

# Chapter 1.1

Introduction to Mobile Application Development

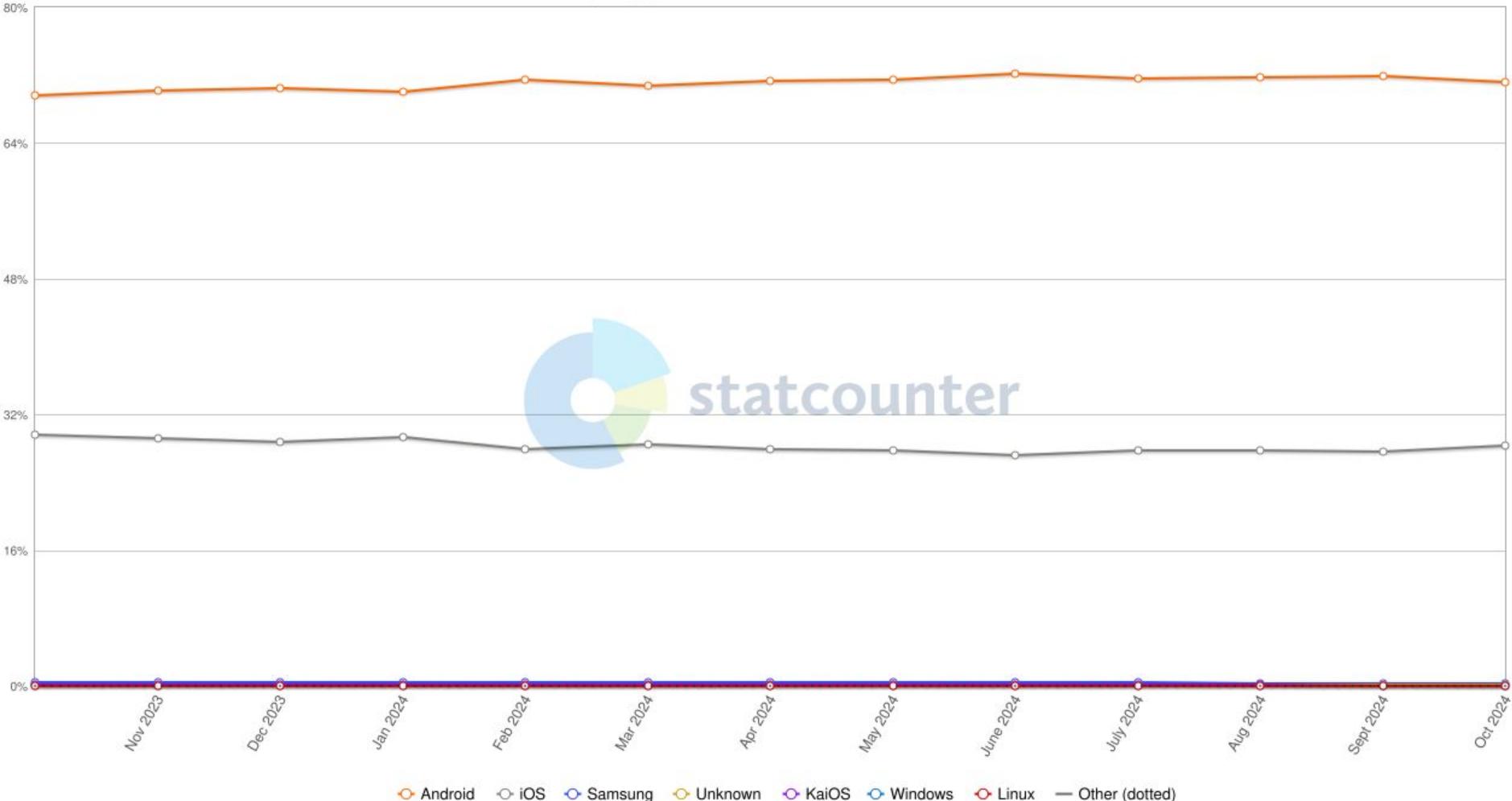
# Objectives

- Mobile operating systems and ecosystem
- Development tools
- Mobile devices
- Mobile application software architecture and framework

# Mobile Operating Systems And Ecosystem

# Mobile Operating Systems

StatCounter Global Stats  
Mobile Operating System Market Share Worldwide from Oct 2023 - Oct 2024



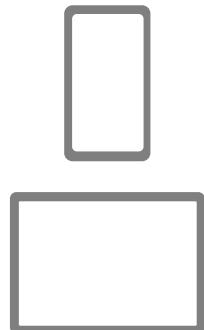
# Mobile OS



# Mobile OS - Android



Linux kernel  
Open Source



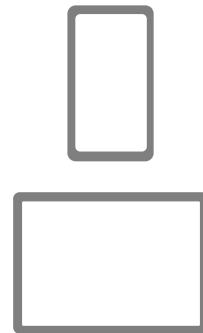
Chrome OS  
Linux-based  
Open Source



# Mobile OS - iOS



Unix kernel



macOS



# Mobile OS - Tizen



Linux kernel

Open Source

# Mobile OS - KaiOS



Linux kernel

Open Source



# Mobile OS – Harmony OS



Microkernel

Open Source



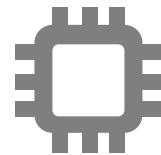
# Mobile OS - Fuchsia



Microkernel

Open Source

Module Architecture



ARM-based



Nest Hub

# Types of Mobile Apps

Native



Hybrid

Web  
+  
Native

Web



# Comparison of Apps

	Native	Hybrid	Mobile Web
Cost	High	Low	Low
Performance	Fast	Depending on speed of network	Depending on speed of network
Distribution	App stores	App stores	None
Device features	Wide	Limited	Very limited
Code maintenance	Multiple codebase	Single codebase	Single codebase

# Question?

According to Medium, the four market moving mobile app trends:

1. App store consumer spending will surpass USD 120 billion globally
2. Mobile gaming market share will increase to 60%
3. Users will spend 10 minutes of every hour consuming streaming video on a mobile device
4. Publishers will produce 60% more apps with in-app ads, making up 62% of the USD 250 billion global digital ad spend

In your opinion, which mobile app development method (native, mobile web, and hybrid) is suitable for each the following entities?

- a. Bernama, the Malaysian national news-agency
- b. Super Mario Bros, a game developed by Sega
- c. Setapak Central, a shopping mall

# Development Tools

# Programming Languages

Native



Hybrid

Web  
+  
Native

Web



# Development Tools - Native



Android  
Studio



XCod  
e



Tizen  
Studio

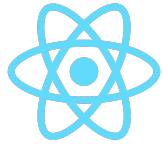


Visual  
Studio



Huawei Quick  
App

# Development Tools – Cross Platform



React Native



Flutter



Xamarin



PhoneGap



Apache Weex



Native Script

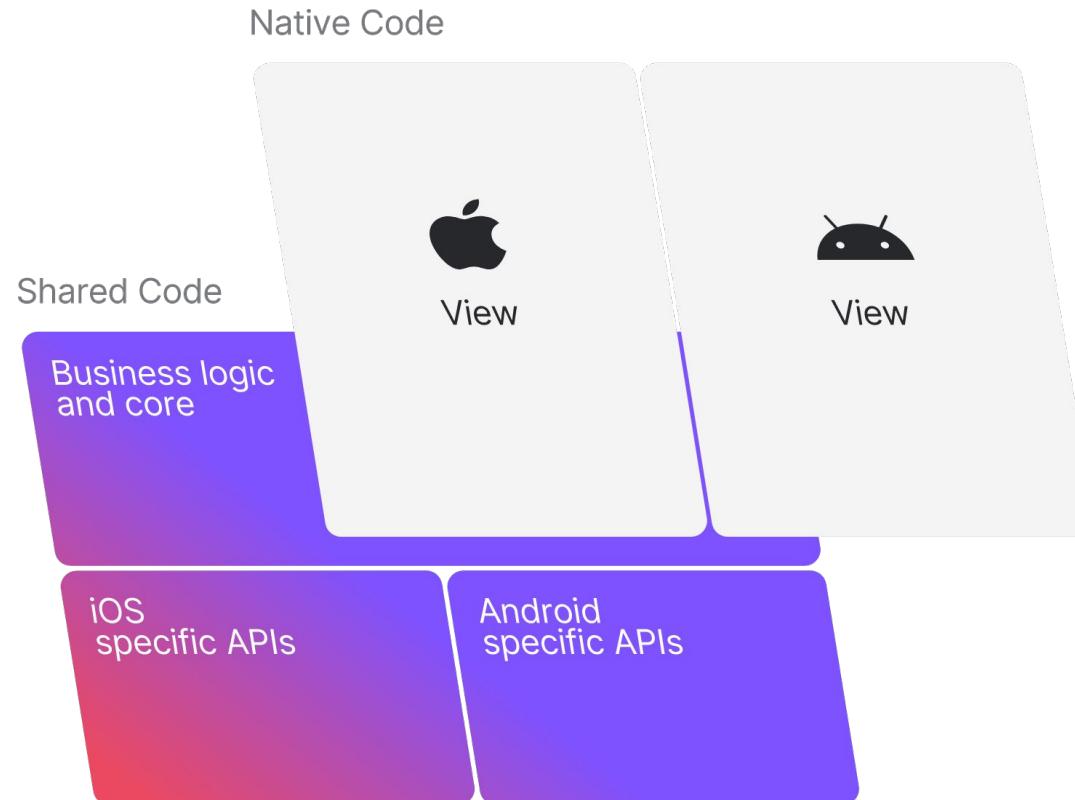


Jasonette



Ionic

# Kotlin Multiplatform Mobile (KMM)



# Mobile Devices

# Mobile devices



Flagship



Mid Range



Entry



Basic



Octa-core (SD8 G3)  
Hexa-core (A17)



Quad-core



12++ GB RAM / 1TB  
ROM



4 GB RAM / 64 GB  
ROM



6000 mAh



2150 mAh

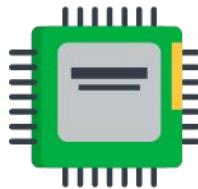


4G/5G Network



4G Network

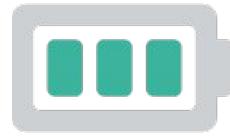
# Key Mobile Challenges



Processing  
Power



Memory and  
storage



Battery



Network

# Question?

India, a country with a large population base, low smartphone adoption and fast growth rate. Mobile phone shipments in the Indian market already exceed those in the US, making it the second largest phone market. The current Indian mobile phone market is still in the first batch of smartphone replacement. The problem for many people there is to have one (Sina Tech, 2018).

Considering the four challenges: processing power, memory and storage, battery, and network, how would you build apps to cater to the needs of users in emerging markets?

# Beyond Mobile Devices

# Beyond Mobile App



Wearabl  
e



Smart Home Appliance



Smart Car Dashboard



Internet of Things  
(IOT)

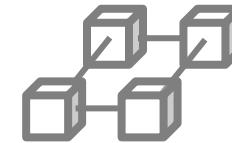
# Beyond Mobile App



Augmented Reality  
(AR)



Artificial Intelligent  
(AI)



Blockchain

AI



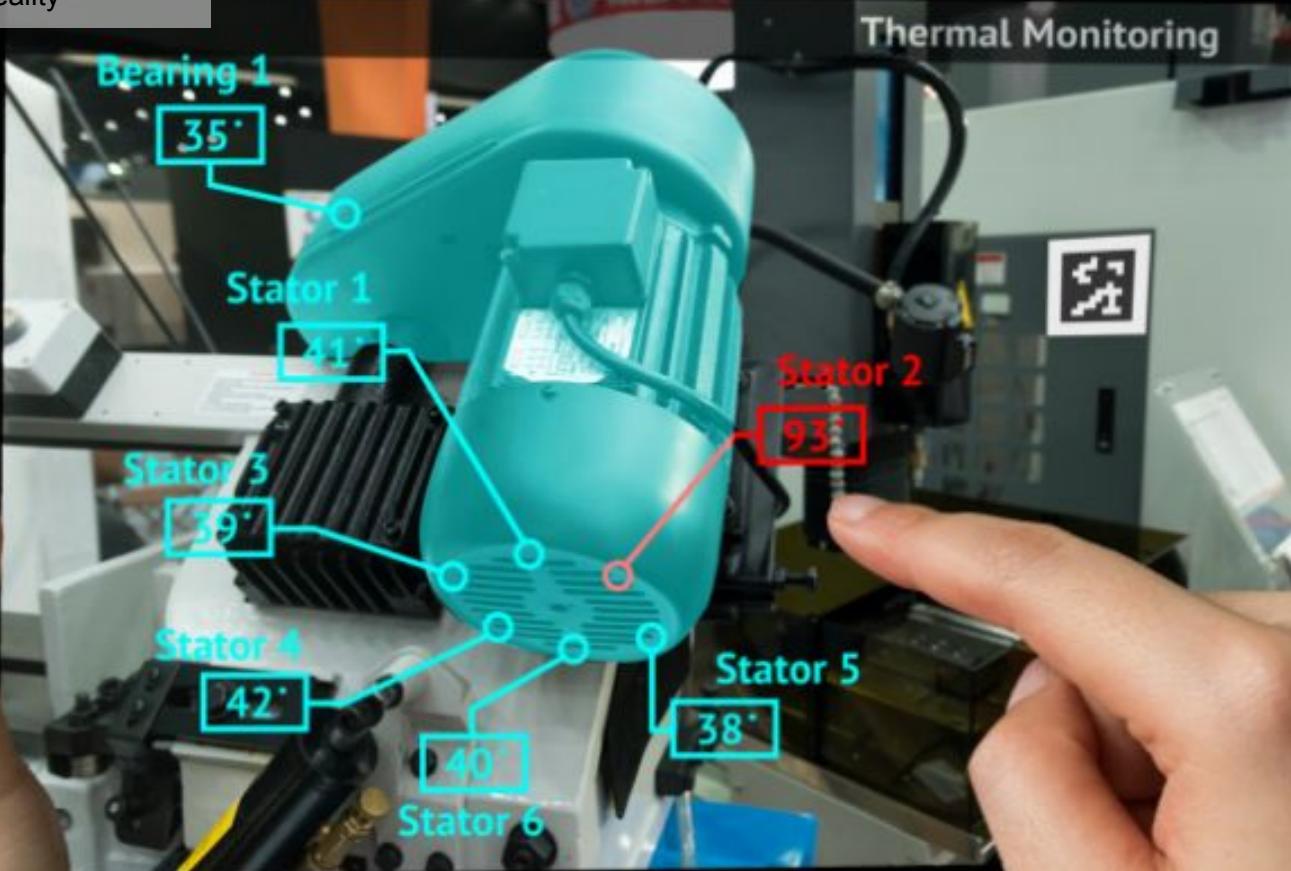
On Device AI

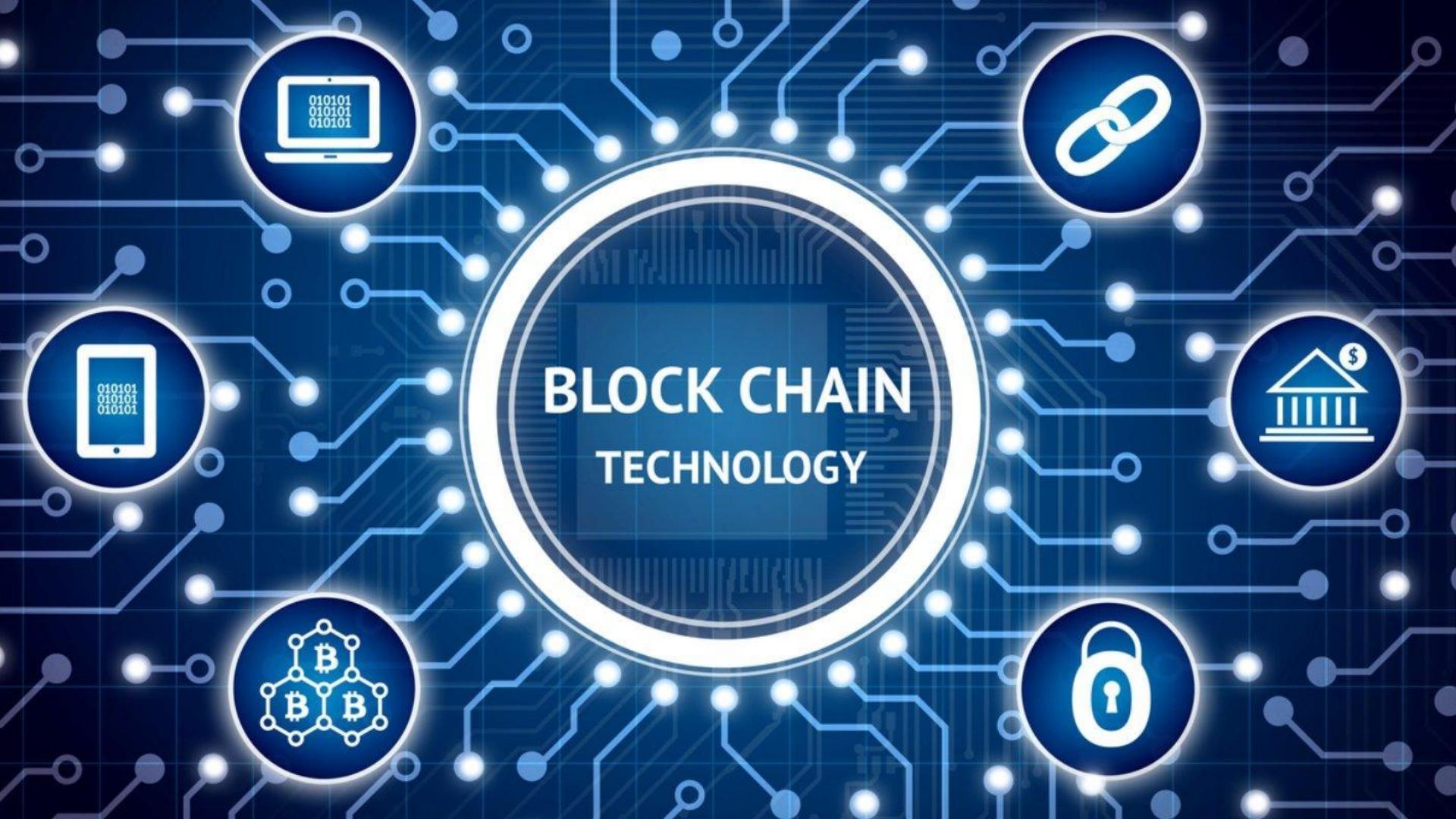


Rabbit R1



## Thermal Monitoring





# BLOCK CHAIN TECHNOLOGY



# Mobile Application Software Architecture And Framework

# Introduction to Flutter

An open-source mobile application development framework created by Google.

It is used to develop applications for Android and iOS, web and desktop (Windows, Mac OS, Linux, etc).

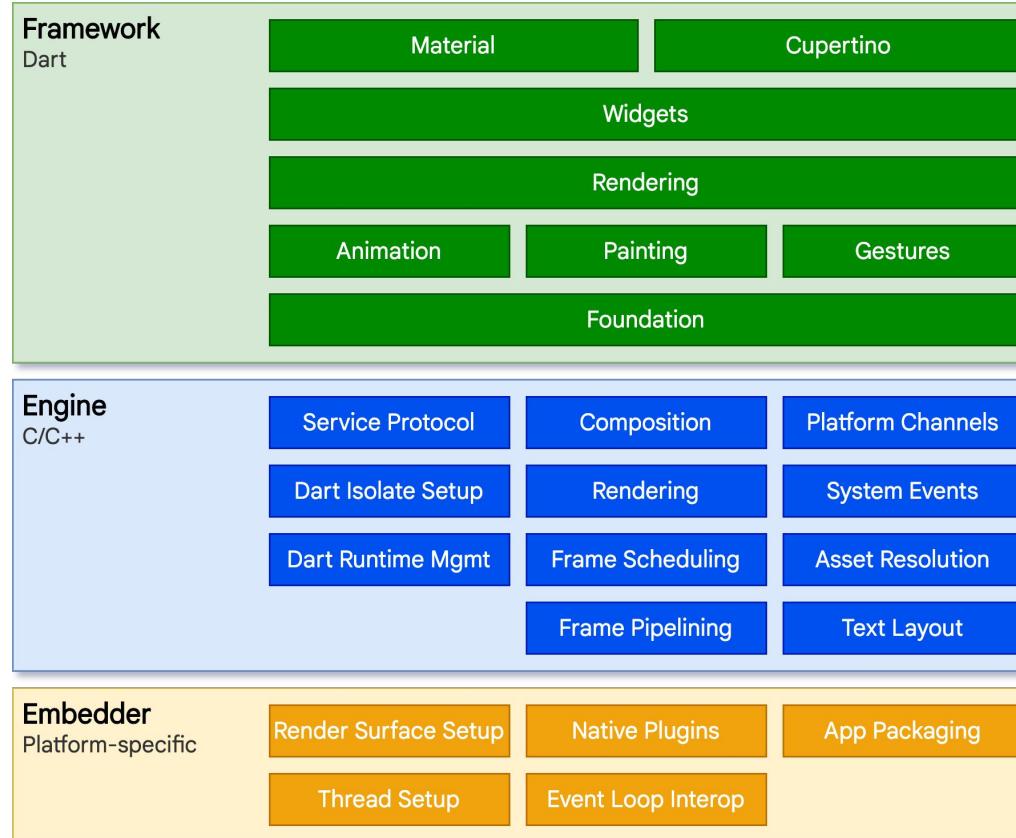
It is an SDK.

Language : DART

# Key Features

- Cross-platform development
- Hot reload
- Rich widgets
- Customizable
- High performance
- Open-source

# Architectural

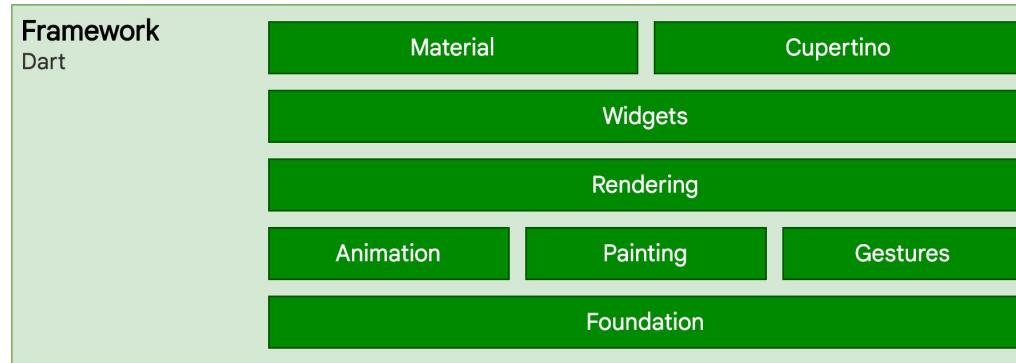


# Architectural - Framework

A programming language used to build Flutter apps.

Offers features like strong typing, asynchronous programming, and just-in-time (JIT) compilation for rapid development.

AOT compilation for production builds to improve performance.

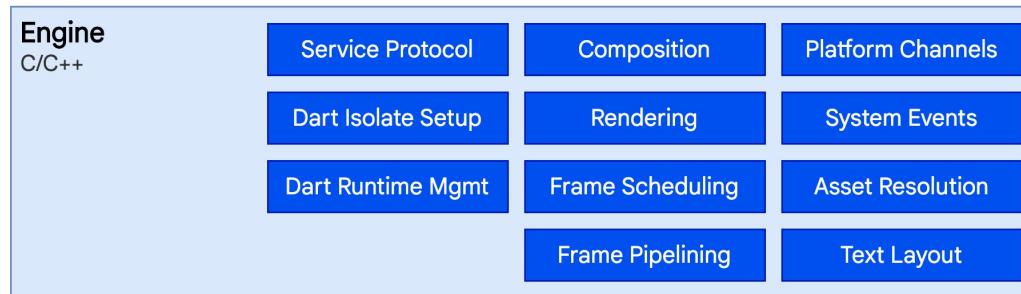


# Architectural - Engine

The foundation of Flutter, written mainly in C++.

Handles platform-specific tasks such as rendering, input, and network requests.

Implements the Flutter framework and Dart runtime.



# Architectural - Embedder

A platform-specific embedder coordinates with the underlying operating system.

Written in a language that is appropriate for the platform:

- Java and C++ for Android
- Objective-C/Objective-C++ for iOS and macOS
- C++ for Windows and Linux

With embedder, Flutter code can be integrated into an existing application as a module, or the code might be the entire content of the application.



# Architectural - Reactive UI

It focuses on data flow and propagation of change.

Building UI that react to changes in data in a declarative and efficient manner.

Three key concepts:

1. Data Flow
2. Declarative UI
3. State Management

# Architectural - Data Flow

## 1. Data Flow

Data flows unidirectionally from the source to the UI.

Changes to the data source trigger updates to the UI.



# Architectural - Data Flow

## 1. Data Flow

Reactive UI - a programming paradigm that focuses on data flow and propagation of change.

UI automatically update whenever the underlying data changes.

This leads to more responsive, efficient, and maintainable applications.

# Architectural - Reactive UI

## 2. Declarative UI

You describe the desired UI state, and Flutter handles the rendering and updates.

This makes it easier to reason about the UI and write less code.

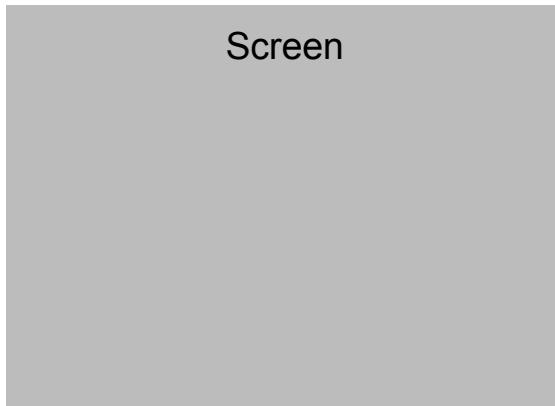
Widget is the fundamental of UI.

```
//Examples of Text widget
Text('You have pushed the button this many times:', ),
Text('${_counter}', style: Theme.of(context).textTheme.headline4, ),
```

# Architectural - Reactive UI

## 2. Declarative UI

The screen is the parent (the root) of all widgets.



# Architectural - Reactive UI

## 2. Declarative UI

E.g.1: The Container is forced to be exactly the same size as the screen.

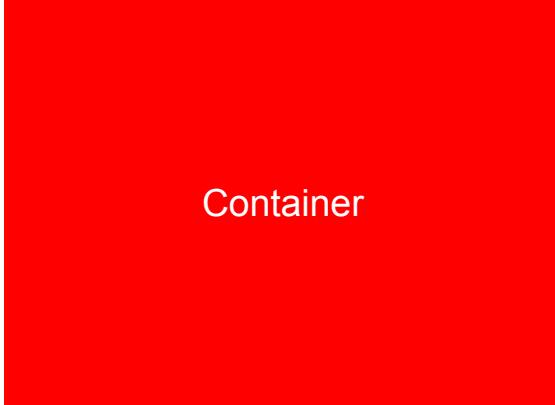


Container(color: red)

# Architectural - Reactive UI

## 2. Declarative UI

E.g.2: The red Container wants to be  $100 \times 100$ , but it can't, because the screen forces it to be exactly the same size as the screen.

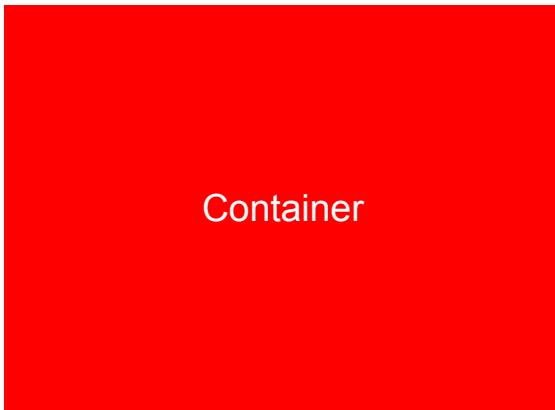


```
Container(  
    width: 100,  
    height: 100,  
    color: red)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.3: The screen forces the Center to be exactly the same size as the screen, so the Center fills the screen.

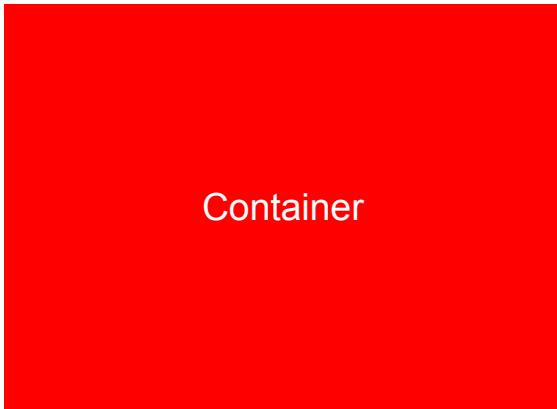


```
Center(  
    child: Container(color: red),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.4: The Center tells the Container that it can be any size but not bigger than the screen. The Container wants to be of infinite size, so it just fills the screen.

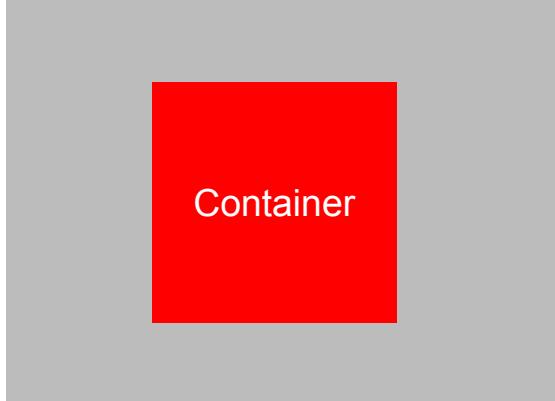


```
Center(  
    child: Container(  
        width: double.infinity,  
        height: double.infinity,  
        color: red),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.5: The Center is the same size as the screen. The Center ensures that the Container is not bigger than the screen. So, the Container can be  $100 \times 100$ .

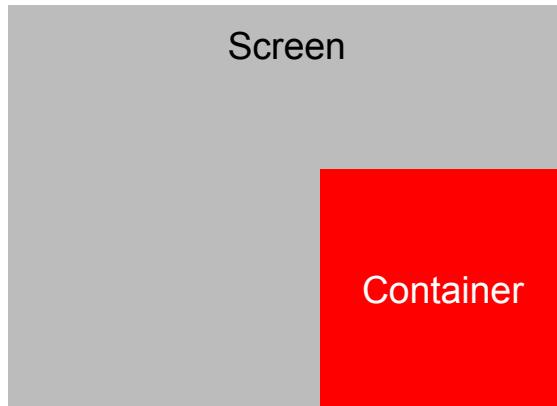


```
Center(  
  child: Container(  
    width: 100,  
    height: 100,  
    color: red),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.6: Align tells the Container that it can be any size. It aligns the Container to the bottom-right of the available space.

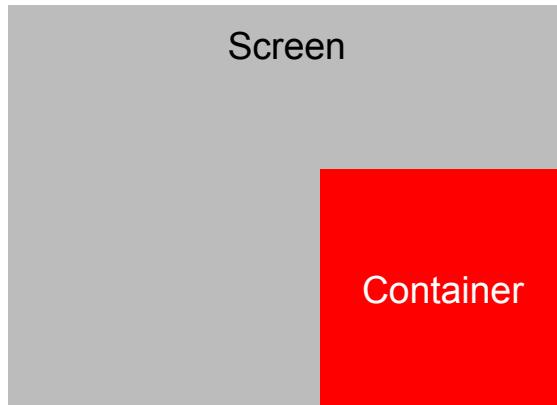


```
Align(  
  alignment: Alignment.bottomRight,  
  child: Container(  
    width: 100,  
    height: 100,  
    color: red),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.6: Align tells the Container that it can be any size. It aligns the Container to the bottom-right of the available space.

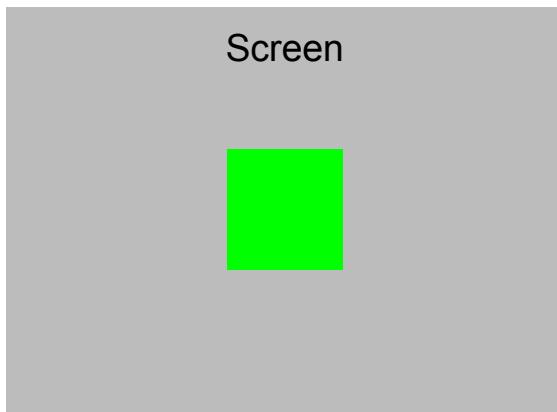


```
Align(  
  alignment: Alignment.bottomRight,  
  child: Container(  
    width: 100,  
    height: 100,  
    color: red),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.7: The Center tells the red Container that it can as big as the screen. However, the red Container has no size, it decides it wants to be the same size as its child.

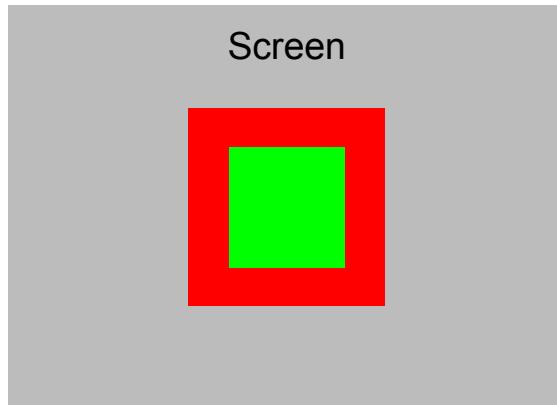


```
Center(  
    child: Container(  
        color: red,  
        child: Container(  
            color: green,  
            width: 30,  
            height: 30),  
    ),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.8: The red Container sizes itself to its children's size, but it takes its own padding into consideration. So it is also  $30 \times 30$  plus padding.

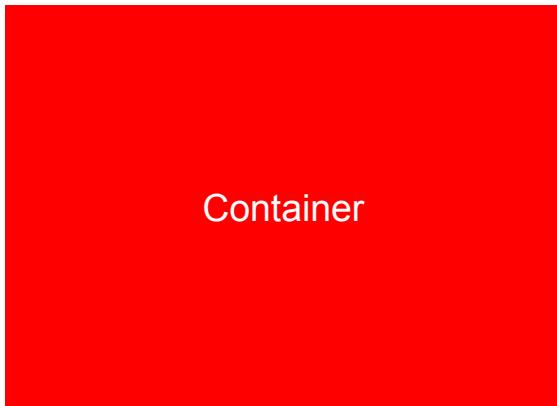


```
Center(  
    child: Container(  
        padding: const EdgeInsets.all(20),  
        color: red,  
        child: Container(  
            color: green,  
            width: 30,  
            height: 30  
        ),  
    ),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.9: The ConstrainedBox is forced to be exactly the same size as the screen, so it tells its child Container to also assume the size of the screen, thus ignoring its constraints parameter.

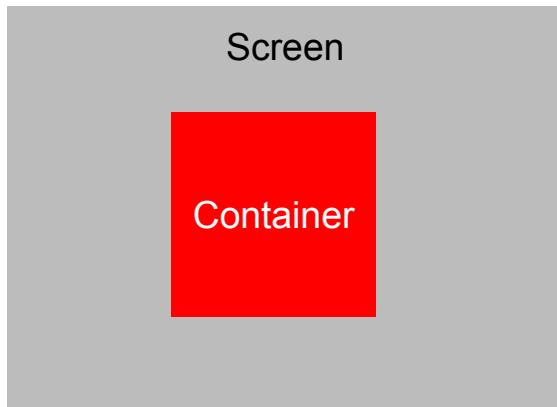


```
ConstrainedBox(  
  constraints: const BoxConstraints(  
    minWidth: 70,  
    minHeight: 70,  
    maxWidth: 150,  
    maxHeight: 150,  
  ),  
  child: Container(color: red, width: 10,  
    height: 10),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.10: The Container must be between 70 and 150 pixels. It wants to have 10 pixels, so it ends up having 70 (the minimum).

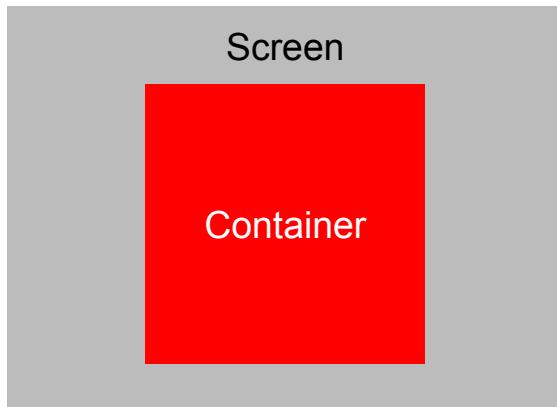


```
Center(  
    child: ConstrainedBox(  
        constraints: const BoxConstraints(  
            minWidth: 70, minHeight: 70,  
            maxWidth: 150, maxHeight: 150,  
        ),  
        child: Container(color: red, width: 10,  
                         height: 10),  
    ),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.11: The Container must be between 70 and 150 pixels. It wants to have 1000 pixels, so it ends up having 150 (the maximum).

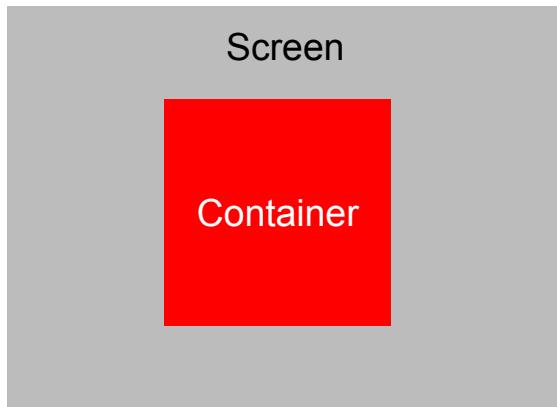


```
Center(  
    child: ConstrainedBox(  
        constraints: const BoxConstraints(  
            minWidth: 70, minHeight: 70,  
            maxWidth: 150, maxHeight: 150,  
        ),  
        child: Container(color: red, width: 1000,  
                          height: 1000),  
    ),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.12: The Container must be between 70 and 150 pixels. It wants to have 100 pixels, and that's the size it has, since that's between 70 and 150.

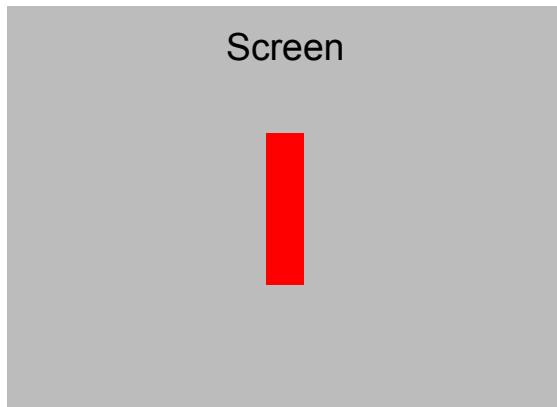


```
Center(  
    child: ConstrainedBox(  
        constraints: const BoxConstraints(  
            minWidth: 70, minHeight: 70,  
            maxWidth: 150, maxHeight: 150,  
        ),  
        child: Container(color: red, width: 100,  
                          height: 100),  
    ),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.13: The screen forces the UnconstrainedBox to be exactly the same size as the screen. However, the UnconstrainedBox lets its child Container be any size it wants.

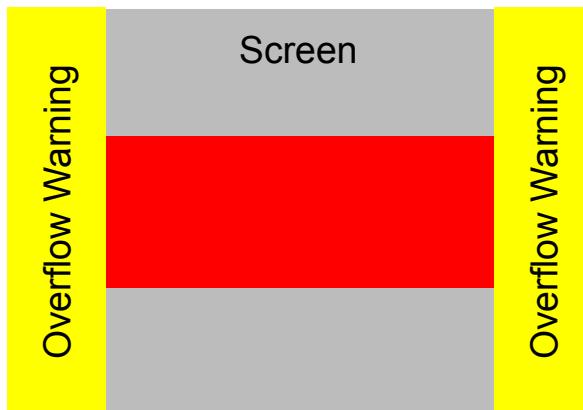


```
UnconstrainedBox(  
    child: Container(  
        color: red,  
        width: 20,  
        height: 50),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.14: The Container is 4000 pixels wide and is too big to fit in the UnconstrainedBox, so the UnconstrainedBox displays the much dreaded "overflow warning".

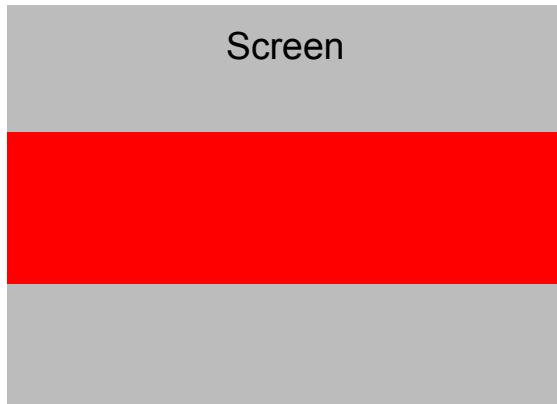


```
UnconstrainedBox(  
    child: Container(  
        color: red,  
        width: 4000,  
        height: 50),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.15: The Container has 4000 pixels of width, and is too big to fit in the OverflowBox, but the OverflowBox simply shows as much as it can, with no warnings given.

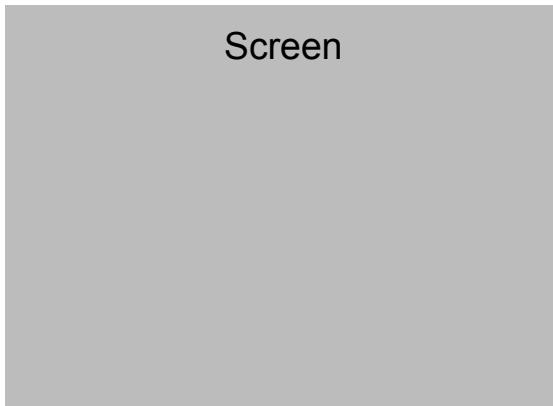


```
OverflowBox(  
    minWidth: 0,  
    minHeight: 0,  
    maxWidth: double.infinity,  
    maxHeight: double.infinity,  
    child: Container(color: red, width: 4000,  
                      height: 50),  
)
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.16: The system can't render infinite sizes, so it throws an error with the following message: BoxConstraints forces an infinite width.

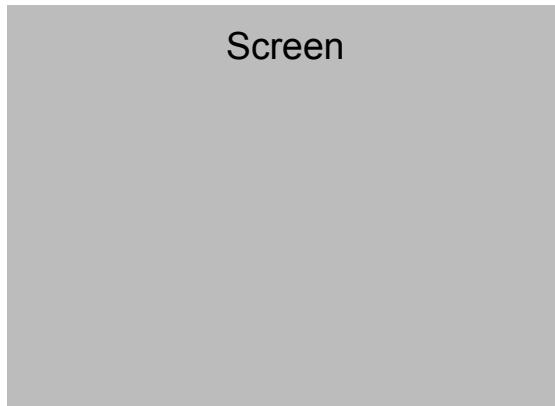


```
Screen  
    UnconstrainedBox(  
        child: Container(  
            color: Colors.red,  
            width: double.infinity,  
            height: 100),  
    )
```

# Architectural - Reactive UI

## 2. Declarative UI

E.g.17: The system can't render infinite sizes, so it throws an error with the following message: BoxConstraints forces an infinite width.



```
Screen  
    UnconstrainedBox(  
        child: Container(  
            color: Colors.red,  
            width: double.infinity,  
            height: 100),  
    )
```

# Architectural - Reactive UI

## 2. Declarative UI

***“Constraints go down. Sizes go up. Parent sets position.”***

- A widget can decide its own size only within the constraints given to it by its parent.
- Widget's parent who decides the position of the widget.
- It's impossible to precisely define the size and position of any widget without taking into consideration the parent-child tree as a whole.
- If a child wants a different size from its parent and the parent doesn't have enough information to align it, then the child's size might be ignored.

# Architectural - State Management

State refers to the condition or data of a system at a particular moment.

It's a snapshot of the system's variables, properties, and other data that define its behavior.

In Flutter, state determines the dynamic behaviour of your app. Two main types of state:

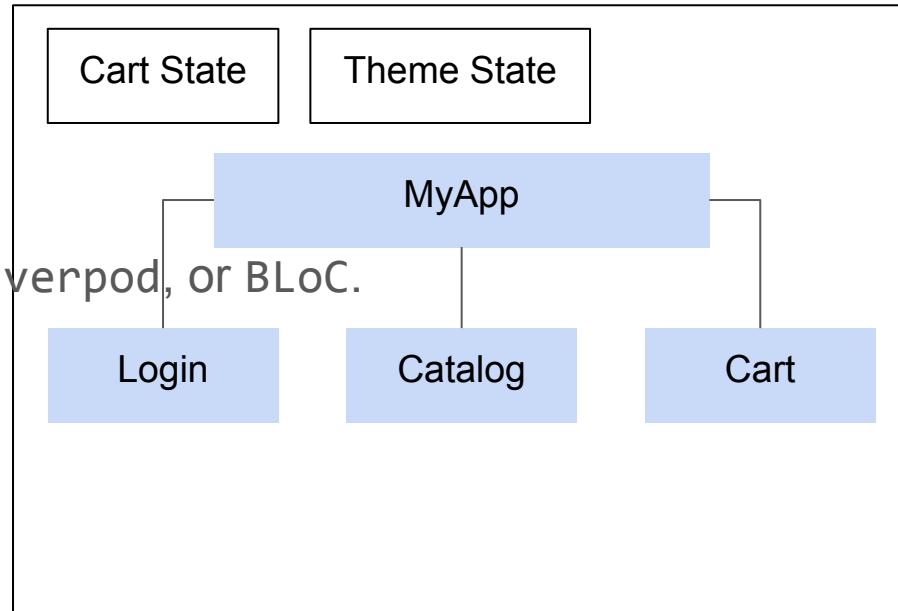
1. App State
2. Widget State

# Architectural - State Management

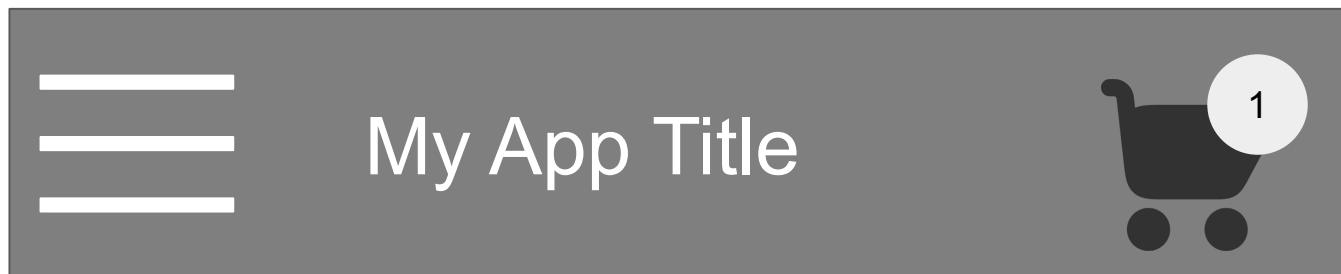
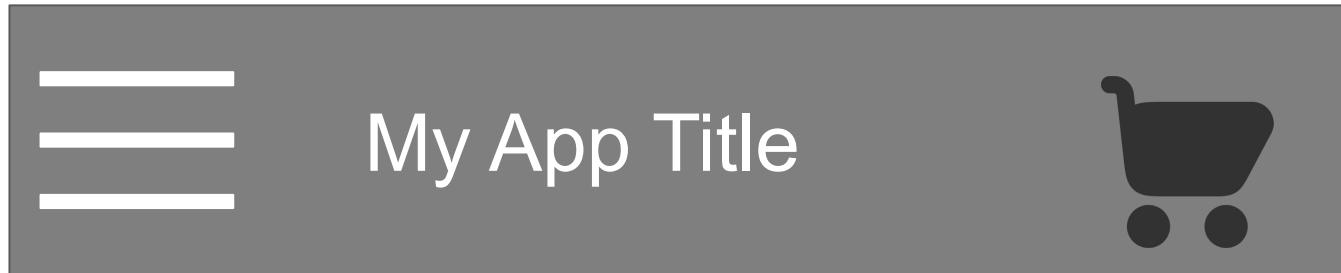
## 1. App State

Global state that affects the entire app.

Managed using libraries like Provider, Riverpod, or BLoC.



# Architectural - State Management

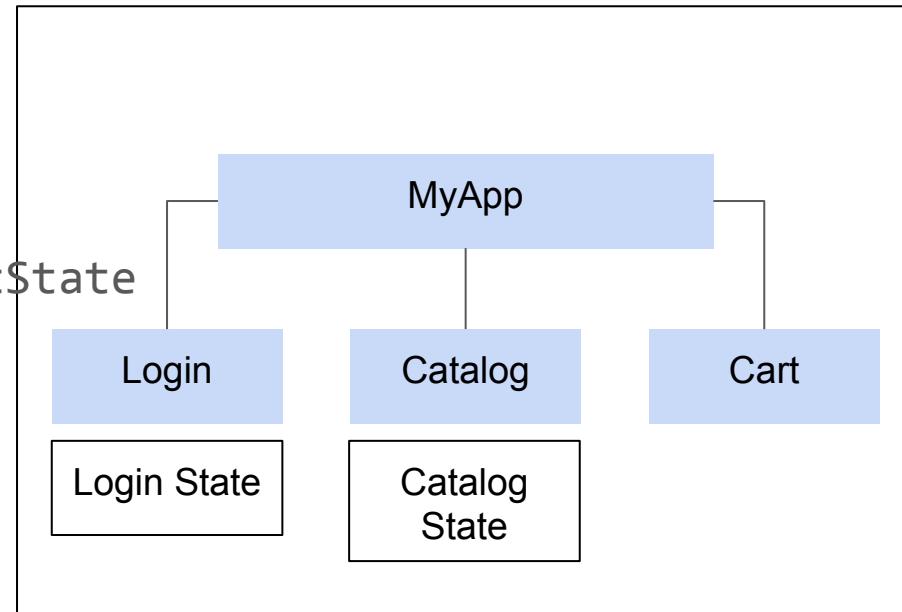


# Architectural - State Management

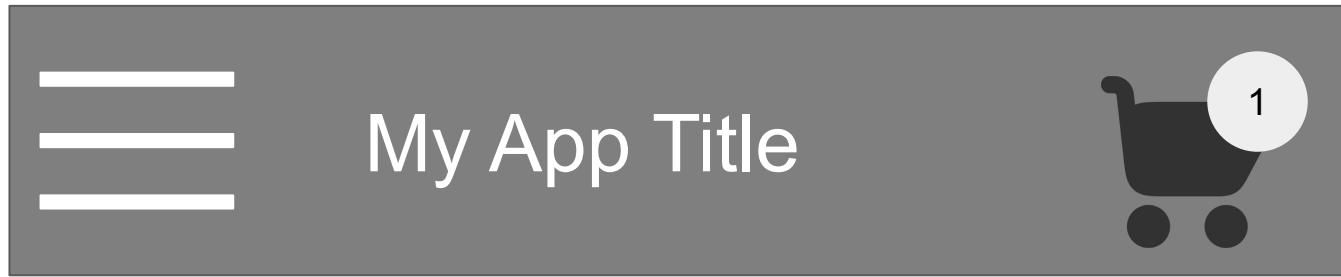
## 2. Widget State

Local state specific to a particular widget.

Managed using `StatefulWidget` and `setState`



# Architectural - State Management



Item 1

Item 2



Item 3

# React Framework

Flutter is a reactive, declaration UI framework.

Widgets are the building blocks of a Flutter app's UI.

Developer provides:

1. A mapping from application state to UI.
2. The framework takes on the task of updating the UI at runtime when the application state changes.

# React Framework

Feature	Traditional UI	Reactive UI
Programming Paradigm	Imperative	Declarative
State Management	Manual	Framework-assisted
UI Updates	Manual	Automatic
Performance	Can be less efficient	More efficient

```
class MainActivity : AppCompatActivity() {  
    private lateinit var textView: TextView  
    // Declare UI state  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        // Layout of UI  
  
        textView = findViewById(R.id.textView)  
        // Linking UI to state  
  
        // Button click handler  
        button.setOnClickListener {  
            val newText = "Hello, World!"  
            textView.text = newText  
        }  
        // Update state and UI  
    }  
}
```

```
class _MyHomePageState extends State<MyHomePage> {
```

```
    String _text = ";
```

Declare UI state

```
    void _toggleText() {  
        setState(() {_text = 'Hello, World!'});  
    }
```

Update state

```
@override
```

```
Widget build(BuildContext context) {  
    return Scaffold(
```

```
        appBar: AppBar(  
            title: const Text('My App'),
```

```
        ),
```

```
        body: Center(
```

```
            child: Column(
```

```
                mainAxisAlignment: MainAxisAlignment.center,
```

```
                children: [
```

```
                    Text(_text, style: const TextStyle(fontSize: 24),  
                ),
```

```
                    ElevatedButton(onPressed: _toggleText,
```

```
                        child: const Text('Click Me'),
```

Layout of UI

```
}
```

# React Framework - Widget

Widgets form a hierarchy based on composition.

Each widget nests inside its parent and can receive context from the parent.

This structure carries all the way up to the root widget (the container that hosts the Flutter app, typically MaterialApp or CupertinoApp).

Note: We will discuss widgets in more detail...

# Conclusion

- Mobile operating systems and ecosystem
- Development tools
- Mobile devices
- Mobile application software architecture and framework

# Find Out More

Flutter architectural overview,

<https://docs.flutter.dev/resources/architectural-overview>

# Chapter 1.2

Introduction to Dart

# Introduction

Dart is an open-source, general-purpose programming language developed by Google

Applications:

- Web development
- Mobile development
- Server-side development

# Introduction

Key features:

1. Object-oriented
2. Strongly typed
3. Fast and efficient
4. Modern features

# Structure

```
// This is a comment  
  
void main() {  
    // This is the main function, program starts here  
    print("Hello, world!");  
}
```

# Variable

## Data Types

Numbers:

`int`: Stores whole numbers (integers).

`double`: Stores numbers with decimals (floating-point numbers).

`num`: Supertype of both `int` and `double`.

# Variable

## Declaration Keywords

```
// Declares an integer variable, needs initialization before use  
int age;  
  
// Declares a double variable  
double pi = 3.14159;  
  
// Declare multiple variables of the same type in a single line  
int x = 5, y = 10;
```

# Variable

## Declaration Keywords

```
// Nullable double, initially null  
double? area = null;
```

# Variable

## Data Types

Text:

**String**: Stores sequences of characters (text).

# Variable

## Declaration Keywords

```
// String type with initial value  
String name = "Alice";  
  
// var infers String based on assignment  
var greeting = "Hello";  
  
// var infers String based on assignment  
var flybyObjects = ['Jupiter', 'Saturn', 'Uranus', 'Neptune'];
```

# Variable

## Data Types

Logical:

`bool`: Stores true or false values (Boolean).

# Variable

## Declaration Keywords

```
bool isLoggedIn = true;
```

```
bool isNightTime = false;
```

# Basic Structures

```
if (year >= 2001) {  
    print('21st century');  
} else if (year >= 1901) {  
    print('20th century');  
}
```

# Basic Structures

```
for (int month = 1; month <= 12; month++) {  
    print(month);  
}  
  
for (final object in flybyObjects) {  
    print(object);  
}
```

# Basic Structures

```
while (year < 2016) {  
    year += 1;  
}
```

# Functions

```
bool isNoble(int atomicNumber) {  
    return _nobleGases[atomicNumber] != null;  
}
```

# Functions

```
int fibonacci(int n) {  
    if (n == 0 || n == 1) return n;  
    return fibonacci(n - 1) + fibonacci(n - 2);  
}  
  
var result = fibonacci(20);
```

# Functions

```
bool isNoble(int atomicNumber) => _nobleGases[atomicNumber] !=  
null;  
  
flybyObjects.where((name) => name.contains('turn')).foreach(print);
```

# Find Out More

[Intro to Dart](#)

# Chapter 2

## Mobile Application Models

# Contents

Route and Navigator

App Life Cycle

# Route and Navigator

# Route and Navigator

In Flutter, screens and pages are called **routes**.

In Android, a route is equivalent to an **Activity**.

In iOS, a route is equivalent to a **ViewController**.

In Flutter, a **route** is just a **widget**.

# Route and Navigator

When a user interacts with your app, they navigate between these routes.

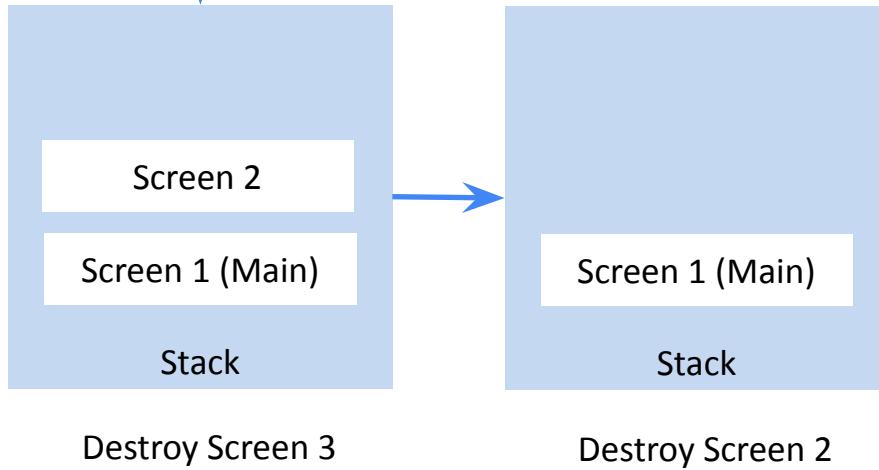
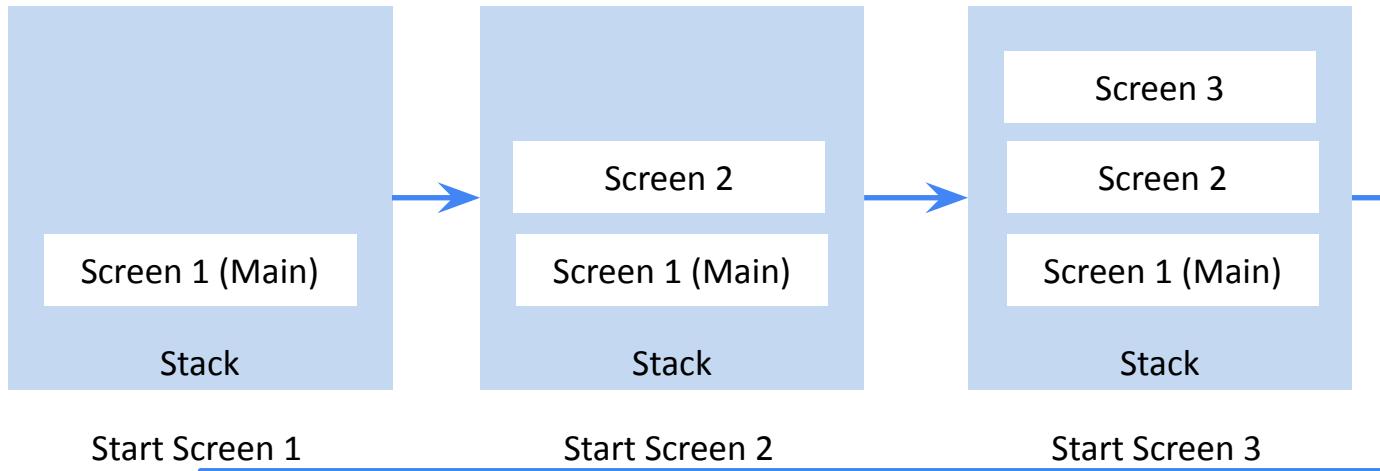
For advanced navigation and routing requirements, use a routing package such as `go_router`.

# Route and Navigator

The Navigator widget displays screens as a **stack** using the correct transition animations for the target platform.

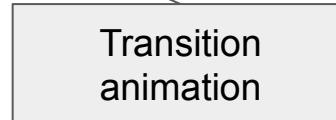
Navigator is suitable for small applications without complex deep linking.

To navigate to a new screen, access the Navigator through the route's `BuildContext` and call imperative methods such as `push()` or `pop()`:



# Route and Navigator

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



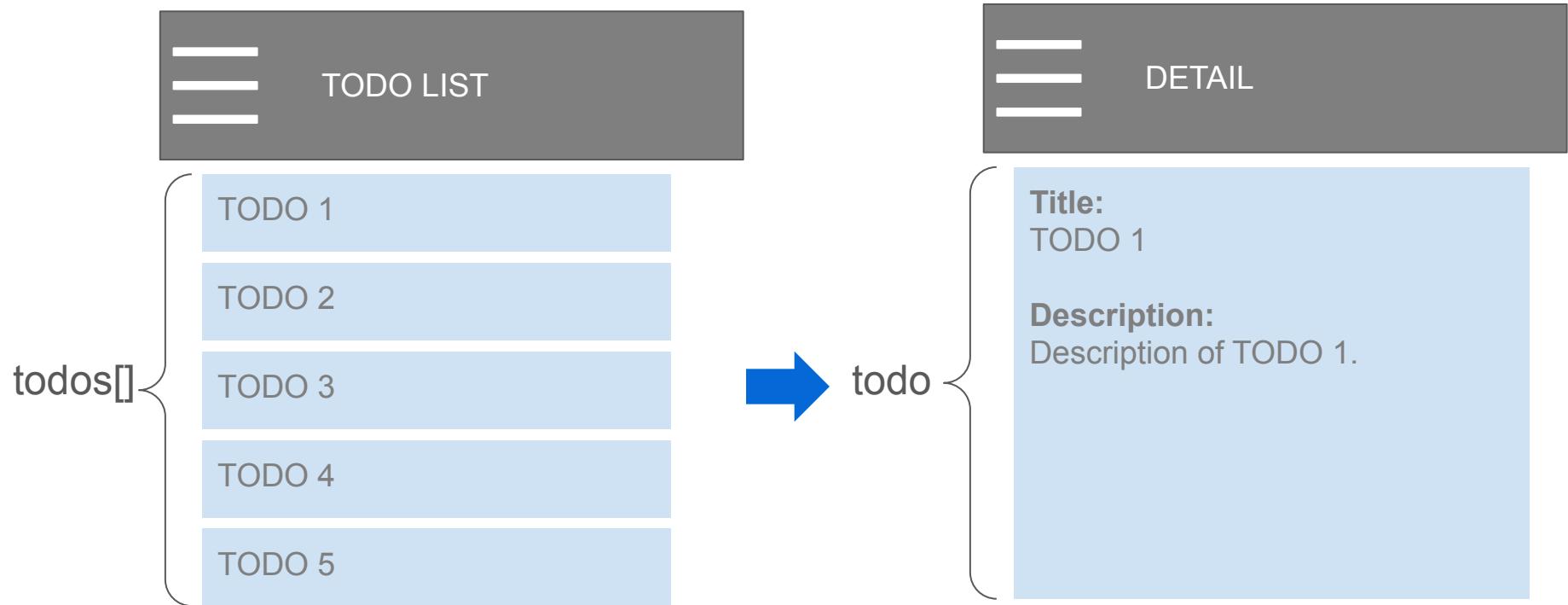
# Route and Navigator

```
// Within the Second Route widget  
onPressed: () {  
  Navigator.pop(context);  
}  
}
```

# Route and Navigator - Send Data

Often, you not only want to navigate to a new screen, but also pass data to the screen as well.

For example, you might want to pass information about the item that's been tapped.



# Route and Navigator - Send Data

```
class Todo {  
    final String title;  
    final String description;  
  
    const Todo(this.title, this.description);  
}
```

# Route and Navigator - Send Data

```
class DetailScreen extends StatelessWidget {  
  // In the constructor, require a Todo.  
  const DetailScreen({super.key, required this.todo});  
  
  // Declare a field that holds the Todo.  
  final Todo todo;  
}
```

...

# Route and Navigator - Send Data

```
//In the Main Screen
Navigator.push(
    context,
    MaterialPageRoute(
        builder: (context) => DetailScreen(todo: todos[index]),
    ),
)
```

Refer to the complete code at <https://docs.flutter.dev/cookbook/navigation/passing-data>

# Route and Navigator - Return Data

Often, you might want to pass data back from a second screen to the first screen after a user interaction, such as selecting an item or entering information.

```
//Within the Second Route  
Navigator.pop(context, 'Returned Data');
```

```
//Within the Second Route
Navigator.pop(context, 'Returned Data');

// Within the First Route
onPressed: () {
  Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => SecondScreen()),
  ).then((value) {
    // Handle the returned data here
    if (value != null) {
      setState(() { _dataFromSecondScreen = value; });
    }
  });
}
```

# Shared State Management

For more complex scenarios, consider using a state management solution like Provider.

This allows you to share data across multiple screens and update it from any part of your app.

# Shared State Management

```
// Provider
class DataProvider extends ChangeNotifier {
    String _data = '';
    String get data => _data;

    void setData(String newData) {
        _data = newData;
        notifyListeners();
    }
}
```

# Shared State Management

```
// First Screen
Consumer<DataProvider>(
    builder: (context, dataProvider, child) {
        return ElevatedButton(
            onPressed: () {
                Navigator.push(
                    context,
                    MaterialPageRoute(
                        builder: (context) => SecondScreen(dataProvider:
                            dataProvider),
                    ),
                );
            },
        ),
        child: Text('Go to Second Screen'),
    },
)
```

# Shared State Management

```
// Second Screen  
SecondScreen(this.dataProvider);  
  
// ...  
  
onPressed: () {  
    dataProvider.setData('Returned data');  
    Navigator.pop(context);  
}  
}
```

# Deep Linking

Deep linking is the technique of linking directly to specific content within an app.

It allows users to open your app to a particular screen or feature, rather than just the home screen.

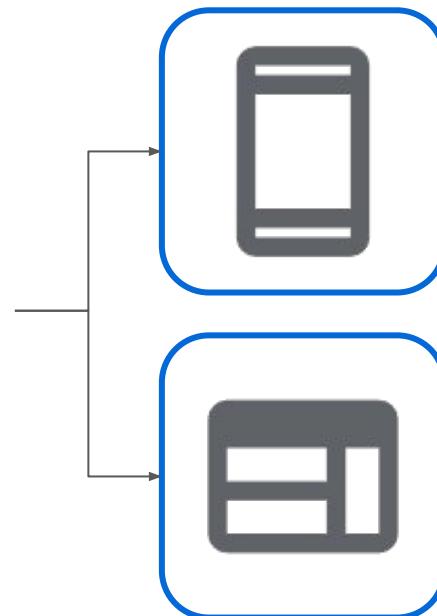
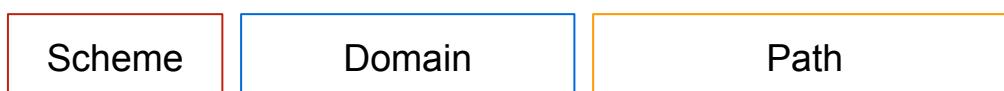
Flutter supports deep linking on iOS, Android, and the web.

Opening a URL displays that screen in your app.

# Deep Linking

You can use the same link for both web and app.

<https://dashcast.net/library/album/1>



# Deep Linking

An **app link** is a type of deep link that uses `http` or `https` and is exclusive to Android devices. Find out more about it here:

<https://docs.flutter.dev/cookbook/navigation/set-up-app-links>

**Universal link** is a deep link implementation for iOS. Find out more about it here:

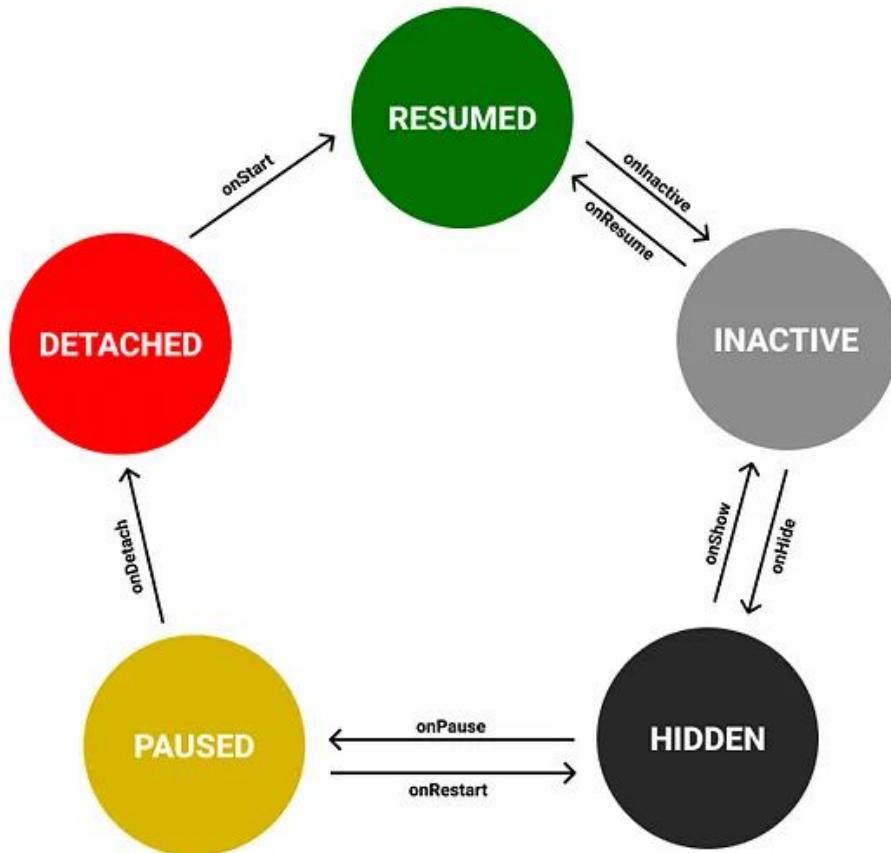
<https://docs.flutter.dev/cookbook/navigation/set-up-universal-links>

# App Lifecycle

# App Lifecycle

- The Flutter app lifecycle refers to the different states a Flutter app can be in throughout its runtime.
- Understanding these states and how the app transitions between them is essential for building efficient and responsive apps.
- By understanding the app lifecycle, you can create a more polished and user-friendly experience in your Flutter applications.

# FLUTTER APPLICATION LIFE CYCLE

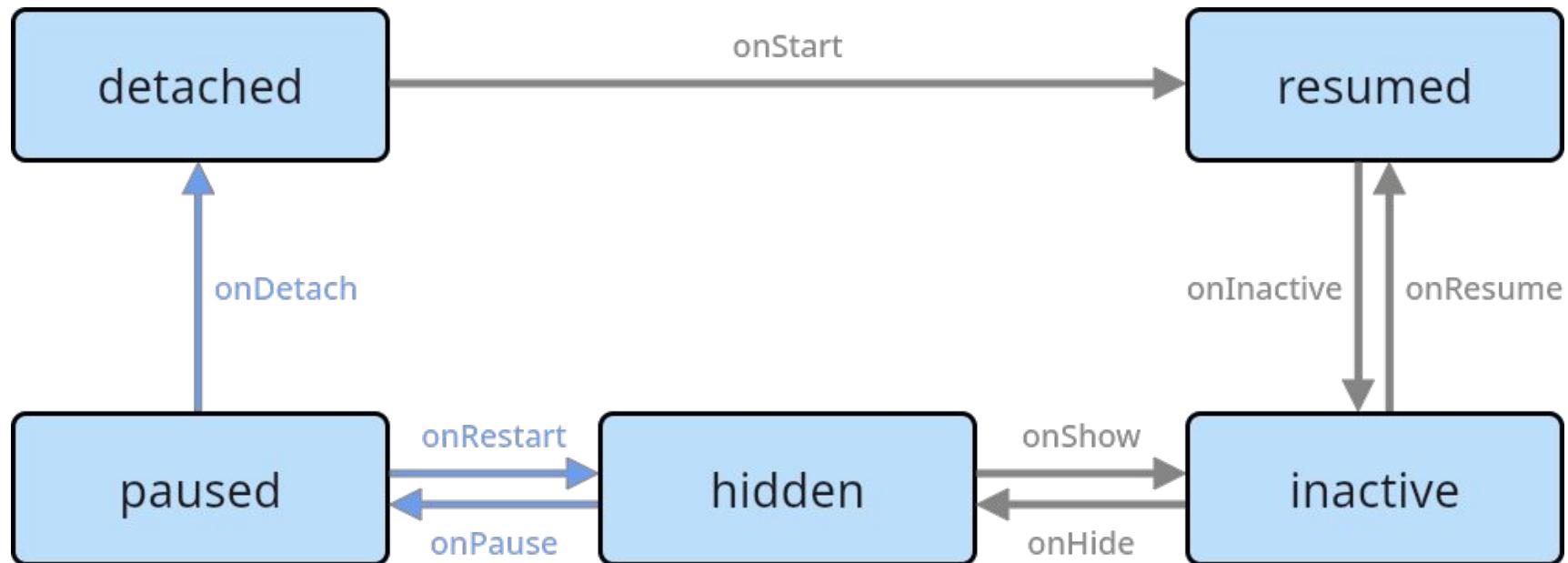


# App Lifecycle

There are four primary states in the Flutter app lifecycle:

1. Resumed
2. Paused
3. Inactive
4. Detached

# App Lifecycle



# App Lifecycle

## 1. Resumed

This is the ideal state where the app is **visible** on the screen, actively **responding to user interactions**.

Here are some actions you typically take:

- Fetch and display data relevant to the user.
- Process user interactions (taps, swipes, form submissions).
- Play audio or video.
- Start animations.

Essentially, any functionality core to your app's active use happens here.

# App Lifecycle

## 2. Paused

In this state, the app is **not visible** to the user and has **limited functionality**.

It's usually in the background. This is a good time for:

- Pausing tasks that consume resources (like animations or location updates).
- Saving any unsaved user data.
- Releasing resources that aren't immediately needed.

You want to **avoid intensive tasks** here as the app might be suspended entirely soon.

# App Lifecycle

## 3. Inactive

This is a **brief transitional state** between **resumed** and **paused**, where the app might **lose focus** but still resides in memory.

You might **not have time for complex actions** here, but you could:

- Briefly pause ongoing tasks (like a network request) in case the app resumes quickly.
- Prepare for a potential state change (resumed or detached).

# App Lifecycle

## 4. Detached

This state signifies the app is **no longer attached to any view** and **isn't actively running**.

It occurs before initialization and potentially after all views are detached (on Android and iOS).

This is a good time for:

- Cleaning up resources (closing databases, releasing memory).
- Persisting any critical data that needs to be saved even after the app is closed.

**Avoid actions that require UI elements or user interaction.**

# App Lifecycle

Here are some key points to remember about the Flutter app lifecycle:

- You can monitor app lifecycle changes using the `AppLifecycleState` enum and the `AppLifecycleListener` class.
- Effectively handling lifecycle transitions allows you to optimize resource usage, perform actions when the app goes into the background (like pausing timers or saving data), and respond appropriately when it resumes.

```
@override
void initState() {
    super.initState();
    _state = SchedulerBinding.instance.lifecycleState;
    _listener = AppLifecycleListener(
        onShow: () => _handleTransition('show'),
        onResume: () => _handleTransition('resume'),
        onHide: () => _handleTransition('hide'),
        onInactive: () => _handleTransition('inactive'),
        onPause: () => _handleTransition('pause'),
        onDetach: () => _handleTransition('detach'),
        onRestart: () => _handleTransition('restart'),
        // This fires for each state change. Callbacks above fire only for
        // specific state transitions.
        onStateChange: _handleStateChange,
    );
    if (_state != null) {
        _states.add(_state!.name);
    }
}
```

# Find Out More

Route and Navigator, <https://docs.flutter.dev/ui/navigation>

AppLifeCycleListener class,

<https://api.flutter.dev/flutter/widgets/AppLifecycleListener-class.html>

# Chapter 3.1

## User Interfaces

# Contents

- Basic Widgets
- Stateless and Stateful Widget
- Design Systems
  - Cupertino Widgets
  - Material Widgets
- Layout
  - Single-child
  - Multi-child
- Sliver Widgets
- Performance best practices

# Basic Widgets

# Widgets

Widgets are UI components of your app. They define the :

- appearance
- layout, and
- behavior of a specific part of your app's interface

A widget represents a single element (e.g. button, text field, image) or even a complex layout container.

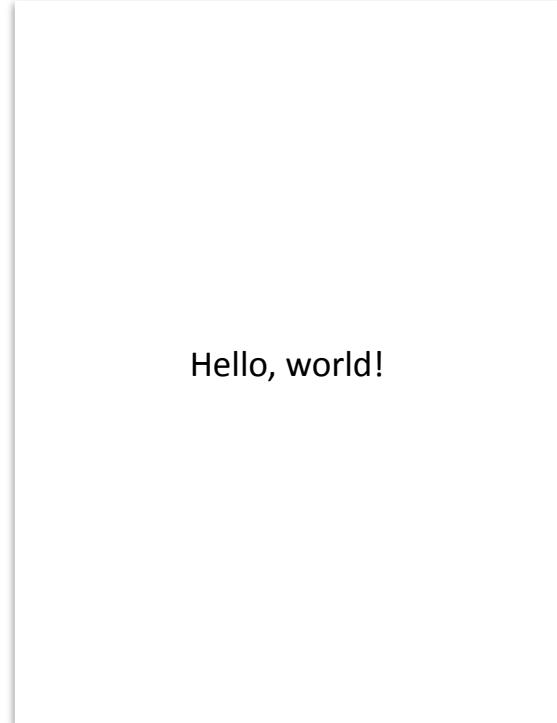
Widgets are nested hierarchically, forming a tree-like structure that defines the overall UI of your app.

# Widgets

The minimum Flutter app with a text widget:

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

void main() {
  runApp(
    const Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```



Hello, world!

# Widget - Characteristics

Customizable:

- Most widgets offer various properties that you can adjust to customize their appearance and behavior.
- This allows you to create unique and personalized UIs.

# Widget - Characteristics

Reusable:

- Widgets are designed to be reused throughout your app, promoting code efficiency and consistency.

```
ElevatedButton( onPressed: () {  
    // Handle button press },  
    child: Text('Button 1'), ),  
    // ... other parts of your app
```

```
CustomButton( text: 'Custom Button',  
    onPressed: () { // Handle button press },  
    color: Colors.green, ),
```

# Basic widgets

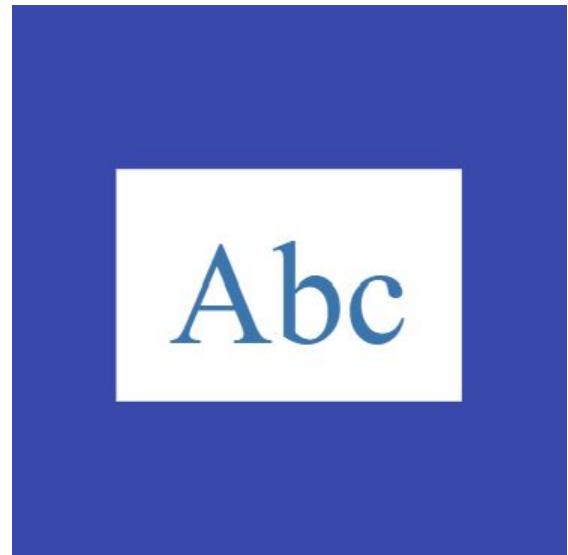
1. Text
2. TextField
3. ElevatedButton
4. Image
5. Row/Column
6. Scaffold
7. Placeholder
8. Icon
9. AppBar

# Text

## Text

Displays text content.

A run of text with a single style.



# Text

The Text widget displays a string of text with single style.

The style argument is optional.

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

void main() {
  runApp(
    Text(
      'Hello, world!',
      textDirection: TextDirection.ltr,
    ),
  );
}
```

# Text

```
Container(  
    width: 100,  
    decoration: BoxDecoration(border:  
Border.all()),  
    child: Text(  
        overflow: TextOverflow.ellipsis,  
        'Hello ${_name}, how are you?',  
        textDirection:  
TextDirection.ltr,),  
    )  
)
```

Hello Ali, ...

# Text

```
const Text.rich(  
  TextSpan(  
    text: 'Hello', // default text style  
    children: <TextSpan>[  
      TextSpan(  
        text: ' beautiful ',  
        style: TextStyle(  
          fontStyle: FontStyle.italic)  
      ),  
      TextSpan(  
        text: 'world',  
        style: TextStyle(  
          fontWeight: FontWeight.bold)  
      ),  
    ],  
  ),  
)
```

Hello, *beautiful world*

# TextField

A text field lets the user enter text, either with hardware keyboard or with an onscreen keyboard.

Enter a search term

# TextField

- InputDecoration includes label, icon, inline hint text, and error text.
- To remove the decoration entirely, set the decoration to null.

`TextField(`

`decoration: InputDecoration(`

`border: OutlineInputBorder(),`

`hintText: 'Enter a search term',`

`), ),`



# TextField

- You can get input value from a TextField in 3 ways:
  - onChanged()
  - onSubmitted()
  - Controller

# Reading Values

A common way to read a value from a `TextField` is to use the `onSubmitted` callback.

This callback is applied to the text field's current value when the user finishes editing.

It can be used to manually move focus to another input widget when a user finishes with the currently focused input widget.

# Reading Values

The default behavior of getting value is an invocation of `onChanged`.

The text field calls the `onChanged` callback whenever the user changes the text in the field.

# TextField

```
TextField(  
    decoration: InputDecoration(  
        labelText: 'Email Address',  
        hintText: 'Enter your email',  
        icon: Icon(Icons.email),  
    ),  
    keyboardType: TextInputType.emailAddress,  
  
    // Reference to a TextEditingController object  
    controller: _emailController,  
    onChanged: (text) => print('Email: $text'),  
,
```

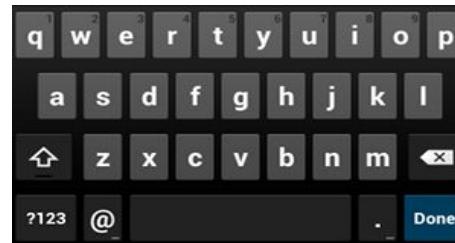
# Text Fields

Selecting the appropriate keyboard type, you can significantly improve the user's input experience and reduce errors.



text

Normal text keyboard



emailAddress

Normal text keyboard  
with the @ character



number

Basic number keypad

phone

Phone-style keypad

# TextField

The `TextEditingController` class is used to manage the text input in a `TextField`.

It allows you to:

1. Set the initial text: Set the default text that appears in the `TextField`.
2. Retrieve the current text: Get the text that the user has entered.
3. Listen to text changes: Respond to changes in the text as the user types.
4. Clear the text: Remove all text from the `TextField`.

# TextField

```
class _MyTextFieldAppState extends State<MyTextFieldApp> {
  final TextEditingController _textController = TextEditingController();

  void _submitText() {
    String text = _textController.text; _textController.text _textController text
    // Do something with the text, e.g., print it to the console
    print("Entered text: $text");
  }
  ...
}

TextField(
  controller: _textController, controller _textController
  hintText: 'Enter your text here',),
ElevatedButton(
  onPressed: _submitText,
  child: Text('Submit'),),
```

\_textController.text \_textController text controller \_textController

Retrieve the current text

Assign controller

# TextField

Clear the text.

```
_textController.clear();
```

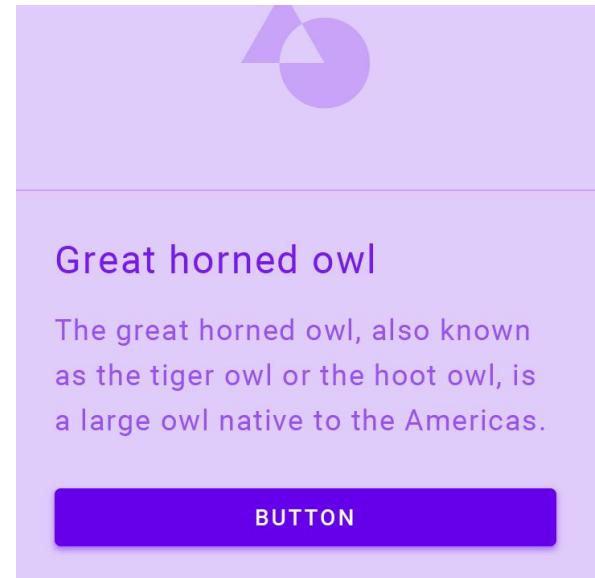
Dispose the Controller when the widget is removed to avoid memory leaks.

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

# ElevatedButton

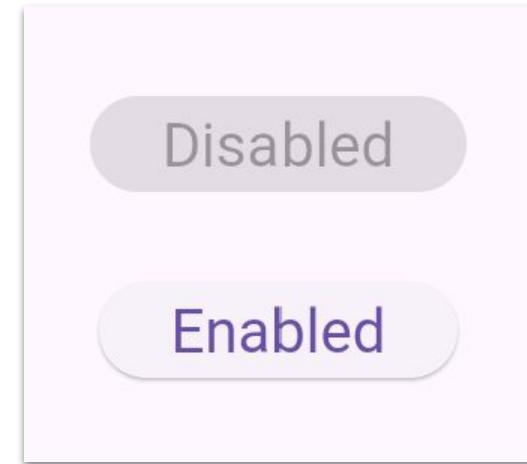
Creates buttons for user interaction.

A filled button whose material elevates when pressed.



# ElevatedButton

```
ElevatedButton(  
    style: style,  
    onPressed: null,  
    child: const Text('Disabled'),  
,  
  
const SizedBox(height: 30),  
  
ElevatedButton(  
    style: style,  
    onPressed: () {},  
    child: const Text('Enabled'),  
,
```



# Image

Image: Displays images.

Ways to display images:

1. Assets folder
2. Explicit device's directory
3. Network



# Image

## 1. Assets folder

Step 1:

Create a folder named '**assets**' in the project **root directory**

Step 2:

Copy all images to the assets folder.

Reference: <https://api.flutter.dev/flutter/widgets/Image/Image.asset.html>

# Image

## 1. Assets folder

Step 3:

Open the project's pubspec.yaml file and enter the image file names

```
flutter:  
  assets:  
    - images/cat.png  
    - images/2x/cat.png  
    - images/3.5x/cat.png
```

# Image

## 1. Assets folder

Step 4:

Write code to display an image from asset

```
Image.asset('images/cat.png')
```

# Image

## 2. Explicit device's directory

Displays an image obtained from a specific location, e.g. from a photo app

```
Image.file(selectedImages,),
```

Reference: <https://api.flutter.dev/flutter/widgets/Image/Image.file.html>

# Image

## 3. Network

To load images from a network URL:

```
Image.network('https://example/image_name.jpg')
```

Note: Reading a file from network involve an asynchronous task, which cannot provide a result immediately when it is started

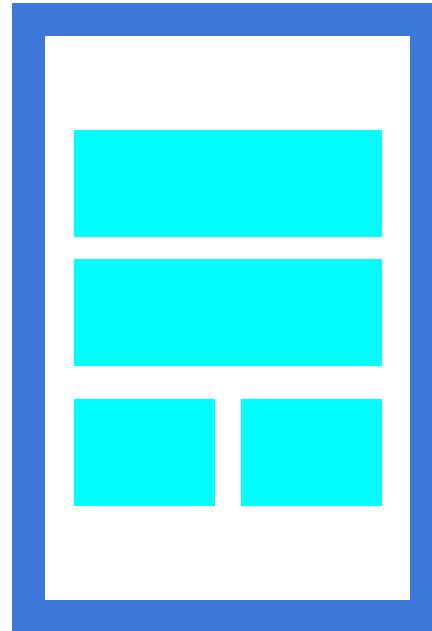
# Image

Format image using

```
Image.asset(  
    file: file_name,  
    height: 100,  
    width: 100,  
    fit: BoxFit.fill,  
    alignment: Alignment.center,  
) ,
```

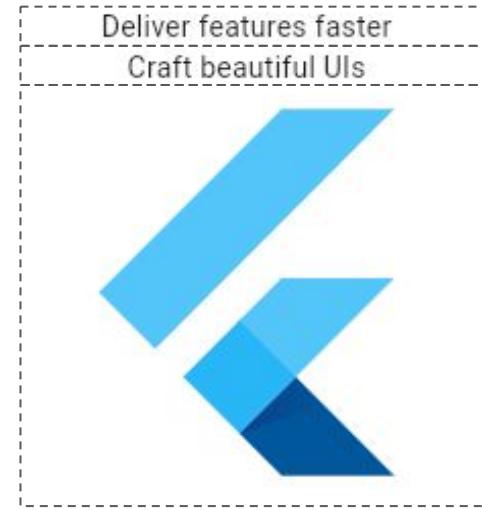
# Row/Column

Arranges widgets horizontally (column) or vertically (row) for layout purposes.



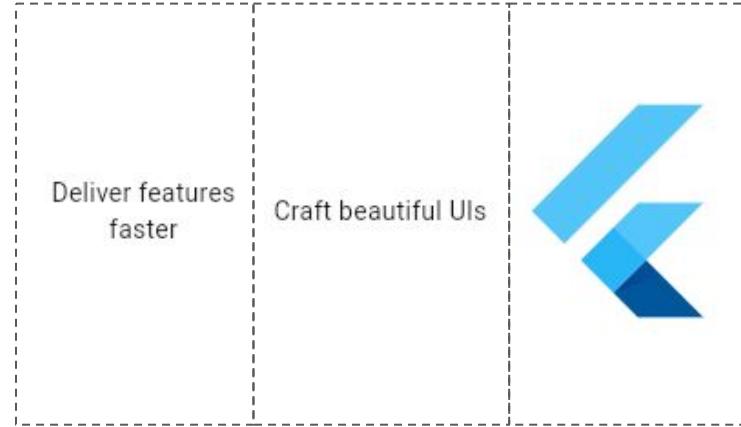
# Row/Column

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



# Row/Column

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



# Icon

Icon: Displays icons.

```
const Row(  
    mainAxisAlignment: MainAxisAlignment.spaceAround,  
    children: <Widget>[  
        Icon(  
            Icons.favorite,  
            color: Colors.pink,  
            size: 24.0,  
            semanticLabel: 'Text to announce in accessibility modes',  
        ),  
        Icon(  
            Icons.audiotrack,  
            color: Colors.green,  
            size: 30.0,  
        ),  
        ...  
    ]  
)
```



# Scaffold

Implements the basic Material Design visual layout structure.

It provides APIs for showing drawers, snack bars, and bottom sheets.

```
Scaffold(  
    appBar: AppBar(  
        title: const Text('Sample Scaffold'),  
    ),  
    body: Center(...),  
);
```

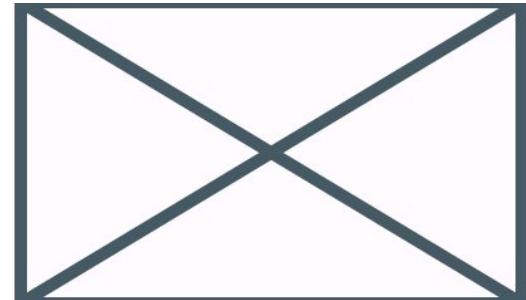


# Placeholder

It represents a place where other widgets will be added later.

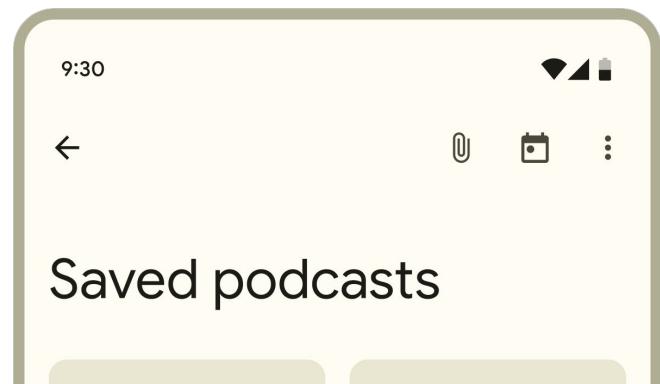
It is useful during development to indicate that the interface is not yet complete.

```
const Placeholder(  
  fallbackHeight: 30.0,  
  fallbackWidth: 100.0,  
  strokeWidth: 5.0,  
)
```



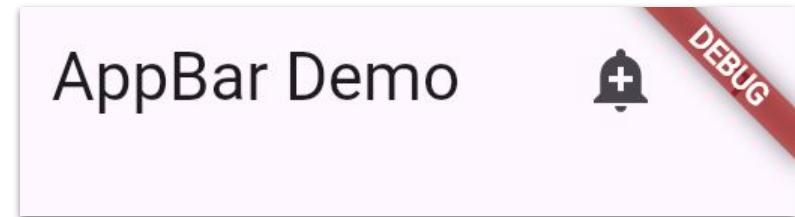
# AppBar

Container that displays content and actions at the top of a screen.



# AppBar

```
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(  
      title: const Text('AppBar Demo'),  
      actions: <Widget>[  
        IconButton(  
          icon: const Icon(Icons.add_alert),  
          tooltip: 'Show Snackbar',  
          onPressed: null,  
        ),  
      ],  
    ),  
    body: Center(  
      child: Text('Hello World'),  
    ),  
  );  
}
```



# Stateless vs Stateful Widget

# Widgets - Stateless vs Stateful

## Stateless widgets

Simpler and have a fixed appearance and behavior

Don't hold any internal state

## Stateful widgets

Maintain their own state, which react to user interactions or changes in data

Manage state through a State object

# Stateless Widget

```
class GreenFrog extends StatelessWidget {  
  const GreenFrog({ super.key });  
  
  @override  
  Widget build(BuildContext context) {  
    return Container(color: const Color(0xFF2DBD3A));  
  }  
}
```

# Stateless Widget

```
class Frog extends StatelessWidget {
  const Frog({
    super.key,
    this.color = const Color(0xFF2DBD3A),
    this.child,
  });

  final Color color;
  final Widget? child;

  @override
  Widget build(BuildContext context) {
    return ColoredBox(color: color, child: child);
  }
}
```

# Stateful Widget

```
class YellowBird extends StatefulWidget {
  const YellowBird({ super.key });

  @override
  State<YellowBird> createState() => _YellowBirdState();
}

class _YellowBirdState extends State<YellowBird> {
  @override
  Widget build(BuildContext context) {
    return Container(color: const Color(0xFFFFE306));
  }
}
```

# Stateful Widget

```
class Bird extends StatefulWidget {  
  const Bird({  
    super.key,  
    this.color = const Color(0xFFFFE306),  
    this.child,  
  });  
  
  final Color color;  
  final Widget? child;  
  
  @override  
  State<Bird> createState() =>  
  _BirdState();  
}
```

```
class _BirdState extends State<Bird> {  
  double _size = 1.0;  
  
  void grow() {  
    setState(() { _size += 0.1; });  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Container(  
      color: widget.color,  
      transform:  
      Matrix4.diagonal3Values(_size,  
                               _size, 1.0),  
      child: widget.child,  
    );  
  }  
}
```

# Design Systems

# Design Systems

## Cupertino

Widgets align with Apple's Human Interface Guidelines for iOS and macOS



## Material Components

Visual, behavioral, and motion-rich widgets implementing the Material 3 design specification for Android, web, and desktop applications.



# Cupertino Widgets

CupertinoPageScaffold: Provides a basic structure for a Cupertino-style app.

CupertinoNavigationBar: Creates a navigation bar for Cupertino-style apps.

CupertinoButton: Creates a button with a Cupertino-style appearance.

# Material Design Widgets

AppBar: Creates an app bar at the top of the screen.

Drawer: Creates a side menu that slides in from the side.

Scaffold: Provides a basic structure for a material design app.

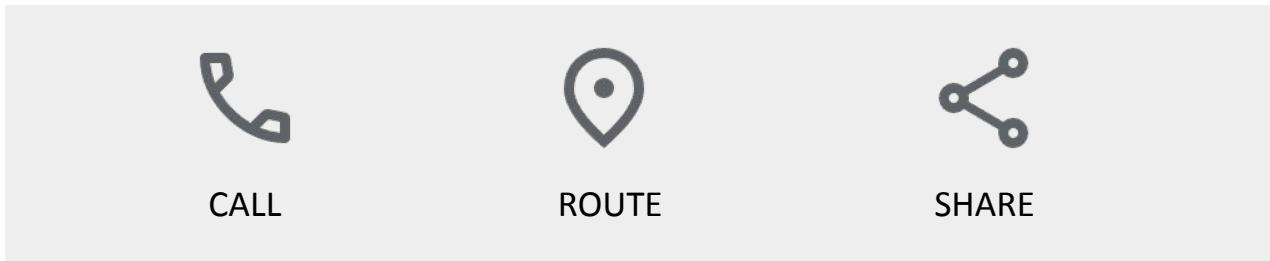
FloatingActionButton: Creates a circular button that floats above the content.

Card: Displays a rectangular card with a shadow.

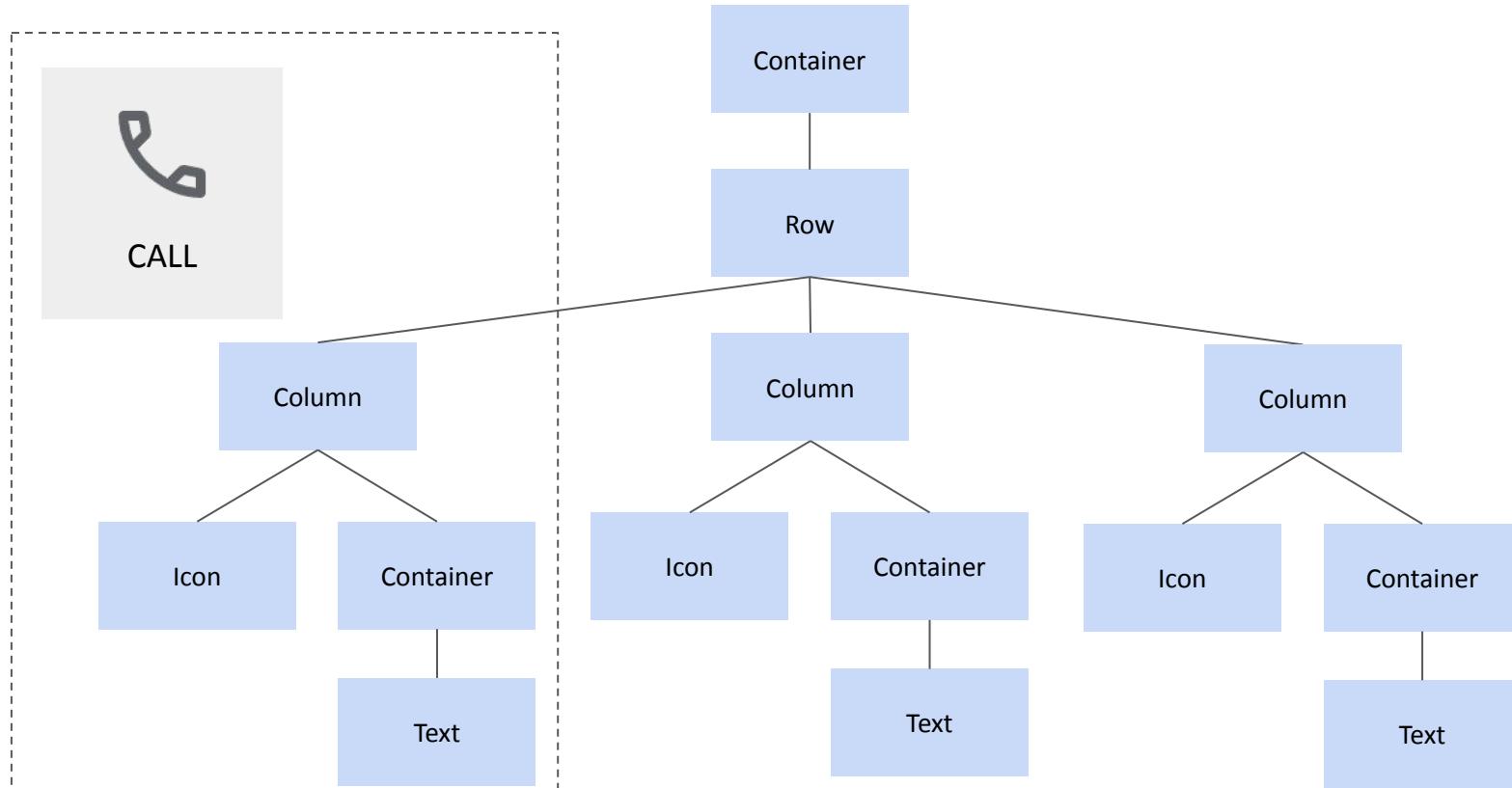
# Layout

# Layout

- A layout is a widget
- A layout is used to build more complex widgets



# Layout



# Layout

- A layout is a widget.
- A layout is used to build more complex widgets.
- Two main categories of layout widgets:
  - Single-child
  - Multi-child

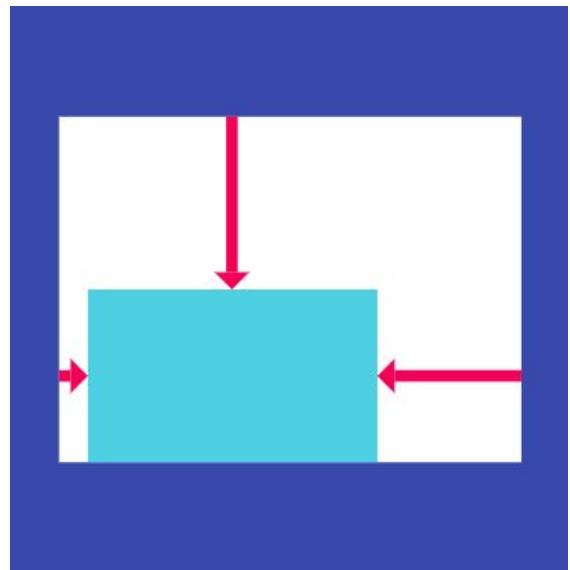
# Single-child layout widgets

# Single-child layout widgets

- These widgets can only hold one child widget.
- They are commonly used for tasks like:
  - Align
  - AspectRatio
  - Center
  - ConstrainedBox
  - Container
  - Padding
  - SizedBox

# Align

A widget that aligns its child within itself and optionally sizes itself based on the child's size.



# Align

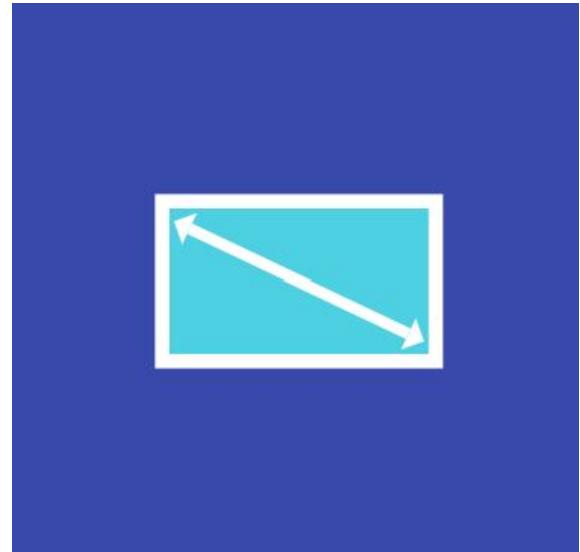
```
Center(  
  child: Container(  
    height: 120.0,  
    width: 120.0,  
    color: Colors.blue[50],  
    child: const Align(  
      alignment: Alignment.topRight,  
      child: FlutterLogo(  
        size: 60,  
      ),  
    ),  
  ),  
)
```

Alignment.topRight



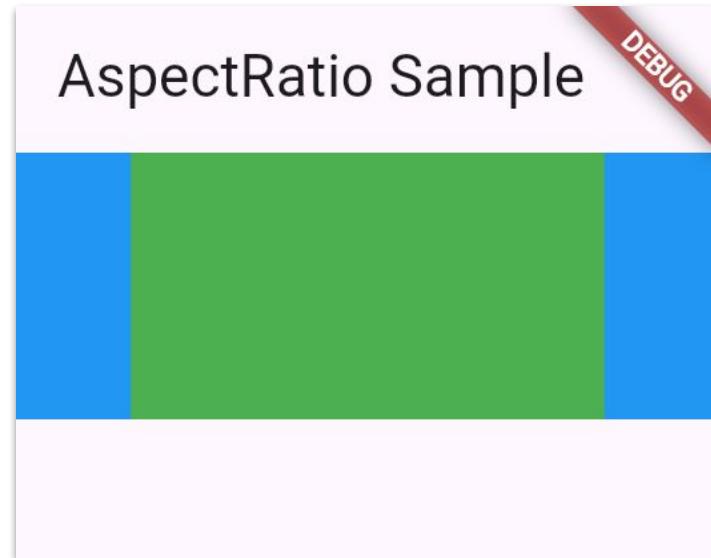
# AspectRatio

A widget that attempts to size the child to a specific aspect ratio.



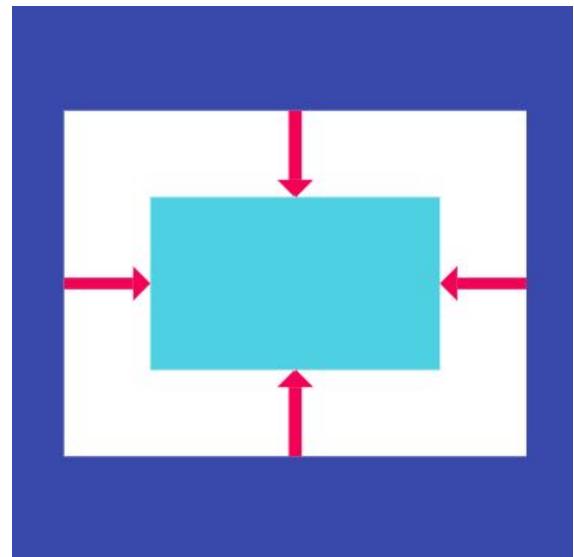
# AspectRatio

```
Container(  
    color: Colors.blue,  
    alignment: Alignment.center,  
    width: double.infinity,  
    height: 100.0,  
    child: AspectRatio(  
        aspectRatio: 16 / 9,  
        child: Container(  
            color: Colors.green,  
        ),  
    ),  
,  
)
```



# Center

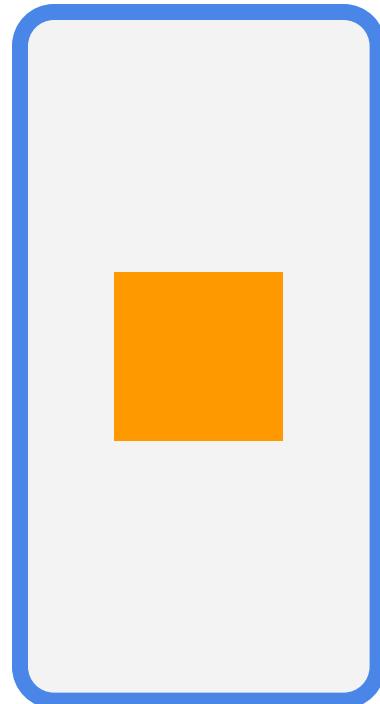
Alignment block that centers its child within itself.



# Container

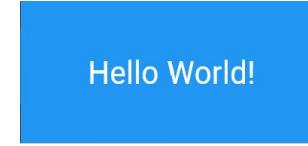
Groups other widgets and provides styling options like padding, margins, and borders.

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



# Container

```
Container(  
    width: 200.0,  
    height: 100.0,  
    color: Colors.blue,  
    padding: EdgeInsets.all(20.0),  
    alignment: Alignment.center,  
    child: Text(  
        "Hello World!",  
        style: TextStyle(  
            color: Colors.white,  
            fontSize: 20.0,  
            ...  
    )  
)
```



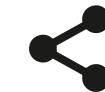
# Container



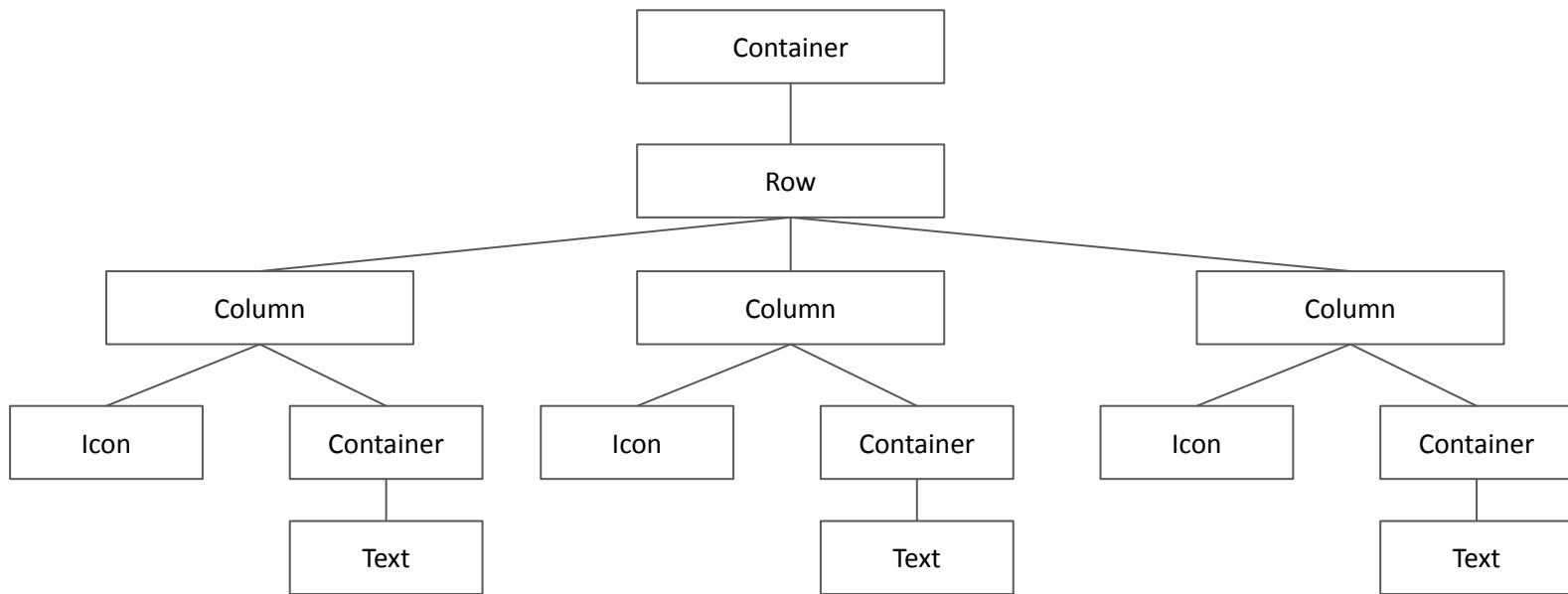
CALL



ROUTE



SHARE



# Padding

A widget that insets its child by the given padding.

```
const Card(  
    child: Padding(  
        padding: EdgeInsets.all(16.0),  
        child: Text('Hello World!'),  
    ),  
)
```



# SizedBox

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

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

# Multi-child layout widgets

# Multi-child layout widgets

These widgets can accommodate multiple child widgets and arrange them in different ways.

Common Multi-child Widgets:

- Row/Column
- GridView
- ListView
- Stack
- Wrap

# Multi-child layout widgets

Key Features:

## 1. Layout Management

These widgets handle the arrangement and positioning of their child widgets within the available space.

## 2. Alignment

You can control how child widgets are aligned within the parent widget using properties like `mainAxisAlignment` (for Row and Column) or `alignment` (for Stack).

# Multi-child layout widgets

Key Features:

## 3. Spacing

Properties like `mainAxisAlignment` (Row/Column) and `spacing` (Row/Column) allow you to control the spacing between child widgets.

## 4. Sizing

Some multi-child widgets offer options to control the size of their child widgets, such as `mainAxisSize` (Row/Column) or `flex` (Flex).

# Row and Column

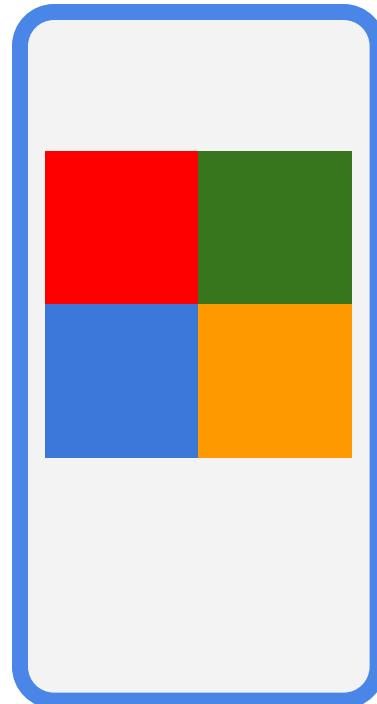
```
Row(  
    children:[  
        Text('Widget 1'),  
        Text('Widget 2'),  
        Text('Widget 3'),  
    ],  
)
```

```
Column(  
    children:[  
        Text('Widget 1'),  
        Text('Widget 2'),  
        Text('Widget 3'),  
    ],  
)
```

# GridView

A scrollable, 2D array of widgets.

```
GridView.count(  
  crossAxisCount: 2,  
  children: [  
    Container(color: Colors.red,),  
    Container(color: Colors.green,),  
    Container(color: Colors.blue,),  
    Container(color: Colors.orange,),  
,  
,
```

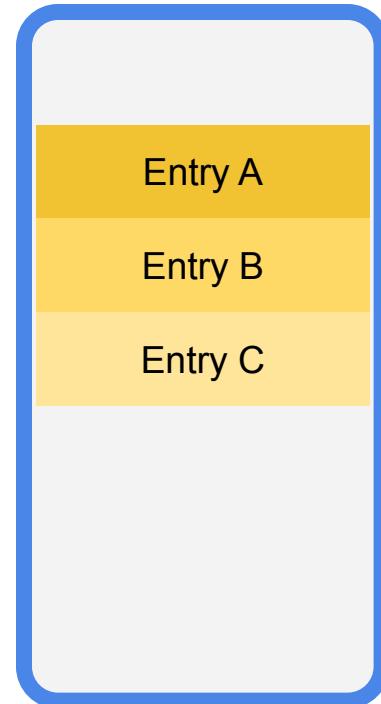


# ListView

A scrollable list of widgets arranged linearly.

Suitable for lists that contain only a few items.

```
body: ListView(  
    children: [  
        Container(height: 50, color: Colors.amber[600],  
            child: const Center(child: Text('Entry A'))),  
        Container(height: 50, color: Colors.amber[500],  
            child: const Center(child: Text('Entry B'))),  
        Container(height: 50, color: Colors.amber[100],  
            child: const Center(child: Text('Entry C'))),  
    ],  
)
```

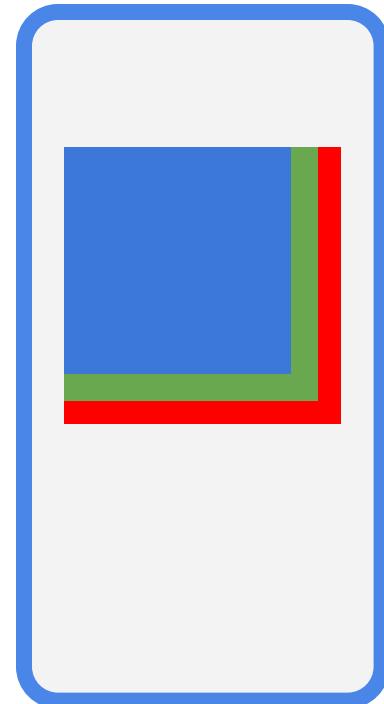


# Stack

A widget that positions its children relative to the edges of its box.

It is useful if you want to overlap several children in a simple way.

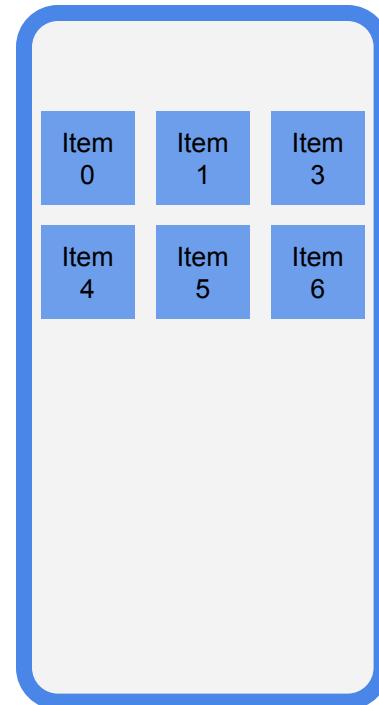
```
Stack(  
  children: <Widget>[  
    Container(width: 100, height: 100,  
      color: Colors.red,),  
    Container(width: 90, height: 90,  
      color: Colors.green,),  
    Container(width: 80, height: 80,  
      color: Colors.blue,),  
,  
)
```



# Wrap

A widget that displays its children in multiple horizontal or vertical runs.

```
Wrap(  
  children: List.generate(6, (index) {  
    return Padding(  
      padding: const EdgeInsets.all(8.0),  
      child: Container( width: 100, height: 100,  
        color: Colors.lightBlue,  
        child: Center(  
          child: Text('Item $index',)),),  
    );  
  }),  
,
```



# Form Widgets

TextField: Allows users to enter text.

Checkbox: Creates a checkbox for user input.

RadioListTile: Creates a radio button for user input.

DropdownButtonFormField: Creates a dropdown menu for selecting a value.

# Layout Widgets

Flex: Provides more control over the layout of its children using flex factors.

Wrap: Wraps its children to the next line when the available space runs out.

CustomScrollView: Allows you to create custom scrolling experiences.

SliverAppBar: Creates an app bar that can expand, collapse, and display different content as the user scrolls.

# Other Widgets

FutureBuilder: Builds a widget based on the result of an asynchronous computation.

StreamBuilder: Builds a widget based on the latest data from a stream.

GestureDetector: Detects gestures on a child widget.

Hero: Creates a hero animation between two widgets in different screens.

# Sliver Widgets

# Sliver widgets

- Sliver widgets are the building blocks for creating custom scrolling experiences.
- They provide a more granular approach to scroll view construction compared to standard widgets like *ListView* or *GridView*.

# Sliver widgets

## Concepts

- Unlike regular box widgets, slivers are designed specifically for scrollable areas.
- They can dynamically adjust their size and shape based on the viewport and scrolling position.
- This allows for creating intricate scrolling effects like parallax effects, expanding headers, and custom layouts within a scrollable view.

# Sliver widgets

## Benefits

- Customization: Sliver widgets offer fine-grained control over scrolling behavior and appearance.
- Performance: For large data sets, slivers can be more efficient than standard scrollable widgets.
- Flexibility: You can combine different slivers to create unique and engaging scrolling experiences.

# Sliver widgets

## Using Sliver Widgets

- Sliver widgets are typically used with the CustomScrollView widget.
- CustomScrollView takes a list of slivers as its slivers property, allowing you to combine different slivers to achieve your desired layout.

# Sliver widgets

Common Sliver Widgets:

- SliverAppBar: Creates an app bar that can expand, collapse, and display different content as the user scrolls.
- SliverList: Renders a list of widgets vertically, similar to ListView, but with more flexibility in sizing and performance optimizations for large lists.
- SliverGrid: Renders a grid of widgets in a similar way to GridView, but within the context of a scrollable area.
- SliverToBoxAdapter: Embeds a regular box widget (like a Text or a Container) within the scrollable area.
- SliverFixedExtentList: Similar to SliverList but forces all its children to have the same height, improving rendering efficiency for uniform lists.

# Performance Best Practices

# Performance Best Practices

## 1. Minimize Widget Rebuilds

Avoid repetitive and costly work in `build()` methods since it can be invoked frequently when ancestor widgets rebuild.

Use `const` for immutable widgets: This tells Flutter that the widget's properties won't change, leading to optimizations.

Utilize `const` constructors: Use `const` constructors for widgets that don't require state.

# Performance Best Practices

## 2. Optimize Layout

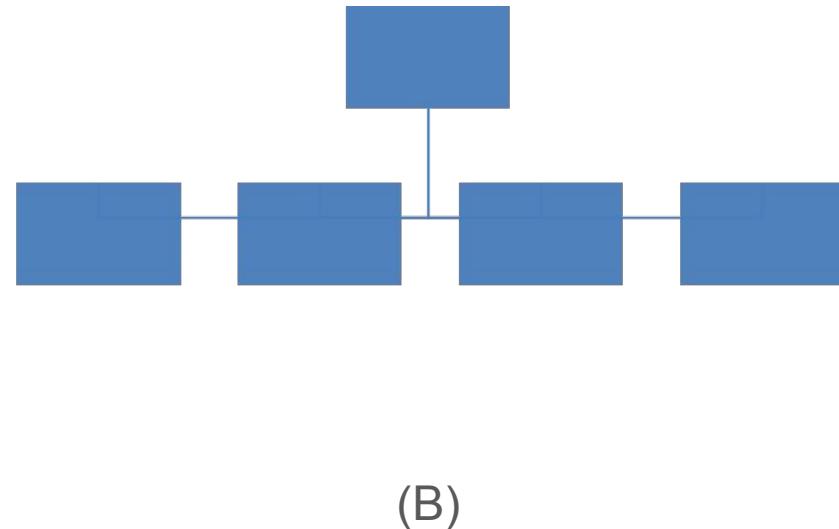
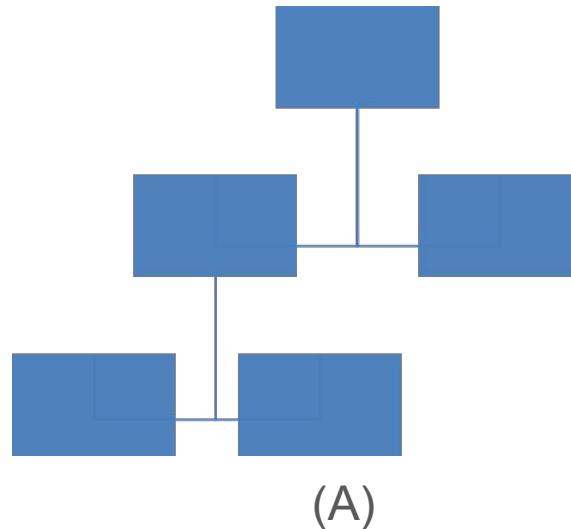
Avoid unnecessary nesting: Keep your widget tree as flat as possible to reduce rendering time.

Use LayoutBuilder: For dynamic layouts that depend on available space, use LayoutBuilder.

# Performance Best Practices

## 2. Optimize Layout

Which layout is better in term of performance?



# Performance Best Practices

## 3. Efficient Image Loading

Use `Image.network`: For network images, use `Image.network` with appropriate caching and loading strategies.

Optimize Image Size: Resize images to the appropriate size for your app.

Use efficient image formats like WebP, which is a raster graphics file format to replace JPEG, PNG and GIF.

# Performance Best Practices

## 4. State Management

Use `setState` judiciously: Only call `setState` when necessary to trigger a rebuild.

Consider state management solutions: For complex state management, explore options like Provider, Riverpod (builds on the Provider package), or Business Logic Component (BLoC).

# Performance Best Practices

## 5. Use Flutter's Build-in Widget

Use Flutter's optimized widgets whenever possible.

# Find Out More

Widgets, <https://docs.flutter.dev/ui/widgets>

TextField, <https://docs.flutter.dev/cookbook/forms/text-input>

Layout, <https://docs.flutter.dev/ui/layout>

# Chapter 3.2

Platforms, Themes, and Design for Everyone

# Contents

Colors

Material Design and Themes

Design for Everyone

# Colors

# Colors

- Color and ColorSwatch constants which represent Material design color palette.
- Most swatches have colors from 100 to 900 in increments of one hundred, plus the color 50.
- The smaller the number, the more pale the color. The greater the number, the darker the color.
- The accent swatches (e.g. redAccent) only have the values 100, 200, 400, and 700.

# Color Class

```
Color c1 = const Color(0xFFCE4EC);
```



```
Color c2 = const Color.fromARGB(0xFF, 0x42, 0xA5, 0xF5);
```

```
Color c3 = const Color.fromARGB(255, 66, 165, 245);
```

```
Color c4 = const Color.fromRGBO(66, 165, 245, 1.0);
```

# Colors

Select a specific color from one of the swatches:

```
// Selects a mid-range green.  
Color selection = Colors.green[400]!;
```

Each ColorSwatch constant is a color and can used directly:

```
Container(  
    // same as Colors.blue[500] or Colors.blue.shade500  
    color: Colors.blue,  
)
```

# ColorSwatch

ColorSwatch is a class that represents a collection of colors related to a single color theme.

It's useful for creating color palettes that vary in shades, tints, and tones of a base color.

```
ColorSwatch myColorSwatch =  
    MaterialColor(0xFF007AFF, {  
        50: Color(0xFF007AFF),  
        100: Color(0xFF007AFF),  
        200: Color(0xFF007AFF),  
        300: Color(0xFF007AFF),  
        400: Color(0xFF007AFF),  
        500: Color(0xFF007AFF),  
        'arrow': Color(0xFF20877E),  
        'border': Color(0xFF20877E),  
    });
```

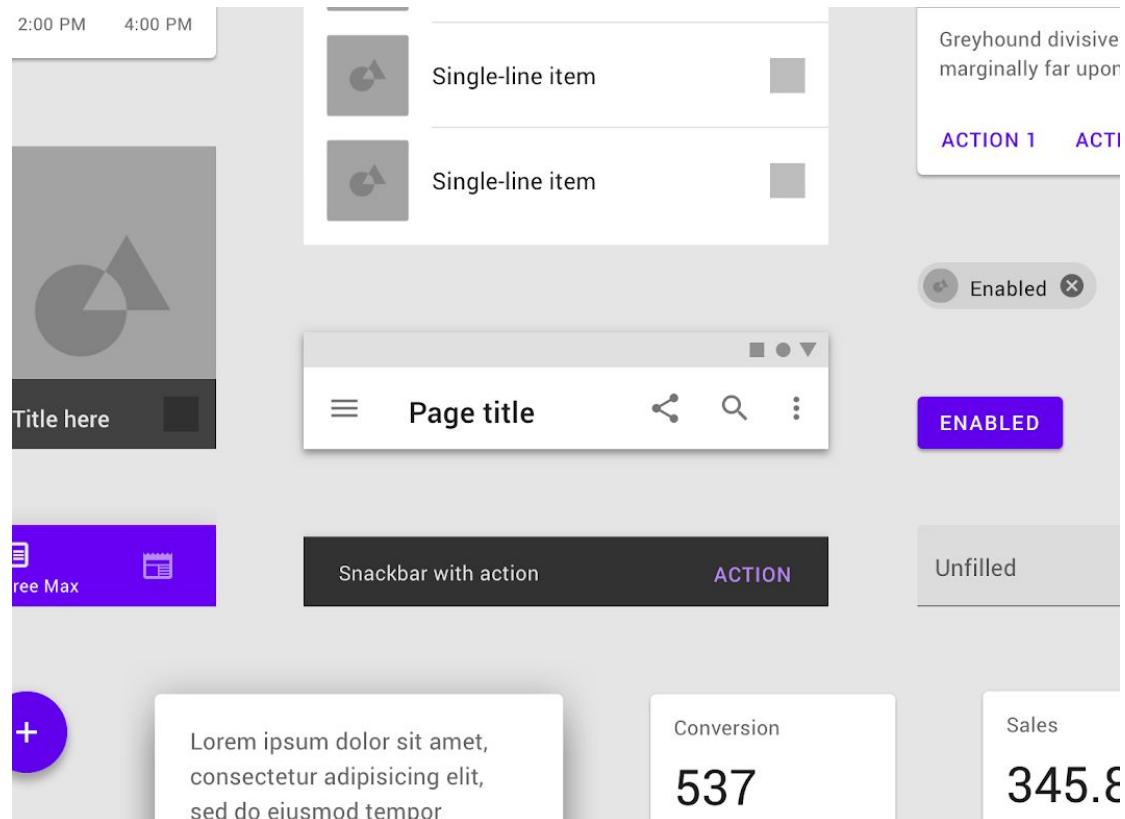
```
Container(  
    // Access a specific shade  
    color: myColorSwatch[500], )  
  
border: Border.all(  
    color: myColorSwatch['border'],  
) ,
```

# Material Design and Themes

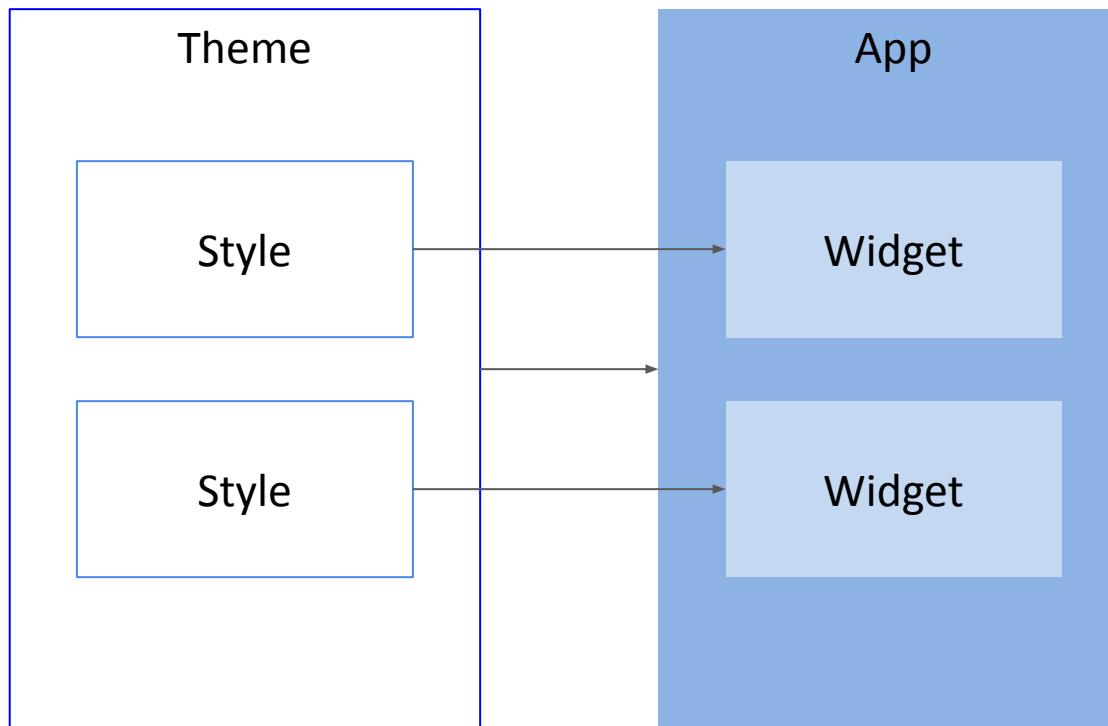
# Material Design

Material Design is an open-source design system built and supported by Google designers and developers.

Default UI design for Flutter app.



# Styles and Themes



# Themes

To share colors and font styles throughout an app.

You can define app-wide themes.

You can extend a theme to change a theme style for one component.

Each theme defines the colors, type style, and other parameters applicable for the type of Material component.

# Themes

Flutter applies styling in the following order:

1. Styles applied to the specific widget.
2. Themes that override the immediate parent theme.
3. Main theme for the entire app.

# Themes

Most instances of ThemeData set values for the following two properties. These properties affect the entire app.

- colorScheme defines the colors.
- textTheme defines text styling.

```
MaterialApp(  
    title: appName,  
    theme: ThemeData(  
        useMaterial3: true,  
  
        // Define the default brightness and colors.  
        colorScheme: ColorScheme.fromSeed(  
            seedColor: Colors.purple,  
            // ...  
            brightness: Brightness.dark,  
        ),  
  
        // Define the default `TextTheme`. Use this to specify the default  
        // text styling for headlines, titles, bodies of text, and more.  
        textTheme: TextTheme(  
            displayLarge: const TextStyle(  
                fontSize: 72,  
                fontWeight: FontWeight.bold,  
            ),  
            // ...  
            titleLarge: GoogleFonts.oswald(  
                fontSize: 30,  
                fontStyle: FontStyle.italic,  
            ),  
            bodyMedium: GoogleFonts.merriweather(),  
            displaySmall: GoogleFonts.pacifico(),  
        ),  
    ),
```

Using GoogleFonts  
from pub.dev package

```
Container(  
  padding: const EdgeInsets.symmetric(  
    horizontal: 12,  
    vertical: 12,  
> ),  
  color: Theme.of(context).colorScheme.primary,  
  child: Text(  
    'Text with a background color',  
    // ...  
    style: Theme.of(context).textTheme.bodyMedium!.copyWith(  
      color: Theme.of(context).colorScheme.onPrimary,  
    ),  
> ),  
> ),
```

# Override a Theme

To override the overall theme in part of an app, wrap that section of the app in a [Theme](#) widget.

You can override a theme in two ways:

1. Create a unique [ThemeData](#) instance.
2. Extend the parent theme.

# Set a unique ThemeData instance

If you want a component of your app to ignore the overall theme, create a ThemeData instance. Pass that instance to the Theme widget.

```
Theme(  
    // Create a unique theme with `ThemeData`.  
    data: ThemeData(  
        colorScheme: ColorScheme.fromSeed(  
            seedColor: Colors.pink,  
        ),  
    ),  
    child: FloatingActionButton(  
        onPressed: () {},  
        child: const Icon(Icons.add),  
    ),  
);
```

# Fonts and Typography

Typography covers the style and appearance of fonts.

Font Style:

1. **Typeface**: a set of common character rules. E.g.Roboto
2. **Size**: E.g. Size 18
3. **Style**: E.g. Regular, Italic
4. **Weight**: E.g. Normal, Bold

# Fonts and Typography

TextTheme could be apply to:

- Display
- Headline
- Title
- Label
- Body

Size Variation:

- Small
- Medium
- Large

Display Large

Display Medium

Display Small

Headline Large

Headline Medium

Headline Small

Title Large

Title Medium

Title Small

Label Large

Label Medium

Label Small

Body Large

Body Medium

Body Small

# Google Fonts Tester

<https://fonts.google.com/>

The screenshot shows the Google Fonts Tester interface. On the left, there's a sidebar with icons for different font families: A (Alegreya), Noto (Noto Sans), Icons (Material Icons), and FAQ. The main area features the Google Fonts logo and a search bar. Below the search bar are navigation links: Specimen, Type tester (which is currently selected), Glyphs, About, and License. A blue "Get font" button is on the right. The main content area displays the "Roboto" font in large, bold letters. Below it, it says "Designed by Christian Robertson". There are dropdown menus for "Select preview text", "Writing system", and "Language". At the bottom, there's a toolbar with buttons for "Heading", font size "48px", font weight "Regular 400", and various styling options like italic, underline, and bullet lists.

Whereas recognition of the inherent dignity

No one shall be subjected to arbitrary arrest, detention or exile. Everyone is entitled in full equality to a fair and public hearing by an independent and impartial tribunal, in the

## Serif vs Sans-Serif

**Serif**

A<sup>1</sup>b<sub>2</sub>c

***Sans-Serif***

A<sub>1</sub>b<sub>2</sub>c

# Serif vs Sans-Serif

**Which font type is easier to read?**

Serif 30

Serif 24

Serif 18

Serif 14

Serif 12

Sans-Serif 30

Sans-Serif 24

Sans-Serif 18

Sans-Serif 14

Sans-Serif 12

# Design for Everyone

# Design for Everyone

Design for Everyone is a design philosophy that aims to create products and services that are **accessible** and **usable** by as many people as possible, regardless of their **age**, **ability**, or **background**.

In the context of **mobile app development**, it means making your app accessible to as many people as possible, regardless of their **ability**, **language**, or **device**.

# Design for Everyone

Methods:

1. Internationalizing
2. Supporting different screens
3. Make apps more accessible

# Internationalizing

# Internationalizing

Internationalizing your app involves making it adaptable to different **languages**, **cultures**, and **regions**.

1. Language Translation
2. Generate Localization Delegates
3. Date, Number and Currency Formatting
4. Text Direction
5. Cultural Nuances

# Setting up

Then import the flutter\_localizations library and specify localizationsDelegates and supportedLocales for your MaterialApp or CupertinoApp:

```
import 'package:flutter_localizations/flutter_localizations.dart';  
  
import 'package:intl/intl.dart' as Intl;
```

# Setting up

The pubspec.yaml file has the following entries:

**dependencies:**

**flutter:**

**sdk: flutter**

**flutter\_localizations:**

**sdk: flutter**

**intl: any**

# Setting up

```
return const MaterialApp(  
  title: 'Localizations Sample App',  
  localizationsDelegates: [  
    GlobalMaterialLocalizations.delegate,  
    GlobalWidgetsLocalizations.delegate,  
    GlobalCupertinoLocalizations.delegate,  
  ],
```

```
  supportedLocales: [  
    Locale('en'), // English  
    Locale('es'), // Spanish  
  ],  
  home: MyHomePage(),  
);
```

# Language Translation

Create .arb files (Android Resource Bundle) for each language.

Use the generate\_localized\_json package to generate Dart code from the .arb files.

```
// en.arb - English
{
  "helloWorld": "Hello, world!",
  "greeting": "Hello, {name}!"
}
```

```
//fr.arb - French
{
  "helloWorld": "Bonjour, le monde!",
  "greeting": "Bonjour, {name}!"
}
```

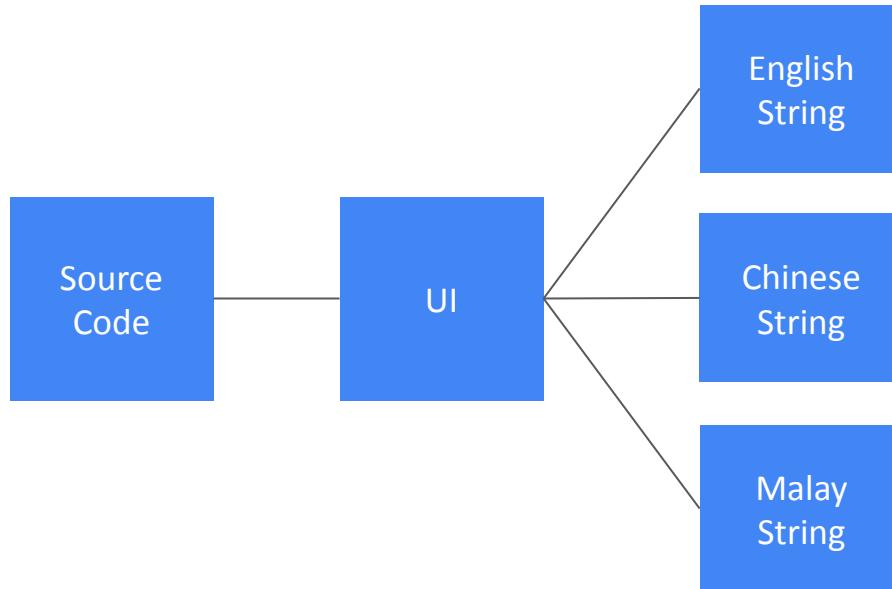
# Language Translation

Mark translatable strings: Use `Intl.message()` to mark strings that need translation.

```
Text(Intl.message('Hello, world!')),
```

# Create Locale Directories and String Files

At runtime, the system uses the appropriate set of string resources based on the locale currently set for the user's device.



# Local Currency Symbol

```
final myCurrency = Intl.NumberFormat('#,##0.00', 'ms_MY');  
  
symbol = myCurrency.currencySymbol;  
  
output = myCurrency.format(total);
```

# ISO 639 Language code

Format: [language code]\_[country code]

Language	Language Code	Country Code	E.g.
US English	en	US	en_US
Malay (Malaysia)	ms	MY	ms_MY
Chinese (China)	zh	CN	zh_CN

ISO Language Code, [https://en.wikipedia.org/wiki/List\\_of\\_ISO\\_639\\_language\\_codes](https://en.wikipedia.org/wiki/List_of_ISO_639_language_codes)

ISO Country Code, [https://en.wikipedia.org/wiki/List\\_of\\_ISO\\_3166\\_country\\_codes](https://en.wikipedia.org/wiki/List_of_ISO_3166_country_codes)

# Supporting Different Screens

# Supporting Different Screens

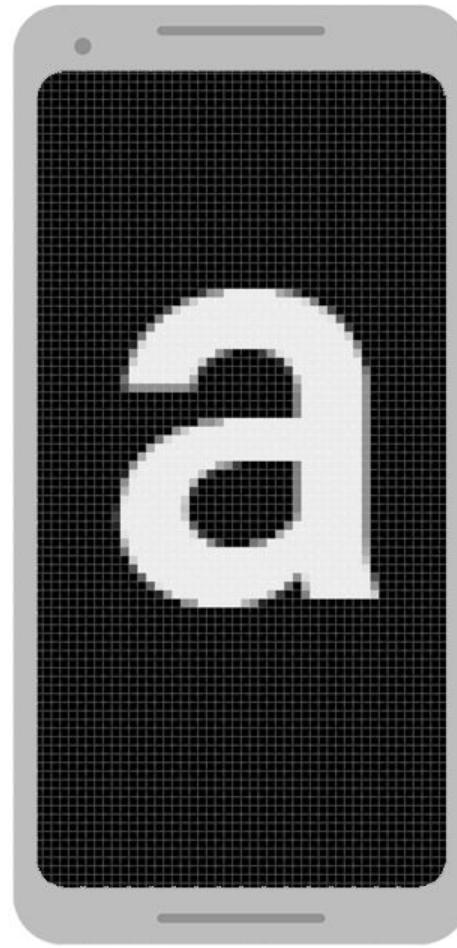
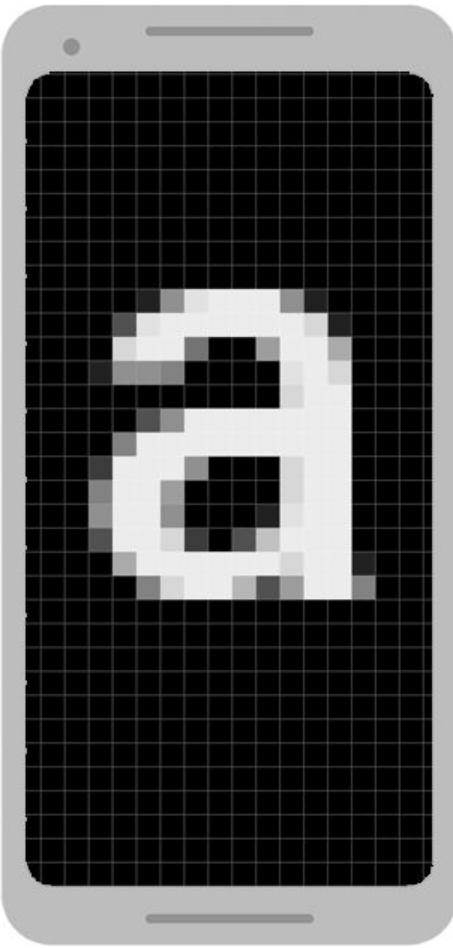
Categories of mobile device screens:



HD  
(720 x 1280)

FHD  
(1080 x 1920)

QHD  
(1440 x 2560)



# Supporting Different Screens

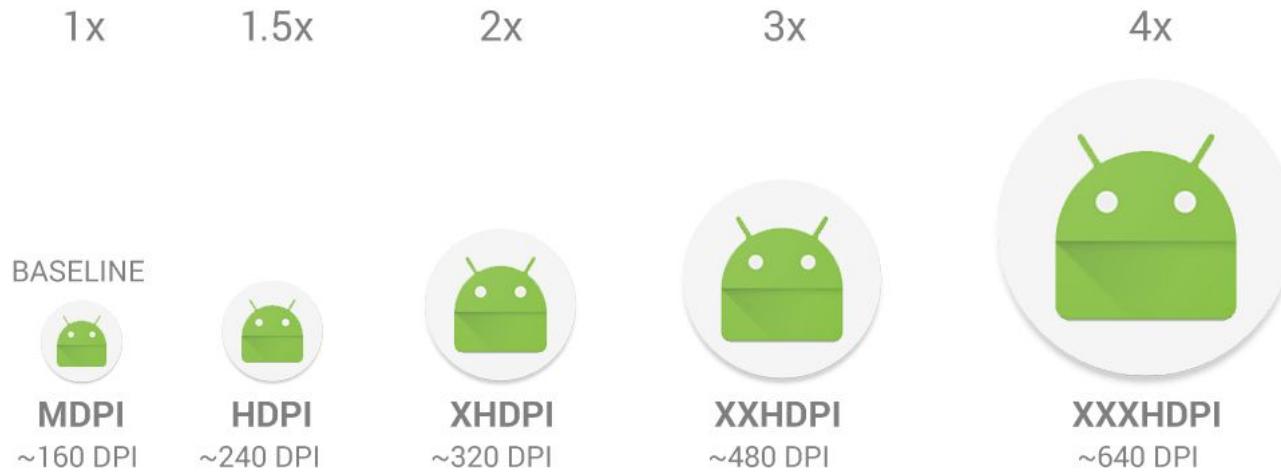
## Adaptive Design

1. Utilize `Flexible` and `Expanded`: These widgets help you distribute space among child widgets.
2. Responsive Images: Use `FittedBox` to scale images to fit the available space.

# Supporting Different Screens

## Platform-Specific Considerations

In the context of Android and iOS, you can provide alternative bitmap resources for good graphical quality and performance



# Supporting Different Screens

Create a directory named `assets` in your Flutter project's root directory.

Within this directory, create subfolders for each resolution: `1x`, `2x`, `3x`, etc.

```
assets/  
  images/  
    logo.png  
    logo@2x.png  
    logo@3x.png
```

# Supporting Different Screens

Add the assets section to your pubspec.yaml file:

```
flutter:  
  assets:  
    - assets/images/logo.png
```

# Supporting Different Screens

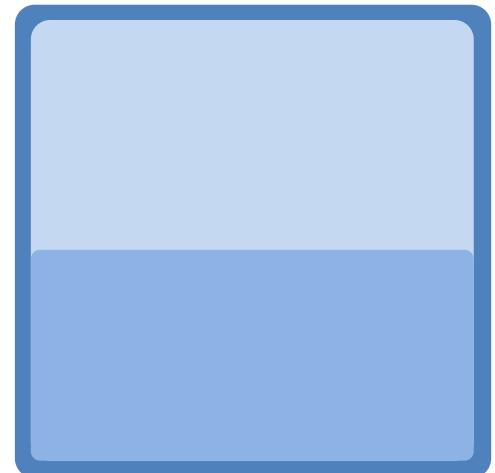
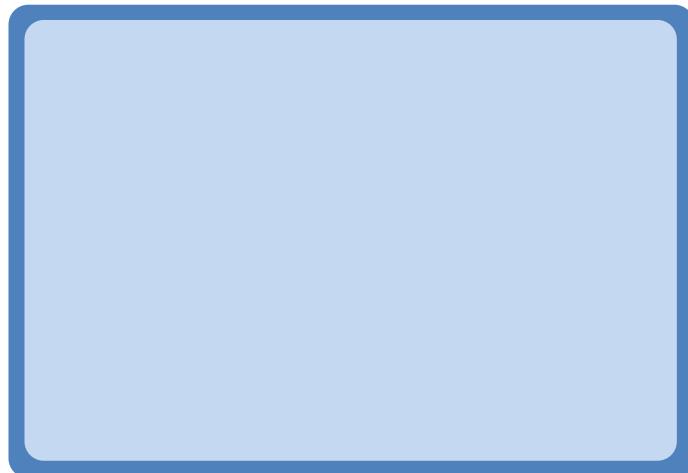
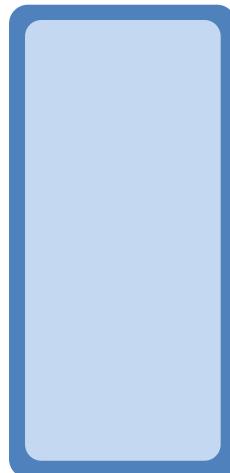
Use the `Image.asset()` widget to load the appropriate asset based on the device's screen density:

```
Image.asset('assets/images/logo.png'),
```

The system selects the most suitable asset based on the device's pixel density.

# Supporting Different Screens

You should also consider screen orientation and slit screen.



# Supporting Different Screens

You can build different layouts depending on a given Orientation.

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

class MyWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: MediaQuery.of(context).orientation ==
          Orientation.portrait ? PortraitLayout() : LandscapeLayout(),
    );
  }
}
```

# Supporting Different Screens

To force your app to a specific orientation, you can use the `SystemChrome` class:

```
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  SystemChrome.setPreferredOrientations([
    DeviceOrientation.portraitUp,
  ]);
  runApp(const MyApp());
}
```

# Support Display Cutouts



Tear  
Drop



Notch



Punch Hole  
(Middle)



Punch Hole  
(Left)

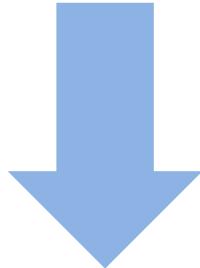


Punch Hole  
(Right)

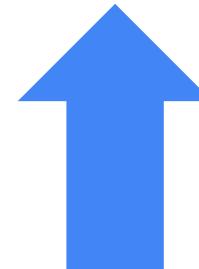
# Support Display Cutouts



# Specify Minimum and Target API Levels



minSdkVersion = the lowest API level with which your app is compatible



targetSdkVersion = the highest API level against which you've designed and tested your app

# Specify Minimum and Target API Levels

In the Project settings:

```
android {  
    . . .  
  
    defaultConfig {  
        . . .  
        minSdkVersion 16  
        targetSdkVersion 34  
        . . .  
    }  
}
```

To support more Android devices

# Specify Minimum and Target API Levels

In the Project settings:

```
android {  
    . . .  
  
    defaultConfig {  
        . . .  
        minSdkVersion 16  
        targetSdkVersion 34  
        . . .  
    }  
}
```

To include the latest OS features

# Make Apps More Accessible

# Accessibility

Regardless of ability, users are able to navigate, understand, and use an app successfully

Consideration:



Navigation



Readability



Guidance and Feedback

# Accessibility - Navigation

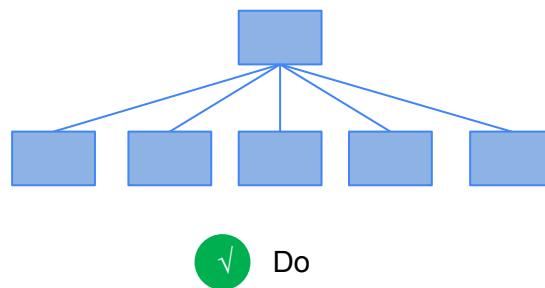
## 1. Support screen readers by Semantics.

```
Semantics(  
    label: 'This is a button',  
    child: ElevatedButton(  
        onPressed: () {},  
        child: Text('Click Me'),  
    ),  
,
```

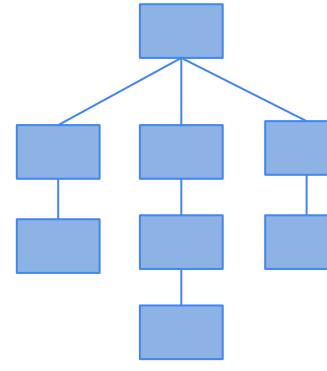
# Accessibility - Navigation

## 2. Create easy-to-follow navigation

- Support keyboards or gestures input
- Avoid having UI elements fade out or disappear after a certain amount of time
- Create flat navigation structure



✓ Do



✗ Don't

# Accessibility - Navigation

## 3. Make touch targets large

- Min size 48 x 48 pixel
- Space between elements min 8dp



Do



Don't

48 px

Cancel

Save



Save

Cancel

Save



Save



Do

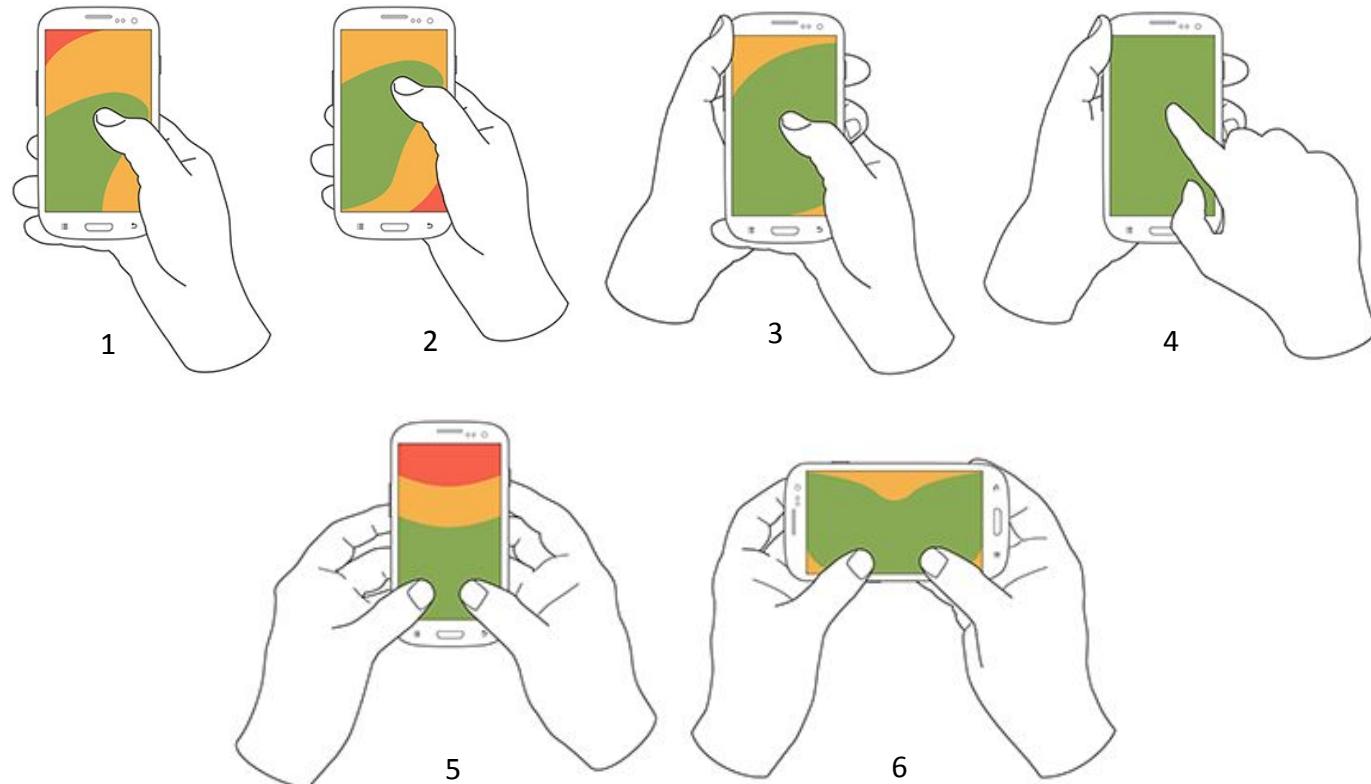


X

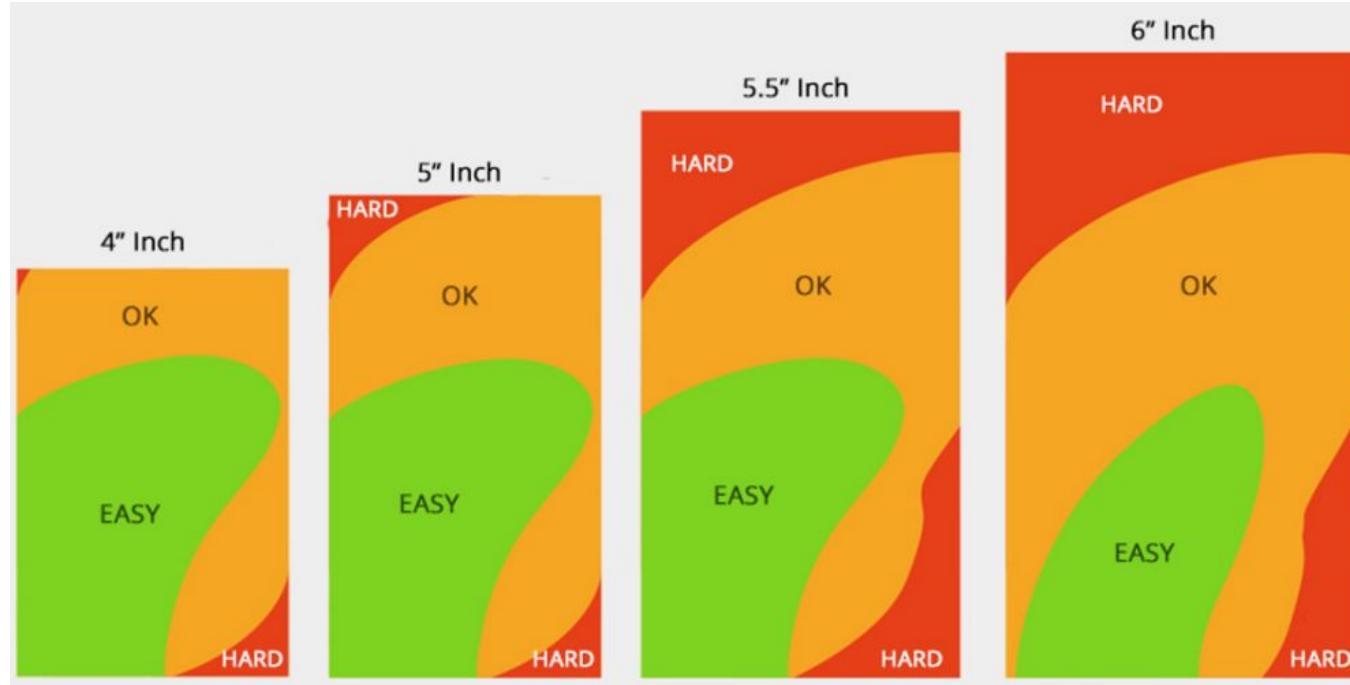
Don't

Index finger fits snugly inside.  
Target edges give visual feedback.

# How users hold their phones?



# Touch Zone



<https://www.probeseven.com/blog/mobile-uiux-design-considerations-outdoors-visually-challenged/>

# Gesture Navigation

Android 10 (API 29) supports fully gestured-based navigation

Ensure your apps:

- Extend content from edge to edge
- Handle conflicting gestures

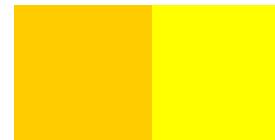


# Accessibility - Readability

## 4. Provide adequate colour contrast

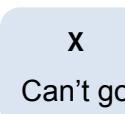
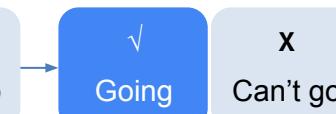
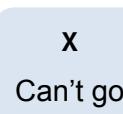
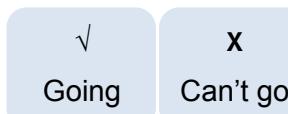


Do



Don't

Use more than just colour to convey information



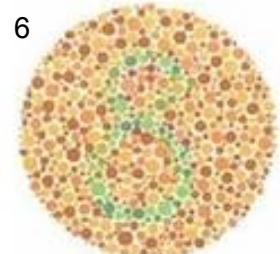
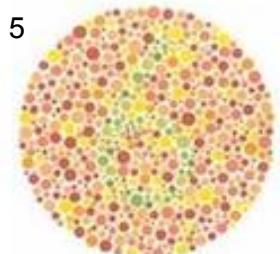
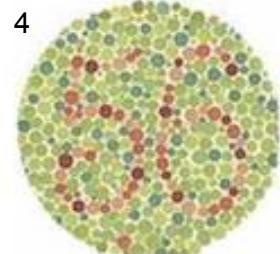
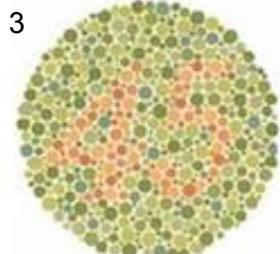
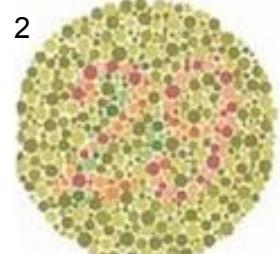
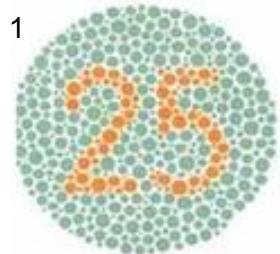
Do



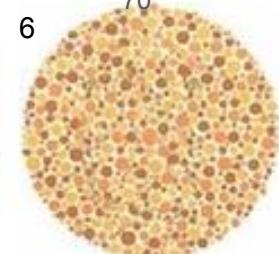
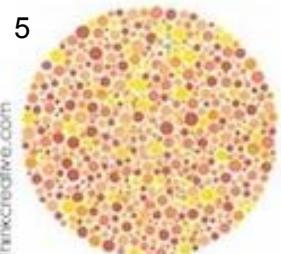
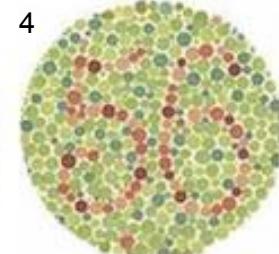
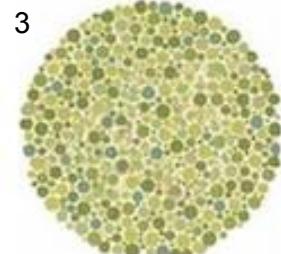
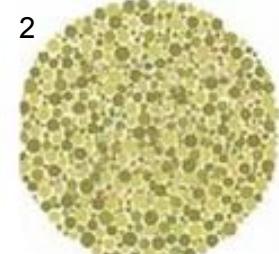
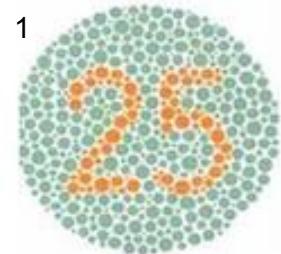
Don't

## Ishihara Test For Color Blindness

What People With Regular Vision  
See



What People With Color Blindness  
See



70

For Normal peoples

YES

Green

NO

Red

For Deuteranopia peoples

YES

NO

Shades of yellow

For Protanopia peoples

YES

Shades of yellow

NO

For Tritanopia peoples

YES

blue

NO

Red

# The Material Design Tools

<https://m2.material.io/design/color/the-color-system.html#tools-for-picking-colors>

Red 50	#FFEBEE	Pink 50	#FCE4EC	Purple 50	#F3E5F5
100	#FFCDD2	100	#F8BBD0	100	#E1BEE7
200	#EF9A9A	200	#F48FB1	200	#CE93D8
300	#E57373	300	#F06292	300	#BA68C8
400	#EF5350	400	#EC407A	400	#AB47BC
500	#F44336	500	#E91E63	500	#9C27B0
600	#E53935	600	#D81B60	600	#8E24AA
700	#D32F2F	700	#C2185B	700	#7B1FA2
800	#C62828	800	#AD1457	800	#6A1B9A
900	#B71C1C	900	#880E4F	900	#4A148C
A100	#FF8A80	A100	#FF80AB	A100	#EA80FC

# Accessibility - Readability

## 5. Make media content more accessible

- Include controls for users to pause or stop video and audio files
- Provide transcript/caption

# Accessibility - Guidance and Feedback

## 6. Make interactive controls clear and discoverable

- Interactive controls have text labels, tooltips, or placeholder text to indicate their purpose
- When naming elements, be consistent in your terminology throughout your app.

# Find Out More

Colors, <https://api.flutter.dev/flutter/dart-ui/Color-class.html>

Themes, <https://docs.flutter.dev/cookbook/design/themes>

Internationalization,

<https://docs.flutter.dev/ui/accessibility-and-internationalization/internationalization>

# Chapter 3.3

State Management and Navigation principles

# Contents

State Management

Navigation Principles

Navigation

Tab

Drawer

Dialog

Picker

Snackbar

# State Management

# State Management

State management involves managing the data that changes over time and triggers UI updates.

State of an app is everything that exists in memory when the app is running.

This includes the app's assets, all the variables that the Flutter framework keeps about the UI, animation state, textures, fonts, and so on.

# State Management

## Ephemeral State

Also called UI state or local state.

It is the state you can neatly contain in a single widget.

## App State

Also called shared state

UI state that you want to share across many parts of your app, and that you want to keep between user sessions

# State Management

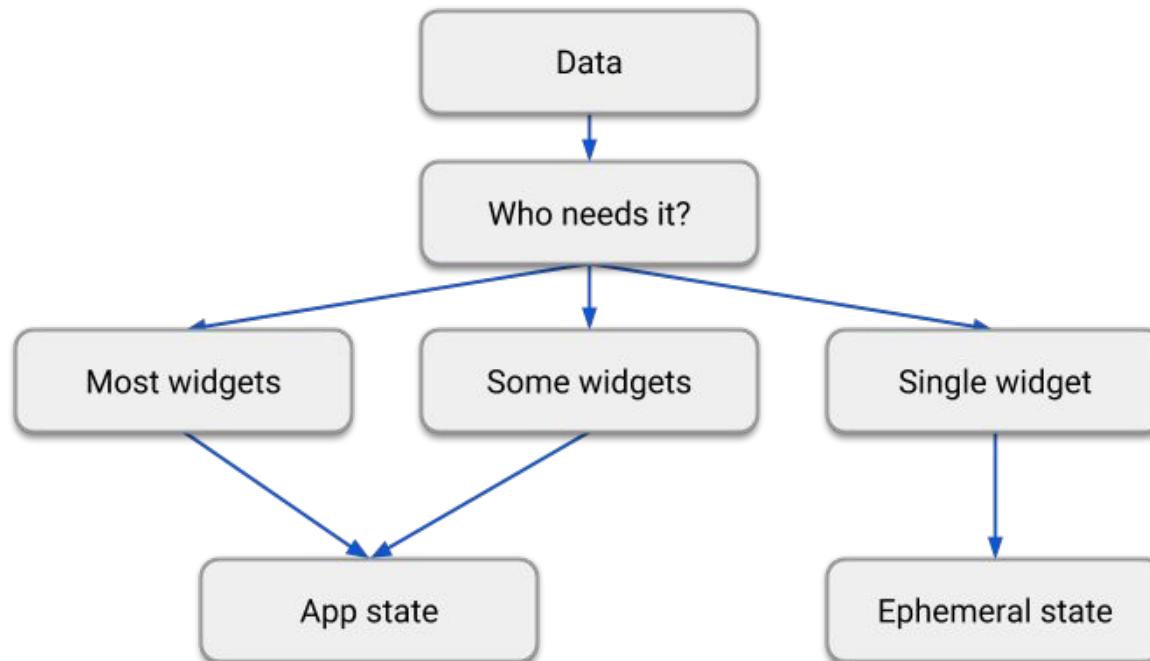
## Examples of Ephemeral State:

- current page in a PageView
- current progress of a complex animation
- current selected tab in a BottomNavigationBar

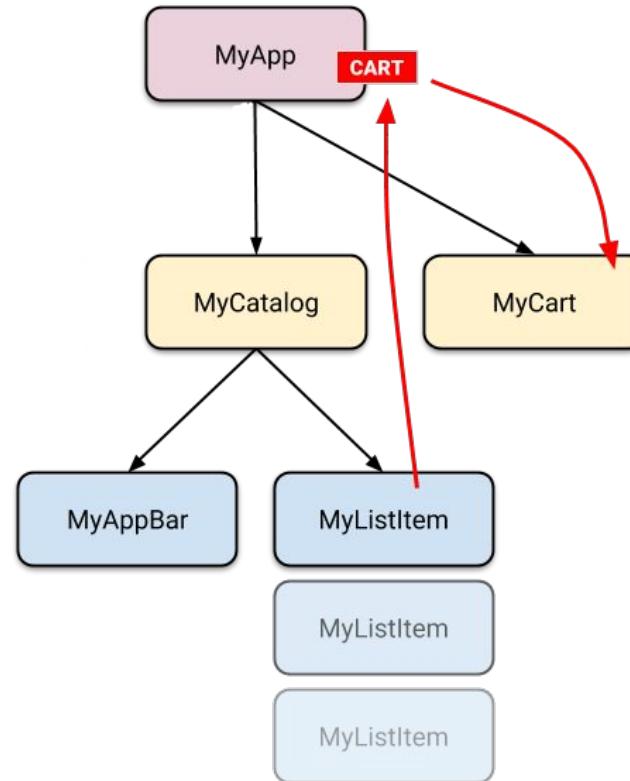
## Examples of Application State:

- user preferences
- login info
- notifications in a social networking app
- the shopping cart in an e-commerce app
- read/unread state of articles in a news app

# State Management



# State Management



# State Management - Provider

Provides a simple way to share state across multiple widgets.

Uses ChangeNotifier to notify listeners when the state changes.

**dependencies:**

**flutter:**

**sdk:** flutter

**provider:**version

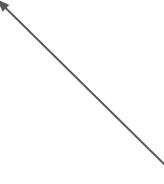
# State Management - Provider

Insert dependency:

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

# State Management - Provider

```
class CounterProvider extends ChangeNotifier {  
    int _counter = 0;  
  
    int get counter => _counter;  
  
    void increment() {  
        _counter++;  
  
        notifyListeners();  
    }  
}
```



It notify listeners when  
the state changes

# State Management - Provider

```
void main() {  
  runApp(  
    ChangeNotifierProvider(  
      create: (context) => CounterProvider(),  
      child: someWidget,  
    ),  
  );  
}
```

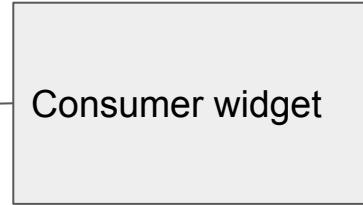
# State Management - Provider

Widget with multiple providers:

```
MultiProvider(  
  providers: [  
    ChangeNotifierProvider(create: (context) => Provider1()),  
    ChangeNotifierProvider(create: (context) => Provider2()),  
  ],  
  child: someWidget,  
,
```

# State Management - Provider

```
body: Center(  
    child: Consumer<CounterProvider>(  
        builder: (context, counter, child) {  
            return Text(counter.counter,);  
        },  
    ),  
,  
  
floatingActionButton: FloatingActionButton(  
    onPressed: () {  
        context.read<CounterProvider>().increment();  
    },  
    tooltip: 'Increment',  
    child: const Icon(Icons.add),  
,
```



# Navigation principles

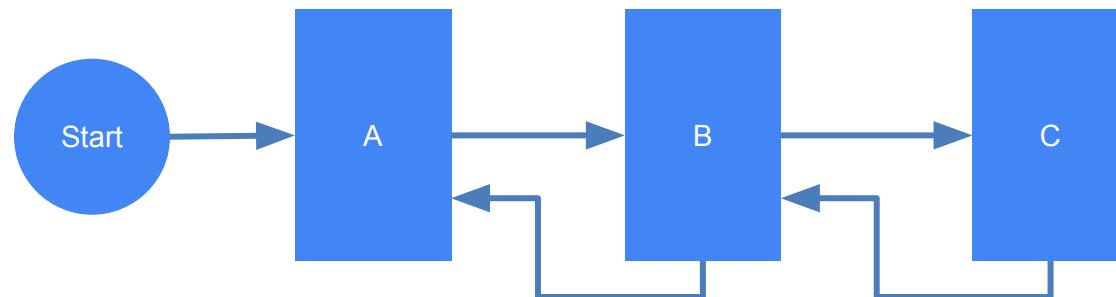
# Navigation Principles

1. Fixed start destination
2. Navigation state is represented as a stack of destinations
3. Up and Back are identical within your app's task
4. The Up button never exits your app
5. Deep linking simulates manual navigation

# Navigation Principles

## 1. Fixed start destination

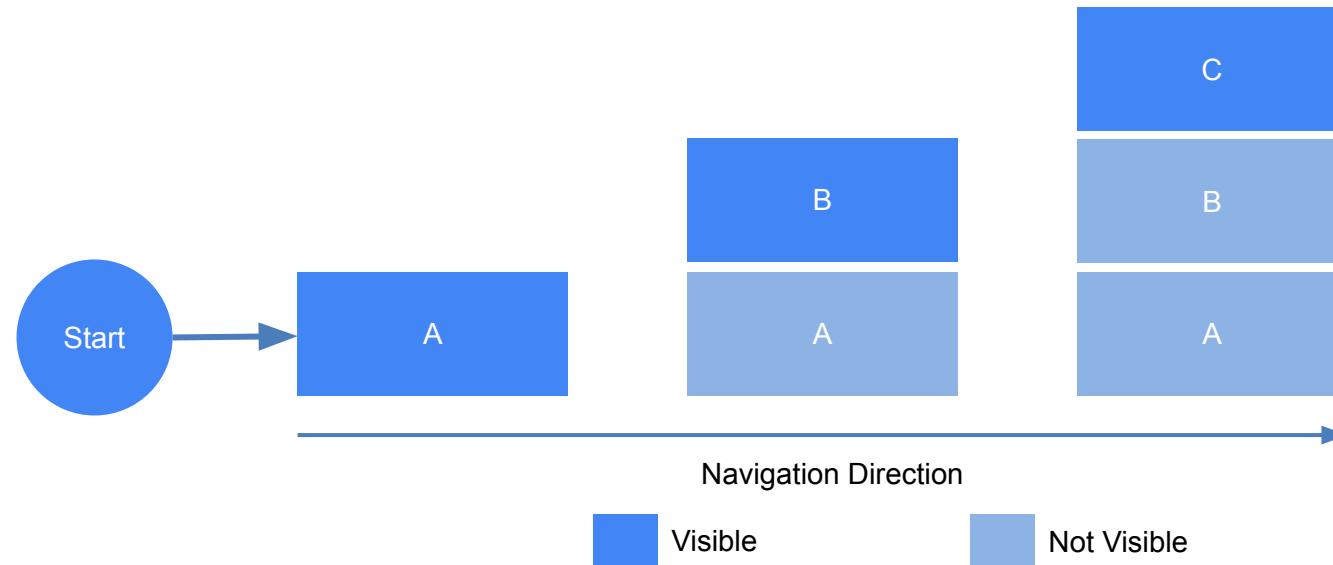
Every app you build has a fixed start destination. This destination is also the last screen the user sees when they return to the launcher after pressing the Back button.



# Navigation Principles

2. Navigation state is represented as a stack of destinations

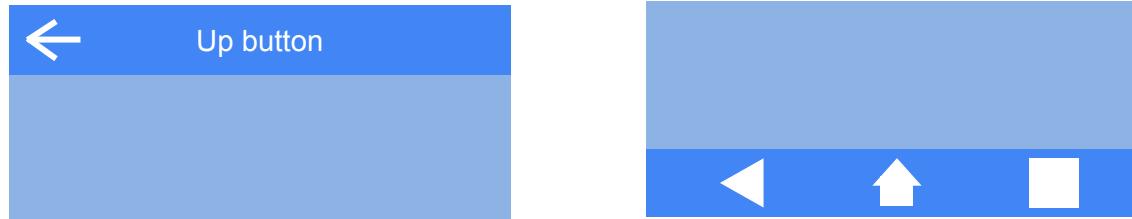
The top of the stack is the current screen, and the previous destinations in the stack represent the history of where you've been.



# Navigation Principles

## 3. Up and Back are identical within your app's task

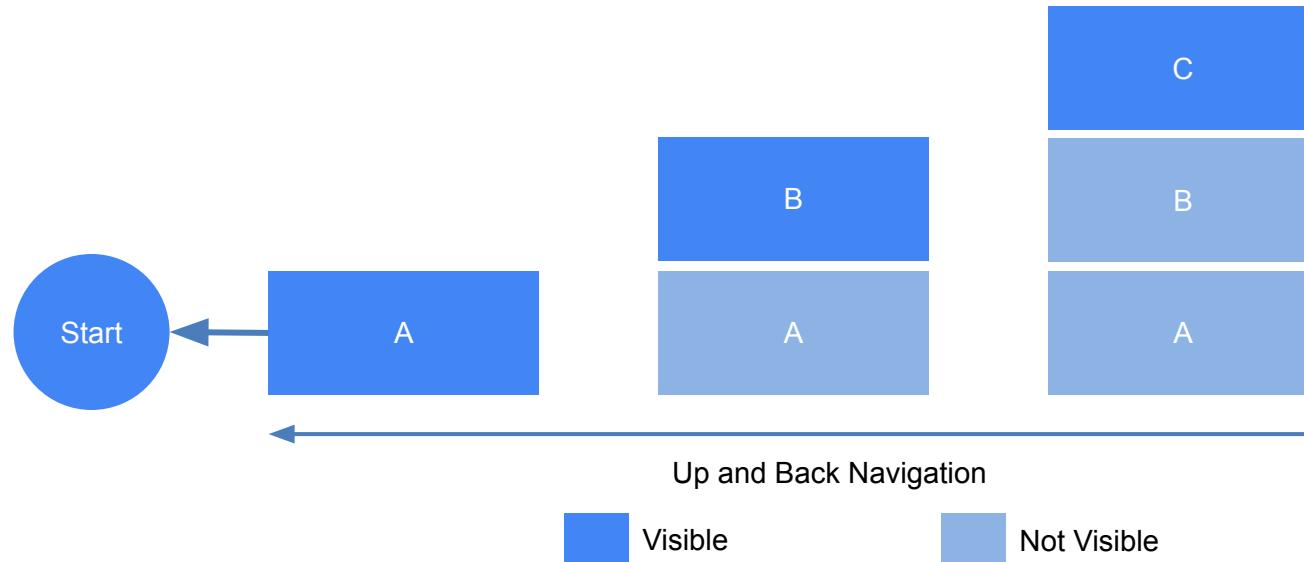
The Up button appears in the app bar at the top of the screen. Within your app's task, the Up and Back buttons behave identically.



# Navigation Principles

## 3. Up and Back are identical within your app's task

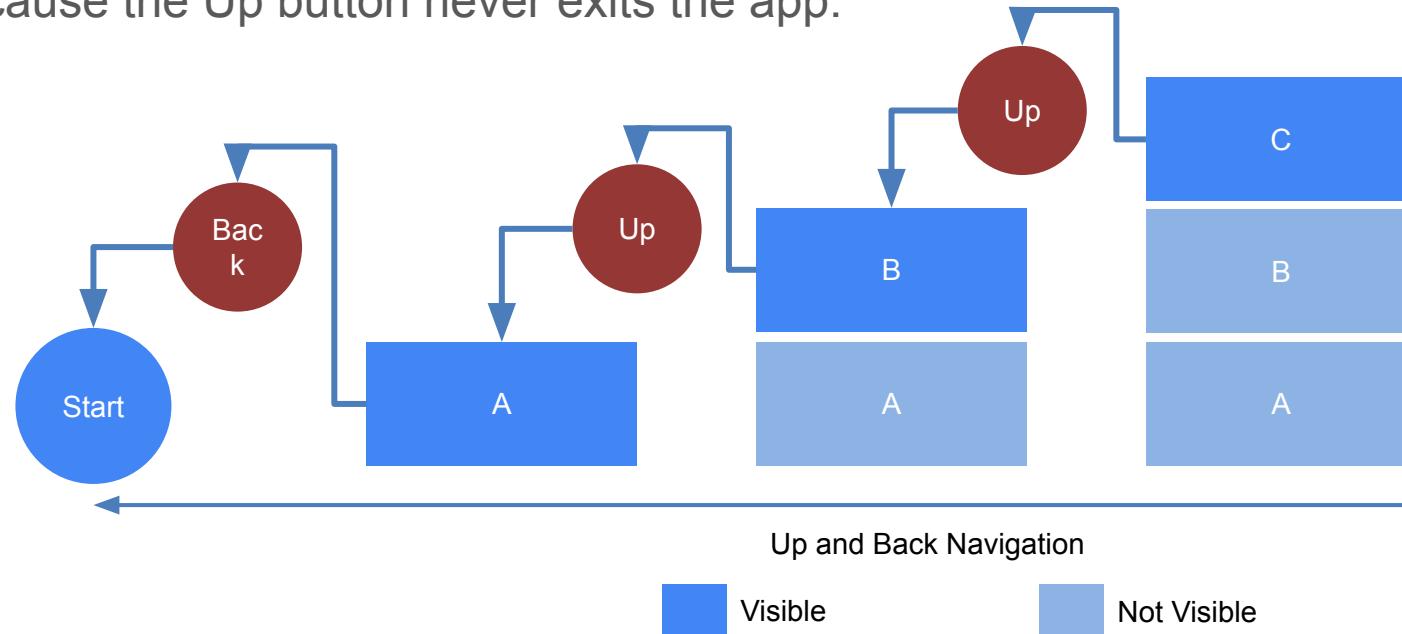
When you press the Back button, the current destination is popped off the top of the back stack, and you then navigate to the previous destination.



# Navigation Principles

## 4. The Up button never exits your app

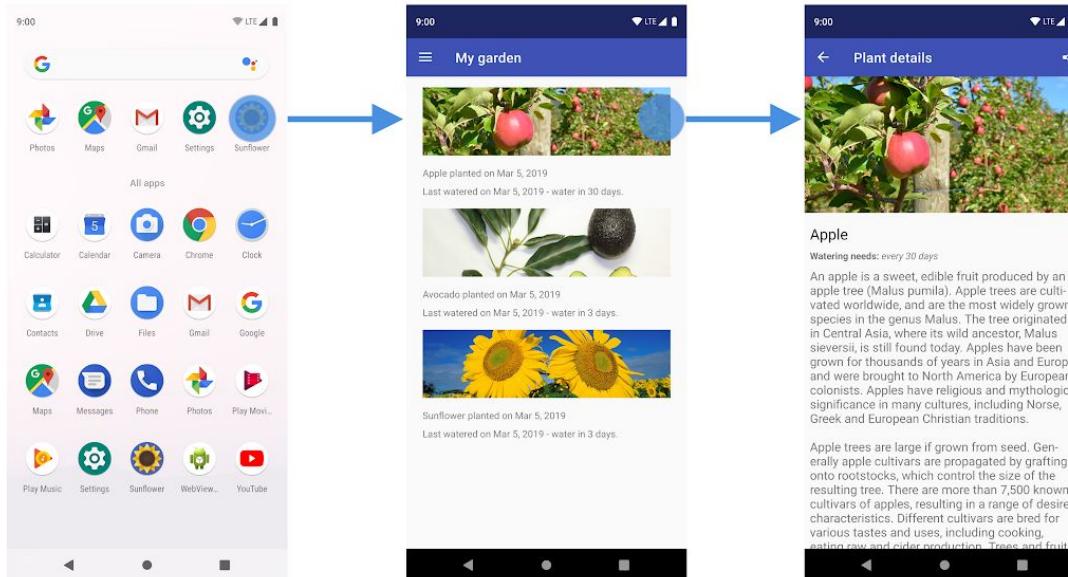
If a user is at the app's start destination, then the Up button does not appear, because the Up button never exits the app.



# Navigation Principles

## 5. Deep linking simulates manual navigation

Navigation design should simulate manual navigation. E.g. From the app navigator to the main screen of an app, and then to a detail screen.



Tab

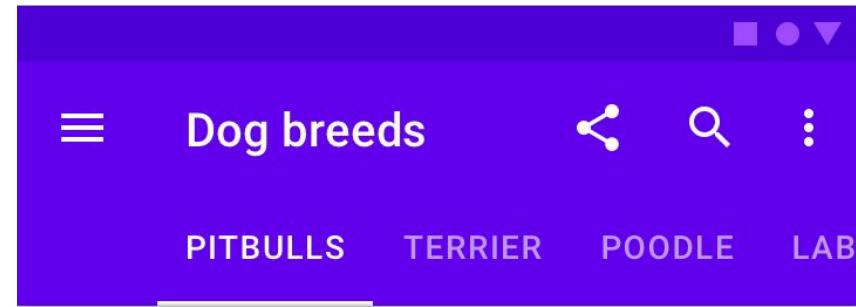
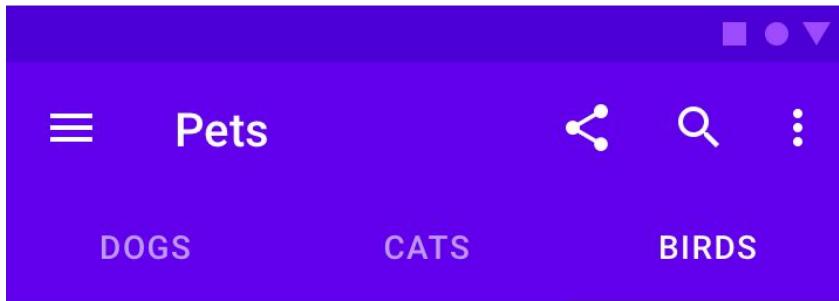
# Tab

A TabBar and TabBarView combination is commonly used to create tabbed interfaces in Flutter.

The TabBar displays the tabs, while the TabBarView displays the content associated with each tab.

The TabController manages the state of the tabs, allowing you to switch between them.

# Tab



1. Fixed Tabs

2. Scrollable Tabs

# Tab

Step 1: Create a TabController

```
return MaterialApp(  
  home: DefaultTabController(  
    length: 3,  
    child: Scaffold(),  
  ),  
);
```

# Tab

Step 2: Create the tabs

```
return MaterialApp(  
  home: DefaultTabController(  
    length: 3,  
    child: Scaffold(  
      appBar: AppBar(  
        bottom: const TabBar(  
          tabs: [  
            Tab(icon: Icon(Icons.directions_car)),  
            Tab(icon: Icon(Icons.directions_transit)),  
            Tab(icon: Icon(Icons.directions_bike)),  
          ],  
        ),  
      ),  
    ),  
  ),  
);
```

# Tab

Step 2: Create content for each tab

```
body: const TabBarView(  
  children: [  
    Icon(Icons.directions_car),  
    Icon(Icons.directions_transit),  
    Icon(Icons.directions_bike),  
  ],  
) ,
```

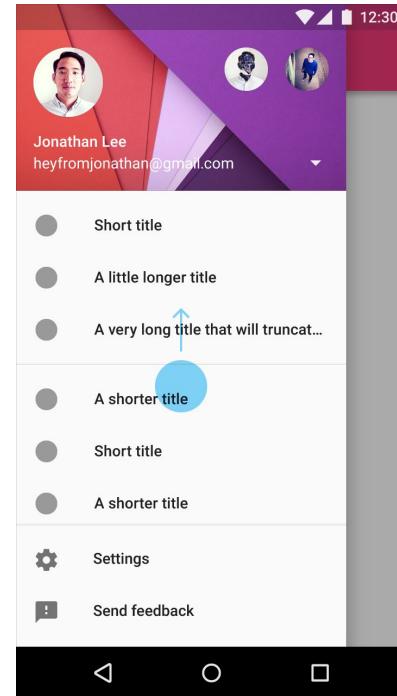
# Drawer

# Drawer

In Flutter, use the Drawer widget in combination with a Scaffold to create a layout with a Material Design drawer.

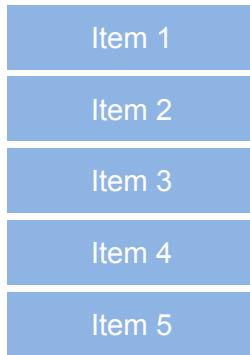
This recipe uses the following steps:

1. Create a Scaffold.
2. Add a drawer.
3. Populate the drawer with items.
4. Close the drawer programmatically.

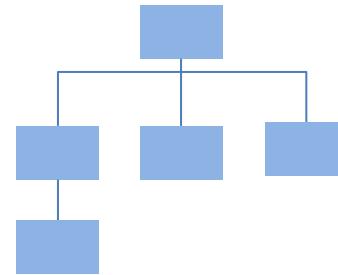


# Navigation Drawer

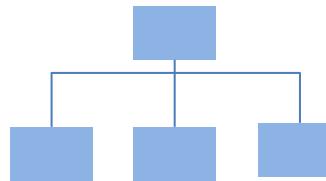
Recommended for:



$\geq 5$  top-level  
destinations



$\geq 2$  levels of  
navigation  
hierarchy



Quick navigation  
between unrelated  
destinations

# Drawer

Step 1: Create a Scaffold

```
Scaffold(  
  appBar: AppBar(  
    title: const Text('AppBar without hamburger button'),  
  ),  
);
```

# Drawer

Step 2: Add a drawer

```
Scaffold(  
  appBar: AppBar(  
    title: const Text('AppBar with hamburger button'),  
  drawer: Drawer(  
    child: // Populate the Drawer in the next step.  
);
```

# Drawer

Step 3: Populate the drawer with items

```
Drawer(  
    // Add a ListView to the drawer.  
    child: ListView(  
        // Important: Remove any padding from the ListView.  
        padding: EdgeInsets.zero,  
        children: [  
            const DrawerHeader(  
                decoration: BoxDecoration(color: Colors.blue,),child: Text('Drawer Header'),),  
            ListTile(title: const Text('Item 1'), onTap: () { ... },),  
            ListTile(title: const Text('Item 2'), onTap: () {... },),  
        ],  
,//End of ListView  
,//End of Drawer
```

# Drawer

Step 4: Open the drawer programmatically.

```
Scaffold(  
  appBar: AppBar(  
    title: const Text('AppBar with hamburger button'),  
    leading: Builder(  
      builder: (context) {  
        return IconButton(  
          icon: const Icon(Icons.menu),  
          onPressed: () { Scaffold.of(context).openDrawer(); },  
        );  
      },  
    ),  
  ),  
  drawer: Drawer(  
    child: // Populate the Drawer in the last step.  
  ),  
);
```

# Drawer

Step 5: Close the drawer programmatically

```
ListTile(  
  title: const Text('Item 1'),  
  onTap: () {  
    // Update the state of the app  
    // ...  
    // Then close the drawer  
    Navigator.pop(context);  
  },  
,
```

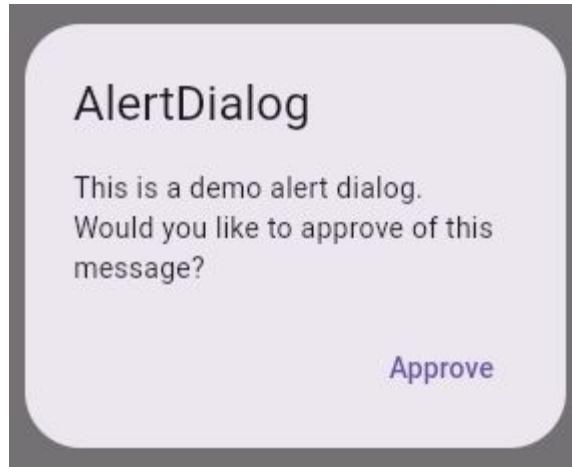
# Dialog

# AlertDialog

An alert dialog (also known as a basic dialog) informs the **user** about situations that require **acknowledgment**.

An alert dialog has an optional title and an optional list of actions.

The title is displayed above the content and the actions are displayed below the content.

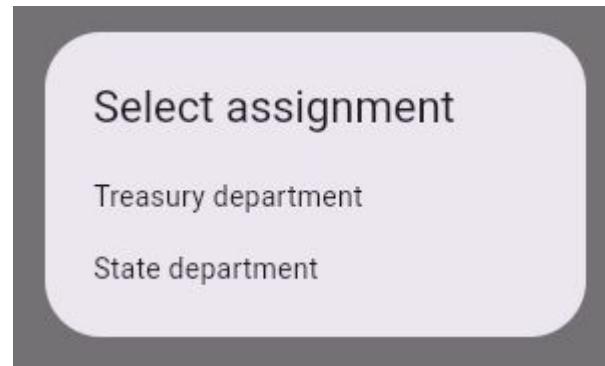


```
Future<void> _showMyDialog() async {
  return showDialog<void>(
    context: context, barrierDismissible: false, // user must tap button!
    builder: (BuildContext context) {
      return AlertDialog(
        title: const Text('AlertDialog Title'),
        content: const SingleChildScrollView(
          child: ListBody(
            children: <Widget>[Text('Would you like to approve of this message?'), ],
          ),
        ),
        actions: <Widget>[
          TextButton( child: const Text('Approve'),
            onPressed: () {
              Navigator.of(context).pop();
            },
          ),
        ],
      ...
    }
}
```

# SimpleDialog

A simple dialog offers the **user** a **choice** between several options.

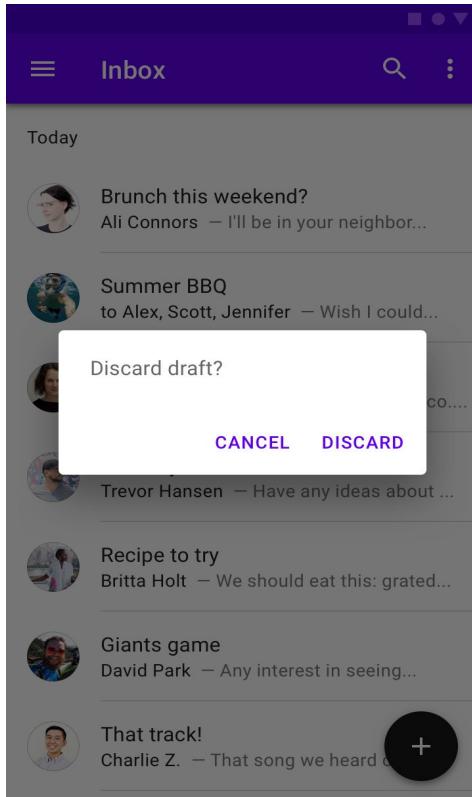
A simple dialog has an optional title that is displayed above the choices.



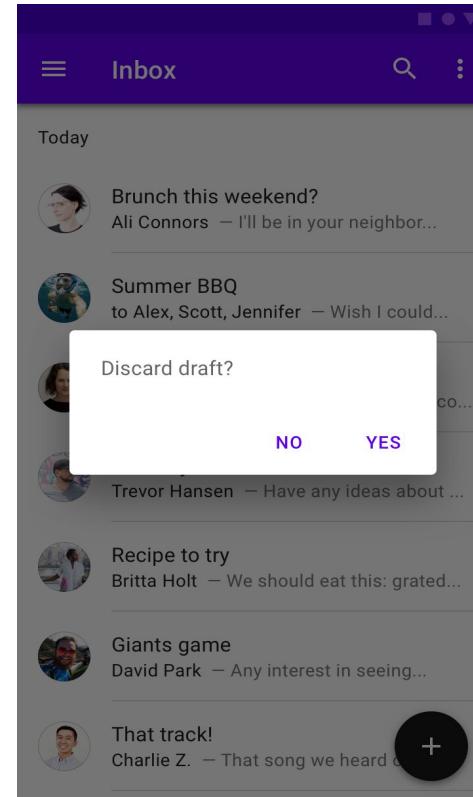
```
Future<void> _askedToLead() async {
  switch (await showDialog<Department>(
    context: context,
    builder: (BuildContext context) {
      return SimpleDialog(
        title: const Text('Select assignment'),
        children: <Widget>[
          SimpleDialogOption(
            onPressed: () { Navigator.pop(context, Department.treasury); },
            child: const Text('Treasury department'),
          ),
          SimpleDialogOption(
            onPressed: () { Navigator.pop(context, Department.state); },
            child: const Text('State department'),
          ),
        ],
      );
    }
  )) {
    case Department.treasury:
      // Let's go.
      // ...
  }
}
```

# Which AlertDialog is better? Why?

(A)



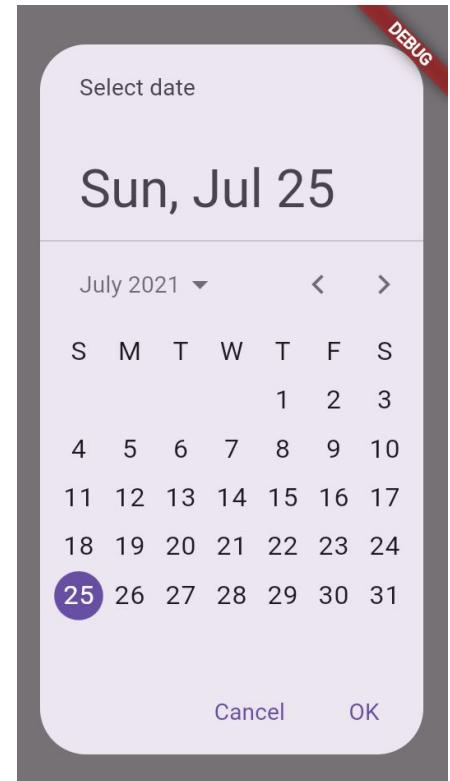
(B)



# Date and Time Pickers

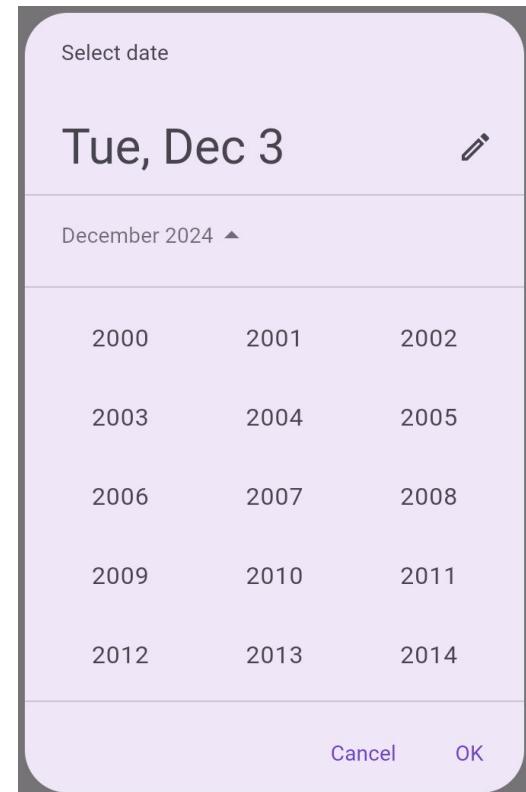
Using `showDatePicker` and `showTimePicker`.

Using these pickers helps ensure that your users can pick a time or date that is valid, formatted correctly, and adjusted to the user's locale.



# Date and Time Pickers

```
DateTime _selectedDate = DateTime.now();  
  
Future<void> _selectDate(BuildContext context) async {  
  final DateTime? picked = await showDatePicker(  
    context: context,  
    initialDate: _selectedDate,  
    firstDate: DateTime(2000),  
    lastDate: DateTime(2100),  
  );  
  
  if (picked != null) {  
    setState(() {  
      _selectedDate = picked;  
    });  
  }  
}
```



# Date and Time Pickers

```
OutlinedButton(  
  onPressed: () {  
    _selectDate(context);  
  },  
  child: Text('$_selectedDate'),  
,
```

2024-12-03 11:21:39.006

# Snackbar

Inform your users when certain actions take place.

For example, a message has been deleted.

Give the user an option to undo an action.

# Snackbar

Insert a snackbar within the Scaffold.

```
final snackBar = SnackBar(  
  content: const Text('Yay! A Snackbar!'),  
  action: SnackBarAction(  
    label: 'Undo',  
    onPressed: () {  
      // Some code to undo the change.  
    },  
  ),  
);  
  
// Find the ScaffoldMessenger in the widget tree and use it to show a SnackBar.  
ScaffoldMessenger.of(context).showSnackBar(snackBar);
```

# Snackbar

Component	Priority	User Action
Snackbar	Low	<p>Optional</p> <p>It disappears automatically</p>
Dialog	High	<p>Required</p> <p>It blocks app usage until the user takes a dialog action or exists the dialog</p>

# Find Out More

Tab, <https://docs.flutter.dev/cookbook/design/tabs>

Drawer, <https://docs.flutter.dev/cookbook/design/drawer>

AlertDialog, <https://api.flutter.dev/flutter/material/AlertDialog-class.html>

SimpleDialog, <https://api.flutter.dev/flutter/material/SimpleDialog-class.html>

Snackbar, <https://docs.flutter.dev/cookbook/design/snackbars>

# Chapter 4.1

## Resources and Data Storage

# Contents

Main vs Background Thread

Saving Data

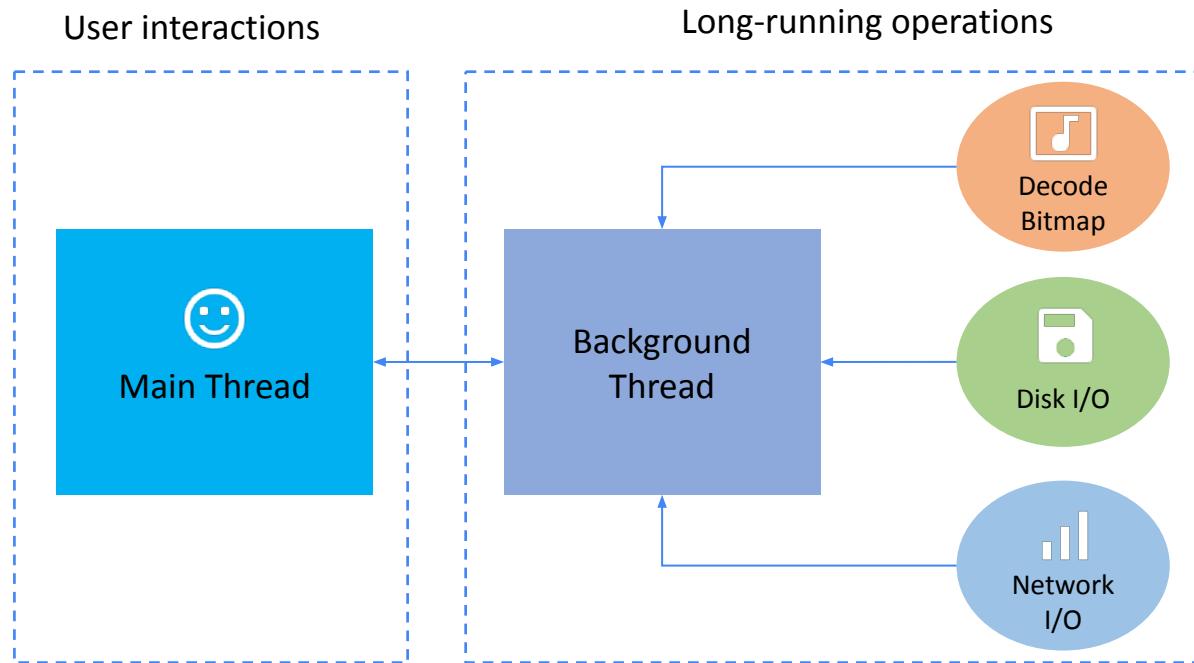
Data File

Shared Preferences

SQLite

Internal vs External Storage

# Main vs Background Thread



# Main Thread

Also called UI Thread, is responsible for rendering the UI and handling user interactions.

All UI updates and user input handling occur on this thread.

Long-running tasks, such as network requests or complex calculations, can block the main thread, leading to UI freezes and poor user experience.

# Background Thread

Used to perform background tasks that don't directly impact the UI.

By offloading tasks to background threads, you ensure the main thread remains responsive.

Common Use Cases: Network requests, database operations, complex calculations, file I/O

# Background Thread

Flutter uses the `Future` class to perform background thread.

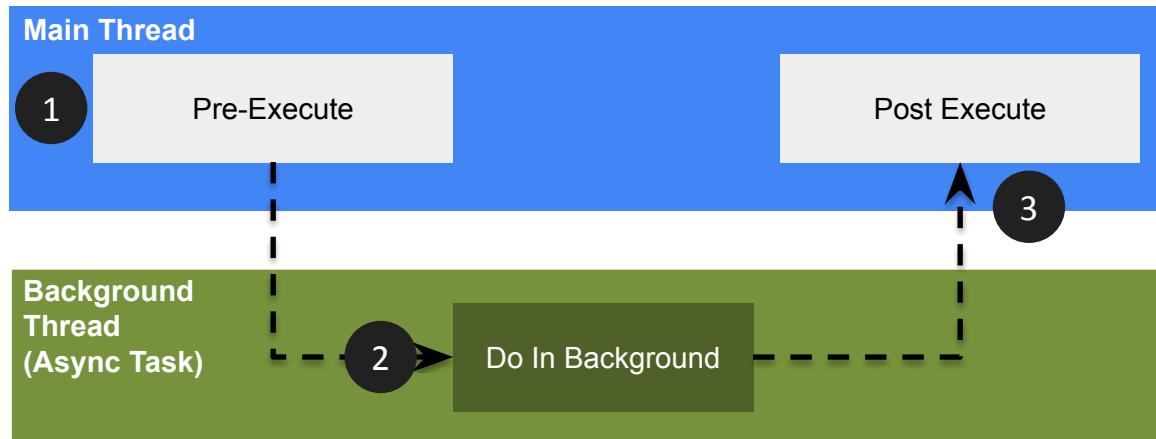
File I/O operations, such as reading and writing files, are inherently asynchronous.

They can take varying amounts of time, and if not handled correctly, they can block the main thread, leading to a frozen UI.

# Background Thread

It performs background operations and publish results on UI thread

Future is a class for working with async operations. A Future object represents a potential value or error that will be available at some time in the future.



# Asynchronous programming: futures, async, await

Asynchronous operations let your program complete work while waiting for another operation to finish.

Here are some common asynchronous operations:

- Fetching data over a network.
- Writing to a database.
- Reading data from a file.

# Asynchronous programming: futures, async, await

Asynchronous tasks produce result as a Future (single item) or Stream (multiple items).

To interact with these asynchronous results, you can use the async and await keywords.

# Future

A Future represents the result of an asynchronous operation, and can have two states:

Uncompleted	Completed
Returns an uncompleted future.	Complete with a value type Future<T>. T could be any valid data type. E.g. String
That Future is waiting for the function's asynchronous operation to finish or to throw an error.	Complete with an error. The future type is Future <void>

# Future

A Future represents the result of an asynchronous operation, and can have two states:

```
void main() {  
    fetchUserOrder();  
    print('Fetching user order...');  
}
```

```
Future<void> fetchUserOrder() {  
    return Future.delayed(const Duration(seconds: 2), () => print('Large Latte'));  
}
```

Output:  
Fetching user order...  
Large Latte

# Async and Await

The async and await keywords provide a declarative way to define asynchronous functions and use their results.

Remember these two basic guidelines when using async and await:

1. Add async before the function body
2. Insert await keyword in async functions.

# Async and Await

```
Future<void> main() async {
  print('Fetching user order...');
  print(await createOrderMessage());
}
```

```
Future<String> createOrderMessage() async {
  var order = await fetchUserOrder();
  return 'Your order is: $order';
}
```

```
Future<String> fetchUserOrder() =>
  Future.delayed(
    const Duration(seconds: 2), () => 'Large Latte',
  );
```

Output:  
Fetching user order...  
Your order is: Large Latte

# Async and Await

```
Future<String> createOrderMessage() async {
  try{
    var order = await fetchUserOrder();
    return 'Your order is: $order';
  }catch(e){
    return 'Failed to obtain order';
  }
}
```

You can write try-catch clauses in synchronous code to handle errors in an async function.

# Stream

Stream returns multiple items. You may use the await for iterates over the events of a stream like a for loop.

```
Future<int> sumStream(Stream<int> stream) async {
  var sum = 0;

  await for (final value in stream) {
    sum += value;
  }

  return sum;
}
```

# Stream

In some cases, an error happens before the stream is done; perhaps the network failed while fetching a file from a remote server.

We may use try-catch to handle errors associated with a stream.

```
Future<int> sumStream(Stream<int> stream) async {
    var sum = 0;

    try {
        await for (final value in stream) {
            sum += value;
        }
    } catch(e){
        return -1;
    }
    return sum;
}
```

## Quiz 4.1.1



# Saving Data

# Saving Data

Flutter, being a cross-platform framework, provides mechanisms to access both internal and external storage.

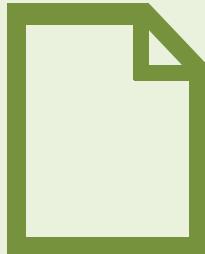
This allows you to store app data locally, ensuring persistence even when the app is offline.

# Saving Data

Data Files

Private or Public

Data file; text,  
sound, images, and  
etc



Shared Preferences

Private or Public

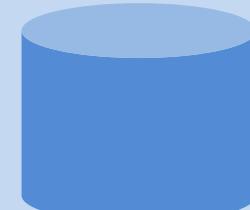
Key-value pair



SQLite

Private

Repeating and  
structured data



# Data File

# Data File

Larger amounts of data, structured data (like JSON), files (like PDF or image) downloaded from the internet, user-generated content.

Use cases:

- Persist data across app launches
- Download data from the internet and save it for later offline use

# Data File

1. Find the correct local path.
2. Create a reference to the file location.
3. Write data to the file.
4. Read data from the file.

# Data File

1: Find the correct local path:

Combine the path\_provider and dart:io libraries.

```
import 'package:path_provider/path_provider.dart';
// ...
Future<String> get _localPath async {
    final directory = await getApplicationDocumentsDirectory();

    return directory.path;
}
```

# Data File

2. Create a reference to the file location:

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

# Data File

## 3. Check storage status:

```
Future<void> checkExternalStorage() async {
    final directory = await getExternalStorageDirectory();

    if (directory != null) {
        print('External storage is available: ${directory.path}');
    } else {
        print('External storage is not available.');
    }
}
```

# Data File

4. Write data to the file:

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

# Shared Preferences

# Shared Preferences

Shared Preferences are suitable for storing small collection of key-values.

Limitations:

- Only primitive types can be used: `int`, `double`, `bool`, `String`, and `List<String>`.
- It's not designed to store large amounts of data.
- There is no guarantee that data will be persisted across app restarts.

# Shared Preferences

1. Add the dependency:

Include the `shared_preferences` package in your `pubspec.yaml` file. Add this line under the dependencies section:

```
shared_preferences: ^2.0.13 # Adjust version if needed
```

2. Import the library:

In your Dart code, import the `shared_preferences` library to access its functionalities.

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

# Shared Preferences

## 3. Saving Data:

Shared Preferences can store various data types including:

Strings (`String`)

Integers (`int`)

Doubles (`double`)

Booleans (`bool`)

# Shared Preferences

Use the appropriate method provided by SharedPreferences to save your data:

- `setString(key, value)`: Saves a string value.
- `setInt(key, value)`: Saves an integer value.
- `setDouble(key, value)`: Saves a double value.
- `setBool(key, value)`: Saves a boolean value.

# Shared Preferences

```
Future<void> saveData(String name, int age) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('name', name);
    await prefs.setInt('age', age);
    print('Data saved successfully!');
}
```

**Important:** Since these methods return futures, it's recommended to use `async` and `await` keywords to ensure data is saved before proceeding.

# Shared Preferences

## 4. Retrieving Data:

To retrieve data, use the corresponding getter methods based on the data type you saved:

- `getString(key)` : Retrieves a string value.
- `getInt(key)` : Retrieves an integer value.
- `getDouble(key)` : Retrieves a double value.
- `getBool(key)` : Retrieves a boolean value.

These methods return the value or null if the key doesn't exist. You can also provide a default value in case the key is missing.

# Shared Preferences

```
Future<String> getName() async {
    final prefs = await SharedPreferences.getInstance();

    // Default if not found
    final name = prefs.getString('name') ?? 'Default Name';
    return name;
}
```

# Shared Preferences

## 5. Removing Data:

Use the `remove(key)` method to remove a specific key-value pair from Shared Preferences.

```
Future<void> removeData(String key) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.remove(key);
    print('Data with key $key removed');
}
```

# SQLite

# SQLite

SQLite is a lightweight, embedded SQL database engine that's perfect for mobile applications.



# SQLite

It provides faster inserts, updates, and queries for large amounts of data on the local storage.

Flutter offers a plugin called `sqflite` to interact with SQLite databases.

The `sqflite` package only supports apps that run on macOS, iOS and Android.

# SQLite

Step 1: Add the dependency to your pubspec.yaml file:

dependencies:

```
sqflite: ^2.3.3+1
```

```
path: ^1.9.0
```

Or run

```
$ flutter pub add sqflite path
```

# SQLite

Step 2: Import packages:

```
import 'dart:async';  
  
import 'package:sqflite/sqflite.dart';
```

# SQLite

Step 3: Define a data model:

```
class Note {  
    final int id;  
    final String title;  
    final String description;  
  
    const Note ({  
        required this.id,  
        required this.title,  
        required this.description,  
    });  
}
```

```
// Convert a Note into a Map.  
Map<String, Object?> toMap() {  
    return {  
        'id': id,  
        'title': title,  
        'description': description,  
    };  
}  
  
// Implement toString  
@override  
String toString() {  
    return 'Note{id: $id, title: $title,  
              description: $description}';  
}
```

# SQLite

Step 4: Open database:

```
// Avoid errors caused by flutter upgrade.  
WidgetsFlutterBinding.ensureInitialized();  
  
// Open the database and store the reference.  
final database = openDatabase(  
    // Set the path to the database.  
    join(await getDatabasesPath(), 'note.db'),  
);
```

# SQLite

Step 5: Create table:

```
final database = openDatabase(  
    // Set the path to the database.  
    join(await getDatabasesPath(), 'note.db'),  
  
    // When the database is first created, create a table to store notes.  
    onCreate: (db, version) {  
        // Run the CREATE TABLE statement on the database.  
        return db.execute(  
            'CREATE TABLE notes(id INTEGER PRIMARY KEY, title TEXT, description TEXT)',  
            );  
    },  
    // Set the version.  
    version: 1,  
);
```

# SQLite

## Step 6: Insert

```
Future<void> insertNote(Note note) async {
    // Get a reference to the database.
    final db = await database;

    // Insert the Note into the correct table.
    await db.insert(
        'notes',
        note.toMap(),
        conflictAlgorithm: ConflictAlgorithm.replace,
    );
}
```

```
var note1 = Note(
    id: 0,
    title: 'Note 1',
    description: 'Desc 1',
);

await insertNote(note1);
```

# SQLite

## Step 6: Retrieve notes

```
Future<List<Note>> notes() async {
    final db = await database; // Get a reference to the database.

    // Query the table for all the dogs.
    final List<Map<String, Object?>> noteMaps = await db.query('notes');

    // Convert each note's fields into a list of Note
    return [
        for (final {'id': id as int, 'title': title as String,
                   'description': description as String,
                   } in noteMaps)
            Note(id: id, title: title, description: description),
    ];
}
...
print(await notes());
```

# SQLite

## Step 6: Retrieve notes

```
Future<List<Note>> notes() async {
    final db = await database; // Get a reference to the database.

    // Query the table for all the dogs.
    final List<Map<String, Object?>> noteMaps = await db.query('notes');

    // Convert each note's fields into a list of Note
    return [
        for (final {'id': id as int, 'title': title as String,
                    'description': description as String,
                    } in noteMaps)
            Note(id: id, title: title, description: description),
    ];
}
...
print(await notes());
```

# SQLite

## Step 7: Update notes

```
Future<void> updateNote(Note note) async {
    // Get a reference to the database.
    final db = await database;

    // Update the given Note.
    await db.update(
        'notes',
        note.toMap(),
        // Ensure that the Note has a matching id.
        where: 'id = ?',
        // Pass the Note's id as a whereArg to prevent SQL injection.
        whereArgs: [note.id],
    );
}

...
await updateNote(newNote);
```

# SQLite

## Step 8: Delete notes

```
Future<void> deleteNote(int id) async {
    // Get a reference to the database.
    final db = await database;

    // Remove the Note from the database.
    await db.delete(
        'notes',
        // Use a `where` clause to delete a specific note.
        where: 'id = ?',
        // Pass the Note's id as a whereArg to prevent SQL injection.
        whereArgs: [id],
    );
}

...
await deleteNote(newNote.id);
```

## Quiz 4.1.2

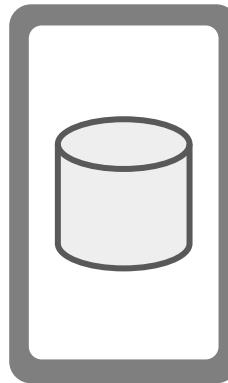


# Internal vs External Storage

# Internal Storage

Storage that is directly accessible to your app.

Typically used for storing app data, preferences, and cache.



# Internal Storage

An app's internal storage directory is specified by the app's package name

No permission is required to perform read/write operations

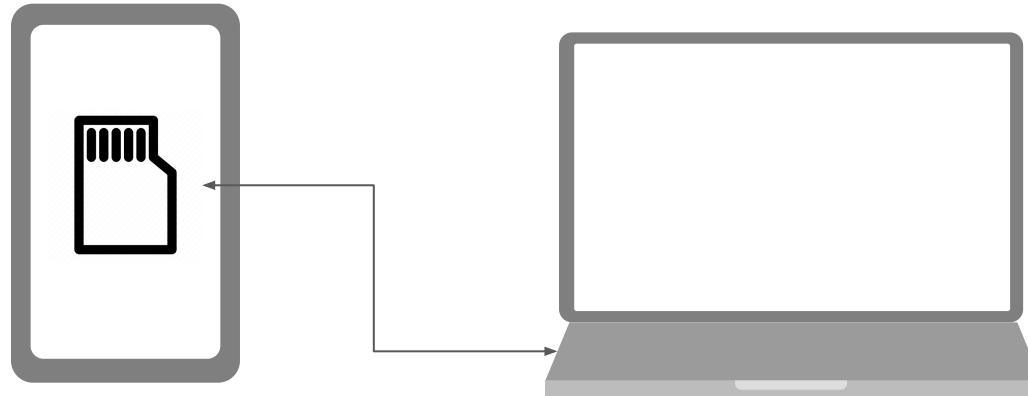
Suitable for:

- Shared Preferences
- SQLite Database

# External Storage

Storage that is accessible to multiple apps on the device and outside of the device.

Typically used for storing large files or media.



# External Storage

Feature	Android	iOS
Expandable Storage	Yes	No (generally)
Removable	Possible (SD Card)	No
File System Access	More flexible	More restricted
Permissions	Requires explicit user permission	More controlled by the system
Data Sharing	Easier to share files between apps	More limited sharing options

# External Storage

1. **Permissions:** Ensure that your app has the necessary permissions to access storage.
2. **Data Security:** Consider encrypting sensitive data before storing it.
3. **Performance:** Optimize file I/O operations to avoid performance bottlenecks.
4. **User Experience:** Provide clear feedback to the user during file operations, such as progress indicators.

## Quiz 4.1.3



## Find out more

Data File, <https://docs.flutter.dev/cookbook/persistence/reading-writing-files>

Key-Value pair, <https://docs.flutter.dev/cookbook/persistence/key-value>

SQLite, <https://docs.flutter.dev/cookbook/persistence/sqlite>

# Chapter 4.2

Network Operations

# Contents

Mobile-to-server communication

Server Options

Networking

Network Resources

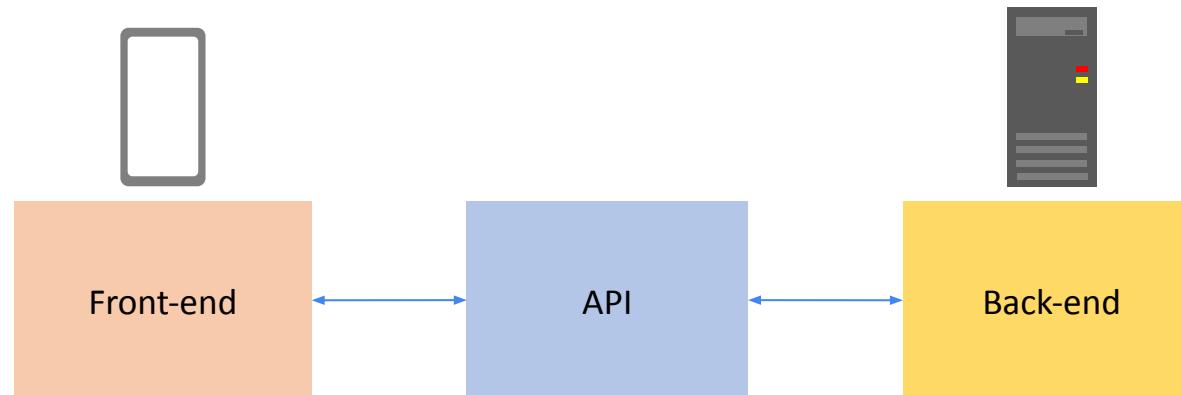
Message Queuing Telemetry Transport

# Mobile-to-server communication

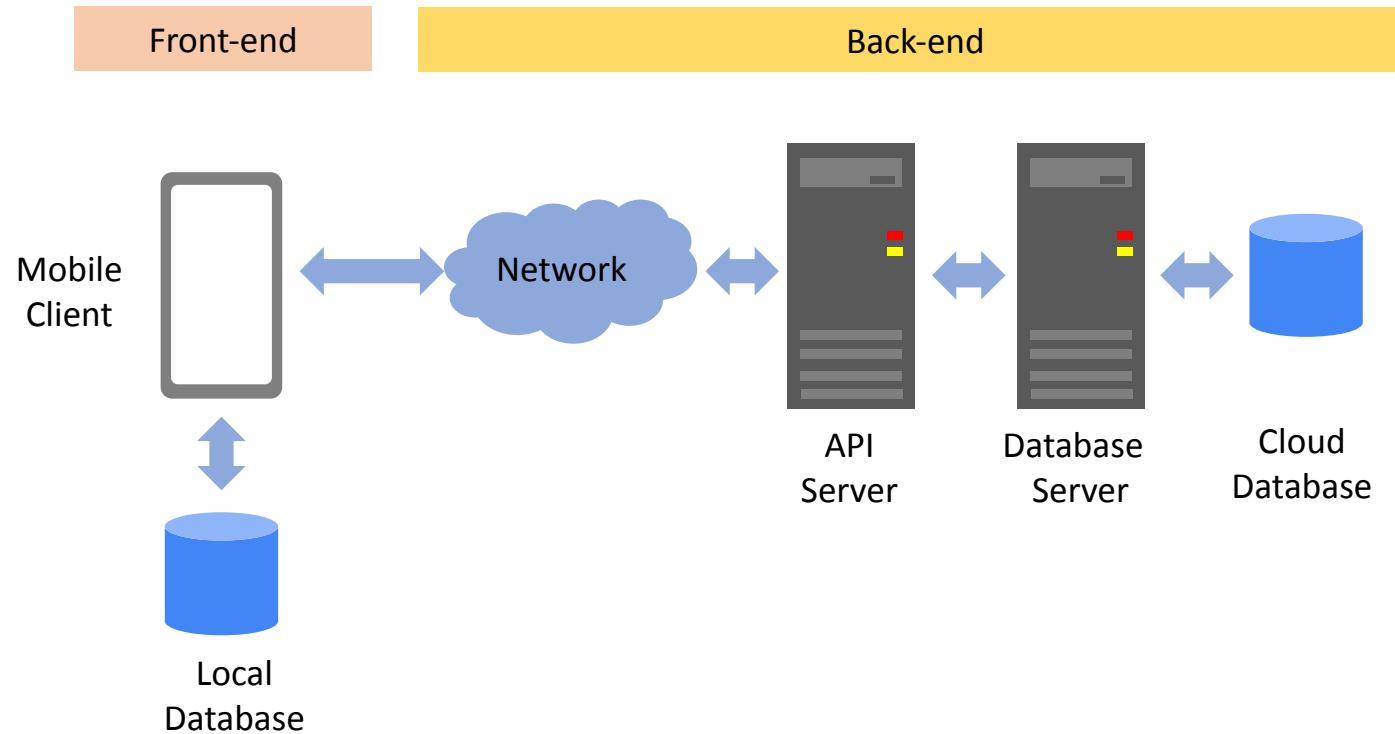
# Mobile-to-server Communication

Most mobile apps are the front-end interfaces of back-end services

The two components use Application Programming Interface (API) to communicate with each other



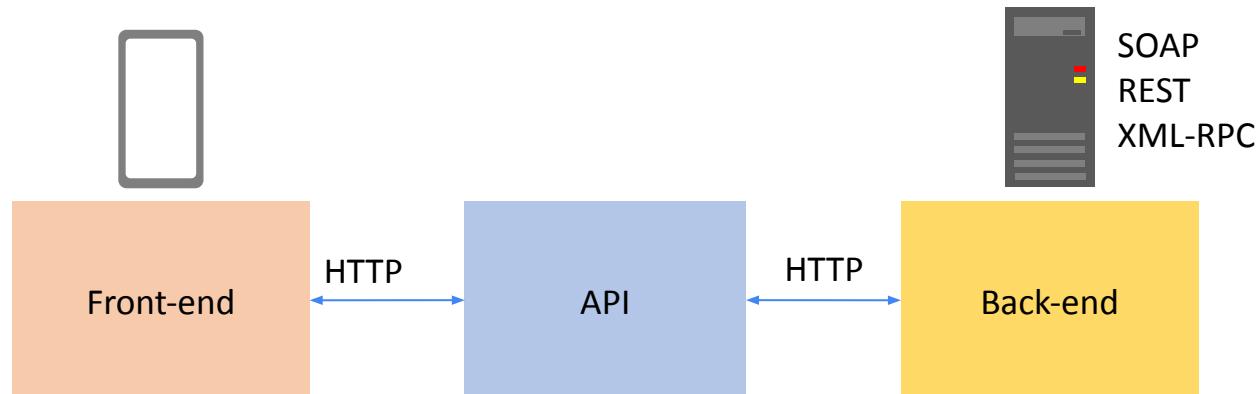
# Mobile System Architecture



# Communication Protocol

HTTP is the most commonly used protocol for communication

Methods of communication: SOAP, REST, and XML-RPC



# Machine-to-machine Communication

## Representational State Transfer ([REST](#))

- Supports data formats: HTML, XML and JavaScript Object Notation ([JSON](#))
- Inherits HTTP operations; GET, POST, PUT and DELETE

<https://www.restapitutorial.com/lessons/whatisrest.html>

# Machine-to-machine Communication

Simple Object Access Protocol ([SOAP](#))

- Supports data formats: XML
- Works with application layer protocol. E.g. HTTP, SMTP, TCP, or UDP

# REST data formats

## XML

```
<?xml version="1.0" encoding="UTF-8"?>
<authentication-context>
    <username>my_username</username>
    <password>my_password</password>
    <validation-factors>
        <validation-factor>
            <name>remote_address</name>
            <value>127.0.0.1</value>
        </validation-factor>
    </validation-factors>
</authentication-context>
```

## JSON

```
{
    "username" : "my_username",
    "password" : "my_password",
    "validation-factors" : {
        "validationFactors" : [ {
            "name" : "remote_address",
            "value" : "127.0.0.1"
        } ]
    }
}
```

# SOAP data formats

```
<env:Envelope  
    xmlns:env="http://www.w3.org/2003/05/soap-envelope">  
    <env:Header>  
        <n:alertcontrol xmlns:n="http://example.org/alertcontrol">  
            <n:priority>1</n:priority>  
            <n:expires>2001-06-22T14:00:00-05:00</n:expires>  
        </n:alertcontrol>  
    </env:Header>  
    <env:Body>  
        <m:alert xmlns:m="http://example.org/alert">  
            <m:msg>Pick up Mary at school at 2pm</m:msg>  
        </m:alert>  
    </env:Body>  
</env:Envelope>
```

# Server Options

# Server Options

DIY

Build  
your own  
servers



Subscribe

Back-End  
as a  
Service  
(BaaS)



Mix and bang

DIY +  
Subscribe



# Server Options

Factors to Consider:

1. Project Scope: How complex is your app? How much data will it handle?
2. Budget: What is your budget for development and ongoing maintenance?
3. Team Expertise: Do you have in-house expertise in server-side development?
4. Scalability Requirements: How much growth do you anticipate for your app?
5. Time-to-Market: How quickly do you need to launch your app?

# Server Options - DIY (Do It Yourself)

## Advantages:



You have complete control over the technology stack, data, and infrastructure.

It can become cost-effective in the long run if you manage it well.

You earn valuable experience in server-side development.

# Server Options - DIY (Do It Yourself)



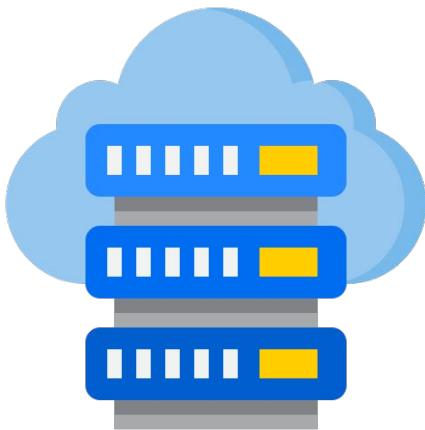
## Disadvantages:

High initial investment for setup, maintenance, and scaling.

Expertise required in server administration, networking, security, and database management.

Requires ongoing maintenance and troubleshooting.

# Server Options - Subscribe



## Advantages:

No need to manage servers, databases, or infrastructure.  
Gain access to pre-build features such as AI model, NLP etc

Providers handle scaling and performance automatically.  
Allows you to focus on building your app's frontend and core functionality.

Fast and cost-effective deployment - you can introduce your services to the public quickly.

# Server Options - Subscribe



## Disadvantages:

Vendor Lock-in. It can be challenging to switch providers later.

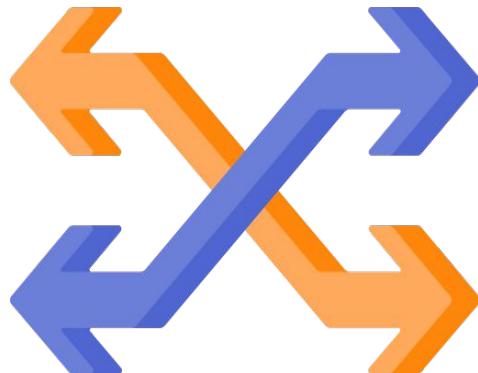
May have limitations on customization and flexibility.

Costs can increase as your app grows and usage increases.

# Server Options - Mix and bang

## Advantages:

It is flexible. Combines the benefits of DIY and BaaS.

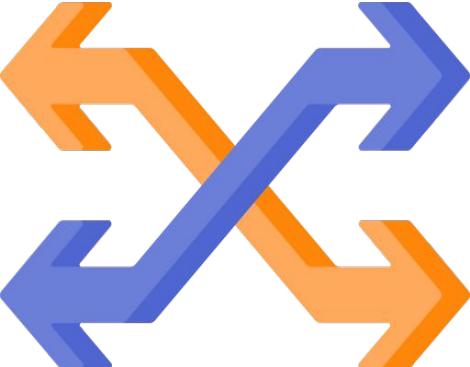


Customizable: Allows you to choose and manage specific components while relying on BaaS for others.

Improved Scalability: Can scale specific components as needed.

# Server Options - Mix and bang

## Disadvantages:



Increased Complexity: Requires careful planning and coordination.

Requires a deeper understanding of both approaches.

# Networking

# Networking

Flutter uses a cross-platform http networking

The `http` package provides the simplest way to issue http requests.

This package is supported on Android, iOS, macOS, Windows, Linux and the web.

# Networking

## Android

Android apps must declare their use of the internet in the Android manifest (AndroidManifest.xml):

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

# Networking

## macOS

macOS apps must allow network access in the relevant \*.entitlements files.

```
<key>com.apple.security.network.client</key>
<true/>
```

# Network Resources

# Getting file from the network

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

```
// Getting a file from the network
```

```
class NetworkImageExample extends StatelessWidget {  
    final String imageUrl = 'https://example.com/image.jpg';
```

```
@override
```

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

```
    return FutureBuilder<http.Response>(  
        future: http.get(Uri.parse(imageUrl)),
```

```
        builder: (context, snapshot) {  
            if (snapshot.hasData) {  
                final bytes = snapshot.data!.bodyBytes;
```

```
                return Image.memory(bytes);  
            } else if (snapshot.hasError) {  
                return Text('Error loading image');  
            } else {  
                return const CircularProgressIndicator();  
            }
```

```
        },  
    );  
}
```

```
}
```

# Network Operations

# Network Operations

Flutter provides various methods to perform network operations, primarily through the `http` package.

This package allows you to make HTTP requests to interact with web APIs, such as:

- GET
- POST
- PUT
- DELETE

# Network Operations

## GET

Retrieves data from a specified resource.

Typically used for fetching information, such as news feeds, product listings, or user profiles.

```
http.get(Uri.parse('https://api.example.com/users'))
  .then((response) {
    // Handle the response
});
```

# Network Operations

## POST

Sends data to a server to create a new resource.

Often used for submitting forms, uploading files, or creating new records in a database.

```
http.post(Uri.parse('https://api.example.com/users'), body: {  
  'name': 'John Doe',  
  'email': 'johndoe@example.com'  
}).then((response) {  
  // Handle the response  
});
```

# Network Operations

## PUT

Updates an existing resource on the server.

Typically used to modify existing data or replace an entire resource.

```
http.put(Uri.parse('https://api.example.com/users/123'), body: {  
  'name': 'Jane Doe'  
}).then((response) {  
  // Handle the response  
});
```

# Network Operations

## DELETE

Removes a resource from the server.

Used to delete existing records or resources.

```
http.delete(Uri.parse('https://api.example.com/users/123'))
  .then((response) {
// Handle the response
});
```

# Fetch data from the internet

This recipe uses the following steps:

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

# Fetch data from the internet

## Step 1: Add the http package

The http package provides the simplest way to fetch data from the internet.

To add the http package as a dependency, run

```
$ flutter pub add
```

Or

Add the http package to your pubspec.yaml file:

```
dependencies:
```

```
  http: ^0.13.5
```

# Fetch data from the internet

## Step 1: Add the http package

Import the http package in your code:

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

# Fetch data from the internet

## Step 1: Add the http package

Android

Edit your AndroidManifest.xml file to add the Internet permission.

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

# Fetch data from the internet

## Step 1: Add the http package

iOS

Edit your macos/Runner/DebugProfile.entitlements and  
macos/Runner/Release.entitlements files to include the network client entitlement.

```
<key>com.apple.security.network.client</key>
<true/>
```

# Fetch data from the internet

## Step 2: Make a network request using the http package

Use the `http.get()` method to retrieve data from a server

```
Future<http.Response> fetchAlbum() {  
    return http.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));  
}
```

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

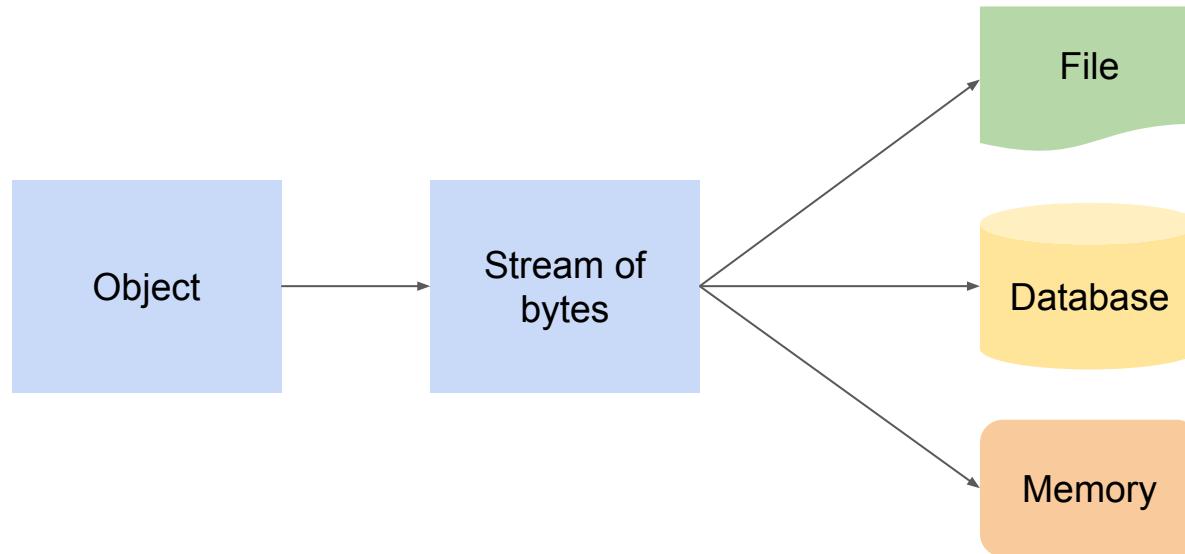
# JSON

JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays.

It is a commonly used data format with diverse uses in electronic data interchange, including that of web and mobile applications with servers.

# Serialization

Serialization refers to the process of translating data structures to and from a more easily readable format.



# JSON Serialization

Flutter offers two general strategies for working with JSON:

1. Manual serialization
2. Automated serialization using code generation

# Manual Serialization

It uses the built-in JSON decoder in `dart:convert`.

It involves passing the raw JSON string to the `jsonDecode()` function, and then looking up the values you need in the resulting `Map<String, dynamic>`.

# Manual Serialization

Advantages:

- It has no external dependencies or particular setup process
- It is good for a quick proof of concept.

# Manual Serialization

## Disadvantages:

- It does not perform well when your project becomes bigger.
- Writing decoding logic by hand can become hard to manage and error-prone.
- If you have a typo when accessing a nonexistent JSON field, your code throws an error during runtime.

# Code generation

Uses an external library generate the encoding boilerplate. E.g.  
`json_serializable` and `built_value` libraries.

Advantages:

- It scales well for a larger project.
- No hand-written boilerplate is needed, and typos when accessing JSON fields are caught at compile-time.

# Code generation

Disadvantages:

- It requires some initial setup
- The generated source files might produce visual clutter in your project navigator.

# Serializing JSON inline

```
{  
  "name": "John Smith",  
  "email": "john@example.com"  
}
```

---

```
final user = jsonDecode(jsonString) as Map<String, dynamic>;  
  
print('Howdy, ${user['name']}!');  
print('We sent the verification link to ${user['email']}.');
```

# Serializing JSON inside model classes

With this approach, the calling code can have type safety, autocompletion for the name and email fields, and compile-time exceptions.

```
class User {  
    final String name;  
    final String email;  
  
    User(this.name, this.email);  
  
    User.fromJson(Map<String, dynamic> json)  
        : name = json['name'] as String,  
          email = json['email'] as String;  
  
    Map<String, dynamic> toJson() => {  
        'name': name,  
        'email': email,  
    };  
}
```

# Serializing JSON inside model classes

The responsibility of the decoding logic is now moved inside the model itself. With this new approach, you can decode a user easily.

```
final userMap = jsonDecode(jsonString) as Map<String, dynamic>;
final user = User.fromJson(userMap);

print('Howdy, ${user.name}!');
print('We sent the verification link to ${user.email}.');
```

# Serializing JSON inside model classes

To encode a user, pass the `User` object to the `jsonEncode()` function. You don't need to call the `toJson()` method, since `jsonEncode()` already does it for you.

```
String json = jsonEncode(user);
```

# Message Queuing Telemetry Transport

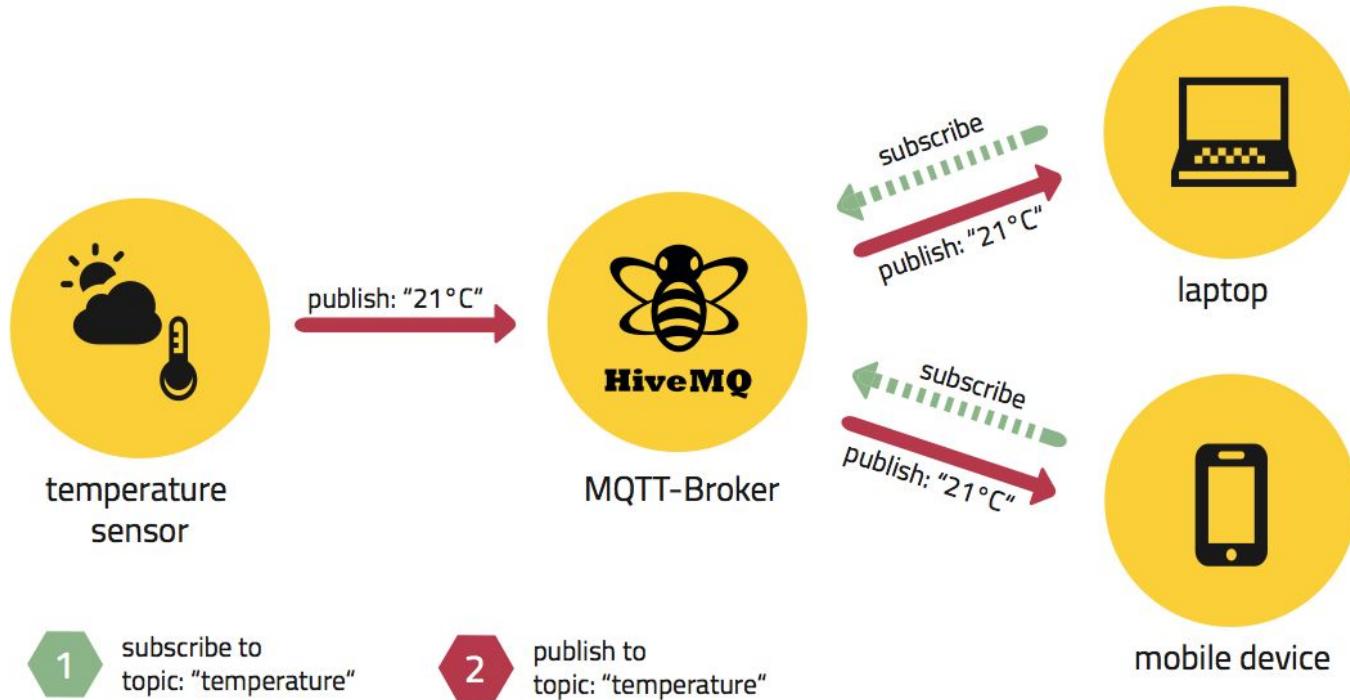
Message Queuing Telemetry Transport ([MQTT](#))

Works on top of TCP/IP

Machine-to-machine connectivity protocol

Uses publish/subscribe messaging transport

# Internet of Things Real-time messaging



# Reference

Networking, <https://docs.flutter.dev/data-and-backend/networking>

JSON and serialization,

<https://docs.flutter.dev/data-and-backend/serialization/json>

Fetch data from the internet,

<https://docs.flutter.dev/cookbook/networking/fetch-data>

## Find out more

Introduction to MQTT, <https://github.com/mqtt/mqtt.github.io/wiki>

MQTT Client, [https://pub.dev/packages/mqtt\\_client](https://pub.dev/packages/mqtt_client)

# Chapter 5

Location-based Services

# Contents

Location Services

Location Strategies

Google Play Services

Location best practices

# Introduction

Mobile users take their devices with them everywhere

Location awareness can be used to offer contextual experience

Location-based services are integral to many modern mobile apps.

To ensure accurate, timely, and efficient location tracking, developers must carefully consider the appropriate strategies.

# Source of location data

Two ways to obtain a user location: GPS and Network-based

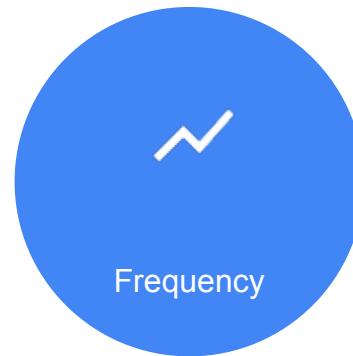
Type	GPS	Network-based
Data Accuracy	High	Low
Speed	Slow	Fast
Power Consumption	High	Low
Environment	Outdoor	Indoor and outdoor

# Battery Drain

Location gathering and battery drain are affected by these aspects:



Accuracy

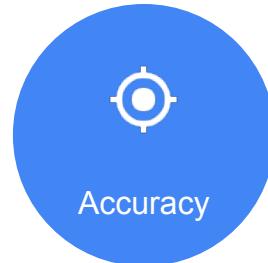


Frequency



Latency

# Battery Drain

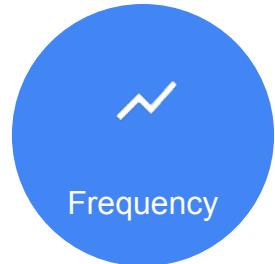


## Accuracy

Refers to the precision of the location data.

Higher accuracy often requires more power and resources.

# Battery Drain

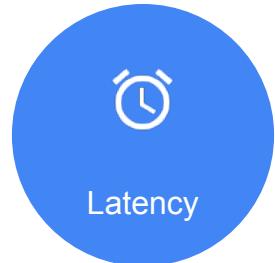


## Frequency

Determines how often the device's location is updated.

A higher frequency can drain the battery faster.

# Battery Drain



## Latency

Measures the delay between the actual location change and the time it takes for the app to receive the updated location.

# Types of accuracy

Type	Precision	Hardware Use	Power
High Accuracy	Most precise location possible	GPS	High
Balanced Power Priority	City block (100 m)	Wi-Fi or cell tower	Less
Low Power	City-level (10 km)	Wi-Fi or cell tower	Less
No power	Receives locations from other apps	None	Very minimum

# Location Strategies

## Global Positioning System (GPS)

High Accuracy: Provides precise location data.

Power Consumption: Can drain the battery quickly, especially when used continuously.

Use Cases: Navigation apps, outdoor activity tracking, geofencing.

# Location Strategies

## Network-Based Location

Lower Accuracy: Relies on cell tower and Wi-Fi triangulation.

Lower Power Consumption: Less demanding on the device's battery.

Use Cases: General location-based services, weather apps, local search.

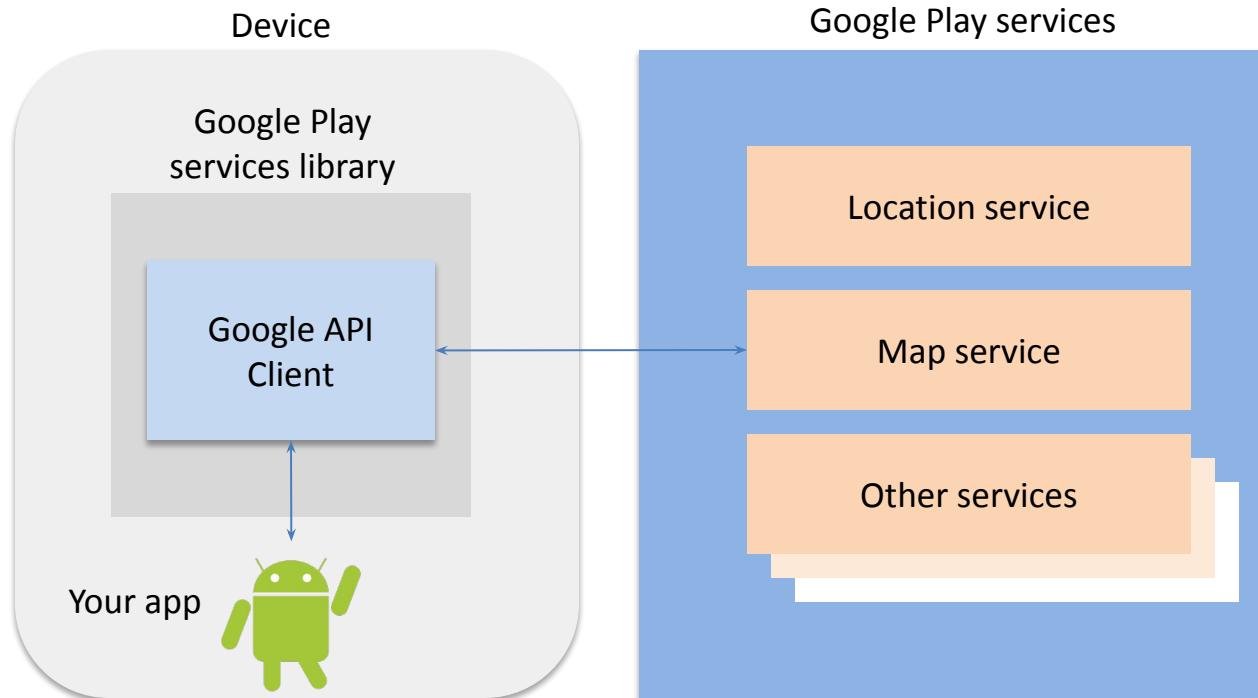
# Location Strategies

## Hybrid Approach:

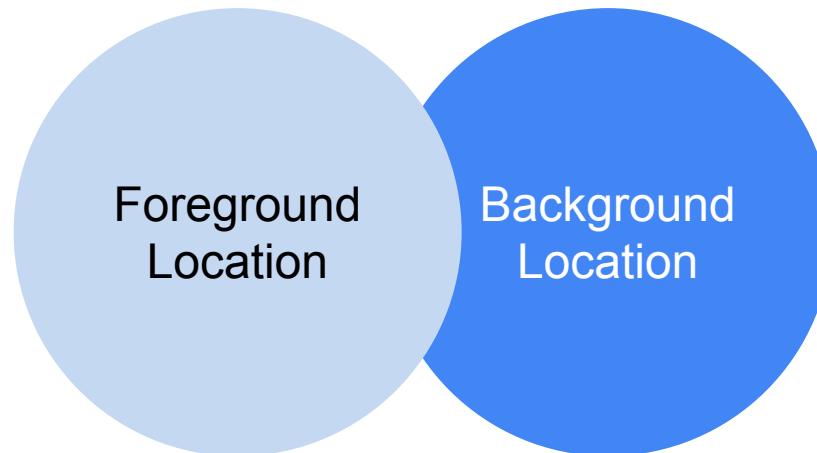
Balanced Accuracy and Power Consumption: Combines GPS and network-based location.

Use Cases: Most mobile apps that require location data.

# Google Play Service



# Types of Location Permission



# Types of Location Permission

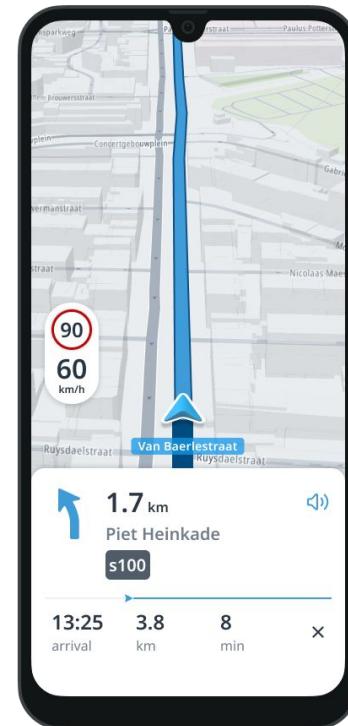
Task	Foreground Location	Background Location
Share Location	Once Predefined period	Constant
Visibility of UI	Visible	Not visible
Show persistent notification	Yes	No

# Foreground Location

## Examples

Within a navigation app, a feature allows users to get turn-by-turn directions.

Within a messaging app, a feature allows users to share their current location with another user.



# Background Location

## Examples

Within a family location sharing app, a feature allows users to continuously share location with family members.

Within an IoT app, a feature allows users to configure their home devices such that they turn off when the user leaves their home and turn back on when the user returns home.



# Google Play Service

Insert permission to manifest file:

Permission	Description
Foreground - Coarse Location	Allows an app to access approximate location. Returns a location with an accuracy approximately equivalent to a city block.
Foreground - Fine Location	Allows an app to access precise location.
Background	For Android 10 (API 29) and above

# Google Play Service

In the context of Android, in the Manifest file:

```
<manifest ... >
    <!-- Always include this permission -->
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

    <!-- Include only if your app benefits from precise location access. -->
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <!-- Required only when requesting background Location access on
        Android 10 (API Level 29) and higher. -->
    <uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />

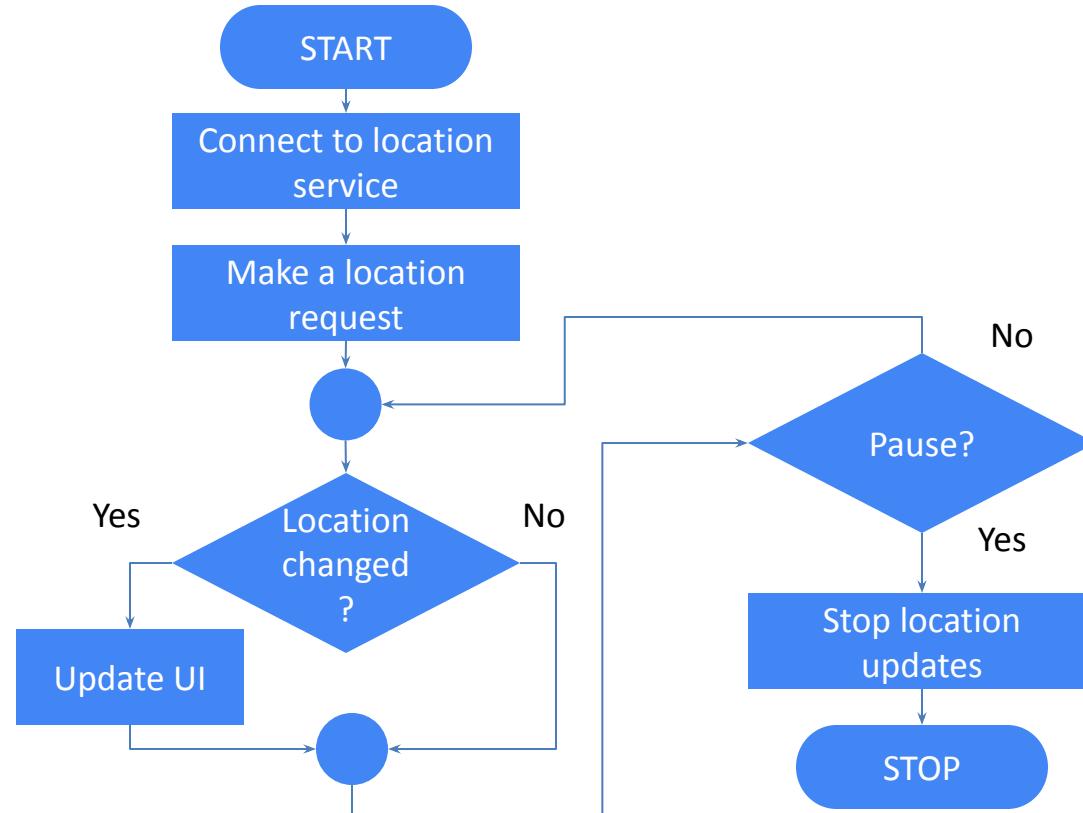
</manifest>
```

# Google Play Service

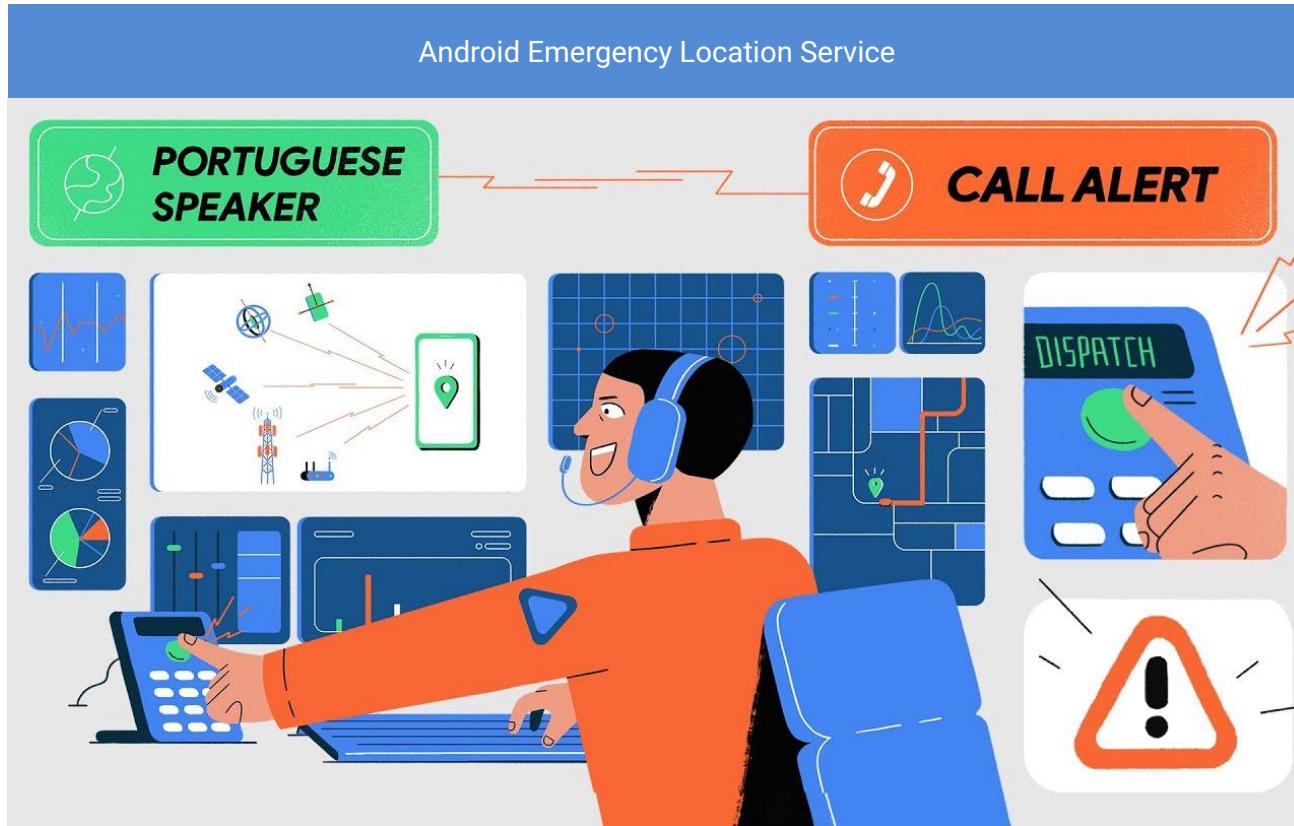
For foreground service type, insert this in the Manifest file within the application tag:

```
<!-- Recommended for Android 9 (API Level 28) and Lower. -->
<!-- Required for Android 10 (API Level 29) and higher. -->
<service
    android:name="MyNavigationService"
    android:foregroundServiceType="location">
    <!-- Any inner elements would go here. -->
</service>
```

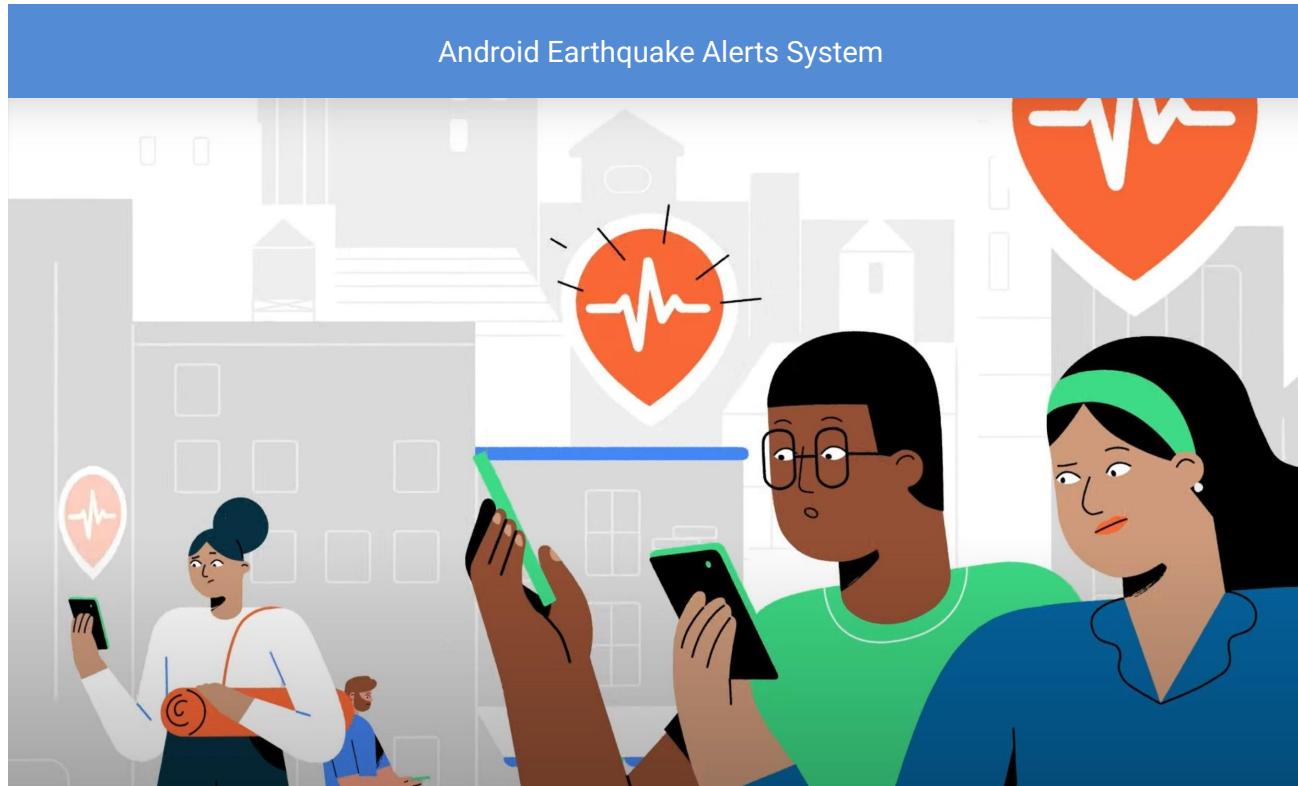
# Location Updates



# New Android Location Service



# New Android Location Service

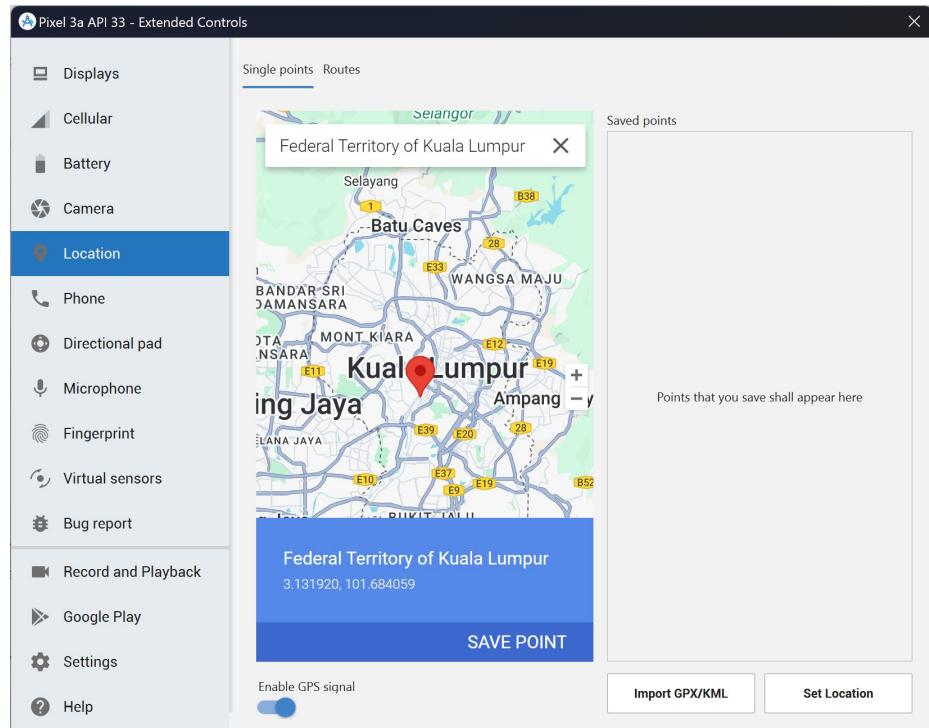


# Mock Location Data

Use mock location for testing purposes

Providing mock location data by injecting GPS location data

Using Dalvik Debug Monitor Server to set location to AVD



# Location Services

# Location Services

1. Geocoding - convert address into geographic coordinates (latitude and longitude) or the reverse
2. Geolocation - determine current location
3. Map - display maps and, markers

# Geocoding and Geolocation

Add the required dependencies, the geolocator and geocoding packages to your pubspec.yaml file.

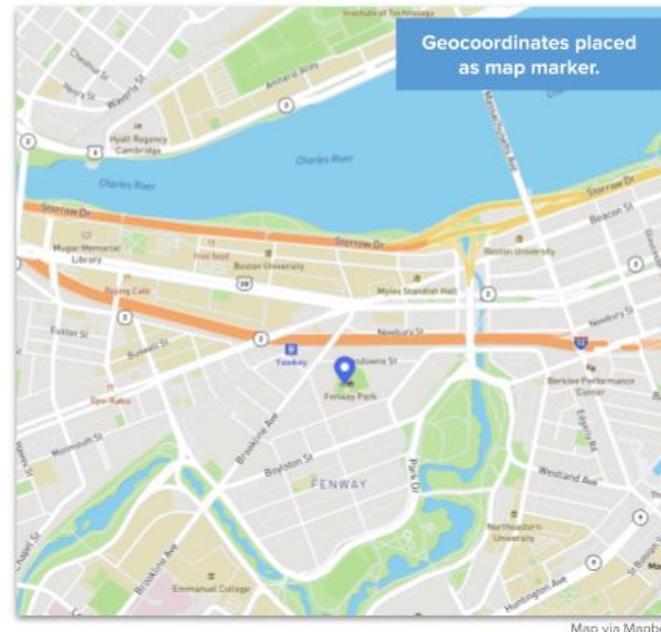
dependencies:

```
geolocator: ^[version code]
```

```
geocoding: ^[version code]
```

# Geocoding

Geocoding = converting a geographic location to an address



# Geocoding

- Geocoding APIs can compensate for ambiguous addresses — ones that are misspelled or inaccurate.
- For example, street addresses can change; address coordinates won't.
- Geocoding integrates and aligns address information with geographic codes to verify the address.

# Geocoding

Possible errors:

- No location data provided
- Invalid latitude or longitude used
- No geocoder available
- No address found

# Geocoding

Dependency: geocoding

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

# Geocoding

```
Future<void> getAddressFromCoordinates(double latitude, double longitude) async {
  try {
    List<Placemark> placemarks = await placemarkFromCoordinates(latitude, longitude,);

    if (placemarks.isNotEmpty) {
      Placemark placemark = placemarks.first;
      String address = '${placemark.street}, ${placemark.locality},
                      ${placemark.country}';
      print(address);
    } else {
      print('Could not obtain address');
    }
  } catch (e) {
    print('Error: $e');
  }
}
```

# Geolocation

Use the device location detection hardware to determine Current Location

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

...
Future<Position> getCurrentPosition() async {
    Position position = await Geolocator.getCurrentPosition(
        desiredAccuracy: LocationAccuracy.best,
    );
    return position;
}
```

# Location Services

## Android

Add the following lines below to your gradle.properties file:

```
android.useAndroidX=true  
android.enableJetifier=true
```

# Location Services

## Android

Make sure to set the compileSdkVersion in your android/app/build.gradle file to >= 33.

```
android {  
    compileSdkVersion 33  
  
    // ...  
}
```

# Location Services

## Android

Add either the ACCESS\_FINE\_LOCATION (previse location) or the ACCESS\_COARSE\_LOCATION (results equal to about a city block) permission your android/app/src/main/AndroidManifest.xml file.

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

# Location Services

## iOS

Add the following lines inside `ios/Runner/Info.plist` to access the device's location

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>This app needs access to location when open.</string>
<key>NSLocationAlwaysUsageDescription</key>
<string>This app needs access to location when in the background.</string>
```

# Map

# Map

There are several map service providers that you can use to integrate maps into your Flutter app:

1. Google Maps
2. Mapbox
3. Apple Maps
4. Here
5. Azure Maps
6. OpenStreetMap
7. TomTom

# Google Maps

To enable Google Maps in your Flutter app, you'll need the `google_maps_flutter` dependency:

```
dependencies:  
  google_maps_flutter: ^2.0.0
```

# Google Maps

Create a new Google Cloud Platform project.

Enable the Google Maps Platform API.

Create an API key and restrict its usage to your app's package name and SHA-1 fingerprint.

# Google Maps

## Android

In the AndroidManifest.xml file, insert:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <application ...>

        <meta-data android:name="com.google.android.geo.API_KEY"
            android:value="YOUR_API_KEY_HERE"/>

    <activity ...>
```

# Google Maps

## iOS

In your Info.plist file (iOS), add the API key to the GoogleMapsApiKey key:

```
<key>GoogleMapsApiKey</key>
<string>YOUR_API_KEY</string>
```

# Google Maps

Import the necessary packages:

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

# Google Maps

## Create a GoogleMap Widget

```
class _GoogleMapExampleState extends State<GoogleMapExample> {

  late GoogleMapController _controller;

  static const _initialCameraPosition = CameraPosition(
    target: LatLng(37.42796133580664, -122.085749655962),
    zoom: 14.4746,
  );

  void _onMapCreated(GoogleMapController controller) {
    _controller = controller;
  }
  ...
}
```

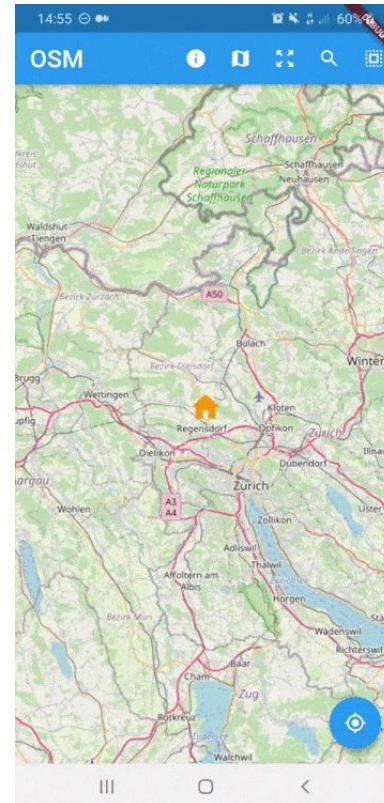
# Google Maps

## Create a GoogleMap Widget

```
...
@Override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: Text('Google Maps'),
        ),
        body: GoogleMap(
            mapType: MapType.hybrid,
            initialCameraPosition: _initialCameraPosition,
            onMapCreated: _onMapCreated,
        ),
    );
}
```

# OpenStreetMap

OpenStreetMap (abbreviated OSM) is an online map that uses an open geographic database, updated and maintained by a community of volunteers via open collaboration.



# OpenStreetMap

Use the Flutter map package:

`flutter_map`

Open access, no account, no fees

# OpenStreetMap

```
class _OSMFlutterMapState extends State<OSMFlutterMap> {
  @override
  Widget build(BuildContext context) {
    return FlutterMap(
      options: const MapOptions(
        initialZoom: 10,
        initialCenter: LatLng(3.140853, 101.693207),
      ),
      children: [
        TileLayer(
          urlTemplate: 'https://tile.openstreetmap.org/{z}/{x}/{y}.png',
          userAgentPackageName: 'com.example.demo_opensreetmap',
        ),
      ],
    );
  }
}
```

# Location best practices

# Location best practices

1. Remove location updates
2. Set timeouts
3. Batch requests
4. Passive location updates

# Location best practices

## 1. Remove location updates

Use `Geolocator.removeListener()`: When you no longer need to track the user's location, remove the listener to conserve battery and resources.

Check App State: Only request location updates when the app is actively using them.

User Interaction: Trigger location updates based on user actions, such as tapping a button or opening a specific screen.

# Location best practices

## 2. Set Timeouts

Reasonable Timeouts: Set appropriate timeouts for location updates to prevent excessive battery drain.

Dynamic Timeouts: Adjust timeouts based on the app's current state and user activity.

Background Mode Timeouts: If using background location updates, consider setting longer timeouts to balance accuracy and power consumption.

# Location best practices

## 3. Batch Requests

Combine Multiple Requests: If your app needs multiple location updates, combine them into a single request to reduce network overhead and battery usage.

Throttle Requests: Limit the frequency of location updates, especially when the user is not actively using the location-based features.

# Location best practices

## 4. Passive Location Updates

To minimize battery consumption and improve user experience, it's often beneficial to rely on system-provided location data rather than continuously tracking the user's location.

# Find Out More

Google Play Services, <https://developers.google.com/android/guides/overview>

Location Request,

<https://developers.google.com/android/reference/com/google/android/gms/location/LocationRequest>

Current Location,

<https://medium.com/@fernandoptr/how-to-get-users-current-location-address-in-flutter-q-elocator-geocoding-be563ad6f66a>

# Chapter 6

Specialized Instrument and Devices

# Contents

## Camera

Use an existing camera app

Build your own camera function

## Media

## Sensors

# Camera

The integration of cameras into mobile devices has revolutionized the way we interact with technology.

Basic app features:

1. Visual content creation - take photos and videos
2. Document scanning
3. QR code scanning

# Camera

Advanced features:

1. Augmented Reality (AR): Cameras can be used to overlay digital content onto the real world, creating immersive experiences.
2. Image Recognition: Apps can recognize objects, text, or faces in images to provide additional information or functionality.
3. Facial Recognition: This technology can be used for secure authentication, unlocking devices, or enabling personalized experiences.

# What can your phone camera do?

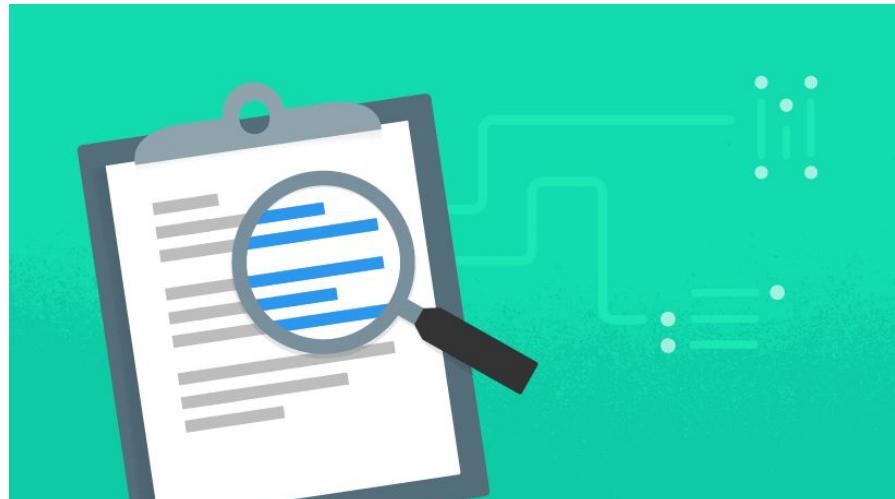
[Claim vehicle insurance](#)

[Cheque deposit](#)

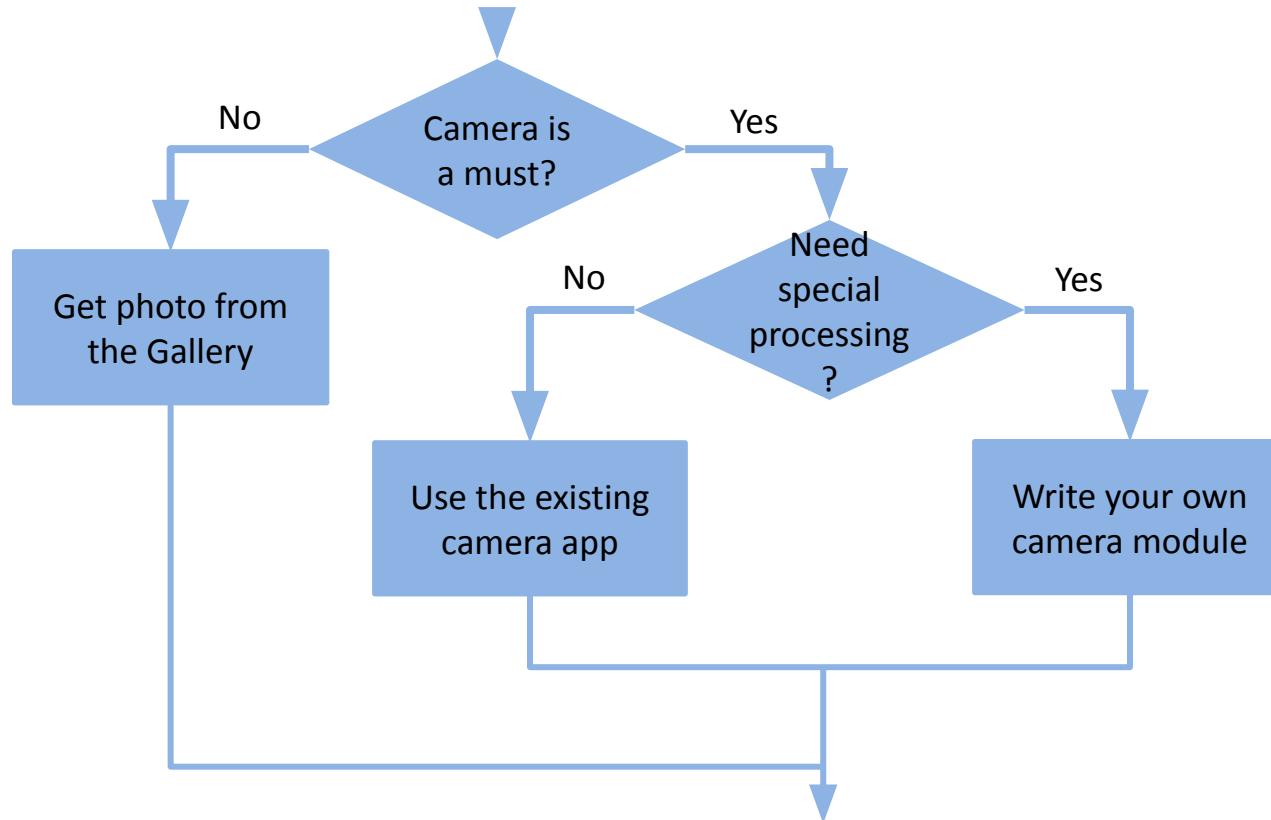
[Search](#)

ഉമാസ്ഥിപ്പയർ

[Eye test](#)



# Camera - Considerations



# Camera Methods

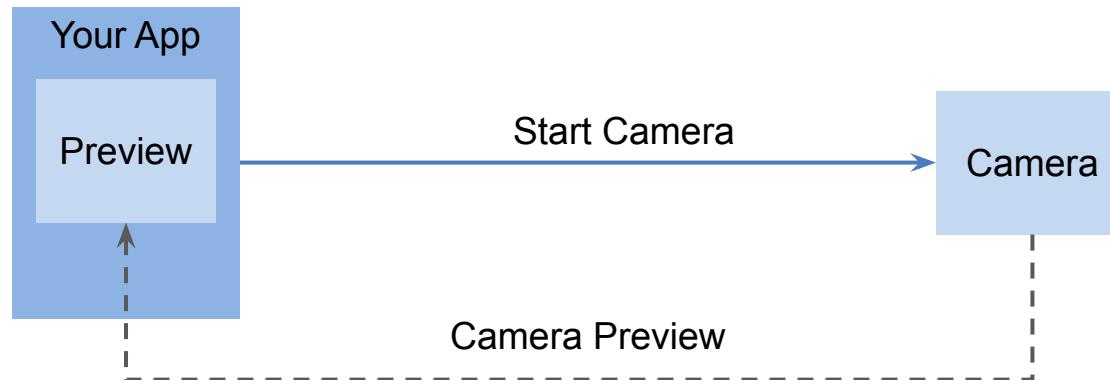
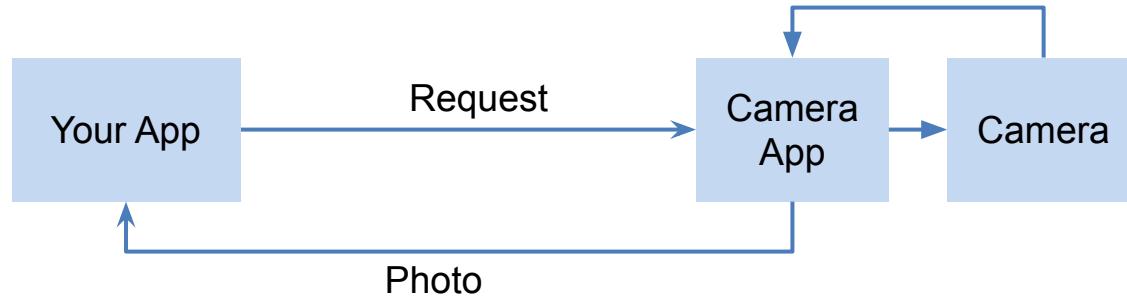
## Use an existing camera app

- Simple implementation - using existing packages
- Obtain a photo from the camera app or the gallery
- No need permission

## Build your own camera function

- Complex implementation - write your own code
- Create a camera preview
- Request the Camera permission

# Camera Methods



Use an existing camera app

# Use an existing camera app

Flutter uses the `image_picker` to capture images and videos from the device's camera or gallery.

Insert dependencies:

```
image_picker: ^version_code
```

Import package:

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

# Use an existing camera app

You may also include the following to find path to store images or to access the device file path:

[path\\_provider](#)

Finds the correct paths to store images.

[path](#)

Creates paths that work on any platform.

# Use an existing camera app

```
Future<void> _pickImage() async {
  final pickedFile = await ImagePicker().pickImage(source: ImageSource.camera);
  if (pickedFile != null) {
    setState(() {
      _image = File(pickedFile.path);
    });
  }

  // Save the image to the device's gallery (Android-specific)
  if (Platform.isAndroid) {
    final galleryPath = '/sdcard/DCIM/Camera';
    final destinationFile =
        File('$galleryPath/${DateTime.now().millisecondsSinceEpoch}.jpg');
    await pickedFile.saveTo(destinationFile.path);
  }
}
```

# Build your own camera function

# Build your own camera function

## Camera Features

In the context of Android, to prevents your app from being installed to devices that do not include a camera

```
<uses-feature android:name="android.hardware.camera" />
```

# Build your own camera function

## Storage Permission

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"  
    android:maxSdkVersion = "18" />
```

## Audio Recording Permission

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

## Location Permission

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

# Build your own camera function

The camera plugin provides tools to get a list of the available cameras, display a preview coming from a specific camera, and take photos or videos.

Steps:

1. Add the required dependencies.
2. Get a list of the available cameras.
3. Create and initialize the CameraController.
4. Use a CameraPreview to display the camera's feed.
5. Take a picture with the CameraController.
6. Display the picture with an Image widget.

# Build your own camera function

Step 1: Add the required dependencies.

`camera`

- Provides tools to work with the cameras on the device.

`path_provider`

- Finds the correct paths to store images.

`path`

- Creates paths that work on any platform.

# Build your own camera function

Step 1: Add the required dependencies.

To add the packages as dependencies, run flutter pub add:

```
$ flutter pub add camera path_provider path
```

# Build your own camera function

Step 1: Add the required dependencies.

Android

- Update minSdkVersion to 21 (or higher).

# Build your own camera function

Step 1: Add the required dependencies.

iOS

- Inside ios/Runner/Info.plist in order to access the camera and microphone.

```
<key>NSCameraUsageDescription</key>
```

```
<string>Explanation on why the camera access is needed.</string>
```

```
<key>NSMicrophoneUsageDescription</key>
```

```
<string>Explanation on why the microphone access is needed.</string>
```

# Build your own camera function

Step 2: Create and initialize the CameraController.

```
Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized(); // Ensure that plugin services are initialized

  // Obtain a list of the available cameras on the device.
  final cameras = await availableCameras();

  // Get a specific camera from the list of available cameras.
  final firstCamera = cameras.first;

  runApp(
    MaterialApp(
      theme: ThemeData.dark(),
      home: TakePictureScreen(
        // Pass the appropriate camera to the TakePictureScreen widget.
        camera: firstCamera,
      ),
    ),
  );
}
```

# Build your own camera function

Step 2: Create and initialize the CameraController.

```
// A screen that allows users to take a picture using a given camera.  
class TakePictureScreen extends StatefulWidget {  
  const TakePictureScreen({  
    super.key,  
    required this.camera,  
  });  
  
  final CameraDescription camera;  
  
  @override  
  TakePictureScreenState createState() => TakePictureScreenState();  
}
```

# Build your own camera function

Step 3: Create and initialize the CameraController.

```
class TakePictureScreenState extends State<TakePictureScreen> {
    late CameraController _controller;
    late Future<void> _initializeControllerFuture;

    @override
    void initState() {
        super.initState();
        _controller = CameraController( // Create a CameraController.
            Widget.camera,           // Get a specific camera
            ResolutionPreset.medium, // Define the resolution to use.
        );
        _initializeControllerFuture = _controller.initialize();
    }
    ...
}
```

# Build your own camera function

Step 3: Create and initialize the CameraController.

```
class TakePictureScreenState extends State<TakePictureScreen> {  
  ...  
  @override  
  void dispose() {  
    _controller.dispose(); // Dispose of the controller  
    super.dispose();  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Container(); // Fill this out in the next steps.  
  }  
}
```

# Build your own camera function

Step 4: Use a CameraPreview to display the camera's feed

```
FutureBuilder<void>(
  future: _initializeControllerFuture,
  builder: (context, snapshot) {
    if (snapshot.connectionState == ConnectionState.done) {
      // If the Future is complete, display the preview.
      return CameraPreview(_controller);
    } else {
      // Otherwise, display a loading indicator.
      return const Center(child: CircularProgressIndicator());
    }
  },
)
```

# Build your own camera function

Step 5 : Take a picture with the CameraController

```
FloatingActionButton(  
    onPressed: () async {  
        try {  
            await _initializeControllerFuture; // Ensure that the camera is initialized.  
            final image = await _controller.takePicture(); // Attempt to take a picture  
        } catch (e) {  
            // If an error occurs, log the error to the console.  
            print(e);  
        }  
    },  
    child: const Icon(Icons.camera_alt),  
)
```

# Build your own camera function

Step 6: Display the picture with an Image widget

```
Image.file(File(imagePath)),
```

# Google ML Kit

Google's ML Kit is a mobile SDKs that allows developers to easily integrate machine learning capabilities into their Android, iOS, and web applications.



Text  
recognition

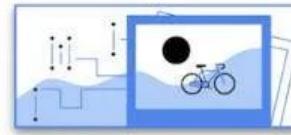
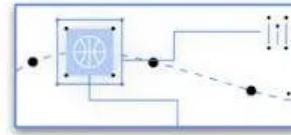


Image  
labeling



Barcode  
scanning



Object  
detection  
and tracking



Face  
detection

# Media

# Media

Mobile devices maintain separate audio streams for playing music, alarms, notifications, the incoming call ringer, system sounds, in-call volume, etc

Most of these streams are restricted to system events

We focus on playing audio using the music stream

# Media

Plays local (assets) and external (streaming) files

Supports any media codec that is provided by the platform and those that are device-specific

Recommendation: use core media formats

Use the `video_player` plugin to play videos stored on the file system, as an asset, or from the internet (Note: Not available on Linux and Windows)

# Media

## Permission declaration

If you are using MediaPlayer to stream network-based content

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

If your player application needs to keep the screen from dimming or the processor from sleeping

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

# Media

Core media file formats:

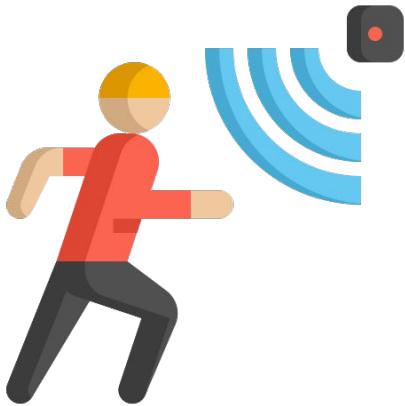
Audio: .3gp .mp3 .mp4 .mid .wav .ogg

Picture: .jpg .gif .png .bmp

Video: .3gp .mp4

# Sensors

# Sensors



Motion



Environmental

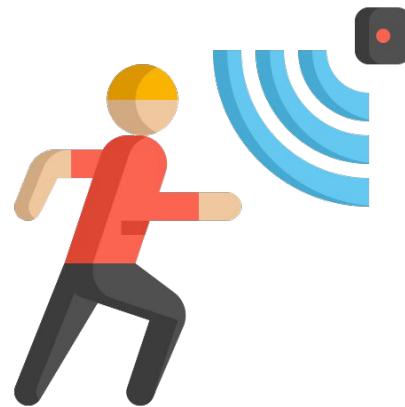


Position

# Motion sensors

Measure acceleration forces and rotational forces along three axes:

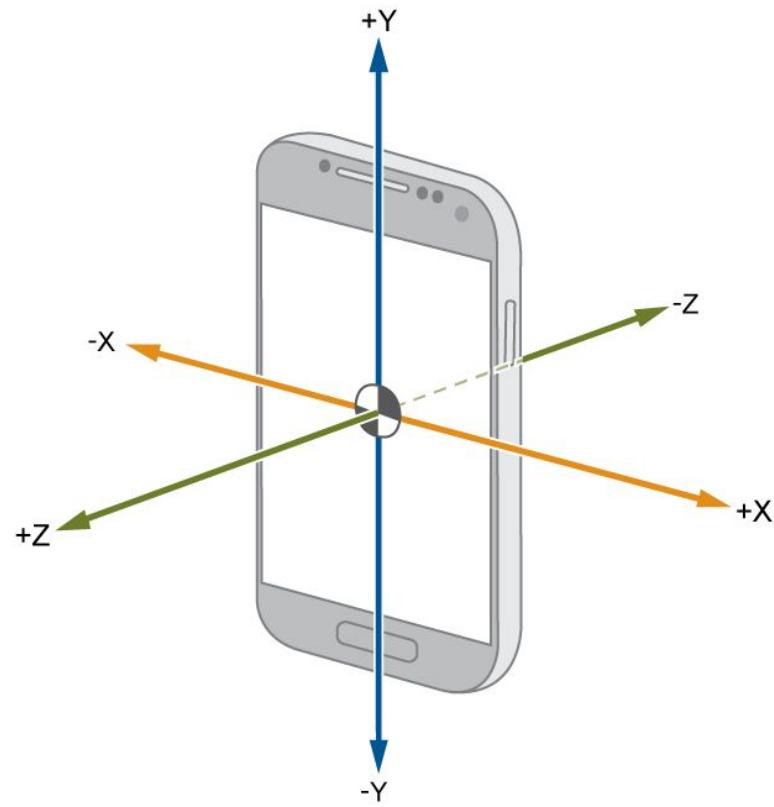
1. Accelerometers
2. Gravity sensors
3. Gyroscopes
4. Rotation vector sensors



# Motion sensors

## 1. Accelerometers

Provides data on the rate of change in **velocity** - the speed in combination with the **direction** of **motion** of an object, excluding gravity

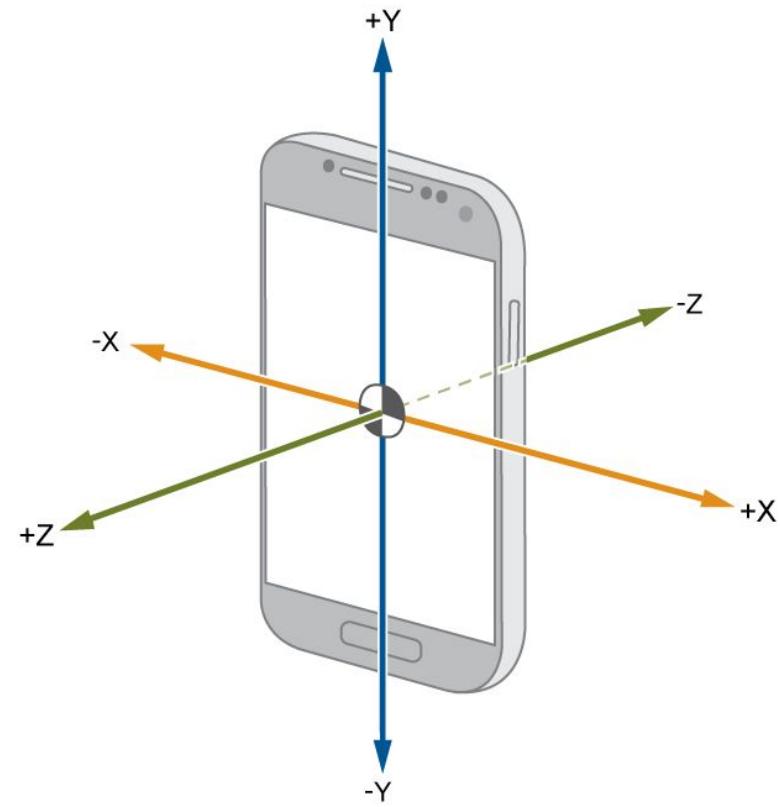


# Motion sensors

## 2. Gravity sensors

It measures the **direction** and **intensity** of gravity

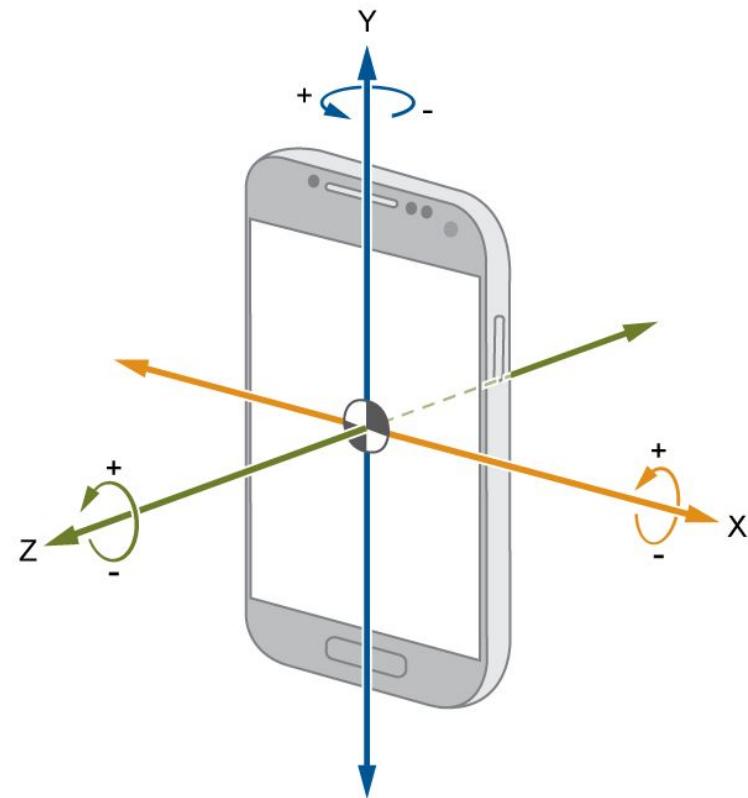
Checks the **relative direction** of a device within a space



# Motion sensors

## 3. Gyroscopes

It helps the accelerometer out with understanding which way your phone is **orientated**.

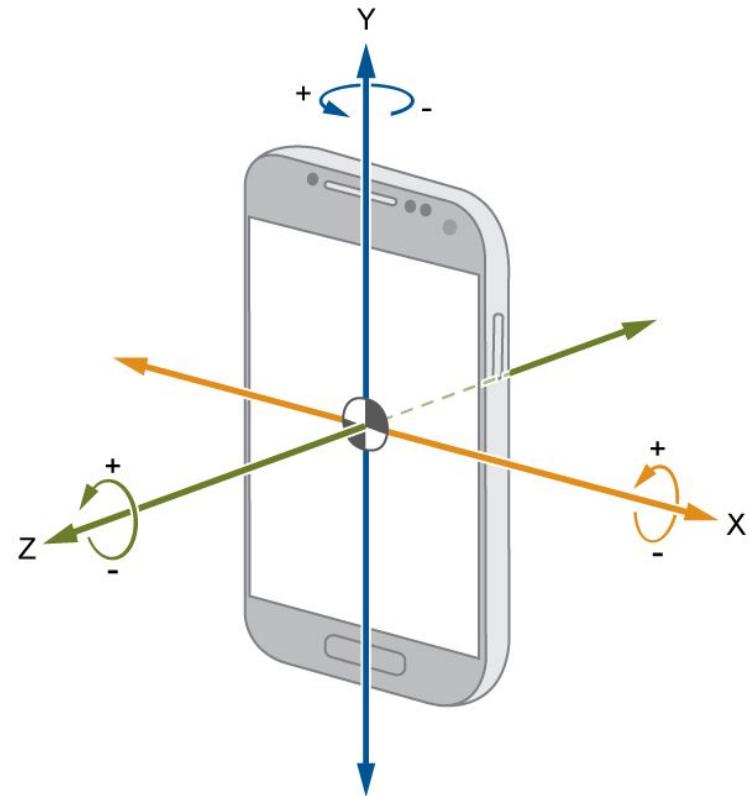


# Motion sensors

## 4. Rotation vector sensors

To monitor and measure **turning movements**

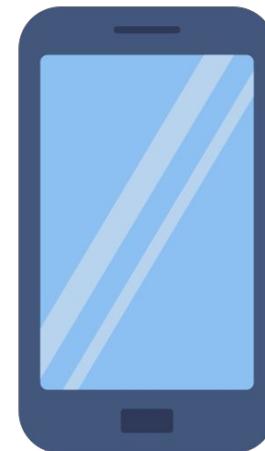
Combination of an angle and an axis



# Environmental sensors

Measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity **near a mobile device**.

1. Barometers
2. Photometers, and
3. Thermometers

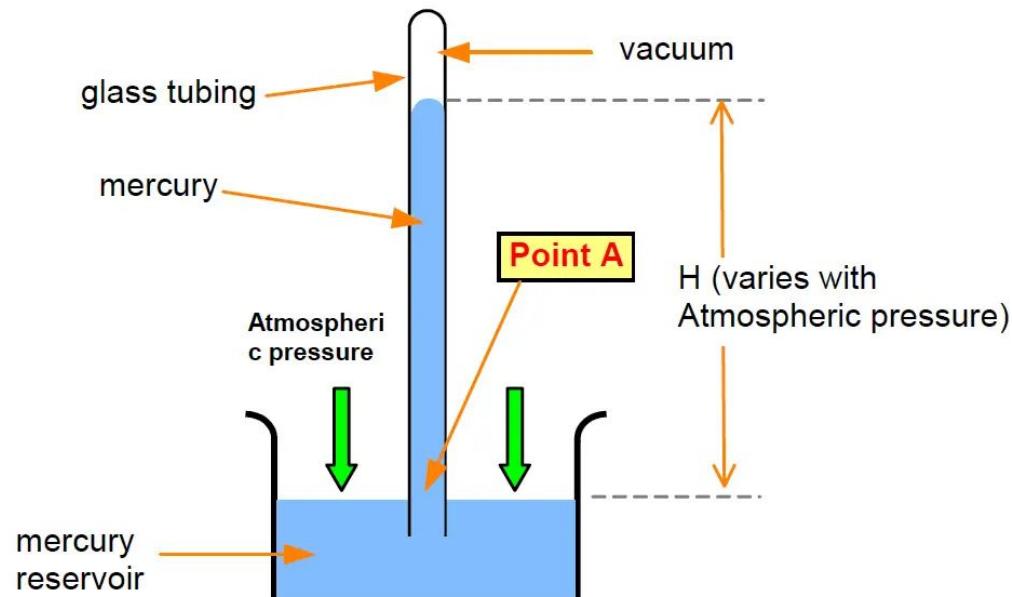


# Environmental sensors

## Barometer

It measures **atmospheric pressure**

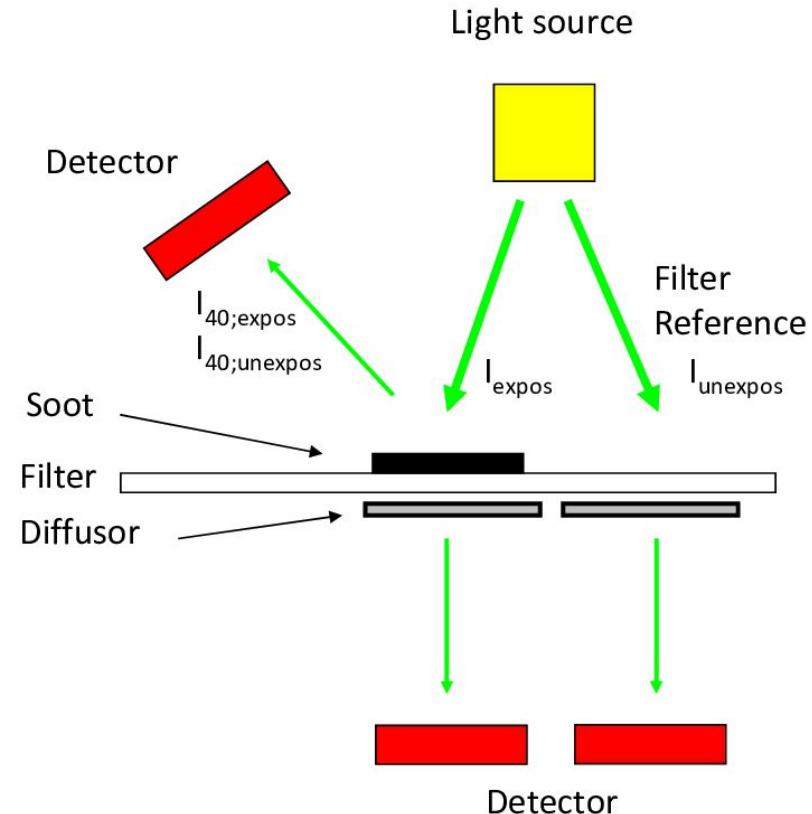
It **assists the GPS** chip inside the device to get a faster lock by instantly delivering **altitude** data (vertical location).



# Environmental sensors

## Photometer / Light sensor

It senses the amount of **ambient light** present

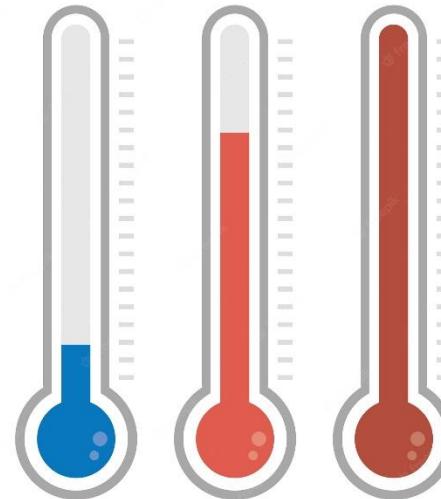


# Environmental sensors

## Thermometer

It measures **temperature** within a mobile device

To avoid overheating of the internal components



# Position sensors

Measure the physical position of a device

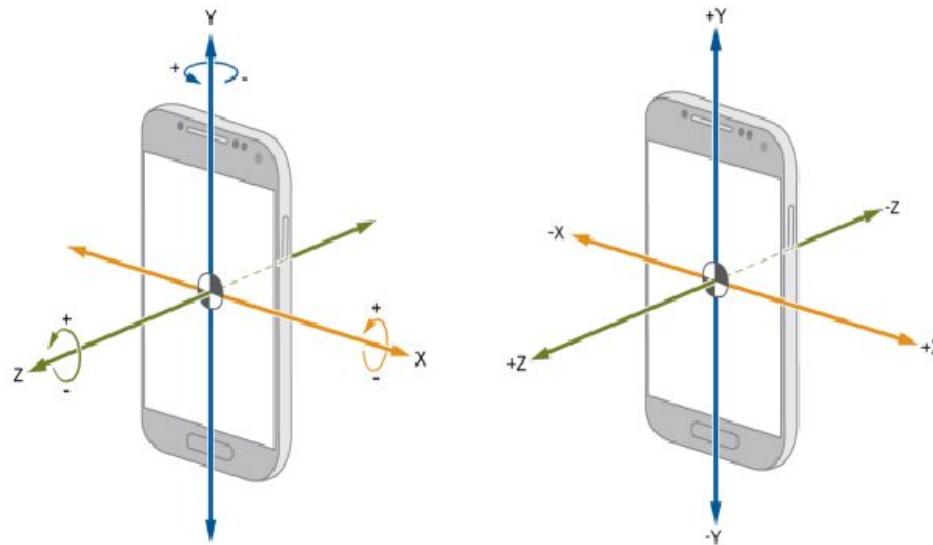
- Orientation sensors
- Magnetometers



# Position sensors

## Orientation sensors

Measures the orientation of a device relative to an orthogonal coordinate frame



# Position sensors

## Magnetometers

Determines your location with respect to Magnetic North (or South!)



# Other Sensors

# Other sensors

## Biometrics

Biometric authentication is an approach to multi-factor authentication (MFA) to verify an individual's identity that uses possession of a mobile device as a first factor and use of that device to verify a unique biometric identifier as a second factor.

Flutter has `local_auth` package, which provides local authentication with biometrics such as fingerprint or facial recognition. (Note: supports Android, iOS, macOS and Windows )

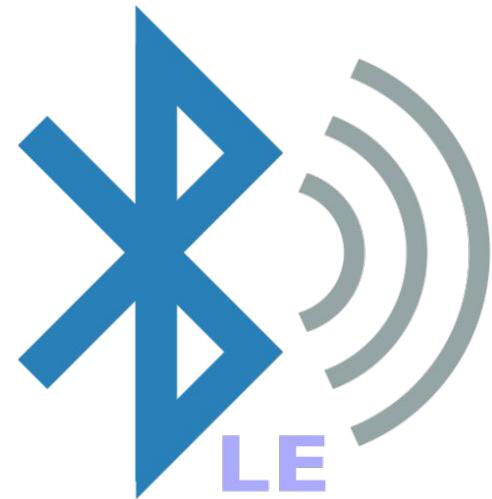
# Other sensors

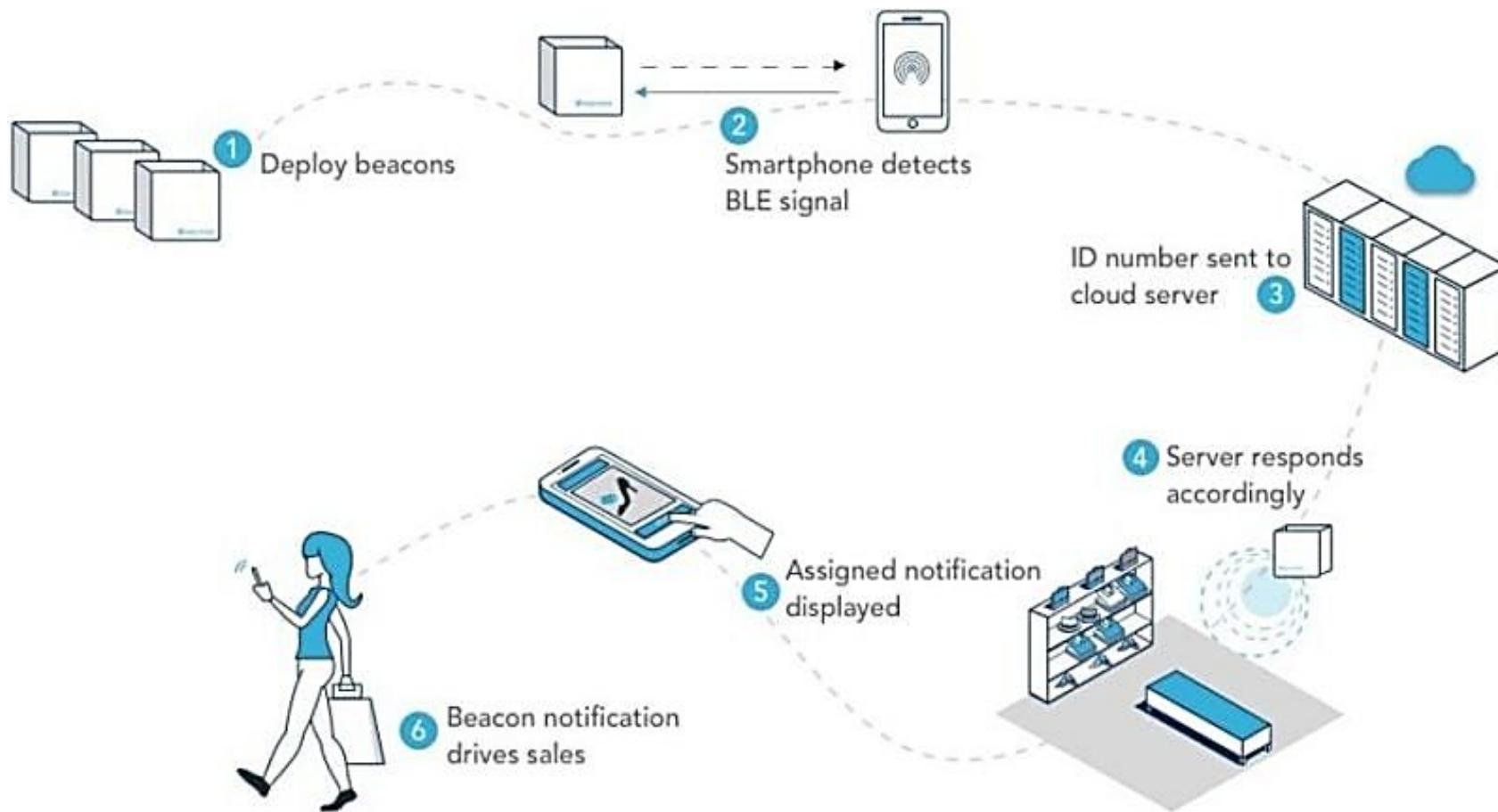
## Bluetooth Low Energy (BLE)

A short-range wireless communication technology designed for low-power applications

Low power consumption, high data rate

Applications: wearables, IoT, beacon technology, healthcare, indoor navigation, etc





# Other sensors

## Near-field communication (NFC)

A short-range wireless technology

Allows two electronic devices to communicate with each other

Applications: mobile payments, data exchange, access control, product information, etc



# Near-field communication (NFC)



# Other sensors

## Nearlink

A short-range wireless communication technology developed by the NearLink Alliance, led by Huawei

High speed, high reliability, high security, low latency, and low energy

Applications: wearable, IoT, automotive, consumer products , etc



**NearLink**



Technology	BLE	NFC	Nearlink
Range	Up to 100 meters	Up to 4 cm	Up to 100 meters
Speed	Up to 2 Mbps	Up to 424 Kbps	Up to 900 Mbps
Power Consumption	Low	Very Low	Lowest (60% of BLE)
Latency	Moderate	Very Low	Ultra-low

# References

Image Picker, [https://pub.dev/packages/image\\_picker](https://pub.dev/packages/image_picker)

Build your own camera function,

<https://docs.flutter.dev/cookbook/plugins/picture-using-camera>

Audio File, [https://pub.dev/packages/just\\_audio](https://pub.dev/packages/just_audio)

Video Player, [https://pub.dev/packages/video\\_player](https://pub.dev/packages/video_player)

Google ML Kit, [https://pub.dev/packages/google\\_ml\\_kit](https://pub.dev/packages/google_ml_kit)

Local authentication, [https://pub.dev/packages/local\\_auth](https://pub.dev/packages/local_auth)

# Chapter 7

Mobile Application Packaging and  
Publication

# Contents

Checklist to plan your app launch

Prepare for release

App Bundles

Android Go

Google Play Instant

Support Chrome OS

Monetize your app

# Checklist

1. Developer Program Policies
2. Developer Account
3. Localization
4. Device Compatibility
5. Quality Test : Alpha & Beta
6. Store Listing

# Check List

## 1. Developer Program Policies

Mobile app stores, such as the Google Play Store, Apple App Store and Huawei App Gallery, have specific guidelines and policies that developers must adhere to

These policies are designed to maintain a safe, secure, and positive user experience.



# Check List - Developer Program Policies



Restricted  
Content



Intellectual  
Property



Privacy and  
Security



Monetization  
and Ads



Store Listing  
and  
Promotion

# Check List - Developer Program Policies

Mobile app stores have strict rules about the type of content allowed, including explicit content, malware, and harmful apps.

Explicit Content: pornography, sexually suggestive content, and nudity



Restricted  
Content

# Check List - Developer Program Policies

App reviewers check for potential IP infringement, such as copyright infringement, trademark infringement, and patent infringement.



Intellectual  
Property

# Check List - Developer Program Policies

Developers must have clear privacy policies that explain how user data is collected, used, and shared

App stores may impose security requirements to protect user data from unauthorized access and breaches



Privacy and  
Security

# Check List - Developer Program Policies

App stores provide guidelines govern in-app purchases and subscriptions

Ad monetization and in-app purchases must adhere to guidelines, including fair pricing and transparent advertising.



Monetization  
and Ads

# Check List - Developer Program Policies

App stores control the listing and promotion (marketing materials) of apps through strict review processes, guidelines, and algorithms

Developers must adhere to these rules to ensure their apps are visible, accessible, and successful. E.g. setting of user's age groups, regions, etc



Store Listing  
and  
Promotion

# Check List

## Developer Program Policies Developer Account



# Developer Account

**It is a publishing account issued by Platform Provider to developer**

**It enables developer to post, display, offer for sale, and distribute apps through the Platform**

**Access developer tools, resources, and support provided by the platform.**

# Developer Account



USD 25  
(one time)



USD 99  
(per year)



FOC

KaiOS

FOC



FOC

 All apps Inbox 16 Policy status Users and permissions Order management▶  Download reports Account details Developer page Associated developer accounts Activity log▶  Setup

## All apps

[Create app](#)

View all of the apps and games that you have access to in your developer account

Pinned apps 

Pin apps here to access them quickly and view key metrics

## 4 apps

Filter by

[All](#) Search by app or package name

App	Installed audience	App status	Update status	Last updated		
 GBA Market Tracker com.mt.gba.gbamarketracker	33	Production		6 Oct 2019		
 Multilingual Storybook mmsr.mystorybook	1	Removed by you	 In review	9 Sep 2021		
 TAR UC Bus my.edu.tarc.testing2	0	Removed by you		12 Oct 2017		
 TAR UC Bus Module (Alpha) com.wy.wenyang.tarucbustracking...	0	Removed by you		3 Nov 2016		

## Welcome

Additional Resources

Documentation

Downloads

Forums

Feedback Assistant

Account Help

Contact Us

# Getting Started

Download Xcode, learn how to build an app, and install it directly on your Apple device.



## Download Tools

Get started with Xcode, Apple's integrated development environment for creating apps.



## Build Your First App

Use Xcode to write a simple "Hello World" app to get familiar with tools, SDKs, and the Swift language.

## Join the Apple Developer Program

Membership in the Apple Developer Program includes everything you need to develop, distribute, and manage your apps on the App Store. You'll also gain access to beta software, advanced app capabilities, beta testing tools, and app analytics.

## Ecosystem services



App services



Content services

## HMS API services



My APIs



API Library



Credentials



Privacy contact info

## Developer center



My reports



Customer service



My messages



My accounts



Settings

Distribution and promotion 

## AppGallery Connect

Your one-stop open platform covers the entire app lifecycle: innovation, development, distribution, operation, and analysis.

Monetize Service 

Publisher Service

Development 

Push Kit



HUAWEI IAP



HUAWEI ID



HUAWEI HIAI



Health Kit



Wear Engine

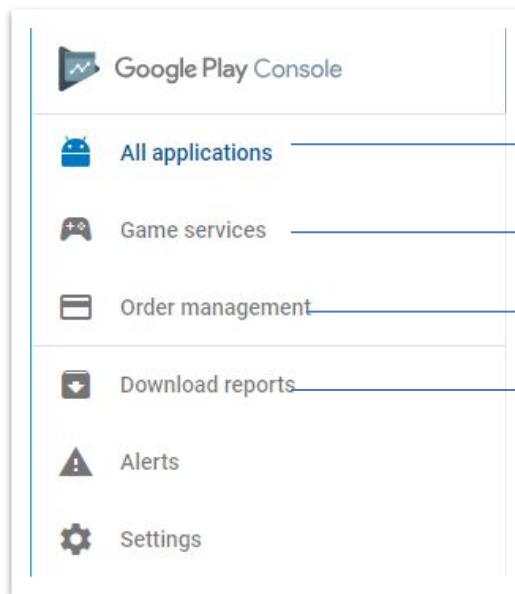
HarmonyOS Development  
Agreement

3D Object Recognition

New

# Developer Account

## Google Play Console - Features



All applications View your apps, active installs, rating, last update and status

Game services Add social gaming features to your games

Order management Earn money with paid apps or in-app products

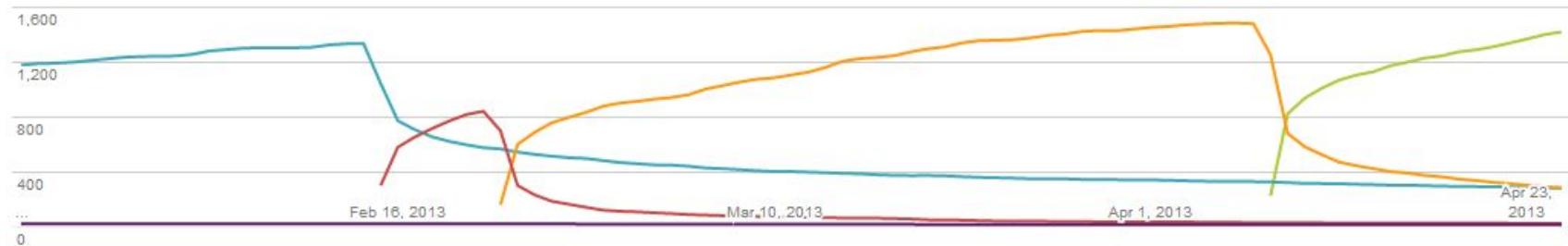
Download reports Know your app's performance

Alerts

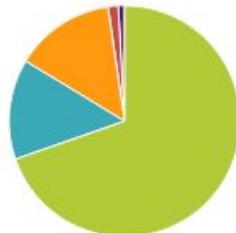
Settings

# Developer Account

## ACTIVE DEVICE INSTALLS BY APP VERSION



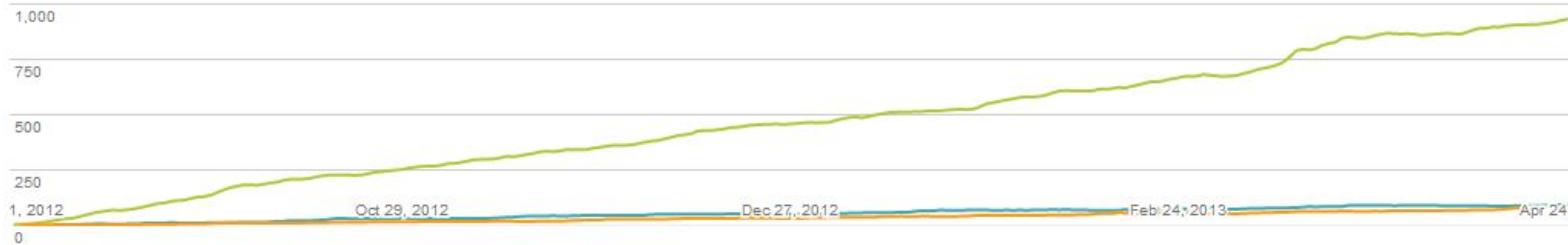
## ACTIVE DEVICE INSTALS ON APR 25, 2013



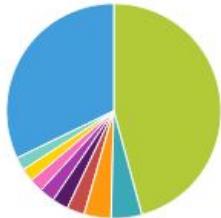
YOUR APP	
<input checked="" type="checkbox"/> 36	1,421 69.66%
<input checked="" type="checkbox"/> 21	286 14.02%
<input checked="" type="checkbox"/> 34	286 14.02%
<input checked="" type="checkbox"/> 31	29 1.42%
<input checked="" type="checkbox"/> 18	18 0.88%

# Developer Account

## ACTIVE DEVICE INSTALLS BY COUNTRY



## ACTIVE DEVICE INSTALS ON APR 25, 2013



YOUR APP		ALL APPS IN TOOLS		TOP 10 COUNTRIES FOR TOOLS	
United States	934	45.78%	23.86%	United States	23.86%
Canada	94	4.61%	1.33%	Japan	11.25%
United Kingdom	83	4.07%	2.94%	South Korea	9.64%
South Korea	54	2.65%	9.64%	Germany	3.88%
Germany	49	2.40%	3.88%	Russia	3.06%
France	47	2.30%	2.86%	Spain	2.97%
Australia	42	2.06%	1.12%	United Kingdom	2.94%
Indonesia	40	1.96%	1.10%	France	2.86%
India	40	1.96%	2.71%	India	2.71%
Others	657	32.21%		Brazil	2.52%

# Check List

Developer Program Policies

Developer Account

Localization



# Localization



The adaptation of an app to meet the needs of a particular **language, culture** or desired population's "look-and-feel."

A successfully localized app is one that appears to have been developed within the **local culture**.







# Check List

Developer Program Policies

Developer Account

Localization

Device Compatibility



# Device Compatibility



It ensures that an app runs efficiently **across mobile devices of different configurations**

Compatibility also should ensure app run well on mobile devices of different **hardware, software, and operating systems**

# Check List

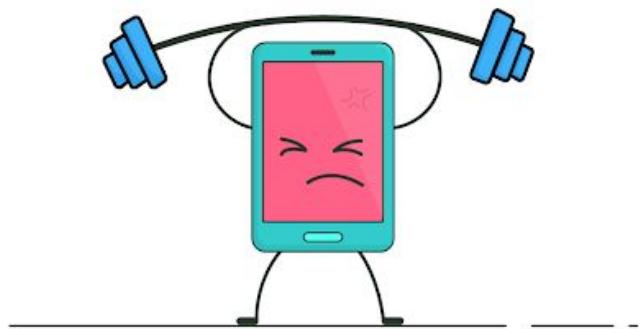
Developer Program Policies

Developer Account

Localization

Device Compatibility

Quality Test : Alpha & Beta



# Quality Test

A process to ensure apps **meet specified regulations and standards**.

It's a series of techniques that developers employ to **prevent issues from occurring and ensure they satisfy the customer** with their finished product.



# Check List

Developer Program Policies

Developer Account

Localization

Device Compatibility

Quality Test; Alpha & Beta

Store Listing



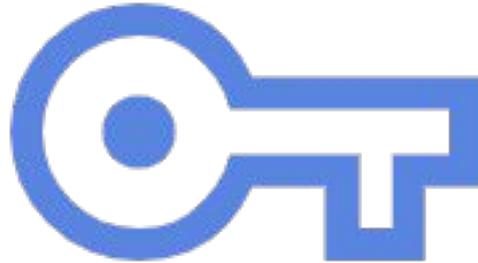
# Preparing for Release



To prepare your app so that users can install and run the app on their  
Android-powered devices

Release-ready .apk file is signed with your own certificate

# Preparing for Release

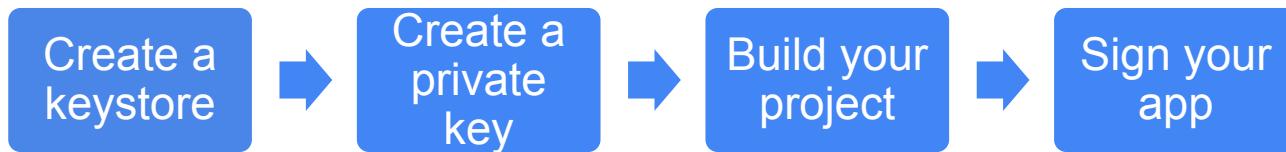


Minimum requirements:

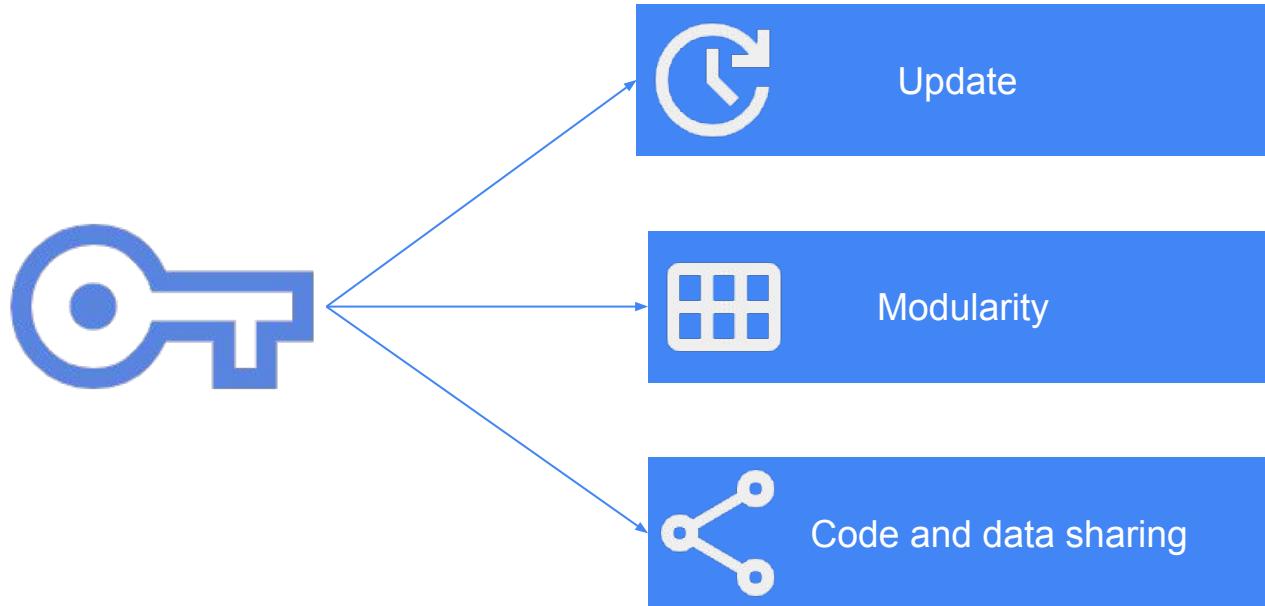
1. Cryptographic keys
2. Application icon
3. End-user License Agreement
4. Promotional and marketing materials

Cryptographic key: digitally signature that is owned by the application's developer

# Signing in Release Mode



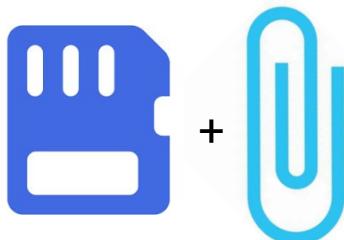
# Signing Consideration



# Store Listing



Distribution



App size  
<150 MB

Expansion  
up to 2GB

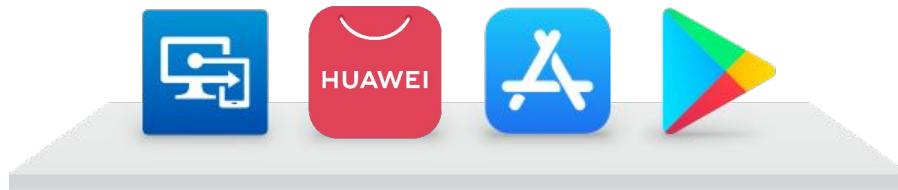


Platforms



Fee or Free

# Distribution



# Distribution Channels



Marketplace



Email



Website or server

# Distribution Methods - Android

1. Android Go
2. App Bundles
3. Google Play Instant
4. Support Chrome OS

# Android Go



Android Go is an OS designed for entry-level smartphones with limited resources

An initiative that brings the benefits of Android to a wider range of users

Android Go is optimized for devices with:

- Limited RAM, typically 1GB or less
- Lower processing power
- Less storage space

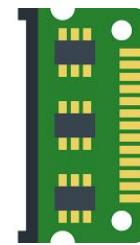
# Android Go



Target Oreo  
(API 26)



App size less  
than 40 MB



RAM usage  
below 50 MB  
(apps)  
150 MB  
(games)



Start your  
app under  
5 seconds

# App Bundles

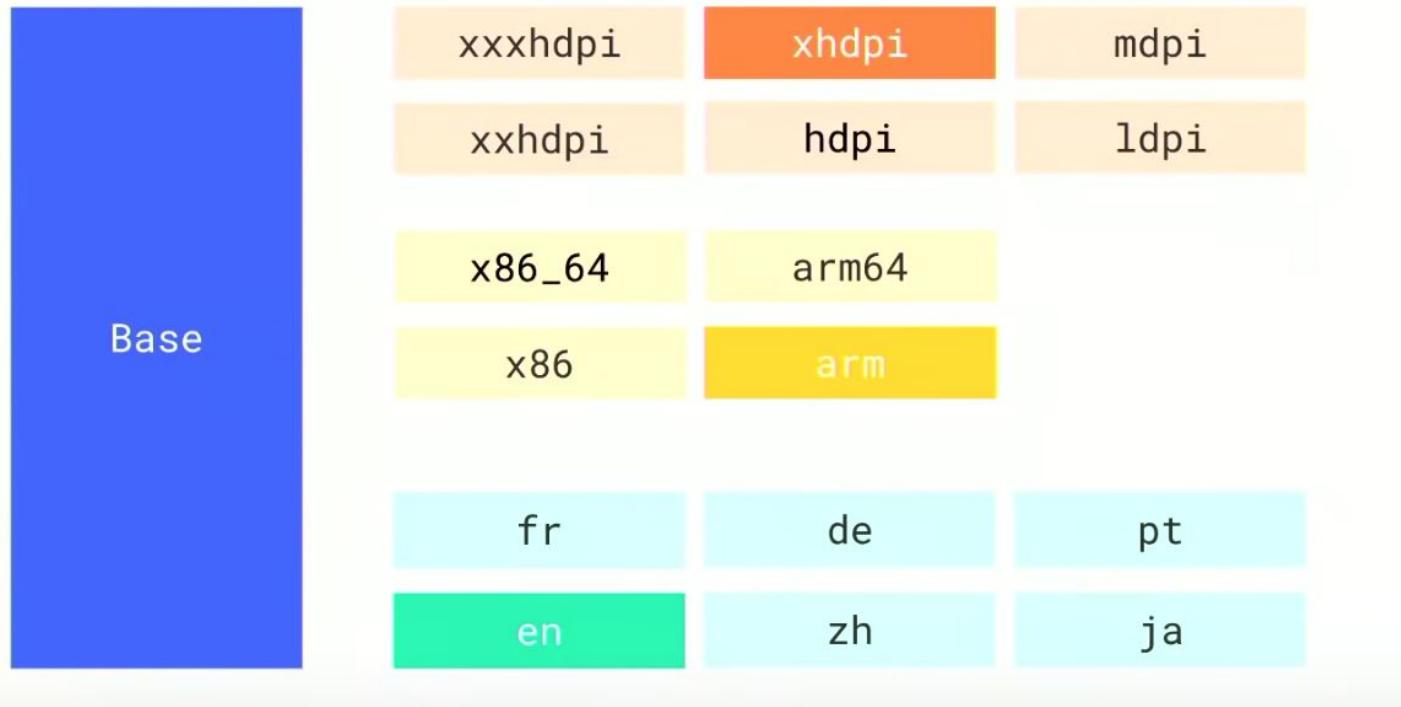


An efficient distribution technology for Android apps, especially on devices with varying screen sizes and capabilities

Instead of building separate APKs for each device configuration (screen size, CPU architecture, etc.), you create a single .aab (Android App Bundle) file.

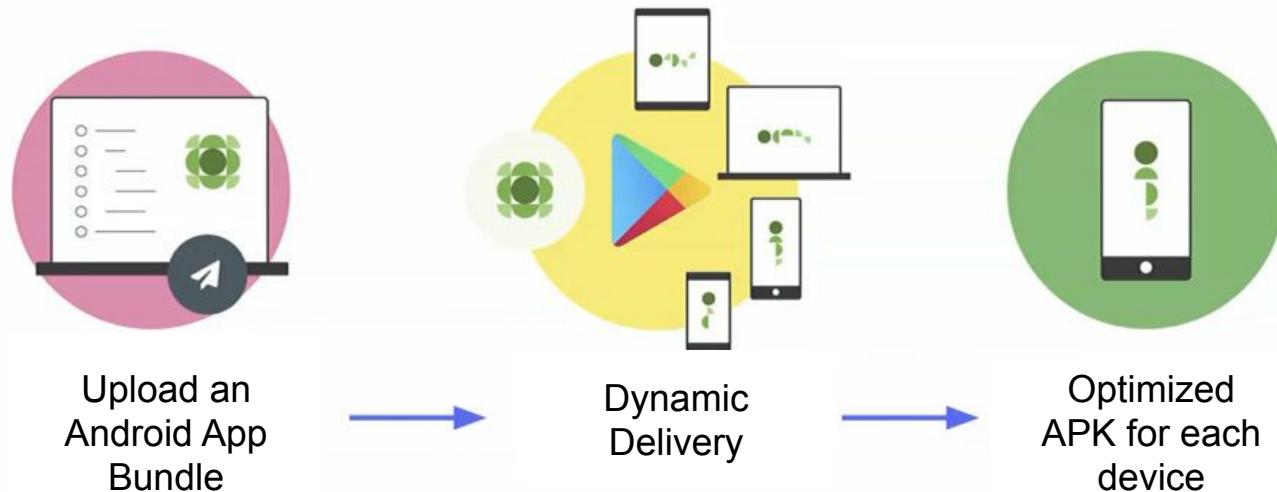
# App Bundles

This bundle contains all the compiled code and resources for your app.



# App Bundles

Google Play then dynamically delivers only the necessary code and resources for a specific user's device during installation.



# App Bundles



## Benefits:

1. Small app file: Users download only the parts of your app they need, resulting in significantly smaller download sizes.
2. Fast installation: Faster installation times due to smaller downloads.
3. Good performance: Optimized for specific devices, leading to better performance and potentially reduced battery consumption.
4. Easy to manage: Easier to manage app releases and updates.

# Google Play Instant

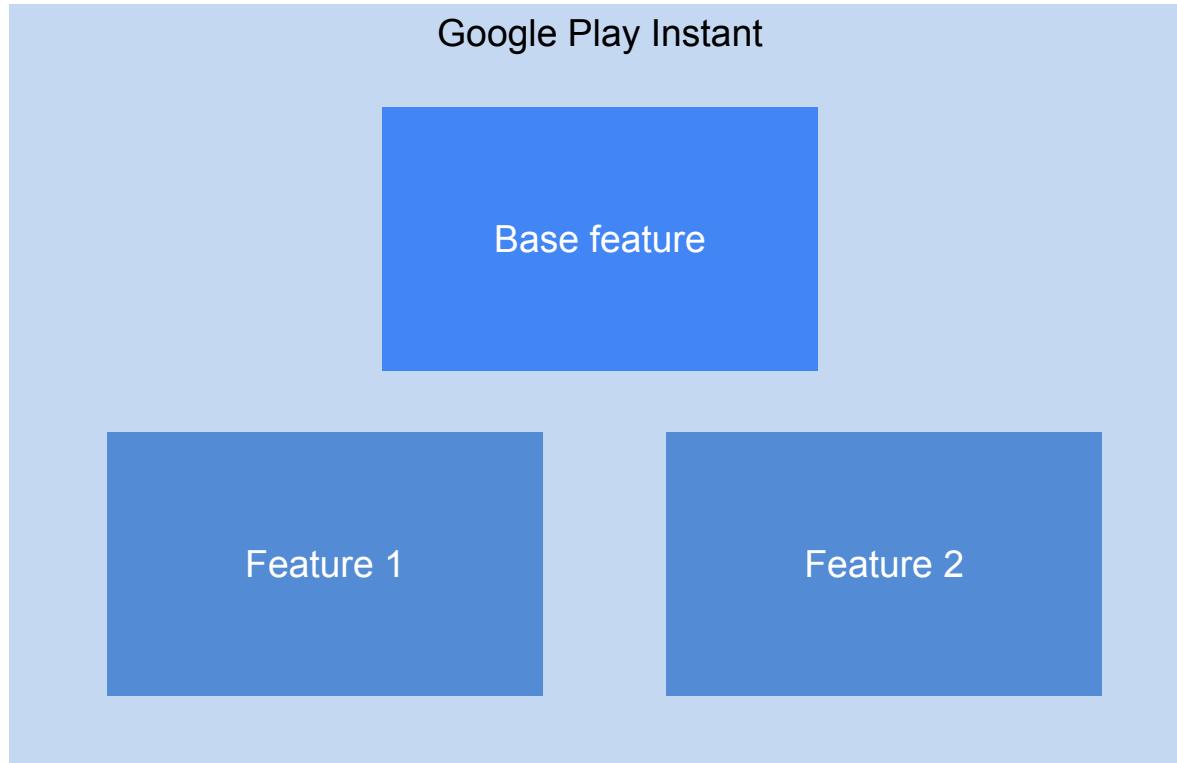


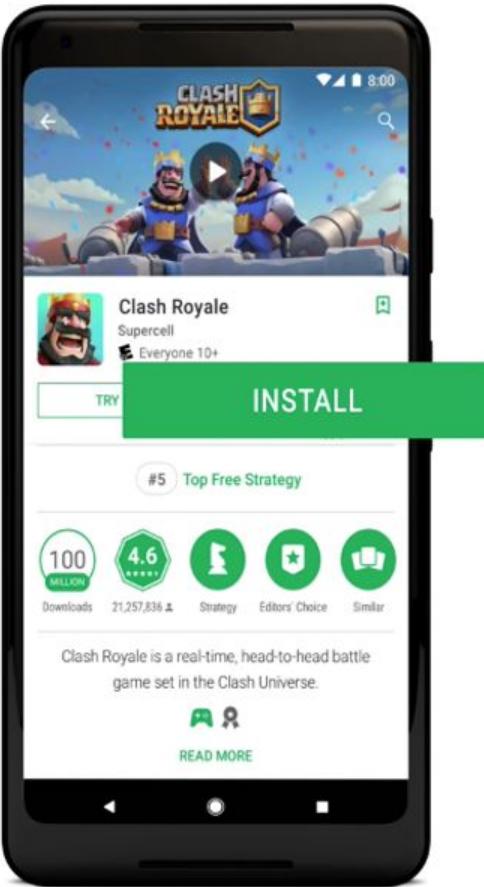
Google Play Instant allows users to try a portion of an app or game before downloading the full version

Users can access instant apps through links in search results, emails, ads, or social media

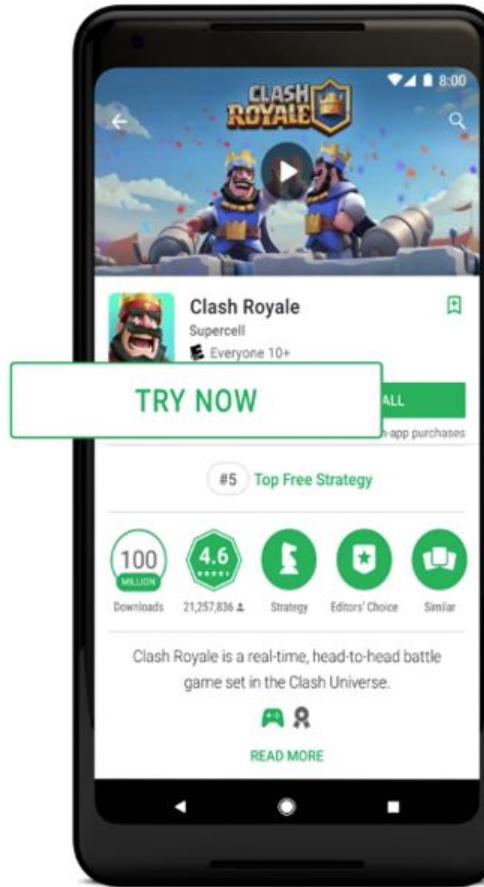
App has limited functionality, typically focuses on core app functionality or a specific game level

# Google Play Instant





Download



Instant

3

# Google Play Instant



## Benefits

Try Before You Buy: Users can experience the app before committing to a full download.

Reduced Storage Space: No need to install the full app to try it out.

Faster Access: Instant apps load quickly, allowing users to start using them immediately.

# Google Play Instant

## Limitations



Size Limitations: Instant apps have size restrictions to ensure fast loading times.

Functionality Limitations: Only a subset of the app's functionality may be available in the instant version.

# Chrome OS

It is an operating system developed by Google that is designed to work primarily with web applications



It uses the Chrome web browser as its main user interface.

# Chrome OS

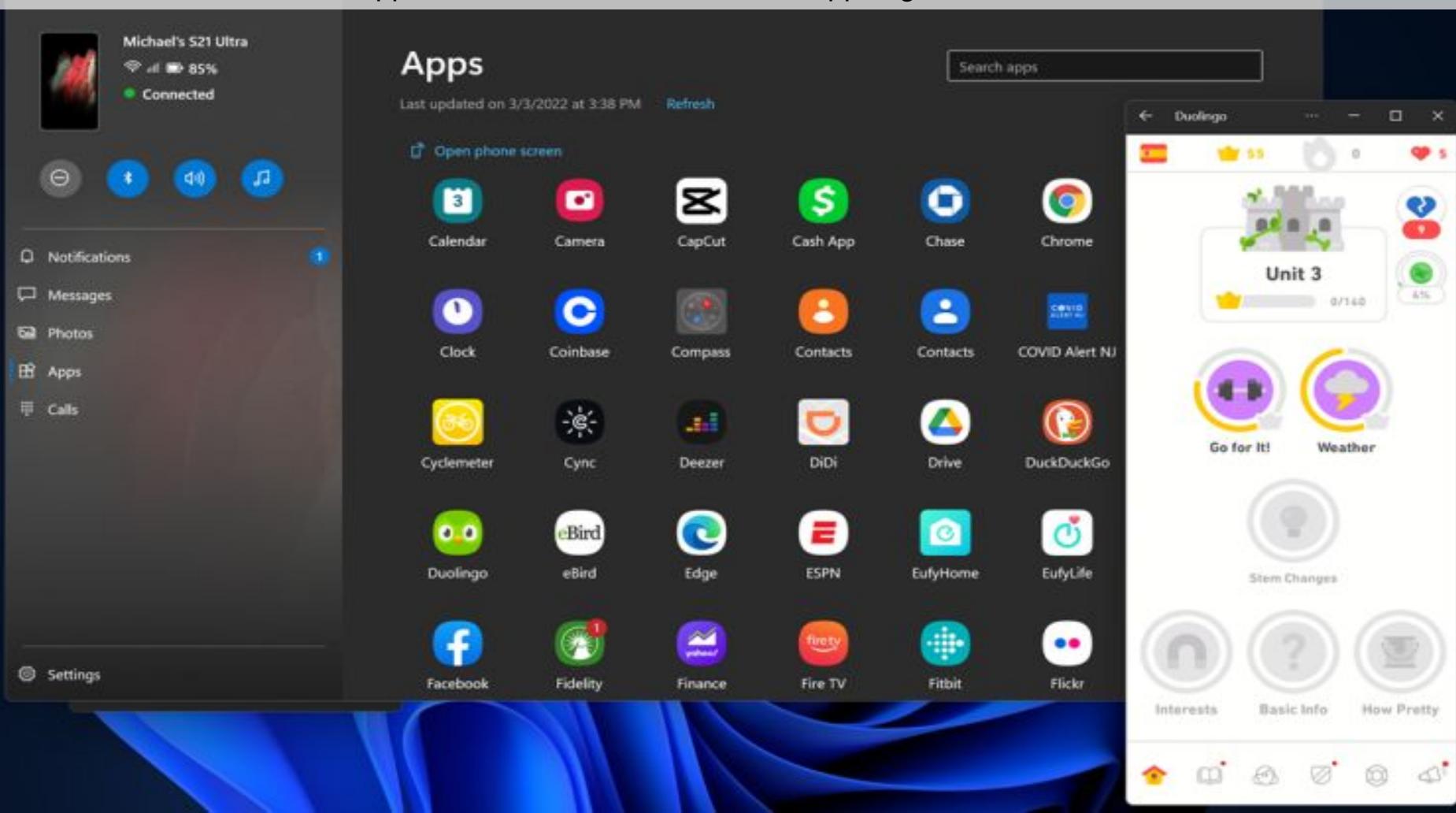
It supports the Google Play Store and Android apps

Preparation:

- Manifest file

```
<uses-feature  
    android:name="android.hardware.touchscreen"  
    android:required="false" />
```

Microsoft's Link to Windows app allows user to access Android apps right on PC



iPhone apps and iPad apps are available on the Mac App Store on Apple silicon Macs without modification



Class Code: 60 15 54

# Monetize Your App



# Monetize Your App

Monetizing your app is how you generate revenue from it



Developing a mobile app involves various costs:

1. Development costs
2. Tools and technologies
3. Marketing and promotion
4. Ongoing costs - updates, server hosting, customer support, etc
5. Legal and regulatory costs

# Monetize Your App



Factors that influence the monetization model:

1. Who are the users? Age, income, tech savviness
2. How valuable is your app? Does it solve a real problem?
3. What are your competitors doing?
4. How much does it cost to make and maintain the app?
5. How will it impact user experience?

# Monetize Your App

Who is paying?

Premium Apps

Freemium Apps

Subscription



End-users

Ads

E-Commerce

Rewarded Products

Service

Data Collection



Merchants

# Monetize Your App

Premium Apps

Paid

In-app Billing

Extensive Features

Narrow niche in the market



# Monetize Your App

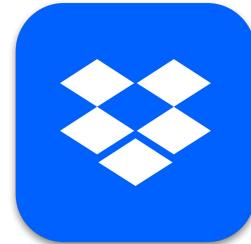
## Freemium Apps

Free

Uses In-app Billing

Digital goods

- Durable
- Consumable



# Monetize Your App

Subscription

Free trial

In-app billing

Subscription fee



# Monetize Your App

Ads

Free

AdMob + Google  
Mobile Ads SDK

Shows  
income-generating  
ads



# Alternative Monetization Options

E-Commerce

Free

B2C

Technology,  
logistics and  
payment solutions

Sales commissions  
+ Setup fees



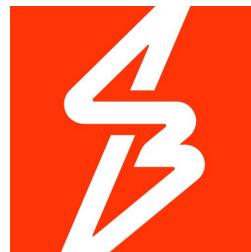
# Alternative Monetization Options

## Rewarded Products

Free

Share content on social media, perform simple tasks (e.g. scan QR code, answer survey and etc)

Get free stuff, earn rewards



Shopback



GigaGigs



Google Opinion Rewards

# Alternative Monetization Options

Service

Free

An extension of  
physical/online  
services



# Alternative Monetization Options

## Data Collection

Free

Partnership/Affiliate

Provide customer service, promotion, run a giveaway and etc



# Find Out More

Build and release for Android, <https://docs.flutter.dev/deployment/android>

Build and release for iOS, <https://docs.flutter.dev/deployment/ios>

Build and release for Huawei App Gallery,

<https://developer.huawei.com/consumer/en/appgallery/>