1. **AnimatedWidget:**

"AnimatedWidget" là một lớp trừu tượng trong Flutter, được sử dụng để xây dựng các widget tương tác hoạt hình. Nó là một phần của Flutter Animation Framework, cùng với các lớp như "AnimationController", "Tween", "CurvedAnimation", và nhiều lớp khác để quản lý và tạo các hiệu ứng hoạt hình trên giao diện người dùng của bạn.

Khi bạn tạo một "AnimatedWidget", bạn cần kế thừa từ lớp này và triển khai phương thức "build" của nó. Lớp con sẽ chứa một Animation (giá trị hoạt hình) và khi giá trị hoạt hình thay đổi, Flutter sẽ tự động gọi phương thức "build" của lớp con để cập nhật giao diện người dùng với các giá trị mới. Điều này giúp bạn xây dựng giao diện đáp ứng các thay đổi trong giá trị hoạt hình một cách hiệu quả và trơn tru..

AnimatedWidget được sử dụng phổ biến nhất với các đối tượng Animation, là Listenable, nhưng nó cũng có thể được sử dụng với bất kỳ Listenable nào, bao gồm cả ChangeNotifier và ValueNotifier.

AnimatedWidget hữu ích nhất cho các widget không trạng thái. Để sử dụng AnimatedWidget, hãy phân lớp nó và triển khai chức năng xây dựng.

Dưới đây là một ví dụ về cách triển khai một "AnimatedWidget" trong Flutter:

import 'dart:math' as math;

import 'package:flutter/material.dart';

/// Flutter code sample for [AnimatedWidget].

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

class AnimatedWidgetExampleApp extends StatelessWidget {

const AnimatedWidgetExampleApp({super.key});

@override

Widget build(BuildContext context) {

return const MaterialApp(

home: AnimatedWidgetExample(),

);

}

}

class SpinningContainer extends AnimatedWidget {

const SpinningContainer({

super.key,

required AnimationController controller,

}) : super(listenable: controller);

Animation<double> get \_progress => listenable as Animation<double>;

@override

Widget build(BuildContext context) {

return Transform.rotate(

angle: \_progress.value \* 2.0 \* math.pi,

child: Container(width: 200.0, height: 200.0, color: Colors.green),

);

}

}

class AnimatedWidgetExample extends StatefulWidget {

const AnimatedWidgetExample({super.key});

@override

State<AnimatedWidgetExample> createState() => \_AnimatedWidgetExampleState();

}

/// [AnimationController]s can be created with `vsync: this` because of

/// [TickerProviderStateMixin].

class \_AnimatedWidgetExampleState extends State<AnimatedWidgetExample>

with TickerProviderStateMixin {

late final AnimationController \_controller = AnimationController(

duration: const Duration(seconds: 10),

vsync: this,

)..repeat();

@override

void dispose() {

\_controller.dispose();

super.dispose();

}

@override

Widget build(BuildContext context) {

return SpinningContainer(controller: \_controller);

}

}

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, Trang web

Mô tả được tạo tự động

1. **AnimatedBuilder**
2. **Giới thiệu về AnimatedBuilder trong Flutter:**

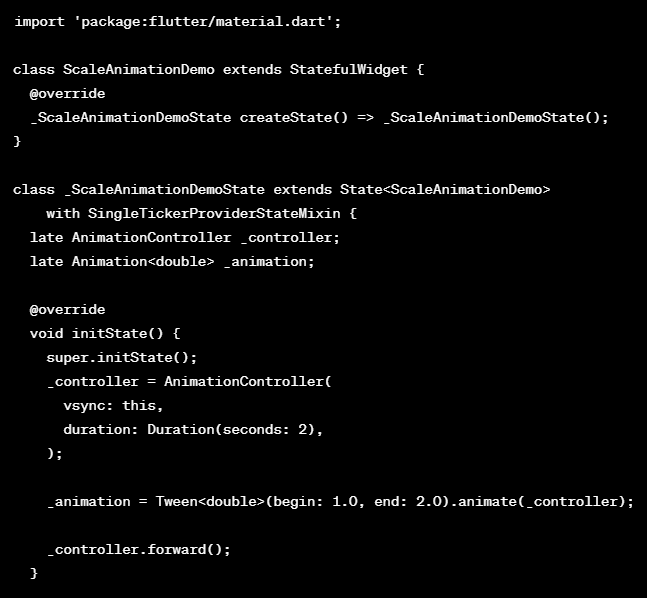
Flutter là một framework phát triển ứng dụng di động và web đa nền tảng. Nó cho phép bạn xây dựng các ứng dụng đẹp, mượt mà và tương tác bằng cách sử dụng ngôn ngữ lập trình Dart. Để thêm hiệu ứng hoạt hình vào các thành phần giao diện người dùng, Flutter cung cấp nhiều widget và API hỗ trợ. Trong đó, AnimatedBuilder là một widget mạnh mẽ giúp bạn tạo các hiệu ứng hoạt hình phức tạp một cách dễ dàng.

1. **Cách hoạt động của AnimatedBuilder:**

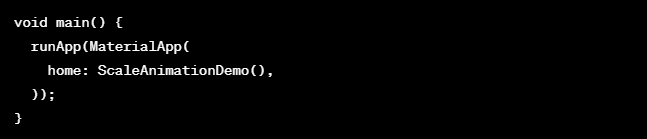
Khi bạn muốn thực hiện một hiệu ứng hoạt hình trong Flutter, bạn thường sẽ sử dụng một đối tượng Animation để quản lý giá trị của hoạt hình. AnimatedBuilder kết nối một đối tượng Animation với một hàm xây dựng UI (builder function). Khi giá trị của Animation thay đổi, AnimatedBuilder sẽ tự động gọi lại builder function để cập nhật giao diện người dùng.

1. **Ví dụ sử dụng AnimatedBuilder trong Flutter:**

Dưới đây là một ví dụ cụ thể về cách sử dụng AnimatedBuilder để thực hiện hiệu ứng hoạt hình đơn giản:







**Giải thích ví dụ:**

* Trong ví dụ này, chúng ta tạo một ‘ScaleAnimationDemo’ kế thừa từ ‘StatefulWidget’.
* Trong ‘initState()’, chúng ta khởi tạo một ‘AnimationController’ để quản lý hiệu ứng hoạt hình trong vòng 2 giây.
* Chúng ta sử dụng ‘Tween’ để xác định phạm vi của giá trị mở rộng từ 1.0 đến 2.0 cho ‘\_animation’.
* ‘AnimatedBuilder’ được sử dụng để liên kết ‘\_animation’ với ‘Transform.scale’ widget. Khi giá trị ‘\_animation.value’ thay đổi, ‘AnimatedBuilder’ sẽ tự động gọi lại builder function và widget ‘Transform.scale’ sẽ được cập nhật với giá trị scale mới, tạo ra hiệu ứng hoạt hình co giãn.
* Khi widget được tạo, ‘\_controller.forward()’ được gọi để bắt đầu hiệu ứng hoạt hình.

**Kết luận:**

AnimatedBuilder là một công cụ mạnh mẽ trong Flutter để thêm các hiệu ứng hoạt hình mượt mà vào giao diện người dùng. Nó giúp tách biệt logic xây dựng UI và logic xử lý hiệu ứng hoạt hình, giúp mã nguồn dễ đọc và bảo trì. Ví dụ trên minh họa cách sử dụng AnimatedBuilder để thực hiện hiệu ứng hoạt hình co giãn cho một widget trong Flutter.

1. **AnimationController**

Một bộ điều khiển cho một hình ảnh động.

Lớp này cho phép bạn thực hiện các tác vụ như:

Phát hoạt ảnh tiến hoặc lùi hoặc dừng hoạt ảnh.

Đặt hoạt ảnh thành một giá trị cụ thể .

Xác định các giá trị UpperBound và LowerBound của hoạt ảnh.

Tạo hiệu ứng hoạt hình bằng cách sử dụng mô phỏng vật lý.

Theo mặc định, AnimationController tạo tuyến tính các giá trị nằm trong khoảng từ 0,0 đến 1,0 trong một khoảng thời gian nhất định. Bộ điều khiển hoạt hình tạo ra một giá trị mới bất cứ khi nào thiết bị chạy ứng dụng của bạn sẵn sàng hiển thị một khung hình mới (thông thường, tốc độ này là khoảng 60 giá trị mỗi giây).

Ticker providers

Một AnimationController cần một TickerProvider , được cấu hình bằng cách sử dụng vsyncđối số trên hàm tạo.

Giao diện TickerProvider mô tả một nhà máy cho các đối tượng Ticker . Ticker là một đối tượng biết cách đăng ký chính nó với SchedulerBinding và kích hoạt một cuộc gọi lại mỗi khung. Lớp AnimationController sử dụng Ticker để chuyển qua hoạt ảnh mà nó điều khiển.

Nếu một AnimationController đang được tạo từ một State , thì State có thể sử dụng các lớp TickerProviderStateMixin và SingleTickerProviderStateMixin để triển khai giao diện TickerProvider . Lớp TickerProviderStateMixin luôn hoạt động cho mục đích này; SingleTickerProviderStateMixin hiệu quả hơn một chút trong trường hợp lớp chỉ cần một Ticker (ví dụ: nếu lớp chỉ tạo một AnimationController duy nhất trong toàn bộ vòng đời của nó).

Khung kiểm tra tiện ích Đối tượng WidgetTester có thể được sử dụng làm nhà cung cấp mã đánh dấu trong bối cảnh kiểm tra. Trong các bối cảnh khác, bạn sẽ phải chuyển một TickerProvider từ cấp cao hơn (ví dụ: gián tiếp từ một Trạng thái trộn lẫn trong TickerProviderStateMixin ) hoặc tạo một lớp con TickerProvider tùy chỉnh .

Sử dụng Futures với AnimationController

Các phương thức bắt đầu hoạt ảnh trả về một đối tượng TickerFuture hoàn thành khi hoạt ảnh hoàn thành thành công và không bao giờ đưa ra lỗi; nếu hoạt ảnh bị hủy, tương lai sẽ không bao giờ hoàn thành. Đối tượng này cũng có thuộc tính TickerFuture.orCancel trả về một tương lai hoàn thành khi hoạt ảnh hoàn tất thành công và hoàn thành với lỗi khi hoạt ảnh bị hủy bỏ.

Điều này có thể được sử dụng để viết mã như fadeOutAndUpdateStatephương pháp dưới đây.

**class** **Foo** **extends** **StatefulWidget** {

**const** Foo({ **super**.key, **required** **this**.duration });

**final** Duration duration;

**@override**

State<Foo> createState() => \_FooState();

}

**class** **\_FooState** **extends** **State**<**Foo**> **with** **SingleTickerProviderStateMixin** {

**late** AnimationController \_controller;

**@override**

**void** initState() {

**super**.initState();

\_controller = AnimationController(

vsync: **this**, ***// the SingleTickerProviderStateMixin***

duration: widget.duration,

);

}

**@override**

**void** didUpdateWidget(Foo oldWidget) {

**super**.didUpdateWidget(oldWidget);

\_controller.duration = widget.duration;

}

**@override**

**void** dispose() {

\_controller.dispose();

**super**.dispose();

}

**@override**

Widget build(BuildContext context) {

**return** Container(); ***// ...***

}

}

Phương thức sau (đối với lớp con Trạng thái ) điều khiển hai bộ điều khiển hoạt ảnh bằng cách sử dụng cú pháp không đồng bộ của Dart để chờ các đối tượng Future :

Future<**void**> fadeOutAndUpdateState() **async** {

**try** {

**await** fadeAnimationController.forward().orCancel;

**await** sizeAnimationController.forward().orCancel;

setState(() {

dismissed = **true**;

});

} **on** TickerCanceled {

***// the animation got canceled, probably because we were disposed***

}

}

1. **CuvedAnimation**

Hoạt ảnh cong xác định tiến trình của hoạt ảnh là một đường cong phi tuyến tính.

*animation = CurvedAnimation(parent: controller, curve: Curves.easeIn);*

**Lưu ý:** Lớp Curves định nghĩa nhiều đường cong thường được sử dụng hoặc bạn có thể tạo đường cong của riêng mình. Ví dụ:

*import 'dart:math';*

*class ShakeCurve extends Curve {*

*@override*

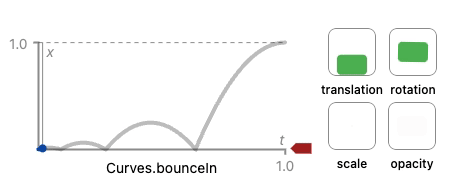
*double transform(double t) => sin(t \* pi \* 2);*

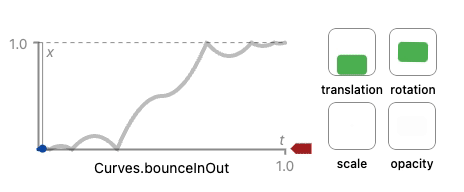
*}*

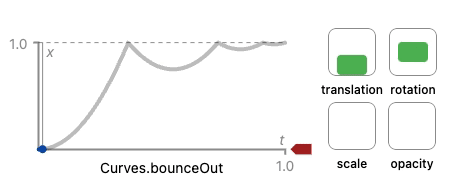
CurvedAnimation và AnimationController (được mô tả trong phần tiếp theo) đều thuộc loại Animation<double>, vì vậy bạn có thể chuyển chúng thay thế cho nhau. CurvedAnimation bao bọc đối tượng mà nó đang sửa đổi - bạn không phân lớp AnimationController để triển khai một đường cong.

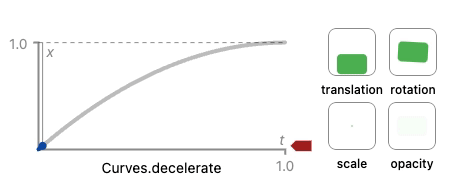
**Ví dụ về Curved Animation:**

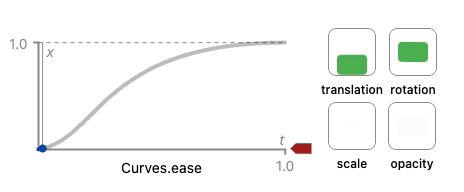
Một số các đường cong hoạt ảnh phổ biến

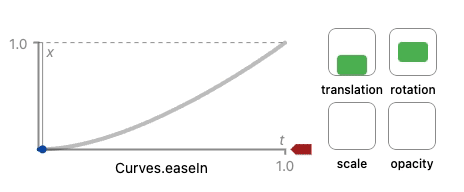


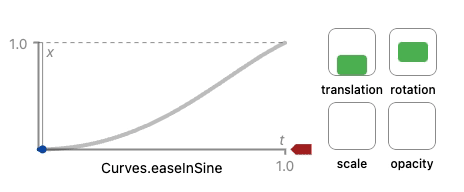


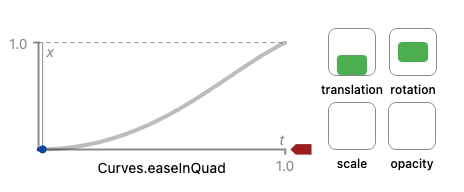


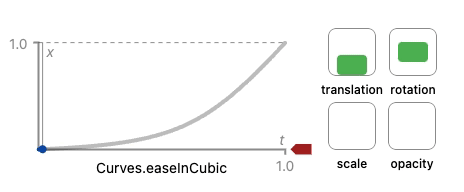


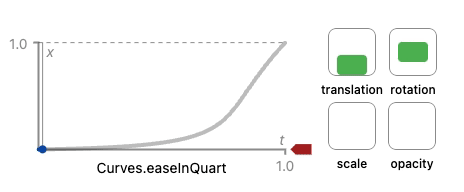


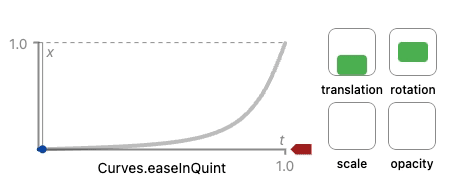


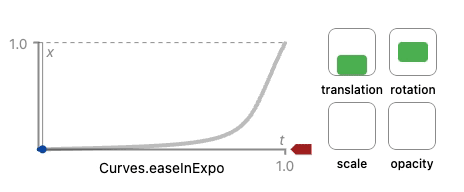


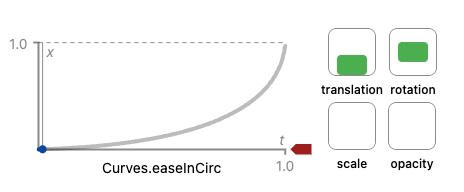


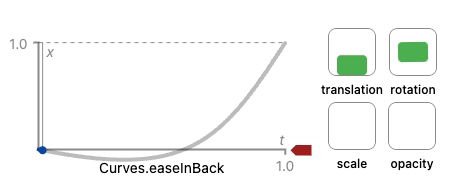


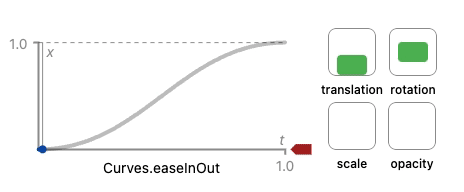


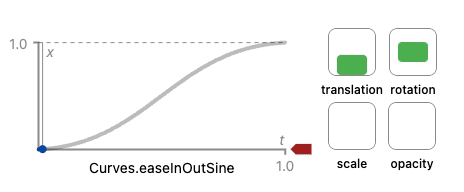


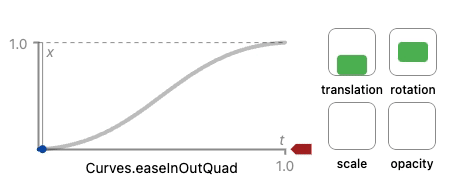


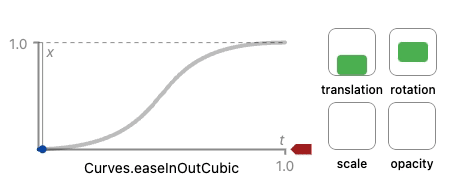


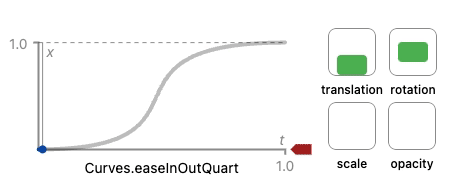


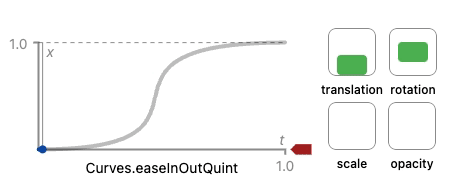


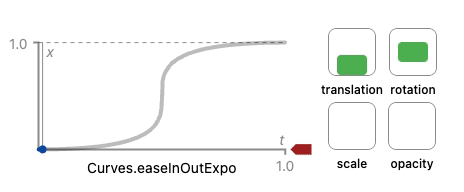












1. **Hero**

Những gì bạn sẽ học "Hero" đề cập đến tiện ích (widget) bay giữa các màn hình. Tạo một hiệu ứng "hero" bằng cách sử dụng tiện ích (widget) "Hero" của Flutter. Đưa "hero" bay từ một màn hình sang màn hình khác. Thực hiện hiệu ứng biến đổi hình dạng của "hero" từ hình tròn thành hình chữ nhật trong quá trình bay nó từ một màn hình sang màn hình khác. Tiện ích (widget) "Hero" trong Flutter thực hiện một loại hiệu ứng thường được gọi là chuyển tiếp phần tử chia sẻ hoặc hiệu ứng phần tử chia sẻ.

Bạn có thể đã thấy nhiều hiệu ứng hero (nhân vật anh hùng) nhiều lần. Ví dụ, một màn hình hiển thị danh sách các hình thu nhỏ đại diện cho các mục hàng bán. Chọn một mục hàng sẽ làm nó bay đến một màn hình mới, chứa nhiều chi tiết hơn và một nút "Mua". Việc bay một hình ảnh từ một màn hình sang màn hình khác được gọi là hiệu ứng hero trong Flutter, mặc dù cùng một chuyển động này đôi khi còn được gọi là chuyển tiếp phần tử chia sẻ.

Bạn có thể tạo hiệu ứng này trong Flutter bằng cách sử dụng các tiện ích (widget) "Hero". Khi hiệu ứng hero bay từ một route nguồn đến một route đích, route đích (trừ hero) sẽ xuất hiện từ từ. Thông thường, hero là các phần nhỏ trong giao diện người dùng, chẳng hạn như hình ảnh, mà cả hai route đều có chung. Từ quan điểm của người dùng, hero "bay" giữa các route. Hướng dẫn này sẽ chỉ bạn cách tạo các hiệu ứng hero sau đây:

**Hiệu ứng hero tiêu chuẩn**

Một hiệu ứng hero tiêu chuẩn bay hero từ một route sang route mới, thường hạ cánh ở một vị trí khác và với kích thước khác nhau.

Video dưới đây (được ghi lại ở tốc độ chậm) cho thấy một ví dụ điển hình. Khi chạm vào các bộ bơi ở trung tâm của route, chúng bay đến góc trên bên trái của một route mới màu xanh, với kích thước nhỏ hơn. Chạm vào các bộ bơi trong route màu xanh (hoặc sử dụng cử chỉ quay lại route trước trên thiết bị), chúng sẽ bay trở lại route ban đầu.

Trong hiệu ứng hero theo kiểu radial, khi hero bay giữa các route, hình dạng của nó xuất hiện thay đổi từ hình tròn thành hình chữ nhật.

**Cấu trúc cơ bản của một hiệu ứng hero:**

Hiệu ứng hero được thực hiện bằng cách sử dụng hai tiện ích (widget) "Hero": một tiện ích (widget) mô tả tiện ích (widget) trong route nguồn và một tiện ích (widget) mô tả tiện ích (widget) trong route đích. Từ quan điểm của người dùng, hero xuất hiện như là một phần được chia sẻ, và chỉ có lập trình viên cần hiểu chi tiết cài đặt này. Mã hiệu ứng hero có cấu trúc như sau:

Xác định một tiện ích (widget) "Hero" bắt đầu, được gọi là "source hero". Tiện ích (widget) "hero" này chỉ định biểu diễn đồ họa của nó (thường là một hình ảnh), và một thẻ xác định, và nó đang tồn tại trong cây tiện ích (widget tree) hiện tại như được định nghĩa bởi route nguồn.

Xác định một tiện ích (widget) "Hero" kết thúc, được gọi là "destination hero". Tiện ích (widget) "hero" này cũng chỉ định biểu diễn đồ họa của nó, và cùng một thẻ với source hero. Việc hai tiện ích (widget) "hero" này được tạo với cùng một thẻ là rất quan trọng, thường là một đối tượng đại diện cho dữ liệu cơ bản. Để có kết quả tốt nhất, các tiện ích (widget) "hero" nên có cây tiện ích (widget tree) hầu như giống nhau.

Tạo một route chứa destination hero. Route đích xác định cây tiện ích (widget tree) tồn tại vào cuối của hiệu ứng.

Kích hoạt hiệu ứng bằng cách đẩy route đích lên ngăn xếp của Navigator. Các hoạt động đẩy và lấy ra (pop) của Navigator kích hoạt hiệu ứng hero cho mỗi cặp hero có thẻ phù hợp trong các route nguồn và đích.

Flutter tính toán tween (giữa hai giá trị) để thực hiện hiệu ứng di chuyển kích thước và vị trí của Hero từ điểm bắt đầu đến điểm kết thúc, và thực hiện hiệu ứng trong một lớp overlay.

Phần tiếp theo sẽ mô tả quá trình của Flutter chi tiết hơn.

**Đằng sau cảnh**

Dưới đây là mô tả về cách Flutter thực hiện quá trình chuyển tiếp từ một route sang route khác:

A screenshot of a computer

Description automatically generated

Trước khi chuyển tiếp diễn ra, hero nguồn đợi trong cây tiện ích (widget tree) của route nguồn. Route đích chưa tồn tại và lớp overlay trống rỗng.

**A diagram of a flight

Description automatically generated**

Khi đẩy một route vào Navigator, hiệu ứng chuyển tiếp sẽ được kích hoạt. Tại thời điểm t=0.0, Flutter thực hiện các bước sau đây:

* Tính toán đường đi của hero đích, ẩn khỏi màn hình, sử dụng chuyển động cong như mô tả trong tiêu chuẩn Material motion. Lúc này, Flutter biết được hero sẽ đến đâu.
* Đặt hero đích vào lớp overlay, ở cùng vị trí và cùng kích thước với hero nguồn. Thêm một hero vào overlay thay đổi thứ tự Z của nó để hiển thị trên tất cả các route.
* Di chuyển hero nguồn ra khỏi màn hình, ẩn khỏi tầm nhìn.

A diagram of a different type of hero

Description automatically generated

Trong quá trình hero bay, các giới hạn hình chữ nhật của nó được thực hiện hiệu ứng bằng cách sử dụng Tween<Rect>, được xác định trong thuộc tính createRectTween của Hero. Mặc định, Flutter sử dụng một thể hiện của MaterialRectArcTween, cho phép các góc đối diện của hình chữ nhật được di chuyển theo một đường cong. (Xem hiệu ứng hero kiểu radial để xem một ví dụ sử dụng Tween animation khác.)

A screenshot of a computer screen

Description automatically generated

Khi quá trình bay hoàn thành:

* Flutter di chuyển tiện ích (widget) "hero" từ lớp overlay vào route đích. Lúc này, lớp overlay sẽ trở nên trống rỗng.
* Tiện ích (widget) "hero" đích hiển thị ở vị trí cuối cùng của nó trong route đích.
* Tiện ích (widget) "hero" nguồn được khôi phục lại vị trí của nó trong route nguồn.
* Khi thực hiện lệnh "pop" để quay lại route trước, quá trình tương tự được thực hiện, hiệu ứng bay hero sẽ diễn ra ngược trở lại với kích thước và vị trí ban đầu trong route nguồn.

**Các lớp cần thiết**

Các ví dụ trong hướng dẫn này sử dụng các lớp sau để thực hiện hiệu ứng hero:

Hero: Tiện ích (widget) bay từ route nguồn đến route đích. Bạn cần xác định một Hero cho route nguồn và một Hero khác cho route đích, và gán cho mỗi Hero cùng một thẻ (tag). Flutter sẽ thực hiện hiệu ứng bay cho các cặp Hero có thẻ trùng khớp.

InkWell: Xác định hành động xảy ra khi chạm vào Hero. Phương thức onTap() của InkWell xây dựng route mới và đẩy nó vào ngăn xếp Navigator.

Navigator: Quản lý một ngăn xếp các route. Việc đẩy một route lên hoặc lấy ra một route khỏi ngăn xếp Navigator kích hoạt hiệu ứng chuyển tiếp.

Route: Xác định một màn hình hoặc trang. Hầu hết các ứng dụng, ngoài các ứng dụng cơ bản nhất, có nhiều route.

**Hiệu ứng hero tiêu chuẩn**

**Điều gì đang diễn ra?**

Bay một hình ảnh từ một route sang route khác là dễ dàng để thực hiện bằng cách sử dụng tiện ích "hero" của Flutter. Khi sử dụng MaterialPageRoute để chỉ định route mới, hình ảnh bay theo một đường cong, như được mô tả trong tiêu chuẩn chuyển động của Material Design.

Tạo một ví dụ Flutter mới và cập nhật nó bằng các tệp từ hero\_animation.

Để chạy ví dụ:

Chạm vào ảnh của route gốc để bay hình ảnh đến route mới hiển thị cùng ảnh tại một vị trí và tỷ lệ khác nhau.

Quay lại route trước bằng cách chạm vào hình ảnh hoặc sử dụng cử chỉ trở về route trước trên thiết bị.

Bạn có thể làm chậm chuyển đổi hơn nữa bằng cách sử dụng thuộc tính timeDilation.

**Lớp PhotoHero**

Lớp tùy chỉnh PhotoHero duy trì hero và kích thước, hình ảnh và hành vi của nó khi chạm vào. PhotoHero xây dựng cây tiện ích (widget tree) sau: (giải thích của cây tiện ích được tiếp tục sau)

Đây là code:

class PhotoHero extends StatelessWidget {

const PhotoHero({ Key key, this.photo, this.onTap, this.width }) : super(key: key);

final String photo;

final VoidCallback onTap;

final double width;

Widget build(BuildContext context) {

return SizedBox(

width: width,

child: Hero(

tag: photo,

child: Material(

color: Colors.transparent,

child: InkWell(

onTap: onTap,

child: Image.asset(

photo,

fit: BoxFit.contain,

),

),

),

),

);

}

}

**Lớp HeroAnimation**

Lớp HeroAnimation tạo ra các PhotoHero nguồn và đích, và thiết lập chuyển tiếp.

Dưới đây là đoạn mã: (giải thích về đoạn mã này sẽ tiếp tục sau)

class HeroAnimation extends StatelessWidget {

Widget build(BuildContext context) {

timeDilation = 5.0; // 1.0 means normal animation speed.

return Scaffold(

appBar: AppBar(

title: const Text('Basic Hero Animation'),

),

body: Center(

child: PhotoHero(

photo: 'images/flippers-alpha.png',

width: 300.0,

onTap: () {

Navigator.of(context).push(MaterialPageRoute<void>(

builder: (BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('Flippers Page'),

),

body: Container(

// The blue background emphasizes that it's a new route.

color: Colors.lightBlueAccent,

padding: const EdgeInsets.all(16),

alignment: Alignment.topLeft,

child: PhotoHero(

photo: 'images/flippers-alpha.png',

width: 100.0,

onTap: () {

Navigator.of(context).pop();

},

),

),

);

}

));

},

),

),

);

}

}

Thông tin quan trọng:

* Khi người dùng chạm vào InkWell chứa hero nguồn, mã lệnh sẽ tạo ra route đích bằng cách sử dụng MaterialPageRoute. Đẩy route đích vào ngăn xếp Navigator kích hoạt hiệu ứng chuyển tiếp.
* Container định vị PhotoHero ở góc trên bên trái của route đích, phía dưới AppBar.
* Phương thức onTap() cho PhotoHero đích "pops" ngăn xếp Navigator, kích hoạt hiệu ứng bay Hero trở lại route ban đầu.
* Sử dụng thuộc tính timeDilation để làm chậm quá trình chuyển tiếp trong quá trình gỡ lỗi.

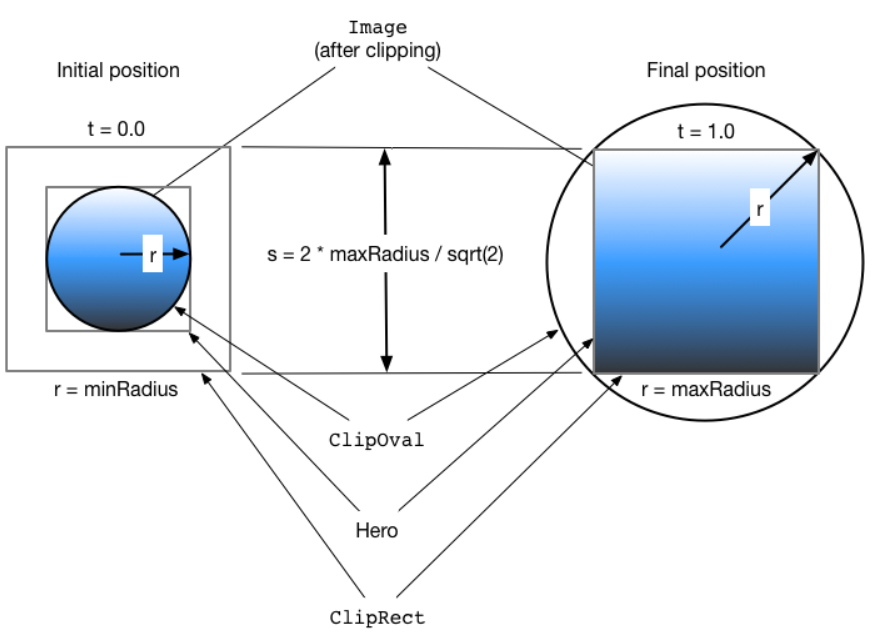
**Hiệu ứng hero theo kiểu radial (Radial hero animations)**

Bay một hero từ một route sang route khác trong khi nó chuyển từ hình tròn thành hình chữ nhật là một hiệu ứng mượt mà mà bạn có thể thực hiện bằng cách sử dụng các tiện ích (widget) "Hero". Để thực hiện điều này, mã lệnh thực hiện hiệu ứng hoán đổi hai hình dạng định cắt: một hình tròn và một hình vuông. Trong suốt quá trình hoạt động, hình tròn định cắt (và hình ảnh) được thu phóng từ minRadius đến maxRadius, trong khi hình vuông định cắt duy trì kích thước không đổi. Đồng thời, hình ảnh bay từ vị trí của nó trong route nguồn đến vị trí của nó trong route đích. Để xem các ví dụ hình ảnh của quá trình chuyển tiếp này, hãy xem Radial transformation trong tiêu chuẩn chuyển động của Material.

Hiệu ứng này có thể dường như phức tạp (và thật sự là phức tạp), nhưng bạn có thể tùy chỉnh ví dụ được cung cấp cho phù hợp với nhu cầu của bạn. Các công việc nặng nhọc đã được thực hiện cho bạn.

**Điều gì đang diễn ra?**

Sơ đồ dưới đây thể hiện hình ảnh đã bị cắt ở thời điểm bắt đầu (t = 0.0) và cuối cùng (t = 1.0) của hiệu ứng chuyển tiếp.



Các vùng màu xanh lam gradient (đại diện cho hình ảnh) cho thấy nơi các hình dạng định cắt giao nhau. Vào thời điểm bắt đầu chuyển tiếp, kết quả của sự giao nhau là một hình cắt hình tròn (ClipOval). Trong quá trình biến đổi, ClipOval co dãn từ minRadius đến maxRadius trong khi ClipRect duy trì kích thước không đổi. Cuối cùng của quá trình chuyển tiếp, sự giao nhau giữa hình cắt hình tròn và hình cắt chữ nhật tạo ra một hình chữ nhật có cùng kích thước với tiện ích hero. Nói cách khác, cuối cùng của quá trình chuyển tiếp, hình ảnh không còn bị cắt.

Tạo một ví dụ Flutter mới và cập nhật nó bằng các tệp từ thư mục GitHub radial\_hero\_animation.

Để chạy ví dụ:

* Chạm vào một trong ba hình thu nhỏ hình tròn để tạo hiệu ứng bay hình ảnh đến một hình vuông lớn được đặt ở giữa một route mới che phủ route gốc.
* Quay lại route trước bằng cách chạm vào hình ảnh hoặc sử dụng cử chỉ quay trở lại route trước trên thiết bị.
* Bạn có thể làm chậm chuyển đổi hơn nữa bằng cách sử dụng thuộc tính timeDilation.

Lớp Photo

Lớp Photo xây dựng cây tiện ích (widget tree) để chứa hình ảnh: (giải thích về cây tiện ích sẽ tiếp tục sau)

class Photo extends StatelessWidget {

Photo({ Key key, this.photo, this.color, this.onTap }) : super(key: key);

final String photo;

final Color color;

final VoidCallback onTap;

Widget build(BuildContext context) {

return Material(

// Slightly opaque color appears where the image has transparency.

color: Theme.of(context).primaryColor.withOpacity(0.25),

child: InkWell(

onTap: onTap,

child: Image.asset(

photo,

fit: BoxFit.contain,

)

),

);

}

}

Thông tin quan trọng:

* InkWell ghi nhận thao tác chạm. Hàm gọi sẽ truyền hàm onTap() vào constructor của Photo.
* Trong quá trình bay, InkWell vẽ hiệu ứng "splash" lên Material đầu tiên mà nó gặp.
* Tiện ích Material có một màu hơi mờ, vì vậy các phần trong suốt của hình ảnh được vẽ với màu. Điều này đảm bảo rằng quá trình chuyển từ hình tròn thành hình chữ nhật dễ nhìn thấy, ngay cả đối với hình ảnh có độ trong suốt.
* Lớp Photo không bao gồm Hero trong cây tiện ích của nó. Để hoạt động hiệu ứng, hero bao bọc RadialExpansion widget.

**Lớp RadialExpansion**

Tiện ích (widget) RadialExpansion, là cốt lõi của ví dụ này, xây dựng cây tiện ích để cắt hình ảnh trong quá trình chuyển tiếp. Hình dạng đã được cắt là kết quả của sự giao nhau giữa hình dạng định cắt hình tròn (mà mở rộng trong quá trình bay), với hình dạng định cắt hình chữ nhật (giữ nguyên kích thước trong suốt quá trình).

Để thực hiện điều này, nó xây dựng cây tiện ích sau đây: (giải thích về cây tiện ích sẽ tiếp tục sau)

Dưới đây là code:  
class RadialExpansion extends StatelessWidget {

RadialExpansion({

Key key,

this.maxRadius,

this.child,

}) : clipRectSize = 2.0 \* (maxRadius / math.sqrt2),

super(key: key);

final double maxRadius;

final clipRectSize;

final Widget child;

@override

Widget build(BuildContext context) {

return ClipOval(

child: Center(

child: SizedBox(

width: clipRectSize,

height: clipRectSize,

child: ClipRect(

child: child, // Photo

),

),

),

);

}

}

Thông tin quan trọng:

* Hero bao gồm RadialExpansion widget.
* Khi hero bay, kích thước của nó thay đổi và, do nó hạn chế kích thước của con, RadialExpansion widget cũng thay đổi kích thước để phù hợp.
* Hoạt động RadialExpansion được tạo bởi hai hình dạng định cắt chồng lấn nhau.
* Ví dụ xác định việc nội suy giữa các điểm dùng MaterialRectCenterArcTween. Đường bay mặc định cho hiệu ứng hero nội suy giữa các điểm dùng tweens sử dụng các góc của các hero. Phương pháp này ảnh hưởng đến tỷ lệ khung hình của hero trong quá trình biến đổi theo kiểu radial, vì vậy đường bay mới sử dụng MaterialRectCenterArcTween để nội suy giữa các điểm dùng tweens bằng cách sử dụng điểm trung tâm của mỗi hero.

Dưới đây là đoạn mã

static RectTween \_createRectTween(Rect begin, Rect end) {

return MaterialRectCenterArcTween(begin: begin, end: end);

}

1. **Opacity**
2. **Giới thiệu về Opacity trong Flutter:**

Trong Flutter, Opacity là một widget được sử dụng để điều chỉnh độ mờ của một widget con. Nó cho phép bạn thay đổi độ trong suốt của widget con từ giá trị 0.0 (hoàn toàn trong suốt - không hiển thị) đến 1.0 (hoàn toàn rõ nét - hiển thị đầy đủ).

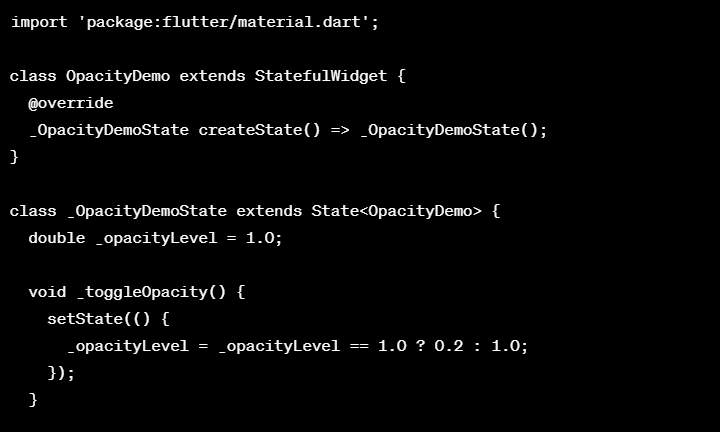
Opacity là một trong những công cụ mạnh mẽ để thực hiện các hiệu ứng hoạt hình như làm mờ, hiển thị ẩn dần hoặc hiển thị xuất hiện dần của các phần tử trong giao diện người dùng.

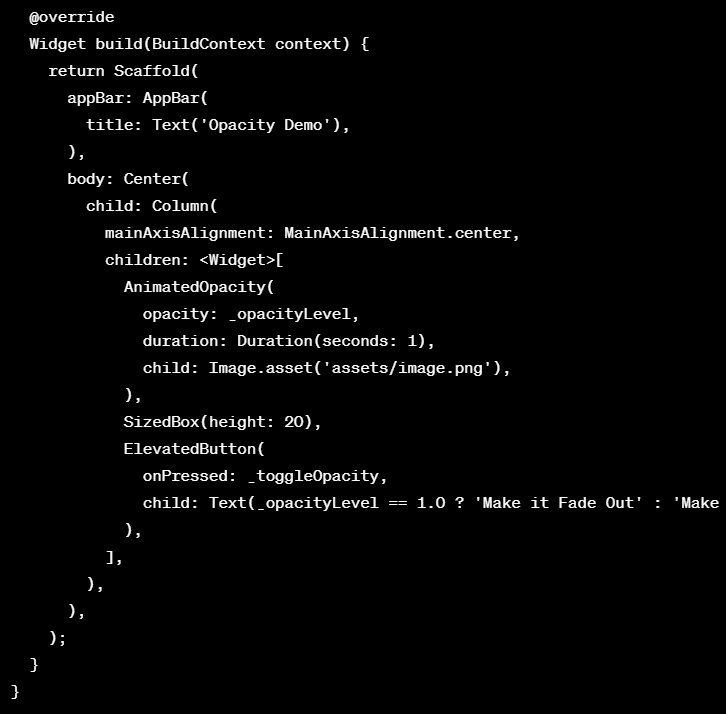
1. **Cách hoạt động của Opacity trong Flutter:**

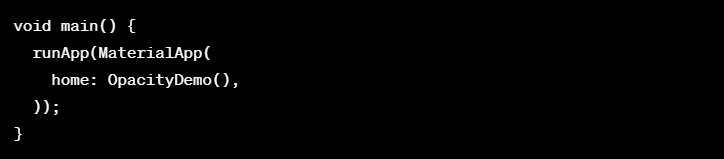
Opacity nhận một widget con (child) làm tham số và một giá trị alpha (từ 0.0 đến 1.0) để chỉ định độ mờ của widget con. Khi giá trị alpha thay đổi, widget con sẽ tự động được cập nhật lại với độ mờ tương ứng.

1. **Ví dụ sử dụng Opacity trong Flutter:**

Dưới đây là một ví dụ đơn giản về cách sử dụng Opacity trong Flutter để tạo hiệu ứng hoạt hình điều chỉnh độ mờ của một hình ảnh khi nhấn vào nút:







**Giải thích ví dụ:**

* Trong ví dụ này, chúng ta tạo một ‘OpacityDemo’ kế thừa từ ‘StatefulWidget’.
* Biến ‘\_opacityLevel’ là một giá trị dạng số duyệt từ 0.0 đến 1.0 để đại diện cho độ mờ của hình ảnh.
* Trong ‘build()’, chúng ta sử dụng ‘AnimatedOpacity’ để tạo một widget hình ảnh (image) với hiệu ứng hoạt hình điều chỉnh độ mờ dựa trên giá trị ‘\_opacityLevel’. Khi ‘\_opacityLevel’ thay đổi, hình ảnh sẽ được tự động cập nhật với độ mờ tương ứng.
* Khi nhấn vào nút "Make it Fade Out" hoặc "Make it Fade In", ‘\_toggleOpacity()’ được gọi để thay đổi giá trị ‘\_opacityLevel’ và kích hoạt hiệu ứng hoạt hình điều chỉnh độ mờ.

**Kết luận:**

Trong báo cáo này, chúng ta đã tìm hiểu về **Opacity** trong Flutter và cách sử dụng nó để điều chỉnh độ mờ của widget con. **Opacity** là một công cụ mạnh mẽ giúp bạn thêm hiệu ứng hoạt hình như làm mờ, hiển thị ẩn dần hoặc hiển thị xuất hiện dần vào ứng dụng của bạn. Ví dụ trên minh họa cách sử dụng **AnimatedOpacity** để tạo hiệu ứng hoạt hình điều chỉnh độ mờ của một hình ảnh khi nhấn vào nút.