

# Flutter Application Development



Android  
Studio

- Covers a wide range of flutter development topics.
- Demonstrates visual, behavioral and motion rich Flutter widgets.
- Displays step-by-step lab exercises to built flutter apps.
- Includes guides to build Google Maps apps.
- Presents Android and iOS app publishing guidelines.

By Android ATC Team

[www.androidatc.com](http://www.androidatc.com)

Android ATC

# Flutter™ Application Development

Exam Code: AFD-200

Hands-on Guide to Flutter Development

*Because this book is being published at a time of a global pandemic, this book is dedicated to all front-line workers who putting their lives at risk battling COVID-19 to save our lives.*

*Doctors, nurses, laboratory staff, researchers, volunteers, janitors and everyone in the health system, you are the superheroes battling and supporting our back in these trying times. Thank you!*

# Flutter Application Development

Exam Code: AFD-200

© 2020 Android ATC

Published by: Android ATC

First Printing: April 2020.

ISBN: 978-0-9900143-9-3

Information in this book, including URL and other Internet Web site references, is subject to change without notice. Complying with all applicable copyright laws, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission from Android ATC.

Android ATC is not responsible for webcasting or any other form of transmission received from any linked site.

Android ATC is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of Android ATC of the site or the products contained therein.

Android ATC may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. As expressly provided in any written license agreement from Android ATC, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Flutter application development is a detailed guide that provides the basics to build Flutter applications. It is a combination of theoretical lessons and practical labs that covers skills and knowledge every Flutter developer should learn before starting the development of real-world applications.

All lessons and their lab exercises in this book were built to comply with the latest versions of Flutter SDK and Android Studio IDE. Since the update of both Flutter SDK and Android Studio is a continuous process, it is highly possible that any of these components has already been updated by the time you start your training using this

book. If this is the case, you might notice some minor difference in the lab steps and the screenshots provided, depending on how major an update has been. Updates neither make the lessons outdated nor the labs incorrect. It is only impractical to release a new version of the book for every update.

This book is intended for trainees with background in object-oriented programming. It is expected to have such differences between the Android Studio version you are using and the one followed in the book; however, this should not constitute an obstacle for learning and following the labs.

Android ATC Training team continuously works on providing the most up to date labs and code samples. Nonetheless, we would like to apologize in advance in case any lab step or screenshot was inaccurate.

## **Warning and Disclaimer:**

This book is designed to provide information about Flutter application development course and exam AFD-200. Every effort has been made to make this book as complete and as accurate as possible.

## **Exam**

You can examine your knowledge on the content of this book by taking the online exam AFD-200 through Pearson-VUE testing centers worldwide. Passing this exam grants the examinee the title: “Flutter Certified Application Developer. For more information, visit:

<http://www.pearsonvue.com/androidatc>

Or, you may schedule your exam at any Android ATC authorized training center worldwide. Check Android ATC web site for more information.

## **Trademark Acknowledge:**

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. The use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Android is a trademark of Google Inc. The Android robot is reproduced or modified from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License.

## **Feedback Information:**

As Android ATC, our goal is to create in-depth technical books of the highest quality and value. Each book is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from professional technical community.

Readers' feedback is constituting natural continuation of this process. If you have any comments regarding how we could improve the quality of this book, or otherwise modify it to better suit your needs, you can contact us through email at: [info@androidatc.com](mailto:info@androidatc.com). Please make sure to include the book title and ISBN in your message.

We greatly appreciate your assistance.

Android ATC Team

# Table of Contents

---

## Lesson 1: Introduction to Flutter and Dart Programming Language

Introduction .....	1-2
Importance of Flutter .....	1-2
Introduction to Dart .....	1-3
Writing Dart code .....	1-3
DartPad .....	1-4
Installing Dart SDK .....	1-4
IntelliJ IDEA .....	1-10
<b>Lab 1: Installing Dart IDE and Writing Dart Program .....</b>	<b>1-12</b>
Installing IntelliJ IDEA .....	1-13
Creating a Dart Project Using IntelliJ IDEA .....	1-25
Using DartPad .....	1-29

## Lesson 2: Dart Programming - Syntax

Introduction .....	2-2
main( ) function .....	2-2
Dart Variables .....	2-4
Dart Data Types .....	2-5
Input of Information to Dart Program .....	2-15
Writing Comments .....	2-17
Dart Conditional Operators .....	2-19
If Statement .....	2-22
If – Else Statement .....	2-24
If...Else and Else...If... Statement .....	2-25
If Else and Logical Operators .....	2-26
For Loops .....	2-28
While Loops .....	2-29
Do-while Loops .....	2-31
Break Statement .....	2-32
Switch Case Statement .....	2-33
<b>Lab 2: Create a Pizza Order Program .....</b>	<b>2-36</b>

## Lesson 3: Dart Functions & Object-Oriented Programming (OOP)

<b>Functions</b> .....	3-2
Function Structure .....	3-2
Creating a Function.....	3-2
Function Return Data Types.....	3-4
Void Function .....	3-7
Function Returning Expression.....	3-9
Functions and Variable Scope.....	3-10
<b>Object-Oriented Programming (OOP)</b> .....	3-12
Object.....	3-12
Class .....	3-13
Creating a Class .....	3-13
Adding Methods to Classes.....	3-18
Providing Constructors for Your Classes.....	3-19
Class – Getters and Setters.....	3-25
Class Inheritance.....	3-27
Abstract Class .....	3-28
<b>Dart Project Structure and Dart Libraries</b> .....	3-33
<b>Lab 3: Create a Small Overtime Payment Program</b> .....	3-40

## Lesson 4: Introduction to Flutter

<b>Understanding Flutter</b> .....	4-2
<b>Flutter Framework</b> .....	4-4
<b>Android Studio</b> .....	4-5
What is Android Studio? .....	4-5
Android Studio Software Prerequisite.....	4-5
Installing Android Studio .....	4-8
<b>Flutter SDK</b> .....	4-14
<b>Installing and Configuring Flutter SDK</b> .....	4-14
<b>Creating a New Flutter Project</b> .....	4-20
<b>Setup an Android Virtual Device</b> .....	4-24
<b>Run a Flutter App</b> .....	4-30
<b>Installing Flutter on Mac</b> .....	4-36
<b>Test Your Flutter App on iOS Phone with Windows O.S</b> .....	4-37
<b>Android Studio Sugar and Spice</b> .....	4-45
<b>Run your Apps on a Hardware Device (Physical Phone)</b> .....	4-51
Run your Flutter App on Android Phone .....	4-52

Run your Flutter App on iPhone Device .....	4-56
<b>Emulator Debug Mode .....</b>	<b>4-56</b>
<b>Introduction to Flutter Widgets .....</b>	<b>4-57</b>
<b>Creating a Flutter App Using Widgets .....</b>	<b>4-59</b>
<b>What is a MaterialApp widget?.....</b>	<b>4-64</b>
<b>Lab 4: Creating a Simple Flutter App .....</b>	<b>4-66</b>

## Lesson 5: Flutter Widgets Fundamentals

<b>Scaffold Widget.....</b>	<b>5-2</b>
<b>Image Widget .....</b>	<b>5-8</b>
<b>Container Widget .....</b>	<b>5-15</b>
<b>Column and Row Widgets .....</b>	<b>5-24</b>
<b>Icon Widget.....</b>	<b>5-29</b>
<b>Layouts in Flutter .....</b>	<b>5-31</b>
<b>Card Widget.....</b>	<b>5-42</b>
<b>App Icons for iOS and Android Apps.....</b>	<b>5-46</b>
<b>Hot Reload and Hot Restart .....</b>	<b>5-50</b>
<b>Stateful and Stateless Widgets.....</b>	<b>5-55</b>
<b>Use a Custom Font.....</b>	<b>5-64</b>
<b>Lab: Creating a Restaurant Menu .....</b>	<b>5-69</b>

## Lesson 6: Navigation and Routing

<b>Button Widget .....</b>	<b>6-2</b>
FloatingActionButton.....	6-2
RaisedButton, FlatButton, and IconButton.....	6-6
DropdownButton .....	6-8
OutlineButton .....	6-13
ackBar.....	6-14
PopupMenuButton .....	6-17
<b>App Structure and Navigation .....</b>	<b>6-21</b>
<b>Navigate to a New Screen and Back .....</b>	<b>6-21</b>
<b>Navigate with Named Routes.....</b>	<b>6-29</b>
<b>Send and Return Data Among Screens .....</b>	<b>6-34</b>
<b>Animate a Widget Across Screens .....</b>	<b>6-36</b>
<b>WebView Widget in Flutter .....</b>	<b>6-40</b>
<b>Lab 6: Navigation and Routing a Pizza Store App .....</b>	<b>6-45</b>

## Lesson 7: Visual, Behavioral, and Motion-Rich Widgets Implementing Material Design Guidelines - Part 1

Introduction .....	7-2
BottomNavigationBar Widget .....	7-2
DefaultTabController, TabBar, and TabBarView Widgets .....	7-5
ListTile Widget .....	7-11
ListView Widget .....	7-15
Drawer Widget .....	7-18
DataTable Widget .....	7-29
SelectableText Widget .....	7-32
Stack Widget .....	7-35
<b>Lab : 7 .....</b>	<b>7-39</b>
Lab A: Creating a Flutter App using BottomNavigationBar Navigation Technique.....	7-40
Lab B: Using DataTable Sorting Built-in function.....	7-45

## Lesson 8: Visual, Behavioral, and Motion-Rich Widgets Implementing Material Design Guidelines - Part 2

### **Input and Selections**

Text Field Widget.....	8-2
CheckboxGroup and RadioButtonGroup Widgets .....	8-11
Date Picker.....	8-17
Time Picker. ....	8-23
Slider Widget.....	8-25
Switch Widget.....	8-29

### **Dialogs, Alerts, and Panels**

Alert Dialog Widget.....	8-32
Cupertino Alert Dialog Widget.....	8-35
Bottom Sheet.....	8-36
Modal Bottom Sheet.....	8-36
Persistent Bottom Sheet.....	8-41
Expansion Panel Widget .....	8-49
SnackBar Widget.....	8-54
<b>Lab 8: Creating a Hotel Reservation App.....</b>	<b>8-60</b>

## Lesson 9: Firebase

<b>Introduction .....</b>	9-2
<b>What is the JSON ? .....</b>	9-3
<b>How does Firebase Database work? .....</b>	9-4
<b>Firebase authentication (Signup and Login to Flutter App) .....</b>	9-5
<b>Configure Your App to use Firebase Services .....</b>	9-17
Adding Firebase to your Android App .....	9-19
Adding Firebase to your iOS App .....	9-26
<b>Configuring Firebase Authentication .....</b>	9-33
Login to an App Using Firebase User Accounts.....	9-46
Logout Configuration .....	9-48
<b>Firebase Database .....</b>	9-53
Which database is right for your project? .....	9-53
Real Time Database .....	9-54
Cloud Firestore .....	9-63
<b>Lab 9 : Create a User Profile Interface using Firebase .....</b>	9-72

## Lesson 10: Location-Aware Apps: Using GPS and Google Maps

<b>Introduction .....</b>	10-2
<b>What is GPS and how does it work? .....</b>	10-2
<b>The Camera Position.....</b>	10-4
<b>Adding Google Maps to a Flutter app .....</b>	10-5
Getting a Google API key .....	10-6
Adding Google Maps Flutter plug-in as a dependency .....	10-11
Adding your API key for your Android app .....	10-12
Adding your API key for your iOS app .....	10-13
Adding a Google Map on Your Flutter App Screen.....	10-14
Adding a Google Map Marker .....	10-18
Google Map Types.....	10-21
Moving the Camera (Camera Animation) .....	10-23
Capturing an App User's Location for iOS and Android Apps.....	10-26
<b>Lab10: Location-Aware Apps Using GPS and Google Maps .....</b>	10-27
Getting a Google API key .....	10-28
Creating an App Interface .....	10-33
Configuring your App to Use Your API Key .....	10-34
Adding a Google Map on your Flutter App Screen .....	10-36
Adding a Google Map Marker .....	10-38

Capturing Users' Location .....	10-39
Configuring User App's Permission.....	10-41

## Lesson 11: App Testing & Publishing

<b>Testing and Feedback for Your App .....</b>	<b>11-2</b>
Setting up a Test Environment .....	11-6
Usability Testing by Participants .....	11-7
Starting your Test Session .....	11-8
Analyzing your Test.....	11-10
<b>Publishing Flutter Apps .....</b>	<b>11-10</b>
Publishing Android App on Google Play Store.....	11-17
Publishing iOS app on Apple Store.....	11-32

# Lesson 1: Introduction to Flutter and Dart Programming Language

<b>Introduction .....</b>	1-2
<b>Importance of Flutter .....</b>	1-2
<b>Introduction to Dart.....</b>	1-3
<b>Writing Dart code .....</b>	1-3
<b>DartPad .....</b>	1-4
<b>Installing Dart SDK.....</b>	1-4
<b>IntelliJ IDEA.....</b>	1-10
<b>Lab 1: Installing Dart IDE and Writing Dart Program .....</b>	1-12
<b>Installing IntelliJ IDEA.....</b>	1-13
<b>Creating a Dart Project Using IntelliJ IDEA.....</b>	1-25
<b>Using DartPad .....</b>	1-29

## Introduction

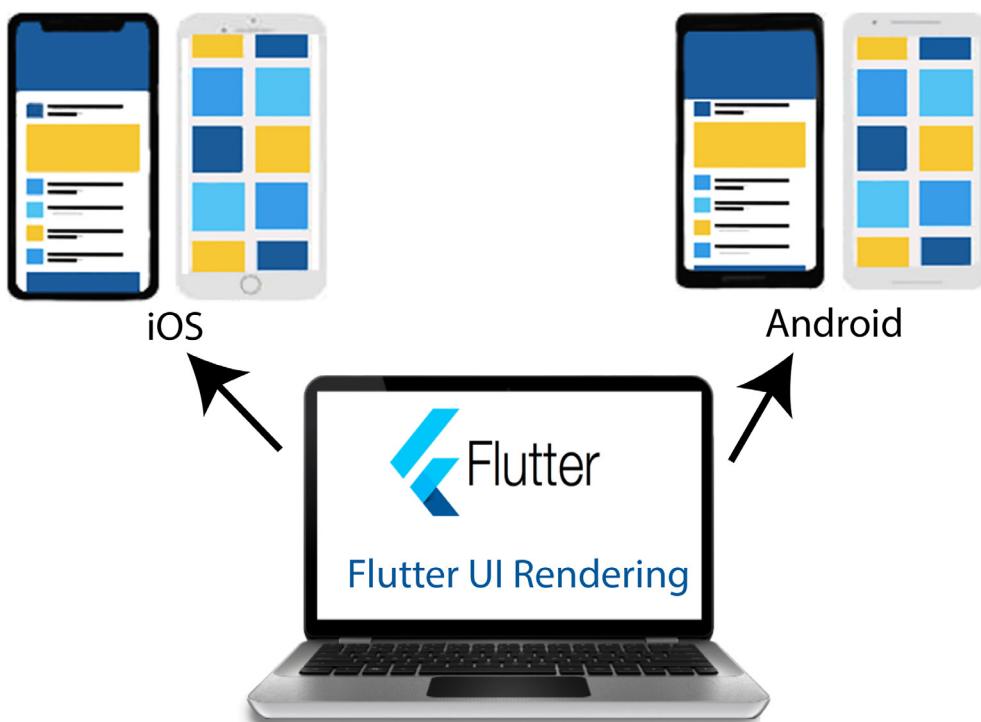
Flutter is an open-source user interface software development kit (app SDK) created by Google. It is for building high-performance, high-fidelity apps for iOS, Android, and web from a single codebase.

The purpose of this course is to enable developers create high-performance and attractive apps that feel natural on iOS & Android devices.

Flutter, is used by companies around the world including Alibaba, Capital One, and Groupon for apps that touch hundreds of millions of users.

## Importance of Flutter

Because any developer or anyone who wants to learn about mobile development can now build native Android and iOS apps with one codebase ONLY! This means, instead of having to learn Objective-C or Swift to build iOS apps, and Java or Kotlin to build Android apps, you can now use Flutter Mobile Development Framework to build apps that run natively on both iOS and Android devices using the *Dart Programming Language*.



This course is designed for you to start learning the Dart Programming Language (which is used by Flutter SDK to build native iOS and Android Apps), Flutter

Framework, and create cross-platform mobile apps.

The following are some advantages of Flutter :

1-Be highly productive:

- o Develop for iOS and Android from a single codebase.
- o Do more with less code, even on a single OS, with a modern, expressive language, and a declarative approach.
- o Prototype and iterate easily where you can change your code and reload it as your app runs (hot reload feature) as you will see in the next lessons. Also, Flutter fixes crashes and continue debugging from where the app is left off.

2- Create beautiful, highly-customized user experiences (UI):

- o Benefit from a rich set of Material Design and Cupertino (iOS-flavor) widgets built using Flutter's own framework.
- o Realize custom, beautiful, brand-driven designs, without the limitations of OEM widget sets.

## Introduction to Dart

Dart is an object-oriented programming language developed by Google. It is an open-source, scalable programming language, with robust libraries and runtimes, for building web, server, and mobile apps.

## Writing Dart Code

To write a Dart program you need two things: First, a graphical user interface software which helps you write, save, edit, and run the Dart code. This software is called IDE (*Integrated Development Environment*) such as Android Studio IDE, or IntelliJ IDE. Second, Dart SDK, since the IDE is a graphical user interface software used to write the code. You need a software to translate these Dart commands which are written in Dart IDE (such as Android Studio) to a lower level language to create an executable program. This software is called Dart SDK (*Software Development Kit* ).

In the following topics, you will know more about the **Dart SDK** and **Dart IDE** :

### 1- Dart SDK

The Dart SDK has the libraries and command-line tools that you need to develop Dart web, command-line, mobile, and server apps.

The Dart SDK has the libraries and command-line tools that you need to write and run Dart code. The Dart SDK includes a **lib** directory for the Dart libraries and a **bin** directory that has the command-line tools. In the next topics of this lesson, you will know more about installing and configuring Dart SDK.

## 2- Dart IDE

We use Dart IDEs (integrated Development Environment) to create a Dart program , where these IDEs include Dart plugins which are used to make a connection between the IDE graphical user interface software and the Dart SDK.

The following are some examples of Dart IDEs which you can use to write Dart code:

- IntelliJ IDEA
- Android Studio
- Visual Studio

## DartPad

DartPad is an open-source tool that lets you work with the Dart language in any modern web browser. It is a great, no-download-required way to learn Dart syntaxes and to experiment with Dart language features.

To test DartPad, go to : <https://dartpad.dev> , then you can test all Dart syntaxes.

In lab 1 at the end of this lesson, you will test DartPad web site and check how you can write Dart commands using your web browser without the need to install any software on your computer and whatever operating system your computer has.



When you later use Flutter SDK to create a mobile app, you don't need to install Dart SDK or Dart plugins to any IDE such as Android Studio or IntelliJ IDEA because Flutter SDK already includes Dart SDK. Also, Flutter plugins already include Dart plugins

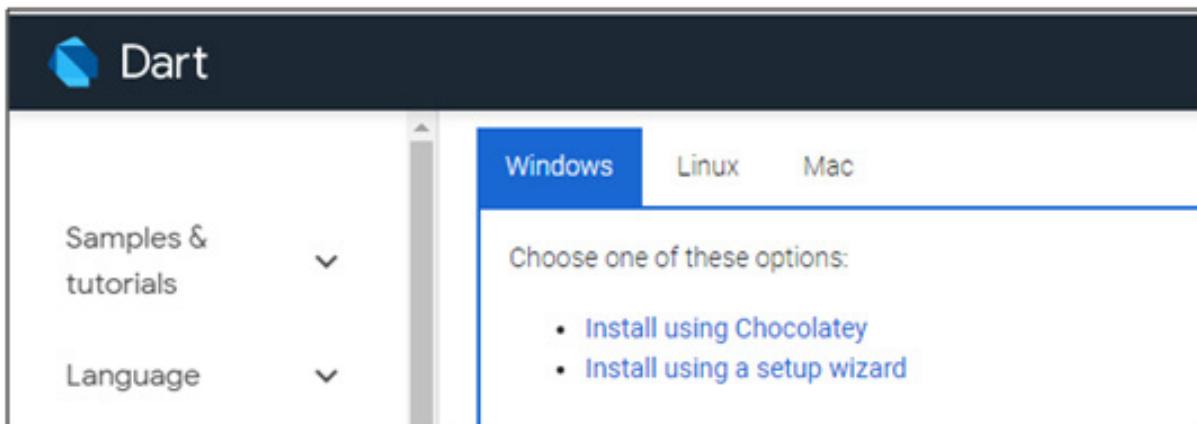
## Installing Dart SDK

To install Dart SDK, perform the following steps:

1- Go to : <https://dart.dev>, then click “**Get Dart**” tab, or go directly to :

<https://dart.dev/get-dart>.

You will get the following web page:

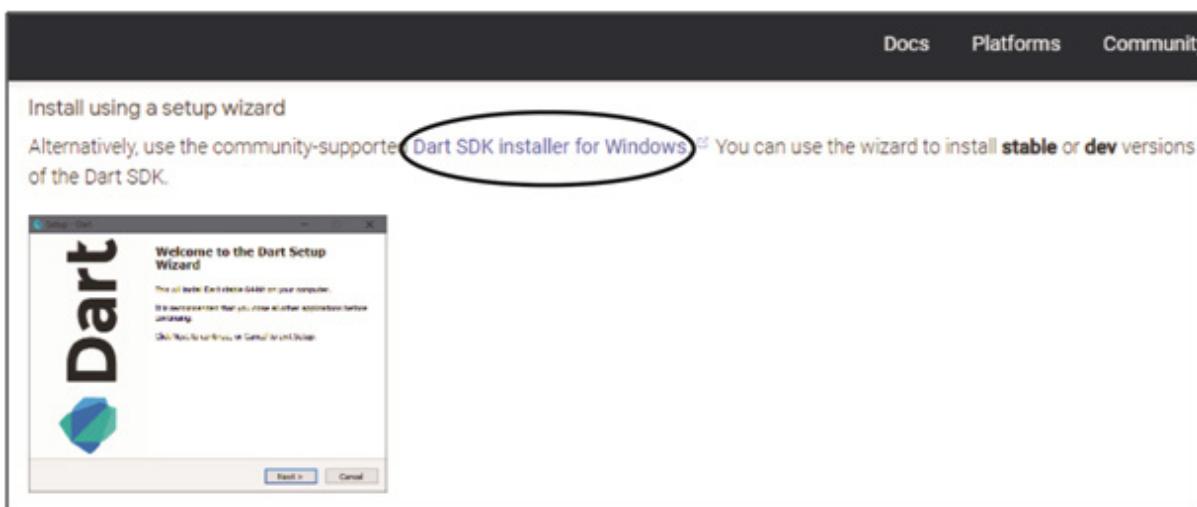


In this exercise, you will install Dart SDK using Microsoft Windows operating system. For other operating systems , you will almost have the same installation wizard.

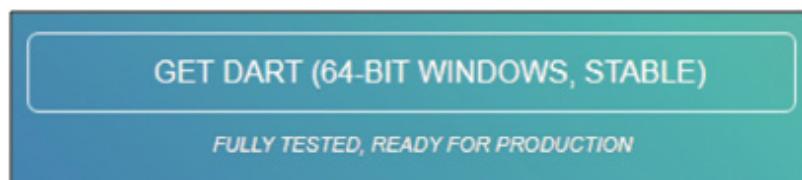
2- Select the operating system which your computer has, Windows, Linux or Mac.

3- Click "**Install using a setup wizard**"

4- Scroll down, and as illustrated in the below figure, click : **Dart SDK installer for Windows** .



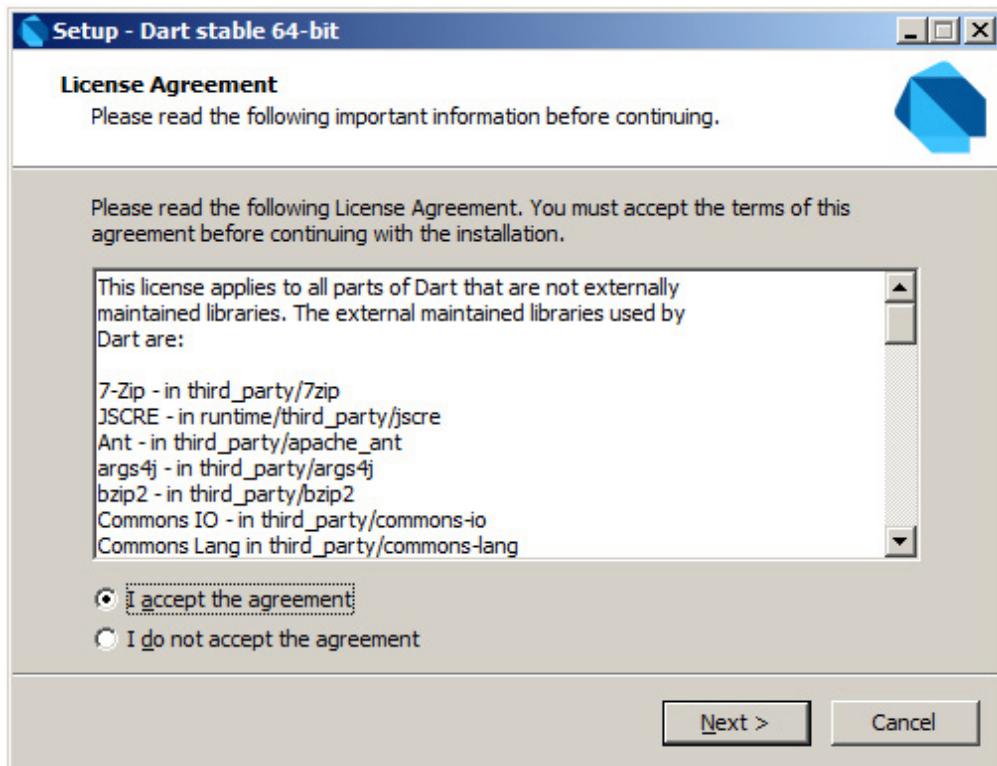
5- As illustrated in the image below, if you have Microsoft Windows operating system 64 bits, you should select **GET DART (64-BIT WINDOWS, STABLE)**.



6- Then, you will get the following file.

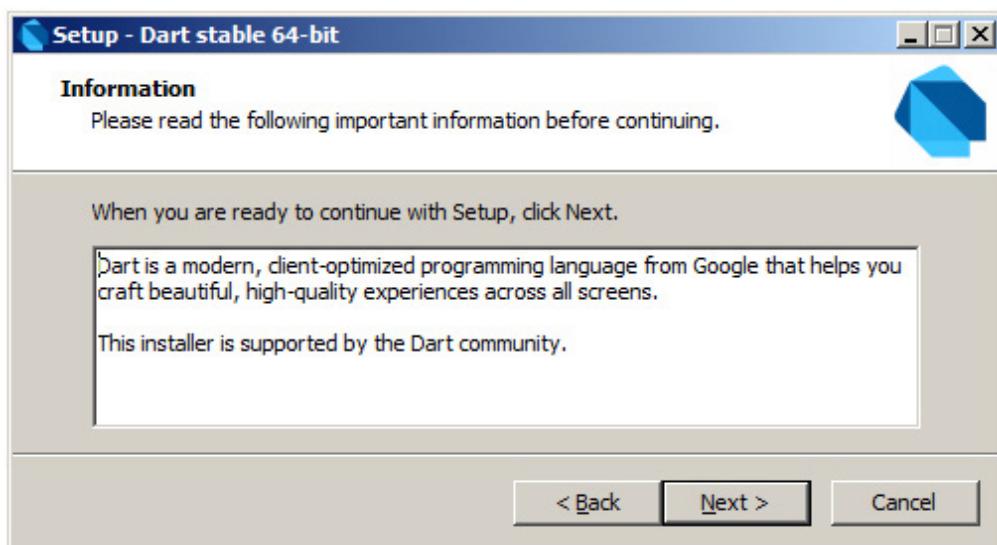


7- Click this file, select Run, and you will get the following installation wizard:

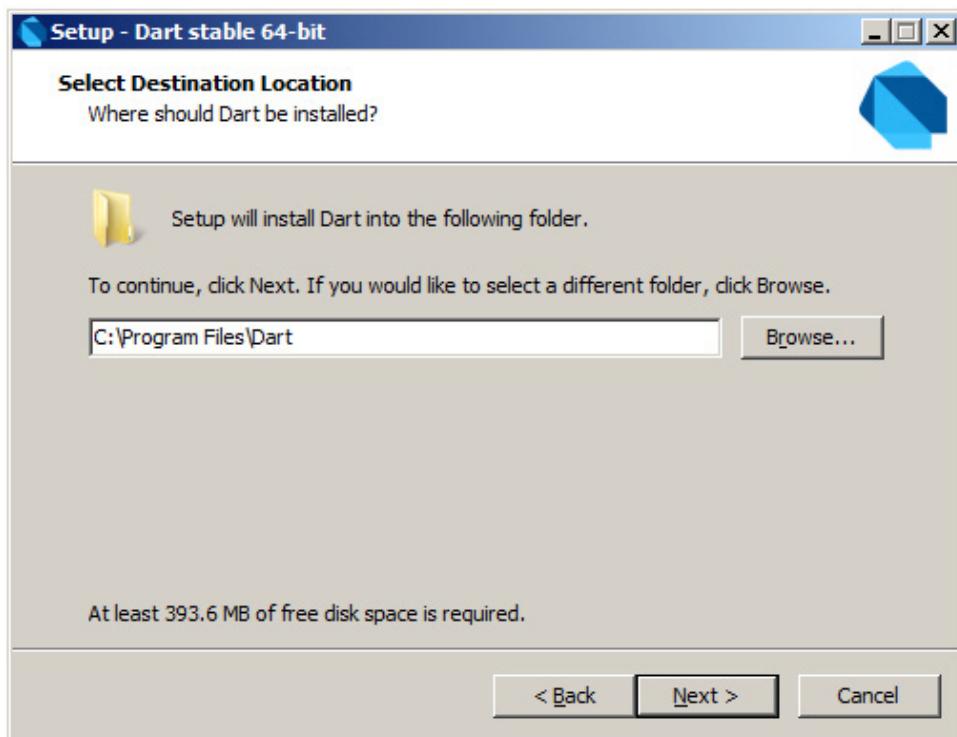


8- Select “I accept the agreement” then, click Next

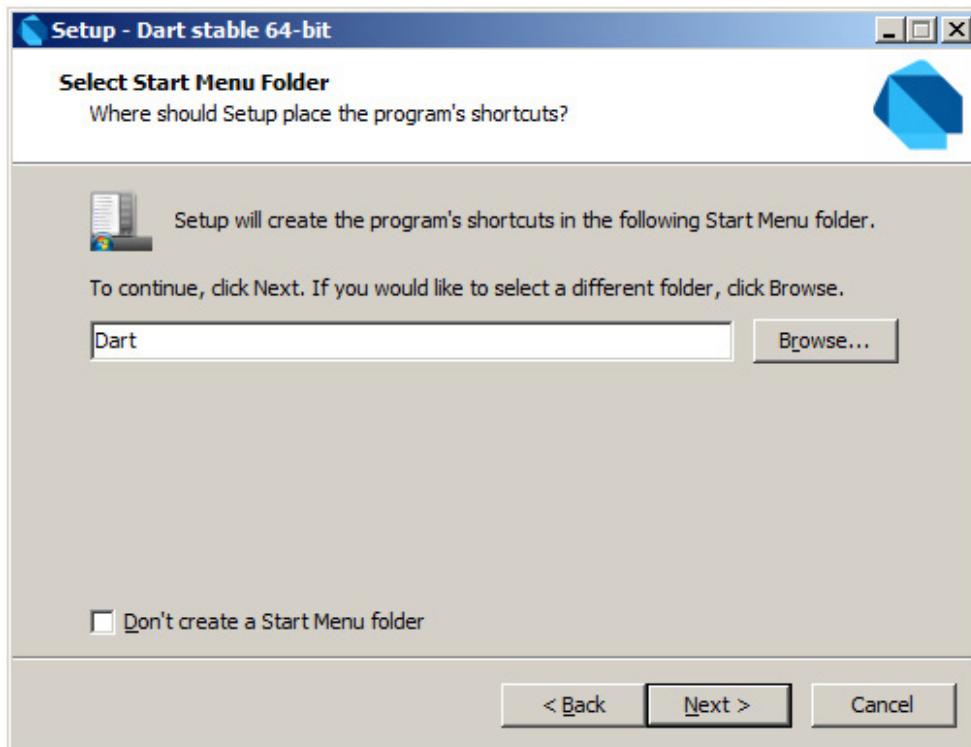
9- In the following figure, click Next



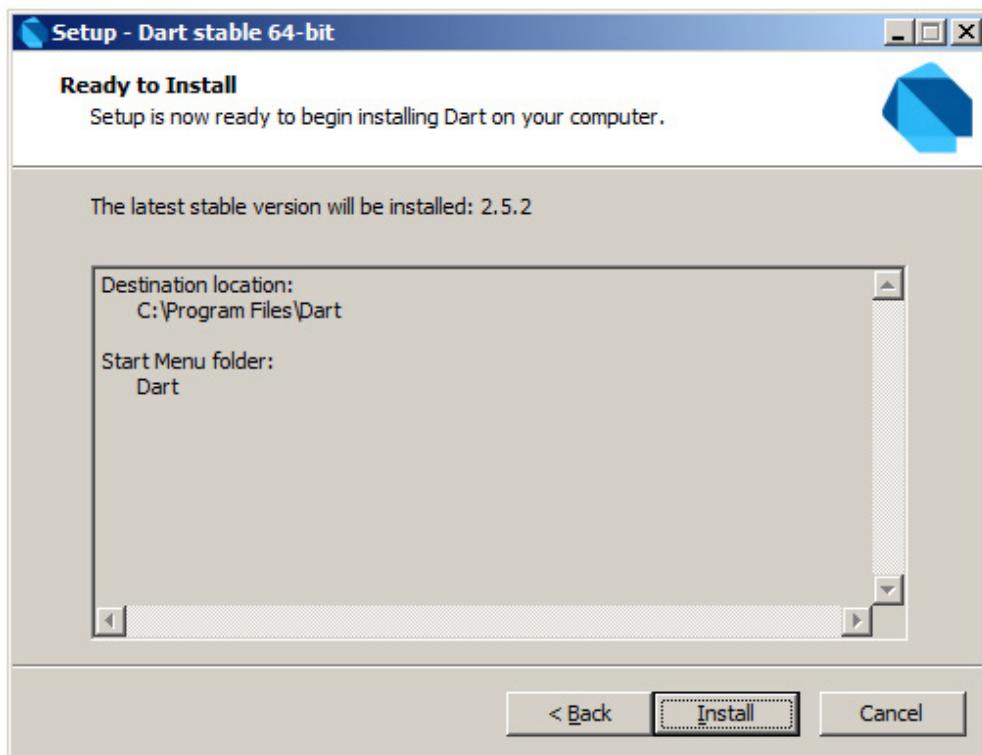
10- Keep the default path as illustrated in the below image, then click **Next**



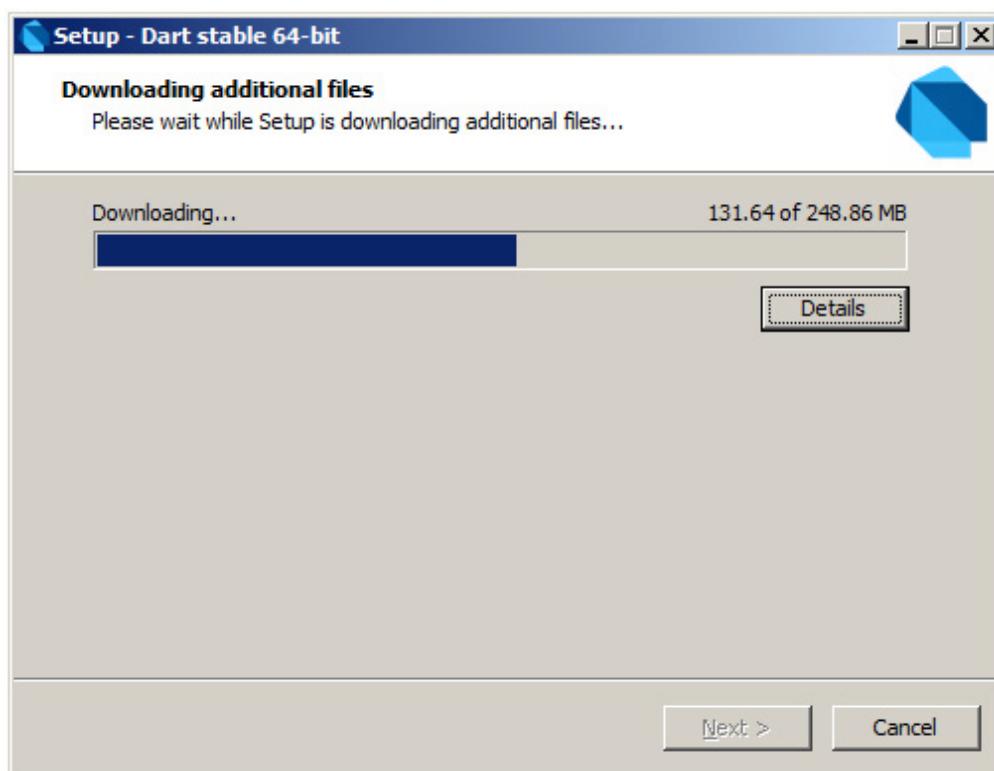
11- Keep the default shortcut location, then click **Next**



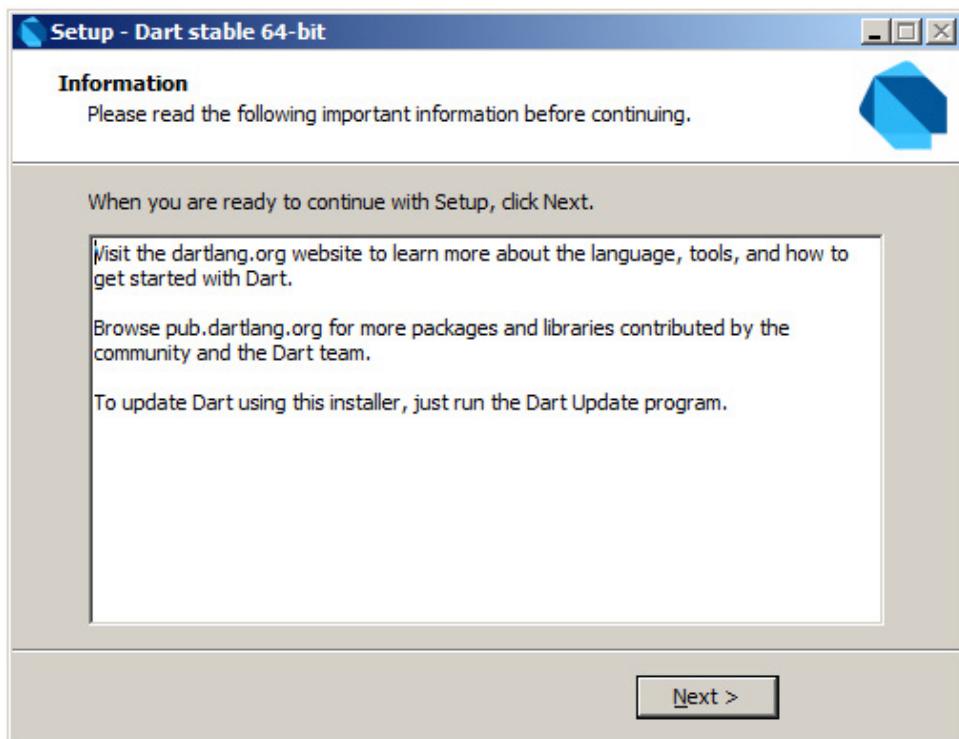
12- Click **Install** button in the following figure:



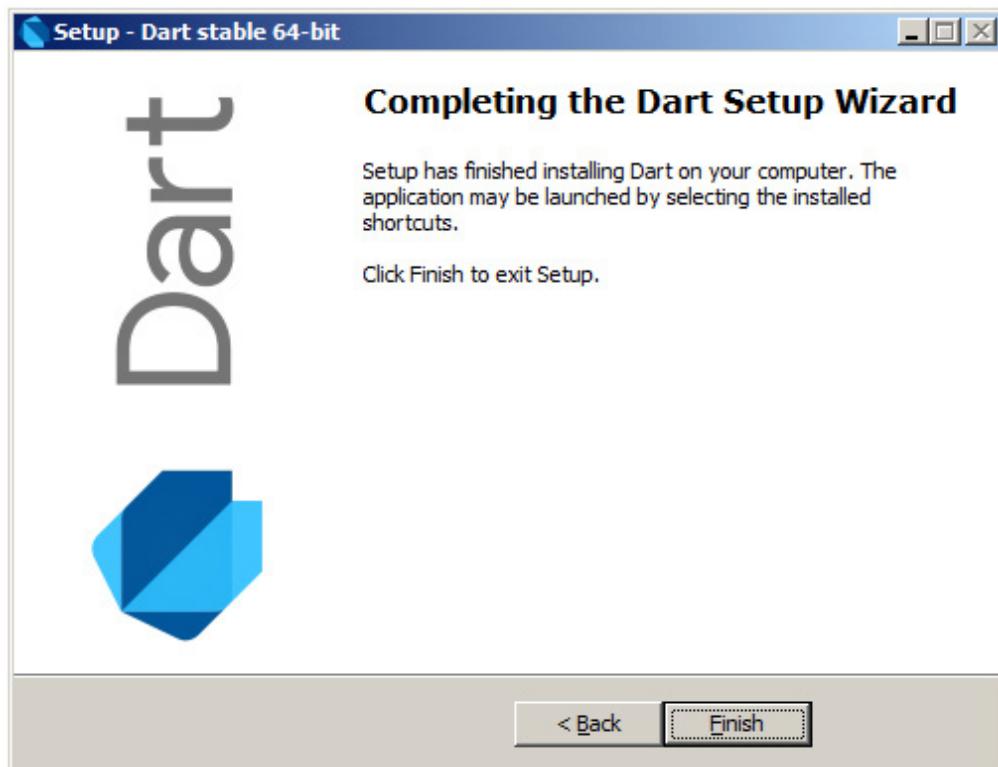
Then, you will get the following :



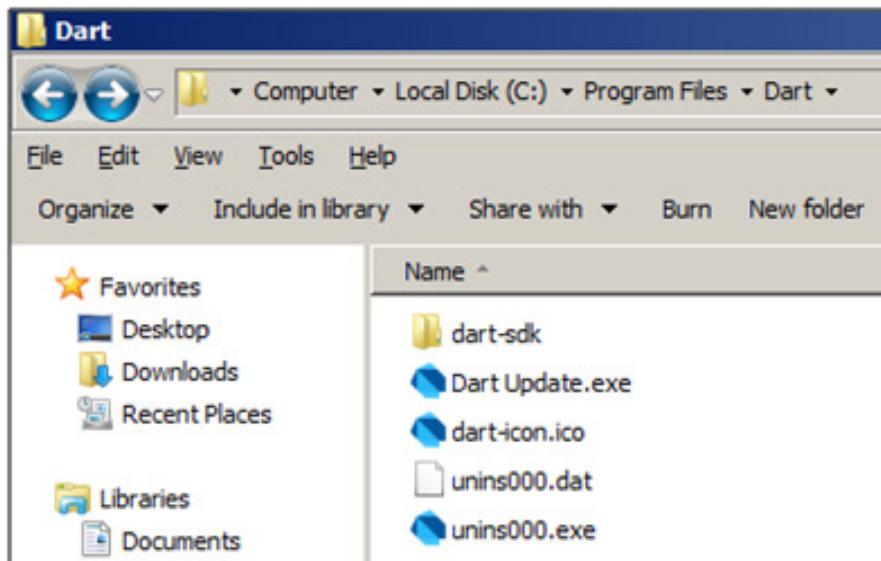
13- Click **Next** in the following figure:



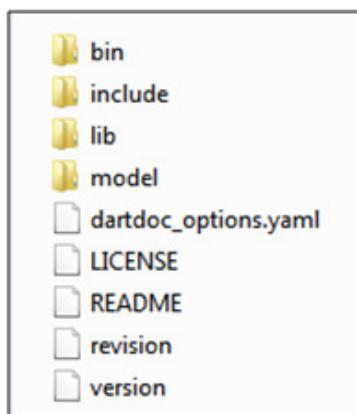
14- Click **Finish** as illustrated in the following figure:



Dart SDK has been downloaded at C:\Program Files\ Dart as illustrated in the following figure :



The following figure includes the content of the dart-sdk folder :



## IntelliJ IDEA

IntelliJ IDEA is a Java integrated development environment for developing computer software. It is developed by JetBrains,

IntelliJ IDEA is free software used to develop Java, Kotlin, Dart, and other programming languages.

The first three lessons of this course discuss the fundamentals of Dart programming language. In these three lessons, you will learn how to write small separate Dart programs to become familiar with the Dart syntaxes - which you will use later in writing the code to create Flutter mobile applications for Android and iOS devices.

We recommend using IntelliJ IDEA as a compiler software to create and run pure Dart programs.

In the lab of this lesson, you will install IntelliJ IDEA step by step and use it to create and run a Dart program.

# Lab 1

## Installing Dart IDE and Writing a Dart Program

- **Installing IntelliJ IDEA**
- **Creating a Dart Program Using IntelliJ IDEA**
- **Using DartPad**

## ➤ Installing IntelliJ IDEA

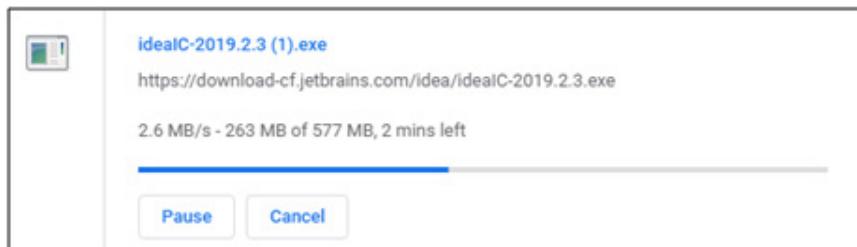
Follow the following steps to download IntelliJ IDEA:

1- Go to: <https://www.jetbrains.com/idea/download>

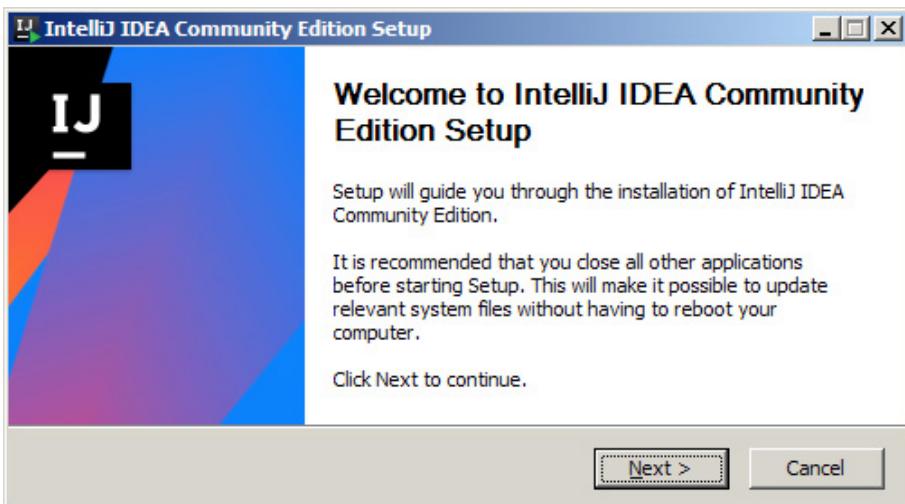
You will get the following download web page:



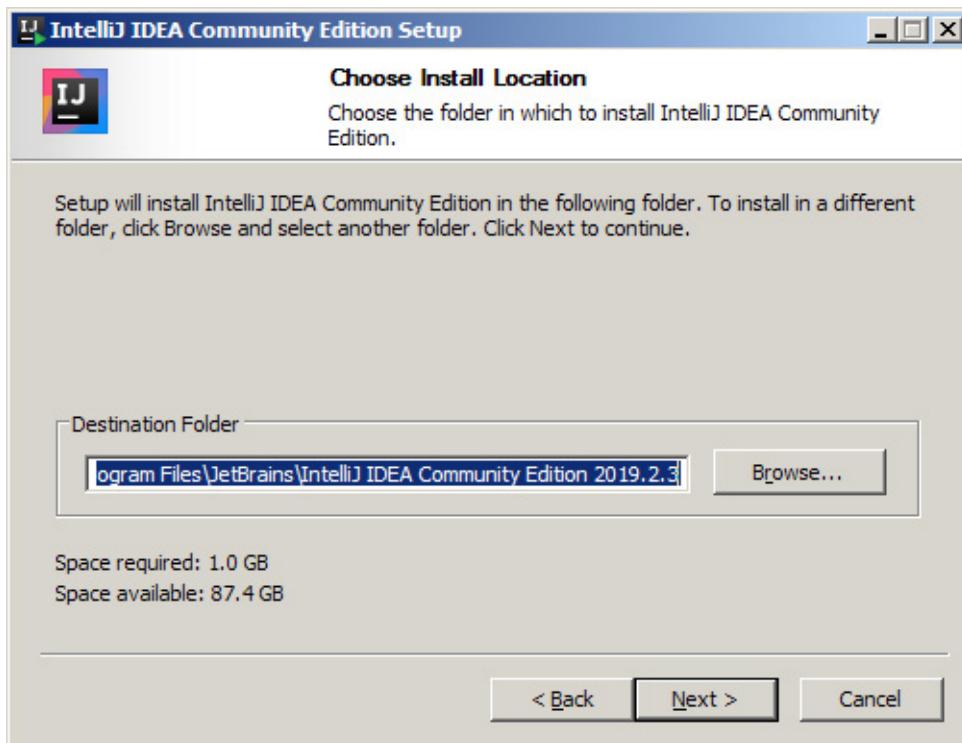
2- Under **Community**, Click **DOWNLOAD** button. The download process will start to download IntelliJ IDEA as illustrated in the following figure. The size of this file is about 577 MB.



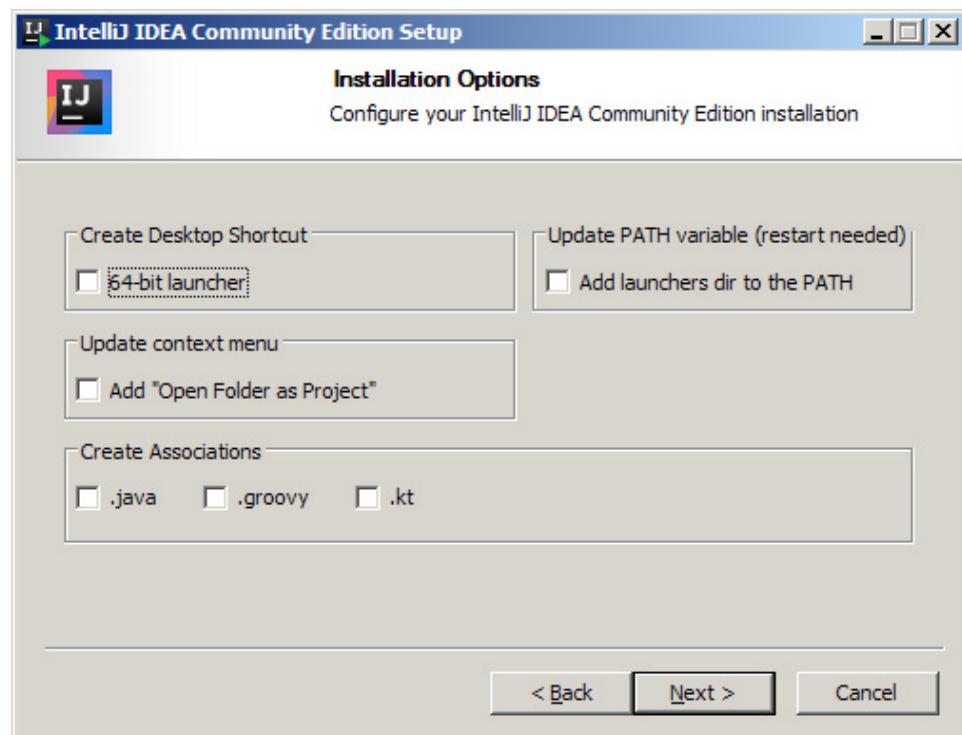
3- In the below figure, click **Next**



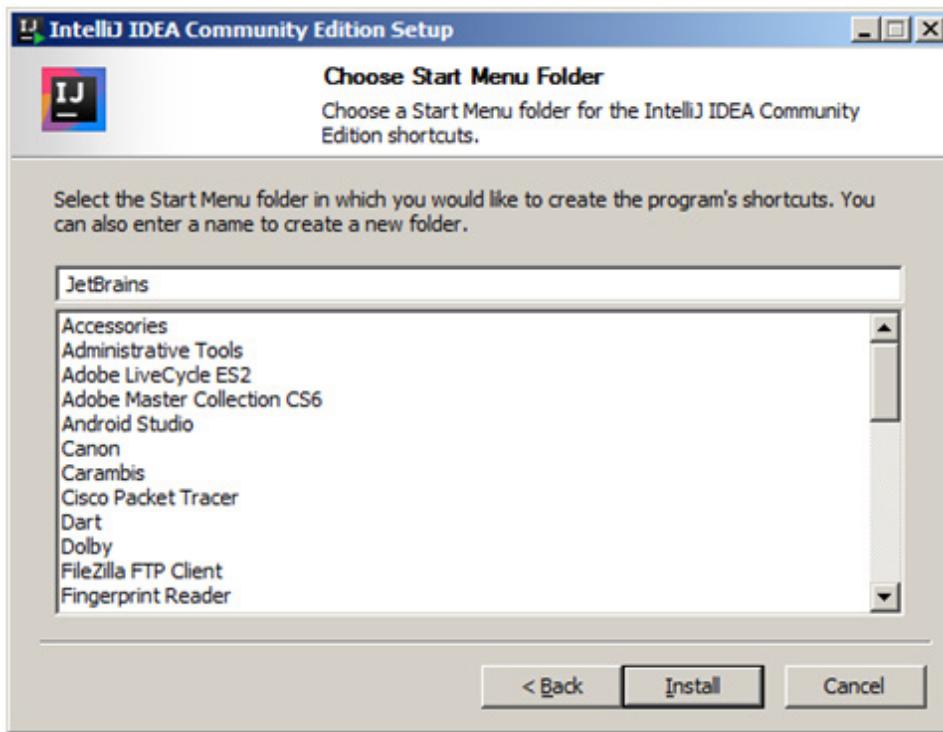
4- Keep the default destination folder, then click **Next**



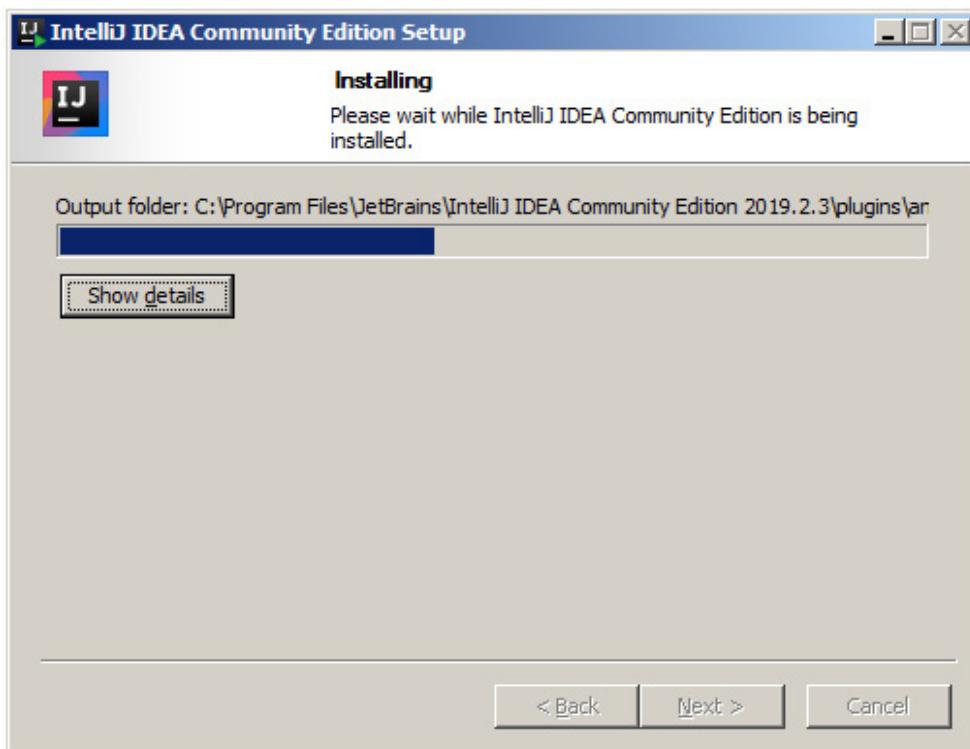
5- Our plan is to use IntelliJ to develop Dart ; therefore , no need to select any of the below choices. Click **Next**



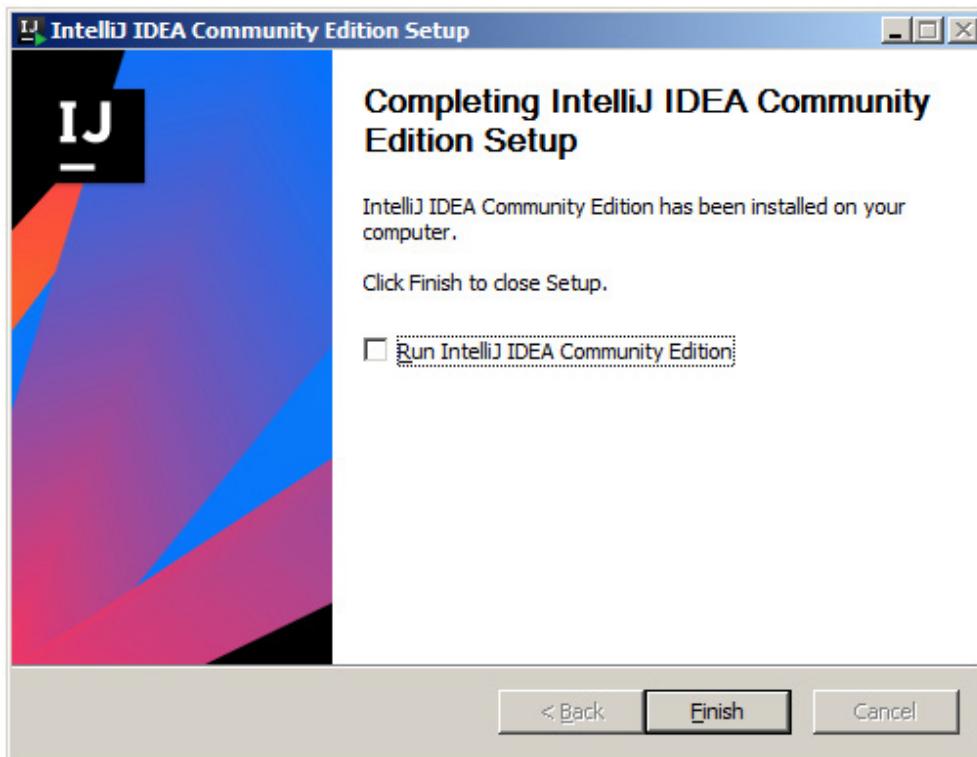
6- Keep the default start menu folder as illustrated in the below figure. Click **Install**



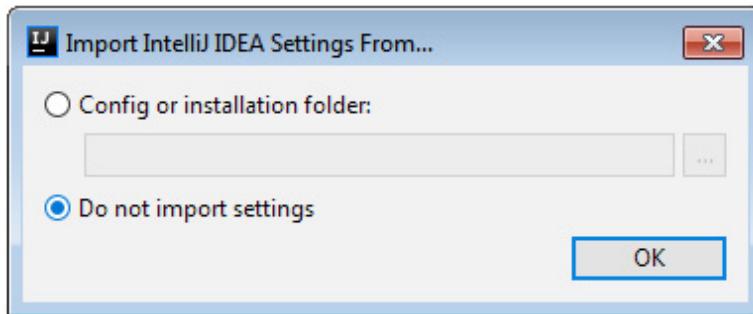
The installation process will start as illustrated in the below figure:



7- You will get the following figure. Click **Finish**



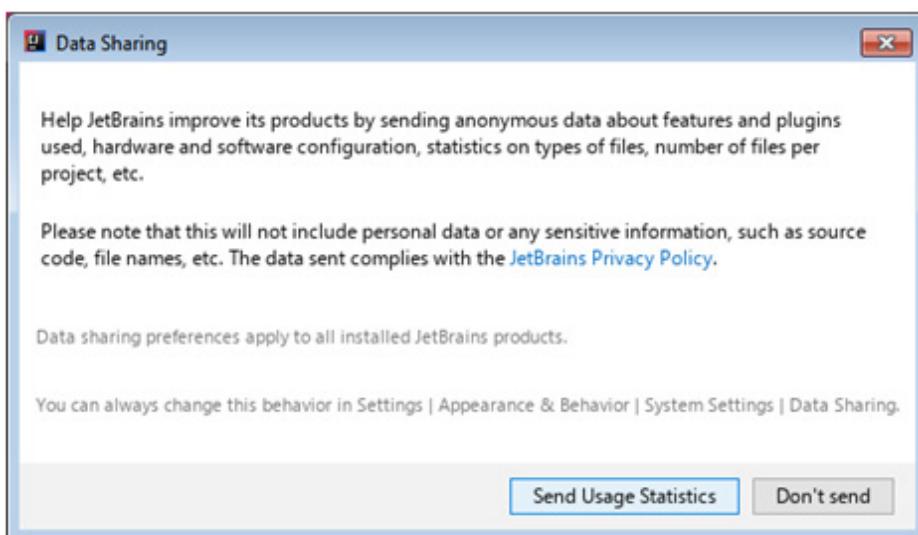
8- As illustrated in the below image, select **Do not import settings** because it is a new installation ; therefore , there are no previous settings to import , then click **OK**



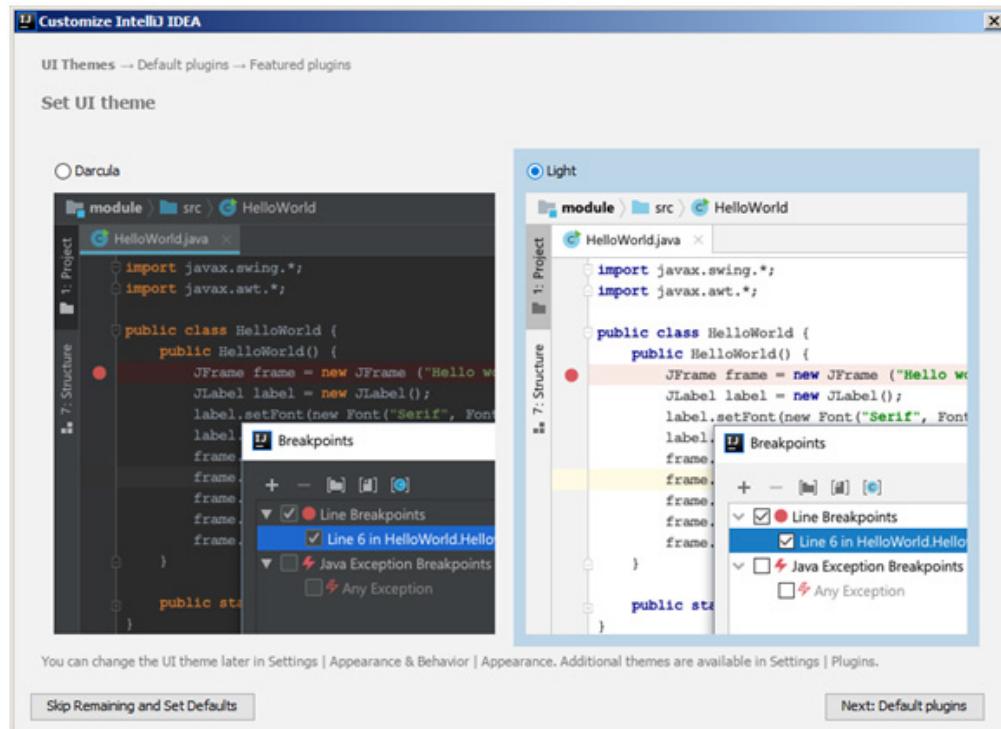
9- In the this step, select **I confirm that I read and accept the terms of this User Agreement**, then click **Continue**.



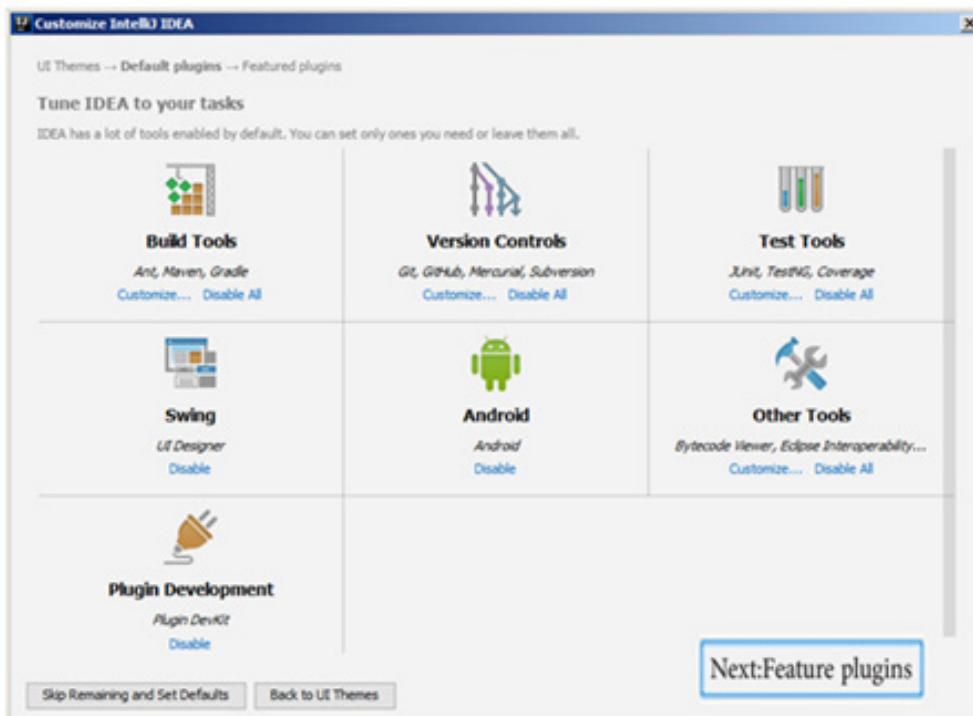
10- In this step, as illustrated in the below figure , click Don ' t send



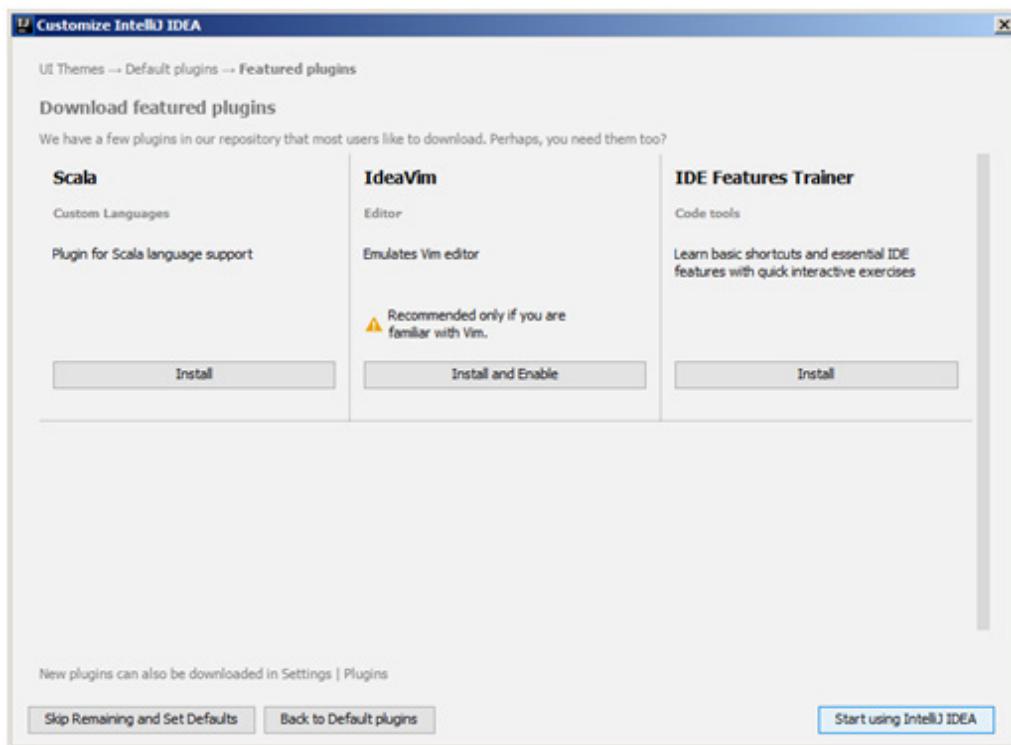
11- In this step, you may select your IntelliJ IDEA user interface theme. Almost most developers select **Darcula** theme because it is more comfortable for eyes; however, we will select **Light** theme for printing considerations. Select **Light** or **Darcula**, then click **Next Default plugins**



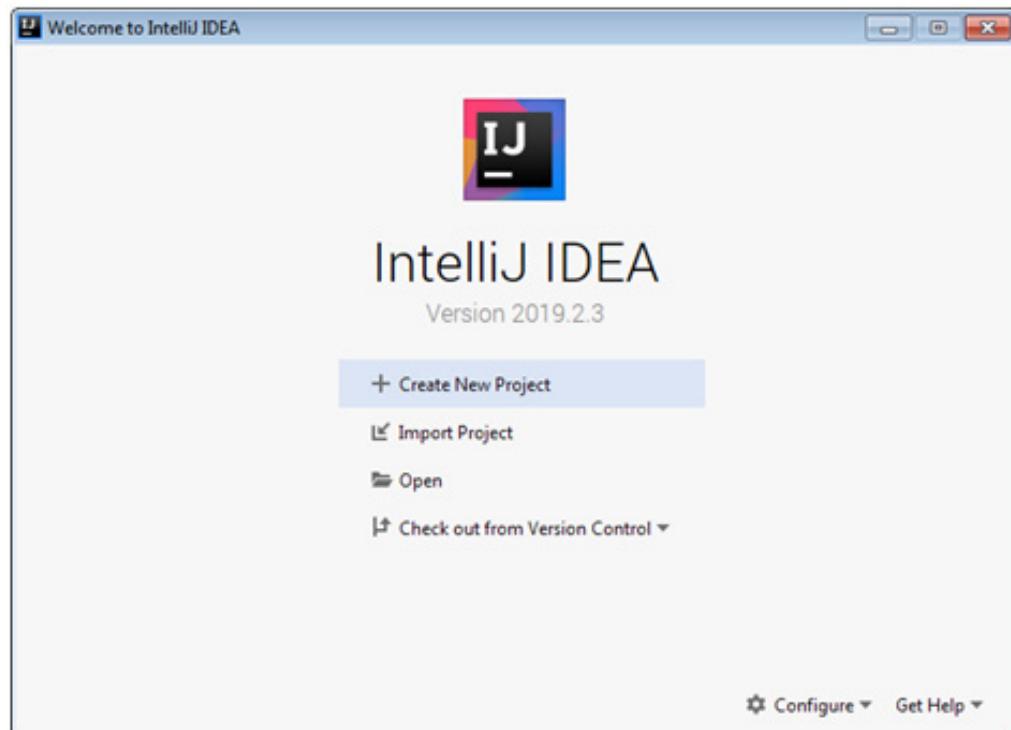
12- In this step, click **Next: Feature plugins**



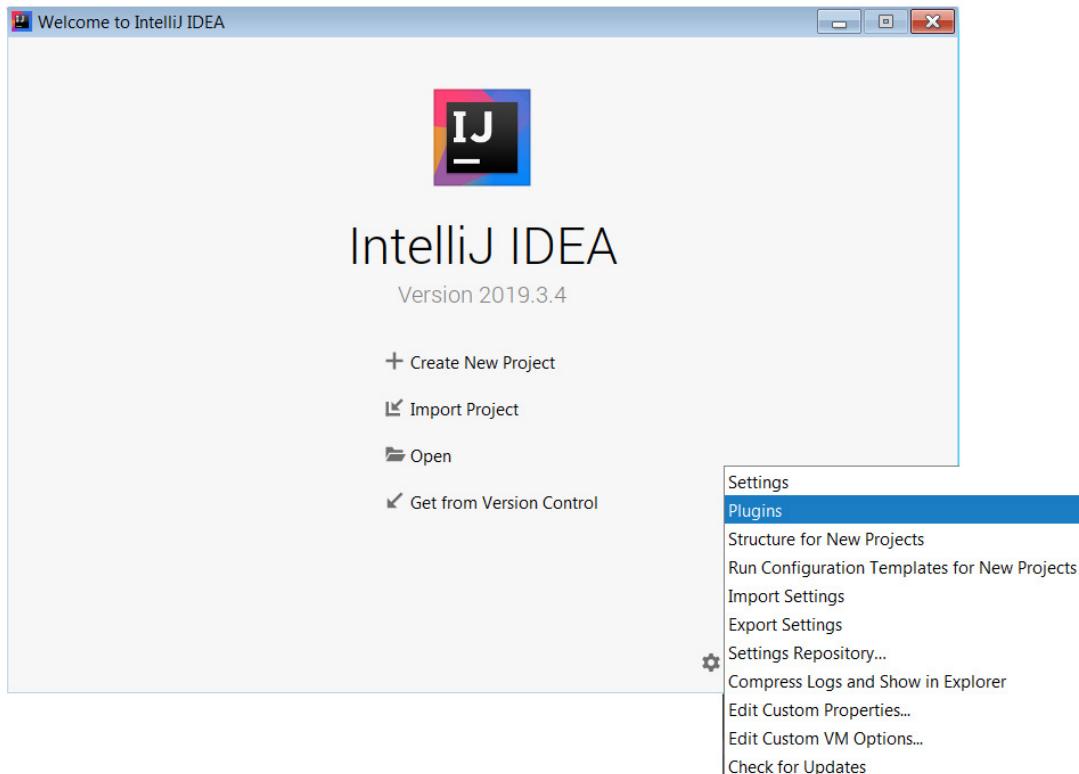
## 13- Click Start using IntelliJ IDEA



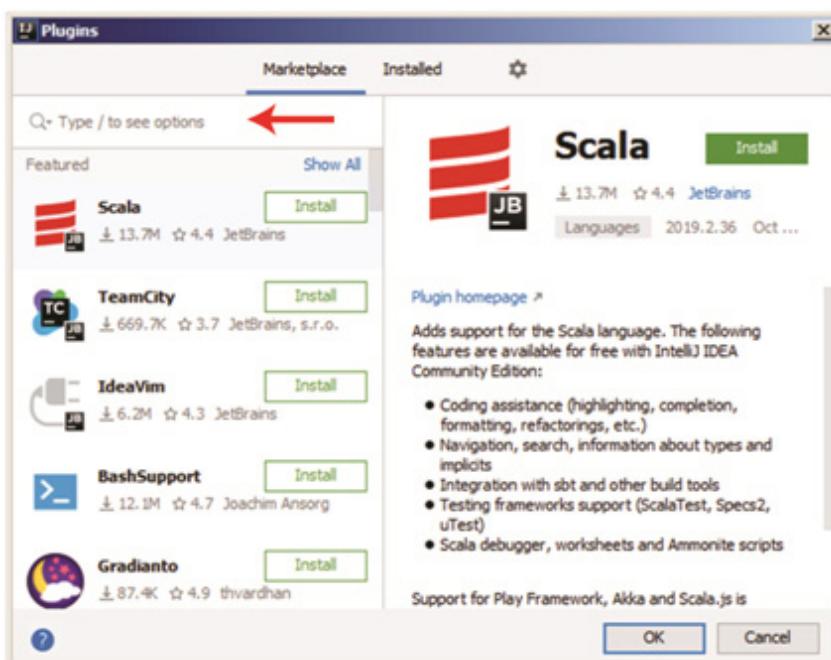
14- You will get the following IntelliJ IDEA startup dialog box.



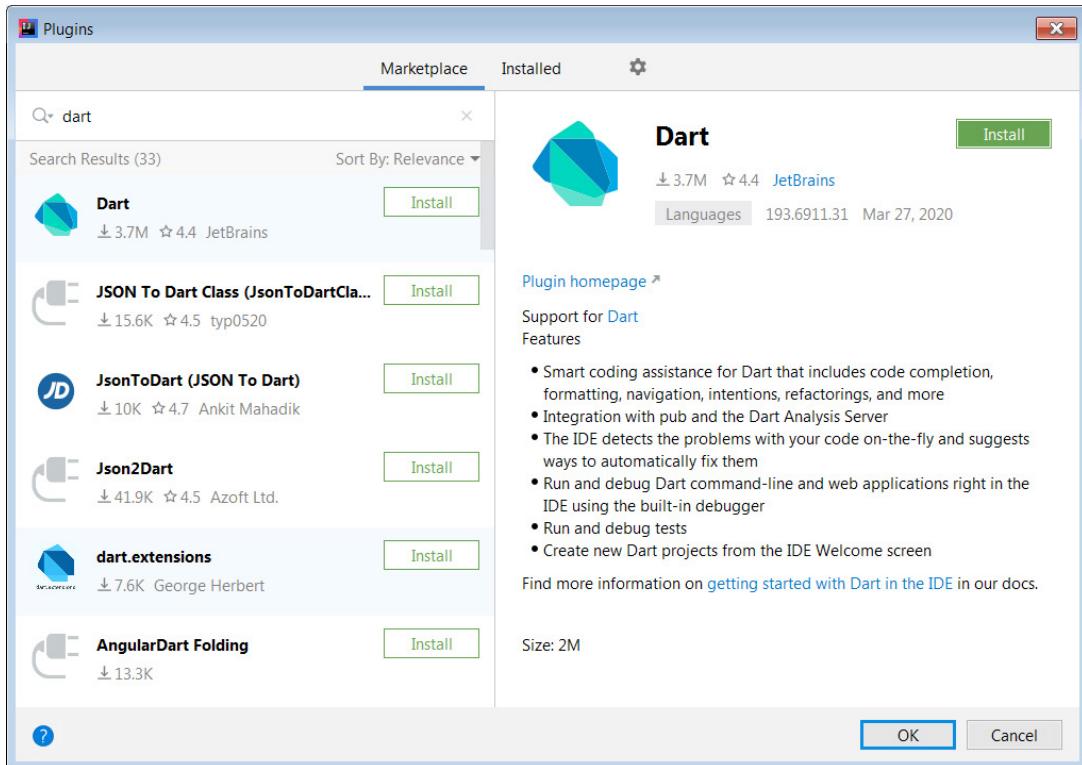
15- Now, you will install Dart plugins which will make the connection between IntelliJ IDEA and Dart SDK. Click **Configure**, then select **Plugins** as illustrated in the below figure:



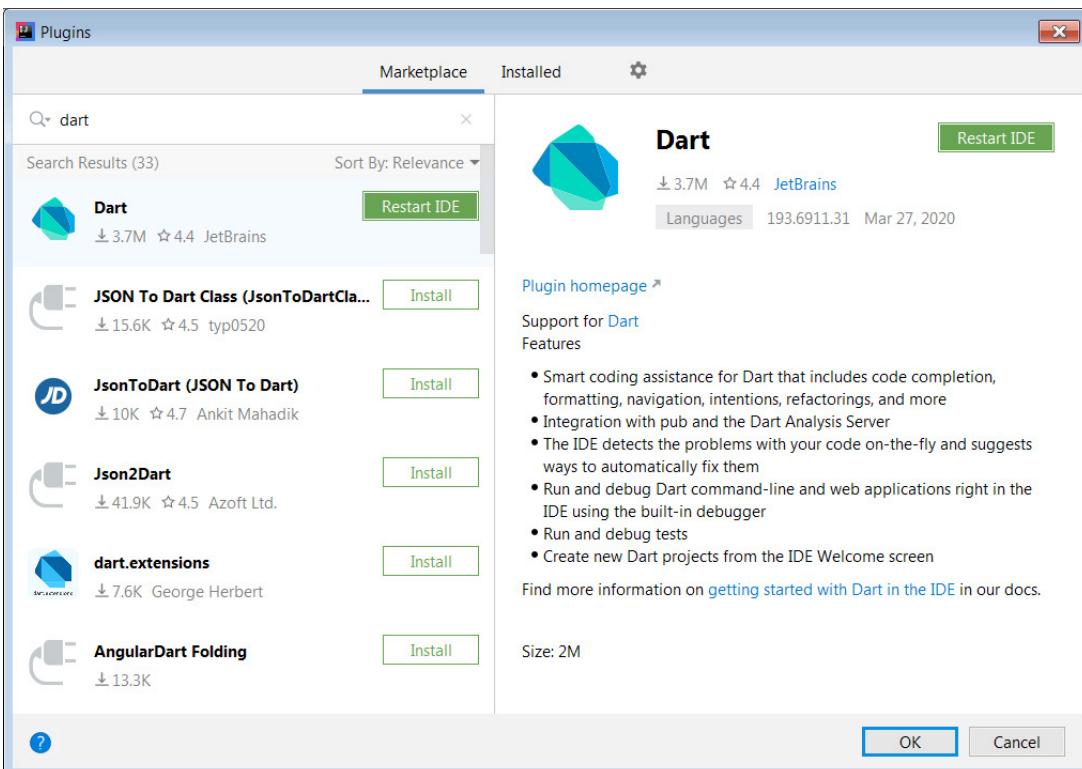
16- You will get the following figure. Move your mouse cursor to the search area as illustrated in the blow figure, type **Dart**, and then press **Enter** key



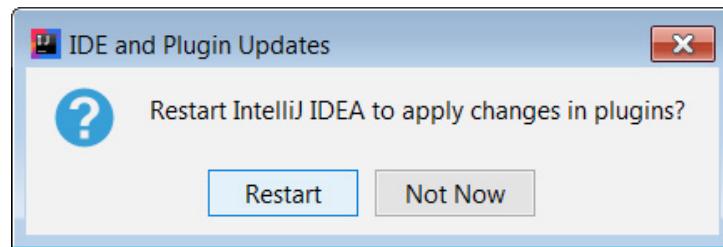
17- You will get the following figure. Click **Install**



18 - After completing the Dart plug-in installation step, click **Restart IDE** .



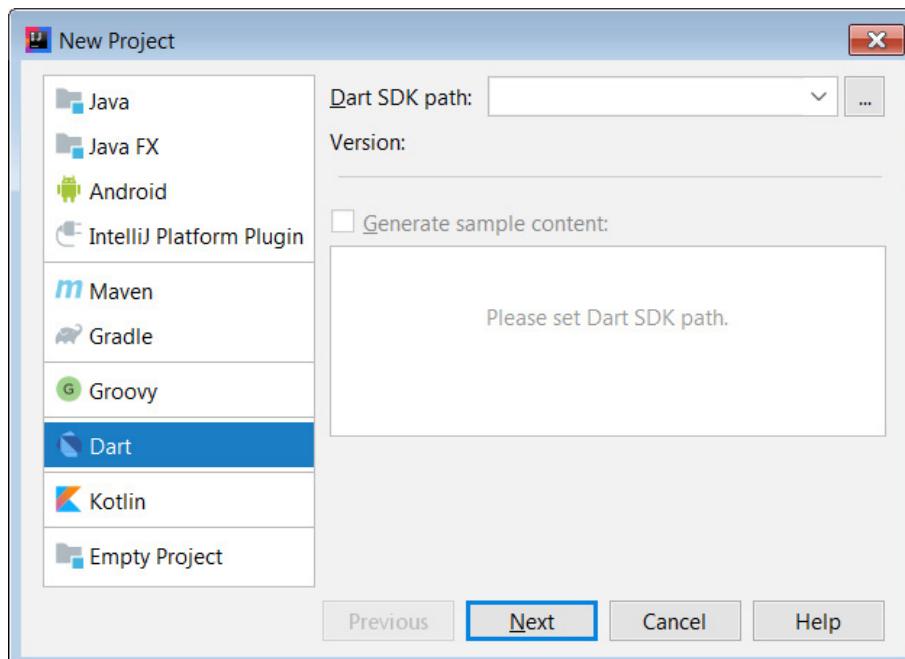
19 - Click **Restart**.



20 - Click **Create New Project**.

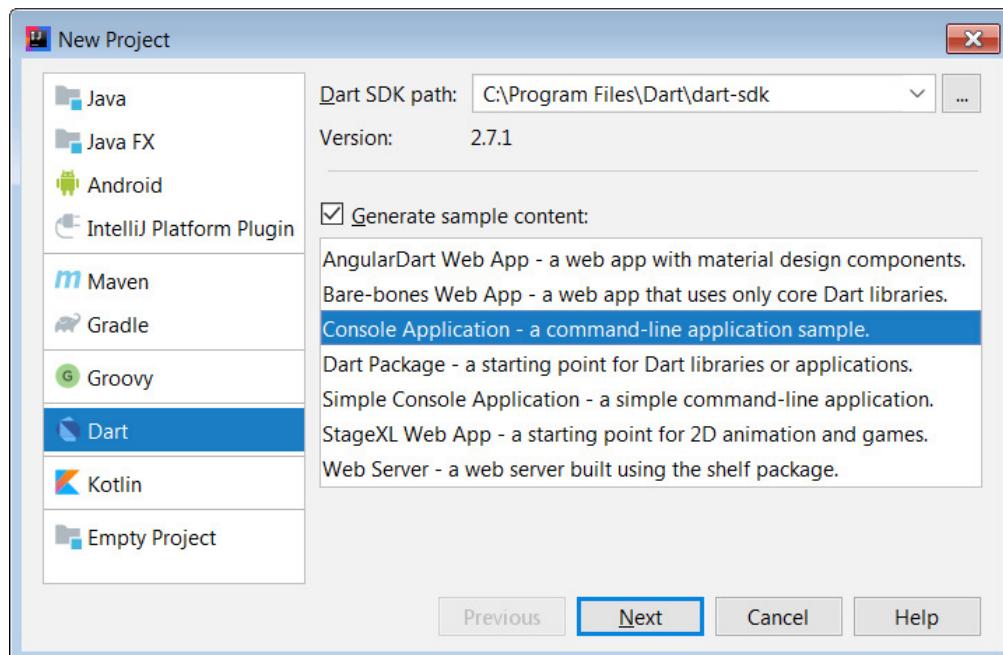


21- To connect IntelliJ IDEA with Dart SDK, as illustrated in the below figure, click **Dart** in the left side.

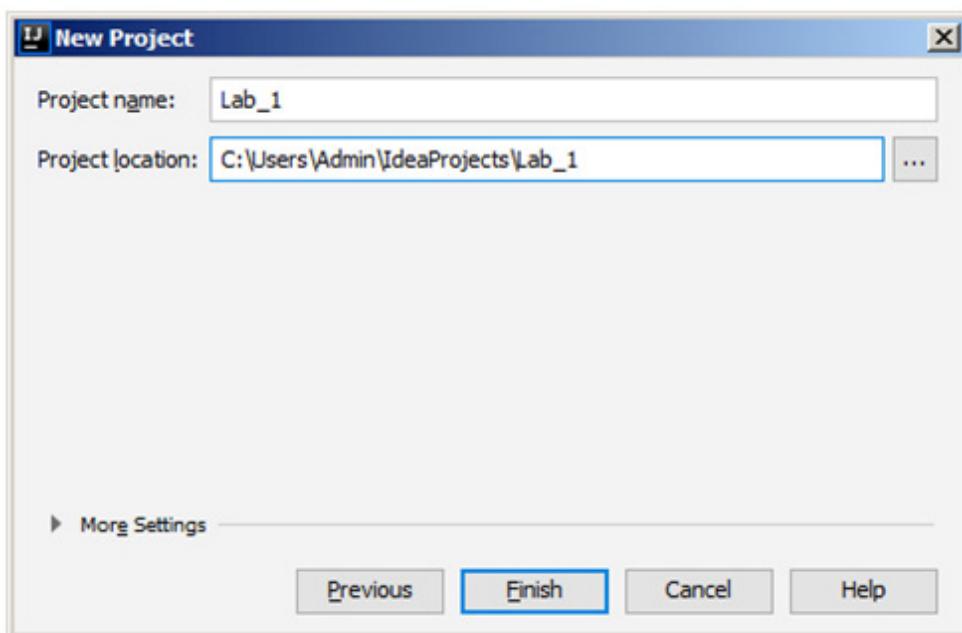


22- Configure the Dart SDK path on your computer. Click the browse button, then click your Dart SDK path. As illustrated in the below figure, **Dart SDK path** is :  
**C:\Program Files\ Dart\dart-sdk**

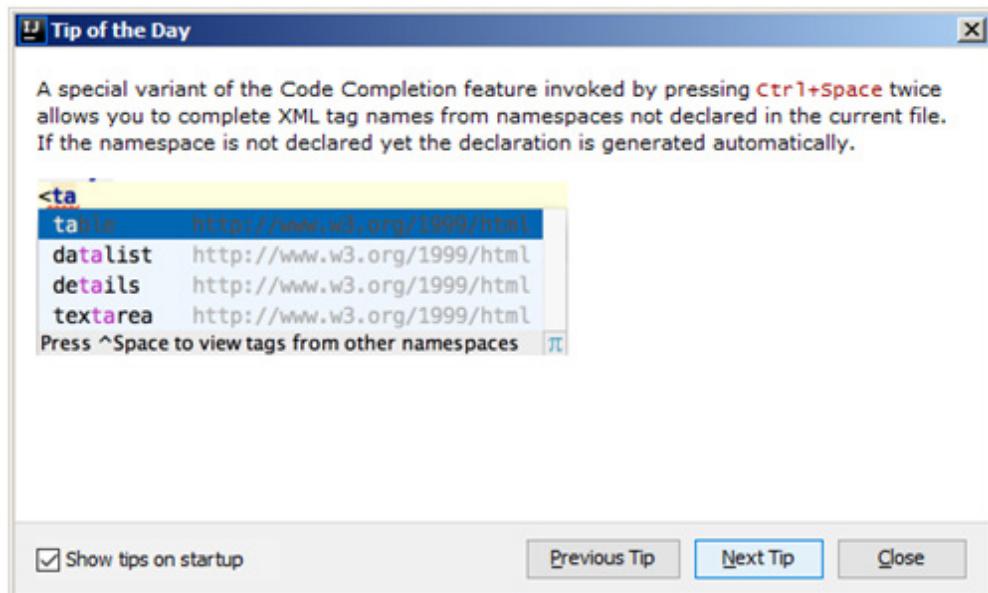
To generate a sample code just to test the working of IntelliJ IDEA with Dart, select **Console Application - a command-line application sample**, and then click **Next**



23- As illustrated in the next figure, type the **Project name: Lab\_1**, then click **Finish**



24- Remove the check mark at **Show tips on startup** as illustrated in the below figure, and click **Close**

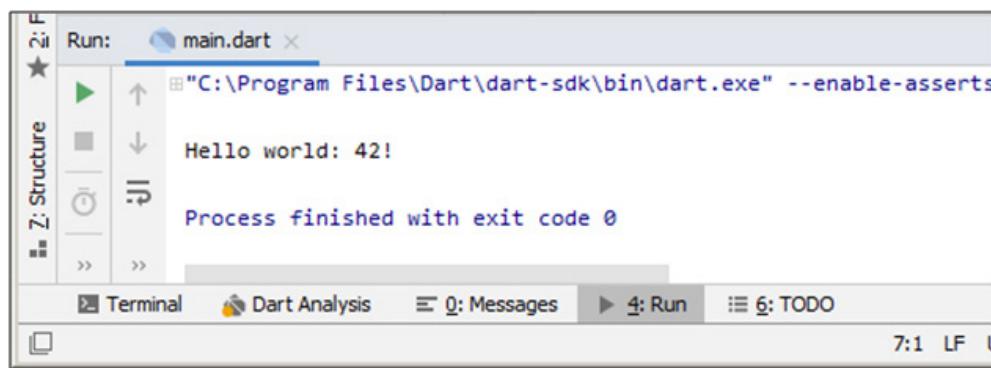


25- You will get the following sample code.

```
main(List<String> arguments) {  
    print('Hello world: ${Test.calculate()}!');  
}
```

Now, we will run this sample code just to be sure that IntelliJ IDEA works fine with Dart configuration. Click **Run** button or click **Run** menu → **Run**  
As illustrated in the below figure, you will get the following run code result:

"Hello world: 42 !", this means IntelliJ IDEA works fine.



## ➤ Creating a Dart Project Using IntelliJ IDEA

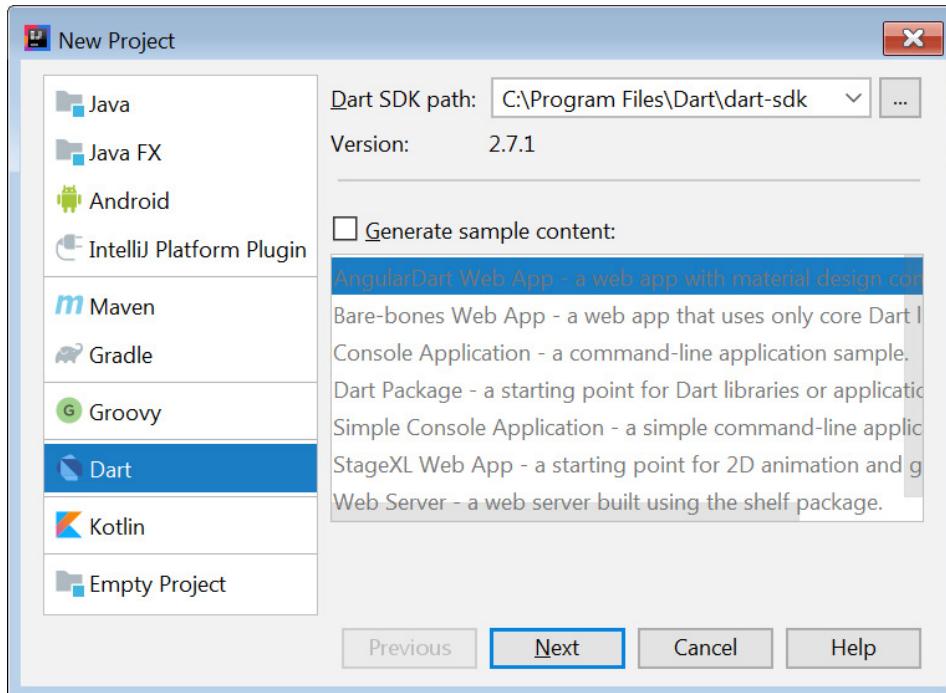
In this lab, you will create and run a small Dart project using IntelliJ IDE.

To create a new Dart project, perform the following steps:

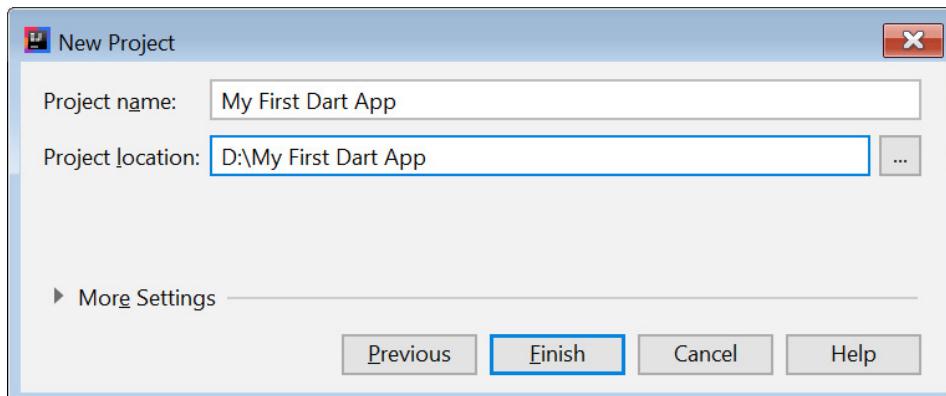
1- Open IntelliJ IDEA

2- Click File → New → Project

You will get the following figure. Remove the check box for : **Generate sample content**, then click **Next**

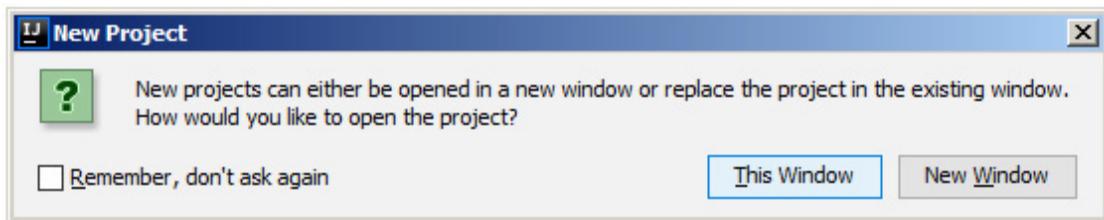


3- In the New Project dialog box, as illustrated in the below figure, type the Project Name: **My First Dart App**, then click **Finish**.

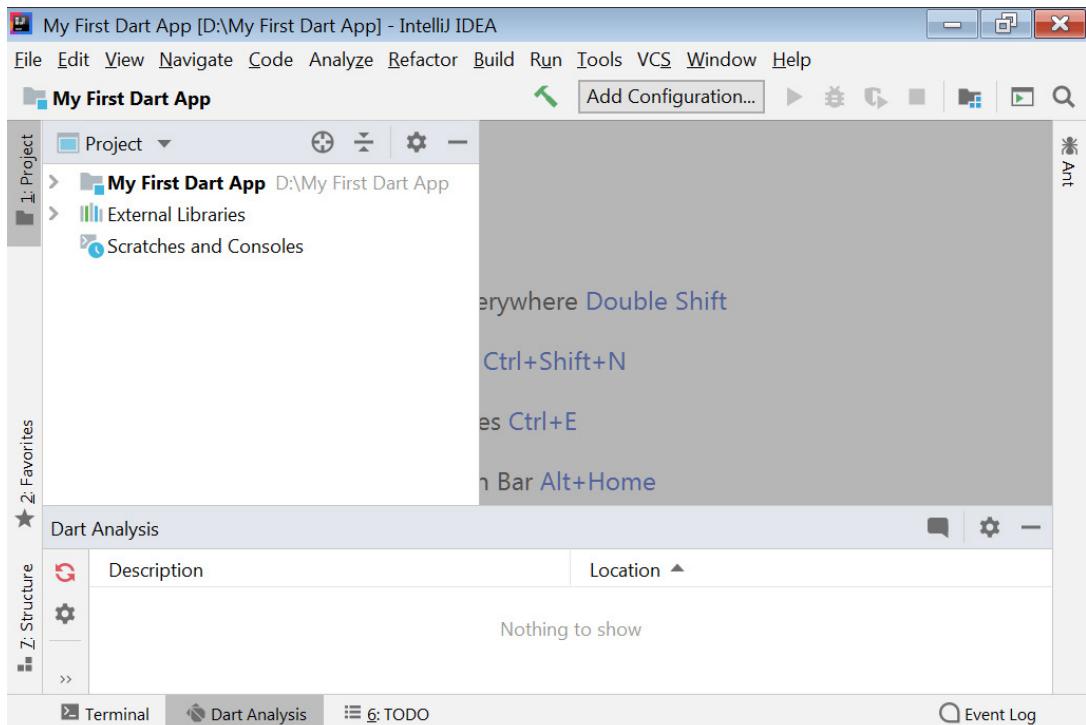


Click **OK** to create this project directory.

4- In the below figure, click **This Window**

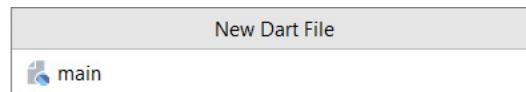


5- You will get the following figure :

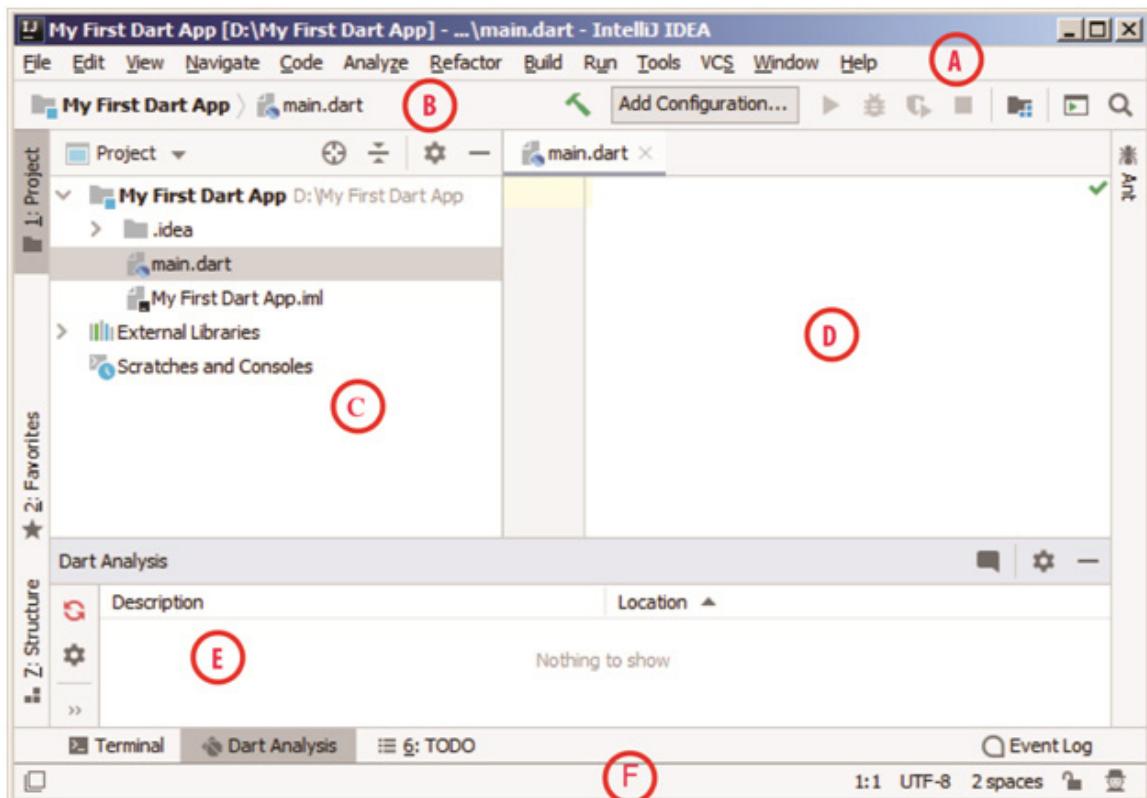


6- In the Project console, right click the project name :  
“My First Dart App” → New → Dart File

As illustrated in the figure below, type : **main** for the file name, then press **Enter**.



You will get the following figure:



In the previous figure, we have labeled the IntelliJ interface parts as follows :

**A** : Menu and Tools Bar

**B**: Navigation Bar

**C**: Project Console

**D** : Editor Window

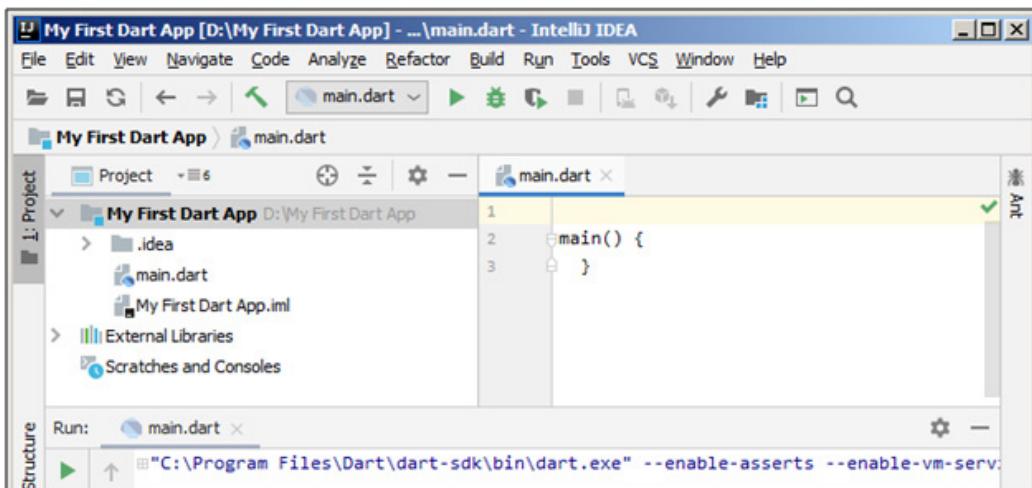
**E**: Tool Window

**F**: Status Bar

7- In the editor window, type the following code:

```
main() {  
}  
}
```

As illustrated in the following figure:



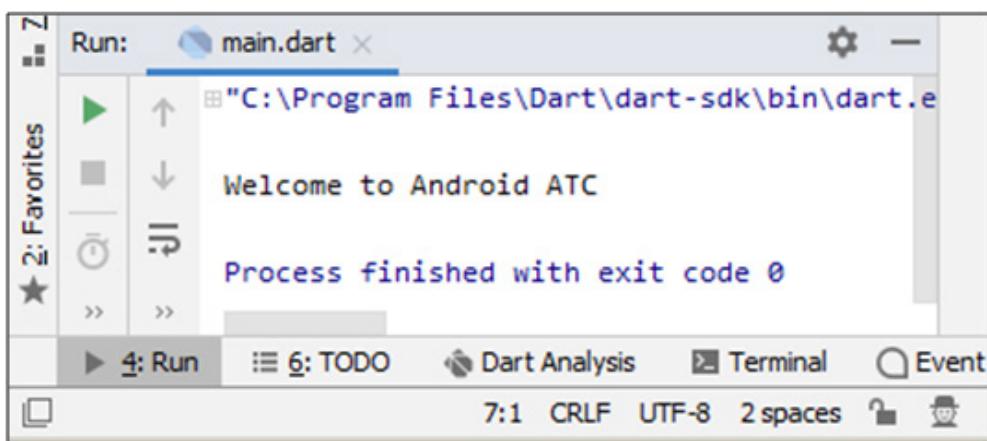
**main()** is a method and you must write the entire code within the two braces {} of this main method.

8- Write the following code within the two braces of the **main** method as follows:

```
main() {
    print('Welcome to Android ATC');
}
```

The **print** method is used in Dart to print text or variable value in the Run console as illustrated in the figure below.

Then, from **Run** menu, click **Run**. You will get the following run result:



## ➤ Using DartPad

**DartPad** is an open-source tool that lets you work with the Dart language in any modern web browser. It is a great, no-download-required way to learn Dart syntaxes and to experiment Dart language features.

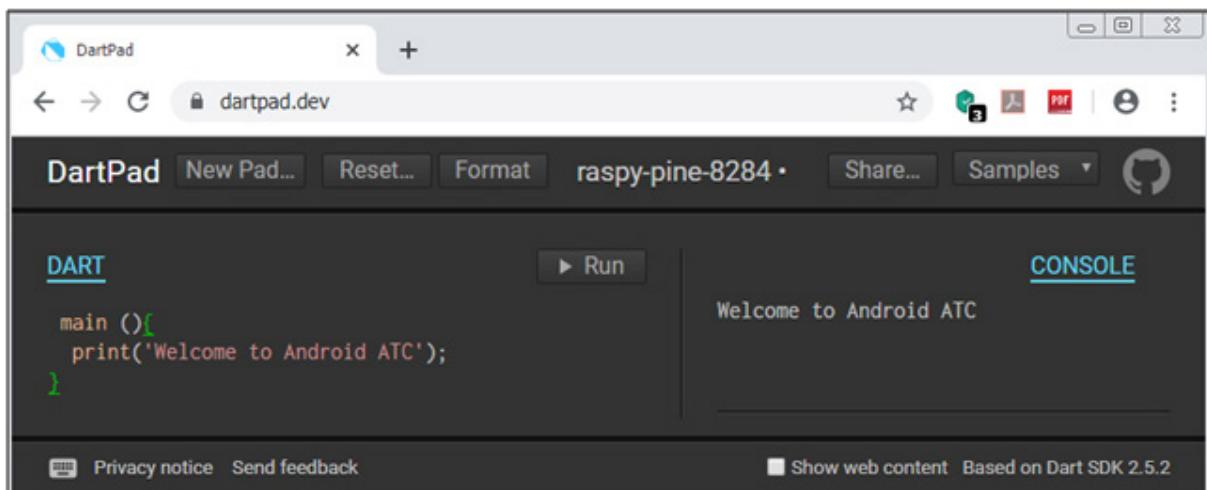
The following steps show how you can write a Dart code using DartPad:

1- Go to: <https://dartpad.dev>

2- Delete the existing code which are in the left side, then write the following code:

```
main() {  
  print('Welcome to Android ATC');  
}
```

3- Click the **Run** button, then you will get the running program output in the **CONSOLE** side as illustrated in the following figure:



## Lesson 2: Dart Programming - Syntax

<b>Introduction .....</b>	2-2
<b>main( ) function.....</b>	2-2
<b>Dart Variables.....</b>	2-4
<b>Dart Data Types.....</b>	2-5
<b>Input of Information to Dart Program.....</b>	2-15
<b>Writing Comments .....</b>	2-17
<b>Dart Conditional Operators.....</b>	2-19
<b>If Statement.....</b>	2-22
<b>If – Else Statement .....</b>	2-24
<b>If...Else and Else...If... Statement .....</b>	2-25
<b>If Else and Logical Operators .....</b>	2-26
<b>For Loops.....</b>	2-28
<b>While Loops .....</b>	2-29
<b>Do-while Loops .....</b>	2-31
<b>Break Statement .....</b>	2-32
<b>Switch Case Statement .....</b>	2-33
<b>Lab 2: Create a Pizza Order Program.....</b>	2-36

## Introduction

This lesson shows you how to use major Dart features including variables, operators, classes and libraries.

Dart is an object oriented language programming that consists of programming syntaxes which define a set of rules for writing programs. Every language specification defines its own syntaxes. A Dart program is composed of the following statements and expressions:

- Variables and Operators
- Comments
- Decision Making and Looping Constructs
- Classes
- Functions
- Expressions and Programming Constructs
- Libraries and Packages
- Data structures represented as Collections / Generics

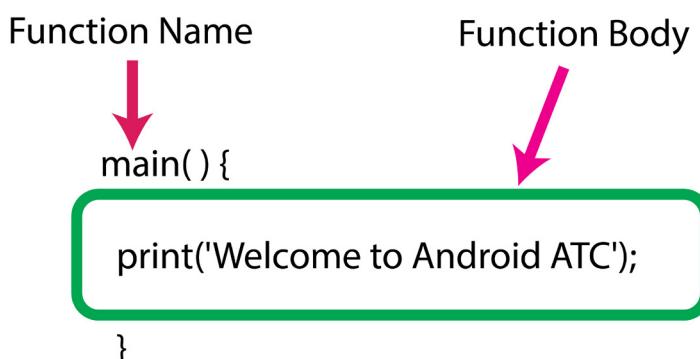
In this lesson, you will learn more about each of the above statements, especially the role and the use of each of them with examples.

## main() function

A function is any close identity which includes a certain code or a collection of statements grouped together to perform an operation.

Every app must have a top-level `main()` function, which serves as the entry point to the app.

The following image displays the `main()` function structure:



All your Dart code must be written inside the `main()` function body.

**Example:**

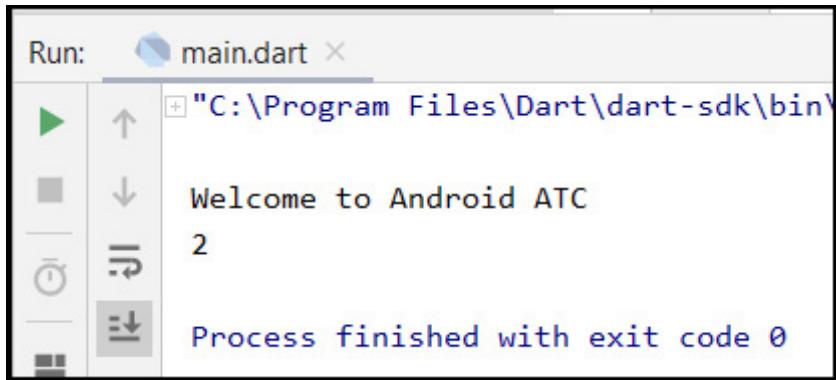
1- Open IntelliJ IDEA

2- Write the following code inside the main function body :

```
main() {  
  print('Welcome to Android ATC');  
  print(1+1);  
}
```

3- Run this Dart file (Click **Run** menu , then select **Run**).

The result follows:



**Example :**

Continue using the previous example, and add another function called `test()` outside the `main()` function body as illustrated in the code below:

```
main() {  
  print('Welcome to Android ATC');  
  print(1+1);  
}  
test(){  
  print('Hello Flutter Developers');  
}
```

When you run this Dart code, you will get the same previous result without any

effect to the content of the `test()` function.

This means that, when you run any Dart code, only the code inside the `main()` function body will run. Also, if you want to run any function outside the `main()` function body, you must add a reference to this external function inside the `main()` function as you will see in the next topics.

## Dart Variables

A Dart variable is a piece of memory that can contain a data value. Variables are typically used to store information which your Dart program needs to do its job. These variables are case sensitive.

To declare a variable, write `var` directly before the variable. The following example gives you an idea about how to use a variable called `x`:

```
main() {
  print('Welcome to Android ATC');
  var x=1;
  print(x);
}
```

When you run this Dart program, you will get the following result for `x` value:

```
Run: main.dart ×
" C:\Program Files\ Dart\dart-sdk\bin\dart.exe "
Welcome to Android ATC
1
Process finished with exit code 0
```

You also can declare the variables outside the `main()` function body as illustrated in the following example:

```
var y=2;
main() {
  print('Welcome to Android ATC');
```

```
var x=1;  
print(x);  
print(y);  
}
```

When you run this Dart program, you will get the following result for x and y values:

```
Run: main.dart  
"C:\Program Files\Android\SDK\bin\flutter.bat"  
Welcome to Android ATC  
1  
2  
Process finished with exit code 0
```

## Dart Data Types

The basic data types used in Dart are strings, Booleans, numbers, lists, and maps. The following are examples with some details about each data type:

### 1) String

String data type is used to store words or sentences. If you want to assign a data type for a specific variable as string, you cannot assign a number or a symbol for this variable

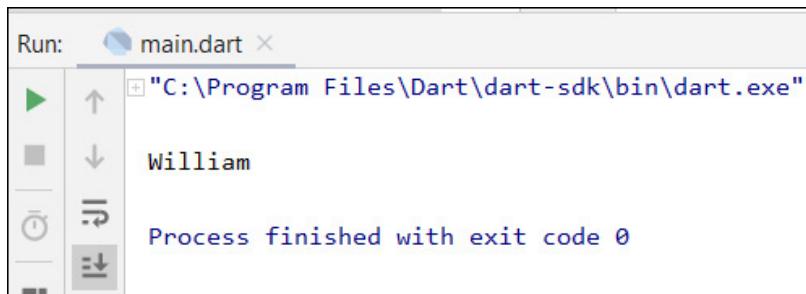
String values in Dart can be represented using either single or double quotes.

#### **Example:**

The following code displays how you must declare a string variable. In this example, the word **String** has been written directly before the *name* variable.

```
main() {  
  String Name= 'William';  
  print(Name);  
}
```

The following is the run result of the above Dart code:



## 2) Booleans

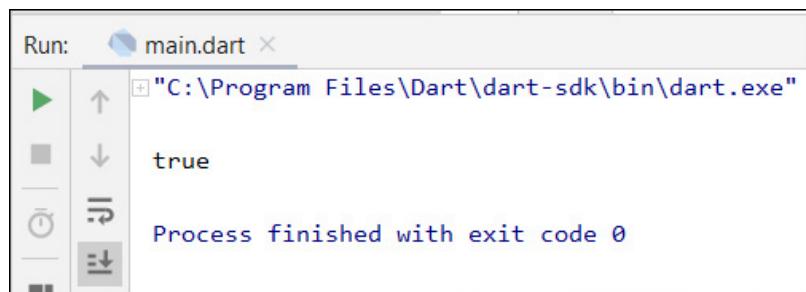
A Boolean data type has two possible values; either true or false. Booleans are used in decision making statements which you can control in the program work flow.

### **Example:**

The following code displays how you must declare a Boolean variable. To declare a Boolean value, you must write **bool** directly before the variable.

```
main() {
  bool xyz;
  xyz = 12 > 5;
  print(xyz);
}
```

The following is the run result of the above Dart code :



If you changed the formula to `xyz=12 < 5;` and run the code, the run output will be : `false`. This is because `xyz` variable has been declared as a Boolean variable.

### 3) Numbers

Dart provides the following built-in types that represent numbers:

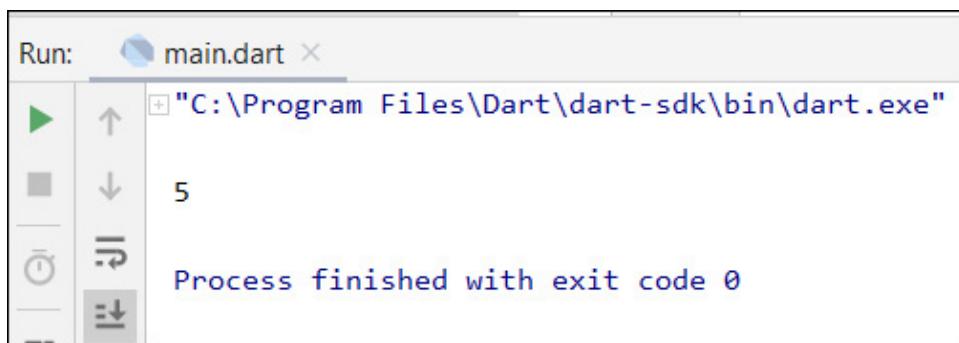
Data Type	Description
Int	The integer data type is a 32-bit signed integer. It has values from -2,147,483,648 to +2,147,483,647
Double	The double data type is a double-precision 64-bit floating point.

To declare an integer variable, write **int** directly before the variable.

The following example shows how to declare integer variables and use them in a sum formula:

```
main() {  
  int x=3;  
  int y=2;  
  int z=x+y;  
  print(z);  
}
```

The following is the run result of the above Dart code:

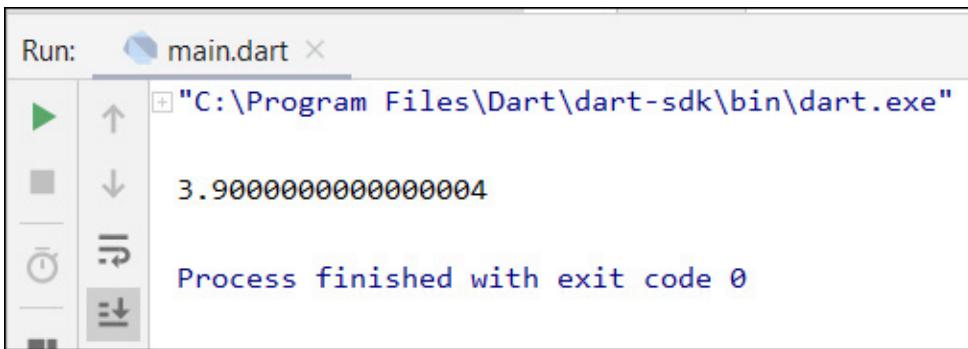


A screenshot of a Dart IDE showing the run output. The window title is "Run: main.dart". The output pane displays the following text:  
"C:\Program Files\ Dart\dart-sdk\bin\dart.exe"  
5  
Process finished with exit code 0

If you want to use a decimal number, you must use double data type. To declare a double variable, write **double** directly before the variable as illustrated in the following code:

```
main() {  
  double height=1.5;  
  double width=2.6;  
  double area=height*width;  
  
  print(area);  
}
```

The following is the run result of the above Dart code :

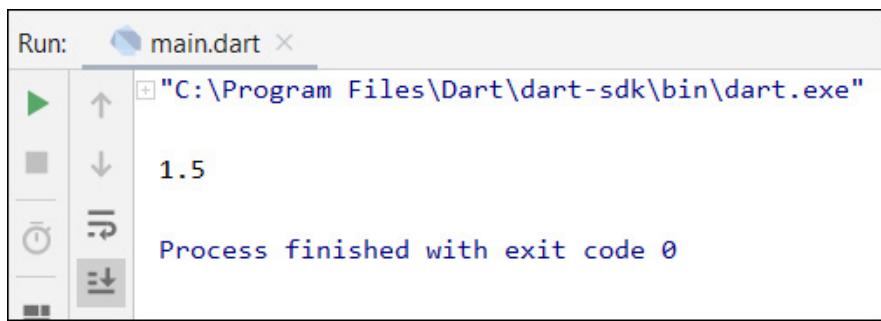


Also, as it illustrated in the following code, you may use "num" to declare an integer or a double number.

```
main() {  
  num x=1;  
  num y=1.5;  
  num z=x*y;  
  print(z);  
}
```

Here, **x** is declared implicitly as an integer number; however, **y** is declared implicitly as a double number.

The following is the run result of the previous Dart code:

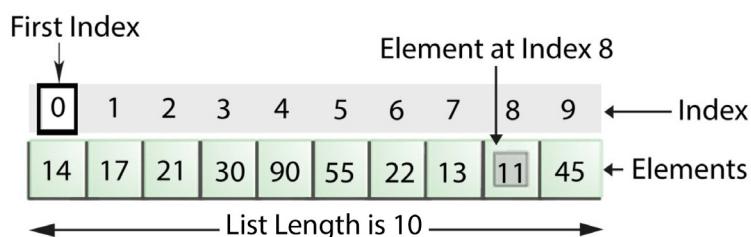


You can say in a simple way, we will use integer for whole numbers and double for decimal numbers.

## 4) Lists

A very commonly used collection in programming is an array. Dart represents arrays in the form of List objects. If you want to store a large number of data items for the same variable, you need to use the list. A list is used to store a group of values, all of which have the same data type.

As you see in the following figure, this list contains 10 values starting from list [0] whose value is 14 until list [9] whose value is 45.

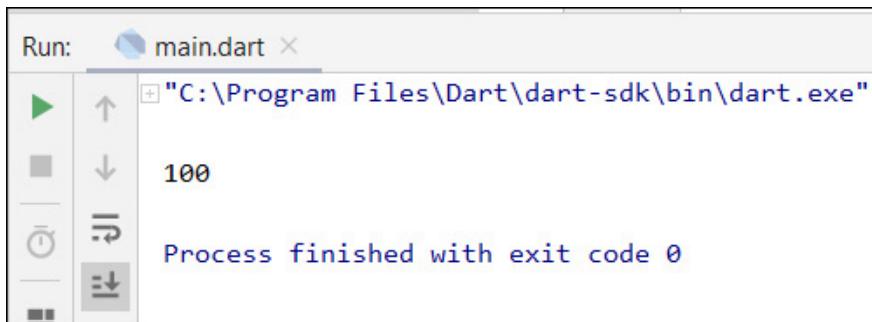


### Example:

The following code includes a list called `test_list` contains 10 different values :

```
main() {
  var test_list = [7,3,100,50,9,30,8,11,6,-4];
  print(test_list[2]);
}
```

The following is the run result of the previous Dart code :



```
Run: main.dart ×
C:\Program Files\...dart.exe"
100
Process finished with exit code 0
```

Also, you can write the same previous list code and declare their data type as integer as follows:

```
main() {
  List<int> test_list = [7,3,100,50,9,30,8,11,6,-4];
  print(test_list[2]);
}
```

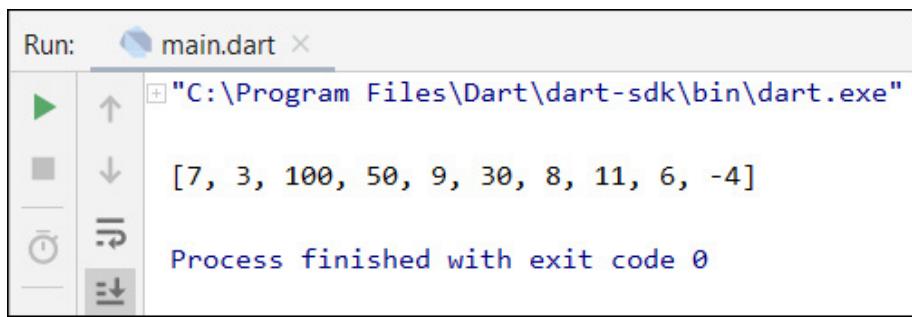
The run result is the same : 100

**Example:**

The following code is to print the all the content of the List test\_list :

```
main() {
  var test_list = [7,3,100,50,9,30,8,11,6,-4];
  print(test_list);
}
```

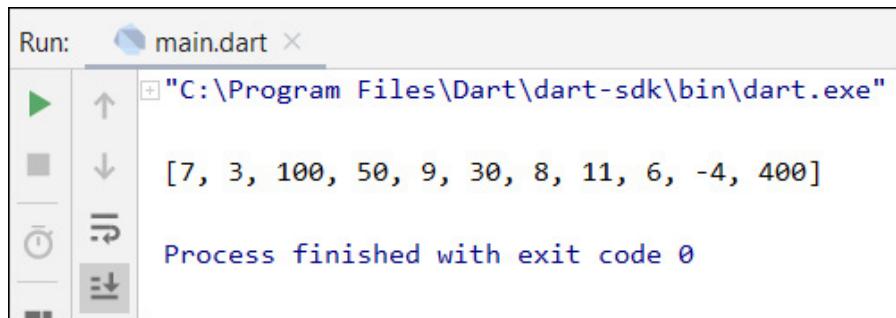
The following is the run result of the above Dart code :

**Example:**

In the following Dart code, you will see how to use the add method to add a new value to the existing List. The new value will be added at the end of this List.

```
main() {  
  var test_list = [7,3,100,50,9,30,8,11,6,-4];  
  test_list.add(400);  
  print(test_list);  
}
```

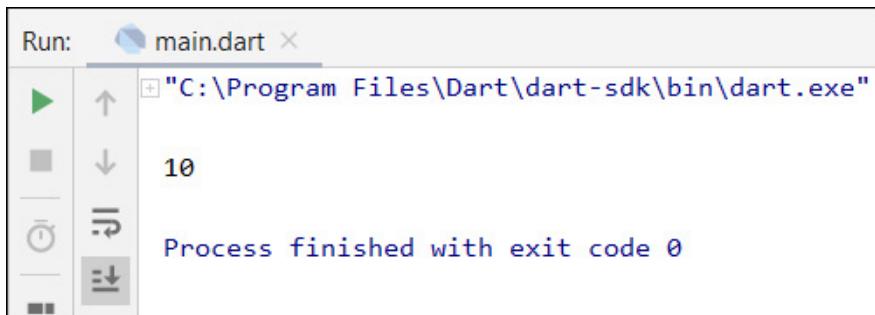
The following is the run result of the above Dart code :

**Example:**

In the following Dart code, **test\_list.length** method represents the number of the list length. Here in this example, the list contains 10 numbers from **test\_list[0]** until **test\_list[9]**.

```
main() {  
    var test_list = [7,3,100,50,9,30,8,11,6,-4];  
  
    print(test_list.length);  
}
```

The following is the run result of the above Dart code :



**Example:**

As illustrated in the following Dart code, the **forEach** method is used to print each list element.

```
main() {  
    var test_list = [7,3,100,50,9,30,8,11,6,-4];  
    test_list.forEach((x){  
  
        print(x);  
    });  
}
```

## 5) Maps

DART maps is an object that associates keys to values. In Dart, maps is an interface designed to manipulate a collection of keys which points to values. Maps can be declared in two ways, using maps literals and using a map constructor as follows:

- **Declare Maps Using Map Literals**

To declare a map using map literals, you need to enclose the key-value pairs within

a pair of braces

"{ }".

**Example:**

```
main() {  
  var info = {'UserName': 'Kevin@androidatc.  
com', 'Password': 'pass123'};  
  print(info);  
}
```

The following is the run result of the above Dart code :



```
Run: main.dart ×  
+ "C:\Program Files\ Dart\dart-sdk\bin\ dart.exe" --enable-  
  {UserName: Kevin@androidatc.com, Password: pass123}  
  Process finished with exit code 0
```

- **Declare Maps Using a Map Constructor :**

To declare a Map using a Map constructor, there are two steps. First, declare the map as follows:

```
var info = new Map();
```

The second, initialize the map as follows:

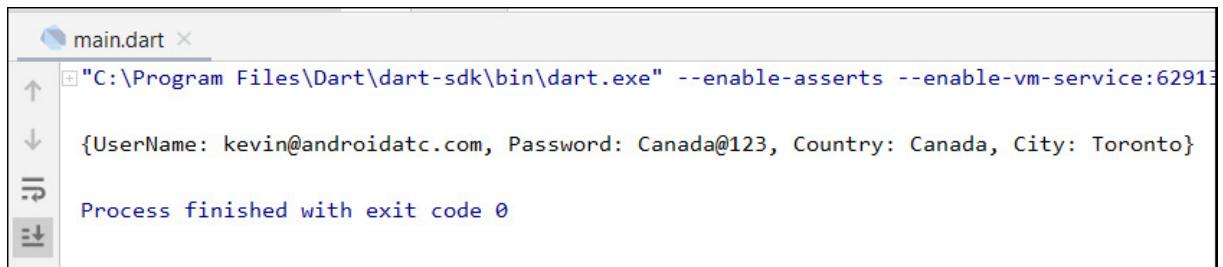
```
info['UserName'] = 'kevin@androidatc.com';
```

**Example:**

```
main() {  
  var info = new Map();  
  info['UserName'] = 'kevin@androidatc.com';  
  info['Password'] = 'Canada@123';  
}
```

```
info['Country'] = 'Canada';
info['City'] = 'Toronto';
print(info);
}
```

The following is the run result of the above Dart code :



```
main.dart ×
+ "C:\Program Files\Dart\dart-sdk\bin\dart.exe" --enable-asserts --enable-vm-service:62913
{UserName: kevin@androidatc.com, Password: Canada@123, Country: Canada, City: Toronto}
Process finished with exit code 0
```

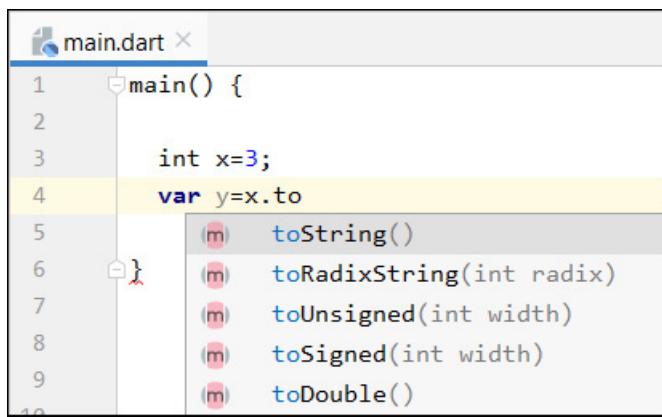
## Explicitly an Implicitly Data type

The following example displays how you can declare a data type both explicitly and implicitly for variables:

```
main() {
  var x=10; // implicitly Integer number
  int y=20; // explicitly Integer number
  var city="Toronto"; // implicitly String
  String country="Canada"; // explicitly String
}
```

## Data type Conversions

In some cases, you may need to convert a data type for a variable to another data type such as changing an Integer to a String. To do that, you need to use `toString()` function explicitly to convert the variable to data type String, as illustrated in the following figure:



## Input of Information to Dart Program

`stdin.readLineSync()` function allows the program user to enter string values or intercept keyboard input from the console as shown in following example:

Here, at the first line of code, add : `import 'dart:io'` to import the `dart:io` library. This library allows IntelliJ to accept the input of information commands.

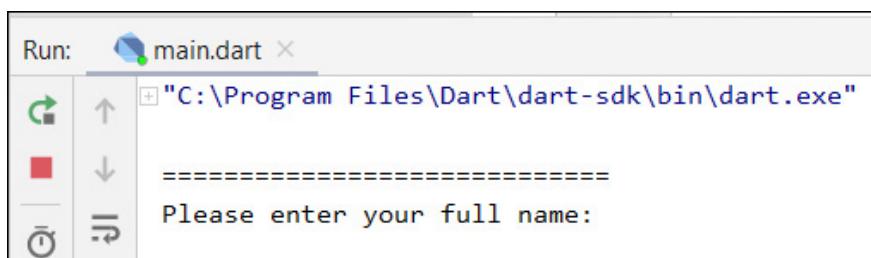
```

import 'dart:io';
main() {
    print('=====');
    print('Please enter your full name:');

    String Full_Name=stdin.readLineSync();
    print('Hello:$Full_Name');
    print('=====');
}

```

When you run this program, you will get the following message:



Then, enter your full name, then press **Enter** or **Return** key, you will get the following result :



The `stdin.readLineSync()` is used to enter string data type only. To configure the Dart program to enter an integer, use `int.parse(stdin.readLineSync())` to convert the string value to integer. The following code displays how to create a program to enter integer values:

```
import 'dart:io';
main() {

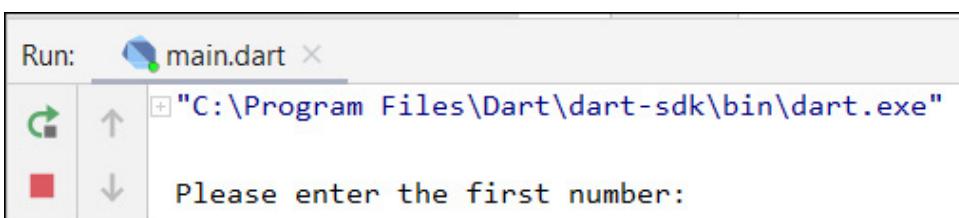
  print('Please enter the first number:');
  int num1=int.parse(stdin.readLineSync());

  print('Please enter the second number:');
  int num2=int.parse(stdin.readLineSync());

  var sum=num1+num2;

  print('The sum result = $sum');
}
```

When you run this program, you will get the following message :



Enter 5, then press **Enter** or **Return** key. You will get the following message:

*Please enter the second number:*

Enter **3**, then press **Enter** or **Return** key, you will get the following result :

```
Run: main.dart
C:\Program Files\ Dart\dart-sdk\bin\dart.exe
Please enter the first number:
5
Please enter the second number:
3
The sum result = 8
Process finished with exit code 0
```

## Writing Comments

Any line starting with a double forward slash will be considered a comment by the Dart compiler. This means that this part will not run or appear to the users of this application because it will remain internal. Comments are used to write notes about different parts of the Dart program.

### **Example:**

In the previous code, you may add the following line as a comment:

```
// The following code to input data to Dart program
```

The code follows:

```
import 'dart:io';
main() {

// The following code to input data to Dart program

print('Please enter the first number:');
int num1=int.parse(stdin.readLineSync());
```

```
print('Please enter the second number:');
int num2=int.parse(stdin.readLineSync());
var sum=num1+num2;
print('The sum result = $sum');
}
```

When you run this program, the part which starts with a double forward slash will not appear to users.

Also, if you want to stop any part of the program temporarily, you can do that by adding a double forward slash at the beginning of this line of code as illustrated in the grey highlighted part of the following code:

```
import 'dart:io';
main() {

// The following code to input data to Dart program

// print('Please enter the first number:');
int num1=int.parse(stdin.readLineSync());

print('Please enter the second number:');
int num2=int.parse(stdin.readLineSync());
var sum=num1+num2;
print('The sum result = $sum');
}
```

Besides, you may configure a block of comments which include more than one line if it starts with /\* and ends with \*/. These /\* & \*/ are used to add multiple lines of comments in the code without adding // at the beginning of each line as illustrated in the following code:

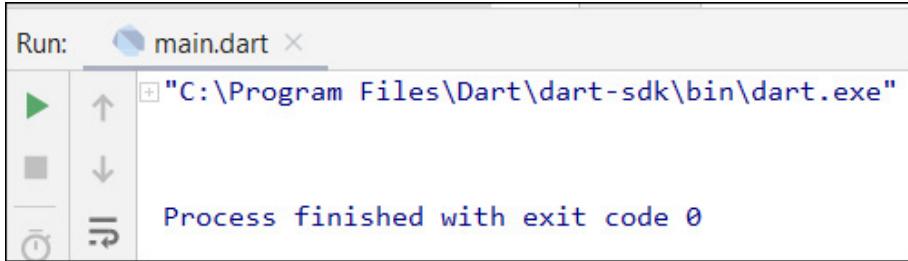
```
import 'dart:io';
main() {

/* print('Please enter the first number:');
int num1=int.parse(stdin.readLineSync());

print('Please enter the second number:');
int num2=int.parse(stdin.readLineSync());
var sum=num1+num2;
print('The sum result = $sum');
```

```
 */  
}
```

If you run this program, you will get the following:



## Dart Conditional Operators

The following table includes some conditional operators that help in the control flow of the Dart program:

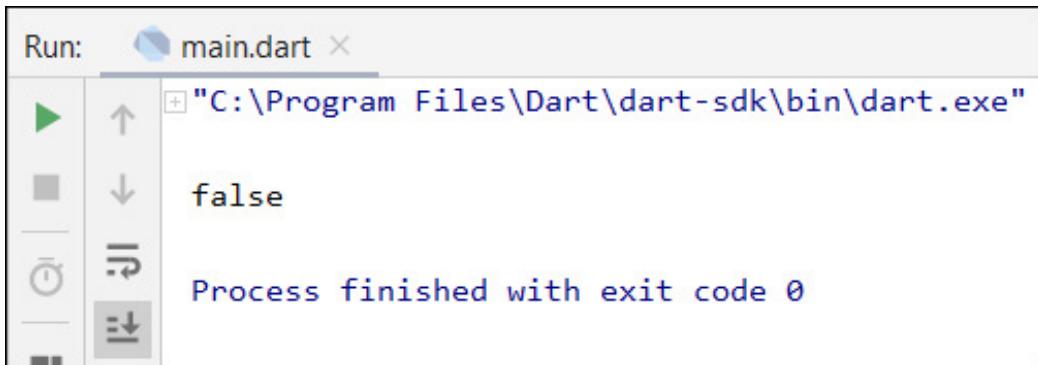
Operator	Description
<code>==</code>	Equal to
<code>!=</code>	Not Equal to
<code>&lt;</code>	Less than
<code>&lt;=</code>	Less than or equal to
<code>&gt;</code>	Greater than
<code>&gt;=</code>	Greater than or equal to
<code>?</code>	Return value of two expressions
<code>??</code>	Return the which is not equal null value of two expressions
<code>is</code>	Is
<code>is!</code>	Is Not

**Example:** In the following code, you will test the role of double equal sign "==" which is used to compare if two values are equal or not. If they are equal, then the run result will be true; otherwise, the run result will be false.

```
main() {  
    int x=3;
```

```
int y=5;  
print(x==y);  
}
```

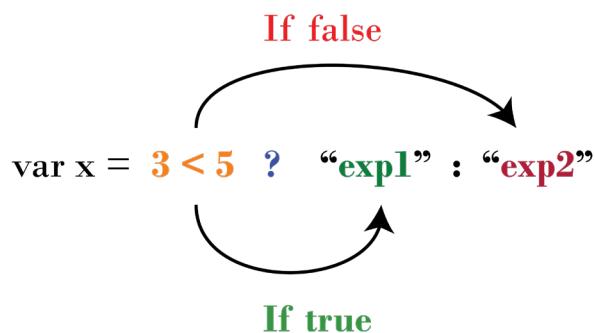
The run result of this code follows:



```
Run: main.dart  
+ "C:\Program Files\ Dart\dart-sdk\bin\dart.exe"  
false  
Process finished with exit code 0
```

**Example:**

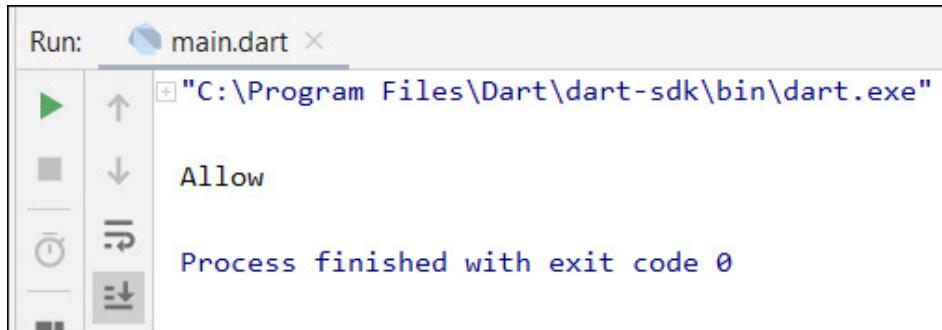
The following Dart conditional operator "?" will return one expression (exp1 or exp2) as illustrated in the following figure :



The following code has an example about using "?" Dart conditional operator:

```
main() {  
  var Age=20;  
  var x=Age >18 ? "Allow" : "Deny";  
  print(x);  
}
```

Because the "Age" value in this example is 20, the condition result is true, so the run result will be Allow as illustrated in the following figure:

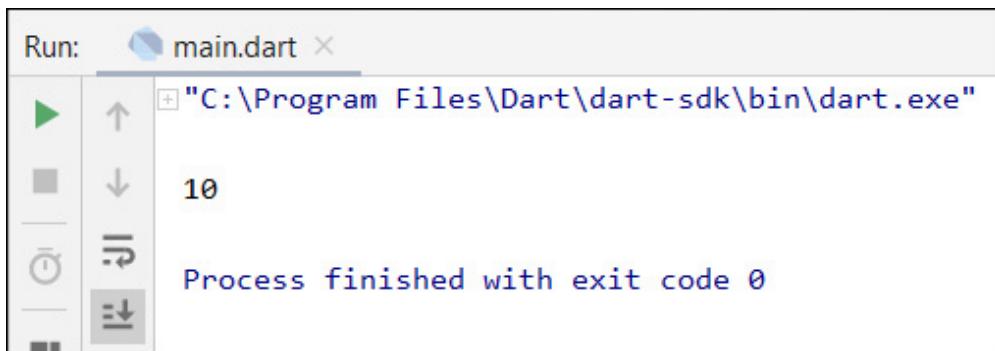


**Example:**

In the following example, the conditional operator "??" is used to compare two expressions, and then return the value of the expression which is not equal to a null value as illustrated in the following code:

```
main() {  
  var x=null;  
  var y=10;  
  var z=x ?? y ;  
  print(z);  
}
```

The run result of this code follows:



**Example:**

The following code displays using "is" operator. If the condition operator result is true, the output will be **true**; otherwise, the result will be **false**.

```
main() {
  int x=5;

  print(x is bool);
}
```

The run result of this code follows:

```
Run: main.dart
+ "C:\Program Files\...dart.exe"
false
Process finished with exit code 0
```

## If Statement

The "if" statement is a programming conditional statement that, if proved true, executes the second part of the statement. Otherwise, if proved false, the program will skip the execution of the second part and do something else.

### *Example:*

In the following example, the Dart compiler will consider the "if" condition and the statement highlighted in grey color in this example as one block. `if x>5` (true) the program will print a hello message, and if it is not (`false`), the program will continue to perform the next action.

```
main() {
  int x=10;

  if(x>5){
    print("Hello, I am If statement running now .....");
  }
  print('The End');

}
```

The run result of this code follows:

The screenshot shows a 'Run' interface with a file named 'main.dart'. The output window displays the following text:  
"C:\Program Files\ Dart\dart-sdk\bin\dart.exe"  
Hello, I am If statement running now .....  
The End  
Process finished with exit code 0

If you change the program as follows,

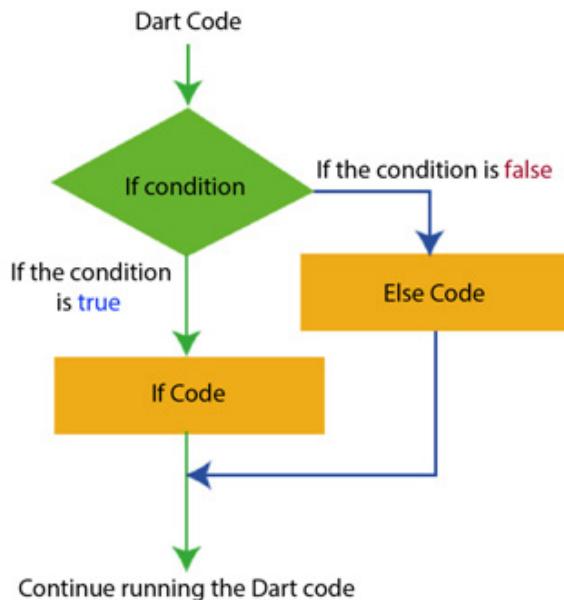
```
main() {  
  int x=2;  
  
  if(x>5){  
    print("Hello, I am If statement running now .....");  
  }  
  print('The End');  
  
}
```

the run result of this code follows:

The screenshot shows a 'Run' interface with a file named 'main.dart'. The output window displays the following text:  
"C:\Program Files\ Dart\dart-sdk\bin\dart.exe"  
The End  
Process finished with exit code 0

## IF – Else Statement

The **if-else** statement is the most basic of all the control flow statements. It invokes your program to execute a specific section of a code only if a particular test proves to be true. The following figure displays the work of the IF - Else statement:



### **Example:**

This example gives an idea of how **if-else** statement works in an easy way:

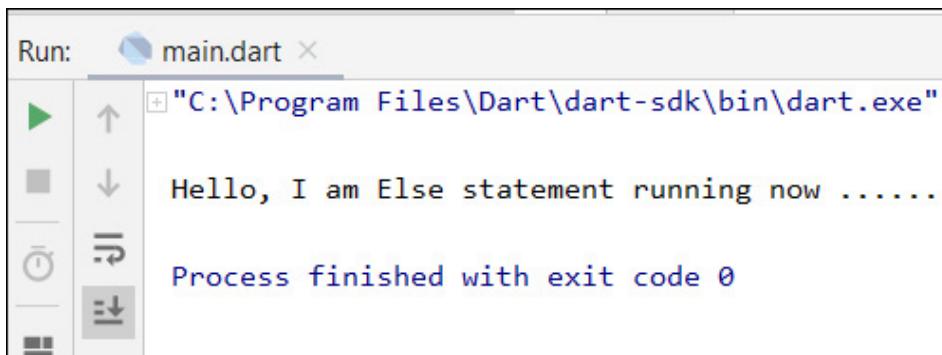
In the Dart program below, when the program starts, the value of the **x** variable =10.

In the next line, the "**if**" statement will work using the greater than condition. After the Dart compiler checks whether this condition is true or not, Dart compiler will directly run the next line of code. However, if the condition is proved not true, Dart compiler will continue to run the code lines under the **Else** statement.

```

main() {
  int x=10;
  if(x>30){
    print("Hello, I am If statement running now .....");
  } else {
    print("Hello, I am Else statement running now .....");
  }
}
  
```

The run result of this code follows:



```
Run: main.dart
C:\Program Files\ Dart\dart-sdk\bin\dart.exe
Hello, I am Else statement running now .....
Process finished with exit code 0
```

## IF...Else and Else...IF... Statement

This statement is the most basic of all control flow statements. It invokes your program to execute a certain section of code only if a particular test evaluates to be true.

An **if**-statement can be followed by an optional **else-if-else** statement when testing multiple conditions.

The following code displays how **if-else** and **else-if** statements work:

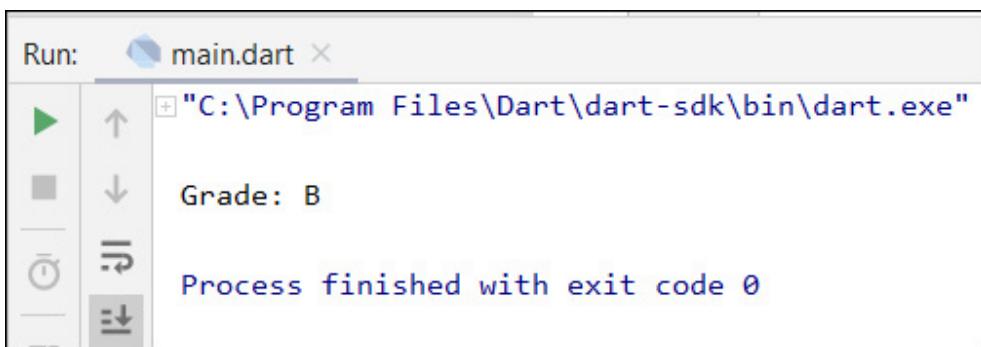
```
main() {
var score=85;

if(score>=90){
  print('Grade: A');
}
else if  (score>=80) {
  print('Grade: B');
}
else if  (score>=70) {
  print('Grade: C');
}

else if  (score>=50) {
  print('Grade: D');
}
else {
  print('Grade: F');
}
```

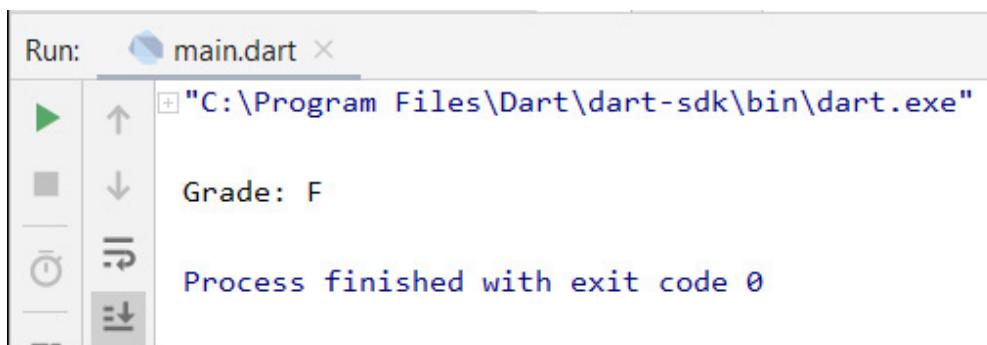
This example includes four "if" statements. Dart compiler follows the code line by line from top to bottom. If any of these "if" statements is true, it will run the action that belongs to it, and then moves directly to run the code which comes after the last else statement. However, if any statement proves to be not true, the program moves to check the next "if" statement.

The run result of this code follows:



```
Run: main.dart
[C:\Program Files\ Dart\dart-sdk\bin\ dart.exe]
Grade: B
Process finished with exit code 0
```

If you change the score value to 40, the program run result will be as follows:



```
Run: main.dart
[C:\Program Files\ Dart\dart-sdk\bin\ dart.exe]
Grade: F
Process finished with exit code 0
```

## IF Else and Logical Operators

Logical operators like **AND** "**&&**" and **OR** "**| |**" may be utilized within **if-else** statements. If you have two conditions (i.e. multiple Boolean expressions) which must be true or one of them is enough to make the statement true respectively, logical operators "if" statement may be used to test more than one condition at the same time.

The following table displays the **|| (OR)** operator results (**true or false**) of the two Boolean expressions. These results will be considered by the "if" statement:

Boolean Expressions (Condition): A	Boolean Expressions (Condition): B	A    B
True	True	True
True	False	True
False	True	True
False	False	False

The following table displays the use of "**&&**" (AND) logical operator to guarantee that the two conditions are true:

Boolean Expressions (Condition): A	Boolean Expressions (Condition): B	A && B
True	True	True
True	False	False
False	True	False
False	False	False

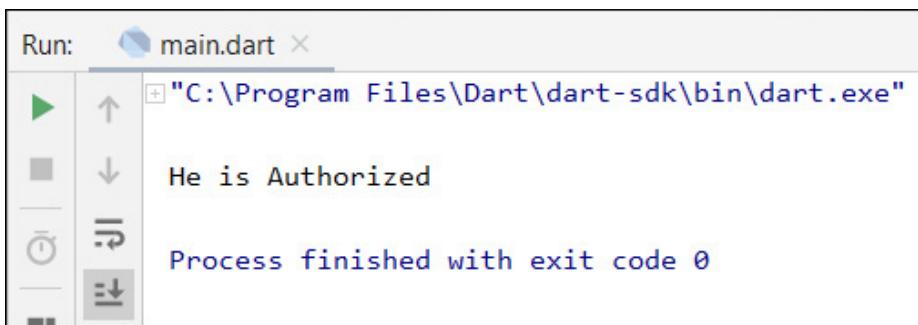
### ***Example:***

The following example includes two conditions. It is enough to prove one of the conditions to consider the "if" condition is true because you used the **||** operator.

```
main() {
  var Age = 16;
  var DOB = 1998;

  if(Age >= 18 || DOB >= 1998){
    print('He is Authorized');
  }
  else {
    print('He is Not Authorized');
  }
}
```

The run result of this code follows:

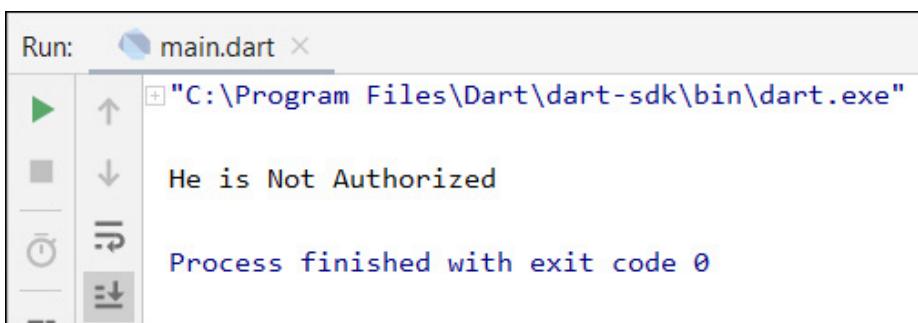


If you replaced the `||` operator with `&&` operator as illustrated in the following code:

```
main() {
  var Age = 16;
  var DOB = 1998;

  if(Age >= 18 && DOB >= 1998){
    print('He is Authorized');
  }
  else {
    print('He is Not Authorized');
  }
}
```

You will get the following run result because the "if" condition will be considered **true** only if the two conditions are true. In this case, if the condition is false, **else** statement will work as shown below:



## For Loops

A **for** statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "**for-loop**" because it repeatedly loops until a particular condition is satisfied.

The following is an example of how the **for-loop** works:

```
main() {  
    // for Loop  
    for (var i = 0; i < 5; i++) {  
        print("i= $i");  
    }  
}
```



(*i++ means  $i=i+1$  (or increment by 1)*

When the program starts, the **i** variable has a value = 0. However, when the Dart compiler runs the code below within the "**for**" loop braces and finishes its loop, the "**i**" variable takes the second value which is 1, and so on until it takes the last value which is 4 here. When the "**i**" variable takes its final value, the Dart compiler will resume its work outside the "**for**" loop braces.

When you run the Dart program, you will get the following result:

The screenshot shows a Dart IDE interface with a "Run" menu open. The selected option is "C:\Program Files\ Dart\dart-sdk\bin\ dart.exe". The output window displays the following text:  
i= 0  
i= 1  
i= 2  
i= 3  
i= 4  
Process finished with exit code 0

## While Loops

A while-statement depends on a Boolean expression and its counter. If the expression proves to be true, the While statement executes the statement(s) in the While block. The while statement continues testing the expression and executing its loop until the expression is proved false.

The following example explains how While loops work:

```
main() {  
  var x=1;  
  
  while (x<=5)  
  {  
    print("x= $x");  
    x++;  
    // this is the counter for the Loop means x=x+1 i.e. increment x by  
    // 1  
    //each time  
  
  }  
}
```

The following steps illustrate the flow of execution of the previous code:

- 1- The initial value of integer variable named x as 1.
- 2- The compiler checks the condition ( $x <= 5$ ) which proves to be true, so it moves into the code block of the **While** statement and prints "x=1".
- 3- The counter variable is incremented to value 1 and the compiler loops to the top.
- 4- The compiler checks again the **While** loop condition ( $x <= 5$ ) which proves to be true, so the compiler prints "x=2" and increments the counter variable to value 2.
- 5- The program will continue working until it reaches  $x=6$ . After that the compiler checks again the **While** loop condition ( $x <= 5$ ) and proves to be false, the compiler will skip the code block and move forward to execute the code of the statements that follows the **While** statement.

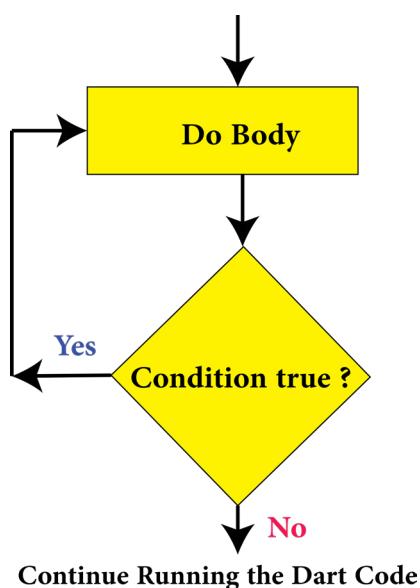
When you run the Dart program, you will get the following result:

```
Run: main.dart
+ "C:\Program Files\ Dart\dart-sdk\bin\dart.exe"
x= 1
x= 2
x= 3
x= 4
x= 5
Process finished with exit code 0
```

## Do-while Loops

Contrary to the while-loop, the **Do-while** loop starts evaluating its expressions from the bottom and not from the top. Thus, the statements within the do block are executed at least once. **Do-while** loops are exit controlled loops, i.e. the compiler will first execute the code block associated with the **Do-while** loop, and then check the associated Boolean expression. If the Boolean expression proves to be true, the compiler will loop and execute the **Do-while** code block again, or else, the compiler will skip the **Do-while** code block and continue to execute the statements that follow the **Do-while** loop.

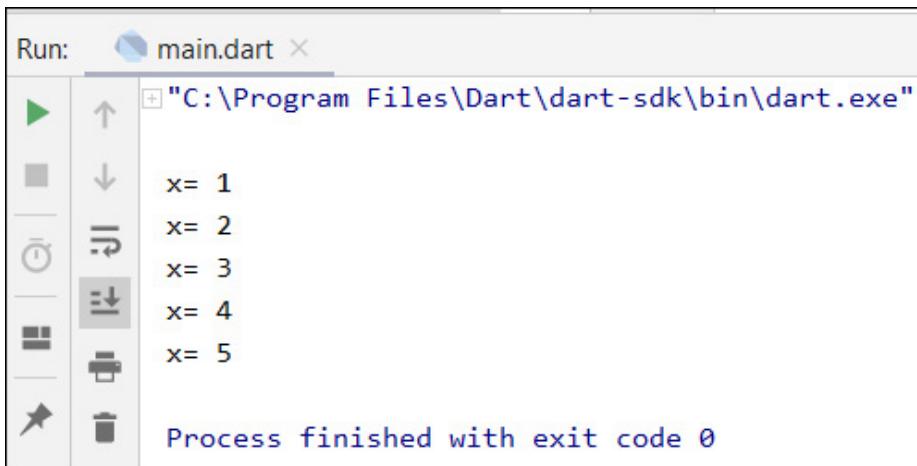
The following figure displays the work flow of the **Do-while** loop:



The following example explains how **Do-while** loops work:

```
main() {  
  var x=1;  
  do {  
    print('x= $x');  
    x++; }  
  
  while (x<=5);  
}
```

When you run the Dart program, you will get the following result:



```
Run: main.dart  
"C:\Program Files\ Dart\dart-sdk\bin\dart.exe"  
x= 1  
x= 2  
x= 3  
x= 4  
x= 5  
Process finished with exit code 0
```

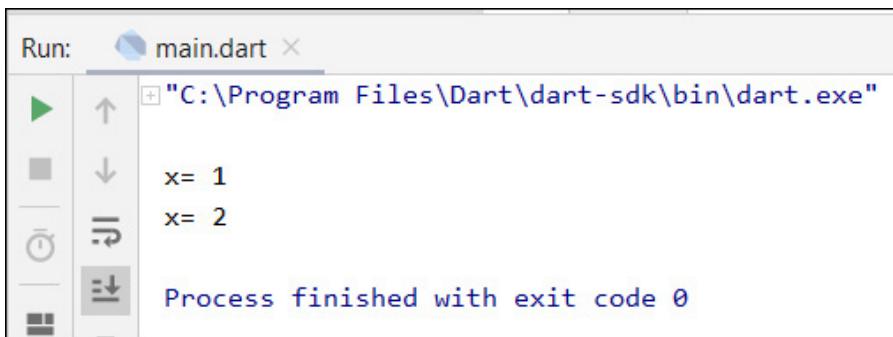
## Break Statement

The **break** statement allows you to exit a loop at any point, and overpasses its normal termination expression. When the break statement is encountered inside a loop, the loop is immediately terminated, and resumes its work starting from the statement that follows the loop. The break statement can be used in any type of loop such as the **While**, **Do-while**, or **For-loops**.

The following code displays the same **Do-while** loops example code used previously. In this example, when the count of variable x becomes equal to 3, the break statement will terminate the **While-loops**. Then, the program control will resume its work starting with the statement following the **While** loops:

```
main() {  
  var x=1;  
  do {  
    print('x= $x');  
    x++;  
    if(x==3) break;  }  
  
  while (x<=5);  
}
```

When you run the Dart program, you will get the following result:



## Switch Case Statement

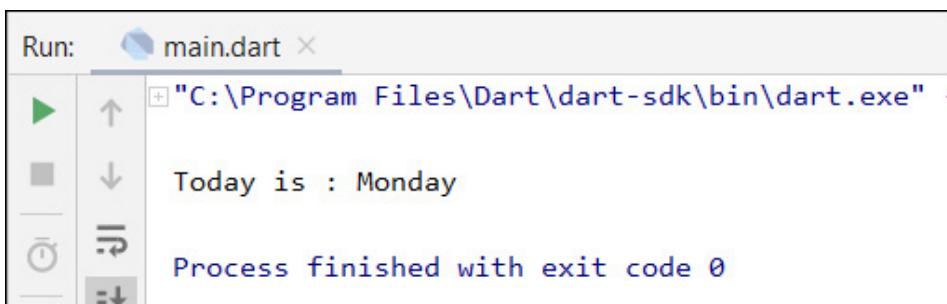
In Dart, switch case statement is a simplified form of the nested **IF-Else** statements. It helps to avoid long chain of **IF...Else** and **Else...IF...** Statements. A switch case statement evaluates an expression against multiple cases in order to identify the block of code to be executed. The switch statement can have a number of possible execution paths.

The following code example, declares an **int** named "day" whose value represents a day. The code displays the name of the day, based on the value of day, using the switch statement. When day=2, case 2 will work, and the **today** variable will be Monday, then the **break** statement will move to outside the switch block of code. In case no statements are running, the **default** statement will be selected.

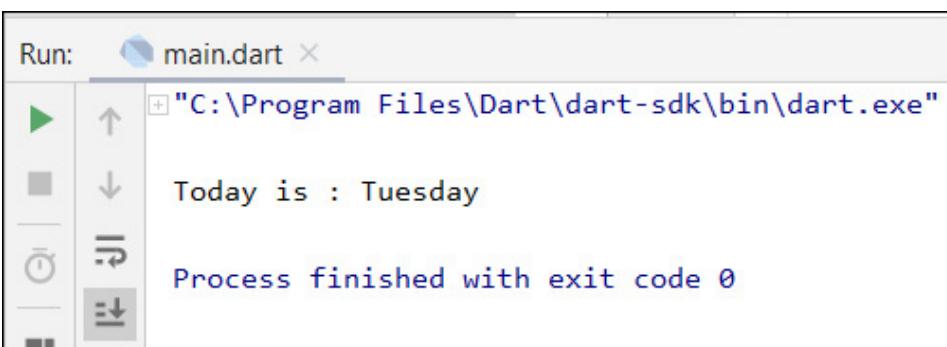
```
main() {  
  int day = 2;  
  String today = null;
```

```
switch (day) {  
  case 1: today = 'Sunday';  
  break;  
  case 2: today = 'Monday';  
  break;  
  case 3: today = 'Tuesday';  
  break;  
  case 4: today = 'Wednesday';  
  break;  
  case 5: today = 'Thursday';  
  break;  
  case 6: today = 'Friday';  
  break;  
  case 7: today = 'Saturday';  
  break;  
 default: today = 'Invalid Day'; }  
print("Today is : $today");  
}
```

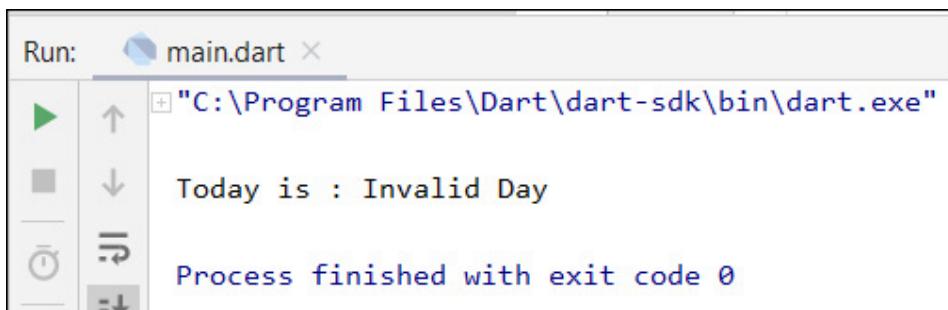
When you run the Dart program, you will get the following result:



When you change "day" variable to 3 , and run the Dart program, you will get the following result:



When you change "day" variable to **9** , the **default** statement will work , and the run result will be as follows:



```
Run: main.dart
+C:\Program Files\ Dart\dart-sdk\bin\dart.exe
Today is : Invalid Day
Process finished with exit code 0
```

## Lab 2: Create a Pizza Order Program

In this lab, you will create a Dart program using simple Dart syntaxes. This lab is just to practice using some code you have learned in lesson 2.

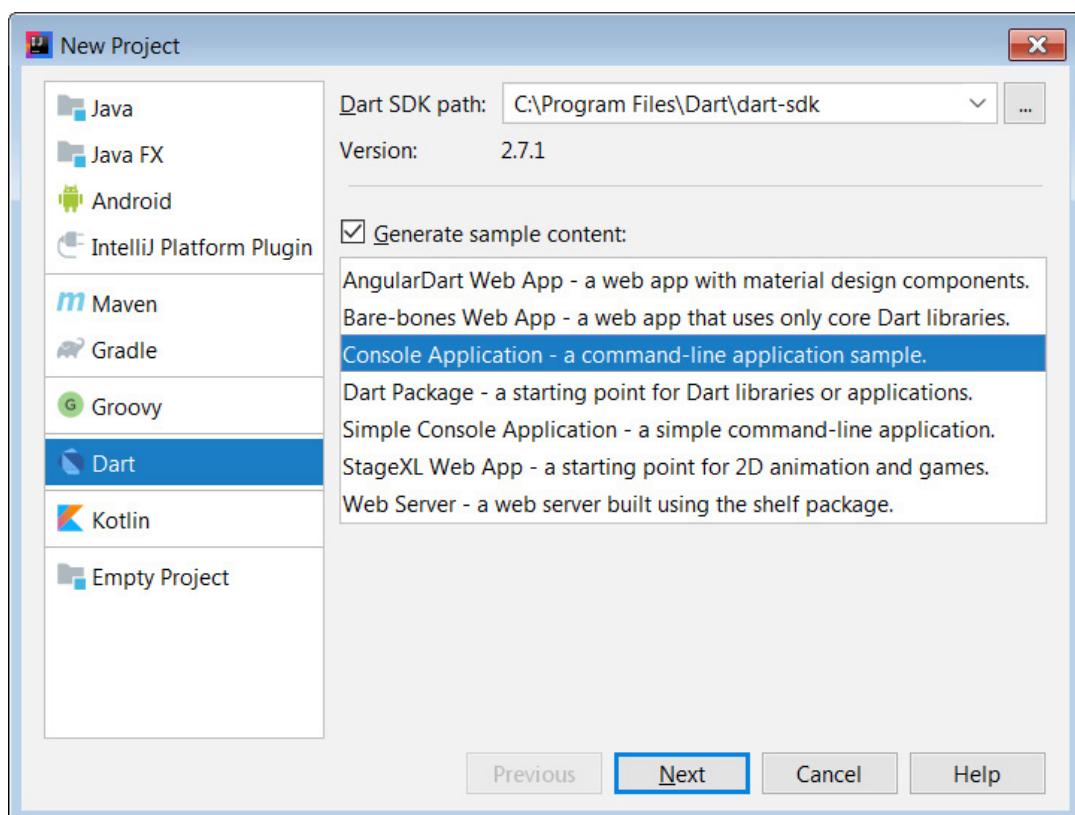
Here, you will create a simple pizza order calculator so that when you run this program, the program will ask the user to enter the pizza size and the quantity of pizza the user wants. Then, the output of the run result will be the total of the user payment.

To create this Dart program, follow the steps below:

1- Open **IntelliJ IDEA**

2- Click **File → New → Project**

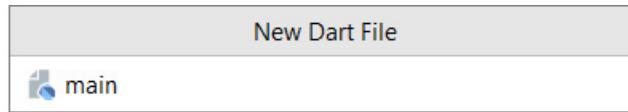
You will get the following dialog box , click **Next**



3- Type the **Project Name : Lab2** , then click **Finish**

4- Right click the project name and select: **New → Dart File**

5- Type the Dart file name "main" as illustrated in the following figure, then press **Enter**:



6- At the beginning of the code (first line), type the following code to enable the input library:

```
import 'dart:io';
```

7- Then, type the following code:

```
import 'dart:io';

main() {
  print('=====');
  print('Pizza Price "Small: 5 USD , Medium : 7 USD ,Large:10
USD"');
  print('Please enter your pizza size (small, medium , or large):');

  String pizza_size = stdin.readLineSync();

  print('How many pizza do you want of $pizza_size');

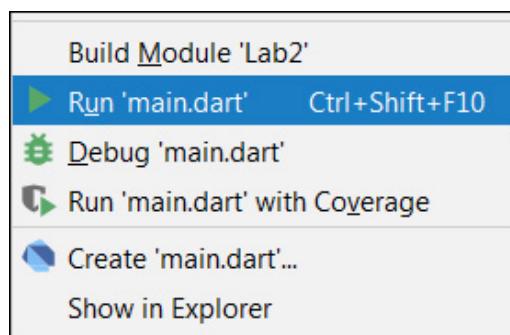
  int Qty = int.parse(stdin.readLineSync());
  var price=null;

  if(pizza_size=="small"){
    price=5;
    var total=price*Qty;
    print("Your Total Payment is : $total");
  }
  else if(pizza_size=="medium") {
    price=7;
    var total=price*Qty;
    print("Your Total Payment is : $total");
  }
  else if(pizza_size=="large") {
    price=10;
    var total=price*Qty;
```

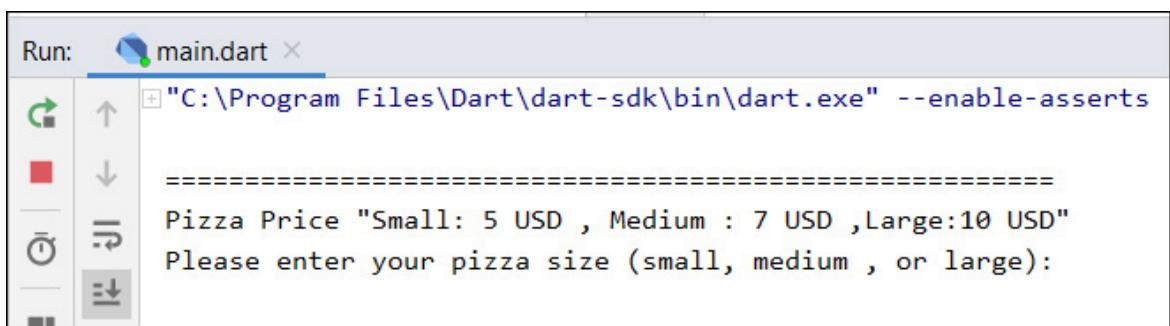
```
        print("Your Total Payment is : $total");
    }

else {(pizza_size==null);
    print('Invalid Pizza Size Input. Please Run This Program Again
!!');
}
}
```

8- Right click **main.dart** file name, then select **Run 'main.dart'** as illustrated in the following figure:



9- You will get the following :



10- Type in the Run console area : **small** and press **Enter** key.

11- You will get the following :

```
Run: main.dart
"C:\Program Files\ Dart\dart-sdk\bin\ dart.exe" --enable-asserts
=====
Pizza Price "Small: 5 USD , Medium : 7 USD ,Large:10 USD"
Please enter your pizza size (small, medium , or large):
small
How many pizza do you want of small
```

12- Type **3** and press **Enter** key. You will get the following result:

```
Run: main.dart
"C:\Program Files\ Dart\dart-sdk\bin\ dart.exe" --enable-asserts
=====
Pizza Price "Small: 5 USD , Medium : 7 USD ,Large:10 USD"
Please enter your pizza size (small, medium , or large):
small
How many pizza do you want of small
3
Your Total Payment is : 15

Process finished with exit code 0
```

13- Run this program again using **medium** or **large** sizes and test your results.

14- If you run this program again using invalid pizza size, such as "**big**", the result will be as follows:

```
Run: main.dart
"C:\Program Files\ Dart\dart-sdk\bin\ dart.exe" --enable-asserts
=====
Pizza Price "Small: 5 USD , Medium : 7 USD ,Large:10 USD"
Please enter your pizza size (small, medium , or large):
big
How many pizza do you want of big
2
Invalid Pizza Size Input. Please Run This Program Again !!

Process finished with exit code 0
```

**Note:** You may create the same program using Switch Case statement.

# Lesson 3: Dart Functions & Object-Oriented Programming (OOP)

<b>Functions.....</b>	3-2
Function Structure .....	3-2
Creating a Function.....	3-2
Function Return Data Types.....	3-4
Void Function .....	3-7
Function Returning Expression .....	3-9
Functions and Variable Scope.....	3-10
<b>Object-Oriented Programming (OOP).....</b>	3-12
Object.....	3-12
Class .....	3-13
Creating a Class .....	3-13
Adding Methods to Classes.....	3-18
Providing Constructors for Your Classes.....	3-19
Class – Getters and Setters .....	3-25
Class Inheritance.....	3-27
Abstract Class .....	3-28
<b>Dart Project Structure and Dart Libraries .....</b>	3-33
<b>Lab 3: Create a Small Overtime Payment Program .....</b>	3-40

## Functions

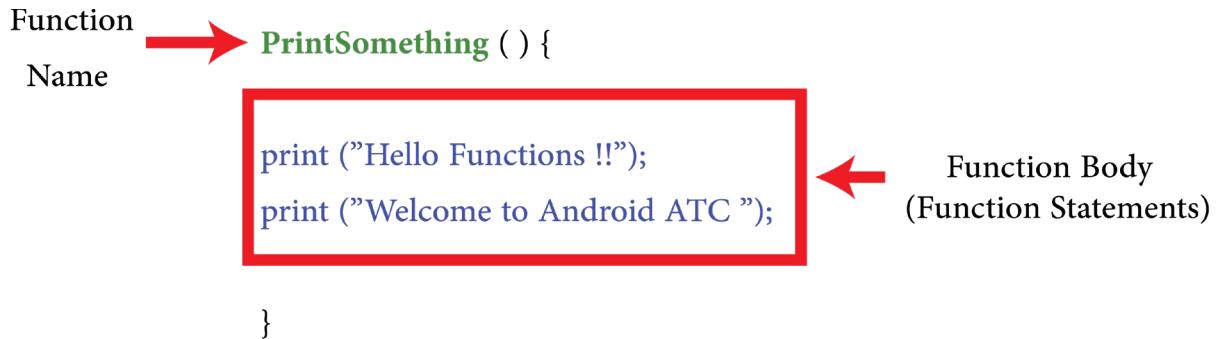
A function in Dart or in any programming language has a specific name and has a set of programming statements. The function can be called at any location of the program to run the statements it has and returns a result, or performs some operations. This process saves time and effort in writing all the function statements one time, then at a certain location of the code call this function by its name to perform a specific procedure or return a value. Also, functions help in organizing the program to structured parts which help in program maintenance and any future modifications.

Each function has a unique name which is used to call it inside the Dart program several times without the need to duplicate statements in multiple source code files.

Most programming languages come with a prewritten set of functions that are kept in a library.

### Function Structure

The following image displays the main components of the main function:



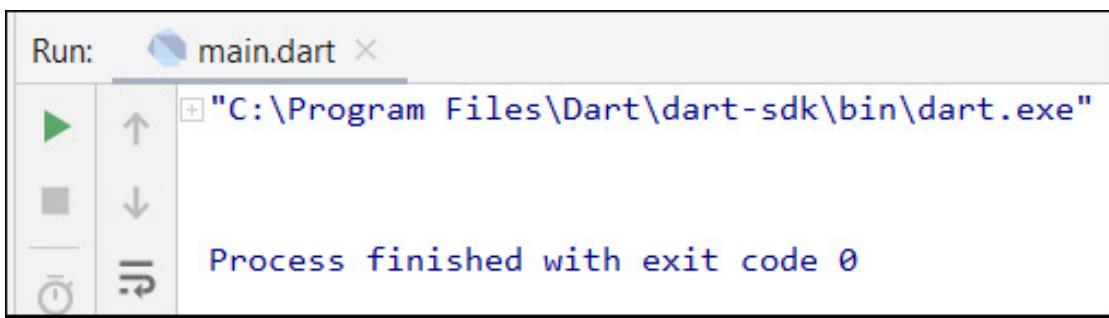
### Creating a Function

If you have a number of code statements that need to be used more than once within your program, you can gather them inside a function which has a specific name and then call this function within the program as many times as needed.

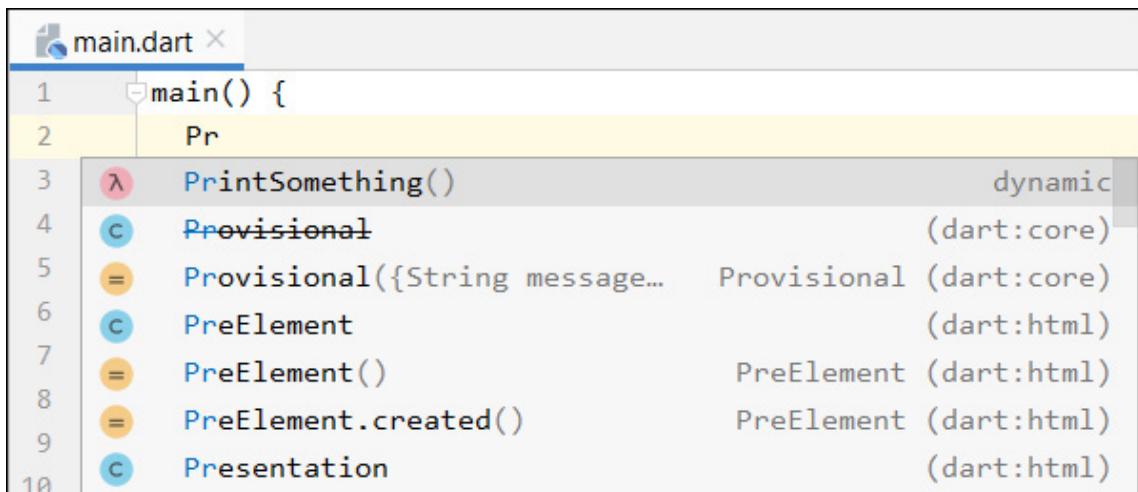
**Example:** In the following example, if you run the following code, the compiler will run the code which is in the main function only.

```
main() {  
}  
  
PrintSomething(){  
    print("Hello Functions !!");  
    print("Welcome to Canada");  
}
```

Since we don't have any code inside the `main()` function, the run output result will be empty as illustrated in the following figure:



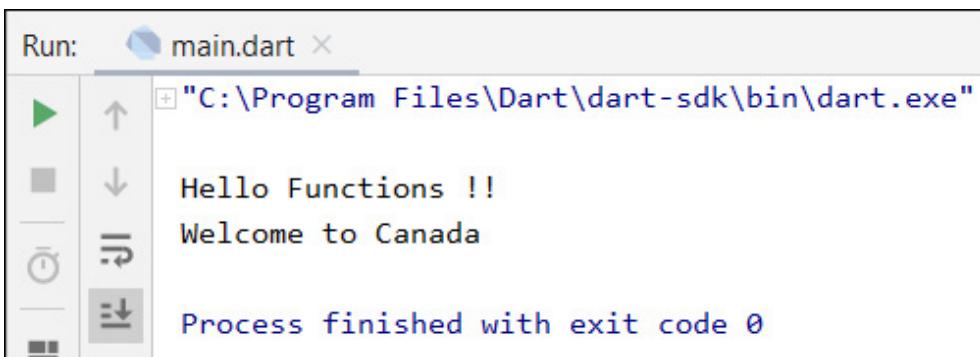
If you call the `PrintSomething()` function inside the `main()` as illustrated in the following figure:



The code will be as follows:

```
main() {  
PrintSomething();  
}  
  
PrintSomething(){  
  print("Hello Functions !!");  
  print("Welcome to Canada");  
}
```

Now the run result will be as follows:



```
Run: main.dart  
+ "C:\Program Files\ Dart\dart-sdk\bin\dart.exe"  
Hello Functions !!  
Welcome to Canada  
Process finished with exit code 0
```

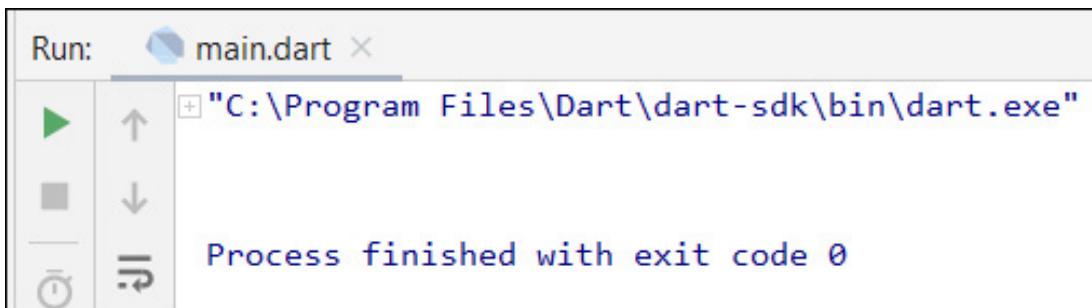
## Function Return Data Types

One purpose of using functions is to return value to the main function. The following example shows how to return value to the main function : `main()`

```
main() {  
  
CourseName();  
  
}  
  
String CourseName(){  
  return "Flutter Application Development";  
}
```

The CourseName() function returns this string sentence : "Flutter Application Development"

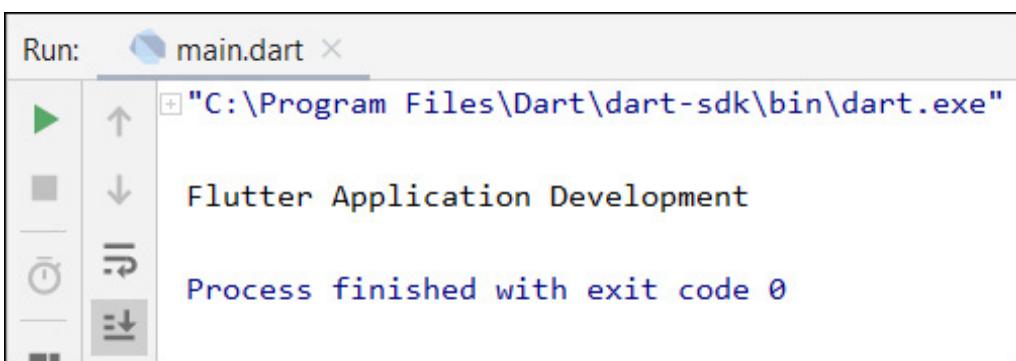
If you run the above code, nothing will happen . The run result will be as follows:



As you see in the next code, we must add a print() function to display the CourseName()function return:

```
main() {  
  print(CourseName());  
}  
  
String CourseName(){  
  return "Flutter Application Development";  
}
```

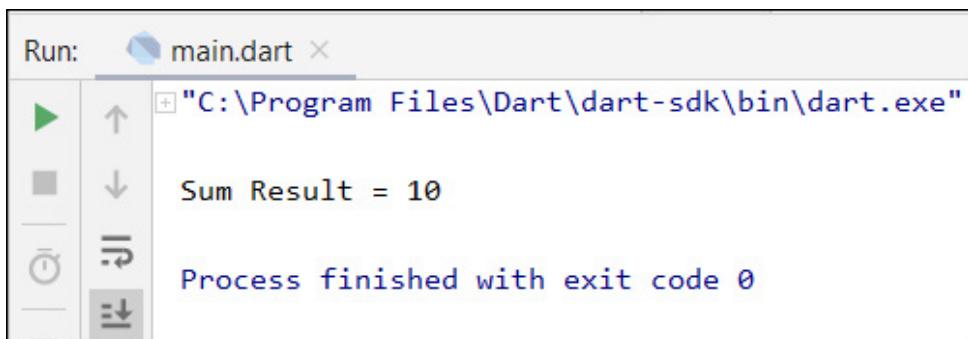
The run output for this code follows:



**Example:** In the following example, the SumCalculator function will return an integer value of the sum of two numbers to the main() function.

```
main() {  
  var result=SumCalculator(3, 7);  
  
  print("Sum Result = $result");  
}  
  
int SumCalculator(x,y){  
  var z=x+y;  
  return z;  
}
```

The run output of this code follows:



The same previous code can be written in the following way and you will get the same run output:

```
main() {  
  
  print("Sum Result = ${SumCalculator(3, 7)}");  
}  
  
int SumCalculator(x,y){  
  var z=x+y;  
  return z;  
}
```

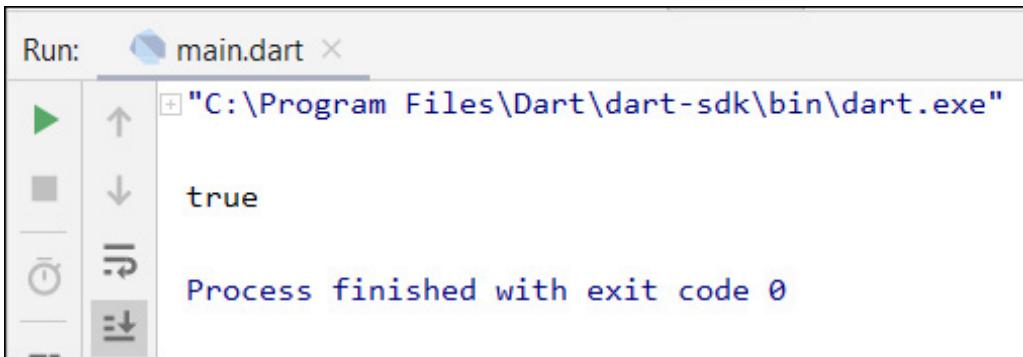
**Note:** In the previous example, it is important to add the `return` command at the end of the `SumCalculator` function. This will return the result of the addition operation, variable `z`, to the location from where the function was called.

**Example:**

In the following code, the `age()` function will return a Boolean value (true or false) depending on the value of age “`x`” :

```
main() {  
  
  print(age(20));  
}  
  
bool age(x){  
  if(x>=18){  
    return true;  
  }  
  else{  
    return false;  
}
```

The run output of this code follows:



```
Run: main.dart ×  
C:\Program Files\ Dart\dart-sdk\bin\dart.exe  
true  
Process finished with exit code 0
```

**Note:** in the previous function `age(x)`, the variable `x` is called function parameter. Also, some functions include more than one parameter; for example, `SumCalculator(x,y)`.

## Void Function

A void function returns values by modifying one or more parameters rather than

using a return statement.

Implicitly, if you don't add anything before the name of the function, then it is a void function.

**Example:**

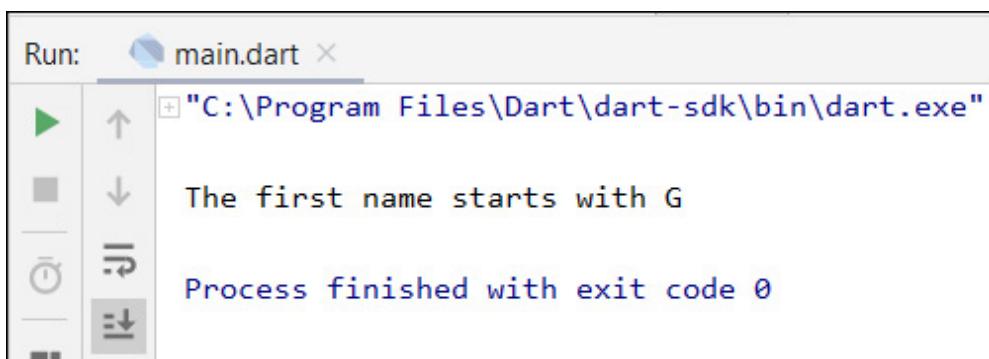
In the following code, the function **MyFunction()** will not return any value. This function will work when you call it inside the **main()** function.

```
main() {
    MyFunction();
}

void MyFunction(){
    var FirstName='George';

    if(FirstName.startsWith('G')){
        print("The first name starts with G");
    }
    else {
        print("The first name does not start with G");
    }
}
```

The run result of this code follows:



```
Run: main.dart ×
C:\Program Files\ Dart\dart-sdk\bin\dart.exe
The first name starts with G
Process finished with exit code 0
```

All functions by default are void functions ; therefore, in the previous example, if you remove the "void" before the function name, you will get the same result.

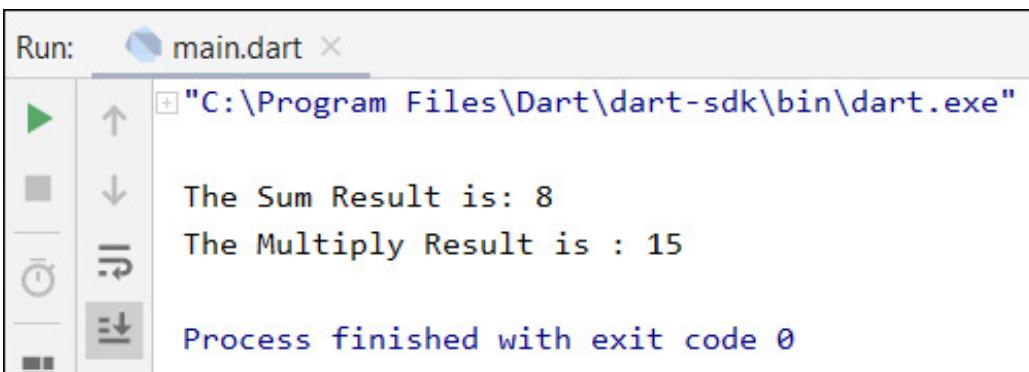
## Function Returning Expression

You can use “=>” operator for returning a value in a brief way.

**Example:** The following code includes a return function:

```
main() {  
  
    print('The Sum Result is: ${sum(3,5)}');  
    print('The Multiply Result is : ${multiply(3,5)}');  
}  
  
sum(x,y){  
    var z=x+y;  
    return z;  
}  
  
multiply(x,y){  
    var w=x*y;  
    return w;  
}
```

The run result follows:

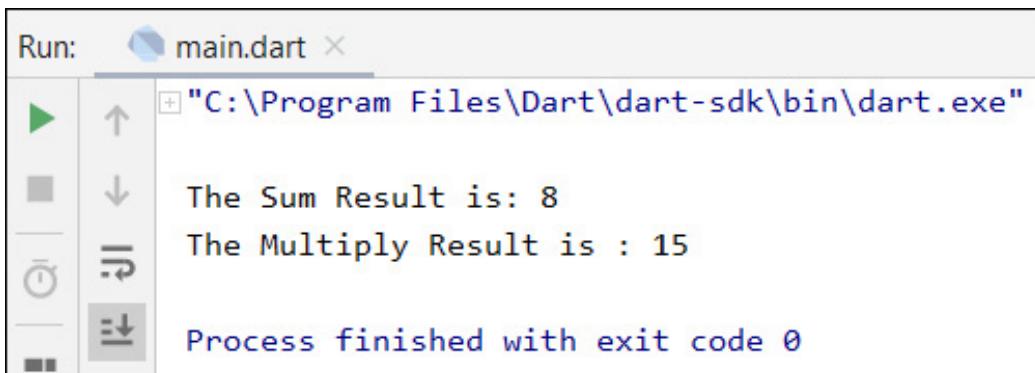


However, if you use the `=>` operator to return the expression "`x+y`" or "`x*y`", you will get the same result but with less number of code lines, as illustrated in the following example:

```
main() {  
    var x=3;  
    var y=5;
```

```
int sum()=>x+y;
int multiply()=>x*y;
print('The Sum Result is : ${sum()}');
print('The Multiply Result is : ${multiply()}');
}
```

The run result follows:



```
Run: main.dart ×
"C:\Program Files\ Dart\dart-sdk\bin\dart.exe"
The Sum Result is: 8
The Multiply Result is : 15
Process finished with exit code 0
```

## Functions and Variable Scope

The location of variables - whether inside or outside the `main()` or other functions - has an important effect on using these variables and the function return value. The following examples display how the program workflow will be affected if we change the location of variables inside or outside the functions.

### Example:

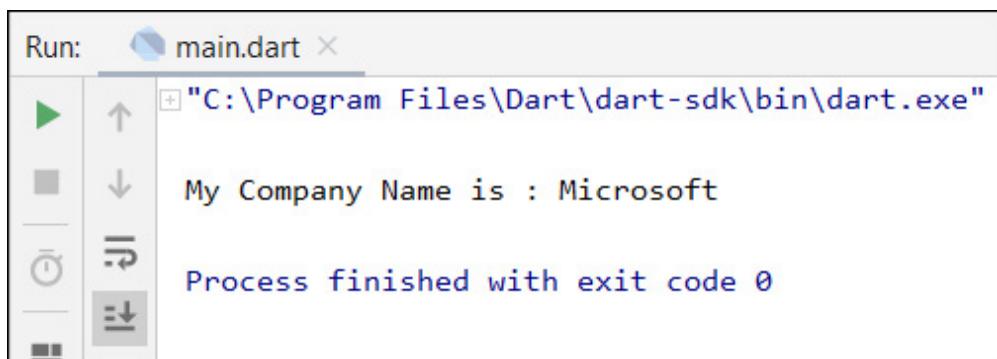
The following Dart code includes a function called `CompanyName()` which is used to print the company name value. In this example, the variable `name` is declared outside any function; therefore, it is called a *global* variable. In this case, it can be used in any function in this Dart program.

```
var name='Microsoft';

main() {
  CompanyName(name);
}

CompanyName(name){
  print('My Company Name is : $name');
}
```

The run result of this Dart program follows:



A screenshot of a Dart IDE showing the run output for a file named 'main.dart'. The output window title is 'Run: main.dart'. It shows the command 'C:\Program Files\ Dart\dart-sdk\bin\dart.exe' and the output text 'My Company Name is : Microsoft' followed by 'Process finished with exit code 0'. The IDE interface includes a toolbar with icons for play, stop, and refresh.

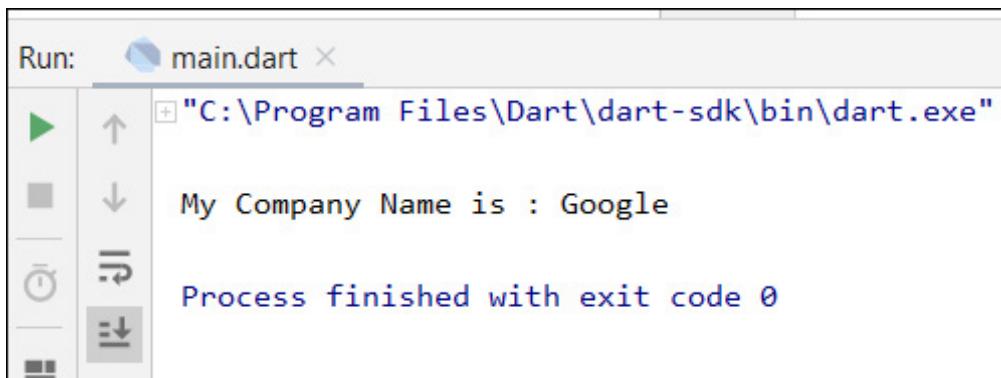
Now, as illustrated in the next code, we will configure the *name* variable inside the *CompanyName(name)* function . This means that *name* variable is considered as a *local* variable for *CompanyName(name)* function.

```
var name='Microsoft';

main() {
  CompanyName(name);
}

CompanyName(name){
  var name='Google';
  print('My Company Name is : $name');
}
```

Since the local variable will be dominant over any other configurations that may come from outside this function, the run result of this code will be follows:



A screenshot of a Dart IDE showing the run output for a file named 'main.dart'. The output window title is 'Run: main.dart'. It shows the command 'C:\Program Files\ Dart\dart-sdk\bin\dart.exe' and the output text 'My Company Name is : Google' followed by 'Process finished with exit code 0'. The IDE interface includes a toolbar with icons for play, stop, and refresh.

Also, the same thing applies in the following code. The local variable will take effect over the global variable:

```
var name='Microsoft';

main() {
  CompanyName('Android ATC');
}

CompanyName(name){
  print('My Company Name is : $name');
}
```

The run result of this code follows:

```
Run: main.dart ×
C:\Program Files\ Dart\dart-sdk\bin\dart.exe
My Company Name is : Android ATC
Process finished with exit code 0
```

## Object-Oriented Programming (OOP)

If you have never used an object-oriented programming language before, you'll need to learn the basic concepts before you can start coding. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Dart programming language. Learning the basics of OOP is necessary before starting any Flutter application development.

### Object

Objects are the key to understanding object-oriented technology. Look around you right now and you will find many examples of real-world objects: your car, your desk, your computer, your house etc.

Real-world objects share two characteristics: state and behavior. Computers have a state (type, color, speed, capacity) and behavior (processing, playing media, browsing). A car also has a state (current gear, current pedal cadence, and current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

## Class

A class is a blueprint or prototype (template) from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can clearly model state and behavior. Classes are characterized by properties and functions.

For example, if the class is a car, it may have the following properties:

- Price
- Type
- Maximum speed
- Number of seats

And it may have the following functions:

- NumberOfSeats
- NumberOfWheels
- getYearBuilt

Class	Object
Blueprint 	Van  Audi  Sports Car 

## Creating a Class

Use the **class** keyword to declare a class in Dart. A class definition starts with the keyword **class** followed by the **class name**; and the class body enclosed by a pair of curly braces.

A class code includes the class name and a group of properties or attributes which will construct the class.

### Example:

The following example shows how you can create a class called **car** step by step:

**Step 1:** Declare an empty class called “**car**”, as illustrated in the following code:

```
class car {  
}  
  
main() {  
}
```

**Step 2:** The car class has been created. Now, you may add some properties to this car class as illustrated in the following code:

```
class car {  
    String type;  
    String color;  
    int MaxSpeed;  
    int NumOfSeats;  
}  
  
main() {  
}
```

**Step 3:** To create an instance of the class or object, use the **new** keyword followed by the class name. Here, you can create an object depending on the “**car**” class (blue print) which you have created in step 1, using the following code. **toyota** is the name of the object which has all car class properties or attributes values.

```
class car {  
    String type;  
    String color;  
    int MaxSpeed;  
    int NumOfSeats;  
}  
  
main() {  
    var toyota=new car();  
}
```

As you see in the following image, the **toyota** object can use all the **car** class properties:

The screenshot shows a code editor window with the file 'main.dart' open. The code defines a 'car' class with properties: type (String), color (String), MaxSpeed (int), and NumOfSeats (int). In the 'main()' function, a variable 'toyota' is created as a new car object. A tooltip is displayed over the 'toyota.' part of the code, listing the properties and methods of the 'car' class. The properties listed are: color (String), hashCode (int), MaxSpeed (int), noSuchMethod (Invocation invocation) (dynamic), NumOfSeats (int), runtimeType (Type), toString() (String), type (String), nn (if (expr != null) {}), notnull (if (expr != null) {}), null (if (expr == null) {}), and ... (with a tooltip 'More ...'). The tooltip also includes a note: 'Ctrl+Down and Ctrl+Up will move caret down and up in the editor' and a link 'Next Tip'.

The following code displays the object **toyota** with some properties values:

```
class car {  
  String type;  
  String color;  
  int MaxSpeed;  
  int NumOfSeats;  
}  
  
main() {  
  var toyota=new car();  
  toyota.MaxSpeed=200;  
  toyota.color="red";  
  toyota.NumOfSeats=5;  
  toyota.type="Camry";  
}
```

For example, if you add the following command to the `main()` function, you can print a specific attribute or property value (type) of `toyota` object :

```
print("Car Type is: ${toyota.type}");
```

The run result of this code follows:

```
Run: main.dart
"C:\Program Files\ Dart\dart-sdk\bin\ dart.exe"
Car Type is: Camry
Process finished with exit code 0
```

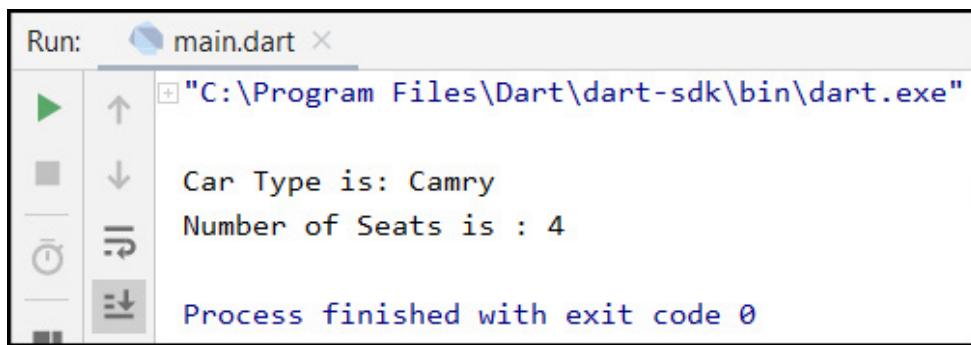
Also, you may assign a default value for these class properties such as `NumOfSeats` as illustrated in the following grey highlighted part of the following code:

```
class car {
  String type;
  String color;
  int MaxSpeed;
  int NumOfSeats = 4;
}

main() {
  var toyota=new car();
  toyota.MaxSpeed=200;
  toyota.color="red";
  toyota.type="Camry";

  print("Car Type is: ${toyota.type}");
  print("Number of Seats is : ${toyota.NumOfSeats}");
}
```

The run result of this program follows:



```
Run: main.dart
C:\Program Files\ Dart\dart-sdk\bin\dart.exe
Car Type is: Camry
Number of Seats is : 4
Process finished with exit code 0
```

However, if you configure a specific value for `NumOfSeats` to the object “`toyota`”, it will override any other values as illustrated in the following code:

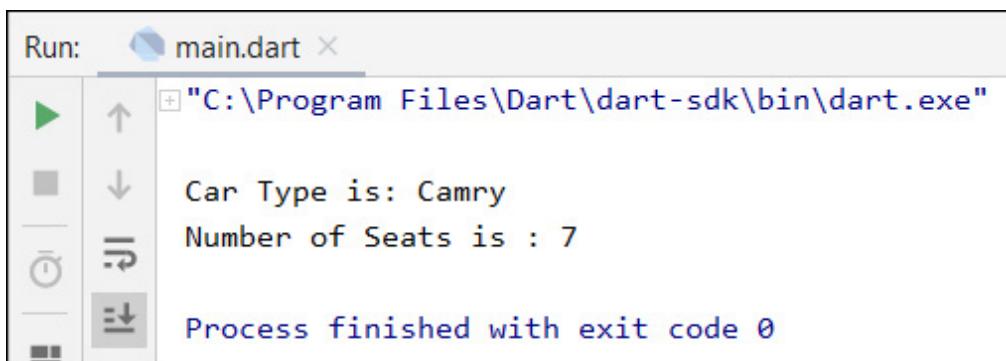
```
class car {
  String type;
  String color;
  int MaxSpeed;
  int NumOfSeats = 4;
}

main() {
  var toyota=new car();
  toyota.MaxSpeed=200;
  toyota.color="red";
  toyota.type="Camry";
  toyota.NumOfSeats=7;

  print("Car Type is: ${toyota.type}");
  print("Number of Seats is : ${toyota.NumOfSeats}");

}
```

The run output will be as follows:



```
Run: main.dart
C:\Program Files\ Dart\dart-sdk\bin\dart.exe
Car Type is: Camry
Number of Seats is : 7
Process finished with exit code 0
```

## Adding Methods to Classes

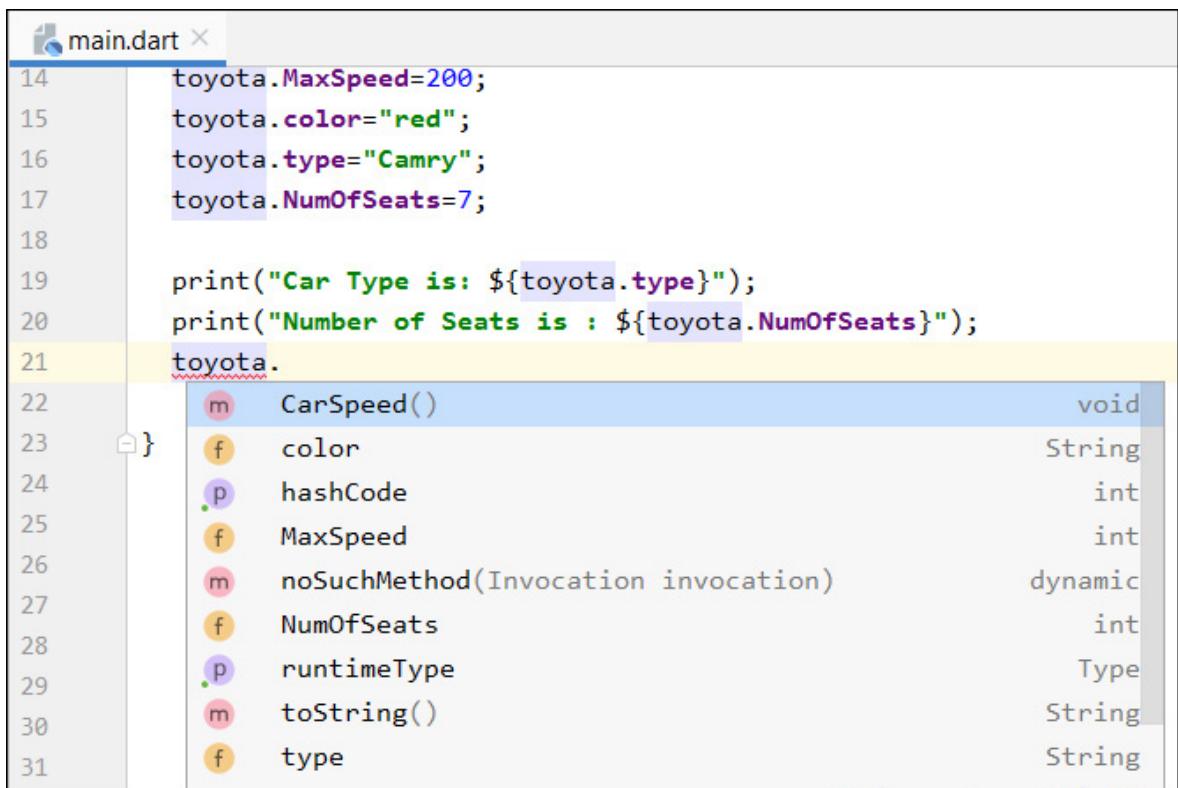
You can add your methods or functions to a class, then conjunct these methods with the objects which you want to initiate from this class.

In the following example, **CarSpeed()** method has been added to the car class as illustrated in the grey highlighted part of the following code:

```
class car {
    String type;
    String color;
    int MaxSpeed;
    int NumOfSeats = 4;

    void CarSpeed(){
        print("Car Speed is : $MaxSpeed");
    }
}
```

As illustrated in the following figure, you can use this **CarSpeed()** method with the **toyota** object:

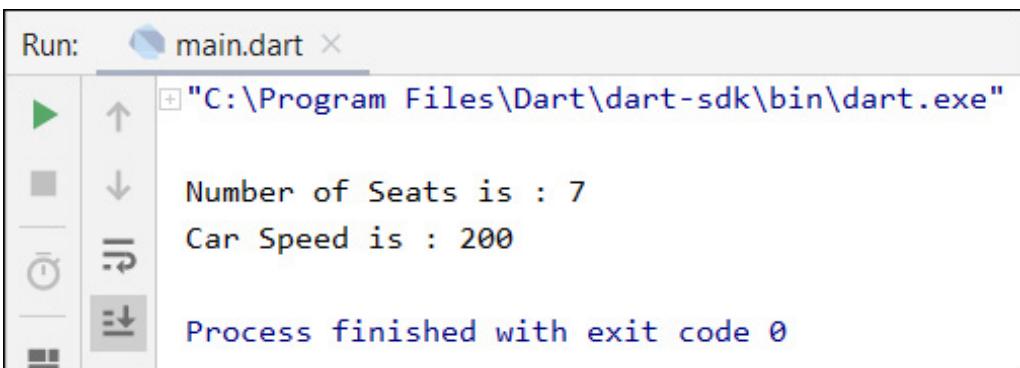


```
main.dart
14     toyota.MaxSpeed=200;
15     toyota.color="red";
16     toyota.type="Camry";
17     toyota.NumOfSeats=7;
18
19     print("Car Type is: ${toyota.type}");
20     print("Number of Seats is : ${toyota.NumOfSeats}");
21     toyota.
22     m  CarSpeed()                      void
23     f  color                           String
24     p  hashCode                      int
25     f  MaxSpeed                      int
26     m  noSuchMethod(Invocation invocation) dynamic
27     f  NumOfSeats                     int
28     p  runtimeType                   Type
29     m  toString()                    String
30     f  type                           String
31 }
```

The full code will be as follows:

```
class car {  
    String type;  
    String color;  
    int MaxSpeed;  
    int NumOfSeats = 4;  
  
    void CarSpeed(){  
        print("Car Speed is : $MaxSpeed");  
    }  
}  
  
main() {  
    var toyota=new car();  
    toyota.MaxSpeed=200;  
    toyota.color="red";  
    toyota.type="Camry";  
    toyota.NumOfSeats=7;  
  
    print("Number of Seats is : ${toyota.NumOfSeats}");  
    toyota.CarSpeed();  
}
```

The run output of this code follows:



The screenshot shows the Dart IDE's run interface. The 'Run' tab is selected, and the file 'main.dart' is chosen. The terminal window displays the following output:  
"C:\Program Files\ Dart\dart-sdk\bin\dart.exe"  
Number of Seats is : 7  
Car Speed is : 200  
Process finished with exit code 0

## Providing Constructors for Your Classes

Constructor is a block of code that initializes the newly created object. A constructor resembles an instance method in Dart but it's not considered a method

because it doesn't have a return type.

Dart defines a constructor with the same name as that of the class.

**Example:**

In the previous example, you initialized **toyota** object from the **car** class using the following command:

```
var toyota= new car();
```

Now, you will construct the same object using a constructor which has the same class name.

First, I will create the **car** class again as illustrated in the following code:

```
class car {  
    String type;  
    String color;  
    int MaxSpeed;  
    int NumOfSeats;  
  
}  
  
main() {  
  
}
```

Now, add the following highlighted grey color code which represents the **car** constructor which has the same class name.

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

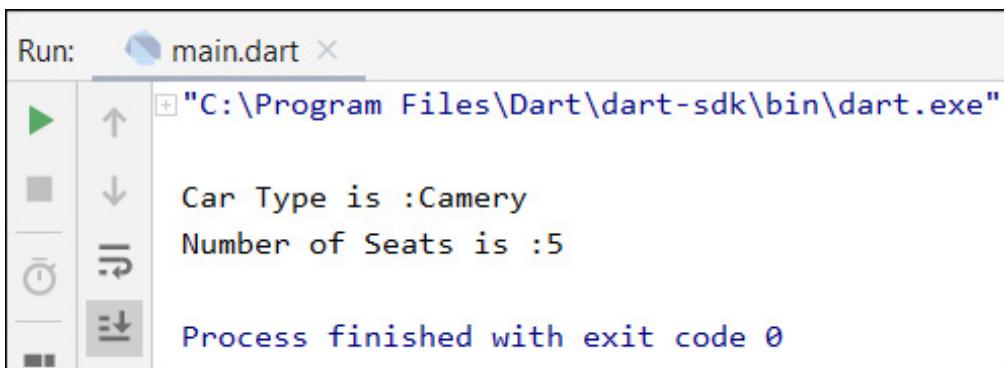
```
class car {  
    String type;  
    String color;  
    int MaxSpeed;  
    int NumOfSeats;
```

```
car(type, color, MaxSpeed, NumOfSeats){  
    this.type=type;  
    this.color=color;  
    this.MaxSpeed=MaxSpeed;  
    this.NumOfSeats=NumOfSeats;  
}  
  
}  
  
main() {  
  
}
```

Now, you can construct or initiate an object using the **car** constructor as illustrated in the grey highlighted color part of the following code:

```
class car {  
String type;  
String color;  
int MaxSpeed;  
int NumOfSeats;  
  
car(type, color, MaxSpeed, NumOfSeats){  
    this.type=type;  
    this.color=color;  
    this.MaxSpeed=MaxSpeed;  
    this.NumOfSeats=NumOfSeats;  
}  
  
}  
  
main() {  
  
    var toyota=new car("Camery","red",200,5);  
  
    print("Car Type is :${toyota.type}");  
    print("Number of Seats is :${toyota.NumOfSeats}");  
}
```

The run output of this code follows:



Also, in the previous example, you can create the previous `car` constructor code in another way using the least number of lines of code by replacing the following part of a code:

```
car(type, color, MaxSpeed, NumOfSeats){
    this.type=type;
    this.color=color;
    this.MaxSpeed=MaxSpeed;
    this.NumOfSeats=NumOfSeats;
}
```

with the following:

```
car(this.type, this.color , this.MaxSpeed , this.NumOfSeats);
```

The full code will be as follows:

```
class car {
    String type;
    String color;
    int MaxSpeed;
    int NumOfSeats;

    car(this.type, this.color , this.MaxSpeed , this.NumOfSeats);
}

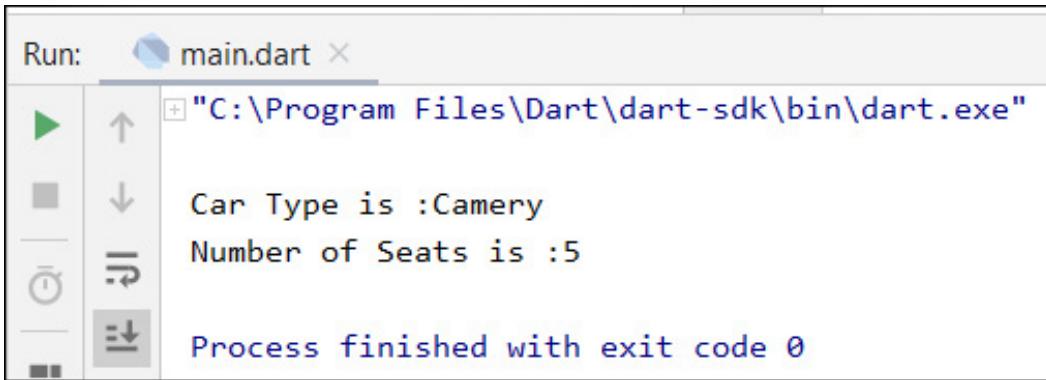
main() {
```

```
var toyota=new car("Camery","red",200,5);

print("Car Type is :${toyota.type}");
print("Number of Seats is :${toyota.NumOfSeats}");

}
```

The run output will be the same as the following:



```
Run: main.dart ×
C:\Program Files\ Dart\dart-sdk\bin\dart.exe
Car Type is :Camery
Number of Seats is :5
Process finished with exit code 0
```

**Example:**

Also, you can initialize an object using a constructor name. The following code creates the **car** class, and then using the constructor **car.initialize()**, you can assign the attributes values as follows:

```
class car {
  String type;
  String color;
  int MaxSpeed;
  int NumOfSeats;

  car.initialize(){
    type="Mini Van";
    color="Green";
    MaxSpeed=230;
    NumOfSeats=2;
  }
}

main() {
```

Then, as you see in the following figure, you can use the constructor name in creating the **toyota** object.

```

15 main() {
16     var toyota=new car.
17
18 }
19
20

```

The screenshot shows a code editor with the following code snippet:

```

15 main() {
16     var toyota=new car.
17
18 }
19
20

```

A code completion dropdown is open at line 16, showing the following options:

- initialize()
- try
- tryon

Below the dropdown, there is a note: "Ctrl+Down and Ctrl+Up will move caret down and up in the editor" and a link "Next Tip".

The full code follows:

```

class car {
    String type;
    String color;
    int MaxSpeed;
    int NumOfSeats;

    car.initialize(){
        type="Mini Van";
        color="Green";
        MaxSpeed=230;
        NumOfSeats=2;
    }
}

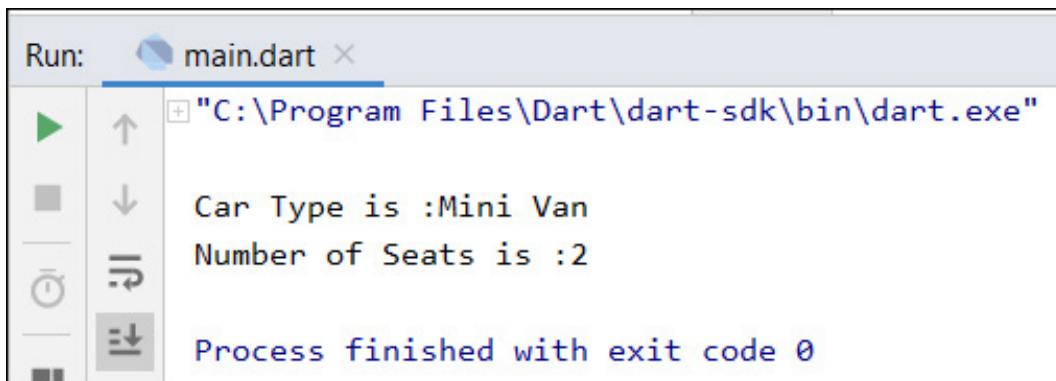
main() {

    var toyota=new car.initialize();

    print("Car Type is :${toyota.type}");
    print("Number of Seats is :${toyota.NumOfSeats}");
}

```

The run output follows:



```
Run: main.dart ×
C:\Program Files\ Dart\dart-sdk\bin\dart.exe
Car Type is :Mini Van
Number of Seats is :2
Process finished with exit code 0
```

## Class – Getters and Setters

Getters and Setters are used to effectively protect your data, particularly when creating classes. For each instance variable, a getter method returns its value while a setter method sets or updates its value.

Getters are defined using the **get** keyword, whereas setters are defined using the **set** keyword.

### **Example:**

In the following example, we will use **set** keyword to set a string value for the **type** class attribute , and **get** keyword to return this class attribute value.

```
class car {
  String type;
  String color;
  int MaxSpeed;
  int NumOfSeats;

  set setType(String value)=>type =value; //setter
  String get getType => type; //getter

}

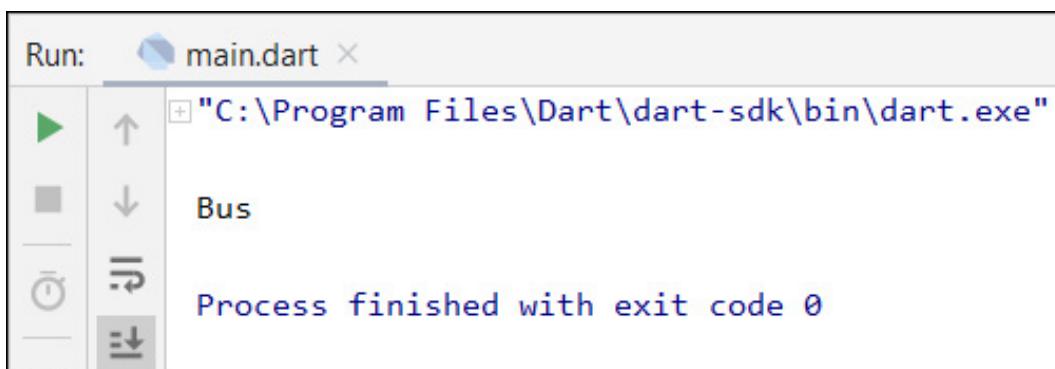
main() {
```

**Note :** This “=>” operator is for returning a value in a brief way.

Now, in the following code, we will use **setType** to set a value and **getType** to return or use this value.

```
class car {  
    String type;  
    String color;  
    int MaxSpeed;  
    int NumOfSeats;  
  
    set setType(String value)=>type =value; //setter  
    String get getType => type; //getter  
}  
  
main() {  
    var toyota= new car();  
    toyota.setType="Bus";  
    print(toyota.getType);  
}
```

The run output follows:



## Class Inheritance

The idea of inheritance is simple but powerful. When you want to create a new class and there is already a class that includes some of the codes that you need, you can derive your new class from the existing class (called parent or super class). While doing this, you can reuse the properties and methods of the existing class without having to write them again.

### **Example:**

Assume that you have a class called *Student*, as illustrated in the following code:

```
class Student {  
    String Name;  
    String Address;  
    int TelNumber;  
    String Email; }  
  
main() {  
}
```

If you want to create a new class called *Teacher*, with the same properties of the previous class *Student*, which is the primary class (parent or super class), you can add the following line to the previous code snippet.

```
class Teacher extends Student{}
```

Now, *Teacher* class has all the properties or attributes of *Student* class.

You may use this *Teacher* class to create a new object “*trainer*” as illustrated in the following code:

```
class Student {  
    String Name;  
    String Address;  
    int TelNumber;  
    String Email;
```

```

}

class Teacher extends Student{}

main() {
  var trainer=new Teacher();

  trainer.Email="test@gmail.com";
  trainer.TelNumber=123456789;
  print("Email Address : ${trainer.Email}");
  print("Telephone Number: ${trainer.TelNumber}");
}

```

The run output of this code follows:

```

Run: main.dart
+ "C:\Program Files\ Dart\dart-sdk\bin\dart.exe"
Email Address : test@gmail.com
Telephone Number: 123456789
Process finished with exit code 0

```

## Abstract Class

A class is called abstract when at least one of its methods or functions does not have an implementation (function body). It is the responsibility of the derived / child class to provide the implementation of the abstract members of the parent class. An abstract class needs to be qualified with the `abstract` keyword. An abstract class cannot be instantiated, and its only purpose is to be inherited.

### **Example:**

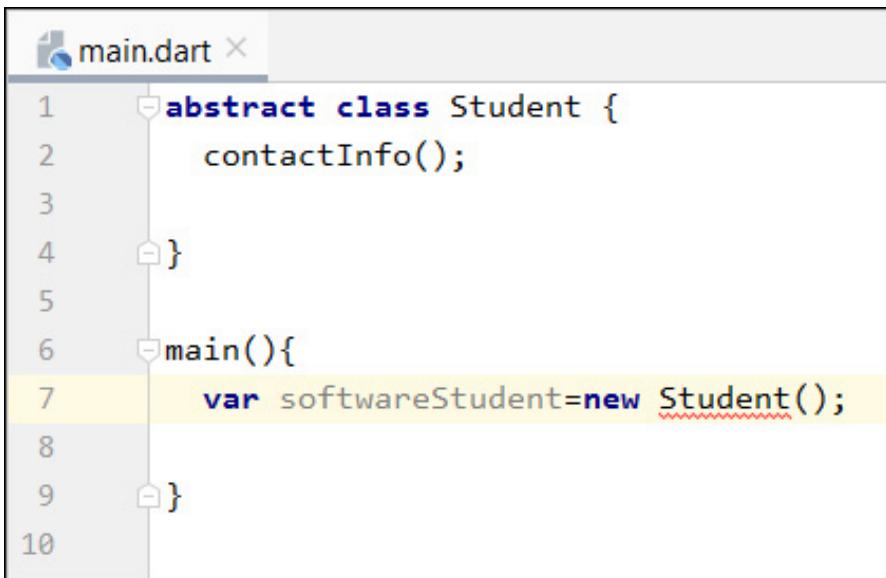
In the following example, `Student` is a normal class. If you added the `contactInfo` method to this class without a method body , you will get an error as illustrated in the following code:

```
class Student {  
    contactInfo();  
}  
  
main(){  
}
```

But if you configure Student class as abstract class, as illustrated in the following code, you can add the **contactInfo** (abstract method) without method body (implementation).

```
abstract class Student {  
    contactInfo();  
}  
  
main(){  
}
```

Also, you can't create an object directly using this abstract class. If you try, you will get an error (red underline) as illustrated in the following screenshot:



The screenshot shows a code editor window titled "main.dart". The code is as follows:

```
1  abstract class Student {  
2      contactInfo();  
3  
4  }  
5  
6  main(){  
7      var softwareStudent=new Student();  
8  
9  }
```

The word "Student" in the line "var softwareStudent=new Student();" is underlined in red, indicating a syntax error.

Instead of that , you should create a child class that inherits these abstract class (Student) properties and abstract methods, then use this child class in creating new objects.

In the following code, **ITStudent** is a child class, and will inherit all the **Student** (parent class) properties and methods. When you type the following code , you will see a red underline below **ITStudent** because the abstract class **ITStudent** includes an empty abstract method called **contactInfo**.

```
abstract class Student {
    contactInfo();

}

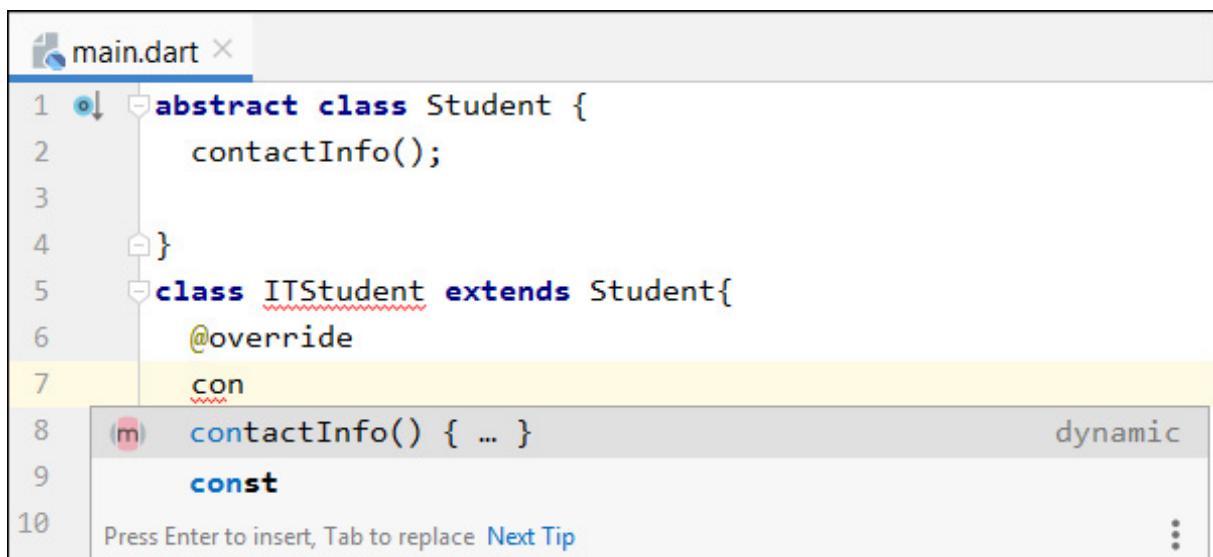
class ITStudent extends Student{

}

main(){

}
```

Now, you must add the abstract method: **contactInfo** to the child class body. When you try to do that, you will get the following menu as illustrated in the following screenshot:



```
main.dart
1  abstract class Student {
2      contactInfo();
3
4  }
5  class ITStudent extends Student{
6      @override
7      con
8      (m) contactInfo() { ... }           dynamic
9      const
10
11 Press Enter to insert, Tab to replace Next Tip
```

When you add the **contactInfo** method, the error will be removed and you will get the following code:

```
abstract class Student {  
    contactInfo();  
  
}  
class ITStudent extends Student{  
    @override  
    contactInfo() {  
        // TODO: implement contactInfo  
        return null;  
    }  
}  
main(){  
}
```

Now, add some properties (method body) to the **contactInfo()** method and remove "return null;" as illustrated in the following code:

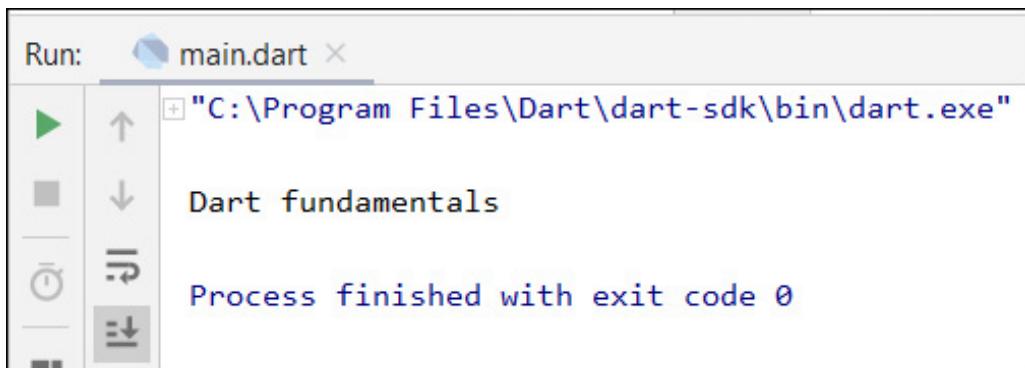
```
class ITStudent extends Student{  
    @override  
    contactInfo() {  
        // TODO: implement contactInfo  
        print("Dart fundamentals");  
    }  
}
```

You can create the **SoftwareStudent** object from the **ITStudent** class as illustrated in the following code:

```
abstract class Student {  
    contactInfo();  
  
}  
class ITStudent extends Student{  
    @override  
    contactInfo() {  
        // TODO: implement contactInfo  
        print("Dart fundamentals");  
    }  
}
```

```
main(){
  var SoftwareStudent= new ITStudent();
  SoftwareStudent.contactInfo();
}
```

The run output of this code follows:



Although this abstract class has abstract methods, it can include normal methods also. For example, you can add **CourseInfo** method with a method body to your abstract class: **Student** as illustrated in the following code :

```
abstract class Student {
  contactInfo();
  CourseInfo(){
    print("Flutter Application Development Course");
  }
}

class ITStudent extends Student{
  @override
  contactInfo() {
    // TODO: implement contactInfo
    print("Dart fundamentals");
  }
}

main(){
  var SoftwareStudent= new ITStudent();
  SoftwareStudent.contactInfo();
  SoftwareStudent.CourseInfo();
}
```

The run output of this code follows:

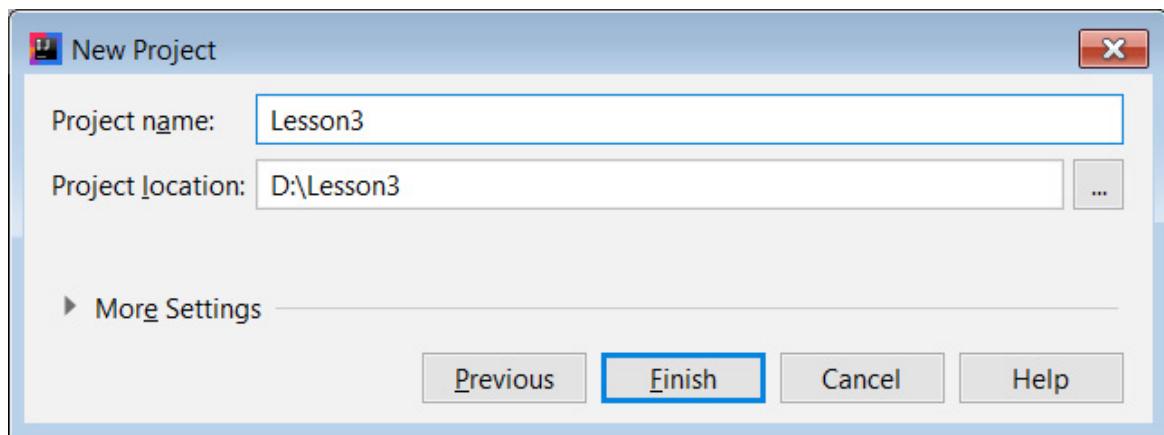
```
Run: main.dart
"C:\Program Files\ Dart\dart-sdk\bin\dart.exe"

Dart fundamentals
Flutter Application Development Course

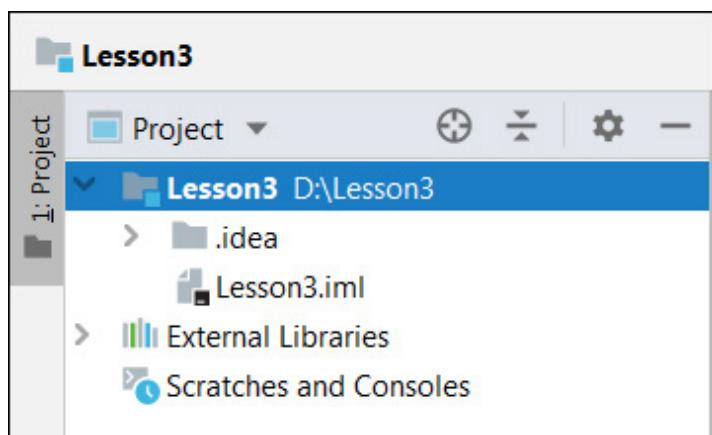
Process finished with exit code 0
```

## Dart Project Structure and Dart Libraries

When you create a new Dart project using IntelliJ IDEA or any other IDE, you will get a dialog box similar to the following :

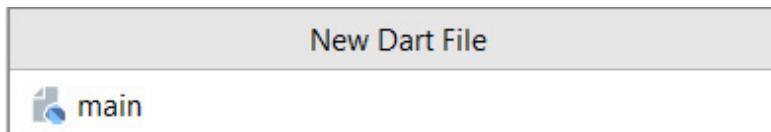


When you click **Finish**, you will get the following window:

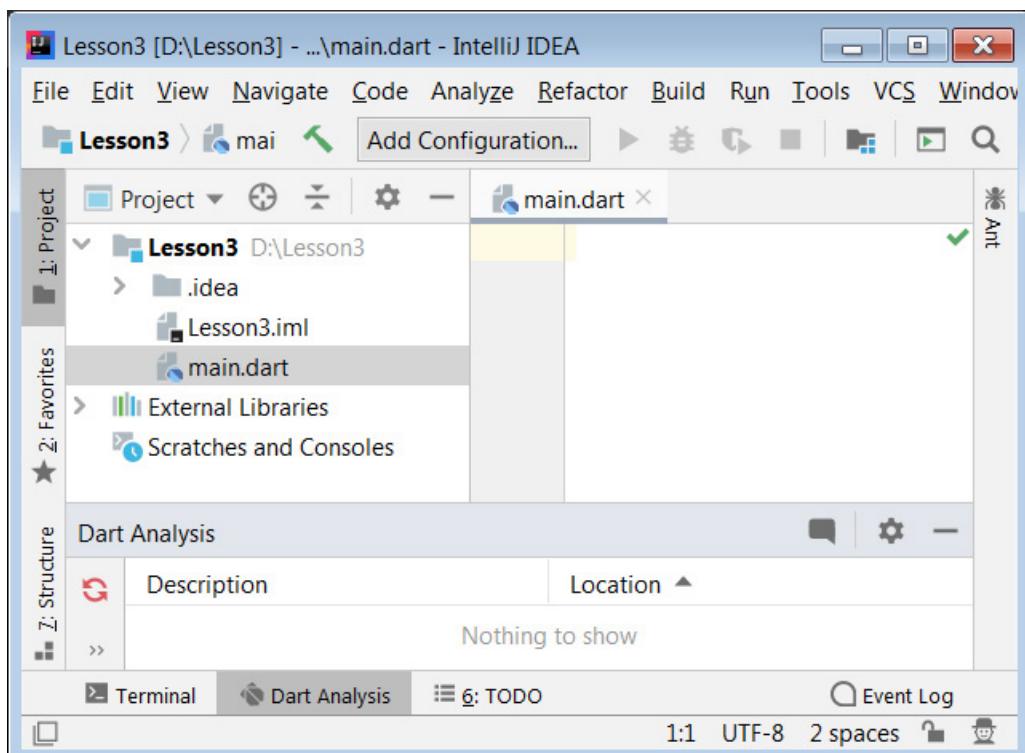


Now, Lesson3 Dart project has been created, and it can include the Dart files which you want.

For example, to create a **main.dart** file, right click the **Lesson3** project name, then select: **New**, then select **Dart File**. Then, type the file name: **main** as illustrated in the following figure and press **Enter** key.



You should get the following figure:



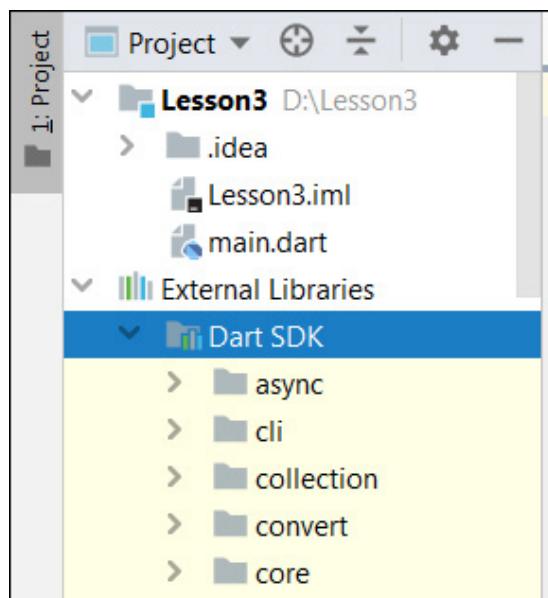
**Lesson3.iml** is a module file created by IntelliJ IDEA. This file stores information about a development module and project configurations such as the module paths, dependencies, and other settings

## Dart Libraries

The Dart SDK has the libraries and command-line tools that you need to develop Dart web, command-line, and mobile apps.

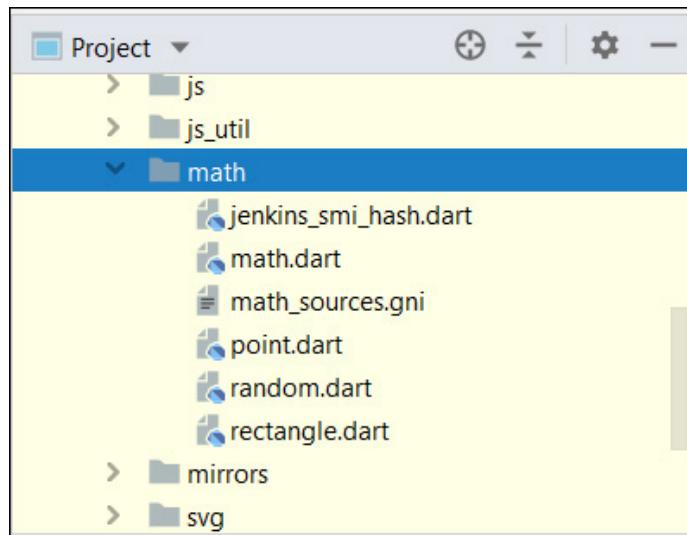
You can use the Dart libraries which are already added to your IDE since installing the Dart SDK, or you can create your own libraries which you may use frequently in developing your Dart projects. It is recommended to consider these libraries similar to Microsoft Word templates which you may use for creating different types of documents such as resumes, invoices or reports.

The Dart SDK includes a **libraries** directory for the Dart libraries. Also, you may use the **External Libraries**, installed with the Dart SDK directory, which includes the libraries created by Google developers. This directory is illustrated in the following figure:



### **Example:**

In this example, you will import the **math.dart** library from:  
**External Libraries → Dart SDK** to your Dart project.

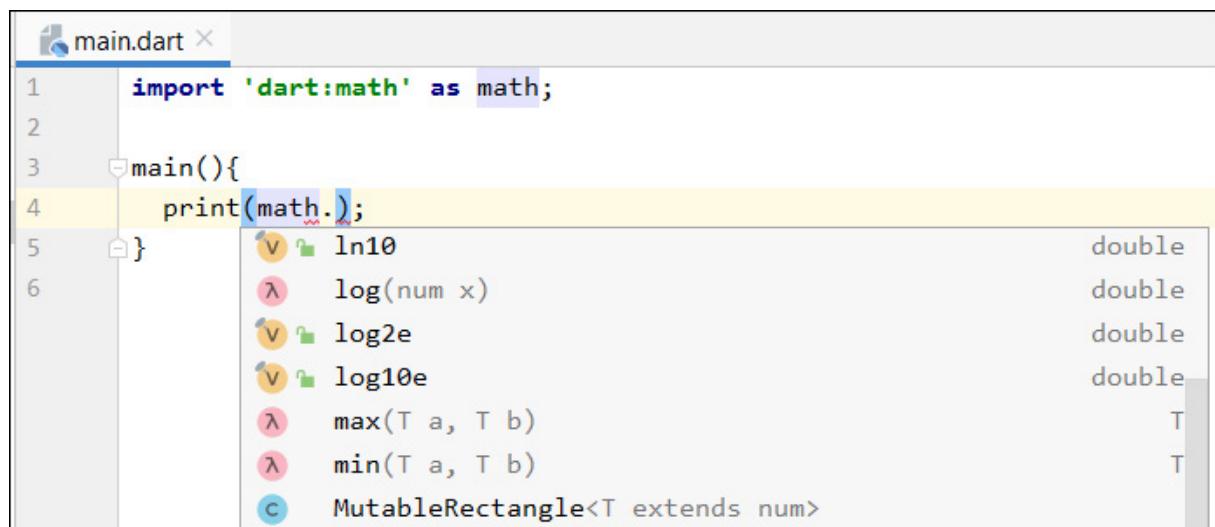


In your **main.dart** file, write the following command at the top of your dart file to import **math.dart** library as illustrated in the following code.

```
import 'dart:math' as math;
```

"**as math**" : means you can use **math** as reference to use this library. It is similar to the object name which will represent this library in writing the code. You can use any word.

Then, in the main function you can use all the methods which are imported to your Dart project when you start with **math**, as illustrated in the following figure:



In this example, we will use the maximum function. This function or method will find the maximum number of two numbers as illustrated in the following code:

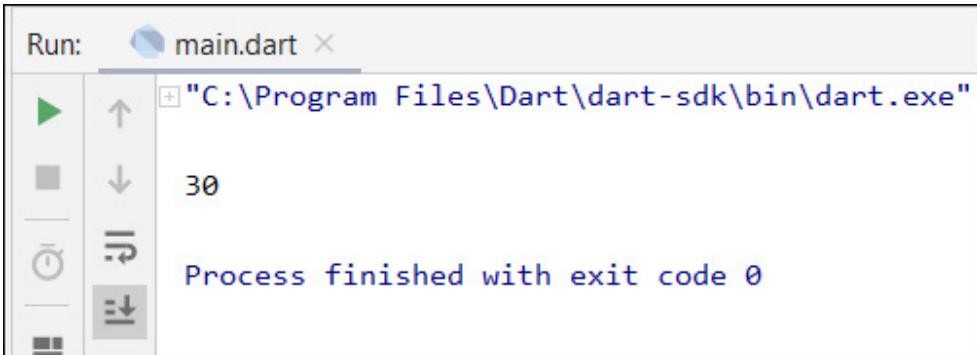
```
print(math.max(a, b));
```

The full code follows:

```
import 'dart:math' as math;

main(){
  print(math.max(5, 30));
}
```

The run output of this code follows:



Using this library, saves time and effort in creating a function that can do the same calculation.

### **Example:**

In this example, continue using the same previous `main.dart` file and create your own library. Then, you may use this library with your Dart files.

1- Right click **Lesson3** directory → **New** → **Dart File**

2- Type the file name : **MyLibraries** as illustrated in the following figure. Then press the **Enter** key:



3- Enter the following code in the **MyLibraries.dart** file (library file):

```
int SumNumbers(int a , int b) => a+b;
```

This function will return the sum of **a** and **b**.

4- To import or use the **MyLibraries.dart** library file in **main.dart** file, open **main.dart** file, and add the following code to first line of code:

```
import 'MyLibraries.dart' as mylib;
```

5- When you use this library with **print** method, you will get the following figure:

Note that all the methods inside this library will be ready for use. In this example, you have created one method.

```
main.dart × MyLibraries.dart ×
1 import 'MyLibraries.dart' as mylib;
2 import 'dart:math' as math;
3
4 main(){
5     print(math.max(5, 30));
6     print(mylib.);
7     ↗ SumNumbers(int a, int b)           int
8         try                      stmt.try -> try {stmt} catch (e,s) {}
9         tryon      stmt.try -> try {stmt} on Exception catch (e,...)
10
11 }
```

The screenshot shows a code editor with two tabs: 'main.dart' and 'MyLibraries.dart'. The 'main.dart' tab contains the following code:

```
import 'MyLibraries.dart' as mylib;
import 'dart:math' as math;

main(){
    print(math.max(5, 30));
    print(mylib.);
}
```

The cursor is on the line 'print(mylib.)'; a tooltip appears, showing the method signature 'SumNumbers(int a, int b)' and a brief description of its parameters. Below the tooltip, there are options to 'Press Enter to insert, Tab to replace' and 'Next Tip'.

The full code follows (replace **a** with 10 and **b** with 50):

```
import 'MyLibraries.dart' as mylib;
import 'dart:math' as math;

main(){
    print(math.max(5, 30));
    print(mylib.SumNumbers(10, 50));
}
```

6- The run output of this code follows:

The screenshot shows the Dart Editor's run interface. The title bar says "Run: main.dart". Below it is a toolbar with icons for play, stop, and refresh. To the right of the toolbar is the command line output area. The output shows the path "C:\Program Files\ Dart\dart-sdk\bin\dart.exe", followed by two lines of text: "30" and "60". At the bottom of the output area, the message "Process finished with exit code 0" is displayed.

7- Add a new function to **MyLibraries.dart** as illustrated in the grey highlighted color in the following code:

```
int SumNumbers(int a , int b) => a+b;  
int MultiplyNumbers(c,d) =>c*d;
```

8- Use **mylib** again in **main.dart** file to run **MultiplyNumbers** function as illustrated in the following code:

```
import 'MyLibraries.dart' as mylib;  
import 'dart:math' as math;  
  
main(){  
    print(math.max(5, 30));  
    print(mylib.SumNumbers(10, 50));  
    print(mylib.MultiplyNumbers(2, 4));  
  
}
```

9- The run output of this code follows:

The screenshot shows the Dart Editor's run interface. The title bar says "Run: main.dart". Below it is a toolbar with icons for play, stop, and refresh. To the right of the toolbar is the command line output area. The output shows the path "C:\Program Files\ Dart\dart-sdk\bin\dart.exe", followed by three lines of text: "30", "60", and "8".

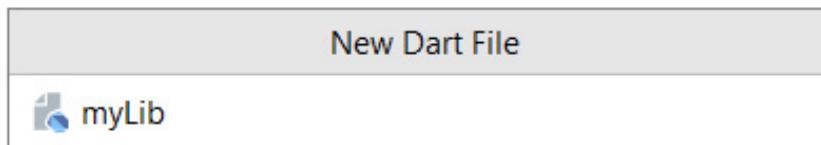
## Lab 3: Create a Small Overtime Payment Program

This is a small Dart program which calculates the total salary including overtime payment in a simple way. This program is just to review how to create a library file and then use it with a Dart file.

In this lab, you will create a library file called **myLib.dart**, which includes a class, create a **main.dart** file and then import this library file into the main Dart file.

The following are the lab steps:

- 1- Open IntelliJ IDEA, then click **File → New → Project**
- 2- Select **Dart** for the project type, then click **Next**
- 3- Type **Lab3** for the Project Name, then click **Finish**.
- 4- Right click **Lab3** directory, then select **New → Dart File**
- 5- Enter the file name **myLib**, as illustrated in the following figure, then press the **Enter** key.



- 6- Open the **myLib.dart** file, then type the following code:

```
class employee{  
    String jobTitle;  
    String location;  
    double salary;  
    double Overtimehr; //Number of Overtime hours  
  
    employee(this.jobTitle, this.location , this.salary, this.  
    Overtimehr);  
}
```

- 7- Right click **Lab3** directory, then select **New → Dart File**

- 8- Enter the file name **main**, then press **Enter** key.

- 9- Open the **main.dart** file, then type the following code:

```
import 'myLib.dart' as emp_Manager;
double extraPayPerhr; // Extra Payment Per Hour

main(){
    var William = new emp_Manager.employee("Manager", "Madrid", 5000,
7);

    if(William.salary>=4000){
        extraPayPerhr=10;
    }
    else{
        extraPayPerhr=20;
    }
    double Total_Salary= William.salary+(William.
Overtimehr*extraPayPerhr);

    print("Total Salary = $Total_Salary USD");
}
```

The run output of this code follows:



# Lesson 4: Introduction to Flutter

<b>Understanding Flutter.....</b>	4-2
<b>Flutter Framework.....</b>	4-4
<b>Android Studio .....</b>	4-5
What is Android Studio? .....	4-5
Android Studio Software Prerequisite.....	4-5
Installing Android Studio .....	4-8
<b>Flutter SDK .....</b>	4-14
<b>Installing and Configuring Flutter SDK.....</b>	4-14
<b>Creating a New Flutter Project .....</b>	4-20
<b>Setup an Android Virtual Device .....</b>	4-24
<b>Run a Flutter App .....</b>	4-30
<b>Installing Flutter on Mac.....</b>	4-36
<b>Test Your Flutter App on iOS Phone with Windows O.S .....</b>	4-37
<b>Android Studio Sugar and Spice .....</b>	4-45
<b>Run your Apps on a Hardware Device (Physical Phone) .....</b>	4-51
Run your Flutter App on Android Phone .....	4-52
Run your Flutter App on iPhone Device .....	4-56
<b>Emulator Debug Mode .....</b>	4-56
<b>Introduction to Flutter Widgets.....</b>	4-57
<b>Creating a Flutter App Using Widgets .....</b>	4-59
<b>What is a MaterialApp widget?.....</b>	4-64
<b>Lab 4: Creating a Simple Flutter App.....</b>	4-66

## Understanding Flutter

Flutter is Google's SDK for crafting beautiful, fast user experiences for apps from a single codebase. It is used for building high-performance, high-fidelity apps for iOS, Android, and web from a single codebase.

iOS and Android developers have many years in using Objective-C, Swift (Xcode) , Java or Kotlin to create iOS or Android apps, and then publish them to Google Play or Apple stores. Also, because customers prefers to work with one company to create the same mobile app for Google Play and Apple stores, IT companies have to hire one or more developers iOS apps and others for Android apps. In addition, mobile app developers need to learn Java or Kotlin for Android development and Objective C or Swift for iOS development to increase their chances of getting better jobs or to provide a complete mobile development solution to their customers.

Now, with Flutter, you will create a single code base only to create one mobile app valid to work with Android or iOS devices. Also, you can publish it to Google Play or Apple store. Therefore, Flutter is an amazing language, because it saves time and effort to create beautiful mobile apps.

Flutter works with existing codes and is used by developers and organizations around the world. It is a free and open source. Flutter will help you create beautiful, fast apps, with a productive, extensible and open development model.

Flutter has the following features:

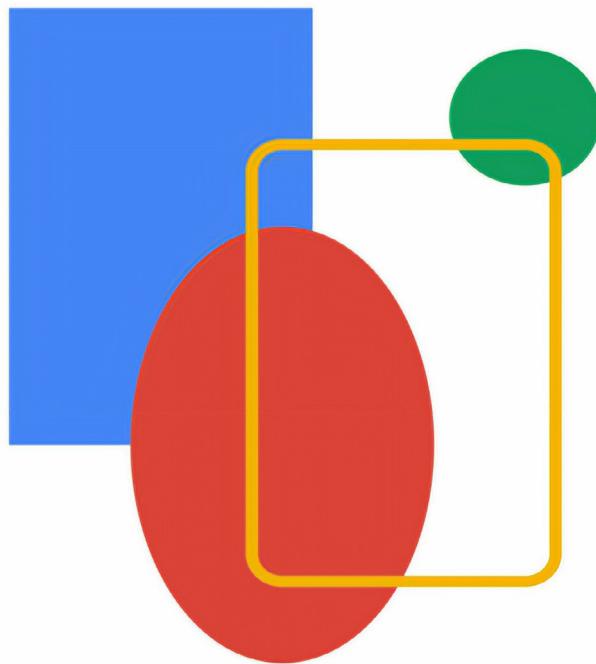
### **1- Beautiful UI (User Experiences):**

Flutter enables designers to deliver their full creative vision without being forced to water it down due to limitations of the underlying framework. Flutter's layered architecture gives you control over every pixel on the screen. It has powerful compositing capabilities which let you overlay and animate graphics, video, text and controls without limitations. Flutter includes a full set of widgets (text box, check box, radio buttons , and others) that deliver pixel-perfect experiences on both iOS and Android.

### **2- Fast Results**

When you run Flutter apps, they will run fast for the following reasons:

- Flutter uses an open source 2D graphics library called Skia , which has been built to support glitch-free, jank-free graphics at the native speed of your device.



- Flutter code is powered by the world-class Dart platform, which enables compilation to 32-bit and 64-bit ARM machine code for iOS and Android, as well as JavaScript for web and Intel x64 for desktop devices.

### 3- Productive Development

Flutter offers stateful hot reload, allowing you to make changes to your code and see the results instantly without restarting your app or losing its state.

### 4- Extensible and Open Model

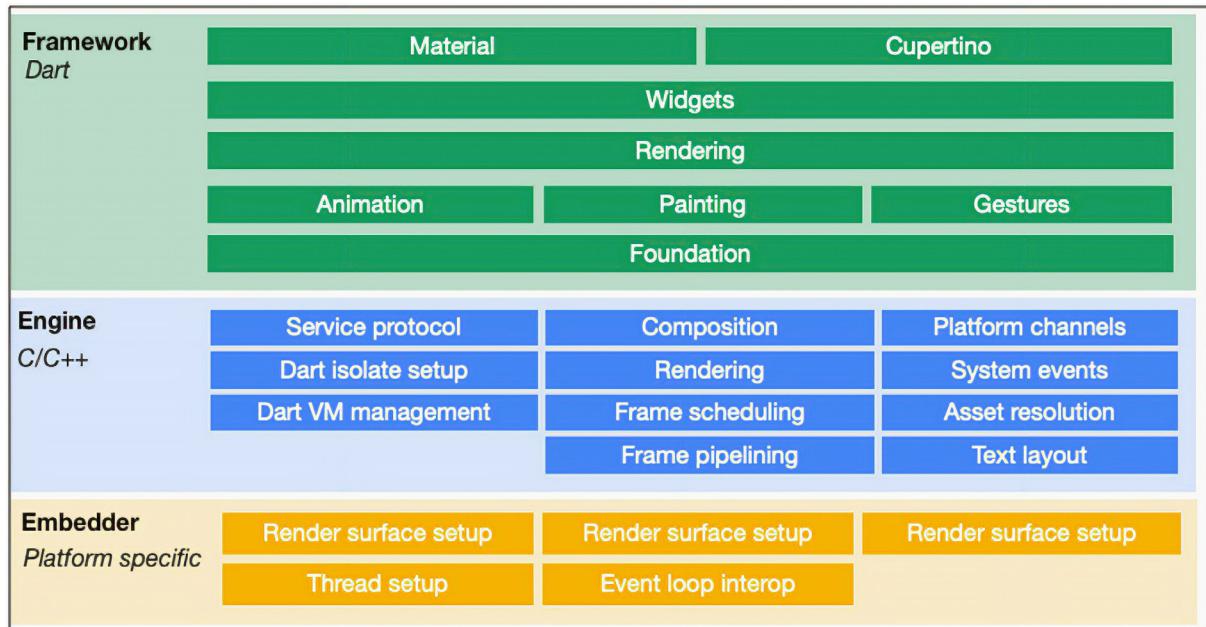
Flutter works with many development tools, which are compatible with Flutter plug-ins (Flutter SDK) such as Visual Studio Code, IntelliJ, and Android Studio.



Flutter provides thousands of packages to speed your development, regardless of your target platform. For example, the following web site : <https://pub.dev/flutter> includes many ready packages which can be used easily in building your Flutter apps.

## Flutter Framework

The Flutter framework is organized into a series of layers. Each layer is built upon the previous layer as illustrated in the following figure:



The upper layers of the framework are used more frequently than the lower layers.

The goal of this design is to help you do more with less code. For example, the Material layer is built by composing basic widgets (checkbox, text box, ...etc) from the widgets layer, and the widgets layer itself is built by orchestrating lower-level objects from the rendering layer.

The layers offer many options for building apps. Choose a customized approach to unlock the full expressive power of the framework, or use building blocks from the widgets layer, or mix and match. You can compose the ready-made widgets Flutter provides, or create your own custom widgets using the same tools and techniques that the Flutter team uses to build the framework.

Nothing is hidden from you. You collect the productivity benefits of a high-level, unified widget concept, without sacrificing the ability to dive as deeply as you wish into the lower layers.

## Android Studio

You can use many IDEs for Flutter development, including Android Studio, IntelliJ, and Visual Studio Code. In this course, we will use Android Studio IDE for Flutter app development.

### What is Android Studio?

Android Studio is the official Integrated Development Environment (IDE) for Android app development, based on IntelliJ IDEA .

### Android Studio Software Prerequisite

#### Installing Java JDK and JRE

The prerequisite to install Android Studio IDE is to install the **JDK** and **JRE**. The following steps summarize the installation process:

1- You can download the JDK and JRE for free from Oracle official website using the following web link:

<https://www.oracle.com/technetwork/java/javase/downloads/index.html>

Click the Java Platform (JDK) download button as illustrated in the following figure:



2- Depending on the operating system which your computer has, that is Windows, Linux or macOS, click the download link. Since I have Microsoft Windows 64 bits operating system, I will click the following link:

**Windows x64 Installer**

3- In the next step, and as illustrated in the following figure, select **I reviewed and accept the Oracle Technology Network License Agreement for Oracle Java SE**, then click Download Jdk button.



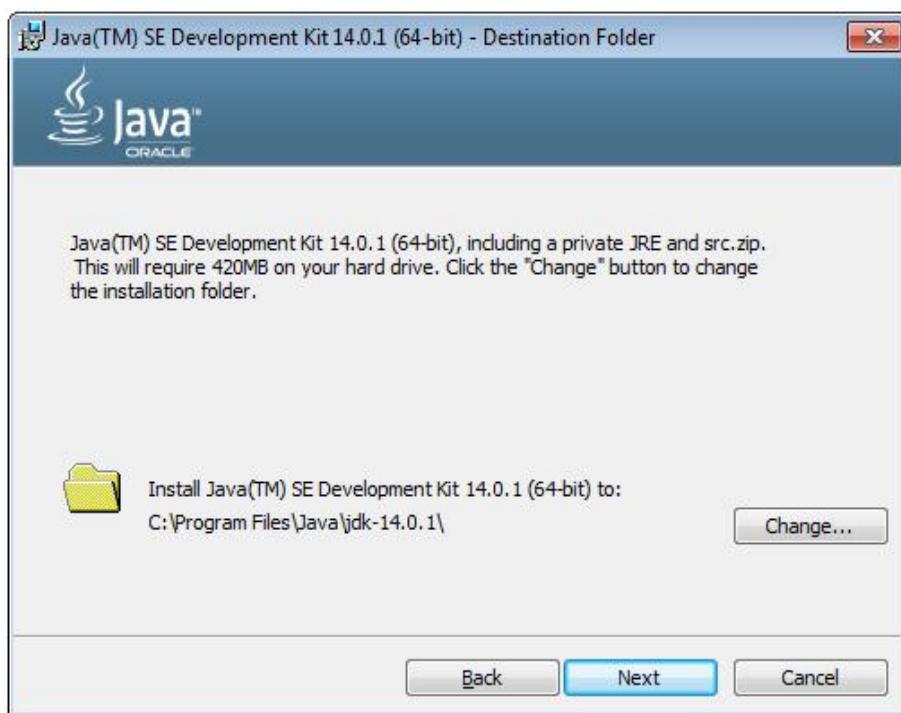
4- After finishing the download process, click the following download button on the bar above your task bar to start the installation process:



5- Click **Next** on the following wizard:



6- Then, click **Next** in the following figure:



7- After completing the installation process, click **Close**.

8- Now, to install Java JRE, go to:

<https://www.oracle.com/java/technologies/javase-downloads.html>

Then, click **JRE Download** as illustrated in the following figure:

The screenshot shows the Java SE 8u251 download page. On the left, there's a sidebar with links like Documentation, Installation Instructions, Release Notes, Oracle License (with sub-links for Binary, Documentation, and BSD Licenses), Java SE Licensing Information User Manual (with a note about third-party licenses), and Certified System Configurations. On the right, under the heading "Oracle JDK", there are five download links, each preceded by a downward arrow icon: "JDK Download", "Server JRE Download", "JRE Download" (which is highlighted in blue), "Documentation Download", and "Demos and Samples Download".

9- Select Windows x64 then click the following download link:



10 - Then, select **I reviewed and accept the Oracle Technology Network License Agreement for Oracle Java SE** then click **Download jre-8u251-windows-x64.exe**

11- You will be asked to create Oracle account. Click : Create Account under Don't have an Oracle Account?

Enter your personal information, then click Create Account button.

You will be asked to verify your email. Check your email inbox, and then verify the Oracle email by clicking Verify email address button. Then, you will get the following message in your web browser :

"Success. Your account is ready to use.". Click the Continue button.

12- Now, go back to the Oracle login dialog box. Enter your email and password for the account which you have created, then click Sign in. The download process will start.

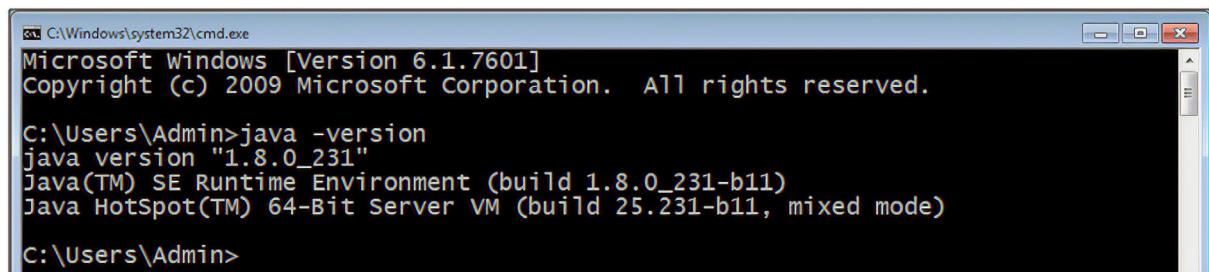
13- After finishing the download process, click the following download button on the bar above your task bar to start the installation process:



14- Click **Install** to start the Java JRE installation wizard.

15- After finishing the installation process, you will get the following message : **You have successfully installed Java.** Click **Close**.

16- To verify that you have successfully installed JDK on your Microsoft Windows computer, use the following command : **java -version**, and then you will get the Java version which you have installed on your computer as illustrated in the following figure:



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>java -version
java version "1.8.0_231"
Java(TM) SE Runtime Environment (build 1.8.0_231-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.231-b11, mixed mode)

C:\Users\Admin>
```

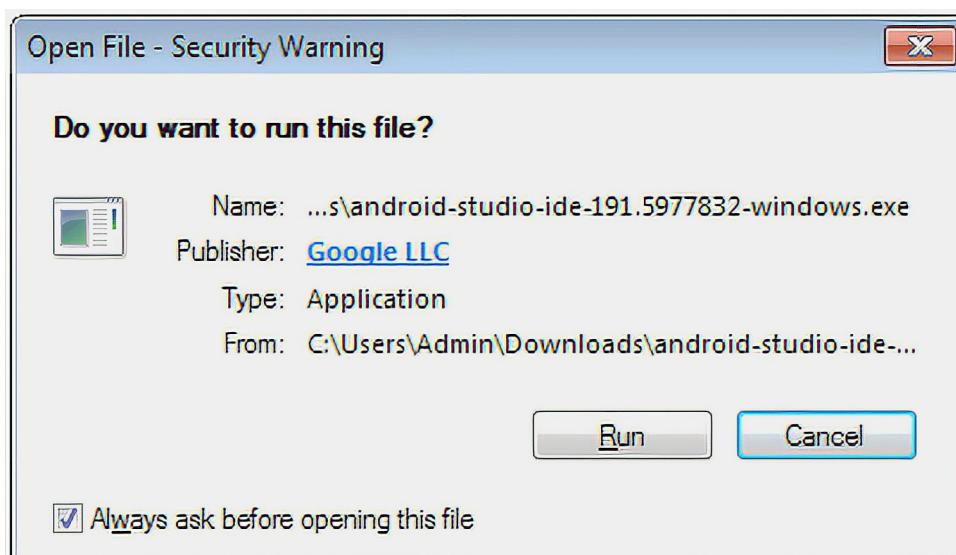
## Install Android Studio

After you complete installing the Android Studio prerequisite files, you can start the Android Studio installation process by following the following steps:

- 1- Go to : <https://developer.android.com/studio/index.html>
- 2- Click **DOWNLOAD ANDROID STUDIO** button.
- 3- In the first step of Android Studio installation wizard, select **I have read and agree with the above terms and conditions**, and then click **DOWNLOAD ANDROID STUDIO FOR WINDOWS**

The download process will start. The download file size is about 719 M; therefore, the download process will take some time depending on the speed of your internet connection.

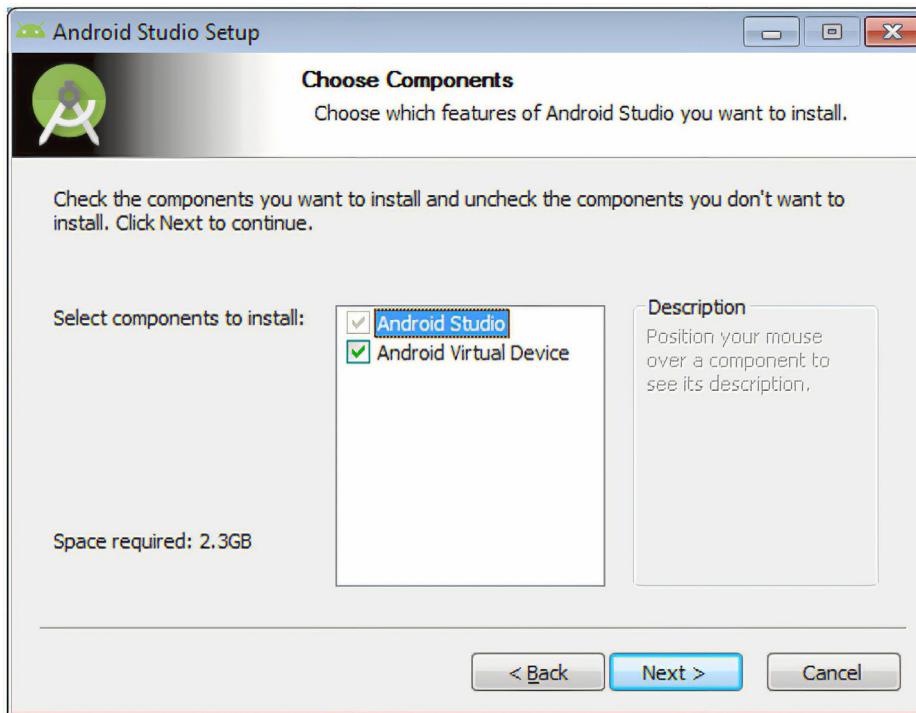
- 4-When the download is complete, click on android-studio-bundle icon on the previous download bar. You will get the following dialog box. Click **Run**.



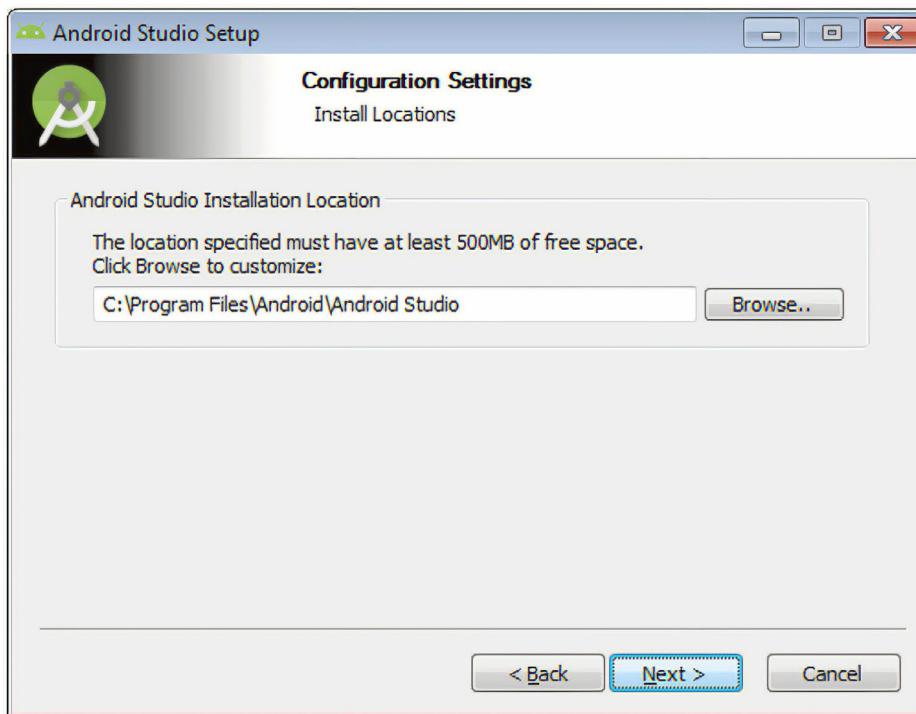
5- The Android Studio installation wizard will start as illustrated in the following figure. Click **Next**.



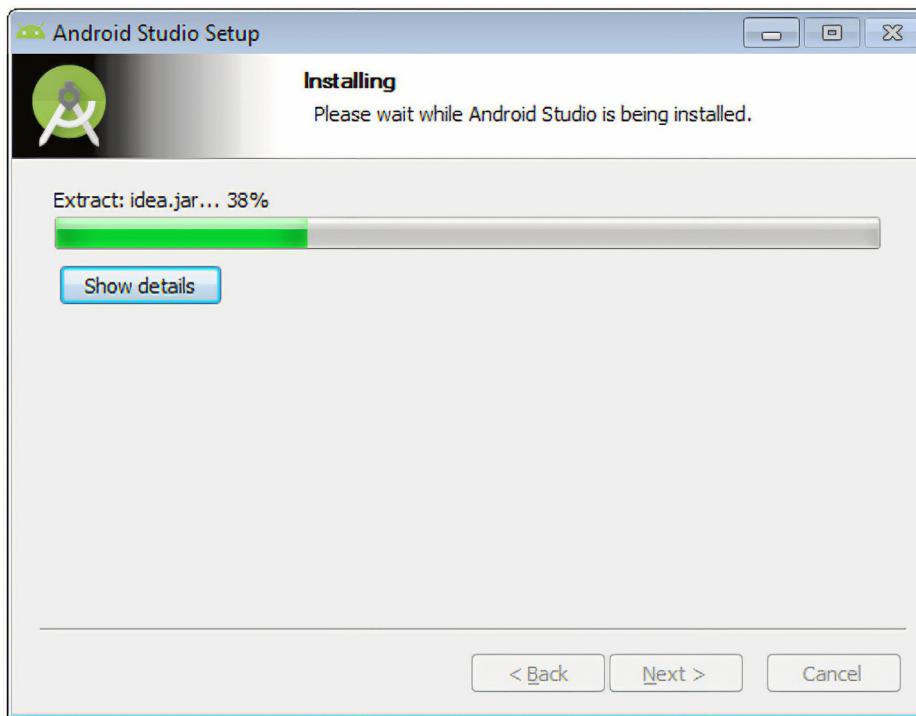
6- In this step and as illustrated in the figure below, keep the default configuration, and click **Next**.



7- Keep the default installation location as illustrated in the figure below, then click **Next**.

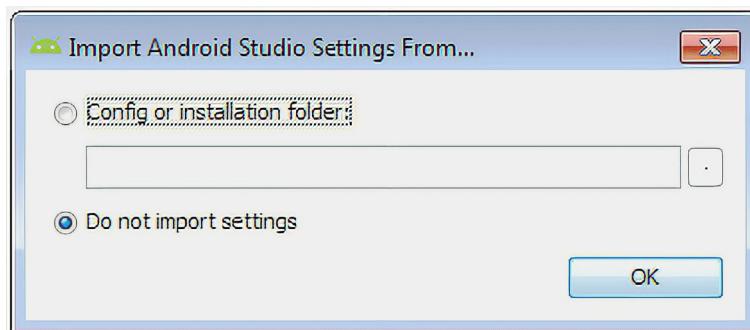


8- Click **Install**, then you will get the following figure, and after the installation complete, click **Next**

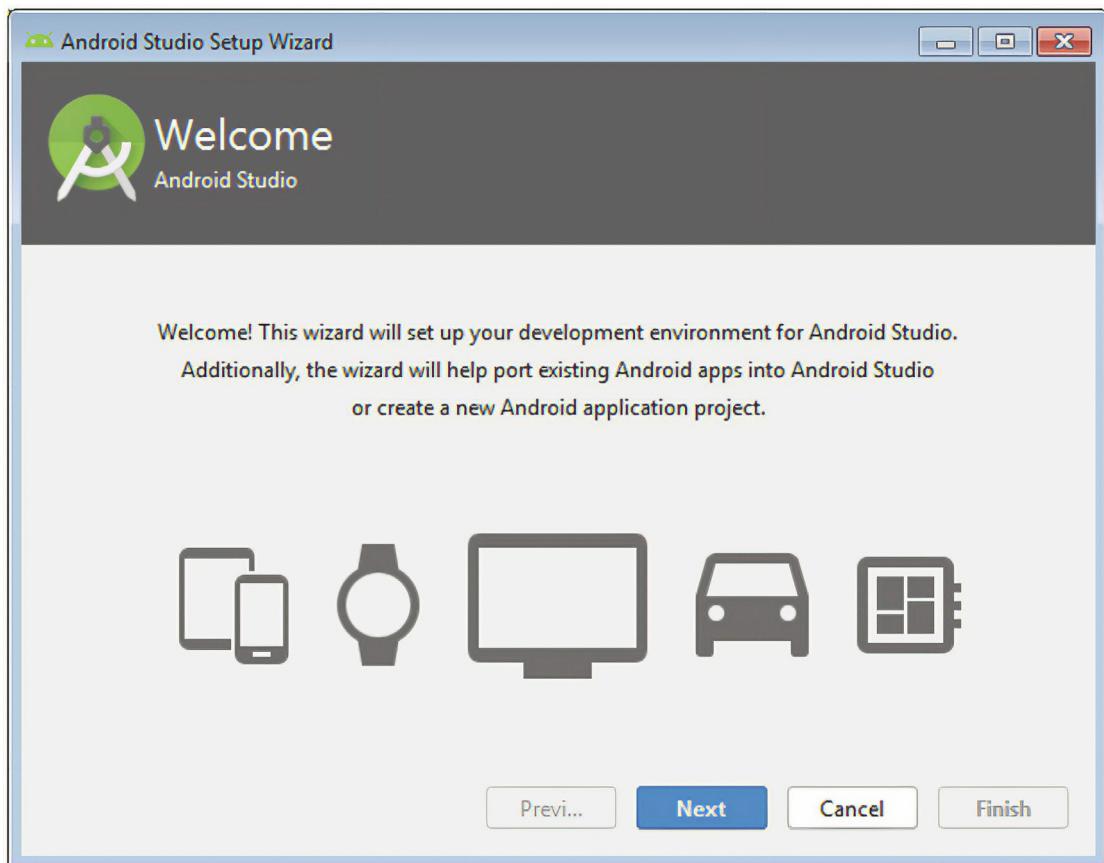


9- Click **Finish**, and then start Android Studio.

10- You will get the following dialog box when you start your Android Studio for the first time. Select **Do not import settings**, then click **OK**



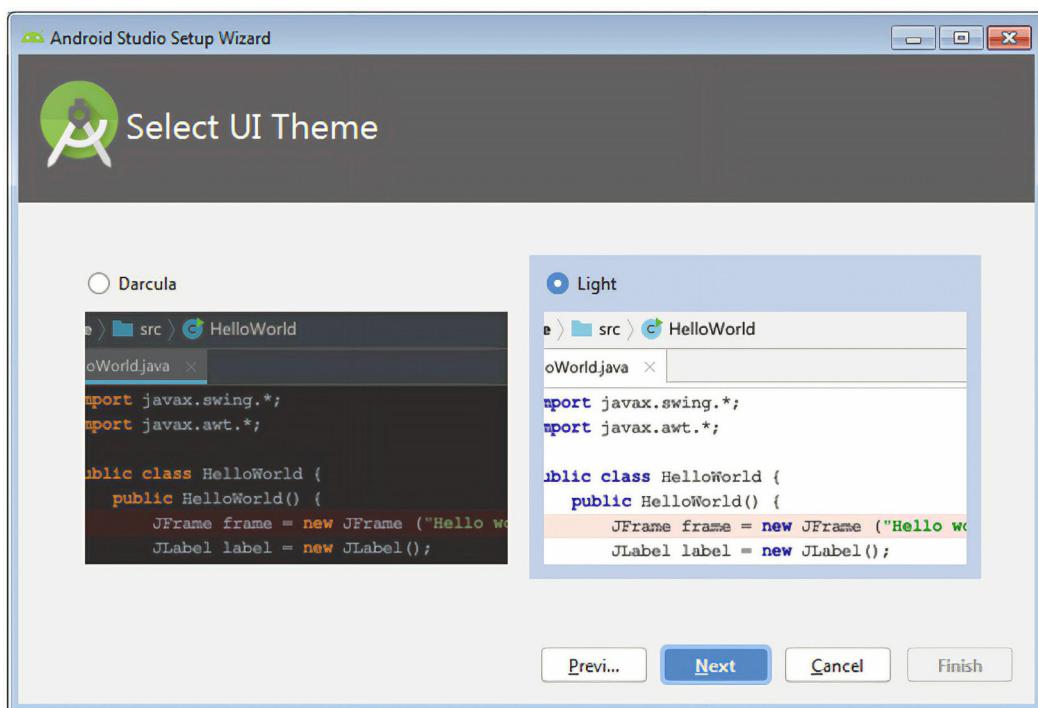
11- You will get the following figure. Click **Next**



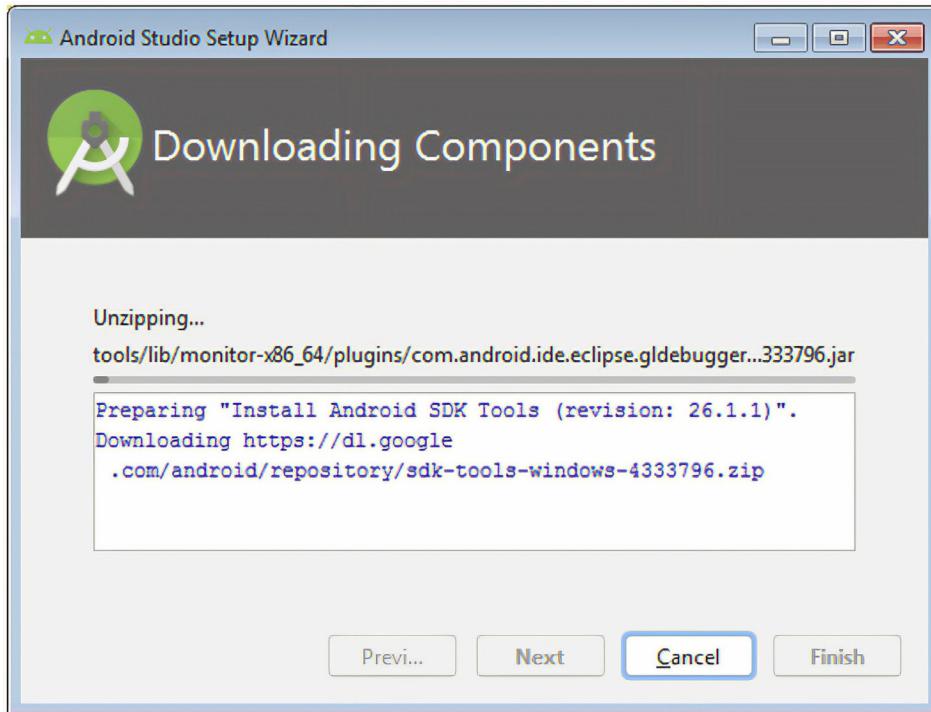
12 - In the following step, select **Standard**, and then click **Next** as illustrated in the following figure:



13- In this step, you have two themes for Android Studio user interface. Usually, professional developers select Darcula theme; however, due to issues related to printing this book, we prefer a white background. I will select **Light**, click **Next**, then click **Finish**.



Now, Android Studio will start installing the Android SDK tools as illustrated in the following figure:



13- After the installation process is completed, click **Finish**, then you will get the following dialog box:



## Flutter SDK

The Flutter SDK has the libraries, command tools, code completion, syntax highlighting, widget editing assists, run & debug support, and all the tools you need to develop Flutter apps.

By default, Android Studio is designed to create an Android app. Now, if you installed the Flutter SDK on your computer and configured it as a plug-in for Android Studio or other IDE software, Android Studio will be able to create Flutter apps.

## Installing and Configuring Flutter SDK

To create a Flutter app, you need to have the following software on your computer.

- Git Tool
- Flutter SDK
- Android Studio
- Android Studio Emulators (Android & iOS emulators or phones) to test your Flutter apps.

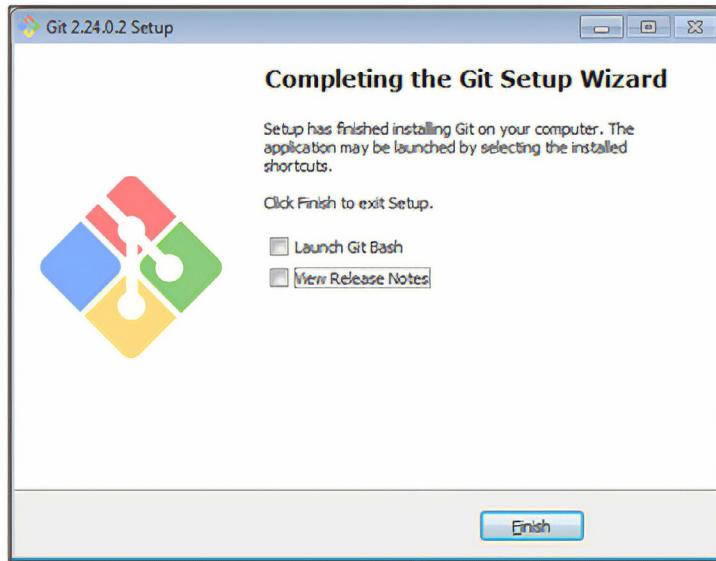
In the previous topic, you already installed Android Studio IDE. In the next topics of this lessons, you will install and configure the other software which will help you to create and test your Flutter apps.

To install and run Flutter, your development environment must meet the following minimum requirements:

- **Operating Systems:** Windows 7 SP1 or later (64-bit).
- **Disk Space:** 400 MB (does not include disk space for IDE/tools).
- **Git Tool :** You may use this tool to push your existing code to **git** version control system like Github ([github.com](https://github.com)). GitHub is a development platform inspired by the way you work. From open source to business, you can host and review code, manage projects, and build software alongside 40 million developers.

### Install Git Tool

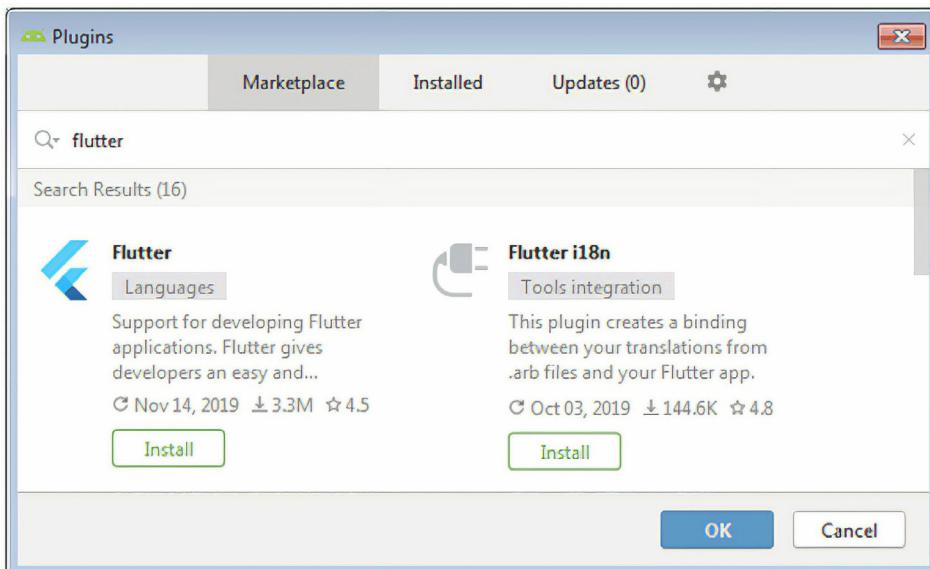
To install Git tool, go to : <https://git-scm.com/download>, and download Git version which is compatible with your operating system. Through the installation wizard, just click **Next** without changing any of the default installation settings. At the end of the installation wizard, you will get the following figure. Click **Finish**.



To install Flutter SDK for Windows, follow the following steps:

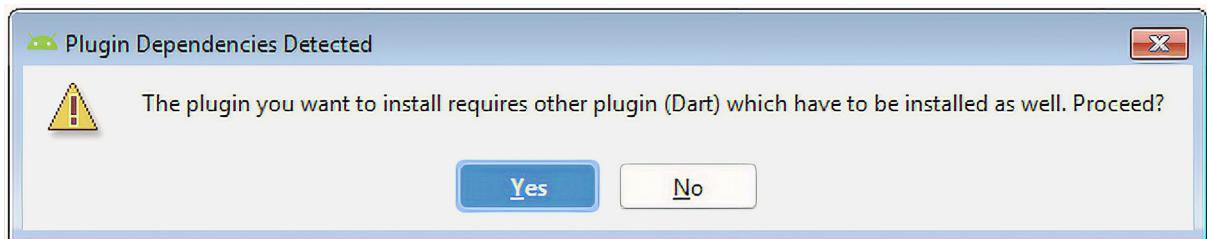
- 1- Go to : <https://flutter.dev/docs/get-started/install/windows>
- 2- Go to: “Get the Flutter SDK” section of this web page, then click : **flutter\_windows\_v1.9.1+hotfix.6-stable.zip** or the latest released file available on this web page. The file size is about 670 MB. This process will take some time depending on the speed of your internet connection.
- 3- Extract the zip file and place the contained flutter in the desired installation location for the **Flutter SDK** . In this course, we will use the following path: **C:\src**
- 4- Start **Android Studio**.

5- Click **Configure → Plugins**, then in the **Marketplace** tab , as illustrated in the following figure type **flutter** in the search area, and then press **Enter**:



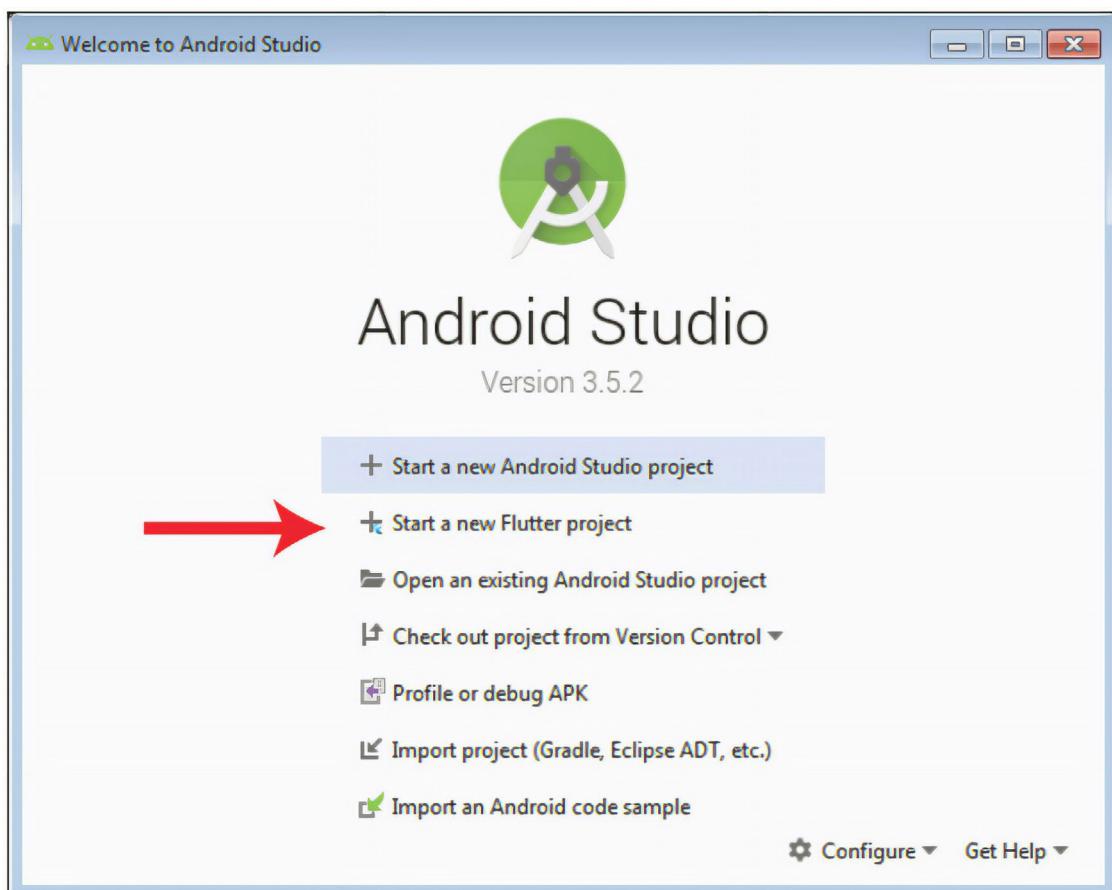
## 6- Click Install

7- Click **Accept** for any warning message, then you will get the following dialog box. Click **Yes**.



8- Click **Restart IDE** to restart Android Studio IDE.

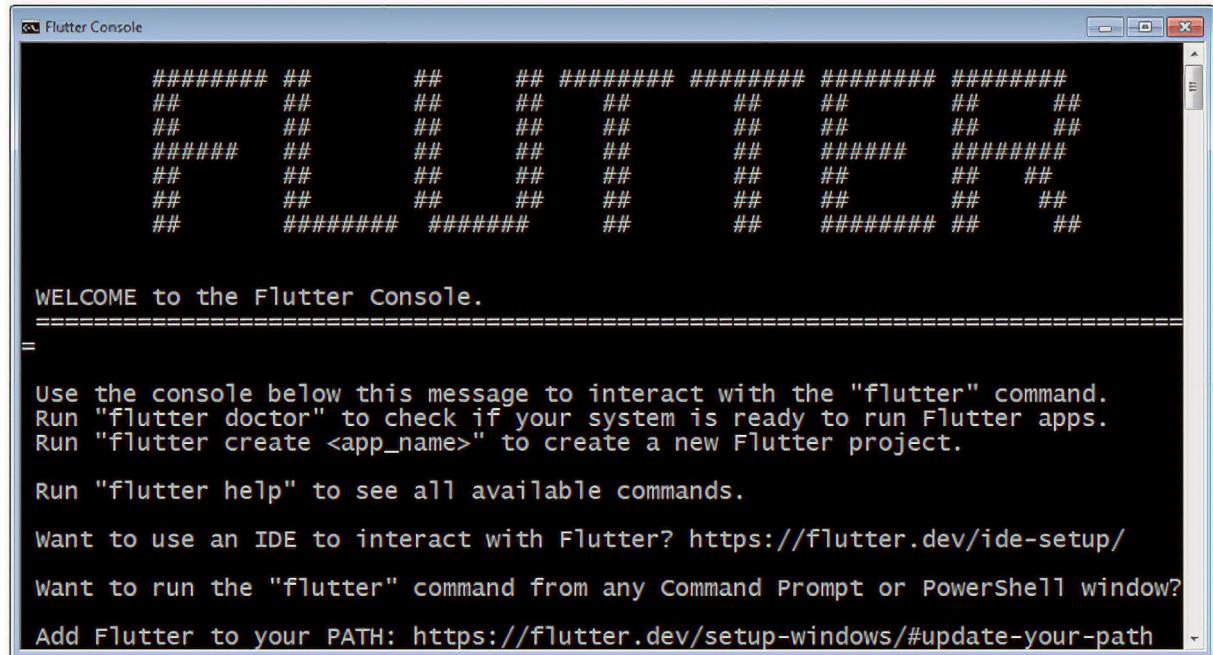
9- Now, as you see in the following figure for the Android Studio welcome dialog box, you have **Start a new Flutter project** choice. If you get this option, it means your Flutter SDK is installed and configured successfully and your Android Studio is ready to create a Flutter project.



Now, before creating any new Flutter project, you must check your installation, and if your Android Studio IDE is ready to create a Flutter project or if there are other steps or prerequisites that must be done. **Flutter Doctor** command checks your

environment and displays a report of the status of your Flutter and Android Studio installation as you will see explained in the next step.

10- You are now ready to run Flutter commands (Flutter doctor) in the Flutter Console. Go to : **C:\src\flutter** , then open : **flutter\_console.bat**, you will get the following console:



```
Flutter Console

#####
##      ##  ##  ##  ##  ##  ##  ##  ##  ##  ##
##      ##  ##  ##  ##  ##  ##  ##  ##  ##  ##
##      ##  ##  ##  ##  ##  ##  ##  ##  ##  ##
#####  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##
##      ##  ##  ##  ##  ##  ##  ##  ##  ##  ##
##      ##  ##  ##  ##  ##  ##  ##  ##  ##  ##
##      ##  ##  ##  ##  ##  ##  ##  ##  ##  ##

WELCOME to the Flutter Console.
=====
=                                         =
Use the console below this message to interact with the "flutter" command.
Run "flutter doctor" to check if your system is ready to run Flutter apps.
Run "flutter create <app_name>" to create a new Flutter project.

Run "flutter help" to see all available commands.

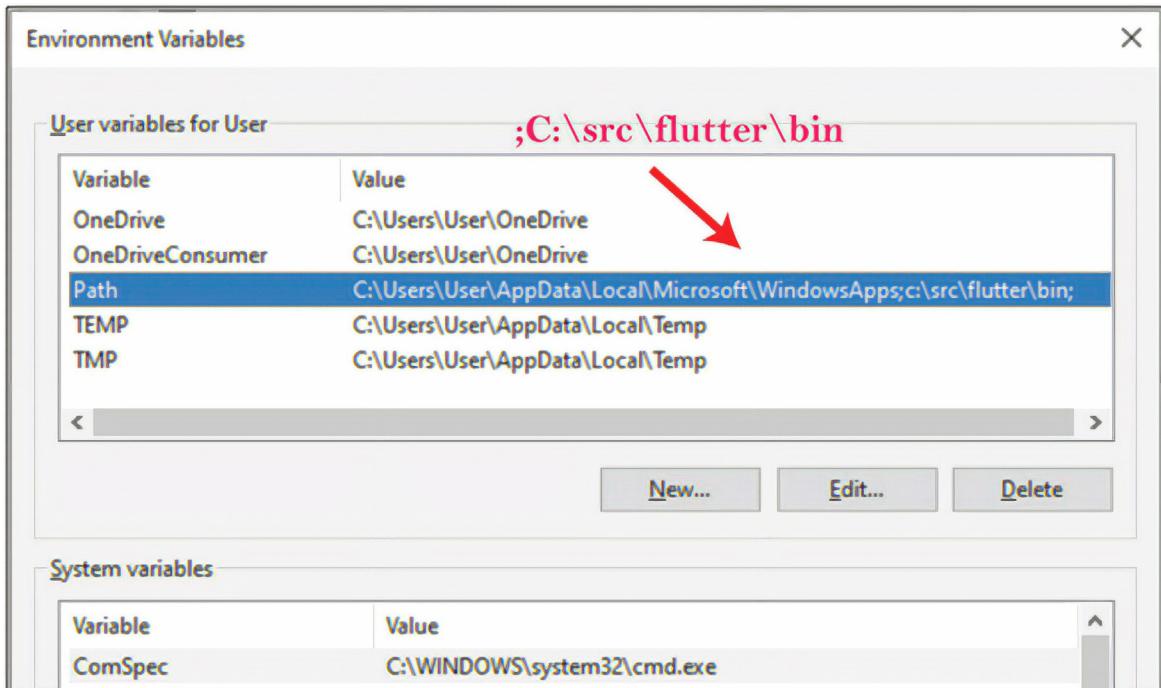
Want to use an IDE to interact with Flutter? https://flutter.dev/ide-setup/
Want to run the "flutter" command from any Command Prompt or PowerShell window?
Add Flutter to your PATH: https://flutter.dev/setup-windows/#update-your-path
```

If you are using Microsoft Windows operating system, and you wish to run Flutter commands in the regular Windows command prompt console, take the following steps to add Flutter to the PATH environment variable:

I- Go to **Control Panel** , then click **System**

II- Open **Advanced System Settings**

III- Click **Environment Variables**, and under **User variables**, check if there is an entry called **Path**. If the entry exists, click **Edit**, and append the full path to **C:\src\flutter\bin** using ; (semicolon) as a separator from existing values. If the Path doesn't exist, create a new user variable named Path with the full path to **C:\src\flutter\bin**, then click **OK** as illustrated in the following figures for Windows 10.



Note that you have to close and reopen any existing console windows for these changes to take effect.

11- Now, open the command prompt **Run → cmd**, then press **Enter**.

12 - Type the following command : **flutter doctor**, then press **Enter**.

This command checks your environment and displays a report of the status of your Flutter installation. The output for my computer was as follows:

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, v1.9.1+hotfix.6, on Microsoft Windows [Version
     6.1.7601], locale en-US)
[!] Android toolchain - develop for Android devices (Android SDK version 29.0.
     X Android licenses not accepted. To resolve this, run: flutter doctor
       --android-licenses
[✓] Android Studio (version 3.5)
[!] Connected device
     ! No devices available

! Doctor found issues in 2 categories.

C:\Users\Admin>
```

13 - As you see in the flutter doctor command result, to complete the Android Studio installation, you must accept some Android licenses. To do that, type the following command, and then press **Enter** :

```
flutter doctor --android-licenses
```

You will get many dialog boxes asking you to accept the Android and Google license parts as follows:

Review licenses that have not been accepted (y/N)?

Type **Y**, and then press **Enter**.

Then, you will get a list including Google License agreement, and the following message:

Accept? (y/N):

Type **Y**, then press **Enter** key many times to accept all the Android SDK license parts. Then, at the end you will get the following figure:

The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. It displays the following text:  
ND **AS AVAILABLE**, POSSIBLY WITH FAULTS, AND WITHOUT REP  
OF ANY KIND.  
  
10.8 Open Source Software. In the event Open Source soft  
aluation Software, such Open Source software is licensed  
ble Open Source software license agreement identified in  
e comments in the applicable source code file(s) and/or  
in the Evaluation Software. Additional detail may be av  
e) in the accompanying on-line documentation. With respe  
ftware, nothing in this Agreement limits any rights unde  
supercede, the terms of any applicable Open Source soft  
-----  
Accept? (y/N): y  
All SDK package licenses accepted  
  
C:\Users\Admin>

Now, type the following command in the command prompt again. Then press **Enter**:

```
flutter doctor
```

Then, you will get the following result:

```
C:\Users\Admin>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, v1.9.1+hotfix.6, on Microsoft Windows [Version 6.1.7601], locale en-US)
[✓] Android toolchain - develop for Android devices (Android SDK version 29.0.2)
[✓] Android Studio (version 3.5)
[!] Connected device
    ! No devices available

! Doctor found issues in 1 category.

C:\Users\Admin>
```

As you see in the previous Flutter doctor command run result, you still need to install or add a virtual device (virtual phone) or Android emulator to your Android Studio IDE. The virtual devices (mobile , tablets , smart watch , TV or other) allow you to test your application without having to have the physical devices on hand. To add this device, you should open Android Studio, create a Flutter application, and then use AVD Manager tool in Android Studio to add an Android emulator (virtual phone).

Remember, that to create a Flutter app, you need to download and install the following:

- Git Tool ( ✓ done)
- Flutter SDK ( ✓ done)
- Android Studio ( ✓ done)
- Android Studio emulator or Android virtual device (in the next step)

## **Creating a New Flutter Project**

Now, Android Studio IDE is ready to create a Flutter project for iOS and Android apps.

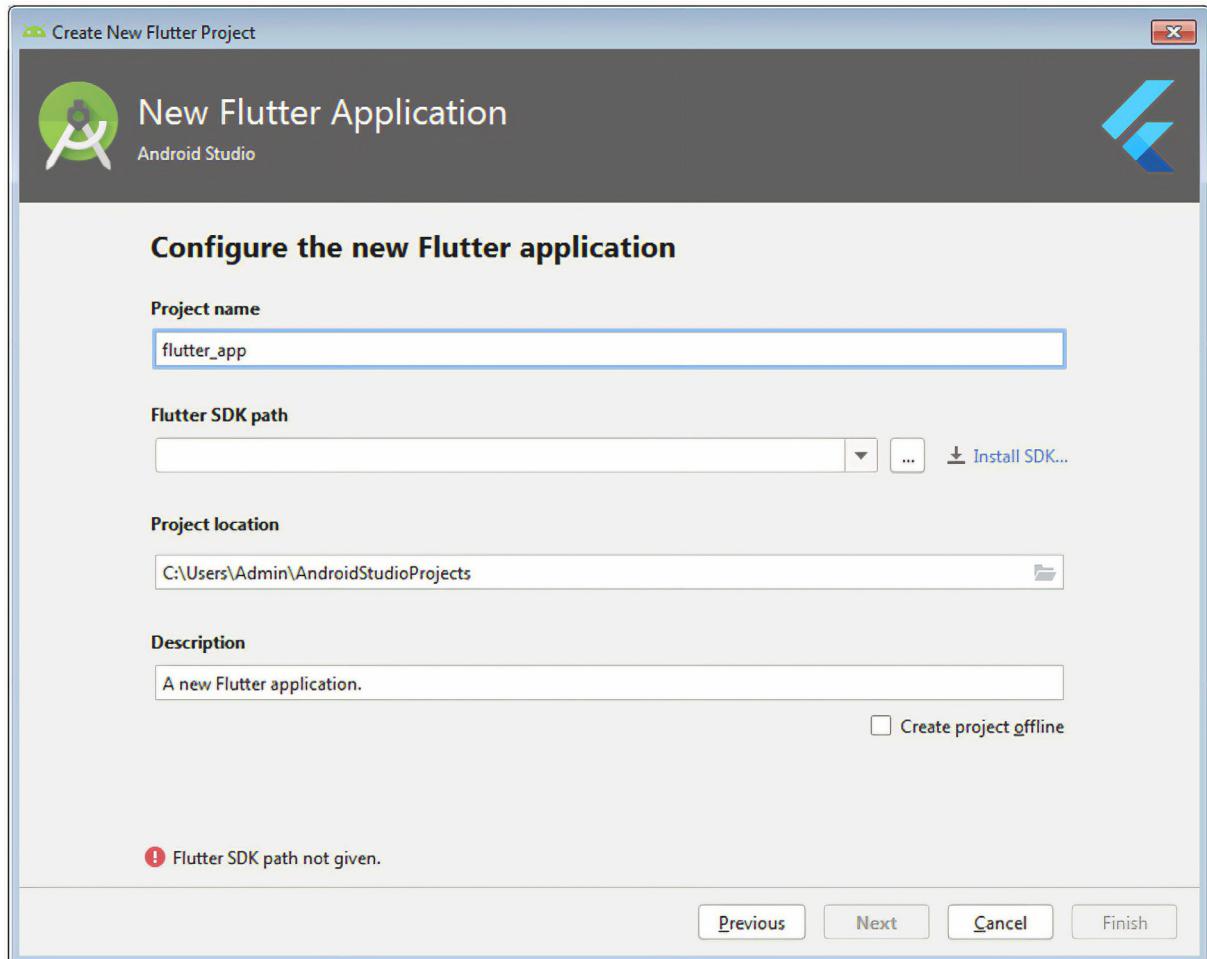
Follow the following steps to create your first Flutter project:

**1- Open Android Studio**

**2- Click Start a new Flutter project (if it is already installed before, click: File → New → New Flutter Project)**

**3- Select Flutter Application, and then click Next**

4- In the next step , and as illustrated in the following figure, you will enter the Flutter project name, Flutter SDK path, and determine the project save location.



Enter the following information, and click **Next**:

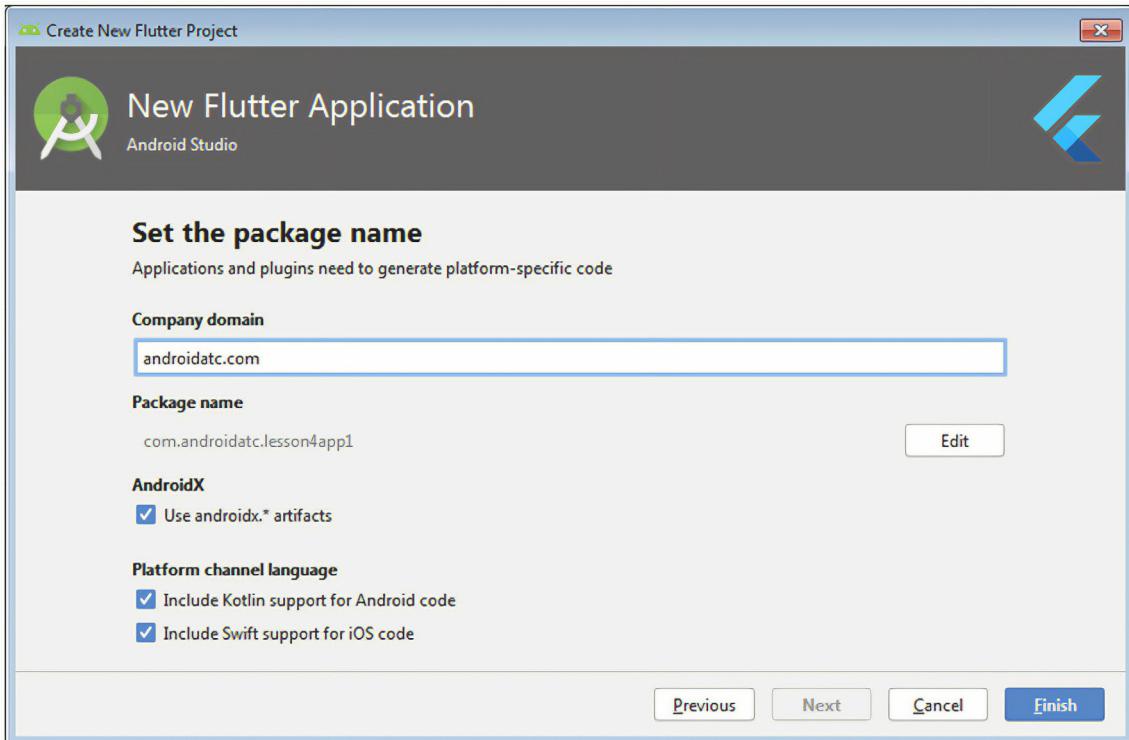
Project Name: **lesson4\_app1**

Flutter SDK path: **C:\src\flutter** (Select where you installed Flutter SDK on your computer)

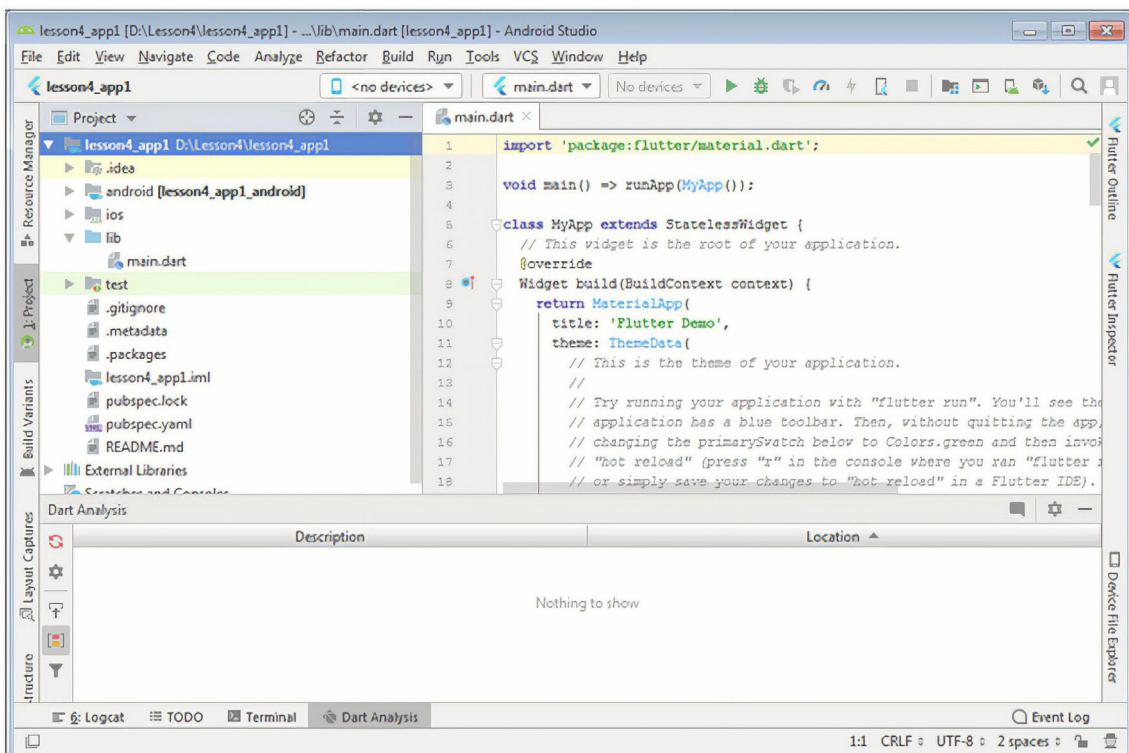
Project Location: **D:\Lesson4**

5- In the next step, Android Studio asks for an organization name in reverse domain order such as com.androidate along with the name of the app used as the package name for Android, and the Bundle ID for iOS when the app is released (published). If you think that you might ever release this app to the Google Play or Apple store, it is better to specify them now. They cannot be changed once the app is released. Your organization name should be unique.

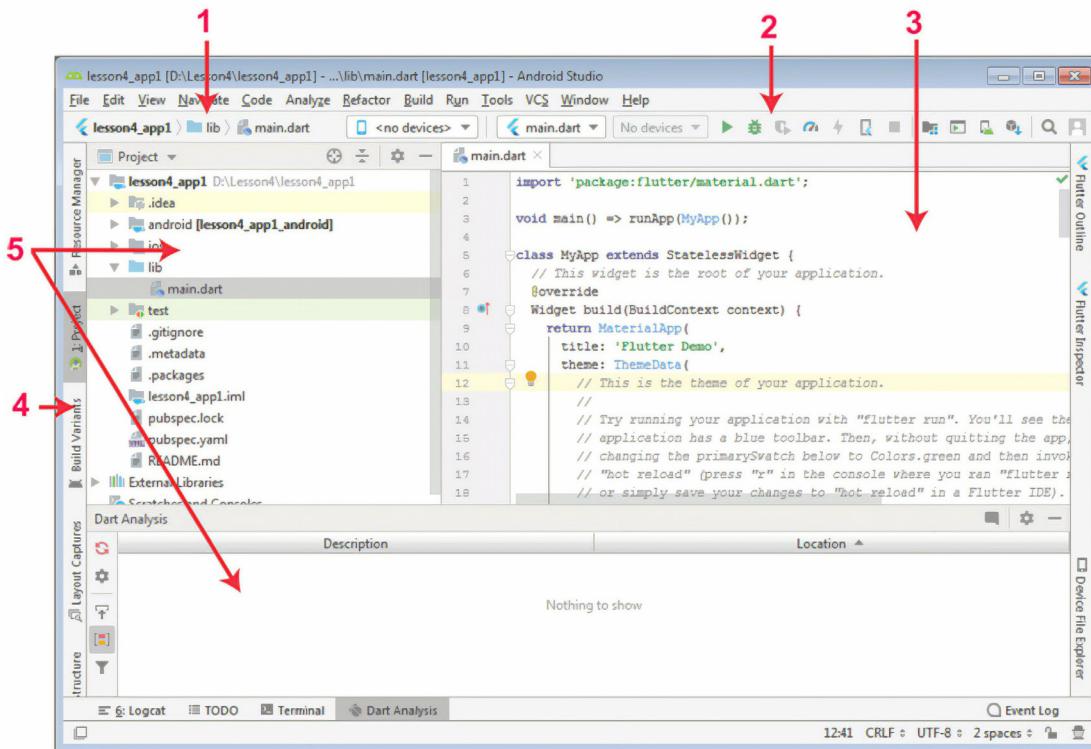
Type only: **androidatc.com**, and then the package name will be : com.androidatc.lesson4app1 as illustrated in the following figure. Then click **Finish**.



Then, you will get the following figure which includes the default **main.dart** file. This file includes Dart code written by Flutter team.



The following figure displays the name of each part of the Android Studio interface:



**1** → The **navigation bar** helps you navigate through your project and open files for editing. It provides a more compact view of the structure visible in the **Project** console.

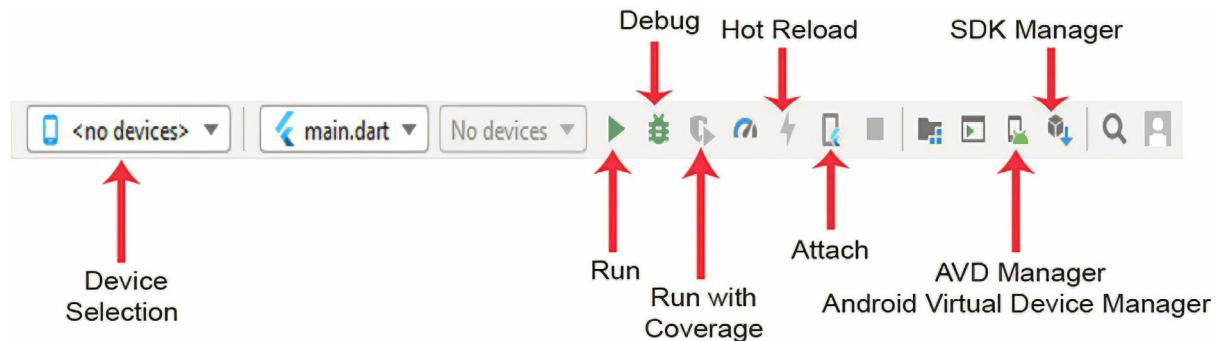
**2** → The **toolbar** lets you carry out a wide range of actions, including running your app and launching Android tools.

**3** → The **editor window** is where you add a new or modify an existing code to flutter app files.

**4** → The **tool window bar** runs around the outside of the IDE window and contains the buttons that allow you to expand or collapse individual tool windows.

**5** → The **tool windows** give you access to specific tasks like project management, search, version control, and more. You can expand and collapse them.

The following figure displays some parts of Android Studio tool bar:

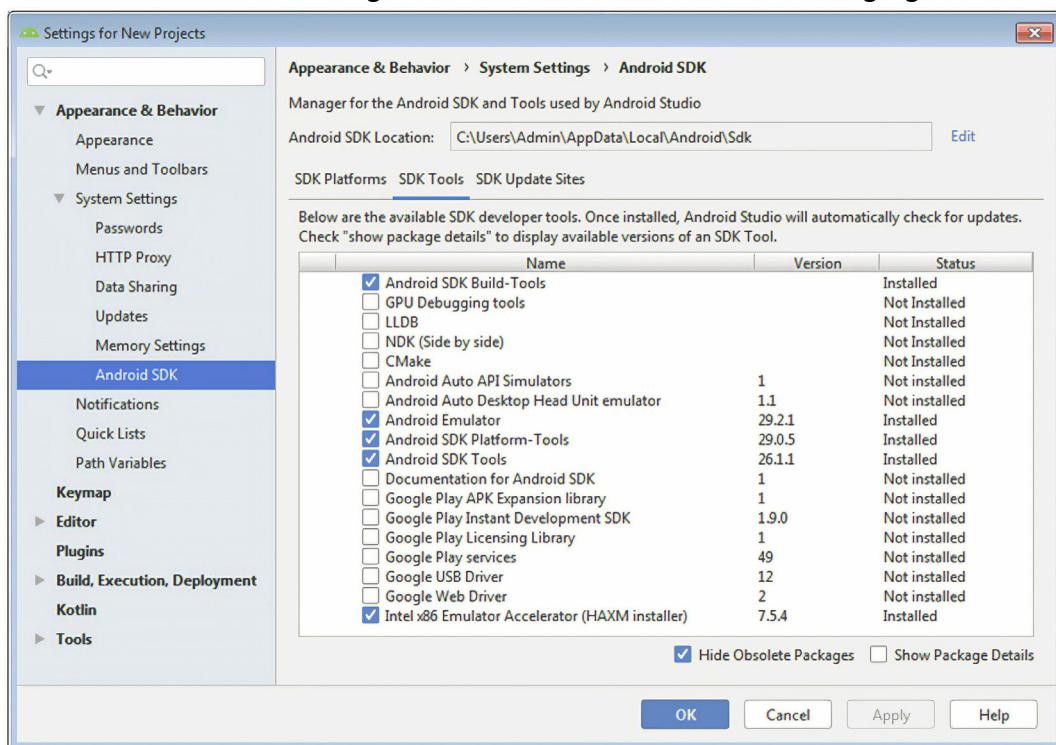


## Setup an Android Virtual Device

An Android Virtual Device (AVD) is a configuration that defines the characteristics of an Android phone, tablet, Wear OS, Android TV, or Automotive OS device that you want to simulate in the Android Emulator. In this course, you will add Android and iOS emulator (phones) to run your app code on Android and iOS phones or emulators.

The AVD Manager is an interface you can launch from Android Studio that helps you create and manage AVDs.

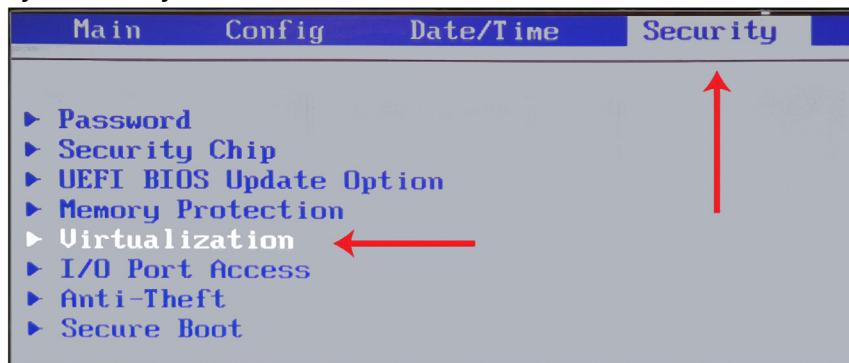
Before adding a virtual device to your Android Studio, be sure that your computer BIOS is configured to enable a virtualization feature. This allows virtual devices such as phone emulators to run on your computer. To know if this feature is enabled on your computer BIOS, open your Android Studio, then click **Tools** menu, select **SDK Manager**, and click **SDK Tools** tab in the right side as illustrated in the following figure:



Here, check if the last choice which is **Intel x86 Emulator Accelerator (HAXM installer)** is installed. This means the virtual feature is already enabled on your computer BIOS.

If this option is not installed, try to install it by checking its checkbox. Then click **Apply** button. If you get an error message such as : *Unable to install Intel HAXM, HAXM device is not found*, you must check your computer BIOS by restarting your computer, and during the startup process continue pressing **F2** , **F1** , or **Esc** key depending on your computer motherboard settings until you start your computer BIOS system. You must enable your computer BIOS virtualization feature as

illustrated in the next two images. Select the **Security** tab. Then select **Virtualization** using your keyboard keys.



**Note:** The location of virtualization feature in the BIOS system may vary depending on the hardware vendor type.

Press **Enter**, to edit the virtualization configuration. Be sure that this feature is enabled as illustrated in the figure below. If not, press the **tab** key to change its status from disabled to enabled.



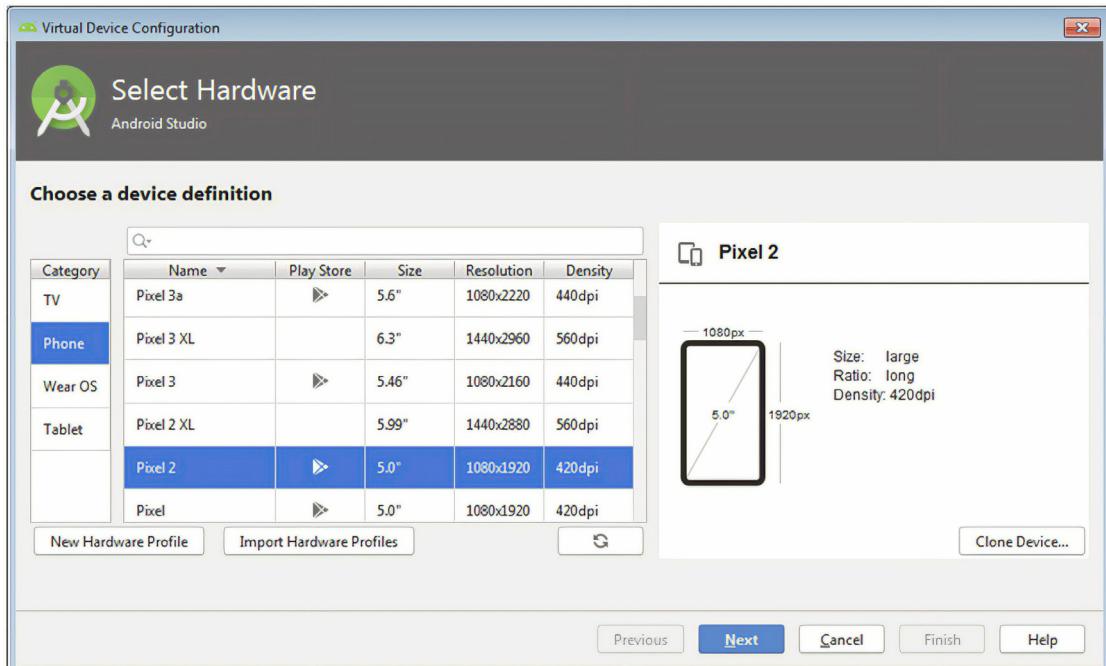
Press **F10** key to save and exit from BIOS configuration.

If everything is configured well, you should follow the following steps to setup your first Android virtual device (phone emulator) by performing the following steps:

1- Click the **AVD Manger** button on the Android Studio tool bar, or click: **Tools → AVD Manger**, then you will get the following dialog box:

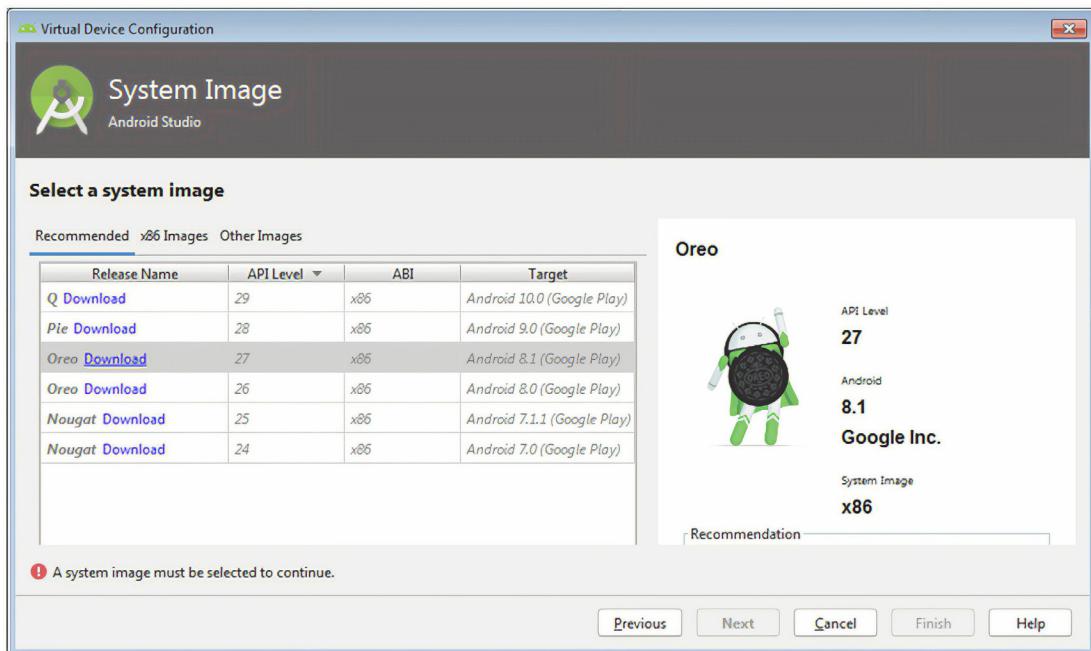


2- Click **Create Virtual Device** button, then you will get the following dialog box:

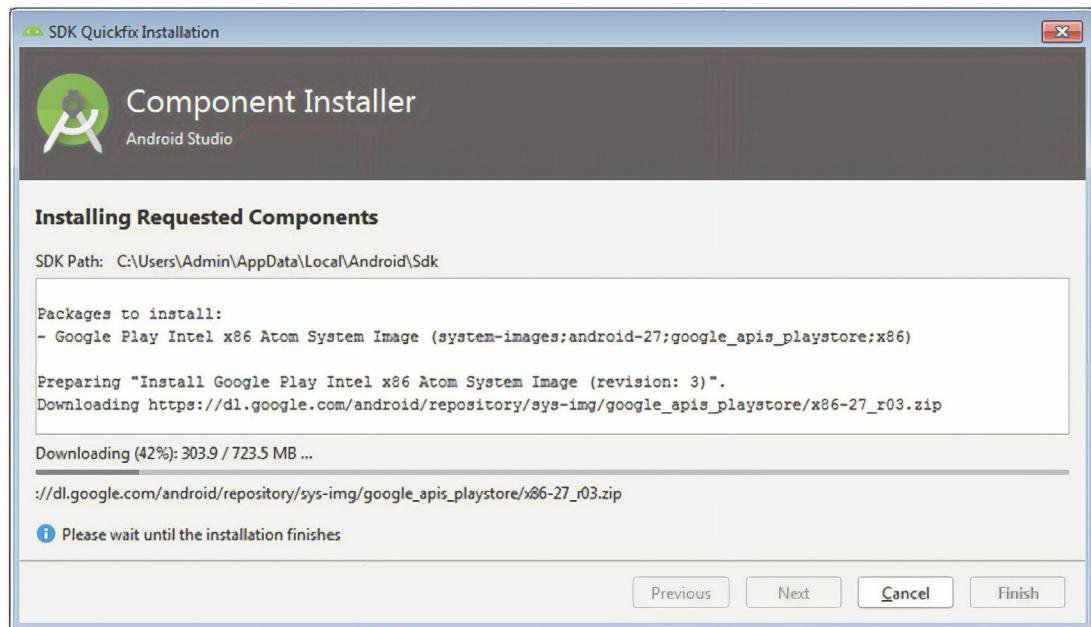


3- Select **Pixel 2** as phone emulator type, and click **Next**.

4- Now, you will select your phone emulator operating system which you will later use to test your app code. Because almost all phone devices in the market use at least Android 8 operating system, and the latest version can test the old but not the other way round, Select **Android Oreo (Android 8.1)** operating system as illustrated in the following screenshot.

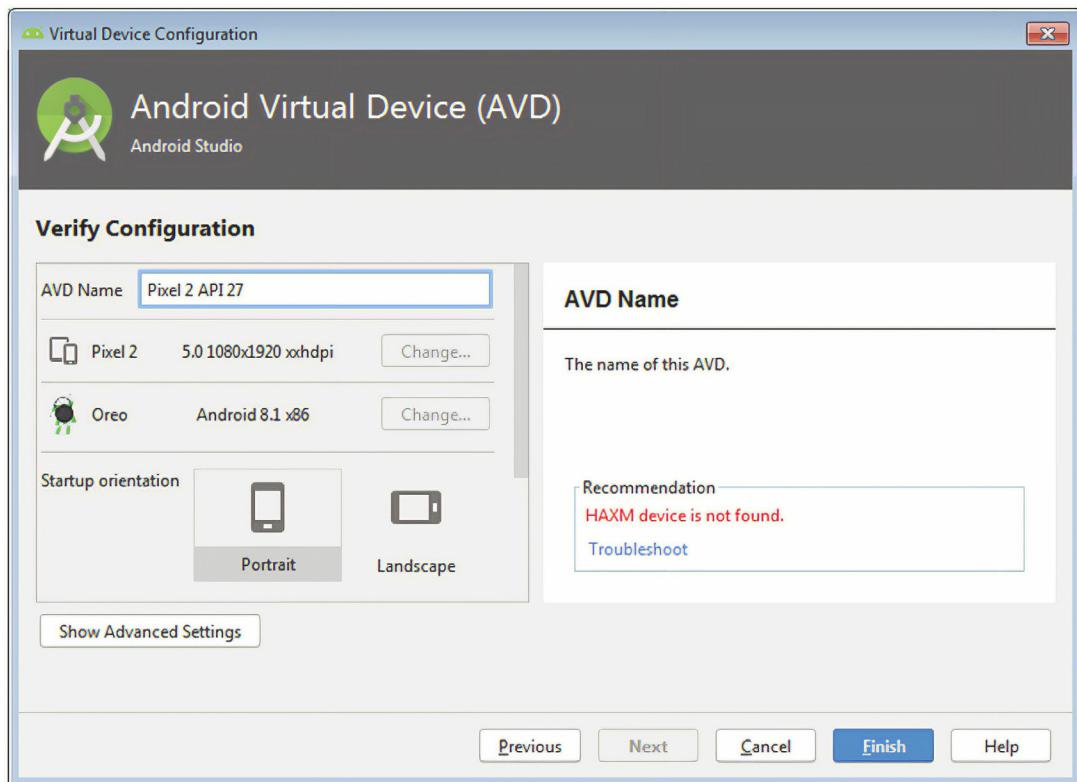


5- Click **Download**. The download process will start as illustrated in the following figure. This file size is about 723 MB. Click **Finish** when you complete the download process.



## 6- Click Next .

If you get a message as illustrated in the below figure in the Recommendation saying : *HAXM device is not found* , just click **Finish**. In the next topic, you will find some techniques that suggest ways to solve this issue.



## HAXM Troubleshooting Techniques:

If your computer has a problem in running your Android Studio phone emulator or in installing the HAXM device, the following topic will help you solve this issue. However, if everything works fine, it is a good idea to read this information, as it helps you gain more experience in solving installation problems and adds more to your technical background.

HAXM (Hardware Accelerated Execution Manager) is a virtualization engine for computers running Intel CPUs. For the best experience on Windows operating system, it is recommended that you use HAXM to accelerate the Android emulator. This engine is a software similar to VGA or sound driver. For any reason, if HAXM is not installed properly on Microsoft operating system, or your operating system does not accept the HAXM driver installation, it could be because its files lack the Microsoft signature file. Please note that the default security settings for any Microsoft operating system is to block installing any driver for any device when the driver files lack a Microsoft signature file. Having no Microsoft signature file, means that these driver files were not tested by

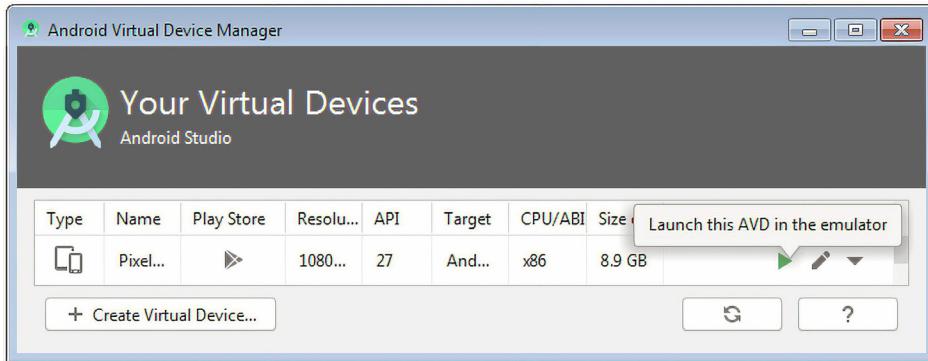
Microsoft and may cause a conflict or damage with any other operation system files. So, if your computer BOIS is already enabled to use virtualization feature, try to use the following techniques to solve this issue:

- 1- Press **Windows** key + **R**, then type **gpedit.msc**
- 2- Click **User Configuration** → **Administrative Templates** → **System** → **Driver Installation**
- 3- In the right side of this console, double click the **Code signing for device drivers**
- 4- Select **Enabled**, and select **Ignore** under the *Option configuration* in the right side, and then click **OK**. Then, close this console. “**Ignore**” option directs the operating system to proceed with the driver files installation even if it includes unsigned files from Microsoft.
- 5-Safe Mode startup. In this step, you must startup your computer in safe mode. If your computer O.S is Windows 7 or 8, follow the following steps:

- **Restart** your computer and press **F8** key in 1 second interval during startup process.
  - Select **Disable Driver signature Enforcement**, then press **Enter**. Your computer will startup normally but without any restriction on installing unsigned files from Microsoft.
- 6-If your computer O.S is Windows 10, you can get the Safe Mode options by observing the following steps:
- Hold **Shift** key, and click **Restart**
  - Click **Troubleshoot**, then click **Advanced options**, and then click **Startup settings**
  - Select **Disable Driver signature Enforcement**, and press **Enter**.

- 7- Disable your antivirus or any third party security software temporarily.
- 8- Open Android Studio IDE . Click Tools → SDK Manager, and then click SDK Tools tab. Uninstall Intel HAXM software by un-checking the : Intel x86 Emulator Accelerator (HAXM installer), click Apply, select it again, then click Apply to install it again. After completing the new installation process, click OK.
- 9- If you have a problem with installing the Intel x86 Emulator Accelerator (HAXM installer), go to <https://github.com/intel/haxm>, and download HAXM for Windows. Then repeat step 7 again.
- 10- Close Android Studio IDE , restart your computer, open your Android Studio, and test your emulator by clicking the Run menu. Then select Run. It will work fine.

Finally, from your **Android Studio**, click **Tools** → **AVD Manager** as illustrated in the following figure. Then click the run green arrow to run your emulator. Then, close the dialog box.



As you know, it is difficult to write a topic about troubleshooting this type of problems because each computer has different hardware, Windows O.S, Windows updates, security software, as well as different settings. Therefore, if you still have this problem with running your Android Studio emulator, you may check the following link:

<https://developer.android.com/studio/run/emulator-acceleration#vm-windows>

Or check <https://stackoverflow.com> web site, and try to find a solution for your case.

Now, you may check Flutter doctor command again. Test flutter doctor command through command prompt or your Android Studio **Terminal** console which is located at the status bar of Android Studio. The run result is illustrated in the following figure.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, v1.9.1+hotfix.6, on Microsoft Windows [Version
     6.1.7601], locale en-US)
[✓] Android toolchain - develop for Android devices (Android SDK version 29.0.2)
[✓] Android Studio (version 3.5)
[✓] Connected device (1 available)

• No issues found!

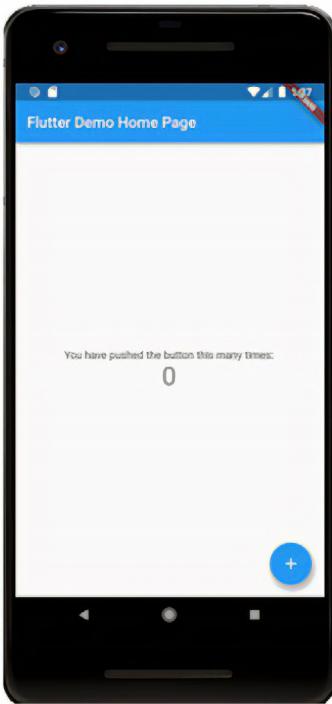
C:\Users\Admin>
```

## Run a Flutter App

In this part of the lesson, you will run the default code of **main.dart** file which includes a Dart code written by Flutter team. This code is only used to test your Android Studio setup and testing Android emulator setup. This file includes a code of counter increment by 1 each time you tap the plus sign button.

To run your app, click **Run** menu, then click **Run**. Also, you may click the Run ➤ button on the tool bar.

The run result will be as follows:



The following is the code of **main.dart** file:

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

  void _incrementCounter() {
    setState(() {

      // This call to setState tells the Flutter framework that something has
      // changed in this State, which causes it to rerun the build method below
      // so that the display can reflect the updated values. If we changed
      // _counter without calling setState(), then the build method would not be
      // called again, and so nothing would appear to happen.
      _counter++;
    });
  }
}
```

Replace the command `_counter++;` with `_counter--;` then run your app using **Run** button.

Then, test your app again.

## What is role of hot reload button ?

Flutter's hot reload feature helps you quickly and easily experiment, build UIs, add features, and fix bugs. Hot reload works by injecting updated source code files into the running Dart Virtual Machine (VM). After the VM updates classes with newer versions of fields and functions, the Flutter framework automatically rebuilds your app structure, allowing you to quickly view the effects of your changes.

Example:

In the previous code, replace in your Dart code the last line of code as illustrated in the following code:

```
child: Icon(Icons.add),
```

with

```
child: Icon(Icons.favorite),
```

Then, click the hot reload button  , and check the changes in your phone emulator. Also, you may change this button using the following suggestions:

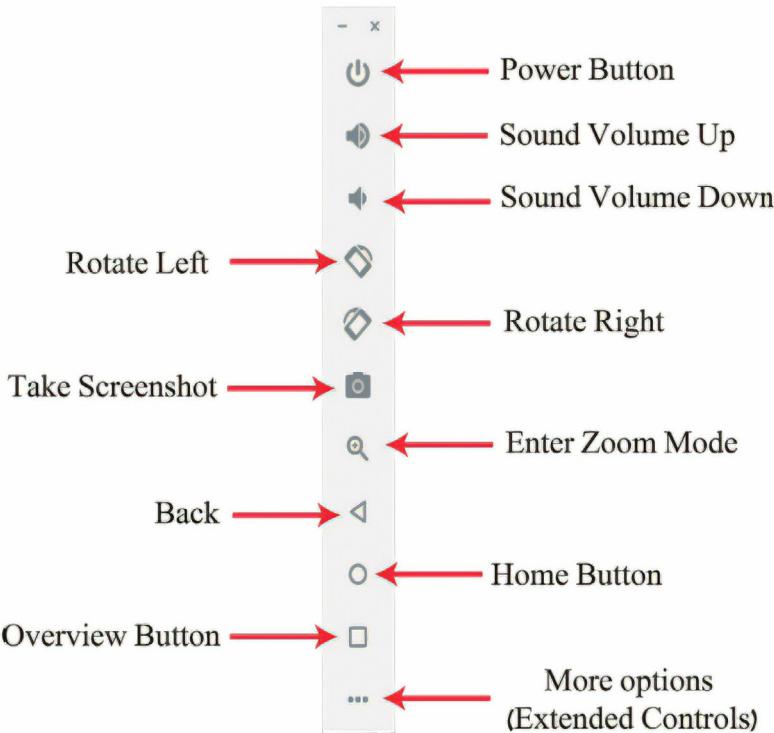
```
child: Icon(Icons.battery_charging_full)
```

Or

```
child: Icon(Icons.call),
```

## The Emulator Toolbar Options

The emulator toolbar provides access to a range of options relating to the appearance and behavior of the emulator environment as illustrated in the following figure:



For example, the following is a description of the role of some of these buttons:

**Overview** : Simulates selection of the standard Android “Overview” button which displays the currently running apps on the device.

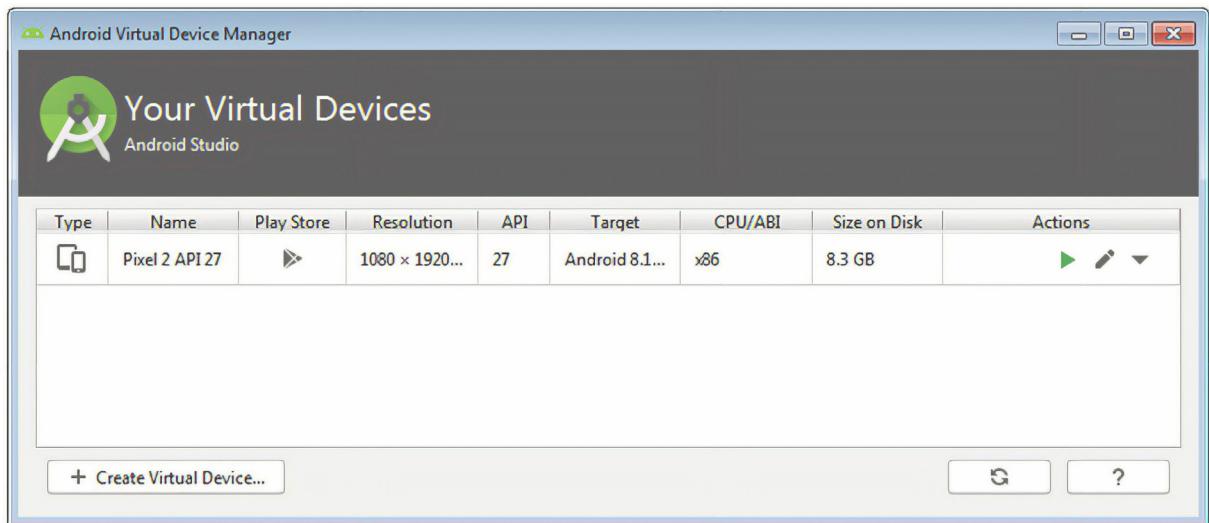
**More (Extended Controls)**: Displays the extended controls panel, allowing for the configuration of options such as simulated location and telephony activity, battery strength, cellular network type and fingerprint identification.



It is recommended to click the update link in any notification message you may get to install your Android Studio updates, especially when you start up your Android Studio. This keeps your Android Studio IDE up to date. To check for updates manually, click **Help → Check for Updates..** (on Windows) & **Android Studio → Check for Updates** (on macOS).

## Manage Android Emulators

If you want to add another Android emulator to test your app with different type of emulators, delete, stop, or run exiting emulators, click **Tools → ADV Manager**, or the ADV Manger button of the tool bar. Then, you will get the following dialog box:

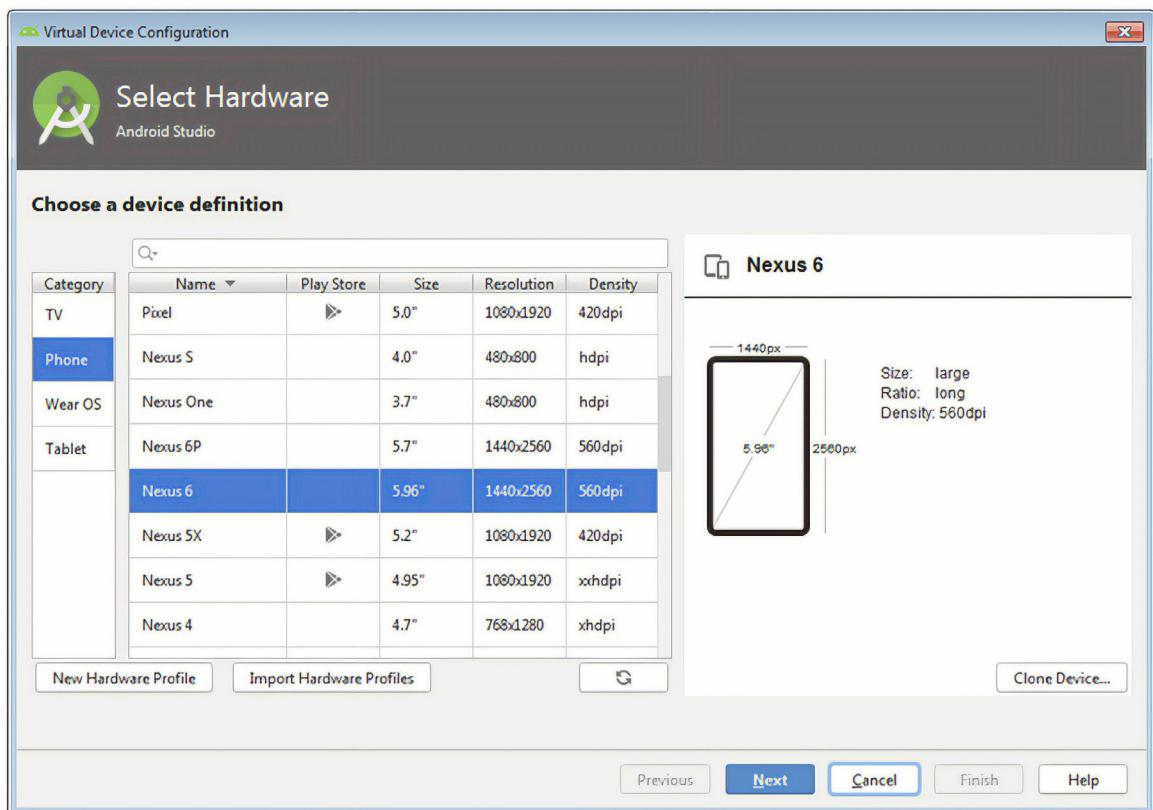


As illustrated in the previous figure, you already have Pixel 2 as Android emulator. Now, you will install another Android phone emulator called Nexus and know why it is better than using Pixel emulator in testing your Flutter apps.

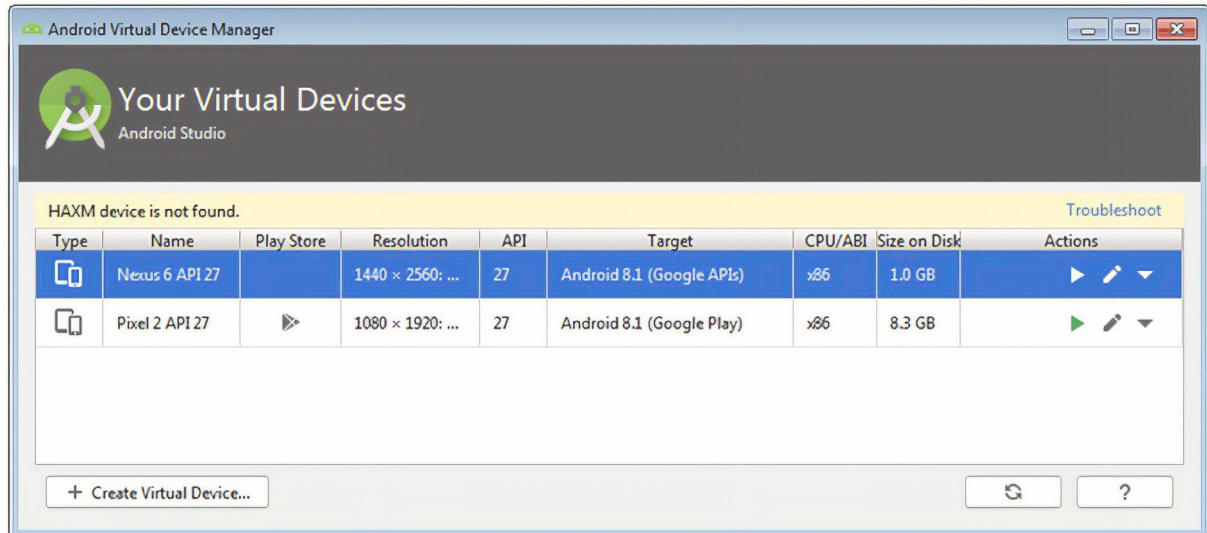
You will repeat the previous steps to add a new Android emulator to Android Studio as follows:

1- In the Android Virtual Device Manager dialog box, click **Create Virtual Device**

2-Then, **Nexus 6**, and click **Next**



3- Select **Oreo** (Android 8.1), and click **Download**. After completing the download process, Click **Next**, you will get a list of the virtual devices or Android emulator which you can run and test your Android apps.



4- Click the pencil icon which is related to the **Nexus 6 API 27** to edit this emulator configurations. You will get the following dialog box:



In this type of emulators, you can select how graphics are rendered in the emulator. You have three choices as follows:

- **Hardware** - Use your computer graphics card for faster rendering (app running result).
- **Software** - Emulate the graphics in software, which is useful if you're having a problem with rendering in your graphics card.
- **Automatic** - Let the emulator decide on the best option based on your graphics card.

We recommended you to use **Nexus** emulator in your test environment and this course labs, because you can select **hardware** graphic type, and get a faster rendering (run) result.

While, if you use **Pixel** emulator, you can't change its graphic setting. Only the automatic choice is available.

Now, we can say, "congratulations"! You have completed all of the steps in Windows setup, and you are now ready to start building Flutter apps for Android development.

## Installing Flutter on Mac

If you have a Mac computer, you can install Flutter SDK , Android Studio and another prerequisite software to create a Flutter project. Also, you need to make some additional configurations depending on the type of your Mac operating system.

The Flutter team has created a comprehensive guide to install Android Studio and Flutter SDK and how to configure them to make the Android Studio ready to create a Flutter project. In the following link, you may select the operating system where you have Windows , Mac, or Linux, then follow the installation guide:

<https://flutter.dev/docs/get-started/install>

The Mac installation guide focuses on installing the main requirements to create a Flutter project as follows:

1. Install the Flutter SDK
2. Install Xcode and command-line tools
3. Install Android Studio
4. Set up the iOS simulator

### What is Xcode ?

Xcode is an IDE – an integrated development environment – created by Apple for developing software for macOS, iOS, iPhone, iPad, Apple Watch, and Apple TV. It is

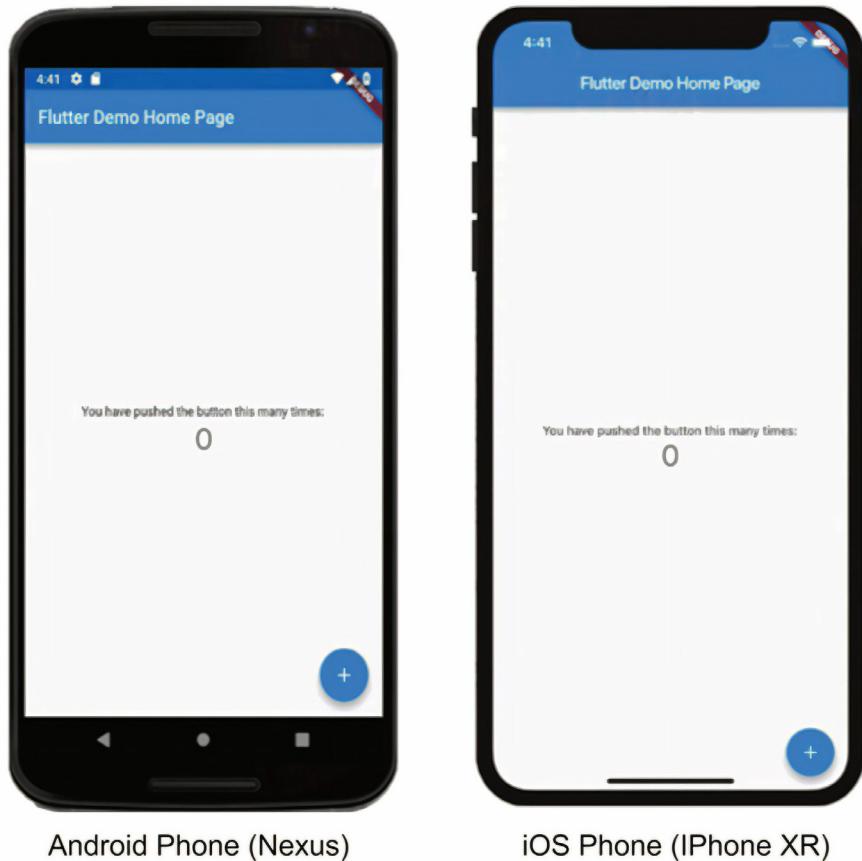
the only officially-supported tool for creating and publishing apps to Apple's app store, and is designed for use by beginners and experienced developers. With Xcode, you'll be able to run Flutter apps on an iOS device or on the simulator. The Xcode is available as a free download from the Mac Apple Store.

## Test Your Flutter App on iOS Phone with Windows O.S

If you installed Android Studio on your Mac computer successfully, then when you install Xcode, another application gets installed by default, and that is the iOS simulator. When you run iOS simulator, you can test the Flutter app on iOS simulator to know how your Flutter will work on iPhone or other iOS devices. However, you must install Android emulator from Android AVD Manager (Tools → AVD Manager) to have an Android emulator such as Nexus or Pixel Android phones to test your Flutter app workflow and see what it looks like.

This means, when you use Android Studio on Mac computer, you can have Android and iOS emulators to test your Flutter apps.

The follow figure displays what the default flutter app looks like with Android phone (Nexus 6) and iOS phone (iPhone XR).



As you have seen in the previous figure, the Flutter app UI (user interface) in the two simulators is almost the same. Only the title bar, which contains the name of the application or the logo is different , where the default format is centered with iOS phones, while with Android phones, it is on the left margin.

**The question is, whether you can test your Flutter app code on iOS phones emulator without the need to use a Mac computer, if your computer has Microsoft Windows operating system?**

This is an important question, because as a Flutter application developer, selecting this programming language gives you the ability to create or maintain Android and iOS apps, using the same codebase. Also, you must know what your app user interface will look like on Android or iOS devices before publishing it to an Apple or Google store.

If your computer operating system is Mac, you can easily, through Android Studio, install different types of Android emulators , and by default, when you install Xcode from Apple store, an iOS emulator will be configured and added automatically to your Android Studio emulators list.

Now , we can say, “yes,” even though you have a Microsoft operating system, you can test your Flutter app and see what it will look like by performing the following steps:

1- Go to <https://github.com>

2- Click **Sign Up** to create a GitHub account. Fill out your information , and then click **Next: Select a plan**

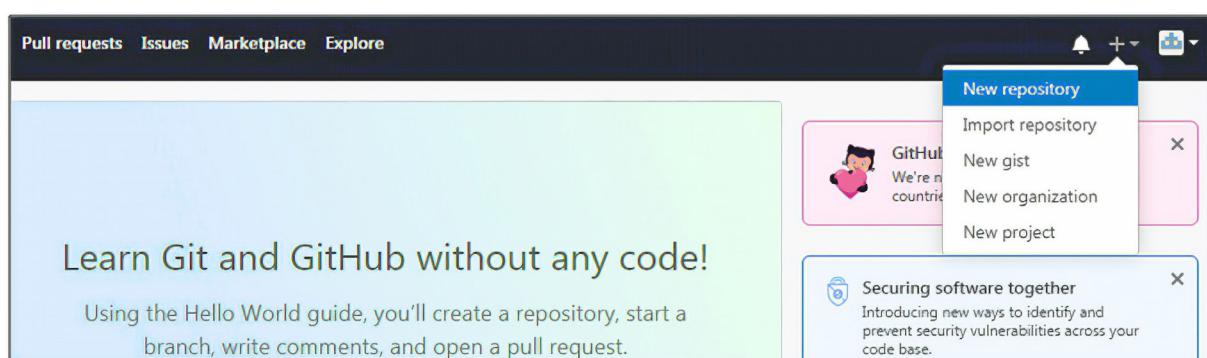
3- Click **Choose Free**

4- Answer the complete setup questions, and then click **Complete setup**.

5- You will get a message asking you “ **Please verify an email address** ”. Login to your email address, open GitHub email . Then click **Verify email address**.

6- Login to GitHub again using your username and password.

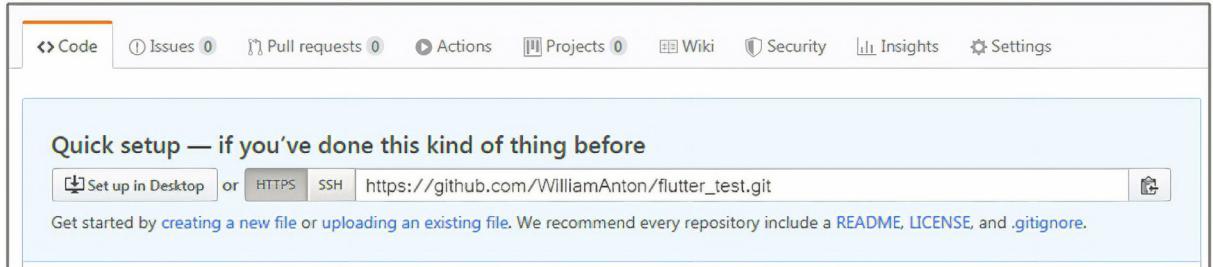
7- Click the plus sign on the upper right corner as illustrated in the following figure. Then click **New repository**



**8- Name your Repository name : flutter\_test**

Also, select your repository type : **Private**, and then click : **Create repository**

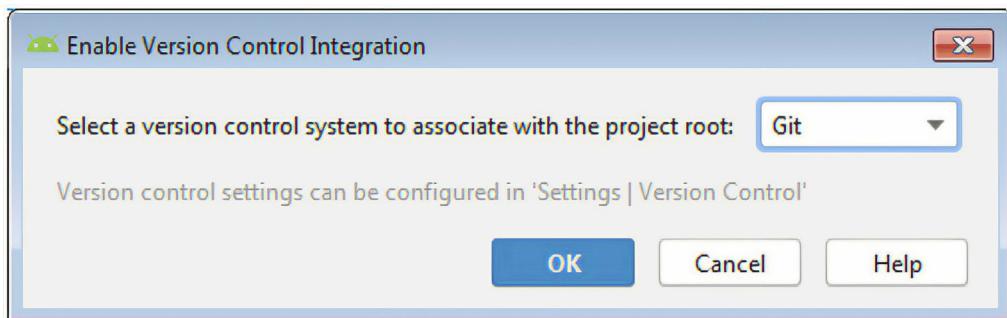
9- Copy your repository URL on GitHub web site, and paste it in a Notepad file for later use.



10- Open your Android Studio IDE

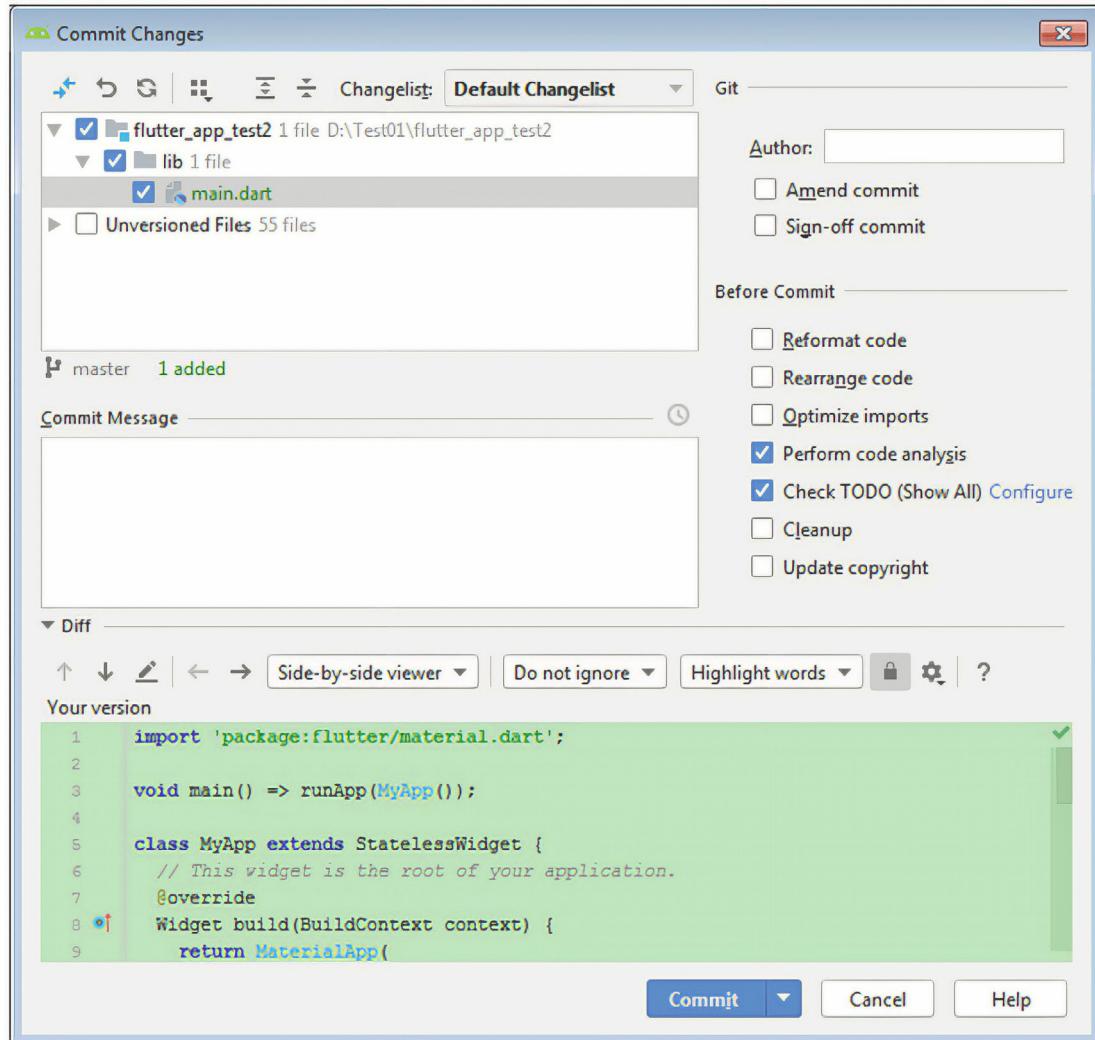
11- Click **VCS → Enable Version Control Integration...**

12- As illustrated in the following figure, select **Git**, and then click **OK**.



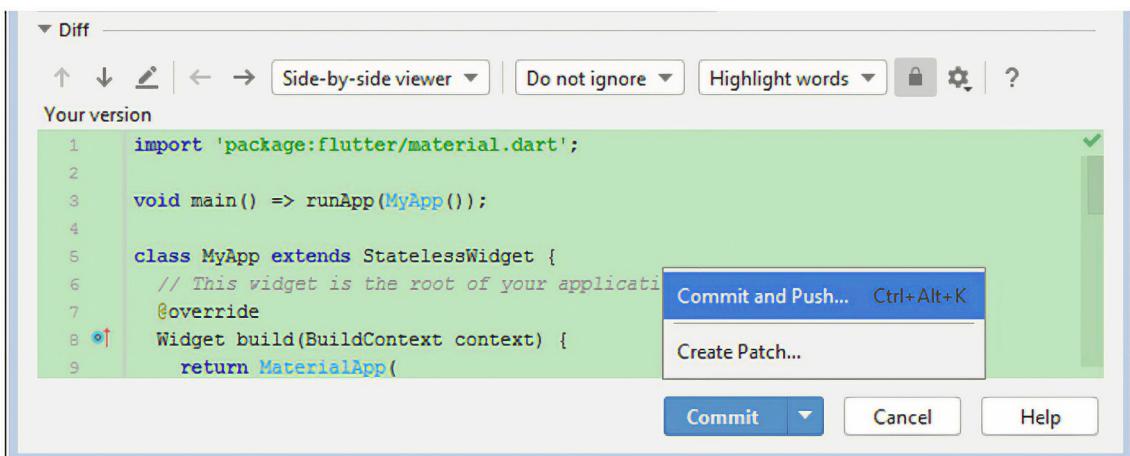
13- Click **VCS → Git → +Add**

14- Click **VCS → Commit ...** Then you will get the following dialog box:



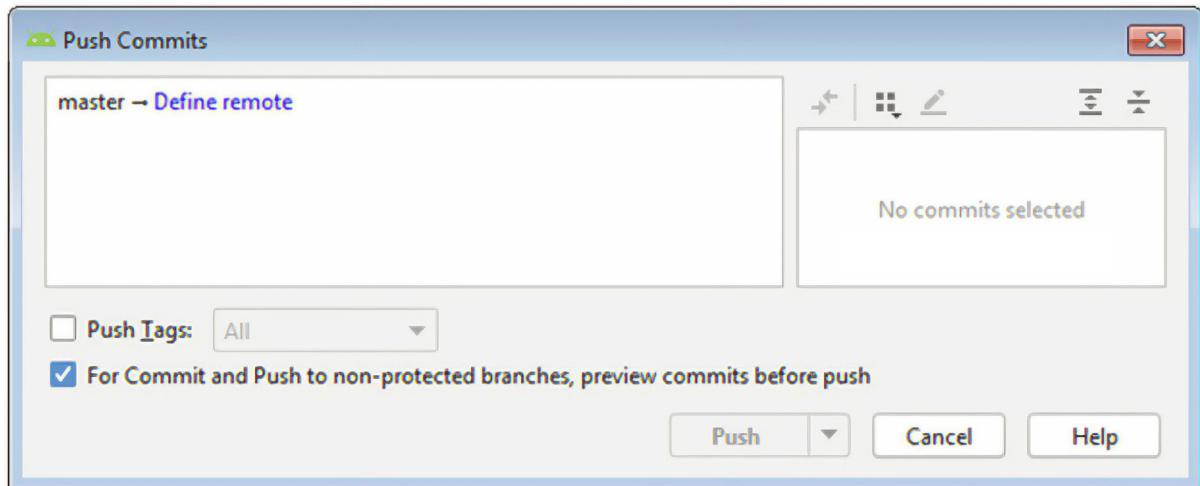
15- Inside the **Commit Message** text box type the following text : **initial commit**

Then, click the arrow which is beside **Commit** button, and then select **Commit and Push ..** as illustrated in the following figure:

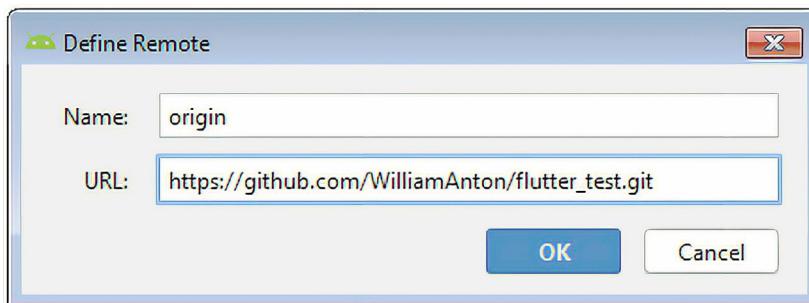


16- If you are asked to enter your GitHub user name or email address, enter them, and then click **OK**.

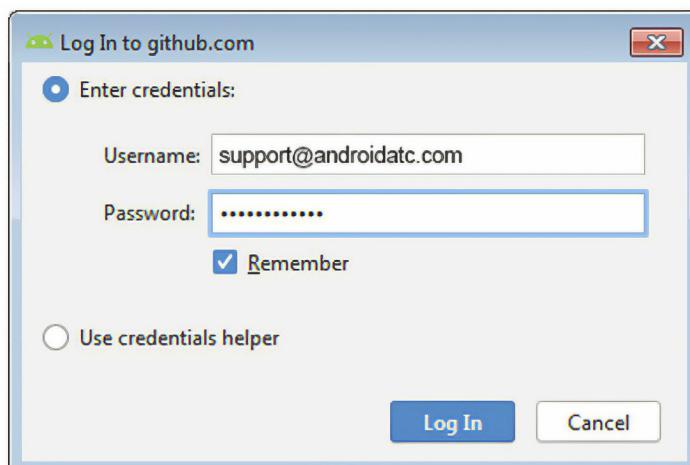
17- You will get the following figure. Click **Define remote**



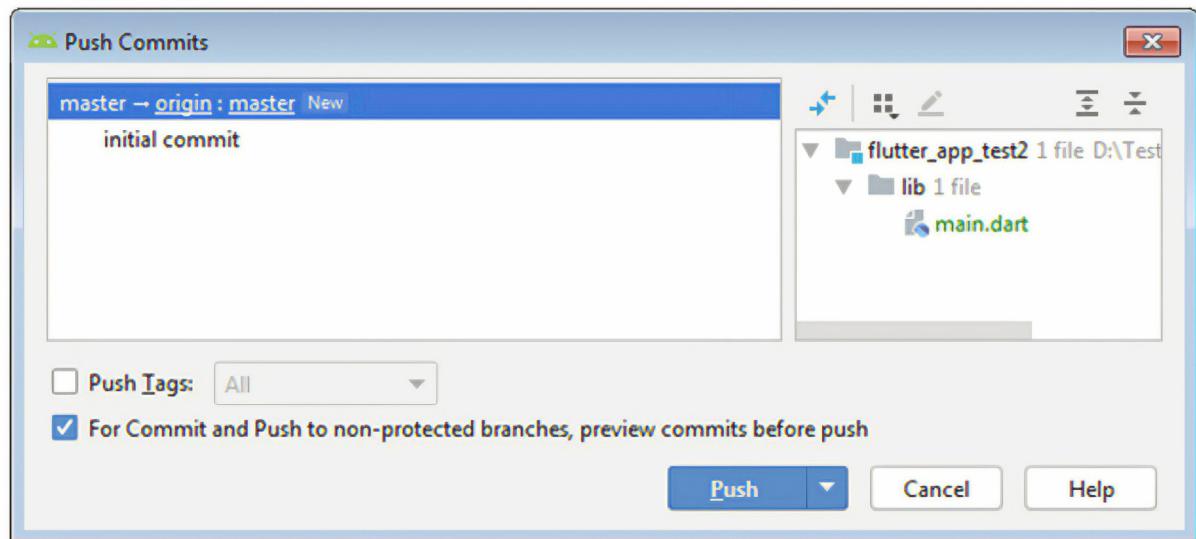
18- In the following dialog box, enter your GitHub repository URL as illustrated in the following figure, then click **OK**.



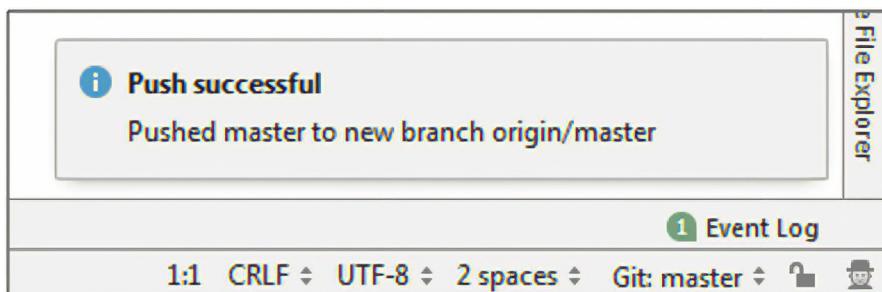
19- You will get the following dialog box asking you to enter your GitHub user name and password. Enter your credentials. Then click **Log In**.



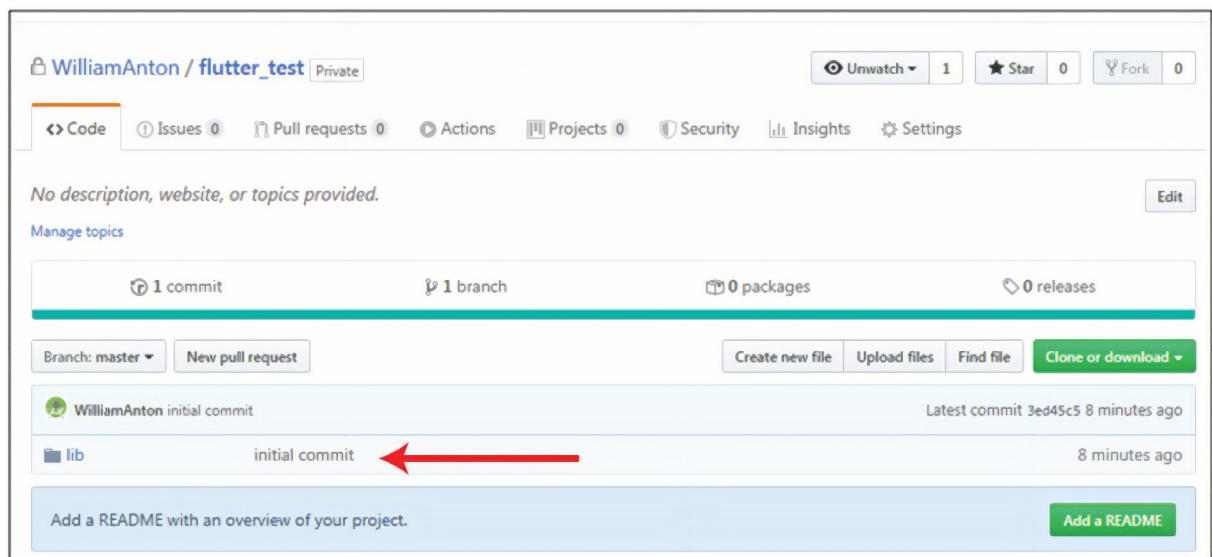
20- Then, you will get the following figure. Click Push



21- You will get the following notification in your Android Studio notification area.



22- As you see in the following figure, your app has been pushed to your GitHub repository



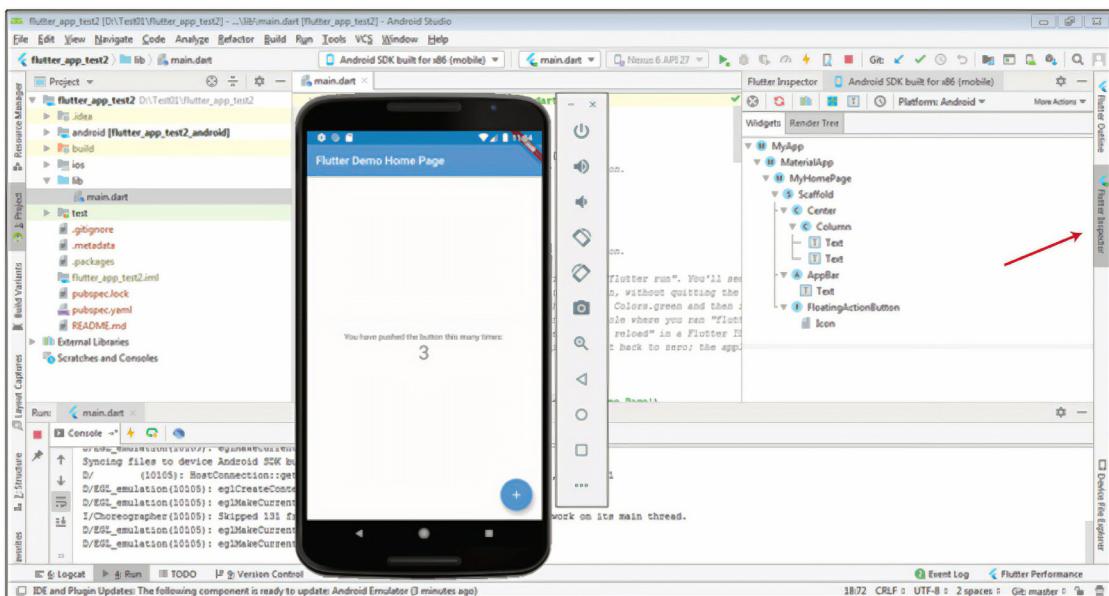
If you click **lib** folder, you will get the Dart files as illustrated in the following figure:



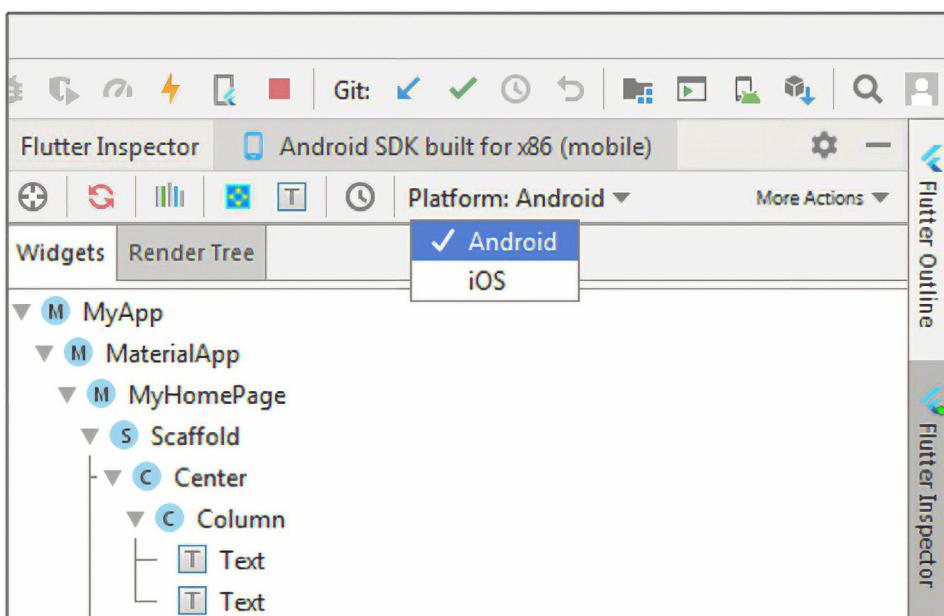
If click the **main.dart** file, you will get the Dart file content

23- Run your Flutter app using your Android emulator (important step).

24- After running your app, click **Flutter Inspector** on the right side of Android Studio as illustrated in the following figure:

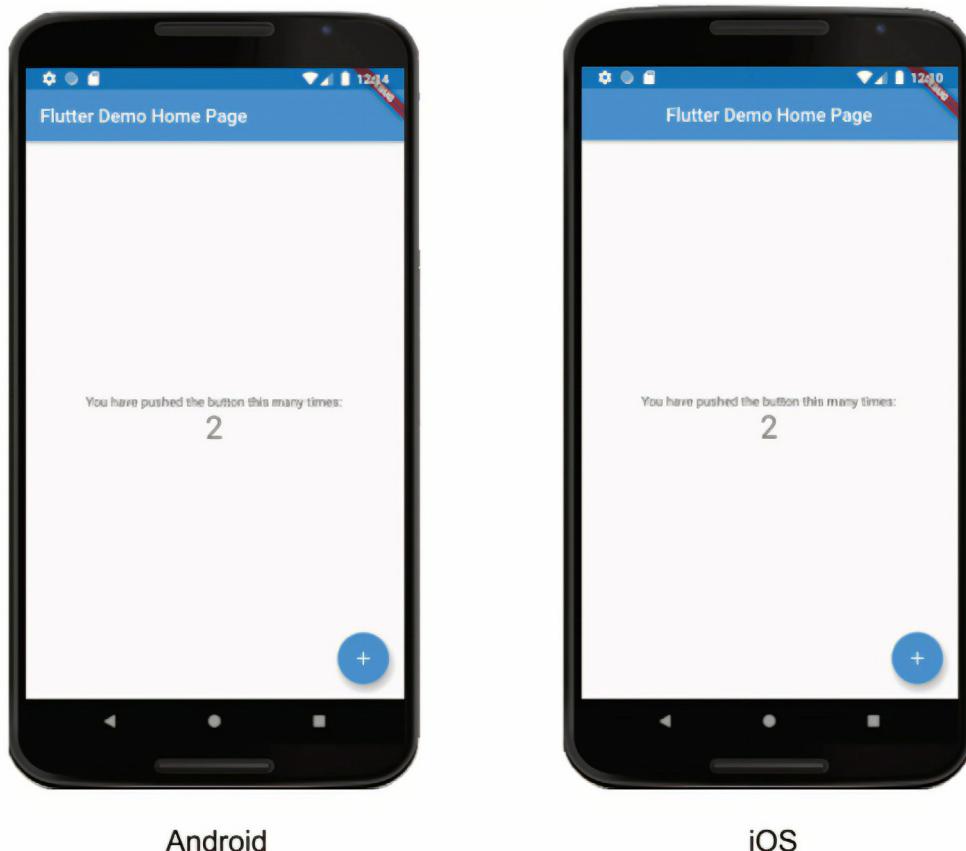


25- Click **Platform Android**, and then select **iOS** as illustrated in the following figure:



Now, check your emulator. You will find your app navigation bar content has moved to the center of your emulator. This means that your emulator run output of your app will give you the same result as it runs on an iOS device. You can switch between Android and iOS, and check what your app looks like .

The following figure displays your code in the same emulator but with two different platform choices (Android & iOS):



This means if you have a Microsoft computer, you don't need to buy a Mac computer to test your Flutter apps on iOS phones. The previous technique gives you an idea about how your app will run and what its user interface looks like.

If you have a Mac computer, you can test your Flutter app on Android and iOS emulators without the need to follow the previous technique when you want to test your app on an Android emulator.

## Android Studio Sugar and Spice

In this part of the lesson, you will perform some techniques in using Android Studio IDE which will help you when you start in creating your Flutter apps. Here, you will configure or customize some Android Studio settings which may make it more comfortable to use and work with some consoles and features which makes Android Studio more helpful in developing and troubleshooting Flutter apps.

### Android Studio Appearance

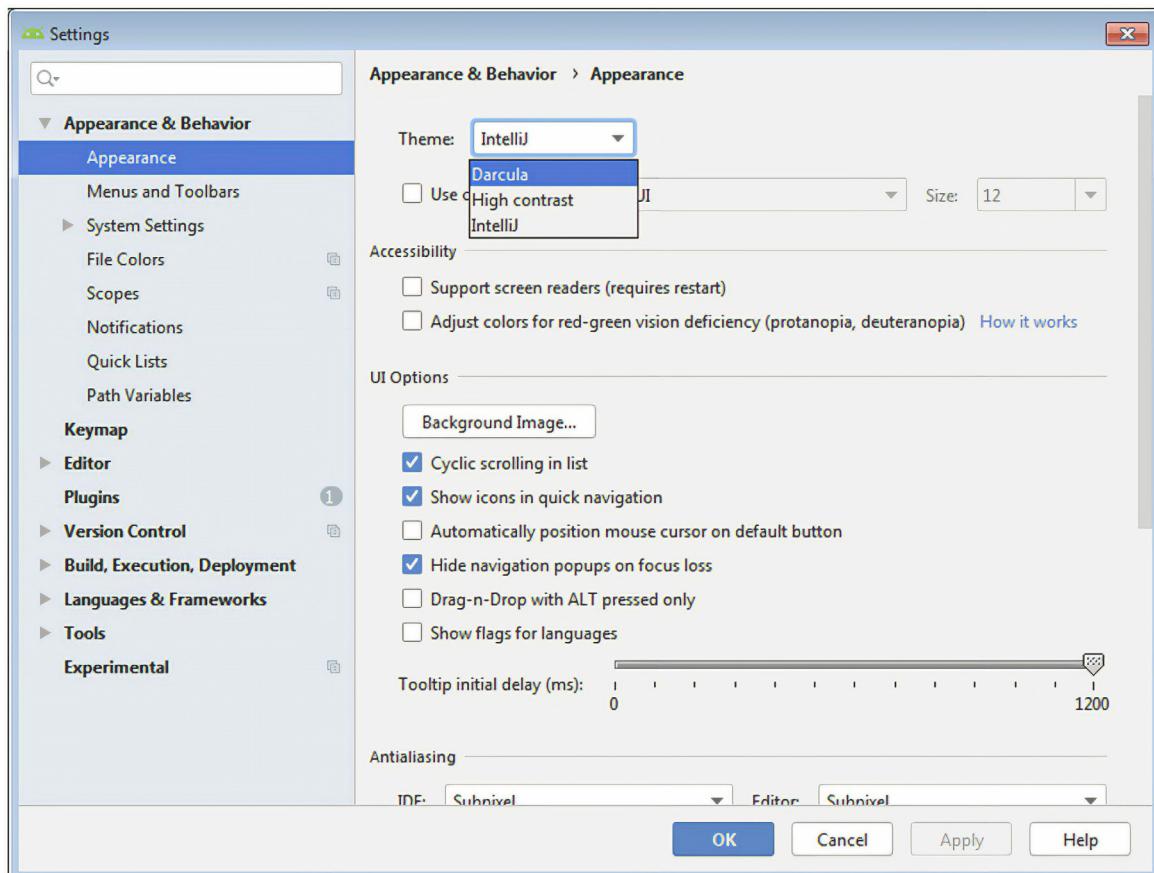
The default Android Studio interface appearance has a white background. Some developers prefer to add a little bit of a contrast in the code, so you can read the code with less effort . This type of configuration is personal and it is good to check if you personally find it more suitable.

To change the theme of an Android Studio to have a dark background, perform the following:

1- If you have Microsoft Windows, click **File** → **Settings**, then click : **Appearance & Behavior**, then select **Appearance** on the left side.

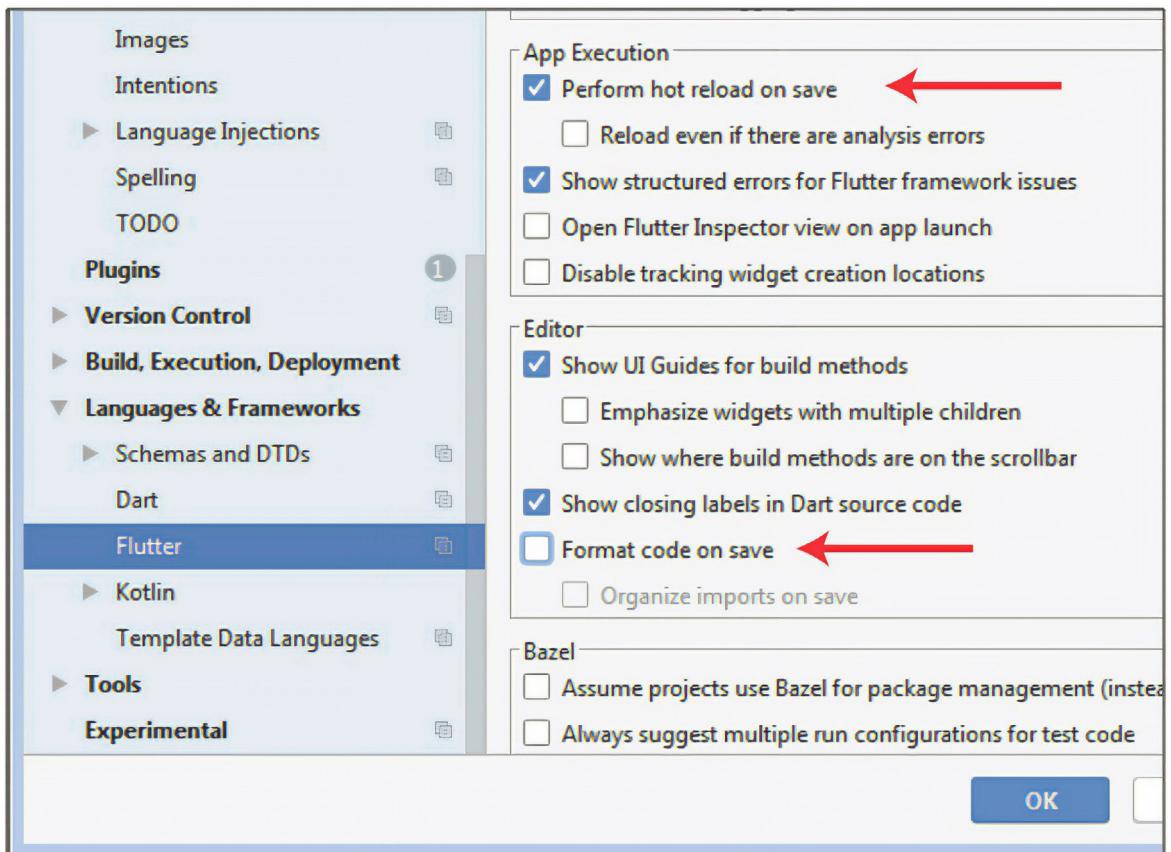
If you have Mac, click **Android Studio** → **Preferences**. Whether you have Microsoft or Mac you will get the same dialog box.

2- The default theme is : IntelliJ . You can select any of the following three themes as illustrated in the following figure. Select **Dracula** theme, and click **Apply**. (this is an option).



Also, you may change the font size by clicking the **Editor** option on the left side of the previous dialog box and then clicking **Font**. Here, you may also select the font type and size you are most comfortable with. Then click **Ok**.

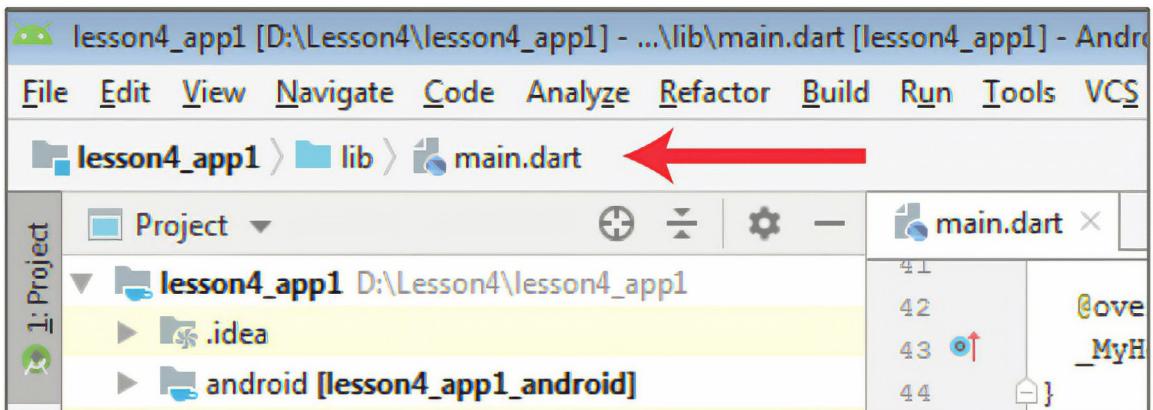
As illustrated in the following figure, make sure **Perform hot reload on save** option is checked, which is selected by default. Also, select **Format code on save**, which means that every single time you will click the save button or select the **Save All** option from the **File** menu, the code will be reformatted in the standard format. Be sure these two choices are selected. Then click **OK**.



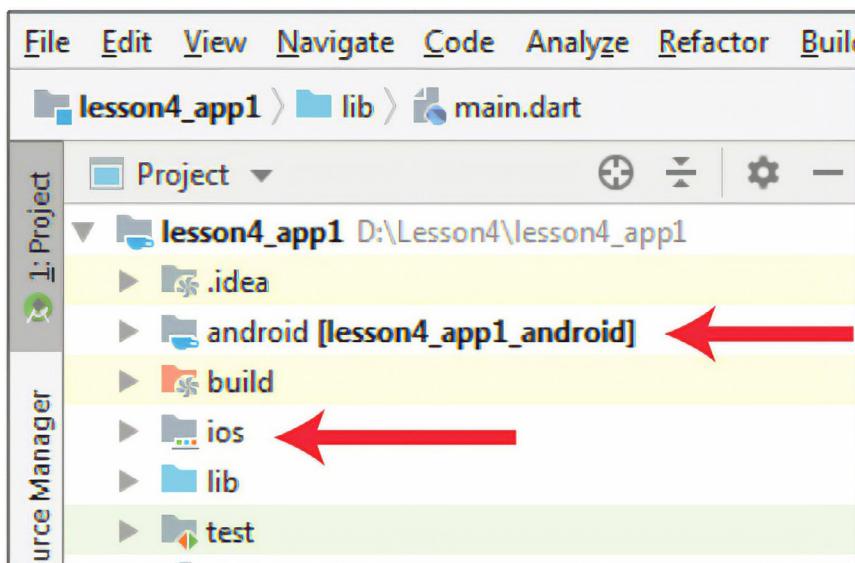
To test the last choice, for example, move some code to the right side using Tab or space bar key, then click **File** → **Save All**, you will find that the code has returned to its original location or format. This makes the code easier to read. Also, you can get the same result if you right click the code. Then select **Reformat Code with dartfmt** of the shortcut menu.

## Navigation Bar & Project Structure

Android Studio is a powerful development tool which makes your experience in developing and coding a lot more enjoyable. For example, check the navigation bar which is at the top of Android Studio, and as illustrated in the following figure, it is basically showing you the location of the current file relative to your Flutter project. This helps you follow the project files sequence. Also, if you click on any part of this sequence such as **lib**, you will see the folder content.

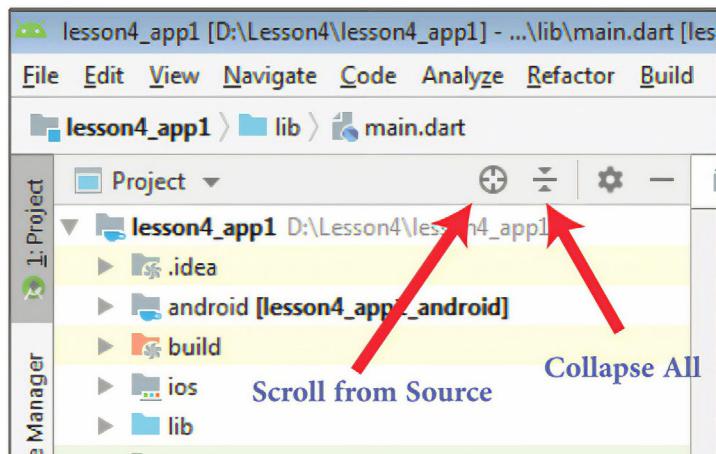


In the project pane, as illustrated in the following figure, you will find a folder for your Android app and another folder for your iOS app.



When you are creating your Flutter app using Android Studio, it will automatically create your iOS and Android apps in the background, and update them as you create your Flutter app.

Also, you may use **Collapse All** button as illustrated in the following figure to collapse all the open project files and folders. In addition, when you click the **Scroll from Source** button, you will locate which file of the Flutter project files are open or show you where you currently are inside your project structure.

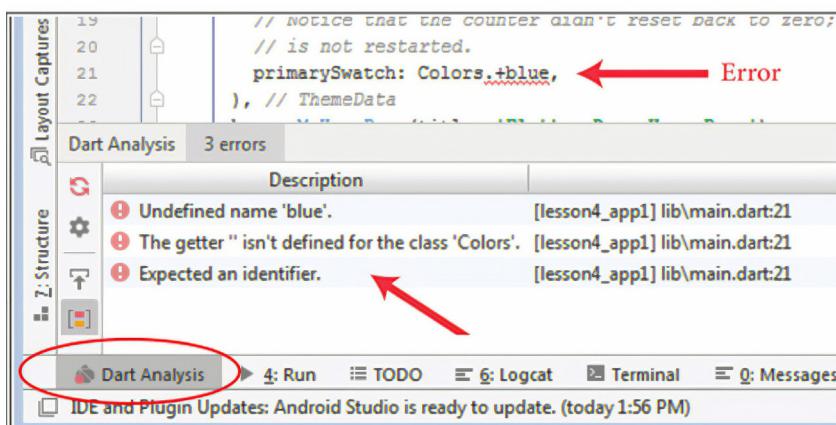


Also, if you navigate the Dart code, you will find a blue square at the margin as illustrated in the following figure. This helps you to locate the part of the code which is related to the blue color. If you change the color to red, this blue square will turn red.



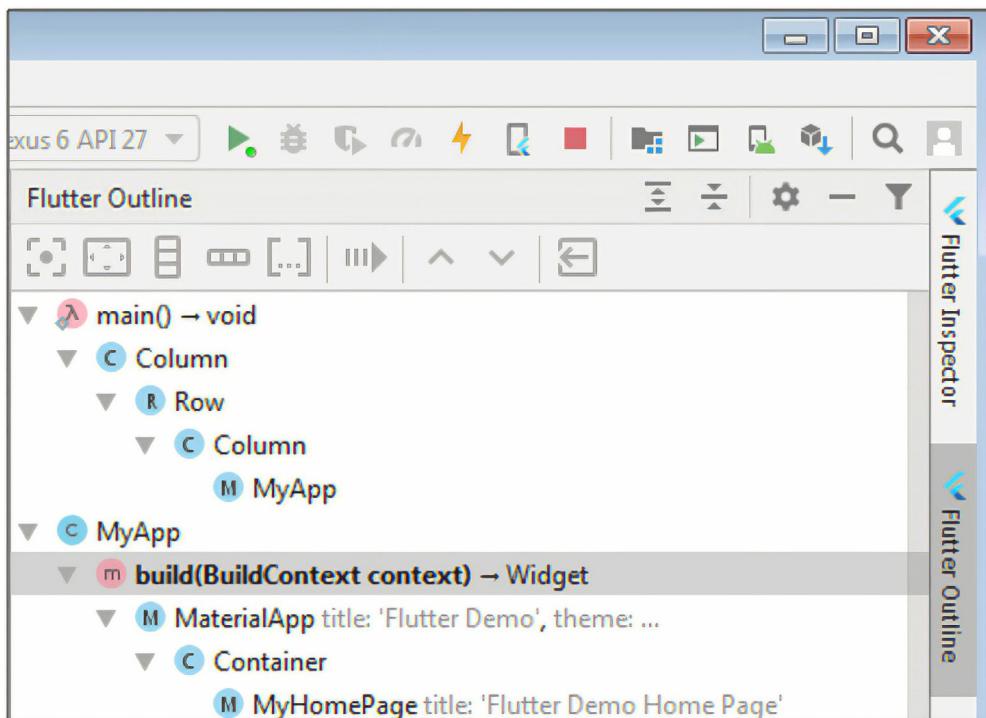
## Dart Analysis

Let us make a small error in this code by replacing the previous line of code which is related to the blue color with the following code: `primarySwatch: Colors.+blue`, here I added only a "+" sign. Then, I clicked run button. If you click the **Dart Analysis** at the status bar of an Android Studio as illustrated in the following figure, you will see the error description and also see which line number of the code the error is in. Also, if you double click the red error message, the mouse pointer will move automatically to the error location. In addition, in almost cases the description field in the Dart Analysis console gives you an idea about the reason of the error.



## Flutter Outline

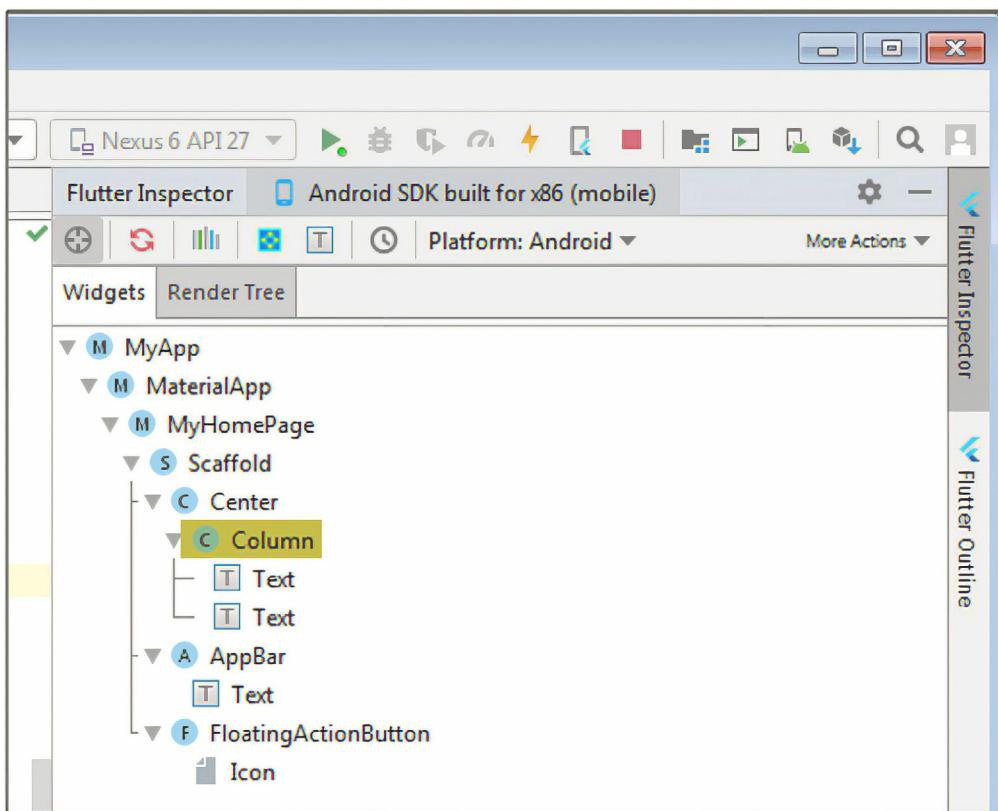
The Flutter outline is on the right side of Android Studio as illustrated in the following figure. It displays the tree structure of your codebase. This outline is really helpful especially when you have a long file full of code and you need to locate a specific part such as a text, row, column, an image , a button etc... So, you can just click on the part you need in the Flutter outline console, and it will highlight that part inside your code on the left side. This is really useful when you are navigating around your code and checking what your code looks like at a glance.



## Flutter Inspector

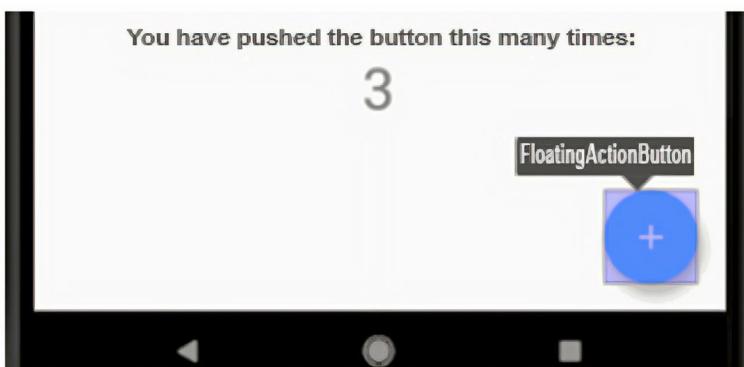
The Flutter Inspector is on the right side of an Android Studio as illustrated in the following figure. It is a powerful tool for visualizing and exploring Flutter widget trees (such as text, buttons, and toggles). The Flutter framework uses widgets as the core building block for anything from controls, to layout (such as centering, padding, rows, and columns). The inspector helps you visualize and explore Flutter widget trees, and can also be used for the following:

- Understanding existing layouts
- Diagnosing layout issues



To test the effect of Flutter inspector, use the same code of lesson 4 and make sure that your app is running on a device (emulator or physical device).

For example, if you click on the *FloatingActionButton* on the Flutter Inspector, you will notice a highlight on your emulator as illustrated in the following figure. You will also get all the properties of this button such as color, font size, and other properties on the lower part of the Flutter inspector pane.



Also, the **widget mode** button on the Flutter Inspector, as illustrated in the following figure has an amazing and important role. This button allows you to select anything you see in the running phone emulator to locate it inside the code. For example, scroll up the Android Studio and click the mouse pointer on the first line of your code, click the widget mode

button once, then click the text “You have pushed the button these many times.” on your phone emulator, your Android Studio will scroll down to show you the part of the code which includes this text, or it will highlight to you the part of your code which actually created this part (text). To test another part of your app, just click the lens icon on your phone emulator, and then test another widget such as the floating button.



Also, if you click the **Debug Paint** button on the Flutter Inspector, you will add on your phone emulator visual debugging hints to the rendering that displays borders, padding, alignment, and spacers. This helps in designing and improving your app interfaces.

For more information about Flutter inspector tool, check the following web site:

<https://flutter.dev/docs/development/tools/devtools/inspector>

## Run your Apps on a Hardware Device (Physical Phone)

When you build a Flutter app, you use an Android or an iPhone emulator to test your app UI (user interface) and its work flow. Also, it is important to always test your app on a real Android or iPhone device before publishing your app on Apple or Google store, and releasing it to end users. In addition, using a physical device during the testing process reduces using your computer hardware. It avoids some problems with running or configuring phone emulators in case you have a computer with low hardware specifications (RAM and CPU speed).

This part of this lesson describes how to run your Flutter app from Android Studio directly on a connected Android or iPhone device.



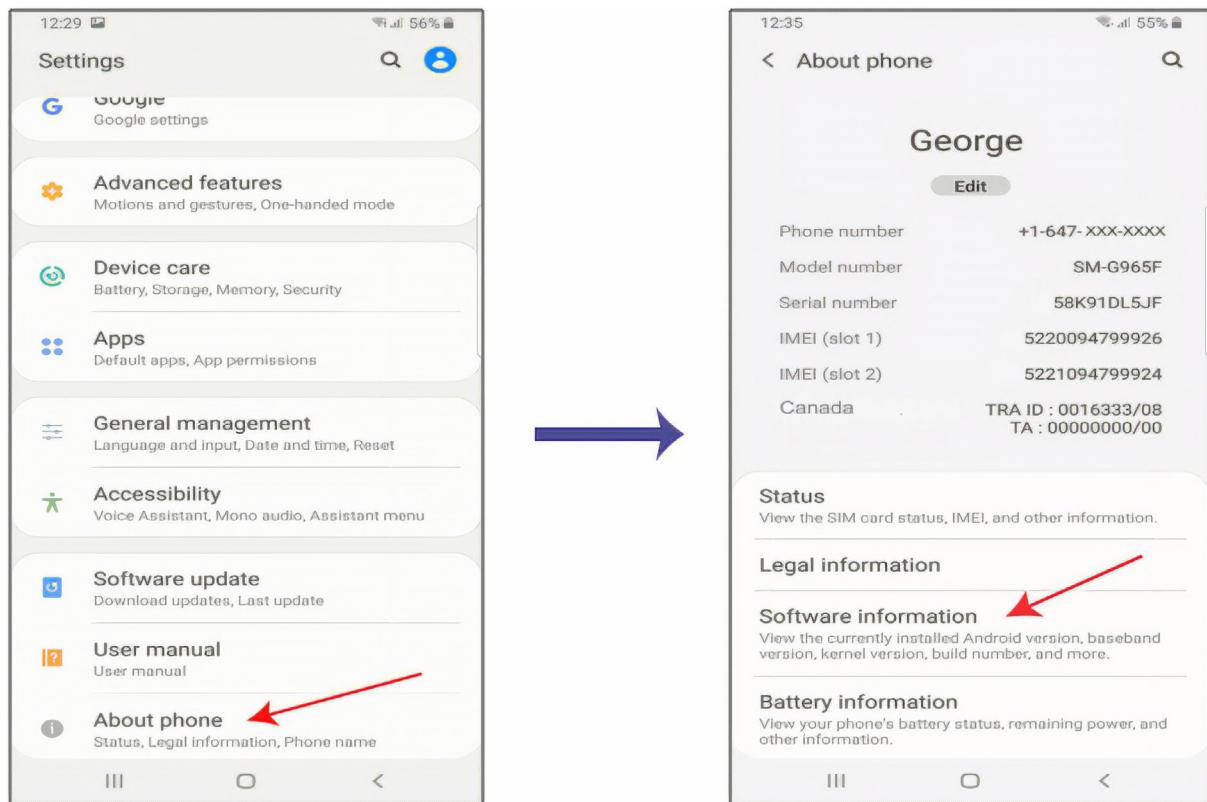
Keep in mind, you cannot test a Flutter app on a connected iPhone device if your computer O.S is Microsoft Windows. However, if your computer is Mac, you can test your app on a connected Android or iPhone device.

## Run your Flutter App on Android Phone

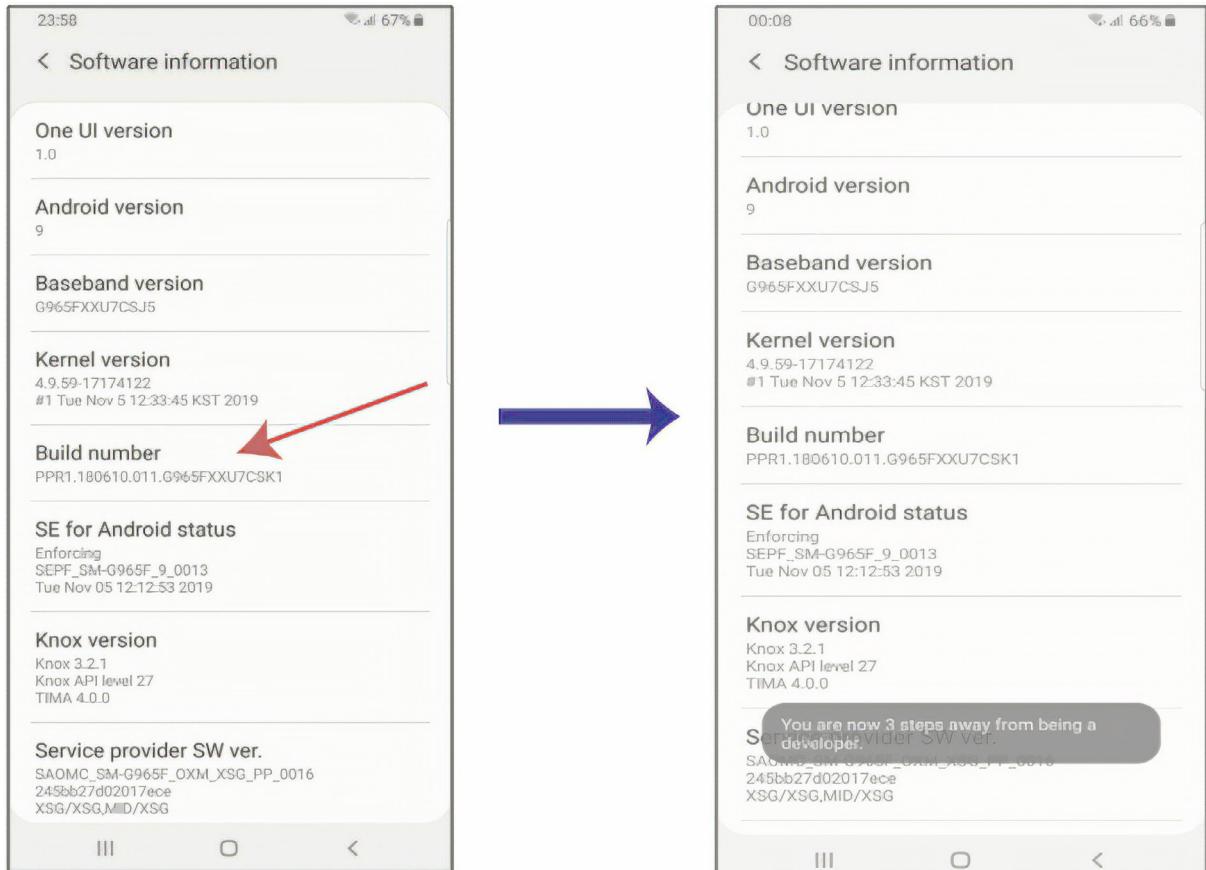
You can run your app from your Android Studio directly to your Android phone if you connect it to your computer through a USB cable. Also, any change in your Dart code can be updated immediately on your phone if you click the hot reload button on your Android Studio as well as see these changes on your phone directly.

The following steps display how to run your app on an Android phone (Samsung):

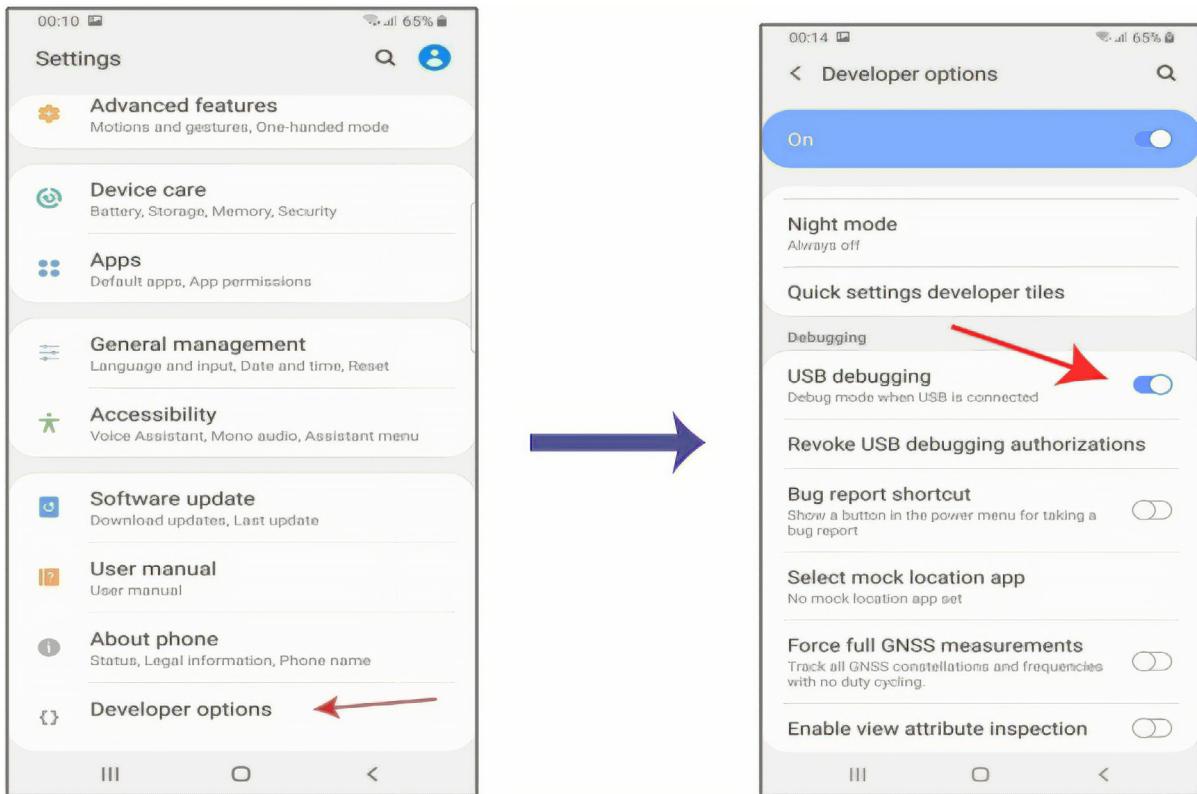
1- On your Android phone, go to **Settings** → **About Phone** → **Software Information** as illustrated in the following two figures:



2- Press 7 times on the **Build number** till you get the following message: " Developer mode has been turned on." as illustrated in the following figure:



3- Go to Settings, then you will find **Developer options** has been added. Select **Developer options**. Be sure this option is **On**. Then enable **USB debugging** as illustrated in the following figures:



4- Select **OK** for this message: "Allow USB debugging?" as illustrated in the following figure:



5- Now, your Android phone is ready to be connected to your computer using a USB cable and run your Flutter app. Before following this step, it is important to pay attention to the following note:

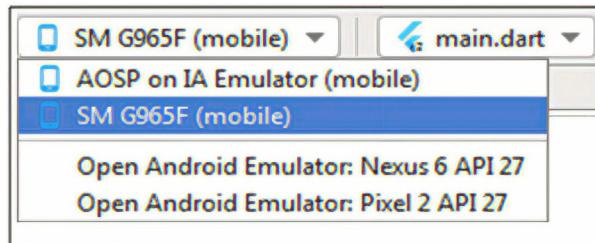
Not all USB cables are created equally. Some cables can transfer power without being able to transfer data. So, even if you see your device charging, it doesn't mean that the cable can transfer the data to allow your phone to be recognized by your computer. Ideally, find the original cable

that came with your device. This usually allows both charging as well as data transfer.

Now, connect your phone through a USB cable to your computer, and check the computer icon on your computer. If your phone appears as storage on your computer, this means that your phone is connected successfully to your computer and it can share data with it.

If you are sure that your USB cable is a good cable to transfer data, and you got a message on your computer notification area displaying that there is a problem in configuring your phone driver with your computer operating system, this means you that you must install your phone USB driver (software) to connect your phone to your computer using a USB cable. To do this, just type on Google search “install Samsung USB driver for Windows”, or use your Android phone type, and then install this driver on your computer. Disconnect, and then connect your phone again to your computer, and accept any warning messages that appear on your phone regarding connecting your phone through the USB.

6- Now, check your emulators list on your Android Studio. You will find your phone name (SM :Samsung) as illustrated in the following figure:



7- Click **Run** on your Android Studio. Then your Flutter app will run on your connected Android phone as illustrated in the following figure:



All Android settings which you applied before on Android Studio Android emulator can be applied on your connected Android phone such as the Hot Reload, and Enable Select Widget Mode (available at the Flutter Inspector Window).

## Run your Flutter App on IPhone Device

To run your Flutter app on your connected IPhone device, your computer operating system must be Mac. To run your Flutter app on your connected IPhone using Android Studio you should have the following:

### 1- An Apple ID.

You must have an Apple ID. So, you must be registered with Apple, at some point give them your email and have a password. This Apple ID, at this step, doesn't have to be a developer account. It could be any sort of Apple account that may be used to download apps from the App Store.

### 2- An IPhone Device

Your Flutter apps are not going to run onto a watch or any other type of Apple device except IPhone or iPad.

### 3- Xcode

Xcode is already installed on your Mac computer as a prerequisite to install Flutter. Also, to avoid any issues of incompatibility between your IPhone software version and XCode, always be sure that you have the latest version of Xcode.

### 4- USB Cable

You need a good quality USB cable that is able to transfer data between your computer and IPhone.

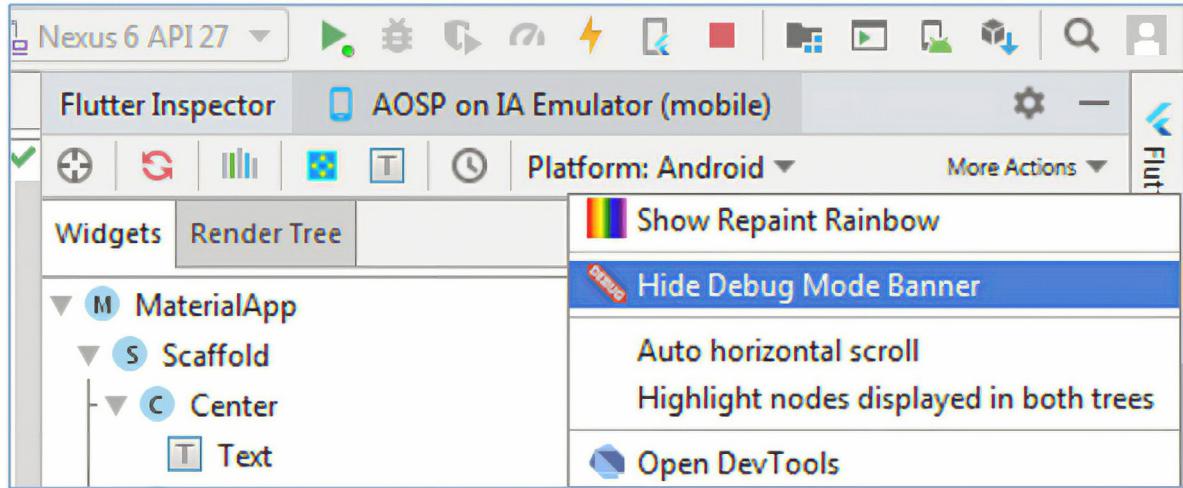
To deploy your Flutter app to a physical iOS device, you need the third-party CocoaPods dependency manager and an Apple Developer account. You'll also need to set up physical device deployment in Xcode. For more information regarding this, please check the subject : “**Deploy to iOS devices**” in the following link :

<https://flutter.dev/docs/get-started/install/macos>

## Emulator Debug Mode

On the top right corner, you can see the “**Debug**” mode title banner. This banner means that this app, running at the moment, is not the final version of the app. It is just for the development and debugging stage. Therefore, when you finally build your app and run it on a real device, then that title banner will be removed.

Now, if you don't want to see this title banner when you run your app, click on **Inspector** console , click on **More Actions** drop down list , and then click on **Hide Debug Mode Banner** as illustrated in the following figure.



The debug mode banner will be removed directly from your phone emulator without the need to run your app again because this part is not related to your app code. However, it is related to your phone emulator settings.

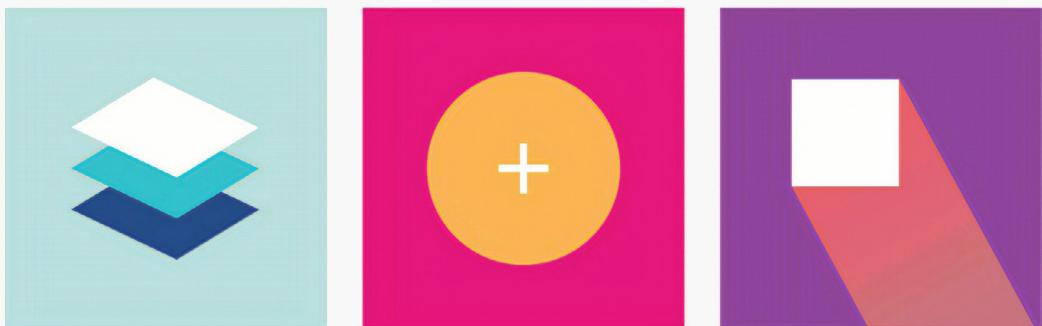
## Introduction to Flutter Widgets

Flutter is different from other frameworks because its UI (user interface) is built in code, not (for example) in an XML file. Widgets are the basic building blocks of a Flutter user interface. Almost everything in Flutter app is a widget such as images, icons, texts, menus, buttons, row, column etc.. A widget is an immutable object that describes a specific part of your app user interface. Also, you can combine existing widgets to make more sophisticated widgets. For example, you may add three rows inside a column and add an image inside each row.

### Example:

If you want to create a document that has text and images and you are looking for a good layout, the most efficient way would be by creating a table and arranging the content of your document inside your table cells. Creating a table, will give you more control about how to distribute your document content. This also applies to designing a web page. The best design to arrange all the web page content of images, animation, forms and texts is in a table as illustrated in the following figure. Using tables results in a stable layout or user interface even to the end user screen size or resolution.

### Principles



Material is the metaphor	Bold, graphic, intentional	Motion provides meaning
A material metaphor is the unifying theory of a rationalized space and a system of motion. The material is grounded in tactile reality, inspired by the study of paper and ink, yet technologically advanced and open to imagination and magic.	The foundational elements of print-based design—typography, grids, space, scale, color, and use of imagery—guide visual treatments. These elements do far more than please the eye. They create hierarchy, meaning, and focus. Deliberate color choices, edge-to-edge imagery, large-scale typography, and intentional white space create a bold and graphic interface that immerses the user in the experience.	Motion respects and reinforces the user as the prime mover. Primary user actions are inflection points that initiate motion, transforming the whole design.
Surfaces and edges of the material provide visual cues that are grounded in reality. The use of familiar tactile attributes helps users quickly understand affordances. Yet the flexibility of the material creates new affordances that supersede those in the physical world, without breaking the rules of physics.	All action takes place in a single environment. Objects are presented to the user without breaking the continuity of experience even as they transform and reorganize.	Motion is meaningful and appropriate, serving to focus attention and maintain continuity. Feedback is subtle yet clear. Transitions are efficient yet coherent.
The fundamentals of light, surface, and movement are key to conveying how objects move, interact, and exist in space and in relation to each other. Realistic lighting shows drama, divides space, and indicates moving parts.	An emphasis on user actions makes core functionality immediately apparent and provides waypoints for the user.	

Similarly, when designing your Flutter app user interface, you should arrange your app content inside rows and columns widgets. This will guarantee that you will get the same UI or layout even though the device type or screen size which will be used in running your app. In Flutter development, usually we use rows and columns as main widgets.

The core of Flutter's layout mechanism is widgets. In Flutter, almost everything is a widget—even layout models are widgets. The images, icons, menus, buttons and text that you see in any Android or iOS app are represented in Flutter development as widgets. But things you don't see are also widgets, such as the rows, columns, and grids that arrange, constrain, and align the visible widgets.

**Row** and **Column** are classes that contain and layout widgets. Widgets inside of a **Row** or **Column** are called **children**, and **Row** and **Column** widgets are referred to as **parents**. Row lays out its widgets horizontally, and Column lays out its widgets vertically.

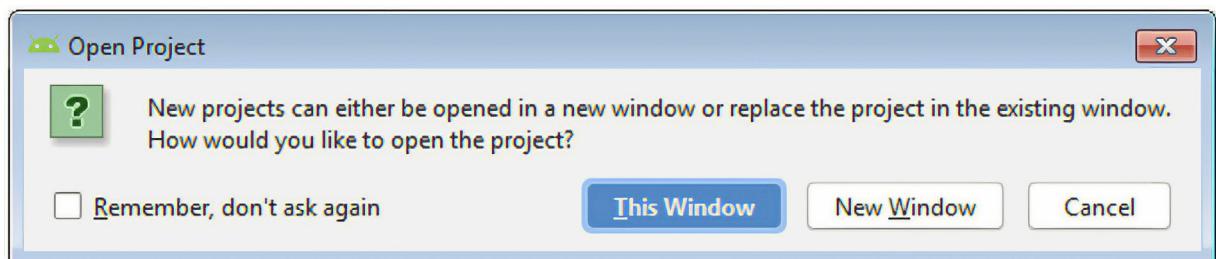
The following figure displays how UI/UX app designer starts with designing the app wireframes or prototypes which are created for the purpose of arranging app elements to best accomplish a particular purpose. The purpose is usually to communicate the business objective as well as a creative idea.

In Flutter development, almost each of these elements (widgets) will be arranged inside columns and rows.

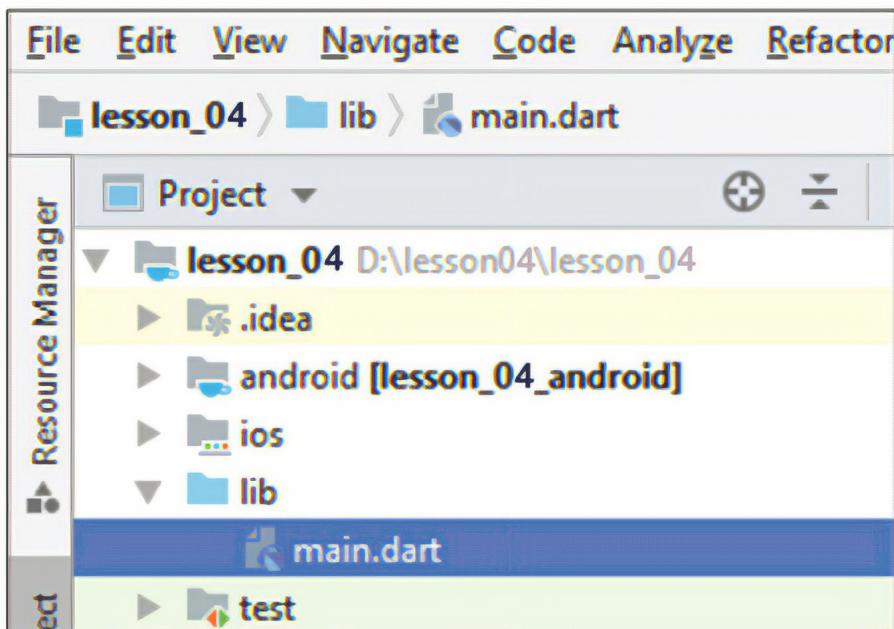


## Creating a Flutter App Using Widgets

- 1- Open **Android Studio**
- 2- Click **File → New → New Flutter Project**
- 3- Select **Flutter Application**. Then click **Next**
- 4- Enter the **Project name** : **lesson\_04** (Capital letters in project name is not accepted), then click **Next**.
- 5- Enter your company domain name or your name, and then click **Finish**. For example, if you used androidatc.com as company domain name, your app name will be **com.androidatc.lesson\_04**. This name (com.androidatc.lesson\_04) must be a unique name when you upload your app later on Google and Apple stores.
- 6- If you get the below dialog box, select **This Window** to work on a new Flutter project.



7- You will get the default Flutter project which is created by the Flutter team, and included by default a simple counter app. The default location of this **main.dart** file is : **lesson\_04 → lib → main.dart** as illustrated in the following figure:



8- Delete all the code lines of **main.dart** except the first lines, then add the **main** function . This means that you must have the following code only in **main.dart** file:

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

void main() {
```

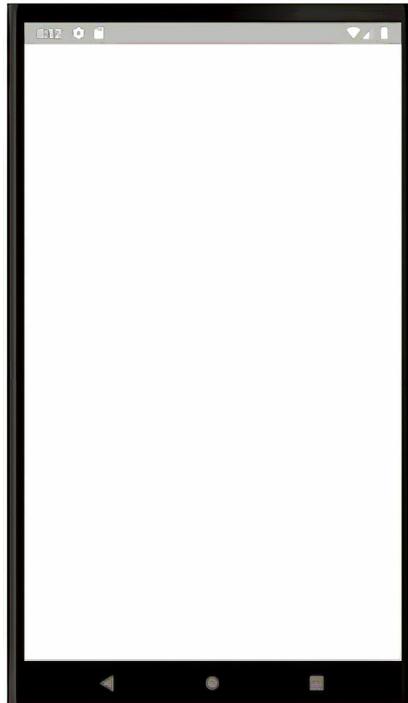
All your app structure and content will be between the two braces { } of the **main()** function.

In Flutter, almost everything is a widget—even layout models are widgets. Let us start with **Text** widget by adding the following code:

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

void main() {
  Text('Hello, world!');
```

If you run your app, you will get an empty app as illustrated in the following figure:



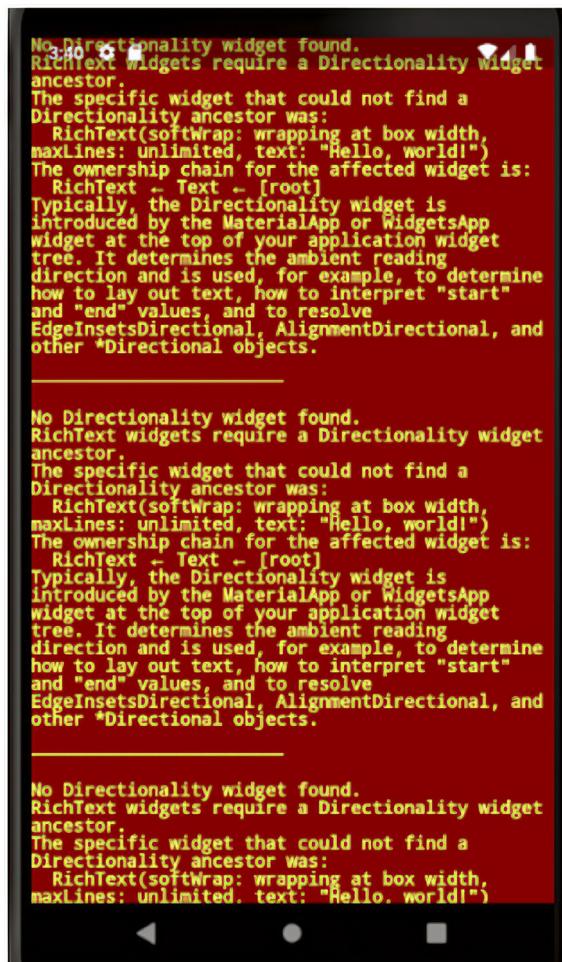
To attach the Flutter widgets to the device screen , you need to use **runApp** function. All widgets will belong to this function as follows:

```
runApp (  
  Widget app  
) ;
```

9- Now, using **runApp** function, the app code will be as follows:

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(Text('Hello, world!'));  
}
```

10- Run your app. The run output will be as illustrated in the following figure. This run output displays: No Directionality widget found. This means that this text widget must have a specific format that determines its location on the device screen especially the alignment format (center, left, or right), and if this text has “right to left” or “left to right” format.



11- Now, we will add **Center** widget as a parent widget for the **Text** widget. This means that the Text widget will be a child widget. Also, we must add more properties for this Text widget as follows:

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

void main() {
  runApp(
    Center(child:Text('Hello,world!',textDirection:
    TextDirection.ltr)));
}
```

Note : **ltr** means : left to right , while you can use **rtl** for right to left.

12- Run your app. The run output follows:



13- Now, all your code is on the same line, and it is really hard to distinguish which widget descends from the other. So, how can you make this easier?

Well, by installing Flutter and Dart packages, we automatically get access to a reformat called **Dartfmt** (Dart Format). This feature is really helpful; however, you will have to help it, so it will help you better. The Flutter team advises that whenever you create a widget which has parentheses or round brackets, add a comma after every round bracket. If you do this, when you **Save** or when you right click the code and select **Reformat Code with Dartfmt** option, the code will automatically be formatted showing an indented and more organized structure.

After adding a comma after each round bracket, right click the code, and select **Reformat Code with Dartfmt** option. The code will be as follows:

```
main.dart
```

```
1 import 'package:flutter/material.dart';
2
3 void main() {
4   runApp(
5     Center(
6       child: Text('Hello, world!',
7           textDirection: TextDirection.ltr), // Text
8     ), // Center
9   );
10 }
```

The image shows a code editor window with the file name "main.dart" at the top. The code itself is a simple Flutter application. It starts with an import statement for the Material Dart package. The main function calls runApp, which is passed a Center widget. The Center widget contains a Text widget with the string "Hello, world!". The Text widget also includes a comment specifying the text direction as "TextDirection.ltr". The entire code is enclosed in a pair of curly braces. The code editor uses color-coded syntax highlighting to distinguish between different parts of the code, such as keywords like "import", "void", and "runApp", and variable names like "main" and "textDirection".

As you see here, this app run output has a poor layout because it does not have any format such as app title bar, buttons or others. To improve your app interactivity, you don't need to create each button, app bar , icon or other items manually by writing a specific code for each. Instead, Flutter has a lot of widgets to work as templates which are ready to add and customize to your app.

Think of a Widget as a visual component (or a component that interacts with the visual aspect of an application). When you need to build anything that directly or indirectly is in relation with the layout, you are using Widgets. Flutter provides a number of widgets that help you build apps that follow Material Design.

Widgets are organized in tree structure(s). A widget that contains other widgets is called **parent Widget** (or *Widget container*). Widgets which are contained in a *parent Widget* are called **children Widgets**.

## What is a MaterialApp widget ?

At the first step in creating a Flutter app, you will add a **MaterialApp** class. MaterialApp is the starting point of your app, it tells Flutter that you are going to use material components widgets and will follow a material design in your app.

MaterialApp class wraps a number of widgets that are commonly required for material design applications such as Scaffold ,AppBar, Bottom navigation bars, Column, Row, Logo, Icon, Image, Raised Button, Text, and other widgets. Almost all apps start with a MaterialApp widget.

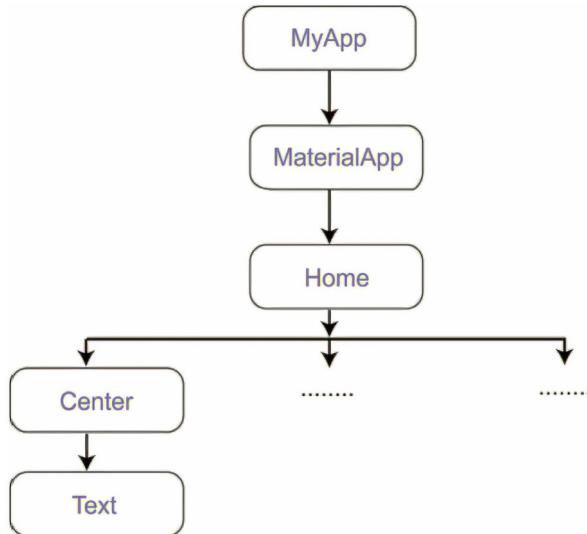
Almost all apps start with MaterialApp widget, which builds a number of useful widgets at the root of any app.

14- Now, using the previous code, you will add a **MaterialApp** widget to our app code, then all other app widgets will be within or related to this class. Also, you will add “**home**” widget which is the route that is displayed first, when the application is started normally. The code will be as follows:

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

void main() {
  runApp(MaterialApp(
    home: Center(
      child: Text('Hello, world!', textDirection: TextDirection.ltr),
    ),
  ));
}
```

This app widget tree is represented by the following diagram. For example, in this widget tree, the Center widget is the partner widget for Text widget.



15- The run output of this app follows:



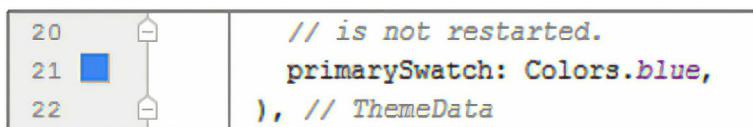
Can you imagine if you design an app that has this user interface, what will be your customer feedback?

# Lab 4

## Creating a Simple Flutter App

In this lab, you will create a simple Flutter project, and browse some Android Studio tools by performing the following steps:

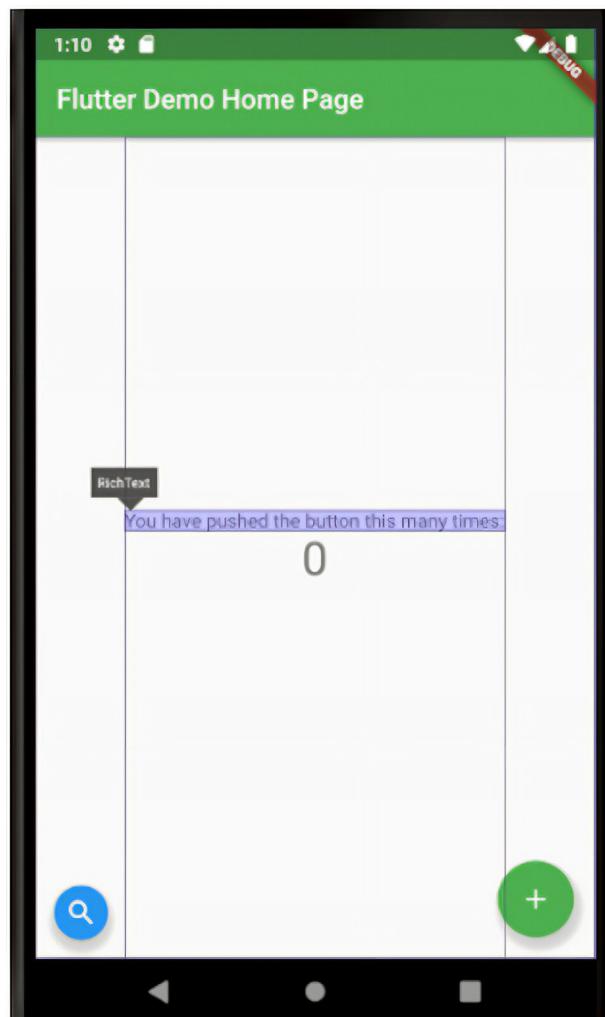
- 1- Open **Android Studio**
- 2- Click **File → New → New Flutter Project**
- 3- Select **Flutter Application**, and click **Next**
- 4- Type **flutter\_lab04** for **Project Name**: , then click **Next**
- 5- Type **androidatc.com** for the **Company name**: , keep the other fields with the default values, and click **Finish**
- 6- Open the **main.dart** file which is inside the **lib** folder.
- 7- Click at the beginning of any line of the Dart code, and press Tab key twice , then right click and select **Reformat Code with dartfmt** option of the shortcut menu. The line of code which moved from its location must be return to its original location.
- 8- Click **Run** button of Android Studio tool bar, and wait until your phone emulator completes its operating system startup process, then startup the Flutter app.
- 9- Scroll down until you find the blue square on the left margin of the code as illustrated in the following figure:



Replace **Colors.blue** with **Colors.green** , then click the **Flutter Hot Reload** button , and check this change take effect on your phone emulator. You will find that the blue color parts of your app change to green color. Also, the blue square is now green.

- 10- Click on **Flutter Inspector** tab, which is on the right side of Android Studio, then click on **Select Widget Mode** button.

- 11- On your phone emulator, click the text : **You have pushed the button this many times:**  
You should get the following figure:



- 12- Check your code. You will find the following part of code has been selected:

```
94     ),
95   ),
96   ),
97   )
```

The screenshot shows a code editor with several lines of Dart code. Lines 94 through 97 are highlighted in yellow. The code consists of three "Text" widgets, each followed by a comma and a space. The first two commas are preceded by a double slash, indicating they are comments. The third comma is preceded by a single slash, indicating it is a regular comment.

- 13- Replace this text : "**You have pushed the button this many times:**" with your name , then click the **Flutter Hot Reload** button.

- 14- Click the **Widget Mode** button on the **Flutter Inspector** window again to disable its effect.

- 15 - Now, Check your phone emulator.

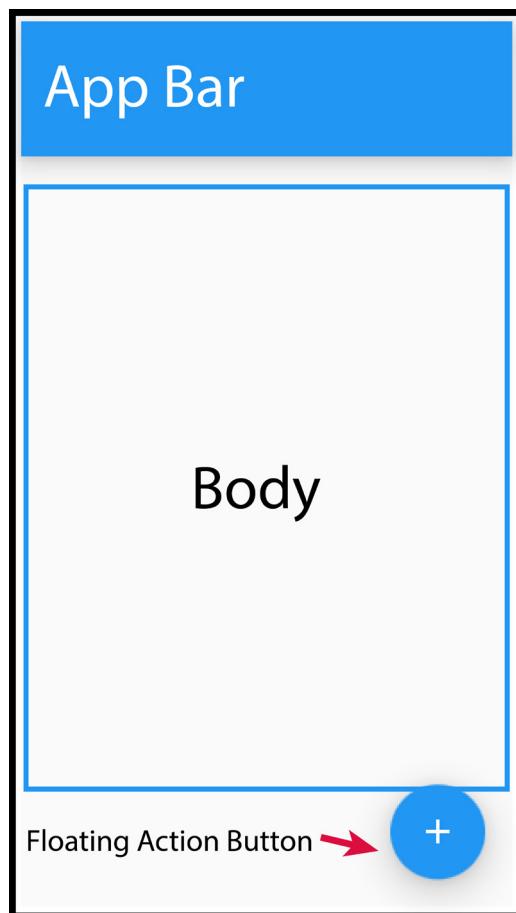
# Lesson 5: Flutter Widgets Fundamentals

<b>Scaffold Widget</b> .....	5-2
<b>Image Widget</b> .....	5-8
<b>Container Widget</b> .....	5-15
<b>Column and Row Widgets</b> .....	5-24
<b>Icon Widget</b> .....	5-29
<b>Layouts in Flutter</b> .....	5-31
<b>Card Widget</b> .....	5-42
<b>App Icons for iOS and Android Apps</b> .....	5-46
<b>Hot Reload and Hot Restart</b> .....	5-50
<b>Stateful and Stateless Widgets</b> .....	5-55
<b>Use a Custom Font</b> .....	5-64
<b>Lab: Creating a Restaurant Menu</b> .....	5-69

## Scaffold Widget

When you want to create a Flutter app, you need to configure a lot of widgets and change their format. You don't need to create everything from scratch. You can easily add the **Scaffold** class or widget to your app. This class implements the basic material design visual layout structure for our app. This class provides application programming interface for displaying drawers, snack bars, bottom sheets , and other Flutter widgets.

This `Scaffold()` widget works as a container for other Flutter widgets, such as app body, which will include the app content and `AppBar` as illustrated in the following figure:



### Example:

In the following example, you will create a small Flutter app using the `Scaffold` widget as a main parent for all your app interface widgets. To create this app, perform the following steps:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** , and then click **Next**.

4- Type : **scaffold\_widget** for Project Name , and create a new folder : **Lesson\_05** for Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain, and then click **Finish**

6- Open **main.dart** file, delete all the code , and type the following code :

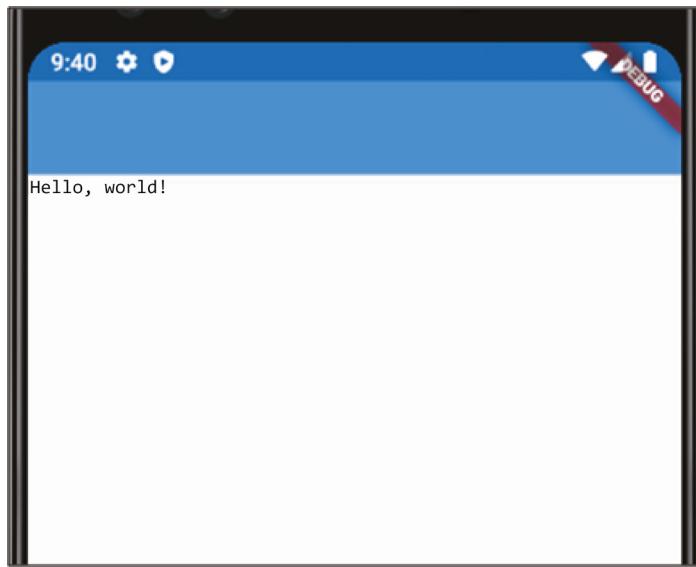
```
import 'package:flutter/material.dart';

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

class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        appBar: AppBar(),
        body: Text('Hello, world!', textDirection: TextDirection.ltr),
      ),
    ),
  );
}
```

7- Now, **Run** your app. The run output will be as follows:



8- As you see in the previous figure, the text “Hello, world!” appears at the top left corner of the app interface, because this is the text default alignment format of the **Text** widget. If you want to move this **Text** widget content to the center of your app interface, configure this **Text** widget as a child widget of the **Center** widget, as illustrated in the following code:

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

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        appBar: AppBar(),
        body: Center(
          child: Text('Hello, world!', textDirection:
TextDirection.ltr),
        ),
      ),
    );
}
```

The run output follows:



As you see here, each widget performs a variety of roles. The Text widget is responsible for displaying and styling text . The Center widget is responsible for aligning objects in the center. Also, the **MaterialApp** is similar to grandpa of all the widgets because it usually sits at the top of the code, and everything that you build is a material app. Later in this lesson, you will learn more about how to control the position of your app items.

9- Now, you can add more properties for your app widgets. Here in this app, you will add some properties for the AppBar widget , for this widget, you can add the title property, that is a Text widget. In addition, you may add some format to this Title such as adding the center Title, and backgroundColor properties as illustrated in the following figure:

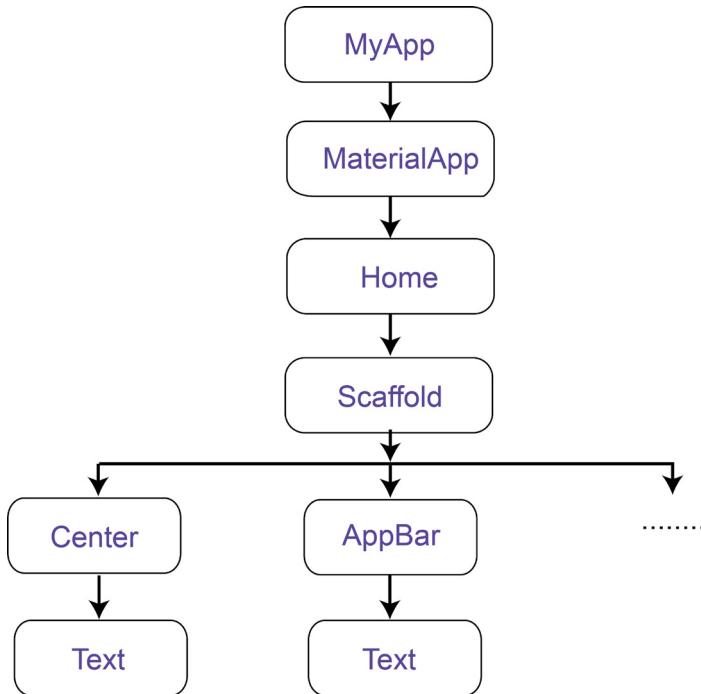
```
import 'package:flutter/material.dart';

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('My First App'),
          centerTitle: true,
          backgroundColor: Colors.green),
        body: Center(
          child: Text('Hello, world!', textDirection: TextDirection.ltr),
        ),
      ),
    );
}
```

10- Run your app, and you will get the following:



Now your app widget tree as follows :



As you see in the previous figure, the AppBar widget has some properties such as backgroundColor which changes your app title bar color. To get more information about colors list check the following web site:

<https://material.io/design/color> and check the color list at the end of this web page. You will find the gradients of each color is from 50 until 900 , this means you can control more in the color gradients by adding a value for each color between 100 and 900 between two square brackets (you can start with 50 , 100, 200 ,...900). For example, in the previous code replace

```
backgroundColor: Colors.green
```

with

```
backgroundColor: Colors.green[900]
```

You will get a dark green color , while if you use 100 , you will get a light green color.

11- Assume that you want to change the background color of your Scaffold widget (appBar and body). You can do that easily by adding the following property to your Scaffold widget as illustrated in the highlighted part of following code:

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

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        backgroundColor: Colors.amberAccent,
        appBar: AppBar(
          title: Text('My First App'),
          centerTitle: true,
          backgroundColor: Colors.green[500]),
        body: Center(
          child: Text('Hello,world!',textDirection:TextDirection.ltr),
        ),
      ),
    ),
  );
}
```

## Image Widget

This class or widget is used to display an image in Flutter app. In Flutter , the following image formats are supported: JPEG, PNG, GIF, Animated GIF, WebP, Animated WebP, BMP, and WBMP.

You can add image to your Flutter app using different sources. In this lesson you will add an image using **asset** and **network** sources.

### Adding an image form Network:

12- Continue using the same code previously. To add an image to your app, add **Image** widget to your code with the source or the path where this image exists. For example, if you want to add a network image, you should have the URL of this image. To do that, go to the web site which has this image, and then right click this image. Then, select **Open image in new tab**, then copy this image URL.

For example, go to Android ATC web site ([www.androidatc.com](http://www.androidatc.com)), right click the

Android ATC logo, and then select: “**Open image in new tab**”. Go to this new tab and copy the URL.

In this example the logo URL is:

<https://androidatc.com/template/style/img/Android-ATC-Logo.jpg>

13- Now, in your code delete: child:Text('Hello,world!',textDirection:TextDirection.ltr), from your app body code, and add the Image widget with the image source (network URL) to your app as illustrated in the following code:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      backgroundColor: Colors.amberAccent,
      appBar: AppBar(
        title: Text('My First App'),
        centerTitle: true,
        backgroundColor: Colors.green[500]),
      body: Center(
        child: Image(
          image: NetworkImage('https://androidatc.com/template/style/img/Android-ATC-Logo.jpg'))),
    ),
  )));
}
```

Before run this app, try to open Google web site using your phone emulator to be sure that your phone emulator has Internet connection and your phone emulator web browser is working fine.

When you run your app, you should get the following run output:



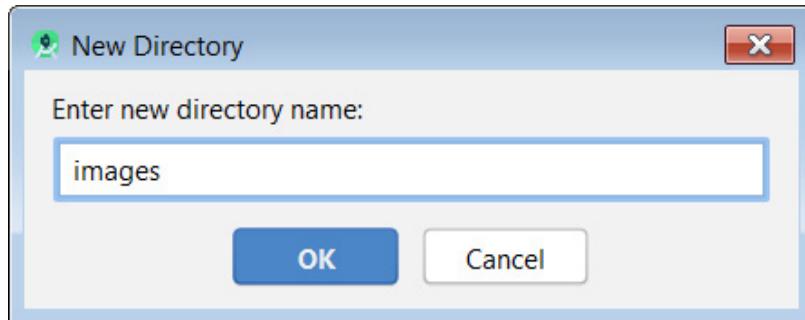
### **Adding an image to and from your App Asset:**

Through this method, the images files will be part of your app files. You will import your app images to a specific folder that is already a part of your app file structure, and then add them to your app code files using AssetImage class.

This method is superior, because your app users will use local images, and they don't need to connect to the internet to see each image in your app interfaces, especially if your app is designed to work offline.

Continue to use the previous code to add an asset image. Please perform the following steps:

14- Right click your Flutter project name (**lesson\_05**) → New → Directory. Type images for the directory name, and then click **OK** as illustrated in the following figure:

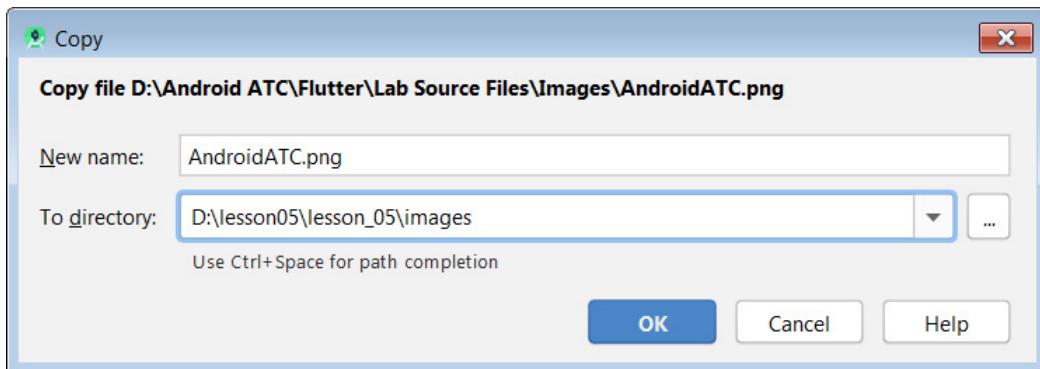


Remember that this folder is created to include all your app images.

15- Now, to add one or more images to this folder, just copy your images from your hard disk and then paste them in this folder. Very easy right !!

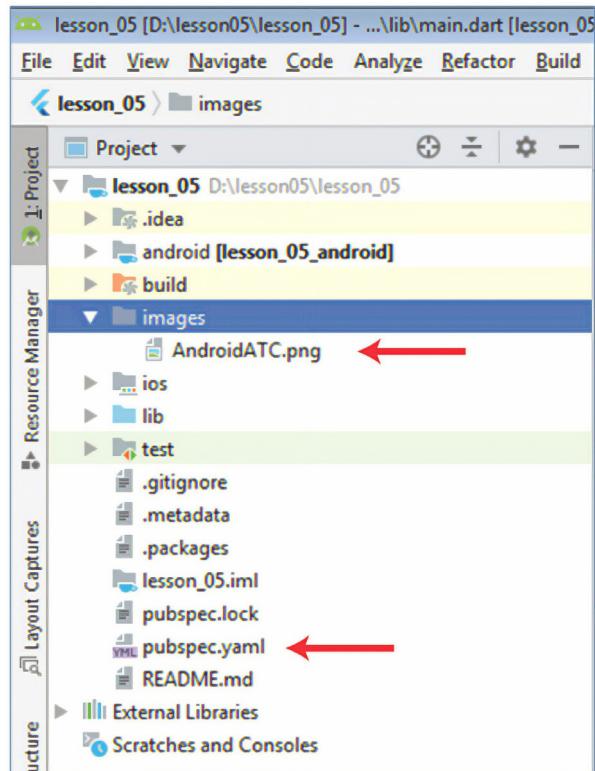
In this example, you will copy the “**AndroidATC.png**” image from our lab files older: **Lab Source Files\Images** (check your hard disk , and if you don’t have this folder on your computer, check with your trainer or contact Android ATC support team at: support@androidatc.com) then **paste** this image in your app **images** folder which you created in the previous step.

When you do this, you will get the following dialog box, just click **OK** to confirm.



**Note:** You can select many images , select copy , and then paste them in your images folder in one step.

Now you have the following project files structure:



16- Now, you should configure your Flutter app to the default location (folder) of the images in this app. To do this you must configure the **pubspec.yaml** file which is an important part of your app files. This file location is illustrated in the previous figure. This file is the most important file in any Flutter project because it includes all your Flutter app configurations. It is the place where you provide all the required dependencies of your Flutter project. This file is written using YAML language (Yaml ain't markup language).

YAML is a human-readable data serialization standard that can be used in conjunction with all programming languages and is often used to write configuration files. For example, in this YAML language when you want to configure a line of code as a comment line , you must add # at the beginning of this line of code, while you should add // in Dart programming language.

You have to be really careful when you are making changes in this **pubspec.yaml** file code, because

in JAML language, almost each configuration consists of a parent and child level are configured by an indent, and is each indent level is represented by two spaces. Therefore, be careful about indentation in this YAML file.

To configure the assets, double click **pubspec.yaml** file, and then scroll down until

you find the part which is related to the assets configurations as illustrated in the following figure:



```

main.dart × pubspec.yaml ×
Flutter commands Packages get Packages upgrade Flutter upgrade

40 # the material Icons class.
41 uses-material-design: true
42
43 # To add assets to your application, add an assets section, like this:
44 # assets:
45 #   - images/a_dot_burr.jpeg
46 #   - images/a_dot_ham.jpeg
47

```

17- Delete ONLY the hash sign for **assets:** and **-images/a\_dot\_burr.jpeg** lines. Also, **delete** the last line : **-images/a\_dot\_ham.jpeg**.

18- In **pubspec.yaml** file, replace the image file name “**a\_dot\_burr.jpeg**” with “**AndroidATC.png**” or your image file which you pasted in the images folder, and move or indent the “**-images/AndroidATC.png**” two spaces of **assets:** as illustrated in the following figure:



```

43 # To add assets to your application, add an assets section, like
44 assets:
45   - images/AndroidATC.png
46

```

2 spaces

19- Click **Packages get** at the top of **pubspec.yaml** file content as illustrated in the following figure:

Packages get Packages upgrade | Flutter upgrade | Flutter doctor

20- In the **main.dart** file, delete the part which is related to add the image from network which is illustrated in the following code:

image: NetworkImage(  
 'https://androidatc.com/template/style/img/Android-ATC-Logo.jpg')

21- To add the image which you added to images folder, use the **AssetImage** class as it is illustrated in the grey highlighted part of the following code:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      backgroundColor: Colors.amberAccent,
      appBar: AppBar(
        title: Text('My First App'),
        centerTitle: true,
        backgroundColor: Colors.green[500]),
      body: Center(
        child: Image(image: AssetImage('images/AndroidATC.png')),
      ),
    ),
  )));
}
```

22- Stop your app , and then Run it again. Your image will be added to your app as illustrated in the following figure:



23- Assume that you want to add 10 images or more to your app images folder. Do you think there is another way to add these images to your app without the need to add them line by line in your app pubspec.yaml file?

The answer is Yes !!. you only have to add the images folder name with a forward slash "/" in the pubspec.yaml file as illustrated in the following figure, then you can add your images to your app files using AssetImage widget.

```
42
43      # To add assets to your application, add an assets section, like this:
44      assets:
45          - images/
46          # - images/a_dot_ham.jpeg
47
```

Replace **-images/AndroidATC.png** with **-images/** in the **pubspec.yaml** file , click **Packages get** to apply the new configuration, and then Run your app again. You will get the same previous run output.

## Container Widget

When we want to have full control in a location when it comes to width and height of any Flutter widget, we use the Container widget as a parent widget and configure the other widget which we want to control as a child widget. You may add a lot of properties to your Container widget such as background color, size, padding, margins, borders or the shape of text, and other properties.

The Container is a widget class that allows you to customize, compose, decorate and position its child widget. Container widget is similar to DIV tag in HTML.

Now, you will continue using your previous code in lesson 5, and check the benefit of using the Container widget in your app.

24- Delete the part which is related to the image. Then, you will have the following code:

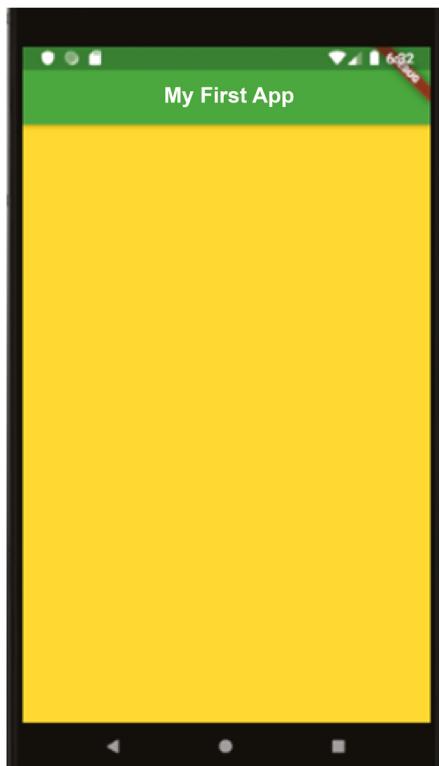
```
import 'package:flutter/material.dart';

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      backgroundColor: Colors.amberAccent,
      appBar: AppBar(
        title: Text('My First App'),

```

```
        centerTitle: true,  
        backgroundColor: Colors.green[500]),  
  
    ),  
));  
}
```

And when you run your app, you will get the following:

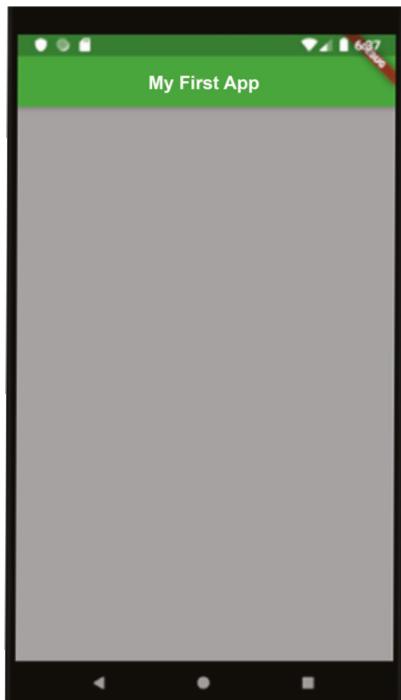


25- Add to your app a Container widget with a grey color property. The code will be as follows:

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(MaterialApp(  
    home: Scaffold(  
      backgroundColor: Colors.amberAccent,  
      appBar: AppBar(  
        title: Text('My First App'),  
        centerTitle: true,
```

```
        backgroundColor: Colors.green[500]),  
  
    body: Container(  
        color: Colors.grey,  
    )),  
,  
});  
}
```

Run your app. As you see in the following figure, the body of the app interface has a grey background color :

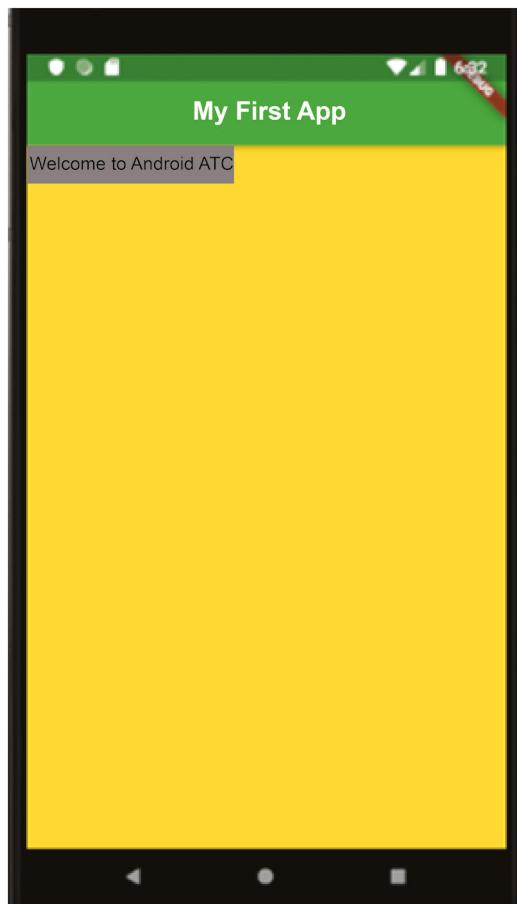


In the previous code, the Container widget did not include any child widget yet, and its effect covered all the app interface body. While when this Container widget has a child widget such as Text widget, it will try to be as small as possible. The following code displays how to add a Text widget as a child widget of a Container widget:

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(MaterialApp(  
    home: Scaffold(  
        backgroundColor: Colors.amberAccent,  
        appBar: AppBar(  
    ),  
  ),  
,  
);  
}
```

```
        title: Text('My First App'),  
        centerTitle: true,  
        backgroundColor: Colors.green[500]),  
  
        body: Container(  
            color: Colors.grey,  
            child: Text('Welcome to Android ATC'),  
        )),  
    ),  
);  
}
```

26- Run your app. As illustrated in the following figure, the Container widget has shrunk to the size of the Text font size.



As you see here, the Container widget size shrinks to match the size of the child widget.

So far, we can conclude the following about the Container widget:

- If you wrap a widget in a Container widget without any other parameters, you will not notice any difference in appearance.

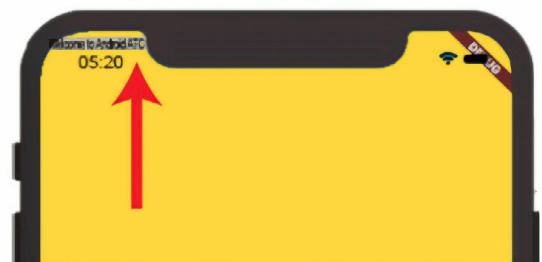
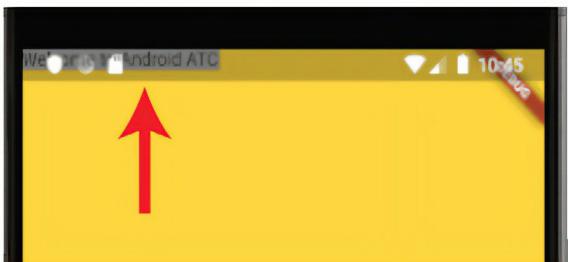
- But if you add the color parameter, your child widget will get a background color. Without anything else, the Container sizes itself to its child.

27- Add a comment sign to the AppBar widget and to all its related widgets as you see in the following code:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      backgroundColor: Colors.amberAccent,
      //appBar: AppBar(
      //  title: Text('My First App'),
      //  centerTitle: true,
      //  backgroundColor: Colors.green[500]),
      body: Container(
        color: Colors.grey,
        child: Text('Welcome to Android ATC'),
      ),
    )));
}
```

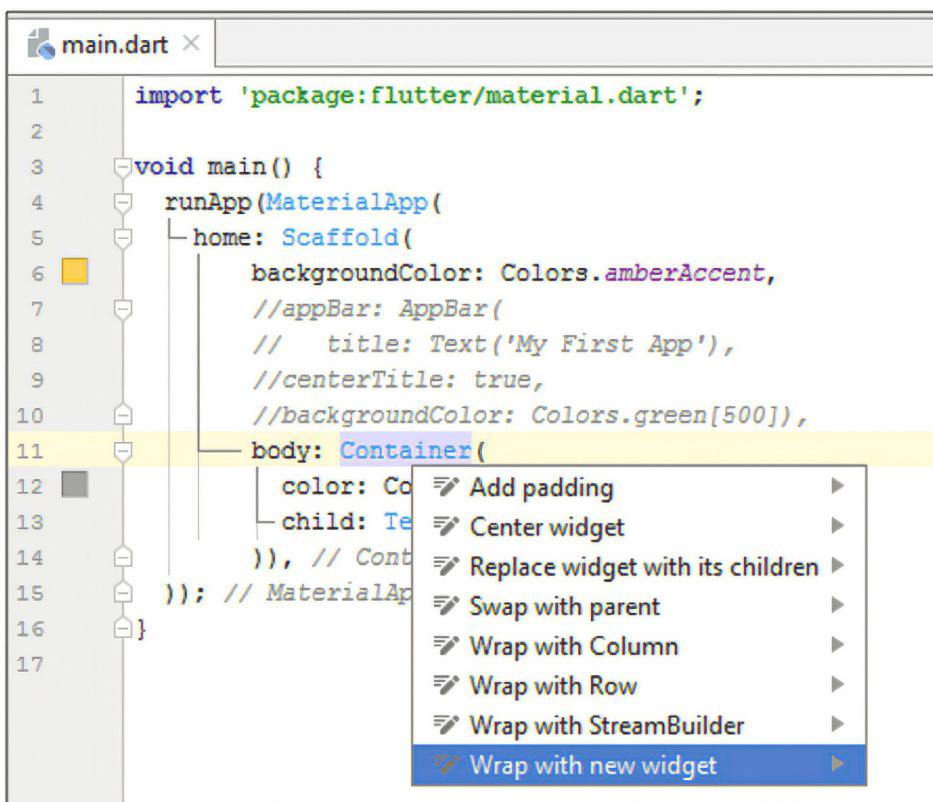
Now run your app again, and you will see in the following figures (Android and iOS emulators), the container widget location at the top left and outside the app body area.



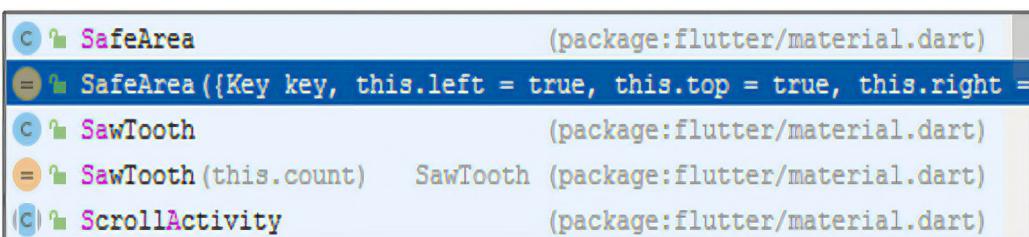
28- To be sure that the Container widget and its child of another widgets will be in a safe area, and will appear inside the app body area, you will configure your Container widget as a child widget of another widget. This is called SafeArea widget.

Instead of configuring the Container widget as a child widget of the SafeArea widget by typing the code manually, you may click on the Container widget that

you want to embed in another widget and either click on the little **yellow light bulb** (lamp) or press (**Alt +Enter**) on Windows or (**Option + Return**) on Mac and you will get a shortcut menu as illustrated in the following figure. Select **Wrap with new widget**, and then type **SafeArea** for the new widget.



**Note :** When you start typing the first letters of **SafeArea** widget name, select the second choice as it is illustrated in the following figure:



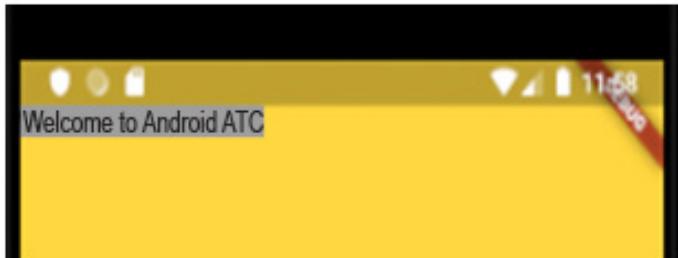
Now, you should have the following code:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
```

```
backgroundColor: Colors.amberAccent,  
//appBar: AppBar(  
//  title: Text('My First App'),  
//centerTitle: true,  
//backgroundColor: Colors.green[500]),  
body: SafeArea(  
  child: Container(  
    color: Colors.grey,  
    child: Text('Welcome to Android ATC'),  
  ),  
),  
);  
}
```

As you see in the following figure, SafeArea widget added the necessary padding to keep your widget from being blocked by the system status bar, notches, holes, rounded corners and other “creative” features by manufacturers.

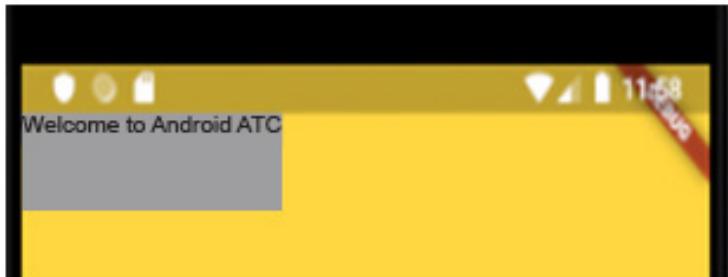


29- Change the size of your Container widget by adding the **height** and **width** properties to your Container widget as it is illustrated in the following code:

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(MaterialApp(  
    home: Scaffold(  
      backgroundColor: Colors.amberAccent,  
      //appBar: AppBar(  
      //  title: Text('My First App'),  
      //centerTitle: true,  
      //backgroundColor: Colors.green[500]),  
      body: SafeArea(  
        child: Container(  
          width: 200,  
          height: 70,  
          color: Colors.grey,
```

```
        child: Text('Welcome to Android ATC'),  
    ),  
),  
));  
}  
}
```

The run output follows:



30- Use the margin property to add empty space that surrounds that Container widget. The following is a part of code that is responsible for adding 30 pixels margin for all directions (top, right, left, and bottom) in the Container widget.

```
child: Container(  
    margin: EdgeInsets.all(30),  
    width: 200,  
    height: 70,  
    color: Colors.grey,  
    child: Text('Welcome to Android ATC'),  
,  
,
```

31- Use the Container padding property to add empty space between the child widget (Text) and the Container boundary as it is illustrated in the following code:

```
import 'package:flutter/material.dart';  
  
void main() {  
    runApp(MaterialApp(  
        home: Scaffold(  
            backgroundColor: Colors.amberAccent,  
            //appBar: AppBar(  
            //    title: Text('My First App'),  
        ),  
    ),  
,
```

```
//centerTitle: true,  
//backgroundColor: Colors.green[500]),  
body: SafeArea(  
    child: Container(  
        padding: EdgeInsets.all(10.0),  
        margin: EdgeInsets.only(left: 30),  
        width: 200,  
        height: 70,  
        color: Colors.grey,  
        child: Text('Welcome to Android ATC'),  
    ),  
),  
));  
}  
}
```

32- With the decoration property, you may add a shape, like a circle, to your Container. The decoration is by default, sized to the container's child.

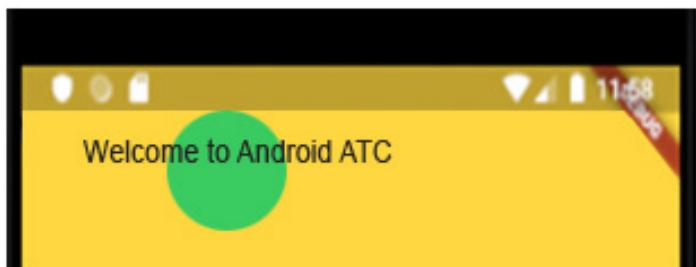
In this case, the Container fits the circle decoration to the narrowest parameter, the Text widget's height, as illustrated in the following code:

Note here , you must change the Container color property to comment.

```
import 'package:flutter/cupertino.dart';  
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(MaterialApp(  
    home: Scaffold(  
        backgroundColor: Colors.amberAccent,  
        //appBar: AppBar(  
        //  title: Text('My First App'),  
        //centerTitle: true,  
        //backgroundColor: Colors.green[500]),  
        body: SafeArea(  
            child: Container(  
                decoration: BoxDecoration(  
                    color: Colors.green,  
                    shape: BoxShape.circle,  
                ),  
                padding: EdgeInsets.all(10.0),  
                margin: EdgeInsets.only(left: 30),  
                width: 200,
```

```
        height: 70,  
        // color: Colors.grey,  
        child: Text('Welcome to Android ATC'),  
      ),  
    )),  
  );  
}
```

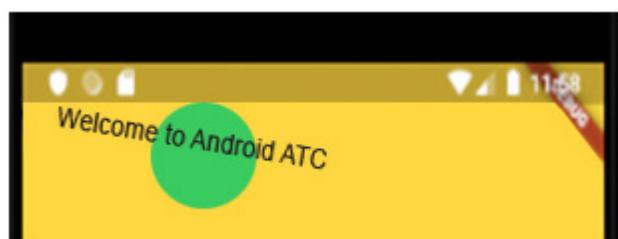
The run output is as follows:



33- The transform property applies a slight rotation to your Container widget if you add the following code to your Container widget:

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

Then the run output will be as follows:



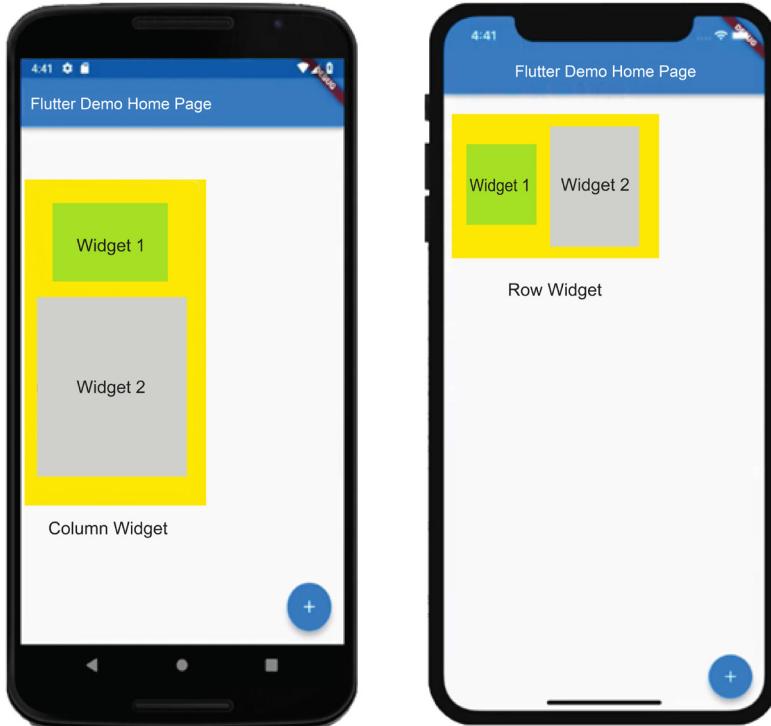
**Note:** The Container widget can have only one child.

## Column and Row Widgets

In the previous section you learnt and made a practice about how to use container widget in a location data on your app interface and transform, background color, size , padding, margins, borders or the shape of text , image or any other widget.

However, the container widget can have only one child widget or object. Assume that you want something that can take lots of children of widgets not just a single widget. The easiest way to do this is using the rows and columns widgets.

Rows and Columns are classes that contain and lay out widgets. Widgets inside of a Row or Column are called *children* widgets, and Rows and Columns are referred to as *parent* widgets. Rows lay out their widgets horizontally, and Columns lay out their widgets vertically.



Continue using the previous code of this lesson. Within the next steps, you will learn more about how to arrange your app content using Column and Row widgets.

34- Delete all the codes which are related to the Container and AppBar widgets. Then you will have the following code:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      body: SafeArea(
        )),
  )));
}
```

35- Add a Column widget as a child widget of the SafeArea widget as illustrated in the following figure:

```
1 import 'package:flutter/material.dart';
2
3 void main() {
4   runApp(MaterialApp(
5     home: Scaffold(
6       body: SafeArea(
7         child: Column(
8
9           ), // Column
10        ), // SafeArea
11      ), // Scaffold
12    )), // MaterialApp
13 }
```

36- Instead of using a child command to add the content of Column widget, you will use children because Column widget can include more than one widget.

To add children widgets to your Column widget, all your Column children widgets must be between two square brackets as illustrated, in the following code:

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

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        body: SafeArea(
          child: Column(
            children: <Widget>[],
          )),
        )),
      );
}
```

37- Add two Text, and two Container widgets to the Column widget as illustrated in the following code:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      body: SafeArea(
        child: Column(
          children: <Widget>[
            Text('Data 1'),
            Text ('Data 2'),
            Container(
              child: Text('Data 3'),
              color: Colors.grey,
              margin: EdgeInsets.only(left: 20.0),
              height: 50.0,
              width: 70.0,
            ),
            Container(
              child: Text('Data 4'),
              color: Colors.amber,
              margin: EdgeInsets.only(left: 20.0),
              height: 50.0,
              width: 70.0,
            ),
            ],
          ),
        ),
      )));
}
```

The run output follows:



38- Currently your Column is laying out its children (Text and Container) from top to bottom.

If you want to go the other direction instead, you can change its vertical direction using the following property :

`verticalDirection: VerticalDirection.up`

The part of the code which is related to the column widget follows:

```
body: SafeArea(  
    child: Column(  
        verticalDirection: VerticalDirection.up,  
        children: <Widget>[  
            Text('Data 1'),  
            Text('Data 2'),
```

The run output follows:



If you replace:

`verticalDirection: VerticalDirection.up,`

with:

`verticalDirection: VerticalDirection.down,`

You will get the following run output :



This means the default vertical direction is down.

39- Delete the previous code :

verticalDirection: VerticalDirection.down,

40- If you want to change the spacing between your Column content (children widgets), you may use the main axis alignment property. Here, because you are working with Column widget, the main axis is the vertical axis as you will see in the next topic (Layouts in Flutter).

## Icon Widget

Icons are essential to many app user interfaces. They visually express objects, actions and ideas. When done correctly, they communicate the core idea and intent of a product or action, and they bring a lot of benefits to the app user interface, such as saving screen real estate and enhancing aesthetic appeal. An app icon is a design pattern that is familiar to users.

When you add an Icon widget to your Flutter app, you are actually calling a specific code which represents this icon name, and it will draw all the sides and the edges of the icon onto the app screen. Therefore, when you change the size of any of your app icons, they will have the same high resolution quality.

You should know the icon name which you want to add to your Flutter app. This name will be used to call the Icon class in your code. The following web site includes a lot of icons which Google has made available for free: <https://material.io/resources/icons>.

The following table includes some icons and their names:

Icon	Name	Icon	Name
	shopping_cart		fingerprint
	alarm		room
	backup		build

In the following code , you will add an icon, and change its color and size.

Continue using the previous code.

41- Delete the Row widget and all its content of other widgets. Then, you will have the following code:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      body: SafeArea(
        ),
      ),
    )));
}
```

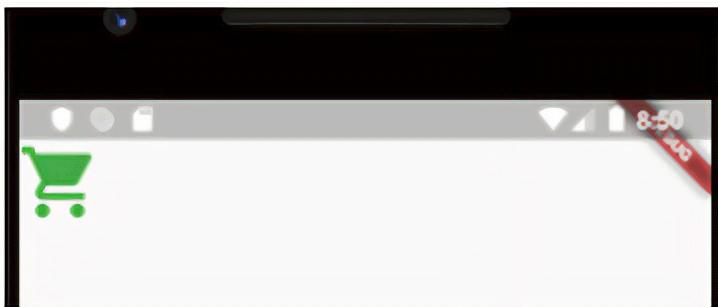
42- In the SafeArea widget , add a new Container widget. In the Container widget, add an Icon widget with size and color properties as follows:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      body: SafeArea(
        ),
      ),
    )));
}
```

```
        child: Container(
            child: Icon(
                Icons.shopping_cart,
                size: 50,
                color: Colors.green,
            ),
        ),
    ),
),
));
}
```

The run output follows:



Try other icon names from the previous icon names table.

## Layouts in Flutter

The core of Flutter layout mechanism is widgets. In Flutter, almost everything is a widget—even layout models are widgets. The images, icons, and text that you see in a Flutter app are all widgets. But things you don't see are also widgets, such as the rows, columns, and grids that arrange, constrain, and align the visible widgets.

You create a layout by composing widgets to build more complex widgets. You need to have full control of the position for each item or widget in your app interfaces.

In this section of this lesson, we will focus on some important widgets properties which help you have more control on your app layout. Some of these properties you may have used before in this or previous lessons ; however, in this section we will focus on them in details.

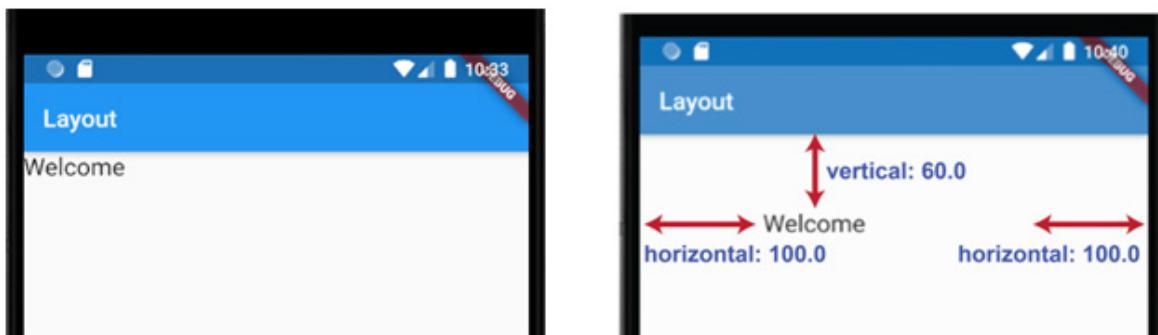
The first important thing you should know, is that you cannot apply directly apply the layout format on any child widget such as `Image` , `Text`, `Checkbox` , ...etc .

Instead, you should apply the layout format on the parent widget which warps these children widgets, such as Container, Column, Row, or another parent widget.

In the following example, the Text widget is wrapped in a Container widget to control its position. Here, using the margins property with the Container widget helps to control the layout of the Text widget as illustrated in the following code:

```
Container(  
    margin: EdgeInsets.symmetric(vertical: 60.0, horizontal: 100.0),  
    child: Text(  
        "Welcome",  
        style: TextStyle(fontSize: 20.0),  
    ),  
,
```

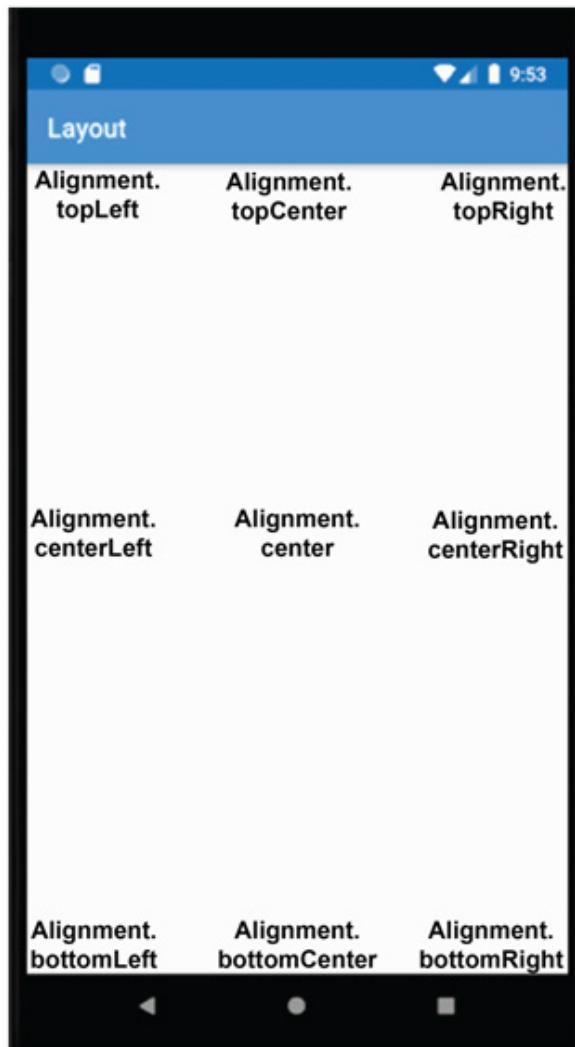
The following two figures display how the text looks like before and after using the margin property:



Also, as illustrated in the following code , using the alignment property with Container widget helps in making changes in the Text widget layout.

```
body: Container(  
    alignment: Alignment.topCenter,  
    child: Text("Flutter App"),  
,
```

The following figure displays the text location when you apply each alignment property value on the Container widget which is wrapped in the Text widget:



If you want to add padding around Row or Column content, it is a good idea to place the Row or Column widget in a Container widget.

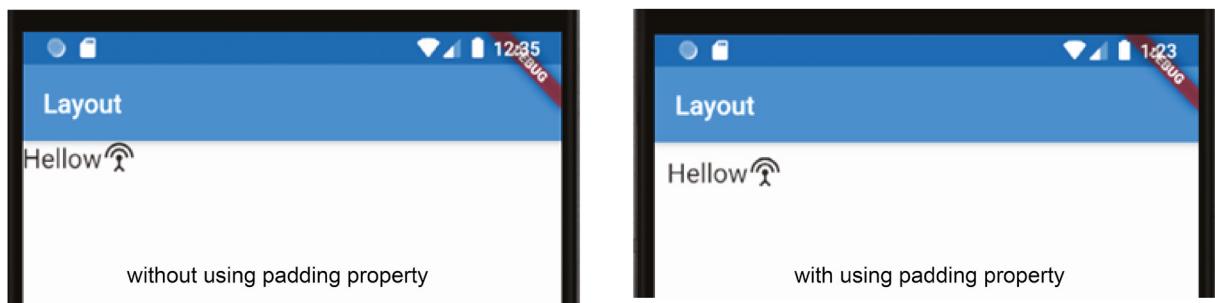
```
Row(  
  children: <Widget>[  
    Text(  
      'Hello',  
      style: TextStyle(fontSize: 20.0),  
    ),  
    Icon(Icons.settings_input_antenna),  
  ],  
>),
```

Add a Container widget as parent widget as follows:

```
Container(  
    padding: EdgeInsets.fromLTRB(10.0, 10.0, 10.0, 10.0),  
    child: Row(  
        children: <Widget>[  
            Text(  
                'Hello',  
                style: TextStyle(fontSize: 20.0),  
            ),  
            Icon(Icons.settings_input_antenna),  
        ],  
    ),
```

**LTRB** means **L**: Left , **T**: Top , **R**: Right , and **B** :Bottom

The following figures display how the text looks like with and without using the padding property.



Also you may use the following padding properties:

The following code gives you an idea about using the padding property for all directions:

```
padding: EdgeInsets.all(20.0),  
padding: EdgeInsets.symmetric(vertical: 20.0, horizontal: 20.0),  
padding: EdgeInsets.only(left:20,top:20,right:20,bottom:20),
```

## Padding Widget

This widget is used to wrap a Column , Row , Container or other widgets. This widget adds a padding size around the child widget.

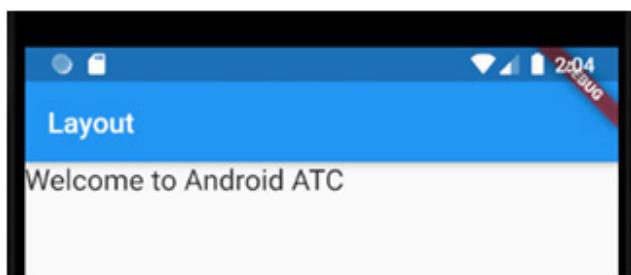
For example, we usually build our app interface of a group of parents and children widgets in tree structure, and we start with a Column widget then use Rows and Containers as illustrated in the following example:

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

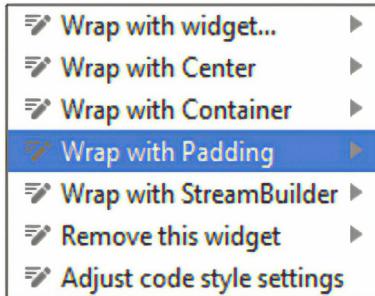
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Layout'),
        ),
        body: Column(
          children: <Widget>[
            Row(
              children: <Widget>[
                Container(
                  child: Text(
                    'Welcome to Android ATC',
                    style: TextStyle(fontSize: 20.0),
                  ),
                ),
              ],
            ),
            ],
          )),
    );
  }
}
```

The run output for this code follows:



However, if you double click the Column widget , and then press **Alt + Enter** ,or click the orange lamp icon , you will get the following wrap choices:



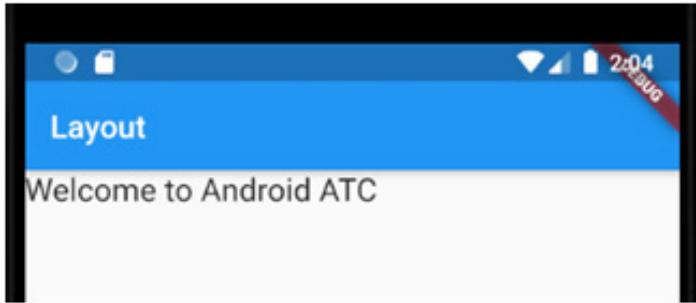
Select **Wrap with Padding** , then the Padding widget will wrap your Column widget and add 8 pixels around your Column widget contents.

The code will be following:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Layout'),
        ),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: Column(
            children: <Widget>[
              Row(
                children: <Widget>[
                  Container(
                    child: Text(
                      'Welcome to Android ATC',
                      style: TextStyle(fontSize: 20.0),
                    ),
                  ),
                ],
              ),
            ],
          )),
    );
}
```

{  
}

The run output after using the Padding widget will be as follows:

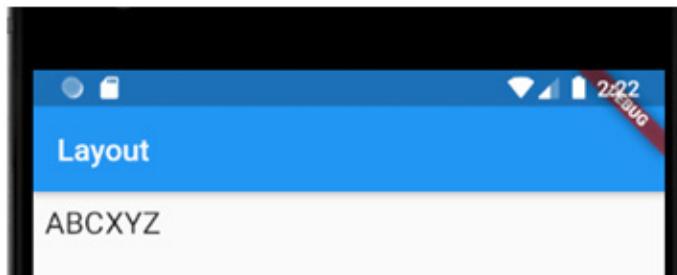


### SizedBox Widget

This widget helps us to have a specific width and/or height between widgets. For example, the following code includes two Text widgets in the same row.

```
Row(  
    children: <Widget>[  
        Text('ABC',style:TextStyle(fontSize: 20.0),),  
        Text('XYZ',style:TextStyle(fontSize: 20.0),),  
        ],  
    )
```

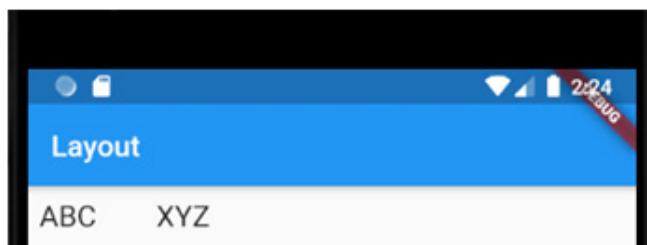
When you run this app , these two texts ( ABC & XYZ) will appear without any horizontal space as illustrated in the following figure:



While when you use the SizedBox widget with a specific width between these two Text widgets, it will be illustrated in the following code:

```
Row(
  children: <Widget>[
    Text('ABC',style:TextStyle(fontSize: 20.0),),
    SizedBox(width: 40,),
    Text('XYZ',style:TextStyle(fontSize: 20.0),),
  ],
)
```

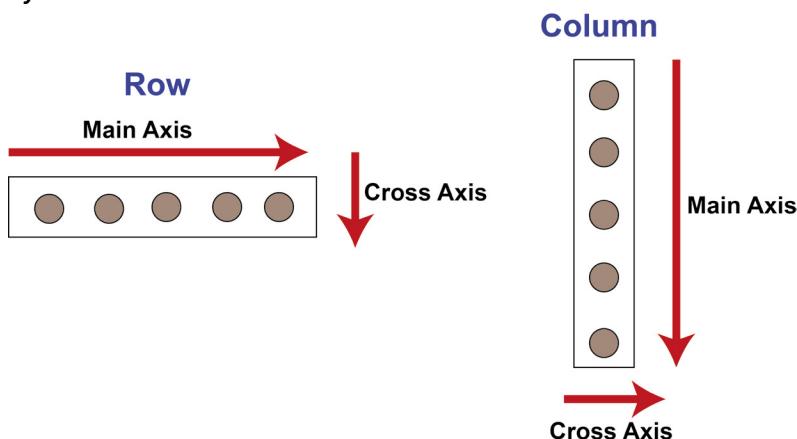
The run output will be as follows:



Also, you may use this `SizedBox` widget to add some vertical space between two `Row` widgets when you use the property of height.

## Aligning widgets

You may control how a `Row` or `Column` widgets aligns its children using the `mainAxisAlignment` and `crossAxisAlignment` properties. For a `Row` widget, as illustrated in the following figure, the main axis runs horizontally and the cross axis runs vertically. For a `Column` widget, the main axis runs vertically and the cross axis runs horizontally:



**Example:**

1- Create a simple Flutter app including the following code:

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

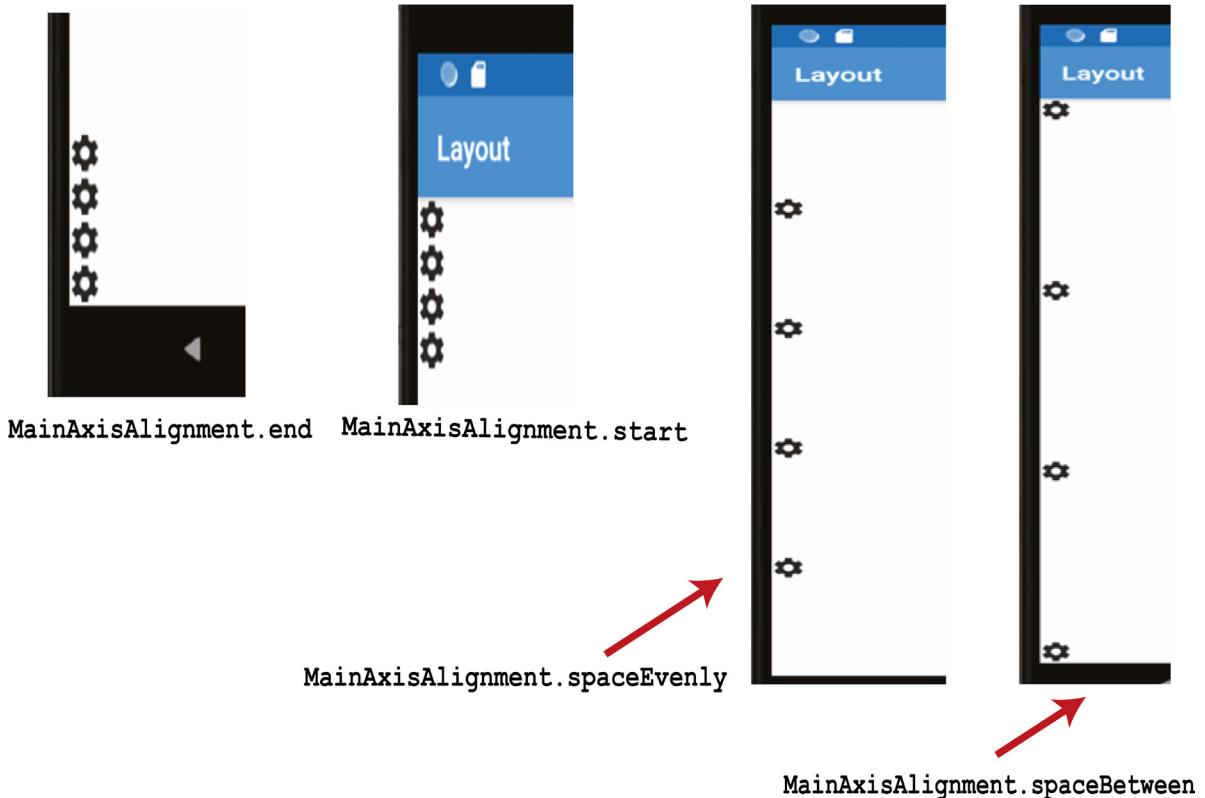
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Layout'),
        ),
        body: Column(
          mainAxisAlignment: MainAxisAlignment.spaceBetween,
          children: <Widget>[
            Icon(Icons.settings),
            Icon(Icons.settings),
            Icon(Icons.settings),
            Icon(Icons.settings),
          ],
        ),
      );
    );
  }
}
```

2- Then, run your app and replace the grey highlighted part of the code with the following values :

```
mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
mainAxisAlignment: MainAxisAlignment.start,  
mainAxisAlignment: MainAxisAlignment.end,
```

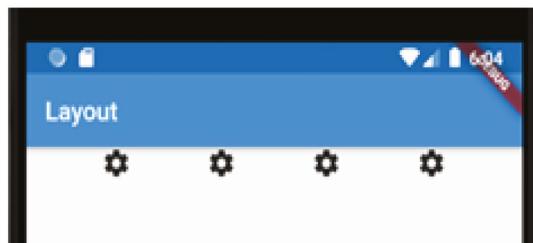
Check the app run output result for each change. You should get the following run outputs:



3- Replace the Column with a Row in the previous code and check the following aligning property values:

```
mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
mainAxisAlignment: MainAxisAlignment.spaceBetween,  
mainAxisAlignment: MainAxisAlignment.start,  
mainAxisAlignment: MainAxisAlignment.end,
```

The run output for each property value follows:



MainAxisAlignment.spaceEvenly



MainAxisAlignment.spaceBetween



MainAxisAlignment.start

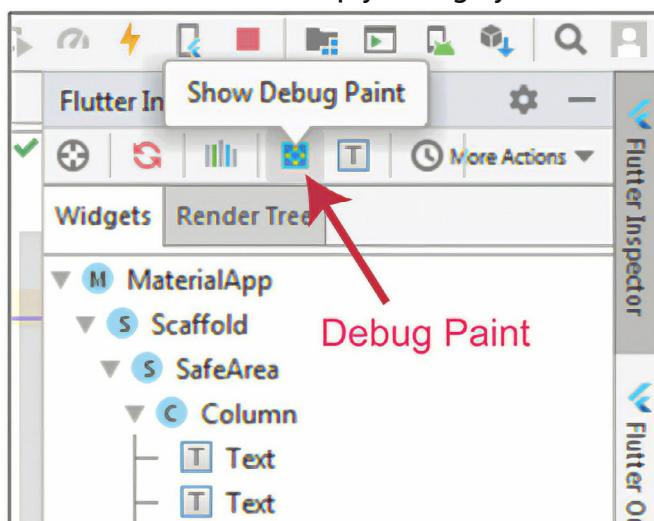


MainAxisAlignment.end

4- Try to use the properties below with the Row and Column widgets :

`crossAxisAlignment: CrossAxisAlignment.center,`  
`crossAxisAlignment: CrossAxisAlignment.stretch,`

In **Flutter Inspector** window, and as illustrated in the following figure, click **Debug Paint** button , and then check your phone emulator. You should get the vertical and horizontal axis for your phone emulator. These axis help you align your column and row contents.



In lesson 7, you will learn more about other layout widgets such as Stack widget.

## Card Widget

A Card class is a material design card. A card has slightly rounded corners and a shadow.

A Card widget is a sheet of material used to represent some related information, for example an album, a geographical location, a meal, contact details, etc.

This widget is similar to a Container widget , where it can include other children widgets such as images, text , and others. However, with this widget you cannot use the padding property to add space between the Card widget border and the card children widgets or Card widget content. While you can use a Padding widget to do this task as you will see in the next example.

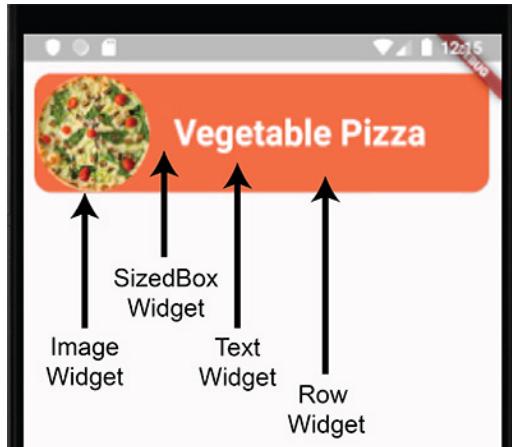
Continue using the same **main.dart** file, and perform the following steps:

1- **Delete** all the previous code, and set the **main.dart** file content as follows:

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

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        body: SafeArea(
          ),
        ),
      ),
    );
}
```

In this example, you will use the Card widget to get a design similar to the following figure. As you see here, the Card widget will be the parent widget for all the children widgets which are illustrated in the following figure. These children widgets are Row, Image, Text , and SizeBox widgets.



2- First, you should add the image : **pizza.png** file in the **images** folder which is on your **Project** navigation window using the copy and paste technique. You can find this image in the images folder (check your lab file folder for this course on your computer). Then click **Ok** to confirm adding this image.

3- Add a Card widget to your code as follow:

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

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        body: SafeArea(
          child: Card(),
        ),
      ),
    ),
  );
}
```

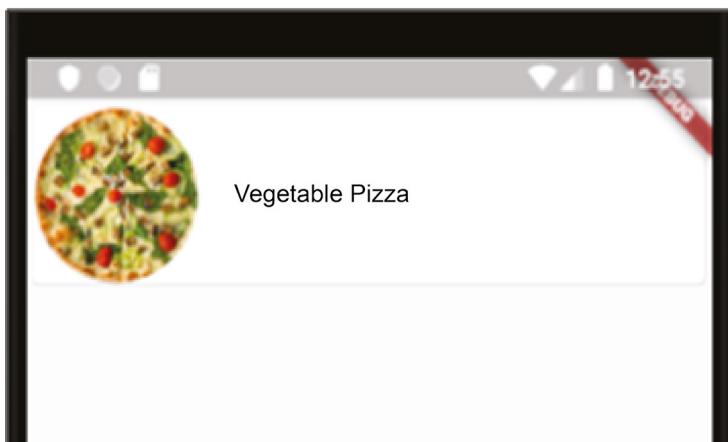
4- Add the Row widget as a child widget of the Card widget. Then, add the Text and Image widgets as children widgets of the Row widget. Then, add the SizedBox widget to make a space between the Image and Text widgets. The code will be as follows:

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

void main() {
```

```
runApp(  
  MaterialApp(  
    home: Scaffold(  
      body: SafeArea(  
        child: Card(  
          child: Row(  
            children: <Widget>[  
              Image(  
                image: AssetImage('images/pizza.png'),  
                width: 100.0,  
                height: 100.0,  
              ),  
              SizedBox(width: 20.0),  
              Text('Vegetable Pizza'),  
            ],  
          ),  
        ),  
      ),  
    ),  
  ),  
);  
}
```

The run output follows:



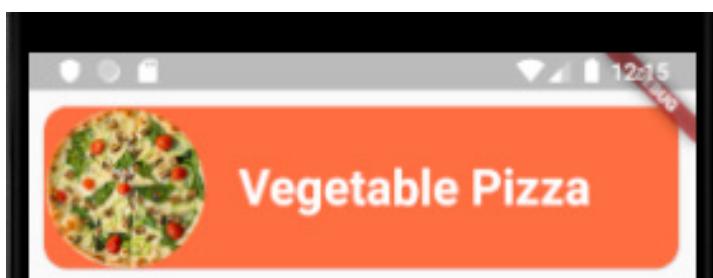
5- You should add some properties to your Card widget such as color, margin, and border.

Also, you may change the style of your Text widget such change the font weight , size and color as illustrated in the following code:

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

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        body: SafeArea(
          child: Card(
            shape:
              RoundedRectangleBorder(borderRadius: BorderRadius.circular(15)),
            margin: EdgeInsets.all(10.0),
            color: Colors.deepOrangeAccent,
            child: Row(
              children: <Widget>[
                Image(
                  image: AssetImage('images/pizza.png'),
                  width: 100.0,
                  height: 100.0,
                ),
                SizedBox(width: 20.0),
                Text('Vegetable Pizza',
                  style: TextStyle(
                    fontWeight: FontWeight.bold,
                    color: Colors.white,
                    fontSize: 30.0),),
              ],
            ),
          ),
        ),
      ),
    );
}
```

The run output follows:



## App Icons for iOS and Android Apps

If you click the home button of your phone emulator (iOS or Android), and check the apps list, you will find that your Flutter app icon is the Flutter logo. This is the default icon setting for any Flutter app.

In this part of the lesson, you will configure your app icon for iOS and Android devices. Also, this icon will represent your app on your customers' smart devices (iOS or Android), and in Apple and Google Stores.

You may ask a graphic designer to create your app icon, use Adobe Illustrator software, search on Google images, or use a free logo design software web site.

The App icon is an image which has an extension JPEG or PNG. It is better to use PNG image format because PNG supports transparency while JPEG does not.

I will use “AndroidATC.png” image in creating the icon image for this Flutter app for Android and iOS apps.

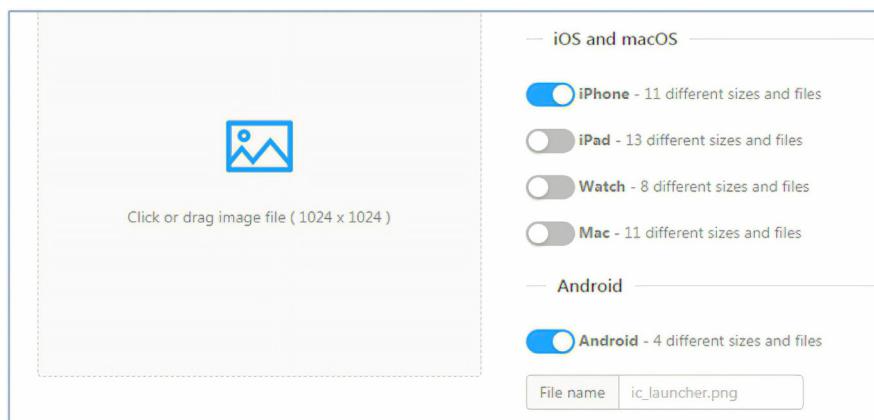


The images folder is a part of the lab files folder of this course. This folder includes this image and other images. If you do not find this folder on your computer, please contact our support team at support@androidatc.com, and ask them about the lab files for Flutter application development course.

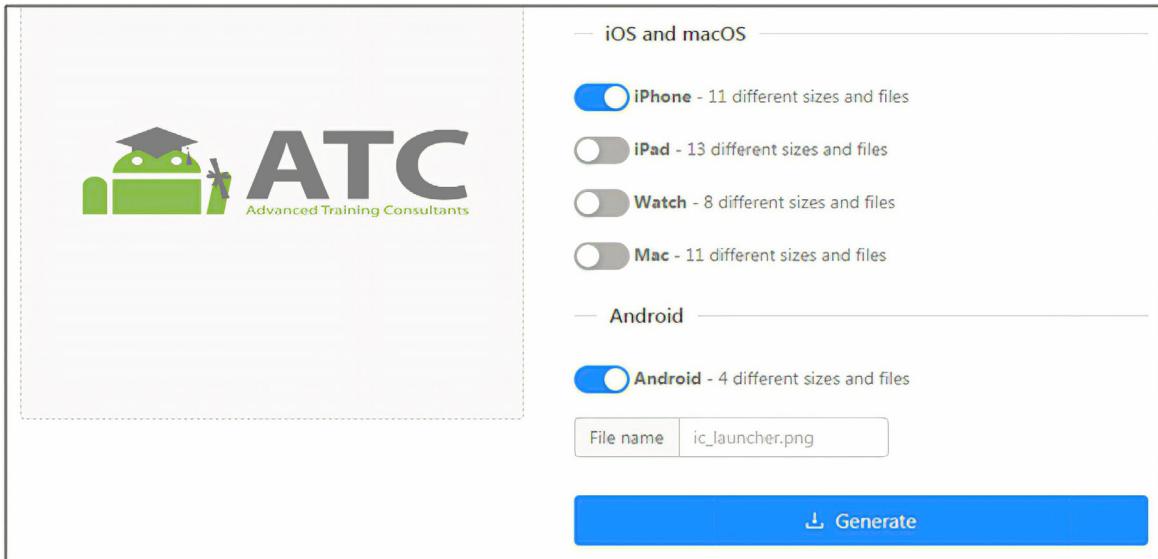
Continue using the previous code for **lesson\_05** project.

To create an app icon for Android and iOS apps, you need to have a PNG image, and then it is easier to use a free web site such as : [appicon.co](http://appicon.co) or any other web site to generate icons for your iOS and Android apps. To add an icon image for your app, perform the following steps:

1- Go to [appicon.co](http://appicon.co) web site, and as you see in the following figure. Select only iPhone and Android.



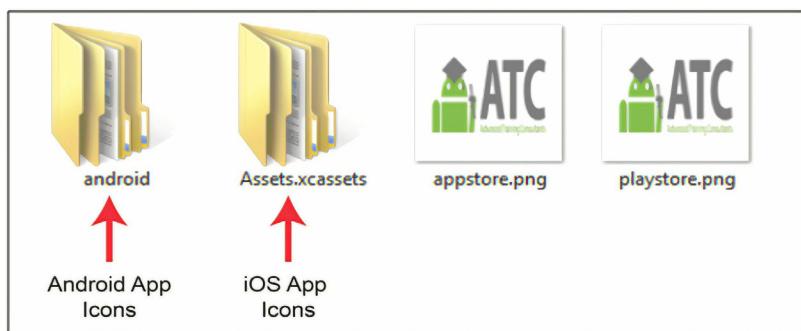
2- Click the image box, select your image as illustrated in the following figure, and then click **Generate** button.



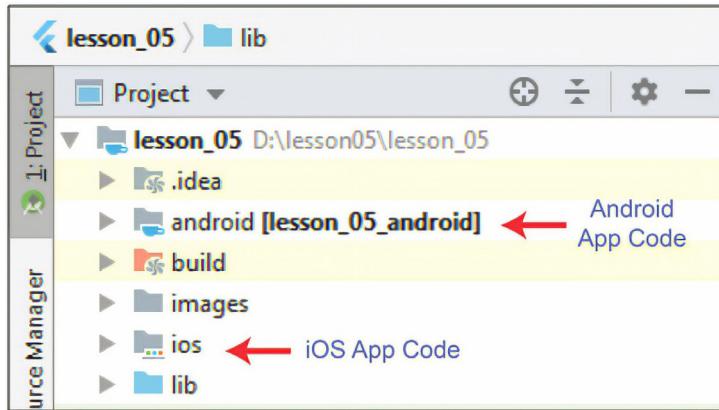
3- This web site will automatically download a compressed file including the Android and iOS app icons.

If you have any problem with this step or in this web site, these icon files are in the images folder with the lab files folder inside a folder called **AppIcons.zip**.

4- Right click the **AppIcons.zip** file, and then select **Extract files**. Extract the content of this folder inside a new folder called : **Icons** in the following path **C:\Icons** or with any other name and at any other location on your computer. You will have two folders. “android” folder which includes the Android app icons , and “ Assets.xcassets” folder which includes the iOS app folders as illustrated in the following figure:

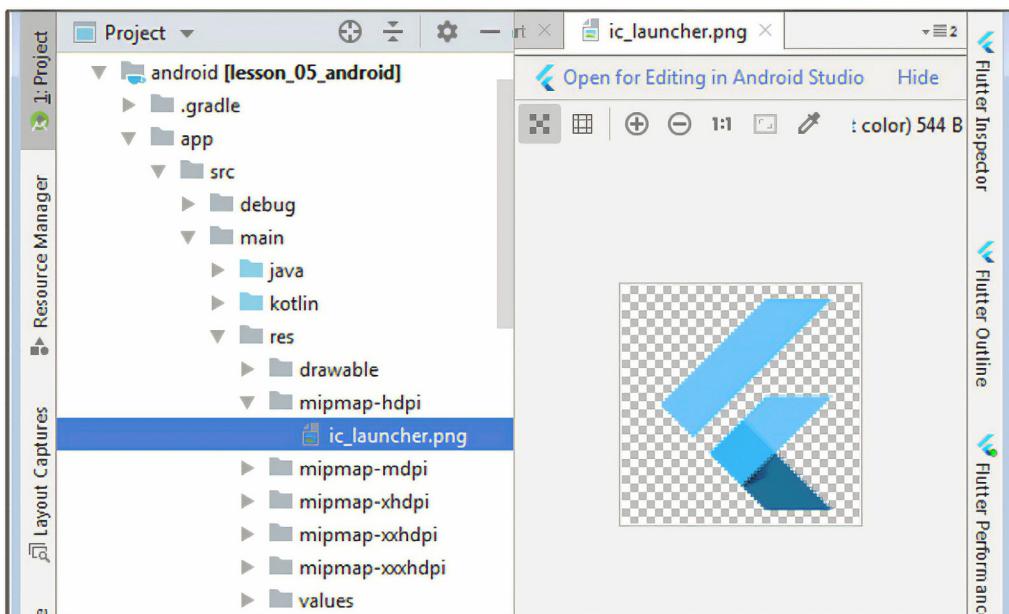


Remember, when you are building your Flutter app code in **main.dart** file or in another file, the Android Studio simultaneously builds your **Android** app files in “**android**” folder and your **iOS** app files in the “**ios**” folder as illustrated in the following figure:



This means you should add your Android app icon inside the “**android**” folder, and your iOS app icon inside the “**ios**” folder.

5- To add an icon to your Android app, at the **Project** console, and as illustrated in the following figure, go to **android** → **app** → **src** → **main** → **res**



You will find the default app icons in different sizes , and as you see here, the Flutter icon is the default app icon for your app. To configure your app icons, just replace these icons (images) with your icons which you got in the previous step. To do that: Right click “**res**” folder , and then select **Show in Explorer** for Windows, or select **Reveal in Finder** for Mac O.S.

6- Open **res** folder, and **Delete** the folders which start with **mipmap** (**mipmap-hdpi** , **mipmap-mdpi**, **mipmap-xhdpi**, **mipmap-xxhdpi**,and **mipmap-xxxhdpi**). There are five folders.

7- Now, you will add your icons folders to the same location from where you deleted the previous folders. Go to : **C:\Icons\android** , **Copy** all folders (five folders) , and

Paste them in the **res** folder.

Check your **res** folder using your Android Studio (**android → app → src → main → res**) , and you should find that your Android app uses the new app icons.

Now, you will repeat the same steps as followed previously, but for iOS app as follows:

8- On Android Studio, in **Project** console, go to:

**ios → Runner → Assets.xcassets → AppIcon.appiconset**

You will find all the iOS default icons in different sizes. Right click : **AppIcon.appiconset** and then select **Show in Explorer** for Windows, or select **Reveal in Finder** for Mac O.S.

9- Open **AppIcon.appiconset** folder, and **Delete** all these folder files.

10- Copy all your iOS files (images and files) which you have at this path : **C:\Icons\Assets.xcassets\AppIcon.appiconset**, and then Paste them inside the **AppIcon.appiconset** folder.

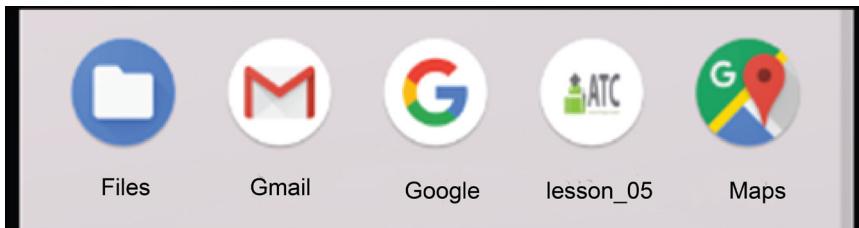
Now, check your iOS app icons which are available in the following path :

**ios → Runner → Assets.xcassets → AppIcon.appiconset**, you should find your icons.

11- To test if your phone emulator is using your app icon, just stop your App by clicking the stop button as illustrated in the following figure, and then run your app again.

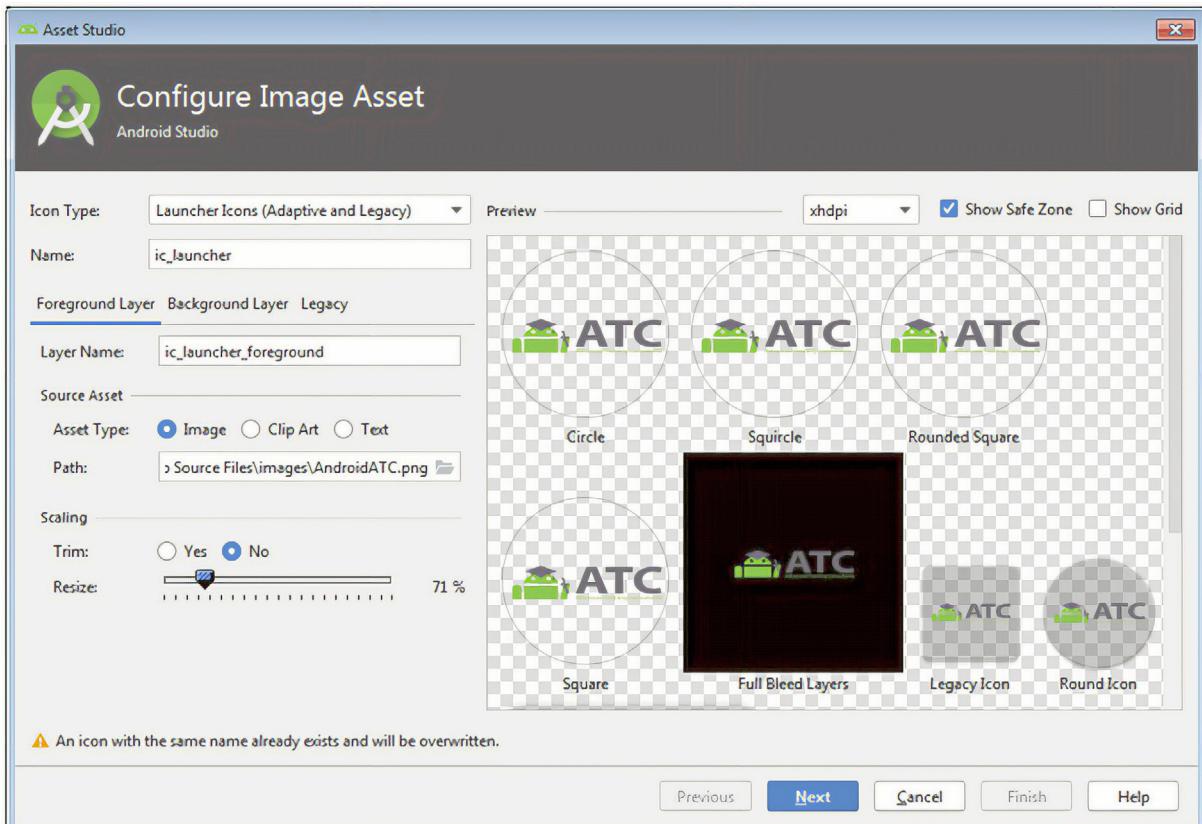


12- Click your phone emulator home button, and check your app list. You will find your app icon with your phone emulator apps list as illustrated in the following figure:



13- For Android app only, if you want to make more changes in your app icon such as making the image stretches all the icon circle. In Android Studio, right click the **android → New → Image Asset**

14- You will get the following figure. Click the folder icon at the Path: and select your original PNG image which you have used to generate the icons. In the Scaling part, slide the arrow to resize your image and to get the desired size of your icon. Be sure your image is still inside the circle, click **Next** , and then click **Finish**.



15- Now, stop your app , and then run it again. You will find the new Android app icon image filling more space of the circle icon, and it is clearer.

To get more information about Google Play icon design specifications, check the following URL:

<https://developer.android.com/google-play/resources/icon-design-specifications>.

To get more information about Apple icon design specifications, check the following URL:

<https://developer.apple.com/design/human-interface-guidelines/ios/icons-and-images/app-icon>

## Hot Reload and Hot Restart

Flutter's hot reload feature helps you experiment quickly and easily, add features, and fix bugs. Hot reload works by injecting updated source code files and changes into your running phone emulator. After the phone emulator updates classes or widgets with the new versions of fields and functions, the Flutter framework automatically rebuilds the widget tree, allowing you to quickly view the effects of your changes on your emulator.

Hot reload allows you to almost instantaneously see your changes in your emulator or on your device that you are testing it on. This means, you don't need to stop and then re-run your app.

Most types of code changes can be hot reloaded.

**Example:**

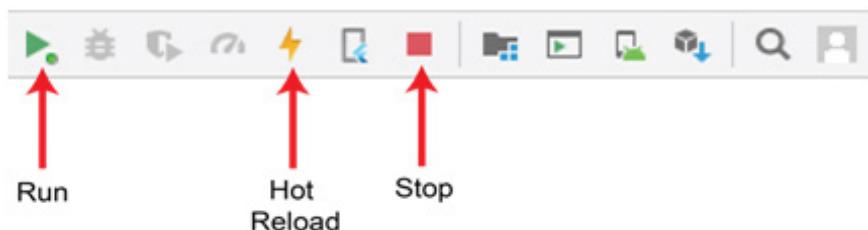
Continue using the previous code in lesson05 project, and perform the following steps :

1- Make your code as follows, and delete the other code:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      backgroundColor: Colors.green,
      body: Container(),
    ),
  )));
}
```

2- Stop and then run your app using the Android Studio toolbar as illustrated in the following figure. You will get an empty app with a green background.



3- Now, change your app background color to red color in your app code.

Replace :

backgroundColor: Colors.green,

with:

backgroundColor: Colors.red,

And then, click the hot reload button. No change will occur in your app background color. It will still be green. However, if you stop your app, and run it again, the background color will change to red.

Why this is happening ? This will be explained in this section.

## How hot-reload works ?

When hot-reload is invoked, the host machine checks the edited code since the last compilation and recompiles the following libraries:

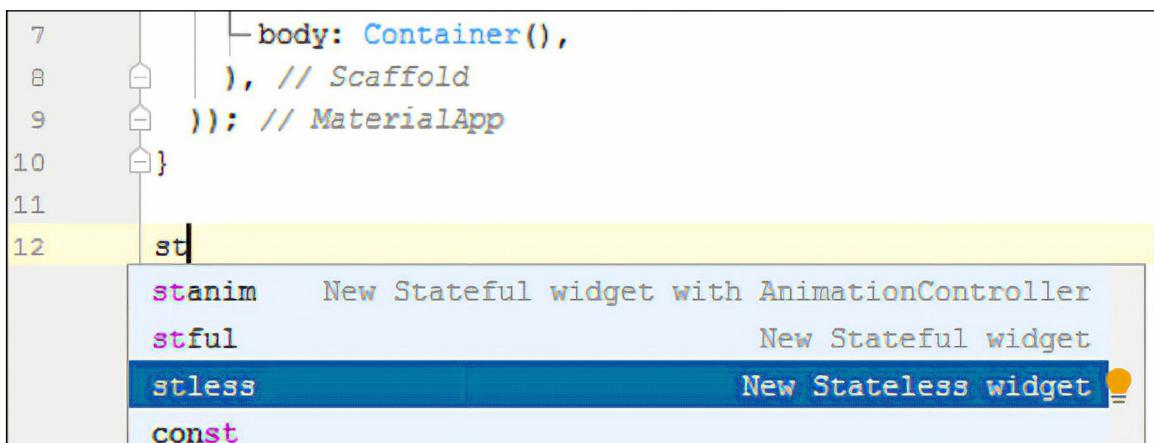
- Any libraries with changed code.
- The application's main library.
- The libraries from the main library leading to affected libraries.

The updated source codes are sent into a running phone emulator. The phone emulator then reloads all the libraries with an updated code.

In the previous example, you modified the code of the Scaffold widget , but this widget version did not change. Therefore, when you clicked the hot reload button on Android Studio tools, Flutter SDK instead of reloading your app from scratch, it searched on your app whether if there is any change in your widgets versions to push these changes only to your running emulator.

If you change your widget status to *Stateless widget*, this will make your widget version compatible with any change to its properties. To change the Scaffold widget to stateless widget, perform the following change in your code.

1- Type outside the main function : **st** , and then select **stless** from the shortcut menu to create a new Stateless widget as illustrated in the following figure:





The next section of this lesson includes more information about the **Stateful** and **Stateless** widgets

2- Name this new stateless widget : **MyApp**

3- Cut the following code of the main () function :

```
MaterialApp(  
  home: Scaffold(  
    backgroundColor: Colors.green,  
    body: Container(),  
)  

```

4- Use paste command to replace your code with the Container(); widget in the new stateless widget (MyApp). Delete the extra bracket in MyApp class.

The full code will be as follows:

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(  
}  

```

5- Now , add your class name “MyApp” to the main() function as illustrated in the grey highlighted part of the following code:

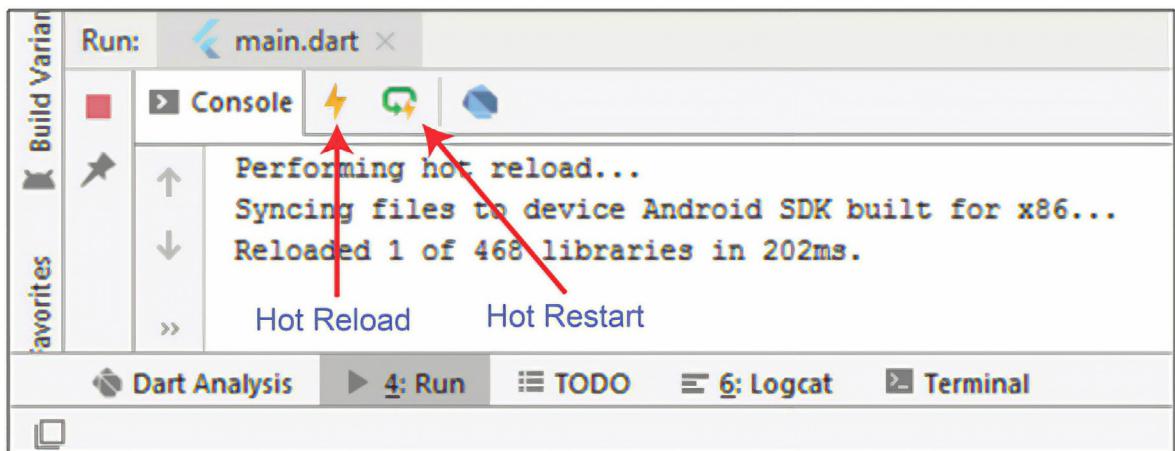
```
void main() {
  runApp(
    MyApp(),
  );
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        backgroundColor: Colors.green,
        body: Container(),
      ),
    );
  }
}
```

6- Now, you should stop your app, and then run it again to be sure that the app which is running on your emulator is MyApp.

7- Change the background color of the Scaffold widget to red color, and then click the hot reload button. Then, you will find that your app user interface has directly been changed to red .

Also, you can use the hot reload button which is available at the run console as illustrated in the following figure:



Now, after you make these changes, you can get the same result of hot reload if you save your app (**Ctrl + S** for Windows) or (**command + S** for Mac).

## Hot Restart

In the same Run console, there is a Hot Restart button. This option resets your app state such as delete any changes in a form data input or counter status, and starts your app from the beginning, without the need to upload all the app data from Android Studio to your emulator. It takes time less than stopping your app and restarting it.

If you are testing something like a form, where you don't want to lose the data that you have used to test the app, but you want to make some changes in your app user interface, just click the hot reload, and then you will not lose this data. However, if you click hot restart, the changes will be tested, but you will lose this data.

## Stateful and Stateless Widgets

A widget is either stateful or stateless. If a widget can change—when a user interacts with it, for example—it is stateful.

A **stateless** widget never changes. Icon, IconButton, and Text are examples of stateless widgets. Stateless widgets subclass StatelessWidget.

A **stateful** widget is dynamic: for example, it can change its appearance in response to events triggered by user interactions or when it receives data. Checkbox, Radio, Slider, Form, and TextField are examples of stateful widgets. Stateful widgets subclass StatefulWidget.

A widget's state is stored in a State object, separating the widget's state from its appearance. The state consists of values that can change, like a checkbox is checked. When the widget's state changes, the state object calls setState(), telling the framework to redraw the widget.

In the next example, you will see how Stateful widget works.

To create a Stateful widget, you should create two classes, a class of StatefulWidget and a class of State. The state class contains the widget mutable state and the widget build() method.

When the widget's state changes, the state object calls setState(), telling the framework to redraw the widget.

**Example:**

In the following example , you will create a simple Flutter app to test the difference between the Stateful and Stateless widgets.

Perform the following steps:

1- Open Android Studio

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** , and then click **Next**.

4- Type : **stateful\_stateless\_widgets** for Project Name , and create a new folder : Lesson\_05 for Project Location. Click **Next**.

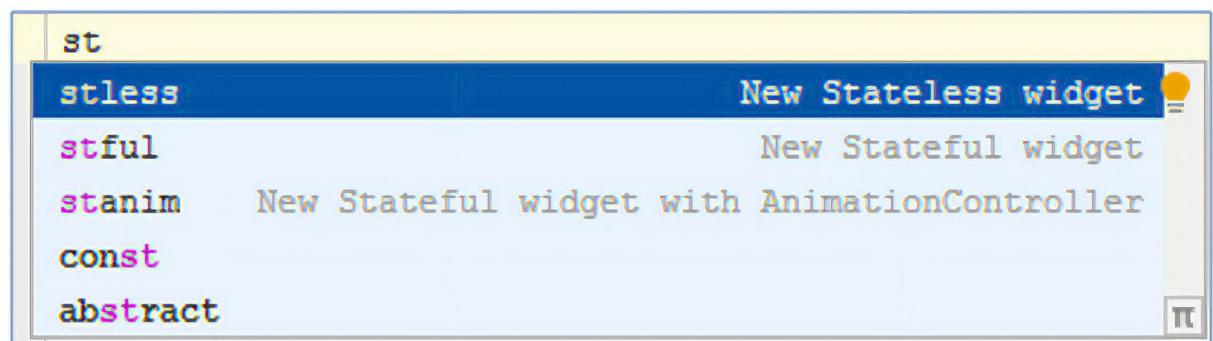
5- Type : **androidatc.com** for Company domain, and then click **Finish**

6- Open **main.dart** file. and delete all the code , and Type the flowing code :

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

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

Then, add a stateless widget by typing **st** , and then select **stless** of the shortcut menu as illustrated in the following figure:



7- Name the Stateless widget : **MyApp** as illustrated in the grey highlighted part of the following code:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

8- Configure this app to use a **Scaffold** widget which includes the basic material design of the visual layout structure of the flutter app. The code follows:

```
import 'package:flutter/material.dart';
void main() {
  runApp(
    MyApp(),
  );
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Statfeul and Stateless Widgets'),
        ),
      ),
    );
  }
}
```

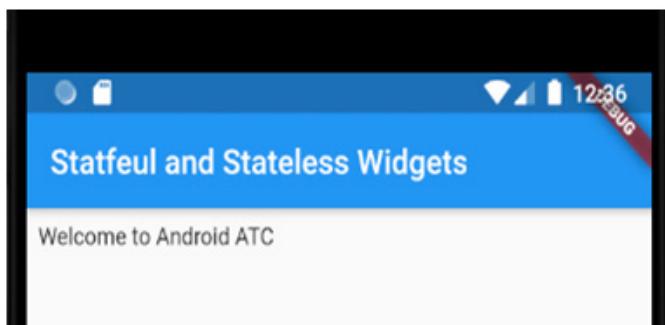
9- Add a Column widget as body of your app and wrap it with a Padding widget. The code follows:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Statfeul and Stateless Widgets'),),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: Column(
            children: <Widget>[
              Text('Welcome to Android ATC'),
            ],
          ),
        ),
      );
    }
}
```

The run output follows:



As you see, the StatelessWidget widget with Text widget runs fine.

10- Now, in this step, you will add a button (Raised Button) to your app. In the next lesson, you will learn in detail about the different types of buttons and how to configure them. Now, in brief , by adding the following code you can add a Raised button to your app:

```
RaisedButton(  
    child: Text('Click Me'),  
    onPressed: () { },  
)
```

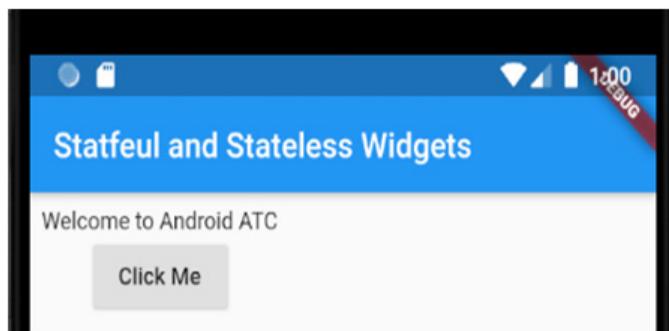
The Text widget here represents the button title. When you press this button , the code which you will type between the two braces {} of the onPressed method will run.

The following code includes a RaisedButton, and the onPressed method includes an IF statement which depends on its work on an integer variable called x which equals 1 , and it is configured outside the StatelessWidget widget as illustrated in the grey highlighted part of the following code:

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(  
    MyApp(),  
  );  
}  
  
int x = 1;  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text('Statfeul and Stateless Widgets'),  
        ),  
        body: Padding(  
          padding: const EdgeInsets.all(8.0),  
        ),  
      ),  
    );  
  }  
}
```

```
        child: Column(
            children: <Widget>[
                Text('Welcome to Android ATC'),
                Container(
                    child: RaisedButton(
                        child: Text('Click Me'),
                        onPressed: () {
                            if (x > 5) {
                                print("Hello, I am If statement ");
                            } else {
                                print("Hello, I am Else statement ");
                            }
                        },
                    ),
                ),
            ],
        )));
    );
}
```

11- Run your app , you will get the following run output:



12- Click the button, you will get the following result on the run console:

```
Performing hot restart...
Syncing files to device Android SDK built for x86...
Restarted application in 1,475ms.
I/flutter (14832): Hello, I am Else statement
```

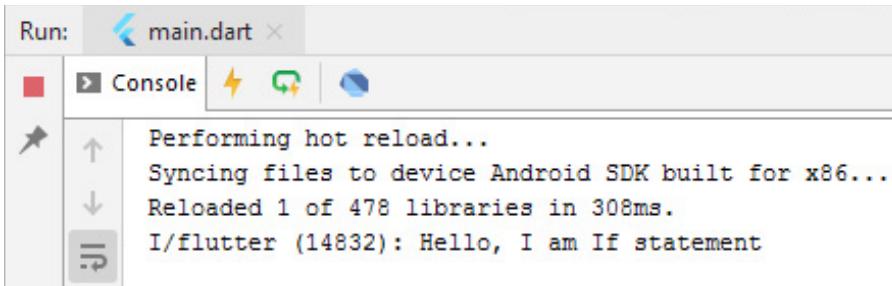
13- Replace the value of the variable **x** with **40** , and then click the **Hot Reload** button  
You will get the same run result : **Hello, I am Else statement**

This means that variable **x** still has value =1 in the app framework , because the hot reload button reloaded only the stateless widget content of this app.

In the previous code, move the **x** variable to inside the **MyApp** stateless widget as illustrated in the following figure:

```
class MyApp extends StatelessWidget {
    int x = 40;
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
```

Now, click the **Hot Reload** button, and then click the **Click Me** button on your app, and the result will be as follows:



Now, to know more about the difference between stateful and stateless widgets, perform the following steps using the same previous app:

14- Right click your app directory, and then select **New → New Directory** . Type **images** for the directory name, and then click **OK**.

15- Configure the **pubspec.yaml** file to use this **images** folder for app images as you did previously in this lesson in the image widget section.

16- Open the **Lab Source Files\Images** on your computer, copy ( **pizza1.png** & **pizza2.png** ) images , and then **paste** them in the **images** directory in Android Studio.

17- Delete the **IF** statement code , and add the **Image** widget as a child widget of the **Column** widget.

The code will be as follows:

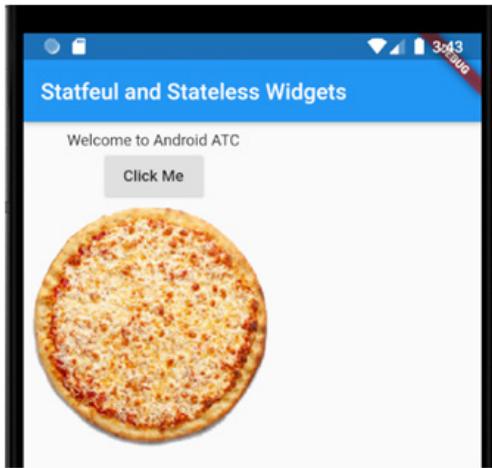
```
import 'package:flutter/material.dart';

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

int x = 1;

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Statfeul and Stateless Widgets'),
        ),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: Column(
            children: <Widget>[
              Text('Welcome to Android ATC'),
              Container(
                child: RaisedButton(
                  child: Text('Click Me'),
                  onPressed: () {}),
              ),
              Image.asset('images/pizza$x.png'),
            ],
          ),
        );
      );
    }
}
```

Note that, the image value depends on the variable **x** , and **x=1**, then when you run this app you will get the cheese pizza (**pizza1.png**) as illustrated in the following figure. But, if **x=2**, this means that the vegetable pizza image (**pizza2.png**) will display.



18 - Add to the button action: `setState()` method and make it change the value of variable `x` from **1** to **2** when the app user clicks this button. This means this app contains dynamic contents now.

Make the code of this button as follows:

```
Container(  
    child: RaisedButton(  
        child: Text('Click Me'),  
        onPressed: () {  
            setState(() {  
                x = 2;  
            });  
        },  
    ),
```

Here, you will get a **red underline** with the `setState` method because you are using `StatelessWidget` widget. Double click the `StatelessWidget`, click the orange lamp icon, and then select `Convert to StatefulWidget` as illustrated in the following figure:



Now, everything works fine, and the red underline is removed.

19- Run your app, and click the app button (**Click Me**) .

The `setState` method will update the app system framework with the new value of variable `x` ; therefore, the image will be updated to `pizza2.png`.

We conclude that, if you are creating a user interface where the state of the widget is not going to change (static content), then you should use a `Stateless` widget. But, if your app content is interactive (dynamic content), such as a user taps on a button or makes a connection with a database that produces changes or updates on the app interface , then you need to use a `stateful` widget.

## Use a Custom Font

Although Android and iOS offer high quality system fonts, one of the most common requests from designers is for custom fonts.

Flutter works with custom fonts and you can apply a custom font across an entire app or to individual widgets. As you did before in this lesson in adding an image to your app, you can do the same technique for importing a new font to your Flutter project. This recipe creates an app that uses custom fonts with the following steps:

- a. Import the font files.
- b. Declare the font in the `pubspec.yaml`.
- c. Use a font in a specific widget.

To achieve these steps, perform the following steps:

Continue using the previous code for lesson05 project.

1- To import a new font , you should go to : <https://fonts.google.com> web site. You will find more than 900 fonts that can be used for commercial use.

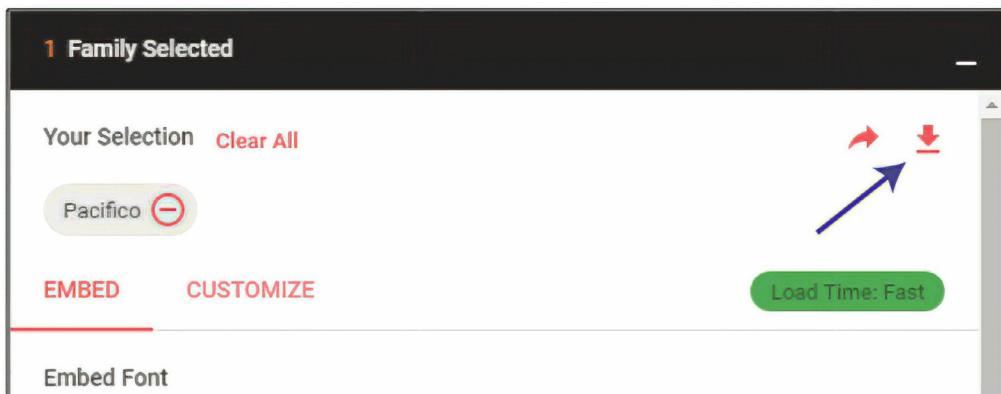
You can search about your desired font using the font name or category. For example, as illustrated in the following figure, click **Categories** , and then select only **Handwriting** category.

The screenshot shows the Google Fonts interface. In the search bar, 'Pacifico' is typed. Below the search bar, under 'Font Properties', the 'Handwriting' category is selected. On the left, a sidebar shows categories: Serif, Sans Serif, Display, Handwriting (which is checked), and Monospace. A preview area displays the text 'blue. cloudless p dark' in the Pacifico font. To the right, the Pacifico font is listed with its subfamily 'Dancing Script Impallari Type (4 styles)'. A red plus sign (+) button is located next to the font name. Below the font listing, there is a sample text: 'I watched the storm, so beautiful yet terrific.' At the top right, there are navigation links: CATALOG, FEATURED, ARTICLES, and ABOUT.

2- Click the plus sign (+) for **Pacifico** font



3- Click the pop up black rectangle window, and then click the download arrow to download this font file as illustrated in the following figure:

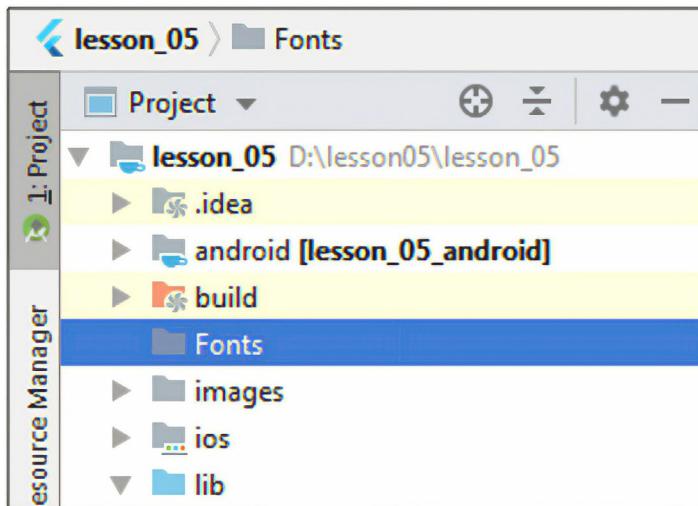


4- Extract the downloaded file, and you will get two files. The text file is the licensed file , and you import the **Pacifico-Regular.ttf** file into your Flutter project.

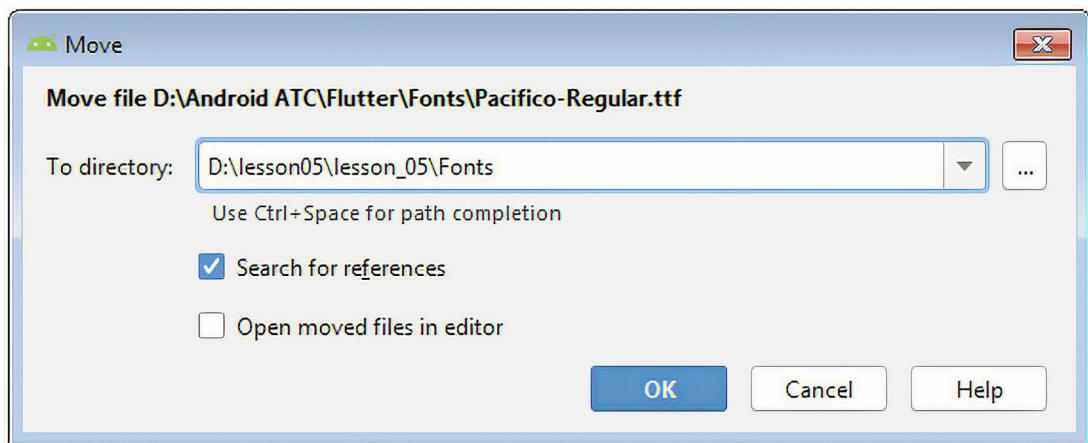
5- Open Your Android Studio, right click your project name, and then select:  
**New → Directory**

type **Fonts** for the directory name, and then click **OK**.

Your project structure follows:



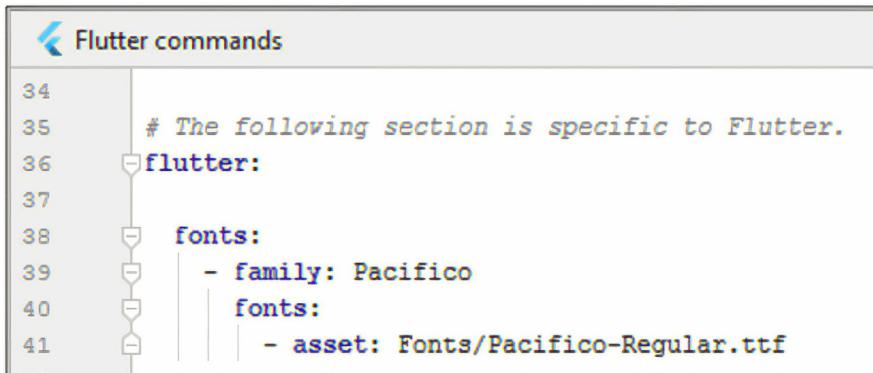
6- Copy your font file “ **Pacifico-Regular.ttf** ” and paste it inside the **Fonts** folder. Click **OK** to confirm the previous step as illustrated in the following figure:



7- Now, as you did before when you imported an image to your project, you must declare the font file in the **pubspec.yaml** file. To do that, double click : **pubspec.yaml** file , and then add the following configuration below **flutter**:

```
flutter:  
  
  fonts:  
    - family: Pacifico  
      fonts:  
        - asset: Fonts/Pacifico-Regular.ttf
```

For importance, keep your font configurations with the same arrangement as illustrated in the following figure:



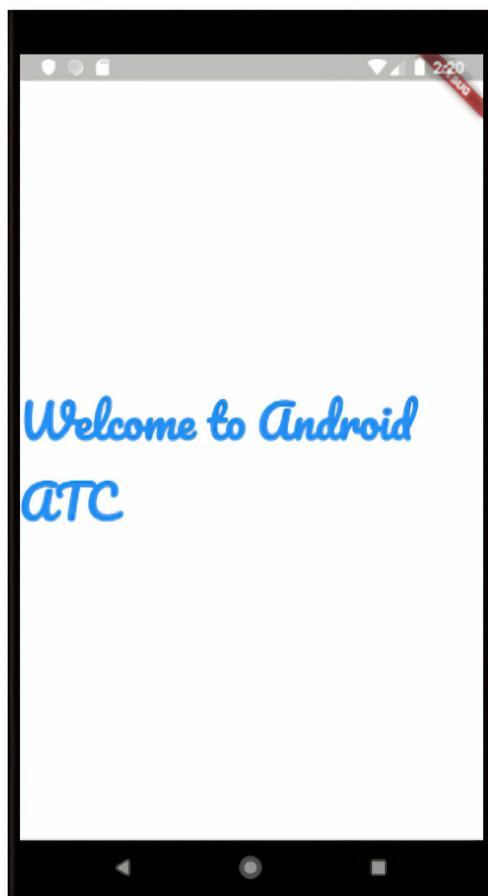
8- Click **Package get** to apply these changes in **pubspec.yaml** file.

9- Now, you can use this font with your text. The following code displays using this font in a Text widget:

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(  
    MyApp(),  
  );  
}  
  
class MyApp extends StatelessWidget {  
@override  
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(backgroundColor: Colors.white,  
      body: SafeArea(  
        child: Center(  
          child: Container(  
            child: Text('Hello World!', style: TextStyle(fontFamily: 'Pacifico'))  
        ),  
      ),  
    ),  
  );  
}
```

```
        child: Text('Welcome to Android ATC', style:  
          TextStyle(fontSize: 40,  
            color: Colors.blue,  
            fontFamily: 'Pacifico',  
            fontWeight: FontWeight.bold),  
        ),  
      ),  
    ),  
  ),  
));  
}  
}  
}
```

The run output follows:



## Lab 5: Creating a Restaurant Menu

In this lab, you will create a Flutter app of one screen only including a list of three Card widgets. Each Card widget includes a type of food. Then, you will add another feature to this app where the app user will move to another app screen when he/she taps any of these Card widgets.

Perform the following steps to build your app step by step:

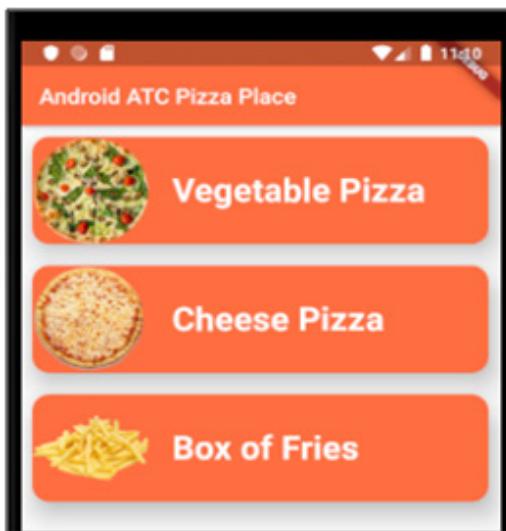
1- Open Android Studio, and select **File → New → New Flutter Project**

2- Enter the following:

Project Name: **lab\_05** , and then click **Next**

Company domain name : **androidatc.com**,and then click **Finish**

3- Your plan is to create an app interface including three Card widgets as illustrated in the following figure:



This means, you want to start with the basic Flutter app elements such as Scaffold, AppBar, and SafeArea widgets. Then, you will add a Column widget which will include these three Card widgets.

Now, start by creating the basic app widgets.

4- Open **main.dart**, delete all the content, and then add the following code:

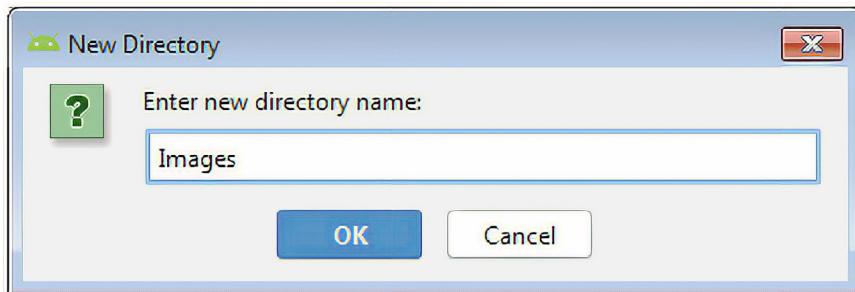
```
import 'package:flutter/material.dart';

void main() {
  runApp(
    lab5(),
  );
}

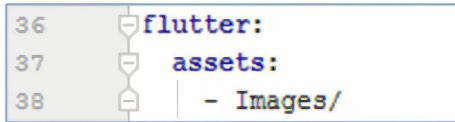
class lab5 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Android ATC Pizza Place'),
          backgroundColor: Colors.deepOrangeAccent,
        ),
        backgroundColor: Colors.white,
        body: SafeArea(
          child: Column(),
        ),
      ),
    );
  }
}
```

5- You need to use three images in your app. First, create a new directory called **Images** to your project folders structure, and then configure your app **pubspec.yaml** file to use this directory (**Images**) for images.

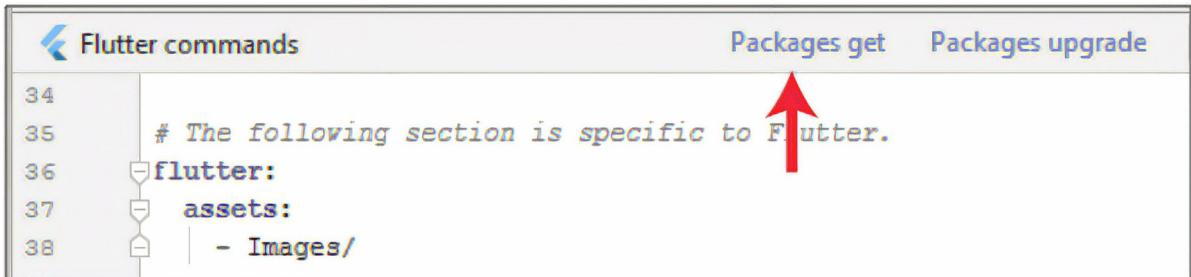
Right click the project name **lab\_05**, and then select **New → Directory**. Type **Images** for the directory name, and then click **OK** as illustrated in the following figure:



6- Open **pubspec.yaml** , and at the **flutter:** section, add the following code as it is exactly illustrated in the following figure , and be sure there are two spaces in the left margin:



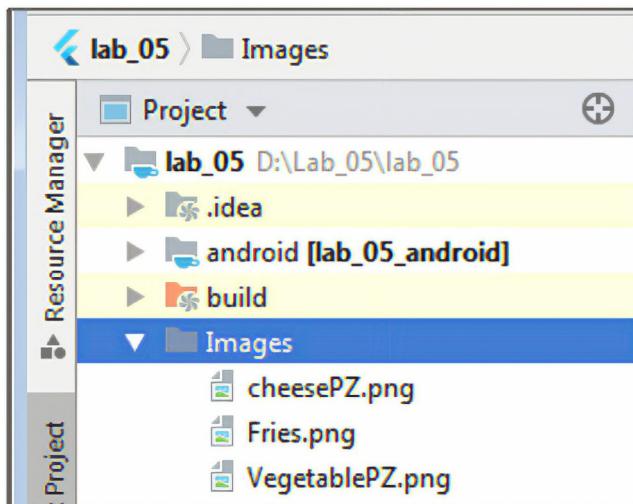
7- Click **Packages get** at the top of the **pubspec.yaml** file of the code editor page of Android Studio as illustrated in the following figure:



8- Now, open the **Lab files folder** on your computer, open **Images → Lab5 .** and paste the three images stated below from **Images\Lab5** in the **Images** folder in your app.

- VegetablePZ.png
- cheesePZ.png
- Fries.png

You should have the following file structure for your app Images folder:



9- Now, add the first **Card** widget to your app as a child widget of the **Column** widget. This **Card** widget will include **Image** and **Text** widgets in addition to properties of

shape, margin, elevation and color. The code follows:

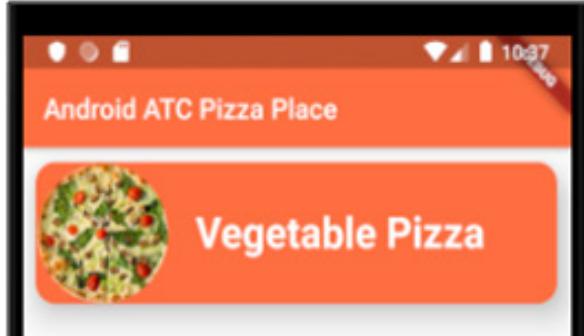
```
import 'package:flutter/material.dart';

void main() {
  runApp(lab5());
}

class lab5 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Android ATC Pizza Place'),
          backgroundColor: Colors.deepOrangeAccent,
        ),
        backgroundColor: Colors.white,
        body: SafeArea(
          child: Column(children: <Widget>[
            Card(
              shape: RoundedRectangleBorder(
                borderRadius: BorderRadius.circular(15)),
              margin: EdgeInsets.all(10.0),
              color: Colors.deepOrangeAccent,
              elevation: 20.0,
              child: Row(
                children: <Widget>[
                  Image(
                    image: AssetImage('Images/VegetablePZ.png'),
                    width: 100.0,
                    height: 100.0,
                  ),
                  SizedBox(width: 20.0),
                  Text('Vegetable Pizza',
                    style: TextStyle(
                      fontWeight: FontWeight.bold,
                      color: Colors.white,
                      fontSize: 30.0),
                  ),
                ],
              ),
            ],
          ],
        ),
      ),
    );
  }
}
```

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

10- Run your app now. You should get the following app interface:



11- Now , add the other two Card widgets as children widgets of the main Column widget. You can Copy only previous Card widget code (between Card and //Card), and then paste it twice, to get two more Card widgets as illustrated in the following code:

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(lab5(),);  
}  
  
class lab5 extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text('Android ATC Pizza Place'),  
          backgroundColor: Colors.deepOrangeAccent,),  
  
        backgroundColor: Colors.white,  
  
        body: SafeArea(  
          child: Column(children: <Widget>[  
            Card(  
              shape: RoundedRectangleBorder(  
                borderRadius: BorderRadius.circular(15)),  
              margin: EdgeInsets.all(10.0),  
              color: Colors.deepOrangeAccent,  
              elevation: 20.0,
```

```
        child: Row(
            children: <Widget>[
                Image(image: AssetImage('Images/VegetablePZ.png'),
                    width: 100.0,
                    height: 100.0,),

                SizedBox(width: 20.0),


                Text('Vegetable Pizza',
                    style: TextStyle(
                        fontWeight: FontWeight.bold,
                        color: Colors.white,
                        fontSize: 30.0),
                ),
            ],
        ),
    ),
),

Card(
    shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(15)),
    margin: EdgeInsets.all(10.0),
    color: Colors.deepOrangeAccent,
    elevation: 20.0,


    child: Row(
        children: <Widget>[
            Image(image: AssetImage('Images/VegetablePZ.png'),
                width: 100.0,
                height: 100.0,),

            SizedBox(width: 20.0),


            Text('Vegetable Pizza',
                style: TextStyle(
                    fontWeight: FontWeight.bold,
                    color: Colors.white,
                    fontSize: 30.0),
            ),
        ],
    ),
),

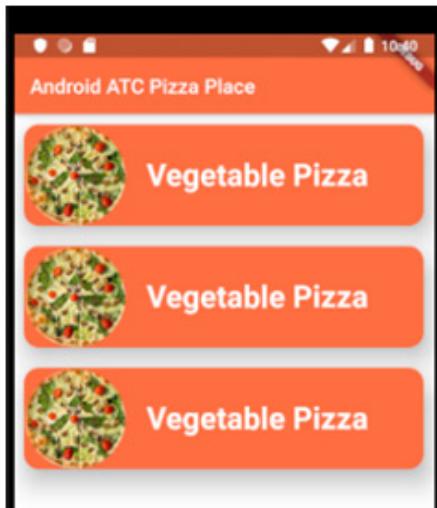
Card(
    shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(15)),
    margin: EdgeInsets.all(10.0),
    color: Colors.deepOrangeAccent,
    elevation: 20.0,
```

```
        child: Row(
            children: <Widget>[
                Image(image: AssetImage('Images/VegetablePZ.png'),
                    width: 100.0,
                    height: 100.0,),

                SizedBox(width: 20.0),


                Text('Vegetable Pizza',
                    style: TextStyle(
                        fontWeight: FontWeight.bold,
                        color: Colors.white,
                        fontSize: 30.0),
                ),
            ],
        ),
    ],
),
),
),
),
),
),
),
),
);
}
}
```

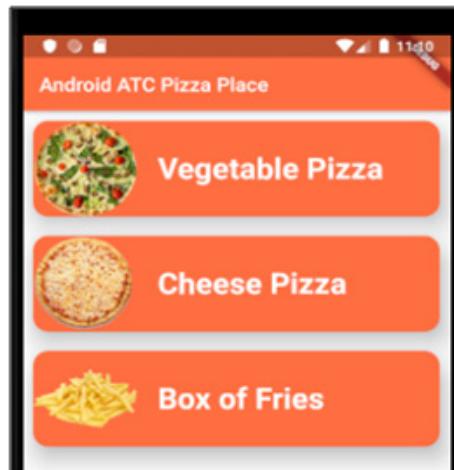
12- Click the Hot Reload button, then you will get the following app interface.



13- In the code of the second Card widget, replace the Image VegetablePZ.png with cheesePZ.png , and the Text value: Vegetable Pizza with Cheese Pizza

14- In the code of the third Card widget, replace the Image **VegetablePZ.png** with **Fries.png** , and the Text value: **Vegetable Pizza** with **Box of Fries**

15 - Click the **Hot Reload** button. You should get the following output run result:



Congratulations !

This is the first app interface you have created. In the next lessons, you will learn how to add amazing features to your app.



These lab source code files are available in the Lab Source Files folder on your computer. If you don't have these lab files, please contact us at [support@androidatc.com](mailto:support@androidatc.com)

# Lesson 6: Navigation and Routing

<b>Button Widget .....</b>	6-2
FloatingActionButton.....	6-2
RaisedButton, FlatButton, and IconButton.....	6-6
DropdownButton .....	6-8
OutlineButton .....	6-13
AppBar.....	6-14
PopupMenuButton.....	6-17
<b>App Structure and Navigation .....</b>	6-21
Navigate to a New Screen and Back.....	6-21
Navigate with Named Routes.....	6-29
Send and Return Data Among Screens .....	6-34
Animate a Widget Across Screens.....	6-36
WebView Widget in Flutter .....	6-40
<b>Lab 6: Navigation and Routing a Pizza Store App .....</b>	6-45

## Button Widget

In Flutter development, you may use the following types of buttons in your app design:

- FloatingActionButton
- RaisedButton
- FlatButton
- IconButton
- ButtonBar
- DropdownButton
- OutlineButton
- PopupMenuItem

### FloatingActionButton Widget

A floating action button is by default a circular icon button that hovers over content to promote a primary action in the application. Use at most a single floating action button per screen. Floating action buttons should be used for positive actions such as “create”, “share”, or “navigate”.

Now, you will create a new empty Flutter app, and then create a simple Flutter app and test the code for which will build each type of the previous buttons.

To create a new Flutter app, perform the following steps:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** , and then click **Next**.

4- Type : **buttons\_06** for Project Name and create a new folder : **Lesson\_06\_Buttons** for Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain, and then click **Finish**

6- Open **main.dart** file ( **lesson\_06 → lib → main.dart** ).

7- Delete all the code contents in **main.dart** file.

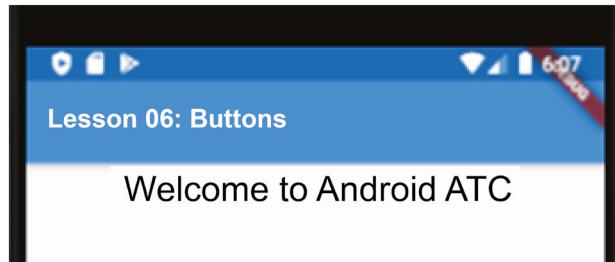
8- Type the flowing code which will create a simple Flutter app. You will use this code later in the next steps to test all types of buttons.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Lesson 06: Buttons'),
        ),
        body: SafeArea(
          child: Column(
            children: <Widget>[
              Center(
                child: SafeArea(
                  child: Text(
                    "Welcome to Android ATC",
                    style: TextStyle(
                      fontWeight: FontWeight.normal,
                      color: Colors.black,
                      fontSize: 25.0),
                  ),
                ),
              ),
            ],
          ),
        );
    );
}
```

9- Run Your app. You should get the following run output:



10- Add the following code to add a **FloatingActionButton** in the Scaffold widget as illustrated in the grey highlighted part of the following code:

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

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

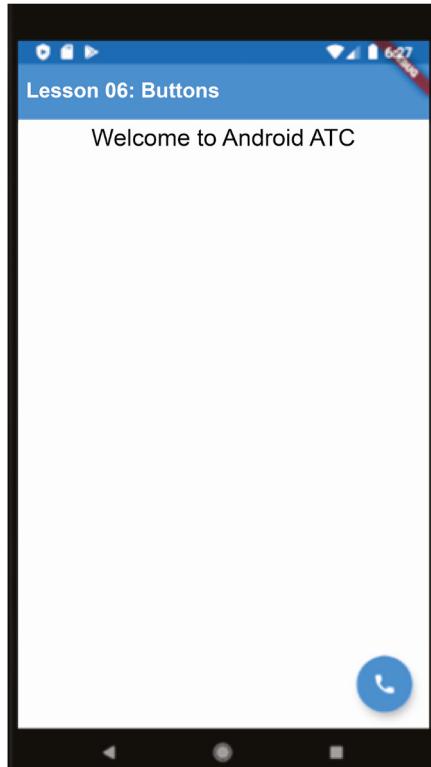
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Lesson 06: Buttons'),
        ),
        body: SafeArea(
          child: Column(
            children: <Widget>[
              Center(
                child: SafeArea(
                  child: Text(
                    "Welcome to Android ATC",
                    style: TextStyle(
                      fontWeight: FontWeight.normal,
                      color: Colors.black,
                      fontSize: 25.0),
                ),
              ),
            ],
          ),
        ),
      ),
    );
}
```

```
        ),  
        ],  
        ),  
        ),  
        floatingActionButton: FloatingActionButton(  
            onPressed: () {},  
            child: Icon(Icons.phone),  
        ),  
        ),  
    );  
}  
}
```

Between the two braces of `onPressed: (){}` function, you can add the action which will be taken if the app user taps this button as you will see in the navigation section in this lesson.

Also, you may replace the phone in `Icon(Icons.phone)` with another icon name such as share , favorite, or other types of icons.

The run output of this app follows:



## RaisedButton, IconButton, and FlatButton Widgets

In the following code, you will add three types of buttons **RaisedButton**, **IconButton**, and **FlatButton**. In general, almost all the buttons have the same purpose which help in navigating the app screens or taking a specific action.

Continue using the same Flutter project buttons\_06, and add these three buttons as illustrated in the following code:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Lesson 06: Buttons'),
        ),
        body: SafeArea(
          child: Column(
            children: <Widget>[
              Center(
                child: SafeArea(
                  child: Text(
                    "Welcome to Android ATC",
                    style: TextStyle(
                      fontWeight: FontWeight.normal,
                      color: Colors.black,
                      fontSize: 25.0),
                  ),
                ),
              ),
            ],
          ),
        ),
      ),
    );
  }
}

// Raised Button
RaisedButton(
  color: Colors.blue,
  child: Text(
    'Raised Button',
    style: TextStyle(
```

```
        fontWeight: FontWeight.normal,
        color: Colors.white,
        fontSize: 20.0),
    ),
    onPressed: () {},
),
),

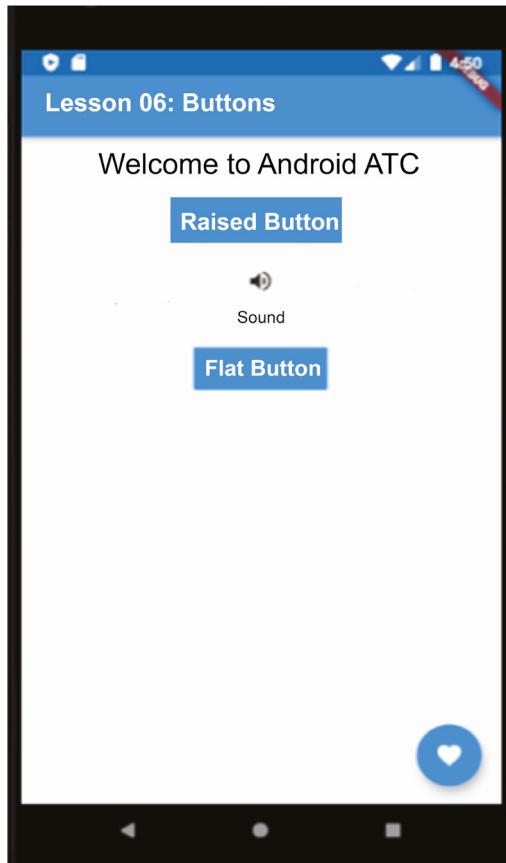
// Icon Button
IconButton(
    icon: Icon(Icons.volume_up),
    tooltip: 'Increase volume by 10',
    onPressed: () {},
),
Text('Sound'),

// Flat Button
FlatButton(
    color: Colors.blue,
    textColor: Colors.white,
    disabledColor: Colors.grey,
    disabledTextColor: Colors.black,
    padding: EdgeInsets.all(8.0),
    splashColor: Colors.blueAccent,
    onPressed: () {},
    child: Text(
        "Flat Button",
        style: TextStyle(fontSize: 20.0),
    ),
),
),

],
),
),
),

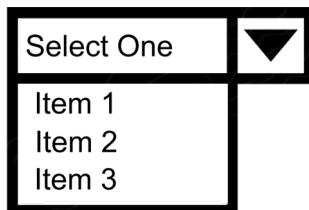
floatingActionButton: FloatingActionButton(
    onPressed: () {},
    child: Icon(Icons.favorite),
),
),
);
}
}
```

The run output follows:



## DropdownButton Widget

A drop down button widget is similar to a drop-down menu that allows your app user to choose only one value from a list. It depends on other Dart methods in its configuration as you will see in the next example.

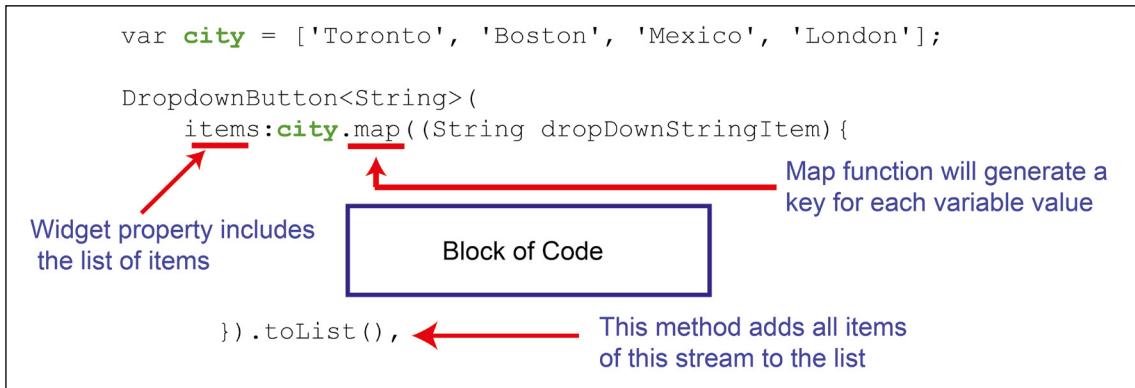


### **Example:**

In this example, to create a drop down button: First, you should define the variable which will have the value of the drop down list. In this example and as illustrated in the following figure, you will use **city** as a list variable. Second, declare the **DropDownButton** widget as a string data type because all items in this drop down

list are string. You will use items widget property to include all the items. Here, you will configure the **city** variable as map, because this variable has many values, and the map function will assign a key for each value. Also, use ".**toList()**" method to add all variable values of this stream to the list in the order they arrive.

In addition "**dropDownStringItem**" will represent one value at each time. The following figure gives an example about how to create a **DropdownButton** :



The block of code includes **DropdownMenuItem** widget which will return a value to the list if the user selects a menu item as illustrated in the following code:

```

var city = ['Toronto', 'Boston', 'Mexico', 'London'];

DropdownButton<String>(
    items: city.map((String dropDownStringItem) {
        return DropdownMenuItem<String>(
            value: dropDownStringItem,
            child: Text(dropDownStringItem),
        );
    }).toList(),
)

```

Now, you will add **setState()** function which will notify your app framework that the selected drop down item (variable value) of this list has been changed. This function works through **onChanged** property. **onChanged** property (Listener) will run **setState()** function every time the value in the drop down list changes.

Then, the code will be as follows:

```
var city = ['Toronto', 'Boston', 'Mexico', 'London'];

DropdownButton<String>(
    items:city.map((String dropDownStringItem){
        return DropdownMenuItem<String>(
            value: dropDownStringItem,
            child: Text(dropDownStringItem),
        );
    }).toList(),
    onChanged: (){
        setState({ Your Code });
    },
);
);
```

Continue using the previous Flutter project and perform the following steps:

1- To work with **setState()** function, change **MyApp** class to **StatefulWidget**. To do that, click the below line of code , click the orange lamp icon, and then select : Change to StatefulWidget.

```
class MyApp extends StatelessWidget {
```

2- After the end of the last button **FlatButton** , add the **DropdownButton** widget code.

The full code follows:

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

main() {
    runApp(
```

```
        MyApp(),
    );
}

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

class _MyAppState extends State<MyApp> {
    var city = ['Toronto', 'Boston', 'Mexico', 'London'];

    var firstcity = 'Toronto';

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(
                    title: Text('Lesson 06: Buttons'),
                ),
                body: SafeArea(
                    child: Column(children: <Widget>[
                        Center(
                            child: SafeArea(
                                child: Text(
                                    "Welcome to Android ATC",
                                    style: TextStyle(
                                        fontWeight: FontWeight.normal,
                                        color: Colors.black,
                                        fontSize: 25.0),
                                ),
                            ),
                        ),
                    ],
                ),
            ),
        );
    }
}

// Raised Button
RaisedButton(
    color: Colors.blue,
    child: Text(
        'Raised Button',
        style: TextStyle(
            fontWeight: FontWeight.normal,
            color: Colors.white,
            fontSize: 20.0),
    ),
)
```

```
        ),  
        onPressed: () {},  
    ),  
  
    // Icon Button  
    IconButton(  
        icon: Icon(Icons.volume_up),  
        tooltip: 'Increase volume by 10',  
        onPressed: () {},  
    ),  
    Text('Sound'),  
  
    // Flat Button  
    FlatButton(  
        color: Colors.blue,  
        textColor: Colors.white,  
        disabledColor: Colors.grey,  
        disabledTextColor: Colors.black,  
        padding: EdgeInsets.all(8.0),  
        splashColor: Colors.blueAccent,  
        onPressed: () {},  
        child: Text(  
            "Flat Button",  
            style: TextStyle(fontSize: 20.0),  
        ),  
    ),  
  
// Start the Code of DropdownButton //  
  
DropdownButton<String>(  
    items: city.map((String dropDownStringItem) {  
        return DropdownMenuItem<String>(  
            value: dropDownStringItem,  
            child: Text(dropDownStringItem),  
        );  
    }).toList(),  
    onChanged: (String NewUserValue) {  
        setState(() {  
            this.firstcity = NewUserValue;  
        });  
    },  
    value: firstcity,  
// Here, firstcity variable value is Toronto.  
// value: firstcity means, the default city value which will  
// appear to the app user.
```

```
        ),  
  
    // End of DropdownButton Code //  
    ]),  
    ),  
  
    floatingActionButton: FloatingActionButton(  
        onPressed: () {},  
        child: Icon(Icons.favorite),  
    ),  
    ),  
);  
}  
}  
}
```

## OutlineButton Widget

**OutlineButton** is similar to a **FlatButton** with a thin grey rounded border.

Perform the following steps to add an outline button to your app interface:

1- Continue using the previous code for Flutter project buttons\_06 , and add the **OutlineButton** as a child widget of the **Column** widget as illustrated in the grey highlighted part of the following code:

```
import 'package:flutter/material.dart';  
  
main() {  
  runApp(  
    MyApp(),  
  );  
}  
  
class MyApp extends StatefulWidget {  
  @override  
  _MyAppState createState() => _MyAppState();  
}  
  
class _MyAppState extends State<MyApp> {  
  var city = ['Toronto', 'Boston', 'Mexico', 'London'];  
  
  var firstcity = 'Toronto';
```

```
@override  
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(  
        title: Text('Lesson 06: Buttons'),  
      ),  
      body: SafeArea(  
        child: Column(children: <Widget>[  
          OutlineButton(  
              
        ),  
      ],  
    ),  
  );  
}
```

2- Add the following code to the **OutlineButton** widget:

```
OutlineButton(  
  shape: StadiumBorder(),  
  highlightedBorderColor: Colors.blue,  
  child: Text('Outline Button'),  
  onPressed: () {},  
,
```

3- Run your app. When you tap this button, you will find a type of change in color in the button background. It is similar to Adobe Flash buttons because it creates a type of interactivity with the app user.

## ButtonBar Widget

This widget is a horizontal arrangement of other types of buttons. For example, this widget can include three or four of *flat* or *outline* buttons and arrange them horizontally to appear as a horizontal navigation bar for your app.

These buttons are children widgets for the **ButtonBar** widget , and it is a good idea to use the **alignment** property with value : **MainAxisAlignment.center** which places other children buttons as close to the middle of the main axis as possible.

The following example displays how to configure the **ButtonBar** widget to include other three **OutlineButton** widgets.

1- Continue using the previous code of project **buttons\_06** , and add the **ButtonBar**

widget as the first widget to the Column widgets as illustrated in the grey highlighted part of the following code:

```
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(  
        title: Text('Lesson 06: Buttons'),  
      ),  
  
      body: SafeArea(  
        child: Column(children: <Widget>[  
          ButtonBar(  
              
          )  
        ]  
      ),  
    ),  
  );  
}
```

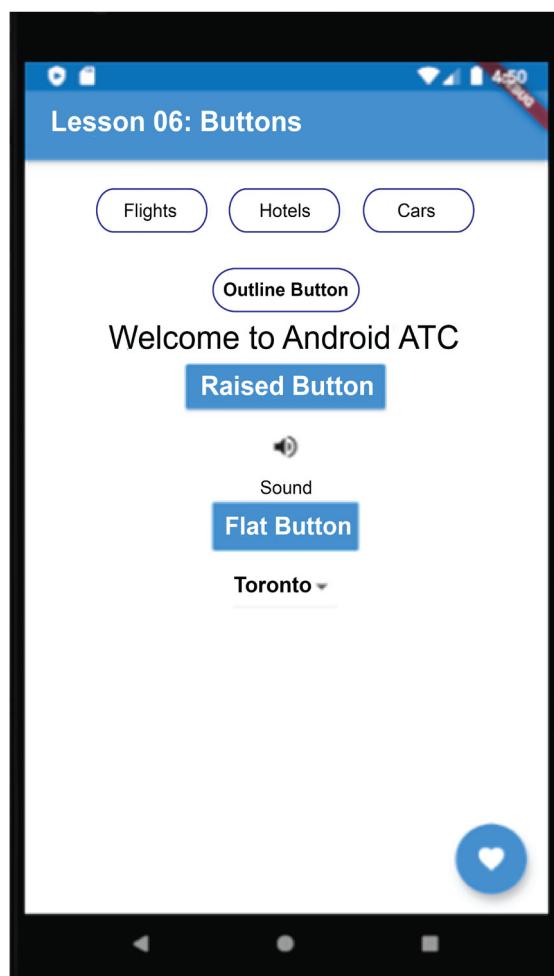
2- Add three **OutlineButton** as children widgets for your **ButtonBar** widgets. Only change the **Text** widget values for these children buttons to have : Flights, Hotels, and Cars . Also, use **alignment: MainAxisAlignment.center** property to place these buttons as close to the middle of the main axis as possible.

The code follows:

```
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(  
        title: Text('Lesson 06: Buttons'),  
      ),  
  
      body: SafeArea(  
        child: Column(children: <Widget>[  
          ButtonBar(  
            alignment: MainAxisAlignment.center,  
            children: [  
  
              OutlineButton(  
                shape: StadiumBorder(),  
                highlightedBorderColor: Colors.blue,  
                child: Text('Flights'),  
                onPressed: () {},  
              )  
            ]  
          ]  
        ),  
      ),  
    ),  
  );  
}
```

```
        ),  
  
        OutlineButton(  
            shape: StadiumBorder(),  
            highlightedBorderColor: Colors.blue,  
            child: Text('Hotels'),  
            onPressed: () {},  
        ),  
        OutlineButton(  
            shape: StadiumBorder(),  
            highlightedBorderColor: Colors.blue,  
            child: Text('Cars'),  
            onPressed: () {},  
        ),  
    ],  
,
```

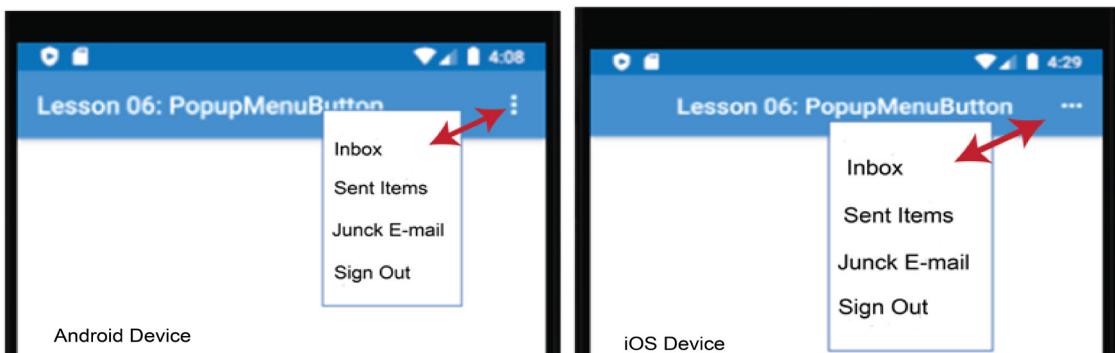
The run output result follows:



## PopupMenuButton Widget

This button can be created at any place in the app. When the app user taps on this button, it displays a menu with many items, and calls `onSelected` property which includes an action for each menu item as you will see in the next example.

The following figure displays how the `PopupMenuButton` widget appears at the app bar for Android and iOS phones:



**PopupMenuButton** uses the `PopupMenuItems` class in its configuration, which is responsible to create the items for the popup menu.

### **Example:**

Here, you will create a new Flutter app to focus more on configuring the `PopupMenuButton` widget and its properties. To create a new Flutter app and configure the `PopupMenuButton` widget, perform the following steps:

- 1- Open **Android Studio**
- 2- Click **File** → **New** → **New Flutter Project**
- 3- Select **Flutter Application** , and then click **Next**.
- 4- Type : `popup_menu_button_06` for Project Name , and create a new folder : **PopupMenuButton** for Project Location. Click **Next**.
- 5- Type : `androidatc.com` for Company domain, and then click **Finish**
- 6- Open **main.dart** file (`popup_menu_button_06 → lib → main.dart`).
- 7- Delete all the code in **main.dart**
- 8- Type the following code in **main.dart**:

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

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

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

class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Lesson 06: PopupMenuItem'),
          actions: <Widget>[
            PopupMenuItem(
              onSelected: choiceAction,
              itemBuilder: (BuildContext context) {
                return item.choices.map((choice) {
                  return PopupMenuItem(
                    value: choice,
                    child: Text(choice),
                  );
                }).toList();
              },
            ),
          ],
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Text('Welcome to Android ATC',),
            ],
        ),
      ),
    );
  }
}
```

```
        ),
        ),
    );
}

void choiceAction(choice) {
    if (choice == item.Inbox) {
        print('Inbox');

    } else if (choice == item.SentItems) {
        print('Sent Items');

    } else if (choice == item.JunckEmail) {
        print('Junck E-mail');

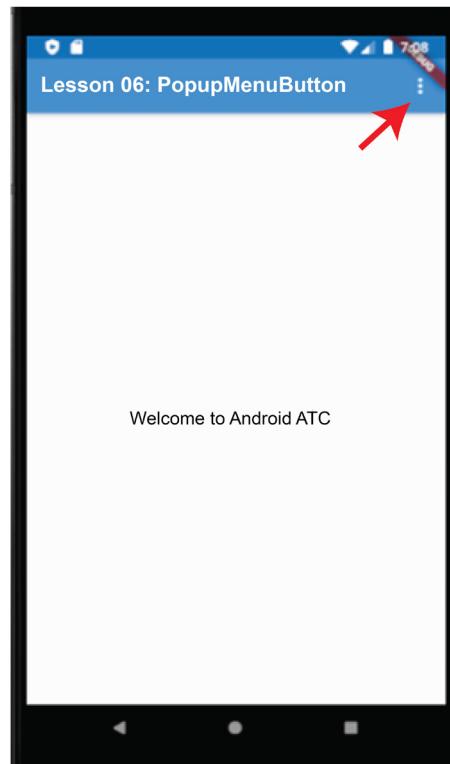
    } else if (choice == item.SignOut) {
        print('Sign Out');

    }
}

class item {
    static const Inbox = 'Inbox';
    static const SentItems = 'Sent Items';
    static const JunckEmail = 'Junck E-mail';
    static const SignOut = 'Sign Out';

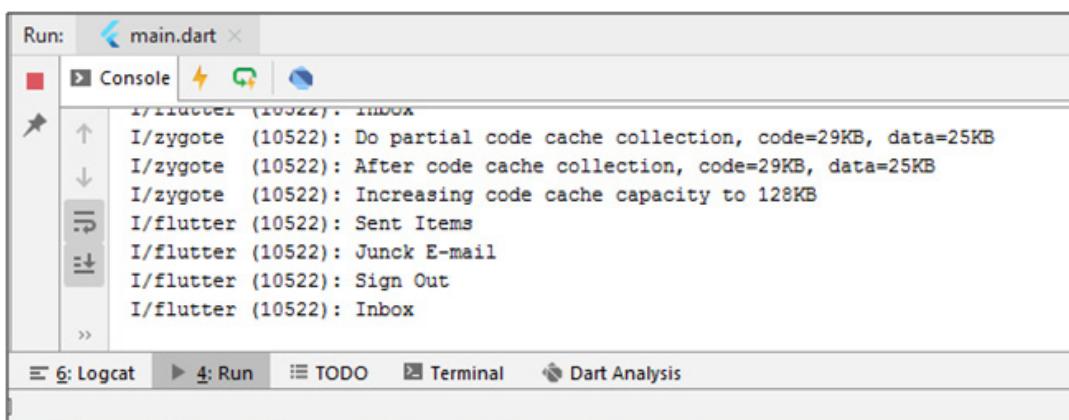
    static const List choices = [Inbox, SentItems, JunckEmail, SignOut];
}
```

9- Run your app and you should get the following run output:



10- Click the Popup Menu Button which exits on your app title bar.

11- The grey highlighted parts of the previous code display the action which will be done when you select any of the popup menu items. In this example , when you select **Inbox** , the app will print **Inbox** on the run console. The same thing applies for other items as illustrated in the following figure:



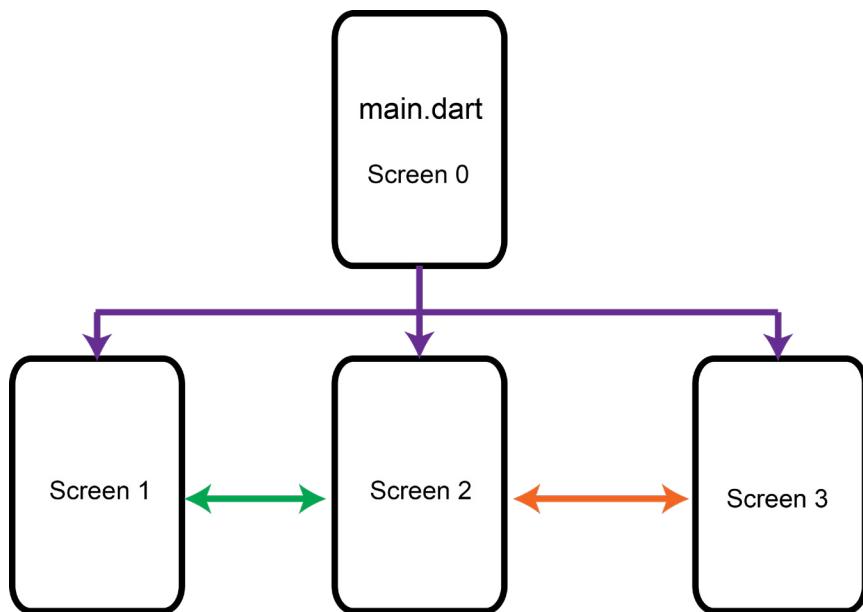
## App Structure and Navigation

The app structure is almost similar to the web site structure. The first app interface which will open when users run your app is called the main interface, while it is called a home page for the web site. In Flutter development your main interface is the **main.dart** file. The main app interface or screen must include links or routes to your other app screens or your app parts.

In your app, you will use various navigation techniques to move from one screen to another. For any app to be successful, it should be easy to use and have a variety of navigation techniques. This would help app users to reach and use app data in a simple and easy manner, using friendly app interfaces

Therefore, try to make your app user friendly, and design it to meet the needs of app users and businesses in more than one way.

In this lesson, you will learn almost all these navigation methods using different types of navigation techniques.



### Navigate to a New Screen and Back

Mobile apps typically reveal their contents via full-screen elements called “**screens**”

or “**pages**”. In Flutter, these elements are called **routes** and they are managed by a *Navigator* class or widget. Flutter navigator takes you on each of these routes to see each of these screens.

The **Navigator** class manages a stack of Route (screen) objects and provides methods for managing the stack, like **Navigator.push** and **Navigator.pop**.

**Navigator.push()** method is used to navigate to the second route. Whereas, **Navigator.pop()** method is used to return to the first route or screen as you will see in the following example:

**Example:**

In this example, you will create a Flutter app including **main.dart** file and other 3 Dart files (screen0.dart, screen1.dart, and screen2.dart).

You will use the navigator class to move from Screen 0 (main.dart) to Screen 1 (screen1.dart), and return from Screen 1 (screen1.dart) to Screen 0 (main.dart). Perform the following steps:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

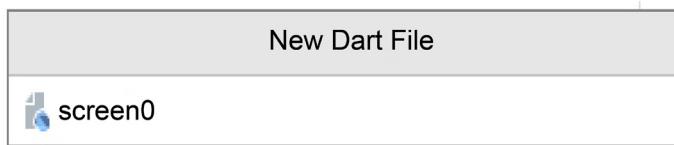
3- Select **Flutter Application**, and then click **Next**.

4- Type : **lesson\_06** for Project Name , and create a new folder :**D:\Lesson\_06** for Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain, and then click **Finish**

6- Now, create **screen0.dart** (Screen 0 interface). Right click **lib** folder, select **New → Dart File**

7- Type the name of the file : **screen0** , and then press **Enter** as illustrated in the following figure:



8- Repeat the previous task to create the **screen1.dart**, and **screen2.dart** files.

9- Open **screen1.dart** , and type the following code:

```
import 'package:flutter/material.dart';
class Screen1 extends StatelessWidget {
```

```

@Override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      backgroundColor: Colors.blue,
      title: Text('Screen 1'),
    ),
    body: Center(
    ),
  );
}
}

```

10- Now, you will add a button on interface 0 (screen0.dart) , and when you tap this button you will go to Screen 1 (screen1.dart).

The following figure displays where you must add the navigator class in the button:

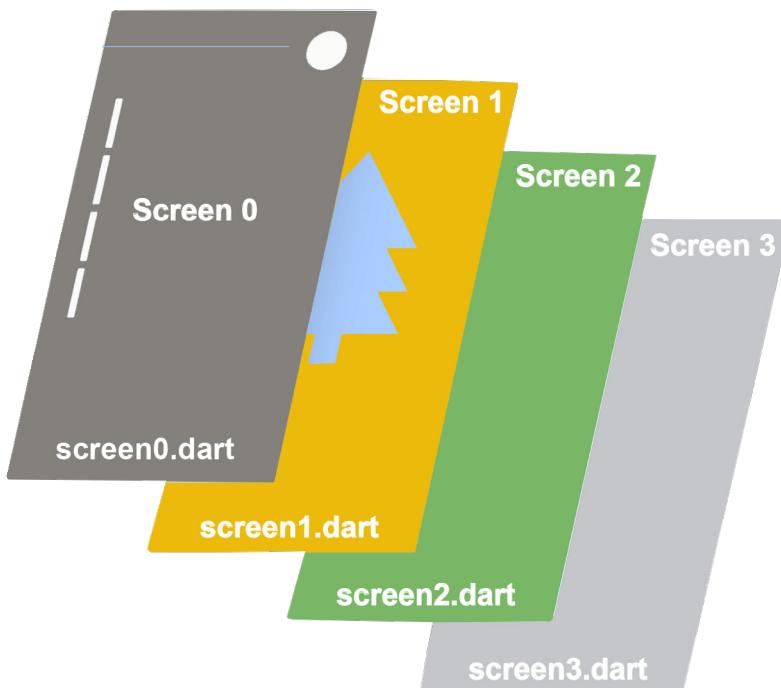


You will add **Navigator.push** method inside the **onPressed** method body. This **push()** method adds a *Route* to the stack of routes (interfaces) managed by the Navigator. The following figure displays the **Navigator.push** method parameters :



The question is, what does `Navigator.push` method do ?

When the user taps the image or button, a new screen (route) displays details about this item. Your app consists of many screens (routes) which are arranged as illustrated in the following figure:



When you run the `Navigator.push()` method on screen 0 (main.dart), you push out this screen and then another screen appears on the smart device depending on this method configuration.

Open `screen0.dart`, and type the following code:

```
import 'package:flutter/material.dart';
import 'screen1.dart';

class Screen0 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.blue,
        title: Text('Screen 0'),
      ),
      body: Center(
        child: Text('Hello from Screen 0'),
      ),
    );
  }
}
```

```
        child: Column(  
            children: <Widget>[  
  
                RaisedButton(  
                    color: Colors.blue,  
                    child: Text('Go To Screen 1'),  
  
                    onPressed: () {  
                        Navigator.push(  
                            context,  
                            MaterialPageRoute(builder: (context) {  
                                return Screen1();  
                            }),  
                        );  
                    },  
                ),  
            ],  
        );  
    }  
}
```

The grey highlighted part of the previous code is related to the *Raised Button*. The **navigator class** in the **OnPressed()** method is responsible to configure the route which you will use to move from **Screen 0** to **Screen 1** when you tap this button.

You must import the **screen1.dart** as you did on the second line of the above code to define **Screen1()** in **screen0.dart**.

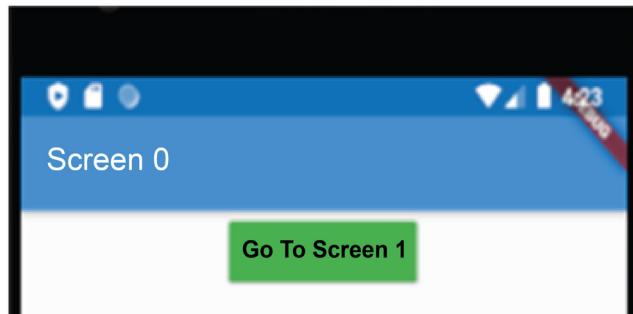
11- Open **main.dart** file, and **delete** all its content. Then type the following code:

```
import 'package:flutter/material.dart';  
import 'screen0.dart';  
  
main() {  
    runApp(  
        MyApp(),  
    );  
}  
  
class MyApp extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {
```

```
    return MaterialApp(  
      home: Screen0(),  
    );  
}  
}
```

Now, when you run your app, the **main.dart** file will run **screen0.dart** file due to the grey highlighted part of **main.dart** file.

12- Run your app. You will get the following figure. Tap this button, and then you will go to Screen 1.



13- Open **screen1.dart**, copy all the code, open **screen2.dart**, and then paste the code.

14- Just change the class name to **Screen 2**. Then change the Scaffold title to **Screen 2**. You should have the following code in **screen2.dart** :

```
import 'package:flutter/material.dart';  
  
class Screen2 extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        backgroundColor: Colors.blue,  
        title: Text('Screen 2'),  
      ),  
      body: Center(),  
    );  
  }  
}
```

15- Open **Screen 1** (screen1.dart), and add a new button that has a route to **Screen 2** as illustrated in the following code:

```
import 'package:flutter/material.dart';
import 'screen2.dart';

class Screen1 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.blue,
        title: Text('Screen 1'),
      ),
      body: Center(
        child: Column(
          children: <Widget>[
            RaisedButton(
              color: Colors.green,
              child: Text('Go To Screen 2'),
              onPressed: () {
                Navigator.push(
                  context,
                  MaterialPageRoute(builder: (context) {
                    return Screen2();
                  }),
                );
              },
            ),
          ],
        )));
  }
}
```

16- Run your app. Tap the button “**Go To Screen 1**”, and then tap the button “**Go To Screen 2**” you will go to **Screen 2**.

Now, use **Navigator.pop()** method to return to the original route. The **pop()** method removes the current Route from the stack of routes managed by the Navigator.

To implement a return to the original route, update the **onPressed()** callback in the Second Route widget using the following code:

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

17- Open **Screen 2** (screen2.dart) and add a button that has a return to original route using **Navigator.pop()** method. When you tap this button, you return to **Screen 1** (screen1.dart).

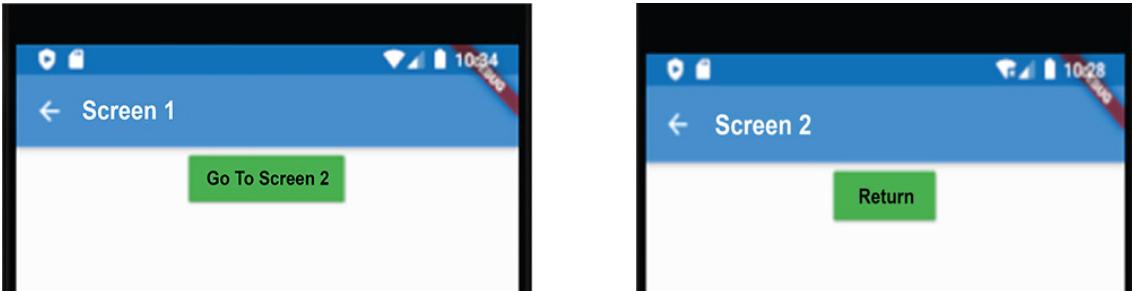
The code follows:

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

class Screen2 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.blue,
        title: Text('Screen 2'),
      ),
      body: Center(
        child: Column(
          children: <Widget>[
            RaisedButton(
              color: Colors.green,
              child: Text('Return'),
              onPressed: () {
                Navigator.pop(context);
              },
            ),
          ],
        ),
      ),
    );
}
```

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

Run your app again. Tap the button “**Go To Screen 1**”, and then tap the button “**Go To Screen 2**”. On **Screen 2**, when you tap **Return** button , you will return to **Screen 1** as illustrated in the following figures:



## Navigate with Named Routes

In most apps, usually, you have many routes on each screen, especially the first screen. You can easily create a navigation for each route using named **navigation route** method. In this method, you will name each route by a name , and then call this route using this name when you want to navigate to it.

You will use Dart map technique to assign a name (key) to each route. The map object has already been explained with examples in lesson 2 of this course.

Remember that DART map is an object that associates keys to values. In DART, map is an interface designed to manipulate a collection of keys which point to values (routes).

To work with named routes, use the **Navigator.pushNamed()** function as illustrated in the following example:

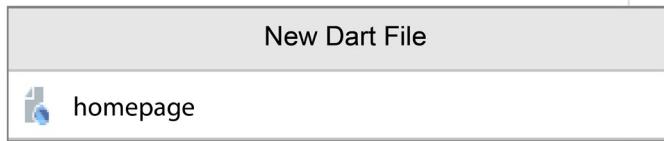
Continue using the previous project (lesson\_06) which includes four Dart files, and perform the following steps:

1- Create a Dart file (**homepage.dart**) including all your app routes.

Right click the **lib** folder, and then select **New → Dart File**

2- Type **homepage** for the file name, and then press **Enter** as illustrated in the

following figure.



3- Open **homepage.dart** file and type the following code to create three navigation buttons:

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

class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.blue,
        title: Text('Home Page'),
      ),
      body: Center(
        child: Column(
          children: <Widget>[
            RaisedButton(
              color: Colors.green,
              child: Text('Go To Screen 0'),
              onPressed: () {}),
            RaisedButton(
              color: Colors.green,
              child: Text('Go To Screen 1'),
              onPressed: () {}),
            RaisedButton(
              color: Colors.green,
              child: Text('Go To Screen 2'),
              onPressed: () {}),
          ],
        ),
      );
    }
}
```

4- If you want to make the **hompeage.dart** as the startup page when you run your app. you would need to open **main.dart** file and then add the following command at the beginning of this file:

```
import 'hompeage.dart';
```

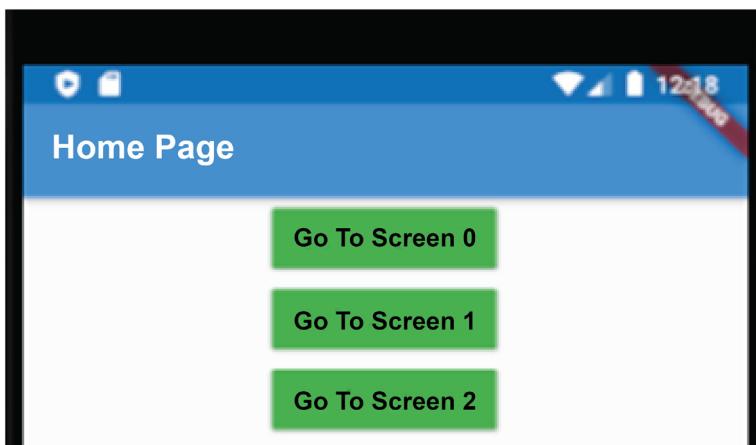
Now, replace the **home** value with **HomePage()** as illustrated in the grey highlighted color in the **main.dart** file as follows:

```
import 'package:flutter/material.dart';
import 'hompeage.dart';
import 'screen0.dart';

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

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

5- Run your app. You should get the following run output:



6- Now, you should add a navigation map including keys to all routes. Open **main.dart** file, and then add your app map of routes as illustrated in the following code:

**Note:** you should import **screen1.dart** and **screen2.dart** in **main.dart** as illustrated at the top of the following code:

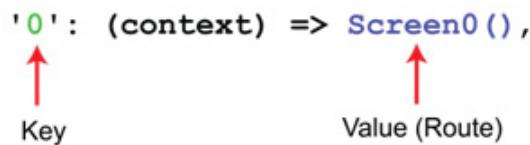
```
import 'package:flutter/material.dart';
import 'homepage.dart';
import 'screen0.dart';
import 'screen1.dart';
import 'screen2.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: HomePage(),

      routes: {
        '0': (context) => Screen0(),
        '1': (context) => Screen1(),
        '2': (context) => Screen2(),
      },
    );
}
}
```

The following figure displays the map structure where each key represents a navigation route.

'0': (context) => Screen0(),  


7- Open **homepage.dart** file, and add the **Navigator.pushNamed()** function to each button with its route key which you configured in the **main.dart** file. The full code of the **homepage.dart** file follows:

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

class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.blue,
        title: Text('Home Page'),
      ),
      body: Center(
        child: Column(
          children: <Widget>[
            RaisedButton(
              color: Colors.green,
              child: Text('Go To Screen 0'),
              onPressed: () {
                Navigator.pushNamed(context, '0');
              },
            ),
            RaisedButton(
              color: Colors.green,
              child: Text('Go To Screen 1'),
              onPressed: () {
                Navigator.pushNamed(context, '1');
              },
            ),
            RaisedButton(
              color: Colors.green,
              child: Text('Go To Screen 2'),
              onPressed: () {
                Navigator.pushNamed(context, '2');
              },
            ),
          ],
        ),
      );
    }
}
```

8- Now, Run your app, and test the three buttons on **homepage.dart**. You will find that you can move to all routes easily.

**Note:** Use the back arrow button to return to the previous interface.

## Send and Return Data Among Screens

Often, you do not only need to navigate to a new screen, but you also need to pass data in between screens as well.

Remember: Screens are just widgets.

### Example:

Continue using the same Flutter project lesson\_06 as used previously.

1- Open **screen2.dart** file.

2- Declare and add a string variable **FirstName** to **Screen2** class. Also, change your **appBar** title value to use this **FirstName** variable as illustrated in the following gray highlighted code:

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

class Screen2 extends StatelessWidget {

    String FirstName = "";
    Screen2({Key key, this.FirstName}) : super(key: key);

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                backgroundColor: Colors.blue,
                title: Text(FirstName),
            ),
            body: Center(
                child: Column(
                    children: <Widget>[
                        RaisedButton(
                            color: Colors.green,
                            child: Text('Return'),
                        ),
                    ],
                ),
            ),
        );
    }
}
```

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

3- Open **screen1.dart** , and configure the value of **FirstName** variable with the return function in the raised button as illustrated in the following figure:

```
    RaisedButton(
        color: Colors.green,
        child: Text('Go To Screen 2'),
        onPressed: () {
            Navigator.push(
                context,
                MaterialPageRoute(builder: (context) {
                    return Screen2(FirstName: "Your First Name",);
                }), // MaterialPageRoute
            );
        },
    ), // RaisedButton
```

Replace “Your First Name” with your real first name in the above code , then save and run your app.

4- Tap “Go To Screen 1” button, and then tap “Go to Screen 2” button. You will get the following figure:



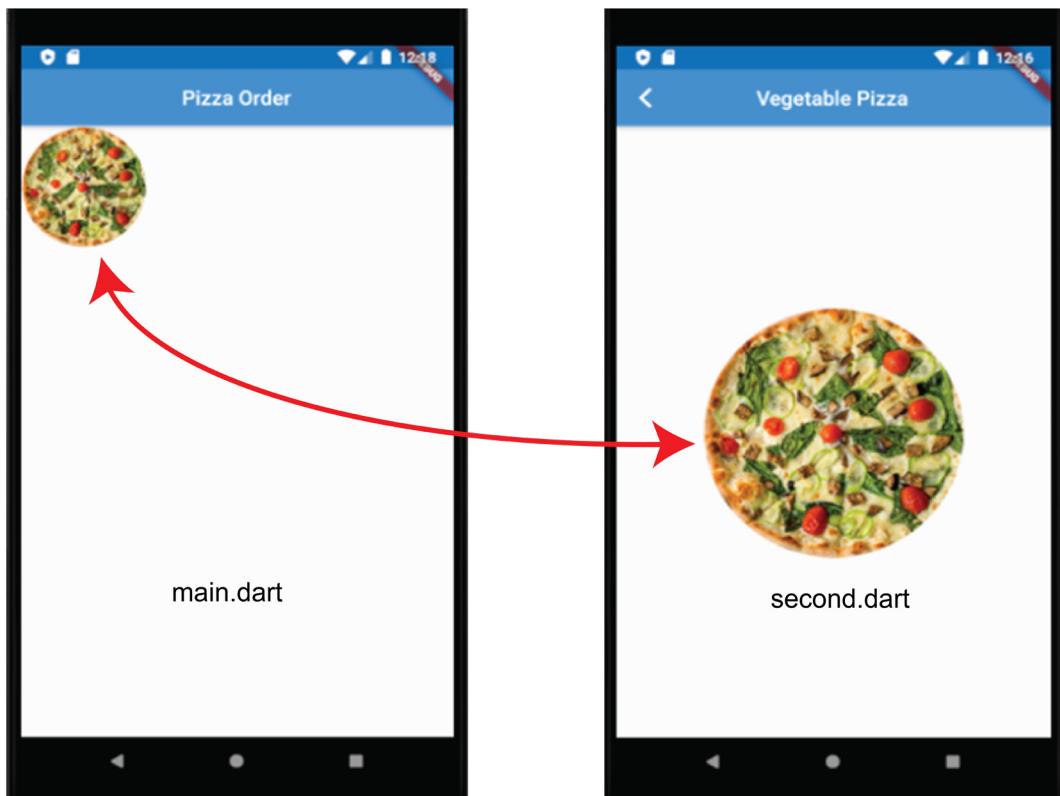
## Animate a Widget Across Screens

It's always helpful to guide users through an app as they navigate from screen to screen. A common technique to lead users through an app is to animate a widget from one screen to the next screen (app interface). This creates a visual anchor connecting the two screens.

Use the **Hero** widget to animate a widget from one screen to the next. This recipe uses the following steps:

- a. Create two screens showing the same image.
- b. Add a **Hero** widget to the first screen.
- c. Add a **Hero** widget to the second screen.

The following example, as illustrated in the following figure, displays how to navigate from one app screen (**main.dart**) to another(**second.dart**) using **Hero** widget:

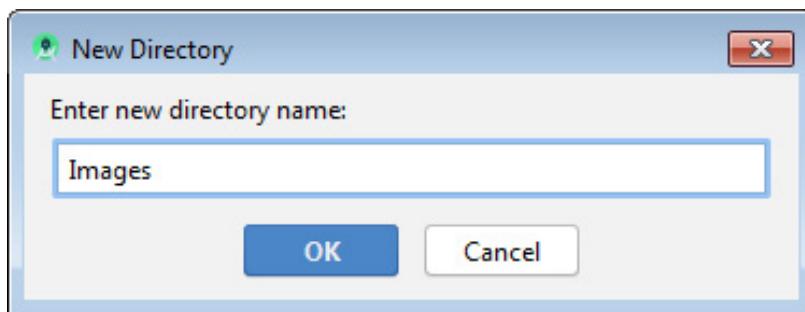


- 1- Open Android Studio
- 2- Click File → New → New Flutter Project

- 3- Select **Flutter Application** , and then click **Next**.
- 4- Type : **animate\_across\_screens\_06** for Project Name , and create a new folder : **Hero\_Widget** for Project Location. Click **Next**.
- 5- Type : **androidatc.com** for Company domain, and then click **Finish**
- 6- Open **main.dart** file (popup\_menu\_button\_06 → lib → main.dart).
- 7- Right click **lib** folder , then select **New → Dart File**  
Type : **second** for the file name , and then press **Enter**.

Here, **second.dart** file will be the destination file. This means when the app user taps the image in the **main.dart** file (first interface) , the app will open **second.dart** file (next interface) which includes the same image with a bigger size. This action produces an animation effect done while navigating between screens.

- 8- Right click the Flutter project name: **animate\_across\_screens\_06** , and then select **New → Directory**. Type **Images** for the directory name , and then click **OK** as illustrated in the following figure:



This folder will include all your app images.

- 9- As you did before in lesson 5 in the adding asset image topic, configure the default location of the asset images folder in the **pubspec.yaml** file.

Remember, the part which is related to add an asset image in **pubspec.yaml** file must be configured as the follows:

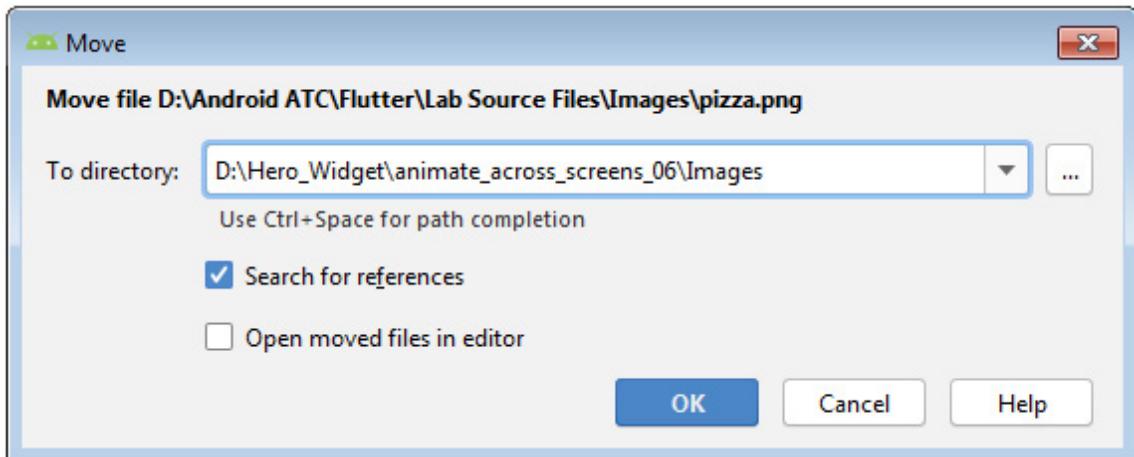
```

43      # To add assets to your application, add
44      assets:
45        - Images/
46        #   - images/a_dot_ham.jpeg
47

```

And then click **Packages get** at the top of **pubspec.yaml** file.

10- Open the “**Lab Source Files → images**” folder on your computer , and then copy the **pizza.png** image and paste it in the **Images** folder in your Flutter app. You will get the following figure. Click **OK**.



11- Open **second.dart** and add the code below. Here, the **GestureDetector** class defers the image for its sizing behavior, and the **Hero** widget will implement a style of animation between **second.dart** and **main.dart** .This Hero widget uses tag: 'imageHero' as a widget property to perform an image animation between the app screens.

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

class Second extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Vegetable Pizza'),
      ),
      body: GestureDetector(
        child: Center(
          child: Hero(
            tag: 'imageHero',
            child: Image(
              image: AssetImage('Images/pizza.png'),
              width: 400.0,
              height: 400.0,
            ),
          ),
        ),
      ),
    );
}
```

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

12- Delete all the code in **main.dart**

13- Type the following code for **main.dart**:

```
import 'package:flutter/material.dart';  
import 'second.dart';  
  
main() {  
    runApp(  
        MyApp(),  
    );  
}  
  
class MyApp extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            title: 'Transition Demo',  
            home: MainScreen(),  
        );  
    }  
}  
  
class MainScreen extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            appBar: AppBar(  
                title: Text('Pizza Order'),  
            ),  
            body: GestureDetector(  
                child: Hero(  
                    tag: 'imageHero',  
                    child: Image(  
                ),  
            ),  
        ),  
    );  
}
```

```
        image: AssetImage('Images/pizza.png'),
        width: 120.0,
        height: 120.0,
    ),
),
),
onTap: () {
    Navigator.push(context, MaterialPageRoute(builder: (_) {
        return Second();
    } ),);
},
),
);
}
}
```

14- **Run** your app, and then click the pizza image on your app interface. You will move to the **second.dart**. Flying the pizza image from the first screen to another is called a ***hero animation*** in Flutter.

*Remember, the **Hero** widget animates from the source to the destination route.*

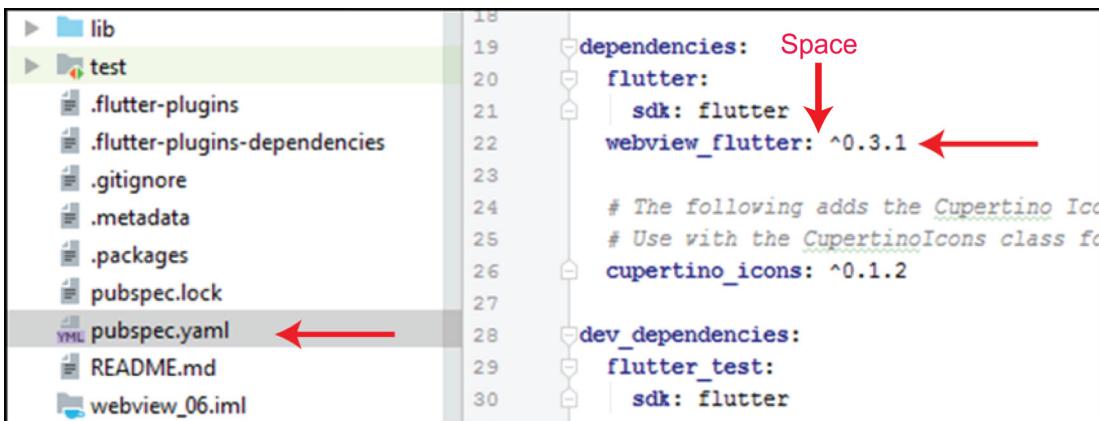
## WebView in Flutter

You can embed a web browser inside your app using **WebView** widget. A **WebView** widget is a crucial component for many apps that need to display website content without worrying about using Android or iOS native views. Incorporating the **WebView** plug-in into your app is extremely simple. **WebView** widget is just like any other widget in Flutter. For example, you can add the following **WebView** widget to display Android ATC web site content in your app interface. By default, JavaScript in your **WebView** widget is disabled, so to enable it, you should add the **javascriptMode** property with **unrestricted** value to your **WebView** widget as illustrated in the code below:

```
WebView(
initialUrl: 'https://www.androidatc.com',
javascriptMode: JavascriptMode.unrestricted,
```

**Example:**

- 1- Open **Android Studio**
- 2- Click **File → New → New Flutter Project**
- 3- Select **Flutter Application** , and then click **Next**.
- 4- Type : **webview\_06** for Project Name , and create a new folder : **Web\_View\_06** for Project Location. Click **Next**.
- 5- Type : **androidatc.com** for Company domain, and then click **Finish**
- 6- Open **main.dart** (**webview\_06 → lib → main.dart**).
- 7- Add the **webview\_flutter** plug-in to your app **pubspec.yaml** file. The following figure displays the **pubspec.yaml** file location and the **webview\_flutter** plug-in configuration.



```

lib
test
  .flutter-plugins
  .flutter-plugins-dependencies
  .gitignore
  .metadata
  .packages
  pubspec.lock
  pubspec.yaml ←
  README.md
  webview_06.iml

18
19   dependencies: Space
20     flutter:
21       sdk: flutter
22     webview_flutter: ^0.3.1 ←
23
24   # The following adds the Cupertino Icons to
25   # Use with the CupertinoIcons class for
26   cupertino_icons: ^0.1.2
27
28   dev_dependencies:
29     flutter_test:
30       sdk: flutter

```

- 8- To use **WebView** on iOS, you are required to add a special setting in your Flutter project. Every iOS project has a special file called **Info.plist**, which is a simple list of configuration settings. You can edit this file by opening: **webview\_06 → iso → Runner → Info.plist** , and add the following to the **Info.plist** file **<dict>** element:

```

<key>io.flutter.embedded_views_preview</key>
<string>YES</string>

```

Add these two lines to **Info.plist** file as illustrated in the following figure:

```

34      </array>
35      <key>UISupportedInterfaceOrientations~ipad</key>
36      <array>
37          <string>UIInterfaceOrientationPortrait</string>
38          <string>UIInterfaceOrientationPortraitUpsideDown</string>
39          <string>UIInterfaceOrientationLandscapeLeft</string>
40          <string>UIInterfaceOrientationLandscapeRight</string>
41      </array>
42      <key>UIViewControllerBasedStatusBarAppearance</key>
43      →<key>io.flutter.embedded_views_preview</key>
44      →<string>YES</string>
45          <false/>
46      </dict>
47  </plist>

```

9- Delete all the lines of code in **main.dart**.

10- Type the following code:

```

import 'package:flutter/material.dart';

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

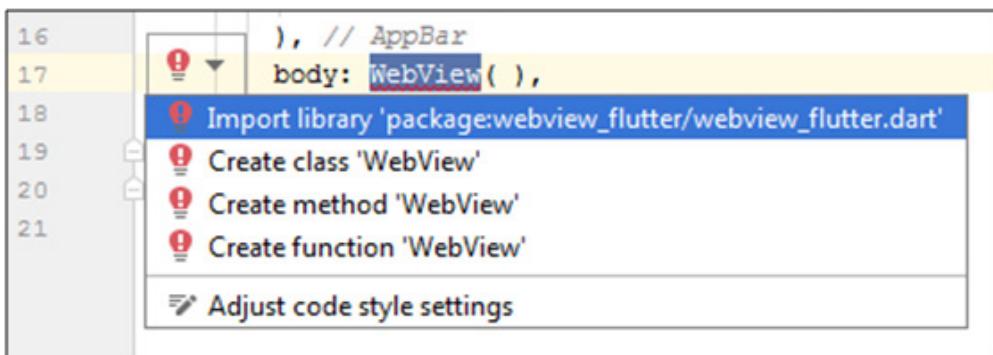
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Lesson 06: Web View'),
        ),
        body: WebView( ),
      );
    }
}

```

11- Double click the **WebView** widget, click the red lamp icon, and then select:

**Import library ' package:webview\_flutter/webview\_flutter.dart'** as illustrated in the

following figure:



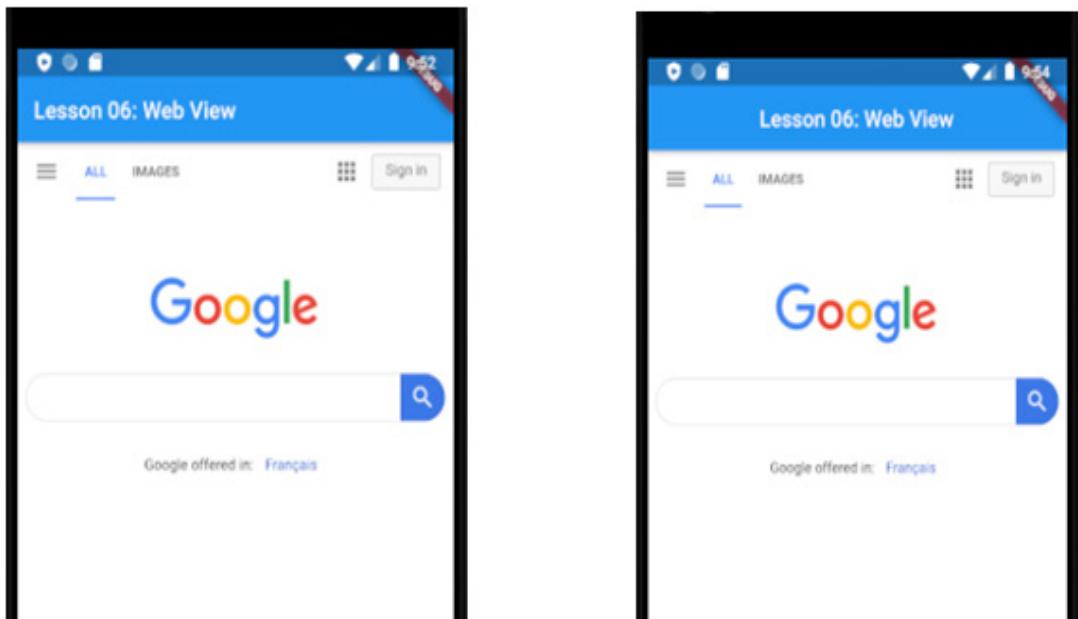
12- Now, add the URL for the destination web site. For example <https://www.google.com>. The full code will be as follows:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Lesson 06: Web View'),
        ),
        body: WebView(initialUrl: 'https://www.google.com'),
      ),
    );
  }
}
```

13- Run your app. The run output for Android and iOS follows:



14- To enable the JavaScript mode, add : **javascriptMode** property with **unrestricted** value to the **WebView** widget as follows:

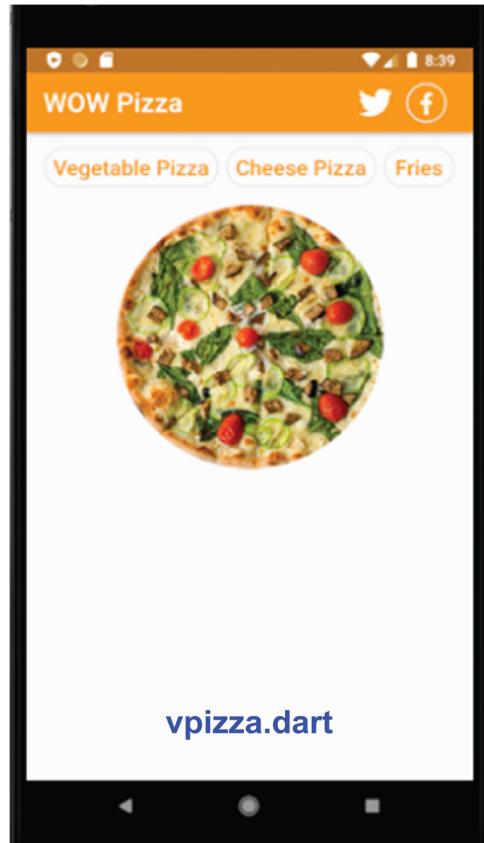
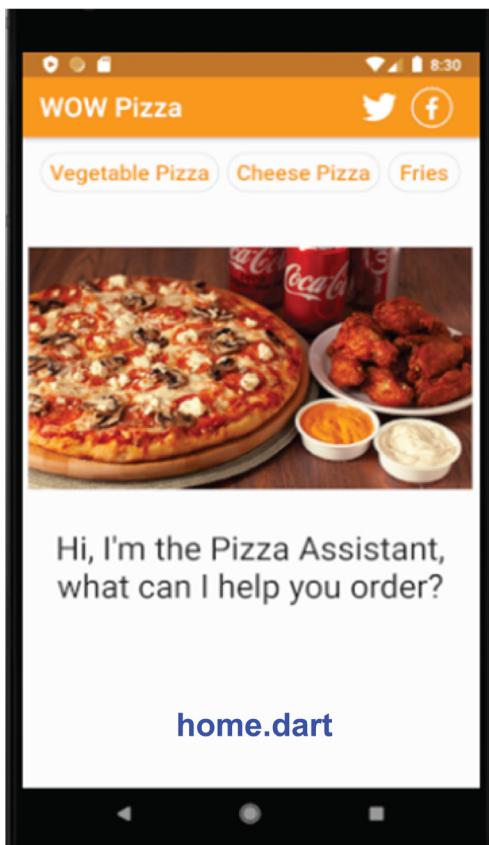
```
class MyApp extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(  
                    title: Text('Lesson 06: Web View'),  
                ),  
                body: WebView(  
                    initialUrl: 'https://www.google.com',  
                    javascriptMode: JavascriptMode.unrestricted),  
            ),  
        );  
    }  
}
```

## Lab 6: Navigation and Routing a Pizza Store App

In this lab, you will create a Flutter app for a pizza store called WOW Pizza. You will use the “navigate with named routes” technique to move from the home interface (home.dart) to other interfaces using the **ButtonBar** widget which includes three buttons and two social media logos.

The twitter image will open a twitter web site on a new interface within the app using the **WebView** widget. Also, the Facebook image will open a new interface including the Facebook web site.

Before starting, just check the following app interfaces which represent the app interfaces you will create in this lab.

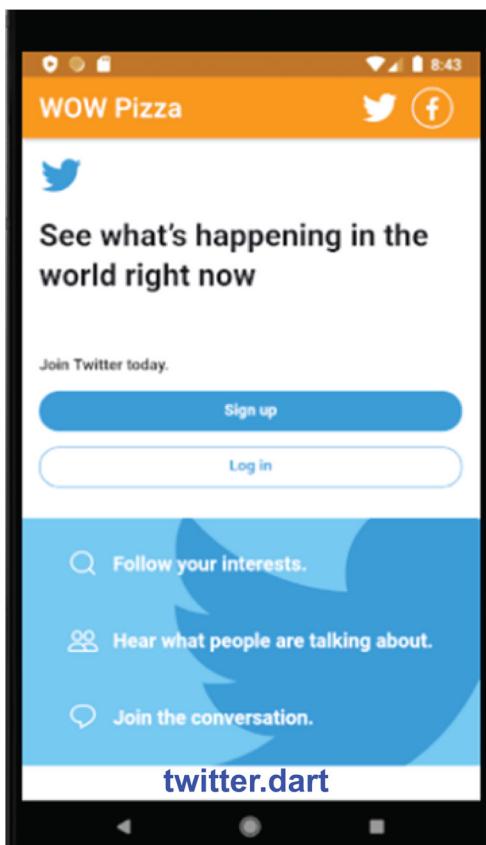




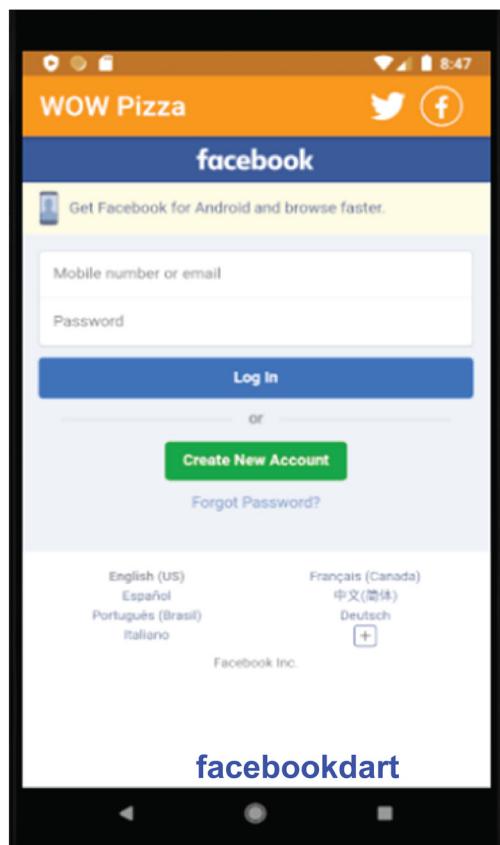
chpizza.dart



fries.dart



twitter.dart



facebookdart

As you see, each interface has the same title bar, Facebook logo, twitter logo, and other three navigation buttons. Therefore, you will design one interface including all these common widgets in a file called **home.dart** first. Then, use the copy and paste technique to repeat the same code for all other dart files (other app interfaces). The **ButtonBar** widget which represents the navigation buttons, will be repeated in **home.dart**, **vpizaa.dart**, **chpizza.dart**, and **fries.dart** files. In the early stage of building the code, the **facebook.dart** and **twitter.dart** files will include the same codes as other Dart files (app interfaces) excluding the three navigation buttons (**ButtonBar** widget).

To create this app perform the following steps:

1- Open **Android Studio**

2- Click **File** → **New** → **New Flutter Project**

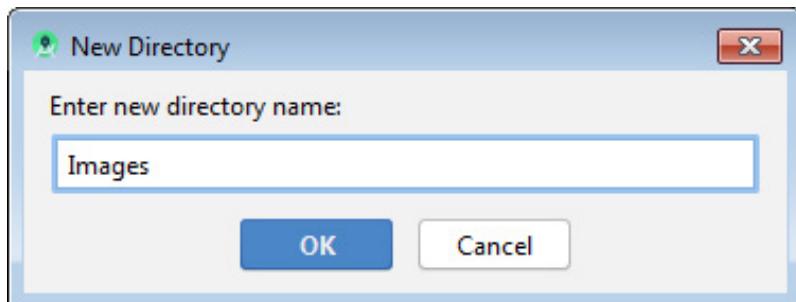
3- Select **Flutter Application** , and then click **Next**.

4- Type : **lab\_6** for Project Name , and create a new folder : **Lab\_06** for Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain, and then click **Finish**

6- Create the **Images** folder which will include all your app images.

Right click the root project name (lab\_6) → **New** → **Directory** . Type **Images** for the directory name, and then click **OK**.



7- All your app images are available in the images folder (**Images\Lab 6**) in “**Lab Source Files**”. Open this folder and then copy the images below:  
**cheesepizza.png**, **Vpizza.png**, **Fpizza.png**, **meal.jpg**, **twitter.png** , and **facebook.png**

Then, paste them in the **Images** folder in Android Studio.

8- You will use the **WebView** widget plug-in to open the Facebook and Twitter web sites in your app; therefore, you must configure the **pubspec.yaml** and **info.plist** files to enable using the **WebView** widget in Android and iOS devices. Also, configure the **Images** folder as the default location for all images.

a) Open the **pubspec.yaml** and add the **webview\_flutter: ^0.3.1** as illustrated in the following figure:

```
19   dependencies:  
20     flutter:  
21       sdk: flutter  
22       webview_flutter: ^0.3.1  
23
```

Scroll down the **pubspec.yaml** and remove the comment sign for the **assets** and add **Images** folder as illustrated in the following figure:

```
43   # To add assets to your application, add an assets section, like this:  
44   assets:  
45     - Images/  
46       #   - images/a_dot_ham.jpeg  
47
```

And then, click : **Packages get**

b) Open **info.plist** file (**lab\_6 → iso → Runner → Info.plist**), and then add the following code to the **Info.plist** file **<dict>** element:

```
<key>io.flutter.embedded_views_preview</key>  
<string>YES</string>
```

Add these two lines to **Info.plist** file as illustrated in the following figure:

```
41   </array>  
42   <key>UIViewControllerBasedStatusBarAppearance</key>  
43   <key>io.flutter.embedded_views_preview</key>  
44   <string>YES</string>  
45   <false/>  
46   </dict>  
47   </plist>
```

9- Now add all your app Dart files as follows:

Right click **lib** directory, and then select **New → Dart File**. Type **home**, and then press **Enter**.

Repeat this step to add **vpizaa.dart** , **chpizza.dart** , **fries.dart**, **facebook.dart** and **twitter.dart**.

10- The **home.dart** file will be the startup file for your app.

Open **home.dart** , and type the following code:

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

class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Colors.orange,
          title: Row(children: [
            Text('WOW Pizza',
                style: TextStyle(fontSize: 20.0),),
            SizedBox(width: 160.0,),

            Image.asset(
              'Images/twitter.png',
              fit: BoxFit.contain,
              height: 30,),

            SizedBox(width: 10.0,),

            Image.asset(
              'Images/facebook.png',
              fit: BoxFit.contain,
              height: 40,),

        ]),
      ),
      body: SafeArea(
        child: Column(
          children: <Widget>[
            ButtonBar(
```

```
        alignment: MainAxisAlignment.center,
        children: [
            OutlineButton(
                shape: StadiumBorder(),
                highlightedBorderColor: Colors.red,
                child: Text('Vegetable Pizza',
                    style: TextStyle(fontSize: 20.0, color: Colors.orange),),
                onPressed: () {
                    Navigator.pushNamed(context, '0');
                },
            ),
            OutlineButton(
                shape: StadiumBorder(),
                highlightedBorderColor: Colors.red,
                child: Text('Cheese Pizza',
                    style: TextStyle(fontSize: 20.0, color: Colors.orange),),
                onPressed: () {
                    Navigator.pushNamed(context, '1');
                },
            ),
            OutlineButton(
                shape: StadiumBorder(),
                highlightedBorderColor: Colors.red,
                child: Text('Fries',
                    style: TextStyle(fontSize: 20.0, color: Colors.orange),),
                onPressed: () {
                    Navigator.pushNamed(context, '2');
                },
            ),
        ],
    ),
    Center(
        child: Column(children: [
            Image.asset(
                'Images/meal.jpg',
                height: 300.0,
                width: 400.0,
            ),
            Text(
                "Hi, I'm the Pizza Assistant, what can I help you order?",
                style: TextStyle(fontSize: 30.0),
                textAlign: TextAlign.center,
            ),
        ]),
    ),

```

```

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

```

11- Open **main.dart** ( lab\_6 → lib → main.dart) and **delete** all its content.

12- The **main.dart** file will be configured to startup **home.dart** content when you run this app, and it will include the navigation and named routes (keys) where you will use the following named routes and keys:

Key	Class Name	File Name
0	Vpizaa()	Vpizza.dart
1	Chpizza()	Chpizza.dart
2	Fries()	fries.dart
3	Twitter()	twitter.dart
4	Facebook()	facebook.dart

Add the following code to **main.dart** :

**Importance note:** You will get a red underline for each class name in the routes map because these classes are not created yet.

```

import 'package:flutter/material.dart';
import 'home.dart';
import 'chpizza.dart';
import 'facebook.dart';
import 'twitter.dart';
import 'vpizaa.dart';
import 'fries.dart';

main() {
  runApp(
    MyApp(),
  )
}

```

```
    );
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Home(),
      routes: {
        '0': (context) => Vpizaa(),
        '1': (context) => Chpizza(),
        '2': (context) => Fries(),
        '3': (context) => Twitter(),
        '4': (context) => Facebook(),
      },
    );
}
```

13- Open **home.dart**, and then type the following code:

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

class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Colors.orange,
          title: Row(
            children: [
              Text(
                'WOW Pizza',
                style: TextStyle(fontSize: 20.0),
              ),
              SizedBox(width: 160.0),
              Image.asset(
                'Images/twitter.png',
                fit: BoxFit.contain,
                height: 30,
              ),
              SizedBox(width: 10.0),
            ],
          ),
        ),
      ),
    );
  }
}
```

```
        Image.asset(
            'Images/facebook.png',
            fit: BoxFit.contain,
            height: 40,),
        ],
    ),
),
),

body: SafeArea(
    child: Column(
        children: <Widget>[

    ButtonBar(
        alignment: MainAxisAlignment.center,
        children: [

            OutlineButton(
                shape: StadiumBorder(),
                highlightedBorderColor: Colors.red,
                child: Text('Vegetable Pizza',
                    style: TextStyle(fontSize: 20.0, color: Colors.orange),),
                onPressed: () {},
            ),

            OutlineButton(
                shape: StadiumBorder(),
                highlightedBorderColor: Colors.red,
                child: Text('Cheese Pizza',
                    style: TextStyle(fontSize: 20.0, color: Colors.orange),),
                onPressed: () {},
            ),

            OutlineButton(
                shape: StadiumBorder(),
                highlightedBorderColor: Colors.red,
                child: Text('Fries',
                    style: TextStyle(fontSize: 20.0, color: Colors.orange),),
                onPressed: () {},
            ),

        ],
    ),
),

Center(
    child: Column(children: [
        Image.asset(
```

```

        'Images/meal.jpg',
        height: 300.0,
        width: 400.0,),

      Text("Hi, I'm the Pizza Assistant, what can I help you order?",
        style: TextStyle(fontSize: 30.0),
        textAlign: TextAlign.center,
        ),
      ],
    ),
  ],
),
),
),
),
),
),
);
}
}

```

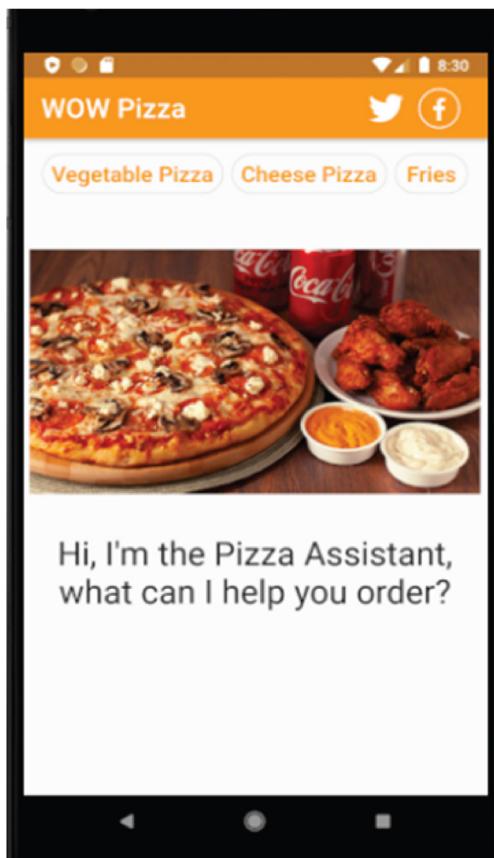
14- To test your app so far, open **main.dart** file, and then change all the routes part to comments by adding a double forward slash as illustrated in the following figure to stop this temporary error.

```

18   home: Home(),
19   // routes: {
20   //   '0': (context) => Vpizaa(),
21   //   '1': (context) => Chpizza(),
22   //   '2': (context) => Fries(),
23   //   '3': (context) => Twitter(),
24   //   '4': (context) => Facebook(),
25   // },
26   // );
27   }
28 }

```

Now, run your app. The **home.dart** will be as follows:



15- Remove the comment signs again from **main.dart**. You may use **Ctrl + /** to add or remove the double forward slash of any command.

16- Now, you want to add the navigation configuration to each **onPressed:()** method in each button. To do that, add the grey highlighted part of the following code to each button in **home.dart** file:

```
ButtonBar(  
  alignment: MainAxisAlignment.center,  
  children: [  
  
    OutlineButton(  
      shape: StadiumBorder(),  
      highlightedBorderColor: Colors.red,  
      child: Text('Vegetable Pizza',  
        style: TextStyle(fontSize: 20.0, color: Colors.orange),),  
      onPressed: () {  
        Navigator.pushNamed(context, '0');  
      },),
```

```
OutlineButton(  
    shape: StadiumBorder(),  
    highlightedBorderColor: Colors.red,  
    child: Text('Cheese Pizza',  
        style: TextStyle(fontSize: 20.0, color: Colors.orange),),  
    onPressed: () {  
        Navigator.pushNamed(context, '1');  
    },),  
  
OutlineButton(  
    shape: StadiumBorder(),  
    highlightedBorderColor: Colors.red,  
    child: Text('Fries',  
        style: TextStyle(fontSize: 20.0, color: Colors.orange),),  
    onPressed: () {  
        Navigator.pushNamed(context, '2');  
    },  
,  
],  
,
```

17- Now , you will add a navigation route for the Facebook and twitter images. But because these are images, you should add the `GestureDetector()` class to add the navigation code. The following is the full code for the `home.dart` file :

```
import 'package:flutter/material.dart';
class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Colors.orange,
          title: Row(
            children: [
              Text('WOW Pizza',
                style: TextStyle(fontSize: 20.0),),
              SizedBox(width: 160.0,),

              Container(
                child: GestureDetector(
                  onTap: () {
```

```
        Navigator.pushNamed(context, '3');
    },
    child: Image.asset(
        'Images/twitter.png',
        fit: BoxFit.contain,
        height: 30,),),),
SizedBox(width: 10.0,),

Container(
    child: GestureDetector(
        onTap: () {
            Navigator.pushNamed(context, '4');
        },
        child: Image.asset(
            'Images/facebook.png',
            fit: BoxFit.contain,
            height: 40,),,
        ),
    ),
],
),
),
),
),

body: SafeArea(
    child: Column(
        children: <Widget>[
            ButtonBar(
                alignment: MainAxisAlignment.center,
                children: [

                    OutlineButton(
                        shape: StadiumBorder(),
                        highlightedBorderColor: Colors.red,
                        child: Text('Vegetable Pizza',
                            style: TextStyle(fontSize: 20.0, color: Colors.orange),),
                        onPressed: () {
                            Navigator.pushNamed(context, '0');
                        },
                    ),,

                    OutlineButton(
                        shape: StadiumBorder(),
                        highlightedBorderColor: Colors.red,
                        child: Text('Cheese Pizza',
                            style: TextStyle(fontSize: 20.0, color: Colors.orange),),
                        onPressed: () {
                            Navigator.pushNamed(context, '1');
                        },
                    ),
                ],
            ),
        ],
    ),
)
```

```
        },),  
  
        OutlineButton(  
            shape: StadiumBorder(),  
            highlightedBorderColor: Colors.red,  
            child: Text('Fries',  
                style: TextStyle(fontSize: 20.0, color: Colors.orange),),  
            onPressed: () {  
                Navigator.pushNamed(context, '2');  
            },  
        ),  
    ],  
),  
  
Center(  
    child: Column(children: [  
        Image.asset(  
            'Images/meal.jpg',  
            height: 300.0,  
            width: 400.0, ),  
        Text("Hi, I'm the Pizza Assistant, what can I help you order?",  
            style: TextStyle(fontSize: 30.0),  
            textAlign: TextAlign.center,  
        ),  
    ],  
),  
],  
),  
),  
);  
};  
}
```

18- Copy all the lines of code in **home.dart**, and paste it in **chpizza.dart**, **vpizza.dart**, **fries.dart**, **facebook.dart**, and **twitter.dart**.

19- Open **chpizza.dart** , and replace the class name **Home** in the first line with **Chpizza**

20- Open **vpizza.dart**, and replace the class name **Home** in the first line with **Vpizza**

21- Open **fries.dart**, and replace the class name **Home** in the first line with **Fries**

22- Open `facebook.dart`, and replace the class name `Home` in the first line with `Facebook`

- 23- Open **twitter.dart**, and replace the class name **Home** in the first line with **Twitter**
- 24- Open **vpizza.dart** and scroll down at the last 6 lines and replace the **meal.jpg** with **Vpizza.png**
- 25- Open **chpizza.dart** and replace the **meal.jpg** with **cheesepizza.png** .
- 26- Open **fries.dart** and replace the **meal.jpg** with **Fpizza.png**.
- 27- Open **facebook.dart**. Remove only the content of the body widget. Then, add the following code to the body widget:

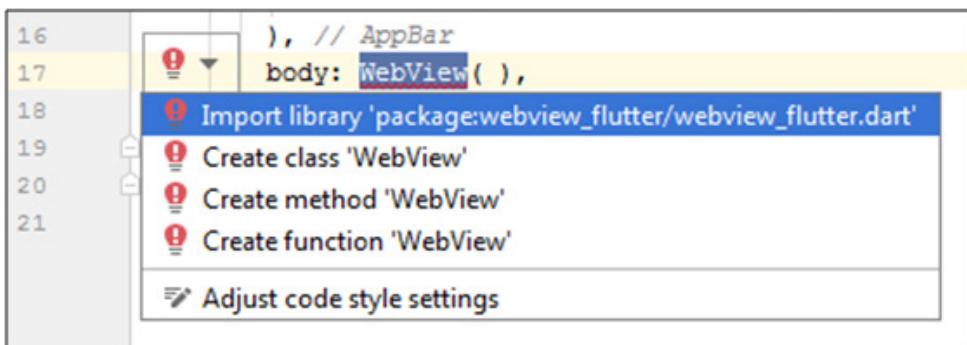
```
WebView(  
  initialUrl: 'https://www.facebook.com',  
  javascriptMode: JavascriptMode.unrestricted),
```

The full code of the **facebook.dart** file follows:

```
import 'package:flutter/material.dart';  
import 'package:webview_flutter/webview_flutter.dart';  
  
class Facebook extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          backgroundColor: Colors.orange,  
          title: Row(  
            children: [  
              Text('WOW Pizza',  
                style: TextStyle(fontSize: 25.0),),  
              SizedBox(width: 160.0,),  
              Container(  
                child: GestureDetector(  
                  onTap: () {  
                    Navigator.pushNamed(context, '3');  
                  },  
                  child: Image.asset(  
                    'Images/twitter.png',  
                  ),  
                ),  
              ),  
            ],  
          ),  
        ),  
      ),  
    );  
  }  
}
```

28-Double click the **WebView** widget, and click the red lamp icon as illustrated in the following figure. Then, select:

```
Import library 'package:webview_flutter/webview_flutter.dart'.
```



29- Open the **twitter.dart**. Remove only the content of the body widget. Then, add the following code to the body widget:

```
WebView(  
  initialUrl: 'https://www.twitter.com',  
  javascriptMode: JavascriptMode.unrestricted),
```

The full code of **twitter.dart** file as follows:

```
import 'package:flutter/material.dart';  
  
class Twitter extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        appBar: AppBar(  
          backgroundColor: Colors.orange,  
          title: Row(  
            children: [  
              Text('WOW Pizza',  
                style: TextStyle(fontSize: 20.0),),  
  
              SizedBox(width: 160.0,),  
  
              Container(  
                child: GestureDetector(  
                  onTap: () {  
                    Navigator.pushNamed(context, '3');  
                  },  
                  child: Image.asset(  
                    'Images/twitter.png',  
                    fit: BoxFit.contain,  
                    height: 30,  
                  )),),  
  
              SizedBox(width: 10.0,),  
  
              Container(  
                child: GestureDetector(  
                  onTap: () {  
                    Navigator.pushNamed(context, '4');  
                  },  
                ),  
              ),  
            ],  
          ),  
        ),  
      ),  
    );  
  }  
}
```

30-Double click the **WebView** widget , and click the red lamp icon. Then, select:

```
Import library 'package:webview_flutter/webview_flutter.dart'.
```

31- Run your app. Test the navigation buttons, Facebook, and twitter images.

Yay..!! You're almost done.

If you understand this lab and how it is designed, you should be confident enough to design a wonderful Android and iOS Apps with Flutter.

Let us share with you the following ideas about this small Flutter project:

- We should replace the Facebook and Twitter web sites with the Facebook and Twitter links of this company (WoW Pizza Store Facebook and Twitter pages).
  - Beware that the main goal of apps is providing services directly to app users with some interactivity such as notifications, offers, tracking system and others, rather than giving details about the company history. Therefore, this app which you have created still needs some other features such as placing an order, selecting the pizza size and components , selecting delivery or pickup options , and selecting the nearest branch to your location, depending on the GPS location on Google map.

These features will be discussed in the next lessons. Good luck !

## Lesson 7: Visual, Behavioral, and Motion-Rich Widgets

### Implementing Material Design Guidelines - Part 1

<b>Introduction .....</b>	7-2
<b>BottomNavigationBar Widget .....</b>	7-2
<b>DefaultTabController, TabBar, and TabBarView Widgets .....</b>	7-5
<b>ListTile Widget .....</b>	7-11
<b>ListView Widget .....</b>	7-15
<b>Drawer Widget .....</b>	7-18
<b>DataTable Widget .....</b>	7-29
<b>SelectableText Widget .....</b>	7-32
<b>Stack Widget .....</b>	7-35
<b>Lab : 7 .....</b>	7-39
Lab A: Creating a Flutter App using BottomNavigationBar Navigation Technique .....	7-40
Lab B: Using DataTable Sorting Built-in function.....	7-45

## Introduction

In this lesson, you will learn more about Flutter widgets especially those related to navigating the app screens in different ways of navigations and configurations techniques.

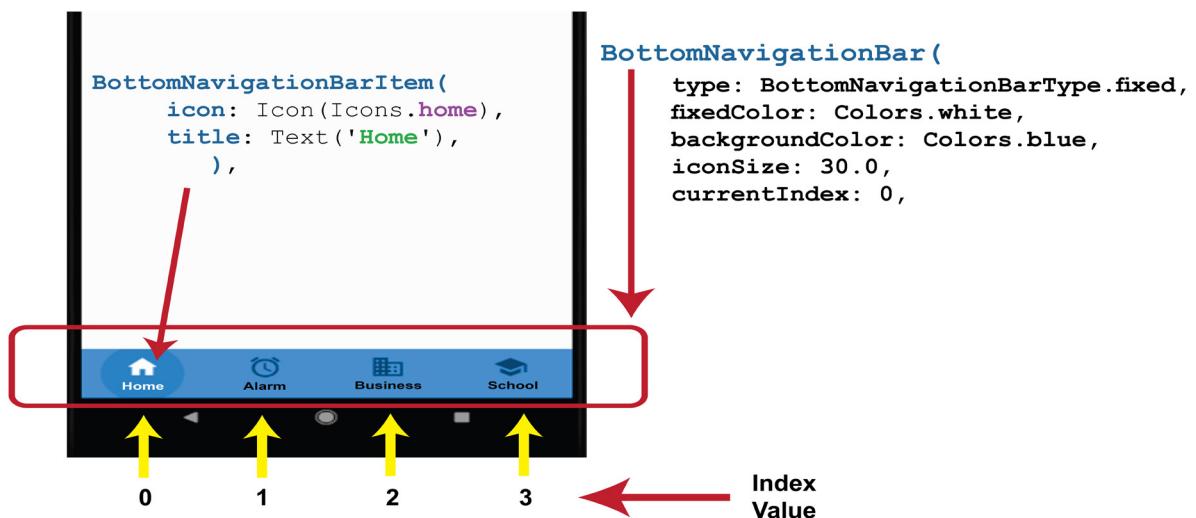
You may add more than one navigation method to your app that depends on your app purpose and your customers' targets. Remember, you should make your app as easy to navigate and to find information as possible.

## BottomNavigationBar Widget

Bottom navigation bar is a material widget that is displayed at the bottom of an app for selecting among a small number of views, typically between three and five.

The bottom navigation bar consists of multiple items in the form of text labels, icons, or both, laid out on top of a piece of material. It provides quick navigation between the top-level views of an app.

**BottomNavigationBar** widget depends on its work on a list of **BottomNavigationBarItem** widgets where each **BottomNavigationBarItem** represents one item of the bottom navigation bar (icon and text) as illustrated in the following figure:



Each item (icon) of this bar in the list has an index value. The first item from the left side has 0 index value. You can configure each of these items to be the default choice. It is quite bigger in size compared to other icons, and you can give it a

distinguished color. In the previous figure, it was white in color. You can configure the default index by adding **currentIndex** property and value to the bottom navigation bar. In the previous figure, the **currentIndex:0**; therefore, the default selection was for the home icon.

Also, you can add many properties to the **BottomNavigationBar** such as icon size to control the size of icons. However, the most important property is the **type** property whose value should be **BottomNavigationBarType.fixed**, if you have more than three items in this bottom navigation bar.

#### **Example:**

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** then click **Next**.

4- Type : **bottom\_navigation\_bar** for Project Name and create a new folder: **Lesson\_07** for Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain, then click **Finish**

6- Open **main.dart** (**bottom\_navigation\_bar → lib → main.dart**).

7- Delete all the code in **main.dart** file.

8- Type the following code which will create a simple Flutter app including bottom navigation bar :

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

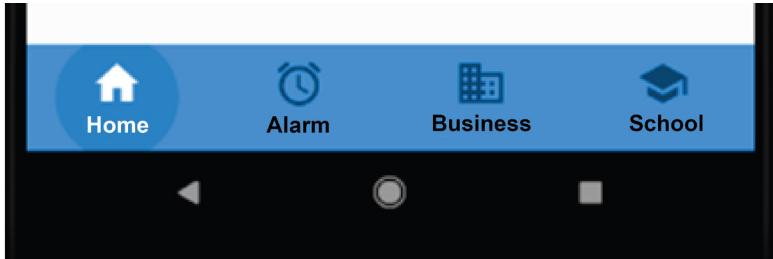
main() {
  runApp(
    MyApp(),
  );
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Lesson 07: Bottom Navigation Bar '),

```

```
    ),  
  
    body: Container(),  
  
    bottomNavigationBar: BottomNavigationBar(  
        type: BottomNavigationBarType.fixed,  
        fixedColor: Colors.white,  
        backgroundColor: Colors.blue,  
        iconSize: 30.0,  
        currentIndex: 0,  
  
        items: [  
            BottomNavigationBarItem(  
                icon: Icon(Icons.home),  
                title: Text('Home'),  
            ),  
  
            BottomNavigationBarItem(  
                icon: Icon(Icons.access_alarm),  
                title: Text('Alarm'),  
            ),  
  
            BottomNavigationBarItem(  
                icon: Icon(Icons.business),  
                title: Text('Business'),  
            ),  
  
            BottomNavigationBarItem(  
                icon: Icon(Icons.school),  
                title: Text('School'),  
            ),  
        ],  
    ),  
);  
}  
}
```

9- The following is your app run output:



Also, you may replace the `fixedColor` property with `selectedItemColor` and test the effect of `selectedFontSize` and `unselectedFontSize` as illustrated in the following code:

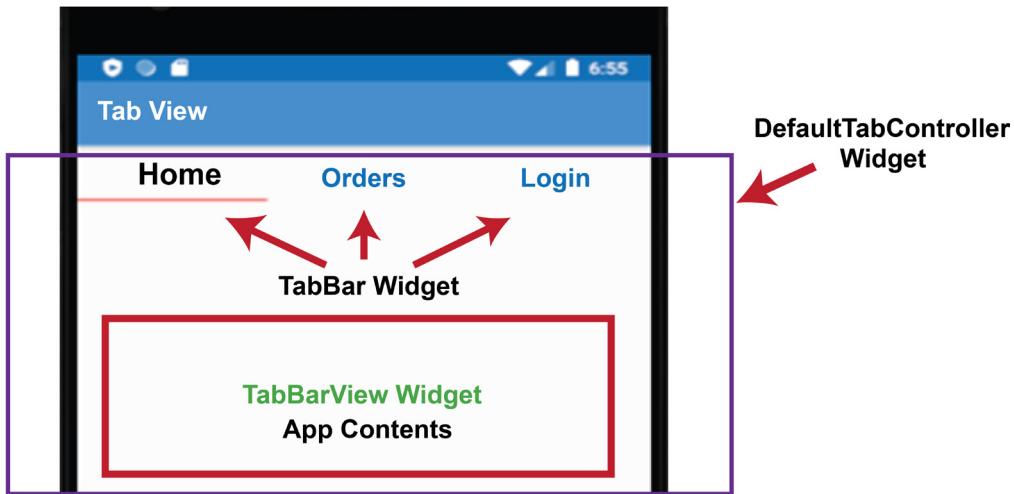
```
bottomNavigationBar: BottomNavigationBar(  
  type: BottomNavigationBarType.fixed,  
  //fixedColor: Colors.white,  
  backgroundColor: Colors.blue,  
  selectedItemColor: Colors.white,  
  selectedFontSize: 19.0,  
  unselectedFontSize: 14.0,  
  iconSize: 30.0,  
  currentIndex: 0,
```

In the lab at the end of this lesson, you will learn how to configure the navigation between app interfaces using the bottom navigation bar icons.

## DefaultTabController , TabBar , and TabBarView Widgets

The `DefaultTabController`, `TabBar`, and `TabBarView` widgets are used to organize your widgets or app contents into tabs.

As illustrated in the below figure, to use tabs, you will first need a tab controller (`DefaultTabController`) to keep the selected tab and the visible content in sync. The easiest way to do this is using the `DefaultTabController` widget. Once you have that, you will need a widget to show the tabs that the app user would use to switch between the different interfaces of your app. This is where `TabBar` comes in. The `TabBar` takes a list of tab widgets. You can have the tab show text, an icon, or a child widget.



In this example, you will create a new Flutter project that explains step by step how to create an app using tab view technique to navigate the app contents.

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** then click **Next**.

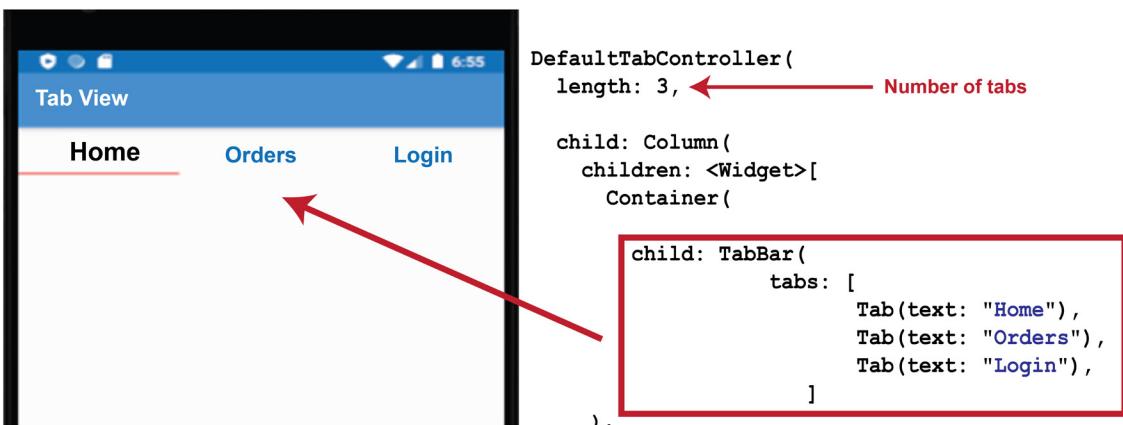
4- Type :**tabbar\_tabbarview** for Project Name, and **Lesson\_07** for Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain, then click **Finish**

6- Open **main.dart** (**tabbar\_tabbarview → lib → main.dart**).

7- Delete all the code in **main.dart** file.

8- The following figure gives you an idea about the widgets which you will add to the **main.dart** file and the role of each widget and property.



9- Type the following code in **main.dart** which will create a simple Flutter app including tab navigation bar :

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Tab View'),
        ),
        body: DefaultTabController(
          length: 3,
          child: Column(
            children: <Widget>[
              Container(
                child: TabBar(
                  labelColor: Colors.black,
                  unselectedLabelColor: Colors.blue,
                  indicatorColor: Colors.red,
                  labelStyle: TextStyle(
                    fontWeight: FontWeight.bold,
                    fontFamily: 'Arial',
                    fontSize: 20.0),
                  tabs: [
                    Tab(text: "Home"),
                    Tab(text: "Orders"),
                    Tab(text: "Login"),
                  ],
                ),
              ),
            ],
          );
        );
    }
}
```

The following table includes the list of properties which you have already added to the `TabBar()` widget in the previous code and the role of each:

Widget Property	Value	Use for
<code>labelColor:</code>	<code>Colors.black</code>	The color of the selected tab.
<code>unselectedLabelColor:</code>	<code>Colors.blue</code>	The color of the unselected tab.
<code>indicatorColor:</code>	<code>Colors.red</code>	The color of the underline.

Finally, you will need to create the content of each tab. This is where you use `TabBarView`. Each widget in the list children corresponds to tabs in the `TabBar` widget. There must be a one to one match between the tabs and the `TabView` children.

10- Create three Dart files to represent the content of each tab.

Right click the `lib` folder → **New** → **Dart File**.

Type **screen1** for the file name, then click **OK**.

Repeat this step to create **screen2** and **screen3** files.

11- Open **screen1.dart** and add the following codes to it:

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

class Screen1 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Container(
          child: Text('Home',
            style: TextStyle(fontSize: 30)),
        )));
  }
}
```

12- Copy all the content of **screen1.dart**, paste it in **screen2.dart** and in **screen3.dart**.

13- Open **screen2.dart** and replace the class name **Screen1** with **Screen2**

and the **Text()** widget value '**Home**' with '**Orders**'.

14- Open **screen3.dart** and replace the class name **Screen1** with **Screen3**

and the **Text()** widget value '**Home**' with '**User**'.

15- Open **main.dart**, and at the first line, add the following code to import screen1.dart, screen2.dart, and screen3.dart to **main.dart** file:

```
import 'screen1.dart';
import 'screen2.dart';
import 'screen3.dart';
```

16- Now, you will use the **TabBarView** and **TabView** widgets to add the contents of each tab as illustrated in the following code:

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'screen1.dart';
import 'screen2.dart';
import 'screen3.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Tab View'),
        ),
        body: TabBarView(
          children: [
            Screen1(),
            Screen2(),
            Screen3(),
          ],
        ),
      ),
    );
}
```

```
),

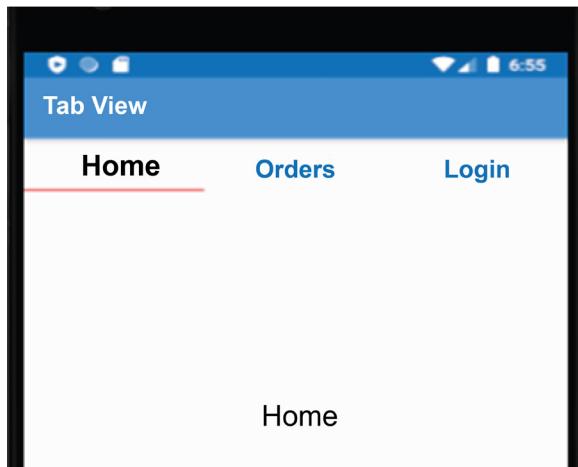
body: DefaultTabController(
    length: 3,
    child: Column(
        children: <Widget>[
            Container(
                child: TabBar(
                    labelColor: Colors.black,
                    unselectedLabelColor: Colors.blue,
                    indicatorColor: Colors.red,
                    labelStyle: TextStyle(
                        fontWeight: FontWeight.bold,
                        fontFamily: 'Arial',
                        fontSize: 20.0),
                    tabs: [
                        Tab(text: "Home"),
                        Tab(text: "Orders"),
                        Tab(text: "Login"),
                    ]),
            ),
            Expanded(
                child: Container(
                    child: TabBarView(
                        children: [
                            Container(
                                child: Screen1(),),
                            Container(
                                child: Screen2(),),
                            Container(
                                child: Screen3(),),
                        ],
                    ),
                ),
            ),
        ],
    ),
),
```

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

You may ask what is the Expanded widget which was used in the previous code?

Almost every Flutter layout uses rows and columns, and they are pretty cool. They can hug their children tightly or spread them out and relax. But what if you would like one of those children to stretch and fill the extra space? Just wrap it in an Expanded widget and watch it grow.

17 - Run your app and test your app tabs. The following figure displays the app run output:



## ListTile Widget

In some app layout designs, you have a list of items that you really want them to follow the material design specifications. You could spend hours working out the perfect item layout using rows, columns, containers, and various amounts of padding. Or, you could just use a **ListTile** widget. **ListTile** widget implements the material design list spec for you with full and easy control on its content. The following figure displays how the **ListTile** widget looks like and some **ListTile()** widget properties:

**Example:**

In this example, you will create a new Flutter project that displays its contents using `ListTile()` widget. Perform the following steps:

- 1- Open **Android Studio**
- 2- Click **File → New → New Flutter Project**
- 3- Select **Flutter Application** then click **Next**.
- 4- Type : **list\_tile\_widget** for Project Name and create a new folder : **Lesson\_07** for Project Location. Click **Next**.
- 5- Type : **androidatc.com** for Company domain then click **Finish**
- 6- Open **main.dart** (`list_tile_widget → lib → main.dart`).
- 7- Delete all the code in **main.dart** .
- 8- Type the following code:

```

import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override

```

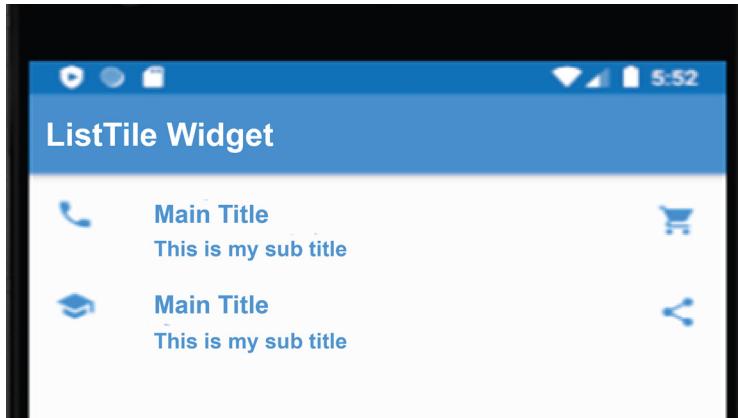
```
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(  
        title: Text('ListTile Widget'),  
      ),  
  
      body: Column(  
        children: <Widget>[  
  
          ListTile(  
            leading: Icon(Icons.phone),  
            title: Text('Main Title'),  
            subtitle: Text('This is my sub title'),  
            trailing: Icon(Icons.shopping_cart),  
            onTap: () => print('onTap Action'),  
            onLongPress: () => print('On Long Press Action'),  
            dense: false, // true means intensive content.  
            selected: true, // sets this tile as default.  
            enabled: true, // means that it can be tapped.  
  
          ),  
        ],  
      )),  
    );  
}
```

9- Add additional **ListTile** widget to your app as illustrated in the following code:

```
import 'package:flutter/material.dart';  
  
void main() => runApp(MyApp());  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'My App',  
      theme: ThemeData(primarySwatch: Colors.teal),  
      home: MyHomePage(),  
    );  
  }  
}  
  
class MyHomePage extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('My Home Page'),  
      ),  
      body: Center(  
        child: Text('Welcome to My Home Page!'),  
      ),  
    );  
  }  
}
```

```
home: Scaffold(  
    appBar: AppBar(  
        title: Text('ListTile Widget'),  
    ),  
  
    body: Column(  
        children: <Widget>[  
  
            ListTile(  
                leading: Icon(Icons.phone),  
                title: Text('Main Title'),  
                subtitle: Text('This is my sub title'),  
                trailing: Icon(Icons.shopping_cart),  
                onTap: () => print('onTap Action'),  
                onLongPress: () => print('On Long Press Action'),  
                dense: false,  
                selected: true,  
                enabled: true,  
            ),  
  
            ListTile(  
                leading: Icon(Icons.school),  
                title: Text('Main Title'),  
                subtitle: Text('This is my sub title'),  
                trailing: Icon(Icons.share),  
                onTap: () => print('onTap Action'),  
                onLongPress: () => print('On Long Press Action'),  
                dense: false,  
                selected: true,  
                enabled: true,  
            ),  
        ],  
    )),  
);  
}  
}
```

10- Run your app. The run output follows:



## ListView Widget

If you want to set your app items to appear in a scrollable list, the **ListView** widget is the perfect choice for your app layout. The default scroll direction of your app items is vertical, however, you can change the scroll direction to horizontal. The following code displays the main component of the **ListView** widget:

```
ListView(  
    children [item1, item2, item3],  
)
```

### **Example:**

In the following example, you will create a Flutter app including a simple **ListView** widget having two **ListTile** widgets and one image as children items.

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** then click **Next**.

4- Type : **list\_view\_widget** for Project Name, and **Lesson\_07** for the Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain then click **Finish**

6- Right click the project main directory (**list\_view\_widget**), select **New → Directory**.

Type **Images**, and click OK.

7- Configure the **pubspec.yaml** file to use the **Images** folder as a default location for all images in this project as illustrated in the following figure:

```
43      # To add assets to your application,  
44      assets:  
45          - Images/  
46              #   - images/a_dot_ham.jpeg
```

8- From your **Lab Source Code** folder, open the **images** folder, **copy** the “**sofa.jpg**” image, then **paste** it in **Images** folder in your app.

9- Open **main.dart** (**list\_tile\_widget** → **lib** → **main.dart**).

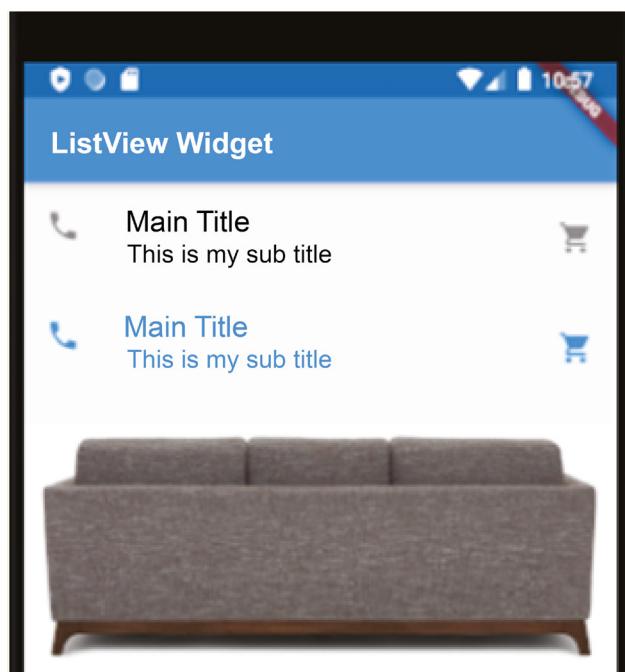
10- **Delete** all the code in **main.dart** file.

11- Type the following code in **main.dart** file. Note that you will add two **ListTile** widgets which you have used in the previous section of this lesson as a **ListView** item, as well as add “**sofa.jpg**” image as **ListView** widget item.

```
import 'package:flutter/material.dart';  
  
void main() => runApp(MyApp());  
  
class MyApp extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(  
                    title: Text('ListView Widget'),  
                ),  
  
                body: ListView(  
                    children: <Widget>[  
  
                        ListTile(  
                            leading: Icon(Icons.phone),  
                            title: Text('Main Title'),  
                            subtitle: Text('This is my sub title'),  
                            trailing: Icon(Icons.shopping_cart),  
                            onTap: () => print('onTap Action'),  
                        ),  
                    ],  
                ),  
            ),  
        );  
    }  
}
```

```
        ),  
  
        ListTile(  
            leading: Icon(Icons.phone),  
            title: Text('Main Title'),  
            subtitle: Text('This is my sub title'),  
            trailing: Icon(Icons.shopping_cart),  
            onTap: () => print('onTap Action'),  
            selected: true,  
        ),  
  
        Image(  
            image: AssetImage('Images/sofa.jpg'),  
            height: 300.0,  
        ),  
  
    ],  
),  
),  
);  
}  
}  
}
```

12- Run your app. You will get the following run output:



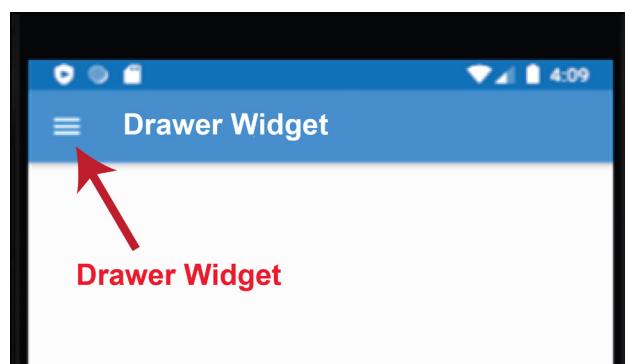
As you see here, these widget items can be any widgets.

Try to copy these two **ListTile** widgets and paste them again as new item for **ListView** widget. Repeat this step five times and run your app. Scroll down (vertically) to see all your app content.

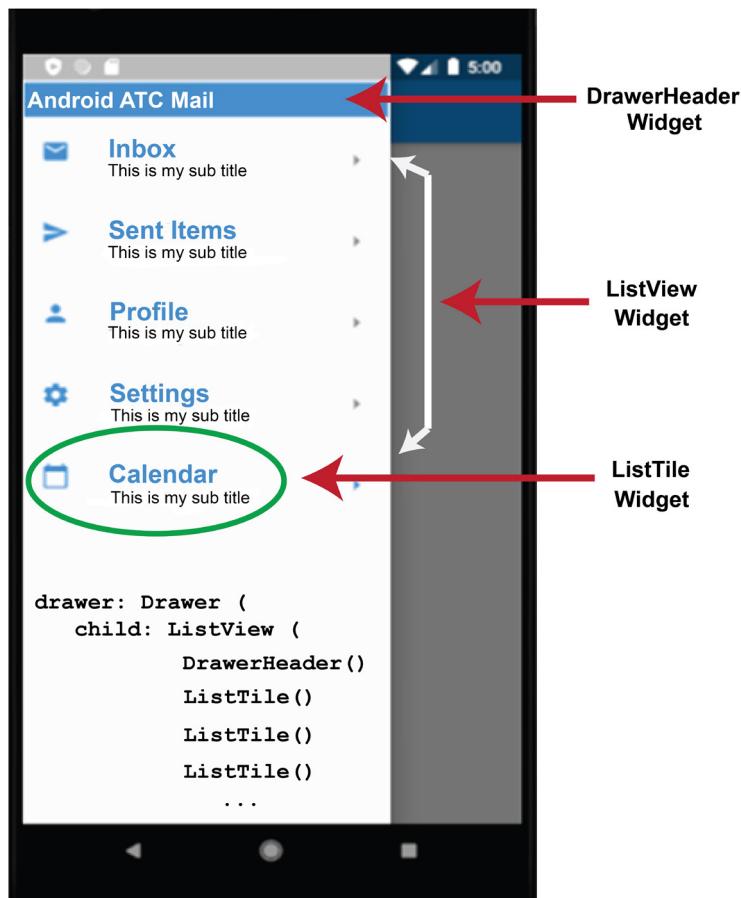
## Drawer Widget

In apps that use Material Design, there are two primary options for navigation: tabs and drawers. When there is insufficient space to support tabs, drawers provide a handy alternative.

The following figure displays how **Drawer** widget looks like:



The following figure displays the **Drawer** widget components (navigation items):



In Flutter, use the **Drawer** widget in combination with a **Scaffold** to create a layout with a Material Design drawer. This recipe uses the following steps:

- a) Create a Scaffold.
- b) Add a drawer.
- c) Populate the drawer with items.
- d) Close the drawer programmatically.

### Example:

In this example, you will create a new Flutter project including a **Drawer()** widget and add **ListView** navigation items .

Perform the following steps:

- 1- Open **Android Studio**

2- Click **File** → **New** → **New Flutter Project**

3- Select **Flutter Application** then click **Next**.

4- Type : **drawer\_widget** for Project Name and **Lesson\_07** for Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain then click **Finish**.

6- In this app, you will create the **main.dart** file to include all navigation routes and startup the **home.dart** file which will also include the app home interface and the Drawer widget. This widget, as illustrated in the previous figure, will include a link to other five pages (interfaces) as follows:

<b>Drawer Items</b>	<b>File</b>	<b>Class Name</b>	<b>Navigation Link Name</b>
Inbox	mail_interface.dart	Mail()	mail
Sent Items	sent_items_interface.dart	SentItems()	sentItem
Profile	profile_interface.dart	Profile()	profile
Settings	settings.dart	Settings()	settings
Calendar	calendar.dart	Calendar ()	Calendar

7- Add a new Dart file: **home.dart** (right click **lib** folder → **New** → **Dart File**). Open this file and add the following code:

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

class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Drawer Widget'),
        ),
        drawer: Drawer(),
        body: Center(
          child: Container(
            child: Text('Welcome',
              style: TextStyle(fontSize: 20.0),
            ),
          ),
        ),
      ),
    );
}
```

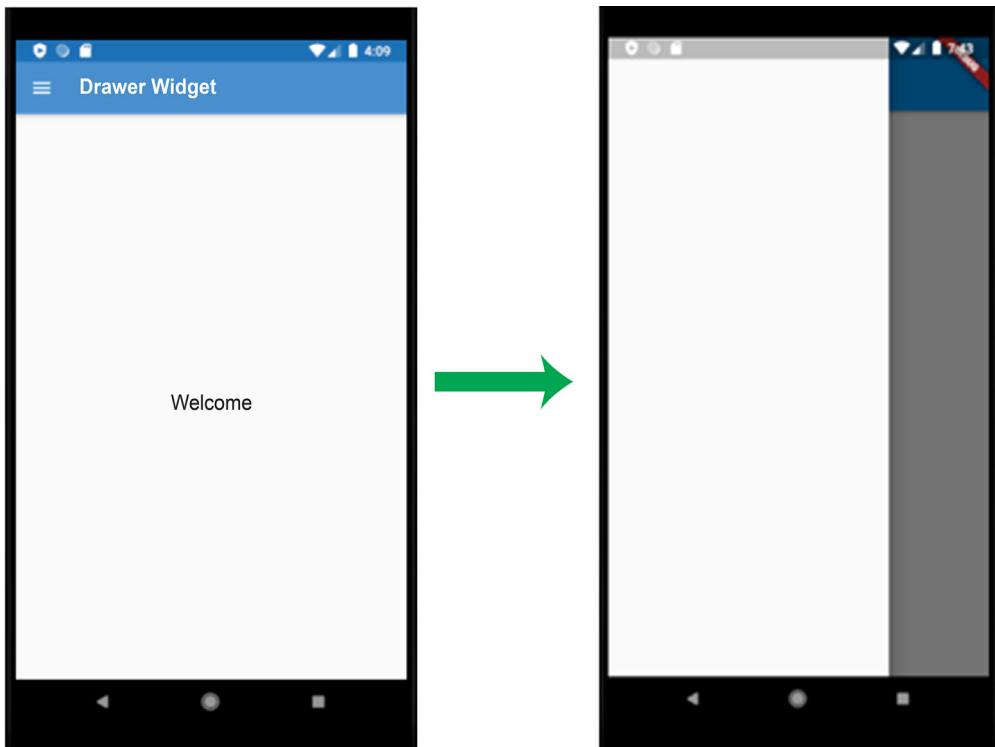
```
        ),  
        ),  
        ),  
    );  
}  
}
```

8- Open **main.dart** file (list\_tile\_widget → lib → main.dart).

Delete all the code in **main.dart** then type the following code:

```
import 'home.dart';  
import 'package:flutter/material.dart';  
  
void main() => runApp(MyApp());  
  
class MyApp extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Home(),  
        );  
    }  
}
```

9- Run your app. You will get an empty Drawer as illustrated in the following figure:



10- Open the **home.dart** file and add the **ListView** widget as a child widget of **Drawer** widget. Also, add five **ListTile** widgets as children widgets of the **ListView** widget to represent the Drawer items (Inbox, Sent Items, Profile, Settings, and Calendar). Let us postpone the navigation routes configurations to the next step:

The **home.dart** file will be as follows :

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

class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Drawer Widget'),
        ),
        drawer: Drawer(
          child: ListView(children: <Widget>[
            ListTile(

```

```
        leading: Icon(Icons.mail,
          color: Colors.blue,),
        title: Text('Inbox',
          style: TextStyle(color: Colors.blue, fontSize: 20.0),),
        subtitle: Text('This is my sub title'),
        trailing: Icon(Icons.arrow_right),
        onTap: () => print('1'),
      ),

      ListTile(
        leading: Icon(Icons.send,
          color: Colors.blue,),
        title: Text('Sent Items',
          style: TextStyle(color: Colors.blue, fontSize: 20.0),),
        subtitle: Text('This is my sub title'),
        trailing: Icon(Icons.arrow_right),
        onTap: () => print('2'),
      ),

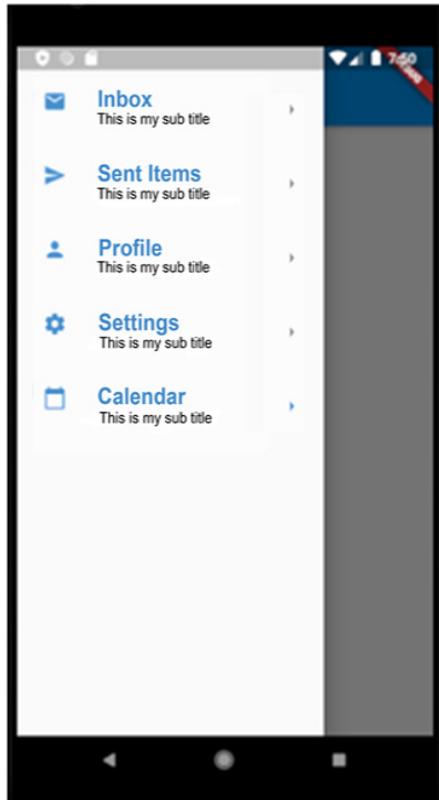
      ListTile(
        leading: Icon(Icons.person,
          color: Colors.blue,),
        title: Text('Profile',
          style: TextStyle(color: Colors.blue, fontSize: 20.0),),
        subtitle: Text('This is my sub title'),
        trailing: Icon(Icons.arrow_right),
        onTap: () => print('3'),
      ),

      ListTile(
        leading: Icon(Icons.settings,
          color: Colors.blue,),
        title: Text('Settings',
          style: TextStyle(color: Colors.blue, fontSize: 20.0),),
        subtitle: Text('This is my sub title'),
        trailing: Icon(Icons.arrow_right),
        onTap: () => print('4'),
      ),

      ListTile(
        leading: Icon(Icons.calendar_today,
          color: Colors.blue,),
        title: Text('Calendar',
          style: TextStyle(color: Colors.blue, fontSize: 20.0),),
        subtitle: Text('This is my sub title'),
```

```
        trailing: Icon(Icons.arrow_right,
            color: Colors.blue,),
        onTap: () => print('5'),
    ),
],
),
),
),
),
),
),
),
),
);
}
}
```

11- Run your app and you will get the following run output:



12- Open **home.dart** file and use the **DrawerHeader()**widget to add a header to your drawer items.

To control this drawer header height and other formats, add this **DrawerHeader** widget as a child of the Container widget. You can do this easily by adding the following gray highlighted part of the code to **home.dart** :

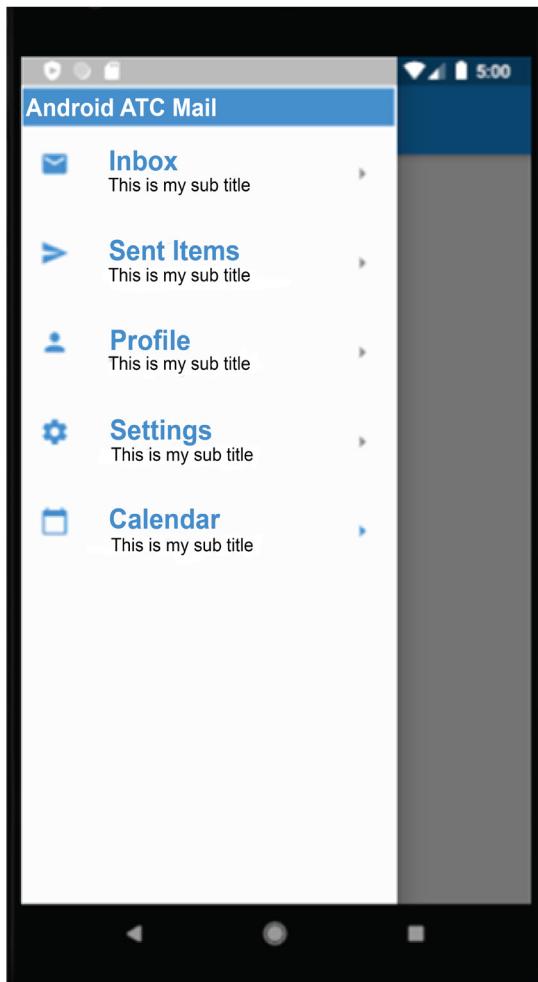
```
import 'package:flutter/material.dart';

class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Drawer Widget'),),
        drawer: Drawer(
          child: ListView(children: <Widget>[
            Container(
              height: 35.0,
              child: DrawerHeader(
                margin: EdgeInsets.all(2.0),
                padding: EdgeInsets.all(2.0),
                child: Text(
                  'Android ATC Mail',
                  style: TextStyle(fontSize: 20.0, color: Colors.white),),
                decoration: BoxDecoration(color: Colors.blue),
              ),
            ),
            ListTile(
              leading: Icon(Icons.mail,
                color: Colors.blue,),,
              title: Text('Inbox',
                style: TextStyle(color: Colors.blue, fontSize: 20.0),),
              subtitle: Text('This is my sub title'),
              trailing: Icon(Icons.arrow_right),
              onTap: () => print('1'),
            ),
          ],
        ),
      ),
    );
  }
}
```

) ,

.....  
.....  
.....

13- Now, run your app and you should get the following run output:



14- Add the following Dart files to the **lib** folder :

**mail\_interface.dart**, **profile\_interface.dart** , and **sent\_items\_interface.dart**.

15- Open **main.dart** file and add the navigation routes for the previous app interfaces (**mail\_interface.dart**, **profile\_interface.dart**, and **sent\_items\_interface.dart**).

The **main.dart** code will be as follows:

```
import 'home.dart';
import 'package:flutter/material.dart';
import 'mail_interface.dart';
import 'sent_items_interface.dart';
import 'profile_interface.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Home(),
      routes: {
        'mail': (context) => Mail(),
        'sentItem': (context) => SentItems(),
        'profile': (context) => Profile(),
      },
    );
  }
}
```

**Important note:** The names of the classes `Mail()`, `SentItems()`, and `Profile()` are not created yet, therefore, you will get an error. After you create these classes, everything will be fine.

16- Open `home.dart` file, and add a navigation route to each drawer item in the `ListTile` widget as illustrated in the grey highlighted part of the `home.dart`

```
.....
.....
.....

ListTile(
  leading: Icon(Icons.mail,
  color: Colors.blue,),
  title: Text('Inbox',
```

```

        style: TextStyle(color: Colors.blue, fontSize: 20.0),),
subtitle: Text('This is my sub title'),
trailing: Icon(Icons.arrow_right),
onTap: () => Navigator.pushNamed(context, 'mail'),
),

ListTile(
  leading: Icon(Icons.send,
    color: Colors.blue,),
  title: Text('Sent Items',
    style: TextStyle(color: Colors.blue, fontSize: 20.0),),
  subtitle: Text('This is my sub title'),
  trailing: Icon(Icons.arrow_right),
  onTap: () => Navigator.pushNamed(context, 'sentItem'),
),

ListTile(
  leading: Icon(Icons.person,
    color: Colors.blue,),
  title: Text('Profile',
    style: TextStyle(color: Colors.blue, fontSize: 20.0),),
  subtitle: Text('This is my sub title'),
  trailing: Icon(Icons.arrow_right),
  onTap: () => Navigator.pushNamed(context, 'profile'),
),

.....
.....

```

17- Copy all the contents of the **home.dart** file and paste them in the following Dart files:

**mail\_interface.dart**, **profile\_interface.dart** and **sent\_items\_interface.dart**

18- Open the **mail\_interface.dart** file, and complete the following tasks:

- Replace the class name **Home** with **Mail**
- Replace the **AppBar** title value : **Drawer Widget** with **Mail Page**
- Replace the **body** widget Text value : **Welcome** with **Mail Page**

19- Open the **sent\_items\_interface.dart** file and complete the following tasks:

- Replace the class name **Home** with **SentItems**
- Replace the **AppBar** title value : **Drawer Widget** with **Sent Items**
- Replace the **body** widget Text value : **Welcome** with **Sent Items**

20- Open the **profile\_interface.dart** file, and complete the following tasks:

- Replace the class name **Home** with **Profile**
- Replace the **AppBar** title value : **Drawer Widget** with **User Profile**
- Replace the **body** widget Text value : **Welcome** with **User Profile**

21- Run your app now and test your app navigation using your app drawer items.

## DataTable Widget

Data tables display information in a grid-like format of rows and columns. They organize information in a way that's easy to scan so that users can look for patterns and insights.

Data tables can contain:

- Interactive components (such as chips, buttons, or menus)
- Non-interactive elements (such as badges)
- Tools to query and manipulate data

Working with **DataTable()** widget is similar to configuring a table tag in HTML or in creating a table in Microsoft Word where you should add columns, rows, and configure the cells contents.

The following figure displays the table which can be added to your app interface and its components:

Car Make	Model	Price
Honda	2010	5000
Honda	2011	6000
Honda	2012	7000

```

DataColumn(label: Text('Car Make')),  

DataColumn(label: Text('Model')),  

DataColumn(label: Text('Price')),  

DataCell(Text('Honda')),  

DataCell(Text('2010')),  

DataCell(Text('5000')),  

DataCell(Text('Honda')),  

DataCell(Text('2011')),  

DataCell(Text('6000')),  

DataCell(Text('Honda')),  

DataCell(Text('2012')),  

DataCell(Text('7000'))

```

**Example:**

In this example, you will create a table using **DataTable** widget.

Perform the following steps:

1- Open **Android Studio**

2- Click **File** → **New** → **New Flutter Project**

3- Select **Flutter Application** then click **Next**.

4- Type : **data\_table\_widget** for Project Name and use **Lesson\_07** for the Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain then click **Finish**

6- Open **main.dart** file and delete all codes, then type the following code:

```

import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Data Table Widget'),
        ),
      ),
    );
  }
}

```

```
body: Center(
    child: Container(
        child: DataTable(columns: [
            DataColumn(label: Text('Car Make')),
            DataColumn(label: Text('Model')),
            DataColumn(label: Text('Price')),
        ]),
        rows: [
            DataRow(
                cells: [
                    DataCell(Text('Honda')),
                    DataCell(Text('2010')),
                    DataCell(Text('5000')),
                ],
            ),
            DataRow(
                cells: [
                    DataCell(Text('Honda')),
                    DataCell(Text('2011')),
                    DataCell(Text('6000')),
                ],
            ),
            DataRow(
                cells: [
                    DataCell(Text('Honda')),
                    DataCell(Text('2012')),
                    DataCell(Text('7000')),
                ],
            ),
        ],
    )));
}
```

7- Run your app. You will get the following run output:



## SelectableText Widget

This widget is important if you want to make your app text selectable. You may add this widget to your app text content if you think that your app user may want to copy any part of your app text to use it another app such as Google translator or Google search.

### Example:

In this example, you will create a small and simple Flutter app including a paragraph. Applying the **SelectableText** widget to these texts, your app users can use text tools such as the copy option.

Perform the following steps:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** then click **Next**.

4- Type : **selectable\_text\_widget** for Project Name and **Lesson\_07** for the Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain then click **Finish**

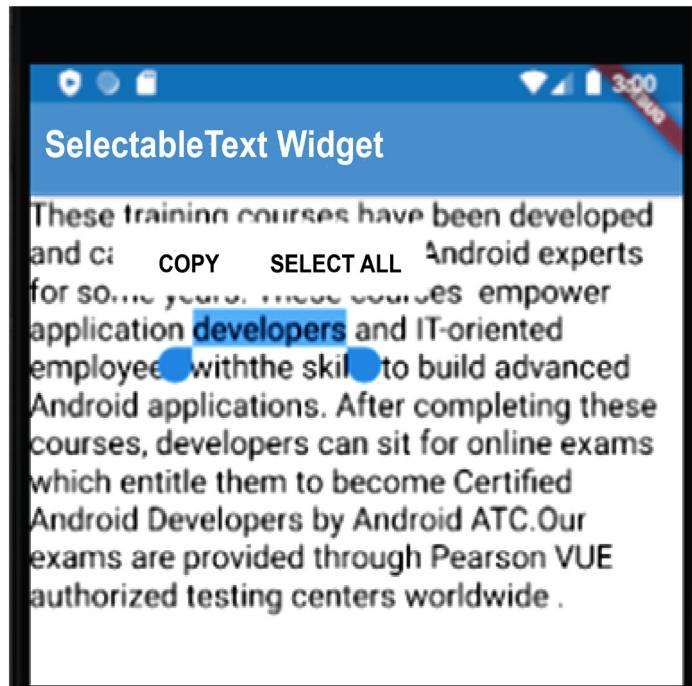
6- Open the **main.dart** file, delete all the file contents, then type the following code:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('SelectableText Widget'),
        ),
        body: SelectableText(
          'These training courses have been developed and calibrated by a team of Android experts for some years. These courses empower application developers and IT-oriented employees with the skills to build advanced Android applications. After completing these courses, developer scan sit for online exams which entitle them to become Certified Android Developers by Android ATC. Our exams are provided through Pearson VUE authorized testing centers worldwide .',
          style: TextStyle(fontSize: 20.0),
        ),
      ),
    );
  }
}
```

7- Run your app, select part of the run result and you will see the copy option as illustrated in the following figure.



8- Also, you can add a cursor to your text when the app user taps on the app text. This can be done by adding the grey highlighted part of code as properties to the **SelectableText** widget as follows:

```
SelectableText(
```

```
    'These training courses have been developed and calibrated by  
    a team of Android experts for some years. These courses empower  
    application developers and IT-oriented employees with the skills to  
    build advanced Android applications. After completing these courses,  
    developers can sit for online exams which entitle them to become  
    Certified Android Developers by Android ATC. Our exams are provided  
    through Pearson VUE authorized testing centers worldwide .',  
    style: TextStyle(fontSize: 20.0),  
    showCursor: true,  
    cursorColor: Colors.blue,  
    cursorWidth: 20,  
,
```

9- Run your app again. You will realize that when you click on your app text, you will get a blue flashing cursor.

## Stack Widget

It is a widget that places its children widgets relative to the edges of its box. This widget (class) is useful if you want to overlap several children widgets in a simple way. Stack allows you to overlay multiple widgets on top of each other. These widgets will appear like layers and you can control the position of these children widgets using Positioned widget.

To know how Stake widget helps in organizing your interface content, check the following example:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** then click **Next**.

4- Type : **stack\_widget** for Project Name and **Lesson\_07** for the project Location.  
Click **Next**.

5- Type : **androidatc.com** for Company domain then click **Finish**

6- Open main.dart file, delete all codes and then type the following code:

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

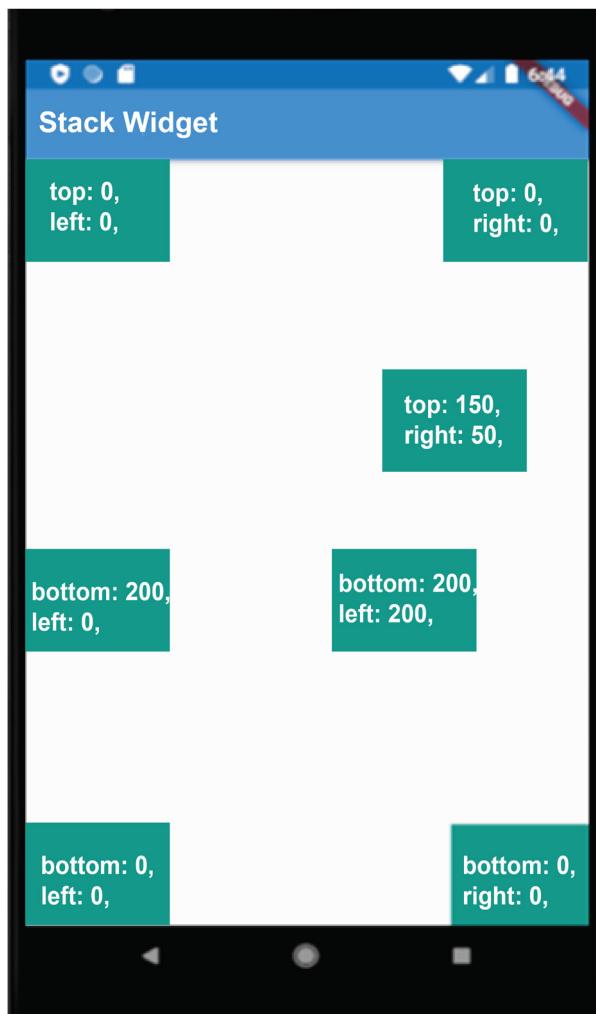
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Stack Widget'),
        ),
        body: Stack(children: <Widget>[
          Positioned(
            bottom: -20,
            right: 0,
            child: Container(

```

```
        color: Colors.teal,  
        width: 100.0,  
        height: 100.0,  
      ),  
    ),  
  ],  
,  
);  
}  
}
```

7- You should know more about the Positioned widget and its properties (right, top, bottom, and left). The following figure displays different positions of the previous Container widget when you change the properties values of the **Positioned** widget.



8- The children widgets of Stack widgets work as layers when they are overlapped in the position.

In the following code, you will add another Container widget which has a yellow color and overlaps in its position with the previous Container widget. The code follows:

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

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

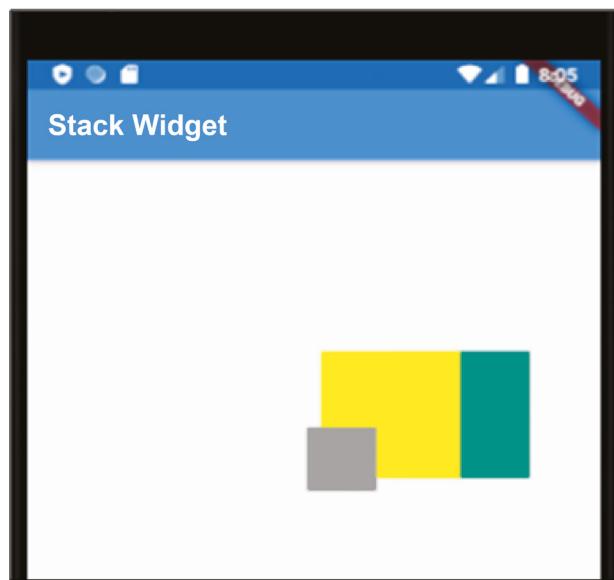
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Stack Widget'),
        ),
        body: Stack(children: <Widget>[
          Positioned(
            top: 150,
            right: 50,
            child: Container(
              color: Colors.teal,
              width: 100.0,
              height: 100.0,
            ),
          ),
          Positioned(
            top: 150,
            right: 100,
            child: Container(
              color: Colors.yellow,
              width: 100.0,
              height: 100.0,
            ),
          ),
        ]),
    );
  }
}
```

9- Run your app and note that the last child **Container** widget (layer) overrides the previous **Container** widget (layer) .

10- Add another child grey widget to your code as follows:

```
Positioned(  
    top: 210,  
    right: 160,  
    child: Container(  
        color: Colors.grey,  
        width: 50.0,  
        height: 50.0,  
    ),  
,  
,
```

The run output follows:



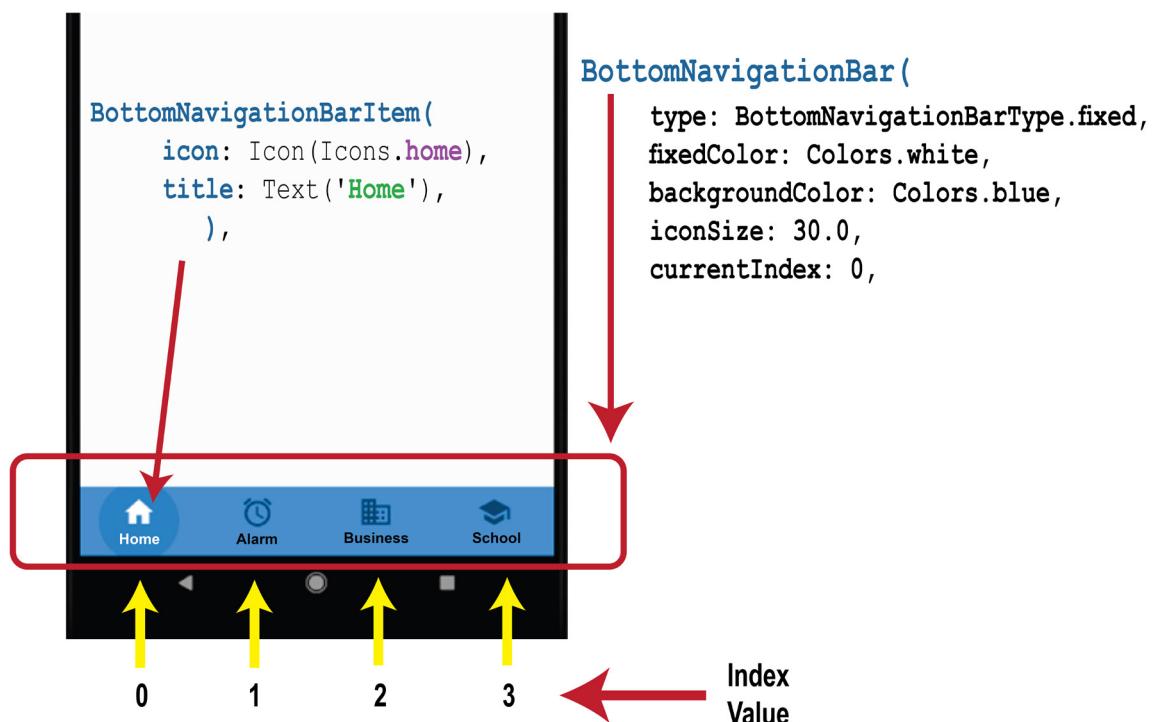
## Lab 7:

This lab includes two projects as follows:

- A. Create a Flutter App using **BottomNavigationBar** Navigation Technique
- B. Using **DataTable** Sorting Built in function

## Lab A: Create a Flutter App using BottomNavigationBar Navigation Technique

In this lab, you will create a Flutter app using the bottom navigation bar to navigate your app screens. The following figure displays the final run of your app. As you see here, and as we mentioned in this lesson, this bottom bar includes four icons and each icon has an index value. You will create four Dart files (screen1.dart , screen2.dart , screen3.dart, and screen4.dart). When the user taps the home icon he/she will go to screen1.dart file content as illustrated in the figure below, and the same thing applies for the other icons. When the user taps the alarm icon he/she will go to the screen2.dart and so on. This design is similar to using frame navigation technique in HTML. This navigation bottom bar will always be available to the app user even though the he/she moves from a screen (app interface) to another.



To create this Flutter project, perform the following steps:

- 1- Open **Android Studio**
- 2- Click **File → New → New Flutter Project**
- 3- Select **Flutter Application** then click **Next**.
- 4- Type : **bottom\_navigator\_bar\_lab** for Project Name and create a new folder : **Lab\_A\_7** for Project Location. Click **Next**.

- 5- Type : **androidatc.com** for Company domain then click **Finish**
- 6- Let us prepare the Flutter interfaces first by right clicking the **lib** folder, then → **New** → **Dart File**
- 7- Type **screen1** then click **OK**
- 8- Repeat step 6 to create other Dart files : **screen2.dart** , **screen3.dart**, and **screen4.dart**.
- 9- Open **screen1.dart** and add to it a text widget. Type the following code:

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

class Screen1 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(child: Text('Home'));
  }
}
```

- 10- Open **screen2.dart** and type the following code (you can copy the content of screen1.dart and paste it in screen2.dart, change the class name to Screen 2 and change the **Text** widget value to Alarm):

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

class Screen2 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(child: Text('Alarm'));
  }
}
```

- 11- Open **screen3.dart** and type the following code:

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

class Screen3 extends StatelessWidget {
  @override
```

```
Widget build(BuildContext context) {  
    return Container(child: Text('Business'));  
}  
}
```

12- Open **screen4.dart** and type the following code:

```
import 'package:flutter/material.dart';  
  
class Screen4 extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return Container(child: Text('School'));  
    }  
}
```

13- Open **main.dart** file, delete all code, then type the following code:

```
import 'package:flutter/material.dart';  
import 'screen1.dart';  
import 'screen2.dart';  
import 'screen3.dart';  
import 'screen4.dart';  
  
main() {  
    runApp(  
        MyApp(),  
    );  
}  
  
class MyApp extends StatefulWidget {  
    @override  
    _MyAppState createState() => _MyAppState();  
}  
  
class _MyAppState extends State<MyApp> {
```

```
final tabs = [
    Center(child: Screen1()),
    Center(child: Screen2()),
    Center(child: Screen3()),
    Center(child: Screen4()),
];
int _currentindex = 0;

@Override
Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
                title: Text(' Bottom Navigation Bar '),
            ),
            body: tabs[_currentindex],
            bottomNavigationBar: BottomNavigationBar(
                type: BottomNavigationBarType.fixed,
                backgroundColor: Colors.blue,
                selectedItemColor: Colors.white,
                selectedFontSize: 19.0,
                unselectedFontSize: 14.0,
                iconSize: 30.0,
                currentIndex: _currentindex,
                items: [
                    BottomNavigationBarItem(
                        icon: Icon(Icons.home),
                        title: Text('Home'),
                    ),
                    BottomNavigationBarItem(
                        icon: Icon(Icons.access_alarm),
                        title: Text('Alarm'),
                    ),
                    BottomNavigationBarItem(
                        icon: Icon(Icons.business),
                        title: Text('Business'),
                    ),
                ],
            ),
        ),
    );
}
```

```
        ),  
  
        BottomNavigationBarItem(  
            icon: Icon(Icons.school),  
            title: Text('School'),  
        ),  
    ],  
  
    onTap: (index) {  
        setState(() {  
            _currentindex = index;  
        });  
    },  
),  
),  
);  
}  
}
```

14- Final step: **Run** your app and test your bottom navigation buttons.

## Lab B: Using DataTable Sorting Built-in function

In this lab, you will use DataTable widget to add data to the app interface and sort the data as ascending or descending using the sort function.

Perform the following steps:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** then click **Next**.

4- Type : **data\_table\_lab** for Project Name and create a new folder : **LAB\_B\_7** for Project Location. Click **Next**.

5- Type : **androidatc.com** for Company domain then click **Finish**

6- Open **main.dart** file, delete all codes, then type the following code:

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

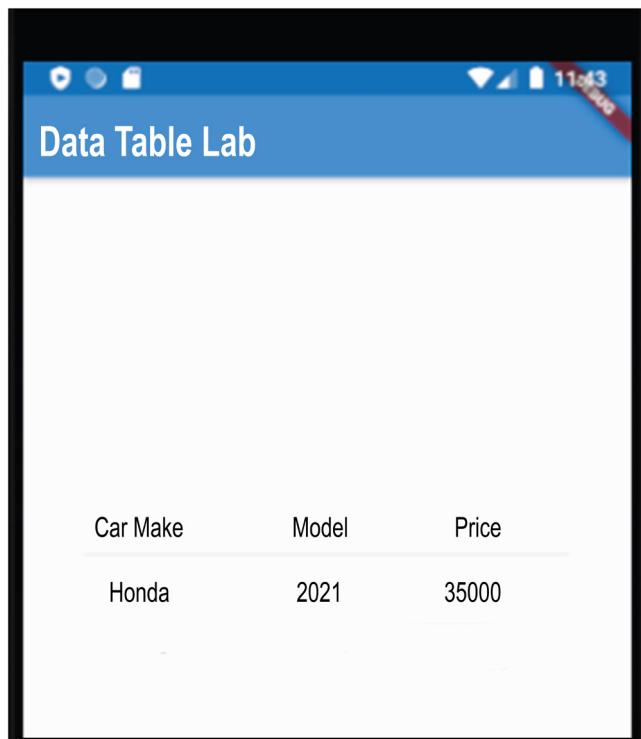
void main() => runApp(MyApp());

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

class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Data Table Lab'),
        ),
        body: Center(
          child: Container(
            child: DataTable(
              columns: <DataColumn>[
```

```
        DataColumn(  
            label: Text('Car Make'),  
        ),  
  
        DataColumn(  
            label: Text("Model"),  
        ),  
  
        DataColumn(  
            label: Text('Price'),  
        ),  
  
    ],  
  
rows: <DataRow>[  
    DataRow(cells: [  
        DataCell(Text('Honda')),  
        DataCell(Text('2021')),  
        DataCell(Text('35000'))  
    ])  
,  
,  
,  
,  
);  
}  
}
```

7- Run your app. Here, you should get the following run output:



8- Now, you will change the table configuration to add the sort feature to the any column in this table. Add the following properties to your **DataTable** widget

```
sortColumnIndex: 2,  
sortAscending: false,
```

In your table, each column has an index number. The first column from the left side has index value : **0**, and the second column has index value : **1**, and so on.

When you added **sortColumnIndex: 2**, to your **TableData** widget, this means that you want to make the sorting based on the third column which is the price.

While the property **sortAscending: false** means the sorting type is descending (A → Z).

9- Add the following two rows to your table :

Toyota	2011	6000
BMW	2010	9000

The full code until this step is the following:

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

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

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

class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Data Table Lab'),
        ),
        body: Center(
          child: Container(
            child: DataTable(
              sortColumnIndex: 2,
              sortAscending: false,
              columns: <DataColumn>[
                DataColumn(
                  label: Text('Car Make'),
                ),
                DataColumn(
                  label: Text("Model"),
                ),
                DataColumn(
                  label: Text('Price'),
                ),
              ],
              rows: <DataRow>[
```

```
        DataRow(cells: [
            DataCell(Text('Honda')),
            DataCell(Text('2021')),
            DataCell(Text('35000'))
        ]),
        DataRow(cells: [
            DataCell(Text('Toyota')),
            DataCell(Text('2011')),
            DataCell(Text('6000'))
        ]),
        DataRow(cells: [
            DataCell(Text('BMW')),
            DataCell(Text('2010')),
            DataCell(Text('9000'))
        ]),
    ],
),
),
),
),
),
),
);
}
}
```

10- Run your app again. If you tap the arrow symbol on the **price** column of your table, you will NOT have any response, because the sorting process is designed to move up or down all the row data which is related to the price column data.

To add a sorting feature to your table, you should make your table data dynamic. This means that you should create a function that has a generation table data role. Then, configure how this data should be arranged in the table cells depending on the sorting option that will be taken whether ascending or descending, or if the sorting will be applied to which column of the table. In other words, we can say that when the app user clicks the sorting arrow, this app user will call the function which will generate the table data to work and distribute this data in a specific way depending on the sorting method

(ascending or descending). To add this, follow the next steps:

11- Add the following code to create a **Car** class and **carData** variable which represents the table data. Add the class and the variable at the end of your code (after the last code bracket):

```
class Car {
    String CarMake;
    String Model;
    double Price;

    Car({this.CarMake, this.Model, this.Price});}

var carData = <Car>[
    Car(CarMake: "Honda", Model: "2021", Price: 35000.0),
    Car(CarMake: "Toyota", Model: "2011", Price: 6000.0),
    Car(CarMake: "BMW", Model: "2010", Price: 9000.0),
];
```

12- Now, if you want to apply sorting on the price column of this table, you should add the following code to **DataColumn** widget for the price column as follows:

```
DataColumn(
    label: Text('Price'),
    numeric: false,
    onSort: (i, b) {
        setState(() {
            carData.sort((a, b) => a.Price.compareTo(b.Price));

        }),
    );
},
```

### Explanation:

When you sort an array with `.sort()`, this assumes that you are sorting strings. When sorting numbers, the default behavior will not sort them properly.

The function that you pass tells how to sort the elements. It takes two parameters (`a` and `b`) that represents any two elements from the array (`carData`). The elements will be sorted depending on the function's return value `a.Price.compareTo(b.Price)`; as follows:

- If it returns a negative value, the value in `a` will be arranged before `b`.
- If it returns 0, the ordering of `a` and `b` will not change.
- If it returns a positive value, the value in `b` will be arranged before `a`.

The `setState` method notifies the framework that the internal state of this column has changed in a way that might impact the user interface in this subtree, which causes the framework to schedule a build for this `State` object.

13- It is a good idea to apply this sorting process to all other columns using the same previous code with just the column name being changed. The full code until this step is the following:

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

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

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

class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Data Table Lab'),
        ),
      ),
    );
}
```

```
body: Center(
  child: Container(
    child: DataTable(
      sortColumnIndex: 2,
      sortAscending: false,
      columns: <DataColumn>[
        DataColumn(
          label: Text('Car Make'),
          numeric: false,
          onSort: (i, b) {
            setState(() {
              carData.sort((a, b) => a.CarMake.compareTo(b.CarMake));
            },
            );
          },
        ),
        DataColumn(
          label: Text("Model"),
          numeric: false,
          onSort: (i, b) {
            setState(() {
              carData.sort((a, b) => a.Model.compareTo(b.Model));
            },
            );
          },
        ),
        DataColumn(
          label: Text('Price'),
          numeric: true,
          onSort: (i, b) {
            setState(() {
              carData.sort((a, b) => a.Price.compareTo(b.Price));
            },
            );
          },
        ),
      ],
    ),
  ),
);
```

```
rows: <DataRow>[  
    DataRow(cells: [  
        DataCell(Text('Honda')),  
        DataCell(Text('2021')),  
        DataCell(Text('35000'))  
    ]),  
  
    DataRow(cells: [  
        DataCell(Text('Toyota')),  
        DataCell(Text('2011')),  
        DataCell(Text('6000'))  
    ]),  
  
    DataRow(cells: [  
        DataCell(Text('BMW')),  
        DataCell(Text('2010')),  
        DataCell(Text('9000'))  
    ]),  
],  
),  
),  
),  
),  
);  
}  
}  
  
class Car {  
    String CarMake;  
    String Model;  
    double Price;  
  
    Car({this.CarMake, this.Model, this.Price});  
}  
  
var carData = <Car>[  
    Car(CarMake: "Honda", Model: "2021", Price: 35000.0),  
    Car(CarMake: "Toyota", Model: "2011", Price: 6000.0),  
    Car(CarMake: "BMW", Model: "2010", Price: 9000.0),  
];
```

14- Now, working on configuring the table data, you don't need to add this table data manually. Remember, this table is a dynamic table. This means you should configure the **DataCell()** class to represent the variables data table. These variables values will be filled out from the **carData** list.

The following code will give you a full idea:

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

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

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

class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Data Table Lab'),
        ),
      ),
      body: Center(
        child: Container(
          child: DataTable(
            sortColumnIndex: 2,
            sortAscending: false,
            columns: <DataColumn>[
              DataColumn(
                label: Text('Car Make'),
                numeric: false,
                onSort: (i, b) {
                  setState(() {
                    carData.sort((a, b) => a.CarMake.compareTo(b.CarMake));
                  },
                );
              );
            ],
            rows: <DataRow>[
              DataRow(
                cells: [
                  DataCell(Text('Audi')),
                  DataCell(Text('A4')),
                  DataCell(Text('2018')),
                ],
              ),
              DataRow(
                cells: [
                  DataCell(Text('BMW')),
                  DataCell(Text('X5')),
                  DataCell(Text('2019')),
                ],
              ),
              DataRow(
                cells: [
                  DataCell(Text('Ferrari')),
                  DataCell(Text('488')),
                  DataCell(Text('2020')),
                ],
              ),
              DataRow(
                cells: [
                  DataCell(Text('Lamborghini')),
                  DataCell(Text('Aventador')),
                  DataCell(Text('2017')),
                ],
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

```
        },
    ),

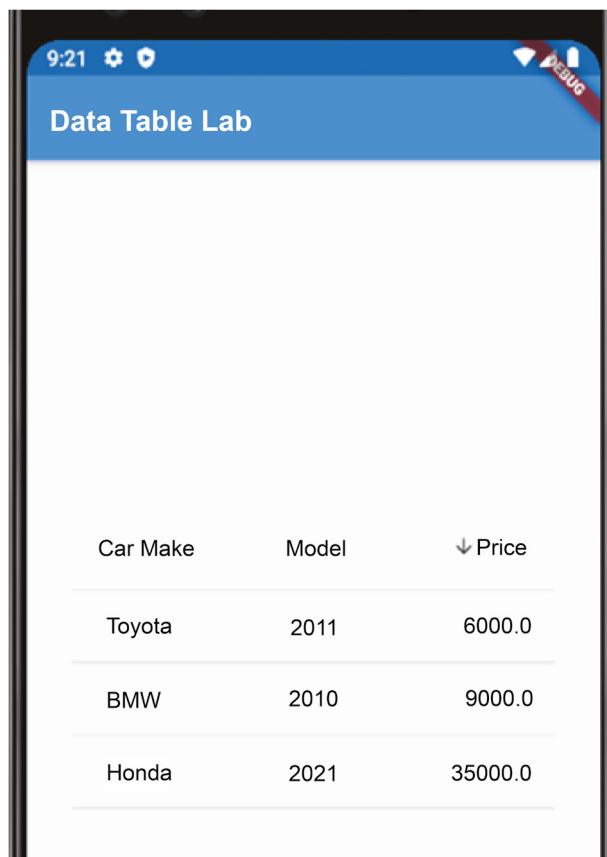
    DataColumn(
        label: Text("Model"),
        numeric: false,
        onSort: (i, b) {
            setState(() {
                carData.sort((a, b) => a.Model.compareTo(b.Model));
            },
            );
        },
    ),
),

DataColumn(
    label: Text('Price'),
    numeric: true,
    onSort: (i, b) {
        setState(() {
            carData.sort((a, b) => a.Price.compareTo(b.Price));},
        );
    },
),
],
rows: carData
    .map(
        (x) => DataRow(
            cells: [
                DataCell(Text(x.CarMake)),
                DataCell(Text(x.Model)),
                DataCell(Text((x.Price).toString())),
            ],
        ),
    ).toList(),
),
),
),
),
);
}
}

class Car {
```

```
String CarMake;  
String Model;  
double Price;  
  
Car({this.CarMake, this.Model, this.Price});  
}  
  
var carData = <Car>[  
  Car(CarMake: "Honda", Model: "2021", Price: 35000.0),  
  Car(CarMake: "Toyota", Model: "2011", Price: 6000.0),  
  Car(CarMake: "BMW", Model: "2010", Price: 9000.0),  
];
```

15- Run your app and tap the arrow symbol. You will find that it works well as illustrated in the following figure:



16- Replace the index value in **sortColumnIndex:2** with **1**, then run your app again. You will get the sort arrow symbol added to the Model column. Test sorting for this column.

## Lesson 8: Visual, Behavioral, and Motion-Rich Widgets

### Implementing Material Design Guidelines - Part 2

#### **Input and Selections**

Text Field Widget.....	8-2
CheckboxGroup and RadioButtonGroup Widgets.....	8-11
Date Picker.....	8-17
Time Picker.....	8-23
Slider Widget.....	8-25
Switch Widget.....	8-29

#### **Dialogs, Alerts, and Panels**

Alert Dialog Widget.....	8-32
Cupertino Alert Dialog Widget.....	8-35
Bottom Sheet.....	8-36
Modal Bottom Sheet.....	8-36
Persistent Bottom Sheet.....	8-41
Expansion Panel Widget.....	8-49
Snack Bar Widget.....	8-54
<b>Lab 8: Creating a Hotel Reservation App.....</b>	<b>8-60</b>

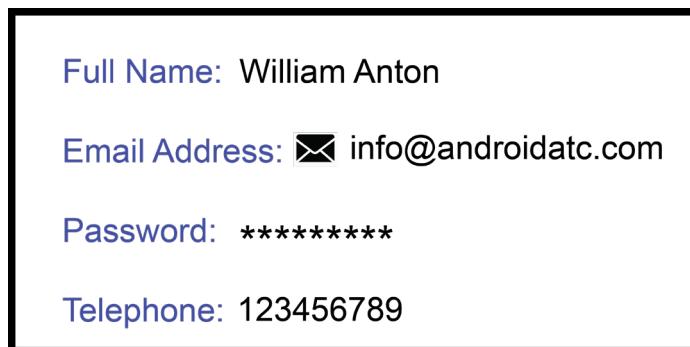
## Text Field Widget

Text fields allow users to type text into an app. They are used to build forms, send messages, create search experiences, and more.

**TextField** widget is the most commonly used text input widget. By default, a **TextField** is decorated with an underline. You can add a label, icon, and inline hint text by supplying an **InputDecoration** as the decoration property of the **TextField**.

### Example:

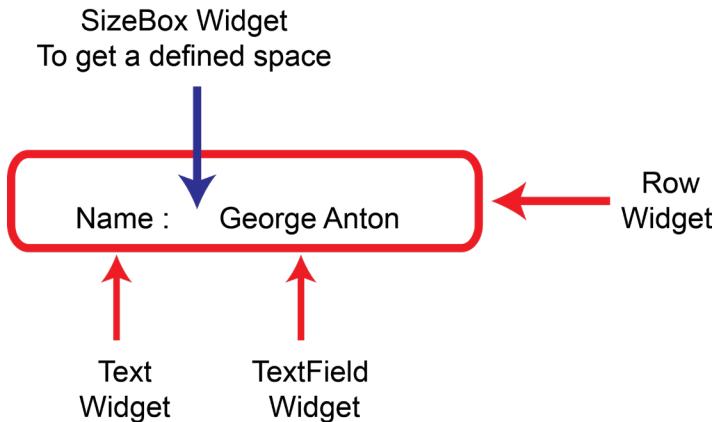
As illustrated in the following figure, you will create a small Flutter app including **TextField** widgets , and you will apply different properties for each **TextField** widget.



The above text fields are usually used in a form or in a login area of a Flutter app. You should focus on the format of these fields ensuring that they will start at the same left margin and set a space between the name of the field (such as Full Name:) and the **TextField** which the user will use to enter his/her information.

It is a good idea to add your form fields inside a **ListView** widget and use its property: **shrinkWrap** with value: **true** (**shrinkWrap: true**). This configuration will guarantee that your app interface will display all your form contents and your app user can easily scroll down to see all your app interface content.

Also, the best way to fully control designing the content of this form fields such as the location of each field and the space between the field title and the field itself, is to arrange each form line in a **Row** widget, each **TextField** inside a **SizeBox** widget and add a **SizeBox** widget between the field title (such as Full Name:) and the **TextField** widget to get a specific space. The following figure gives you an idea about arranging form widgets using Row and **SizeBox** widgets.



In addition, configuring each **TextField** widget as a child widget of the **SizeBox** widget gives you control on the width of the **TextField** widget as you will see in the next example.

To create this simple form, perform the following steps:

1- Open **Android Studio**

2- Click **File** → **New** → **New Flutter Project**

3- Select **Flutter Application**, and then click **Next**

4- Type: **form\_items** for Project Name and create a new folder: **Lesson\_08** for Project Location. Click **Next**

5- Type: **androidatc.com** for Company domain and then click **Finish**

6- Open **main.dart** file, delete all the code, and type the following code:

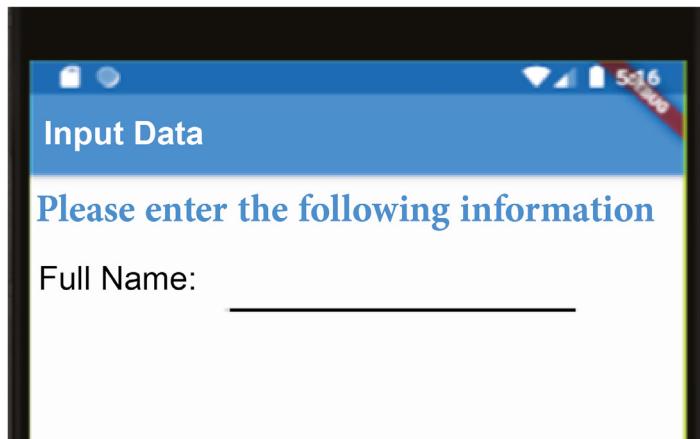
```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Input Data'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Text('Name :'),
              TextField(),
            ],
          ),
        ),
      ),
    );
  }
}
```

```
    ),  
  
    body: Padding(  
        padding: const EdgeInsets.all(8.0),  
        child: Stack(  
            children: <Widget>[  
                ListView(  
                    shrinkWrap: true,  
                    children: <Widget>[  
                        Text(  
                            'Please enter the following information',  
                            style: TextStyle(  
                                fontSize: 22.0,  
                                color: Colors.blue,  
                                fontWeight: FontWeight.bold),  
                        ),  
                        SizedBox(height: 10.0,),  
  
                        Row(  
                            children: [  
                                Text('Full Name:',  
                                    style: TextStyle(fontSize: 20.0),),  
  
                                SizedBox(width: 20.0,),  
  
                                SizedBox(width: 220.0,  
                                    child: TextField(),  
                                ),  
                            ],  
                        ),  
                        ],  
                    ),  
                ),  
            ],  
        );  
    }  
}
```

7- Run your app. The run output follows:



As you can see here in your app run output, the **TextField** has been represented by an underline and its width is the width of the **SizeBox** widget which is its parent widget.

8- Before adding any property to the **TextField** widget, check first the following table which includes almost of these widget properties:

Property	Description
<b>keyboardType:</b> <code>TextInputType.number,</code>	The device keys include only numbers. This means the user can enter numbers only in this field.
<b>textInputAction:</b> <code>TextInputAction.done,</code>	After filling out the form information, the Enter or Return button which will be used in your smart device keys to send or submit this data will be <b>Done</b> for iOS devices and <b>check mark</b> for Android devices. Also, Instead of done you may use search or send.
<b>maxLength: 25,</b>	You cannot type or enter more than 25 characters.
<b>obscureText: true,</b>	The characters in this field will appear as stars. This property is suitable if this field is assigned to enter a password.
<b>autocorrect: true,</b>	If this property value is true, the app user will get suggestions when he/she start typing any data in this field.
<b>cursorColor: Colors.red,</b>	You can determine the cursor color .

## InputDecoration Widget:

When you add this widget to the **TextField** widget, you can control the format of your **TextField** widget. The following table includes properties you may add to this widget:

Property	Description
<code>icon: Icon(Icons.email),</code>	By adding this property to the <b>InputDecoration</b> widget you can add an icon inside the typing area of the <b>TextField</b> ..
<code>hintText: 'Enter your name',</code>	This configuration gives the app user a hint about the format or data he/she should enter.
<code>border: InputBorder.none,</code>	When the input border value is none, this will hide the default underline for the <b>TextField</b> widget.
<code>focusedBorder: OutlineInputBorder( borderRadius: BorderRadius.all(Radius.circular(7.0)), borderSide: BorderSide(color: Colors.blue), )</code>	This property will add a radius border around the <b>TextField</b> widget.

9- Now, you will configure a form similar to the following figure by adding additional **TextField** widgets and use the suitable properties for each **TextField** widget depending on the data type (password, normal text, numbers, or etc.).

Full Name: William Anton

Email Address:  info@androidatc.com

Password: \*\*\*\*\*

Telephone: 123456789

The full code is the following:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Input Data'),
        ),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: Stack(
            children: <Widget>[
              ListView(
                shrinkWrap: true,
                children: <Widget>[
                  Text('Please enter the following information',
                      style: TextStyle(
                        fontSize: 22.0,
                        color: Colors.blue,
                        fontWeight: FontWeight.bold)),
                  SizedBox(height: 10.0,),

                  Row(
                    children: [
                      Text('Full Name:',
                          style: TextStyle(fontSize: 20.0)),,
                      SizedBox(width: 20.0,),

                      child: TextField(
                        maxLength: 25,
                        style: TextStyle(fontSize: 20, color: Colors.blue),
                        keyboardType: TextInputType.text,
                       textInputAction: TextInputAction.done,
                        autocorrect: true,

```



```
        ),
    ],
),

SizedBox(height: 10.0,),

// ***** Password *****

Row(
    children: [
        Text('Password:',
            style: TextStyle(fontSize: 20.0),),
        SizedBox(width: 20.0,),

        SizedBox(width: 220.0,
            child: TextField(
                maxLength: 50,
                obscureText: true,
                style: TextStyle(fontSize: 20, color: Colors.blue),
               textInputAction: TextInputAction.done,
                autocorrect: false,
                cursorColor: Colors.red,
                decoration: InputDecoration(
                    border: InputBorder.none,
                    focusedBorder: OutlineInputBorder(
                        borderRadius:
                            BorderRadius.all(Radius.circular(7.0)),
                    borderSide: BorderSide(color: Colors.blue),
                )));
        ),
        ],
),
),

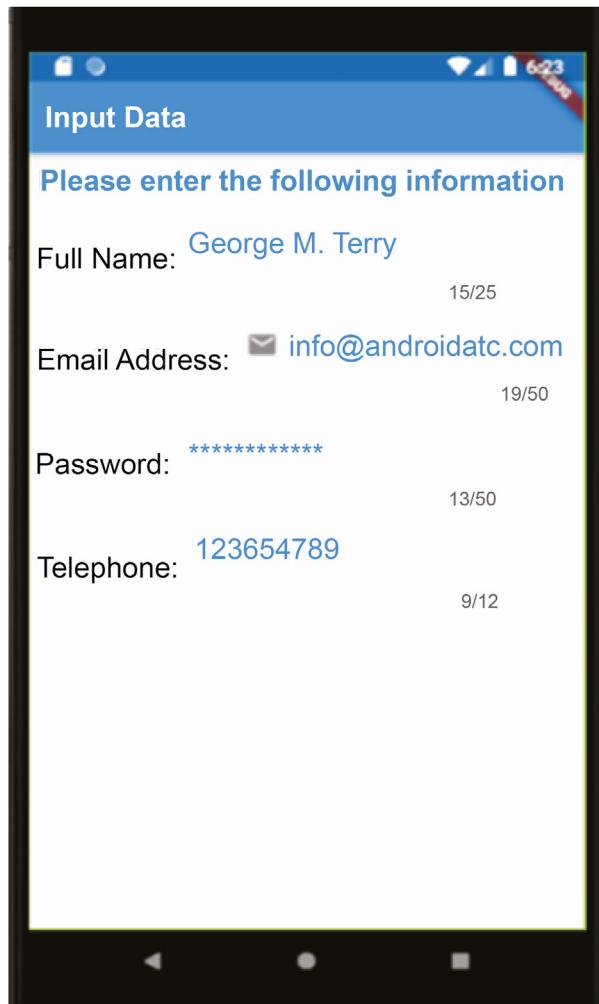
SizedBox(height: 10.0,),

// ***** Telephone *****

Row(
    children: [
        Text('Telephone:',
            style: TextStyle(fontSize: 20.0),),
```

```
        SizedBox(width: 20.0),  
  
        SizedBox(width: 220.0,  
            child: TextField(  
                maxLength: 12,  
                style: TextStyle(fontSize: 20, color: Colors.blue),  
               textInputAction: TextInputAction.done,  
                keyboardType: TextInputType.number,  
                autocorrect: false,  
                cursorColor: Colors.red,  
                decoration: InputDecoration(  
                    border: InputBorder.none,  
                    focusedBorder: OutlineInputBorder(  
                        borderRadius:  
                            BorderRadius.all(Radius.circular(7.0)),  
                        borderSide: BorderSide(color: Colors.blue),  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),  
                    );  
    }  
}
```

10- Run your app now and you will get the following run output:



You may repeat any of the previous **TextField** widgets many times in your code to be sure that there is no problem in the scrolling process.

## CheckboxGroup and RadioButtonGroup Widgets

These widgets are important to any form because they help in collecting the information exactly as you need it. We use the **CheckboxGroup** widget which includes many check boxes if the app user has an option to select more than one choice. If the app user has only one option out of the many choices, you should use **RadioButtonGroup** widget which includes many number of radio buttons related to one group, and the user can select one choice only.

For example, the following code for a **CheckboxGroup** widget includes five check boxes (5 labels):

```
CheckboxGroup(  
  
  labels: <String>[  
    "Onions",  
    "Mushrooms",  
    "Black olives",  
    "Green peppers",  
    "Extra cheese",  
  ],),
```

The following example of code represents a **RadioButtonGroup** widget which includes three radio buttons (3 labels) :

```
RadioButtonGroup(  
  labels: <String>[  
    "One-Way Ticket",  
    "Round-Trip Ticket",  
    "Multi-City Ticket",  
  ],),
```

### **Example:**

In the following example, you will create a small Flutter app including **CheckboxGroup** and **RadioButtonGroup** widgets.

1- Open **Android Studio**

2- Click **File** → **New** → **New Flutter Project**

3- Select **Flutter Application**, and then click **Next**

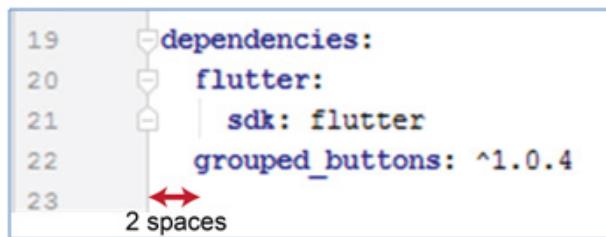
4- Type: **checkbox\_radio\_widgets** for the Project Name. Click **Next**

5- Type: **androidatc.com** for Company domain, and then click **Finish**

6- Add the following to your **pubspec.yaml** file:

```
grouped_buttons: ^1.0.4
```

Your **pubspec.yaml** file should be as follows:



```
19 dependencies:
20   flutter:
21     sdk: flutter
22     grouped_buttons: ^1.0.4
23
```

Then click **Packages get** in the **pubspec.yaml** file.

7- Open **main.dart** file, delete all the code, and type the following code:

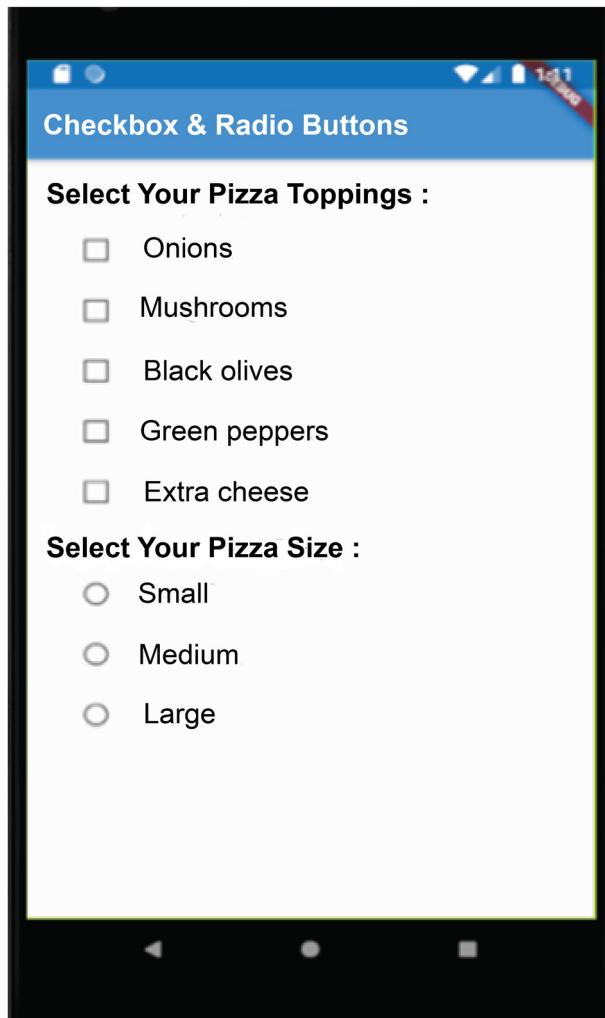
```
import 'package:flutter/material.dart';
import 'package:grouped_buttons/grouped_buttons.dart';

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

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Checkbox & Radio Buttons'),
        ),
        body: Padding(
          padding: const EdgeInsets.all(15.0),
          child: Column(
            mainAxisAlignment: MainAxisAlignment.start,
            children: <Widget>[
              SizedBox(height: 10.0,),
              Text('Select Your Pizza Toppings :',
                style: TextStyle(fontSize: 20.0, fontWeight: FontWeight.bold),
              ),
              CheckboxGroup(
                labelStyle: TextStyle(fontSize: 20.0),
                labels: <String>[
                  "Onions",
                  "Mushrooms",
                  "Black olives",
                ],
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

```
        "Green peppers",
        "Extra cheese",
    ],
onSelected: (List<String> checked) => print(checked.toString()),),
SizedBox(height: 10.0,),,
Text('Select Your Pizza Size :',
style: TextStyle(fontSize: 20.0, fontWeight: FontWeight.bold),),
RadioButtonGroup(
labelStyle: TextStyle(fontSize: 20.0),
labels: <String>[
    "Small",
    "Medium",
    "Large",
],
onSelected: (String selected) => print(selected),
),
],
),
),
),
),
),
);
}
}
```

7- Run your app and you should get the following result:



When you click your form items (check boxes and radio buttons), you will get the following result on the run console:

```
Run: main.dart ×

Console ⚡ 📺 🎨

Performing hot restart...
Syncing files to device Android SDK built for x86...
Restarted application in 1,438ms.
I/flutter (24112): [Onions]
I/flutter (24112): [Onions, Mushrooms]
I/flutter (24112): [Onions, Mushrooms, Black olives]
I/flutter (24112): [Onions, Mushrooms, Black olives, Green peppers]
I/flutter (24112): [Onions, Mushrooms, Black olives, Green peppers, Extra cheese]
I/flutter (24112): Small
I/flutter (24112): Medium
I/flutter (24112): Large
```

The following table includes the properties which you may apply with the **CheckboxGroup** or **RadioButtonGroup** widget:

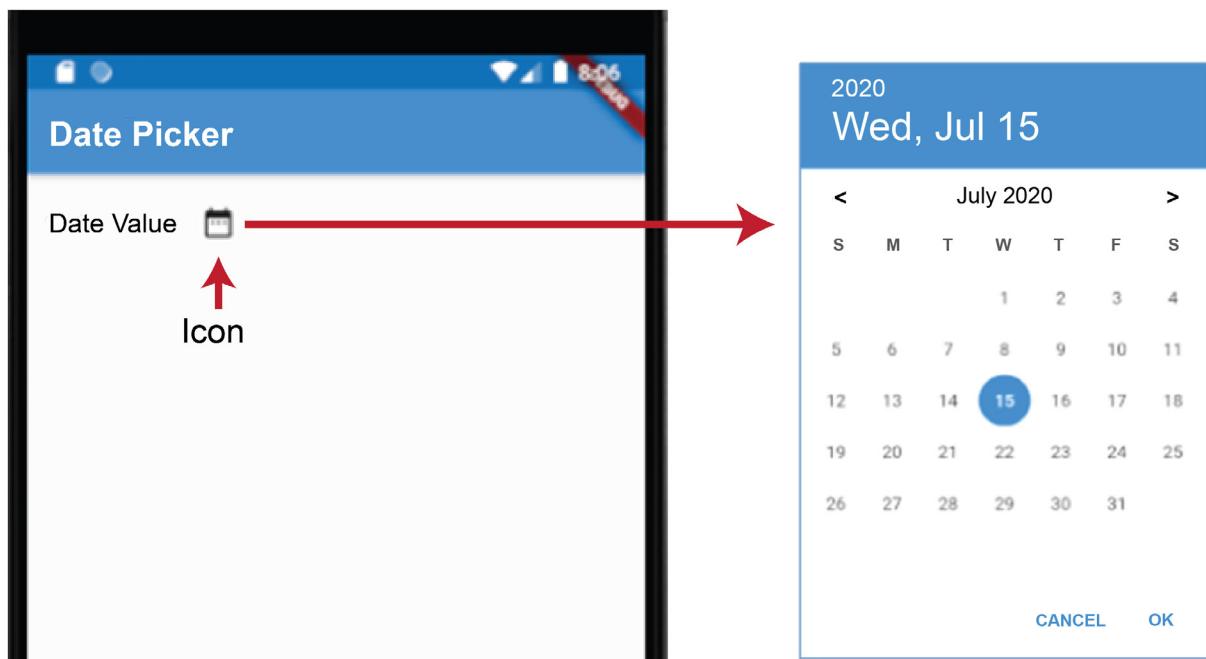
Properties	Description
<b>activeColor</b>	The color to use when a Checkbox or Radio button is checked.
<b>checkColor</b>	The color to use for the check icon when a Checkbox is checked (with <b>CheckboxGroup</b> only).
<b>checked</b>	Specifies which boxes to be automatically checked. Every element must match a label. This is useful for clearing all selections (set it to [ ] ). If this is non-null, then the user must handle updating this list, otherwise, the state of the <b>CheckboxGroup</b> won't change (with <b>CheckboxGroup</b> only).
<b>disabled</b>	Specifies which checkboxes or Radio buttons should be disabled. If this is non-null, no boxes will be disabled. The strings passed to this must match the labels.
<b>itemBuilder</b>	Called when needed to build a <b>CheckboxGroup</b> or <b>RadioButtonGroup</b> element.
<b>labels</b>	A list of strings that describes each Checkbox or Radio button. Each label must be distinct.
<b>labelStyle</b>	The style to use for the labels.
<b>margin</b>	Empty space surrounding the <b>CheckboxGroup</b> or <b>RadioButtonGroup</b> .
<b>onChange</b>	Called when the value of the <b>CheckboxGroup</b> or <b>RadioButtonGroup</b> changes.
<b>onSelected</b>	Called when the user makes a selection.
<b>orientation</b>	Specifies the orientation to display elements. Either <b>GroupedButtonsOrientation.HORIZONTAL</b> or <b>GroupedButtonsOrientation.VERTICAL</b> .
<b>padding</b>	Empty space in which to insert the <b>CheckboxGroup</b> or <b>RadioButtonGroup</b>

## Date Picker

In this topic, you will use the Date Picker plug-in to enter the date in your app interface instead of asking the app user to enter it manually. Also, you will select the date format (day, month and year).

### Example:

In the following example, you will create a small Flutter app including all **DatePicker** plug-in features to display and add the date to your app content. Your design should be similar to the following figure:



To do that, perform the following steps:

- 1- Open **Android Studio**
- 2- Click **File** → **New** → **New Flutter Project**
- 3- Select **Flutter Application** and then click **Next**
- 4- Type: **date\_picker** for Project Name, and select **Lesson\_08** for the Project Location. Click **Next**
- 5- Type: **androidatc.com** for Company domain, and then click **Finish**
- 6- Open the **main.dart** file, delete all its content, and then type the following code:

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

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

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

class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Date Picker'),
        ),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: ListView(shrinkWrap: true, children: <Widget>[
            Row(children: [
              Text('Date Value',
                style: TextStyle(fontSize: 20.0),
              ),
              IconButton(
                icon: Icon(Icons.date_range),
                onPressed: () {},
              ),
            ],
            ],
            ],
            ),
            );
  }
}
```

**Important note:** Be sure that you have configured the **main()**function as follows:

```
runApp(MaterialApp(home: MyApp()));
```

7- Add the following code to **\_MyAppState** class.

In following code, you will configure a local variable **picked** as a **DateTime** object. Also, the date picker will start from 1960 until 2030.

In addition, you will configure **date1** as a **DateTime** object and have the current date (The smart device current date).

The below **If** statement will enable the **setState** method which will notify the app framework that the internal state of **DateTime** object has been changed when the user selects a new date value. Then update the values of **date1** object.

```
DateTime date1 = DateTime.now();

Future<Null> selectDate(BuildContext context) async {
    final DateTime picked = await showDatePicker(
        context: context,
        initialDate: date1,
        firstDate: DateTime(1960),
        lastDate: DateTime(2030),
    );

    if (picked != null && picked != date1) {
        setState(() {
            date1 = picked;
        });
    }
}
```

8- Add **selectDate(context)**; to the **onPressed** function of your app **IconButton**.

The full code until this step is as follows:

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

void main() {
    runApp(MaterialApp(home: MyApp()));
```

```
}

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

class _MyAppState extends State<MyApp> {
  @override

  DateTime date1 = DateTime.now();
  Future<Null> selectDate(BuildContext context) async {
    final DateTime picked = await showDatePicker(
      context: context,
      initialDate: date1,
      firstDate: DateTime(1960),
      lastDate: DateTime(2030),
    );

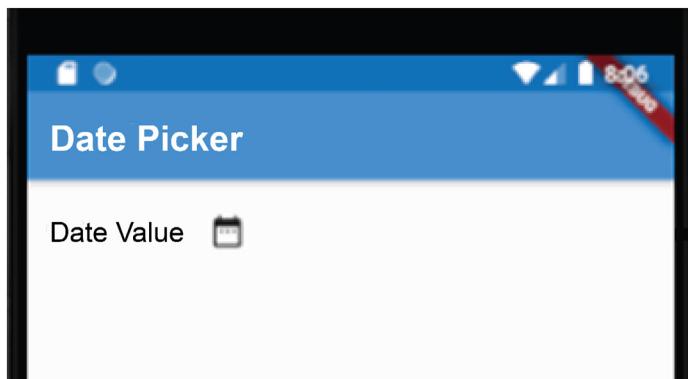
    if (picked != null && picked != date1) {
      setState(() {
        date1 = picked;
      });
    }
  }

  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Date Picker'),
        ),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: ListView(shrinkWrap: true, children: <Widget>[
            Row(children: [
              Text('Date Value', style: TextStyle(fontSize: 20.0),),

              IconButton(
                icon: Icon(Icons.date_range),
                onPressed: () {
                  selectDate(context);
                },
              ),
            ],
          ],
        ),
      ),
    );
  }
}
```

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

9-Run your app. The run output follows:

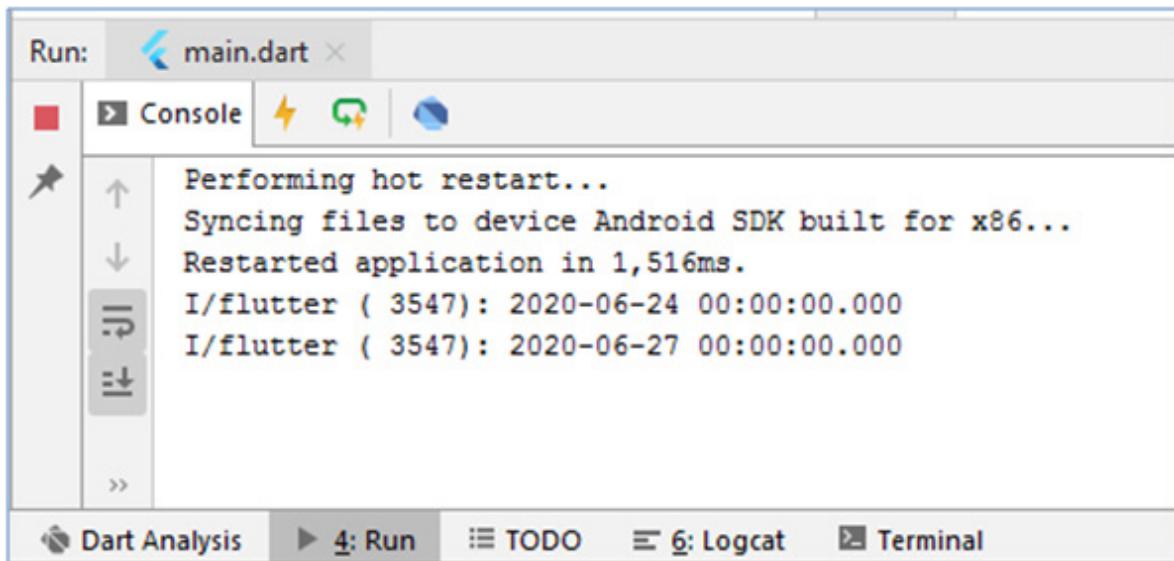


If you click the date icon, you will get the date picker dialog box. However, when you select a new date value and click **OK**, nothing will happen because you did not configure any type of the output result yet.

You may add the grey highlighted part of the below code to your app:

```
if (picked != null && picked != date1) {  
    setState(() {  
        date1 = picked;  
        print(date1.toString());  
    });  
}
```

The **print** method will print the new date value on the run console as illustrated in the following figure:



10- Now, replace the following **Text** widget :

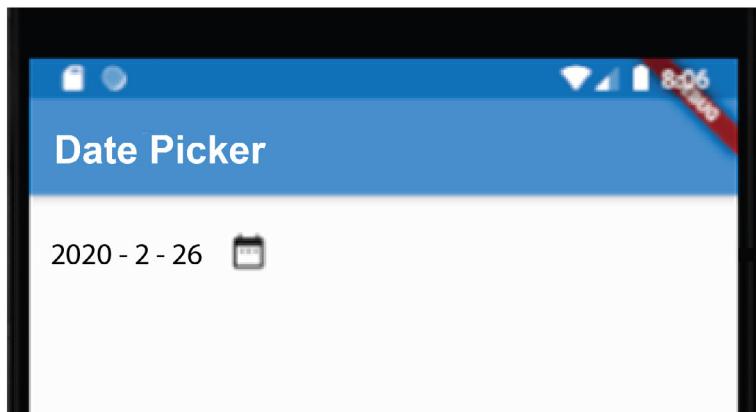
```
Text(
  'Date Value',
  style: TextStyle(fontSize: 20.0),
),
```

with the following **Text** widget:

```
Text(
  ('${date1.year} - ${date1.month} - ${date1.day}').toString(),
  style: TextStyle(fontSize: 20.0),
),
```

The date value will appear as a value of this **Text** widget with format : year - month - day format.

11- Run your app again and select a new date value of the date picker. You will get a date value on your app interface similar to the following figure:



## Time Picker

It is similar to date picker plug-in. Here, you will add an icon to ask the app user to enter the time value using the time picker plug-in instead of typing it manually.

### Example:

In this example, you will follow the same previous steps as that of creating the date picker app with some changes.

Perform the following steps:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** and then click **Next**

4- Type: **time\_picker** for Project Name and select folder: **Lesson\_08** for the Project Location. Click **Next**

5- Type: **androidatc.com** for Company domain and then click **Finish**

6- Open the **main.dart** file, delete all its content, and then type the following code:

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

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

class MyApp extends StatefulWidget {
```

```
@override
_MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
@override

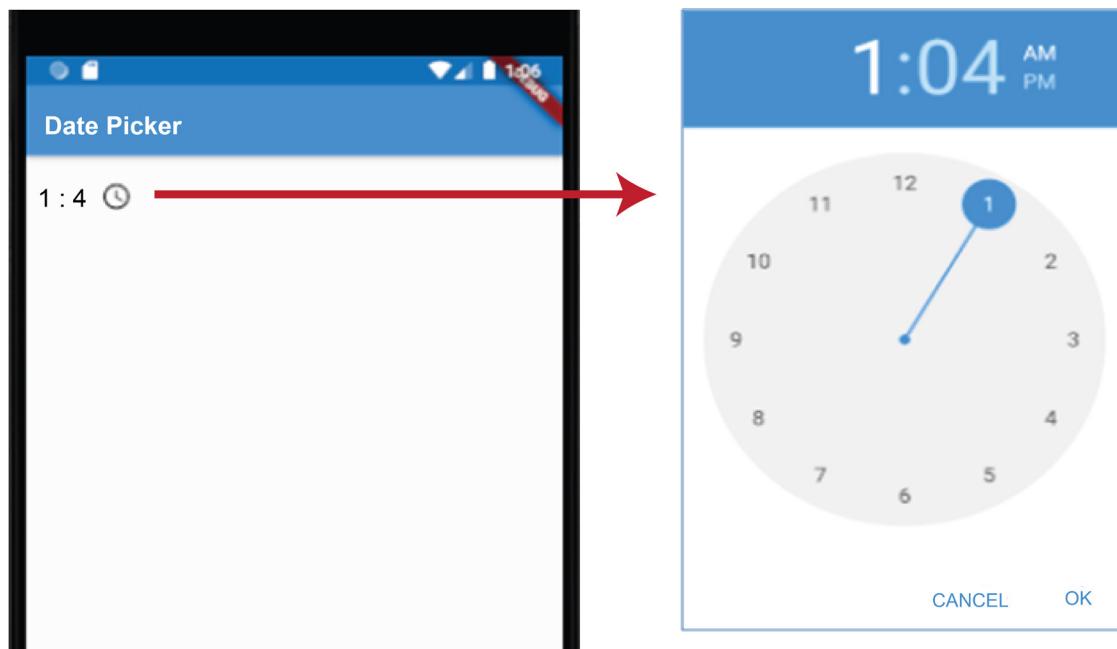
TimeOfDay time1 = TimeOfDay.now();
TimeOfDay picked;
Future<Null> selectTime(BuildContext context) async {
    picked = await showTimePicker(context: context, initialTime: time1);

    setState(() {
        time1 = picked;
        print(time1.toString());
    });
}

Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
                title: Text('Date Picker'),
            ),
            body: Padding(
                padding: const EdgeInsets.all(8.0),
                child: ListView(shrinkWrap: true, children: <Widget>[
                    Row(children: [
                        Text('${time1.hour} : ${time1.minute}'.toString(),
                            style: TextStyle(fontSize: 20.0),),

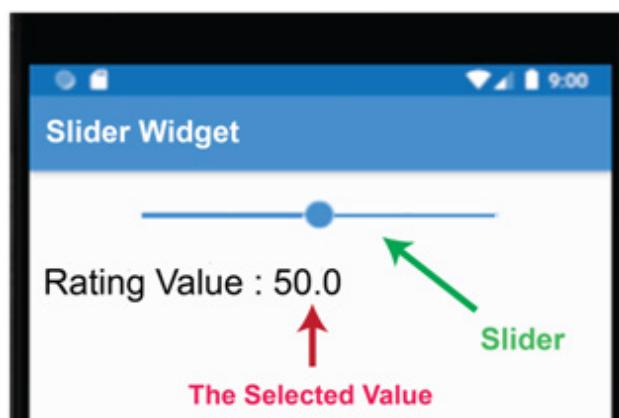
                        IconButton(
                            icon: Icon(Icons.access_time),
                            onPressed: () {
                                selectTime(context);
                            },
                        ),
                    ],
                ],
            ),
        );
    );
}
```

7- Run your app and you will get the following run result:



## Slider Widget

The Slider widget selects a single value from a range. As illustrated in the following figure, you may use Slider widget as a method to input the app user's information into the app. You may use this widget to add more user interactivity to your app such as giving app users control for customizing the app interface and changing some settings.



**Example:**

The following code includes the Slider widget and its properties:

```
var rating = 0.0;
...
...

Slider(
  value: rating,
  onChanged: (newRating) {
    setState(() => rating = newRating);
  },
  divisions: 4,
  label: "$rating",
  min: 0,
  max: 100,
),
```

This widget configuration depends on the properties which are used in the previous code. The following table includes description of each property:

Property	Description
<b>divisions</b>	Integer represents the number of discrete divisions. In the previous example, it equals to 4. This means the slider will have 4 values other than the start or the minimum value.
<b>min</b>	The minimum value the user can select.
<b>max</b>	The maximum value the user can select. Here, in this example, the maximum value is 100. It means this slider will be divided into 4 values other than the start value (min value = 0), and the user can select 0, 25, 50, 75, or 100.
<b>label</b>	String value represents a label to show the value above the slider when the slider is active.
<b>onChanged</b>	Called during a drag when the user is selecting a new value for the slider by dragging.
<b>value</b>	The currently selected value for this slider (Double data type).

**Example:**

In the following example, you will create a small Flutter app including a Slider widget. Perform the following steps:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** and then click **Next**

4- Type: **slider\_picker** for Project Name and select Lesson\_08 for the **Project Location**. Click **Next**

5- Type: **androidatc.com** for Company domain and then click **Finish**

6- Open the **main.dart** file, delete all its content, and then type the following code:

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

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

class MyApp extends StatefulWidget {
    // This widget is the root of your application.
    @override
    _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
    @override
    // double rating = 0.0;
    var rating = 0.0;

    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(
                    title: Text('Slider Widget'),
                ),
                body: Column(children: <Widget>[
                    Padding(
                        padding: const EdgeInsets.all(8.0),
                        child: Container(
                            width: 300.0,
```

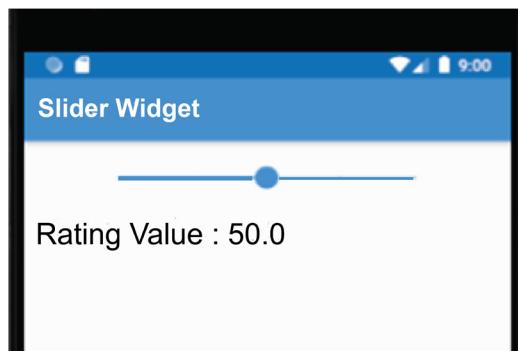
```
        child: Slider(
            value: rating,
            onChanged: (newRating) {
                setState(() => rating = newRating);
            },
            divisions: 4,
            label: "$rating",
            min: 0,
            max: 100,
        ),
    ),
),

Row(children: [
    SizedBox(width: 20.0,),

    Text(
        'Rating Value: '
        '$rating',
        style: TextStyle(fontSize: 25.0),
    ),
],
]),
]),
]);
},
});
```

**Important note:** The Slider widget container must have a suitable width to show your app slider on the app interface.

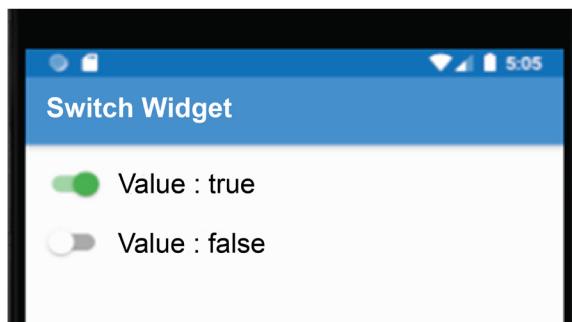
7- Run your app. You should get the following run output. Test your slider values.



## Switch Widget

This widget is used to toggle the on/off state of a single setting. The switch itself does not maintain any state. Instead, when the state of the switch changes, the **Switch** widget calls the **onChanged** callback function which will rebuild the switch with a new value to update the visual appearance of the switch.

The following figure displays how **Switch** widgets look like when they are on or off.



### Example:

In the following example, you will create a simple Flutter app including a **Switch** widget.

Perform the following steps:

- 1- Open **Android Studio**
- 2- Click **File** → **New** → **New Flutter Project**
- 3- Select **Flutter Application** and then click **Next**
- 4- Type: **switch\_widget** for Project Name and select **Lesson\_08** folder for the **Project Location**. Click **Next**

5- Type: **androidatc.com** for Company domain and then click **Finish**

Before you start typing the **Switch** widget code, you should check the following:

Switch widget depends on its configuration on a property called: **value**. This is a **bool** (Boolean data type) variable (on or off).

The following configuration will represent the **Switch** widget on your app interface:

```
Switch(  
  value: _value,  
  onChanged: _onChanged,  
) ,
```

As you see, this configuration depends on the **value** (Boolean variable) and **onChanged** function. You should configure them before adding **Switch** widget. The configuration follows:

```
bool _value = false;  
  
void _onChanged(bool value) {  
  setState(() {  
    _value = value;  
  });  
}
```

Now, you may continue creating this Switch app.

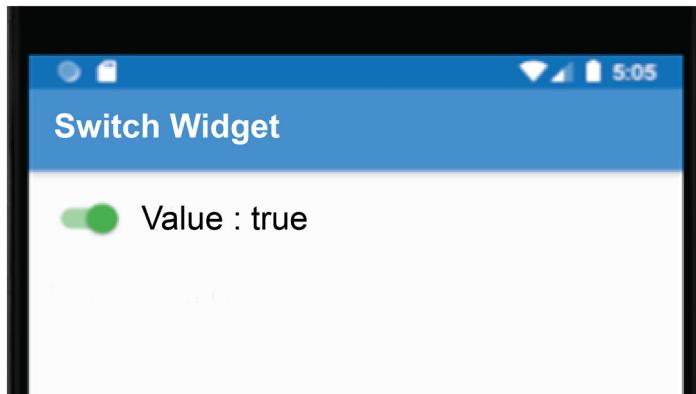
6- Open the **main.dart** file, delete all its content, and then type the following code:

```
import 'package:flutter/material.dart';  
  
void main() => runApp(MyApp());  
  
class MyApp extends StatefulWidget {  
  // This widget is the root of your application.  
  @override  
  _MyAppState createState() => _MyAppState();  
}  
  
class _MyAppState extends State<MyApp> {
```

```
bool _value = false;
void _onChanged(bool value) {
    setState(() {
        _value = value;
    });
}

@Override
Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
                title: Text('Switch Widget'),
            ),
            body: Padding(
                padding: const EdgeInsets.all(8.0),
                child: Column(
                    children: <Widget>[
                        Row(
                            children: <Widget>[
                                Switch(
                                    value: _value,
                                    onChanged: _onChanged,
                                    activeColor: Colors.green,
                                ),
                                Text(
                                    'Value : $_value',
                                    style: TextStyle(fontSize: 20.0),
                                ),
                            ],
                        ),
                    ],
                ),
            );
}
}
```

7- Run your app. Tap the **Switch** widget (on & off). The run output follows:



## AlertDialog Widget

An alert dialog informs the app user about situations that require acknowledgement. An alert dialog has an optional title and an optional list of actions.

The following figure displays an example of the alert dialog widget and its properties. The title is displayed above the content and the actions (buttons) are displayed below the content.



### Example:

In this example, you will create a small Flutter app having a button (**RaisedButton**). When the app user clicks this button, the **AlertDialog** will be generated. This alert dialog box as it is illustrated in the previous figure has a title, content, and two action buttons.

When the app user taps **No**, a function will close this alert dialog box, but when the user taps **Yes**, two functions will work: the first function will print a text in the run console, and the second function will close this alert dialog box.

To do that, perform the following steps:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** and then click **Next**

4- Type: **alert\_dialog** for Project Name and select **Lesson\_08** folder for the **Project Location**. Click **Next**

5- Type: **androidatc.com** for Company domain and then click **Finish**

6- Open the **main.dart** file, delete all its content, and then type the following code:

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

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

class MyApp extends StatefulWidget {
    // This widget is the root of your application.

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

class _MyAppState extends State<MyApp> {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(
                    title: Text('Alert Dialog Widget'),
                ),
                body: Center(
                    child: Padding(
                        padding: const EdgeInsets.all(8.0),
                        child: Column(
                            children: <Widget>[
                                Row(
                                    children: <Widget>[
                                        Container(
                                            width: 150,
                                            height: 50,
                                        )
                                    ],
                                )
                            ],
                        ),
                    ),
                ),
            ),
        );
}
```



```
        ),
        onPressed: () => Navigator.pop(context),
    ],
]);

showDialog(
    context: context,
    builder: (BuildContext context) {
        return alertDialog;
    },
);
}

}
```

7- Run your app, tap the Alert Dialog button, click **OK**, and check if you got the “**Thanks, I got it.**” message at the run console and the alert dialog box has been closed.

8- Reload your app again, tap the Alert Dialog button and click **NO**. The alert dialog box must be closed.

9- Don’t close this app because you will use it in the next section.

## CupertinoAlertDialog Widget

This widget is similar to the previous **AlertDialog** widget. However, **CupertinoAlertDialog** widget displays the dialog contents that look like standard iOS dialog buttons, and titles as illustrated in the following figure:

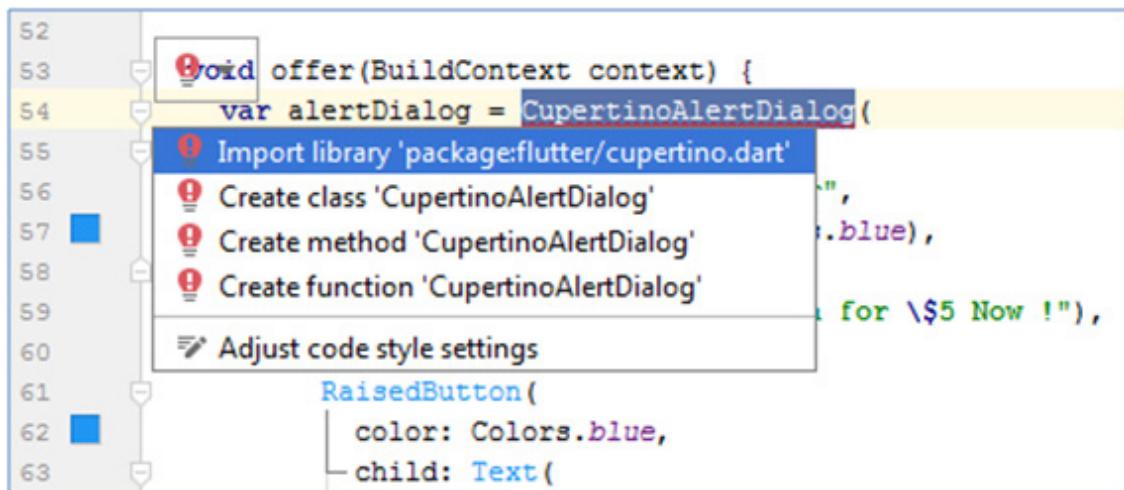


To configure **CupertinoAlertDialog** widget, use the same previous project (**alert\_dialog**), and just replace the **AlertDialog** with **CupertinoAlertDialog**.

A red underline will appear under the **CupertinoAlertDialog** widget. To fix this error, double click the **CupertinoAlertDialog** widget, click the red lamp icon, and

then select:

`Import library'package:flutter/cupertino.dart'` as illustrated in the following figure:



Then run your app again. You will get the `CupertinoAlertDialog` dialog box and find that its buttons work similarly as the `AlertDialog` widget in the previous example.

## Bottom Sheet Widget

This sheet is similar to a small interface which appears at the bottom of the app user interface when the app user takes a specific action such as clicking a button.

There are two types of bottom sheets in material design, **Modal** and **Persistent** bottom sheets. In the following section of this lesson, you will learn in details about the differences between, and how to configure the modal and persistent bottom sheets in Flutter.

### 1- Modal Bottom Sheet

A modal bottom sheet is an alternative to a menu or a dialog which prevents the user from interacting with the rest of the app. Modal bottom sheets can be created and displayed with the `showModalBottomSheet` function.

**Example:**

In this example, you will create a small Flutter app including a raised button. When the app user clicks this button, he/she will get a modal bottom sheet.

Perform the following steps:

1- Open **Android Studio**

2- Click **File → New → New Flutter Project**

3- Select **Flutter Application** and then click **Next**

4- Type: **bottom\_sheet** for Project Name and select **Lesson\_08** folder for the Project Location. Click **Next**

5- Type: **androidatc.com** for Company domain, and then click **Finish**

6- Open the **main.dart** file, delete all its contents, and then type the following code:

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

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

class MyApp extends StatelessWidget {

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(
                    title: Text('Alert Dialog Widget'),
                ),
                body: Center(
                    child: Padding(
                        padding: const EdgeInsets.all(8.0),
                        child: Column(
                            children: <Widget>[
                                Row(
                                    children: <Widget>[
                                        Container(
                                            width: 200,
                                            height: 50,
                                        )
                                    ],
                                )
                            ],
                        ),
                    ),
                ),
            ),
        );
    }
}
```

```
        child: RaisedButton(
            color: Colors.blue,
            child: Text('Bottom Sheet',
                style: TextStyle(color: Colors.white, fontSize: 20.0),
            ),
            onPressed: () {
            },
            ),
            ),
            ],
            ),
            ],
            ),
            ),
            ),
            );
        );
    }
}
```

7- In this step, you will create the modal bottom sheet using the `showModalBottomSheet` function. Create this function and use a `Container` widget with a specific height to represent the bottom sheet body. You may add a child widget to this `Container` widget to present the bottom sheet content. The function code follows:

```
_showModalBottomSheet(context) {
    showModalBottomSheet(
        context: context,
        builder: (BuildContext context) {
            return Container(
                height: 200.0,
                alignment: Alignment.topCenter,
                child: Text(
                    'Android ATC',
                    style: TextStyle(
                        fontSize: 20.0,
                    ),
                ),
            );
        });
}
```

8- Configure a raised button to call this function when the app user taps this button.  
The full code of this app follows:

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

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

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  _showModalBottomSheet(context) {
    showModalBottomSheet(
      context: context,
      builder: (BuildContext context) {
        return Container(
          height: 200.0,
          alignment: Alignment.topCenter,
          child: Text(
            'Android ATC',
            style: TextStyle(
              fontSize: 20.0,
            ),
          ),
        );
      );
    });
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Alert Dialog Widget'),
        ),
        body: Center(
          child: Padding(
            padding: const EdgeInsets.all(8.0),
            child: Column(
              children: <Widget>[
                Row(
                  children: <Widget>[
                    Container(
                      width: 200,
                      height: 50,
                      child: RaisedButton(
                        color: Colors.blue,
                        child: Text('Bottom Sheet'),
                      ),
                    ),
                  ],
                ),
              ],
            ),
          ),
        ),
      ),
    );
  }
}
```

```
        style: TextStyle(color: Colors.white, fontSize: 20.0),
        ),
        onPressed: () {
            _showModalBottomSheet(context);
        },
        ),
        ),
        ],
        ],
        ],
        ],
        ),
        ),
        ),
        );
    }
}
```

9- Run your app, click the button and you will get the modal bottom sheet as illustrated in the following figure. To close this bottom sheet and return to your app content, just click on any area of the app interface above this bottom sheet.

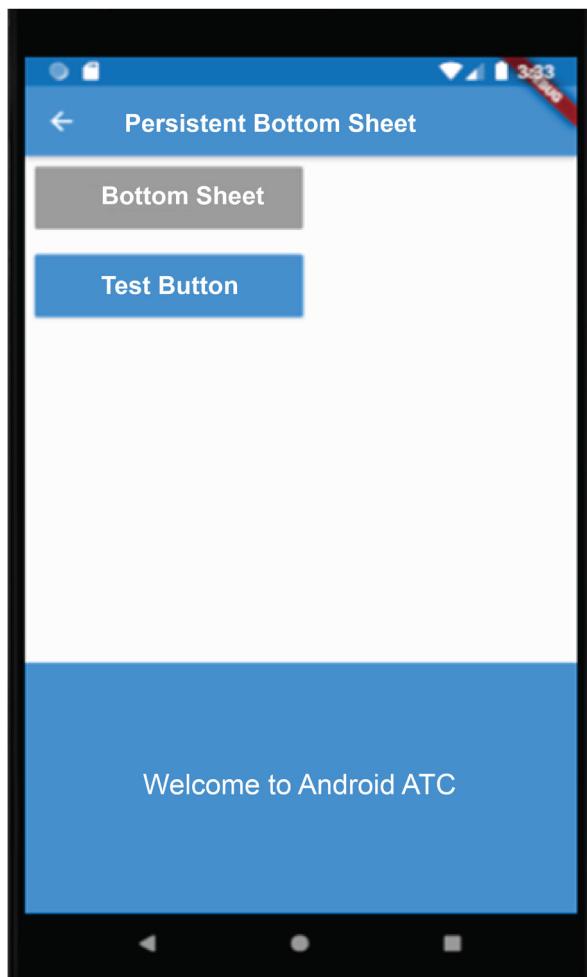


## 2- Persistent Bottom Sheet

A persistent bottom sheet shows information that supplements the primary content of the app. A persistent bottom sheet remains visible even when the user interacts with other parts of the app. Persistent bottom sheets can be created and displayed with the `ScaffoldState.showBottomSheet` function.

### Example:

In this example, you will create a small Flutter app including two raised buttons as illustrated in the following figure:



When the app user taps the **Bottom Sheet** button, he/she will get the *Persistent* bottom Sheet which has at Text widget: "Welcome to Android ATC", and at the same time the app user can tap the **Test Button** or any other part of this app within the Persistent bottom sheet while it is active. This is the main difference between the *Persistent* bottom sheet and the *Modal* bottom sheet.

To create this flutter app, perform the following steps:

1- Open **Android Studio**

2- Click **File** → **New** → **New Flutter Project**

3- Select **Flutter Application** and then click **Next**

4- Type: **persistent\_bottom\_sheet** for Project Name and click **Next**

5- Type: **androidatc.com** for Company domain and then click **Finish**

6- Open the **main.dart** file, delete all its contents, and then type the following code:

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

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

class MyApp extends StatefulWidget {
// This widget is the root of your application.

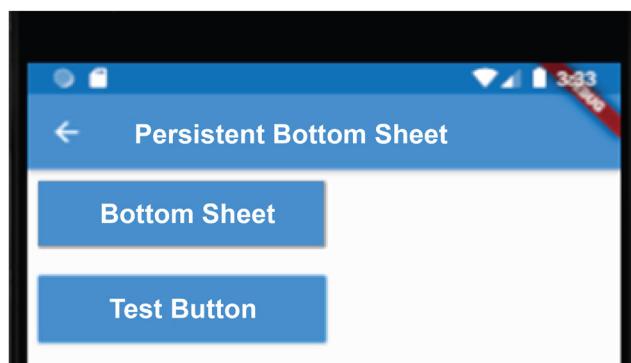
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Persistent Bottom Sheet'),
        ),
        body: Center(
          child: Padding(
            padding: const EdgeInsets.all(8.0),
            child: Column(
              children: <Widget>[
                Row(
                  children: <Widget>[
                    Container(
                      width: 200,
                      height: 50,
                      child: RaisedButton(
                        color: Colors.blue,
                        child: Text('Bottom Sheet'),
                      ),
                    ),
                  ],
                ),
              ],
            ),
          ),
        ),
      ),
    );
  }
}
```

```
        style:  
          TextStyle(color: Colors.white, fontSize: 20.0),  
        ),  
        onPressed: () {},  
      ),  
    ],  
  ),  
  
SizedBox(height: 20.0,),  
  
Row(  
  children: <Widget>[  
  Container(  
    width: 200,  
    height: 50,  
    child: RaisedButton(  
      color: Colors.blue,  
      child: Text('Test Button',  
        style: TextStyle(color: Colors.white, fontSize: 20.0),  
      ),  
      onPressed: () {},  
    ),  
  ),  
  ],  
),  
],  
,  
);  
}  
}  
}
```

7- Run your app and you will get the following run output:



8- Now, add the following code to `_MyAppState` class. Exactly below:

```
class _MyAppState extends State<MyApp> { }

final _scaffoldkey = GlobalKey<ScaffoldState>();
VoidCallback _showpersistentSheet;

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

    _showpersistentSheet = _sheetBottomSheet;
}

void _sheetBottomSheet() {
    setState(() {
        _showpersistentSheet = null;
    });
    _scaffoldkey.currentState
        .showBottomSheet((content) {

    return Container(
        color: Colors.blue,
        height: 200,
        child: Center(
            child: Text(
                'Welcome to Android ATC',
                style: TextStyle(fontSize: 20, color: Colors.white),
            ),
        ),
    ); //Container
})
.closed
.whenComplete(() {
    if (mounted) {
        setState(() {
            _showpersistentSheet = _sheetBottomSheet;
        });
    }
});
}
```

**Explanation:**

```
final _scaffoldkey = GlobalKey<ScaffoldState>();
```

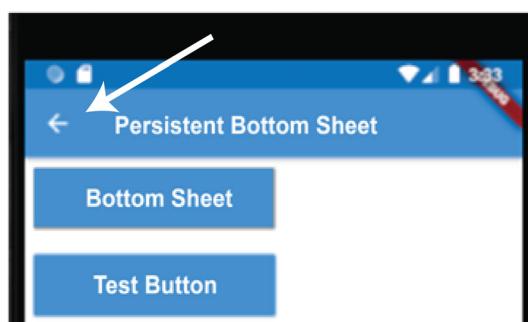
This part of the code configures `_scaffoldkey` as a global key which will be added to the Scaffold widget in your app to make a connection between the Scaffold widget and this bottom sheet.

The widgets that have global keys re-parent their sub-trees when they are moved from one location in the tree to another location in the same tree. In order to re-parent its sub-tree, a widget must arrive at its new location in the tree in the same animation frame in which it was removed from its old location in the tree.

Also, this part of the code created the function : `_showpersistentSheet` which will be called later by the “**Bottom Sheet**” button.

In addition, the `showBottomSheet` function in this code is used to display the bottom sheet content using a Container widget. This Container widget includes the content of the Persistent bottom sheet and its height property value represents the height of the bottom sheet (in this example is : 200).

The purpose of adding “`.closed.whenComplete`” is to return the value of the function : `_showpersistentSheet` again to the app interface before the app user taps this button (Bottom Sheet) and gets the bottom sheet. If you click the arrow button in the app title bar as illustrated in the following figure, the function `_showpersistentSheet` will be active or returns its original value. This means, by this configuration, the app user can tap this button again. When you run your app code, these details will be clearer.



9- Add the **ScaffoldState** key: `_scaffoldkey` to your **Scaffold** app widget as illustrated in the grey highlighted part of the following code:

```
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      key: _scaffoldkey,  
      appBar: AppBar(  
        title: Text('Persistent Bottom Sheet'),
```

10- Configure the button “**Bottom Sheet**” to call the function : `_showpersistentSheet` when the user taps this button. Only configure the grey highlighted part of the following code:

```
child: RaisedButton(  
  color: Colors.blue,  
  child: Text('Bottom Sheet',  
    style: TextStyle(color: Colors.white, fontSize: 20.0),  
,  
  onPressed: _showpersistentSheet,  
,  
,
```

The full code of this app follows:

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(MyApp());  
}  
  
class MyApp extends StatefulWidget {  
  // This widget is the root of your application.  
  
  @override  
  _MyAppState createState() => _MyAppState();  
}  
  
class _MyAppState extends State<MyApp> {  
  final _scaffoldkey = GlobalKey<ScaffoldState>();  
  VoidCallback _showpersistentSheet;
```

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

    _showpersistentSheet = _sheetBottomSheet;
}

void _sheetBottomSheet() {
    setState(() {
        _showpersistentSheet = null;
    });
    _scaffoldkey.currentState
        .showBottomSheet((content) {
            return Container(
                color: Colors.blue,
                height: 200,
                child: Center(
                    child: Text(
                        'Welcome to Android ATC',
                        style: TextStyle(fontSize: 20, color: Colors.white),
                    ),
                ),
            );
        })
        .closed
        .whenComplete(() {
            if (mounted) {
                setState(() {
                    _showpersistentSheet = _sheetBottomSheet;
                });
            }
        });
}

@Override
Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            key: _scaffoldkey,
            appBar: AppBar(
                title: Text('Persistent Bottom Sheet'),
            ),
            body: Center(
                child: Padding(
```

```
padding: const EdgeInsets.all(8.0),
child: Column(
  children: <Widget>[
    Row(
      children: <Widget>[
        Container(
          width: 200,
          height: 50,
          child: RaisedButton(
            color: Colors.blue,
            child: Text('Bottom Sheet',
            style: TextStyle(color: Colors.white, fontSize: 20.0),
            ),
            onPressed: _showpersistentSheet,
          ),
        ),
      ],
    ),
  ],
),
SizedBox(height: 20.0,),

Row(
  children: <Widget>[
    Container(
      width: 200,
      height: 50,
      child: RaisedButton(
        color: Colors.blue,
        child: Text('Test Button',
        style: TextStyle(color: Colors.white, fontSize: 20.0),
        ),
        onPressed: () {},
      ),
    ),
  ],
),
),
),
),
),
),
);
}
}
```

11- Run your app. Click the **Bottom Sheet** button to get the bottom sheet.

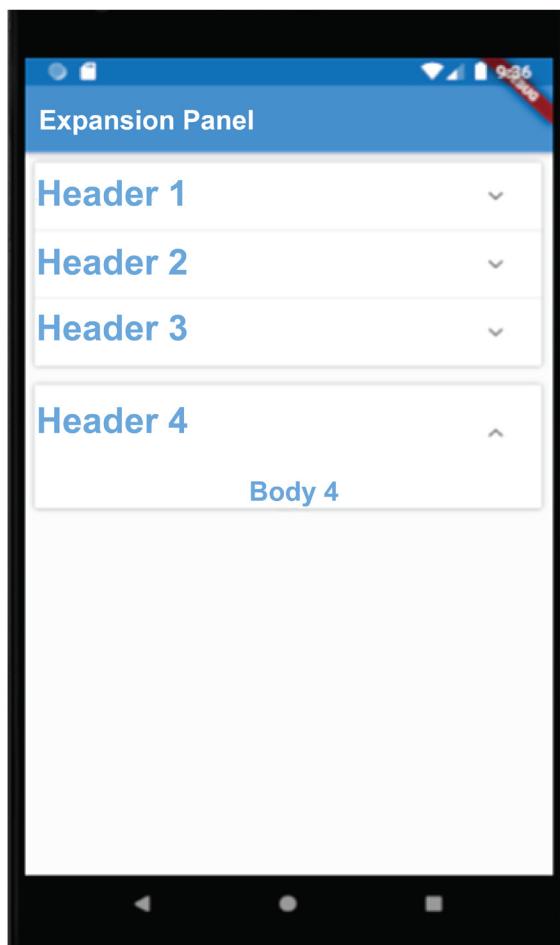
It should be noted that you can still click the **Text Button** even though the bottom sheet is working. This is the main difference between Persistent and Modal bottom sheets. If this bottom sheet was Modal type, you could not have tapped this Text Button.

Tap the white arrow on your app title bar and if you click the Bottom Sheet button again, you will find that it works fine.

## Expansion Panel Widget

This widget has a header and a body which can be either expanded or collapsed. The body of the panel is

only visible when it is expanded. This widget is used to display information about a list of items and if the app user needs more information or details about a specific item, he/she will tap the item arrow sign to expand it. The following figure displays how the **ExpansionPanel** widget looks like:



**Example:**

In this example, you will create a small Flutter app using the **ExpansionPanelList** class which lays out its items and builds their contents using the **ExpansionPanel** widget.

To create this Flutter app, perform the following steps:

1- Open **Android Studio**

2- Click **File** → **New** → **New Flutter Project**

3- Select **Flutter Application** and then click **Next**

4- Type : **expansion\_panel** for Project Name and click **Next**

5- Type: **androidatc.com** for Company domain and then click **Finish**

6- Open the **main.dart** file, delete all its contents, and then type the following code:

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

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

class MyApp extends StatefulWidget {
// This widget is the root of your application.

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

class _MyAppState extends State<MyApp> {

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Expansion Panel'),
        ),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: ListView(
            children: <Widget>[
            ],
          ),
        ),
      ),
    );
  }
}
```

```

        ),
        ),
        );
    }
}

```

7- Run your app. You should get an empty interface and have only the title bar.

### **Expansion Panel**

8- Create a new class including each expanding item components such as header and body. In this example, this class is called : **MyItem**

Add this class above the following code:

```
class _MyAppState extends State<MyApp> {
```

The class code follows:

```
class MyItem {
    MyItem({this.isExpanded: false, this.header, this.body});
    bool isExpanded;
    final String header;
    final String body;
}
```

9- Use the **MyItem** class to configure **\_items** object as a list. Each header and body of this list represent one item of the expansion panel items.

In this example, this **\_items** list declares four headers and bodies as illustrated in the following code:

```
List<MyItem> _items=<MyItem>[
    MyItem(header:"Header 1", body: "Body 1"),
    MyItem(header:"Header 2", body: "Body 2"),
    MyItem(header:"Header 3", body: "Body 3"),
    MyItem(header:"Header 4", body: "Body 4"),
];
```

The following figure displays where exactly you must add this code:

```

21   class _MyAppState extends State<MyApp> {
22     List<MyItem> _items = <MyItem>[
23       MyItem(header: "Header 1", body: "Body 1"),
24       MyItem(header: "Header 2", body: "Body 2"),
25       MyItem(header: "Header 3", body: "Body 3"),
26       MyItem(header: "Header 4", body: "Body 4"),
27     ]; // <MyItem>[]
28
29   @override
30   Widget build(BuildContext context) {
31     return MaterialApp(
32       home: Scaffold(
33         appBar: AppBar(
34           title: Text('Expansion Panel'),

```

10 - Now, you will add the **ExpansionPanelList** widget to configure the expansion items layout and add the **ExpansionPanel** widget which is responsible to configure and add the values of the `_items` list to the expansion panel items. You should configure these two widgets as children widgets of the **ListView** widget.

The following is the full code of this app :

```

import 'package:flutter/material.dart';

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

class MyApp extends StatefulWidget {
// This widget is the root of your application.

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

class MyItem {
  MyItem({this.isExpanded: false, this.header, this.body});
  bool isExpanded;
  String header;
  String body;
}

```

```
final String header;
final String body;
}

class _MyAppState extends State<MyApp> {
List<MyItem> _items = <MyItem>[
    MyItem(header: "Header 1", body: "Body 1"),
    MyItem(header: "Header 2", body: "Body 2"),
    MyItem(header: "Header 3", body: "Body 3"),
    MyItem(header: "Header 4", body: "Body 4"),
];
}

@Override
Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
                title: Text('Expansion Panel'),
            ),
        ),
        body: Padding(
            padding: const EdgeInsets.all(8.0),
            child: ListView(
                children: <Widget>[
                    ExpansionPanelList(
                        expansionCallback: (int index, bool isExpanded) {
                            setState(() {
                                _items[index].isExpanded = !_items[index].isExpanded;
                            });
                        },
                        children: _items.map((MyItem item) {
                            return ExpansionPanel(
                                headerBuilder: (BuildContext context, bool isExpanded) {
                                    return Text(
                                        item.header,
                                        style: TextStyle(fontSize: 30, color: Colors.blue),
                                    );
                                },
                                isExpanded: item.isExpanded,
                                body: Container(
                                    child: Text(
                                        item.body,
                                        style: TextStyle(fontSize: 20, color: Colors.blue),
                                    ),
                                ),
                            );
                        }).toList(),
                    ),
                ],
            ),
        ),
    );
}

class MyItem {
    final String header;
    final String body;
}
```

```
        ));  
    }).toList(),  
    ),  
    ],  
    ),  
    ),  
    ),  
    );  
}  
}
```

11- Run your app and test your expansion panel by expanding and collapsing its items.

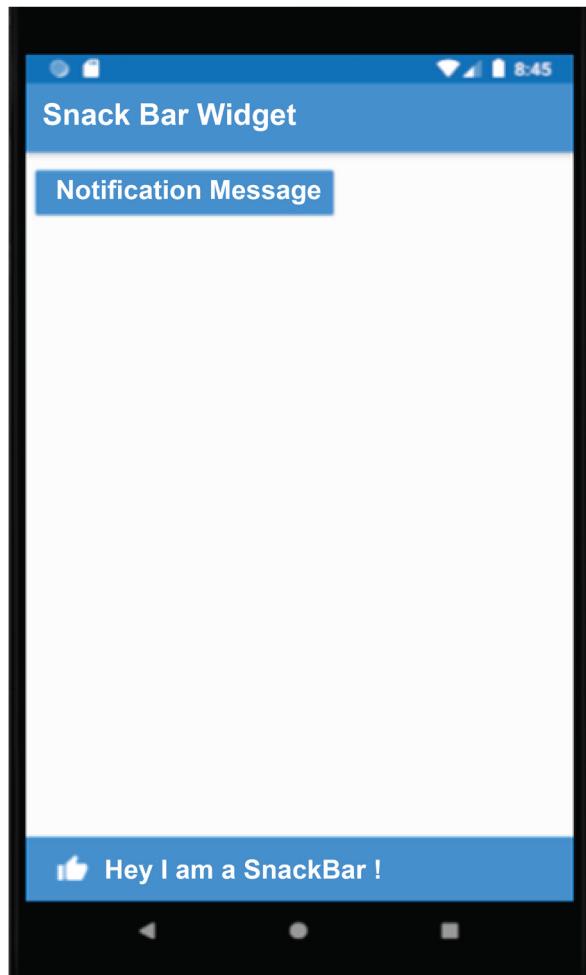
## SnackBar Widget

**SnackBar** widget is used if you want to let your app pop up a message for a few seconds at the bottom of your app interface. The following figure displays how and where the **SnackBar** message looks like.

An example of **SnackBar** message is when you send an email using the Gmail app, you usually get a message at the bottom of your phone device as a notification telling you that your message has been sent.

This notification message appears temporarily for seconds and then it is hidden automatically.

In this section, you will add a **SnackBar** widget to your app code, configure the notification message content, message duration, and configure the action which will generate this notification message.



### Example:

In this example, you will create a small Flutter app including a raised button. When the app user taps this button, a **SnackBar** message appears at the bottom of the smart device.

To create this Flutter app, perform the following steps:

- 1- Open Android Studio
- 2- Click **File** → **New** → **New Flutter Project**
- 3- Select **Flutter Application** and then click **Next**
- 4- Type: **snack\_bar\_widget** for Project Name and click **Next**
- 5- Type: **androidatc.com** for Company domain and then click **Finish**
- 6- Open the **main.dart** file, delete all its contents, and then type the following code:

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

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

class MyApp extends StatefulWidget {
// This widget is the root of your application.

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

class _MyAppState extends State<MyApp> {

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Snack Bar Widget'),
        ),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: RaisedButton(
            color: Colors.blue,
            child: Text('Notification Message',
              style: TextStyle(color: Colors.white, fontSize: 20.0),
            ),
            onPressed: () {},
          ),
        ),
      ),
    );
  }
}
```

7- Run your app. You should get an interface including a raised button only without any action.

8- Now, create a new function called `_showSnackBar()` which includes the `SnackBar`

widget configurations such as message content, duration and the Scaffold key.

You should add these configurations directly below the following code:

```
class _MyAppState extends State<MyApp> {
```

The configuration code is the following:

```
final GlobalKey<ScaffoldState> _scaffoldKey =  
GlobalKey<ScaffoldState>();  
_showSnackBar() {  
    final snackBar = SnackBar(  
        content: Row(  
            children: [  
                Icon(Icons.thumb_up),  
  
                SizedBox(width: 10.0,),  
  
                SizedBox(  
                    child: Text('Hey I am a Snackbar !',  
                        style: TextStyle(fontSize: 20, color: Colors.white),  
                    ),  
                ),  
            ],  
        ),  
        duration: Duration(seconds: 3),  
        backgroundColor: Colors.blue,  
    );  
    _scaffoldKey.currentState.showSnackBar(snackBar);  
}
```

9- Add the Scaffold key to the Scaffold widget by adding the grey highlighted part of the following code:

```
@override  
Widget build(BuildContext context) {  
    return MaterialApp(  
        home: Scaffold(  
            key: _scaffoldKey,  
            appBar: AppBar(  
                title: Text('Snack Bar Widget'),
```

10- The last step is configuring the raised button to generate the **SnackBar** message when the app user taps this button. Configure this button as follows:

```
onPressed: _showSnackBar,
```

The full code follows:

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

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

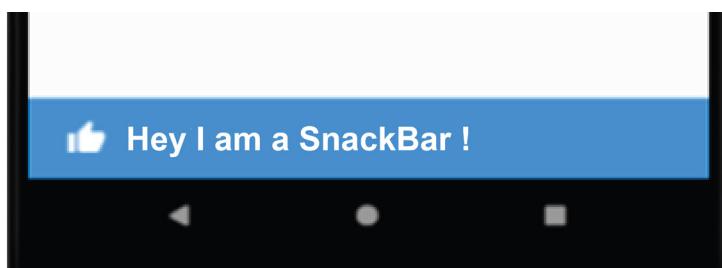
class MyApp extends StatefulWidget {
// This widget is the root of your application.

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

class _MyAppState extends State<MyApp> {
  final GlobalKey<ScaffoldState> _scaffoldKey =
  GlobalKey<ScaffoldState>();
  _showSnackBar() {
    final snackBar = SnackBar(
      content: Row(
        children: [
          Icon(Icons.thumb_up),
          SizedBox(width: 10.0,),
          SizedBox(
            child: Text('Hey I am a SnackBar !',
              style: TextStyle(fontSize: 20, color: Colors.white),
            ),
          ),
        ],
      ),
      duration: Duration(seconds: 3),
      backgroundColor: Colors.blue,
    );
    _scaffoldKey.currentState.showSnackBar(snackBar);
  }
}
```

```
@override
Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            key: _scaffoldKey,
            appBar: AppBar(
                title: Text('Snack Bar Widget'),
            ),
            body: Padding(
                padding: const EdgeInsets.all(8.0),
                child: RaisedButton(
                    color: Colors.blue,
                    child: Text('Notification Message',
                        style: TextStyle(color: Colors.white, fontSize: 20.0),
                    ),
                    onPressed: _showSnackBar,
                ),
            ),
        ),
    );
}
```

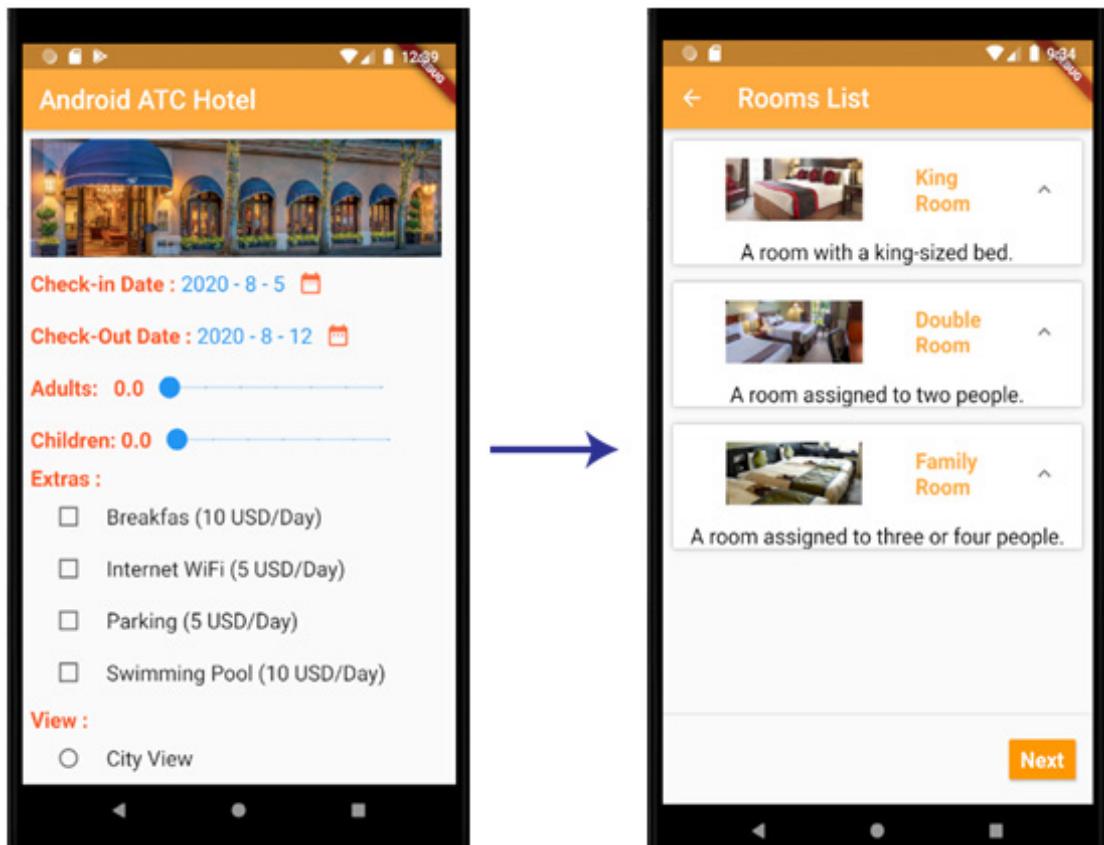
11- Run your app, tap the button and check the **SnackBar** notification message which will appear at the bottom of your app interface for 3 seconds as illustrated in the following figure:



## Lab 8: Creating a Hotel Reservation App

In this lab, you will create a Flutter app for a hotel using almost all the topics of this lesson such as Date picker plug-in, Slider, Check box, Radio button, and Expansion Panel widgets.

The following figure includes two interfaces of the app which you will create. The first interface includes a form asking the app user to enter what he/she needs, and when the app user clicks the button at the end of this form, he/she will be directed to another interface including an expansion panel widget. This expansion widget includes images of room categories and some details or information about each room (expansion panel item).



To create this app, perform the following actions step by step. Or if you don't have enough time, you may use the lab source code for this lab to test how it works.

The following steps are:

- 1- Open Android Studio

2- Click **File** → **New** → **New Flutter Project**

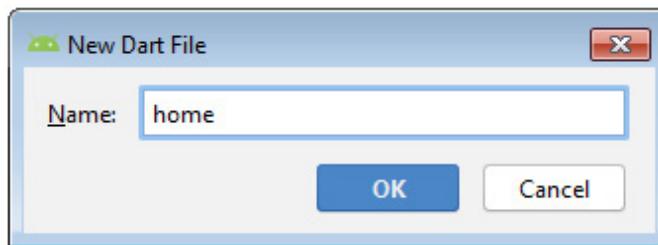
3- Select **Flutter Application** and then click **Next**

4- Type : **lab\_08** for Project Name and create a new folder : **Lab\_08** for the Project Location. Click **Next**

5- Type: **androidatc.com** for Company domain and then click **Finish**

6- Right click the **lib** folder (**lab\_08**→ **lib**) and select **New** → **Dart File**

As illustrated in the following figure, type: **home** for the file name and then click **OK**



7- Repeat the same previous step to add a new Dart file: **roomsPanel.dart**

8 - Open **home.dart** file and add the following code which will add the basic components to this app interface such as the title bar and a Container widget. This is just to be sure that the design works fine. The code is the following:

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

class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Colors.orangeAccent,
          title: Text(
            'Android ATC Hotel',
            style: TextStyle(fontSize: 25.0, color: Colors.white),
          ),
        ),
        body: Container(
          child: Text('Home'),
        ),
      ),
    );
}
```

```
    );  
}  
}
```

9- Copy all the content of **home.dart** file.

10- Open the **roomsPanel.dart** file and **paste** the code of **home.dart** in **roomsPanel.dart** file.

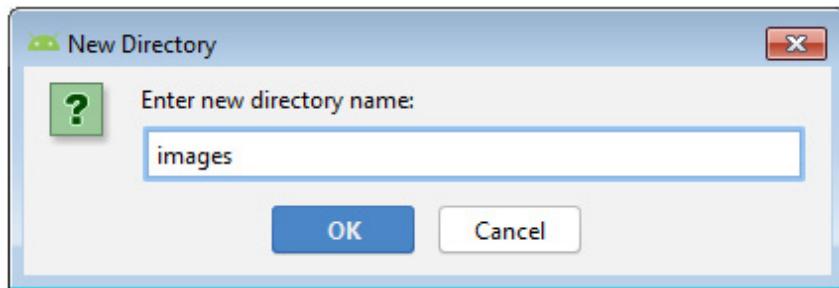
11-In **roomsPanel.dart** replace the class name: **Home** with **RoomsPanel**

12- Open **main.dart** file and **Delete** all its code contents. The **main.dart** will be configured to include the navigation names only. Open **main.dart** and type the following code:

```
import 'package:flutter/material.dart';  
import 'home.dart';  
import 'roomsPanel.dart';  
  
void main() => runApp(MyApp());  
  
class MyApp extends StatelessWidget {  
  // This widget is the root of your application.  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Home(),  
      routes: {  
        'homeLink': (context) => Home(),  
        'RoomsPanel': (context) => RoomsPanel(),  
      },  
    );  
  }  
}
```

13- Run your app. You should get the content of **home.dart** file which includes a simple interface just containing the title bar and the text “Home”.

14- Add “**images**” directory to your project. Right click your project name (**lab\_08**) then select: **New → New Directory**. Then type the directory name “**images**”, and then click **OK** as illustrated in the following figure:



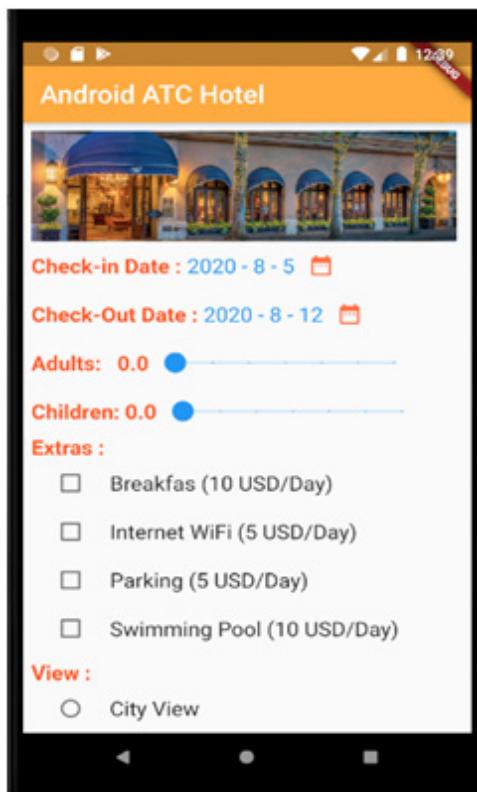
15- Open the **pubspec.yaml** file and configure **images** directory to be the default location for all your images in this app. Just remove the hash sign for **assets** and configure the images directory name as illustrated in the following figure:

```
43      # To add assets to your application,  
44      assets:  
45          - Images/  
46              #     - images/a_dot_ham.jpeg
```

Then, click **Packages get**.

16- Your computer should have the lab source files for this course. Open the following path on your computer: **Lab Source Files\Images\Lab 8**, copy all the image files, right click the **images** directory in your app, select **paste** and then click **OK**.

17- Open **home.dart** and you should design the following interface which will ask the app user to reserve a room depending on specific criteria.



In the **home.dart** file, add the following code to its **body** property :

```
body: Padding(
    padding: const EdgeInsets.all(8.0),
    child: Column(children: <Widget>[
        Row(
            children: <Widget>[
                Image(
                    alignment: Alignment.topCenter,
                    width: 350.0,
                    image: AssetImage('images/entrance.jpg'),
                ),
            ],
        ),
        Row(
// This row for Check-in Date : Android ATC - Lab 8
            children: <Widget>[
                Text('Check-in Date : ',
                    style: TextStyle(
                        fontSize: 20.0,
```

```
        color: Colors.deepOrange,  
        fontWeight: FontWeight.bold,  
    ),  
),  
  
Text('Next code is coming' ),  
IconButton(  
    icon: Icon(  
        Icons.date_range,  
        color: Colors.deepOrangeAccent,  
    ),  
    onPressed: () { },  
),  
],  
)  
],  
)  
,
```

18- In `home.dart`, double click : `StatelessWidget` in the following code:

```
class Home extends StatelessWidget {
```

And then, click the orange lamp and select : **Convert to StatefulWidget**

19- You will now add the following code to configure the hotel check-in date for the app user. In `home.dart` file, click directly below the following line of code:

```
class HomeState extends State<Home> {
```

Then, add the following Date Picker code:

```
DateTime checkInDate = DateTime.now();

Future<Null> selectDate(BuildContext context) async {
    final DateTime picked = await showDatePicker(
        context: context,
        initialDate: checkInDate,
        firstDate: DateTime(2020),
        lastDate: DateTime(2030),
    );
    if (picked != null && picked != checkInDate) {
```

```
    setState(() {
      checkInDate = picked;
    });
}
```

20- In **home.dart** file, you will add the code to display the value of the selected check-in Date.

Replace the following code :

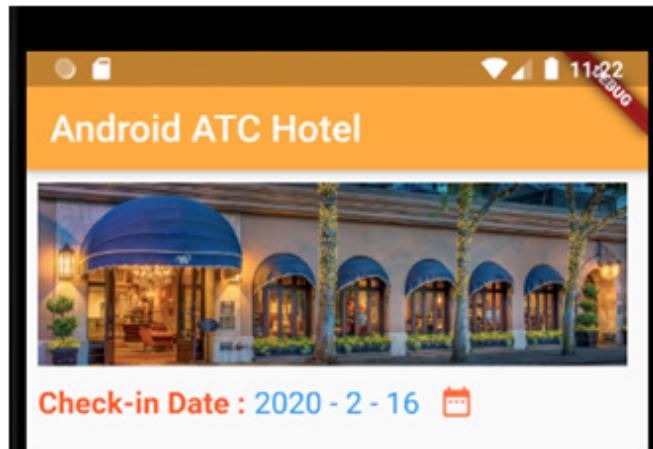
```
Text('Next code is coming' ),
IconButton(
  icon: Icon(
    Icons.date_range,
    color: Colors.deepOrangeAccent,
  ),
  onPressed: () { },
),
```

with the following code:

```
Text(
  '${checkInDate.year} - ${checkInDate.month} - ${checkInDate.day}')
  .toString(),
style: TextStyle(fontSize: 20.0, color: Colors.blue),
),

IconButton(
  icon: Icon(
    Icons.date_range,
    color: Colors.deepOrangeAccent, ),
  onPressed: () {
    selectDate(context);
  },
),
```

21- Run your app and test the check-in section. At this stage, you should get the following figure :



22- Now, you should add the check-out section. Open **home.dart** file and directly above the following code:

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

add the following code to configure the checkOutDate:

```
DateTime checkOutDate = DateTime.now();  
  
Future<Null> selectDate2(BuildContext context) async {  
    final DateTime pickedout = await showDatePicker(  
        context: context,  
        initialDate: checkOutDate,  
        firstDate: DateTime(2020),  
        lastDate: DateTime(2030),  
    );  
  
    if (pickedout != null && pickedout != checkOutDate) {  
        setState(() {  
            checkOutDate = pickedout;  
        });  
    }  
}
```

23- Under the **Row** which represents the Check-in date, add a new **Row** widget including the following code:

```
Row(
// This row for Check-out Date
children: <Widget>[
    Text('Check-Out Date : ',
        style: TextStyle(
            fontSize: 20.0,
            color: Colors.deepOrange,
            fontWeight: FontWeight.bold,
        ),
    ),
    Text(
        '${checkOutDate.year} - ${checkOutDate.month} -
${checkOutDate.day}')
        .toString(),
        style: TextStyle(fontSize: 20.0, color: Colors.blue),),
    IconButton(
        icon: Icon(
            Icons.date_range,
            color: Colors.deepOrangeAccent,),

        onPressed: () {
            selectDate2(context);
        },
    ),
],
)
```

24- Now, in **home.dart**, you will add two Slider widgets to represent the number of adults and children. Before configuring these slider widgets, configure the following two variables (**adult\_Number** and **child\_Number**) directly above the following :

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

The variables follow:

```
var adult_Number = 0.0;  
var child_Number = 0.0;
```

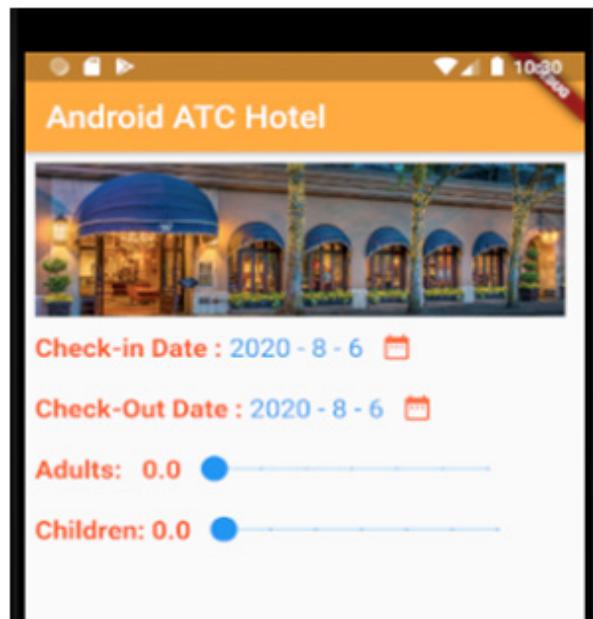
25- In **home.dart** , add the following two Row widgets to configure two sliders to your app interface.

This code must be written after closing the Row widget which wraps the Check-out date.

```
/** Here the code for Slider Widget : Number of Adults  
  
Row(  
    children: <Widget>[  
        Text(  
            'Adults: '  
            '$adult_Number',  
            style: TextStyle(  
                fontSize: 20.0,  
                color: Colors.deepOrange,  
                fontWeight: FontWeight.bold,  
            ),  
        ),  
        Container(  
            width: 250.0,  
            child: Slider(  
                value: adult_Number,  
                onChanged: (newRating) {  
                    setState(() => adult_Number = newRating);  
                },  
                divisions: 6,  
                label: '$adult_Number.Adult',  
                min: 0,  
                max: 6,  
            ),  
        ),  
    ],  
)  
  
***** Here the code for Slider Widget : Number of Children  
  
Row(  
    children: <Widget>[
```

```
Text(  
    'Children: '  
    '$child_Number',  
    style: TextStyle(  
        fontSize: 20.0,  
        color: Colors.deepOrange,  
        fontWeight: FontWeight.bold,  
    ),  
,  
  
Container(  
    width: 250.0,  
    child: Slider(  
        value: child_Number,  
        onChanged: (newRating) {  
            setState(() => child_Number = newRating);  
        },  
        divisions: 6,  
        label: '$child_Number.Children',  
        min: 0,  
        max: 6,  
    ),  
,  
],  
,
```

26- Run your app. The run output should be as follows:



27- Before adding the checkboxes and radio buttons to your app, it is recommended to wrap your app body's entire content using a **ListView** widget to avoid any problem in vertical scrolling of your app interface.

To do that, make the parent widget for your app **body** as **ListView**, and then wrap the **ListView** widget using the **Padding** widget to get a margin for your app interface content.

Modify your body code to be as illustrated in the following figure:

```
body: Padding(
  padding: const EdgeInsets.all(8.0),
  child: ListView(
    children: <Widget>[
      Column(
        children: <Widget>[
          Row(
            children: <Widget>[
              Image(
                alignment: Alignment.topCenter,
                width: 390.0,
                image: AssetImage('images/entrance.jpg'),
              ), // Image
            ],
          ),
        ],
      ),
    ],
  ),
)
```

28- To add a group of checkboxes and radio buttons to your app interface, you should configure the **pubspec.yaml** file as follows:

```
grouped_buttons: ^1.0.4
```

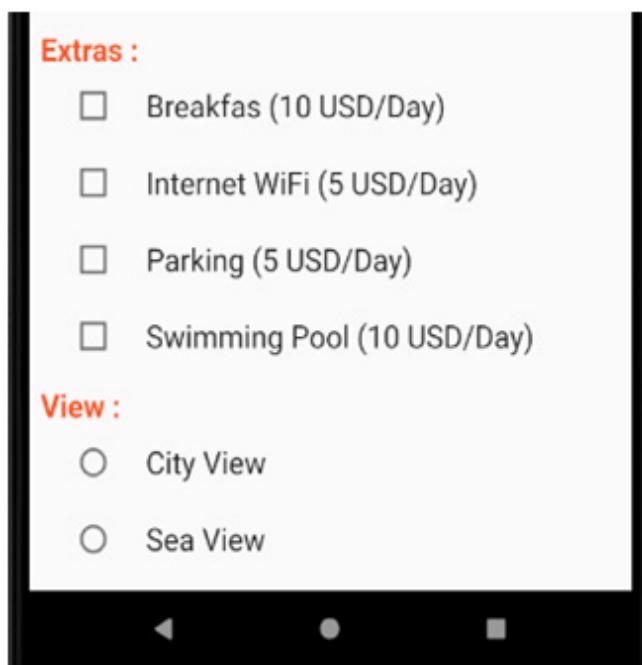
Or, as illustrated in the following figure:

```
19   dependencies:
20     flutter:
21       sdk: flutter
22     grouped_buttons: ^1.0.4
23
2 spaces
```

Then, click **Packages get** in the pubspec.yaml file .

29- Now, add the following code to add the checkboxes and radio buttons which are

illustrated in the following figure:



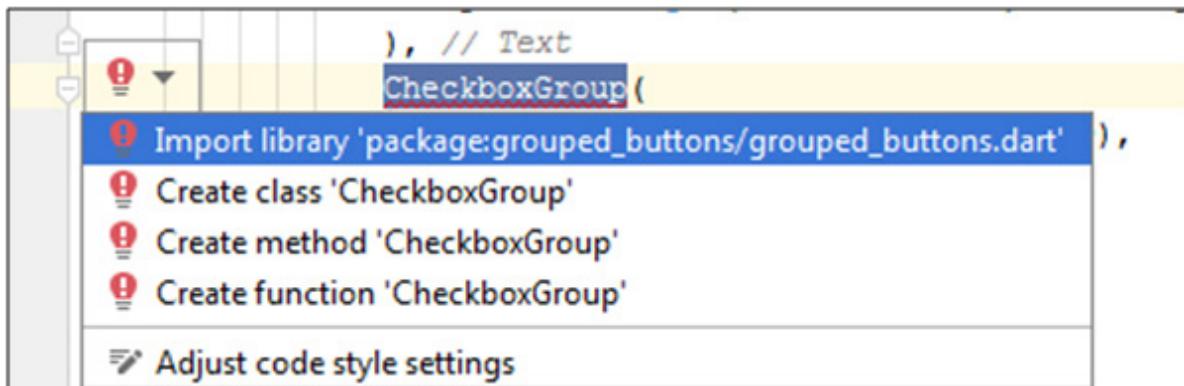
To do that, add the following code after closing the last **Row** widget (Slider widget).

```
Row(
  children: <Widget>[
    Text(
      'Extras :',
      style: TextStyle(
        fontSize: 20.0,
        color: Colors.deepOrange,
        fontWeight: FontWeight.bold,
      ),
    ),
  ],
),
Container(
  height: 200.0,
  child: CheckboxGroup(
    labelStyle: TextStyle(fontSize: 20.0),
    labels: <String>[
      "Breakfast (10 USD/Day)",
      "Internet WiFi (5 USD/Day)",
      "Parking (5 USD/Day)",
      "Swimming Pool (10 USD/Day)",
    ],
    onSelected: (List<String> checked) =>
```

```
        print(checked.toString()),
    ),
),
Row(
    children: <Widget>[
        Text('View :',
            style: TextStyle(
                fontSize: 20.0,
                color: Colors.deepOrange,
                fontWeight: FontWeight.bold,
            ),
        ),
    ],
),
Container(
    child: RadioButtonsGroup(
        orientation: GroupedButtonsOrientation.VERTICAL,
        labelStyle: TextStyle(fontSize: 20.0),
        labels: <String>[
            "City View",
            "Sea View",
        ],
        onSelected: (String selected) => print(selected),
    ),
),
```

30- You will get a red underline below **CheckboxGroup** widget. Double click this widget, click the red lamp, and then select the following choice as illustrated in the following figure:

```
Import library 'package:grouped_buttons/grouped_buttons.dart'
```



31 - Run your app to be sure that you get the first app interface.

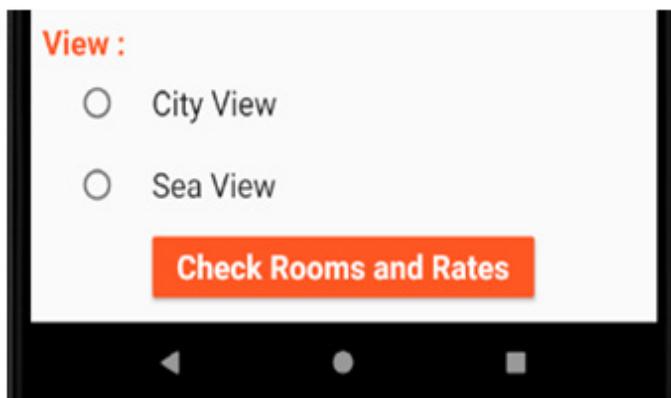
32- Now, you will add the last part of your first app interface which is a search button.

To do that, add a new **Row** widget and add a **RaisedButton** widget.

The code for this Row and **RaisedButton** widgets follows. When the app user taps this button, he/she will move to **roomsPanel.dart** file contents. You already configured the navigation configurations for your app in the **main.dart** file.

```
Row(  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: <Widget>[  
        RaisedButton(  
            color: Colors.deepOrange,  
            child: Text(  
                'Check Rooms and Rates',  
                style: TextStyle(  
                    color: Colors.white,  
                    fontSize: 20.0,  
                    fontWeight: FontWeight.bold),  
            ),  
            onPressed: () {  
                Navigator.pushNamed(context, 'RoomsPanel');  
            },  
        ),  
    ],  
) ,
```

33- Run your app and scroll down the interface. You should get the following figure:

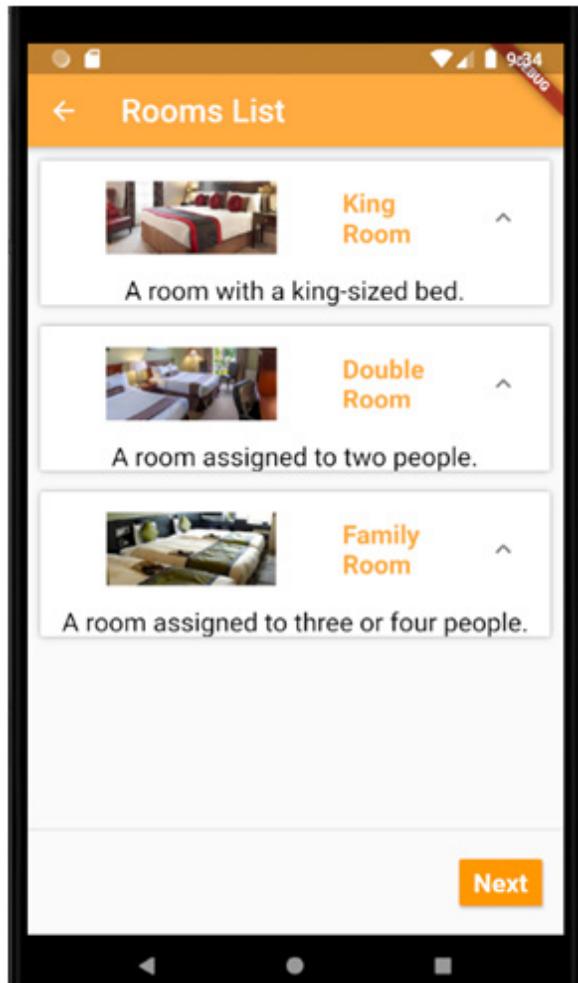


34- Open **roomsPanel.dart** file and then double click the **StatelessWidget{** of the following code:

```
class RoomsPanel extends StatelessWidget {
```

Then click the orange lamp and select **Convert to StatefulWidget**

35- In **roomsPanel.dart** file, you should configure an Expansion Panel widget to display a list of hotel rooms and some details about each item (room). You should get an interface as illustrated in the following figure:



**Note:** Before typing the code of **roomsPanel.dart** file, you should understand what you will do here. This Expansion Panel will display text and images depending on its work on a class called **NewItem** in this lab. This class includes four variables which are:

**isExpanded** (Boolean) determines if this panel is expandable or not.

**header** (Widget): Represents the images.

**selection** (String): Represents the text which will appear beside the image.

**body** (String): The text which will appear in the expanded area of the expansion panel.

The class code follows (Don't type the following code in your lab):

```
class NewItem {  
  bool isExpanded;  
  final Widget header;  
  final String selection;  
  final String body;  
  
  NewItem(  
    this.isExpanded,  
    this.header,  
    this.selection,  
    this.body,  
  );  
}
```

Also, you will use a list called **items** which will have the values of the class items (header, selection and body). You may add more items to this list as you have data.

36- Type the following code in **roomsPanel.dart**:

```
import 'package:flutter/material.dart';  
  
class RoomsPanel extends StatefulWidget {  
  @override  
  _RoomsPanelState createState() => _RoomsPanelState();  
}  
  
class NewItem {  
  bool isExpanded;  
  final Widget header;  
  final String selection;  
  final String body;  
  
  NewItem(  
    this.isExpanded,  
    this.header,  
    this.selection,  
    this.body,  
  );  
}
```

```
    this.isExpanded,
    this.header,
    this.selection,
    this.body,
  );
}

class _RoomsPanelState extends State<RoomsPanel> {
  List<NewItem> items = <NewItem>[

    NewItem(
      false,
      Container(
        child: Image(
          height: 100,
          width: 200,
          image: AssetImage('images/1bed.jpg'),
        ),
      ),
      "King Room",
      "A room with a king-sized bed.",

    NewItem(
      false,
      Container(
        child: Image(
          height: 100,
          width: 200,
          image: AssetImage('images/2beds.jpg'),
        ),
      ),
      "Double Room",
      "A room assigned to two people.",

    NewItem(
      false,
      Container(
        child: Image(
          height: 100,
          width: 200,
          image: AssetImage('images/3beds.jpg'),
        ),
      ),
      "Family Room",
      "A room assigned to three or four people."),
  ];
}
```

```
ListView List_Criteria;  
Widget build(BuildContext context) {  
    List_Criteria = new ListView(  
        children: [  
            new Padding(  
                padding: new EdgeInsets.all(10.0),  
                child: new ExpansionPanelList(  
                    expansionCallback: (int index, bool isExpanded) {  
                        setState(() {  
                            items[index].isExpanded = !items[index].isExpanded;  
                        });  
                    },  
                    children: items.map((NewItem item) {  
                        return new ExpansionPanel(  
                            headerBuilder: (BuildContext context, bool isExpanded) {  
                                return new ListTile(  
                                    leading: item.header,  
                                    title: new Text(  
                                        item.selection,  
                                        textAlign: TextAlign.left,  
                                        style: new TextStyle(  
                                            fontSize: 20.0,  
                                            fontWeight: FontWeight.bold,  
                                            color: Colors.orangeAccent,  
                                        ),  
                                    ),  
                                );  
                            },  
                            isExpanded: item.isExpanded,  
                            body: Text(  
                                item.body,  
                                style: TextStyle(  
                                    fontSize: 20,  
                                    color: Colors.black,  
                                ),  
                            ),  
                        );  
                    }).toList(),  
                ),  
            ),  
        ],  
    );  
  
    Scaffold scaffold = Scaffold(  
        appBar: AppBar(  
            backgroundColor: Colors.orangeAccent,  
            title: Text("Rooms List",  
                style: TextStyle(fontSize: 25.0, color: Colors.white),  
            ),  
    );
```

```
        ),
        ),

body: List_Criteria,
persistentFooterButtons: <Widget>[
    ButtonBar(children: <Widget>[
        RaisedButton(
            color: Colors.orange,
            child: Text('Next',
                style: TextStyle(
                    color: Colors.white,
                    fontSize: 20.0,
                    fontWeight: FontWeight.bold),
            ),
            onPressed: () {},
        ),
    ],
),
],
);
return scaffold;
}
}
```

37- Run your app. Tap the button : **Check Rooms and Rates** and expand the expansion panel by tapping the arrow signs.

You may add additional configurations to this app such as when the app user taps the **Next** button, he/she will move to another interface (for example **reserve.dart** file) to enter his/her personal information or generate an alert dialog box.

# Lesson 9: Firebase

<b>Introduction .....</b>	9-2
<b>What is the JSON ? .....</b>	9-3
<b>How does Firebase Database work? .....</b>	9-4
<b>Firebase authentication (Signup and Login to Flutter App) .....</b>	9-5
<b>Configure Your App to use Firebase Services .....</b>	9-17
Adding Firebase to your Android App .....	9-19
Adding Firebase to your iOS App .....	9-26
<b>Configuring Firebase Authentication .....</b>	9-33
Login to an App Using Firebase User Accounts.....	9-46
Logout Configuration .....	9-48
<b>Firebase Database .....</b>	9-53
Which database is right for your project? .....	9-53
Real Time Database .....	9-54
Cloud Firestore .....	9-63
<b>Lab 9 : Create a User Profile Interface using Firebase .....</b>	4-72

## Introduction

**Firebase** is a NoSQL document database that simplifies storing, syncing, and querying data for your Android, iOS, and web apps at a global scale. Its client libraries provide live synchronization and offline support, while its security features and integrations with the Firebase and Google Cloud platforms accelerate building truly server-less apps.

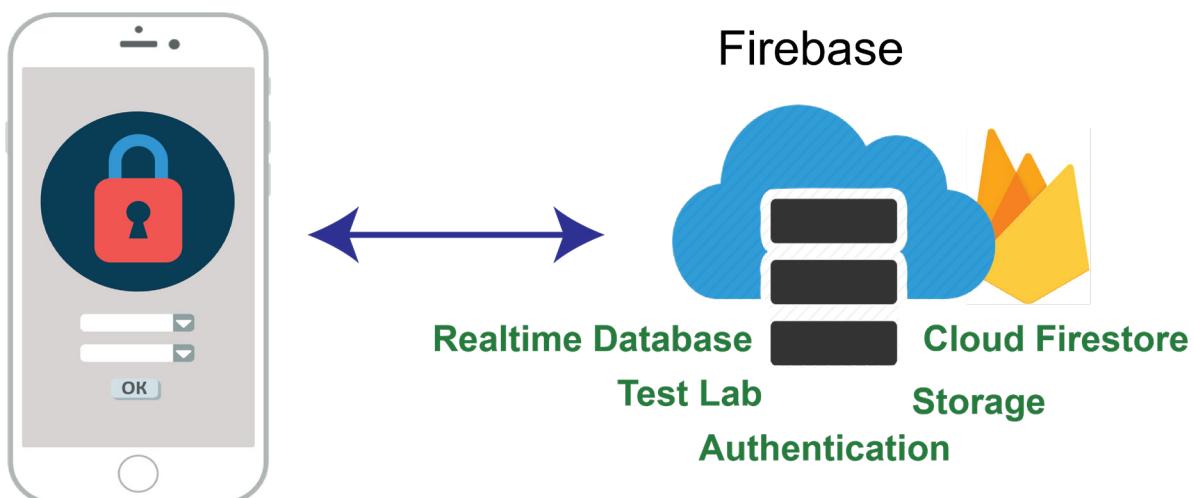
Firebase provides many products as follows:

### 1- Authentication

Most apps require to know the identity of the user. Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook, Twitter and more. In Firebase Authentication, developers can define who has access to what data, and how they can access it. In this lesson, you will learn how to configure your Flutter app by using this feature in order to create user accounts on Firebase and how to login to your app using these accounts.

### 2- Realtime Database

The Firebase Realtime Database is a cloud-hosted database. Data is stored as JSON and is synchronized in real time to every connected client. In this lesson, you will configure your app to user Firebase real time database to save app data and retrieve (query) the existing Firebase data.



### 3- Cloud Firestore

Cloud Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud Platform. Like Firebase Realtime Database, Cloud Firestore keeps your data in sync across client apps through real time listeners and offers offline support for mobile and web applications so you can build responsive apps that work regardless of network latency or Internet connectivity. Cloud Firestore also offers seamless integration with other Firebase and Google Cloud Platform products, including Cloud Functions. In this lesson, you will learn how to create a cloud Firestore database and how to use this data in Flutter app.

### 4- Cloud Storage

Cloud Storage for Firebase is a powerful, simple, and cost-effective object storage service built for Google scale. The Firebase SDKs for Cloud Storage adds Google security to file uploads and downloads for your Firebase apps, regardless of network quality. You can use our SDKs to store images, audio, video, or other user-generated content.

Firebase provides other services and products which are all important for mobile and web development domains such as hosting, cloud functions, test lab, ML Kit, analytics, predictions, and others. To get more details about other Firebase products, check the following web site: <https://firebase.google.com/products#develop-products>

The Firebase Data is stored as JSON and is synchronized in real-time to every connected client. Here is some information about JSON.

### What is the JSON?

**JSON** (JavaScript Object Notation) is a lightweight data-interchange format. People can easily learn to read and write JSON. It is based on a subset of the JavaScript Programming Language. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including Java, JavaScript, Kotlin, Dart and many others. These properties make JSON an ideal data-interchange language.

Google allows web sites, mobile apps, and other media to connect to its database and import it in JSON format. For example, Google maps database is available to import or use by iOS, Android, and web apps in JSON format. Also, these apps establish connection with Google database using https connection and API key as you will see in the next lesson (Using GPS and Google Maps).

You can customize your import of the Google database by selecting whatever object you want it to appear on your app content .

JSON objects are surrounded by curly braces {} and they are written in key/value pairs.

Keys must be string and values must be a valid JSON data type (string, number, array, Boolean or null).

In this lesson, you will get more information about how to configure your app to use JSON in access Firebase database.

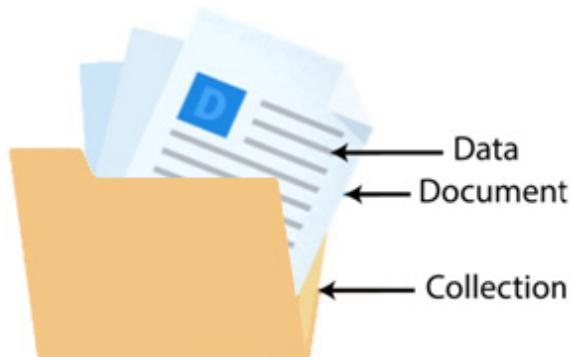
## How does Firebase Database work?

Firebase has many products and services. In this lesson, we will focus on Firebase authentication and database because these services are mostly used with mobile apps. In the connection between the mobile app and the Firebase database, the Firebase uses data synchronization instead of typical HTTP requests every time data changes. Any connected device receives that update within milliseconds.

Firebase apps remains responsive even when it is offline because the Firebase Database SDK keeps your data on the device storage. When the device regains connection, the Database synchronizes the local data changes with the remote updates that occurred while the client was offline, merging any conflicts automatically.

The Firebase Database can be accessed directly from a mobile device or web browser. Therefore, there is no need for an application server. Security and data validation are available through the Firebase Database Security Rules and expression-based rules that are executed when data is read or written.

Unlike an SQL database, in Firebase database, there are no tables or rows. Instead, you store data in *documents*, which are organized into *collections*. Each document contains a set of key-value pairs (data) as you will see later in this lesson.



The Firebase Database lets you build rich, collaborative applications by allowing secure access to the database directly from client-side code.

The Firebase Database is a NoSQL database and as such has different optimizations and functionality compared to a relational database. The Firebase Database API is designed to only allow operations that can be executed quickly. This enables you to build great real-time experience that can serve millions of users without compromising on responsiveness. Because of this, it is important to think about how users need to access your app data and then structure it accordingly.

When you work with Firebase database, you will find there are two types of Firebase database, Realtime Firebase database and Cloud Firestore. Later in this lesson, you will know in details about the difference between Firebase Realtime database and Cloud Firestore.

In this lesson, we are going to mostly be concerned with cloud Firestore which is the latest way of using Firebase to store data in the cloud. We are also going to be using the authentication package so that you can design an app, which allows app users to register their accounts (Signup) and login to this app using their credentials (User Login).

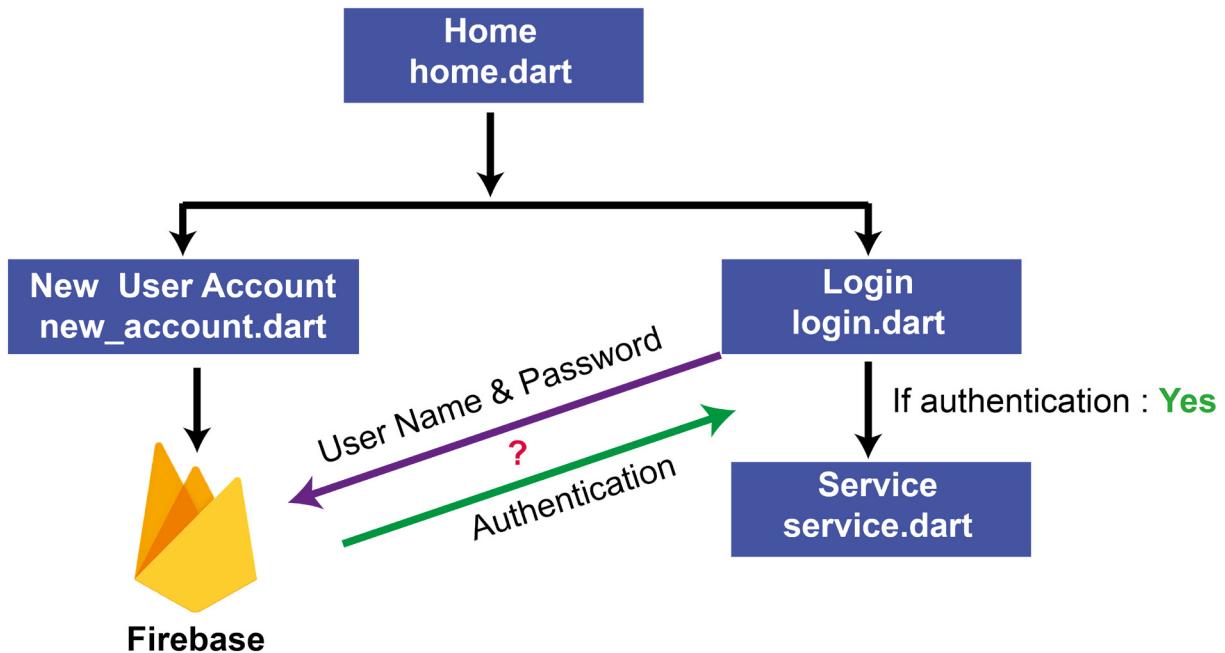
## **Firebase authentication (Signup and Login Flutter App)**

In this lesson, you will create a simple Flutter app using the Firebase authentication package to allow the app users to create a user account (signup) on Firebase and use this account to login to a specific part of this app such as use a specific service.

Most apps need to know the identity of a user. Knowing a user's identity allows an app to securely save his/her data in the cloud and provide the same personalized experience across all of the user's devices.

Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more.

The following figure includes a diagram of the Flutter app which you will create :



As you see in the previous diagram, you will create a home interface (**home.dart**) which includes two buttons. The first button is (New User Account) to create a new user account . This account consists of a user name (email) and password which will be created at Firebase web. The second button in **home.dart** will be for login (Login). When the app user taps the login button, he/she will be asked to enter the user name and the password which the app user has created before. Then, when the app user taps the Login button, this credential information (user name & password) will be sent to Firebase to check the authentication. If they are correct, then the user can login or move to another app interface (**service.dart**) to use the app services which are in the interface (Service Interface).

Note: In this app, you will use a **TextField** widget. If you need more details about using this widget, please review lesson 8.

To create this Flutter app, perform the following steps:

1- Open **Android Studio**

2- Click **File** → **New** → **New Flutter Project**

3- Select **Flutter Application** , then click **Next**.

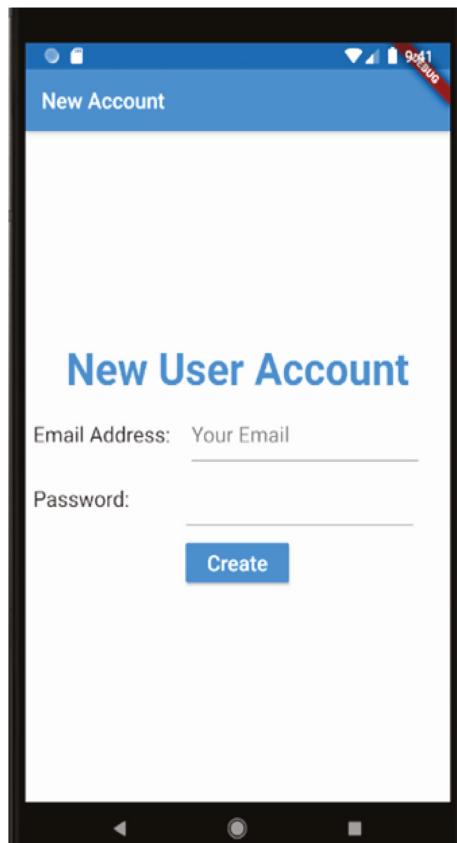
4- Type : **firebase\_authentication** for **Project Name**, and create a new folder : **Lesson\_09** for **Project Location**. Click **Next**.

5- Type : **androidatc.com** for Company domain, and then click **Finish**

6- Now, you should create three new interfaces (Dart files). Right click the **lib** folder, and select: **New → Dart File**. Type home for the file name , and then click OK.

7- Repeat the previous step to create another two dart files **new\_account.dart** and **service.dart**.

8- In the **new\_account.dart** file, create an interface having two text fields to ask the app user to enter his/her username, password, and then click **create button** as illustrated in the following figure:



The code of **new\_account.dart** which will create this interface follows:

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

class NewAccount extends StatefulWidget {
  @override
  _NewAccountState createState() => _NewAccountState();
}

class _NewAccountState extends State<NewAccount> {
```

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: Text('New Account '),
      ),
      body: Padding(
        padding: const EdgeInsets.all(8.0),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            ListView(
              shrinkWrap: true,
              children: <Widget>[
                Container(
                  alignment: Alignment.center,
                  child: Text(
                    'New User Account',
                    style: TextStyle(
                      fontSize: 40,
                      color: Colors.blue,
                      fontWeight: FontWeight.bold,)),
                ),
              ],
            ),
            SizedBox(height: 10.0,),

// ***** User Name *****
Row(
  children: [
    Text('Email Address:',style:TextStyle(fontSize: 20.0),),
    SizedBox(width: 20.0,),
    SizedBox(width: 220.0,
      child: TextField(
        onChanged: (value) {},
        style: TextStyle(fontSize: 20, color: Colors.blue),
        keyboardType: TextInputType.emailAddress,
```

```
        textInputAction: TextInputAction.done,
        autocorrect: false,
        cursorColor: Colors.red,
        decoration: InputDecoration(
            hintText: 'Your Email',
            ),
            ),
            ),
            ],
        ),
    ),

SizedBox(height: 10.0,),

// ***** Password *****
Row(
    children: [
        Text('Password:      ',style: TextStyle(fontSize: 20.0),),
        SizedBox(width: 20.0,),
        SizedBox(width: 220.0,

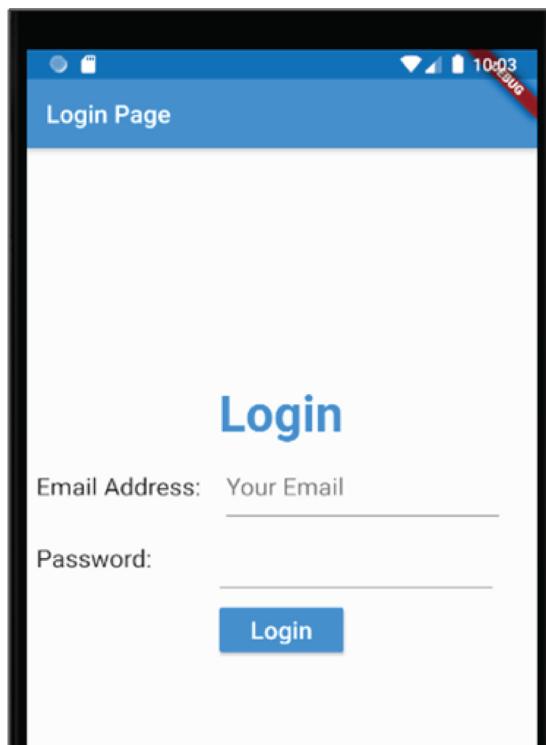
            child: TextField(
                onChanged: (value) {},
                //obscureText: true,
                style: TextStyle(fontSize: 20, color: Colors.blue),
                textInputAction: TextInputAction.done,
                autocorrect: false,
                ),
                ),
                ],
            ),
        )

SizedBox(height: 10.0,),

// ***** Create Button *****
Center(
    child: Container(
        width: 100,
        child:RaisedButton(
            color: Colors.blue,
            child: Text("Create",
```

```
        style:TextStyle(fontSize:20,color: Colors.white),),
        onPressed: () {}),
      ),
    ),
  ],
),
),
],
),
),
),
),
),
),
),
),
),
);
}
}
```

9- Open **login.dart** file and add the code which creates a login interface. The app user will use this interface to login using the user name and password which he/she has created in the create interface. The login interface is illustrated in the following figure:



You may copy the code of the **new\_account.dart** file and then paste it in the **login.dart**

file and change the classes names, title bar, and other text values which are illustrated in the grey highlighted color in the following code or type all the code:

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

class Login extends StatefulWidget {
    @override
    _Login createState() => _Login();
}

class _Login extends State<Login> {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(
                    title: Text('Login'),
                ),
                body: Padding(
                    padding: const EdgeInsets.all(8.0),
                    child: Column(
                        mainAxisAlignment: MainAxisAlignment.center,
                        children: <Widget>[
                            ListView(
                                shrinkWrap: true,
                                children: <Widget>[
                                    Container(
                                        alignment: Alignment.center,
                                        child: Text('Login',
                                            style: TextStyle(
                                                fontSize: 40,
                                                color: Colors.blue,
                                                fontWeight: FontWeight.bold,)),
                                    ),
                                ],
                            ),
                            SizedBox(height: 10.0,),
                        ],
                    ),
                ),
            ),
        );
}
```

```
// ***** User Name *****
Row(
  children: [
    Text('Email Address:',style:TextStyle(fontSize: 20.0),),

    SizedBox(width: 20.0,),
    SizedBox(width: 220.0,
      child: TextField(
        onChanged: (value) {},
        style: TextStyle(fontSize: 20, color: Colors.blue),
        keyboardType: TextInputType.emailAddress,
       textInputAction: TextInputAction.done,
        autocorrect: false,
        cursorColor: Colors.red,
        decoration: InputDecoration(hintText: 'Your Email',
          ),
          ),
          ),
        ],
      ),
    ),

SizedBox(height: 10.0,),

// ***** Password *****
Row(
  children: [
    Text('Password: ',style: TextStyle(fontSize: 20.0),),
    SizedBox(width: 20.0,),
    SizedBox(width: 220.0,

      child: TextField(
        onChanged: (value) {},
        //obscureText: true,
        style: TextStyle(fontSize: 20, color: Colors.blue),
       textInputAction: TextInputAction.done,
        autocorrect: false,
        ),
        ),
        ),
      ],
    ),
```

```
        SizedBox(height: 10.0,),  
  
    // ***** Login Button *****  
  
    Center(  
        child: Container(  
            width: 100,  
            child:RaisedButton(  
                color: Colors.blue,  
                child: Text("Login",  
                    style:TextStyle(fontSize:20,color:  
Colors.white),),  
                onPressed: () {},  
            ),  
        ),  
    ),  
    ),  
    ),  
    ],  
    ),  
    ],  
    ),  
    ),  
    ),  
    ),  
    ),  
    );  
}  
}
```

10- Open the **service.dart** file and type the following code. This interface should include the app services which the user login for.

```
import 'package:flutter/material.dart';  
  
class Service extends StatefulWidget {  
    @override  
    _ServiceState createState() => _ServiceState();  
}  
  
class _ServiceState extends State<Service> {  
    @override  
    Widget build(BuildContext context) {
```

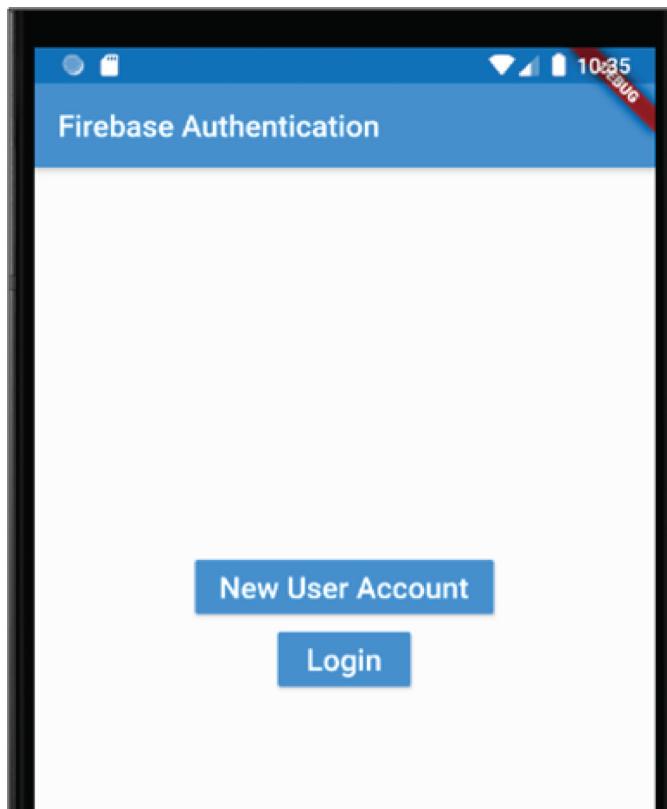
```
return MaterialApp(  
    home: Scaffold(  
        appBar: AppBar(  
            title: Text('Android ATC Services'),  
        ),  
  
        body: Center(  
            child: Container(  
                child: Text('Welcome to Android ATC',  
                    style: TextStyle(fontSize: 20.0,  
                )),  
            ),  
        ),  
    ),  
);  
}  
}
```

11- Open **main.dart** and delete all its content and then type the following code which includes the navigation routes. During the startup of the app, this main file will forward the app users to **home.dart** interface.

```
import 'login.dart';  
import 'new_account.dart';  
import 'package:flutter/cupertino.dart';  
import 'package:flutter/material.dart';  
import 'home.dart';  
import 'service.dart';  
  
void main() => runApp(MyApp());  
  
class MyApp extends StatefulWidget {  
    @override  
    _MyAppState createState() => _MyAppState();  
}  
  
class _MyAppState extends State<MyApp> {  
    @override  
    Widget build(BuildContext context) {
```

```
return MaterialApp(  
    home: Home(),  
  
    routes: {  
        'Login': (context) => Login(),  
        'NewAccount': (context) => NewAccount(),  
        'Service': (context) => Service(),  
    },  
);  
}  
}
```

12- Open **home.dart** file. This interface is the startup interface for your app as illustrated in the following figure. When the app user taps on the **New User Account** button, he/she will move to the **new\_account.dart** file and when the app user clicks on **Login** button, he/she will move to **login.dart**.



The code of **home.dart** follows:

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

class Home extends StatefulWidget {
    @override
    _HomeState createState() => _HomeState();
}

class _HomeState extends State<Home> {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(
                    title: Text('Firebase Authentication'),
                ),
                body: Center(
                    child: Column(
                        mainAxisAlignment: MainAxisAlignment.center,
                        children: <Widget>[
                            RaisedButton(
                                color: Colors.blue,
                                child: Text('New User Account',
                                    style: TextStyle(fontSize: 20, color: Colors.white),),
                                onPressed: () {
                                    Navigator.pushNamed(context, 'NewAccount');
                                },
                            ),
                            SizedBox(height:40.0,),

                            RaisedButton(
                                color: Colors.blue,
                                child: Text('Login',
                                    style: TextStyle(fontSize:20,color:Colors.white),),
                            )
                        ],
                    ),
                ),
            ),
        );
}
```

```
        onPressed: () {
            Navigator.pushNamed(context, 'Login');
        },
    ],
),
),
),
),
);
}
}
```

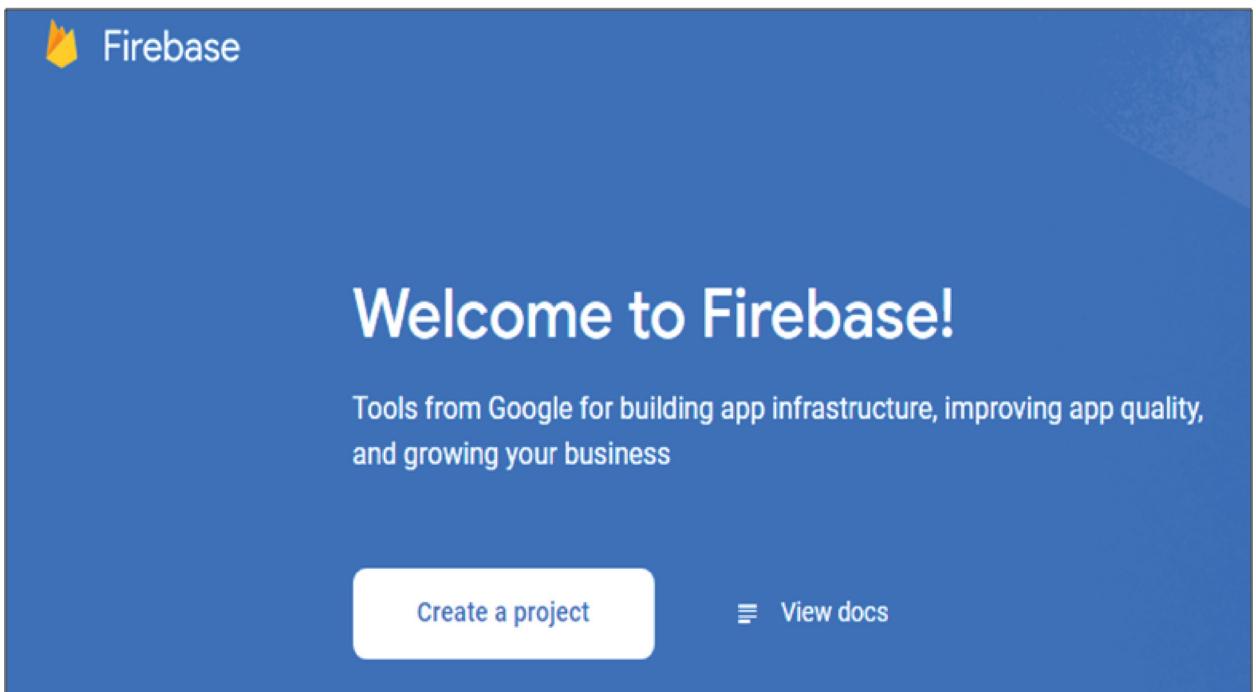
13- Run your app and check your navigation buttons (New User Account & Login).

## Configure Your App to use Firebase Services

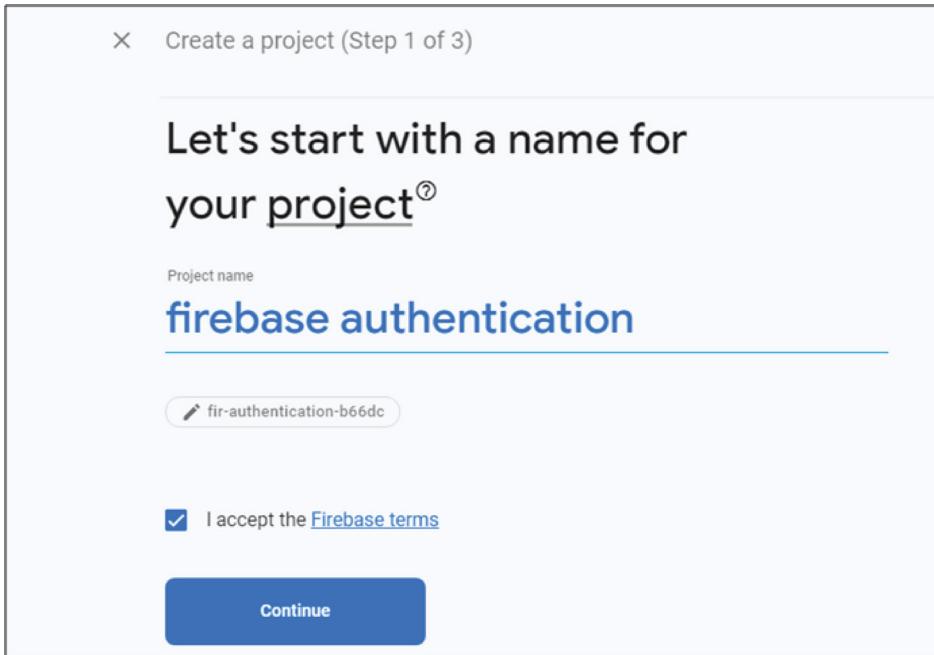
14- Go to : <https://console.firebaseio.google.com>

15- Sign into Firebase using your Google account.

16- You will get the following web page:

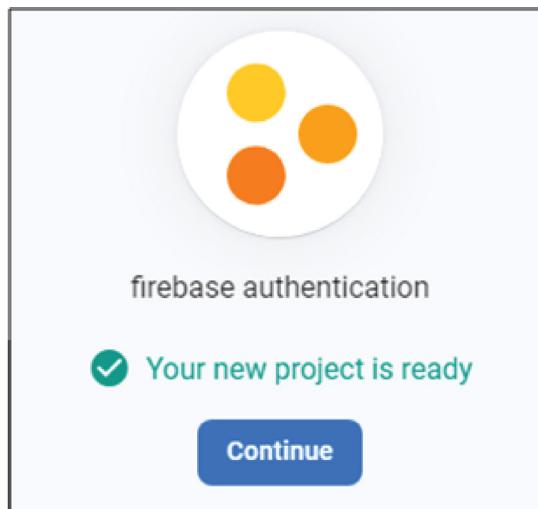


17- Click **Get Started** and click **Create a project** . Fill out your project name “firebase authentication” or any other name. Check **I accept the Firebase terms** and then click **Continue** as illustrated in the following figure:

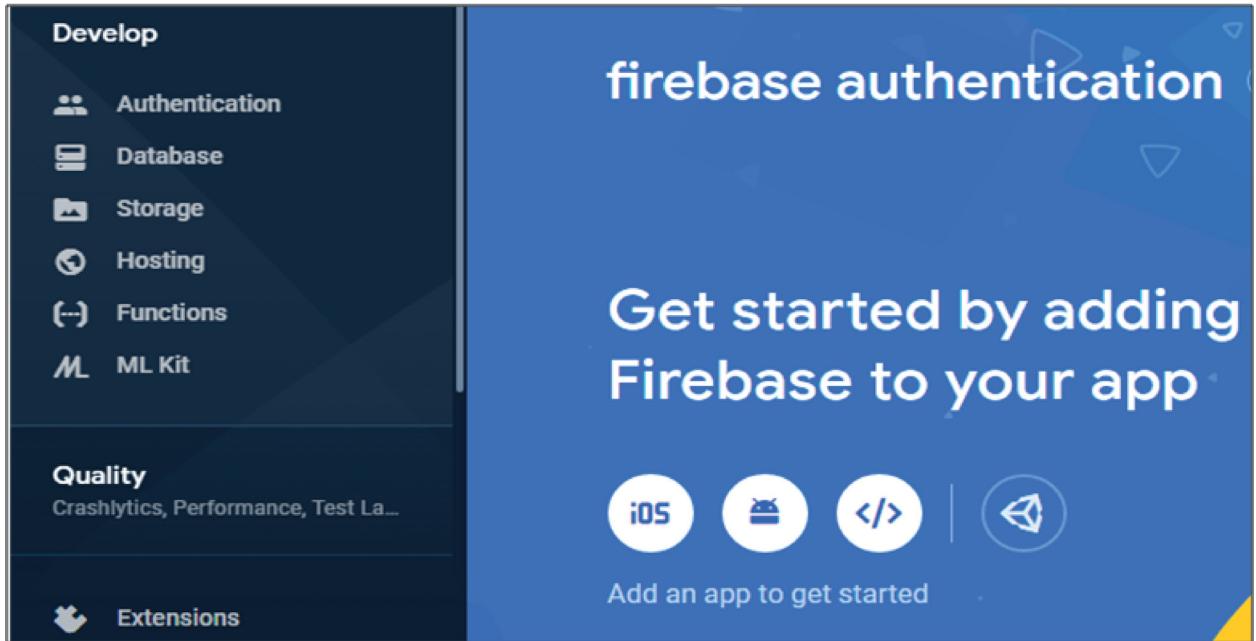


18- Click **Enable Google Analytics for this project to disable** this feature, because this app is just for testing how Firebase authentication works. However, these Google features are important for releasing a report about your live apps in the future.

Click **Create Project**. Then, after seconds, you should get the following message as illustrated in the following figure telling you that your project has been created on Google Firebase. Click **Continue**



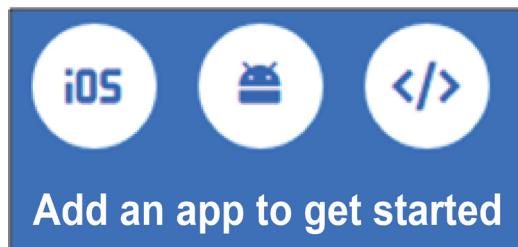
19- Now, as you see in the following figure, your project name: **firebase authentication** is created at the Firebase and if you click **Develop** on the left scroll frame, you will get all the Firebase services which you may use for your app.



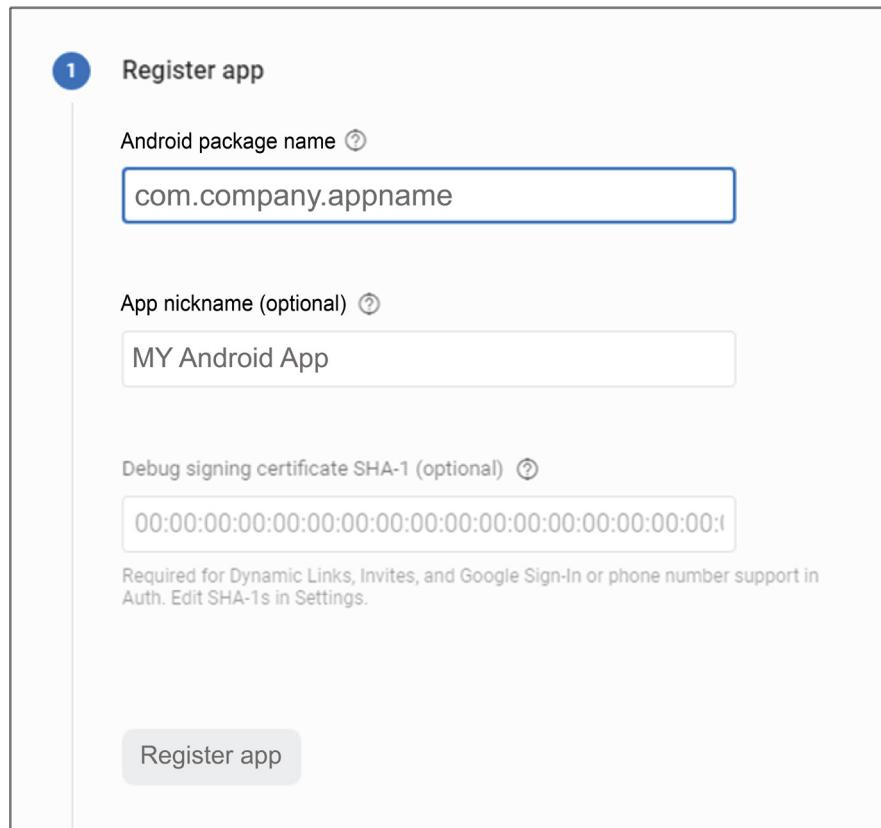
20- To add Firebase services to your Android, you should click the Android icon in the previous step, then click iOS icon to add these services to iOS app.

### Adding Firebase to your Android App

First, click the **Android icon** to add Firebase to your app.



You should get the following dialog box to register your app. As you will see below, you should enter your project name which must be a unique name in Google Play store.



If you forgot your app name, go to : **Android Studio** and open the following path:

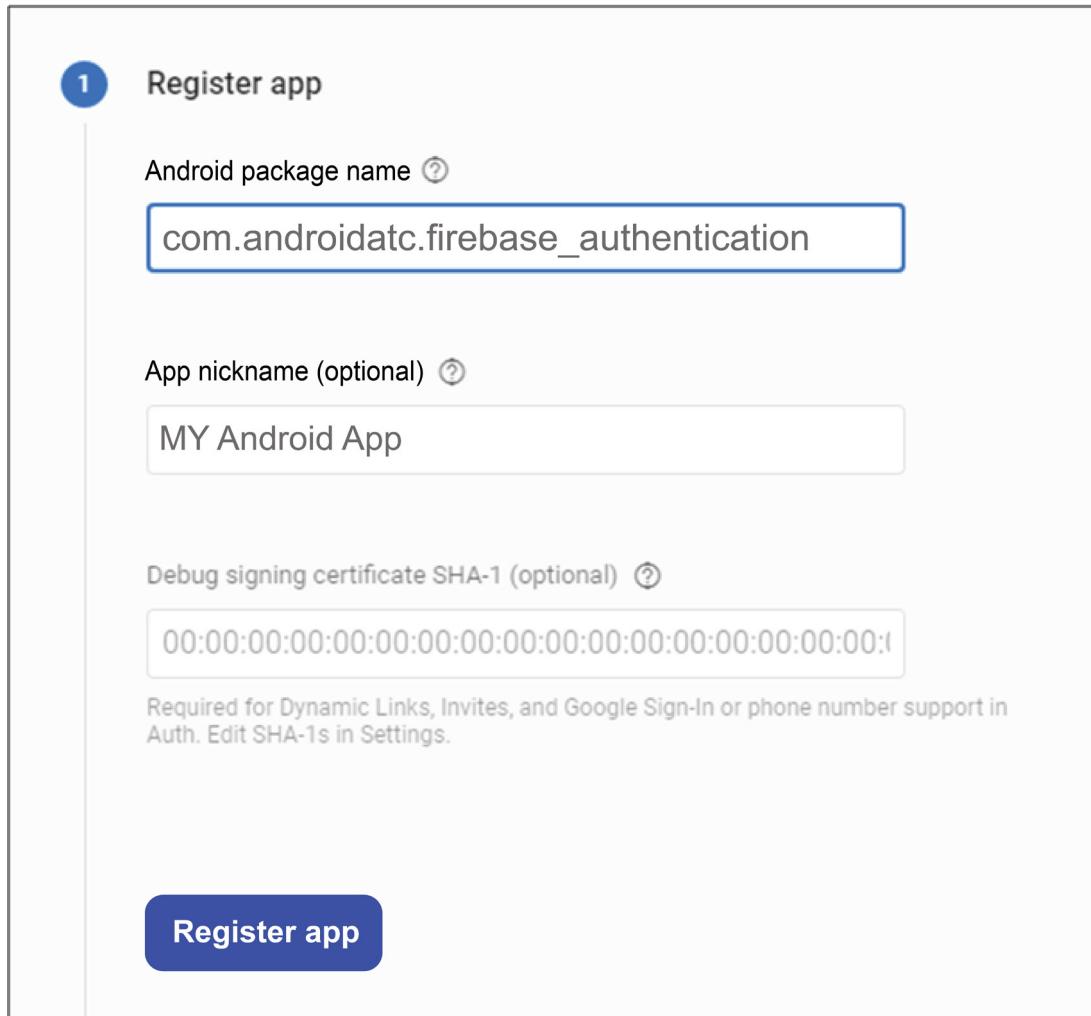
**Project name (firebase\_authentication\_android) → android → app → build.gradle**

Then, scroll down the file content, as illustrated in the grey highlighted part of the **application Id** in following code, your app name appears:

```
defaultConfig {
    // TODO: Specify your own unique Application ID (https://
developer.android.com/studio/build/application-id.html).
    applicationId "com.androidatc.firebaseio_authentication"
    minSdkVersion 16
    targetSdkVersion 28
    versionCode flutterVersionCode.toInt()
    versionName flutterVersionName
    testInstrumentationRunner "androidx.test.runner.
AndroidJUnitRunner"
}
```

Here, you found com.androidatc because when you created this Flutter app, you typed androidatc.com as your domain name. With your app, if you don't have a domain name, just type yourname.com or .net.

21- Copy the application Id, go to the [Firebase website](#), paste it in your **Android package name**, and then click **Register app** button as illustrated in the following figure:



22- In this step, as illustrated in the following figure, you should click: **Download google-service.json** button to download the JSON configuration file. This configuration file includes the https connection settings between your Android app and the Firebase services. If you download this file many times, you should use the file which has exactly this name : **google-services.json** without any number.

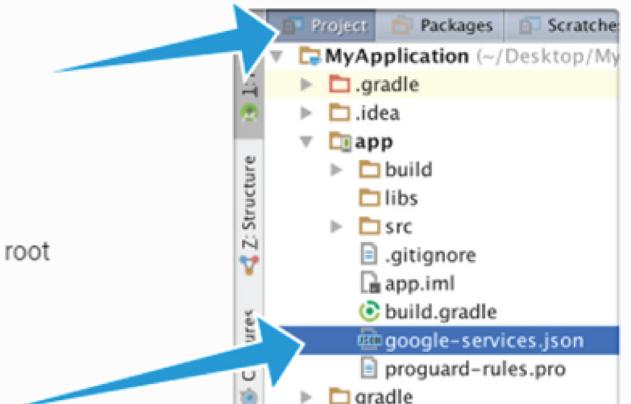
## 2 Download config file

Instructions for Android Studio below | [Unity](#) [C++](#)

[Download google-services.json](#)

Switch to the Project view in Android Studio to see your project root directory.

Move the google-services.json file you just downloaded into your Android app module root directory.



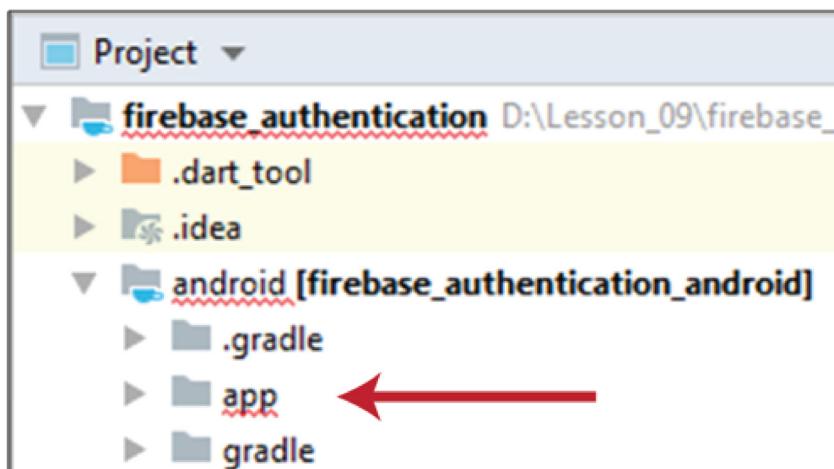
[Previous](#)

[Next](#)

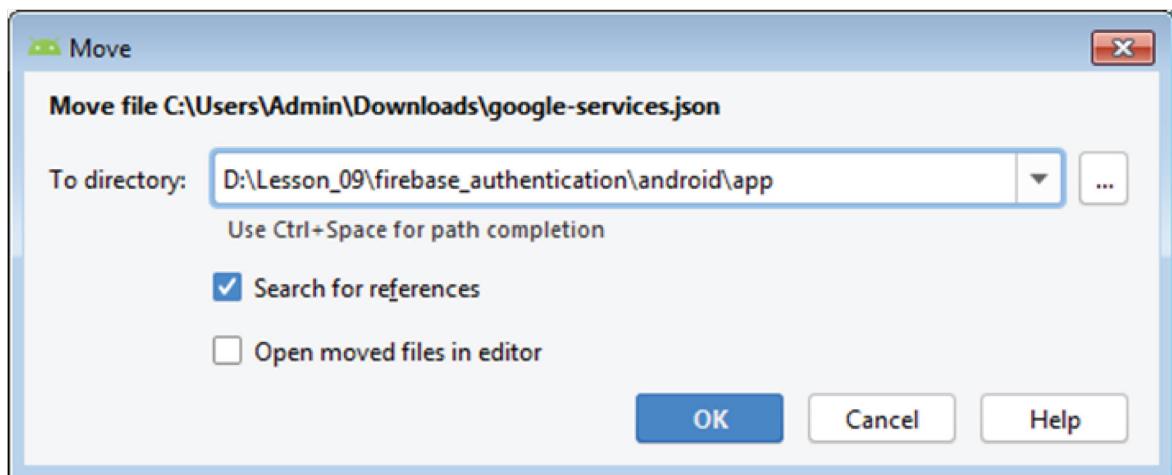
23- Move this file : google-services.json from your download folder to **app** folder using drag and drop technique. This **app** folder exists in Android Studio in the following path:

**Project name (firebase\_authentication) → android → app**

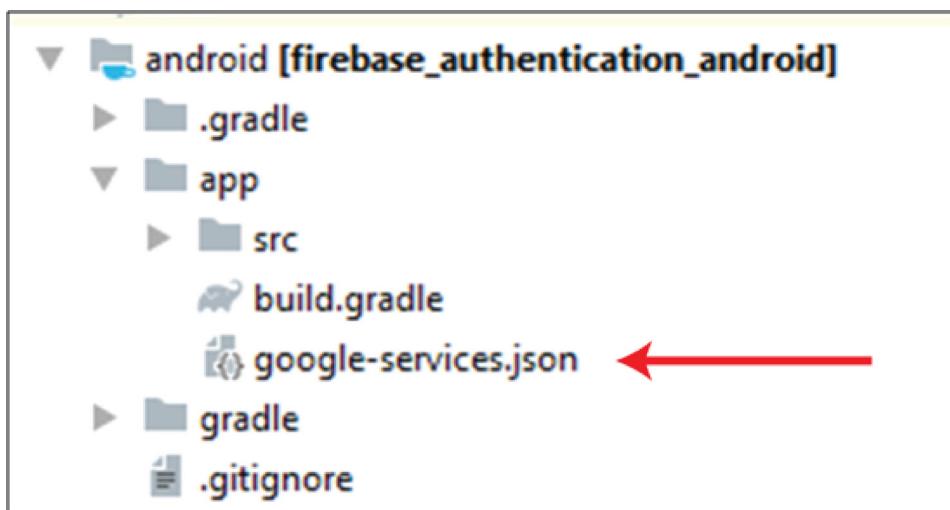
Or as illustrated in the following figure:



and then click **OK** to confirm moving as illustrated in the following figure:



If you expand the app folder, you will find this json file added to your Android app as illustrated in the following figure:



24- Now, return to the Firebase web site to complete the setup steps. Click **Next**.

25- In the Add Firebase SDK step , copy the line which is illustrated in the following figure, or click the copy icon.



26- In Android Studio, open the **build.gradle** file which exists in the following path :

Your Project name → **android** → **build.gradle**

And paste this class path within the dependencies braces as illustrated in the following figure:

```
dependencies {  
    classpath 'com.android.tools.build:gradle:3.5.0'  
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"  
    classpath 'com.google.gms:google-services:4.3.3'  
}
```

27- Now, return to the Firebase web site and from the configuration wizard, copy the following two lines :

```
apply plugin: 'com.android.application'  
apply plugin: 'com.google.gms.google-services'
```

Go to Android Studio, open the **build.gradle** file which is in the following path:

Your Project name → **android** → **app** → **build.gradle**

Then, paste the other two lines as separate lines at the end of this file as illustrated in the following figure:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test:runner:1.2.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.1'  
    implementation 'androidx.multidex:multidex:2.0.1'  
  
}  
apply plugin: 'com.android.application' ←  
apply plugin: 'com.google.gms.google-services' ←
```

Also, in the same **build.gradle** file add :**multiDexEnabled true** as illustrated in the grey highlighted part of the following configuration:

```
defaultConfig {  
  
    applicationId "com.androidatc.firebaseio_authentication"  
    minSdkVersion 16  
    targetSdkVersion 28  
    versionCode flutterVersionCode.toInt()  
    versionName flutterVersionName  
    testInstrumentationRunner "androidx.test.runner.  
AndroidJUnitRunner"  
    multiDexEnabled true  
}
```

28- Now, close all your Gradle files and be sure that your Android emulator is selected. Now, stop your app and then run your app again to apply the changes in your Android Studio Gradle files.

Also, it is better to restart your **Android Studio**.

29- Return to your Firebase configuration web page, click **Next** and you will get the following figure:

The screenshot shows the 'Add Firebase to your Android app' step in the Firebase setup process. It lists four steps: 'Register app' (completed), 'Download config file' (completed), 'Add Firebase SDK' (completed), and 'Read the Get Started Guide for Android' (step 4). Below the steps, a message says 'You're ready to start building with Firebase!' followed by instructions to follow the 'Firebase Getting Started Guide for Android' or continue to the console. Navigation buttons 'Previous' and 'Continue to console' are at the bottom.

× Add Firebase to your Android app

- 1 Register app  
Android package name: com.androidatc.firebaseio\_authentication
- 2 Download config file
- 3 Add Firebase SDK
- 4 Read the Get Started Guide for Android

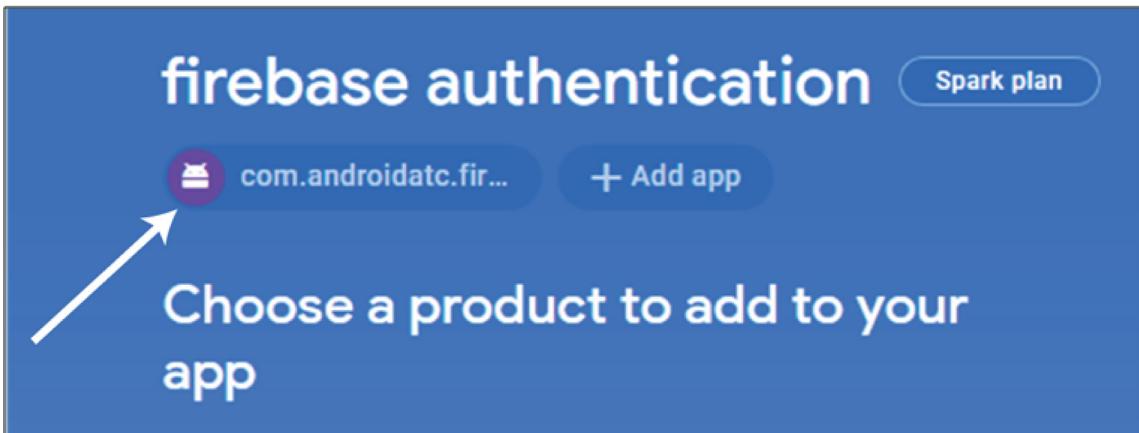
You're ready to start building with Firebase!

Follow the [Firebase Getting Started Guide for Android](#), where you'll find details about the various Firebase SDKs you can add to your app and more.

Or, continue to the console to explore Firebase.

Previous [Continue to console](#)

30 - Click **Continue to console**, then you will get the following figure:



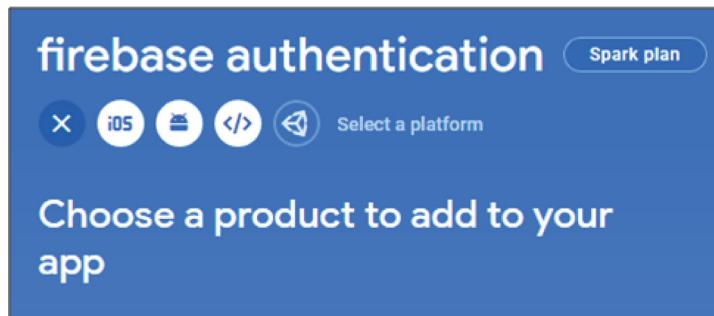
Your Flutter app is creating the Android and iOS apps simultaneously. Now, as you see, your Android app is configured with Firebase. Now, your Android app is ready to exchange or use the Firebase plug-in services, such as database or user authentication services.

### **Adding Firebase to your iOS App**

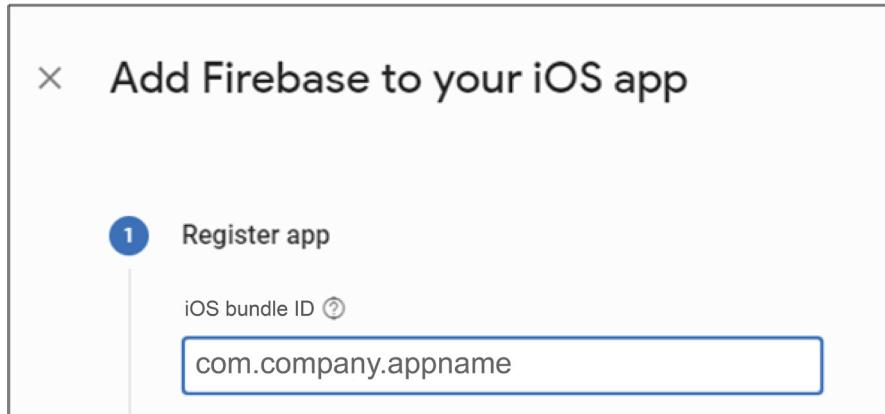
In the next step, you will configure the other part of your Flutter app which is iOS app with Firebase. If you have a Microsoft operating system (Windows) and it is not possible to complete the next steps using a Mac computer, you may skip the following steps until step **40** , or just try to do what you can without the need to run your app with an iPhone emulator ☺ .

31- Now, we will configure Firebase for your iOS app.

In Firebase web site, click **+Add app** button and then click **iOS** icon as illustrated in the following figure:



32- You will get the following wizard which asks you to enter the iOS bundle ID, or the app name.



Here, the iOS app name is Not the same name as Android package name. When you configured your Android app with Firebase , you used the following Android package name: **com.androidatc.firebaseio\_authentication** , but iOS doesn't accept the underscore symbol.

To find your iOS app name (iOS bundle name), open the following path in Android Studio:

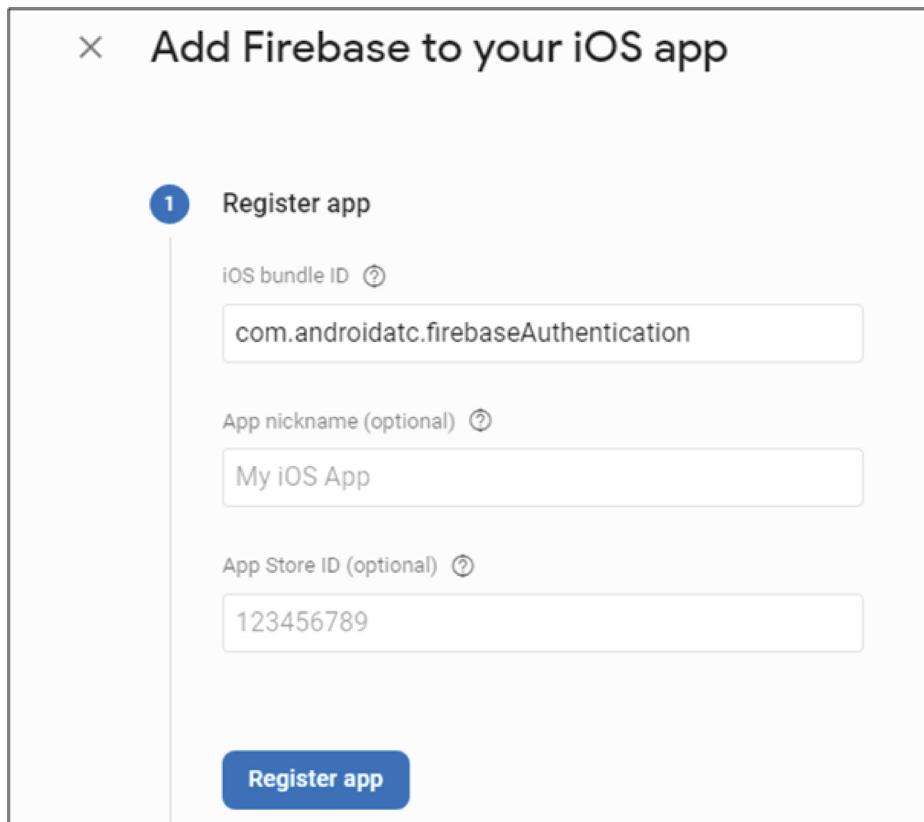
Your Flutter Project → **ios** → Runner.xcodeproj → xcshareddata → **project.pbxproj**

Double click this file : **project.pbxproj** , scroll down or press **Ctrl + F** to find the following :

**PRODUCT\_BUNDLE\_IDENTIFIER = com.androidatc.firebaseioAuthentication;**

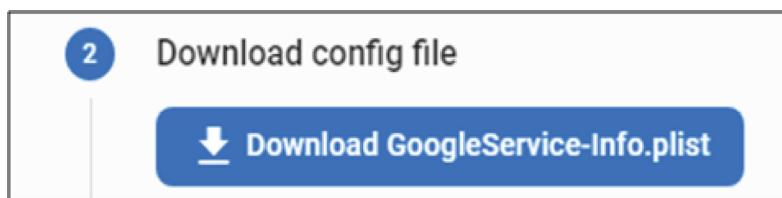
This means that the iOS bundle id is : **com.androidatc.firebaseioAuthentication** for this Flutter app.

33- Copy this iOS bundle id, and paste it in your Firebase and iOS app configuration wizard as illustrated in the following figure. Please don't add a space before or after this bundle id. Then, click **Register app** button.

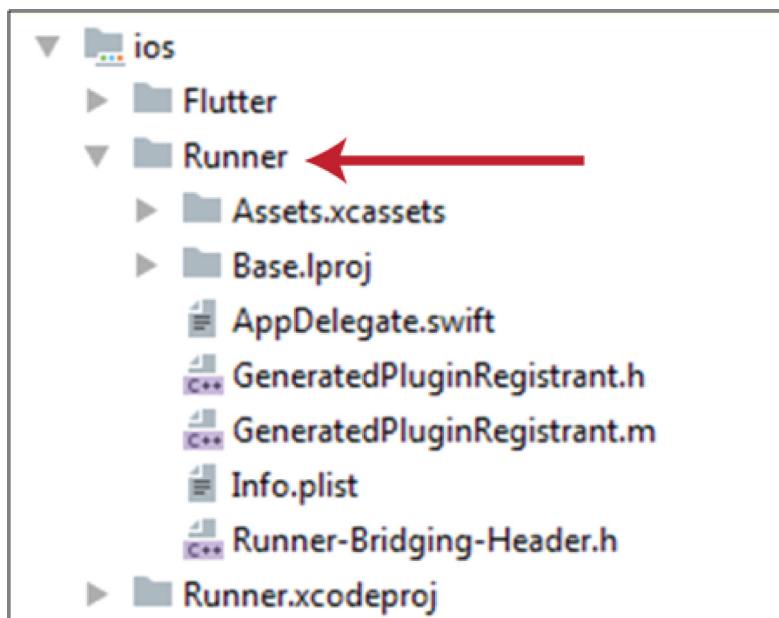


34- Now, as illustrated in the following figure, click the :

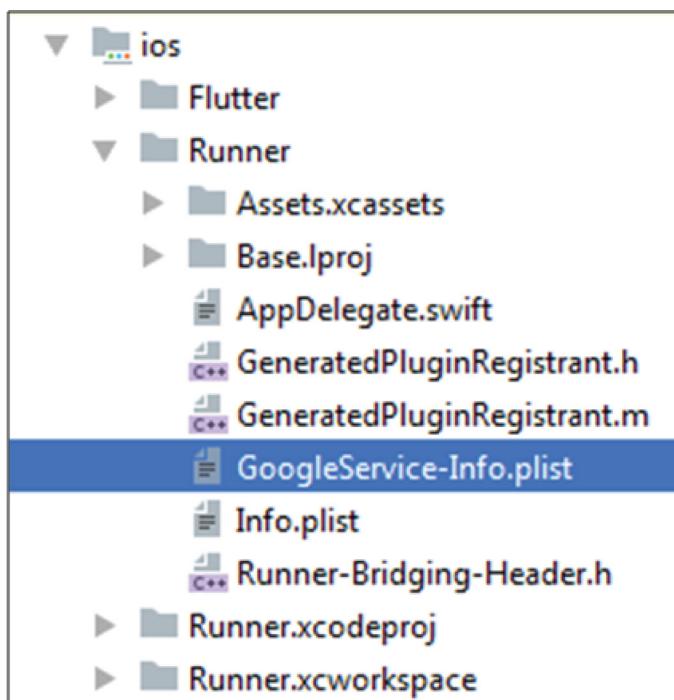
**Download GoogleService-Info.plist** button to download the Firebase configuration file.



35- Drag this downloaded file (**GoogleService-Info.plist**) to Android Studio and drop it inside the **Runner** folder which is illustrated in the following figure:



Then click **OK**. The iOS directory file structure will be as follows:



If you are using a Mac computer and want to update these changes, select your iPhone emulator, stop your app, and then run it again.

36- Return to Firebase configuration wizard in your web browser and click **Next**.

37-Now, in this step : **Add Firebase SDK** as illustrated in the following figure:

3 Add Firebase SDK Instructions for CocoaPods | [Download ZIP](#) [Unity](#) [C++](#)

Google services use [CocoaPods](#) to install and manage dependencies. Open a terminal window and navigate to the location of the Xcode project for your app.

Create a Podfile if you don't have one:

```
$ pod init
```

Open your Podfile and add:

```
# add pods for desired Firebase products
# https://firebase.google.com/docs/ios/setup#available-pods
```

Save the file and run:

```
$ pod install
```

This creates an .xcworkspace file for your app. Use this file for all future development on your application.

[Previous](#) [Next](#)

CocoaPods is a dependency manager for Swift and Objective-C which will help you to install third-party packages like Firebase to your Xcode project.

Because you are using Flutter, it is actually going to do this step for you automatically. So, you do NOT need to perform any of these commands. Instead, you may go back to your Android Studio, change your emulator type to the iPhone simulator (or connected iPhone physical device), then click run to build your app . You should get the following run result:

The screenshot shows the Android Studio interface with the 'Run' tab selected. The 'Console' tab is active, displaying the following log output:

```
Run: main.dart
Run: Console ⚡ 🚀
Launching lib/main.dart on iPhone XR in debug mode...
Running Xcode build...
Xcode build done. 22.2S
Syncing files to device iPhone XR ...
```

Return to your web browser (Add Firebase to your iOS app) and click **Next**.

38-In this step, (**Add initialization code**) and as you see in the following figure you should add two lines of code to : **AppDelegate.swift** file:

4 Add initialization code

To connect Firebase when your app starts up, add the initialization code below to your main **AppDelegate** class.

Swift  Objective-C

```
import UIKit
import Firebase

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

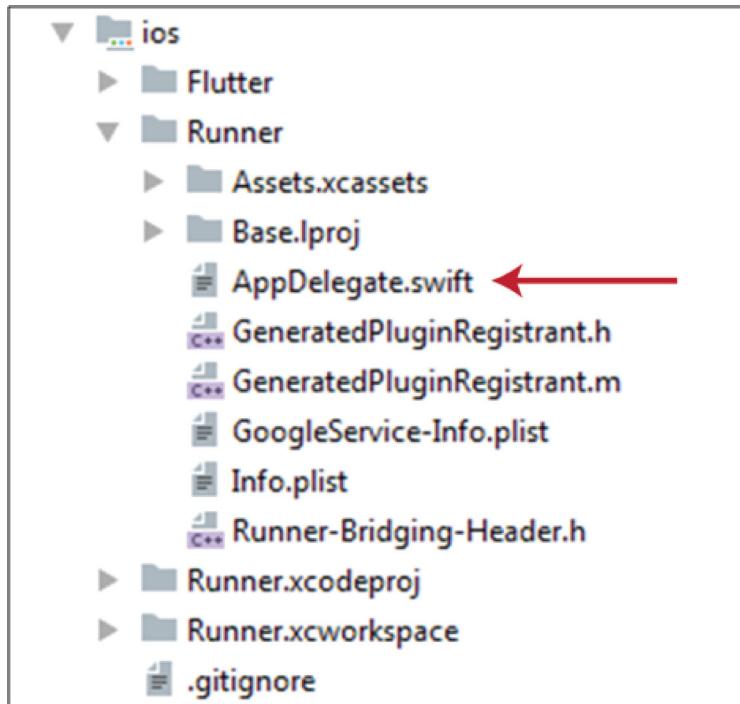
    var window: UIWindow?

    func application(_ application: UIApplication,
                    didFinishLaunchingWithOptions launchOptions:
                        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        FirebaseApp.configure()
        return true
    }
}
```

Previous

Next

Open this file which is in the following path:



From your web browser, copy the first line of code: **import Firebase**

In your **AppDelegate.swift** file, add this line of code as illustrated in the following figure:

```
1 import UIKit
2 import Firebase ←
3 import Flutter
4
5 @UIApplicationMain
```

Then, copy the second line of code from your web browser which is:  
`FirebaseApp.configure()`

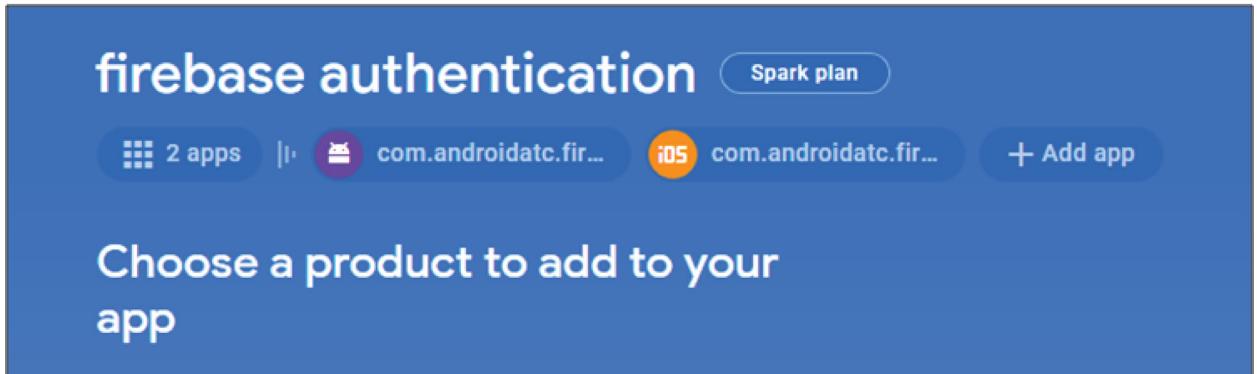
and paste it in **AppDelegate.swift** file as illustrated in the following screen shot:

```
) -> Bool {
    FirebaseApp.configure() ←
    GeneratedPluginRegistrant.register(with: self)
    return super.application(application, didFinishLaunchingWithOptions:
}
```

39- Return to your web browser, and click **Next**.

It is recommended to run your app again in your IPhone emulator to be sure that everything is configured right.

40- Open your web browser, and click **Continue to Console**. Now, you have finished: **Add Firebase to your iOS app** configuration wizard and you should have the following figure on your web browser which displays that you have configured Firebase to your Android and iOS app.



**Note:** To get more information about add Firebase to your iOS project, check the following web site: <https://firebase.google.com/docs/ios/setup>

Now, after configuring your Flutter app and Firebase to exchange data and services, it is time to start using the Firebase services. The first Firebase service which you will start using in this example is authentication as illustrated in the next section.

## Configuring Firebase Authentication

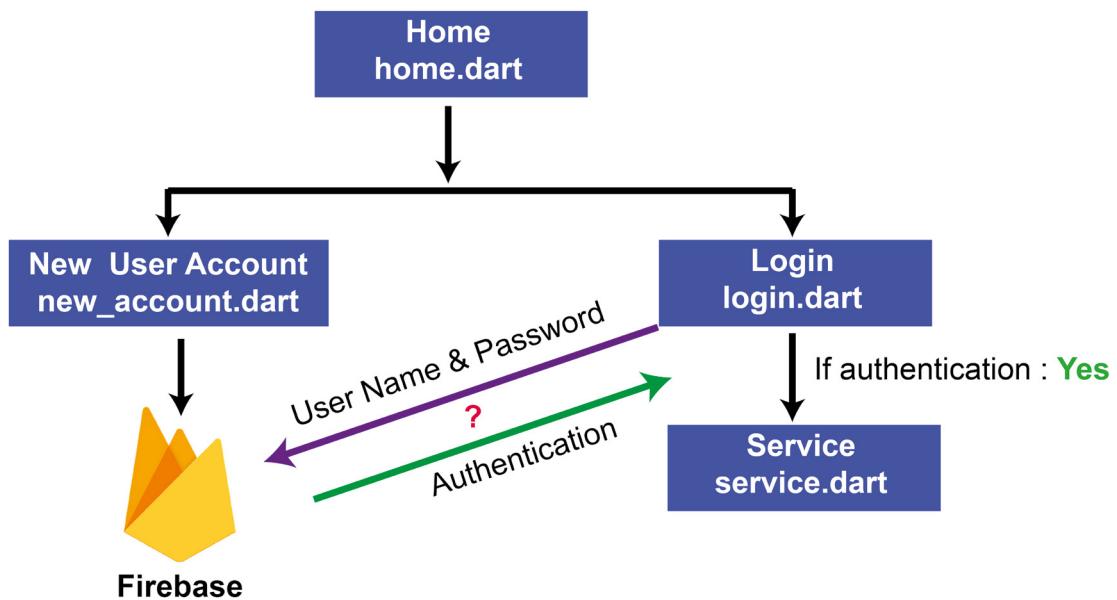
To sign a user into your app, you should first get authentication credentials from the user (Sing-up or Create a New User). These credentials can be the user's email address and password. Then, pass these credentials to the Firebase Authentication SDK. Firebase backend services will then verify those credentials and return a response to the client.

After a successful sign in, you can access the user's basic profile information and you can control the user's access to data stored in other Firebase products. Also, you can use the authentication tokens provided to verify the identity of users in your own backend services.



By default, authenticated users can read and write data to the Firebase Realtime Database and Cloud Firestore. You can control the access of those users by modifying your Firebase Realtime Database and Cloud Storage Security Rules.

Remember, the figure below explains how your Flutter app in this example works. In the next step, you will configure the **new\_account.dart** file Dart code to allow each app user to create a new user account (user name & password) at your Firebase Authentication web site.



To do that, you should add some Firebase plugins to your app as illustrated in the next steps.

41- The Firebase settings are always updated; therefore, to be up to date, you should follow the last setting in the following web link:  
<https://github.com/FirebaseExtended/flutterfire>

If this link is not available , just type it on Google search, then you will get the last updated link.

In this link, you will get all the information about the Firebase settings such as authentication , database and others. Go to :

<https://github.com/FirebaseExtended/flutterfire> and scroll down, then you will get the following figure:

Plugin	Version	Firebase feature	Source code
cloud_firestore	pub v0.13.3	Cloud Firestore	<a href="#">cloud_firestore</a>
cloud_functions	pub v0.4.2	Cloud Functions	<a href="#">cloud_functions</a>
firebase_admob	pub v0.9.1	Firebase AdMob	<a href="#">firebase_admob</a>
firebase_analytics	pub v5.0.11	Firebase Analytics	<a href="#">firebase_analytics</a>
firebase_auth	pub v0.15.4	Firebase Authentication	<a href="#">firebase_auth</a>
firebase_core	pub v0.4.4	Firebase Core	<a href="#">firebase_core</a>
firebase_crashlytics	pub v0.1.3	Firebase Crashlytics	<a href="#">firebase_crashlytics</a>
firebase_database	pub v3.1.1	Firebase Realtime Database	<a href="#">firebase_database</a>

Now, you should configure your app settings or add Firebase plug-in services to your app by configuring: **pubspec.yaml** file for Firebase authentication and database. Start with **firebase\_core** because all the other packages depend on it. Click **firebase\_core**, then click **Installing** tab as illustrated in the following figure:

**firebase\_core 0.4.4**

Published Feb 13, 2020 • flutter.dev 73 likes

FLUTTER ANDROID IOS WEB

Readme Changelog Example **Installing** Versions

42- Copy the following dependencies value or the last updated value which you will find :

**firebase\_core: ^0.4.4**

43- Open your **pubspec.yaml** file in your Android Studio and paste this value under the dependencies as illustrated in the following figure:

```
19   dependencies:  
20     flutter:  
21       sdk: flutter  
22       firebase_core: ^0.4.4
```

44- Click **Back** on your web browser toolbar to get the “**Available FlutterFire plugins**” web page again or this web like: <https://github.com/FirebaseExtended/flutterfire> , then click: **firebase\_auth**

45- Click the **Installing** tab, then copy the existing dependencies value:

`firebase_auth: ^0.15.4`

46- Paste this value in your **pubspec.yaml** file under the dependencies as illustrated in the following figure:

```
19   dependencies:  
20     flutter:  
21       sdk: flutter  
22       firebase_core: ^0.4.4  
23       firebase_auth: ^0.15.4  
24
```

47- Back on your web browser, to the “**Available FlutterFire plugins**” web page or this web link: <https://github.com/FirebaseExtended/flutterfire> , then click the : **firebase\_database**

48- Click the **Installing** tab, then copy the existing dependencies value:

`firebase_database: ^3.1.1`

49- Paste this value in your **pubspec.yaml** file under the dependencies as illustrated in the following figure:

```
19   dependencies:  
20     flutter:  
21       sdk: flutter  
22       firebase_core: ^0.4.4  
23       firebase_auth: ^0.15.4  
24       firebase_database: ^3.1.1  
25
```

50- Back on your web browser to the “**Available FlutterFire plugins**” web page or this web link: <https://github.com/FirebaseExtended/flutterfire> , then click the : **cloud\_firestore**

51- Click the **Installing** tab, then copy the existing dependencies value:  
**cloud\_firestore: ^0.13.3**

52- Paste this value in your **pubspec.yaml** file under the dependencies as illustrated in the following figure:

```
19     dependencies:
20       flutter:
21         sdk: flutter
22       firebase_core: ^0.4.4
23       firebase_auth: ^0.15.4
24       firebase_database: ^3.1.1
25       cloud_firestore: ^0.13.3
```

53- Now, at the top of the **pubspec.yaml** file content, click **Packages Get** to incorporate all of those settings into your Flutter project.

You should get the following result in the Messages console:

```
Messages: [firebase_authentication] Flutter ×
C:\src\flutter\bin\flutter.bat --no-color packages get
Running "flutter pub get" in firebase_authentication...          0.9s
Process finished with exit code 0
```

54- Now, it is important to test your app running on your emulators; therefore, select your Android emulator, stop your app, and then run it again.

Repeat the same thing for your iPhone emulator if you are using a Mac computer. If not, skip this step and follow the next steps.

**Note :** (For Mac users only): If you are using an old version of CocoaPods, you are recommended to update it to avoid any incompatibility problems in the last version of Firebase packages. To do that, type the following command in Android Studio terminal console:

```
pod repo update
```

This command may take some time !

After completing the installing the update, type the following command in the terminal console: **sudo gem install cocoapods**

Then, the final step: **pod setup**

Now, stop your iPhone emulator and run it again.

55- Now, it is the time to use Firebase authentication plug-in service in your app.

First, open the **new\_account.dart** (New User Account Interface) and declare the following three variables as illustrated in the following figure:

```
class _NewAccountState extends State<NewAccount> {  
  String email;  
  String password;  
  final FirebaseAuth _auth = FirebaseAuth.instance;  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        body: Center(  
          child: Column(  
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
            children: [  
              Text("Create New Account",  
                style: TextStyle(fontSize: 20, color: Colors.purple),  
              ),  
              Padding(  
                padding: EdgeInsets.all(10),  
                child: TextField(  
                  decoration: InputDecoration(labelText: "Email Address"),  
                ),  
              ),  
              Padding(  
                padding: EdgeInsets.all(10),  
                child: TextField(  
                  decoration: InputDecoration(labelText: "Password"),  
                ),  
              ),  
              Padding(  
                padding: EdgeInsets.all(10),  
                child: ElevatedButton(  
                  onPressed: () {  
                    // Implement sign-up logic here  
                  },  
                  child: Text("Sign Up"),  
                ),  
              ),  
            ],  
          ),  
        ),  
      ),  
    );  
  }  
}
```

**email**: This variable represents the value which the app user will fill out in the email address **TextField** in the new user app interface whose type is: **String** data type.

**password**: This variable represents the value which the app user will fill out in the password **TextField** in the new user app interface. The Firebase authentication password must be at least 6 characters and its type is: **String** data type.

**\_auth**: This object is a static instance of the **FirebaseAuth** class and it represents the app user name (email) and password. Here, you have initialized the **\_auth** Firebase instance which will be used later in creating and authenticating your Dart code .

56- In **new\_account.dart** file, configure the email and password variables as values for these two **TextField** widgets (email address & password) as illustrated in the following figures:

For user name (email address):

```
// ***** User Name *****
Row(
  children: [
    Text(
      'Email Address:',
      style: TextStyle(fontSize: 20.0),
    ), // Text
    SizedBox(width: 20.0,),
    SizedBox(
      width: 220.0,
      child: TextField(
        onChanged: (value1) {
          email = value1;
        },
        style: TextStyle(fontSize: 20, color: Colors.blue),
        keyboardType: TextInputType.emailAddress,
```

For Password:

```
// ***** Password *****
Row(
  children: [
    Text(
      'Password:         ',
      style: TextStyle(fontSize: 20.0),
    ), // Text
    SizedBox(width: 20.0,),
    SizedBox(
      width: 220.0,
      child: TextField(
        onChanged: (value2) {
          password = value2;
        },
```

Now, when the user taps the **Create** button, value 1 (email) & value 2 (password) who will be sent to the Firebase authentication users list.

57- In **new\_account.dart** file, add the following import package to your Dart code:

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

58 - Open your Firebase web site again. On the left side of your web browser under the **Develop** console, click **Authentication**. Then click **Sign-in method** tab as illustrated in the following figure:

The screenshot shows the Firebase console's Authentication section. On the left sidebar, under the 'Develop' heading, 'Authentication' is selected. The main page title is 'Authentication'. Below it, there are tabs for 'Users', 'Sign-in method' (which is underlined in blue), 'Templates', and 'Usage'. The 'Sign-in providers' section lists two providers: 'Email/Password' and 'Phone', both of which are currently 'Disabled'.

59 - Click **Email/Password**, click **Enable** switch button, then click **Save** as illustrated in the following figure:

This screenshot shows the 'Email/Password' sign-in provider settings. A large red arrow points to the 'Enable' switch button, which is currently turned on (blue). Below the provider name, there is descriptive text about email address verification and password recovery, followed by a 'Learn more' link. At the bottom right, there are 'Cancel' and 'Save' buttons, with 'Save' being highlighted in blue.

60 - Now, add the following code to the **onPressed{}** method which belongs to the **Create** button to register the app user.

```
 onPressed: () async {  
  
    try {  
  
        final newUser =  
            await _auth.createUserWithEmailAndPassword(  
                email: email, password: password);  
  
        if (newUser != null) {  
            Navigator.pushNamed(context, 'Home');  
        }  
  
    }  
  
    catch (e) {  
        print(e);  
    }  
,
```

### ***Explanation:***

#### **createUserWithEmailAndPassword Method:**

createUserWithEmailAndPassword ({String email, String password})

This method contains the library: **firebase\_auth** and class: **FirebaseAuth**. It tries to create a new user account with the given email address and password.

#### **async & await :**

**async** and **await** are keywords that work together. **async** can be placed before a function, which always returns a result. **async** is used to invoke an asynchronous code. While the keyword **await** works only inside **async** functions and it makes Dart code wait until this data settles and returns the result. Here, in this example, **async** performs asynchronous for creating the new user and **await** delays the **async** until this process is completed.

#### **try & catch**

**try** and **catch** are keywords that represent the handling of exceptions due to data or coding errors during Dart program execution. A **try** block is the block of code in which exceptions occur. A **catch** block catches and handles **try** block exceptions.

## If statement

In this code, if the newUser has a value (not empty), this means that after creating the new user account (email & password) on Firebase, the app user will move to home.dart .

61- The full code of **new\_account.dart** file follows:

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

class NewAccount extends StatefulWidget {
  @override
  _NewAccountState createState() => _NewAccountState();
}

class _NewAccountState extends State<NewAccount> {
  String email;
  String password;
  final FirebaseAuth _auth = FirebaseAuth.instance;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('New Account '),
        ),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              ListView(
                shrinkWrap: true,
                children: <Widget>[
                  Container(
                    alignment: Alignment.center,
                    child: Text('New User Account',
                      style: TextStyle(
                        fontSize: 40,
```



```
        style: TextStyle(fontSize: 20, color: Colors.blue),
        textInputAction: TextInputAction.done,
        autocorrect: false,
            ),
            ),
        ],
    ),
),

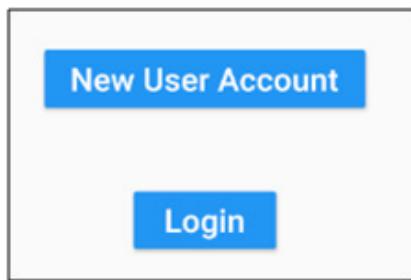
SizedBox(height: 10.0,),

// ***** Create Button *****
Center(
    child: Container(width: 100,
        child: RaisedButton(color: Colors.blue,
            child: Text("Create",
                style: TextStyle(fontSize: 20, color: Colors.white),
            ),
            onPressed: () async {
                try {
                    final newUser =
                        await _auth.createUserWithEmailAndPassword(
                            email: email, password: password);

                    if (newUser != null) {
                        Navigator.pushNamed(context, 'Home');
                    }
                } catch (e) {print(e);}

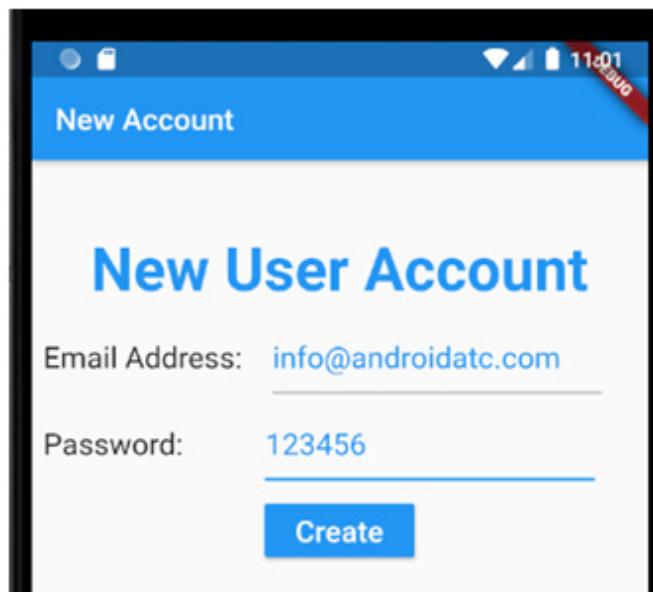
                },
                ),
                ),
                ],
            ),
            ],
        ),
        ),
        );
    }
}
```

62- Run your app and click: **New User Account** button to move to the **new\_account.dart** interface.



63- Type your email address and any password (at least 6 characters) and tap the Create button as illustrated in the following figure:

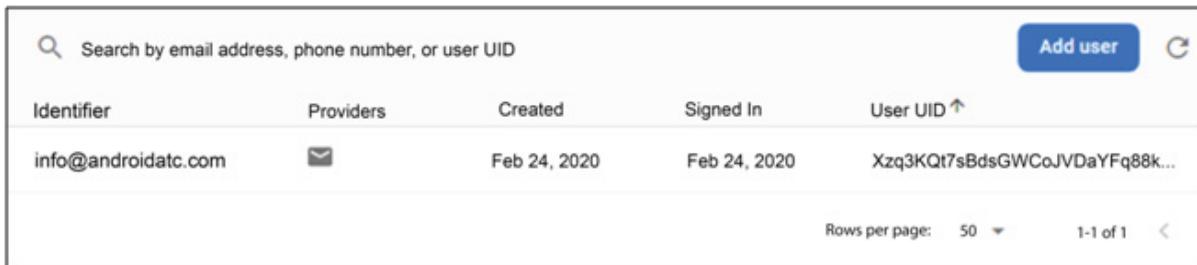
**Important Note:** In the login interface, don't add a space before or after your email address. Also, if you press the **Tab** key to move from the email address text field to enter the password login, there will be an extra space added to your email address. This will cause an error message in your Android Studio run console such as : (ERROR\_INVALID\_EMAIL, The email address is badly formatted., null).



The app user will be moved to the **home.dart** interface. In the next step, you will configure the **login.dart** (Login interface) to use this user account (email & password) which you created in the login process.

64- Back to your Firebase web site, in the Authentication part, click the **Users** tab. You should find a new user account that has been created as illustrated in the

following figure:



A screenshot of the Firebase console's 'Users' table. The table has columns: Identifier, Providers, Created, Signed In, and User UID. One row is visible for the email 'info@androidatc.com', which was created and signed in on Feb 24, 2020. The User UID is 'Xzq3KQt7sBdsGWC0jVDaYFq88k...'. At the top right, there is a search bar, an 'Add user' button, and a refresh icon. At the bottom right, there are buttons for 'Rows per page:' (set to 50), '1-1 of 1', and a back arrow.

Identifier	Providers	Created	Signed In	User UID ↑
info@androidatc.com	✉	Feb 24, 2020	Feb 24, 2020	Xzq3KQt7sBdsGWC0jVDaYFq88k...

## Login to App Using a Firebase User Accounts

Now , you will configure the **login.dart** interface to allow users which you created in Firebase to login to your app. Also, if the users login successfully, then they will be moved to the **services.dart** interface.

65- Open **login.dart** file and import the following Firebase package:

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

66- Configure the same previous two variables (email & password) and the **\_auth** Firebase instance as illustrated in the grey highlighted color in the following code:

```
class _Login extends State<Login> {
  String email;
  String password;
  final FirebaseAuth _auth = FirebaseAuth.instance;

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

67- Configure the **TextField** of the Email address as follows:

```
child: TextField(
  onChanged: (value1) {
    email = value1;
  },
```

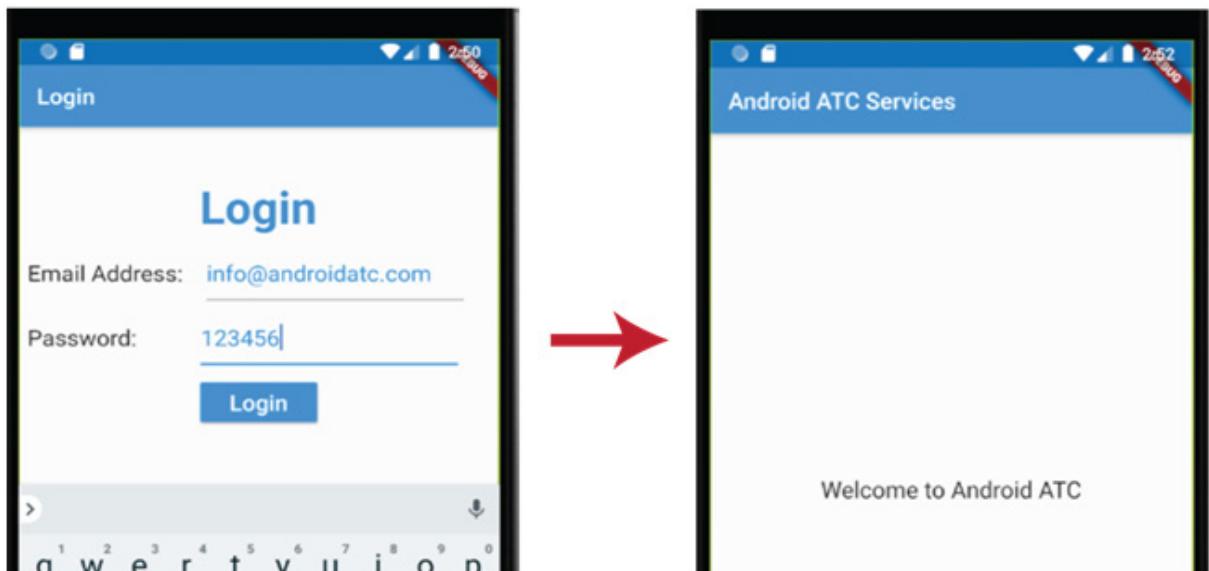
and Configure the **TextField** of the password as follows:

```
child: TextField(  
    onChanged: (value2) {  
        password= value2;  
    },
```

68- Now to configure **Login** button to authenticate the app user credentials (email & password), you should configure this button **onPressed** method in the same way which you used to configure the user account creation step; however, here you will use **signInWithEmailAndPassword** method in the login process as illustrated in the following code:

```
child: RaisedButton(  
    color: Colors.blue,  
    child: Text("Login",  
        style: TextStyle(fontSize: 20, color: Colors.white),),  
    onPressed: () async {  
        try {  
            final User = await _auth.signInWithEmailAndPassword(  
                email: email, password: password);  
  
            if (User != null) {  
                Navigator.pushNamed(context, 'Service');  
            }  
        } catch (e) {  
            print(e);  
        }  
    },  
,
```

69- Run your app, tap the **Login** button, enter the email and password which you created on Firebase in the previous section, then tap the **Login** button. The app user should login successfully, and the user will be moved to the **service.dart** app interface as illustrated in the following figures:



If you login in at a different date of the user creation date, you should find the **Signed In** field in the **Users** tab of Firebase authentication web page has the date of the user's sign in.

## Logout Configuration

Now, after completing the login step successfully, the app user will get the **service.dart** interface where all your app services are. Keep in mind, your app user will expect to find a button or icon that has a logout action added to this service interface. To add a logout icon to your Flutter app interface, perform the following steps:

70- Open **service.dart** file.

71- Import the Firebase authentication package to this file using the following command at the top of the code :

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

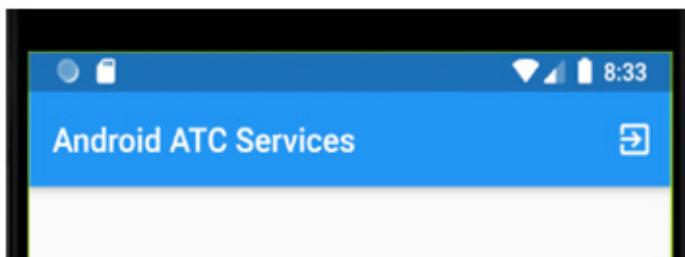
72- Declare the **auth** Firebase instance as illustrated in the grey highlighted color in **service.dart** file :

```
class _ServiceState extends State<Service> {  
  final FirebaseAuth auth = FirebaseAuth.instance;  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      home: Scaffold(  
        body: Center(  
          child: Container(  
            padding: EdgeInsets.all(20),  
            child: Column(  
              mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
              children: [  
                Text('Welcome to'),  
                Text('Android ATC Services'),  
                Text('Please log in'),  
                Text('to continue'),  
              ],  
            ),  
          ),  
        ),  
      ),  
    );  
  }  
}
```

73- Add to this file title bar a logout icon (`exit_to_app`) by adding the following code:

```
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(title: Text('Android ATC Services'),  
        actions: <Widget>[  
          IconButton(  
            icon: Icon(Icons.exit_to_app),  
            onPressed: () { },  
          ),  
        ],  
      ),  
      body: Center(  
        child: Container(  
          padding: EdgeInsets.all(20),  
          child: Column(  
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
            children: [  
              Text('Welcome to'),  
              Text('Android ATC Services'),  
              Text('Please log in'),  
              Text('to continue'),  
            ],  
          ),  
        ),  
      ),  
    ),  
  );  
}
```

74- Run your app again, tap **Login** button, enter your user and password , then tap **Login**. You should get the following icon in the **service.dart** interface:



75- Now, add the `auth.signOut()` method as an action to the icon `onPressed()` method. Also, when the app user taps the logout icon, he/she will be moved to the previous interface where this user was (login interface). The full code of `service.dart` follows:

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

class Service extends StatefulWidget {
    @override
    _ServiceState createState() => _ServiceState();
}

class _ServiceState extends State<Service> {
    final FirebaseAuth auth = FirebaseAuth.instance;

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text('Android ATC Services'),
                actions:<Widget>[
                    // action button
                    IconButton(
                        icon: Icon(Icons.exit_to_app),
                        onPressed: () {
                            auth.signOut();
                            Navigator.pop(context);
                        },
                    ),
                ],
            ),
            body: Center(
                child: Container(
                    child: Text('Welcome to Android ATC',
                        style: TextStyle(fontSize: 20.0,)),
                ),
            ),
        );
    }
}
```

76- Run your app again. Login to this **service.dart** interface using your user name and password, which you have created at Firebase web site previously.

Now, before you tap the logout icon, open the Run console and focus on the messages that will be generated when you tap this logout icon. Now, tap this logout icon. You will logout of this interface and get the following messages in the Run console:

```
D/FirebaseAuth(17081): Notifying id token listeners about a sign-out event.  
D/FirebaseAuth(17081): Notifying auth state listeners about a sign-out event.
```

Also, you will be moved to the previous interface. But it does not end here as you can see the login interface clearly still has the user name (email) and password .

You can make this password appear as (\*\*\*\*) by adding the following property to the password **TextField** in the **login.dart** as follows:

```
obscureText: true,
```

But, also when the user logs out form the service interface, he/she will find the password value still existing.

77- The best design is removing the password which the app user typed in the Password **TextField** after he/she taps the Login button. To do that , open **login.dart** file, and add the grey highlighted part of the following code which declares the object **\_controller** using the **TextEditingController** class.

```
class _Login extends State<Login> {  
  String email;  
  String password;  
  final FirebaseAuth _auth = FirebaseAuth.instance;  
  
 TextEditingController _controller = new TextEditingController();  
  
  @override  
  Widget build(BuildContext context) {
```

78- In the **login.dart**, add the following grey highlighted part of the following code in the password **TextFiled** widget :

```
Row(  
  children: [  
    Text('Password:      ',  
        style: TextStyle(fontSize: 20.0),  
    ),  
    SizedBox(  
      width: 20.0,  
    ),  
    SizedBox(  
      width: 220.0,  
      child: TextField(  
  
        controller: _controller,  
  
        onChanged: (value2) {  
          password = value2;  
        },
```

79- In the **login.dart**, add the following grey highlighted part of the following code in the **onPress()** method of the Login button :

```
child: RaisedButton(  
  color: Colors.blue,  
  child: Text("Login",  
    style: TextStyle(fontSize: 20, color: Colors.white),  
  ),  
  onPressed: () async {  
    try {  
      final User = await _auth.signInWithEmailAndPassword(  
          email: email, password: password);  
      if (User != null) {  
        Navigator.pushNamed(context, 'Service');  
  
        _controller.clear();  
      }  
    }
```

```
    } catch (e) {
      print(e);
    }
),
),
```

80- Now, run your app again, and test your Logout icon in the **service.dart** interface. You will find that when you tap this icon, you be moved to the previous interface (**login.dart**) and the password **TextField** is clear.

## Firebase Database

Firebase offers two cloud-based, client-accessible database solutions that support realtime data syncing:

**a- Realtime Database:** is Firebase's original database. It's an efficient, low-latency solution for mobile apps that require synced states across clients in real-time.

**b- Cloud Firestore :** is Firebase's newest database for mobile app development. It builds on the successes of the Realtime Database with a new, more intuitive data model. Cloud Firestore also features richer, faster queries and scales further than the Realtime Database.

## Which database is right for your project?

Google recommends Cloud Firestore for most developers starting a new project. Cloud Firestore offers additional functionality, performance, and scalability on an infrastructure designed to support more powerful features in future releases. Expect to see new query types, more robust security rules and improvements in performance among the advanced features planned for Cloud Firestore.

Both Realtime Database and Cloud Firestore are **NoSQL** Databases.

As you're choosing between database solutions, consider the following differences between Cloud Firestore and the Realtime Database.

Realtime Database	Cloud Firestore
Stores data as one large JSON tree.	Stores data as collections of documents which are very similar to JSON.
Offline support for iOS and Android clients.	Offline support for iOS, Android, and web clients.
It can record client connection status(online or offline) and provide updates every time the client's connection state changes.	Presence is Not supported natively.
Queries can sort <i>or</i> filter a property, but not both.	You can chain filters and combine filtering and sorting on a property in a single query.
Databases are limited to zonal availability in a single region.	Cloud Firestore is a multi-region solution that scales automatically.
Scaling requires sharding.	Scaling is automatic.
Security: <ul style="list-style-type: none"> <li>Reads and writes from mobile SDKs secured by Realtime Database Rules.</li> <li>Reads and writes rules cascade.</li> <li>Can validate data separately using the validate rule.</li> </ul>	Security: <ul style="list-style-type: none"> <li>Reads and writes from mobile SDKs secured by Cloud Firestore Security Rules.</li> <li>Reads and writes from server SDKs secured by Identity and Access Management (IAM).</li> <li>Rules don't cascade unless you use a wildcard.</li> <li>Rules can constrain queries: If a query's results might contain data the user doesn't have access to, the entire query fails.</li> </ul>
Charges only for bandwidth and storage, but at a higher rate.	Charges primarily on operations performed in your database (read, write, delete) and, at a lower rate, bandwidth and storage.

## Real Time Database

Here, you will continue using the same Flutter project which you used to configure the Firebase authentication. This Flutter project is already configured with Firebase web site which you configured in the previous steps. The **pubspec.yaml** file

dependencies uses the Firebase authentication and database as follows:

```
19     dependencies:  
20       flutter:  
21         sdk: flutter  
22         firebase_core: ^0.4.4  
23         firebase_auth: ^0.15.4  
24         firebase_database: ^3.1.1  
25         cloud_firestore: ^0.13.3
```

Also, you downloaded your project Firebase JSON file : **google-service.json**, added it to your project Android files and configured the Gradle files.

In addition, you configured your iOS directories of your Flutter project to have the suitable configurations and files.

If you want to review these previous settings from the beginning, you will need to go back to step 19 of this example.

Now, you will learn how to use the Firebase real time database in a Flutter project. In this example, you will use this Firebase service in the **service.dart** file. To do that, perform the following steps which will take you back to the previous steps:

81- Open **service.dart** file and import the Firebase database package by adding the following code at the top of this file:

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

82- Configure the body of this app to wrapped by Column widget, then wrap it using Padding widget . The body code of **service.dart** will be as follows:

```
body: Padding(  
    padding: const EdgeInsets.all(8.0),  
    child: Column(  
        children: <Widget>[  
            Center(  
                child: Container(  
                    child: Text('Welcome to Android ATC',  
                        style: TextStyle(  
                            fontSize: 20.0,  
                        )),  
            ),  
        ),  
    ),
```

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

To check the new configuration in the interface of **services.dart** file, no need to run your app from the beginning, tap login and enter the user name & password to login or open the **service.dart** file. Instead of that, just login one time and click the hot reload button each time you want to check your new configurations.

83- Now, in this step add a Raised button widget to **service.dart**. You will use this button **onPressed** method to save or send your app data to the Firebase database. The code will be as follows:

```
body: Padding(
    padding: const EdgeInsets.all(8.0),
    child: Column(
        children: <Widget>[
            Center(
                child: Text('Welcome to Android ATC',
                    style: TextStyle(
                        fontSize: 20.0,
                    ),
                ),
            ),
            SizedBox(height: 20.0),
            RaisedButton(child: Text("Save"), onPressed:(){}),
        ],
    ),
),
),
);
});
```

84- Add a Firebase database instance to your project by adding the following code to **service.dart** as illustrated in the grey highlighted color in the following code:

```
class _ServiceState extends State<Service> {
    final FirebaseAuth auth = FirebaseAuth.instance;

    final DatabaseReference databse =
        FirebaseDatabase.instance.reference().child("My_Service");

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

Here, **database** is an instance that represents a group of database values which will be sent from your app to Firebase database, and **My\_Service** represents the database name which will include these database values. The format of this database is JSON (JavaScript Object Notation).

85- Now, you will configure the function which will send or push your app database values to Firebase database using **push()** and **set()** methods. In this example, you may name this function: **sendData** (or any name) as illustrated in the following code:

```
class _ServiceState extends State<Service> {
    final FirebaseAuth auth = FirebaseAuth.instance;

    final DatabaseReference databse =
        FirebaseDatabase.instance.reference().child("My_Service");

    sendData() {
        databse.push().set(
            {'Name': 'William', 'Country': 'USA', 'City': 'Denver'},
        );
    }

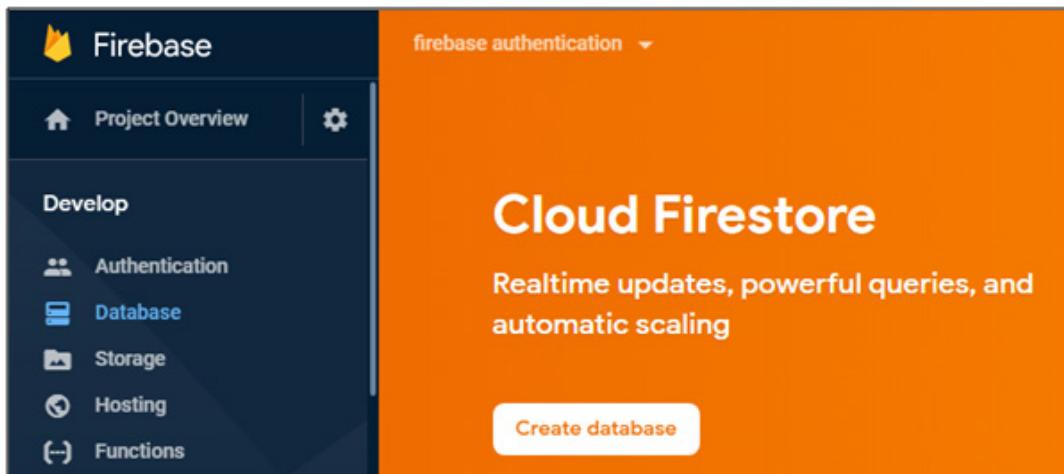
    @override
```

86- Configure your app to call the sendData function when your app user taps the **Save** Raised button by adding the following code to the onPressed method of this button :

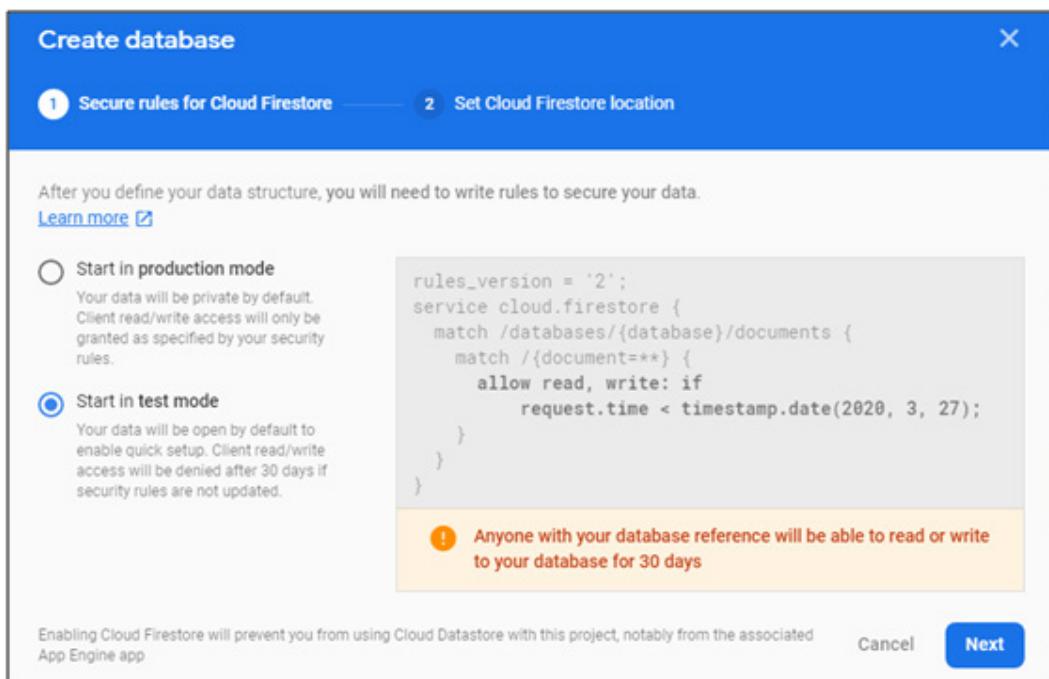
```
RaisedButton(child: Text("Save"), onPressed:() => sendData()),
```

87- Now, you should configure or enable your app Firebase database on Firebase web site.

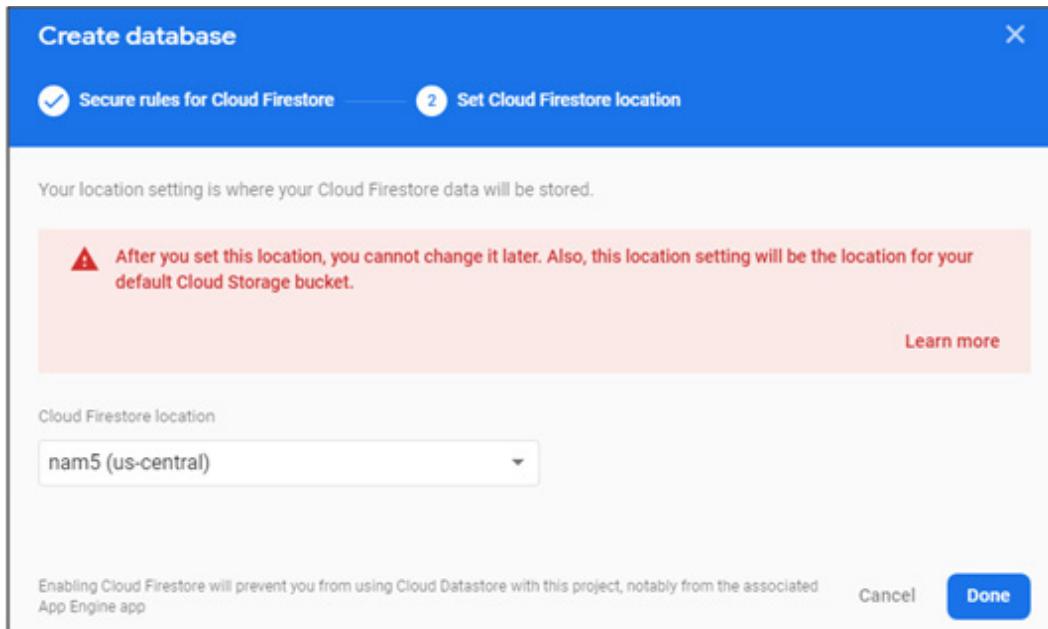
Open your browser , on the left side and as illustrated in the following figure, click **Database** , then click **Create database** button.



88- Select **Start in test mode** and click **Next**.



89- In this step and as illustrated in the following figure, you should select where you will host your database. Before you use *Cloud Firestore*, you must choose a *location* for your database. To reduce latency and increase availability, store your data close to the users and services that need it. For example, depending on my location, I will select US-central. Click **Done**.



99- Now, as you see in the following figure, your database has been created and is ready to receive your app data.

Firebase

Project Overview

Develop

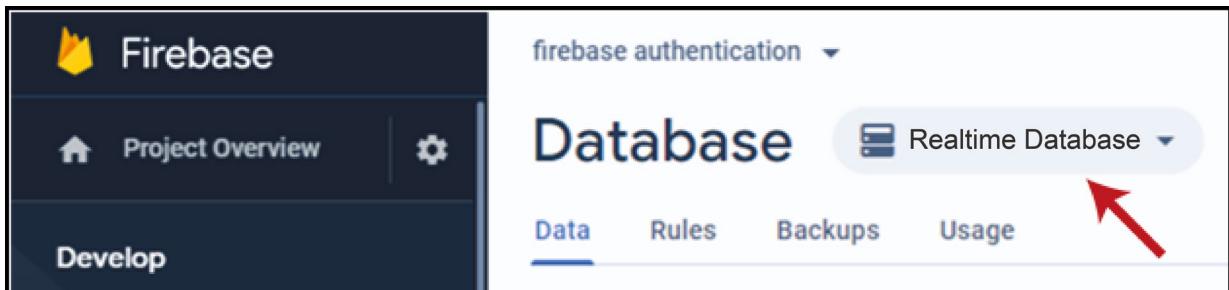
- Authentication
- Database
- Storage
- Hosting
- Functions
- ML Kit

Database

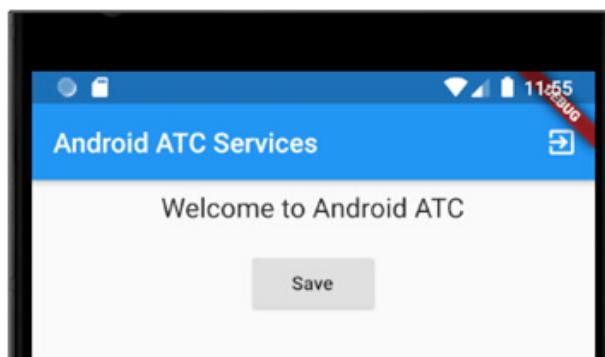
Cloud Firestore

	fir-authentication-b66dc
	Start collection

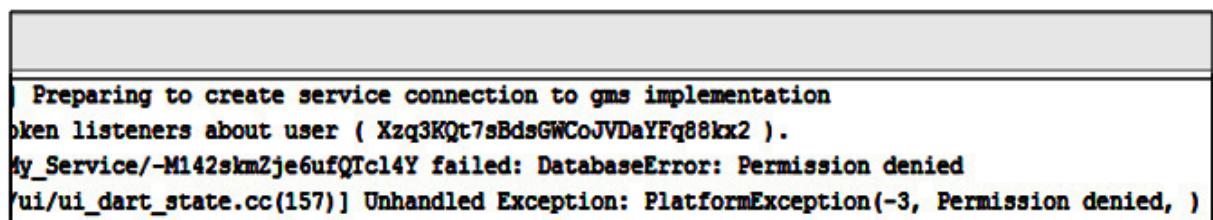
100- Change your database type to Realtime Database as illustrated in the following figure:



101- Now, back to your app, click hot Reload, then click **Save** button on **service.dart** file



Check your Android Studio **Run** console. You will find the following message: **failed DatabaseError: Permission denied** as illustrated in the following figure:



Your app will try to send your data to Firebase real time database, but the default security settings for any new Firebase database will deny read and write.

102- To change the security settings for your Firebase database, open your Firebase web site, click **Database**, click **Realtime Database**, then click the **Rules** tab as illustrated in the following figure:

The screenshot shows the Firebase Realtime Database Rules editor. At the top, there are tabs for Data, Rules, Backups, and Usage. The Rules tab is selected. Below the tabs, there is a code editor with the following content:

```
1 * Visit https://firebase.google.com/docs/database/security to learn more about sec
2 ".read": false,
3 ".write": false
4 }
5 }
6 }
7 }
```

A blue button labeled "Simulator" is located in the top right corner of the code editor area.

As you see in this figure, the default rules disable read and write access to your database by users. With these rules, you can only access the database through the Firebase console.

It is essential that you configure these rules correctly before launching your app to ensure that your users can only access data that they are supposed to.

During development, you can use the public rules in place of the default rules to set your files publicly readable and writable. This can be useful for prototyping, as you can get started without setting up Authentication. This level of access means that anyone can read or write on your database. You should configure more secure rules before launching your app.

103- Replace **false** with **true** for read and write rules as illustrated in the following figure. Then, click **Publish**.

The screenshot shows the "Unpublished changes" dialog for publishing database rules. It has three buttons: "Unpublished changes" (disabled), "Publish" (highlighted in white), and "Discard".

A warning message is displayed: ★ Default security rules are locked from access

The code editor shows the updated security rules:

```
1 * Visit https://firebase.google.com/
2 "rules": {
3     ".read": true,
4     ".write": true
5 }
6 }
7 }
```

104- Now, back to your app and tap the **Save** button once.

105 - Check your Realtime database one Firebase web site. Your database should be created and should look similar to the following figure:

The screenshot shows the Firebase Realtime Database interface. At the top, there's a dropdown menu labeled "firebase authentication". Below it, a large blue header says "Database" and "Realtime Database". Underneath, there are tabs for "Data", "Rules", "Backups", and "Usage", with "Data" being the active tab. A URL "https://fir-authentication-b66dc.firebaseio.com/" is displayed. The main content area shows a single node named "fir-authentication-b66dc", which contains a child node "My\_Service".

If you expand your database, you will get your data as follows:

This screenshot shows the same database structure as above, but the "My\_Service" node is expanded. It reveals three child nodes: "-M1CGb\_pHMkznxkcuVlw", "City: "Denver", and "Country: "USA"". The "-M1CGb\_pHMkznxkcuVlw" node is further expanded, showing "Name: "William".

This format of database is the same JSON database format and if you click the three vertical ellipsis button which is illustrated in the following figure, you will get a submenu including export and import JSON options.

This screenshot shows the top navigation bar of the Firebase Realtime Database console. It includes a URL "https://fir-authentication-b66dc.firebaseio.com/", a refresh icon, and three icons: a plus sign, a minus sign, and a three-dot menu. A red arrow points to the three-dot menu icon.

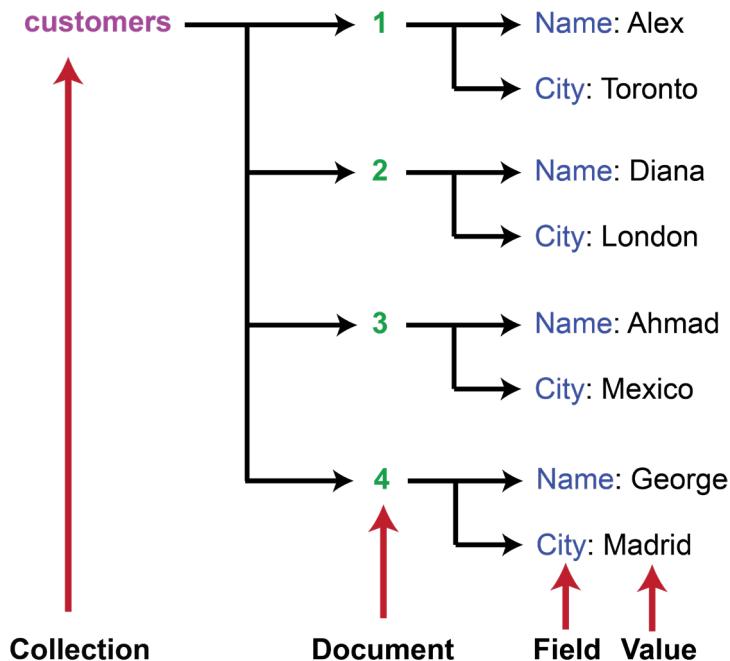
## Cloud Firestore

One of the most common data structures is a database table. A database table consists of columns (fields) and rows (records or values) as illustrated in the table below:

Customer Id	Name	City
1	Alex	Toronto
2	Diana	London
3	Ahmad	Mexico
4	George	Madrid

In the cloud Firestore stores, data structure is very similar to JSON tree as collection of documents.

For example, the previous table will be stored in cloud Firestore as illustrated in the following figure:



The container of data is called **collection** and the database may include one or more collections (tables). The primary key or id for each record (row) is called **document** and each document includes a group of field values that belong to the same document.

For example, in the previous database diagram, Alex is the value of the Name field and Toronto is the value of City field. Also, all of these values belong to the same document

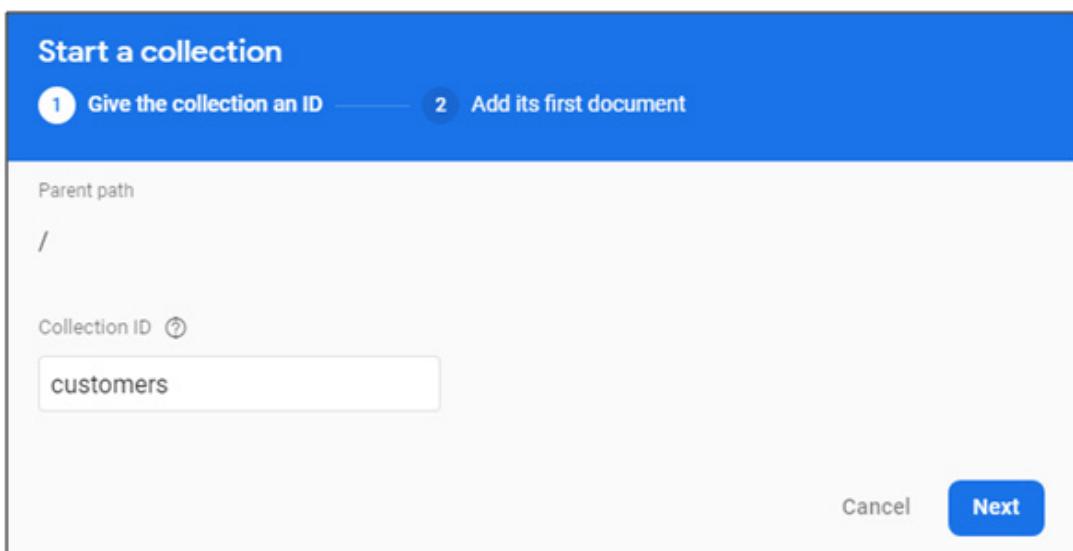
which is : 1

The document value can be auto and a random value or you can enter the values that you want, but each document value must be a unique value in the same collection.

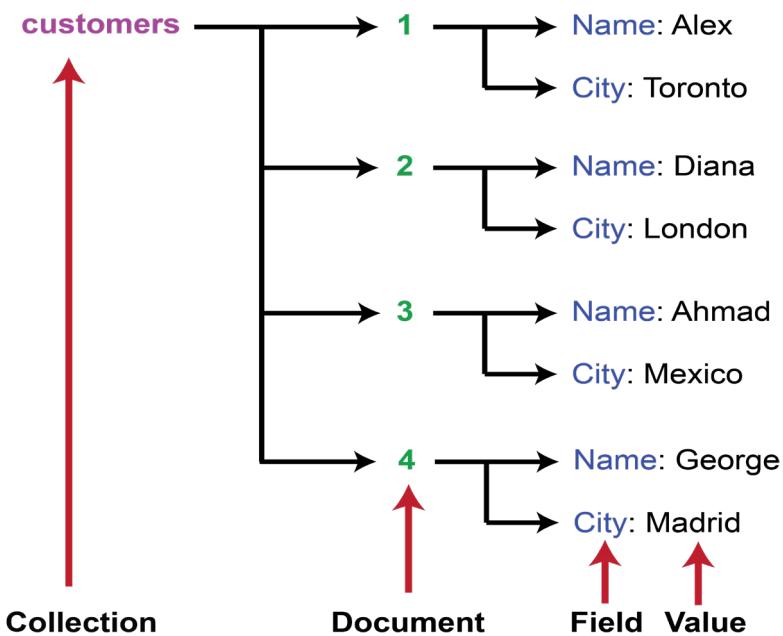
Now, you will continue using the same Flutter project (`firebase_authentication`) as used previously. In the following part, you will create a cloud **Firebase** database using the same as your **Firebase** web account, create a new app interface (`my_firestore.dart` file), and configure this app interface to retrieve or query your cloud firesore data. To do that, perform the following steps:

106- At your Firebase web site, click **Database** on the left panel, then click **Cloud Firestore**.

107- Click **Start collection** to add the first Cloud Firestore, type **customers** for the Collection ID (name), then click **Next**.



108- Now, you will add to your cloud Firestore collection (customers) its **Documents** one by one to get the following data structure:



109 - For the first document, type **Document ID: 1**, **Name** for the first field, **Alex** for the first field value, then click the plus sign to add the second field **City** type **Toronto** for the second field value, then click **Save** as illustrated in the following figure:

Screenshot of the Firebase "Start a collection" dialog:

- Give the collection an ID:** /customers
- Document ID:** 1
- Fields:**
  - Name: string = Alex
  - City: string = Toronto
- Buttons:** Cancel, Save

110 - Repeat the previous step three more times to add the remaining 3 *Documents* to add the following data:

Document ID	1st Filed	2nd Filed
2	Name : Diana	City : London
3	Name : Ahmad	City : Mexico
4	Name : George	City : Madrid

When you finish, you should have the following figure:

Document ID	1st Filed	2nd Filed
1	Name : George City : London	
2	Name : Diana City : Toronto	
3	Name : Ahmad City : Mexico	
4	Name : George City : Madrid	

Now, you should create a new interface to retrieve or query the database which you have created in your Cloud Firestore.

111- Right click **lib** folder, select **New → Dart File**. Type: **my\_firestore.dart** for the file name, then click **OK**.

107- Open the **my\_firestore.dart** file, then add the following Scaffold code, import the Firebase and Cloud Firestor libraries to create a simple Flutter app interface:

```
import 'package:flutter/material.dart';
import 'package:firebase_database.firebaseio_database.dart';
import 'package:cloud_firestore/cloud_firestore.dart';

class MyCloud extends StatefulWidget {
  @override
  _MyCloudState createState() => _MyCloudState();
}

class _MyCloudState extends State<MyCloud> {
  @override
```

```
Widget build(BuildContext context) {  
  return MaterialApp(  
    home: Scaffold(  
      appBar: AppBar(  
        title: Text('My Cloud Firestore'),  
      ),  
      body: Container(),  
    ),  
  );  
}
```

To check how this interface looks like, go to **home.dart** file, and add a button to open this interface.

108- First, add the navigation route name to your interface on **main.dart**.

Open **main.dart** file and add the following code at the top of its code:

```
import 'my_firestore.dart';
```

Then, add the following navigation route to the existing routes:

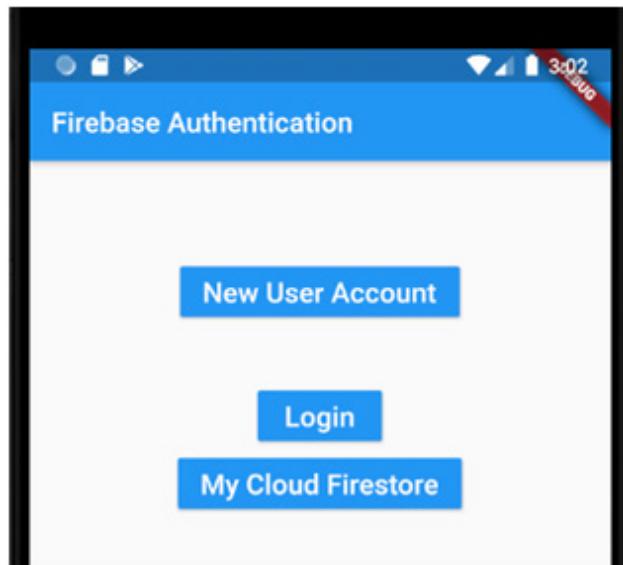
```
'Firestore': (context) => MyCloud(),
```

109- Open **home.dart** file to add the button which will open the file : **my\_firestore.dart** .

Add the following code below the previous buttons:

```
RaisedButton(  
  color: Colors.blue,  
  child: Text('My Cloud Firestore',  
    style: TextStyle(fontSize: 20, color: Colors.white),  
,  
  onPressed: () {  
    Navigator.pushNamed(context, 'Firestore');  
  },  
,
```

110- Run your app. You should get the following figure:



Click : **My Cloud Firestore** button to go to **my\_firestore.dart** interface. If everything works fine, complete the next steps.

Now, to connect your app with your Cloud Firestore you should use **StreamBuilder** widget. This widget or class depends in its work on two classes, **stream** and **builder**.

As illustrated in the following code, the **StreamBuilder** widget will listen to data events flowing from the **stream** class, where the **stream** class provides a way to receive a sequence of data events from your cloud Firestore collection (customers).

```
StreamBuilder (  
  stream:Firestore.instance.collection("customers").snapshots(),  
  builder:(context.snapshot){return ListView();}  
)
```

When you set a listener (**stream** class), Cloud Firestore sends your listener an initial snapshot of the data, and then another snapshot each time the document changes (live update). This will be done using the **snapshot** function.

For every new data event, it will rebuild or update your app data, giving them the latest data event to work with.

The best way to display your database on an app interface is using **ListView** widget.

111- Open **my\_firestore.dart** file. Add the following code get the data from your cloud Firestore and return it in a **ListView** widget. Remember here, your Cloud Firestore collection is: customers.

```
class _MyCloudState extends State<MyCloud> {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(
                    title: Text('My Cloud Firestore'),
                ),
                body: Container(
                    child: StreamBuilder(
                        stream: Firestore.instance.collection('customers').snapshots(),
                        builder: (context, snapshot) {
                            return ListView(
                                children: <Widget>[],
                            );
                        },
                    ),
                ),
            ),
        );
    }
}
```

112- Now, configure the widget function which will build your data in your app interface. Add the following code below **\_MyCloudState** and using **ListTile** widget as illustrated in the following code:

```
class _MyCloudState extends State<MyCloud> {

    List<Widget> dataListWidget(AsyncSnapshot snapshot) {
        return snapshot.data.documents.map<Widget>((document) {
            return ListTile(

```

```

        title: Text(document["Name"]),
        subtitle: Text(document["City"]),
    );
}).toList();
}

```

Important note: here the name of the document fields are case sensitive.

113- Now use `dataListWidget(snapshot)` to display your data in the `ListView` widget as illustrated in the grey highlighted code of the following full code of `my_firestore.dart`:

```

import 'package:flutter/material.dart';
import 'package:firebase_database.firebaseio_database.dart';
import 'package:cloud_firestore/cloud_firestore.dart';

class MyCloud extends StatefulWidget {
    @override
    _MyCloudState createState() => _MyCloudState();
}

class _MyCloudState extends State<MyCloud> {

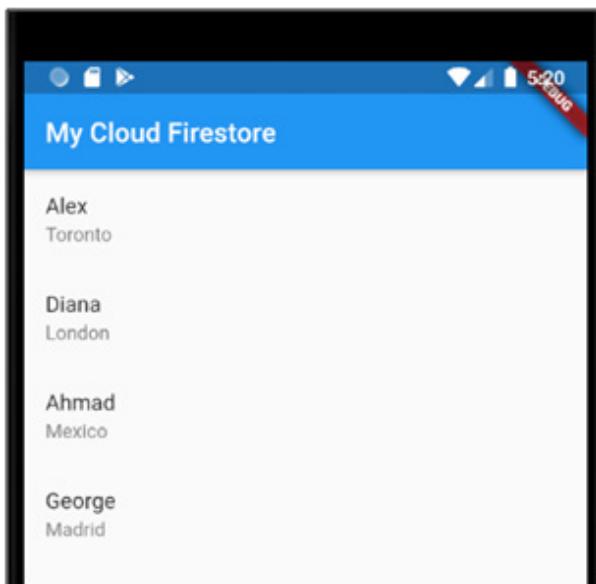
    List<Widget> dataListWidget(AsyncSnapshot snapshot) {
        return snapshot.data.documents.map<Widget>((document) {
            return ListTile(
                title: Text(document["Name"]),
                subtitle: Text(document["City"]),
            );
        }).toList();
    }

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(
                    title: Text('My Cloud Firestore'),
                ),
                body: Container(

```

```
        child: StreamBuilder(  
            stream: Firestore.instance.collection('customers').  
snapshots(),  
            builder: (context, snapshot) {  
                return ListView(  
                    children: dataListWidget(snapshot),  
                );  
            },  
        ),  
    ),  
),  
);  
}  
}
```

114- Click the hot reload button. You should get the following figure:



115- Back to your Cloud Firestore on Firebase web site, add a new document including the following fields values :

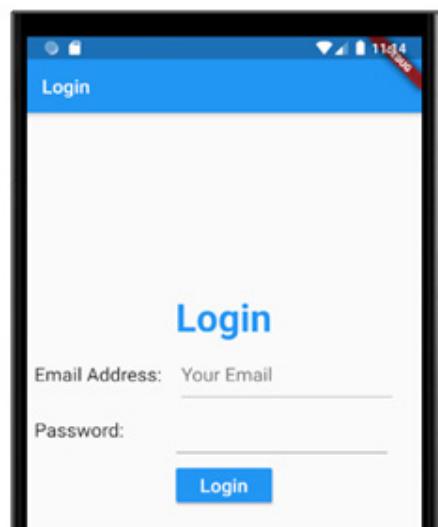
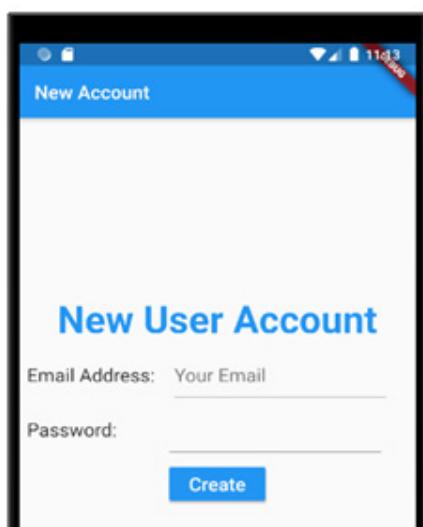
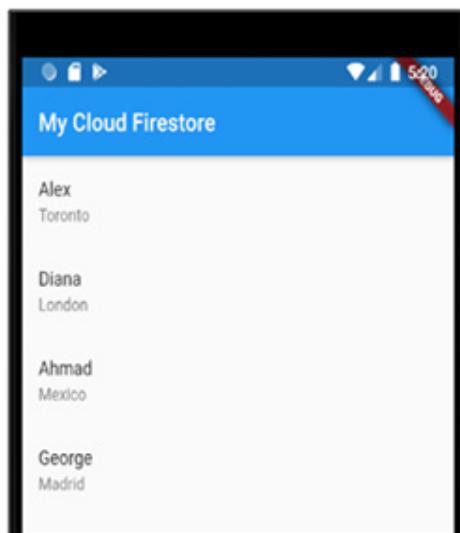
Name: **Angela**, and City: **Hanoi**.

Just check your phone emulator, no need to click **Run** or **Hot Reload**. You will find this new data updated due to using snapshots function in **StreamBuilder** widget in your code.

## Lab 9 : Create a User Profile Interface using Firebase

In this lab, you will create a Flutter app that allows the app user to create their accounts (user name & password) to access the app service. You will create an authentication procedure depending on Firebase authentication service.

You will create the startup interface which includes the “**New User Account**” and “**Login**” buttons. When the user taps the “**New User Account**” button he/she will move to the New Account interface which will be used to create the new app user account. Also, if the user taps the **Login** button, he/she can login to the app using the account which he/she has created in the **New Account** interface. To do this, you should configure your Flutter app to use Firebase authentication service.

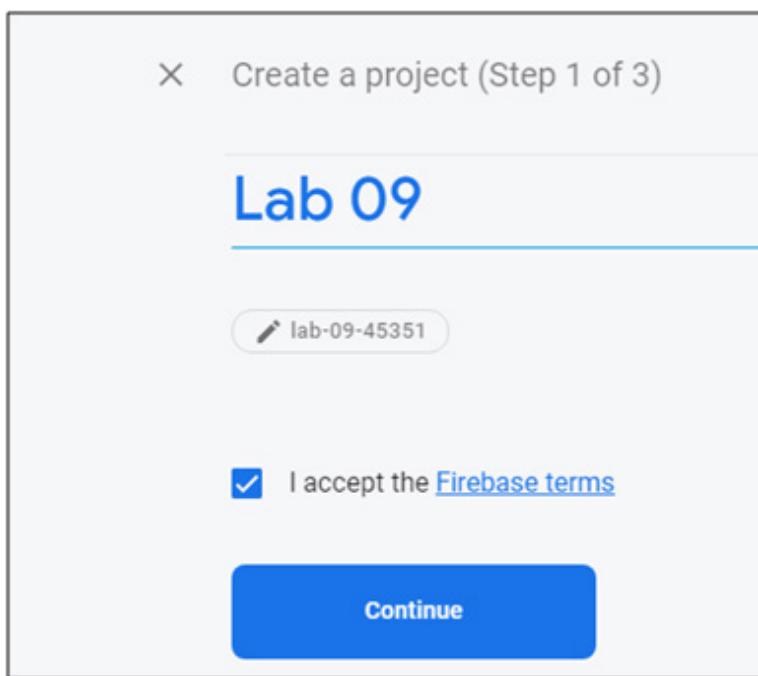


To achieve the previous lab scenario, perform the following steps:

- 1- Open **Android Studio**
- 2- Click **File → New → New Flutter Project**
- 3- Select **Flutter Application**, then click **Next**.
- 4- Type : **lab\_09** for Project Name, and create a new folder : **Lab\_09** for Project Location. Click **Next**.
- 5- Type : **androidatc.com** for Company domain, then click **Finish**

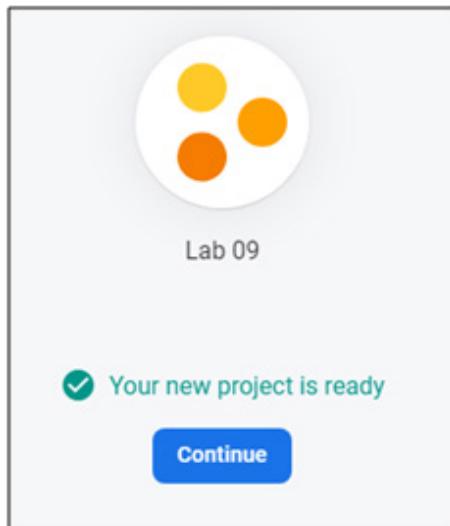
#### Configure Your Flutter App to use Firebase Services:

- 6- Go to : <https://console.firebaseio.google.com>
- 7- Sign into Firebase using your Google account.
- 8- Click **Get Started**, then click **Create a project**. Fill out your project name “**Lab 09**” or any other name as illustrated in the following figure. Check **I accept the Firebase terms**, then click **Continue**.



- 9- Click **Enable Google Analytics for this project** to **disable** this feature because this app is just for testing. Google features are important to get a report about your live apps in the future.

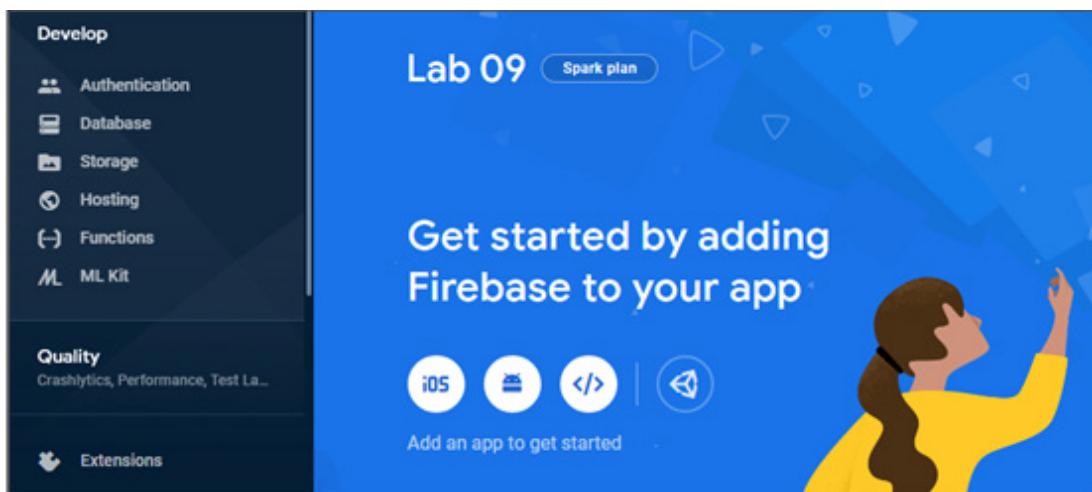
Click **Create Project**. Then within seconds, you should get the following message as illustrated in the following figure. It displays your project that has been created on Google Firebase as illustrated in the following figure. Click **Continue**.



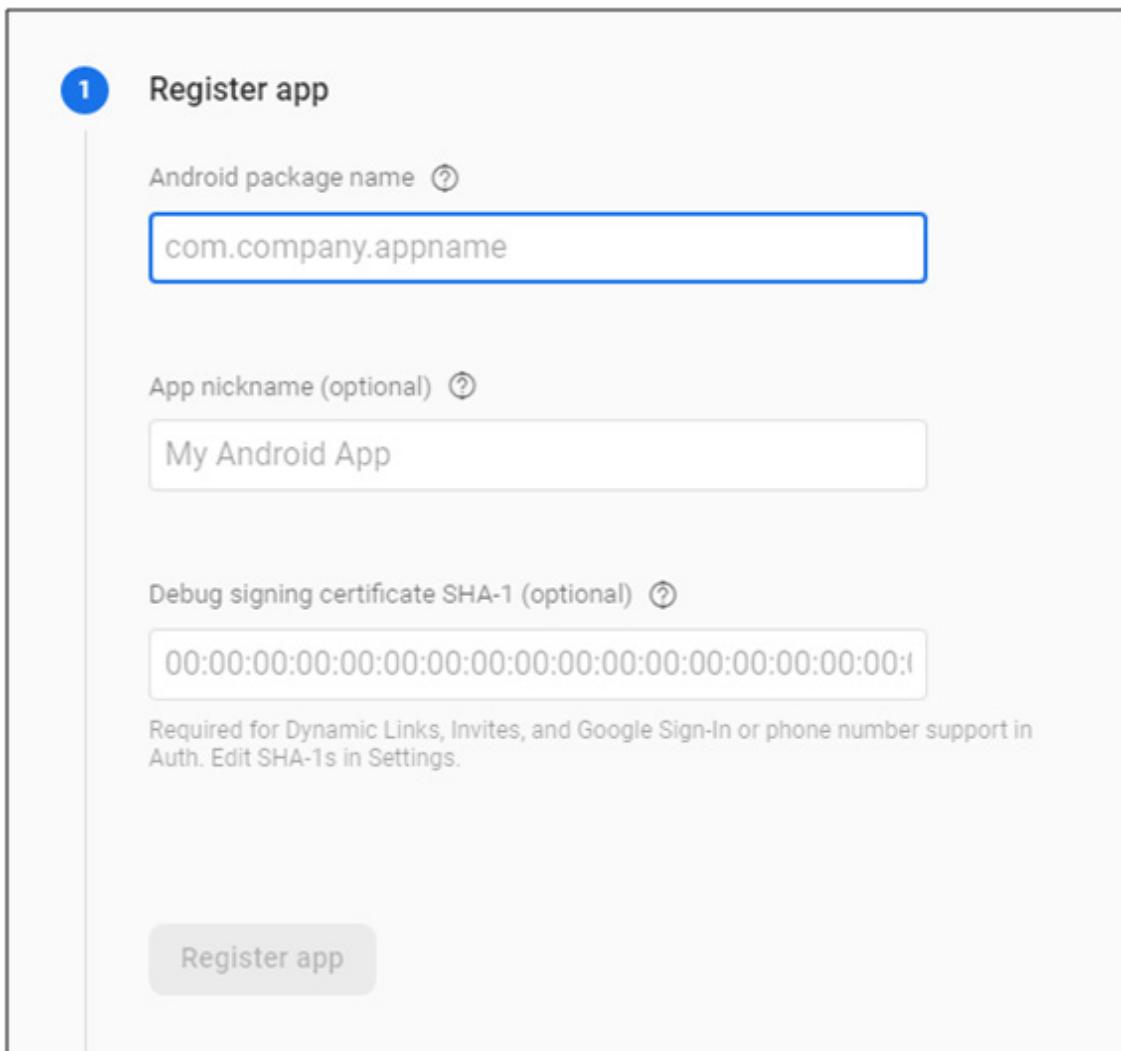
Now, your project has been created on Firebase web site and is ready to configure.

Lesson 9 includes how to add Firebase configuration to iOS and Android files in the Flutter app step by step; however, in this lab you will configure only the Android part only, and if you need more info about iOS, please review the content of the lesson.

10- To add Firebase configurations to your Android files, click on the **Android icon** in the following figure:



11- You should get the following dialog box to register your app. As you will see, you should enter your project name which must be a unique name on Google Play store.



To get your Android project name, go to **Android Studio**, then open the following path:

Project name (**lab\_09\_android**) → **android** → **app** → **build.gradle**

Then, scroll down the file content and as illustrated in the following figure, the application Id is : **com.androidatc.lab\_09**

```

defaultConfig {
    // TODO: Specify your own unique Application ID (https://developer.android.com/studio/publish/app-indexing.html)
    applicationId "com.androidatc.lab_09"
    minSdkVersion 16
    targetSdkVersion 28
    versionCode flutterVersionCode.toInt()
    versionName flutterVersionName
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}

```

**Copy** this Id and **paste** it on your [Firebase web site](#) for the **Android package name**. Then click the **Register app** button.

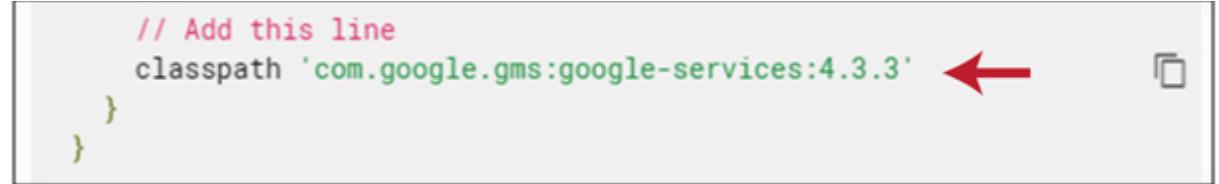
12- Click **Download google-service.json** button to download the JSON configuration file. Open your download folder and should find this file name : **google-services.json** without any number.

13- Move this file : **google-services.json** from your download folder to the **app** folder using the drag and drop technique. This **app** folder is in **Android Studio** in the following path:

**Project name (lab\_09) → android(lab\_09\_android) → app**, then click **OK**.

14- Now, return to the Firebase web site to complete the setup steps. Click **Next**.

15- In the Add Firebase SDK step, copy the line illustrated in the following figure or click the copy icon.



```

// Add this line
classpath 'com.google.gms:google-services:4.3.3' ←
}

```

16- Go to Android Studio, open the **build.gradle** file which is in the following path :

**Your Project name → android(lab\_09\_android) → build.gradle**

Paste this class path within the dependencies braces as illustrated in the following figure:

```
dependencies {  
    classpath 'com.android.tools.build:gradle:3.5.0'  
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"  
    classpath 'com.google.gms:google-services:4.3.3'  
}  
}
```

17- Now, return to the Firebase web site and from the configuration wizard, copy the other two lines :

```
apply plugin: 'com.android.application'  
apply plugin: 'com.google.gms.google-services'
```

go to Android Studio, open the **build.gradle** file which is in the following path:

Your Project name → **android (lab\_09\_android)**→ **app** → **build.gradle**

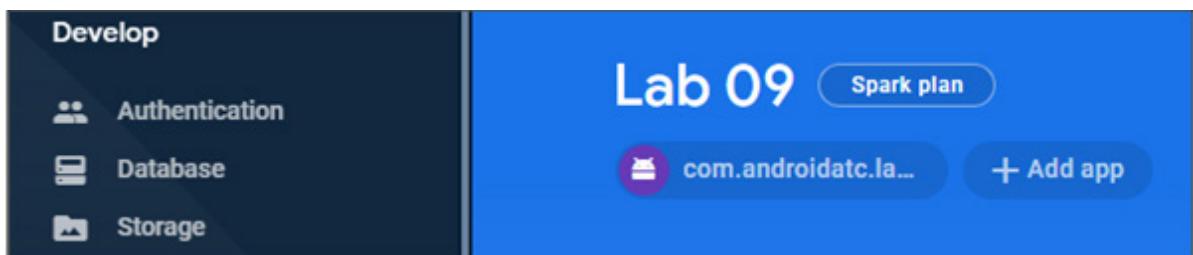
Then, paste the other two lines as separate lines at the end of this file as illustrated in the following figure:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test:runner:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'  
}  
apply plugin: 'com.android.application'  
apply plugin: 'com.google.gms.google-services'
```

Also, in the same **build.gradle** file add :**multiDexEnabled true** as illustrated in the grey highlighted part of the following configuration:

```
defaultConfig {  
  
    applicationId "com.androidatc.lab_09"  
    minSdkVersion 16  
    targetSdkVersion 28  
    versionCode flutterVersionCode.toInt()  
    versionName flutterVersionName  
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
    multiDexEnabled true  
}
```

18- Return to the Firebase web site, click **Next**. Then click the **Continue to console** button. You should get the following figure:



19- Now, close all your Gradle files. Be sure that your Android emulator is selected. Stop your app, then run it again to be sure that all changes in your Android Studio Gradle files were applied.

20- The Firebase settings and configurations are always updated. Therefore, to be up to date, you should follow the last setting in the following web link: <https://github.com/FirebaseExtended/flutterfire>

21- Scroll down until you get the following figure:

## Available FlutterFire plugins

Plugin	Version	Firebase feature	Source code
cloud_firestore	pub v0.13.4	Cloud Firestore	cloud_firestore
cloud_functions	pub v0.4.2+2	Cloud Functions	cloud_functions
firebase_admob	pub v0.9.1+3	Firebase AdMob	firebase_admob
firebase_analytics	pub v5.0.11	Firebase Analytics	firebase_analytics
firebase_auth	pub v0.15.5+2	Firebase Authentication	firebase_auth
firebase_core	pub v0.4.4+2	Firebase Core	firebase_core
firebase_crashlytics	pub v0.1.3	Firebase Crashlytics	firebase_crashlytics
firebase_database	pub v3.1.3	Firebase Realtime Database	firebase_database

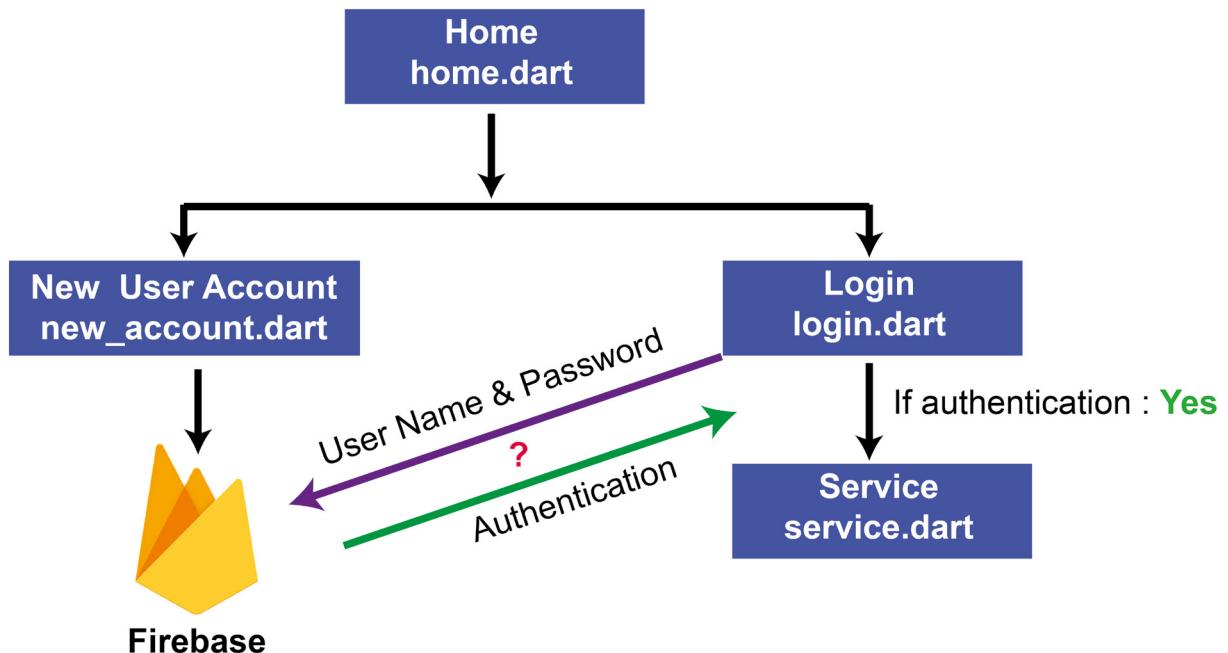
22- Now, you should configure your app settings or add Firebase plug-in services to your app by configuring: **pubspec.yaml** file for Firebase authentication and database.

Click the **cloud\_firestore** plug-in , click the **Installing** tab and copy the dependencies value : `cloud_firestore: ^0.13.4` or the latest update value which you will find at the time you perform this lab. The following figure displays the current `cloud_firestore` configuration:

The screenshot shows the official GitHub repository for the `cloud_firestore` package. The top navigation bar includes links for Readme, Changelog, Example, **Installing**, and Versions. A blue circular badge indicates 100 stars. Below the tabs, the heading "Use this package as a library" is displayed. Under the "1. Depend on it" section, instructions advise adding the dependency line `cloud_firestore: ^0.13.4` to the `pubspec.yaml` file. The dependency line is shown in red, indicating it is a required dependency.

```
dependencies:
  cloud_firestore: ^0.13.4
```

- 23- Open your **pubspec.yaml** file in your Android Studio and paste this value under dependencies.
- 24- Click **Back** on your web browser toolbar to get the “**Available FlutterFire plugins**” web page again or go to the web link: <https://github.com/FirebaseExtended/flutterfire>, then click the : **firebase\_auth** , click the **Installing** tab , then copy the existing dependencies value: **firebase\_auth: ^0.15.4**
- 25- Paste this value in your **pubspec.yaml** file under the dependencies.
- 26- Back on your web browser to the “**Available FlutterFire plugins**” web page or the web link: <https://github.com/FirebaseExtended/flutterfire>, then click the : **firebase\_database**
- 27- Click the **Installing** tab, then copy the existing dependencies value:
- firebase\_database: ^3.1.3**
- 28- Paste this value in your **pubspec.yaml** file under the dependencies. The dependencies of the **pubspec.yaml** file should be as illustrated in the following figure:
- ```
19     dependencies:
20       flutter:
21         sdk: flutter
22       firebase_core: ^0.4.4
23       firebase_auth: ^0.15.4
24       firebase_database: ^3.1.1
25       cloud_firestore: ^0.13.3
```
- 29- Now, at the top of the **pubspec.yaml** file content, click **Packages Get** to incorporate all of those settings into your Flutter project.
- Now, you should create the Flutter app structure as illustrated in the following figure, where the startup interface will be **home.dart** which includes two buttons: The first button (**New User Account**), will forward the app user to (**new\_account.dart**) to create a new user account at Firebase web site. The second button (Login) will forward the app user (**login.dart**) to login using the user name (email address) and password which the app user has created in the “**creating new user**” step. If the user can login successfully, he/she will login to the user profile (**user\_profile.dart**) interface to complete his/her profile by adding an image to his/her profile, and when the app user clicks Save, all new information along with the image will upload to the cloud Firestore database.



20- You should create three new interfaces (Dart files). Right click the **lib** folder and select **New → Dart File** . Type **home** for the file name , and then press **Enter** (or Return for Mac).

21- Repeat the previous step to create other three dart files: **new\_account.dart**, **login.dart**, and **user\_profile.dart**

22- Open the **new\_account.dart** file and create the interface which is illustrated in the figure below. This interface contains two text fields to ask the app user to enter his/her username and password. Then click the **Create** button.

**New User Account**

Email Address:

Password:

**Create**

The code of **new\_account.dart** will create this interface as follows:

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

class NewAccount extends StatefulWidget {
  @override
  _NewAccountState createState() => _NewAccountState();
}

class _NewAccountState extends State<NewAccount> {
  String email;
  String password;
  final FirebaseAuth _auth = FirebaseAuth.instance;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('New Account '),
        ),
        body: Padding(
          padding: const EdgeInsets.all(8.0),
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              ListView(
                shrinkWrap: true,
                children: <Widget>[
                  Container(
                    alignment: Alignment.center,
                    child: Text(
                      'New User Account',
                      style: TextStyle(fontSize: 40,
                      color: Colors.blue,
                      fontWeight: FontWeight.bold,)),
                ],
              ),
              SizedBox(height: 10.0,),
            ],
          ),
        ),
      ),
    );
  }
}
```

```
// ***** User Name *****
Row(
  children: [
    Text('Email Address:',
        style: TextStyle(fontSize: 20.0),),
    SizedBox(width: 20.0),

    SizedBox(width: 220.0,

      child: TextField(
        onChanged: (value1) {
          email = value1;},

        style: TextStyle(fontSize: 20, color: Colors.blue),
        keyboardType: TextInputType.emailAddress,
        autocorrect: false,
        cursorColor: Colors.red,
        decoration: InputDecoration(hintText: 'Your Email',),
      ),
    ],
  ),
),
SizedBox(height: 10.0,),

// ***** Password *****
Row(
  children: [
    Text('Password:         ',
        style: TextStyle(fontSize: 20.0),),
    SizedBox(width: 20.0,),

    SizedBox(width: 220.0,

      child: TextField(
        onChanged: (value2) {
          password = value2;},

        style: TextStyle(fontSize: 20, color: Colors.blue),
        textInputAction: TextInputAction.done,
```

```
        autocorrect: false,
    ),
),
],
),
),

SizedBox(height: 10.0,),

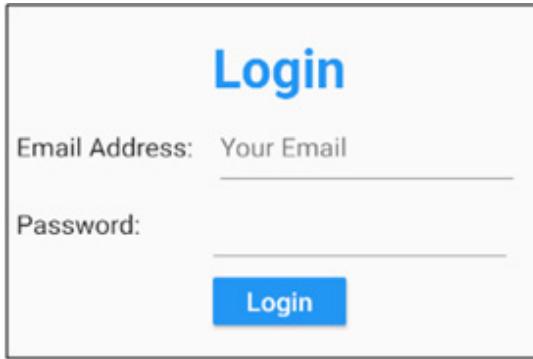
// ***** Create Button *****
Center(
    child: Container(
        width: 100,
        child: RaisedButton(
            color: Colors.blue,
            child: Text("Create",
style: TextStyle(fontSize: 20, color: Colors.white),),

onPressed: () async {
    try {
        final newUser =await _auth.createUserWithEmailAndPassword(
            email: email, password: password);

        if (newUser != null) {
            Navigator.pushNamed(context, 'Home');}

        catch (e) {
            print(e);}
        },
        ),
        ),
        ],
        ),
        ],
        ),
        ),
        );
    }
}
```

23- Open **login.dart** file and add the code which will create a login interface. The app user will use this interface to login using the user name and password which he/she has created in the **create interface** step. The login interface should be as illustrated in the following figure:



The full code of the **login.dart** follows:

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

class Login extends StatefulWidget {
  @override
  _Login createState() => _Login();
}

class _Login extends State<Login> {
  String email;
  String password;
  final FirebaseAuth _auth = FirebaseAuth.instance;

  TextEditingController _controller = new TextEditingController();
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Login'),
        ),
        body: Padding(
```

```
padding: const EdgeInsets.all(8.0),
child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
        ListView(
            shrinkWrap: true,
            children: <Widget>[
                Container(
                    alignment: Alignment.center,
                    child: Text(
                        'Login',
                        style: TextStyle(
                            fontSize: 40,
                            color: Colors.blue,
                            fontWeight: FontWeight.bold,
                        ),
                    ),
                ),
            ],
        ),
        SizedBox(height: 10.0,),

// ***** User Name *****
Row(
    children: [
        Text('Email Address:',
            style: TextStyle(fontSize: 20.0),),
        SizedBox(width: 20.0,),

        SizedBox(width: 220.0,
            child: TextField(
                onChanged: (value1) {
                    email = value1;},

                style: TextStyle(fontSize: 20, color: Colors.blue),
                keyboardType: TextInputType.emailAddress,
               textInputAction: TextInputAction.done,
                autocorrect: false,
                cursorColor: Colors.red,
                decoration: InputDecoration(hintText: 'Your Email',),
            ),
        ),
    ],
)
```

```
        ),
    ],
),

SizedBox(height: 10.0,),

// ***** Password *****
Row(
    children: [
        Text('Password:      ',
            style: TextStyle(fontSize: 20.0),),
        SizedBox(width: 20.0,),

        SizedBox(width: 220.0,
            child: TextField(
                controller: _controller,
                onChanged: (value2) {
                    password = value2;},
                style: TextStyle(fontSize: 20, color: Colors.blue),
               textInputAction: TextInputAction.done,
                autocorrect: false,
            ),
        ),
    ],
),
),

SizedBox(height: 10.0,),

// ***** Login Button *****
Center(
    child: Container(
        width: 100,
        child: RaisedButton(
            color: Colors.blue,
            child: Text("Login",
style: TextStyle(fontSize: 20, color: Colors.white),),

            onPressed: () async {
                try {
```

```

        final User = await _auth.signInWithEmailAndPassword(
            email: email, password: password);
        if (User != null) {
            Navigator.pushNamed(context, 'Service');
            _controller.clear();
        }
    catch (e) {
        print(e);
    }
}

```

Now, you should design the interface which includes the services that the user logged in for. Remember, the purpose of this lab is to practice configuring login and logout of an app interface. Here, we will not focus on the services the user will find after login because there are many services that can be added here, such as creating a user profile, shopping, payment, subscription and others.

24- To create this app interface, open the **user\_profile.dart** file and type the following code:

```

import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:firebase_database.firebaseio_database.dart';

class Profile extends StatefulWidget {
    @override
    _ProfileState createState() => _ProfileState();
}

```

```
class _ProfileState extends State<Profile> {
    final FirebaseAuth auth = FirebaseAuth.instance;

    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            home: Scaffold(
                appBar: AppBar(title: Text('User Profile'),
                actions: <Widget>[
                    // action button
                    IconButton(
                        icon: Icon(Icons.exit_to_app),
                        onPressed: () {
                            auth.signOut();
                            Navigator.pop(context);},),],),
                body: Padding(
                    padding: const EdgeInsets.all(8.0),
                    child: Column(
                        children: <Widget>[
                            Center(
                                child: Text('Welcome to Android ATC',
                                    style: TextStyle(
  fontSize: 20.0,)),),
                            SizedBox(height: 20.0),
                            ],
                    ),
                ),
            );
    }
}
```

25- Open **home.dart** file and add the following code which creates the startup interface, which has the navigation buttons to all app interfaces:

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

class Home extends StatefulWidget {
  @override
  _HomeState createState() => _HomeState();
}

class _HomeState extends State<Home> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Firebase Authentication'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              RaisedButton(
                color: Colors.blue,
                child: Text('New User Account',
                  style: TextStyle(fontSize: 20, color: Colors.white),
                ),
                onPressed: () {
                  Navigator.pushNamed(context, 'NewAccount');
                },
              ),
              SizedBox(height: 40.0,),

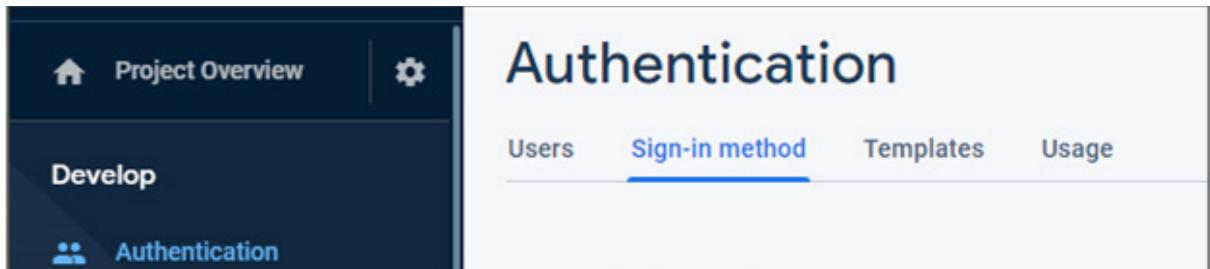
              RaisedButton(
                color: Colors.blue,
                child: Text('Login',
                  style: TextStyle(fontSize: 20, color: Colors.white),
                ),
                onPressed: () {
                  Navigator.pushNamed(context, 'Login');},
              ),
            ],
        ),
      ),
    );
  }
}
```

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

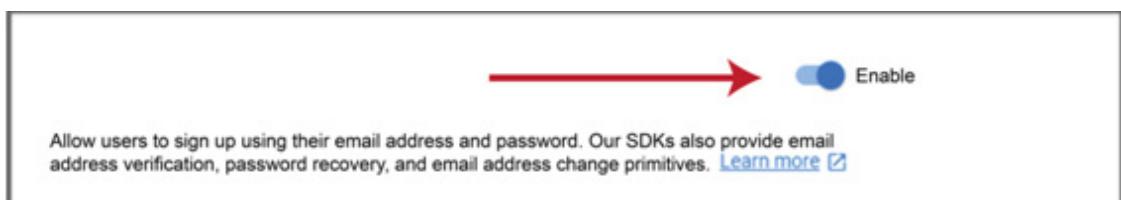
25- Open **main.dart** file, **delete** all the code and add the following code which makes the **home.dart** file interface the start up interface when you run this app. This file includes all the route navigation:

```
import 'login.dart';  
import 'new_account.dart';  
import 'package:flutter/cupertino.dart';  
import 'package:flutter/material.dart';  
import 'home.dart';  
import 'user_profile.dart';  
  
void main() => runApp(MyApp());  
  
class MyApp extends StatefulWidget {  
    @override  
    _MyAppState createState() => _MyAppState();  
}  
  
class _MyAppState extends State<MyApp> {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Home(),  
  
            routes: {  
                'Login': (context) => Login(),  
                'NewAccount': (context) => NewAccount(),  
                'Profile': (context) => Profile(),  
                'Home': (context) => Home(),  
            },  
        );  
    }  
}
```

26- Before creating any user account on Firebase, you should open your Firebase web site again and on the left side of your web browser under the Develop console, click **Authentication**. Then click the **Sign-in method** tab as illustrated in the following figure:



27 - Click **Email/Password**, click the **Enable** switch button, then click **Save** as illustrated in the following figure:



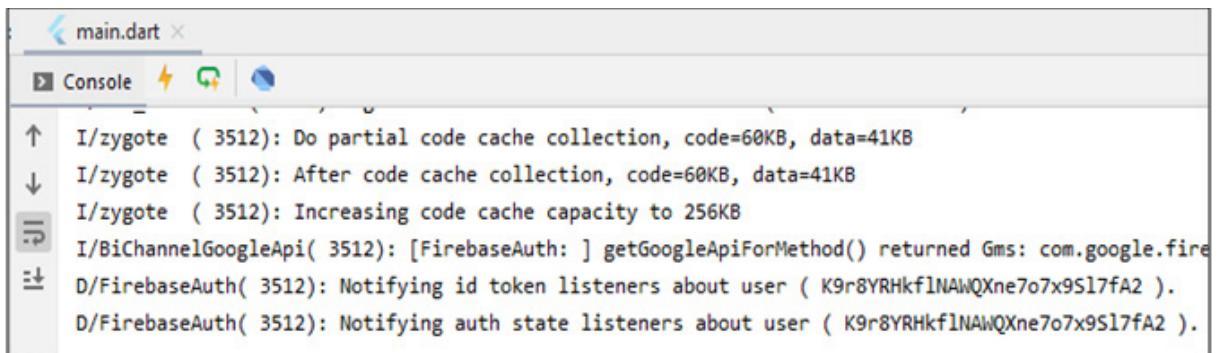
28- Stop your emulator, click **Run** your app, then click : **New User Account** button to move to the **new\_account.dart** interface.

**Important Note:** In the login or create the user account interface, make sure not to add a space before or after your email address. Also, don't press the **Tab** key to move from the email address text field to the password login since there will be an extra space (extra character) added to your email address. This extra character will result in an error message in your Android Studio run console such as : (ERROR\_INVALID\_EMAIL, the email address is badly formatted., null).

Type your email address and any password (at least 6 characters), then tap the **Create** button as illustrated in the following figure:

A screenshot of the 'New User Account' form. It has two text input fields: 'Email Address' containing 'test@androidatc.com' and 'Password' containing '123456'. Below the password field is a large blue 'Create' button.

You should get the following result in the Run console:

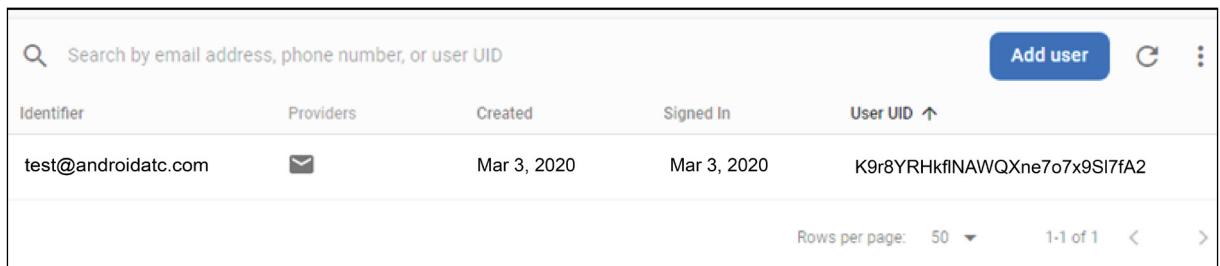


The screenshot shows the Android Studio Run console with the title "main.dart". The "Console" tab is selected. The logcat output displays several messages:

```
I/zygote ( 3512): Do partial code cache collection, code=60KB, data=41KB
I/zygote ( 3512): After code cache collection, code=60KB, data=41KB
I/zygote ( 3512): Increasing code cache capacity to 256KB
I/BiChannelGoogleApi( 3512): [FirebaseAuth: ] getGoogleApiForMethod() returned Gms: com.google.firebaseio
D/FirebaseAuth( 3512): Notifying id token listeners about user ( K9r8YRHkf1NAWQXne7o7x9Sl7fA2 ).
D/FirebaseAuth( 3512): Notifying auth state listeners about user ( K9r8YRHkf1NAWQXne7o7x9Sl7fA2 ).
```

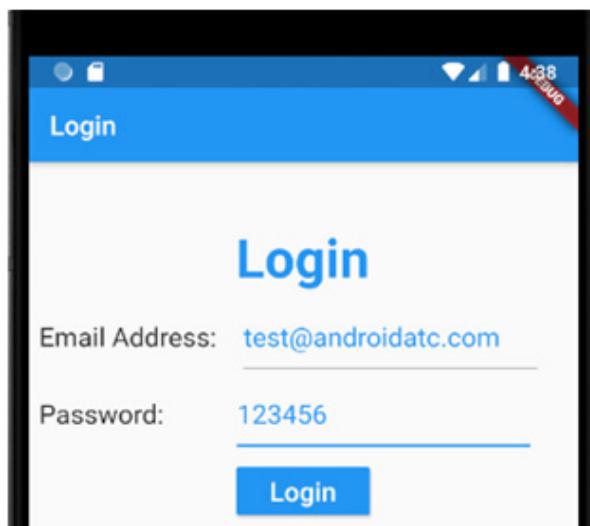
29- Back to your Firebase web site , and in the **Authentication** part, click the **Users** tab where you should find that there is a new user account that has been created as illustrated in the following figure:

Click **Refresh** if you did not find a new user account.

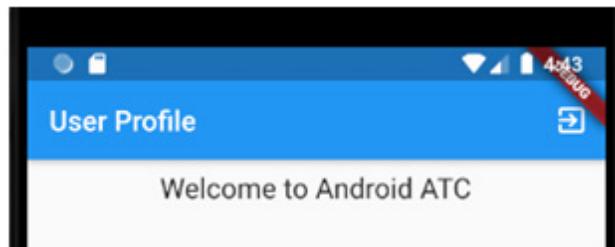


| Identifier          | Providers | Created     | Signed In   | User UID                     |
|---------------------|-----------|-------------|-------------|------------------------------|
| test@androidatc.com | ✉         | Mar 3, 2020 | Mar 3, 2020 | K9r8YRHkf1NAWQXne7o7x9Sl7fA2 |

30- Tap the **Login** button. Enter the user name (email) and the password for the account which you have created in the previous step (without pressing Tab key when you move from the email address Text Field to the Password Text Field), then tap the **Login** button as illustrated in the following figure:



You will get to the content of the `user_profile.dart` file as illustrated in the following figure:



31- In the right corner of the title bar of the `user_profile.dart` interface, tap the exit icon to logout of this interface. Then you will go back to the Login interface.

This is the end of the lab. However, please note the following:

- I- The explanation of all the previous steps is available in the example of this lesson.
- II- Try to complete this lab by adding an extra part to create a user profile interface. For example, try to add an image to the Firebase storage or add the user information such as full name and address to the cloud Firestore.
- III- You may use the lab source files to import these lab files to your Android Studio IDE, and test how this lab works, but you should configure your Firebase web account to be compatible with this lab code. Also, add your Firebase `google-services.json` file to your Android Studio app folder instead of the existing file.

# Lesson 10: Location-Aware Apps: Using GPS and Google Maps

|                                                                   |       |
|-------------------------------------------------------------------|-------|
| <b>Introduction .....</b>                                         | 10-2  |
| <b>What is GPS and how does it work? .....</b>                    | 10-2  |
| <b>The Camera Position.....</b>                                   | 10-4  |
| <b>Adding Google Maps to a Flutter app .....</b>                  | 10-5  |
| Getting a Google API key .....                                    | 10-6  |
| Adding Google Maps Flutter plug-in as a dependency .....          | 10-11 |
| Adding your API key for your Android app .....                    | 10-12 |
| Adding your API key for your iOS app .....                        | 10-13 |
| Adding a Google Map on Your Flutter App Screen.....               | 10-14 |
| Adding a Google Map Marker .....                                  | 10-18 |
| Google Map Types.....                                             | 10-21 |
| Moving the Camera (Camera Animation) .....                        | 10-23 |
| Capturing an App User's Location for iOS and Android Apps.....    | 10-26 |
| <b>Lab10: Location-Aware Apps Using GPS and Google Maps .....</b> | 10-27 |
| Getting a Google API key .....                                    | 10-28 |
| Creating an App Interface.....                                    | 10-33 |
| Configuring your App to Use Your API Key .....                    | 10-34 |
| Adding a Google Map on your Flutter App Screen .....              | 10-36 |
| Adding a Google Map Marker .....                                  | 10-38 |
| Capturing Users' Location .....                                   | 10-39 |
| Configuring User App's Permission.....                            | 10-41 |

## Introduction

Most of the mobile applications nowadays rely on users' Geo-location and web mapping services. GPS is considered one of the most accurate Geo-location providers. In order to increase users' experience of location awareness, geo-coordinates should be represented graphically and this can be achieved using web-mapping services such as Google Maps.

This lesson describes the different techniques used to capture geo-coordinates especially the GPS. It also covers some important topics related to Google Maps such as adding maps to your Flutter application, drawing shapes over maps, adding markers, capturing app users' location, etc...

## What is GPS and how does it work?

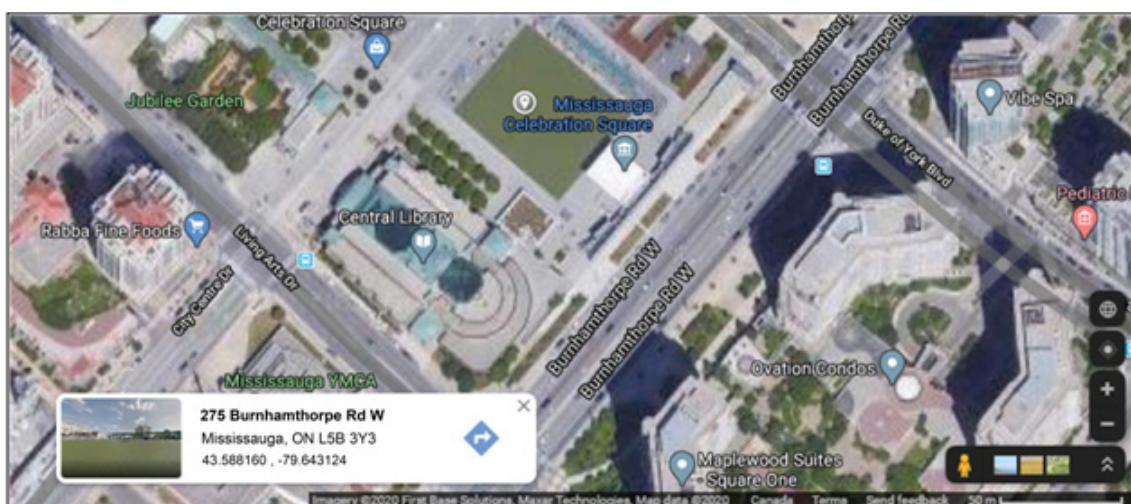
GPS or Global Positioning System is a navigation system based on satellites. At over 20,000 kilometers above sea level (20,180 Kilometers) there is a constellation of satellites, each orbiting the Earth every 11 hours and 58 minutes. These satellites are continually sending data down to the Earth, which in turn is being received by GPS receivers such as mobile phones, allowing the device to calculate its position on earth.

For example, open Google maps: [google.com/maps](http://google.com/maps) web site, then check your home address on Google maps.

The following figure displays the Latitude and Longitude values as follows:

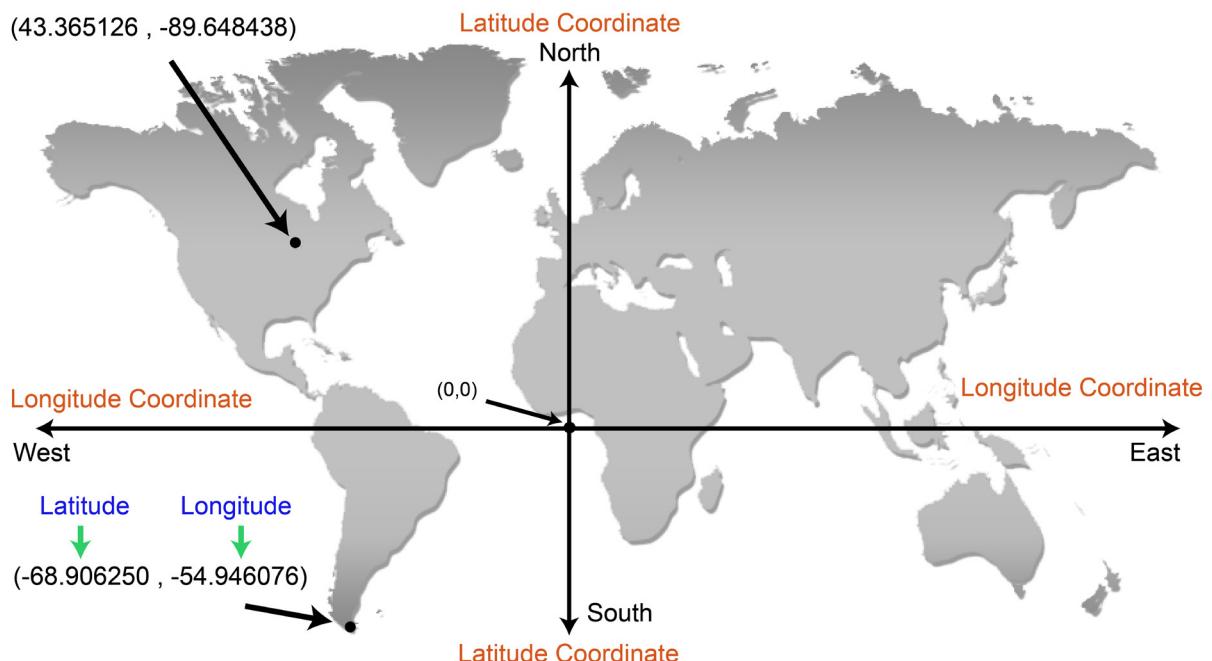
**Latitude:** 43.588160

**Longitude:** -79.643124



The GPS Coordinates depend on the **latitude** and **longitude** values. Latitude and Longitude are the units that represent the coordinates in a geographic coordinate system. Just like every actual house has an address (which includes the number, the name of the street, the city, etc ), every single point on the surface of earth can be specified by *latitude* and *longitude* coordinates. Therefore, using latitude and longitude, you can specify virtually any point on earth.

The following illustration displays the latitude and longitude coordinates:



As a Flutter developer, you can use the Geo-location term which is an identification or estimation of the real-world geographic location of an object, such as a radar source, mobile phone, or Internet-connected computer terminal. In its simplest form, a geo-location involves the generation of a set of geographic coordinates (Latitude and Longitude) and is closely related to the use of positioning systems.

The connection between a GPS satellite and receivers (such as smart devices) is a one-way connection; receivers don't send any information to satellites. The connection works through the use of a procedure called *Trilateration*.

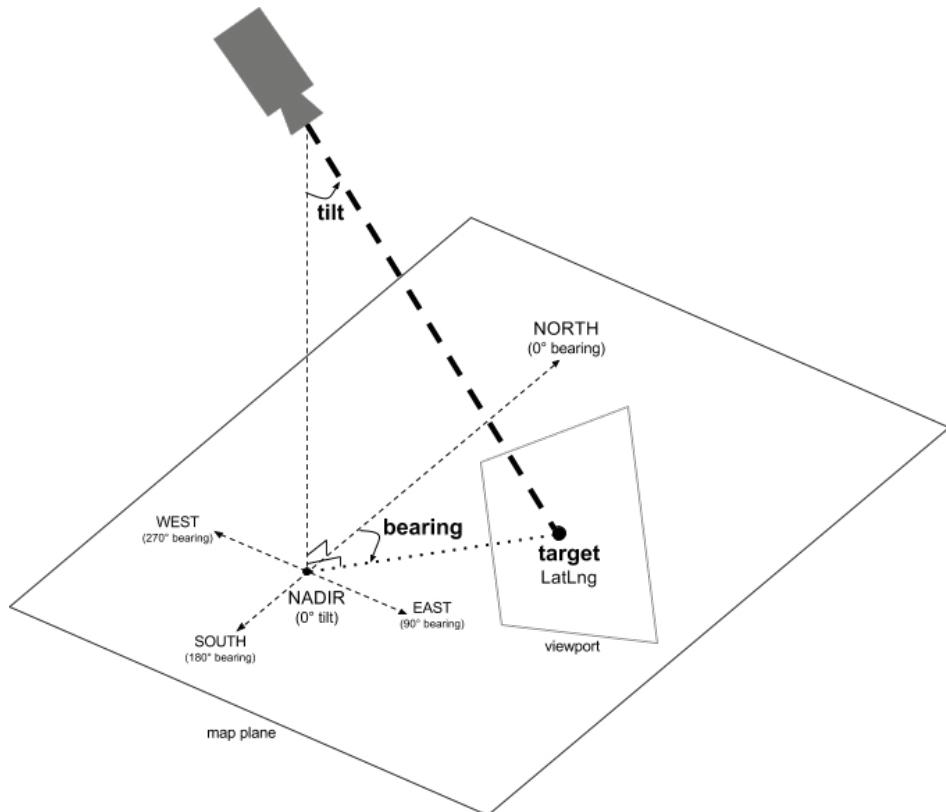
There are various types of navigational satellite systems. The most important one is **NAVSTAR**, which is basically used in all mobile devices. In **NAVSTAR** system, every single point on Earth is covered by at least four GPS satellites, which is enough to calculate the device geo-location coordinates.

As mentioned before, GPS is one of the most accurate geo-location systems. It has accuracy between 7.8~3 meters.

The Android and iOS system provides a reliable API that helps developers capture the user's coordinates using the GPS and other location service providers, which will be listed in the next topic.

## The Camera Position

The map view is modeled as a **camera** looking down on a flat plane. The position of the camera (and hence the rendering of the map) is specified by the following properties: **target** (latitude/longitude location), **bearing**, **tilt**, and **zoom**. You will use these camera attributes later in this lesson to control the camera position or motion.



### Target (location)

The camera target is the location of the center of the map, specified as latitude and longitude co-ordinates.

## Bearing (orientation)

The camera bearing is the direction in which a vertical line on the map points, measured in degrees clockwise from the north. Someone driving a car often turns a road map to align it with their direction of travel, while hikers use a map and a compass usually to orient the map so that a vertical line points north. The Maps API lets you change a map's alignment or bearing. In the "moving the camera" topic of this lesson, you will add this attribute to the map camera position and see how its value affects in changing the camera direction.

## Tilt (viewing angle)

The tilt defines the camera's position on an arc between directly over the map's center position and the surface of the Earth, measured in degrees from the nadir (the direction pointing directly below the camera). When you change the viewing angle, the map appears in perspective, with far-away features appearing smaller, and nearby features appearing larger. In the "moving the camera" topic of this lesson, you will add this attribute to the map camera position and see how its value effects in changing the camera position.

## Zoom

The zoom level of the camera determines the scale of the map. At larger zoom levels, more detail can be seen on the screen, while at smaller zoom levels more of the world can be seen on the screen. At zoom level 0, the scale of the map is such that the entire world has a width of approximately 256 dpi (density-independent pixels).

## Adding Google Maps to a Flutter app

Flutter is Google's mobile app SDK for crafting high-quality native experiences on iOS and Android in record time.

The Google Maps Flutter API lets you display a Google map in your Flutter application. The maps are similar to those you see on Google Maps for mobile apps, and the API exposes many of the same features. With the **Google Maps Flutter plug-in**, you can add maps based on Google maps data to your application. The plug-in automatically handles access to the Google Maps servers, map display, and response to user gestures such as clicks and drags. You can also add markers to your map. These objects provide additional information for map locations, and allow the user to interact with the map.

## Example:

In the following example, you are going to create a new Flutter project that includes a Google map. In this project, you are going to learn about the classes and methods which are used to add and configure a Google map in Flutter applications (Android & iOS) and you will also learn how to determine a specific location on the map (add a marker).

- 1- Open **Android Studio**, Click **File → New → New Flutter Project**
- 2- Select **Flutter Application**, then click **Next**.
- 3- Type: **google\_map** for Project Name, and create a new folder: **Lesson10** for Project Location. Click **Next**.
- 4- Type: **androidatc.com** for Company domain, then click **Finish**

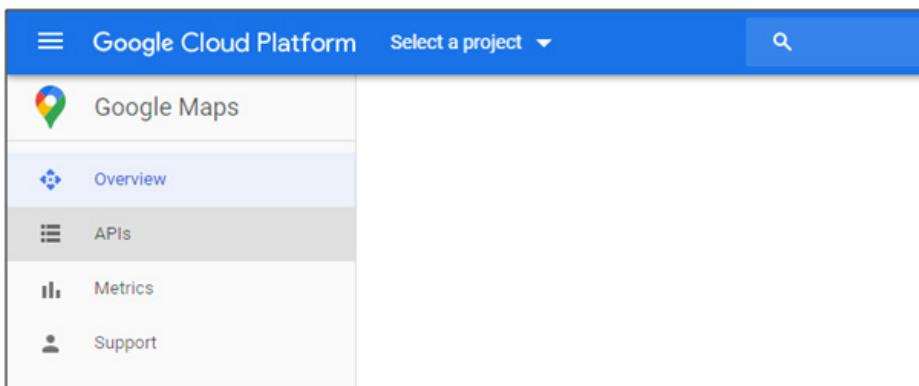
## Getting a Google API key

To use or add Google Maps to your Flutter app, your application needs an API key to access or connect with the Google Maps servers. The type of key you need is called: **API key**.

An API key is a unique identifier that is used to authenticate requests associated with your project for usage and billing purposes. To use the Google Maps Android & iOS API, you must register your app project in the Google API Console and get a Google API key which you can add to your app. To get this API key for Android apps perform the following steps:

- 5- Go to Google cloud platform console: <https://console.cloud.google.com/google/maps-apis>
- 6- Login using your Gmail (user name & password).
- 7- Select your **country**, click **I Agree**, then click **AGREE AND CONTINUE**

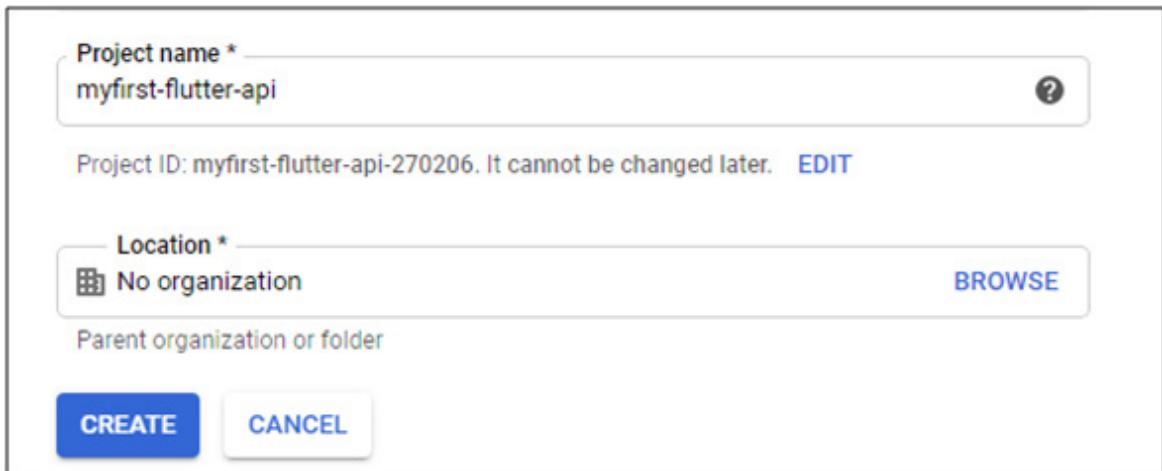
You will get the following web page:



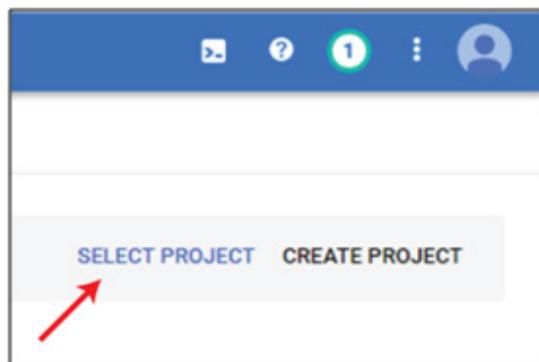
8- In the next step in getting an API key, you should add your project to Google Cloud Platform. To do that, click the **Select a project** drop down list, then you will get the following figure. Click **NEW PROJECT**



9- As illustrated in the following figure, enter your project name. For our example, we used : **myfirst-flutter-api**, you may type what you want as a project name. Click **CREATE**



10- Click: **SELECT PROJECT** as illustrated in the following figure:



11- Click your project name : **myfirst-flutter-api** as illustrated in the following figure:

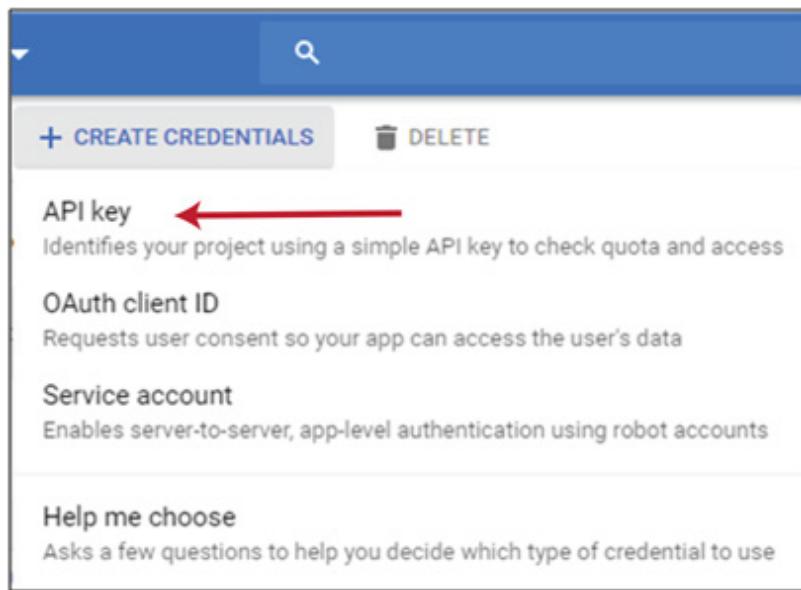
| RECENT                | ALL                 |
|-----------------------|---------------------|
| Name                  | ID                  |
| ▼ No organization     | 0                   |
| • myfirst-flutter-api | myfirst-flutter-api |

12- As illustrated in the following figure, click the:

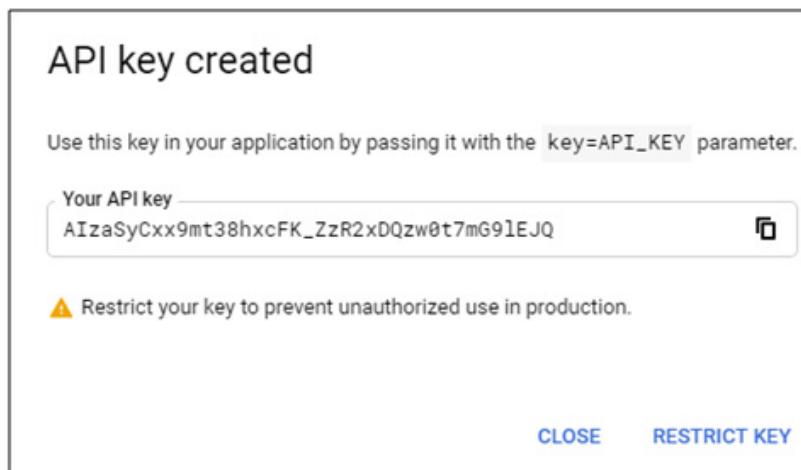
**Google Cloud Platform menu → API & Services → Credentials.**

The screenshot shows the Google Cloud Platform dashboard. A red arrow labeled '1' points from the top-left navigation bar to the 'myfirst-flutter-api' project name. A second red arrow labeled '2' points from the main navigation menu on the left to the 'APIs & Services' option. A third red arrow labeled '3' points from the sub-menu under 'APIs & Services' to the 'Credentials' item, which is highlighted with a light gray background.

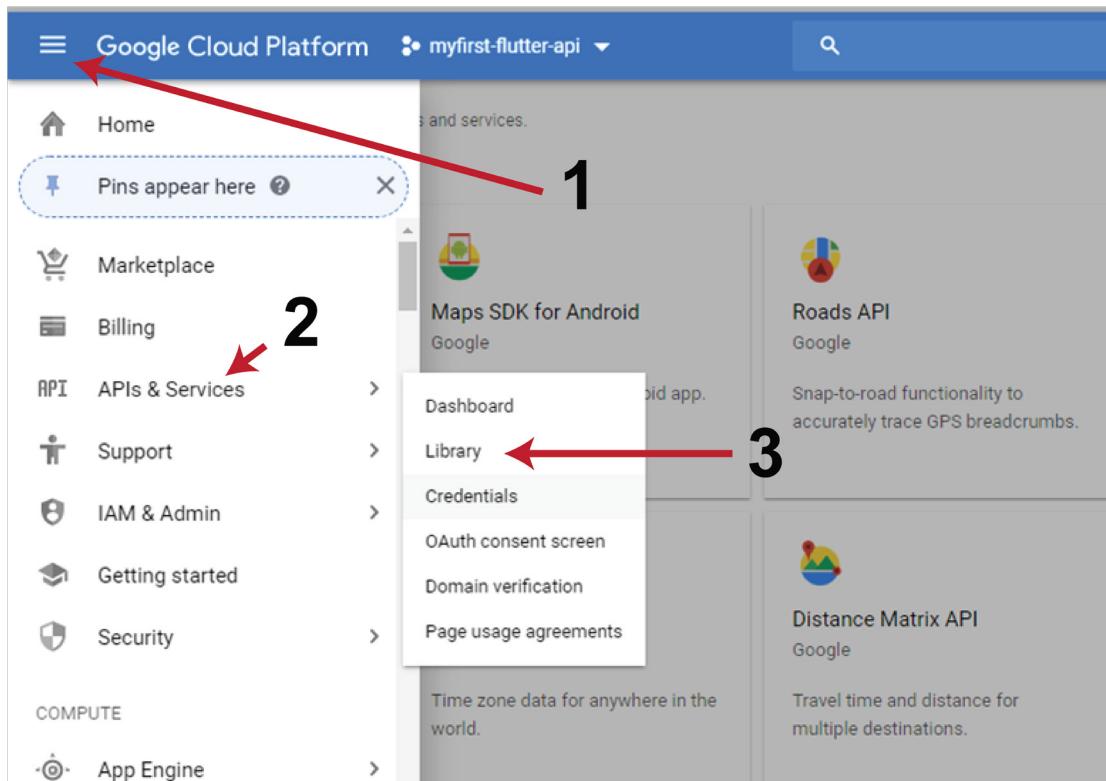
13- Click “**+ CREATE CREDENTIALS**”, then select: **API Key** as illustrated in the following figure:



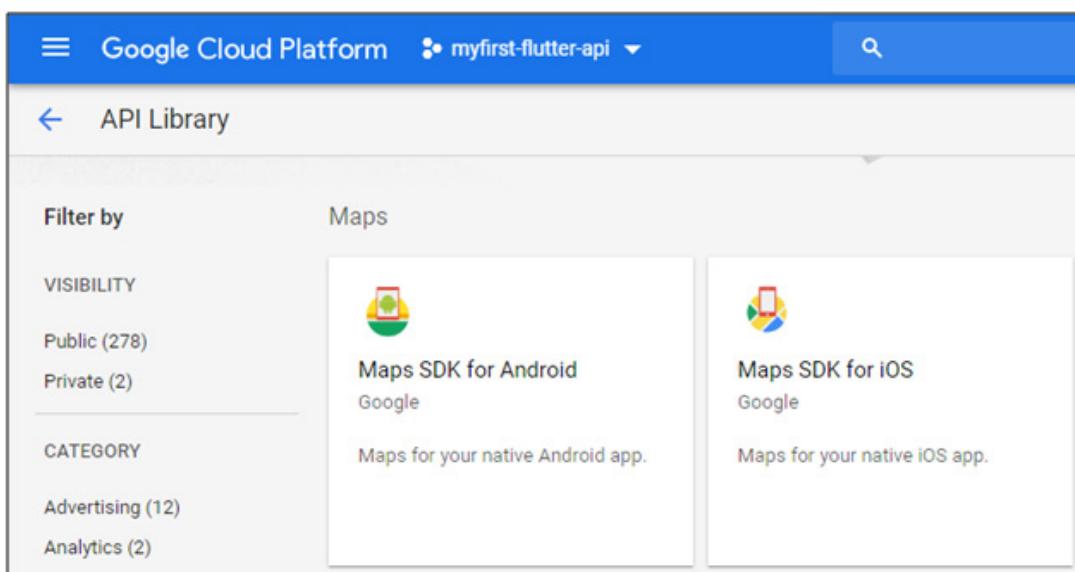
14- You will get the following dialog box which includes your API key. Copy your API key in a safe location (in a separate Notepad or text file) because you will use it later in your Flutter app configurations within the next steps. Click **CLOSE**.



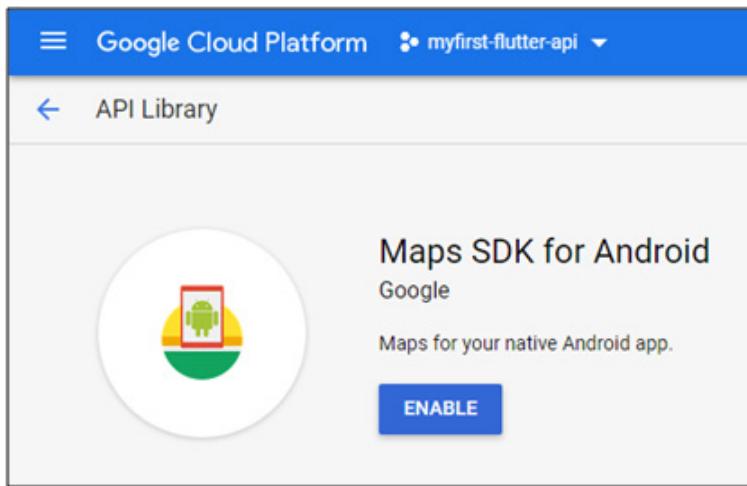
15- Now, to enable **Maps SDK** for iOS and Android click: **Google Cloud Platform → APIs & Services → Library** as illustrated in the following figure:



16- Here, as illustrated in the following figure, click **Maps SDK for Android**



Then, click **Enable** as illustrated in the following figure:



17- Click: **Google Cloud Platform** → **APIs & Services** → **Library**, click **Maps SDK for iOS**, then click **Enable**

Now, the Maps SDK for Android and iOS has been enabled.

This was the last step in configuring your Google Cloud Platform. Now, your Google maps service is ready to exchange the information with your iOS and Android app using your API Key.

## Adding Google Maps Flutter plug-in as a dependency

18- To get the last Google maps plug-in dependency configurations, go to the following web site:

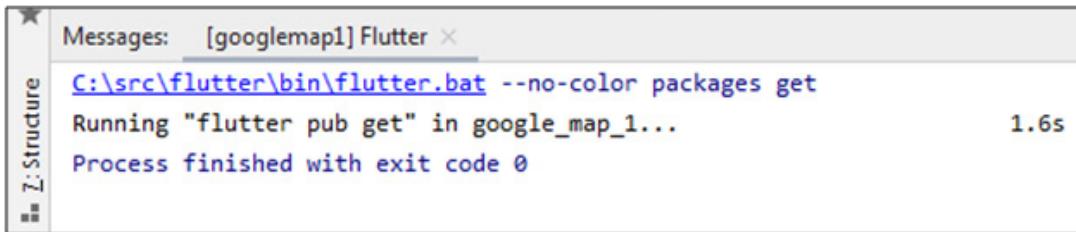
[https://pub.dev/packages/google\\_maps\\_flutter](https://pub.dev/packages/google_maps_flutter), then click the **Installing** tab. Copy the dependencies value. Always copy the value which you will find here to be sure that you will not have any problem with connecting your app to Google maps. Until preparing this example, the value is the following:

```
google_maps_flutter: ^0.5.24+1
```

Go back to your Flutter app and open the **pubspec.yaml** file, then add this dependency value as illustrated in the following figure:

```
dependencies:  
  flutter:  
    sdk: flutter  
  google_maps_flutter: ^0.5.24+1
```

Then, click **Packages get** to update the new configuration in **pubspec.yaml** file. You should not have any error in the Message console. The following figure displays that everything works fine after clicking the **Packages get**:



The screenshot shows the Android Studio message console. The title bar says "Messages: [googlemap1] Flutter". The main area contains the command "C:\src\flutter\bin\flutter.bat --no-color packages get" followed by its execution log: "Running \"flutter pub get\" in google\_map\_1... 1.6s Process finished with exit code 0". The left sidebar shows a tree structure with "L: Structure" selected.

**Note:** Now, you can use the following command in your Dart command:

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

## Adding your API key for your Android app

19- To add your API key to your Android code of your Flutter app, open Android Studio, and open the **AndroidManifest.xml** file which is in the following path:

**android** → **app** → **src** → **main** → **AndroidManifest.xml**

Then, add only the grey highlighted part of the following code in the **application** tag. Here, use your API key which you have generated on the Google cloud console instead of in this example.

```
<application
    android:name="io.flutter.app.FlutterApplication"
    android:label="googlemap1"
    android:icon="@mipmap/ic_launcher">

    <meta-data android:name="com.google.android.geo.API_KEY"
        android:value="AIzaSyBu8jtYmJyL9FCsG_L2r9sbldoT7oKFoQ4"/>

<activity
```

Click **File** → **Save ALL** and close the **AndroidManifest.xml** file.

## Adding your API key for your iOS app

Unlike Android, adding an API key on iOS requires changes to the source code of the *Runner* app. The **AppDelegate** is the core singleton that is part of the app initialization process.

20- In **Android Studio**, open the following:

**ios → Runner → AppDelegate.swift**

Make two changes in this file. First, add an **import** statement to pull in the Google Maps headers, then call the **provideAPIKey()** method of the **GMServices** singleton. This API key enables Google Maps to correctly serve map tiles. To do that, perform the following steps:

Add only the grey highlighted code to this file and use your API key instead, which you have in this example:

```
import UIKit
import Flutter
import GoogleMaps

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
    override func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [UIApplication
LaunchOptionsKey: Any]?
    ) -> Bool {
        GMServices.provideAPIKey("AIzaSyCxx9mt38hxcFK_ZzR2xDQzw0t7mG91EJQ")

        GeneratedPluginRegistrant.register(with: self)
        return super.application(application,
        didFinishLaunchingWithOptions: launchOptions)
    }
}
```

Close the **AppDelegate.swift** file.

21 - You also need to add a setting to **Info.plist** file (**ios/Runner/Info.plist**). This entry forces Flutter on iOS into a single threaded mode, which is required for the platform view embedding to work. This technical restriction is being worked on and will be lifted before Google Maps moves out of Developer Preview.

Open the following file: **ios → Runner → Info.plist**, then add only two grey highlighted lines of code to the **Info.plist** file as illustrated in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>io.flutter.embedded_views_preview</key>
    <true/>
    <key>CFBundleDevelopmentRegion</key>
    <string>$(DEVELOPMENT_LANGUAGE)</string>
    <key>CFBundleExecutable</key>
    <string>$(EXECUTABLE_NAME)</string>
    <key>CFBundleIdentifier</key>
    <string>$(PRODUCT_BUNDLE_IDENTIFIER)</string>
    <key>CFBundleInfoDictionaryVersion</key>
```

Click **File → Save ALL**, then close the **Info.plist** file.

## Adding a Google Map on Your Flutter App Screen

In this step, you will configure your app interface to display the Google map using **GoogleMap()** widget.

Before adding the code which will add the Google map to your app interface, check the following details which help in understanding the classes and methods that you need to use in adding **GoogleMap** widget to your app widget tree.

To add Google map to your app interface, you should use the **GoogleMap** widget or class. This is the main class of Google Maps SDK for Android and iOS apps and is the entry point for all methods related to the map. The following code is an example of using **GoogleMap** widget:

```
GoogleMap(  
    onMapCreated: _myMapCreated,  
    initialCameraPosition: CameraPosition(target:_location, zoom:11.0),  
)
```

To instantiate **GoogleMap**, you need to use **GoogleMapController** class, where the map view can be controlled with the **GoogleMapController** class, that is passed to the **GoogleMap**'s **onMapCreated** callback.

The following code is an example of configuring **GoogleMapController** class, and **onMapCreated** callback. You should configure this class before using **GoogleMap** widget in your app.

```
GoogleMapController mapController;  
  
void _myMapCreated(GoogleMapController controller) {  
    mapController = controller;  
}
```

You can adjust the viewpoint of a map by changing the position of the camera (as opposed to moving the map). You may use the map's camera to set parameters such as location, zoom level, tilt angle, and bearing. In the previous code of the **GoogleMap** widget, we used the following configuration to determine the position on the Google map.

```
initialCameraPosition: CameraPosition(target:_location, zoom:11.0),
```

**\_location** function should be declared before using the **GoogleMap** widget as follows:

```
final LatLng _location = const LatLng(45.521563, -122.677433);
```

**LatLng** is a public final class (immutable class) representing a pair of latitude and longitude coordinates.

The zoom value in the **CameraPosition** class represents the zoom level of the map (zoom in or out).

22- You can now add a **GoogleMap** widget to your widget tree. Open **main.dart** file, delete all code, then add the following code:

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

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

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

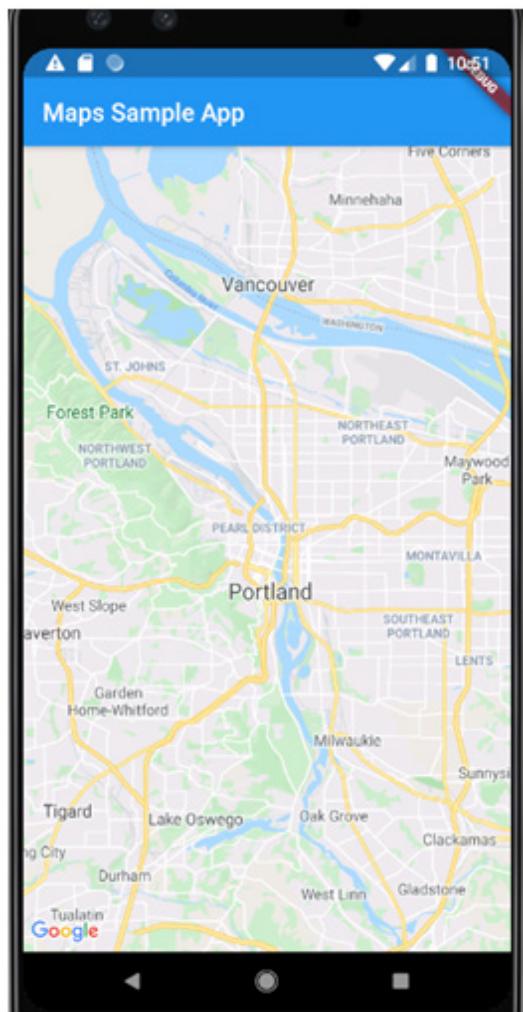
class _MyAppState extends State<MyApp> {

  final LatLng _location = const LatLng(45.521563, -122.677433);

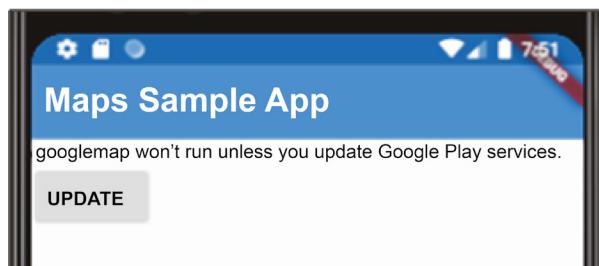
  GoogleMapController mapController;
  void _myMapCreated(GoogleMapController controller) {
    mapController = controller;
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Maps Sample App'),
        ),
        body: GoogleMap(
          onMapCreated: _myMapCreated,
          initialCameraPosition: CameraPosition(target:_location,
          zoom:13.0),
          ),
        ),
      );
  }
}
```

23- Run your app. You should get the following figure:



**Note:** Depending on your emulator type and settings, you may get a message asking you to update Google Play services before getting the Google map as illustrated in the following figure. Click this **UPDATE** button, following the next steps, and then your emulator will work fine.

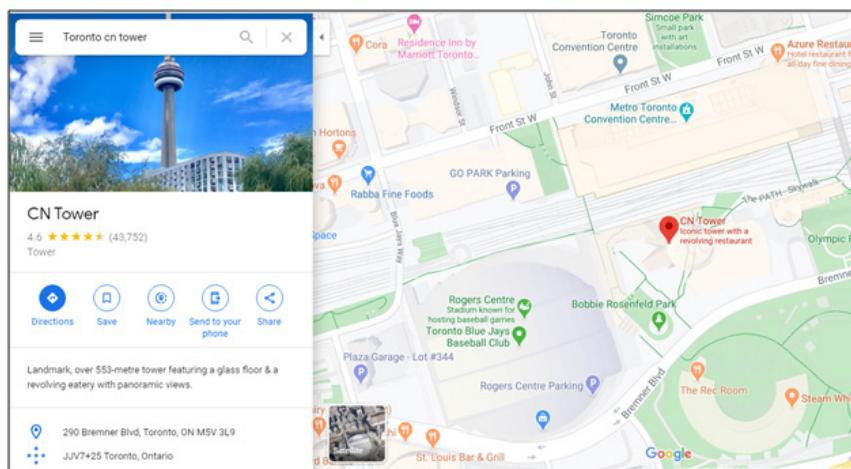


If you make any changes in your code, before running your app to see the new Google map configurations, stop your app and run it again.

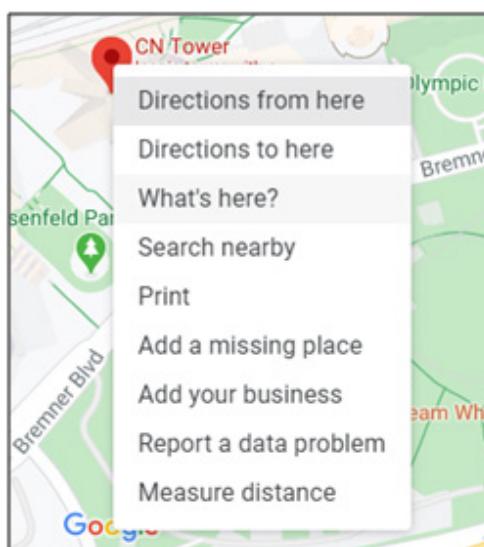
## Adding a Google Map Marker

In the previous steps, you configured your app to show Google map for a specific location depending on its latitude and longitude values.

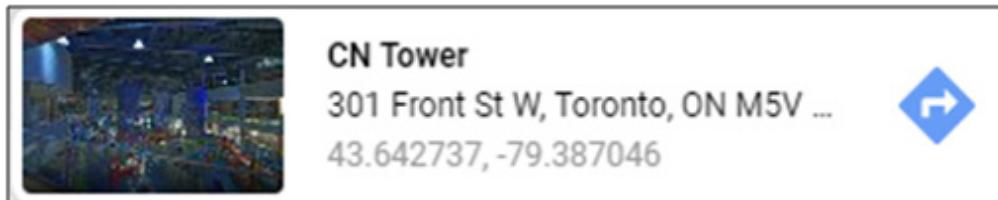
In this topic you will add a marker (pin) to your map to display the location you want. In this example by using the same previous code, you are going to add a map marker for Toronto CN tower in Toronto, Canada. You must have the latitude and longitude for this location to add it to your Google map. To get the latitude and longitude coordinates for any location, you can simply go to [www.google.com/maps](http://www.google.com/maps) web site and type the address you want, to know its latitude and longitude coordinates. In this example, just type Toronto CN tower, then press Enter (or Return for Mac). You will get the following figure:



To get this address latitude and longitude coordinates, right click the marker as illustrated below, then select → **What's here?**



You will get the following result:



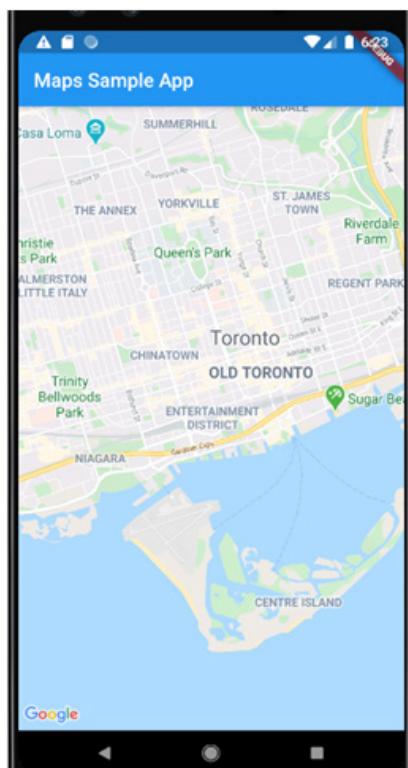
Also, you can go to the address bar of your web browser and copy the values that come after the @ sign. The first value is the latitude and the second value is the longitude as illustrated in the following figure:



24- Copy the latitude and longitude values from your web address bar, then use these values in the previous code as follows:

```
final LatLng _location = const LatLng(43.6425701, -79.3892455);
```

Then, stop and run your app again. You will get the following run output:





There are a lot of web sites that provide free location services, all you need to do is type the address and you will get the latitude and longitude coordinates.

**Important Note:** If you want to know all the methods and configurations you may use with **GoogleMap** class, hold Ctrl key (or command key for Mac), then click the GoogleMap widget in your code. Another tab will open including **google\_map.dart** file content. You may use these methods in your app with this widget. You may use the same technique with all other Flutter widgets to know which methods you may use.

25- If you are creating an app for a restaurant or another company which has many branches, and you want to display these branches on your app Google map, you should create and add a **marker** for each branch, then add them to your app Google map widget. Therefore, it is a good idea to organize your app code by creating a separate Dart file including all the markers which you want to add to your app map, then call these markers to your app interface.

Each **Marker** class has its latitude and longitude values.

Right click **lib** folder, then select **New → Dart File**

Type: **MapMarkers** for the file name and then press **Enter**.

26- Open **MapMarkers.dart** file, then add the following code which is related to creating the Marker class for Toronto CN tower.

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

Marker cnTowerMarker = Marker(
  markerId: MarkerId("CN Tower"),
  position: const LatLng(43.6425701, -79.3892455),
  infoWindow: InfoWindow(title: "Toronto Tower"),
  icon: BitmapDescriptor.defaultMarkerWithHue(BitmapDescriptor.hueMagenta));
```

27- Now, open **main.dart** file and add at the top of this file the following:

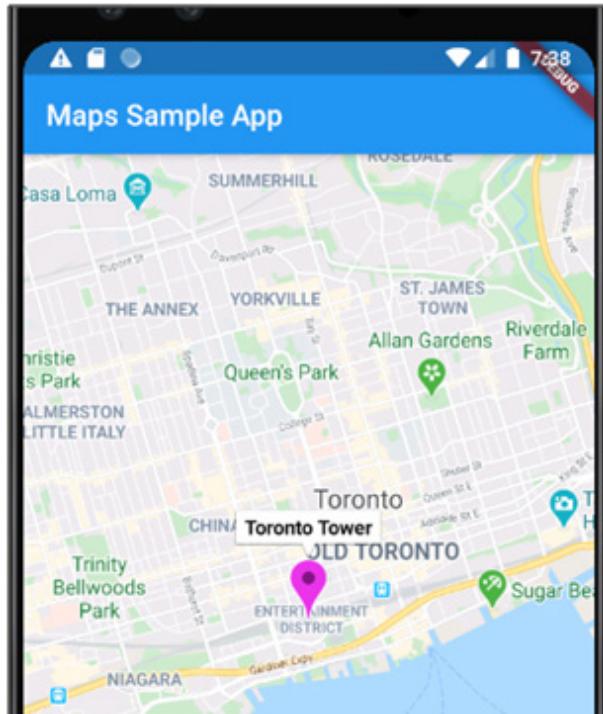
```
import 'MapMarkers.dart';
```

Then, add the `cnTowerkarker` marker to your `GoogleMap` widget as illustrated in the grey highlighted part of the following code:

```
body: GoogleMap(  
  markers: {cnTowerkarker},  
  onMapCreated: _myMapCreated,  
  initialCameraPosition: CameraPosition(target: _location, zoom:13.0),  
)
```

28- Stop your app and run it again. You should get the following run output:

If you click the marker (pin), you will get the marker title value: **Toronto Tower**.



## Google Map Types

There are about five types of Google maps available to configure with `GoogleMap` widgets. These are as follows:

**Normal:** It is the default value that you get when you run your app in the last time.

**Satellite:** Displays Google Earth satellite images.

**Hybrid:** Displays a mixture of normal and satellite views.

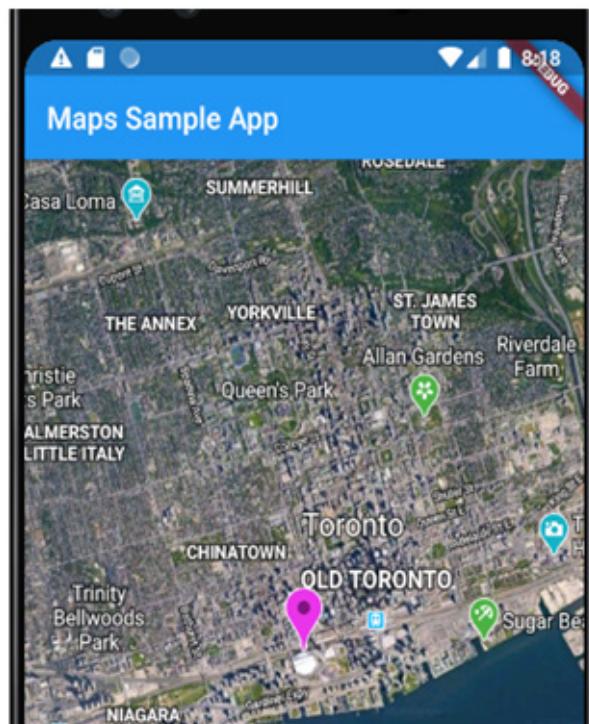
**Terrain:** Displays a physical map based on terrain information.

**None:** You will get an empty map. Only you can see the map markers.

29- Configure your map type to be: **hybrid** as illustrated in the following code:

```
body: GoogleMap(  
    mapType: MapType.hybrid,  
    markers: {cnTowerkarker},  
    onMapCreated: _myMapCreated,  
    initialCameraPosition: CameraPosition(target: _location, zoom:13.0),  
) ,
```

30- Stop your app and run it again. The run output follows:



Try other types of maps to understand how each one looks like.

## Moving the Camera (Camera Animation)

The Maps API allows you to change which part of the world is visible on the map. This is achieved by changing the position of the camera (as opposed to moving the map).

When you change the camera, you have the option of animating the resulting camera movement. The animation interpolates between the current camera attributes and the new camera attributes.

This will add some interactivity to your Google map. For example, when you add a floating action button or icon to your app map and when the app user taps this button or icon, the camera position will be moved from the current position to a specific location. To do that, you should add and configure the **GoogleMapController** class in your app as you will see in this topic.

Use the same code as used previously and add a floating button to your current map. When the app user taps this button, the camera position will be moved from the current coordinates (Toronto) to **Niagara Falls** ( this location is on the border between Ontario, Canada and the American state of New York ). If you want to get the new location coordinates, open Google maps and find its coordinates. Niagara Falls (new location) coordinates are :

Latitude: 43.0807790 and Longitude: 79.0785500

You need to add the following code to your app:

```
static final CameraPosition NiagaraFalls =  
    CameraPosition(target: LatLng(43.0807790, -79.0785500),  
zoom:14.0);  
  
Future<void> gotoNiagaraFalls() async {  
    final GoogleMapController controller = await mapController;  
    controller.animateCamera(  
        CameraUpdate.newCameraPosition(NiagaraFalls),  
    );  
}
```

Also, you should add a flat action button to your interface. When the app user taps this button, he/she will call **gotoNiagaraFalls** function. The flat action button code follows:

```
floatingActionButton: FloatingActionButton.extended(  
    onPressed: gotoNiagaraFalls,  
    label: Text('To Niagara Falls!'),  
    icon: Icon(Icons.directions_boat),  
) ,
```

31- The full code of **main.dart** file follows:

```
import 'package:flutter/material.dart';  
import 'package:google_maps_flutter/google_maps_flutter.dart';  
  
import 'MapMarkers.dart';  
  
void main() => runApp(MyApp());  
  
class MyApp extends StatefulWidget {  
    @override  
    _MyAppState createState() => _MyAppState();  
}  
  
class _MyAppState extends State<MyApp> {  
    final LatLng _location = const LatLng(43.6425701, -79.3892455);  
  
    GoogleMapController mapController;  
    void _myMapCreated(GoogleMapController controller) {  
        mapController = controller;  
    }  
  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Scaffold(  
                appBar: AppBar(  
                    title: Text('Maps Sample App'),  
                ),  
                body: GoogleMap(  
                    initialCameraPosition:  
                        CameraPosition(  
                            target: _location,  
                            zoom: 12.0,  
                        ),  
                    markers:  
                        Set<Marker>(  
                            MapMarkers().getMarkers(),  
                        ),  
                    onMapCreated: _myMapCreated,  
                ),  
            ),  
        );  
    }  
}
```

```
markers: {cnTowerkarker},
onMapCreated: _myMapCreated,
initialCameraPosition: CameraPosition(target: _location, zoom:13.0),
),

floatingActionButton: FloatingActionButton.extended(
    onPressed: gotoNiagaraFalls,
    label: Text('To Niagara Falls!'),
    icon: Icon(Icons.directions_boat),
),
),
);
}

static final CameraPosition NiagaraFalls =
    CameraPosition(target: LatLng(43.0807790, -79.0785500), zoom:14.0);

Future<void> gotoNiagaraFalls() async {
    final GoogleMapController controller = await mapController;
    controller.animateCamera(
        CameraUpdate.newCameraPosition(NiagaraFalls),
    );
}
}
```

32- Stop your app and run it again. When you tap the **To Niagara Falls** button, your map camera position will be moved to Niagara Falls map.

33- Add to the current **CameraPosition** class the following grey highlighted attributes (bearing & lilt) which will add the type of camera animation when your camera position moves from the current location (Toronto) to Niagara Falls.

```
static final CameraPosition NiagaraFalls = CameraPosition(
    target: LatLng(43.0807790, -79.0785500),
bearing: 45.0,
tilt: 50.0,
    zoom: 14.0,
);
```

Stop your app , run it again ,and click the button: **To Niagara Falls**. Now, you have added an animation effect to your camera motion.

## Capture an App User's Location

Some apps use the users' location coordinates to give them information related to their locations such as nearby restaurant, fuel stations, health or transportation services, etc.

Flutter SDK provides a simplified way to implement both a GPS and a Network provider. This topic explains how to implement both providers and which provider is more suitable for the application.

Capturing a user's location is a time consuming process especially when using a GPS. Moreover, capturing any location is not allowed to be executed on Main thread or it will be blocked by the operating system. Surely, this will lead to creating a new thread. Flutter SDK will handle the way of capturing coordinates using a thread creation.

You can get the app user's location on Google map if you use the **Flutter Geo-locator Plug-in** which is available at: <https://pub.dev/packages/geolocator#readme-tab>

A Flutter Geo-location plug-in which provides easy access to the platform specific location services. This plug-in has the following features:

- Get the current location of the device.
- Get the last known location.
- Get continuous location updates.
- Check if location services are enabled on the device.
- Translate an address to geo-coordinates and vice versa (a.k.a. Geo-coding);
- Calculate the distance (in meters) between two geo-coordinates.
- Check the availability of Google Play services (on Android only).

The lab of this lesson includes how to configure your Flutter app (iOS & Android) step by step to capture the app user's location (coordinates) on Google map.

## Lab10: Location-Aware Apps Using GPS and Google Maps

- Getting a Google API key
- Creating an App Interface
- Configuring Your App to Use Your API Key
- Adding a Google Map on Your Flutter App Screen
- Adding a Google Map Marker
- Capturing Users' Location
- Configuring User App's Permission

In this lab, you will create a new Flutter project that includes a Google map. You will configure this map to have a map marker for a specific location, and configure this app to get the app user's coordinates and display them as a location on the Google map. This part is important in the real mobile development because a lot of companies need their app users to find the nearest store or branch depending on their location on the map.

If you find any step of this lab unclear, please go back to the lesson where you will find everything in detail. Also, the lab source code files include the files of this lab. It is important to create this lab step by step to obtain the maximum benefits.

## Getting a Google API key

To use or add Google Maps to your Flutter app, your application needs an API key to access or connect with the Google Maps servers. The type of key you need is called: **API key**.

To get this API Key, perform the following steps:

- 1- Go to Google cloud platform console: <https://console.cloud.google.com/google/maps-apis>
- 2- Login using your Gmail (user name & password).
- 3- Select your **country**, click **I Agree**, then click **AGREE AND CONTINUE**

You will get the following web page:

The screenshot shows the Google Cloud Platform dashboard. At the top, there is a blue header bar with the text "Google Cloud Platform" and "Select a project". Below the header is a sidebar with the following options: "Google Maps" (selected), "Overview", "APIs" (selected), "Metrics", and "Support". The main content area is currently empty.

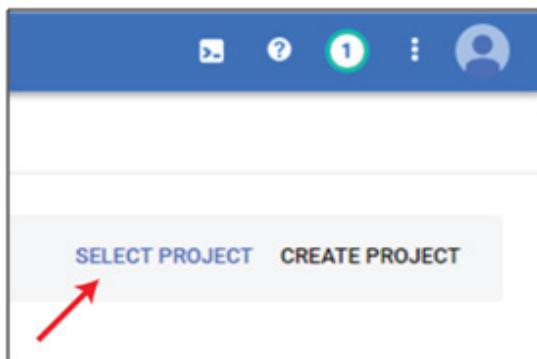
4- In the next step for getting an API key, you should add your project to Google Cloud Platform. To do that, click the **Select a project** drop down list, then you will get the following figure. Click **NEW PROJECT**

The screenshot shows a modal dialog box titled "Select a project". It contains a search bar labeled "Search projects and folders" with a magnifying glass icon. Below the search bar are two tabs: "RECENT" and "ALL" (which is selected). A table lists one item: "No organization" with an ID of "0". In the top right corner of the dialog, there is a button labeled "NEW PROJECT" with a gear icon, and a red arrow points to this button.

5- As illustrated in the following figure, type your project name. For our example, we used : **myfirst-flutter-api**, you may type what you want as a project name. Click **CREATE**

The screenshot shows a dialog box for creating a new project. At the top, there is a field labeled "Project name \*" containing "myfirst-flutter-api". To the right of this field is a help icon (a question mark inside a circle). Below this, a note says "Project ID: myfirst-flutter-api-270206. It cannot be changed later." with a "EDIT" link. Under "Location \*", there is a dropdown menu showing "No organization" and a "BROWSE" button. A "Parent organization or folder" field is below it. At the bottom left is a blue "CREATE" button, and at the bottom right is a white "CANCEL" button.

6- Click : **SELECT PROJECT** as illustrated in the following figure:

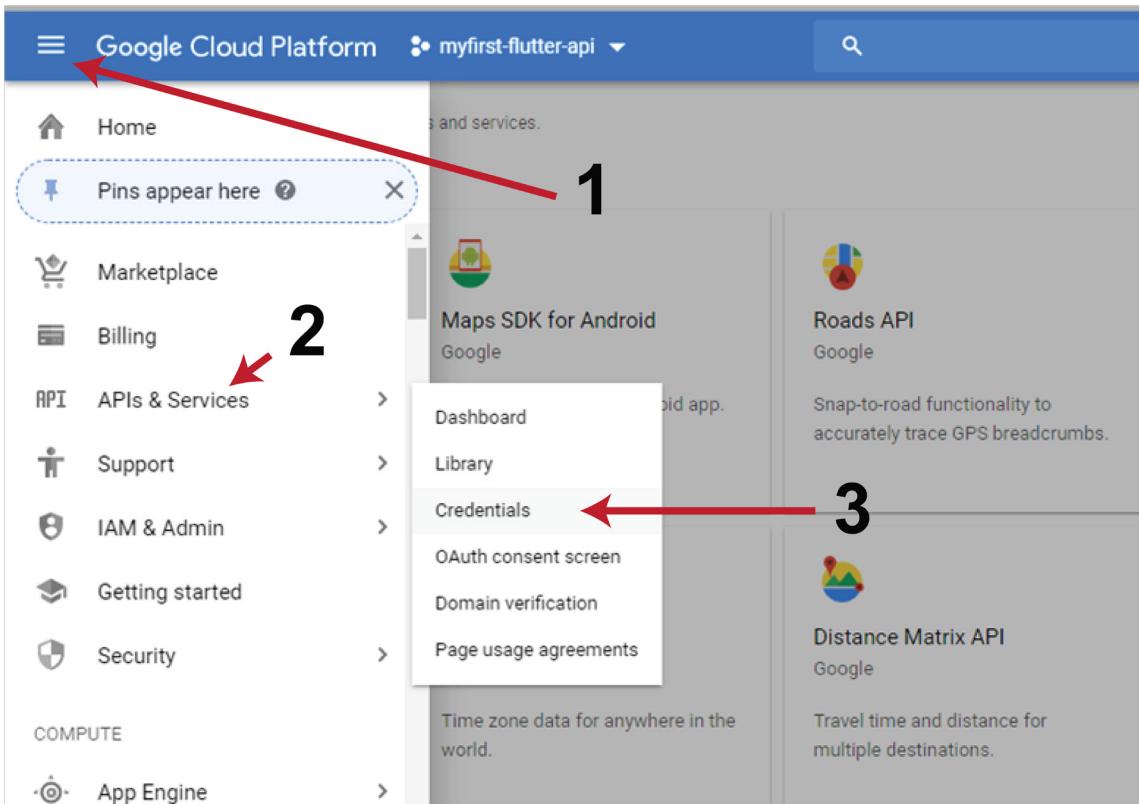


7- Click your project name: **myfirst-flutter-api** as illustrated in the following figure:

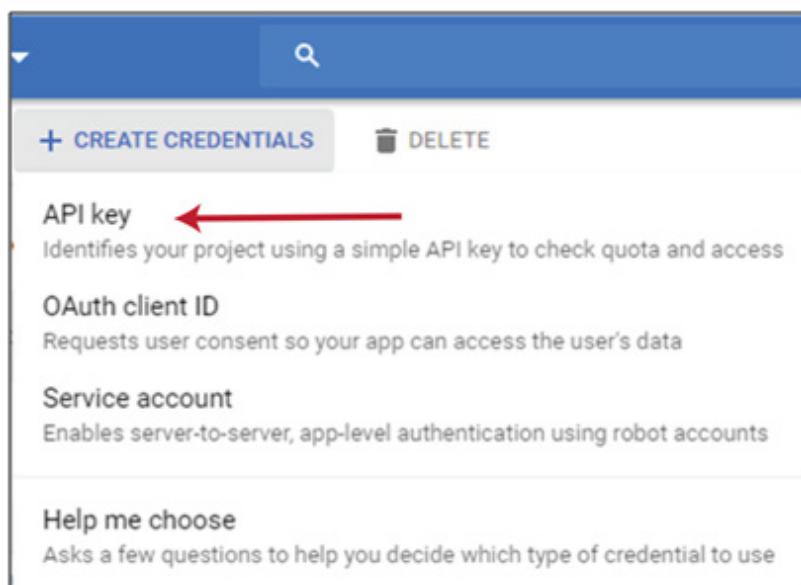
The screenshot shows a list of projects. At the top, there are tabs for "RECENT" and "ALL", with "ALL" being selected. Below this is a table with columns for "Name", "ID", and "Last updated". The first row shows "No organization" with ID "0". The second row shows "myfirst-flutter-api" with ID "myfirst-flutter-api". The "Last updated" column is empty for both rows.

Name	ID	Last updated
No organization	0	
myfirst-flutter-api	myfirst-flutter-api	

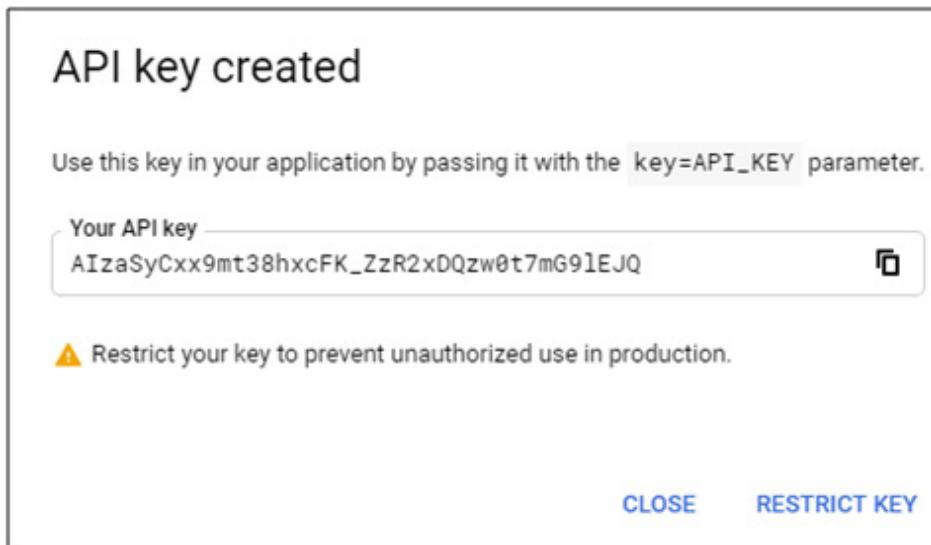
8- As illustrated in the following figure, click the **Google Cloud Platform** menu → **API & Services** → **Credentials**.



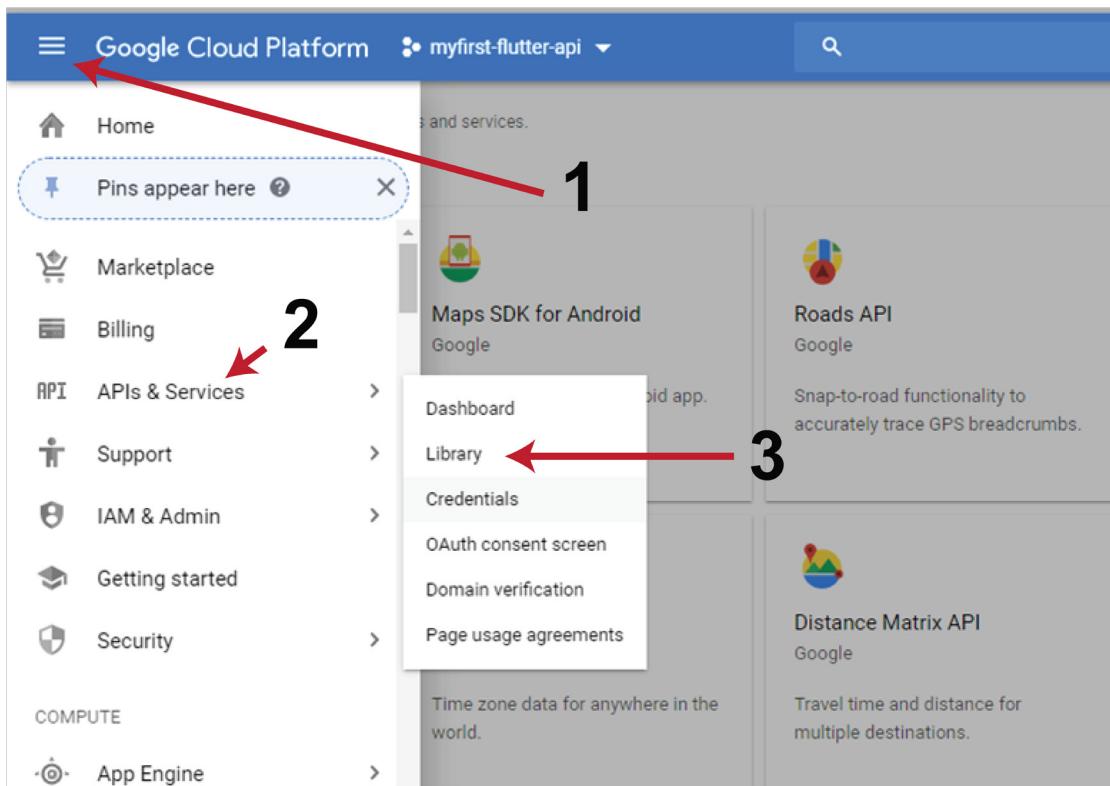
9- Click “**+ CREATE CREDENTIALS**”, then select: **API Key** as illustrated in the following figure:



10- Then, you will get the following dialog box which includes your API key. Copy your API key in a safe location (in a separate Notepad or text file) because you will use it later in your Flutter app configurations in the next steps. Click **CLOSE**.



11- Now, to enable **Maps SDK** for iOS and Android click: **Google Cloud Platform → APIs & Services → Library** as illustrated in the following figure:



12- Here, as illustrated in the following figure, click **Maps SDK for Android**

The screenshot shows the Google Cloud Platform API Library interface. The left sidebar has 'Filter by' sections for 'VISIBILITY' (Public 278, Private 2) and 'CATEGORY' (Advertising 12, Analytics 2). The main area is titled 'Maps' and contains two items:

- Maps SDK for Android** (Google): Description: 'Maps for your native Android app.' A blue 'ENABLE' button is visible at the bottom right.
- Maps SDK for iOS** (Google): Description: 'Maps for your native iOS app.'

Click **Enable** as illustrated in the following figure.

This screenshot shows the details for the 'Maps SDK for Android'. It features a circular icon with an Android logo. The title is 'Maps SDK for Android' (Google), with the description 'Maps for your native Android app.' Below the description is a prominent blue 'ENABLE' button.

13- Click: **Google Cloud Platform → APIs & Services → Library**, click **Maps SDK for iOS**, then click **Enable**

Now, the Maps SDK for Android and iOS has been enabled.

## Creating an App Interface

14- Open Android Studio, click **File** → **New** → **New Flutter Project**

15- Select **Flutter Application** and click **Next**.

16- Type: **lab\_10** for Project Name, and create a new folder: **Lab\_10** for Project Location. Click **Next**.

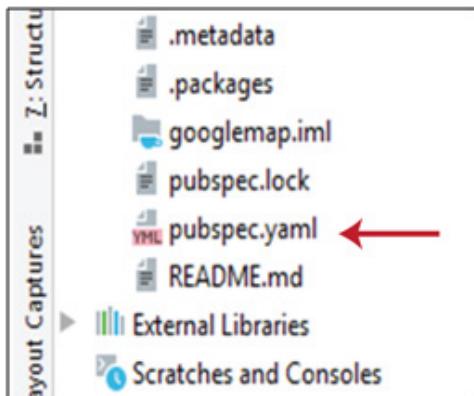
17- Type: **androidatc.com** for Company domain and click **Finish**

18- To get the last Google maps plug-in dependency configurations, go to the following web site:

[https://pub.dev/packages/google\\_maps\\_flutter](https://pub.dev/packages/google_maps_flutter) and click the Installing tab. Copy the dependencies value. Always copy the value which you will find here to be sure that you will not have any problem with connecting your app with Google maps. Until this lab was being prepared, the value was as follows:

**google\_maps\_flutter: ^0.5.24+1**

Then go back to your Flutter app and open the **pubspec.yaml** file which is the location in the following path : **lab\_10 → pubspec.yaml** , as illustrated in the following figure:



Then, add the dependency value as illustrated in the following figure:

```
dependencies:  
  flutter:  
    sdk: flutter  
  google_maps_flutter: ^0.5.24+1
```

19- Click **Packages get** to update the new configuration in **pubspec.yaml** file. You should not have any error in your Android Studio Message console.

## Configuring Your App to Use Your API Key

20- To add your API key to your Android code of your Flutter app, open **Android Studio**, then open the **AndroidManifest.xml** file which is in the following path:

**android → app → src → main → AndroidManifest.xml**

Then, add only the grey highlighted part of the following code in the **application** tag. Here, use your API key which you have generated on Google cloud console instead of the one in this lab.

```
<application  
    android:name="io.flutter.app.FlutterApplication"  
    android:label="googlemap1"  
    android:icon="@mipmap/ic_launcher">  
  
    <meta-data android:name="com.google.android.geo.API_KEY"  
              android:value="AIzaSyBu8jtYmJyL9FCsG_L2r9sbldoT7oKFoQ4"/>  
  
<activity
```

21- Click **File → Save ALL** and close the **AndroidManifest.xml** file.

22- Unlike Android, adding an API key on iOS requires changes to the source code of the Runner app. Open **AppDelegate.swift** file which is in the following path:

**ios → Runner → AppDelegate.swift**

Make two changes in this file. First, add an **import** statement to pull in the Google Maps headers, and then call the **provideAPIKey()** method from the **GMServices** singleton. This API key enables Google Maps to correctly serve map tiles. To do that, perform the following steps:

Add the grey highlighted code to this file and use your API key instead of which you have in this lab:

```
import UIKit
import Flutter
import GoogleMaps

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
    override func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
    ) -> Bool {
        GMSServices.provideAPIKey("AIzaSyCxx9mt38hxcFK_ZzR2xDQzw0t7mG91EJQ")
        GeneratedPluginRegistrant.register(with: self)
        return super.application(application,
            didFinishLaunchingWithOptions: launchOptions)
    }
}
```

Close the **AppDelegate.swift** file.

23 - You also need to add a setting to **Info.plist** file (ios/Runner/Info.plist).

Open the following file: **ios → Runner → Info.plist** , and then to this file, add the only two grey highlighted lines of code to the **Info.plist** file as illustrated in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>io.flutter.embedded_views_preview</key>
    <true/>
    <key>CFBundleDevelopmentRegion</key>
    <string>$(DEVELOPMENT_LANGUAGE)</string>
    <key>CFBundleExecutable</key>
    <string>$(EXECUTABLE_NAME)</string>
```

```
<key>CFBundleIdentifier</key>
<string>$(PRODUCT_BUNDLE_IDENTIFIER)</string>
<key>CFBundleInfoDictionaryVersion</key>
```

Click File → Save ALL and close the Info.plist file.

## Adding a Google Map on Your Flutter App Screen

In this step, you will configure your app interface to display the Google map using `GoogleMap()` widget.

24- To add a `GoogleMap` widget to your widget tree. Open `main.dart` file, delete all code and add the following code:

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

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

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

class _MyAppState extends State<MyApp> {

  final LatLng _location = const LatLng(43.6425701, -79.3892455);

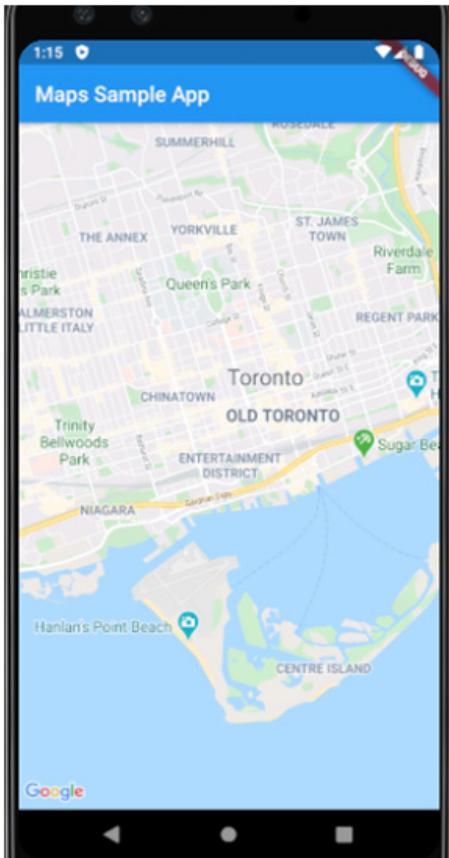
  GoogleMapController mapController;
  void _myMapCreated(GoogleMapController controller) {
    mapController = controller;
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Maps Sample App'),

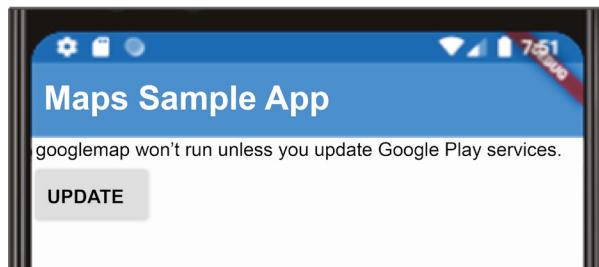
```

```
    ),  
  
body: GoogleMap(  
    onMapCreated: _myMapCreated,  
    initialCameraPosition: CameraPosition(target:_location, zoom:13.0),  
  
    ),  
    ),  
);  
}  
}
```

25- Run your app. You should get the following figure:



**Note:** Depending on your emulator type and settings, you may get a message asking you to update Google Play services before getting Google map as illustrated in the following figure. Click this **UPDATE** button, following the next steps and your emulator will be working fine.



**Note:** If you make any changes in your code, before you run your app to see the new Google map configurations, stop your app, and then run it again.

## Adding a Google Map Marker

In this part of this lab, you will add a map marker for Toronto CN Tower which has the following coordinates: 43.6425701, -79.3892455

You will create a separate Dart file to configure the map marker.

26 - Right click **lib** folder, and then select **New → Dart File**

Type: **MapMarkers** for the file name and press **Enter**.

27- Open **MapMarkers.dart** file, then add the following code which is related to create the Marker class for Toronto CN tower.

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

Marker cnTowerMarker = Marker(
  markerId: MarkerId("CN Tower"),
  position: const LatLng(43.6425701, -79.3892455),
  infoWindow: InfoWindow(title: "Toronto Tower"),
  icon: BitmapDescriptor.defaultMarkerWithHue(BitmapDescriptor.hueMagenta));
```

28- Now, open **main.dart** file and add the following line of code at the top of this file:

```
import 'MapMarkers.dart';
```

Then, add the `cnTowerMarker` marker to your `GoogleMap` widget as illustrated in the grey highlighted part of the following code:

```
body: GoogleMap(  
    markers: {cnTowerkarker},  
    onMapCreated: _myMapCreated,  
    initialCameraPosition: CameraPosition(target: _location, zoom:13.0),  
)
```

29- Stop your app and run it again. You should get the following run output:

If you click the red marker (red pin), you will get the marker title value: **Toronto Tower**.

## Capturing Users' Location

To configure your app to capture the user's location on Google map, perform the following steps using the previous Flutter app code:

30- To add the Flutter **Geolocator** plug-in to your app and use all its libraries, add the **geolocator** as a dependency in your **pubspec.yaml** file as illustrated in the following figure:

```
dependencies:  
  flutter:  
    sdk: flutter  
  google_maps_flutter: ^0.5.24+1  
  geolocator: ^5.3.0
```

Then, click the **Packages get** to download all **geolocation** libraries.

31- To enable **Geolocation** plug-in for Android devices, upgrade Android **Gradle** by open the following file:

**android → gradle.properties**

Be sure that you have the following settings in your **gradle.properties** file.

```
android.useAndroidX=true  
android.enableJetifier=true
```

If Not, add them. The file should have the following settings:

```

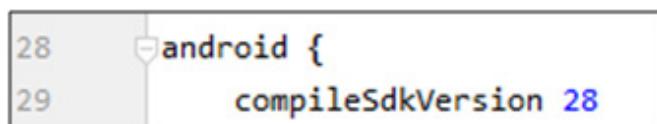
Flutter commands
1 org.gradle.jvmargs=-Xmx1536M
2 android.enableR8=true
3 android.useAndroidX=true
4 android.enableJetifier=true

```

32- Open the **build.gradle** file which is in the following path:

**android → app → build.gradle**

Make sure that the SDK version is at least 28, as illustrated in the following figure:

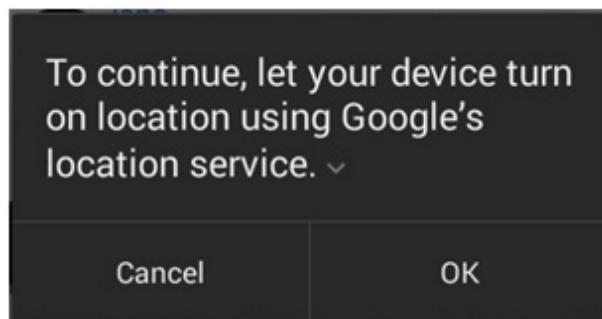


**Note:** If your SDK version is less than 28, you cannot just replace it here in **build.gradle** file with value 28. You must install at least Android SDK API level 28. To do that, in your Android Studio, click **File → Settings**, then go to: **Appearance & Behavior → System Settings → Android SDK** in the Settings dialog box. Check in the right side the **SDK Platforms** as illustrated in the following figure and install at least Android 9.0 if you don't have it.

Name	API Level	Revision	Status
Android R Preview	R	1	Not installed
Android 10.0 (Q)	29	4	Installed
Android 9.0 (Pie)	28	6	Installed
Android 8.1 (Oreo)	27	3	Installed
Android 8.0 (Oreo)	26	2	Not installed
Android 7.1.1 (Nougat)	25	3	Partially installed

## Configuring User App's Permission

33- Usually, when you use an app including Google map on your Android device and this app tries to determine your current location, the following message asking you to turn on your GPS service appears on your screen. When the GPS service is turned on, your device will connect to the GPS provider to get its own or your current location (coordinates).



You should add these configurations to your Android and iOS code to ask your app user for permission to turn on the **Location** feature on his/her device to get the app user location coordinates.

To configure this permission on Android devices you need to add the:

**ACCESS\_FINE\_LOCATION** permission to your Android Manifest file. To do so open the **AndroidManifest.xml** file which is located in the following path:  
**android → app → src → main → AndroidManifest.xml**

Then, add the following line as a direct child of the <manifest> tag:

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

The grey highlighted part of the following **AndroidManifest.xml** file displays exactly where you should add this location permission code:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.androidatc.googlemap">
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

  <application
    ...
    ...
```

34 - To configure the permission on iOS devices, open the **Info.plist** file which is located in the following path: **ios → Runner → Info.plist**, then add the following code:

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>This app needs access to location when open.</string>
<key>NSLocationAlwaysUsageDescription</key>
<string>This app needs access to location when in the background.</string>
<key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
<string>This app needs access to location when open and in the background.</string>
```

The following figure displays where you should add the previous code in **Info.plist** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>NSLocationWhenInUseUsageDescription</key>
    <string>This app needs access to location when open.</string>
    <key>NSLocationAlwaysUsageDescription</key>
    <string>This app needs access to location when in the background.</string>
    <key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
    <string>This app needs access to location when open and in the background.</string>
    <key>io.flutter.embedded_views_preview</key>
    <true/>
    <key>CFBundleDevelopmentRegion</key>
```

Click **File → Save All**

35- Open **main.dart** file. You should import the geolocator package; therefore, add the following code at the top of the main.dart file.

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

36- In the following code you will configure **getCurrentLocation()** function which has been configured to get the app user coordinates and then move the map camera to the app user coordinates depending on **setState()** method.

To run this function, you should add a Floating action button. When the app user clicks the button, he/she will be moved to his/her location.

If you have any question about the next code, review the “Moving the Camera” topic in this lesson.

The full code of **main.dart** follows:

```
import 'package:flutter/material.dart';
import 'package:google_maps_flutter/google_maps_flutter.dart';
import 'MapMarkers.dart';
import 'package:geolocator/geolocator.dart';

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

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

class _MyAppState extends State<MyApp> {
  final LatLng _location = const LatLng(43.6425701, -79.3892455);

  GoogleMapController mapController;
  void _myMapCreated(GoogleMapController controller) {
    mapController = controller;
  }

  void getCurrentLocation() async {
    final Position position = await Geolocator()
        .getCurrentPosition(desiredAccuracy: LocationAccuracy.high);

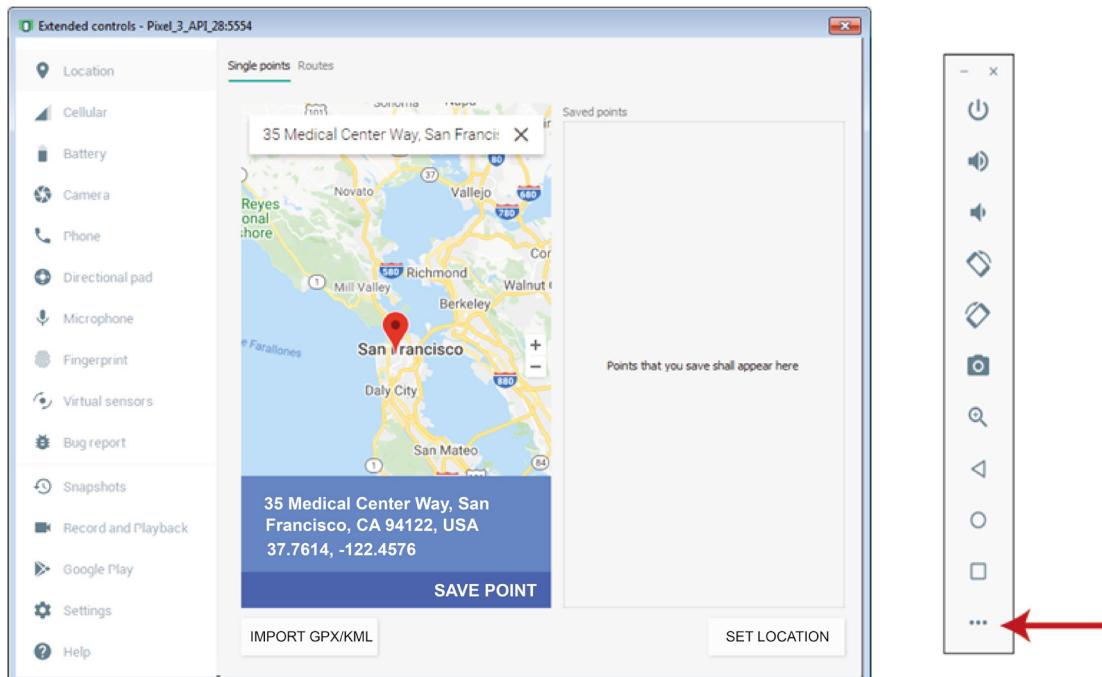
    final CameraPosition MyLocation = CameraPosition(
      target: LatLng(position.latitude, position.longitude),
      bearing: 45.0,
      tilt: 50.0,
      zoom: 11.0,
    );
  }

  setState(() async {
    final GoogleMapController controller = await mapController;
```

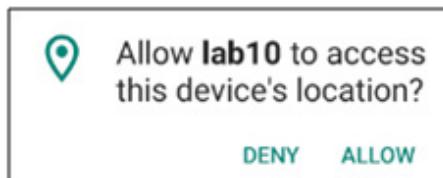
```
        controller.animateCamera(
            CameraUpdate.newCameraPosition(MyLocation),
        );
    });
}

@Override
Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
                title: Text('Maps Sample App'),
            ),
            body: GoogleMap(
                markers: {cnTowerkarker},
                onMapCreated: _myMapCreated,
                initialCameraPosition: CameraPosition(target: _location, zoom:13.0),
            ),
            floatingActionButton: FloatingActionButton.extended(
                onPressed: getCurrentLocation,
                label: Text('To My Location'),
                icon: Icon(Icons.directions_boat),
            ),
        ),
    );
}
```

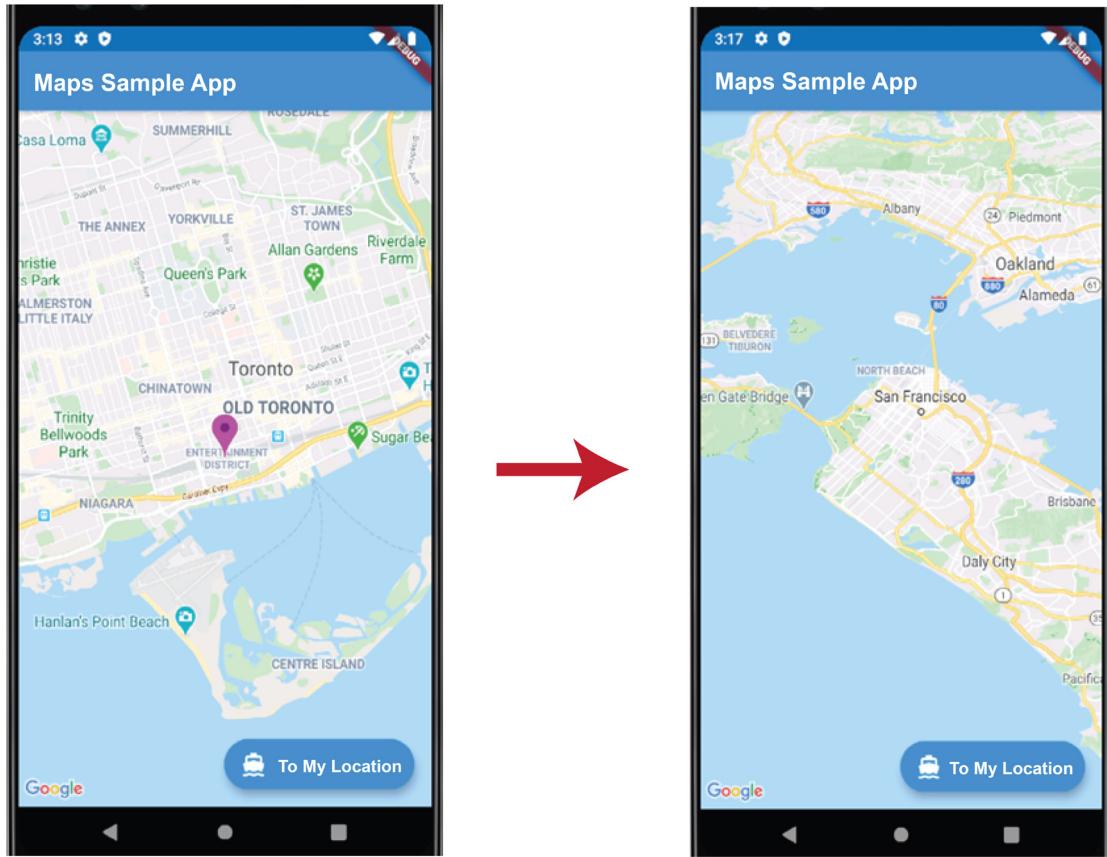
37- Before running your app, remember that you are working with a virtual device (emulator); therefore, configure your location (latitude and longitude) manually by clicking the more button of the virtual device tools bar (See illustration below), then select you location on the map (for example select your home address). In this lab, I selected San Francisco city in California and Click **SET LOCATION**.



38- Run your app. You should get the following message asking for permission to have your coordinates for the first run time only.



Click **Allow**. Then, when you get your map as illustrated in the following figures, tap the button : **To My Location**, your map camera will move to your location which you set on this emulator. In the following figure, the map camera has moved to San Francisco.



# Lesson 11: App Testing & Publishing

<b>Testing and Feedback for Your App .....</b>	11-2
Setting up a Test Environment .....	11-6
Usability Testing by Participants .....	11-7
Starting your Test Session .....	11-8
Analyzing your Test.....	11-10
<b>Publishing Flutter Apps .....</b>	11-10
<b>Publishing Android App on Google Play Store.....</b>	11-17
<b>Publishing iOS app on Apple Store.....</b>	11-32

## Testing and Feedback Your App

After completing all the steps of designing a Flutter app which satisfies users' needs and businesses with complete content and ease to access and use, we come to the important step which is testing and evaluating your application.

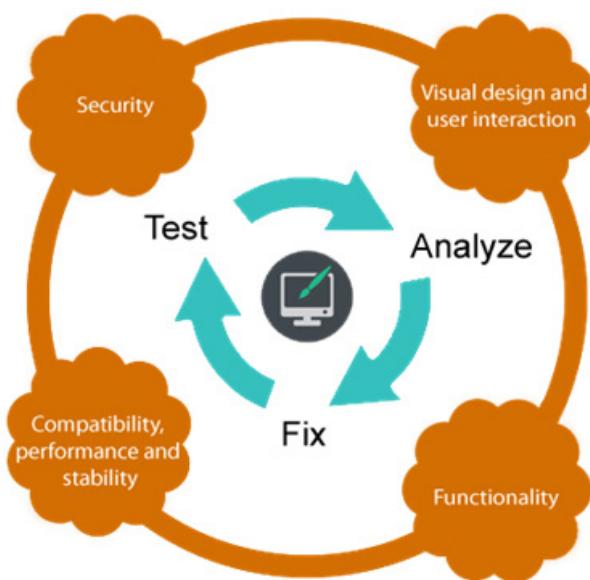
Testing the released version of your application helps to ensure that your application runs properly under realistic device and network conditions. Ideally, you should test your application on at least one handset-sized device and one tablet-sized device to verify that your user interface elements are sized correctly and that your application's performance and battery efficiency are acceptable.

Android and iOS users expect high-quality apps. App quality directly influences the long-term success of an app—in terms of installs, users' ratings and reviews, engagement, and user retention.

You should assess the core aspects of quality in your app, through a compact set of quality criteria and associated tests. All Android and iOS apps should meet these criteria.

Before publishing your apps, test them against these criteria to ensure that they function well on many devices, meet Android and iOS standards for navigation and design, and are prepared for promotional opportunities in the Google Play store and Apple App store.

Flutter apps must meet the following essential quality criteria:



## 1- Visual design and user interaction

These criteria ensure that your app provides standard visual design and interaction patterns where appropriate, for a consistent and intuitive user experience as follows:

- a- The app does not redefine the expected function of a system icon (such as the Back button).
- b- The app does not replace a system icon with a completely different icon if it triggers the standard user interface behavior.
- c- If the app provides a customized version of a standard system icon, the icon strongly resembles the system icon and triggers the standard system behavior.

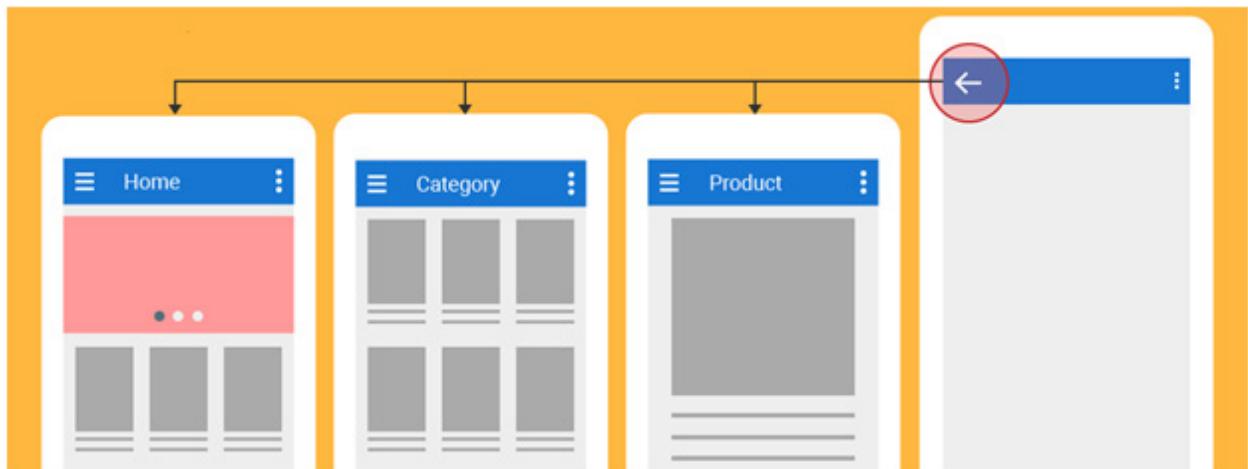


d- The app does not redefine or misuse Android or iOS UI (user interface) patterns, such that icons or behaviors could be misleading or confusing to users.

e- Consistent navigation is an essential component of the overall user experience. Nothing frustrates users more than a basic navigation that behaves inconsistently and in unexpected ways.

The Up button is used to navigate within an app based on the hierarchical relationships between screens. For instance, if screen A displays a list of items and selecting an item leads to screen B (which presents that item in more details), then screen B should offer an Up button that returns to screen A.

If a screen is the topmost one in an app (that is, the app's home), it should not present an Up button.



The system **Back** button is used to navigate, in reverse chronological order, through the history of screens the user has recently worked with. It is generally based on the temporal relationships between screens, rather than the app's hierarchy.

When the previously viewed screen is also the hierarchical parent of the current screen, pressing the **Back** button has the same result as pressing an Up button which is a common occurrence. However, unlike the Up button, which ensures that the user remains within your app, the **Back** button can take the user back to the Home screen, or even to a different app.

All workflow diagrams are dismissible using the **Back** button and pressing the **Home** button at any point navigates to the Home screen of the device.

f- The app uses notifications only to indicate a change in context related to the user personally (such as an incoming message), or exposes information/controls related to an ongoing event (such as music playback or a phone call).

## 2- Functionality

These criteria ensure that your app provides the expected functional behavior.

- a- The app requests only the absolute minimum permissions that it needs to support core functionality.
- b- The app does not seek permissions to access sensitive data (such as Contacts or the System Log) or services that can cost the user money (such as the Dialer or SMS), unless related to a core capability of the app.
- c- The app functions normally when installed on an SD card.
- d- Audio does not play when the screen is off, unless this is a core feature (for

example, the app is a music player).

- e- Audio does not play behind the lock screen, unless this is a core feature.
- f- Audio does not play on the home screen or over another app, unless this is a core feature.
- g- Audio resumes when the app returns to the foreground, or there's what indicates to the user that playback is in a paused state.
- h- The app supports both landscape and portrait orientations (if possible). Minor changes in content or views are acceptable.
- i- The app uses the whole screen in both orientations and does not letterbox to account for orientation changes.
- j- The app correctly handles rapid transitions between display orientations without rendering problems.
- k- The app should not leave any service running when the app is in the background, unless related to a core capability of the app.

*For example*, the app should not leave services running to maintain a network connection for notifications, to maintain a Bluetooth connection, or to keep the GPS powered-on.

- l- When the app resumes from the recent app switcher, the app returns the user to the exact state in which it was when last used.
- m- When the app resumes after the device wakes from a state of sleep (locked), the app returns the user to the exact state in which it was last used.
- n- When the app is re-launched from Home or All Apps, the app restores the app state as closely as possible to the previous state.
- o- When pressing the Back key, the app gives the user the option of saving any app or user state that would otherwise be lost on back-navigation.

### **3- Compatibility, Performance and Stability**

These criteria ensure that apps provide the compatibility, performance, stability, and responsiveness expected by users.

- a-The app does not crash, force close, freeze, or otherwise function abnormally on any targeted device.

- b- The app loads quickly or provides onscreen feedback to the user (a progress indicator or similar cue) if the app takes longer than two seconds to load.
- c- The app runs on the latest public version of the Android and iOS platform without crashing or losing core function.
- d- The app targets the latest SDK by setting the target SDK value in order to minimize the use of any platform-provided compatibility fallbacks.
- e- The app is built with the latest SDK by setting the compile SDK value.
- f- Music and video playback is smooth which means there are no crackle, stutter, or other artifacts, during normal app use and load.
- g- The app displays graphics, text, images, and other UI elements without noticeable distortion, blurring, or pixelation. The app provides high-quality graphics for all targeted screen sizes and form factors.
- h- The app displays text and text blocks in an acceptable manner. Make sure that there is sufficient spacing between texts and surrounding elements.

## 4- Security

These criteria ensure that apps handle user data and personal information safely:

- a- All private data is stored in the app's internal storage.
- b- All data from external storage is verified before being accessed.
- c- All intents and broadcasts follow best secure practices. Intents that contain data and payload are verified before use.
- d- No personal or sensitive user data is logged to the system or app-specific log.
- e- All network traffic is sent over SSL.
- f- All libraries, SDKs, and dependencies are up to date.
- g- JavaScript is disabled in all Web Views (unless required).
- h- The app does not dynamically load code from outside the app.

## Setting up a Test Environment

To assess the quality of your app, you need to set up a suitable hardware or emulator environment for testing.

The ideal testing environment would include a small number of actual hardware devices that represent key form factors and hardware/software combinations currently available to consumers. It's not necessary to test the app on every device that's on the market — rather, you should focus on a small number of representative devices, even using one or two devices per form factor.

If you are not able to obtain actual hardware devices for testing, you should set up emulated devices (AVDs) to represent the most common form factors and hardware/software combinations.

To go beyond basic testing, you can add more devices, more form factors, or new hardware/software combinations to your testing environment. You can also increase the number or complexity of tests and quality criteria.

## Usability Testing by Participants

Usability tests identify areas where users struggle with an app and thus help you make recommendations for improvement. The goal is to better understand how real users interact with your Flutter (Android and iOS) app and to improve the product based on the results. The primary purpose of a usability test is to improve a design. In a typical usability test, real users try to accomplish typical goals, or tasks, with a product under controlled conditions. UX designers, UI designers, and development team members watch, listen, collect data, and take notes. Since usability testing employs real customers accomplishing real tasks, it can provide objective performance data, such as time on task, error-rate, and task success. There is also no substitute for watching users struggle with or have great success in completing a task when using an app. This observation helps designers and developers gain empathy with users and encourage them to think of alternative designs that better support tasks and workflow.

You should check the usability test by participants before publishing your app on Google Play store or Apple store because your purpose is to gain users' satisfaction.

If these users start using your app and find something that does not work or is not designed properly, your app will get negative reviews level on Google Play store or Apple store. If this happens, you will then need to put a lot of effort to change this feedback and sometimes negative reviews may lead to the end of a project because other users would prefer to use other apps with positive reviews.



Your research participants must be able to represent your target group or end users; otherwise, your results will not translate into something you can use.

To recruit participants that will represent real users for your app later, you have to take into consideration many factors when making this choice.

In most projects, you need to consider the age group of your participants, their geographical location, and if there is any specific type of experience they might have that may be suitable.

## Starting Your Test Session

When you start your test session, follow the following steps:

- Welcome your participant and make them comfortable.
- Use a script to help you remember what you need to do and say.
- The participants should read the task scenarios and begin working on the scenario step by step.
- The participants should write comments, errors and completion (success or failure) on each task.
- The participants should register the time needed to complete each task and the total time to achieve the full scenario.
- The participants should be asked to give their opinion about the way they used to find items or navigation tools and if they were satisfied with the navigation method used to move from one task to the next.



You should ask participants after completing the task scenario the following questions:

- How did you find the app workflow?
- How did you find yourself with this type of app? Were you familiar with it?
- What is your opinion about the theme used?
- Did you find what you need easily?
- If this app was your app, what would you add or remove?
- Did you find a guide or any other way to tell you how to use this app?
- Are you satisfied to use this app for the purpose for which it is created?
- Do you still remember the purpose of the app? This question is asked to make sure that participants are serious in their testing tasks.

## Analyzing Your Test

Analyzing is a three-step process:

1. Identify exactly what you observed
2. Identify the causes of any problems
3. Determine Solutions

If you don't want or don't have enough time for this test step, you may delegate the testing step to a company that provides this service. This way, you can get a detailed report about your app and how you may improve the performance and suggest solutions for problems if there are any. Google Firebase provides a test lab service .

## Firebase Test Lab

Firebase Test Lab is a cloud-based app-testing infrastructure. With one operation, you can test your Android or iOS app across a wide variety of devices and device configurations as well as see the results—including logs, videos, and screenshots—in the Firebase console.

For more information about the Firebase test lab, check the following link: <https://firebase.google.com/docs/test-lab>

## Publishing Flutter Apps

When you finish the design process of your Flutter app, and complete the test and fix steps, the last step is to publish the Android part of your Flutter app in Google play store as well as publish the iOS part of your Flutter app in Apple store.

The App Store is a digital distribution platform, developed and maintained by Apple Inc., for mobile apps on its iOS operating system. The store allows users to browse and download iOS apps.

Google Play, formerly Android market, is a digital distribution service operated and developed by Google. It serves as the official app store for the Android operating system, allowing users to browse and download applications developed with the Android software development kit and

published through Google.

The following steps display how you should prepare your Flutter app to be ready to be published in Google play store and Apple store:

### Step A: Creating your app icons

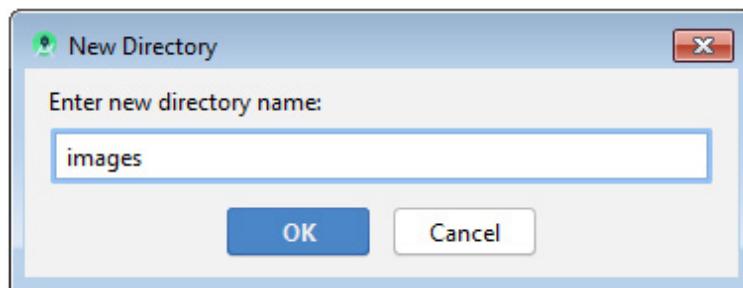
This topic is already explained step by step in lesson 5. However, in this lesson, you will create app icons for your Android and iOS apps in an easier way as follows:

1- Create your app logo image, and be sure that it has a PNG extension and a transparent background. To create your PNG image, you may use a software such as Adobe Photoshop to remove the image background and save the image as a PNG image. You may also use Google search engine to find a safe web site to create a free logo image. In this lesson, you may use **logo.png** image which is in your:

### **Lab Source Files → images**

2- Go to: [https://pub.dev/packages/flutter\\_launcher\\_icons](https://pub.dev/packages/flutter_launcher_icons) web site. Follow the steps in this web site as shown in next steps. Then you will add a package to your Android Studio to create icons for your Android and iOS apps using your logo image (logo.png).

3- In your project structure console, right click your project name → **New** → **Directory** as illustrated in the following figure:

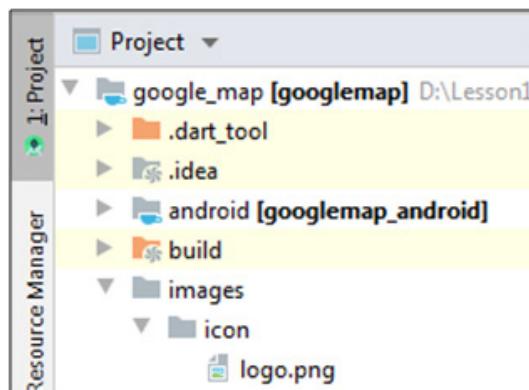


Then, create a sub-folder in this **images** folder and name it **icon**

4- From your local hard disk, open : **Lab Source Files → Images** , then copy : **logo.png** image. In your Android Studio, paste this logo image in the following folder: **images → icon**,

then click **OK**.

You should get the following file structure:



5- In your Android Studio, open: **pubspec.yaml** file, then add the following code under **dev\_dependencies**:

```
flutter_launcher_icons: "^0.7.3"
```

**Note :** **dev\_dependencies** are not related to app code; otherwise, they are related to the Flutter development tools.

6- Add the following in **pubspec.yaml** file as a separate code.

```
flutter_icons:  
  android: true  
  ios: true  
  image_path: "images/icon/logo.png"
```

You should have the following structure in your **pubspec.yaml** file:

```
dev_dependencies:  
  flutter_test:  
    sdk: flutter  
  flutter_launcher_icons: "^0.7.3"  
  
flutter_icons:  
  android: true  
  ios: true  
  image_path: "images/icon/logo.png"
```

Click **Packages Get** at the top of the **pubspec.yaml** file.

7- In your Android Studio, open the **Terminal** console (on the status bar), then type the following command:

```
flutter pub get
```

Then, press the **Enter** key (or Return for Mac) and type the following command:

```
flutter pub run flutter_launcher_icons:main
```

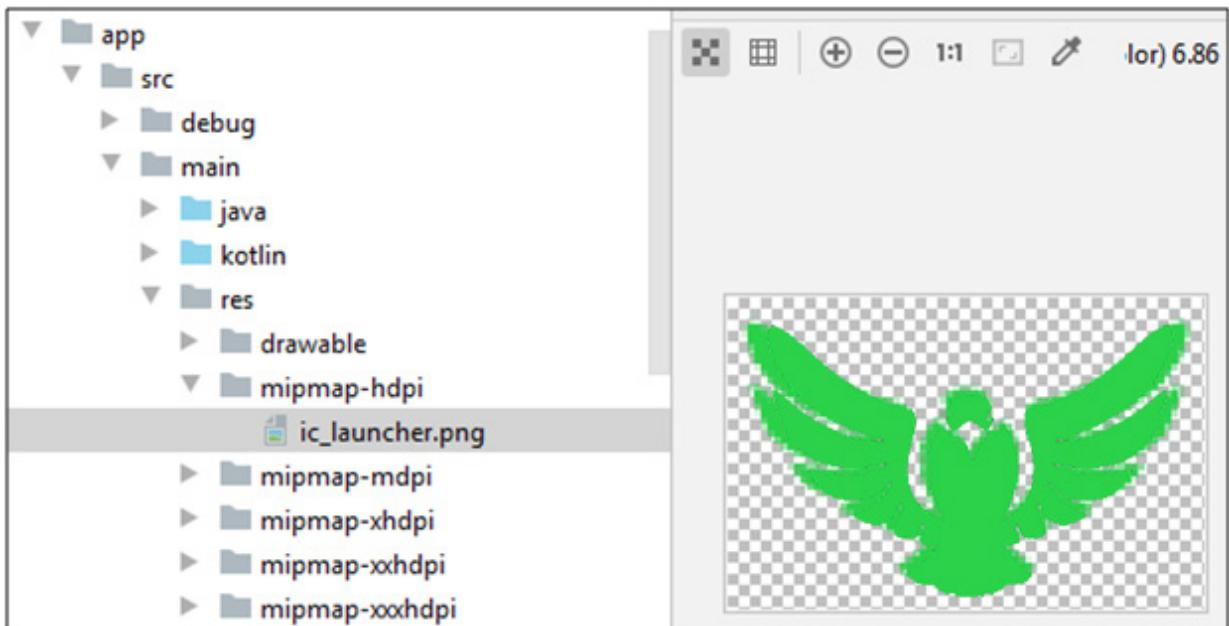
Then, press the **Enter** key. You should get the following message:

```
Android minSdkVersion = 16
Creating default icons Android
Overwriting the default Android launcher icon with a new icon
Overwriting default iOS launcher icon with new icon
```

Now, check your Android app icons in the following path:

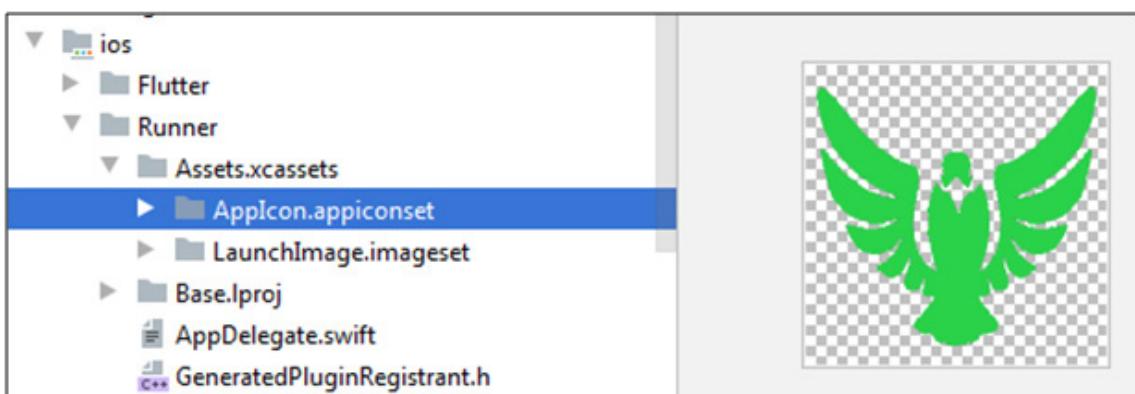
**android → app → src → main → res**

Then, check all the **mipmap** folders which include your Android app icon in different sizes because your app may run on different screen sizes as illustrated in the following figure:

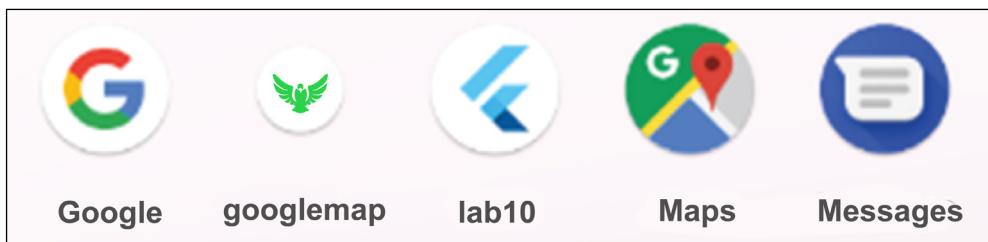


To check your app icons for iOS, check the following path:

ios → Runner → Assets.xcassets → AppIcon.appiconset as illustrated in the following figure:



Stop your app and then run it again. Press your phone emulator home button, then check your apps list. You should get the icon which you added in the previous step as illustrated in the following figure:



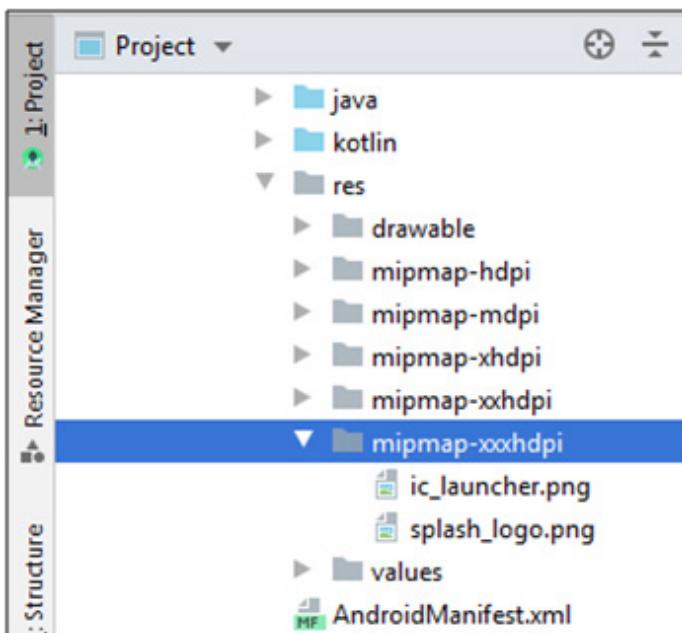
## Step B: Adding Splash Screen

Each Flutter experience in an app requires a few moments to initialize the Dart isolate that runs the code. This means a user momentarily sees a blank screen until Flutter renders its first interface. Flutter supports an improved user experience by displaying an Android View as a splash screen while Flutter initializes.

To add a splash image to your Android app, perform the following steps:

- 1- From your local hard disk, open : **Lab Source Files** → **Images**, then **copy** the **splash\_logo.png** image ( remember, the image name must include an underscore character as part of image file name).
- 2- Open the following path: **android** → **app** → **src** → **main** → **res**

Then, paste this image inside all the **mipmap** folders (five folders) as illustrated in the following figure:



- 3- Open **android** → **app** → **src** → **main** → **res** → **drawable** → **launch\_background.xml**

Remove the comment “`<!--`” symbols and add the image which you added to the **mipmap** folders without file extension as illustrated in the grey part of the following code.:.

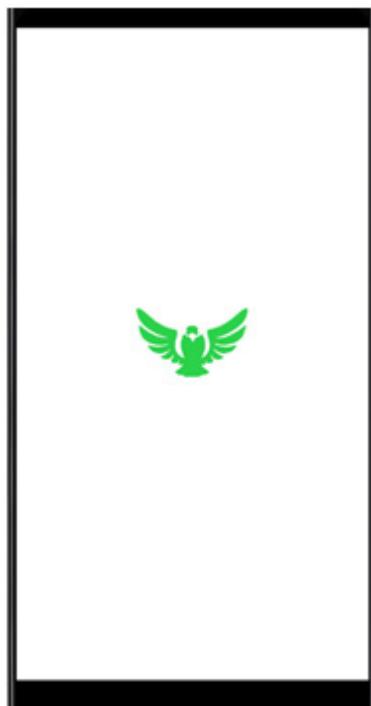
```
<?xml version="1.0" encoding="utf-8"?>
<!-- Modify this file to customize your launch splash screen -->
<layer-list xmlns:android="http://schemas.android.com/apk/res/
    android">
```

```
<item android:drawable="@android:color/white" />

<!-- You can insert your own image assets here -->
<item>
    <bitmap
        android:gravity="center"
        android:src="@mipmap/splash_logo" />
</item>
</layer-list>
```

Now, Click **File → Save ALL**

4- Stop your app and run it again. You should get your splash image within the startup process of your app as illustrated in the following figure:



To add a splash image to your iOS app, perform the following steps:

1- In your Android Studio, go to:

**ios → Runner → Assets.xcassets → LaunchImage.imageset**

You will find three images in this folder (LaunchImage.png , LaunchImage@2x.png , and LaunchImage@3x.png) .

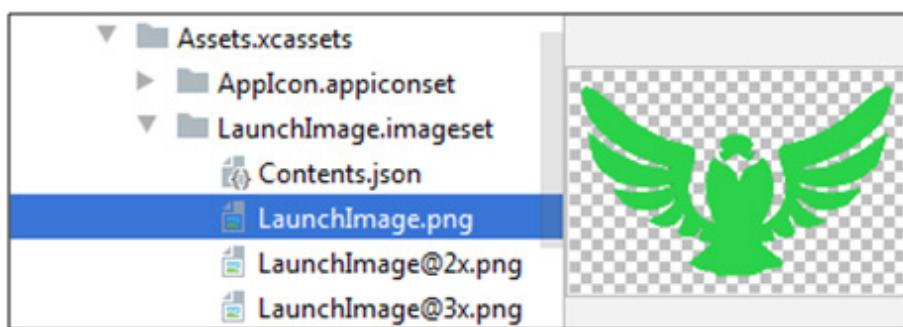
2- Create three copies of the image which you want to make as the splash or startup image for your iOS app, and name these images to have the same names (LaunchImage.png , LaunchImage@2x.png , and LaunchImage@3x.png) .

3- Now, delete the three images which exist in the **LaunchImage.imageset**

4- Copy your three images. Then, paste them in the **LaunchImage.imageset** folder.

Open your lab source files folder on your computer, then open the following path:

**Lab Source Files → Images → iOS Lunch Images**, you will find these three images ready to move to your **LaunchImage.imageset** folder in Android Studio as illustrated in the following figure:



Now, you may test your app on your iPhone emulator. Stop your app and run it again. You should get this image within the first launch of your app.

The next step in publishing Android and iOS app is for Android apps only.

## Publishing Android App on Google Play Store

After you complete testing, analyzing and fixing your app, it is the time to publish it on Google Play Store. You need to give your Android app a digital signature.

This app signature approves that you are the owner of this app, and only you have the permission to publish and update it on Google Play store. Use the following instructions to sign your app:

**Note:** All the commands in the next steps are available in the web link below:  
<https://flutter.dev/docs/deployment/android>

To avoid any typing errors or if you want to save time, you may copy and paste these commands from this web site. However, at the same time follow the instructions below of this book.

1- In your Android Studio IDE, in the Terminal console, type either of the following lines of code, as applies to your operating system:

On Mac/Linux, use the following command:

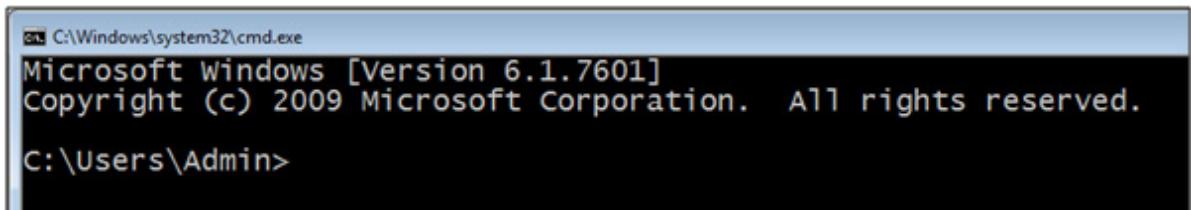
```
keytool -genkey -v -keystore ~/key.jks -keyalg RSA -keysize 2048  
-validity 10000 -alias key
```

On Windows, use the following command

```
keytool -genkey -v -keystore c:/Users/USER_NAME/key.jks -storetype  
JKS -keyalg RSA -keysize 2048 -validity 10000 -alias key
```

Then, press **Enter** or **Return** key.

For Windows, you should replace **USER\_NAME** with your computer admin user name. To know what is your computer user name, just hold the Windows logo key and then press **R** letter key, type **cmd** and press **Enter**. As you see in the snapshot below, my computer user name is Admin.



```
keytool -genkey -v -keystore c:/Users/admin/key.jks -storetype JKS  
-keyalg RSA -keysize 2048 -validity 10000 -alias key
```

Also, you should use the command prompt to run this *keytool* command. This command should run on the same directory which includes this *keytool.exe* file. To know where this tool exists, go to:

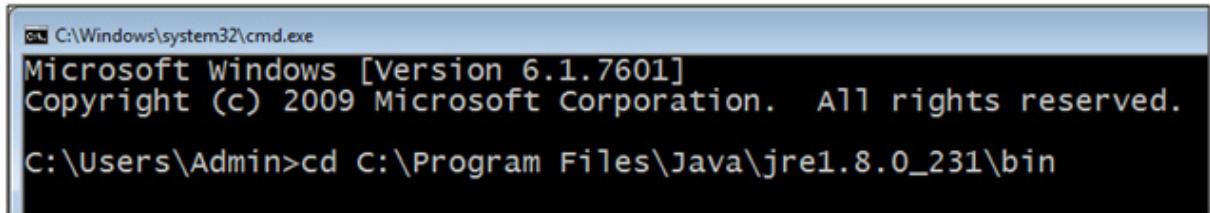
C:\ → Program Files → Java → jre1.xxx → bin (xxx depending on your version number).

Make sure that this **bin** folder includes the **keytool.exe** file, then perform the

following steps:

1- Open command prompt.

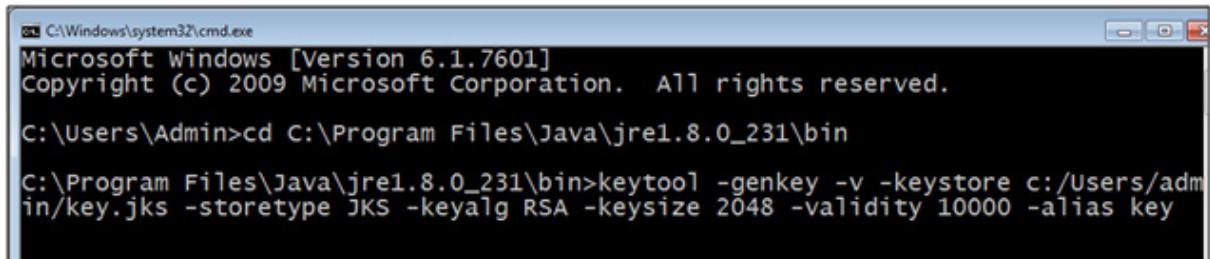
2- Use **cd** command (**C**hange **D**irectory) to move to the directory of the tool. Then, add the path of this bin folder as you see in the following figure:



```
cmd C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd C:\Program Files\Java\jre1.8.0_231\bin
```

3- Type the **keytool** command using your user name as illustrated in the following figure:

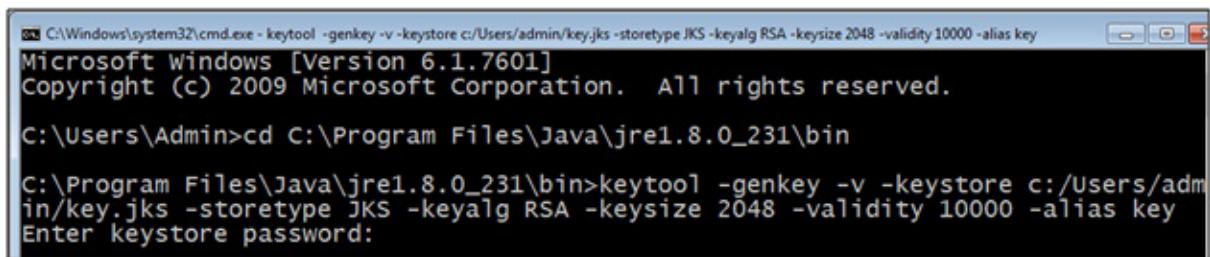


```
cmd C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd C:\Program Files\Java\jre1.8.0_231\bin

C:\Program Files\Java\jre1.8.0_231\bin>keytool -genkey -v -keystore c:/Users/admin/key.jks -storetype JKS -keyalg RSA -keysize 2048 -validity 10000 -alias key
```

4- You will be asked to enter and confirm the password for keystore as illustrated in the following figure:



```
cmd C:\Windows\system32\cmd.exe - keytool -genkey -v -keystore c:/Users/admin/key.jks -storetype JKS -keyalg RSA -keysize 2048 -validity 10000 -alias key
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd C:\Program Files\Java\jre1.8.0_231\bin

C:\Program Files\Java\jre1.8.0_231\bin>keytool -genkey -v -keystore c:/Users/admin/key.jks -storetype JKS -keyalg RSA -keysize 2048 -validity 10000 -alias key
Enter keystore password:
```

5- Now, answer the following questions such as your first and last name, organization name, organization unit (the folder name which will include your profile data), city, state, and the country (first two letters only), type Yes if all the information is correct, and press **Enter**. Then, you will be asked to enter the store password as illustrated in the following figure:

```
C:\Windows\system32\cmd.exe - keystore -genkey -v -keystore c:/Users/admin/key.jks -storetype JKS -keyalg RSA -keysize 2048 -validity 10000 -alias key
in/key.jks -storetype JKS -keyalg RSA -keysize 2048 -validity 10000 -alias key
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: Android ATC
What is the name of your organizational unit?
[Unknown]: Lesson_10
What is the name of your organization?
[Unknown]: Android ATC
What is the name of your City or Locality?
[Unknown]: Toronto
What is the name of your State or Province?
[Unknown]: Ontario
What is the two-letter country code for this unit?
[Unknown]: CA
Is CN=Android ATC, OU=Lesson_10, O=Android ATC, L=Toronto, ST=Ontario, C=CA correct?
[no]: yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 10,000 days
for: CN=Android ATC, OU=Lesson_10, O=Android ATC, L=Toronto, ST=Ontario, C=CA
Enter key password for <key>
(RTURN if same as keystore password):
```

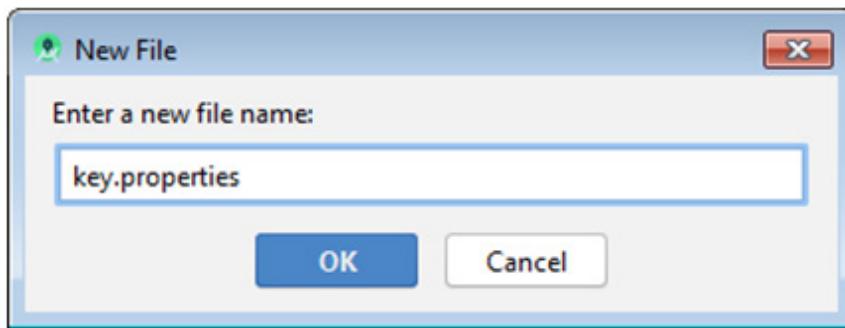
Now, your signature file (**key.jks**) has been created and saved on your hard disk as illustrated in the following figure:



The next step is: Reference the **keystore** from the app. To do this, perform the following steps:

1- In your Android Studio, right click the **android** folder, then select → **New** → **File**

Type: **key.properties** for the file name, then click **OK** as illustrated in the following figure:



Remember : In my previous configurations, my app signature file (**key.jks**) was stored in the following location **c:/Users/admin/key.jks** as illustrated in the following figure:

```
Generating 2,048 bit RSA key pair and self-signed certificate  
with a validity of 10,000 days  
for: CN=Android ATC, OU=Lesson_10, O=Android ATC,  
C=CA  
Enter key password for <key>  
(RETURN if same as keystore password):  
Re-enter new password:  
[Storing c:/Users/admin/key.jks]
```

2- In Android Studio, open this file: **key.properties** , then type the following configurations:

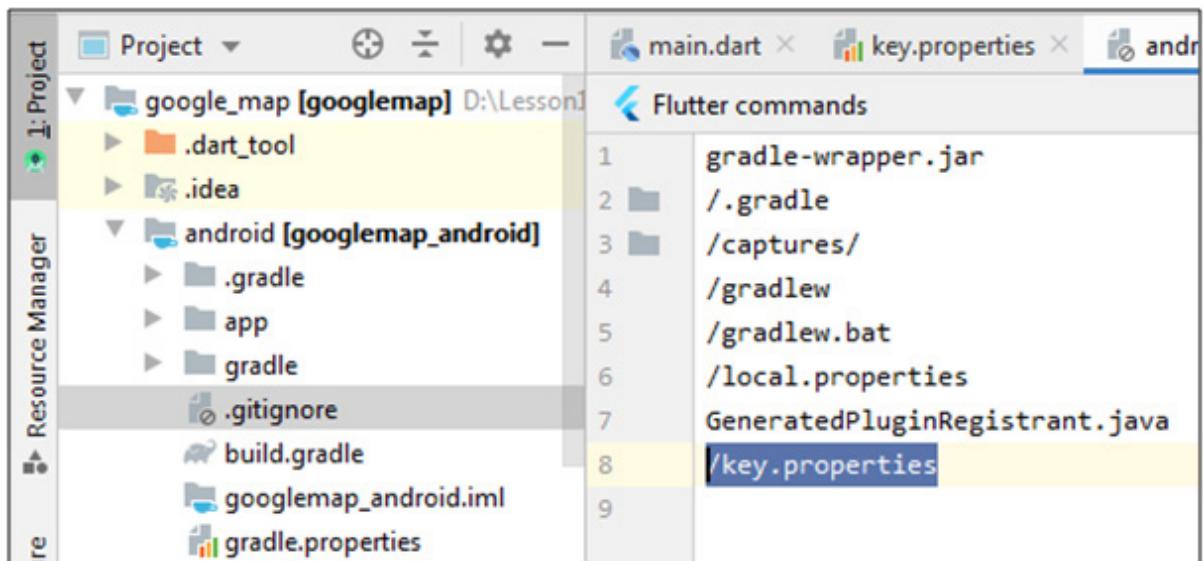
```
storePassword= Type Your Store Password here  
keyPassword= Type Your Key Password here  
keyAlias=key  
storeFile= c:/Users/admin/key.jks
```

Remember to keep the **key.properties** file private; do not check it into public source control.

In some cases, you share your app files with others using Github web site, it is important to exclude this file. To do that, open the following file in **Android Studio**:

**android → gradle → .gitignore**

Then, add this file name to the list as illustrated in the following figure:



3- Open **android → app → build.gradle** file and make the following two changes in this file:

a- To Load the **key.properties** file into the **keystoreProperties** object, add only the following grey highlighted code before the android block code :

```
def keystoreProperties = new Properties()
def keystorePropertiesFile = rootProject.file('key.properties')
if (keystorePropertiesFile.exists()) {
    keystoreProperties.load(new
        FileInputStream(keystorePropertiesFile))
}

android {
    compileSdkVersion 28
    .....
    .....
```

b- To configure the **signingConfigs** block in your module's **build.gradle** file. Replace the following code:

```
buildTypes {
    release {
        // TODO: Add your own signing config for the release build. //
        // Signing with the debug keys for now, so 'flutter run --release' works.
        signingConfig signingConfigs.debug
    }
}
```

With the following code:

```
signingConfigs {
    release {
        keyAlias keystoreProperties['keyAlias']
        keyPassword keystoreProperties['keyPassword']
        storeFile keystoreProperties['storeFile'] ?
```

```
file(keystoreProperties['storeFile']) : null  
    storePassword keystoreProperties['storePassword']  
}  
}  
buildTypes {  
    release {  
        signingConfig signingConfigs.release  
    }  
}
```

Now, save your app configuration ( **File → Save ALL** )

## Reviewing the App Manifest file

Review the default App **Manifest** file, **AndroidManifest.xml**, located in

<app dir> → **android** → **app** → **src** → **main** → **AndroidManifest.xml** and if your app needs internet connection, add only the following grey highlighted code to your **AndroidManifest.xml** to give your app the permission to access the internet:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.androidatc.googlemap">  
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>  
    <uses-permission android:name="android.permission.INTERNET"/>  
...  
...
```

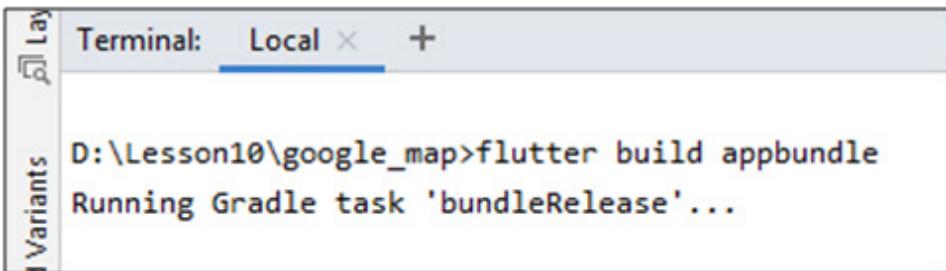
## Building the App bundle for Release

This section describes how to build a release app bundle. By default, the app bundle contains your Dart code and the Flutter runtime compiled. To create your Android app bundle, perform the following steps:

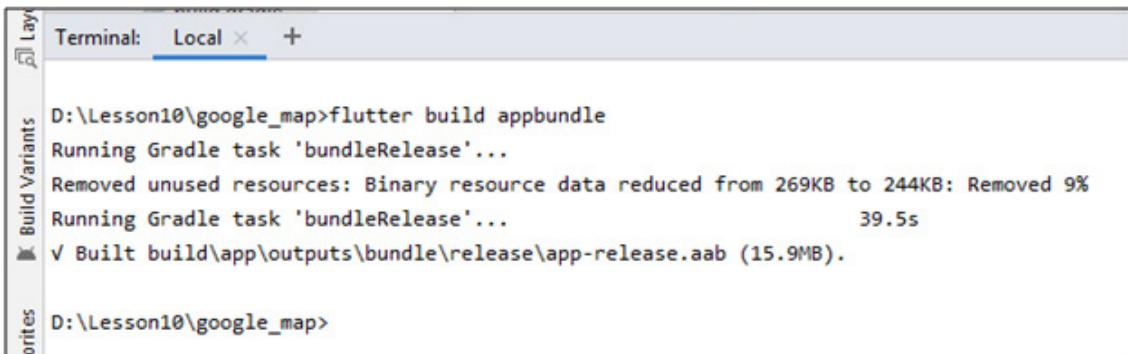
1- In Android Studio, open the **Terminal** console.

2- Type the following command: **flutter build appbundle** and press Enter.

You should wait for seconds as illustrated in the following figure:



You should get the following result:



The previous figure displays that my bundle file has been created and it is found in the following path on my computer. **D:\Lesson10\google\_map\build\app\outputs\bundle\release**

This file will be uploaded on the Google Play store.



Before you go to the Google Play store, there is an important part of your Flutter app that needs explanation. Open your **pubspec.yaml** file. At the beginning of this file, you will find the following configuration:

**version: 1.0.0+1**

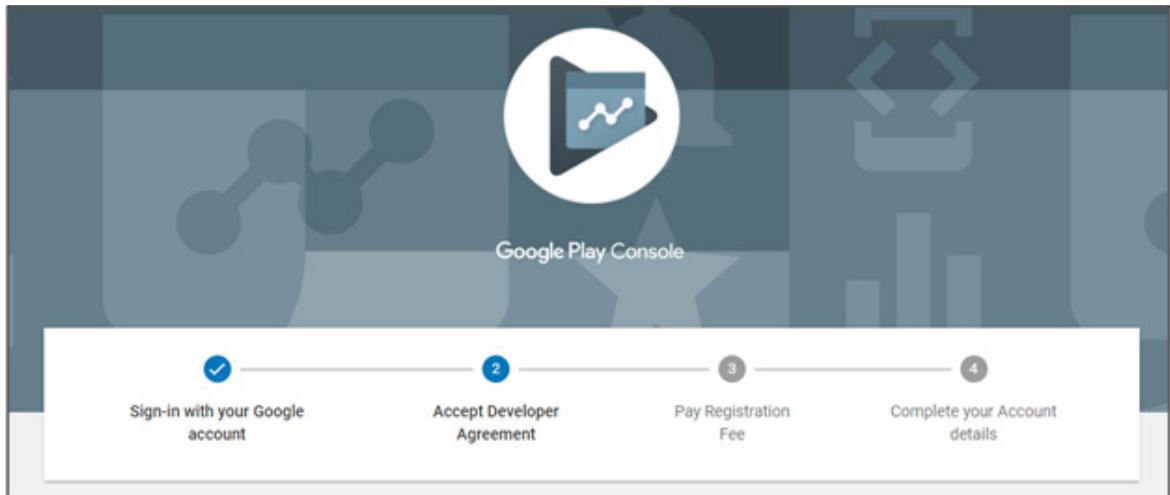
This number represents your Flutter app version. When you want to publish a new version of your app including your app updates, you must increment this number, then upload the new version to our Google Play store. Then, your app users will receive a notification displaying a new update for your app.

## Publish App to Google Play Store

To publish an Android app on Google Play store, you should create a publisher account which allows you to access the tools in Google Play Developer Console and eventually publish your Android applications. The following steps tell you how to register for a Google Play publisher account:

1- Go to: <https://play.google.com/apps/publish/signup>

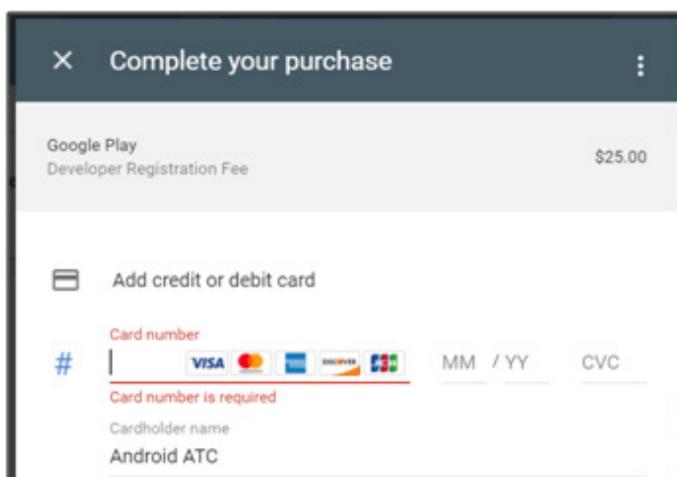
Login using your Gmail account, then you will get the following figure:



2- Scroll down this web page, in the **Accept developer agreement** section, select **I agree**, then click : **CONTINUE TO PAYMENT** button.

3- You will get the payment page. The Developer Registration Fee is 25 USD for a life time.

If you want to continue the publishing your app, you will be required to enter your credit card information. The following screenshots will give you enough knowledge about how to publish an Android app on Google Play store without the need to pay at this time being.



4-After completing the payment step, you will be asked to complete your account details. Here, you will fill out a form that verifies you about your name, email address, web site, and phone number. After filling out this data, click the **COMPLETE REGISTRATION** button as illustrated in the following figure:

The screenshot shows a registration completion screen. At the top, a horizontal progress bar has four steps: 'Sign-in with your Google account' (checkmark), 'Accept Developer Agreement' (checkmark), 'Pay Registration Fee' (checkmark), and 'Complete your Account details' (step 4). Below the progress bar, a message says 'You are almost done...'. A note below it states: 'Just complete the following details. You can change this information later in your account settings if you need to.' The 'Developer Profile' section contains the following fields:

- Developer name \***: AndroidATC (10/50 characters)
- Email address \***: info@androidatc.com
- Website**: http://www.androidatc.com/

A note next to the developer name field says: 'The developer name is displayed to users under your application name. Changes to the developer name will be reviewed by Google and can take up to 7 days.'

5- To add your app to Google Play store, click : **CREATE APPLICATION** button as illustrated in the following figure:

The screenshot shows the 'All applications' page in the Google Play developer console. The table lists one application:

App name	Active installs	Google Play rating	Last update	Status
Android ATC - My First App com.androidatc.androidatc_myfirs...	-	★ -	Mar 31, 2018	Removed

A blue 'CREATE APPLICATION' button is located at the top right of the table area. The bottom right corner of the table shows 'Page 1 of 1'.

6- Then, type your app name. For example, this app is to test publishing Android app only, type:

**Android-app-publishing-test** for the app name, then click **CREATE** as illustrated in the following figure:

Create application

Default language \*

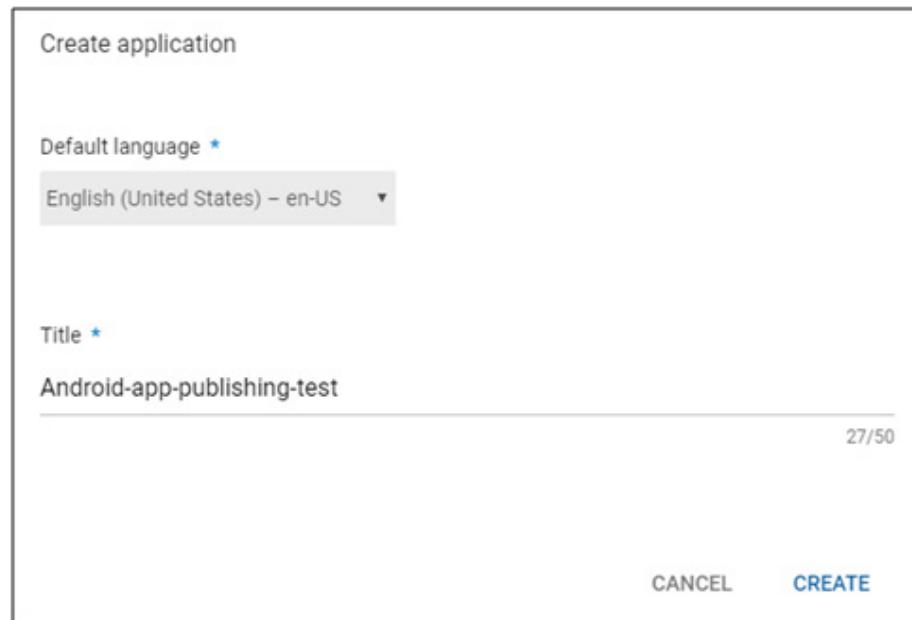
English (United States) – en-US ▾

Title \*

Android-app-publishing-test

27/50

CANCEL CREATE



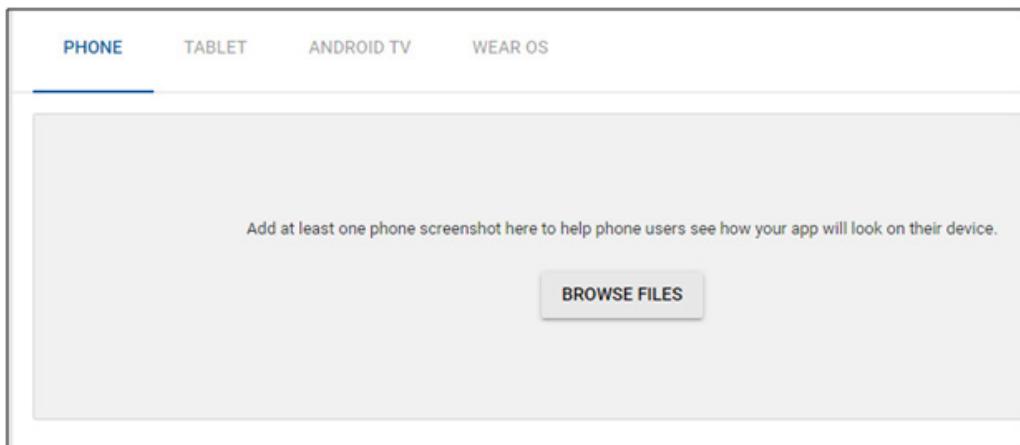
7- You will get a form asking you to fill out some information about your app as follows :

**Short description** (max: 80 characters):

**Full description** (max: 4000 characters):

The above details are important because they help app users to have an idea about the purpose of creating this app, and how to use it.

8- Scroll down until reaching the following part which is related to uploading screenshots to your app as illustrated in the following figure. Add some screenshots to your app. These images must be carefully selected because this will create the first impression for your app, and if these are clear and organized to give the app user a good idea about your app, the user will most likely download your app, otherwise he/she may select another competitor app.



Complete all the required information in this page such as email, web site, telephone number, and so on.

9- On the left side of your Google Play Console, click **Pricing & distribution**. Click **Free** to make your app free for download . If it is Paid, you should configure the price template and other configurations.

Also, under Countries section, click **Available** to make your app available to download across all countries as illustrated in the following figure:

The screenshot shows the 'Pricing & distribution' section of the Google Play Console. On the left, there's a sidebar with various options like 'Android Instant Apps', 'Artifact library', etc. The main area is titled 'Pricing & distribution' and shows a table of country availability. The table has columns for 'Status' (radio buttons for 'Unavailable' or 'Available'), and a 'Carrier options' column. Most countries listed have the 'Available' status selected. A 'SAVE DRAFT' button is at the bottom right.

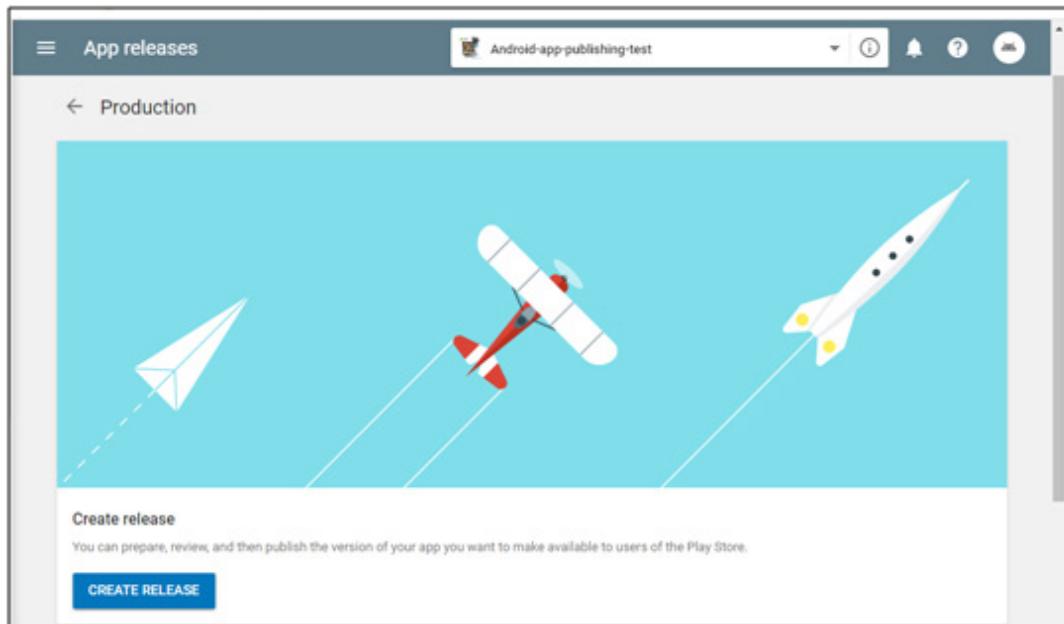
	Status	Carrier options
Argentina	Available (Production, Beta, and Alpha)	
Armenia	Available (Production, Beta, and Alpha)	
Aruba	Available (Production, Beta, and Alpha)	
Australia	Available (Production, Beta, and Alpha)	
Austria	Available (Production, Beta, and Alpha)	
Azerbaijan	Available (Production, Beta, and Alpha)	
Bahamas	Available (Production, Beta, and Alpha)	
Bahrain	Available (Production, Beta, and Alpha)	

Scroll down and answer the following question: **Does your application have ads?**  
Select **No, it has no ads.**

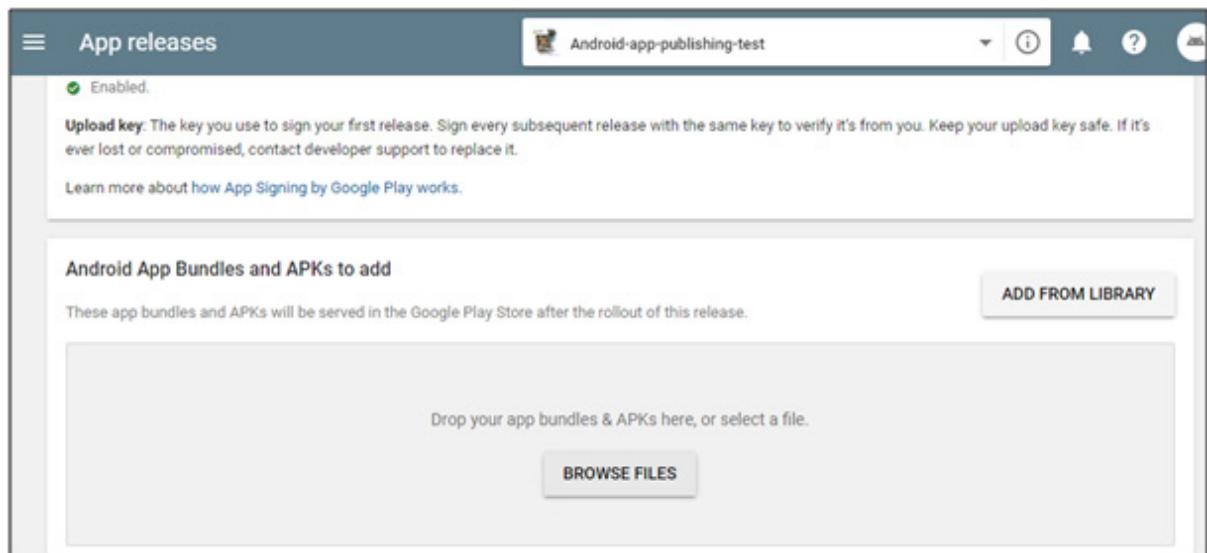
On "Actions on Google" select: → accept the Actions on Google

Then, click **SAVE DRAFT**

10- On the left side of your Google Play Console, click **App releases**, then on the right side under **Production**, click **MANAGE** , then you will get the following figure:



Click **CREATE RELEASE** button and click **Continue** to get the following figure. Click **BROWSE FILES** button to upload your app bundle which you have created in the previous steps. My file is in the following path : D:\Lesson10\google\_map\build\app\outputs\bundle\release. Select your file, and wait till it completes uploading.

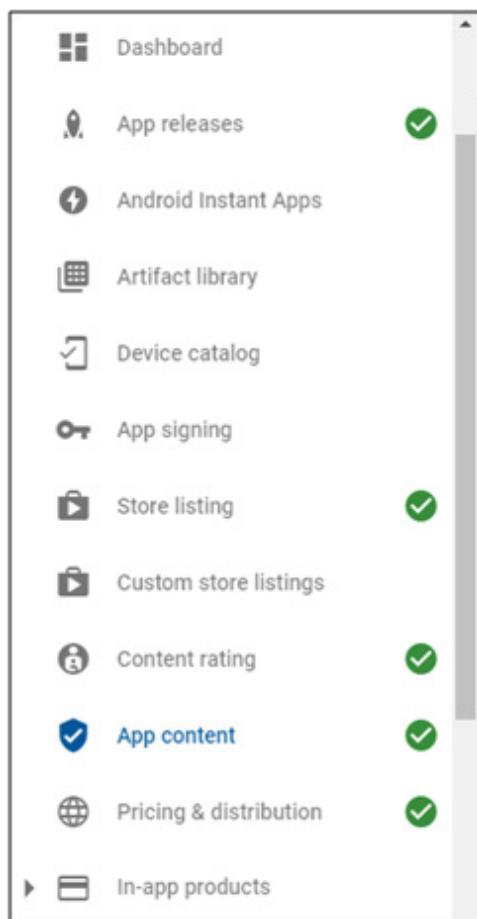


After uploading your app bundle, you will get the following figure which displays that your app has been uploaded successfully.

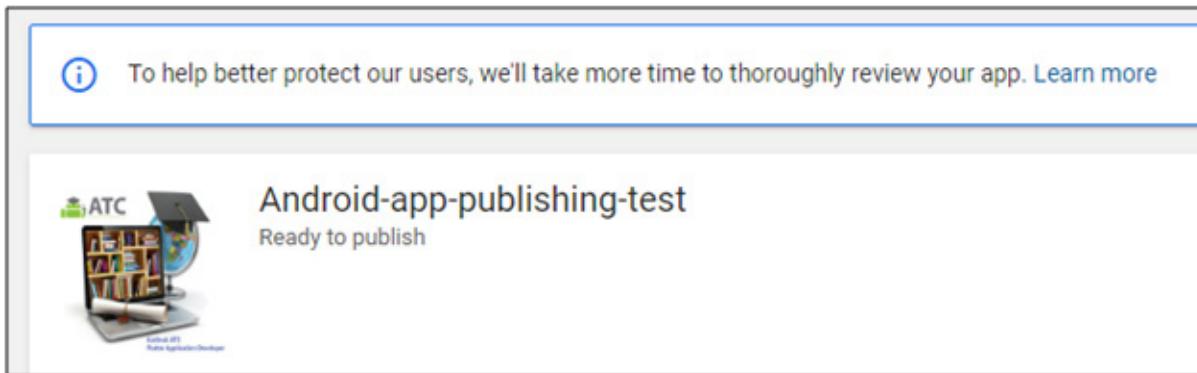
Type	Version code	App download size <small>(?)</small>	
Android App Bundle	1	5.65 to 6.18 MB	<a href="#">REMOVE</a>

Scroll down and click the **SAVE** button.

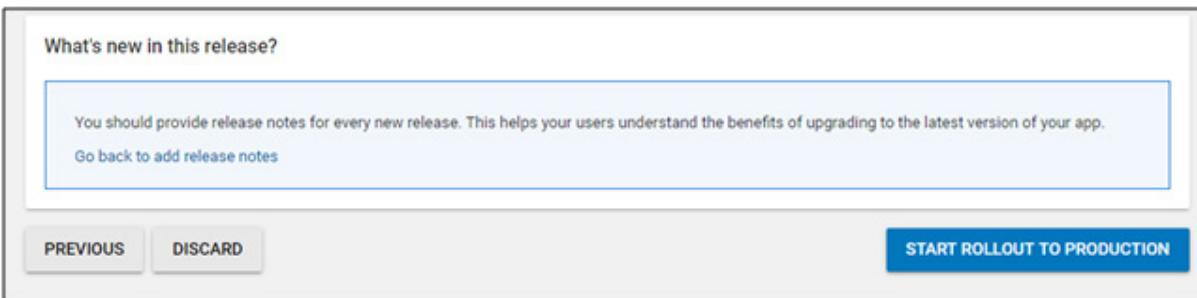
11- After you complete filling all the required information about your app, you should get green marks on the right side of your Google Play Console as illustrated in the following figure:



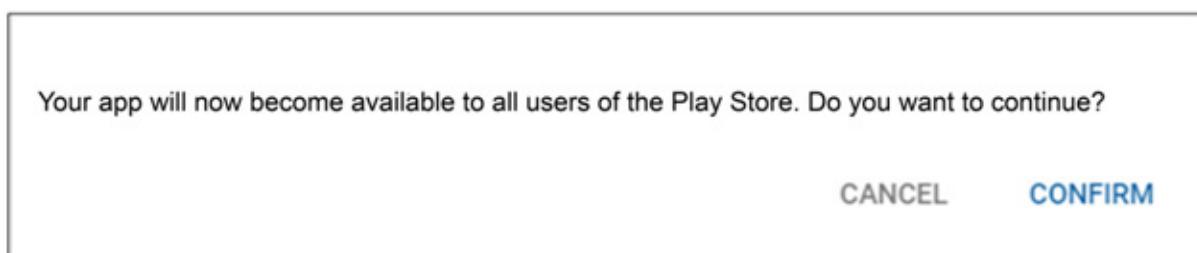
If you click the Dashboard, you will find the following message from Google Play store as illustrated in the following figure:



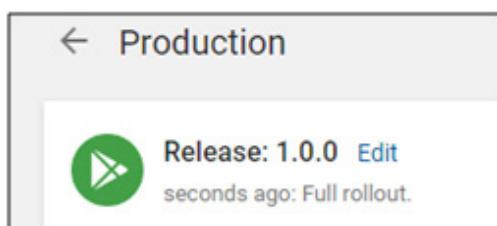
It will take up to 7 days or longer in exceptional cases for your app to be thoroughly reviewed before it is finally published. In my case, I waited for 10 minutes, then I clicked **App release** in the Google Play Console, then as illustrated in the following figure, I clicked **START ROLLOUT TO PRODUCTION**



I got the following message, then clicked **CONFIRM**.

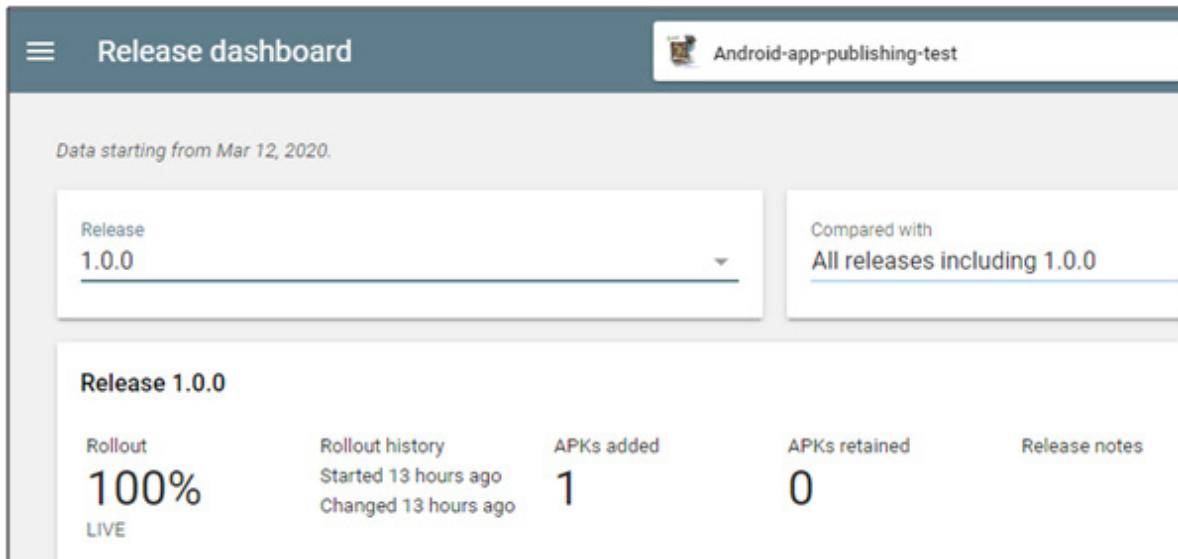


After a few seconds, I got the following figure:



Finally, this app became available for all users on Google Play store.

The following figure displays the release dashboard after publishing your Android app.



## Publishing iOS app on Apple Store

The Apple Developer website provides the tools and information you need to make great apps for Apple platforms. If you are new to development on Apple platforms, you can get started for free and create a free Apple ID by performing the following steps:

1-To create your Apple ID go to: <https://appleid.apple.com/account?appID>

2- Fill out the application form, then click **Continue**

3- Verify your email address.

*Remember, your email address is your Apple ID.*

To publish your iOS app on Apple store, you must enroll in the Apple Developer Program.

The cost of membership is **99 USD** per year. To enroll in this program, check the following web site: <https://developer.apple.com/programs/enroll>

Now, assume that your Apple account is your Apple developer account and you want to register your iOS app on App Store Connect. App Store Connect is where you will manage your app's life cycle. You will define your app name and description, add screenshots, set pricing, and manage releases to the App Store .

Registering your app involves two steps: registering a unique Bundle ID and creating an application record on App Store Connect.

**Note:** all the web links for creating Apple ID and registering a bundle id are available in the following web link: <https://flutter.dev/docs/deployment/ios>

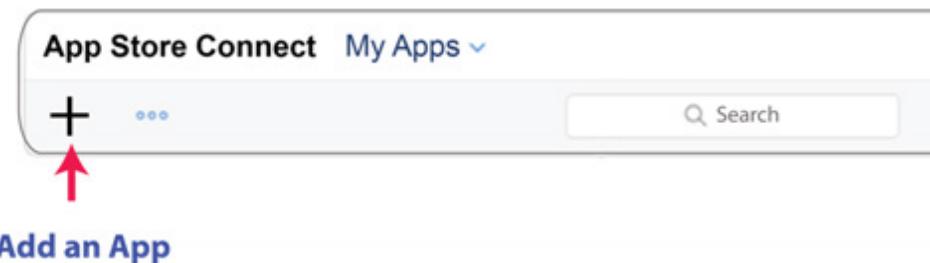
## Register a Bundle ID:

Every iOS application is associated with a Bundle ID, a unique identifier registered with Apple.

To register a Bundle ID for your app, follow these steps:

Note: You should use a Mac device to complete the following steps:

1. Login to : <https://appleid.apple.com/signin> using your Apple developer account.
2. Go to **App Store Connect** web site: <https://appstoreconnect.apple.com>
3. Click + to create a new Bundle ID as illustrated in the following figure:



4. Enter an app name, select **Explicit App ID** and enter an ID.
5. Select the services your app will use, then click **Continue**.
6. On the next page, confirm the details and click **Register** to register your Bundle ID.

After completing registering your iOS app and getting its bundle ID, the next step is registering your app on App Store Connect.

## Create an Application Record on App Store Connect

1. Open App Store Connect in your browser.
2. On the App Store Connect landing page, click **My Apps**.
3. Click + in the top-left corner of the My Apps page, then select **New App**.
4. Fill in your app details in the form that appears. In the Platforms section, ensure that iOS is checked. Since Flutter does not currently support tvOS, leave that checkbox unchecked. Click **Create**.
5. Navigate to the application details for your app and select **App Information** from the sidebar.

6. In the General Information section, select the **Bundle ID** you registered in the preceding step.

## Review Xcode project settings

In this step, you'll review the most important settings in the Xcode workspace.

Navigate to your target's settings in Xcode:

1. In Xcode, open **Runner.xcworkspace** in your app's **ios** folder.
2. To view your app's settings, select the Runner project in the Xcode project navigator. Then, in the main view sidebar, select the Runner target.
3. Select the General tab.

Next, you'll verify the most important settings.

**In the Identity section:**

**Display Name:**

The name of the app to be displayed on the home screen and elsewhere.

**Bundle Identifier:**

The App ID you registered on App Store Connect.

**In the Signing section:**

**Automatically manage signing:**

Xcode should automatically manage app signing and provisioning. This is set true by default, and it should be sufficient for most apps. For more complex scenarios, see the [Code Signing Guide](#).

**Team:**

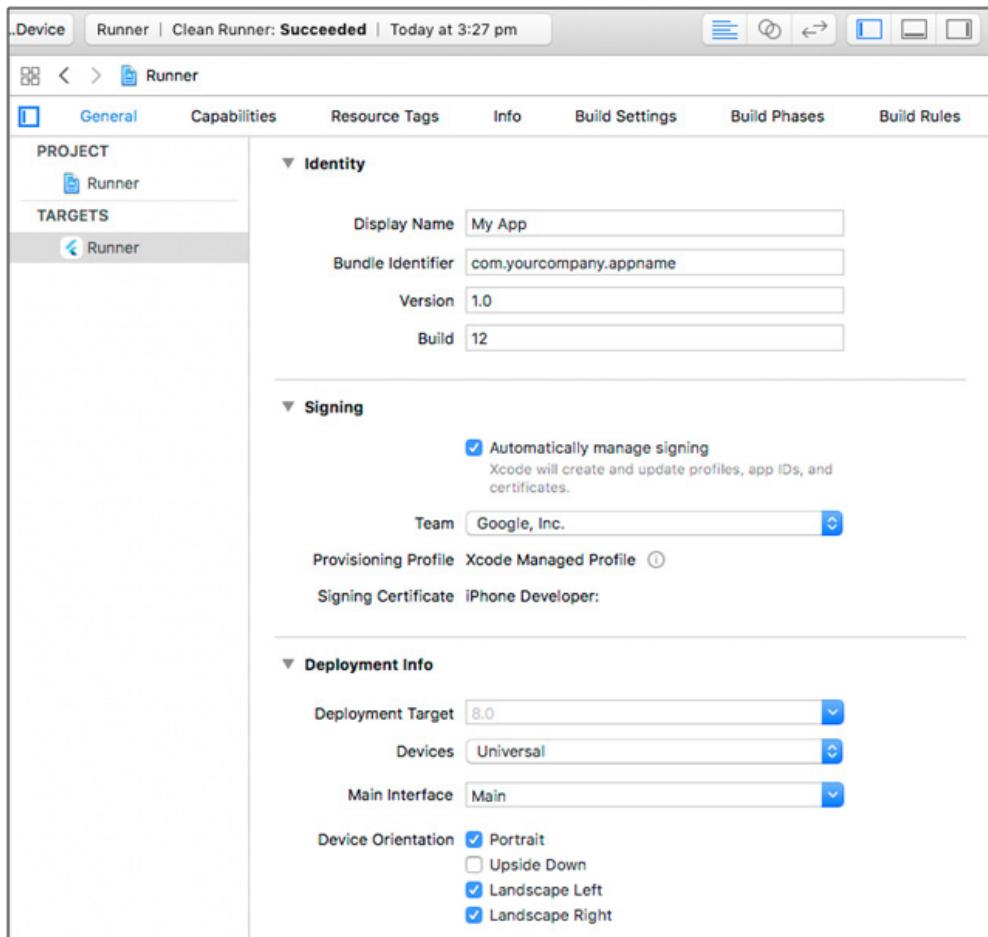
Select the team associated with your registered Apple Developer account. If required, select **Add Account...**, then update this setting.

**In the Deployment Info section:**

**Deployment Target:**

The minimum iOS version that your app will support. Flutter supports iOS 8.0 and later. If your app includes Objective-C or Swift code, it makes use of APIs which was unavailable in iOS 8. So, update these settings appropriately.

The **General** tab of your project settings should resemble the following:



## Create a Build Archive:

In this step, you'll create a build archive and upload your build to App Store Connect.

During development, you've been building, debugging, and testing with debug builds. When you're ready to ship your app to users on the App Store or TestFlight, you'll need to prepare a release build.

On the command line, follow these steps in your application directory:

1. Run flutter build ios to create a release build (flutter build defaults to --release).
2. To ensure that Xcode refreshes the release mode configuration, close and re-open your Xcode workspace. For Xcode 8.3 and later, this step is not required.

In Xcode, configure the app version and build:

1. In Xcode, open **Runner.xcworkspace** in your app's **ios** folder.

2. Select **Product** → **Scheme** → **Runner**.
3. Select **Product** → **Destination** → **Generic iOS Device**.
4. Select **Runner** in the Xcode project navigator, then select the **Runner** target in the settings view sidebar.
5. In the Identity section, update the **Version** to the user-facing version number you wish to publish.
6. In the Identity section, update the **Build** identifier to a unique build number used to track this build on App Store Connect. Each upload requires a unique build number.

Finally, create a build archive and upload it to App Store Connect:

1. Select **Product** → **Archive** to produce a build archive.
2. In the sidebar of the Xcode Organizer window, select your iOS app, then select the build archive you just produced.
3. Click the Validate App button. If any issues are reported, address them and produce another build. You can reuse the same build ID until you upload an archive.
4. After the archive has been successfully validated, click Distribute App. You can follow the status of your build in the Activities tab of your app's details page on App Store Connect.

You should receive an email within 30 minutes notifying you that your build has been validated and is available to be released to testers on **TestFlight**. At this point you can choose whether to release **TestFlight**, or go ahead and release your app to the App Store.

### **Release your app on TestFlight**

**TestFlight** allows developers to push their apps to internal and external testers. In this optional step, you'll release your build on **TestFlight**.

Navigate to the **TestFlight** tab of your app's application details page on App Store Connect.

1. Select **Internal Testing** in the sidebar.
2. Select the build to publish to testers, then click Save.
3. Add the email addresses of any internal testers. You can add additional internal users in the **Users and Roles** page of App Store Connect available

from the dropdown menu at the top of the page. You can invite up to 10,000 testers using their email address or by sharing a public link.

## Publish your iOS on Apple Store

When you're ready to release your app to the world, follow these steps to submit your app for review and release to the App Store:

1. Select **Pricing and Availability** from the sidebar of your app's application details page on **App Store Connect**, and complete the required information.
2. Select the status from the sidebar. If this is the first release of this app, its status will be **1.0 Prepare for Submission**. Complete all required fields.
3. Click **Submit for Review**.

Apple will notify you when their app review process is complete. Your app will be released according to the instructions you specified in the **Version Release** section.

Congratulations on completing the course!

You can now count yourself among Flutter application developers who can create Android and iOS apps from A to Z.

At this point, you are confident to apply for the Flutter application development exam (Exam code: AFD-200) and become a Flutter Certified application developer by Android ATC.

To know more information about how to apply for this exam, check Android ATC web site [www.androidatc.com](http://www.androidatc.com) (Certifications & Exams tab). You will also have access to more information about the exam procedures, exam samples and other Android ATC services.

Last but not least, we would love to know your feedback after completing this course and how we can make it even better for you.

Wishing you all the success in the world!

Android ATC Team