**TRƯỜNG ĐẠI HỌC BÁCH KHOA - ĐẠI HỌC ĐÀ NẴNG**

**KHOA ĐIỆN TỬ - VIỄN THÔNG**

****

**BÁO CÁO**

**LẬP TRÌNH ĐA NỀN TẢNG**

**Chủ đề: Flutter Architecture và Widget Tree**

|  |  |  |
| --- | --- | --- |
| **Sinh viên thực hiện:** |  |  |
| **Nguyễn Việt Hoàng** | **Lớp: 22KTMT1** | **MSSV: 106220217** |
| **Lê Minh Nhật** | **Lớp: 22DT2** | **MSSV: 106220066** |

|  |  |
| --- | --- |
| **Người hướng dẫn** | **: TS. Nguyễn Duy Nhật Viễn** |

**Đà Nẵng, 10/2025**

**THUYẾT MINH**

**BÁO CÁO**

**LẬP TRÌNH ĐA NỀN TẢNG**

**Flutter Architecture và Widget Tree**

**BẢNG PHÂN CÔNG 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: [VietHoang301/ltdntbttuan11\_12](https://github.com/VietHoang301/ltdntbttuan11_12)

**MỤC LỤC**

[**1.** **Flutter là gì?:** 3](#_Toc212431136)

[**2.** **Kiến trúc của Flutter:** 3](#_Toc212431137)

[ **Tầng 1: Application layer (Tầng ứng dụng):** 4](#_Toc212431138)

[ **Tầng 2: Framework layers (Tầng khung):** 4](#_Toc212431139)

[ **Tầng 3: Engine Layer (Tầng động cơ):** 5](#_Toc212431140)

[ **Tầng 4: Embedder Layer (Tầng nhúng):** 6](#_Toc212431141)

[**3.** **Demo Widget Tree với MaterialApp, Scaffold, AppBar:** 7](#_Toc212431142)

[**4.** **StatelessWidget với StatefulWidget:** 11](#_Toc212431143)

[**4.1.** **StatelessWidget:** 11](#_Toc212431144)

[**4.2.** **StatefulWidget:** 11](#_Toc212431145)

[**5.** **Widget lifecycle và rebuild mechanism:** 13](#_Toc212431146)

[**5.1.** **Widget lifecycle:** 13](#_Toc212431147)

[**5.2.** **Vòng đời của StatefulWidget và StatelessWidget:** 13](#_Toc212431148)

[**5.3.** **Rebuild mechanism:** 14](#_Toc212431149)

[**Tài liệu tham khảo:** 16](#_Toc212431150)

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.

**- Các đặc điểm chính của Flutter:**

**+** **Phát triển đa nền tảng:** Cho phép tạo ra ứng dụng cho iOS, Android, web, Windows, macOS và Linux chỉ từ một cơ sở mã duy nhất.

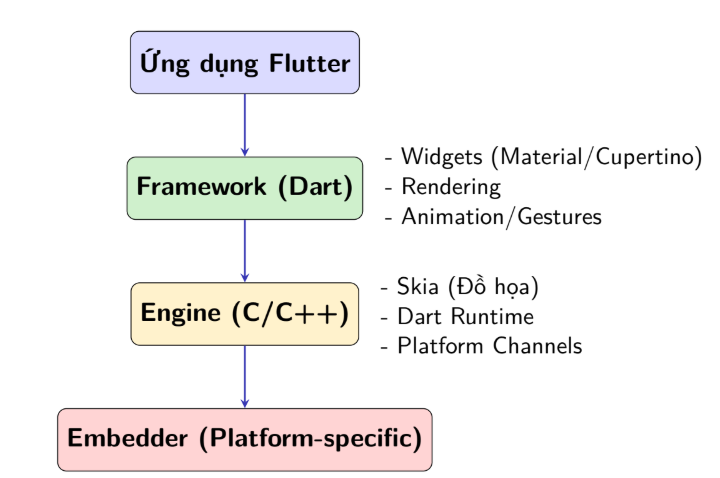
**+ Hiệu suất cao**: Biên dịch mã nguồn trực tiếp thành mã máy, mang lại hiệu suất gần giống với ứng dụng native (ứng dụng được viết riêng cho từng hệ điều hành).

**+ Hệ sinh thái mạnh:** Thư viện phong phú, tích hợp dễ dàng dịch vụ ngoài.

**+ Giao diện người dùng đẹp mắt:** Cung cấp khả năng tùy chỉnh hoàn toàn giao diện người dùng, với các widget có thể tái sử dụng để tạo ra giao diện đẹp và mượt mà.

1. **Kiến trúc của Flutter:**

- Flutter có 4 tầng chính: **Application → Framework → Engine → Embedder**.



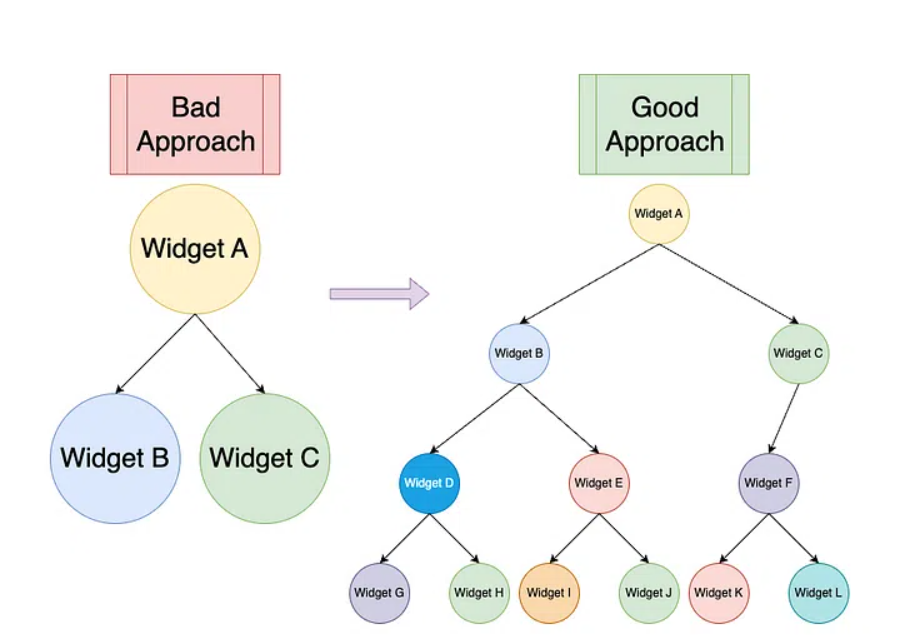
* **Tầng 1: Application layer (Tầng ứng dụng):**
* Đây là nơi toàn bộ mã nguồn và logic của ứng dụng được viết ra. Nó không chỉ đơn thuần là các widget, mà là một hệ thống hoàn chỉnh.
* **Nhiệm vụ chính:**
* **Giao diện người dùng (UI)**: Toàn bộ cây widget mà bạn xây dựng để định hình nên giao diện của ứng dụng, từ các màn hình lớn đến các nút bấm nhỏ nhất.
* **Logic nghiệp vụ (Business Logic)**: Các quy tắc, luồng dữ liệu và cách xử lý thông tin đặc thù của ứng dụng. (Ví dụ: logic xác thực người dùng, tính toán giỏ hàng, xử lý đơn hàng…).
* **Quản lý trạng thái (State Management)**: Cách bạn tổ chức và truyền dữ liệu qua các màn hình. Ở quy mô nhỏ, bạn có thể dùng StatefulWidget. Khi ứng dụng lớn hơn, bạn sẽ cần các giải pháp như Provider, BLoC, GetX, v.v. để quản lý trạng thái ứng dụng một cách hiệu quả.
* **Tương tác**: Bạn định nghĩa cách ứng dụng phản hồi lại hành động của người dùng (ví dụ: onPressed cho một nút bấm) tại tầng này.
* **Cách thức hoạt động**: Tập trung vào việc tạo ra cây widget (Widget Tree) để mô tả UI. Khi trạng thái ứng dụng thay đổi (ví dụ: người dùng nhấn nút) 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.
* **Tầng 2: Framework layers (Tầng khung):**
* Tầng này cung cấp tất cả mọi thứ cần thiết để xây dựng ứng dụng một cách nhanh chóng và hiệu qua từ bản thiết kế ở tầng trên. Tầng này được viết hoàn toàn bằng Dart.
* **Nhiệm vụ chính:**
* **Cung cấp Widgets**: Đây là cốt lõi của tầng này. Nó cung cấp một bộ sưu tập widget phong phú, từ những thứ cơ bản như Text, Container, Row, Column cho đến các widget phức tạp như ListView và Scaffold. Các bộ widget Material (cho Android) và Cupertino (cho iOS) giúp ứng dụng có giao diện quen thuộc trên từng nền tảng.
* **Xử lý Rendering:** Tầng này nhận lấy cây widget đã tạo ra và biến nó thành thứ có thể vẽ được. Nó quản lý một cây trung gian gọi là Element Tree, giúp so sánh phiên bản cũ và mới của UI một cách hiệu quả để xác định những gì cần được vẽ lại. Thực hiện theo một chuỗi các bước như sau:
* **Bước 1: Widget Tree**: Cây cấu hình bất biến mà bạn tạo ra trong mã.
* **Bước 2**: **Element Tree**: Một cây trung gian, lưu giữ trạng thái của UI và tham chiếu đến widget và đối tượng render. Đây là "bộ não" của việc cập nhật UI hiệu quả.
* **Bước 3**: **RenderObject Tree**: Cây này chịu trách nhiệm tính toán **bố cục** (kích thước, vị trí) và **vẽ (painting)** các widget lên màn hình.
* **Cung cấp API cấp cao**: Cung cấp các công cụ để xử lý hoạt ảnh (animation), cử chỉ (gestures) và vẽ tùy chỉnh (painting). Nó cũng chứa thư viện nền tảng (Foundation) với các lớp và hàm cơ bản mà mọi ứng dụng đều cần.
* **Cách thức hoạt động**: 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.
* **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.
* **Nhiệm vụ chính**:
* **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. Đây là lý do Flutter có thể tạo ra giao diện đẹp mắt, mượt mà và đồng nhất trên mọi nền tảng mà không cần dùng đến widget gốc của hệ điều hành Android hay iOS. Thay vào đó, nó mang theo một "toan vẽ" (canvas) của riêng mình và dùng Skia để **vẽ trực tiếp mọi thứ** lên đó, từ văn bản, hình ảnh, đến các hiệu ứng phức tạp. Điều này cho phép Flutter toàn quyền kiểm soát từng pixel, mang lại sự linh hoạt và hiệu suất đồ họa vượt trội.
* **Cung cấp Dart Runtime**: Cung cấp môi trường để mã Dart của bạn có thể chạy.

Trong lúc phát triển, nó dùng trình biên dịch **JIT (Just-In-Time)** để kích hoạt tính năng Hot Reload (cho phép nạp mã mới vào máy ảo đang chạy mà không cần khởi động lại toàn bộ ứng dụng). Khi phát hành, nó dùng trình biên dịch **AOT (Ahead-Of-Time)** để biên dịch mã Dart thành mã máy native. Điều này giúp ứng dụng khởi động nhanh và đạt được hiệu suất gần như tương đương với ứng dụng gốc.

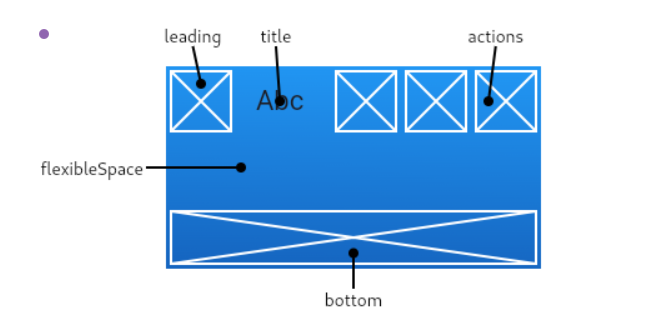
* **Platform Channels**: Đây là cây cầu giao tiếp giữa thế giới Dart và thế giới Native. Cung cấp cơ chế cho phép mã Dart giao tiếp với mã native (Java/Kotlin trên Android, Swift/Obj-C trên iOS) và ngược lại. Khi bạn cần sử dụng một API đặc thù của hệ điều hành (ví dụ: đọc thông tin pin, sử dụng cảm biến vân tay), bạn sẽ gửi một thông điệp qua Platform Channel. Tầng Engine sẽ nhận thông điệp này và chuyển tiếp đến mã native tương ứng ở tầng Embedder.
* **Quản lý cấp thấp**: Xử lý các tác vụ như quản lý bộ nhớ (Garbage Collection), quản lý luồng (threading), và giao tiếp mạng cấp thấp.
* **Cách thức hoạt động:** Tầng này chính là **động cơ hiệu suất cao** của Flutter, được viết bằng C++. Sử dụng thư viện đồ họa **Skia** để vẽ trực tiếp mọi pixel lên màn hình, đảm bảo giao diện nhất quán. Đồng thời, nó cung cấp môi trường để thực thi mã Dart (với Hot Reload qua JIT và hiệu năng cao qua AOT) và làm cầu nối giao tiếp với nền tảng native.
* **Tầng 4: Embedder Layer (Tầng nhúng):**
* Đây là tầng dành riêng cho từng nền tảng, hoạt động như một "bộ chuyển đổi" (adapter) để "nhúng" (embed) Engine của Flutter vào ứng dụng gốc của hệ điều hành. Mỗi nền tảng (Android, iOS, Windows, Web) có một Embedder riêng.
* **Nhiệm vụ chính**:
* **Khởi tạo và Lưu trữ**: Embedder là một ứng dụng native tối giản, có nhiệm vụ khởi động Engine Flutter. Trên Android, nó là một **Activity**. Trên iOS, nó là một **UIViewController**. Nhiệm vụ của nó là khởi tạo **Engine Flutter** và quản lý vòng đời của nó.
* **Điều phối Vòng lặp Sự kiện (Event Loop)**: Tích hợp và điều phối vòng lặp sự kiện của Flutter với vòng lặp sự kiện của hệ điều hành. Khi có một sự kiện từ hệ điều hành (như chạm màn hình, một phím được nhấn), Embedder sẽ nhận nó và chuyển tiếp đến Engine.
* **Cung cấp "Bề mặt" để vẽ**: Yêu cầu hệ điều hành cấp cho một "khung tranh" (một Surface trên Android hoặc CALayer trên iOS) để Skia có thể vẽ UI lên đó.
* **Triển khai các Plugin Native**: Là nơi chứa mã native của các plugin. Khi mã Dart của bạn gọi một hàm từ plugin, thông điệp sẽ đi qua Platform Channel đến Engine và Engine sẽ chuyển nó đến mã native tương ứng trong Embedder để thực thi.
* **Cách thức hoạt động:** 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.

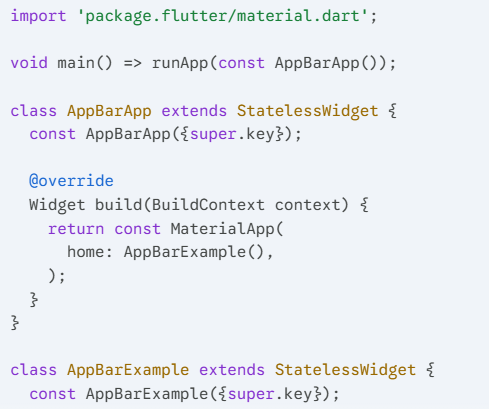
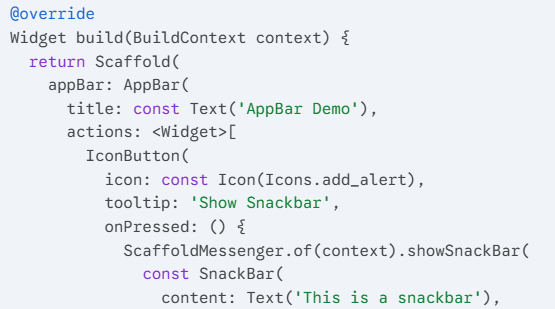
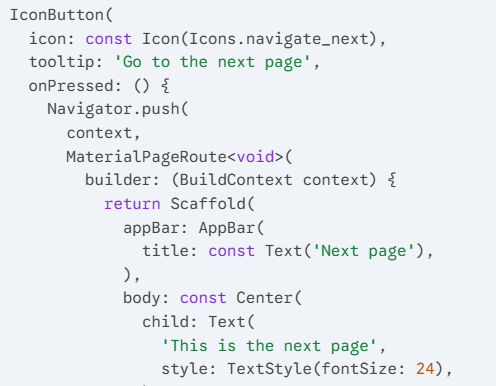
1. **Demo Widget Tree với MaterialApp, Scaffold, AppBar:**

* **Widget Tree** là cấu trúc cơ bản, nền tảng của mọi giao diện người dùng trong Flutter. Nó mô tả cách các widget được lồng vào nhau, từ widget gốc của ứng dụng cho đến từng nút bấm và dòng văn bản nhỏ nhất, để tạo nên bố cục cuối cùng mà người dùng nhìn thấy.
* **Cách một widget tree làm việc được mô tả theo 2 cách sau:**

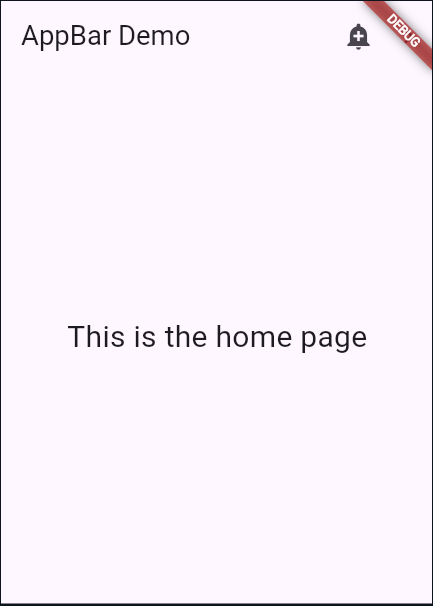
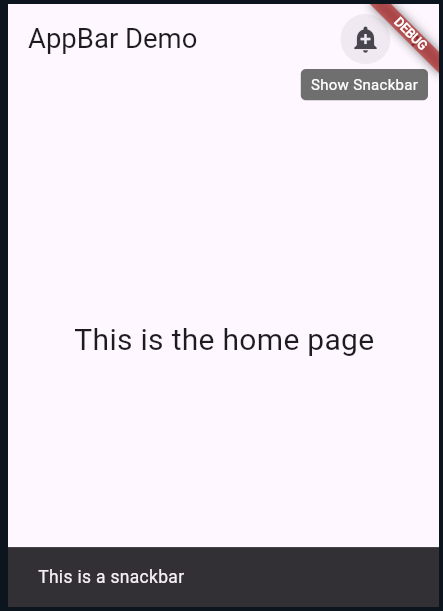
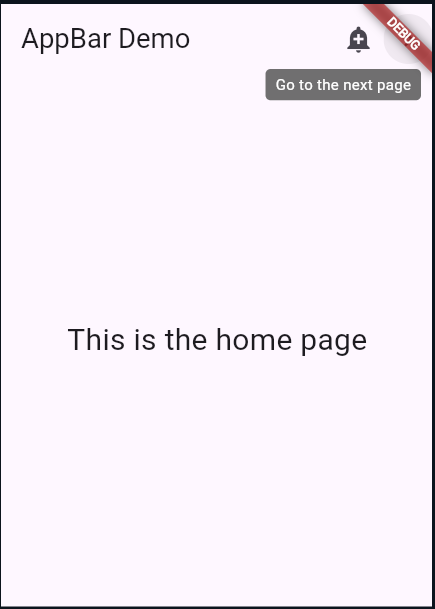
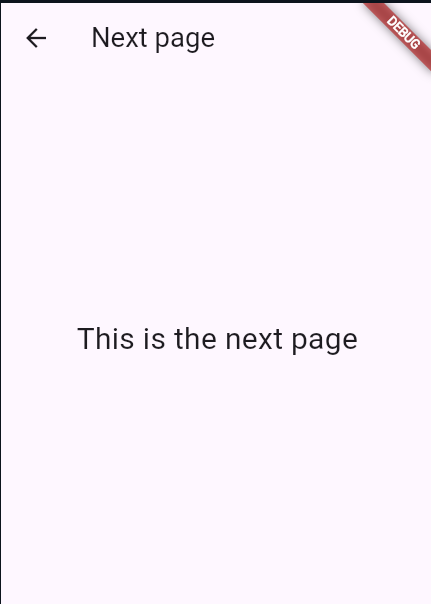
****

* 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. Tất nhiên điều này không phải một ý tưởng tốt để chúng ta scalable ứng dụng.
* **MaterialApp**: Widget gốc của ứng dụng. Nó cung cấp "vỏ" Material Design, bao gồm điều hướng (navigation), chủ đề (theme), và các thiết lập chung**.**
* **Scaffold**: Widget "giàn giáo" cho một màn hình cụ thể. Nó 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.
* **Appbar (Thanh ứng dụng)** thường được sử dụng trong thuộc tính Scaffold.appBar , thuộc tính này đặt thanh ứng dụng dưới dạng một tiện ích có chiều cao cố định ở đầu màn hình.
* **AppBar** hiển thị các tiện ích thanh công cụ, bao gồm leading, title và actions, nằm phía trên phần dưới cùng (nếu có). Phần dưới cùng thường được sử dụng cho TabBar . Nếu một tiện ích flexibleSpace được chỉ định, nó sẽ được xếp chồng lên phía sau thanh công cụ và tiện ích phía dưới cùng. Sơ đồ sau đây cho thấy vị trí xuất hiện của từng ô này trên thanh công cụ khi ngôn ngữ viết là từ trái sang phải:

****

****  



1. **StatelessWidget với StatefulWidget:**

## **StatelessWidget:**

* **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!');

  }

}

## **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: Phản hồi hành động của người dùng và thay đổi dữ liệu. Linh hoạt: Thích ứng với nhiều tình huống khác nhau. Mạnh mẽ: Xây dựng giao diện phức tạp và năng động.

import 'package:flutter/material.dart';

void main() {

  runApp(MyApp()); }

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      home: MyHomePage(),

    ); } }

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),

      ), ); }

}

* 1. **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() được gọi. |
| 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 state. | Nặng hơn vì phải quản lý và cập nhật state. |
| Vòng đời | Đơn giản – chỉ có build() | Có vòng đời phức tạp: initState(), build(), setState(), dispose(), ... |
| Dùng khi | Giao diện chỉ hiển thị dữ liệu tĩnh, không thay đổi theo thời gian. | Khi có tương tác người dùng hoặc dữ liệu thay đổi liên tục. |

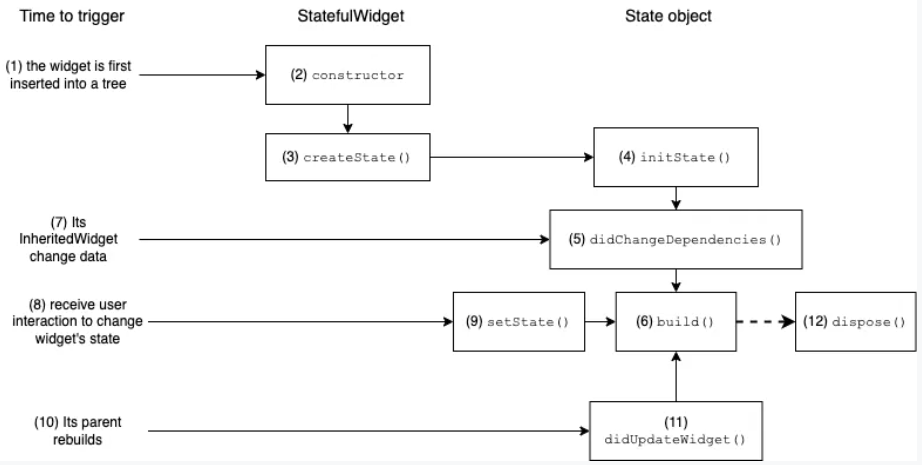
1. **Widget lifecycle và rebuild mechanism:**

## **Widget lifecycle:**

* 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 **buid()** 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.
* **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**.
* Trên hình chia ra làm 3 phần là:
* **Time to trigger:** thời điểm kích hoạt
* **StatefulWidget:** phần widget mà mình nhắc đến của StatefulWidget
* **State object:** phần trạng thái có thể thay đổi được của StatefulWidget
* **Ý nghĩa của từng con số trên hình:**

(1): Thời điểm đầu tiên widget được khởi tạo và chèn vào trong widget tree.

(2) và (3): Sau khi gọi constructor, Flutter sẽ gọi hàm createState() để tạo ra đối tượng của State cho widget. Kể từ đó về sau, Flutter đa số chỉ làm việc State object mà thôi (vì đối tượng của StatefulWidget là bất biến, như đã đề cập ở trên).

(4): Trên State object, sau khi gọi hàm createState() thì hàm initState() sẽ được gọi. Và hàm này chỉ được gọi chỉ một lần duy nhất cho mỗi 1 đối tượng của State.

(5): Phương thức didChangeDependencies() được gọi ngay sau initState lần đầu tiên.

(6): Hàm build() được gọi để render UI. Vậy là hoàn thành các bước kể từ lần đầu tiên widget được khởi tạo.

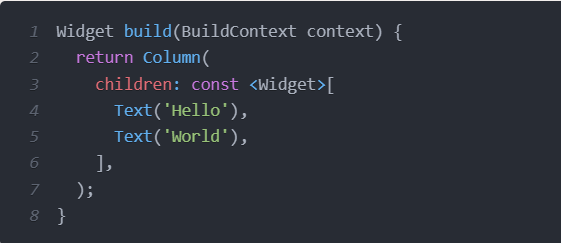
(7): Trường hợp InheritedWidget mà widget có dữ liệu phụ thuộc vào thay đổi thì sẽ kích hoạt hàm didChangeDependencies(), sau đó Flutter sẽ gọi build() để render lại UI.

(8) và (9): Giả sử như user tác động lên UI để thay đổi state của widget (ví dụ như like / unlike) thì khi đó việc của các developer sẽ là gọi hàm setState() để thay đổi trạng thái của object. Việc này sẽ giúp cho framework được thông báo về sự thay đổi này và sẽ gọi hàm build() để render lại UI tương ứng.

(10) và (11): Khi widget cha của nó build lại và yêu cầu các widget con build cập nhật lại, Flutter sẽ thực hiện việc gọi hàm didUpdateWidget() . Sau đó, hàm build() cũng sẽ được gọi.

(12): Hàm dispose() được gọi khi state object bị xoá khỏi widget tree vĩnh viễn.

* 1. **Rebuild mechanism:**
* **Rebuild widget** trong Flutter là quá trình cập nhật lại giao diện người dùng (UI) dựa trên những thay đổi về dữ liệu hoặc trạng thái của ứng dụng. Khi một widget được "rebuild", phương thức **build()** của widget đó sẽ được gọi lại để tạo ra một cây widget mới với các giá trị được cập nhật.
* **Nguyên nhân dẫn đến Widget Rebuild:**
* **Thay đổi trạng thái state của StatefulWidget**: Đây là nguyên nhân phổ biến nhất. Khi bạn gọi phương **thức setState()** bên trong một StatefulWidget, Flutter sẽ đánh dấu widget đó là (**dirty**) và lên lịch để gọi lại phương thức **build()** của nó ở khung hình tiếp theo.
* **Widget cha được reubild lại:** Khi một widget cha được rebuild, tất cả các widget con của nó cũng sẽ được rebuild theo một cách mặc định.
* **Thay đổi Key của widget:** Vài trường hợp, khi một widget được tạo lại với một Key khác so với lần trước, Flutter sẽ coi đó là một widget hoàn toàn mới và rebuild nó. Điều này thường được sử dụng để ép buộc một widget được tạo lại hoàn toàn.
* **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. Điều này giúp giới hạn phạm vi rebuild chỉ trong một phần nhỏ của giao diện, thay vì toàn bộ màn hình.
* **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/.