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

Flutter Text

A Text is a widget in Flutter that allows us to **display a string of text with a single line in our application**. Depending on the layout constraints, we can break the string across multiple lines or might all be displayed on the same line. If we do not specify any styling to the text widget, it will use the closest **DefaultTextStyle** class style. This class does not have any explicit style. In this article, we are going to learn how to use a Text widget and how to style it in our application.

Here is a simple example to understand this widget. This example shows our **project's title** in the application bar and a **message** in the application's body.

```
child:Text("Welcome to World ")
```

Text Widget Constructor:

The text widget constructor used to make the custom look and feel of our text in Flutter

```
const Text(String data,{
        Key key,
        TextStyle style,
        StrutStyle strutStyle,
        TextAlign textAlign,
        TextDirection textDirection,
        TextOverflow overflow,
        bool softWrap,
        double textScaleFactor,
        int maxLines,
        String semanticsLabel,
        TextWidthBasis textWidthBasis,
        TextHeightBehavior textHeightBehavior
        }
    )
```

The following are the essential properties of the Text widget used in our application:

TextAlign: It is used to specify how our text is aligned horizontally. It also controls the text location.

TextDirection: It is used to determine how textAlign values control the layout of our text. Usually, we write text from left to right, but we can change it using this parameter.

Overflow: It is used to determine when the text will not fit in the available space. It means we have specified more text than the available space.

TextScaleFactor: It is used to determine the scaling to the text displayed by the Text widget. Suppose we have specified the text scale factor as 1.5, then our text will be 50 percent larger than the specified font size.

SoftWrap: It is used to determine whether or not to show all text widget content when there is not enough space available. If it is true, it will show all content. Otherwise, it will not show all content.

MaxLines: It is used to determine the maximum number of lines displayed in the text widget.

TextWidthBasis: It is used to control how the text width is defined.

TextHeightBehavior: It is used to control how the paragraph appears between the first line and descent of the last line.

Style: It is the most common property of this widget that allows developers to styling their text. It can do styling by specifying the foreground and background color, font size, font weight, letter and word spacing, locale, shadows, etc. See the table to understand it more easily:

Attributes	Descriptions
foreground	It determines the paint as a foreground for the text.
background	It determines the paint as a background for the text.
fontWeight	It determines the thickness of the text.

fontSize	It determines the size of the text.
fontFamily	It is used to specify the typeface for the font. For this, we need to download a typeface file in our project, and then keep this file into the assets/font folder. Finally, config the pubspec.yaml file to use it in the project.
fontStyle	It is used to style the font either in bold or italic form.
Color	It is used to determine the color of the text.
letterSpacing	It is used to determine the distance between the characters of the text.
wordSpacing	It is used to specify the distance between two words of the text.
shadows	It is used to paint underneath the text.
decoration	We use this to decorate text using the three parameters: decoration, decorationColor, decorationStyle. The decoration determines the location, decorationColor specify the color, decorationStyle determine the shape.

```
 Text('hello World',
  style: TextStyle(
    fontSize: 35,
    color: Colors.purple,
    fontWeight: FontWeight.w700,
    fontStyle: FontStyle.italic,
    letterSpacing: 8,
    wordSpacing: 20,
    backgroundColor: Colors.yellow,
      shadows: [
        Shadow(color: Colors.blueAccent, offset:
Offset(2,1), blurRadius:10)
      ],
  ),
),
```

Output:

When we run this application in the emulator or device, we should get the UI similar to the below screenshot:

Flutter TextField

A TextField or TextBox is an **input element** which holds the alphanumeric data, such as name, password, address, etc. It is a GUI control element that enables the user to **enter text information** using a programmable code. It can be of a single-line text field (when only one line of information is required) or multiple-line text field (when more than one line of information is required).

TextField in Flutter is the most commonly used **text input widget** that allows users to collect inputs from the keyboard into an app. We can use the **TextField** widget in building forms, sending messages, creating search experiences, and many more. By default, Flutter decorated the TextField with an underline. We can also add several attributes with TextField, such as label, icon, inline hint text, and error text using an InputDecoration as the decoration. If we want to remove the decoration properties entirely, it is required to set the decoration to **null**.

The following code explains a **demo example of TextFiled widget** in **Flutter:**

```
TextField (
  decoration: InputDecoration(
      border: InputBorder.none,
      labelText: 'Enter  Name',
      hintText: 'Enter Your Name'
```

```
      ),
    ),
```

Some of the most common attributes used with the TextField widget are as follows:

- **decoration:** It is used to show the decoration around TextField.
- **border:** It is used to create a default rounded rectangle border around TextField.
- **labelText:** It is used to show the label text on the selection of TextField.
- **hintText:** It is used to show the hint text inside TextField.
- **icon:** It is used to add icons directly to the TextField.

We are going to see how to use TextField widget in the Flutter app through the following steps:

```
TextField(
 obscureText: true,
 decoration: InputDecoration(
   border: OutlineInputBorder(),
   labelText: 'Password',
   hintText: 'Enter Password',
 ),),
```

Let us see the complete source code that contains the TextField Widget. This Flutter application takes **two TextFields and one RaisedButton**. After filling the details, the user clicks on the button. Since we have not specified any value in the **onPressed ()** property of the button, it cannot print them to console.

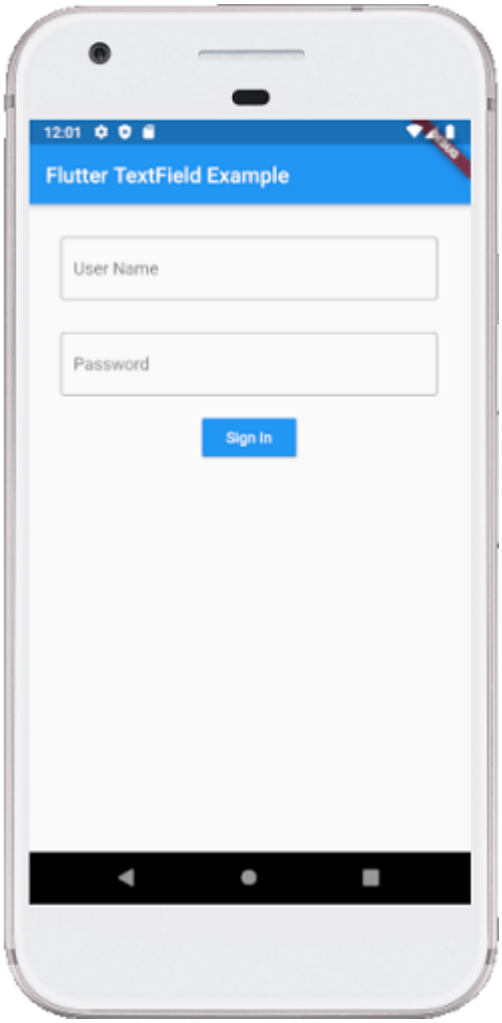
Replace the following code in the **main.dart** file and see the output.

```
   body: Padding(
    padding: EdgeInsets.all(15),
    child: 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: (){},
        ),
  ],
    ),
),
```

Output

When we run the application in android emulator, we should get UI similar to the following screenshot:



If we click inside the text box, we can see that a keyboard has appeared from the bottom of the screen, the label goes into the top left corner of the border, and the hint text shown into the field. The below screenshot explains it more clearly:

How to retrieve the value of a TextField?

We know that Flutter does not have an ID like in Android for the TextField widget. Flutter allows the user to **retrieve the text in mainly two ways**: First is the onChanged method, and another is the controller method. Both are discussed below:

1. onChanged method: It is the easiest way to retrieve the text field value. This method store the current value in a simple variable and then use it in the TextField widget. Below is the sample example:

```
String value = "";
TextField(
  onChanged: (text) {
    value = text;
  },
)
```

2. Controller method: It is a popular method to retrieve text field value using **TextEditingController**. It will be attached to the TextField widget and then listen to change and control the widget's text value.

How to make TextField expandable?

Sometimes, we want to expand a TextField that means it can have more than one line. Flutter can do this very easily by adding the attributes **maxLines** and set it to **null**, which is one by default. We can also specify the exact value to expand the number of lines by default.

```
1. TextField(
2.   maxLines: 4,
3. )
```

How to control the size of TextField value?

TextField widget also allows us to restrict the maximum number of characters inside the text field. We can do this by adding the **maxLength** attributes as below:

```
1. TextField(
2.   maxLength: 10,
3. ),
```

How to obscure text field value?

Obscure means to make the field not readable or not understandable easily. The obscure text cannot visible clear. In Flutter, it is mainly used with a text field that contains a password. We can make the value in a TextField obscure by setting the **obscureText** to **true**.

```
1. TextField(
```

2. obscureText: true,
3.),

Flutter Buttons

Buttons are the graphical control element that **provides a user to trigger an event** such as taking actions, making choices, searching things, and many more. They can be placed anywhere in our UI like dialogs, forms, cards, toolbars, etc.

Buttons are the Flutter widgets, which is a part of the material design library. Flutter provides several types of buttons that have different shapes, styles, and features.

Features of Buttons

The standard features of a button in Flutter are given below:

1. We can easily apply themes on buttons, shapes, color, animation, and behavior.
2. We can also theme icons and text inside the button.
3. Buttons can be composed of different child widgets for different characteristics.

Types of Flutter Buttons

Following are the different types of button available in Flutter:

- Flat Button
- Raised Button
- Floating Button
- Drop Down Button
- Icon Button
- Inkwell Button
- PopupMenu Button
- Outline Button

Let us discuss each button in detail.

1. Flat Button

It is a **text label button** that does not have much decoration and displayed **without any elevation**. The flat button has two required properties that are: **child and onPressed()**. It is mostly used in toolbars, dialogs, or inline with other content. By default, the flat button has no color, and its text is black. But, we can use color to the button and text using **color and textColor** attributes, respectively.

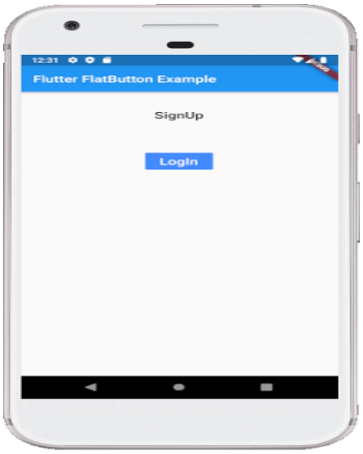
Example:

Open the **main.dart** file and replace it with the below code.

```
Container(
 margin: EdgeInsets.all(25),
  child: FlatButton(
   child: Text('SignUp', style: TextStyle(fontSize: 20.0),),
    onPressed: () {},
 ),
),
Container(
   margin: EdgeInsets.all(25),
  child: FlatButton(
     child: Text('LogIn', style: TextStyle(fontSize: 20.0),),
    color: Colors.blueAccent,
     textColor: Colors.white,
    onPressed: () {},
   ),
),
```

Output:

If we run this app, we will see the following screen:



2. Raised Button

It is a button, which is based on the material widget and has a **rectangular body**. It is similar to a flat button, but it **has an elevation** that will increases when the button is pressed. It adds dimension to the UI along Z-axis. It has several properties like text color, shape, padding, button color, the color of a button when disabled, animation time, elevation, etc.

This button has **two callback functions**.

onPressed(): It is triggered when the button is pressed.

onLongPress(): It is triggered when the button is long pressed.

It is to note that this button is in a **disabled state** if onPressed() and onLongPressed() callbacks are not specified.

A FAB button is a **circular icon button** that triggers the primary action in our application. It is the most used button in today's applications. We can use this button for adding, refreshing, or sharing the content. Flutter suggests using at most one FAB button per screen. There are two types of Floating Action Button:

FloatingActionButton: It creates a simple circular floating button with a child widget inside it. It must have a child parameter to display a widget.

FloatingActionButton.extended: It creates a wide floating button along with an icon and a label inside it. Instead of a child, it uses labels and icon parameters.

4. DropDown Button

A drop-down button is used to create a nice overlay on the screen that allows the user to select any item from multiple options. Flutter allows a simple way to implement a drop-down box or drop-down button. This button shows the currently selected item and an arrow that opens a menu to select an item from multiple options.

Flutter provides a **DropdownButton widget** to implement a drop-down list. We can place it anywhere in our app.

Example

Open the **main.dart** file and replace it with the below code.

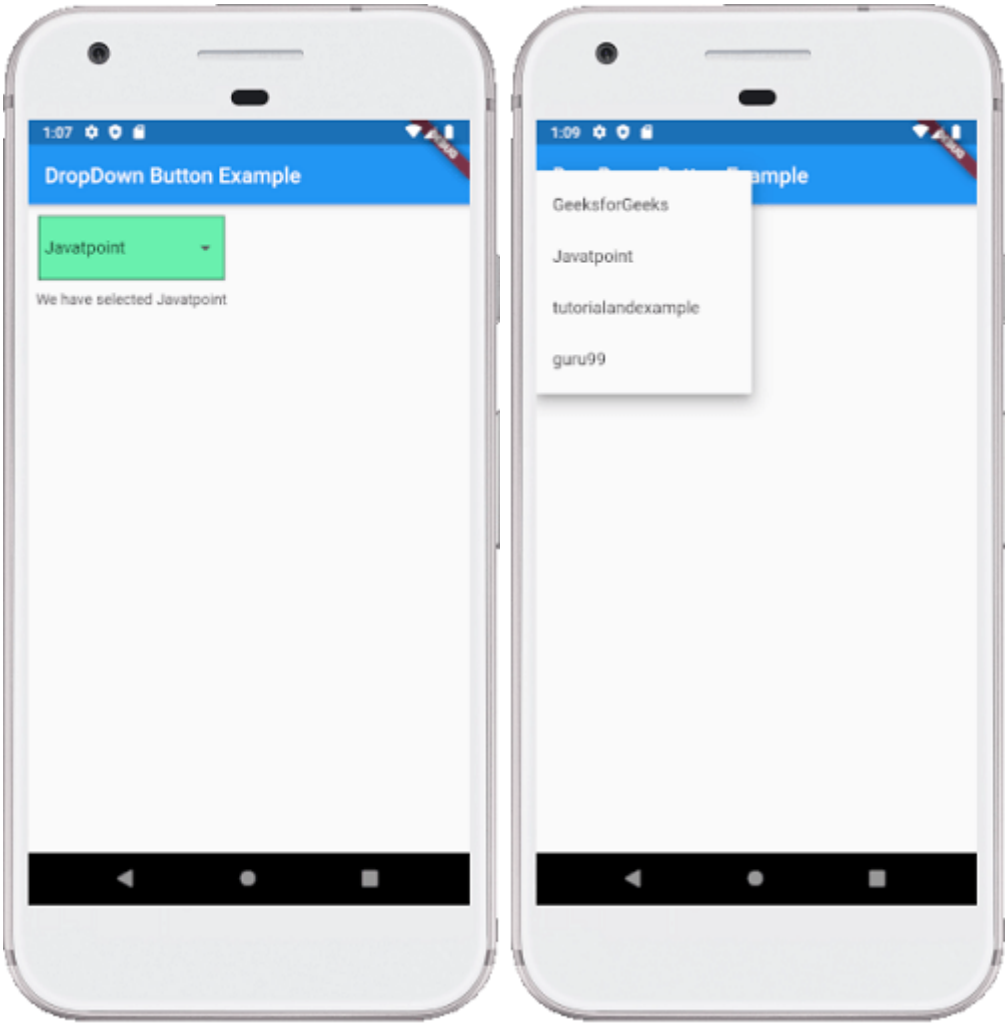
```
import 'package:flutter/material.dart';

void main() => runApp(MaterialApp(
  home: MyApp(),
));

class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  List<ListItem> _dropdownItems = [
    ListItem(1, "GeeksforGeeks"),
    ListItem(2, "Javatpoint"),
    ListItem(3, "tutorialandexample"),
    ListItem(4, "guru99")
  ];

  List<DropdownMenuItem<ListItem>> _dropdownMenuItems;
  ListItem _itemSelected;

  void initState() {
```

```
  super.initState();
  _dropdownMenuItems = buildDropDownMenuItems(_dropdownItems);
  _itemSelected = _dropdownMenuItems[1].value;
}

List<DropdownMenuItem<ListItem>> buildDropDownMenuItems(List listItems) {
  List<DropdownMenuItem<ListItem>> items = List();
  for (ListItem listItem in listItems) {
    items.add(
      DropdownMenuItem(
        child: Text(listItem.name),
        value: listItem,
      ),
    );
  }
  return items;
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("DropDown Button Example"),
    ),
    body: Column(
      children: <Widget>[
        Padding(
          padding: const EdgeInsets.all(10.0),
          child: Container(
            padding: const EdgeInsets.all(5.0),
            decoration: BoxDecoration(
                color: Colors.greenAccent,
              border: Border.all()),
            child: DropdownButtonHideUnderline(
```

```
            child: DropdownButton(
                value: _itemSelected,
               items: _dropdownMenuItems,
                onChanged: (value) {
                setState(() {
                    _itemSelected = value;
                });
                }),
          ),
          ),
        ),
         Text("We have selected ${_itemSelected.name}"),
      ],
     ),
   );
  }
}

class ListItem {
  int value;
  String name;

  ListItem(this.value, this.name);
}
```

Output

Run the application in android emulator, and it will give the UI similar to the following screenshot. The second image is an output of the list contains in the drop drown button.



5. Icon Button

An IconButton is a **picture printed** on the Material widget. It is a useful widget that gives the Flutter UI a material design feel. We can also customize the look and feel of this button. In simple terms, it is an icon that reacts when the user will touch it.

Example:

Open the **main.dart** file and replace it with the below code.

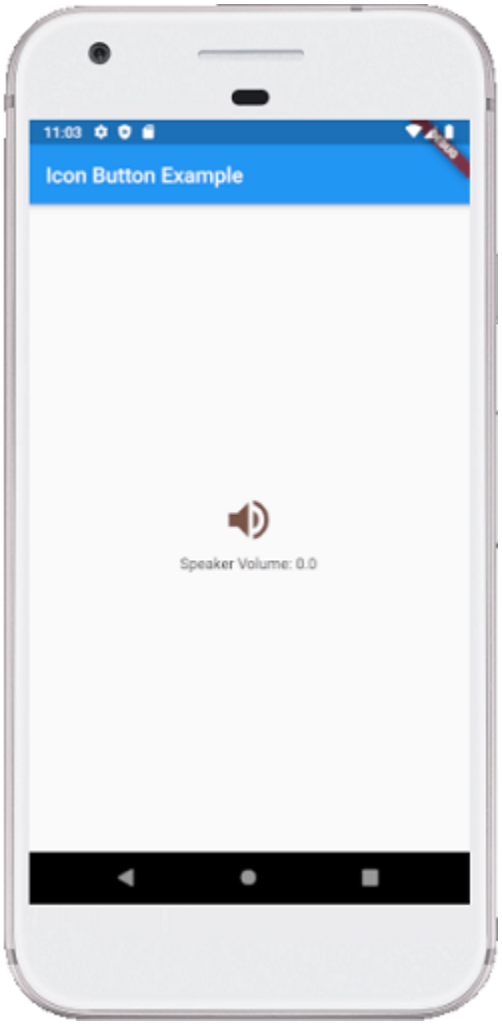
```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("Icon Button Example"),
        ),
        body: Center(
          child: MyStatefulWidget(),
        ),
      ),
    );
  }
}
double _speakervolume = 0.0;

class MyStatefulWidget extends StatefulWidget {
  MyStatefulWidget({Key key}) : super(key: key);

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

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  Widget build(BuildContext context) {
    return Column(
      mainAxisSize: MainAxisSize.min,
      children: <Widget>[
        IconButton(
          icon: Icon(Icons.volume_up),
          iconSize: 50,
          color: Colors.brown,
          tooltip: 'Increase volume by 5',
          onPressed: () {
```

```
            setState(() {
              _speakervolume += 5;
            });
          },
        ),
        Text('Speaker Volume: $_speakervolume')
      ],
    );
  }
}
```

Output:

Run the application in android emulator, and it will give the UI similar to the following screenshot. When we press the **volume button**, it will always increase by 5.



6. Inkwell Button

InkWell button is a material design concept, which is used for **touch response**. This widget comes under the Material widget where the ink reactions are actually painted. It creates the app UI interactive by adding gesture feedback. It is mainly used for adding **splash ripple effect**.

Example:

Open the **main.dart** file and replace it with the below code.

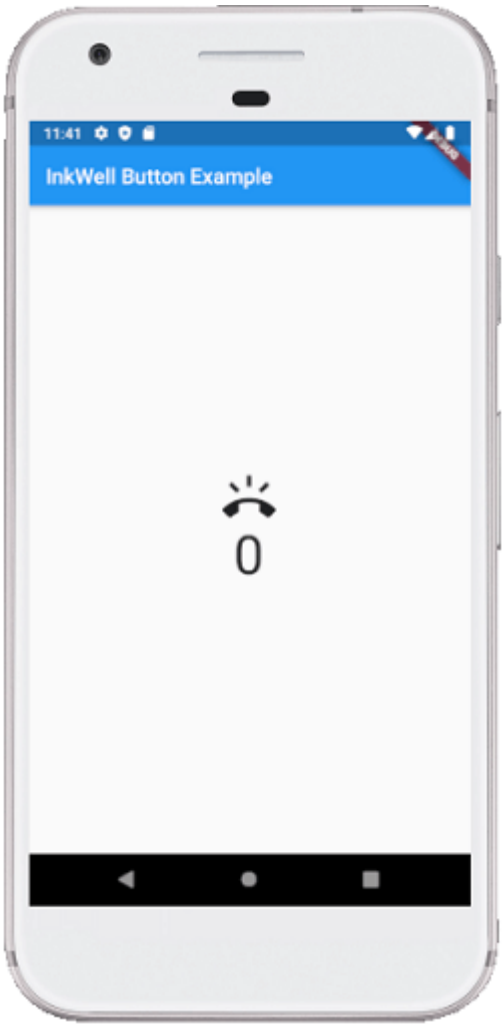
```
import 'package:flutter/material.dart';

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

```
class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  int _volume = 0;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('InkWell Button Example'),
        ),
        body: Center(
          child: new Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              InkWell(
                splashColor: Colors.green,
                highlightColor: Colors.blue,
                child: Icon(Icons.ring_volume, size: 50),
                onTap: () {
                  setState(() {
                  _volume += 2;
                  });
                },
              ),
              Text (
                  _volume.toString(),
                style: TextStyle(fontSize: 50)
              ),
            ],
```

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

Output:

Run the application in android emulator, and it will give the UI similar to the following screenshot. Every time we press the ring volume button, it will increase the volume by 2.



7. PopupMenu Button

It is a button that **displays the menu** when it is pressed and then calls the **onSelected** method the menu is dismissed. It is because the item from the multiple options is selected. This button contains a text and an image. It will mainly use with **Settings** menu to list all options. It helps in making a great user experience.

Example:

Open the **main.dart** file and replace it with the below code.

```
import 'package:flutter/material.dart';
 void main() { runApp(MyApp());}
class MyApp extends StatefulWidget {
 @override
 _MyAppState createState() => _MyAppState();
}
class _MyAppState extends State<MyApp> {
 Choice _selectedOption = choices[0];
 void _select(Choice choice) {
  setState(() {
   _selectedOption = choice;
  });
 }
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   home: Scaffold(
    appBar: AppBar(
     title: const Text('PopupMenu Button Example'),
     actions: <Widget>[
      PopupMenuButton<Choice>(
       onSelected: _select,
        itemBuilder: (BuildContext context) {
         return choices.skip(0).map((Choice choice) {
          return PopupMenuItem<Choice>(
```

```
                value: choice,
                  child: Text(choice.name),
                );
                }).toList();
            },
            ),
          ],
        ),
        body: Padding(
          padding: const EdgeInsets.all(10.0),
          child: ChoiceCard(choice: _selectedOption),
        ),
      ),
    );
  }
}

class Choice {
  const Choice({this.name, this.icon});
  final String name;
  final IconData icon;
}

const List<Choice> choices = const <Choice>[
  const Choice(name: 'Wi-Fi', icon: Icons.wifi),
  const Choice(name: 'Bluetooth', icon: Icons.bluetooth),
  const Choice(name: 'Battery', icon: Icons.battery_alert),
  const Choice(name: 'Storage', icon: Icons.storage),
];

class ChoiceCard extends StatelessWidget {
  const ChoiceCard({Key key, this.choice}) : super(key: key);

  final Choice choice;
```

```
@override
Widget build(BuildContext context) {
  final TextStyle textStyle = Theme.of(context).textTheme.headline;
  return Card(
    color: Colors.greenAccent,
    child: Center(
      child: Column(
        mainAxisSize: MainAxisSize.min,
        crossAxisAlignment: CrossAxisAlignment.center,
        children: <Widget>[
          Icon(choice.icon, size: 115.0, color: textStyle.color),
          Text(choice.name, style: textStyle),
        ],
      ),
    ),
  );
 }
}
```

Output:

Run the application in android emulator, and it will give the UI similar to the following screenshot. When we click the **three dots** shown at the top left corner of the screen, it will pop up the multiple options. Here, we can select any option, and it will keep it in the card, as shown in the second image.

8. Outline Button

It is similar to the flat button, but it contains a thin grey rounded rectangle border. Its outline border is defined by the shape attribute.

Example:

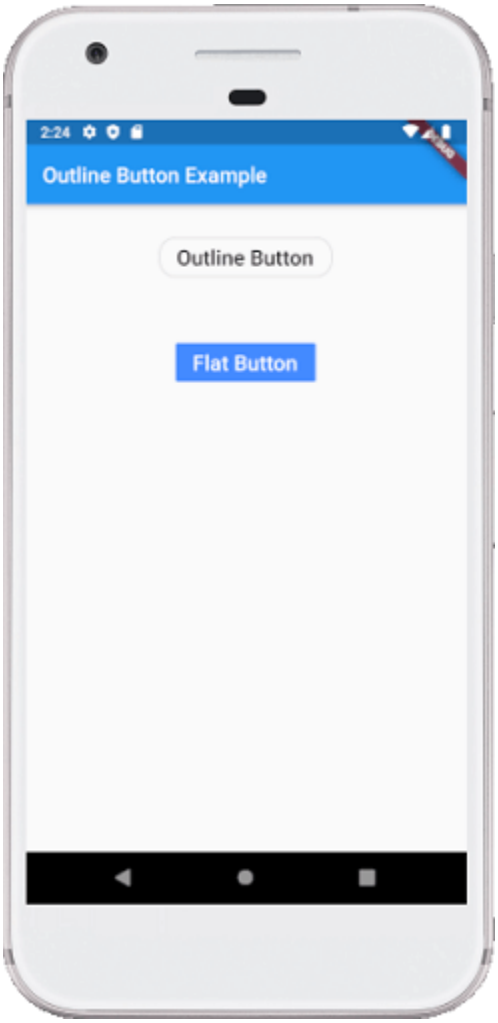
Open the **main.dart** file and replace it with the below code.

```
Container(
 margin: EdgeInsets.all(25),
  child: OutlineButton(
   child: Text("Outline Button", style: TextStyle(fontSize: 20.0),),
    highlightedBorderColor: Colors.red,
   shape: RoundedRectangleBorder(
```

```
            borderRadius: BorderRadius.circular(15)),
        onPressed: () {},
      ),
    ),
    Container(
     margin: EdgeInsets.all(25),
      child: FlatButton(
       child: Text('Flat Button', style: TextStyle(fontSize: 20.0),),
        color: Colors.blueAccent,
       textColor: Colors.white,
        onPressed: () {},
     ),
    ),
```

Output:

Run the application in android emulator, and it will give the UI similar to the following screenshot.



Button Bar

Flutter provides the flexibility to **arrange the buttons in a bar or a row**. ButtonBar widget contains three properties: **alignment, children, and mainAxisSize**.

- Alignment is used to present the aligning option to the entire button bar widget.
- Children attribute is used to take the number of buttons in a bar.
- mainAxisSize attribute is used to provide the horizontal space for the button bar.

Example:

Open the **main.dart** file and replace it with the below code.

```
body: Padding(
  padding: EdgeInsets.all(10),
  child: Column(
```

```
        children: <Widget>[
    Padding(
    padding: EdgeInsets.all(15),
    child: new ButtonBar(
      mainAxisSize: MainAxisSize.min,
      children: <Widget>[
        RaisedButton(
          child: new Text('DART'),
          color: Colors.lightGreen,
          onPressed: () {/** */},
        ),
        FlatButton(
          child: Text('Flutter'),
          color: Colors.lightGreen,
          onPressed: () {/** */},
        ),
        FlatButton(
          child: Text('MySQL'),
          color: Colors.lightGreen,
          onPressed: () {/** */},
        ),
      ],
    ),
  ),
],
    ),
  ),
```

Output:

Run the application in android emulator, and it will give the UI similar to the following screenshot. Here, we can see that the three buttons are placed in a horizontal bar or row.

Flutter Slider

A slider in Flutter is a material design widget used for selecting a range of values. It is an input widget where we can **set a range of values by dragging or pressing on the desired position**. In this article, we are going to show how to use the slider widget in Flutter in setting the range of values and how to customize the look of a slider.

Usually, we use the slider widget for changing a value. So, it is required to store the value in a variable. This widget has a slider class that requires the **onChanged()** function. This function will be called whenever we change the slider position.

A slider can be used for selecting a value from a **continuous or discrete** set of values. By default, it uses a continuous range of values. If we want to use discrete values, we must have to use a non-null value for divisions. This discrete division indicates the number of discrete intervals. Before getting the value, we have to set

the **minimum** and **maximum** value. Slider provides min and max arguments to set the minimum and maximum limitations. **For example**, we have a set of values from 0.0 to 50.0, and divisions are 10, the slider would take values like 0.0, 10.0, 20.0, 30.0, 40.0, and 50.0.

Slider Properties

The following are the slider attributes used in Flutter. It has two required arguments, and all others are optional.

Attributes	Type	Descriptions
Value	double	It is a required argument and used to specify the slider's current value.
onChanged	double	It is a required argument and called during dragging when the user selects a new value for the slider. If it is null, the slider is disabled.
onChangedStart	double	It is an optional argument and called when we start selecting a new value.
onChangedStart	double	It is an optional argument and called when we have done in selecting a new value for the slider.
Max	double	It is an optional argument and determines the maximum value that can be used by the user. By default, it is 1.0. It should be greater than or equal to min.
Min	double	It is an optional argument that determines the minimum value that can be used by the user. By default, it is 0.0. It should be less than or equal to max.

Divisions	int	It determines the number of discrete divisions. If it is null, the slider is continuous.
Label	string	It specifies the text label that will be shown above the slider. It displays the value of a discrete slider.
activeColor	class Color	It determines the color of the active portion of the slider track.
inactiveColor	class Color	It determines the color of the inactive portion of the slider track.
SemanticFormatterCallback		It is a callback that is used to create a semantic value. By default, it is a percentage.

The slider in Flutter uses the following terms:

Thumb: It is a round shape that slides horizontally when we change values by dragging.

Track: It is a horizontal line where we can slide the thumb.

Overlay: It appears around the thumb while dragging.

Tick marks: It is used to mark the discrete values of a slider.

Value indicators: It will show the labels for the thumb values when we defined the labels.

Active: It is the active side of the slider, which is in between the thumb and the minimum value.

Inactive: It is the inactive side of the slider, which is in between the thumb and the maximum value.

How to use the slider widget in Flutter?

Here is the basic example of using a slider widget in Flutter.

```
Slider(
    min: 0,
```

```
 max: 100,
value: _value,
 onChanged: (value) {
  setState(() {
    _value = value;
  });
 },
),
```

Example

Let us understand how to use the slider in Flutter with the help of an example. In the following code, we had stored the value as an integer that must be cast to double first when it passed as a value argument and then rounded to integer inside the onChanged method. We have also specified the **active portion** of the slider as **green**, while the **inactive portion** is **orange**.

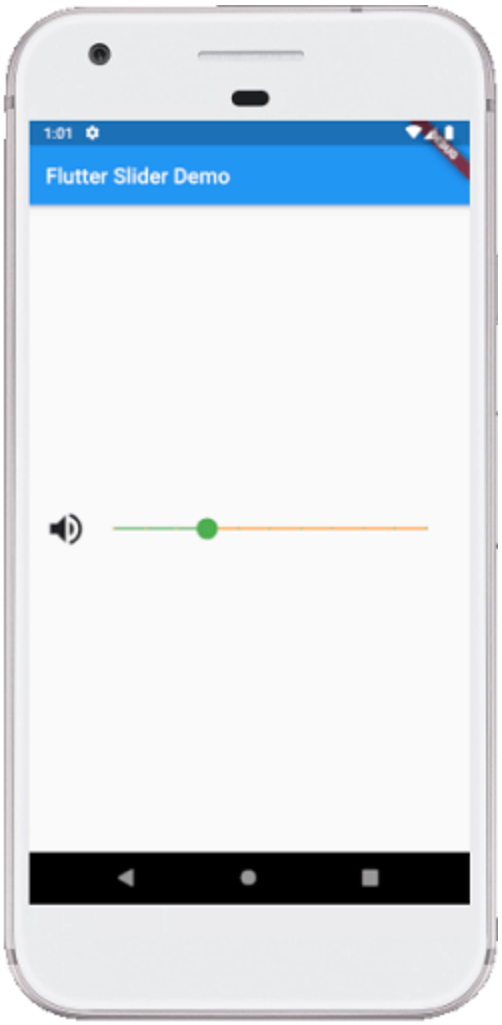
```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  int _value = 6;
  void _incrementCounter() {
    setState(() {
      // This call to setState tells the Flutter framework
that something has
      // changed in this State, which causes it to rerun
the build method below
      // so that the display can reflect the updated
values. If we changed
      // _counter without calling setState(), then the
build method would not be
      // called again, and so nothing would appear to
happen.
      _counter++;
    });
  }
  @override
  Widget build(BuildContext context) {
    // This method is rerun every time setState is called,
for instance as done
    // by the _incrementCounter method above.
    //
```

```
    // The Flutter framework has been optimized to make
rerunning build methods
    // fast, so that you can just rebuild anything that
needs updating rather
    // than having to individually change instances of
widgets.
    return Scaffold(
      appBar: AppBar(
        // Here we take the value from the MyHomePage
object that was created by
        // the App.build method, and use it to set our
appbar title.
        title: Text(widget.title),
        actions: <Widget>[
          IconButton(icon: Icon(Icons.camera_alt),
onPressed: () => {}),
          IconButton(icon: Icon(Icons.account_circle),
onPressed: () => {})
        ],

      ),


    body: Padding(
    padding: EdgeInsets.all(15.0),
    child: Center(
    child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    mainAxisSize: MainAxisSize.max,
    children: [
    Icon(
    Icons.volume_up,
    size: 40,
    ),
    new Expanded(
    child: Slider(
    value: _value.toDouble(),
    min: 1.0,
    max: 20.0,
    divisions: 10,
    activeColor: Colors.green,
    inactiveColor: Colors.orange,
    label: 'Set volume value',
    onChanged: (double newValue) {
```

```
    setState(() {
    _value = newValue.round();
    });
    },
    semanticFormatterCallback: (double newValue) {
    return '${newValue.round()} dollars';
    }
    )
    ),
    ]
    )
    ),
    ),



    floatingActionButton: FloatingActionButton(
        child: Icon(Icons.navigation),
        backgroundColor: Colors.green,
        foregroundColor: Colors.white,
        onPressed: () => {},
      ),

      // This trailing comma makes auto-formatting nicer
for build methods.
    );


    }
  }
```

Output:

When we run this app in the IDE, we should get the UI similar to the below screenshot. Here we can **drag the slider** to set the volume label.



Flutter Range Slider

It is a highly customizable component that selects a value from a range of values. It can be selected either from a continuous or discrete set of values.

Why Range Slider?

A slider component can provide single or multiple selections based on the continuous or discrete set of values. Here we must have to predetermine either a minimum or maximum value to adjust the selection in one direction. Unlike the slider, the range sliders allow two selection points that provide flexible adjustment to set the maximum and minimum value. This adjustment makes it a useful feature when we want to control a specific range, like indicating the length of time or price points.

See the below code where range values are in the intervals of 10 because we have divided the slider into ten divisions from 0 to 100. It means our values are split between 0, 10, 20, 30, 40, and so on to 100. Here, we will initialize the range values with 20 and 50.

```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  int _value = 6;
  void _incrementCounter() {
    setState(() {
      // This call to setState tells the Flutter framework
that something has
      // changed in this State, which causes it to rerun
the build method below
      // so that the display can reflect the updated
values. If we changed
      // _counter without calling setState(), then the
build method would not be
      // called again, and so nothing would appear to
happen.
      _counter++;
    });
  }
  @override
  Widget build(BuildContext context) {
    // This method is rerun every time setState is called,
for instance as done
    // by the _incrementCounter method above.
    //
    // The Flutter framework has been optimized to make
rerunning build methods
    // fast, so that you can just rebuild anything that
needs updating rather
    // than having to individually change instances of
widgets.
    return Scaffold(
      appBar: AppBar(
        // Here we take the value from the MyHomePage
object that was created by
        // the App.build method, and use it to set our
appbar title.
        title: Text(widget.title),
        actions: <Widget>[
```

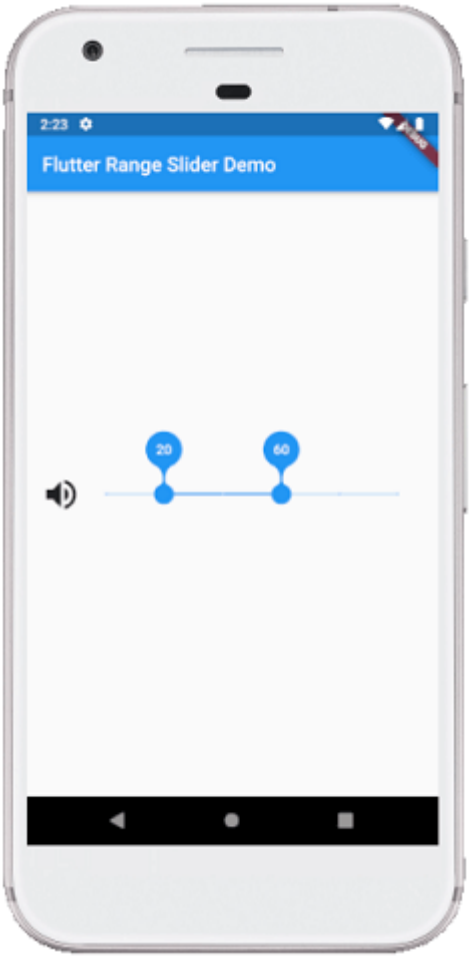
```
          IconButton(icon: Icon(Icons.camera_alt),
onPressed: () => {}),
          IconButton(icon: Icon(Icons.account_circle),
onPressed: () => {})
        ],

      ),


    body: Padding(
    padding: EdgeInsets.all(15.0),
    child: Center(
    child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    mainAxisSize: MainAxisSize.max,
    children: [
    Icon(
    Icons.volume_up,
    size: 40,
    ),
    new Expanded(
    child: Slider(
    value: _value.toDouble(),
    min: 1.0,
    max: 20.0,
    divisions: 10,
    activeColor: Colors.green,
    inactiveColor: Colors.orange,
    label: 'Set volume value',
    onChanged: (double newValue) {
    setState(() {
    _value = newValue.round();
    });
    },
    semanticFormatterCallback: (double newValue) {
    return '${newValue.round()} dollars';
    }
    )
    ),
    ]
    )
    ),
    ),
```

```
    floatingActionButton: FloatingActionButton(
        child: Icon(Icons.navigation),
        backgroundColor: Colors.green,
        foregroundColor: Colors.white,
        onPressed: () => {},
      ),

      // This trailing comma makes auto-formatting nicer
for build methods.
    );


    }
  }
```

Output:

When we run this app in the IDE, we should get the UI similar to the below screenshot.

When we drag the slider, we can see the range of values to set the volume label.



Flutter Image Slider

Image slider is a convenient way to display images, videos, or graphics in our app. Generally, it will show one big image at a time on our app screen. It helps to make our screen more attractive to the user.

Let us see how we can make an image slider in Flutter. There are many dependencies available in the Flutter library to create a sliding image in your app. Here, we are going to use **flutter_swipper dependency**. So first, we need to add the below dependency in the **puspec.yaml** file:

```
1. dependencies:
2.   flutter:
3.     sdk: flutter
4.   flutter_swiper: ^1.1.6
```

Next, add this dependency in your dart file as below:

1. `import 'package:flutter_swiper/flutter_swiper.dart';`

Example

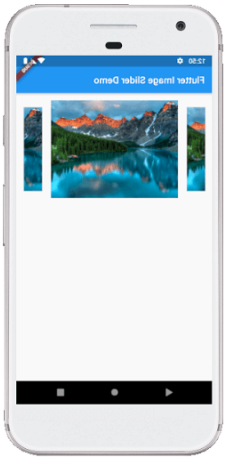
The following code explains the use of the flutter_swipper library in a simple way. Here, we have taken an image from the **network** that displays on the screen.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
       home: MyHomeScreen()
    );
  }
}

class MyHomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Flutter Image Slider Demo"),),
      body: Container(
          padding: EdgeInsets.all(10),
          margin: EdgeInsets.all(5),
          alignment: Alignment.center,
          constraints: BoxConstraints.expand(
              height: 225
          ),
          child: imageSlider(context)),
```

```
    );
  }
}

Swiper imageSlider(context){
  return new Swiper(
    autoplay: true,
    itemBuilder: (BuildContext context, int index) {
      return new Image.network(
        "https://lh3.googleusercontent.com/wlcl3tehFmOUpq-Jl3hlVbZVFrLHePRtIDWV5lZwBVDr7kEAgLTChyvXUclMVQDRHDEcDhY=w640-h400-e365-rj-sc0x00ffffff",
        fit: BoxFit.fitHeight,
      );
    },
    itemCount: 10,
    viewportFraction: 0.7,
    scale: 0.8,
  );
}
```

Output:

When we run this app in the IDE, we should get the UI similar to the below screenshot. Here the **image slides automatically**.



Flutter Checkbox

A checkbox is a type of input component which holds the Boolean value. It is a GUI element that allows the user **to choose multiple options from several selections**. Here, a user can answer only in yes or no value. A marked/checked checkbox means yes, and an unmarked/unchecked checkbox means no value. Typically, we can see the checkboxes on the screen as a **square box** with white space or a tick mark. A label or caption corresponding to each checkbox described the meaning of the checkboxes.

In this article, we are going to learn how to use checkboxes in Flutter. In Flutter, we can have two types of checkboxes: a compact version of the Checkbox named **"checkbox"** and the **"CheckboxListTile"** checkbox, which comes with header and subtitle. The detailed descriptions of these checkboxes are given below:

Checkbox:

Attributes	Descriptions
Value	It is used whether the checkbox is checked or not.
onChanged	It will be called when the value is changed.
Tristate	It is false, by default. Its value can also be true, false, or null.
activeColor	It specified the color of the selected checkbox.
checkColor	It specified the color of the check icon when they are selected.
materialTapTargetSize	It is used to configure the size of the tap target.

Example:

Below is the demo example of checkbox:

```
1. Checkbox(
2.   value: this.showvalue,
3.   onChanged: (bool value) {
4.     setState(() {
5.       this.showvalue = value;
6.     });
7.   },
8. ),
```

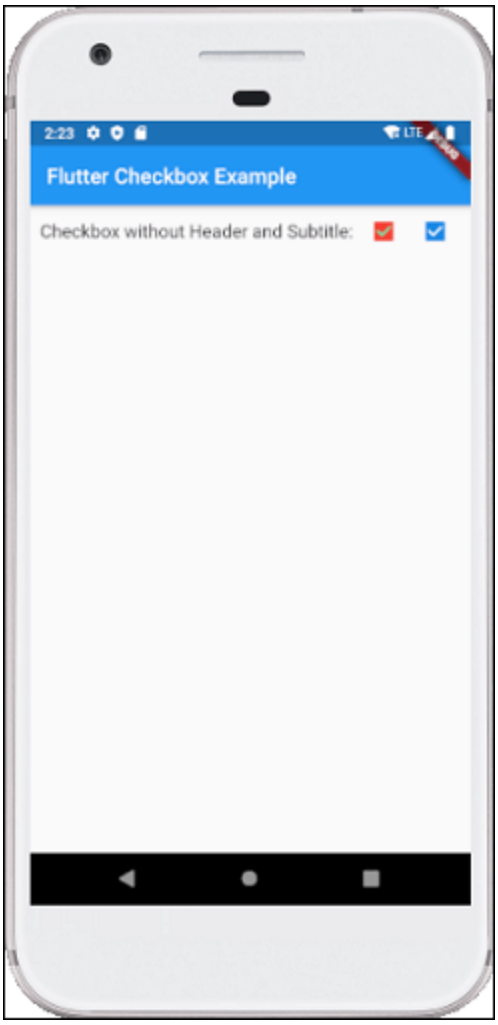
Let us write the complete code to see how checkbox is displayed in Flutter. First, create a project in android studio, open the **main.dart** file, and replace the code given below:

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

void main() {
  runApp(MaterialApp( home: MyHomePage(),));
}

class MyHomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}

class _HomePageState extends State<MyHomePage> {
  bool valuefirst = false;
  bool valuesecond = false;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text('Flutter Checkbox Example'),),
        body: Container(

            child: Column(
```

```
            children: <Widget>[
              Row(
               children: <Widget>[
                 SizedBox(width: 10,),
                Text('Checkbox without Header and Subtitle: ',style: TextStyle(fontSize: 17.0),
),
                 Checkbox(
                 checkColor: Colors.greenAccent,
                  activeColor: Colors.red,
                 value: this.valuefirst,
                  onChanged: (bool value) {
                  setState(() {
                     this.valuefirst = value;
                  });
                  },
                ),
                 Checkbox(
                 value: this.valuesecond,
                  onChanged: (bool value) {
                  setState(() {
                     this.valuesecond = value;
                  });
                  },
                ),
                ],
              ),
             ],
           )
         ),
       ),
     );
  }
}
```

Output

Now execute the app in the emulator or device, we will see the below screen:



CheckboxListTitle:

Attributes	Descriptions
value	It is used whether the checkbox is checked or not.
onChanged	It will be called when the value is changed.
titile	It specified the main title of the list.

subtitle	It specified the subtitle of the list. Usually, it is used to add the description.
activeColor	It specified the color of the selected checkbox.
activeColor	It specified the color of the selected checkbox.
selected	By default, it is false. It highlights the text after selection.
secondary	It is the widget, which is displayed in front of the checkbox.

Example:

Below is the demo example of CheckboxListTitle:

```
1. CheckboxListTile(
2.   secondary: const Icon(Icons.abc),
3.   title: const Text('demo mode'),
4.     subtitle: Text('sub demo mode'),
5.   value: this.subvalue,
6.   onChanged: (bool value) {
7.     setState(() {
8.       this.subvalue = value;
9.     });
10.  },
11. ),
```

Let us write the complete code to see how CheckboxListTitle is displayed in Flutter. First, create a project in android studio, open the **main.dart** file, and replace the code given below:

```
1. import 'package:flutter/material.dart';
2.
3. void main() {
4.   runApp(MaterialApp( home: MyHomePage(),));
5. }
```

```
6.
7.  class MyHomePage extends StatefulWidget {
8.    @override
9.    _HomePageState createState() => _HomePageState();
10. }
11.
12. class _HomePageState extends State<MyHomePage> {
13.  bool valuefirst = false;
14.  bool valuesecond = false;
15.
16.  @override
17.  Widget build(BuildContext context) {
18.   return MaterialApp(
19.     home: Scaffold(
20.      appBar: AppBar(title: Text('Flutter Checkbox Example'),),
21.       body: Container(
22.          padding: new EdgeInsets.all(22.0),
23.           child: Column(
24.            children: <Widget>[
25.              SizedBox(width: 10,),
26.             Text('Checkbox with Header and Subtitle',style: TextStyle(fontSize: 20.0), ),
27.              CheckboxListTile(
28.               secondary: const Icon(Icons.alarm),
29.                title: const Text('Ringing at 4:30 AM every day'),
30.               subtitle: Text('Ringing after 12 hours'),
31.                value: this.valuefirst,
32.               onChanged: (bool value) {
33.                  setState(() {
34.                  this.valuefirst = value;
35.                  });
36.               },
37.               ),
38.             CheckboxListTile(
39.                controlAffinity: ListTileControlAffinity.trailing,
```

```
40.                 secondary: const Icon(Icons.alarm),
41.                  title: const Text('Ringing at 5:00 AM every day'),
42.                 subtitle: Text('Ringing after 12 hours'),
43.                  value: this.valuesecond,
44.                 onChanged: (bool value) {
45.                    setState(() {
46.                    this.valuesecond = value;
47.                    });
48.                 },
49.                ),
50.              ],
51.            )
52.         ),
53.       ),
54.     );
55.   }
56. }
```

Output

Now execute the app in the emulator or device, we will get the following screen:

Flutter Radio Button

A radio button is also known as the options button which holds the **Boolean value**. It allows the user to choose only one option from a predefined set of options. This feature makes it different from a checkbox where we can select more than one option and the unselected state to be restored. We can arrange the radio button in a **group of two or more** and displayed on the screen as **circular holes** with white space (for unselected) or a dot (for selected). We can also provide a **label** for each corresponding radio button describing the choice that the radio button represents. A radio button can be selected by clicking the mouse on the circular hole or using a keyboard shortcut.

In this section, we are going to explain how to use radio buttons in Flutter. Flutter allows us to use radio buttons with the help of 'Radio', 'RadioListTile', or 'ListTitle' Widgets.

The flutter radio button does not maintain any state itself. When we select any radio option, it invokes the **onChanged** callback and passing the value as a parameter. If the value and **groupValue** match, the radio option will be selected.

Let us see how we can create radio buttons in the Flutter app through the following steps:

Step 1: Create a Flutter project in the IDE. Here, I am going to use Android Studio.

Step 2: Open the project in Android Studio and navigate to the **lib** folder. In this folder, open the **main.dart** file and create a RadioButtonWidget class (**Here: MyStatefulWidget**). Next, we will create the Column widget and put three RadioListTile components. Also, we will create a Text widget for displaying the selected item. The ListTitle contains the following properties:

groupValue: It is used to specify the currently selected item for the radio button group.

title: It is used to specify the radio button label.

value: It specifies the backhand value, which is represented by a radio button.

onChanged: It will be called whenever the user selects the radio button.

```
ListTile(
  title: const Text('www.javatpoint.com'),
  leading: Radio(
    value: BestTutorSite.javatpoint,
    groupValue: _site,
    onChanged: (BestTutorSite value) {
      setState(() {
        _site = value;
      });
    },
  ),
),
```

Let us see the complete code of the above steps. Open the **main.dart** file and replace the following code.

Here, the Radio widgets wrapped in **ListTiles** and the currently selected text is passed into groupValue and maintained by the example's State. Here, the first Radio button will be selected off because _site is initialized to **BestTutorSite.javatpoint**. If the second radio button is pressed, the example's State is updated with **setState**, updating _site to **BestTutorSite.w3schools**. It rebuilds the button with the updated groupValue, and therefore it will select the second button.

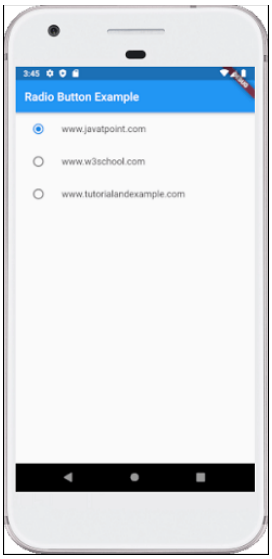
```
import 'package:flutter/material.dart';

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

/// This Widget is the main application widget.
class MyApp extends StatelessWidget {
  static const String _title = 'Radio Button Example';

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: _title,
      home: Scaffold(
```

```
14.         appBar: AppBar(title: const Text(_title)),
15.          body: Center(
16.            child: MyStatefulWidget(),
17.          ),
18.        ),
19.      );
20.    }
21. }
22.
23. enum BestTutorSite { javatpoint, w3schools, tutorialandexample }
24.
25. class MyStatefulWidget extends StatefulWidget {
26.    MyStatefulWidget({Key key}) : super(key: key);
27.
28.    @override
29.    _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
30. }
31.
32. class _MyStatefulWidgetState extends State<MyStatefulWidget> {
33.    BestTutorSite _site = BestTutorSite.javatpoint;
34.
35.    Widget build(BuildContext context) {
36.      return Column(
37.        children: <Widget>[
38.          ListTile(
39.            title: const Text('www.javatpoint.com'),
40.            leading: Radio(
41.              value: BestTutorSite.javatpoint,
42.              groupValue: _site,
43.              onChanged: (BestTutorSite value) {
44.                setState(() {
45.                  _site = value;
46.                });
47.              },
```

```
48.          ),
49.        ),
50.        ListTile(
51.          title: const Text('www.w3school.com'),
52.          leading: Radio(
53.            value: BestTutorSite.w3schools,
54.            groupValue: _site,
55.            onChanged: (BestTutorSite value) {
56.              setState(() {
57.                _site = value;
58.              });
59.            },
60.          ),
61.        ),
62.        ListTile(
63.          title: const Text('www.tutorialandexample.com'),
64.          leading: Radio(
65.            value: BestTutorSite.tutorialandexample,
66.            groupValue: _site,
67.            onChanged: (BestTutorSite value) {
68.              setState(() {
69.                _site = value;
70.              });
71.            },
72.          ),
73.        ),
74.      ],
75.    );
76.  }
77. }
```

Output

When we run the app, the following output appears. Here, we have three radio buttons, and only one is selected by default. We can also select any other option.



Flutter Progress Bar

A progress bar is a graphical control element used to **show the progress of a task** such as downloading, uploading, installation, file transfer, etc. In this section, we are going to understand how to show a progress bar in a flutter application.

Flutter can display the progress bar with the help of two widgets, which are given below:

1. LinearProgressIndicator
2. CircularProgressIndicator

Let us understand it in detail.

LinearProgressIndicator

The linear progress bar is used to show the progress of the task in a **horizontal line**.

Flutter provides mainly **two types** of linear progress indicators:

Determinate: Determinate progress bar indicates the **actual amount of progress at each point** in making the task. Its value will increase monotonically from **0.0 to 1.0** to show the amount of task completed at that time. We need to use a non-null value from 0.0 to 1.0 for creating a determinate progress indicator.

Indeterminate: Indeterminate progress bar does not indicate the amount of progress in completing the task. It means we do not know when the task is finished. **It makes progress without indicating how much progress remains**. We can make an indeterminate progress indicator by using a **null** value.

Properties

The following are the most common attributes of linear progress indicator:

double value: It is used to specify the non-null value between 0.0 to 1.0, representing the completion of task progress.

Color backgroundColor: It is used to specify the background color of the widget.

Animation<Color> valueColor: It is used to specify the progress indicator's color as an animated value.

Example

The following code explains the use of an **indeterminate linear progress bar** that shows a download where we do not know when it will be finished. A **floating button** is used to change the state from not downloading to downloading. When there is no downloading, it shows a text; otherwise, it will show the progress indicator:

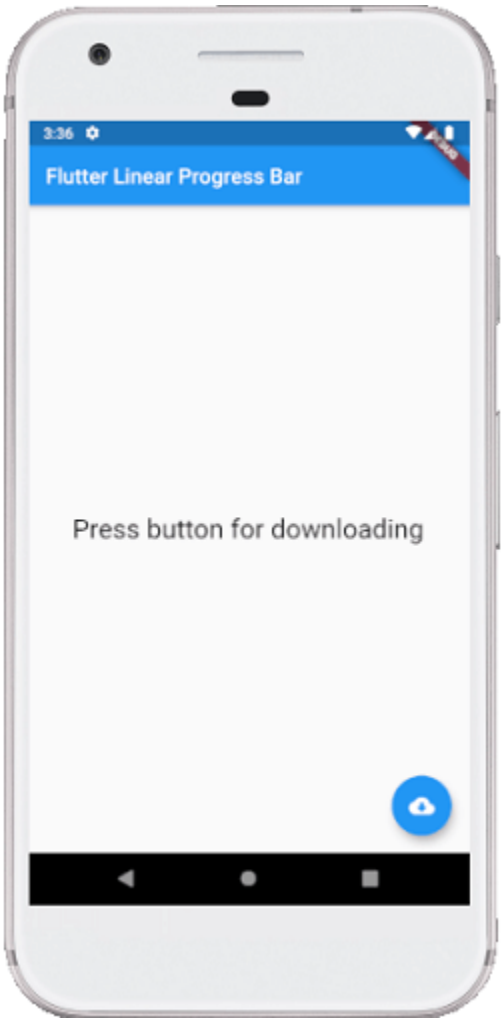
```
1. import 'package:flutter/material.dart';
2.
3. void main() => runApp(MyApp());
4.
5. class MyApp extends StatelessWidget {
6.   @override
7.   Widget build(BuildContext context) {
8.     return MaterialApp(
9.       home: LinearProgressIndicatorApp(),
10.    );
11.  }
12. }
13.
14. class LinearProgressIndicatorApp extends StatefulWidget {
```

```
15.  @override
16.  State<StatefulWidget> createState() {
17.    return LinearProgressIndicatorAppState();
18.  }
19. }
20.
21. class LinearProgressIndicatorAppState extends State<LinearProgressIndicatorApp> {
22.  bool _loading;
23.
24.  @override
25.  void initState() {
26.    super.initState();
27.    _loading = false;
28.  }
29.
30.  @override
31.  Widget build(BuildContext context) {
32.    return Scaffold(
33.      appBar: AppBar(
34.        title: Text("Flutter Linear Progress Bar"),
35.      ),
36.      body: Center(
37.        child: Container(
38.          padding: EdgeInsets.all(12.0),
39.          child: _loading ? LinearProgressIndicator() : Text(
40.              "Press button for downloading",
41.              style: TextStyle(fontSize: 25)),
42.        ),
43.      ),
44.      floatingActionButton: FloatingActionButton(
45.        onPressed: () {
46.          setState(() {
47.            _loading = !_loading;
48.          });
```

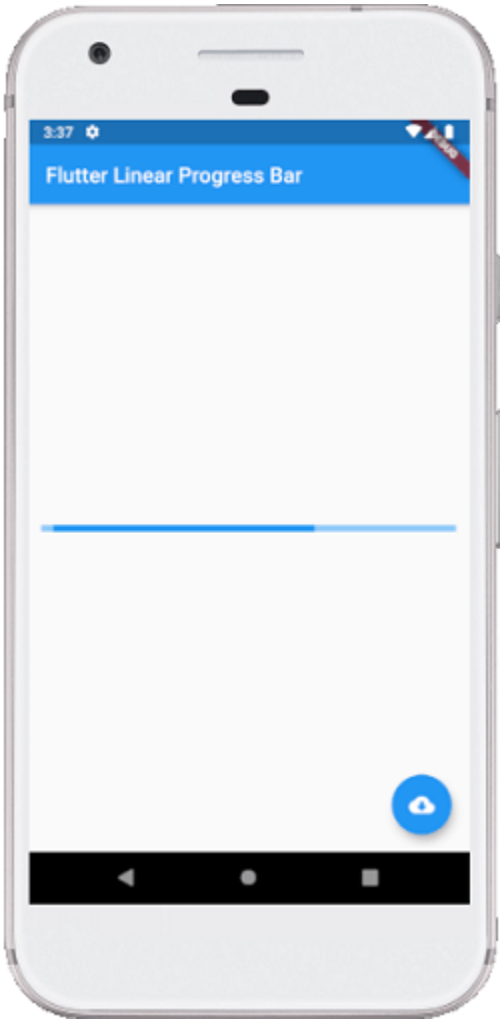
```
49.         },
50.         tooltip: 'Download',
51.         child: Icon(Icons.cloud_download),
52.       ),
53.     );
54.   }
55. }
```

Output:

Now, run the app in your IDE. We can see the UI of the screen as below screenshot.



When we press the floating button, it changes the state from not downloading to downloading and shows the progress indicator like the below screenshot:



Sometimes we want to make a **determinate progress bar** that means we will show how long it will take time to finish the task. In that case, we can simulate a download that will take time to finish the task and **updates** the value of **LinearProgressIndicator** as follows:

```
1.  import 'dart:async';
2.  import 'package:flutter/material.dart';
3.
4.  void main() => runApp(MyApp());
5.
6.  class MyApp extends StatelessWidget {
7.    @override
8.    Widget build(BuildContext context) {
9.      return MaterialApp(
10.       home: LinearProgressIndicatorApp(),
```

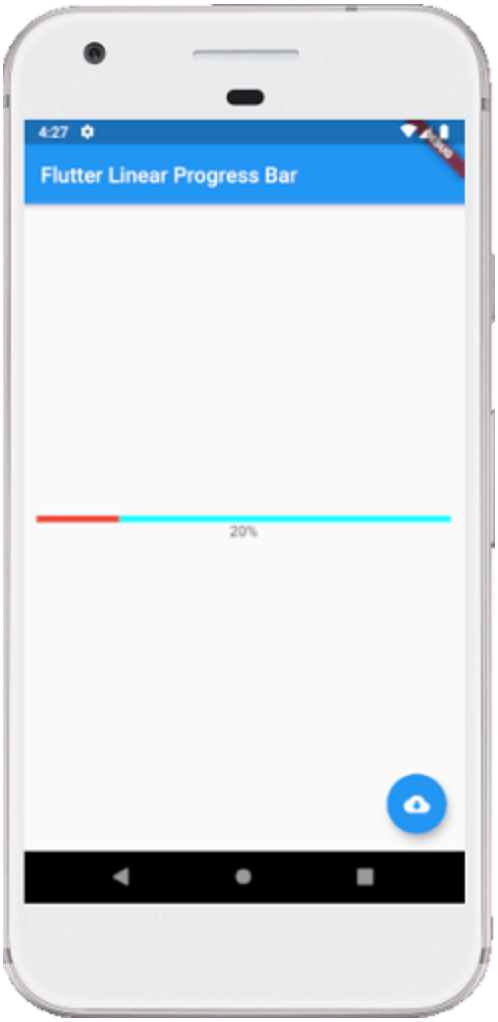
```
11.     );
12.   }
13. }
14.
15. class LinearProgressIndicatorApp extends StatefulWidget {
16.   @override
17.   State<StatefulWidget> createState() {
18.     return LinearProgressIndicatorAppState();
19.   }
20. }
21.
22. class LinearProgressIndicatorAppState extends State<LinearProgressIndicatorApp> {
23.   bool _loading;
24.   double _progressValue;
25.
26.   @override
27.   void initState() {
28.     super.initState();
29.     _loading = false;
30.     _progressValue = 0.0;
31.   }
32.   @override
33.   Widget build(BuildContext context) {
34.     return Scaffold(
35.       appBar: AppBar(
36.         title: Text("Flutter Linear Progress Bar"),
37.       ),
38.       body: Center(
39.         child: Container(
40.           padding: EdgeInsets.all(12.0),
41.           child: _loading
42.               ? Column(
43.             mainAxisAlignment: MainAxisAlignment.center,
44.             children: <Widget>[
```

```
45.              LinearProgressIndicator(
46.               backgroundColor: Colors.cyanAccent,
47.                valueColor: new AlwaysStoppedAnimation<Color>(Colors.red),
48.               value: _progressValue,
49.              ),
50.             Text('${(_progressValue * 100).round()}%'),
51.            ],
52.          )
53.              : Text("Press button for downloading", style: TextStyle(fontSize: 25)),
54.        ),
55.      ),
56.      floatingActionButton: FloatingActionButton(
57.        onPressed: () {
58.         setState(() {
59.           _loading = !_loading;
60.           _updateProgress();
61.         });
62.        },
63.        tooltip: 'Download',
64.        child: Icon(Icons.cloud_download),
65.      ),
66.    );
67.  }
68.  // this function updates the progress value
69.  void _updateProgress() {
70.    const oneSec = const Duration(seconds: 1);
71.    new Timer.periodic(oneSec, (Timer t) {
72.      setState(() {
73.        _progressValue += 0.1;
74.        // we "finish" downloading here
75.        if (_progressValue.toStringAsFixed(1) == '1.0') {
76.          _loading = false;
77.          t.cancel();
78.          return;
```

```
79.         }
80.       });
81.     });
82.   }
83. }
```

Output:

Now, run the app in your IDE. When we press the button, it changes the state from not downloading to downloading and shows how much progress is finished at that time like the below screenshot:



CircularProgressIndicator

It is a widget, which **spins to indicate the waiting process** in your application. It shows the progress of a task in a **circular shape**. It also displays the progress bar in two ways: Determinate and Indeterminate.

A **determinate progress bar** is used when we want to show the progress of ongoing tasks such as the percentage of downloading or uploading files, etc. We can show the progress by specifying the value between 0.0 and 1.0.

An **indeterminate progress bar** is used when we do not want to know the percentage of an ongoing process. By default, CircularProgressIndicator shows the indeterminate progress bar.

Example

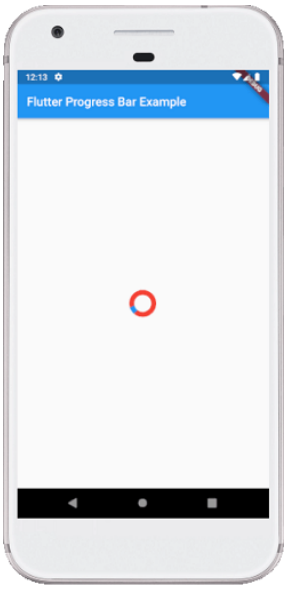
In the below example, we will see the **circular progress** indicator in an indeterminate mode that does not show any task's progress. It displays the circles continuously, which indicates that something is being worked out, and we have to wait for its completion. For this, there is no need to specify any value to the CircularProgressIndicator() constructor. See the following code:

```
1.  import 'package:flutter/material.dart';
2.
3.  void main() => runApp(MyApp());
4.
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 Progress Bar Example'),
12.         ),
13.         body: Center(
14.           child: CircularProgressIndicatorApp()
15.         ),
16.       ),
17.     );
```

```
18.   }
19. }
20.
21. /// This is the stateless widget that the main application instantiates.
22. class CircularProgressIndicatorApp extends StatelessWidget {
23.   @override
24.   Widget build(BuildContext context) {
25.     return CircularProgressIndicator(
26.       backgroundColor: Colors.red,
27.       strokeWidth: 8,);
28.   }
29. }
```

Output:

Now, run the app in your IDE. We will see the output of the flutter circular progress indicator like the below screenshot:



Sometimes you want to make a **determinate circular progress** bar to show how much it will take time to finish the task. In that case, we can simulate a download that will take time to finish the task and updates the value of CircularProgressIndicator as follows:

```
1. import 'dart:async';
2. import 'package:flutter/material.dart';
```

```
3.
4.  void main() => runApp(MyApp());
5.
6.  class MyApp extends StatelessWidget {
7.    @override
8.    Widget build(BuildContext context) {
9.      return MaterialApp(
10.       home: CircularProgressIndicatorApp(),
11.     );
12.   }
13. }
14.
15. class CircularProgressIndicatorApp extends StatefulWidget {
16.   @override
17.   State<StatefulWidget> createState() {
18.     return CircularProgressIndicatorAppState();
19.   }
20. }
21.
22. class CircularProgressIndicatorAppState extends State<CircularProgressIndicatorApp>{
23.   bool _loading;
24.   double _progressValue;
25.
26.   @override
27.   void initState() {
28.     super.initState();
29.     _loading = false;
30.     _progressValue = 0.0;
31.   }
32.   @override
33.   Widget build(BuildContext context) {
34.     return Scaffold(
35.       appBar: AppBar(
36.         title: Text("Flutter Circular Progress Bar"),
```

```
    ),
    body: Center(
      child: Container(
       padding: EdgeInsets.all(14.0),
        child: _loading
          ? Column(
           mainAxisAlignment: MainAxisAlignment.center,
           children: <Widget>[
             CircularProgressIndicator(
              strokeWidth: 10,
               backgroundColor: Colors.yellow,
              valueColor: new AlwaysStoppedAnimation<Color>(Colors.red),
               value: _progressValue,
             ),
              Text('${(_progressValue * 100).round()}%'),
           ],
          )
          : Text("Press button for downloading", style: TextStyle(fontSize: 25)),
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: () {
        setState(() {
          _loading = !_loading;
          _updateProgress();
        });
      },
      child: Icon(Icons.cloud_download),
    ),
  );
}
// this function updates the progress value
void _updateProgress() {
  const oneSec = const Duration(seconds: 1);
```

```
71.    new Timer.periodic(oneSec, (Timer t) {
72.      setState(() {
73.        _progressValue += 0.2;
74.        // we "finish" downloading here
75.        if (_progressValue.toStringAsFixed(1) == '1.0') {
76.          _loading = false;
77.          t.cancel();
78.          return;
79.        }
80.      });
81.    });
82.  }
83. }
```

Output:

Now, run the app in your IDE. When we press the button, it shows how much progress is finished at that time like the below screenshot:

Flutter Lists

Lists are the most popular elements of every web or mobile application. They are made up of multiple rows of items, which include text, buttons, toggles, icons, thumbnails, and many more. We can use it for displaying various information such as menus, tabs, or to break the monotony of pure text files.

In this section, we are going to learn how we can work with Lists in the Flutter. Flutter allows you to work with Lists in different ways, which are given below:

- Basic Lists
- Long Lists
- Grid Lists
- Horizontal Lists

Let us see all the above lists one by one.

Basic Lists

Flutter includes a **ListView** widget for working with Lists, which is the fundamental concept of displaying data in the mobile apps. The ListView is a perfect standard for displaying lists that contains only a few items. ListView also includes **ListTitle** widget, which gives more properties for the visual structure to a list of data.

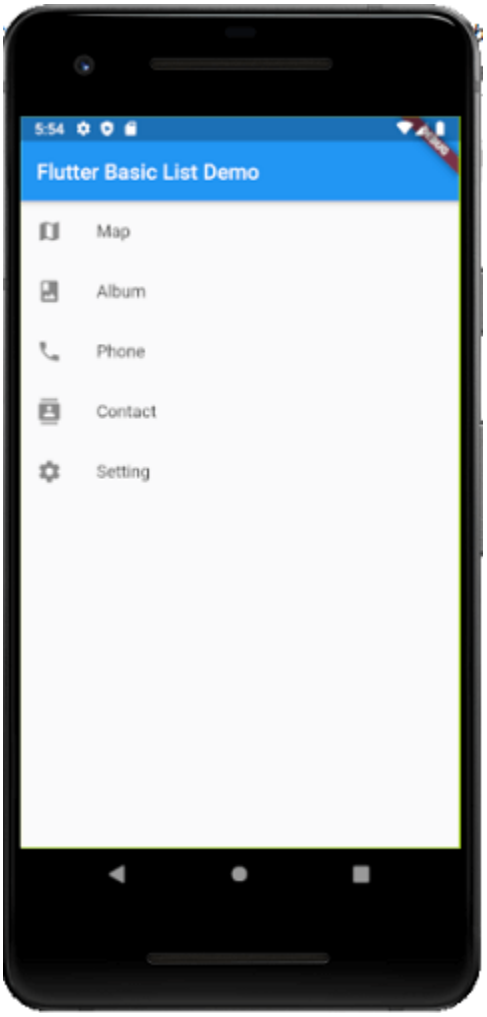
The following example displays a basic list in the Flutter application.

```
1. import 'package:flutter/material.dart';
2.
3. void main() => runApp(MyApp());
4.
5. class MyApp extends StatelessWidget {
6.   @override
7.   Widget build(BuildContext context) {
8.     final appTitle = 'Flutter Basic List Demo';
9.
10.    return MaterialApp(
11.      title: appTitle,
12.      home: Scaffold(
13.        appBar: AppBar(
14.          title: Text(appTitle),
15.        ),
16.        body: ListView(
```

```
17.          children: <Widget>[
18.           ListTile(
19.             leading: Icon(Icons.map),
20.             title: Text('Map'),
21.           ),
22.           ListTile(
23.             leading: Icon(Icons.photo_album),
24.             title: Text('Album'),
25.           ),
26.           ListTile(
27.             leading: Icon(Icons.phone),
28.             title: Text('Phone'),
29.           ),
30.           ListTile(
31.             leading: Icon(Icons.contacts),
32.             title: Text('Contact'),
33.           ),
34.           ListTile(
35.             leading: Icon(Icons.settings),
36.             title: Text('Setting'),
37.           ),
38.         ],
39.       ),
40.     ),
41.   );
42.  }
43. }
```

Output

Now, run the app in Android Studio. You can see the following screen in your emulator or connected device.



Working with Long Lists

Sometimes you want to display a very long list in a single screen of your app, then, in that case, the above method for displaying the lists is not perfect. To work with a list that contains a very large number of items, we need to use a **ListView.builder()** constructor. The main difference between ListView and ListView.builder is that ListView creates all items at once, whereas the ListView.builder() constructor creates items when they are scrolled onto the screen.

Let us see the following example. Open the **main.dart** file and replace the following code.

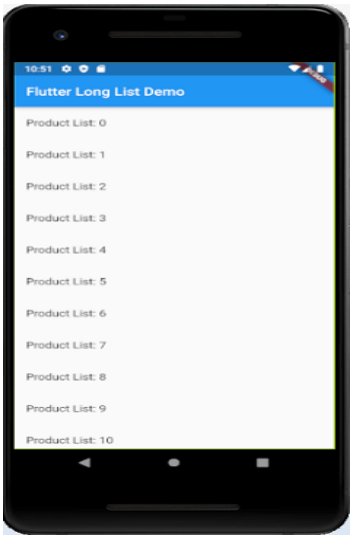
```
1. import 'package:flutter/material.dart';
2.
3. void main() {
4.   runApp(MyApp(
```

```
5.      products: List<String>.generate(500, (i) => "Product List: $i"),
6.    ));
7.  }
8.
9.  class MyApp extends StatelessWidget {
10.   final List<String> products;
11.
12.   MyApp({Key key, @required this.products}) : super(key: key);
13.
14.   @override
15.   Widget build(BuildContext context) {
16.     final appTitle = 'Flutter Long List Demo';
17.
18.     return MaterialApp(
19.       title: appTitle,
20.       home: Scaffold(
21.         appBar: AppBar(
22.           title: Text(appTitle),
23.         ),
24.         body: ListView.builder(
25.           itemCount: products.length,
26.           itemBuilder: (context, index) {
27.             return ListTile(
28.               title: Text('${products[index]}'),
29.             );
30.           },
31.         ),
32.       ),
33.     );
34.   }
35. }
```

In the above code, the **itemCount** gives how many numbers of items you want to display in a list. The **itemBuilder** tells about where you want to return the item that you want to display.

Output

Now, run the app in Android Studio. You will get the following screen where you can see all product list by scrolling onto the screen.



Creating a Grid Lists

Sometimes we want to display the items in a grid layout rather than the normal list that comes one after next. A **GridView** widget allows you to create a grid list in Flutter. The simplest way to create a grid is by using the **GridView.count()** constructor, which specifies the number of rows and columns in a grid.

Let us see the following example of how GridView works in Flutter for creating grid lists. Open the main.dart file and insert the following code.

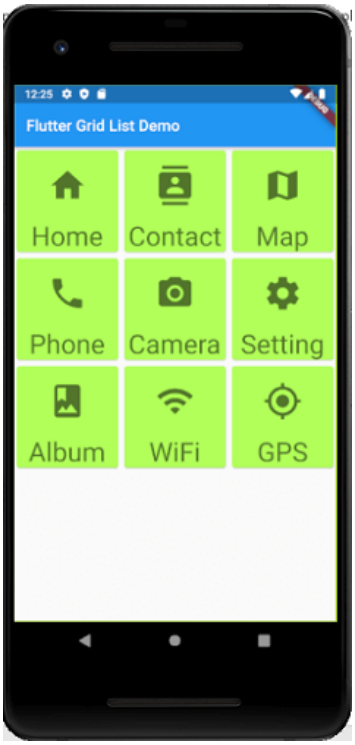
```
1. import 'package:flutter/material.dart';
2.
3. void main() {runApp(MyApp());}
4.
5. class MyApp extends StatelessWidget {
6.   @override
7.   Widget build(BuildContext context) {
8.     final appTitle = "Flutter Grid List Demo";
9.
10.    return MaterialApp(
```

```
11.         title: appTitle,
12.         home: Scaffold(appBar: AppBar(
13.           title: Text(appTitle),
14.         ),
15.             body: GridView.count(
16.               crossAxisCount: 3,
17.                children: List.generate(choices.length, (index) {
18.                 return Center(
19.                     child: SelectCard(choice: choices[index]),
20.                 );
21.                }
22.               )
23.             )
24.         )
25.     );
26.   }
27. }
28.
29. class Choice {
30.   const Choice({this.title, this.icon});
31.   final String title;
32.   final IconData icon;
33. }
34.
35. const List<Choice> choices = const <Choice>[
36.   const Choice(title: 'Home', icon: Icons.home),
37.   const Choice(title: 'Contact', icon: Icons.contacts),
38.   const Choice(title: 'Map', icon: Icons.map),
39.   const Choice(title: 'Phone', icon: Icons.phone),
40.   const Choice(title: 'Camera', icon: Icons.camera_alt),
41.   const Choice(title: 'Setting', icon: Icons.settings),
42.   const Choice(title: 'Album', icon: Icons.photo_album),
43.   const Choice(title: 'WiFi', icon: Icons.wifi),
44.   const Choice(title: 'GPS', icon: Icons.gps_fixed),
```

```
];

class SelectCard extends StatelessWidget {
  const SelectCard({Key key, this.choice}) : super(key: key);
  final Choice choice;

  @override
  Widget build(BuildContext context) {
    final TextStyle textStyle = Theme.of(context).textTheme.display1;
    return Card(
        color: Colors.lightGreenAccent,
        child: Center(child: Column(
            mainAxisSize: MainAxisSize.min,
            crossAxisAlignment: CrossAxisAlignment.center,
            children: <Widget>[
              Expanded(child: Icon(choice.icon, size:50.0, color: textStyle.color)),
              Text(choice.title, style: textStyle),
            ]
        ),
        )
    );
  }
}
```

Output

Now, run the app in Android Studio. You can see the following screen in your Android Emulator.



Creating a Horizontal List

The ListView widget also supports horizontal lists. Sometimes we want to create a list that can scroll horizontally rather than vertically. In that case, ListView provides the horizontal **scrollDirection** that overrides the vertical direction. The following example explains it more clearly. Open the main.dart file and replace the following code.

```
1.  import 'package:flutter/material.dart';
2.
3.  void main() => runApp(MyApp());
4.
5.  class MyApp extends StatelessWidget {
6.    @override
7.    Widget build(BuildContext context) {
8.      final title = 'Flutter Horizontal Demo List';
9.
10.     return MaterialApp(
11.       title: title,
12.       home: Scaffold(
```

```
13.       appBar: AppBar(
14.        title: Text(title),
15.       ),
16.       body: Container(
17.         margin: EdgeInsets.symmetric(vertical: 25.0),
18.        height: 150.0,
19.         child: ListView(
20.          scrollDirection: Axis.horizontal,
21.           children: <Widget>[
22.            Container(
23.              width: 150.0,
24.             color: Colors.blue,
25.              child: new Stack(
26.               children: <Widget>[
27.                 ListTile(
28.                 leading: Icon(Icons.home),
29.                   title: Text('Home'),
30.                ),
31.                ],
32.             ),
33.            ),
34.            Container(
35.              width: 148.0,
36.             color: Colors.green,
37.              child: new Stack(
38.               children: <Widget>[
39.                 ListTile(
40.                 leading: Icon(Icons.camera_alt),
41.                   title: Text('Camera'),
42.                ),
43.                ],
44.             ),
45.            ),
46.            Container(
```

```
47.              width: 148.0,
48.             color: Colors.yellow,
49.              child: new Stack(
50.               children: <Widget>[
51.                 ListTile(
52.                 leading: Icon(Icons.phone),
53.                   title: Text('Phone'),
54.                 ),
55.                ],
56.             ),
57.            ),
58.            Container(
59.              width: 148.0,
60.             color: Colors.red,
61.              child: new Stack(
62.               children: <Widget>[
63.                 ListTile(
64.                 leading: Icon(Icons.map),
65.                   title: Text('Map'),
66.                 ),
67.                ],
68.             ),
69.            ),
70.            Container(
71.              width: 148.0,
72.             color: Colors.orange,
73.              child: new Stack(
74.               children: <Widget>[
75.                 ListTile(
76.                 leading: Icon(Icons.settings),
77.                   title: Text('Setting'),
78.                 ),
79.                ],
80.             ),
```

```
81.                 ),
82.               ],
83.             ),
84.           ),
85.         ),
86.       );
87.   }
88. }
```

Output

Now, run the app in Android Studio. It will give the following screen where you can scroll horizontally to see all the lists.

Flutter Stack

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

Key Points Related to Stack Widget

The following are the key points of the Flutter stack widget:

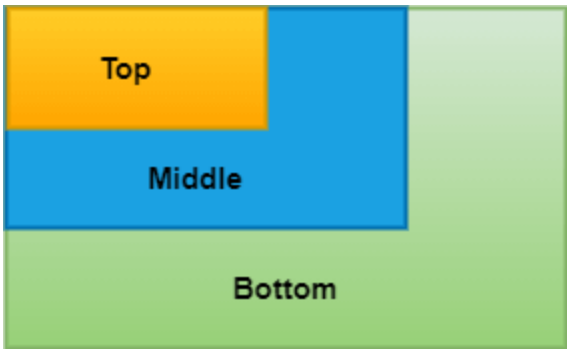
- The child widget in a stack can be either **positioned** or **non-positioned**.
- Positioned items are wrapped in Positioned widget and must have a one non-null property
- The non-positioned child widgets are aligned itself. It displays on the screen based on the stack's alignment. The default position of the children is in the top left corner.
- We can use the alignment attribute to change the alignment of the widgets.
- Stack places the children widgets in order with the first child being at the bottom and the last child being at the top. If we want to reorder the children's widget, it is required to rebuild the stack in the new order. By default, the **first widget of each stack has the maximum size** compared to other widgets.

How to use a stack widget in Flutter?

The below example helps to understand the use of stack widget quickly that contains three containers of shrinking size:

```
Stack(
  children: <Widget>[
    // Max Size
    Container(
      color: Colors.green,
    ),
    Container(
      color: Colors.blue,
    ),
    Container(
      color: Colors.yellow,
    )
  ],
),
```

It will give the following output:



Properties of the Stack Widget

The following are the properties used with the stack widget:

alignment: It determines how the children widgets are positioned in the stack. It can be top, bottom, center, center-right, etc.

```
1. Stack(
2.   alignment: Alignment.topCenter, // Center of Top
3.   children: <Widget>[ ]
4. )
```

textDirection: It determines the text direction. It can draw the text either ltr (left to right) or rtl (right to the left).

```
1. Stack(
2.   textDirection: TextDirection.rtl, // Right to Left
3.   children: <Widget>[ ]
4. )
```

fit: It will control the size of non-positioned children widgets in the stack. **It has three types**: loose, expand and passthrough. The **loose** used to set the child widget small, the **expand** attribute makes the child widget as large as possible, and the **passthrough** set the child widget depending on its parent widget.

```
1. Stack(
2.   fit: StackFit.passthrough,
3.   children: <Widget>[ ]
4. )
```

overflow: It controls the children widgets, whether visible or clipped, when it's content overflowing outside the stack.

```
Stack(
  overflow: Overflow.clip, // Clip the Content
  children: <Widget>[ ]
)
```

clipBehavior: It determines whether the content will be clipped or not.

Positioned

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:

```
const Positioned({
Key key,
this.left,
this.top,
this.right,
this.bottom,
this.width,
this.height,
@required Widget child,
})
```

Stack Widget Example

The below code explains how to use the stack widget in Flutter. In this code, we are going to try most of the essential attributes of the stack widget.

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

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

/// This Widget is the main application widget.
class MyApp extends StatelessWidget {
  @override
```

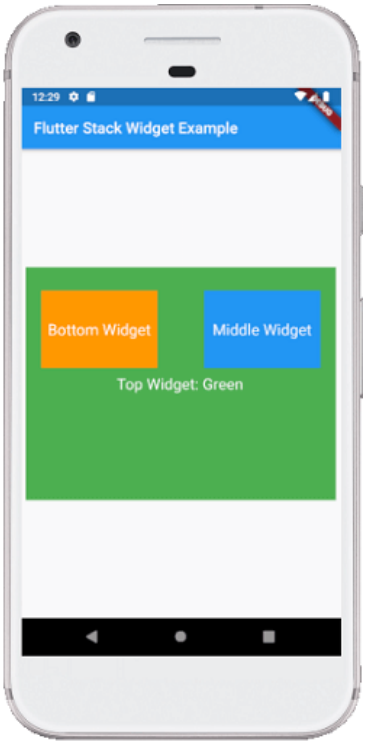
```
8.    Widget build(BuildContext context) {
9.      return MaterialApp(
10.       home: MyStackWidget(),
11.     );
12.   }
13. }
14.
15. class MyStackWidget extends StatelessWidget {
16.   @override
17.   Widget build(BuildContext context) {
18.     return MaterialApp(
19.       home: Scaffold(
20.         appBar: AppBar(
21.           title: Text("Flutter Stack Widget Example"),
22.         ),
23.         body: Center(
24.           child: Stack(
25.             fit: StackFit.passthrough,
26.             overflow: Overflow.visible,
27.             children: <Widget>[
28.               // Max Size Widget
29.               Container(
30.                 height: 300,
31.                 width: 400,
32.                 color: Colors.green,
33.                 child: Center(
34.                   child: Text(
35.                     'Top Widget: Green',
36.                     style: TextStyle(color: Colors.white, fontSize: 20),
37.                   ),
38.                 ),
39.               ),
40.               Positioned(
41.                 top: 30,
```

```
42.                 right: 20,
43.                  child: Container(
44.                  height: 100,
45.                    width: 150,
46.                  color: Colors.blue,
47.                    child: Center(
48.                    child: Text(
49.                       'Middle Widget',
50.                     style: TextStyle(color: Colors.white, fontSize: 20),
51.                     ),
52.                   ),
53.                  ),
54.               ),
55.                Positioned(
56.                 top: 30,
57.                  left: 20,
58.                 child: Container(
59.                    height: 100,
60.                  width: 150,
61.                    color: Colors.orange,
62.                  child: Center(
63.                     child: Text(
64.                     'Bottom Widget',
65.                      style: TextStyle(color: Colors.white, fontSize: 20),
66.                    ),
67.                    ),
68.                )
69.                ),
70.             ],
71.           ),
72.         )
73.       ),
74.     );
75.   }
```

```
76. }
```

Output:

When we run the app, we should get the UI of the screen similar to the below screenshot:



Flutter IndexedStack

It is another stack widget in Flutter that ***displayed only one element at one time by specifying its index***. See the below code snippet:

```
1. IndexedStack(
2.   index: 1,
3.   children: <Widget>[
4.     Container(
5.       color: Colors.green,
6.     ),
7.     Container(
8.       color: Colors.blue,
9.     ),
```

```
10.     Container(
11.       color: Colors.yellow,
12.     )
13.   ],
14. )
```

IndexedStack takes children like a usual stack, but it will display only one child at a time. Therefore, it is not a stack. We use it for easily switching between one child to another child according to our needs.

IndexedStack Widget Example

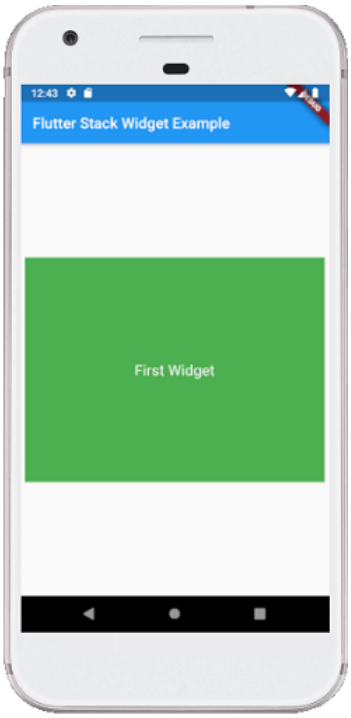
The below code explains how to use indexed stack widget in Flutter:

```
1.  import 'package:flutter/material.dart';
2.
3.  void main() => runApp(MyApp());
4.
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: MyStackWidget(),
11.     );
12.   }
13. }
14.
15. class MyStackWidget extends StatelessWidget {
16.   @override
17.   Widget build(BuildContext context) {
18.     return MaterialApp(
19.       home: Scaffold(
20.         appBar: AppBar(
21.           title: Text("Flutter Stack Widget Example"),
22.         ),
```

```
23.         body: Center(
24.          child: IndexedStack(
25.            index: 0,
26.           children: <Widget>[
27.             Container(
28.              height: 300,
29.               width: 400,
30.              color: Colors.green,
31.               child: Center(
32.                child: Text(
33.                  'First Widget',
34.                 style: TextStyle(color: Colors.white, fontSize: 20),
35.                ),
36.              ),
37.             ),
38.            Container(
39.              height: 250,
40.              width: 250,
41.               color: Colors.blue,
42.              child: Center(
43.                child: Text(
44.                'Second Widget',
45.                  style: TextStyle(color: Colors.white, fontSize: 20),
46.               ),
47.              ),
48.            ),
49.           ],
50.         ),
51.        )
52.      ),
53.    );
54.  }
55. }
```

Output:

When we run the app, we should get the UI of the screen similar to the below screenshot:



Is it possible to wrap stack inside stack in Flutter?

Yes, it is possible to wrap stack inside stack in Flutter. We can do this by wrapping the second stack inside the container with height and width property.

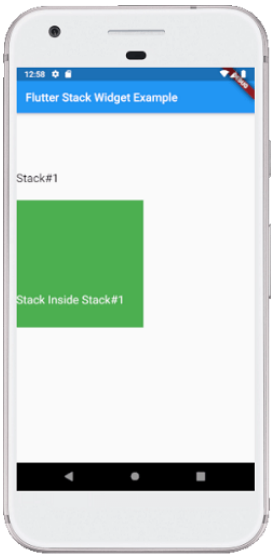
See the below code to understand it more clearly:

```
1.  import 'package:flutter/material.dart';
2.
3.  void main() => runApp(MyApp());
4.
5.  class MyApp extends StatelessWidget {
6.    @override
7.    Widget build(BuildContext context) {
8.      return MaterialApp(
9.        home: MyStackWidget(),
10.     );
```

```
11.   }
12. }
13.
14. class MyStackWidget extends StatelessWidget {
15.   @override
16.   Widget build(BuildContext context) {
17.     return MaterialApp(
18.       home: Scaffold(
19.         appBar: AppBar(
20.           title: Text("Flutter Stack Widget Example"),
21.         ),
22.         body: Center(
23.           child: Stack(
24.             children: [
25.               Positioned(
26.                top: 100,
27.                 child: Text(
28.                   "Stack#1",
29.                     style: TextStyle(color: Colors.black, fontSize: 20)
30.                ),
31.               ),
32.              Positioned(
33.                 top: 150.0,
34.                child: Container(
35.                   height: 220,
36.                  width: 220,
37.                   color: Colors.green,
38.                  child: Stack(
39.                     children: [
40.                     Positioned(
41.                       top:160,
42.                      child: Text(
43.                         "Stack Inside Stack#1",
44.                       style: TextStyle(color: Colors.white, fontSize: 20)
```

```
45.                         ),
46.                     )
47.                     ],
48.                 ),
49.                 ),
50.             )
51.             ],
52.         ),
53.         )
54.     ),
55.   );
56.  }
57. }
```

When we run the app, we should get the UI of the screen similar to the below screenshot:



Flutter Forms

Forms are an integral part of all modern mobile and web applications. It is mainly used to interact with the app as well as gather information from the users. They can perform many tasks, which depend on the nature of your business requirements and logic, such as authentication of the user, adding user, searching, filtering, ordering, booking, etc. A form can contain text fields, buttons, checkboxes, radio buttons, etc.

Creating Form

Flutter provides a **Form widget** to create a form. The form widget acts as a container, which allows us to group and validate the multiple form fields. When you create a form, it is necessary to provide the **GlobalKey**. This key uniquely identifies the form and allows you to do any validation in the form fields.

The form widget uses child widget **TextFormField** to provide the users to enter the text field. This widget renders a material design text field and also allows us to display validation errors when they occur.

Let us create a form. First, create a Flutter project and replace the following code in the main.dart file. In this code snippet, we have created a custom class named **MyCustomForm**. Inside this class, we define a global key as **_formKey**. This key holds a **FormState** and can use to retrieve the form widget. Inside the **build** method of this class, we have added some custom style and use the TextFormField widget to provide the form fields such as name, phone number, date of birth, or just a normal field. Inside the TextFormField, we have used **InputDecoration** that provides the look and feel of your form properties such as borders, labels, icons, hint, styles, etc. Finally, we have added a **button** to submit the form.

```
1.  import 'package:flutter/material.dart';
2.
3.  void main() => runApp(MyApp());
4.
5.  class MyApp extends StatelessWidget {
6.    @override
7.    Widget build(BuildContext context) {
8.      final appTitle = 'Flutter Form Demo';
9.      return MaterialApp(
10.       title: appTitle,
11.       home: Scaffold(
12.         appBar: AppBar(
13.           title: Text(appTitle),
14.         ),
15.         body: MyCustomForm(),
```

```
      ),
    );
  }
}
// Create a Form widget.
class MyCustomForm extends StatefulWidget {
  @override
  MyCustomFormState createState() {
    return MyCustomFormState();
  }
}
// Create a corresponding State class. This class holds data related to the form.
class MyCustomFormState extends State<MyCustomForm> {
  // Create a global key that uniquely identifies the Form widget
  // and allows validation of the form.
  final _formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    // Build a Form widget using the _formKey created above.
    return Form(
      key: _formKey,
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
          TextFormField(
            decoration: const InputDecoration(
              icon: const Icon(Icons.person),
              hintText: 'Enter your name',
              labelText: 'Name',
            ),
          ),
          TextFormField(
            decoration: const InputDecoration(
```

```
50.             icon: const Icon(Icons.phone),
51.             hintText: 'Enter a phone number',
52.             labelText: 'Phone',
53.           ),
54.         ),
55.         TextFormField(
56.           decoration: const InputDecoration(
57.           icon: const Icon(Icons.calendar_today),
58.           hintText: 'Enter your date of birth',
59.           labelText: 'Dob',
60.           ),
61.          ),
62.         new Container(
63.             padding: const EdgeInsets.only(left: 150.0, top: 40.0),
64.             child: new RaisedButton(
65.               child: const Text('Submit'),
66.                 onPressed: null,
67.             )),
68.       ],
69.     ),
70.   );
71.  }
72. }
```

Output

Now, run the app, you can see the following screen in your Android Emulator. This form contains three field name, phone number, date of birth, and submit button.

Form validation

Validation is a method, which allows us to correct or confirms a certain standard. It ensures the authentication of the entered data.

Validating forms is a common practice in all digital interactions. To validate a form in a flutter, we need to implement mainly three steps.

Step 1: Use the Form widget with a global key.

Step 2: Use TextFormField to give the input field with validator property.

Step 3: Create a button to validate form fields and display validation errors.

Let us understand it with the following example. In the above code, we have to use **validator()** function in the TextFormField to validate the input properties. If the user gives the wrong input, the validator function returns a string that contains an **error**

message; otherwise, the validator function return **null**. In the validator function, make sure that the TextFormField is not empty. Otherwise, it returns an error message.

The validator() function can be written as below code snippets:

```
validator: (value) {
      if (value.isEmpty) {
          return 'Please enter some text';
      }
      return null;
},
```

Now, open the **main.dart** file and add validator() function in the TextFormField widget. Replace the following code with the main.dart file.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final appTitle = 'Flutter Form Demo';
    return MaterialApp(
      title: appTitle,
      home: Scaffold(
        appBar: AppBar(
          title: Text(appTitle),
        ),
        body: MyCustomForm(),
      ),
    );
  }
}
// Create a Form widget.
class MyCustomForm extends StatefulWidget {
```

```
22.   @override
23.   MyCustomFormState createState() {
24.     return MyCustomFormState();
25.   }
26. }
27. // Create a corresponding State class, which holds data related to the form.
28. class MyCustomFormState extends State<MyCustomForm> {
29.   // Create a global key that uniquely identifies the Form widget
30.   // and allows validation of the form.
31.   final _formKey = GlobalKey<FormState>();
32.
33.   @override
34.   Widget build(BuildContext context) {
35.     // Build a Form widget using the _formKey created above.
36.     return Form(
37.       key: _formKey,
38.       child: Column(
39.         crossAxisAlignment: CrossAxisAlignment.start,
40.         children: <Widget>[
41.           TextFormField(
42.             decoration: const InputDecoration(
43.               icon: const Icon(Icons.person),
44.               hintText: 'Enter your full name',
45.               labelText: 'Name',
46.             ),
47.             validator: (value) {
48.               if (value.isEmpty) {
49.                 return 'Please enter some text';
50.               }
51.               return null;
52.             },
53.           ),
54.           TextFormField(
55.             decoration: const InputDecoration(
```

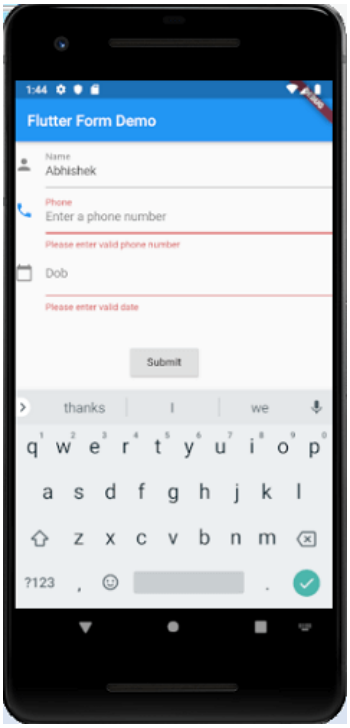
```
56.             icon: const Icon(Icons.phone),
57.             hintText: 'Enter a phone number',
58.             labelText: 'Phone',
59.           ),
60.           validator: (value) {
61.             if (value.isEmpty) {
62.               return 'Please enter valid phone number';
63.             }
64.             return null;
65.           },
66.         ),
67.         TextFormField(
68.           decoration: const InputDecoration(
69.           icon: const Icon(Icons.calendar_today),
70.           hintText: 'Enter your date of birth',
71.           labelText: 'Dob',
72.           ),
73.           validator: (value) {
74.             if (value.isEmpty) {
75.               return 'Please enter valid date';
76.             }
77.             return null;
78.           },
79.          ),
80.         new Container(
81.             padding: const EdgeInsets.only(left: 150.0, top: 40.0),
82.             child: new RaisedButton(
83.               child: const Text('Submit'),
84.               onPressed: () {
85.                 // It returns true if the form is valid, otherwise returns false
86.                 if (_formKey.currentState.validate()) {
87.                   // If the form is valid, display a Snackbar.
88.                   Scaffold.of(context)
89.                       .showSnackBar(SnackBar(content: Text('Data is in processing.')));
```

```
90.                     }
91.                   },
92.                 )),
93.           ],
94.         ),
95.     );
96.   }
97. }
```

Output

Now, run the app. The following screen appears.

In this form, if you left any input field blank, you will get an error message like below screen.



Flutter Alert Dialogs

An alert dialog is a useful feature that notifies the user with important information to make a decision or provide the ability to choose a specific action or list of actions. It is a pop-up box that appears at the top of the app content and the middle of the screen. It can be dismissed manually by the user before resuming interaction with the app.

An alert can be thought of as a floating modal which should be used for a quick response such as password verification, small app notifications, and many more. The alerts are very flexible and can be customized very easily.

In Flutter, the AlertDialog is a widget, which informs the user about the situations that need acknowledgment. The Flutter alert dialog contains an optional title that displayed above the content and list of actions displayed below the content.

Properties of Alert Dialog

The main properties of the AlertDialog widget are:

Title: This property gives the title to an AlertDialog box that occupies at the top of the AlertDialog. It is always good to keep the title as short as possible so that the user knows about its use very easily. We can write the title in AlertDialog as below:

```
1. AlertDialog(title: Text("Sample Alert Dialog"),
```

Action: It displays below the content. For example, if there is a need to create a button to choose yes or no, then it is defined in the action property only. We can write an action attribute in AlertDialog as below:

```
1. actions: <Widget>[
2.     FlatButton(child: Text("Yes"),),
3.     FlatButton(child: Text("No"),)
4. ],)
```

Content: This property defines the body of the AlertDialog widget. It is a type of text, but it can also hold any kind of layout widgets. We can use the Content attribute in AlertDialog as below:

```
1. actions: <Widget>[
2.     FlatButton(child: Text("Yes"),),
3.     FlatButton(child: Text("No"),)
4. ],)
5. content: Text("It is the body of the alert Dialog"),
```

ContentPadding: It gives the padding required for the content inside the AlertDialog widget. We can use ContentPadding attribute in AlertDialog as below:

```
1. contentPadding: EdgeInsets.all(32.0),
```

Shape: This attribute provides the shape to the alert dialog box, such as curve, circle, or any other different shape.

```
1. shape: CircleBorder(),
2. shape: CurveBorder(),
```

We can categories the alert dialog into multiple types, which are given below:

1. Basic AlertDialog

2. Confirmation AlertDialog
3. Select AlertDialog
4. TextField AlertDialog

Basic AlertDialog

This alert notifies the users about new information, such as a change in the app, about new features, an urgent situation that requires acknowledgment, or as a confirmation notification to the user that an action was successful or not. The following example explains the use of basic alerts.

Example

Create a Flutter project in Android Studio and replace the following code with **main.dart** file. To show an alert, you must have to call **showDialog()** function, which contains the context and **itemBuilder** function. The itemBuilder function returns an **object** of type **dialog,** the AlertDialog.

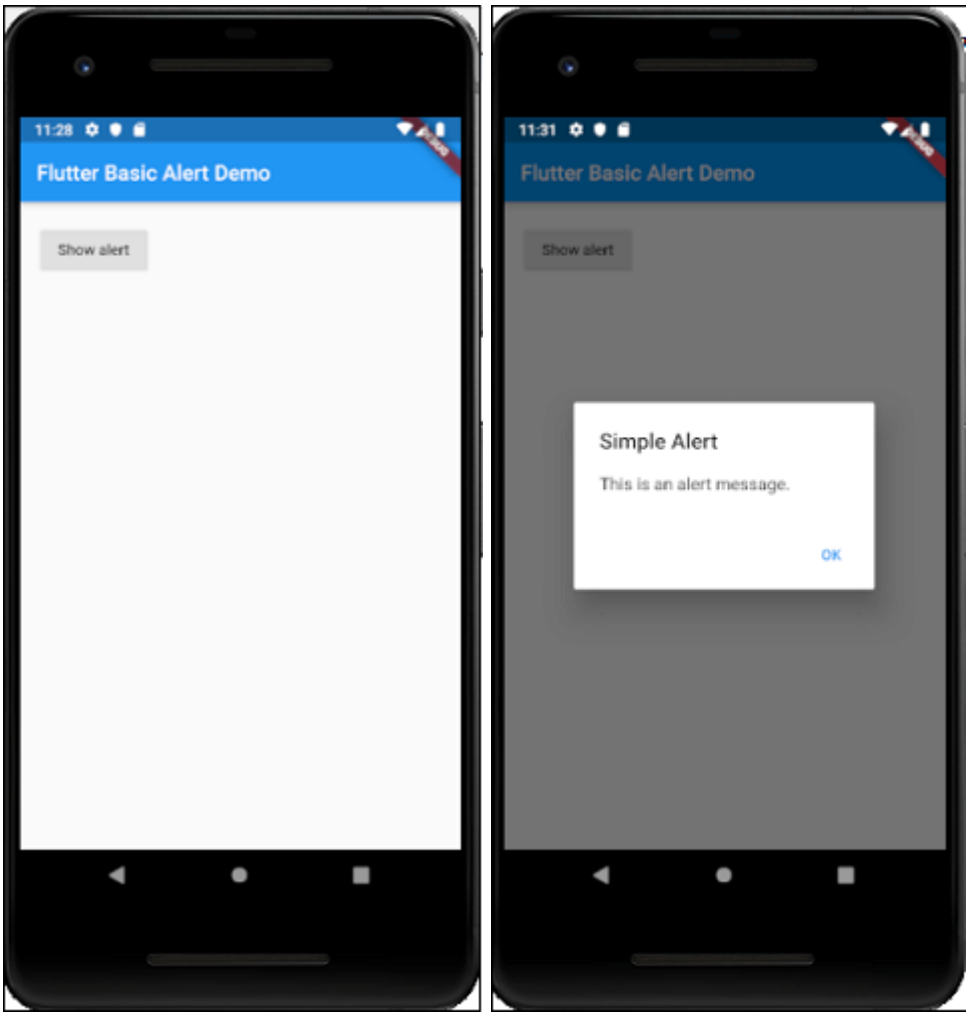
```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final appTitle = 'Flutter Basic Alert Demo';
    return MaterialApp(
      title: appTitle,
      home: Scaffold(
        appBar: AppBar(
          title: Text(appTitle),
        ),
        body: MyAlert(),
      ),
    );
  }
```

```
19. }
20.
21. class MyAlert extends StatelessWidget {
22.   @override
23.   Widget build(BuildContext context) {
24.     return Padding(
25.       padding: const EdgeInsets.all(20.0),
26.       child: RaisedButton(
27.         child: Text('Show alert'),
28.         onPressed: () {
29.           showAlertDialog(context);
30.         },
31.       ),
32.     );
33.   }
34. }
35.
36. showAlertDialog(BuildContext context) {
37.   // Create button
38.   Widget okButton = FlatButton(
39.     child: Text("OK"),
40.     onPressed: () {
41.       Navigator.of(context).pop();
42.     },
43.   );
44.
45.   // Create AlertDialog
46.   AlertDialog alert = AlertDialog(
47.     title: Text("Simple Alert"),
48.     content: Text("This is an alert message."),
49.     actions: [
50.       okButton,
51.     ],
52.   );
```

```
53.
54.   // show the dialog
55.   showDialog(
56.     context: context,
57.     builder: (BuildContext context) {
58.       return alert;
59.     },
60.   );
61. }
```

Output

Now, run the app, it will give the following output. When you click on the button Show Alert, you will get the alert message.



TextField AlertDialog

This AlertDialog makes it able to accept user input. In the following example, we are going to add text field input in the alert dialog. Open the main.dart file and insert the following code.

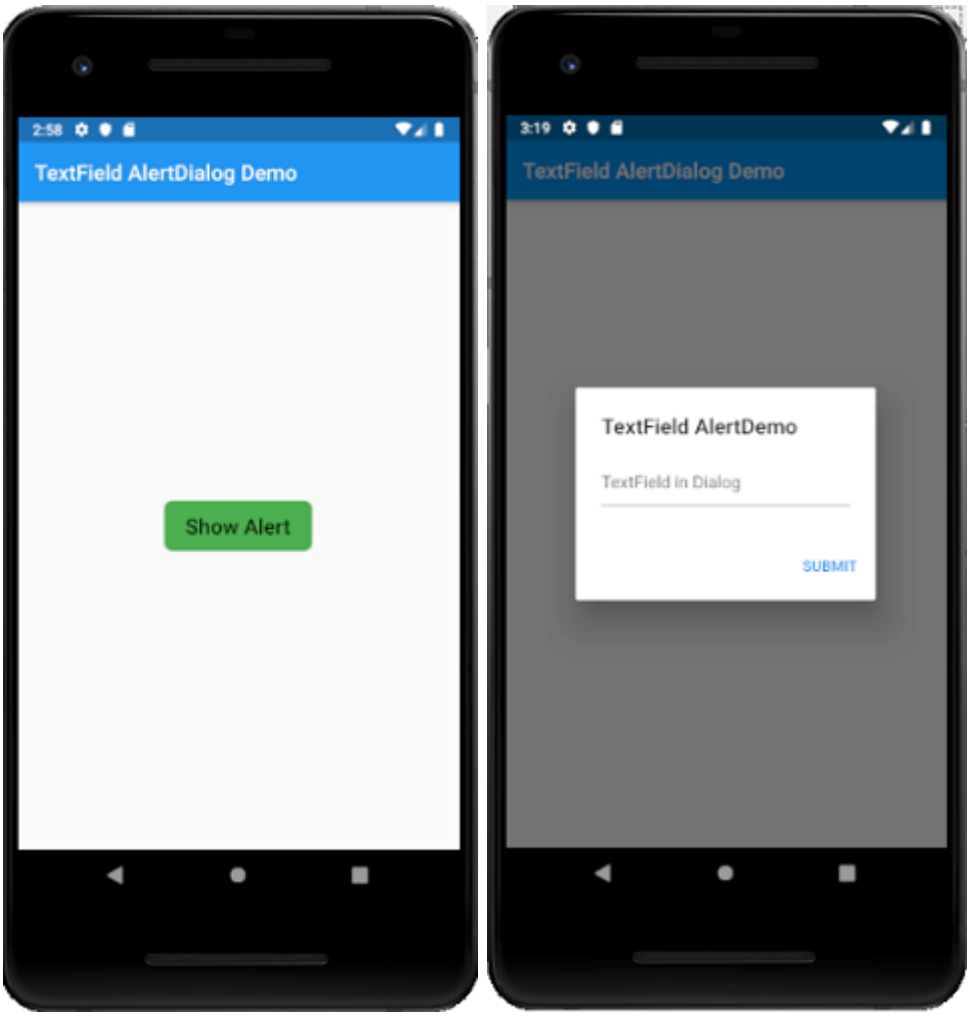
```
1. import 'package:flutter/material.dart';
2.
3. void main() => runApp(MyApp());
4.
5. class MyApp extends StatelessWidget {
6.   // This widget is the root of your application.
7.   @override
8.   Widget build(BuildContext context) {
9.     return MaterialApp(
10.      title: 'Flutter Alert Demo',
11.      debugShowCheckedModeBanner: false,
12.      theme: ThemeData(
13.        primarySwatch: Colors.blue,
14.      ),
15.      //home: MyHomePage(title: 'Flutter Demo Home Page'),
16.      home: TextFieldAlertDialog(),
17.    );
18.  }
19.}
20. class TextFieldAlertDialog extends StatelessWidget {
21.  TextEditingController _textFieldController = TextEditingController();
22.
23.  _displayDialog(BuildContext context) async {
24.    return showDialog(
25.       context: context,
26.       builder: (context) {
27.         return AlertDialog(
28.          title: Text('TextField AlertDemo'),
29.           content: TextField(
```

```
            controller: _textFieldController,
             decoration: InputDecoration(hintText: "TextField in Dialog"),
           ),
           actions: <Widget>[
            new FlatButton(
              child: new Text('SUBMIT'),
             onPressed: () {
                Navigator.of(context).pop();
             },
            )
           ],
          );
       });
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('TextField AlertDialog Demo'),
    ),
    body: Center(
      child: FlatButton(
        child: Text(
         'Show Alert',
          style: TextStyle(fontSize: 20.0),),
         padding: EdgeInsets.fromLTRB(20.0,12.0,20.0,12.0),
          shape: RoundedRectangleBorder(
           borderRadius: BorderRadius.circular(8.0)
          ),
        color: Colors.green,
         onPressed: () => _displayDialog(context),
      ),
    ),
```

```
64.     );
65.   }
66. }
```

Output

Now, run the app, it will give the following output. When you click on the button **Show Alert,** you will get the text input alert message.



Confirmation AlertDialog

The confirmation alert dialog notifies a user to confirm a particular choice before moving forward in the application. For example, when a user wants to delete or remove a contact from the address book.

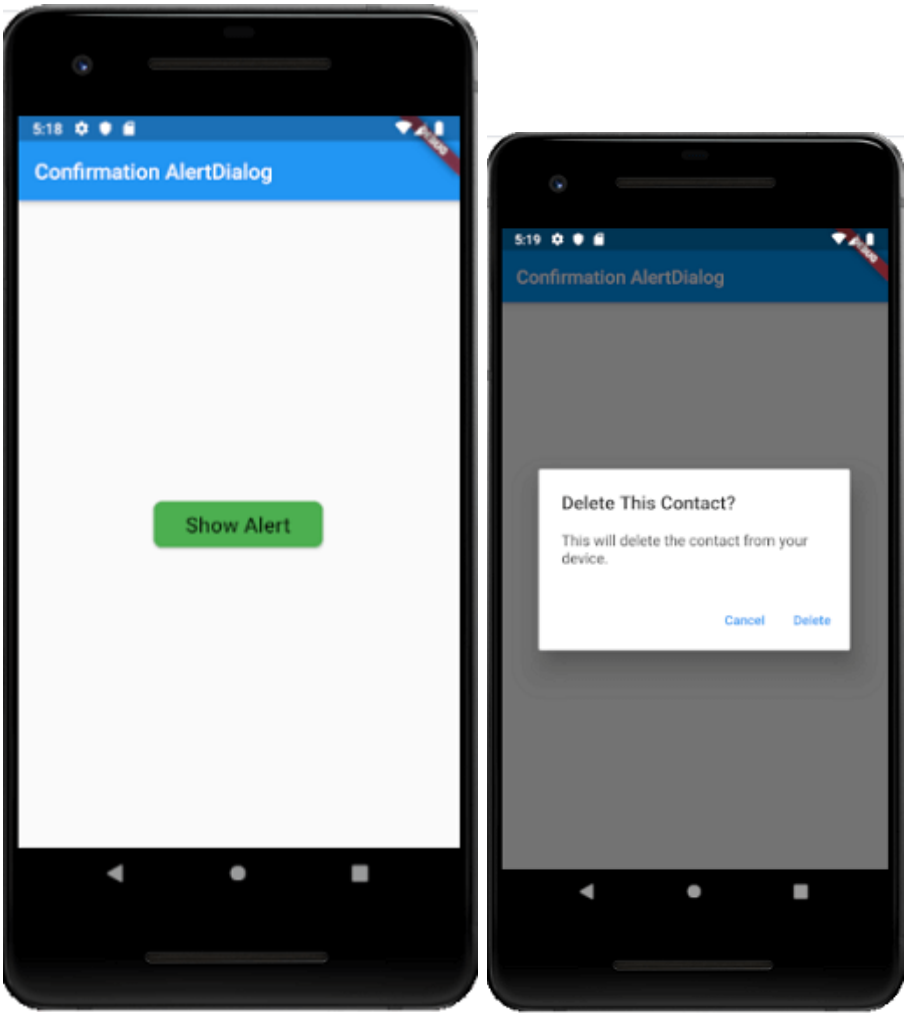
Example

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

void main() {
  runApp(new MaterialApp(home: new MyApp()));
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    // TODO: implement build
    return new Scaffold(
      appBar: AppBar(
        title: Text("Confirmation AlertDialog"),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            new RaisedButton(
              onPressed: () async {
                final ConfirmAction action = await _asyncConfirmDialog(context);
                print("Confirm Action $action" );
              },
              child: const Text(
                "Show Alert",
                style: TextStyle(fontSize: 20.0),),
              padding: EdgeInsets.fromLTRB(30.0,10.0,30.0,10.0),
                shape: RoundedRectangleBorder(
                  borderRadius: BorderRadius.circular(8.0)
                ),
              color: Colors.green,
              ),
          ],
```

```
35.         ),
36.       ),
37.     );
38.   }
39. }
40. enum ConfirmAction { Cancel, Accept}
41. Future<ConfirmAction> _asyncConfirmDialog(BuildContext context) async {
42.   return showDialog<ConfirmAction>(
43.     context: context,
44.     barrierDismissible: false, // user must tap button for close dialog!
45.     builder: (BuildContext context) {
46.       return AlertDialog(
47.         title: Text('Delete This Contact?'),
48.         content: const Text(
49.             'This will delete the contact from your device.'),
50.         actions: <Widget>[
51.           FlatButton(
52.             child: const Text('Cancel'),
53.             onPressed: () {
54.               Navigator.of(context).pop(ConfirmAction.Cancel);
55.             },
56.           ),
57.           FlatButton(
58.             child: const Text('Delete'),
59.             onPressed: () {
60.               Navigator.of(context).pop(ConfirmAction.Accept);
61.             },
62.           )
63.         ],
64.       );
65.     },
66.   );
67. }
```

Output

When you run the app, it will give the following output. Now, click on the button Show Alert, you will get the confirmation alert box message.



Select Option AlertDialog

This type of alert dialog displays the list of items, which takes immediate action when selected.

Example

```
1. import 'package:flutter/material.dart';
2.
3. void main() {
```

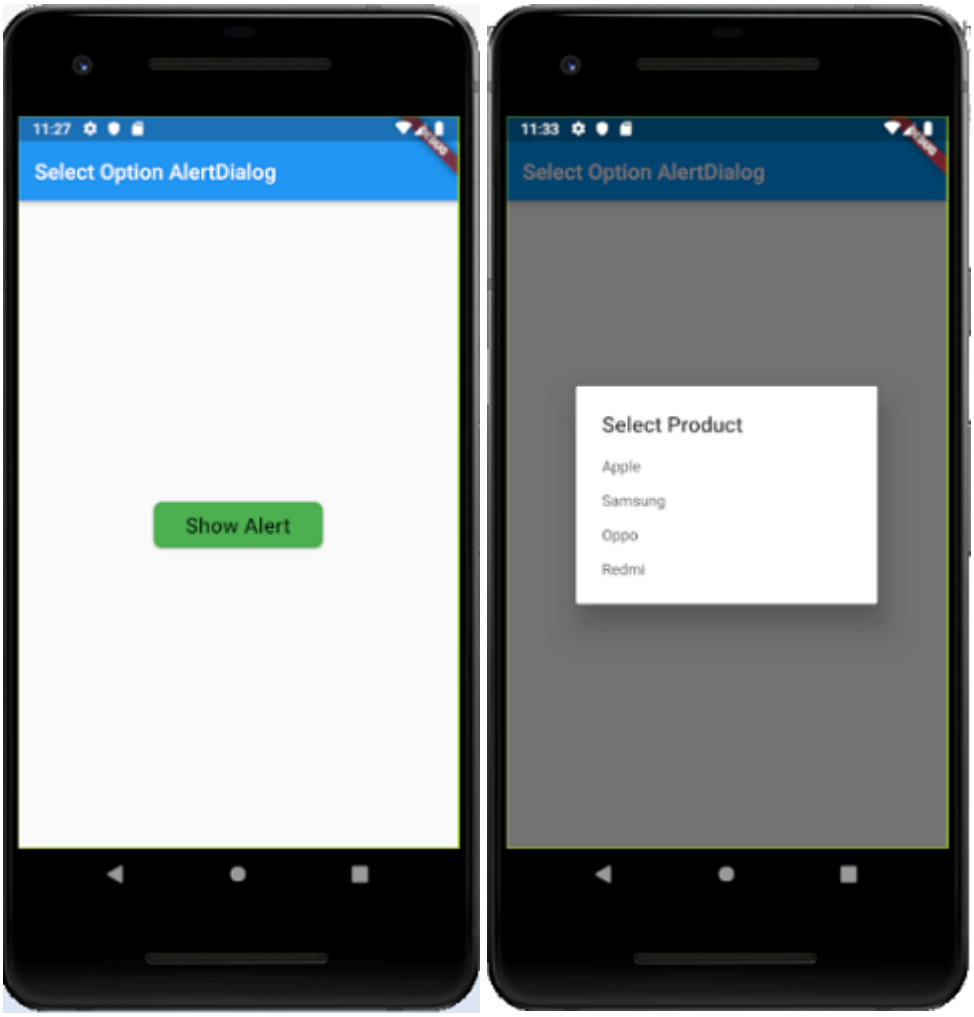
```
4.    runApp(new MaterialApp(home: new MyApp()));
5.  }
6.
7.  class MyApp extends StatelessWidget {
8.    // This widget is the root of your application.
9.    @override
10.   Widget build(BuildContext context) {
11.     // TODO: implement build
12.     return new Scaffold(
13.       appBar: AppBar(
14.         title: Text("Select Option AlertDialog"),
15.       ),
16.       body: Center(
17.         child: Column(
18.           mainAxisAlignment: MainAxisAlignment.center,
19.           children: <Widget>[
20.             new RaisedButton(
21.               onPressed: () async {
22.                 final Product prodName = await _asyncSimpleDialog(context);
23.                 print("Selected Product is $prodName");
24.               },
25.               child: const Text(
26.                 "Show Alert",
27.                 style: TextStyle(fontSize: 20.0),),
28.               padding: EdgeInsets.fromLTRB(30.0,10.0,30.0,10.0),
29.                 shape: RoundedRectangleBorder(
30.                   borderRadius: BorderRadius.circular(8.0)
31.                 ),
32.               color: Colors.green,
33.             ),
34.           ],
35.         ),
36.       ),
37.     );
```

```
38.   }
39. }
40. enum Product { Apple, Samsung, Oppo, Redmi }
41.
42. Future<Product> _asyncSimpleDialog(BuildContext context) async {
43.   return await showDialog<Product>(
44.     context: context,
45.     barrierDismissible: true,
46.     builder: (BuildContext context) {
47.       return SimpleDialog(
48.         title: const Text('Select Product '),
49.         children: <Widget>[
50.           SimpleDialogOption(
51.             onPressed: () {
52.               Navigator.pop(context, Product.Apple);
53.             },
54.             child: const Text('Apple'),
55.           ),
56.           SimpleDialogOption(
57.             onPressed: () {
58.               Navigator.pop(context, Product.Samsung);
59.             },
60.             child: const Text('Samsung'),
61.           ),
62.           SimpleDialogOption(
63.             onPressed: () {
64.               Navigator.pop(context, Product.Oppo);
65.             },
66.             child: const Text('Oppo'),
67.           ),
68.           SimpleDialogOption(
69.             onPressed: () {
70.               Navigator.pop(context, Product.Redmi);
71.             },
```

```
72.                 child: const Text('Redmi'),
73.               ),
74.             ],
75.           );
76.       });
77. }
```

Output

When you run the app, it will give the following output. Now, click on the button Show Alert, you will get the select option alert box message. As soon as you select any of the available options, the alert message disappears, and you will get a message of the selected choice in the console.



Flutter Tooltip

A tooltip is a material design class in Flutter that **provides text labels to explain the functionality** of a button or user interface action. In other words, it is used to show additional information when the user moves or points over a particular widget. It increases the accessibility of our application. If we wrap the widget with it, then it is very useful when the user long presses the widget because, in that case, it appears as a floating label.

Properties:

The following are the properties used to customize the application.

message: It is a string message used to display in the tooltip.

height: It is used to specify the height of the tooltip's child.

textStyle: It is used to determine the style for the message of the tooltip.

margin: It is used to determine the empty space surrounds the tooltip.

showDuration: It is used to specify the length of time for showing the tooltip after a long press is released. By default, it is 1.5 seconds.

decoration: It is used to define the shape and background color of the tooltip. The default tooltip shape is a rounded rectangle that has a border-radius of 4.0 PX.

verticalOffset: It determines the vertical gap between the tooltip and the widget.

waitDuration: It is used to specify the time when a pointer hovers over a tooltip's widget before showing the tooltip. When the pointer leaves the widget, the tooltip message will be disappeared.

padding: It determines the space to inset the tooltip's child. By default, it is 16.0 PX in all directions.

preferBelow: It is used to specify whether the tooltip is being displayed below the widget or not. By default, it is true. The tooltip will be displayed in the opposite direction if we have not sufficient space to display the tooltip in the preferred direction.

How to use Tooltip Widget in Flutter?

We can use the tooltip in Flutter as the below code:

```
Tooltip(
        message: 'User Account',
         child: IconButton(
        icon: Icon(Icons.high_quality),
         onPressed: () {
        /* your code */
         },
    ),
),
```

Output

Example

Let us understand it with the help of an example where we are trying to cover most of the above properties. In the following example, we are going to use a **FlatButton** with **Icon** as a child and surround this button with a **tooltip**. If we **long press** on this button, it will display a label with the message provided for the tooltip widget.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: MyHomePage()
    );
  }
}

class MyHomePage extends StatefulWidget {
```

```
15.  @override
16.  _MyHomePageState createState() => _MyHomePageState();
17. }
18.
19. class _MyHomePageState extends State<MyHomePage> {
20.  @override
21.  Widget build(BuildContext context) {
22.   return Scaffold(
23.    appBar: AppBar(
24.     title: Text("Flutter Tooltip Example"),
25.    ),
26.    body: Row(
27.        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
28.       children:<Widget>[
29.         Container(
30.          margin: EdgeInsets.all(10),
31.           child: Tooltip(
32.             waitDuration: Duration(seconds: 1),
33.              showDuration: Duration(seconds: 2),
34.             padding: EdgeInsets.all(5),
35.              height: 35,
36.             textStyle: TextStyle(
37.                 fontSize: 15, color: Colors.white, fontWeight: FontWeight.normal),
38.             decoration: BoxDecoration(
39.                 borderRadius: BorderRadius.circular(10), color: Colors.green),
40.             message: 'My Account',
41.              child: FlatButton(
42.               child: Icon(
43.                 Icons.account_box,
44.                size: 100,
45.                ),
46.             )),
47.         ),
48.         Container(
```

```
49.              margin: EdgeInsets.all(10),
50.            child: Tooltip(
51.                message: 'My Account',
52.               child: FlatButton(
53.                  child: Icon(
54.                  Icons.account_box,
55.                    size: 100,
56.                ),
57.                )
58.            ),
59.           )
60.         ]
61.      ),
62.    );
63.  }
64. }
```

Output:

When we run the app in our IDE, we will see the UI as below screenshot:

If we long-press the icon, we will see the tooltip as below screenshot.

Flutter Toast Notification

Flutter Toast is also called a **Toast Notification** message. It is a very small message which mainly popup at the bottom of the device screen. It will disappear on its own after completing the time provided by the developers. A developer mostly used the toast notification for showing feedback on the operation performed by the user.

Showing toast notification message is an essential feature in android applications. We can achieve it by using simple lines of code. In this section, we are going to learn how to show toast message in android and iOS by implementing it in Flutter. To implement toast notification, we need to import **fluttertoast** library in Flutter.

The following steps are needed to show toast notification in Flutter:

- Create a Flutter Project
- Add the Flutter Toast Dependencies in project
- Import the fluttertoast dart package in library
- Implement the code for showing toast message in Flutter

Flutter provides several properties to the user for showing the toast message, which is given below:

Property	Description
msg	String(Required)
toastlength	Toast.LENGTH_SHORT or Toast.LENGTH_LONG
gravity	ToastGravity.TOP or ToastGravity.CENTER or ToastGravity.BOTTOM
timeInSecForIos	It is used only for Ios (1 sec or more)
backgroundColor	It specifies the background color.

textColor	It specifies text color.
fontSize	It specifies the font size of the notification message.

FlutterToast.cancel(): This function is used when you want to cancel all the requests to show message to the user.

Let us see how we can show toast notification in the Flutter app through the following steps:

Step 1: Create a Flutter project in the IDE. Here, I am going to use Android Studio.

Step 2: Open the project in Android Studio and navigate to the **lib** folder. In this folder, open the **pubspec.yaml** file. Here, we need to add the flutter toast library in the dependency section and then click on **get package** link to import the library in your **main.dart** file.

pubspec.yaml

```
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^0.1.2
  fluttertoast: ^3.1.0
```

It ensures that while adding the dependencies, you have **left two spaces** from the left side of a fluttertoast dependency. The fluttertoast dependency provides the capability to show toast notification in a simple way. It can also customize the look of the toast popup very easily.

Step 3: Open the **main.dart** file and create a toast notification in the widget as the code given below.

```
Fluttertoast.showToast(
        msg: 'This is toast notification',
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.BOTTOM,
        timeInSecForIos: 1,
```

```
6.          backgroundColor: Colors.red,
7.          textColor: Colors.yellow
8.      );
```

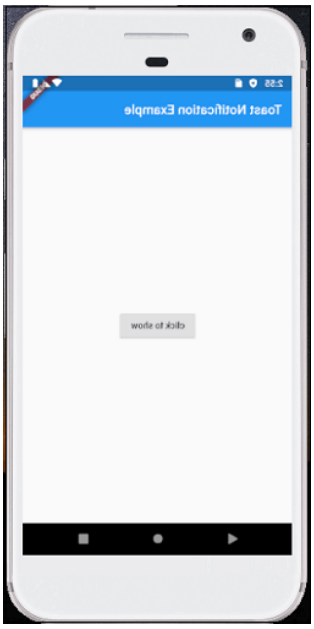
Let us see the complete code of the above steps. Open the main.dart file and replace the following code. This code contains a button, and when we pressed on this, it will display the toast message by calling **FlutterToast.showToast**.

```
1.  import 'package:flutter/material.dart';
2.  import 'package:fluttertoast/fluttertoast.dart';
3.
4.  class ToastExample extends StatefulWidget {
5.    @override
6.    _ToastExampleState createState() {
7.      return _ToastExampleState();
8.    }
9.  }
10.
11. class _ToastExampleState extends State {
12.   void showToast() {
13.     Fluttertoast.showToast(
14.         msg: 'This is toast notification',
15.         toastLength: Toast.LENGTH_SHORT,
16.         gravity: ToastGravity.BOTTOM,
17.         timeInSecForIos: 1,
18.         backgroundColor: Colors.red,
19.         textColor: Colors.yellow
20.     );
21.   }
22.
23.   @override
24.   Widget build(BuildContext context) {
25.     return MaterialApp(
26.       title: 'Toast Notification Example',
27.       home: Scaffold(
```

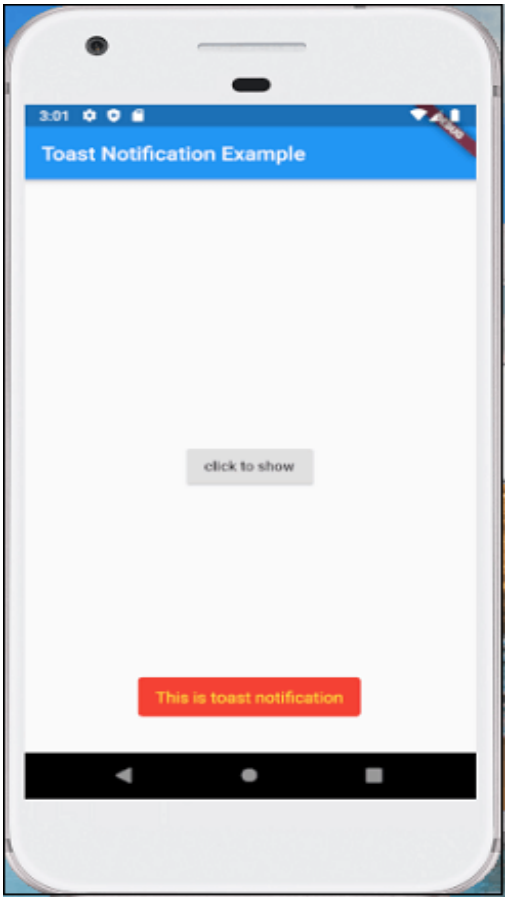
```
28.          appBar: AppBar(
29.            title: Text('Toast Notification Example'),
30.          ),
31.           body: Padding(
32.            padding: EdgeInsets.all(15.0),
33.             child: Center(
34.              child: RaisedButton(
35.                 child: Text('click to show'),
36.                onPressed: showToast,
37.               ),
38.            ),
39.          )
40.       ),
41.     );
42.   }
43. }
44.
45. void main() => runApp(ToastExample());
```

Output

Now, run the app in Android Studio. It will give the following screen.



When we click on the **"click to show"** button, we can see the toast message at the bottom of the screen. See the below image:



Flutter Switch

A switch is a two-state user interface element used to toggle between **ON (Checked) or OFF (Unchecked) states**. Typically, it is a button with a thumb slider where the user can drag back and forth to choose an option in the form of ON or OFF. Its working is similar to the house electricity switches.

In Flutter, the switch is a widget used to select between two options, either ON or OFF. It does not maintain the state itself. To maintain the states, it will call the **onChanged** property. If the value return by this property is **true**, then the switch is ON and false when it is OFF. When this property is null, the switch widget is disabled. In this article, we are going to understand how to use a switch widget in the Flutter application.

Properties of Switch Widget

Some of the essential attributes of switch widget are given below:

Attributes	Descriptions
onChanged	It will be called whenever the user taps on the switch.
value	It contains a Boolean value true or false to control whether the switch functionality is O or OFF.
activeColor	It is used to specify the color of the switch round ball when it is ON.
activeTrackColor	It specifies the switch track bar color.
inactiveThubmColor	It is used to specify the color of the switch round ball when it is OFF.
inactiveTrackColor	It specifies the switch track bar color when it is OFF.
dragStartBehavior	It handled the drag start behavior. If we set it as DragStartBehavior.start, the drag move the switch from on to off.

Example

Keep Watching

In this application, we have defined a **switch widget**. Every time we toggled the switch widget, the onChanged property is called with a new state of the switch as value. To store the switch state, we have defined a **boolean variable isSwitched** that can be shown in the below code.

Open the IDE you are using, and create a Flutter application. Next, open the **lib** folder and replace **main.dart** with the following code.

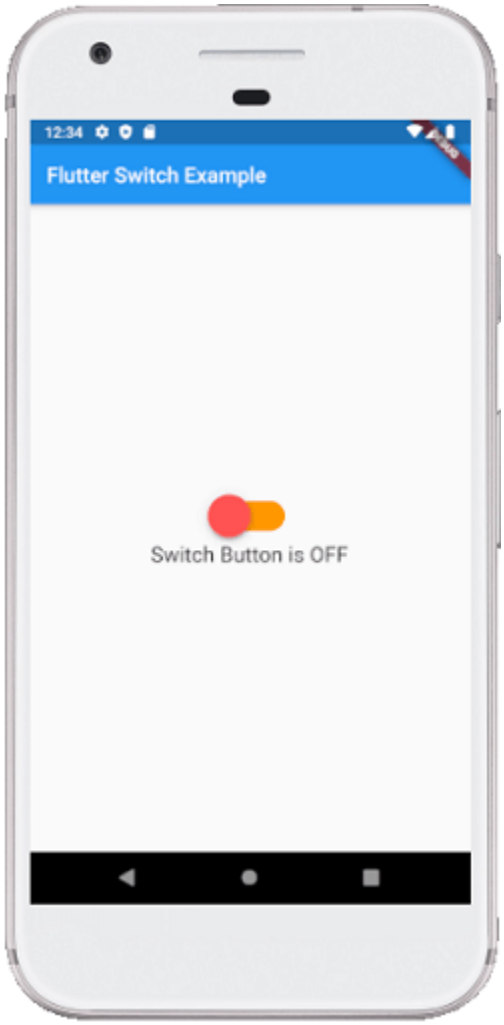
```
1. import 'package:flutter/material.dart';
2.
3. void main() => runApp(MyApp());
```

```
4.
5.  class MyApp extends StatelessWidget {
6.    @override
7.    Widget build(BuildContext context) {
8.      return MaterialApp(
9.          home: Scaffold(
10.           appBar: AppBar(
11.              backgroundColor: Colors.blue,
12.             title: Text("Flutter Switch Example"),
13.            ),
14.           body: Center(
15.                 child: SwitchScreen()
16.           ),
17.         )
18.    );
19.  }
20. }
21.
22. class SwitchScreen extends StatefulWidget {
23.   @override
24.   SwitchClass createState() => new SwitchClass();
25. }
26.
27. class SwitchClass extends State {
28.   bool isSwitched = false;
29.   var textValue = 'Switch is OFF';
30.
31.   void toggleSwitch(bool value) {
32.
33.    if(isSwitched == false)
34.    {
35.      setState(() {
36.       isSwitched = true;
37.        textValue = 'Switch Button is ON';
```

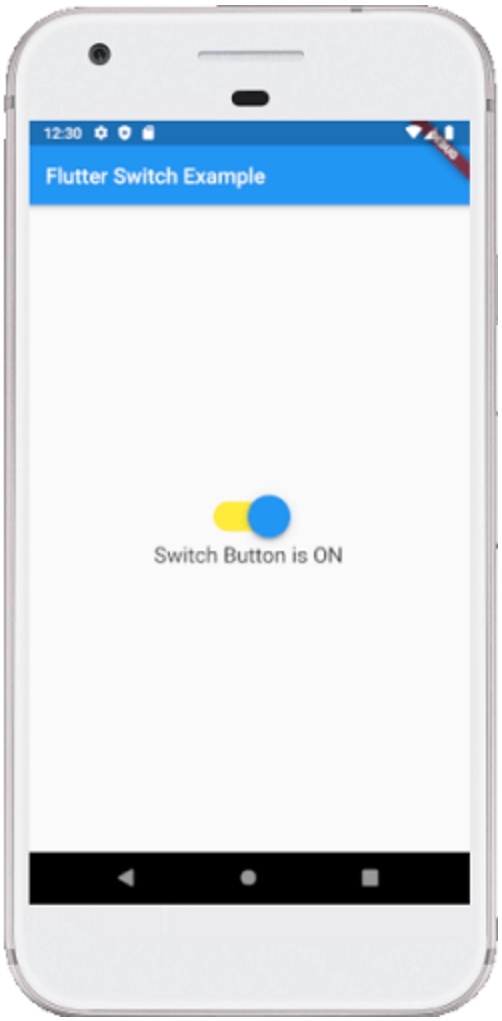
```
38.      });
39.      print('Switch Button is ON');
40.    }
41.    else
42.    {
43.      setState(() {
44.        isSwitched = false;
45.        textValue = 'Switch Button is OFF';
46.      });
47.      print('Switch Button is OFF');
48.    }
49.  }
50.  @override
51.  Widget build(BuildContext context) {
52.    return Column(
53.       mainAxisAlignment: MainAxisAlignment.center,
54.       children:[ Transform.scale(
55.            scale: 2,
56.           child: Switch(
57.              onChanged: toggleSwitch,
58.             value: isSwitched,
59.              activeColor: Colors.blue,
60.             activeTrackColor: Colors.yellow,
61.              inactiveThumbColor: Colors.redAccent,
62.             inactiveTrackColor: Colors.orange,
63.            )
64.         ),
65.          Text('$textValue', style: TextStyle(fontSize: 20),)
66.       ]);
67.  }
68. }
```

Output:

When we run the application in the emulator or device, we should get UI similar to the following screenshot



If we press on the switch, it will change their state from OFF to ON. See the below screenshot:



How to customize the Switch button in Flutter?

Flutter also allows the user to customize their switch button. Customization makes the user interface more interactive. We can do this by adding the **custom-switch dependency** in the **pubspec.yaml** file and then import it into the dart file.

Example:

Open the **main.dart** file and replace it with the following code:

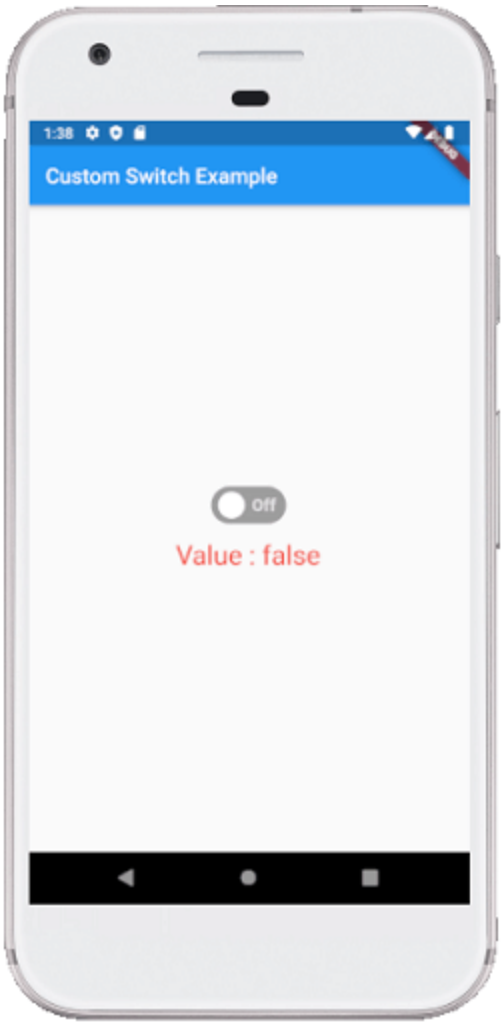
```
import 'package:flutter/material.dart';
import 'package:custom_switch/custom_switch.dart';

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.           backgroundColor: Colors.blue,
13.            title: Text("Custom Switch Example"),
14.          ),
15.          body: Center(
16.              child: SwitchScreen()
17.          ),
18.      )
19.   );
20.  }
21. }
22.
23. class SwitchScreen extends StatefulWidget {
24.  @override
25.  SwitchClass createState() => new SwitchClass();
26. }
27.
28. class SwitchClass extends State {
29.  bool isSwitched = false;
30.  @override
31.  Widget build(BuildContext context) {
32.   return Column(
33.     mainAxisAlignment: MainAxisAlignment.center,
34.     children:<Widget>[
35.          CustomSwitch(
36.           value: isSwitched,
37.            activeColor: Colors.blue,
38.           onChanged: (value) {
39.              print("VALUE : $value");
```

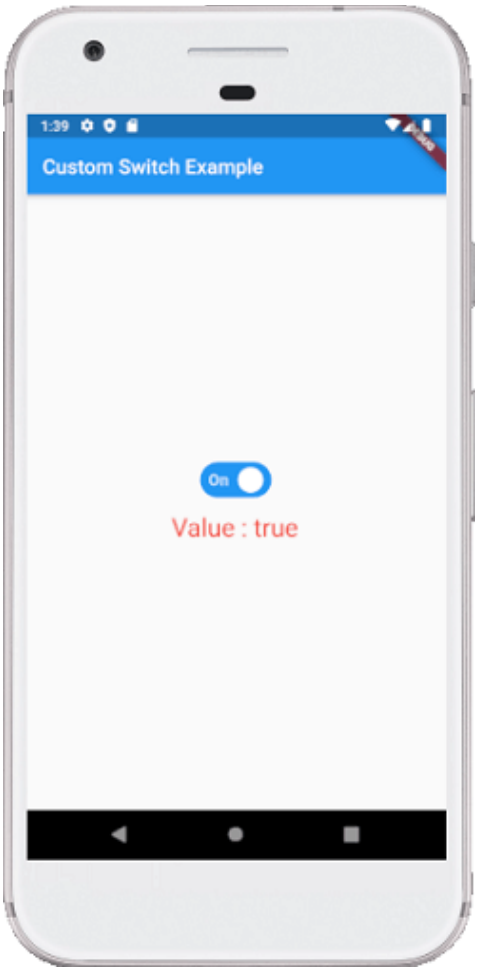
```
40.                 setState(() {
41.                     isSwitched = value;
42.                 });
43.                 },
44.             ),
45.           SizedBox(height: 15.0,),
46.           Text('Value : $isSwitched', style: TextStyle(color: Colors.red,
47.               fontSize: 25.0),)
48.         ]);
49.     }
50. }
```

Output:

When we run the application in the emulator or device, we should get UI similar to the following screenshot:



If we press on the switch, it will change their state from OFF to ON. See the below screenshot:



Flutter Charts

A chart is a graphical representation of data where data is represented by a **symbol** such as a line, bar, pie, etc. In Flutter, the chart behaves the same as a normal chart. We use a chart in Flutter to **represent the data in a graphical way** that allows the user to understand them in a simple manner. We can also plot a graph to represents the rise and fall of our values. The chart can easily read the data and helps us to know the performance on a monthly or yearly basis whenever we need it.

Supported Chart Types in Flutter

Flutter supports mainly three types of charts, and each chart comes with several configuration options. The following are the chart used in Flutter application:

1. Line Chart
2. Bar Chart

3. Pie and Donut Chart

Line Chart

A line chart is a graph that uses lines for connecting individual data points. It displays the information in a series of data points. It is mainly used to track changes over a short and long period of time.

We can use it as below:

```
LineChart(
  LineChartData(
    // write your logic
  ),
);
```

Bar Chart

It is a graph that represents the categorical data with rectangular bars. It can be horizontal or vertical.

We can use it as below:

```
BarChart(
  BarChartData(
    // write your logic
  ),
);
```

Pie or Donut Chart

It is a graph that displays the information in a circular graph. In this graph, the circle is divided into sectors, and each shows the percentage or proportional data.

We can use it as below:

```
PieChart(
  PieChartData(
    // write your logic
  ),
```

```
);
```

Let us understand it with the help of an example.

Example

Open the IDE and create the new Flutter project. Next, open the project, navigate to the lib folder, and open the **pubspec.yaml** file. In this file, we need to add the chart dependency. Flutter provides several chart dependency, and here we are going to use **fl_chart dependency**. So add it as below:

```
dependencies:
  flutter:
    sdk: flutter
  fl_chart: ^0.10.1
```

After adding the dependency, click on the **get packages** link shown on the screen's top left corner. Now, open the **main.dart** file and replace it with the below code:

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

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

/// This Widget is the main application widget.
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomePage(),
    );
  }
}

class HomePage extends StatelessWidget {
 @override
  Widget build(BuildContext context) {
```

```
    return Scaffold(
     appBar: AppBar(
       title: const Text('Flutter Chart Example'),
        backgroundColor: Colors.green
     ),
     body: Center(
       child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
          LineCharts(),
           Padding(
            padding: const EdgeInsets.all(16.0),
             child: Text(
              "Traffic Source Chart",
                style: TextStyle(
                  fontSize: 20,
                    color: Colors.purple,
                  fontWeight: FontWeight.w700,
                    fontStyle: FontStyle.italic
               )
             )
          ),
         ],
       ),
     ),
    );
  }
}

class LineCharts extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    const cutOffYValue = 0.0;
    const yearTextStyle =
```

```
53.    TextStyle(fontSize: 12, color: Colors.black);
54.
55.    return SizedBox(
56.     width: 360,
57.     height: 250,
58.     child: LineChart(
59.       LineChartData(
60.        lineTouchData: LineTouchData(enabled: false),
61.         lineBarsData: [
62.          LineChartBarData(
63.            spots: [
64.             FlSpot(0, 1),
65.              FlSpot(1, 1),
66.             FlSpot(2, 3),
67.              FlSpot(3, 4),
68.             FlSpot(3, 5),
69.              FlSpot(4, 4)
70.            ],
71.             isCurved: true,
72.            barWidth: 2,
73.             colors: [
74.              Colors.black,
75.             ],
76.            belowBarData: BarAreaData(
77.              show: true,
78.             colors: [Colors.lightBlue.withOpacity(0.5)],
79.              cutOffY: cutOffYValue,
80.             applyCutOffY: true,
81.             ),
82.            aboveBarData: BarAreaData(
83.              show: true,
84.             colors: [Colors.lightGreen.withOpacity(0.5)],
85.              cutOffY: cutOffYValue,
86.             applyCutOffY: true,
```

```
87.              ),
88.            dotData: FlDotData(
89.              show: false,
90.            ),
91.           ),
92.         ],
93.          minY: 0,
94.         titlesData: FlTitlesData(
95.           bottomTitles: SideTitles(
96.             showTitles: true,
97.              reservedSize: 5,
98.             textStyle: yearTextStyle,
99.              getTitles: (value) {
100.                       switch (value.toInt()) {
101.                         case 0:
102.                         return '2016';
103.                         case 1:
104.                         return '2017';
105.                         case 2:
106.                         return '2018';
107.                         case 3:
108.                         return '2019';
109.                         case 4:
110.                         return '2020';
111.                         default:
112.                         return '';
113.                       }
114.                     }),
115.                   leftTitles: SideTitles(
116.                    showTitles: true,
117.                     getTitles: (value) {
118.                      return '\$ ${value + 100}';
119.                     },
120.                   ),
```

```
            ),
            axisTitleData: FlAxisTitleData(
                leftTitle: AxisTitle(showTitle: true, titleText: 'Value', margin: 10),
                bottomTitle: AxisTitle(
                    showTitle: true,
                    margin: 10,
                    titleText: 'Year',
                    textStyle: yearTextStyle,
                    textAlign: TextAlign.right)),
            gridData: FlGridData(
              show: true,
              checkToShowHorizontalLine: (double value) {
                return value == 1 || value == 2 || value == 3 || value == 4;
              },
            ),
          ),
        ),
      );
    }
  }
```

Output:

When we run the app in the device or emulator, we will get the UI of the screen similar to the below screenshot:

Flutter Forms

Forms are an integral part of all modern mobile and web applications. It is mainly used to interact with the app as well as gather information from the users. They can perform many tasks, which depend on the nature of your business requirements and logic, such as authentication of the user, adding user, searching, filtering, ordering, booking, etc. A form can contain text fields, buttons, checkboxes, radio buttons, etc.

Creating Form

Flutter provides a **Form widget** to create a form. The form widget acts as a container, which allows us to group and validate the multiple form fields. When you create a form, it is necessary to provide the **GlobalKey**. This key uniquely identifies the form and allows you to do any validation in the form fields.

The form widget uses child widget **TextFormField** to provide the users to enter the text field. This widget renders a material design text field and also allows us to display validation errors when they occur.

Let us create a form. First, create a Flutter project and replace the following code in the main.dart file. In this code snippet, we have created a custom class named **MyCustomForm**. Inside this class, we define a global key as **_formKey**. This key holds a **FormState** and can use to retrieve the form widget. Inside the **build** method of this class, we have added some custom style and use the TextFormField widget to provide the form fields such as name, phone number, date of birth, or just a normal field. Inside the TextFormField, we have used **InputDecoration** that provides the look and feel of your form properties such as borders, labels, icons, hint, styles, etc. Finally, we have added a **button** to submit the form.

```
1.  import 'package:flutter/material.dart';
2.
3.  void main() => runApp(MyApp());
4.
5.  class MyApp extends StatelessWidget {
6.    @override
7.    Widget build(BuildContext context) {
8.      final appTitle = 'Flutter Form Demo';
9.      return MaterialApp(
10.       title: appTitle,
11.       home: Scaffold(
12.         appBar: AppBar(
13.           title: Text(appTitle),
14.         ),
15.         body: MyCustomForm(),
16.       ),
17.     );
18.   }
19. }
20. // Create a Form widget.
21. class MyCustomForm extends StatefulWidget {
```

```
22.  @override
23.  MyCustomFormState createState() {
24.    return MyCustomFormState();
25.  }
26. }
27. // Create a corresponding State class. This class holds data related to the form.
28. class MyCustomFormState extends State<MyCustomForm> {
29.  // Create a global key that uniquely identifies the Form widget
30.  // and allows validation of the form.
31.  final _formKey = GlobalKey<FormState>();
32.
33.  @override
34.  Widget build(BuildContext context) {
35.    // Build a Form widget using the _formKey created above.
36.    return Form(
37.      key: _formKey,
38.      child: Column(
39.        crossAxisAlignment: CrossAxisAlignment.start,
40.        children: <Widget>[
41.          TextFormField(
42.            decoration: const InputDecoration(
43.              icon: const Icon(Icons.person),
44.              hintText: 'Enter your name',
45.              labelText: 'Name',
46.            ),
47.          ),
48.          TextFormField(
49.            decoration: const InputDecoration(
50.              icon: const Icon(Icons.phone),
51.              hintText: 'Enter a phone number',
52.              labelText: 'Phone',
53.            ),
54.          ),
55.          TextFormField(
```

```
            decoration: const InputDecoration(
            icon: const Icon(Icons.calendar_today),
            hintText: 'Enter your date of birth',
            labelText: 'Dob',
            ),
          ),
          new Container(
              padding: const EdgeInsets.only(left: 150.0, top: 40.0),
              child: new RaisedButton(
                child: const Text('Submit'),
                  onPressed: null,
              )),
        ],
      ),
    );
  }
}
```

Output

Now, run the app, you can see the following screen in your Android Emulator. This form contains three field name, phone number, date of birth, and submit button.

Form validation

Validation is a method, which allows us to correct or confirms a certain standard. It ensures the authentication of the entered data.

Validating forms is a common practice in all digital interactions. To validate a form in a flutter, we need to implement mainly three steps.

Step 1: Use the Form widget with a global key.

Step 2: Use TextFormField to give the input field with validator property.

Step 3: Create a button to validate form fields and display validation errors.

Let us understand it with the following example. In the above code, we have to use **validator()** function in the TextFormField to validate the input properties. If the user gives the wrong input, the validator function returns a string that contains an **error**

message; otherwise, the validator function return **null**. In the validator function, make sure that the TextFormField is not empty. Otherwise, it returns an error message.

The validator() function can be written as below code snippets:

```
validator: (value) {
      if (value.isEmpty) {
          return 'Please enter some text';
      }
      return null;
},
```

Now, open the **main.dart** file and add validator() function in the TextFormField widget. Replace the following code with the main.dart file.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final appTitle = 'Flutter Form Demo';
    return MaterialApp(
      title: appTitle,
      home: Scaffold(
        appBar: AppBar(
          title: Text(appTitle),
        ),
        body: MyCustomForm(),
      ),
    );
  }
}
// Create a Form widget.
class MyCustomForm extends StatefulWidget {
```

```
22.  @override
23.  MyCustomFormState createState() {
24.    return MyCustomFormState();
25.  }
26. }
27. // Create a corresponding State class, which holds data related to the form.
28. class MyCustomFormState extends State<MyCustomForm> {
29.  // Create a global key that uniquely identifies the Form widget
30.  // and allows validation of the form.
31.  final _formKey = GlobalKey<FormState>();
32.
33.  @override
34.  Widget build(BuildContext context) {
35.    // Build a Form widget using the _formKey created above.
36.    return Form(
37.      key: _formKey,
38.      child: Column(
39.        crossAxisAlignment: CrossAxisAlignment.start,
40.        children: <Widget>[
41.          TextFormField(
42.            decoration: const InputDecoration(
43.              icon: const Icon(Icons.person),
44.              hintText: 'Enter your full name',
45.              labelText: 'Name',
46.            ),
47.            validator: (value) {
48.              if (value.isEmpty) {
49.                return 'Please enter some text';
50.              }
51.              return null;
52.            },
53.          ),
54.          TextFormField(
55.            decoration: const InputDecoration(
```

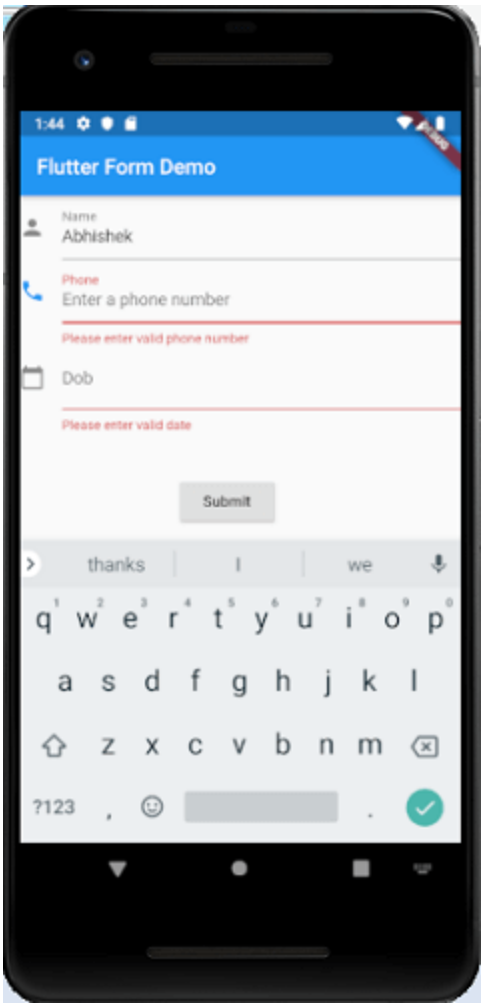
```
56.            icon: const Icon(Icons.phone),
57.            hintText: 'Enter a phone number',
58.            labelText: 'Phone',
59.          ),
60.          validator: (value) {
61.            if (value.isEmpty) {
62.              return 'Please enter valid phone number';
63.            }
64.            return null;
65.          },
66.        ),
67.        TextFormField(
68.          decoration: const InputDecoration(
69.          icon: const Icon(Icons.calendar_today),
70.          hintText: 'Enter your date of birth',
71.          labelText: 'Dob',
72.          ),
73.          validator: (value) {
74.            if (value.isEmpty) {
75.              return 'Please enter valid date';
76.            }
77.            return null;
78.          },
79.         ),
80.        new Container(
81.            padding: const EdgeInsets.only(left: 150.0, top: 40.0),
82.            child: new RaisedButton(
83.              child: const Text('Submit'),
84.              onPressed: () {
85.                // It returns true if the form is valid, otherwise returns false
86.                if (_formKey.currentState.validate()) {
87.                  // If the form is valid, display a Snackbar.
88.                  Scaffold.of(context)
89.                      .showSnackBar(SnackBar(content: Text('Data is in processing.')));
```

```
90.                     }
91.                   },
92.                 )),
93.           ],
94.         ),
95.     );
96.   }
97. }
```

Output

Now, run the app. The following screen appears.

In this form, if you left any input field blank, you will get an error message like below screen.


1:44
Flutter Form Demo
Name
Abhishek
Phone
Enter a phone number
Please enter valid phone number
Dob
Please enter valid date
Submit
thanks
I
we
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123