Mobile App Development

(Flutter)

Student Handouts

Session-1 Handouts

Flutter Guide

Flutter is Google's UI toolkit for creating beautiful, natively compiled iOS and Android apps from a single code base. The base language for flutter is dart.

To build any application we start with widgets-the building block of flutter applications. Widgets describe what their view should look like given their current configuration and state. It includes a text widget, row widget, column widget, container widget, and many more.

Widgets: Each elements on a screen of the flutter app is a widget.

Flutter widgets are divided into 14 categories. They are mainly 14 categories in which the flutter widgets are divided. They are mainly segregated on the basis of the functionality they provide in a flutter application.

1. Accessibility: These are the set of widgets that make a flutter app more easily accessible.
2. Animation and Motion: These widgets add animation to other widgets.
3. Assets, Images, and Icons: These widgets take charge of assets such as display images and show icons.
4. Async: These provide async functionality in the flutter application.
5. Basics: These are the bundle of widgets which are absolutely necessary for the development of any flutter application.
6. Cupertino: These are the ios designed widgets.
7. Input: This set of widgets provide input functionality in a flutter application.
8. Interaction Models: These widgets are here to manage touch events and route users to different views in the application.
9. Layout: This bundle of widgets helps in placing the other widgets on the screen as needed.
10. Material Components: This is a set of widgets that mainly follow material design by Google.
11. Painting and effects: This is the set of widgets which apply visual changes to their child widgets without changing their layout or shape.
12. Scrolling: This provides sacrollability of to a set of other widgets that are not scrollable by default.
13. Styling: This deals with the theme, responsiveness, and sizing of the app.
14. Text: This display text.

1. Structure of Flutter Application

.idea: It holds the configuration for Android Studio. We are not going to work with Android Studio so that the content of this folder can be ignored.

.android: This folder holds a complete Android project and used when you build the Flutter application for Android. When the Flutter code is compiled into the native code, it will get injected into this Android project, so that the result is a native Android application.

.ios: This folder holds a complete MAC project and used when you build the Flutter application for iOS.

.lib: it is a folder where we will do our 99 percent of project work. Here we have the Dart files which contain the code of our Flutter application, like main.dart, which is the entry file of the flutter application.

.test: This folder contains a Dart code, which is written for the Flutter application to perform the automated test when building the app. It won't be too important for us here.

.gitignore: It is a text file containing a list of files, file extensions, and folders that tells Git which files should be ignored in a project.

.metadata: It is an auto-generated file by the flutter tools, which is used to track the properties of the Flutter project. This file performs the internal tasks.

.packages: It is an auto-generated file by the Flutter SDK, which is used to contain a list of dependencies for your Flutter project.

flutter_demoapp.iml: It is always named according to the Flutter project's name that contains additional settings of the project. This file performs the internal tasks, which is managed by the Flutter SDK.

pubspec.yaml: It is the project's configuration file that will use a lot during working with the Flutter project. It allows you how your application works. This file contains:

- Project general settings such as name, description, and version of the project.
- Project dependencies.
- Project assets (e.g., images).

2. Flutter First Application

To create Flutter first application

Step 1: Open the Android Studio.

Step 2: Create the Flutter project. To create a project, go to File-> New->New Flutter Project. The following screen helps to understand it more clearly.

Step 3: In the next wizard, you need to choose the Flutter Application. For this, select Flutter Application-> click Next

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

- Project Name: Write your Application Name.
- Flutter SDK Path: <path_to_flutter_sdk>
- Project Location: <path_to_project_folder>
- Descriptions: <A new Flutter hello world application>.

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

After clicking the Finish button, it will take some time to create a project. When the project is created, you will get a fully working Flutter application with minimal functionality.

Step 7: Open the **main.dart** file, Here you can view the Flutter code of your first App.

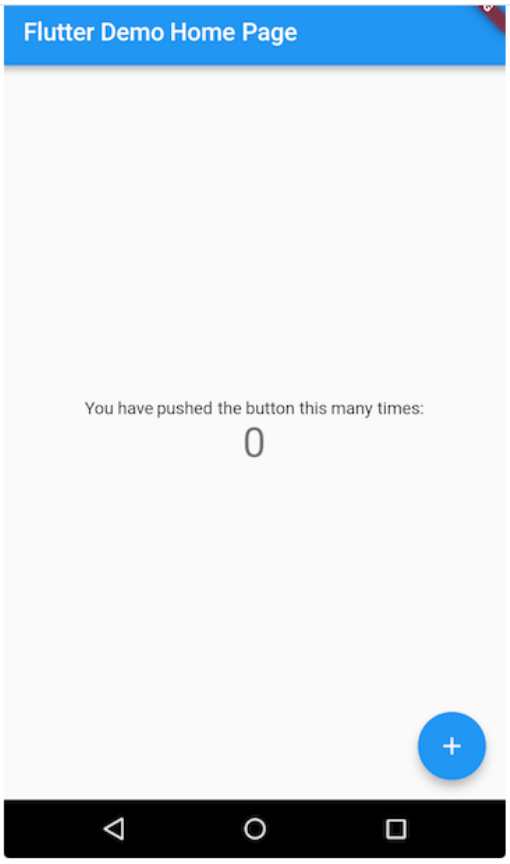
```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);

 // This widget is the root of your application.
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   title: 'Flutter Demo',
   theme: ThemeData(

    primarySwatch: Colors.blue,
   ),
```

```
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}

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

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

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

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
```

```
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ), // This trailing comma makes auto-formatting nicer for build methods.
   );
  }
}
```

Run Our App

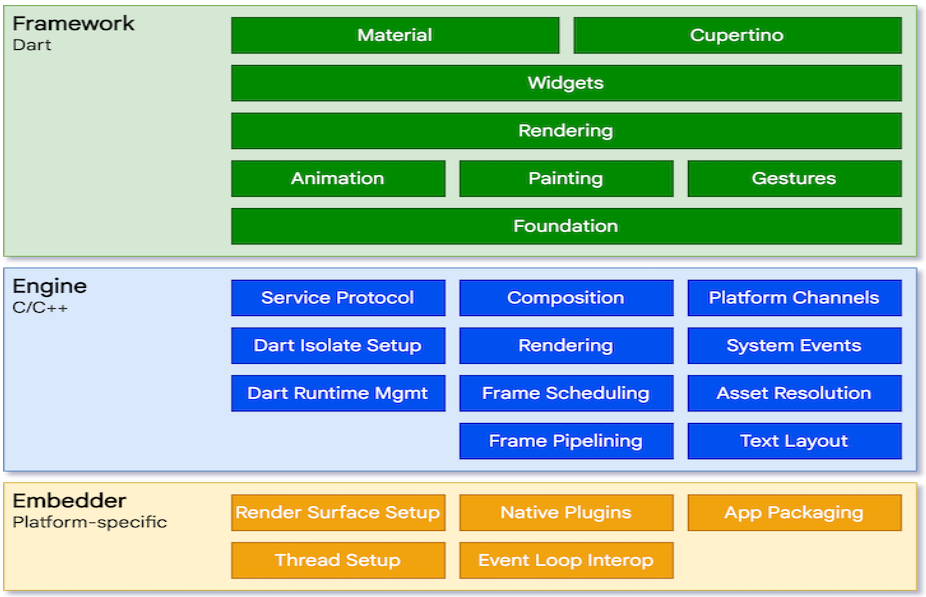
In this counter example, every time you click the floating button with a "+" sign in the lower right corner, the number in the center of the screen will increase by 1.



Session-2 Handouts

Flutter Architecture

Flutter is designed as an extensible, layered system. It exists as a series of independent libraries that each depend on the underlying layer. No layer has privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable.The Flutter architecture mainly comprises of three components.



Embedder: To the underlying operating system, Flutter applications are packaged in the same way as any other native application. A platform-specific embedder provides an entrypoint; coordinates with the underlying operating system for access to services like rendering surfaces, accessibility, and input; and manages the message event loop. The embedder is written in a language that is appropriate for the platform: currently Java and C++ for Android, Objective-C/Objective-C++ for iOS and macOS, and C++ for Windows and Linux. Using the embedder, Flutter code can be integrated into an existing application as a module, or the code may be the entire content of the application. Flutter includes a number of embedders for common target platforms, but other embedders also exist.

Engine C/C++: At the core of Flutter is the **Flutter engine**, which is mostly written in C++ and supports the primitives necessary to support all Flutter applications. The engine is responsible for rasterizing composited scenes whenever a new frame needs to be painted. It provides the low-level implementation of Flutter's core API, including graphics (through Skia), text layout, file and network I/O, accessibility support, plugin architecture, and a Dart runtime and compile toolchain.

The engine is exposed to the Flutter framework through dart:ui, which wraps the underlying C++ code in Dart classes. This library exposes the lowest-level primitives, such as classes for driving input, graphics, and text rendering subsystems.

Flutter framework: Typically, developers interact with Flutter through the **Flutter framework**, which provides a modern, reactive framework written in the Dart language. It includes a rich set of platform, layout, and foundational libraries, composed of a series of layers. Working from the bottom to the top, we have:

Basic foundational classes, and building block services such as animation**,** painting**, and** gestures that offer commonly used abstractions over the underlying foundation.

The rendering layer provides an abstraction for dealing with layout. With this layer, you can build a tree of renderable objects. You can manipulate these objects dynamically, with the tree automatically updating the layout to reflect your changes.

The widgets layer is a composition abstraction. Each render object in the rendering layer has a corresponding class in the widgets layer. In addition, the widgets layer allows you to define combinations of classes that you can reuse. This is the layer at which the reactive programming model is introduced.

The Material and Cupertino libraries offer comprehensive sets of controls that use the widget layer's composition primitives to implement the Material or iOS design languages.

Session-3 Handouts

Dart Student Guide

Dart is the open-source programming language originally developed by Google.

The Dart SDK comes with its compiler – the **Dart VM** and a utility dart2js which is meant for generating **Javascript equivalent of a Dart Script.**

Dart is extensively use to create single-page websites

Execution of program:

1. **Online Compiler:** The online compiler which support Dart is **Dart Pad**.
2. **IDE:** The IDEs which support Dart are WebStorm, Intellij, Eclipse, etc. Among them WebStorm from Jetbrains is available for Mac OS, Windows and Linux.
3. The dart program can also be compile through terminal by executing the code **dart file_name.dart**.

Dart Function

Function with Optional

1) **Optional Positional Parameter**
2) **Optional Named Parameter**
3) **Optional Parameter with Default Value**

Dart's optional parameters are optional in that **the caller isn't required to specify a value for the parameter when calling the function**. Optional parameters can only be declared after any required parameters.

3. Constants

Dart Constant is defined as an immutable object, which means it can't be changed or modified during the execution of the program. Once we initialize the value to the constant variable, it cannot be reassigned later.

The Dart constant can be defined in the following two ways.

- Using the final keyword
- Using the const keyword

Constant using final Keyword

final const_name; Or final data_type const_name

Constant Using const Keyword

const const_name Or const data_type const_name

4. Static

The static keyword is used to declare the class variable and method. It generally manages the memory for the global data variable. The static variables and methods are the member of the class instead of an individual instance. The static variable or methods are the same for every instance of the class, so if we declare the data member as static then we can access it without creating an object. The class object is not required to access the static method or variable we can access it by putting the class name before the static variable or method.

Static Variable

static [data_type] [variable_name];

Static Method

The static methods can use only static variables and can invoke the static method of the class.

static return_type method_name() { //statement(s) }

Control Flow Statements

5. If-else

```
if (condition1) {    // statement(s)  }
        else if(condition2) {  // statement(s)  }
                else {   // statement(s)   }
```

6. Switch

```
int n = 3;
    switch (n) {
      case 1:
            print("Value is 1");
             break;
       case 2:
```

```
            print("Value is 2");


            break;
        default:
            print("Out of range");
            break;
        }
    }
```

7. For & For in Loop

The for..in loop is similar to for loop but different in its syntax.

```
for (var name in expression) {
    print('$name')
}
```

Normally use to get information in List, Map, and Sets

8. While Loop

```
while(condition){
        //statement(s);
        // Increment (++) or Decrement (--) Operation;
    }
while(true) {
    //Infinite while loop statement
}

do {
    // loop body
}while(condition);
```

Normal Function

```
return_type func_name (parameter_list):
    {
      //statement(s)
     return value;
    }
```

Anonymous Function

An anonymous function is like a named function but do not have names associated with it.

```
(parameter_list)
{
   statement(s)
}
```

Here is an example, where an Anonymous function is returned.

```
Function makeAdder(int addBy) {
        return (int i)  => addBy + I;
}
```

Recursion

Recursion is a technique, where a function calls itself again and again as its subroutine until problem is not solved. In recursion, there must be a base cause to terminate the recursion. This technique is used for searching, sorting, Inorder/Preorder/Postorder, Tree Traversal, and DFS of Graph algorithms.

Class Declaration

class class_name {
  <fields>
  <getters/setters>
  <constructors>
  <functions>
}

A default getter/ setter is associated with every class. However, the default ones can be overridden by explicitly defining a setter/ getter.

Constructors

A constructor is a special function of the class that is responsible for initializing the variables of the class. If you don't declare a constructor, a default **no-argument constructor** is provided.

Named Constructor

Named constructors to enable a class define multiple constructors.
Class_name.constructor_name(param_list)

```
Car.namedConst(String engine) {
    print("The engine is : ${engine}");
  }
```

This Keyword

The **this** keyword refers to the current instance of the class.

Getter & Setter

Dart Language use get keyword for getter and set keyword for setters. A default getter/setter is associated with every class. However, the default ones can be overridden by explicitly defining a setter/ getter.

Return_type get identifier { }

set identifier { }

Inheritance

In Dart language Child classes inherit all properties and methods except constructors from the parent class.

Interface

Interface defines the same as the class where any set of methods can be accessed by an object. The Class declaration can interface itself.

The keyword **implement** is needed to be writing, followed by class name to be able to use the interface.

Implementing class must provide a complete definition of all the functions of the implemented interface.

class InterfaceName { }

class ClassName implements InterfaceName { }

9. Number

The **parse()** static function allows parsing a string containing numeric literal into a number.

```
print(num.parse('12'));
  print(num.parse('10.91'));
```

The parse function throws a **FormatException** if it is passed any value other than numerals.

10. String

The operator plus (+) is a commonly used mechanism to concatenate / interpolate strings.

You can use "${}" can be used to interpolate the value of a Dart expression within strings.

```
String str1 = "The sum of 1 and 1 is ${n}";
  print(str1);

  String str2 = "The sum of 2 and 2 is ${2+2}";
  print(str2);
```

String Properties

codeUnits

isEmpty

Length

Methods to manipulate Strings

toLowerCase()

toUpperCase()

trim(): Returns a new string by removing all leading and trailing spaces.

compareTo(): Compare two strings. str1.compareTo(str2) it returns {0: when strings are equal, 1 or -1: when strings are not equal}

replaceAll(): Replaces all substrings that match the specified pattern

```
String str1 = "Hello World";
  print("New String: ${str1.replaceAll('World','ALL')}");
```

split()

substring()

toString()

codeUnitAt()

Session-4 Handouts

11. Dart Collection

12. ➔ List

A very commonly used collection in programming is an array. Dart represents arrays in the form of List objects. A list is an ordered group (array) of objects. The dart:core library provies the list class that enables creation and manipulation of lists. It has two components

Elements: Values, contained by list known as elements

Index: Each element in the List is identified by a unique number called the index, start from zero.

Lists can be **Fixed Length List** or **Growable List**

Fixed Length List: Length of List cannot change at runtime. It requires two steps to create!

Step -1: var list_name = new List(initial_size)

Step-2: list_name[index] = value

Example:-

```
var lst = new List(3);
  lst[0] = 12;
  lst[1] = 13;
  lst[2] = 11;
  print(lst);
```

Growable List: Length of List can change at runtime. It requires declaring and initializing steps!

Step-1: var list_name = new List()

Step-2: list_name[index] = value; or list_name.add(value)

OR

var list_name = [val1, val2, val3]

Example

```
var lst = new List();
  lst.add(12);
  lst.add(13);
  print(lst);
```

Null Safety

Null Safety means a variable cannot contain a 'null' value, unless you initialized with null to that variable. When null safety is enabled, the default 'List' constructor is not available. Therefore you need extra functionality to declare a list.

If an initial size needs to be provided and there is a single reasonable initial value for the elements, then use List.filled:

```
var fixed_list = List.filled(3,0)
fixed_list[1] = 33;
print(fixed_list)
```

Output

```
[0,33,0]
```

While for the Growable Length List, you can try this

```
var growableList = new List<int>.filled(0,0, growable:true);  //filled(length, fill)
growableList = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90];
print('The elements in the growable list include: $growableList');
//for String type List
 var  myType = new List<String>.filled(0,'',growable:true);
```

List Properties:

first:	Returns the first element in the list.
isEmpty:	Returns true if the collection has no elements.
isNotEmpty:	Returns true if the collection has at least one element.
length:	Returns the size of the list.
last:	Returns the last element in the list.
reversed:	Returns an iterable object containing the lists values in the reverse order
Single:	Checks if the list has only one element and returns it.

Inserting Elements into List:- add(val), addAll([val1, val2 ..]), insert(index, value), insertAll(index,[val1,val2 ..])

Removing Elements from List:- remove(val1), removeAt(index), removeLast(), removeRange()

List Copy (Iterating List Elements): List can be iterating using the forEach method.

```
list1.forEach((item){}
```

Adding Widget in a list.

```
List<Icon>  scorekeeper = [
Icon(
  Icons.check,
  color: Colors.green,),
Icon(
  Icons.check,


  color: Colors.red,),
];

Widget build() {
….
child: FlatButton(
   onPressed() {
      setState(()
           scorekeeper.add(
             Icon(
                Icons.check,
                color: Colors.green,),)}
)}
)
}
```

13. ➔ <u>Set</u>

The Dart Set is the unordered collection of different values of the same type. It is special type of List where all the inputs are unique (doesn't contain any repeated input)

(A list is an ordered collection of elements where the same element may occure several time at different position)

It can be declare as

var variable_name = <variable_type>{};

Or

Set <variable_type> variable_name = {}

Add Elements in Set

Set_variable_name.add(value);

Access Element in Set

Set_variable_name.elementAt(index);

Search Element in Set

Set_variable_name.contains(value); *return true/false*

Remove Set Element

Set_variable_name.remove(value);

Iterating Over a Set Element

Set_variable_name.forEach(value) {print('$value')}

Remove All Set Elements

Set_variable_name.clear();

TypeCast Set to List: The set Object can convert into the List Object using the toList() method.

List <type> list_name = Set_variable_name.toList();

First	It is used to get the first element in the given set.
isEmpty	If the set does not contain any element, it returns true.
isNotEmpty	If the set contains at least one element, it returns true
Length	It returns the length of the given set.
Last	It is used to get the last element in the given set.
Hashcode	It is used to get the hash code for the corresponding object.
Single	It is used to check whether a set contains only one element.

Set Properties

14. Set Operations

Union - The union is set to combine the value of the two given sets a and b.

x.union(y)

Intersection - The intersection of the two set a and b returns all elements, which is common in both sets.

x.intersection(y)

Subtracting - The subtracting of two sets a and b (a-b) is the element of set b is not present in the set a.

y.difference(z)

15. ➔ Map

Dart Map is an object that stores data in the form of a key-value pair. Each value is associated with its key, and it is used to access its corresponding value. Map can be declared by using curly braces {} ,and each key-value pair is separated by the commas(,). Remember key must be unique, but the same value can occur multiple times.

Map Declaration

Map can be defined in two methods.

- Using Map Literal
- Using Map Constructor

Using Map Literal

```
var map_name = {key1:value1, key2:value2 [.....,key_n: value_n]}
var student = {'name':'Tom','age':'23'};
```

Adding Value at runtime

```
var student = {'name':' tom', 'age':23};
student['course'] = 'B.tech';
```

Using Map Constructor

```
var map_name = new map()
map_name[key] = value
```

Map Properties

Keys	It is used to get all keys as an iterable object.
Values	It is used to get all values as an iterable object.
Length	It returns the length of the Map object.
isEmpty	If the Map object contains no value, it returns true.
isNotEmpty	If the Map object contains at least one value, it returns true.

Map Methods

addAll() - It adds multiple key-value pairs.

remove() - It eliminates all pairs from the map.

forEach() - It is used to iterate the Map's entries.

Dart Libraries

To use a library in your code, 'import' keyword is used. Importing makes the components in a library available to the caller code

import 'dart:io'

if you want to use only part of a library, you can selectively import the library as

```
import 'package: lib1/lib1.dart' show foo, bar;
// Import only foo and bar.
import 'package: mylib/mylib.dart' hide foo;
// Import all names except foo
```

Some commonly used libraries are

dart:io - File, socket, HTTP, and other I/O support for server applications. This library does not work in browser-based applications. This library is imported by default.

dart:core - Built-in types, collections, and other core functionality for every Dart program. This library is automatically imported.

dart:math - Mathematical constants and functions, plus a random number generator.

```
print("Square root of 36 is: ${sqrt(36)}");
```

dart:convert - Encoders and decoders for converting between different data representations, including JSON and UTF-8.

dart:typed_data.- Lists that efficiently handle fixed sized data (for example, unsigned 8 byte integers).

16. Custom Libraries

Dart also allows you to use your own code as a library. Creating a custom Library involves the following step

Declaring a Library

Use the following library statement in your library file

library library_name
// library contents go here

Associating a Library

To access library, if exist in same folder

import 'library_name'

To access library, if exist in different folder

Import 'dir/library_name'

Example:

```
library calculator_lib;
import 'dart:math';

//import statement after the libaray statement
int add(int firstNumber,int secondNumber){
  print("inside add method of Calculator Library ") ;
  return firstNumber+secondNumber;
}
```

Now we will import the library

```
import 'calculator.dart';
void main() {
  var num1 = 10;
  var num2 = 20;
  var sum = add(num1,num2);
  print("$num1 + $num2 = $sum");
}
```

Widgets: In Flutter, everything is a widget, which is the core concept of this framework. Widget in the Flutter is basically a user interface component that affects and controls the view and interface of the app. It includes graphics, text, shapes, and animations that are created using widgets. The widgets are similar to the React components.
In Flutter, the application is itself a widget that contains many sub widgets. It means the app is the top-level widget, and its UI is build using one or more children widgets, which again includes sub child widgets.

Design Specific Widgets: The Flutter framework has two sets of widgets that conform to specific design languages. These are Material Design for Android application and Cupertino Style for IOS application.

17. Flutter Widgets

Whenever you are going to code for building anything in Flutter, it will be inside a widget. Widgets are nested with each other to build the app. Its means the root of your app is itself a widget, and all the way down is a widget also. The road map of widgets are
MyApp ➔ MaterialApp ➔ MyHomePage ➔ Scaffold ➔AppBar (=> Text) / Center (=> Column) / Floating Action Button (=> Icon)

Types of Widget

1) Visible: related to the user input and output data, like Text, Button, Image, Icon
2) Invisible: related to the layout and control of widgets, like Column, Row, Center, Padding, Scaffold, Stack

Widget State

1) StatefulWidget: These are dynamic because it can change the inner data during the widget lifetime. Examples are Checkbox, Radio, Slider, InkWell, Form, and TextField.
2) StatelessWidget: Thses do not have state information. It remains static throughout its lifecycle. Examples are Text, Row, Column, Container, etc. This is also known as Immutable state that can NOT be changed during runtime.

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

The widgets whose state () can not be altered once they are built are called stateless widgets.

When a Flutter builds a Stateful Widget, it creates a State object. You have so many option for state management like

18. State Management in Flutter

State Management is the strategic approach to manage all the interactions that a user performs on an application and then reflect those changes to UI, Update database, server requests etc. State management helps to align and integrate the core business logic inside the application with servers and databases.

18.1.1. List of Flutter State Management Approach / Techniques

StatefulWidget	Async Redux
ChangeNotifier with Provider	RxDart view models
BLoC library	States Rebuilder
MobX	Osam
Redux	Flutter Hooks
Fish Redux	StreamProvider
	StateNotifier

Your application will be in either of two below states:

a. Local / Ephemeral State

Whenever you need to change the state of a single page view which may contain UI controls or animation then it is considered as a local state. You can easily do that using the Stateful widget. Here is an 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";
        });
      },
    );
  }
}
```

b. Shared / App state

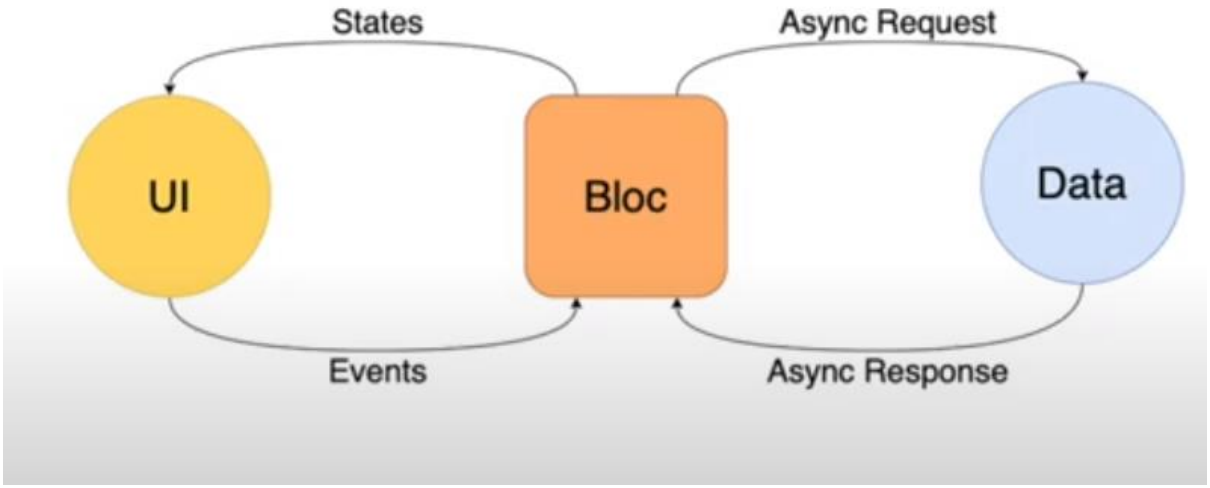
Whenever you need to change the state of multiple widgets which are shared across your application then it is Share app State. Some of the examples of this state are User preferences, Login info, notifications in a social networking app, the shopping cart in an e-commerce app, read/unread state of articles in a news app, etc.

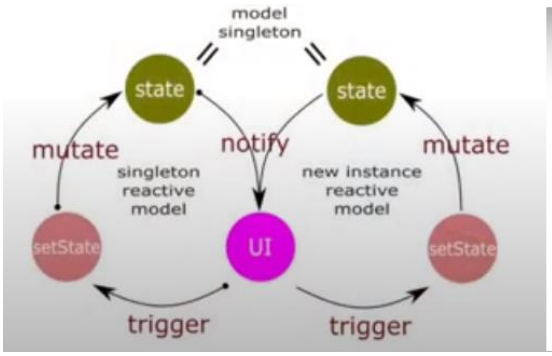
The simplest example of app state management can be learned by using the provider package. The state management with the provider is easy to understand and requires less coding. A provider is a third-party library. Here, we need to understand three main concepts to use this library.

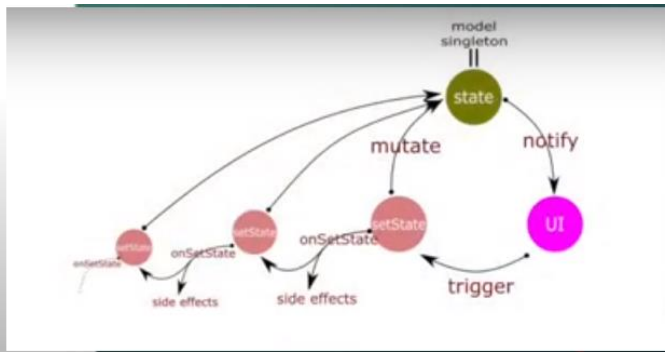
1. ChangeNotifier
2. ChangeNotifierProvider
3. Consumer

ChangeNofifier: ChangeNotifier is a simple class, which provides change notification to its listeners through notifyListener.

ChangeNotifierProvider:







19. State Management with Provider

Provider is one of the many state management options when using Flutter. It's one of the first state manager recommended by Flutter itself and one of the simplest.

Session-6 Handouts

20. Flutter First App

To start Flutter programming, you need first to import the Flutter package. Here, we have imported a Material package.

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

It is similar to the main method in other programming languages. It calls the runApp function and pass it an object of MyApp. The primary purpose of this function is to attach the given widget to the screen. MyHomePage is similar to MyApp, except it will return the Scaffold widget is a top-level widget after the MaterialApp widget for creating the user interface. This widget contains two properties appBar and body. The appBar shows the header of the app, and body property shows the actual content of the application. AppBar render the header of the application, Center widget is used to center the child widget, and Text is the final widget used to show the text content and display in the center of the screen.

Create Your Own App

```
void main() => runApp(MaterialApp (
  home: Text('Hi Hello'),
) );
```

Press RUN

Another way to write above code

```
void main() {
   runApp(MyApp());
}
Class MyApp extends Stateless {
@override
  Widget build(BuildContext context){
   return MaterialApp(
   home: Text('Hi Hello'),);
}
```

MaterialApp is a predefined class in flutter. It is likely the main or core component of flutter.it contains widgets that are used for the material design of an application.

We can access all the other components and widgets provided by Flutter SDK like Text, Dropdownbutton, AppBar, Scaffold, ListView, StatelessWidget, StatefulWidget IconButton, TextField, Padding, ThemeData, etc. are the widgets that can be access using MaterialApp Class.

```
void main() =>  runApp(MaterialApp (
  home: Scaffold (
    appBar: AppBar (
      title: Text('MyApp Flutter'),
   ),  //AppBar
)    //scaffold
));  //MatrialApp
```

Scaffold widget allow us to implement basic layout for our app. This class used in flutter for application design. This class provide API for showing snackBar, drawers, bottom sheets.
AppBar is used to display the toolbar and other widgets at the top of an application. It is placed at the top of the application.

- How to import and use custom fonts in an application.
- Create your own widgets with the help of Stateless Widgets.

20.1. Images

Image can be declare as
Method-1 *child: Image(image: AssetImage('images/dice1.png'),)*
or
Method-2 *child: Image.asset('images/dice1.png'))*
When you create an app in Flutter, it includes both code and assets (resources). An asset is a file, which is bundled and deployed with the app and is accessible at runtime. An asset can include static data, configuration files, icons, and images.
To display an image in Flutter, first, we create a new folder inside the root of the Flutter project and named it assets. Inside this folder, add one or more images manually. After that update the pubspec.yaml file as

```
assets:
    - assets/tablet.png
    - assets/background.png
```

if the assets folder contains more than one imag, we can include it by specifying the directory name with the slash "/" character at the end as

```
assets:
    - assets/
```

Now write code inside main.dart file, where image as widget declare as

```
body: Center(
      child: Image (
           image: AssetImage('assets/tablet.png'),
           image: AssetImage('Images/diamond.png')
),),
```

If image source is Internet then

```
body: Center(
      child: Image (
           image: NetworkImage(' https://www.pexels.com/photo/green-and-blue-peacock'),
),),
```

You can also set the height and width as

```
body: Center(
      child: Image (
           image: NetworkImage(' https://www.pexels.com/photo/green-and-blue-peacock'),
           height: 400
           width: 250
),),
```

A sort way

```
body: Center(
      child: Image.asset('assets/tablet.png')
),
OR
body: Center(
      child: Image.network('assets/tablet.png')
),
```

If you want to add all images in your application available in Images folder just mention the following line in .yaml file

```
assets:
   - Images/
```

20.2. Icon

An icon is a graphic image. It can be selectable or non-selectable. It can contains a hyperlink to go to another page

```
Body: Center (
  child: Icon (
     Icons.airport_shuttle,
     color: Colors.lightBlue,
     size: 50.0,
),),
```

```
child: Icon()
child: RaisedButton()
child: FlatButton()
child: Text()
child: Container()
child: Image.assets() / Image.network()
child: Row()
```

```
body: Center()
body: Container()
body: Row()
```

21. TextField

Retrieving Value from TextField

Flutter allows the user to **retrieve the text in mainly two ways**:

1) onChanged Method

This method store the current value in a simple variable and then use it in the TextField widget

```
TextField(
  onChanged: (text) {
    value = text;
   print("TextField Name value is = $value");
   },
)
```

2) Controller Method

```
class _myProjState extends State<myProj> {
    TextEditingController nameController = TextEditingController();

    printTextField(){
       print("User Name: ${nameController.text}") ;
    }
   TextField(
      controller: nameController,),
   ElevatedButton(
      onPressed: () { printTextField(); }
```

In above code, first, we create a TextEditingController object (nameController) and then add the controller to a TextField. Create a function printTextField to be invoked when a button is pressed to print controller text.

You can notify about changes to TextField using notifyListeners as

```
class _myProjState extends State<myProj> {
    TextEditingController nameController = TextEditingController();

TextField(
  controller: nameController,
),

@override
void initState() {
 // TODO: implement initState
 super.initState();
 nameController.addListener(() { textListener(); });
}
textListener() {
 print("Current Text is ${nameController.text}");
}
```

You can set the initial value of text field by

```
nameController.text = 'initial value'
```

You can also clear the TextField value after submission then use

```
nameController.clear()
```

Use the despose() method of to free the resources used by TextEditingController

```
void dispose(){
  nameController.dispose();
  super.dispose();
}
```

onSubmitted vs onEditingComplete

These two properties are called when user writes some information in TextField and press enter just like whatsapp.

22. Layout Widgets:

23. Single Child Layout Widgets

Align / AspectRatio / Baseline / Container / Padding / Center / ConstrainedBox / Expanded / FittedBox / CustomSingleChildLayout / FractionallySizedBox / IntrinsicHeight / IntrinsicWidth / LimitedBox / Offstage / OverflowBox / SizeBox / SizedOverflowBox / Transform

24. Container Class:

Container class in flutter is a convenience widget that combines common painting, positioning, and sizing of widgets. A Container class can be used to store one or more widgets and position it on the screen according to our convenience. Basically a container is like a box to store contents.

- A basic container element that stores a widget has a **margin**, which separates the present container with other contents.
- The total container can be given a border of different shapes, for example, rounded rectangles, etc.
- A container surrounds its child with **padding** and then applies additional constraints to the padded extent (incorporating the width and height as constraints, if either is non-null).
- Containers with no children try to be as big as possible. The width, height and constaints arguments to the constructor override this.

Properties of Container Class:

- **1. child**: Container widget has a property 'child:' which stores its children. The child class can be any widget. Let us take an example, taking a text widget as a child.

```
: Container(child
    : Text("Hello! i am inside a container!",
        style
        : TextStyle(fontSize : 20)),
```

- **2. color**: The color property sets the background color of the entire container. Now we can visualize the position of the container using a background color.

```
Body
: Container(color
            : Colors.purple,
             child
            : Text("Hello! i am inside a container!",
                 style
                 : TextStyle(fontSize : 20)),
```

- **3. height and width:** By default, a container class takes the space that is required by the child. We can also specify height and width to the container based on our requirements.

```
Body
: Container(
   color : Colors.purple,
  height: 200,
   width: double.infinity,

            child
           : Text("Hello! i am inside a container!",
                 style
                 : TextStyle(fontSize : 20)),
```

- **4. margin:** The margin is used to create an empty space around the container. Observe the white space around the container. Here EdgeInsets.geometry is used to set the margin .all() indicates that margin is present in all four directions equally.

```
Body
   : Container(
      color : Colors.purple,
     height: 200,
      width: double.infinity,
      margin: EdgeInsets.all(20),

      child
     : Text("Hello! i am inside a container!",
          style
```

```
: TextStyle(fontSize : 20)),
```

- **5. padding:** The padding is used to give space form the border of the container form its children. Observe the space between the border and the text.

```
padding: EdgeInsets.all(30),
```

- **6. alignment:** The alignment is used to position the child within the container. We can align in different ways: bottom, bottom center, left, right, etc. here the child is aligned to the bottom center.

```
alignment: Alignment.bottomCenter,
```

- **decoration**: The decoration property is used to decorate the box(e.g. give a border). This paints behind the child. Whereas foregroundDecoration paints in front of a child. Let us give a border to the container. But, both color and border color cannot be given.

```
decoration: BoxDecoration(
    border: Border.all(color: Colors.black, width: 3),
),
```

- **transform:** This property of container helps us to rotate the container. We can rotate the container in any axis, here we are rotating in the z-axis.

```
transform: Matrix4.rotationZ(0.1),
```

- **9. constraints:** When we want to give additional constraints to the child, we can use this property.
- **10. clipBehaviour :** This property takes in Clip enum as the object. This decides whether the content inside the container will be clipped or not.
- **11. foregroundDecoration:** This parameter holds Decoration class as the object. It controls the decoration in front of the Container widget.

Note: You can wrap the Container with SafeArea widget to avoid the intrusions by the operating system.

25. SizedBox

SizedBox is a simple box with a specified size. Normally it can use to create some space between two widgets.
If not given a child, SizedBox will try to size itself as close to the specified height and width

```
const SizedBox(
  width: 200.0,
  height: 300.0,
  child: Card(child: Text('Hello World!')),
)
```

26. Card

A card is a sheet used to represent the information related to each other, such as an album, a geographical location, contact details, etc. A card in Flutter is in rounded corner shape and has a shadow. Normally card properties are

- It has margin but do not have padding property. You can use padding as parent to configure padding for card and vice versa.

```
Padding( padding: EdgeInsets.all(8),
            child: Card(child: Text('Hello World!')),)
```

Note: if you make the Card as a child of Padding, padding shrinks the constraints by the given padding, causing the child to layout at a smaller size. Padding then sizes itself to its child's size.

- If you need shadow for container, use Card instead of container. But if you need height and width for card, use container as parent of Card.
- Shape could be OutlineInputBorder or CircleBorder

```
shape:  OutlineInputBorder( borderRadius: BorderRadius.circular(10),
            borderSide: BorderSide(color: Colors.green, width: 1)
```

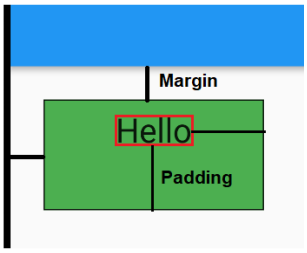
```
shape: CircleBorder(side: BorderSide(width: 1, color: Colors.white),
```

- Cards comes with white color by default.

27. ListTile:

It can be use to display icons (in leading or trailing position) and text in a card. It contains one or three lines of text (title, subtitle) optionally flanked by icons or other widgets, such as check boxes.
List tiles are typically used in ListViews, or arranged in Columns in Drawers and Cards.

28. Padding and Margin:



Margin means the spacing outside of the border, while padding is the spacing inside the border.

In other words, a margin can be defined as the space between two widgets and padding is the distance of a widget from its outer boundary.

There are two ways to set Padding in flutter first is using the Padding Widget and second is Wrap the widget in container widget and apply the padding on Container widget.

```
//Creating Padding widget and make a Text widget as child in Padding widget.
body: Padding (
  padding: EdgeInsets.all(90),
  child: Text("Hello"),
),
//Creating a Container widget and make a child Text widget in Container widget.
body: Container (
  padding: EdgeInsets.all(20),
  color: Colors.grey[400],
  child: Text('Hello'),
),
```

You can provide padding through EdgeInsets with following properties

```
//same padding to all the four sides: left, top, right, bottom
padding: EdgeInsets.all(10)

///provide padding to left, top, right, bottom respectively
padding: EdgeInsets.fromLTRB(10, 15, 20, 5)

///provide padding only to the specified sides
padding: EdgeInsets.only(left: 20, top:30)

//provide padding for vertical and horizontal
Padding: EdgeInsets.symmetric(horizontal:30, vertical:20)
```

Margin is only define with EdgeInsets in Container with similar properties.

28.1. Expended Wiget

Using an Expended widget makes a child of a Row, Column, or Flex expand to fill the available space in the main axis.

Expanded widget makes a child of a Row, Column, or Flex expand to fill the available space along the main axis (e.g., horizontally for a Row or vertically for a Column).

```
body: Center(
      child: Row(`
       children: <Widget>[
        Expanded(
         flex: 2,
         child: Container(
          color: Colors.amber,
          height: 100,
         ),
        ),
        Expanded(
         flex: 1
         child: Container(
          color: Colors.amber,
          height: 100,
         ),
        ),
       ],
      ),
```

The flex property of Expanded decide the area covered by container in a row. It is just like wight property in android XML.

29. Divider

Divider is a once device pixel thick horizontal line, with padding on either side (up and down). Its width depends on parent widget width. Dividers can be used in lists, Drawers, and elsewhere to separate content vertically or horizontally.

```
SizedBox(
  height: 20.0,
  width: 150.0
  child: Divider(
    color: Colors.teal.shade100),)
```

30. <u>Multi-Child Layout Widgets</u>

Column / CustomMultiChildLayout / Flow / GridView / IndexedStack / LayoutBuilder / ListBody / ListView / Row / Stack / Table / Wrap

31. Flutter Rows & Columns

Flutter Row widget is used to display its widgets in horizontal array

The syntax to display a Row widget with children is shown as

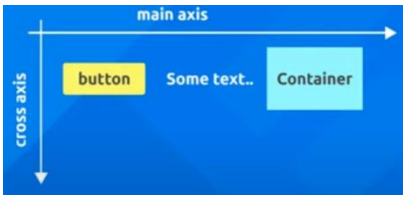
```
Row(
  children: const <Widget>[
],)
```

Column widget is used to display its widgets in vertical array

```
Column(
  children: const <Widget>[
],)
```

Widgets Alignment inside Rows & Colums

All widgets can be align in row/column with horizontally or vertically with mainAxisAlignment and crossAxisAlignment respectively.



MainAxisAlighment: mainAxisAlignment property of Row/Column align the widgets with horizontal and vertical values.

MainAxisAlignment.center - put all widgets in row/colum centerly without and padding

MainAxisAlignment.spaceBetween - put all widget from left to right or up to down without edge space

MainAxisAlighment.spaceEvenly - put all widget from left to right or up to down with edge space

MainAxisAlighment.end - put all widget at right or bottom side

MainAxisAlighment.start - put all widget at left or top side (default)

crossAxisAlignment:

CrossAxisAlignment.stretch - stretch the widget from top to bottom

The size of the row and column can be fixed by using expanded or flexible widgets.

- Widget under Flexible are by default WRAP_CONTENT although you can change it using parameter fit.
- Widget under Expanded is MATCH_PARENT you can change it using flex.

31.1. Flexible Widget

Using a Flexible widget gives a child of a Row, Column, or Flex the flexibility to expand to fill the available space in the main axis (e.g., horizontally for a Row or vertically for a Column), but, unlike Expanded, Flexible does not require the child to fill the available space.

You can also tell Flexible how strict its sizing should be by specifying a fit, which can be either loose or tight.

fit: FlexFit.loose or fit: FlexFit.tight

```
Column(children: [
  Flexible(
    flex: 2,
    fit: FlexFit.loose,
    child: Container(
        height: 100,
        color: Colors.cyan
    ),
  ),
  Flexible(
    flex: 3,
    child: Container(color: Colors.teal),
  ),
  Flexible(
    flex: 1,
    child: Container(color: Colors.indigo),
  ),
]);
```

32. Scaffold

Scaffold is a class in flutter which provides many widgets or we can say APIs like

appBar, body, floatingActionButton, floatingActionButtonLocation, persistentFooterButtons, drawer, endDrawer, bottomNavigationBar, bottomSheet, floatingActionButtonAnimator, backgroundColor,

Scaffold is able to expand or occupy the whole device screen. In other words, we can say that it is mainly responsible for creating a base to the app screen on which the child widgets hold on and render on the screen.
Scaffold will provide a framework to implement the basic material design layout of the application.
Note: Flutter widget tree structure based on properties and its values.
appBar: It is a horizontal bar that is mainly display at the top of the Scaffold widget. Its properties could be elevation, title, brightness, etc.

```
appBar: AppBar(
  title: Text(“First Flutter Application”),
),
```

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

```
  body: Center(
  child: Text("Welcome to Javatpoint",
    style: TextStyle( color: Colors.black, fontSize: 30.0, fontWeight: FontWeight.bold
                      letterSpacing: 2.0, fontFamily: serif
    ), ),
```

In the above code, we have displayed a text **"Welcome to Javatpoint!!"** in the body attribute. This text is aligned in the **center** of the page by using the **Center widget**.

drawer: It is a **slider panel** that is displayed at the side of the body. Usually, it is hidden on the mobile devices, but the user can swipe it left to right or right to left to access the drawer menu.
The mobile apps that use Material Design have two primary options for navigation. There navigations are Tabs and Drawers. A drawer is an alternative option for tabs because sometimes the mobile apps do not have sufficient space to support tabs.

```
Scaffold(
  drawer: Drawer ( )
);
```

Now add a ListView as a child of Drawer Widget

```
drawer: Drawer(
child: ListView(
children: <Widget>[],
),),
```

Now you can add as many as Widgets inside the ListView

```
drawer: Drawer(
child: ListView(
children: <Widget>[
ListTile(
title: Text("Ttem 1"),
trailing: Icon(Icons.arrow_forward),
),
ListTile(
title: Text("Item 2"),
trailing: Icon(Icons.arrow_forward),
),],),),
```

floatingActionButton: It is a button displayed at the bottom right corner and floating above the body. It is a circular icon button that floats over the content of a screen at a fixed place to promote a primary action in the application. While scrolling the page, its position cannot be changed.

```
appBar: AppBar(title: Text('First Flutter Application')),
body: Center(
  child: Text("Welcome to Javatpoint!!"),
),
floatingActionButton: FloatingActionButton(
  elevation: 8.0,
  child: Icon(Icons.add),
  onPressed: (){
    print('I am Floating Action Button');
  }
```

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

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

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

```
primary: true/false,
```

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

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

Session-10 Handouts

33. ListView

ListView is a list of scrolling widgets. which displays its children one after another in scroll direction. There are different types of ListViews:

ListView: The default behavior of ListView, normally use for display a small number of children. Use it to display a static list of children whose count don't change. Basic structure of this listview i

```
ListView(
      children: [
        child widget1,
        child widget2,
        .....
      ],
      scrollDirection: Axis.horizontal,
    )
```

Important properties of ListView are

padding:-

padding: EdgeInsets.all(10),

shrinkWrap:- It should be true to save memory wastage and increase app performance.

shrinkWrap: true

reverse:- Use this property to display the list in reverse order.

reverse: true

scrollDirection:- Use this property to change the scroll direction of the ListView. Default is Axis.vertical.

scrollDirection: Axis.horizontal,

itemExtent:- Use this property to extend (increase) the item in scroll direction. When the scroll direction is vertical it increases height and when vertical it increases the width of the item.

itemExtent: 100,

ListView can include ListTile widget for its child property for more structured information.

```
ListView(
 children: [
   ListTile(
     leading: Icon(Icons.map)
     //Or leading: CircleAvatar(backgroundImage: AssetImage('assets/img3.jpg')),
     title: Text('Apple'),
   ),
   ListTile(…),
   ListTile(…),
 ],
);
```

34. ListView.builder

Use this constructor to generate the ListView dynamically or with data from API (backend). One important property ListView.builder is **itemBuilde.** We use this property to generate the children for the listview. Another property **itemCount** us used to improve the ability of the ListView to estimate the maximum scroll extent.

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.

```
return ListView.builder(
 itemBuilder: (BuildContext context, int index) {
 return ListTile(
   title: Text('This is title'),
   subtitle: Text('This is subtitle'),
   trailing: Text('This is trailing'),
   leading: CircleAvatar(backgroundImage: AssetImage(imgList[index]),),

 );
},
 itemCount: imgList.length,
 shrinkWrap: true,
 padding: EdgeInsets.all(10),

);
```

35. ListView.Seperated

To display a listview with separators / dividers. It is almost similar to listView.builder() but it has an extra property **separatorBuilder**, which is used to build separator to the items of the listview.

```
separatorBuilder: (BuildContext context, int index) { return Divider(height: 5,); },
```

35. ListTile:

It can be use to display icons (in leading or trailing position) and text in a card. It contains one or three lines of text (title, subtitle) optionally flanked by icons or other widgets, such as check boxes.
List tiles are typically used in ListViews, or arranged in Columns in Drawers and Cards.

36. Themes

In order to share colors and font styles throughout our app, we can take advantage of themes. There are two ways to define themes:

- App-wide themes are created at the root of our apps by the MaterialApp for whole application.
- Theme Widgets are created for a particular part of our application

```
MaterialApp(
  theme: ThemeData()  //use to customize application theme.
  theme: ThemeLight() //default flutter application theme.
  theme: ThemeDark()  // use for dark black background application theme.
)
```

Properties

primaryColor: Colors.red ///change red color for top appbar
accentColor: Color.purple /// change purple color for

Colors

Color class is a an immutable 32 bit color value in ARGB format. The first letter 'A' is used for transparency.

Colors can be defined with following format

- Color(int value) = Color(0xff, 0x2e, 0x3a, 0xf5) where '0xff' for transparency, '0x2e' for red density, '0x3a' for green density, '0xf5' for blue density.
  Color(0xFFFFFF) //fully transparent white (invisible)
  Color(0xFFFFFFFF) //fully opaque white (visible)

- Color.fromARGB(255,65,165, 245)

- Color.fromRGBO(66, 165,245, 1.0)

Colors are mixture of Red Green Blue

Normally main page set themes, fonts etc for whole application and every separate screen is a separate dart file.

37. Flutter Gesture

The gesture system in Flutter has two separate layers. The first layer has raw pointer events that describe the location and movement of pointers (for example, touches, mice, and styli) across the screen. The second layer has *gestures* that describe semantic actions that consist of one or more pointer movements.

Pointers represent raw data about the user's interaction with the device's screen. There are four types of pointer events:

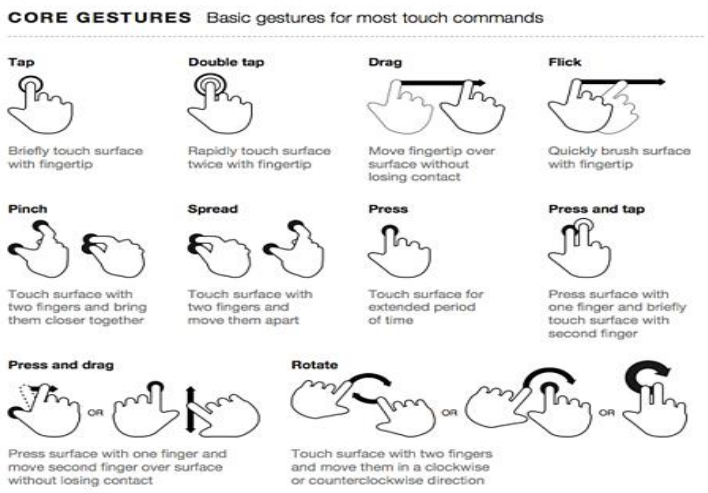
PointerDownEvent: The pointer has contacted the screen at a particular location.
PointerMoveEvent: The pointer has moved from one location on the screen to another.
PointerUpEvent: The pointer has stopped contacting the screen.
PointerCancelEvent: Input from this pointer is no longer directed towards this app.

On pointer down, the framework does a *hit test* on your app to determine which widget exists at the location where the pointer contacted the screen. The pointer down event (and subsequent events for that pointer) are then dispatched to the innermost widget found by the hit test. From there, the events bubble up the tree and are dispatched to all the widgets on the path from the innermost widget to the root of the tree. There is no mechanism for canceling or stopping pointer events from being dispatched further.

38. Gestures

Gestures represent semantic actions (for example, tap, drag, and scale) that are recognized from multiple individual pointer events, potentially even multiple individual pointers. Gestures can dispatch multiple events, corresponding to the lifecycle of the gesture (for example, drag start, drag update, and drag end):

Every app, game or tool you open on your phone must includes a swipe, tap or pinch to function. These gestures are the secret to making great mobile apps work. 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



Some of the widely used gestures are mentioned here:

Tap: Touching the surface of the device with fingertip for a short period.
Double Tap: Tapping twice in a short time.
Drag: Touching the surface of the device with fingertip and then moving the fingertip in a steady manner and then finally releasing the fingertip.
Flick: Similar to dragging, but doing it in a speeder way.
Pinch: Touch the surface of the device using two fingers and bring them closer together.
Spread/Zoom: Touch the surface with two fingers and move them in opposite direction.
Panning: Touching the surface of the device with fingertip and moving it in any direction without releasing the fingertip

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

Example: Let us modify the hello world application to include gesture detection feature and try to understand the concept.
Change the body content of the *MyHomePage* widget as shown below:

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

Session-12 Handouts

39. Animation

Animations are an essential part of making the UI of a mobile application feel natural and smooth to the user. Smooth transitions and interactive elements make an app pleasant to use. On the other hand, badly made animations can make the app look clumsy or worse, break the application altogether. For this reason, learning the fundamentals of animations in any framework is an essential step towards delivering a superior user experience.

Every component is either a Stateful or Stateless Widget. A Widget is just a component we can render on the screen. Every app consists of a widget tree which displays the hierarchy of components inside the app.

Animation is a process of showing a series of images / picture in a particular order within a specific duration to give an illusion of movement. The most important aspects of the animation are

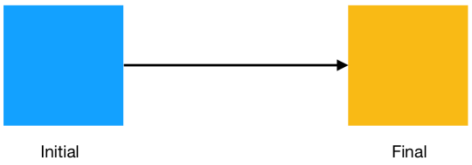
Animation have two distinct values: Start value and End value.

The ability to control the animation process like starting the animation, stopping the animation, repeating the animation to set number of times, reversing the process of animation, etc.

Every Flutter animation needs at least two elements to be created:

1. A Tween to get generate values for the animation
2. An AnimationController as parent

Tween: It is the short form of **in-betweening**. In a tween animation, it is required to define the **start** and **endpoint** of animation. It also provides the timeline and curve, which defines the time and speed of the transition.



We have a box with a blue background initially and we need to animate it to change the color to yellow. Now, an abrupt color change looks terrible to the user. The change has to be smooth. However, it is impossible for us to show all colors that lie between. In such cases, we create a ColorTween, which gives us all the values in between (between -> tween) blue and yellow so that we can display them.

Simply put, a Tween gives us intermediate values between two values like colors, integers, alignments and almost anything you can think of.

ColorTween {

begin: color.blue,

end: color.orange,

}

39.1. Animation Controllers

An animation controller is, as the name suggests, a way to control (trigger, fling or stop) an animation.

However, the main function of the controller is to drive the animation. This means it will cause the animation to change its value and return a new value from the Tween based on the progression of the Animation.

The AnimationController also gives us control over how the animation behaves. For example, we can repeat an animation using controller.repeat()

To start an animation, we can use the controller as *controller.forward()* and to reverse, *controller.reverse() and controller.fling()* which flings (runs an animation quickly regardless of duration) the animation.

we use the setState() function to rebuild the box and controller.addListener() to listen the update.

```
//In initState
controller.addListener((){
  setState((){});
});
```

Here is an example code

```
class _MyHomePageState extends
State<MyHomePage> with
SingleTickerProviderStateMixin{
  AnimationController controller;
  Animation colorAnimation;
  Animation sizeAnimation;

  @override
  void initState() {
    super.initState();

    // Defining controller with
animation duration of two seconds
    controller =
AnimationController(vsync: this,
duration: Duration(seconds: 2));

    // Defining both color and size
animations
    colorAnimation =
ColorTween(begin: Colors.blue,
end:
Colors.yellow).animate(controller);
    sizeAnimation =
Tween<double>(begin: 100.0, end:
200.0).animate(controller);

    // Rebuilding the screen when
animation goes ahead
    controller.addListener(() {
      setState(() {});
    });

    // Repeat the animation after
finish
    controller.repeat();

    //For single time
```

```
    //controller.forward()



  //Reverses the animation instead
of starting it again and repeats
    //controller.repeat(reverse:
true);
  }

  @override
  Widget build(BuildContext
context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Animation
Demo"),
      ),
      body: Center(
        child: Container(
          height: sizeAnimation.value,
          width: sizeAnimation.value,
          color: colorAnimation.value,
        ),
      ),
    );
  }
}
```

40. Navigation and Routes

Every mobile application contains several screens and pages and each screen/pages known as routes (Activity in Android, and ViewController in iOS).

Flutter provides a basic routing class **MaterialPageRoute** and two methods **Navigator.push()** and **Navigator.pop()** that shows how to navigate between two routes. To perform this task,

- Navigator.push() is used to navigate from first route to second route

```
TextButton(onPressed: () {
  Navigator.push(context, MaterialPageRoute(builder: (context)=> sroutApp()));
},
```

- Navigator.pop() is used to return to the first route.

```
TextButton(onPressed: () {
  Navigator.pop(context);
},
```

40.1.1. Navigation With Named routes

The Navigator maintains the stack-based history of routes. If there is a need to navigate to the same screen in many parts of the app or we want to move from first screen to third and from third to fifth. We can work with named routes by using the Navigator.pushNamed() function.

First we have to define the routes as

```
initialRoute: '/'       //here it tells the start of the page
routes: {
'/': (context) => HomeScreen()
'/second': (context) => SecondScreen()
},
```

Now define route on pressed

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

Now use the Navigator.pop() method to return on the first screen.

```
onPressed: () {
  Navigator.pushNamed(context, '/second');
},
```

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

1. Create two screens.
2. Define the routes.
3. Navigate to the second screen using Navigator.pushNamed().
4. Return to the first screen using Navigator.pop().

40.2. Create two screens

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

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

class FirstScreen extends StatelessWidget {
  const FirstScreen({Key? key}) : super(key: key);

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

```
    );
  }
}

class SecondScreen extends StatelessWidget {
  const SecondScreen({Key? key}) : super(key: key);

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

40.3. Define the routes

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

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

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

40.4. Navigate to the second screen

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

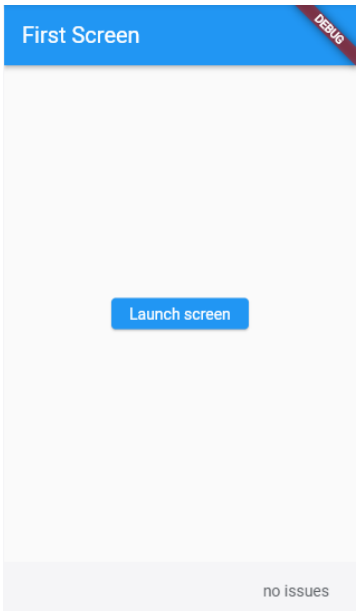
In the build() method of the FirstScreen widget, update the onPressed() callback:

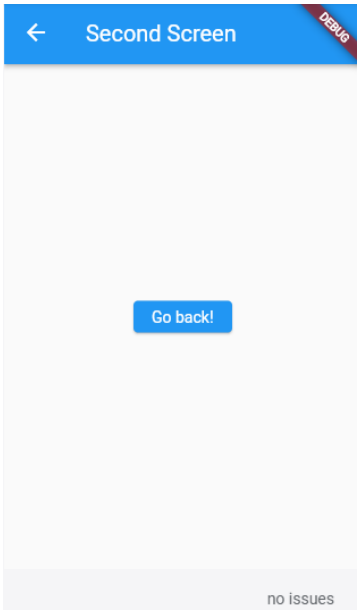
```
// Within the `FirstScreen` widget
onPressed: () {
  // Navigate to the second screen using a named route.
  Navigator.pushNamed(context, '/second');
}
```

40.5 Return to the first screen

To navigate back to the first screen, use the Navigator.pop() function.

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





Session-14 Handouts

41. Themes

In order to share colors and font styles throughout our app, we can take advantage of themes. There are two ways to define themes:

- App-wide themes are created at the root of our apps by the MaterialApp for whole application.
- Theme Widgets are created for a particular part of our application

```
MaterialApp(
  theme: ThemeData()  //use to customize application theme.
  theme: ThemeLight() //default flutter application theme.
  theme: ThemeDark()  // use for dark black background application theme.
)
```

Properties

primaryColor: Colors.red ///change red color for top appbar
accentColor: Color.purple /// change purple color for

Colors

Color class is a an immutable 32 bit color value in ARGB format. The first letter 'A' is used for transparency.

Colors can be defined with following format

- Color(int value) = Color(0xff, 0x2e, 0x3a, 0xf5) where '0xff' for transparency, '0x2e' for red density, '0x3a' for green density, '0xf5' for blue density.
  Color(0xFFFFFF) //fully transparent white (invisible)
  Color(0xFFFFFFFF) //fully opaque white (visible)

- Color.fromARGB(255,65,165, 245)

- Color.fromRGBO(66, 165,245, 1.0)

Colors are mixture of Red Green Blue

Normally main page set themes, fonts etc for whole application and every separate screen is a separate dart file.

42. APIs

Fetching data from the internet is necessary for most apps. Dart and Flutter provides **http package** to use http resources.

This package contains a set of high-level functions and classes that make it easy to consume HTTP resources.

The http package uses **await** and **async** features and provides many high-level methods such as read, get, post, put, head, and delete methods for sending and receiving data from remote locations.

The http package also provides a standard **http client class** that supports the persistent connection. This class is useful when a lot of requests to be made on a particular server. It should be closed properly using the **close**() method.

The core methods of the http package are as follows:

- **Read:** This method is used to read or retrieve the representation of resources. I
- **Get:** This method requests the specified url from the get method and returns a response as Future<response>.
- **Post:** This method is used to submit the data to the specified resources.
- **Put:** This method is utilized for update capabilities.
- **Head:** It is similar to the Get method, but without the response body.
- **Delete:** This method is used to remove all the specified resources.

API recipe uses the following steps:

1. Add the http package.
2. Make a network request using the http package.
3. Convert the response into a custom Dart object.
4. Fetch and display the data with Flutter.

To install the http package, add it to the dependencies section of the pubspec.yaml file. You can find the latest version of the http package the pub.dev.

```
dependencies:

        http: ^0.13.4
```

Import the http package.

```
import 'package:http/http.dart' as http;
```

Additionally, in your AndroidManifest.xml file, add the Internet permission.

```
<uses-permission android:name="android.permission.INTERNET" />
```

```
Future<http.Response> fetchWeather() async {

 return await http.get(Uri.parse(' api.openweathermap.org/data/2.5/weather?q={city name}&appid={API_key}'));

}
```

Future is a core Dart class for working with async operations.

The http.get() method returns a Future that contains a Response.

The http.Response class contains the data received from a successful http call.

HTTP response codes

- Informational responses (100–199)
- Successful responses (200–299)
- Redirection messages (300–399)
- Client error responses (400–499)
- Server error responses (500–599)

```
Future<Post> fetchPost() async {

 final response = await http.get( Give the link of JSON file');
```

```
 if (response.statusCode == 200) {
   // If the server returns an OK response, then parse the JSON.
   return Post.fromJson(json.decode(response.body));
 } else {
   // If the response was umexpected, throw an error.
   throw Exception('Failed to load post');
 }
}
```

Finally, display the data. You can display the data by using the FutureBuilder widget. This widget can work easily with async data sources.

```
FutureBuilder<Post>(
 future: post,
 builder: (context, abc) {
   if (abc.hasData) {
     return Text(abc.data.title);
   } else if (abc.hasError) {
     return Text("${abc.error}");
   }
       return CircularProgressIndicator();   // By default, it show a loading spinner.
 },  );
```

43. Navigation and Routes

Every mobile application contains several screens and pages and each screen/pages known as routes (Activity in Android, and ViewController in iOS).

Flutter provides a basic routing class **MaterialPageRoute** and two methods **Navigator.push()** and **Navigator.pop()** that shows how to navigate between two routes. To perform this task,

- Navigator.push() is used to navigate from first route to second route

```
TextButton(onPressed: () {
  Navigator.push(context, MaterialPageRoute(builder: (context)=> sroutApp()));
},
```

- Navigator.pop() is used to return to the first route.

```
TextButton(onPressed: () {
  Navigator.pop(context);
},
```

43.1.1. Navigation With Named routes

The Navigator maintains the stack-based history of routes. If there is a need to navigate to the same screen in many parts of the app or we want to move from first screen to third and from third to fifth. We can work with named routes by using the Navigator.pushNamed() function.

First we have to define the routes as

```
initialRoute: '/'       //here it tells the start of the page
routes: {
'/': (context) => HomeScreen()
'/second': (context) => SecondScreen()
},
```

Now define route on pressed

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

Now use the Navigator.pop() method to return on the first screen.

```
onPressed: () {
  Navigator.pushNamed(context, '/second');
},
```

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

5. Create two screens.
6. Define the routes.
7. Navigate to the second screen using Navigator.pushNamed().
8. Return to the first screen using Navigator.pop().

43.2. Create two screens

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

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

class FirstScreen extends StatelessWidget {
  const FirstScreen({Key? key}) : super(key: key);

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

```
    );
  }
}

class SecondScreen extends StatelessWidget {
  const SecondScreen({Key? key}) : super(key: key);

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

43.3. Define the routes

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

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

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

43.4. Navigate to the second screen

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

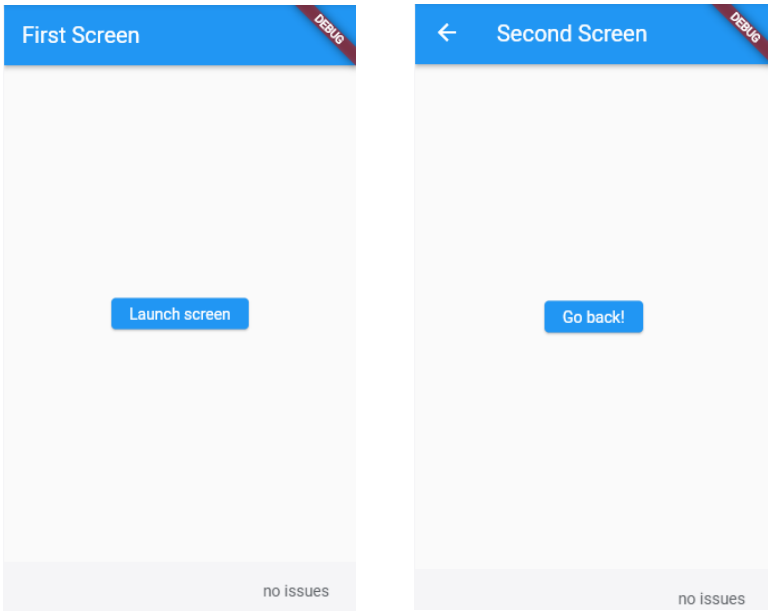
In the build() method of the FirstScreen widget, update the onPressed() callback:

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

43.5. Return to the first screen

To navigate back to the first screen, use the Navigator.pop() function.

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



44. Themes

In order to share colors and font styles throughout our app, we can take advantage of themes. There are two ways to define themes:

- App-wide themes are created at the root of our apps by the MaterialApp for whole application.
- Theme Widgets are created for a particular part of our application

MaterialApp(
theme: ThemeData() //use to customize application theme.
theme: ThemeLight() //default flutter application theme.
theme: ThemeDark() // use for dark black background application theme.
)

Properties

primaryColor: Colors.red ///change red color for top appbar
accentColor: Color.purple /// change purple color for

Colors

Color class is a an immutable 32 bit color value in ARGB format. The first letter 'A' is used for transparency.

Colors can be defined with following format

- Color(int value) = Color(0xff, 0x2e, 0x3a, 0xf5) where '0xff' for transparency, '0x2e' for red density, '0x3a' for green density, '0xf5' for blue density.
  Color(0xFFFFFF) //fully transparent white (invisible)
  Color(0xFFFFFFFF) //fully opaque white (visible)

- Color.fromARGB(255,65,165, 245)

- Color.fromRGBO(66, 165,245, 1.0)

Colors are mixture of Red Green Blue

Normally main page set themes, fonts etc for whole application and every separate screen is a separate dart file.

Session-16 Handouts

45. APIs

Fetching data from the internet is necessary for most apps. Dart and Flutter provides **http package** to use http resources.

This package contains a set of high-level functions and classes that make it easy to consume HTTP resources.

The http package uses **await** and **async** features and provides many high-level methods such as read, get, post, put, head, and delete methods for sending and receiving data from remote locations.

The http package also provides a standard **http client class** that supports the persistent connection. This class is useful when a lot of requests to be made on a particular server. It should be closed properly using the **close**() method.

The core methods of the http package are as follows:

- **Read:** This method is used to read or retrieve the representation of resources. I
- **Get:** This method requests the specified url from the get method and returns a response as Future<response>.
- **Post:** This method is used to submit the data to the specified resources.
- **Put:** This method is utilized for update capabilities.
- **Head:** It is similar to the Get method, but without the response body.
- **Delete:** This method is used to remove all the specified resources.

API recipe uses the following steps:

5. Add the http package.
6. Make a network request using the http package.
7. Convert the response into a custom Dart object.
8. Fetch and display the data with Flutter.

To install the http package, add it to the dependencies section of the pubspec.yaml file. You can find the latest version of the http package the pub.dev.

```
dependencies:

        http: ^0.13.4
```

Import the http package.

```
import 'package:http/http.dart' as http;
```

Additionally, in your AndroidManifest.xml file, add the Internet permission.

```
<uses-permission android:name="android.permission.INTERNET" />
```

```
Future<http.Response> fetchWeather() async {

 return await http.get(Uri.parse(' api.openweathermap.org/data/2.5/weather?q={city name}&appid={API_key}'));

}
```

Future is a core Dart class for working with async operations.

The http.get() method returns a Future that contains a Response.

The http.Response class contains the data received from a successful http call.

HTTP response codes

- Informational responses (100–199)
- Successful responses (200–299)
- Redirection messages (300–399)
- Client error responses (400–499)
- Server error responses (500–599)

```
Future<Post> fetchPost() async {
  final response = await http.get( Give the link of JSON file');


  if (response.statusCode == 200) {
    // If the server returns an OK response, then parse the JSON.
    return Post.fromJson(json.decode(response.body));
  } else {
    // If the response was umexpected, throw an error.
    throw Exception('Failed to load post');
  }
}
```

Finally, display the data. You can display the data by using the FutureBuilder widget. This widget can work easily with async data sources.

```
FutureBuilder<Post>(
  future: post,
  builder: (context, abc) {
    if (abc.hasData) {
      return Text(abc.data.title);
    } else if (abc.hasError) {
      return Text("${abc.error}");
    }
       return CircularProgressIndicator();   // By default, it show a loading spinner.
  },  );
```

Supporting Material

1. https://flutter.dev/?gclid=Cj0KCQjwjN-SBhCkARIsACsrBz6_QN0tKEx3gl8HLbGL86TXs4aawamiWkGSJMlg7tBDhyaP4Skli70aAmyMEALw_wcB&gclsrc=aw.ds
2. https://docs.flutter.dev/
3. https://docs.flutter.dev/resources/books
4. https://dart.dev/guides/language/effective-dart/documentation
5. https://www.freecodecamp.org/news/how-to-learn-flutter-in-2020/
6. https://www.youtube.com/watch?v=p4xh7zTt6i0
7. https://www.youtube.com/watch?v=x0uinJvhNxI
8. https://www.youtube.com/watch?v=aiTTClKJbnw