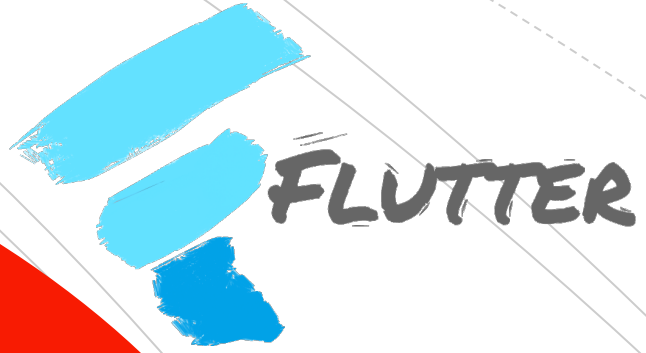


Flutter From Zero to Hero

SITTIPONG JANSORN

Setup and Introduction

- What is Flutter?
- Flutter Architecture
- Setup Development Tools
- Flutter CLI
- Project Structure

Flutter

Dart programming language

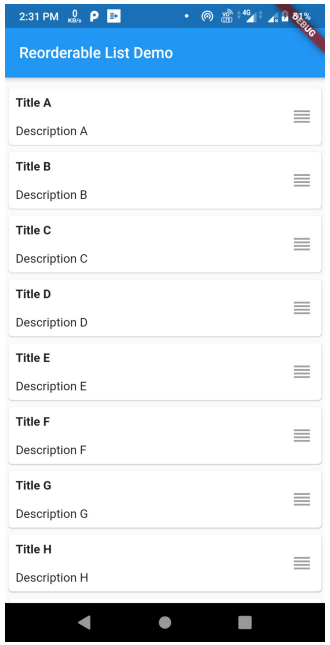
- Introduction
- Variables
- Functions
- Operators
- Control Flow
- Handle Exceptions
- Classes

Layout and Widget

- Understanding layouts
- Stateless Widget
- Stateful Widget
- Life cycle
- Scaffold
- AppBar
- Widget
  - Layout
  - Form
- Dialog

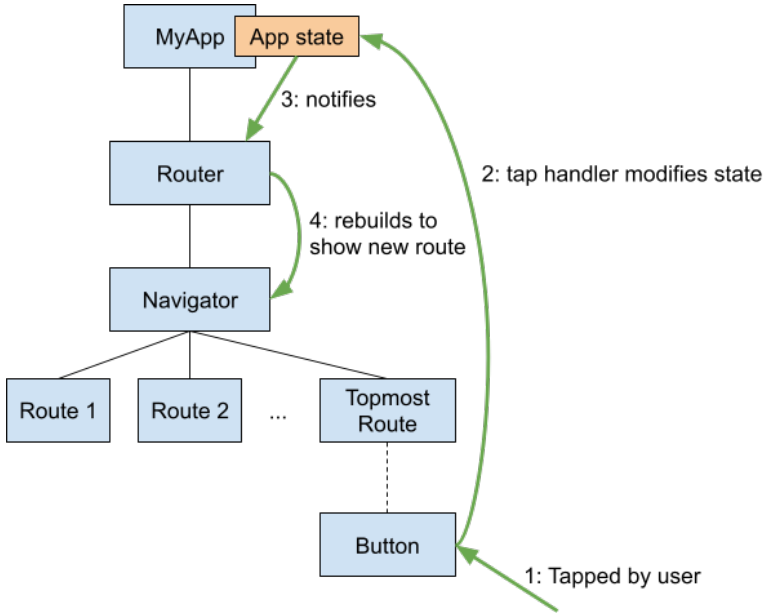
Using List Layout

- Creating Scrollable Lists
- Custom Layout
- Custom Style
- Grid



Navigator

- Push, Pop
- PushNamed
- PushReplacementNamed
- PopAndPushNamed
- PushNamedAndRemoveUntil
- PopUntil



Basic State Management

- What is state management
- State management framework

Login & Register Workshop

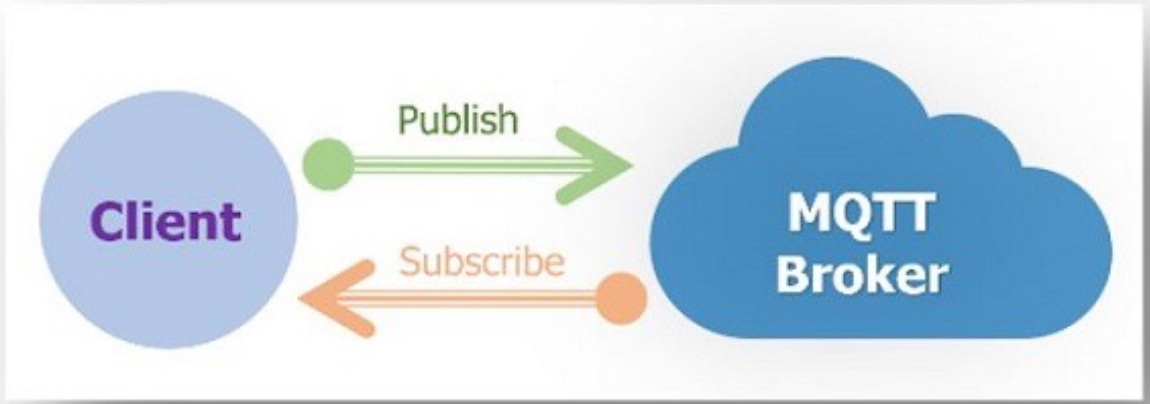
- Authentication
- Handle Form
- Deep Dive Into TextField
- Form validation
- Shared preference
- Login-Logout
- Send data between page

JSON Restful Feed Workshop

- The HTTP Package
- Http method (GET/POST/PUT/DELETE/..)
- Binding JSON
- Parsing complex JSON
- Displaying a Loading Spinner
- Login-Logout
- Send data between page

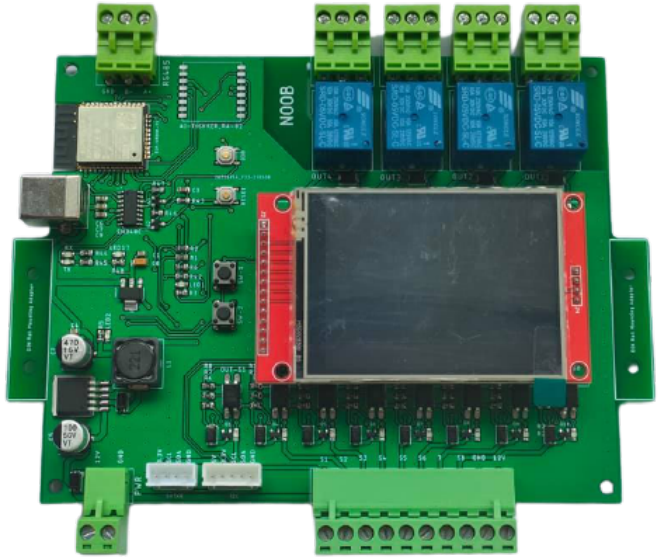
Realtime Data By MQTT Workshop

- MQTT Package
- MQTT Client instant
- Connection state
- Publish
- Subscribe



Hardware Control By MQTT Workshop

- Json API
  - Arduino
  - MicroPython
- Arduino Sample
- MicroPython Aample



Publishing App

- App Icon
- State management framework
- Resources Management
- Build APK
- Deploy Real device (Android)
- Publishing to Google Play

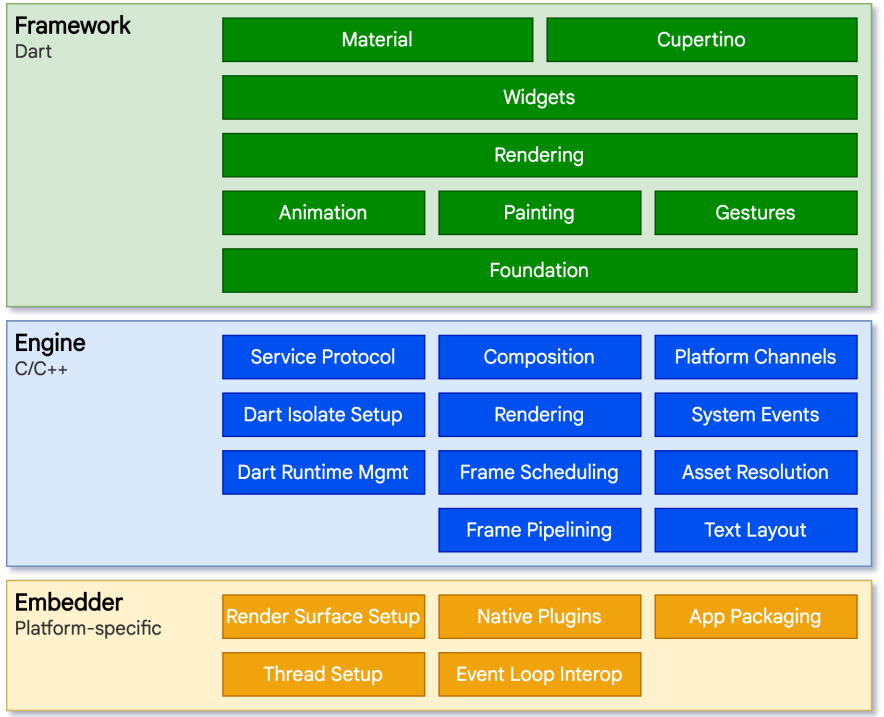
What is Flutter?

Flutter is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, desktop, and embedded devices from a single codebase.

- Fast Development
  - Paint your app to life in milliseconds with Stateful Hot Reload. Use a rich set of fully-customizable widgets to build native interfaces in minutes.
- Expressive and Flexible UI
  - Quickly ship features with a focus on native end-user experiences. Layered architecture allows for full customization, which results in incredibly fast rendering and expressive and flexible designs.
- Native Performance
  - Flutter’s widgets incorporate all critical platform differences such as scrolling, navigation, icons and fonts, and your Flutter code is compiled to native ARM machine code using Dart's native compilers.

Flutter architectural overview

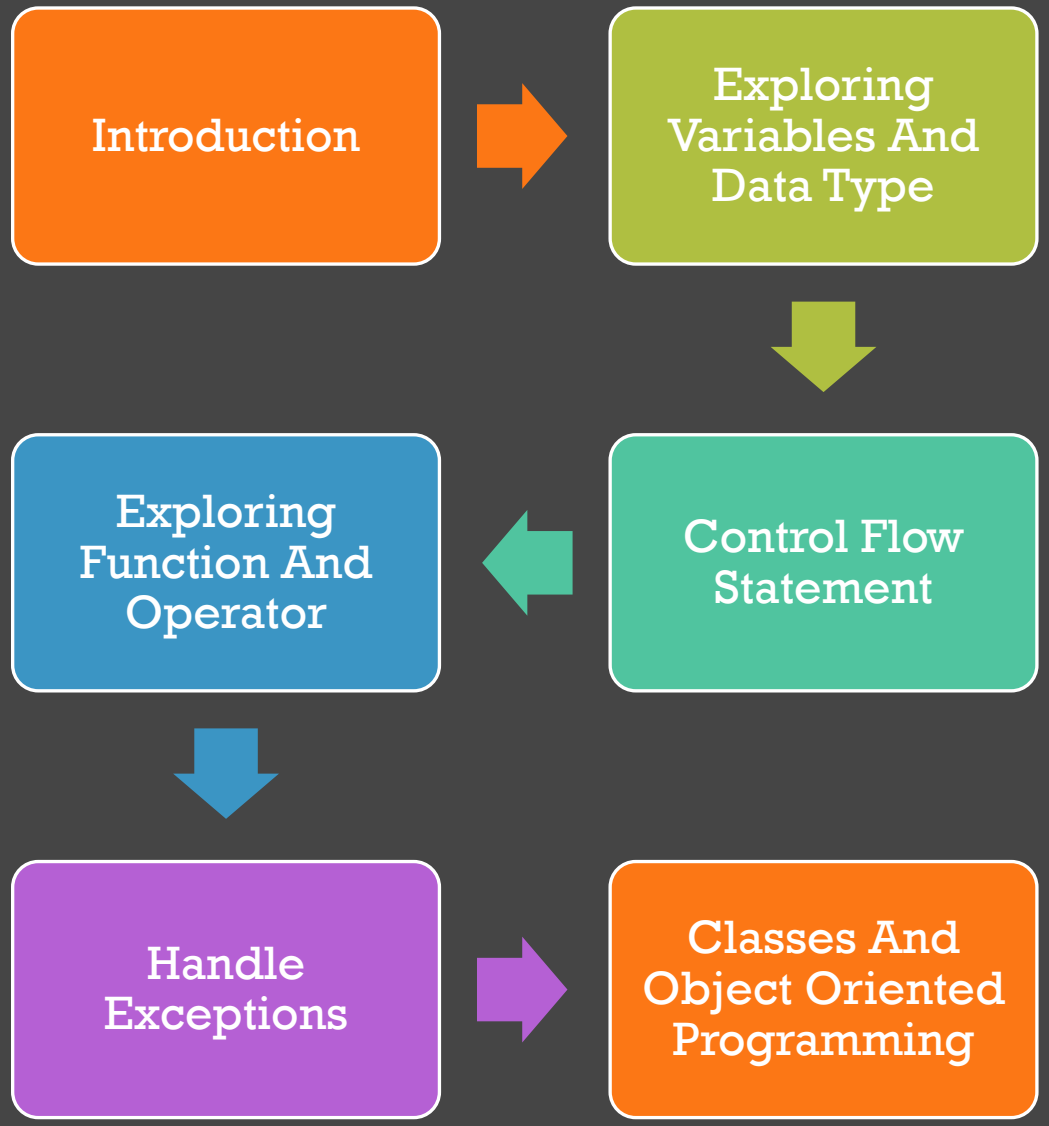
- The **layer model**: The pieces from which Flutter is constructed.
- **Reactive user interfaces**: A core concept for Flutter user interface development.
- An introduction to **widgets**: The fundamental building blocks of Flutter user interfaces.
- The **rendering process**: How Flutter turns UI code into pixels.
- An overview of the **platform embedders**: The code that lets mobile and desktop OSes execute Flutter apps.
- **Integrating Flutter with other code**: Information about different techniques available to Flutter apps.
- **Support for the web**: Concluding remarks about the characteristics of Flutter in a browser environment.



Setup and Introduction

- Android Studio
  - https://bit.ly/38fWWt8
- Visual Studio Code
  - https://bit.ly/3my5mEG
- Flutter SDK
  - https://bit.ly/3gFf2JH

Dart programming language



Introduction

- Effective Dart: Style
  - A surprisingly important part of good code is good style. Consistent naming, ordering, and formatting helps code that *is* the same *look* the same. It takes advantage of the powerful pattern-matching hardware most of us have in our ocular systems. If we use a consistent style across the entire Dart ecosystem, it makes it easier for all of us to learn from and contribute to each others' code.

Identifiers

- **UpperCamelCase** names capitalize the first letter of each word, including the first.
- **lowerCamelCase** names capitalize the first letter of each word, *except* the first which is always lowercase, even if it's an acronym.
- **lowercase_with_underscores** names use only lowercase letters, even for acronyms, and separate words with _

UpperCamelCase

- Classes, enum types, typedefs, and type parameters should capitalize the first letter of each word (including the first word), and use no separators.

```
good
   class SliderMenu { ... }

   class HttpRequest { ... }

   typedef Predicate<T> = bool Function(T value);
```

lowercase_with_underscores

- Some file systems are not case-sensitive, so many projects require filenames to be all lowercase. Using a separating character allows names to still be readable in that form. Using underscores as the separator ensures that the name is still a valid Dart identifier, which may be helpful if the language later supports symbolic imports

good

```
library peg_parser.source_scanner;

import 'file_system.dart';
import 'slider_menu.dart';
```

bad

```
library pegparser.SourceScanner;

import 'file-system.dart';
import 'SliderMenu.dart';
```

lowercase_with_underscores

- name import prefixes

good

```
import 'dart:math' as math;
import 'package:angular_components/angular_components'
    as angular_components;
import 'package:js/js.dart' as js;
```

bad

```
import 'dart:math' as Math;
import 'package:angular_components/angular_components'
    as angularComponents;
import 'package:js/js.dart' as JS;
```

lowerCamelCase

- name other identifiers
  - Class members, top-level definitions, variables, parameters, and named parameters should capitalize the first letter of each word *except* the first word, and use no separators

good

```
const pi = 3.14;
const defaultTimeout = 1000;
final urlScheme = RegExp('^([a-z]+):');

class Dice {
  static final numberGenerator = Random();
}
```

bad

```
const PI = 3.14;
const DefaultTimeout = 1000;
final URL_SCHEME = RegExp('^([a-z]+):');

class Dice {
  static final NUMBER_GENERATOR = Random();
}
```

Exploring Variables And Data Type

- Now it is time to question what are Dart data types and variables. In this lesson we will cover the concept of dart data types and variables . We will learn how to declare variables in Dart and then we will focus on what are the various data types available in Dart.

Data Types in Dart

- Dart provide us various built in data types in Dart.
  - Numbers
    - int
    - Double
- Like other languages (C, C++, Java), whenever a variable is created, each variable has an associated data type. In Dart language, there is the type of values that can be represented and manipulated in a programming language. The data type classification is as given below.
  - Strings
  - Booleans
  - Lists
  - Maps

Dart Variable Declaration

- รูปแบบการประกาศค่าตัวแปรในภาษา **Dart** ก็เหมือนในกับภาษาอื่น ๆ ทั่ว ๆ ไปตามรูปแบบข้างล่างนี้

```
String firstName = 'Sittipong';
```

  - **String** is the Data Type
  - **firstName** is the Variable name
  - **Sittipong** is the Value

var

- การใช้ **keyword *var*** ในการประกาศค่าตัวแปรนั้น ประเภทข้อมูลของตัวแปรนั้นจะขึ้นอยู่กับค่าที่เรากำหนดให้
- กำหนดประเภทตัวแปรเป็น **String**
- กำหนดค่าใหม่เป็น **Integer**
  - **Error: A value of type 'int' can't be assigned to a variable of type 'String'**

```
var y = 'Hello World';

y = 100;
```

dynamic

- การใช้ **keyword *dynamic*** จะทำให้เราใช้ตัวแปรที่เราประกาศขึ้นมารับค่าของข้อมูลเป็นอะไรก็ได้

```
dynamic x = 'Hello World';
x = 100;
x = 2.5;
x = [1, 2, 3, 4];
```

const and final

- ในบางครั้งเมื่อเราประกาศตัวแปรขึ้นมาแล้ว แต่ไม่ต้องการให้มีการเปลี่ยนแปลงค่าในภายหลัง เราสามารถใช้ **const** หรือ **final** นำหน้าประเภทข้อมูล หรือ **var, dynamic**
- ความแตกต่างในความเหมือนระหว่าง **const** และ **final**
  - **const** เมื่อกำหนดค่าให้แล้ว จะไม่สามารถเปลี่ยนแปลงค่าได้ และต้องกำหนดค่าให้ทันที่ที่ประกาศตัวแปล
  - **final** เมื่อกำหนดค่าให้แล้ว จะไม่สามารถเปลี่ยนแปลงค่าได้ จะกำหนดค่าให้ทันทีหรือภายหลังก็ได้ แต่จะกำหนดได้แค่ครั้งแรกครั้งเดียว

Number

- ข้อมูลแบบตัวเลขในภาษา **Dart** จะแบ่งเป็น **2** กลุ่ม
  - **Integer** เป็นข้อมูลตัวเลขแบบจำนวนเต็ม เช่น100
  - **Decimal** เป็นข้อมูลตัวเลขแบบ ทศนิยม เช่น 100.25

String

- เป็นประเภทข้อมูลที่ใช้เก็บค่าของข้อความ

```dart
void main() {
  String firstName = 'Sittipong';
  print(firstName);
}
```

Boolean

- ชนิดข้อมูลแบบ **Boolean** เป็นการเก็บข้อมูลที่เป็นจริง **(false)** หรือเป็นเท็จ**(false)**

```
1 void main() {
2   bool isEMployee = true;
3   print(isEMployee);
4 }
```

List

- รายการประเภทข้อมูลที่ใช้เพื่อแสดง **collection** ของ **object** รายการเป็นกลุ่มของ **object** ที่เรียงลำดับ ชนิดข้อมูล **List** ในภาษา **Dart** มีความหมายเหมือนกันกับแนวคิดของ **Array** ในภาษาโปรแกรมอื่น ๆ

```dart
void main() {
  List vals = [2, 4, 6, 8, 10] ;
  print(vals);
}
```

List Methods

- add(E value) → void
- addAll(Iterable<E> iterable) → void
- *any*(bool test(E element)) → bool
- clear() → void
- *contains*(Object? element) → bool
- *elementAt*(int index) → E
- *forEach*(void f(E element)) → void
- indexOf(E element, [int start = 0]) → int
- insert(int index, E element) → void
- insertAll(int index, Iterable<E> iterable) → void

List Methods

- remove(Object? value) → bool
- removeAt(int index) → E
- removeLast() → E
- removeRange(int start, int end) → void
- *where*(bool test(E element)) → Iterable<E>

Map

- รายการประเภทข้อมูลที่ใช้เพื่อแสดง **collection** ของ **object** ที่อยู่ในรูปแบบของ **key** กับ **value** ในการเข้าถึงข้อมูลจะใช้ **key** ในการอ้างอิง ซึ่งจะเหมือนกับข้อมูลประเภท **JSON**

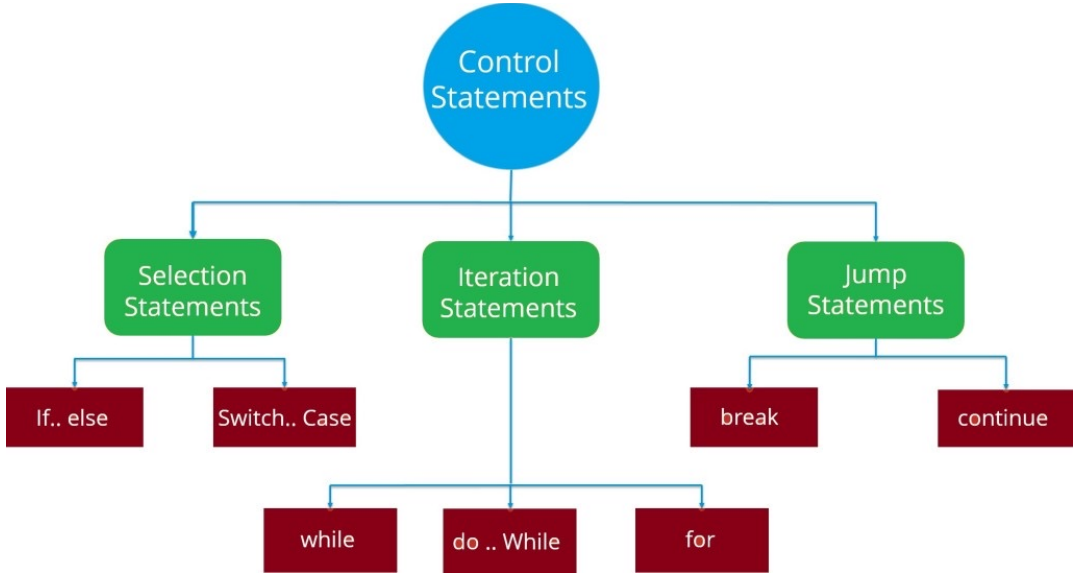
```
1 void main() {
2   Map vals = {'x': 1, 'y': 2} ;
3   print(vals['y']);
4 }
```

Map Methods

- addAll(Map<K, V> other) → void
- clear() → void
- containsKey(Object? key) → bool
- containsValue(Object? value) → bool
- forEach(void action(K key, V value)) → void
- remove(Object? key) → V?
- removeWhere(bool test(K key, V value)) → void

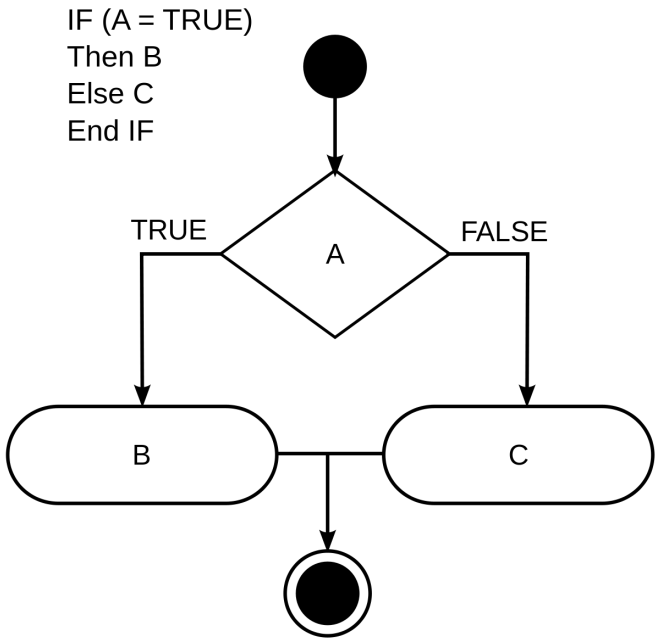
Control Flow Statement

- โฟลว์การควบคุมหรือโฟลว์ของการควบคุมคือลำดับที่คำสั่ง คำสั่ง และการเรียกใช้ฟังก์ชันถูกดำเนินการหรือประเมินผลเมื่อโปรแกรมกำลังทำงาน คำสั่งควบคุมการไหลเรียกอีกอย่างว่าคำสั่งควบคุมการไหล ใน **Dart** คำสั่งในโค้ดของเราโดยทั่วไปจะดำเนินการตามลำดับจากบนลง ชุดคำสั่งบางชุดตามเงื่อนไข ข้ามไปยังคำสั่งอื่น หรือดำเนินการชุดคำสั่งซ้ำ ๆ ใน **Dart** คำสั่งควบคุมโฟลว์ถูกใช้เพื่อแก้ไข เปลี่ยนเส้นทาง หรือเพื่อควบคุมโฟลว์ของการทำงานของโปรแกรมตามตรรกะของแอปพลิเคชัน



Selection statements

- if Statement
- If else Statement
- if else if Statement
- Switch Case Statement



if Statement

- สำหรับตรวจสอบเงื่อนไขว่าเป็นจริงหรือเป็นเท็จ

```
if(condition){
  // statements
}
```

```
int n = 30;
if (n < 100) {
  print('n < 100');
}
```

If else Statement

- สำหรับตรวจสอบเงื่อนไขว่าถ้าเป็นจริงให้ทำงานอย่างหนึ่งถ้าไม่ใช่ให้ทำงานอีกอย่างหนึ่ง

```
if(condition){
  // statements
} else {
  // statements
}
```

```
int n = 100;
if (n < 100) {
  print('n < 100');
} else {
  print('n >= 100');
}
```

if else if Statement

- สำหรับตรวจสอบเงื่อนไขหลาย ๆ เงื่อนไข

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

```
int n = 20;
if (n <= 50) {
  print('Low');
} else if (n > 50 && n <= 100) {
  print('Medium ');
} else {
  print('Heigh ');
}
```

Switch Case Statement

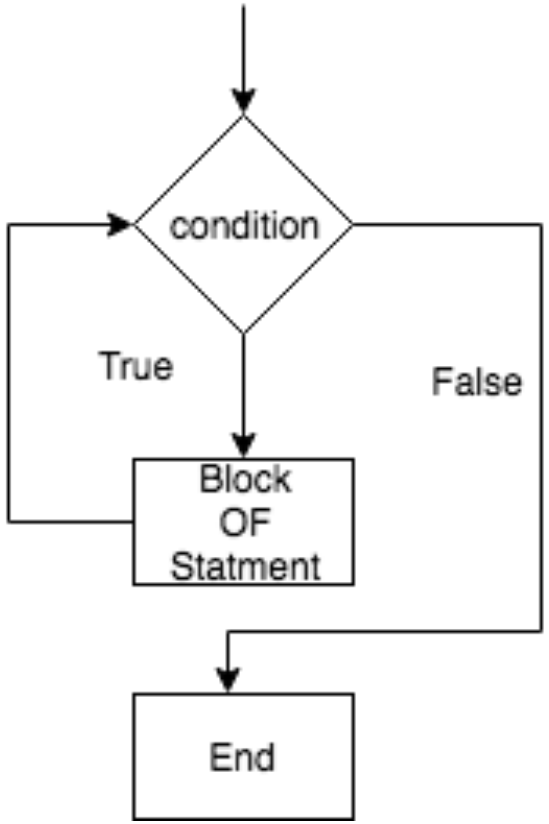
- สำหรับตรวจสอบเงื่อนไขหลาย ๆ เงื่อนไข

```
switch(expression){
case value1: {
// statements
}
      break;
case value2: {
// statements
}
      break;
}
default: {
// statements
}
      break;
}
```

```
int n = 2;
switch (n) {
  case 1: print('Low');
    break;
  case 2: print('Medium');
    break;
  case 3: print('Heigh');
    break;
  default: print('Unknow');
    break;
}
```

Iteration statements

- for loop
- for…in loop
- while loop
- do while loop



for loop

- **for loop**
  - คือการวนลูปที่แน่นอนมาใช้ **for loop** ดำเนินการบล็อกของโค้ดตามจำนวนครั้งที่ระบุ
- รูปแบบของคำสั่ง

```
for (variablename in object){
   statement or block to execute
}
```

```
void main() {
   var num = 5;
   var factorial = 1;

   for( var i = num ; i >= 1; i-- ) {
      factorial *= i ;
      print(factorial);
   }
}
```

for.. in loop

- for...in loop
  - ใช้เพื่อวนรอบคุณสมบัติของอ็อบเจ็กต์
- รูปแบบของคำสั่ง

```
for (variablename in object){
   statement or block to execute
}
```

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

while loop

- **while loop**
  - รันคำสั่งทุกครั้งที่เงื่อนไขที่ระบุเป็นจริง กล่าวอีกนัยหนึ่ง ลูปจะประเมินเงื่อนไขก่อนที่จะดำเนินการบล็อกของโค้ด
- รูปแบบคำสั่ง

```
while (expression) {
   Statement(s) to be executed if expression is true
}
```

```
void main() {
  var num = 5;
  var factorial = 1;
  while(num >=1) {
    factorial = factorial * num;
    num--;
  }
  print("The factorial  is ${factorial}");
}
```

do while loop

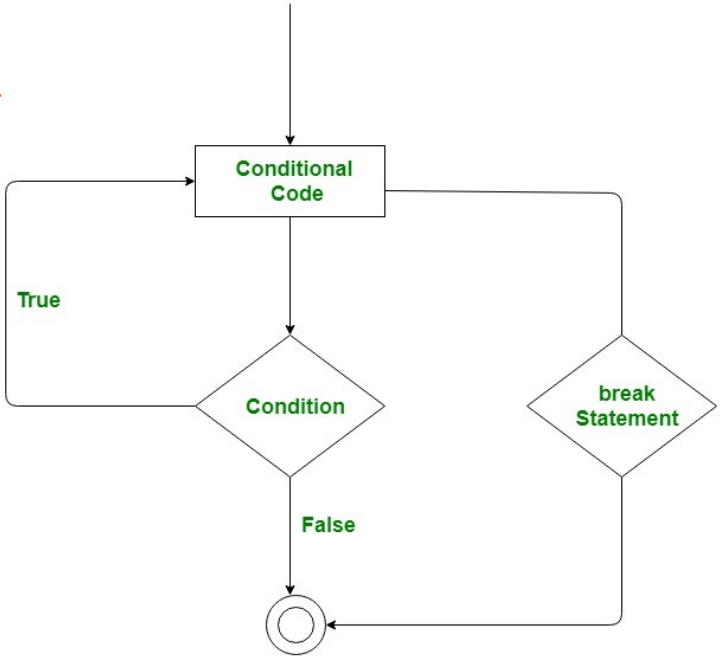
- **do .. while loop**
  - คล้ายกับ **while loop** ยกเว้น **do...while loop** จะไม่ประเมินเงื่อนไขในครั้งแรกที่ **loop** ดำเนินการ เงื่อนไขจะได้รับการประเมินสำหรับการทำซ้ำในภายหลัง กล่าวคือบล็อกโค้ดจะถูกดำเนินการอย่างน้อยหนึ่งครั้งใน **do...while loop**
- รูปแบบของคำสั่ง

```
do {
   Statement(s) to be executed;
} while (expression);
```

```
void main() {
  var n = 10;
  do {
    print(n);
    n--;
  }
  while(n>=0);
}
```

Jump statements

- Break Statement
- Continue Statement



Break Statement

- **Break;**
  - คำสั่งนี้ใช้เพื่อการหยุดการควบคุมคำสั่งของลูป เมื่อเข้าเงื่อนไขทีเรากำหนด เช่น หากใช้ภายในลูป มันจะยุติลูปทุกครั้งที่พบ

```
void main() {
   int count = 1;

   while (count <= 10) {
      print("Geek, you are inside loop $count");
      count++;
      if (count == 4) {
        break;
      }
   }
   print("Geek, you are out of while loop");
}
```

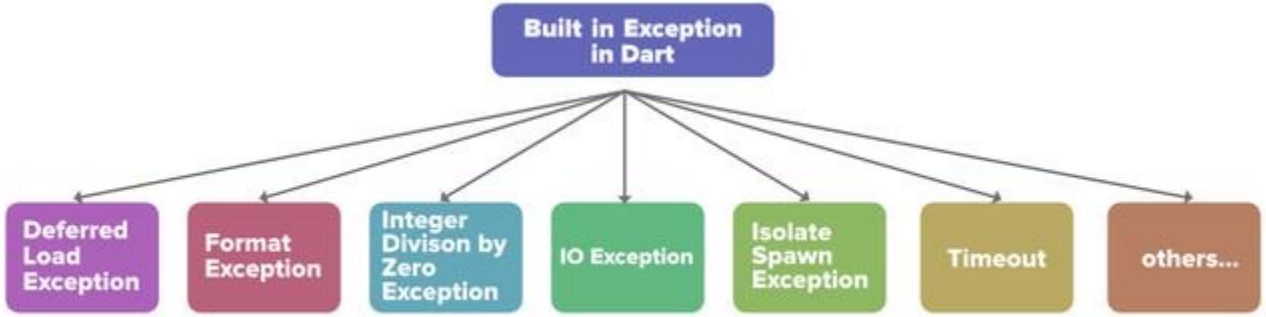
Continue Statement

- **continue**
  - ในขณะที่ **break** ใช้เพื่อสิ้นสุดการไหลของการควบคุม ในทางกลับกัน จะใช้เพื่อดำเนินการไหลของการควบคุมต่อไป เมื่อพบคำสั่ง **continue** ในลูป จะไม่ยุติการวนซ้ำ แต่จะข้ามโฟลว์ไปยังการวนซ้ำถัดไป

```
void main() {
   int count = 1;
   while (count <= 10) {
      count++;

      if (count == 4) {
         print("Number 4 is skipped");
         continue;
      }

      print("Geek, you are inside loop $count");
   }
   print("Geek, you are out of while loop");
}
```

Handle Exceptions

- **Exceptions** คือข้อผิดพลาดที่เกิดขึ้นภายในโปรแกรม เมื่อมีข้อยกเว้นเกิดขึ้นภายในโปรแกรม โฟลว์ปกติของโปรแกรมจะหยุดชะงักและหยุดทำงานอย่างผิดปกติ โดยแสดง **Exceptions** และ **Stack** ดังนั้นต้องมีข้อยกเว้นเพื่อป้องกันไม่ให้แอปพลิเคชันถูกยกเลิก
- **Built-in Exceptions in Dart:**



Program from exception

- The try / on / catch Blocks

```
try {
    // program that might throw an exception
}
on Exception1 {
    // code for handling exception 1
}
catch Exception2 {
    // code for handling exception 2
}
```

Using the ON Block

- ตัวอย่างของการประมวลผลโดยการหารด้วย 0 จะทำให้เกิดข้อผิดพลาดขึ้น โดยจะใช้ **Build in Exception** ***IntegerDivisionByZeroException***

```
void main() {
   int x = 12;
   int y = 0;
   int res;

   try {
      res = x ~/ y;
      print(res);
   }
   on IntegerDivisionByZeroException catch(e) {
      print(e);
   }
}
```

Using the catch Block

- ในบ้างครั้งเราอาจจะไม่รู้ร่วงหน้าว่าจะเกิด **Exception** อะไรขึ้นเราสามารถใช้ **cast** ในการควบคุม **Exception** ที่จะเกิดขึ้น

```
void main() {
   int x = 12;
   int y = 0;
   int res;

   try {
      res = x ~/ y;
      print(res);
   }
   catch(e) {
      print(e);
   }
}
```

Exploring Function And Operators

- **Function**
  - **Function** คือส่วนประกอบสำคัญของโค้ดที่อ่านได้ บำรุงรักษาได้ และนำกลับมาใช้ใหม่ได้ ฟังก์ชันคือชุดของคำสั่งเพื่อทำงานเฉพาะ **Function** จัดระเบียบโปรแกรมเป็นบล็อกตรรกะของโค้ด เมื่อกำหนดแล้ว **Function** อาจถูกเรียกใช้เพื่อเข้าถึงโค้ด ทำให้โค้ดนี้ใช้ซ้ำได้ นอกจากนี้ **Function** ยังช่วยให้อ่านและดูแลโค้ดของโปรแกรมได้ง่าย
- **Operators**
  - **Operands**
    - จัดการเกี่ยวกับข้อมูล
  - **Operator**
    - กำหนดวิธีการประมวลผลตัวถูกดำเนินการเพื่อสร้างค่า

Defining a Function

- **function definition**
  - การประกาศ **Function** ในภาษา **Dart** นั้นจะมีรูปร่างคล้าย ๆ กับภาษา **C** หรือ **C++**
- มีรูปแบบมาตรฐานดังนี้

```
function_name() {
   //statements
}
```

```
void function_name() {
   //statements
}
```

```
return_type function_name(){
   //statements
   return value;
}
```

Calling a Function

- การเรียกใช้ฟังก์ชันเพื่อดำเนินการ กระบวนการนี้เรียกว่า **function invocation**

Function Define

Function Invocation

Returning Functions

- เมื่อมีการเรียกใช้ฟังก์ชันในการทำงานบางอย่างแล้วต้องการส่งผมที่ได้กลับมา
- รูปแบบ **Return Function**

```
return_type function_name(){
   //statements
   return value;
}
```

  - **return_type** ทุก ๆ ประเภทข้อมูล
  - คำสั่ง **return** เป็นทางเลือก ถ้าไม่ได้ระบุฟังก์ชันส่งค่าคืน **null;**

Function Define
Return type String

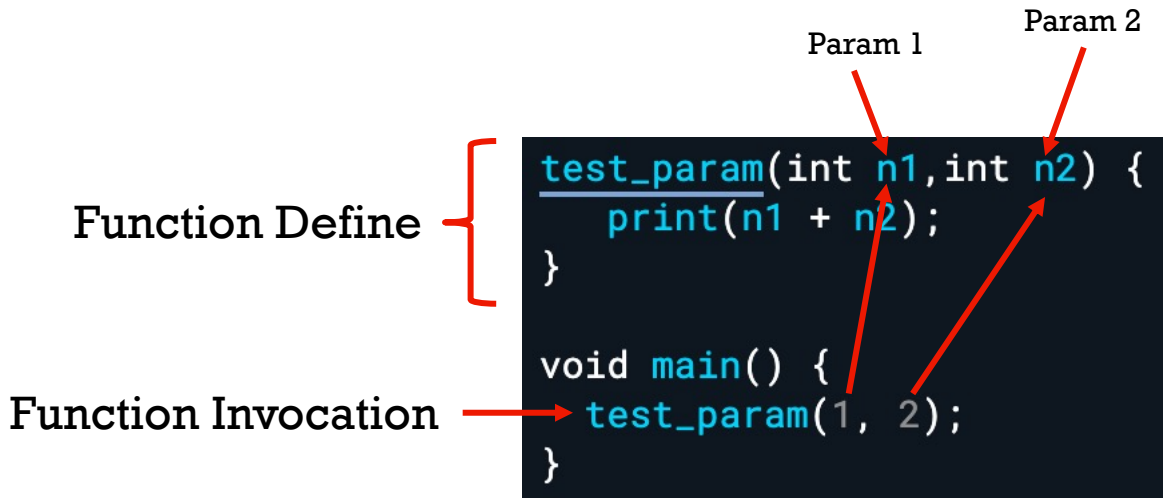
Function Invocation

```
String test() {
    return "hello world";
}

void main() {
    print(test());
}
```

Parameterized Function

- พารามิเตอร์เป็นกลไกในการส่งผ่านค่าไปยังฟังก์ชัน พารามิเตอร์เป็นส่วนหนึ่งของลายเซ็นของฟังก์ชัน ค่าพารามิเตอร์จะถูกส่งไปยังฟังก์ชันในระหว่างการเรียกใช้ เว้นแต่จะระบุไว้อย่างชัดเจน จำนวนค่าที่ส่งไปยังฟังก์ชันต้องตรงกับจำนวนพารามิเตอร์ที่กำหนดไว้
- รูปแบบ **Function Parameter**

```
Function_name(data_type param_1, data_type param_2[…]) {
   //statements
}
```



Optional Parameters

- Optional Positional Parameter
- Optional named parameter
- Optional Parameters with Default Values

Optional Positional Parameter

- หากต้องการระบุพารามิเตอร์ตำแหน่งที่เป็นตัวเลือก ให้ใช้วงเล็บเหลี่ยม []
- โดยมีรูปแบบดังนี้

```
void function_name(param1, [optional_param_1, optional_param_2]) { }
```

```
test_param(n1,[s1]) {
   print(n1);
   print(s1);
}

void main() {
  test_param(1);
}
```

Optional Parameter

Optional named parameter

- ต่างจากพารามิเตอร์ตำแหน่ง ต้องระบุชื่อของพารามิเตอร์ในขณะที่ส่งค่า วงเล็บปีกกา {} สามารถใช้เพื่อระบุพารามิเตอร์ชื่อตัวเลือก **optional**
- รูปแบบการประกาศ ฟังก์ชัน

```
void function_name(a, {optional_param1, optional_param2}) { }
```

- รูปแบบการเรียกใช้งาน

```
function_name(optional_param:value,…);
```

Optional named parameter

```
test_param(n1,{s1,s2}) {
   print(n1);
   print(s1);
   print(s2);
}

void main() {
  test_param(1, s1: 100, s2: 'Helo');
}
```

Optional Parameters with Default Values

- พารามิเตอร์ของฟังก์ชันสามารถกำหนดค่าเริ่มต้น อย่างไรก็ตาม พารามิเตอร์ดังกล่าวสามารถส่งผ่านค่าได้เช่นกัน
- รูปแบบของการประกาศฟังก์ชัน

```
function_name(param1,{param2= default_value}) {
   //......
}
```

```
void test_param(n1,{s1:12}) {
   print(n1);
   print(s1);
}

void main() {
  test_param(1);
}
```

กำหนดค่าเริ่มต้น

Lambda Functions

- **Lambda functions** เป็นกลไกที่กระชับเพื่อแสดงถึงฟังก์ชัน ฟังก์ชันเหล่านี้เรียกอีกอย่างว่า **Arrow functions**
- รูปแบบการประกาศ **Lambda functions**

```
[return_type]function_name(parameters)=>expression;
```

```
printMsg()=> print("hello");

int test()=>123;

void main() {
  printMsg();
  print(test());
}
```

Operators

- Arithmetic Operators
- Equality and Relational Operators
- Type test Operators
- Bitwise Operators
- Assignment Operators
- Logical Operators

Arithmetic Operators

Sr.No	Operators	Meaning
1	+	Add
2	-	Subtract
3	-expr	Unary minus, also known as negation (reverse the sign of the expression)
4	*	Multiply
5	/	Divide
6	~/	Divide, returning an integer result
7	%	Get the remainder of an integer division (modulo)
8	++	Increment
9	--	Decrement

Equality and Relational Operators

Operator	Description	Example
>	Greater than	(A > B) is False
<	Lesser than	(A < B) is True
>=	Greater than or equal to	(A >= B) is False
<=	Lesser than or equal to	(A <= B) is True
==	Equality	(A==B) is False
!=	Not equal	(A!=B) is True

Type test Operators

Operator	Meaning
is	True if the object has the specified type
is!	False if the object has the specified type

Bitwise Operators

Operator	Description	Example
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a | b	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	a ^ b	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bitwise NOT	~ a	Inverts the bits of its operand.
Left shift	a ≪ b	Shifts a in binary representation b (< 32) bits to the left, shifting in zeroes from the right.
right shift	a ≫ b	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off.

Assignment Operators

Sr.No	Operator & Description
1	=(Simple Assignment)Assigns values from the right side operand to the left side operand Ex:C = A + B will assign the value of A + B into C
2	??=Assign the value only if the variable is null
3	+=(Add and Assignment)It adds the right operand to the left operand and assigns the result to the left operand. Ex: C += A is equivalent to C = C + A
4	–=(Subtract and Assignment)It subtracts the right operand from the left operand and assigns the result to the left operand. Ex: C -= A is equivalent to C = C – A
5	*=(Multiply and Assignment)It multiplies the right operand with the left operand and assigns the result to the left operand. Ex: C *= A is equivalent to C = C * A
6	/=(Divide and Assignment)It divides the left operand with the right operand and assigns the result to the left operand.

Logical Operators

Operator	Description	Example
&&	OR – The operator returns true if at least one of the expressions specified return true	(A > 10 && B > 10) is False.
||	OR – The operator returns true if at least one of the expressions specified return true	(A > 10 || B > 10) is True.
!	NOT – The operator returns the inverse of the expression's result. For E.g.: !(7>5) returns false	!(A > 10) is True.

Classes And Object Oriented Programming

- Dart is an object-oriented language. It supports object-oriented programming features like classes, interfaces, etc. A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. Dart gives built-in support for this concept called class
- Declaring 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. The syntax for the same is given below

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

Declaring a Class

- The class keyword is followed by the class name. The rules for identifiers must be considered while naming a class.
- A class definition can include the following
  - **Fields** – A field is any variable declared in a class. Fields represent data pertaining to objects.
  - **Setters and Getters** – Allows the program to initialize and retrieve the values of the fields of a class. A default getter/ setter is associated with every class. However, the default ones can be overridden by explicitly defining a setter/ getter.
  - **Constructors** – responsible for allocating memory for the objects of the class.
  - **Functions** – Functions represent actions an object can take. They are also at times referred to as methods.

Declaring a class

- The example declares a class Car. The class has a field named engine. The disp() is a simple function that prints the value of the field engine.

```
class Car {
  Car ();
  // field
  String engine = "E1001";

  // function
  void disp() {
    print(engine);
  }
}
```

Creating Instance of the class

- To create an instance of the class, use the new keyword followed by the class name. The syntax for the same is given below

```
var object_name = new class_name([ arguments ])
```

  - The new keyword is responsible for instantiation.
  - The right-hand side of the expression invokes the constructor. The constructor should be passed values if it is parameterized

- Example: Instantiating a class

```
void main() {
  var obj = new Car();
}
```

Accessing Attributes and Functions

- A class's attributes and functions can be accessed through the object. Use the '.' dot notation (called as the period) to access the data members of a class

```
//accessing an attribute
obj.field_name

//accessing a function
obj.function_name()
```

- The following example to understand how to access attributes and functions in Dart

```
void main() {
  var obj = new Car();
  obj.disp();
}
```

Dart Constructors

- A constructor is a special function of the class that is responsible for initializing the variables of the class. Dart defines a constructor with the same name as that of the class. A constructor is a function and hence can be parameterized. However, unlike a function, constructors cannot have a return type. If you don't declare a constructor, a default no-argument constructor is provided for you

```
Class_name(parameter_list) {
   //constructor body
}
```

- The following example shows how to use constructors in Dart

```
void main() {
  var obj = new Car("E1001");
  obj.disp();
}
```

```
class Car {
  Car (String engine){
    this.engine = engine;
  }
  // field
  String engine = '';

  // function
  void disp() {
    print(engine);
  }
}
```

Named Constructors

- Dart provides named constructors to enable a class define multiple constructors. The syntax of named constructors is as given below

```
Class_name.constructor_name(param_list)
```

- The following example shows how you can use named constructors in Dart

```dart
void main() {
  var obj =  Car.namedConst("E1005");
  obj.disp();
}
```

```dart
class Car {
  Car (String engine){
    this.engine = engine;
  }
  Car.namedConst(String engine) {
    print("The engine is : ${engine}");
    this.engine = engine;
  }
  // field
  String engine = '';

  // function
  void disp() {
    print(engine);
  }
}
```

The this Keyword

- The this keyword refers to the current instance of the class. Here, the parameter name and the name of the class's field are the same. Hence to avoid ambiguity, the class's field is prefixed with the this keyword. The following example explains the same
- The following example explains how to use the this keyword in Dart

```
class Car {
  Car (String engine){
    this.engine = engine;
  }
  Car.namedConst(String engine) {
    print("The engine is : ${engine}");
    this.engine = engine;
  }
  // field
  String engine = '';

  // function
  void disp() {
    print(engine);
  }
}
```

Getters and Setters

- Getters and Setters, also called as accessors and mutators, allow the program to initialize and retrieve the values of class fields respectively. Getters or accessors are defined using the get keyword. Setters or mutators are defined using the set keyword
- Define getter

```
Return_type  get identifier
{
}
```

- Define setter

```
set identifier
{
}
```

Getters and Setters Example

```
void main() {
  Student s1 = new Student();
  s1.stud_name = 'MARK';
  s1.stud_age = 8;
  print(s1.stud_name);
  print(s1.stud_age);
}
```

```
class Student {
   String name = '';
   int age = 0;

   Student();

   String get stud_name {
      return name;
   }

   void set stud_name(String name) {
      this.name = name;
   }

   void set stud_age(int age) {
      if(age<= 0) {
        print("Age should be greater than 5");
      }  else {
         this.age = age;
      }
   }

   int get stud_age {
      return age;
   }
}
```

Class Inheritance

- Dart supports the concept of Inheritance which is the ability of a program to create new classes from an existing class. The class that is extended to create newer classes is called the parent class/super class. The newly created classes are called the child/sub classes.
- A class inherits from another class using the 'extends' keyword. Child classes inherit all properties and methods except constructors from the parent class

```
class child_class_name extends parent_class_name
```

Class Inheritance Example

```
class Shape {
   void cal_area() {
      print("calling calc area defined in the Shape class");
   }
}
class Circle extends Shape {}
```

```
void main() {
  var obj = new Circle();
  obj.cal_area();
}
```

Class Inheritance and Method Overriding

- Method Overriding is a mechanism by which the child class redefines a method in its parent class. The following example

```
class Shape {
   void cal_area() {
      print("calling calc area defined in the Shape class");
   }
}
class Circle extends Shape {
  @override
  void cal_area() {
      print("calling calc area defined in the Circle class");
   }
}
```

```
void main() {
  var obj = new Circle();
  obj.cal_area();
}
```

The super Keyword

- The super keyword is used to refer to the immediate parent of a class. The keyword can be used to refer to the super class version of a variable, property, or method. The following example

```
class Shape {
   void cal_area() {
      print("calling calc area defined in the Shape class");
   }
}
class Circle extends Shape {
  @override
  void cal_area() {
      super.cal_area();
      print("calling calc area defined in the Circle class");
   }
}
```

```
void main() {
  var obj = new Circle();
  obj.cal_area();
}
```

The static Keyword

- The **static** keyword can be applied to the data members of a class, i.e., **fields** and **methods**. A static variable retains its values till the program finishes execution. Static members are referenced by the class name

```
class StaticMem {
   static int num = 0;
   static disp() {
      print("The value of num is ${StaticMem.num}")  ;
   }
}
```

```
void main() {
  StaticMem.num = 12;
  // initialize the static variable }
  StaticMem.disp();
}
```

Interface

- An interface defines the syntax that any entity must adhere to. Interfaces define a set of methods available on an object. Dart does not have a syntax for declaring interfaces. Class declarations are themselves interfaces in Dart.
- Classes should use the implements keyword to be able to use an interface. It is mandatory for the implementing class to provide a concrete implementation of all the functions of the implemented interface. In other words, a class must redefine every function in the interface it wishes to implement

```
class identifier implements interface_name
```

Interface Example

```dart
class interface1 {
  void print1(){}
}

class interface2 {
  void print2(){}
}


class implementer implements interface1, interface2 {
  void print1(){
    print('interface1');
  }
  void print2(){
    print('interface2');
  }
}
```

```dart
void main() {
  var obj = implementer();
  obj.print1();
  obj.print2();
  print(obj is interface1);
  print(obj is interface2);

}
```

abstract class

- You can create an abstract class to be extended (or implemented) by a concrete class. Abstract classes can contain abstract methods (with empty bodies).

```
abstract class Sharp {
  double area();
}
```

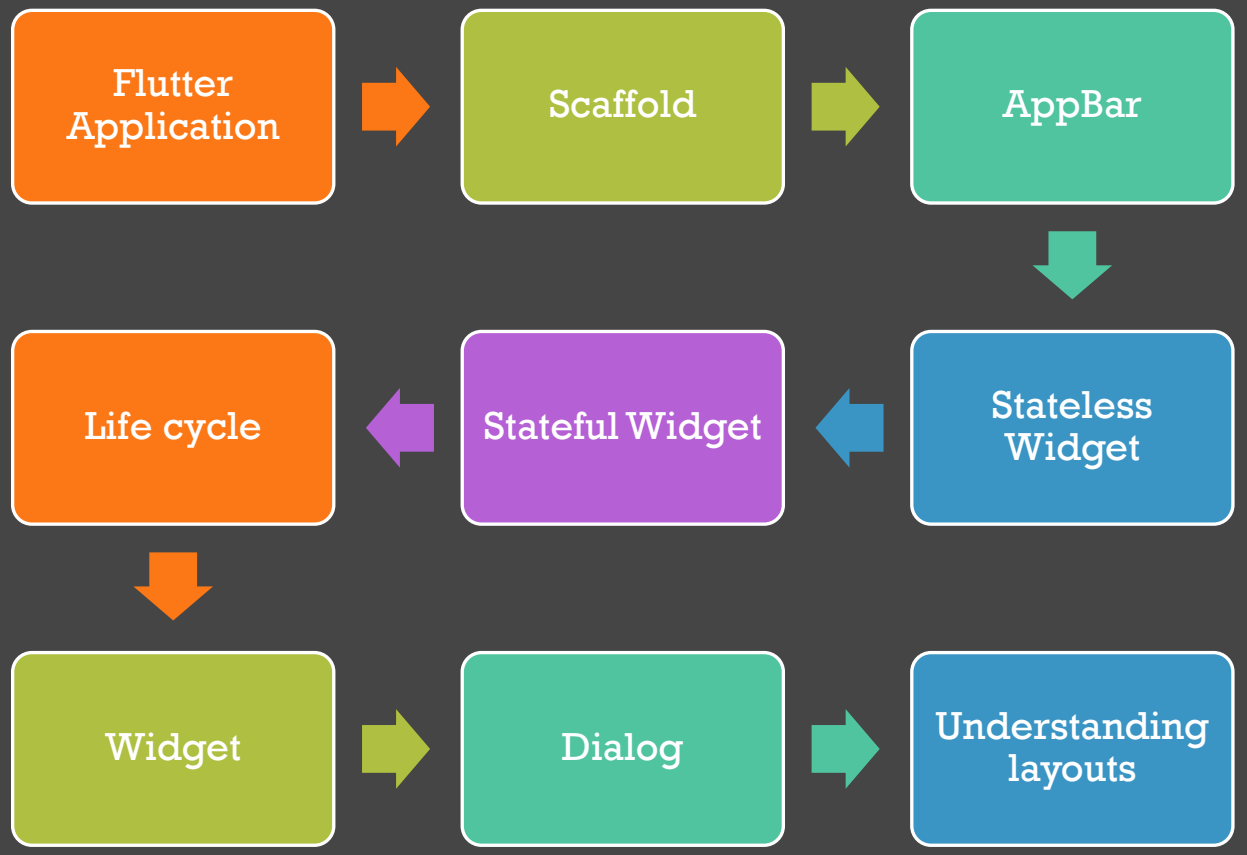
abstract class Example

```dart
abstract class Sharp {
  double area();
}

class Rectangle  exte  s Sharp {
  final double width;
  final double height
  Rectangle({this.wid    = 0, this.height = 0});
  double area(){
    return this.width    this.height;
  }
}

class Circle extends  arp {
  final double radius
  final double pi = 3   4;
  Circle({this.radius     0});
  double area(){
    return pi * (radius * radius);
  }
}

void cal (Sharp sharp) {
  print(sharp.area());
}
```

override

```dart
void main() {
  Sharp sharp1 = Rectangle(width: 10, height: 10);
  Sharp sharp2 = Circle(radius: 10,);
  cal(sharp1);
  cal(sharp2);
}
```

Layout and Widget



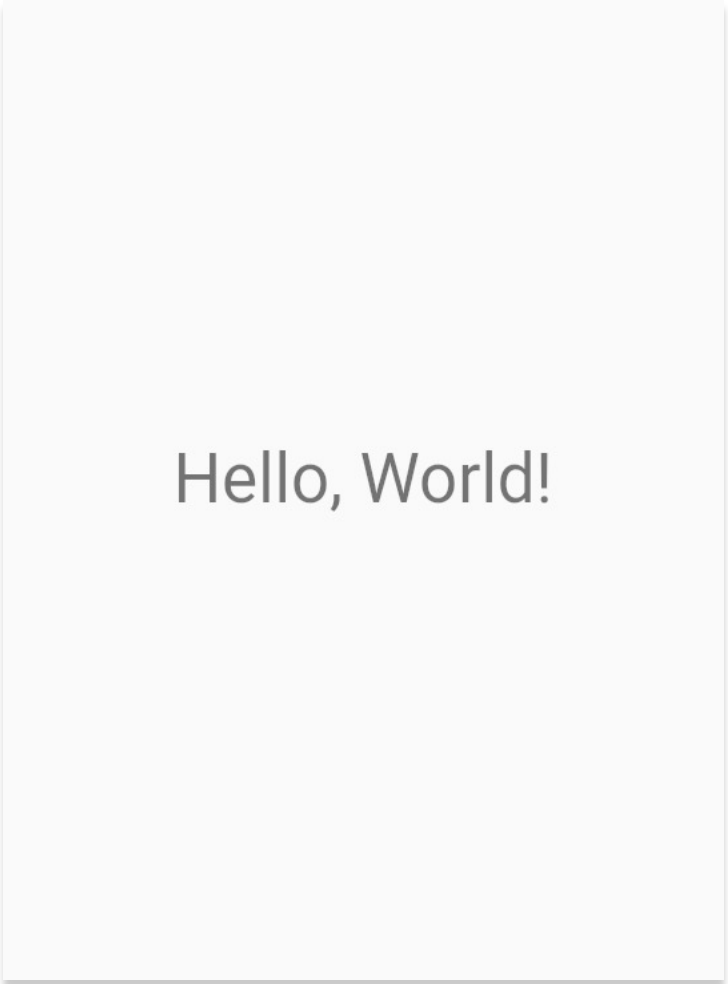
Flutter Application

- Fast Development
  - Paint your app to life in milliseconds with Stateful Hot Reload. Use a rich set of fully-customizable widgets to build native interfaces in minutes
- Expressive and Flexible UI
  - Quickly ship features with a focus on native end-user experiences. Layered architecture allows for full customization, which results in incredibly fast rendering and expressive and flexible designs.
- Native Performance
  - Flutter’s widgets incorporate all critical platform differences such as scrolling, navigation, icons and fonts, and your Flutter code is compiled to native ARM machine code using Dart's native compilers

Flutter Hello World

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        body: Center(
          child: Text('Hello, World!',
                      style: Theme.of(context).textTheme.headline4),
        ),
      ),
    );
  }
}
```

Hello World

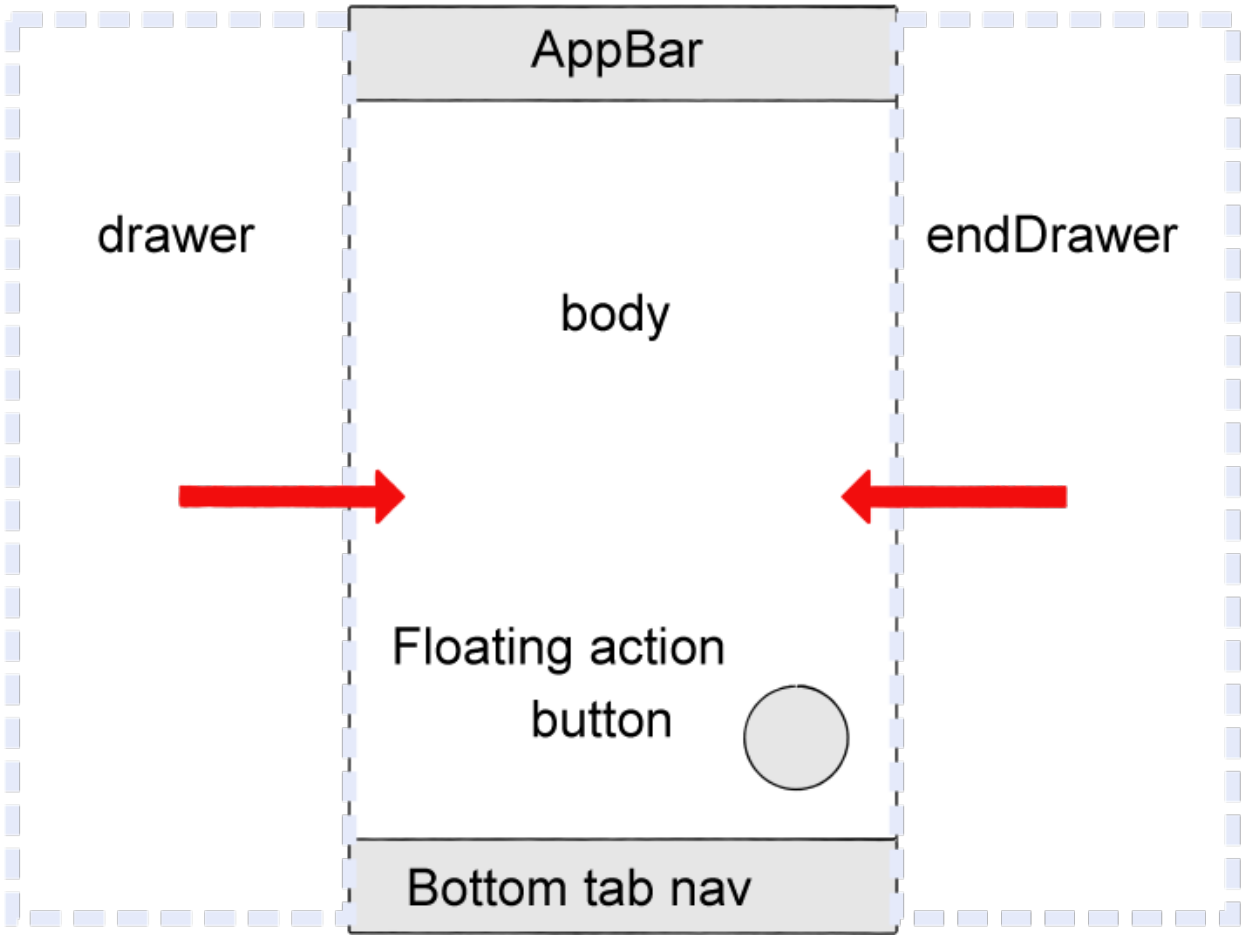


MaterialApp

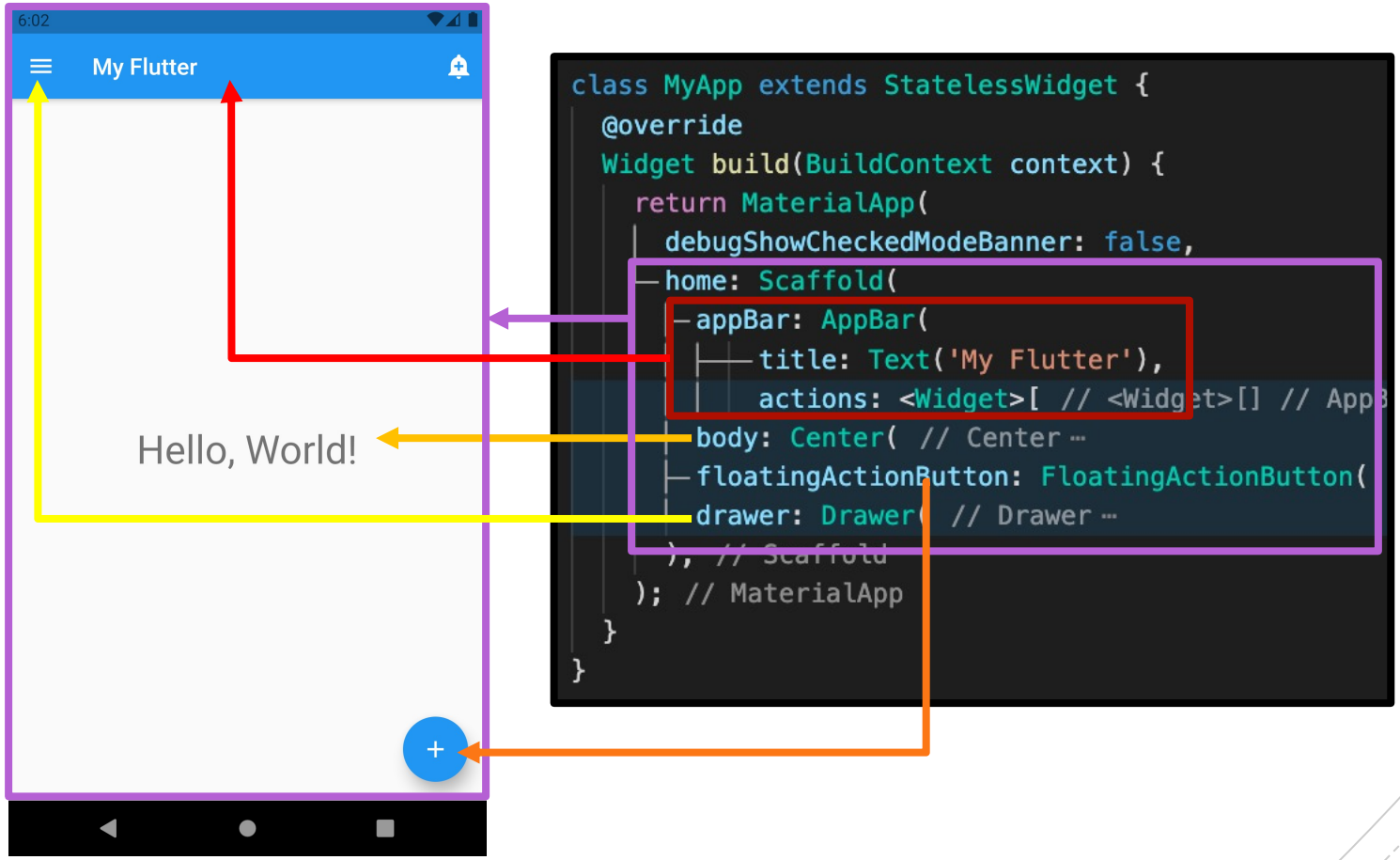
- The MaterialApp widget provides a top of benefits that effect its entire widget subtree. This section is the beginning of our discussion of many widgets that provide helpful functionality for free

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Center(child: Text('Hello'))
    );
  }
}
```

Scaffold


AppBar
drawer
endDrawer
body
Floating action
button
Bottom tab nav

Scaffold


6:02
My Flutter
Hello, World!
class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
debugShowCheckedModeBanner: false,
home: Scaffold(
appBar: AppBar(
title: Text('My Flutter'),
actions: <Widget>[// <Widget>[] // App
body: Center(// Center …
floatingActionButton: FloatingActionButton(
drawer: Drawer(// Drawer …
); // MaterialApp
}
}

Scaffold

- Scaffold is a class in **flutter** which provides many widgets or we can say **APIs** like Drawer, SnackBar, BottomNavigationBar, FloatingActionButton, AppBar etc. **Scaffold** will expand or occupy the whole device screen. It will occupy the available space. Scaffold will provide a framework to implement the basic material design layout of the application.
- Scaffold Constructor

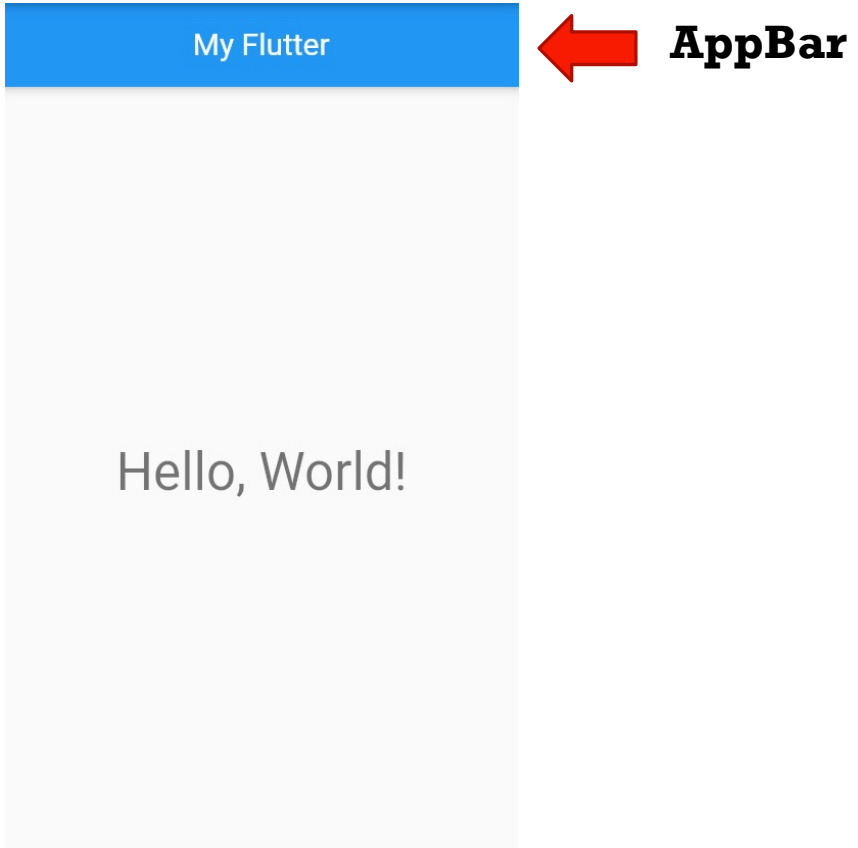
```
const Scaffold({
  Key key,
  this.appBar,
  this.body,
  this.floatingActionButton,
  this.floatingActionButtonLocation,
  this.floatingActionButtonAnimator,
  this.persistentFooterButtons,
  this.drawer,
  this.endDrawer,
  this.bottomNavigationBar,
  this.bottomSheet,
  this.backgroundColor,
  this.resizeToAvoidBottomPadding = true,
  this.primary = true,
})
```

Scaffold

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        body: Center(
          child: Text('Hello, World!',
                      style: Theme.of(context).textTheme.headline4),
        ),
      ),
    );
  }
}
```

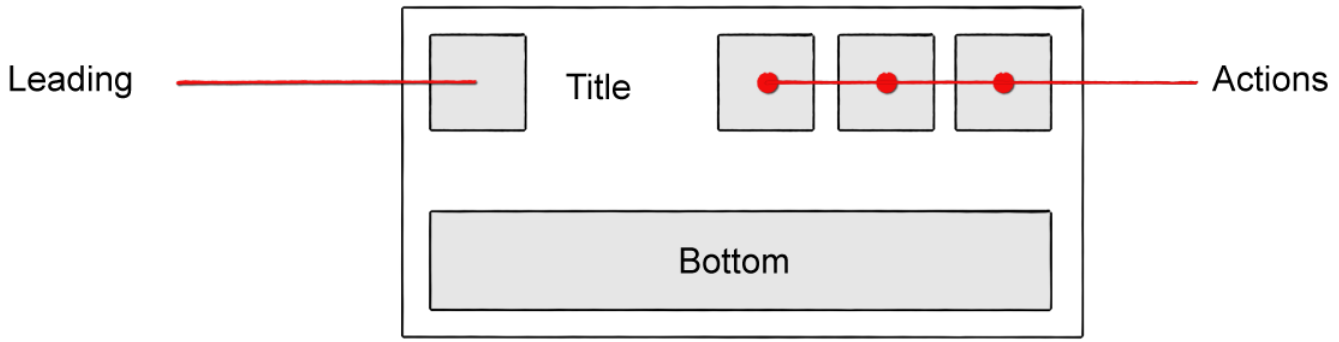
AppBar

- It displays a horizontal bar which mainly placed at the top of the Scaffold. appBar uses the widget AppBar which has its own properties like elevation, title, brightness, etc.



AppBar

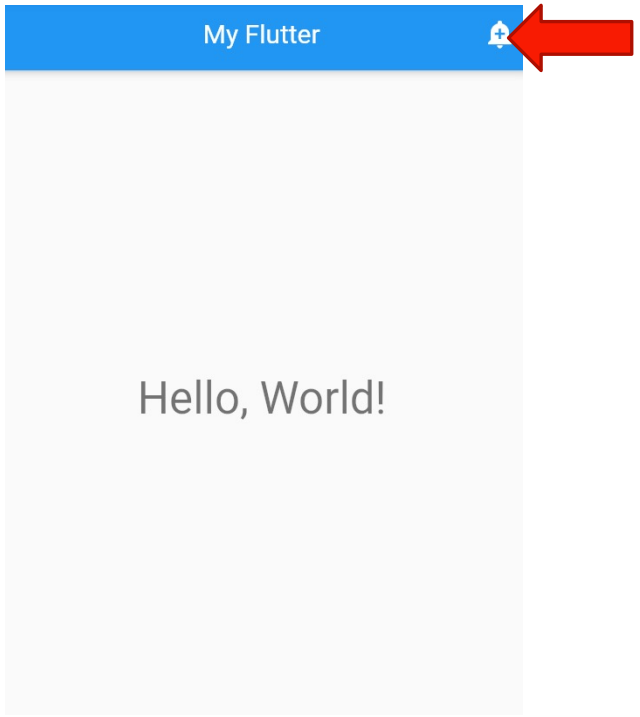
A material design app bar



AppBar Action

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
            title: Text('My Flutter'),
            actions: <Widget>[
              IconButton(
                icon: const Icon(Icons.add_alert),
                tooltip: 'Show Snackbar',
                onPressed: () {
                  ScaffoldMessenger.of(context).showSnackBar(
                      const SnackBar(content: Text('This is a snackbar')));
                },
              ),
            ]),
        body: Center(
          child: Text('Hello, World!',
              style: Theme.of(context).textTheme.headline4),
        ),
      ),
    );
  }
}
```

AppBar Action



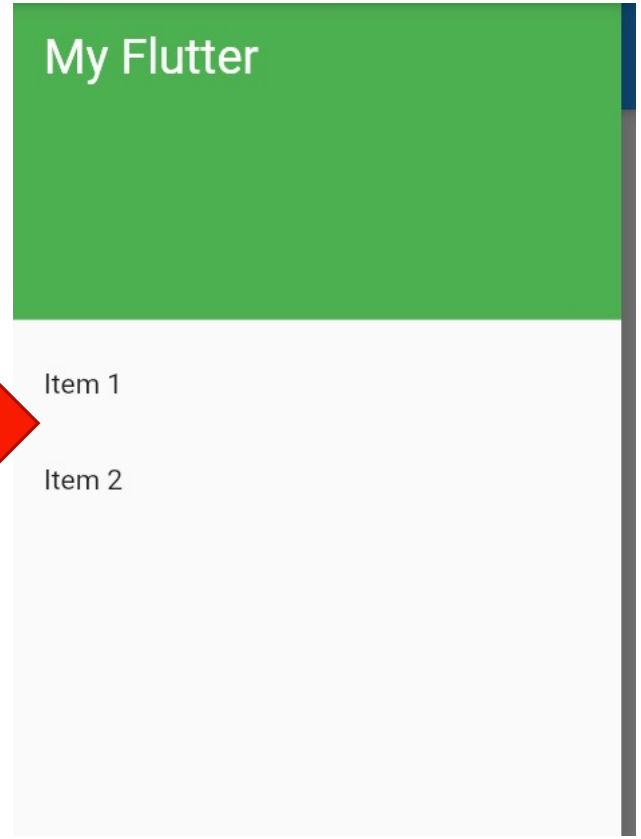

```
actions: <Widget>[
  IconButton(
    icon: const Icon(Icons.add_alert),
    tooltip: 'Show Snackbar',
    onPressed: () {
      ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text('This is a snackbar')));
    },
  ),
]),
```

drawer

- **drawer** is a slider menu or a panel which is displayed at the side of the Scaffold. The user has to swipe left to right or right to left according to the action defined to access the drawer menu. In the Appbar, an appropriate icon for the drawer is set automatically at a particular position. The gesture to open the drawer is also set automatically. It is handled by the Scaffold

drawer

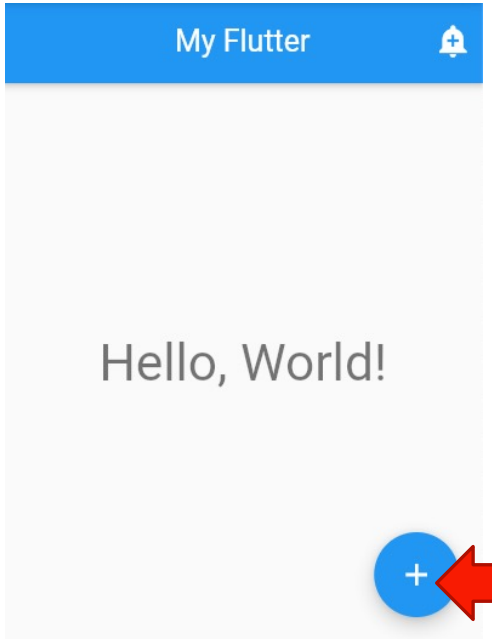
```
drawer: Drawer(
  child: ListView(
    children: const <Widget>[
      DrawerHeader(
        decoration: BoxDecoration(
          color: Colors.green,
        ),
        child: Text(
          'My Flutter',
          style: TextStyle(
            color: Colors.white,
            fontSize: 24,
          ),
        ),
      ),
      ListTile(
        title: Text('Item 1'),
      ),
      ListTile(
        title: Text('Item 2'),
      ),
    ],
  ),
),
```



floatingActionButton

- FloatingActionButton is a button that is placed at the right bottom corner by default. FloatingActionButton is an icon button that floats over the content of the screen at a fixed place. If we scroll the page its position won't change, it will be fixed

floatingActionButton




```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(title: Text('My Flutter'), actions: <Widget>[
          IconButton(
            icon: const Icon(Icons.add_alert),
            tooltip: 'Show Snackbar',
            onPressed: () {
              ScaffoldMessenger.of(context).showSnackBar(
                  const SnackBar(content: Text('This is a snackbar')));
            },
          ),
        ]),
        body: Center(
          child: Text('Hello, World!',
              style: Theme.of(context).textTheme.headline4),
        ),
        floatingActionButton: FloatingActionButton(
            elevation: 10.0,
            child: Icon(Icons.add),
            onPressed: () {
              // action on button press
            }),
      ),
    );
  }
}
```

Stateful vs Stateless Widgets

- The state of an app can very simply be defined as anything that exists in the memory of the app while the app is running. This includes all the widgets that maintain the UI of the app including the buttons, text fonts, icons, animations, etc. So now as we know what are these states let's dive directly into our main topic i.e what are these stateful and stateless widgets and how do they differ from one another
- **State**
  - *The State is the information that can be read synchronously when the widget is built and might change during the lifetime of the widget.*

Stateless Widget

- The widgets whose state can not be altered once they are built are called stateless widgets. These widgets are immutable once they are built i.e any amount of change in the variables, icons, buttons, or retrieving data can not change the state of the app. Below is the basic structure of a stateless widget. Stateless widget overrides the build() method and returns a widget. For example, we use Text or the Icon is our flutter application where the state of the widget does not change in the runtime. It is used when the UI depends on the information within the object itself. Other examples can be Text, RaisedButton, IconButtons.

Stateless Widget Structure

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


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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

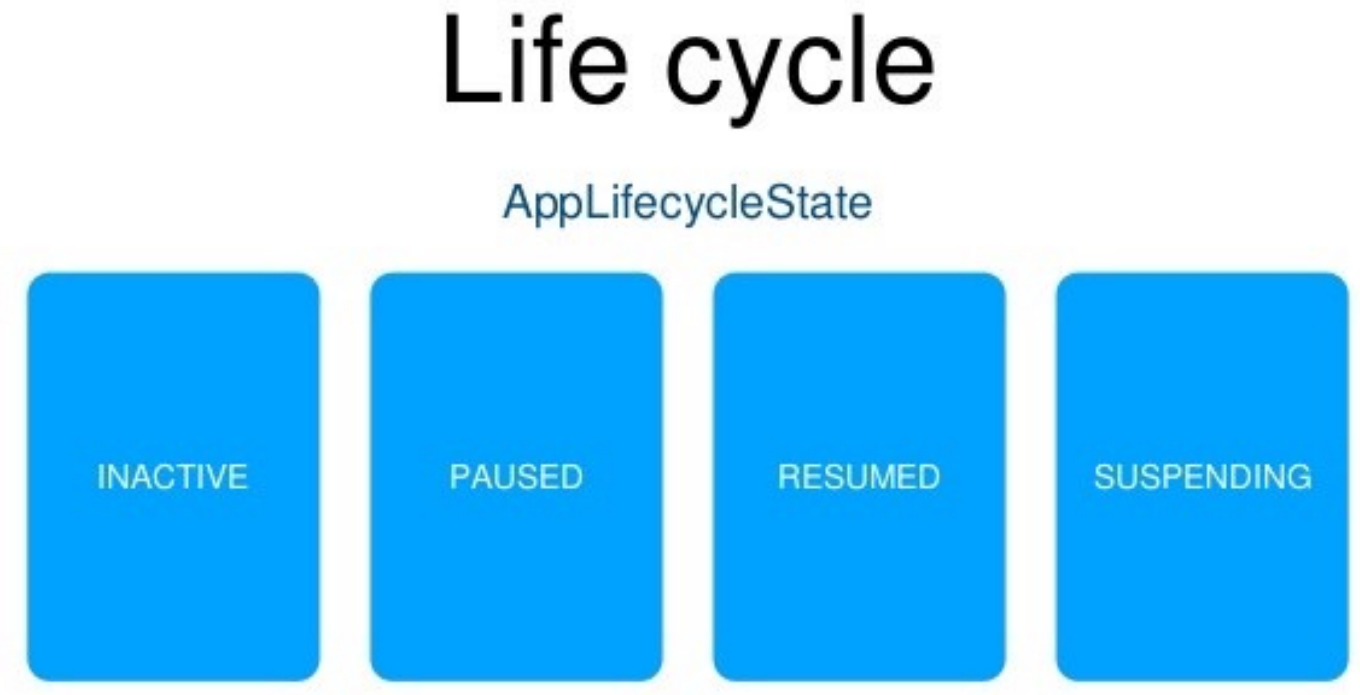
Stateful Widget

- The widgets whose state can be altered once they are built are called stateful Widgets. These states are mutable and can be changed multiple times in their lifetime. This simply means the state of an app can change multiple times with different sets of variables, inputs, data. Below is the basic structure of a stateful widget. Stateful widget overrides the createState() and returns a State. It is used when the UI can change dynamically. Some examples can be CheckBox, RadioButton, Form, TextField

Stateful Widget Structure

```dart
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 Container();
  }
}
```

Life Cycle State



Life cycle

- **inactive** — The application is in an inactive state and is not receiving user input. This event only works on iOS, as there is no equivalent event to map to on Android
- **paused** — The application is not currently visible to the user, not responding to user input, and running in the background. This is equivalent to onPause() in Android
- **resumed** — The application is visible and responding to user input. This is equivalent to onPostResume() in Android
- **suspending** — The application is suspended momentarily. This is equivalent to onStop in Android; it is not triggered on iOS as there is no equivalent event to map to on iOS

Implementation

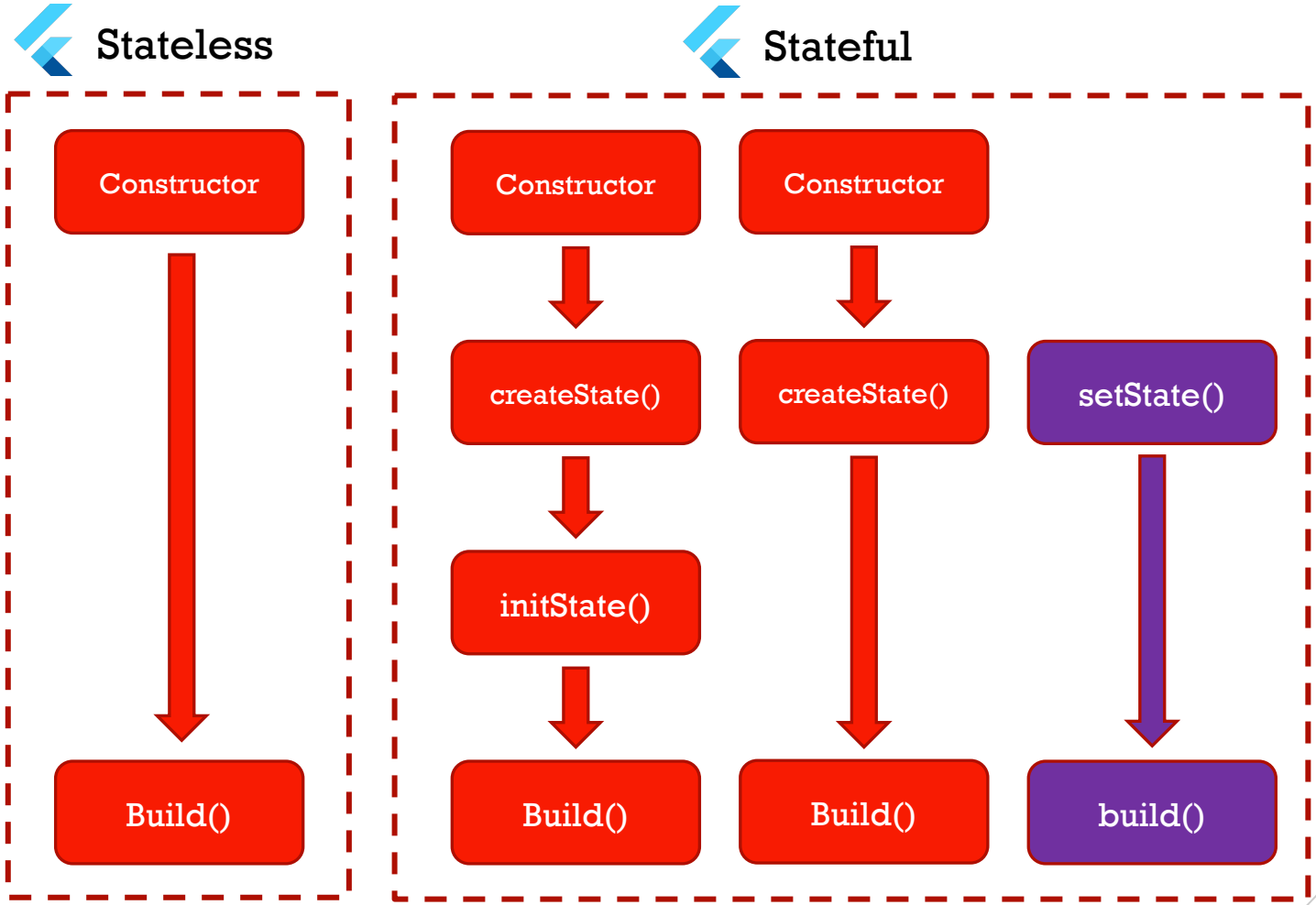
- In order to observe the different lifecycle states from the widgets layer, we use WidgetsBindingObserver class

```dart
class _MyHomePageState extends State<MyHomePage> with WidgetsBindingObserver {
  @override
  void didChangeAppLifecycleState(AppLifecycleState state) {
    print('state = $state');
  }
  @override
  void initState() {
    WidgetsBinding.instance!.addObserver(this);
    super.initState();
  }
  @override
  void dispose() {
    WidgetsBinding.instance!.removeObserver(this);
    super.dispose();
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Flutter Tutorial Lifecycle'),
      ),
      body: Center(),
    );
  }
}
```

Widget Lifecycle Methods

- *createState*
- *initState*
- *didChangeDependencies*
- *build*
- *setState*
- *dispose*

Widget Life Cycle



createState

- **createState():** When we create a stateful widget, our framework calls a createState() method and it must be overridden

```
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

initState

- **initState():** Flutter's initState() method is the first method that is used while creating a stateful class, here we can initialize variables, data, properties, etc. for any widget

```dart
@override
void initState() {
  WidgetsBinding.instance!.addObserver(this);
  super.initState();
}
```

build

- The build method is used each time the widget is rebuilt. This can happen either after calling initState, didChangeDependencies, didUpdateWidget, or when the state is changed via a call to setState

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Flutter Tutorial Lifecycle'),
    ),
    body: Center(),
  );
}
```

didChangeDependencies

- This method is called immediately after *initState* and when dependency of the State object changes via InheritedWidget.

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

dispose

- We use this method when we remove permanently like should release resource created by an object like stop animation

```dart
@override
void dispose() {
  WidgetsBinding.instance!.removeObserver(this);
  super.dispose();
}
```

Widget

- Flutter widgets are built using a modern framework that takes inspiration from **React**. The central idea is that you build your UI out of widgets. Widgets describe what their view should look like given their current configuration and state. When a widget's state changes, the widget rebuilds its description, which the framework diffs against the previous description in order to determine the minimal changes needed in the underlying render tree to transition from one state to the next.

Widget catalog

- Assets, Images, and Icons
  - Manage assets, display images, and show icons.
- Async
  - Async patterns to your Flutter application.
- Basics
  - Widgets you absolutely need to know before building your first Flutter app.
- Input
  - Take user input in addition to input widgets in Material Components and Cupertino
- Layout
  - Arrange other widgets columns, rows, grids, and many other layouts.
- Material Components
  - Visual, behavioral, and motion-rich widgets
- Styling
  - Manage the theme of your app, makes your app responsive to screen sizes, or add padding
- Text
  - Display and style text.
- etc.
  - https://flutter.dev/docs/development/ui/widgets

Assets, Images, and Icons

- Icon
  - A Material Design icon
- Image
  - A widget that displays an image

Basics

- Container
  - A convenience widget that combines common painting, positioning, and sizing widgets.
- ElevatedButton
  - A Material Design elevated button. A filled button whose material elevates when pressed.

Input

- Form
  - An optional container for grouping together multiple form field widgets (e.g. TextField widgets).
- FormField
  - A single form field. This widget maintains the current state of the form field, so that updates and validation errors are visually reflected in the…
- Autocomplete
  - A widget for helping the user make a selection by entering some text and choosing from among a list of options

Material Components

- App structure and navigation
- Buttons
- Input and selections
- Dialogs, alerts, and panels
- Information displays

App structure and navigation

- BottomNavigationBar
  - Bottom navigation bars make it easy to explore and switch between top-level views in a single tap. The BottomNavigationBar widget implements this component.
- Drawer
  - A Material Design panel that slides in horizontally from the edge of a Scaffold to show navigation links in an application.
- TabBar
  - A Material Design widget that displays a horizontal row of tabs.
- TabBarView
  - A page view that displays the widget which corresponds to the currently selected tab. Typically used in conjunction with a TabBar.
- TabController
  - Coordinates tab selection between a TabBar and a TabBarView.

Buttons

- DropdownButton
  - Shows the currently selected item and an arrow that opens a menu for selecting another item.
- FloatingActionButton
  - A floating action button is a circular icon button that hovers over content to promote a primary action in the application. Floating action buttons are…
- IconButton
  - An icon button is a picture printed on a Material widget that reacts to touches by filling with color (ink).
- OutlinedButton
  - A Material Design outlined button, essentially a TextButton with an outlined border.
- PopupMenuButton
  - Displays a menu when pressed and calls onSelected when the menu is dismissed because an item was selected.
- TextButton
  - A Material Design text button. A simple flat button without a border outline.

Input and selections

- Checkbox
- Radio
- Slider
- Switch

Dialogs, alerts, and panels

- AlertDialog
  - Alerts are urgent interruptions requiring acknowledgement that inform the user about a situation. The AlertDialog widget implements this component.
- BottomSheet
  - Bottom sheets slide up from the bottom of the screen to reveal more content. You can call showBottomSheet() to implement a persistent bottom sheet or…
- SnackBar
  - A lightweight message with an optional action which briefly displays at the bottom of the screen.

Styling

- MediaQuery
  - Establishes a subtree in which media queries resolve to the given data.
- Padding
  - A widget that insets its child by the given padding.

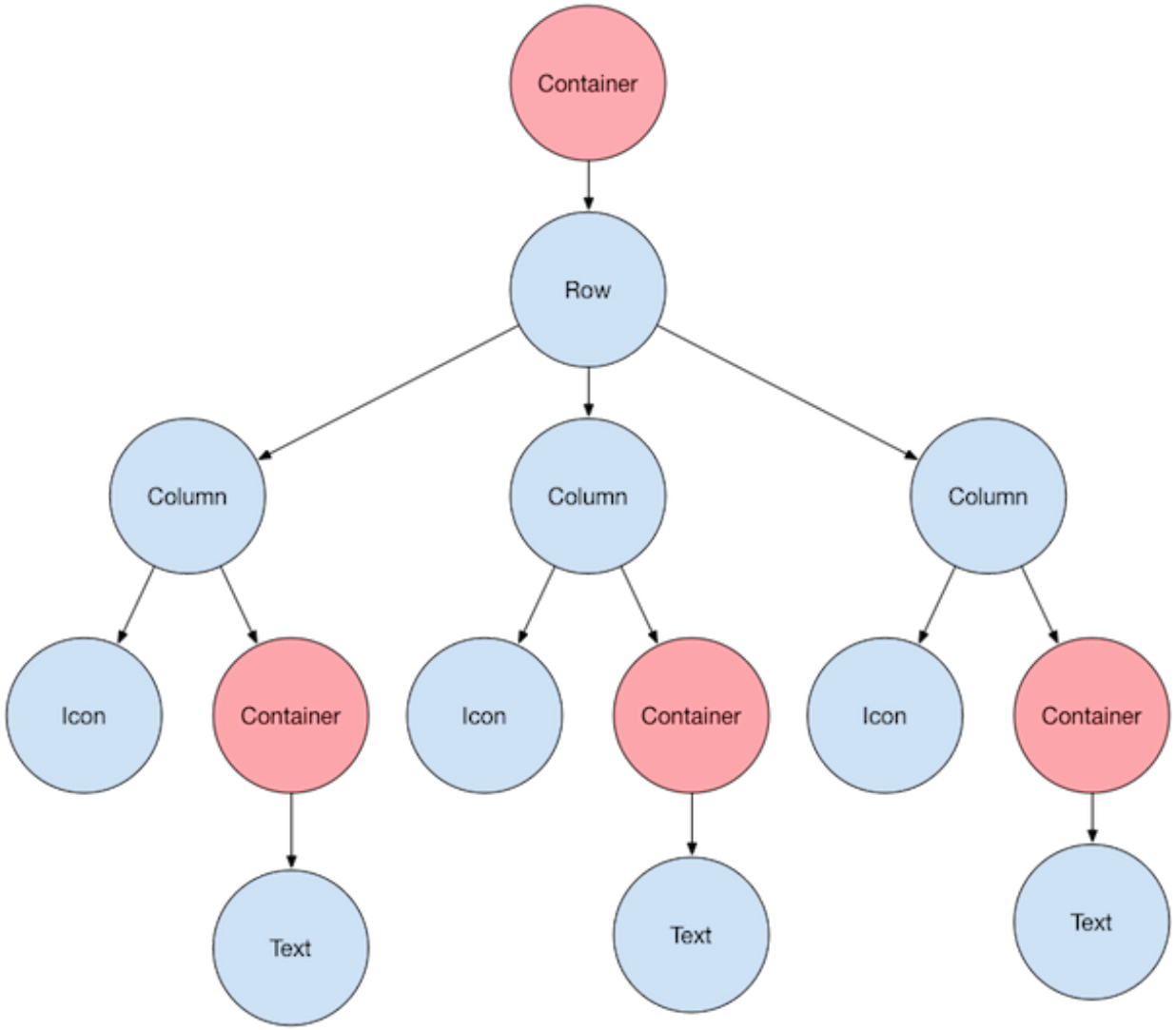
Text

- DefaultTextStyle
  - The text style to apply to descendant Text widgets without explicit style.
- RichText
  - The RichText widget displays text that uses multiple different styles. The text to display is described using a tree of TextSpan objects, each of which...
- Text
  - A run of text with a single style.

Understanding layouts

- The core of Flutter's 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. For example, the first screenshot below shows 3 icons with a label under each one

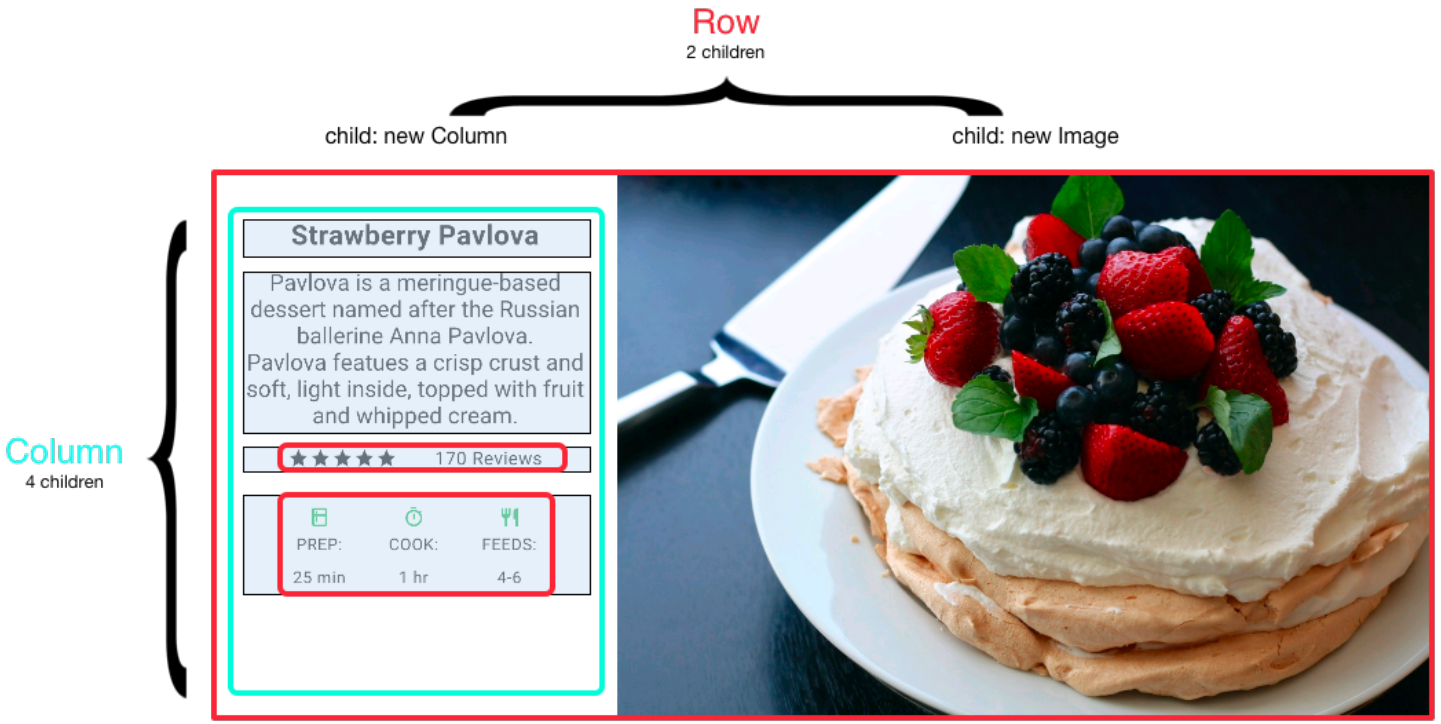
Widget tree


Container
Row
Column
Column
Column
Icon
Container
Icon
Container
Icon
Container
Text
Text
Text

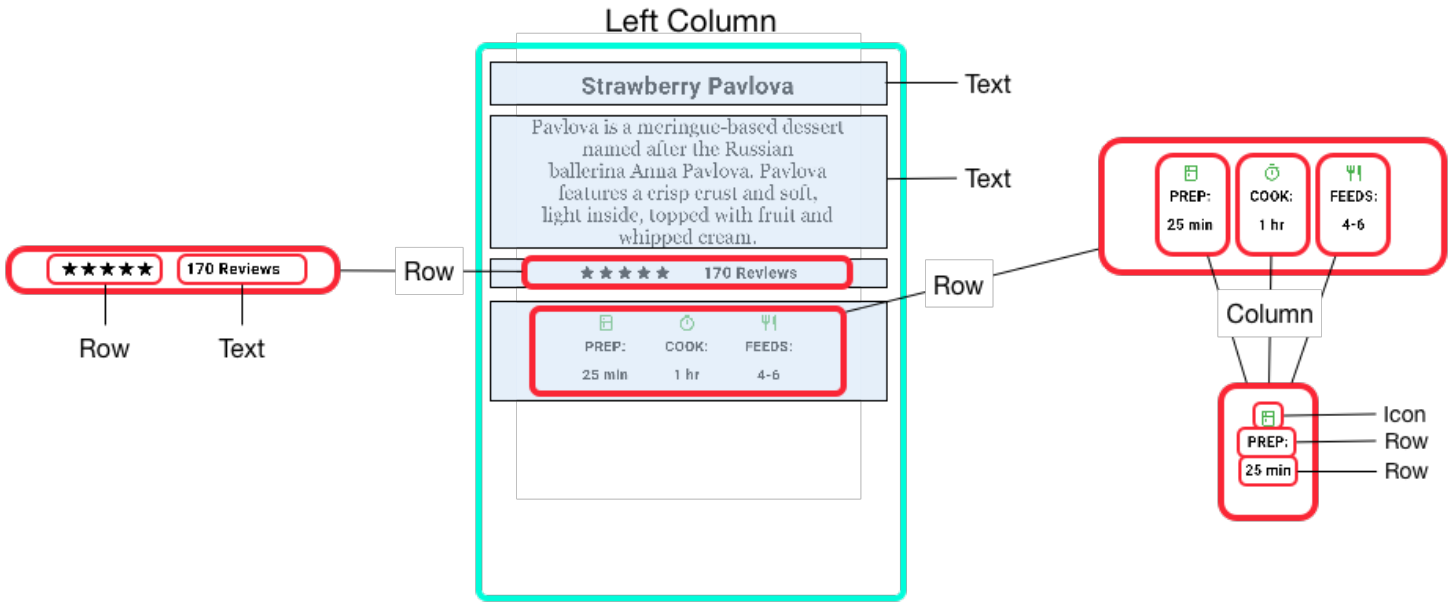
Layout widgets

- How do you lay out a single widget in Flutter? This section shows you how to create and display a simple widget. It also shows the entire code for a simple Hello World app
  - Single-child layout widgets
  - Multi-child layout widgets
  - Sliver widgets

Multiple layout widgets



Multiple layout widgets

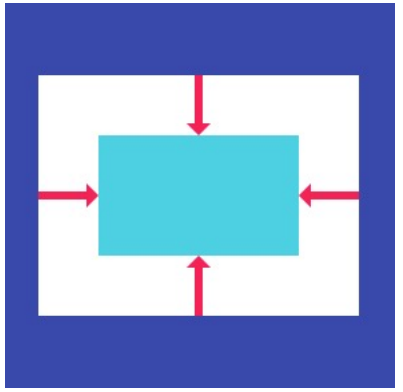


Single-child layout widgets

- Center
- Align
- Container
- Expanded
- Padding
- SizedBox

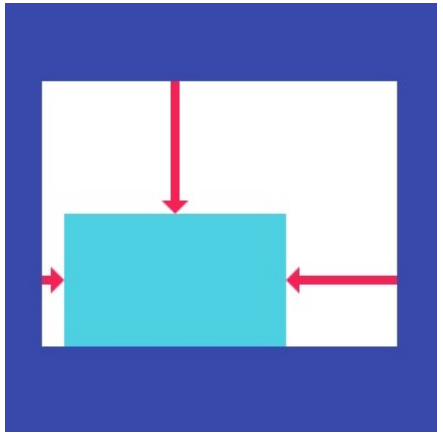
Center

- A widget that centers its child within itself.
- This widget will be as big as possible if its dimensions are constrained and widthFactor and heightFactor are null. If a dimension is unconstrained and the corresponding size factor is null then the widget will match its child's size in that dimension. If a size factor is non-null then the corresponding dimension of this widget will be the product of the child's dimension and the size factor. For example if widthFactor is 2.0 then the width of this widget will always be twice its child's width.



Align

- A widget that aligns its child within itself and optionally sizes itself based on the child's size.
- For example, to align a box at the bottom right, you would pass this box a tight constraint that is bigger than the child's natural size, with an alignment of Alignment.bottomRight.

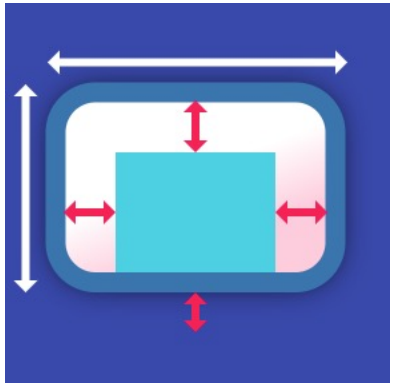


AlignmentDirectional

- **bottomCenter**
  - The bottom corner on the "end" side.
- **bottomEnd**
  - The bottom corner on the "end" side.
- **bottomStart**
  - The bottom corner on the "start" side.
- **center**
  - The center point, both horizontally and vertically.
- **centerEnd**
  - The center point along the "end" edge.
- **centerStart**
  - The center point along the "start" edge.
- **topCenter**
  - The center point along the top edge
- **topEnd**
  - The top corner on the "end" side.
- **topStart**
  - The top corner on the "start" side

Container

- A convenience widget that combines common painting, positioning, and sizing widgets.
  - decoration → Decoration
    - The decoration to paint behind the child.
  - color → Color
    - The color to paint behind the child
  - alignment → AlignmentGeometry
    - Align the child within the container
  - margin → EdgeInsetsGeometry
  - padding → EdgeInsetsGeometry



BoxDecoration

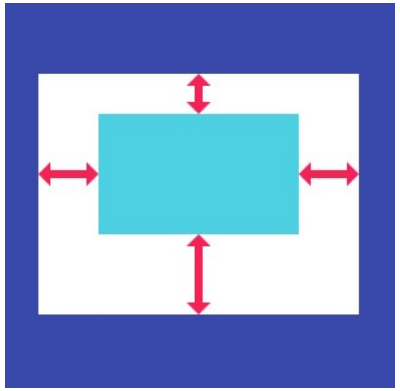
- The **BoxDecoration** class provides a variety of ways to draw a box.
- The box has a **border**, a body, and may cast a **boxShadow**.
- The shape of the box can be a circle or a rectangle. If it is a rectangle, then the **borderRadius** property controls the roundness of the corners.
- The body of the box is painted in layers. The bottom-most layer is the color, which fills the box. Above that is the gradient, which also fills the box. Finally there is the image, the precise alignment of which is controlled by the **DecorationImage** class.
- The **border** paints over the body
- the **boxShadow**, naturally, paints below it.

Expanded

- A widget that expands a child of a Row, Column, or Flex so that the child fills the available space.
- Using an Expanded widget makes a child of a Row, Column, or Flex expand to fill the available space along the main axis (e.g., horizontally for a Row or vertically for a Column). If multiple children are expanded, the available space is divided among them according to the flex factor.
- An Expanded widget must be a descendant of a Row, Column, or Flex, and the path from the Expanded widget to its enclosing Row, Column, or Flex must contain only StatelessWidgets or StatefulWidgets (not other kinds of widgets, like RenderObjectWidgets).

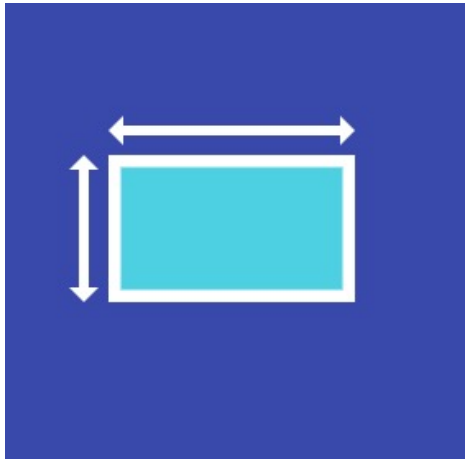
Padding

- When passing layout constraints to its child, padding shrinks the constraints by the given padding, causing the child to layout at a smaller size. Padding then sizes itself to its child's size, inflated by the padding, effectively creating empty space around the child



SizedBox

- A box with a specified size. If given a child, this widget forces its child to have a specific width and/or height

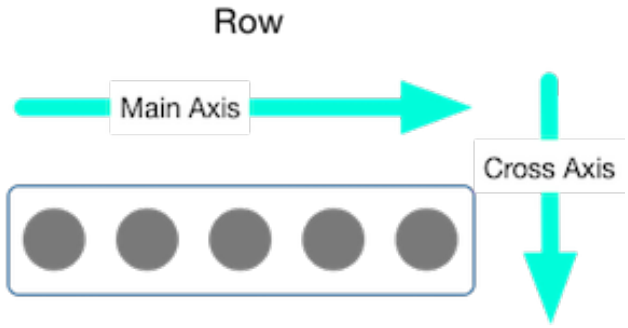


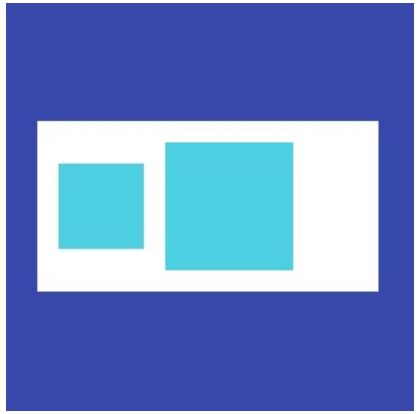
Multi-child layout widgets

- Row
- Column
- GridView
- Stack
- ListView
- LayoutBuilder

Row

- Layout a list of child widgets in the horizontal direction.





Row Aligning widgets

- For a row, the main axis runs horizontally and the cross axis runs vertically
  - mainAxisAlignment axis runs horizontally
  - crossAxisAlignment axis runs vertically

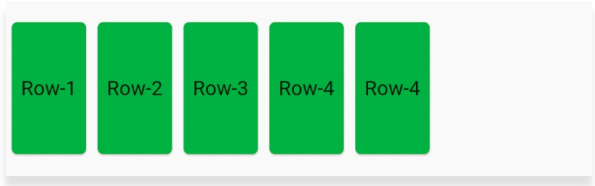
```
@override
Widget build(BuildContext context) {
  return Container(
    height: 300,
    width: double.infinity,
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      crossAxisAlignment: CrossAxisAlignment.stretch,
      children: [
        Container( // Container …
        Container( // Container …
        Container( // Container …
        Container( // Container …
        Container( // Container …
      ],
    ), // Row
  ); // Container
}
```

```
Container(
  width: 60,
  height: 100,
  child: Card(
    color: Colors.green,
    child: Center(child: Text('Row-1')),
    ), // Card
  ), // Container
```

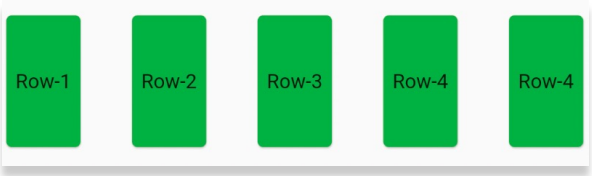
MainAxisAlignment

- start
  - Place the children as close to the start of the main axis as possible
- end
  - Place the children as close to the end of the main axis as possible
- center
  - Place the children as close to the middle of the main axis as possible
- spaceBetween
  - Place the free space evenly between the children
- spaceAround
  - Place the free space evenly between the children as well as half of that space before and after the first and last child
- spaceEvenly
  - Place the free space evenly between the children as well as before and after the first and last child.

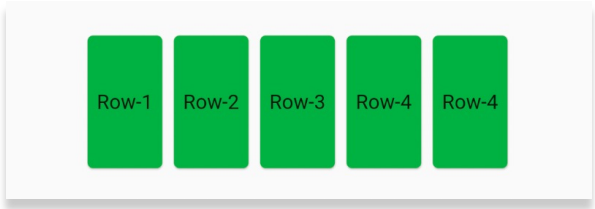
MainAxisAlignment



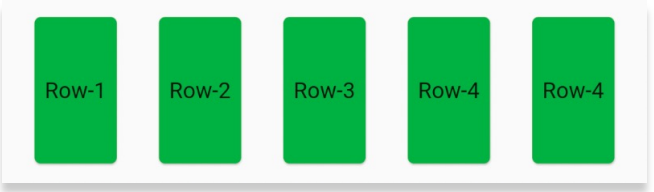
start



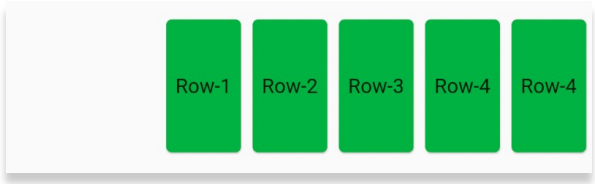
spaceBetween



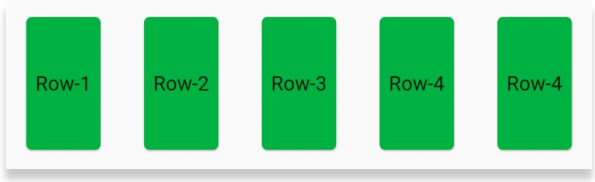
center



spaceEvenly



end

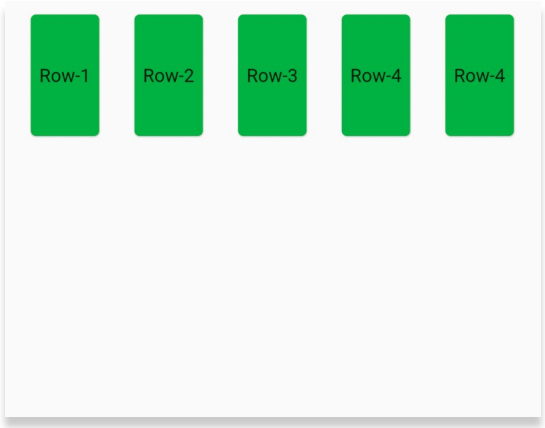


spaceAround

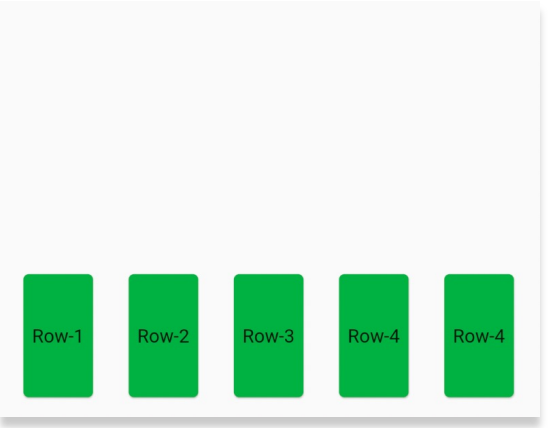
CrossAxisAlignment

- start
  - Place the children with their start edge aligned with the start side of the cross axis
- end
  - Place the children as close to the end of the cross axis as possible.
- center
  - Place the children so that their centers align with the middle of the cross axis
- stretch
  - Place the children along the cross axis such that their baselines match.

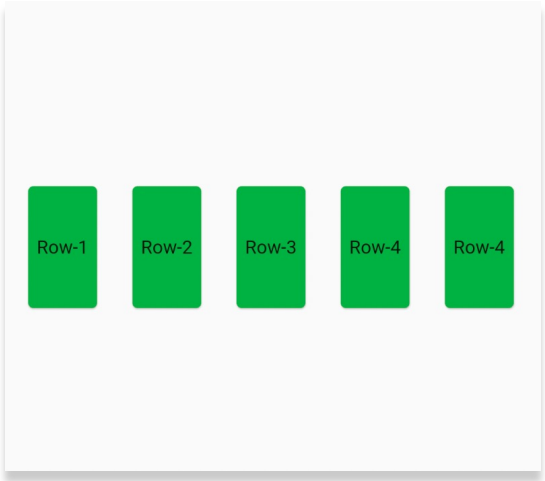
CrossAxisAlignment



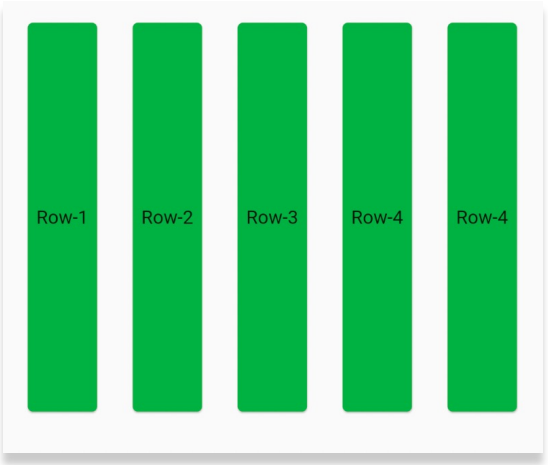
start



end



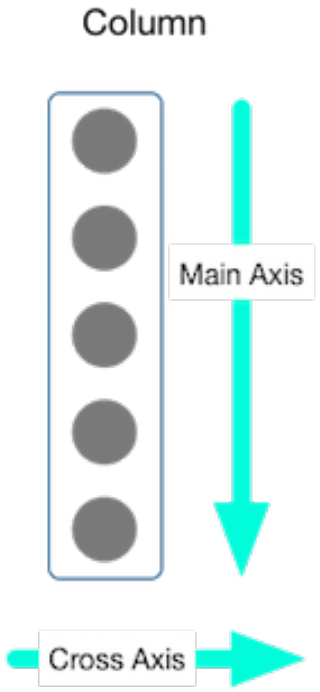
center

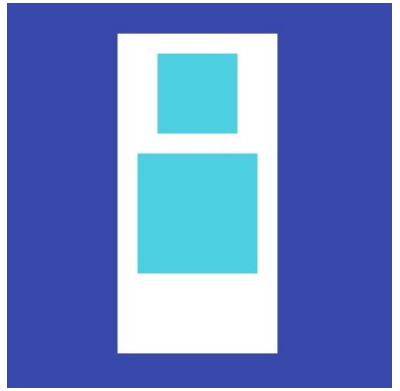


stretch

Column

- Layout a list of child widgets in the vertical direction.





Column Aligning widgets

- For a column, the main axis runs vertically and the cross axis runs horizontally

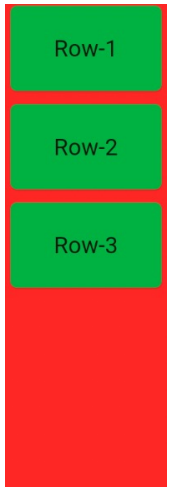
```
@override
Widget build(BuildContext context) {
  return Container(
    child: Column(
      children: [
        Container( // Container
        Container( // Container
        Container( // Container
      ], // Column
    ); // Container
}
```

```
Container(
  width: 100,
  height: 60,
  child: Card(
    color: Colors.green,
    child: Center(child: Text('Row-2')),
    ), // Card
  ), // Container
```

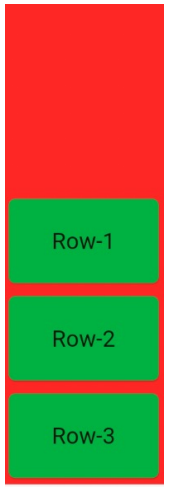
MainAxisAlignment

- start
  - Place the children as close to the start of the main axis as possible
- end
  - Place the children as close to the end of the main axis as possible
- center
  - Place the children as close to the middle of the main axis as possible
- spaceBetween
  - Place the free space evenly between the children
- spaceAround
  - Place the free space evenly between the children as well as half of that space before and after the first and last child
- spaceEvenly
  - Place the free space evenly between the children as well as before and after the first and last child.

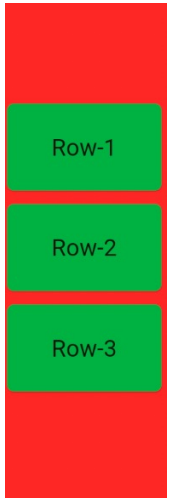
MainAxisAlignment



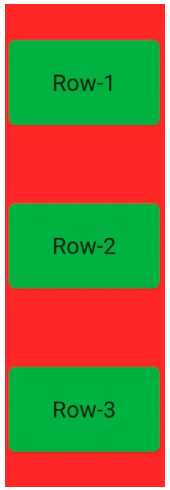
start



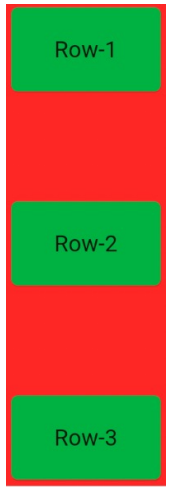
end



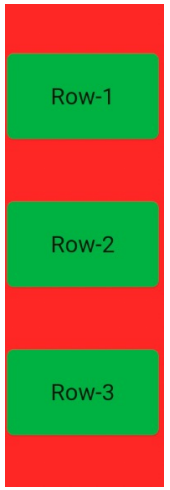
center



spaceAround



spaceBetween

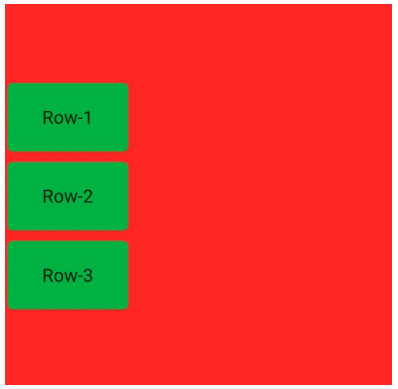


spaceEvenly

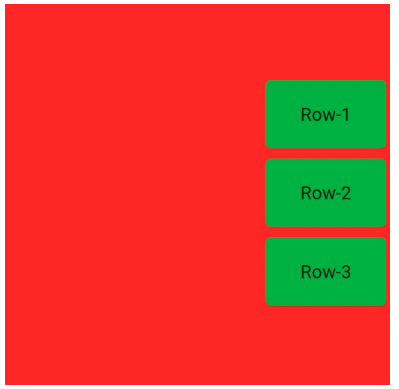
CrossAxisAlignment

- start
  - Place the children with their start edge aligned with the start side of the cross axis
- end
  - Place the children as close to the end of the cross axis as possible.
- center
  - Place the children so that their centers align with the middle of the cross axis
- stretch
  - Place the children along the cross axis such that their baselines match.

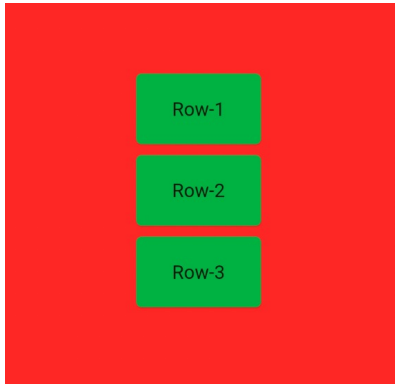
CrossAxisAlignment



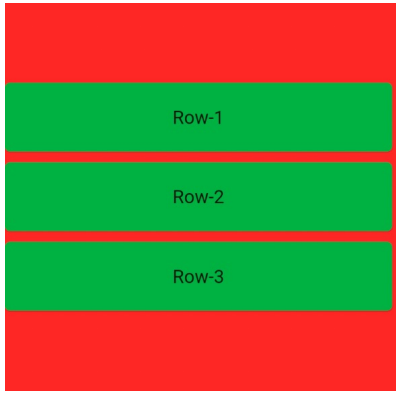
start



end



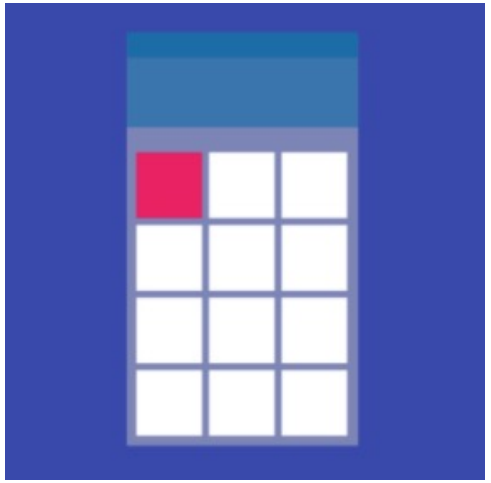
center



stretch

GridView

- A grid list consists of a repeated pattern of cells arrayed in a vertical and horizontal layout. The GridView widget implements this component.



GridView

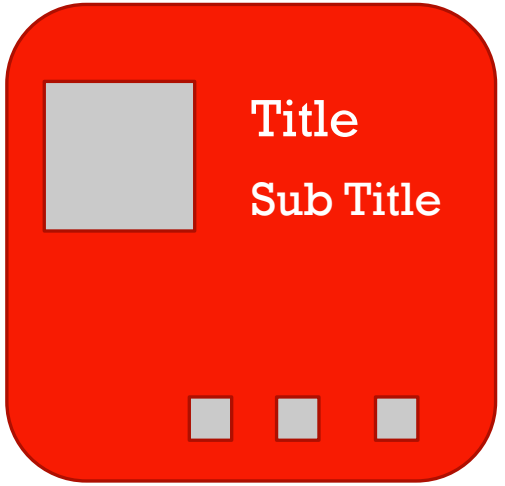
- Demo

```
@override
Widget build(BuildContext context) {
  return Container(
    child: Padding(
      padding: const EdgeInsets.all(20.0),
      child: GridView.count(
        crossAxisCount: 2,
        crossAxisSpacing: 20,
        mainAxisSpacing: 20,
        children: [
          Container( // Container …
          Container( // Container …
          Container( // Container …
          Container( // Container …
        ],
      ), // GridView.count
    ), // Padding
  ); // Container
}
```

```
Container(
  padding: const EdgeInsets.all(8),
  child: const Text("1"),
  color: Colors.teal[100],
), // Container
```

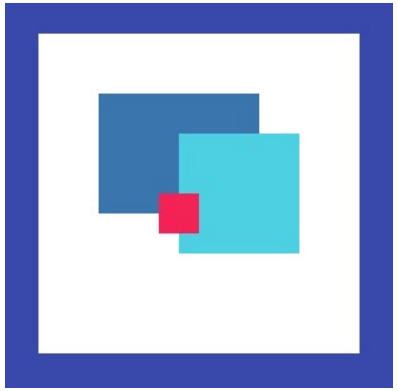
ID: deviceId
deviceName

Workshop



Stack

- This class is useful if you want to overlap several children in a simple way, for example having some text and an image



Stack

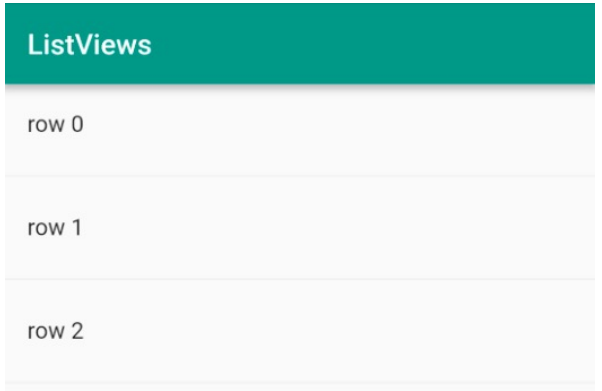
- Demo

```
@override
Widget build(BuildContext context) {
  return Center(
    child: Stack(
      children: <Widget>[
        Container(  // Container …
        Container(  // Container …
        Container(  // Container …
      ], // <Widget>[]
    ), // Stack
  ); // Center
}
```

```
Container(
  width: 120,
  height: 120,
  color: Colors.red,
), // Container
```

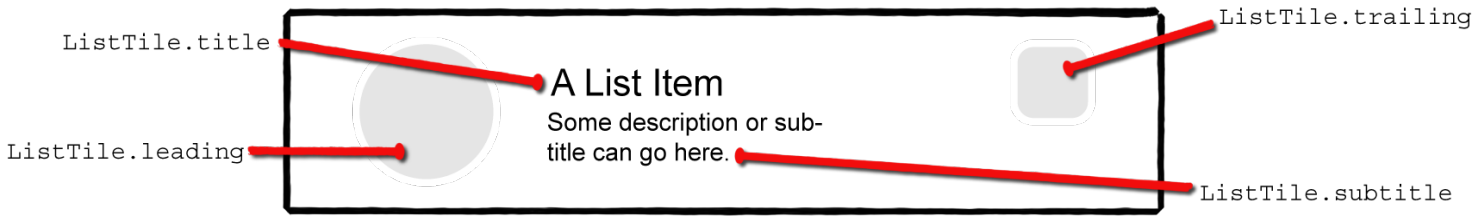
ListView

- A scrollable, linear list of widgets. ListView is the most commonly used scrolling widget. It displays its children one after another in the scroll direction



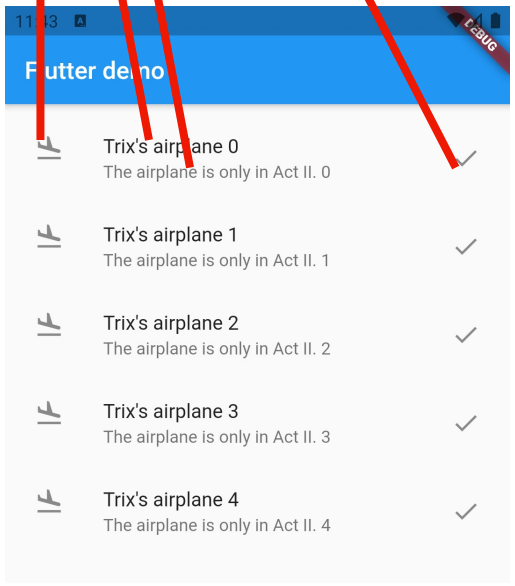
ListTile

- **ListTile** widget is used to populate a ListView in Flutter. It contains title as well as leading or trailing icons.
  - title
    - title to be given to ListTile widget.
  - subtitle
    - additional content displayed below the title.
  - leading
    - leading widget of the ListTile.
  - Trailing
    - trailing widget of the *ListTile*
  - onTap
    - function to be called when the list tile is pressed



ListTile

```dart
@override
Widget build(BuildContext context) {
  return Container(
    child: ListView.builder(
      padding: const EdgeInsets.all(8),
      itemCount: 5,
      itemBuilder: (BuildContext context, int index) {
        return ListTile(
          leading: const Icon(Icons.flight_land),
          title: Text("Trix's airplane $index"),
          subtitle: Text('The airplane is only in Act II. $index'),
          trailing: Icon(Icons.done),
          onTap: () {},
        ); // ListTile
      }), // ListView.builder
  ); // Container
}
```

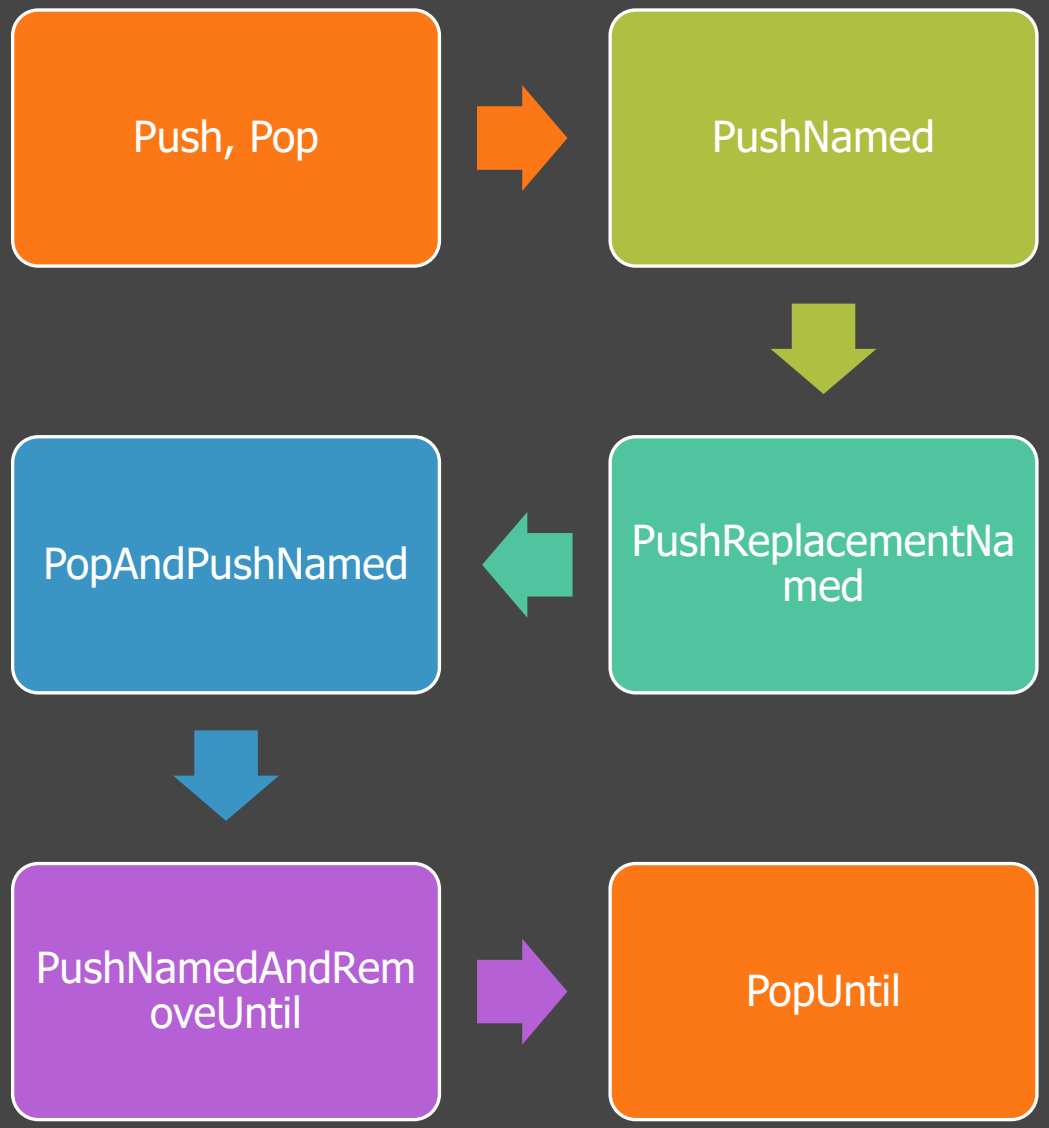


LayoutBuilders

- **LayoutBuilder** helps to create a widget tree in the widget flutter which can depend on the size of the original widget. flutter can take the layout builder as a parameter. It has two parameters. build context and Boxconstrant. BuildContext refers to a widget. But box constraint is more important, it gives the width to the parent widget which is used to manage the child according to the size of the parent

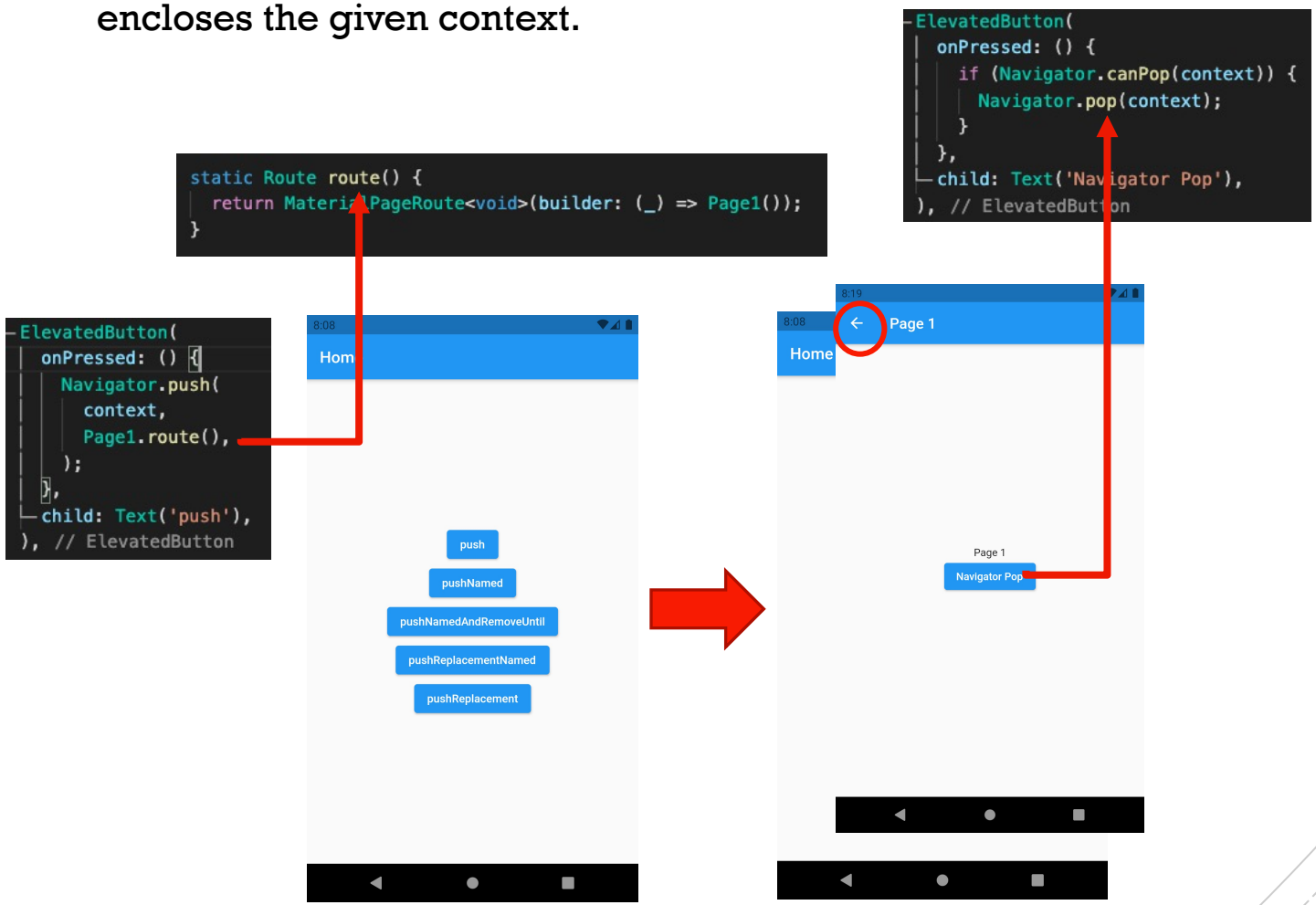
```
Widget build(BuildContext context) {
  return Scaffold(
  ├─appBar: AppBar(title: const Text('LayoutBuilder Example')),
  └─body: LayoutBuilder(
      builder: (BuildContext context, BoxConstraints constraints) {
        if (constraints.maxWidth > 600) {
          return _buildWideContainers();
        } else {
          return _buildNormalContainer();
        }
      },
    ), // LayoutBuilder
  ); // Scaffold
}
```

Navigator


Push, Pop
PushNamed
PushReplacementNamed
PopAndPushNamed
PushNamedAndRemoveUntil
PopUntil

push

- Push the given route onto the navigator that most tightly encloses the given context.

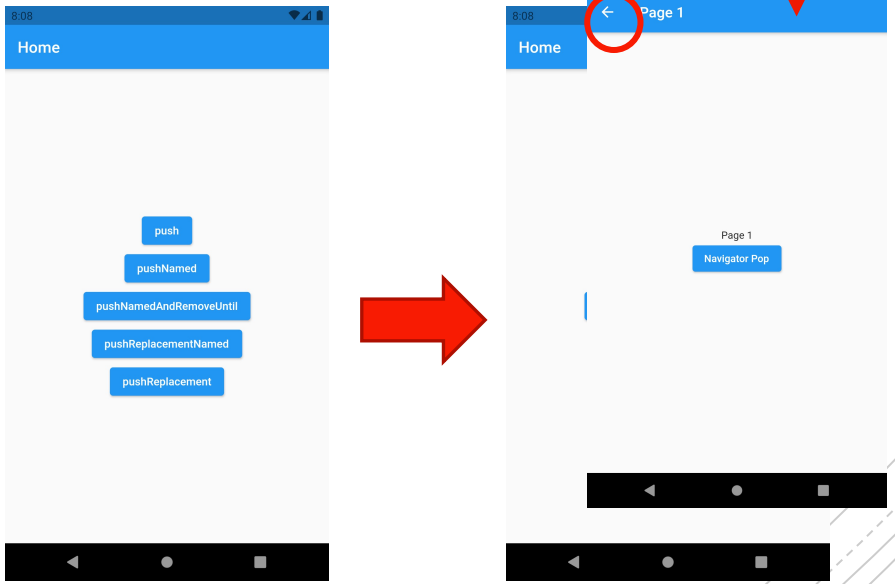


pushNamed

- Push a named route onto the navigator that most tightly encloses the given context.

```dart
@override
Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    initialRoute: '/',
    routes: <String, WidgetBuilder>{
      NavigatorHome.rountName(): (BuildContext context) =>
          const NavigatorHome(),
      Page1.rountName(): (BuildContext context) => const Page1(),
      Page2.rountName(): (BuildContext context) => const Page2(),
      Page3.rountName(): (BuildContext context) => const Page3(),
    },
  ); // MaterialApp
}
```

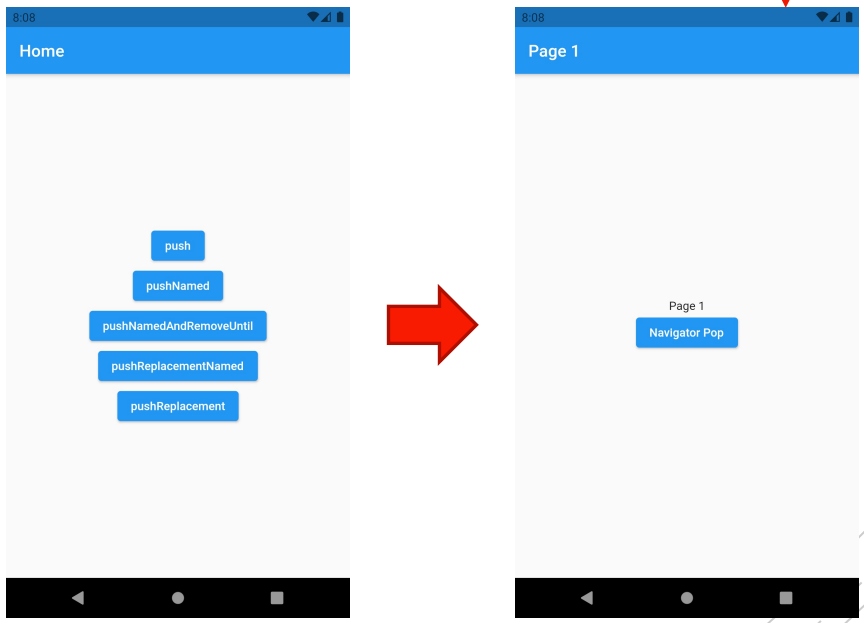
```dart
ElevatedButton(
  onPressed: () {
    Navigator.pushNamed(context,
        Page1.rountName());
  },
  child: Text('pushNamed'),
), // ElevatedButton
```



pushReplacementNamed

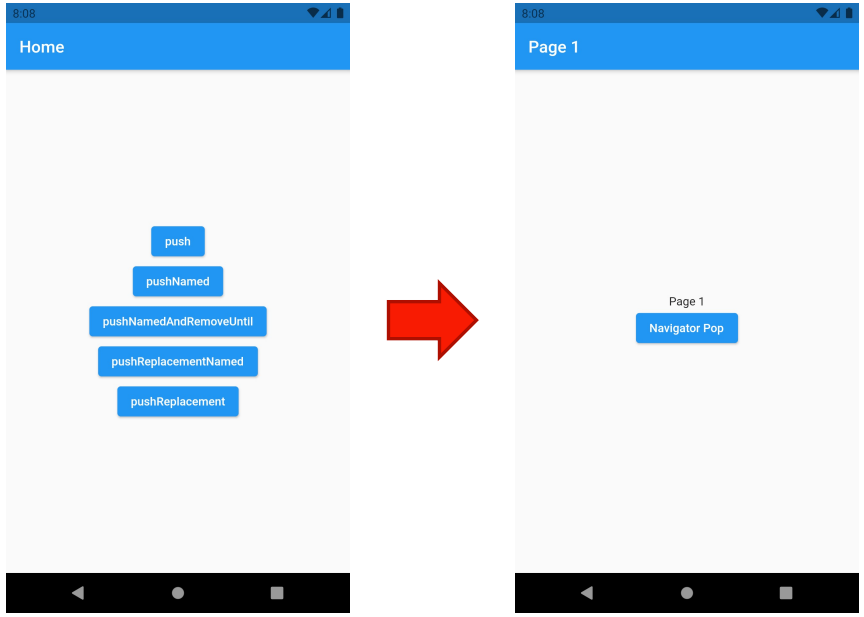
- Replace the current route of the navigator that most tightly encloses the given context by pushing the route named routeName and then disposing the previous route once the new route has finished

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    initialRoute: '/',
    routes: <String, WidgetBuilder>{
      NavigatorHome.rountName(): (BuildContext context) =>
          const NavigatorHome(),
      Page1.rountName(): (BuildContext context) => const Page1(),
      Page2.rountName(): (BuildContext context) => const Page2(),
      Page3.rountName(): (BuildContext context) => const Page3(),
    },
  ); // MaterialApp
}
```



pushReplacement

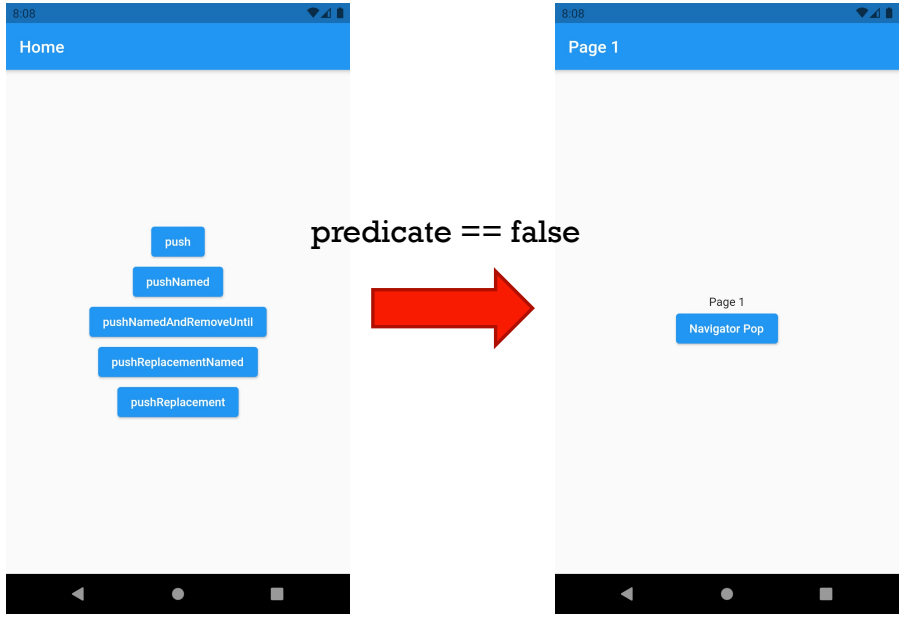
- Replace the current route of the navigator that most tightly encloses the given context by pushing the given route and then disposing the previous route once the new route has finished



pushAndRemoveUntil

- Push the given route onto the navigator that most tightly encloses the given context, and then remove all the previous routes until the predicate returns true

```
ElevatedButton(
  onPressed: () {
    Navigator.pushAndRemoveUntil(context,
        Page1.route(),
        ModalRoute.withName(Page1.rountName()));
  },
  child: Text('pushAndRemoveUntil'),
), // ElevatedButton
```

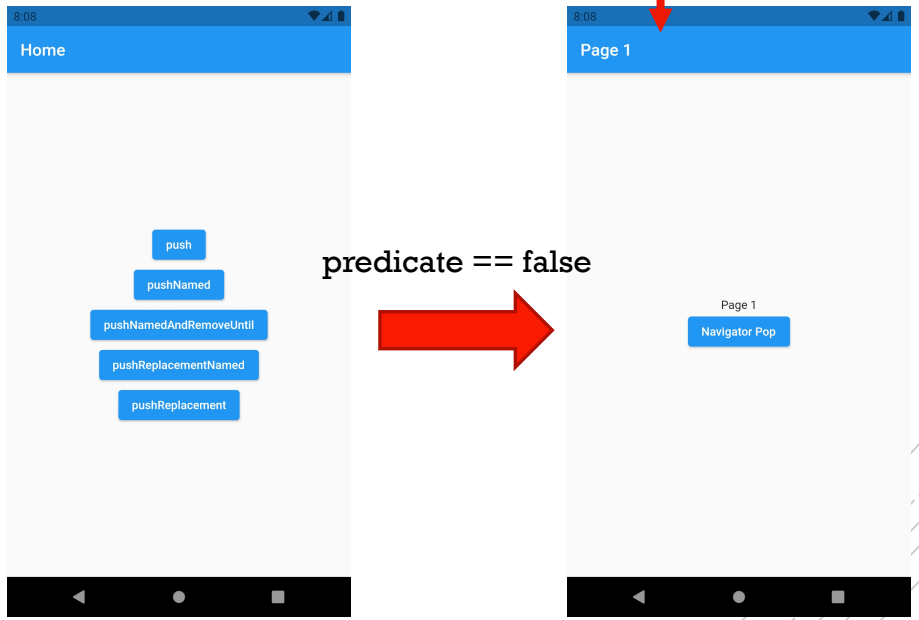


pushNamedAndRemoveUntil

- Push the route with the given name onto the navigator that most tightly encloses the given context, and then remove all the previous routes until the predicate returns true

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    initialRoute: '/',
    routes: <String, WidgetBuilder>{
      NavigatorHome.rountName(): (BuildContext context) =>
          const NavigatorHome(),
      Page1.rountName(): (BuildContext context) => const Page1(),
      Page2.rountName(): (BuildContext context) => const Page2(),
      Page3.rountName(): (BuildContext context) => const Page3(),
    },
  ); // MaterialApp
}
```

```
ElevatedButton(
  onPressed: () {
    Navigator.pushNamedAndRemoveUntil(
        context, Page1.rountName(),
        ModalRoute.withName(Page1.rountName()));
  },
  child: Text('pushNamedAndRemoveUntil'),
), // ElevatedButton
```



arguments to a named route

1. Define the arguments you need to pass.
2. Create a widget that extracts the arguments.
3. Register the widget in the routes table.
4. Navigate to the widget

1

```
class ScreenArguments {
  final String title;
  final String message;
  ScreenArguments(this.title, this.message);
}
```

2

```
final args = ModalRoute.of(context)!.settings.arguments as ScreenArguments;
```

3

```
routes: <String, WidgetBuilder>{
  NavigatorHome.rountName(): (BuildContext context) =>
      const NavigatorHome(),
  Page1.rountName(): (BuildContext context) => const Page1(),
  Page2.rountName(): (BuildContext context) => const Page2(),
  Page3.rountName(): (BuildContext context) => const Page3(),
```

4

```
ElevatedButton(
  onPressed: () async {
    var result = await Navigator.pushNamed(context, '/page3',
        arguments:
            ScreenArguments('pushNamed Args', 'Hello World'));
    print(result);
  },
  child: Text('pushNamed Args'),
```

Navigate pass data

1. Define Route
2. Navigate and pass data

1

```dart
static Route<String> route(String? title) {
  return MaterialPageRoute<String>(builder: (_)
      => Page2(title: title)); // MaterialPageRoute
}
```

2

```dart
ElevatedButton(
  onPressed: () async {
    var result = await Navigator.push<String>(
      context,
      Page2.route('Hello World'),
    );
    print(result);
  },
  child: Text('push'),
```

Pop and return data

1. await Navigate to route
2. pop and return data

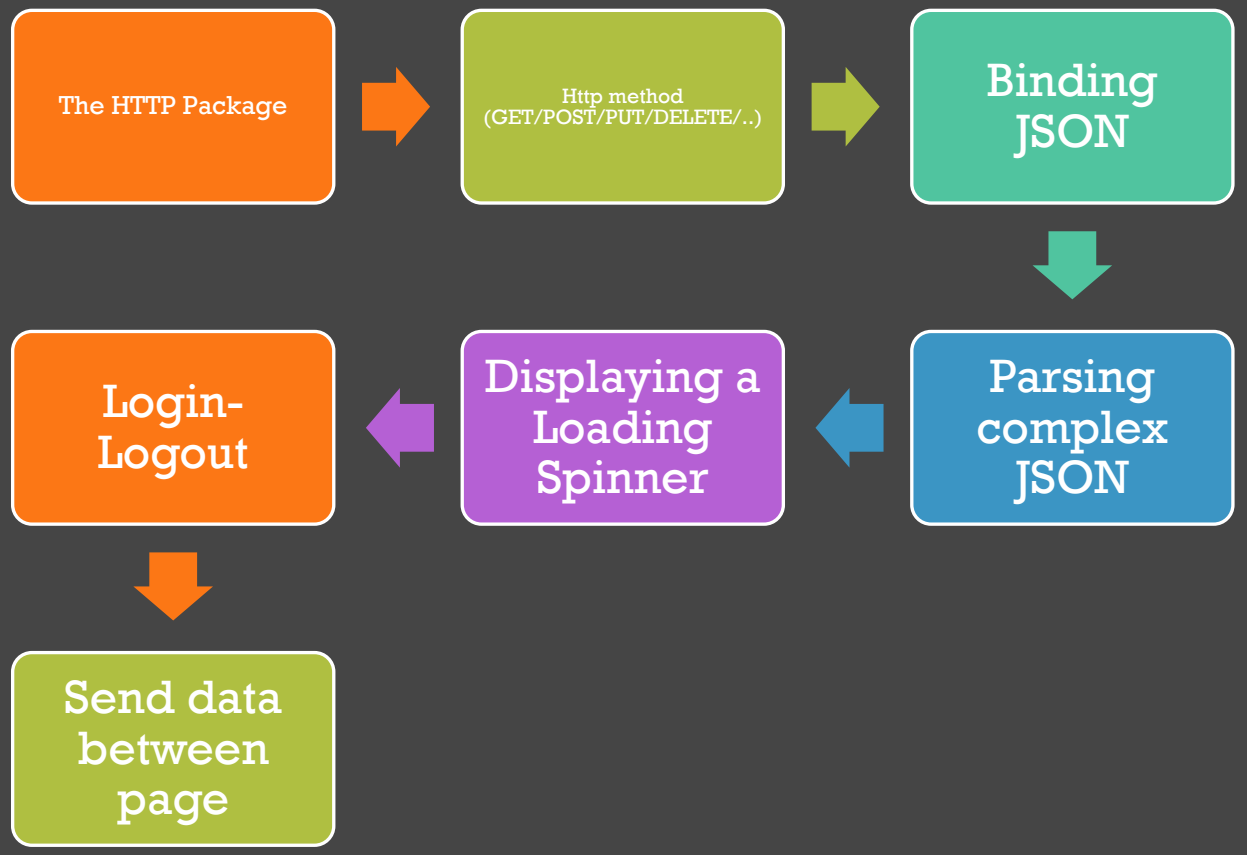
1

```
ElevatedButton(
  onPressed: () async {
    var result = await Navigator.push<String>(
      context,
      Page2.route('Hello World'),
    );
    print(result);
  },
  child: Text('push'),
), // ElevatedButton
```

2

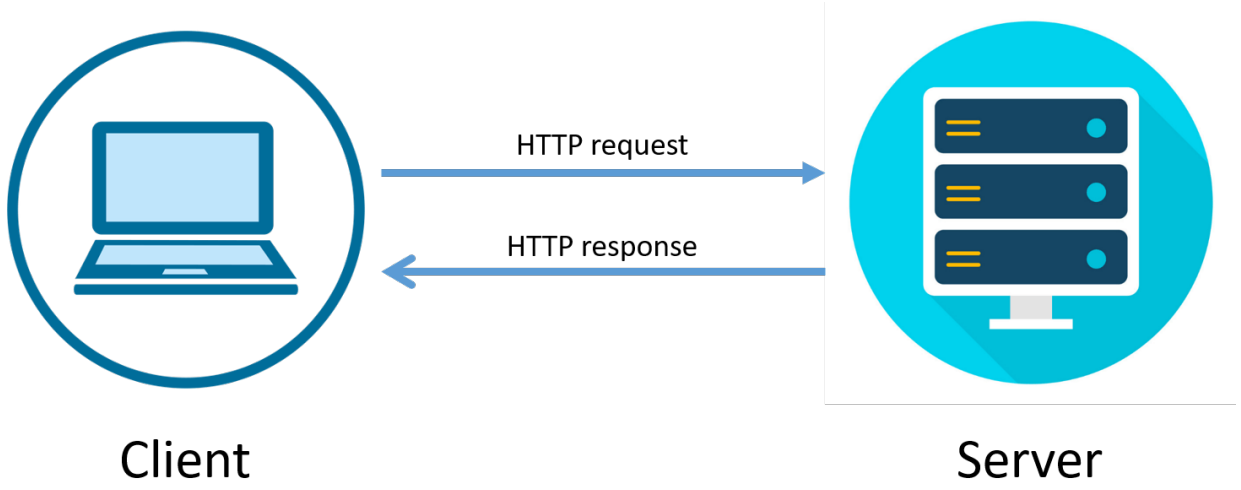
```
ElevatedButton(
  onPressed: () {
    if (Navigator.canPop(context)) {
      Navigator.pop<String>(context, 'Result: ${args.message}');
    }
  },
  child: Text('Navigator Pop'),
), // ElevatedButton
```

JSON RESTful Feed Workshop

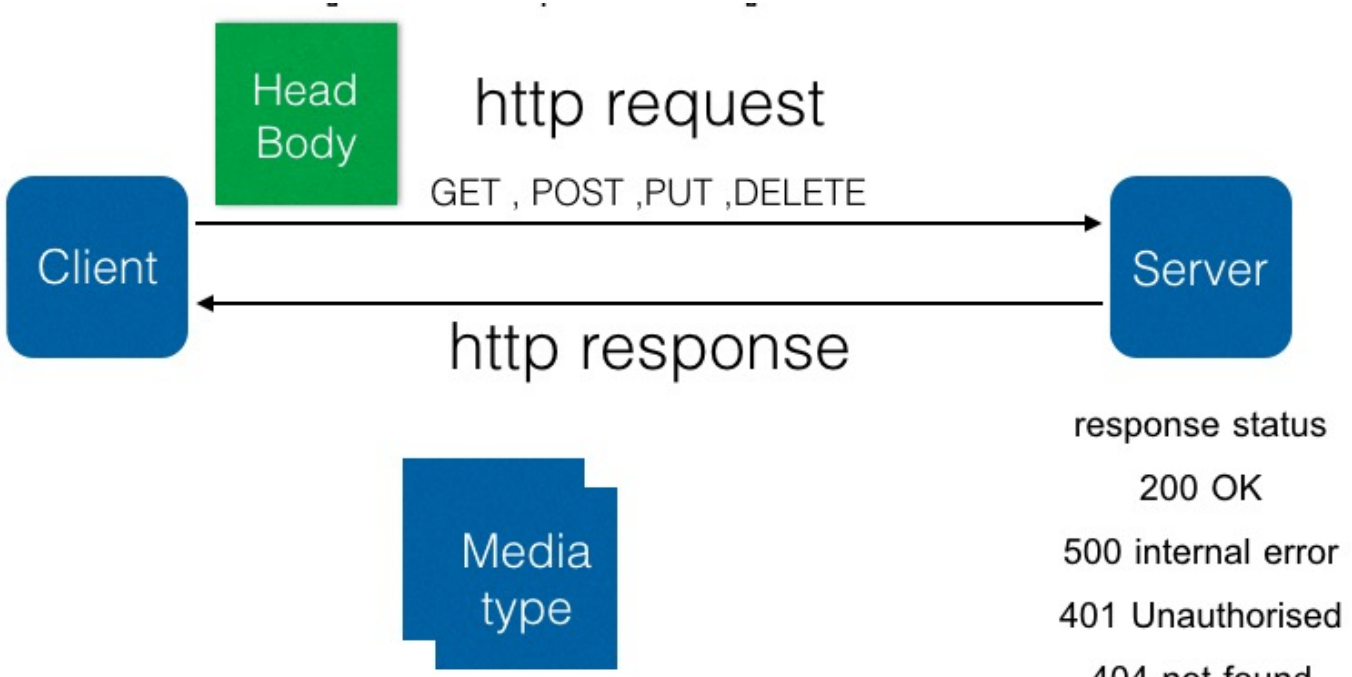


RESTful API

- REST
  - **Re**presentational **S**tate **T**ransfer
- API
  - **A**pplication **P**rogram **I**nterface
- HTTP Web Protocol



HTTP Web Protocol



HTTP Method

- GET
  - The GET method requests a representation of the specified resource. Requests using GET should only retrieve data
- POST
  - The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server
- PUT
  - The PUT method replaces all current representations of the target resource with the request payload.
- DELETE
  - The DELETE method deletes the specified resource.
- etc.

HTTP response status codes

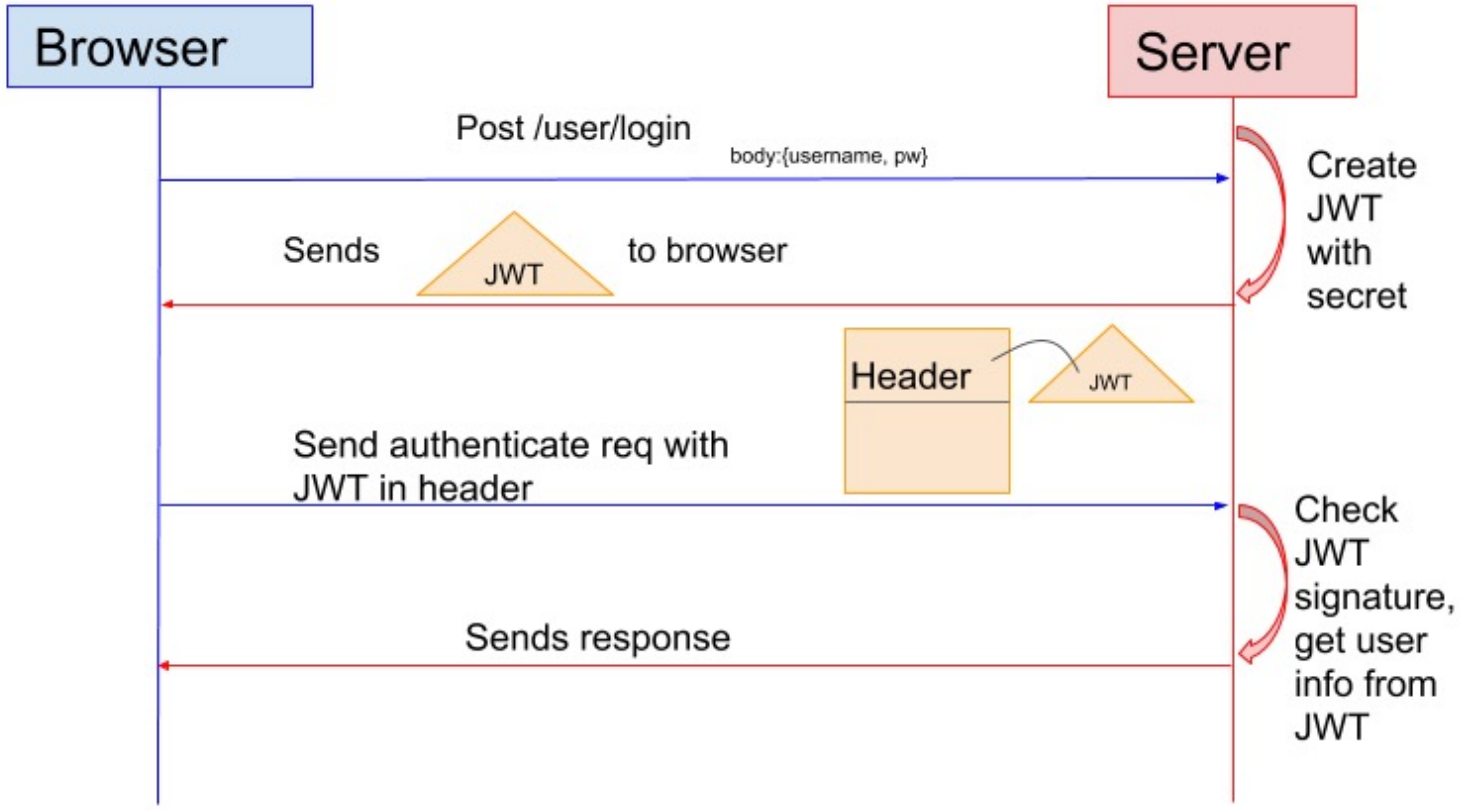
- 200 – Ok
  - The request has succeeded. The meaning of the success depends on the HTTP method
- 201 - Create
  - The request has succeeded and a new resource has been created as a result. This is typically the response sent after POST requests, or some PUT requests.
- 400 – Bad Request
  - The server could not understand the request due to invalid syntax.
- 401 Unauthorized
  - Although the HTTP standard specifies "unauthorized", semantically this response means "unauthenticated". That is, the client must authenticate itself to get the requested response.
- 404 Not Found
- 500 Internal Server Error
  - The server has encountered a situation it doesn't know how to handle.

JWT

- JWT
  - **J**SON **W**eb **T**oken. is an open standard (**RFC 7519**) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.
- JSON Web Token structure
  - In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:
    - Header
    - Payload
    - Signature

```
<base64url-encoded header>.<base64url-encoded payload>.<base64url-encoded signature>
```

Stateless Authentication



NodeJS API Service

- Install NodeJs
  - https://nodejs.org/en/download/
- Install Node Package
  - express
  - cors
  - dotenv
  - Morgan
  - nodemon
    - **npm** install express cors dotenv morgan –save
    - **npm** install nodemon -D

Node App Module

```
const express = require('express')
const cors = require('cors')
const logger = require('morgan')
const app = express()
require("dotenv").config()

app.use(cors());
app.use(logger('dev'));
app.use(express.json());
app.use(function (err, req, res, next) {
    console.error(err.stack)
    res.status(err.status || 500).send({
        message: err.message,
        result: false
    })
})

app.get('/test', (req, res) => {
    return res.send('Received a GET HTTP method');
})
app.post('/test', (req, res) => {
    return res.send('Received a POST HTTP method');
})

app.put('/test', (req, res) => {
    return res.send('Received a PUT HTTP method');
})

app.delete('/test', (req, res) => {
    return res.send('Received a DELETE HTTP method');
})

module.exports = app;
```

Package Script

- Edit package.json

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "nodemon ./bin/www.js"
},
```

- Run script
  - npm start

Flutter Http Library

- http 0.13.3
  - https://pub.dev/packages/http
- Install dependencies
  - pubspec.yaml

```yaml
dependencies:
  flutter:
    sdk: flutter


  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.2
  http: ^0.13.3
  mqtt_client: ^9.5.0
  uuid: ^3.0.4
  sprintf: ^6.0.0
  sleek_circular_slider: ^2.0.1
```

HTTP Methods

- **HTTP GET**
  - Use GET requests to retrieve resource representation/information only – and not to modify it in any way
- **HTTP POST**
  - Use POST APIs to create new subordinate resources, e.g., a file is subordinate to a directory containing it or a row is subordinate to a database table.
- **HTTP PUT**
  - Use PUT APIs primarily to update existing resource (if the resource does not exist, then API may decide to create a new resource or not). If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and if an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request
- **HTTP DELETE**
  - As the name applies, DELETE APIs are used to delete resources (identified by the Request-URI).
  - A successful response of DELETE requests SHOULD be HTTP response code 200 (OK) if the response includes an entity describing the status, 202 (Accepted) if the action has been queued, or 204 (No Content) if the action has been performed but the response does not include an entity

```
factory Uri.http(String authority, String unencodedPath,[Map<String, dynamic>? queryParameters])
```

Create URI

- authority
  - Base API URL
- unencodedPath
  - API Path
- Import Library
  - import 'package:http/http.dart' as http;

```
var url = Uri.https('www.googleapis.com', '/books/v1/volumes');
```

JSON encode & decode

- import
  - 'dart:convert'
- decode => Map
  - json.decode('<jsonformat>')
- encode => String
  - json.decode(Map<String, dynamic>)

Map To Object

```
factory GpioModel.fromJson(Map<String, dynamic> json) {
  return GpioModel(
    index: json['index'],
    title: json['title'],
    subTitle: json['subTitle'],
    state: json['state'] == 1,
  );
}
```

Object To Map

```
Map<String, dynamic> toJson() {
  return {
    "id": this.id != null ? this.id : -1,
    "userId": this.userId,
    "index": this.index,
    "title": this.title,
    "subTitle": this.subTitle,
    "state": this.state
  };
}
```

http.get

```
Future<Response> get(Uri url, {Map<String, String>? headers})
```

- url

```
var url = Uri.https('www.googleapis.com', '/books/v1/volumes');
```

- headers

```
var _headers = {
  'Content-type': 'application/json; charset=utf-8',
  'Accept': 'application/json',
  'x-token': 'jwtToken'
};
```

- Response
  - statusCode
  - Body

```
Future<List<Map<String, dynamic>>> getDevices() async {
  var url = Uri.https('www.googleapis.com', '/books/v1/volumes');
  var _headers = {
    'Content-type': 'application/json; charset=utf-8',
    'Accept': 'application/json',
    'x-token': 'jwtToken'
  };
  final response = await http.get(url, headers: _headers);
  if (response.statusCode == 200) {
    Map<String, dynamic> _dataMap =
        Map<String, dynamic>.from(json.decode(response.body));

    List<Map<String, dynamic>> list =
        List<Map<String, dynamic>>.from(_dataMap['data']);
    return list;
  }
  return [];
}
```

http.post

```
Future<Response> get(Uri url, {Map<String, String>? headers})
```

- url

```
var url = Uri.https('www.googleapis.com', '/books/v1/volumes');
```

- headers

```
var _headers = {
  'Content-type': 'application/json; charset=utf-8',
  'Accept': 'application/json',
  'x-token': 'jwtToken'
};
```

- Response
  - statusCode
  - Body

```
Future<GpioModel?> postDevice(GpioModel gpio) async {
  var url = Uri.http(API_BASE_URL, '$API_VERSION/device');
  var _headers = {
    'Content-type': 'application/json; charset=utf-8',
    'Accept': 'application/json',
    'x-access-token': TEST_TOKEN
  };
  gpio.userId = ID;
  var body = json.encode(gpio.toJson());
  final response = await http.post(url, body: body, headers: _headers);
  Map<String, dynamic>? _dataMap;
  if (response.statusCode == 201) {
    _dataMap = Map<String, dynamic>.from(json.decode(response.body));
    if (_dataMap['result'] == true) {
      return GpioModel.fromJson(_dataMap['data']);
    }
  } else if (response.statusCode == 400) {
    _dataMap = Map<String, dynamic>.from(json.decode(response.body));
    print(_dataMap['data']);
    throw Exception(_dataMap['data']);
  } else if (response.statusCode == 500) {
    _dataMap = Map<String, dynamic>.from(json.decode(response.body));
    print(_dataMap['data']);
    throw Exception(_dataMap['data']);
  }
  return null;
}
```

http.delete

```
Future<Response> get(Uri url, {Map<String, String>? headers})
```

- url

```
var url = Uri.https('www.googleapis.com', '/books/v1/volumes');
```

- headers

```
var _headers = {
  'Content-type': 'application/json; charset=utf-8',
  'Accept': 'application/json',
  'x-token': 'jwtToken'
};
```

- Response
  - statusCode
  - Body

```
Future<Map<String, dynamic>?> deleteDevice(GpioModel gpio) async {
  var url = Uri.http(API_BASE_URL, '$API_VERSION/device');
  var _headers = {
    'Content-type': 'application/json; charset=utf-8',
    'Accept': 'application/json',
    'x-access-token': TEST_TOKEN
  };
  var body = json.encode({'userId': ID, 'index': gpio.index!});
  final response = await http.delete(url, body: body, headers: _headers);
  if (response.statusCode == 200) {
    Map<String, dynamic> _dataMap =
        Map<String, dynamic>.from(json.decode(response.body));
    return _dataMap;
  }
  return null;
}
```

http.update

```
Future<Response> get(Uri url, {Map<String, String>? headers})
```

- url

```
var url = Uri.https('www.googleapis.com', '/books/v1/volumes');
```

- headers

```
var _headers = {
  'Content-type': 'application/json; charset=utf-8',
  'Accept': 'application/json',
  'x-token': 'jwtToken'
};
```

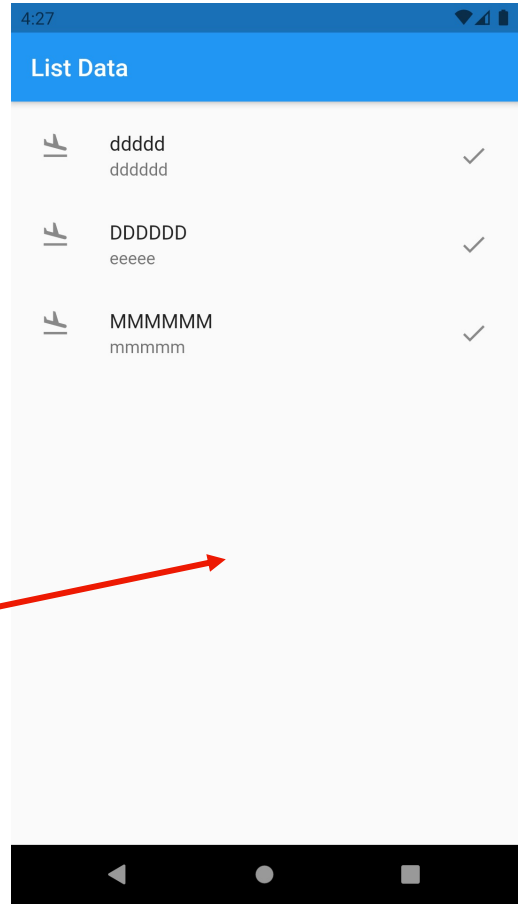
- Response
  - statusCode
  - Body

```
Future<Map<String, dynamic>?> updateDevice(GpioModel gpio) async {
  var url = Uri.http(API_BASE_URL, '$API_VERSION/device');
  var _headers = {
    'Content-type': 'application/json; charset=utf-8',
    'Accept': 'application/json',
    'x-access-token': TEST_TOKEN
  };
  gpio.userId = ID;
  var body = json.encode(gpio.toJson());
  final response = await http.put(url, body: body, headers: _headers);
  if (response.statusCode == 200) {
    Map<String, dynamic> _dataMap =
        Map<String, dynamic>.from(json.decode(response.body));
    return _dataMap;
  }
  return null;
}
```

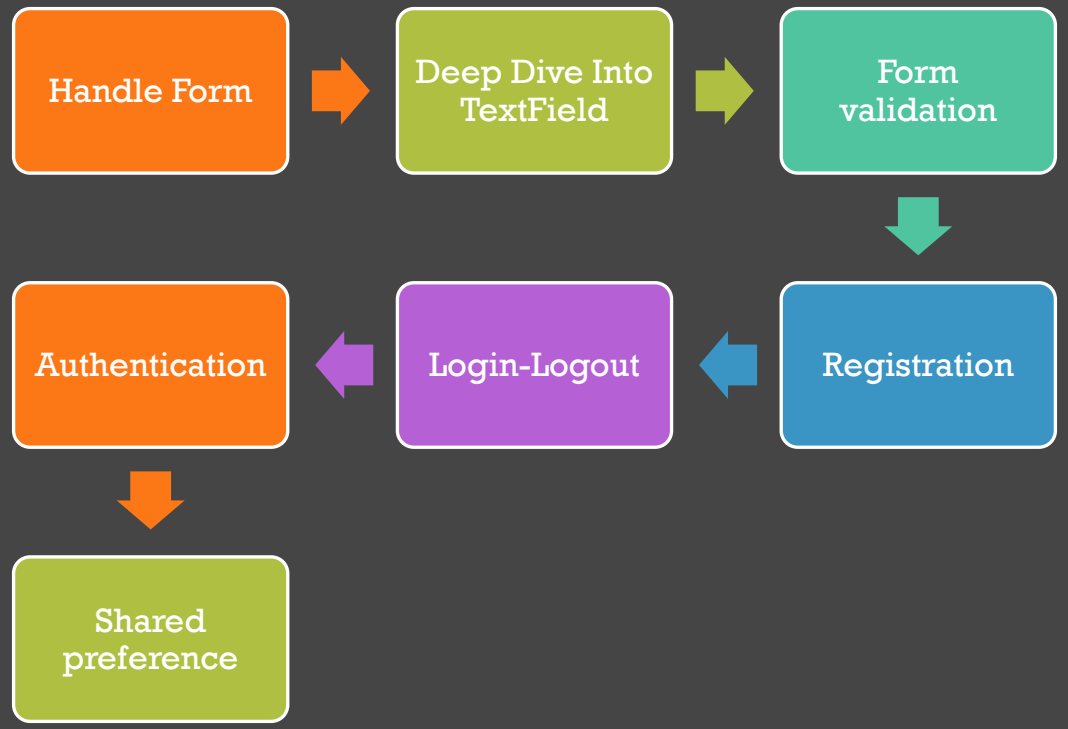
Load Data To List

```dart
Future<List<GpioModel>> getDate() async {
  await Future<void>.delayed(Duration(seconds: 1));
  List<GpioModel> _innerList = await Api().getDevices();
  return _innerList;
}
```

```dart
child: FutureBuilder(
  future: getDate(),
  builder:
      (BuildContext context,
       AsyncSnapshot<List<GpioModel>> snapshot) {
    if (snapshot.hasData) {
      List<GpioModel> _innerList = snapshot.data!;
      return Center(
        child: ListView.builder( // ListView.builder // Center …
    } else {
      return Center( // Center …
    }
  },
), // FutureBuilder
```

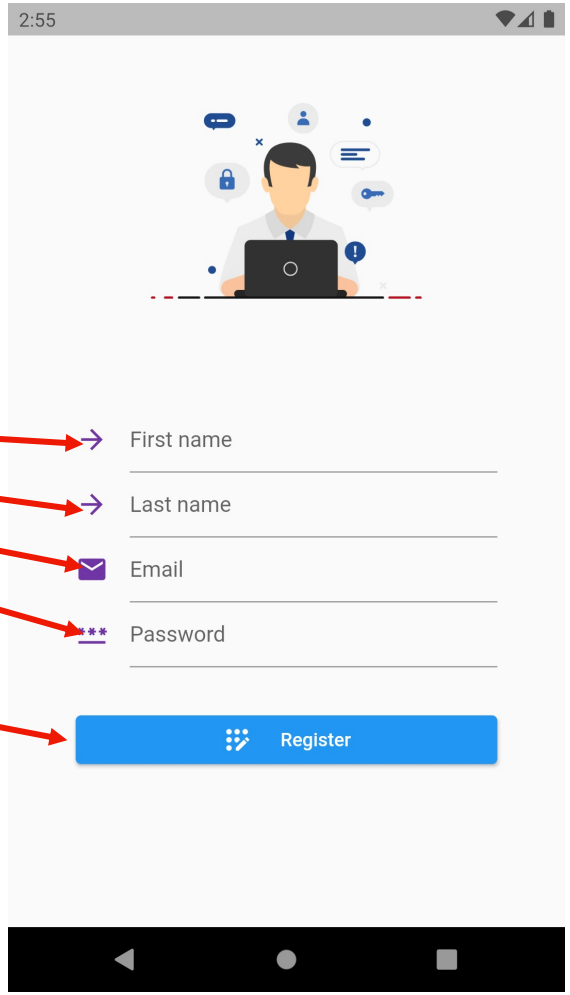


Login & Register Workshop



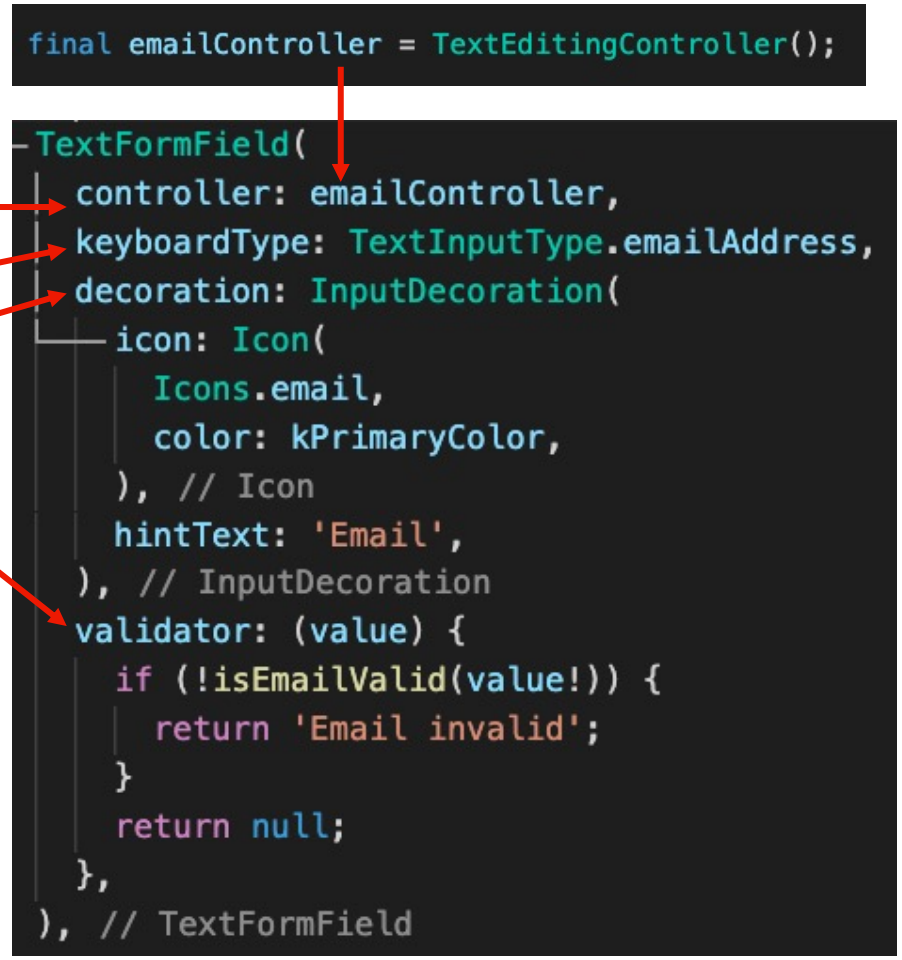
Handle Form

```
Form(
  key: _formKey,
  child: Column(
    children: [
      TextFormField( // TextFormField …
      TextFormField( // TextFormField …
      TextFormField( // TextFormField …
      TextFormField( // TextFormField …
      SizedBox( // SizedBox …
      ElevatedButton( // Row // ElevatedButton …
    ],
  ), // Column
) // Form
```



Deep Dive Into TextFormField

- TextFormField
  - controller
  - keyboardType
  - decoration
  - validator

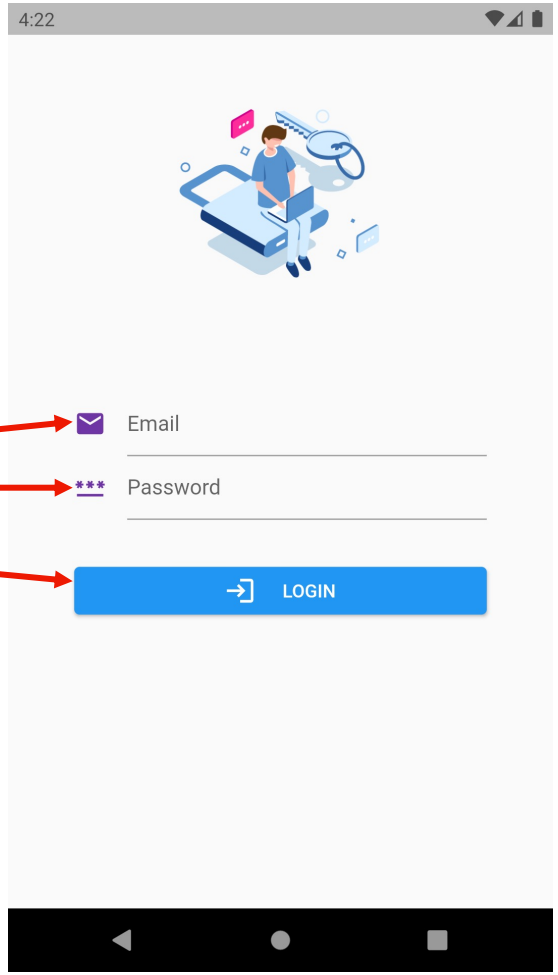


TextInputType

- text
- multiline
- number
- phone
- datetime
- emailAddress
- url
- visiblePassword
- name
- streetAddress

Login

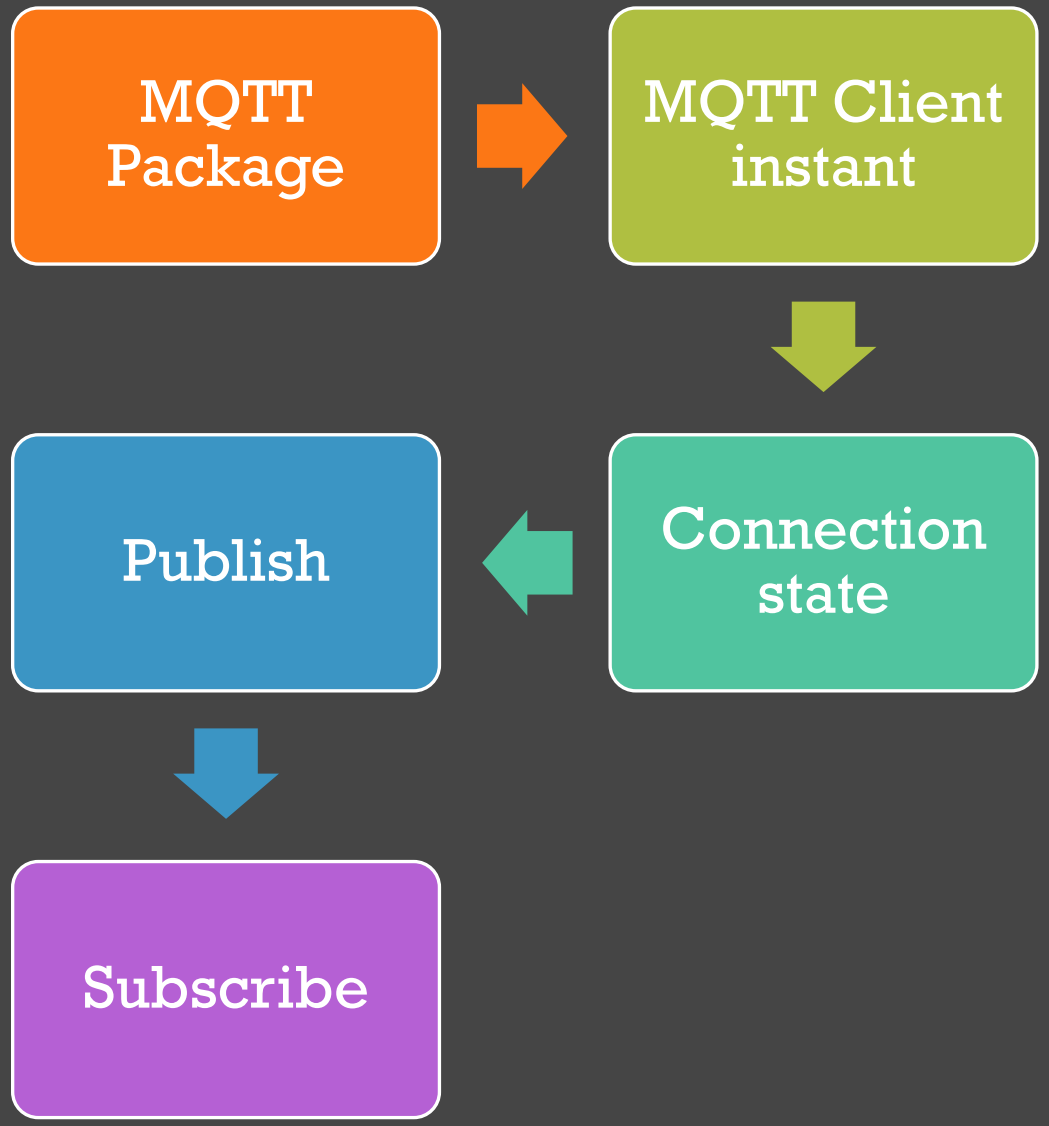
```
Form(
  key: _formKey,
  child: Column(
    children: [
      TextFormField( // TextFormField …
      TextFormField( // TextFormField …
      SizedBox( // SizedBox …
      ElevatedButton( // Row // ElevatedButton …
    ],
  ), // Column
) // Form
```



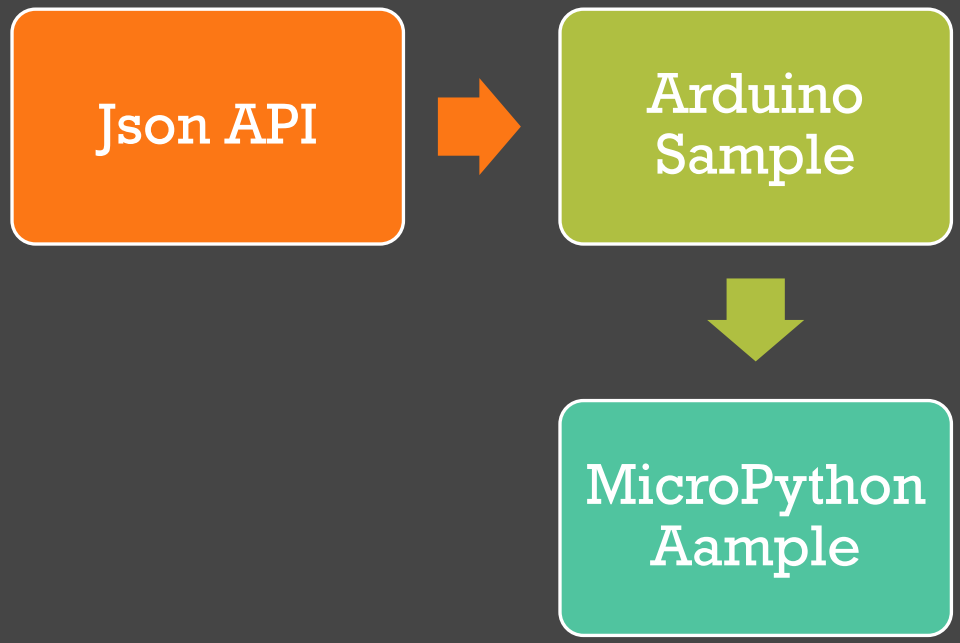
Authentication

Shared preference

Realtime Data By MQTT Workshop



Hardware Control By MQTT Workshop



Publishing App