



TRƯỜNG ĐẠI HỌC BÁCH KHOA - ĐẠI HỌC ĐÀ NẴNG

KHOA ĐIỆN TỬ - VIỄN THÔNG

LẬP TRÌNH ĐA NỀN TẢNG

Flutter Architecture & Widget Tree

SINH VIÊN THỰC HIỆN:

NGUYỄN VIỆT HOÀNG 22KTMT1 106220216

LÊ MINH NHẬT 22DT2 106220066

GVHD: TS. NGUYỄN DUY NHẬT VIỄN

ĐÀ NẴNG, 10/2025



BẢNG PHÂN CÔNG VIỆC TRONG NHÓM

| STT | HỌ VÀ TÊN | NHIỆM VỤ | KHỐI LƯỢNG |
|-----|-------------------|---|------------|
| 17 | NGUYỄN VIỆT HOÀNG | Tìm hiểu và giải thích 4 tầng kiến trúc Flutter (Application, Framework, Engine, Embedder). Demo Widget Tree với MaterialApp, Scaffold, AppBar. Hoàn thiện báo cáo và slide các nội dung. | 65% |
| 07 | LÊ MINH NHẬT | Tìm hiểu lý thuyết và so sánh StatelessWidget với StatefulWidget. Widget lifecycle và rebuild mechanism. | 35% |

Link code github: https://github.com/VietHoang301/ltdntbtuan11_12

NỘI DUNG

- 1. Flutter là gì?**
- 2. Kiến trúc của Flutter**
- 3. Demo Widget Tree với MaterialApp, Scaffold, AppBar**
- 4. StatelessWidget với StatefulWidget**
- 5. Widget lifecycle và rebuild mechanism**

1. Flutter là gì?:

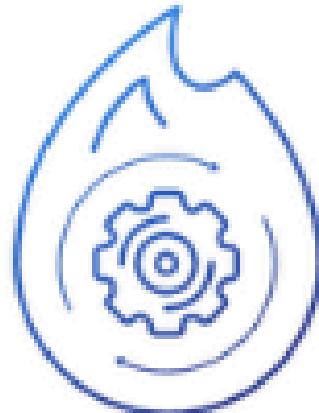
Flutter là một bộ công cụ phát triển UI (khung ứng dụng) mã nguồn mở của Google, cho phép xây dựng ứng dụng đa nền tảng (như iOS, Android, web, desktop) từ một cơ sở mã duy nhất.



Đặc điểm chính

Đa nền tảng

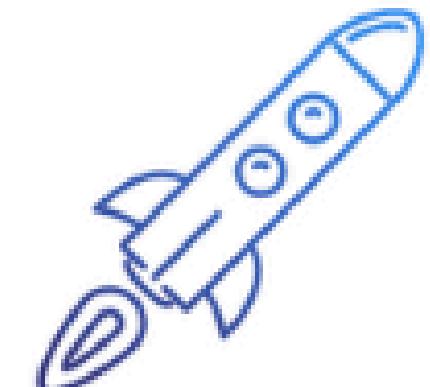
Flutter cho phép phát triển ứng dụng trên **nhiều nền tảng** như iOS, Android, web và desktop từ một mã nguồn duy nhất, giúp tiết kiệm thời gian và công sức.



Hot Reload

Hiệu suất cao

Flutter cung cấp hiệu suất vượt trội nhờ vào khả năng biên dịch **native code** và tối ưu hóa đồ họa, giúp ứng dụng mượt mà và nhanh chóng trên mọi thiết bị.



Fast development

Giao diện đẹp

Bộ widget phong phú và khả năng tùy biến cao, Flutter cho phép tạo ra các **giao diện bắt mắt** và thân thiện, nâng cao trải nghiệm người dùng.



Screen reader



Quick rendering

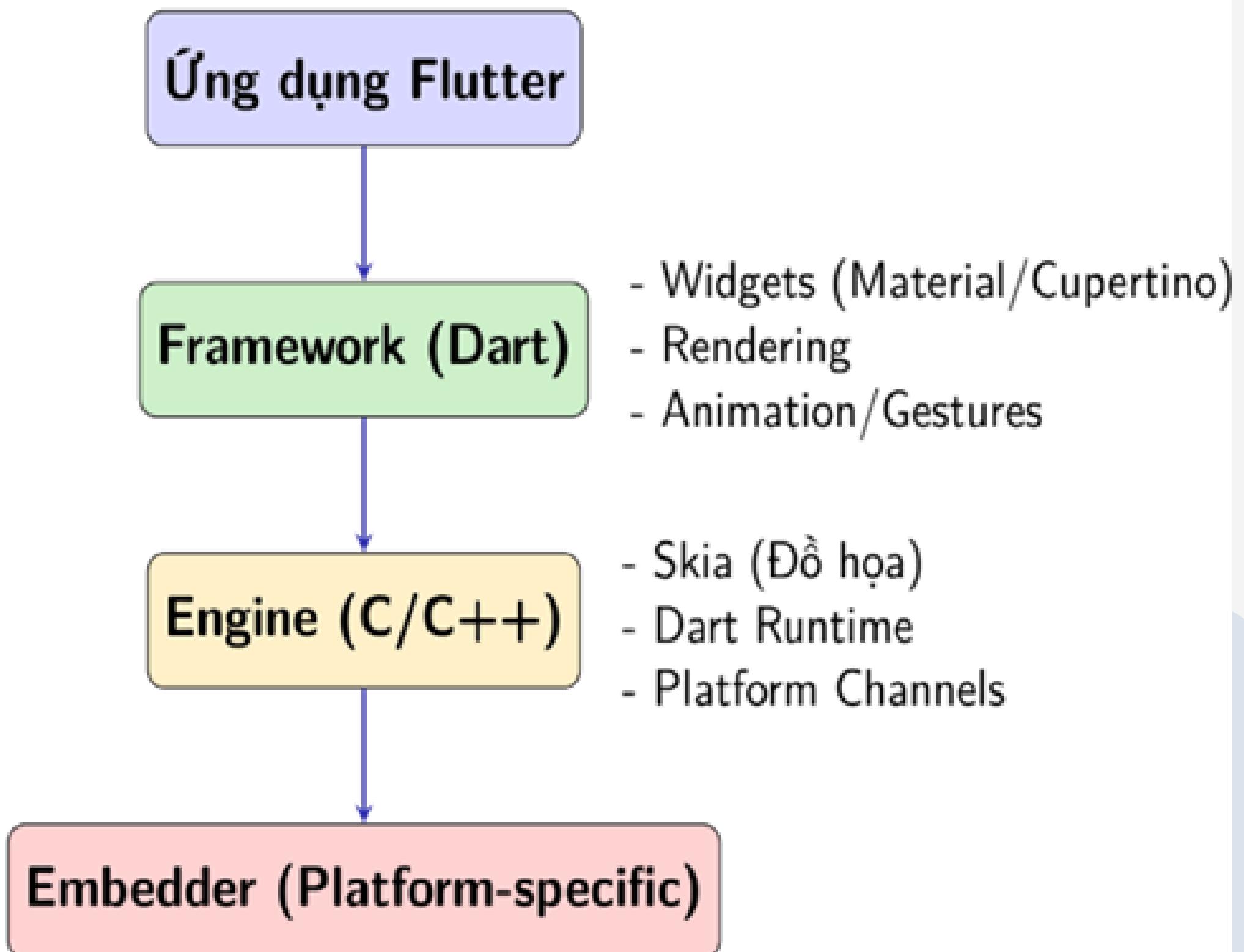
2. Kiến trúc của Flutter:

Flutter có 4 tầng chính:

Application →

Framework → Engine →

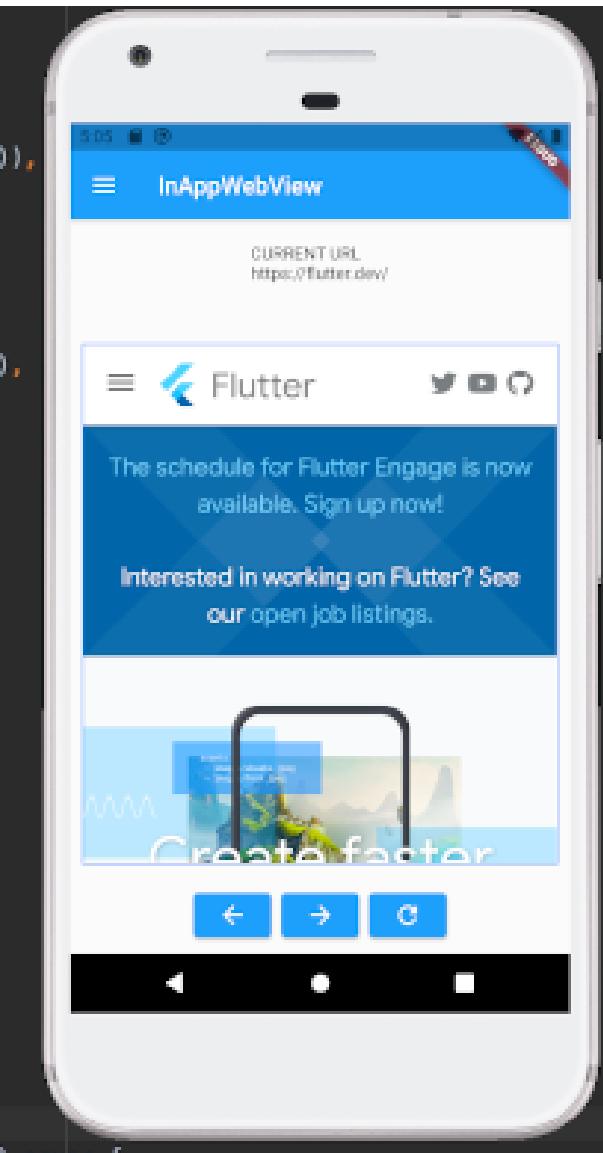
Embedder



Tầng 1: Application layer (Tầng ứng dụng):

Đây là nơi toàn bộ mã nguồn và logic của ứng dụng được viết ra. Nó không chỉ đơn thuần là các widget, mà là một hệ thống hoàn chỉnh.

```
Expanded(  
  child: Container(  
    margin: const EdgeInsets.all(10.0),  
    decoration:  
      BoxDecoration(border: Border.all(color: Colors.blueAccent)),  
    child: InAppWebView(  
      key: webViewKey,  
      initialURLRequest: URLRequest(  
        url: Uri.parse("https://flutter.dev"))  
      ), // URLRequest  
      initialUserScripts: UnmodifiableListView<UserScript>([]),  
      initialOptions: InAppWebViewGroupOptions(  
        crossPlatform: InAppWebViewOptions(  
          useShouldOverrideUrlLoading: false,  
          mediaPlaybackRequiresUserGesture: false,  
        ), // InAppWebViewOptions  
        android: AndroidInAppWebViewOptions(  
          useHybridComposition: true,  
        ), // AndroidInAppWebViewOptions  
        ios: IOSInAppWebViewOptions(  
          allowsInlineMediaPlayback: true,  
        ), // IOSInAppWebViewOptions  
      ), // InAppWebViewGroupOptions  
      onWebViewCreated: (controller) {  
        webView = controller;  
        print("onWebViewCreated");  
      },  
      onLoadStart: (controller, url) {  
        print("onLoadStart $url");  
        setState(() {  
          this.url = url.toString();  
        });  
      },  
      shouldOverrideUrlLoading: (controller, navigationAction) {  
        return false;  
      },  
    ),  
  ),  
);
```



Tầng 1: Application layer (Tầng ứng dụng):

Giao diện người dùng

Flutter cho phép **tạo ra giao diện** người dùng mượt mà và trực quan. Các widget tùy chỉnh có thể được xây dựng để đáp ứng nhu cầu cụ thể của ứng dụng.

Xử lý logic

Logic ứng dụng được quản lý thông qua **quy trình đơn giản** và rõ ràng. Flutter sử dụng Dart để xử lý các sự kiện và cập nhật trạng thái một cách hiệu quả.

Quản lý trạng thái

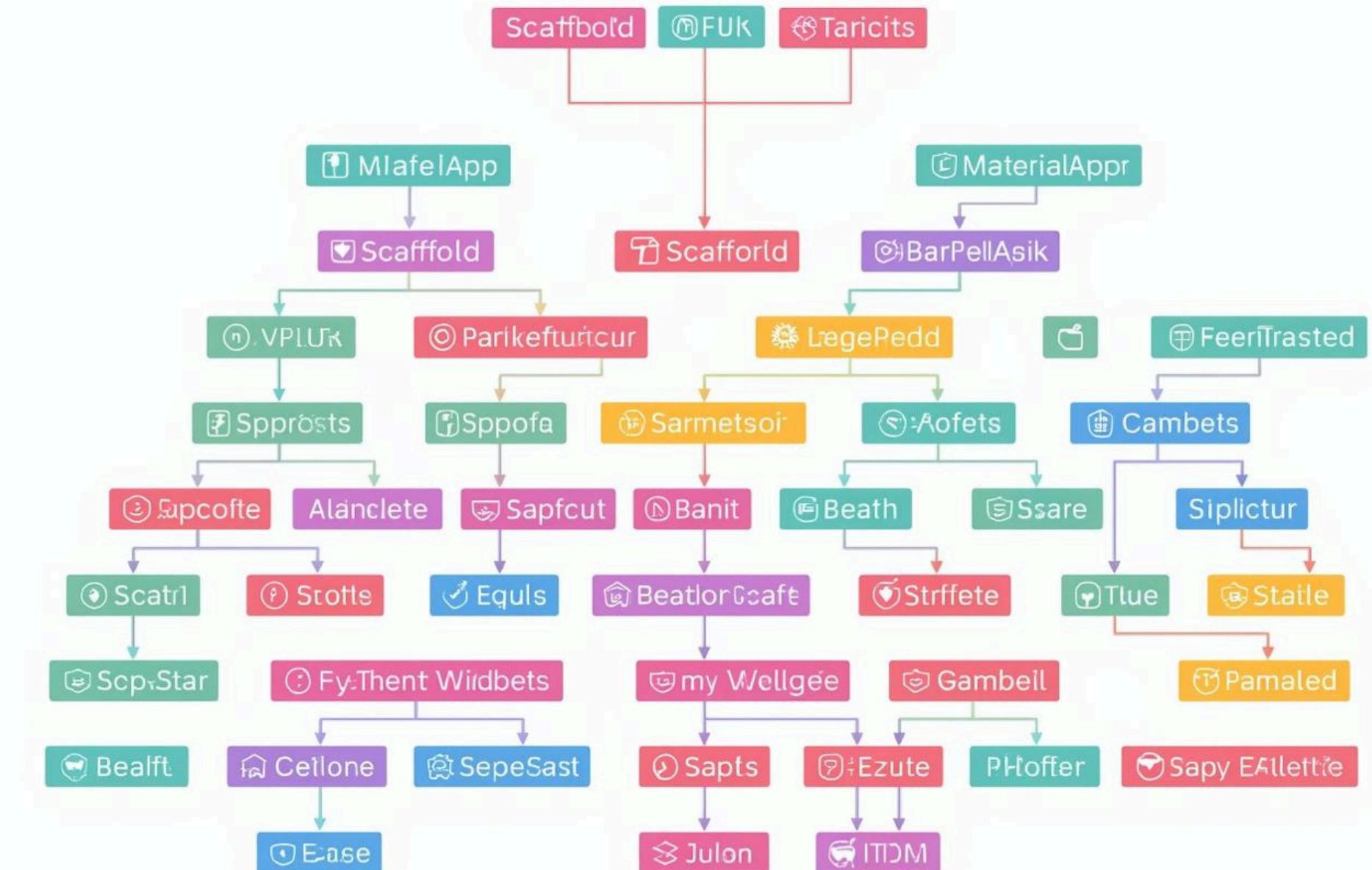
Quản lý trạng thái trong Flutter cho phép **dễ dàng theo dõi** sự thay đổi của dữ liệu. Các phương pháp như setState và Provider giúp đảm bảo UI luôn nhất quán.

Tầng 1: Application layer (Tầng ứng dụng):

Tập trung vào việc tạo ra cây widget (Widget Tree) để mô tả UI. Khi trạng thái ứng dụng thay đổi thì thực hiện cập nhật trạng thái đó và Flutter sẽ tự động tìm cách hiệu quả nhất để vẽ lại những phần giao diện bị ảnh hưởng.

Flutter widget widget

Enation senthfectial color in the peoples.



Tầng 2: Framework layers (Tầng khung):

Tầng Framework (viết bằng Dart) cung cấp bộ widget (Text, Container, Row, Column, ListView, Scaffold, Material/Cupertino), xử lý rendering qua Widget → Element → RenderObject, và cung cấp API cao cấp cho animation, gestures, painting.



Tầng 2: Framework layers (Tầng khung):

Khi bạn gọi **runApp()**, tầng Framework bắt đầu hoạt động. Nó xây dựng cây widget, sau đó tạo ra cây Element tương ứng. Khi **setState()** được gọi, nó so sánh cây widget mới với cây cũ, tìm ra sự khác biệt và chỉ ra lệnh cho tầng Engine bên dưới vẽ lại những phần thực sự đã thay đổi.

Widget Tree:

Cây cấu hình bất biến mà bạn tạo ra trong mã.

Element Tree:

Một cây trung gian, lưu giữ trạng thái của UI và tham chiếu đến widget và đối tượng render.

RenderObject Tree:

Cây này chịu trách nhiệm tính toán bố cục (kích thước, vị trí) và vẽ các widget lên màn hình.

Tầng 3: Engine Layer (Tầng động cơ):

Đây là tầng cấp thấp, được viết chủ yếu bằng C++ để đạt được hiệu suất tối đa. Tầng Engine là cầu nối giữa thế giới Dart (Framework) và thế giới native (hệ điều hành). Nó thực hiện các chỉ thị từ tầng Framework thành các pixel trên màn hình.



Flutter's Rendering Evolution: Skia vs Impeller!

vs

Skia

- Reliable
- can stutter

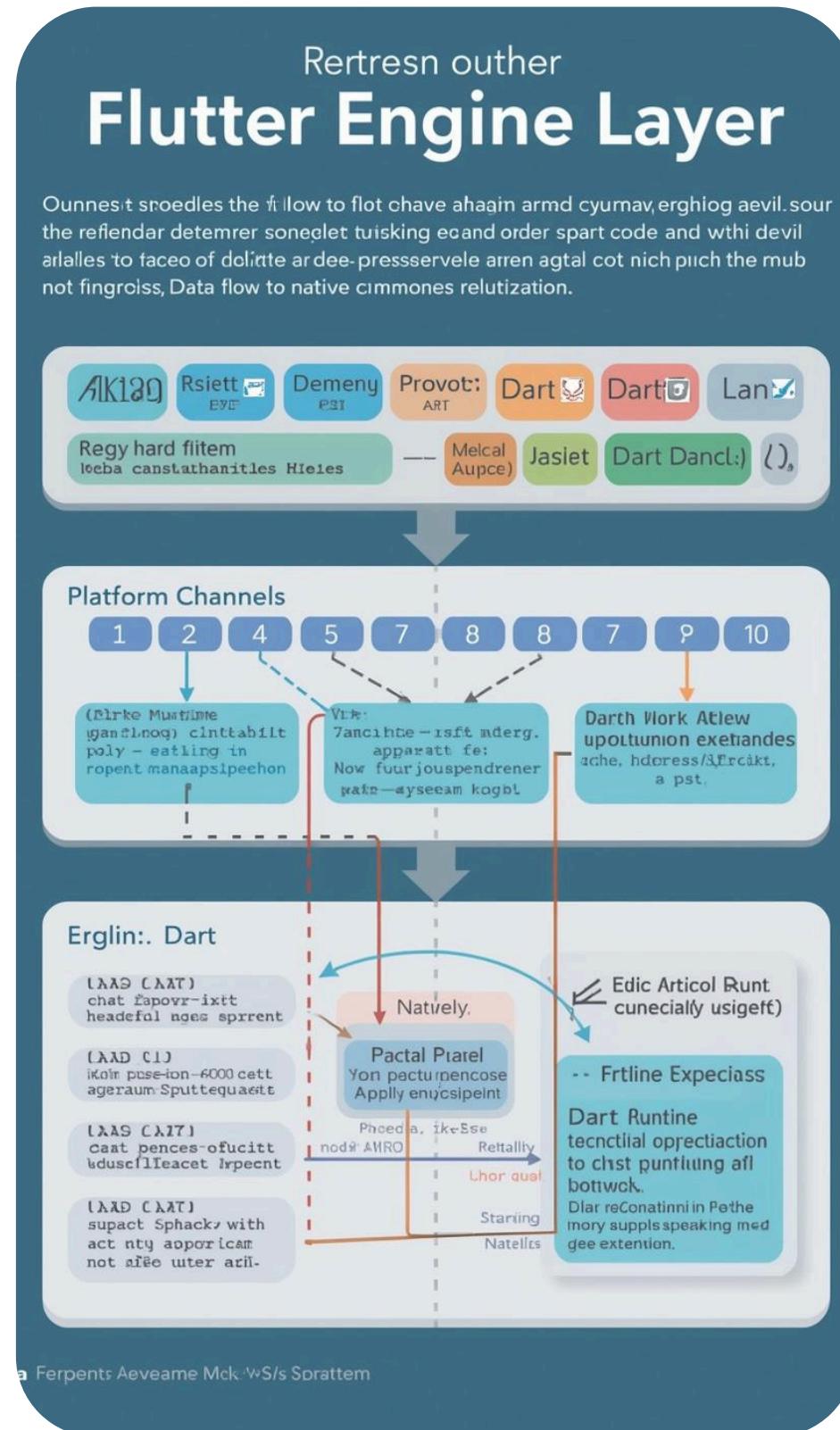
Impeller

- Future-proof
- buttery smooth

Tầng 3: Engine Layer (Tầng động cơ):

Vẽ đồ họa (Graphics Rendering):

Trách nhiệm quan trọng nhất của Engine là vẽ UI. Nó sử dụng thư viện đồ họa 2D mã nguồn mở Skia của Google để vẽ từng pixel lên màn hình.

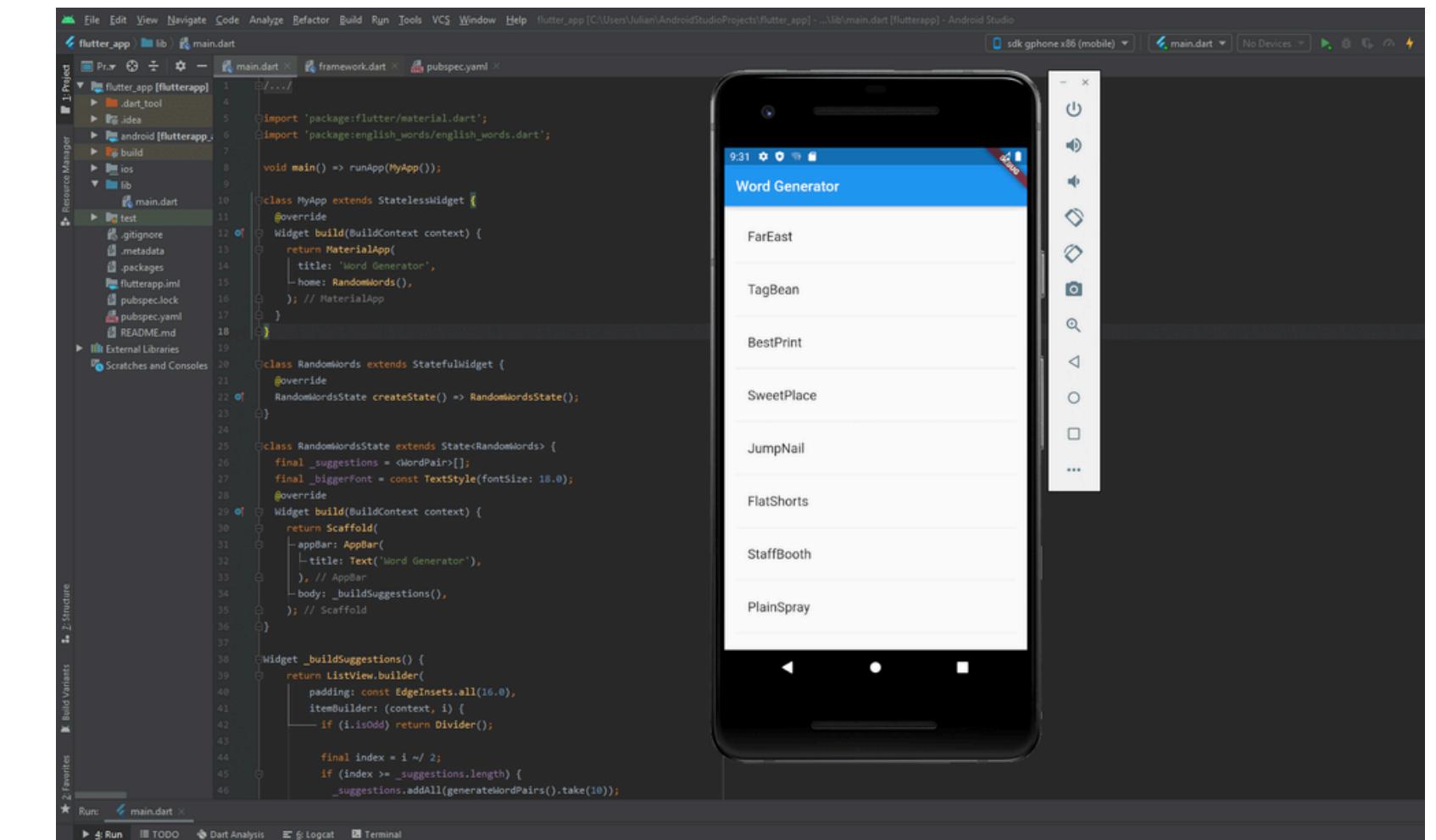


Cung cấp Dart Runtime:

Trong lúc phát triển, nó dùng trình biên dịch JIT (Just-In-Time) để kích hoạt tính năng Hot Reload. Khi phát hành, nó dùng trình biên dịch AOT (Ahead-Of-Time) để biên dịch mã Dart thành mã máy native.

Tầng 4: Embedder Layer (Tầng nhúng):

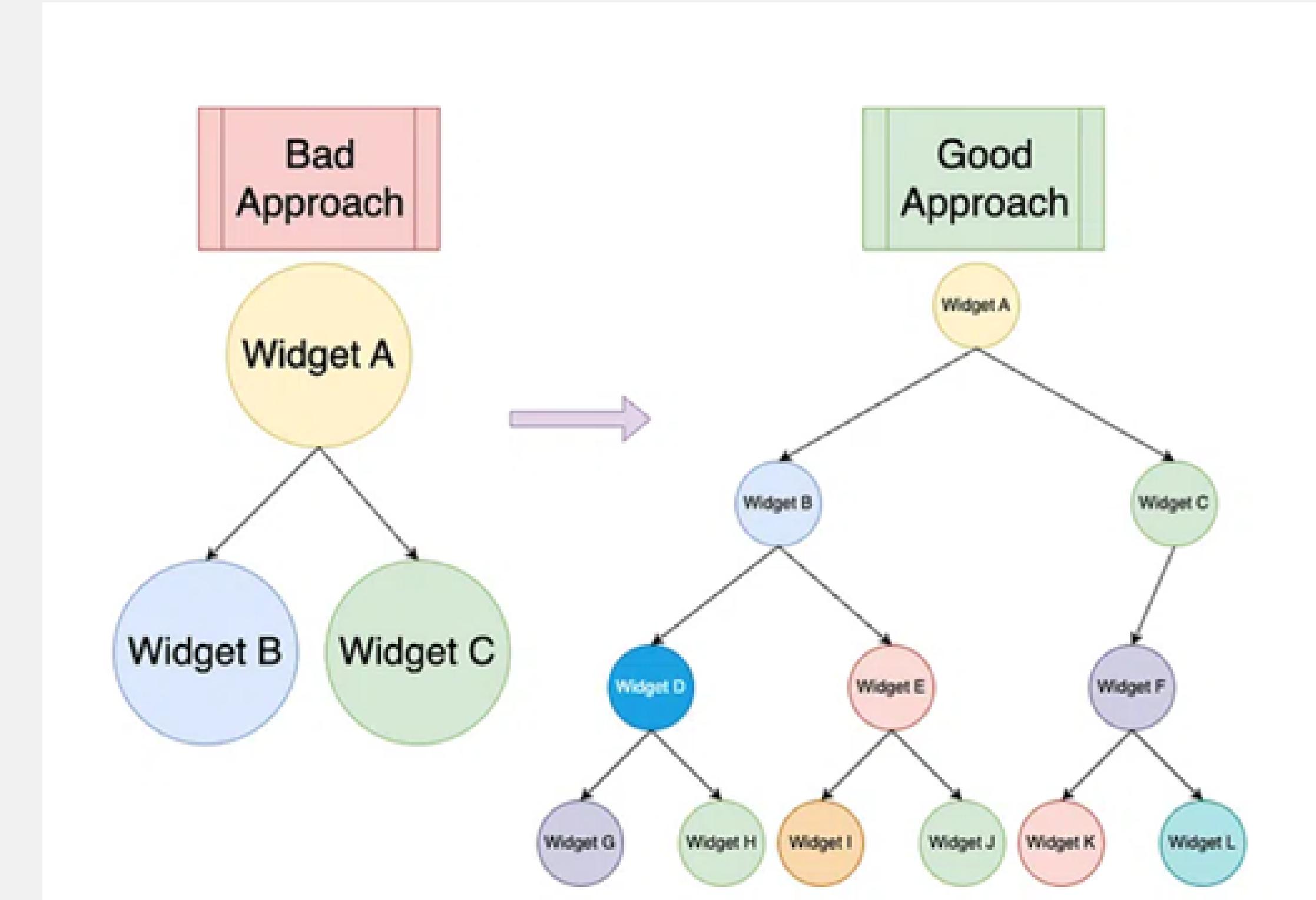
Embedder là lớp nhúng riêng cho từng nền tảng, hoạt động như một bộ chuyển đổi để nhúng Engine của Flutter vào ứng dụng gốc của hệ điều hành. Mỗi nền tảng (Android, iOS, Windows, Web) có một Embedder riêng.



Khởi tạo và nhúng Engine vào nền tảng, điều phối các sự kiện của hệ điều hành (như chạm, nhấn phím) và cung cấp một bề mặt trống để Engine có thể vẽ giao diện lên đó.

3. Demo Widget Tree với MaterialApp, Scaffold, AppBar:

Qua bức ảnh bên trên chúng ta thấy là có 2 cách để tạo một widget tree. Và để tiết kiệm thời gian thì hầu hết chúng ta sẽ làm theo cách đầu tiên.

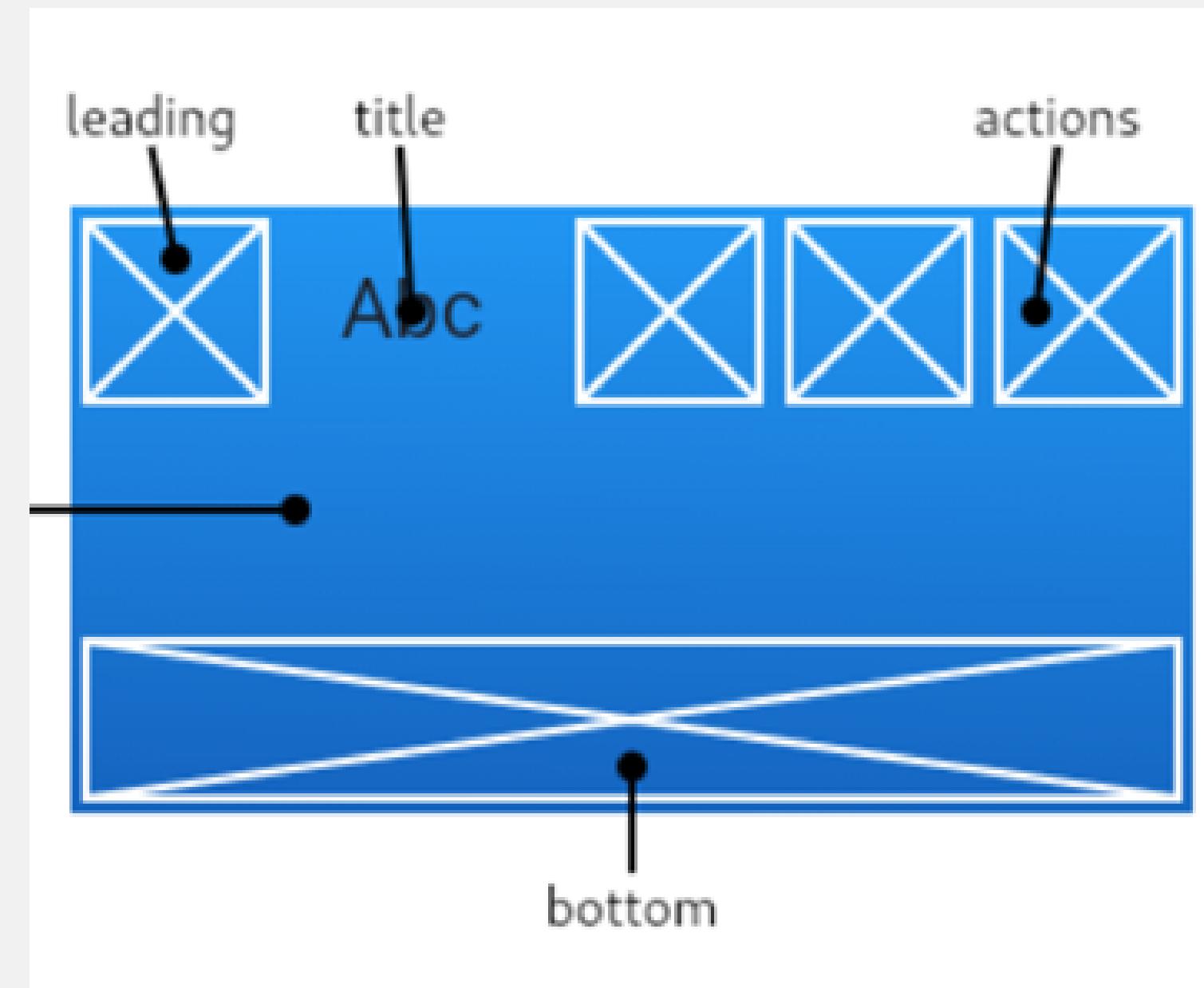


3. Demo Widget Tree với MaterialApp, Scaffold, AppBar:

MaterialApp: Widget gốc của ứng dụng, bao gồm điều hướng (navigation), chủ đề (theme), và các thiết lập chung.

Scaffold: cung cấp cấu trúc bố cục chuẩn, như thanh ứng dụng (appBar), thân (body), và nút hành động nổi (floatingActionButton).

AppBar: Widget thanh ứng dụng, hiển thị ở đầu màn hình. Nó thường chứa tiêu đề và các nút hành động.



3. Demo Widget Tree với MaterialApp, Scaffold, AppBar:

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

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

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: AppBarExample(),
    );
  }
}

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

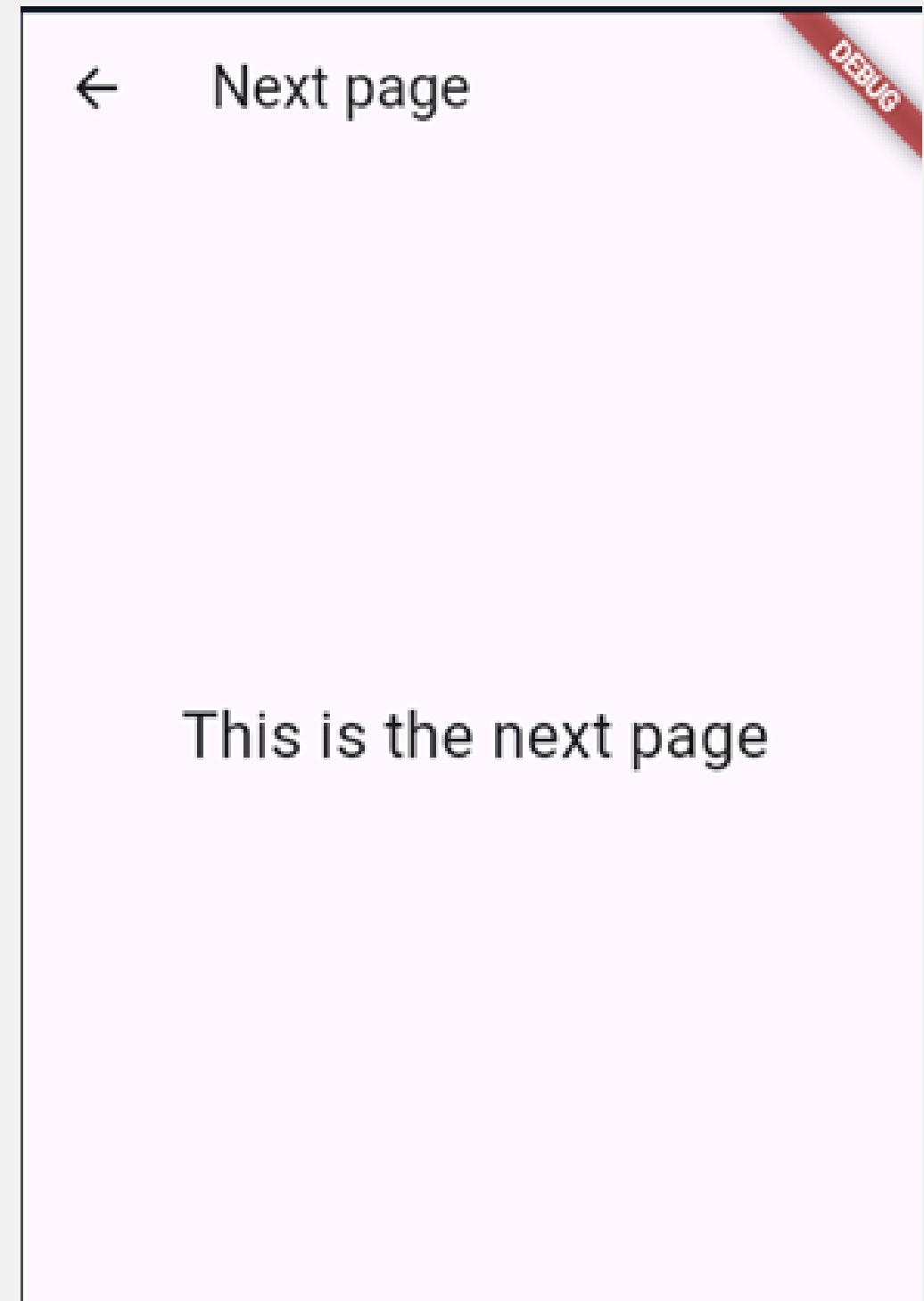
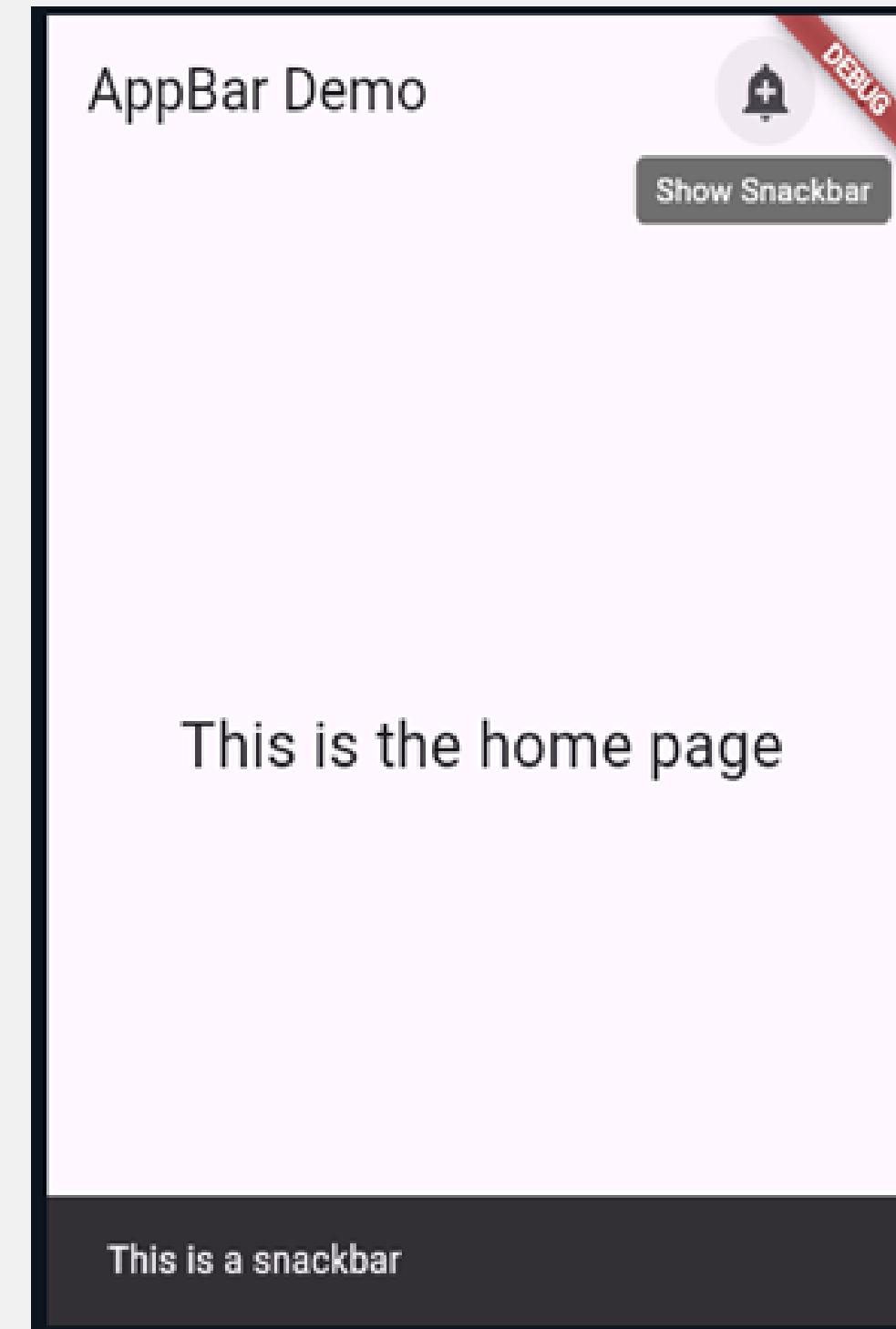
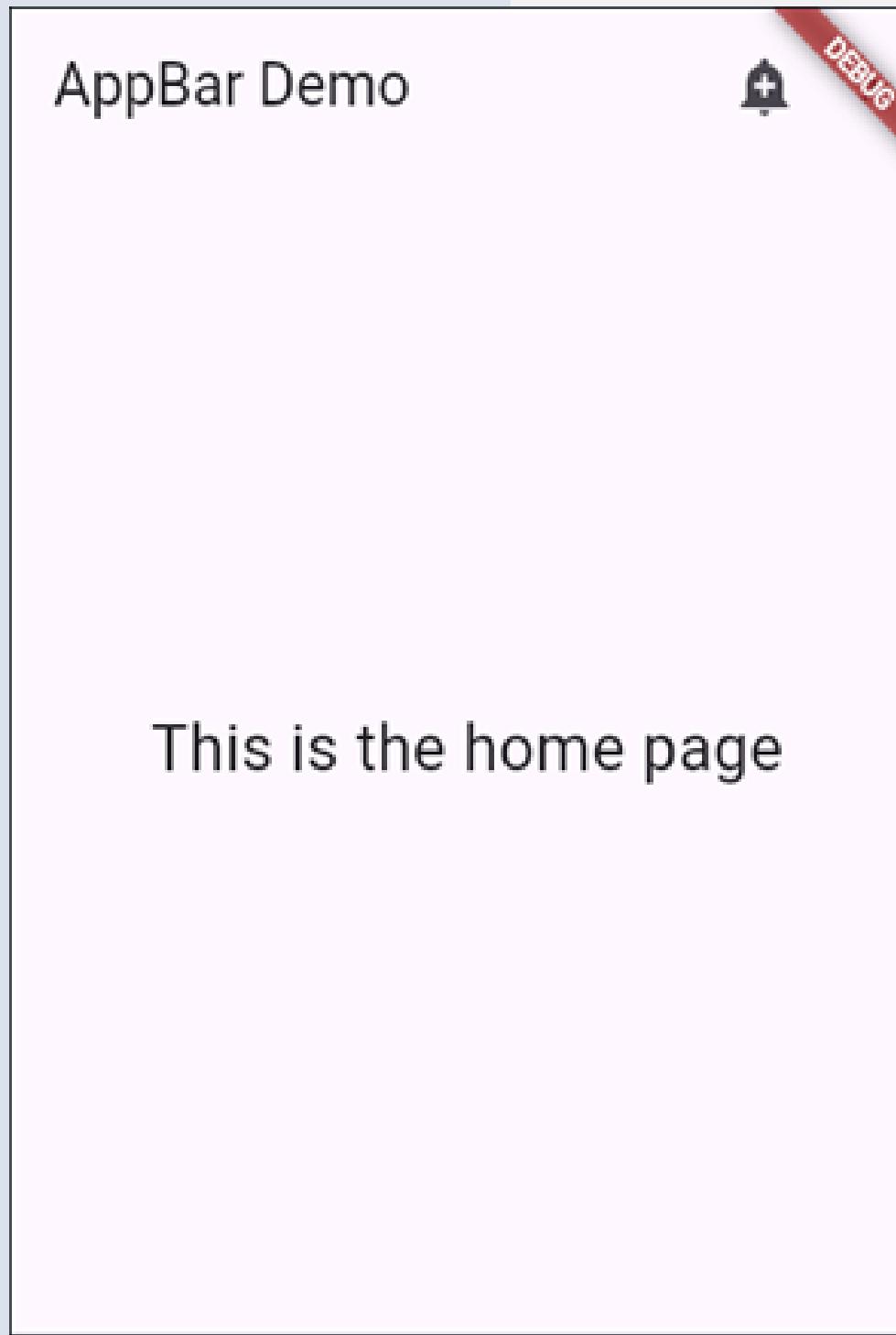
3. Demo Widget Tree với MaterialApp, Scaffold, AppBar:

```
@override  
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(  
      title: const Text('AppBar Demo'),  
      actions: <Widget>[  
        IconButton(  
          icon: const Icon(Icons.add_alert),  
          tooltip: 'Show Snackbar',  
          onPressed: () {  
            ScaffoldMessenger.of(context).showSnackBar(  
              const SnackBar(  
                content: Text('This is a snackbar'),
```

3. Demo Widget Tree với MaterialApp, Scaffold, AppBar:

```
IconButton(  
    icon: const Icon(Icons.navigate_next),  
    tooltip: 'Go to the next page',  
    onPressed: () {  
        Navigator.push(  
            context,  
            MaterialPageRoute<void>(  
                builder: (BuildContext context) {  
                    return Scaffold(  
                        appBar: AppBar(  
                            title: const Text('Next page'),  
                        ),  
                        body: const Center(  
                            child: Text(  
                                'This is the next page',  
                                style: TextStyle(fontSize: 24),  
                            ),  
                        ),  
                ),  
            );  
    },  
);
```

3. Demo Widget Tree với MaterialApp, Scaffold, AppBar:



4. StatelessWidget với StatefulWidget:

StatelessWidget là các widget có trạng thái không thể thay đổi sau khi chúng được build. Chúng chỉ đơn thuần nhận dữ liệu và hiển thị một cách thụ động. Nếu muốn render lại thì ta phải khởi tạo lại chúng.

Đặc điểm: Cấu trúc đơn giản có nghĩa là chúng được kết xuất nhanh chóng. Dễ hiểu và dễ sử dụng. Một khi đã xây xong, chúng luôn trông giống nhau.

```
class Widget extends StatelessWidget {  
  const Widget({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return const Text('Xin chào!');  
  }  
}
```

4. StatelessWidget với StatefulWidget:

Stateful widget là các widget có thể thay đổi trạng thái. Khác với stateless widget, chúng ta không cần thiết phải khởi tạo lại stateful widget để thay đổi chúng mà chỉ cần thay đổi state của chúng và Flutter sẽ gọi hàm build() để render lại UI cho phù hợp với state đó.

Đặc điểm: Tính tương tác cao, linh hoạt, mạnh mẽ

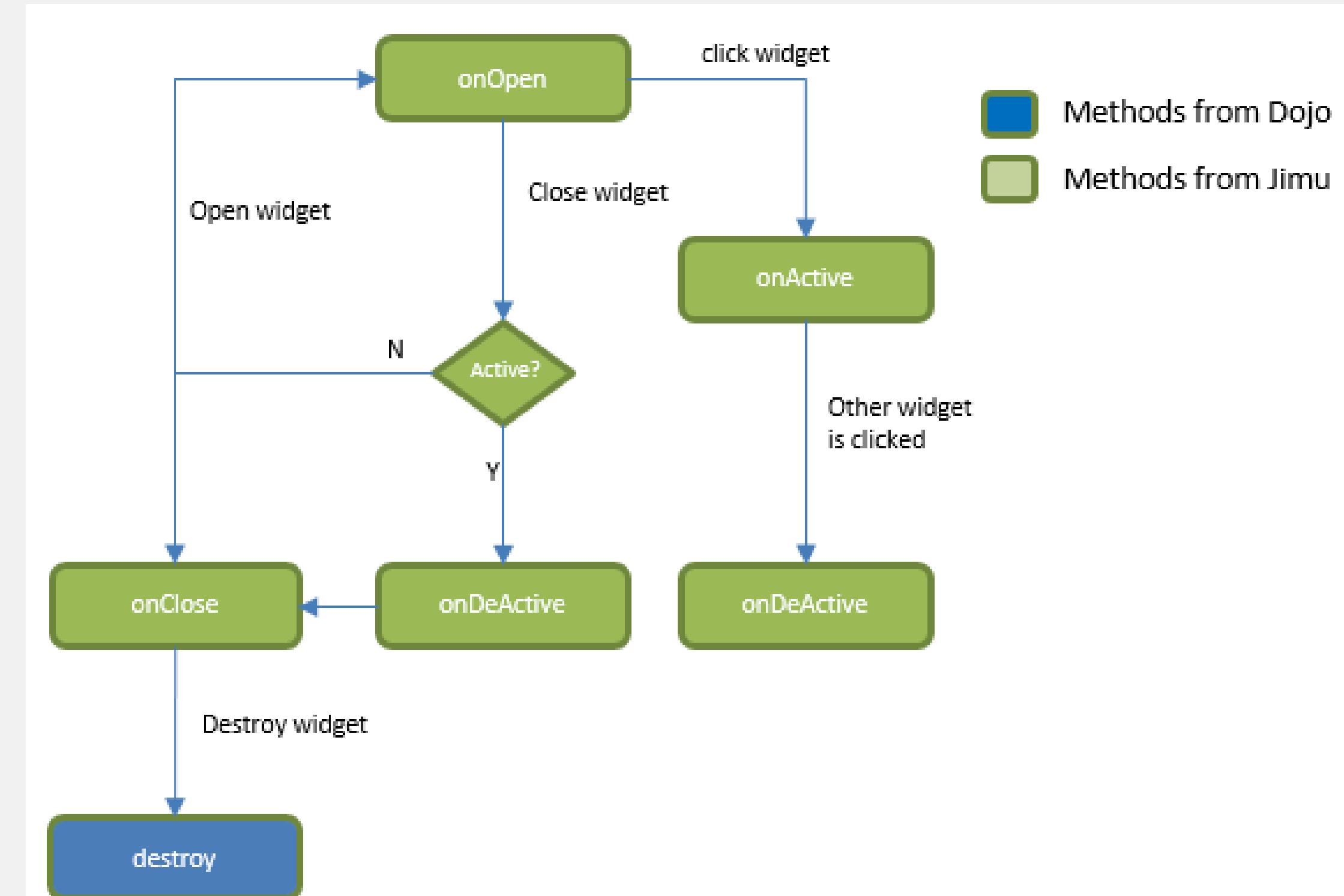
```
class MyHomePage extends StatefulWidget {  
  @override  
  MyHomePageState createState() => MyHomePageState(); }  
  
class MyHomePageState extends State<MyHomePage> {  
  int counter = 0;  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Center(  
        child: Text(' Data của hiện tại là: $counter')),  
      floatingActionButton: FloatingActionButton(  
        onPressed: () {  
          setState(() {  
            counter++;  
          });  
        },  
        child: Icon(Icons.add),  
      ),  
    );  
  }  
}
```

So sánh chi tiết về StatefulWidget và StatelessWidget:

| Tiêu chí | StatelessWidget | StatefulWidget |
|----------------------------|---|--|
| Tính thay đổi (mutable) | Không thay đổi – nội dung cố định. | Có thể thay đổi, nội dung thay đổi khi setState() |
| Lưu trữ trạng thái (state) | Không có trạng thái nội bộ. | Có trạng thái nội bộ được quản lý trong class State. |
| Hiệu năng | Nhẹ hơn, render nhanh hơn vì không cần theo dõi | Nặng hơn vì phải quản lý và cập nhật state. |
| Vòng đời | Đơn giản – chỉ có build() | Có vòng đời phức tạp: initState(), build(), |
| Dùng khi | Giao diện chỉ hiển thị dữ liệu tĩnh, không thay đổi | Khi có tương tác người dùng hoặc dữ liệu thay đổi |

5. Widget lifecycle và rebuild mechanism:

Widget lifecycle là quá trình mô tả các giai đoạn mà một widget trong Flutter trải qua từ khi được tạo ra, hiển thị, cập nhật, cho đến khi bị hủy bỏ khỏi widget tree.



Vòng đời của StatefulWidget và StatelessWidget:

StatelessWidget:

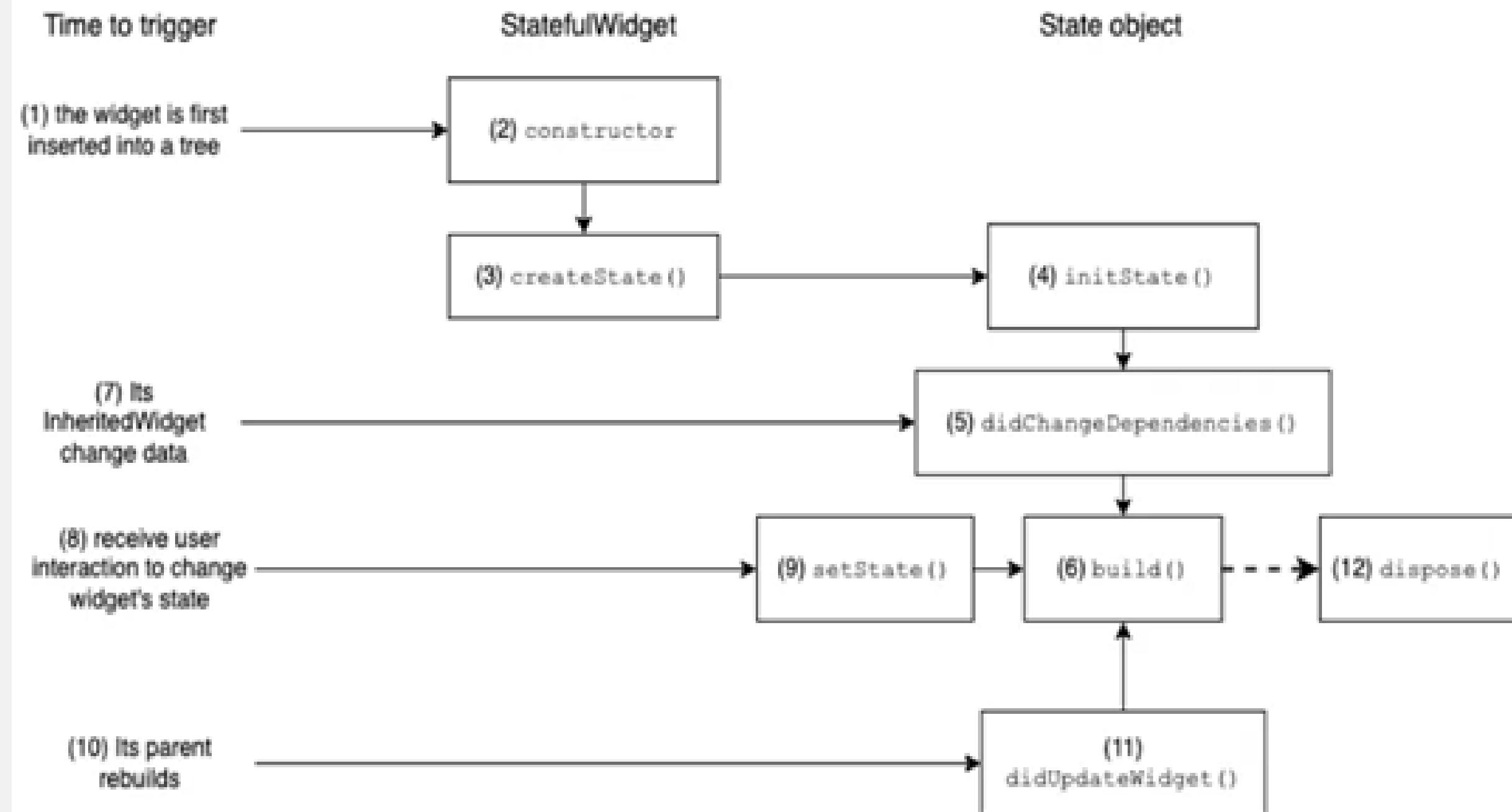
Vì không thay đổi nên vòng đời của stateless widget khá đơn giản, chỉ xoay quanh hàm **build()**. Vậy hàm **build()** sẽ được gọi khi nào?

- Lần đầu tiên widget được khởi tạo và chèn vào trong widget tree.
- Khi widget cha thay đổi cấu hình thì cũng sẽ kích hoạt hàm build() của các widget con và khiến chúng được render lại.

Vòng đời của StatefulWidget và StatelessWidget:

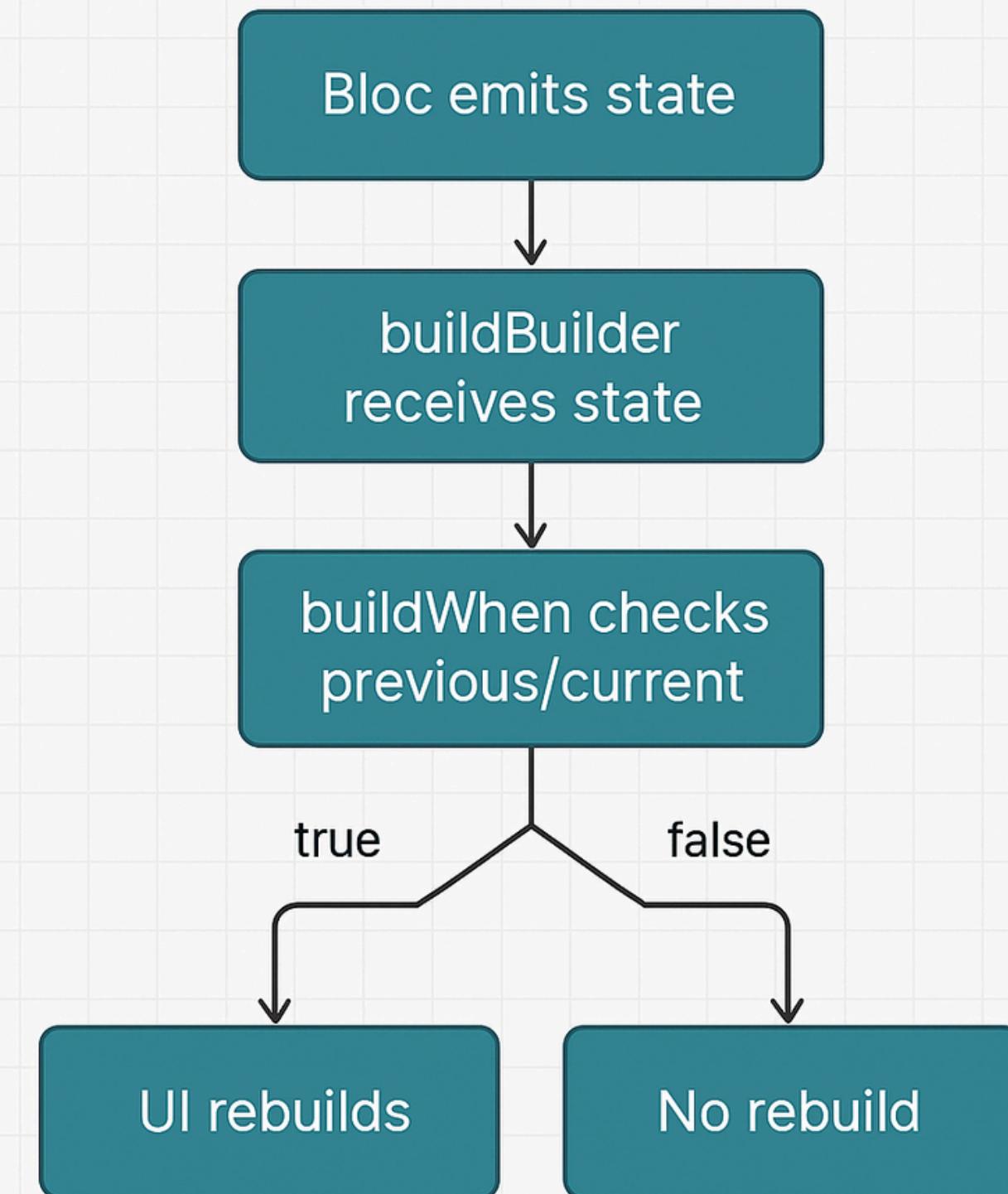
StatefulWidget:

Một StatefulWidget sẽ bao gồm hai phần là một widget và một State object. Bản thân đối tượng StatefulWidget là bất biến (immutable) và lưu trữ trạng thái có thể thay đổi (mutable) của chúng trong State object.



5. Rebuild mechanism:

Rebuild trong Flutter là quá trình cập nhật giao diện (UI) khi có thay đổi. Nó được kích hoạt bởi sự thay đổi về dữ liệu hoặc trạng thái ứng dụng. Khi rebuild, phương thức `build()` sẽ được gọi lại để tạo ra cây widget mới.



5. Rebuild mechanism:

Nguyên nhân dẫn đến Widget Rebuild:

- Thay đổi trạng thái state của StatefulWidget
- Widget cha được rebuild lại
- Thay đổi Key của widget

Cách phòng tránh rebuild không cần thiết:

- **Sử dụng const:** Dùng từ khóa const cho các widget không bao giờ thay đổi để Flutter không cần phải rebuild chúng nữa.
- **Tách nhỏ widget:** Tạo các widget nhỏ hơn và chỉ đặt **setState()** ở cấp thấp nhất có thể.
- **Dùng Key hợp lý:** Sử dụng ValueKey hoặc các Key khác để giúp Flutter nhận diện widget tốt hơn, từ đó tránh rebuild không cần thiết trong các danh sách (list) động.

Tài liệu tham khảo:

- [1] Google. "Flutter documentation": <https://docs.flutter.dev/>.
- [2] Google. "Flutter API documentation": <https://api.flutter.dev/>.