UNIT-4

Introduction to gestures and state management:

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

Tap: It means touching the surface of the screen from the fingertip for a short time and then releasing them. This gesture contains the following events:

- onTapDown
- onTapUp
- onTap
- onTapCancel

Double Tap: It is similar to a Tap gesture, but you need to tapping twice in a short time. This gesture contains the following events:

- onDoubleTap

Long Press: It means touching the surface of the screen at a particular location for a long time. This gesture contains the following events:

- onLongPress

Drag: It allows us to touch the surface of the screen with a fingertip and move it from one location to another location and then releasing them. Flutter categories the drag into two types:

Horizontal Drag: This gesture allows the pointer to move in a horizontal direction. It contains the following events:

- onHorizontalDragStart
- onHorizontalDragUpdate
- onHorizontalDragEnd

Vertical Drag: This gesture allows the pointer to move in a vertical direction. It contains the following events:

- onVerticalDragStart
- onVerticalDragStart
- onVerticalDragStart

Pan: It means touching the surface of the screen with a fingertip, which can move in any direction without releasing the fingertip. This gesture contains the following events:

- onPanStart
- onPanUpdate
- onPanEnd

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

Types of dialogs in a flutter

- AlertDialog
- SimpleDialog
- showDialog

AlertDialog

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

Properties:

- **Title:** It is always recommended to make our dialog title as short as possible. It will be easily understandable to the user.
- **Action:** It is used to show the content for what action has to perform.
- **Content:** The body of the **alertDialog** widget is defined by the content.
- **Shape:** It is used to define the shape of our dialog box whether it is circular, curved, and many more.

Here is the snippet code for creating a dialog box.

```
AlertDialog(
          title: Text('Welcome'),          // To display the title it is optional
          content: Text('GeeksforGeeks'),  // Message which will be pop up on the screen
                                  // Action widget which will provide the user to acknowledge
the choice
           actions: [
           FlatButton(                  // FlatButton widget is used to make a text to work like a
button
```

```
        textColor: Colors.black,
        onPressed: () {},            // function used to perform after pressing the button
        child: Text('CANCEL'),
      ),
      FlatButton(
        textColor: Colors.black,
        onPressed: () {},
        child: Text('ACCEPT'),
      ),
    ],
  ),
```

SimpleDialog

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

Properties:

- **Title:** It is always recommended to make our dialog title as short as possible. It will be easily understandable to the user.
- **Shape:** It is used to define the shape of our dialog box whether it is circular, curve, and many more.
- **backgroundcolor:** It is used to set the background color of our dialog box.
- **TextStyle:** It is used to change the style of our text.

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

),

showDialog

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

Properties:

- **Builder:** It returns the child instead of creating a child argument.

- **Barriercolor:** It defines the modal barrier color which darkens everything in the dialog.
- **useSafeArea:** It makes sure that the dialog uses the safe area of the screen only not overlapping the screen area.

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

Flutter State Management

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

What is State?

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

The state management is one of the most popular and necessary processes in the lifecycle of an application. According to official documentation, Flutter is declarative. It means Flutter builds its

UI by reflecting the current state of your app. The following figure explains it more clearly where you can build a UI from the application state.

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

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

- Ephemeral State
- App State

Ephemeral State

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

Example

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

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

  @override
  Widget build(BuildContext context) {
```

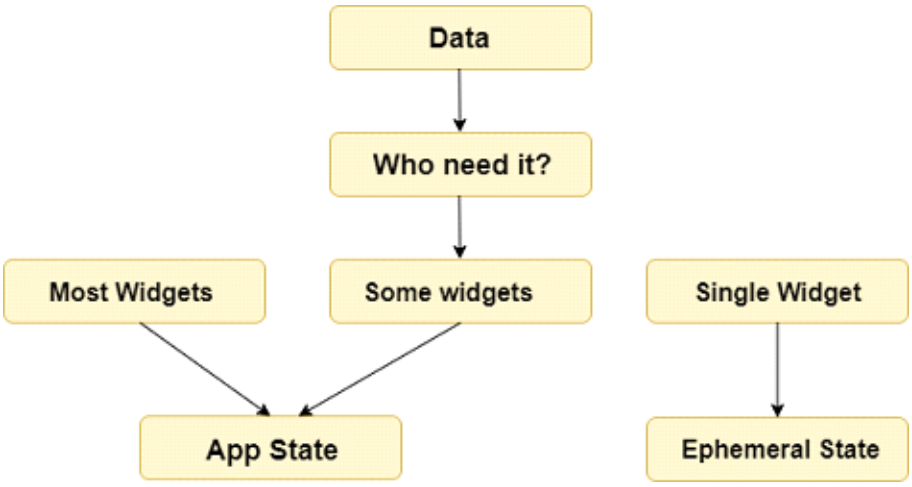
```
    return RaisedButton(
      child: Text(_name),
      onPressed: () {
        setState(() {
          _name = _name == "Peter" ? "John" : "Peter";
        });
      },
    );
  }
}
```

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

App State

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

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



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

- ChangeNotifier

- ChangeNotifierProvider
- Consumer

Flutter Navigation and Routing

Navigation and routing are some of the core concepts of all mobile application, which allows the user to move between different pages. We know that every mobile application contains several screens for displaying different types of information. **For example,** an app can have a screen that contains various products. When the user taps on that product, immediately it will display detailed information about that product.

In Flutter, the screens and pages are known as **routes,** and these routes are just a widget. In Android, a route is similar to an **Activity,** whereas, in iOS, it is equivalent to a **ViewController.**

In any mobile app, navigating to different pages defines the workflow of the application, and the way to handle the navigation is known as **routing.** Flutter provides a basic routing class **MaterialPageRoute** and two methods **Navigator.push()** and **Navigator.pop()** that shows how to navigate between two routes. The following steps are required to start navigation in your application.

Step 1: First, you need to create two routes.

Step 2: Then, navigate to one route from another route by using the Navigator.push() method.

Step 3: Finally, navigate to the first route by using the Navigator.pop() method.

Let us take a simple example to understand the navigation between two routes:

Create two routes

Here, we are going to create two routes for navigation. In both routes, we have created only a **single button.** When we tap the button on the first page, it will navigate to the second page. Again, when we tap the button on the second page, it will return to the first page. The below code snippet creates two routes in the Flutter application.

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

```
        ),
      ),
    );
  }
}

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

Navigate to the second route using Navigator.push() method

The Navigator.push() method is used to navigate/switch to a new route/page/screen. Here, the **push()** method adds a page/route on the stack and then manage it by using the **Navigator.** Again we use MaterialPageRoute class that allows transition between the routes using a platform-specific animation. The below code explain the use of the Navigator.push() method.

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

```
}
```

Return to the first route using Navigator.pop() method

Now, we need to use Navigator.pop() method to close the second route and return to the first route. The **pop()** method allows us to remove the current route from the stack, which is managed by the Navigator.

To implement a return to the original route, we need to update the **onPressed()** callback method in the SecondRoute widget as below code snippet:

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

Now, let us see the full code to implement the navigation between two routes. First, create a Flutter project and insert the following code in the **main.dart** file.

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

void main() {
runApp(MaterialApp(
   title: 'Flutter Navigation',
   theme: ThemeData(
     // This is the theme of your application.
     primarySwatch: Colors.green,
   ),
   home: FirstRoute(),
 ));
}

class FirstRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First Screen'),
      ),
      body: Center(
```

```
        child: RaisedButton(
          child: Text('Click Here'),
          color: Colors.orangeAccent,
          onPressed: () {
          Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => SecondRoute()),
            );
          },
        ),
      ),
    );
  }
}

class SecondRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Second Screen"),
      ),
      body: Center(
        child: RaisedButton(
          color: Colors.blueGrey,
          onPressed: () {
          Navigator.pop(context);
          },
          child: Text('Go back'),
        ),
      ),
    );
  }
}
```

Output

When you run the project in the **Android Studio,** you will get the following screen in your emulator. It is the first screen that contains only a single button.

Click the button **Click Here,** and you will navigate to a second screen as below image. Next, when you click on the button **Go Back,** you will return to the first page.

–

Statefulwidget_widgets

Statefulwidget provides an option for a widget to create a state, State (where T is the inherited widget) when the widget is created for the first time through createState method and then a method, setState to change the state whenever needed. The state change will be done through gestures.

input:

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:

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

Step 1: Create a Flutter project in the IDE you used. 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 import the material.dart package as given below:

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

Step 3: Next, call the main MyApp class using void main run app function and then create your main widget class named as MyApp extends with StatefulWidget:

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

```
class MyApp extends StatefulWidget { }
```

Step 4: Next, we need to create the Scaffold widget -> Column widget in the class widget build area as given below:

```
class MyApp extends StatefulWidget {
  @override
```

```
 Widget build(BuildContext context) {
   return Scaffold(
       appBar: AppBar(
         title: Text('Flutter TextField Example'),
       ),
       body: Padding(
           padding: EdgeInsets.all(15),
           child: Column(
             children: <Widget> [


             ]
             )
           )
             )
        );
  }
}
```

Step 5: Finally, create the TextField widget as the below code.

```
child: 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.

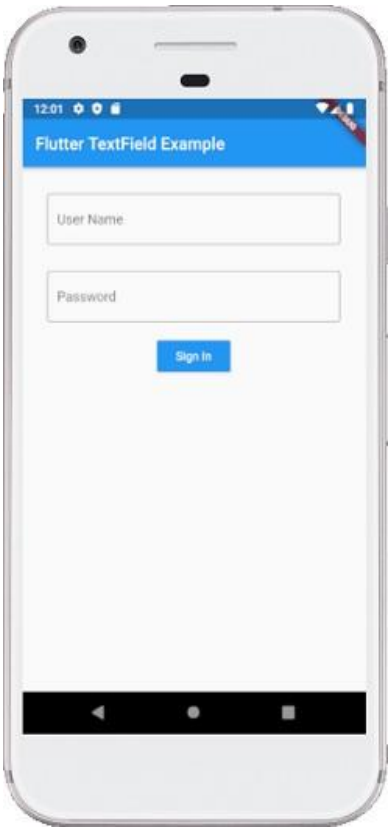
```
import 'package:flutter/material.dart';

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

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

class _State extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
```

```
  title: Text('Flutter TextField Example'),
),
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:



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 CheckboxListTitle:

```
CheckboxListTile(
  secondary: const Icon(Icons.abc),
  title: const Text('demo mode'),
     subtitle: Text('sub demo mode'),
  value: this.subvalue,
  onChanged: (bool value) {
    setState(()  {
    this.subvalue = value;
    });
  },
),
```

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:

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

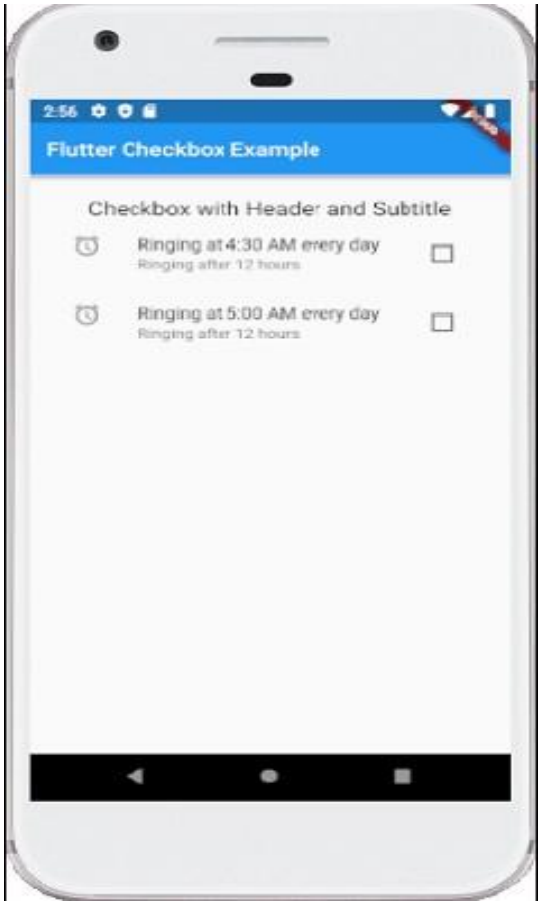
```
void main() {
  runApp(MaterialApp( home: MyHomePage(),));
}

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(
            padding: new EdgeInsets.all(22.0),
            child: Column(
              children: <Widget>[
                SizedBox(width: 10,),
                Text('Checkbox with Header and Subtitle',style: TextStyle(fontSize: 20.0), ),
```

```
    CheckboxListTile(
      secondary: const Icon(Icons.alarm),
      title: const Text('Ringing at 4:30 AM every day'),
      subtitle: Text('Ringing after 12 hours'),
      value: this.valuefirst,
      onChanged: (bool value) {
      setState(() {
          this.valuefirst = value;
        });
      },
    ),
    CheckboxListTile(
      controlAffinity: ListTileControlAffinity.trailing,
      secondary: const Icon(Icons.alarm),
      title: const Text('Ringing at 5:00 AM every day'),
      subtitle: Text('Ringing after 12 hours'),
      value: this.valuesecond,
      onChanged: (bool value) {
      setState(() {
          this.valuesecond = value;
        });
      },
    ),
  ],
)
```

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

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 sts:

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';
```

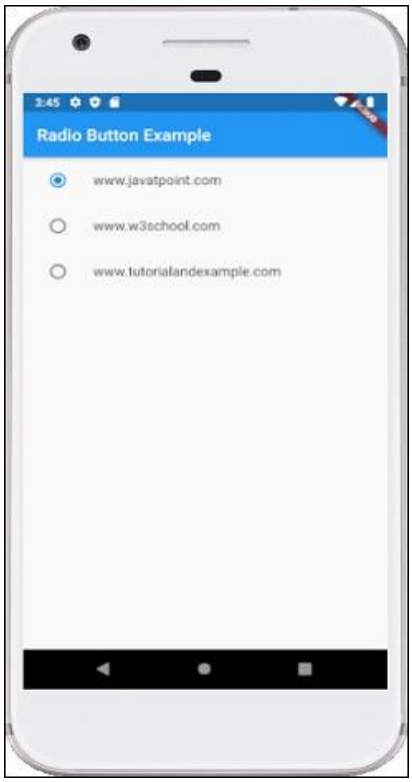
```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: _title,
      home: Scaffold(
        appBar: AppBar(title: const Text(_title)),
        body: Center(
          child: MyStatefulWidget(),
        ),
      ),
    );
  }
}

enum BestTutorSite { javatpoint, w3schools, tutorialandexample }

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

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

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  BestTutorSite _site = BestTutorSite.javatpoint;
```

```
Widget build(BuildContext context) {
  return Column(
    children: <Widget>[
      ListTile(
        title: const Text('www.javatpoint.com'),
        leading: Radio(
          value: BestTutorSite.javatpoint,
          groupValue: _site,
          onChanged: (BestTutorSite value) {
            setState(() {
              _site = value;
            });
          },
        ),
      ),
      ListTile(
        title: const Text('www.w3school.com'),
        leading: Radio(
          value: BestTutorSite.w3schools,
          groupValue: _site,
          onChanged: (BestTutorSite value) {
            setState(() {
              _site = value;
            });
          },
        ),
      ),
```

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

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.



Date:

dates in Flutter according to the requirements is very limited and restrictive. While dealing with dates it should be in human-readable format but unfortunately, there's no way of formatting dates in flutter unless you make use of a third-party package.

we will look into one such package known as the intl package.

Using intl package:

Add the following dependencies to your pubspec.yaml file, you can find the latest dependencies here.

dependencies:

intl: ^0.17.0

Add using terminal:

You can also get the latest intl library using terminal easily:

flutter pub add intl

Import it:

That’s it now import intl package in your Dart code:

```
import 'package:intl/intl.dart';
```

Still, if you face any error using intl, simply use the following command:

```
flutter pub get
```

Now let’s take a look at the below example.

Example:

In the below code we will not be using the intl package for formatting. Also, take a look at the output of the below code.

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

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

class dateDemo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(

      // browser tab title
      title: "Geeksforgeeks",
```

```
// Body
home: Scaffold(

  // AppBar
  appBar: AppBar(

    // AppBar color
    backgroundColor: Colors.green.shade900,

    // AppBar title
    title: Text("Geeksforgeeks"),
  ),

  // Container or Wrapper
  body: Container(
    margin: EdgeInsets.fromLTRB(95, 80, 0, 0),

    // printing text on screen
    child: Text(

      // Formatted Date
      // Builtin format / without formatting
      DateTime.now().toString(),
      style: TextStyle(

        // Styling text
        fontWeight: FontWeight.bold, fontSize: 30),
```

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

Output:

List view;

List view is the most commonly used scrolling widget. It displays its children one after another in the scroll direction. In the cross axis, the children are required to fill the ListView.

If non-null, the itemExtent forces the children to have the given extent in the scroll direction.

If non-null, the prototypeItem forces the children to have the same extent as the given widget in the scroll direction.

Specifying an itemExtent or an prototypeItem is more efficient than letting the children determine their own extent because the scrolling machinery can make use of the foreknowledge of the children's extent to save work, for example when the scroll position changes drastically.

You can't specify both itemExtent and prototypeItem, only one or none of them.

There are four options for constructing a ListView:

1. The default constructor takes an explicit List<Widget> of children. This constructor is appropriate for list views with a small number of children because constructing the List requires doing work for every child that could possibly be displayed in the list view instead of just those children that are actually visible.

2. The ListView.builder constructor takes an IndexedWidgetBuilder, which builds the children on demand. This constructor is appropriate for list views with a large (or infinite) number of children because the builder is called only for those children that are actually visible.

3. The ListView.separated constructor takes two IndexedWidgetBuilders: itemBuilder builds child items on demand, and separatorBuilder similarly builds separator children which appear in between the child items. This constructor is appropriate for list views with a fixed number of children.

4. The ListView.custom constructor takes a SliverChildDelegate, which provides the ability to customize additional aspects of the child model. For example, a SliverChildDelegate can control the algorithm used to estimate the size of children that are not actually visible.

To control the initial scroll offset of the scroll view, provide a controller with its ScrollController.initialScrollOffset property set.

By default, ListView will automatically pad the list's scrollable extremities to avoid partial obstructions indicated by MediaQuery's padding. To avoid this behavior, override with a zero padding property.

This example uses the default constructor for ListView which takes an explicit List<Widget> of children. This ListView's children are made up of Containers with Text.

A ListView of 3 amber colored containers with sample text.

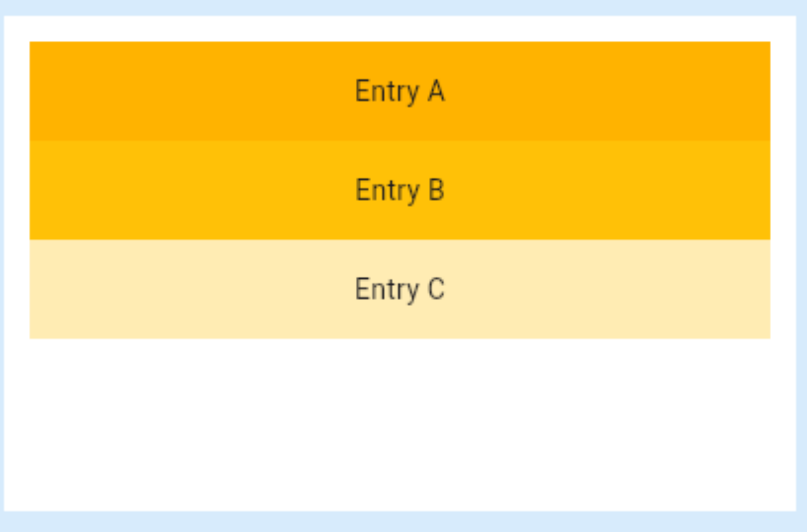
link

content_copy

```
ListView(
  padding: const EdgeInsets.all(8),
```

```
  children: <Widget>[
    Container(
      height: 50,
      color: Colors.amber[600],
      child: const Center(child: Text('Entry A')),
    ),
    Container(
      height: 50,
      color: Colors.amber[500],
      child: const Center(child: Text('Entry B')),
    ),
    Container(
      height: 50,
      color: Colors.amber[100],
      child: const Center(child: Text('Entry C')),
    ),
  ],
)
```

Output


Entry A
Entry B
Entry C