Unit-3: Introduction of Flutter:

3.1 Fundamentals of Flutter:

Flutter is a UI toolkit for building fast, beautiful, natively compiled applications for mobile, web, and desktop with one programing language and single codebase. It is free and open-source. Initially, it was developed from **Google** and now manages by an **ECMA standard**. Flutter apps use Dart programming language for creating an app.

The first version of Flutter was announced in the year **2015** at the Dart Developer Summit. It was initially known as codename **Sky** and can run on the Android OS. On **December 4, 2018**, the first stable version of the Flutter framework was released, denoting Flutter 1.0. The current stable release of the framework is Flutter 3.16.9 hotfix.6 on October 24, 2019. The dart current version is 3.2.6

What is Flutter?

In general, creating a mobile application is a very complex and challenging task. There are many frameworks available, which provide excellent features to develop mobile applications. For developing mobile apps, Android provides a native framework based on Java and Kotlin language, while iOS provides a framework based on Objective-C/Swift language. Thus, we need two different languages and frameworks to develop applications for both OS. Today, to overcome form this complexity, there are several frameworks have introduced that support both OS along with desktop apps. These types of the framework are known as **cross-platform** development tools.

The cross-platform development framework has the ability to write one code and can deploy on the various platform (Android, iOS, and Desktop). It saves a lot of time and development efforts of developers. There are several tools available for cross-platform development, including web-based tools, such as Ionic from Drifty Co. in 2013, **Phonegap** from Adobe, **Xamarin** from Microsoft, and **React-Native** form Facebook. Each of these frameworks has varying degrees of success in the mobile industry. In recent, a new framework has introduced in the cross-platform development family named **Flutter** developed from Google.

Flutter is a UI toolkit for creating fast, beautiful, natively compiled applications for mobile, web, and desktop with one programing language and single codebase. It is free and open-source. It was initially developed from **Google** and now manages by an **ECMA** standard. Flutter apps use Dart programming language for creating an app. The **dart programming** shares

several same features as other programming languages, such as Kotlin and Swift, and can be trans-compiled into JavaScript code.

Flutter is mainly optimized for 2D mobile apps that can run on both Android and iOS platforms. We can also use it to build full-featured apps, including camera, storage, geolocation, network, third-party SDKs, and more.
What makes Flutter unique?

Flutter is different from other frameworks because it neither uses **WebView** nor the **OEM** widgets that shipped with the device. Instead, it uses its own high-performance rendering engine to draw widgets. It also implements most of its systems such as animation, gesture, and widgets in Dart programing language that allows developers to read, change, replace, or remove things easily. It gives excellent control to the developers over the system.

3.1.1 Installation and Architecture of Flutter

In this section, we are going to learn how to set up an environment for the successful development of the Flutter application.
System requirements for Windows

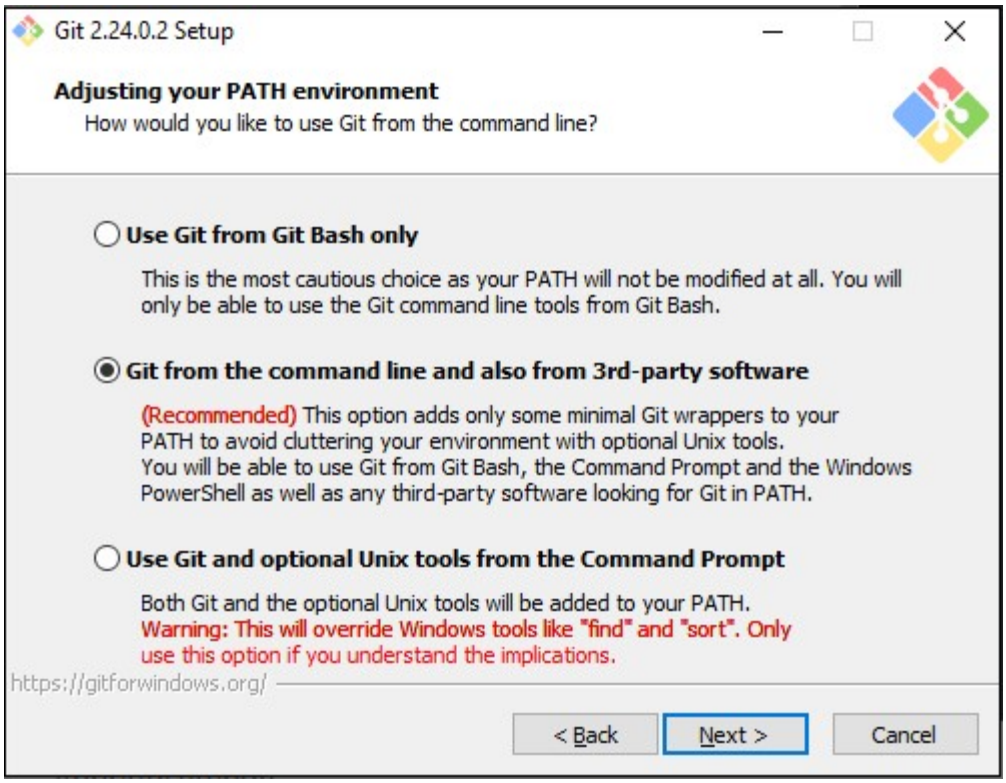
To install and run Flutter on the Windows system, you need first to meet these requirements for your development environment.

Operating System	Windows 7 or Later (I am Windows 10. You can also use Mac or Linux OS.).
Disk Space	400 MB (It does not include disk space for IDE/tools).
Tools	1.Windows PowerShell 2. Git for Windows 2.x (Here, Use Git from Windows Command Prompt option).
SDK	Flutter SDK for Windows
IDE	Android Studio (Official)

Install Git

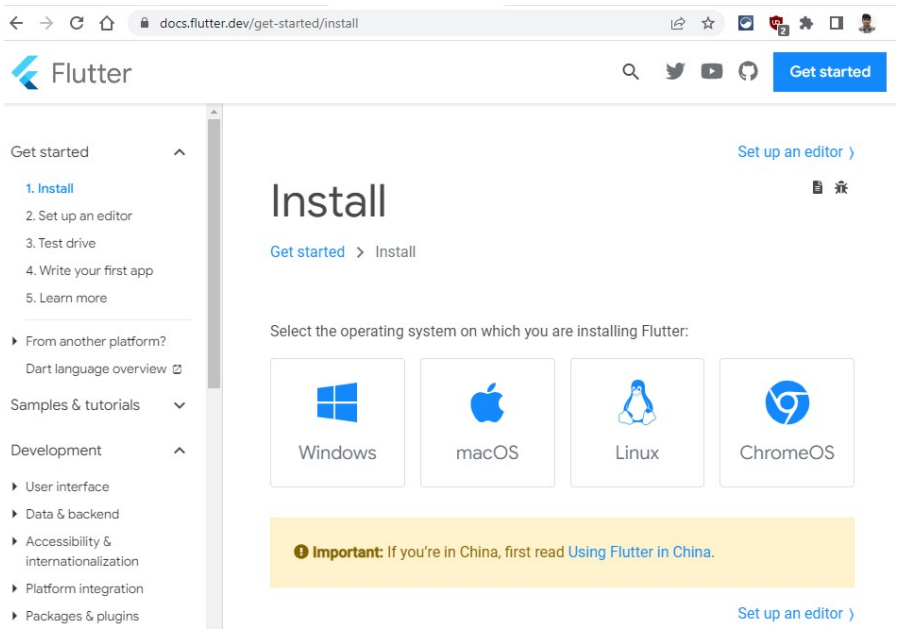
Step 1: To download Git, https://git-scm.com/download/win

Step 2: Run the **.exe** file to complete the installation. During installation, make sure that you have selected the recommended option.



Install the Flutter SDK

Step 1: Download the installation bundle of the Flutter Software Development Kit for windows. To download Flutter SDK, Go to its official https://flutter.dev/, click on **Get started** button, you will get the following screen.

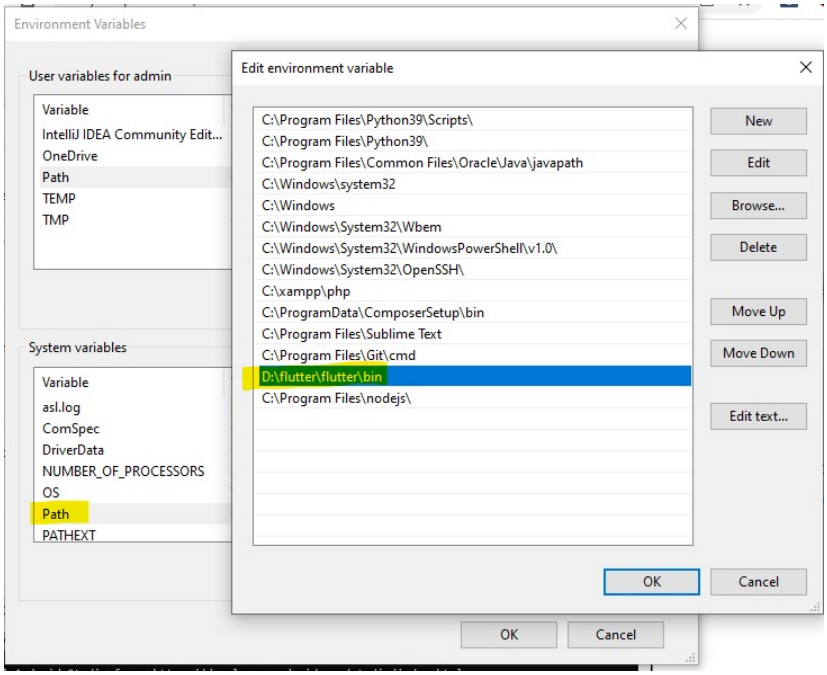


Step 2: Next, to download the latest Flutter SDK, click on the Windows **icon**. Here, you will find the download link for https://flutter.dev/docs/get-started/install/windows.

Step 3: When your download is complete, extract the **zip** file and place it in the desired installation folder or location, for example, D:/Flutter.

Step 4: To run the Flutter command in regular windows console, you need to update the system path to include the flutter bin directory. The following steps are required to do this:

Step 4.1: Go to MyComputer properties -> advanced tab -> environment variables. You will get the following screen.

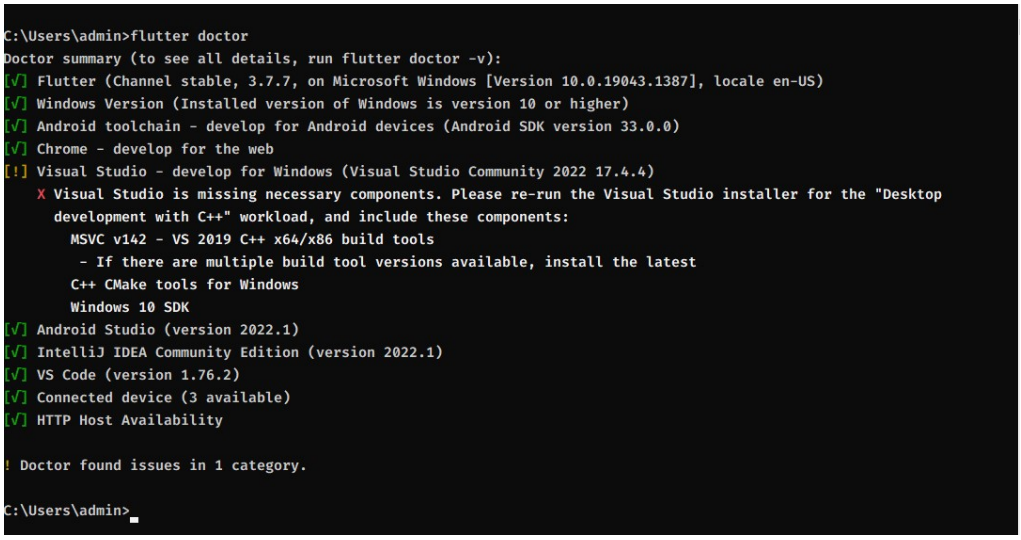


Step 4.3: In the above window, click on New->write path of Flutter bin folder in variable value -> ok -> ok -> ok.

Step 5: Now, run the $ **flutter doctor** command. This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation.

```
$ flutter doctor
```

Step 6: When you run the above command, it will analyze the system and show its report, as shown in the below image. Here, you will find the details of all missing tools, which required to run Flutter as well as the development tools that are available but not connected with the device.

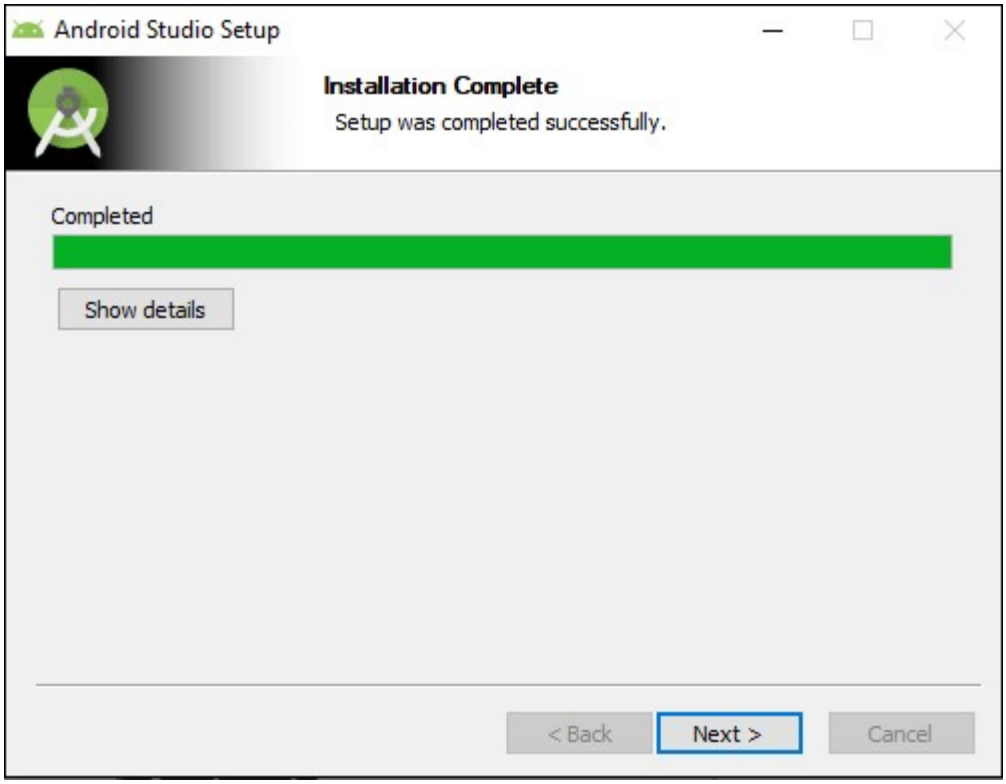


Step 7: Install the Android SDK. If the flutter doctor command does not find the Android SDK tool in your system, then you need first to install the Android Studio IDE. To install Android Studio IDE, do the following steps.

Step 7.1: Download the latest Android Studio executable or zip file from the https://developer.android.com/studio/#downloads.

Step 7.2: When the download is complete, open the **.exe** file and run it. You will get the following dialog box.

Step 7.3: Follow the steps of the installation wizard. Once the installation wizard completes, you will get the following screen.



Flutter Architecture

In this section, we are going to discuss the architecture of the Flutter framework. The Flutter architecture mainly comprises of four components.

1. Flutter Engine
2. Foundation Library
3. Widgets
4. Design Specific Widgets

Flutter Engine

It is a portable runtime for high-quality mobile apps and primarily based on the C++ language. It implements Flutter core libraries that include animation and graphics, file and network I/O, plugin architecture, accessibility support, and a dart runtime for developing, compiling, and running Flutter applications. It takes Google's open-source graphics library, **Skia**, to render low-level graphics.

Foundation Library

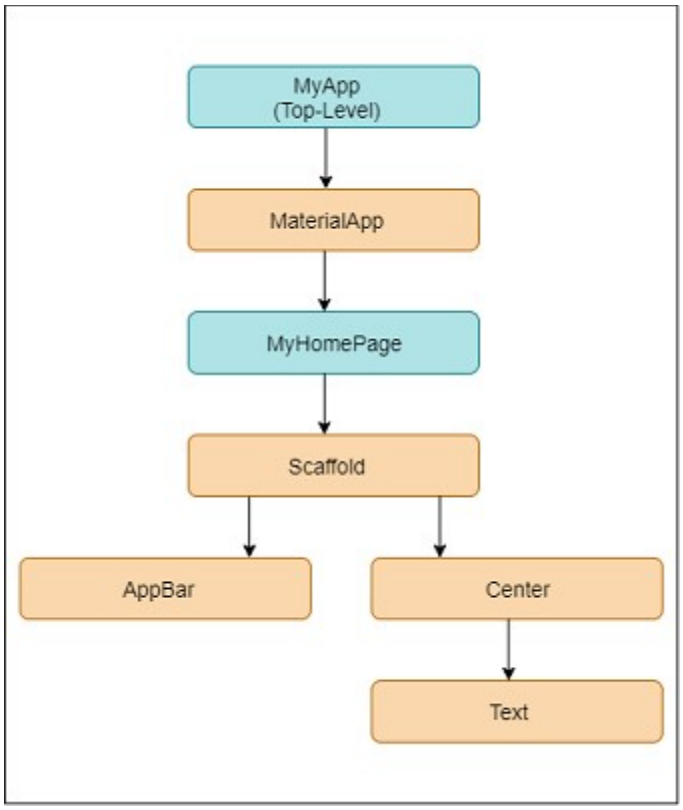
It contains all the required packages for the basic building blocks of writing a Flutter application. These libraries are written in Dart language.

Widgets

In Flutter, everything is a widget, which is the core concept of this framework. Widget in the Flutter is basically a user interface component that affects and controls the view and interface of the app. It represents an immutable description of part of the user interface and includes graphics, text, shapes, and animations that are created using widgets. The widgets are similar to the React components.

In Flutter, the application is itself a widget that contains many sub widgets. It means the app is the top-level widget, and its UI is build using one or more children widgets, which again includes sub child widgets. This feature helps you to create a complex user interface very easily.

We can understand it from the hello world example created in the previous section. Here, we are going to explain the example with the following diagram.



In the above example, we can see that all the components are widgets that contain child widgets. Thus, the Flutter application is itself a widget.

Design Specific Widgets

The Flutter framework has two sets of widgets that conform to specific design languages. These are Material Design for Android application and Cupertino Style for IOS application.

Gestures

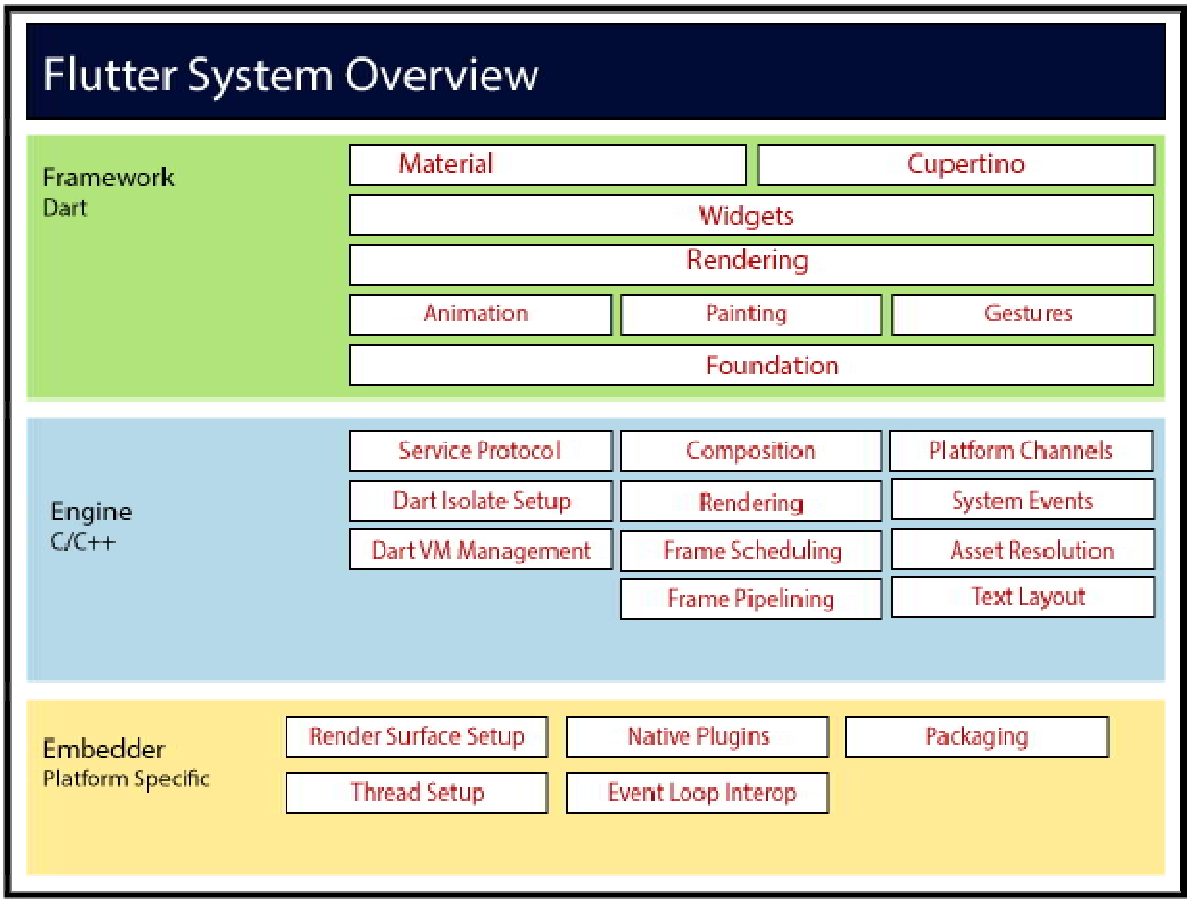
It is a widget that provides interaction (how to listen for and respond to) in Flutter using GestureDetector. **GestureDector** is an invisible widget, which includes tapping, dragging, and scaling interaction of its child widget. We can also use other interactive features into the existing widgets by composing with the GestureDetector widget.

State Management

Flutter widget maintains its state by using a special widget, StatefulWidget. It is always auto re-rendered whenever its internal state is changed. The re-rendering is optimized by calculating the distance between old and new widget UI and render only necessary things that are changes.

Layers

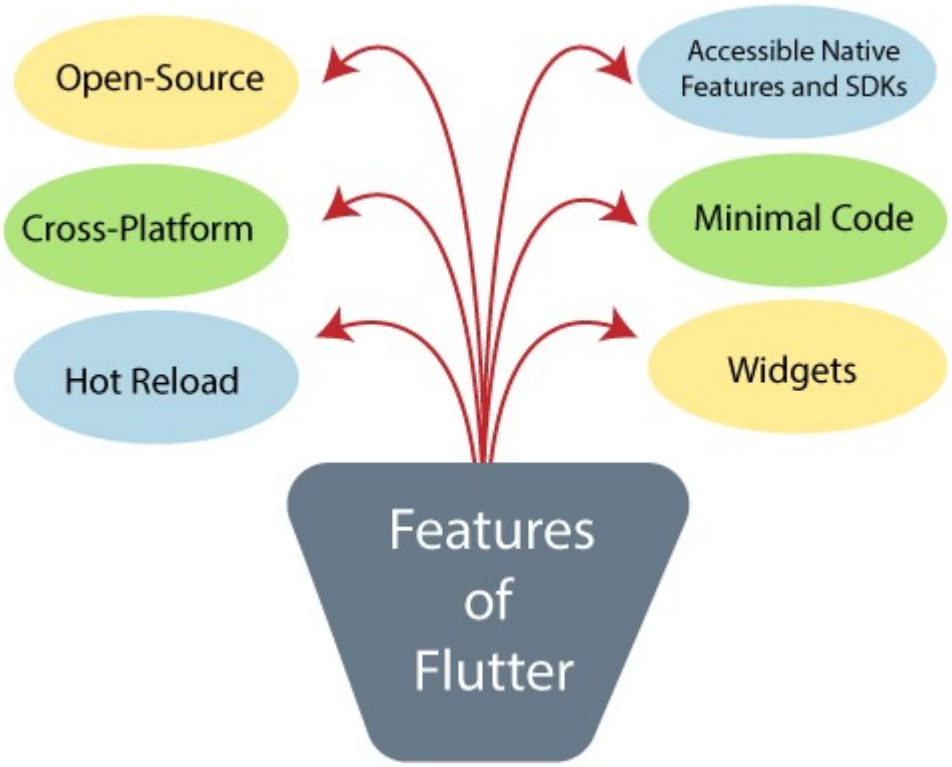
Layers are an important concept of the Flutter framework, which are grouped into multiple categories in terms of complexity and arranged in the top-down approach. The topmost layer is the UI of the application, which is specific to the Android and iOS platforms. The second topmost layer contains all the Flutter native widgets. The next layer is the rendering layer, which renders everything in the Flutter app. Then, the layers go down to Gestures, foundation library, engine, and finally, core platform-specific code. The following diagram specifies the layers in Flutter app development.



3.1.2 Features of Flutter

Features of Flutter

Flutter gives easy and simple methods to start building beautiful mobile and desktop apps with a rich set of material design and widgets. Here, we are going to discuss its main features for developing the mobile framework.



Open-Source: Flutter is a free and open-source framework for developing mobile applications.

Cross-platform: This feature allows Flutter to write the code once, maintain, and can run on different platforms. It saves the time, effort, and money of the developers.

Hot Reload: Whenever the developer makes changes in the code, then these changes can be seen instantaneously with Hot Reload. It means the changes immediately visible in the app itself. It is a very handy feature, which allows the developer to fix the bugs instantly.

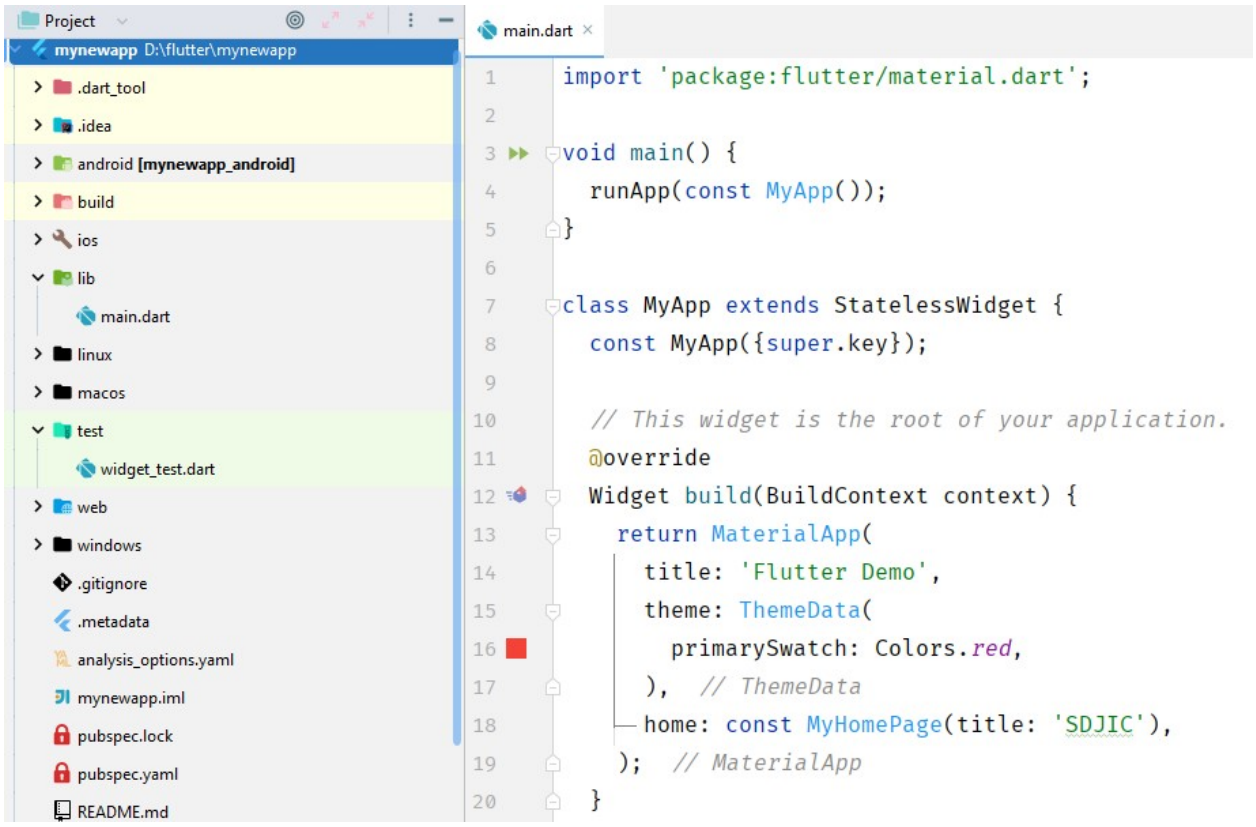
Accessible Native Features and SDKs: This feature allows the app development process easy and delightful through Flutter's native code, third-party integration, and platform APIs. Thus, we can easily access the SDKs on both platforms.

Minimal code: Flutter app is developed by Dart programming language, which uses **JIT** and **AOT** compilation to improve the overall start-up time, functioning

and accelerates the performance. JIT enhances the development system and refreshes the UI without putting extra effort into building a new one.

Widgets: The Flutter framework offers widgets, which are capable of developing customizable specific designs. Most importantly, Flutter has two sets of widgets: Material Design and Cupertino widgets that help to provide a glitch-free experience on all platforms.

3.1.3 Creating basic flutter project using Android Studio



.idea: This folder is at the very top of the project structure, which holds the configuration for Android Studio. It doesn't matter because we are not going to work with Android Studio so that the content of this folder can be ignored.

.android: This folder holds a complete Android project and used when you build the Flutter application for Android. When the Flutter code is compiled into the native code, it will get injected into this Android project, so that the result is a native Android application. **For Example:** When you are using the Android emulator, this Android project is used to build the Android app, which further deployed to the Android Virtual Device.

.ios: This folder holds a complete Mac project and used when you build the Flutter application for iOS. It is similar to the android folder that is used when developing an app for Android. When the Flutter code is compiled into the native code, it will get injected into this iOS project, so that the result is a native iOS application. Building a Flutter application for iOS is only possible when you are working on macOS.

/lib: It is an essential folder, which stands for the library. It is a folder where we will do our 99 percent of project work. Inside the lib folder, we will find the Dart files which contain the code of our Flutter application. By default, this folder contains the file **main.dart**, which is the entry file of the Flutter application.

.test: This folder contains a Dart code, which is written for the Flutter application to perform the automated test when building the app. It won't be too important for us here.

We can also have some default files in the Flutter application. In 99.99 percent of cases, we don't touch these files manually. These files are:

.gitignore: It is a text file containing a list of files, file extensions, and folders that tells Git which files should be ignored in a project. Git is a version-control file for tracking changes in source code during software development Git.

.metadata: It is an auto-generated file by the flutter tools, which is used to track the properties of the Flutter project. This file performs the internal tasks, so you do not need to edit the content manually at any time.

.packages: It is an auto-generated file by the Flutter SDK, which is used to contain a list of dependencies for your Flutter project.

flutter_demoapp.iml: It is always named according to the Flutter project's name that contains additional settings of the project. This file performs the internal tasks, which is managed by the Flutter SDK, so you do not need to edit the content manually at any time.

pubspec.yaml: It is the project's configuration file that will use a lot during working with the Flutter project. It allows you how your application works. This file contains:

- Project general settings such as name, description, and version of the project.
- Project dependencies.
- Project assets (e.g., images).

pubspec.lock: It is an auto-generated file based on the **.yaml** file. It holds more detail setup about all dependencies.

README.md: It is an auto-generated file that holds information about the project. We can edit this file if we want to share information with the developers.

Step 7: Open the **main.dart** file and replace the code with the following code snippets.

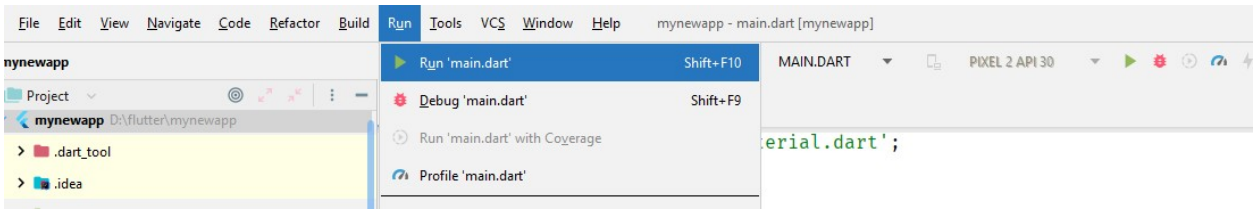
```
1.  import 'package:flutter/material.dart';
2.
3.  void main() => runApp(MyApp());
4.
5.  class MyApp extends StatelessWidget {
6.    // This widget is the root of your application.
7.    @override
8.    Widget build(BuildContext context) {
9.      return MaterialApp(
10.       title: 'Hello World Flutter Application',
11.       theme: ThemeData(
12.         // This is the theme of your application.
13.         primarySwatch: Colors.blue,
14.       ),
15.       home: MyHomePage(title: 'Home page'),
16.     );
17.   }
18. }
19. class MyHomePage extends StatelessWidget {
20.   MyHomePage({Key key, this.title}) : super(key: key);
21.   // This widget is the home page of your application.
22.   final String title;
23.
24.   @override
25.   Widget build(BuildContext context) {
26.     return Scaffold(
27.       appBar: AppBar(
28.         title: Text(this.title),
29.       ),
30.       body: Center(
```

```
31.         child: Text('Hello World'),
32.       ),
33.     );
34.   }
35. }
```

Step 8: Let us understand the above code snippet line by line.

- To start Flutter programming, you need first to import the Flutter package. Here, we have imported a **Material package**. This package allows you to create user interface according to the Material design guidelines specified by Android.
- The second line is an entry point of the Flutter applications similar to the main method in other programming languages. It calls the **runApp** function and pass it an object of **MyApp** The primary purpose of this function is to attach the given widget to the screen.
- Line 5 to 18 is a widget used for creating UI in the Flutter framework. Here, the **StatelessWidget** does not maintain any state of the widget. MyApp extends StatelessWidget that overrides its **build** The build method is used for creating a part of the UI of the application. In this block, the build method uses MaterialApp, a widget to create the root level UI of the application and contains three properties - title, theme, and home.
    1. **Title:** It is the title of the Flutter application.
    2. **Theme:** It is the theme of the widget. By default, it set the blue as the overall color of the application.
    3. **Home:** It is the inner UI of the application, which sets another widget (MyHomePage) for the application.
- Line 19 to 35, the **MyHomePage** is similar to MyApp, except it will return the **Scaffold** Scaffold widget is a top-level widget after the MaterialApp widget for creating the user interface. This widget contains two properties **appBar** and **body**. The appBar shows the header of the app, and body property shows the actual content of the application. Here, **AppBar** render the header of the application, **Center** widget is used to center the child widget, and **Text** is the final widget used to show the text content and displays in the center of the screen.

Step 9: Now, run the application. To do this, go to Run->Run main.dart, as shown in the below screen.



Step 10: Finally, you will get the output as below screen.

3.2 Flutter Widget:

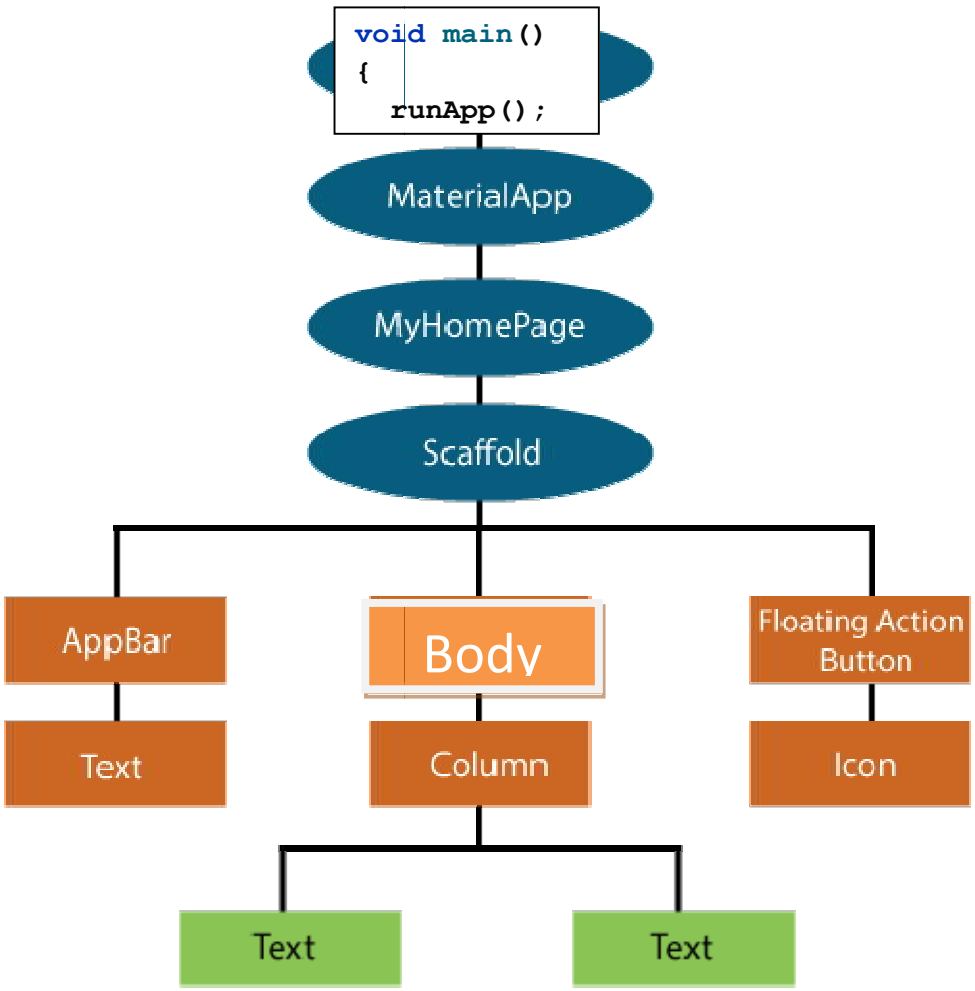
In this section, we are going to learn the concept of a widget, how to create it, and their different types available in the Flutter framework. We have learned earlier that everything in Flutter is a widget.

If you are familiar with React or Vue.js, then it is easy to understand the Flutter.

Whenever you are going to code for building anything in Flutter, it will be inside a widget. The central purpose is to build the app out of widgets. It describes how your app view should look like with their current configuration and state. When you made any alteration in the code, the widget rebuilds its description by calculating the difference of previous and current widget to determine the minimal changes for rendering in UI of the app.

Widgets are nested with each other to build the app. It means the root of your app is itself a widget, and all the way down is a widget also. For example, a widget can display something, can define design, can handle interaction, etc.

The below image is a simple visual representation of the widget tree.



We can create the Flutter widget like this:

```
Class ImageWidget extends StatelessWidget {
        // Class Stuff
}
```

Hello World Example

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

class MyHomePage extends StatelessWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  // This widget is the home page of your application.
  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(this.title),
      ),
      body: Center(
        child: Text('Hello World'),
      ),
    );
  }
}
```

3.2.1 Types of flutter widget:

3.2.1.1 Visible and Invisible

We can split the Flutter widget into two categories:

1. Visible (Output and Input)
2. Invisible (Layout and Control)

Visible widget

The visible widgets are related to the user input and output data. Some of the important types of this widget are:

Text

A Text widget holds some text to display on the screen. We can align the text widget by using **textAlign** property, and style property allow the customization of Text that includes font, font weight, font style, letter spacing, color, and many more. We can use it as like below code snippets.

```
new Text(
'Hello, Javatpoint!',
textAlign: TextAlign.center,
style: new TextStyle(fontWeight: FontWeight.bold),
)
```

Button

This widget allows you to perform some action on click. Flutter does not allow you to use the Button widget directly; instead, it uses a type of buttons like a **FlatButton** and a **RaisedButton**. We can use it as like below code snippets.

```
//FlatButton Example
new FlatButton(
  child: Text("Click here"),
  onPressed: () {
    // Do something here
  },
),

//RaisedButton Example
new RaisedButton(
  child: Text("Click here"),
  elevation: 5.0,
  onPressed: () {
    // Do something here
  },
),
```

In the above example, the **onPressed** property allows us to perform an action when you click the button, and **elevation** property is used to change how much it stands out.

Image

This widget holds the image which can fetch it from multiple sources like from the asset folder or directly from the URL. It provides many constructors for loading image, which are given below:

- **Image:** It is a generic image loader, which is used by **ImageProvider**.
- **asset:** It load image from your project asset folder.
- **file:** It loads images from the system folder.
- **memory:** It load image from memory.
- **network:** It loads images from the network.

To add an image in the project, you need first to create an assets folder where you keep your images and then add the below line in **pubspec.yaml** file.

```
assets:
  - assets/
```

Now, add the following line in the dart file.

```
Image.asset('assets/computer.png')
```

The complete source code for adding an image is shown below in the **hello world** example.

```
1.  class MyHomePage extends StatelessWidget {
2.    MyHomePage({Key key, this.title}) : super(key: key);
3.    // This widget is the home page of your application.
4.    final String title;
5.
6.    @override
7.    Widget build(BuildContext context) {
8.      return Scaffold(
9.        appBar: AppBar(
10.         title: Text(this.title),
11.       ),
12.       body: Center(
```

```
13.          child: Image.asset('assets/computer.png'),
14.        ),
15.      );
16.    }
17. }
```

When you run the app, it will give the following output.

Icon

This widget acts as a container for storing the Icon in the Flutter. The following code explains it more clearly.

```
1. new Icon(
```

```
  Icons.add,
  size: 34.0,
)
```

Invisible widget

The invisible widgets are related to the layout and control of widgets. It provides controlling how the widgets actually behave and how they will look onto the screen. Some of the important types of these widgets are:

Column

A column widget is a type of widget that arranges all its children's widgets in a vertical alignment. It provides spacing between the widgets by using the **mainAxisAlignment** and **crossAxisAlignment** properties. In these properties, the main axis is the vertical axis, and the cross axis is the horizontal axis.

Example

The below code snippets construct two widget elements vertically.

```
new Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    new Text(
      "VegElement",
    ),
    new Text(
      "Non-vegElement"
    ),
  ],
),
```

Row

The row widget is similar to the column widget, but it constructs a widget horizontally rather than vertically. Here, the main axis is the horizontal axis, and the cross axis is the vertical axis.

Example

The below code snippets construct two widget elements horizontally.

```
new Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: <Widget>[
    new Text(
      "VegElement",
    ),
    new Text(
      "Non-vegElement"
    ),
  ],
),
```

Center

This widget is used to center the child widget, which comes inside it. All the previous examples contain inside the center widget.

Example

```
Center(
  child: new clumn(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: <Widget>[
      new Text(
        "VegElement",
      ),
      new Text(
        "Non-vegElement"
      ),
```

```
    ],
  ),
),
```

Padding

This widget wraps other widgets to give them padding in specified directions. You can also provide padding in all directions. We can understand it from the below example that gives the text widget padding of 6.0 in all directions.

Example

```
Padding(
  padding: const EdgeInsets.all(6.0),
  child: new Text(
    "Element 1",
  ),
),
```

Scaffold

This widget provides a framework that allows you to add common material design elements like AppBar, Floating Action Buttons, Drawers, etc.

Stack

It is an essential widget, which is mainly used for **overlapping** a widget, such as a button on a background gradient.

3.2.1.2 StatelessWidget, StatefulWidget

State Management Widget

In Flutter, there are mainly two types of widget:

- StatelessWidget
- StatefulWidget

StatefulWidget

A StatefulWidget has state information. It contains mainly two classes: the **state object** and the **widget**. It is dynamic because it can change the inner data during the widget lifetime. This widget does not have a **build()** method. It has **createState()** method, which returns a class that extends the Flutters State Class. The examples of the StatefulWidget are Checkbox, Radio, Slider, InkWell, Form, and TextField.

Example

```
1.  class Car extends StatefulWidget {
2.    const Car({ Key key, this.title }) : super(key: key);
3.
4.    @override
5.    _CarState createState() => _CarState();
6.  }
7.
8.  class _CarState extends State<Car> {
9.    @override
10.   Widget build(BuildContext context) {
11.     return Container(
12.       color: const Color(0xFEEFE),
13.           child: Container(
14.            child: Container( //child: Container() )
15.         )
16.     );
17.   }
```

```
18. }
```

StatelessWidget

The StatelessWidget does not have any state information. It remains static throughout its lifecycle. The examples of the StatelessWidget are Text, Row, Column, Container, etc.

Example

```
1.  class MyStatelessCarWidget extends StatelessWidget {
2.    const MyStatelessCarWidget ({ Key key }) : super(key: key);
3.
4.    @override
5.    Widget build(BuildContext context) {
6.      return Container(color: const Color(0x0xFEEFE));
7.    }
8.  }
```

3.2.1.3 Single child widget and Multiple child widget

We can categories the layout widget into two types:

1. Single Child Widget
2. Multiple Child Widget

1. Single Child Widgets

These widgets can have only **one** direct child. They are useful for applying styling, alignment, or constraints to a single widget.

Examples of Single Child Widgets:

- **Container**
- **Padding**
- **Align**
- **Center**
- **Expanded**
- **FittedBox**
- **AspectRatio**

Example of a Single Child Widget (Container). **Container** is a single child widget that contains a **Text** widget.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text('Single Child Widget Example')),
        body: Center(
          child: Container(
            width: 200,
            height: 100,
            color: Colors.blue,
            child: Center(
              child: Text(
                'Hello Flutter!',
                style: TextStyle(color: Colors.white, fontSize: 18),
              ),
            ),
          ),
        ),
      ),
    );
  }
}
```

2. Multi Child Widgets

These widgets can have **multiple** child widgets. They are used for building complex UI layouts.

Examples of Multiple Child Widgets:

- **Row** (horizontal layout)
- **Column** (vertical layout)
- **Stack** (overlapping layout)
- **ListView** (scrollable list)
- **GridView** (grid-based layout)
- **Wrap** (responsive wrapping layout)

Example of a Multiple Child Widget (Column)

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text('Multiple Child Widget Example')),
        body: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text('First Child', style: TextStyle(fontSize: 18)),
            Text('Second Child', style: TextStyle(fontSize: 18)),
            Text('Third Child', style: TextStyle(fontSize: 18)),
          ],
        ),
      ),
    );
  }
}
```

Key Differences Between Single Child and Multiple Child Widgets

Feature	Single Child Widget	Multiple Child Widget
Number of Children	Only 1 child	Multiple children
Common Examples	Container, Center, Padding	Row, Column, Stack, ListView
Usage	For styling, spacing, constraints	For layouts and UI structures
Performance	More efficient	May affect performance with too many children

- **Use Single Child Widgets** when you need to wrap and modify a single widget.
- **Use Multiple Child Widgets** when arranging multiple widgets in a structured layout.

Unit 4 : Flutter Basic Widgets

4.1 Visible widget(Constructor and Properties):
Text, Image, Button, Icon

Flutter provides several visible widgets that are essential for UI development. Here, we'll cover four important visible widgets:

1 Text
2 Image
3 Button
4 Icon

Each widget has its constructor and properties to customize its appearance and behavior.

1. **Text Widget**

The Text widget displays text with styling options.

Constructor:

```
Text(
  String data,
  {Key? key,
  TextStyle? style,
  TextAlign? textAlign,
  int? maxLines,
  TextOverflow? overflow,
  TextDirection? textDirection}
)
```

Example

```
Text(
  'Hello, Flutter!',
  style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold, color: Colors.blue),
  textAlign: TextAlign.center,
)
```

Common Properties:

Property	Description
style	Defines text style (color, size, weight, etc.)
textAlign	Aligns text (left, right, center, justify)
maxLines	Limits number of text lines
overflow	Handles text overflow (ellipsis, fade, etc.)

2. Image Widget.

The Image widget is used to display images from assets, network, or memory.

Constructor:

```
Image.asset(String name, {Key? key, double? width, double? height, BoxFit? fit})
Image.network(String url, {Key? key, double? width, double? height, BoxFit? fit})
```

Example

```
Image.network('https://www.sdjic.org/storage/pages/tajxmzy5a4n49kcoltl6mzu4kwz7s9-metac2xpzgvyms5qcgc=-.jpg',
  width: 200,
  height: 100,
  fit: BoxFit.cover,
)
```

Common Properties:

Property	Description
width	Sets image width
height	Sets image height

fit	Defines how the image should fit (cover, contain, fill)
alignment	Aligns image within its container

3. Button Widget

Flutter provides several button widgets like **ElevatedButton**, **TextButton**, and **OutlinedButton**.

```
ElevatedButton(
  {Key? key, required VoidCallback onPressed, required Widget child}
)
```

```
ElevatedButton(
  onPressed: () {
    print('Button Pressed!');
  },
  child: Text('Click Me'),
)
```

Common Properties:

Property	**Description**
onPressed	Function triggered when button is clicked
child	Widget inside the button (Text, Icon, etc.)
style	Customizes button (background color, padding, etc.)

4. Icon Widget

The Icon widget displays an icon from Material Icons.

Constructor:

```
Icon(IconData icon, {Key? key, double? size, Color? color})
```

Example

```
Icon(
  Icons.star,
  size: 50,
  color: Colors.orange,
)
```

Common Properties:

Property	Description
size	Sets icon size
color	Defines icon color
semanticLabel	Adds accessibility label

4.2 Invisible widget(Constructor and Properties): column, row, center, padding, scaffold, stack

Invisible widgets in Flutter do not display content themselves but are used to structure and arrange visible widgets. These include:

1 **Column** – Arranges widgets vertically
2 **Row** – Arranges widgets horizontally
3 **Center** – Centers a widget
4 **Padding** – Adds spacing around a widget
5 **Scaffold** – Provides a basic app structure
6 **Stack** – Overlaps widgets on top of each other

---

1. Column Widget

The Column widget arranges its children vertically.

Constructor

```
Column({Key? key,
  MainAxisAlignment mainAxisAlignment = MainAxisAlignment.start,
  CrossAxisAlignment crossAxisAlignment = CrossAxisAlignment.center,
  List<Widget> children = const <Widget>[]}
)

```

Example

```
Column(
  mainAxisAlignment: MainAxisAlignment.center,
  crossAxisAlignment: CrossAxisAlignment.start,
  children: [
    Text("First Item"),
    Text("Second Item"),
  ],
)
```

Common Properties:

Property	Description
mainAxisAlignment	Aligns widgets along the vertical axis
crossAxisAlignment	Aligns widgets along the horizontal axis
children	Contains multiple child widgets

2. Row Widget

The Row widget arranges its children horizontally.

Constructor:

```
Row(
  {Key? key,
  MainAxisAlignment mainAxisAlignment = MainAxisAlignment.start,
  CrossAxisAlignment crossAxisAlignment = CrossAxisAlignment.center,
  List<Widget> children = const <Widget>[]}
)
```

Example

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceBetween,
  children: [
    Icon(Icons.star, color: Colors.blue),
    Icon(Icons.favorite, color: Colors.red),
  ],
)
```

Common Properties:

Property	Description
mainAxisAlignment	Aligns widgets along the horizontal axis
crossAxisAlignment	Aligns widgets along the vertical axis
children	Contains multiple child widgets

3. Center Widget

The Center widget centers a child widget inside its parent.

Constructor

```
Center(
  {Key? key, Widget? child}
)
```

Example

```
Center(
  child: Text("Hello, Flutter!"),
)
```

Common Properties:

Property	Description
child	The widget to be centered

4. Padding Widget

The Padding widget adds space around a child widget.

Constructor:

```
Padding(
  {Key? key, required EdgeInsetsGeometry padding, Widget? child}
)
```

Example:

```
Padding(
  padding: EdgeInsets.all(16.0),
  child: Text("Padded Text"),
)
```

Common Properties:

Property	Description
padding	Defines spacing around the child widget
child	The widget inside the padding

5. Scaffold Widget

The Scaffold widget provides a basic structure for a Flutter app, including an AppBar, Body, Floating Button, etc.

Constructor:

```
Scaffold(
  {Key? key,
  PreferredSizeWidget? appBar,
  Widget? body,
  Widget? floatingActionButton}
)

```

Example:

```
Scaffold(
  appBar: AppBar(title: Text("Scaffold Example")),
  body: Center(child: Text("Hello, Flutter!")),
  floatingActionButton: FloatingActionButton(
    onPressed: () {},
    child: Icon(Icons.add),
  ),
)
```

6. Stack Widget

The Stack widget overlaps widgets on top of each other.

Constructor:

```
Stack(
  {Key? key, AlignmentGeometry alignment = AlignmentDirectional.topStart,
  StackFit fit = StackFit.loose, List<Widget> children = const <Widget>[]}
)
```

Example:

```dart
Stack(
  alignment: Alignment.center,
  children: [
    Container(width: 200, height: 200, color: Colors.blue),
    Text("Stacked Text", style: TextStyle(color: Colors.white)),
  ],
)
```

Common Properties:

Property	**Description**
alignment	Defines how children are aligned inside the stack
children	List of widgets stacked on top of each other

4.3 Text and TextField

Flutter provides two primary widgets for displaying and collecting text input:

1. Text Widget – Used for displaying static text
2. TextField Widget – Used for accepting user input

1. Text Widget

The Text widget is used to display static text **in Flutter apps.**

Constructor:

```
Text(
  String data, {
  Key? key,
  TextStyle? style,
  TextAlign? textAlign,
  TextDirection? textDirection,
  bool softWrap = true,
  TextOverflow overflow = TextOverflow.clip,
  double? textScaleFactor,
  int? maxLines,
})
```

Example:

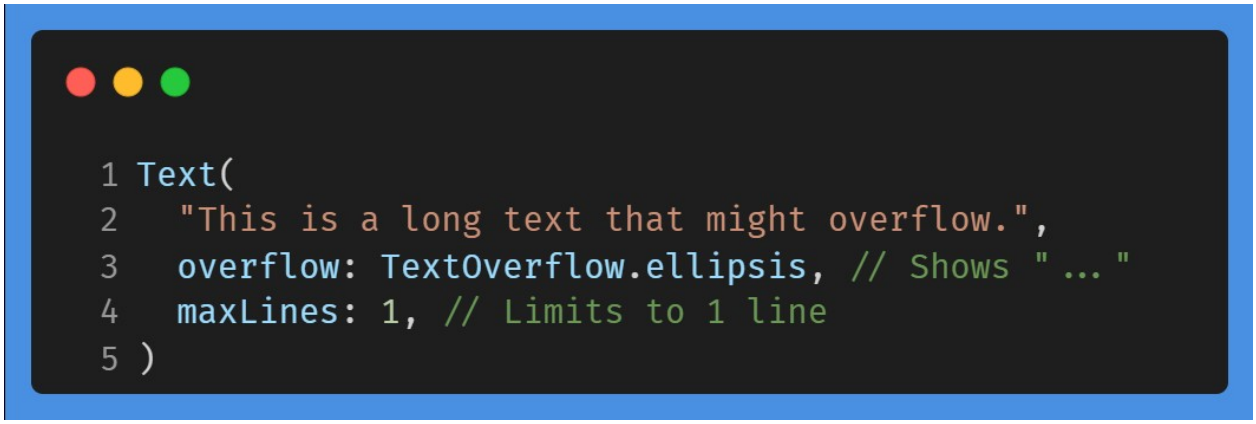
```
Text(
  "Hello, Flutter!",
  style: TextStyle(
    fontSize: 24,
    fontWeight: FontWeight.bold,
    color: Colors.blue,
  ),
  textAlign: TextAlign.center,
)
```

Common Properties of Text Widget:

Property	**Description**
style	Defines text styling (font size, color, weight)
textAlign	Aligns text (left, right, center, justify)
textDirection	Sets text direction (LTR or RTL)

softWrap	Determines if text wraps or not
overflow	Handles text overflow (ellipsis, fade, clip)
maxLines	Limits the number of lines for text

Text Overflow Example:



```dart
Text(
  "This is a long text that might overflow.",
  overflow: TextOverflow.ellipsis, // Shows "..."
  maxLines: 1, // Limits to 1 line
)
```

1. **TextField Widget**

The TextField widget is used for taking user input in Flutter applications.

Constructor:

```dart
TextField({
  Key? key,
  TextEditingController? controller,
  FocusNode? focusNode,
  InputDecoration? decoration,
  TextInputType? keyboardType,
  TextInputAction? textInputAction,
  bool obscureText = false,
  bool autofocus = false,
  bool readOnly = false,
  int? maxLength,
  ValueChanged<String>? onChanged,
  VoidCallback? onEditingComplete,
  ValueChanged<String>? onSubmitted,
})
```

Example:

```dart
TextField(
  decoration: InputDecoration(
    labelText: "Enter your name",
    hintText: "Nehal Patel",
    border: OutlineInputBorder(),
  ),
)
```

Common Properties of TextField Widget:

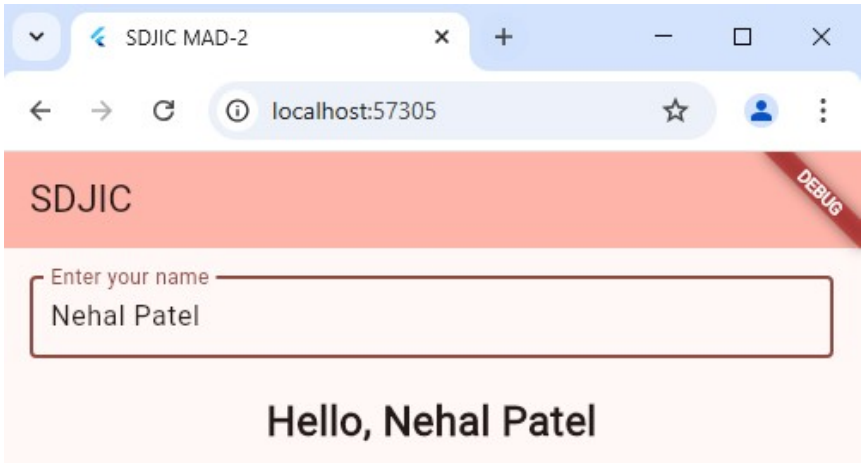
Property	Description
controller	Controls the text input
decoration	Adds styling and labels to the input field
keyboardType	Defines the keyboard type (text, number, email)
obscureText	Hides text input (useful for passwords)
readOnly	Makes the field read-only
autofocus	Automatically focuses the field when the screen opens
maxLength	Limits the number of characters
onChanged	Listens for input changes
onSubmitted	Called when the user submits input

Example with Controller:

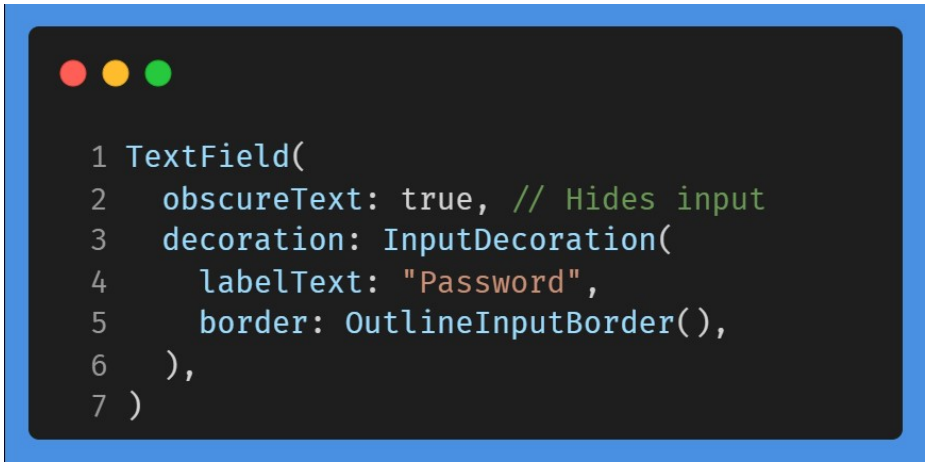
```
TextEditingController _controller = TextEditingController();

TextField(
  controller: _controller,
  decoration: InputDecoration(labelText: "Enter text"),
);
```

Example:

```dart
class _MyHomePageState extends State<MyHomePage> {
  final TextEditingController _controller = TextEditingController();
  String displayText = "";

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(widget.title),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: [
            TextField(
              controller: _controller,
              decoration: InputDecoration(
                labelText: "Enter your name",
                border: OutlineInputBorder(),
              ),
              onChanged: (value) {
                setState(() {
                  displayText = value;
                });
              },
            ),
            SizedBox(height: 20),
            Text(
              "Hello, $displayText",
              style: TextStyle(fontSize: 24, fontWeight:
FontWeight.bold),
            ),
          ],
        ),
      ),
    );
  }
}
```



Password Input Example:



```dart
TextField(
  obscureText: true, // Hides input
  decoration: InputDecoration(
    labelText: "Password",
    border: OutlineInputBorder(),
  ),
)
```

Key Differences:

Feature	Text Widget	TextField Widget
Purpose	Displays static text	Accepts user input
Editable	❌No	✅Yes
Controller	❌No	✅Yes (`TextEditingController`)
Focus	❌No	✅Yes (`FocusNode`)
Keyboard Type	❌No	✅Yes (`keyboardType`)
Styling	✅Yes (`TextStyle`)	✅Yes (`InputDecoration`)

4.4 Buttons and Slider

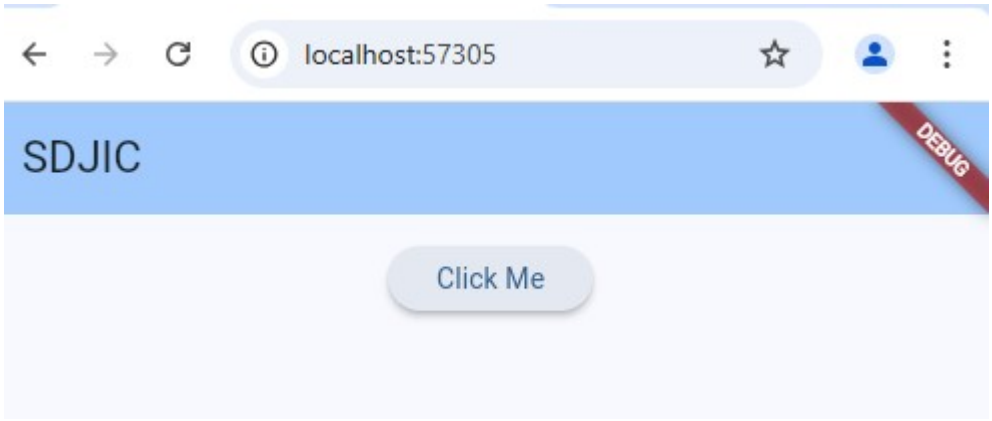
Flutter provides multiple button widgets:

- **ElevatedButton** (Raised button with elevation)
- **TextButton** (Flat button with no elevation)
- **OutlinedButton** (Bordered button)
- **IconButton** (Clickable icon)
- **FloatingActionButton** (FAB) (Used for primary actions)

1.1 ElevatedButton

A button with elevation and a filled background.

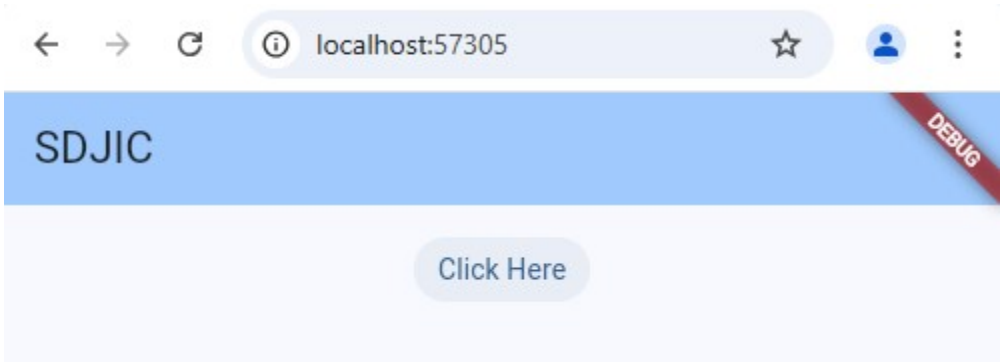
```dart
ElevatedButton(
  onPressed: () {
    print("ElevatedButton Clicked!");
  },
  child: Text("Click Me"),
)
```



1.2 TextButton

A flat button without elevation.

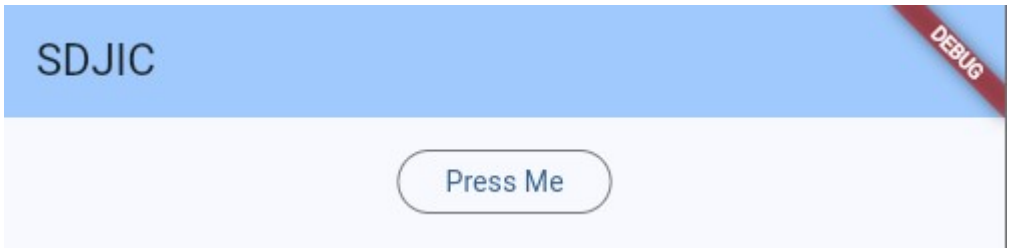
```dart
TextButton(
  onPressed: () {
    print("TextButton Clicked!");
  },
  child: Text("Click Here"),
)

```



1.3 OutlinedButton

A button with a border and transparent background.

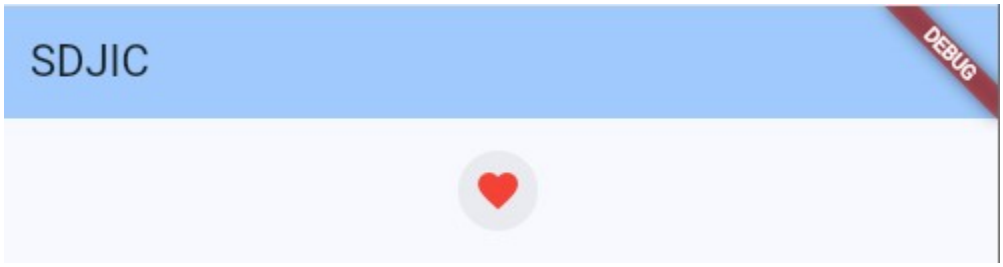
```dart
OutlinedButton(
  onPressed: () {
    print("OutlinedButton Clicked!");
  },
  child: Text("Press Me"),
)
```



1.4 IconButton

A button with an icon instead of text.

```dart
IconButton(
  onPressed: () {
    print("IconButton Clicked!");
  },
  icon: Icon(Icons.favorite, color: Colors.red),
)
```



1.5 FloatingActionButton (FAB)

A floating circular button, commonly used in Scaffold.

```dart
FloatingActionButton(
  onPressed: () {
    print("FAB Clicked!");
  },
  child: Icon(Icons.add),
)
```

Sliders

The Slider widget lets users select a value from a range.

Slider Constructor

```
Slider(
  value: currentValue,
  min: 0,
  max: 100,
  divisions: 10,
  label: currentValue.round().toString(),
  onChanged: (double value) {
    setState(() {
      currentValue = value;
    });
  },
)
```

```dart
Text("Slider Value: ${sliderValue.round()}"),
Slider(
  value: sliderValue,
  min: 0,
  max: 100,
  divisions: 10,
  label: sliderValue.round().toString(),
  onChanged: (double value) {
    setState(() {
      sliderValue = value;
    });
  },
),
```

Summary Table

Widget	**Description**
ElevatedButton	Button with elevation & background color
TextButton	Flat button (no elevation)
OutlinedButton	Button with a border

IconButton	Button with an icon
FloatingActionButton	Circular floating button
Slider	Selects a value within a range

4.5 Checkbox & RadioButton

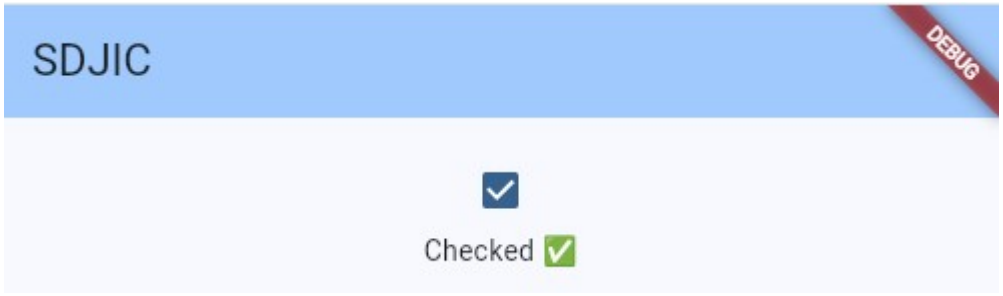
A Checkbox allows multiple selections. It has two states:

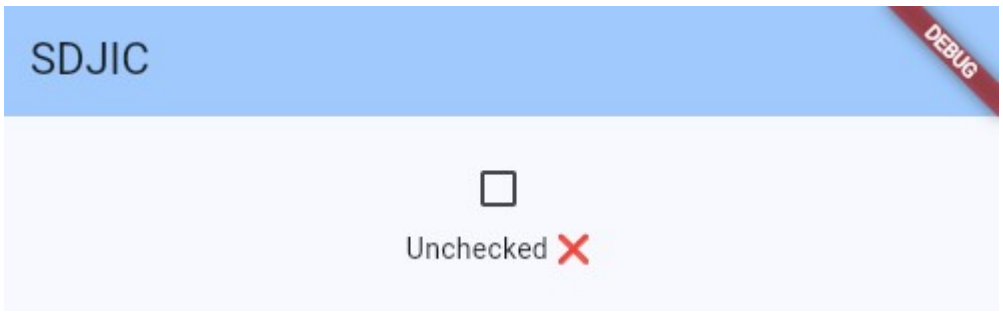
☑ Checked (true)

✕ Unchecked (false)

Checkbox Constructor

```
Checkbox(
  value: isChecked,
  onChanged: (bool? newValue) {
    setState(() {
      isChecked = newValue!;
    });
  },
)
```





RadioButton Widget in Flutter

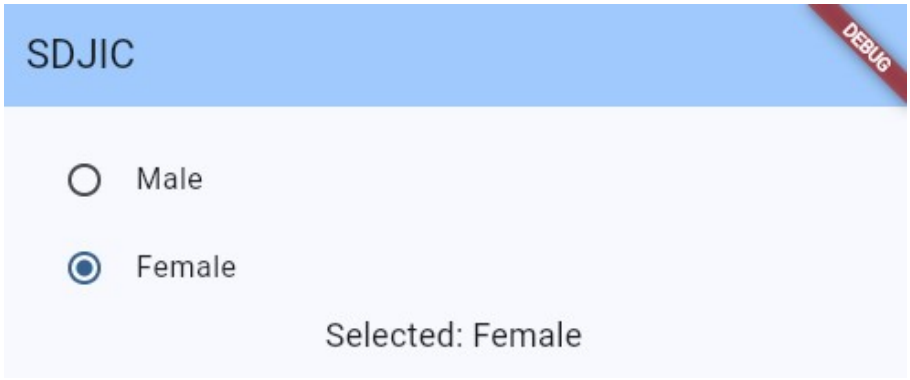
A RadioButton allows a single selection from multiple options.

Constructor:

```dart
Radio(
  value: "Male",
  groupValue: selectedGender,
  onChanged: (String? newValue) {
    setState(() {
      selectedGender = newValue!;
    });
  },
)
```

Example

```dart
ListTile(
  title: Text("Male"),
  leading: Radio(
    value: "Male",
    groupValue: selectedGender,
    onChanged: (String? newValue) {
      setState(() {
        selectedGender = newValue!;
      });
    },
  ),
),
ListTile(
  title: Text("Female"),
  leading: Radio(
    value: "Female",
    groupValue: selectedGender,
    onChanged: (String? newValue) {
      setState(() {
        selectedGender = newValue!;
      });
    },
  ),
),
Text("Selected: $selectedGender", style: TextStyle(fontSize: 18)),
```



Summary Table

Widget	**Description**
Checkbox	Allows multiple selections ☑
RadioButton	Allows a single selection ◻

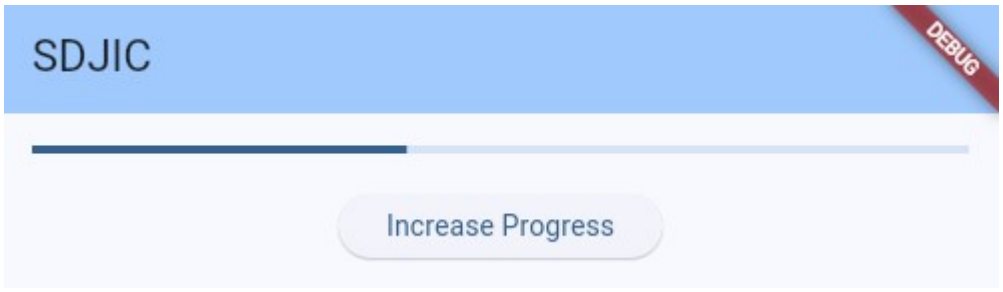
Unit-5: Flutter basic widget (Constructor, attributes and Properties)

5.1 Progress Bar, Stack

1. Progress Bar (LinearProgressIndicator & CircularProgressIndicator)

Flutter provides **LinearProgressIndicator** and **CircularProgressIndicator** for showing progress.

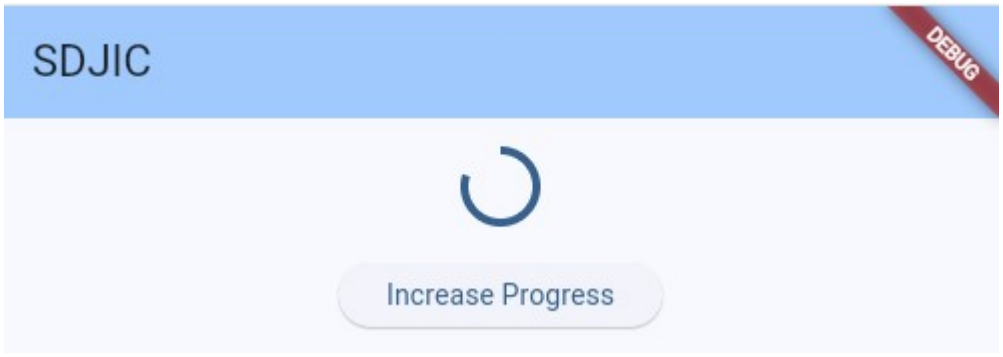
```dart
LinearProgressIndicator(value: progress),
SizedBox(height: 20),
ElevatedButton(
  onPressed:
      () => {
        setState(() {
          if (progress < 1.0) progress += 0.1;
        }),
      },
  child: Text("Increase Progress"),
),
```



2. Circular Progress Bar (Indeterminate & Determinate)

CircularProgressIndicator is used to show a circular loading indicator

```dart
CircularProgressIndicator(value: progress),
SizedBox(height: 20),
ElevatedButton(
  onPressed:
      () => {
        setState(() {
          if (progress < 1.0) progress += 0.1;
        }),
      },
  child: Text("Increase Progress"),
),
```

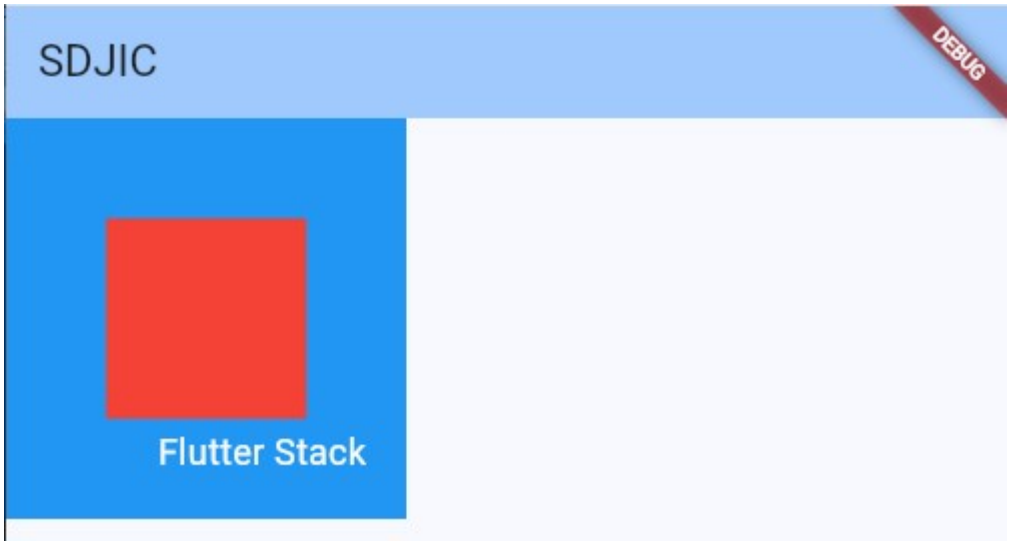


2. Stack Widget in Flutter

Stack is used to place widgets **on top of each other** (like layers).

Example:

```dart
Stack(
children: [
  Container(
    width: 200,
    height: 200,
    color: Colors.blue, // Base box
  ),
  Positioned(
    top: 50,
    left: 50,
    child: Container(
      width: 100,
      height: 100,
      color: Colors.red, // Smaller box on top
    ),
  ),
  Positioned(
    bottom: 20,
    right: 20,
    child: Text(
      "Flutter Stack",
      style: TextStyle(fontSize: 18, color: Colors.white),
    ),
  ),
],
),
```



Summary Table

Widget	**Description**
LinearProgressIndicator	A straight-line progress bar
CircularProgressIndicator	A circular loading indicator
Stack	A widget for layering elements on top of each other

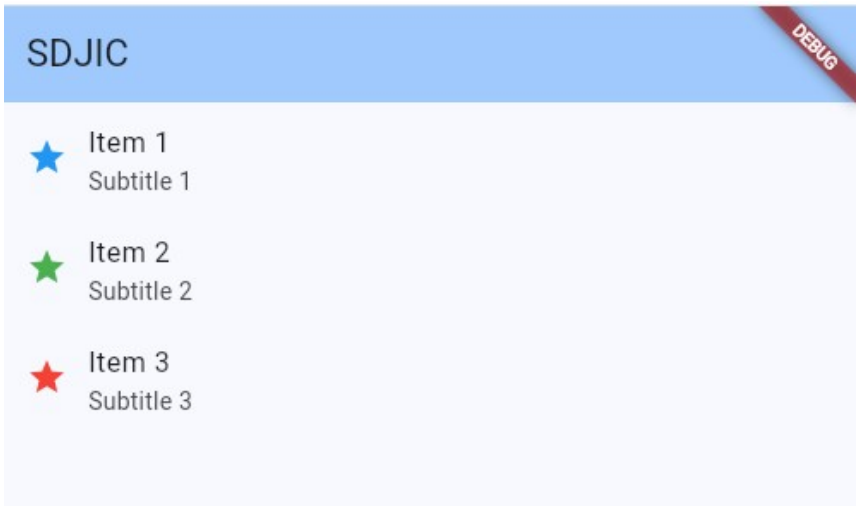
5.2 Lists

1. ListView (Static List)

ListView is a scrollable list of widgets. This is useful when we have a **small number of items**.

Example: ListView (Simple Static List)

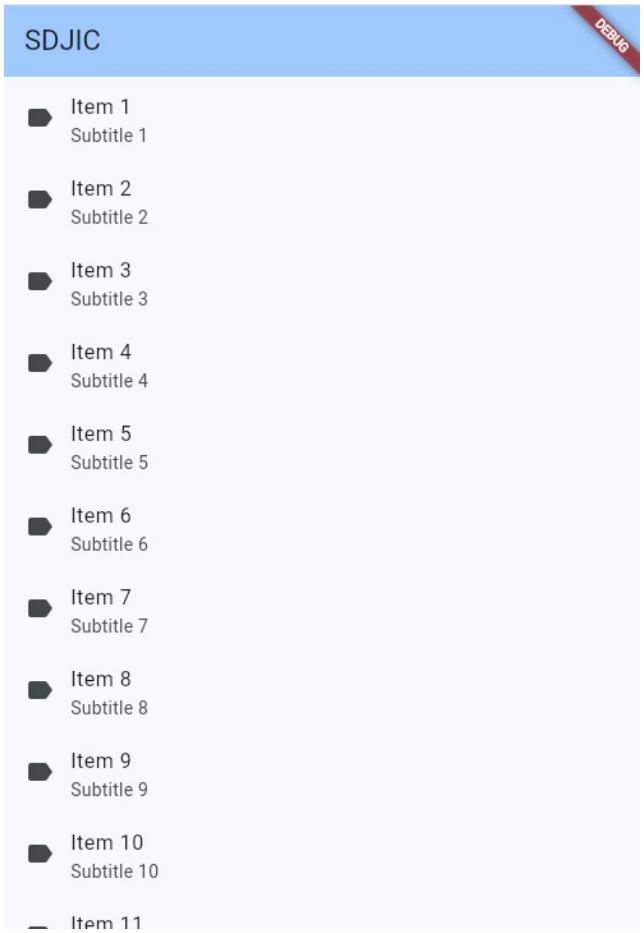
```dart
ListView(
  children: [
    ListTile(
      leading: Icon(Icons.star, color: Colors.blue),
      title: Text("Item 1"),
      subtitle: Text("Subtitle 1"),
    ),
    ListTile(
      leading: Icon(Icons.star, color: Colors.green),
      title: Text("Item 2"),
      subtitle: Text("Subtitle 2"),
    ),
    ListTile(
      leading: Icon(Icons.star, color: Colors.red),
      title: Text("Item 3"),
      subtitle: Text("Subtitle 3"),
    ),
  ],
),
```



2. ListView.builder (Dynamic List)

ListView.builder is used when the list length is unknown or dynamic. It **only loads visible items**, improving performance.

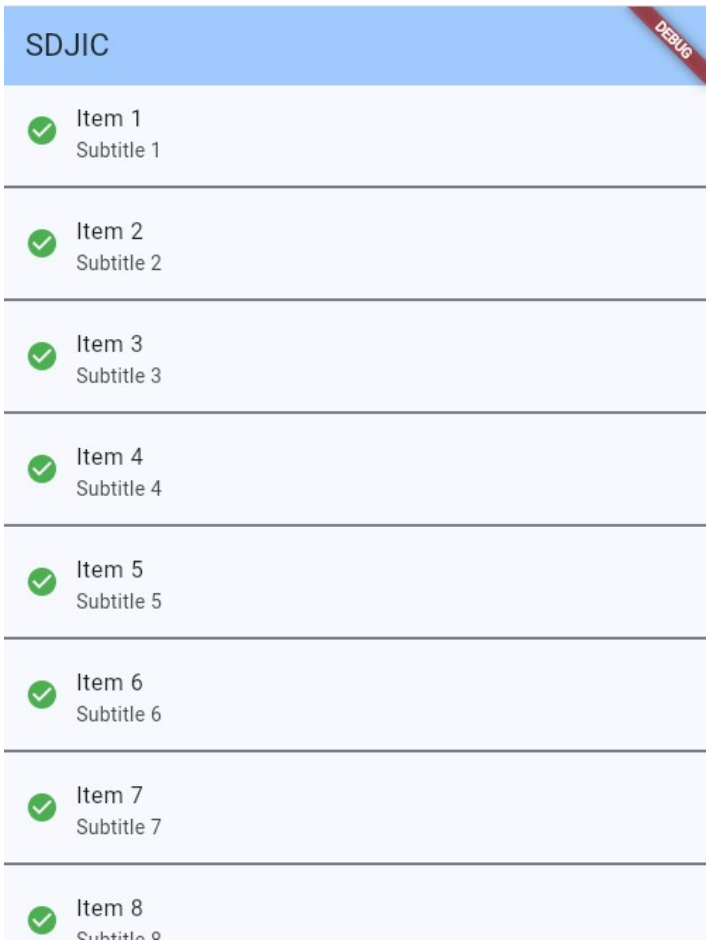
```dart
final List<String> items = List.generate(20, (index) => "Item ${index + 1}");

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text("Dynamic ListView Example")),
    body: ListView.builder(
      itemCount: items.length,
      itemBuilder: (context, index) {
        return ListTile(
          leading: Icon(Icons.label),
          title: Text(items[index]),
          subtitle: Text("Subtitle ${index + 1}"),
        );
      },
    ),
  );
}
```



3. ListView.separated (Custom Dividers)

ListView.separated is used when we want custom dividers between list items.

```dart
final List<String> items = List.generate(10, (index) => "Item ${index + 1}");

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("ListView with Separator")),
      body: ListView.separated(
        itemCount: items.length,
        separatorBuilder: (context, index) => Divider(color: Colors.black),
        itemBuilder: (context, index) {
          return ListTile(
            leading: Icon(Icons.check_circle, color: Colors.green),
            title: Text(items[index]),
            subtitle: Text("Subtitle ${index + 1}"),
          );
        },
      ),
    );
  }
```



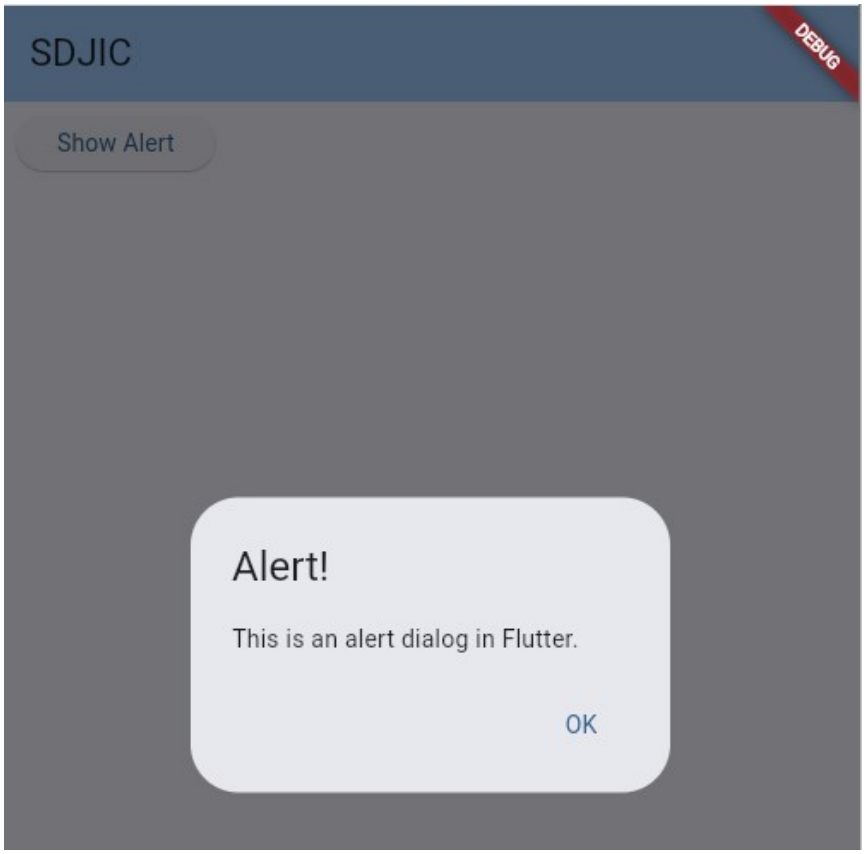
5.3 AlertDialog, Tooltip

In Flutter, **AlertDialog** and **Tooltip** are used to enhance user interaction.

- **AlertDialog**: Used to show pop-up messages, confirmations, or prompts.
- **Tooltip**: Displays a small message when a user hovers over or long-presses a widget.

1. AlertDialog

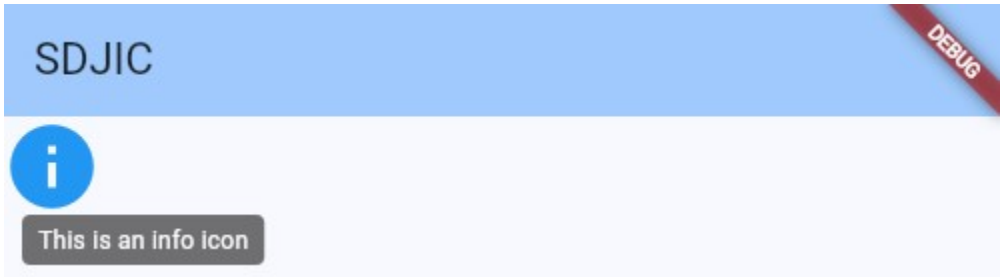
```dart
class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key, required this.title});

  final String title;

  void showMyDialog(BuildContext context) {
    showDialog(
      context: context,
      builder: (context) {
        return AlertDialog(
          title: Text("Alert!"),
          content: Text("This is an alert dialog in Flutter."),
          actions: [
            TextButton(
              onPressed: () {
                Navigator.of(context).pop(); // Closes dialog
              },
              child: Text("OK"),
            ),
          ],
        );
      },
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(title),
      ),
      body: Padding(
        padding: const EdgeInsets.all(8.0),
        child: ElevatedButton(
          onPressed: () => showMyDialog(context),
          child: Text("Show Alert"),
        ),
      ),
    );
  }
}
```



3. Tooltip (Hover/Long-Press Message)

Tooltip provides additional information when a user hovers over or long-presses a widget.

```dart
Tooltip(
  message: "This is an info icon",
  child: Icon(Icons.info, size: 50, color: Colors.blue),
),
```



5.4 Toast, Switch,

Toast: Displays a temporary message at the bottom of the screen.

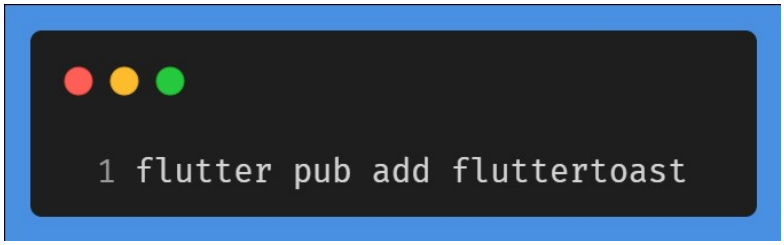
Switch: A toggle button for enabling/disabling settings.

1. Toast (Short Message Notification)

Flutter does not have a built-in Toast widget, but we can use the **fluttertoast** package.

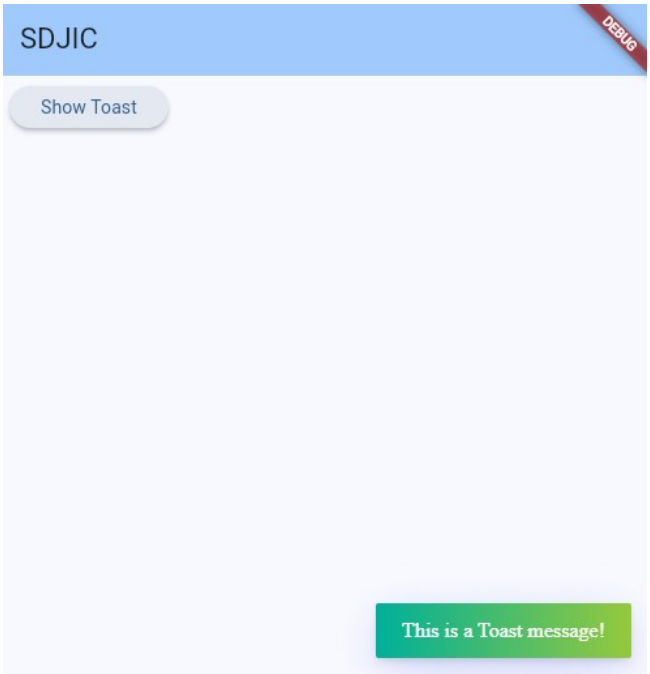
Steps to Use Toast

1. **Add Dependency**:
   Run this command in your terminal:



2. Use the Toast in Code:

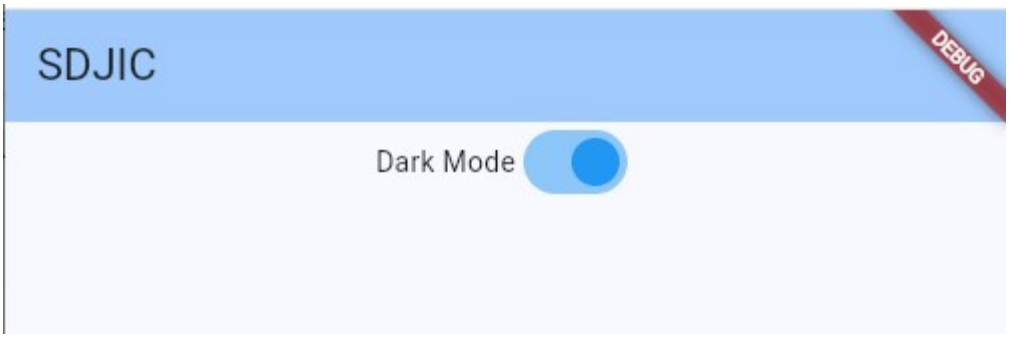
```dart
class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key, required this.title});

  final String title;

  void showToast() {
    Fluttertoast.showToast(
      msg: "This is a Toast message!",
      toastLength: Toast.LENGTH_SHORT,
      gravity: ToastGravity.BOTTOM, // Position
      backgroundColor: Colors.black,
      textColor: Colors.white,
      fontSize: 16.0,
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(title),
      ),
      body: Padding(
        padding: const EdgeInsets.all(8.0),
        child: ElevatedButton(onPressed: showToast, child: Text("Show Toast")),
      ),
    );
  }
}
```



Switch (Toggle Button)

Switch is used to turn settings ON/OFF.

```dart
class _MyHomePageState extends State<MyHomePage> {
  bool isSwitched = false;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(widget.title),
      ),
      body: Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Text("Dark Mode"),
          Switch(
            value: isSwitched,
            onChanged: (value) {
              setState(() {
                isSwitched = value;
              });
            },
            activeColor: Colors.blue,
          ),
        ],
      ),
    );
  }
}
```



5.5 Charts, Flutter Form.

Flutter: Charts & Forms

In Flutter, you can use:

- **Charts**: Visual representation of data using bar, line, or pie charts.
- **Forms**: Used for input fields like text fields, checkboxes, and validation.

---

1 Charts (Using fl_chart Library)

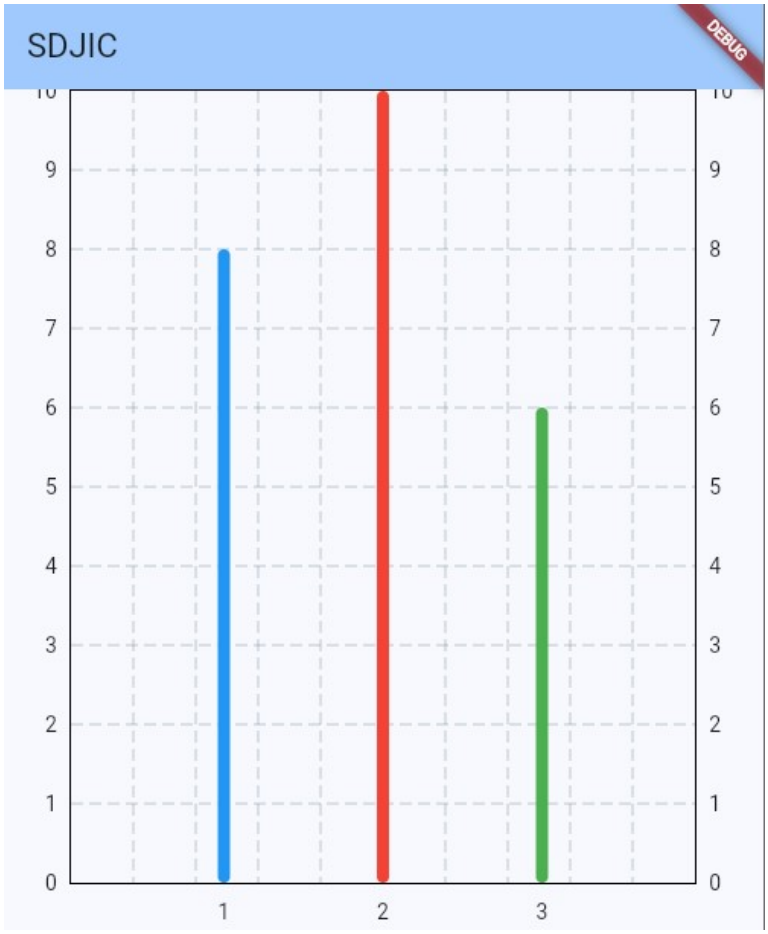
Flutter does not have built-in chart support, but we can use the **fl_chart** package.

Steps to Use Charts in Flutter

1. **Add Dependency**:
   Run this command in your terminal:

2. Use a Chart in Your Code

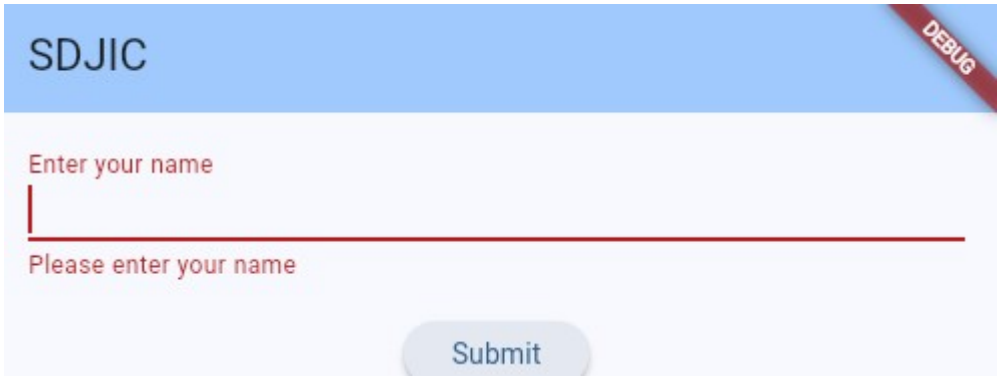
```dart
import 'package:fl_chart/fl_chart.dart';
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'SDJIC MAD-2',
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.blue),
        useMaterial3: true,
      ),
      home: const MyHomePage(title: 'SDJIC'),
    );
  }
}

class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key, required this.title});

  final String title;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(title),
      ),
      body: BarChart(
        BarChartData(
          barGroups: [
            BarChartGroupData(
             x: 1,
              barRods: [BarChartRodData(toY: 8, color: Colors.blue)],
            ),
            BarChartGroupData(
              x: 2,
              barRods: [BarChartRodData(toY: 10, color: Colors.red)],
            ),
            BarChartGroupData(
              x: 3,
              barRods: [BarChartRodData(toY: 6, color: Colors.green)],
            ),
          ],
        ),
      ),
    );
  }
}
```



2. Flutter Form (Text Input & Validation)

Forms allow users to input data with validation.

```dart
class _MyHomePageState extends State<MyHomePage> {
  final _formKey = GlobalKey<FormState>();
  String _name = '';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(widget.title),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: Column(
            children: [
              TextFormField(
                decoration: InputDecoration(labelText: "Enter your name"),
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return "Please enter your name";
                  }
                  return null;
                },
                onSaved: (value) {
                  _name = value!;
                },
              ),
              SizedBox(height: 20),
              ElevatedButton(
                onPressed: () {
                  if (_formKey.currentState!.validate()) {
                    _formKey.currentState!.save();
                    ScaffoldMessenger.of(
                      context,
                    ).showSnackBar(SnackBar(content: Text("Hello, $_name!")));
                  }
                },
                child: Text("Submit"),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```


SDJIC
DEBUG
Enter your name
Please enter your name
Submit

Difference Between MainAxis And CrossAxis

In Flutter, **MainAxis** and **CrossAxis** are used in **Row** and **Column** widgets to control the layout and alignment of children.

1. **What is MainAxis?**

The **MainAxis** refers to the **primary direction** in which a widget (like Row or Column) arranges its children.

Widget	**Main Axis Direction**
Row	Horizontal (Left to Right)
Column	Vertical (Top to Bottom)

It is controlled using mainAxisAlignment.

2. **What is CrossAxis?**

The **CrossAxis** is **perpendicular** to the **MainAxis** and controls the secondary alignment of children.

Widget	**Cross Axis Direction**
Row	Vertical (Top to Bottom)
Column	Horizontal (Left to Right)

It is controlled using crossAxisAlignment.