**Flutter**

**An introduction to the open source SDK by Google**

Flutter is Google’s Mobile SDK to build native iOS and Android, Desktop (Windows, Linux, macOS), and Web apps from a single codebase.

When building applications with Flutter everything towards Widgets – the blocks with which the flutter apps are built.

They are structural elements that ship with a bunch of material design-specific functionalities and new widgets can be composed out of existing ones too. The process of composing widgets together is called composition.

The User Interface of the app is composed of many simple widgets, each of them handling one particular job.

That is the reason why Flutter developers tend to think of their flutter app as a tree of widgets.

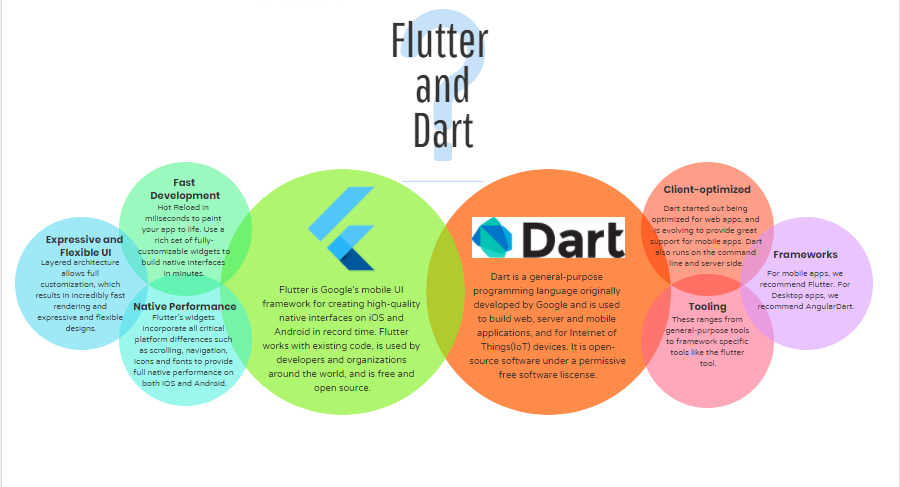
**Crowd**  
This instructional exercise is ready for experts who are seeking to make a lifelong in the field of versatile applications. This instructional exercise is expected to make you agreeable in getting everything rolling with Ripple system and its different functionalities.

**Essentials**  
This instructional exercise is composed expecting that the perusers are as of now mindful about what a System is and that the perusers have a sound information on Item Situated Programming and fundamental information on Android structure and Dart programming. On the off chance that you are a fledgling to any of these ideas, we recommend you to go through instructional exercises connected with these first, before you start with Vacillate.

**Types of widgets:**

* Stateless Widgets
* Stateful Widgets

**Flutter, Dart, and equivalent technologies**



First and foremost, let’s state the core differences between **Flutter** and **React Native.**

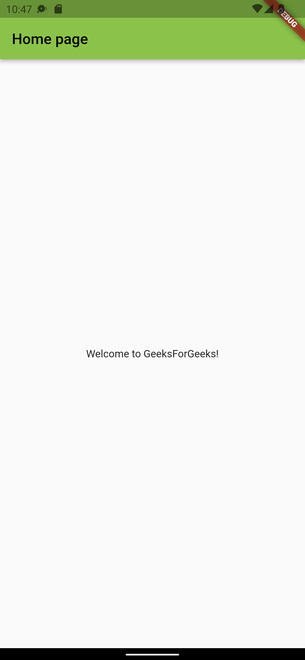
| Flutter | React Native |
| --- | --- |
| Initial release in 2017 | Initial release in 2015 |
| Based on Dart | Based on React Js |
| Controls every pixel on the screen | Controls via the native mobile components |
| Cross-Platform ( Mobile, Web, Desktop ) | Cross-Platform ( Mobile, React Native Web ) |
| Developed by Google | Developed by Facebook |
| Current Version 3.3.9 | Current Version 0.70 |
| App performance is higher. Flutter 60 fps or 120 fps animation. Flutter itself paints and controls every single pixel on the screen | High. It requires the JavaScript bridge to interact with the native components. |
| Flutter is the fasted growing framework for cross-platform development. Community support for flutter is amazing, with over 11100 Github stars, 15000 forks and over 41000 closed issues, it is leading the industry. | Community support for React Native is also good but it is not growing as fast as flutter. It has over 9300 Github stars, 20000 forks and over 19700 closed issues. |

,

* Dart

|  |
| --- |
| // Importing important packages require to connect  // Flutter and Dart  import 'package:flutter/material.dart';    // Main Function  **void** main() {  // Giving command to runApp() to run the app.    /\* The purpose of the runApp() function is to attach  the given widget to the screen. \*/    runApp(**const** MyApp());  }    // Widget is used to create UI in flutter framework.    /\* StatelessWidget is a widget, which does not maintain  any state of the widget. \*/    /\* MyApp extends StatelessWidget and overrides its  build method. \*/  **class** MyApp extends StatelessWidget {  **const** MyApp({Key? key}) : super(key: key);    // This widget is the root of your application.    @override    Widget build(BuildContext context) {  **return** MaterialApp(        // title of the application        title: 'Hello World Demo Application',        // theme of the widget        theme: ThemeData(          primarySwatch: Colors.lightGreen,        ),        // Inner UI of the application        home: **const** MyHomePage(title: 'Home page'),      );    }  }    /\* This class is similar to MyApp instead it  returns Scaffold Widget \*/  **class** MyHomePage extends StatelessWidget {  **const** MyHomePage({Key? key, required **this**.title}) : super(key: key);    final String title;      @override    Widget build(BuildContext context) {  **return** Scaffold(        appBar: AppBar(          title: Text(title),        ),        // Sets the content to the        // center of the application page        body: **const** Center(            // Sets the content of the Application            child: Text(          'Welcome to GeeksForGeeks!',        )),      );    }  } |

**Output**



And, the biggest selling point of Flutter Tech are two things:

1. **High-Performance App**: The Apps developed using Flutter are highly expressive and have flexible UI. Its fast development due to hot reloading brings the app to life and its expressiveness provides features that are keened for native end-user experiences.
2. **Expressive and Flexible UI:**Flutter lets developers build beautiful-looking apps with ease by using prebuild material widgets. Even though many widgets are prebuilt still flutter enables full customization of the widget.
3. **Fast Development & Hot Reloading**: Hot Reloading refers to the injection of new versions of the files that you edited at runtime while keeping the app running.

**The Pros and Cons of Flutter:**

**Pros:**

* Flutter uses a single codebase, called, Dart for both platforms, Android and iOS which is a simple language ensuring type safety.
* Both Flutter *language* and *community* are developing with great speed, releasing new features, widgets, and add-ons.
* Flutter has its own set of widgets rather than using the widgets provided by the host operating system which means the user provides its own gesture recognition model, thus, having greater control over the precise rendering or customization of the widgets.
* The hot-reloading is a game-changer in the productivity of the development process. It gives a lively effect to the app under development, thus making the whole development cycle more exciting for the UI/UX developer using Flutter.
* Flutter is not bound to the ROM w.r.t. the widget system. So, it enhances its portability over a wide ambit of Android versions and thus, lowering its dependencies on the host platform.
* Dart and Flutter unite closely to optimize dart Virtual Machine(VM) for those mobiles which are specifically needed by Flutter.
* Flutter is an established player in the field of cross-platform application development with amazing community support.

**Cons:**

* In reality, there are no cons to flutter because there isn’t any other framework as effective and elaborate as flutter. Even though if we have to list any it would be related to Dart programming language as while converting dart to JavaScript there are some bugs to be fixed, dart doesn’t have a framework for backend, etc.,

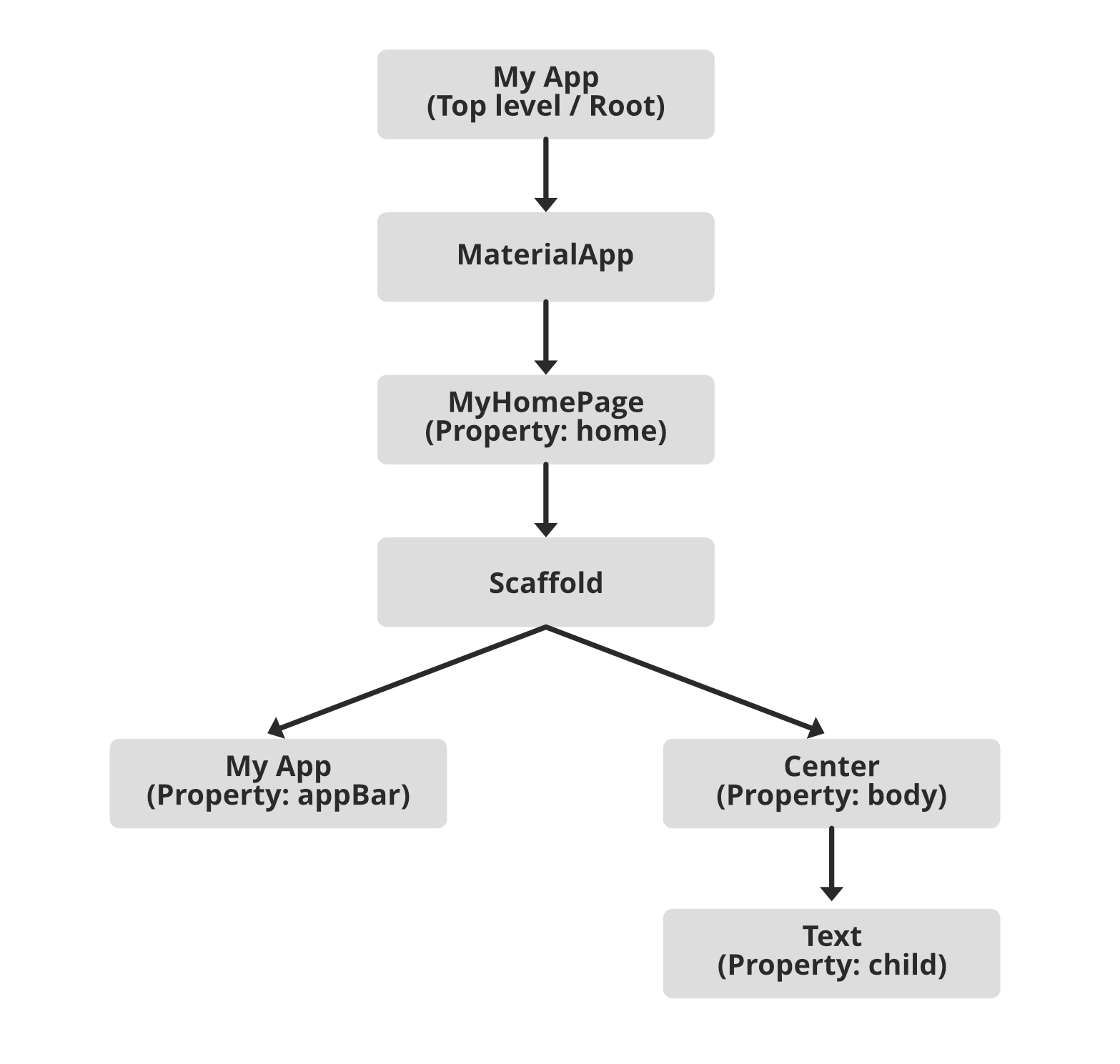
# Flutter – Architecture Application

Flutter architecture application mainly consists of:

* **Widgets**
* **Gestures**
* **Concept of State**
* **Layers**

## Widgets

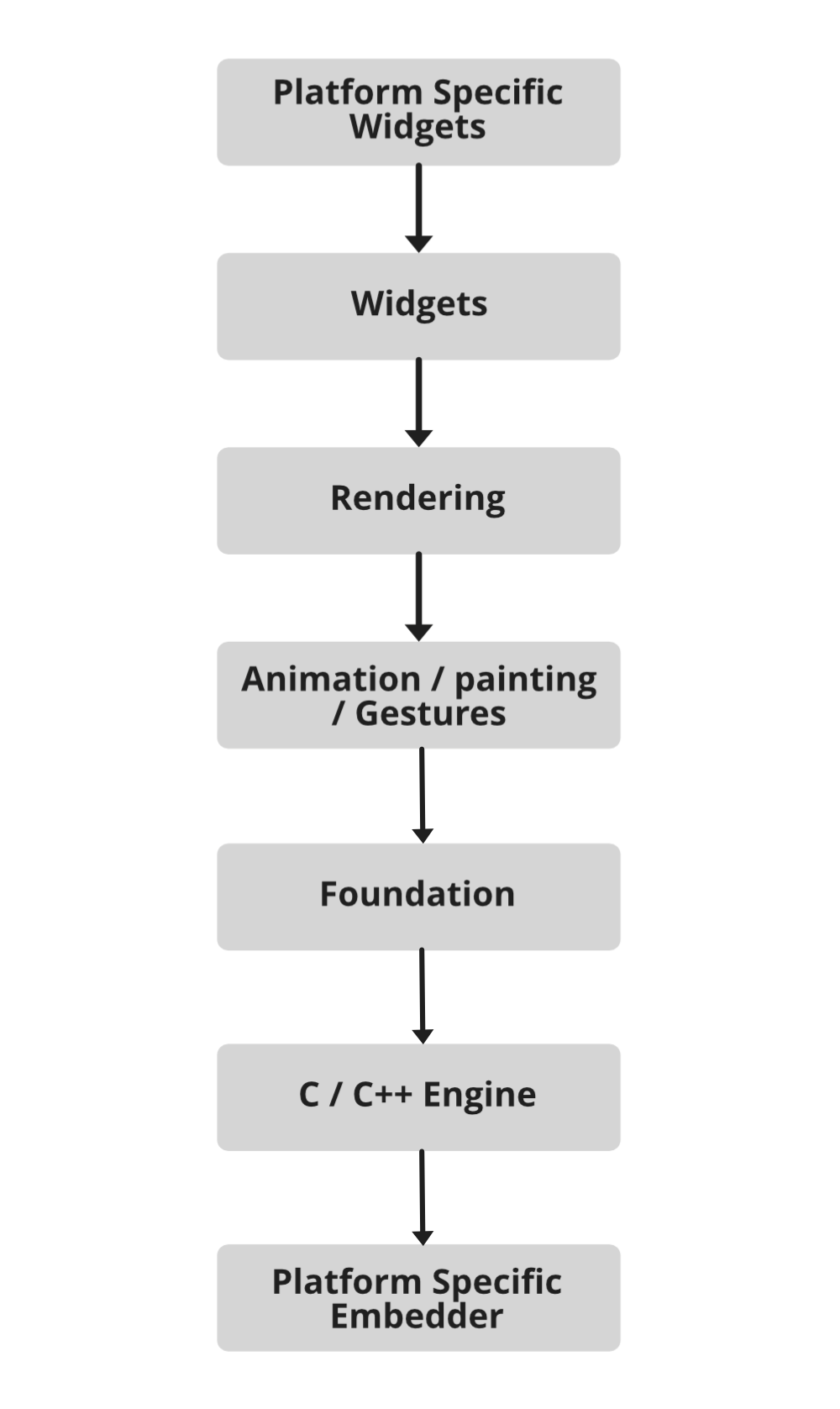
Widgets are the primary component of any flutter application. It acts as a UI for the user to interact with the application. Any flutter application is itself a widget that is made up of a combination of widgets. In a standard application, the root defines the structure of the application followed by a Material App widget which basically holds its internal components in place. This is where the properties of the UI and the application itself is set. The Material App has a Scaffold widget that consists of the visible components (widgets) of the application. The Scaffold has two primary properties namely the body and  appbar. It holds all the child widgets and this is where all its properties are defined. The below diagram shows the hierarchy of a flutter application:.



Inside Scaffold, there is usually an appbar widget, which as the name suggests define appbar of the application. The scaffold also has a body where all the component widgets are placed. This is where these widget’s properties are set. All these widgets combined form the Homepage of the application itself. The Center widget has a property, Child, which refers to the actual content and it is built using the Text widget.

## Layers

The Flutter framework is categorized based on its complexity and establishes a hierarchy based on the decreasing level of these complexities. These categories are often called Layers. These layers are built on top of one another. The topmost layer is a widget specific to the operating system of the device (ie, Android or iOS). The second layer consists of the native flutter widgets, which comprise structural UI components, gesture detectors, state management components, etc. This third layer is where all the Ui and state rendering occurs. It is the layer that includes all the visible components of the flutter application. The following layer consists of animations used in transitions, image flow, and gestures. These further go on to the very high level of system design that is not the target of this article. The below diagram gives an overview of the same:



## Gestures

All physical form of interaction with a flutter application is done through pre-defined gestures. Gesture-Detectors are used for the same. It is an invisible widget that is used to process physical interaction with the flutter application. The interaction includes gestures like tapping, dragging, and swiping, etc. These features can be used to creatively enhance the user experiences of the app by making it perform desired actions based on simple gestures.

## Concept of State

If you have ever worked with React-js, you might be familiar with the concept of a state. The states are nothing but data objects. Flutter also operates on similar turf. For the management of state in a Flutter application, Stateful-Widget is used. Similar to the concept of state in React-js, the re-rendering of widgets specific to the state occurs whenever the state changes. This also avoids the re-rendering of the entire application, every time the state of a widget changes.

The architecture of a Flutter app or the flutter framework generally consists of a combination of small and larger widgets interacting in conjuncture to build the application. All of its layers are integral to its design and functioning. As simple as building an application in flutter is, it is built with equally complex components at its core.

**Android Studio Setup for Flutter Development**

This article will show how to set up **Android Studio** to run **Flutter Applications**. Android Studio is one of the popular IDE( integrated development environment  ) developed by Google itself to create cross-platform android applications. First, you have to install **Android Studio** of **version 3.0 or later**, as it offers an integrated IDE experience for a **Flutter**. You can refer this for details: [**Android Studio**](https://developer.android.com/studio)

**Install the Flutter and Dart plugins:**

After the successful installation of Android Studio, you have to install Flutter and Dart plugins. To do so follow the steps mentioned below:

1. **Start** Android Studio.
2. Open plugin preferences **(Configure > Plugins as of v3.6.3.0 or later)**.
3. Select the **Flutter** plugin and click Install.
4. Click Yes when prompted to install the **Dart** plugin.
5. Click **Restart** when prompted.

Open plugin preferences:

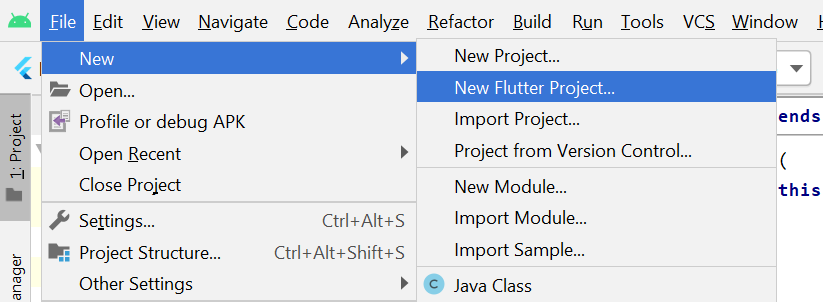
* For macOS: Preferences > Plugins on macOS,
* For Linux and Windows: File > Settings > Plugins

Now, Select Marketplace, select the Flutter plugin and click Install.

**Creating the application:**

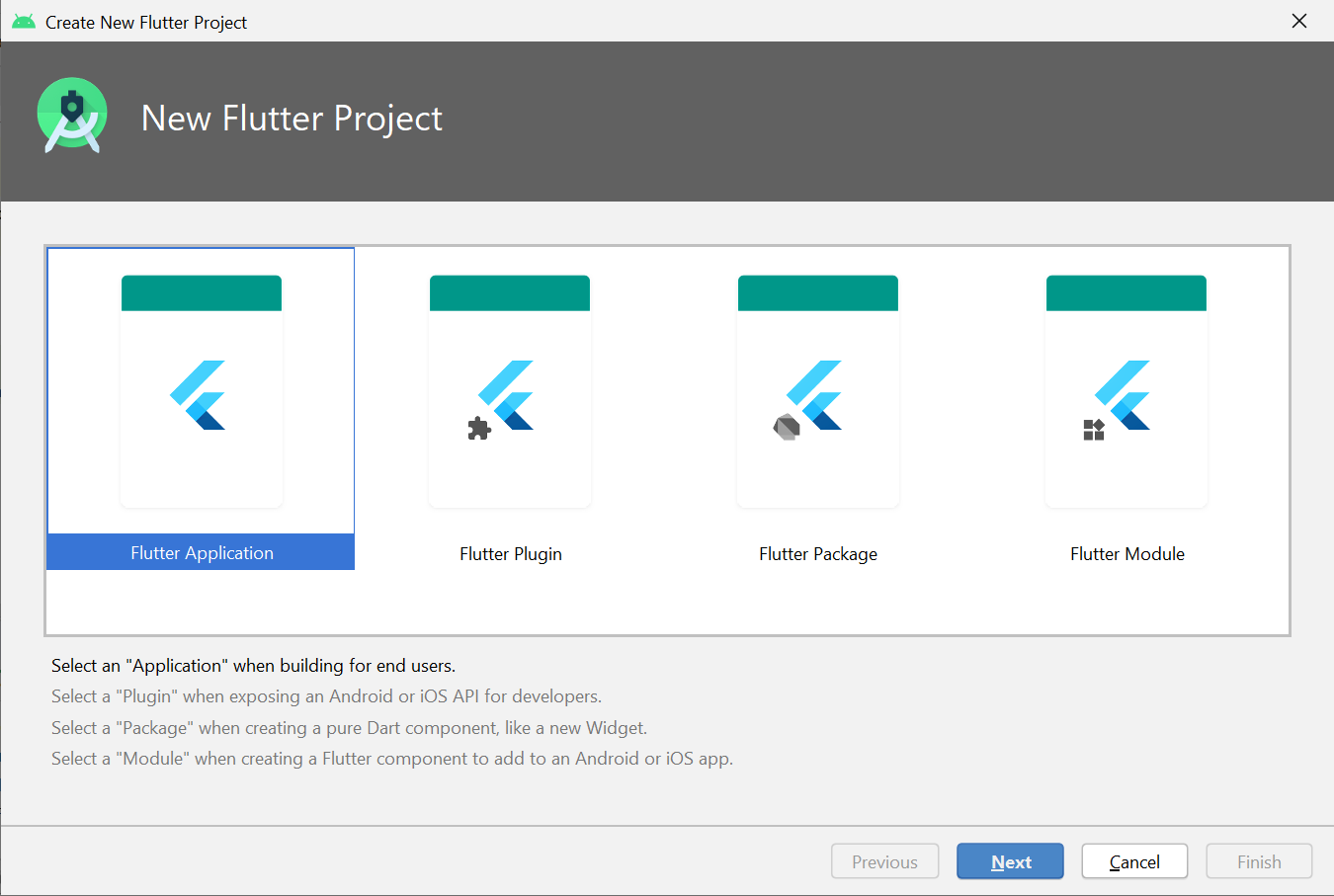
After installing Dart and Flutter plugins create a flutter app to check if it is working properly or not, to do so follow the steps mentioned below:

**Step 1:** Open the **IDE** and select Start a **new Flutter project**.



*Select New Flutter Project from the file menu*

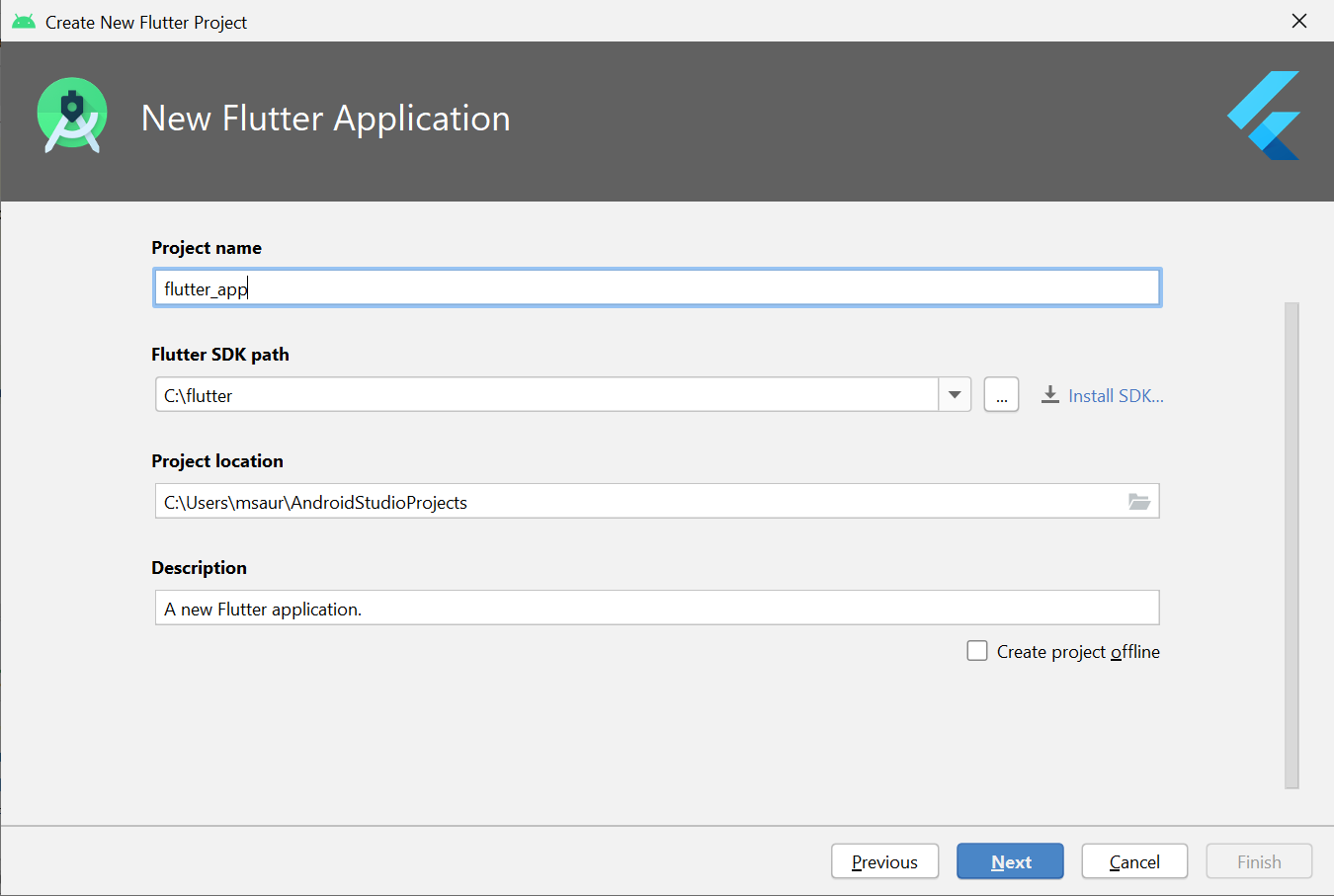
**Step 2:** Select the **Flutter Application** as the project type. Then click **Next**.



*Select the type of flutter application*

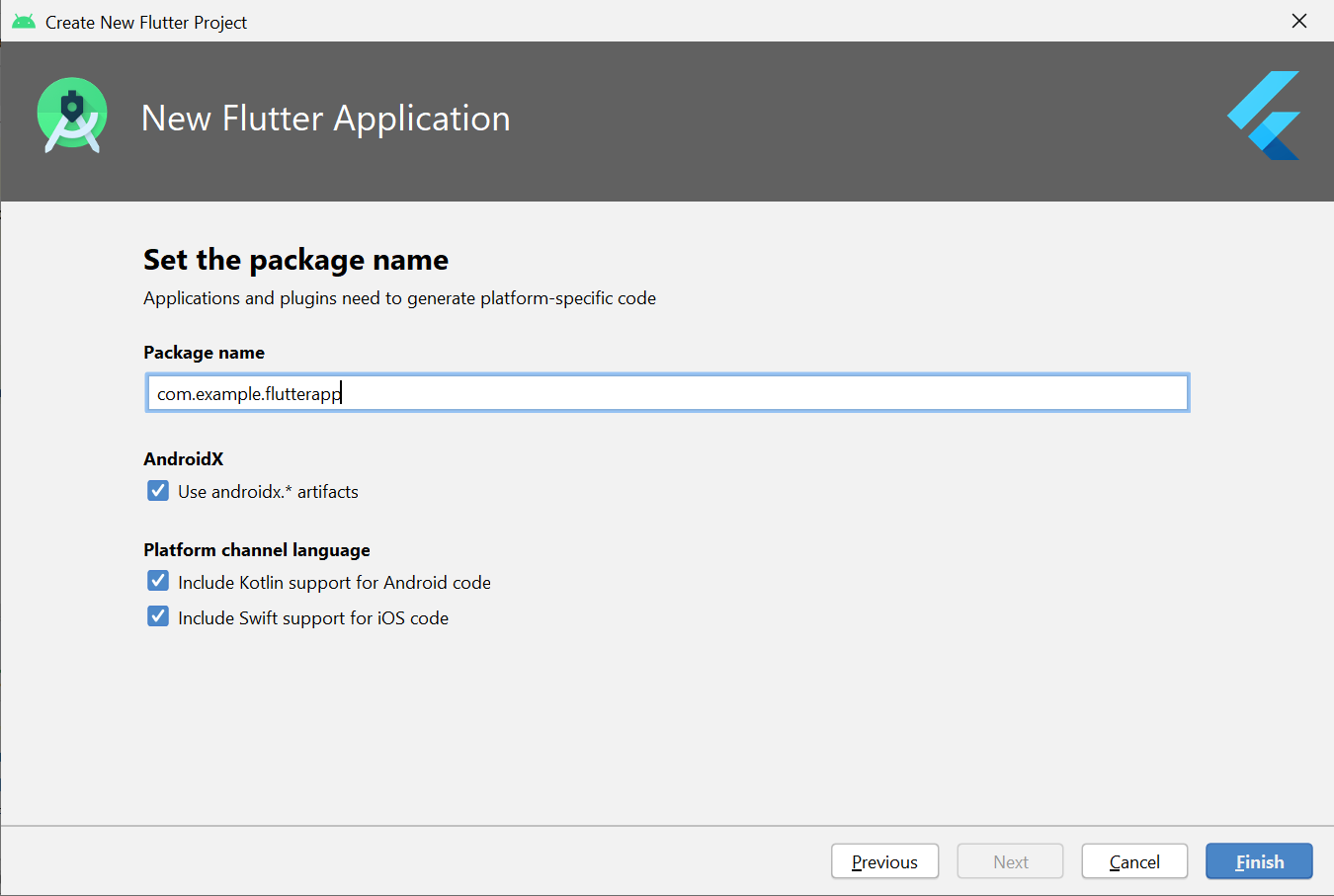
**Step 3:** Verify the**Flutter SDK path**specifies the SDK’s location **(select Install SDK… if the text field is blank)**.

**Step 4:** Enter a project name (for example, *myapp*). Then click **Next**.



*Enter the name of the project and also its description. Also, add the path of the Flutter SDK in the****Flutter SDK path****or click on install SDK.*

**Step 5:** Click**Finish**.



*Hit the Finish Button and wait till the project is created by the IDE.*

**Step 6:** Wait for Android Studio to install the SDK and create the project.

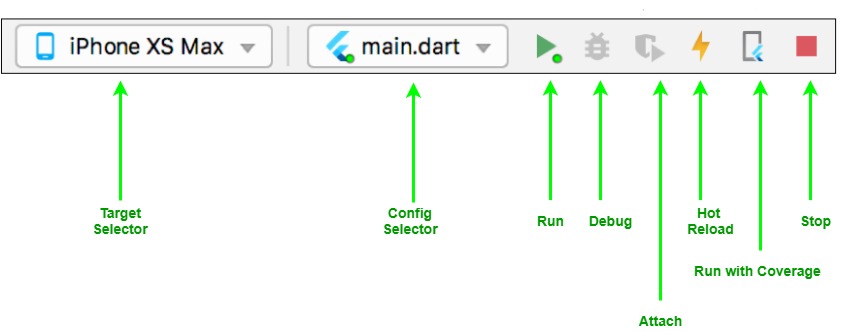
**Note:** When creating a new Flutter app, some Flutter IDE plugins ask for a company domain name in reverse order, something like com. example. The company domain name and project name are used together as the package name for Android (the Bundle ID for iOS) when the app is released. If you think that the app might be released, it’s better to specify the package name now. The package name can’t be changed once the app is released, so make the name unique.

The above steps create a Flutter project directory called *flutter\_app*that contains a simple demo app that uses Material Components.

**Running the application:**

Follow the below steps to run the flutter application that was structured above:

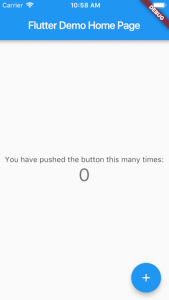
**Step 1:** Locate the main Android Studio toolbar:



**Step 2:** In the **target selector**, select an Android device for running the app. If none are listed as available, select **Tools> Android > AVD Manager** and create one there.

**Step 3:** Click the run icon in the toolbar, or invoke the menu item **Run > Run**.

After the app build completes, you’ll see the starter app on your device.



**Getting Started with Cross-Platform Mobile Application using Flutter**

**Flutter** is an **open-source** mobile application development SDK created by **Google** to develop **cross-platform mobile applications**. Flutter makes it extremely easy and fast for even **novice programmers** (An computer programmers who is not experienced at any programming languages) .to build high-quality and responsive native mobile apps. No prior app development experience is required! .This article will show you how to build a simple **Hello Flutter** application and then run it on your own Android device!

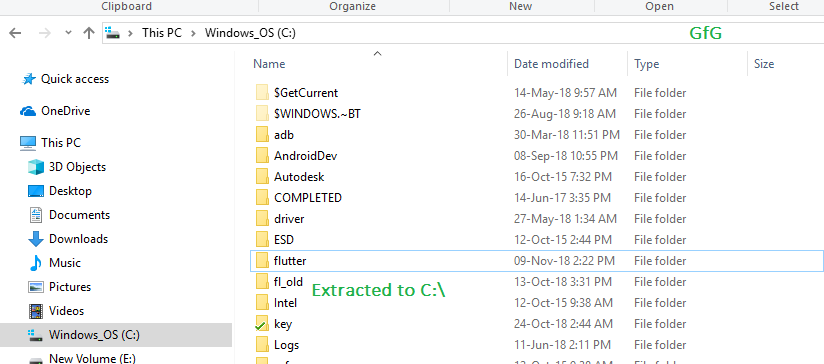
**Why use Flutter over native Java development?**

Flutter has some unique features compared to native Java and other SDK such as:

* **Fast Development**: Flutter makes use of customizable Widgets that can be nested together to create the app interface, similar to how HTML is structured. One can refer to this for details: [Flutter for Web Devs](https://flutter.io/docs/get-started/flutter-for/web-devs).
* **Hot Reload**: Instantly see changes made in code, without recompiling.
* **It’s Native in the end**: Under the hood, the codebase is compiled into Java, ensuring the same performance as that offered by native apps.
* **Light on system resources**: Flutter can be comfortably run on a lower resources system.

**Setting Up Flutter**

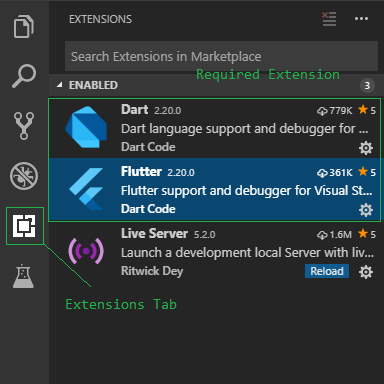
* **Step 1: Set up the Flutter SDK**
  1. Download the latest SDK here: [Flutter SDK Archive](https://flutter.io/docs/development/tools/sdk/archive)
  2. Extract the zip file and place the contained ‘*flutter*‘ folder in the desired directory.



*Here it is extracted the folder to the C:\ drive*

1. **Note**: It is recommended to not install Flutter in a directory which may require admin privileges, like ‘C:\Program Files\’

* **Step 2: Add Flutter to PATH**: Though not required, it is recommended to set the **PATH variable** to make Flutter easily accessible from anywhere on the system.
  1. Go to **Edit environment variables for your account** in Control Panel. You can search for this setting in your search bar.
  2. Under **User variables** check if there is an entry called PATH:
     + If it exists, add a new path to ‘flutter\bin’.
     + If the entry does not exist, create a new entry named Path, and then add the full location to ‘flutter\bin’ (see example image)
  3. Reboot Windows after setting the PATH variable for it to work.
* **Step 3: Set up Android Studio**: Android Studio automatically downloads the development tools required for Flutter to work with Android.
  1. Download Android Studio here: [Android Studio](https://developer.android.com/studio/index.html)
  2. Start Android Studio and follow the SDK Manager wizard to download all the required build tools.
* **Step 4: Set up Visual Studio Code**: Visual Studio Code (or VS Code) is a light code editor that can be used in Flutter development. In this article, VS is used instead of Android Studio as it is lighter and has the minimal required features.
  1. Download and install VS Code: [VS Code Download](https://code.visualstudio.com/).
  2. To help in development, two VS Code extensions are recommended. Install the **Flutter**and **Dart**plugins from VS Code’s Extension Tab. You can refer: [Setting Up VS Code Extensions for Flutter](https://flutter.io/docs/get-started/editor?tab=vscode#vscode)

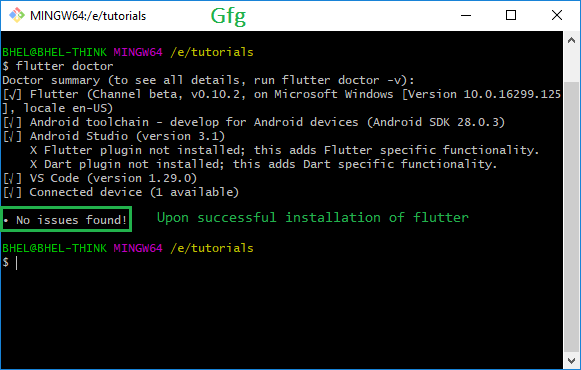


*The VS Code Extensions to be installed*

1. **Install Git Bash**: Optional, but a recommended command prompt. Git Bash provides many common unix commands that are useful for some quick tasks. Download Git Bash for Windows here: [Git Downloads](https://git-scm.com/downloads)
2. **Run Flutter Doctor**: Flutter doctor is an inbuilt tool by flutter that can be used to check the status of the Flutter installation. After setting up the PATH variable, you can open up the Command Prompt and execute:

flutter doctor

1. This will check everything required for Flutter to run and give a detailed report of any error it encounters.



*Upon successful setup, flutter doctor would show no errors*

**Creating an empty template project**

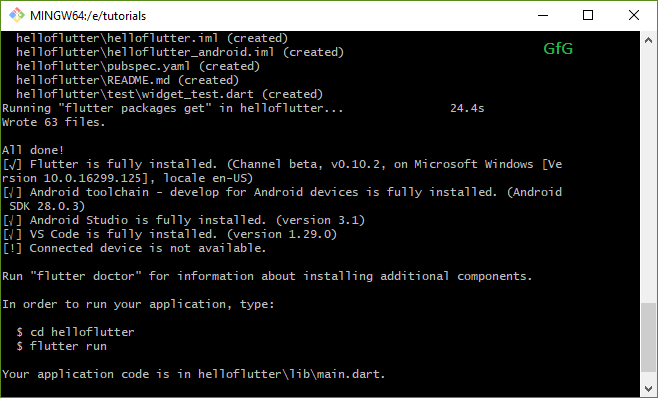
1. Navigate to the place where you want your project to be created. Open a command prompt (you can also use the context **‘Git Bash here’** by right clicking, to open Git Bash in this location) and type in the command for creating a new project:

flutter create project\_name

1. For example: for a project named **‘helloflutter’**executing

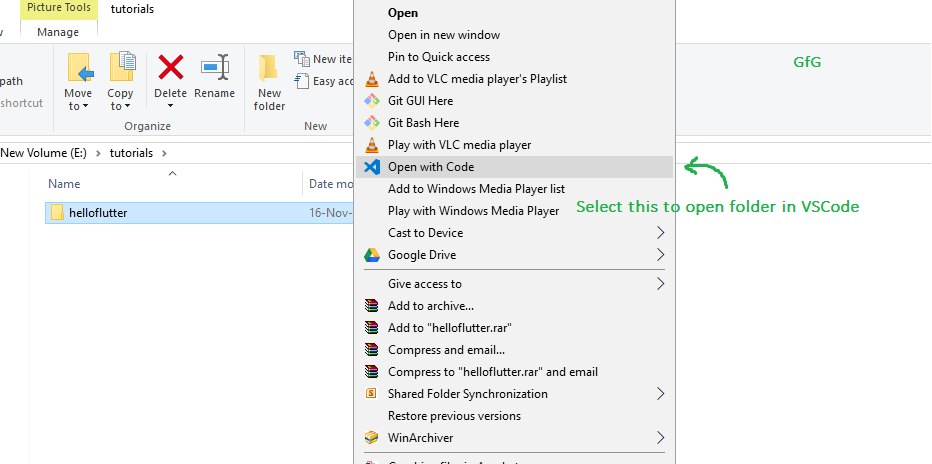
flutter create helloflutter

1. will create a new Flutter project with name *helloflutter*



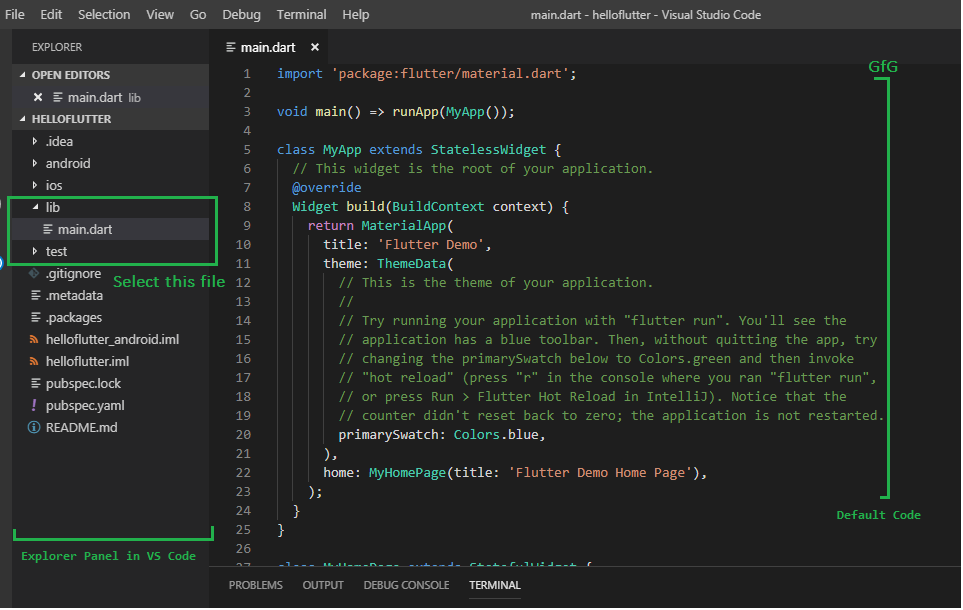
*The new project named helloflutter is created*

1. Open this folder in VS Code. You can right-click and use the context menu to open directly into VS Code, or start VS Code first and then open this folder as a project.



*Context menu to open folder in VS Code*

1. The large panel on the left that displays all the files and folders is known as the **Explorer Panel**. Navigate to **‘lib’** folder and select the **‘main.dart’** file. This file is the entry point from where the app starts its execution.

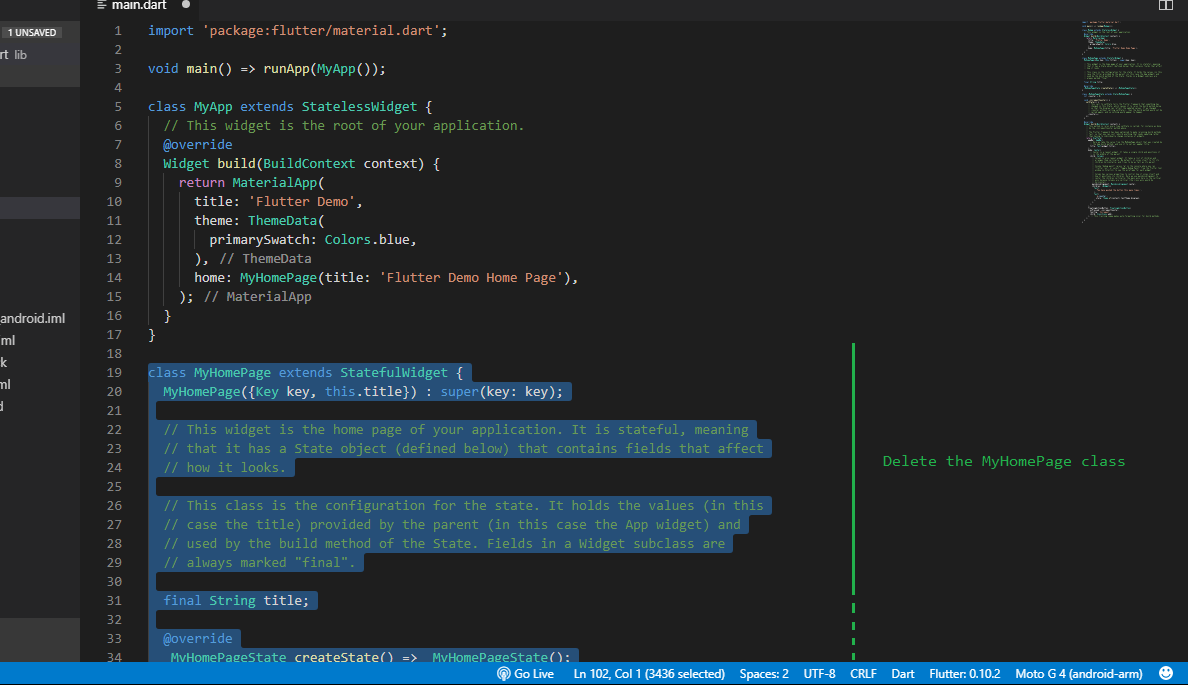


*Locating the main.dart file in the lib folder*

1. The code that opens up is that of the template application. Try running this simple app right away!

**Saying Hello Flutter!**

1. Delete the MyHomePage Widget.



*Delete the MyHomePage Widget*

1. Create a new **Stateless Widget** and name it **HelloFlutter**. [Stateless Widgets](https://docs.flutter.io/flutter/widgets/StatelessWidget-class.html) are used to define Widgets which don’t have to deal with changes to its internal state. They are mostly used to build components which once drawn, are not required to update.

* Dart

|  |
| --- |
| **class** HelloFlutter extends StatelessWidget {  **const** HelloFlutter({Key? key}) : super(key: key);    @override  Widget build(BuildContext context) {  **return** Container(        );  }  } |



*Added the new Stateless Widget HelloFlutter*

1. Replace the **Container**widget with a **Scaffold**widget: A [Scaffold](https://docs.flutter.io/flutter/material/Scaffold-class.html)implements basic material design visual layout structure. This Widget provides APIs for showing drawers, appbars and the body of the app. The **body**property of the Scaffold will be used here to display the contents of the app.

* Dart

|  |
| --- |
| **class** HelloFlutter extends StatelessWidget {  **const** HelloFlutter({Key? key}) : super(key: key);      @override    Widget build(BuildContext context) {  **return** **const** Scaffold(        );    }  } |

1. Declare a **Container** Widget in the body of the Scaffold.   
   A [Container](https://docs.flutter.io/flutter/widgets/Container-class.html) Widget is a useful widget which combines common painting, positioning, and sizing widgets. You can wrap any widget with a Container and control the above mentioned properties.

* Dart

|  |
| --- |
| **class** HelloFlutter extends StatelessWidget {  **const** HelloFlutter({Key? key}) : super(key: key);      @override    Widget build(BuildContext context) {  **return** Scaffold(        body: Container(          ),      );    }  } |

1. The Container widget has an **alignment** property which will help to position the Widget to the center of the screen. Set the alignment with the Alignment class:

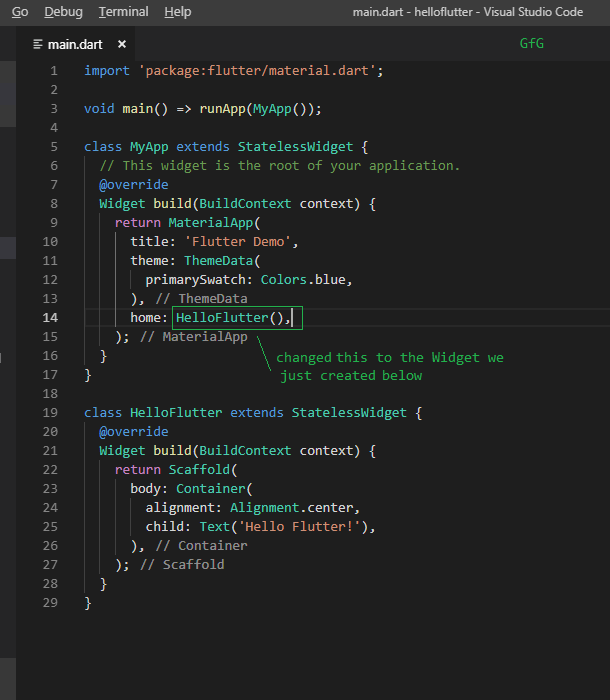
alignment: Alignment.center

1. In the **child** property of the Container Widget, declare a **Text** Widget: The [Text](https://docs.flutter.io/flutter/widgets/Text-class.html)Widget deals with displaying and handling text. After creating the Text Widget, put in **‘Hello Flutter’** between the parentheses in single quotes. Whatever is put in between the single quotes is displayed by the Text Widget.

* Dart

|  |
| --- |
| **class** HelloFlutter extends StatelessWidget {  **const** HelloFlutter({Key? key}) : super(key: key);      @override    Widget build(BuildContext context) {  **return** Scaffold(        body: Container(          alignment: Alignment.center,          child: **const** Text('Hello Flutter!'),        ),      );    }  } |

1. Finally, in the **home**property of the main My App class above, change it from MyHomePage(…) to HelloFlutter(). This allows the main MyApp class to refer to the Hello Flutter just created.



*Changing the home property to call the Widget we made*

1. Now run the app using the ‘*flutter run*‘ command.

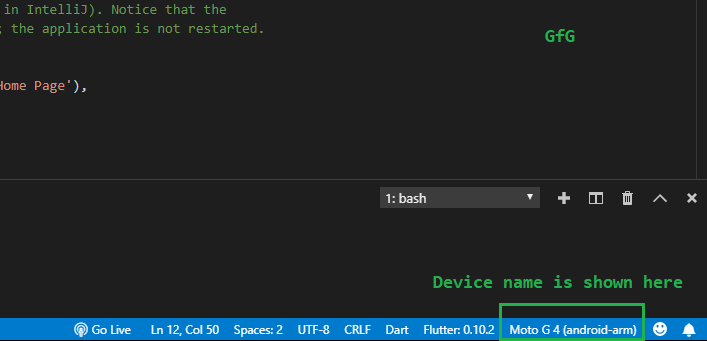


*Running the App*

1. A text ‘Hello Flutter!’ will appear written in the middle of the screen.

**Running the HelloFlutter App**

1. Connect a physical device to the PC and enable [Developer Mode](https://developer.android.com/studio/debug/dev-options). If the device is successfully recognised by the PC, the device name would appear in the lower-right corner of VS Code.

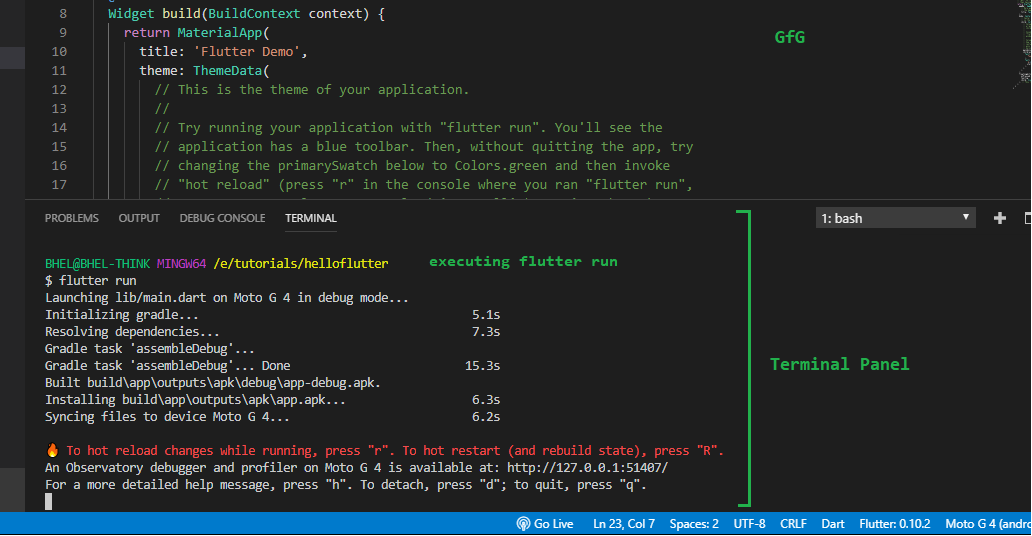


*The device name will be shown here*

1. If you would like to setup an emulator instead, see: [Set up the Android emulator](https://flutter.io/docs/get-started/install/windows#set-up-the-android-emulator). The emulated device would also show up here similarly.
2. Open the integrated **terminal**by pressing the key combination **[CTRL + `]** (Control key + backtick).
3. Run the command:

flutter run

1. Wait for a few minutes. As this is first run, some downloads and installation take place in the background related to gradle. Subsequent compilations would be a lot faster.



*Executing ‘flutter run’ and compilation of the default app*

1. After compilation, the app will get installed and run on the connected device or emulator automatically.



*Running the App*

1. Stop the app by pressing ‘d’ in the terminal. This is what the compilation and running of any app will be like.

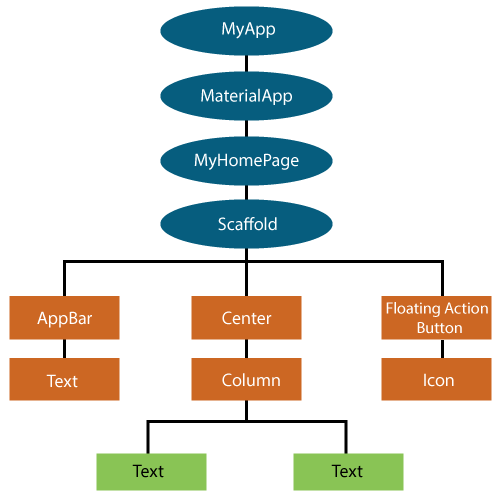
# Flutter Widgets

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.

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:

1. Class ImageWidget **extends** StatelessWidget {
2. // Class Stuff
3. }

**Hello World Example**

1. **import** 'package:flutter/material.dart';
3. **class** MyHomePage **extends** StatelessWidget {
4. MyHomePage({Key key, **this**.title}) : **super**(key: key);
5. // This widget is the home page of your application.
6. **final** String title;
8. @override
9. Widget build(BuildContext context) {
10. **return** Scaffold(
11. appBar: AppBar(
12. title: Text(**this**.title),
13. ),
14. body: Center(
15. child: Text('Hello World'),
16. ),
17. );
18. }
19. }

## **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.

1. **new** Text(
2. 'Hello, Javatpoint!',
3. textAlign: TextAlign.center,
4. style: **new** TextStyle(fontWeight: FontWeight.bold),
5. )

**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
2. **new** FlatButton(
3. child: Text("Click here"),
4. onPressed: () {
5. // Do something here
6. },
7. ),
9. //RaisedButton Example
10. **new** RaisedButton(
11. child: Text("Click here"),
12. elevation: 5.0,
13. onPressed: () {
14. // Do something here
15. },
16. ),

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.

1. assets:
2. - assets/

Now, add the following line in the dart file.

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

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

1. **class** MyHomePage **extends** StatelessWidget {
2. MyHomePage({Key key, **this**.title}) : **super**(key: key);
3. // This widget is the home page of your application.
4. **final** String title;
6. @override
7. Widget build(BuildContext context) {
8. **return** Scaffold(
9. appBar: AppBar(
10. title: Text(**this**.title),
11. ),
12. body: Center(
13. child: Image.asset('assets/computer.png'),
14. ),
15. );
16. }
17. }

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



**Icon**

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

1. **new** Icon(
2. Icons.add,
3. size: 34.0,
4. )

### 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.

1. **new** Column(
2. mainAxisAlignment: MainAxisAlignment.center,
3. children: <Widget>[
4. **new** Text(
5. "VegElement",
6. ),
7. **new** Text(
8. "Non-vegElement"
9. ),
10. ],
11. ),

**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.

1. **new** Row(
2. mainAxisAlignment: MainAxisAlignment.spaceEvenly,
3. children: <Widget>[
4. **new** Text(
5. "VegElement",
6. ),
7. **new** Text(
8. "Non-vegElement"
9. ),
10. ],
11. ),

**Center**

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

**Example**

1. Center(
2. child: **new** clumn(
3. mainAxisAlignment: MainAxisAlignment.spaceEvenly,
4. children: <Widget>[
5. **new** Text(
6. "VegElement",
7. ),
8. **new** Text(
9. "Non-vegElement"
10. ),
11. ],
12. ),
13. ),

**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**

1. Padding(
2. padding: **const** EdgeInsets.all(6.0),
3. child: **new** Text(
4. "Element 1",
5. ),
6. ),

**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**

1. **class** Car **extends** StatefulWidget {
2. **const** Car({ Key key, **this**.title }) : **super**(key: key);
4. @override
5. \_CarState createState() => \_CarState();
6. }
8. **class** \_CarState **extends** State<Car> {
9. @override
10. Widget build(BuildContext context) {
11. **return** Container(
12. color: **const** Color(0xFEEFE),
13. child: Container(
14. child: Container( //child: Container() )
15. )
16. );
17. }
18. }

### StatelessWidget

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

**Example**

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

# 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](https://www.javatpoint.com/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.

1. **const** Scaffold({
2. Key key,
3. **this**.appBar,
4. **this**.body,
5. **this**.floatingActionButton,
6. **this**.floatingActionButtonLocation,
7. **this**.persistentFooterButtons,
8. **this**.drawer,
9. **this**.endDrawer,
10. **this**.bottomNavigationBar,
11. **this**.bottomSheet,
12. **this**.floatingActionButtonAnimator,
13. **this**.backgroundColor,
14. **this**.resizeToAvoidBottomPadding = **true**,
15. **this**.primary = **true**,
16. })

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:

1. Widget build(BuildContext context)
2. {
3. **return** Scaffold(
4. appBar: AppBar(
5. title: Text('First Flutter Application'),
6. ), )
7. }

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:

1. Widget build(BuildContext context) {
2. **return** Scaffold(
3. appBar: AppBar(
4. title: Text('First Flutter Application'),
5. ),
6. body: Center(
7. child: Text("Welcome to Javatpoint",
8. style: TextStyle( color: Colors.black, fontSize: 30.0,
9. ),
10. ),
11. ),
12. }

In the above code, we have displayed a text **"Welcome to Javatpoint!!"** 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:** It is a **slider panel** that is displayed at the side of the body. Usually, it is hidden on the mobile devices, but the user can swipe it left to right or right to left to access the drawer menu. It uses the **Drawer widget properties** slides in a **horizontal direction** from the Scaffold edge to show navigation links in the application. An appropriate **icon** for the drawer is set automatically in an appBar property. The **gesture** is also set automatically to open the drawer. See the following code.

1. drawer: Drawer(
2. child: ListView(
3. children: **const** <Widget>[
4. DrawerHeader(
5. decoration: BoxDecoration(
6. color: Colors.red,
7. ),
8. child: Text(
9. 'Welcome to Javatpoint',
10. style: TextStyle(
11. color: Colors.green,
12. fontSize: 30,
13. ),
14. ),
15. ),
16. ListTile(
17. title: Text('1'),
18. ),
19. ListTile(
20. title: **new** Text("All Mail Inboxes"),
21. leading: **new** Icon(Icons.mail),
22. ),
23. Divider(
24. height: 0.2,
25. ),
26. ListTile(
27. title: **new** Text("Primary"),
28. ),
29. ListTile(
30. title: **new** Text("Social"),
31. ),
32. ListTile(
33. title: **new** Text("Promotions"),
34. ),
35. ],
36. ),
37. ),

In the above code, we use the drawer property of Scaffold for creating a drawer. We have also used some other widgets to make it attractive. In the **ListView** widget, we have divided the panel into two parts, **Header** and **Menu**. The DrawerHeader property modifies the panel header that also contains an icon or details according to the application. Again, we have used **ListTile** to add the list items in the menu.

**4. floatingActionButton:** It is a button displayed at the bottom right corner and floating above the body. It is a circular icon button that floats over the content of a screen at a fixed place to promote a primary action in the application. While scrolling the page, its position cannot be changed. It uses the FloatingActionButton widget properties using **Scaffold.floatingActionButton**. See the below code:

1. Widget build(BuildContext context) {
2. **return** Scaffold(
3. appBar: AppBar(title: Text('First Flutter Application')),
4. body: Center(
5. child: Text("Welcome to Javatpoint!!"),
6. ),
7. floatingActionButton: FloatingActionButton(
8. elevation: 8.0,
9. child: Icon(Icons.add),
10. onPressed: (){
11. print('I am Floating Action Button');
12. }
13. );
14. }

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.

1. 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.

1. persistentFooterButtons: <Widget>[
2. RaisedButton(
3. onPressed: () {},
4. color: Colors.blue,
5. child: Icon(
6. Icons.add,
7. color: Colors.white,
8. ),
9. ),
10. RaisedButton(
11. onPressed: () {},
12. color: Colors.green,
13. child: Icon(
14. Icons.clear,
15. color: Colors.white,
16. ),
17. ),
18. ],

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:** This property is like a **menu that displays a navigation bar** at the bottom of the Scaffold. It can be seen in most of the mobile applications. This property allows the developer to add multiple icons or texts in the bar as items. It should be rendered below the body and persistentFooterButtons. See the below code:

1. bottomNavigationBar: BottomNavigationBar(
2. currentIndex: 0,
3. fixedColor: Colors.grey,
4. items: [
5. BottomNavigationBarItem(
6. title: Text("Home"),
7. icon: Icon(Icons.home),
8. ),
9. BottomNavigationBarItem(
10. title: Text("Search"),
11. icon: Icon(Icons.search),
12. ),
13. BottomNavigationBarItem(
14. title: Text("User Profile"),
15. icon: Icon(Icons.account\_circle),
16. ),
17. ],
18. onTap: (**int** itemIndex){
19. setState(() {
20. \_currentIndex = itemIndex;
21. });
22. },
23. ),

In the above code, we have used the BottomNavigationBar widget to display the menu bar. The **fixedColor** property is used for the **color** of the active icon. The **BottomNavigationBarItems** widget is used to add items in the bar containing text and icon as its child property. We had also used **onTap(int itemIndex)** function to perform an action when we tapped on the items, which works according to their index position.

**9. endDrawer:** It is similar to a drawer property, but they are displayed at the right side of the screen by default. It can be swiped right to left or left to right.

**10. resizeToAvoidBottomInset:** If it is **true**, the body and the Scaffold's floating widgets should adjust their size themselves to avoid the onscreen keyboard. The bottom property defines the onscreen keyboard height.

**11. floatingActionButtonLocation:** By default, it is positioned at the bottom right corner of the screen. It is used to determine the position of the floatingActionButton. It contains many predefined constants, such as centerDocked, centerFloat, endDocked, endFloat, etc.

That's all about the Scaffold's various properties that give us an overview of the Scaffold widget. Its main purpose is to get familiar with different properties and their usage in Flutter application. If we want to learn it in more detail, refer to the flutter documentation [here](https://api.flutter.dev/flutter/material/Scaffold-class.html).

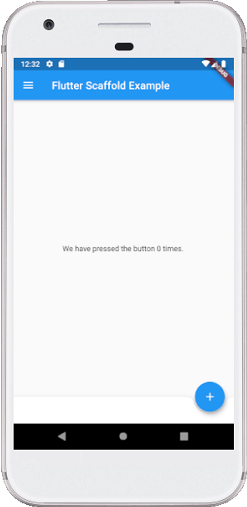
Let us see the example where we have tried to use most of the Scaffold properties to understand this widget quickly and easily.

In this example, we are going to see a Scaffold widget with an AppBar, BottomAppBar, FloatingActionButton, floatingActionButtonLocation, and drawer properties.

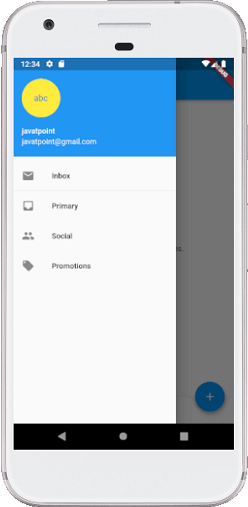
1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
5. /// This Widget is the main application widget.
6. **class** MyApp **extends** StatelessWidget {
7. @override
8. Widget build(BuildContext context) {
9. **return** MaterialApp(
10. home: MyStatefulWidget(),
11. );
12. }
13. }
15. **class** MyStatefulWidget **extends** StatefulWidget {
16. MyStatefulWidget({Key key}) : **super**(key: key);
18. @override
19. \_MyStatefulWidgetState createState() => \_MyStatefulWidgetState();
20. }
22. **class** \_MyStatefulWidgetState **extends** State<MyStatefulWidget> {
23. **int** \_count = 0;
25. Widget build(BuildContext context) {
26. **return** Scaffold(
27. appBar: AppBar(
28. title: Text('Flutter Scaffold Example'),
29. ),
30. body: Center(
31. child: Text('We have pressed the button $\_count times.'),
32. ),
33. bottomNavigationBar: BottomAppBar(
34. shape: **const** CircularNotchedRectangle(),
35. child: Container(
36. height: 50.0,
37. ),
38. ),
39. floatingActionButton: FloatingActionButton(
40. onPressed: () => setState(() {
41. \_count++;
42. }),
43. tooltip: 'Increment Counter',
44. child: Icon(Icons.add),
45. ),
46. floatingActionButtonLocation: FloatingActionButtonLocation.endDocked,
47. drawer: Drawer(
48. elevation: 20.0,
49. child: Column(
50. children: <Widget>[
51. UserAccountsDrawerHeader(
52. accountName: Text("javatpoint"),
53. accountEmail: Text("javatpoint@gmail.com"),
54. currentAccountPicture: CircleAvatar(
55. backgroundColor: Colors.yellow,
56. child: Text("abc"),
57. ),
58. ),
59. ListTile(
60. title: **new** Text("Inbox"),
61. leading: **new** Icon(Icons.mail),
62. ),
63. Divider( height: 0.1,),
64. ListTile(
65. title: **new** Text("Primary"),
66. leading: **new** Icon(Icons.inbox),
67. ),
68. ListTile(
69. title: **new** Text("Social"),
70. leading: **new** Icon(Icons.people),
71. ),
72. ListTile(
73. title: **new** Text("Promotions"),
74. leading: **new** Icon(Icons.local\_offer),
75. )
76. ],
77. ),
78. ),
79. );
80. }
81. }

**Output:**

When we run this project in the IDE, we will see the UI as the following screenshot.



If we click on the three lines that can be seen in the top left corner of the screen, we will see the drawer. The drawer can be swiped right to left or left to right. See the below image.



# Flutter Container

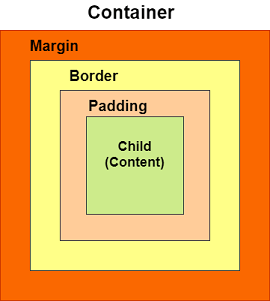
The container in Flutter is a **parent widget that can contain multiple child widgets** and manage them efficiently through width, height, padding, background color, etc. It is a widget that combines common painting, positioning, and sizing of the child widgets. It is also a class to store one or more widgets and position them on the screen according to our needs. Generally, it is similar to a box for storing contents. It allows many attributes to the user for decorating its child widgets, such as using **margin**, which separates the container with other contents.

A container widget is same as **<div>** tag in html. If this widget does not contain any child widget, it will fill the whole area on the screen automatically. Otherwise, it will wrap the child widget according to the specified height & width. It is to **note that** this widget cannot render directly without any parent widget. We can use Scaffold widget, Center widget, Padding widget, Row widget, or Column widget as its parent widget.

### Why we need a container widget in Flutter?

If we have a widget that needs some background styling may be a color, shape, or size constraints, we may try to **wrap it in a container widget**. This widget helps us to compose, decorate, and position its child widgets. If we wrap our widgets in a container, then without using any parameters, we would not notice any difference in its appearance. But if we add any properties such as color, margin, padding, etc. in a container, we can style our widgets on the screen according to our needs.

A basic container has a margin, border, and padding properties surrounding its child widget, as shown in the below image:



### Constructors of the container class

The following are the syntax of container class constructor:

1. Container({Key key,
2. AlignmentGeometry alignment,
3. EdgeInsetsGeometry padding,
4. Color color,
5. **double** width,
6. **double** height,
7. Decoration decoration,
8. Decoration foregroundDecoration,
9. BoxConstraints constraints,
10. Widget child,
11. Clip clipBehavior: Clip.none
12. });

## **Properties of Container widget**

Let us learn some of the essential properties of the container widget in detail.

**1. child:** This property is used to store the child widget of the container. Suppose we have taken a Text widget as its child widget that can be shown in the below example:

1. Container(
2. child: Text("Hello! I am in the container widget", style: TextStyle(fontSize: 25)),
3. )

**2. color:** This property is used to set the **background color of the text**. It also changes the background color of the entire container. See the below example:

1. Container(
2. color: Colors.green,
3. child: Text("Hello! I am in the container widget", style: TextStyle(fontSize: 25)),
4. )

**3. height and width:** This property is used to set the container's height and width according to our needs. By default, the container always takes the space based on its child widget. See the below code:

1. Container(
2. width: 200.0,
3. height: 100.0,
4. color: Colors.green,
5. child: Text("Hello! I am in the container widget", style: TextStyle(fontSize: 25)),
6. )

**4. margin:** This property is used to surround the e**mpty space around the container**. We can observe this by seeing white space around the container. Suppose we have used the **EdgeInsets.all(25)** that set the equal margin in all four directions, as shown in the below example:

1. Container(
2. width: 200.0,
3. height: 100.0,
4. color: Colors.green,
5. margin: EdgeInsets.all(20),
6. child: Text("Hello! I am in the container widget", style: TextStyle(fontSize: 25)),
7. )

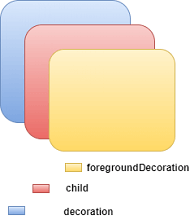
**5. padding:** This property is used to **set the distance** between the border of the container (all four directions) and its child widget. We can observe this by seeing the space between the container and the child widget. Here, we have used an EdgeInsets.all(35) that set the space between text and all four container directions:

1. Container(
2. width: 200.0,
3. height: 100.0,
4. color: Colors.green,
5. padding: EdgeInsets.all(35),
6. margin: EdgeInsets.all(20),
7. child: Text("Hello! I am in the container widget", style: TextStyle(fontSize: 25)),
8. )

**6. alignment:** This property is used to **set the position** of the child within the container. [Flutter](https://www.javatpoint.com/flutter) allows the user to align its element in various ways such as center, bottom, bottom center, topLeft, centerRight, left, right, and many more. In the below example, we are going to align its child into the bottom right position.

1. Container(
2. width: 200.0,
3. height: 100.0,
4. color: Colors.green,
5. padding: EdgeInsets.all(35),
6. margin: EdgeInsets.all(20),
7. alignment: Alignment.bottomRight,
8. child: Text("Hello! I am in the container widget", style: TextStyle(fontSize: 25)),
9. )

**7. decoration:** This property allows the developer to **add decoration on the widget**. It decorates or paint the widget behind the child. If we want to decorate or paint in front of a child, we need to use the **forgroundDecoration** parameter. The below image explains the difference between them where the foregroundDecoration covers the child and decoration paint behind the child.



The decoration property supported many parameters, such as color, gradient, background image, border, shadow, etc. It is to make sure that **we can either use the color property in a container or decoration, but not in both**. See the below code where we have added a border and shadow property to decorate the box:

1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
5. /// This Widget is the main application widget.
6. **class** MyApp **extends** StatelessWidget {
8. @override
9. Widget build(BuildContext context) {
10. **return** MaterialApp(
11. home: Scaffold(
12. appBar: AppBar(
13. title: Text("Flutter Container Example"),
14. ),
15. body: Container(
16. padding: EdgeInsets.all(35),
17. margin: EdgeInsets.all(20),
18. decoration: BoxDecoration(
19. border: Border.all(color: Colors.black, width: 4),
20. borderRadius: BorderRadius.circular(8),
21. boxShadow: [
22. **new** BoxShadow(color: Colors.green, offset: **new** Offset(6.0, 6.0),),
23. ],
24. ),
25. child: Text("Hello! I am in the container widget decoration box!!",
26. style: TextStyle(fontSize: 30)),
27. ),
28. ),
29. );
30. }
31. }

We will see the output as below screenshot:



**8. transform:** The transform property allows developers to **rotate the container**. It can rotate the container in any direction, i.e., change the container coordinate in the parent widget. In the below example, we will rotate the container in the **z-axis**.

1. Container(
2. width: 200.0,
3. height: 100.0,
4. color: Colors.green,
5. padding: EdgeInsets.all(35),
6. margin: EdgeInsets.all(20),
7. alignment: Alignment.bottomRight,
8. transform: Matrix4.rotationZ(0.1),
9. child: Text("Hello! I am in the container widget", style: TextStyle(fontSize: 25)),
10. )

**9. constraints:** This property is used when we want to **add additional constraints to the child**. It contains various constructors, such as tight, loose, expand, etc. Let's see how to use these constructors in our app:

**tight:** If we use size property in this, it will give fixed value to the child.

1. Container(
2. color: Colors.green,
3. constraints: BoxConstraints.tight(Size size)
4. : minWidth = size.width, maxWidth = size.width,
5. minHeight = size.height, maxHeight = size.height;
6. child: Text("Hello! I am in the container widget", style: TextStyle(fontSize: 25)),
7. )

**expand:** Here, we can choose the height, width, or both values to the child.

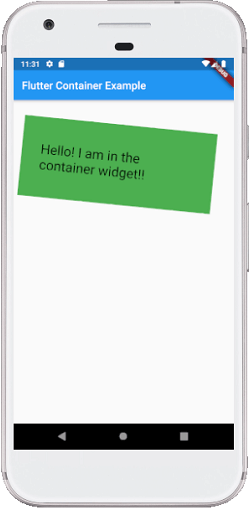
1. Container(
2. color: Colors.green,
3. constraints: BoxConstraints.expand(height: 60.0),
4. child: Text("Hello! I am in the container widget", style: TextStyle(fontSize: 25)),
5. )

Let us understand it with an example where we will try to cover most of the container properties. Open the **main.dart** file and replace it with the below code:

1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
5. /// This Widget is the main application widget.
6. **class** MyApp **extends** StatelessWidget {
8. @override
9. Widget build(BuildContext context) {
10. **return** MaterialApp(
11. home: MyContainerWidget(),
12. );
13. }
14. }
16. **class** MyContainerWidget **extends** StatelessWidget {
17. @override
18. Widget build(BuildContext context) {
19. **return** MaterialApp(
20. home: Scaffold(
21. appBar: AppBar(
22. title: Text("Flutter Container Example"),
23. ),
24. body: Container(
25. width: **double**.infinity,
26. height: 150.0,
27. color: Colors.green,
28. margin: EdgeInsets.all(25),
29. padding: EdgeInsets.all(35),
30. alignment: Alignment.center,
31. transform: Matrix4.rotationZ(0.1),
32. child: Text("Hello! I am in the container widget!!",
33. style: TextStyle(fontSize: 25)),
34. ),
35. ),
36. );
37. }
38. }

**Output**

When we run this app, it will give the following screenshot:



# Flutter Row and Column

In the previous sections, we have learned to create a simple Flutter application and its basic styling to the widgets. Now, we are going to learn how to **arrange the widgets in rows and columns on the screen**. The rows and columns are not a single widget; they are two different widgets, namely Row and Column. Here, we will integrate these two widgets together because they have similar properties that help us understand them efficiently and quickly.

Row and column are the two essential widgets in [Flutter](https://www.javatpoint.com/flutter) that allows developers to **align children horizontally and vertically according to our needs**. These widgets are very necessary when we design the application user interface in Flutter.

## **Key Points**

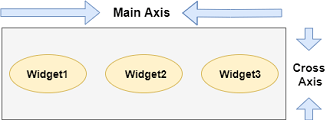
1. Row and Column widgets are the most commonly used layout patterns in the Flutter application.
2. Both may take several child widgets.
3. A child widget can also be a row or column widget.
4. We can stretch or constrain a particular children's widget.
5. Flutter also allows developers to specify how child widgets can use row and column widgets' available space.

## **Row Widget**

**This widget arranges its children in a horizontal direction on the screen**. In other words, it will expect child widgets in a horizontal array. If the child widgets need to fill the available horizontal space, we must wrap the children widgets in an Expanded widget.

A row widget does **not appear scrollable** because it displays the widgets within the visible view. So it is considered wrong if we have more children in a row which will not fit in the available space. If we want to make a scrollable list of row widgets, we need to use the ListView widget.

We can control how a row widget aligns its children based on our choice using the property **crossAxisAlignment** and **mainAxisAlignment**. The row's **cross-axis** will run **vertically**, and the **main axis** will run **horizontally**. See the below visual representation to understand it more clearly.



#### **Note: Flutter row widget has several other properties like mainAxisSize, textDirection, verticalDirection, etc. Here, we will discuss only mainAxisAlignment and crossAxisAlignment properties.**

We can align the row's children widget with the help of the following properties:

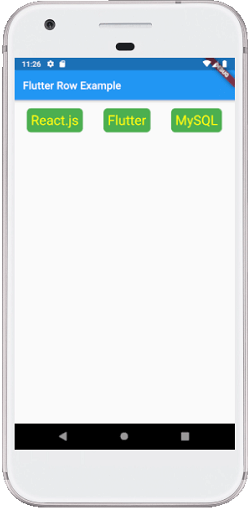
* **start:** It will place the children from the starting of the main axis.
* **end:** It will place the children at the end of the main axis.
* **center:** It will place the children in the middle of the main axis.
* **spaceBetween:** It will place the free space between the children evenly.
* **spaceAround:** It will place the free space between the children evenly and half of that space before and after the first and last children widget.
* **spaceEvenly:** It will place the free space between the children evenly and before and after the first and last children widget.

Let us understand it with the help of an example where we are going to align the content such that there is an even space around the children in a row:

1. **import** 'package:flutter/material.dart';
3. **void** main() { runApp(MyApp()); }
4. **class** MyApp **extends** StatelessWidget {
5. @override
6. Widget build(BuildContext context) {
7. **return** MaterialApp(
8. home: MyHomePage()
9. );
10. }
11. }
13. **class** MyHomePage **extends** StatefulWidget {
14. @override
15. \_MyHomePageState createState() => \_MyHomePageState();
16. }
18. **class** \_MyHomePageState **extends** State<MyHomePage> {
19. @override
20. Widget build(BuildContext context) {
21. **return** Scaffold(
22. appBar: AppBar(
23. title: Text("Flutter Row Example"),
24. ),
25. body: Row(
26. mainAxisAlignment: MainAxisAlignment.spaceEvenly,
27. children:<Widget>[
28. Container(
29. margin: EdgeInsets.all(12.0),
30. padding: EdgeInsets.all(8.0),
31. decoration:BoxDecoration(
32. borderRadius:BorderRadius.circular(8),
33. color:Colors.green
34. ),
35. child: Text("React.js",style: TextStyle(color:Colors.yellowAccent,fontSize:25),),
36. ),
37. Container(
38. margin: EdgeInsets.all(15.0),
39. padding: EdgeInsets.all(8.0),
40. decoration:BoxDecoration(
41. borderRadius:BorderRadius.circular(8),
42. color:Colors.green
43. ),
44. child: Text("Flutter",style: TextStyle(color:Colors.yellowAccent,fontSize:25),),
45. ),
46. Container(
47. margin: EdgeInsets.all(12.0),
48. padding: EdgeInsets.all(8.0),
49. decoration:BoxDecoration(
50. borderRadius:BorderRadius.circular(8),
51. color:Colors.green
52. ),
53. child: Text("MySQL",style: TextStyle(color:Colors.yellowAccent,fontSize:25),),
54. )
55. ]
56. ),
57. );
58. }
59. }

**Output:**

When we run this app, we should get the UI as the below screenshot.

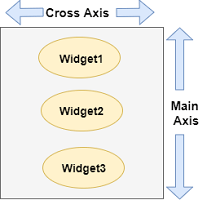


## **Column**

**This widget arranges its children in a vertical direction on the screen**. In other words, it will expect a vertical array of children widgets. If the child widgets need to fill the available vertical space, we must wrap the children widgets in an Expanded widget.

A column widget does **not appear scrollable** because it displays the widgets within the visible view. So it is considered wrong if we have more children in a column which will not fit in the available space. If we want to make a scrollable list of column widgets, we need to use the ListView Widget.

We can also control how a column widget aligns its children using the property mainAxisAlignment and crossAxisAlignment. The column's **cross-axis**will run **horizontally**, and the **main axis** will run **vertically**. The below visual representation explains it more clearly.



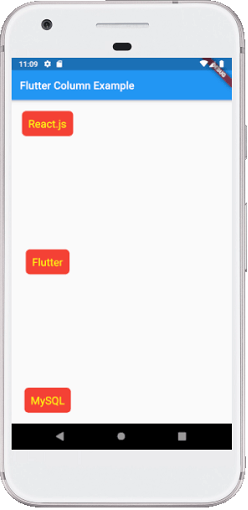
#### **Note: Column widget also aligns its content by using the same properties as we have discussed in row widget such as start, end, center, spaceAround, spaceBetween, and spaceEvenly.**

Let us understand it with the help of an example where we are going to align the content such that there is a free space between the children evenly in a column:

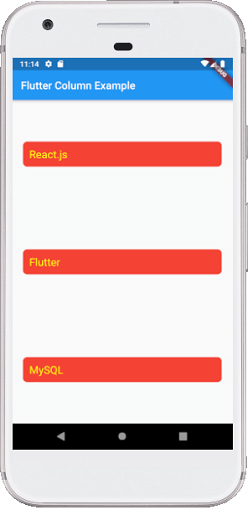
1. **import** 'package:flutter/material.dart';
3. **void** main() { runApp(MyApp()); }
4. **class** MyApp **extends** StatelessWidget {
5. @override
6. Widget build(BuildContext context) {
7. **return** MaterialApp(
8. home: MyHomePage()
9. );
10. }
11. }
13. **class** MyHomePage **extends** StatefulWidget {
14. @override
15. \_MyHomePageState createState() => \_MyHomePageState();
16. }
18. **class** \_MyHomePageState **extends** State<MyHomePage> {
19. @override
20. Widget build(BuildContext context) {
21. **return** Scaffold(
22. appBar: AppBar(
23. title: Text("Flutter Column Example"),
24. ),
25. body: Column(
26. mainAxisAlignment: MainAxisAlignment.spaceBetween,
27. children:<Widget>[
28. Container(
29. margin: EdgeInsets.all(20.0),
30. padding: EdgeInsets.all(12.0),
31. decoration:BoxDecoration(
32. borderRadius:BorderRadius.circular(8),
33. color:Colors.red
34. ),
35. child: Text("React.js",style: TextStyle(color:Colors.yellowAccent,fontSize:20),),
36. ),
37. Container(
38. margin: EdgeInsets.all(20.0),
39. padding: EdgeInsets.all(12.0),
40. decoration:BoxDecoration(
41. borderRadius:BorderRadius.circular(8),
42. color:Colors.red
43. ),
44. child: Text("Flutter",style: TextStyle(color:Colors.yellowAccent,fontSize:20),),
45. ),
46. Container(
47. margin: EdgeInsets.all(20.0),
48. padding: EdgeInsets.all(12.0),
49. decoration:BoxDecoration(
50. borderRadius:BorderRadius.circular(8),
51. color:Colors.red
52. ),
53. child: Text("MySQL",style: TextStyle(color:Colors.yellowAccent,fontSize:20),),
54. )
55. ]
56. ),
57. );
58. }
59. }

**Output:**

When we run this app, we should get the UI as the below screenshot.



**Flutter also allows developers to align the child widget with a combination of crossAxisAlignment and mainAxisAlignment for both row and column widget**. Let us take the above example of column widget where we will set mainAxisAlignment as MainAxisAlignment.spaceAround and crossAxisAlignment is CrossAxisAlignment.stretch. It will make the height of the column is equal to the height of the body. See the below screenshot.



### Drawbacks OF Row and Column Widget:

* Row widget in Flutter does not have horizontal scrolling. So if we have inserted a large number of children in a single row that cannot be fit in that row, we will see the Overflow message.
* Column widget in Flutter does not have vertical scrolling. So if we have inserted a large number of children in a single column whose total children size is not equal to the height of the screen, we will see the Overflow message.

# 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](https://www.javatpoint.com/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. |
| textDirection | It is used to specify to which direction the icon will be rendered. |

Let us understand Flutter icons using different examples.Play Vide

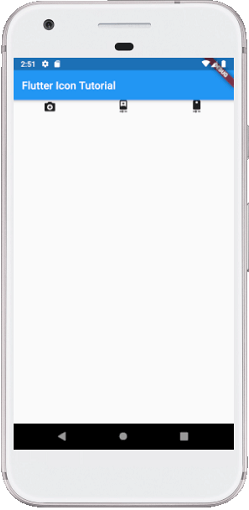
**Example 1:**

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:

1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
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. theme: ThemeData(
11. primarySwatch: Colors.blue,
12. ),
13. home: MyIconPage(),
14. );
15. }
16. }
18. **class** MyIconPage **extends** StatefulWidget {
19. @override
20. \_MyIconPageState createState() => \_MyIconPageState();
21. }
23. **class** \_MyIconPageState **extends** State<MyIconPage> {
24. @override
25. Widget build(BuildContext context) {
26. **return** Scaffold(
27. appBar: AppBar(
28. title: Text('Flutter Icon Tutorial'),
29. ),
30. body: Row(
31. mainAxisAlignment: MainAxisAlignment.spaceAround,
32. children: <Widget>[
33. Icon(Icons.camera\_enhance),
34. Icon(Icons.camera\_front),
35. Icon(Icons.camera\_rear),
36. ]),
37. );
38. }
39. }

**Output:**

When we run this project, it will show the UI similar to the following screenshot in the emulator or device we are using:



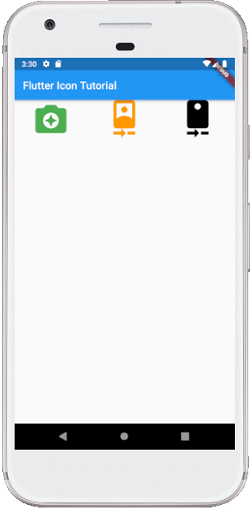
**Example 2:**

In this example, we will see how to customize the icons. Here, we will use the **size attribute** to adjust the icon size according to our needs. We will also see the **color property** to change the icon default color. So, open the main.dart file and replace it with the below code:

1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
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. theme: ThemeData(
11. primarySwatch: Colors.blue,
12. ),
13. home: MyIconPage(),
14. );
15. }
16. }
18. **class** MyIconPage **extends** StatefulWidget {
19. @override
20. \_MyIconPageState createState() => \_MyIconPageState();
21. }
23. **class** \_MyIconPageState **extends** State<MyIconPage> {
24. @override
25. Widget build(BuildContext context) {
26. **return** Scaffold(
27. appBar: AppBar(
28. title: Text('Flutter Icon Tutorial'),
29. ),
30. body: Row(
31. mainAxisAlignment: MainAxisAlignment.spaceAround,
32. children: <Widget>[
33. Icon(
34. Icons.camera\_enhance,
35. size: 70,
36. color:Colors.green
37. ),
38. Icon(
39. Icons.camera\_front,
40. size: 70,
41. color:Colors.orange
42. ),
43. Icon(
44. Icons.camera\_rear,
45. size: 70,
46. color:Colors.black
47. ),
48. ]),
49. );
50. }
51. }

**Output:**

When we run this project, it will show the UI similar to the following screenshot in the emulator or device we are using:



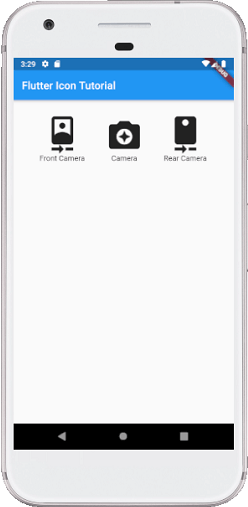
**Example 3:**

Most of the time, we have seen that the icons always contains a text below it in our application. In this example, we will see how to use the **Text widget** with the Icons widget. So, open the main.dart file and replace it with the below code:

1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
5. **class** MyApp **extends** StatelessWidget {
6. @override
7. Widget build(BuildContext context) {
8. **return** MaterialApp(
9. theme: ThemeData(
10. primarySwatch: Colors.blue,
11. ),
12. home: MyIconPage(),
13. );
14. }
15. }
17. **class** MyIconPage **extends** StatefulWidget {
18. @override
19. \_MyIconPageState createState() => \_MyIconPageState();
20. }
22. **class** \_MyIconPageState **extends** State<MyIconPage> {
23. @override
24. Widget build(BuildContext context) {
25. **return** Scaffold(
26. appBar: AppBar(
27. title: Text('Flutter Icon Tutorial'),
28. ),
29. body: Column(children: <Widget>[
30. //icon with label below it
31. Container(
32. padding: EdgeInsets.all(30),
33. child: Row(
34. mainAxisAlignment: MainAxisAlignment.spaceAround,
35. children: <Widget>[
36. Column(children: <Widget>[
37. Icon(
38. Icons.camera\_front,
39. size: 70
40. ),
41. Text('Front Camera'),
42. ]),
43. Column(children: <Widget>[
44. Icon(
45. Icons.camera\_enhance,
46. size: 70
47. ),
48. Text('Camera'),
49. ]),
50. Column(children: <Widget>[
51. Icon(
52. Icons.camera\_rear,
53. size: 70
54. ),
55. Text('Rear Camera'),
56. ]),
57. ]
58. ),
59. )
60. ],
61. )
62. );
63. }
64. }

**Output:**

When we run this project, it will show the UI similar to the following screenshot in the emulator or device we are using:



# 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:

1. assets:
2. - assets/tablet.png
3. - assets/background.png

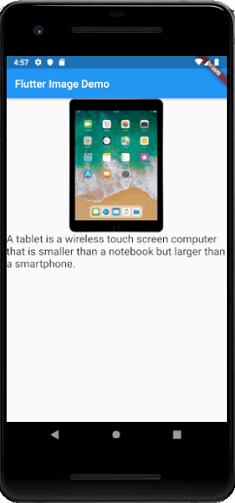
If the assets folder contains more than one image, we can include it by specifying the directory name with the **slash (/)** character at the end.

1. flutter:
2. assets:
3. - assets/

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

1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
5. **class** MyApp **extends** StatelessWidget {
6. @override
7. Widget build(BuildContext context) {
8. **return** MaterialApp(
9. home: Scaffold(
10. appBar: AppBar(
11. title: Text('Flutter Image Demo'),
12. ),
13. body: Center(
14. child: Column(
15. children: <Widget>[
16. Image.asset('assets/tablet.png'),
17. Text(
18. 'A tablet is a wireless touch screen computer that is smaller than a notebook but larger than a smartphone.',
19. style: TextStyle(fontSize: 20.0),
20. )
21. ],
22. ),
23. ),
24. ),
25. );
26. }
27. }

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



### Display images from the internet

Displaying images from the internet or network is very simple. Flutter provides a built-in method **Image.network** to work with images from a URL. The Image.network method also allows you to use some optional properties, such as height, width, color, fit, and many more. We can use the following syntax to display an image from the internet.

1. Image.network(
2. 'https://picsum.photos/250?image=9',
3. )

The Imag.Network gives one useful thing that supports animated gifs. We can use the following syntax for displaying gifs from the internet.

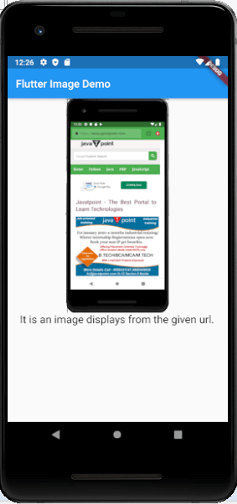
1. Image.network(
2. 'https://github.com/flutter/plugins/raw/master/packages/video\_player/doc/demo\_ipod.gif?raw=true',
3. );

Let us understand how to display an image from the network with the following example:

1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
5. **class** MyApp **extends** StatelessWidget {
6. @override
7. Widget build(BuildContext context) {
8. **return** MaterialApp(
9. home: Scaffold(
10. appBar: AppBar(
11. title: Text('Flutter Image Demo'),
12. ),
13. body: Center(
14. child: Column(
15. children: <Widget>[
16. Image.network(
17. 'https://static.javatpoint.com/tutorial/flutter/images/flutter-creating-android-platform-specific-code3.png',
18. height: 400,
19. width: 250
20. ),
21. Text(
22. 'It is an image displays from the given url.',
23. style: TextStyle(fontSize: 20.0),
24. )
25. ],
26. ),
27. ),
28. ),
29. );
30. }
31. }

**Output**

When you run the app in Android Emulator, the following screen appears. Here, you can see the image of the given url.



### Display Fade-In Images

When we display an image, it simply pops onto the screen as they are loaded. It does not assume useful between the users. To overcome this issue, the Image uses a FadeInImage widget that shows a placeholder image while the target image is loading, then fades in the new image when it loads. The FadeInImage can work with various types of images, such as local assets, in-memory, or images from the internet.

### From asset bundle

Flutter also allows us to use local assets for placeholders. To use local assets, you need to add the asset in your project pubspec.yaml file.

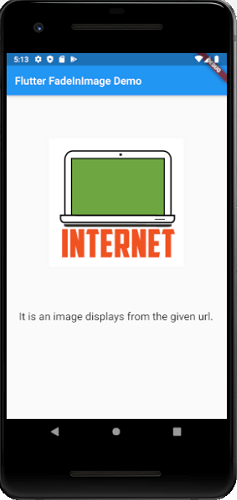
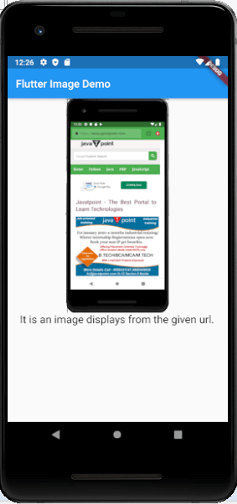
1. flutter:
2. assets:
3. - assets/loading.gif

Let us see the following example, which helps you to understand it more clearly. Open the **main.dart** file and insert the following code.

1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
5. **class** MyApp **extends** StatelessWidget {
6. @override
7. Widget build(BuildContext context) {
8. **return** MaterialApp(
9. home: Scaffold(
10. appBar: AppBar(
11. title: Text('Flutter FadeInImage Demo'),
12. ),
13. body: Center(
14. child: Column(
15. children: <Widget>[
16. FadeInImage.assetNetwork(
17. placeholder: 'assets/tablet.png',
18. image: 'https://static.javatpoint.com/tutorial/flutter/images/flutter-creating-android-platform-specific-code3.png',
19. height: 400,
20. width: 250
21. ),
22. Text(
23. 'It is an image displays from the given url.',
24. style: TextStyle(fontSize: 20.0),
25. )
26. ],
27. ),
28. ),
29. ),
30. );
31. }
32. }

**Output**

Now, run the app, it will give the laptop image (placeholder) before the image displayed from given url.

**In-Memory**

Let us understand it with the following example where FadeInImage works with In-Memory. Here, you must have to use a transparent\_image package for transparent placeholder and update the dependencies of pubspec.yaml file as below:

1. transparent\_image: ^1.0.0

Now, open the main.dart file and insert the following code. When you run the app, it will give a transparent image as a placeholder.

1. **import** 'package:flutter/material.dart';
2. **import** 'package:transparent\_image/transparent\_image.dart';
4. **void** main() => runApp(MyApp());
6. **class** MyApp **extends** StatelessWidget {
7. @override
8. Widget build(BuildContext context) {
9. **return** MaterialApp(
10. home: Scaffold(
11. appBar: AppBar(
12. title: Text('Flutter FadeInImage Demo'),
13. ),
14. body: Center(
15. child: Column(
16. children: <Widget>[
17. FadeInImage.memoryNetwork(
18. placeholder: kTransparentImage,
19. image: 'https://static.javatpoint.com/tutorial/flutter/images/flutter-creating-android-platform-specific-code3.png',
20. height: 400,
21. width: 250
22. ),
23. Text(
24. 'It is an image displays from the given url.',
25. style: TextStyle(fontSize: 20.0),
26. )
27. ],
28. ),
29. ),
30. ),
31. );
32. }
33. }

# Flutter Card

A card is a sheet used to represent the information related to each other, such as an album, a geographical location, contact details, etc. **A card in Flutter is in rounded corner shape and has a shadow**. We mainly used it to store the content and action of a single object. In this article, we are going to learn how to create a card widget in [Flutter](https://www.javatpoint.com/flutter). We will also learn how to customize the card widget.

Card creation in Flutter is very simple. We just need to call the **card constructor** and then pass a widget as child property for displaying the content and action inside the card. See the below code of simple card creation:

1. **return** Card(
2. child: Column(
3. mainAxisSize: MainAxisSize.min,
4. children: <Widget>[
5. **const** ListTile(
6. leading: Icon(Icons.album, size: 45),
7. title: Text('Sonu Nigam'),
8. subtitle: Text('Best of Sonu Nigam Song'),
9. ),
10. ],
11. ),
12. );

### Flutter Card Properties

We can customize the card using the properties. Some of the essential properties are given below:

|  |  |
| --- | --- |
| **Attribute Name** | **Descriptions** |
| borderOnForeground | It is used to paint the border in front of a child. By default, it is true. If it is false, it painted the border behind the child. |
| color | It is used to color the card's background. |
| elevation | It controls the shadow size below the card. The bigger elevation value makes the bigger shadow distance. |
| margin | It is used to customize the card's outer space. |
| shape | It is used to specify the shape of the card. |
| shadowColor | It is used to paint the shadow of a card. |
| clipBehavior | It is used to clip the content of the card. |

If we want to customize the card's size, it is required to place it in a **Container or SizedBox** widget. Here, we can set the **card's height and width** that can be shown in the below code:

1. Container(
2. width: 150,
3. height: 150,
4. child: Card(
5. ...
6. ),
7. )

Let us understand how to use a card widget in Flutter with the help of an example.

**Example:**

In this example, we will create a card widget that shows the **album information and two actions named Play and Pause**. Create a project in the IDE, open the main.dart file and replace it with the following code.

1. **import** 'package:flutter/material.dart';
3. **void** main() => runApp(MyApp());
5. /// This Widget is the main application widget.
6. **class** MyApp **extends** StatelessWidget {
7. @override
8. Widget build(BuildContext context) {
9. **return** MaterialApp(
10. home: Scaffold(
11. appBar: AppBar(title: Text('Flutter Card Example')),
12. backgroundColor: Colors.yellow,
13. body: MyCardWidget(),
14. ),
15. );
16. }
17. }
19. /// This is the stateless widget that the main application instantiates.
20. **class** MyCardWidget **extends** StatelessWidget {
21. MyCardWidget({Key key}) : **super**(key: key);
23. @override
24. Widget build(BuildContext context) {
25. **return** Center(
26. child: Container(
27. width: 300,
28. height: 200,
29. padding: **new** EdgeInsets.all(10.0),
30. child: Card(
31. shape: RoundedRectangleBorder(
32. borderRadius: BorderRadius.circular(15.0),
33. ),
34. color: Colors.red,
35. elevation: 10,
36. child: Column(
37. mainAxisSize: MainAxisSize.min,
38. children: <Widget>[
39. **const** ListTile(
40. leading: Icon(Icons.album, size: 60),
41. title: Text(
42. 'Sonu Nigam',
43. style: TextStyle(fontSize: 30.0)
44. ),
45. subtitle: Text(
46. 'Best of Sonu Nigam Music.',
47. style: TextStyle(fontSize: 18.0)
48. ),
49. ),
50. ButtonBar(
51. children: <Widget>[
52. RaisedButton(
53. child: **const** Text('Play'),
54. onPressed: () {/\* ... \*/},
55. ),
56. RaisedButton(
57. child: **const** Text('Pause'),
58. onPressed: () {/\* ... \*/},
59. ),
60. ],
61. ),
62. ],
63. ),
64. ),
65. )
66. );
67. }
68. }

**Output:**

When we run this app, it will show the UI of the screen as below screenshot.

