



TRƯỜNG ĐẠI HỌC FPT

Module 1 : Introduction to Flutter

Contents

- Overview of Flutter
 - What is Flutter - Why Flutter
 - Flutter architecture ?
- Dart Language Basics
 - Introduction to Dart and its role in Flutter
- Development Environments
 - DartPad, VS Code, Android Studio
- Setting Up Flutter
 - Installing SDK & Android Studio
 - Configuring Emulator
- Creating & Running Your First App

What is Flutter?

- Flutter is a UI toolkit developed by Google for building natively compiled apps from a single codebase.
- Supports Android, iOS, Web, macOS, Linux, and Windows.
- Written in Dart using its own rendering engine (Skia).
- Enables rapid UI development through reusable widgets.

Why Flutter?

Feature	Description
Fast Development	Hot Reload / Hot Restart
Beautiful UI	Material & Cupertino widgets
Single Codebase	Write once → deploy everywhere
Performance	Nearly native speed

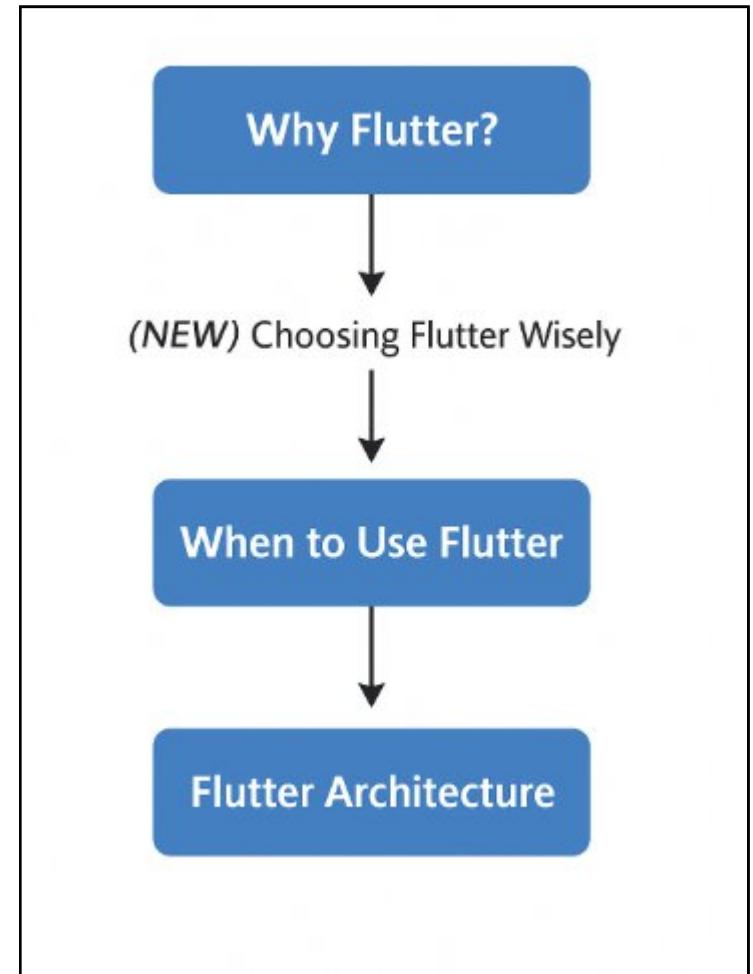
Choosing Flutter Wisely

From Features to Project Decisions

- Flutter offers many powerful features
- However, no technology fits all projects
- Choosing the right tool depends on project context

Key message:

- *Understanding when to use Flutter is as important as knowing why Flutter is powerful.*



When Is Flutter a Good Choice?

- Cross-platform applications with shared UI and logic
- MVPs, startups, and rapid prototyping
- Business applications (e-commerce, booking, dashboards)
- Apps with moderate use of device features
- Teams with limited native Android/iOS resources
- *Flutter is most effective when the goal is time to market, reduced development costs, and maintaining a single codebase.*

Flutter with Native Integration

Content:

- Flutter handles UI and business logic
- Native code handles platform-specific features
- Communication via Platform Channels

Examples of native integration:

- Camera, Bluetooth, NFC
- Payment SDKs
- OS-level services

Flutter doesn't eliminate native code, but integrates with it.

- When Flutter doesn't directly support a feature,
- we can write native Android code (Kotlin/Java)
- or native iOS code (Swift) and connect via Platform Channels.
- This is a very common model in real-world projects.

When Native Development Is Preferable

- Applications requiring deep OS customization
- High-performance games or real-time graphics
- Very low-level hardware access
- Strict platform-specific UI/UX requirements

Key takeaway:

- Flutter is powerful, but not always the optimal choice.

A Brief History

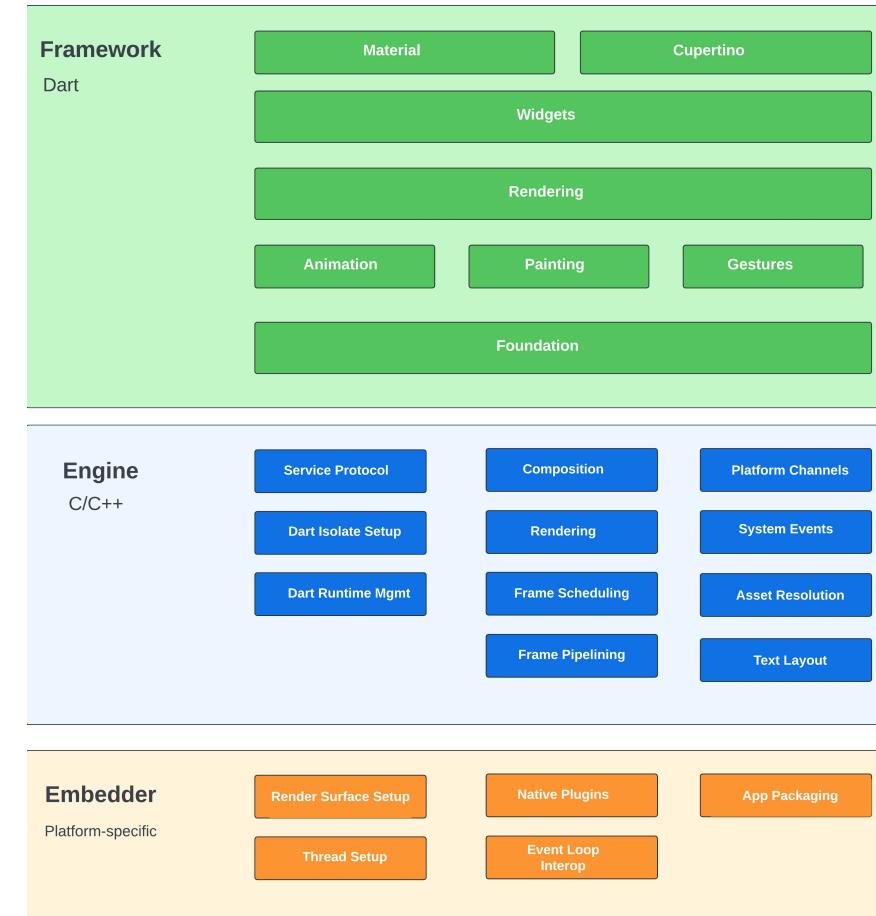
- 2015: Sky project.
- 2017: Flutter Alpha.
- 2018: Flutter 1.0.
- 2021: Flutter 2.0 adds Web/Desktop.
- 2023–2025: Flutter 3.x stable.

Flutter Architecture Overview

Three Layers:

- 1. Framework (Dart)
- 2. Engine (C++)
- 3. Embedder

Workflow: Widget Tree → Render Tree → Composited Scene → GPU Rendering



Framework Layer

- Widgets (Stateless, Stateful)
- Rendering and Layout system
- Foundation libraries
- Key concept: Everything in Flutter is a Widget.

Engine Layer - Embedder Layer

Engine Layer:

- Written in C++, powered by Skia.
- Handles rendering, accessibility, text layout.
- GPU acceleration ensures consistency.

Embedder Layer:

- Platform-specific integration.
- Handles input, lifecycle, window management.
- Custom embedders for Raspberry Pi.

Benefits of Flutter

- Performance: Compiles to ARM code.
- Consistency: Same UI across devices.
- Flexibility: Rich widget set.
- Flutter vs Others

Framework	Language	Output
Flutter	Dart	Native UI + Engine
React Native	JS	Bridges to Native
Native (Android/iOS)	Kotlin/Swift	Pure Native

Dart: Flutter's Programming Language

- Designed for UI creation.
- Combines AOT and JIT compilation.
- Supports null safety and strong typing.

Example:

```
void main() {  
    print('Welcome to Flutter!');  
}
```

Dart Core Features

- Object-Oriented.
- Functions as first-class objects.
- Null safety.
- Async/Await.

Flutter Development Environments Overview

- Flutter can be developed in three main environments, depending on the learner's goal and hardware setup:

Environment	Description	Ideal For
DartPad	Online editor, runs Flutter code in a browser	Quick demos, learning syntax
VS Code	Lightweight IDE with Flutter & Dart extensions	Fast prototyping, students
Android Studio	Full IDE with SDK, emulator, profiler	Complete mobile app projects

- Start students with DartPad → then move to VS Code / Android Studio.
- Emphasize that all tools compile to the same Flutter runtime.

Using DartPad (Online Environment)

- URL: <https://dartpad.dev>
- Key Features
 - No installation required
 - Run Flutter code instantly in browser
 - Supports core widgets (Scaffold, Text, Column, etc.)
 - Ideal for live demo and exercises

Installing the Flutter SDK

- Download from flutter.dev
- Extract the folder (e.g., C:\src\flutter or /Users/you/flutter)
- Add Flutter to the system PATH
- Verify installation:

```
01. flutter doctor
02. Doctor summary (to see all details, run flutter doctor -v):
03. [✓] Flutter (Channel stable, 3.19.6, on macOS 14.2.1 23C71 darwin-arm64, locale en-US)
04. [✓] Android toolchain - develop for Android devices (Android SDK version 34.0.0)
05. [✓] Xcode - develop for iOS and macOS (Xcode 15.1)
06. [✓] Chrome - develop for the web
07. [✓] Android Studio (version 2022.3)
08. [✓] VS Code (version 1.88.1)
09. [✓] Connected device (4 available)
10. [✓] Network resources
11.
12.
```

Figure 1.2: Flutter doctor

Installing Android Studio

Purpose: Android Studio provides IDE, SDK tools, and emulator to build and run Flutter apps.

Steps:

- Download and install Android Studio
- Install Flutter Plugin
 - Go to Preferences → Plugins → Flutter
 - Dart plugin installs automatically

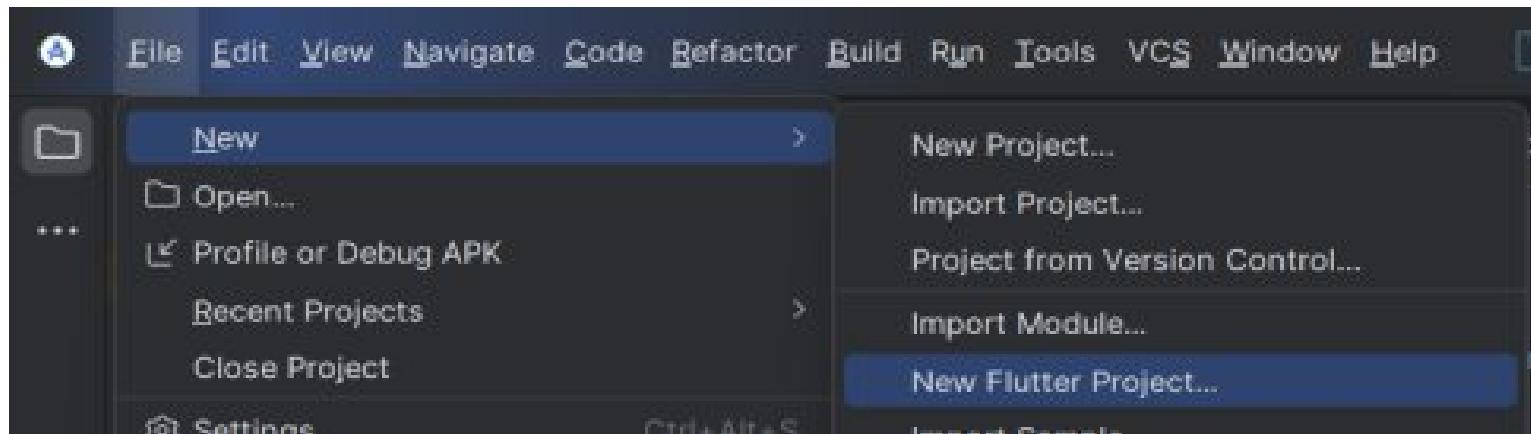
Note: Ensure Android SDK and command-line tools are installed.

Android Studio Interface Overview

Section	Description
Project View	Manage project folders and files
Editor	Write and edit Dart code
Run / Debug Toolbar	Build, run, and debug
Terminal	Access Flutter CLI commands
Device Manager	Manage emulators and devices
Logcat	View Android logs and debugging info

Create a New Flutter Project

- Click File → New → New Flutter Project
- Select Flutter SDK path
- Enter project name: movies
- Organization ID: com.fptu.movies
- Choose platforms: ✓ Android ✓ iOS ✓ Web ✓ macOS ✓ Windows



Flutter Project Structure

Folder	Description
lib/	Contains main Dart code
android/	Android platform-specific code
ios/	iOS platform-specific code
web/	Web app files
pubspec.yaml	Dependencies and assets configuration
test/	Unit test files

lib/main.dart is the app entry point.

Launching an Emulator

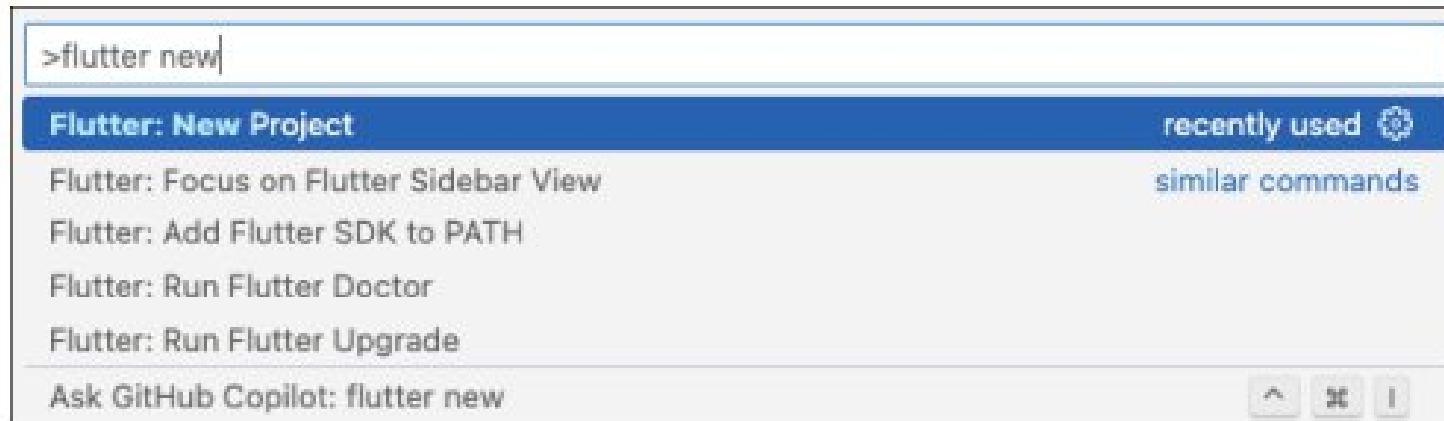
To create an emulator:

- Open Device Manager → Create Device
- Choose a Pixel model (e.g., Pixel 8)
- Select API 33+ (Android 13)
- Set Internal Storage: 4GB
- Note: Use Pixel 3a / API 30 for low-spec PCs.



Using VS Code (Lightweight IDE)

- **Setup Steps**
 - Install VS Code
 - Add Flutter & Dart extensions
 - Open a folder → Ctrl + Shift + P → Flutter: New Project
 - Run the app with F5 or terminal command flutter run



Your First Flutter App (“Hello World”)

Goal: Show how one simple Flutter program runs identically in all 3 development environments.

Run the App:

Environment	Steps
DartPad	Open dartpad.dev → New → Flutter → Paste code → ► Run
VS Code	Ctrl + Shift + P → Flutter: New Project → Replace lib/main.dart → flutter run or F5
Android Studio	<i>New Flutter Project</i> → Paste code → ► Run on emulator

Summary

- Understand what Flutter is and how it evolved.
- Describe the three-layer architecture: Framework → Engine → Embedder.
- Recognize the benefits of Flutter vs other frameworks.
- Understand Dart basics and its role in Flutter.
- Set up the development environments: DartPad, VS Code, Android Studio.
- Install Flutter SDK and verify setup using flutter doctor.
- Create and run your first app – “Hello World” – on three platforms.

References

- Mastering Flutter 2025 – Ch.1
- docs.flutter.dev
- <https://dartpad.dev>
- dart.dev/guides
- github.com/flutter/flutter

Module 2 - Dart Essentials

Contents

- Structure
- Data types
- Variables
- Collections
- Control Flow
- OOP
- Null Safety
- Async

What is Dart?

Concept

- Dart is a client-optimized, class-based language by Google.

Key Points

- Designed for UI frameworks (Flutter)
- Multi-platform: mobile (iOS/Android), web, and desktop applications
- Fast development: JIT
- Fast production: AOT
- Modern asynchronous model (Futures, async/await, Streams)

Notes: Relate Dart vs JavaScript vs Kotlin for context

Program Structure

- Every Dart program starts at the main() function.
- print() writes a line to the console.

```
void main() {  
    print("Hello Dart");  
}
```

⌚ Hello Dart

Data Types

Type	Description
Numbers	int (whole numbers), double (fractional)
String	Text enclosed in single or double quotes
bool	Boolean value: true or false
List	Ordered collection of elements
Set	Unordered collection of unique elements
Map	Key–value pairs
null	Represents absence of a value
Object	Base class for all Dart objects
enum	Fixed list of constant values
Iterable	Abstract collection type used in loops
Never	For functions that never return (always throw)
Dynamic	Can hold any type; resolved at runtime
Void	Indicates no return value

Data Types

- Core types: int, double, String, bool, List, Set, Map
- Strong typing with generics (List<int>, Map<String, int>)

```
int age = 20;  
double gpa = 3.75;  
String campus = "FPT HCM";  
bool pass = true;
```

Operators

Concept

- Operators allow you to perform mathematical, comparison, and logical operations.

Explanation

- Arithmetic: + - * / %
- Comparison: > < >= <= ==
- Logical: && || !

```
int a = 5, b = 2;  
print(a + b);  
print(a > b);
```

Variables & Memory Concept

- var = inferred type, dynamic = runtime type.
- final = runtime-constant (assigned once)
- const = compile-time constant

```
var name = "FPT University";
final year = 2024;
const pi = 3.14159;
dynamic x = 10; x = "Hello";
```

Strings & Interpolation

- Use \$var or \${expr} to interpolate values.

```
String name = "FPT University";
print("Hello $name");
print("Length: ${name.length}");
```

Hello FPT University
Length: 16

Collections (List, Set, Map)

- List: ordered • Set: unique • Map: key-value

```
List<int> nums = [1,2,3];
Set<int> s = {1,2,2,3};
var user = {"id":1, "name":"Nam"};
print(user["name"]);
```

- Output:

{1,2,3}	
	Nam

Control Flow: if / else - switch

- If/else: Conditional branching with boolean expressions.

```
var day = "Mon";  
  
switch(day) {  
    case "Mon": print("Start"); break;  
  
    default: print("Other");
```

- Switch is clearer than many if-else for discrete cases.

```
int score = 80;  
  
if(score >= 50) print("Pass");  
else print("Fail");
```

Loops: for-in & forEach

- Classic counted loop.

```
for(int i=0; i<3; i++) {  
    print(i);  
}
```

- for-in: iterate collections • forEach: functional style

```
for(var x in [1,2,3]) print(x);  
  
[1,2,3].forEach((e)=>print(e));
```

Functions: basics - arrow syntax

- Typed params & return values.

```
int add(int a,int b){  
    return a+b;  
}  
  
print(add(2,3));
```

- Short form for single-expression functions.

```
int add(int a,int b)=>a+b;  
  
print(add(2,3));
```

Functions — optional & named params

- Use {} for named optionals; provide defaults.

```
void greet({String name = "Guest"}) { print(name); }
```

```
greet(); // Guest
```

```
greet(name: "Minh"); // Minh
```

- Output:

Guest
Minh

Intro to OOP

Concept

- Dart fully supports OOP

Key ideas

- Class / Object
- Constructor
- Inheritance
- Polymorphism
- Method Override

Notes

OOP will be the foundation when building the Flutter widget tree UI.

Classes & Objects

- Class defines structure; object is an instance.

```
class Car {  
    String brand;  
    Car(this.brand);  
    void start(){ print("Started: $brand"); }  
}  
  
var c = Car("Toyota");  
c.start();
```

Started: Toyota

Constructors - default & named

- Default, named, and redirecting constructors.

```
class Point {  
    int x,y;  
    Point(this.x,this.y);  
    Point.origin(): x=0, y=0;  
}  
  
var p = Point.origin();
```

Inheritance & Override

- Use **extends** to inherit and **@override** to change behavior.

```
class Animal {  
    void sound()=>"Some";  
}  
  
class Dog extends Animal {  
    @override void sound()=>"Woof";  
}  
  
Dog()..sound();
```

Null Safety - operators ?, ??, !

- Dart enforces null-safety at compile-time to prevent null reference errors.

Operator	Meaning
?	Declares a nullable variable
??	Default fallback if null
!	Force non-null assertion

- Example:

```
String? name;           // nullable variable
print(name);           // prints null

print(name ?? "Guest"); // default value if null

name = "Minh";
print(name!.length);   // force non-null after assignment
```

Collections - map/filter

Concept: Functional operations over iterables (e.g.,
map, where)

- map: transform each element → new Iterable
- where (filter): keep elements that satisfy a predicate
- Often chained; lazy until consumed (toList,
foreach, etc.)

Explanation

- Works on any Iterable<T> (e.g., List, Set)
- map preserves length, transforms type
- where preserves type, may change length
- Use toList() or toSet() to materialize results

```
final numbers = [1, 2, 3, 4, 5];

// map: square each number
final squares = numbers.map((n) => n * 2);

// filter: keep even numbers
final evens = numbers.where((n) => n.isEven);

// chain: square then keep > 5
final big = numbers.map((n) => n * 2).where((n) => n > 5);

// materialize
print(squares.toList());
print(evens.toList());
print(big.toList());

// map can change type
final labels = numbers.map((n) => 'Item $n').toList();
print(labels);
```

Exceptions — try/catch/finally

Concept: Handle runtime exceptions without crashing the application.

- try: the risky block
- catch / on: handle exceptions
- finally: always runs (cleanup)
- rethrow to propagate after logging

Explanation

- Use on SpecificException to target types
- catch (e, s) captures error + stack trace
- Prefer throwing custom exceptions for domain errors

Example:

```
int parsePositiveInt(String s) {  
    final value = int.parse(s); // may throw FormatException  
    if (value < 0) throw ArgumentError('Must be >= 0');  
    return value;  
}  
  
void main() {  
    try {  
        print(parsePositiveInt("12"));  
        print(parsePositiveInt("-3")); // throws ArgumentError  
    } on FormatException catch (e, stack) {  
        print("Format error: $e");  
    } on ArgumentError catch (e) {  
        print("Argument error: ${e.message}");  
    } catch (e, stack) {  
        print("Unknown error: $e");  
        // rethrow; // (optional) bubble up  
    } finally {  
        print("Done parsing.");  
    }  
}
```

Async — Future & async/await

- `async` marks an `async` function; `await` pauses until `Future` completes.

```
Future<String> fetch() async {
    await Future.delayed(Duration(seconds:1));
    return "Done";
}

void main() async {
    print(await fetch());
}
```

Streams — sequence of async values

Exam:

- A Stream emits a sequence of asynchronous values over time.
- Single-subscription (default): one listener, sequential consumption.
- Broadcast: multiple listeners, events shared.

Explanation:

- Subscribe with `listen`, cancel with `subscription.cancel()`.
- Transform with `map` / `where` / `take` / `debounce` (package `).
- Consume with `await for` inside `async` functions.

```
// Example 1: periodic stream (single-subscription)
final Stream<int> stream = Stream.periodic(const Duration(milliseconds: 400), (i) => i)
    .take(5); // 0..4 then complete

stream.listen(
    (value) => print('Value: $value'),
    onDone: () => print('Done'),
);
}

// Example 2: transform & filter
final Stream<int> evens = stream.where((n) => n.isEven).map((n) => n * 10);
evens.listen((v) => print('Even*10: $v'));
```

Mini-lab - Student list

- Task: create Student class, store in List, print, then add async delay to simulate loading.

```
class Student { String name; Student(this.name);}

void main() async {
    var list = [Student("An"), Student("Binh")];
    await Future.delayed(Duration(seconds:1));
    list.forEach((s)=>print(s.name));
}
```

Summary

We covered:

- Dart structure
- data types -Variables
- Collections
- Control flow
- Functions
- OOP
- Null safety
- Async programming and streams.

References

- Mastering Flutter (2025), Chapter 2
- docs.flutter.dev
- dart.dev/tools/dartpad

Module 3 – Advanced Dart for Flutter Development

Overview

What you will learn

- Deep OOP concepts used in Flutter internals
- Generic programming for reusable components
- How async and event loops really work in Dart
- Stream architecture behind StreamBuilder
- Repository pattern to prepare for state management

Learning Roadmap

Advanced Language Features

- Abstract classes & implicit interfaces
- Mixins
- Factory constructors
- Generics

Async mastery

- Microtask queue vs Event queue
- Future chaining vs async/await
- Streams, broadcast, async*, yield, transformers

Architecture foundation

- Model / Repository separation
- Preparing for BLoC / Riverpod concepts

Abstract Classes

- Abstract classes define structure without full behavior.
- They force subclasses to implement functions.
- In Flutter, `State<T>` is an abstract class - Widgets override methods to define behavior.
- Example:

```
abstract class Shape {  
    double area(); // no body = must be implemented  
}  
  
class Circle extends Shape {  
    final double r;  
    Circle(this.r);  
    double area() => 3.14 * r * r;  
}
```

Implicit Interfaces

- Every class in Dart automatically defines an interface.
implements forces full override.
- Example:

```
class Bird {  
    void fly() => print("Bird flying");  
}  
  
class Eagle implements Bird {  
    void fly() => print("Eagle gliding");  
}
```

Extends vs Implements

Keyword	Meaning	Use case
extends	Inherit behavior	Modify or extend existing class
implements	Only signature, no behavior	Define contract, custom behavior
with	Mix in code units	Share code reusable blocks

Flutter real examples

- StatelessWidget extends Widget
- AnimationController implements Listenable

Mixins

- Mixins = shared behavior without inheritance chain
- Example:

```
mixin Logger {  
    void log(String msg) => print("[LOG] $msg");  
}  
  
class AuthService with Logger {  
    void login() { log("login called"); }  
}
```

- **Flutter uses mixins for animation, scrolling, lifecycle support.**

Mixin Constraints

- Limit mixin usage to specific parent type → ensures capability exists.
- Example:

```
class Pet { String name; Pet(this.name); }

mixin CanBark on Pet {

    void bark() => print("$name says woof!");

}

class Dog extends Pet with CanBark {

    Dog():super("Buddy");

}
```

Factory Constructors

- Factory = control over instance creation, instead of always returning new object.
- Control object creation, caching, JSON parse.
- Example:

```
class User {  
    final String name;  
    factory User.fromJson(Map j) => User._(j['name']);  
    User._(this.name); // private  
}
```

- Flutter uses factories for theme, text styles, colors, widgets.

Factory Use Cases

Scenario	Example
Return existing instance	Singleton services
Convert format	.fromJson()
Check type before creation	Database adapters
Async creation	Load asset before building object

Generics Overview

- Generics = templates for types — write once, reuse safely.
Dart: Future<T>, Stream<T>, List<T>, Map<K, V>
- Example:

```
List<String> students = ["Anna", "Ben"];
Map<String, int> scores = {"Anna": 95};
```
- Why important
 - Compile-time type safety
 - Avoids dynamic misuse
 - Used everywhere in Flutter (widgets, async, forms)

Generic Class Example

- One class → many data types safely
Like <String> dropdowns in Flutter or <Widget> lists

- Example:

```
class Box<T> {  
    T value;  
    Box(this.value);  
    void show() => print("Value = $value");  
}  
void main() => Box<int>(10).show();
```

Generic Constraints

- Force generic to be subclass of certain type
- Example:

```
abstract class Animal { String sound(); }

class Cat extends Animal { sound() => "Meow"; }

class Box<T extends Animal> {

    T pet;

    Box(this.pet);

    void play() => print(pet.sound());

}
```

- Used in animation system & provider patterns

Collection-if / for / spread

- Dart provides functional collection building
- Example:

```
var base = [1,2,3];
var list = [ ...base,      // spread
            if(true) 99, // conditional insert
            for (var x in base) x * 10 // loop insert
];
print(list);
```

- Output: [1, 2, 3, 99, 10, 20, 30]

Custom Exceptions

Purpose

- Domain-specific error context (instead of raw text strings)
- Example

```
class LoginException implements Exception {  
    final String msg;  
    LoginException(this.msg);  
}
```

- Throw meaningful exceptions in repositories → easy to debug UI logic.

Event Loop & Microtasks

- Understanding async order = smoother UI.
- Dart async engine has two queues:

Queue	Priority	Handles
Microtask queue	▲ Highest	Immediate internal ops
Event queue	Normal	Futures, UI events, timers

- Microtasks run before event queue.

ScheduleMicrotask()

- Understanding queue = smooth animations + predictable async
- Example

```
import 'dart:async';
void main() {
    print("A");
    scheduleMicrotask(() => print("micro"));
    Future(() => print("future"));
    print("B");
}
```

Output ↴ A, B, micro, future

Future Chaining

- Chain async steps without nesting
- Use when you don't need variables or async/await blocks.
- Example:

```
Future(() => 1)  
    .then((v) => v + 1)  
    .then((v) => print(v));
```

Output ↴ 2

Stream Generators - Controller & broadcast

- *async* and *yield*

```
Stream<int> nums() async* {  
    for(int i=1;i<=3;i++) yield i;  
}
```

- Controller & broadcast

```
var c = StreamController.broadcast();  
c.stream.listen((v)=>print("A:$v"));  
c.stream.listen((v)=>print("B:$v"));  
c..add(1)..add(2);
```

Output		A:1 B:1
		A:2 B:2

Repository Pattern Intro - Example (Future)

- Separate data fetching from presentation/UI
UI -> Repository -> Data (API/DB)
- Testable, Maintainable, Works with Bloc, Riverpod, Provider
- Example (Future): UI calls: await repo.getUser()

```
class Repo {  
    Future<String> getUser() async {  
        await Future.delayed(Duration(milliseconds:300));  
        return "Anna";  
    }  
}
```

UI calls: await repo.getUser()

Repository Stream Example

- Example:

```
Stream<int> counter() async* {  
    for(int i=1;i<=3;i++){  
        await Future.delayed(Duration(milliseconds:300));  
        yield i;  
    }  
}
```

- Flutter widget

```
StreamBuilder(stream: repo.counter(), builder:... )
```

- Foundation for real-time apps

Putting It Together - Performance Best Practices

- Architecture flow

Model -> Repo -> Stream/Future -> Widgets rebuild

- Performance Best Practices
 - Avoid blocking main isolate (no heavy loops in UI)
 - Use const widgets to reduce rebuild cost
 - Use async functions for I/O
 - Use Isolates for heavy CPU tasks (later module)

Practice Task

Goal: Build data layer simulation using Dart

- Create Product model

- Create repository returning:

Future<List<Product>>

Stream<Product> (live add)

- Print to console

Summary

- Advanced OOP
- Generics
- Collections
- Exceptions
- Async & Streams
- Repo pattern

References

- Mastering Flutter (2025), Chapter 2,3,6
- docs.flutter.dev
- dart.dev/tools/dartpad

Module 4 - Widgets & Basic Screens

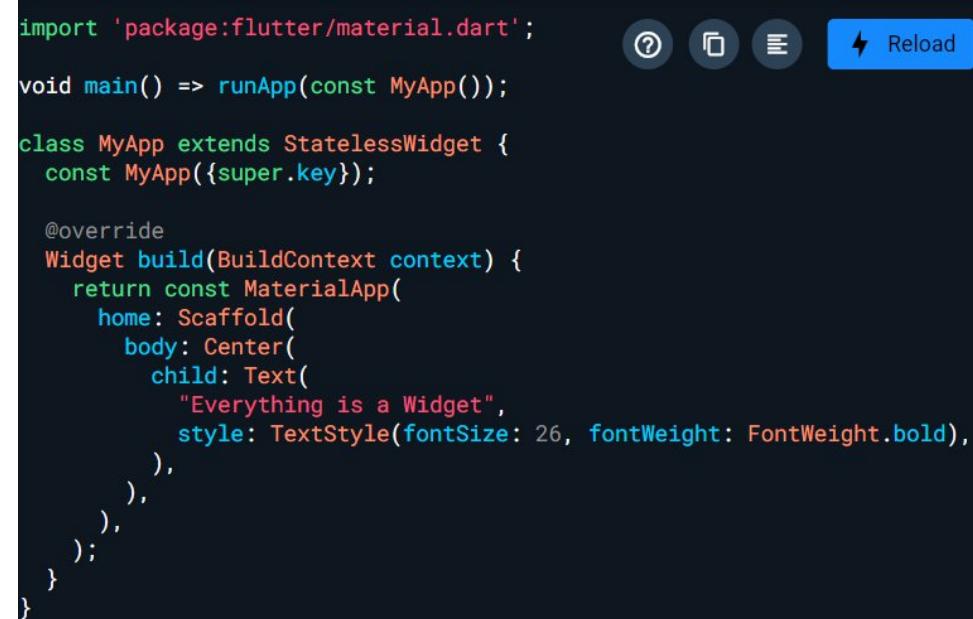
Contents

- Introduction: Everything is a Widget
- Core Widgets (Text, Image, Icon, Card, ListTile)
- Input Widgets (Slider, Switch, Radio, Pickers)
- Layout Basics (Column, Row, ListView, Padding)
- App Structure (Scaffold, Theme, Navigation)
- Design Polish (Spacing, Consistency)
- Common Errors & Fixes
- Practice Task & Summary

Everything is a Widget

- In Flutter, the UI is built by composing widgets.
- Widgets describe structure (layout), appearance (style), and behavior (interaction).
- Every screen, every button, every text = widget

Example run with Dartpad



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

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

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

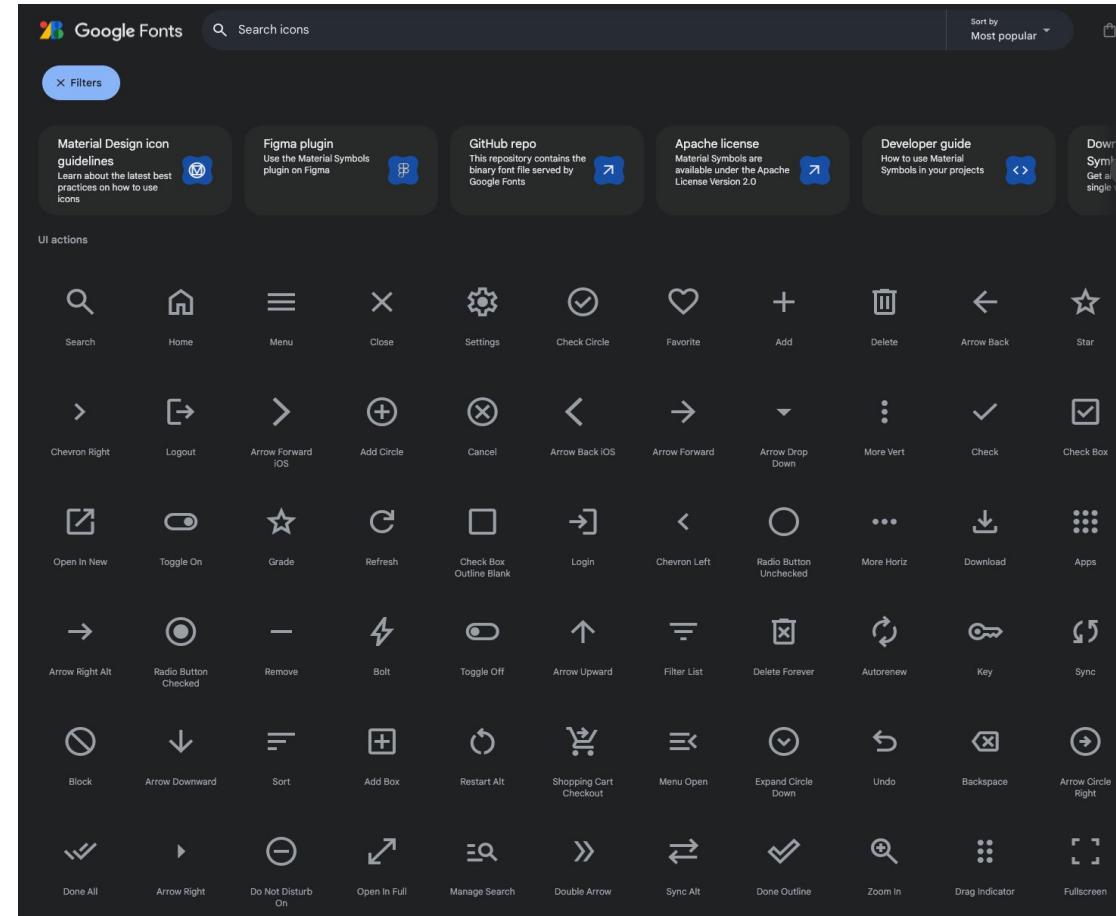
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: Scaffold(
        body: Center(
          child: Text(
            "Everything is a Widget",
            style: TextStyle(fontSize: 26, fontWeight: FontWeight.bold),
          ),
        ),
      ),
    );
  }
}
```

Output Preview:

Everything is a Widget

Material Icons

- Icons in Flutter come from Material Design Icons library. They give consistent UI across devices.
- When to use
 - Navigation buttons
 - Action buttons (add, delete)
 - Status indicators



Example Material Icons

- Example: Test in DartPad:
<https://dartpad.dev/?id=icon-widget>
- Output 

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

void main() => runApp(const IconDemo());

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: Scaffold(
        body: Center(
          child: Icon(
            Icons.home,
            size: 50,
            color: Colors.blue,
          ),
        ),
      ),
    );
  }
}
```

DatePicker

- Material date picker allows users to select a date from a calendar UI
- Real uses
 - Booking apps
 - Calendar tasks
 - Form inputs (date of birth, appointment)

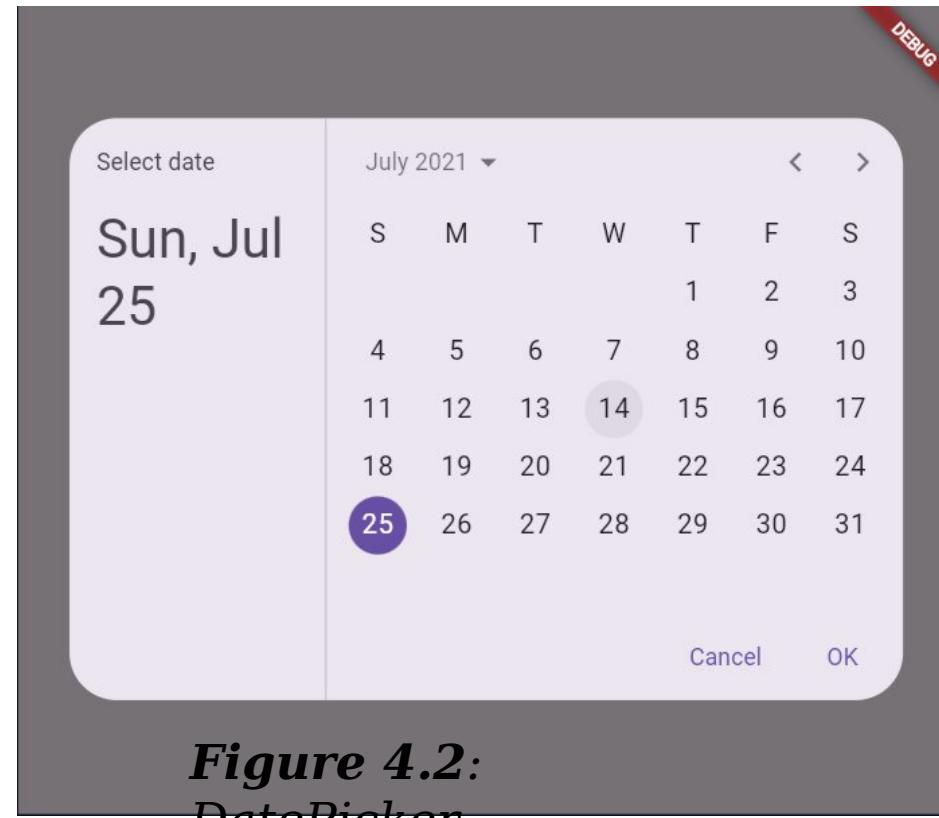


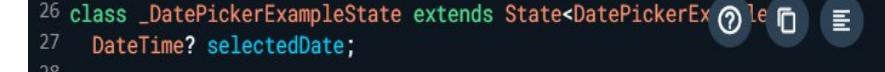
Figure 4.2:
DatePicker

Code Example DatePicker

```
1 import 'package:flutter/material.dart';
2
3 void main() => runApp(const DatePickerApp());
4
5 class DatePickerApp extends StatelessWidget {
6   const DatePickerApp({super.key});
7
8   @override
9   Widget build(BuildContext context) {
10     return MaterialApp(
11       home: Scaffold(
12         appBar: AppBar(title: const Text("Date Picker Demo")),
13         body: const DatePickerExample(),
14       ),
15     );
16   }
17 }
18
19 class DatePickerExample extends StatefulWidget {
20   const DatePickerExample({super.key});
21
22   @override
23   State<DatePickerExample> createState() => _DatePickerExampleState();
24 }
```

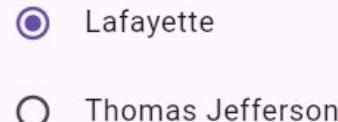


```
26 class _DatePickerExampleState extends State<DatePickerExample> {
27   DateTime? selectedDate;
28
29   Future<void> pickDate() async {
30     final date = await showDatePicker(
31       context: context,
32       initialDate: DateTime.now(),
33       firstDate: DateTime(2000),
34       lastDate: DateTime(2100),
35     );
36     if (date != null) setState(() => selectedDate = date);
37   }
38
39   @override
40   Widget build(BuildContext context) {
41     return Center(
42       child: Column(
43         mainAxisAlignment: MainAxisAlignment.center,
44         children: [
45           Text(selectedDate == null
46               ? "No date selected"
47               : "Selected: ${selectedDate!.toLocal().split(' ')[0]}"),
48           const SizedBox(height: 16),
49           ElevatedButton(
50             onPressed: pickDate,
51             child: const Text("Pick Date"),
52           ),
53         ],
54       ),
55     );
56 }
```



RadioListTile (Single Choice)

- RadioListTile lets users pick one option from a group.

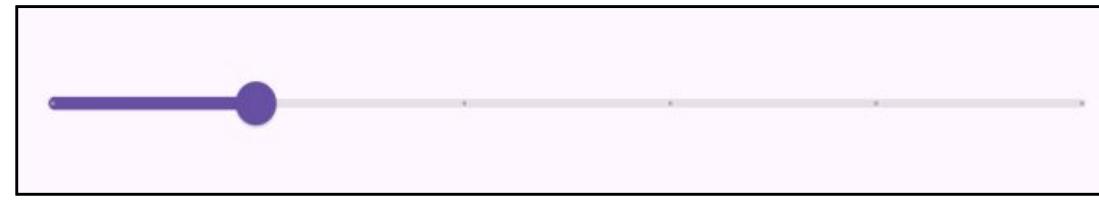


- **Structure (template)**

```
RadioListTile<T>(  
    title: Text('Label'),  
    subtitle: Text('Optional'),  
    value: /* T */,  
    groupValue: /* T? */,  
    onChanged: (T? v) { /* setState(() => groupValue = v); */ },  
    secondary: Icon(Icons.info), // optional trailing icon  
    dense: false,           // compact style  
    selected: false,        // highlight when selected  
)
```

Slider (Continuous Value)

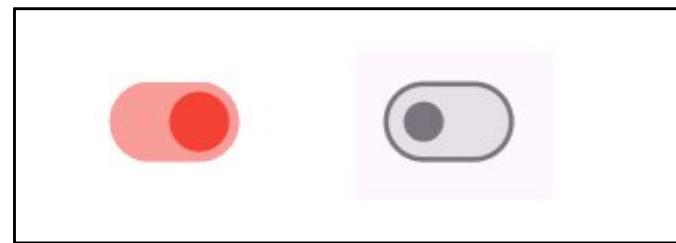
- Slider allows users to adjust a value continuously between a range.
- Structure (template) ☺



```
Slider(  
    value: /* double */,  
    onChanged: (double v) { /* setState(() => value = v); */ },  
    min: 0.0,  
    max: 1.0,  
    divisions: null, // e.g. 10 for discrete steps  
    label: null, // shows tooltip when divisions != null  
    activeColor: null,  
    inactiveColor: null,  
)
```

Switch (Boolean Toggle)

- Switch toggles a boolean value (ON/OFF).



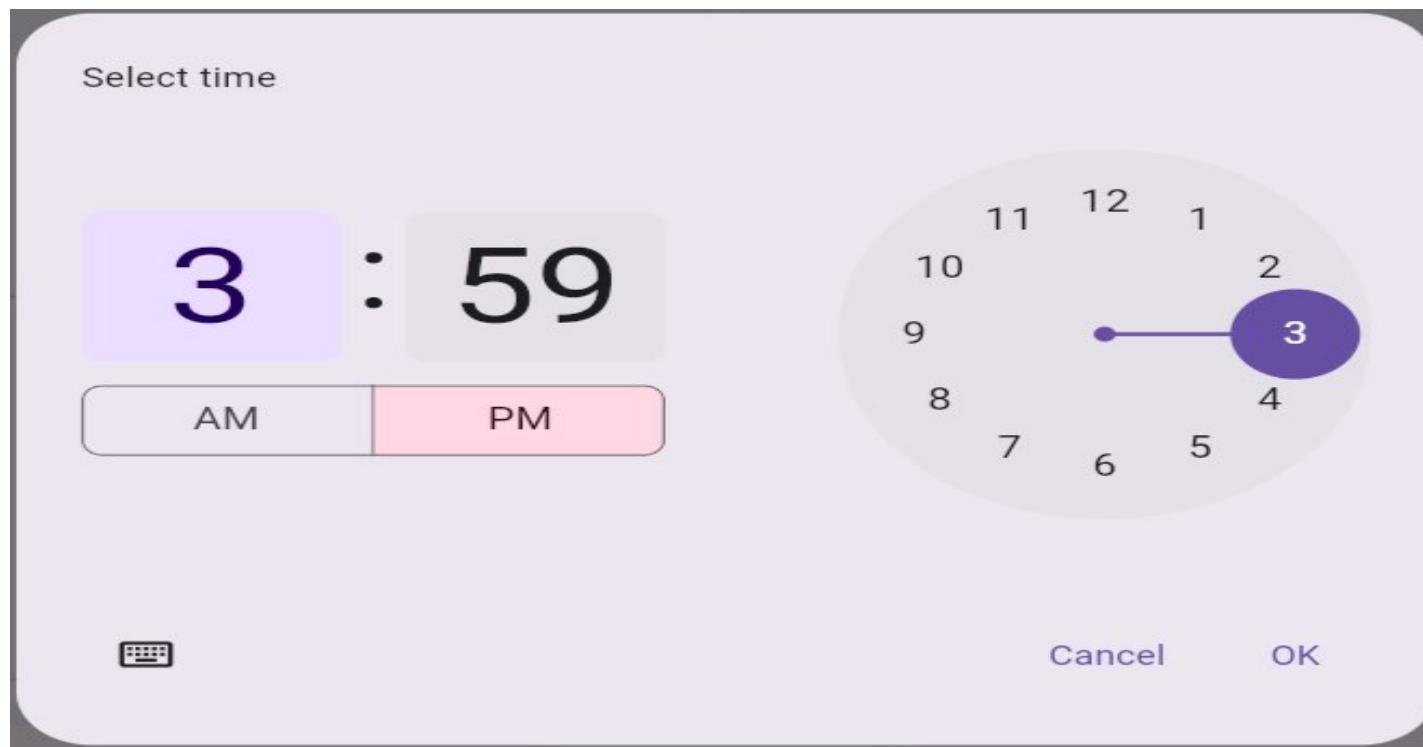
- Use: Enable notifications, dark mode, or toggling options.

- Structure (template)

```
Switch(  
    value: /* bool */,  
    onChanged: (bool b) { /* setState(() => value = b); */ },  
    activeColor: null,  
    activeTrackColor: null,  
    inactiveThumbColor: null,  
    inactiveTrackColor: null,  
    thumbIcon: null, // Material 3  
)
```

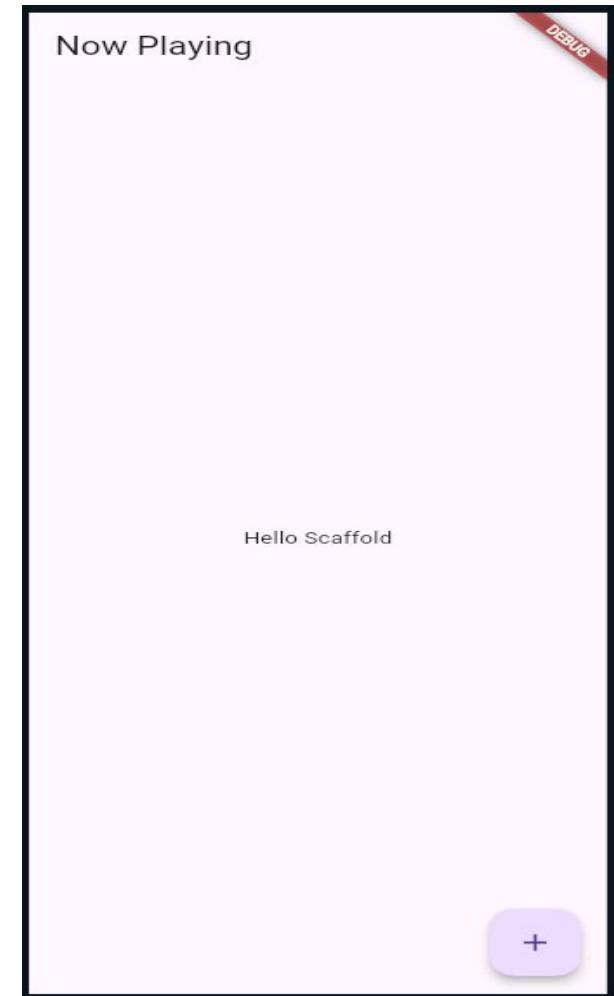
TimePicker (dialog)

- Concept: show TimePicker opens a Material-style time selection dialog.
- Use: Appointment scheduling, alarms, or showtime selection.



Scaffold (First Screen)

- Concept: Scaffold is the backbone of a Material screen.
- It provides AppBar, Body, FloatingActionButton, and other structural elements.
- Use: Every major screen in your app.



Theme & Dark Mode

- Concept: ThemeData defines global color, typography, and shapes across your app.
- Use: darkTheme and themeMode to switch between modes.

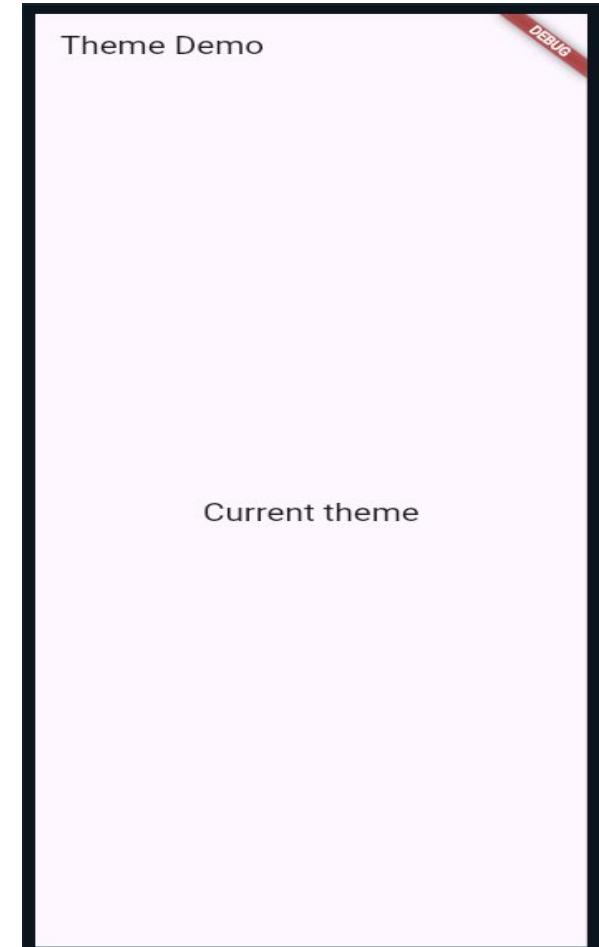
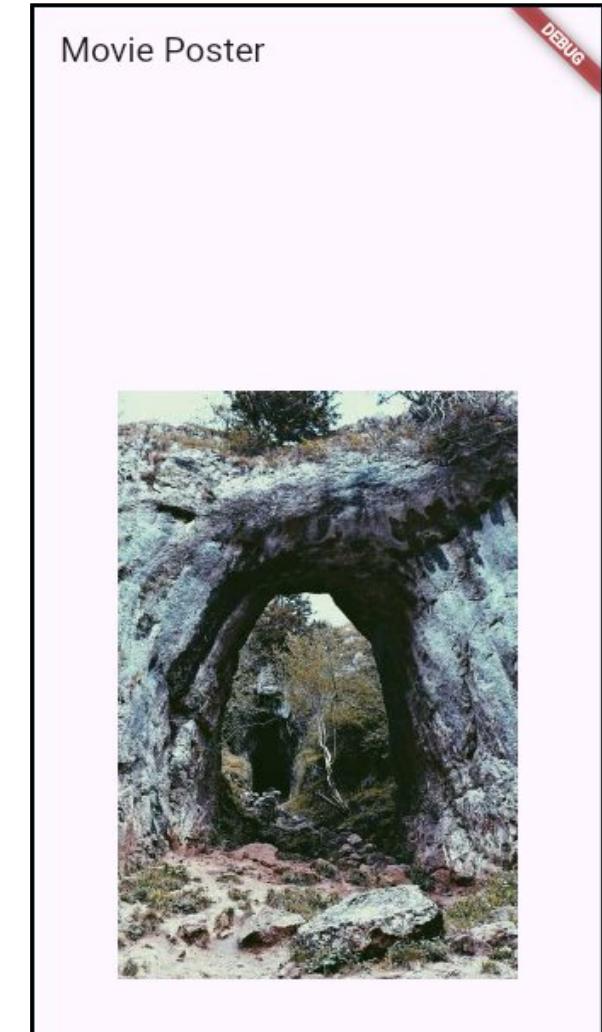


Image (Poster Detail)

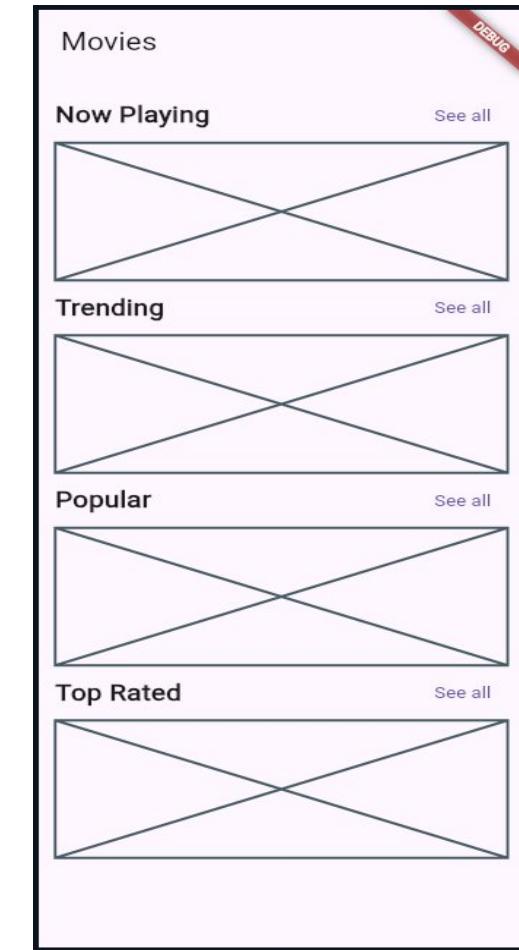
- Concept: Image.network or Image.asset displays pictures with scaling via BoxFit.
- Use Case: Used for posters, banners, or product images.

```
1 import 'package:flutter/material.dart';
2 void main() => runApp(const PosterApp());
3 class PosterApp extends StatelessWidget { const PosterApp({super.key});
4   @override Widget build(BuildContext context) =>
5     const MaterialApp(home: PosterScreen());
6 }
7 class PosterScreen extends StatelessWidget { const PosterScreen({super.key});
8   @override Widget build(BuildContext context) {
9     return Scaffold(
10       appBar: AppBar(title: const Text('Movie Poster')),
11       body: Center(
12         child: Image.network(
13           'https://picsum.photos/400/600',
14           fit: BoxFit.cover, width: 250, height: 380,
15         ),
16       ),
17     );
18   }
19 }
```



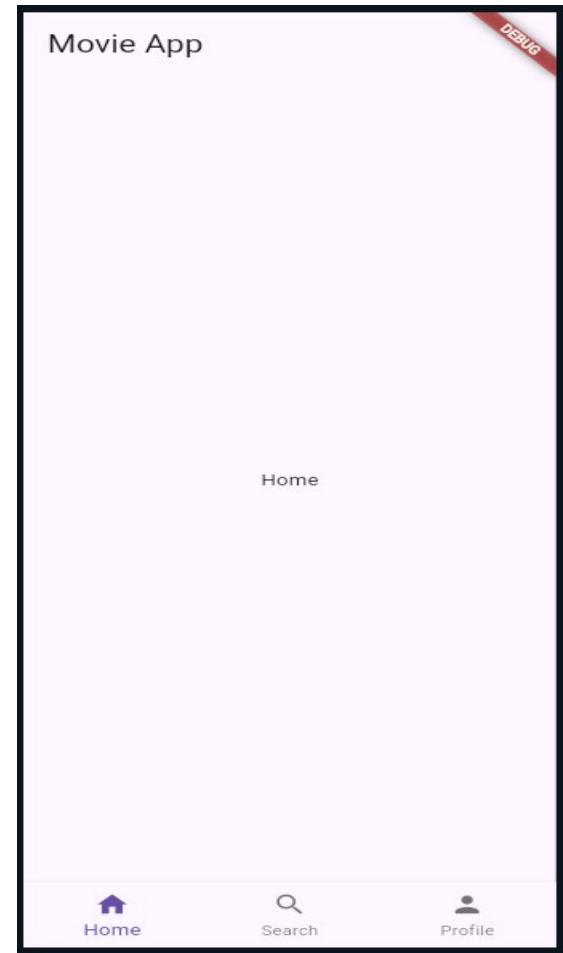
Sectioned Home Screen (Layout)

- Concept: Column, Padding, and ListView build vertical UI sections.
- Common in movie, product, or news apps.



Bottom Navigation Bar

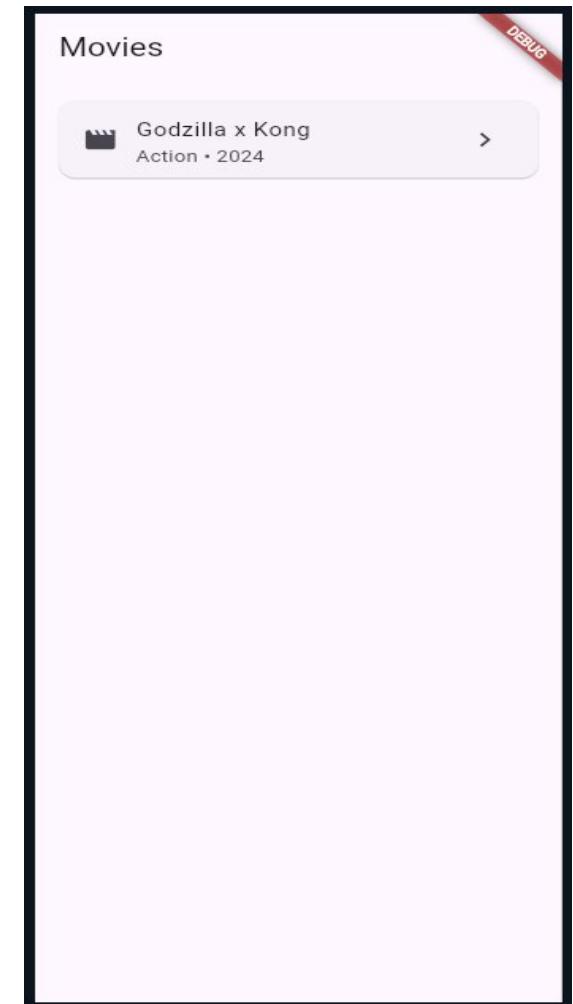
- Concept: Displays 2–5 navigation tabs at the bottom.
- Used for quick switching between app views.



Card + ListTile

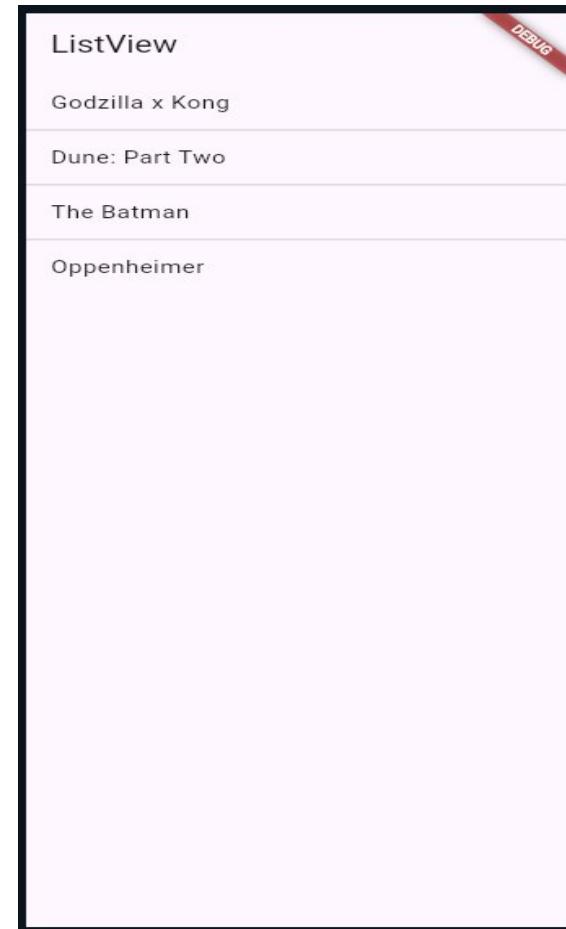
- Card + ListTile = material pattern for displaying concise information.

```
import 'package:flutter/material.dart';
void main() => runApp(const CardDemo());
class CardDemo extends StatelessWidget { const CardDemo({super.key});
  @override Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text('Movies')),
        body: ListView(
          padding: const EdgeInsets.all(12),
          children: const [
            Card(
              child: ListTile(
                leading: Icon(Icons.movie),
                title: Text('Godzilla x Kong'),
                subtitle: Text('Action • 2024'),
                trailing: Icon(Icons chevron_right),
              ),
            ),
            Card(
              child: ListTile(
                leading: Icon(Icons.movie),
                title: Text('Godzilla x Kong'),
                subtitle: Text('Action • 2024'),
                trailing: Icon(Icons chevron_right),
              ),
            ),
            Card(
              child: ListTile(
                leading: Icon(Icons.movie),
                title: Text('Godzilla x Kong'),
                subtitle: Text('Action • 2024'),
                trailing: Icon(Icons chevron_right),
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```



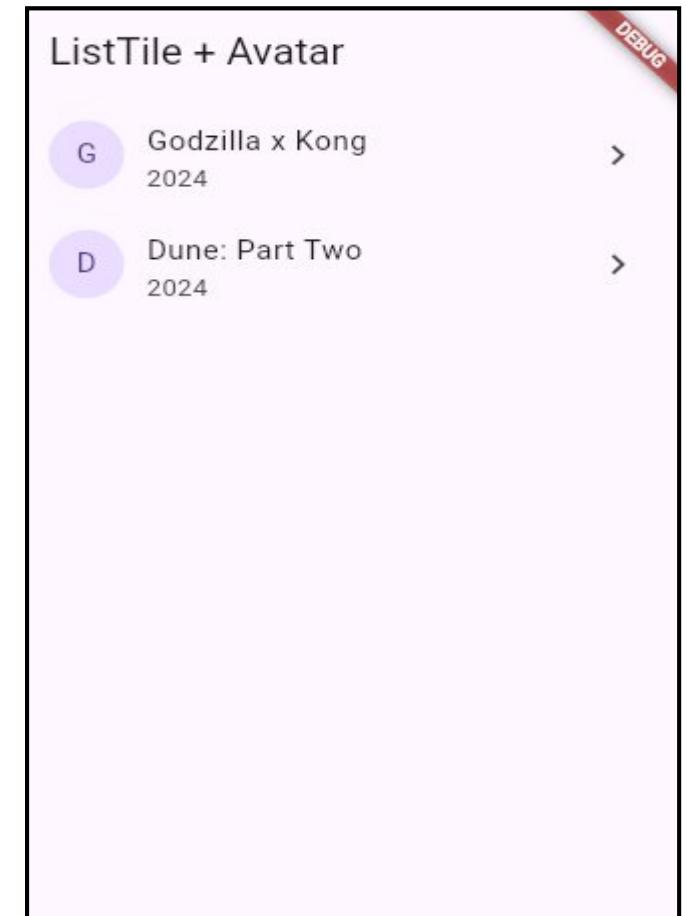
ListView Basics (Movie Titles)

- ListView creates a scrollable column; use separated or builder for long lists.
- Use Case: Movie titles, product names, chat items.



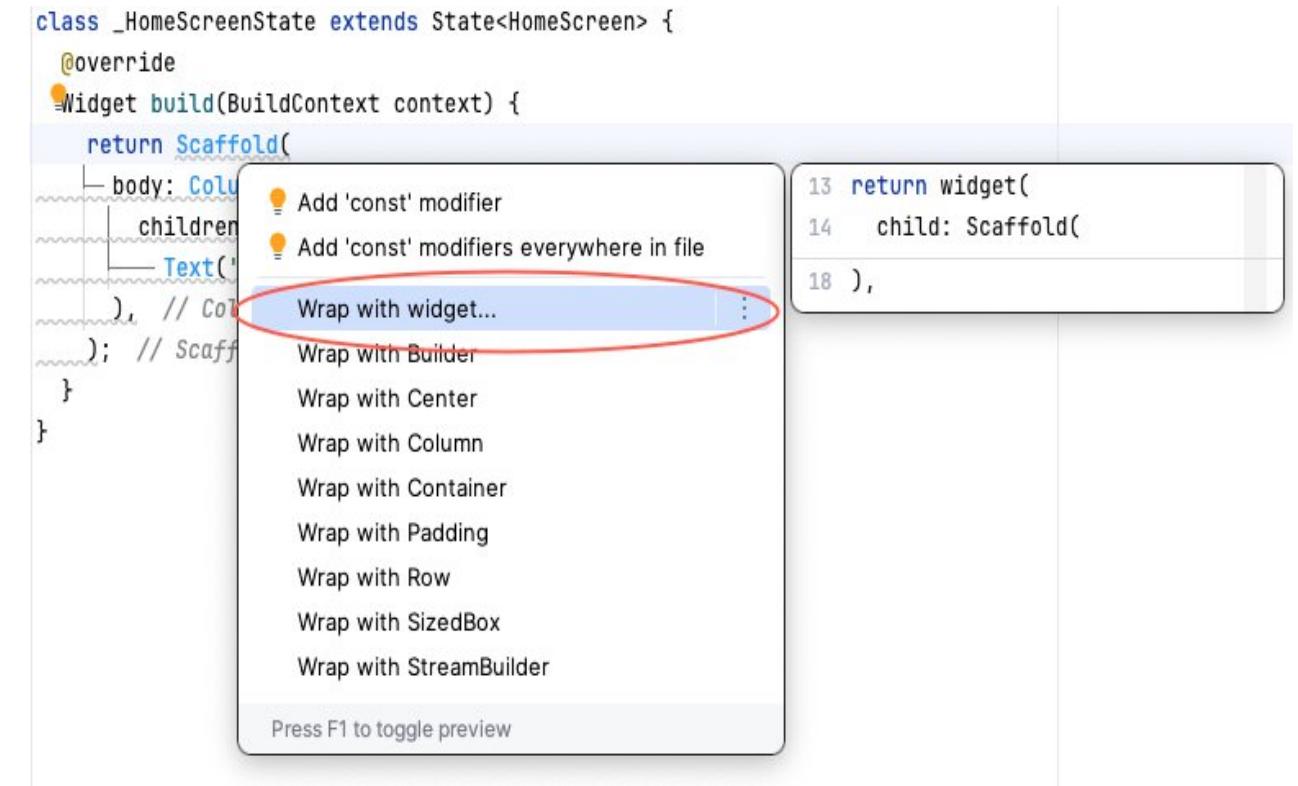
ListTile + CircleAvatar Widgets

- Concept: ListTile provides a standard row layout; CircleAvatar gives a modern leading visual.
- Use Case: Contact list, movie list, product list.



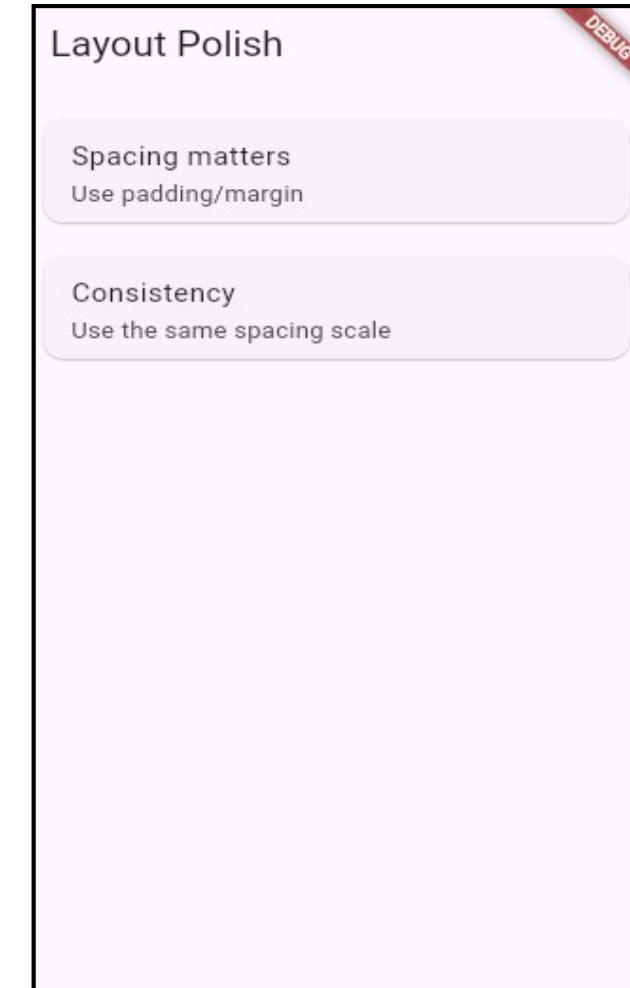
VS Code “Wrap with Widget”

- Editor-assisted refactor: wrap any widget with Padding, Center, Expanded, etc.
- Use Case: Speed up layout composition; enforce consistent spacing.



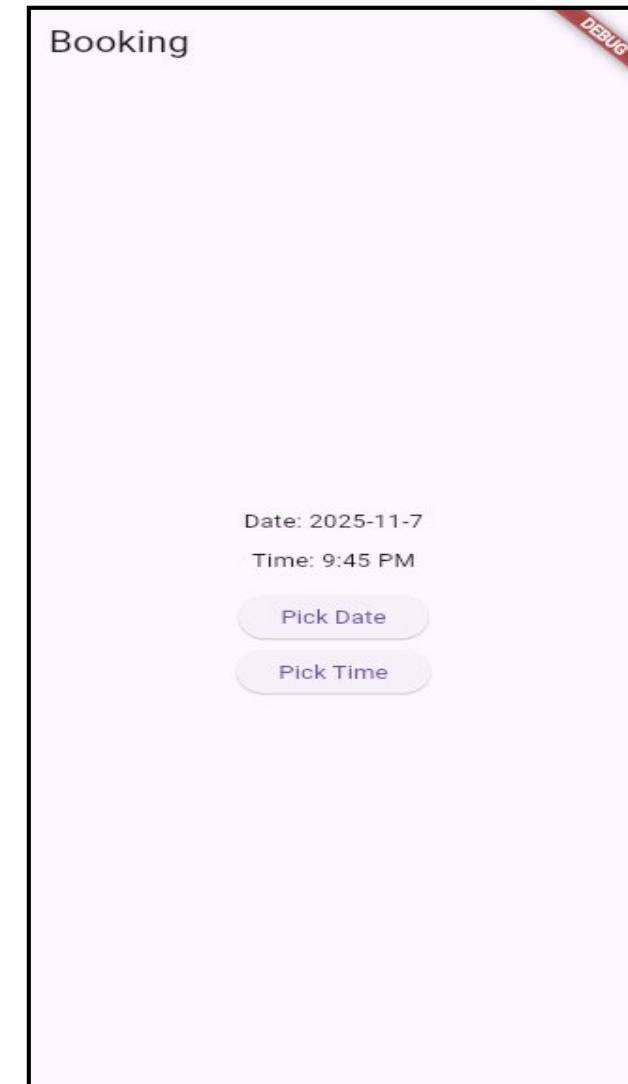
Layout Polish (Padding & SizedBox)

- Concept: Good spacing improves readability and perceived quality.
- Use Case: Apply consistent spacing scale across the app.



Date & Time Pickers (Combined)

- Concept
Use both pickers to build booking workflows.
- Use Case
Movie ticket booking: select both date and time



Common Errors & Fixes

Typical issues when building basic UI.

Checklist

- ListView inside Column → wrap with Expanded
- Overflow on small screens → use Flexible / SingleChildScrollView
- UI not updating → forgot setState()
- Picker call fails → make sure you call with a valid BuildContext

Practice Task

Task

- Build a movie list: title, year
- Use ListView.builder + ListTile
- onTap → show Snackbar(title)
- Bonus: replace avatar with an emoji or poster image

Summary

You Learned:

- The “Everything is a Widget” concept in Flutter
- Building screens using Scaffold, AppBar, and Body
- Using core widgets: Image, Card, ListTile, ListView
- Managing interactivity with Switch, Slider, RadioListTile, and Pickers
- Applying ThemeData and handling Dark Mode
- Improving design with Padding, SizedBox, and consistent spacing
- Identifying and fixing common layout errors

References

- Mastering Flutter 2025 – Ch.4
- docs.flutter.dev
- <https://dartpad.dev>

Module 5 - Navigation & State Management in Flutter

Contents

- Understand how Flutter manages navigation between screens.
- Learn both Navigator 1.0 (imperative) and Navigator 2.0 (declarative).
- Explore state management with setState, InheritedWidget, and Provider.
- Build a multi-screen app that handles state transitions correctly.

What is Navigation?

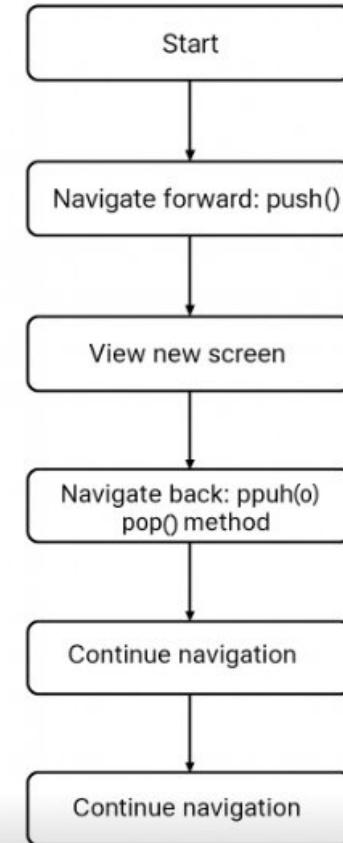
- The Navigator widget has an array of pages. A Page is an abstract class that has a name and argument.
- Navigation in Flutter controls movement between screens (routes).
- Each new page is pushed onto a stack, and when you go back, the top page is popped off. There are a few important methods, as follows:
 - push(context, route): Push a new route/page to the stack.
 - pushNamed(context, routeName, arguments): Find the route with the given name and push it onto the stack.
 - pop(context): Pop the current, top-level page off the stack, showing the next page on the stack

Navigator 1.0 Flow

Core Navigator 1.0 flowchart

- Start: App begins at the initial screen.
- push(): Navigate to a new screen; it is added to the top of the stack.
- View screen: User sees the new screen; the stack grows by one layer.
- pop(): Return to the previous screen; the top screen is removed.
- Repeat: Continue navigating forward (push) and backward (pop).ving it from the stack

Core Navigator 1.0 flowchart



Routes in Flutter

- A Route is a screen or page inside an app.
MaterialPageRoute is a common implementation.
- Code example

```
Navigator.push(  
    context,  
    MaterialPageRoute(builder: (context) => ProfilePage()),  
);
```

Note:

The widget returned by builder defines the next screen's UI.

Navigator Class

Core Methods:

Method	Purpose
push ()	Adds a new route
pop ()	Removes the current route
pushReplacement ()	Replaces current route
pushNamed ()	Navigate by route name
maybePop ()	Conditionally pop a route

Example:

```
Navigator.pushReplacementNamed(context, '/home');
```

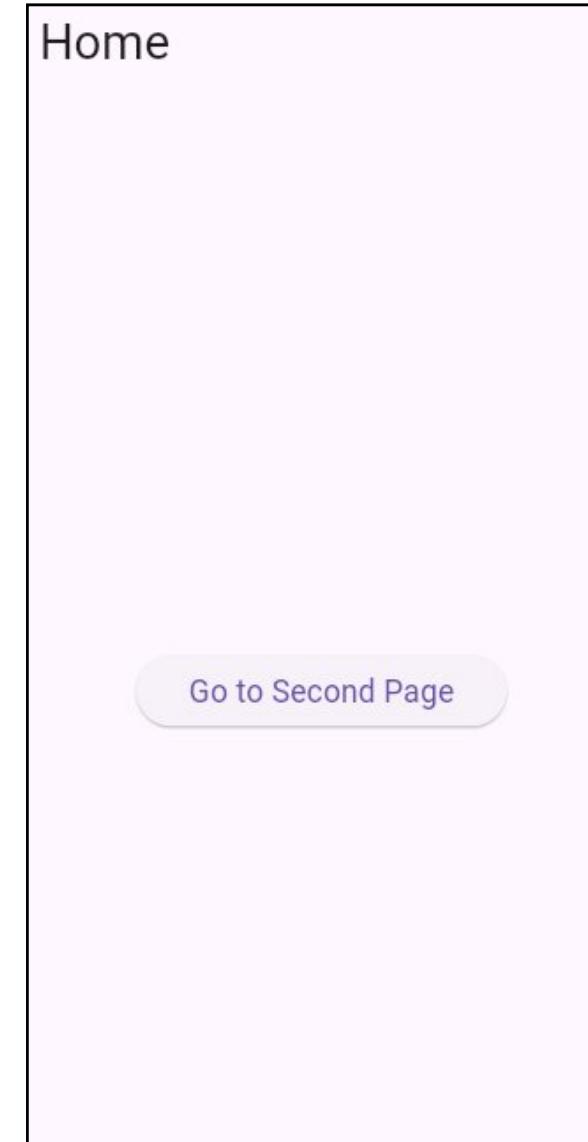
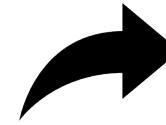
Example: Basic Navigation

Explanation of why the original code does not run in DartPad

```
class HomePage extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(title: Text("Home")),  
      body: Center(  
        child: ElevatedButton(  
          onPressed: () {  
            Navigator.push(  
              context,  
              MaterialPageRoute(builder: (_) => SecondPage()),  
            );  
          },  
          child: Text('Go to Second Page'),  
        ),  
      ),  
    );  
  }  
}
```

Example: Basic Navigation

```
1 import 'package:flutter/material.dart';
2
3 void main() {
4   runApp(MyApp());
5 }
6
7 class MyApp extends StatelessWidget {
8   @override
9   Widget build(BuildContext context) {
10     return MaterialApp(
11       debugShowCheckedModeBanner: false,
12       home: HomePage(),
13     );
14   }
15 }
16
17 class HomePage extends StatelessWidget {
18   @override
19   Widget build(BuildContext context) {
20     return Scaffold(
21       appBar: AppBar(title: Text("Home")),
22       body: Center(
23         child: ElevatedButton(
24           onPressed: () {
25             Navigator.push(
26               context,
27               MaterialPageRoute(builder: (_) => SecondPage()),
28             );
29           },
30           child: Text('Go to Second Page'),
31         ),
32       ),
33     );
34   }
35 }
36
37 class SecondPage extends StatelessWidget {
38   @override
39   Widget build(BuildContext context) {
40     return Scaffold(
41       appBar: AppBar(title: Text("Second Page")),
42       body: Center(
43         child: ElevatedButton(
44           onPressed: () {
45             Navigator.pop(context);
46           },
47           child: Text("Back"),
48         ),
49       ),
50     );
51   }
52 }
```



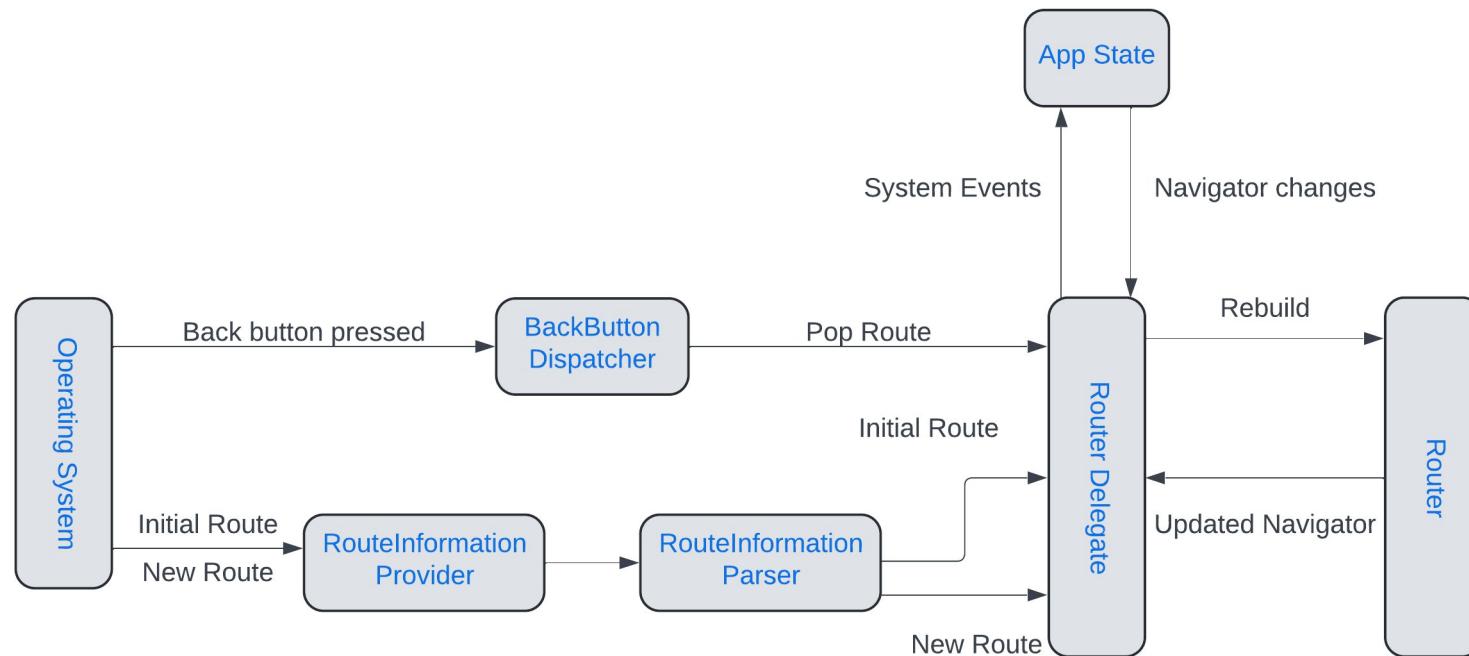
Named Routes

- Concept: Named routes simplify app navigation by creating a map of names to screens. Structure (template)
- Code Example:

```
MaterialApp(  
    routes: {  
        '/': (context) => HomePage(),  
        '/details': (context) => DetailPage(),  
    },  
);  
Navigator.pushNamed(context, '/details');
```

Navigator 2.0 Architecture

- Concept: Navigator 2.0 introduces a declarative model.
- Instead of calling push() and pop() manually, you describe the desired stack.



Deep Linking Overview

- Concept: Deep linking allows users to open a specific screen via a URL or external source.
- Example URL

`https://website/moviedetails/?id=1234&img=4234`

↑
Scheme

↑
Host

↑
Path

↑
Query Parameter

Setting Up Android Deep Links

- Step 1: Modify AndroidManifest.

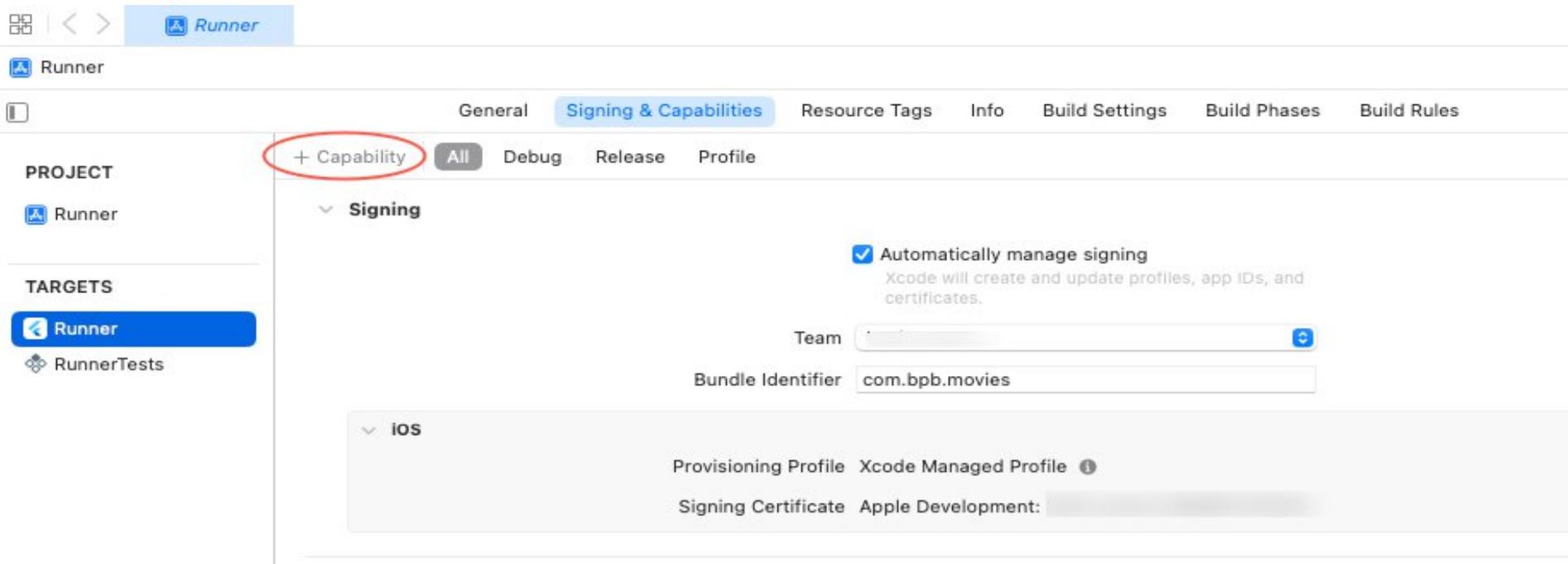
```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="movieapps" android:host="movies" />
</intent-filter>
```

- Step 2: Add a test link in browser or terminal

```
adb shell am start -a android.intent.action.VIEW -d"movieapps://movies?id=12"
```

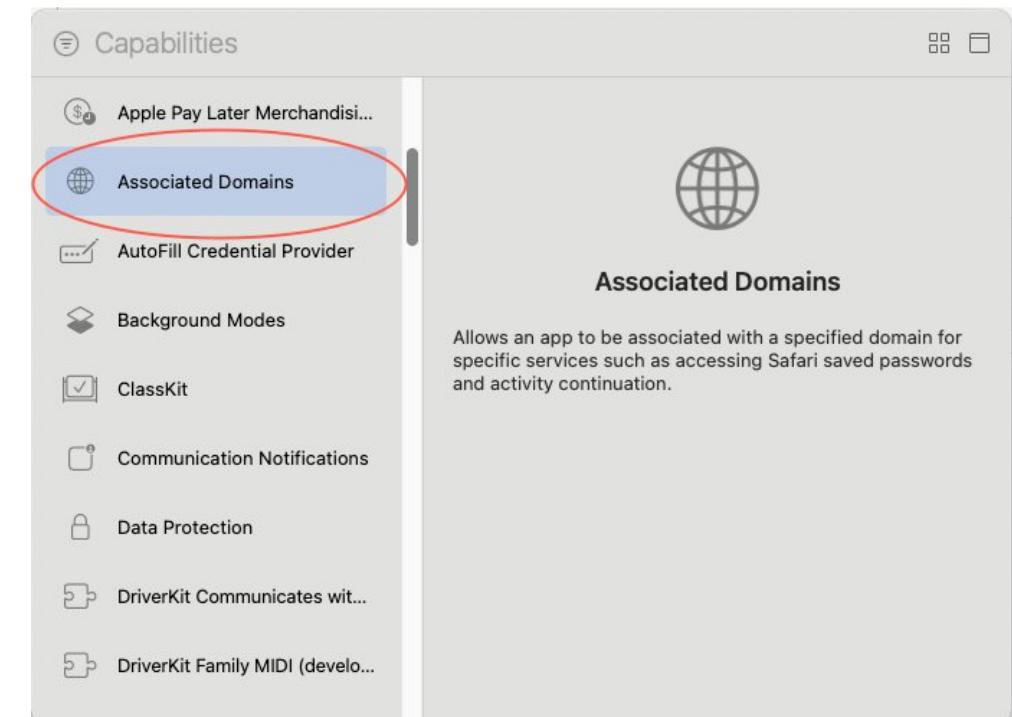
Setting Up iOS Deep Links (Part 1)

- Concept: Enable Associated Domains capability inside Xcode.
- Steps:
 - 1 Open project → select Runner target
 - 2 Open Signing & Capabilities tab
 - 3 Click + Capability → select Associated Domains



Setting Up iOS Deep Links (Part 2)

- Steps continued:
- 4 Add domain entry:
applinks:movieapps.com



Associated Domains

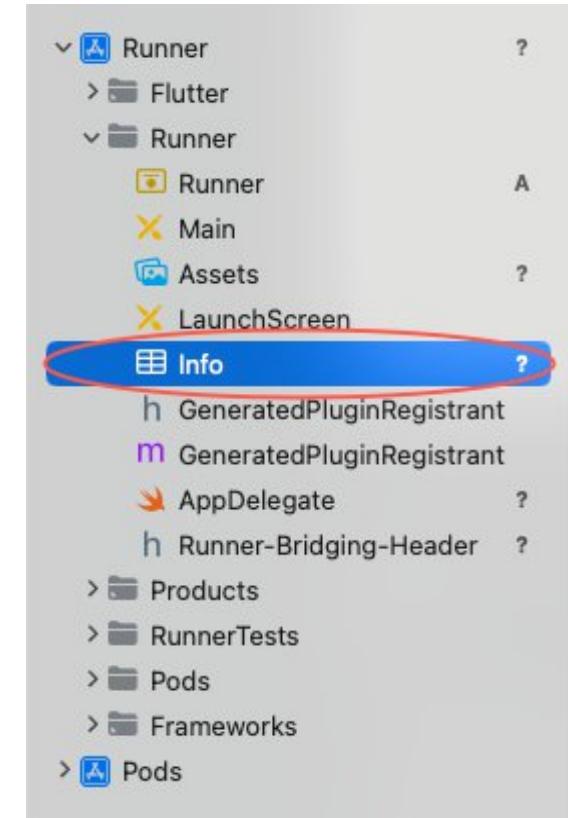
Domains

+ -

Editing Info.plist

- Concept: Add URL Types to Info.plist to define the custom scheme.
- Steps: Navigate to Runner/Info.plist
 - Add a new entry: URL Types → URL Schemes → movieapps

Key	Type	Value
Information Property List	Dictionary	(18 items)
URL types	Array	(1 item)
Item 0 (movieapps)	Dictionary	(2 items)
URL Schemes	Array	(1 item)
Item 0	String	movieapps
URL identifier	String	movieapps
Default localization	String	\$(DEVELOPMENT_LANGUAGE)



Info.plist XML Example

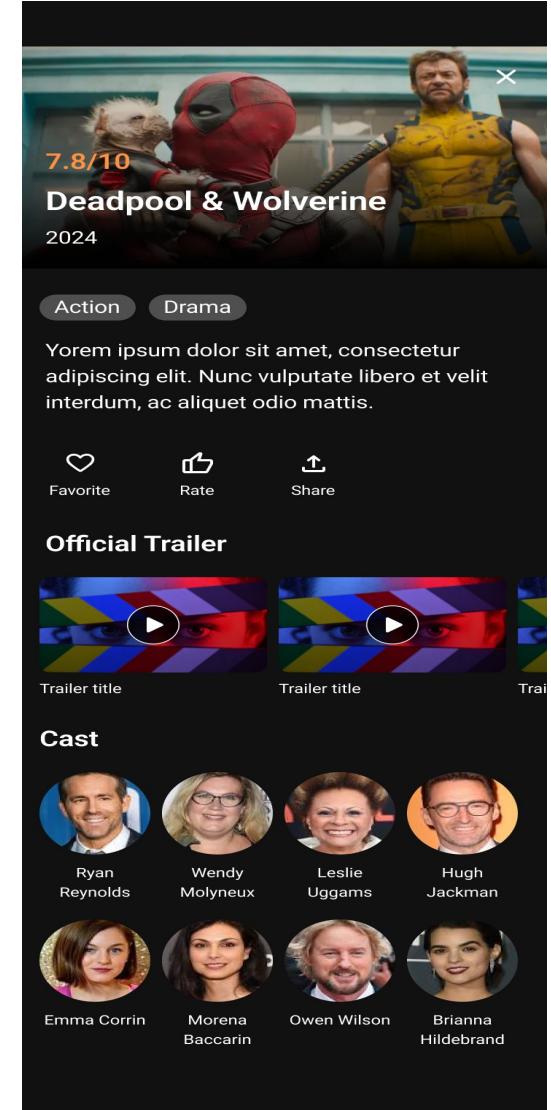
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleURLTypes</key>
    <array>
        <dict>
            <key>CFBundleURLSchemes</key>
            <array>
                <string>movieapps</string>
            </array>
            <key>CFBundleURLName</key>
            <string>movieapps</string>
        </dict>
    </array>
</dict>
</plist>
```

Show how this snippet connects to the scheme movieapps://movies?id=1.

Demo Start: Blank Scaffold

- Goal: Start MovieDetailPage with a blank scaffold.
- Content:

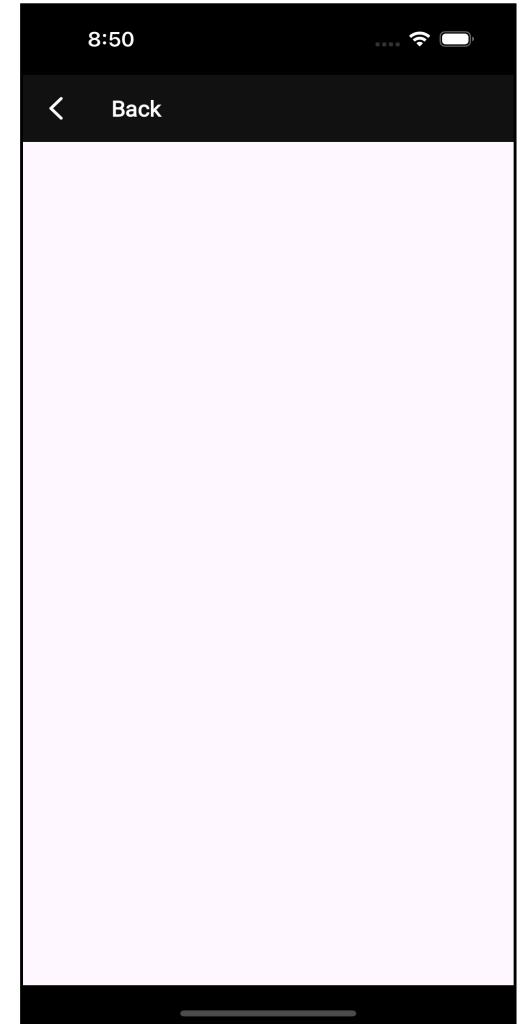
```
class MovieDetailPage extends  
StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(title: Text("Movie Detail")),  
      body: Center(child: Text("Coming  
soon...")),  
    );  
  }  
}
```



Step 1: Add Hero Banner

- Goal: Add movie poster with gradient overlay.
- Content:

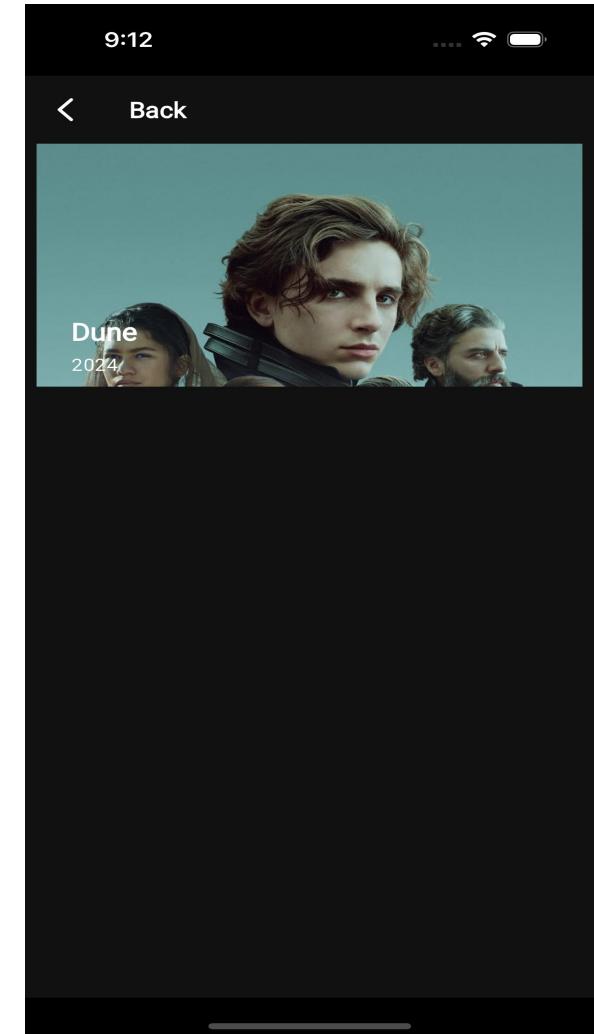
```
Stack(  
    children: [  
        Image.network(movie.posterUrl, fit:  
BoxFit.cover),  
        Positioned(bottom: 0, child:  
Container(height: 80, decoration: gradient)),  
    ],  
)
```



Step 2: Add Title & Genres

- Goal: Add movie title and genre chips.
- Content:

```
Column(  
    crossAxisAlignment:  
    CrossAxisAlignmentAlignment.start,  
    children: [  
        Text(movie.title, style:  
            TextStyle(fontSize: 22, fontWeight:  
                FontWeight.bold)),  
        Wrap(spacing: 6, children:  
            movie.genres.map((g) => Chip(label:  
                Text(g))).toList(),  
        ],  
    )
```



Step 3: Add Overview Text

- Goal: Add movie description text.
- Content:

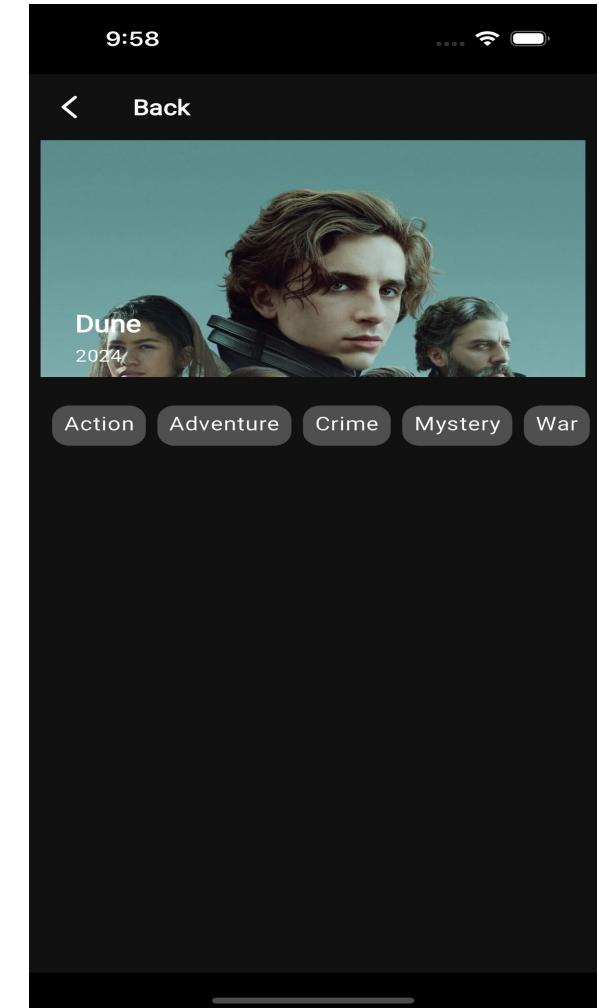
```
Padding(
```

```
    padding: const EdgeInsets.all(12.0),
```

```
    child: Text(movie.overview, style:
```

```
        TextStyle(fontSize: 16)),
```

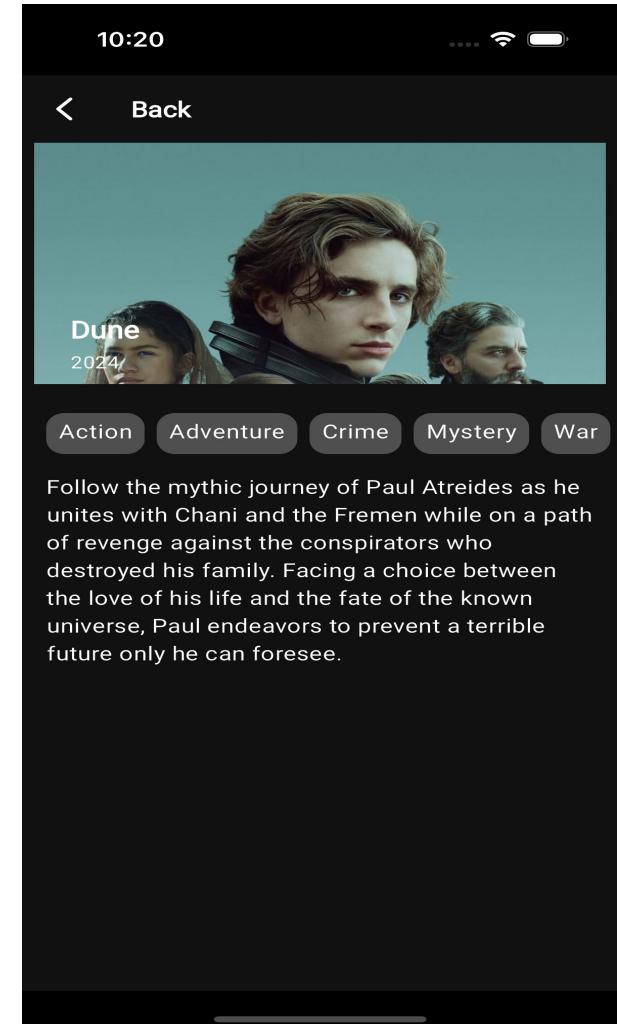
```
)
```



Step 4: Add Action Buttons

- Goal: Add “Favorite”, “Rate”, and “Share” icons.
- Content:

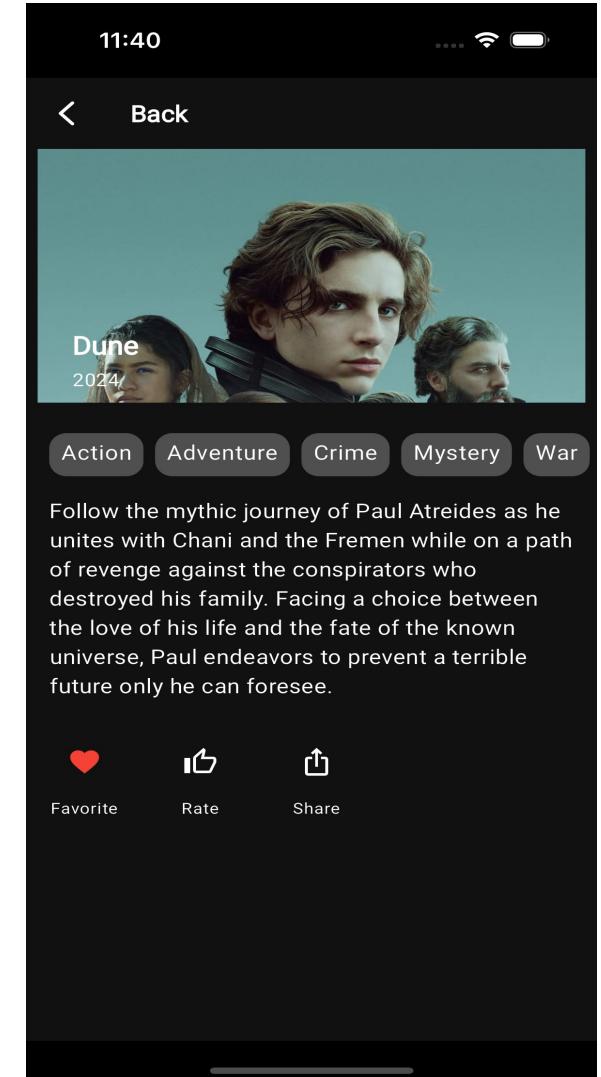
```
Row(  
    mainAxisAlignment:  
    MainAxisAlignment.spaceEvenly,  
    children: [  
        IconButton(icon: Icon(Icons.favorite),  
        onPressed: () {}),  
        IconButton(icon: Icon(Icons.star_rate),  
        onPressed: () {}),  
        IconButton(icon: Icon(Icons.share),  
        onPressed: () {}),  
    ],  
)
```



Step 5: Add Trailer List

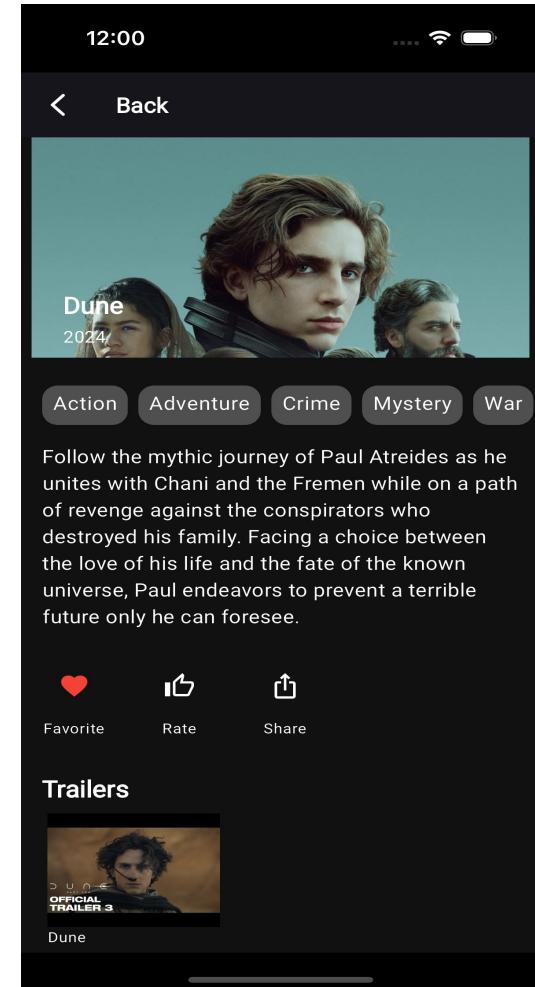
- Goal: Display list of trailers below description..
- Content:

```
ListView.builder(  
    itemCount: movie.trailers.length,  
    itemBuilder: (_, i) => ListTile(  
        leading: Icon(Icons.play_circle_fill),  
        title: Text(movie.trailers[i].title),  
    ),  
)
```



Step 6: Final Layout: Complete Detail Screen

- Goal: Combine all widgets for the final view



Implementing Route Handling

- Concept:

Handle incoming deep links via onGenerateRoute.

Key	Type	Value
Information Property List	Dictionary	(17 items)
Default localization	String	\$(DEVELOPMENT_LANGUAGE)
Bundle display name	String	Movies
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	movies
Bundle OS Type code	String	APPL
Bundle version string (short)	String	\$(FLUTTER_BUILD_NAME)
Bundle creator OS Type code	String	????
Bundle version	String	\$(FLUTTER_BUILD_NUMBER)
Application requires iPhone environment	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Supported interface orientations	Array	(3 items)
Supported interface orientations (iPad)	Array	(4 items)
CADisableMinimumFrameDurationOnPhone	Boolean	YES
Application supports indirect input events	Boolean	YES

Navigator 2.0 with State Integration

- Concept:
 - Navigator 2.0 works with state objects to react to URL changes.

Key	Type	Value
Information Property List	Dictionary	(18 items)
URL types	String	
Supports Full Screen Accessibility Access	String	\$(DEVELOPMENT_LANGUAGE)
Supports HDR color mode	String	Movies
Supports Live Activities	String	\$(EXECUTABLE_NAME)
Supports Live Activities Frequent Updates	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
Supports opening documents in place	String	6.0
Supports Print Command	String	movies
Tools owned after installation	String	APPL
URL types	String	\$(FLUTTER_BUILD_NAME)
View controller-based status bar appearance	String	????
White Point Adaptivity Style	String	\$(FLUTTER_BUILD_NUMBER)
Widget Wants Location	Boolean	YES
Bundle version	String	LaunchScreen
Application requires iPhone environment	String	Main
Launch screen interface file base name	Array	(3 items)
Main storyboard file base name	Array	(4 items)
Supported interface orientations	Boolean	CADisableMinimumFrameDurationOnPhone
Supported interface orientations (iPad)	Boolean	YES
Application supports indirect input events	Boolean	YES

Managing App State

- Concept:
State determines what the user sees. Managing it properly keeps the UI consistent.
- Example:

```
class AppState extends ChangeNotifier {  
    bool loggedIn = false;  
    void login() { loggedIn = true; notifyListeners(); }  
}
```

- Note: Explain why state is separated from UI (architecture clarity, testability).

Stateless vs Stateful Widgets

- Comparison Table:

Type	Rebuilds on Change	Example Use
StatelessWidget	No	Static UI – Logo, Text
StatefulWidget	Yes	Interactive UI – Form, Counter

Note: Highlight that Stateful widgets use `setState()` to trigger rebuilds.

setState() Usage - Lifting State Up

- Example: setState() Usage

```
class CounterPage extends StatefulWidget { ... }  
setState(() {  
    count++;  
});
```

- Concept: Lifting State Up

Share state between child widgets by moving the state to a common ancestor.

```
class ParentWidget extends StatefulWidget {  
    @override  
    _ParentWidgetState createState() => _ParentWidgetState();  
}
```

InheritedWidget

- Pass data down the widget tree efficiently.
- Code Example:

```
class AppData extends InheritedWidget {  
    final String theme;  
    const AppData({required this.theme, required Widget child, Key? key})  
        : super(key: key, child: child);  
    static AppData of(BuildContext context) =>  
        context.dependOnInheritedWidgetOfExactType<AppData>()!;  
}
```

Provider

Provider Overview

- Provider simplifies state sharing and listening to changes.

```
ChangeNotifierProvider(  
    create: (_) => CounterModel(),  
    child: CounterPage(),  
);
```

Provider Example

```
class CounterModel with ChangeNotifier {  
    int count = 0;  
    void increment() { count++;  
        notifyListeners(); }  
}  
Consumer<CounterModel>(  
    builder: (_, counter, __) =>  
        Text('${counter.count}'),  
);
```

Combining Navigation & State

- Integrate state with routing logic.

```
if (appState.loggedIn)  
    Navigator.pushNamed(context, '/dashboard');  
  
else  
    Navigator.pushNamed(context, '/login');
```

Practice Task

Task

- Build a small “Movie App” with:
- Deep link opening movie detail page
- Provider for favorite movies state
- Working back button navigation

Summary

- Navigator 1.0 vs 2.0 architecture
- Deep link setup on Android & iOS
- State management patterns: setState, InheritedWidget, Provider
- Integration of navigation and state

References

- Mastering Flutter 2025 – Ch.6, Ch.8
- docs.flutter.dev
- <https://dartpad.dev>

Module 6 - Responsive UI & Adaptive Layouts

Contents

- What Responsive UI means
- Screen sizes, device types, and safe areas
- MediaQuery for screen information
- LayoutBuilder for constraint-based rendering
- OrientationBuilder for rotation
- Responsive widgets (Expanded, Wrap, Flexible)
- Building a real case study: Movie Genre Screen
- How to test responsive UI on multiple devices

Why Responsive UI Matters

Flutter runs on:

- Small phones
- Large phones
- Tablets
- Foldables
- Desktop browsers

Without responsive design:

- Overflow errors
- Misaligned widgets
- Text clipping
- Bad user experience

Responsive vs Adaptive UI

Responsive UI: Layout changes depending on available space.

Examples:

- Column → Row
- Grid 2 columns → 4 columns
- Padding automatically adjusts based on screen width.
- Adaptive UI: Layout changes depending on device type.

Examples:

- Bottom Nav → NavigationRail on tablet
- Drawer → Sidebar on desktop

■ *Code Snippet*

```
if (width < 600) return MobileView();  
return TabletView();
```

MediaQuery: Getting Screen Size

MediaQuery allows you to access device and screen information, such as:

- Screen width
- Height
- Orientation
- Safe area padding
- Pixel density

Code demo:

```
final width = MediaQuery.of(context).size.width;
```

```
final height = MediaQuery.of(context).size.height;
```

Using Breakpoints

Breakpoints help define layout logic.

- Example Rule:
 - < 400px → Small layout
 - 400–800px → Medium layout
 - 800px → Large layout

```
if (width < 400) return SmallLayout();
else if (width < 800) return
MediumLayout();
return LargeLayout();
```

Code Demo (Runnable):

```
Widget build(BuildContext context) {
final w =
MediaQuery.of(context).size.width;
return Scaffold(
body: Center(
child: Text(
w < 600 ? "Mobile Layout" : "Tablet
Layout",
style: TextStyle(fontSize: 24),
),
),
);
}
```

LayoutBuilder

- **LayoutBuilder gives parent constraints, not full screen size.**

Code Example:

```
LayoutBuilder(  
    builder: (context, c) {  
        return c.maxWidth > 500  
            ? WideCard()  
            : NarrowCard();  
    },  
);
```

Code example:

```
LayoutBuilder(  
    builder: (context, c) {  
        return Container(  
            color: c.maxWidth > 300 ? Colors.blue  
                : Colors.red,  
            height: 120,  
        );  
    },  
);
```

OrientationBuilder

- Detects device rotation.

```
OrientationBuilder(  
    builder: (context, orientation) {  
        return orientation == Orientation.portrait  
            ? PortraitUI()  
            : LandscapeUI();  
    },  
);
```

- Demo:
 - Portrait: list
 - Landscape: two-column layout

Responsive Widgets Overview

Useful widgets:

- Expanded – prevents overflow
- Flexible – controls how a child flexes within a Row or Column
- Wrap – auto line wrapping
- AspectRatio – avoids image stretching
- FractionallySizedBox – percent-based size
- GridView – multi-column responsive layout

Responsive GridView

- GridView allows building scrollable grids that automatically adapt to screen width by adjusting the number of columns.
- Code Example:

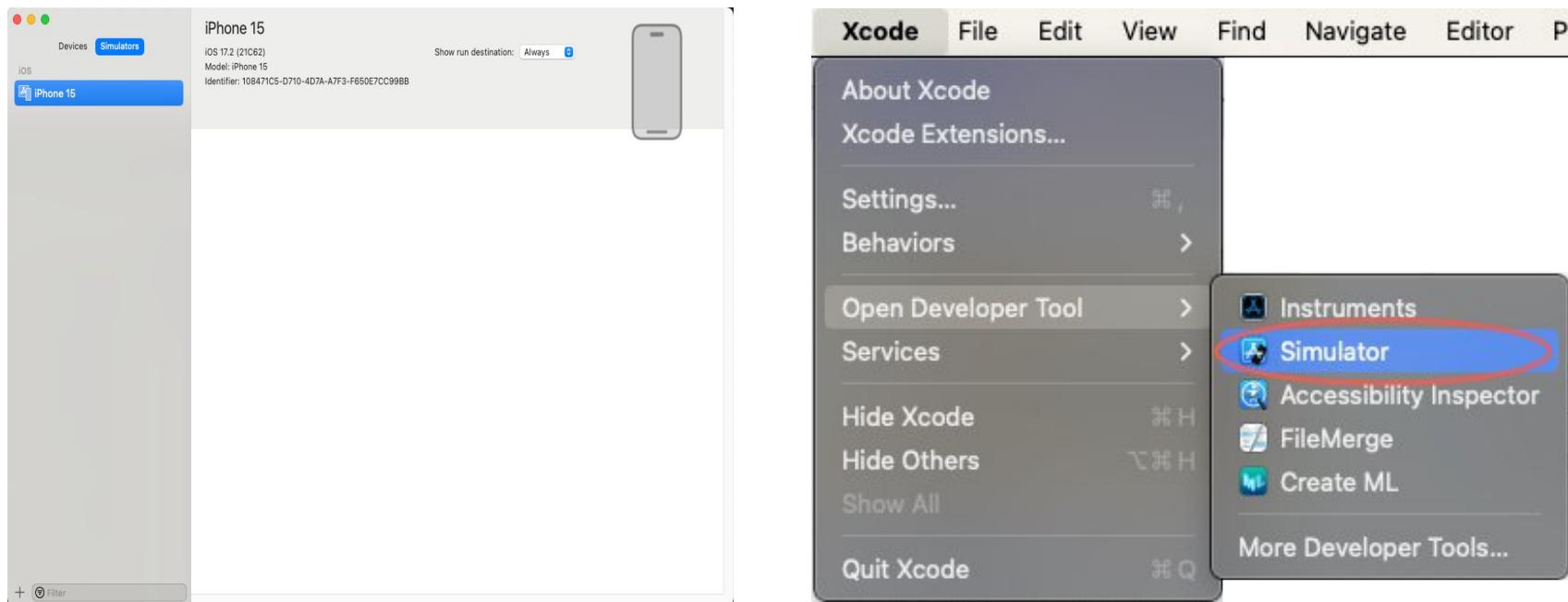
```
GridView.count(  
    crossAxisCount: width < 600 ? 2 : 4,  
);
```

- Mobile: 2 columns
- Tablet: 3–4 columns
- Desktop: 5–6 columns

SafeArea & Modern Devices

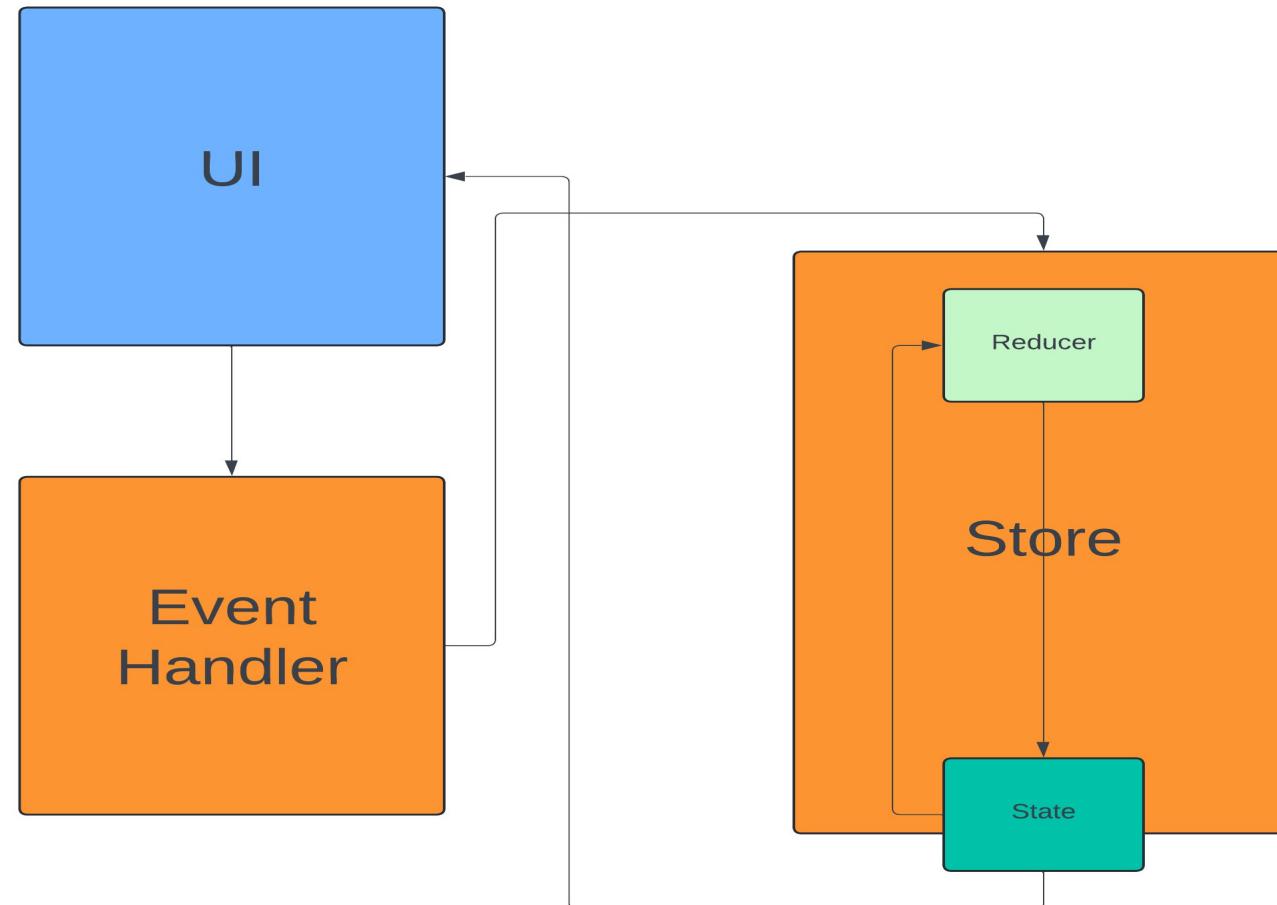
- Dynamic Island / notches / curved screens
→ Some space cannot be used.
- Use:

SafeArea(child: ...)



Real Case Study: Genre Screen

- We now apply all concepts to a real UI



Responsive Search Bar

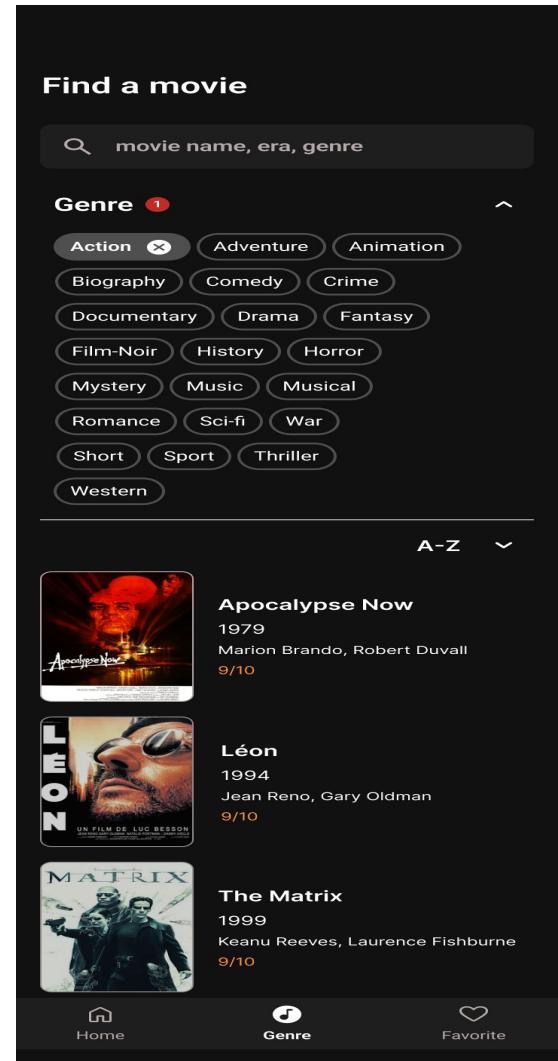
- Key behaviors:
 - Full width
 - Padded for large screens
 - Wrapped in SafeArea
- **Mini Code:**

Container(

padding: EdgeInsets.symmetric(horizontal: 16),

child: TextField(...),

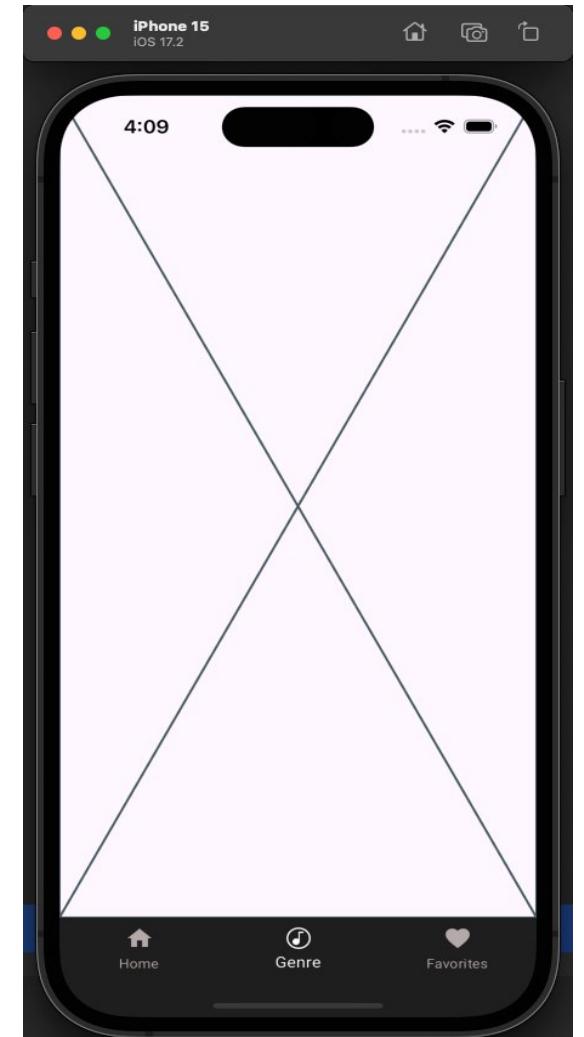
)



Genre Chips (Collapsed State)

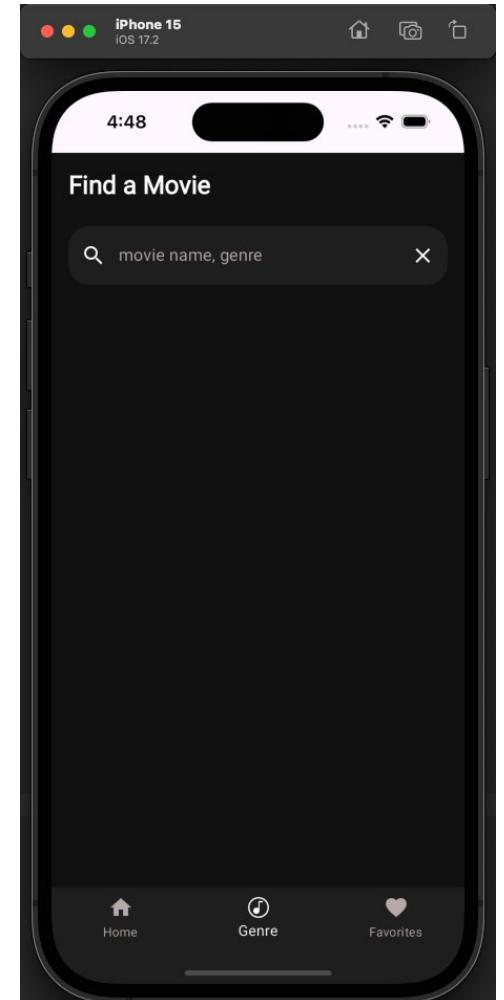
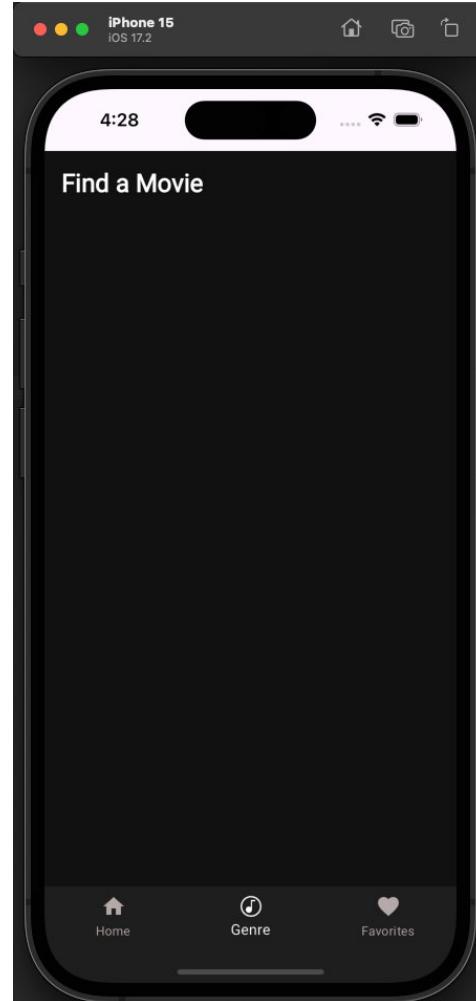
- Use the Wrap widget to ensure chips automatically flow to the next line when horizontal space is insufficient:

```
Wrap(  
    spacing: 8,  
    runSpacing: 8,  
    children: genres.map(_buildChip).toList(),  
)
```



Genre Chips (Expanded State)

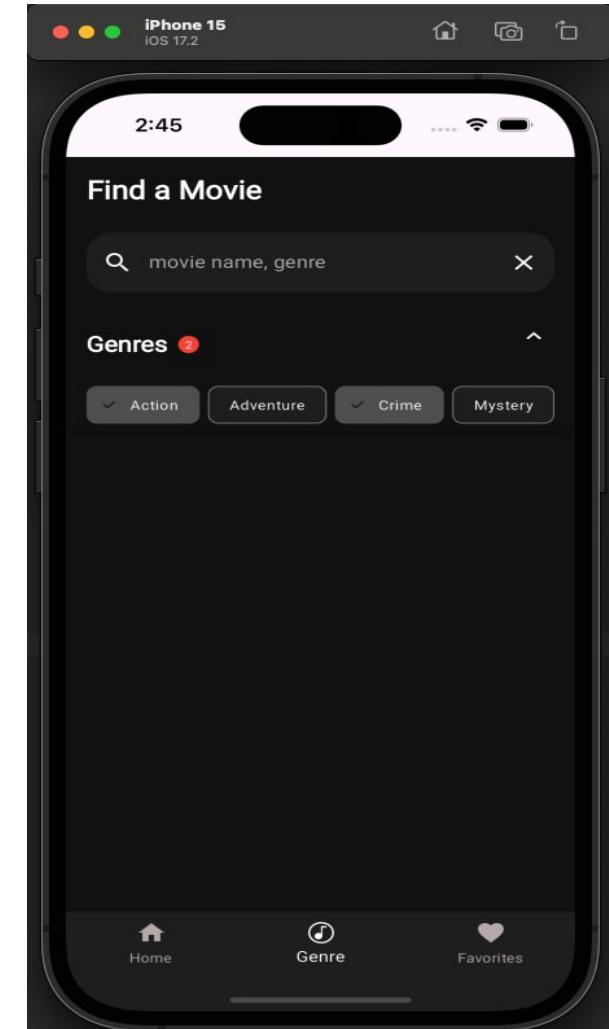
- Chips wrap automatically
- More space → more chips per row
- Less space → fewer chips per row
- This behavior is achieved automatically by using Wrap, without conditional logic.



Sort Dropdown

- Sorting options include A–Z, Z–A, Rating, and Year.
- Code Snippet:

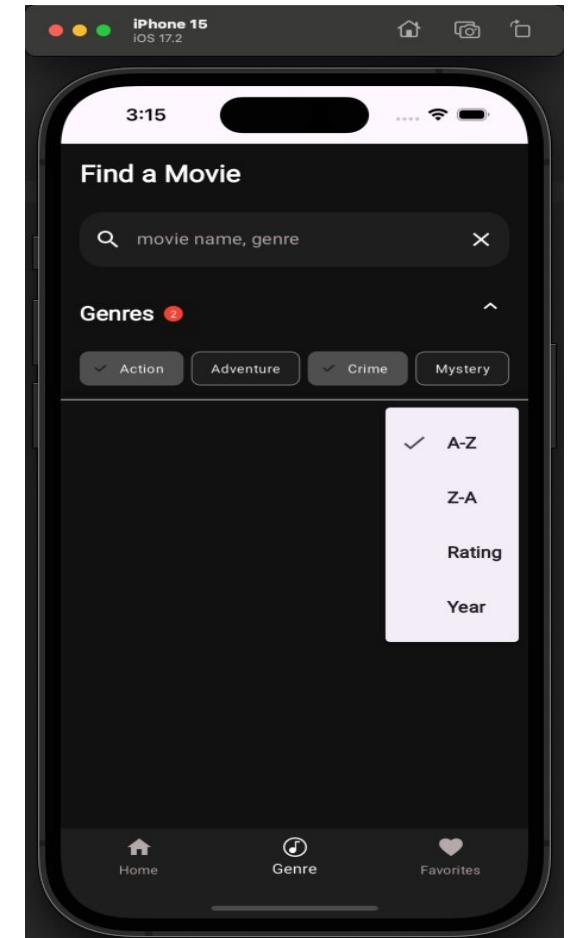
```
DropdownButton(  
    value: selectedSort,  
    items: [...],  
    onChanged: (v) => setState(() =>  
        selectedSort = v!),  
)
```



Responsive Movie Card

- **Code Snippet:**

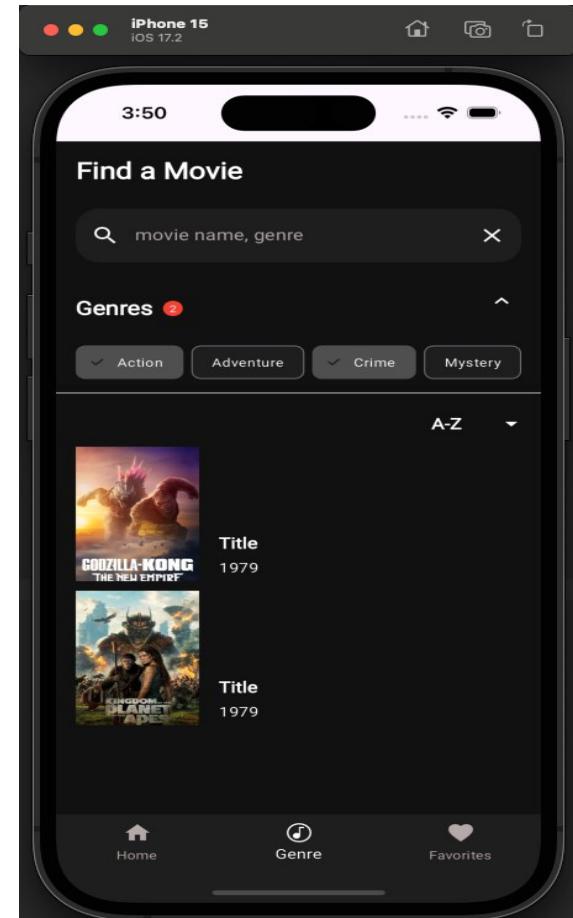
```
LayoutBuilder(  
    builder: (context, c) {  
        final isWide = c.maxWidth > 600;  
        return Row(  
            children: [  
                Image.network(poster, width: isWide ? 150 : 100),  
                SizedBox(width: 16),  
                Expanded(child: Text(title))  
            ],  
        );  
    },  
)
```



Completed Responsive Screen

Elements included:

- Responsive search
- Dynamic chips
- Sort menu
- Responsive movie list



Lab 6.1

Responsive Hero Section

- Students must create a responsive home banner.
 - Width scaling
 - Height scaling
 - Breakpoints
 - Buttons adapt size

Lab 6.2

Rebuild the Genre Screen (Figures 6.1–6.9)

Requirements:

- Responsive search bar
- Wrap-based chips
- Sort dropdown
- Movie list responsive
- Test on phone + tablet sizes

Lab 6.3

Tablet Layout Enhancement

For screens > 800px:

- Add sidebar filter
- 2-column movie list
- Increased spacing

Best Practices

- Avoid fixed values
- Use constraints-aware components
- Prefer Wrap over Row for long chip lists
- Test often on multiple screen sizes
- Use Flutter Inspector's “Layout Explorer”

Summary

- Responsive vs Adaptive
- MediaQuery / LayoutBuilder / OrientationBuilder
- Responsive Widgets
- Implemented a full responsive Genre screen
- Applied GridView and Wrap techniques

References

- Mastering Flutter 2025 – Ch.5, 6
- docs.flutter.dev
- <https://dartpad.dev>

Module 7 - Forms and Validation

User Input • Validation • UX • Async Checks

Contents

- Build forms using Form and TextFormField
- Use FormState and GlobalKey to validate and save data
- Apply built-in and custom validation rules
- Manage focus and keyboard interactions
- Design user-friendly error messages
- Implement a complete signup form with validation
- This module leads into Module 8 – API Networking

Why Forms Matter

Why forms are critical in apps:

Common use cases:

- Login / Signup
- Profile update
- Feedback & contact forms
- Booking and checkout
- Search and filtering inputs

If forms are done poorly:

- Users submit invalid or incomplete data
- Users get frustrated and abandon the app
- Security and data quality issues appear

Key idea: Good apps = good forms + good validation

Input Widgets Overview

Core input widgets in Flutter

- `TextField` – basic text input
- `TextFormField` – text input with validation in a form
- `Checkbox`, `Switch`, `Radio` – boolean choices
- `DropdownButton` – pick from a list of options
- Date & time pickers – select from dialogs

When to use which?

- Use `TextField` for simple, one-off inputs
- Use `TextFormField` inside a `Form` when you need validation

Form & FormState

What is a Form?

- A Form groups multiple input fields
- A GlobalKey<FormState> lets you:
 - Validate all fields at once
 - Save all field values
 - Reset the form

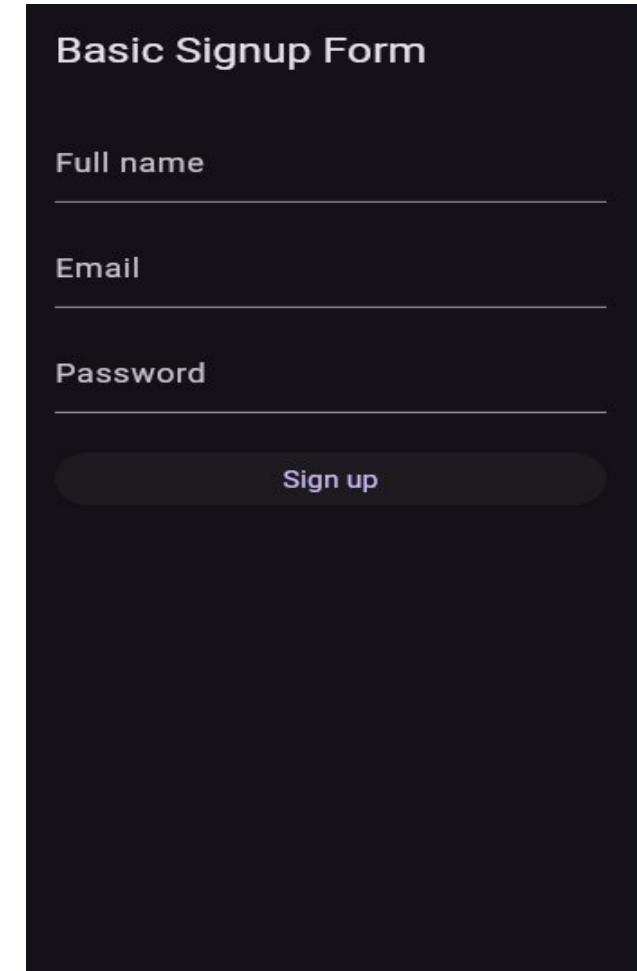
Core methods:

- formKey.currentState!.validate()
- formKey.currentState!.save()
- formKey.currentState!.reset()

```
final _formKey = GlobalKey<FormState>();  
Form(  
  key: _formKey,  
  child: Column(  
    children: [  
      TextFormField(  
        decoration: const InputDecoration(labelText: 'Name'),  
        validator: (value) {  
          if (value == null || value.trim().isEmpty) {  
            return 'Name is required';  
          }  
          return null;  
        },  
      ),  
      ElevatedButton(  
        onPressed: () {  
          if (_formKey.currentState!.validate()) {  
            _formKey.currentState!.save();  
          }  
        },  
        child: const Text('Submit'),  
      ),  
    ],  
  );
```

Demo 1- Basic Signup Form

- Goal: Create a simple signup form using Form + TextFormField.
- Requirements
 - Fields: Name – Email – Password
 - All fields required
 - Email must contain @ and .
 - Password \geq 6 characters
- Submit Behavior
 - validate() → show inline errors
 - If valid → show Snackbar: “Signup successful”



Basic Signup Form - Structure

- Minimal signup screen structure
 - Scaffold
 - AppBar
 - Form with `_formKey`
 - `TextFormFields` for name, email, password
 - A submit button

```
Scaffold(  
  appBar: AppBar(title: const Text('Basic Signup Form')),  
  body: Padding(  
    padding: const EdgeInsets.all(16),  
    child: Form(  
      key: _formKey,  
      child: ListView(  
        children: [  
          TextFormField(  
            decoration: const InputDecoration(labelText: 'Full name'),  
          ),  
          TextFormField(  
            decoration: const InputDecoration(labelText: 'Email'),  
          ),  
          TextFormField(  
            decoration: const InputDecoration(labelText: 'Password'),  
            obscureText: true,  
          ),  
          const SizedBox(height: 24),  
          ElevatedButton(  
            onPressed: _submit,  
            child: const Text('Sign up'),  
          ),  
        ],  
      ),  
    ),  
  );
```

Basic Signup Form - Built-in Validators

- **The validator function**
 - Signature: String? Function(String? value)
 - If valid → return null
 - If invalid → return error message (String)
- **Common checks:**
 - Required field (not empty)
 - Email format (contains @ and .)
 - Minimum length

```
TextField(  
  decoration: const InputDecoration(labelText: 'Email'),  
  validator: (value) {  
    final text = value?.trim() ?? '';  
    if (text.isEmpty) return 'Email is required';  
    if (!text.contains('@') || !text.contains('.')) {  
      return 'Enter a valid email';  
    }  
    return null;  
  },  
);
```

Basic Signup Form - Submit Flow & Button State

- **Typical submit flow:**
 - User taps Submit
 - Call validate()
 - If valid → call save() and submit data
 - If invalid → show inline errors
 - Display error when form invalid
 - Display Snackbar when valid
 - “FormState standard” Submit logic
- **(Simple submit flow → validate → save → Snackbar)**

```
void _submit() {  
    final isValid = _formKey.currentState!.validate();  
    if (!isValid) return;  
  
    _formKey.currentState!.save();  
  
    ScaffoldMessenger.of(context).showSnackBar(  
        SnackBar(content: Text('Form is valid!')),  
    );  
}
```

Demo 2 - Validation Rules

- Goal: Add strong password + confirm password logic.
- Requirements:
 - Password \geq 8 chars & includes number
 - Confirm password must match
 - Use helper validator functions
 - Enable AutovalidateMode.onUserInteraction

← Lab 7.2 – Validation Rules

Email

Password

Confirm password

Create account

Demo 2 - Custom Validation Rules

- **Use custom validators for:**
 - Business rules (username rules, banned words)
 - Password strength (length, numbers, symbols)
 - More advanced email checks (pattern / regex)

```
String? validatePassword(String? value) {  
    final text = value ?? '';  
    if (text.isEmpty) return 'Password is required';  
    if (text.length < 8) return 'At least 8 characters';  
    if (!text.contains(RegExp(r'[0-9]'))) {  
        return 'Add at least one number';  
    }  
    return null;  
}  
  
String? validateConfirmPassword(String? value, String password) {  
    if (value == null || value.isEmpty) return 'Please confirm password';  
    if (value != password) return 'Passwords do not match';  
    return null;  
}
```

Demo 2 - Confirm Password Logic

- Why confirm password?
 - Prevent typos
 - Improve user confidence
- Pattern:
 - Keep `_password` in state
 - Compare confirm password with `_password`
 - Show error if they differ

```
TextField(  
  decoration: const InputDecoration(labelText: 'Confirm password'),  
  obscureText: true,  
  onChanged: (value) => _confirmPassword = value,  
  validator: (value) {  
    if (value == null || value.isEmpty) {  
      return 'Please confirm password';  
    }  
    if (value != _password) {  
      return 'Passwords do not match';  
    }  
    return null;  
  },  
);
```

Demo 2 - AutovalidateMode

- **Autovalidate modes:**
 - AutovalidateMode.disabled
 - AutovalidateMode.always
 - AutovalidateMode.onUserInteraction
(recommended)
- **UX trade-offs:**
 - Too early → annoying (red errors everywhere from the start)
 - Too late → confusing (errors appear only after submit)

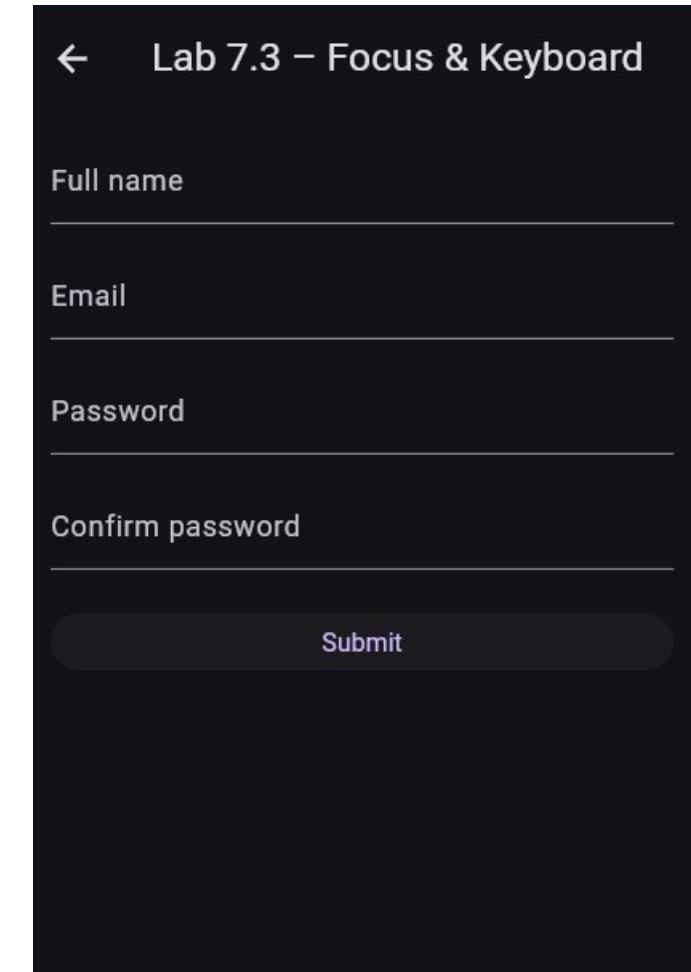
```
Form(  
  key: _formKey,  
  autovalidateMode: AutovalidateMode.onUserInteraction,  
  child: Column(  
    children: [ /* fields */ ],  
  ),  
,
```

Error Messages & UX

- Good error messages:
 - Short and clear
 - Specific to the field
 - Polite and helpful
 - Examples:
 - “Enter a valid email”
 - “Password must be at least 8 characters”
 - “ERROR_400_INVALID_FIELD”
- Best practices:**
- Show error under the field
 - Avoid blaming language
 - Highlight only relevant fields

Demo 3 - Focus & Keyboard UX

- **Goal:**
 - Improve input experience on mobile.
- **Requirements:**
 - Create FocusNode for each field
 - Use Next/Done with textInputAction
 - Move focus using onFieldSubmitted
 - Dismiss keyboard using GestureDetector
 - Wrap form with ListView to prevent overflow



Demo 3 - Managing Focus & Keyboard Content

- Why manage focus?
 - Mobile keyboards can hide fields
 - Users expect Next/Done to work smoothly
- Tools:
 - FocusNode
 - textInputAction
 - onFieldSubmitted
 - FocusScope.of(context).requestFocus(nextFocusNode)

```
final _nameFocus = FocusNode();
final _emailFocus = FocusNode();

TextField(
    focusNode: _nameFocus,
    textInputAction: TextInputAction.next,
    onFieldSubmitted: (_){},
    FocusScope.of(context).requestFocus(_emailFocus),
),
);

TextField(
    focusNode: _emailFocus,
    textInputAction: TextInputAction.done,
);
```

Demo 3 - Dismissing the Keyboard

- Common UX requirement:
 - Tap outside the form → keyboard hides
 - Pattern: Wrap with GestureDetector
- Also:
 - Wrap form in ListView or SingleChildScrollView to avoid overflow.

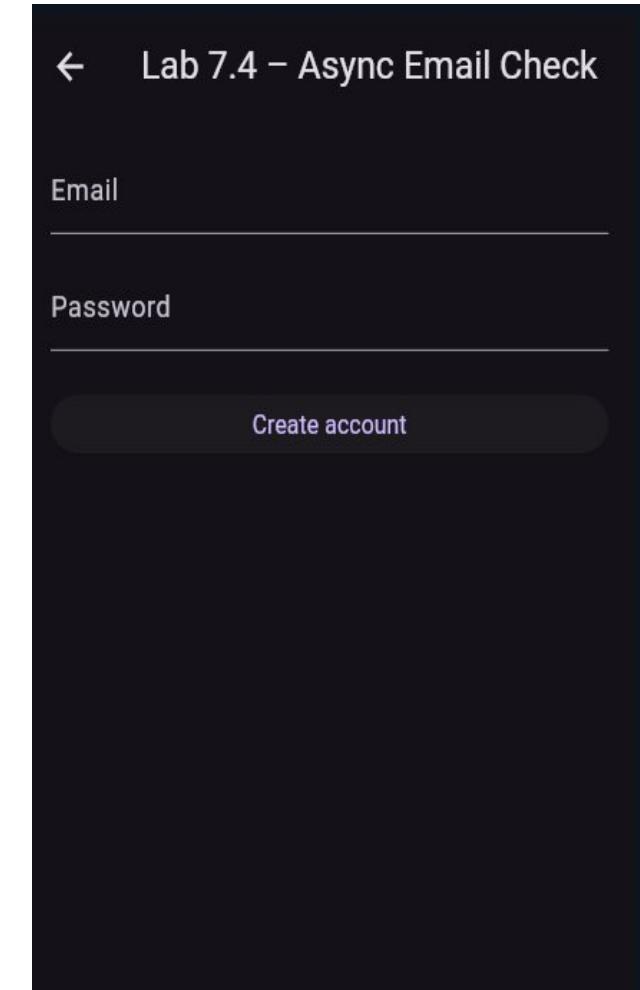
```
return GestureDetector(  
    onTap: () => FocusScope.of(context).unfocus(),  
    child: Scaffold(  
        appBar: AppBar(title: const Text('Focus & Keyboard')),  
        body: /* Form here */,  
    ),  
);
```

Form Layout Patterns

- **Common layouts:**
 - Single-page scrollable form
 - Grouped sections (e.g. Personal Info / Account Info)
 - Multi-step wizard (optional, conceptual)
- **Avoid:**
 - Too many fields on screen without grouping
 - Keyboard covering the submit button
 - Fixed height containers that don't scroll

Demo 4 - Async Email Check (Optional)

- **Goal:**
 - Simulate server-side validation before account creation.
- **Requirements**
 - Validate form locally
 - Show loading state during async check
 - Delay 2s: Future.delayed()
 - If email starts with “taken” → show error: “Email already taken”
 - Disable Submit button during check



Async Email Check (Optional)

Why async validation?

- Check if email/username already exists
 - Validate codes or coupons
 - Confirm availability (e.g. booking slot)
-
- **Pattern (simplified):**



```
bool _isCheckingEmail = false;  
Future<void> _submit() async {  
    if (!_formKey.currentState!.validate()) return;  
    setState(() => _isCheckingEmail = true);  
    await Future.delayed(const Duration(seconds: 2));  
    final emailTaken = _email.toLowerCase().startsWith('taken');  
    setState(() => _isCheckingEmail = false);  
    if (emailTaken) {  
        ScaffoldMessenger.of(context).showSnackBar(  
            const SnackBar(content: Text('This email is already taken')),  
        );  
        return;  
    }  
    ScaffoldMessenger.of(context).showSnackBar(  
        SnackBar(content: Text('Account created for ${_email}')),  
    );  
}
```

From Form to Backend (Suggested Content)

What happens after validation?

- User fills the form
- Local validation checks (Module 7)
- Send data via API (Module 8)
- Server performs deeper checks
- Server returns success/error
- UI updates accordingly

UI Form → Validation → POST Request → Server → Response → UI Update

Full Signup Demo (Combined)

Features combined:

- Name, email, password, confirm password
- Built-in + custom validation
- Autovalidate on user interaction
- Focus management (Next/Done)
- Show/hide password
- Async email check with loading button

Lab 7 – Forms & Validation Demos

Lab 7.1 – Basic Signup Form >
Form + TextFormField + basic validation

Lab 7.2 – Validation Rules >
Strong password + confirm password + autovalidate

Lab 7.3 – Focus & Keyboard >
FocusNode, Next/Done actions, dismiss keyboard

Lab 7.4 – Async Email Check >
Fake server delay + disable button while loading

Full Demo – Complete Signup Flow >
Combine validation + focus + async + UX tweaks

Lab Forms & Validation

Lab 7 – Forms & Validation

- Lab 7.1 – Basic registration form
- Lab 7.2 – Validation rules & password strength
- Lab 7.3 – Focus & keyboard management
- Lab 7.4 – Optional async email check
- **You will:**
 - Implement the signup form step by step
 - Run it on DartPad and on emulator/device
 - Capture screenshots and submit your code

Summary

In this module, you learned how to:

- Build forms using Form and TextFormField
- Use FormState to validate, save, and reset data
- Implement required fields and custom validation rules
- Manage focus and keyboard interactions on mobile
- Show clear error messages and success feedback
- Combine everything into a complete signup form

References

- Mastering Flutter 2025 – Ch.4, 14
- docs.flutter.dev
- <https://dartpad.dev>

Module 8 – Working with RESTful APIs & JSON in Flutter

Learning Objectives

Content:

- Understand basic HTTP and RESTful API concepts
- Send GET requests in Flutter using the http package
- Parse JSON responses into Dart model classes
- Display API data using FutureBuilder + ListView
- Handle loading, empty, and error states
- Send POST requests to create or submit data
- Organize API logic using a reusable Service Layer pattern

What is an API? What is HTTP?

API (Application Programming Interface)

- A contract that allows clients to talk to a server
- Often exposed as a set of URLs (endpoints)

HTTP fundamentals

- Methods: GET, POST, PUT, DELETE
- Status codes:
- 200–299: success
- 400–499: client error
- 500–599: server error

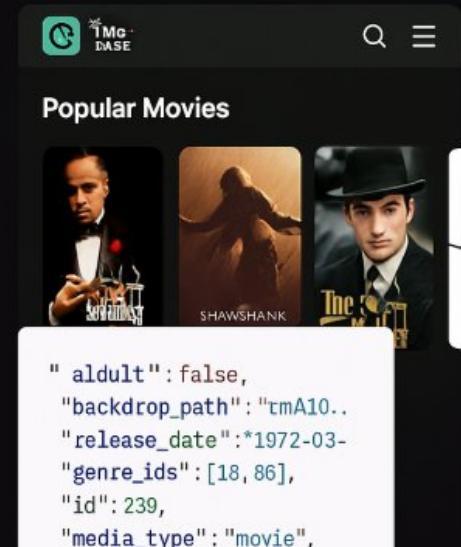
Real-world API Examples

Weather API



```
{  
  "weather": "800",  
  "icon": "61d"  
},  
  "main": {>284485,  
  "pressure": 1913,  
  "visibility": 19  
  "wind": {> speed: 15,  
  "clouds": 0,  
  "time": 166  
  "system": 166157059,  
  "country": "28": 4,  
  "sunrise": 166183487,  
  "sunset": 1661662248  
}
```

Movie API



```
"adult": false,  
"backdrop_path": "tmA10..  
"release_date": "1972-03-  
"genre_ids": [18, 86],  
"id": 239,  
"media_type": "movie",  
"original_language":  
"en"
```

JSONPlaceholder



1. Posts
 1. 1. sunt aut facere repellat provident occaecati excepturi optio
 1. 2. qui est esse
 1. 3. ea molestias quasi exercitationem repellat qui ioss sit aut
 1. 4. eum et est occaecati
 1. 5. nesciunt quas odio
 1. 6. dolorem eum magni eos aperiam quia

What is JSON?

JSON (JavaScript Object Notation)

- A lightweight text format for data exchange
- Key-value pairs and arrays
- Easy to read and parse
- Maps naturally to Dart structures:
Map<String, dynamic> and List<dynamic>
- **JSON in APIs**
 - APIs return JSON → your Flutter app decodes → maps to Model classes.

JSON Structure

```
{  
  "id": 1,  
  "name": "Alice",  
  "email": "example.com"  
}
```

A single JSON object (key-value pairs)

```
[  
  { "id": 1, "title":  
    { "title": "Post B" }  
  }  
]
```

An array containing multiple JSON obj

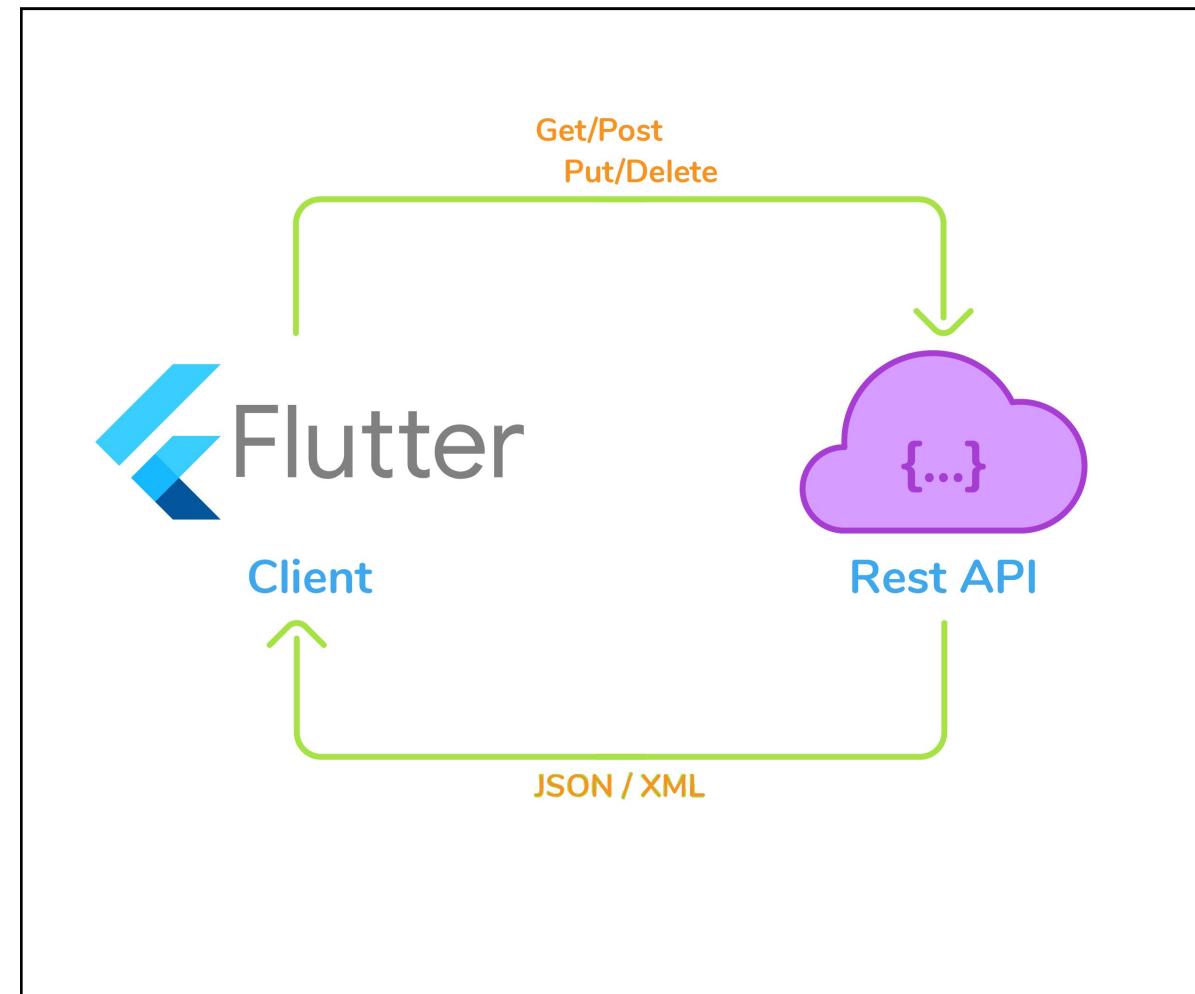
```
{  
  "user": {  
    "id": 1, title:"Hello"  
    "id": 2, title:"Hi again"  
  }  
}
```

Objects and arrays nested inside another object

API Networking in Flutter — The Big Picture

Steps to call an API in Flutter

1. Choose an endpoint (URL + method)
2. Send the request using http client
3. Receive JSON response
4. Decode JSON → Dart models
5. Update UI with the data
6. Handle loading & error states



Using the http Package

Common approach in Flutter

- Add dependency in pubspec.yaml: http
- Import in Dart file:

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

Typical usage

- http.get(Uri.parse(url))
- http.post(Uri.parse(url), body: {...})

Demo 8.1: Simple GET Request

Goal: Fetch data from a public REST API using an HTTP GET request and print the response.

- This demo demonstrates how to make a basic GET request in Flutter using the `http` package.
- The app will fetch a list of posts from the public API (<https://jsonplaceholder.typicode.com/posts>) and display the title of the first item.
- Steps to follow
 - Send an asynchronous HTTP GET request
 - Decode JSON into Dart objects
 - Handle loading status
 - Update UI when data arrives

Demo 8.1 – Simple GET

- 1 sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- 2 qui est esse
- 3 ea molestias quasi exercitationem repellat qui ipsa sit aut
- 4 eum et est occaecati
- 5 nesciunt quas odio
- 6 dolorem eum magni eos aperiam quia
- 7 magnam facilis autem
- 8 dolorem dolore est ipsam
- 9 nesciunt iure omnis dolorem tempora et accusantium
- 10 optio molestias id quia eum
- 11 et ea vero quia laudantium autem

Demo 8.1: Simple GET Request

```
1 import 'package:flutter/material.dart';
2 import 'package:http/http.dart' as http;
3 import 'dart:convert';
4
5 void main() => runApp(const Demo81App());
6
7 class Demo81App extends StatelessWidget {
8   const Demo81App({super.key});
9
10  @override
11  Widget build(BuildContext context) {
12    return MaterialApp(
13      title: 'Demo 8.1 - Simple GET',
14      theme: ThemeData.dark(useMaterial3: true),
15      home: const SimpleGetScreen(),
16      debugShowCheckedModeBanner: false,
17    );
18  }
19}
20
21 class Post {
22   final int id;
23   final String title;
24
25   Post({required this.id, required this.title});
26
27   factory Post.fromJson(Map<String, dynamic> json) {
28     return Post(
29       id: json['id'] as int,
30       title: json['title'] as String,
31     );
32   }
33 }
34
35 class SimpleGetScreen extends StatefulWidget {
36   const SimpleGetScreen({super.key});
37
38   @override
39   State<SimpleGetScreen> createState() => _SimpleGetScreenState();
40 }
41
42 class _SimpleGetScreenState extends State<SimpleGetScreen> {
43   late Future<List<Post>> _futurePosts;
44
45   void initState() {
46     super.initState();
47     _futurePosts = fetchPosts();
48   }
49 }
```

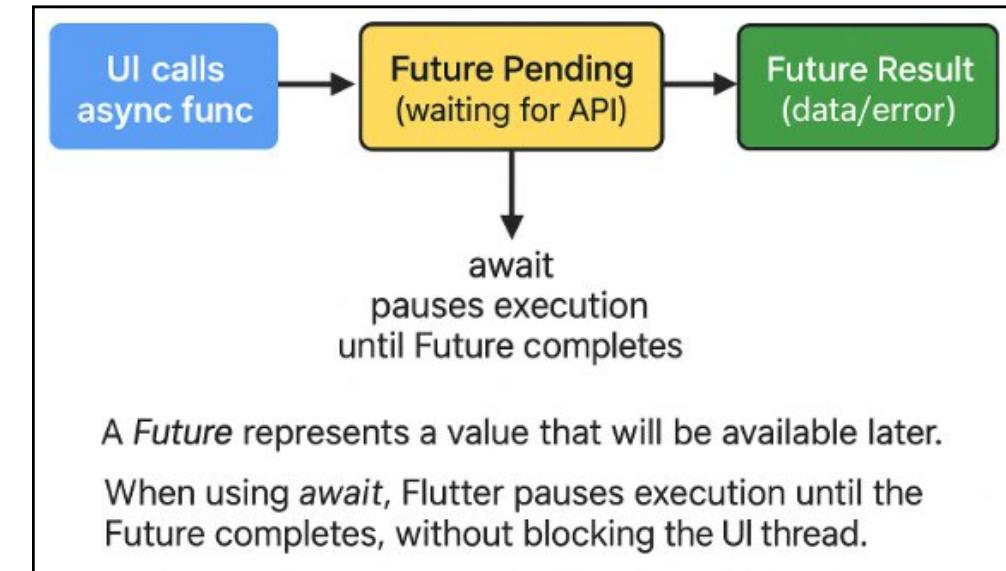
```
48 Future<List<Post>> fetchPosts() async {
49   final url = Uri.parse('https://jsonplaceholder.typicode.com/posts');
50   final response = await http.get(url);
51
52   if (response.statusCode != 200) {
53     throw Exception('Failed to load posts');
54   }
55   final List<dynamic> jsonList = json.decode(response.body);
56   return jsonList.map((item) => Post.fromJson(item)).toList();
57 }
58
59 @override
60 Widget build(BuildContext context) {
61   return Scaffold(
62     appBar: AppBar(title: const Text('Demo 8.1 - Simple GET')),
63     body: FutureBuilder<List<Post>>(
64       future: _futurePosts,
65       builder: (context, snapshot) {
66         if (snapshot.connectionState == ConnectionState.waiting) {
67           return const Center(child: CircularProgressIndicator());
68         } else if (snapshot.hasError) {
69           return Center(
70             child: Text('Error: ${snapshot.error}'),
71           );
72         } else if (!snapshot.hasData || snapshot.data!.isEmpty) {
73           return const Center(child: Text('No posts found'));
74         }
75         final posts = snapshot.data!;
76         return ListView.builder(
77           itemCount: posts.length,
78           itemBuilder: (context, index) {
79             final post = posts[index];
80             return ListTile(
81               title: Text(post.title),
82               leading: CircleAvatar(child: Text(post.id.toString())),
83             );
84           },
85         );
86       },
87     );
88   }
89 }
```

Futures & async/await (Quick Review)

Key Concepts

- Future<T> represents a value that will be available later
- async marks a function as asynchronous
- await pauses execution until a Future completes
- Exam:

```
Future<void> fetchData() async {  
    final result = await getDataFromServer();  
    print(result);  
}
```



Why async matters for Networking?

- Prevents UI freezing while waiting for network responses
- Lets Flutter handle long-running tasks efficiently
- Required for all HTTP operations

FutureBuilder: Binding Data to UI

Why FutureBuilder?

- Integrates async calls into the widget tree
- Handles loading, success, and error states in one place

This FutureBuilder listens to `fetchPosts()` and rebuilds the UI based on the Future state:

- waiting: show a loading spinner
- error: display the error message
- no data: show a “No data” message
- data loaded: build a ListView of post titles

```
FutureBuilder<List<Post>>(  
  future: fetchPosts(),  
  builder: (context, snapshot) {  
    if (snapshot.connectionState == ConnectionState.waiting) {  
      return const Center(child: CircularProgressIndicator());  
    } else if (snapshot.hasError) {  
      return Center(child: Text('Error: ${snapshot.error}'));  
    } else if (!snapshot.hasData || snapshot.data!.isEmpty) {  
      return const Center(child: Text('No data'));  
    }  
    final posts = snapshot.data!;  
    return ListView.builder(  
      itemCount: posts.length,  
      itemBuilder: (_, index) => Text(posts[index].title),  
    );  
  },  
);
```

Parsing JSON into Dart Models

- **Why use model classes?**
 - Type safety
 - Clear structure
 - Easier to maintain and reuse

Example model:

```
class Post {  
    final int id;  
    final String title;  
    final String body;  
  
    Post({  
        required this.id,  
        required this.title,  
        required this.body,  
    });  
  
    factory Post.fromJson(Map<String, dynamic> json) {  
        return Post(  
            id: json['id'] as int,  
            title: json['title'] as string,  
            body: json['body'] as String,  
        );  
    }  
}
```

Demo 8.2 – JsonModelScreen

Objective

- Convert JSON into a Dart model
- Display list items and navigate to a Detail Screen with full data

Steps

- Create a model with fromJson
- Fetch and decode JSON
- Show list of items
- Tap an item → open Detail Screen
- Display selected item's fields

Demo 8.2 – Posts

sunt aut facere repellat provident occaecati
excepturi optio reprehenderit

qui est esse

ea molestias quasi exercitationem repellat qui ipsa
sit aut

eum et est occaecati

nesciunt quas odio

dolorem eum magni eos aperiam quia

magnam facilis autem

dolorem dolore est ipsam

nesciunt iure omnis dolorem tempora et
accusantium

optio molestias id quia eum

et ea vero quia laudantium autem

Converting JSON List to List<Model>

- **Key Idea**
 - Convert a JSON array (list of objects) into a typed List<Model> using decoding + mapping.
- **JSON → Dart Conversion**

Concepts

- JSON list = array of multiple objects
- jsonDecode() → returns List<dynamic>
- .map() → converts each JSON item to a Model
- .toList() → final typed list (List<Post>)

```
final List<dynamic> jsonList = jsonDecode(response.body);

final posts = jsonList
    .map((e) => Post.fromJson(e))
    .toList();      // List<Post>
```

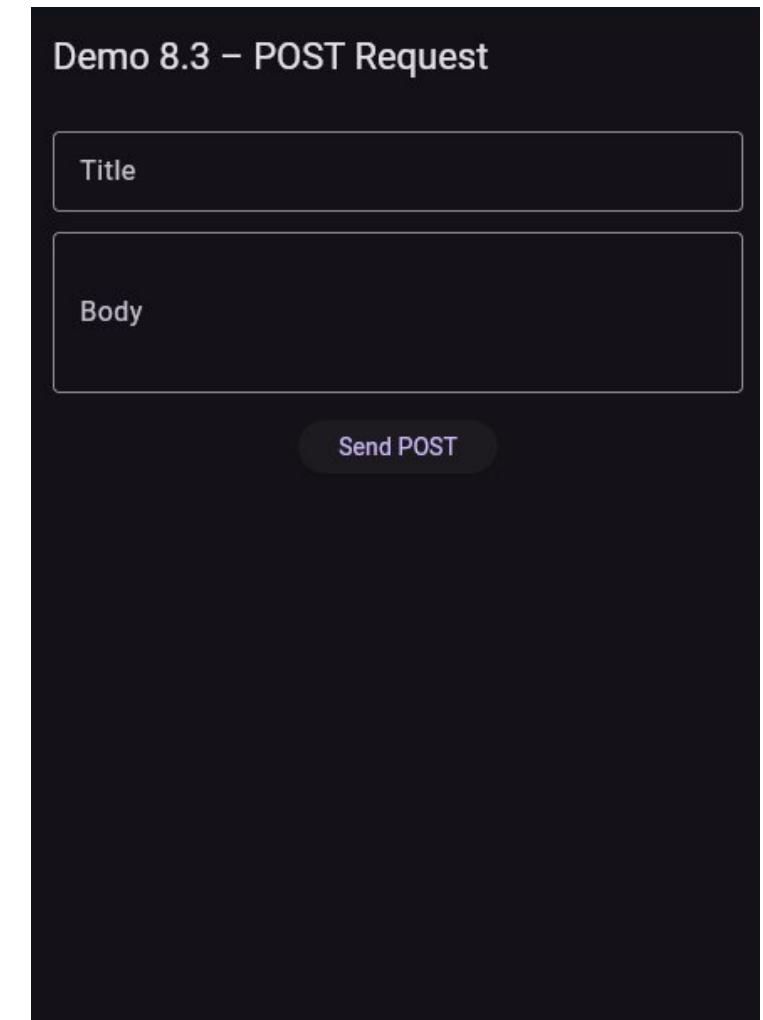
Demo 8.3: Sending Data via POST

Objectives

- Send data to an API using POST
- Show loading, success, and error states
- Allow user to retry on failure

Steps

1. Build a simple input form
2. Send POST request (`http.post`)
3. Encode JSON body
4. Show loading indicator
5. Handle error → show message + Retry button
6. Display success response



Error Handling & Retry

- **Common failure cases**
 - No internet connection
 - Server error (500)
 - Invalid JSON
 - Timeout
- **Basic strategies**
 - Throw exceptions in fetch functions
 - Show friendly error messages in UI
 - Offer a “Retry” button

Simple UI example:

```
if (snapshot.hasError) {  
  return Column(  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: [  
      Text('Error: ${snapshot.error}'),  
      const SizedBox(height: 8),  
      ElevatedButton(  
        onPressed: () => setState(() {}),  
        child: const Text('Retry'),  
      ),  
    ],  
  );  
}
```

POST Request + Error Handling Pattern

Explanation

- This block handles errors during the POST request.
- If the FutureBuilder detects an error (snapshot.hasError), it shows:
 - The error message
 - A Retry button that triggers setState() to run the request again
- It provides simple error feedback and a way for the user to retry the API call.

Basic pattern:

```
Future<void> createPost(String title, String body) async {  
    final url = Uri.parse('https://jsonplaceholder.typicode.com/posts');  
    final response = await http.post(  
        url,  
        headers: {'Content-Type': 'application/json; charset=UTF-8'},  
        body: json.encode({  
            'title': title,  
            'body': body,  
            'userId': 1,  
        }),  
    );  
  
    if (response.statusCode == 201) {  
        print('Post created: ${response.body}');  
    } else {  
        throw Exception('Failed to create post');  
    }  
}
```

Loading & Button States (UX)

Good UX patterns

- Disable submit button while request is in progress
- Show a loading indicator (spinner)
- Prevent double-submit

Key idea:

- Avoid double submit while loading

```
ElevatedButton(  
    onPressed: isSubmitting ? null : _submit,  
    child: isSubmitting  
        ? const SizedBox(  
            width: 16,  
            height: 16,  
            child: CircularProgressIndicator(strokeWidth: 2),  
        )  
        : const Text('Submit'),  
);
```

Demo 8.4 – ApiService

Objectives

- Move HTTP logic into a separate service
- Keep UI clean and focused
- Reuse the same service across screens

Steps

1. Create Post model
2. Implement ApiService.fetchPosts()
3. Call service in initState()
4. Use FutureBuilder to load + show list

Demo 8.4 – ApiService

- 1 sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- 2 qui est esse
- 3 ea molestias quasi exercitationem repellat qui ipsa sit aut
- 4 eum et est occaecati
- 5 nesciunt quas odio
- 6 dolorem eum magni eos aperiam quia
- 7 magnam facilis autem
- 8 dolorem dolore est ipsam
- 9 nesciunt iure omnis dolorem tempora et accusantium
- 10 optio molestias id quia eum
- 11 et ea vero quia laudantium autem
- 12 in quibusdam tempore odit est dolorem

Organizing Networking Code (Demo 8.4)

Service Layer Pattern

- Create a class to encapsulate API logic
- Keep widget code clean and focused on UI

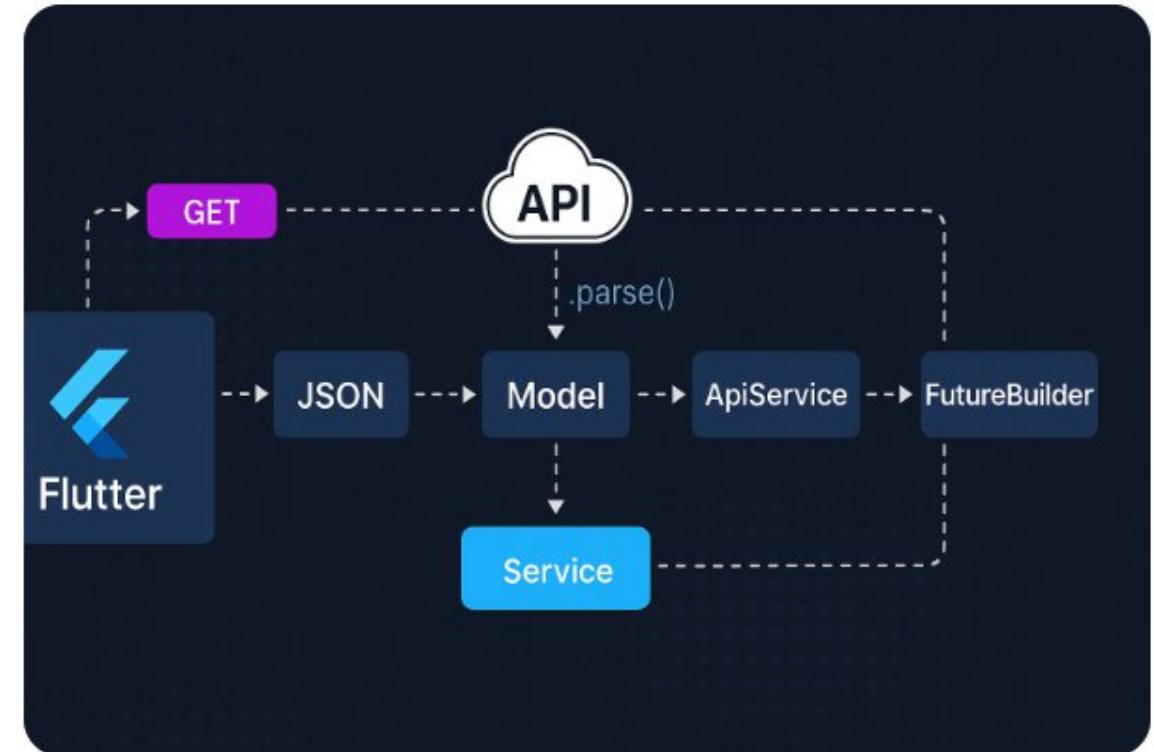
Example:

```
class ApiService {  
    final http.Client client;  
  
    ApiService({http.Client? client}) : client = client ?? http.client();  
  
    Future<List<Post>> fetchPosts() async {  
        final url = Uri.parse('https://jsonplaceholder.typicode.com/posts');  
        final response = await client.get(url);  
        // ...parse & return list of Post  
    }  
  
    Future<Post> createPost(String title, String body) async {  
        // ...call POST and return created Post  
    }  
}
```

From API to UI: Putting It All Together

End-to-end flow

- User opens screen
- App calls ApiService.fetchPosts()
- FutureBuilder shows loading → data / error
- User taps “Add” and opens form
- Form submits createPost()
- On success: show message, update list (re-fetch or optimistic update)



Custom Flutter API Pipeline Diagram PNG

Common Pitfalls & Best Practices

Pitfalls

- Calling APIs in build() instead of initState()
- Not handling errors → app crashes
- Blocking the UI thread with heavy parsing
- Hardcoding URLs everywhere

Best Practices

- Call API once in initState()
- Centralize base URL and endpoints
- Wrap networking in service classes):
- Log responses during development

Lab 8 Overview

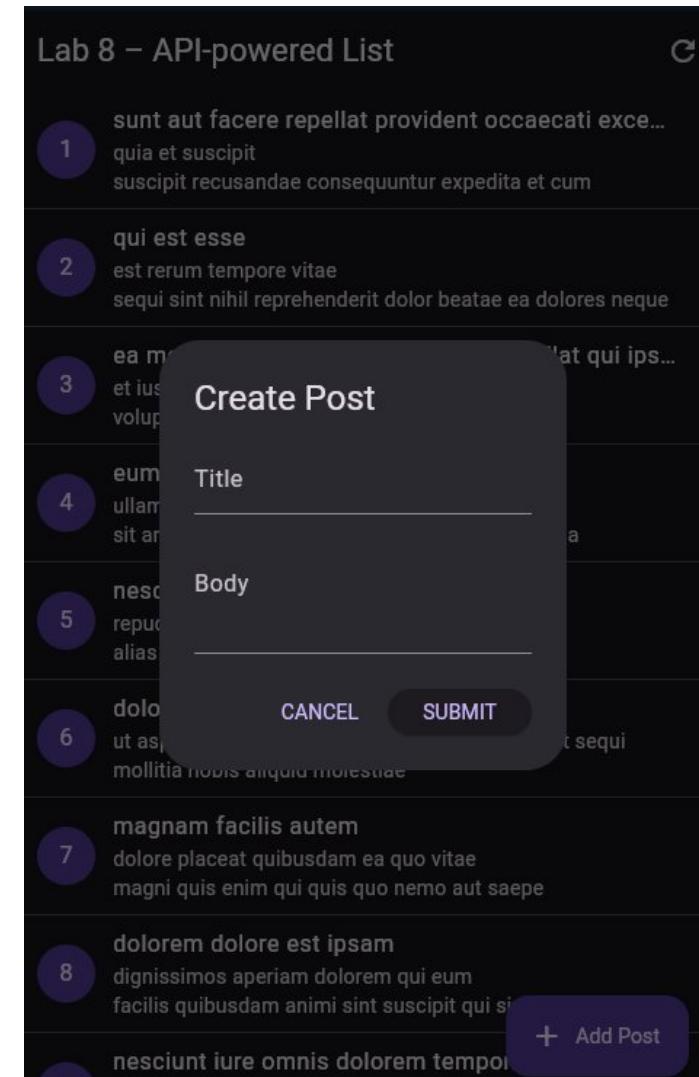
Objectives: Building a Simple API-powered List Screen

You will:

1. Call a public REST API
2. Parse JSON into model classes
3. Display a list of items using ListView
4. Show loading + error states

(Optional) Add a simple POST form to create a new item

Note: *This lab combines all concepts from Demos 8.1 → 8.4*



Summary

In this module, you learn how to:

- Understand how RESTful APIs work
- Send HTTP GET and POST requests in Flutter
- Decode JSON strongly into typed Dart model classes
- Display API data with FutureBuilder and ListView
- Implement loading, empty, and error UI states
- Organize networking code into a clean, reusable service layer

(Optional): Build a simple form that sends data via POST

References

- Mastering Flutter 2025 – Ch.3, 4, 9
- docs.flutter.dev
- <https://dartpad.dev>

Module 9 - Local Storage & Persistence in Flutter

Learning Objectives

Content:

- Understand the 3 main local storage methods in Flutter
- Store & retrieve key–value data with SharedPreferences
- Read & write local JSON files using Dart IO
- Build CRUD operations using SQLite (sqflite)
- Apply offline-first strategies to improve user experience
- Combine multiple storage methods into a practical mini-app

Why Local Storage?

Key Concepts

- Local storage is data saved directly on the user's device:
- Available offline
- Persistent across restarts
- Faster than API calls
- Stores personalized user choices

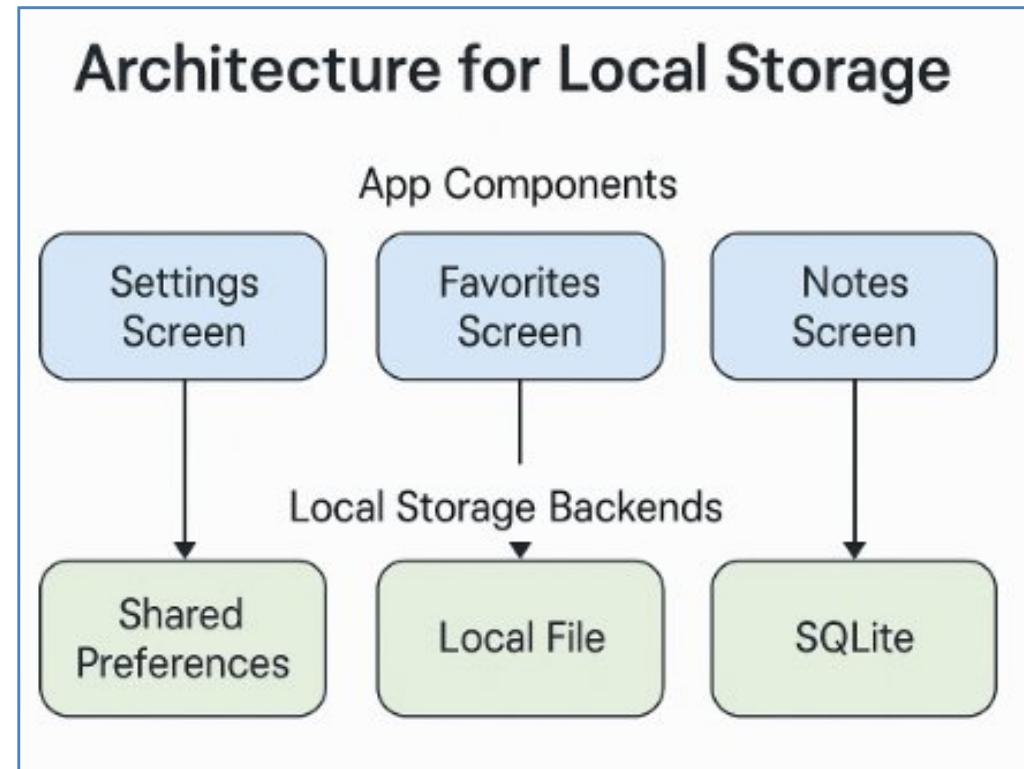
Examples:

- Dark mode settings
- Recently viewed items
- Saved filters
- Favorites/watchlist
- Onboarding flag (“first-time user?”)

Local storage = better UX + higher app reliability.

Types of Local Storage in Flutter

Storage Type	Best Use	Example
SharedPreferences	Simple key–value	Settings, flags
JSON File (dart:io)	Simple lists, export	Notes, history
SQLite (sqflite)	Structured relational	Todo app



What is SharedPreferences?

Lightweight and easy to use

- Stores simple key-value pairs
- Persistent across app restarts
- Works synchronously and asynchronously
- NOT for complex objects or large data

Example use cases:

- Dark/Light theme preference
- “Onboarding completed” flag
- User login state
- Default filter/sort mode

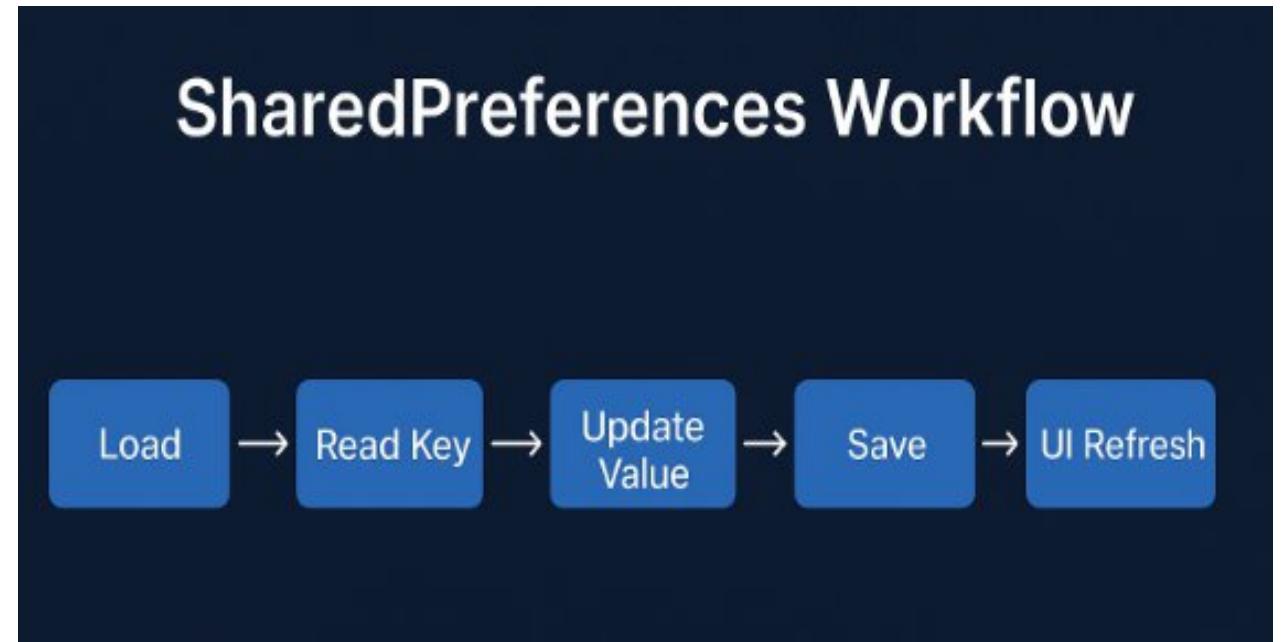
SharedPreferences Workflow

What it does

- Stores simple key–value data on the device.
- Data persists even after the app is closed or restarted.

Process

- Load SharedPreferences instance
- Read stored values
- Update values when the user interacts
- Save the new key–value pairs
- Refresh UI to reflect updated data



SharedPreferences Support Across Environments

Environment	SharedPreferences Support?	Notes
Flutter App (Android/iOS)	✓ Yes	Saves data to the device's persistent storage. Reopen the app → data remains.
Flutter Web	✓ Yes (via localStorage)	SharedPreferences automatically maps to browser localStorage.
DartPad (Flutter Mode)	✗ Not supported	No native storage → must use a FakePrefs mock class.
DartPad (Dart Mode)	✗ No	No plugins and no Flutter widgets. Pure Dart environment only.

Demo 9.1: Saving App Settings

OBJECTIVES

- Understand how to store simple data using SharedPreferences.
- Save theme state (dark/light) and retain it when reopening the app.
- Practice the lifecycle: load → update → save → refresh UI.
- Compare how to use SharedPreferences on a real device and when there is no native storage (DartPad).

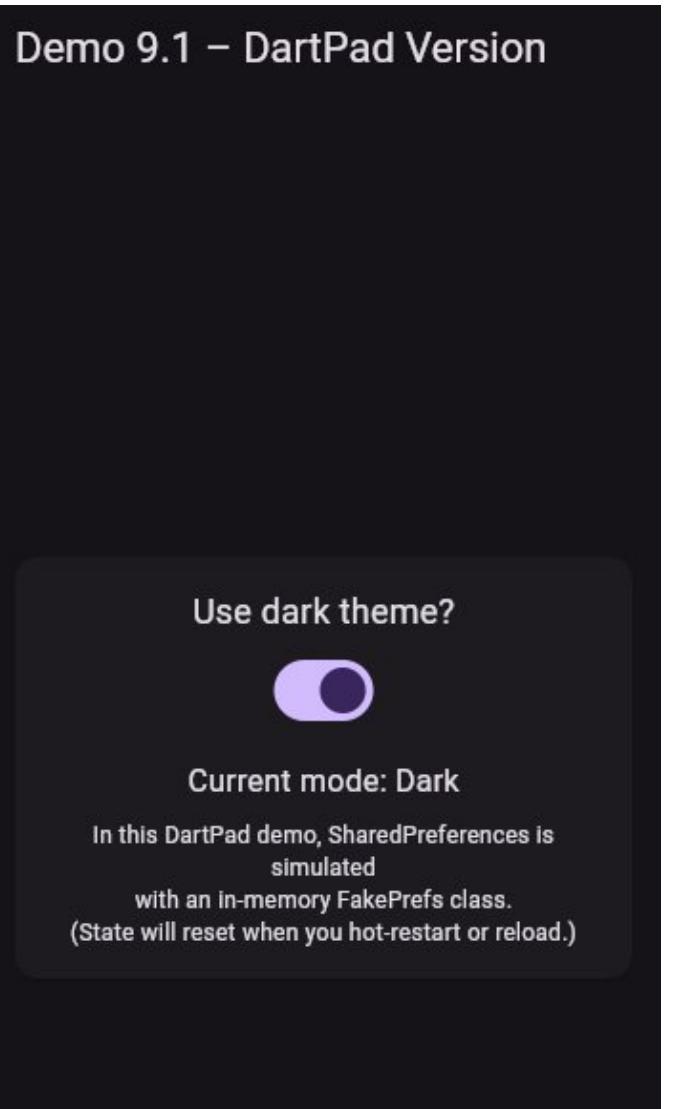
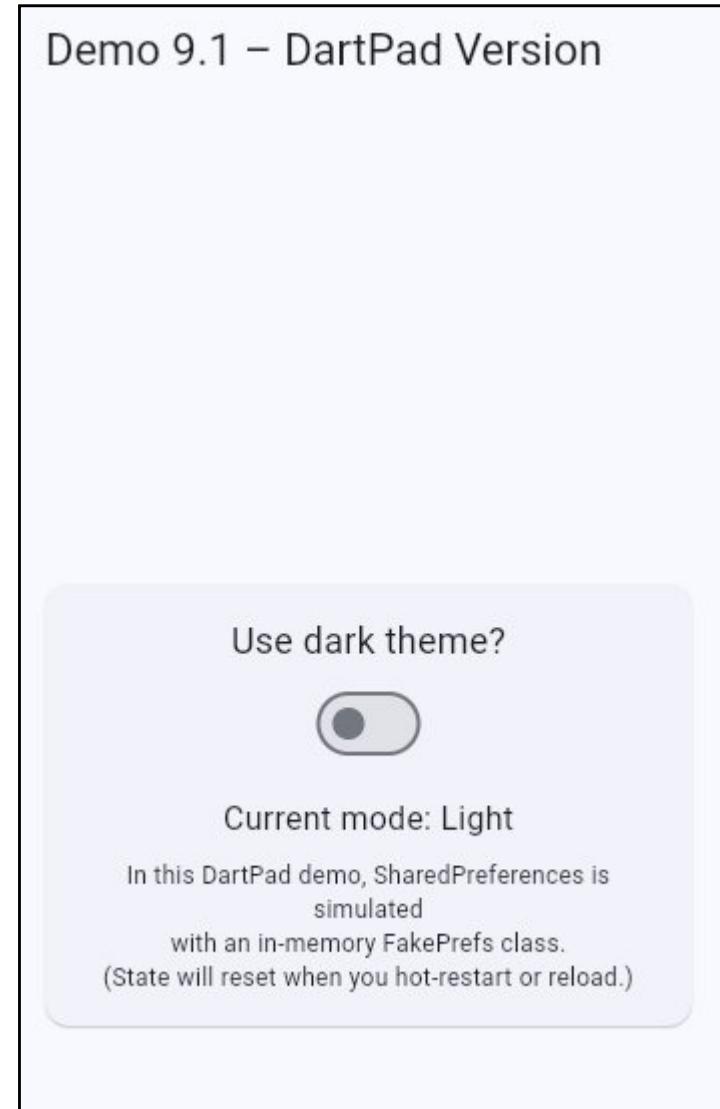
Steps

1. Add dependency: shared_preferences: ^2.x.x
2. Load saved theme: use SharedPreferences.getInstance() and read "isDarkMode"
3. Update state: toggle the switch to change the theme
4. Save value: prefs.setBool("isDarkMode", value)
5. Refresh UI: call setState() (or use a state management tool)
6. Restart app: the saved theme is restored automatically

Demo 9.1: Saving App Settings

Steps (DartPad Version)

1. No SharedPreferences plugin in DartPad
→ use a mock class (FakePrefs)
2. Load initial value from an in-memory map
3. Update state when the switch is toggled
4. Save value back into the mock storage
5. Refresh UI using setState()
6. Note: data resets on reload (not persistent)



Why Use Local Files? File Storage Basics

You need files when:

- Data is bigger than key–value storage
- Data has structure but doesn't require SQL
- You want simple offline documents
- You want to export/import JSON
- You want to cache API responses locally

Key packages:

- path_provider → find app directories
- dart:io → read/write
- json.encode / json.decode

Short snippet:

```
final file = File(path);  
await file.writeAsString(jsonString);  
final content = await file.readAsString();
```

Key-Value Persistence Overview

What Is Key-Value Storage?

- Lightweight way to store small app settings
- Data saved by key → value pairs
- Used for: theme mode, username, onboarding flags, user preferences

How It Works Across Environments

- Flutter App (Android/iOS): SharedPreferences → device storage
- Flutter Web: maps to browser localStorage
- DartPad: plugins not supported → use custom FakePrefs

High-Level Flow

- Load storage instance ↗ Read existing value ↗ Update value based on user action ↗ Save back to storage ↗ Refresh the UI

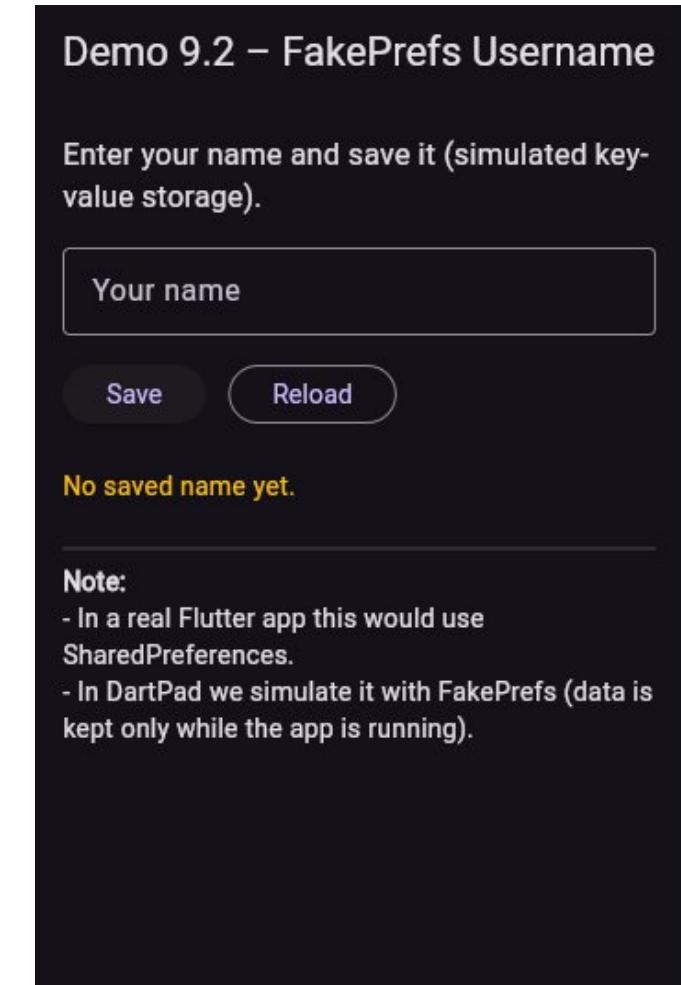
Demo 9.2: Key-Value Storage (DartPad Version)"

Objective

- Simulate key-value storage using a mock class (FakePrefs)
- Save and load a simple value (e.g., username)

Steps

1. Create a FakePrefs class to simulate SharedPreferences
2. Initialize storage using FakePrefs.getInstance()
3. Load saved value into the TextField on startup
4. Update value when user edits the input
5. Save value using setString(key, value)
6. Refresh UI with setState()



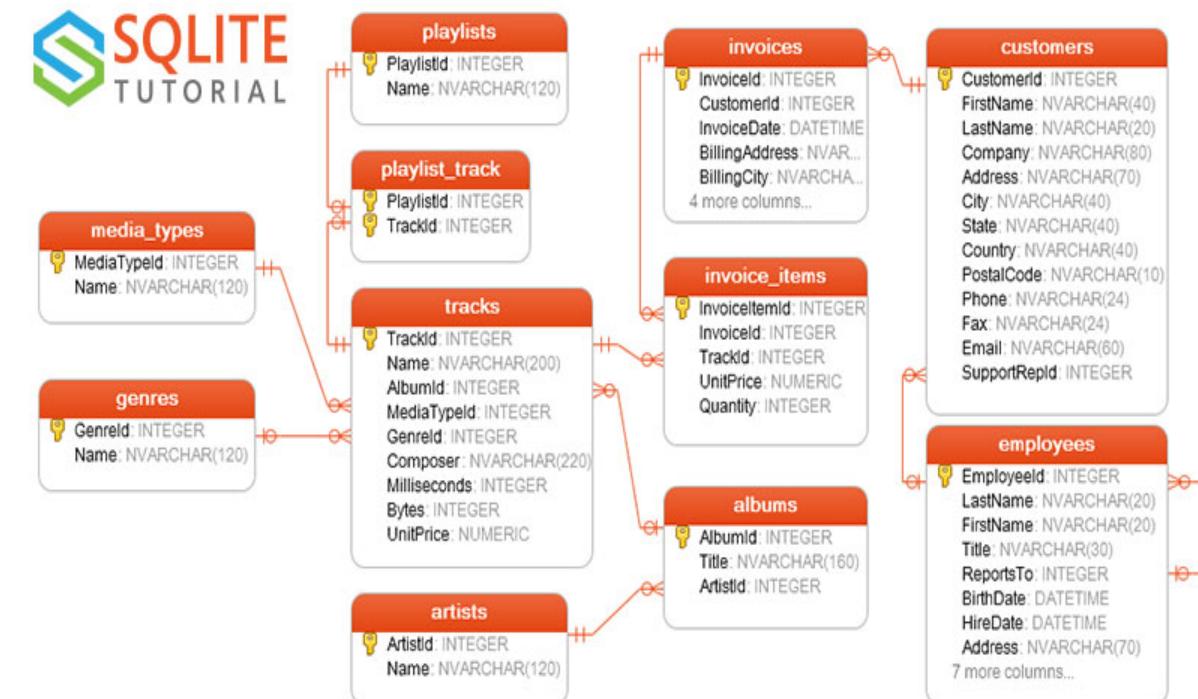
Why SQLite?

Use SQLite when your app needs:

- Structured data (rows, columns)
- Fast read/write operations
- Sorting & filtering (ORDER BY, WHERE)
- Data relationships (1–N, N–N)
- Long-term reliable storage
- Offline capability

Real Use Cases

- To-Do / Notes app - Inventory tracking
- Offline e-commerce cart - Local user profile database - Bookmark / favorites storage



SQFLite vs SQLite: What's the Difference?

Why not name it SQLite?

- SQLite = the actual storage engine
- SQFlite = Flutter SDK wrapper built on top of SQLite
- The “F” in SQFlite = Flutter (SQLite for Flutter)

Why this matters?

- SQFlite is not a database, but a plugin
- Helps clarify why SQFlite doesn't work on Web
- Explains that database operations still follow real SQL rules

How is SQFlite different from SQLite?



SQLite is the database engine; SQFlite is a Flutter plugin that interfaces with SQ.



SQLite vs SQFlite

Component	What it is	Role
SQLite Engine	A lightweight relational database engine	Stores and queries data locally
SQFlite Plugin	A Flutter plugin written in Dart + native code	Bridges Flutter → SQLite
App UI (Flutter Widgets)	Your application screens	Sends commands to SQFlite
Database File (.db)	Local file stored on the device	Holds tables & records

SQFlite Basics (CRUD + Setup)

Key points

- sqflite = SQLite database for Flutter (Android, iOS)
- Work with Database object:
 - openDatabase() – open/create DB
 - onCreate – create tables (run CREATE TABLE)
 - insert() – add a row
 - query() – read rows
 - update() – change rows
 - delete() – remove rows

```
// 1. Open database
final db = await openDatabase(
  'notes.db',
  version: 1,
  onCreate: (db, version) {
    return db.execute('''
      CREATE TABLE notes(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT,
        content TEXT
      )
    ''');
  },
);

// 2. Insert
await db.insert('notes', {
  'title': 'First note',
  'content': 'Hello SQLite'
});

// 3. Query
final rows = await db.query('notes'); // List<Map<String, Object?>>
```

SQL SELECT Recap (for SQFlite)

Syntax block

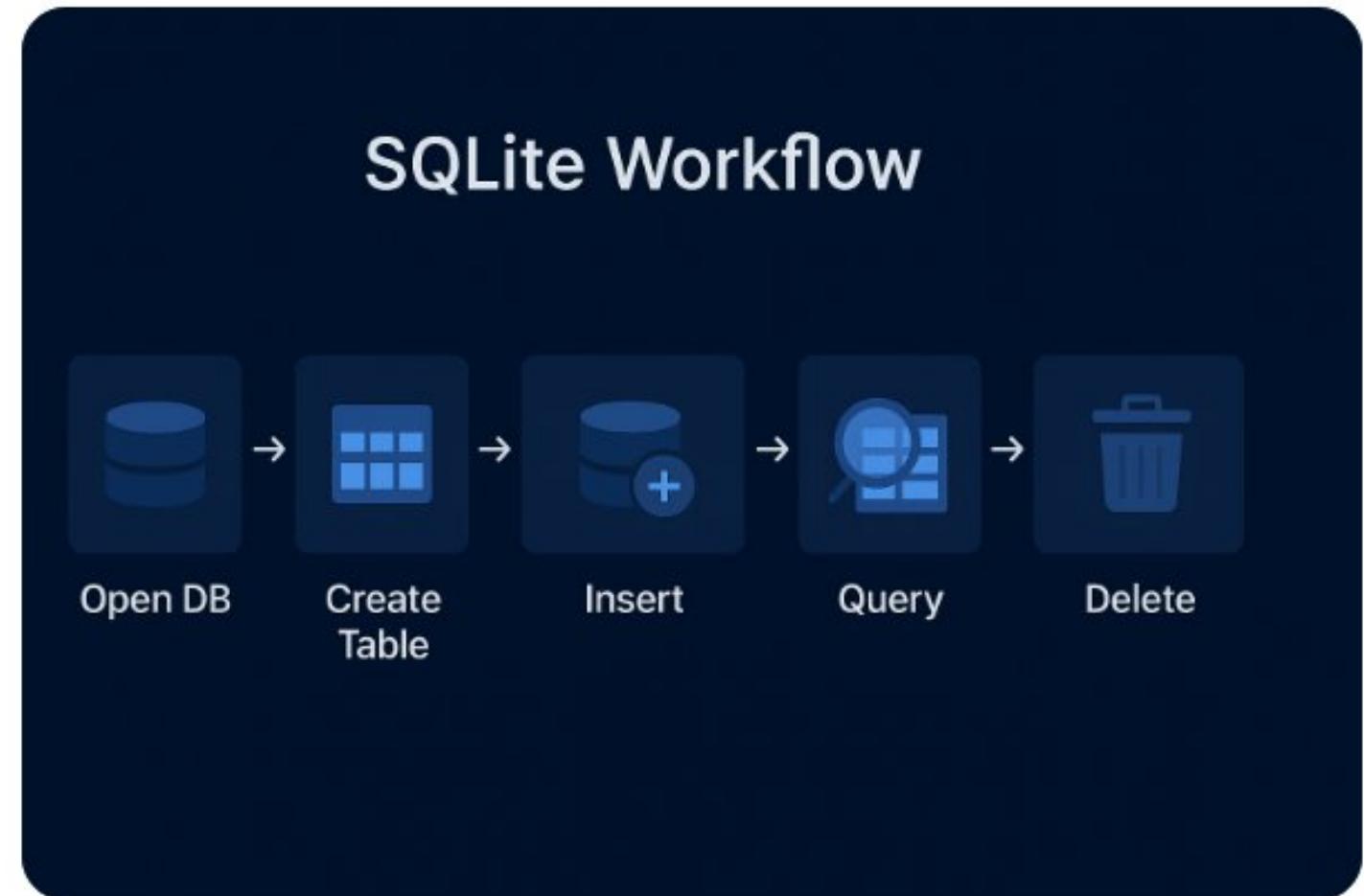
```
SELECT  column_list / *
FROM    table_name
WHERE   condition
GROUP BY group_columns
HAVING  group_condition
ORDER BY sort_column [ASC|DESC]
LIMIT   n   OFFSET m;
```

- SELECT – which columns to return
- FROM – which table
- WHERE – filter rows
- GROUP BY + HAVING – group & filter groups (for SUM, COUNT, ...)
- ORDER BY – sort result
- LIMIT / OFFSET – pagination (useful for mobile lists)

Database Workflow Diagram

Why this matters

- SQLite operations follow a predictable lifecycle.
- Understanding the flow helps reduce bugs and avoid bad database practices.
- Applies to: SQFlite, Moor/Drift, Hive (conceptually), and most SQL engines.



Demo 9.3: Offline Todo App (SQLite)

Objective

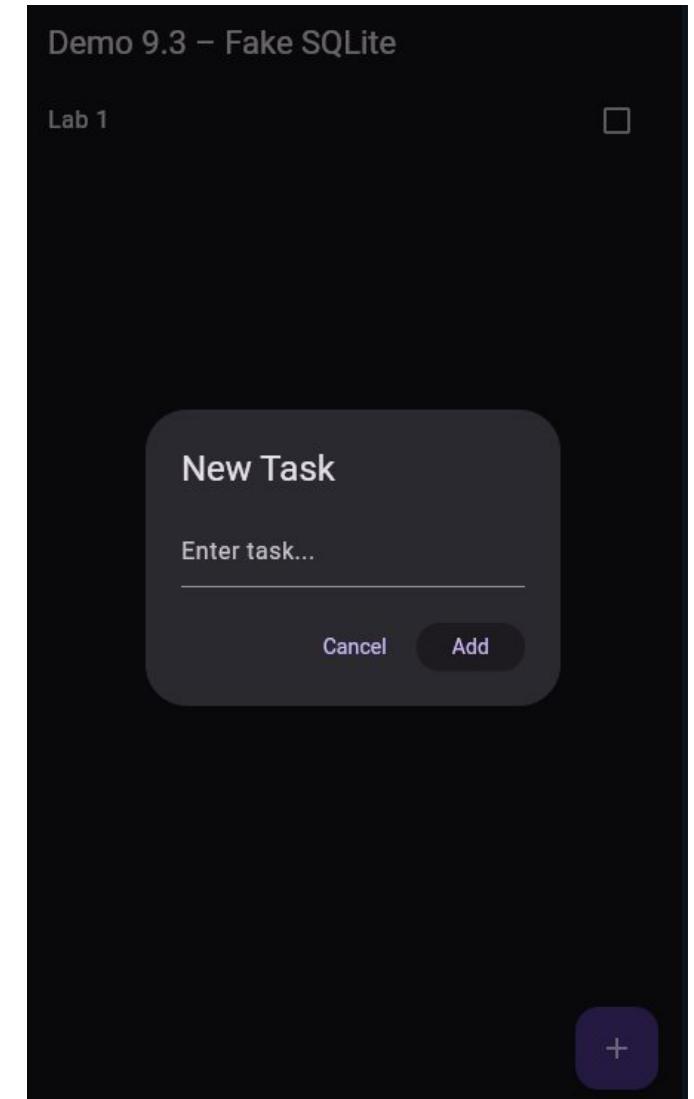
Build a simple offline Todo list.

Simulate SQLite CRUD using an in-memory FakeDatabase

Practice insert, query, update, delete before real SQFLite

Steps

1. Create Todo model
2. Implement FakeDatabase (insert/query/update/delete)
3. Load todos on startup
4. Add new todo (dialog → insert)
5. Toggle completed (update)
6. Delete task (Dismissible → delete)
7. Refresh UI after each operation



Offline-First Concept

Offline First = App still works when offline

Key Idea:

- Load local data first (SharedPreferences / JSON / SQLite)
- Sync server when online (Module 10)
- Minimize errors, increase user experience

How it works:

- Store offline → Use offline → Sync online
- Resolve conflicts
- Cache expiration

When to Use Which Storage?

Rules of thumb:

- Key-value → SharedPreferences
- Small list → JSON file
- Complex/multi-screen data → SQLite
- Need sync → SQLite + backend

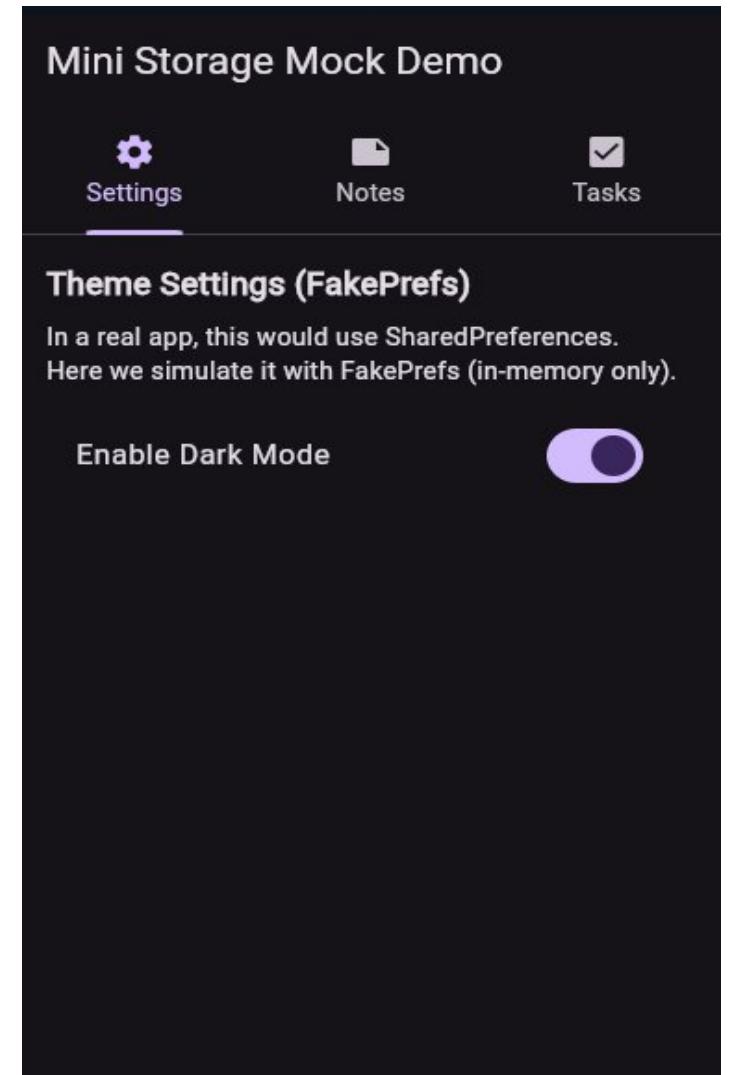
Which storage for which use case?

Use Case	Recommended
Theme, small flags	SharedPreferences
Simple notes, 1 file	Local JSON file
Large structured data	SQLite (SQFlite)
Offline-first + Sync	SQLite + Firebase/REST

Lab 9 Overview (Offline-First Mini App)

Objectives: Build an Offline-First Multi-Storage App

- You will:
 - Use SharedPreferences to save app theme (Dark/Light mode)
 - Save & load notes using a local JSON file
 - CRUD tasks using SQLite (or FakeSQLite)
 - Handle UI refresh after local storage updates
 - Understand how three storage layers can work together
- Note:
 - *This lab combines concepts from all Demos 9.1 → 9.3.*



Summary

You learned the three main local-storage methods in Flutter:

SharedPreferences, Local JSON files, and SQLite/SQFlite.

- SharedPreferences: best for small key-value data (theme, flags), fast and simple.
- JSON files: good for document-style data (notes, cache), flexible and easy to import/export.
- SQLite/SQFlite: ideal for large, structured data with full CRUD and query support.
- Demos 9.1 → 9.3 built an Offline-First mini app: save theme → save notes → manage todos with SQLite.

References

- Mastering Flutter 2025, Chapters 9, 11
- docs.flutter.dev
- <https://dartpad.dev>
- path_provider Flutter plugin
- SharedPreferences Flutter plugin

Module 10 - Authentication, Session Management & Notifications

Learning Objectives

Content:

- Backend authentication
- Token-based login (JWT)
- Sessions & secure token storage
- Login + Signup flow
- Auto-login & logout
- Protected routes
- Introduction to Firebase Auth
- Notifications after authentication (Local Notification)

What is Authentication?

Authentication = verifying who the user is.

Used to:

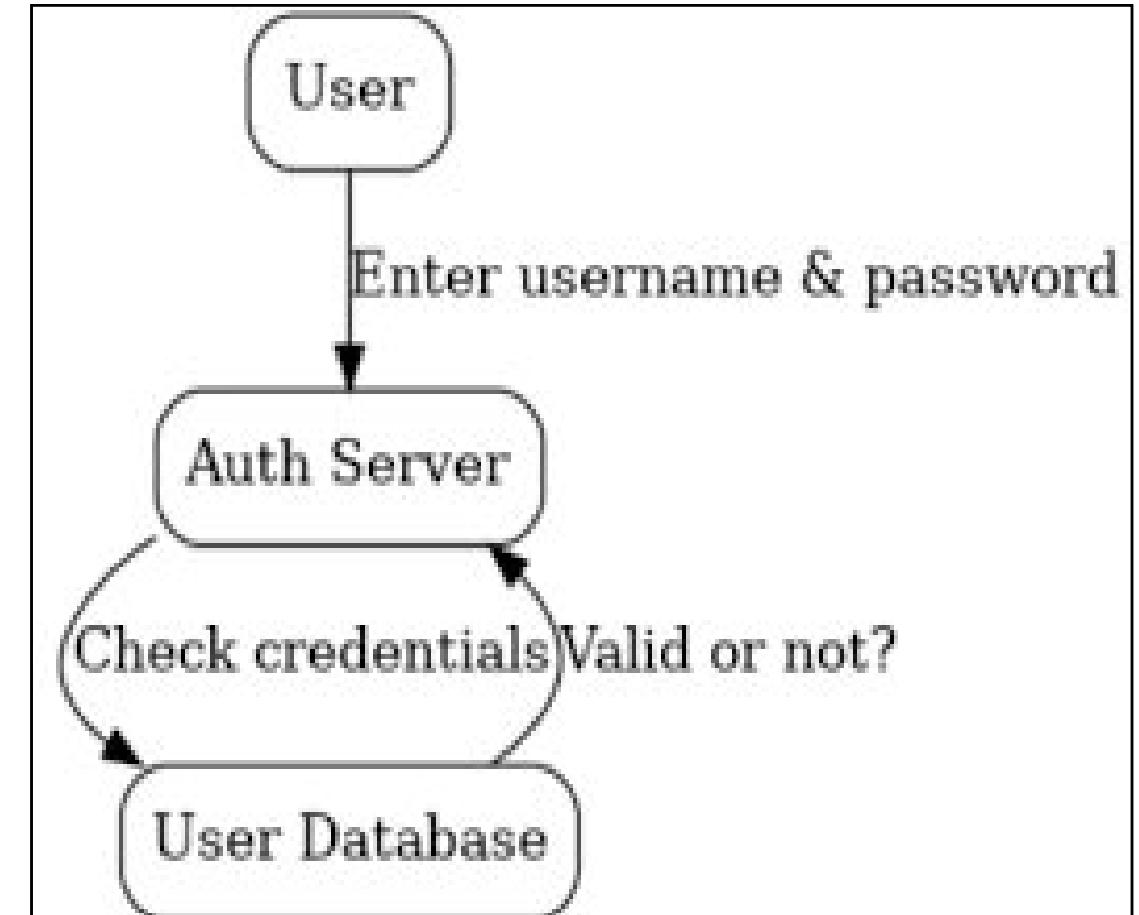
- Protect user accounts
- Allow personalized features
- Control access to private data

Authentication vs Authorization

- Authentication → Who are you?
- Authorization → What are you allowed to do?

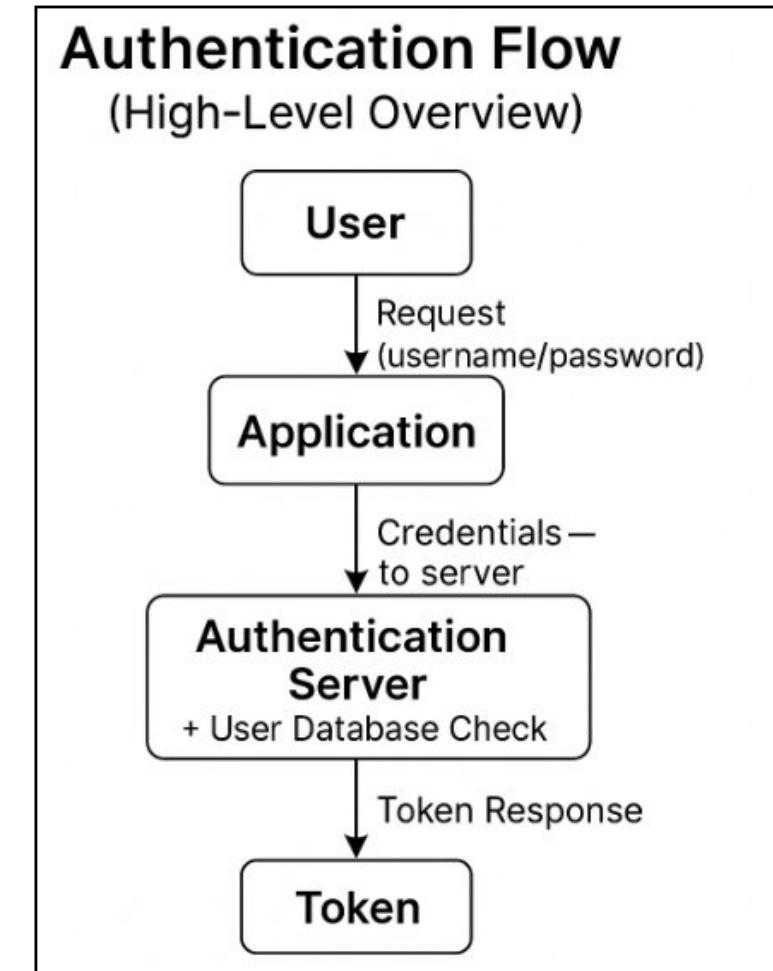
Use real examples:

- Shopee, GrabFood, Tiki.



Authentication Flow (High-Level)

- User enters credentials
- App sends API request
- Backend verifies
- Backend returns token (JWT)
- App stores token locally
- App uses token on every API request
- Session maintained until logout or token expires



Token-Based Authentication

What is a token?

- A token is a short-lived digital credential generated by the server to prove a user's identity.

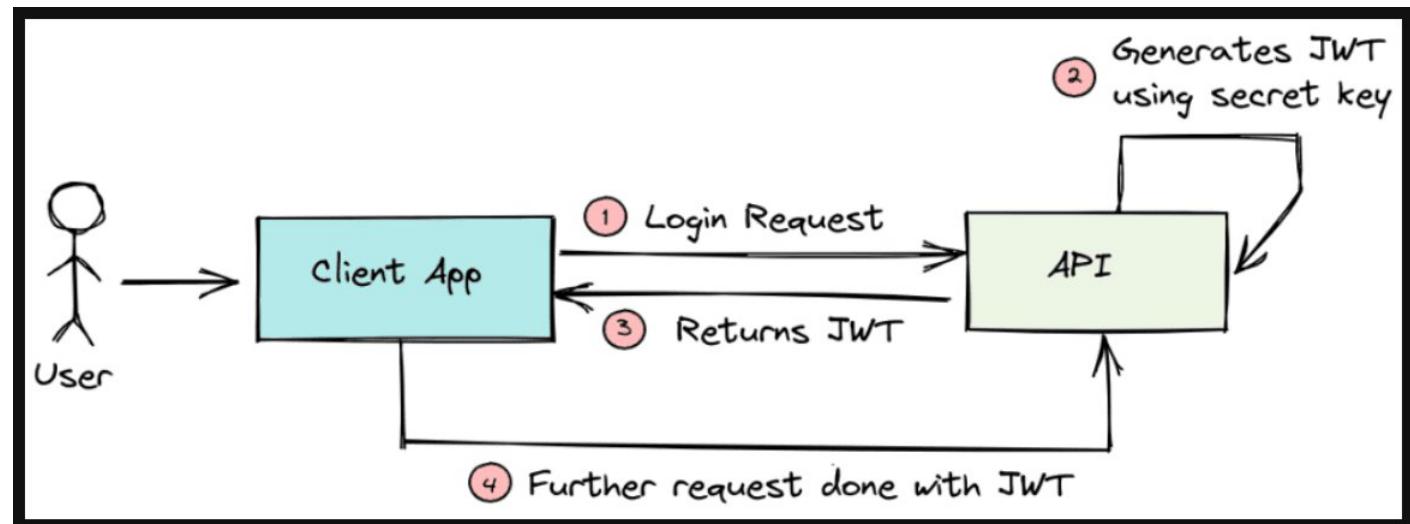
Benefits

- Stateless
- Secure
- Works across mobile/web
- Fast

Types

- Access tokens
- Refresh tokens

Token Flow Diagram (Recommended)

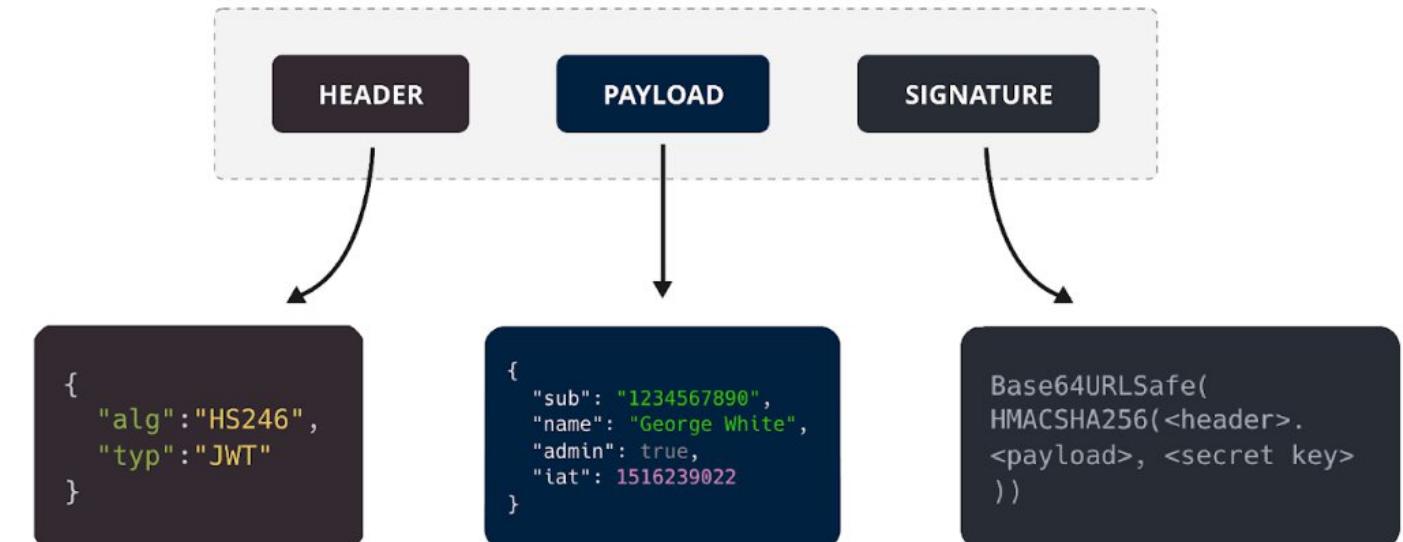


1. User sends username/password → server checks
2. Server returns Access Token + Refresh Token
3. Application sends Access Token in each request
4. When Access Token expires → send Refresh Token to get new

JWT (JSON Web Token) Explained

- **Structure of a JWT:**
 - Header (algorithm, type)
 - Payload (user ID, email, role)
 - Signature (verification)
- **Properties:**
 - Base64 encoded
 - Signed (not encrypted)
 - Has expiry time

Structure of a JSON Web Token (JWT)



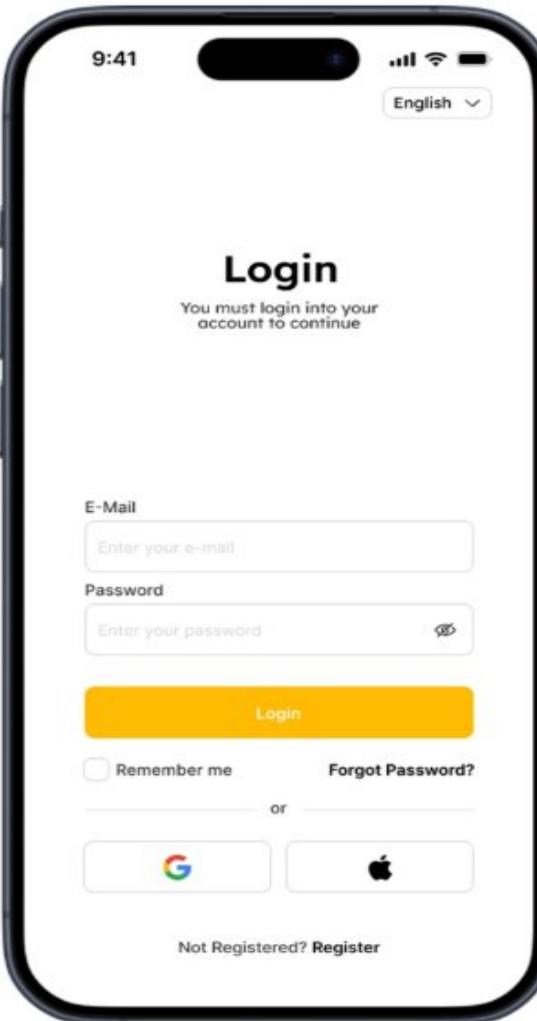
Login Screen UI Structure

Components

- Email TextField
- Password TextField
- Login Button
- “Forgot Password?”
- Navigation to Signup

Good Practices

- Inline field validation
- Disable button while loading
- Hide/show password



Designing Login Form

Email Field

- Type: email
- Validator: regex or basic contains '@'

Password Field

- obscureText: true
- Min 6 characters

Buttons

- Primary submit
- Secondary link to signup

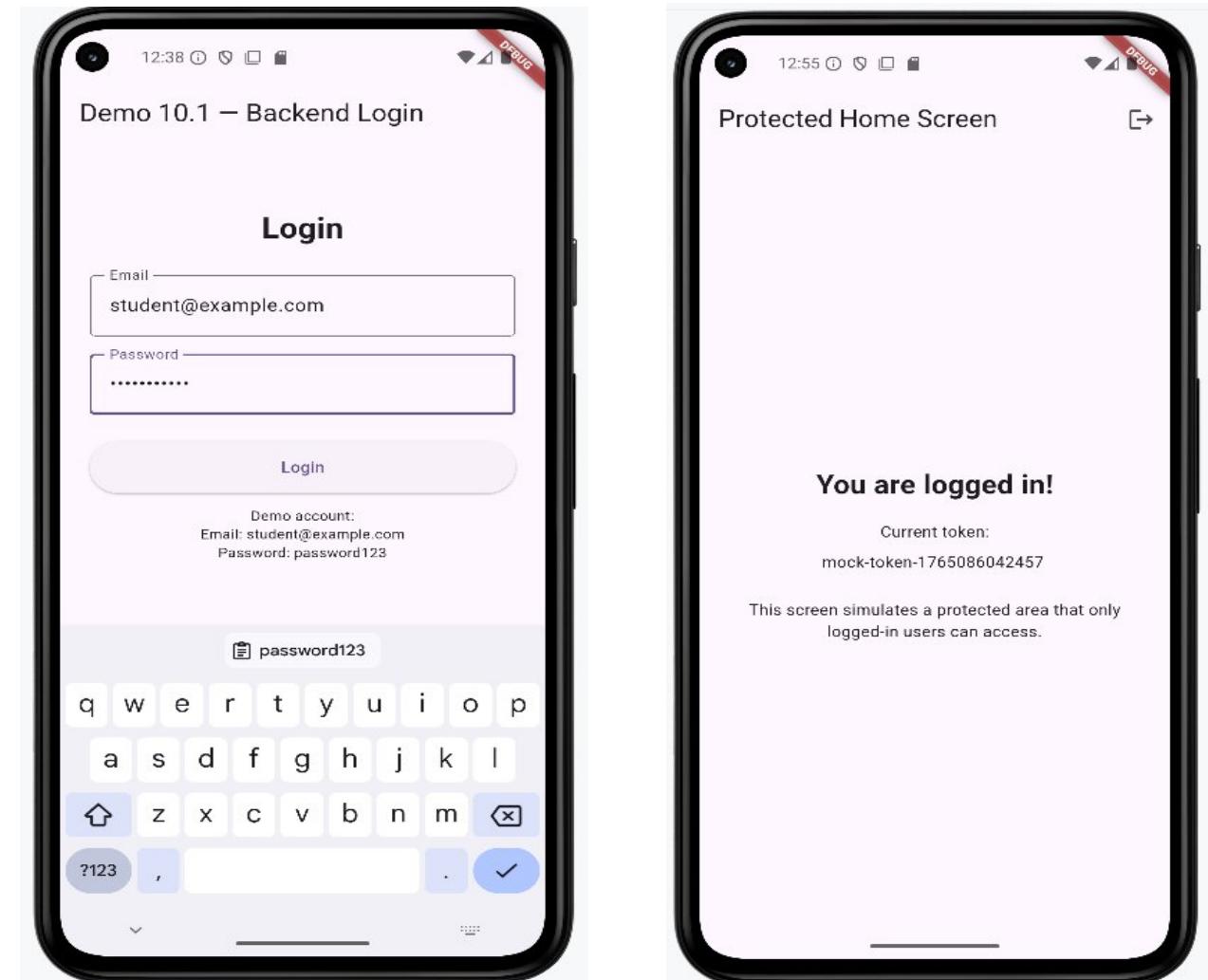
Demo 10.1 - Backend Login Flow (Overview)

Objectives

- Build a login screen with email + password fields
- Validate inputs using Flutter Form
- Simulate backend authentication
- Receive & store a mock token
- Navigate to a protected screen

Step-by-Step Flow

- Step 1 — Build UI
- Step 2 — Simulate backend request
- Step 3 — Handle mock response
- Step 4 — Navigate to protected screen



Step 2 - Simulated Authentication Request

Because public API services may fail or block emulators,

Demo 10.1 uses a mock backend inside AuthService.

What Happens Here?

- The app pretends to send credentials to a server
- Adds a 2-second delay to mimic real network latency
- Checks if email/password match the demo account
- Returns a mock JWT token when valid

Code Snippet

```
await Future.delayed(const Duration(seconds: 2));  
if (email == correctEmail && password == correctPassword) {  
    return "mock-token...";  
}  
return null;
```

Step 3 - Authentication Result Processing

If credentials are valid

- Receive mock token
- Save token inside AuthService
- Prepare for navigation

If invalid

- Return null
- Show error Snackbar

Code Snippet

```
final token = await _auth.login(email, password);

if (token != null) {
    Navigator.pushReplacementNamed(context, '/home');
} else {
    ScaffoldMessenger.of(context)
        .showSnackBar(SnackBar(content: Text("Invalid credentials")));
}
```

Error Handling (Step 1 + Step 3)

Error & Validation Handling

- Client-side Validation (Step 1)

```
if (!_formKey.currentState!.validate()) return;
```

- Common validation errors:
 - Empty email/password
 - Invalid email format
- Authentication errors (Step 3)

```
if (token == null) {  
    Snackbar("Invalid credentials");  
}
```

A screenshot of a mobile application's login screen. It features two input fields: 'Email' containing 'Student@example.com' and 'Password' which is currently empty. Below the password field is a red error message: 'Password Is required'. At the bottom is a light blue 'Login' button.

A screenshot of the same mobile application's login screen. The 'Email' field still contains 'Student@example.com'. The 'Password' field now has a single dot ('.') entered. A red error message below the field reads 'At least 6 characters'. The 'Login' button remains at the bottom.

Mock API vs Real API in Authentication

Why Two Versions?

In Demo 10.1, we use Mock Authentication to ensure that the demo works in all environments.

The Real API version is required to prepare students for real backend integration.

When to Use Which Version?

- Mock API
 - Classroom demonstrations
 - Early-stage learning
 - When students use unstable devices
- Real API
 - Assignments
 - Capstone projects
 - When the real HTTP request/response pattern

Comparison Table

Feature	Mock API	Real API
Internet needed	✗ No	✓ Yes
Error-free	✓ Always	✗ May fail (network, server, TLS, CORS)
Works on all devices	✓ Yes	✗ Not always
Predictability	✓ High	✗ Medium
Good for teaching basics	✓ Yes	✓ Yes
Good for real project	✗ No	✓ Yes

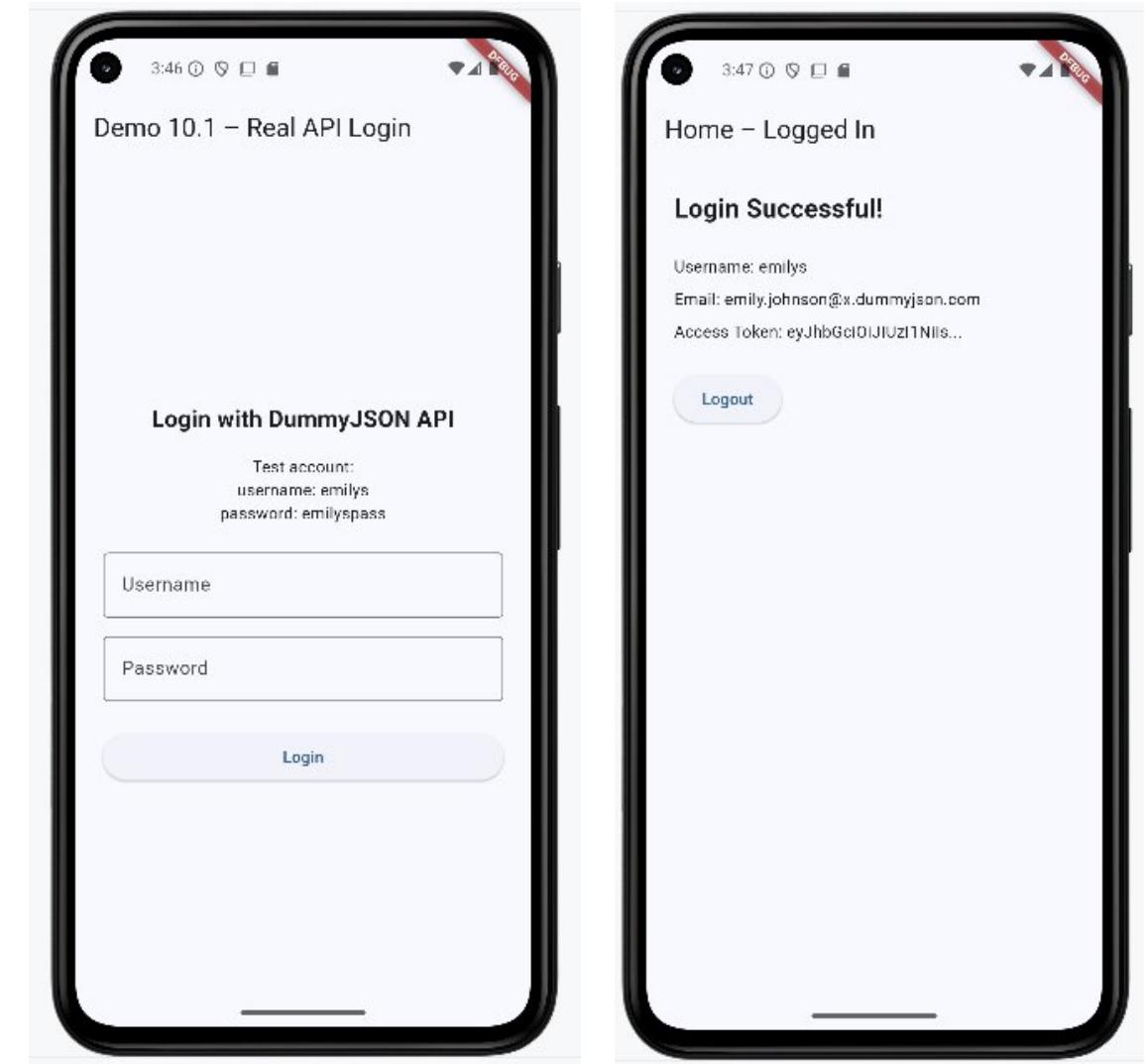
Demo1.1.R - Real API in Authentication

Description:

A real authentication demo using the DummyJSON API. The app sends login credentials via HTTP, receives an access token, handles errors, and navigates to the Home screen on success.

Key Steps

1. Build Login UI
2. Send Real API Request
3. Process Response
4. Navigate to Home Screen



Signup Flow Overview

Goal

- Extend Demo 10.1 by adding a Signup screen
- Reuse the same AuthService and error handling patterns
- Allow users to create an account, then go back to Login

Flow

1. User taps “Create account” on the Login screen
2. Navigate to SignupScreen
3. User fills in: name, email/username, password, confirm password
4. Form validates input (field + cross-field rules)
5. App calls AuthService.signup(...)
6. On success → show success message → navigate back to Login
7. On failure → show API error (e.g., email already used)

Signup API Integration (Reuse AuthService)

Backend Integration Pattern

- Login and Signup share the same service layer: AuthService
- Add a new method:

```
Future<void> signup(SignupRequest dto);
```

Relation to Demo 10.1

- Demo 10.1 → AuthService.login(...)
- Demo 10.2 → same AuthService, just add signup(...)
- UI uses the same error handling + loading patterns as login

Steps

- Build a DTO (data object) for signup: name, email/username, password
- Send POST /signup (or equivalent) to backend
- Parse JSON response
- Handle common errors:
 - Duplicate email / username
 - Weak password
 - Invalid data

Password Validation Rules (Used in Signup Form)

Field-level rules

- Minimum 6–8 characters
- At least 1 uppercase letter
- At least 1 number
- No leading / trailing whitespace

Form-level rules

- “Password” and “Confirm Password” must match
- Show clear, friendly error messages

UX Tips

- Show validation before sending request to backend
- Use inline error text below fields
- Disable “Create account” button when form is invalid or loading

Relation to Previous Modules

- Reuse patterns from Module 7 – Forms & Validation
- Now applied in a real Signup scenario

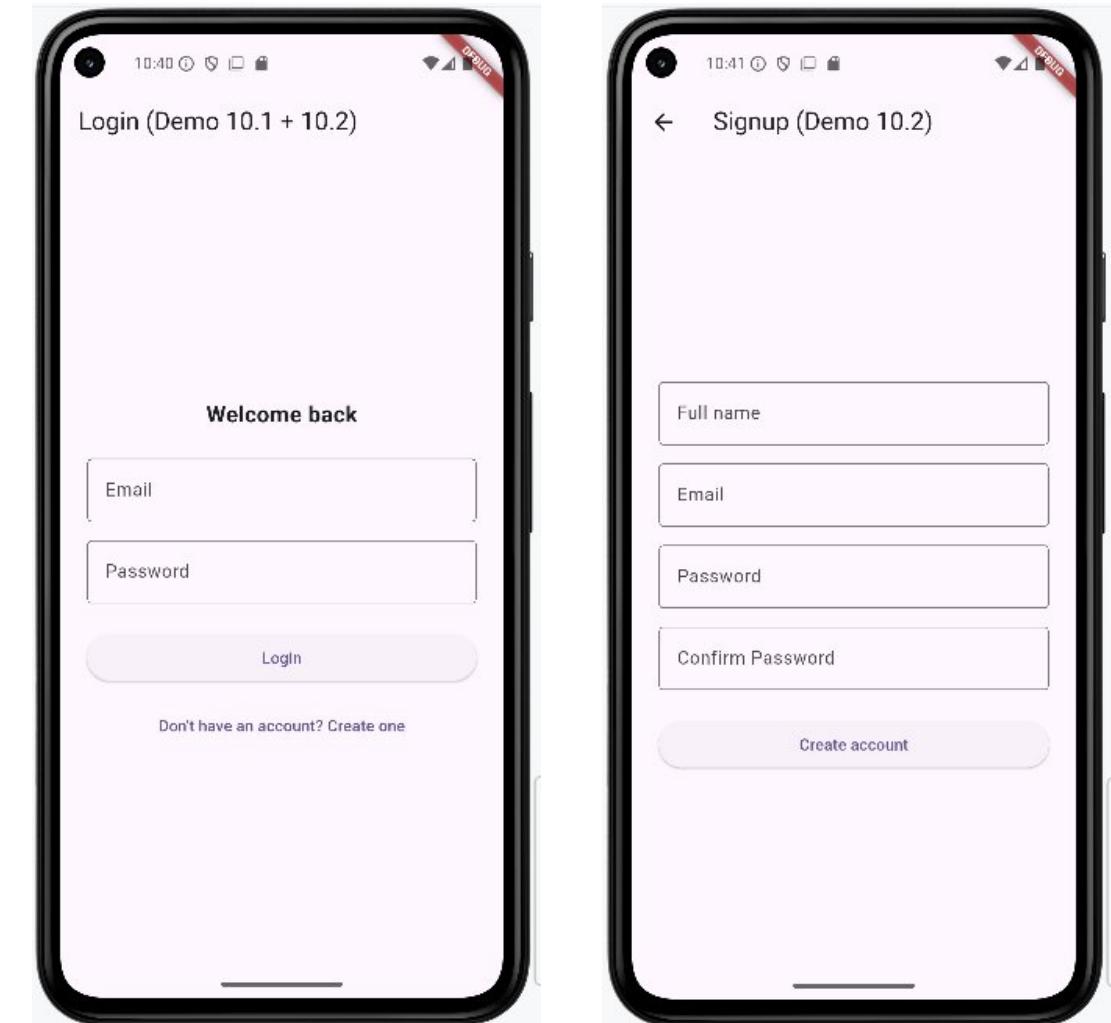
DEMO 10.2 — Signup Flow Implementation

What this demo adds on top of Demo 10.1

- New SignupScreen with:
 - Name
 - Email/username
 - Password
 - Confirm password
- New method in AuthService: signup(...)
- Navigation between Login ↔ Signup

Demo Features

- Validated signup form
- Loading state while sending request
- Display backend error (e.g. “Email already exists”)
- On successful signup → show a success message and return to Login screen



Understanding User Session

What is a Session?

- A session represents the authenticated state of a user.
- It typically includes:
 - Access token
 - User profile
 - Token expiry timestamp

How a session starts

- User logs in (Demo 10.1)
- App receives token
- App stores token locally (SharedPreferences)
- App restores token automatically next time (Demo 10.3)

How a session ends

- Logout
- Token expired
- Force logout from server (401 Unauthorized)

Persisting Session with SharedPreferences

Why store the token locally?

- Keep user logged in between app restarts
- Faster startup → auto skip login screen
- Required for protected APIs

Session Logic

- If token exists → user is authenticated
- If missing/expired → navigate to
LoginScreen

Code example

```
final prefs = await SharedPreferences.getInstance();
prefs.setString("auth_token", token);

// read
final token = prefs.getString("auth_token");
```

Auto Login Flow (SPLASH → LOGIN/HOME)

Flow

- Start SplashScreen
- Load token from SharedPreferences
- If token exists → go to Home
- If no token → go to Login
- HomeScreen allows Logout (delete token)

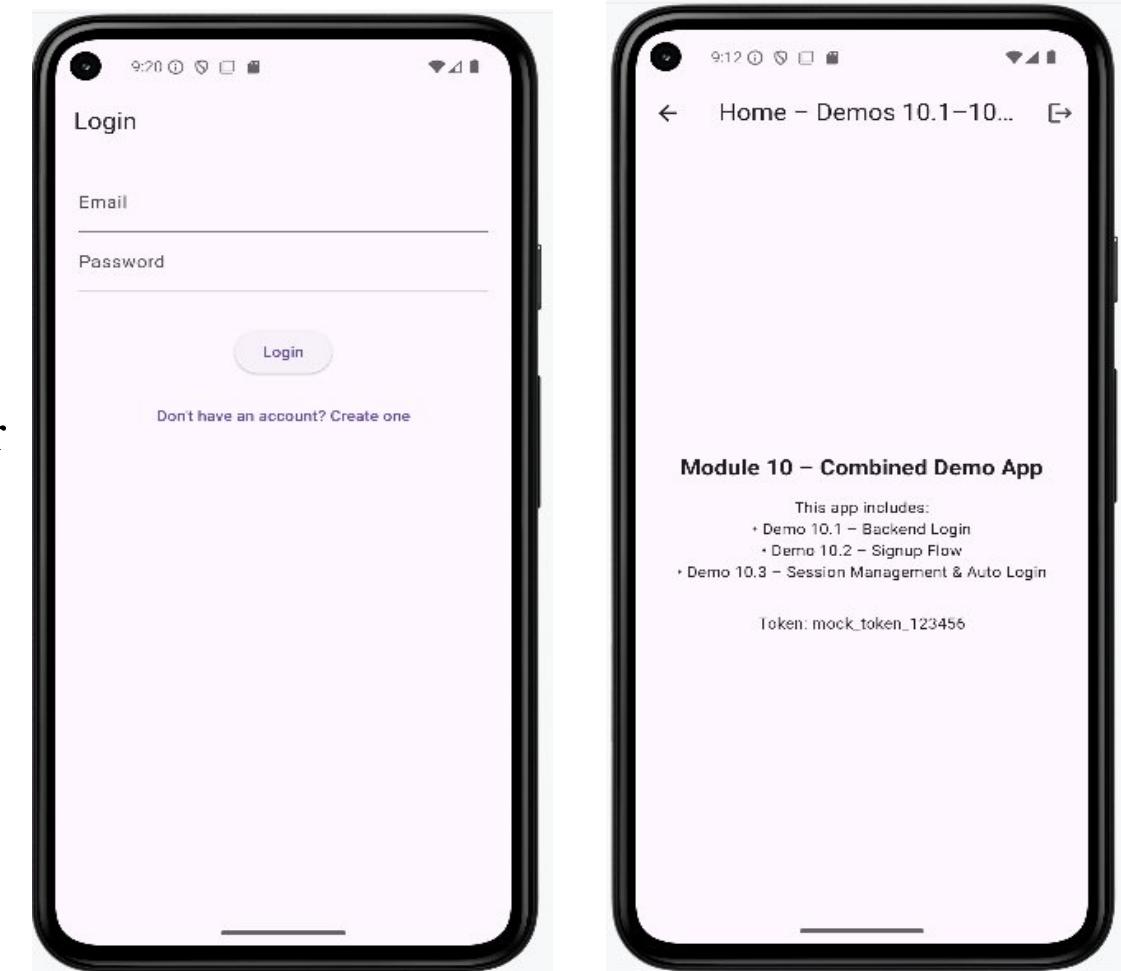
Flow Diagram:

Splash → (has token?) → Home / Login

DEMO 10.3 — Auto Login + Logout Implementation

Description

- Based on Demo 10.1 (Backend Login) and Demo 10.2 (Register).
- Upon successful login, save the auth_token app to SharedPreferences.
- The next time you open the app, the token checks for SplashScreen:
- With token → go straight to Home (auto login).
- Without token → go back to Login screen.
- The Logout button on Home deletes the token, forcing the user to log in again next time.



Test Scenarios for Demo 10.3

Scenario	Steps	Expected Result
1. First Login & Session Creation	<ol style="list-style-type: none">1. Open the app → Login screen appears.2. Enter valid credentials (demo@test.com / 123456).3. Tap Login.	<ul style="list-style-type: none">• User is navigated to Home – Demos 10.1–10.3.• Token is displayed (e.g., mock_token_123456).• Token is saved to SharedPreferences.
2. Auto Login (App Relaunch)	<ol style="list-style-type: none">1. Close the app completely (remove from Recent Apps).2. Reopen the app from the launcher icon.	<ul style="list-style-type: none">• Login screen is skipped.• SplashScreen loads token.• User is taken directly to Home with the same token displayed.
3. Logout & Session Clearing	<ol style="list-style-type: none">1. From Home, tap the Logout icon.2. App returns to Login screen.3. Close and reopen the app.	<ul style="list-style-type: none">• User stays on Login screen (auto-login disabled).• Token is removed from SharedPreferences.• Must log in again to access Home.

What is Firebase Authentication?

Firebase Authentication is a cloud-based identity platform that provides:

- Secure and scalable login
- Google, Apple, Facebook sign-in
- Email/password authentication
- Automatic session persistence
- Built-in token refresh
- Industry usage: e-commerce, travel, fintech apps

Why we introduce Firebase here?

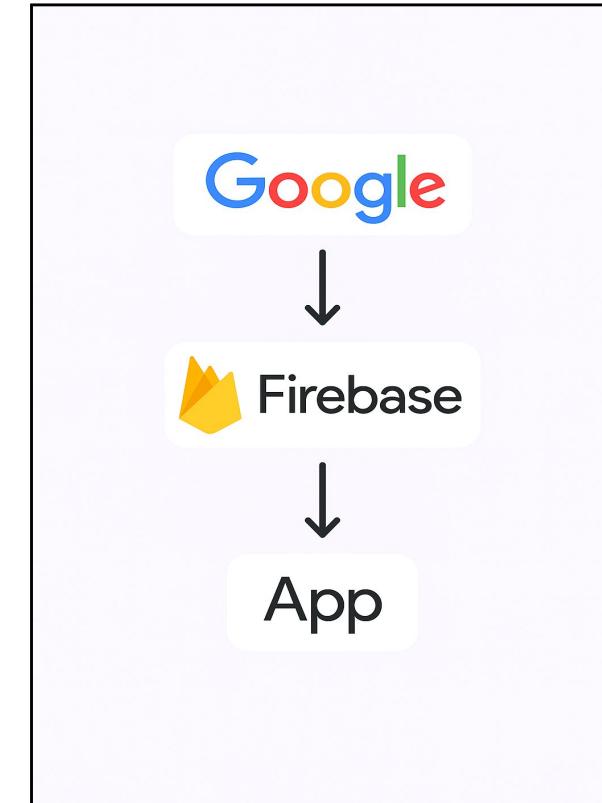
- Easier identity management than building your own backend
- Industry standard for mobile apps
- Used in modern e-commerce, booking, social apps



Google Sign-In Authentication Flow

Flow:

1. User taps Sign in with Google
2. Google account chooser appears
3. Google issues OAuth tokens
4. Convert to Firebase credential
5. Firebase signs in the user
6. App receives Firebase User object
7. Navigate to Home



Key Idea:

Firebase handles identity → our app only handles UI + navigation

Required Configurations

Before coding Google Sign-In, the following must be configured:

1. Create Firebase Project
2. Add Android App (correct package name)
3. Add SHA-1 + SHA-256 fingerprints
4. Download and place google-services.json → android/app/
5. Add dependencies:
 - firebase_core
 - firebase_auth
 - google_sign_in
6. Enable Google Sign-In in Firebase Console

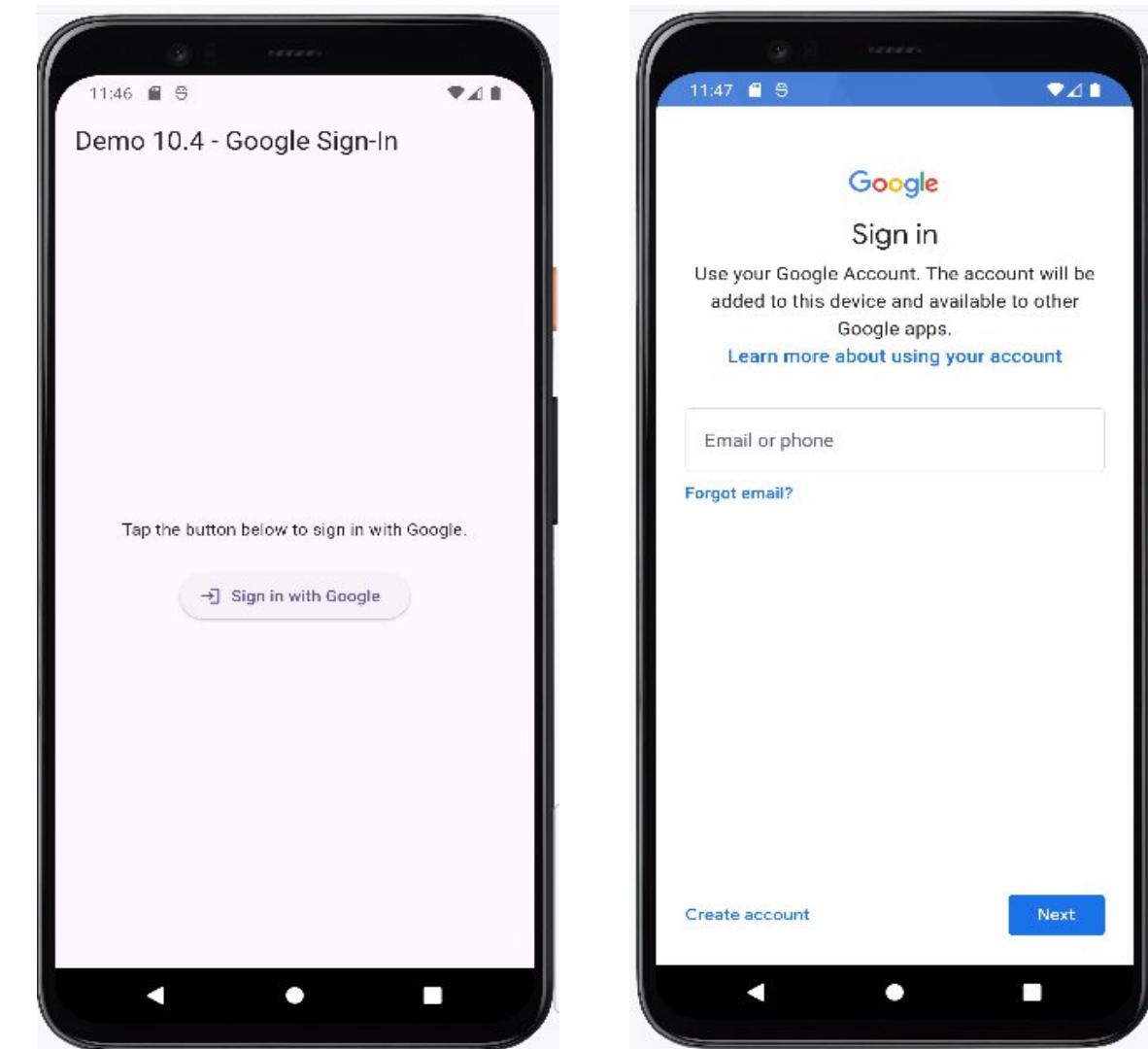
Demo 10.4: Google Sign-In (Overview)

This demo demonstrates:

- Google Sign-In (FirebaseAuth)
- Display user profile (photo, email, name)
- Persist login automatically
- Logout from Firebase & Google

Files Used:

- firebase_auth_service.dart
- google_sign_in_button.dart
- google_profile_screen.dart
- main.dart configuration update



Common Errors & Fixes

Error	Cause	Fix
Google Sign-In screen does not appear	Missing SHA-1	Add SHA-1 → rebuild app
“DEVELOPER_ERROR”	Wrong package name	Ensure package name matches Firebase
Firebase initialization error	Missing google-services.json	Place file inside /android/app/
App crashes at launch	Missing Gradle plugin	Add com.google.gms.google-services
Login returns null	Emulator lacks Google Play Services	Use emulator with Play Store or real device

What is a Notification?

Notifications in Mobile Applications

- A notification is a system-level message displayed outside the app UI.
- It allows the app to inform users about important events.
- Notifications improve user engagement and awareness.

Examples:

- Login success
- Account activity alerts
- Reminders

Local vs Push Notification

Local Notification	Push Notification
Triggered by the app	Sent from a server
No internet required	Internet required
No backend needed	Requires backend
Simple to implement	More complex

Why Notifications After Authentication?

Notifications in Authentication Flow

- Notifications are commonly used after important user actions such as:
- Successful login
- Account verification
- Password reset

Example:

- “Login successful. Welcome back!”

Purpose:

- Connect notifications with real application workflows.

Demo 10.5: Local Notification Flow

Objective

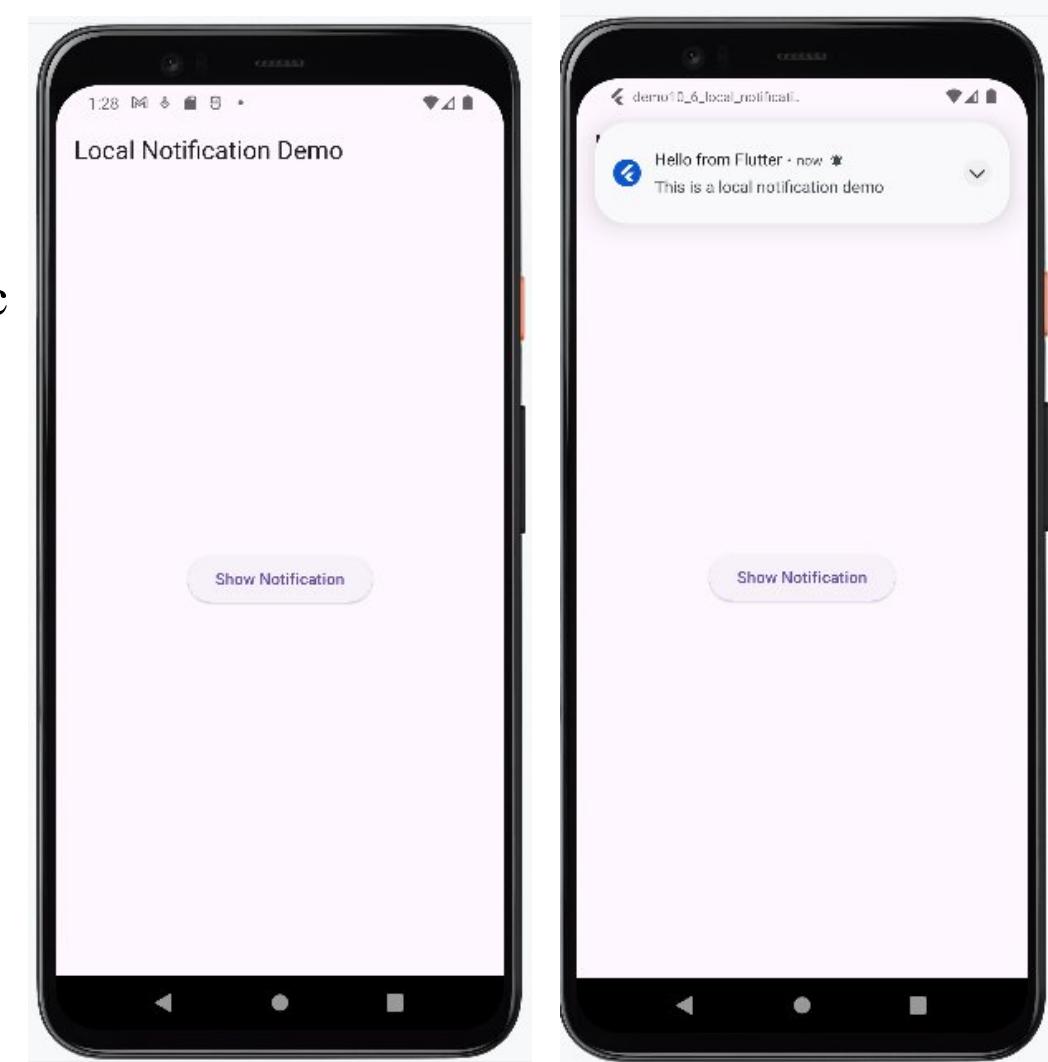
- Integrate local notifications in a Flutter application
- Trigger a notification after a successful login
- Keep notification logic separate from authentication logic

Step 1 – Setup Notification Service

- Initialize plugin
- Configure Android settings & permission

Step 2 – Trigger Notification After Login

- Call notification logic
- System displays notification



Best Practices & Patterns for Authentication

Security

- Hash & salt passwords (server-side).
- Never store raw tokens → use secure storage.
- Always use HTTPS.

User Experience

- Clear and friendly error messages.
- Show loading indicators for all network calls.
- Support auto-login with persisted session.

Architecture Patterns

- AuthService (Singleton) separating UI from logic.
- Standard flow: Splash → Login → Home.
- Use pushReplacementNamed() to prevent navigating back to Login.

Error Handling

- Wrap API calls in try/catch.
- Display errors gracefully (Snackbar / inline text).

Summary

In this module, students learned:

- Core concepts of authentication, authorization, sessions, and JWT tokens
- How to design and validate login and signup forms in Flutter
- How to implement authentication using:
 - Mock backend service (Demo 10.1)
 - Real REST API integration (Demo 10.2 – DummyJSON)
- How to persist user sessions and implement auto-login & logout using SharedPreferences (Demo 10.3)
- How to integrate Firebase Authentication and Google Sign-In (Demo 10.4)
- (Optional) How to integrate local notifications to enhance user experience after key actions (Demo 10.5)

References

- Mastering Flutter 2025: Learn to Develop Flutter Apps, Kevin Moore, Packt Publishing. Chapter 9, 10, 12, 17, 19
- Flutter Documentation – Authentication: <https://docs.flutter.dev>
- Firebase Authentication Documentation: <https://firebase.google.com/docs/auth>
- SharedPreferences Flutter Plugin: https://pub.dev/packages/shared_preferences
- Path Provider Plugin: https://pub.dev/packages/path_provider
- DummyJSON Authentication API: <https://dummyjson.com/docs/auth>
- Flutter Local Notifications Plugin: https://pub.dev/packages/flutter_local_notifications

Module 11 - Testing & Debugging in Flutter

Learning Objectives

Content:

- Identify common issues in Flutter apps
- Test logic, UI behavior, and navigation
- Validate complete user flows
- Use DevTools for debugging UI and performance
- Apply these techniques in Lab 11

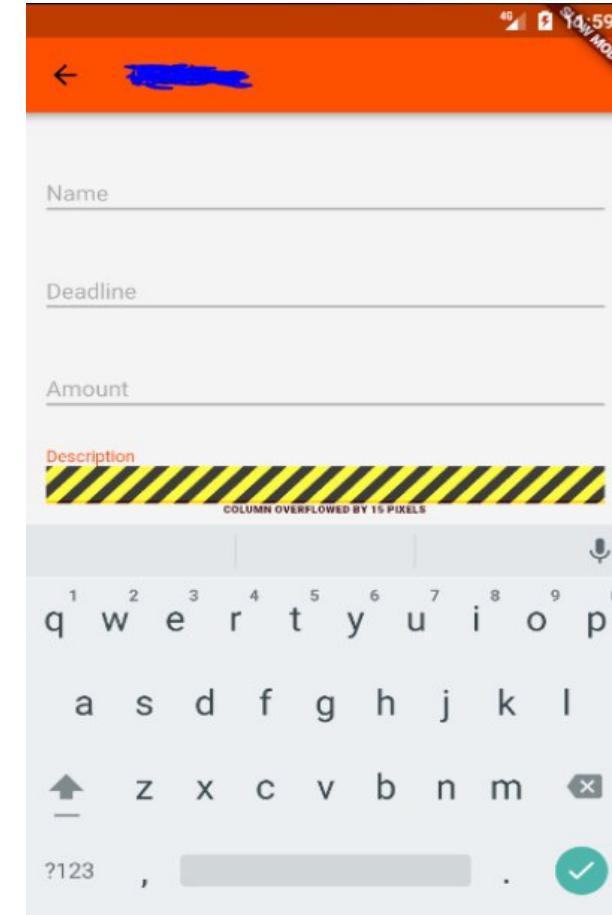
Why Performance Matters

- Users expect smooth 60–120 FPS experiences
- Poor rebuild patterns increase CPU load
- Large assets slow startup times
- Inefficient list rendering affects scrolling
- App size impacts downloads & Play Store acceptance

Example of a Flutter UI Problem

UI Overflow Error Example

- Occurs when a widget exceeds its layout constraints.
- Explain that these issues do not crash the app, so tests or DevTools are needed to detect them.

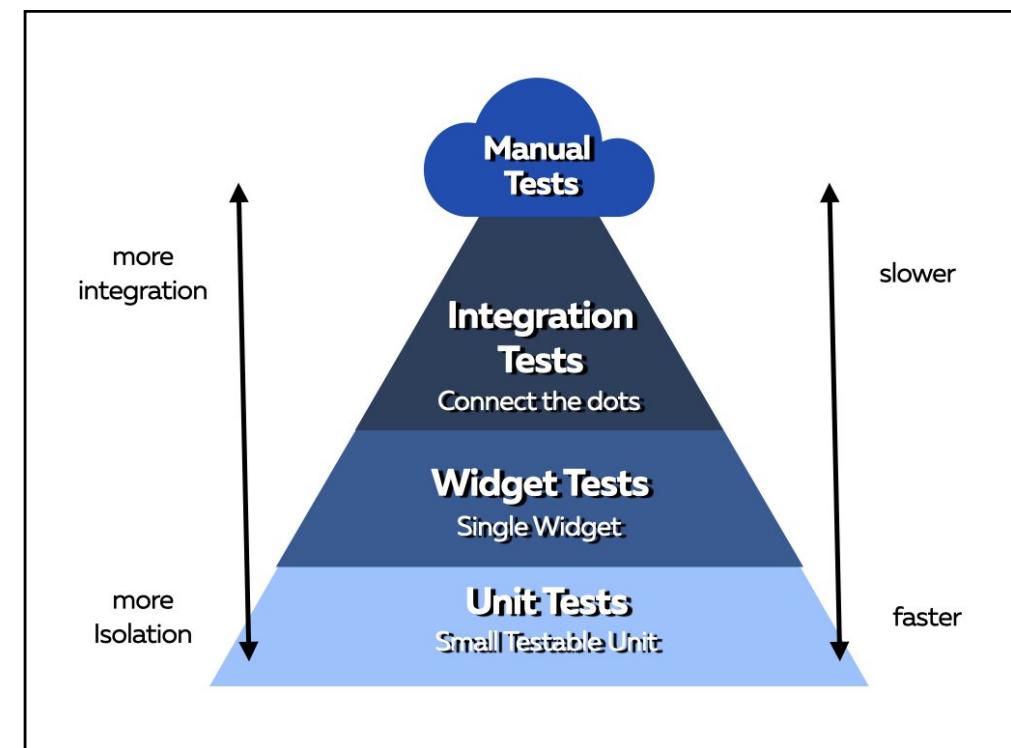


Types of Tests in Flutter

Three essential test types:

1. Unit Tests
 - Validate logic without UI
2. Widget Tests
 - Validate UI rendering & interaction
3. Integration Tests
 - Validate full user flows from start to finish

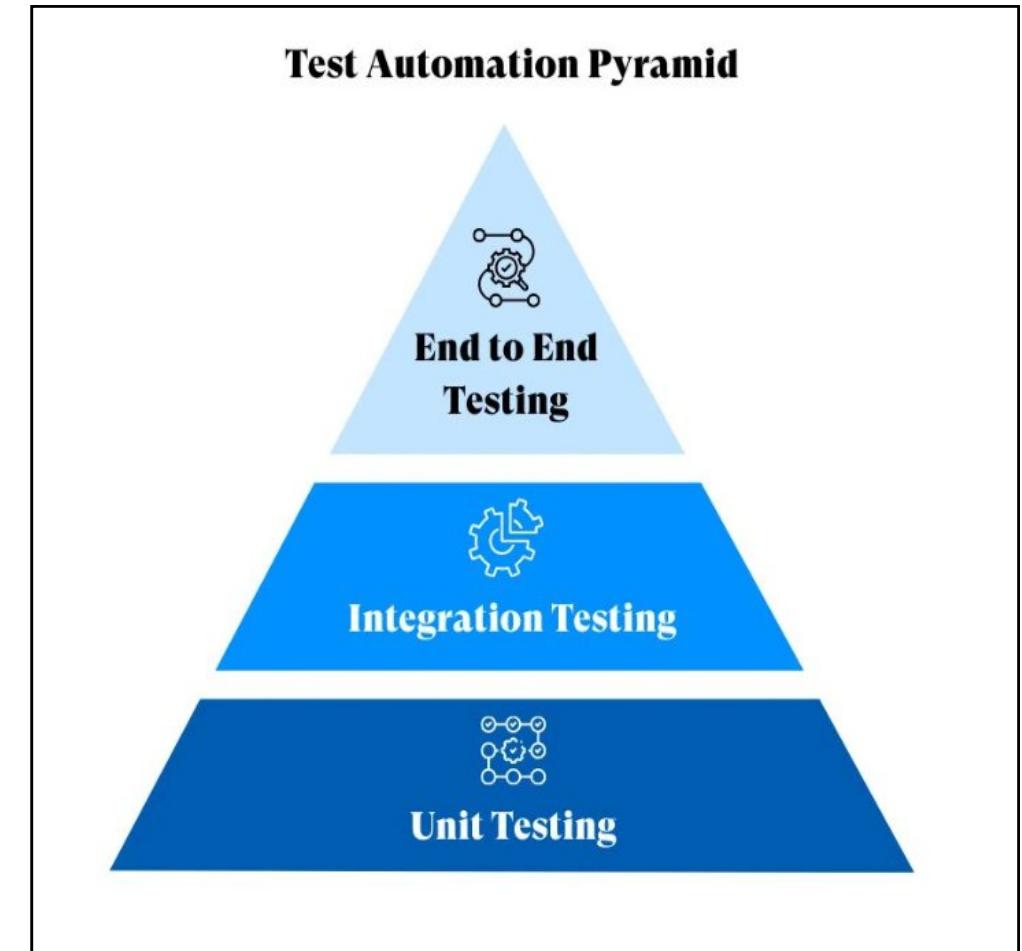
Layered Architecture Diagram



The Flutter Testing Pyramid

Why It Matters

- **Unit Tests → many**
 - Fast, stable, cheap. Catch logic bugs early.
- **Widget Tests → moderate**
 - Validate UI behavior and interactions.
- **Integration Tests → few**
 - Slow & expensive. Use for critical user flows only.



Flutter Testing Workflow

General steps for any Flutter test:

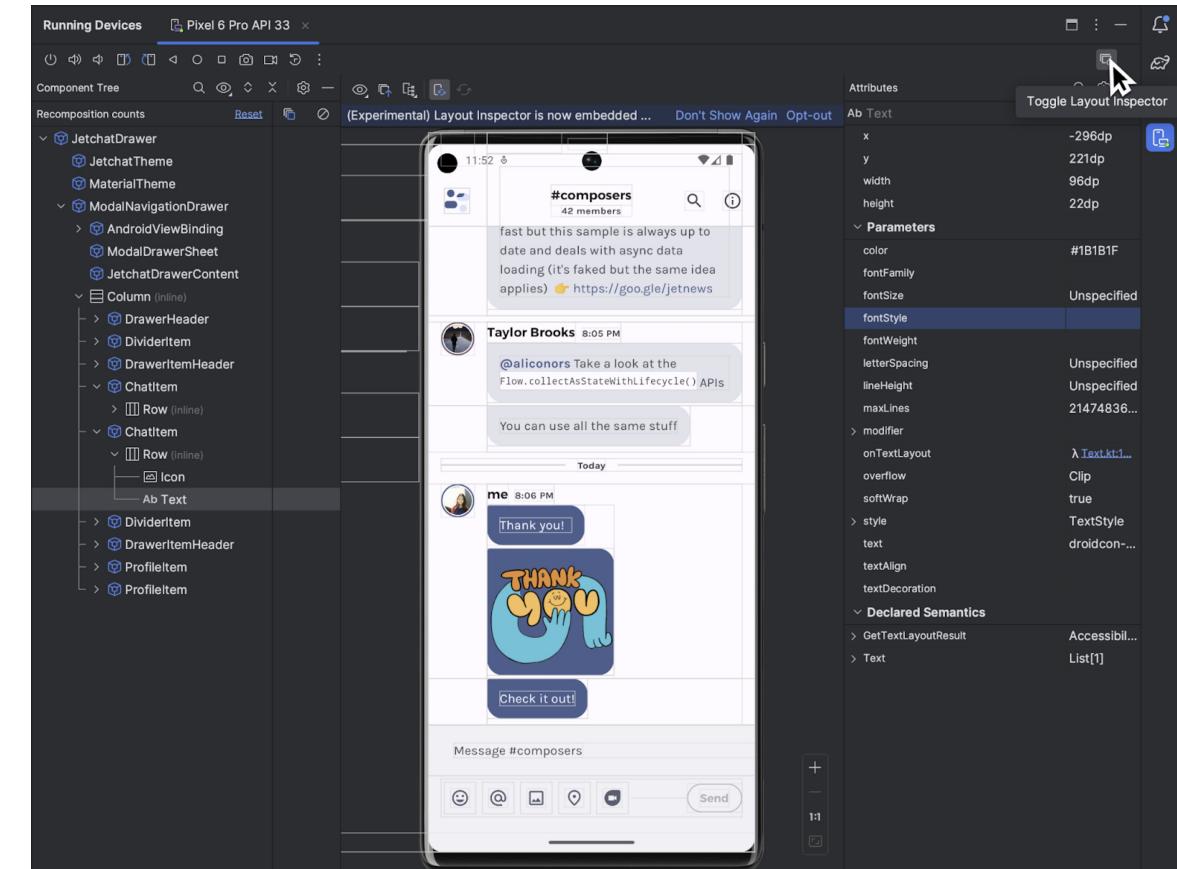
- Define the expected behavior
- Prepare the test environment
- Execute logic / render widget
- Simulate interactions
- Use expect() to verify the results



Debugging Goals in Flutter

Debugging in Flutter focuses on:

- Layout issues (constraints, overflow)
- State update failures
- Performance slowdowns
- Frequent unnecessary rebuilds
- Navigation issues



How Tests Map to Real Flutter Problems

Key Insight

- Most UI issues start from logic → unit tests catch problems early.
- Layout & interaction problems → widget tests detect what unit tests cannot see.
- Multi-screen or user-flow bugs → integration tests validate the entire experience.

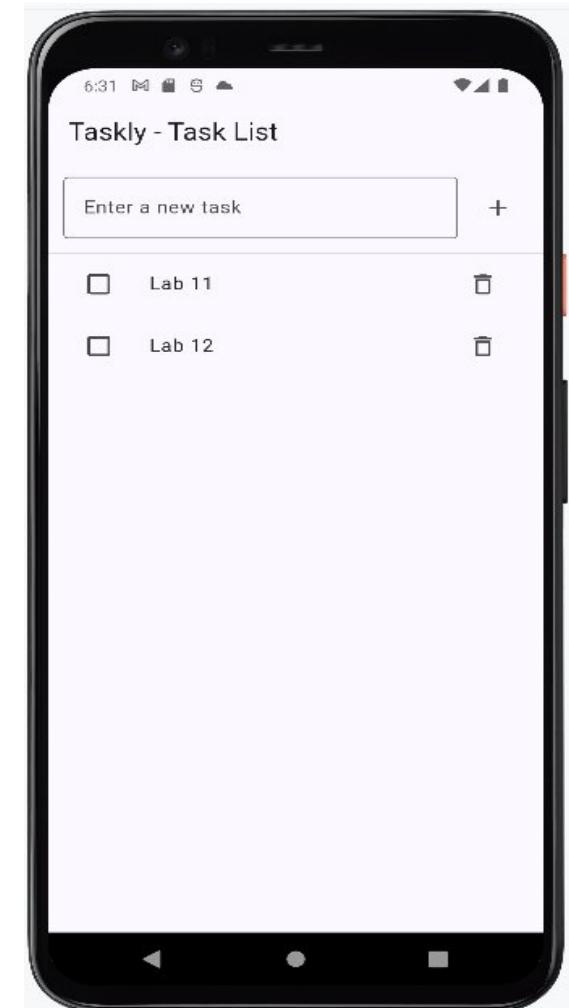
Flutter Issue	Recommended Test	Why
Logic failure	Unit test	Fast + stable
UI not updating	Widget test	Validate rebuild
Wrong navigation	Navigation test	Checks route + data
Entire feature broken	Integration test	Simulates real use

Introducing the Taskly Demo App

We use Taskly, a simple Todo App, to demonstrate each test type:

Features:

- Task List screen
- Task Detail screen
- Add, delete, toggle tasks
- Edit and save tasks



Taskly App Overview

App Structure

- TasklyApp is the application's entry point.
- Provider (TaskProvider) manages app-wide state.
- TaskRepository handles CRUD logic for tasks.
- TaskListScreen is the primary **UI used in testing**.

Why This Matters for Testing

- Shows which widget is pumped during tests.
- Defines the state flow: UI → Provider → Repository.
- Provides context for unit, widget, navigation, and integration testing.

```
class TasklyApp extends StatelessWidget {  
  const TasklyApp({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return ChangeNotifierProvider(  
      create: (_) => TaskProvider(TaskRepository()),  
      child: const MaterialApp(  
        home: TaskListScreen(),  
      ),  
    );  
  }  
}
```

Demo 11.1 – Unit Test: Task Model

File:

test/unit/demo_11_1_task_model_test.dart

```
import 'package:flutter_test/flutter_test.dart';
import 'package:demo11_taskly/models/task.dart';

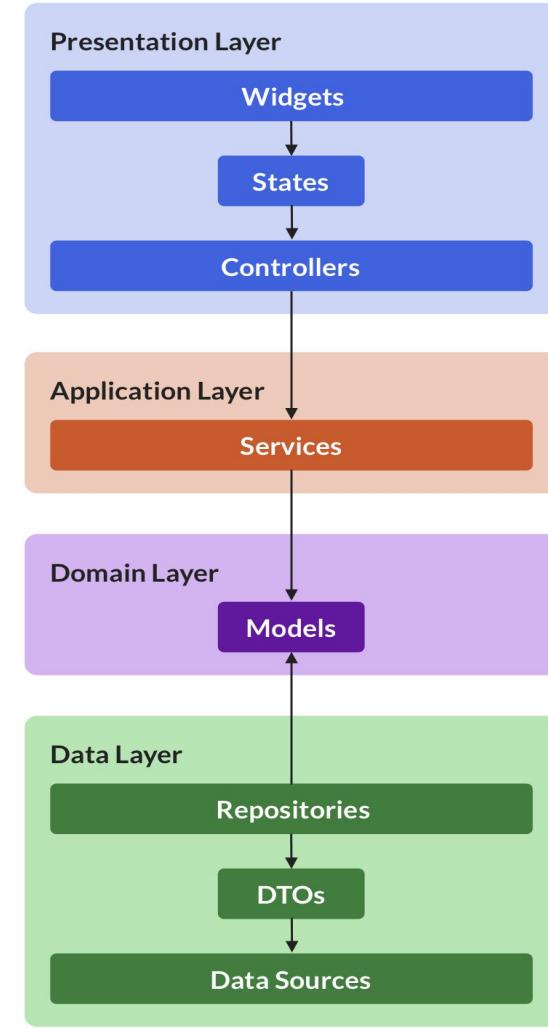
void main() {
  group('Task model - Demo 11.1', () {
    test('toggle() should switch completed state', () {
      // STEP 1 - Arrange
      final task = Task(id: '1', title: 'Demo task');

      expect(task.completed, false);

      // STEP 2 - Act
      task.toggle();

      // STEP 3 - Assert
      expect(task.completed, true);

      // Optional: toggle again
      task.toggle();
      expect(task.completed, false);
    });
  });
}
```



Unit Test Demonstration (Task Model)

What we test here

- Toggle completion
- Update model state
- Logic stays predictable

Expected Output

- Initial: completed = false
- After toggle(): completed = true

Snippet

```
final task = Task(id:'1', title:'A');  
task.toggle();  
expect(task.completed, true);
```

Explanation

Line of Code	User Action Simulated	Purpose in the Test	What It Verifies
final task = Task(...)	Creates a new Task object	Arrange: prepare test data	Model starts with known state
task.toggle();	Toggles completion flag	Act: perform operation	Business logic changes state
expect(task.completed, true);	Check result	Assert: validate output	Logic behaves correctly

Demo 11.2 – Widget Test: Add Task Updates UI

File: test/widget/demo_11_2_task_list_add_test.dart

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

import 'package:demo11_taskly/providers/task_provider.dart';
import 'package:demo11_taskly/repository/task_repository.dart';
import 'package:demo11_taskly/screens/task_list_screen.dart';

Widget createTaskListTestApp() {
  return ChangeNotifierProvider(
    create: (_) => TaskProvider(TaskRepository()),
    child: const MaterialApp(
      home: TaskListScreen(),
    ),
  );
}
```

```
void main() {
  group('Task List - Demo 11.2', () {
    testWidgets('adding a task should display it in the list',
        (WidgetTester tester) async {
      // STEP 1 - Arrange
      await tester.pumpWidget(createTaskListTestApp());
      expect(find.text('No tasks yet. Add one!'), findsOneWidget);

      // STEP 2 - Act
      await tester.enterText(
          find.byKey(const Key('taskTextField')),
          'Buy milk',
      );
      await tester.tap(find.byKey(const Key('addTaskButton')));
      await tester.pump();

      // STEP 3 - Assert
      expect(find.text('Buy milk'), findsOneWidget);
      expect(find.text('No tasks yet. Add one!'), findsNothing);
    });
  });
}
```

Widget Test Demonstration (Add Task Interaction)

What we test

- Enter text
- Tap “Add”
- List updates with newly added task

Expected Output

- Before: “No tasks yet. Add one!”
- After: “Buy milk”

What this proves

- UI rebuilds correctly
- Provider state updates
- ListView reacts to new data

Snippet

```
await tester.enterText(find.byKey(Key('taskTextField')), 'Buy milk');
await tester.tap(find.byKey(Key('addTaskButton')));
await tester.pump();
```

Explanation

Line of Code	User Action Simulated	Purpose in Test	What It Verifies
enterText(..., 'Buy milk')	User types into the input field	Act: simulate typing	The TextField accepts input
tap (addTaskButton)	User presses the Add button	Act: trigger UI action	Provider receives update request
pump()	UI rebuild	Assert preparation	The new task appears on screen

Demo 11.3 – Navigation Test: List → Detail

File : test/widget/demo_11_3_navigation_to_detail_test.dart

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

import 'package:demo11_taskly/models/task.dart';
import 'package:demo11_taskly/providers/task_provider.dart';
import 'package:demo11_taskly/repository/task_repository.dart';
import 'package:demo11_taskly/screens/task_list_screen.dart';

Widget createTaskListWithSeededTask() {
    final repo = TaskRepository();
    repo.addTask(Task(id: '1', title: 'Demo task for navigation'));

    return ChangeNotifierProvider(
        create: (_) => TaskProvider(repo),
        child: const MaterialApp(
            home: TaskListScreen(),
        ),
    );
}
```

```
void main() {
    group('Navigation - Demo 11.3', () {
        testWidgets('tapping a task navigates to Task Detail screen',
            (WidgetTester tester) async {
                // STEP 1 - Arrange
                await tester.pumpWidget(createTaskListWithSeededTask());
                expect(find.text('Demo task for navigation'), findsOneWidget);

                // STEP 2 - Act
                await tester.tap(find.text('Demo task for navigation'));
                await tester.pumpAndSettle();

                // STEP 3 - Assert
                expect(find.text('Task Detail'), findsOneWidget);
                expect(find.byKey(const Key('detailTitleField')), findsOneWidget);
            });
    });
}
```

Navigation Test Demonstration (List → Detail)

What We Test

- Click on the item in the list
- Navigate to the detail screen
- Data passed to the detail is correct

Expected output

- New screen shows: "Task Details"
- Detail form loaded with task data

Why this is important

- Navigation errors are common
- Ensure routes and widget tree are correct

Snippet

```
await tester.tap(find.text('Demo task'));  
await tester.pumpAndSettle();  
expect(find.text('Task Detail'), findsOneWidget);
```

Explanation

Line of Code	User Action Simulated	Purpose in Test	What It Verifies
tap(find.text('Demo task'))	User taps a list item	Act: trigger navigation	Navigation event is fired
pumpAndSettle()	Wait for animations & navigation	Arrange for assertion	Destination screen fully loads
expect(find.text('Task Detail')...)	Check screen title	Assert: confirm navigation result	App navigated to the correct page

Demo 11.4 – Integration Test: Full CRUD Flow

File: test/widget/demo_11_4_integration_flow_test.dart

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

import 'package:demo11_taskly/providers/task_provider.dart';
import 'package:demo11_taskly/repository/task_repository.dart';
import 'package:demo11_taskly/screens/task_list_screen.dart';

Widget createFullAppForIntegration() {
  return ChangeNotifierProvider(
    create: (_) => TaskProvider(TaskRepository()),
    child: const MaterialApp(
      home: TaskListScreen(),
    ),
  );
}

void main() {
  group('Integration flow - Demo 11.4', () {
    testWidgets('add, open detail, edit, save and return to updated list',
      (WidgetTester tester) async {
        // STEP 1 - Arrange
        await tester.pumpWidget(createFullAppForIntegration());
```

```
        // STEP 2 - Act: add a new task
        await tester.enterText(
          find.byKey(const Key('taskTextField')),
          'Original title',
        );
        await tester.tap(find.byKey(const Key('addTaskButton')));
        await tester.pump();

        // Open detail
        await tester.tap(find.text('Original title'));
        await tester.pumpAndSettle();
        expect(find.text('Task Detail'), findsOneWidget);

        // Edit title
        await tester.enterText(
          find.byKey(const Key('detailTitleField')),
          'Updated title',
        );
        await tester.tap(find.byKey(const Key('detailSaveButton')));
        await tester.pumpAndSettle();

        // STEP 3 - Assert
        expect(find.text('Updated title'), findsOneWidget);
        expect(find.text('Original title'), findsNothing);
      });
  });
}
```

Integration Test Demonstration (Full Task Flow)

What we test

- Add task
- Open detail
- Edit title
- Save
- Confirm new title appears in list

Expected Output

- Old title disappears
- New title displayed

Snippet

```
await tester.enterText(find.byKey(Key('taskTextField')), 'Original');  
...  
await tester.enterText(find.byKey(Key('detailTitleField')), 'Updated');  
await tester.tap(find.byKey(Key('detailSaveButton')));
```

Explanation

Line of Code	User Action Simulated	Purpose in Test	What It Verifies
enterText(taskTextField, 'Original')	User creates a new task	Step 1 – Add task	Input field and add flow work
enterText(detailTitleField, 'Updated')	User edits the task title	Step 2 – Edit task	Detail screen correctly receives data
tap(detailSaveButton)	User saves the edited task	Step 3 – Save operation	Provider updates data + UI refresh

Best Practices

- Keep tests small and focused
- Use keys for widget tests
- Mock repositories when possible
- Avoid testing Flutter framework functionality
- Don't overuse integration tests

Summary

In this module, students learned:

- In this module, students learned how to:
- Identify common UI and state-related issues in Flutter apps
- Apply the Flutter testing pyramid (Unit → Widget → Integration)
- Write unit tests to verify logic and model behavior
- Write widget tests to validate UI rendering and interactions
- Write integration tests to confirm full user flows
- Use DevTools to diagnose layout, rebuild, and navigation issues

References

- Mastering Flutter 2025: Learn to Develop Flutter Apps, Kevin Moore, Packt Publishing. Chapter 13, 14, 15
- Flutter Documentation – Testing: <https://docs.flutter.dev/cookbook/testing>
- Flutter Documentation – DevTools: <https://docs.flutter.dev/tools/devtools/overview>
- Provider Package Documentation: <https://pub.dev/packages/provider>
- flutter_test Library API Reference: https://api.flutter.dev/flutter/flutter_test/flutter_test-library.html

Module 12 - Performance Optimization & App Deployment

Learning Objectives

Content:

- Explain how Flutter renders widgets and what causes performance overhead
- Identify common performance bottlenecks in Flutter UI
- Use DevTools to diagnose rebuilds, repaints, memory, and CPU load
- Apply best practices to optimize task lists, images, and widget trees
- Analyze and reduce app size before release
- Build a release APK / AppBundle suitable for distribution

Why Performance Matters in Flutter

Performance affects:

1. Frame Rendering
 - Flutter aims for 60 FPS → 16.6ms per frame
 - New devices: 120 FPS → 8ms per frame
2. User Experience
 - Stutters, dropped frames, slow scrolling → feels cheap
 - Poorly optimized lists = biggest cause of student app lag
3. Device Constraints
 - Limited CPU/GPU (especially mid-range Android devices)
 - Flutter rebuilds are cheap but not free
4. Real-world publishing
 - Store policies require reasonable size
 - Large apps increase uninstall probability

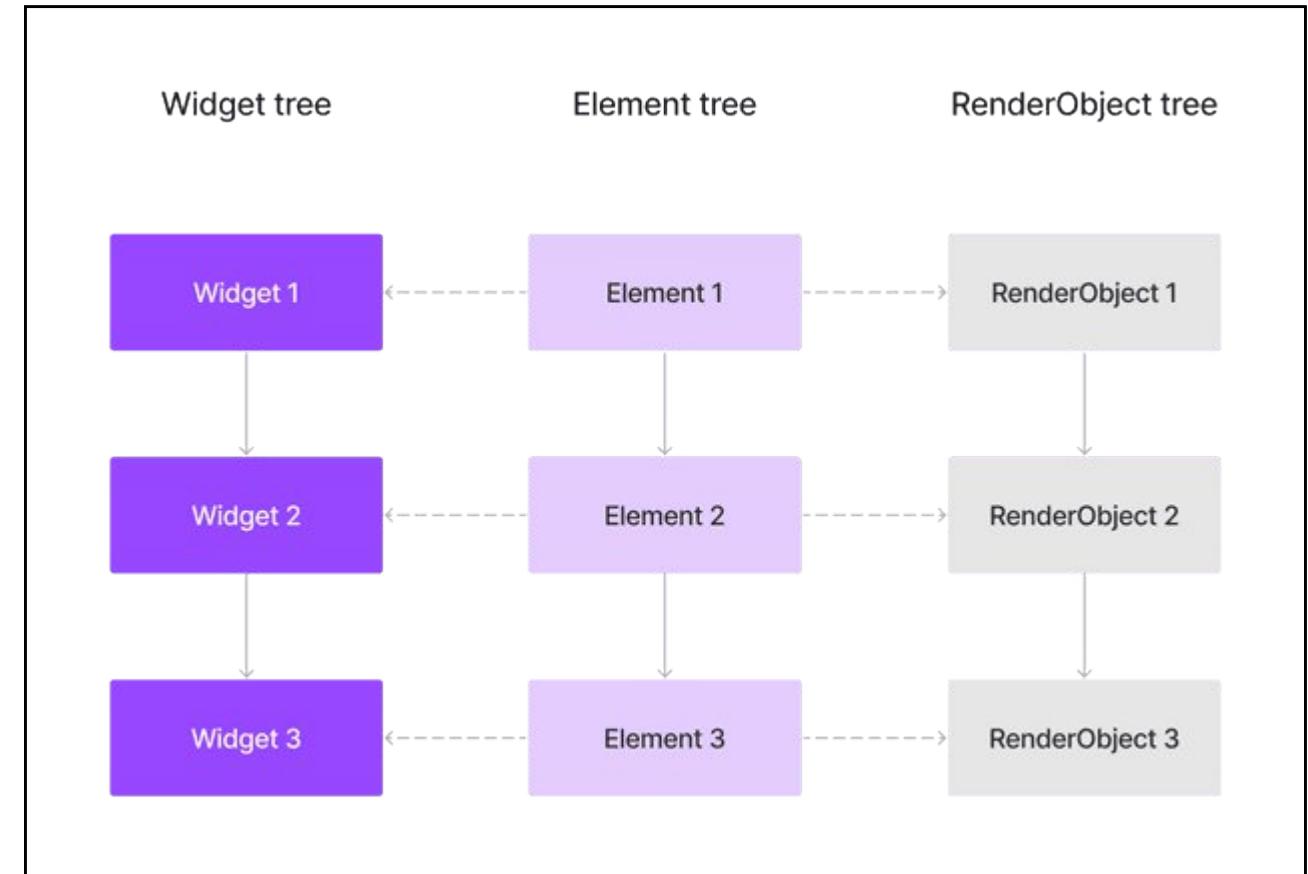
How Flutter Builds the UI (Render Pipeline)

Key Concepts:

- Widgets are immutable blueprints
- Elements handle lifecycle & state
- RenderObjects perform layout, painting, compositing

Performance Principle:

- Avoid extra work in build(), reduce unnecessary rebuilds.



What Causes Performance Problems?

Common issues:

- Too many widget rebuilds
- Re-running heavy logic inside build()
- Large images decoded at full resolution
- Overly deep widget trees
- Incorrect use of setState or Provider listeners
- Inefficient list rendering
- Expensive synchronous operations in the UI thread
- Use of GlobalKey everywhere

Example issues in student apps:

- ListView lag with many items
- Constant rebuild of whole screens
- Loading full-resolution images from network

Flutter DevTools Overview

Key Tools:

- Performance: FPS timeline, frame rendering cost
- CPU Profiler: execute-time hotspots
- Memory: leaks & allocations
- Widget Inspector: identify rebuild sources
- Repaint Rainbow: highlight repaint operations
- Rebuild Tracker (extensions): count widget rebuilds

DevTools Usage Tips:

- Run app in Profile Mode for accurate data
- Avoid measuring performance in Debug mode.

Identifying Rebuild Issues

Flutter rebuild issues typically come from:

1. State misplacement:

- If state lives too high → entire UI rebuilds
- If state lives too low → data not propagated properly

2. Missing const constructors:

- const lets Flutter skip rebuilding widgets

3. Overuse of setState:

- Calling setState() on large widgets triggers mass rebuilds

4. Provider misconfiguration:

- Using context.watch<T>() instead of Selector<T>() leads to redundant rebuilds

Optimization Technique 1: Use const Effectively

Why const matters:

- Reduces rebuilds
- Saves memory
- Helps Flutter tree diffing

Note:

- If the widget never changes →
use const.

Example:

Before:

```
Text("Task Title")
```

After:

```
const Text("Task Title")
```

Optimization Technique 2: Extract Widgets

Large widgets rebuilding unnecessarily = lag.

Benefits:

- Only TaskTile rebuilds
- Parent ListView stays untouched
- Cleaner code structure

Note:

- Use extraction + const for maximal effect.

Example (Taskly):

Before:

```
ListTile(  
    title: Text(task.title),  
    trailing: Checkbox(...),  
)
```

After:

```
TaskTile(task: task) // separate widget
```

Optimization Technique 3: Avoid Heavy Work in build()

**Absolutely do not do it inside the build()
function.**

- Sorting
- Parsing JSON
- Complex calculations
- Database reads
- API calls

Example Anti-pattern:

```
Widget build(context) {  
    final sorted = tasks..sort(); // NO  
}
```

Correct:

```
sortTasks(); // call once
```

Optimization Technique 4: Using Keys Properly

When to use keys:

- Preserving state
- Animations
- Reordering lists
- Improving diffing performance

GlobalKey Warning:

- Powerful but slow
- Avoid unless needed for navigation or accessing widget state

Good:

`ValueKey(task.id)`

Bad:

`UniqueKey() // forces rebuild every`

`frame`

Taskly Optimization Overview

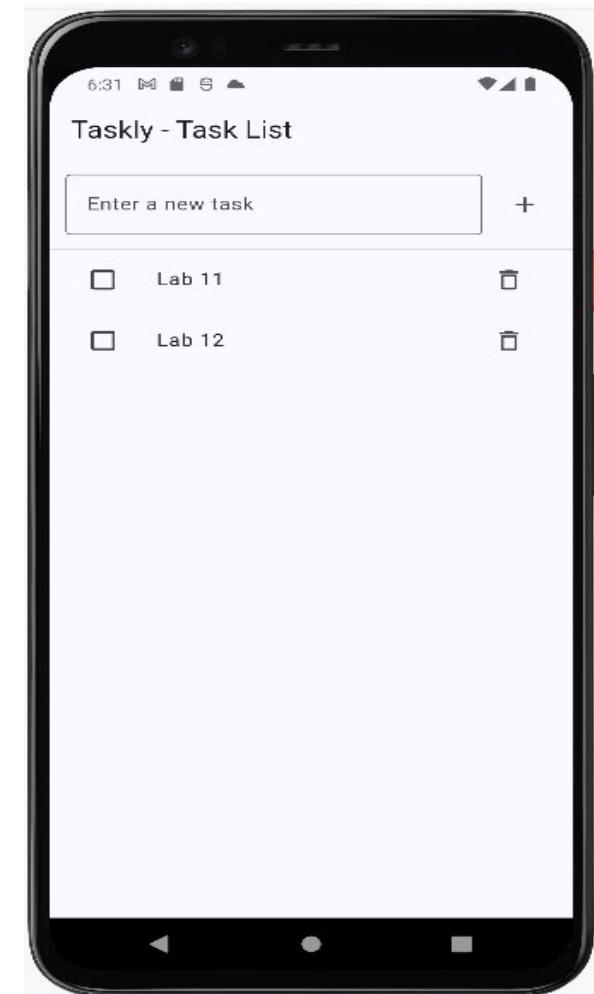
We will apply the module's performance techniques to optimize and deploy the Taskly app

Why Taskly?

- Small enough to understand quickly
- Complex enough to reveal real performance issues
- Perfect for demonstrating measurable improvements

Focus Areas

- UI Optimization
- List Performance
- Resource Efficiency
- App Size Analysis
- Deployment Prep



Demo 12.1 - Optimize List & UI Rebuilds (Taskly)

Improve performance by reducing unnecessary widget rebuilds

Objectives

- Understand why the old Taskly implementation caused full-screen rebuilds
- Apply widget extraction (TaskTile)
- Use Selector to minimize rebuild scope
- Add const where possible for stable widgets
- Observe rebuild reduction using debug logs or DevTools

The Problem (Before Optimization)

Issue: The entire TaskListScreen rebuilds whenever tasks change.

Why it's slow

- `context.watch<TaskProvider>()` used at the top of the widget
- List items rendered inline in itemBuilder
- No separation between static UI (input row) and dynamic UI (list)
- ListView rebuilds all items even when only one task changes

Demo 12.1 - (continued)

Before Code (Inefficient)

```
// Entire screen listens for changes ✗  
final taskProvider = context.watch<TaskProvider>();  
  
Expanded(  
    child: ListView.builder(  
        itemCount: taskProvider.tasks.length,  
        itemBuilder: (context, index) {  
            final task = taskProvider.tasks[index];  
  
            // Inline ListTile ✗  
            return ListTile(  
                title: Text(task.title),  
                leading: Checkbox(  
                    value: task.completed,  
                    onChanged: (_ ) => taskProvider.toggleTask(task.id),  
                ),  
                trailing: IconButton(  
                    icon: const Icon(Icons.delete),  
                    onPressed: () => taskProvider.deleteTask(task.id),  
                ),  
            );  
        },  
    ),  
);  
);
```

Solution (After Optimization)

We isolate list rebuilds and extract item UI.

1) Extract TaskTile widget

```
class TaskTile extends StatelessWidget {  
  
    const TaskTile({  
        super.key,  
        required this.task,  
        required this.onToggleCompleted,  
        required this.onDelete,  
    });  
  
    // ...  
}
```

Full-screen rebuild → laggy scrolling, poor performance on long lists.

Demo 12.1 - (continued)

Solution (After Optimization)

2) Use Selector to rebuild ONLY the list

```
Selector<TaskProvider, List<Task>>(
    selector: (_, provider) => provider.tasks,
    builder: (context, tasks, _) =>
        return ListView.builder(
            itemCount: tasks.length,
            itemBuilder: (_, index) =>
                final task = tasks[index];
                return TaskTile(
                    key: ValueKey(task.id), // Efficient diffing
                    task: task,
                    onToggleCompleted: () =>
                        context.read<TaskProvider>().toggleTask(task.id),
                    onDelete: () =>
                        context.read<TaskProvider>().deleteTask(task.id),
                );
        ),
    );
);
```

3) Make input UI independent of task list

```
const _TaskInputRow(); // No rebuild on task changes
```

Performance Result

Before:

- Entire screen rebuilds
- All list items rebuilt on every action

After:

- Only the list rebuilds
- Only the affected TaskTile rebuilds
- Smoother scrolling
- Cleaner, maintainable architecture

Demo 12.2 - Image Optimization

Improve image loading performance and reduce layout cost.

Objectives

- Understand why large or uncompressed images affect performance
- Apply resizing, caching, and placeholders
- Avoid heavy work in build()

Key Optimization Techniques

- Resize images before use

Avoid loading 2000px images for 50px thumbnails.

```
Image.asset(  
  'assets/category.png',  
  width: 48,  
  height: 48,  
)
```

- Pre-cache frequently used images
 - precacheImage(const AssetImage('assets/category.png'), context);
- Use FadeInImage for remote images

```
FadeInImage.assetNetwork(  
  placeholder: 'assets/loading.png',  
  image: task.imageUrl,  
)
```

- Avoid decoding images in build()

→ Decode once in initState if needed.

Demo 12.3 - App Size Analysis

Objectives

- Generate size analysis report using Flutter CLI
- Understand breakdown of: code size, assets, packages
- Identify unnecessary dependencies or large assets

1. Build debug vs release

```
flutter build apk --debug  
flutter build apk --release
```

2. Analyze app size

```
flutter build apk --analyze-size
```

Output: JSON + HTML tree map file Dart AOT code size

- Native libraries
- Assets folder breakdown
- Package contributions

3. Common Size Optimization Tips

- Remove unused assets
- Compress large images
- Delete unused dependencies from pubspec.yaml
- Enable tree-shaking icons (Material icons off)

```
flutter:  
  uses-material-design: false
```

Demo 12.4: Building & Deploying The Release App (Taskly)

Objectives

- Understanding Release vs Debug vs Profile modes
- Build Release APK / AppBundle
- Configure app version
- (Optional) Sign the application for Google Play
- Validate final build size and performance

1. Preparing Project for Release

Check versioning

File: android/app/build.gradle

```
defaultConfig {  
    versionCode 2  
    versionName "1.1.0"  
}
```

Version rules:

- `versionCode` = integer (must increase every release)
- `versionName` = display version

2. Run App in Profile Mode

Command

```
flutter run --profile
```

Purpose

- Runs the app with near-release performance
- Removes debugging overhead
- Helps you check smoothness, scrolling, and build/render cost

Notes

- Profile mode is recommended before building the final Release APK
- No code changes required, just run and observe the behavior

Demo 12.4 (continued)

3. Build Release APK

- **Command**

```
flutter build apk --release
```

- **Output location:**

```
build/app/outputs/flutter-apk/app-release.apk
```

- **What students should verify:**

- App starts instantly
 - Performance smoother than Debug mode
 - No debug banners, no hot reload

4. Build AppBundle (AAB) for Google Play

- **Google Play requires .aab instead of .apk.**

```
flutter build appbundle --release
```

- **Output:**

```
build/app/outputs/bundle/release/app-release.aab
```

Demo 12.4 (continued)

5. Optional - App Signing Configuration

- If publishing to Google Play, generate a keystore:

```
keytool -genkey -v -keystore key.jks -keyalg RSA -keysize 2048 -validity 10000
```

- Add file `android/app/src/main/keystore.properties`

```
storePassword=*****
keyPassword=*****
keyAlias=my-key-alias
storeFile=key.jks
```

- Update `build.gradle`

```
signingConfigs {
    release {
        storeFile file('key.jks')
        storePassword keystoreProperties['storePassword']
        keyPassword keystoreProperties['keyPassword']
        keyAlias keystoreProperties['keyAlias']
    }
}
```

Demo 12.4 (Completed)

After completing Demo 12.4, you will have:

- A fully optimized Release build of Taskly (APK/AAB)
- A verified performance-checked version using Profile mode
- A project configured and ready for real deployment
- → Your app is now production-ready.

Best Practices

1. Rendering

- Use const
- Extract widgets
- Avoid heavy work in build()

2. State & Rebuilds

- Use Selector / fine-grained updates
- Keep state close to where it's used

3. Lists

- Use ListView.builder
- Extract list items
- Use stable keys (ValueKey)

4. Images

- Resize & compress
- Pre-cache
- Avoid decoding in build()

5. App Size

- Remove unused assets/deps
- Enable icon tree-shaking

6. DevTools & Deployment

- Test in Profile mode
- Fix jank/rebuild hotspots
- Build Release APK/AAB and update versioning

Summary

In Module 12, students learned to:

- Understand Flutter's rendering pipeline and common performance bottlenecks
- Identify issues such as unnecessary rebuilds, heavy image decoding, inefficient lists
- Use DevTools for profiling FPS, memory, rebuilds, and CPU
- Apply practical optimizations: const, widget extraction, Selector, list tuning
- Analyze and reduce app size using Flutter's size tools
- Build Release APK/AAB, verify performance in Profile mode, and prepare the app for deployment
- Optimize the Taskly app and produce a production-ready release build

References

- Kevin Moore, *Mastering Flutter 2025: Learn to Develop Flutter Apps*, Packt Publishing.
→ Relevant chapters: Ch. 13 – Performance, Ch. 14 – DevTools, Ch. 15 – Deployment
- Flutter Docs
 - Rendering & Performance: <https://docs.flutter.dev/perf/>
 - DevTools: <https://docs.flutter.dev/tools/devtools/overview>
 - App Size: <https://docs.flutter.dev/perf/app-size>
 - Deployment (Android): <https://docs.flutter.dev/deployment/android>
- Packages
 - Provider: <https://pub.dev/packages/provider>
 - Flutter Test API: https://api.flutter.dev/flutter/flutter_test/flutter_test-library.html