOP’S in Dart:

Dart is an object-oriented programming language, which means that it is designed to work with objects, classes, and other OOP concepts. Here's a detailed explanation of OOP in Dart:

### Classes and Objects:

In Dart, a class is a blueprint or a template that defines the properties and methods of an object. You can create a class using the class keyword, followed by the name of the class and a pair of curly braces that enclose the class body. Here's an example of a simple class in Dart:

class Person {

String name;

int age;

Person(this.name, this.age);

void sayHello() {

print('Hello, my name is $name and I am $age years old.');

}

}

In this example, we have created a class named Person that has two properties, name and age, and a method called sayHello. This keyword is used to refer to the current instance of the class. The constructor, Person, takes two parameters, name and age, and assigns them to the properties of the class.

To create an instance of the Person class, you can use the new keyword followed by the name of the class and the constructor parameters, like this:

void main() {

final person = Person(

'SUBHASH CHANDRA SHUKLA',

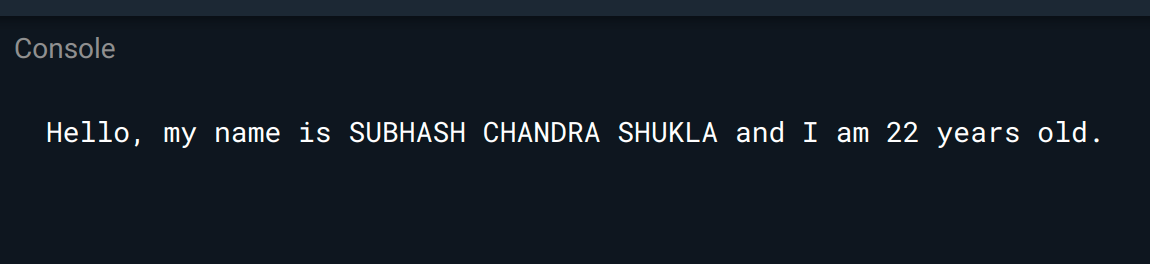
22,

);

person.sayHello();

}

**Output:**

****

Overall, classes in Dart provide a way to encapsulate data and behavior into reusable components, allowing you to write more modular and maintainable code.

## Constructor in dart:

In Dart, a constructor is a special method that is used to create and initialize objects of a class. Constructors have the same name as the class and can be used to set initial values for object properties or perform other setup tasks.

Dart supports several types of constructors, including **default constructors, named constructors, and factory constructors**.

**A default constructor is created automatically by Dart if no constructor is defined explicitly. A named constructor is a constructor with a name other than the class name, while a factory constructor is a constructor that returns an instance of a class.**

**For example 👍**

**class Person {**

**String? name;**

**int? age;**

**// Default Constructor**

**Person(this.name, this.age);**

**// Named Constructor**

**Person.withNameOnly(this.name) {**

**age = 0;**

**}**

**// Factory Constructor**

**factory Person.fromBirthYear(String name, int birthYear) {**

**final now = DateTime.now();**

**final age = now.year - birthYear;**

**return Person(name, age);**

**}**

**}**

**void main() {**

**// Using the default constructor**

**Person me = Person("SUBHASH CHANDRA SHUKLA", 22);**

**// Using the named constructor**

**Person bro = Person.withNameOnly("BRIJ BHOOSHAN SHUKLA");**

**// Using the factory constructor**

**Person secBro = Person.fromBirthYear("SHIVAM", 2000);**

**print(me.name); // Output: SUBHASH CHANDRA SHUKLA**

**print(bro.age); // Output: 0**

**print(secBro.age); // Output: 23**

**}**

In this example, the Person class has a **default constructor** that takes two arguments **(name and age)**, a named constructor called **withNameOnly** that takes only the name argument and sets the age to 0, and a factory constructor called **fromBirthYear** that takes the name and birthYear arguments, calculates the age based on the current year, and returns a new Person object with the calculated age.

In the main function, we create three Person objects using each of the constructors and print out their properties.

**Output:**

****

## this and super keyword in dart:-

In Dart, **this** and **super** are both keywords that are used in **object-oriented programming** to refer to the current instance and the parent class, respectively.

this refers to the current instance of the class in which it is used. It is often used to disambiguate between instance variables and local variables that have the same name. For example, **this.variableName** r**efers to the instance variable variableName.**

**super refers to the parent class of the current class. It is often used to call a method or constructor in the parent class. For example, super.methodName() calls the method methodName in the parent class.**

Here is an example that shows how to use this and super in Dart:

**class Person {**

**String name;**

**Person(this.name);**

**void printName() {**

**print("My name is $name.");**

**}**

**}**

**class Student extends Person {**

**int grade;**

**Student(String name, this.grade) : super(name);**

**void printInfo() {**

**super.printName(); // Call parent class method**

**print("I am in grade $grade.");**

**}**

**}**

**void main() {**

**Student subhash = Student("SUBHASH CHANDRA SHUKLA", 12);**

**subhash.printInfo();**

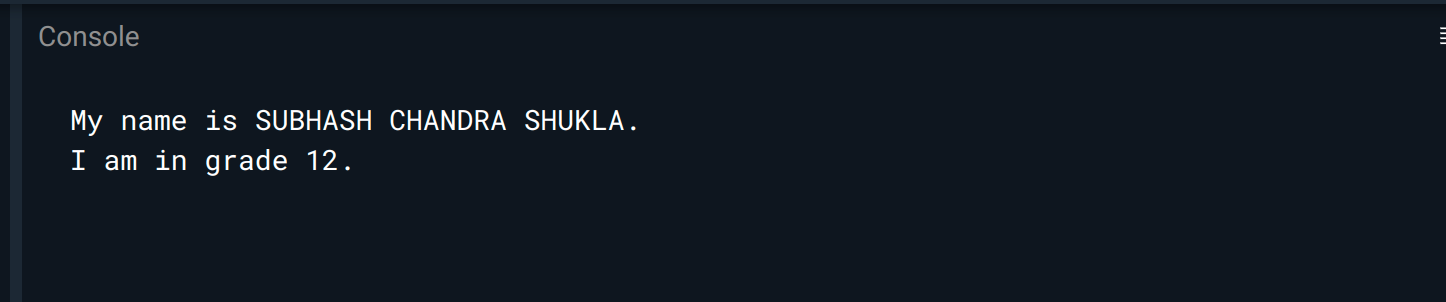
**}**

In this example, we define a Person class with an instance variable name and a method printName that prints out the name. We also define a Student class that extends Person and adds an instance variable grade.

In the Student constructor, we call the parent class constructor using **super(name)** to set the name variable. In the printInfo method, we call the printName method in the parent class using **super.printName()** and print out the grade variable.

In the main function, we create a new Student object and call the printInfo method, which prints out the name and grade of the student. The output shows how **super** is used to call the parent class method and this is used to access the **current** **instance variable**.

**Output:**



## Encapsulation:-

Encapsulation is the concept of bundling data and methods together within a single unit, such as a class, and restricting access to the internal details of the object. In Dart, you can achieve encapsulation by declaring your class properties as private, using the underscore symbol (\_). Private properties can only be accessed within the class itself.

Here's an example of encapsulation in Dart:

class Person {

String \_name;

int \_age;

Person(this.\_name, this.\_age);

void sayHello() {

print('Hello, my name is $\_name and I am $\_age years old.');

}

}

In this example, we have defined the \_name and \_age properties of the Person class as private by using the **underscore symbol (\_)** before their names. **This means that they can only be accessed from within the Person class.**

**The constructor takes two parameters, \_name and \_age, and assigns them to the private properties of the class. The sayHello method is public and can be accessed from outside the class, but it uses the private properties to display a message.**

To create an instance of the Person class and access its properties, you can use the following code:

void main() {

final person = Person(

'SUBHASH CHANDRA SHUKLA',

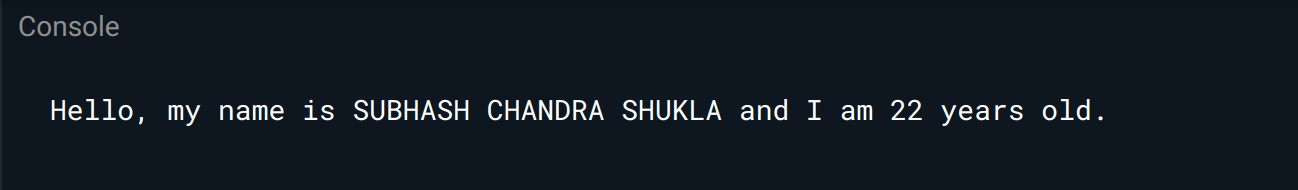
22,

);

person.sayHello();

}

**Output:-**

****

## Inheritance:

Inheritance is a key feature of object-oriented programming that allows classes to inherit properties and methods from another class. In Dart, you can use the "extends" keyword to create a subclass that inherits from a superclass. Here's an example:

// Define a superclass called 'Animal'

class Animal {

void makeSound() {

print('The animal makes a sound');

}

}

// Define a subclass called 'Dog' that extends 'Animal'

class Dog extends Animal {

void makeSound() {

print('The dog barks');

}

}

void main() {

// Create an instance of 'Dog' and call the 'makeSound' method

Dog dog = Dog();

dog.makeSound(); // Output: The dog barks

}

In this example, the superclass Animal defines a method called makeSound(). The subclass Dog extends Animal and overrides the **makeSound()** method with its own implementation. When we create an instance of Dog and call the **makeSound()** method, the output is **"The dog barks"** instead of **"The animal makes a sound"** that would have been printed if the Dog class did not override the **makeSound()** method.

**Output:-**

****

## Polymorphism:

Polymorphism is the ability of objects to take on different forms or behaviors. In Dart, polymorphism is achieved through inheritance and method overriding. Polymorphism allows you to write more flexible and reusable code, and to treat different objects in a consistent manner. Here's an example of polymorphism in Dart:

// Define a superclass called 'Shape'

class Shape {

void draw() {

print('Drawing a shape');

}

}

// Define a subclass called 'Circle' that extends 'Shape'

class Circle extends Shape {

void draw() {

print('Drawing a circle');

}

}

// Define a subclass called 'Rectangle' that extends 'Shape'

class Rectangle extends Shape {

void draw() {

print('Drawing a rectangle');

}

}

void main() {

// Create an array of 'Shape' objects

List<Shape> shapes = [Circle(), Rectangle()];

// Call the 'draw' method on each object in the array

for (Shape shape in shapes) {

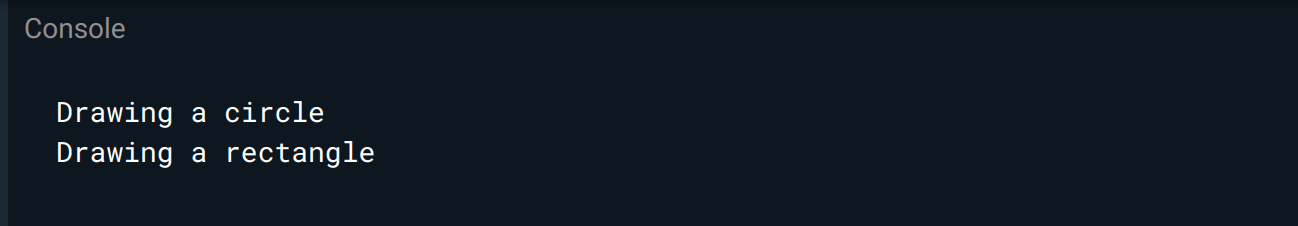
shape.draw();

}

}

In this example, we have a superclass Shape with a method draw(), and two subclasses Circle and Rectangle that override the draw() method with their own implementation. In the main() function, we create an array of Shape objects that contains a Circle and a Rectangle. We then loop through the array and call the draw() method on each object. Because each object is an instance of a different class, the draw() method is called with a different implementation for each object. This is an example of polymorphism in action. The output of the program is:

**Output:**



## Abstraction:

Abstraction in Dart is the process of hiding implementation details and exposing only the necessary functionalities to the user. This is achieved through the use of abstract classes and interfaces, which define a set of methods or properties that must be implemented by any class that inherits from them. By using abstraction, developers can create more modular and flexible code, making it easier to maintain and reuse.

abstract class Animal {

void makeSound(); // abstract method

}

class Cat extends Animal {

@override

void makeSound() {

print('Meow'); // implementation of abstract method

}

}

class Dog extends Animal {

@override

void makeSound() {

print('Woof');

}

}

void main() {

Animal cat = Cat(); // using abstract class

Animal dog = Dog();

cat.makeSound(); // Meow

dog.makeSound(); // Woof

}

In this example, the Animal class is an abstract class that defines an abstract method called **makeSound()**. Any class that inherits from Animal must implement this method. The Cat and Dog classes inherit from Animal and implement the **makeSound()** method with their respective sounds.

The **main()** function creates instances of Cat and Dog using the Animal class, which demonstrates how we can use abstraction to create more flexible and modular code. By using an abstract class, we can write code that works with any subclass of Animal, without needing to know the details of each individual subclass.

**Output:**

****

Overall, OOP concepts in Dart can help you write more modular, reusable, and maintainable code, allowing you to build complex and scalable apps more efficiently.

# Package Manager in flutter:

Flutter uses a package manager called "pub" which is included in the Flutter SDK. Developers can use the **"pub"** command-line tool to manage packages, including installing, upgrading, and removing packages. The **"pubspec.yaml"** file in the root of a Flutter project lists the dependencies and other configuration details for the project, and the **"pub"** tool uses this file to manage packages.

**Dart pub** and **flutter pub** both are command-line tools for managing packages in Dart and Flutter projects, respectively. However, there are some differences between the two:

* dart pub is the general-purpose package manager for Dart, which can be used for any Dart project, not just Flutter. It uses the same package repository as flutter pub (pub.dev).
* flutter pub is a wrapper around dart pub that is specific to Flutter projects. It adds some additional functionality, such as automatically running code generators and providing Flutter-specific commands (such as flutter pub get and flutter pub upgrade).

In general, if you're working on a Flutter project, it's recommended to use flutter pub for managing packages, as it provides some additional convenience features specifically for Flutter development. However, if you're working on a non-Flutter Dart project, you should use a dart pub.

# Storage in flutter:

Flutter provides several options for storing data:

1. Shared Preferences: used for storing small amounts of key-value pairs.
2. SQLite: used for storing relational databases.
3. File Storage: used for storing and reading files in device storage.
4. Firebase: used for storing data in the cloud.

The choice of storage depends on the size and complexity of data, security concerns, and the need for offline access.

## Shared Preferences (Key-Value) Storage:

**Shared Preferences** in Flutter is a key-value store that is used for storing small amounts of data, such as user preferences or settings. It allows developers to store and retrieve primitive data types, such as integers, booleans, strings, and doubles.

**To use Shared Preferences in Flutter, you need to add the shared\_preferences package to your project. After that, you can instantiate a SharedPreferences object and use it to store and retrieve data.** Here is an example:

import 'package:shared\_preferences/shared\_preferences.dart';

// Store data

Future<void> store() async{

SharedPreferences prefs = await SharedPreferences.getInstance();

await prefs.setInt('counter', 42);

}

// Retrieve data

Future<int?> retrieve() async{

SharedPreferences prefs = await SharedPreferences.getInstance();

int counter = prefs.getInt('counter') ?? 0;

return counter;

}

In this example, we store an integer value with the key **'counter'**, and then **retrieve** it later. The **??** operator is used to provide a default value of 0 if the key is not found in the SharedPreferences.

## SQFLite Storage:

Flutter provides support for local database management through the SQLite database engine, which is a popular and widely used relational database management system.

Flutter's implementation of SQLite is provided through the sqflite package. This package allows you to create and manage local databases, as well as perform common database operations such as creating tables, inserting, updating, and deleting records.

*Here is an example of how to use sqflite in Flutter:*

import 'package:sqflite/sqflite.dart';

class SQLiteProvider{

Database? db;

// Create or open a database

Future<void> init() async{

db = await openDatabase('my\_database.db', version: 1,

onCreate: (Database db, int version) async {

// Create the table

await db.execute('''

CREATE TABLE my\_table (

id INTEGER PRIMARY KEY,

name TEXT,

value INTEGER

)

''');

});

}

Future<void> insertData() async{

// Insert a record

await db?.insert('my\_table', {'name': 'my\_name', 'value': 42});

}

Future<List<Map<String, dynamic>>> getData() async{

// Query for records

List<Map<String, Object?>>? rows = await db?.query('my\_table',

where: 'name = ?', whereArgs: ['my\_name']);

return rows ?? [];

}

Future<void> updateRecord() async{

// Update a record

await db?.update('my\_table', {'value': 43},

where: 'name = ?', whereArgs: ['my\_name']);

}

Future<void> deleteRecord() async{

// Delete a record

await db?.delete('my\_table', where: 'name = ?', whereArgs: ['my\_name']);

}

Future<void> close() async{

// Close the database

await db?.close();

}

}

In this example, **we create a database, create a table, insert a record, query for records, update a record, delete a record, and finally close the database.** Note that database operations should be performed asynchronously and can be awaited using the **await** keyword.

## Firebase Storage:

Flutter Firebase refers to the combination of Google's Flutter framework for building mobile applications and Google Firebase, a backend service that provides a variety of tools and services for building and managing mobile and web applications. Using Flutter Firebase, developers can quickly create and deploy high-quality mobile applications with features such as **user authentication, real-time database management, cloud storage**, and more.

For more detail please go through the link - <https://firebase.flutter.dev/docs/overview>

# Working with API’s -

Flutter can work with **APIs (Application Programming Interfaces)** by making **HTTP** requests to the API endpoints and handling the responses. **The http package is commonly used in Flutter to make HTTP requests to APIs**.

To make a request, the developer creates an instance of the http.Client class and uses it to send a http.Request object to the API endpoint. The response is then handled in the form of a http.Response object, which can be parsed to extract the required data.

Flutter also provides tools to handle different types of APIs, such as REST (Representational State Transfer) APIs, which are commonly used in web and mobile applications. There are also packages available in Flutter, such as **dio and retrofit**, which provide more advanced features and simplify the process of working with APIs.

## Flutter JSON Parsing:

In Flutter, you can use the **dart:convert** library to parse JSON data. Here is an example of how to parse a JSON string into a Dart object:

import 'dart:convert';

String jsonString = '{"name": "Subhash Chandra", "age": 30}';

Map<String, dynamic> jsonMap = json.decode(jsonString);

String name = jsonMap['name'];

int age = jsonMap['age'];

In this example, we have a JSON string containing a person's name and age. We use the **json.decode()** function to parse the string into a Dart Map object. We can then access the values of the Map by their keys, and convert them to the appropriate data types.

Here's another example, where we have a JSON array of objects:

import 'dart:convert';

String jsonString = '[{"name": "Subhash Chandra", "age": 30}, {"name": "Subhash Chandra", "age": 25}]';

List<dynamic> jsonArray = json.decode(jsonString);

jsonArray.forEach((person) {

String name = person['name'];

int age = person['age'];

print('$name is $age years old');

});

In this example, we use the **json.decode()** function to parse a JSON array of objects. We store the result in a List variable, and then use a **forEach() loop** to iterate over each object in the **array**. We can access the values of each object by their keys, and perform any necessary processing.

These are just a couple of examples of how to parse JSON data in Flutter. There are other functions and classes in the dart:convert library that you can use for more advanced scenarios, such as encoding Dart objects to JSON, or working with JSON streams.

## Flutter **http** package:

Flutter provides the http package for making HTTP requests and handling responses in a Flutter app. The package provides classes for creating **HTTP requests**, **handling responses**, and working with **headers, cookies, and other HTTP features**.

* To use the http package in a Flutter app, the developer needs to include it as a dependency in their pubspec.yaml file, and then import it in their Dart code.
* The package provides methods for making HTTP requests, such as get(), post(), put(), delete(), and others.
* The package also provides classes for working with HTTP headers and cookies, such as the Headers and Cookie classes. Additionally, the package includes support for parsing and encoding data in popular formats such as JSON and XML.

Overall, the http package is a useful tool for working with HTTP requests in a Flutter app, and it is widely used by developers to communicate with APIs and other backend services.

**Functions on HTTP:**

The http package in Flutter provides several functions for making HTTP requests and handling responses. Here are some of the most commonly used functions:

**get():** Sends an HTTP GET request to the specified URL and returns the response as a Future<http.Response>.

http.get(Uri.parse('https://example.com/data.json'))

**post():** Sends an HTTP POST request to the specified URL and returns the response as a Future<http.Response>.

http.post(Uri.parse('https://example.com/api/post'), body: {'key1': 'value1', 'key2': 'value2'})

**put():** Sends an HTTP PUT request to the specified URL and returns the response as a Future<http.Response>.

http.put(Uri.parse('https://example.com/api/put'), body: {'key1': 'value1', 'key2': 'value2'})

**delete()**: Sends an HTTP DELETE request to the specified URL and returns the response as a Future<http.Response>.

http.delete(Uri.parse('https://example.com/api/delete'))

**read()**: Sends an HTTP GET request to the specified URL and returns the response body as a Future<String>.

http.read(Uri.parse('https://example.com/data.json'))

**Post Json data ()**: Sends an HTTP POST request with a JSON body to the specified URL and returns the response as a Future<http.Response>.

http.post(Uri.parse('https://example.com/api/post'), headers: {'Content-Type': 'application/json'}, body: json.encode({'key1': 'value1', 'key2': 'value2'}))

These are just some of the functions provided by the http package in Flutter. **For more advanced use cases, there are additional functions and classes available, such as MultipartRequest, MultipartFile, and others**.

For project example go through the github repo - <https://github.com/subhashDev11/flutter_mvc>

In this project, I have used a http package for API queries.

WebSockets are used to connect with the server just like the http package. It supports two-way communication with a server without polling.

In this article we will explore the below-listed topics related to WebSockets in Flutter:

1. Connecting to a WebSocket server
2. Listen to messages from the server.
3. Send data to the server.
4. Close the WebSocket connection.

In this article as an example, we will connect to the [test server provided by websocket.org.](http://www.websocket.org/echo.html)

### Connect to a WebSocket Server:

The web\_socket\_channel package has tools that are needed to connect to a WebSocket server. The package provides a WebSocketChannel that allows users to both listen to messages from the server and push messages to the server.

In Flutter, use the following line to create a WebSocketChannel that connects to a server:

**final channel = IOWebSocketChannel.connect('ws://echo.websocket.org');**

### **Listen to messages from the server:**

Now that we have established the connection to the server, we will send a message to it and get the same message as a response:

StreamBuilder(

stream: widget.channel.stream,

builder: (context, snapshot) {

return Text(snapshot.hasData ? '${snapshot.data}' : '');

},

);

### **Send Data to the Server:**

To send data to the server, add() messages to the sink provided by the WebSocketChannel as shown below:

**channel.sink.add('Hello Subhash!');**

### Close the Connection:

To close the connection to the WebSocket use the below:

**channel.sink.close();**

**Example -**

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

**import 'package:web\_socket\_channel/io.dart';**

**void main() {**

**runApp(const MaterialApp(**

**home: WebSocketExample(),**

**));**

**}**

**class WebSocketExample extends StatefulWidget {**

**const WebSocketExample({super.key});**

**@override**

**\_WebSocketExampleState createState() => \_WebSocketExampleState();**

**}**

**class \_WebSocketExampleState extends State<WebSocketExample> {**

**IOWebSocketChannel? channel;**

**final TextEditingController \_controller = TextEditingController();**

**List<String> messages = [];**

**@override**

**void initState() {**

**super.initState();**

**connectToWebSocket();**

**}**

**void connectToWebSocket() {**

**channel = IOWebSocketChannel.connect('wss://echo.websocket.org');**

**channel?.stream.listen((message) {**

**setState(() {**

**messages.add(message);**

**});**

**});**

**}**

**void sendMessage() {**

**if (\_controller.text.isNotEmpty) {**

**channel?.sink.add(\_controller.text);**

**\_controller.clear();**

**}**

**}**

**@override**

**Widget build(BuildContext context) {**

**return Scaffold(**

**appBar: AppBar(**

**title: const Text('WebSocket Example'),**

**),**

**body: Column(**

**children: [**

**Expanded(**

**child: ListView.builder(**

**itemCount: messages.length,**

**itemBuilder: (BuildContext context, int index) {**

**return Padding(**

**padding: const EdgeInsets.all(8.0),**

**child: Text(messages[index]),**

**);**

**},**

**),**

**),**

**Padding(**

**padding: const EdgeInsets.all(8.0),**

**child: TextField(**

**controller: \_controller,**

**decoration: InputDecoration(**

**hintText: 'Enter a message',**

**suffixIcon: IconButton(**

**icon: const Icon(Icons.send),**

**onPressed: sendMessage,**

**),**

**),**

**),**

**),**

**],**

**),**

**);**

**}**

**@override**

**void dispose() {**

**super.dispose();**

**channel?.sink.close();**

**}**

**}**

Output:



In this example, we're using the **IOWebSocketChannel** class from the web\_socket\_channel package to connect to the wss://echo.websocket.org server, which echoes back any messages it receives. We're also using a TextField widget to allow the user to enter messages and a ListView widget to display the messages that have been received. When the user enters a message and taps the send button, we send the message to the server using the sink property of the **IOWebSocketChannel** **instance**. The stream property of the IOWebSocketChannel instance allows us to listen for messages that the server sends back, and we update the messages list in the setState method to trigger a rebuild of the UI. Finally, we close the connection when the widget is disposed of using the close method of the sink property.

# Advance dart:

After basic dart and basic flutter tutorials we need to move to advance dart concepts.

In the advanced dart section we will learn about dart core libraries, dart collections and asynchronous programming.

## Core libraries:

Dart is a programming language that has a wide range of core libraries that provide a variety of functionalities for developers. Here are some of the Dart core libraries:

1. **dart:core -** This library contains the fundamental classes and functions that are used throughout Dart. It includes basic types such as String, int, double, and bool, as well as functions for working with collections, dates, and exceptions.
2. **dart:async -** This library provides support for asynchronous programming in Dart. It includes classes for working with futures, streams, and timers, and is essential for building modern web and mobile applications.
3. **dart:io -** This library provides classes for working with files, directories, sockets, and other I/O operations. It also includes classes for working with HTTP requests and responses.
4. **dart:math -** This library provides a wide range of mathematical functions, including trigonometric, logarithmic, and exponential functions. It also includes functions for generating random numbers and working with complex numbers.
5. **dart:convert -** This library provides classes for encoding and decoding data in different formats, such as JSON, XML, and UTF-8. It is an essential library for working with data in web and mobile applications.
6. **dart:html -** This library provides classes for working with the HTML document object model (DOM) in web applications. It includes classes for working with HTML elements, events, and styles.

These are just some of the core libraries in Dart, and there are many more available for different use cases, such as working with databases, cryptography, and internationalization.

## Streams in dart:

Streams in Dart are a fundamental concept in **asynchronous programming**. A stream is a sequence of **asynchronous events**, and it provides a way for code to handle events as they occur rather than waiting for a single result to be returned.

In Dart, streams are defined in the **dart:async library**. A stream can be seen as a source of asynchronous data. The stream can emit a sequence of values over time, and Dart code can listen to the stream to receive these values.

Here is an example of creating and using a simple stream in Dart:

import 'dart:async';

void main() {

final streamController = StreamController<int>();

streamController.stream.listen((value) {

print(value);

});

streamController.sink.add(1);

streamController.sink.add(2);

streamController.sink.add(3);

streamController.close();

}

In this example, we create a **StreamController** that can emit integers. We then listen to the stream using the listen method, which takes a callback that will be called every time a new value is emitted. We then add three values to the stream using the sink property, and finally, we close the stream using the close method.

**Streams are very useful in Dart for building responsive and interactive applications, especially in web and mobile development. By using streams, you can easily handle multiple asynchronous events in a more organized way, making your code more readable and maintainable.**

**Flutter stream builder:**

Flutter provides a StreamBuilder widget that allows you to easily listen to a stream and rebuild a widget whenever new data is received. The StreamBuilder widget listens to the stream and automatically rebuilds the widget tree with the latest snapshot data.

Here is an example of how to use the StreamBuilder widget to build a ListView of messages received over a stream:

import 'package:flutter/material.dart';

import 'dart:async';

void main(){

runApp(const MaterialApp(

home: MessageList(),

));

}

class MessageList extends StatefulWidget {

const MessageList({super.key});

@override

\_MessageListState createState() => \_MessageListState();

}

class \_MessageListState extends State<MessageList> {

final StreamController<List<String>> \_streamController =

StreamController<List<String>>();

@override

void initState() {

addMessage();

super.initState();

}

void addMessage(){

Stream.periodic(const Duration(microseconds: 200)).listen((event) {

\_streamController.sink.add(

[

'New message1 after 200 ms',

'New message2 after 200 ms',

],

);

});

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('Message List'),

),

body: StreamBuilder<List<String>>(

stream: \_streamController.stream,

builder: (BuildContext context, AsyncSnapshot<List<String>> snapshot) {

if (snapshot.hasData) {

return ListView.builder(

itemCount: snapshot.data!.length,

itemBuilder: (BuildContext context, int index) {

return ListTile(

title: Text(snapshot.data![index]),

);

},

);

} else {

return const Center(

child: CircularProgressIndicator(),

);

}

},

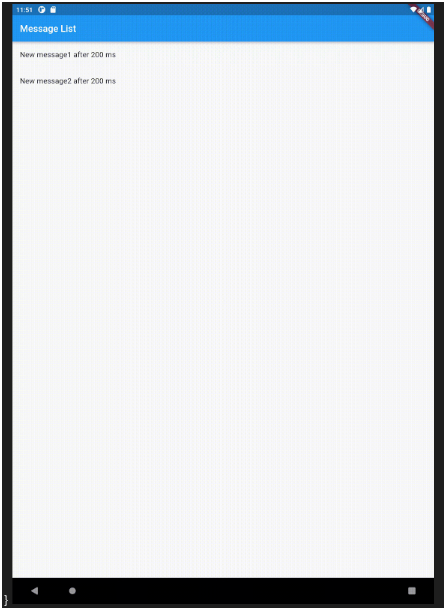
),

);

}

In this example, we have a **StreamController** that emits a list of messages as strings. The StreamBuilder widget listens to this stream and rebuilds the widget tree with the latest snapshot data. If the snapshot has data, we build a ListView that displays the list of messages. If there is no data in the snapshot, we show a CircularProgressIndicator while waiting for data to arrive.

The builder callback is called every time the stream emits a new value. The snapshot argument contains the latest value emitted by the stream. You can use this value to update the UI of your app. The **StreamBuilder** widget takes care of unsubscribing from the stream when the widget is no longer in the widget tree.



## Future in dart:

In Dart, a Future represents a computation that may not have completed yet, but will eventually produce a result. A Future is used for asynchronous operations that take some time to complete, such as network requests or file I/O. You can think of a Future as a placeholder for a result that will be available in the future.

Here is an example of how to create and use a Future in Dart:

Future<int> fetchNumber() {

return Future.delayed(Duration(seconds: 2), () => 42);

}

void main() {

print('Fetching number...');

fetchNumber().then((value) {

print('Number: $value');

});

}

Output:



In this example, we define a function fetchNumber that returns a Future that will resolve to the number 42 after a delay of two seconds. We then call fetchNumber and use the then method to print the value of the number when it becomes available.

When a Future is created, it is in an uncompleted state. The then method allows you to register a callback that will be called when the Future completes, with the result of the computation as an argument.

**You can also use the await keyword to wait for a Future to complete before continuing execution of your code.**

Future<int> fetchNumber() {

return Future.delayed(Duration(seconds: 2), () => 42);

}

void main() async {

print('Fetching number...');

int number = await fetchNumber();

print('Number: $number');

}



In this example, we use the await keyword to wait for the fetchNumber Future to complete before printing the number. The await keyword can only be used in an async function.

Futures are an essential part of asynchronous programming in Dart and are used extensively in Flutter for building responsive and interactive applications.

## 

## Collections in dart:

In Dart, a collection is an object that groups multiple elements into a single unit. There are three types of collections in Dart: **lists, sets, and maps**.

* **Lists:** are ordered collections of elements, and they can contain duplicates. You can access elements of a list using an index.
* **Sets:** are unordered collections of unique elements. They do not allow duplicates, and you cannot access elements using an index. Instead, you can use methods like contains to check if an element is in the set.
* **Maps:** are collections of key-value pairs, where each key is unique. You can access values in a map using a key.

Here are some examples of how to use collections in Dart:

**void main() {**

**// Lists**

**List<int> numbers = [1, 2, 3];**

**print(numbers[0]); // prints 1**

**numbers.add(4);**

**print(numbers); // prints [1, 2, 3, 4]**

**// Sets**

**Set<String> names = {'Subhash', 'Chandra', 'Shukla'};**

**print(names.contains('Subhash')); // prints true**

**names.add('Brij');**

**print(names); // prints {Subhash, Chandra, Shukla, Brij}**

**// Maps**

**Map<String, int> ages = {'Subhash': 25, 'Chandra': 30, 'Shukla': 35};**

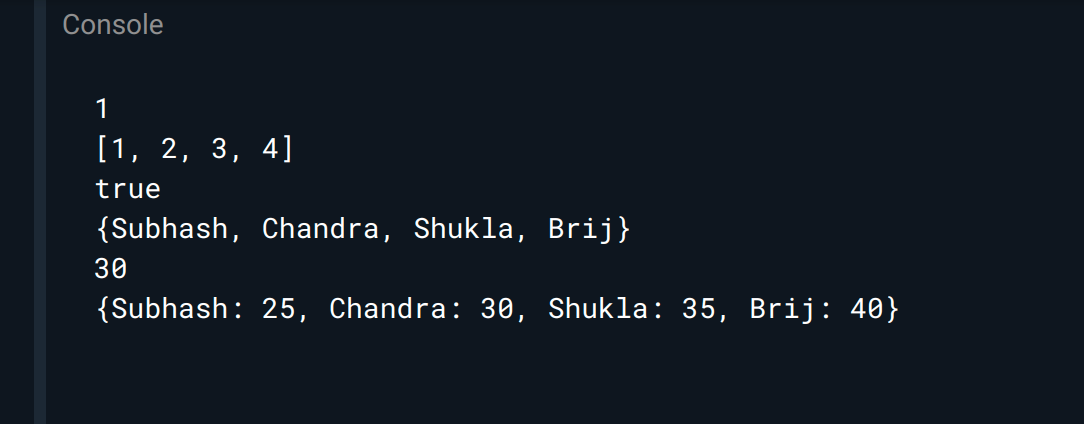
**print(ages['Chandra']); // prints 30**

**ages['Brij'] = 40;**

**print(ages); // prints {Subhash: 25, Chandra: 30, Shukla: 35, Brij: 40}**

**}**

Output:

****

In this example, we create a list of integers, a set of strings, and a map of strings to integers. We then perform several operations on each collection, such as adding elements, accessing elements, and checking for the presence of an element.

Dart also provides several methods for working with collections, such as forEach, map, and reduce, which allow you to perform operations on the elements of a collection. The Iterable class provides many of these methods, and most collection types in Dart implement the Iterable interface.

Collections are an essential part of Dart and Flutter programming, and they provide a flexible and efficient way to store and manipulate groups of data.

### 

### List in dart:

In Dart, a List is a collection of ordered elements, and it is similar to an array in other programming languages. Lists can be used to store homogeneous data types (e.g. a list of integers) or heterogeneous data types (e.g. a list of objects).

Here is an example of how to create and use a List in Dart:

void main() {

///Initialize list

List<int> numbers = [1, 2, 3, 4, 5];

///print list items

print(numbers); // prints [1, 2, 3, 4, 5]

///print list index item

print(numbers[0]); // prints 1

/// add item in list

numbers.add(6);

print(numbers); // prints [1, 2, 3, 4, 5, 6]

/// remove item from index 0

numbers.removeAt(0);

print(numbers); // prints [2, 3, 4, 5, 6]

/// insert item at index - 0

numbers.insert(0, 1);

print(numbers); // prints [1, 2, 3, 4, 5, 6]

/// shuffle list item

numbers.shuffle();

print(numbers); // prints a shuffled version of the list

}

**Output:**



In this example, we create a List of integers and perform several operations on it. We can access elements of the list using square brackets and an index. We can add elements to the end of the list using the add method, remove elements at a specific index using the removeAt method, insert elements at a specific index using the insert method, and shuffle the list using the shuffle method.

You can also create an empty list and add elements to it later:

**void main() {**

**List<String> names = [];**

**names.add('SUBHASH');**

**names.add('CHANDRA');**

**names.add('SHUKLA');**

**print(names); // prints ['SUBHASH', 'CHANDRA', 'SHUKLA']**

**}**

In this example, we create an empty List of strings and add elements to it using the add method.

Lists are used extensively in Dart and Flutter, and they provide a flexible and efficient way to store and manipulate collections of data.

### Map in dart:

In Dart, a map is a collection of key-value pairs, where each key is associated with a value. Maps can be used to represent data that has a structure similar to a dictionary.

To create a map in Dart, you can use curly braces {} to enclose the key-value pairs, with a colon : separating each key-value pair. For example:

var myMap = {'apple': 1, 'banana': 2, 'orange': 3};

In this example, the keys are strings ('apple', 'banana', and 'orange') and the values are integers (1, 2, and 3).

You can also create an empty map using the Map constructor, like this:

var myEmptyMap = Map();

To add a new key-value pair to a map, you can use the square bracket notation ([]) and assign a value to the key. For example:

**myMap['pear'] = 4;**

To access the value associated with a key in a map, you can also use the square bracket notation. For example

**var valueOfBanana = myMap['banana'];**

You can iterate over the keys or values of a map using a for-in loop or a forEach method. For example:

**myMap.keys.forEach((key) => print(key));**

**myMap.values.forEach((value) => print(value));**

Maps can also be nested to represent more complex data structures. For example:

**var myNestedMap = {**

**'fruits': {**

**'apple': 1,**

**'banana': 2,**

**'orange': 3,**

**},**

**'vegetables': {**

**'carrot': 4,**

**'broccoli': 5,**

**'spinach': 6,**

**},**

**};**

In this example, the keys are strings ('fruits' and 'vegetables') and the values are maps, each with their own key-value pairs.

### Sets in dart:

In Dart, a set is a collection of unique elements. That means that each element in the set is unique, and there can be no duplicates.

Here's an example of how to create a set in Dart:

**Set<String> mySet = {'apple', 'banana', 'orange'};**

In this example, we create a set of strings called mySet. We use curly braces to define the set and separate the elements with commas.

We also specify the type of the elements in the set by using the Set<String> syntax. This tells Dart that the set will only contain strings.

You can also create an empty set in Dart:

**Set<int> mySet = {};**

In this example, we create an empty set of integers called mySet. Note that we have to specify the type of the set, even though it's empty.

You can add elements to a set using the add() method:

**mySet.add(1);**

**mySet.add(2);**

**mySet.add(3);**

In this example, we add the integers 1, 2, and 3 to the set.

You can check the size of a set using the length property:

**print(mySet.length);**

This will output 3, which is the number of elements in the set.

You can remove elements from a set using the remove() method:

**mySet.remove(2);**

In this example, we remove the integer 2 from the set.

You can also check if a set contains a specific element using the contains() method:

**if (mySet.contains(1)) {**

**print('The set contains 1.');**

**}**

This will output The set containing 1. because the set contains the integer 1.

**Sets are useful for many applications, such as removing duplicates from a list or checking if an element is in a collection.**

## Lambdas in dart:

In Dart, a lambda function is also known as a "closure". A closure is an anonymous function that can be used as a value. It can capture variables from the enclosing scope and use them within its body.

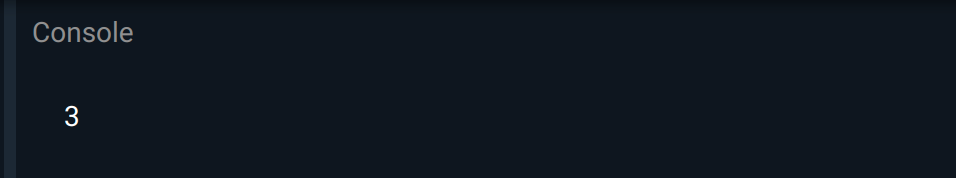
Here is an example of a closure in Dart:

**void main() {**

**var addNumbers = (int a, int b) => a + b;**

**print(addNumbers(1, 2)); // Output: 3**

**}**



In this example, we create a closure called addNumbers that takes two integer parameters and returns their sum. We use **the => symbol** to define the function body. We can then call the closure like any other function.

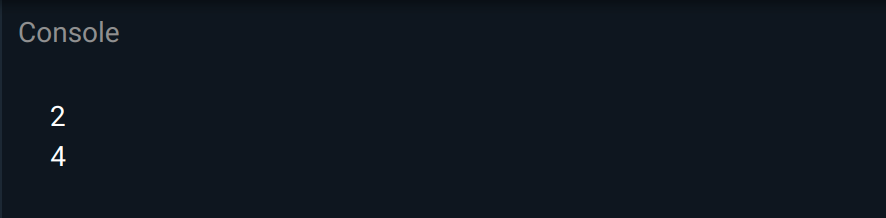
We can also use closures with Dart's collection methods such as map, forEach, and where. Here's an example:

void main() {

var numbers = [1, 2, 3, 4, 5];

numbers.where((n) => n % 2 == 0).forEach(print); // Output: 2 4

}



In this example, we use the **where** method to filter the even numbers from the numbers list. We pass a closure that takes one parameter n and returns a boolean that is true if n is even. We then use the forEach method to print out the filtered list.

## Functional programing in dart:

Dart supports functional programming, which is a programming paradigm that emphasizes the use of functions to solve problems. In functional programming, functions are treated as first-class citizens, meaning that they can be passed around as arguments, returned as values, and stored in variables.

Here are some examples of functional programming concepts in Dart:

**Anonymous Functions or Closures**

As we saw in the previous question, Dart supports closures or anonymous functions. We can define a closure and use it to pass as an argument to another function or store it in a variable.

**Higher-Order Functions**

Dart also supports higher-order functions, which are functions that take other functions as arguments or return functions as results. Here's an example:

**void main() {**

**var numbers = [1, 2, 3, 4, 5];**

**var squaredNumbers = numbers.map((n) => n \* n);**

**print(squaredNumbers); // Output: (1, 4, 9, 16, 25)**

**}**

****

In this example, we use the map function, which takes a closure as an argument and applies it to each element in the numbers list. The closure returns the square of the number, and the map function returns a new list with the squared numbers.

### Immutability:

Functional programming promotes immutability, which means that data structures and objects are not modified directly. Instead, new objects are created with the updated values. Dart has built-in support for immutable data structures, such as List.unmodifiable and Map.unmodifiable.

### Recursion:

Functional programming also heavily relies on recursion, which is a technique that involves a function calling itself until it reaches a base case. In Dart, we can use recursion to implement algorithms such as factorial, Fibonacci sequence, and others.

In summary, Dart supports several functional programming concepts that enable developers to write more concise and elegant code.

## Isolates in dart:

In Dart, isolates are a mechanism for concurrent programming. An isolate is an independent unit of execution that has its own memory heap, and it cannot share memory with other isolates. Isolates communicate with each other by passing messages.

Isolates are similar to threads, but they have some important differences. Threads share the same memory space, which can lead to problems such as race conditions and deadlock. Isolates, on the other hand, do not share memory, so they cannot have these problems.

Here's an example of creating and using an isolate in Dart:

import 'dart:isolate';

void main() async {

ReceivePort receivePort = ReceivePort();

await Isolate.spawn(isolateFunction, receivePort.sendPort);

receivePort.listen((message) {

print('Received message: $message');

});

}

void isolateFunction(SendPort sendPort) {

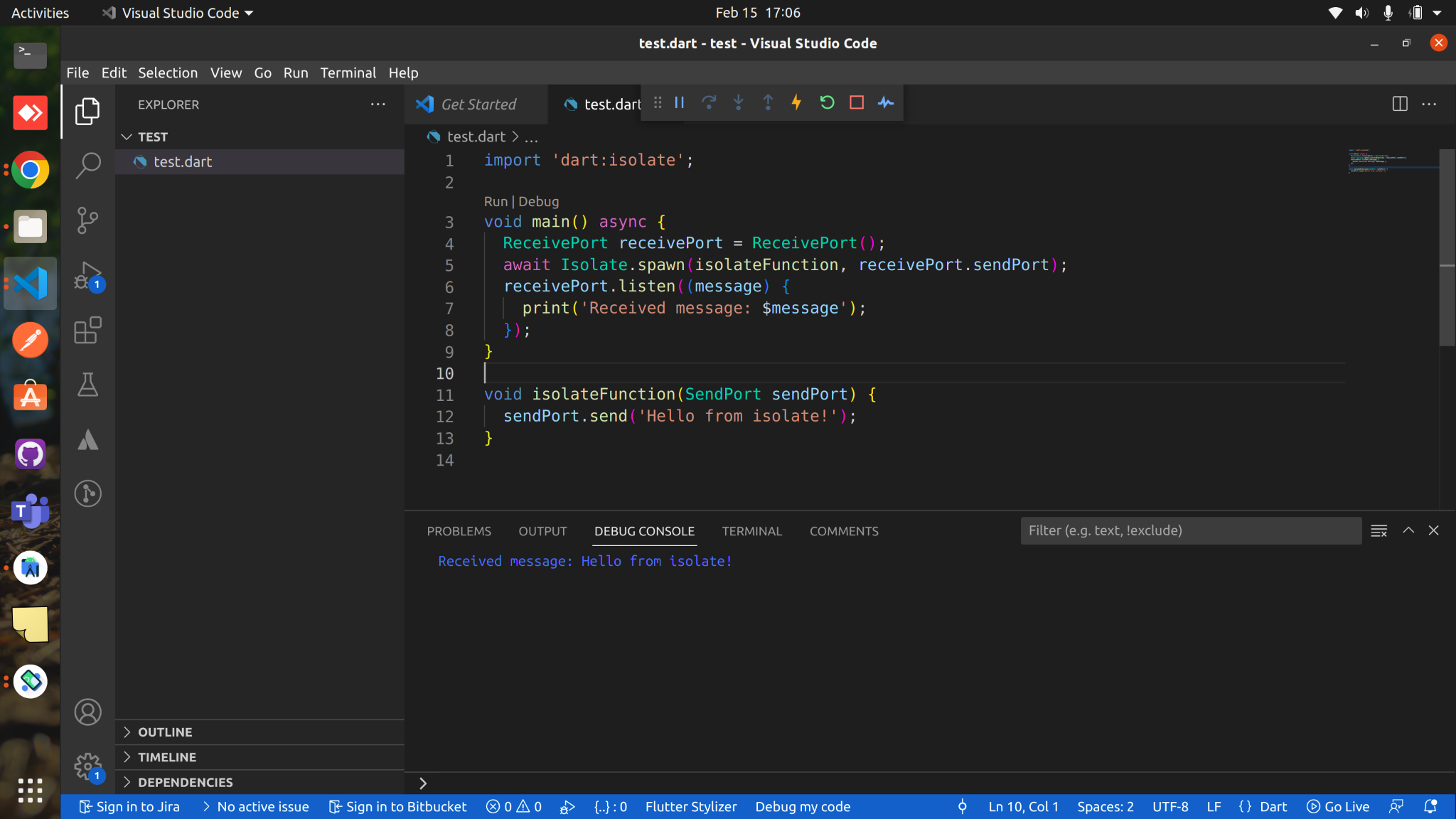
sendPort.send('Hello from isolate!');

}

In this example, we create an isolate using the Isolate.spawn method. We pass a function called isolateFunction as an argument, and we also pass a SendPort that will be used to send messages back to the main isolate.

The isolateFunction sends a message back to the main isolate using the sendPort.send method. The main isolate listens for messages using a ReceivePort that we created earlier.

When we run this code, the output will be:



**This is just a simple example, but isolates can be used for more complex tasks such as parallel processing, background computation, and more. Note that isolates have some overhead, so they are not suitable for all use cases. However, they can be a powerful tool for concurrent programming in Dart.**

## async() and await in dart():

In Dart, async and await are keywords that enable developers to write asynchronous code in a more readable and understandable way. Asynchronous code is code that can run concurrently without blocking the execution of the program. Common examples of asynchronous operations include network requests, file I/O, and user input.

Here's an example of using async and await to perform a network request:

import 'dart:async';

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

Future<void> main() async {

try {

var response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/todos/1'));

print('Response status: ${response.statusCode}');

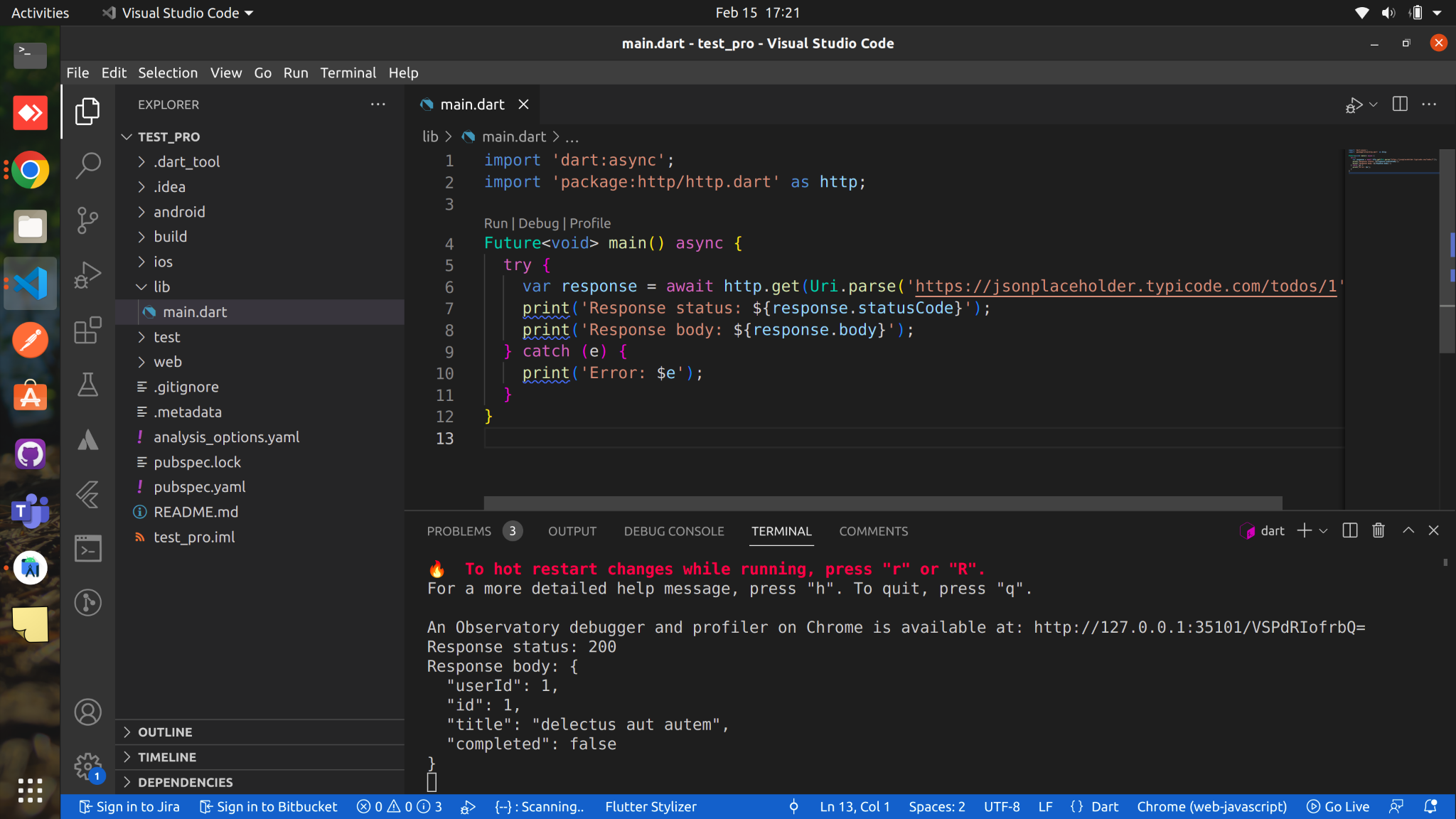
print('Response body: ${response.body}');

} catch (e) {

print('Error: $e');

}

}



In this example, we import the http package, which provides a get method for making HTTP requests. We use the **async keyword** to mark the main **function as asynchronous**. We also use the **await keyword** to wait for the response from the server.

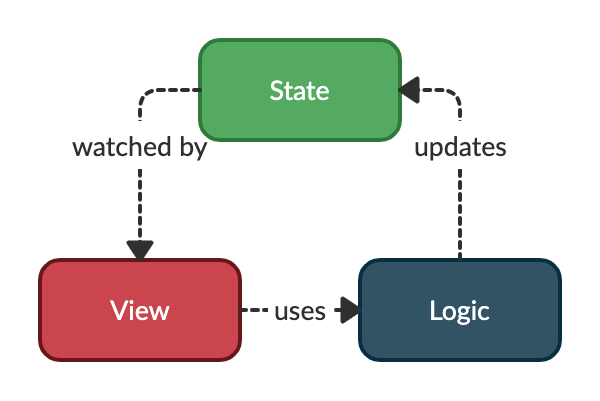
The **http.get** method returns a Future object, which represents a value that may not be available yet. By using the await keyword, we are telling Dart to wait for the Future to complete and return a value. If the Future throws an error, we can catch it using a **try-catch block**.

In summary, **async** and **await** are powerful tools that enable developers to write **asynchronous** code in a more natural and readable way. By using these keywords, we can avoid callbacks and complex control flow, and write code that is easier to understand and maintain.

# Flutter State Management:

## What is the State?

In the software development world, the state is the information you require to rebuild your UI at any moment.



**Why State Management?**

State management is very important in application development. It centralizes all the states of various UI controls to handle data flow across the application. A proper state management can save a lot of computational overhead by avoiding rebuilding of unnecessary widgets.

**🤔Thinks some declarative**

***Flutter is declarative. This means that Flutter builds its user interface to reflect the current state of your app***



***When the state of your app changes (for example, you press that increment counter button), you change the state, and that triggers a redraw of the user interface. There is no imperative changing of the UI itself (like widget.setText) — you change the state, and the UI rebuilds from scratch.***

**Types of state in flutter :**

**1. Ephemeral State**

Ephemeral state (sometimes called UI state or local state) is the state you can nearly contain in a single widget. **Example: on/off state of a switch widget.**

**2. App State**

State that is not ephemeral, that you want to share across many parts of your app, and that you want to keep between user sessions, is what we call an application state . **Example: Add to Cart — count on dashboard and list on Cart List page.**

**Managing the State:**

Consider an e-commerce app, you add a product in the cart and you want your orders screen (cart screen) or any screen where changes need to be made to update its state without calling **setState()** in that screen’s widget hierarchy.

F**lutter State Management Options…**

Provider- <https://medium.com/@subhashchandrashukla/flutter-provider-c9516d024c0>

Bloc/Cubit- <https://medium.com/@subhashchandrashukla/flutter-cubit-7a5b4d62f36>

GetX-

Redux-

Mobx-

Riverpod- <https://medium.com/@subhashchandrashukla/riverpod-in-flutter-a4835aca9f17>

SetState — StateFull widget state

InheritedWidget- widget state

## Provider-

Flutter Provider is a Flutter library for **DI (dependency injection) and state management**. It allows you to create objects **(called “providers”)** that hold a value that can be accessed and shared throughout your Flutter app.

One of the key benefits of using Provider is that it allows you to manage the state of your app in a predictable and testable way. Instead of having to pass data down through multiple levels of widgets using context and BuildContext, you can simply create a provider and access its value from anywhere in your app.

Provider package installation guideline:

1. Add the following dependency to your pubspec.yaml file:dependencies:

dependencies:

flutter:

sdk: flutter

# The following adds the Cupertino Icons font to your application.

# Use the CupertinoIcons class for iOS style icons.

cupertino\_icons: ^1.0.2

provider: ^6.0.5

2. Install packages from the command line: **flutter pub get**

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

**import 'package:provider/provider.dart';**

**class Counter with ChangeNotifier {**

**int \_count = 0;**

**int get count => \_count;**

**void increment() {**

**\_count++;**

**notifyListeners();**

**}**

**}**

**void main() {**

**runApp(**

**ChangeNotifierProvider(**

**create: (\_) => Counter(),**

**child: MyApp(),**

**),**

**);**

**}**

**class MyApp extends StatelessWidget {**

**@override**

**Widget build(BuildContext context) {**

**final counter = Provider.of<Counter>(context);**

**return MaterialApp(**

**home: Scaffold(**

**body: Center(**

**child: Column(**

**mainAxisAlignment: MainAxisAlignment.center,**

**children: [**

**const Text(**

**'You have pushed the button this many times:',**

**),**

**Text(**

**'${counter.count}',**

**style: Theme.of(context).textTheme.headlineMedium,**

**),**

**],**

**),**

**),**

**floatingActionButton: FloatingActionButton(**

**onPressed: counter.increment,**

**tooltip: 'Increment',**

**child: const Icon(Icons.add),**

**),**

**),**

**);**

**}**

**}**

****

## Cubit-

In the Flutter framework, a **Cubit** is a variant of the **BLoC (Business Logic Component) pattern** that is used to manage the state of a Flutter app. Like the BLoC pattern, the Cubit pattern uses reactive streams to communicate between the business logic and the user interface.

The main difference between a Cubit and a BLoC is that a Cubit is a more lightweight version of the BLoC pattern. It is designed to be simpler to implement and easier to understand, while still providing many of the same benefits as the BLoC pattern.

A Cubit has a single state object that represents the current state of the app. The state is modified by emitting new states using the emit method, and the user interface can be rebuilt whenever the state changes by using a BlocBuilder widget.

Like the BLoC pattern, the Cubit pattern can help to make the code more maintainable and easier to test by separating the business logic from the user interface. It can also make it easier to reason about the state of the app, as the state is explicitly defined and can be easily tracked.

import 'package:flutter/material.dart';

import 'package:flutter\_bloc/flutter\_bloc.dart';

main() {

runApp(const MyApp());

}

class CounterCubit extends Cubit<int> {

CounterCubit() : super(0);

void increment() => emit(state + 1);

void decrement() => emit(state - 1);

}

class MyApp extends StatelessWidget {

const MyApp({Key? key}) : super(key: key);

@override

Widget build(BuildContext context) {

return BlocProvider(

create: (\_) => CounterCubit(),

child: const MaterialApp(

home: CounterPage(),

),

);

}

}

class CounterPage extends StatelessWidget {

const CounterPage({super.key});

@override

Widget build(BuildContext context) {

final counterCubit = context.read<CounterCubit>();

return Scaffold(

body: BlocBuilder<CounterCubit, int>(

builder: (context, count) {

return Center(

child: Text(

'$count',

style: const TextStyle(fontSize: 24),

),

);

},

),

floatingActionButton: Column(

mainAxisAlignment: MainAxisAlignment.end,

children: <Widget>[

FloatingActionButton(

onPressed: counterCubit.increment,

child: const Icon(Icons.add),

),

const SizedBox(height: 10),

FloatingActionButton(

onPressed: counterCubit.decrement,

child: const Icon(Icons.remove),

),

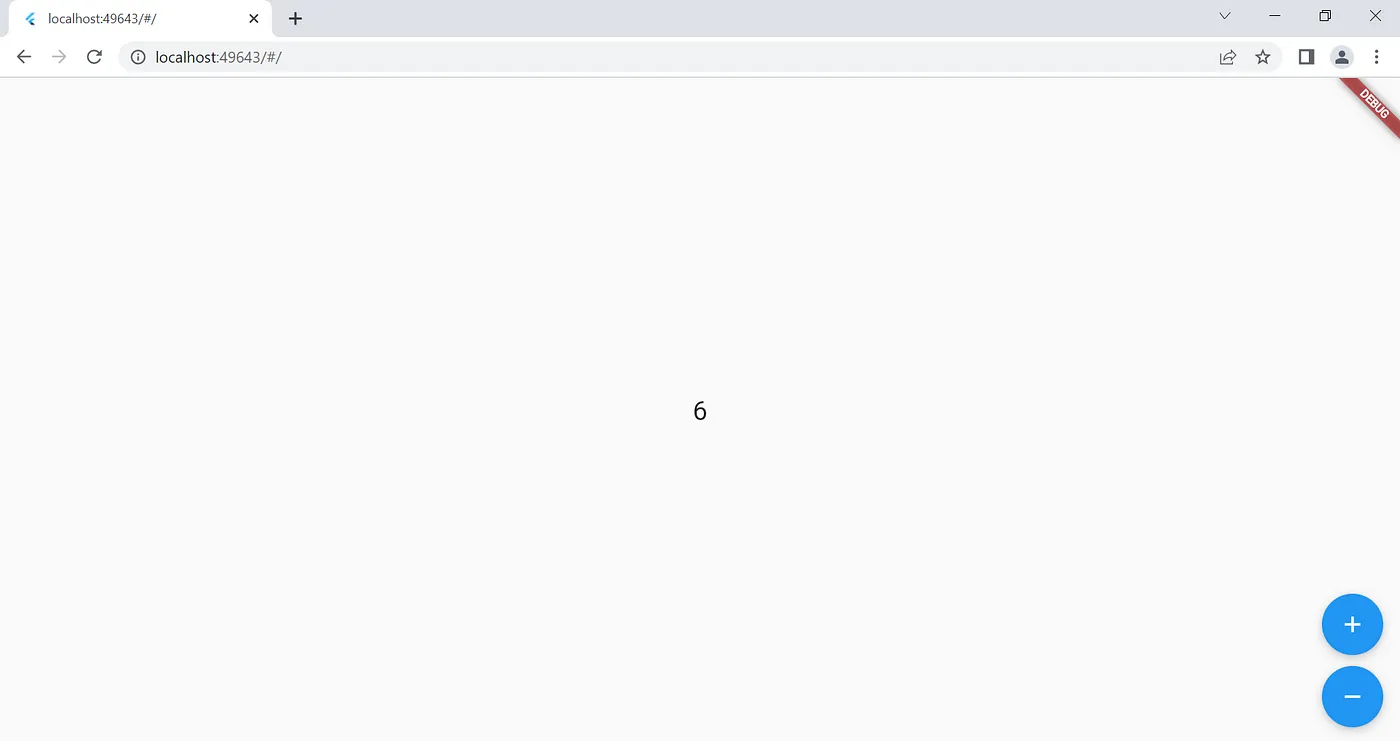
],

),

);

}

}



This example shows a simple counter app that displays the current count on the screen and has buttons to increment and decrement the count. The count is managed by a CounterCubit object, which is a Cubit that stores an integer value. The CounterCubit has methods to increment and decrement the count, and it emits a new state whenever the count changes.

The CounterPage widget displays the current count using a BlocBuilder widget, which rebuilds the user interface whenever the count changes. The CounterPage also has floating action buttons that call the increment and decrement methods of the CounterCubit when they are pressed.

## Riverpod-

Flutter Riverpod is a state management library for Flutter that allows you to write predictable and testable code. It is built on top of the popular Provider library, but adds additional features such as support for async actions and improved type safety.

One of the key features of Riverpod is the ability to manage state using “providers”. Providers are simply objects that contain a value that can be accessed by other parts of your app. You can create a provider for a particular piece of state by using the Provider class and then access the value of the provider using the Provider.of method.

Riverpod package installation guideline:

1. Add the following dependency to your pubspec.yaml file:dependencies:

dependencies:

flutter:

sdk: flutter

# The following adds the Cupertino Icons font to your application.

# Use the CupertinoIcons class for iOS style icons.

cupertino\_icons: ^1.0.2

flutter\_riverpod: ^2.2.0

2. Install packages from the command line: **flutter pub get**

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

**import 'package:flutter\_riverpod/flutter\_riverpod.dart';**

**// A Counter example implemented with riverpod**

**void main() {**

**runApp(**

**// Adding ProviderScope enables Riverpod for the entire project**

**const ProviderScope(child: MyApp()),**

**);**

**}**

**class MyApp extends StatelessWidget {**

**const MyApp({super.key});**

**@override**

**Widget build(BuildContext context) {**

**return const MaterialApp(home: Home());**

**}**

**}**

**/// Providers are declared globally and specify how to create a state**

**final counterProvider = StateProvider((ref) => 0);**

**class Home extends ConsumerWidget {**

**const Home({super.key});**

**@override**

**Widget build(BuildContext context, WidgetRef ref) {**

**return Scaffold(**

**appBar: AppBar(title: const Text('Counter example')),**

**body: Center(**

**// Consumer is a widget that allows you reading providers.**

**child: counterDisplay(ref),**

**),**

**floatingActionButton: counterIncrement(ref),**

**);**

**}**

**Widget counterIncrement(ref) {**

**return FloatingActionButton(**

**// The read method is a utility to read a provider without listening to it**

**onPressed: () => ref.read(counterProvider.notifier).state++,**

**child: const Icon(Icons.add),**

**);**

**}**

**Widget counterDisplay(ref) {**

**final count = ref.watch(counterProvider);**

**return Consumer(**

**builder: (context, ref, \_) {**

**return Text('$count');**

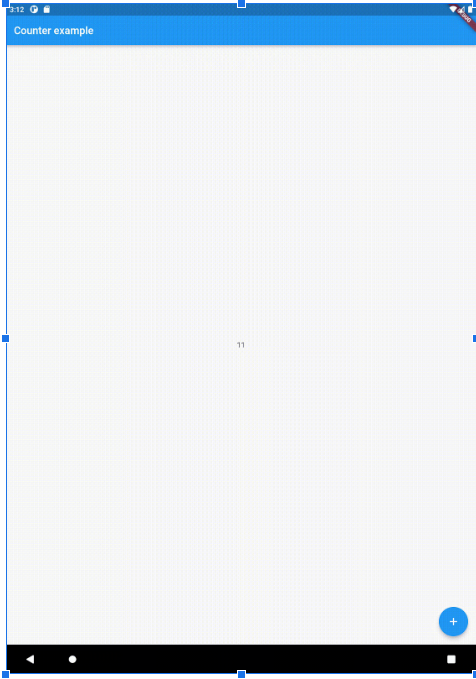
**},**

**);**

**}**

**}**

**Output:**

****

This example defines a counterProvider that holds an integer state value, which is initialized to 0. The CounterDisplay widget displays the current value of the counter, and the CounterIncrementor widget has a button that increments the counter when pressed. The ProviderScope widget at the root of the app's widget tree allows the counterProvider to be accessed by the CounterDisplay and CounterIncrementor widgets using the Consumer widget.

# Animation in flutter:

**Animation** in Flutter refers to the process of creating dynamic and engaging user interfaces by gradually changing the properties of a widget over time. **Animations** can be used to provide visual feedback, improve the user experience, and make your app feel more lively and responsive.

In Flutter, **animations** are managed by the **Animation API**. The API provides several built-in **animation widgets and classes, such as AnimatedContainer, AnimatedOpacity, AnimatedBuilder, Tween, and AnimationController,** which allow you to create and control **animations** easily.

**Animations** in Flutter are created by defining an **animation controller**, which is responsible for controlling the duration and behavior of the **animation**. You then define a tween, which specifies the range of values that the widget property will **animate** between. Finally, you attach the controller to the **animation** widget using the controller property, and the widget's property gradually changes over time.

Flutter provides excellent support for animation and can separate the animation into two main categories, which are given below:

* Tween Animation
* Physics-based Animation

## Tween Animation

It is the short form of in-betweening. In a tween animation, it is required to define the start and endpoint of animation. It means the animation begins with start value, then goes through a series of intermediate values and finally reached the end value. It also provides the timeline and curve, which defines the time and speed of the transition. The widget framework provides a calculation of how to transition from the start and endpoint.

ColorTween {

begin: color.green,

end: color.blue,

}

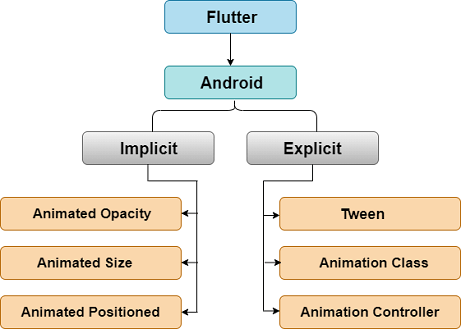
## Physics-based Animation

It is a type of animation which allows you to make an app interaction feels realistic and interactive. It simulates the real-world animation/movement, such as you want to animate a widget like spring, falling, or swinging with gravity. Thus, it is an animation that animates in response to user input/movement. The simplest example is the time of flight, and the distance of travel covered will be calculated according to the laws of physics.

Flutter provides two types of techniques for animation. These techniques are:

1. Implicit Animation
2. Explicit Animation

The following figure sets out the animation hierarchy in Flutter and explains it more clearly about the implicit and explicit animation.



Now, we are going to see how we can create explicit animation in Flutter. There are mainly three pillars of an animation, which are given below:

1. Ticker
2. Animation Class
3. AnimationController

## Ticker:

The Ticker is a class which sends a signal at a regular interval, i.e., around 60 times per second. You can understand it with your watch, which tics at regular intervals. At each tick, Ticker provides a callback method with the duration since the first ticks at each second, after it was started. Even if the tickers started at different times, it always synchronized automatically. The reason behind this is that the tickers give their elapsed time relative to the first tick after it was started.

## Animation:

The Animation class is the core building block of the animation system. The animation is nothing else, but it represents a value (specific type) that can change over the lifetime of an animation. In Flutter, the widgets which perform an animation take an animation object as a parameter. This Animation object gives the information from which they read the current value of the animation and to which they listen for changes to that value. The animation class contains two methods addListener() and addStatusListener(). When the value of animation changes, it notifies all the listeners added with addListener(). Again, when the status of the animation changes, it notifies all the listeners added with addStatusListener().

The most common Animation classes are:

* **Animation<double>:** It interpolates values between two decimal numbers over a certain duration.
* **Animation<Color>:** It interpolates colors between two color values.
* **Animation<Size>:** It interpolates sizes between two size values.

### Animation Controller:

The animation controller is a class that allows us to control the animation. It always generates new values whenever the application is ready for a new frame. For example, it gives the control of start, stop, forward, or repeat of the animation. Once the animation controller is created, we can start building other animation based on it, such as reverse animation and curved animation.

late AnimationController animationController;

@override

void initState() {

animationController = AnimationController(

vsync: this, duration: const Duration(

milliseconds: 2500,

),

);

super.initState();

}

Here, the duration option controls the duration of the animation process, and the vsync option is used to optimize the resource used in the animation.

The basic steps necessary for using an AnimationController are:

**Step 1:** First, instantiate an AnimationController with parameters, such as duration and vsync.

**Step 2:** Add the required listeners like addListener() or addStatusListener().

**Step 3:** Start the animation.

**Step 4:** Perform the action in the listener callback methods (for example, setState).

**Step 5:** Last, dispose of the animation.

Let us see a simple animation example, which uses an animation class and animation controller. The following example shows the tween animation that gives the start and endpoint of animation. Open the project and replace the following code in the main.dart file.

**import 'package:flutter/foundation.dart';**

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

**void main() => runApp(const MyApp());**

**class MyApp extends StatelessWidget {**

**const MyApp({super.key});**

**// This widget is the root of your application.**

**@override**

**Widget build(BuildContext context) {**

**return MaterialApp(**

**title: 'Flutter Animation',**

**theme: ThemeData(**

**// This is the theme of your application.**

**primarySwatch: Colors.blue,**

**),**

**home: const MyHomePage(),**

**);**

**}**

**}**

**class MyHomePage extends StatefulWidget {**

**const MyHomePage({super.key});**

**// ignore: library\_private\_types\_in\_public\_api**

**\_HomePageState createState() => \_HomePageState();**

**}**

**class \_HomePageState extends State<MyHomePage> with SingleTickerProviderStateMixin {**

**late Animation<double> animation;**

**late AnimationController animationController;**

**@override**

**void initState() {**

**super.initState();**

**animationController = AnimationController(vsync: this, duration: const Duration(milliseconds: 2500));**

**animation = Tween<double>(begin: 0.0, end: 1.0).animate(animationController);**

**animation.addListener((){**

**setState((){**

**if (kDebugMode) {**

**print (animation.value.toString());**

**}**

**});**

**});**

**animation.addStatusListener((status){**

**if(status == AnimationStatus.completed){**

**animationController.reverse();**

**} else if(status == AnimationStatus.dismissed) {**

**animationController.forward();**

**}**

**});**

**animationController.forward();**

**}**

**@override**

**Widget build(BuildContext context) {**

**return Center(**

**child: AnimatedLogo(**

**animation: animation,**

**)**

**);**

**}**

**}**

**class AnimatedLogo extends AnimatedWidget {**

**final Tween<double> \_sizeAnimation = Tween<double> (begin: 0.0, end: 500.0);**

**AnimatedLogo({Key? key,required Animation<double> animation}):super(key: key, listenable: animation);**

**@override**

**Widget build(BuildContext context) {**

**final Animation<double> animation = listenable as Animation<double>;**

**return Transform.scale(**

**scale: \_sizeAnimation.evaluate(animation),**

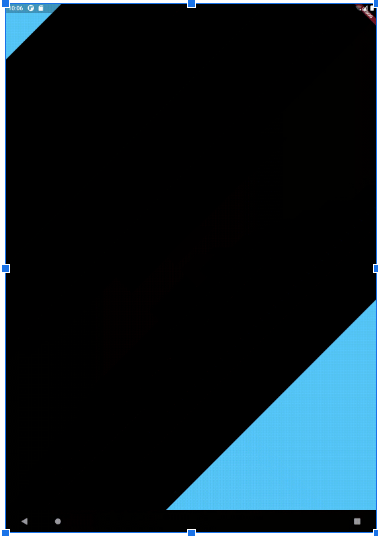
**child: const FlutterLogo(),**

**);**

**}**

**}**

**Output:** When you run this project , you will see the scaling logo on screen.



### Curved Animation:

The curved animation is very useful when you need to apply a non-linear curve with an animation object. Thus, it defines the animation's progress as a non-linear curve.

### Syntax:

CurvedAnimation(parent: animationController, curve: Curves.bounceOut);

Let us understand it with the previous example. To add a curve, open the previous app in the android studio and add **CurvedAnimation** instead of **animationController**. Or replace the following line:

animation = Tween<double>(begin: 0.0, end: 1.0).animate(animationController);

With the line below.

animation = Tween<double>(begin: 0.0, end: 1.0).animate(animationController);

Now, when you run the app, you will see a bouncing effect with the Flutter logo during scaling in forward and reverse direction.

## Hero Animation:

A hero animation is a type of animation where an element of one screen flies to a new screen when the app goes to the next page. We can understand it with the following example where an animation takes an element like icon/image, and once you tap on the icon, the screen flies to the next page. The following example explains it more clearly.

Open the Flutter app and replace the following code in the main.dart file.

import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

const MyApp({super.key});

@override

Widget build(BuildContext context) {

return MaterialApp(

title: 'Flutter Application',

theme: ThemeData(

primarySwatch: Colors.brown,

),

home: HeroAnimation(title: 'Hero Animation'),

);

}

}

class HeroAnimation extends StatefulWidget {

HeroAnimation({Key? key, this.title}) : super(key: key);

final String? title;

@override

\_HeroAnimationState createState() => \_HeroAnimationState();

}

class \_HeroAnimationState extends State<HeroAnimation> {

Widget \_greenRectangle() {

return Container(

width: 75,

height: 75,

color: Colors.greenAccent,

);

}

Widget \_detailPageRectangle() {

return Container(

width: 150,

height: 150,

color: Colors.redAccent,

);

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text(widget.title ?? ''),

),

body: buildDemoWidget(context),

);

}

Widget buildDemoWidget(BuildContext context) {

return Center(

child: Column(

crossAxisAlignment: CrossAxisAlignment.start,

children: <Widget>[

const SizedBox(

height: 30.0,

),

ListTile(

leading: GestureDetector(

child: Hero(

tag: 'hero-rectangle',

child: \_greenRectangle(),

),

onTap: () => \_gotoDetailsPageNav(context),

),

title: const Text('Tap on the this to analyze hero animation transition.'),

),

],

),

);

}

void \_gotoDetailsPageNav(BuildContext context) {

Navigator.of(context).push(MaterialPageRoute(

builder: (ctx) => Scaffold(

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

Hero(

tag: 'hero-rectangle',

child: \_detailPageRectangle(),

),

const Text('Detail page after navigation.'),

],

),

),

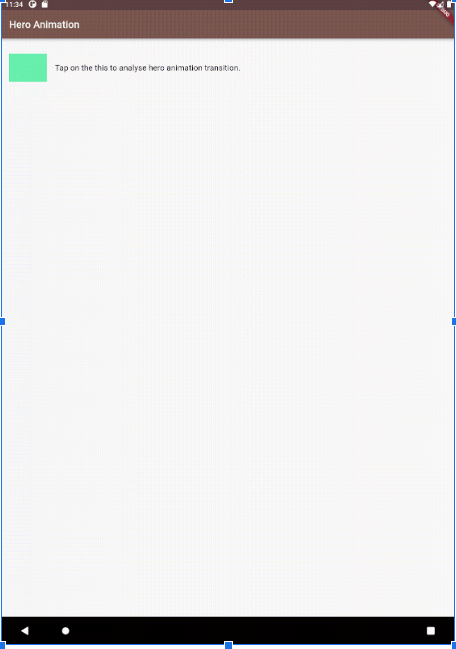
),

));

}

}

**Output:**



## Opacity Animation:

In Flutter, you can create an **opacity** animation using the **Opacity** widget and the **AnimatedOpacity** widget.

The **Opacity** widget is used to control the **opacity** of a child widget. You can set its **opacity** value with the **opacity** property, which takes a value between 0.0 (completely transparent) and 1.0 (completely opaque).

Here's an example of using the Opacity widget to animate the opacity of a child widget:

import 'package:flutter/material.dart';

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {

const MyApp({super.key});

@override

Widget build(BuildContext context) {

return MaterialApp(

title: 'Flutter Application',

theme: ThemeData(

primarySwatch: Colors.amber,

),

home: const OpacityWidget(),

);

}

}

class OpacityWidget extends StatefulWidget {

const OpacityWidget({super.key});

@override

// ignore: library\_private\_types\_in\_public\_api

\_OpacityWidgetState createState() => \_OpacityWidgetState();

}

class \_OpacityWidgetState extends State<OpacityWidget> {

bool \_isVisible = true;

@override

Widget build(BuildContext context) {

return Scaffold(

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Opacity(

opacity: \_isVisible ? 1.0 : 0.4,

child: const Text('Subhash, chandra!'),

),

ElevatedButton(

child: Text(\_isVisible ? 'Hide' : 'Show'),

onPressed: () {

setState(() {

\_isVisible = !\_isVisible;

});

},

),

],

),

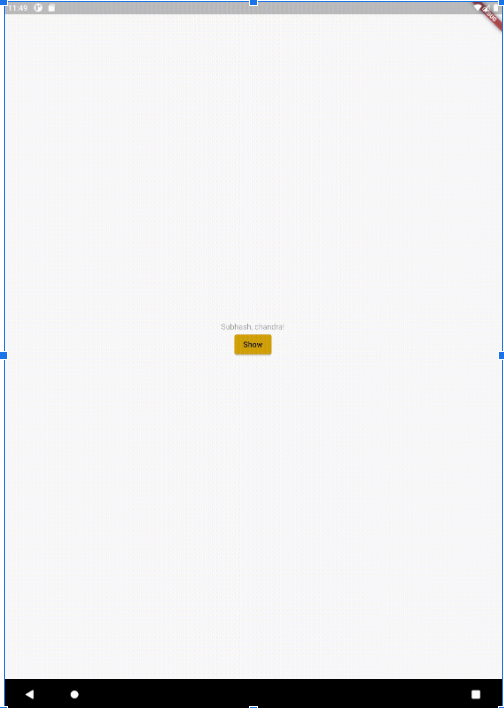
),

);

}

}

**Output:**



In this example, we use the \_isVisible boolean variable to control the opacity of the Text widget. When \_isVisible is true, the opacity is set to 1.0 (fully opaque), and when it's false, the opacity is set to 0.0 (fully transparent).

To create an animation that gradually changes the opacity value over time, we can use the AnimatedOpacity widget instead of the Opacity widget. The AnimatedOpacity widget is similar to the Opacity widget, but it animates the opacity value when it changes.

Here's an example of using the AnimatedOpacity widget to animate the opacity of a child widget:

# Testing in flutter:

Testing in Flutter is an essential part of building high-quality mobile applications. Flutter provides different types of tests to ensure the quality of the app, including unit tests, widget tests, and integration tests.

## Unit tests

Unit tests are used to test individual functions or classes of the app. These tests are written in Dart and run on a local machine or a continuous integration server. Flutter provides a test package, which includes a set of APIs for writing and running unit tests.

## Widget tests

Widget tests are used to test individual widgets in isolation. These tests simulate user interactions, such as tapping buttons or entering text, and check if the widget displays the expected output. Widget tests are also written in Dart and run on a local machine or a continuous integration server.

## Integration tests

Integration tests are used to test the interactions between different parts of the app. These tests simulate real-world scenarios, such as navigating between screens or making API calls, and check if the app behaves as expected. Integration tests are written in Dart and run on a device or an emulator.

Flutter provides a test framework, which makes it easy to write and run tests for the app. The framework includes APIs for mocking and stubbing, which allow developers to isolate and test individual components of the app.

Overall, testing is an important part of building a high-quality mobile app, and Flutter provides a robust set of tools and frameworks for testing the app at different levels of granularity.

# **Reactive programing in dart:**

Reactive programming in Dart is a programming paradigm that is used to handle asynchronous and event-driven programming. Dart provides a set of libraries for reactive programming, including **Stream** and **StreamController**.

The **Stream** library in Dart provides a way to represent a sequence of events or values over time. A **Stream** can be created from a variety of sources, such as an HTTP request, a file, or a user input. Once created, the **Stream** can be used to handle the events or values emitted over time, using methods such as **listen, map, where,** and **reduce**.

The **StreamController** library in Dart provides a way to control the flow of events in a **Stream**. A **StreamController** can be used to add events to a **Stream**, close a **Stream**, or handle errors in a **Stream**. Additionally, a **StreamController** can be used to create a new **Stream** from an existing **Stream**, by applying transformations or filters to the events emitted by the **Stream**.

Reactive programming in Dart is commonly used in the development of mobile and web applications, as it provides an efficient and intuitive way to handle asynchronous and event-driven programming. By using reactive programming, developers can write code that is more concise, efficient, and easier to maintain.

## RxDart:

rxdart is a popular library for reactive programming in Dart. It builds on top of the Stream and StreamController libraries in Dart and provides additional features and operators for working with reactive streams.

Some of the key features of rxdart include:

1. **Observable and Subject classes:** rxdart introduces these classes as abstractions for working with streams. An Observable is a read-only stream, while a Subject is a stream that allows for both reading and writing of events.
2. **Additional operators:** rxdart provides many additional operators for working with streams, including operators for combining streams, filtering events, transforming events, and more.
3. **Schedulers:** rxdart provides a way to control how and when events are emitted using schedulers. Schedulers can be used to run events on specific threads or at specific intervals.
4. **Error handling:** rxdart provides additional support for error handling in streams. For example, it provides operators for catching and handling errors in streams, and for retrying failed streams.

For example -

import 'package:rxdart/rxdart.dart';

/// generate n-amount of fibonacci numbers

///

/// for example: dart fibonacci.dart 10

/// outputs:

/// 1: 1

/// 2: 1

/// 3: 2

/// 4: 3

/// 5: 5

/// 6: 8

/// 7: 13

/// 8: 21

/// 9: 34

/// 10: 55

/// done!

void main(List<String> arguments) {

// read the command line argument, if none provided, default to 10

var n = (arguments.length == 1) ? int.parse(arguments.first) : 10;

// seed value: this value will be used as the

// starting value for the [scan] method

const seed = IndexedPair(1, 1, 0);

Rx

// amount of numbers to compute

.range(1, n)

// accumulator: computes a new accumulated

// value each time a [Stream] event occurs

// in this case, the accumulated value is always

// the latest Fibonacci number

.scan((IndexedPair seq, \_, \_\_) => IndexedPair.next(seq), seed)

// finally, print the output

.listen(print, onDone: () => print('done!'));

}

class IndexedPair {

final int n1, n2, index;

const IndexedPair(this.n1, this.n2, this.index);

factory IndexedPair.next(IndexedPair prev) => IndexedPair(

prev.n2, prev.index <= 1 ? prev.n1 : prev.n1 + prev.n2, prev.index + 1);

@override

String toString() => '$index: $n2';

}

Output:

Launching lib/main.dart on Android SDK built for x86 in debug mode...

✓ Built build/app/outputs/flutter-apk/app-debug.apk.

Connecting to VM Service at ws://127.0.0.1:42257/69qwuA\_XWcU=/ws

I/flutter ( 4982): 1: 1

I/flutter ( 4982): 2: 1

I/flutter ( 4982): 3: 2

I/flutter ( 4982): 4: 3

I/flutter ( 4982): 5: 5

I/flutter ( 4982): 6: 8

I/flutter ( 4982): 7: 13

I/flutter ( 4982): 8: 21

I/flutter ( 4982): 9: 34

I/flutter ( 4982): 10: 55

I/flutter ( 4982): done!

This code uses the Rx library from the rxdart package to generate a specified amount of Fibonacci numbers. The main function takes an argument n, which sets the amount of numbers to generate. If no argument is provided, it defaults to 10 numbers.

The function uses the scan method to compute new accumulated values each time a stream event occurs. This scan method receives as generators a previous sequence, \_ and \_\_ (which are not used in this case). The scan method then calls the IndexedPair() constructor which creates two indexed values n1 and n2 (the two immediately preceding Fibonacci numbers) and an index counter. A factory constructor for IndexedPair is then declared to create the next sequence of numbers based on the last one and add one to the counter.

Lastly, listen is used to print the new sequence until the total specified amount of numbers has been reached, and onDone prints done! when the process is complete.

Overall, rxdart is a powerful library for reactive programming in Dart, and it provides many useful features for working with streams and building reactive applications.

# Flutter internal

Flutter is a mobile development framework developed by Google, which allows developers to build high-performance, natively compiled applications for mobile, web, and desktop from a single codebase. Flutter provides a rich set of APIs, widgets, and tools for building beautiful and responsive user interfaces, as well as tools for managing state, handling networking, and integrating with other systems and services.

At its core, Flutter is built on top of the Dart programming language, which is an object-oriented, client-optimized language that is optimized for building high-performance, scalable applications. Flutter uses a reactive programming model, which allows developers to build UIs that are reactive to user input and can change over time based on changes to the app state.

Flutter has a layered architecture, with each layer providing specific functionality and APIs for building and managing different aspects of the app. Some of the key layers of the Flutter architecture include:

1. **The framework layer:** This layer provides the core widgets and APIs for building UIs in Flutter, such as Material Design and Cupertino widgets, as well as tools for managing app state, handling navigation, and integrating with other services.
2. **The engine layer:** This layer provides the low-level graphics and rendering capabilities for Flutter, including support for 2D and 3D graphics, animations, and effects.
3. **The embedder layer:** This layer provides the platform-specific code that allows Flutter to run on different platforms, such as Android, iOS, and web.
4. **The tooling layer:** This layer provides the development tools and workflows for building and testing Flutter apps, including the Flutter SDK, the Dart SDK, and tools like the Flutter DevTools and the Flutter CLI.

Overall, Flutter's architecture is designed to provide developers with a powerful, flexible, and efficient platform for building high-quality mobile, web, and desktop applications. By using Flutter, developers can take advantage of a rich set of APIs and tools to build beautiful and responsive apps that run smoothly across a variety of platforms and devices.

## Renderer engine in flutter:

Flutter uses a rendering engine called **"Skia"** to draw the user interface and graphics on the screen. Skia is an open-source 2D graphics library developed by Google that is widely used in many of their products, including Chrome, Android, and Chrome OS.

In Flutter, **Skia** is used as a platform-agnostic graphics engine that can draw on different platforms, including iOS, Android, web, and desktop. **Skia** draws graphics using a retained mode approach, which means that instead of rendering graphics directly to the screen, **Skia** maintains an internal representation of the graphics that can be modified and updated as needed.

Flutter's rendering engine also includes a **"compositor"** that takes the output from **Skia** and composites it with other layers in the app, such as text, images, and animations, to produce the final visual output that is displayed on the screen.

Overall, Flutter's rendering engine is designed to provide a fast and smooth user interface that is responsive to user input, even on devices with lower-end hardware.

# Flutter app deployment

Flutter apps can be deployed to both Android and iOS platforms. Here are the steps to deploy a Flutter app:

## Deploying to Android:

1. Update the pubspec.yaml file with the latest version of all dependencies and then run flutter pub get command to update the packages.
2. Run the following command to generate a release build of the app:

flutter build apk --release

This command will create a release APK file for your app in the build/app/outputs/apk/release/ directory.

1. Create a new keystore file, which will be used to sign the APK. You can create a new keystore file using the keytool command that comes with the Java Development Kit (JDK).

keytool -genkey -v -keystore key.jks -keyalg RSA -keysize 2048 -validity 10000 -alias key

1. Next, add the signing configuration to the android/app/build.gradle file

android {

// ...

signingConfigs {

release {

keyAlias 'key'

keyPassword 'password'

storeFile file('/path/to/key.jks')

storePassword 'password'

}

}

buildTypes {

release {

// ...

signingConfig signingConfigs.release

}

}

}

1. Replace /path/to/key.jks with the path to your keystore file and password with your chosen passwords.
2. Finally, run the following command to sign and align the release APK file:

-verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore key.jks /path/to/app-release-unsigned.apk key

-v 4 /path/to/app-release-unsigned.apk /path/to/app-release.apk

Replace /path/to/app-release-unsigned.apk with the path to your unsigned release APK file and /path/to/app-release.apk with the path to your signed and aligned release APK file.

Once you have the signed and aligned APK file, you can upload it to the Google Play Console for distribution.

## Deploying to iOS:

1. Open the terminal and navigate to the root of your Flutter project.
2. Connect your iOS device to your computer using a USB cable and open Xcode.
3. In Xcode, select your device from the list of available devices.
4. Run the command flutter run in the terminal. This will build and install the app on your connected iOS device.
5. If you want to build the app for release to the App Store, you will need to create a distribution certificate and provisioning profile, and then build and archive the app in Xcode. Once you have the archive, you can upload it to the App Store using App Store Connect.

These are the basic steps for deploying a Flutter app to Android and iOS platforms. However, the deployment process may vary depending on your app's requirements and the platform you are targeting.

Conclusion

Reading Rainbow Tip: It’s important to give your opinion! Would you recommend this book to someone else?

short dash