UNIT-4

Flutter Tutorial

Our Flutter Tutorial provides basic and advanced concepts of the Flutter framework. 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 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 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.

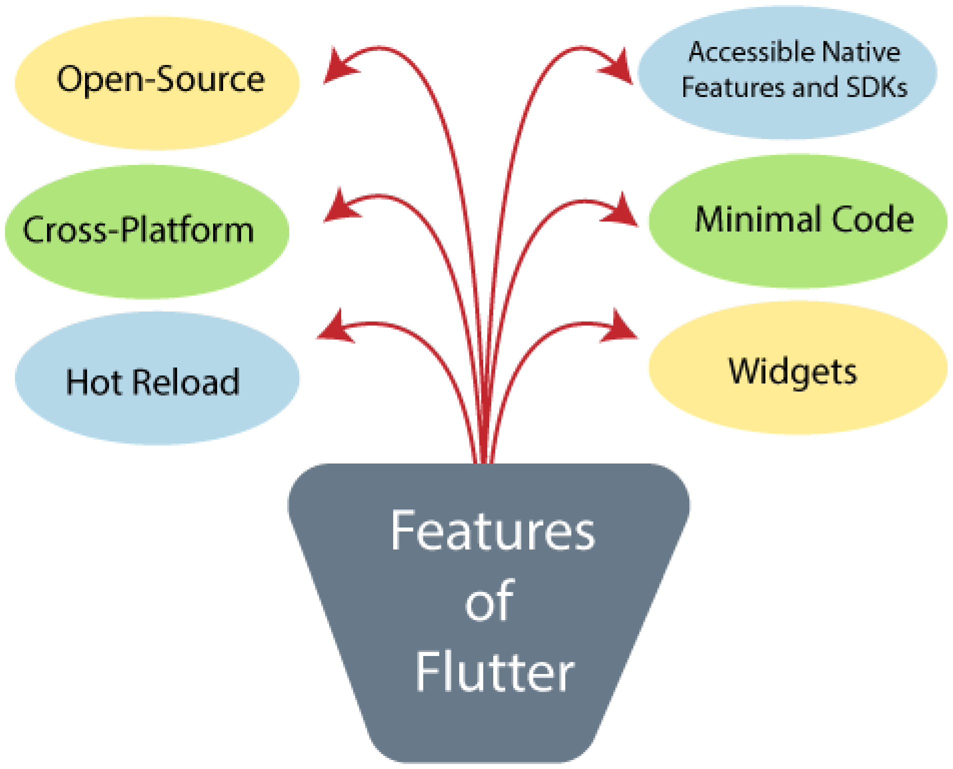
Flutter is mainly optimized for 2D mobile apps that can run on both Android and iOS platforms. We can also use it to build full-featured apps, including camera, storage, geolocation, network, third-party SDKs, and more.

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.

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.

Advantage of Flutter

Flutter fulfills the custom needs and requirements for developing mobile applications. It also offers many advantages, which are listed below.

- It makes the app development process extremely fast because of the hot-reload feature. This feature allows us to change or update the code are reflected as soon as the alterations are made.
- It provides the smoother and seamless scrolling experiences of using the app without much hangs or cuts, which makes running applications faster in comparison to other mobile app development frameworks.
- Flutter reduces the time and efforts of testing. As we know, flutter apps are cross-platform so that testers do not always need to run the same set of tests on different platforms for the same app.
- It has an excellent user interface because it uses a design-centric widget, high-development tools, advanced APIs, and many more features.
- It is similar to a reactive framework where the developers do not need to update the UI content manually.
- It is suitable for MVP (Minimum Viable Product) apps because of its speedy development process and cross-platform nature.

Disadvantages of Flutter

We have seen earlier that the Flutter has many advantages, but it also contains some disadvantages, which are given below.

- The Flutter is a comparatively new language that needs continuous integration support through the maintenance of scripts.

Flutter Installation

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

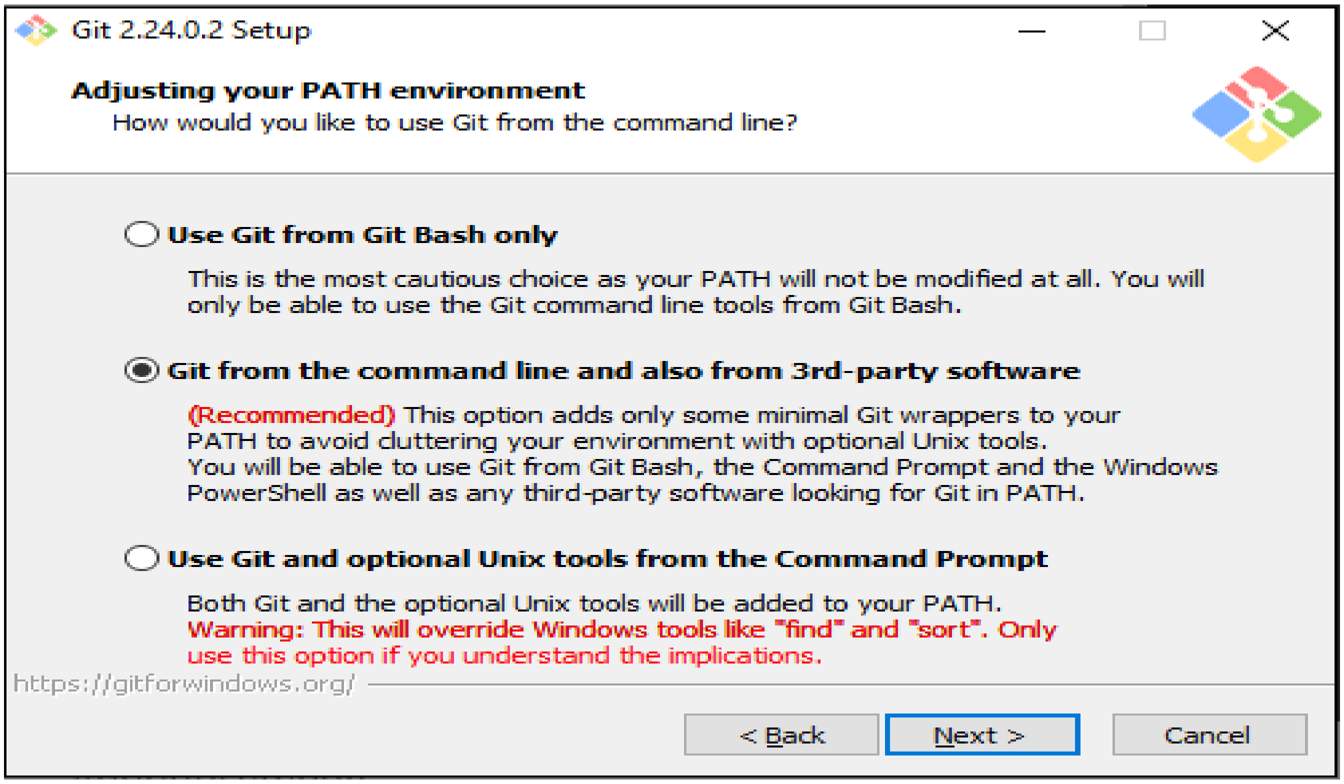
To install and run Flutter on the Windows system, you need first to meet these requirements for your development environment.

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, click here.

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**

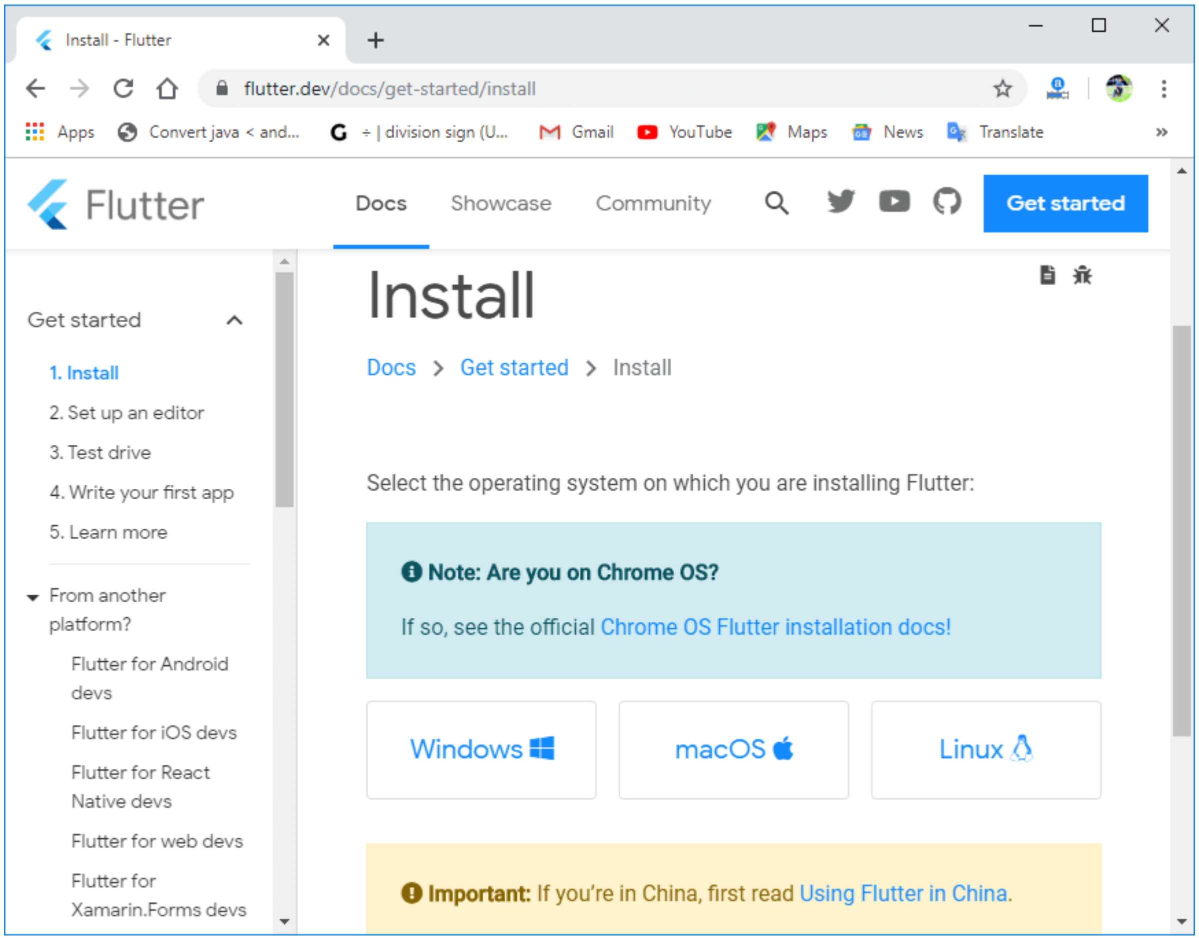
Both Git and the optional Unix tools will be added to your PATH. Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.

https://gitforwindows.org/

To read more information about installing Git, click here.

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 website, click on **Get started** button, you will get the following screen.



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.

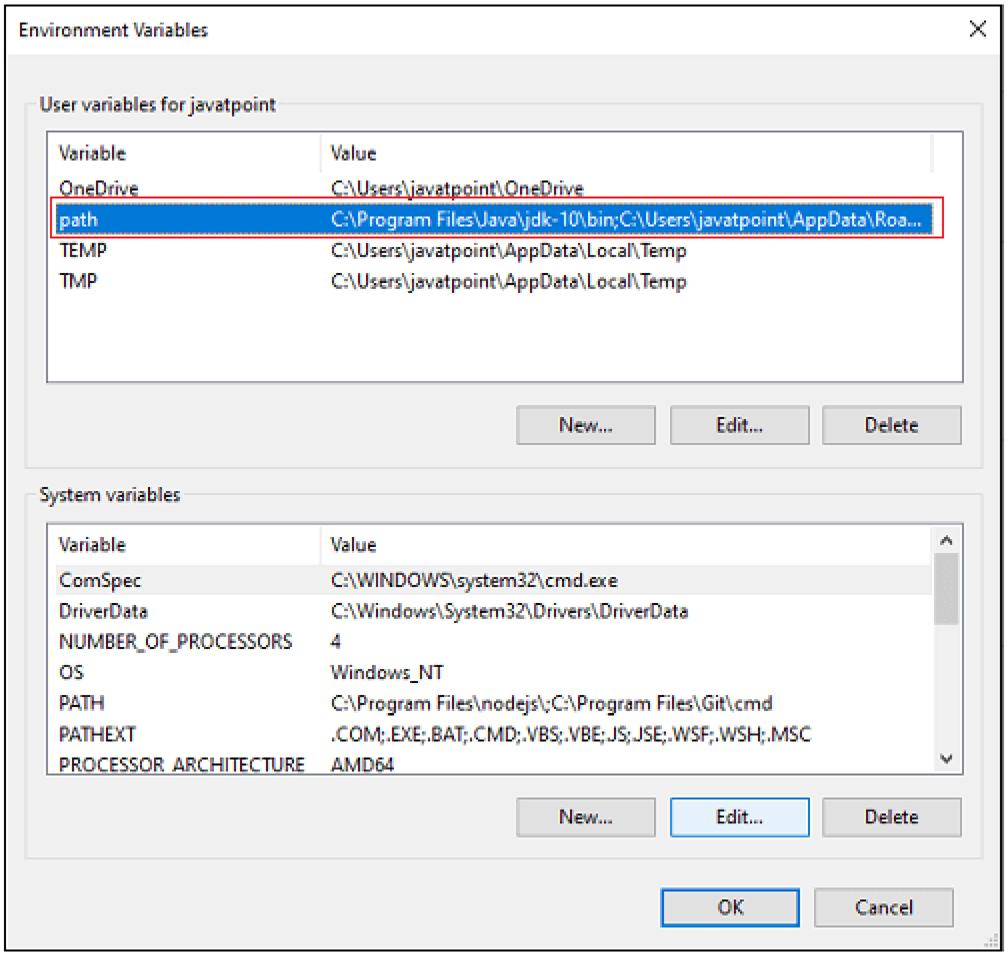
Step 2: Next, to download the latest Flutter SDK, click on the Windows **icon**. Here, you will find the download link for SDK.

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.

Note: ***The Flutter SDK should not be placed where the administrator's permission is required.***

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.


Environment Variables

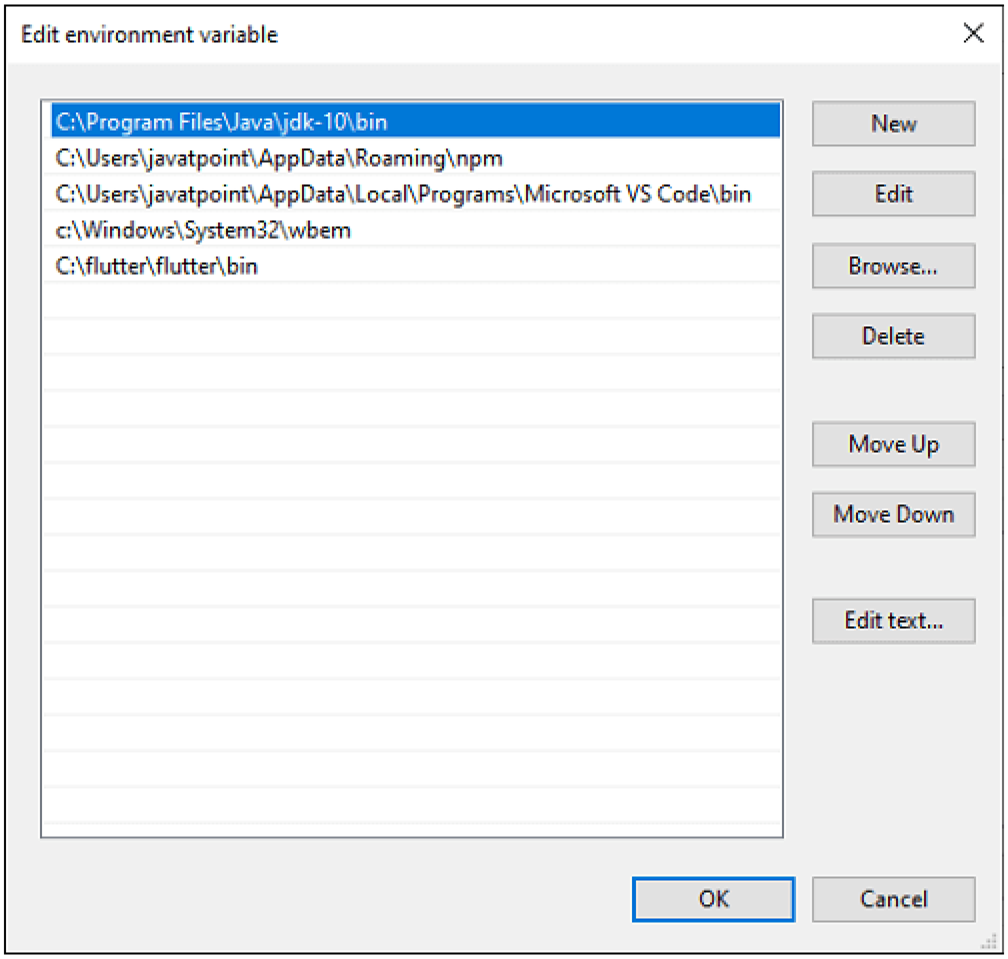
User variables for javatpoint

Variable	Value
OneDrive	C:\Users\javatpoint\OneDrive
path	C:\Program Files\Java\jdk-10\bin;C:\Users\javatpoint\AppData\Roa...
TEMP	C:\Users\javatpoint\AppData\Local\Temp
TMP	C:\Users\javatpoint\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	4
OS	Windows_NT
PATH	C:\Program Files\nodejs\;C:\Program Files\Git\cmd
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64

Step 4.2: Now, select path -> click on edit. The following screen appears.



Edit environment variable

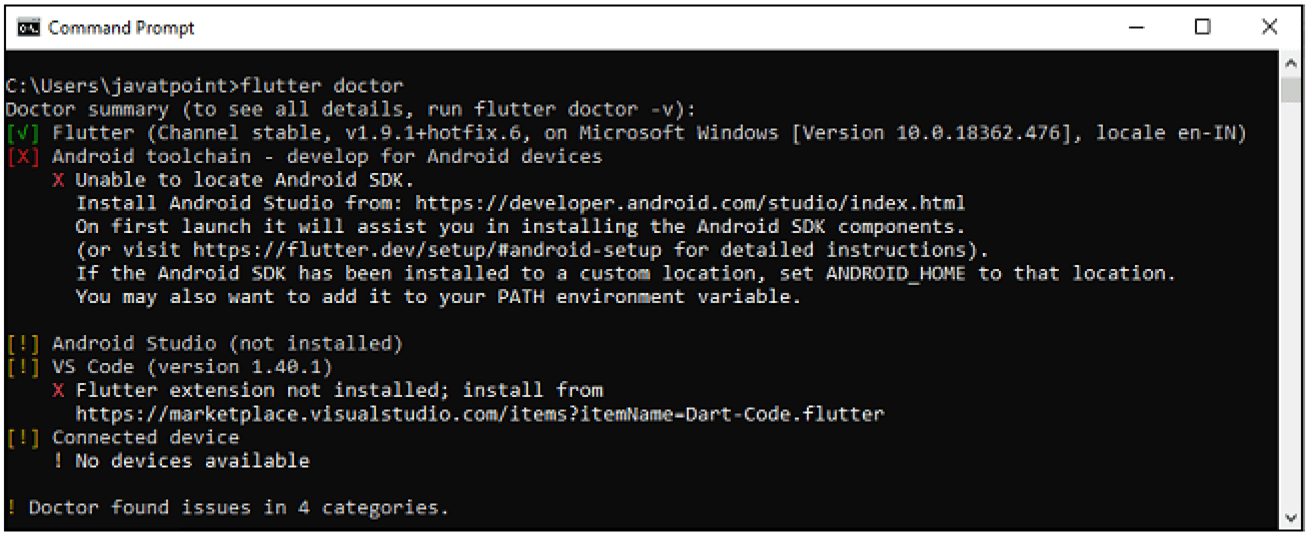
C:\Program Files\Java\jdk-10\bin
C:\Users\javatpoint\AppData\Roaming\npm
C:\Users\javatpoint\AppData\Local\Programs\Microsoft VS Code\bin
c:\Windows\System32\wbem
C:\flutter\flutter\bin

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.

1. $ 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.

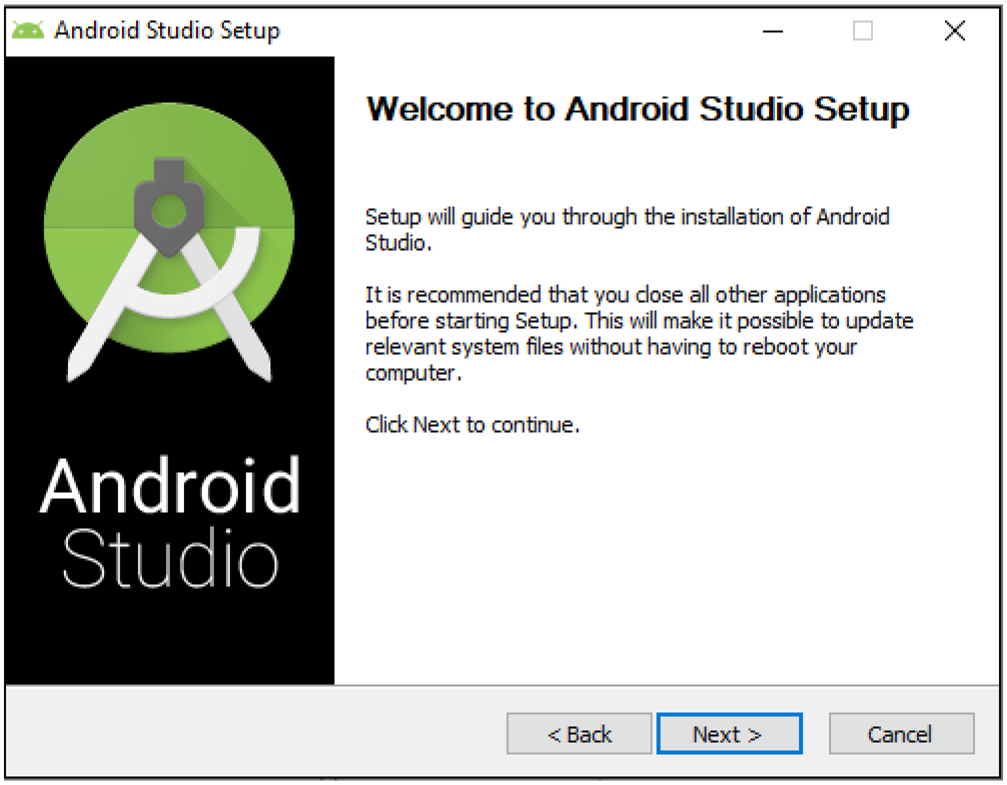


```
C:\Users\javatpoint>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, v1.9.1+hotfix.6, on Microsoft Windows [Version 10.0.18362.476], locale en-IN)
[X] Android toolchain - develop for Android devices
    X Unable to locate Android SDK.
      Install Android Studio from: https://developer.android.com/studio/index.html
      On first launch it will assist you in installing the Android SDK components.
      (or visit https://flutter.dev/setup/#android-setup for detailed instructions).
      If the Android SDK has been installed to a custom location, set ANDROID_HOME to that location.
      You may also want to add it to your PATH environment variable.

[!] Android Studio (not installed)
[!] VS Code (version 1.40.1)
    X Flutter extension not installed; install from
      https://marketplace.visualstudio.com/items?itemName=Dart-Code.flutter
[!] Connected device
    ! No devices available

! Doctor found issues in 4 categories.
```

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 official site.

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



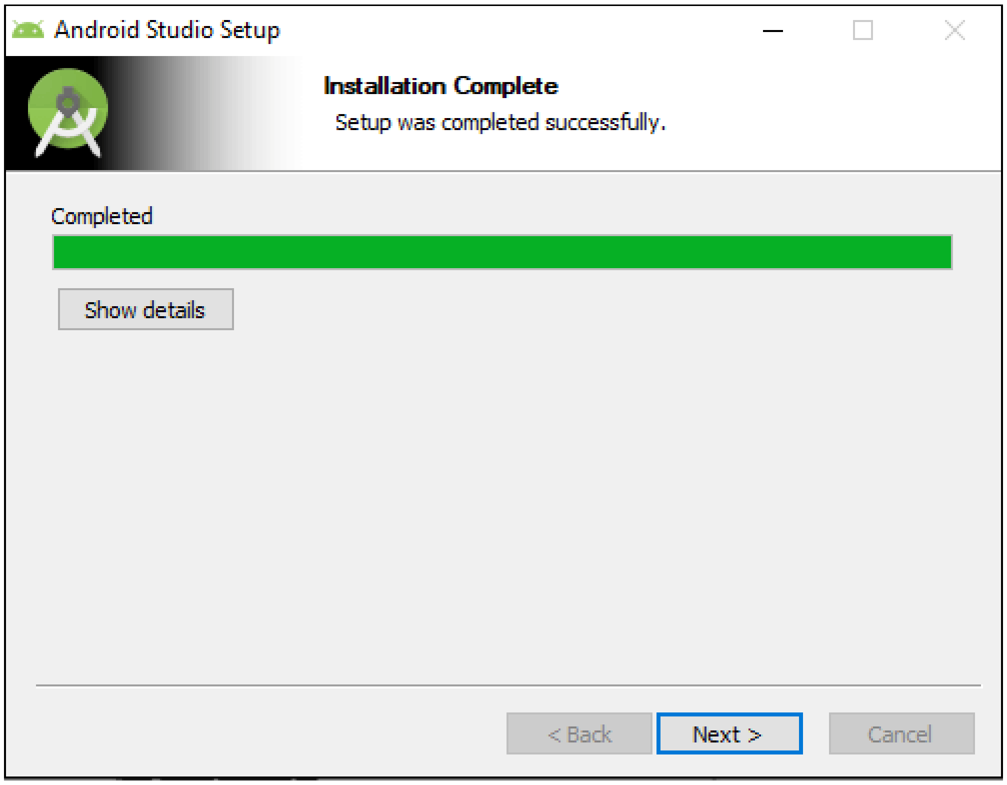
Welcome to Android Studio Setup

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.


Android Studio Setup

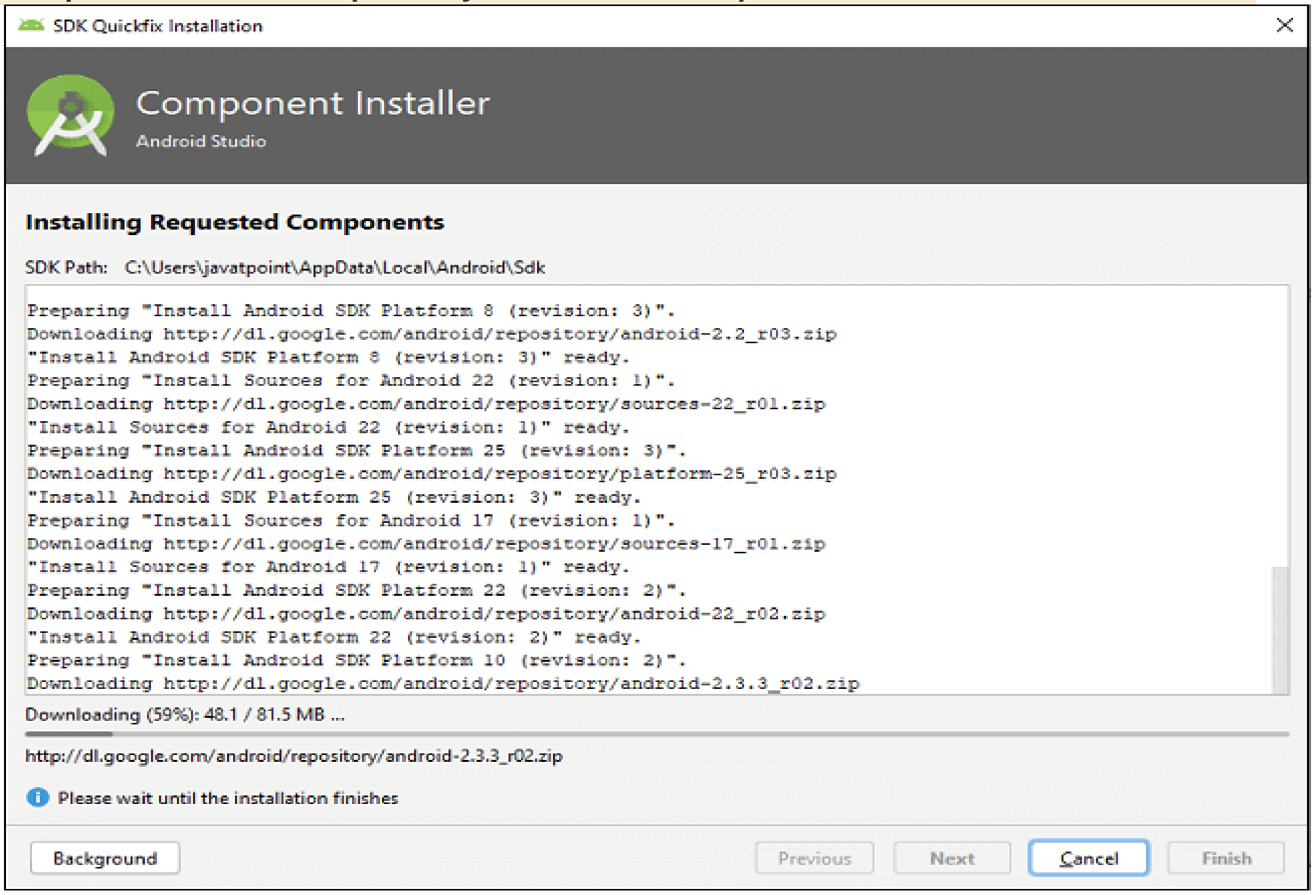
Installation Complete

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.

Note:> ***Meanwhile, the installation wizard also includes downloading Android SDK components that are required by Flutter for development.***



Installing Requested Components

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

```
Preparing "Install Android SDK Platform 8 (revision: 3)".
Downloading http://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 http://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 http://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 http://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 http://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 http://dl.google.com/android/repository/android-2.3.3_r02.zip
```

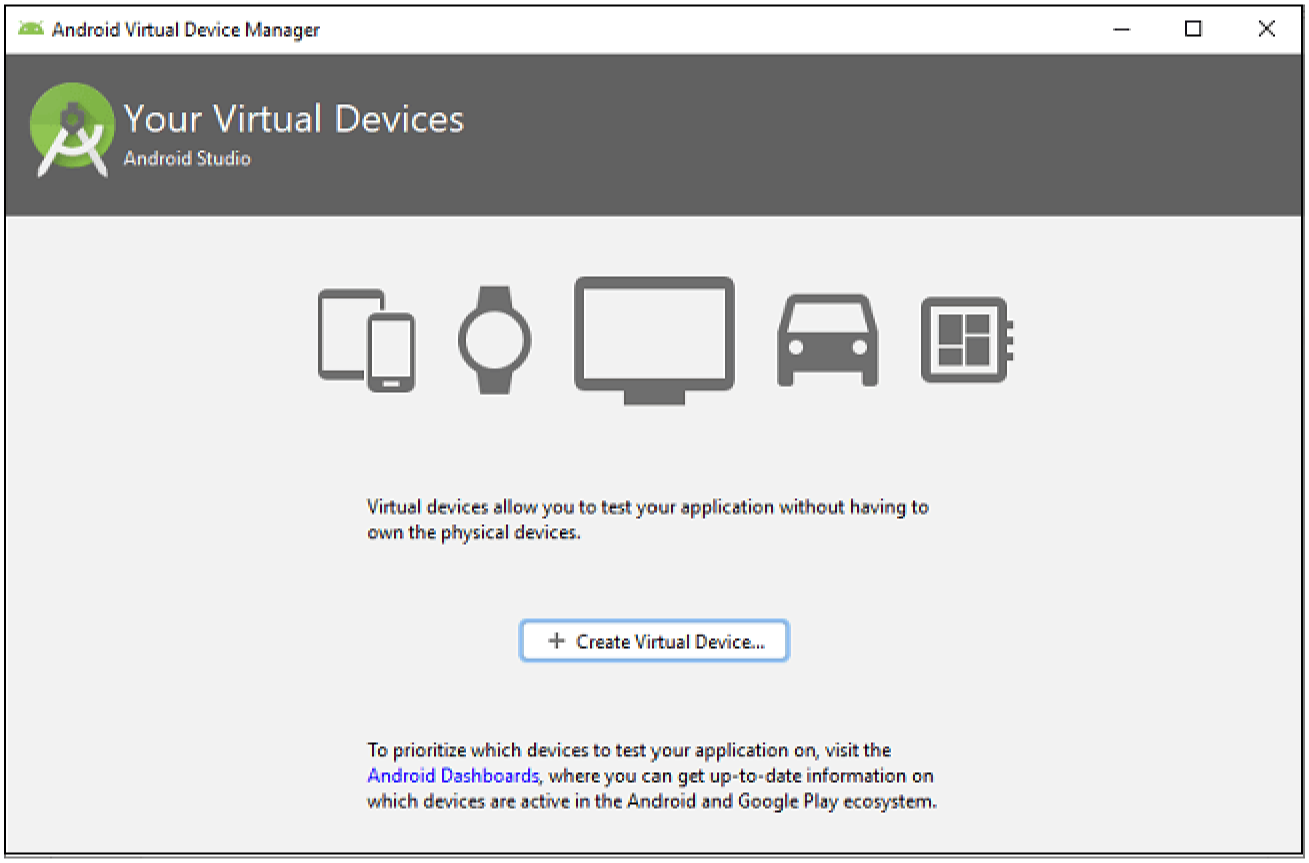
Downloading (59%): 48.1 / 81.5 MB ...

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.



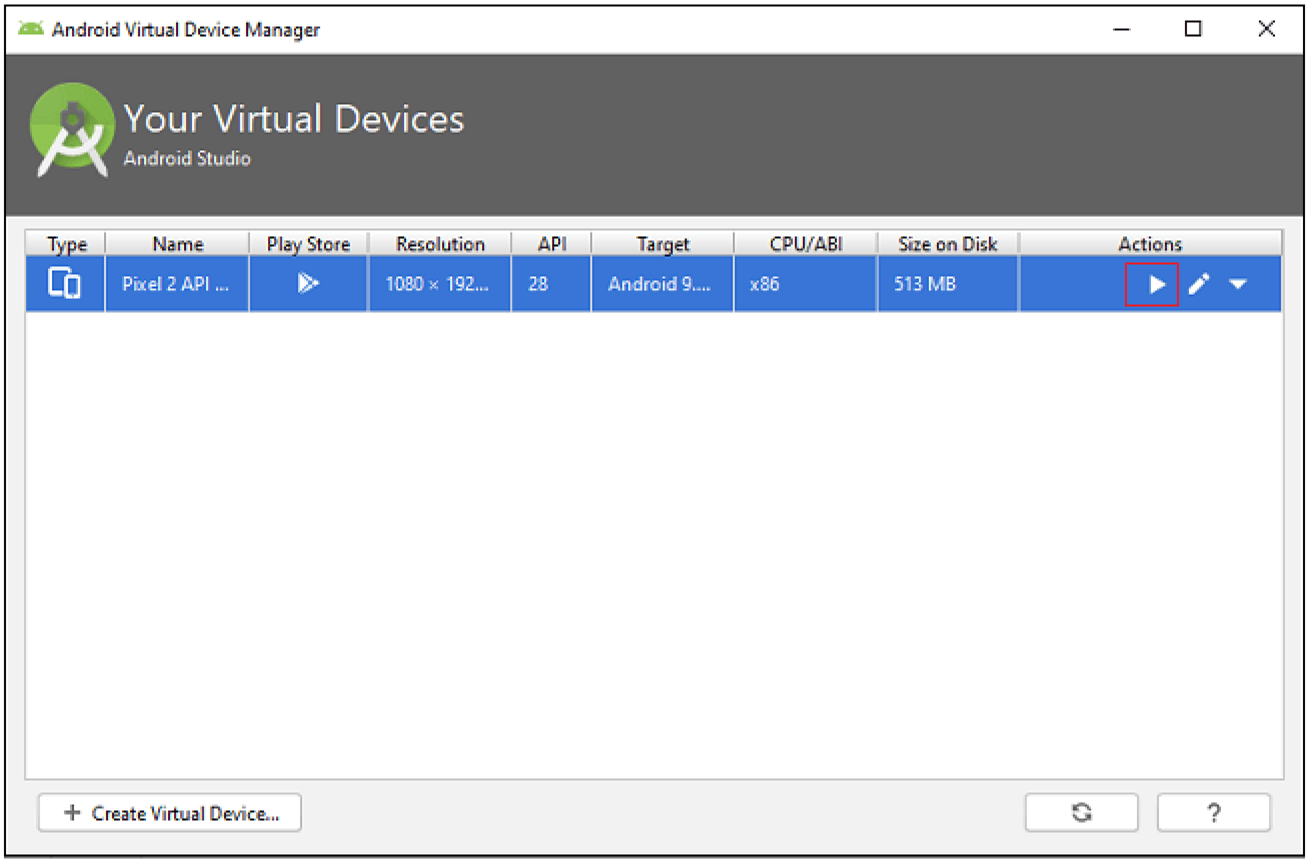
Virtual devices allow you to test your application without having to own the physical devices.

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.

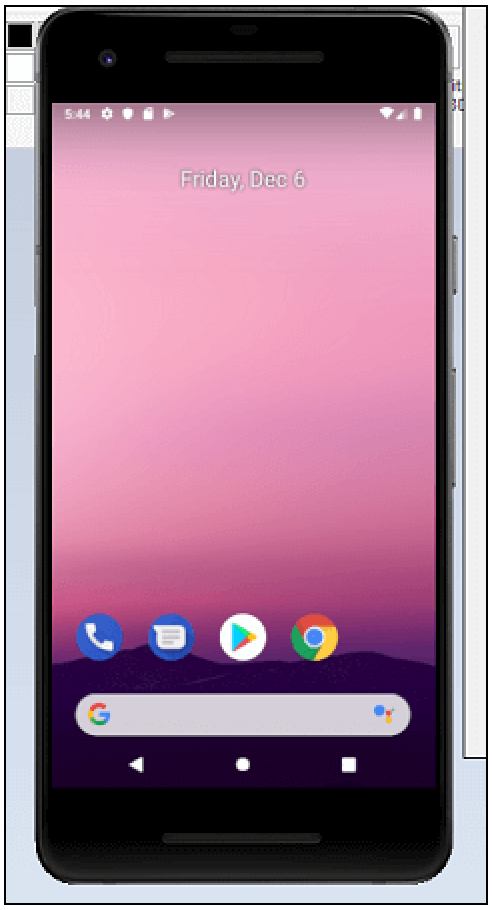

Android Virtual Device Manager

Your Virtual Devices

Android Studio

Type	Name	Play Store	Resolution	API	Target	CPU/ABI	Size on Disk	Actions
	Pixel 2 API ...		1080 × 192...	28	Android 9....	x86	513 MB	

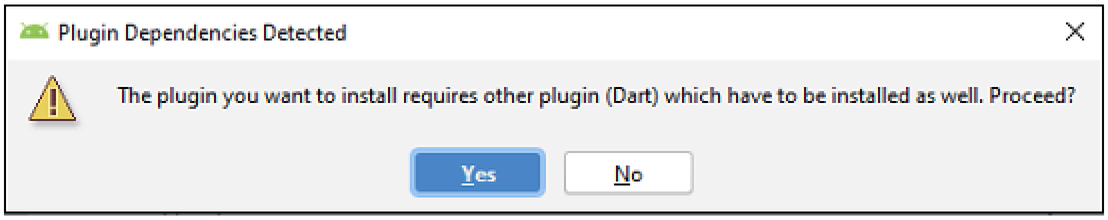
Step 8.5: Last, click on the icon pointed into the red color rectangle. The Android emulator displayed as below screen.



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.


Plugin Dependencies Detected

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.

System Requirements for macOS

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

Operating System	macOS (64-bit)
Disk Space	2.8 GB (It does not include disk space for IDE/tools).
Tools	bash curl git 2.x mkdir rm unzip which
IDE	Xcode (Official)

Get the Flutter SDK

Step 1: Download the installation bundle of the Flutter Software Development Kit for macOS. To download Flutter SDK, Go to its official website.

Step 2: When your download is complete, extract the zip file and place it in the desired installation folder or location.

Step 3: To run the Flutter command, you need to update the system path to include the flutter bin directory.

```
1. $ export PATH="$PATH:`pwd`/flutter/bin"
```

Step 4: Next, enable the updated path in the current terminal window using the below command and then verify it also.

```
1. source ~/.bashrc
2. source $HOME/.bash_profile
3. echo $PATH
```

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.

1. $ flutter doctor

Step 6: When you run the above command, it will analyze the system and 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 latest Xcode tools if reported by the Flutter doctor tool.
Step 8: Install the latest Android Studio and SDK, if reported by the Flutter doctor tool.
Step 9: Next, you need to set up an iOS simulator or connect an iPhone device to the system for developing an iOS application.
Step 10: Again, set up an android emulator or connect an android device to the system for developing an android application.
Step 11: 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.

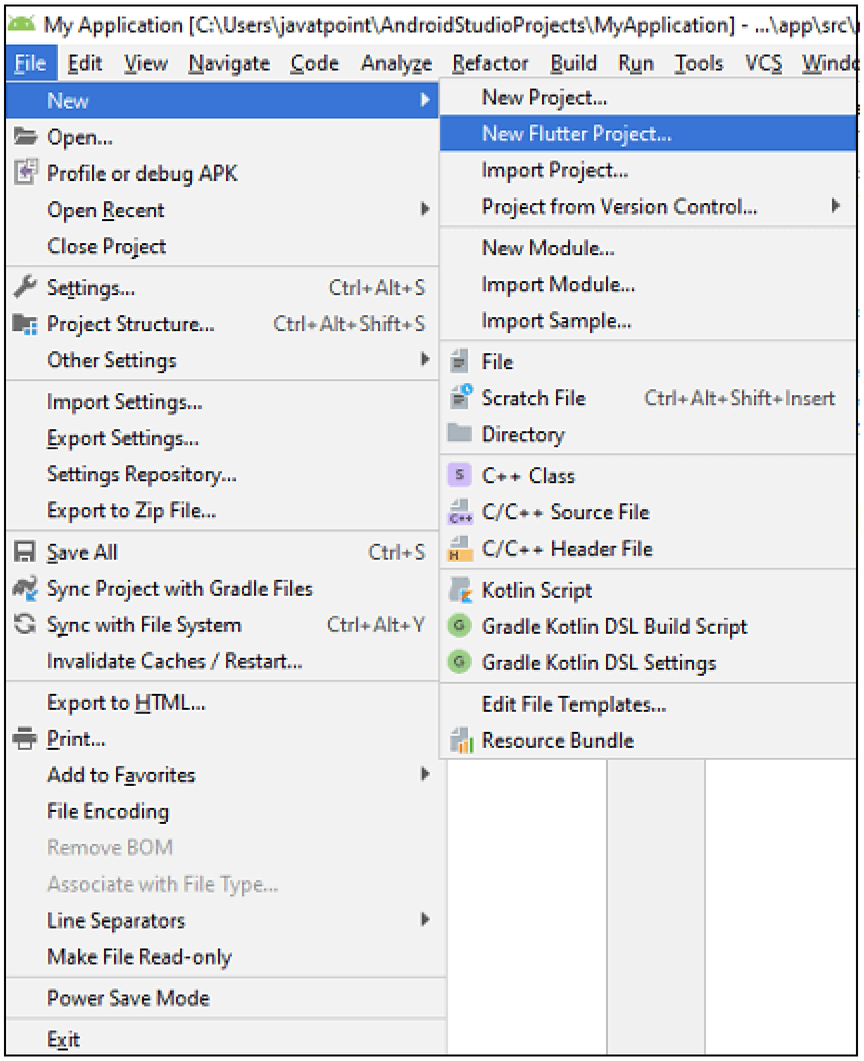
- It provides very limited access to SDK libraries. It means a developer does not have a lot of functionalities to create a mobile application. Such types of functionalities need to be developed by the Flutter developer themselves.
- The Flutter apps do not support the browser. It only supports Android and iOS platforms.
- It uses Dart programming for coding, so a developer needs to learn new technologies. However, it is easy to learn for developers.

Flutter First Application

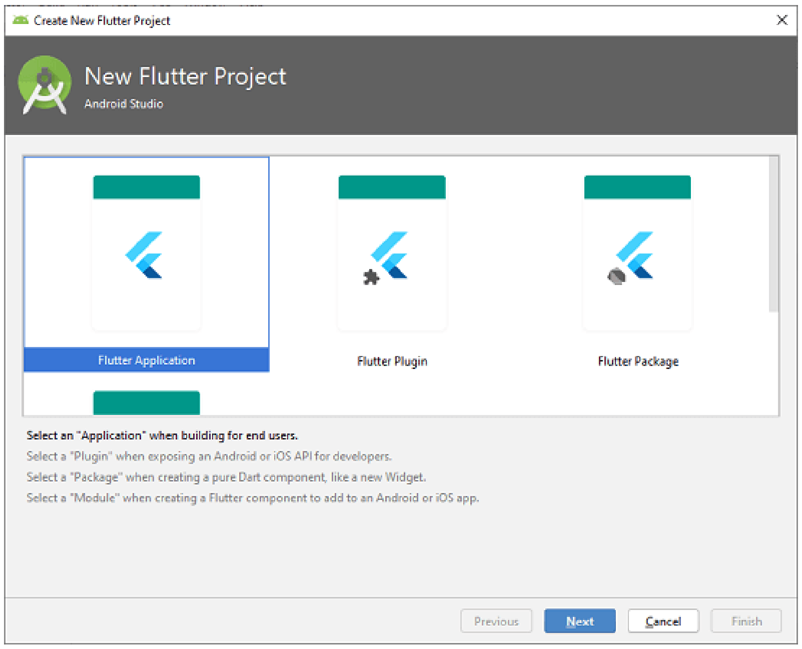
In this section, we are going to learn how to create a simple application in Android Studio to understand the basics of the Flutter application. To create Flutter application, do the following steps:

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.

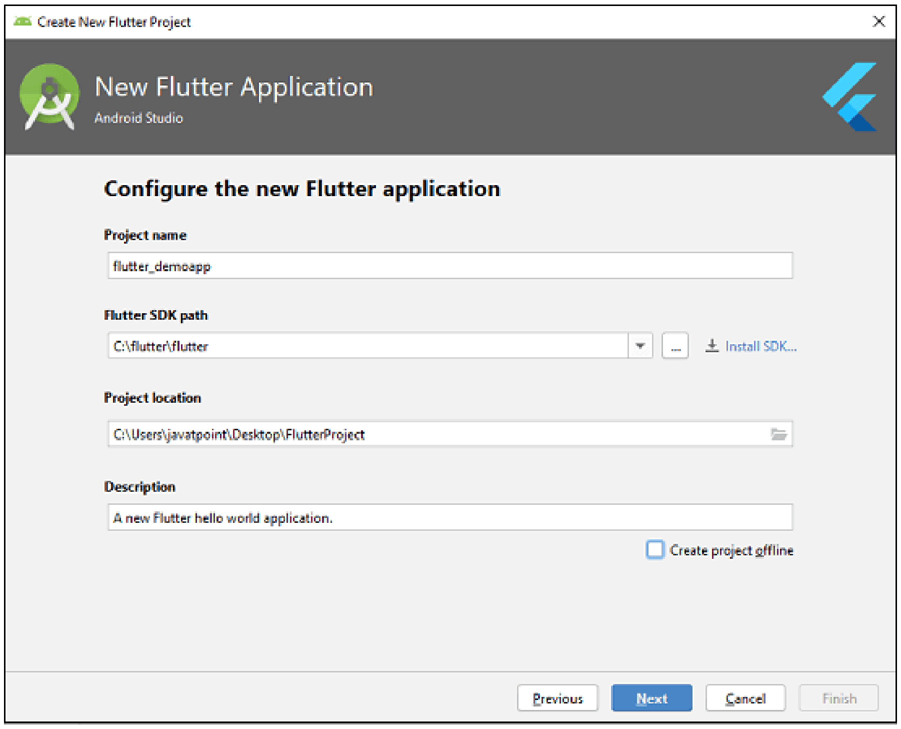
Step 4: Next, configure the application details as shown in the below screen and click on the Next button.

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

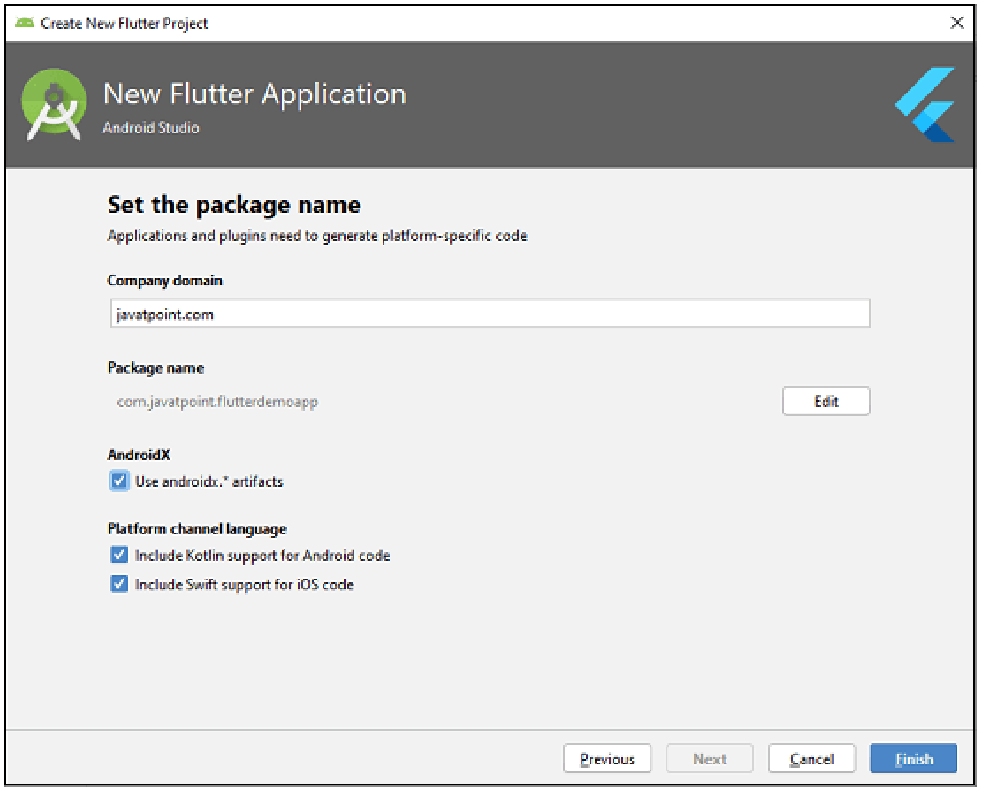
Flutter SDK path: C:\flutter\flutter

Project location: C:\Users\javatpoint\Desktop\FlutterProject

Description: A new Flutter hello world application.

☐ Create project offline

Step 5: In the next wizard, you need to set the company domain name and click the Finish button.



Set the package name

Applications and plugins need to generate platform-specific code

Company domain: javatpoint.com

Package name: com.javatpoint.flutterdemoapp

AndroidX

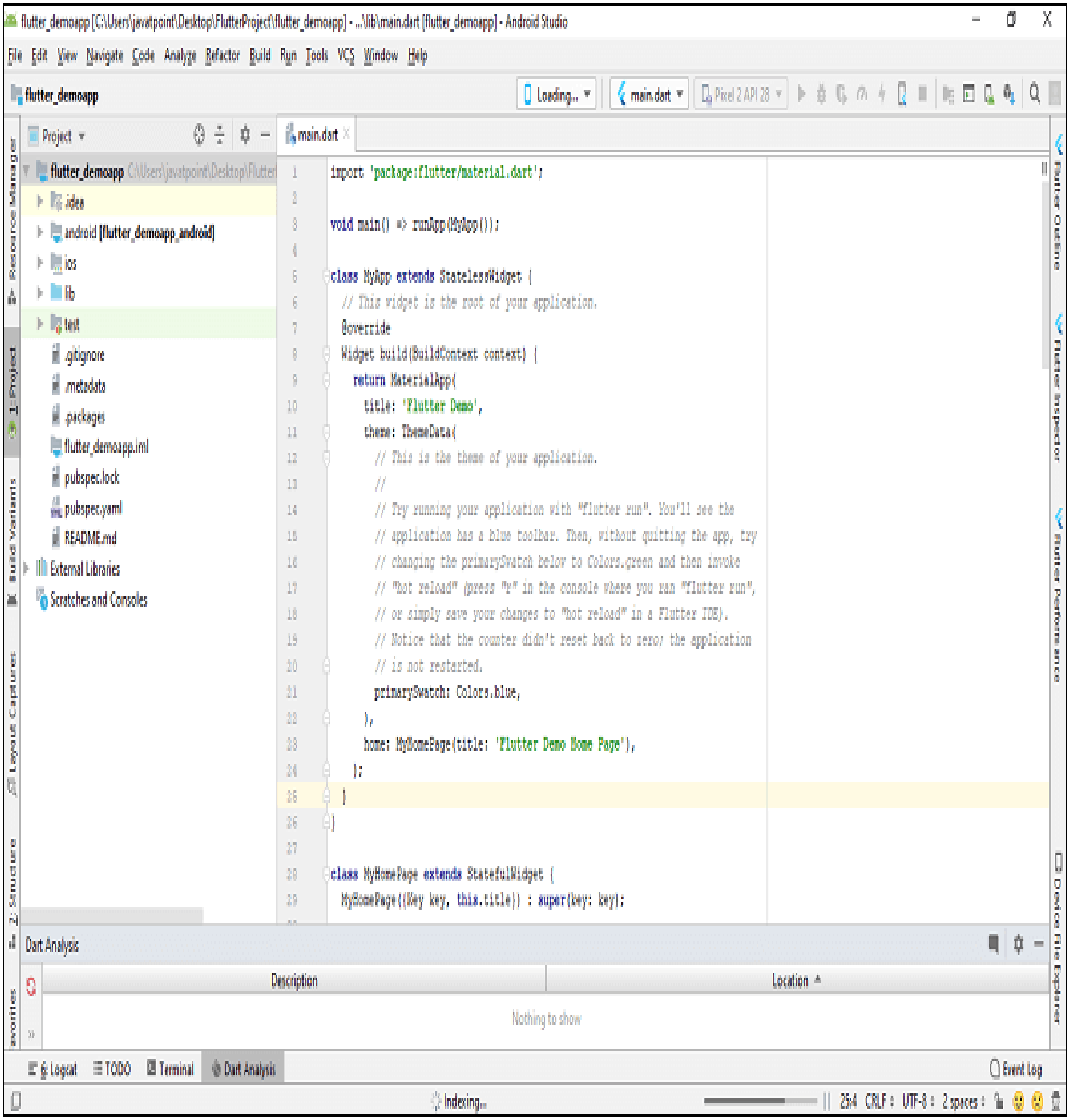
✓ Use androidx.* artifacts

Platform channel language

✓ Include Kotlin support for Android code

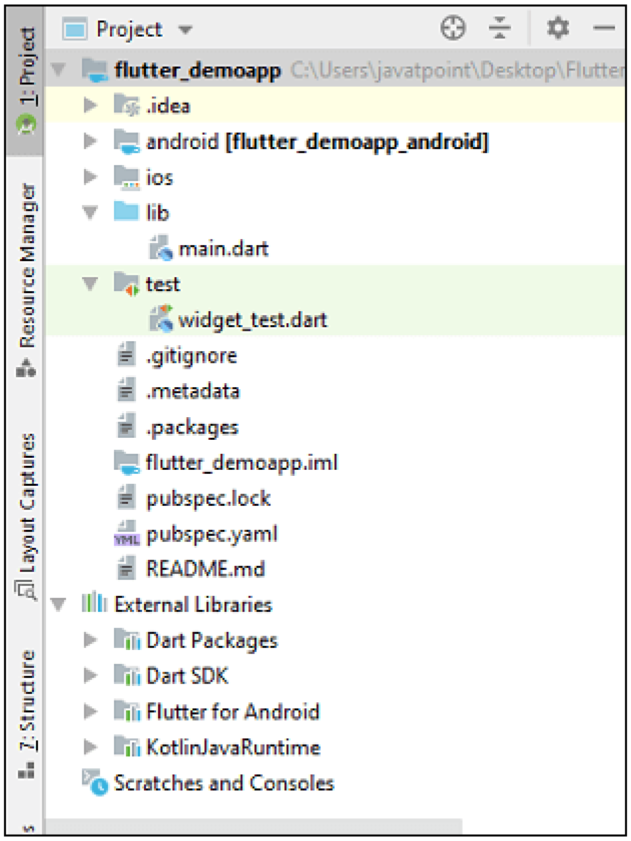
✓ Include Swift support for iOS code

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.



```dart
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(
      title: 'Flutter Demo',
      theme: ThemeData(
        // This is the theme of your application.
        //
        // Try running your application with "flutter run". You'll see the
        // application has a blue toolbar. Then, without quitting the app, try
        // changing the primarySwatch below to Colors.green and then invoke
        // "hot reload" (press "r" in the console where you ran "flutter run",
        // or simply save your changes to "hot reload" in a Flutter IDE).
        // Notice that the counter didn't reset back to zero; the application
        // is not restarted.
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

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

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: a Dart code, which is written for the Flutter application to perform the automated This folder contains 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.

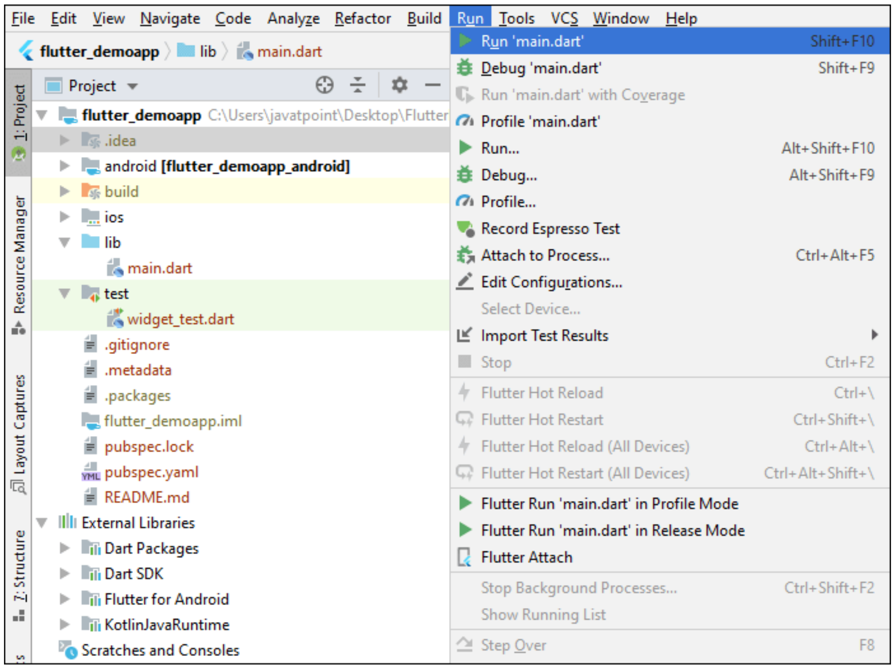
```
import 'package:flutter/material.dart';

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

```

```
class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Hello World Flutter Application',
      theme: ThemeData(
        // This is the theme of your application.
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Home page'),
    );
  }
}
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'),
      ),
    );
  }
}
```

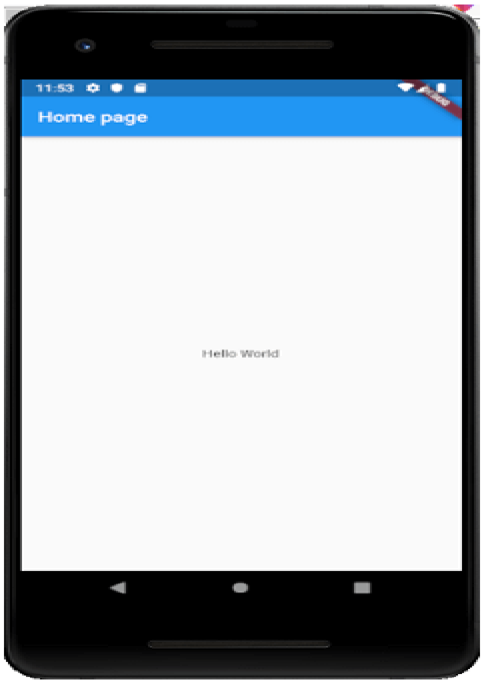
Step 8: Let us understand the above code snippet line by line.

- 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.


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
flutter_demoapp lib main.dart
Project
flutter_demoapp C:\Users\javatpoint\Desktop\Flutter
.idea
android [flutter_demoapp_android]
build
ios
lib
main.dart
test
widget_test.dart
.gitignore
.metadata
.packages
flutter_demoapp.iml
pubspec.lock
pubspec.yaml
README.md
External Libraries
Dart Packages
Dart SDK
Flutter for Android
KotlinJavaRuntime
Scratches and Consoles
1: Project
Resource Manager
Layout Captures
Z: Structure
Run 'main.dart' Shift+F10
Debug 'main.dart' Shift+F9
Run 'main.dart' with Coverage
Profile 'main.dart'
Run... Alt+Shift+F10
Debug... Alt+Shift+F9
Profile...
Record Espresso Test
Attach to Process... Ctrl+Alt+F5
Edit Configurations...
Select Device...
Import Test Results
Stop Ctrl+F2
Flutter Hot Reload Ctrl+\
Flutter Hot Restart Ctrl+Shift+\
Flutter Hot Reload (All Devices) Ctrl+Alt+\
Flutter Hot Restart (All Devices) Ctrl+Alt+Shift+\
Flutter Run 'main.dart' in Profile Mode
Flutter Run 'main.dart' in Release Mode
Flutter Attach
Stop Background Processes... Ctrl+Shift+F2
Show Running List
Step Over F8

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

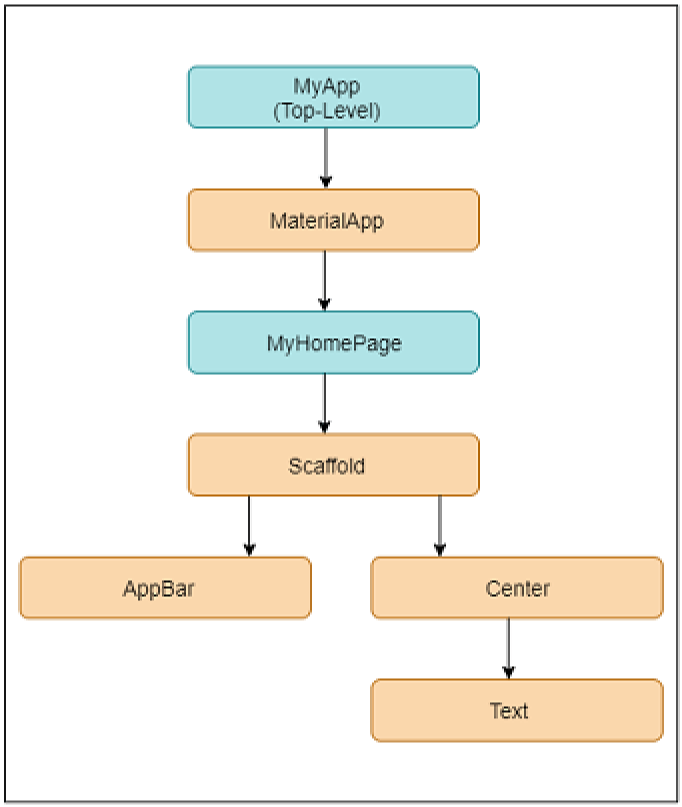
It contains all the required packages for the basic building blocks of writing a Flutter application. These libraries are written in Dart language.

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

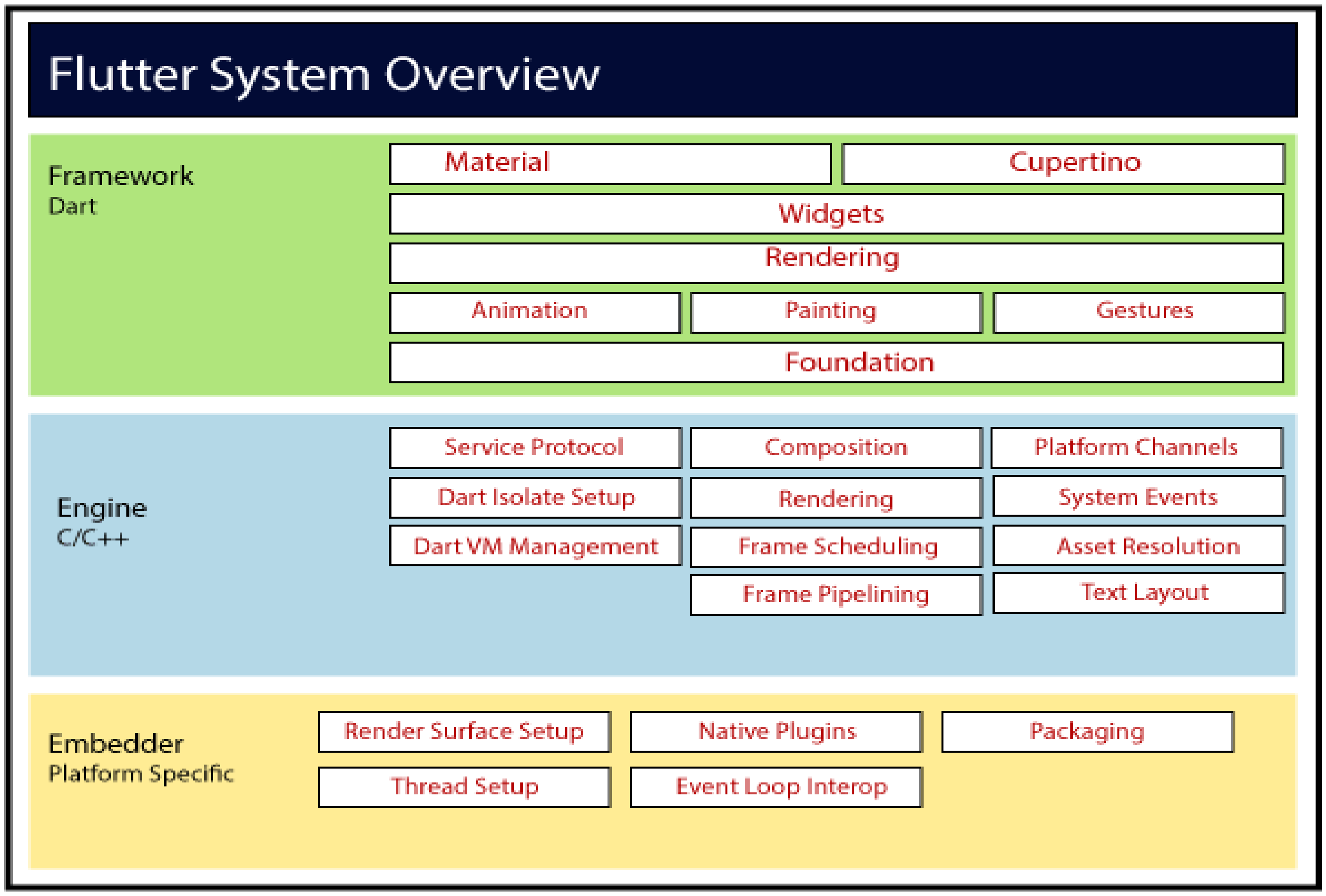
It is a widget that provides interaction (how to listen for and respond to) in Flutter using GestureDetector. **GestureDector** is an invisible widget, which includes tapping, dragging, and scaling interaction of its child widget. We can also use other interactive features into the existing widgets by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

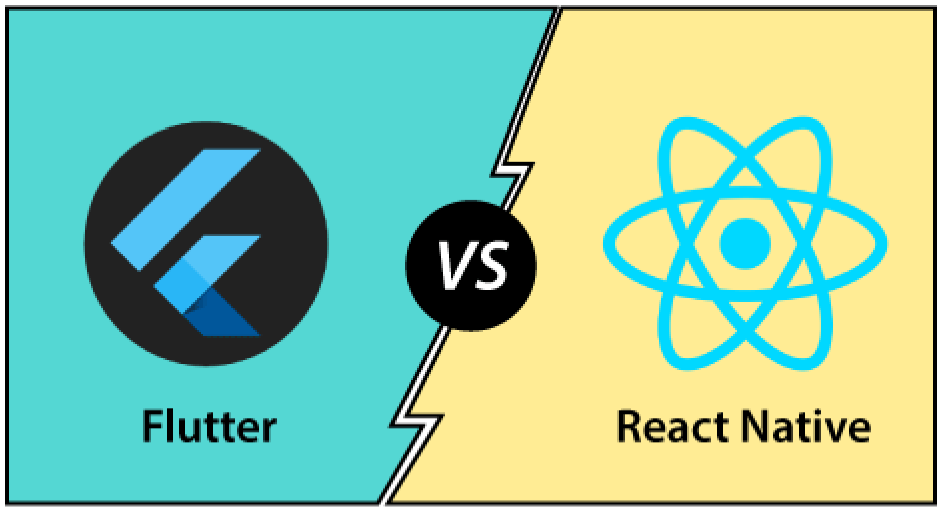
Layers

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.



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.

Let us understand the essential differences between Flutter and React Native with the following comparison chart.



Concept	Flutter	React Native
Develop By	It is first introduced by Google.	It is first introduced by Facebook.
Release	May 2017	June 2015
Programming Language	It uses **Dart** language to create a mobile app.	It uses **JavaScript** to create mobile apps.
Architecture	Flutter uses Business Logic Component (BLoC) architecture.	React Native uses Flux and Redux architecture. Flux created by Facebook, whereas Redux is the

		preferred choice among the community.
User Interface	It uses custom widgets to build the UI of the app.	It uses native UI controllers to create UI of the app.
Documentation	Flutter documentation is good, organize, and more informative. We can get everything that we want to be written in one place.	React native documentation is user-friendly but disorganized.
Performance	The performance of the Flutter application is fast. Flutter compiles the application by using the arm C/C++ library that makes it closer to machine code and gives the app a better native performance.	The performance of the React Native app is slow in comparison to the Flutter app. Here, sometimes developers face issues while running the hybrid application architecture.
Testing	Flutter provides a very rich set of testing features. This feature allows the developer to perform unit testing, integration testing, and widget testing.	React Native uses third-party tools that are available for testing the app.
Community Support	It has less community support as compared to React Native.	It has very strong community support where the questions and issues can be solved quickly.
Hot Reload	Supported	Supported
Popularity	81200 stars on GitHub (December 2019)	83200 stars on GitHub (December 2019)
Latest Version	Flutter-v1.12.13	React Native-v0.61.0

Industry Adoption	Google Ads Hamilton Reflectly Xianyu	Facebook Instagram LinkedIn Skype

Flutter Widgets

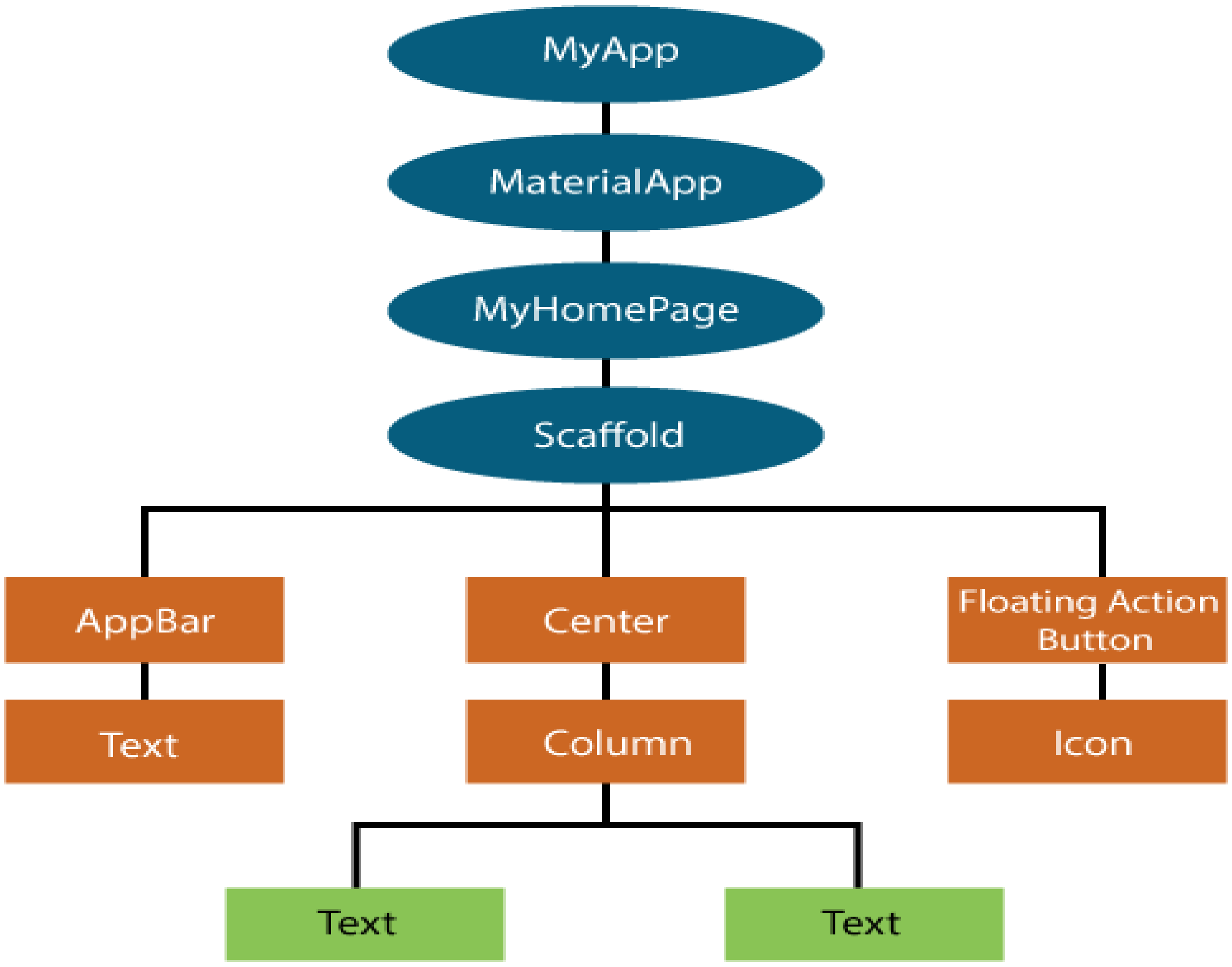
In this section, we are going to learn the concept of a widget, how to create it, and their different types available in the Flutter framework. We have learned earlier that everything in Flutter is a widget.

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

Types of Widget

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.

```
const Text('Hello, Wolrd!',
  textAlign: TextAlign.left,
  style: 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.

```
1. //FlatButton Example
   TextButton(
     style: TextButton.styleFrom(
       primary: Colors.red,
     ),
     onPressed: () { },
     child: Text('TextButton'),
   ),
```

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:** I
- t 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.

```
1. assets:
2.   - assets/
```

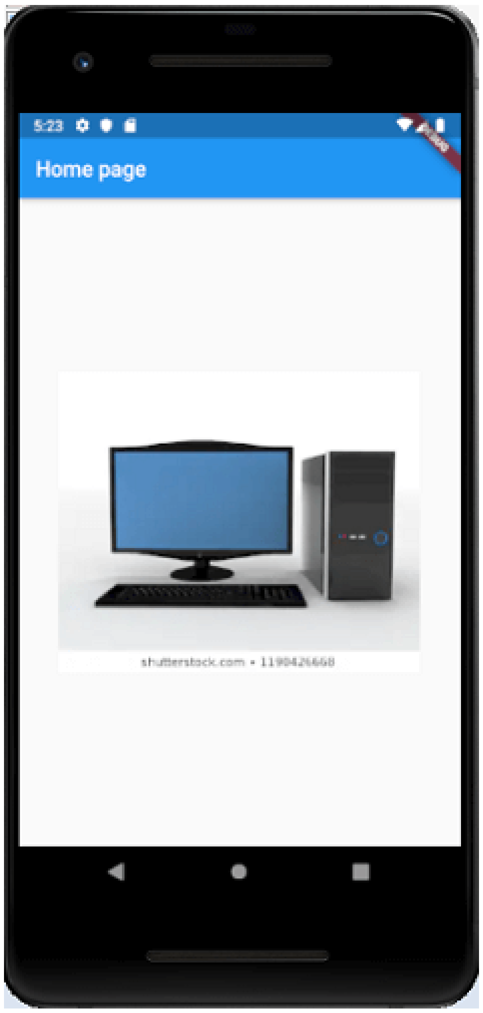
Now, add the following line in the dart file.

```
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.

```
const Icon(Icons.accessibility_new,size: 30,),
```

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.

```
Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    Text(
      "VegElement",
    ),
    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.

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: <Widget>[
    Text(
      "VegElement",
    ),
    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:  column(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: <Widget>[
      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:  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.

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));
  }
}
```

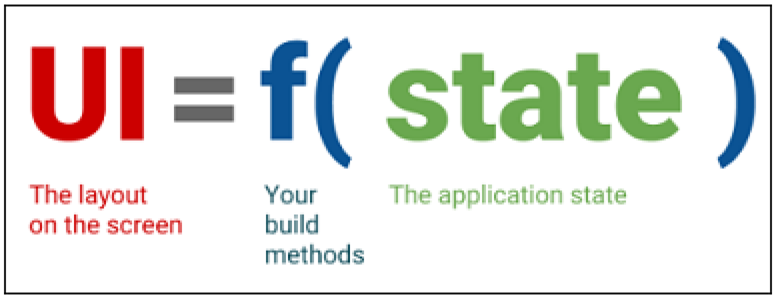
Flutter State Management

In this section, we are going to discuss state management and how we can handle it in the Flutter. We know that in Flutter, everything is a widget. The widget can be classified into two categories, one is a **Stateless widget,** and another is a **Stateful widget.** The Stateless widget does not have any internal state. It means once it is built, we cannot change or modify it until they are initialized again. On the other hand, a Stateful widget is dynamic and has a state. It means we can modify it easily throughout its lifecycle without reinitialized it again.

What is State?

A state is information that can be **read** when the widget is built and might **change or modified** over a lifetime of the app. If you want to change your widget, you need to update the state object, which can be done by using the setState() function available for Stateful widgets. The **setState()** function allows us to set the properties of the state **object** that triggers a redraw of the UI.

The state management is one of the most popular and necessary processes in the lifecycle of an application. According to official documentation, Flutter is declarative. It means Flutter builds its UI by reflecting the current state of your app. The following figure explains it more clearly where you can build a UI from the application state.



Let us take a simple example to understand the concept of state management. Suppose you have created a list of customers or products in your app. Now, assume you have added a new customer or product dynamically in that list. Then, there is a need to refresh the list to view the newly added item into the record. Thus, whenever you add a new item, you need to refresh the list. This type of programming requires state management to handle such a situation to improve performance. It is because every time you make a change or update the same, the state gets refreshed.

In Flutter, the state management categorizes into two conceptual types, which are given below:

1. Ephemeral State
2. App State

Ephemeral State

This state is also known as UI State or local state. It is a type of state which is related to the **specific widget,** or you can say that it is a state that contains in a single widget. In this kind of state, you do not need to use state management techniques. The common example of this state is **Text Field**.

Example

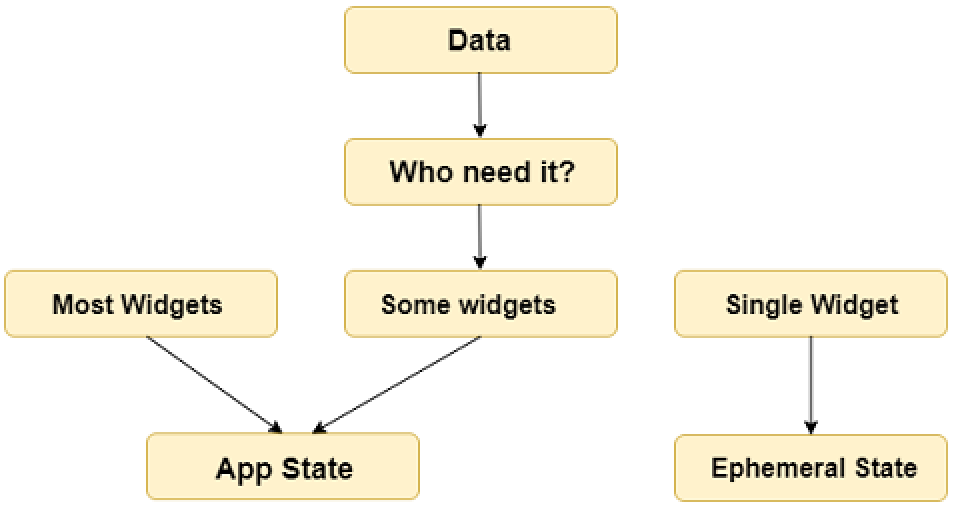
```
class MyHomepage extends StatefulWidget {
  @override
  MyHomepageState createState() => MyHomepageState();
}

class MyHomepageState extends State<MyHomepage> {
  String _name = "Peter";

  @override
  Widget build(BuildContext context) {
    return RaisedButton(
        child: Text(_name),
        onPressed: () {
          setState(() {
              _name = _name == "Peter" ? "John" : "Peter";
          });
        },
      );
  }
}
```

In the above example, the **_name** is an ephemeral state. Here, only the setState() function inside the StatefulWidget's class can access the _name. The build method calls a setState() function, which does the modification in the state variables. When this method is executed, the widget object is replaced with the new one, which gives the modified variable value.

App State

It is different from the ephemeral state. It is a type of state that we want to **share** across various parts of our app and want to keep between user sessions. Thus, this type of state can be used globally. Sometimes it is also known as application state or shared state. Some of the examples of this state are User preferences, Login info, notifications in a social networking app, the shopping cart in an e-commerce app, read/unread state of articles in a news app, etc.

The following diagram explains the difference between the ephemeral state and the app state more appropriately.



The simplest example of app state management can be learned by using the **provider package.** The state management with the provider is easy to understand and requires less coding. A provider is a **third-party** library. Here, we need to understand three main concepts to use this library.

1. ChangeNotifier
2. ChangeNotifierProvider
3. Consumer

ChangeNotifier

ChangeNotifier is a simple class, which provides change notification to its listeners. It is easy to understand, implement, and optimized for a small number of listeners. It is used for the listener to observe a model for changes. In this, we only use the **notifyListener()** method to inform the listeners.

For example, let us define a model based on ChangeNotifier. In this model, the **Counter** is extended with ChangeNotifier, which is used to notify its listeners when we call notifyListeners(). It is the only method that needs to implement in a ChangeNotifier model. In this example, we declared two functions the **increment** and **decrement,** which are used to increase and decrease the value. We can call notifyListeners() method any time the model changes in a way that might change your app's UI.

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

class Counter with ChangeNotifier {
  int _counter;

  Counter(this._counter);

  getCounter() => _counter;
  setCounter(int counter) => _counter = counter;

  void increment() {
    _counter++;
    notifyListeners();
  }

  void decrement() {
    _counter--;
    notifyListeners();
  }
}
```

ChangeNotifierProvider

ChangeNotifierProvider is the widget that provides an **instance** of a ChangeNotifier to its descendants. It comes from the provider package. The following code snippets help to understand the concept of ChangeNotifierProvider.

Here, we have defined a **builder** who will create a new instance of the **Counter** model. ChangeNotifierProvider does not rebuild Counter unless there is a need for this. It will also automatically call the **dispose()** method on the Counter model when the instance is no longer needed.

```
class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   theme: ThemeData(
    primarySwatch: Colors.indigo,
   ),
   home: ChangeNotifierProvider<CounterModel>(
    builder: (_) => CounterModel(),
    child: CounterView(),
   ),
  );
 }
}
```

If there is a need to provide more than one class, you can use **MultiProvider.** The MultiProvider is a list of all the different Providers being used within its scope. Without using this, we would have to nest our Providers with one being the child of another and another. We can understand this from the below code.

```
void main() {
 runApp(
  MultiProvider(
   providers: [
    ChangeNotifierProvider(builder: (context) => Counter()),
    Provider(builder: (context) => SomeOtherClass()),
   ],
   child: MyApp(),
  ),
 );
}
```

Consumer

It is a type of provider that does not do any fancy work. It just calls the provider in a new widget and delegates its build implementation to the builder. The following code explains it more clearly./p>

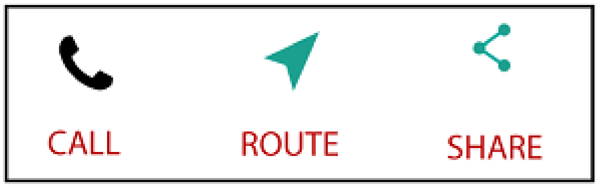
```
return Consumer<Counter>(
  builder: (context, count, child) {
    return Text("Total price: ${count.total}");
  },
);
```

In the above example, you can see that the **consumer widget** only requires a builder function, which is called whenever the ChangeNotifier changes. The builder function contains **three** arguments, which are **context, count, and child.** The first argument, context, contain in every build() method. The second argument is the instance of the ChangeNotifier, and the third argument is the child that is used for optimization. It is the best idea to put the consumer widget as deep as in the tree as possible.

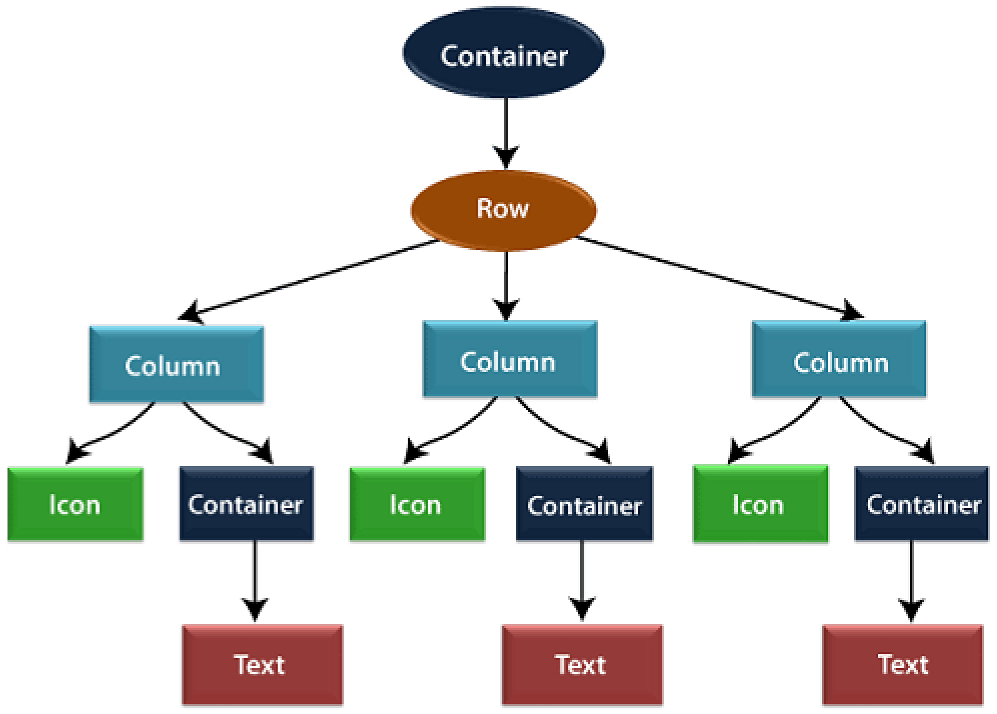
Flutter Layouts

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(10.0),
    color: Colors.amber[600],
    width: 48.0,
    height: 48.0,
  ),)
```

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 World!'),
  ),
) ,
```

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 World!')),
)
```

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 Widget

ConstrainedBox is a built-in widget in flutter SDK. Its function is to add size constraints to its child widgets. It comes quite handy if we want a container or image to not exceed a certain height and width. It is also good to keep text in a wrapped layout by making the *Text* widget a child on *ConstrainedBox*. This functionality can also be found in SizedBox widget or else.

Constructor of ConstrainedBox Class:

```
ConstrainedBox(

{Key key,

@required BoxConstraints constraints,

Widget child}

)
```

Property of ConstrianedBox Widget:

- **constraints:** This property takes in the *BoxConstrain* Class as the object. It puts constraints it's child widget using the functions of the *BoxConstraints* class.

Example 1:

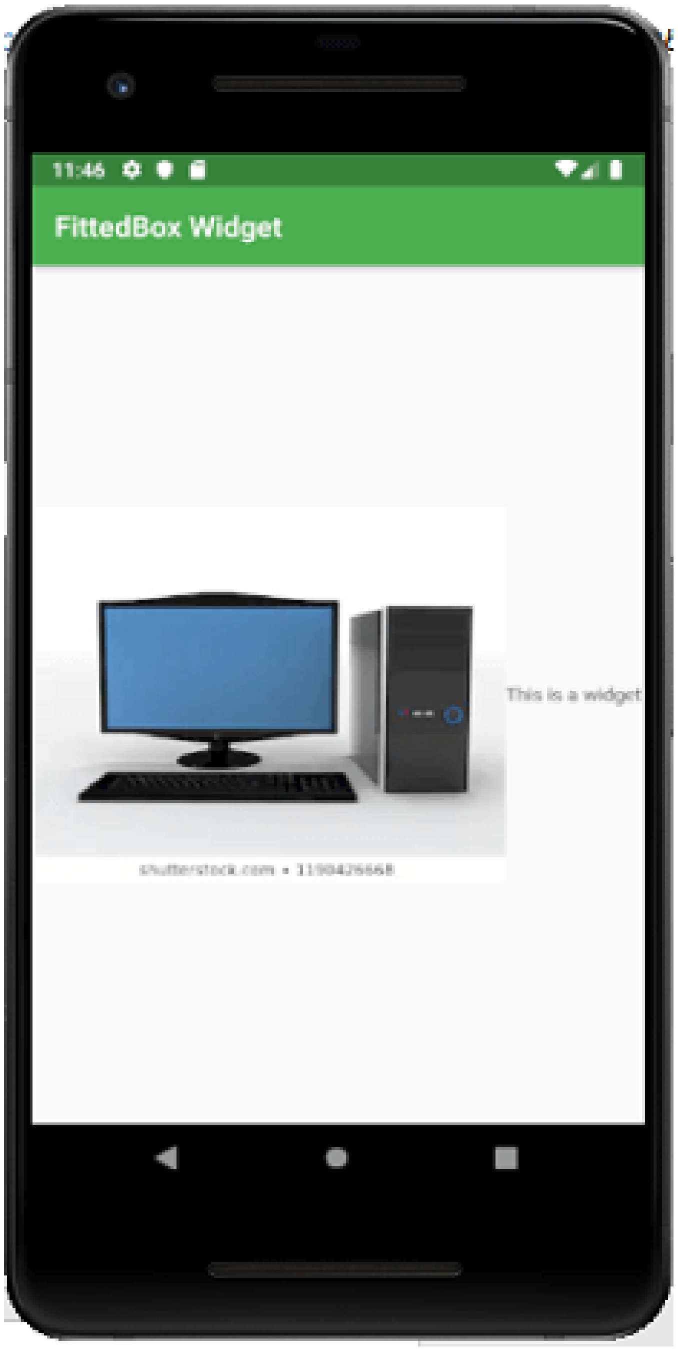
```
 body: Center(
          /** ConstrainedBox Widget **/
          child: ConstrainedBox(
            constraints: BoxConstraints.expand(height: 200, width: 200),
            child: Container(
              color: Colors.green,
            ), //Container widget
          ), //ConstrainedBox
        ), //Center
```

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,
        )
      ),
    );
  }
}
```

Output



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.

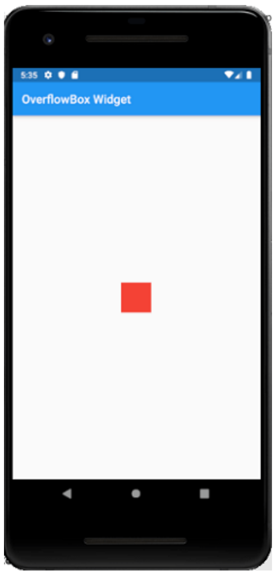
Offstage: It is used to measure the dimensions of a widget without bringing it on to the screen.

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

Output



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.

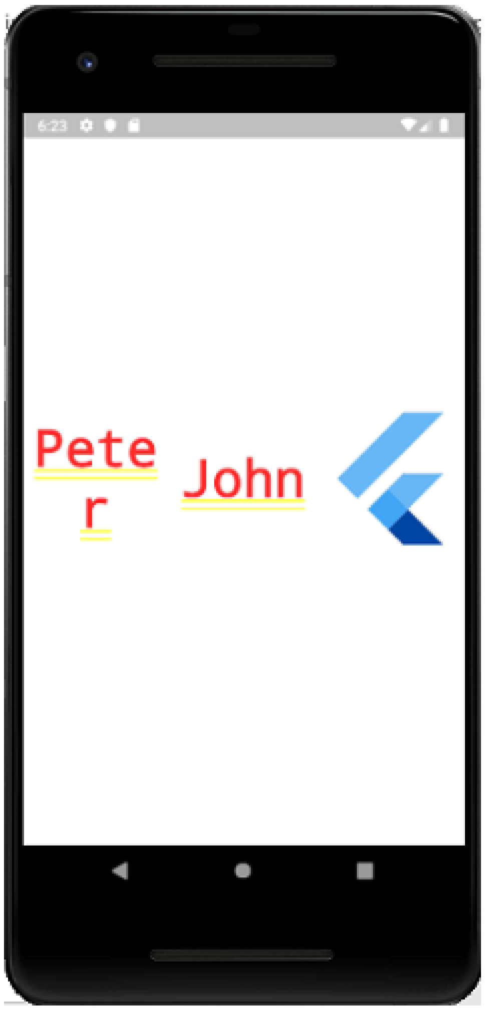
Here, we are going to learn different types of multiple child widgets:

Row: It allows to arrange 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(
      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(),
         ),  ),   ],  ),  ),  ); } }
```

Output



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.

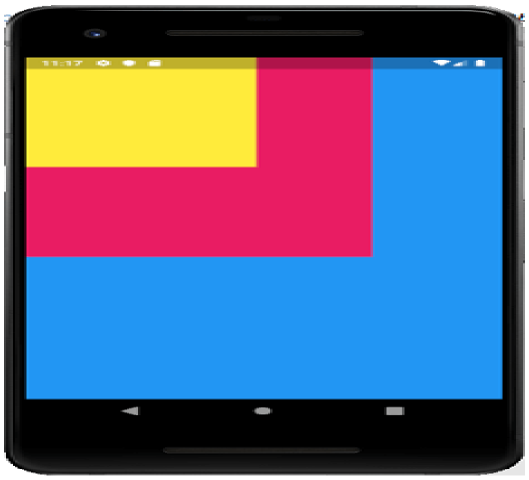
Table: It is a widget that allows us to arrange its children in a table based widget.

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. The following example helps to understand it.

```
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.blue,
     ),
    home: MyHomePage(),
   );
 }
}
class MyHomePage extends StatelessWidget {
  @override
 Widget build(BuildContext context) {
   return Center(
    child: Container(
      alignment: Alignment.center,
      color: Colors.white,
      child: Stack(
children: <Widget>[
  // Max Size
 Container(
   color: Colors.blue,
 ),
  Container(
   color: Colors.pink,
    height: 400.0,
   width: 300.0,
  ),
 Container(
   color: Colors.yellow,
   height: 220.0,
    width: 200.0,
 ) , ],  ),),  );  }  }
```

Output



Flutter Scaffold

The Scaffold is a widget in Flutter used to implements the basic material **design visual layout structure**. It is quick enough to create a general-purpose mobile application and contains almost everything we need to create a functional and responsive Flutter apps. This widget is able to occupy the whole device screen. In other words, we can say that it is mainly responsible for creating a base to the app screen on which the child widgets hold on and render on the screen. It provides many widgets or APIs for showing Drawer, SnackBar, BottomNavigationBar, AppBar, FloatingActionButton, and many more.

The Scaffold class is a shortcut to set up the look and design of our app that allows us not to build the individual visual elements manually. It saves our time to write more code for the look and feel of the app. The following are the **constructor and properties** of the Scaffold widget class.

```
const Scaffold({
  Key key,
  this.appBar,
  this.body,
  this.floatingActionButton,
  this.floatingActionButtonLocation,
  this.persistentFooterButtons,
  this.drawer,
  this.endDrawer,
  this.bottomNavigationBar,
  this.bottomSheet,
  this.floatingActionButtonAnimator,
  this.backgroundColor,
  this.resizeToAvoidBottomPadding = true,
  this.primary = true,
})
```

Let us understand all of the above properties in detail.

1. appBar: It is a **horizontal bar** that is mainly displayed at the **top** of the Scaffold widget. It is the main part of the Scaffold widget and displays at the top of the screen. Without this property, the Scaffold widget is incomplete. It uses the appBar widget that itself contains various properties like elevation, title, brightness, etc. See the below example:

```
Widget build(BuildContext context)
{
 return Scaffold(
   appBar: AppBar(
     title: Text('First Flutter Application'),
   ), )
}
```

In the above code, the title property uses a **Text widget** for displaying the text on the screen.

2. body: It is the other primary and required property of this widget, which will **display the main content in the Scaffold**. It signifies the place below the appBar and behind the floatingActionButton & drawer. The widgets inside the body are positioned at the top-left of the available space by default. See the below code:

```
  body: Center(
      child: Text("Welcome to GeeksforGeeks!!!",
        style: TextStyle(
          color: Colors.black,
          fontSize: 40.0,
        ),
      ),
```

In the above code, we have displayed a text **"Welcome to world!!"** in the body attribute. This text is aligned in the **center** of the page by using the **Center widget**. Here, we have also styled the text by using the **TextStyle** widget, such as color, font size, etc.

3. drawer: *Drawer* is a slider menu or a panel which is displayed at the side of the Scaffold. The user has to swipe left to right or right to left according to the action defined to access the drawer menu. In the Appbar, an appropriate icon for the drawer is set automatically at a particular position. The gesture to open the drawer is also set automatically. It is handled by the Scaffold.

```
drawer: Drawer(
          child: ListView(
        children: const <Widget>[
          DrawerHeader(
            decoration: BoxDecoration(
              color: Colors.green,
            ),
            child: Text(
              ‘Main Menu’,
              style: TextStyle(
                color: Colors.green,
                fontSize: 24,
              ),
            ),
          ),
          ListTile(
            title: Text('Item 1'),
             leading
             : Icon(Icons.people), ),
          ),
          ListTile(
            title: Text('Item 2'),
          ),
        ],
      ),
    ),
```

As a parent widget we took *ListView* and inside it, we divided the panel into two parts, Header and Menu.*DrawerHeader* is used to modify the header of the panel. In header we can display icon or details of user according to the application. We have used *ListTile* to add the items to the menu.

4. floatingActionButton: *FloatingActionButton* is a button that is placed at the right bottom corner by default. *FloatingActionButton* is an icon button that floats over the content of the screen at a fixed place. If we scroll the page its position won’t change, it will be fixed.

```
floatingActionButton: FloatingActionButton(
         elevation: 10.0,
         child: Icon(Icons.add),
         onPressed: (){
       // action on button press
         }
     );
```

In the above code, we have used the **elevation** property that gives a **shadow effect** to the button. We have also used the Icon widget to give an icon to the button using preloaded Flutter SDK icons. The **onPressed()** property will be called when the user taps the button, and the statements **"I am Floating Action Button"** will be printed on the console.

5. backgroundColor: This property is used to set the background color of the whole Scaffold widget.

```
backgroundColor: Colors.yellow,
```

6. primary: It is used to tell whether the Scaffold will be displayed at the top of the screen or not. Its default value is **true** that means the height of the appBar extended by the height of the screen's status bar.

```
1. primary: true/false,
```

7. persistentFooterButton: It is a list of buttons that are displayed at the bottom of the Scaffold widget. These property items are always visible; even we have scroll the body of the Scaffold. It is always wrapped in a **ButtonBar widget**. They are rendered below the body but above the bottomNavigationBar.

```
        persistentFooterButtons: <Widget>[
          RaisedButton(
            onPressed: () {},
            color: Colors.blue,
            child: Icon(
              Icons.add,
              color: Colors.white,
            ),
          ),
          RaisedButton(
            onPressed: () {},
            color: Colors.green,
            child: Icon(
              Icons.clear,
              color: Colors.white,
            ),
1.   ),
2. ],
```

In the above code, we have used the **RaisedButton** that displays at the bottom of the Scaffold. We can also use the **FlatButton** instead of the RaisedButton.

8. bottomNavigationBar: *bottomNavigation Bar* is like a menu at the bottom of the Scaffold. We have seen this navigation bar in most of the applications. We can add multiple icons or texts or both in the bar as items.

```
bottomNavigationBar
    : BottomNavigationBar(
    currentIndex : 0,
    fixedColor
        : Colors.green,
    items
        : [
      BottomNavigationBarItem(
        title
            : Text("Home"),
        icon
            : Icon(Icons.home), ),
      BottomNavigationBarItem(
        title
            : Text("Search"),
        icon
            : Icon(Icons.search), ),
      BottomNavigationBarItem(
        title
            : Text("Profile"),
        icon
            : Icon(Icons.account_circle), ),
    ],
    onTap
        : (int indexOfItem){

    }),
```

- **backgroundColor:** used to set the color of the whole *Scaffold* widget.
- **floatingActionButtonAnimator:** used to provide animation to move *floatingActionButton*.
- **primary:** to tell whether the *Scaffold* will be displayed or not.
- **drawerScrimColor:** used to define the color for the primary content while a drawer is open.
- **bottomSheet**: This property takes in a widget (final) as the object to display it at the bottom of the screen.

- **drawerDragStartBehaviour**: This property holds *DragStartBehavior enum* as the object to determine the drag behaviour of the drawer.
- **drawerEdgeDragWidth**: This determines the area under which a swipe or a drag will result in the opening of the drawer. And it takes in a *double* as the object.
- **drawerEnableOpenGesture**: This property holds in a *boolean* value as the object to determine the drag gesture will open the drawer or not, by default it is set to true.
- **endDrawer**: The endDrawer property takes in a widget as the parameter. It is similar to the Drawer, except the fact it opens in the opposite direction.
- **endDrawerEnableOpenGesture**: Again this property takes in a *boolean* value as the object to determine whether the drag gesture will open the *endDrawer* or not.
- **extendBody**: The extendBody property takes in a *boolean* as the object. By default, this property is always false but it must not be null. If it is set to true in the presence of a *bottomNavigationBar* or *persistentFooterButtons*, then the height of these is added to the body and they are shifted beneath the body.
- **extendBodyBehindAppBar**: This property also takes in a *boolean* as the object. By default, this property is always false but it must not be null. If it is set to true the *appBar* instead of being on the body is extended above it and its height is added to the body. This property is used when the color of the *appBar* is not fully opaque.
- **floatingActionButtonLocation**: This property is responsible for the location of the *floatingActionBotton*.
- **persistentFooterButton**: This property takes in a list of widgets. Which are usually buttons that are displayed underneath the *scaffold*.
- **resizeToAvoidBottomInsets**: This property takes in a *boolean* value as the object. If set to true then the floating widgets on the *scaffold* resize themselves to avoid getting in the way of the on-screen keyboard.