Code Academy
أكاديميةالبرمجة

Flutter

Eng : Eslam Fareed

Table of Content

Table of Content

Table of Content

What is Flutter ?

Flutter – a simple and high performance framework based on Dart language, provides high performance by rendering the UI directly in the operating system's canvas rather than through native framework.
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.

Installing

To Install Flutter

First We need an IDE

IDE is The program where we wrote code, i recommend **VS Code**

Then We need to install **Android Studio** to build Android Applications

Then download **Flutter SDK**.

Then identify Flutter SDK for **Enviroment Variables** related to your system and Open Cmd then type **flutter doctor** then hit enter, wait some minutes then accept **Android Licenses**, Now You are ready to work.

Files and project

First Main folder for you is **lib** folder

this folder contains everything about your dart files.

then you have to know that every application you are going to support will have a folder with native project with native code.

Then p**ubspec.yaml** this file contains all dependecies you use for your project, app version, your resourses and files you will use like (audios - videos - images - icons ...etc)

What is Widget ?

Widget is a description of a part of UI.

widgets is like tree we build, widget abover widget to make a screen

widgets could contain another widget or more than widget Like tree with branches ...etc

Widget is everthing you see

image, text, anything

There are 2 types of widgets

we will know each one next

Stateful & Stateless
so let's first talk about **State**!

The **state** of an app can very simply be defined as anything that exists in the memory of the app while the app is running. This includes all the widgets that maintain the UI of the app including the buttons, text fonts, icons, animations, etc. So now as we know what are these states let's dive directly into our main topic i.e what are these stateful and stateless widgets and how do they differ from one another.

The **State** is the information that can be read synchronously when the widget is built and might change during the lifetime of the widget.

In other words, the state of the widget is the data of the objects that its properties (parameters) are sustaining at the time of its creation (when the widget is painted on the screen). The state can also change when it is used for example when a CheckBox widget is clicked a check appears on the box.

Stateless Widgets: The widgets whose state can not be altered once they are built are called stateless widgets. These widgets are immutable once they are built i.e any amount of change in the variables, icons, buttons, or retrieving data can not change the state of the app. Below is the basic structure of a stateless widget. Stateless widget overrides the build() method and returns a widget. For example, we use Text or the Icon in our flutter application where the state of the widget does not change in the runtime. It is used when the UI depends on the information within the object itself. Other examples can be Text, RaisedButton, IconButtons.

Stateful Widgets: The widgets whose state can be altered once they are built are called stateful Widgets. These states are mutable and can be changed multiple times in their lifetime. This simply means the state of an app can change multiple times with different sets of variables, inputs, data. Below is the basic structure of a stateful widget. Stateful widget overrides the createState() method and returns a State. It is used when the UI can change dynamically. Some examples can be CheckBox, RadioButton, Form, TextField.

Classes that inherit “Stateful Widget” are immutable. But the State is mutable which changes in the runtime when the user interacts with it.

* **Simple to say That Stateless Widgets cannot change thier state, Statefull is the opposite meaning**

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.

Simple Way to understand is That Scaffold is a screen in flutter, you need to build a new screen so you must use Scaffold

let’s see code above

```
Scaffold(
   appBar: AppBar( ),
   body: Text("Welcome to Javatpoint")
);
```

Text is simply a widget that show some string to screen.

```
Text(“Hello World”)
```

We can put some styles and font to this text by using style

```
Text(
          "Hello World!",
          style: TextStyle(
            fontSize: 35,
            color: Colors.purple,
            fontWeight:
FontWeight.w700,
            fontStyle: FontStyle.italic,
            letterSpacing: 8,
            wordSpacing: 20,
            backgroundColor:
Colors.yellow,
          ),
```

Container is a widget that contains another widget to simply put some styles on it.

Like width,height, background color, some shapes ....etc

```
Container(
   width: 200.0,
   height: 100.0,
   color: Colors.green,
   margin: EdgeInsets.all(20),
   child: Text("Hello World", style: TextStyle(fontSize: 25)),
)
```

Drawer is a menu which appeares at the bar at op of screen
This menu take this look  at the top left.

This usually used to make navigations to another screens or just showen some information.

Let’s take a look at code

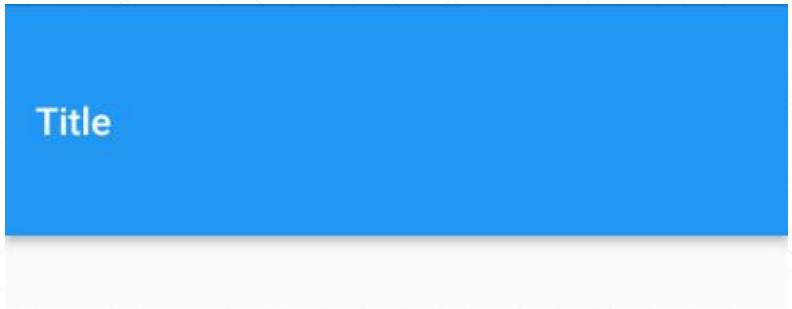
```
Scaffold(
  drawer: Drawer(child:Text(“Hello World”) )
);
```

Appbar is the bar at the top of screen take the full width of the screen and specified height but you could change height.

Appbar can be used to show application name and put some icons like search, notifications ... etc.

Let’s see how to make it

```
Scaffold(
      appBar: AppBar(
        title: const Text(Hello World),
      ),
);
```



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.

Row Widget

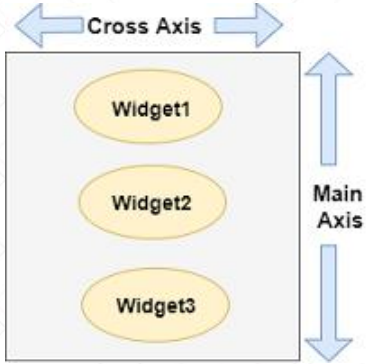
```
body: Row(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children:<Widget>[
          Container(
            margin: EdgeInsets.all(12.0),
            padding: EdgeInsets.all(8.0),
            decoration:BoxDecoration(
               borderRadius:BorderRadius.circular(8),
               color:Colors.green
            ),
            child: Text("React.js",style:        TextStyle(color:Colors.yellowAccent,fontSize:25),),
          ),
          Container(
            margin: EdgeInsets.all(15.0),
            padding: EdgeInsets.all(8.0),
            decoration:BoxDecoration(
               borderRadius:BorderRadius.circular(8),
               color:Colors.green
            ),
            child: Text("Flutter",style:          TextStyle(color:Colors.yellowAccent,fontSize:25),),
          ),
          Container(
            margin: EdgeInsets.all(12.0),
            padding: EdgeInsets.all(8.0),
            decoration:BoxDecoration(
               borderRadius:BorderRadius.circular(8),
               color:Colors.green
            ),
            child: Text("MySQL",style:        TextStyle(color:Colors.yellowAccent,fontSize:25),),
          )
        ]
      ),
```

Column Widget

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.

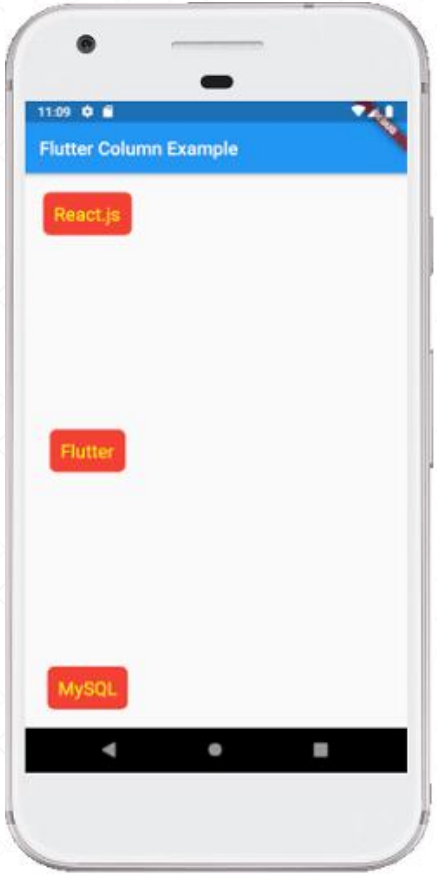
A column widget does not appear scrollable because it displays the widgets within the visible view. So it is considered wrong if we have more children in a column which will not fit in the available space. If we want to make a scrollable list of column widgets, we need to use the ListView Widget.

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.



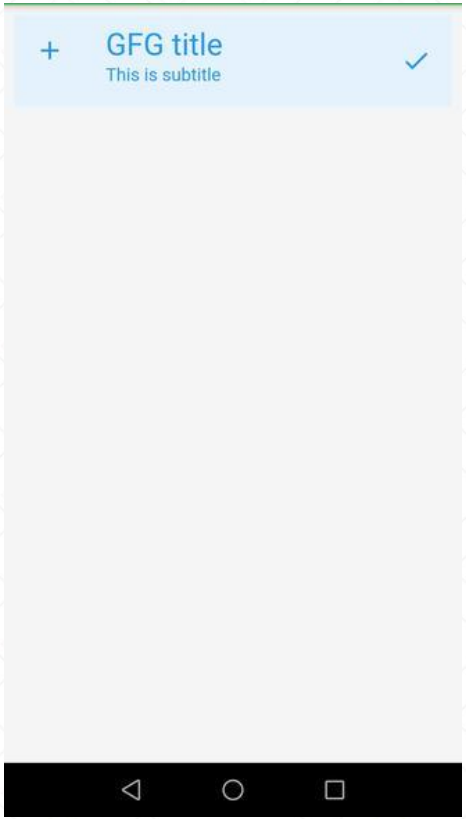
Column Widget

```
body: Column(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children:<Widget>[
          Container(
            margin: EdgeInsets.all(20.0),
            padding: EdgeInsets.all(12.0),
            decoration:BoxDecoration(
                borderRadius:BorderRadius.circular(8),
                color:Colors.red
            ),
            child: Text("React.js",style: TextStyle(color:Colors.yellowAccent,fontSize:20),),
          ),
          Container(
            margin: EdgeInsets.all(20.0),
            padding: EdgeInsets.all(12.0),
            decoration:BoxDecoration(
                borderRadius:BorderRadius.circular(8),
                color:Colors.red
            ),
            child: Text("Flutter",style: TextStyle(color:Colors.yellowAccent,fontSize:20),),
          ),
          Container(
            margin: EdgeInsets.all(20.0),
            padding: EdgeInsets.all(12.0),
            decoration:BoxDecoration(
                borderRadius:BorderRadius.circular(8),
                color:Colors.red
            ),
            child: Text("MySQL",style: TextStyle(color:Colors.yellowAccent,fontSize:20),),
          )
        ]
    ),
```



ListTile widget is used to populate a ListView in Flutter. It contains title as well as leading or trailing icons. Let’s understand this with the help of an example.

```
Container(
            color: Colors.blue[50],
            child: ListTile(
              leading: const Icon(Icons.add),
              title: const Text(
                'GFG title',
                textScaleFactor: 1.5,
              ),
              trailing: const Icon(Icons.done),
              subtitle: const Text('This is subtitle'),
              selected: true,
              onTap: () {},),)
```



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 table.png, then pubspec.yaml file is:

assets:
- assets/table.png

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

or use image from network using url :
this is an example :

```
Image.network(
'https://github.com/flutter/plugins/raw/master/packages/video_player/doc/demo_ipod.gif?raw=true',
);
```

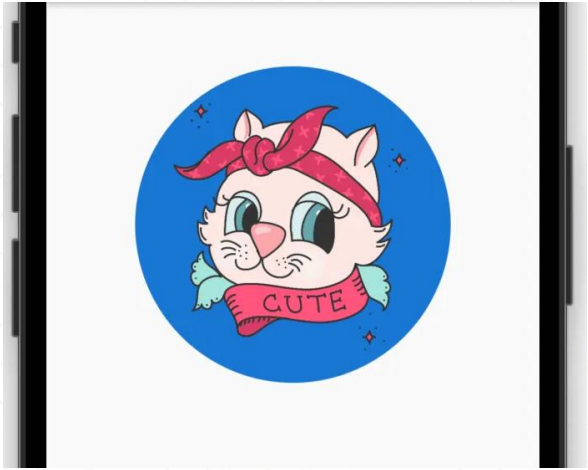
Sized box widget used to add width and height to any widget or to make a space between widgets in row or column.

SizedBox is a built-in widget in flutter SDK. It is a simple box with a specified size. It can be used to set size constraints to the child widget, put an empty SizedBox between the two widgets to get some space in between, or something else. It is somewhat similar to a Container widget with fewer properties.

```
SizedBox(
          width: 200.0,
          height: 100.0,
          child: Card(
            color: Colors.green,
            child: Center(
              child: Text(
                “Hello World”,
                style: TextStyle(color: Colors.white),
            ))))
```

To create a Circular Image in Flutter, use a widget called CircleAvatar. CircleAvatar is simply a widget used to display a user profile picture. In the absence of a user's profile picture, CircleAvatar can display the user's initials.

```
CircleAvatar(
  backgroundImage: AssetImage('assets/images/cat3.png'),
)
```



Themes are an integral part of UI for any application. Themes are used to design the fonts and colors of an application to make it more presentable. In Flutter, the Theme widget is used to add themes to an application. One can use it either for a particular part of the application like buttons and navigation bar or define it in the root of the application to use it throughout the entire app.

```
MaterialApp(
  theme: ThemeData(
   // UI
   brightness: Brightness.dark,
   primaryColor: Colors.lightBlue[800],
   accentColor: Colors.cyan[600],
   // font
   fontFamily: 'Georgia',
   //text style
   textTheme: TextTheme(
     headline1: TextStyle(fontSize: 72.0, fontWeight: FontWeight.bold),
     headline6: TextStyle(fontSize: 36.0, fontStyle: FontStyle.italic),
     bodyText2: TextStyle(fontSize: 14.0, fontFamily: 'Hind'),),));
```

We Can Use Colors we did in theme in material app like that

```
Container(
color: Theme.of(context).accentColor,
child: Text(
                'Hello!',
                style:
Theme.of(context).textTheme.headline6,
),
);
```

We have two modes light and dark mode

```
 darkTheme: ThemeData.dark(), // For Dark
theme: ThemeData.light(), // For Light
```

Wrap your widget tree inside a SingleChildScrollView:

```
//You can use SingleChildScrollView
SingleChildScrollView(
child: Column(
    children: [
                Container(width:500,height:100,color:Colors.Green),
                Container(width:500,height:100,color:Colors.Red),
 );
```

Now You Can use This To make scrollable widget

In Flutter, ListView is a scrollable list of widgets arranged linearly. It displays its children one after another in the scroll direction i.e, vertical or horizontal.

There are different types of ListViews :

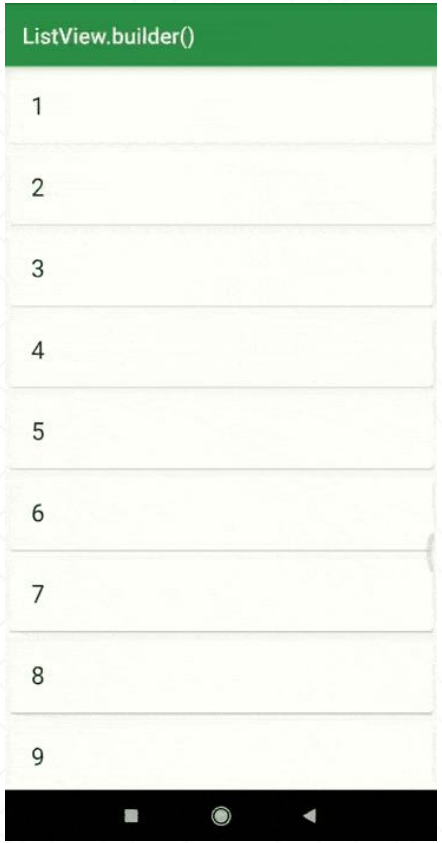
ListView
ListView.builder
ListView.separated

```
ListView(
        padding: EdgeInsets.all(20),
        children: <Widget>[
          CircleAvatar(
            maxRadius: 50,
            backgroundColor: Colors.black,
            child: Icon(Icons.person, color: Colors.white, size: 50),
          ),
          Center(
            child: Text(
              'Sooraj S Nair',
              style: TextStyle(
                fontSize: 50,
              ),
            ),
          ),
          Text(
            "Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a gallery of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum,It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using Lorem Ipsum is that it has a more-or-less normal distribution of letters, as opposed to using 'Content here, content here', making it look like readable English. Many desktop publishing packages and web page editors now use Lorem Ipsum as their default model text, and a search for 'lorem ipsum' will uncover many web sites still in their infancy. Various versions have evolved over the years, sometimes by accident, sometimes on purpose (injected humour and the like).",
            style: TextStyle(
              fontSize: 20,
            ),
          ),
        ],
      ),
```

The builder() constructor constructs a repeating list of widgets. The constructor takes two main parameters:

An itemCount for the number of repetitions for the widget to be constructed (not compulsory).
An itemBuilder for constructing the widget which will be generated ‘itemCount‘ times (compulsory).
If the itemCount is not specified, infinite widgets will be constructed by default.

```
ListView.builder(
        itemCount: 20,
        itemBuilder: (context, position) {
          return Card(
            child: Padding(
              padding: const EdgeInsets.all(20.0),
              child: Text(
                position.toString(),
                style: TextStyle(fontSize: 22.0),
              ),
            ),
          );
        },
      ),
```



The ListView.separated() constructor is used to generate a list of widgets, but in addition, a separator widget can also be generated to separate the widgets. In short, these are two intertwined list of widgets: the main list and the separator list. Unlike the builder() constructor, the itemCount parameter is compulsory here.

```
ListView.separated(
        itemBuilder: (context, position) {
          return Card(
            child: Padding(
              padding: const EdgeInsets.all(15.0),
              child: Text(
                'List Item $position',
              ),
            ),
          );
        },
        separatorBuilder: (context, position) {
          return Card(
            color: Colors.grey,
            child: Padding(
              padding: const EdgeInsets.all(5.0),
              child: Text(
                'Separator $position',
                style: TextStyle(color: Colors.white),
              ),
            ),);},
        itemCount: 20,
      ),
```

A grid view is a graphical control element used to show items in the tabular form. In this section, we are going to learn how to render items in a grid view in the Flutter application.
GridView is a widget in Flutter that displays the items in a 2-D array (two-dimensional rows and columns). As the name suggests, it will be used when we want to show items in a Grid. We can select the desired item from the grid list by tapping on them. This widget can contain text, images, icons, etc. to display in a grid layout depending on the user requirement. It is also referred to as a scrollable 2-D array of widgets. Since it is scrollable, we can specify the direction only in which it scrolls.
The grid view can be implemented in various ways, which are given below:

GridView.builder()
This property is used when we want to display data dynamically or on-demand. In other words, if the user wants to create a grid with a large (infinite) number of children, then they can use the GridView.builder() constructor with either a SliverGridDelegateWithFixedCrossAxisCount or a SliverGridDelegateWithMaxCrossAxisExtent.

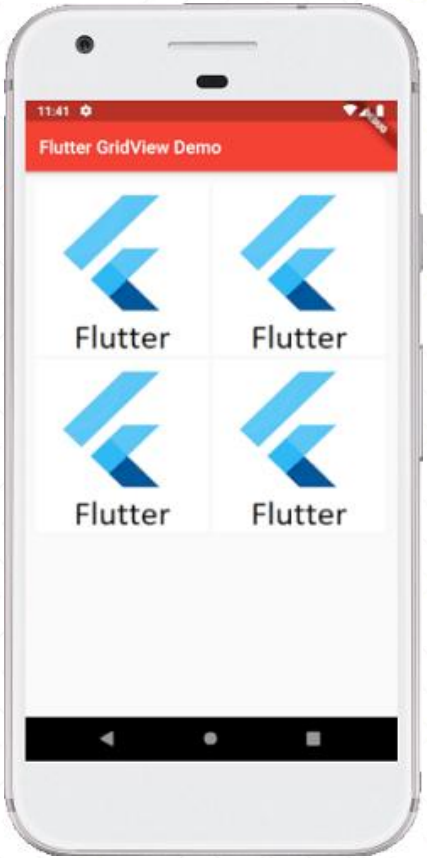
The common attributes of this widget are:

itemCount: It is used to define the amount of data to be displayed.

gridDelegate: It determines the grid or its divider. Its argument should not be null.

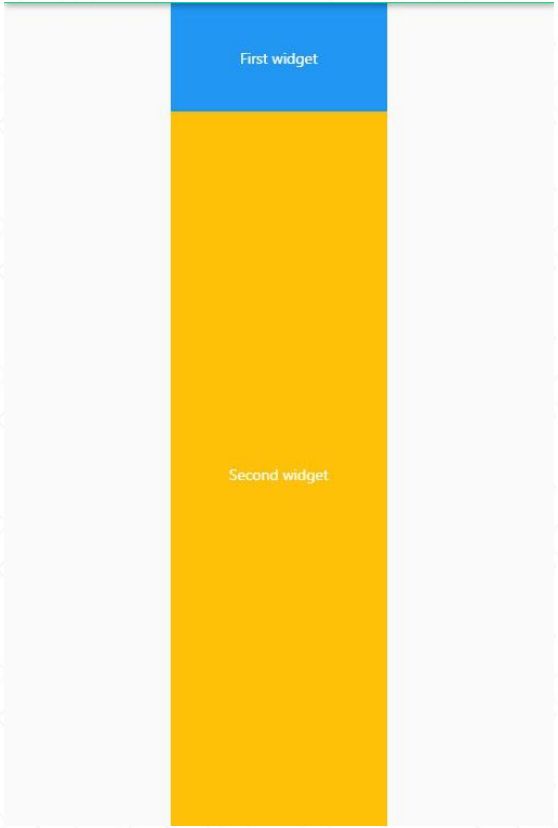
itemBuilder: It is used to create items that will be displayed on the grid view. It will be called only when the indices >= zero && indices < itemCount.

```
List<String> images = [
    "https://static.javatpoint.com/tutorial/flutter/images/flutter-logo.png",
    "https://static.javatpoint.com/tutorial/flutter/images/flutter-logo.png",
    "https://static.javatpoint.com/tutorial/flutter/images/flutter-logo.png",
    "https://static.javatpoint.com/tutorial/flutter/images/flutter-logo.png"
  ];

body: Container(
          padding: EdgeInsets.all(12.0),
          child: GridView.builder(
            itemCount: images.length,
            gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
                crossAxisCount: 2,
                crossAxisSpacing: 4.0,
                mainAxisSpacing: 4.0
            ),
            itemBuilder: (BuildContext context, int index){
              return Image.network(images[index]);
            },
          )),
    ),
```



Expanded widget in flutter comes in handy when we want a child widget or children widgets to take all the available space along the main-axis (for Row the main axis is horizontal & vertical for Column). Expanded widget can be taken as the child of Row, Column, and Flex. And in case if we don't want to give equal spaces to our children widgets we can distribute the available space as our will using flex factor. Expanded widget is similar to the Flexible widget in flutter, with its fit property set to FlexFit.tight as default. Expanded widget is basically a shorthand of Flexible widget. But if you are planning to build responsive apps or web apps, then you should definitely switch to Flexible to get more fit options.

```
Column(
      children: <Widget>[
        Container(
          child: Center(
            child: Text(
              'First widget',
              style: TextStyle(
                color: Colors.white,),),),
          color: Colors.blue,
          height: 100,
          width: 200,),
        Expanded(
          child: Container(
            child: Center(
              child: Text(
                'Second widget',
                style: TextStyle(
                  color: Colors.white,),),),
            color: Colors.amber,
            width: 200,),),],)
```



The stack is a widget in Flutter that contains a list of widgets and positions them on top of the other. In other words, the stack allows developers to overlap multiple widgets into a single screen and renders them from bottom to top. Hence, the first widget is the bottommost item, and the last widget is the topmost item

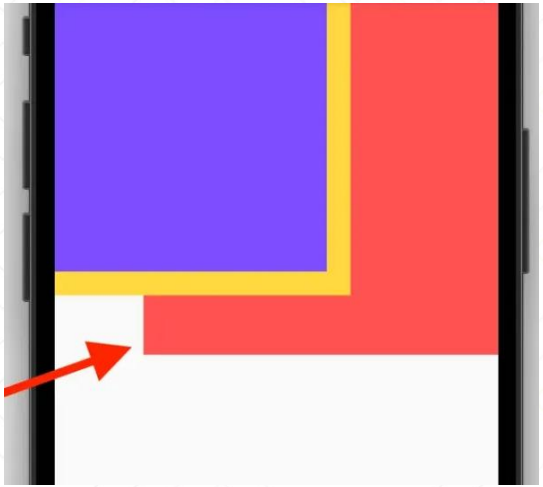
```
Stack(
alignment: Alignment.topCenter, // Center of Top
  children: <Widget>[
    // Max Size
    Container(
      color: Colors.green,
    ),
    Container(
      color: Colors.blue,
    ),
    Container(
      color: Colors.yellow,
    )
  ],
),
```

It is not the stack parameter but can be used in the stack to locate the children widgets. The following are the constructor of the positioned stack:

```
Positioned(
              top: 30,
              right: 20,
              child: Container(
                height: 100,
                width: 150,
                color: Colors.blue,
                child: Center(
                  child: Text(
                    'Middle Widget',
                    style: TextStyle(color: Colors.white, fontSize: 20),
                  ),
                ),
              ),
            ),
```

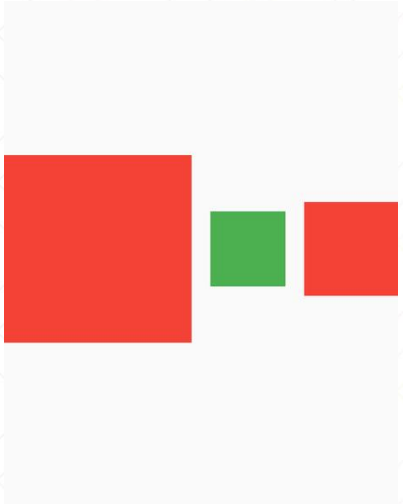
You can use the Align widget, to align a particular widget in a specific position. If you use the Align widget, your child widget will not follow the Stack's alignment and then you can place the widget anywhere on the screen using the Align's alignment property.

```
Stack(
  children: [
    Align(
      alignment: AlignmentDirectional.topEnd, //
      child: Container(
        width: 300,
        height: 300,
        color: Colors.redAccent,),),
Container(
      width: 250,
      height: 250,
      color: Colors.amberAccent,),
    Container(
      width: 230,
      height: 230,
      color: Colors.deepPurpleAccent,),])
```



Padding widget in flutter does exactly what its name says, it adds padding or empty space around a widget or a bunch of widgets. We can apply padding around any widget by placing it as the child of the Padding widget. The size of the child widget inside padding is constrained by how much space is remaining after adding empty space around. The Padding widget adds empty space around any widget by using the abstract EdgeInsetsGeometry class.

```
Padding(
            padding: EdgeInsets.fromLTRB(20, 0, 20, 0),
            child: Container(
              padding: const EdgeInsets.all(0.0),
              color: Colors.green,
              width: 80.0,
              height: 80.0,
            ), //Container
          ), //Padding
```



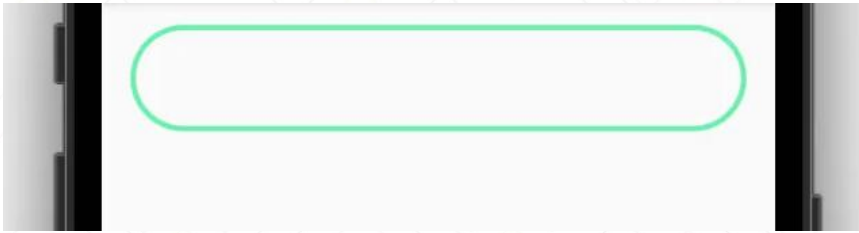
Card is a build-in widget in flutter which derives its design from Google's Material Design Library. The functionality of this widget on screen is, that it is a bland space or panel with round corners and a slight elevation on the lower side. It comes with many properties like color, shape, shadow color, etc which lets developers customize it the way they like. Below we will go through all its properties and an example to see its implementation.

```
Card(
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(15.0),
        ),
        color: Colors.red,
        elevation: 10,
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: <Widget>[
            const ListTile(
              leading: Icon(Icons.album, size: 60),
              title: Text(
                'Sonu Nigam',
                style: TextStyle(fontSize: 30.0)
              ),
              subtitle: Text(
                'Best of Sonu Nigam Music.',
                style: TextStyle(fontSize: 18.0)
              ),
            ),  ],  ),  ),
```

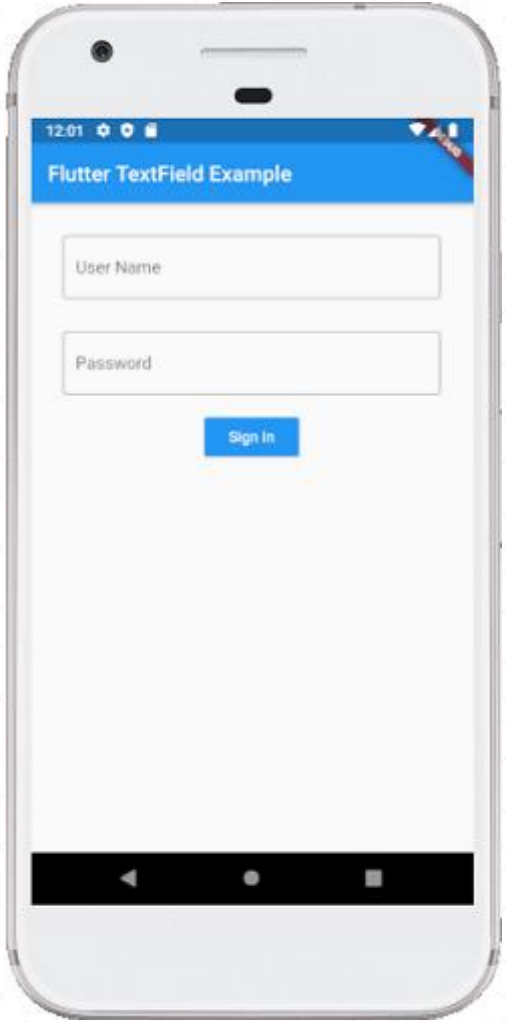
The TextField and TextFormField widgets in Flutter are the most used widgets. It is used to get user input in a variety of forms such as email, password, phone, home address, etc. But after adding the default TextField/TextFormField, sometimes you might need to customize the border of TextField/TextFormField. So in this tutorial, we will see how to add and customize TextField/TextFormField border in Flutter.

```
TextField(
  decoration: InputDecoration(
    enabledBorder: OutlineInputBorder(
      borderSide: BorderSide(
          width: 3, color: Colors.greenAccent),      ),
  ),
)
```

```
TextField(
  decoration: InputDecoration(
    enabledBorder: OutlineInputBorder(
      borderSide:
          BorderSide(width: 3, color: Colors.greenAccent),
      borderRadius: BorderRadius.circular(50.0),
    ),
  ),
)
```



```
Column(
        children: <Widget>[
          Padding(
            padding: EdgeInsets.all(15),
            child: TextField(
              decoration: InputDecoration(
                border: OutlineInputBorder(),
                labelText: 'User Name',
                hintText: 'Enter Your Name',  ),),),
          Padding(
            padding: EdgeInsets.all(15),
            child: TextField(
              obscureText: true,
              decoration: InputDecoration(
                border: OutlineInputBorder(),
                labelText: 'Password',
                hintText: 'Enter Password',  ),  ),  ),
          RaisedButton(
            textColor: Colors.white,
            color: Colors.blue,
            child: Text('Sign In'),
            onPressed: (){},  )  ],  )
```



You must Add Form With Form Key To Make a validation for text form field
Validation is checking the string entered from user, First We Use Validator attribute and then check the condition then return an error message if the condition is true we make return of string with the error.
This error will be shown for the user under text form field. This is full example :

```
return Form(
    key: _formKey,
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        TextFormField(
          // The validator receives the text that the user has entered.
          validator: (value) {
            if (value == null || value.isEmpty) {
              return 'Please enter some text';}
            return null;},),
        Padding(
          padding: const EdgeInsets.symmetric(vertical: 16.0),
          child: ElevatedButton(
            onPressed: () {
              // Validate returns true if the form is valid, or false otherwise.
              if (_formKey.currentState!.validate()) {
                // If the form is valid, display a snackbar. In the real world,
                // you'd often call a server or save the information in a database.
                ScaffoldMessenger.of(context).showSnackBar(
                  const SnackBar(content: Text('Processing Data')),);}},
            child: const Text('Submit'),),),],),);
```

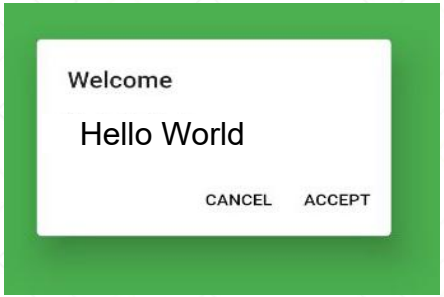
The dialog is a type of widget which comes on the window or the screen which contains any critical information or can ask for any decision. When a dialog box is popped up all the other functions get disabled until you close the dialog box or provide an answer. We use a dialog box for a different type of condition such as an alert notification, or simple notification in which different options are shown, or we can also make a dialog box that can be used as a tab for showing the dialog box.

Types of dialogs in a flutter

- AlertDialog
- SimpleDialog
- showDialog

Alert dialog tells the user about any condition that requires any recognition. The alert dialog contains an optional title and an optional list of actions. We have different no of actions as our requirements. Sometimes the content is too large compared to the screen size so for resolving this problem we may have to use the expanded class.

```
AlertDialog(
        title: Text('Welcome'),          // To display the title it is optional
        content: Text(Hello World),   // Message which will be pop up on the screen
                                      // Action widget which will provide the user to acknowledge the choice
          actions: [
          FlatButton(                 // FlatButton widget is used to make a text to work like a button
            textColor: Colors.black,
            onPressed: () {},         // function used to perform after pressing the button
            child: Text('CANCEL'),
          ),
          FlatButton(
            textColor: Colors.black,
            onPressed: () {},
            child: Text('ACCEPT'),
          ),],),
```



A simple dialog allows the user to choose from different choices. It contains the title which is optional and presented above the choices. We can show options by using the padding also. Padding is used to make a widget more flexible.

```
SimpleDialog(
            title:const Text('Hello World'),
            children: <Widget>[
              SimpleDialogOption(
                  onPressed: () { },
                    child:const Text('Option 1'),
                ),
              SimpleDialogOption(
                onPressed: () { },
              child: const Text('Option 2'),
            ),
      ],
  ),
```

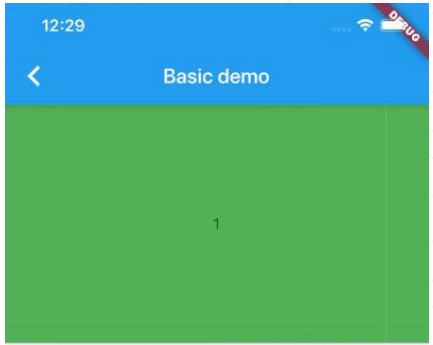
It basically used to change the current screen of our app to show the dialog popup. You must call before the dialog popup. It exits the current animation and presents a new screen animation. We use this dialog box when we want to show a tab that will popup any type of dialog box, or we create a front tab to show the background process.

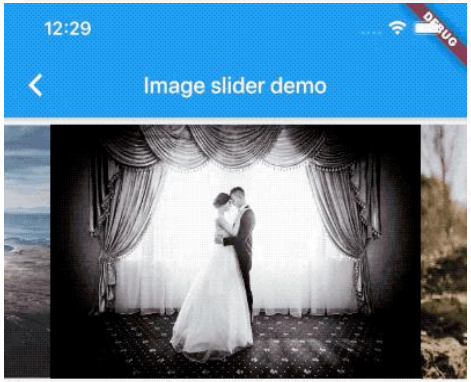
```
showDialog(
        context: context,
        builder: (BuildContext context) {
                return Expanded(
                  child: AlertDialog(
                    title: Text('Welcome'),
                    content: Text('GeeksforGeeks'),
                    actions: [
                      FlatButton(
                        textColor: Colors.black,
                        onPressed: () {},
                        child: Text('CANCEL'),),
                      FlatButton(
                        textColor: Colors.black,
                        onPressed: () {},
                        child: Text('ACCEPT'),),],),);},);
```

A carousel slider widget. is a package : https://pub.dev/packages/carousel_slider

It’s like a slider for items scrolling horizontal or vertical

```
CarouselSlider(
  options: CarouselOptions(height: 400.0),
  items: [1,2,3,4,5].map((i) {
    return Builder(
      builder: (BuildContext context) {
        return Container(
          width: MediaQuery.of(context).size.width,
          margin: EdgeInsets.symmetric(horizontal: 5.0),
          decoration: BoxDecoration(
            color: Colors.amber
          ),
          child: Text('text $i', style: TextStyle(fontSize: 16.0),)
        );
      },
    );
  }).toList(),
)
```





A simple toggle switch widget. It can be fully customized with desired icons, width, colors, text, corner radius, animation etc. It also maintains selection state.
it’s a package : https://pub.dev/packages/toggle_switch

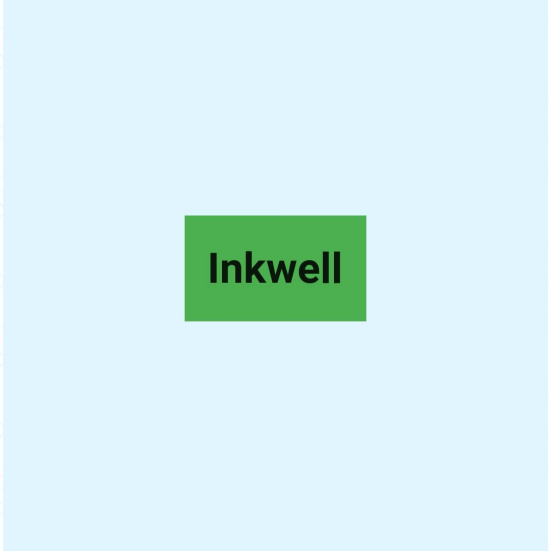
```
ToggleSwitch(
  initialLabelIndex: 0,
  totalSwitches: 3,
  labels: ['America', 'Canada', 'Mexico'],
  onToggle: (index) {
    print('switched to: $index');
  },
),
```

It’s a package for simply show the svg icons
https://pub.dev/packages/flutter_svg

```
final String assetName = 'assets/image.svg';
final Widget svg = SvgPicture.asset(
  assetName,
  semanticsLabel: 'Acme Logo'
);
```

InkWell is the material widget in flutter. It responds to the touch action as performed by the user. Inkwell will respond when the user clicks the button. There are so many gestures like double-tap, long press, tap down, etc. Below are the so many properties of this widget. We can set the radius of the inkwell widget using radius and also border-radius using borderRadius. We can give the splash color using splashColor and can do a lot of things.

```
InkWell(
            onTap: () {},
            onLongPress: () {},
            child: Container(
                color: Colors.green,
                width: 120,
                height: 70,
                child: Center(
                    child: Text(
                      'Inkwell',
                      textScaleFactor: 2,
                      style: TextStyle(fontWeight: FontWeight.bold),
                    ))),
          ),
```



Gestures are primarily a way for a user to interact with a mobile (or any touch based device) application. Gestures are generally defined as any physical action / movement of a user in the intention of activating a specific control of the mobile device. Gestures are as simple as tapping the screen of the mobile device to more complex actions used in gaming applications.

Flutter provides an excellent support for all type of gestures through its exclusive widget, GestureDetector. GestureDetector is a non-visual widget primarily used for detecting the user's gesture. To identify a gesture targeted on a widget, the widget can be placed inside GestureDetector widget. GestureDetector will capture the gesture and dispatch multiple events based on the gesture.

Some of the gestures and the corresponding events are given below −

onTapDown
onTapUp
onTap
onTapCancel
onDoubleTap
onLongPress

onVerticalDragStart
onVerticalDragUpdate
onVerticalDragEnd
onHorizontalDragStart
onHorizontalDragUpdate
onHorizontalDragEnd
onPanStart
onPanUpdate
onPanEnd

```
body: Center(
   child: GestureDetector(
      onTap: () { },
      child: Text( 'Hello World', )
   )
),
```

Here We Make a tap action in text widget

Components is a way that make your code more usable and easy to edit.

we make a new file named with your customized widget that you want to make .

let’s clear that point with an example

if you want to make a login screen so you want to make textformfield twice for email/phone and password .

so what if you make a class for both and use it with instance of this class or object

Let’s see this code example

```
class DefaultTextField extends StatefulWidget {
  DefaultTextField(
      {super.key, this.controller, this.icon, this.isEmail, this.label});
  TextEditingController? controller;
  bool? isEmail;
  String? label;
  IconData? icon;
  @override
  State<DefaultTextField> createState() => _DefaultTextFieldState();
}
class _DefaultTextFieldState extends State<DefaultTextField> {
  bool isVisible = true;
```

in this example we make a class to be usable for our ui

```
@override
  Widget build(BuildContext context) {
    return Container(
      margin: const EdgeInsets.all(20),
      child: TextFormField(
        controller: widget.controller,
        keyboardType: TextInputType.emailAddress,
        obscureText: !isVisible,
        obscuringCharacter: "*",
        decoration: InputDecoration(
          labelText: widget.label,
          prefixIcon: Icon(widget.icon),
          suffixIcon: widget.isEmail!? Container(width:
1,)
              : IconButton(
                  onPressed: () {
                    isVisible = !isVisible;
                    setState(() {});},
                  icon: isVisible
                      ? const Icon(Icons.visibility)
                      : const Icon(Icons.visibility_off)),
          border: OutlineInputBorder(
            borderRadius: BorderRadius.circular(10),
            borderSide: const BorderSide(color:
Colors.red),),),),);}}
```

This package for picking image from gallery you could check it from here :
https://pub.dev/packages/image_picker
or this package : https://pub.dev/packages/file_picker
let's see an example :

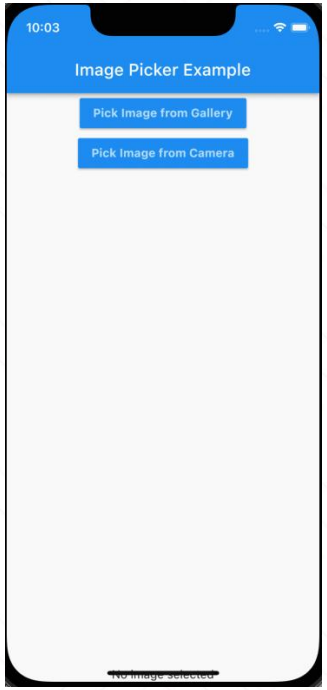
First open pubspec.yaml file and add package image_picker: ^1.0.4 . This is the package that will provide us with methods to access our gallery and camera.
Now, once the package is installed, we would need to make some changes to our an iOS and Android config files. For this article purpose, I am using iOS simulator.
Add the following keys to your Info.plist file, located in <project root>/ios/Runner/Info.plist:

NSPhotoLibraryUsageDescription - describe why your app needs permission for the photo library. This is called Privacy - Photo Library Usage Description in the visual editor.
NSCameraUsageDescription - describe why your app needs access to the camera. This is called Privacy - Camera Usage Description in the visual editor.

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("Image Picker Example"),),
      body: Center(
        child: Column(
          children: [
            MaterialButton(
              color: Colors.blue,
              child: const Text(
                "Pick Image from Gallery",
                style: TextStyle(
                  color: Colors.white70, fontWeight: FontWeight.bold)),
              onPressed: () {}),
            MaterialButton(
              color: Colors.blue,
              child: const Text(
                "Pick Image from Camera",
                style: TextStyle(
                    color: Colors.white70, fontWeight: FontWeight.bold)),
              onPressed: () {}),],),));}
```

Put this code in press button

```
File? image;
Future pickImage() async {
    try {
      final image = await ImagePicker().pickImage(source:
ImageSource.gallery);
if(image == null) return;
final imageTemp = File(image.path);
setState(() => this.image = imageTemp);
    } on PlatformException catch(e) {
      print('Failed to pick image: $e');
    }}
```



In some situations, you want to update the display of an app when the user rotates the screen from portrait mode to landscape mode. For example, the app might show one item after the next in portrait mode, yet put those same items side-by-side in landscape mode.

In Flutter, you can build different layouts depending on a given Orientation. In this example, build a list that displays two columns in portrait mode and three columns in landscape mode using the following steps:

1. Build a GridView with two columns.
2. Use an OrientationBuilder to change the number of columns.

1 -
```
return GridView.count(
  // A list with 2 columns
  crossAxisCount: 2,
  // ...
);
```

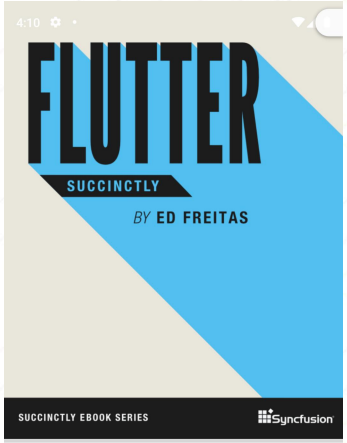
2 -
```
body: OrientationBuilder(
  builder: (context, orientation) {
    return GridView.count(
      // Create a grid with 2 columns in portrait mode,
      // or 3 columns in landscape mode.
      crossAxisCount: orientation == Orientation.portrait ? 2 : 3,
    );
  },
),
```

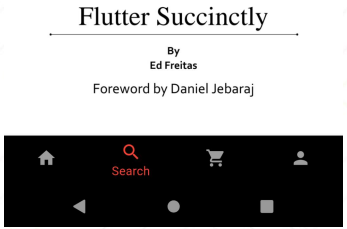
How to show pdf file in your screen and make some actions on it.

Best Package to solve this situation is syncfusion_flutter_pdfviewer You could check it from here :
https://pub.dev/packages/syncfusion_flutter_pdfviewer

Let’s see this example

```
SfPdfViewer.network(
        'https://cdn.syncfusion.com/content/PDFViewer/flutter-succinctly.pdf')
```



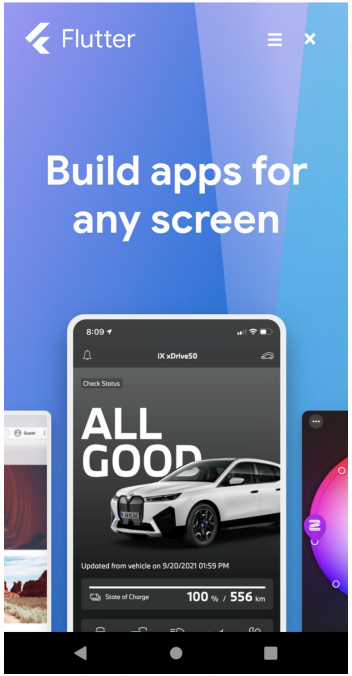


This package is so simple, used to view a web page in your application

and you could check it from here : https://pub.dev/packages/webview_flutter

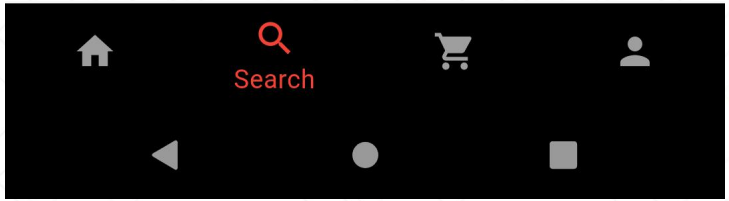
```
WebViewController? controller;
@override
void initState() {
  controller = WebViewController()
    ..setJavaScriptMode(JavaScriptMode.unrestricted)
    ..setBackgroundColor(const Color(0x00000000))
    ..setNavigationDelegate(
      NavigationDelegate(
        onProgress: (int progress) {
          // Update loading bar.},
        onPageStarted: (String url) {},
        onPageFinished: (String url) {},
        onWebResourceError: (WebResourceError error) {},
        onNavigationRequest: (NavigationRequest request) {
          // return NavigationDecision.prevent;
          if (request.url.startsWith('https://www.youtube.com/')) {
            return NavigationDecision.prevent;}
          return NavigationDecision.navigate;},),)
    ..loadRequest(Uri.parse('https://flutter.dev'));
  super.initState();}
```

```
WebViewWidget(
        controller: controller!,
    ),
```



Bottom Naviagtion Bar come bottom of the screen
and used to navigate between screens so easly and has some beautiful UI

Let’s See this full exmaple :



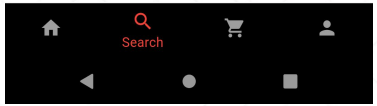
First We Must Make a List of screens that will be shown using bottom navitaion bar

```
List<Widget> screens = [
    HomeScreen(),
    PdfScreen(),
    OrientationScreen(),
    ImagePickerScreen()
  ];
```

Then Make List of Bottom Navigation Bar Items and this means Icon and label for each item in the bottom nav bar

```
List<BottomNavigationBarItem> items = [
    BottomNavigationBarItem(icon: Icon(Icons.home), label: "Home"),
    BottomNavigationBarItem(icon: Icon(Icons.search), label: "Search"),
    BottomNavigationBarItem(icon: Icon(Icons.shopping_cart), label: "Cart"),
    BottomNavigationBarItem(icon: Icon(Icons.person), label: "Profile"),
  ];
```

Next You must make a variable to detect the changes between items to check if the item index 0 shown so you will show screen index 0 from it's list
let's see how to make that :



Final Result

```
int i = 0; // this is the variable used to check any change
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: screens[i], // here we use this variable to show the screen related to item tapped
    bottomNavigationBar: BottomNavigationBar(
      type: BottomNavigationBarType.fixed, // this used to make a new type of this bottom nav bar
      selectedItemColor: Colors.red, // color for selected item
      unselectedItemColor: Colors.grey, // color for uns
      backgroundColor: Colors.black,// background color of the whole bottom nav bar
      showUnselectedLabels: false,// used to hide label when item is not selected
      currentIndex: i, // this is the variable we made above to detect any change
      onTap: (value) { // this function used to listen to and tap user will make and save the new index //tapped to the
variable we made above
        i = value;
        setState(() {});},
      items: items,),);} // this for bottom nav bar items we mmade before
```

You could use any package to make bottom nav bar for more beauti animtion or Ui

Like This :

https://pub.dev/packages/persistent_bottom_nav_bar
https://pub.dev/packages/animated_bottom_navigation_bar
https://pub.dev/packages/floating_bottom_navigation_bar
https://pub.dev/packages/circular_bottom_navigation
https://pub.dev/packages/scroll_bottom_navigation_bar

Screen Util

A flutter plugin for adapting screen and font size.Let your UI display a reasonable layout on different screen sizes!

Note: This plugin is still under development, and some APIs might not be available yet.

```
@override
  Widget build(BuildContext context) {
    //Set the fit size (Find your UI design, look at the dimensions of the device screen and fill it in,unit in dp)
    return ScreenUtilInit(
      designSize: const Size(360, 690),
      minTextAdapt: true,
      splitScreenMode: true,
      // Use builder only if you need to use library outside ScreenUtilInit context
      builder: (_ , child) {
        return MaterialApp(
          debugShowCheckedModeBanner: false,
          title: 'First Method',
          // You can use the library anywhere in the app even in theme
          theme: ThemeData(
            primarySwatch: Colors.blue,
          ),
          home: child,
        );
      },
      child: const HomePage(title: 'First Method'),);}
```

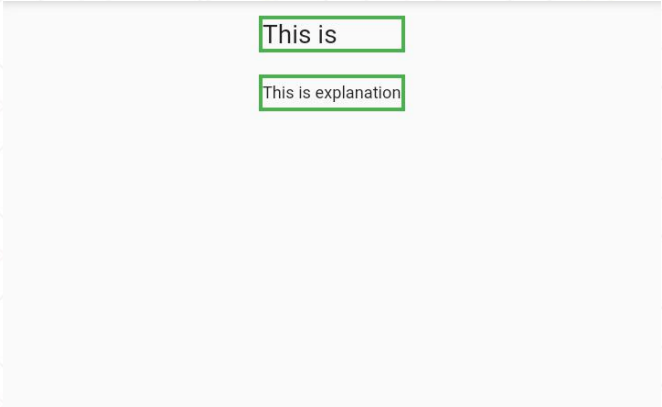
```
ScreenUtil().setWidth(540)  (dart sdk>=2.6 : 540.w) //Adapted to screen width
ScreenUtil().setHeight(200) (dart sdk>=2.6 : 200.h) //Adapted to screen height , under normal circumstances, the height still uses x.w
ScreenUtil().radius(200)    (dart sdk>=2.6 : 200.r)    //Adapt according to the smaller of width or height
ScreenUtil().setSp(24)      (dart sdk>=2.6 : 24.sp) //Adapter font
12.sm   //return min(12,12.sp)

ScreenUtil().pixelRatio       //Device pixel density
ScreenUtil().screenWidth   (dart sdk>=2.6 : 1.sw)    //Device width
ScreenUtil().screenHeight  (dart sdk>=2.6 : 1.sh)    //Device height
ScreenUtil().bottomBarHeight  //Bottom safe zone distance, suitable for buttons with full screen
ScreenUtil().statusBarHeight  //Status bar height , Notch will be higher
ScreenUtil().textScaleFactor  //System font scaling factor

ScreenUtil().scaleWidth //The ratio of actual width to UI design
ScreenUtil().scaleHeight //The ratio of actual height to UI design

ScreenUtil().orientation  //Screen orientation
0.2.sw  //0.2 times the screen width
0.5.sh  //50% of screen height
20.setVerticalSpacing  // SizedBox(height: 20 * scaleHeight)
20.horizontalSpace  // SizedBox(height: 20 * scaleWidth)
const RPadding.all(8)   // Padding.all(8.r) - take advantage of const key word
EdgeInsets.all(10).w    //EdgeInsets.all(10.w)
REdgeInsets.all(8)       // EdgeInsets.all(8.r)
EdgeInsets.only(left:8,right:8).r // EdgeInsets.only(left:8.r,right:8.r).
BoxConstraints(maxWidth: 100, minHeight: 100).w     //BoxConstraints(maxWidth: 100.w, minHeight: 100.w)
Radius.circular(16).w          //Radius.circular(16.w)
BorderRadius.all(Radius.circular(16)).w
```

Fitted Box

Flutter is all about widgets. There are many widgets that are used for positioning and styling of text. In this article, we'll learn about the FittedBox widget. FittedBox is a very useful widget that scales and positions its child within itself according to fit and alignment. Consider an app, in which, you have to take input from the user and in a certain scenario, the user enters a large input that overflows and scatters other widgets. As many of the widgets are dynamic, which means they can grow and shrink in size, according to their child widget's size. So, in this case, the user interface wouldn't be adaptive. In order to overcome this problem, we can use the FittedBox widget.

FittedBox restricts its child widgets from growing its size beyond a certain limit. It re-scales them according to the size available. For instance, if the text is displayed inside a container, and the text is to be entered by the user. If the user enters a large string of text, then the container would grow beyond its allocated size. But, if we wrap it with FittedBox, then it would fit the text according to the size available. For large string, it would shrink its size, hence would fit in the container.

```
Container(
            decoration: BoxDecoration(
              border: Border.all(width: 2,
              color: Colors.green)),
            child: Text('This is explanation'),
            width: 80,
            height: 20,),

Container(
            decoration: BoxDecoration(
              border: Border.all(width: 2,
              color: Colors.green),),
            child: FittedBox(
              child: Text('This is explanation')),
            width: 80,
            height: 20,)
```



Navigate Between Screens

Most apps contain several screens for displaying different types of information. For example, an app might have a screen that displays products. When the user taps the image of a product, a new screen displays details about the product.

In Flutter, screens and pages are called routes. The remainder of this recipe refers to routes.

In Android, a route is equivalent to an Activity. In iOS, a route is equivalent to a ViewController. In Flutter, a route is just a widget.

This recipe uses the Navigator to navigate to a new route.

The next few sections show how to navigate between two routes, using these steps:

- Create two routes.
- Navigate to the second route using Navigator.push().
- Return to the first route using Navigator.pop().

First, create two routes to work with. Since this is a basic example, each route contains only a single button. Tapping the button on the first route navigates to the second route. Tapping the button on the second route returns to the first route.

First, set up the visual structure:

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Route'),
      ),
      body: Center(
        child: ElevatedButton(
          child: const Text('Open route'),
          onPressed: () {
            // Navigate to second route when tapped.},),),);}}
```

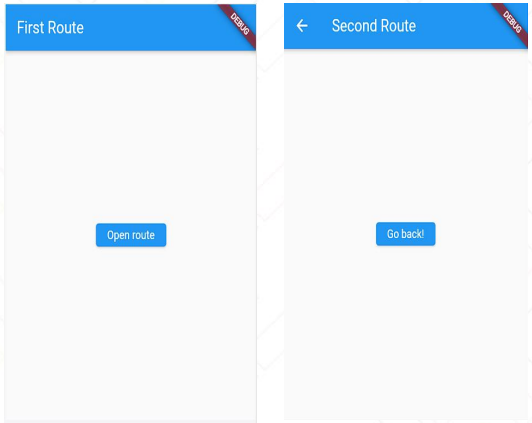
```
class SecondRoute extends StatelessWidget {
  const SecondRoute({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Second Route'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Navigate back to first route when tapped.},
          child: const Text('Go back!'),),),);}}
```

To switch to a new route, use the Navigator.push() method. The push() method adds a Route to the stack of routes managed by the Navigator. Where does the Route come from? You can create your own, or use a MaterialPageRoute, which is useful because it transitions to the new route using a platform-specific animation.

In the build() method of the FirstRoute widget, update the onPressed() callback:

```
// Within the `FirstRoute` widget
onPressed: () {
  Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => const
SecondRoute()),
  );
}
```

```
// Within the SecondRoute widget
onPressed: () {
  Navigator.pop(context);
}
```



Navigate By Name

In the Navigate to a new screen and back recipe, you learned how to navigate to a new screen by creating a new route and pushing it to the Navigator.

However, if you need to navigate to the same screen in many parts of your app, this approach can result in code duplication. The solution is to define a named route, and use the named route for navigation.

To work with named routes, use the Navigator.pushNamed() function. This example replicates the functionality from the original recipe, demonstrating how to use named routes using the following steps:

- Create two screens.
- Define the routes.
- Navigate to the second screen using Navigator.pushNamed().
- Return to the first screen using Navigator.pop().

First, create two screens to work with. The first screen contains a button that navigates to the second screen. The second screen contains a button that navigates back to the first.

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Screen'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Navigate to the second screen when tapped.
          },
          child: const Text('Launch screen'),
        ),
      ),
    );
  }
}
```

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Second Screen'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Navigate back to first screen when tapped.
          },
          child: const Text('Go back!'),
        ),
      ),
    );
  }
}
```

Next, define the routes by providing additional properties to the MaterialApp constructor: the initialRoute and the routes themselves.

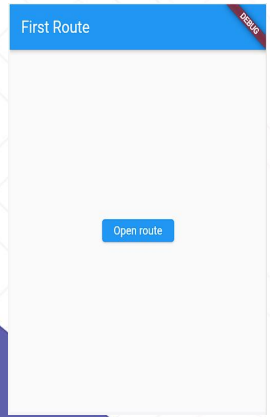
The initialRoute property defines which route the app should start with. The routes property defines the available named routes and the widgets to build when navigating to those routes.

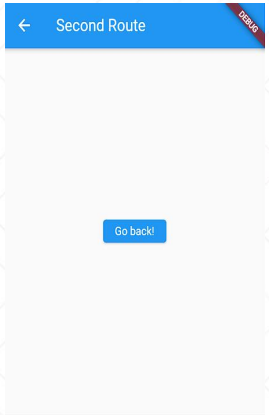
```
MaterialApp(
  title: 'Named Routes Demo',
  // Start the app with the "/" named route. In this case, the app starts
  // on the FirstScreen widget.
  initialRoute: '/',
  routes: {
    // When navigating to the "/" route, build the FirstScreen widget.
    '/': (context) => const FirstScreen(),
    // When navigating to the "/second" route, build the SecondScreen widget.
    '/second': (context) => const SecondScreen(),
  },
)
```

With the widgets and routes in place, trigger navigation by using the Navigator.pushNamed() method. This tells Flutter to build the widget defined in the routes table and launch the screen.

In the build() method of the FirstScreen widget, update the onPressed() callback:

```
// Within the `FirstScreen` widget
onPressed: () {
  // Navigate to the second screen using a named route.
  Navigator.pushNamed(context, '/second');
}
```






```
// Within the SecondScreen widget
onPressed: () {
  // Navigate back to the first screen by popping the current route
  // off the stack.
  Navigator.pop(context);
}
```

Often, you not only want to navigate to a new screen, but also pass data to the screen as well. For example, you might want to pass information about the item that's been tapped.

Remember: Screens are just widgets. In this example, create a list of todos. When a todo is tapped, navigate to a new screen (widget) that displays information about the todo. This recipe uses the following steps:

- Define a todo class.
- Display a list of todos.
- Create a detail screen that can display information about a todo.
- Navigate and pass data to the detail screen.

First, you need a simple way to represent todos. For this example, create a class that contains two pieces of data: the title and description.

```
class Todo {
  final String title;
  final String description;

  const Todo(this.title, this.description);
}
```

Second, display a list of todos. In this example, generate 20 todos and show them using a ListView. For more information on working with lists, see the Use lists recipe.

```
final todos = List.generate(
  20,
  (i) => Todo(
    'Todo $i',
    'A description of what needs to be done for Todo $i',
  ),
);
```

Display the list of todos using a ListView

```
ListView.builder(
  itemCount: todos.length,
  itemBuilder: (context, index) {
    return ListTile(
      title: Text(todos[index].title),
    );
  },
),
```

For this, we create a StatelessWidget. We call it TodosScreen. Since the contents of this page won't change during runtime, we'll have to require the list of todos within the scope of this widget.

We pass in our ListView.builder as body of the widget we're returning to build(). This'll render the list on to the screen for you to get going!

```
class TodosScreen extends StatelessWidget {
  // Requiring the list of todos.
  const TodosScreen({super.key, required this.todos});
  final List<Todo> todos;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Todos'),),
      //passing in the ListView.builder
      body: ListView.builder(
        itemCount: todos.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(todos[index].title),);},),);}}
```

Now, create the second screen. The title of the screen contains the title of the todo, and the body of the screen shows the description.

Since the detail screen is a normal StatelessWidget, require the user to enter a Todo in the UI. Then, build the UI using the given todo.

```
class DetailScreen extends StatelessWidget {
  // In the constructor, require a Todo.
  const DetailScreen({super.key, required this.todo});
  // Declare a field that holds the Todo.
  final Todo todo;
  @override
  Widget build(BuildContext context) {
    // Use the Todo to create the UI.
    return Scaffold(
      appBar: AppBar(
        title: Text(todo.title),),
      body: Padding(
        padding: const EdgeInsets.all(16),
        child: Text(todo.description),),);}}
```

Return Data from Other Screen

In some cases, you might want to return data from a new screen. For example, say you push a new screen that presents two options to a user. When the user taps an option, you want to inform the first screen of the user's selection so that it can act on that information.

You can do this with the Navigator.pop() method using the following steps:

- Define the home screen
- Add a button that launches the selection screen
- Show the selection screen with two buttons
- When a button is tapped, close the selection screen
- Show a snackbar on the home screen with the selection

The home screen displays a button. When tapped, it launches the selection screen.

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Returning Data Demo'),
      ),
      // Create the SelectionButton widget in the next
step.
      body: const Center(
        child: SelectionButton(),
      ),
    );
  }
}
```

Now, create the SelectionButton, which does the following:

- Launches the SelectionScreen when it’s tapped.
- Waits for the SelectionScreen to return a result.

```
class SelectionButton extends StatefulWidget {
  const SelectionButton({super.key});

  @override
  State<SelectionButton> createState() =>
_SelectionButtonState();
}

class _SelectionButtonState extends
State<SelectionButton> {
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () {
        _navigateAndDisplaySelection(context);
      },
      child: const Text('Pick an option, any option!'),
    );
  }
```

```
Future<void> _navigateAndDisplaySelection(BuildContext
context) async {
    // Navigator.push returns a Future that completes after calling
    // Navigator.pop on the Selection Screen.
    final result = await Navigator.push(
      context,
      // Create the SelectionScreen in the next step.
      MaterialPageRoute(builder: (context) => const
SelectionScreen()),
    );
  }
}
```

Now, build a selection screen that contains two buttons. When a user taps a button, that app closes the selection screen and lets the home screen know which button was tapped.

This step defines the UI. The next step adds code to return data.

```
class SelectionScreen extends StatelessWidget {
  const SelectionScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Pick an option'),),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Padding(
              padding: const EdgeInsets.all(8),
              child: ElevatedButton(
                onPressed: () {
                  // Pop here with "Yep"...},
                child: const Text('Yep!'),),),
            Padding(
              padding: const EdgeInsets.all(8),
              child: ElevatedButton(
                onPressed: () {
                  // Pop here with "Nope"...},
                child: const Text('Nope.'),),)],),),);}}
```

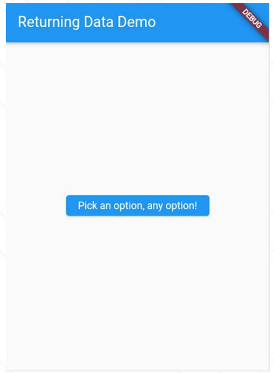
Now, update the onPressed() callback for both of the buttons. To return data to the first screen, use the Navigator.pop() method, which accepts an optional second argument called result. Any result is returned to the Future in the SelectionButton.

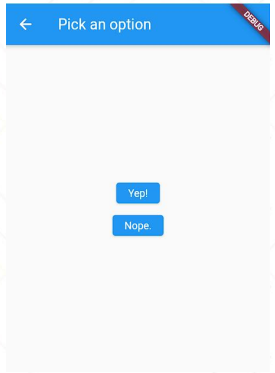
```
ElevatedButton(
  onPressed: () {
    // Close the screen and return "Yep!" as the result.
    Navigator.pop(context, 'Yep!');
  },
  child: const Text('Yep!'),
)
```

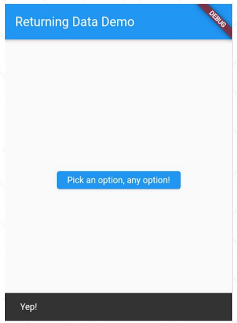
```
ElevatedButton(
  onPressed: () {
    // Close the screen and return "Nope." as the result.
    Navigator.pop(context, 'Nope.');
  },
  child: const Text('Nope.'),
)
```

Now that you're launching a selection screen and awaiting the result, you'll want to do something with the information that's returned.
In this case, show a snackbar displaying the result by using the _navigateAndDisplaySelection() method in SelectionButton:

```
// A method that launches the SelectionScreen and awaits the result from
// Navigator.pop.
Future<void> _navigateAndDisplaySelection(BuildContext context) async {
  // Navigator.push returns a Future that completes after calling
  // Navigator.pop on the Selection Screen.
  final result = await Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => const SelectionScreen()),
  );
  // When a BuildContext is used from a StatefulWidget, the mounted property
  // must be checked after an asynchronous gap.
  if (!mounted) return;
  // After the Selection Screen returns a result, hide any previous snackbars
  // and show the new result.
  ScaffoldMessenger.of(context)
    ..removeCurrentSnackBar()
    ..showSnackBar(SnackBar(content: Text('$result')));
}
```







Pass arguments to a named route

The Navigator provides the ability to navigate to a named route from any part of an app using a common identifier. In some cases, you might also need to pass arguments to a named route. For example, you might wish to navigate to the /user route and pass information about the user to that route.

Note: Named routes are no longer recommended for most applications. For more information, see Limitations in the navigation overview page.

You can accomplish this task using the arguments parameter of the Navigator.pushNamed() method. Extract the arguments using the ModalRoute.of() method or inside an onGenerateRoute() function provided to the MaterialApp or CupertinoApp constructor.

This recipe demonstrates how to pass arguments to a named route and read the arguments using ModalRoute.of() and onGenerateRoute() using the following steps:

- Define the arguments you need to pass.
- Create a widget that extracts the arguments.
- Register the widget in the routes table.
- Navigate to the widget.

First, define the arguments you need to pass to the new route. In this example, pass two pieces of data: The title of the screen and a message.

To pass both pieces of data, create a class that stores this information.

```
// You can pass any object to the arguments parameter.
// In this example, create a class that contains both
// a customizable title and message.
class ScreenArguments {
  final String title;
  final String message;

  ScreenArguments(this.title, this.message);
}
```

Next, create a widget that extracts and displays the title and message from the ScreenArguments. To access the ScreenArguments, use the ModalRoute.of() method. This method returns the current route with the arguments.

```
// A Widget that extracts the necessary arguments from
// the ModalRoute.
class ExtractArgumentsScreen extends StatelessWidget {
  const ExtractArgumentsScreen({super.key});
  static const routeName = '/extractArguments';
  @override
  Widget build(BuildContext context) {
    // Extract the arguments from the current ModalRoute
    // settings and cast them as ScreenArguments.
    final args = ModalRoute.of(context)!.settings.arguments as ScreenArguments;
    return Scaffold(
      appBar: AppBar(
        title: Text(args.title),
      ),
      body: Center(
        child: Text(args.message),
      ),
    );
  }
}
```

Next, add an entry to the routes provided to the MaterialApp widget. The routes define which widget should be created based on the name of the route.

```
MaterialApp(
  routes: {
    ExtractArgumentsScreen.routeName: (context) =>
        const ExtractArgumentsScreen(),
  },
)
```

Finally, navigate to the ExtractArgumentsScreen when a user taps a button using Navigator.pushNamed(). Provide the arguments to the route via the arguments property. The ExtractArgumentsScreen extracts the title and message from these arguments.

```
// A button that navigates to a named route.
// The named route extracts the arguments
// by itself.
ElevatedButton(
  onPressed: () {
    // When the user taps the button,
    // navigate to a named route and
    // provide the arguments as an optional
    // parameter.
    Navigator.pushNamed(
      context,
      ExtractArgumentsScreen.routeName,
      arguments: ScreenArguments(
        'Extract Arguments Screen',
        'This message is extracted in the build method.',
      ),
    );
  },
  child: const Text('Navigate to screen that extracts arguments'),
),
```

A design language, such as Material, defines standard behaviors when transitioning between routes (or screens). Sometimes, though, a custom transition between screens can make an app more unique. To help, PageRouteBuilder provides an Animation object. This Animation can be used with Tween and Curve objects to customize the transition animation. This recipe shows how to transition between routes by animating the new route into view from the bottom of the screen.

To create a custom page route transition, this recipe uses the following steps:

- Set up a PageRouteBuilder
- Create a Tween
- Add an AnimatedWidget
- Use a CurveTween
- Combine the two Tweens

To start, use a PageRouteBuilder to create a Route. PageRouteBuilder has two callbacks, one to build the content of the route (pageBuilder), and one to build the route's transition (transitionsBuilder).

Note: The child parameter in transitionsBuilder is the widget returned from pageBuilder. The pageBuilder function is only called the first time the route is built. The framework can avoid extra work because child stays the same throughout the transition.

The following example creates two routes: a home route with a "Go!" button, and a second route titled "Page 2".

```
void main() {
  runApp(
    const MaterialApp(
      home: Page1(),),);}
class Page1 extends StatelessWidget {
  const Page1({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.of(context).push(_createRoute());},
          child: const Text('Go!')
,),),);}}
```

```
Route _createRoute() {
  return PageRouteBuilder(
    pageBuilder: (context, animation, secondaryAnimation) => const
Page2(),
    transitionsBuilder: (context, animation, secondaryAnimation, child) {
      return child;
    },
  );
}

class Page2 extends StatelessWidget {
  const Page2({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: const Center(
        child: Text('Page 2'),
      ),
    );
  }
}
```

To make the new page animate in from the bottom, it should animate from Offset(0,1) to Offset(0, 0) (usually defined using the Offset.zero constructor). In this case, the Offset is a 2D vector for the ‘FractionalTranslation’ widget. Setting the dy argument to 1 represents a vertical translation one full height of the page.

The transitionsBuilder callback has an animation parameter. It’s an Animation<double> that produces values between 0 and 1. Convert the Animation into an Animation using a Tween:

```
transitionsBuilder: (context, animation, secondaryAnimation, child) {
  const begin = Offset(0.0, 1.0);
  const end = Offset.zero;
  final tween = Tween(begin: begin, end: end);
  final offsetAnimation = animation.drive(tween);
  return child;
},
```

Flutter has a set of widgets extending AnimatedWidget that rebuild themselves when the value of the animation changes. For instance, SlideTransition takes an Animation<Offset> and translates its child (using a FractionalTranslation widget) whenever the value of the animation changes.

AnimatedWidget Return a SlideTransition with the Animation<Offset> and the child widget:

```
transitionsBuilder: (context, animation, secondaryAnimation, child) {
  const begin = Offset(0.0, 1.0);
  const end = Offset.zero;
  final tween = Tween(begin: begin, end: end);
  final offsetAnimation = animation.drive(tween);

  return SlideTransition(
    position: offsetAnimation,
    child: child,
  );
},
```

Flutter provides a selection of easing curves that adjust the rate of the animation over time. The Curves class provides a predefined set of commonly used curves. For example, Curves.easeOut makes the animation start quickly and end slowly.

To use a Curve, create a new CurveTween and pass it a Curve:

```
var curve = Curves.ease;
var curveTween = CurveTween(curve: curve);
```

To combine the tweens, use chain():

```
const begin = Offset(0.0, 1.0);
const end = Offset.zero;
const curve = Curves.ease;

var tween = Tween(begin: begin, end: end).chain(CurveTween(curve: curve));
```

Then use this tween by passing it to animation.drive(). This creates a new Animation<Offset> that can be given to the SlideTransition widget:

```
return SlideTransition(
  position: animation.drive(tween),
  child: child,
);
```

This new Tween (or Animatable) produces Offset values by first evaluating the CurveTween, then evaluating the Tween<Offset>. When the animation runs, the values are computed in this order:

The animation (provided to the transitionsBuilder callback) produces values from 0 to 1.
The CurveTween maps those values to new values between 0 and 1 based on its curve.
The Tween<Offset> maps the double values to Offset values.
Another way to create an Animation<Offset> with an easing curve is to use a CurvedAnimation:

```
transitionsBuilder: (context, animation, secondaryAnimation, child) {
  const begin = Offset(0.0, 1.0);
  const end = Offset.zero;
  const curve = Curves.ease;
  final tween = Tween(begin: begin, end: end);
  final curvedAnimation = CurvedAnimation(
    parent: animation,
    curve: curve,
  );
  return SlideTransition(
    position: tween.animate(curvedAnimation),
    child: child,
  );
}
```