# Unit-4: Introduction of Flutter:

## 4.1 Fundamentals of Flutter:

Flutter is a UI toolkit for building fast, beautiful, natively compiled applications for mobile, web, and desktop with one programming 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 v1.9.1+hotfix.6 on October 24, 2019.

**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 programming 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(original equipment manufacturer widgets)** 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 programming language that allows developers to read, change, replace, or remove things easily. It gives excellent control to the developers over the system.

### 4.1.1 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

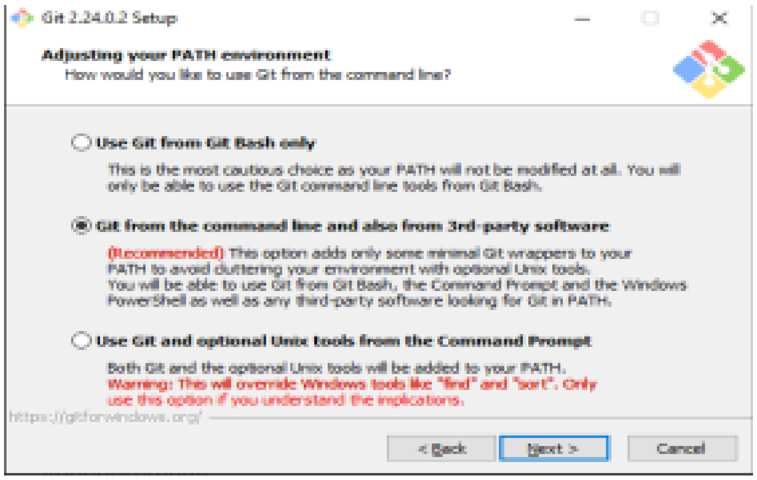
To install and run Flutter on the Windows system, you need first to meet these requirements for your development environment.

| | |
|---|---|
| **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<br>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.


Git 2.24.0.2 Setup

**Adjusting your PATH environment**
How would you like to use Git from the command line?

○ **Use Git from Git Bash only**
This is the most cautious choice as your PATH will not be modified at all. You will only be able to use the Git command line tools from Git Bash.

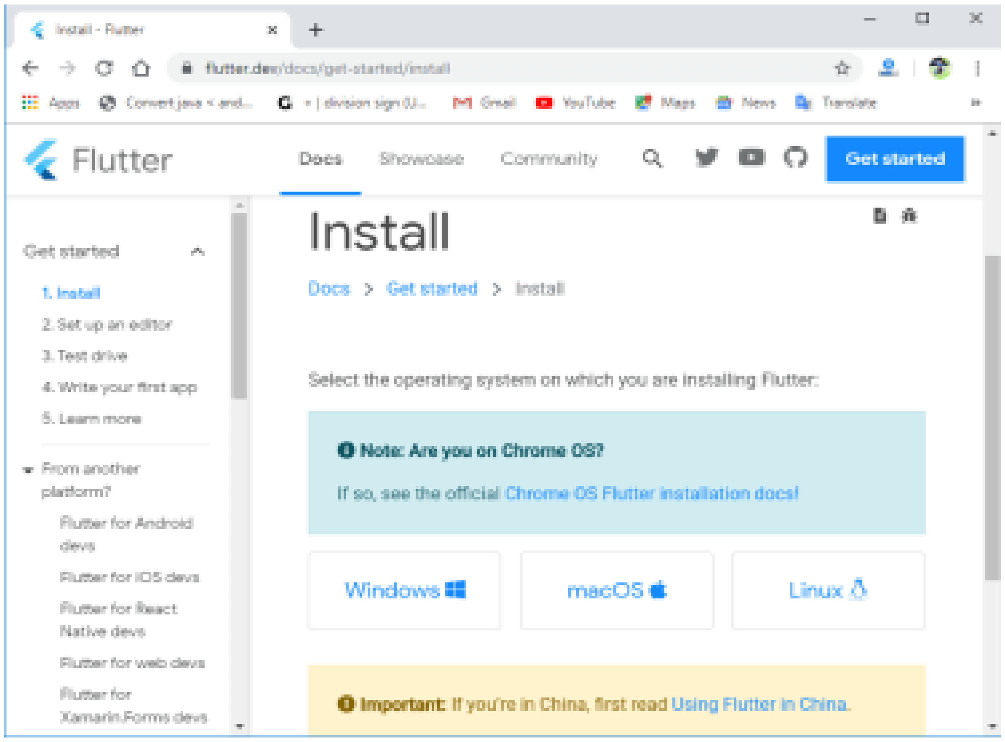
[✓ **Git from the command line and also from 3rd-party software**]
(Recommended) This option adds only some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools.
You will be able to use Git from Git Bash, the Command Prompt and the Windows PowerShell as well as any third-party software looking for Git in PATH.

○ **Use Git and optional Unix tools from the Command Prompt**
Both Git and the optional Unix tools will be added to your PATH.
Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.

https://gitforwindows.org/

**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.



# Install

Select the operating system on which you are installing Flutter:

**Note: Are you on Chrome OS?**

If so, see the official Chrome OS Flutter installation docs!

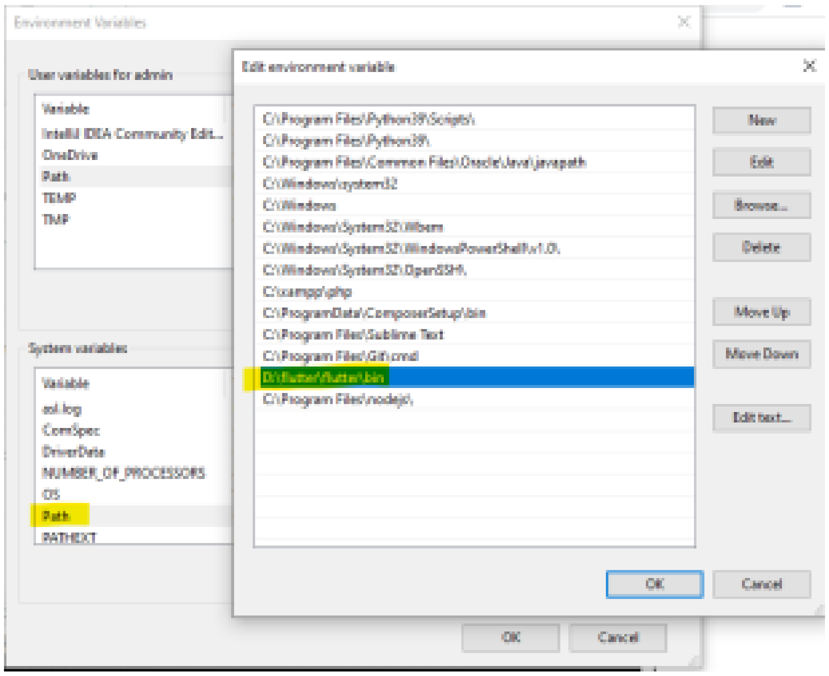
**Important:** If you're in China, first read Using Flutter in China.

**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.



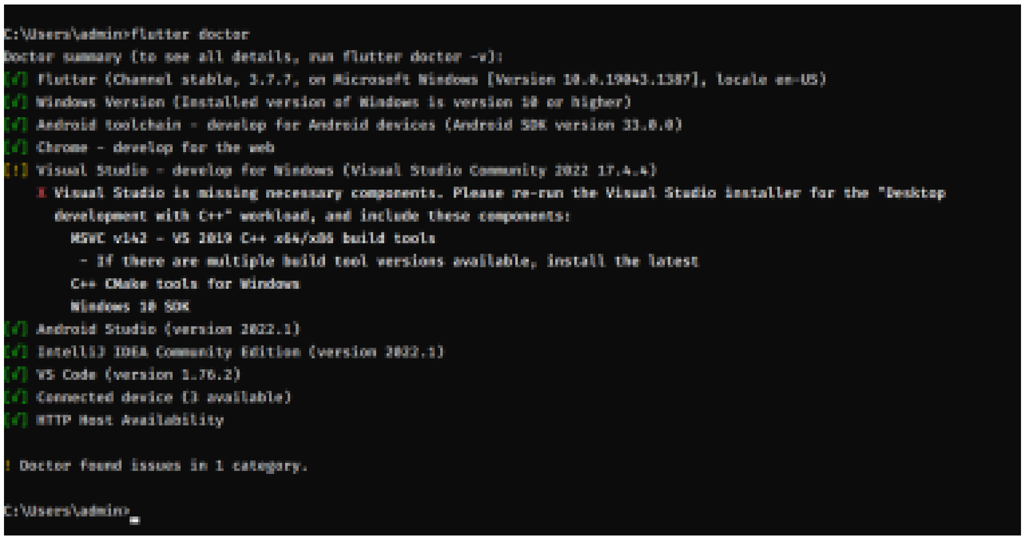
```
C:\Program Files\Python39\Scripts\
C:\Program Files\Python39\
C:\Program Files\Common Files\Oracle\Java\javapath
C:\Windows\system32
C:\Windows
C:\Windows\System32\Wbem
C:\Windows\System32\WindowsPowerShell\v1.0\
C:\Windows\System32\OpenSSH\
C:\xampp\php
C:\ProgramData\ComposerSetup\bin
C:\Program Files\Sublime Text
C:\Program Files\Git\cmd
D:\flutter\flutter\bin
C:\Program Files\nodejs\
```

**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.

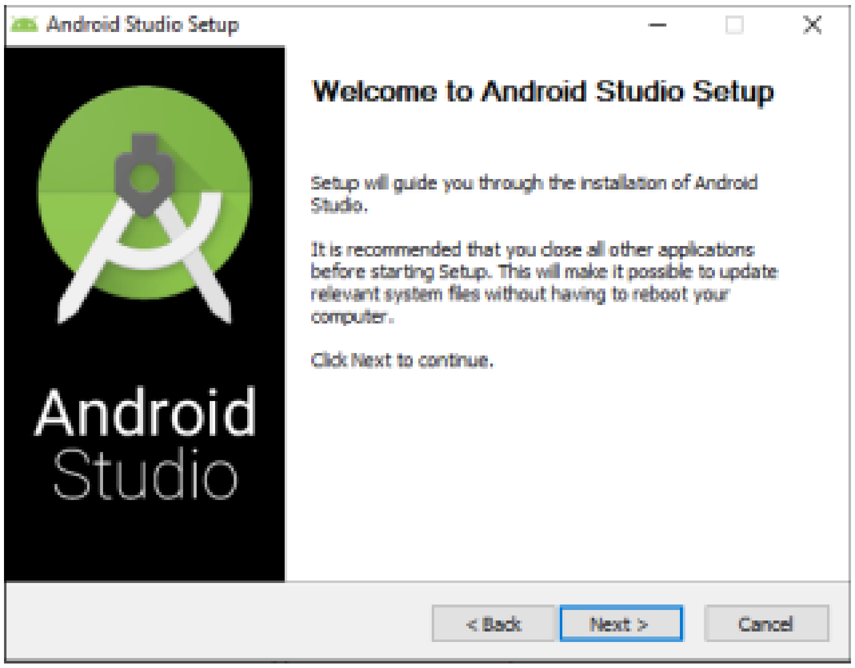


```
C:\Users\admin>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 3.7.7, on Microsoft Windows [Version 10.0.19043.1387], locale en-US)
[√] Windows Version (Installed version of Windows is version 10 or higher)
[√] Android toolchain - develop for Android devices (Android SDK version 33.0.0)
[√] Chrome - develop for the web
[!] Visual Studio - develop for Windows (Visual Studio Community 2022 17.4.4)
    X Visual Studio is missing necessary components. Please re-run the Visual Studio installer for the "Desktop
      development with C++" workload, and include these components:
        MSVC v142 - VS 2019 C++ x64/x86 build tools
         - If there are multiple build tool versions available, install the latest
        C++ CMake tools for Windows
        Windows 10 SDK
[√] Android Studio (version 2022.1)
[√] IntelliJ IDEA Community Edition (version 2022.1)
[√] VS Code (version 1.76.2)
[√] Connected device (3 available)
[√] HTTP Host Availability

! Doctor found issues in 1 category.

C:\Users\admin>
```

**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.



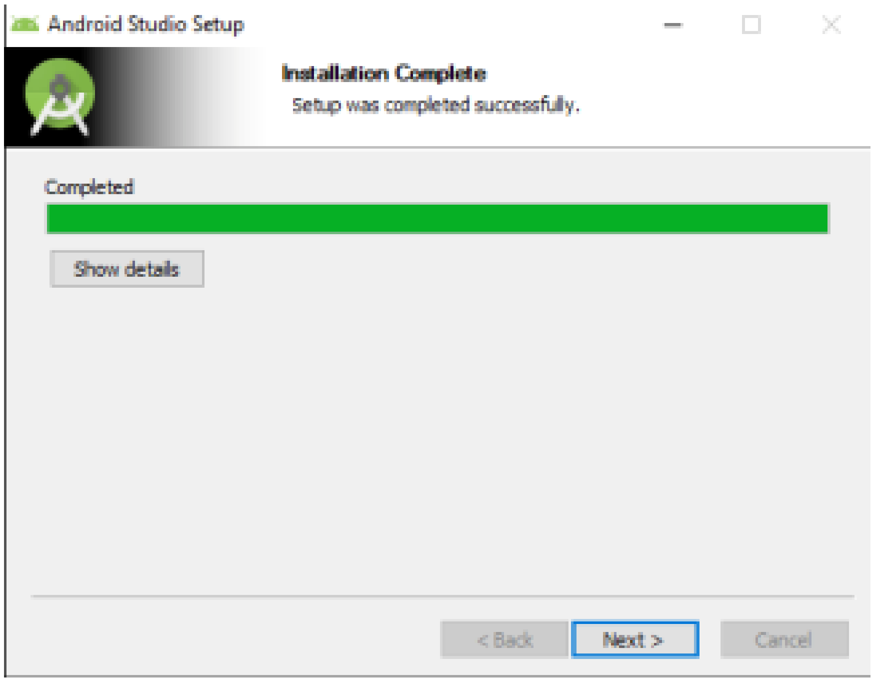
Welcome to Android Studio Setup

Setup will guide you through the installation of Android Studio.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

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

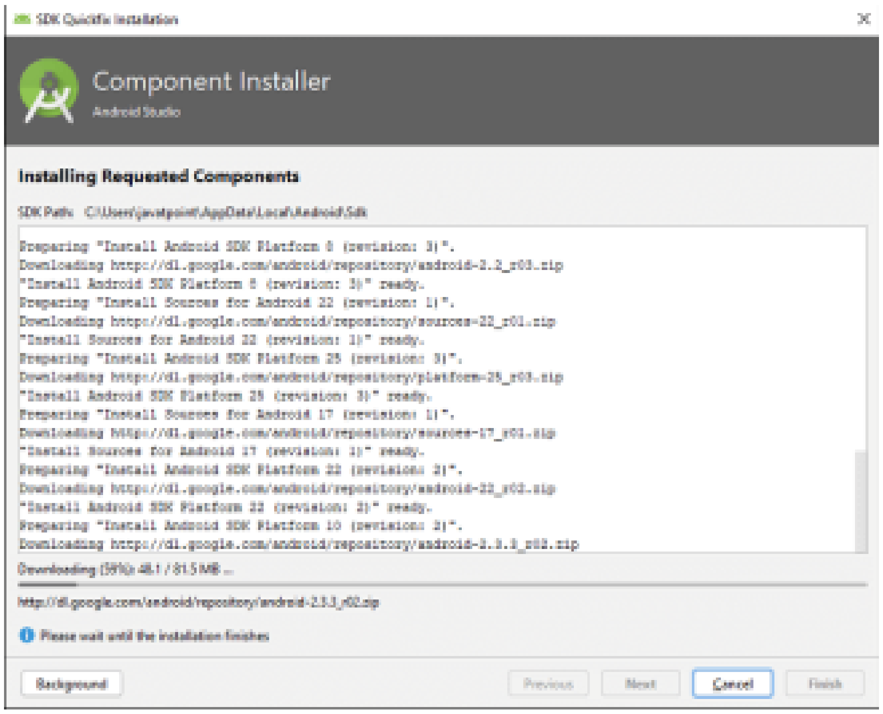


Installation Complete

Setup was completed successfully.

Completed

**Step 7.4**: In the above screen, click Next-> Finish. Once the Finish button is clicked, you need to choose the 'Don't import Settings option' and click OK. It will start the Android Studio.



Installing Requested Components

SDK Path: C:\Users\javatpoint\AppData\Local\Android\Sdk

```
Preparing "Install Android SDK Platform 8 (revision: 3)".
Downloading https://dl.google.com/android/repository/android-2.2_r03.zip
"Install Android SDK Platform 8 (revision: 3)" ready.
Preparing "Install Sources for Android 22 (revision: 1)".
Downloading https://dl.google.com/android/repository/sources-22_r01.zip
"Install Sources for Android 22 (revision: 1)" ready.
Preparing "Install Android SDK Platform 25 (revision: 3)".
Downloading https://dl.google.com/android/repository/platform-25_r03.zip
"Install Android SDK Platform 25 (revision: 3)" ready.
Preparing "Install Sources for Android 17 (revision: 1)".
Downloading https://dl.google.com/android/repository/sources-17_r01.zip
"Install Sources for Android 17 (revision: 1)" ready.
Preparing "Install Android SDK Platform 22 (revision: 2)".
Downloading https://dl.google.com/android/repository/android-22_r02.zip
"Install Android SDK Platform 22 (revision: 2)" ready.
Preparing "Install Android SDK Platform 10 (revision: 2)".
Downloading https://dl.google.com/android/repository/android-2.3.3_r02.zip
```

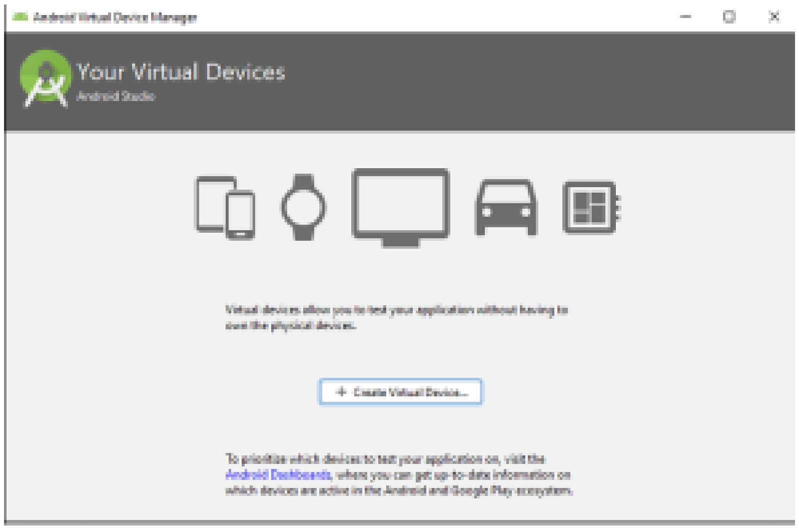
Downloading (59%): 48.1 / 81.5 MB ...

http://dl.google.com/android/repository/android-2.3.3_r02.zip

Please wait until the installation finishes

**Step 8:** Next, you need to set up an Android emulator. It is responsible for running and testing the Flutter application.

**Step 8.1:** To set an Android emulator, go to Android Studio > Tools > Android > AVD Manager and select Create Virtual Device. Or, go to Help->Find Action->Type Emulator in the search box. You will get the following screen.



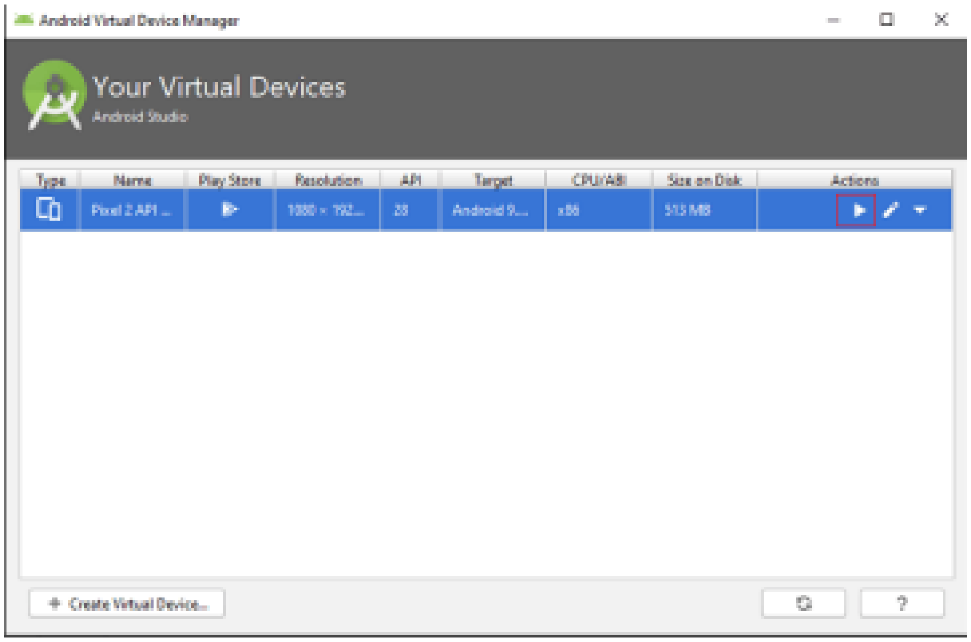
Virtual devices allow you to test your application without having to own the physical devices.

To prioritize which devices to test your application on, visit the Android Dashboards, where you can get up-to-date information on which devices are active in the Android and Google Play ecosystem.

**Step 8.2:** Choose your device definition and click on Next.

**Step 8.3:** Select the system image for the latest Android version and click on Next.

**Step 8.4:** Now, verify the all AVD configuration. If it is correct, click on Finish. The following screen appears.

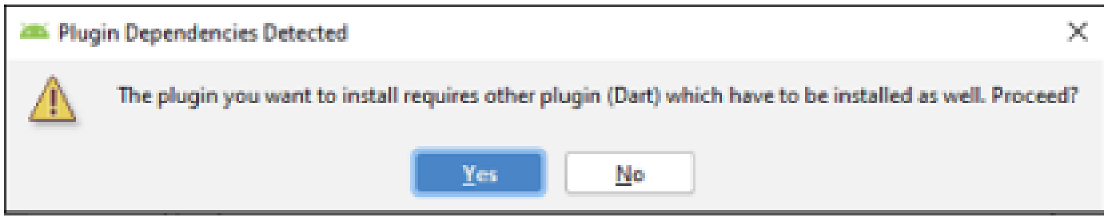


Step 8.5: Last, click on the icon pointed into the red color rectangle. The Android emulator displayed as below screen.

**Step 9:** Now, install Flutter and Dart plugin for building Flutter application in Android Studio. These plugins provide a template to create a Flutter application, give an option to run and debug Flutter application in the Android Studio itself. Do the following steps to install these plugins.

**Step 9.1:** Open the Android Studio and then go to File->Settings->Plugins.

**Step 9.2:** Now, search the Flutter plugin. If found, select Flutter plugin and click install. When you click on install, it will ask you to install Dart plugin as below screen. Click yes to proceed.

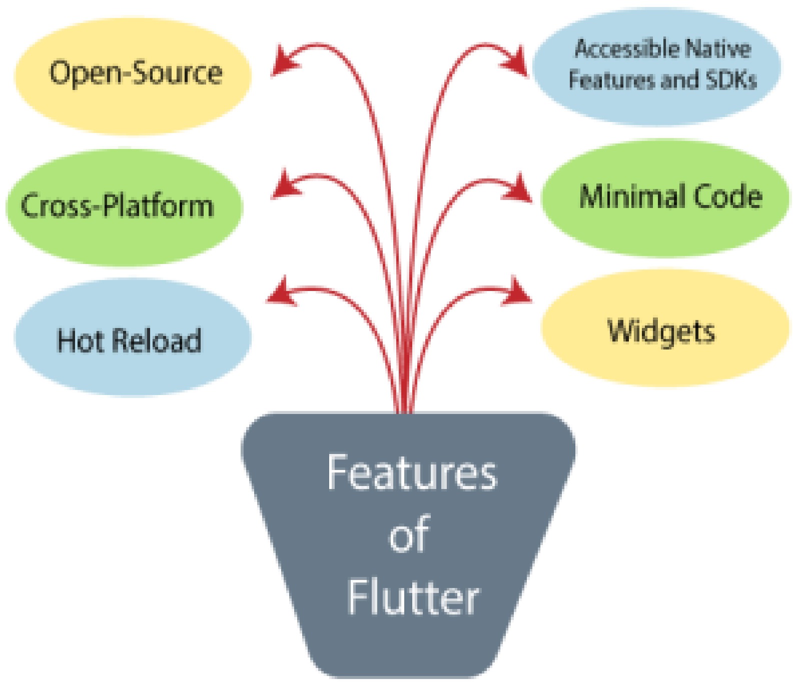


The plugin you want to install requires other plugin (Dart) which have to be installed as well. Proceed?

**Step 9.3:** Restart the Android Studio.

### 4.1.2 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.

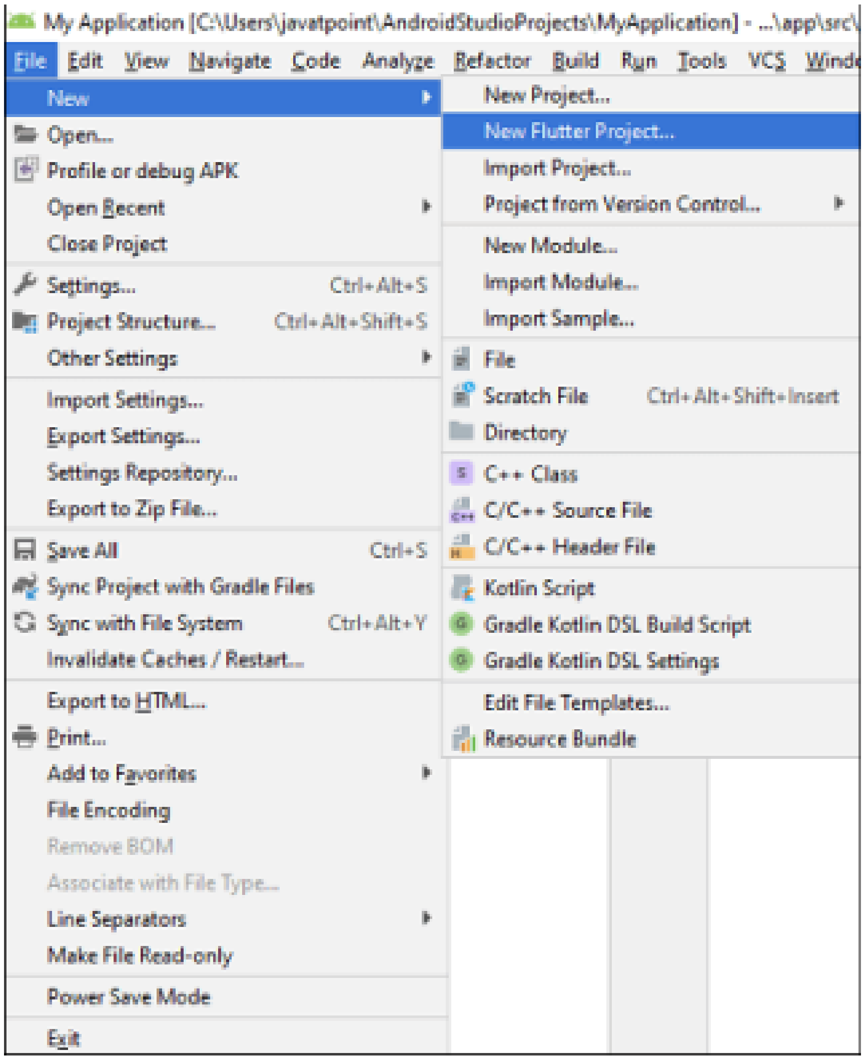
### 4.1.3 Creating basic flutter project using Android Studio

**Flutter First Application**

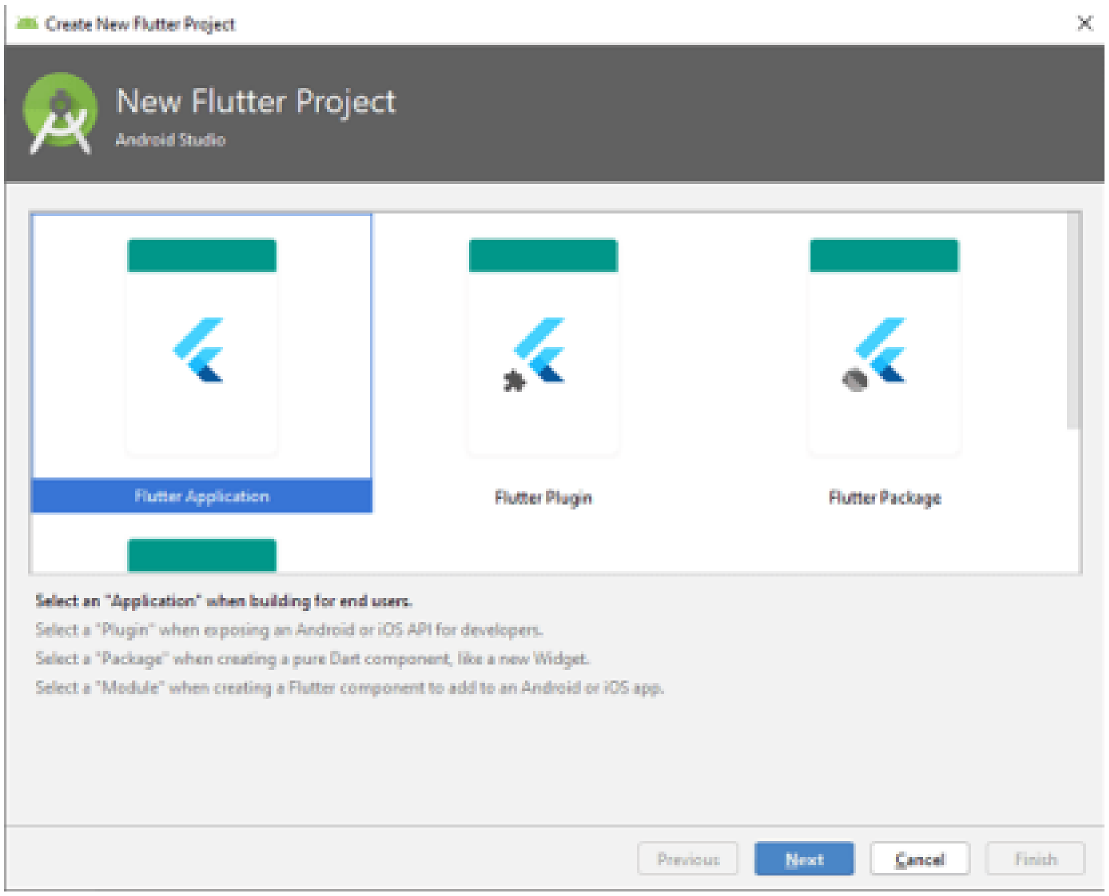
In this section, we are going to learn how to create a simple application in Android Studio to understand the basics of the Flutter application. To create Flutter application, do the following steps:

**Step 1:** Open the Android Studio.

**Step 2:** Create the Flutter project. To create a project, go to File-> New->New Flutter Project. The following screen helps to understand it more clearly.



**Step 3:** In the next wizard, you need to choose the Flutter Application. For this, select Flutter Application-> click Next, as shown in the below screen.



Select an "Application" when building for end users.
Select a "Plugin" when exposing an Android or iOS API for developers.
Select a "Package" when creating a pure Dart component, like a new Widget.
Select a "Module" when creating a Flutter component to add to an Android or iOS app.

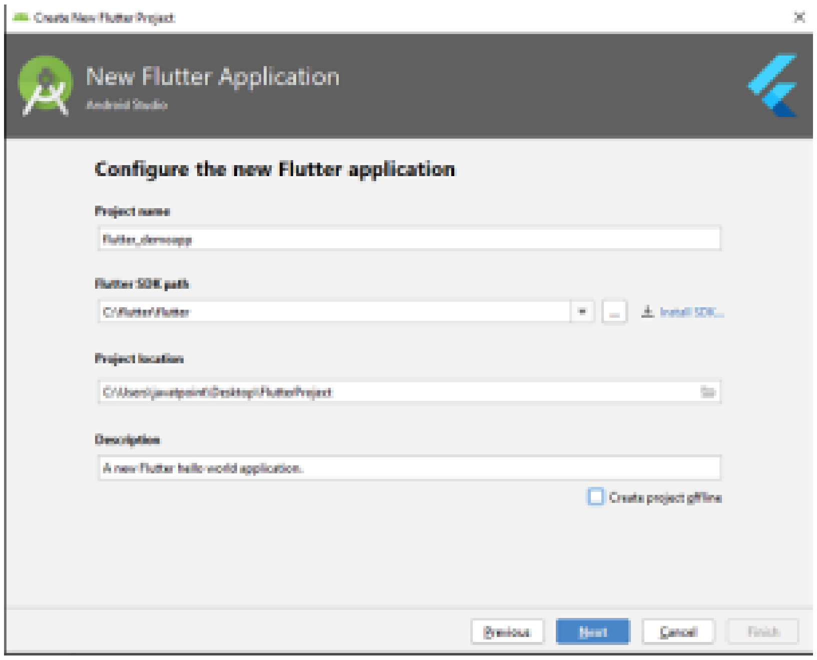
**Step 4:** Next, configure the application details as shown in the below screen and click on the Next button.

**Project Name:** Write your Application Name.

**Flutter SDK Path**: <path_to_flutter_sdk>

**Project Location:** <path_to_project_folder>

**Descriptions:** <A new Flutter hello world application>.



**Configure the new Flutter application**

Project name

flutter_demoapp

Flutter SDK path

C:\flutter\flutter

Project location

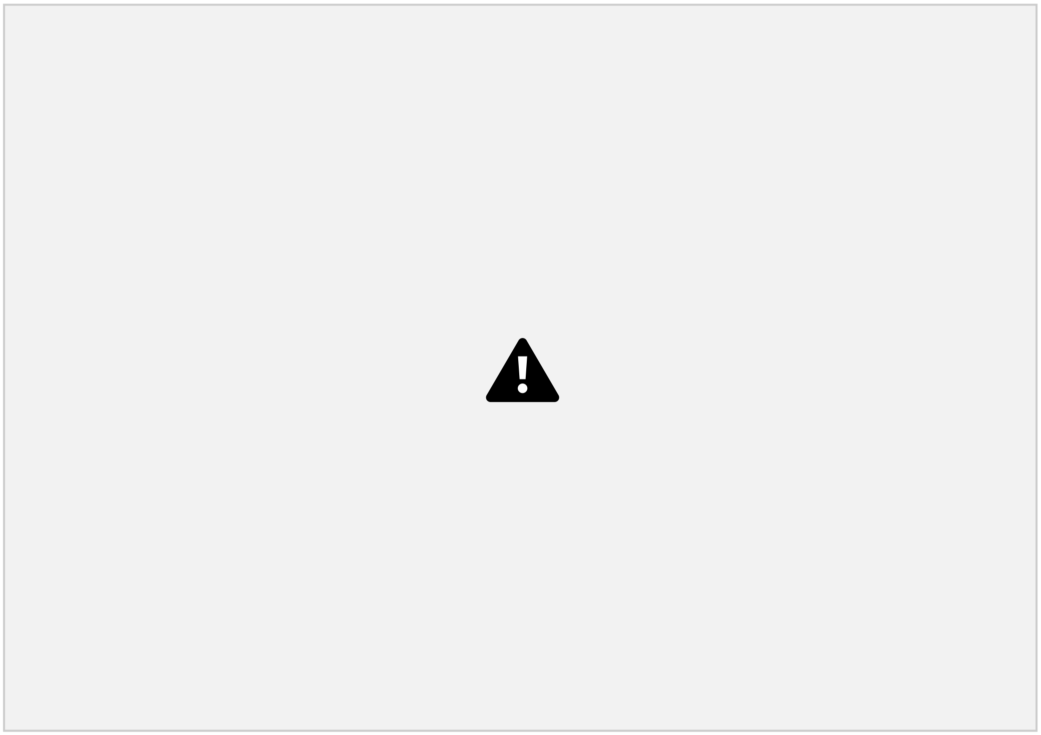
C:\Users\javatpoint\Desktop\FlutterProject

Description

A new Flutter hello world application.

**Step 5:** In the next wizard, you need to set the company domain name and click the Finish button.
After clicking the Finish button, it will take some time to create a project. When the project is created, you will get a fully working Flutter application with minimal functionality.

**Step 6:** Now, let us check the structure of the Flutter project application and its purpose. In the below image, you can see the various folders and components of the Flutter application structure, which are going to discuss here.



**.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.

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

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.

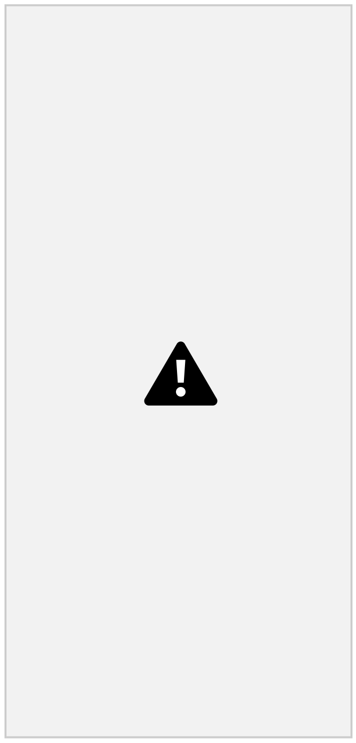
- To start Flutter programming, you need first to import the Flutter package. Here, we have imported a **Material package**. This package allows you to create user interface according to the Material design guidelines specified by Android.
- The second line is an entry point of the Flutter applications 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.



**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**

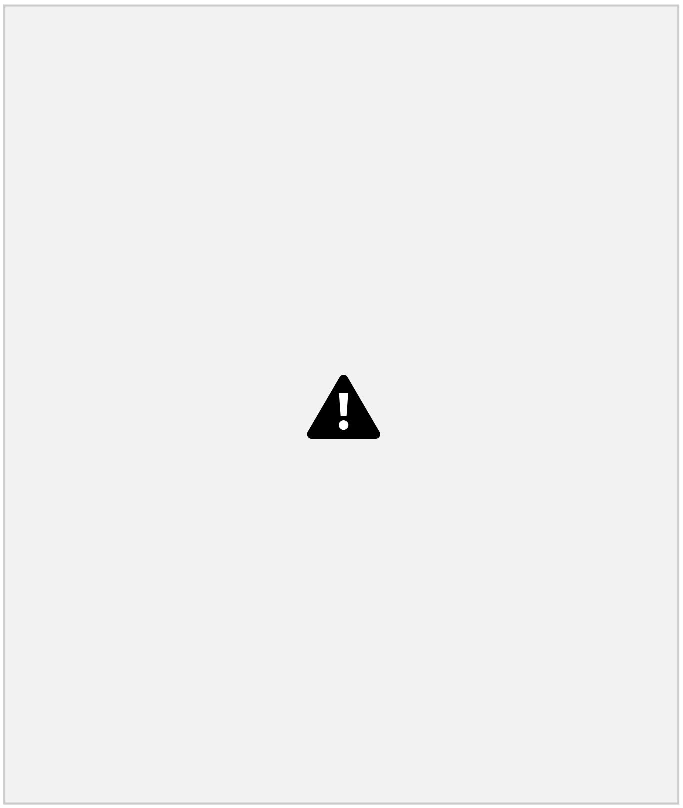
It contains all the required packages for the basic building blocks of writing a Flutter application. These libraries are written in Dart language.

**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**

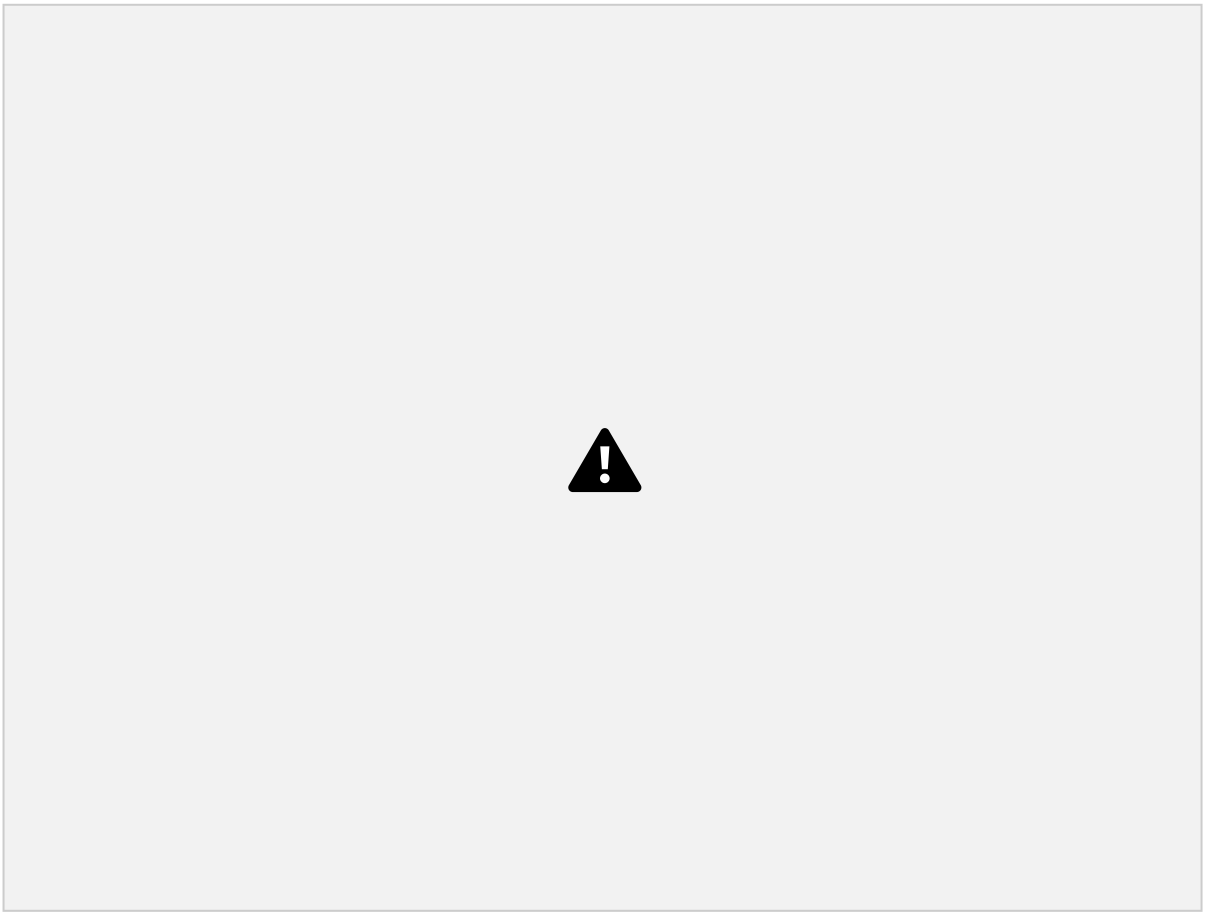
It is a widget that provides interaction (how to listen for and respond to) in Flutter using GestureDetector. **GestureDector** is an invisible widget, which includes tapping, dragging, and scaling interaction of its child widget. We can also use other interactive features into the existing widgets by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

**State Management**

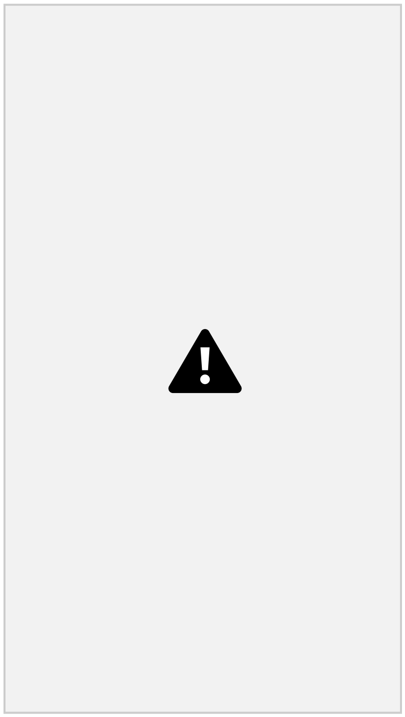
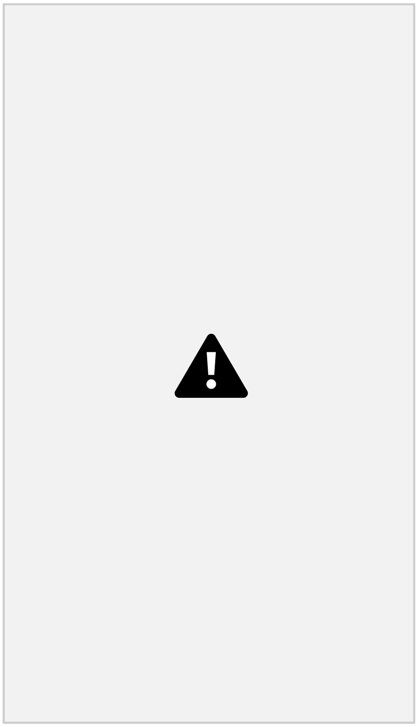
Flutter widget maintains its state by using a special widget, StatefulWidget. It is always auto re-rendered whenever its internal state is changed. The re rendering is optimized by calculating the distance between old and new widget UI and render only necessary things that are changes.

**Layers**

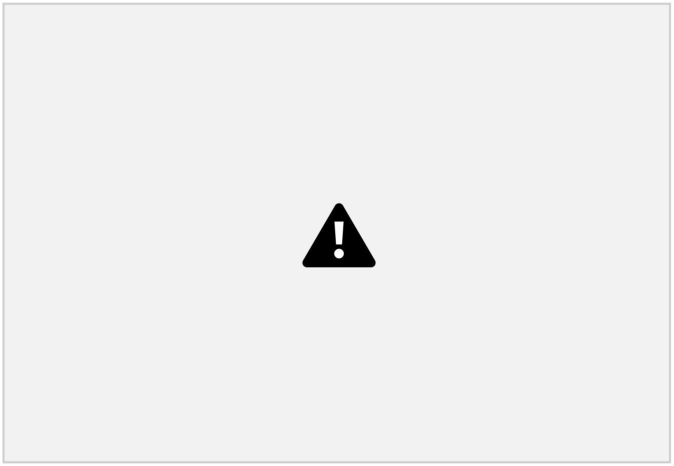
Layers are an important concept of the Flutter framework, which are grouped into multiple categories in terms of complexity and arranged in the top-down approach. The topmost layer is the UI of the application, which is specific to the Android and iOS platforms. The second topmost layer contains all 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.



(Scaffold is a class in flutter which provides many widgets or we can say APIs like Drawer, Snack-Bar, Bottom-Navigation-Bar, Floating-Action-Button, App Bar, etc. Scaffold will expand or occupy the whole device screen. It will occupy the available space.)

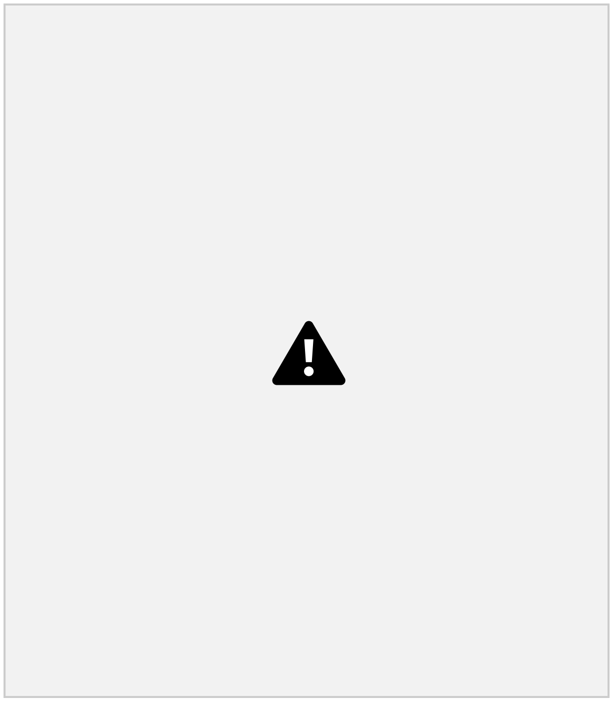
**4.2 Flutter Widget:**



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'),
),
);
}
}
```

### 4.2.1 Types of flutter widget:

#### 4.2.1.1 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/
```

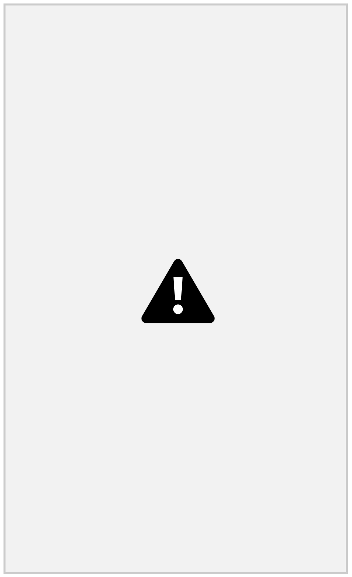
Now, add the following line in the dart file.

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

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

```
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: Image.asset('assets/computer.png'),
),
);
}
}
```

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.

```
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.

### 4.2.1.2 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**

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

@override
_CarState createState() => _CarState();
}

class _CarState extends State<Car> {
@override
Widget build(BuildContext context) {
return Container(
color: const Color(0xFEEFE),
child: Container(
child: Container( //child: Container() )
)
);
}
}
```

**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**

```
class MyStatelessCarWidget extends StatelessWidget
{
const MyStatelessCarWidget ({ Key key }) : super(key: key);

@override
Widget build(BuildContext context) {
return Container(color: const Color(0x0xFEEFE));
}
}
```

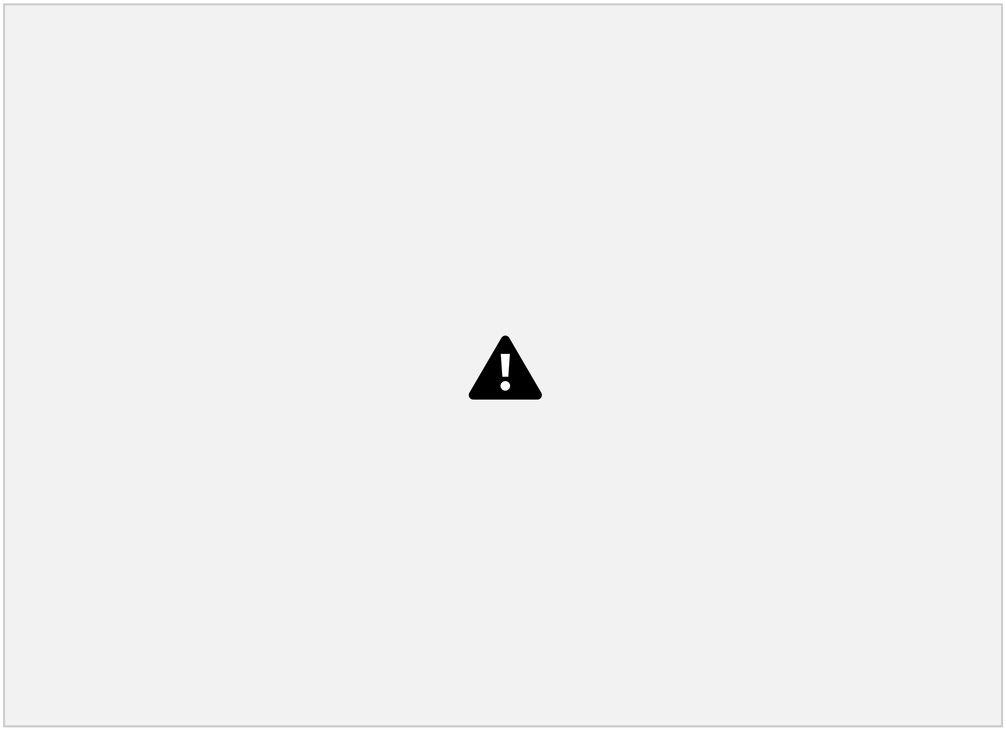
### 4.2.1.3 Single child widget and multiple child widgets

**Flutter Layouts**

The main concept of the layout mechanism is the widget. We know that flutter assume everything as a widget. So the image, icon, text, and even the layout of your app are all widgets. Here, some of the things you do not see on your app UI, such as rows, columns, and grids that arrange, constrain, and align the visible widgets are also the widgets.

Flutter allows us to create a layout by composing multiple widgets to build more complex widgets. For example, we can see the below image that shows three icons with a label under each one.

In the second image, we can see the visual layout of the above image. This image shows a row of three columns, and these columns contain an icon and label.



In the above image, the container is a widget class that allows us to customize the child widget. *It is mainly used to add borders, padding, margins, background color, and many more.* Here, the text widget comes under the container for adding margins. The entire row is also placed in a container for adding margin and padding around the row. Also, the rest of the UI is controlled by properties such as color, text.style, etc.

**Layout a widget**
Let us learn how we can create and display a simple widget. The following steps show how to layout a widget:
**Step 1:** First, you need to select a Layout widget.

**Step 2:** Next, create a visible widget.

**Step 3:** Then, add the visible widget to the layout widget.

**Step 4:** Finally, add the layout widget to the page where you want to display.

**Types of Layout Widgets**
We can categories the layout widget into two types:
1. Single Child Widget
2. Multiple Child Widget

**Single Child Widgets**
The single child layout widget is a type of widget, which can have only **one child widget** inside the parent layout widget. These widgets can also contain special layout functionality. Flutter provides us many single child widgets to make the app UI attractive. If we use these widgets appropriately, it can save

our time and makes the app code more readable. The list of different types of single child widgets are:

**Container:** It is the most popular layout widget that provides customizable options for painting, positioning, and sizing of widgets.

```
Center(
child: Container(
margin: const EdgeInsets.all(15.0),
color: Colors.blue,
width: 42.0,
height: 42.0,
),
)
```

**Padding:** It is a widget that is used to arrange its child widget by the given padding. It contains **EdgeInsets** and **EdgeInsets.fromLTRB** for the desired side where you want to provide padding.

```
const Greetings(
child: Padding(
padding: EdgeInsets.all(14.0),
child: Text('Hello SDJIC!'),
),
)
```

**Center:** This widget allows you to center the child widget within itself.

**Align:** It is a widget, which aligns its child widget within itself and sizes it based on the child's size. It provides more control to place the child widget in the exact position where you need it.

```
Center(
child: Container(
height: 110.0,
width: 110.0,
color: Colors.blue,
child: Align(
alignment: Alignment.topLeft,
child: FlutterLogo(
size: 50,
),
),
),
)
```

**SizedBox:** This widget allows you to give the specified size to the child widget through all screens.

```
SizedBox(
width: 300.0,
height: 450.0,
child: const Card(child: Text('Hello JavaTpoint!')),
)
```

**AspectRatio:** This widget allows you to keep the size of the child widget to a specified aspect ratio.

```
AspectRatio(
aspectRatio: 5/3,
child: Container(
color: Colors.bluel,
),
),
```

**Baseline:** This widget shifts the child widget according to the child's baseline.

```
child: Baseline(
baseline: 30.0,
baselineType: TextBaseline.alphabetic,
child: Container(
height: 60,
width: 50,
color: Colors.blue,
),
)
```

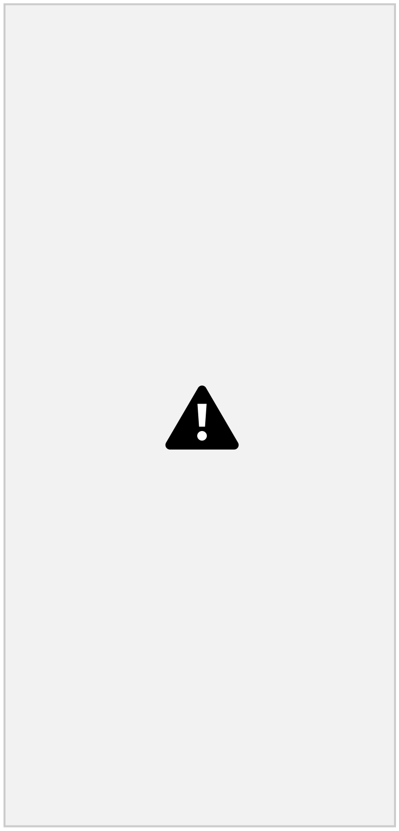
**ConstrainedBox:** It is a widget that allows you to force the additional constraints on its child widget. It means you can force the child widget to have a specific constraint without changing the properties of the child widget.

```
ConstrainedBox(
constraints: new BoxConstraints(
minHeight: 150.0,
minWidth: 150.0,
maxHeight: 300.0,
maxWidth: 300.0,
),
child: new DecoratedBox(
decoration: new BoxDecoration(color: Colors.red),
),
),
```

**CustomSingleChildLayout:** It is a widget, which defers from the layout of the single child to a delegate. The delegate decides to position the child widget and also used to determine the size of the parent widget.

**FittedBox:** It scales and positions the child widget according to the specified **fit**.

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

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

class MyApp extends StatelessWidget {
// It is the root widget of your application.
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Multiple Layout Widget',
debugShowCheckedModeBanner: false,
theme: ThemeData(
// This is the theme of your application.
primarySwatch: Colors.green,
),
home: MyHomePage(),
);
}
}
class MyHomePage extends StatelessWidget {

@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(title: Text("FittedBox Widget")),
body: Center(
child: FittedBox(child: Row(
children: <Widget>[
Container(
child: Image.asset('assets/computer.png'),
),
Container(
child: Text("This is a widget"),
)
],
),
fit: BoxFit.contain,
)
),
);
}
}
```



**FractionallySizedBox:** It is a widget that allows to sizes of its child widget according to the fraction of the available space.

**IntrinsicHeight and IntrinsicWidth:** They are a widget that allows us to sizes its child widget to the child's intrinsic height and width.

**LimitedBox:** This widget allows us to limits its size only when it is unconstrained.

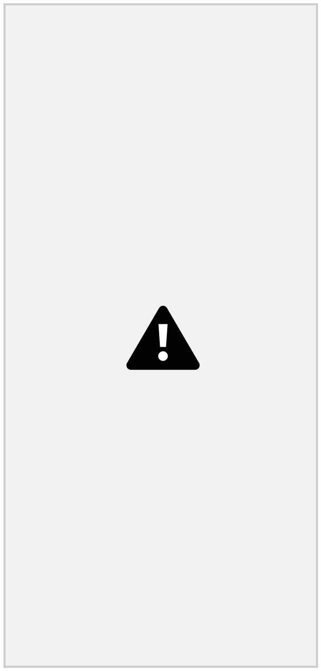
**Offstage:** It is used to measure the dimensions of a widget without bringing it on to the screen.

**OverflowBox:** It is a widget, which allows for imposing different constraints on its child widget than it gets from a parent. In other words, it allows the child to overflow the parent widget.

**Example**

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

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

class MyApp extends StatelessWidget {
// It is the root widget of your application.
@override
Widget build(BuildContext context) {
```

```
return MaterialApp(
title: 'Single Layout Widget',
debugShowCheckedModeBanner: false,
theme: ThemeData(
// This is the theme of your application.
primarySwatch: Colors.blue,
),
home: MyHomePage(),
);
}
}
class MyHomePage extends StatelessWidget {

@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text("OverflowBox Widget"),
),
body: Center(
child: Container(
height: 50.0,
width: 50.0,
color: Colors.red,
child: OverflowBox(
minHeight: 70.0,
minWidth: 70.0,
child: Container(
height: 50.0,
width: 50.0,
color: Colors.blue,
),
),
),
),
);
}
}
```



**Multiple Child widgets**

The multiple child widgets are a type of widget, which contains more than one child widget, and the layout of these widgets are unique. For example, Row widget laying out of its child widget in a horizontal direction, and Column widget laying out of its child widget in a vertical direction. If we combine the Row and Column widget, then it can build any level of the complex widget.

Here, we are going to learn different types of multiple child widgets: **Row:** It allows arranging its child widgets in a horizontal direction. **Example:**

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

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

class MyApp extends StatelessWidget {
// It is the root widget of your application.
@override
Widget build(BuildContext context) {
return MaterialApp(
```

```dart
title: 'Multiple Layout Widget',
debugShowCheckedModeBanner: false,
theme: ThemeData(
// This is the theme of your application.
primarySwatch: Colors.blue,
),
home: MyHomePage(),
);
}
}
class MyHomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Center(
child: Container(
alignment: Alignment.center,
color: Colors.white,
child: Row(
children: <Widget>[
Expanded(
child: Text('Peter', textAlign: TextAlign.center),
),
Expanded(
child: Text('John', textAlign: TextAlign.center ),

),
Expanded(
child: FittedBox(
fit: BoxFit.contain, // otherwise the logo will be tiny
child: const FlutterLogo(),
),
),
],
),
),
);
}
}
```

**Column:** It allows to arrange its child widgets in a vertical direction.

**ListView:** It is the most popular scrolling widget that allows us to arrange its child widgets one after another in scroll direction.

**GridView:** It allows us to arrange its child widgets as a scrollable, 2D array of widgets. It consists of a repeated pattern of cells arrayed in a horizontal and vertical layout.

**Expanded:** It allows to make the children of a Row and Column widget to occupy the maximum possible area.

**Table:** It is a widget that allows us to arrange its children in a table based widget.

**Flow:** It allows us to implements the flow-based widget.

**Stack:** It is an essential widget, which is mainly used for overlapping several children widgets. It allows you to put up the multiple layers onto the screen.

### 4.2.2 Visible widget(Constructor and Properties):

Text, Image, Button, Icon
Flutter Images
In this section, we are going to see how we can display images in Flutter. When you create an app in Flutter, it includes both code and assets (resources). An asset is a file, which is bundled and deployed with the app and is accessible at runtime. The asset can include static data, configuration files, icons, and images. The Flutter supports many image formats, such as JPEG, WebP, PNG, GIF, animated WebP/GIF, BMP, and WBMP.
Displaying images is the fundamental concept of most of the mobile apps. Flutter has an Image widget that allows displaying different types of images in the mobile application.

**How to display the image in Flutter**

To display an image in Flutter, do the following steps:

**Step 1:** First, we need to create a new **folder** inside the root of the Flutter project and named it assets. We can also give it any other name if you want.

**Step 2:** Next, inside this folder, add one image manually.

**Step 3:** Update the **pubspec.yaml** file. Suppose the image name is **tablet.png,** then pubspec.yaml file is:

```
assets:
- assets/tablet.png
- assets/background.png
```

If the assets folder contains more than one image, we can include it by specifying the directory name with the slash (/) character at the end.

```
flutter:
assets:
- assets/
```

**Step 4:** Finally, open the**main.dart** file and insert the following code.

```
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('Flutter Image Demo'),
),
body: Center(
child: Column(
children: <Widget>[
Image.asset('assets/tablet.png'),
Text(
'A tablet is a wireless touch screen computer that is smaller than a notebook but larger than a
 smartphone.',
style: TextStyle(fontSize: 20.0),
)
],
),
),
),
);
}
}
```

**Step 5:** Now, run the app. You will get something like the screen below.

**Flutter Icons**

An icon is a **graphic image** representing an application or any specific entity containing meaning for the user. It can be selectable and non-selectable. **For example**, the company's logo is non-selectable. Sometimes it also contains a **hyperlink** to go to another page. It also acts as a sign in place of a detailed explanation of the actual entity.

Flutter provides an **Icon Widget** to create icons in our applications. We can create icons in Flutter, either using inbuilt icons or with the custom icons. Flutter provides the list of all icons in the **Icons class**. In this article, we are going to learn how to use Flutter icons in the application.

**Icon Widget Properties**

Flutter icons widget has different properties for customizing the icons. These properties are explained below:

| Property | Descriptions |
|---|---|

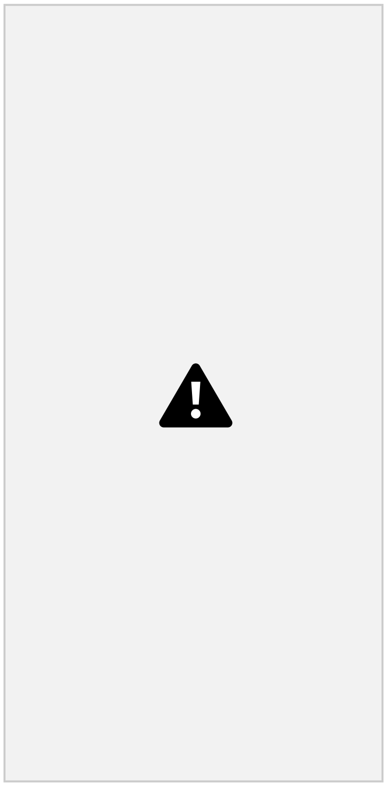
|  |  |
|---|---|
| icon | It is used to specify the icon name to display in the application. Generally, Flutter uses material design icons that are symbols for common actions and items. |
| color | It is used to specify the color of the icon. |
| size | It is used to specify the size of the icon in pixels. Usually, icons have equal height and width. |
| textDirectio n | It is used to specify to which direction the icon will be rendered. |

Let us understand Flutter icons using different examples.

**Example 1:**
In this example, we will see the basic icon widget that has default values. First, create a project in the IDE, navigate to the lib folder, and then open the main.dart file. Now, replace the below code in the main.dart file:

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

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

class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
return MaterialApp(
theme: ThemeData(
primarySwatch: Colors.blue,
),
home: MyIconPage(),
);
}
}

class MyIconPage extends StatefulWidget {
@override
_MyIconPageState createState() => _MyIconPageState();
}

class _MyIconPageState extends State<MyIconPage> {
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text('Flutter Icon Tutorial'),
),
body: Row(
mainAxisAlignment: MainAxisAlignment.spaceAround,
children: <Widget>[
Icon(Icons.camera_enhance),
Icon(Icons.camera_front),
Icon(Icons.camera_rear),
]),
);
}
}
```

**OUTPUT:**



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

**Unit-5: Basic Flutter widget ( Constructor, attributes and Properties)** 5.1 Text, TextField, Buttons, Slider
5.2 Checkbox, Radio Button, Progress Bar, Lists

5.3 Stack, Forms, AlertDialog, Tooltip
5.4 Toast, Switch, Charts, Flutter Form.

**For UNIT 5 Refer GIVEN PDFs**

**<u>Reference</u>**

1. **https://www.javatpoint.com/flutter**
2. **https://docs.flutter.dev/get-started/codelab**
3. **https://docs.flutter.dev/ui/widgets**
4. **https://docs.flutter.dev/ui/layout**
5. **https://www.geeksforgeeks.org/flutter-tutorial/**
6. **https://codelabs.developers.google.com/codelabs/flutter-codelab first#0**