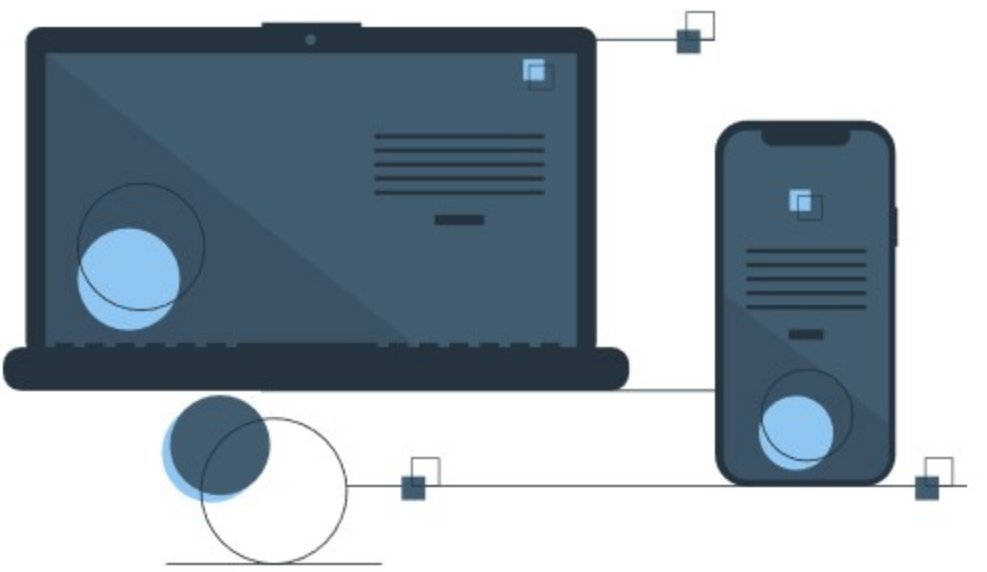
Application Development Using Flutter and Dart



Application Development Using Flutter and Dart Learner's Guide

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2022

Aptech®
Unleash your potential

Aptech®
COMPUTER EDUCATION
Unleash your potential

Onlinevarsity

Preface

Flutter framework is an open-source platform by Google that allows building native cross-platform (such as for Android or iOS devices) applications with one programming language (Dart) and one codebase. This book introduces Flutter and covers platform-specific Widgets, platform-independent Widgets, Single Child Layout Widgets, Multiple Child Layout Widgets, and so on. The book also explores state management, dependency injection, navigation, routing, and controllers in Flutter. Finally, the book concludes with an explanation on database concepts with Sqflite and Firebase and REST API.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

MANY COURSES ONE PLATFORM

Onlinevarsity

Onlinevarsity App

for Android devices

Download from Google Play Store

Table of Contents

Sessions

Session 1
Introduction to Flutter

Learning Objectives

In this session, students will learn to:

- Define Flutter framework
- List and describe features of Flutter
- List the advantages of using Flutter
- Outline the Flutter architecture
- Identify components of Flutter
- Explain installation of Flutter in Android Studio

Session Goals:

The session introduces the Flutter framework and explain its features. Further, the session lists the advantages of creating an application with Flutter. The session finally concludes by elaborating the steps involved in installing Flutter in Android studio.

1.1 What is Flutter?

In the application development world, one can develop applications using various platforms. Android and iOS are examples of platforms for which wireless or mobile applications can be developed. Until recently, when a company required to build mobile applications targeting multiple platforms, two categories of developers were hired. One specifically for Android and the other for iOS. This not only increased the time taken for developing the app, but also increased the cost of pushing updates, overhead costs, and so on.

What is the solution now?
Flutter, React Native, and Ionic are frameworks have solved this problem of having to develop multiple applications for several platforms by using cross-platform development approach. In this approach, applications are developed that can be used across platforms.

What is Flutter?

A 'tool' that allows building native cross-platform (for Android or iOS devices) applications with one programming language and codebase. Figure 1.1 shows how Flutter can be utilized for developing applications across several platforms.

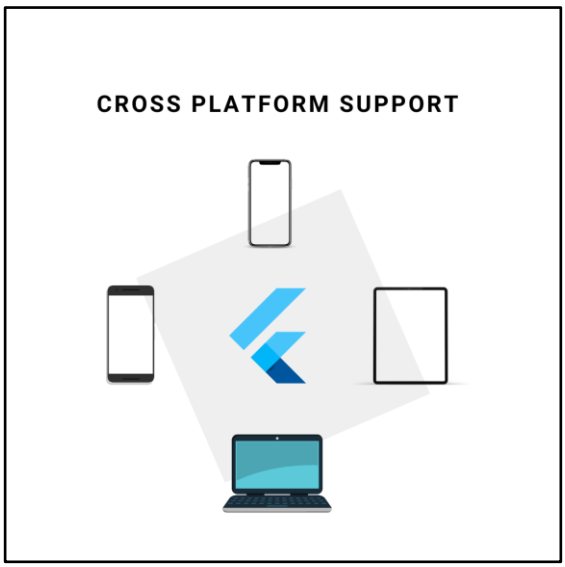


Figure 1.1: Cross-Platform Apps Using Flutter

Flutter uses Dart which is focused on frontend, that is, User Interface (UI) development. Dart is developed by Google and is an Object-Oriented programming language. The syntax for Dart is a mixture of JavaScript, Java, and C#.

1.2 Features and Advantages of Flutter

Following are the features and advantages of Flutter:

- **Hot reload**: Hot reload makes it possible to see changes in the code immediately by making it visible in the UI. This speeds up the process of working on the application concept; moreover, it enables developers to fix costs that save money and effort.

- **Cross-platform development:** Flutter enables developers to write code that works on a variety of platforms. Two separate applications can use the same codebase. In addition to sharing the UI code, the UI can itself be shared. This makes maintaining one codebase easier when compared to maintaining several codes for several domains.
- **Widget Library:** Everything in Flutter is defined as a widget. The widget can be anything such as a padding, color, or menu. Flutter is able to create complex customizable widgets depending on application requirements. Other examples include the Cupertino package and Material Design with widget sets that give a seamless user experience.
- **Native performance:** In Flutter, Google Fuchsia, iOS, and Android provide field-specific widgets. Integration of these widgets into the Flutter app for several location-based functions is possible. Existing Swift, Java, and Objective-C codes can be made to use native features such as camera and geolocation. Hence, Flutter can easily have third-party integrations with APIs.
- **Open Source:** Flutter is an open-source platform by Google. To create Flutter applications, several design options can be explored by developers. Applications that are easy-to-use can be created with the help of Cupertino widgets and Material Design. Flutter Form is a community of Flutter lovers who come together to answer Flutter-related questions and discuss them. Flutter is completely free and there are detailed tutorials and communities available online.

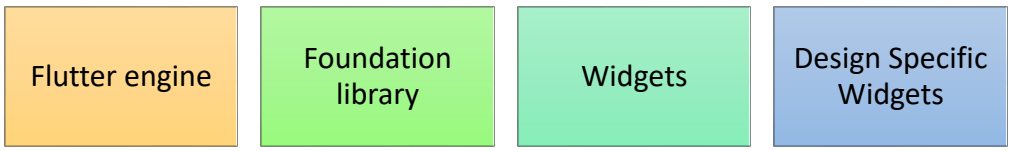
Using Flutter for Application Development is beneficial because of following reasons:

One Code:	With one code, developers can target Android and iOS-based devices, Desktops, and Web.
Simple:	Flutter is very easy to learn and use.
Easy Testing:	App Testing is very easy in Flutter because of Flutter unit testing functionality.
UI Library:	Flutter framework offers inbuilt UI functions which enhances the UI/UX.

Custom Interface:	Flutter has in-built functions that helps us to create Custom Interface.
Scalability:	Flutter scales well that helps in Future Scope.

1.3 Architecture and Components of Flutter

Flutter framework broadly comprises four components:



The Foundation library contains the required packages for writing full-fledged Flutter applications. Since the Flutter engine being Portable, it can be utilized for high-quality mobile apps. Flutter UI is composed of Widgets which are classified as per design languages of Google and iOS. They are Material Design and Cupertino Style. Figure 1.2 shows the architecture of Flutter in detail.

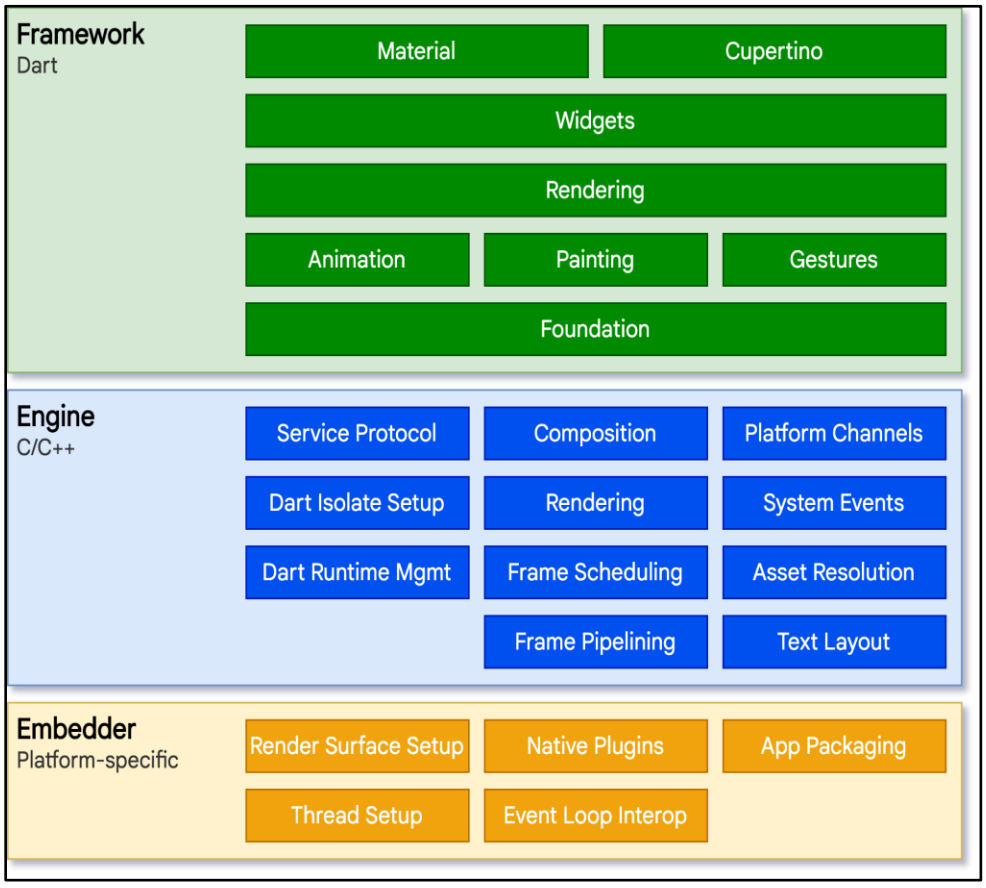


Figure 1.2: Detailed Architecture of Flutter

1.4 Installation of Flutter in Android Studio

Before setting up Flutter, one must ensure some system requirements are in place specific to Operating Systems.

Following is the link to know more about the system requirements: https://docs.flutter.dev/get-started/install

Once the requirements are met, developer has to download the Flutter SDK using following link:
Windows: https://docs.flutter.dev/get-started/install/windows

After downloading the SDK for Flutter, the next step is to set the SDK Path which is automatically done while installing the SDK for Flutter.

Then, set up an editor to install and run Flutter Apps.

Flutter apps can be built with any text editor that has command-line tools.

However, officially, the Flutter team suggests using following editors:

Following are the links to get started with each of these in Windows:
https://docs.flutter.dev/get-started/editor?tab=androidstudio
https://docs.flutter.dev/get-started/editor?tab=vscode
https://docs.flutter.dev/get-started/editor?tab=emacs

Android Studio is one of the widely used IDEs for developing Flutter applications.

Install Android Studio – Windows

Once the system requirements are met and Flutter SDK is set up, developer has to follow the steps provided.

Step 1: Download Android Studio using following link: https://developer.android.com/studio. Refer to Figure 1.3.

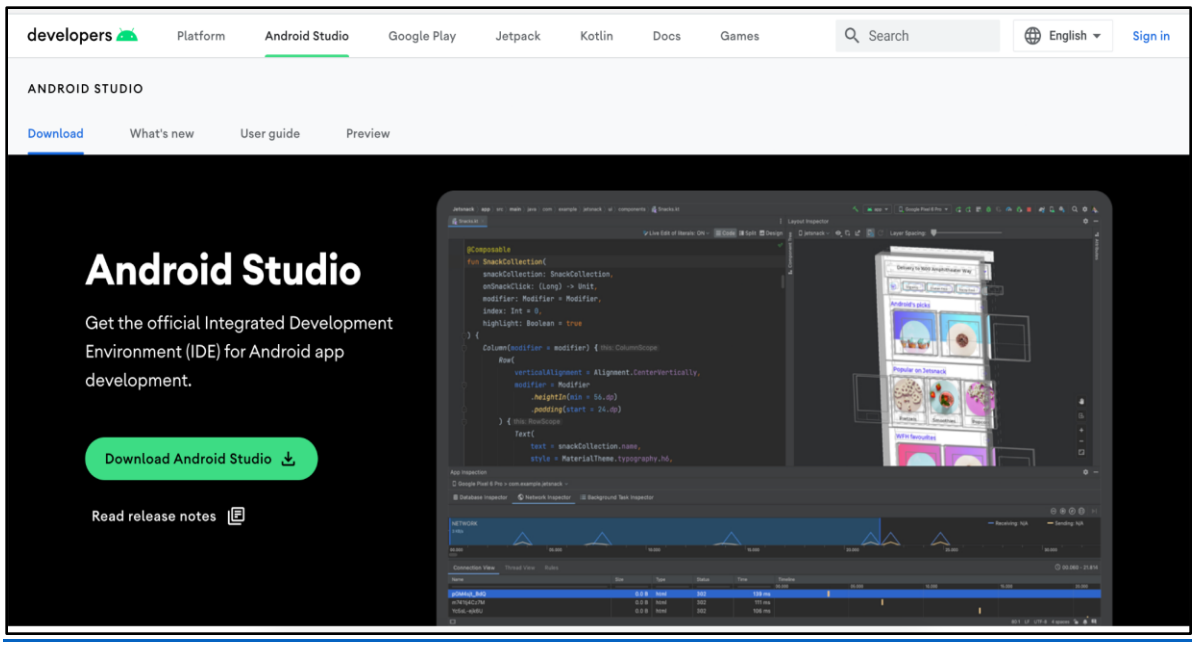


Figure 1.3: Selecting Android Studio for Download

Step 2: Accept the license and click Download. Wait for the file to download. Refer to Figure 1.4.

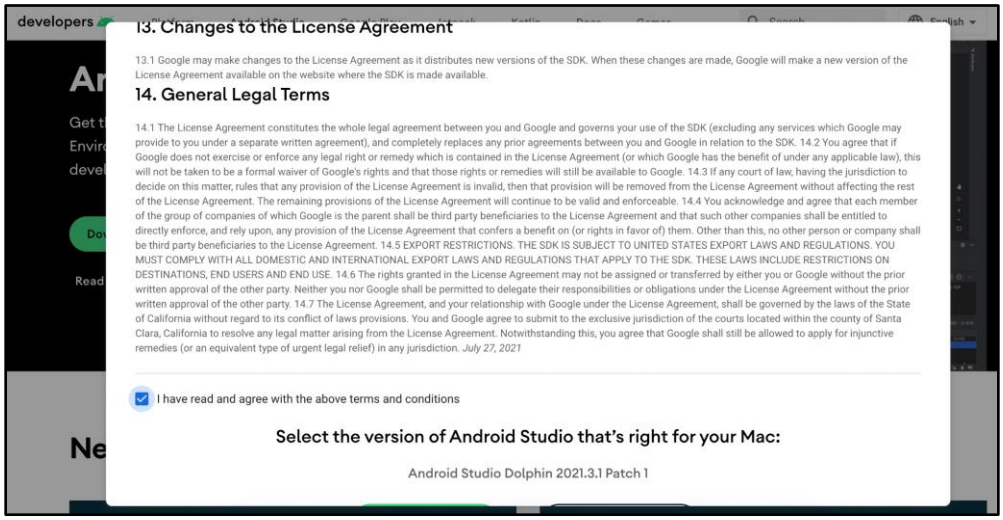


Figure 1.4: Downloading Android Studio

Step 3: Go to the location where Android Studio is downloaded. Figure 1.5 shows the downloaded file under the Downloads folder.

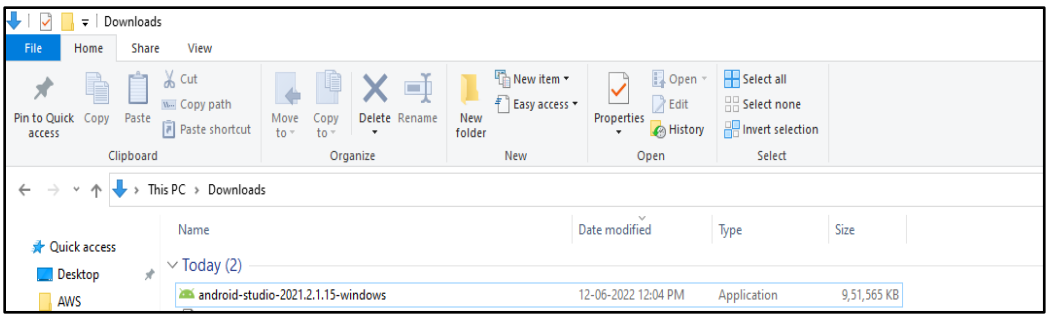


Figure 1.5: Android Studio Download File

Step 4: Double-click the exe file. Wait for the launcher to show up and click 'Yes' to reveal the Android Studio Setup screen. Figure 1.6 shows the Android Studio Setup screen.

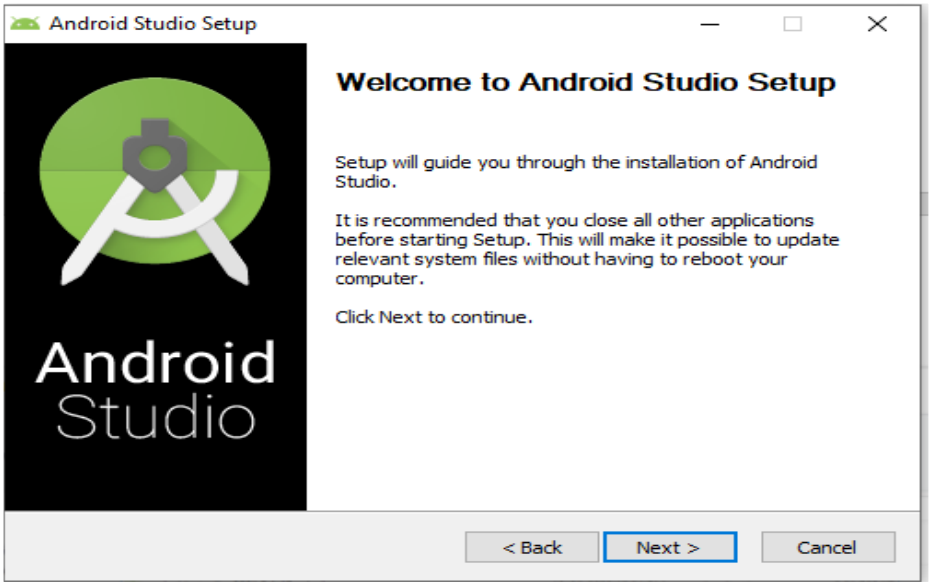


Figure 1.6: Android Studio Setup Screen

Step 5: Click Next. Once done, the setup wizard will ask to choose components. Select the default options that are pre-checked and click Next. Figure 1.7 shows the screen used for selecting components

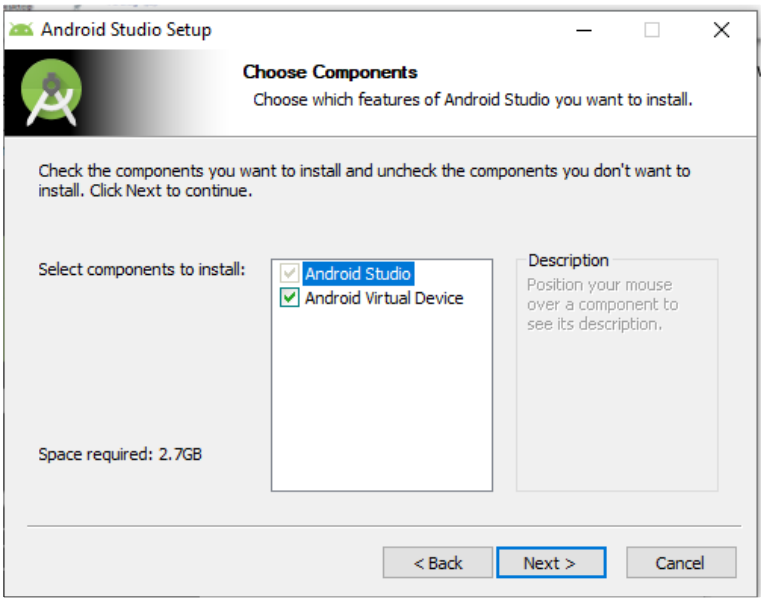


Figure 1.7: Android Studio Setup – Choosing Components

Step 6: The subsequent step is to choose the location where Android Studio is to be installed. Figure 1.8 shows the path where Android Studio is going to be installed.

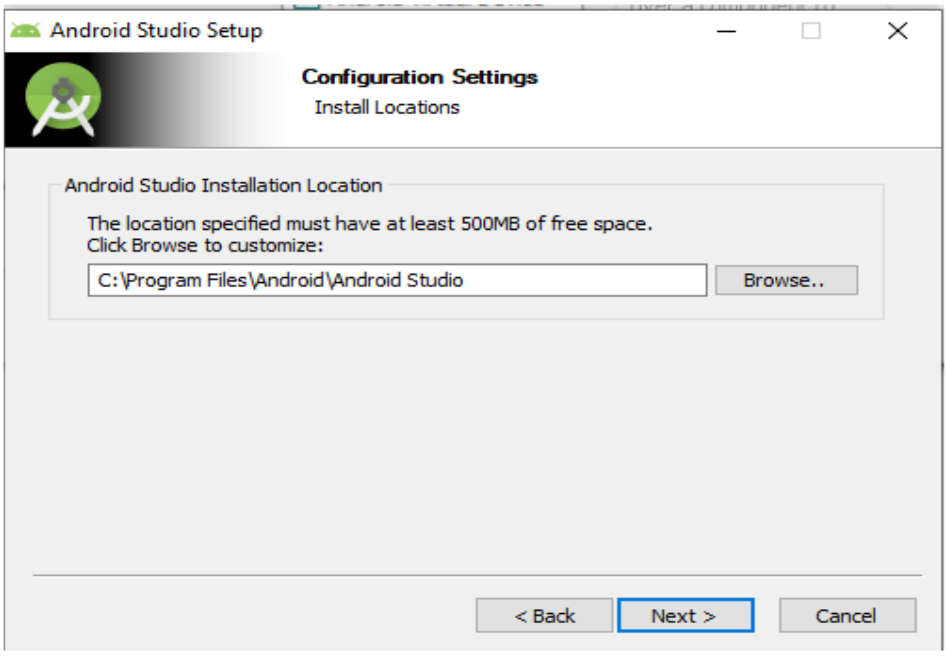


Figure 1.8: Location Settings for Android Studio

Step 7: After selecting the right drive, choose the Start Menu Folder or shortcut and click Install. Figure 1.9 shows the 'Start Menu Folder' selection menu.

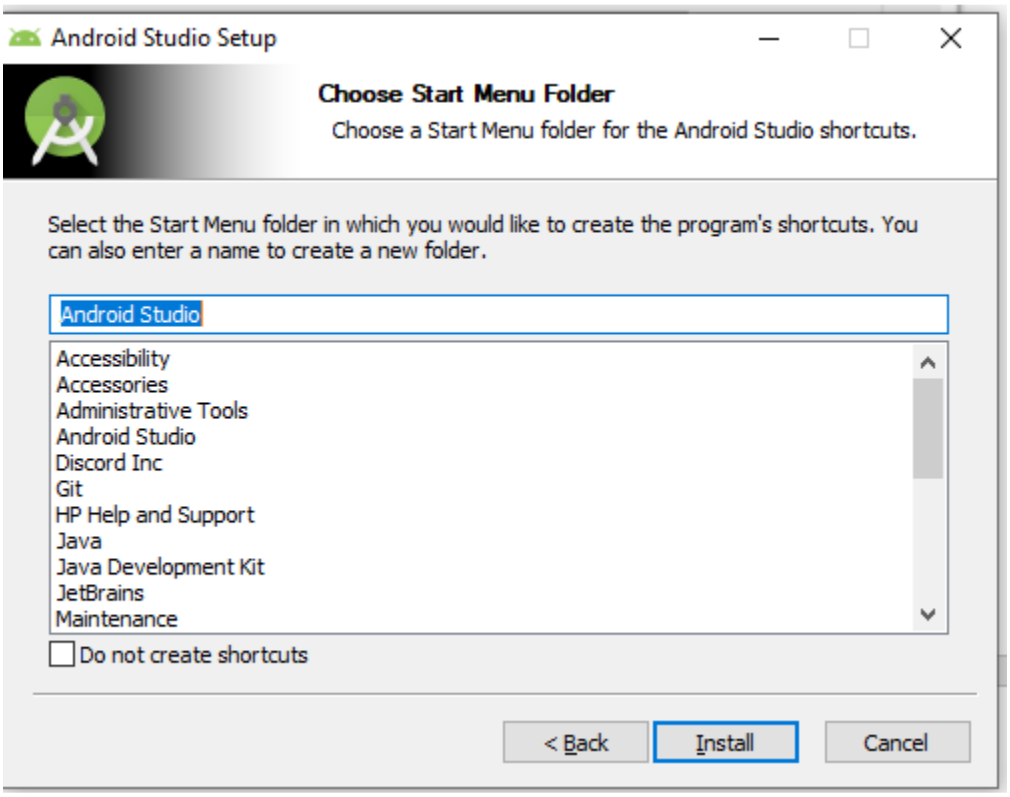


Figure 1.9: Start Menu Folder Selection Menu

Step 8: Once installation is completed, click Next to finish. Figure 1.10 shows final installation screen.

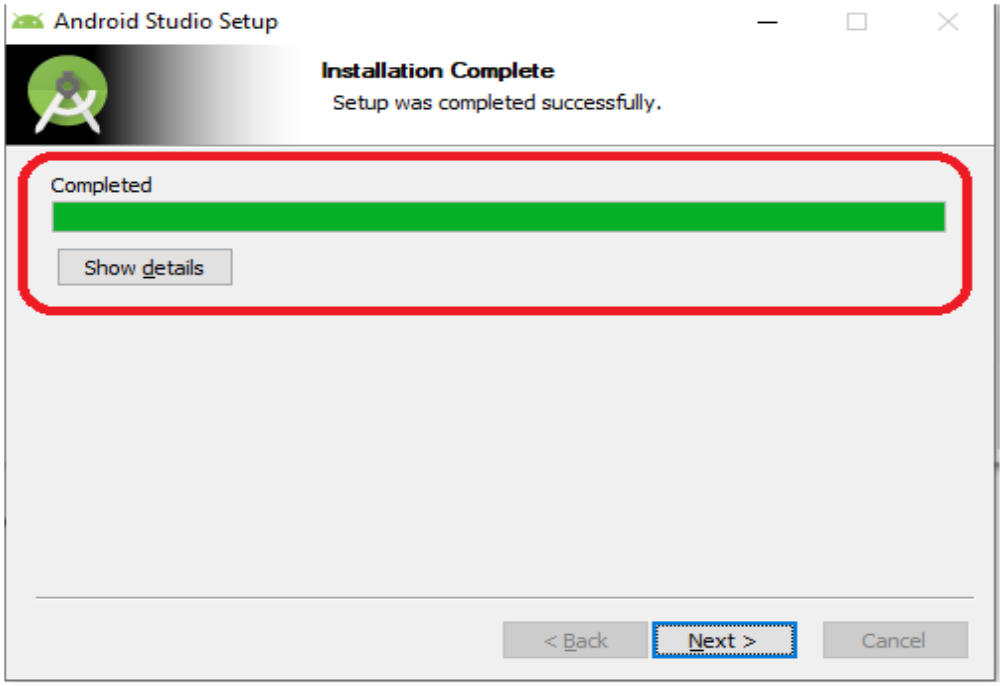


Figure 1.10: Final Installation Screen

Step 9: Finally, click Finish and start Android Studio. Refer to Figure 1.11.

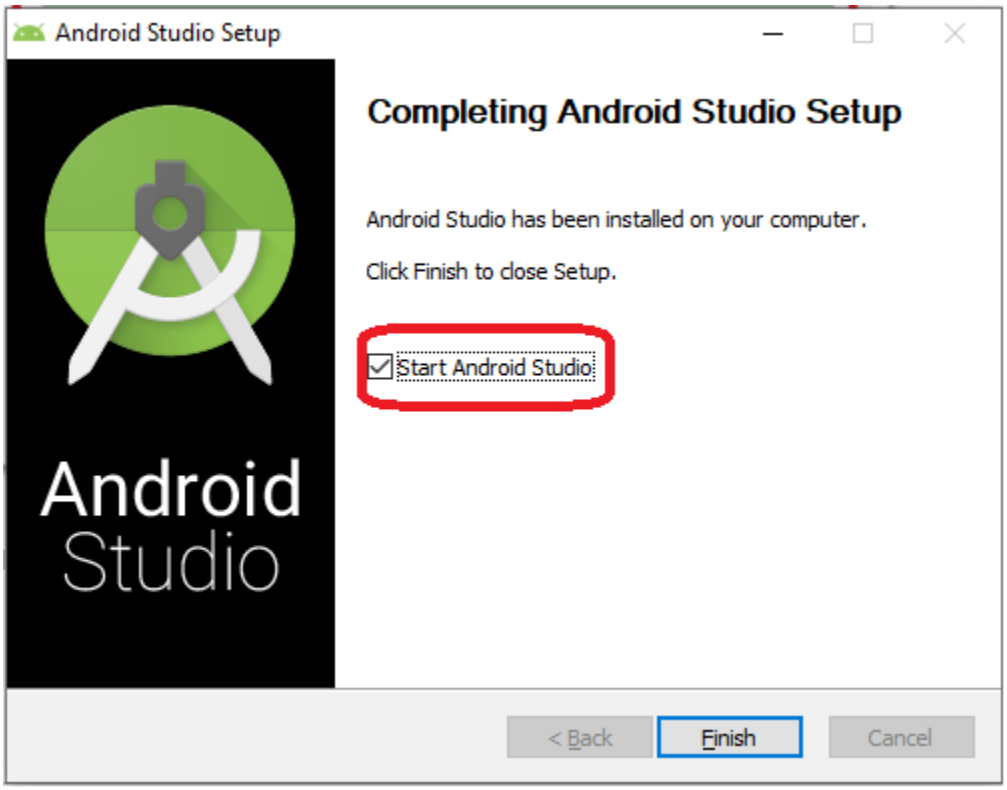


Figure 1.11: Launch Android Studio Screen

Step 10: Once the installation process is complete, set up SDK Manager. Figure 1.12 shows how to navigate to Android Studio SDK Manager.

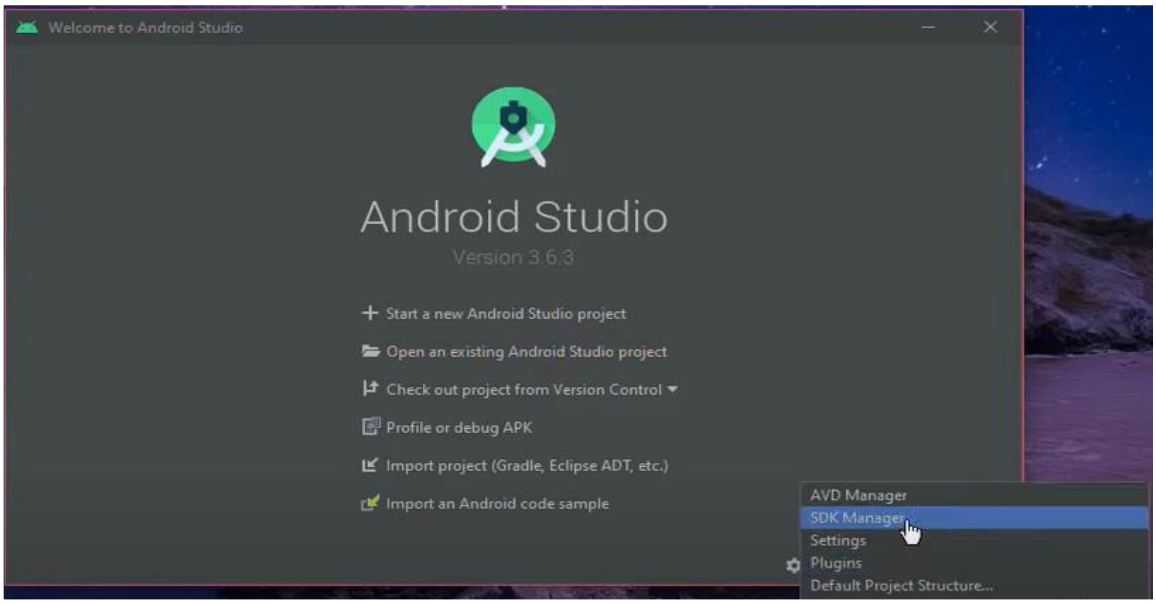


Figure 1.12: Android Studio SDK Manager

Step 11: Once done, install Plugins – Flutter and Dart. Figure 1.13 shows where to select the plugin option and Figure 1.14 shows the marketplace from where the required plugins can be downloaded.

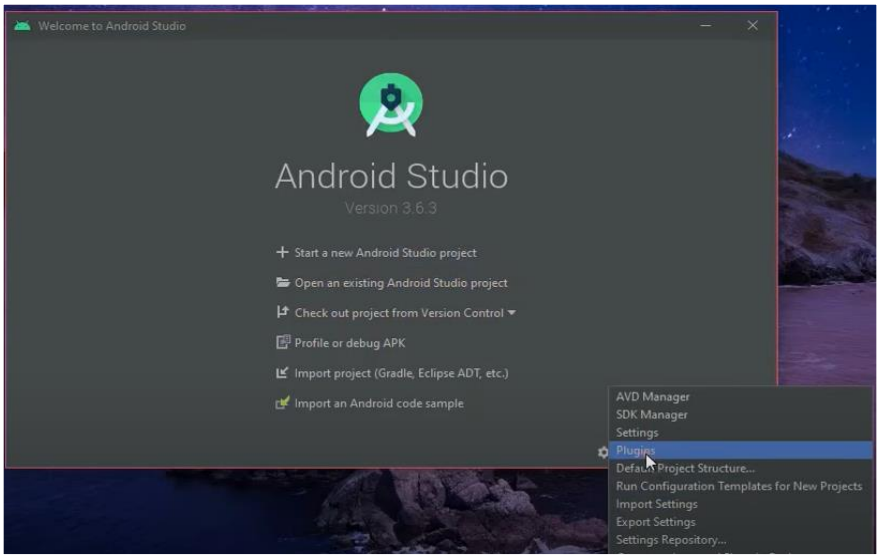


Figure 1.13: Plugin Option

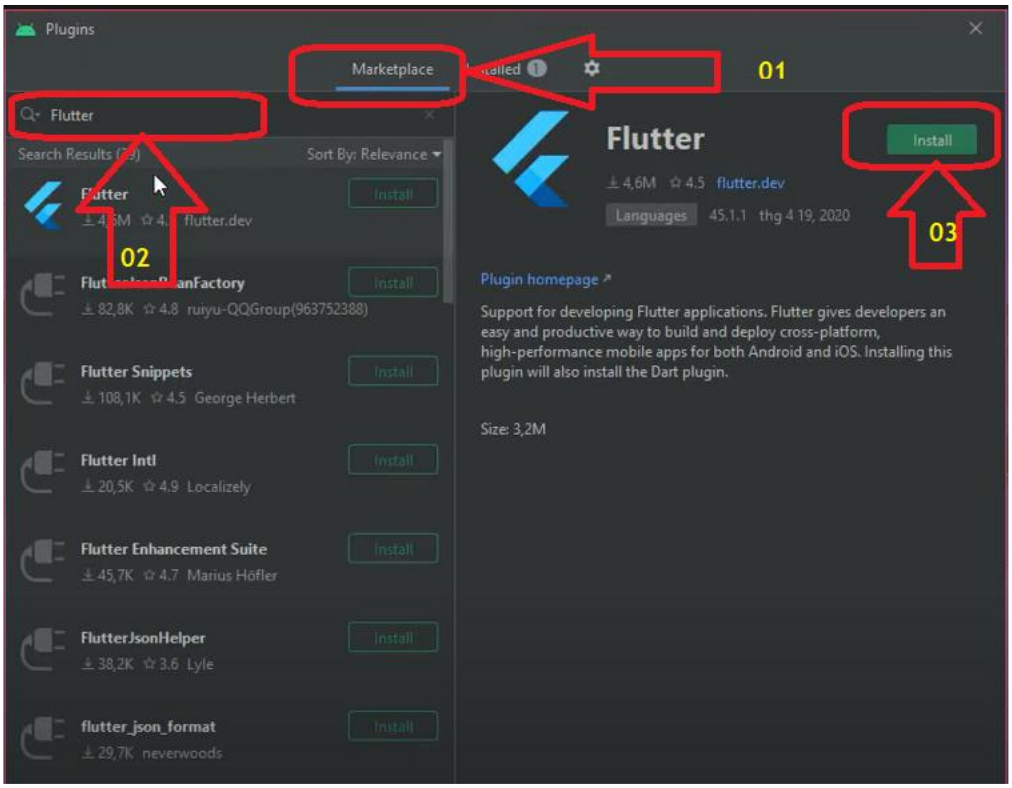


Figure 1.14: Plugin Marketplace

Step 12: Once plugins are downloaded, press Restart IDE to re-launch the IDE. Figure 1.15 shows how to restart the IDE once the plugin is downloaded.

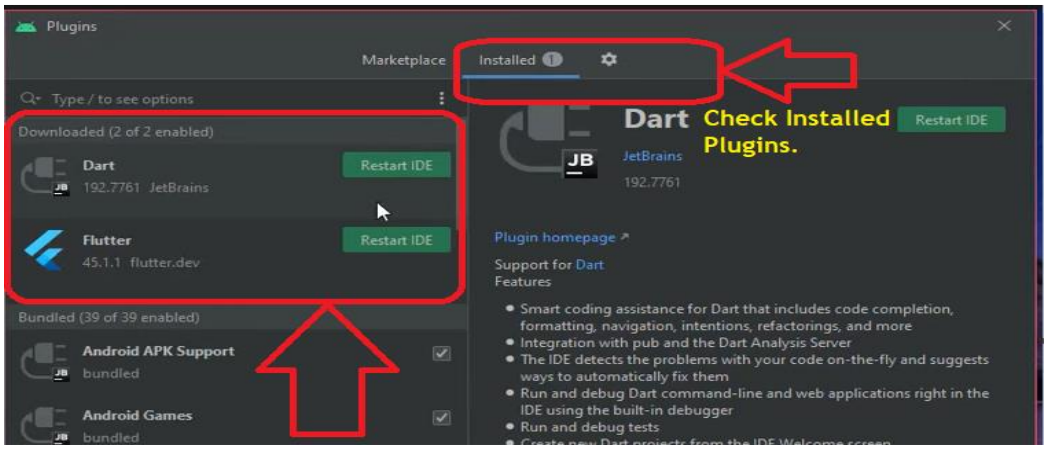


Figure 1.15: Restarting IDE

Now. Flutter is ready to be used in a project/application. Use **File → New → New Flutter Project** to create a new project.

1.5 Summary

- Flutter is a UI toolkit and framework for developing cross-platform applications.
- Advantages of Flutter include - Hot Reload, Hot Restart, Open Source, and Cross-Platform.
- Architectural components of Flutter are - Framework, Engine, and Platform.
- Flutter can be installed on any Operating System.
- Flutter apps can be developed using various editors or IDEs such as VS Code Editor or Android Studio.

1.6 Test Your Knowledge

1. What is Flutter?
   a. UI SDK
   b. Programming Language
   c. Tool
   d. Library

2. Is Flutter open-source?
   a. True
   b. False

3. Feature of Flutter does not include -
   a. Hot Reload
   b. Hot Restart
   c. Widget Library
   d. Multi-threading

4. For which platforms can Flutter be used to produce applications?
   a. Android
   b. iOS
   c. Website
   d. All of these

5. Which company developed Flutter?
   a. Samsung
   b. Amazon
   c. Meta
   d. Google

Answers - Test Your Knowledge

1. UI SDK
2. True
3. Multi-Threading
4. All of these
5. Google

1.7 Try It Yourself

1) Install Android Studio, Flutter, and Dart in your computer and create a new Flutter Project. Run the project which already has some predefined program.

2) Visit https://flutter.dev/ and study their documentation. Also, take a look at their showcase to know what applications are built using Flutter.

Session 2

Applications and Widgets in Flutter

Learning Objectives

In this session, students will learn to:

- Explain how to create a simple Flutter application in Android Studio
- Explain Widgets in Flutter and uses
- List different types of widgets in Flutter
- List different types of layout Widgets

Session Goals:

Flutter applications can be created through a variety of means. In this session, creating a Flutter app in Android Studio is discussed. Further, the session takes you through various widget types in Flutter and their usage.

2.1 Create a Flutter Application in Android Studio

Flutter applications can be created with different code editors. The best way to do this is through Android Studio. The step-by-step process of starting the application creation in Flutter is as follows:

Step 1: Open Android Studio.

Step 2: Create Flutter Project. To do this, click File → New → New Flutter Project.

Step 3: Select Flutter Application. To do this, click Flutter Application in the left pane and then, click **Next**.

Step 4: Configure the application with following details and then, click **Next**.

1. Project name: *flutter_first*
2. Flutter SDK Path: *<path_to_flutter_sdk>* (for example, *D:\fluttersdk\flutter*)
3. Project Location: *<path_to_project_folder>* (for example, *D:\flutterexamples\flutterfirst*)
4. Description: Flutter application

Step 5: Complete the Project creation.
Enter the company domain as *flutter.demo.com* and click Finish.

Step 6: Finish.

2.1.1 Auto-generated Components in Flutter

Whenever Flutter application is created, few files and folders will be generated automatically. Refer to Figure 2.1.

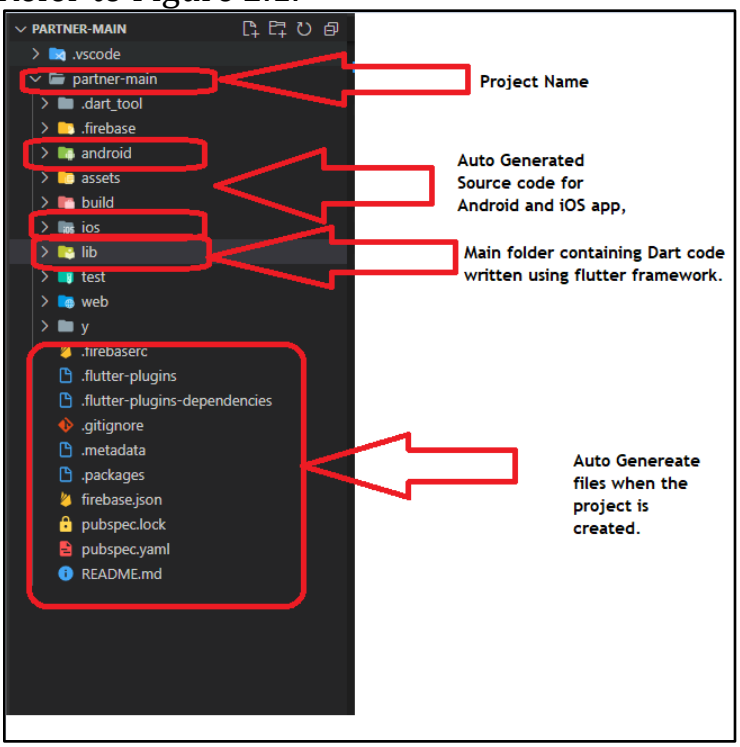


Figure 2.1: Auto-Generated Components in Flutter

In the Flutter project following folders are important:

a. **android:** This folder contains Android specific files. For example, manifest file, gradle file, and so on.
b. **ios**: This folder contains ios specific files. For example, podfile, icons, and so on.
c. **lib:** It contains all the project files. Fox example, Assets, Components, and so on.
d. **lib/main.dart:** Contains the main file at the starting of the app and usually has one sample code by default.
e. **.packages**: Contains information regarding packages that are being used in projects.
f. **.iml:** This file is basically metadata for IntelliJ IDEA to know how to structure the project and how each folder will be used.
g. **pubspec.yaml:** This file contains all dependencies. Execute Flutter get package **package name** to get all required packages.
h. **pubspec.lock:** Lock file lets the user test the package against the latest compatible versions of its dependencies.
i. **README.md:** The README is a basic guide that provides developers with a complete illustration of GitHub project.

2.1.2 Code Structure in Main.dart file

`Main.dart` is the main file for Flutter application. It is generated automatically while creating a Flutter application. Code Snippet 1 shows the auto generated code for `Main.dart`.

Code Snippet 1:

```
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 core of the application.
```

```
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       /*This is the concept of the application.
       Try to run the application with "flutter run". The user
       can see a
       blue toolbar. Then, do not quit the app, try to
       change the primarySwatch  to Colors.green and call on
       "hot reload" (press "r" in the console of "flutter run",
       or just save changes to "hot reload" in the Flutter IDE).
       Note that the counter did not reset back to zero and the
       application
       application is not restarted.*/
       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);

/*This widget is home page of the application. It is stateful,
means that it has a State object that contains some fields that
may affect the apperance.

This class is the configuration for the state. It contains the
values (in this case the title) given by the parent (here it is
App widget) and used by the build method of the State. Fields in
a Widget subclass are at all times marked "final".*/

 final String title;

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

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

 void _incrementCounter() {
   setState(() {

     /*This invoking of setState shows Flutter framework that
     some change has
     happened here, which is the cause to rerun the build method
     hence, the updated values can reflect on the display. If
     _counter is changed without calling setState(), then  build
     method cannot be
     recalled, and nothing will happen.*/

     _counter++;
   });
 }

 @override
 Widget build(BuildContext context) {

/*This method reruns every time the setState is called, as done in
the _incrementCounter method earlier. The Flutter framework is
developed to make rerunning build methods fast, in a way to rebuild
anything that requires updating rather than changing the instances
of the widget individually.*/

   return Scaffold(
     appBar: AppBar(

/*Here the value is taken from the MyHomePage object created by
the App.build method, and that can be used to set the appbar
title.*/

       title: Text(widget.title),
     ),
     body: Center(
       /*Center is a layout widget. Here single child is taken and
positioned in the center of the parent.*/
       child: Column(
```

```
/*Column is also a layout widget. Here the children list is taken
and vertically arranged. By default, it alters itself to fit the
children horizontally, and tries to appear as tall as its parent.
Call on "debug painting" (press "p" in the console, pick the
"Toggle Debug Paint" action from the Flutter Inspector in Android
Studio, or the "Toggle Debug Paint" command in Visual Studio Code)
wireframe can be seen for each widget.Column has properties to
control altering the size itself and how children are positioned.
Here mainAxisAlignment is used to bring the children to center
vertically; the main axis here is vertical axis as the Columns are
vertical (the cross axis might be horizontal).*/

          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 is used to auto-format nicer for
build methods.
    );
  }
}
```

To run Code Snippet 1, first, a device should be connected. Either connect to a mobile phone or create an emulator using Android Studio. Once the device is connected, the project can be executed.

Click the Run button (Green Play button in the menu bar) in Android Studio. The Console will appear and after a few minutes, the application will be loaded

in the device. Refer to Figures 2.2 and 2.3.

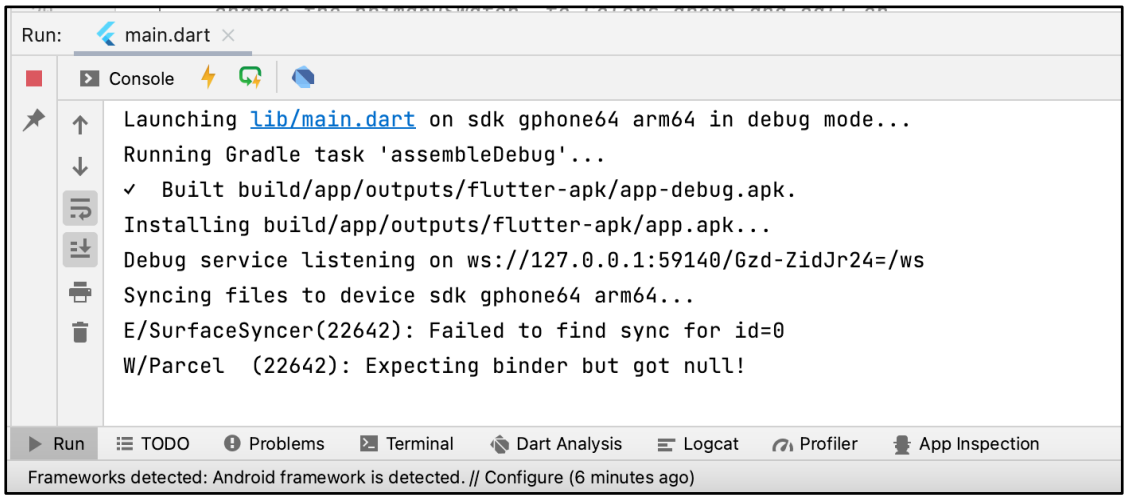


Figure 2.2: Android Studio Console

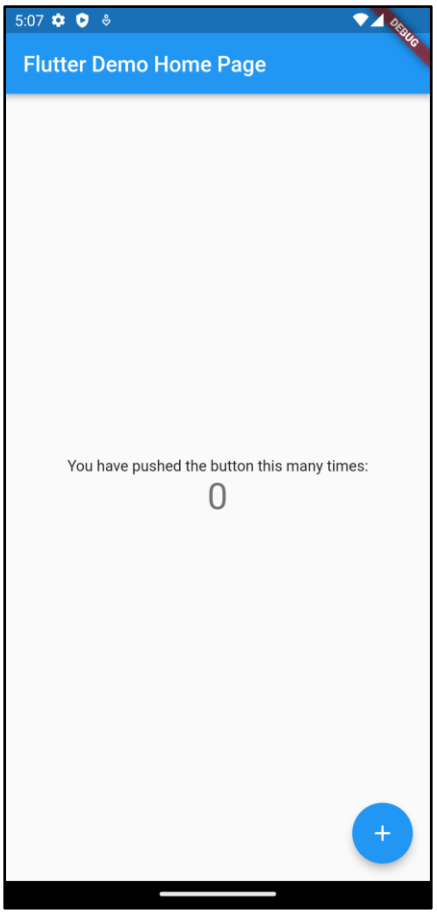


Figure 2.3: Output of Code Snippet 1

2.2 Introduction to Widgets in Flutter and Uses

In Flutter, everything is a widget. If the user wants to add anything, anywhere, and anyhow then, widgets are used.

The main purpose is to create an application from widgets and to describe how the application view should look with the current configuration and state. When any change is made in the code, the widget remakes its description by figuring out the difference between previous widget and the current widget to settle the minimum changes for rendering in the application user interface.

Figure 2.4 depicts the categories of Flutter widgets.

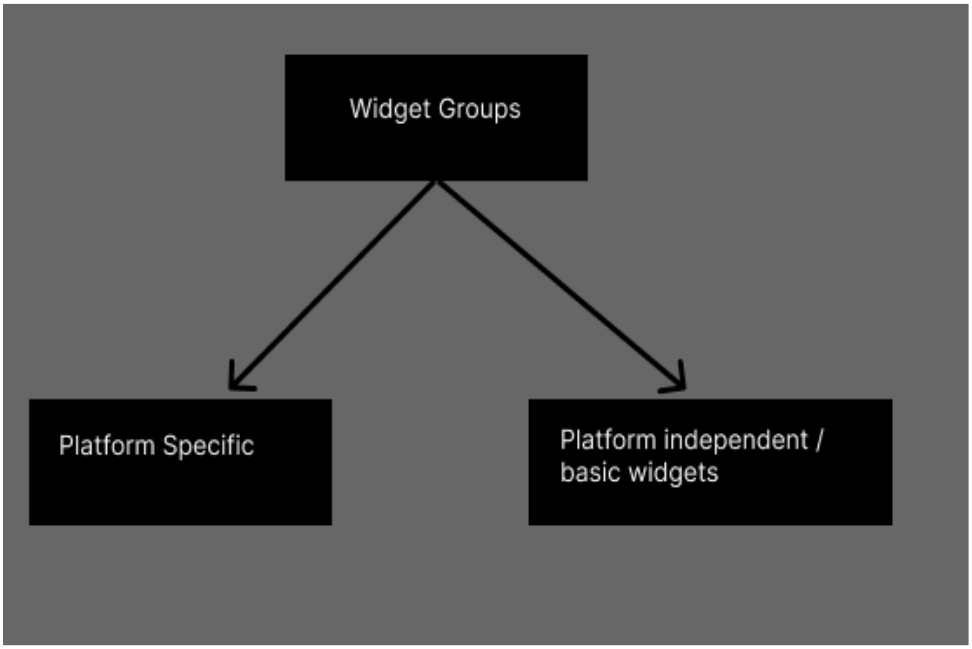


Figure 2.4: Flutter Widgets

2.3 Types of Widgets in Flutter

As seen in Figure 2.4, Flutter widget groups can be grouped into two categories based on the platform.

2.3.1 Platform-Specific

Under platform-specific, one majorly develops applications for Android and iOS and each platform has their own guidelines. For Android, it is Material design guideline and Android-specific widget are called Material Widgets in Flutter.

Some of the Material Widgets are as follows:

- `TabBarView`
- `ListTile`
- `RaisedButton`
- `FloatingActionButton`
- `FlatButton`
- `IconButton`
- `DropdownButton`
- `PopupMenuButton`
- `ButtonBar`
- `TextField`

For iOS (Apple), it is Human Interface Guidelines and iOS-specific widgets called Cupertino widgets.

Some of the Cupertino widgets are as follows:

- `CupertinoTimerPicker`
- `CupertinoNavigationBar`
- `CupertinoTabBar`
- `CupertinoTabScaffold`
- `CupertinoTabView`
- `CupertinoTextField`
- `CupertinoDialog`
- `CupertinoDialogAction`

2.3.2 Platform Independent/Basic Widgets

Under platform-independent Widgets, Flutter provides a large number of basic widgets to create simple or complex UI in a platform-independent manner.

- Text (string, options)
- Image (source)
- Icon (icon_name)

2.3.3 State Maintenance Widgets

State maintenance widgets are those in which the widgets are classified according to state. Figure 2.5 shows different widgets in Flutter.

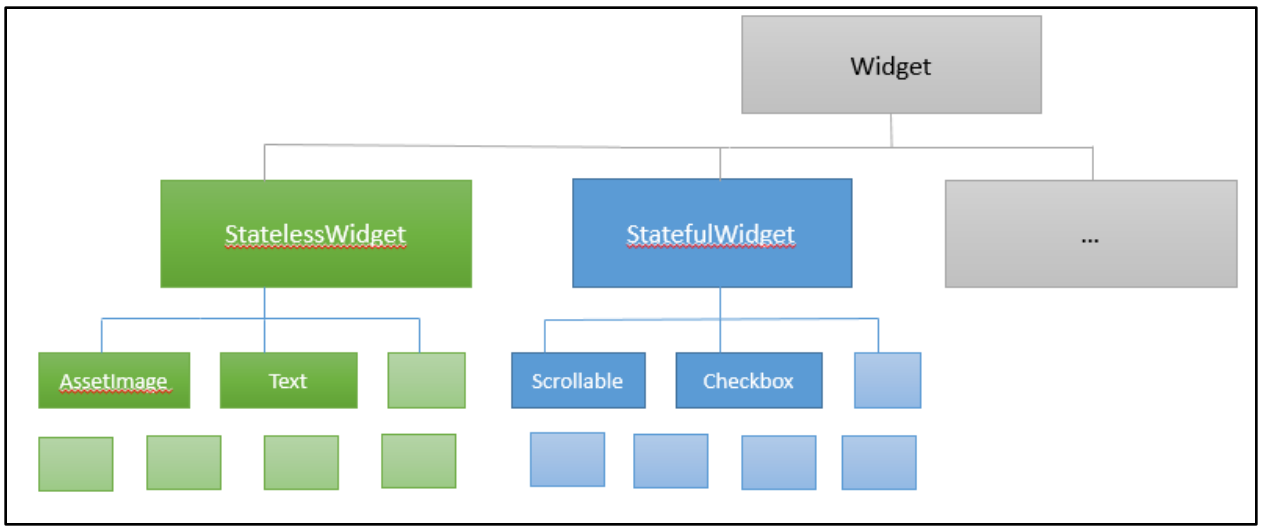


Figure 2.5: Widgets in Flutter

Stateless Widgets:
Stateless widgets are fixed, and its behavior does not change.
Example - icon, text.

Stateful Widgets:
Stateful widgets change with user interface and this widget has one variable named state. In this variable, current state of the widget is stored.

Stateful widgets have one method called `setState` which is important to call when one wants to change any widget. Refer to Code Snippet 2.

Code Snippet 2:

```
class MyHomePage extends StatelessWidget {
   MyHomePage({Key key, this.title}) : super(key: key);
   final String title;
   @override
   Widget build(BuildContext context) {
     return Scaffold(
        appBar: AppBar(title: Text(this.title), ),
        body: Center(child: Text( 'Hello World',),
     ),
       );
    }
}
```

Code Snippet 2 shows a new widget is created by extending `StatelessWidget`. It is to be noted that `StatelessWidget` requires only a single method build to be executed in its derived class.

Figure 2.6 shows the widget tree for the code in Code Snippet 2.

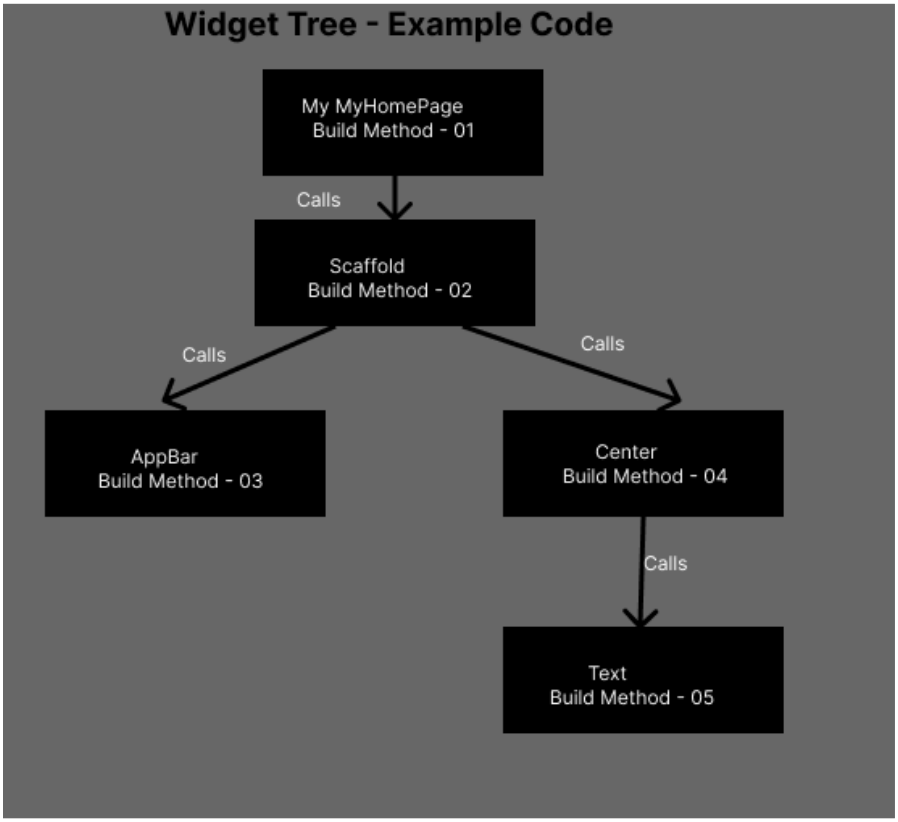


Figure 2.6: Widget Tree

2.3.4 Layout Widgets

In Flutter, a widget may contain one or more widgets. Flutter comes up with huge number of widgets along with layout features to merge multiple widgets into a single one. For example, the center widget helps to bring the child widget to the center. Some of the widgets that work with layout widgets are as follows:

- Container
- Column
- Row
- Stack
- Center

2.4 Types of Layout Widgets

Layout widgets are classified into two types. It is classified on the basis of the child it has.

2.4.1 Single-child Layout Widgets

Single-child layout widgets can have only one child. They tend to have children on their property, to which only one child can be assigned. They are:

- **`Align`**: This widget aligns the child inside itself and it optionally resize to fit the size of the child.
- **`AspectRatio`**: This widget tries to resize the child to a particular aspect ratio.
- **`Baseline`**: A widget that places its child according to the baseline of the child.
- **`Center`**: This widget centers its child within itself.
- **`ConstrainedBox`**: This widget imposes additional limitations on its own child.
- **`Container`**: A convenience widget that combines the common paint, position, and size widgets.
- **`CustomSingleChildLayout`**: This widget holds up the layout of the single child to a representative.
- **`Expanded`**: This widget expands row, column, or bend child.
- **`FittedBox`**: It measures and fits baby in it based on how they fit.
- **`FractionallySizedBox`**: A utility that resizes the child to a fraction of the total available space.
- **`IntrinsicHeight`**: A widget that will adjust the size of its child to the actual height of the child.
- **`IntrinsicWidth`**: A widget that will adjust the size of its child to the intrinsic width of the child.
- **`LimitedBox`**: This box limits its own size only if it is not limited.
- **`Offstage`**: A widget that keeps out the child as if is in a tree, without painting, without the child being available for intervention.

- **`OverflowBox`**: This widget imposes different restrictions on the child than it receives from the parent, which may also allow the child to overflow the parent.
- **`Padding`**: This widget inserts the child conforming to the padding given.
- **`SizedBox`**: This box has a framed size. If this is given to the child, this widget forces the child to have specified width and/or height.
- **`SizedOverflowBox`**: This widget has a certain size, but shifts its actual limitations to its child, which is likely to overflow.
- **`Transform`**: This widget applies a transform before painting the child.

2.4.2 Multi-child Layout Widgets

Multi-child layout widgets can have more than one child and the layout of these widgets are unique. Some of the multi-child widgets are as follows:

- **`Column`**: Lays out the child widget list in vertical direction.
- **`CustomMultiChildLayout`**: This widget uses a representative to resize and position the multiple children.
- **`Flow`**: This widget executes the flow distribution algorithm.
- **`GridView`**: A grid list includes repeating cell patterns arranged in a vertical and horizontal layout. The `GridView` widget executes this component.
- **`IndexedStack`**: This stack displays only one child from the children list.
- **`LayoutBuilder`**: Creates a tree of widgets that can depend on the parent widget size.
- **`ListBody`**: This widget sets its children consecutively along the given axis and forces them to the other axis which is the parent's dimension.
- **`ListView`**: A scrollable, direct widgets list. `ListView` is the commonly used scrolling widget. It exhibits the children one by one in the scrolling direction.
- **`Row`**: This lays out the Child widget list in horizontal direction.
- **`Stack`**: This is used when multiple children want to be overlaid in an easy way, such as overlaying some text and an image.
- **`Table`**: In this widget table layout algorithm is used for its children.
- **`Wrap`**: The widget that exhibits the children in several horizontal or vertical cycles.

2.5 Summary

- It generates many files and folders when a new Flutter project is created.
- The `pubspec.yaml` file contains all information about dependencies, assets, fonts, and so on.
- `MyApp` is the core widget of the Application.
- In Flutter application, everything is a widget. It can be of any type based on usage.
- State management widgets are classified into two types - Stateful and Stateless Widgets.
- Layout widgets are classified into two types - Single-child Layout Widgets and Multi-child Layout Widgets. It is classified on the basis of the child it has.

2.6 Test Your Knowledge

1. Which of the following is not a Layout Widget?
   a. Container
   b. Column
   c. Row
   d. Text
2. What all does `pubspec.yaml` file contain?
   a. Dependencies
   b. Assets declaration
   c. Fonts declaration
   d. All of these
3. Which of these is Single-Child layout widget?
   a. Column
   b. Row
   c. Container
   d. Stack
4. Which file contains information about Guidance to developer?
   a. README.md
   b. pubspec.yaml
   c. pubspec.lock
   d. .iml
5. Android folder does not contain _____.
   a. gradle files
   b. manifest file
   c. pod file
   d. Android app folder

Answers - Test Your Knowledge

1. Text
2. All of these
3. Container
4. README.md
5. pod file

2.7 Try It Yourself

1. Make following changes to Code Snippet 1 given in this session.
    a. Add an image before the '`You have pushed the button this many times:`' Text. It can be any image and give it a height of 100.
    b. Change the color of FloatingActionButton and AppBar.
    c. Import Roboto font into the application and change all texts to Roboto font.

2. Replicate the Profile Screen Design (Second image) from the given reference link. https://dribbble.com/shots/19117061-Baims-App-Design-Concept

Session 3
Building Flutter Application Using Platform Specific Widgets

Learning Objectives

In this session, students will learn to:

- Explain Scaffold and `AppBar`
- Describe `BottomNavigationBar`
- Describe `TabBar` and `TabBarView`
- Describe `ListTile`
- Identify different types of Buttons and explain them
- Define User Input Widgets
- Explain DatePickers and Dialogs

Session Goals:

In Flutter, widgets are classified into many different types. Out of them, one classification is on the basis of platform. Some widgets in Flutter are platform specific which means that based on platform used, they will produce a different output. This session introduces widgets such as `Scaffold`, `AppBar`, `BottomNavigationBar`, `TabBar`, and more. The session finally concludes by explaining User Input Widgets and DatePickers and Dialogs.

3.1 Scaffold

Scaffold is a built-in widget in which all visual layouts are implemented. It is also utilized to form basic structure of an application's functional layout.

Using Scaffold, it is possible to quickly implement functional widgets such as `FloatingActionButton`, `AppBar`, `Drawer`, `ButtonNavigationBar`, and many other widgets into an application.

3.2 AppBar

`AppBar` is a major component of Scaffold and it is displayed horizontally on the topmost section of the screen. It is often used to display the title of the application.

Figure 3.1 shows the definition of constructor of Scaffold in the Dart library and the properties that the constructor accepts.

```
Scaffold({
  Key key,
  PreferredSizeWidget appBar,
  Widget body,
  Widget floatingActionButton,
  FloatingActionButtonLocation floatingActionButtonLocation,
  FloatingActionButtonAnimator floatingActionButtonAnimator,
  List persistentFooterButtons,
  Widget drawer,
  Widget endDrawer,
  Widget bottomNavigationBar,
  Widget bottomSheet,
  Color backgroundColor,
  bool resizeToAvoidBottomPadding,
  bool resizeToAvoidBottomInset,
  bool primary: true,
  DragStartBehavior drawerDragStartBehavior:
  DragStartBehavior.start,
  bool extendBody: false,
  bool extendBodyBehindAppBar: false,
  Color drawerScrimColor,
  double drawerEdgeDragWidth,
  bool drawerEnableOpenDragGesture: true,
  bool endDrawerEnableOpenDragGesture: true
})
```

Figure 3.1: Scaffold Constructor and Properties

3.3 BottomNavigationBar

`BottomNavigationBar` resembles the `AppBar`, but is displayed at the bottom section of the page. It is possible to use `BottomNavigationBar()` or `BottomAppBar()` widget on Scaffold.

Code Snippet 1 shows an example of using `Scaffold`, `AppBar`, `BottomNavigationBar`, and `FloatingActionButton`.

Code Snippet 1:

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
/// This Widget is the main application widget.
Class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyStatefulWidget(),
    );
  }
}
class MyStatefulWidget extends StatefulWidget {
  MyStatefulWidget({Key? Key}) : super(key: key);
  @override
  _MyStatefulWidgetState createState() =>
_MyStatefulWidgetState();
}
class _MyStatefulWidgetState extends State<MyStatefulWidget>
{
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.amber,
      appBar: AppBar(
        title: Text('Scaffold Example'),
      ),
      body: Center(
        child: Text('Hi, happy learning Flutter!!!'),
      ),
      bottomNavigationBar: BottomNavigationBar(
        items: [
          //items inside navigation bar
          BottomNavigationBarItem(
            icon: Icon(Icons.home),
            label: "Home",
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.search),
            label: "Search",
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.camera),
            label: "Take Photo",
          ),
          //put more items here
```

```
        ],
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => setState(() {}),
        child: Icon(Icons.add),
      ),
      floatingActionButtonLocation:
             FloatingActionButtonLocation.centerDocked,
      drawer: Drawer(
        elevation: 20.0,
        child: Column(
          children: <Widget>[
            UserAccountsDrawerHeader(
              accountName: Text("Flutter Learning"),
              accountEmail:
                   Text("Flutterlearning@gmail.com"),
              currentAccountPicture: CircleAvatar(
                backgroundColor: Colors.blue,
                child: Text("Ross"),
              ),
            ),
            ListTile(
              title: new Text("Inbox"),
              leading: new Icon(Icons.mail),
            ),
            Divider(
              height: 0.1,
            ),
            ListTile(
              title: new Text("Primary"),
              leading: new Icon(Icons.inbox),
            ),
            ListTile(
              title: new Text("Social"),
              leading: new Icon(Icons.people),
            ),
            ListTile(
              title: new Text("Promotions"),
              leading: new Icon(Icons.local_offer),
            )
          ],
        ),
      ),
    );
  }
}
```

Figure 3.2 shows the output for the program in Code Snippet 1. The components such as `AppBar`, `BottomNavigationBar`, and Floating Action Bar are clearly mentioned in Figure 3.2.

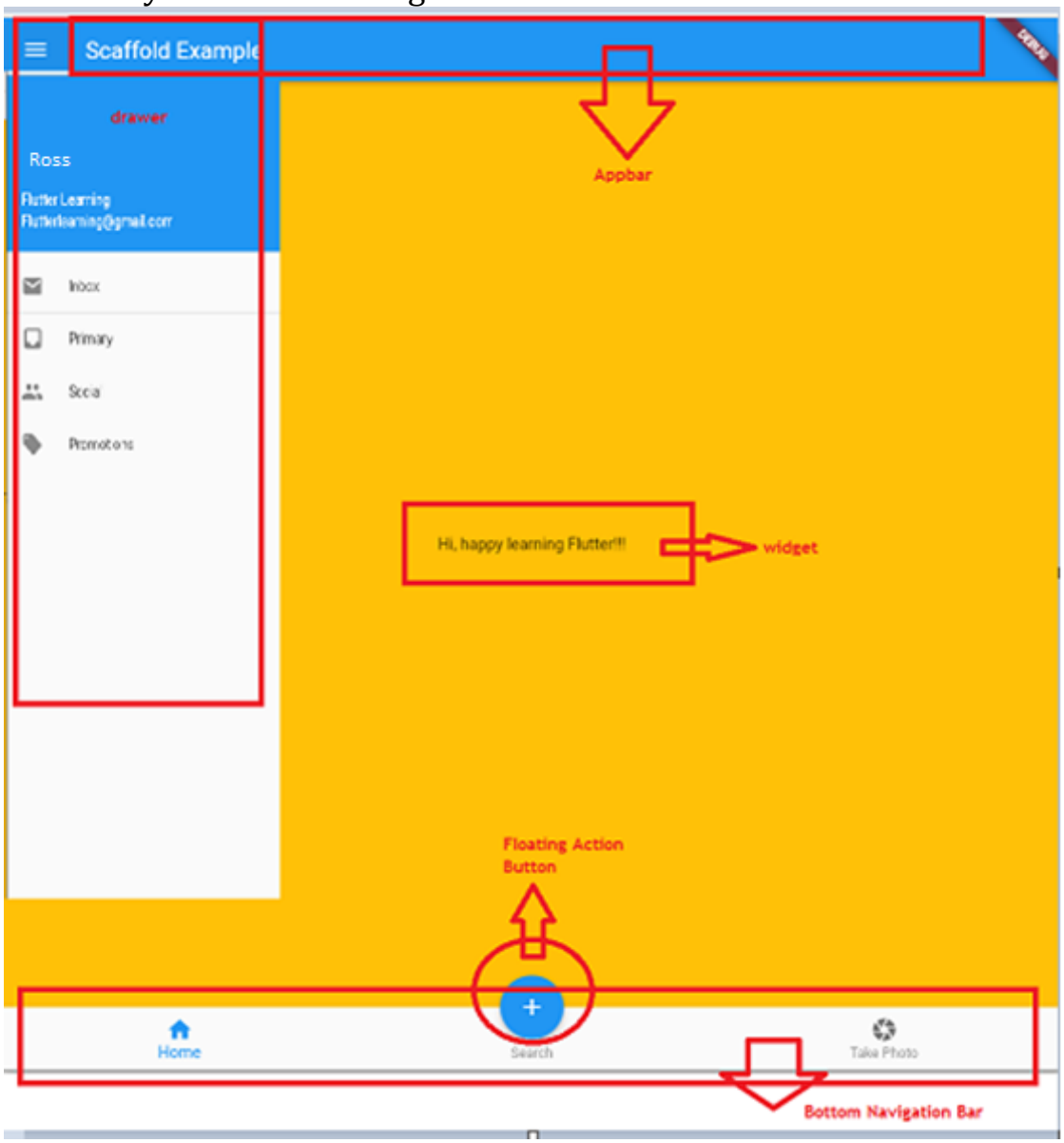


Figure 3.2: Scaffold, AppBar, BottomNavigationBar, and FloatingActionButton Sample Output

3.4 `TabBar` and `TabBarView`

Tabs are common features in a number of applications. Built-in applications with an OS such as File Explorer in Windows 10 also make use of tabbed views when showing file information. In Flutter, `TabBar` is utilized to create tabs while `TabBarView` is used to create content of each panel within tabs. When

using `TabBar` and `TabBarView`, `TabController` must be utilized to handle several aspects of a tab. It is preferable to use Default`t`TabController since it is easy to use. Additionally, it is also possible to use a custom `TabController`, but it requires varied inputs such as length, children, and tabs that are in use.

Code Snippet 2 shows a sample program with `TabBar` implemented and the corresponding output for the program is showcased in Figure 3.3.

Code Snippet 2:

```
MaterialApp(
      home: DefaultTabController(
        length: 3,
        child: Scaffold(
          appBar: AppBar(
            bottom: const TabBar(
              tabs: [
                Tab(icon: Icon(Icons.directions_car)),
                Tab(icon: Icon(Icons.directions_transit)),
                Tab(icon: Icon(Icons.directions_bike)),
              ],
            ),
            title: const Text('Travel Application - Demo'),
          ),
          body: const TabBarView(
            children: [
              Icon(Icons.directions_car),
              Icon(Icons.directions_transit),
              Icon(Icons.directions_bike),
            ],
          ),
        ),
      ),
    );
```

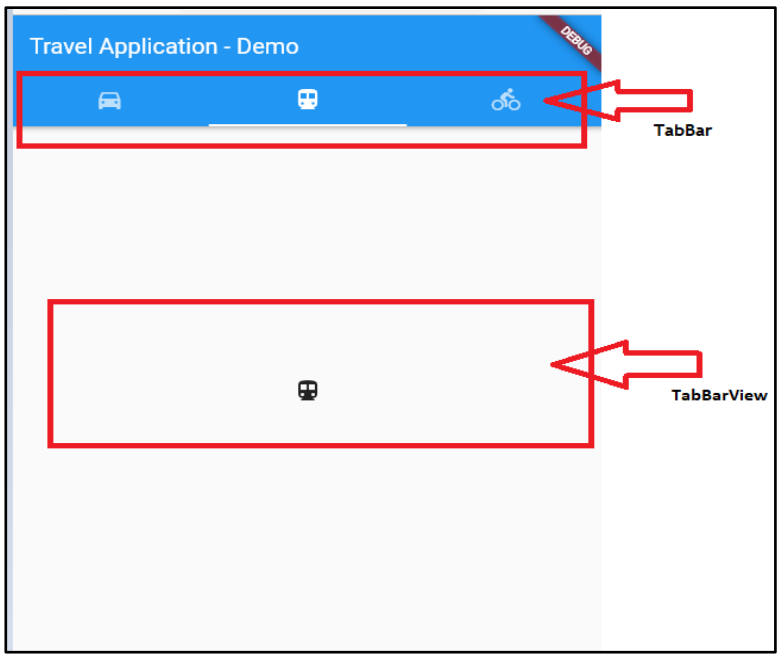


Figure 3.3: TabBar and TabBarView Sample

Figure 3.3 shows the `TabBar` at the top of the page and the contents that are displayed using the `TabBarView`. Markers are used to point separate components.

3.5 ListTile

The `ListTile` widget contains `title`, `subtitle`, `leading`, and `trailing` widgets which makes it perfect for use cases such as Profile, Row Buttons, FAQ Sections, and so on. Code Snippet 3 demonstrates the usage of `ListTile`.

Code Snippet 3:

```
ListTile(
                title: Text('Demo ListTile'),
                subtitle: Text('Usage of ListTile'),
                trailing: Icon(Icons.favorite,
                color: Colors.red),
)
```

Figure 3.4 shows the output of the program that implements `ListTile`. The tile, title, and trailing icon are highlighted.

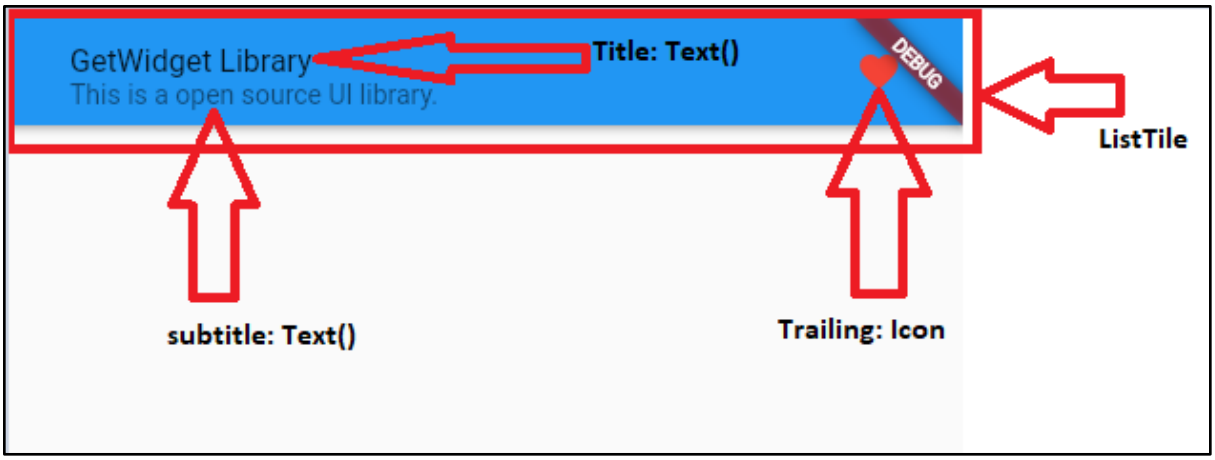


Figure 3.4: Output for ListTile Implementation

3.6 Buttons

When creating a simple app with a login page, once the user enters the username and password, the app should take the authenticated user to the welcome screen. Here, the logic for authentication happens when the button is clicked after the user enters the username and the password.

Buttons are Material components in Flutter which are used to trigger actions such as submitting forms, making selections, and so on. Buttons are one of the relative user components while designing an application.

For example, a text button to display text and an icon button to display a clickable icon. In addition to buttons, Flutter also provides a wide range of features for styling and shaping buttons.

Following are the types of buttons in Flutter:

1. Raised Button
2. Floating Action Button
3. Icon Button
4. Popup Menu Button

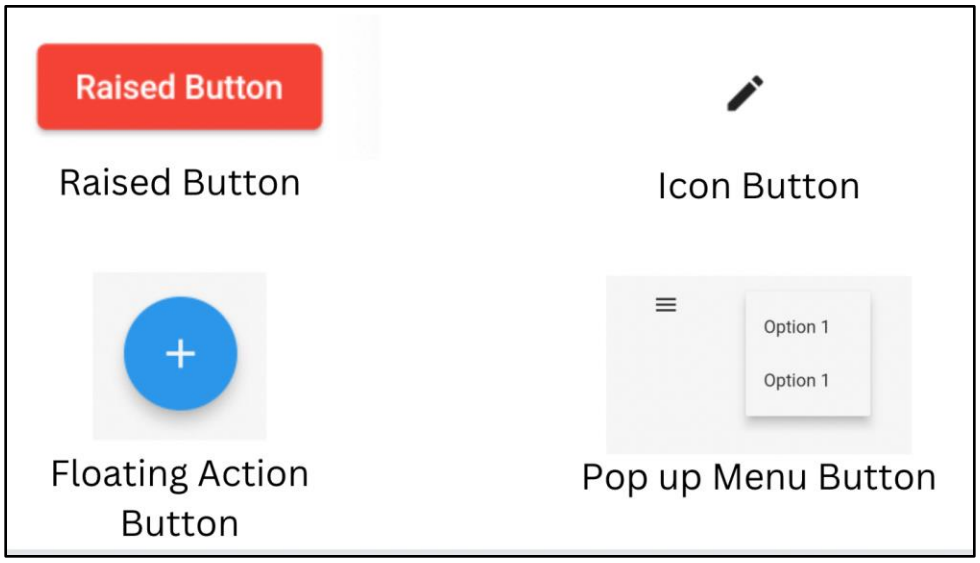


Figure 3.5: Types of Button in Flutter

3.6.1 Raised Button

In Flutter, the `RaisedButton` widget gives the dimensionality to the UI along the z-axis with hints of shadow.

`RaisedButton` is deprecated. To display a button in the same manner as a `RaisedButton`, one can use `ElevatedButton`. In `ElevatedButton`, parameter styles were present, but in `RaisedButton` only a few predefined style parameters are available. Other than that, all functionalities are the same. There are many properties that can be set for `RaisedButton`. Some of the most important ones are button color, text color, and disabled color when a button is disabled, height, animation time, shape, offset, and so on. Raised buttons also have callback functions.

Some of these include:

`onPressed()`	`onLongPress()`
➢ This callback is fired when the user clicks the button.	➢ This callback is fired when the user presses and holds the button for a long time.

Note that the Raised button will be in a disabled state if the `onPressed()` and `onLongPress()` callbacks are not available.

3.6.2 Floating Action Button

This is a circular button with an icon which hangs at the top of the content. It supports the primary action in the app, which means that it is used to give an extra focus that users can easily get attracted to. Floating action buttons are most often used in the `Scaffold.floatingActionButton` field. It is a parameter in the Scaffold widget using which you can add a floating button.

3.6.3 Icon Button

Flutter, by default, has several types of buttons such as `ElevatedButton`, `TextButton`, `OutlinedButton`, and so on. However, most of the buttons only contain text. There is one other type of button that does not use any text, but instead uses icons in place of text. These buttons called Icon Buttons are supposedly one of the most used buttons in the Flutter library for creating UIs.

3.6.4 Popup Menu Button

This button displays a pop-up menu when pressed and an item can be selected from the menu that pops up. There can be several options in the popup menu that appears, and the user is free to choose any option they please based on the requirement at that particular stage.

3.7 User Input Widgets

Whenever input from user is required, then user input widgets are used. For example, if profile information of a user is required then, username, gender, and types of services provided by the user are expected. An application can use user input widgets to accept these data. `TextField` can be used to get the name, `RadioButton` to accept gender, and `CheckBox` to accept different services the user is providing.

In Flutter, there are many input data widgets to help the developer get several kinds of information from users.

Following are some Input Widgets available in Flutter:

- `TextField`
- `RadioButton`
- `CheckBox`

3.7.1 TextField

It is used to accept alphanumeric input data (such as username, password, product price, publication year, and so on) from the keyboard into the

application. It is important to note that, by default, the content from this widget is decorated with underlines and it is possible to change the behavior of the widget using extra decoration options.

3.7.2 RadioButton

This button is commonly used when one wants to choose a single option from a list of options. Options such as Male/Female/Other, True/False, Single/Married/Widow, and so on can be given and one value can be chosen from this list using the RadioButton. It is important to note that the default behavior of a RadioButton is to choose only one option. This behavior cannot be changed.

3.7.3 CheckBox

This option helps to choose multiple options at one time. For example, when developing a feature which involves selecting multiple inputs or parameters from the user, checkboxes can be utilized. For example, to check what services a vendor is offering, inputs from a checkbox can be taken. Figure 3.6 shows a few User Input widgets in Flutter.

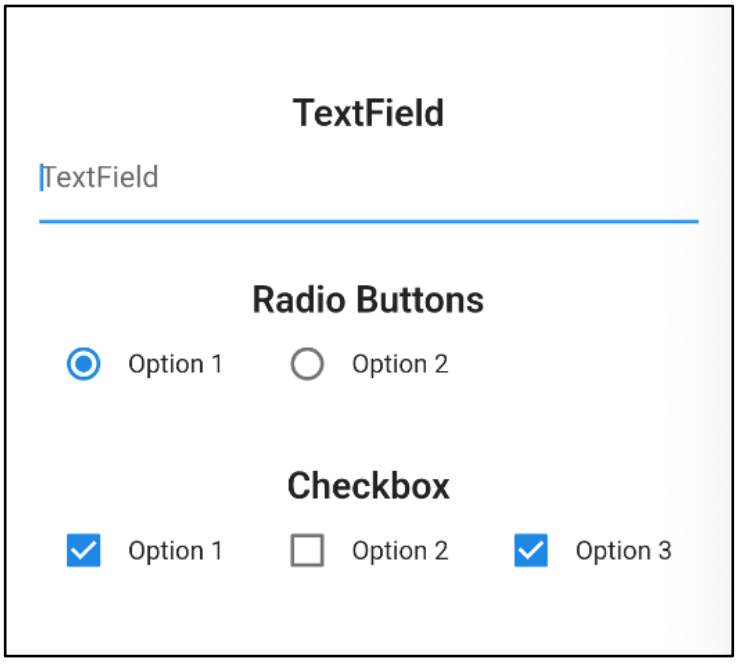


Figure 3.6: User Input Widgets in Flutter

3.8 DatePickers and Dialogs

DatePickers and dialogs are simple widgets that allow users to select a date, view alert dates, or confirm any action. For example - to confirm if the user wants to log out or not or to confirm whether to make a payment or not.

3.8.1 DatePicker and TimerPicker

This is a Material widget in a Flutter that allows the user to pick a date and time. If there is no widget available to create a date and time picker, the `showDatePicker()` and `showTimePicker()` functions can be utilized to achieve the same. These two functions display the Material Design date and time selection in a dialog by calling the built-in Flutter functions `showDatePicker()` and `showTimePicker()` respectively. Additionally, it is important to note that datePicker constructor have three major values.

Following are some examples of values or options that can be given in an application while using `DatePicker`:

1. **`firstDate`**: The first date from which the user can choose. They cannot choose a date before the specified starting date.
2. **`initialDate`**: The initial date that will be selected when dialog is first opened.
3. **`lastDate`**: The last date beyond which the user cannot select.

Time selector constructor

The time selector constructor has an `initialTime` parameter. `DatePicker` cannot be used without `initialTime`. Additionally, the start time has a `TimeOfDay` type that describes the time in the beginning when the user opens the dialog box.

It is important to note that `initialTime` can also display the current time. The current time represents time as of today and will be highlighted in the time grid. If no value is specified, the `TimeOfDay.now()` which denotes the present time and date will be used. Figures 3.7 A and B show the `DatePicker` and `TimerPicker` widget dialogs in Flutter.

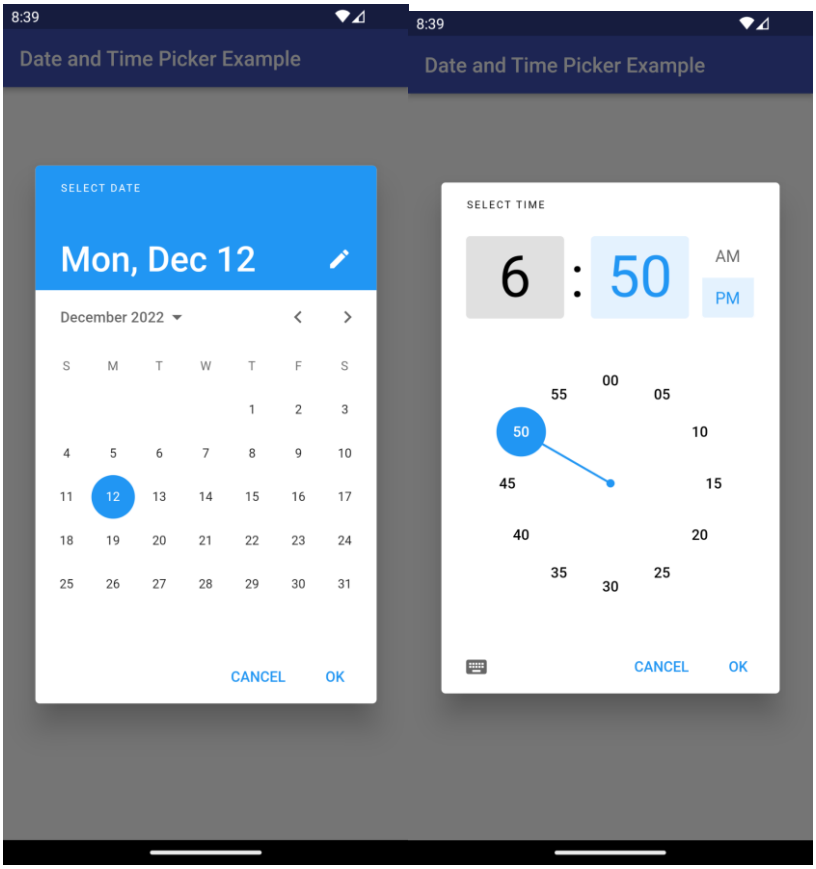


Figure 3.7 A and B: DatePicker and TimerPicker in Flutter

3.8.2 Simple Dialog and AlertDialog

A dialog is a widget in Flutter that appears on the screen whenever a user has to decide regarding any critical piece of information.

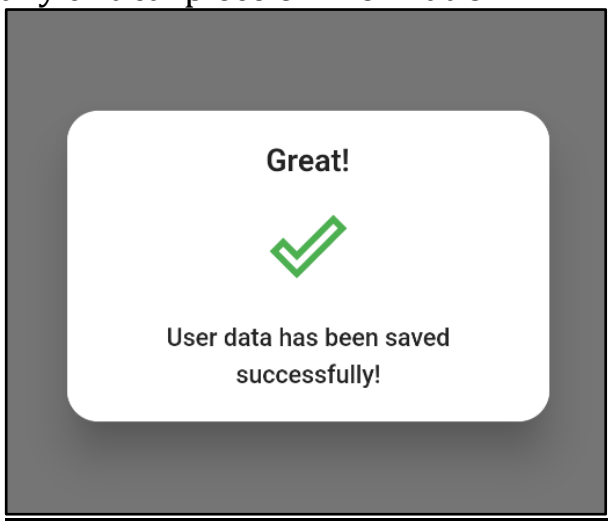


Figure 3.8: Simple Dialog in Flutter

Alert dialogs are a type of Material design widget that inform the user about some scenarios to be aware of. Code Snippet 4 shows the `AlertDialog` widget in Flutter.

Code Snippet 4:

```
showDialog(
      context: context,
      builder: (_) {
      return AlertDialog(
          title: Text('Alert Dialog'),
          content: Text('This is an Alert Dialog!'),
          actions: [
        TextButton(
          onPressed: () => Navigator.pop(context),
          child: Text('Cancel')),
        TextButton(
          onPressed: () => Navigator.pop(context),
          child: Text('Ok'))
    ],
   );
 });
```

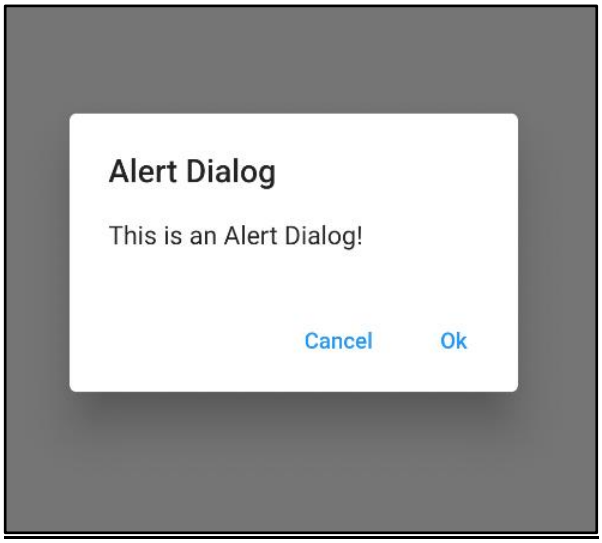


Figure 3.9: AlertDialog in Flutter - Output of Code Snippet 4

3.9 Summary

- A Scaffold is a base widget for a single page.
- `AppBar` is a widget that is displayed at the top of the screen. It is the space where one can add other widgets and can modify those widgets according to their respective wants.
- Flutter has four types of buttons - Raised Button, Floating Action Button, Icon Button, and Popup Menu Button.
- User Input widgets are of three main types - `TextField`, `RadioButton`, and `CheckBox`.
- Simple Dialog and Alert Dialog are two types of dialogs found in a Flutter.

3.10 Test Your Knowledge

1. AppBar is a component of _____.
   a. Scaffold
   b. Column
   c. Row
   d. Stack

2. What is `BottomNavigationBar`?
   a. Component of Scaffold
   b. Component of `AppBar`
   c. None of these
   d. All of these

3. `ElevatedButton` is a replacement of _____.
   a. Raised Button
   b. Icon Button
   c. Container
   d. Stack

4. Which of the following contains title as well as leading or trailing icons?
   a. `ListTile`
   b. `ElevatedButton`
   c. Raised Button
   d. `Textfield`

5. Which of the following is not a User Input Widget?
   a. `TextField`
   b. `RadioButton`
   c. `Checkbox`
   d. Raised Button

Answers - Test Your Knowledge

1. Scaffold
2. Component of Scaffold
3. Raised Button
4. `ListTile`
5. Raised Button

3.11 Try It Yourself

1. Design an application with the `BottomNavigationBar` similar to the Instagram application.

2. Design a good looking form for collecting user details in Flutter. The Form will contain following fields in order:
   a. `TextFields` for First Name and Last Name in the same Row.
   b. Radio Button with options for selecting Gender.
   c. A Slider with value between 0 and 100 for choosing age.
   d. A `TextField` with a `SuffixIcon` which on clicking should open the DatePicker. This field is for entering Date of Birth.
   e. A `RaisedButton` with text 'Submit'. On clicking this button, every field listed in the form should reset to empty values.

Session 4
Building Flutter Application Using Platform Independent Widgets

Learning Objectives

In this session, students will learn to:

- Explain `Text` widget
- Explain `Image` widget
- Explain `Icon` widget
- Explain `Form` widget

Session Goals:

This session explains different widgets present in Flutter. It is possible to build Flutter application using independent widgets. There are basic widgets that are called independent widgets. These widgets do not have children within them.

4.1 Text Widget

The `Text` widget helps one create Text Elements and display them in the application. The `Text` widget can be customized according to what the user wants. For example, animation, colors, size, and so on can be customized.

Create `Text` widget

`Text` widget takes in a mandatory string parameter. It can have other parameters such as `TextStyle`, key, and so on. However, they are optional.

Following is the syntax for `Text` widget:

```
Text("Flutter Development")
```

Adjust the style of the `Text` widget

The `Text` widget can have a `style` parameter. This is an extra parameter that can be added along with the syntax to add style-based additions. Text attributes such as font weight, color, font size, and so on can all be changed using this parameter. Code Snippet 1 shows an example of `Text` widget in Flutter.

Code Snippet 1:

```
Text( 'This is a text widget', style: TextStyle( fontSize:
32.0, fontWeight: FontWeight.bold, color: Colors.red, ), )
```

Figure 4.1 shows a `Text` widget used as part of a Flutter application.

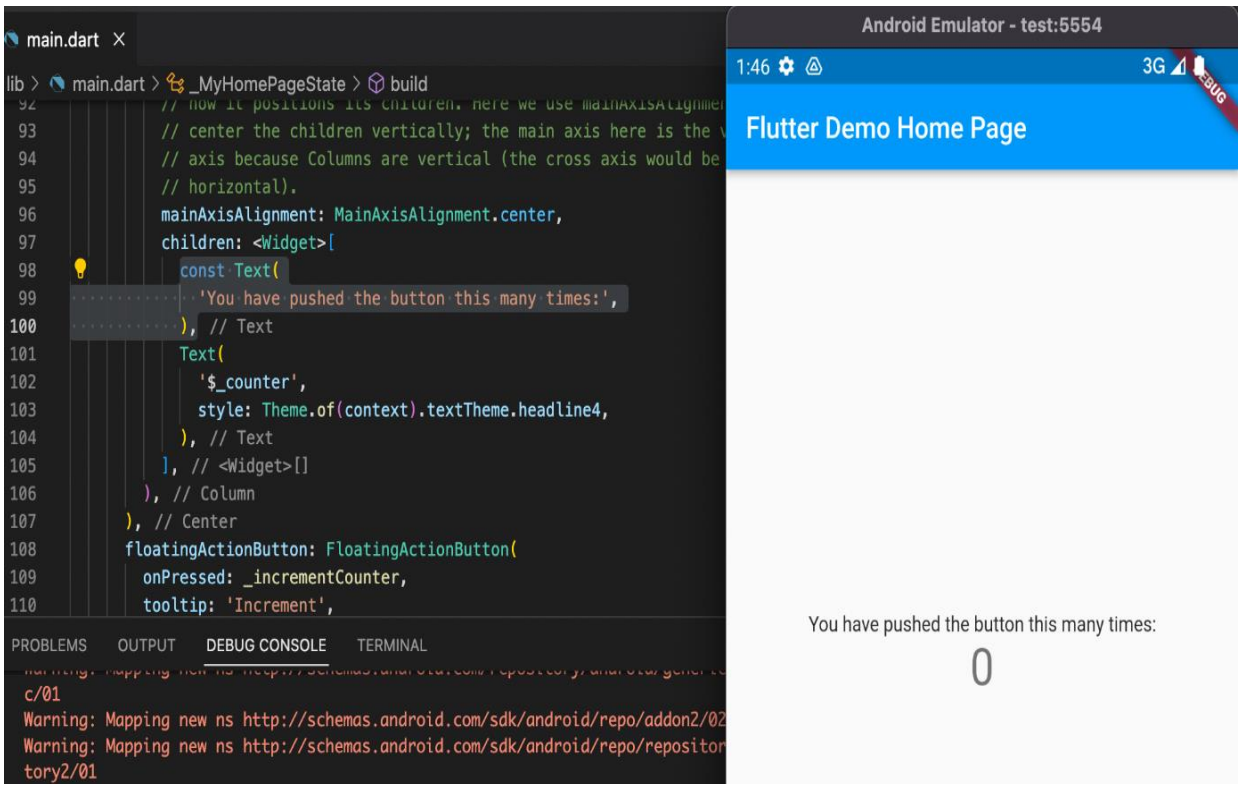


Figure 4.1: `Text` widget

4.2 Image Widget

When a user wants to add an image to an application, one can do it by adding those images to the assets folder. Asset images are present as part of the asset bundle of the app and are deployed with the app so that they are readily available during run time. It is important to note that asset images can be displayed using the `Image` class in Flutter.

Constructors to load images in `Image` widget

1. `Image.asset` - Utilized for displaying images from the assets bundle.
2. `Image.file` – Utilized for displaying images from a specific file.
3. `Image.memory` – Utilized for displaying images from memory. The image will be stored as `Uint8List` which is the fastest way to load an image from memory.
4. `Image.network` – Utilized for displaying images from URL

`Image` class has several properties.

Code Snippet 2 shows the properties of the `Image` Class.

Code Snippet 2:

```
 String name, {
 Key? key,
 AssetBundle? bundle,
 Widget Function(BuildContext, Widget, int?, bool)?
frameBuilder,
 Widget Function(BuildContext, Object, StackTrace?)?
errorBuilder,
 String? semanticLabel,
 bool excludeFromSemantics = false,
 double? scale,
 double? width,
 double? height,
 Color? color,
 Animation<double>? opacity,
 BlendMode? colorBlendMode,
 BoxFit? fit,
 AlignmentGeometry alignment = Alignment.center,
 ImageRepeat repeat = ImageRepeat.noRepeat,
 Rect? centerSlice,
```

```
 bool matchTextDirection = false,
 bool gaplessPlayback = false,
 bool isAntiAlias = false,
 String? package,
 FilterQuality filterQuality = FilterQuality.low,
 int? cacheWidth,
 int? cacheHeight,
```

Utilizing the properties as in Code Snippet 2, users can adjust parameters such as design, style, height, width, and so on.

Out of the properties, the `BoxFit` property is an important one for the Image Class. This property can be utilized to set images according to the users' requirements. If one wants to cover, fill, or contain the image, they can set the respective parameter along with the `BoxFit` Parameter. Code Snippet 3 shows the `BoxFit` property being utilized in a sample code and Figure 4.2 shows the output for the code in Code Snippet 3.

Code Snippet 3:

```
Image.network(
"https://storage.googleapis.com/cms-storage-
bucket/70760bf1e88b184bb1bc.png",
              fit: BoxFit.contain,
              height: 100,
              width: 100,
            ),
```

Figure 4.2: Output for Code with `BoxFit` Property

4.3 Icon Widget

Icons can be used as a representative symbol for quick understanding of functionality, navigation of path, and so on.

Following are some examples that are discussed:

1. Basic Icon widget with default values.
2. Resize the icon using the size property.
3. Change icon color using the color property.

Basic Icon

Code Snippet 4 shows the basic `Icon` widget with default Values.

Code Snippet 4

```
@override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[Icon(Icons.ac_unit_outlined)],
       ),
     )
   );
 }
}
```

Resizing icon

Code Snippet 5 shows the resize `Icon` using `size` property.

Code Snippet 5

```
@override
 Widget build(BuildContext context) {
   return Scaffold(
       appBar: AppBar(
```

```
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Icon(
              Icons.ac_unit_outlined,
              size: 80,
            )
          ],
        ),
      ));
 }
```

Changing Color of the icon

Code Snippet 6 shows the code to implement `Icon` color change.

Code Snippet 6

```
Icon(
           Icons.ac_unit_outlined,
           size: 80,
            color: Colors.blue,
    )
```

Figure 4.3 depicts the output.

Output:

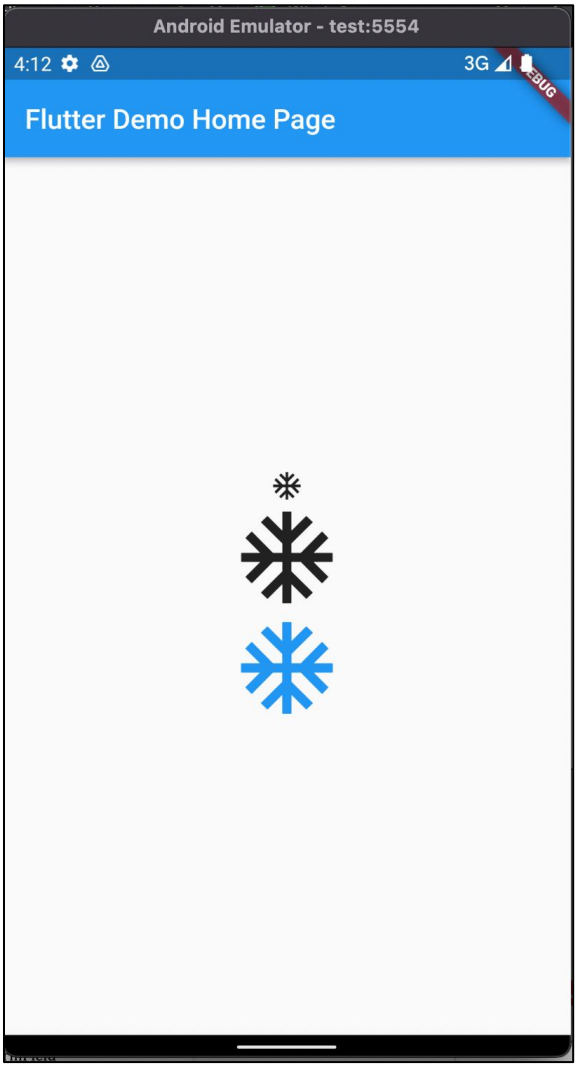


Figure 4.3: Demonstration Use of Icon widget

4.4 Form Widget

Form validation and form submission are two things that the `Form` widget helps to implement. For an application to be easy to use and safe, it is important to check if the information provided by the user is valid. If the user happens to fill out the form correctly, the information can be processed but if the user submits incorrect information, the widget displays an error message which is user friendly to let the users know what exactly happened.

4.4.1 Properties of Form

Some of the properties discussed in Figure 4.9 plays an important role in shaping the output of a form. The property called 'key' is assigned a value if user wants to do any kind of validation. The 'child' property on the other hand is used to have widget inside the form. Autovalidate has a boolean value and can be used to validate fields. If the user marks autovalidate to true, AutovalidateMode will take over and manual validation will not happen.

Code Snippet 7 shows the properties associated with form.

Code Snippet 7:

```
Form Form({
 Key key,
 Widget child,
 bool autovalidate = false,
 Future<bool> Function() onWillPop,
 void Function() onChanged,
 AutovalidateMode autovalidateMode,
})
```

4.4.2 Different Types of Form widgets

There are two types of widgets. They are `Form` and `FormField<T>`. The `Form` widget maintains `FormField` states and stores data such as current value and validity.

It is important point to note is that a form widget acts as a container where multiple form fields are grouped and validated. An example of this is shown with the username and password fields. Syntax of `Form` widget is as follows:

Syntax:

```
Form({
key key,
@required Widget child,
bool autovalidate: false,
WillPopCallback onWillPop,
VoidCallback onChanged
})
```

When one wants to create a Form widget, it is mandatory to create a `GlobalKey`. The `GlobalKey` allows one to link to the form in the future.

```
final _formKey = GlobalKey<FormState>();
```

The `GlobalKey` also gives one the ability to invoke methods such as `save` in order to save the value of each child `FormField` in the `Form` widget.

4.5 Summary

- Texts can be customized in `Text` widget with `TextStyle`. It is possible to define the size, weight, color, and all other attributes related to texts using the `Text` widget.
- Images in Flutter can be displayed by four methods - assets, file, memory, and network
- `Image` under a certain size can be fitted using the `Image` widget.
- In Flutter, `Icon` widget can show several inbuilt Flutter icons. One can also import icons by utilizing the `Icon` widget.
- The `Form` widget in Flutter helps to implement validation and form submission.

4.6 Test Your Knowledge

1. `Form` widget helps in:
   a. Creating a style
   b. Developing our icons
   c. Create validation in order to make any kind of form
   d. Creating APK

2. `Boxfit` helps in ____.
   a. Fitting image in defined box
   b. Give border to a container
   c. Fitting any widget
   d. Adjusting the border

3. Text widget in Flutter can be customized using _____.
   a. `TextStyle`
   b. `Size`
   c. `Custom`
   d. `FormField`

4. Which of the following widget uses a Key?
   a. `Text`
   b. `Form`
   c. `Icon`
   d. `Image`

5. `Image.memory` widget wants image in which format _____.
   a. `Uint8List`
   b. `String`
   c. `Bytes`
   d. `file`

Answers - Test Your Knowledge

1. Create validation in order to make any kind of form
2. Fitting image in defined box
3. `TextStyle`
4. `Form`
5. `Uint8List`

4.7 Try It Yourself

1. Create a Flutter application where Image will be shown based on the selected Image type. The screen will have a large space for displaying Image and three buttons in a Row at the bottom.
    a. The three Buttons are: Asset Image, Network Image, and Memory Image.
    b. Declare three Image widgets each for Asset, Network, and Memory
    c. On clicking the button, the respective image must be displayed.
    d. **Bonus:** Try using `FadeInImage` and see how the image fades between toggling the buttons.

2. Design a Flutter application which has a `TextField` with Form validation.
    a. Create following TextStyles and store it to a variable.
        i. fontSize: 30, fontWeight: 600
        ii. fontSize: 25, fontWeight: 600, color: Black
        iii. fontSize: 20, fontWeight: 300, Color: Blue
        iv. fontSize: 15, fontWeight: 500, Color: Yellow
        v. fontSize: 12, fontWeight: 400, Color: Red, decoration: Underlined
    b. First Screen will contain a `TextField`, some set of buttons each representing the `TextStyle` created and a Submit Button.
    c. The TextField will take a name and proper validations are to be handled such that length of given text should be more than three.
    d. Then, the desired `TextStyle` has to be selected by clicking the button.
    e. Finally, upon clicking, the submit button should navigate to the Next screen, where the provided name will be displayed in the next screen with selected `TextStyle`. Restrict Navigation and show warning if the validation fails.

Session 5
Building a Flutter Application Using Single Child Layout Widgets

Learning Objectives

In this session, students will learn to:

- Describe Single Child Layout widgets
- Explain `Container` widget, `Padding` widget, `Align` widget, `Center` widget, and `SizedBox` widget

Session Goals:

Single Child Layout widgets are widgets in which users can define only one widget at a time. Flutter has a widget tree that can show the parent as well as the children widgets. A Single Child Layout widget can be made as a parent widget and a Multi-Child Layout widget can be placed inside it. This way, users can have multiple widgets in a view and can use the widget tree to view the hierarchy. This session explains Single Child Layout widgets and `Container` widget, `Padding` widget, `Align` widget, `Center` widget, and `SizedBox` widget.

5.1 Container Widget

A `Container` widget is one of the most popular widgets in Flutter. It can be used to accomplish various purposes such as creating a box with color, customizing with a neat, curved border, placing any other widget inside as a child, and so on. Container also has properties such as padding, margin, shape, alignment, and constraints.

Figure 5.1 shows a `Container` widget with a child content.

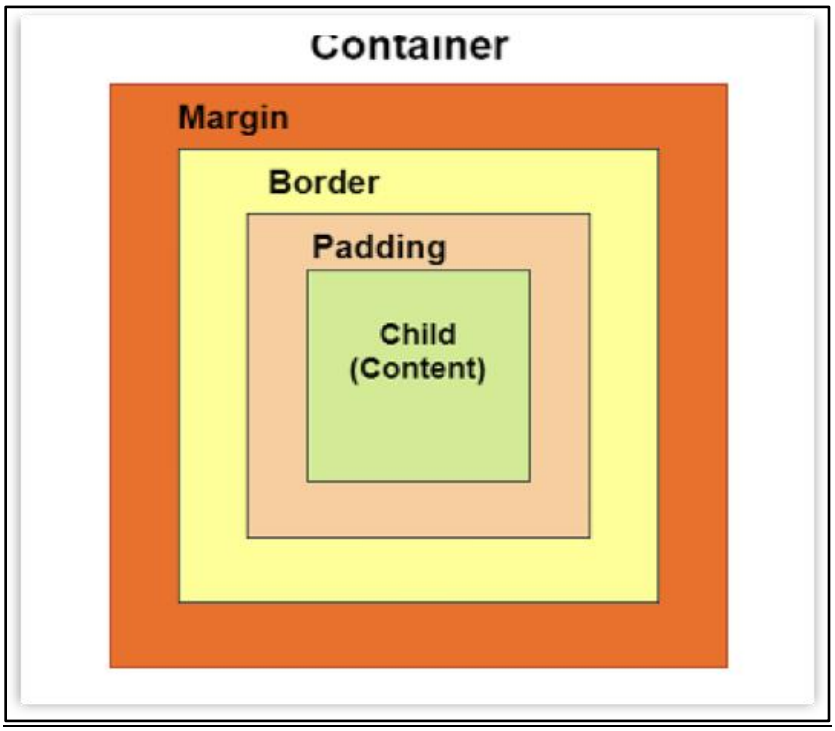


Figure 5.1: `Container` widget

5.1.1 Properties of `Container` widget

Following are the properties of the `Container` widget:

1. **`child`:** This property permits to store the child widget of the container.

```
Container (child: Text ("Hey There!"),);
```

2. **`color`:** This property helps in setting the color of the background for the entire container.

```
Container (color: Colors.blue, child: Text ("Hey There!"),);
```

3. **`height` and `width`:** This property is used for setting the height and width of the container. By default, this takes the space depending on its child widget.

```
Container (width: 200.0, height: 100.0, color: Colors.blue,
child: Text ("Hey There!"),);
```

4. **margin:** Used to cover the empty space throughout the container.

```
Container (width: 200.0, height: 100.0, color: Colors.blue,
child: Text ("Hey There!"), margin: EdgeInsets.all(10),  );
```

5. **EdgetInsets.all:** This specifies offsets in all four directions.

6. **padding:** For setting the distance between the container's border and its child widget, this property of Flutter container is used.

```
Container (width: 200.0, height: 100.0, color: Colors.blue,
child: Text ("Hey There!"), margin: EdgeInsets.all(20),
padding: EdgeInsets.all(20),  );
```

7. **alignment:** This property helps in setting the position of the child widget in the Flutter container.

```
Container (width: 200 Container (width: 200.0, height: 100.0,
color: Colors.blue, child: Text ("Hey There!"), margin:
EdgeInsets.all(20),  padding: EdgeInsets.all(20), alignment:
Alignment.bottomRight,    );
```

8. **decoration:** For adding decoration to the widget this property will be useful. This decorates or paints the widget behind the child. In the case of decorating the widget in front of the child, the foreground decoration parameter is to be utilized. When it comes to decoration property, there are many options. For instance, colors, gradients, background images, and so on.

9. **transform:** For rotating the container in any direction this property will be useful. In addition, this allows changing the coordinates of the container in the parent widget as well.

10. **constraints:** For adding extra constraints for the child widget, the constraint property is utilized. This property has different constructors

such as tight, loose, and so on.

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

5.1.2 When and How to Use a Container?

The `Container` is used when developers want common color, position, and sizing for their widgets. Any widget can be positioned or shaped according to the requirements of the user. It is possible to give margin, padding, width, height, or any kind of transformation a developer wants when using the container widget. Containers can also be used if developers want to make different decorations in the background and foreground. Figure 5.2 shows decorations in a container.

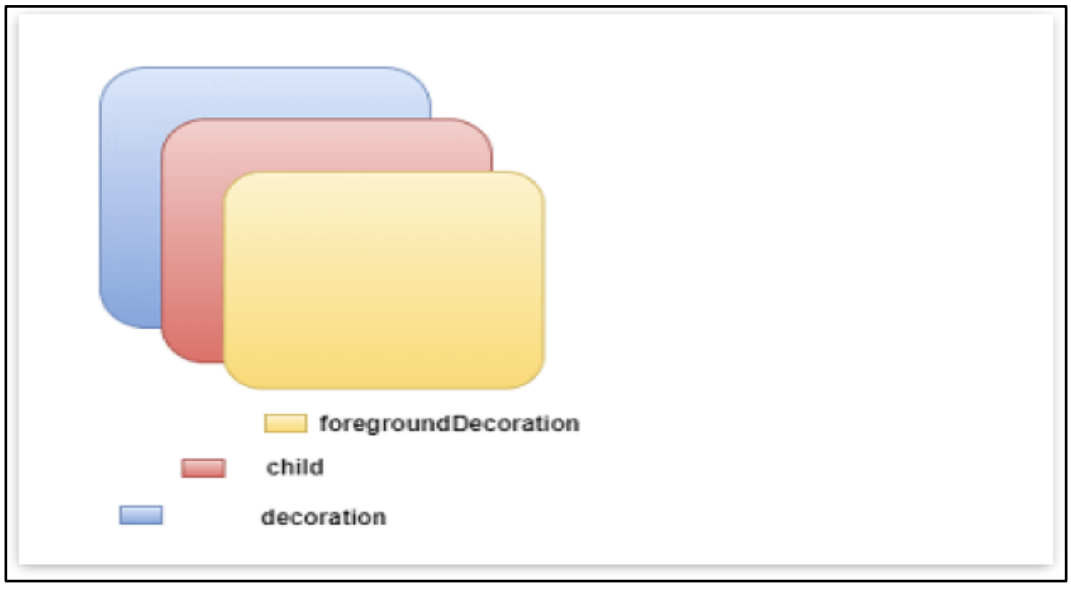


Figure 5.2: Decoration in `Container`

Code Snippet 1 is a sample code with container implementation. Code Snippet 1 demonstrates how to get the required shape and how to implement foreground and background decoration. Here, two containers are utilized.

Code Snippet 1

```
class _MyHomePageState extends State<MyHomePage> {
 @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(
               'Container Demo',
               style: Theme.of(context).textTheme.headline4,
             ),
             Container(
               decoration:
                   BoxDecoration(shape: BoxShape.circle, color:
Colors.red),
               height: 200,
             ),
             Container(
               foregroundDecoration: BoxDecoration(
                   color: Colors.blue,
                   borderRadius: BorderRadius.circular(100),
                   boxShadow: const [
                     BoxShadow(
                       color: Colors.black,
                       blurRadius: 20.0,
                     ),
                   ]),
               decoration: BoxDecoration(
                   color: Colors.red,
                   borderRadius: BorderRadius.circular(50),
                   boxShadow: const [
                     BoxShadow(
                       color: Colors.black,
                       blurRadius: 20.0,
                     ),
                   ]),
               height: 200,
             ),
```

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

Figure 5.3 shows the output when Code Snippet 1 is executed.

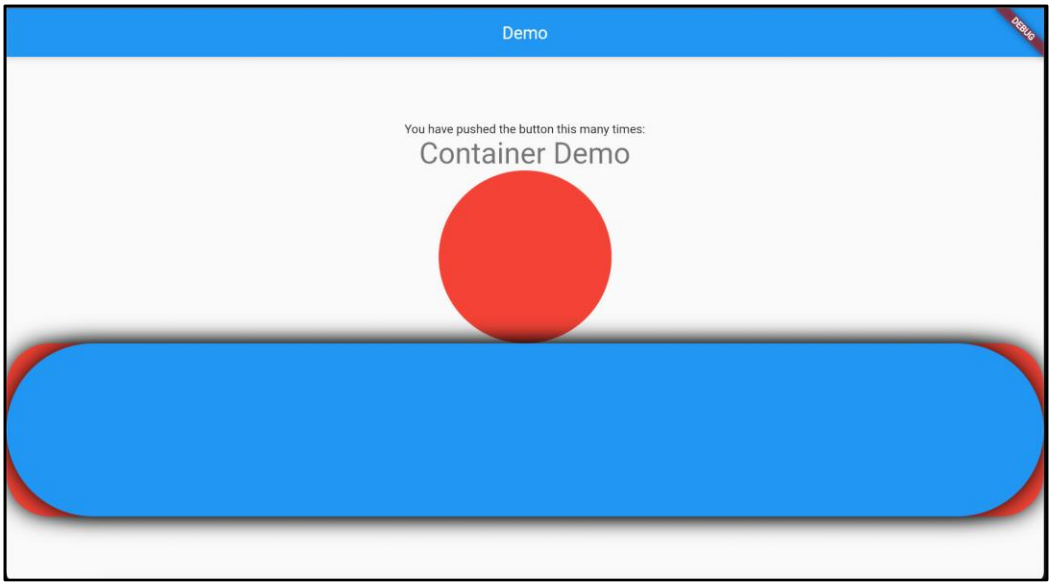


Figure 5.3: `Container` Widget Output

5.2 Padding widget

The `Padding` widget adds space between widgets. It is possible to apply padding around any widget by placing the padding widget as the child. It is a single-child layout widget.

5.2.1 Properties of `Padding` widget

1. **`EdgeInsets.all():`** Helps to apply white spaces on all sides.

   Example:

```
padding: EdgeInsets.all(15)
```

2. **`EdgeInsets.fromLTRB():`** Provides padding to individual sides. Here, LTRB refers to L – Left, T – Top, R – Right, and B – Bottom.

```
padding: EdgeInsets.fromLTRB(10, 10, 20, 25)
```

3. **`EdgeInsets.only():`** Helps to apply whitespaces to a given corner or padding to a specific side of the widget.

   Example:

```
padding: EdgeInsets.only(left: 20, top:30)
```

5.2.2 When and How to Use a Padding widget?

`Padding` can be used whenever the user wants to add some extra space around any widget. There are several types of padding available and it is possible to choose which type to be implemented based on the user's requirement.

Code Snippet 2 shows how `Padding` widgets can be utilized.

Code Snippet 2

```
class _MyHomePageState extends State<MyHomePage> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
       appBar: AppBar(
         title: Text(widget.title),
       ),
       body: Center(
         child: Column(
           mainAxisAlignment: MainAxisAlignment.center,
           children: <Widget>[
             Text(
               'Padding Demo',
               style: Theme.of(context).textTheme.headline4,
             ),
             Padding(
                 padding: const EdgeInsets.all(10),
                 child: Container(
                   foregroundDecoration: BoxDecoration(
                       color: Colors.blue,
```

```
                borderRadius:
            BorderRadius.circular(100),
                boxShadow: const [
                  BoxShadow(
                    color: Colors.black,
                    blurRadius: 20.0,
                  ),
                ]),
            decoration: BoxDecoration(
                color: Colors.red,
                borderRadius: BorderRadius.circular(50),
                boxShadow: const [
                  BoxShadow(
                    color: Colors.black,
                    blurRadius: 20.0,
                  ),
                ]),
            height: 20,
          )),
        Padding(
           padding: const EdgeInsets.fromLTRB(0, 1, 40,
           0),
            child: Container(
              foregroundDecoration: BoxDecoration(
                color: Colors.blue,
                borderRadius:
           BorderRadius.circular(100),
                boxShadow: const [
                  BoxShadow(
                    color: Colors.black,
                    blurRadius: 20.0,
                  ),
                ]),
            decoration: BoxDecoration(
                color: Colors.red,
                borderRadius: BorderRadius.circular(50),
                boxShadow: const [
                  BoxShadow(
                    color: Colors.black,
                    blurRadius: 20.0,
                  ),
                ]),
            height: 20,
          )),
        Padding(
            padding: const EdgeInsets.only(left: 50),
```

```
                child: Container(
                  foregroundDecoration: BoxDecoration(
                      color: Colors.blue,
                      borderRadius:
                   BorderRadius.circular(100),
                      boxShadow: const [
                        BoxShadow(
                          color: Colors.black,
                          blurRadius: 20.0,
                        ),
                      ]),
                  decoration: BoxDecoration(
                      color: Colors.red,
                      borderRadius: BorderRadius.circular(50),
                      boxShadow: const [
                        BoxShadow(
                          color: Colors.black,
                          blurRadius: 20.0,
                        ),
                      ]),
                  height: 20,
                )),
            ],
          ),
        ));
  }
}
```

Figure 5.4 displays the output when Code Snippet 2 is executed.

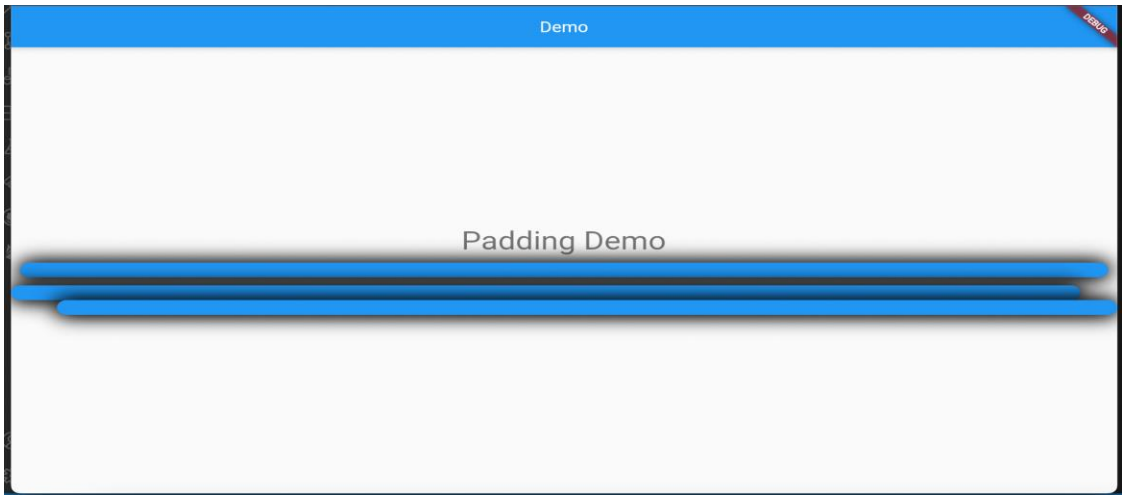


Figure 5.4: `Padding` Widget Output

5.3 Align Widget

`Align` widget as the name suggests, is utilized for moving the widget to one side or the other. Widgets can be aligned left, right, or in any other direction.

5.3.1 Properties of `Align` widget

Properties of Align widget are:

1. **`key`** - It is a unique key to control the widget.
2. **`alignment`** – Used for setting the type of alignment.
3. **`widthFactor`** – Used for setting the width for the align widget.
4. **`heigthFactor`** – Used to set the height for the align widget.
5. **`child`** - defines the child of the `Align` widget.

Figure 5.5 shows the properties of the `Align` widget.

```
Align(
  {
  Key key, //- It will control how to widget replaces the other one.
AlignmentGeometry alignment: Alignment.center, //- We can set the alignment
double widthFactor, //- Sets the width
double heightFactor, //- Sets the height
Widget child
        }
      )
```

Figure 5.5: Properties of `Align` Widget

5.3.2 When and How to Use `Align` widget?

`Align` widget is utilized to align its child widget to the defined alignment. If a particular widget must be aligned to the bottom of the screen, the align widget can be utilized for that operation to be performed.

5.4 `Center` Widget

`Center` widget aligns its child widgets to the center of the screen. If any kind of widget is inside the center widget, that widget will also be in the center of the screen

Figure 5.6 shows the properties and the description for each property.

```
Center(
  {
    Key key,
double widthFactor,//This property takes a double value as a parameter and it sets the Center widget's width
same as the child's width multiplied by this factor.

double heightFactor, //This property also takes in a double value as a parameter and it sets the Center widge
height as the same as the child's height multiplied by this factor.

Widget child
       }
      )
```

Figure 5.6: Properties of Center Widget

5.4.1 When and How to Use the `Center` widget?

If child widgets must be aligned to the center, the center widget can be utilized for that.

5.5 SizedBox Widget

The `SizedBox` widget is used for sizing widgets that are assigned as a child. A `SizedBox` widget is termed as a box with a specific size. Similar to the container or other widgets, the color for this widget cannot be set.

Syntax:

```
SizedBox({ Key key, this.width, this.height, Widget child })
```

Properties:

`Sizedbox` takes width and height as parameters.

1. **`this.width`**: Width will be applied to the child.
2. **`this.height`**: Height will be applied to the child.

Figure 5.7 shows the properties of the `SizedBox` widget.

Figure 5.7: Properties of SizedBox Widget

5.5.1 When and How to Use SizedBox?

`SizedBox` is used to provide some extra space. If the user wants to give some defined space in the form of a box, they can use `SizedBox` for that. It has an optional parameter called child that is utilized for adding a child widget.

5.6 Summary

- `Container` and `Padding` are single-child widgets.
- `Container` help in customizing painting, positioning, and sizing.
- `Padding` helps in adding some extra space to its children.
- It is possible to assign padding to all edges, one of the edges, or symmetric edge according to the requirements of the user.
- `Align` widget helps to align its child widget in the required position.
- `Center` widget helps to align its child widget to the center.
- `SizedBox` helps to give some space in the form of a box.

5.7 Test Your Knowledge

1. Shadow can be defined with _____.
   a. `Container`
   b. `Padding`
   c. Every widget
   d. `Text`

2. Which of the following is a property of `Container`?
   a. `padding`
   b. `shadow`
   c. `height`
   d. All of these

3. With decoration it is possible to _____.
   a. Arrange the size of widget
   b. Assign border to widget
   c. Assign children to Container
   d. Assign color to widget

4. Padding is _____.
   a. Single-child widget
   b. Multi-child widget
   c. Property of Container
   d. Appending widget to container

5. `widthFactor` of Align widget helps to _____.
   a. Align according to width described
   b. Assign width of its child
   c. Give space upto given width
   d. Remove the space given to width

Answers - Test Your Knowledge

1. `Container`
2. All of these
3. Assign border to widget
4. Single-child widget
5. Align according to width described

Try It Yourself

1. Create a Flutter application which has a Container of any color and two Sliders.
    a. First slider value will decide the size of the Container.
    b. Second slider value will modify the margin of the Container.
    Note: Make sure no overflow error occurs

2. Refer to the design in the given link and create the design in Flutter. Make use of `Container`, `Padding`, `SizedBox`, and other widgets.
    https://dribbble.com/shots/15865091-The-Brainbob-mobile-app

Session 6
Building a Flutter Application Using Multi Child Layout Widgets

Learning Objectives

In this session, students will learn to:

- Explain Multi Child Layout widgets
- Define and explain `Row` widget, `Column` widget, `Stack` widget, and other child layout widgets

Session Goals:

This session covers in detail the widgets that can have multiple children. Further, explains `Row` widget, `Column` widget, `Stack` widget, and other child layout widgets. Multi-Child Layout widgets are those widgets inside which one can define as many widgets as required at a time. In Flutter, for every application, there can be a widget tree that shows parent and children widgets. In the widget tree, the parent represents the widget that is on the top, whereas the child represents the widget that is assigned as a child of any widget.

6.1 Row Widget

`Row` widget displays their widgets in a horizontal array. If a few widgets are to be aligned in the same line, those are placed as children of the `Row` widget.

6.1.1 Properties of Row Widget

Some of the properties of the `Row` widget are as follows:

- `key` - Key is used to identify uniquely.
- `mainAxisAlignment` - It defines the alignment type of the x-axis.

- `mainAxisSize` - It defines the size of the x-axis.
- `crossAxisAlignment` - It defines the alignment type of the y-axis.
- `textDirection` - As the name defines, it helps in directing the text.
- `verticalDirection` - Defines the order to lay children out.
- `textBaseline` - Defines baseline.
- `children` - Defines children inside this widget.

Refer to Figure 6.1 which shows the built-in definition of properties of `Row` widget.

```
Row({
  Key? key,
  MainAxisAlignment mainAxisAlignment = MainAxisAlignment.start,
  MainAxisSize mainAxisSize = MainAxisSize.max,
  CrossAxisAlignment crossAxisAlignment = CrossAxisAlignment.center,
  TextDirection? textDirection,
  VerticalDirection verticalDirection = VerticalDirection.down,
  TextBaseline? textBaseline, // NO DEFAULT: we don't know what the text's baseline should be
  List<Widget> children = const <Widget>[],
}) : super(
  children: children,
  key: key,
  direction: Axis.horizontal,
  mainAxisAlignment: mainAxisAlignment,
  mainAxisSize: mainAxisSize,
  crossAxisAlignment: crossAxisAlignment,
  textDirection: textDirection,
  verticalDirection: verticalDirection,
  textBaseline: textBaseline,
);
```

Figure 6.1: Properties of `Row` Widget

Code Snippet 1 shows the usage of a `Row` widget. In this code, three text widgets will be used and space between them will be assigned evenly. Figure 6.2 shows the output of Code Snippet 1.

Code Snippet 1

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

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

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

```
 @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> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           const Text(
             'Row Demo',
           ),
           Row(
             crossAxisAlignment: CrossAxisAlignment.start,
             mainAxisAlignment: MainAxisAlignment.spaceEvenly,
             children: [
               const Text(
                 'Demo 1',
               ),
               const Text(
                 'Demo 2',
               ),
               const Text(
                 'Demo 3',
```

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

Output:

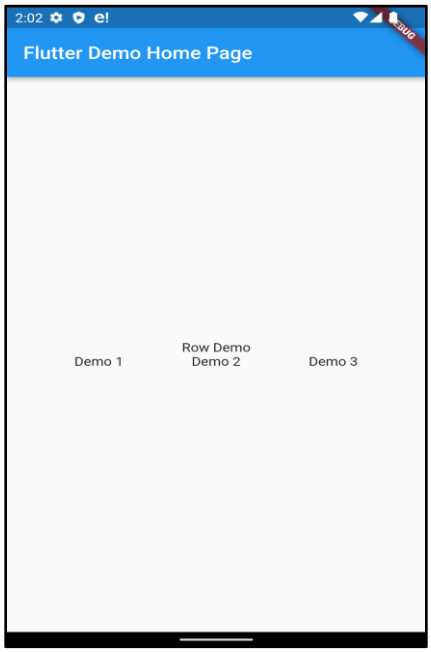


Figure 6.2: Row Widget Example

6.2 Column Widget

A `column` widget is used to display its widgets in a vertical array. If each widget is to be assigned to its child such that each widget comes in a new line, then `Column` widget is used. In simple words, rows are similar to people standing next to each other whereas, columns are similar to people standing one behind another as a Queue.

6.2.1 Properties of Column Widget

`Column` widget has same properties as `Row` widget. It behaves in the same way, to bring children into a new line. Refer to Figure 6.3 which shows the built-in definition of properties of `Column` widget.

```
Column({
  Key? key,
  MainAxisAlignment mainAxisAlignment = MainAxisAlignment.start,
  MainAxisSize mainAxisSize = MainAxisSize.max,
  CrossAxisAlignment crossAxisAlignment = CrossAxisAlignment.center,
  TextDirection? textDirection,
  VerticalDirection verticalDirection = VerticalDirection.down,
  TextBaseline? textBaseline,
  List<Widget> children = const <Widget>[],
}) : super(
  children: children,
  key: key,
  direction: Axis.vertical,
  mainAxisAlignment: mainAxisAlignment,
  mainAxisSize: mainAxisSize,
  crossAxisAlignment: crossAxisAlignment,
  textDirection: textDirection,
  verticalDirection: verticalDirection,
  textBaseline: textBaseline,
);
}
```

Figure 6.3: Properties of `Column` Widget

Code Snippet 2 shows usage of `Column` widget. In this example, three child widgets will be used to demonstrate columns. Figure 6.4 shows the output for the `Column` widget.

Code Snippet 2

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

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

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);
 @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> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           const Text(
             'Row Demo',
           ),
           Column(
             crossAxisAlignment: CrossAxisAlignment.start,
             mainAxisAlignment: MainAxisAlignment.spaceEvenly,
             children: [
               const Text(
                 'Demo 1',
               ),
               const Text(
                 'Demo 2',
               ),
               const Text(
                 'Demo 3',
               ),
             ],
           )
         ],
       ),
     ),
   );
 }
}
```

Output:

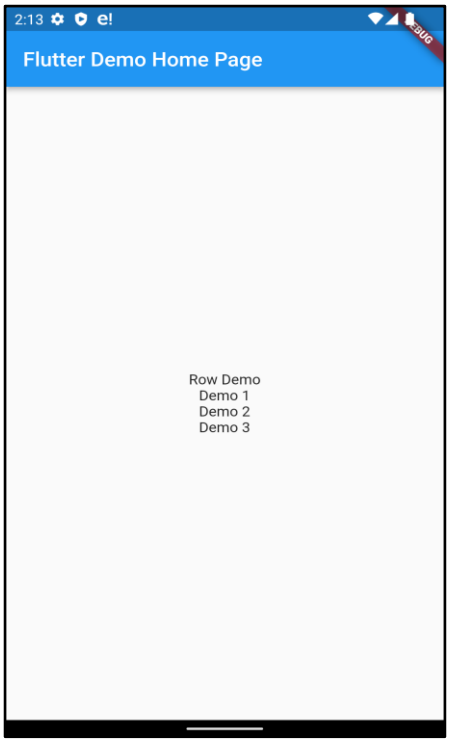


Figure 6.4: `Column` Widget Example

6.3 Stack Widget

A `Stack` widget is a multi-child widget whose children overlap one another. It is used when different widgets are to be shown over one another.

6.3.1 Properties of Stack Widget

Following are the properties of `Stack` widget:

- `alignment`: Determines the widget placed especially Non-positioned in Stack.
- `textDirection`: Change the direction of the text from Left to Right (LTR) or RTL (Right to Left).
- `fit`: The size of the non-positioned widget in Stack. It has other basic three properties:
    - `StackFit.loose`: example: 10*20 => 0 to 10 width and 0 to 20 height. – Non- positioned.
    - `StackFit.expand`: example: 10*20 => 10 will be width and 20 will be height
    - `StackFit.passthrough`: Remains unmodified.

- `overflow`: It will manage to overflow the content of the child widget. Clipped or visible.

Refer to Figure 6.5 which shows the built-in definition of properties of `Stack` widget.

Figure 6.5: `Stack` Widget Properties

Code Snippet 3 demonstrates how to use the `Stack` widget. Children of `Stack` will overlap as there would not be any gap applied. `Row` or `Column` widgets can be used when widgets should be placed beside or below. If widgets in Stack are to be positioned in a certain way, then `Align` or `Positioned` widgets can be used.

Code Snippet 3

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

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

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);
 @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> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           const Text(
             'Row Demo',
           ),
           Stack(
             children: [
               const Text(
                 'Demo 1',
                 style: TextStyle(fontSize: 20),
               ),
               const Text(
                 'Demo 2',
                 style: TextStyle(fontSize: 30),
               ),
               const Text(
                 'Demo 3',
                 style: TextStyle(fontSize: 40),
               ),
             ],
           )
         ],
       ),
     ),
   );
 }
}
```

Figure 6.6 shows the output of widget of Code Snippet 3. Observe how the children elements are all stacked upon one another, causing an overlap.

Output:

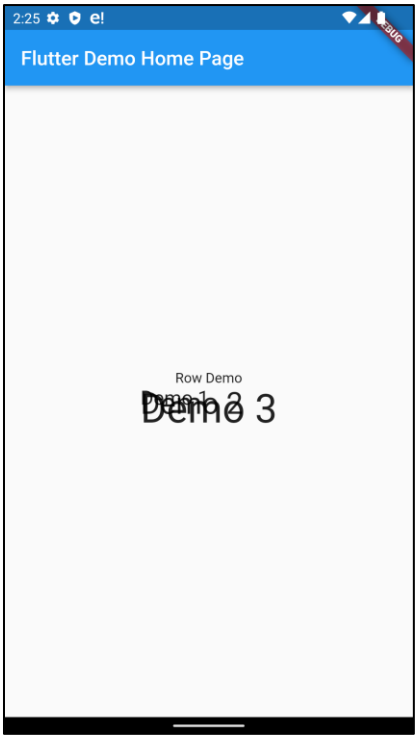


Figure 6.6: `Stack` widget Example

Here, each of the children in the Stack are `Text` widgets. However, in practical scenarios, one can use `Containers`, which in turn can have colors and/or text. Figure 6.7 shows an example of a `Stack` widget using `Container` widgets of different shapes and colors stacked on top of each other.

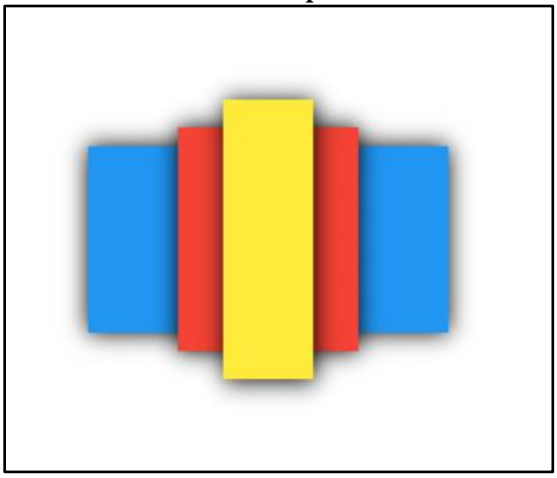


Figure 6.7: Example of `Stack` Widget Using Containers

6.4 ListView Widget

`ListView()` is similar to the `Column` widget with Scroll feature. It takes a list of widgets as children and make them scrollable. It renders only items that are visible.

`ListView.builder()`: The builder constructor simplifies the repeating list of items. This can be used when the list of children are of the same widget. It takes two main parameters:

- `itemCount`: Specify the number of items in the list.
- `itemBuilder`: Specify the widget to be built. This widget will be built as per `itemCount`.

6.4.1 Properties of ListView

Following are some of the properties of `ListView`:

- **`childrenDelegate`:** `SliverChildDelegate` is provided as delegate to serve the `ListView`.
- **`clipBehaviour`:** The clipping of the content is controlled by this property.
- **`itemExtent`:** Controls the scrollable area is the `ListView`.
- **`padding`:** Provides space between `ListView` and its children.
- **`scrollDirection`:** Defines the direction of the Scroll in `ListView`.
- **`shrinkWrap`:** Takes in a boolean value which in turn decides whether the scrollable space is determined by the children inside the `ListView`.

Refer to Code Snippet 4 for `ListView` example. Figure 6.8 shows the output for Code Snippet 4.

Code Snippet 4:

```
ListView.builder(
   itemCount: 10,
   itemBuilder: (BuildContext context, int index) {
     return ListTile(
         leading: const Icon(Icons.account_circle, size: 40,),

         title: Text("User ${index+1}", style:
```

```
TextStyle(fontWeight: FontWeight.w600),));
   }),
```

Output:

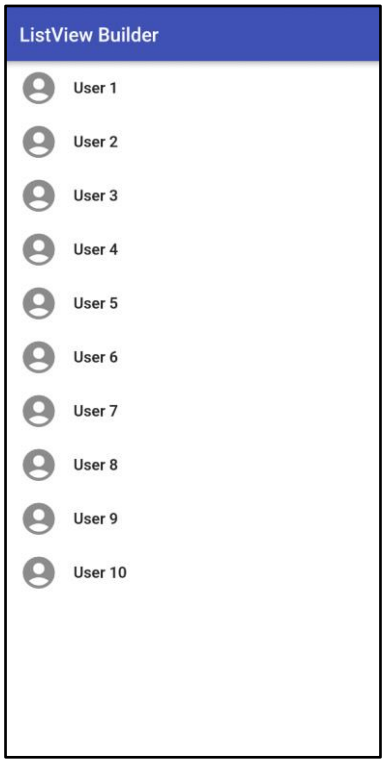


Figure 6.8: `ListView.builder` Widget Output

6.5 GridView Widget

`GridView` is similar to `ListView`. However, it displays the data in two-dimensional rows and columns. It is similar to how products are viewed in a shopping application.

To display data from `GridView`, the most commonly used widget is `GridView.count()` and `GridView.builder()` when the user wants to show the data dynamically.

6.5.1 Properties of `GridView`

Following are the properties of `GridView`:

- **`itemCount`**: Number of data to be displayed.

- **gridDelegate**: Provides the delegate property to the `GridView` similar to spacing and aspect ratio between widgets.
- **itemBuilder:** Widget to be built and displayed.
- **primary:** Primary will take true or false values and will define whether scrolling is associated with a parent or not.
- **crossAxisSpacing:** Specifies the spacing between cross axes.
- **mainAxisSpacing:** Specifies the spacing between the main axes.
- **crossAxisCount**: Specifies the number of data in the cross-axis.

Refer to Code Snippet 5 for `GridView` example. Figure 6.9 is the output for Code Snippet 5.

Code Snippet 5:

```
GridView.builder(
 gridDelegate:
SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 2,
mainAxisSpacing: 10, crossAxisSpacing: 10),
   itemCount: 5,
   itemBuilder: (BuildContext context, int index) {
     return Card(
       color: Colors.blueGrey,
       child: Center(child: Text('User ${index+1}', style:
TextStyle(
         fontWeight: FontWeight.w700,
         fontSize: 25,
         color: Colors.white
       ),)),
     );
   }),
```

Output:

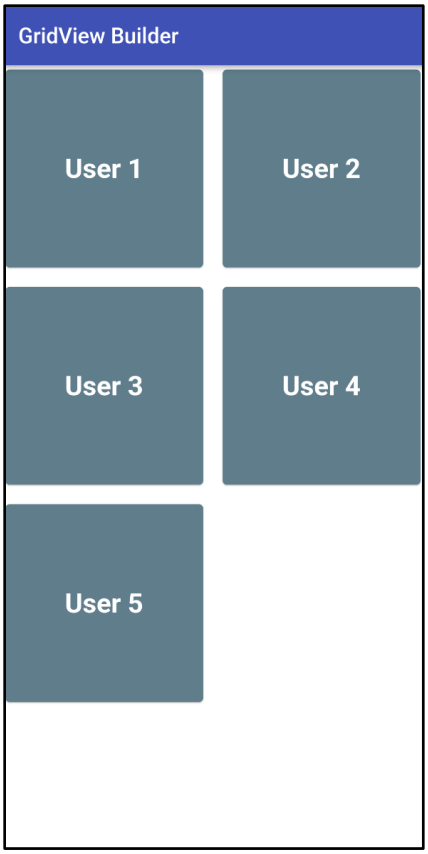


Figure 6.9: `GridView.builder` Widget Output

6.6 Expanded Widget

`Expanded` widget as the name explains expands a child of `Row` or `Column` or `Flex` so that it will occupy the available space.

For example, when you are working with `Row` widget or `Column` widget and using `mainAxisAlignment.spaceBetween`, it leaves a gap between children. In that case, this widget can be used to fill the available space.

`Flex` and `Child` are the properties of `Expanded` widget. Using Flex expansion the widgets can be controlled. The `Child` is used to hold another widget when multiple widgets are expanded.

6.7 Table Widget

`Table` widget displays the items in a table layout, hence it is not necessary to use rows and columns to create a table. Using the `Table` widget is the appropriate method when there is the same column width for multiple rows.

`columnWidths` and `defaultColumnWidth` are the properties of the Table widget. `columnWidth` helps in specifying the specific column width. `defaultColumnWidth` is used to specify default width of the column. Refer to Code Snippet 6. Figure 6.10 shows the output for Code Snippet 6.

Code Snippet 6:

```
Table(

 // textDirection: TextDirection.rtl,
 // defaultVerticalAlignment:
TableCellVerticalAlignment.bottom,
 // border:TableBorder.all(width: 2.0,color: Colors.red),
 children: [
   TableRow(
       children: [
         Text("Name",textScaleFactor: 1.5, style:
TextStyle(fontWeight: FontWeight.w600),),
         Text("Subject",textScaleFactor: 1.5, style:
TextStyle(fontWeight: FontWeight.w600),),
         Text("Marks",textScaleFactor: 1.5, style:
TextStyle(fontWeight: FontWeight.w600),),
       ]
   ),
   TableRow(
       children: [
         Text("Alex",textScaleFactor: 1.5),
         Text("Maths",textScaleFactor: 1.5),
         Text("98",textScaleFactor: 1.5),
       ]
   ),
   TableRow(
       children: [
         Text("Henry",textScaleFactor: 1.5),
         Text("Chemistry",textScaleFactor: 1.5),
         Text("35",textScaleFactor: 1.5),
       ]
   ),
   TableRow(
       children: [
         Text("Luke",textScaleFactor: 1.5),
         Text("Physics",textScaleFactor: 1.5),
         Text("40",textScaleFactor: 1.5),
       ]
   ),
```

```
  ],
),
```

Output:

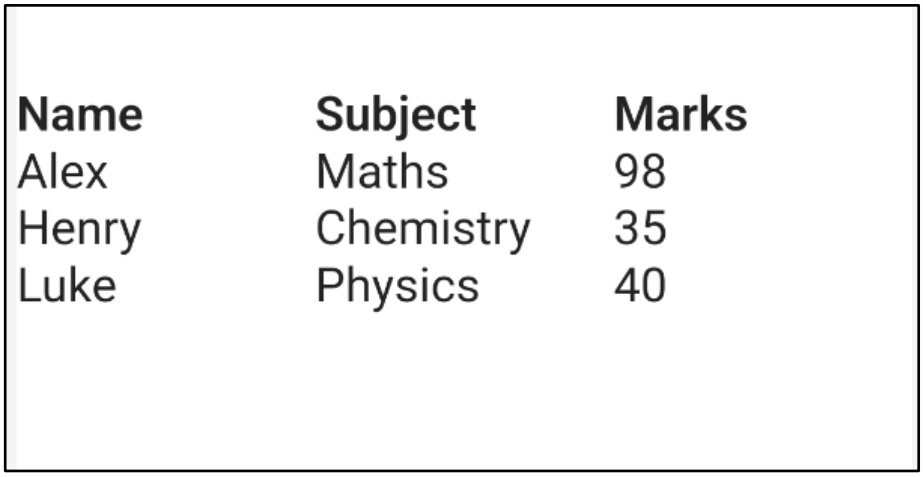


Figure 6.10: `Table` Widget Output

6.8 Summary

- `Row` widget is used to show its children in a single line.
- `Column` widget is used to show each child in a new line.
- `Stack` is used for overlapping one widget over another.
- `ListView` gives us a scrollable list of widgets that are arranged linearly.
- `GridView` widget in Flutter is used to display the data in two-dimensional rows and columns.
- `Expanded` widget is used to expand its child up to available space.
- `Table` widget is used to display items in a table layout.

6.9 Test Your Knowledge

1. Which property defines the order to lay children out?
   a. `verticalDirection`
   b. `children`
   c. `style`
   d. `order`

2. Which of the following is a property of `Table`?
   a. `defaultColumnWidth`
   b. `Shadow`
   c. `height`
   d. All of these

3. Which widget can show data in 2D?
   a. `Table`
   b. `ListView.builder`
   c. `GridView`
   d. None of these

4. Which widget is used to show data dynamically?
   a. `ListView.builder`
   b. `GridView.builder`
   c. Both
   d. None of these

5. Which property controls the scrolling area in `ListView`?
   a. `itemExtent`
   b. `childCount`
   c. `padding`
   d. None of these

Answers - Test Your Knowledge

1. `verticalDirection`
2. `defaultColumnWidth`
3. `GridView`
4. Both
5. `itemExtent`

6.10 Try It Yourself

1. Create a Flutter application for storing Customers data in the form of a Table. The application will contain following features:
    a. Home Screen will have a Table where it shows the Customer data in the form of a Table.
    Table will have following Columns: Serial Number, Customer Name, and their Age.
    b. The Home Screen will have a `FloatingActionButton` which on clicking will navigate to a new screen where Customer Details are entered. The screen will take Customer's Name and Age as input.
    c. Have an 'Add Customer' button in the end of the Page. On clicking that, the Provided Customer data should be added to the Table and Navigate back to Home Screen.

    Hint: Store all the Customers data in a List variable.

2. Design an Online Shopping application. There should be one screen where all the available products are displayed as a Grid (The products can be anything of your choice). Each product in the Grid should have an image, price, and `wishlist` button.

Session 7
State Management and Dependency Injection in Flutter

Learning Objectives

In this session, students will learn to:

- Describe State Management in Flutter
- Explain State Management types in Flutter
- Define Dependency Injection in Flutter
- Outline example of dependency injection in Flutter

Session Goals:

This session explains state management and dependency injection in Flutter.

7.1 State Management in Flutter

State management is the most important concept in Flutter Application Development. Managing state means managing data of the application in runtime.

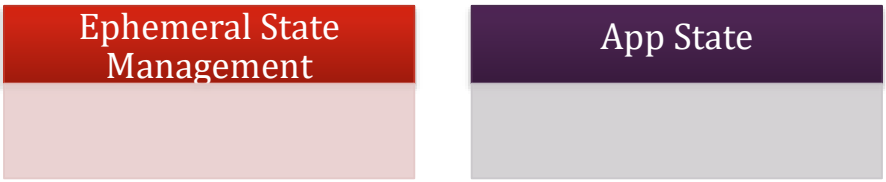
In Flutter, there are two types of widgets namely Stateful and Stateless widgets. For a stateful widget, the data can be changed in runtime by managing the states. In the stateless widget, the state of the widget once built cannot be changed. Hence, there is no state management for the stateless widgets.

Following is an example to understand the concept of State Management:

Assume that an application for selling something is being built. On the item page, it is possible to click the **Add to cart** button and use the ‘add' or 'subtract’ icon to increase or decrease the quantity of any item respectively. State management helps to achieve this increase and decrease. Further, in the example discussed, it is important to maintain cart data across several purchases or browsing and this is also achieved with the help of state management. The information that is used to add the list of items in cart is stored in a separate class. This class is used to keep track of all the activities.

7.2 State Management types in Flutter

Based on usability, state management is classified into two types:



7.2.1 Ephemeral State Management

In ephemeral state management, the management of state lasts only for a few seconds. For example, when user clicks **Add to cart**, the button style may change to the ‘add' and 'subtract’ button with the number of items displayed to the side. This state management where the change from the **Add to cart** to ‘add' and 'subtract’ happens in just a few seconds is a good example to showcase Ephemeral State Management.

Stateful Widget is another example of this type of State Management. In the stateful widget, the current state of the widget is known and if any change must be made to the state, the value is altered. This alteration takes only a fraction of a second and is easy to implement.

7.2.2 App State

In App state management, the state management lasts for the entirety of the application or throughout the scope of the application. For example, the data in the cart has to be maintained throughout the application irrespective of the page the user lands in. There are many ways to achieve this, but

`scoped_model` is the best way to implement and track App State because it is easy and quick to use. App state consists of three major classes.

Following are the major classes for App State:

`Model`	Models can be used to extend the `Model` class to listen to changes.
`ScopedModel`	`ScopedModel` is used to wrap UI components and access data from the `Model`.
`ScopedModelDescendant`	`ScopedModelDescendant` is used to identify the correct `ScopedModel` from the widget hierarchy.

7.3 Dependency Injection in Flutter

Dependency Injection in Flutter is a mechanism by which an object supplies dependency to another object. In simpler terms, it is a programming technique that makes the class independent of its dependencies. Further, dependency injection allows the creation of dependent objects outside the class and exposes those objects to the class in several ways. It also lets the user offload the creation and binding of dependent objects from classes. Lastly, dependency injection provides easier testing, a higher level of flexibility, and isolation.

7.4 Creating an Application with Dependency Injection in Flutter

Dependency Injection in an application can be achieved in two ways:

1. Inherited Widget
2. Provider Package

7.4.1 Dependency Injection using Inherited Widget

Dependency Injection using inherited widget allows the data down the widget tree easily. This means data can be transferred easily from parent widget to child widget.

In Code Snippets 1 and 2, `InheritedWidget` class is used to create a widget and Text.

Code Snippet 1: `main.dart`

```
import 'package:flutter/material.dart';
import 'InheritedWidget.dart';
void main() {
 runApp(const MyApp());
}
class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
   return InheritedHomeWidget(
       title: 'Dependency Injection Text',
       key: key,
       child: MaterialApp(
         title: 'Flutter Demo',
         theme: ThemeData(
           primarySwatch: Colors.blue,
         ),
         home: const MyHomePage(title: 'Demo'),
       ));
 }
}
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> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(InheritedHomeWidget.of(context).title),
     ),
     body: Column(
       mainAxisAlignment: MainAxisAlignment.start,
       crossAxisAlignment: CrossAxisAlignment.center,
       children: const <Widget>[
         Text(
           'Dependency Injection Demo',
         )
       ],
     ),
```

```
    );
  }
}
```

Code Snippet 2: `InheritedWidget.dart`

```
import 'package:flutter/material.dart';
class InheritedHomeWidget extends InheritedWidget {
 const InheritedHomeWidget(
     {Key? key, required this.title, required this.child})
     : super(key: key, child: child);
 final Widget child;
 final String title;
 static InheritedHomeWidget of(BuildContext context) {
   return
(context.dependOnInheritedWidgetOfExactType<InheritedHomeWidget
>()
       as InheritedHomeWidget);
 }
 @override
 bool updateShouldNotify(InheritedHomeWidget oldWidget) {
   return true;
 }}
```

In this example, an `InheritedWidget` class called `InheritedHomeWidget` is created which is used in the `HomePage` widget. Here, data from the Inherited widget is available for `HomeWidget`. Figure 7.1 depicts the output.

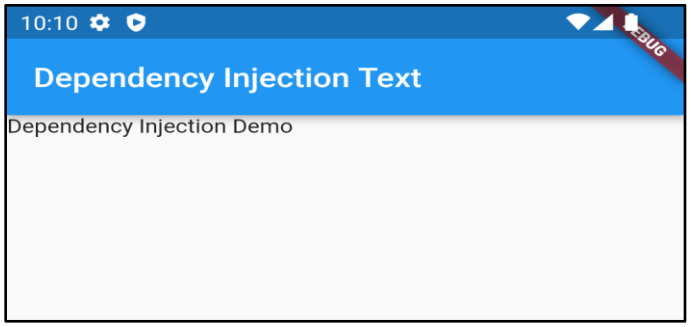


Figure 7.1: Output of Dependency Injection

7.4.2 Dependency Injection Using Provider package in Flutter

For dependency injection using the provider package, one has to add a provider package to the Flutter project. Add `provider: ^6.0.4` under `dependencies` in the file `pubspec.yaml` and then, run `flutter pub get`

to add Provider package to Flutter project. Then, this package has to be imported in files where it will be used.

In Code Snippets 3 and 4, a text is added in the provider and subsequently, the provider will be added in the main function. Another thing to observe will be how one can access the text anytime inside the application.

Code Snippet 3: `main.dart`

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'package:works/api.dart.dart';

void main() {
 Provider.debugCheckInvalidValueType = null;
 runApp(const MyApp());
}

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
   return MultiProvider(
       providers: [Provider.value(value: Api())],
       child: MaterialApp(
         title: 'Flutter Demo',
         theme: ThemeData(
           primarySwatch: Colors.blue,
         ),
         home: const MyHomePage(title: 'Demo'),
       ));
 }
}
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> {
 @override
 Widget build(BuildContext context) {
   return MultiProvider(
     providers: [
```

```
      Provider.value(
        value: Api(),
      ),
      ProxyProvider<Api, HomeModel>(
        // Dependency injection
        update: (context, api, homeModel) => HomeModel(api:
         api),
      )
    ],
    child: MaterialApp(
      theme: ThemeData(primaryColor: Colors.red),
      home: Scaffold(
        body: Consumer<HomeModel>(
          builder: (context, model, child) => Center(
            child: Text(model.api.loggedIn),
          ),
        ),
      ),
    ),
  );
 }
}
```

Code Snippet 4: `Api.dart`

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

class Api {
 String get loggedIn => 'I am logged in';
}

class HomeModel extends ChangeNotifier {
 final Api api;
 HomeModel({required this.api});
}
```

In this example, `Api.dart` is appended with the help of a provider. This provider is a state management technique and because of this, it is possible for `main.dart` to access the String named `loggedIn`. To summarize, with the help of dependency injection, data in `Api.dart` can be accessed anywhere in the application with the help of the Provider. Figure 7.2 depicts the output.

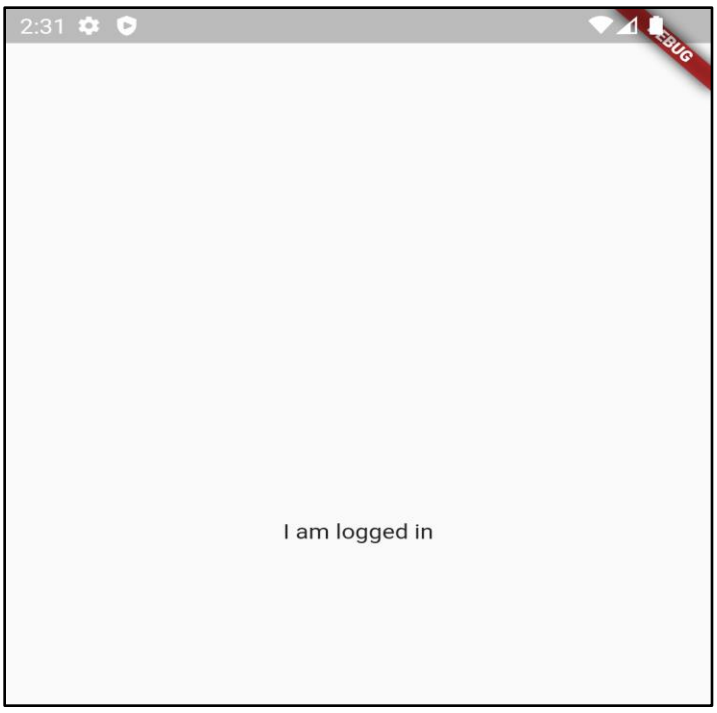


Figure 7.2: Output of Dependency Injection Using Provider Package

7.5 Summary

- State Management is a technique by which one can change views during runtime.
- Ephemeral State management and App state are two types of state management.
- Dependency Injection is a design pattern used to implement inversion of control.
- Inherited Widget and Provider Package are two ways to implement Dependency Injection.
- In provider dependency injection, it is required to include the provider library.

7.6 Test Your Knowledge

1. Which part in scoped_model is used to wrap UI components and access data from the Model?
   a. ScopedModel
   b. Model
   c. ScopedModelDescendant
   d. None of these

2. Which part of scoped_model can be used to identify the correct ScopedModel from the widget hierarchy?
   a. ScopedModel
   b. Model
   c. ScopedModelDescendant
   d. None of these

3. _____ class must be extended to listen to the changes.
   a. Model
   b. Navigator
   c. Route
   d. Provider

4. Dependency injection is _____.
   a. Notifying the value changed to its defined places
   b. A designing pattern used to implement inversion of control
   c. Used to rebuild whole UI with new values
   d. Used to access data from the application

5. How is Dependency Injection useful?
   a. Flexibility
   b. Isolation
   c. Easier Testing
   d. All of these

Answers - Test Your Knowledge

1. ScopedModel
2. ScopedModelDescendant
3. Model
4. A designing pattern used to implement inversion of control
5. All of these

7.7 Try It Yourself

1. Study various state management solutions available in Flutter such as `GetX` Provider, `BloC`, `Riverpod`, `MobX`, and so on. Understand how they differ from each other, their pros and cons.
2. Implement the Code Snippets 1 and 2 in an application using `Riverpod` state management and `BloC` state management.

Session 8
Navigation, Routing, and Controllers in Flutter

Learning Objectives

In this session, students will learn to:

- Define Navigation and Routing
- Outline Routes in Flutter
- Explain Routing class
- Describe navigation concept in Flutter
- Explain Controllers in Flutter
- Describe working with Controllers in Flutter
- Define Custom controller in Flutter

Session Goals:

Flutter supports many different pages within applications, similar to several other technologies. For navigating from one page to another, Navigation and Routing are utilized. This session introduces the navigation and explains the concepts of navigation in Flutter. Further, the session details about routing class. To control many different widgets during navigation, controllers come in handy. The session finally concludes by explaining controllers and detailing the working of controllers on Flutter.

8.1 Navigation and Routing in Flutter

In Flutter, navigation is a concept of moving to and from a Page/Screen. Routing is a mechanism to specify the route for Navigation. Routing in Flutter helps to implement navigation from one page to another.

Consider a simple real-world example to understand navigation. When a customer browses a retail shopping app/site and clicks a product, he/she is redirected to a new screen/page displaying the product details. Upon clicking a back button or link, the customer can go back to the previous page. This is navigation in action where the customer 'navigates' from one page to another.

8.2 Routes in Flutter

Every page in Flutter has a route. In the Navigator, it is possible to mention any route about the user requirements.

Code Snippet 1 demonstrates sample routes used in an application.

Code Snippet 1:

```
routes: {
       '/': (context) => const MyHomePage(title: 'Flutter
Demo'),
       '/second': (context) => const MyHomePage(title: 'Flutter
Demo'),
     },
```

8.3 Routing Class in Flutter

Route is an abstract class that is used to manage the entry to the Navigator class. The Navigator class in Flutter is a widget which manages a set of child widgets with a stack discipline. Navigator class is responsible for managing the routes between these elements. An abstract class is a class that can have abstract methods which does not have a body.

8.3.1 MaterialPageRoute

`MaterialPageRoute` is the modal route that replaces the entire screen with platform-adaptive transition. In Android, the navigation to next screen takes place by a fade animation whereas in iOS, the navigation takes place with a sliding animation. In simple words, `MaterialPageRoute` takes care of the transition between pages within the application.

Two methods are important to route from one page to another and they are as follows:

a. **`Navigator.push()`**: This method is used to push a new page to the stack.
b. **`Navigator.pop()`**: This method is used to pop the current page from the stack.

8.4 Implementing Navigation in Flutter Applications

Navigation in Flutter is used to navigate from one page to another.
Figure 8.1 shows how Routes help with Navigation. Observe that Routes are assigned to `MaterialApp`. From `MaterialApp`, users can navigate to specific pages using the Routes.

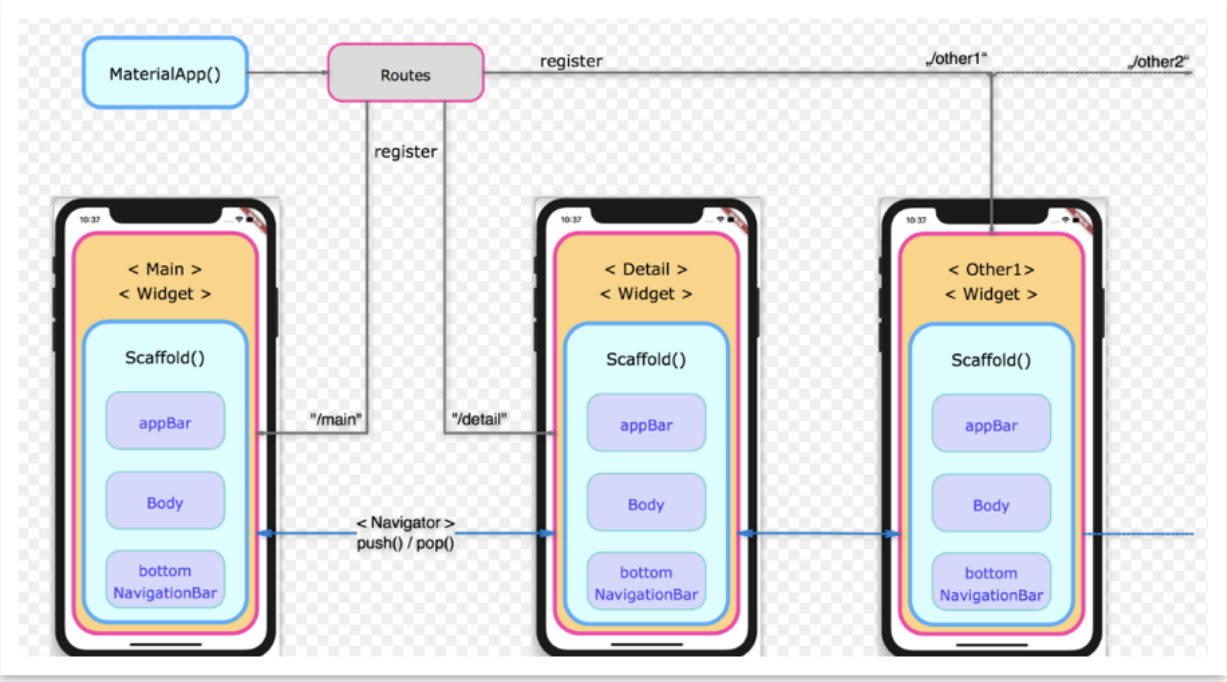


Figure 8.1: Understanding of Routes and Navigation

Refer Code Snippet 2 to get a better understanding of how routes work in Flutter.

In this code, routes are defined for `MaterialApp` and these routes help navigate from page one to page two.

Code Snippet 2:

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

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

class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     routes: {
       '/': (context) => const MyHomePage(title: 'Flutter
Demo'),
       '/second': (context) => const MyHomePage2(title:
'Flutter Demo'),
     },
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
   );
 }
}

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> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           const Text(
```

```
            'Row Demo',
          ),
          const Text(
            'Page 1',
            style: TextStyle(fontSize: 20),
          ),
          ElevatedButton(
            onPressed: () {
              Navigator.of(context).pushNamed('/second');
            },
            child: Container(
              alignment: Alignment.center,
              child: const Text('Go to page two with named'),
            ),
          ),
          ElevatedButton(
            onPressed: () {
              Navigator.push(
                  context,
                  MaterialPageRoute(
                      builder: (context) =>
                          const MyHomePage2(title: 'Flutter
Demo')));
            },
            child: Container(
              alignment: Alignment.center,
              child: const Text('Go to page two without
named'),
            ),
          ),
        ],
      ),
    ),
  );
 }
}

class MyHomePage2 extends StatefulWidget {
 const MyHomePage2({Key? key, required this.title}) :
super(key: key);
 final String title;
 @override
 State<MyHomePage2> createState() => _MyHomePageState2();
}

class _MyHomePageState2 extends State<MyHomePage2> {
```

```
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           const Text(
             'Row Demo',
           ),
           const Text(
             'Page 2',
             style: TextStyle(fontSize: 20),
           ),
           ElevatedButton(
             onPressed: () {
               Navigator.pop(context);
             },
             child: Container(
               alignment: Alignment.center,
               child: const Text('Go to page two'),
             ),
           ),
         ],
       ),
     ),
   );
 }
}
```

Output:

Figures 8.2 and 8.3 show outputs when the application is executed. Two buttons exist in the home screen. Refer to Figure 8.2. When any of these buttons are clicked, they take the user to the second page. Refer to Figure 8.3. Another button exists in the second page to enable the user to navigate to previous screen. This action of jumping from one screen to another and back is possible because of the Navigation and Routes concept. The controller for the first or the second page gets loaded when the user clicks the appropriate buttons and the router aids in the navigation from one to another.

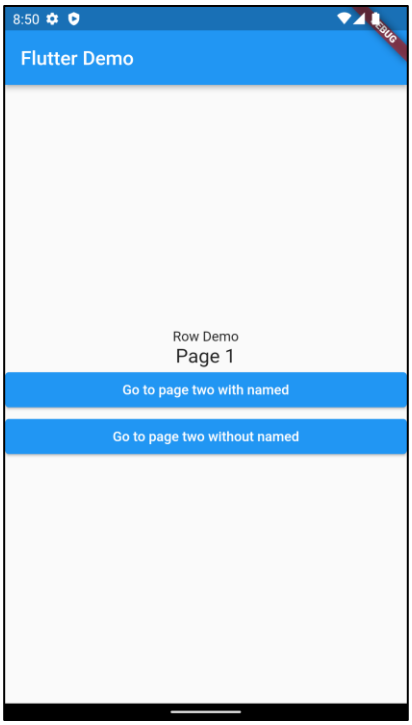


Figure 8.2: Demo Application Output Upon Code Execution

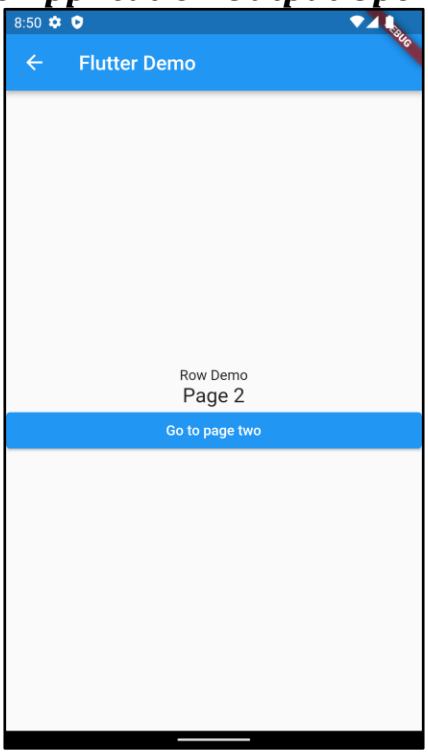


Figure 8.3: Demo Application Output After Navigation

8.5 Controllers in Flutter

Controllers in Flutter are used to control widgets. The control action is limited to properties of the widget to which the controller is applied. With controllers,

the requirement for use of global keys to get access to the widget state is avoided. Controllers can be considered as objects that can attach to certain widgets. These objects can be used to direct the actions of that widget.

8.6 Working with Controllers in Flutter

A controller can be best explained with a couple of examples.

8.6.1 TextEditingController

`TextEditingController` is used to control `TextField` or `TextFormField`. If the `TextEditingController` is applied on a `TextField` widget, it is possible to obtain the input text from the widget by using the controller.

8.6.2 ScrollController

`ScrollController` is used to control scroll view. If applied to a `ScrollView` widget, the scrollbar, scrolling type, and all other properties of that widget can be controlled using the `ScrollController`.

Code Snippet 3 shows the `MyHomePage` class from Code Snippet 2 and demonstrates how controllers work in Flutter.

Code Snippet 3:

```
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> {
 TextEditingController tv = TextEditingController();
 ScrollController sc = ScrollController();
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
```

```
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          const Text(
            'Controller Demo',
          ),
          TextFormField(
            controller: tv,
          ),
          SizedBox(
            height: 100,
            child: SingleChildScrollView(
              controller: sc,
              child: Column(children: const [
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
```

```
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
                Text(
                  'Demo',
                  style: TextStyle(fontSize: 20),
                ),
              ]),
            ),
          ),
          ElevatedButton(
            onPressed: () {
              sc.jumpTo(5);
              print(tv.text);
            },
            child: Container(
              alignment: Alignment.center,
              child: const Text('Go to page two without
named'),
            ),
          ),
        ],
      ),
    ),
  );
 }
}
```

Here, both `TextEditingController` and `ScrollController` are used. When a button is clicked, the appropriate value is printed in `TextFormField` while simultaneously, the `ScrollView` is scrolled down to the fifth index. Refer to Figures 8.4 and 8.5.

Output:

Figure 8.4: Output for Working of Controllers

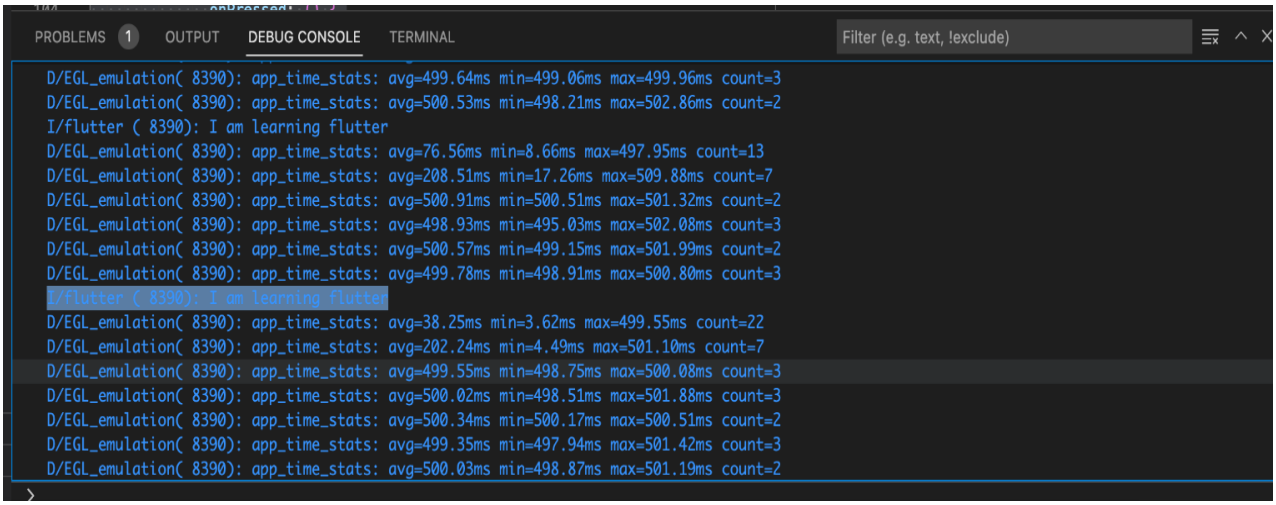


Figure 8.5: Console Output

8.7 Creating a Custom Controller in Flutter

Custom controllers are used in Flutter to control custom widgets that are also used in combination with different controllers defined in Flutter.

To create custom controllers, users can use the `changenotifier` class. The `changenotifier` is a class which can be used to notify to subclasses about any changes that happens in the parent class using the `notifyListeners` method.

8.7.1 Implementing a Custom Controller using `ChangeNotifier`

To create a custom controller, developers have to implement `ChangeNotifier` to the controller class, and on changing any data such as

changing color, shape, and size developers can notify users using `notifyListeners`.

Code Snippet 4 shows a sample `CustomController` class which contains properties such as `buttonShape`, `buttonColor`, and `buttonHeight` and methods to change them.

Code Snippet 4:

```
class CustomController extends ChangeNotifier {
  Color buttonColor = Colors.blue;
  String buttonShape = 'Rectangle';
  double buttonHeight = 10;

  void changeColor(Color color) {
    buttonColor = color;
    notifyListeners();
  }

  void changeShape(String shape) {
    buttonShape = shape;
    notifyListeners();
  }

  void changeHeight(double height) {
    buttonHeight = height;
    notifyListeners();
  }
}
```

8.8 Summary

- The `Navigator` class is used to navigate from one page to another.
- Routing is a path that users define inside the Navigation library in order to define where the application has to head to.
- The `push()` and `pop()` methods in `Navigator` class are used to change pages based on requirement.
- Controllers in Flutter are used to control any widget, get data from it, and also define properties.
- It is possible to build a custom controller in Flutter using the `changenotifier` class.

8.9 Test Your Knowledge

1. `pop()` method in `Navigator` is used to _____.
   a. Pop out current page and go back in application navigation
   b. Go to a new page
   c. Delete previous pages in stack
   d. Navigate between pages

2. `push()` method in `Navigator` is used to _____.
   a. Push new activity in stack and go to new activity
   b. Pop out current page and go back in application navigation
   c. Go to a new page
   d. Notify on new activity

3. `notifyListeners` is a method of _____ class.
   a. `changeNotifier`
   b. `Navigator`
   c. `Route`
   d. `ScrollController`

4. `changeNotifier` is used for _____.
   a. Notifying the value changed to its defined places
   b. Pushing new activity in stack and go to new activity
   c. Rebuilding whole UI with new values
   d. Notifying the new activity

5. `MaterialPageRoute` is _____.
   a. To change whole page
   b. To define route
   c. To navigate between pages
   d. To control the routes

Answers - Test Your Knowledge

1. Pop out current page and go back in application navigation
2. Push new activity in stack and go to new activity
3. `changeNotifier`
4. Notify the value changed to its defined places
5. To change whole page

8.10 Try It Yourself

1. Create a new Flutter project with three Separate Stateless widgets (Screens) called Page1, Page2, and Page3. Page1 will be the initial page when the app is launched. All three Screens will have a different solid color background and an `ElevatedButton` in the center of the screen.
    a. Page 1 button will have the text 'Navigate to Page 2' and on clicking that should move to Page 2.
    b. Page 2 button will have the text 'Navigate to Page 3' and on clicking that should move to Page 3.
    c. Page 3 button will have the text 'Navigate back to Page 1' and on clicking, that should move back to Page 1.

2. Create a new Flutter project with a Screen that has a Circle-shaped `Container` with initial radius 20, a `TextField`, and a `TextButton` named 'Update'.
    a. Make use of a custom controller class with `ChangeNotifier` to store the container radius, `TextEditingController` and button function.
    b. The `TextField` will take radius as input. Upon clicking the update button, the radius of the container should change to the specified radius.
    c. Handle all validations. For example, the `TextField` should take only numbers as input and values should be between 0 and 200. All the properties and methods should be inside the custom controller class.

Session 9
Styling and Creating Responsive Applications in Flutter

Learning Objectives

In this session, students will learn to:

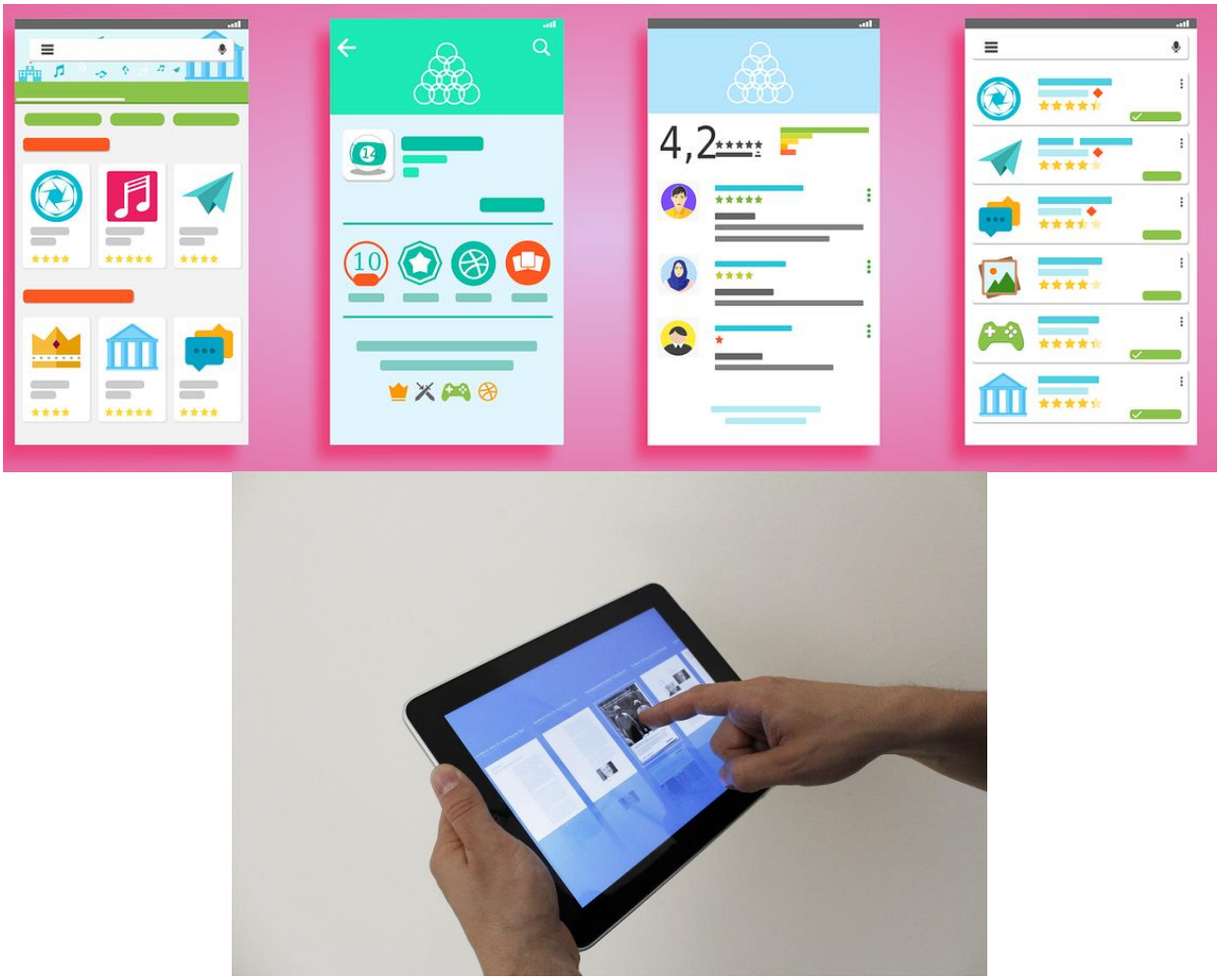
- ➢ Explain working of themes in Flutter
- ➢ Outline the Styling of most used Widgets: `Text`, `Container`, and `RaisedButton`
- ➢ Illustrate Toggling between themes
- ➢ Explain responsive Flutter applications
- ➢ Describe `MediaQuery.of()`
- ➢ Explain `LayoutBuilder` and other widgets used for responsiveness

Session Goals:

At the end of this session, students will learn to style text, buttons, and other components they use in the Flutter application. They will further learn to use multiple themes and toggle between them. Additionally, they will be able to create responsive applications, which can adapt according to orientation and the type of device such as Mobile, Tablet, or Browser.

9.1 Styling in Flutter

Styling is an essential requirement for any application. A good-looking UI can attract a lot of users.



Flutter offers many styling methods and the option to style the whole theme of the application.

9.1.1 Create a Theme for App in Flutter

Creating themes for apps in Flutter is very simple. The `MaterialApp` in `main.dart` is the one wrapping the whole application and it has an argument called `theme` where developers can specify the theme for the whole app. Making any change here will reflect throughout the application thus, preventing boilerplate code. Code Snippet 1 shows the theme argument for auto generated `main.dart` of a sample application.

Code Snippet 1:

```
class MyApp extends StatelessWidget {
 const MyApp({Key? key}) : super(key: key);
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     debugShowCheckedModeBanner: false,
     title: 'Styling Demo',
     //theme
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: StylingExample(),
   );
 }
}
```

Code Snippet 2 shows an example of a simple UI. It is the `StylingExample()` class referred to in the `main.dart`.

Here, text, icons, buttons, sliders, and check boxes are utilized without any styling properties. By default, the primary color will be blue. This means that all the widgets that are created without any styling will be in blue color.

Code Snippet 2:

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

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

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('App Bar'),
     ),
     body: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: [
         Text(
           'Heading 4',
           style: Theme.of(context).textTheme.headline4,
         ),
```

```
          Slider(value: 0, onChanged: (s) {}),
          Icon(Icons.collections_sharp),
          Checkbox(value: true, onChanged: (s) {}),
          TextButton(onPressed: () {}, child: Text('Text
Button')),
          ElevatedButton(onPressed: () {}, child:
Text('Elevated Button')),
        ],
      ),
    );
  }
}
```

Figure 9.1 shows the output of the `StylingExample` widget.

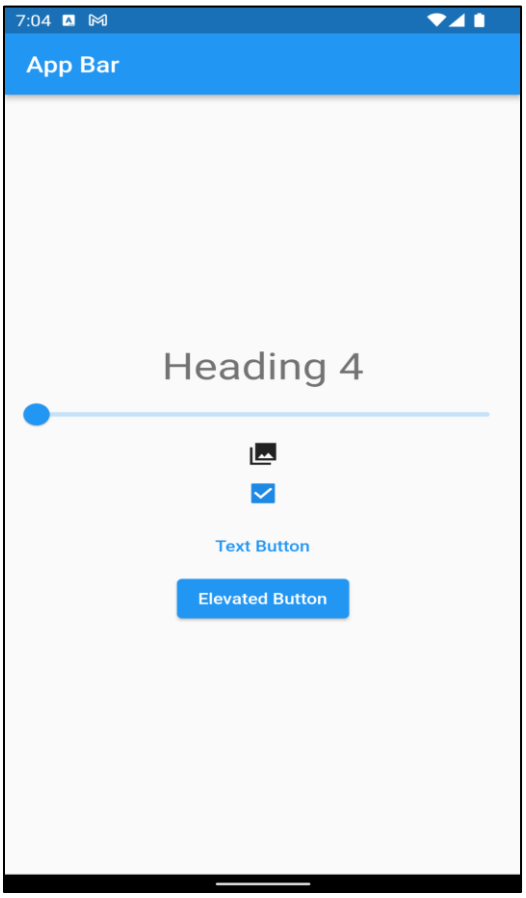


Figure 9.1: Output of `StylingExample` Widget

9.1.2 Style a Specific Widget in Flutter

a. Text:

`Text` can be styled using the `TextStyle` method which can modify `color`, `backgroundColor`, `fontSize`, `fontWeight`, `lineHeight`, space between words/letters, underlines, style, and so on.
Syntax for the `TextStyle` is given as follows:

```
TextStyle? style,
```

Code Snippet 3 shows the usage of the `TextStyle` method.

Code Snippet 3:

```
Text(
            'New Text',
            style: TextStyle(
              //size
              fontSize: 30,
              //font weight
              fontWeight: FontWeight.w600,
              //color
              color: Colors.teal,
              //line height
              height: 1.5,
              //word spacing
              wordSpacing: 2,
              //letter spacing
              letterSpacing: 2,
              //background color of text
              backgroundColor: Colors.yellow,
            ),
          ),
```

Figure 9.2 depicts the output of styled text.

Figure 9.2: Output of Styled Text

b. Container:

The `Container` has a decoration property that can be used to specify `border`, `borderRadius`, `shadow`, `gradients`, images, and more.

Syntax for decoration is given as follows:

```
Decoration? decoration,
```

A simple Container with `borderRadius`, `boxShadow`, and `border` is created within the Code Snippet 4.

Code Snippet 4:

```
Container(
            height: 100,
            width: 100,
            decoration: BoxDecoration(
              //color
              color: Colors.orange,
              //border radius
              borderRadius: BorderRadius.circular(10),
              //border with width 5
              border: Border.all(color:
               Colors.orange.shade900, width: 5),
              //shadow for container
              boxShadow: [
                BoxShadow(
                  // offset where shadow should fall
                  offset: Offset(5, 10),
                  //shadow color
                  color: Colors.grey,
                  //blur radius
                  blurRadius: 5,
                  //spread radius
                  spreadRadius: 1,
                ),
              ],
            ),
          ),
```

Figure 9.3 shows the output of the execution of Code Snippet 4.

Figure 9.3: Output of Styled Container

Note: If decoration property is used, then color has to be specified inside `BoxDecoration`, else it will throw an error.

c. RaisedButton:

The `RaisedButton` is deprecated in Flutter and it is advised not to use it anymore. `ElevatedButton` can be used instead.

Following are the properties of `ElevatedButton`:

```
ButtonStyle styleFrom({
 Color? primary,
 Color? onPrimary,
 Color? onSurface,
 Color? shadowColor,
 Color? surfaceTintColor,
 double? elevation,
 TextStyle? textStyle,
 EdgeInsetsGeometry? padding,
 Size? minimumSize,
 Size? fixedSize,
 Size? maximumSize,
 BorderSide? side,
 OutlinedBorder? shape,
 MouseCursor? enabledMouseCursor,
 MouseCursor? disabledMouseCursor,
 VisualDensity? visualDensity,
 MaterialTapTargetSize? tapTargetSize,
 Duration? animationDuration,
 bool? enableFeedback,
 AlignmentGeometry? alignment,
 InteractiveInkFeatureFactory? splashFactory,
})
```

Code Snippet 5 and Figure 9.4 show an example and result for `ElevatedButton` respectively.

Code Snippet 5:

```
ElevatedButton(
            onPressed: () {},
            child: Text('Button'),
            style: ElevatedButton.styleFrom(
                //button color
                primary: Colors.amber.shade700,
                //shape and border radius
                shape: RoundedRectangleBorder(
                  borderRadius: BorderRadius.circular(20),
                ),
                //to specify fixed size for a button
                fixedSize: Size(100, 100)),
          ),
```

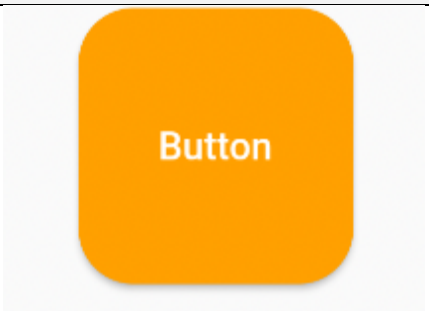


Figure 9.4: Output of Styled ElevatedButton

9.2 Toggling Between Themes In Flutter

Many applications today have the option to switch between light and dark themes. Some apps even have the option to set custom themes. Users should feel comfortable using the app as they wish to see. Themes in Flutter offer a way by which they can set a custom color, font, size, and border for several widgets that will be reflected all over the application.

Custom themes can also be created as per requirements.

9.2.1 Creating a Dark Theme in Flutter

Step 1: Create a new class file called `AppTheme` where themes can be specified. Create a new static variable called `darkTheme`. Refer to Code Snippet 6.

Code Snippet 6:

```
class AppTheme  {
 static ThemeData darkTheme = ThemeData();
}
```

Step 2: Now, call the theme inside `MaterialApp`. Code Snippet 7 shows the theme being called inside the `MaterialApp` in the `main.dart` file.

Code Snippet 7:

```
MaterialApp(
     //..
     darkTheme: AppTheme.darkTheme,
     //..
     home: StylingExample(),
   );
```

Code Snippet 8 shows the dark theme styling for the Flutter application.

Code Snippet 8:

```
static ThemeData darkTheme = ThemeData(
     //scaffold bg
     scaffoldBackgroundColor: Colors.black54,
     //app bar theme
     appBarTheme: AppBarTheme(color: Colors.purple),
     //elevated button theme
     elevatedButtonTheme: ElevatedButtonThemeData(
       style: ElevatedButton.styleFrom(
         primary: Colors.red,
       ),
     ),
     //icon
     iconTheme: IconThemeData(color: Colors.white),
     //text theme - heading 4
     textTheme: TextTheme(
       headline4:
           TextStyle(color: Colors.white70, fontWeight:
FontWeight.w500),
     ));
```

9.2.2 Creating a Light Theme in Flutter

Step 1: Follow the same steps that were followed when creating the dark theme to create the light theme in the same class. Code Snippet 9 shows how to create Light Theme.

Code Snippet 9:

```
static ThemeData lightTheme = ThemeData(
     //scaffold bg
     scaffoldBackgroundColor: Colors.white,
     //app bar theme
     appBarTheme: AppBarTheme(color: Colors.indigo),
     //elevated button theme
     elevatedButtonTheme: ElevatedButtonThemeData(
       style: ElevatedButton.styleFrom(
         primary: Colors.red,
       ),
     ),
     //text theme - heading 4
     textTheme: TextTheme(
       headline4:
           TextStyle(color: Colors.blueGrey, fontWeight:
FontWeight.w500),
     ));
```

Step 2: Inside the `MaterialApp`, apply the light theme as shown here.

```
theme: AppTheme.lightTheme,
```

9.2.3 Toggling Between Dark and Light Theme in Flutter

Step 1: To toggle between themes, create a button that on click will toggle the theme. There are two themes defined in the `MaterialApp`. Add another property called `themeMode` and change the `AppTheme` class by extending a `ChangeNotifier`. Refer to Code Snippet 10. It is a new file called `apptheme.dart`

Step 2: Create a boolean variable to toggle dark and light themes followed by a function to change themes and notify the listeners.

Code Snippet 10:

```
class AppTheme extends ChangeNotifier {
 bool _isDarkTheme = true;
 ThemeMode get currentTheme => _isDarkTheme ? ThemeMode.dark
: ThemeMode.light;

 void toggleTheme() {
   _isDarkTheme = !_isDarkTheme;
   notifyListeners();
 }
 //..
}
```

Step 3: Create a global instance for the `AppTheme` in the `main.dart` file as shown.

```
final AppTheme appTheme = AppTheme();
```

Step 4: Now, define these themes in `MaterialApp` in `main.dart`. Change the `MyApp` to a `StatefulWidget`. In the `initState` add a listener to `appTheme`, so that theme change will be updated immediately. Refer to Code Snippet 11.

Code Snippet 11:

```
class MyApp extends StatefulWidget {
 const MyApp({Key? key}) : super(key: key);
 @override
 State<MyApp> createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
 @override
 void initState() {
   //add a listener
   appTheme.addListener(() {
     setState(() {});
   });
   super.initState();
 }

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

```
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Styling Demo',
      //theme
      theme: AppTheme.lightTheme,
      darkTheme: AppTheme.darkTheme,
      //theme mode
      themeMode: appTheme.currentTheme,
      home: StylingExample(),
    );
  }
}
```

Step 5: Finally, create a button to toggle the theme. In the `appBar` of `StylingExample` add an `IconButton` and provide the `toggleTheme` function. Refer to Code Snippet 12.

Code Snippet 12:

```
actions: [
          IconButton(
              onPressed: () {
                appTheme.toggleTheme();
              },
              icon: Icon(Icons.brightness_4)),
        ],
```

Figures 9.5 A and B show the toggling between the light and the dark themes respectively when the button is clicked.

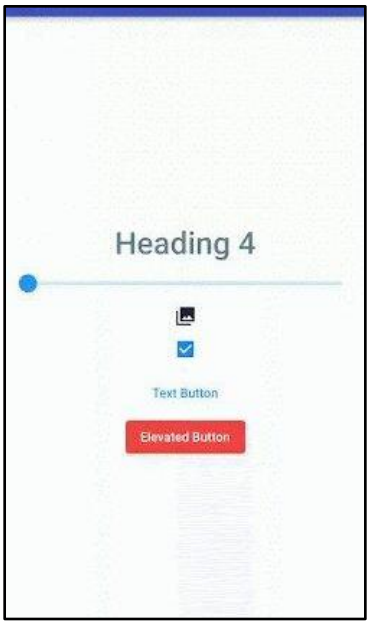


Figure 9.5A: Output of ToggleTheme - Light Theme

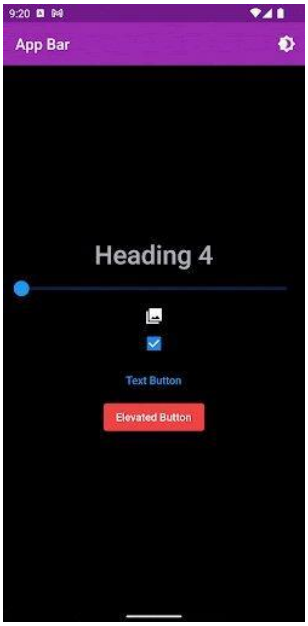


Figure 9.5B: Output of ToggleTheme - Dark Theme

9.3 Responsive Flutter Application

Flutter is a cross-platform framework that can be used to build applications varying from smartwatches to large TVs. In these cases, the same design and layout cannot be used. The layout and designs have to be changed to adapt to screen sizes. Flutter offers some widgets through which responsiveness can be achieved.

Following are the steps to achieve responsiveness in a Flutter application:

Step 1: For the Flutter app to support all orientations, add the code given in Code Snippet 13 inside the build of `MyApp()` from `main.dart`.

Code Snippet 13:

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

SystemChrome.setPreferredOrientations([
     DeviceOrientation.landscapeLeft,
     DeviceOrientation.landscapeRight,
     DeviceOrientation.portraitDown,
     DeviceOrientation.portraitUp
   ]);
```

Step 2: Using `setPreferredOrientations`, specify the list of orientations the device should support. For example, if the app has to work in landscape orientation alone, specify only these two:

```
DeviceOrientation.landscapeLeft
DeviceOrientation.landscapeRight
```

Similarly, use the portrait options if the app has to support portrait.

9.3.1 LayoutBuilder Class in Flutter

Using `LayoutBuilder`, it is possible to determine the `maxHeight` and `maxWidth` of the Widget.

In Code Snippet 14, two `LayoutBuilders` are defined inside a Row. `Expanded` widget can be used to allocate maximum space. Inside these layout builders, a `Text` widget is implemented to show the maximum width the widget takes.

Figures 9.6 A and B show the usage of `LayoutBuilder`.

Code Snippet 14:

```
Scaffold(
     appBar: AppBar(),
```

```
    body: Row(
      children: [
        //widget 1
        Expanded(
          child: LayoutBuilder(
              builder: (context, constraints) => Container(
                    color: Colors.yellow.shade200,
                    child: Center(
                        child: Text('widget width\n' +
                  constraints.maxWidth.toStringAsFixed(2))),
                  )),
        ),
        //widget 2
        Expanded(
          flex: 2,
          child: LayoutBuilder(
              builder: (context, constraints) => Container(
                    color: Colors.teal,
                    child: Center(
                        child: Text('widget width\n' +
                  constraints.maxWidth.toStringAsFixed(2))),
                  )),
        ),
      ],
    ),
  )
```

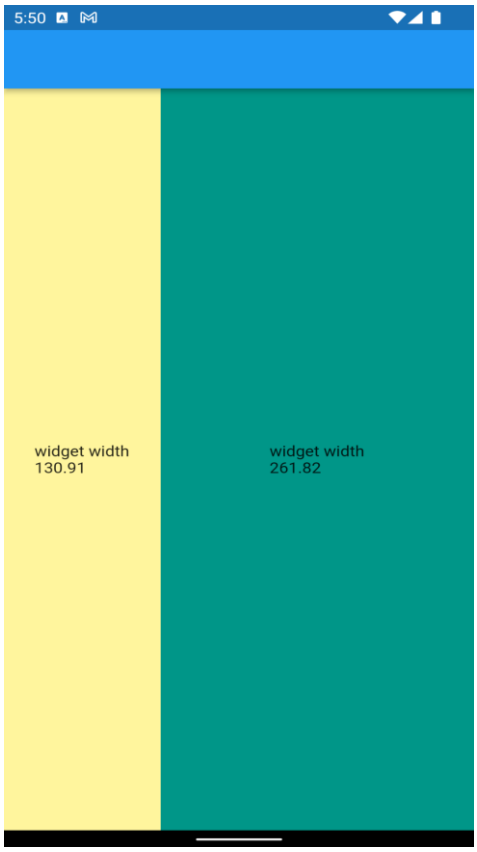


Figure 9.6A: Usage of LayoutBuilder - Portrait View

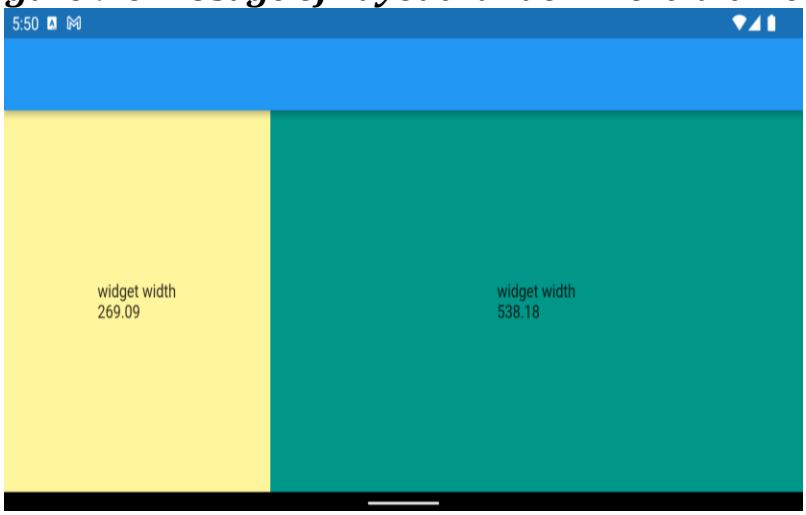


Figure 9.6B: Usage of LayoutBuilder - Landscape View

9.3.2 `MediaQuery.of()` Method in Build Function

`MediaQuery` in Flutter can be used to retrieve the size (width and height) and orientation (portrait/landscape) of the device.

Note: `MediaQuery` can be used in the place where the total size of the screen is required. Whereas, `LayoutBuilder` can be used in the place where height/width of the particular widget is required.

In Code Snippets 15 and 16, it can be seen that a container is used in a `Scaffold` and wrapped inside a `Center` widget. The height and width of the `Container` is defined using `MediaQuery` which will return a double value. 0.25 specifies the container height as 25% of the total height. Output in Figures 9.7 A and B show the difference when `MediaQuery` is used to specify dynamic sizes for orientations.

Code Snippet 15:

```
//To get orientation of device
   Orientation orientation =
MediaQuery.of(context).orientation;
   //To get height of the device
   double deviceHeight = MediaQuery.of(context).size.height;
   //To get width of the device
   double deviceWidth = MediaQuery.of(context).size.width;
```

Code Snippet 16:

```
Container(
         height: deviceHeight * 0.25,
         width: deviceWidth * 0.5,
         color: Colors.red,
       ),
```

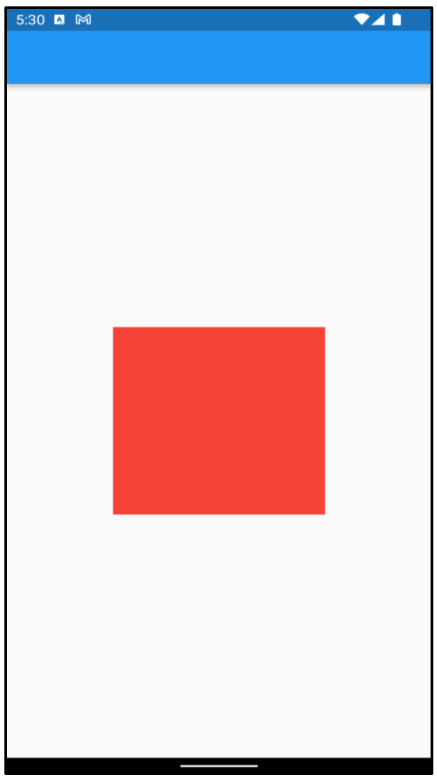


Figure 9.7A: Output of Setting Dynamic Size Using `MediaQuery.of()`- Portrait View

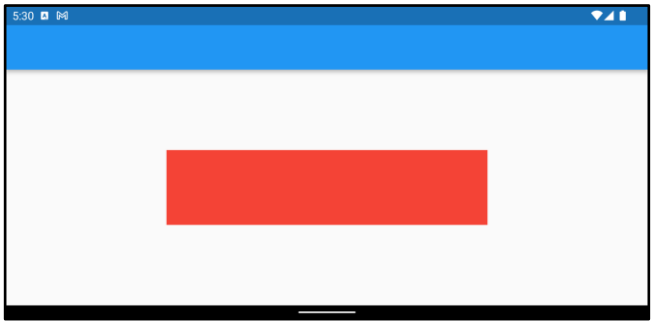


Figure 9.7B: Output of Setting Dynamic Size Using `MediaQuery.of()`- Landscape View

Note: `MediaQuery.of(context)` can be accessed only from inside the build method.

9.4 Useful Widgets to Build a Responsive UI in Flutter

Flutter offers some other widgets other than `LayoutBuilder` by which the application can be made responsive.

9.4.1 Aspect Ratio

Aspect ratio can be used to assign a specific size to a child widget. This widget first tries the largest width permitted by the layout constraints and then, decides the height by applying a given aspect ratio to the width.

The `aspectRatio` should be specified in a double value. Refer to Code Snippet 17.

Code Snippet 17:

```
AspectRatio(
         aspectRatio: 16 / 9,
         child: Container(
           color: Colors.amber,
         ),
       )
```

9.4.2 CustomSingleChildLayout

`SingleChildLayout` widgets are those which allow only one widget as their child such as `Container`, `Align`, `Center`, `SizedBox`, `Positioned`, and so on. A custom single-child widget can be created using the `CustomSingleChildLayout` widget. It is important to note that the developers have to specify the delegate and the child widget. The delegate has two override methods `performLayout` and `shouldRelayout` which can also be customized as required.

Code Snippet 18 shows how to override the default `performLayout` and `shouldLayout` to customize the `SingleChildLayout`.

Code Snippet 18:

```
class MySingleChildDelegate extends SingleChildLayoutDelegate
{
 @override
 void performLayout(Size size) {
   // TODO: implement performLayout
 }
  @override
 bool shouldRelayout(covariant SingleChildLayoutDelegate
          oldDelegate) {
```

```
    // TODO: implement shouldRelayout
    throw UnimplementedError();
  }
}
```

9.4.3 CustomMultiChildLayout

`MultiChildLayout` widgets allow multiple widgets as their children. `Row`, `Column`, `Stack`, `ListView`, and many more can be highlighted as examples of this. Using `CustomMultiChildLayourDelegate` it is possible to customize a MultiChild Widget.

Following is the syntax for `CustomMultiChildLayout`:

Syntax

```
CustomMultiChildLayout CustomMultiChildLayout({
 Key? key,
 required MultiChildLayoutDelegate delegate,
 List<Widget> children = const <Widget>[],
})
```

It is important to note that the children in this widget should be inside a layoutId widget with each child having a unique id.

9.4.4 FittedBox

`FittedBox` restricts its child from growing beyond a certain limit. It rescales them according to the size available. Code Snippet 19 shows how this works.

Code Snippet 19:

```
FittedBox(
          child: Container(
            color: Colors.amber,
            child: Text('Fitted Box Container'),
          ),
        ),
```

9.4.5 MediaQueryData

`MediaQuery.of(context)` itself is a `MediaQueryData` which will return the `padding`, `ViewInsets`, and `ViewPadding` of the current screen. This information will help us to calculate the padding for orientations, screen sizes, and keyboard pop-ups. Figure 9.8 differentiates `padding`, `viewInsets`, and `viewPadding`.

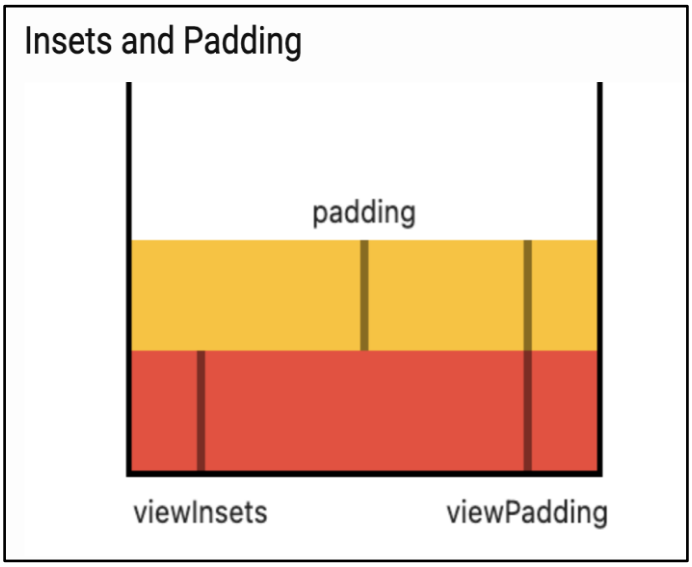


Figure 9.8: Differentiation Between Padding, ViewInsets, and ViewPadding

9.4.6 OrientationBuilder

`OrientationBuilder` can be used to layout widgets as per device orientation (portrait or landscape). Code Snippet 20 demonstrates the use of the `orientationBuilder`. `GridView` has been used where the `crossAxisCount` and `childAspectRatio` are adapted as per orientation. There will be two widgets when in portrait and three widgets when in landscape orientation as in Figures 9.9 A and B.

Code Snippet 20:

```
Scaffold(
   body: OrientationBuilder(
     builder: (context, orientation) => GridView.builder(
       //gridDelegate is where we specify the spacing,
        // count and aspect ratio..
       gridDelegate:
```

```
          SliverGridDelegateWithFixedCrossAxisCount(
              //change count according to orientation
              crossAxisCount: orientation ==
          Orientation.portrait ? 2 : 3,
              //change aspect ratio as per orientation
              childAspectRatio:
                  orientation == Orientation.portrait ? 1 : 16
                   / 9,
              mainAxisSpacing: 10,
              crossAxisSpacing: 10),
          itemBuilder: (context, index) => Container(
            color: Colors.blue,
          ),
        ),
      ),
    );
```

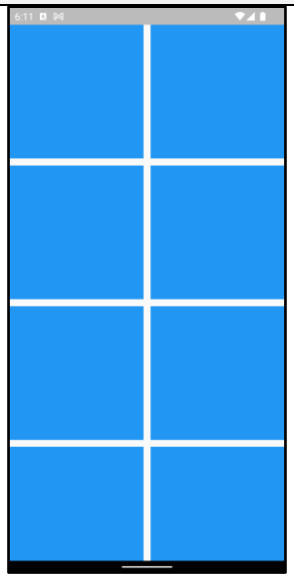


Figure 9.9A: OrientationBuilder Output - Portrait View

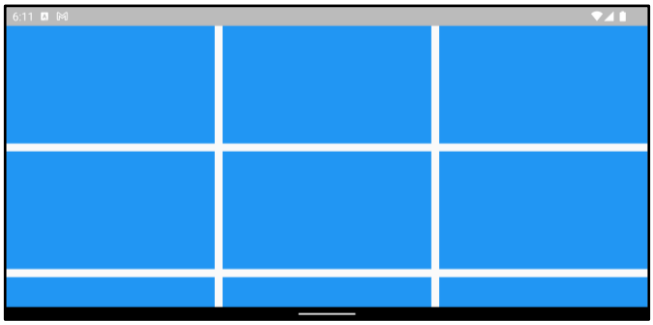


Figure 9.9B: OrientationBuilder Output - Landscape View

9.5 Summary

- `Text`, `Buttons`, and many more widgets in Flutter can be styled.
- Custom Themes can be created for the Flutter application.
- Two or more themes can be created and can be toggled between
- Changes made for a widget in `themeData` will be applied throughout the application.
- Responsiveness is a feature by which the application layout will be modified according to the screen size and orientation.
- Flutter has some widgets which help in creating responsive layouts: `LayoutBuilder`, `OrientationBuilder`, and so on.
- `MediaQueryData` can be used to fetch the size, height, width, orientation, padding, and much more information about the current screen.
- Custom Single and Multi Child Layout Widgets can be created in Flutter applications.
- `AspectRatio`, `FittedBox`, `Expanded`, and `Flexible` are some other widgets useful in creating responsive apps.

9.6 Test Your Knowledge

1) To specify a shadow for a container, which property from `BoxDecoration` can be used?
   a. `boxShadow`
   b. `border`
   c. `shape`
   d. `borderRadius`

2) Which one of the following data cannot be obtained from `MediaQuery.of(context)`?
   a. Total size of the device (height and width)
   b. Orientation
   c. Padding
   d. Size of a particular widget from a screen

3) Why is `ChangeNotifier` used?
   a. Avoids unnecessary rebuilds
   b. Provides change notification to its listeners
   c. When combined with proper state management, performance can be improved
   d. All of these

4) For a Flutter Application to support only portrait orientation, which command can be used?
   a. `SystemChrome.setPreferredOrientations([DeviceOrientation.landscapeRight, DeviceOrientation.landscapeLeft]);`
   b. `SystemChrome.setPreferredOrientations([DeviceOrientation.portraitUp, DeviceOrientation.landscapeRight]);`
   c. `SystemChrome.setPreferredOrientations([DeviceOrientation.portraitUp, DeviceOrientation.portraitDown]);`
   d. `SystemChrome.setPreferredOrientations([]);`

5) Which one of the following is not true?
   a. `AspectRatio` decides the height by applying a given aspect ratio to the width
   b. `FittedBox` rescales the size according to maximum space available
   c. Custom single and multi-child Widgets can be created in Flutter
   d. `OrientationBuilder` is used to build different layouts for different orientations

Answers - Test Your Knowledge

1. `boxShadow`
2. Size of a particular widget from a screen
3. All of these
4. `SystemChrome.setPreferredOrientations([DeviceOrientation.portraitUp, DeviceOrientation.portraitDown]);`
5. `FittedBox` rescales the size according to maximum space available

9.7 Try It Yourself

1) Design a login screen. The screen must have a linear gradient background with any three colors, a large welcome text, two text fields for email and password and a Login Button at the bottom. The password text fields must be obscured.

2) Create a GridView that supports orientation. The two tiles should be in portrait mode and four in landscape mode. In addition to this, add a `FloatingActionButton` with toggle theme function that switches between dark and light mode.

   Hint: Refer to the image given in following link:

   https://blog.logicwind.com/content/images/size/w2000/2018/08/a112.png

Session 10
FutureBuilder and StreamBuilder in Flutter

Learning Objectives

In this session, students will learn to:

- ➢ Explain `Future` and `FutureBuilder`
- ➢ Explain `Stream` and `StreamBuilder`
- ➢ List the differences between `FutureBuilder` and `StreamBuilder`

***Session Goals**:*
The session introduces concepts of `FutureBuilder` and `StreamBuilder` in Flutter. These are essential widgets as they are used in most apps which include fetching data from external servers or databases. By the end of this session, students will be ready to fetch and display future and stream data in their Flutter applications.

10.1 `FutureBuilder` in Flutter

There is a concept called asynchronous programming where the function cannot generate results immediately. It requires some time to fetch or process and return the data. A normal function would process the input and generate an output of any type. Whereas, these asynchronous functions will process and generate an output of type `Future`, which essentially contains the output of any type.

Here are some examples to help you understand better.

A function to perform addition between two numbers will execute in no time. It will operate and return the result immediately. This is a synchronous function.

On the other hand, consider opening a social media application such as Instagram. Once the app opens it will be loading data, one can see a loading indicator denoting that Newsfeed posts are being fetched. Once the data is fetched, the home page will be filled with posts and stories. This is one example of an asynchronous function.

Some other examples are Google search, fetching data from a database, uploading files from a device, and so on.

10.1.1 `FutureBuilder` Widget in Flutter

Flutter has a widget called `FutureBuilder` which helps in displaying `Future` data with minimal effort.

Syntax of `FutureBuilder` class:

```
FutureBuilder FutureBuilder({
 Key? key,
 Future<T>? future,
 T? initialData,
 required Widget Function(BuildContext, AsyncSnapshot<T?>)
builder,
})
```

Important arguments to be considered from the code are:

1. **`Future<T>? future`**: This is where one specifies the `Future` function that returns a value of a type *T* (`int`, `String`, anything).

2. **`builder`**: This is where one builds the data to be displayed. Two arguments `Context` and `snapshot` are passed here. `Snapshot` is a type available in Flutter through which one can process the data.

An asynchronous function may not always return data. There are possibilities that it may return an error, take more time to load, and so on. This is where a snapshot helps us.

10.1.2 States in FutureBuilder

As seen earlier, an asynchronous function requires time to return data, so it is required to indicate to a user whether it is loading, fetched, or has some error. In Flutter, there are states to denote these and are called `ConnectionStates`. Different `ConnectionStates` in Flutter are as follows:

`ConnectionState.waiting`	Connected to asynchronous computation and awaiting interaction or in other words. Data is being fetched and the process has not been completed yet.
`ConnectionState.done`	Connection with asynchronous computation has been terminated or in other words either data fetched or returned an error, but the connection is closed.
`ConnectionState.active`	Connected to an active asynchronous computation. Connection is established. This is ideal for streams.
`ConnectionState.none`	Not currently connected to any asynchronous computation that is no Future or stream function is available for connection.

10.1.3 Create an Application to Demonstrate FutureBuilder in Flutter

To understand all the concepts better, create a Flutter application with a `Future` function that will return the current time after a delay of two seconds. Code Snippet 1 shows the starter code for how the `FutureBuilder` can be used for this scenario.

Code Snippet 1:

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

class FutureBuilderExample extends StatefulWidget {
 const FutureBuilderExample({Key? key}) : super(key: key);

 @override
 State<FutureBuilderExample> createState() =>
_FutureBuilderExampleState();
}

class _FutureBuilderExampleState extends
State<FutureBuilderExample> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: const Text('Future Builder Example'),
     ),
     body: Container(),
   );
 }
}
```

Code Snippet 2 shows how to create the `Future` function.

Code Snippet 2:

```
Future<String> fetchTime() async {
 await Future.delayed(const Duration(seconds: 2));
 var date = DateTime.now();
 return
'${date.hour}:${(date.minute).toString().padLeft(2,'0')}:${
(date.second).toString().padLeft(2,'0')}';
}
```

- The `fetchTime()` function used in the Code Snippet 2 will return the current time in an asynchronous way.
- Since this is an asynchronous function, the keywords `async` and `Future<String>` are used where `String` denotes the return type.
- Since the execution is to be delayed for two seconds, `Future.delayed` method is used.

- The `await` keyword is used before `Future.delayed` to notify the function to wait till the execution is completed. Hence, the function will proceed only after the execution is completed.
- Next, `DateTime.now()` is used to get the current date and time and is assigned to `date` variable.
- Finally, the `date` value is formatted in a presentable manner so the user can understand. One can format it in whatever way they prefer.

Next comes the `FutureBuilder`:

Code Snippet 3 shows how to define `FutureBuilder` within the `Center` widget.

Code Snippet 3:

```
body: Center(
       child: FutureBuilder(
        future: fetchTime(),
        builder: (context, snapshot) {},
      ),),
```

In the code, `fetchTime()` returns a String and specifies that in the widget as `FutureBuilder<String>`.

Consider multiple scenarios while handling asynchronous data. Code Snippet 4 demonstrates the scenario of handling asynchronous data and `ConnectionStates` inside the `FutureBuilder` widget.

Code Snippet 4:

```
body: Center(
        child:FutureBuilder<String>(
       future: fetchTime(),
       builder: (context, snapshot) {
         if (snapshot.connectionState ==
ConnectionState.waiting) {
           return CircularProgressIndicator();
         }
         if (snapshot.hasError) {
           return Column(
             mainAxisAlignment: MainAxisAlignment.center,
             children: [
               const Text(
```

```
                'Error',
                style: TextStyle(fontWeight:
                       FontWeight.w500, fontSize: 16),
              ),
              Text(
                snapshot.error.toString(),
                style: const TextStyle(
                    color: Colors.red,
                    fontWeight: FontWeight.w600,
                    fontSize: 20),
              ),
            ],
          );
        }
        if (snapshot.hasData) {
          return Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              const Text(
                'Time Now',
                style: TextStyle(fontWeight:
                       FontWeight.w500, fontSize: 16),
              ),
              Text(
                snapshot.data.toString(),
                style: const TextStyle(
                    color: Colors.black,
                    fontWeight: FontWeight.w600,
                    fontSize: 20),
              ),
            ],
          );
        }
        return Text('state: ${snapshot.connectionState}');
      },
    ),),
```

- In Code Snippet 4, inside the `builder` method of the `FutureBuilder` widget, the `connectionState` is first verified using *`snapshot.connectionState == ConnectionState.waiting`* to check if the `fetchTime()` method has returned the result or is still loading. If it is still loading, then a `CircularLoadingIndicator()` widget can be returned to notify the user that data is loading.

- Next, check if the data returned has any errors using *`snapshot.hasError`*. If it has an error, then display the error using the `Text` widget.
- If the data has been loaded and there are no errors, then the data is present which is verified using `snapshot.hasData`. The data which is a `String` in this scenario can be obtained using `snapshot.data` and is displayed inside a `Text` widget.
- By default, FutureBuilder has to return something. So, use some widgets to fill that space even though all possible conditions have been handled Here, a `Text` widget is used which returns the `connectionState`.

Figure 10.1 depicts the output.

Figure 10.1: Output of the application

10.2 `StreamBuilder` in Flutter

Streams are similar to `Future`. Streams are also asynchronous data, but in the form of a sequence of events. A `Future` will terminate once it produces the result. Whereas, a stream is an established connection and data will be received as long as the stream is active.
A good example would be a chat application where both sender and receiver will receive data as long as the participants are active. Some other examples would be video streaming, playing music, and live location.

10.2.1 `StreamBuilder` Widget in Flutter

`StreamBuilder` in Flutter helps to fetch and display streams of data. Similar to `FutureBuilder`, instead of future arguments, there is a `stream` argument.

Syntax for `StreamBuilder` is given as follows:

Syntax

```
StreamBuilder<Object?> StreamBuilder({
 Key? key,
 Object? initialData,
 Stream<Object?>? stream,
 required Widget Function(BuildContext,
AsyncSnapshot<Object?>) builder,
})
```

Important arguments to be considered from the Syntax are:

`builder`

This is where the data to be displayed is built. Two arguments have to be passed here. One can make use of the `connectionStates` seen in `FutureBuilder`. `ConnectionState`.active: Denotes that the stream is active and is being listened to. `ConnectionState`.done: Denotes that the listening to stream function has ended.

`Stream<Object>? stream`

This is where the stream function that returns a sequence of values of a type `T (int, String, anything)` is specified.

10.2.2 Generator Function

A Generator function allows the user to produce a value of the sequence. In simple words, a user can produce a collection of objects which can be accessed sequentially. The two main keywords used in generator function are `async*` and `yield`.

`async*`: It will always return a stream and offer some syntax sugar to emit a value through the yield keyword.

`yield`: It adds a value to the output stream of the surrounding `async*` function. It is like a return function but does not terminate the function.
To understand with an example, consider Code Snippet 5 where the `fetchTimer()` function is a Timer that will count from 10 to 0 and end.

Code Snippet 5:

```
int i = 10;
Stream<String> fetchTimer() async* {
 while (i > 0) {
   await Future.delayed(Duration(seconds: 1));
   i--;
   yield i.toString();
 }
}
```

- Code Snippet 5 has a `fetchTimer()` function which will count from 10 to 0 and end.
- An integer variable is initialized with value 10 outside the `fetchTimer()` function.
- The `fetchTimer()` is a Stream function that returns a sequence of asynchronous values. Hence, the keywords `async*` and `Stream<String>` are used, where `String` denotes the return type.
- Inside the function, a `while` loop is used which checks whether the variable of `i` is greater than zero. It continues to loop until the condition becomes false.
- Inside the loop, a two-second delay is created using `Future.delayed` method and `async` keyword.
- Then, the value of `i` is decremented by one.
- Finally, the value of *`i`* is returned as *`String`* using the *`yield`* keyword.

10.2.3 Create an Application to Demonstrate StreamBuilder in Flutter

Use the generic function as an example here.

In this, a 10-second timer app is built, which begins after a second and ends after 10 counts reducing to 0. After that, show a Time Up message.
Code Snippet 6 shows how to create a new widget called `StreamBuilderExample`. Start by creating the `StreamBuilder` and wrapping it inside the `Center` widget.

Code Snippet 6:

```
body: Center(
       child: StreamBuilder<String>(
         stream: fetchTimer(),
         builder: (context, snapshot) {},
       ),
     ),
```

The `fetchTimer()` function is declared as the stream. Then, start building the UI.

Code Snippet 7:

```
builder: (context, snapshot) {
           if (snapshot.connectionState ==
                    ConnectionState.waiting) {
             return CircularProgressIndicator();
           }
           if (snapshot.hasError) {
             return Column(
               mainAxisAlignment: MainAxisAlignment.center,
               children: [
                 const Text(
                   'Error',
                   style: TextStyle(fontWeight:
                     FontWeight.w500, fontSize: 16),
                 ),
                 Text(
                   snapshot.error.toString(),
                   style: const TextStyle(
                       color: Colors.red,
                       fontWeight: FontWeight.w600,
                       fontSize: 20),
                 ),
               ],
             );
           }
           if (snapshot.connectionState ==
                    ConnectionState.active) {
             return Column(
               mainAxisAlignment: MainAxisAlignment.center,
               children: [
                 const Text(
                   'Time Ends in:',
                   style: TextStyle(fontWeight:
                     FontWeight.w500, fontSize: 16),
```

```
                ),
                Text(
                  snapshot.data.toString(),
                  style: const TextStyle(
                      color: Colors.black,
                      fontWeight: FontWeight.w600,
                      fontSize: 20),
                ),
              ],
            );
          }
          if (snapshot.connectionState ==
                   ConnectionState.done) {
            return Text(
              'Time Up!\n${snapshot.connectionState}',
              textAlign: TextAlign.center,
              style: const TextStyle(
                  color: Colors.black,
                  fontWeight: FontWeight.w500,
                  fontSize: 20),
            );
          }
          return Container();
        }
```

While executing the code, first, check whether the data is loading. Use `ConnectionState.waiting` to check loading and use `CircularLoadingIndicator()` to notify the user.

Next, check for errors using `snapshot.hasError` method inside an `if` condition. If it has any errors, show the `snapshot.error` in the `Text` widget. Then, make use of the other connection states and display data as per the state. `ConnectionState.active` means that the connection has been established and data is incoming. Display the timer in a `Text` widget.

`ConnectionState.done` means that the connection is closed and data is no longer received. Show a Time up message. Refer to Figures 10.2 A and B which show the output of the Time application created using `StreamBuilder`.

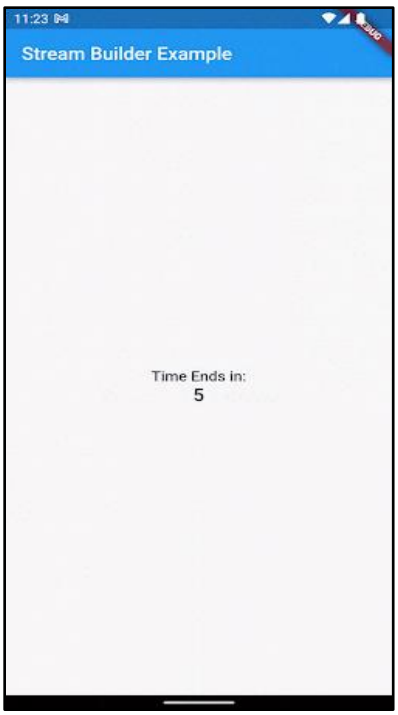


Figure 10.2 A: Output of the Time application

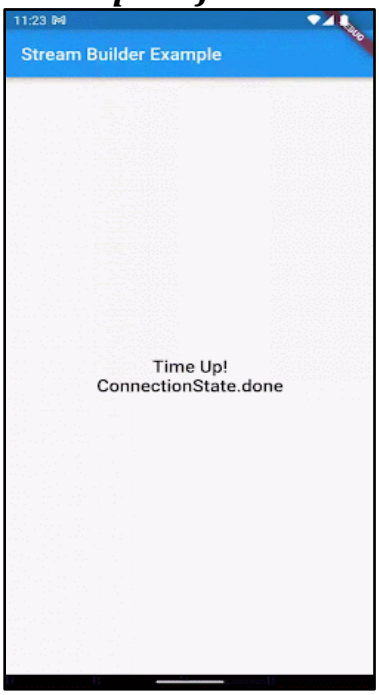


Figure 10.2 B: Output of the Time application - `ConnectionState.done`

10.3 Difference between FutureBuilder and StreamBuilder

Table 10.1 shows a few differences between FutureBuilder and StreamBuilder.

FutureBuilder	StreamBuilder
`FutureBuilder` is used for one-time response	`StreamBuilder` is for fetching data more than once. Like listening to a data change
Once the `FutureBuilder` fetches the result, it is done. Widget will not update if the date changes and the connection will end	`StreamBuilder` will rebuild the UI whenever the data changes till the connection is active
Main keywords: `Future`, `async`	Main keywords: `Stream`, `async*`, `yield`
Use Case examples: HTTP request, uploading a file from the device, taking a picture from the camera, getting device info such as location and battery.	*Use Case examples*: live location, streaming video/music, listening to a WebSocket.

Table 10.1: Difference between FutureBuilder and StreamBuilder

10.4 Summary

- Future represents an asynchronous computation and returns data after a certain time.
- `FutureBuilder` can be used to fetch and display `Future` data.
- Stream is also an asynchronous data but it is a sequence of asynchronous events.
- `StreamBuilder` can be used to fetch and display a stream of data.
- Keep in mind to handle all possible scenarios like what to display during loading time, in case of error, after the connection is closed, and so on.
- Using these widgets, HTTP requests and other asynchronous communications can be made.

10.5 Test Your Knowledge

1) Which one of the following is an example of asynchronous computation?
   a. Printing a statement
   b. Addition of two numbers
   c. Weather data from the API
   d. DataType conversion

2) Which of the following are valid reasons to use `FutureBuilder`?
   a. To fetch data from asynchronous function and display in UI
   b. To fetch data only once
   c. Handle multiple connection states in the UI during data fetching
   d. All of these

3) What is the difference between `return` and `yield`?
   a. Both keywords are used to return a value and end the function.
   b. `yield` is used to return value from iterable or a stream as long as it is active whereas, `return` will end the function.
   c. `return` does not end the function, but the `yield` will.
   d. `return` is used to return an integer value whereas, `yield` is used to return a double value.

4) When to use `StreamBuilder`?
   a. To fetch data from an asynchronous function and keep listening to data changes and update the UI
   b. To Fetch data once and close the connection immediately
   c. Handle exceptions thrown by synchronous functions
   d. To perform all of these

5) Which of the following are examples of a Stream?
   I. Uploading a file to a server
   II. Streaming movies from a streaming platform
   III. Making API calls
   IV. Watching live location of a friend
   a. I and III
   b. II and IV
   c. All of these
   d. None of these

Answers - Test Your Knowledge

1. Weather data from the API
2. All of these
3. The `yield` is used to return value from iterable or a stream as long as it is active whereas return will end the function.
4. To fetch data from an asynchronous function and keep listening to data changes and update the UI.
5. II and IV (Streaming movies from a streaming platform and Watching live location of a friend)

10.6 Try It Yourself

1) Design a Flutter application to display an image from gallery.
    a) Write a function to fetch an image from gallery. Since it will be a `Future` function, make use of the `FutureBuilder` widget.
    b) The UI will be a Column with Image and a Button to upload an Image.
    c) Make sure to show 'No Image Uploaded' text when no image is uploaded and to handle all other states.

2) Create a Flutter application which will display the time remaining for your next birthday. Make use of `StreamBuilder` and design an attractive UI with confetti animations.

Session 11
Database Concepts with Sqflite and Firebase Database

Learning Objectives

In this session, students will learn to:

- ➢ Describe Sqflite in Flutter and its usage
- ➢ Elaborate the process of installing SQLite in Flutter
- ➢ Explain executing SQLite Queries in Flutter
- ➢ Explain working with Firebase Database in Flutter

Session Goals:

Databases play a major role in any application. A database is necessary to store, access, and modify data as required. This session describes how to use SQLite database to store data locally on the device and also, the method to modify the same. The session also explains about Firebase real-time databases to store data in NoSQL format in a remote database. In addition to this, one will also learn to authenticate users, sign up, and login using Firebase Authentication feature.

11.1 Sqflite in Flutter and Its Usage

Consider a simple scenario of a login application built using Flutter. There is a UI with login and password fields and a button upon clicking which user login validation can happen. However, to validate the entered user id and password against stored credentials, one must have a storage mechanism. The credentials would then be stored in the database and retrieved to check against values entered by a user in the application.

This is just one example of how data storage is crucial to most applications. Storing, retrieving, and modifying data are common tasks in many applications. In a cross-platform application, it is important to have a lightweight storage mechanism, rather than a heavy database software.

SQLite is a relational Database management system in which data is stored in the form of tables. Any number of tables can be created and they can be related to each other. SQL Server is also a Relational Database Management System which can also be used in Flutter by adding the `sql_conn` plugin. When it comes to Cross-Platform mobile apps, SQLite is preferred over SQL Server because SQLite is much faster compared to SQL Server and read and write operations can be performed much quicker. SQLite loads only the required part and not the entire file. It overwrites only the parts which are changed. Hence, SQLite is preferred over SQL Server for Flutter.

Flutter has a `sqflite` plugin, through which developers can use this database to store data locally in any Flutter applications.

11.2 Installing SQLite in Flutter

It is required to add a dependency to use SQLite in Flutter. By adding the dependency, all the features of `sqflite` can be integrated with Flutter applications.

11.2.1 Adding Sqflite Dependency

In the *`pubspec.yaml`* file of the application, add the line `sqflite: ^2.0.3+1`, where `sqflite` is the name of the package, followed by the latest version.
To find the latest version of the package and more, check https://pub.dev/packages/sqflite. Also, run `flutter pub get` each time after updating dependencies in the `pubspec.yaml` as shown in Code Snippet 1.

Code Snippet 1:

```
dependencies:
 flutter:
   sdk: flutter
 cupertino_icons: ^1.0.2
 sqflite: ^2.0.3+1
```

11.2.2 Adding Path_Provider Dependency and Reasons to Add It

`path_provider` is a Flutter plugin for finding commonly used locations on a file system so that developers can read/write to a local location on a device.

Add following line to the dependencies in `pubspec.yaml`:

```
path_provider: ^2.0.11
```

11.2.3 Create SQLite Database

Before creating a database in Flutter, it is required to initialize it.

Consider an example. In this example, an SQflite Create, Read, Update, and Delete (CRUD) app will be created where developers can create, edit, delete, and view a list of users. The user instance will contain details such as ID, name, email, and age.

In the new project, add all the dependencies as explained earlier. Then, create a model class file to perform CRUD operations as shown in Code Snippet 2.

Code Snippet 2: `user_model.dart`

```
class UserModel {
 final String id;
 final String name;
 final String email;
 final int age;

 UserModel({
   required this.id,
   required this.name,
   required this.email,
   required this.age,
 });

 factory UserModel.fromJson(Map<String, dynamic> data) =>
UserModel(
       id: data['id'],
       name: data['name'],
       email: data['email'],
       age: data['age'],
     );
```

```
 Map<String, dynamic> toMap() => {
       'id': id,
       'name': name,
       'email': email,
       'age': age,
     };
}
```

The `fromJson()` and `toMap()` functions are used to convert the data model to Map Object and vice versa.

Code Snippet 3 demonstrates how to create a new class called `DatabaseService` where all the database operations will be handled. Here, the `Singleton` pattern is being used to create the service class.

Code Snippet 3: `database_service.dart`

```
import 'dart:developer';
import 'package:path_provider/path_provider.dart';
import 'package:sqflite/sqflite.dart';

class DatabaseService {
 static final DatabaseService _databaseService =
DatabaseService._internal();
 factory DatabaseService() => _databaseService;
 DatabaseService._internal();
}
```

Now, the database must be initialized before creating any tables or performing read/write operations. This is shown in Code Snippet 4. Inside the class, create a variable of type Database. A Getter function has been used to fetch the database as shown in Code Snippet 4.

Code Snippet 4: `database_service.dart`

```
static Database? _database;
 Future<Database> get database async {
   if (_database != null) return _database!;
   _database = await initDatabase();
   return _database!;
 }
```

In Code Snippet 5, it is checked if the database is present, if not, developers can create it using `initDatabase()` function.

Code Snippet 5: `database_service.dart`

```
Future<Database> initDatabase() async {
   final getDirectory = await
getApplicationDocumentsDirectory();
   String path = getDirectory.path + '/users.db';
   log(path);
   return await openDatabase(path, onCreate: _onCreate,
version: 1);
 }
```

In this function, first, the path is declared for creating/accessing the database by making use of the `path_provider` package. The file is named as `users.db`. Then, the `openDatabase()` function is called, which creates the database using `_onCreate()` function. Refer to Code Snippet 6.

Code Snippet 6: `database_service.dart`

```
void _onCreate(Database db, int version) async {
   await db.execute(
       'CREATE TABLE Users(id TEXT PRIMARY KEY, name TEXT,
email TEXT, age INTEGER)');
   log('TABLE CREATED');
 }
```

Table 11.1 lists the data types between Dart and SQLite.

Dart Type	SQLite Type
int	INTEGER
num	REAL
String	TEXT
Uint8List	BLOB

Table 11.1: Data Types Between Dart and SQLite

11.3 Executing SQLite Queries in Flutter

To execute SQL Queries in Flutter, `sqflite` package offers two ways. One, using raw queries and other using built-in SQL Helper functions. Both of them offer the same features so it is upto the user to select.

11.3.1 Executing Raw SQL Queries

SQL queries have to be written in a traditional method called `String`. This will be useful in cases where complex queries have to be written, otherwise helper functions are enough. Alternatively, this can be used if the developer possesses knowledge in writing SQL queries. Refer to Code Snippet 7 for examples of SQL queries.

Code Snippet 7: Examples of using Raw Queries

```
//create

db.execute(
       'CREATE TABLE Users(id TEXT PRIMARY KEY, name TEXT,
email TEXT, age INTEGER)');

//rawInsert
db.rawInsert(
       'INSERT INTO Users(id, name, email, age )
VALUES(?,?,?,?)',
       [user.id, user.name, user.email, user.age]);

//rawUpdate
db.rawUpdate('UPDATE Users SET name=?,email=?,age=? WHERE
ID=?', [user.name, user.email, user.age, user.id]);

//rawQuery
db.rawQuery('SELECT * FROM Users');
```

11.3.2 Executing Queries Using SQL Helpers

SQL helper functions make writing queries easier. Only the necessary arguments have to be passed to execute any query.

Code Snippet 8 shows the `insert` helper.

Code Snippet 8:

```
db.insert('Users', user.toMap());
```

The table name and data object must be passed in the form of a `Map<String, dynamic>` as arguments. The return type will be a `Future<int>` which will be the inserted `id` from the table.

Code Snippet 9 shows the `update` helper.

Code Snippet 9:

```
db.update('Users', user.toMap(), where: 'id = ?', whereArgs:
[user.id]);
```

For this, developers have to pass the table name, data to be updated, and conditions to be checked before inserting. If no conditions match, then the update will not take place.

Code Snippet 10 shows the `query` helper.

Code Snippet 10:

```
db.query('Users');
```

This is a simple query to return all the data from the table in the form of `List<Map<String, dynamic>>` where conditions can also be specified.

Code Snippet 11 shows the `delete` helper.

Code Snippet 11:

```
db.delete('Users', where: 'id = ?', whereArgs: [id]);
```

To delete a row from the table, this method is used along with the `where` condition.

11.3.3 Understanding Raw SQL queries

Developers can design a simple UI as shown in Code Snippet 12 to work with Raw SQL queries.

Code Snippet 12: `sqflite_example.dart`

```
import 'package:flutter/material.dart';
import 'package:session11/sqflite/database_service.dart';
import 'package:session11/sqflite/user_model.dart';
import 'package:uuid/uuid.dart';

class SqfliteExampleScreen extends StatefulWidget {
 const SqfliteExampleScreen({Key? key}) : super(key: key);
```

```
 @override
 State<SqfliteExampleScreen> createState() =>
_SqfliteExampleScreenState();
}

class _SqfliteExampleScreenState extends
State<SqfliteExampleScreen> {
 final dbService = DatabaseService();
 final nameController = TextEditingController();
 final ageController = TextEditingController();
 final emailController = TextEditingController();

 void showBottomSheet(String functionName, Function()?
onPressed) {
   showModalBottomSheet(
       context: context,
       elevation: 5,
       isScrollControlled: true,
       builder: (_) => Container(
             padding: EdgeInsets.only(
               top: 15,
               left: 15,
               right: 15,
               bottom:
MediaQuery.of(context).viewInsets.bottom + 120,
             ),
             child: Column(
               mainAxisSize: MainAxisSize.min,
               crossAxisAlignment: CrossAxisAlignment.end,
               children: [
                 TextField(
                   controller: nameController,
                   decoration: const
InputDecoration(hintText: 'Name'),
                 ),
                 const SizedBox(
                   height: 10,
                 ),
                 TextField(
                   controller: emailController,
                   keyboardType: TextInputType.emailAddress,
                   decoration: const
InputDecoration(hintText: 'Email'),
                 ),
                 const SizedBox(
                   height: 10,
```

```
                ),
                TextField(
                  controller: ageController,
                  keyboardType: TextInputType.number,
                  decoration: const
InputDecoration(hintText: 'Age'),
                ),
                const SizedBox(
                  height: 20,
                ),
                ElevatedButton(
                  onPressed: onPressed,
                  child: Text(functionName),
                )
              ],
            ),
          ));
 }

 void addUser() {
   showBottomSheet('Add User', () async {
     var user = UserModel(
         id: Uuid().v4(),
         name: nameController.text,
         email: emailController.text,
         age: int.parse(ageController.text));
     dbService.insertUser(user);
     setState(() {});
     nameController.clear();
     emailController.clear();
     ageController.clear();
     Navigator.of(context).pop();
   });
 }

 void editUser(UserModel user) {
   nameController.text = user.name;
   emailController.text = user.email;
   ageController.text = user.age.toString();
   showBottomSheet('Update User', () async {
     var updatedUser = UserModel(
         id: user.id,
         name: nameController.text,
         email: emailController.text,
         age: int.parse(ageController.text));
     dbService.editUser(updatedUser);
```

```
    nameController.clear();
    emailController.clear();
    ageController.clear();
    setState(() {});
    Navigator.of(context).pop();
  });
 }

 void deleteUser(String id) {
   dbService.deleteUser(id);
   setState(() {});
 }

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: const Text('Sqflite Example'),
     ),
     body: FutureBuilder<List<UserModel>>(
         future: dbService.getUsers(),
         builder: (context, snapshot) {
           if (snapshot.connectionState ==
ConnectionState.waiting) {
             return const Center(child:
CircularProgressIndicator());
           }
           if (snapshot.hasData) {
             if (snapshot.data!.isEmpty) {
               return const Center(
                 child: Text('No Users found'),
               );
             }
             return ListView.builder(
               itemCount: snapshot.data!.length,
               itemBuilder: (context, index) => Card(
                 color: Colors.yellow[200],
                 margin: const EdgeInsets.all(15),
                 child: ListTile(
                     title: Text(snapshot.data![index].name +
                         ' ' +

snapshot.data![index].age.toString()),
                     subtitle:
Text(snapshot.data![index].email),
                     trailing: SizedBox(
```

```
                        width: 100,
                        child: Row(
                          children: [
                            IconButton(
                              icon: const Icon(Icons.edit),
                              onPressed: () =>
editUser(snapshot.data![index]),
                            ),
                            IconButton(
                              icon: const Icon(Icons.delete),
                              onPressed: () =>

deleteUser(snapshot.data![index].id),
                            ),
                          ],
                        ),
                      )),
                ),
              );
            }
            return const Center(
              child: Text('No Users found'),
            );
          }),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: () => addUser(),
      ),
    );
  }
}
```

Explanation:

- List of users - name, age, and email will be displayed as *`ListTile`* inside a `ListView`.
- The `ListView` has been wrapped inside a `FutureBuilder` since the list of users will be fetched from the database. All the functions involving communication with the database is a *`future`* function.
- The loading indicator and 'no user found' text are displayed to enhance the UI.
- Floating action button is used, which on clicking opens a bottom area containing three text fields. In this area, name, email, and age can be entered.

- A package called `uuid` is used which will create a unique id for the user.
- Individual functions have been written to perform CRUD operations and communicate with the `DatabaseService` class.
- `addUser` - adds a user to the database, refreshes the app using `setState` which will rebuild the UI and text fields will be cleared in the same way as other functions.

Note: `setState` is used here for demonstration purposes only. Even though this works, it is not a good practice to use `setState`. Proper state management has to be used wherever possible, to avoid unnecessary UI rebuilds.

Code Snippet 13 shows adding of these functions in `database_service.dart` to use raw SQL queries:

Code Snippet 13:

```
Future<List<UserModel>> getUsers() async {
   final db = await _databaseService.database;
   var data = await db.rawQuery('SELECT * FROM Users');
   List<UserModel> users =
       List.generate(data.length, (index) =>
UserModel.fromJson(data[index]));
   print(users.length);
   return users;
 }

 Future<void> insertUser(UserModel user) async {
   final db = await _databaseService.database;
   var data = await db.rawInsert(
       'INSERT INTO Users(id, name, email, age )
VALUES(?,?,?,?)',
       [user.id, user.name, user.email, user.age]);
   log('inserted $data');
 }

 Future<void> editUser(UserModel user) async {
   final db = await _databaseService.database;
   var data = await db.rawUpdate(
       'UPDATE Users SET name=?,email=?,age=? WHERE ID=?',
       [user.name, user.email, user.age, user.id]);
   log('updated $data');
 }
```

```
 Future<void> deleteUser(String id) async {
   final db = await _databaseService.database;
   var data = await db.rawDelete('DELETE from Users WHERE
id=?', [id]);
   log('deleted $data');
 }
```

11.3.4 Using SQL Queries with SQL Helpers

Modify the database function in `database_service.dart` file as given in Code Snippet 14, to see how SQL Helper functions work.

Code Snippet 14:

```
Future<List<UserModel>> getUsers() async {
   final db = await _databaseService.database;
   var data = await db.query('Users');
   List<UserModel> users =
       List.generate(data.length, (index) =>
UserModel.fromJson(data[index]));
   print(users.length);
   return users;
 }

 Future<void> insertUser(UserModel user) async {
   final db = await _databaseService.database;
   var data = await db.insert('Users', user.toMap());
   log('inserted $data');
 }

 Future<void> editUser(UserModel user) async {
   final db = await _databaseService.database;
   var data = await db
       .update('Users', user.toMap(), where: 'id = ?',
whereArgs: [user.id]);
   log('updated $data');
 }

 Future<void> deleteUser(String id) async {
   final db = await _databaseService.database;
   var data = await db.delete('Users', where: 'id = ?',
whereArgs: [id]);
   log('deleted $data');
 }
```

Figure 11.1 shows output for the application after Code Snippet 14 is added to `databaseservice.dart`.

Output:

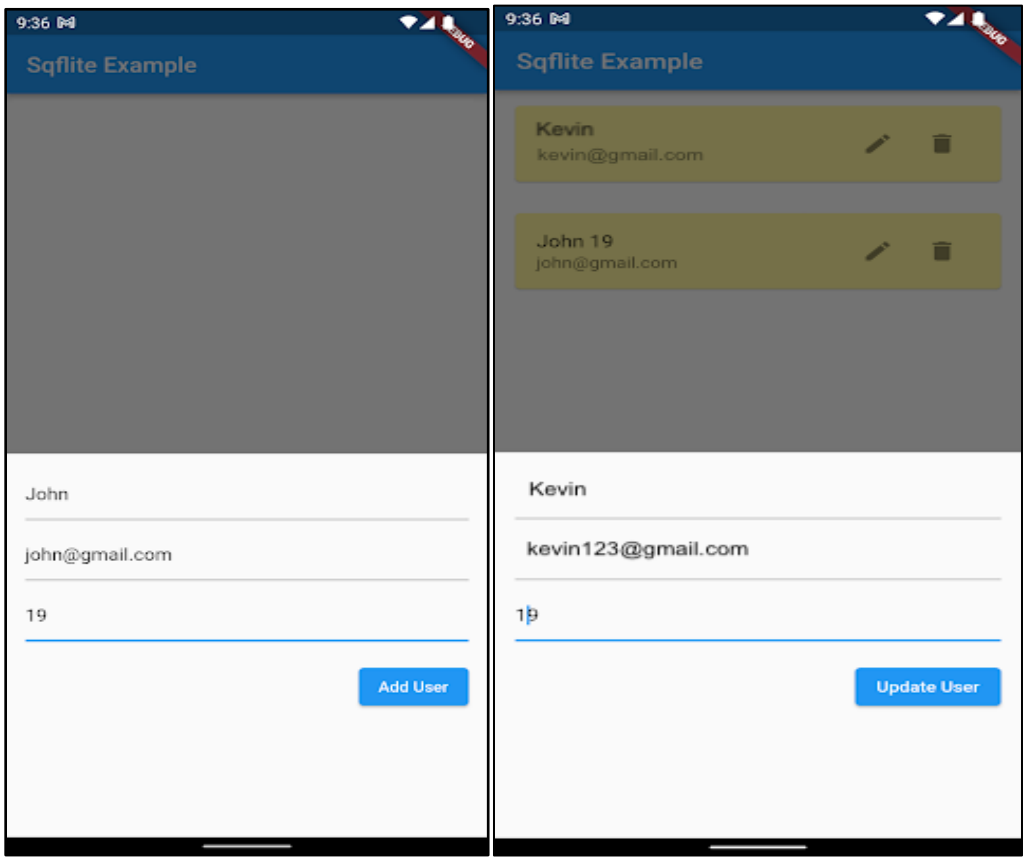


Figure 11.1: Output of SqfliteExample Application

11.4 Working with Firebase Database in Flutter

It would be immensely useful if data can be stored in a cloud and can be accessed from anywhere, through any device. Google's Firebase Database is a NoSQL cloud database where data is synced across all devices in real-time.

Hence, one can access, update, and sync data across multiple cross-platform applications.

11.4.1 Firebase Database Setup

To get started with Firebase, go to https://console.firebase.google.com and sign in using a Google account. Then, click 'New project' to create a new Firebase project. Refer to Figure 11.2.

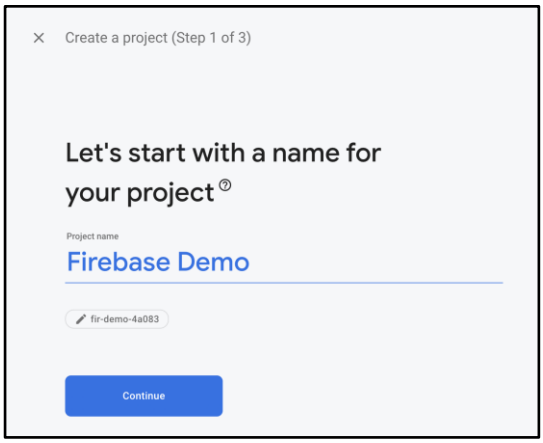


Figure 11.2: Firebase Project Setup

Provide a name for the project and click 'Continue'. The next step would be to enable/disable Google Analytics for this project. This step is optional and can be skipped. After a few minutes, the project will be ready. Firebase offers many features such as authentication and hosting. To begin with Firebase real-time database, follow the steps as shown in Figure 11.3.

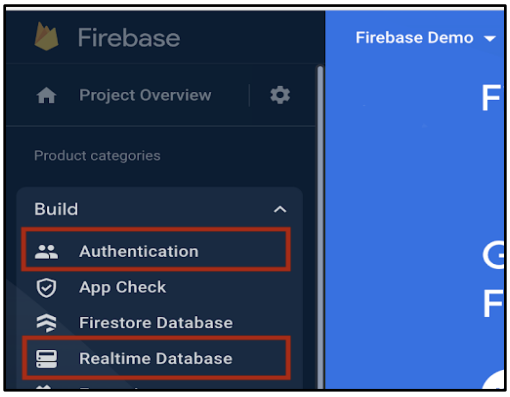


Figure 11.3: Firebase Features Setup

Click the real-time database under build from the left panel. Then, click 'create database'. Choose the database location (any location) and select/skip 'security rules'. For now, developers can start using the database in test mode which can be modified later. Select 'start' in test mode and click 'enable'. Real-time database is ready to use.

11.4.2 Adding Real-time Database to Flutter Application

Let us build a Flutter application where a real-time database is used to perform CRUD operations. Here, the same example is used as mentioned in the SQLite section earlier.

The app allows creating and displaying users and the user data will be stored in the Firebase real-time database.

Add following dependency in the `pubspec.yaml` file of the Flutter project:

```
firebase_database: ^9.1.0
```

11.4.3 Implementing Read and Write in Firebase Database

Before interacting with the database, an instance of `DatabaseReference` has to be created. Developers can create a `FirebaseService` class where all the Firebase operations will be handled and then, create this instance.

Refer to Code Snippet 15.

Code Snippet 15: `firebase_service.dart`

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

DatabaseReference databaseRef =
FirebaseDatabase.instance.ref('/users');
```

The `databaseRef` variable has been initialized pointing to the '*/users*' collection. Using this `databaseRef`, one can now communicate with the Firebase real-time database.

To read from a database, use the `get()` method which will return `Future<DataSnapshot>`. Since this is a NoSQL pattern, the data will be nested and the children and child methods can be used to access the required branch.

The data structure looks as shown in Figure 11.4.

Figure 11.4: Real-time Database Structure

Under users, there is a list of users with user id as key and the value will be the user model created.

Hence, the `databaseRef` created will point to the user's list and can be accessed using *`get()`* method.

```
databaseRef.get();
```

To access all the users from the list, use the getter children and convert them to a list as shown in Code Snippet 16.

Code Snippet 16: `firebase_service.dart`

```
var data = await databaseRef.get();
   List<DataSnapshot> userSnapshotList =
       data.children.toList();
```

To write to a database, the `set` method is used. Data has to be provided in the form of `Map<String, dynamic>`.

```
databaseRef.child('/${user.id}').set(user.toMap());
```

11.4.4 Adding HTTP Package to `pubspec.yaml`

Firebase database communication can also be made as a *REST API*. To use that, one must add the `http` package to Flutter app. For the current example, a REST API will not be used.

11.4.5 Creating an Application to Add Data to Firebase Database

An `addUser()` function can be created that will retrieve user data from input and add that to the Firebase database. Code Snippet 17 shows the code for this.

Code Snippet 17: `firebase_service.dart`

```
import 'package:firebase_database/firebase_database.dart';
import 'user_model.dart';

class FirebaseService {
 static DatabaseReference databaseRef =
     FirebaseDatabase.instance.ref('/users');
 addUser(User user) {
   databaseRef.child('/${user.id}').set(user.toMap());
 }
}
```

11.4.6 Creating an Application to Read Data From Firebase Database

The `getUsers` will return `List<UserModel>`. Here, `get()` and get method for children can be used to convert to `List<DataSnapshot>`. Then, `DataSnapshot` is converted to a user using the `User.fromSnapshot` function which is specified in the `UserModel` file. Refer to Code Snippets 18 and 19.

Code Snippet 18: `user_model.dart`

```
factory UserModel.fromSnapshot(DataSnapshot data) =>
UserModel(
       id: data.child('id').value as String,
       name: data.child('name').value as String,
       email: data.child('email').value as String,
       age: data.child('age').value as int,
     );
```

Code Snippet 19: `firebase_service.dart`

```
Future<List<UserModel>> getUsers() async {
   List<UserModel> users = [];
   var data = await databaseRef.get();
   List<DataSnapshot> userSnapshotList =
data.children.toList();
   for (var user in userSnapshotList) {
```

```
    users.add(UserModel.fromSnapshot(user));
  }
  return users;
 }
```

For the UI part, it is similar to the one changed in `SqfliteExample`. Before using Firebase, it is required to make some changes in `main.dart`. Refer to Code Snippet 20.

Code Snippet 20:

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

void main() async {
 WidgetsFlutterBinding.ensureInitialized();
 await Firebase.initializeApp();

 runApp(const MyApp());
}
```

The Firebase is being initialized using `Firebase.initializeApp()` which is obtained from `firebase_core` package. Code Snippet 21 is only to explain how to read and write to the Firebase database from Flutter application.

Code Snippet 21: `firebase_example.dart`

```
import 'package:flutter/material.dart';
import 'package:session11/firebase_service.dart';
import 'package:session11/user_model.dart';
import 'package:uuid/uuid.dart';


class FirebaseExampleScreen extends StatefulWidget {
 const FirebaseExampleScreen({Key? key}) : super(key: key);

 @override
 State<FirebaseExampleScreen> createState() =>
_FirebaseExampleScreenState();
}

class _FirebaseExampleScreenState extends
State<FirebaseExampleScreen> {
 final dbService = FirebaseService();
```

```
 // declare text editing controllers here

 void showBottomSheet(String functionName, Function()?
onPressed) {
  //same code from sqflite example
 }

 void addUser() {
   showBottomSheet('Add User', () async {
     var user = UserModel(
         id: Uuid().v4(),
         name: nameController.text,
         email: emailController.text,
         age: int.parse(ageController.text));
     dbService.addUser(user);
     setState(() {});
     nameController.clear();
     emailController.clear();
     ageController.clear();
     Navigator.of(context).pop();
   });
 }

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: const Text('Firebase database Example'),
     ),
     body: FutureBuilder<List<UserModel>>(
         future: dbService.getUsers(),
         builder: (context, snapshot) {
           if (snapshot.connectionState ==
ConnectionState.waiting) {
             return const Center(child:
CircularProgressIndicator());
           }
           if (snapshot.hasData) {
             if (snapshot.data!.isEmpty) {
               return const Center(
                 child: Text('No Users found'),
               );
             }
             return ListView.builder(
                 itemCount: snapshot.data!.length,
                 itemBuilder: (context, index) {
```

```
                    var userData = snapshot.data![index];
                    return Card(
                      color: Colors.yellow[200],
                      margin: const EdgeInsets.all(15),
                      child: ListTile(
                        title: Text('${userData.name}
${userData.age}'),
                        subtitle: Text(userData.email),
                      ),
                    );
                  });
            }
            return const Center(
              child: Text('No Users found'),
            );
          }),
      floatingActionButton: FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: () => addUser(),
      ),
    );
  }
}
```

So far, a function to get all users from Firebase database and a factory method to parse the return type data to a UserModel are written.

In Code Snippet 21, a UI similar to `SqfliteExample` is created.

Here, a `FutureBuilder` is used with the Future method `getUsers()`. This method is implemented inside `firebase_service.dart` and all users will be listed inside the `ListView` widget. 'No Users found' will be displayed if the list is empty.

To add a user to the database, click `FloatingActionButton` and design a similar sheet as shown in `SqfliteExample` and call the `addUser` function (As shown in Code Snippet 17).

11.4.7 Firebase Authentication

Firebase offers authentication features by which end users can sign up or sign-in using email, password, and many other ways such as Google sign-in, mobile number sign-in, and so on.

To enable Firebase Authentication for the project, go to the project in the Firebase console. Under build, click 'authentication'. After enabling, click the sign-in methods tab and enable the Email/Password sign-in method. Next, it will be explained as to how this feature can be used in the Flutter app. Refer to Figure 11.5.

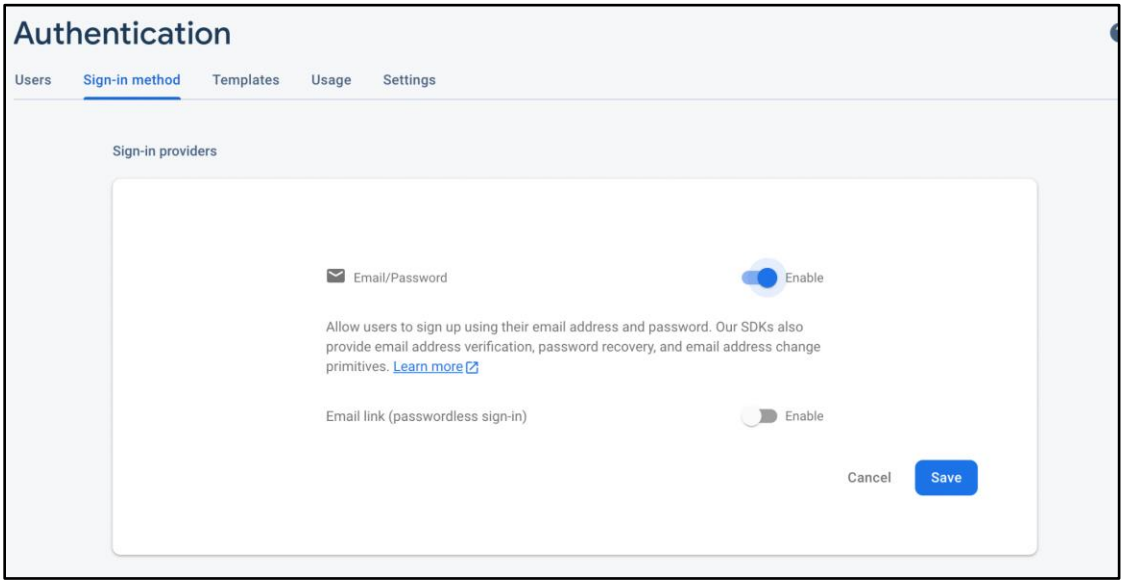


Figure 11.5: Firebase Auth Setup

11.4.8 Setting Up Firebase for Android, IOS, and Web

Setting up Firebase for one's project requires adding some dependencies in the application and defining those apps from the Firebase console. Figure 11.6 shows three icons for Android, iOS, and Web.

Click the type of app required to be connected and follow the instructions as given on the screen.

Provide the package name, download the `google-services.json` file, and place it in an appropriate location on local system. Refer to Figure 11.6.

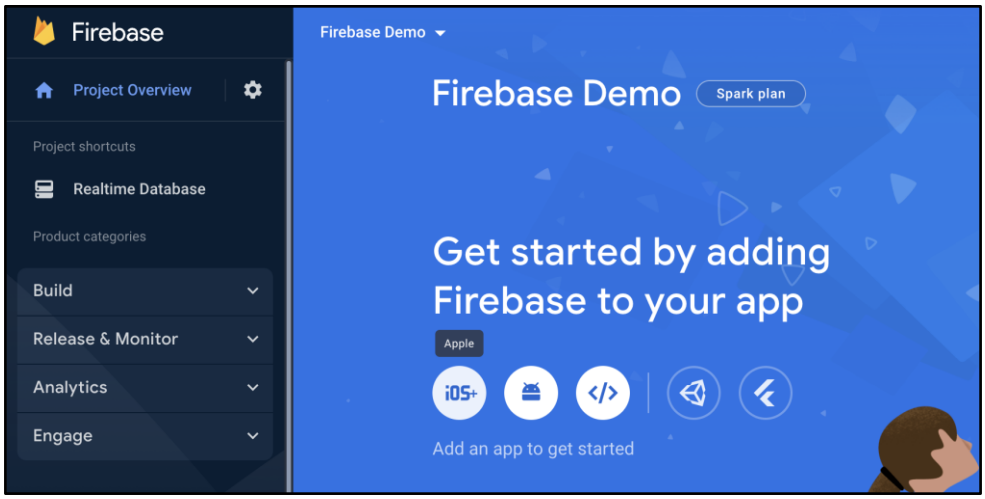


Figure 11.6: Setup Firebase for Android, IOS, and Web

11.4.9 Adding Dependencies to `pubspec.yaml`

a. firebase_core

This package is used to initialize Flutter applications with Firebase. It is responsible for establishing connections between the Firebase project and Flutter app.

b. firebase_auth

The authentication feature from Firebase can be used by installing this package. Add following dependencies in `pubspec.yaml` as shown in Code Snippet 22.

Code Snippet 22:

```
dependencies:
 flutter:
   sdk: flutter
 cupertino_icons: ^1.0.2
 firebase_core: ^1.20.0
 firebase_auth: ^3.6.2
```

11.4.10 Initializing Firebase App

To initialize Firebase in the application, add following code as shown in Code Snippet 23 in `main.dart` file:

Code Snippet 23:

```
void main() async {
```

```
 WidgetsFlutterBinding.ensureInitialized();
 // initialize firebase
 await Firebase.initializeApp();
 runApp(MyApp());
}
```

11.4.11 Register a New User in Firebase

The Firebase operations will be handled inside a service class as the norm. Create an instance of `FirebaseAuth` using `FirebaseAuth.instance`.

Here, in Code Snippet 24, the email and password sign-in method is being used. To create a new registration, email, and password are used, as well as `displayName` for the user.

Code Snippet 24: `firebase_service.dart`

```
class FirebaseService {
 FirebaseAuth firebaseAuth = FirebaseAuth.instance;
  Future<bool> register(String name, String email, String
password) async {
   firebaseAuth.currentUser!.reload();
   try {
     var userCredential = await
firebaseAuth.createUserWithEmailAndPassword(
         email: email, password: password);
     await firebaseAuth.currentUser!.updateDisplayName(name);
     if (userCredential.user != null) {
       return true;
     }
     return false;
   } catch (e) {
     print(e);
     return false;
   }
 }

}
```

Use the `createUserWithEmailAndPassword` function to register a new user in Firebase. Then, to give a name to the user, update the username by the `updateDisplayName` method. One can also provide phoneNumber, photoUrl, and so on.

11.4.12 Create User Sign-in and Sign-Out

After registering, `SignIn` and `SignOut` functions must be used. Refer to Code Snippet 25.

Code Snippet 25: `firebase_service.dart`

```
Future<bool> signIn(String email, String password) async {
   try {
     var userCredential = await
firebaseAuth.signInWithEmailAndPassword(
         email: email, password: password);
     if (userCredential.user != null) {
       return true;
     }
     return false;
   } catch (e) {
     print(e);
     return false;
   }
 }
```

Use the `signInWithEmailAndPassword()` function to sign in. The use of `try- catch` blocks is a good practice to catch any errors. If the user is present in Firebase, the return type `UserCredential` will contain an Object called *User*, which has the user data such as `email` and `displayName`.

If the User object is null, then the user is not present and signIn has failed.

For SignOut, use the `signOut()` function. Refer to Code Snippet 26.

Code Snippet 26: `firebase_service.dart`

```
Future signOut() async {
  await firebaseAuth.signOut();
}
```

11.4.13 Refresh User and Define Validators

`FirebaseAuth` has an object called `currentUser` which will hold the data of the user logged in currently.

To refresh the `currentuser`, use following `currentUser.reload()` method which will update the current user:

```
firebaseAuth.currentUser!.reload();
```

11.4.14 Build Sign-in Form and Profile Page

The existing Firebase database example can be used here. In addition to that, a login page and register page is added. Hence, only those who are authenticated will be able to navigate to the Firebase database screen. The Firebase database screen will also contain email and `displayName` of the user which will be fetched using the `firebaseAuth` instance and a logout button.

Design a simple login screen with email and password fields and a register screen with name, email, and password fields. Refer to Figure 11.7.

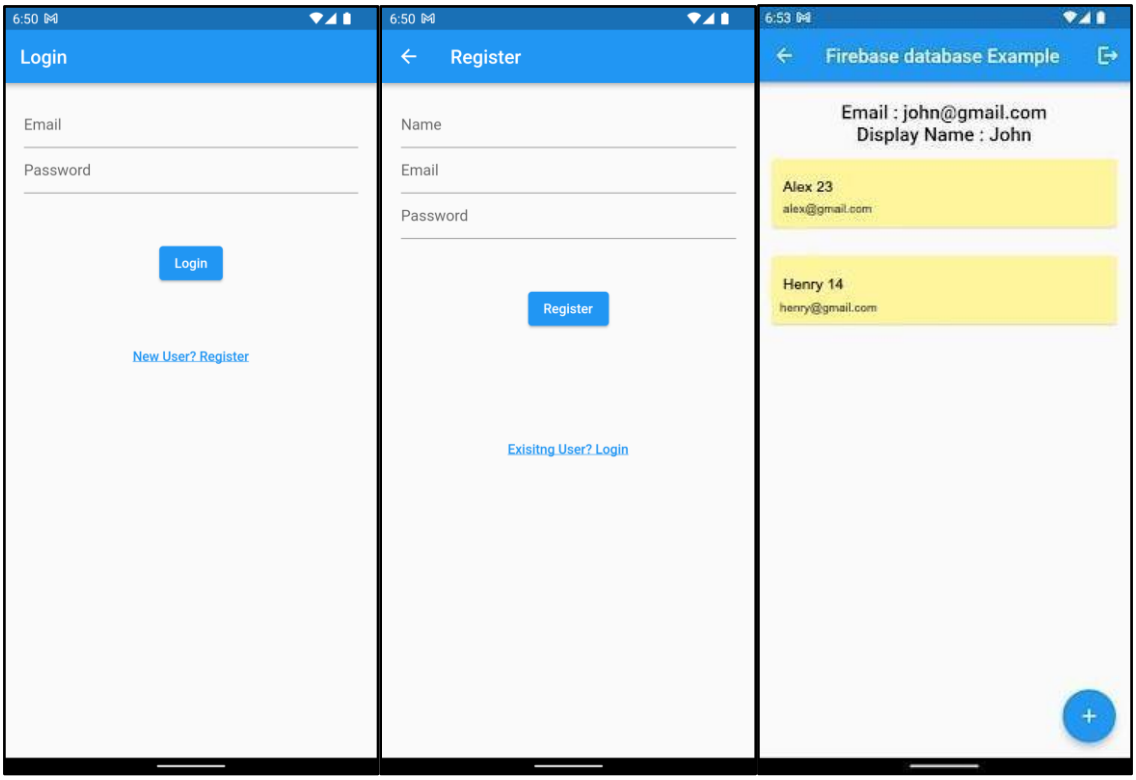


Figure 11.7: Flutter Firebase Application

Use of the Register and Login buttons functions are shown in Code Snippets 27 and 28.

Code Snippet 27: Register functionality in register_screen.dart

```
ElevatedButton(
              onPressed: () async {
                bool isRegistered = await
firebaseService.register(nameController.text,
                    emailController.text,
passController.text);
                if (isRegistered) {
                  Navigator.push(
                      context,
                      MaterialPageRoute(
                          builder: (context) =>
FirebaseExampleScreen()));
                } else {
                  log('registration failed');
                }
              },
              child: Text('Register'),
            ),
```

Code Snippet 28: Login functionality in login_screen.dart

```
ElevatedButton(
              onPressed: () async {
                bool isLogin = await firebaseService.signIn(
                    emailController.text,
passController.text);
                if (isLogin) {
                  Navigator.push(
                      context,
                      MaterialPageRoute(
                          builder: (context) =>
FirebaseExampleScreen()));
                } else {
                  log('login failed');
                }
              },
              child: Text('Login'),
            ),
```

The `email` and `displayName` can be fetched using Code Snippet 29.

Code Snippet 29

```
final fbService = FirebaseService();

fbService.firebaseAuth.currentUser!.email
fbService.firebaseAuth.currentUser!.displayName
```

11.4.15 Retaining Login State for User

It is not a good practice to force the user to login each time they open the app. Firebase gives the option to retain the login state of the user.

To maintain the auth state in the application, use `authStateChanges()` stream which will return a user if logged in, or null if no user is logged in.
In the `main.dart` file, this condition can be checked and based on whether the user is logged in or not, one can navigate to `LoginScreen` or `DatabaseScreen`. Refer to Code Snippets 30 and 31.

Code Snippet 30: `main.dart` -> Inside `void main()`

```
. . .
//check auth state

FirebaseService().firebaseAuth.authStateChanges().listen((use
r) {
   if (user == null) {
     runApp(const MyApp(isLogin: false));
   } else {
     runApp(const MyApp(isLogin: true));
   }
 });
```

Code Snippet 31: `main.dart`

```
. . .
class MyApp extends StatelessWidget {
 final bool isLogin;
 const MyApp({Key? key, required this.isLogin}) : super(key:
key);
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     debugShowCheckedModeBanner: false,
     title: 'Firebase Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
```

```
      ),
      home: isLogin ? FirebaseExampleScreen() : LoginScreen(),
    );
  }
}
```

In this manner, developers can use Firebase to create and manipulate real-time data in Flutter applications. Many practical real-world applications make use of this.

11.5 Summary

- SQLite is a relational DBMS through which data can be stored in the form of tables. SQLite in Flutter helps in easy access and performing complex queries on the locally stored data.
- Flutter has `sqflite` plugin using which data can be stored locally on user device as Relational Tables.
- All operations related to database are Future.
- Both raw queries and helper functions can be used to write SQL queries.
- Firebase offers a lot of features and is a good option for mobile backend.
- Real-time database offered by Flutter is a NoSQL database where data will be stored in the form of JSON.
- Return type of Firebase Database will be `DataSnapshot`.
- Firebase Auth can be used to authenticate users of an application.
- Firebase auth has many sign-in methods such as email and phone number, and GoogleSignIn.
- Firebase auth also has the option to retain the auth state of the user in the application.
- Firebase offers multiple features, thus, it is fully capable of being used as a backend for mobile and cross platform apps.

11.6 Test Your Knowledge

1) SQLite is a _____.
   a. Relational Database
   b. NoSQL Database
   c. Cloud Database
   d. Object-Oriented Database

2) Which one of following is the correct code to insert data in a table using sqflite?
   a. `db.rawUpdate('UPDATE Table(name, age) VALUES(?,?)', [name, age]);`
   b. `db.rawInsert('INSERT INTO Table(name, age) VALUES(?,?)', [name, age]);`
   c. `db.insert('INSERT INTO Table(name, age) VALUES(?,?)', [name, age]);`
   d. `db.rawQuery('INSERT INTO Table(name, age) VALUES(?,?)', [name, age]);`

3) Which one among following is not a feature of Firebase?
   a. Firebase Authentication
   b. Real-time Database
   c. Internationalization
   d. Hosting

4) What method of FirebaseAuth should be used to register a new user using email and password?
   a. `createUserWithEmailAndPassword(email: email, password: password);`
   b. `signInWithEmailAndPassword(email: email, password: password);`
   c. `signUpWithEmailAndPassword(email: email, password: password);`
   d. `createUser(email: email, password: password);`

5) What dependencies should be added to use Firebase, perform authentication, and use real-time database?
   a. `firebase_auth, firebase_database`
   b. `cloud_firestore, firebase_auth`
   c. `firebase_core, firebase_auth, firebase_database`
   d. `firebase_storage, firebase_database`

Answers - Test Your Knowledge

1. Relational Database
2. `db.rawInsert('INSERT INTO Table(name, age) VALUES(?,?)', [name, age]);`
3. Internationalization
4. `createUserWithEmailAndPassword(email: email, password: password);`
5. `firebase_core`, `firebase_auth`, `firebase_database`

11.7 Try It Yourself

1) Create a Chat application using Firebase.
    a) Users can create an account and view all available members.
    b) Use FirebaseAuth for SignIn. Make sure to preserve the Auth state.
    c) They can click any user and start chatting. The chat should be live, that is, any new message received or sent should be updated as Stream.
    d) **Bonus:** Make use of FirebaseStorage and enable users to share images.

2) Enhance the chat application by enabling offline feature.
    a) Store all the chat messages offline using SQLite.
    b) Users can read existing messages when they have no Internet connection. They can send messages once they are online.

Session 12
REST API in Flutter

Learning Objectives

In this session, students will learn to:

- ➢ Explain REST APIs
- ➢ Define and explain utilizing APIs for REST in Flutter
- ➢ Explain the use of `http` package
- ➢ Outline the core methods of the HTTP component
- ➢ Explain the use of Flutter apps to showcase REST APIs
- ➢ Explain how to register and log in utilizing sample Web APIs

***Session Goals**:*
The session introduces concepts of REST Application Programming Interfaces (APIs), describes their usage in Flutter programs, and explains core functions of the HTTP component request, response, headers, body, and other essential parameters of HTTP requests. In addition, examples of how to fetch data via REST APIs inside a Flutter app, Registration, and Login employing REST APIs are included.

12.1 REST API in Flutter and its Uses

There are plenty of modes by which Flutter displays information in our application. For instance, one could use data such as static information and data via databases, files, and open APIs.

APIs are the most popular form of data that can be integrated with applications.

Any API that adheres to the restrictions of a REST Architecture style and permits communication among RESTful Web services is defined as a REST API (often called RESTful API). In simple terms, APIs are the communication bridge between the front and back end. Backend data will be shared as REST APIs, which can be consumed in the front end.

Flutter offers some packages, such as `http` and `Dio`, through which one can make API calls and fetch data. These data could be JSON (most common), XML, or simple String, which could be modeled as per convenience.

HTTP module supplies top-level classes with `http` in making Web requests. These functions take a URL and other content via Dart Map (post metadata, extra headers, and more). Then, it sends a query to the host and waits for the answer before sending it back.

12.2 Getting Started with REST API in Flutter

Steps involved in utilizing REST APIs with Flutter are:
Step 1: Adding dependency.
Step 2: Get the API URLs and Endpoint and have a good understanding of those APIs.
Step 3: Send an HTTP request and receive a response.
Step 4: Map the reply to a custom model class (it is optional but will be very helpful for large and complex responses).
Step 5: Utilize `FutureBuilder` and display information within UI.

12.2.1 Adding HTTP dependency to `pubspec.yaml`

Append HTTP dependence and recent version inside `pubspec.yaml` document as shown in Code Snippet 1.

Code Snippet 1:

```
dependencies:
 flutter:
   sdk: flutter
 cupertino_icons: ^1.0.2
 http: ^0.13.5
```

Postman is the best tool to test an API. Utilize this to see how APIs return the information. Check the Postman official site at www.postman.com to discover more regarding it.

12.3 Core Methods of `http` Package

`http` delivers better standard HTTP client classes. Clients are automatically initialized whenever requests are created and are closed once requests are complete.

Classes in the `http` package deliver functionality to serve all sorts of HTTP requests.

12.3.1 `Read`

Utilize the `READ` function to make the necessary URL requests, then provide the reply as a `Future<String>`.

12.3.2 `Get`

Use the `GET` function to send a query to the URL and then, return the reply as a `Future<Response>`. Individual 'response' is a class containing the response information.

The difference between `READ` and `GET` lies in the response. Read returns only the body's response, whereas `GET` returns a response class that contains `headers`, `body`, `statusCode`, `contentLength`, `bodyBytes`, and many more.

12.3.3 `Post`

Call URL using `POST` by submitting supplied information and producing responses as `Future<Response>`.

Syntax of this function is:

```
package:http/http.dart
…
Future<Response> post(
 Uri url, {
 Map<String, String>? headers,
 Object? body,
 Encoding? encoding,
})
Type: Future<Response> Function(Uri, {Object? body, Encoding?
encoding, Map<String, String>? headers})
```

Here, the `post` method transmits HTTP `POST` queries to a specified link, including described headers and the content of the body. The `body` places the body of the request. It could be `String`, `Map<String, String>`, or `List`. When it is String, `encoding` is appended to it before being utilized as one content of the request. The query's standard content style is `text/plain`.

Each query body is treated as a byte collection when the `body` would be List. When the `body` would be Map, `encoding` transforms it into form fields. `application/x-www-form-urlencoded` would be utilized as content of the query types. Also, it cannot change. By design, `encoding` uses `utf8`. Utilize `Request` or even `StreamedRequest` for extra granular hold well across requests.

12.3.4 `Put`

Use a `PUT` function to make a query to the supplied link, and then, revert the reply as `Future<Response>`.

12.3.5 `Head`

Use a `HEAD` function to make a query to the supplied URLs and then deliver the result as `Future<Response>`.

12.3.6 `Delete`

Use a `DELETE` function to make a request for the provided Hyperlink and produce replies as `Future<Response>`.

12.4 Fetching Data from Server Using REST API

Let us implement a simple yet interesting API to fetch data from a Web server using REST API.

Cat Fact API is a readymade API that contains a list of APIs which will return some facts about cats.

The first API from this documentation is a `GET` method which will return a random fact about cats each time it is called as shown here:

```
// GET METHOD
// https://catfact.ninja/fact

response:
{
  "fact": "string",
  "length": 0
}
```

The response is simple with a fact of type `String` and the length of the fact as an `integer`. Figure 12.1 shows the output that will be generated by the application.

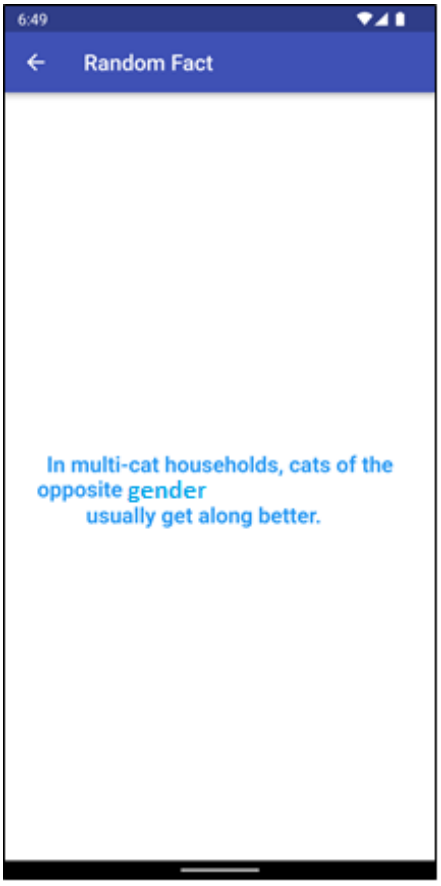


Figure 12.1: Output of the `RandomFactAPI` Application

Follow these steps to implement this API in Flutter:

Step 1: Start a brand-new Flutter project named ***`RandomFactAPI`*** and then, append the `http` dependency as given in Code Snippet 1.

Step 2: Make a brand-new class file named `ApiCall.dart` in which the ApiCall class would manage API calls.
Step 3: Load the `http` component as HTTP.
Step 4: Create a new function named `getRandomFact()`, which is a `Future` and will return the fact (of `string` type).
Step 5: Use `http.get()` function and pass URLs inside `Uri.parse()`. The responses here will be of type `response`.
Step 6: Retrieve the fact-value from the body. Then, decode `response.body` with `jsonDecode()` and access the `['fact']` value.
Step 7: Return the fact from the method after displaying it on the console.

Code for all steps from 2 onwards is shown in Code Snippet 2.

Code Snippet 2: `ApiCall.dart`

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

class ApiCall {
 Future<String?> getRandomFact() async {
   var response = await
http.get(Uri.parse('https://catfact.ninja/fact'));
   print(response);
   String result = jsonDecode(response.body)['fact'];
   print(result);
   return result;
 }
}
```

Step 8: Create a simple UI. The body must be a `FutureBuilder`, where one will call the `getRandomFact()` function and display the fact in a Text Widget. Refer to Code Snippet 3.

Code Snippet 3: `FactScreen.dart`

```
...
class FactScreen extends StatelessWidget {
...
body: Center(
       child: FutureBuilder<String?>(
           future: ApiCall().getRandomFact(),
           builder: (context, snapshot) {
```

```
            if (snapshot.connectionState ==
                          ConnectionState.waiting) {
              return const CircularProgressIndicator();
            }
            if (snapshot.hasData) {
              if (snapshot.data == null) {
                return const Text('No fact found');
              }
              return Text(
                snapshot.data!,
                textAlign: TextAlign.center,
                style: const TextStyle(
                    fontSize: 20,
                    fontWeight: FontWeight.w600,
                    color: Colors.blue),
              );
            }
            return Container();
          }),
    ),
```

That is all.

In `main.dart`, call `FactScreen` as follows:

```
import 'fact_screen.dart';
...
home: const FactScreen(),
```

API will be successfully called using the HTTP function and the fact displayed inside the UI.

The output will be the same as was shown in Figure 12.1.
Now, what if suppose developer wanted to show the facts only to valid authorized users and not any random user? To do this, one has to implement authorization.

12.5 Registering a User Using REST API

Authorization can be performed via REST APIs. Consider an open-source authentication API using which one can complete sign-up and sign-in. The URLs provided in Code Snippets 4 and 5 are sample APIs used for testing purposes.

For example, a secure authentication API would return a token, which will be used for further API queries inside the application. Tokens are Strings containing user data, yet the token would be validated for extra individual requests.

Step 1: Construct a new function `register` in the `ApiCall` class. The function will have `usernames`, `email`, and `password` parameters.
Step 2: Use `http.post()` function and pass any URLs.
Step 3: The content type should be passed within the headers.
Step 4: Define the name, e-mail, and password inside a body and encode it as JSON using `jsonEncode`.
Step 5: If the message key from the response body returns success, the user has been created successfully. Hence, one can return true. If not, there is an error and it can be handled. Here, the response is printed and returned false. Refer to Code Snippet 4.

Code Snippet 4:

```
...
Future<bool> registerAPI(String uname, String e-mail, String
password) async {
   var response = await http.post(

Uri.parse('http://restapi.adequateshop.com/api/authaccount/regi
stration'),
     headers: {'Content-Type': 'application/json'},
     body: jsonEncode({'name': uname, 'email': e-mail,
'password': password}),
   );
   var result = jsonDecode(response.body);
   print(result);
   if (result['message'] == 'success') {
     return true;
   }
   return false;
 }
```

Step 6: Create a simple Registration Screen UI, where username, e-mail, and password are fetched using `TextField` and `TextEditingControllers`.

Step 7: Add following method to the register button. The register API function is called `Future` and is based on the return bool type and navigation is handled. Refer to Code Snippet 5.

Code Snippet 5:

```
...
ElevatedButton(
             onPressed: () async {
               bool isRegistered = await
                         ApiCall().registerAPI(
                    nameController.text,
                    emailController.text,
                    passController.text);
               if (isRegistered) {
                 Navigator.push(context,
                     MaterialPageRoute(builder: (context) =>
                         FactScreen()));
               } else {
                 log('registration failed');
               }
             },
             child: Text('Register'),
           ),
```

Note that for the application to work successfully, a unique username and password must be supplied during registration. If the username and password are already taken, the application will not proceed.

Code Snippet 6 shows data for a sample Register API.

Code Snippet 6:

```
/* POST Function
Url:
http://restapi.adequateshop.com/api/authaccount/registration*/
//Request body:
{

            "name":"Enter full name",
            "email":"Input email",
            "password":"Enter the password."
}
//Content-Type: application/json
```

```
//Example API Response
{
            "$id": "01",
            "code": 0,
            "message": "success",
            "data": {
            "$id": "02",
            "Id": 7075,
            "Name": "App Developer",
            "Email": "App_Developer1@gmail.com",
            "Token": "3030401c-c5a5-43c8-8b73-2ab9e6f2ca22"
    }
}
//If e-mail is already in usage, then API Response.
{
            "$id": "01",
            "code": 1,
            "message": "Address was already stored here.",
            "data": null
}
```

API for Registrations is a POST method where one will pass the full name, e-mail, and password inside a body, including the request. The body consists of '`content-type: application/json`,' which remains to specify while passing the query. Each reply holds a message key that will return success with no failures.

Additionally, the reply includes big data, which could be modeled and operated throughout the app. Nevertheless, only registration success or none would be considered.

Code Snippet 7 shows sample response data from Sign-in API.

Code Snippet 7:

```
//POST Function
//Url: http://restapi.adequateshop.com/api/authaccount/login
//Request body:
{
            "email":"Input email",
            "password":"Enter the password"
}
//Content-Type: application/json
```

```
//Example API Response
{
    "$id": "01",
    "code": 0,
    "message": "success",
    "data": {
        "$id": "02",
        "Id": 7075,
        "Name": "App Developer",
        "Email": "App_Developer1@gmail.com",
        "Token": "02b869e4-ea45-4b5c-b764-642a39e95bb7"
    }
}
//API Reply when email or password is incorrect
{
    "$id": "01",
    "code": 1,
    "message": "Mail or perhaps Passwords are inexact.",
    "data": null
}
```

Output of Code Snippets 4 and 5 for a simple Sign-up and Login Pages are shown in Figure 12.2. Once authenticated, one might be redirected to a `FactScreen`.

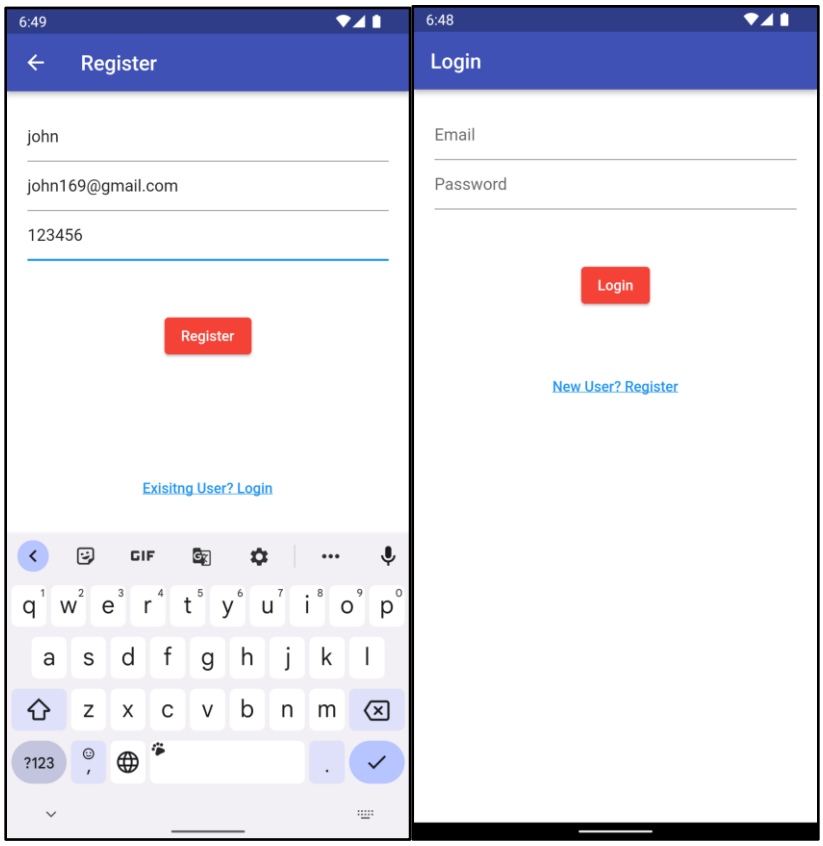


Figure 12.2: Output of Sign-up and Login Screens

12.6 Validating a Registered User using REST API

Following are the steps to create an application to login or validate a registered user using REST API:

Step 1: Create a login function with `email` and `password` as parameters.

Step 2: Use `http.post` method and pass the URL.

Step 3: Define content classification in the header.

Step 4: Pass e-mails and passwords as `Map<String, dynamic>`.

Step 5: Encode equivalent JSON.

Step 6: Based on whether the reply body notification is a success or not, return true or false. Refer Code Snippet 8.

Code Snippet 8:

```
Future<bool> loginAPI(String e-mail, String password) async {
   var response = await http.post(
Uri.parse('http://restapi.adequateshop.com/api/authaccount/login
'),
     headers: {'Content-Type': 'application/json'},
```

```
     body: jsonEncode({'email': e-mail, 'password':
password}),
   );
   var result_1 = jsonDecode(response.body);
   print(result_1);
   if (result_1['message'] == 'success') {
     return true;
   }
   return false;
 }
```

Step 7: Create a simple `LoginScreen` UI.
Step 8: Handle the login function in the Login Button.
Step 9: The `LoginAPI` is called inside the Button and it returns true or false in Future. The return value is stored in the `isLogin` variable. Based on the `isLogin` value, Navigate to Next Screen if the user has logged in or print login failed if `isLogin` is false. Refer Code Snippet 9.

Code Snippet 9:

```
ElevatedButton(
              onPressed: () async {
                bool isLogin = await
ApiCall().loginAPI(emailController.text, passController.text);
                if (isLogin) {
                  Navigator.push(
                      context,
                      MaterialPageRoute(
                          builder: (context) =>
FactScreen()));
                } else {
                  log('login failed'); }
              },
              child: Text('Login'),
            ),
```

Additionally, the `auth` state can be maintained using local storage similar `SharePreferences`. Once logged in, store the login state in a boolean value and keep it in `SharedPreferences`. So based on that value, one will be logged in directly and it will no longer be necessary to pass the login page.

12.7 Summary

- Representational State Transfer - Application Programming Interface is what REST API means.
- Flutter has packages comparable to `http`, `dio`, and `chopper` to communicate with APIs.
- The HTTP package provides class functionalities to perform `GET`, `POST`, `PUT`, `PATCH`, and `DELETE` methods.
- HTTP requests are asynchronous functions, and hence, they are `Future` functions.
- `FutureBuilder` is recommended for displaying HTTP responses in the UI.
- Dart model can simplify the reading and processing of HTTP responses.
- There are secure APIs that would verify the user token each time request is made.
- For Authentication APIs, `SharedPreferences` can be used to store user info and `auth` state of the particular user in a device.

12.8 Test Your Knowledge

1) How do `GET` and `POST` functions vary from one another?
   a. `POST` is more secure than `GET` since information is not sent in the URL.
   b. `GET` - information is transmitted in URL, `POST` - data is sent via body.
   c. `GET` – there can only be ASCII characters; `POST` - all data types are allowed.
   d. All of these.

2) Which among the following is factually incorrect?
   a. HTTP requests are asynchronous.
   b. Return types of HTTP requests are `Future<Response>` except for `READ` method.
   c. Default content-type for HTTP requests are `application/json`.
   d. `GET` requests are the least secure HTTP method.

3) What is the correct code to retrieve the length as `Strings` from the JSON response beneath?

```
{
  "fact": "string",
  "length": 0
}
```

   a. `jsonDecode(response.body)["length"].toString();`
   b. `response.body["length"];`
   c. `response["length"];`
   d. `jsonDecode(response)["length"].toString();`

4) What script should be used to submit the following HTTP query?

```
/*PUT method
URL: http://restapi.adequateshop.com/api/users
content-type: application/json*/
API Request
{
    "id": 7076,
     "name":"traveler change nickname",
     "e-mail":"traveler@protonmail.com",
     "location":"USA"
}
```

API Response

```
{
    "$id": "01",
    "id": 7076,
    "name": "traveler change nickname",
    "e-mail": "traveler@protonmail.com",
    "profilepicture":
"https://www.adequatetravel.com/ATMultimedia//images/userimageicon
.png",
    "location": "USA",
    "createdat": "2021-10-14T12:05:59.7235182"
}
```

a)

```
var response = await http.post(
     Uri.parse('http://restapi.adequateshop.com/api/users'),
     headers: {'Content-Type': 'application/json'},
     body: jsonEncode({
       "id": 7076,
       "name": "traveler change nickname",
       "e-mail": "traveler@protonmail.com",
       "location": "USA"
     }),
   );
```

b)

```
var response = await http.put(
     Uri.parse('http://restapi.adequateshop.com/api/users'),
     headers: {'Content-Type': 'application/json'},
     body: jsonEncode({
       "id": 7076,
       "name": "traveler change nickname",
       "e-mail": "traveler@protonmail.com",
       "location": "USA"
     }),
   );
```

c)

```
var response = await http.put(
     Uri.parse('http://restapi.adequateshop.com/api/users'),
     body: jsonEncode({
       "id": 7076,
       "name": "traveler change nickname",
       "e-mail": "traveler@protonmail.com",
       "location": "USA"
     }),
   );
```

d)
```
var response = await http.put(
     Uri.parse('http://restapi.adequateshop.com/api/users'),
          headers: {'Content-Type': 'application/json'},
          body:{
            "id": 7076,
            "name": "traveler change nickname",
            "e-mail": "traveler@protonmail.com",
            "location": "USA"
          },
        );
```

5) What are the differences between `http.get` and `http.read`?
   a. Both are the same
   b. They have a difference in request type
   c. They have a difference in response type
   d. They have a difference in headers and URL

Answers - Test Your Knowledge

1. All of these
2. Default content-type for HTTP requests are `application/json`
3. `jsonDecode(response.body)["length"].toString();`
4. Option - b
5. They have a difference in response type

12.9 Try It Yourself

1) For the application given in the session, add a new Text Widget that displays Login Failed when login is unsuccessful.
2) Go to https://openweathermap.org/api. Read the documentation carefully and use the APIs in a new Flutter Weather Application.
    a) Design an attractive UI with Splash Screen, Home Screen where Weather Data including details such as Humidity, Temperature, and Weather Status will be displayed.
    b) UI should contain graphics as per the weather that is Rainy, Cloudy, Hot, and so on.
    c) Get Location Permission from User while opening the app and display the weather data as per their location.

    Hint: You must create an account in OpenWeatherMap to get an api key and use their features.

3) In the Same Weather app, add a search feature, where users can get weather data for a particular city. OpenWeatherMap has an API for this feature.