

Mobile and Embedded Computing

**Lecture 4. Debugging & Tools, State management
techniques intro, Equatable**

Flutter Dev Tools

- Usually on <http://127.0.0.1:9100/home>
- You will need this URL that is printed in the console:

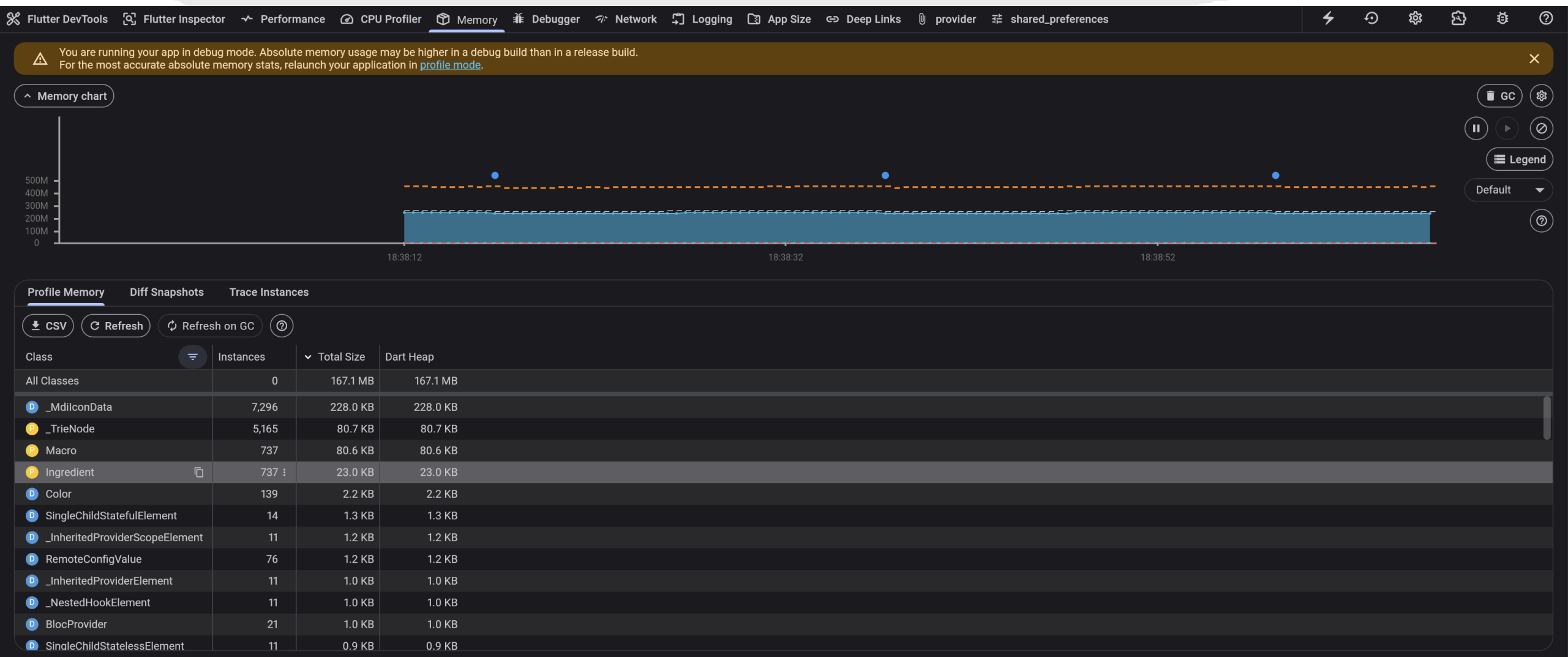


The screenshot shows the Flutter DevTools interface. At the top, there's a tab labeled 'App (Pixel 9a)' with a close button. Below the tab, there's a 'Console' panel. The console output shows the following log messages:

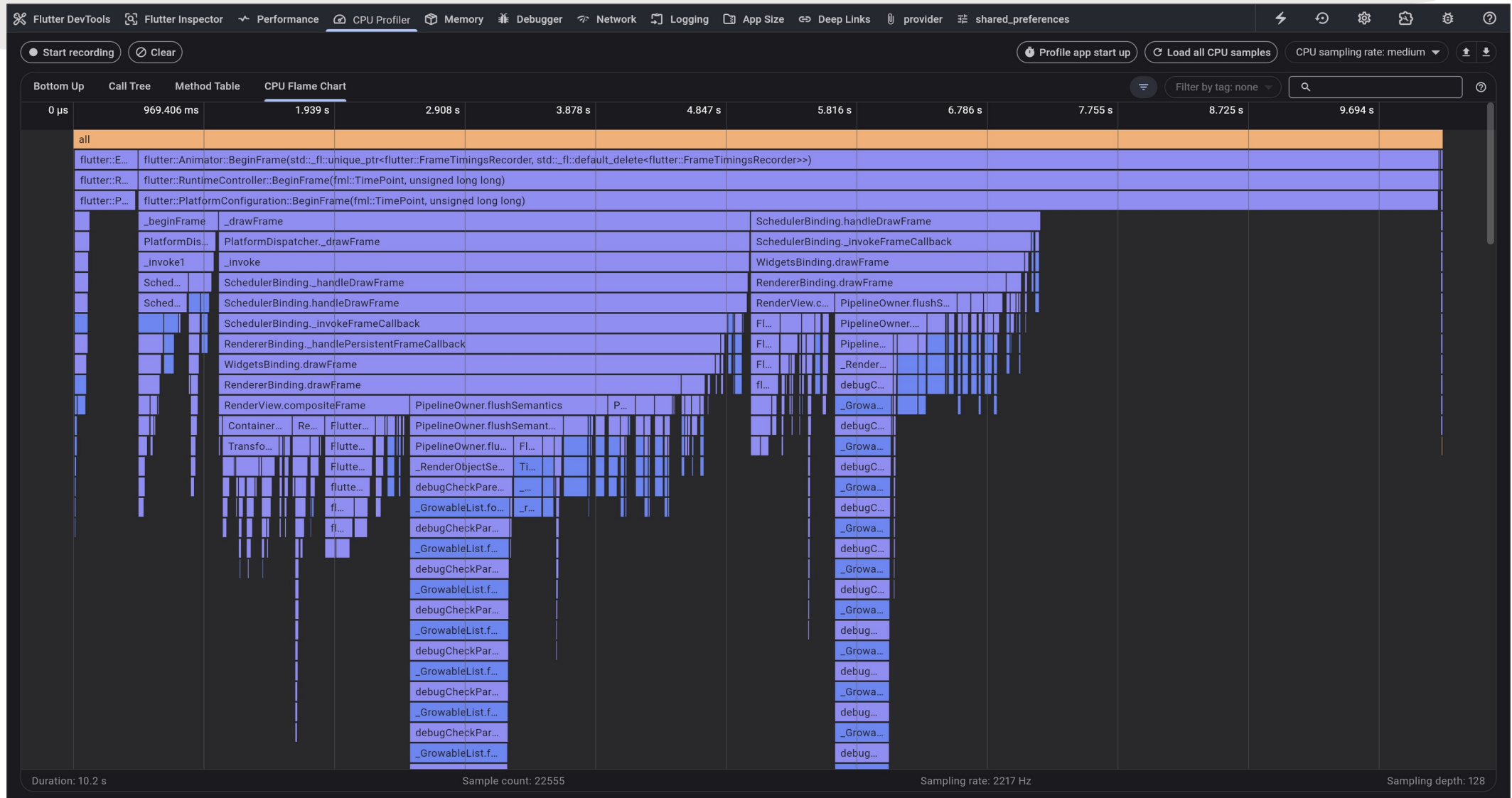
```
D/FlutterJNI(16715): Beginning load of flutter...
D/FlutterJNI(16715): flutter (null) was loaded normally!
I/flutter (16715): [IMPORTANT:flutter/shell/platform/android/android_context_gl_impeller
I/WindowExtensionsImpl(16715): Initializing Window Extensions, vendor API level=9, activ
Debug service listening on ws://127.0.0.1:60336/jFp9ew5_rNE=/ws
Syncing files to device sdk gphone64 arm64...
I/scanneralimente(16715): Compiler allocated 5111KB to compile void android.view.ViewRoc
D/WindowLayoutComponentImpl(16715): Register WindowLayoutInfoListener on Context=com.coc
```

The URL `ws://127.0.0.1:60336/jFp9ew5_rNE=/ws` is highlighted in blue in the original image.

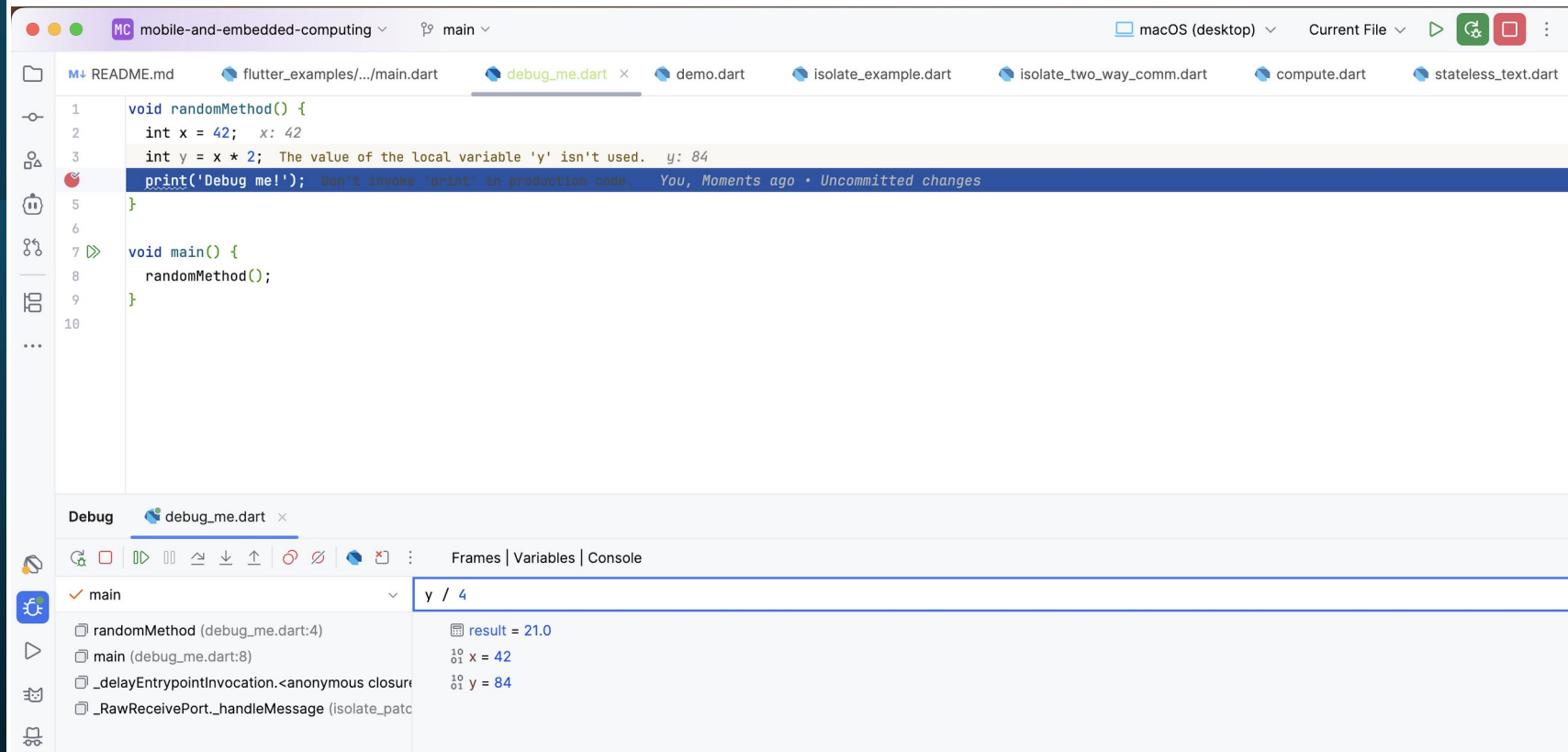
Flutter Dev Tools



Flutter Dev Tools



Using a debugger



Step Over –
Moves line by
line without
entering in any
method

```
1 void randomMethod() {
2   int x = 42;   x: 42
3   int y = x * 2; The value of the local variable 'y' isn't used.
4   anotherRandomMethod();
5   print('Debug me!'); Don't invoke 'print' in production code.
6 }
7
8 void anotherRandomMethod() {
9   String message = "Hello, Debugging!";
10  print(message); Don't invoke 'print' in production code.
11 }
12
13 void main() {
14   randomMethod();
15 }
16
```

Debug debug_me.dart

Step Over F8

main

randomMethod (debug_me.dart:5)

main (debug_me.dart:14)

_delayEntrypointInvocation.<anonymous closure>

Evaluate expression (⇧) or add

$10_{01} x = 42$

$10_{01} y = 84$

Step Into – If applied on a method, the debugger will move into that method

```
2      int x = 42;    x: 42
3      int y = x * 2; The value of the local variable x is 42
4      anotherRandomMethod(); You, 6 minutes ago
5      print('Debug me!'); Don't invoke 'print' in production
6  }
7
8  void anotherRandomMethod() {
9      String message = "Hello, Debugging!";
10     print(message); Don't invoke 'print' in production
11 }
12
13 >> void main() {
14     randomMethod();
15 }
16
```

Debug debug_me.dart x

Step Into F7

main

- randomMethod (debug_me.dart:4)
- main (debug_me.dart:14)
- _delayEntrypointInvocation.<anonymous closure>
- _RawReceivePort._handleMessage (isolate_patch.dart:261)

Evaluate expression

10 x = 42
01
10 y = 84
01

Step Out – Goes out of a method back to the previous one

```
1 void randomMethod() {
2   int x = 42;   x: 42
3   int y = x * 2; The value of the local variable 'y' isn't used.   y: 84
4   anotherRandomMethod();
5   print('Debug me!'); Don't invoke 'print' in production code.   You, 4 minutes
6 }
7
8 void anotherRandomMethod() {
9   String message = "Hello, Debugging!";
10  print(message); Don't invoke 'print' in production code.
11 }
12
13 void main() {
14   randomMethod();
15 }
```

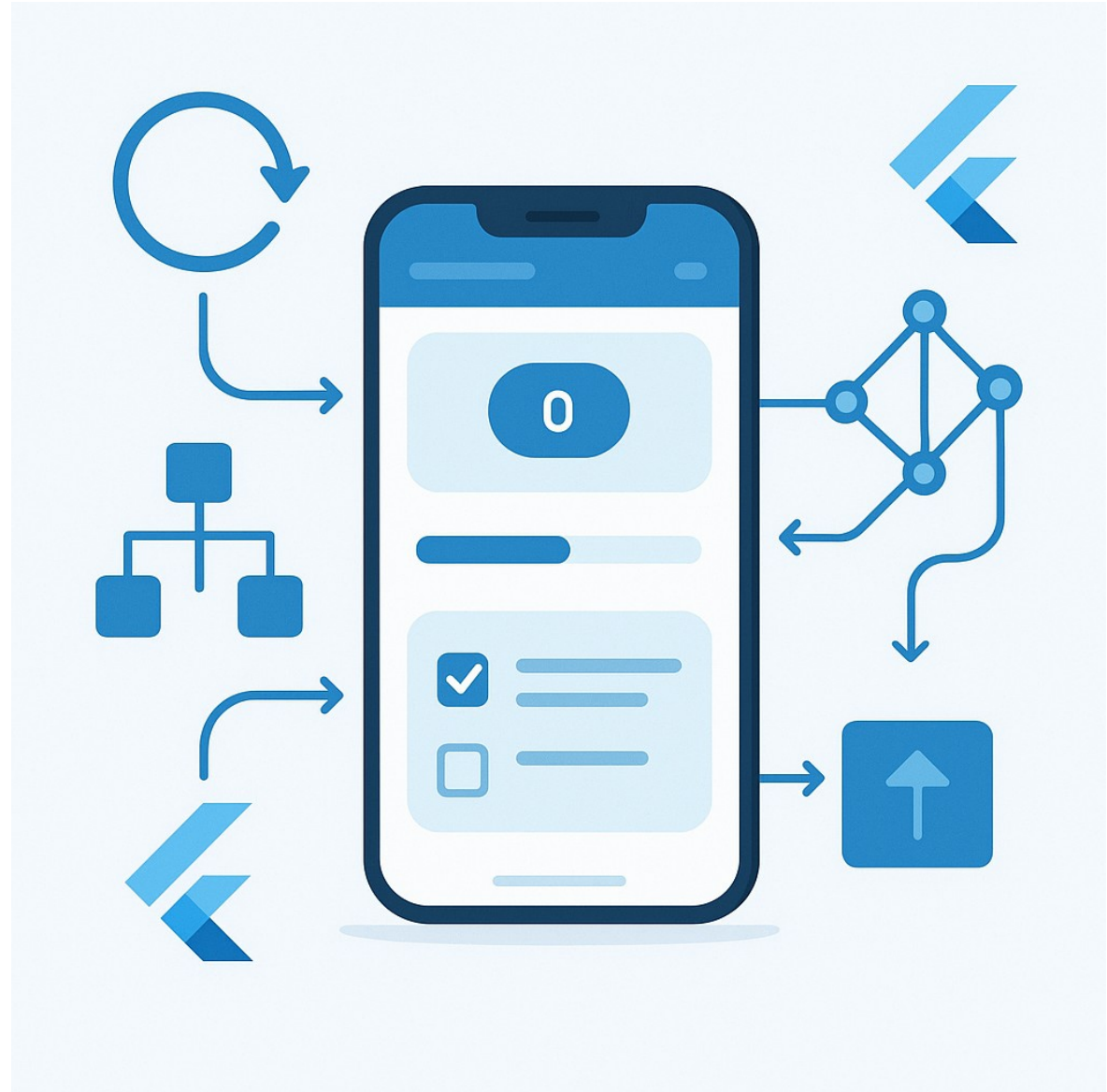
debug debug_me.dart x

Frames | Variables | Console

main Step Out ⌘F8 Evaluate expression (⌘E) or add a watch (⌘W)

randomMethod (debug_me.dart:5)	$\frac{10}{01}$ x = 42
main (debug_me.dart:14)	$\frac{10}{01}$ y = 84
_delayEntrypointInvocation.<anonymous closure>	
_RawReceivePort._handleMessage (isolate_patch.dart:288)	

State managem nt techniques



+

•

○

Introduction to State

State is the data that changes over time in your app.

Whenever that data changes, your UI must update to reflect it.

Example:

The number on a counter app.

The text in a form field.

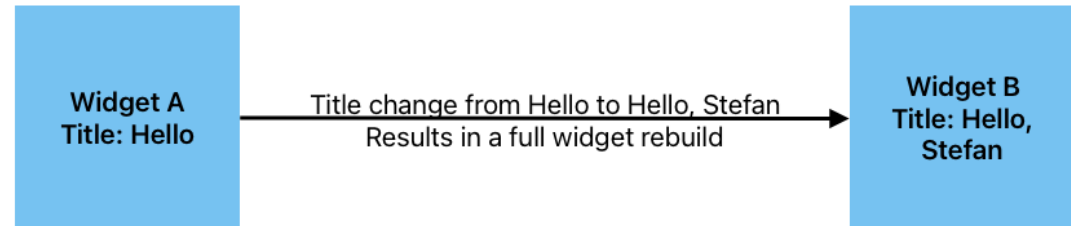
Whether a user is logged in or not.

In Flutter, widgets themselves are immutable once created, they can't change.

So when the state changes, Flutter rebuilds widgets with the new data.

What is Immutability?

- “Immutable” means unchangeable, once an object is created, it cannot be modified.
- Instead of changing the object, Flutter creates a new instance when something changes.



Why are widgets immutable?

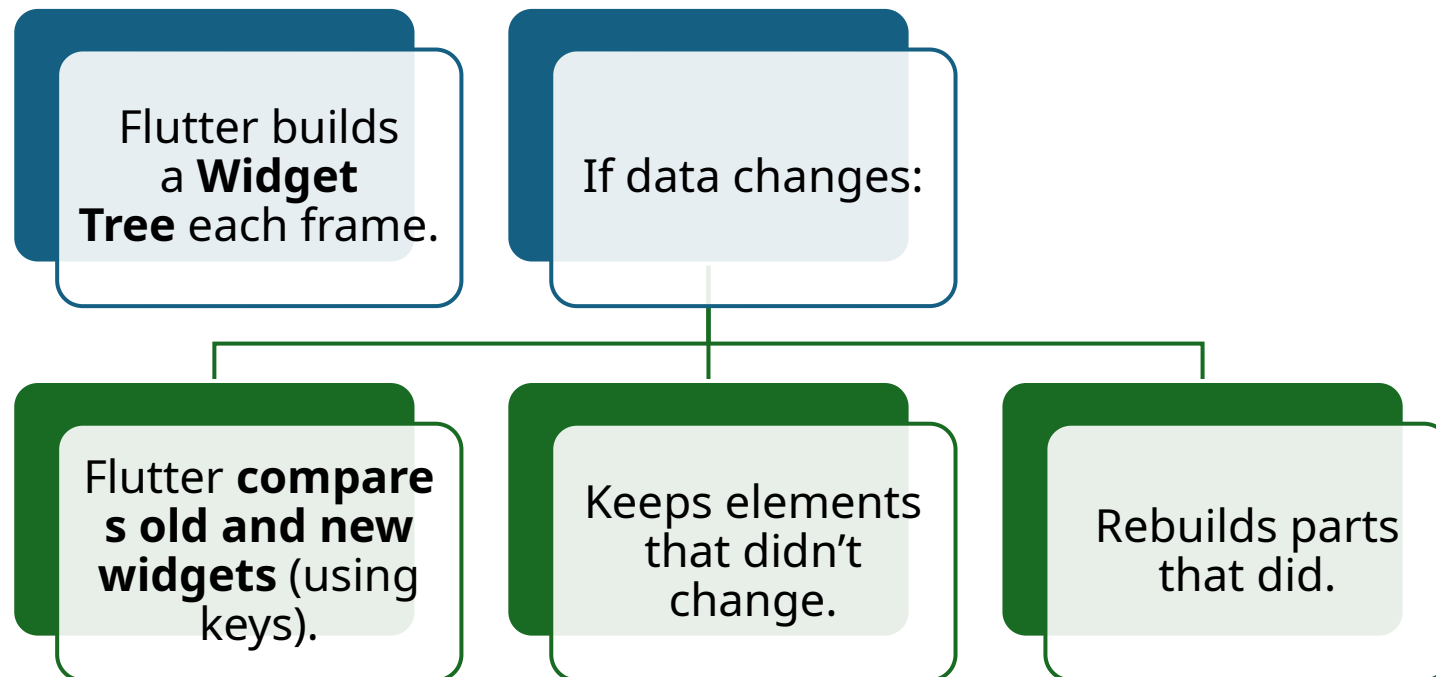
Predictability:
UI doesn't
mutate
unexpectedly

Performance:
Flutter can
efficiently
rebuild only
what
changes.

Simplicity: No
complex
change
tracking; just
rebuild.

Debugging
ease: Each UI
frame is a
pure function
of its state.

Widget Lifecycle and Rebuilds





Type	Description	Example Use
StatelessWidget	Immutable, builds once.	Labels, Icons
StatefulWidget	Has internal state; triggers rebuilds.	Forms, Buttons



-
- Widgets are blueprints for UI elements, not the actual UI on screen.
 - When you “change” something, Flutter just uses a new blueprint to draw the updated UI.

```
// Stateless: fixed data
class TitleText extends StatelessWidget {
  final String title;
  const TitleText({super.key, required this.title});

  @override
  Widget build(BuildContext context) => Text(title);
}

// Stateful: can change data
class CounterWidget extends StatefulWidget {
  const CounterWidget({super.key});

  @override
  CounterWidgetState createState() => CounterWidgetState();
}
```

Types of state

Type	Description	Example
Ephemeral (Local) State	State that lives in a single widget.	Toggle switch, counter, checkbox.
App (Global) State	Shared across multiple widgets or screens.	Logged-in user, theme mode, shopping cart.

setState() – The Basics

- The setState() method is the most basic way to manage state in Flutter.
- It's used inside a StatefulWidget to update local state.
- Logic and UI get mixed together.
- Hard to reuse state in other widgets.
- Doesn't scale for complex apps.

```
int _count = 0;

void _increment() {
    setState(() {
        _count++;
    });
}
```

Lifting State Up

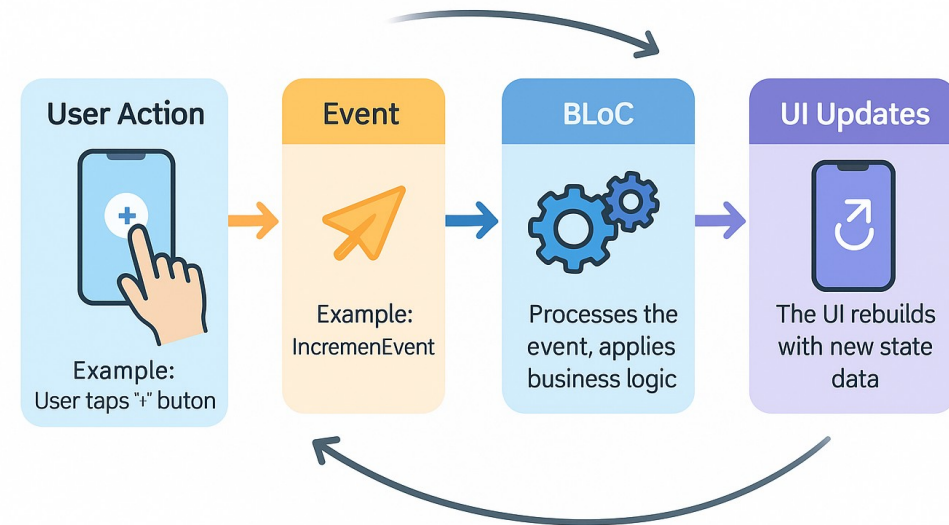
- When two widgets need to share the same data, you can lift state up to their common parent.
- Keeps widgets reusable and independent.
- Promotes clear data flow: Parent → Child (data) and Child → Parent (events).
- If you lift state up too far, it becomes hard to manage. Passing data through many layers (“prop drilling”).

```
class ParentWidget extends StatefulWidget {  
  const ParentWidget({super.key});  
  // You, Moments ago • Uncommitted changes  
  @override  
  ParentWidgetState createState() => ParentWidgetState();  
}  
  
class ParentWidgetState extends State<ParentWidget> {  
  bool _active = false;  
  
  void _handleTapboxChanged(bool newValue) {  
    // Typo: In widget  
    setState(() {  
      _active = newValue;  
    });  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Container(  
      child: ChildWidget(  
        active: _active,  
        onChanged: _handleTapboxChanged,  
      ),  
    );  
  }  
}
```

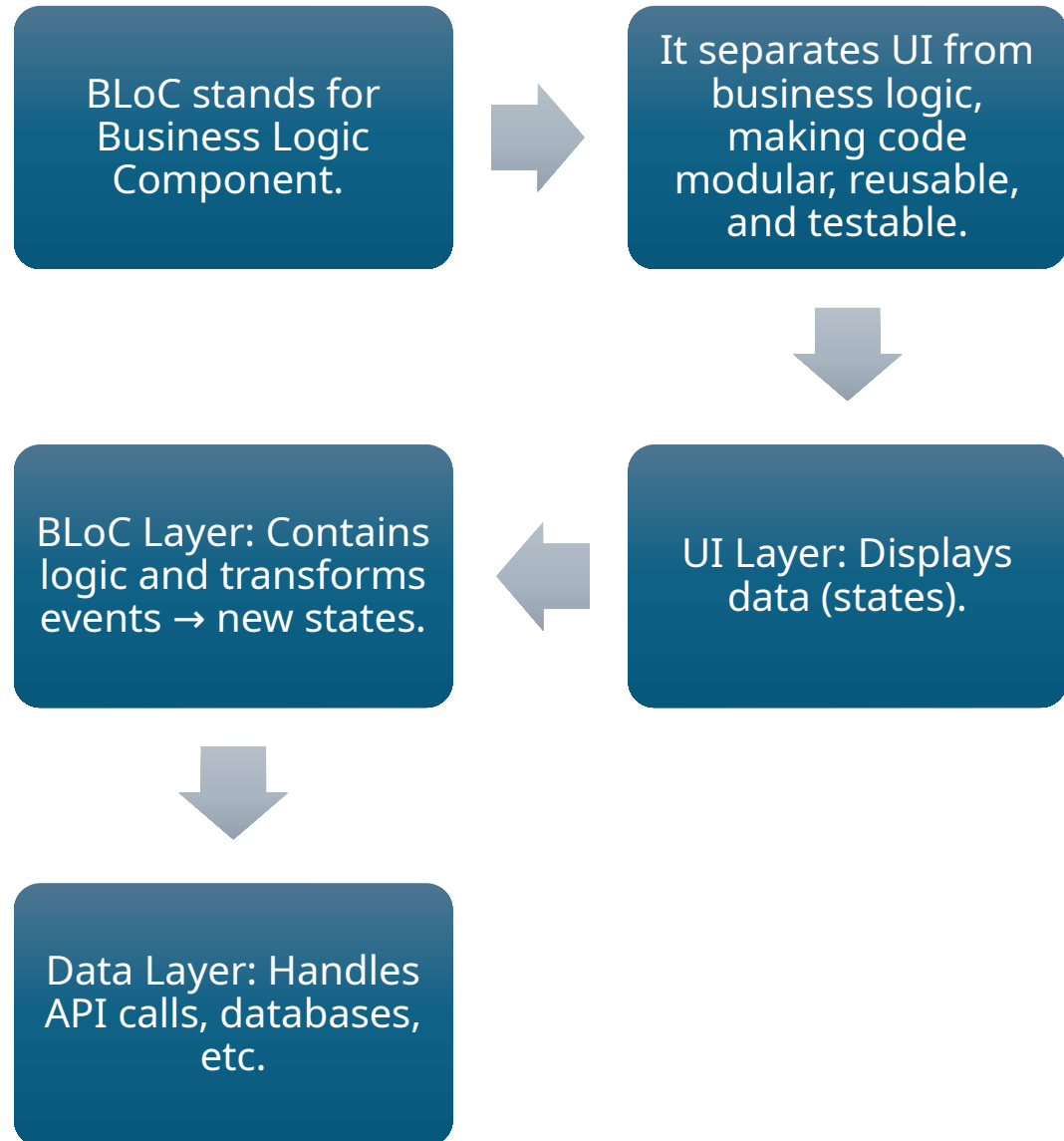
```
class ChildWidget extends StatelessWidget {  
  final bool active;  
  final ValueChanged<bool> onChanged;  
  
  const ChildWidget({super.key, required this.active, required this.onChanged});  
  
  void _handleTap() {  
    onChanged(!active);  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return GestureDetector(  
      onTap: _handleTap,  
      child: Container(  
        width: 150,  
        height: 150,  
        color: active ? Colors.green : Colors.grey,  
        child: Center(  
          child: Text(  
            active ? 'Active' : 'Inactive',  
            style: TextStyle(fontSize: 24, color: Colors.white),  
          ),  
        ),  
      ),  
    );  
  }  
}
```

BLoC (& Cubit pattern)

- Separates UI from business logic with a stream-based approach
- BLoC: Uses events and states via Stream and Sink.
- Cubit: A simplified BLoC that emits states directly.



BLoC



Cubit

Cubit is a simpler version of BLoC.

Cubit: Emits new states directly (no separate events).

BLoC: Works with **events** and **states**, offering more structure.

Bloc & Cubit Required dependencies

- You can add new dependencies in pubspec.yaml
- Then you will need to run flutter pub get to fetch them
- Upgrading dependencies is done using flutter pub upgrade

```
dependencies:  
  flutter_bloc: ^9.1.1  
  equatable: ^2.0.7
```

Terminal Local × + ▾

```
dinu@Mac mobile-and-embedded-computing % flutter pub get
```

Equatable

- In Flutter BLoC, the UI only rebuilds when the state changes. But how does Flutter know if the new state is different from the previous one?
- Without Equatable, Dart's default equality check compares object references, not values. So even if two CounterState(1) objects have the same count, Flutter thinks they're different, causing unnecessary rebuilds. Equatable fixes this by comparing objects by value.

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

// without equatable - and Flutter won't know
// that the state has changed
class CounterStateBad {
  final int count;
  CounterStateBad(this.count);
}

// with equatable - Flutter knows that the state has changed
class CounterState extends Equatable {
  final int count;
  const CounterState(this.count);

  @override
  List<Object> get props => [count];
}
```

You, Moments ago • Uncommitted changes

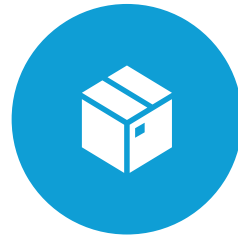
Best Practices and Recommendations



KEEP UI (WIDGETS)
AND LOGIC
SEPARATE.



USE IMMUTABLE
DATA MODELS
WHEN POSSIBLE.



DISPOSE STREAMS
AND CONTROLLERS
PROPERLY.



START SIMPLE
(SETSTATE →
RIVERPOD/BLOC).



PICK WHAT FITS
YOUR PROJECT.