LECTURE NOTES

ON

Mobile Application Development using Flutter 2301CS413

B.Tech. 4th Semester
(Computer Science and Engineering)

<u>Prepared By</u>

Mehul D. Bhundiya

Assistant Professor

mehul.bhundiya@darshan.ac.in

9428231065

Darshan UNIVERSITY

Table of Content

Unit-1 4
Introduction to Flutter 4
1.1. Dart Programming - Overview 4
1.2. Write command-line apps 5
1.3. Variables 5
1.4. Sound null safety 7
1.5. Operators 8
1.6. Control flow 10
1.7. Function 13
1.8. Class 16
1.9. Object 17
1.10. Assessing Variable and Function 17
1.11. Flutter Introduction 18
1.12. Flutter Architecture 18
1.13. Pubspec file in flutter 20
1.14. Flutter widgets 22
Unit-2 23
UI Design, State Management, Navigation 23
2.1. Scaffold 23
2.2. Safearea 23
2.3. Text 24
2.4. Row 24
2.5. Column 25
2.6. Stack 25
2.7. Container 26
2.8. Material Components 26
2.9. Flutter App Life Cycle 27
2.10. MaterialPageRoute 31
2.11. Navigation with Named Routes 34
Unit-3 36
Database Connectivity 36
3.1. Shared Preferences 36
3.2. Sqlite Database 37
3.3. CRUD Operations in SQLite 37
3.4. FutureBuilder 41
3.5. Future, Async, Await 43

Unit-4 45
Accessing Rest API 45
4.1. Explain What is Rest Api.................................... 45
4.2. What is JSON? Explain JSON Parsing........................ 45
4.3. How to call rest api from flutter project.................. 46
Unit-5 50
Advance Application Development 50
5.1. Requesting Permission at run time.......................... 50
5.2. Work with SD Card and Files.................................. 51
5.3. Working with Camera.. 52
5.4. Generate and upload key and keystore...................... 56
5.5. How can you publish your application in Google Play Store? Explain the entire process. 57

Unit-1
Introduction to Flutter

1.1. Dart Programming - Overview

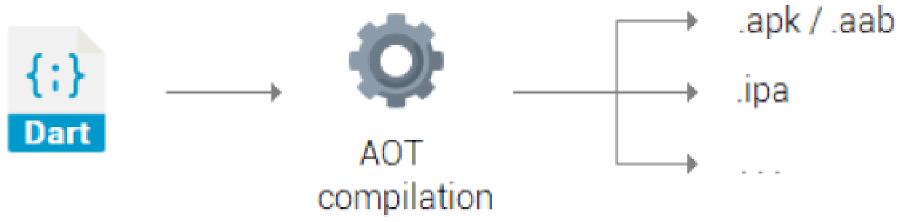
- Dart is Object-oriented language and is quite similar to that of Java Programming
- Dart is a client-optimized language for developing fast apps on any platform like Stand-alone, Web and Mobile.
- Its goal is to offer the most productive programming language for multi-platform development, paired with a flexible execution runtime platform for app frameworks.
- Dart is the open-source programming language originally developed by Google. It is meant for both server side as well as the user side.
- The Dart SDK comes with its compiler – the Dart VM and a utility dart2js.
- Which is meant for generating JavaScript equivalent of a Dart Script so that it can be run on those sites also which don't support Dart.

1.1.1. Why you use Dart

- Google created Dart and uses it internally with some of its big products such as Google AdWords.
- Made available publicly in 2011, Dart is used to build mobile, web, and server applications.
- Dart is productive, fast, portable, approachable, and most of all reactive.

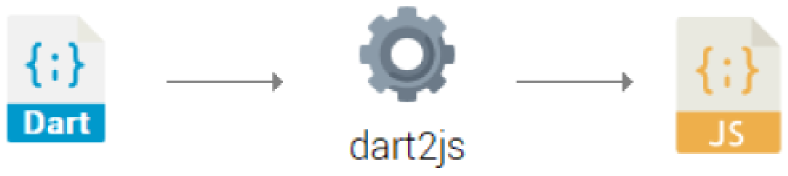
1.1.2. Supported platforms

- Dart is a very flexible language, Once the source code has been written and tested it can be deployed in many different ways:
- *Stand-alone: -*
  - Dart program is executed with the Dart Virtual Machine (DVM).
  - There's the need to download and install the DVM which to execute Dart in a command-line environment.
- *AOT Complied: -*
  - The **A**head **O**f **T**ime compilation is the act of translating a high-level programming language, like Dart, into native machine code.
  - Basically, starting from the Dart source code you can obtain a single binary file that can execute natively on a certain operating system.
  - AOT is really what makes Flutter fast and portable.



- *Web: -*

- Thanks to the dart2js tool, your Dart project can be "transpiled" into fast and compact JavaScript code.
- By consequence Flutter can be run, for example, on Firefox or Chrome and the UI will be identical to the other platforms.



1.1.3. Package system

- Dart's core API offers different packages, such as *dart:io* or *dart:collection*, that expose classes and methods for many purposes.
- In addition, there is an official online repository called pub containing packages created by the Dart team, the Flutter team or community users.
- If you head to https://pub.dev you will find an endless number of packages for any purpose: I/O handling, XML serialization/de-serialization, localization, SQL/NoSQL database utilities and much more.

1.2. Write command-line apps

- In dart main() function is predefined method.
- main() method acts as the entry point to the application.
- A dart script needs the main() method for execution of the code.
- The program code goes like this

```
void main() {
 print("Welcome to Darshan University");
}
```

1.3. Variables

- Variable is used to store the value and refer the memory location in computer memory.
- When we create a variable, the Dart compiler allocates some space in memory.
- The size of the memory block of memory is depended upon the type of variable.

1.3.1. Rule to Create Variable

- In Dart, all variables are declared public by default.
- By starting the variable name with an underscore (_), you can declare it as private.
- By declaring a variable private, you are saying it cannot be accessed from outside classes/functions. it can be used only from within the declaration class/function
- The variable cannot contain special characters such as whitespace, mathematical symbol, Unicode character, and keywords.

- The first character of the variable should be an alphabet (A to Z & a to z).
- Digits are not allowed as the first character.
- The special character such as #, @, ^, &, * are not allowed.
- The variable name should be meaningful.

1.3.2. Number Variable

- Declaring variables as numbers restricts the values to numbers only. Dart allows numbers to be int(integer) or double.
- int: - Use the int declaration if your numbers do not require decimal point precision, like 8 or 22.
- **double: -** Use the double declaration if your numbers require decimal point precision, like 8.5 or 88.25.
- Both int and double allow for positive and negative numbers, and you can enter extremely large numbers and decimal precision since they both use 64-bit (computer memory) values.
- e.g.,

```
void main() {
 int age=19;
 double price=25.50;
}
```

1.3.3. Strings Variable

- Declaring variables as String allows values to be entered as a sequence of text characters.
- To add a single line of characters, you can use single or double quotes like 'Darshan University' or "Darshan University".
- e.g.

```
void main() {
 String name = 'Darshan University';
 String name = "Darshan University";
}
```

1.3.4. Booleans Variable

- Declaring variables as bool (Boolean) allows a value of true or false to be entered.
- e.g.

```
void main() {
 bool isDone = false;
 isDone = true;
}
```

1.3.5. Lists

- Declaring variables as List (comparable to arrays) allows multiple values to be entered.
- List is an ordered group of objects.

- The **dart:core** library provides the List class that enables creation and manipulation of lists.
- To access elements, the List uses zero-based indexing, where the first element index is at 0, and the last element is at the List length is minus 1.
- A List can be a **fixed length** or **growable**, depending on your needs.
- By default, a List is created as growable by using List() or [].
- To create a fixed-length List, you add the number of rows required by using this format: List(15).
- e.g.

```
void main() {
 // List Growable
 List days = List();

 // or
 List days = [];
 List days = ['Sunday', 'Monday, 'Tuesday];

 // List fixed-length
 List days = List(7);
}
```

1.3.6. Maps

- The Map data type represents a set of values as key-value pairs.
- Keys and values on a map may be of any data type.
- Mapping allows recalling values by their Key ID.
- Keep in mind that the Key needs to be unique since the Value is retrieved by the Key.
- e.g.

```
void main() {
 // Maps - An object that associates keys and values.
 Map <int, String>  students = {'101': "Ravi", '102':
"Shyam", '103': "Mehul"};
}
```

1.4. Sound null safety

- Null Safety in simple words means a variable cannot contain a 'null' value.
- It prevents errors that result from unintentional access of variables set to null.
- There are two types on variables:
  - Non-nullable types
  - Nullable Types

1.4.1. Nullable & Non-Nullable Types

- *Non-nullable types*
    - When we use null safety, all types are by default non-nullable.
    - For example, an int variable must have an integer value.

```
void main() {
 int number;
  number = 0;
}
```

-
    - If a variable is non-nullable, it must always be set to a non-null value.
- *Nullable types*
    - Nullable type (?) operators specify if a variable can be null.
    - A nullable variable does not need to be initialized before being used.

```
void main() {
 String? houseLocationName;  // By default, it's set
to null.
 int? number = 36;  // By default, it's set to non-
null.
 number = null; // It's possible to reassign it to
null
```

1.5. Operators

- An operator is a symbol used to perform arithmetic, equality, relational, type test, assignment, logical, conditional, and cascade notation.

1.5.1. Arithmetic operators

- Arithmetic operators are commonly used on int and double to build expressions.

Operator	Description	Sample Code
+	Add two values	5 + 2 = 7
-	Subtract two values	5 - 2 = 3
*	Multiply two values	5 * 2 = 10
/	Divide two values	5 / 2 = 2.5
~/	Integer division of two values	5 ~/ 2 = 2
%	Modulo, Remainder of an int division	5 % 2 = 1

1.5.2. Relational operators

- Equality and relational operators are used in boolean expression, generally inside if statements or as a stop condition of a while loop

Operator	Description	Sample Code
==	Equal	5 == 2 = false
!=	Not equal	5 != 2 = true
>	Greater than	5 > 2 = true

<	Less than	5 < 2 = false
>=	Greater than or equal to	5 >= 2 = true 5 >= 5 true
<=	Less than or equal to	5<= 2 = false 2 <= 2 = true

1.5.3. Type test operators

- They are used to check the type of an object at runtime.

Operator	Description	Sample Code
as	Cast a type to another	obj as String
is	True if the object has a certain type	obj is double
is!	False if the object has a certain type	obj is! int

1.5.4. Assignment operators

- Operators which are used to perform comparison operation on the operands.

Operator	Description	Sample Code
=	Assigns value	a=5
??=	Assigns value only if variable being assigned to has a value of null	Null ??= 2 (ans: 2) 7 ??= 2 **=** (ans: 7)
+=	Adds to current value	5 += 2 = 7
-=	Subtracts from current value	5 -= 2 = 3
*=	Multiplies from current value	5 *= 2 = 10
/=	Divides from current value	5 /= 2 = 2.5

1.5.5. Logical operators

- When you have to create complex conditional expressions you can use the logical operators

Operator	Description	Sample Code
!	! is a logical 'not'. Returns the opposite value of the variable/expression.	if (!(7 > 3)) = false
&&	&& is a logical 'and'. Returns true if the values of the variable/expression are all true.	if ((7 > 3) &&(3 > 7)) = false
||	|| is a logical 'or'. Returns true if at least one value of the variable/expression is true.	if ((7 > 3) ||(3 > 7)) = true

1.5.6. Conditional/Ternary Operator

- The conditional operator is a shorthand version of an if-else condition.
- There are two types of conditional operator syntax in Dart.
- One with a null safety check and the other is the same old one we encounter normally.

Operator	Description	Sample Code
condition ? value1 : value2	If the condition evaluates to true, it returns value1. If the condition evaluates to false, it returns value2. The value can also be obtained by calling methods.	(7 > 3) ? true : false = true (7 < 3) ? true : false = false

1.5.7. Null Aware Operator

- It depicts a conditional statement that is similar to a conditional operator statement.
- when you want to evaluate and return an expression if another expression resolves to null.
- It is also called the if-null operator.

Operator	Description	Sample Code

expression1 ?? Value	The null-aware operator is ??, which returns the expression on its left unless that expression's value is null	var b;
		String a = b ?? 'Hello';
		print(a)//output: Hello

1.5.8. Cascade Notation Operators

- This operator allows you to perform a sequence of operation on the same element.

Operator	Description	Sample Code

..	The cascade notation is represented by double dots (..) and allows you to make a sequence of operations on the same object.	Matrix4.identity()
		..scale(1.0, 1.0)
		..translate(30,30);

1.6. Control flow

1.6.1. if statement

- The if statement compares an expression, and if true, it executes the code logic.
- The if statement also supports multiple optional else statements, which are used to evaluate multiple scenarios.
- There are two types of else statements: else if and else.
- You can use multiple else if statements, but you can have only one else statement, usually used as a catchall scenario.

```
void main() {
 final random = 13;
 if (random % 2 == 0)
       print("Number is even");
 else
       print("Number is odd");
}
```

1.6.2. switch statement

- The switch statement compares integer, string, or compile-time constants using == (equality).
- The switch statement is an alternative to the if and else statements.
- The switch statement evaluates an expression and uses the *case* clause to match a condition and executes code inside the matching case.
- Each *case* clause ends by placing a break statement as the last line.
- Use the *default* clause to execute code that is not matched by any of the case clauses, placed after all the case clauses.

```
void main() {
 String operation = '+';
 switch (operation) {
     case '+':
         print(5+2);
         break;
      case '-':
        print(5-2);
        break;
     case '*':
        print(5*2);
        break;
       default:
        print(5/2);
     }
}
```

1.6.3. for Loop

- The standard for loop allows you to iterate a List of values. Values are obtained by restricting the number of loops by a defined length.

```
void main() {
 //for(initialization; condition; expression)
 for (int i = 0; i < 5; i++) {
         print('Darshan University');
     }
}
```

1.6.4. for…in Loop

- The for...in loop is used to loop through an object's properties.
- In each iteration, one property from the object is assigned to the variable.

- This loop continues till all the properties of the object are

```
void main() {
 var obj = [12,13,14];
 //for(variablename in listOfObject)
    for (var prop in obj) {
        print(prop);
    }
}
```

exhausted.

1.6.5. for each loop

- The for-each loop iterates over all elements in some collection and passes the elements to some specific function.

```
void main() {
 var obj = [12,13,14];
 //collection.forEach(void f(value))
   obj.forEach((var num)=> print(num));
}
```

1.6.6. while and do-while

- Both the while and do-while loops evaluate a condition and continue to loop as long as the condition returns a value of true.
- The while loop evaluates the condition before the loop is executed.
- The do-while loop evaluates the condition after the loop is executed

```
void main() {
 // while - evaluates the condition before the loop
 while (isClosed){
     askToOpen();
 }
 // do while - evaluates the condition after the loop
 do {
     askToOpen();
 } while (isClosed);
}
```

at least once.

1.6.7. Break Statement

- Using the break statement allows you to stop looping by evaluating a condition inside the loop.

1.6.8. Continue Statement

- By using the continue statement, you can stop at the current loop location and skip to the start of the next loop iteration.

1.7. Function

- Functions are the building blocks of readable, maintainable, and reusable code.
- A function is a set of statements to perform a specific task.
- Functions organize the program into logical blocks of code.
- Once defined, functions may be called to access code.
- This makes the code reusable, easy to read & maintain the program’s code.

```
return_type  function_name ( parameters ) {
    // Body of function return value;
}
```

- A function declaration tells the compiler about a function's name, return type, and parameters.
  - o function_name defines the name of the function.
  - o return_type defines the data type in which output is going to come.
  - o return value defines the value to be returned from the function.
- **Function Call: -** function_name (argument_list);
  - o function_name defines the name of the function.
  - o argument_list is the list of the parameter that the function

```
void main() {
    // Calling the function
   var output = add(10, 20);
   // Printing output
   print(output);
}
int add(int a, int b)
{
  // Creating function
  int result = a + b;
  // returning value result
  return result;
}
```

requires.

1.7.1. Arrow/Lambda Function

- The arrow function allows us to create with single expression.

- We can omit the curly brackets and the return keyword.

```
return_type function_name(parameters...) =>
expression;
```

  - return_type consists of datatypes like void,int,bool, etc..
  - function_name defines the name of the function.
  - parameters are the list of parameters function requires.

1.7.2. Optional Parameter Function

```
void main() {
 perimeterOfRectangle(47, 57);
}
void perimeterOfRectangle(int length, int breadth) =>
print('The perimeter of rectangele is ${2 * (length +
breadth)}');
```

- Function overloading is not supported in Dart at all.
- Optional parameters are those parameters that don't need to be specified when calling the function.
- Optional parameters allow us to pass default values to parameters that we define.
- There are two types of optional parameters
  - Ordered (positional) optional parameters
  - Named optional parameters
  - Anonymous function
- ***Ordered (positional) optional parameter Function:***
  - The square brackets [] are used to specify optional positional parameters.
  - Optional parameters can be used when arguments need not be compulsorily passed for a function’s execution.

```
void main() {
   print(pow(2, 2));
    print(pow(2, 3));
    print(pow(3));
}
int pow(int x, [int y = 2]) {
    int r = 1;
    for (int i = 0; i < 2; i++) {
        r *= x;
    }
    return r;
}
```

- ***Named optional parameters Function:***
  - Optional named parameters are specified inside curly {} brackets.
  - When passing the optional named parameter, we have to specify both the parameter name and value, separated with colon.

  - It is necessary for you to use the name of the parameter if you want to pass your argument.

```
void main() {
   var name = "John Doe";
     var occupation = "carpenter";
     info(name, occupation: occupation);
 }
void info(String name, {String occupation}) {
    print("$name is a $occupation");
}
```

- ***Anonymous function***
  - Anonymous functions do not have a name.
  - An anonymous function can have zero or more parameters with optional type annotations.

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

  - Example:

```
void main() {
  var list = ["James","Patrick","Mathew","Tom"];
  print("Example of anonymous function");
  list.forEach((item) {
      print('${list.indexOf(item)}: $item');
});
}
```

1.7.3. Nested function

- A nested function, also called an inner function.
- It is a function defined inside another function.
- We have a helper buildMessage function which is defined inside the main function.

```
void main() {
    String buildMessage(String name, String occupation) {
        return "$name is a $occupation";
    }
    var name = "John Doe";
    var occupation = "gardener";
    var msg = buildMessage(name, occupation);
    print(msg);
}
```

1.7.4. Function as parameter

- A function can be passed to other functions as a parameter.
- This function is called a higher-order function.

```
void main() {
    exec(10, plusOne);
    exec(10, double);
}
void exec(int i, Function f) {
    print(f(i));
}
int plusOne(int i) => i + 1;
int double(int i) => i * i;
```

1.8. Class

- Dart is an object-oriented programming language.
- It supports the concept of class, object … etc.
- We use the class keyword to define class.

```
class ClassName {
    // Body of class
}
```

- Classes are the blueprint of the object.
- A class can contain fields, functions, constructors, etc.
- It is a wrapper that binds/encapsulates the data and functions together; that can be accessed by creating an object.

```
void main() {
   // Defining class
 class Student {
   var stdName;
   var stdAge;
   var stdRoll_nu;

   // Class Function
    showStdInfo() {
        print("Student Name is : ${stdName}");
        print("Student Age is : ${stdAge}");
        print("Student Roll Number is : ${stdRoll_nu}")
}
```

1.9. Object

- Dart is object-oriented programming, and everything is treated as an object.
- An object is a variable or instance of the class used to access the class's properties.
- Objects have two features - state and behaviour.
- Syntax (with new Keyword)

```
var object_name  = new ClassName(<constructor_arguments>);
```

- Syntax (without new Keyword)

```
var object_name  =  ClassName(<constructor_arguments>);
```

- Example

```
void main() {
    var std = Student();
 }

 // Defining class
 class Student {
   var stdName;
   var stdAge;
   var stdRoll_nu;

   // Class Function
    showStdInfo() {
        print("Student Name is : ${stdName}");
        print("Student Age is : ${stdAge}");
        print("Student Roll Number is : ${stdRoll_nu}")
}
```

1.10. Assessing Variable and Function

- We can access the fields and methods of the class using (.) operator with the object name.
- like objectName.propName or objectName.methoName()

1.10.1. Access Modifiers

- In Dart we don't have keywords like public, protected, and private to control the access scope for a property or method.
- It uses _ (underscore) at the start of the variable name to make a data member private.
- In Dart, the privacy is at library level rather than class level.

- It means other classes and functions in the same library still have the access.
- So, a data member is either public (if not preceded by _) or private (if preceded by _).
- Example

```
class A
{
    String first=‘ ’;
    String _second=‘ ’;
}
void main()
{
     A a = new A();
     a.first = 'New first';
     a._second = 'New second';
     print('${a.first}: ${a._second}');
}
```

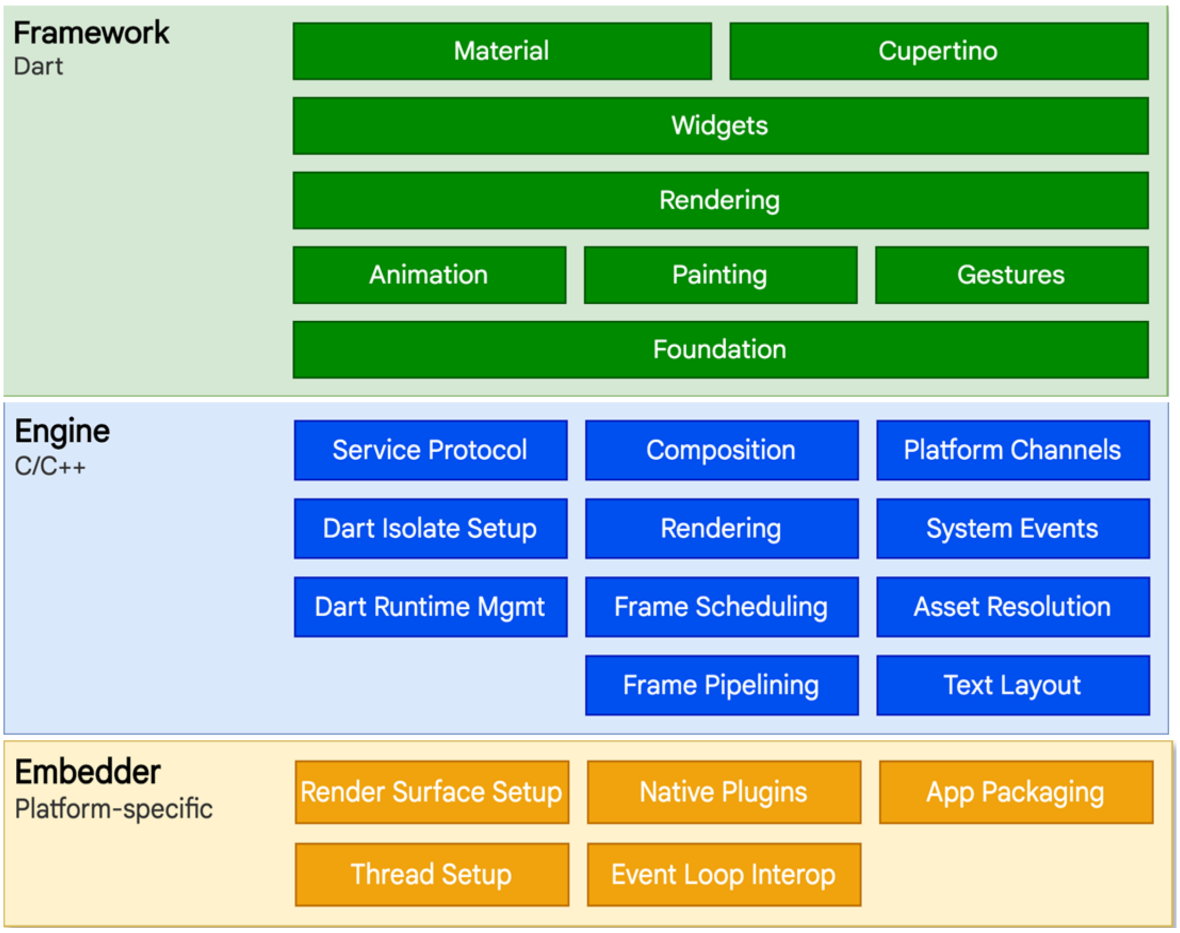
1.11. Flutter Introduction

- Flutter is an open-source framework developed by Google
- It used for building beautiful, natively compiled, multi-platform applications from a single codebase.
- It uses dart as a programming language for fast apps on any platform.
- The first version of Flutter was known as Sky and ran on the Android operating system.
- Flutter 1.0 was release on 11 December, 2019.
- Flutter 3.3 was announced On August 30, 2022.
- It Support two themes
  - Material Design widgets implement Google's design language of the same name.
  - Cupertino widgets implement Apple's iOS Human interface guidelines
- Flutter Supports IDES and Editors via plugin.
  - IntelliJ IDEA
  - Android Studio
  - Visual Studio Code
  - Emacs

1.12. Flutter Architecture

- Flutter is a cross-platform UI toolkit.
- It is designed to allow code reuse across operating systems such as iOS and Android.
- During development apps run in a VM that offers stateful hot reload of changes without needing a full recompile.

- For release apps are compiled directly to machine code or to JavaScript if targeting the web.
- It is designed as an extensible, layered system.
- Every part of the framework level is designed to be optional and replaceable.



- Flutter is designed as an extensible, layered system.
- It exists as a series of independent libraries that each depend on the underlying layer.
- No layer has privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable.
- Flutter applications are packaged in the same way as any other native application.
    - Embedder (lowest layer)
    - Engine
    - Framework (highest layer)

1.12.1. Embedder layer

- An entry point is provided by a platform-specific embedder.
- It coordinates with the underlying operating system to access services such as
    - accessibility,
    - rendering surfaces,
    - input.

- The embedder is written in a platform-specific language like.
    - Java and C++ for Android
    - Objective-C/Objective-C++ for iOS and macOS
    - C++ for Windows and Linux
- Flutter code can be embedded into an existing application as a module or as the complete application’s content using the embedder.

1.12.2. Engine layer

- The engine layer is written in C/C++.
- It takes care of the input, output, network requests.
- It handles the difficult translation of rendering whenever a frame needs to be painted.
- Flutter uses Skia as its rendering engine.
- It is revealed to the Flutter framework through the dart : ui.
- It wraps the principal C++ code in Dart classes.

1.12.3. Framework layer

- The framework layer is the part where users can interact with Flutter.
- The Flutter framework provides a reactive framework that is written in Dart.
- It comprises of
    - Rendering
    - Widgets
    - Material and Cupertino
- It also has foundational classes and building block services like animation, drawing, and gestures, which are required for writing a Flutter application.

1.13. Pubspec file in flutter

- Every Flutter project includes a pubspec.yaml file, often referred to as the pubspec.
- A basic pubspec is generated when you create a new Flutter project.
- It’s located at the top of the project tree and contains metadata about the project that the Dart and Flutter tooling needs to know.
- The pubspec is written in YAML, which is human readable, but be aware that white space (tabs v spaces) matters.
- The pubspec file specifies dependencies that the project requires, such as particular packages (and their versions), fonts, or image files.
- It also specifies other requirements, such as dependencies on developer packages or particular constraints on the version of the Flutter SDK.

```
name: <project name>
description: A new Flutter project.

publish_to: 'none'

version: 1.0.0+1

environment:
  sdk: ">=2.7.0 <3.0.0"

dependencies:
  flutter:       # Required for every Flutter project
    sdk: flutter # Required for every Flutter project

  cupertino_icons: ^1.0.2 # Only required if you use Cupertino
(iOS style) icons

dev_dependencies:
  flutter_test:
    sdk: flutter # Required for a Flutter project that includes
tests

flutter:

  uses-material-design: true # Required if you use the Material
icon font

  assets:  # Lists assets, such as image files
    - images/a_dot_burr.jpeg
    - images/a_dot_ham.jpeg

  fonts:              # Required if your app uses custom fonts
    - family: Schyler
      fonts:
        - asset: fonts/Schyler-Regular.ttf
        - asset: fonts/Schyler-Italic.ttf
          style: italic
    - family: Trajan Pro
      fonts:
        - asset: fonts/TrajanPro.ttf
        - asset: fonts/TrajanPro_Bold.ttf
          weight: 700
```

1.14. Flutter widgets

- The first thing to note is that in Flutter, everything is a widget.
- A widget is simply an instruction that you place within your code and they are the basic building blocks of a Flutter application's UI.
- Widgets indicate how its configuration and status should appear in their display.
- When a widget's state changes, it rebuilds its description.
- The framework compares to the previous description to see what changes in the underlying render tree to transition from one state to the next.
- A widget can be in the form of a button, an image, an icon, or a layout, and placing the widgets together creates a widget tree.

1.14.1. Widget States

- State is the behavior of an app at any given moment.
- It is a widget's information when it is first created and how it defines the properties of that widget.
- This information might change during the lifetime of the widget.
- To build the UI in Flutter there is two types of widgets:
  - Stateless widgets
  - Stateful widgets
- *Stateless Widget:*
  - A stateless widget is a widget that describes part of the user interface by building a design that describe the user interface.
  - Stateless widget are useful when user interface does not depend other than the configuration information.
  - The build method of a stateless widget is typically only called in three situations:
    - The first time the widget is inserted in the tree.
    - Widget's parent changes its configuration.
    - When an InheritedWidget it depends on changes.

```
class GreenFrog extends StatelessWidget
{
    const GreenFrog({ super.key });

    @override
    Widget build(BuildContext context)
    {
        return Container(
      color: const Color(0xFF2DBD3A),
              );
    }
}
```

Unit-2
UI Design, State Management, Navigation

2.1. Scaffold

- The Scaffold widget implements the basic Material Design visual layout, allowing you to easily add various widgets such as AppBar, BottomAppBar, FloatingActionButton, Drawer, SnackBar, BottomSheet, and more.
- AppBar: The AppBar widget usually contains the standard title, toolbar, leading, and actions properties, as well as many customization options.
- title: The title property is typically implemented with a Text widget. You can customize it with other widgets such as a DropdownButton widget.
- leading: The leading property is displayed before the title property. Usually this is an IconButton or BackButton.
- actions: The actions property is displayed to the right of the title property. It's a list of widgets aligned to the upper right of an AppBar widget usually with an IconButton or PopupMenuButton.
- flexibleSpace: The flexibleSpace property is stacked behind the Toolbar or TabBar widget. The height is usually the same as the AppBar widget's height. A background image is commonly applied to the flexibleSpace property, but any widget, such as an Icon, could be used.

```
Scaffold(
 appBar: AppBar(
       actions:[ IconButton(
                   icon: constIcon(Icons.info),
       onPressed:() {...} ),
             ]
 )
)
```

2.2. Safearea

- The SafeArea widget automatically adds sufficient padding to the child widget to avoid intrusions by the operating system.
- You can optionally pass minimum padding or a Boolean value to not enforce pad- ding on the top, bottom, left, or right.

```
    body: Padding(
        padding: EdgeInsets.all(16.0),
        child: SafeArea(
              child: SingleChildScrollView(
               child: Column( children: <Widget>[], ),
              ),
        ),
    ),
```

2.3. Text

- The Text widget displays a string of text with a single style.
- The string might break across multiple lines or might all be displayed on the same line depending on the layout constraints.

```
    Text(
      'Hello, $_name! How are you?',
      textAlign: TextAlign.center,
      overflow: TextOverflow.ellipsis,
      style: const TextStyle(fontWeight: FontWeight.bold),
    )
```

2.4. Row

- A widget that displays its children in a horizontal array.
- To cause a child to expand to fill the available horizontal space, wrap the child in an Expanded widget.
- The Row widget does not scroll.
- If you have a line of widgets and want them to be able to scroll if there is insufficient room, consider using a ListView.

```
Row(
  children: const <Widget>[
    Expanded(
      child: Text('Deliver features faster', textAlign:
TextAlign.center),
    ),
    Expanded(
      child: Text('Craft beautiful UIs', textAlign:
TextAlign.center),
    ),
    Expanded(
      child: FittedBox(
        child: FlutterLogo(),
      ),
    ),
  ],
)
```

2.5. Column

- To cause a child to expand to fill the available vertical space, wrap the child in an Expanded widget.
- The Column widget does not scroll.
- If you have a line of widgets and want them to be able to scroll if there is insufficient room, consider using a ListView.

```
Column(
  children: const <Widget>[
    Text('Deliver features faster'),
    Text('Craft beautiful UIs'),
    Expanded(
      child: FittedBox(
        child: FlutterLogo(),
      ),
    ),
  ],
)
```

2.6. Stack

- This class is useful if you want to overlap several children in a simple way, e.g., having some text and an image, overlaid with a gradient and a button attached to the bottom.
- Each child of a Stack widget is either positioned or non-positioned.
- Positioned children are those wrapped in a Positioned widget that has at least one non-null property.

- The stack sizes itself to contain all the non-positioned children, which are positioned according to alignment.

```
Stack(
  children: <Widget>[
    Container(
      width: 100,
      height: 100,
      color: Colors.red,
    ),
    Container(
      width: 90,
      height: 90,
      color: Colors.green,
    ),
    Container(
      width: 80,
      height: 80,
      color: Colors.blue,
    ),
  ],
)
```

2.7. Container

- A convenience widget that combines common painting, positioning, and sizing widgets.
- A container first surrounds the child with padding and then applies additional constraints to the padded extent.
- The container is then surrounded by additional empty space described

```
Center(
  child: Container(
    margin: const EdgeInsets.all(10.0),
    color: Colors.amber[600],
    width: 48.0,
    height: 48.0,
  ),
)
```

from the margin.

2.8. Material Components

- Material Components for Flutter unite design and engineering with a library of components that create a consistent user experience across apps and platforms.

- These components are updated to ensure consistent pixel-perfect implementation, adhering to Google's front-end development standards.
- It provides visual, behavioural, and motion-rich widgets implementing the Material Design guidelines for the below types of widgets.
  - App structure and navigation
  - Buttons
  - Input and selections
  - Dialogs, alerts, and panels
  - Information displays
  - Layout

2.9. Flutter App Life Cycle

- The lifecycle of the Flutter App is the show of how the application will change its State.
- It helps in understanding the idea driving the smooth progression of our applications.
- Everything in Flutter is a Widget, so before thinking about Lifecycle, we should think about Widgets in Flutter.
- Flutter has majorly two types of widgets:
  - Stateless Widgets
  - Stateful Widgets

2.9.1. Stateless Widgets

- Stateless Widgets are those widgets that don't need to deal with the State as they don't change powerfully at runtime.
- It becomes permanent like on variables, buttons, symbols, and so forth, or any express that can't be changed on the application to recover information.
- Returns a widget by overwriting the build method. We use it when the UI depends on the data inside the actual item.

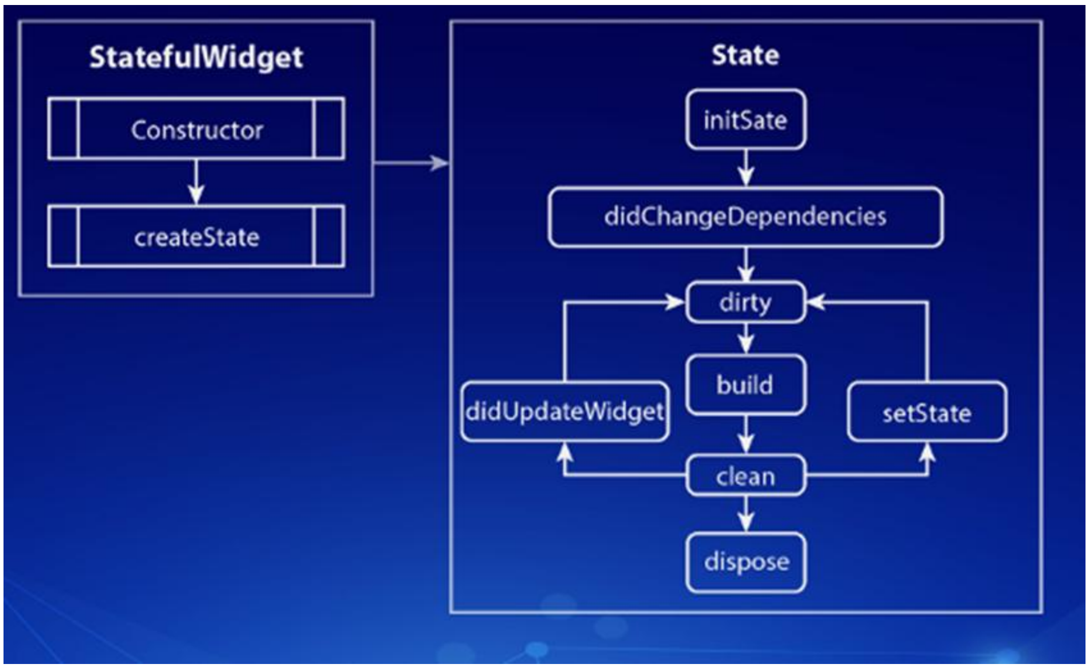
2.9.2. Stateful Widgets

- Stateful Widgets are those widgets that hold the State and the UI being portrayed can change progressively at runtime.
- It is a changeable widget, so it is attracted numerous times during its lifetime.
- We utilize this when the user progressively updates the application screen.
- This is the most significant of the multitude of widgets, as it has state widgets, everybody realizes that something has been updated on our screen.
- Thus, as we are examining the LifeCycle of Flutter, we will focus on 'Stateful Widgets' as they need to deal with the State.

2.9.3. Stages Of App Lifecycle

- The life cycle depends on the state and how it changes.
- A stateful widget has a state so we can clarify the life cycle of flutter dependent on it.
- Stage of the life cycle:
  - createState()
  - initState()

- didChangeDependencies()
- build()
- didUpdateWidget()
- setState()
- deactivate()
- dispose()



- *createState():* - This method is called when we create another Stateful Widget. It is an obligatory strategy. The createState() returns a case of a State-related with it.

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

@override
HomeScreenState<StatefulWidget> createState() =>
HomeScreen();
}
```

- *initState():* -
  - This is the strategy that is considered when the Widget is made interestingly and it is called precisely once for each State object.
  - If we characterize or add some code in the initState() method this code will execute first even before the widgets are being built.
  - This method needs to call super.initState() which essentially calls the initState of the parent widget (Stateful widget).

  - Here you can initialize your variables, objects, streams, AnimationController, and so on.

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

- *didChangeDependencies():* - □
  - This method is called following the initState() method whenever the widget initially is constructed.
  - You can incorporate not many functionalities like API calls dependent on parent data changes, variable re-initializations, and so forth.

```
@override
void didChangeDependencies() {

}
```

- *build ():* - □
  - This strategy is the main method as the rendering of all the widgets relies upon it.
  - It is called each time when we need to render the UI Widgets on the screen.
  - At whatever point you need to update your UI or on the other hand on the off chance that you click hot-reload, the Flutter structure modifies the build() strategy!.
  - Assuming you need to expressly revamp the UI if any information is transformed, you can utilize setState() which teaches the framework to again run the form method

```
@override
Widget build(BuildContext context) {
  return Scaffold()
}
```

- *didUpdateWidget():* - □
  - This strategy is utilized when there is some adjustment of the configuration by the Parent widget.
  - It is essentially called each time we hot reload the application for survey the updates made to the widget.
  - If the parent widget changes its properties or designs, and the parent needs to modify the child widget, with a similar Runtime Type, then, at that point, didUpdateWidget is triggered.
  - This withdraws to the old widget and buys into the arrangement changes of the new widget.

```
@protected
void didUpdateWidget(Home oldWidget) {
  super.didUpdateWidget(oldWidget);
}
```

- *setState(): -*
  - The setState() method illuminates the framework that the internal state of this item has changed in a manner that may affect the UI which makes the structure plan a build for this State of the object.
  - It is an error to call this method after the system calls dispose().
  - This inside state could conceivably influence the UI apparent to the user and subsequently, it becomes important to rebuild the UI.

```
void function(){
  setState(() {});
}
```

- *deactivate(): -*
  - This method is considered when the State is removed out from the tree, however, this strategy can be additionally be re-embedded into the tree in another part.
  - This strategy is considered when the widget is as of now not joined to the Widget Tree yet it very well may be appended in a later stage.
  - The best illustration of this is the point at which you use Navigator.
  - push to move to the following screen, deactivate is called because the client can move back to the past screen and the widget will again be added to the tree.

```
@override
void deactivate(){
  super.deactivate();
}
```

- *dispose(): -*
  - This strategy is essentially something contrary to the initState() method and is likewise important.
  - It is considered when the object and its State should be eliminated from the Widget Tree forever and won't ever assemble again.
  - Here you can unsubscribe streams, cancel timers, dispose animations controllers, close documents, and so on.
  - At the end of the day, you can deliver every one of the assets in this strategy.
  - Presently, later on, if the Widget is again added to Widget Tree, the whole lifecycle will again be followed

```
@override
void dispose(){
  super.dispose();
}
```

2.9.4. Stateless vs Stateful Widget

Stateless widget	**Stateful widget**
They are static widgets; they are updated only when initialized.	They are dynamic in nature.
It is not dependent on data changes or behavior changes.	It is dependent and changes when the user interacts.
Examples are Text, Icons, or a RaisedButton.	Examples are Checkbox, RadioButton, or Slider.
Doesn't include a setState().	It has an internal setState().
Cannot be updated during the application's runtime. An external event is necessary for the trigger.	It can be updated during the runtime.
It doesn't have life cycle	It contains lifecycle

2.10. MaterialPageRoute

- A modal route that replaces the entire screen with a platform-adaptive transition.
- For Android, the entrance transition for the page zooms in and fades in while the existing page zooms out and fades out.
- The exit transition is similar, but in reverse.
- For iOS, the page slides in from the right and exits in reverse.
- The page also shifts to the left in parallax when another page enters to cover it.
- By default, when a modal route is replaced by another, the previous route remains in memory.
- To free all the resources when this is not necessary, set *maintainState* to false.
- The *fullscreenDialog* property specifies whether the incoming route is a *fullscreen* modal dialog.
- On iOS, those routes animate from the bottom to the top rather than horizontally.
- The type T specifies the return type of the route which can be supplied as the route is popped from the stack via *Navigator.pop* by providing the optional result argument.

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

void main() {
  runApp(const MaterialApp(
    title: 'Navigation Basics',
    home: FirstRoute(),
  ));
}

class FirstRoute extends StatelessWidget {
  const FirstRoute({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Route'),
      ),
      body: Center(
        child: ElevatedButton(
          child: const Text('Open route'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => const
SecondRoute()),
            );
          },
        ),
      ),
    );
  }
}

class SecondRoute extends StatelessWidget {
  const SecondRoute({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Second Route'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: const Text('Go back!'),
        ),
      ),
import 'package:flutter/material.dart';

class SecondRoute extends StatelessWidget {
  const SecondRoute({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Second Route'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: const Text('Go back!'),
        ),
      ),
    );
  }
}
```

- To switch to a new route, use the Navigator.push() method.
- The push() method adds a Route to the stack of routes managed by the Navigator.

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

- By using the Navigator.pop() method.

- The pop() method removes the current Route from the stack of routes managed by the Navigator.

```
// Within the SecondRoute widget

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

2.11. Navigation with Named Routes

- If you need to navigate to the same screen in many parts of your app, this approach can result in code duplication. The solution is to define a named route, and use the named route for navigation.
- To work with named routes, use the Navigator.pushNamed() function. This example replicates the functionality from the original recipe, demonstrating how to use named routes using the following steps:
  - Create two screens
  - Define the routes
  - Navigate to the second screen using Navigator.pushNamed()
  - Return to the first screen using Navigator.pop()

```
import 'package:flutter/material.dart';
void main() {
  runApp(
    MaterialApp(
      title: 'Named Routes Demo',
      initialRoute: '/',
      routes: {
        '/': (context) => const FirstScreen(),
        '/second': (context) => const SecondScreen(),
      },
    ),
  );
}

class FirstScreen extends StatelessWidget {
  const FirstScreen({super.key});
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Screen'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pushNamed(context, '/second');
          },
              child: const Text('Launch screen'),

          ),
        ),
        );
    }
  }

  class SecondScreen extends StatelessWidget {
    const SecondScreen({super.key});

    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: const Text('Second Screen'),
        ),
        body: Center(
          child: ElevatedButton(
            onPressed:  () {
              Navigator.pop(context);
            },
            child: const Text('Go back!'),
          ),
        ),
      );
    }
  }
```

Unit-3
Database Connectivity

3.1. Shared Preferences

- SharedPreferences is used for storing data key-value pair in the Android and iOS.
- SharedPreferences in flutter uses **NSUserDefaultson** iOS and **SharedPreferences** on Android, providing a persistent store for simple data.
- SharedPreferences is the best option to store a small amount of data in flutter projects.
- Shared Preferences is the way in which one can store and retrieve small amounts of primitive data as key/value pairs to a file on the device storage such as String, integer, float, Boolean that make up your preferences in an XML file inside the app on the device storage.
- For example – Storing the username and login status in the **shared_preferences**.
- In Android Application consider a case when the user login or not, we can store the state of login- logout in the **shared_preferences**, so that the user does not need to write a password, again and again, we can save the login state in bool variable it is a small amount of data there is no need of a database, we can store it in **shared_preferences** to access it fast.
- To use this plugin, add **shared_preferences** as a dependency in your **pubspec.yaml** file.
- **Example**:

```
// Obtain shared preferences.
final prefs = await SharedPreferences.getInstance();

// Save an integer value to 'num' key.
await prefs.setInt('num', 10);

// Save an boolean value to 'flag' key.
await prefs.setBool('flag', true);

// Save an double value to 'decnum' key.
await prefs.setDouble('decnum', 1.5);

// Save an String value to 'name' key.
await prefs.setString('name', 'Start');

// Save an list of strings to 'items' key.
await prefs.setStringList('items', <String>['Earth', 'Moon', 'Sun']);
```

- Delete Data

```
// Remove data for the 'counter' key.
final success = await prefs.remove('counter');
```

- Read Data

```
// Obtain shared preferences.
final prefs = await SharedPreferences.getInstance();

// get an integer value from 'num' key.
await prefs.getInt('num');

// get an boolean value from 'flag' key.
await prefs.getBool('flag');

// get an double value from 'decnum' key.
await prefs.getDouble('decnum');

// get an String value from 'name' key.
await prefs.getString('name');

// get an list of strings from 'items' key.
await prefs.getStringList('items');
```

3.2. Sqlite Database

- If you are writing an app that needs to persist and query large amounts of data on the local device, consider using a database instead of a local file or key-value store.
- In general, databases provide faster inserts, updates, and queries compared to other local persistence solutions.
- Flutter apps can make use of the **SQLite databases** via the **sqflite** plugin available on pub.dev.
- This recipe demonstrates the basics of using sqflite to insert, read, update, and remove data about various Dogs.
- Add the dependencies

```
dependencies:
  flutter:
    sdk: flutter
  sqflite:
  path:
```

3.3. CRUD Operations in SQLite

- CRUD is short form of Create, Read, Update, Delete.
- Before creating the table to store information on Dogs, take a few moments to define the data that needs to be stored.
- For this example, define a Dog class that contains three pieces of data: A unique id, the name, and the age of each dog.
- **Open the database**
- Before reading and writing data to the database, open a connection to the database. This involves two steps:

- Define the path to the database file using getDatabasesPath() from the sqflite package, combined with the join function from the path package.
- Open the database with the openDatabase() function from sqflite.
-

```
// Avoid errors caused by flutter upgrade.
// Importing 'package:flutter/widgets.dart' is required.
WidgetsFlutterBinding.ensureInitialized();
// Open the database and store the reference.
final database = openDatabase(
// Set the path to the database. Note: Using the `join`
function from the
// `path` package is best practice to ensure the path is
correctly
  // constructed for each platform.
  join(await getDatabasesPath(), 'doggie_database.db'),
);
```

- **Define the Dog data model.**

```
class Dog {
  final int id;
  final String name;
  final int age;

  const Dog({
    required this.id,
    required this.name,
    required this.age,
  });
}
```

- **Create**
- Create the dogs table
- create a table to store information about various Dogs.
- For this example, create a table called dogs that defines the data that can be stored.
- Each Dog contains an id, name, and age.
- Therefore, these are represented as three columns in the dogs table.

```
final database = openDatabase(
  // Set the path to the database. Note: Using the `join` function
from the
  // `path` package is best practice to ensure the path is
correctly
  // constructed for each platform.
  join(await getDatabasesPath(), 'doggie_database.db'),
```

- **Insert**
- Insert a Dog data into the table
- First, insert a Dog data into the dogs table.
- This involves two steps:
- Convert the Dog into a Map
- Use the **insert()** method to store the Map in the dogs table.

```
Future<void> insertDog(id, name, age) async {
  // Get a reference to the database.
      final db = await database;

      Map<String, dynamic> map = {
       'id': id,
       'name': name,
       'age': age,
     };

  await db.insert(
      'dogs',
      map,
  );
}
```

- **Read**
- Retrieve the list from table
- Now that a Data is stored in the database, query the database for a specific data or a list of all data.

- Run a query against the dogs table. This returns a List<Map>.
- Convert the List<Map> into a List<Dog>.

```
Future<List<Dog>> dogs() async {
      final db = await database;
      final List<Map<String, dynamic>> maps = await
db.query('dogs');

      return List.generate(maps.length, (i) {
            return Dog(
                  id: maps[i]['id'],
                  name: maps[i]['name'],
                  age: maps[i]['age'],
                  );
            },
      );
}
```

- **Update**
- Update a Data in the database
- After inserting information into the database, need to update that information at a later time.
- can do this by using the update() method from the sqflite library.

1. Convert the Dog into a Map.
2. Use a where clause to ensure you update the correct Data.

```
Future<void> updateDog(id, name, age) async {
    final db = await database;
     Map<String, dynamic> map = {
          'id': id,
          'name': name,
          'age': age,
        };

    await db.update(
          'dogs',
          map,
          where: 'id = ?',whereArgs: [id],
     );
}
```

- **Delete**
- Delete a Record from the database.
- In addition to inserting and updating information.
- Can also remove dogs from the database.
- To delete data, use the delete() method from the sqflite library.
- Create a function that takes an id and deletes the record with a matching id from the database.

- To make this work, you must provide a where clause to limit the records being deleted.
-

```
Future<void> deleteDog(int id) async {
    final db = await database;

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

3.4. FutureBuilder

- Let's say you want to fetch some data from the backend on page launch and show a loader until data comes.
- FutureBuilder is a Widget that will help you to execute some asynchronous function and based on that function's result your UI will update.
- FutureBuilder is Stateful by nature i.e it maintains its own state as we do in StatefulWidgets.
- **Syntax**:
-

```
FutureBuilder(
  Key key,
  this.future,
  this.initialData,
  @required this.builder,
)
```

- builder should not be null.
- When we use the FutureBuilder widget we need to check for future state i.e future is resolved or not and so on. There is various State as follows:
- In future builder, it calls the future capacity to wait for the outcome, and when it creates the outcome it calls the builder function where we assemble the widget.
- **There are some parameters of FutureBuilderare:**
- **Key? key**: The widget's key, used to control how a widget is replaced with another widget.
- **Future<T>? future**: A Future whose snapshot can be accessed by the builder function.
- **T? initialData**: The data that will be used to create the snapshots until a non-null Future has completed.
- **required AsyncWidgetBuilder<T> builder**: The build strategy used by this builder.
- we need to make a Future to be passed as the future argument. In this article, we will utilize the capacity beneath, which returns Future<String>
-

```
Future<String> getData() async {
  await Future.delayed(Duration(seconds: 3));
  return 'Flutter Devs';
}
```

- **Example:**

```
FutureBuilder(
builder: (ctx, snapshot) {

      return Center(
      child: CircularProgressIndicator(),
      );
},
future: getData(),
),
```

- **Tasks for FutureBuilder:**
- Give the async task in **future** of Future Builder.
- Based on **connectionState,** show message (loading, active(streams), done).
- Based on **data(snapshot.hasError),** show view.
- To check whether the snapshot contains non-null information, you can utilize the hasData property.
- The asynchronous activity must be finished with non-null information altogether for the value to turn out to be valid.
- Nonetheless, if the asynchronous activity finishes without information (for example Future<void>), the value will be false.
- If the snapshot has data, you can acquire it by getting to the data property.

```
FutureBuilder<String>(
future: _value,
initialData: 'Demo Name',
builder: (
      BuildContext context,
      AsyncSnapshot<String> snapshot,
      ) {
      if (snapshot.connectionState == ConnectionState.waiting) {
            return Column(
                  crossAxisAlignment: CrossAxisAlignment.center,
                  mainAxisAlignment: MainAxisAlignment.center,
                  children: [
                        CircularProgressIndicator(),
                        Visibility(
                              visible: snapshot.hasData,
                              child: Text(
                                    snapshot.data,
                                    style: const TextStyle(color:
                                    Colors.black, fontSize: 24),
                                    ),
                              )
                        ],
                  );
      } else if (snapshot.connectionState == ConnectionState.done)
{
                  if (snapshot.hasError) {
                        return const Text('Error');
                  } else if (snapshot.hasData) {
                        return Text(snapshot.data,
                                    style: const TextStyle(color:
                                    Colors.cyan, fontSize: 36)
                              );
                  } else {
                        return const Text('Empty data');
                  }
            } else {
                  return Text('State: ${snapshot.connectionState}');
            }
      },
),
```

3.5. Future, Async, Await

- An object representing a delayed computation.
- The function is called, runs and returns it's result. Until then, the caller waits.
- Some functions, especially when they access resources like hardware or network, take a little time to do so.
- A Future can be in one of three states.
- **Uncompleted:** The output is closed.
- **Completed with value:** The output is open, and data is ready.
- **Completed with an error:** The output is open, but something went wrong.

- A Future is characterized precisely like a function in Dart, yet rather than Void, you utilize Future.
-

```
Future fetchUserOrder() {
      return Future.delayed(Duration(seconds: 2), () =>
      print('DATA'));
}
void main() {
      fetchUserOrder();
      print('Fetching user order..');
}
```

- There are two different ways to execute a Future and utilize the value it returns.
- The most well-known way is to **await** on the Future to return.
- For everything to fall into work, your function that is calling the code must be checked **async**.

```
Future getProductCostForUser() async {
      var user = await getUser();
      var order = await getOrder(user.uid);
      var product = await getProduct(order.productId);
      return product.totalCost;
}
main() async {
      var cost = await getProductCostForUser();
      print(cost);
}
```

- When an async function summons awaits, it is changed over into a Future and put into the execution line.
- At the point when the awaited Future is finished, the value is contained inside a Future object.
- This is achieved through a pattern known as **async - await.** It's not specific to flutter or dart, it exists under the same name in many other languages.
- **async - await** is just a declarative way of defining asynchronous functions and using their results into Future.
- If awaits will be utilized, we need to ensure that both the caller function and any capacities we call inside that function utilize the **async** modifier.

```
Future getProductCostForUser() async {
      var user = await getUser();
      var order = await getOrder(user.uid);
      var product = await getProduct(order.productId);
      return product.totalCost;
}
main() async {
      var cost = await getProductCostForUser();
      print(cost);
}
```

Unit-4
Accessing Rest API

4.1. Explain What is Rest Api.

- A REST API (also known as RESTful API) is an application programming interface.
- REST stands for representational state transfer.
- **Api:**
- An API is a set of definitions and protocols for building and integrating application software.
- It's sometimes referred to as a contract between an information provider and an information consumer.
- For example, the API design for a weather service could specify that the user supply a zip code and that the producer reply with a 2-part answer, the first being the high temperature, and the second being the low.
- **Rest API**
- REST is a set of architectural constraints, not a protocol or a standard.
- API developers can implement REST in a variety of ways.
- When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint.
- This information, or representation, is delivered in one of several formats via HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text.
- JSON is the most generally popular file format to use because, despite its name, it's language-agnostic, as well as readable by both humans and machines.

4.2. What is JSON? Explain JSON Parsing.

- JSON stands for JavaScript Object Notation.
- JSON is a lightweight format for storing and transporting data.
- JSON is often used when data is sent from a server to a web page.
- JSON is "self-describing" and easy to understand.
- The JSON format is syntactically identical to the code for creating JavaScript objects.
- JSON data is written as name/value pairs, just like JavaScript object properties.
- A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
"firstName":"John"
```

- **JSON Objects**
- JSON objects are written inside curly braces.
- Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName":"John", "lastName":"Doe"}
```

- **JSON Arrays**

- JSON arrays are written inside square brackets.
- Just like in JavaScript, an array can contain objects.
- In the example below, the object "employees" is an array. It contains three objects.

```
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
```

- **JSON Parsing**
- JSON is the standard for transferring data from one point to another point.
- If you query an API you're going to have to use JSON as response as String.
- We can easily parse JSON strings in Flutter.
- We can then use jsonEncode or jsonDecode to convert string to JSON.
- Above mentioned json can be parse as below.

```
Map<String,dynamic> jsonData = jsonDecode("{"firstName":"John",
"lastName":"Doe"}");
```

- When you decode json from string it will convert to Map<String,dynamic> from that object you can easily get data.

```
String name = jsonData["firstName"];

Output : John
```

4.3. How to call rest api from flutter project.

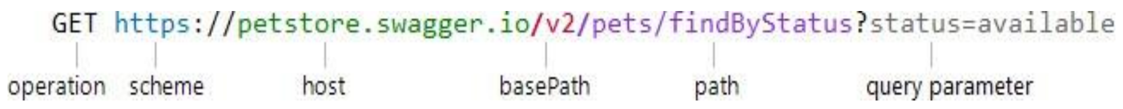
- A user application can make GET, POST, PUT, or DELETE HTTP requests to a database.
- In return, the database sends us data, results, or responses in the form of JSON, HTML, or XML (with the JSON format being the most widely used).
- We then parse the JSON into a proper model class and use it in our app.
- There are a few steps that we can follow to easily integrate an API into our Flutter app.
  - Get the API URL and endpoints.
  - Add relevant packages into the app (http, dio, chopper, etc.).
  - Create a constant file that stores URLs and endpoints.
  - Create a model class to parse the JSON.
  - Create a file that handles the API call, and write specific methods to fetch and parse data.
  - Use the data in your app.

1. Get the API URL and endpoints.

- The base URL is defined by schemes, host and basePath on the root level of the API specification.

```
host: petstore.swagger.io
basePath: /v2
schemes:
https
```

- cAll API paths are relative to this base URL, for example, /users actually means ://///users.


GET https://petstore.swagger.io/v2/pets/findByStatus?status=available
operation
scheme
host
basePath
path
query parameter

- In above url https://perstore.swagger.io/v2 is base url & **/pets/findByStatus** is end points.

2. Add relevant packages into the app (http, dio, chopper, etc.).

- There are many packages available on pub.dev that we can use to integrate APIs in Flutter. The most widely used packages are http, dio, chopper.
- There are many more packages, though http is the most basic and easy to use.
- Add the http package into it. Your pubspec.yaml file.

```
dependencies:
  cupertino_icons: ^1.0.2
  flutter:
    sdk: flutter
  http: ^0.13.4
```

3. Create a constant file that stores URLs and endpoints.

- Now, it's time to create a simple file named constants.dart that will hold all our URLs and endpoints.

```
class ApiConstants {
 static String baseUrl ='https://jsonplaceholder.typicode.com';
 static String usersEndpoint = '/users';
}
```

- Here, we have created a class called ApiConstants and two static variables.
- We can access them without creating an instance of the class like ApiConstants.baseUrl.

4. Create a model class to parse the JSON.

- One way to access the data is by using the key directly.
- However, it's an effective and easy approach to create a model class, parse the JSON, and get an object out of that JSON response.
- Use this to access the data directly.

```
final data = json[0]['id'];
```

5. **Create a file that handles the API call, and write specific methods to fetch and parse data.**

- create a file called api_service.dart that will handle the API calls.
- we create a function called getUsers that returns a List.
- The first step is to hit the **GET HTTP** request.
- The next step is to check whether the API call was successful or not using response.statusCode.
- If the API call is successful, the statusCode will be 200.

```
import 'dart:developer';

import 'package:http/http.dart' as http;
import 'package:rest_api_example/constants.dart';
import 'package:rest_api_example/model/user_model.dart';

class ApiService {
  Future<List<UserModel>?> getUsers() async {
  try {
  var url = Uri.parse(ApiConstants.baseUrl +
ApiConstants.usersEndpoint);
var response = await http.get(url); if
(response.statusCode == 200) {
List<UserModel> _model = userModelFromJson(response.body);
return _model;
}
  } catch (e) {
log(e.toString());
  }
 }
}
```

6. **Use the data in your app.**

- We have created all the required files and methods for the API, which we can call from our app's back end.
- just make a method call and load that result onto the UI.
- we create an object of the type List.

```
@override
void initState() {
super.initState();
_getData();
}

void _getData() async {
_userModel = (await ApiService().getUsers())!;
Future.delayed(const Duration(seconds: 1)).then((value) =>
setState(() {}));
}

@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
 title: const Text('REST API Example'),
),
    body: _userModel == null || _userModel!.isEmpty
    ? const Center(
    child: CircularProgressIndicator(),
    )
    : ListView.builder(
     itemCount: _userModel!.length, itemBuilder: (context, index)
     {
          return Card( child: Column(
               children: [ Row(
                    mainAxisAlignment:
                    MainAxisAlignment.spaceEvenly,
                         children: [
                         Text(_userModel![index].id.toString()),
                         Text(_userModel![index].username),
                         ],
                    ),
          const SizedBox( height: 20.0,),
          Row(
               mainAxisAlignment: MainAxisAlignment.spaceEvenly,
               children: [
                    Text(_userModel![index].email),
                    Text(_userModel![index].website),
               ],
               ),
          ],
          ),
          );
     },
     ),
    );
```

Unit-5
Advance Application Development

5.1. Requesting Permission at run time

- Developers can ask the user for permissions while the app is running using the library permission_handler.
- This plugin provides a cross-platform (iOS, Android) API to request permissions and check status.
- You can also open the device&#39;s app settings so users can grant a permission.
- While the permissions are being requested during runtime, you'll still need to tell the OS which permissions your app might potentially use.
- That requires adding permission configuration to Android- and iOS-specific files.
    - Androids
    - iOS
- Add library in your project.
- Add a line like this to your package's pubspec.yaml

```
dependencies:
      permission_handler: ^10.0.2
```

- Now in your Dart code, you can use:

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

- There are a number of Permissions. You can get a Permission status, which is either granted, denied, restricted or permanentlyDenied.

```
var status = await Permission.camera.status; if
(status.isDenied) {
// The permission has been denied before but not permanently.
}
// You can also directly ask permission about its status.
if (await Permission.location.isRestricted) {
// The OS restricts access, for example because of parental controls.
}
```

- Call request() on a Permission to request it.
- If it has already been granted before, nothing happens.
- request() returns the new status of the Permission.

```
if (await Permission.contacts.request().isGranted) {
// Permission was already granted before or just granted
}
//  You  can  request  multiple  permissions  at  once.
Map<Permission,  PermissionStatus>  statuses  =  await  [
Permission.location,
Permission.storage,
].request();
print(statuses[Permission.location]);
```

5.2. Work with SD Card and Files

- To save files to disk, combine the path_provider plugin with the dart:io library. Follow the steps:
- Find the correct local path.
  - The path_provider package provides a platform-agnostic way to access commonly used.
  - locations on the device’s file system.
  - The plugin currently supports access to two file system locations:
- Temporary directory
  - A temporary directory (cache) that the system can clear at any time.
  - On iOS, this corresponds to the NSCachesDirectory.
  - On Android, this is the value that getCacheDir() returns.
- Documents directory
  - A directory for the app to store files that only it can access.
  - The system clears the directory only when the app is deleted.
  - On iOS, this corresponds to the NSDocumentDirectory.
  - On Android, this is the AppData directory.
- Can find the path to the documents directory as follows:

```
Future<String> get_localPath async {
      final directory = await getApplicationDocumentsDirectory();
      return directory.path;
}
```

- Create a reference to the file location
  - Once you know where to store the file, create a reference to the file’s full location.
  - You can use the File class from the dart:io library to achieve this.

```
Future<File>   get_localFile   async   {
      final  path  =  await  _localPath;
      return File('$path/counter.txt');
}
```

- Write data to the file

  - Now that you have a File to work with, use it to read and write data.
  - First, write some data to the file.
  - The counter is an integer, but is written to the file as a string using the '$counter' syntax.

```
Future<File> writeCounter(int counter) async {
      final file = await _localFile;
      // Write the file
      return file.writeAsString('$counter');
}
```

- Read data from the file
  - Now that you have some data on disk, you can read it.
  - Once again, use the File class.

```
Future<int>  readCounter()  async  {
      try {
            final file = await _localFile;
            // Read the file
            final contents = await file.readAsString();
            return int.parse(contents);
     } catch (e) {
            // If encountering an error, return 0
            return 0;
     }
}
```

5.3. Working with Camera

- Many apps require working with the device's cameras to take photos and videos.
- Flutter provides the camera plugin for this purpose.
- The camera plugin provides tools to get a list of the available cameras, display a preview coming from a specific camera, and take photos or videos.
- The following demonstrates how to use the camera plugin to display a preview, take a photo, and display it:
- Add the required dependencies
  - You need to add three dependencies to your app:

Dependency Name	Description
Camera	Provides tools to work with the cameras on the device.
path_provider	Finds the correct paths to store images.
Path	Creates paths that work on any platform.

- Your pubspec.yaml will now contains:

```
dependencies:
  flutter:
    sdk:      flutter
  camera:
  path_provider:
  path:
```

- Get a list of the available cameras.
- Next, get a list of available cameras using the camera plugin.

```
// Ensure that plugin services are initialized so that
'availableCameras()' can be called before 'runApp()'
WidgetsFlutterBinding.ensureInitialized();
// Obtain a list of the available cameras on the device.
final cameras = await availableCameras();
// Get a specific camera from the list of available cameras. final
firstCamera = cameras.first;
```

- Create and initialize the CameraController
  - Once you have a camera, use the following steps to create and initialize a CameraController.
  - This process establishes a connection to the device's camera that allows you to control the camera and display a preview of the camera's feed.
  - Create a StatefulWidget with a companion State class.
  - Add a variable to the State class to store the CameraController.
  - Add a variable to the State class to store the Future returned from CameraController.initialize().
  - Create and initialize the controller in the initState() method.
  - Dispose of the controller in the dispose() method.

```
// Screen allows users to take a picture using a given camera.
class TakePictureScreen extends StatefulWidget {
      const          TakePictureScreen({
            super.key,
            required this.camera,
      });

      final  CameraDescription  camera;
      @override
      TakePictureScreenState            createState()            =>
      TakePictureScreenState();
}

class  TakePictureScreenState  extends  State<TakePictureScreen>  {
      late CameraController _controller;
      late     Future<void>     _initializeControllerFuture;
      @override
      void initState() {
            super.initState();
            _controller     =     CameraController(
                  widget.camera,
                  ResolutionPreset.medium,
            );
            _initializeControllerFuture = _controller.initialize();
      }

      @override
      void dispose() {
            _controller.dispose();
            super.dispose();
      }

      @override
      Widget   build(BuildContext   context)   {
            return Container();
      }
}
```

- Use a CameraPreview to display the camera’s feed.
- Use the CameraPreview widget from the camera package to display a preview of the camera’s feed.

```
FutureBuilder<void>(
      future:_initializeControllerFuture,
      builder: (context, snapshot) {
            if  (snapshot.connectionState  ==  ConnectionState.done)  {
                  return CameraPreview(_controller);
            } else {
                  return const Center(child: CircularProgressIndicator());
            }
      },
)
```

- Take a picture with the CameraController
  - You can use the CameraController to take pictures using the takePicture() method, which returns an XFile, a cross-platform, simplified File abstraction.
  - On both Android and IOS, the new image is stored in their respective cache directories, and the path to that location is returned in the XFile.
  - Taking a picture requires 2 steps:
    1. Ensure that the camera is initialized.
    2. Use the controller to take a picture and ensure that it returns a Future<XFile>.
  - It is good practice to wrap these operations in a try / catch block in order to handle any errors that might occur.

```
FloatingActionButton(
      try {
            await _initializeControllerFuture;
            final image = await _controller.takePicture();
      } catch (e) {
            print(e);
      },
      child: const Icon(Icons.camera_alt),
)
```

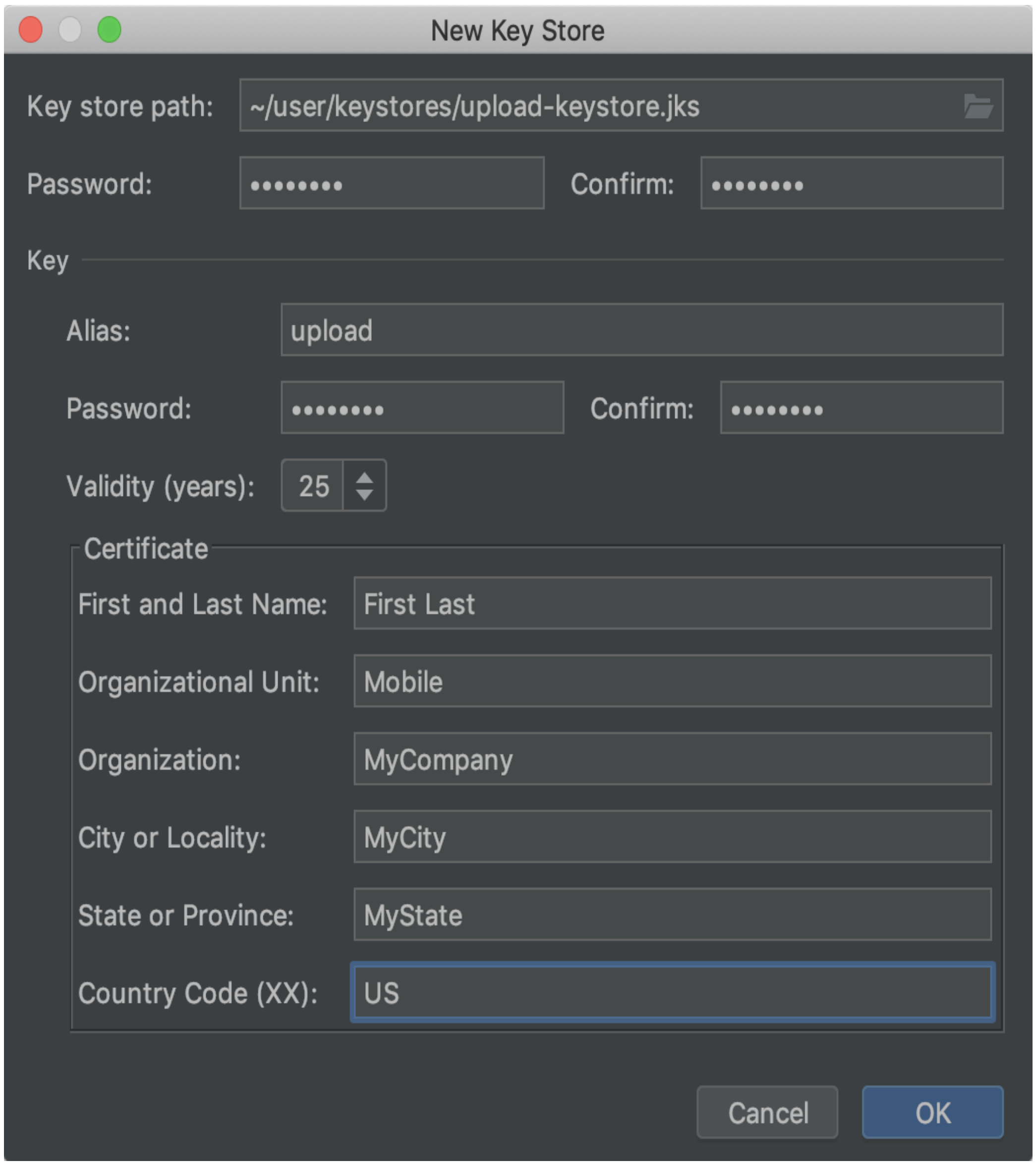
- Display the picture with an Image widget
  - If you take the picture successfully, you can then display the saved picture using an Image widget.
  - In this case, the picture is stored as a file on the device.
  - Therefore, you must provide a File to the Image.file constructor.
  - You can create an instance of the File class by passing the path created in the previous step.

```
Image.file(File('path/to/my/picture.png'));
```

5.4. Generate and upload key and keystore

- If you don't already have an upload key, which is useful when configuring Play App Signing, you can generate one using Android Studio as follows:

1. In the menu bar, click Build > Generate Signed Bundle/APK.
2. In the Generate Signed Bundle or APK dialog, select Android App Bundle or APK and click Next.
3. Below the field for Key store path, click Create new.
4. On the New Key Store window, provide the following information for your keystore and key, as shown in figure.



New Key Store

Key store path:	~/user/keystores/upload-keystore.jks		
Password:	••••••••	Confirm:	••••••••

Key

Alias:	upload		
Password:	••••••••	Confirm:	••••••••
Validity (years):	25		

Certificate

First and Last Name:	First Last
Organizational Unit:	Mobile
Organization:	MyCompany
City or Locality:	MyCity
State or Province:	MyState
Country Code (XX):	US

5. Keystore
   - **Key store path:** Select the location where your keystore should be created. Also, a file name should be added to the end of the location path with the .jks extension.
   - **Password:** Create and confirm a secure password for your keystore.

6. Key
   - **Alias:** Enter an identifying name for your key.
   - **Password:** Create and confirm a secure password for your key. This should be the same as your keystore password.(Please refer to the known issue for more information).
   - **Validity (years):** Set the length of time in years that your key will be valid. Your key should be valid for at least 25 years, so you can sign app updates with the same key through the lifespan of your app.
   - **Certificate:** Enter some information about yourself for your certificate. This information is not displayed in your app, but is included in your certificate as part of the APK.
7. Once you complete the form, click OK.
8. If you would like to build and sign your app with your upload key, continue to the section about how to Sign your app with your upload key. If you only want to generate the key and keystore, click Cancel.

5.5. How can you publish your application in Google Play Store? Explain the entire process.

- After you've created your Google Play developer account, you can create apps and set them up using Play Console. Open Play Console.
- Select All apps > Create app.
- Select a default language and add the name of your app as you want it to appear on Google Play. You can change this later.
- Specify whether your application is an app or a game. You can change this later.
- Specify whether your application is free or paid.
- Add an email address that Play Store users can use to contact you about this application.
- In the "Declarations" section:
  - Acknowledge the "Developer Program Policies" and "US export laws" declarations.
  - Accept the Play App Signing Terms of Service.
- Select **Create app**.
- Set up your app.
- Manage your app and app bundle.
- Set up your app store listing and settings.
  - Complete app details like App name, short description, and full description.
  - Select an app category and apply tags to your app.

    - Provide contact information.
    - Add screenshots and a video that showcases your app.
- Release your app
    - Set up internal or closed tests.
    - Prepare and roll out your release to the test track.
    - Invite testers via email.
    - You can even directly release your app to the production track, if you have already tested the app with your team.