**NỘI DUNG ANIMATIONS**

1. **AnimatedWidget:**

AnimatedWidget là một widget trong Flutter dùng để tạo hiệu ứng hoạt hình cho các thành phần giao diện của ứng dụng. Nó là một lớp con của StatefulWidget và thường được sử dụng kết hợp với các lớp khác như Animation và Tween để tạo ra các hiệu ứng di chuyển, thay đổi kích thước, màu sắc, v.v.

Cấu trúc của một AnimatedWidget bao gồm:

* Một Animation được cung cấp thông qua constructor.
* Một phương thức builder được triển khai để xác định trạng thái widget dựa trên giá trị hiện tại của Animation.

**Ví dụ minh họa:**

import 'package:flutter/material.dart';

void main() => runApp(MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text('AnimatedWidget Example')),

body: ColorChangeDemo(),

),

));

class ColorChangeDemo extends StatefulWidget {

@override

\_ColorChangeDemoState createState() => \_ColorChangeDemoState();

}

class \_ColorChangeDemoState extends State<ColorChangeDemo>

with SingleTickerProviderStateMixin {

AnimationController \_controller;

Animation<Color> \_animation;

@override

void initState() {

super.initState();

\_controller = AnimationController(

duration: const Duration(seconds: 2),

vsync: this,

);

\_animation = ColorTween(begin: Colors.blue, end: Colors.red)

.animate(CurvedAnimation(parent: \_controller, curve: Curves.easeInOut));

\_controller.repeat(reverse: true);

}

@override

void dispose() {

\_controller.dispose();

super.dispose();

}

@override

Widget build(BuildContext context) {

return Center(

child: ColorChangingBox(animation: \_animation),

);

}

}

class ColorChangingBox extends AnimatedWidget {

ColorChangingBox({Key key, Animation<Color> animation})

: super(key: key, listenable: animation);

@override

Widget build(BuildContext context) {

final Animation<Color> animation = listenable;

return Container(

width: 100,

height: 100,

color: animation.value,

);

}

}

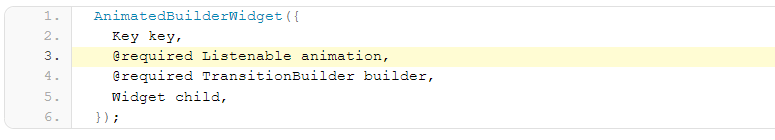
Trong ví dụ này, sử dụng AnimatedWidget có tên ColorChangingBox để thay đổi màu sắc của hộp từ màu xanh đến màu đỏ. AnimatedWidget giúp tái sử dụng giá trị Animation \_animation mà không cần phải viết lại mã lệnh cập nhật UI. Khi giá trị animation thay đổi, widget ColorChangingBox sẽ tự động được tái build và cập nhật hiển thị màu sắc mới của hộp.

**2. AnimatedBuilder:**

AnimatedBuilder Widget là một tiện ích có mục đích chung để xây dựng hoạt ảnh. AnimatedBuilder Widget hữu ích cho các widget phức tạp hơn muốn bao gồm hoạt ảnh như một phần của chức năng xây dựng lớn hơn. Để sử dụng AnimatedBuilder , chỉ cần xây dựng tiện ích con và chuyển cho nó một chức năng trình tạo.

AnimatedBuilder , một tiện ích sử dụng lệnh gọi lại của trình tạo để xây dựng lại bất cứ khi nào một Listenable nhất định kích hoạt thông báo của nó. Tiện ích này thường được sử dụng với các lớp con Hoạt hình, do đó có tên như vậy, nhưng không có nghĩa là giới hạn ở hoạt ảnh, vì nó có thể được sử dụng với bất kỳ Có thể nghe nào . Nó là một lớp con của AnimatedWidget , có thể được sử dụng để tạo các widget được điều khiển từ một Listenable .

Trình xây dựng mặc định của nó sẽ giống như bên dưới:

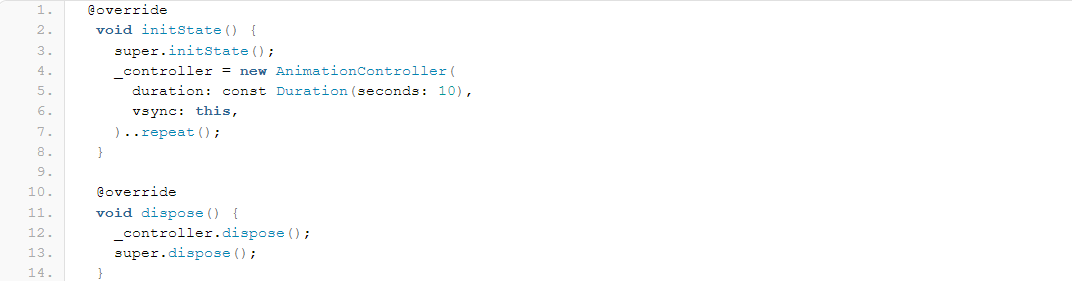


**Làm cách nào để sử dụng Widget AnimatedBuilder trong Flutter?**

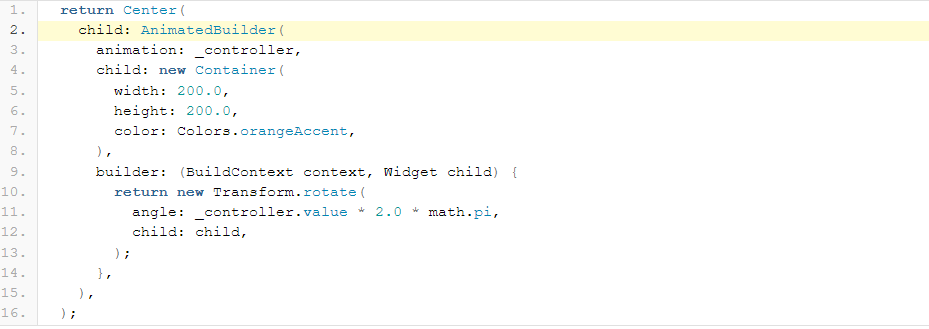
Đoạn mã sau cho chúng ta biết cách triển khai AnimatedBuilder Widget trong Flutter.

Hãy xem xét một đoạn mã như bên dưới với sự trợ giúp của Ví dụ như bên dưới.

Tạo một Stateful Widget như bên dưới và xác định một initState() và disconize() sẽ có một đoạn mã như bên dưới:



Bây giờ một build() sẽ có một đoạn mã như bên dưới:



Mã nguồn hoàn chỉnh sẽ giống như bên dưới:



## 3. AnimationController:

AnimationController là một đối tượng Animation đặc biệt tạo ra một giá trị mới bất cứ khi nào phần cứng sẵn sàng cho một khung mới. Mặc định, AnimationContoder tạo tuyến tính các số từ 0,0 đến 1,0 trong một khoảng thời gian nhất định. Ví dụ sẽ tạo ra một AnimationController

final AnimationController controller = AnimationController (

duration: const Duration (milliseconds: 2000), vsync: this);

[AnimationController](https://docs.flutter.io/flutter/animation/AnimationController-class.html) hỗ trợ phương thức để điều khiển animation.

* Điều khiển animation [forward](https://docs.flutter.io/flutter/animation/AnimationController/forward.html) hoặc [reverse](https://docs.flutter.io/flutter/animation/AnimationController/reverse.html), hoặc [stop](https://docs.flutter.io/flutter/animation/AnimationController/stop.html)
* Đặt animation thành một [value](https://docs.flutter.io/flutter/animation/AnimationController/value.html) cụ thể.
* Định nghĩa các giá trị [upperBound](https://docs.flutter.io/flutter/animation/AnimationController/upperBound.html) and [lowerBound](https://docs.flutter.io/flutter/animation/AnimationController/lowerBound.html) trong một animation.
* Tạo ra một [fling](https://docs.flutter.io/flutter/animation/AnimationController/fling.html) animation với hiệu ứng sử dụng một mô phỏng vật lý.

Khi tạo AnimationController, bạn truyền cho nó một đối số vsync, ngăn animations tiêu tốn tài nguyên không cần thiết bên ngoài màn hình. Bạn có thể sử dụng đối tượng trạng thái của mình làm vsync bằng cách thêm SingleTickerProviderStateMixin vào định nghĩa lớp. Ví dụ

class \_LogoAppState extends State<LogoApp> with SingleTickerProviderStateMixin {

...

}

## 3.1. Tween:

Mặc định, đối tượng AnimationController có giá trị nằm trong khoảng từ 0,0 đến 1,0. Nếu bạn cần thay đổi khoảng giá trị khác hoặc một loại dữ liệu khác, bạn có thể sử dụng [Tween](https://docs.flutter.io/flutter/animation/Tween-class.html) .

Tween là một đối tượng không có trạng thái, chỉ có lưu giá trị begin và end. Công việc duy nhất của Tween là xác định ánh xạ từ phạm vi đầu vào sang phạm vi đầu ra. Phạm vi đầu vào thường là 0,0 đến 1,0.

Ví dụ sau cho phép ta thay đổi giá trị của một Tween<double> nằm trong khoảng từ -200.0 tới 0.0 :

final Tween doubleTween = Tween<double>(begin: -200.0, end: 0.0);

Tween kế thừa từ Animatable<T>, chứ không phải từ Animation<T>. Một Animatable tương tự như Animation hỗ trợ nhiều kiểu khác ngoài **double**

**Ví dụ** như [ColorTween](https://docs.flutter.io/flutter/animation/ColorTween-class.html):

final Tween colorTween =ColorTween(begin: Colors.transparent, end: Colors.black54);

[IntTween](https://docs.flutter.io/flutter/animation/IntTween-class.html):

final Tween intTween = IntTween(begin: 0, end: 255);

Một đối tượng Tween không lưu trữ bất kỳ trạng thái. Thay vào đó, nó cung cấp phương thức evaluate(Animation<double> animation) áp dụng chức năng ánh xạ cho giá trị hiện tại của animation. Giá trị hiện tại của đối tượng Animation có thể được tìm thấy trong phương thức .value.

### **3.2. Tween.animate**

Để sử dụng Tween, bạn cần gọi phương thức animate() và đặt vào một AnimationController. Phương thức này trả về một **Animation**, chứ không phải Animatable

Ví dụ sau sẽ tạo ra các giá trị từ 0 -> 255 trong 500 ms cho một AnimationController

final AnimationController controller = AnimationController(

duration: const Duration(milliseconds: 500), vsync: this);

Animation<int> alpha = IntTween(begin: 0, end: 255).animate(controller);

Hiển thị AnimationController, CurvedAnimation và Tween.

final AnimationController controller = AnimationController(

duration: const Duration(milliseconds: 500), vsync: this);

final Animation curve =

CurvedAnimation(parent: controller, curve: Curves.easeOut);

Animation<int> alpha = IntTween(begin: 0, end: 255).animate(curve);

**4. CuvedAnimation:**

**4.1. Định nghĩa CuvedAnimation:**

CuvedAnimation là lệnh thay đổi giá trị cuved của các hoạt ảnh trong flutter.

Tùy vào giá trị được cho mà kết quả đầu ra của CurvedAnimation sẽ lớn hơn nhiêu so với đầu vào.

**Ví dụ như**: tầm elastic curves sẽ lớn hoặc nhỏ hơn đáng kể dù đầu vào của nó chỉ từ 0.0 - 1.0

**4.2. Cấu trúc của CuvedAnimation:**

CurvedAnimation({required Animation<double> parent, required Curve curve, Curve? reverseCurve})

* Một số lệnh CuvedAnimation:
* curve ↔ Curve
* hashCode → int
* isCompleted → bool
* isDismissed → bool
* isDisposed ↔ bool
* parent → Animation<double>
* reverseCurve ↔ Curve?
* runtimeType → Type
* status → AnimationStatus
* value → double

**4.3. Một số cách sử dụng:**

addListener(VoidCallback listener) → void

* Gọi “listener” mỗi khi giá trị animation thay đổi

addStatusListener(AnimationStatusListener listener) → void

* Gọi “listener” mỗi khi trạng thái của animation thay đổi

dispose() → void

* Xóa những “listeners” đã được thêm bởi CurvedAnimation

drive<U>(Animatable<U> child) → Animation<U>

* Kết Tween (hoặc CurveTween) vào animation

noSuchMethod(Invocation invocation) → dynamic

* Truy cập khi method không tồn tại hoặc bị hán chế truy cập

removeListener(VoidCallback listener) → void

* Ngừng gọi “listener” mỗi khi giá trị của animation thay đổi

removeStatusListener(AnimationStatusListener listener) → void

* Ngừng gọi “listener” mỗi khi trạng thái animation thay đổi

toString() → String

* Đặt đối tượng “string” thành “giá trị này”

toStringDetails() → String

* Cung cấp một chuổi mô tả trạng thái của “object” nhưng ngoại trừ thông tin của chính “object”

**5. Hero:**

**5.1. Giới thiệu về Hero animations:**

Hero Animation là một widget giúp chúng ta tạo các hiệu ứng khi chuyển màn. Dưới đây là một ví dụ về Hero Animations.

A screenshot of a phone

Description automatically generated

Hero Animation sử dụng một biểu tượng hiện được gọi là “Hero” và khi quá trình chuyển đổi trang được kích hoạt, thường bằng cách nhấp vào biểu tượng, trên màn hình sẽ xuất hiện hoạt ảnh “bay” đến trang tiếp theo. Khi người dùng điều hướng trở lại trang trước đó, hoạt ảnh sẽ đi theo hướng khác và biểu tượng quay trở lại vị trí ban đầu của nó.

Chúng ta sẽ thảo luận không chỉ về các Hero Animations cơ bản mà còn những thứ chúng ta có thể tùy chỉnh về nó. Trước tiên, hãy xem những hiệu ứng cơ bản.

**5.2. Tạo Hero Animation cơ bản:**

Hero Animations là một trong những hiệu ứng dễ làm nhất trong Flutter và không yêu cầu thiết lập nhiều. Nhìn vào ví dụ dưới đây, chúng ta có thể thấy rằng cùng một biểu tượng tồn tại trên cả hai trang. Tất cả những gì chúng ta cần là nói cho Flutter rằng cả hai được liên kết với nhau. Ta thực hiện việc này bằng cách bao bọc một widget như một biểu tượng trong Hero widget.

Hero(

tag: "DemoTag",

child: Icon(

Icons.add,

size: 70.0,

),

),

Ta cung cấp cho mỗi Hero widget một tên cụ thể. Điều này là cần thiết vì nếu chúng ta có thể có nhiều Hero trên cùng một màn hình và mỗi Hero wigdet sẽ có những điểm đến khác nhau

Bây giờ ứng dụng biết rằng có một Hero widget muốn chuyển đến trang tiếp theo. Bây giờ tất cả những gì chúng ta cần làm là xác định nơi mà Hero Widget sẽ bay tới.

Tất cả những gì chúng ta cần là tạo một Hero Widget ở màn hình thứ hai với tên giống với tên hero widget ở màn hình thứ nhất.

Hero(

tag: "DemoTag",

child: Icon(

Icons.add,

size: 150.0,

),

),

Và đây là kết quả chúng ta thu được:

A white background with small dots

Description automatically generated

**5.3. Customizing Hero Animations**

Hero Animations cho phép chúng ta tùy chỉnh các thông số để tạo ra các hoạt ảnh mà chúng ta mong muốn. Cùng xem cách làm như nào nhé.

**5.3.1. Thêm placeholders**

Trong khi tiện ích con bay từ màn hình này qua màn hình khác, sẽ có không gian trống ở điểm đến. Ta có thể placeholder vào vị trí này. Bây giờ chúng ta hãy sử dụng CircularProgressIndicator để làm placeholder.

Hero(

tag: "DemoTag",

child: Icon(

Icons.add,

size: 150.0,

),

placeholderBuilder: (context, widget) {

return Container(

height: 150.0,

width: 150.0,

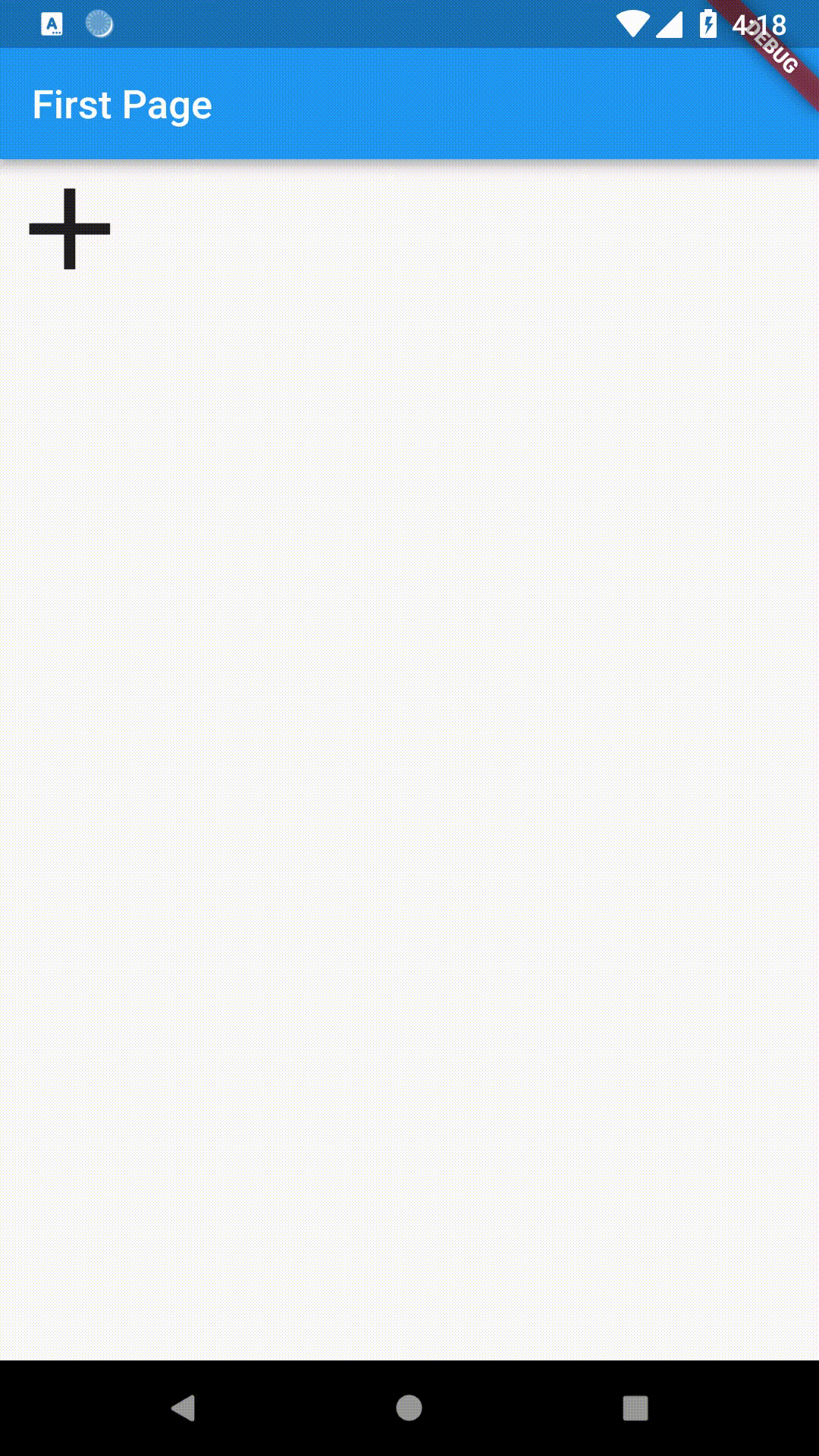
child: CircularProgressIndicator(),

);

},

),

Và đây là kết quả chúng ta thu được:



**5.3.2 Thay đổi hero widget:**

Flutter cho phép chúng ta thay đổi widget thực sự di chuyển từ màn hình này sang màn hình khác mà không thay đổi các widget mà chúng ta setup từ đầu trên cả hai màn hình. Dưới đây ta sẽ dùng biểu tượng tênlửatronglúcdi chuyển từ màn hình này sang màn hình khác thay cho icon "+"

A white background with small dots

Description automatically generated

Ta sử dụng thuộc tính \*\*flightShuttleBuilder \*\* của hero animation để làm điều đó.

Hero(

tag: "DemoTag",

child: Icon(

Icons.add,

size: 150.0,

),

flightShuttleBuilder: (flightContext, animation, direction,

fromContext, toContext) {

return Icon(FontAwesomeIcons.rocket, size: 150.0,);

},

),

Phương thức flightShuttleBuilder có 5 tham số và cho chúng ta hoạt ảnh cũng như hướng của hoạt ảnh . Hiện tại, kích thước biểu tượng tên lửa vẫn ở mức 150,0 cho cả hai hướng. Chúng ta có thể có các cấu hình khác nhau cho mỗi hướng bằng cách sử dụng tham số **direction** của phương thức.

// push : từ màn hình thứ nhất về màn hình thứ hai.

// pop: từ màn hình thứ hai quay lại màn hình thứ nhất.

if(direction == HeroFlightDirection.push) {

return Icon(

FontAwesomeIcons.rocket,

size: 150.0,

);

} else if (direction == HeroFlightDirection.pop){

return Icon(

FontAwesomeIcons.rocket,

size: 70.0,

);

}

A white background with small dots

Description automatically generated

**5.3.3 Tạo hiệu ứng vuốt ngược giống với IOS bằng Hero Animation:**

Mặc định, Hero Animation hoạt động khi người dùng sử dụng phím **back** chứ không hoạt động với hành động back swipe. Tuy nhiên chúng ta có thể tạo ra hiệu ứng này một cách dễ dàng với Hero Animations

Sử dụng back button :

A white background with small dots

Description automatically generated

Sử dụng back swipe :

A black cross on a white background

Description automatically generated

Để làm việc này chúng ta sử dụng tham số **transitionOnUserGestures** của Hero Widget ở cả 2 màn hình:

Hero(

tag: "DemoTag",

child: Icon(

Icons.add,

),

transitionOnUserGestures: true,

),

**6. Opacity:**

Opacity class là một widget khiến lớp child trở nên trong suốt. Lớp này sẽ đè lên lớp child vào vùng đệm trung gian khiến nó trong suốt một phần.

**Ví dụ như** với giá trị từ 0.0 đến 1.0 sẽ ẩn lớp child tương ứng vào vùng đệm từ 0% đến 100%.

Công dụng của vùng đệm trung gian này sẽ khiến chức năng của nhiều lớp child thay đổi hoàn toàn.

**Ví dụ như** “BackdropFilter child” sẽ chỉ có thể ứng dụng vào những nội dụng nằm giữa widget này và backdrop child.

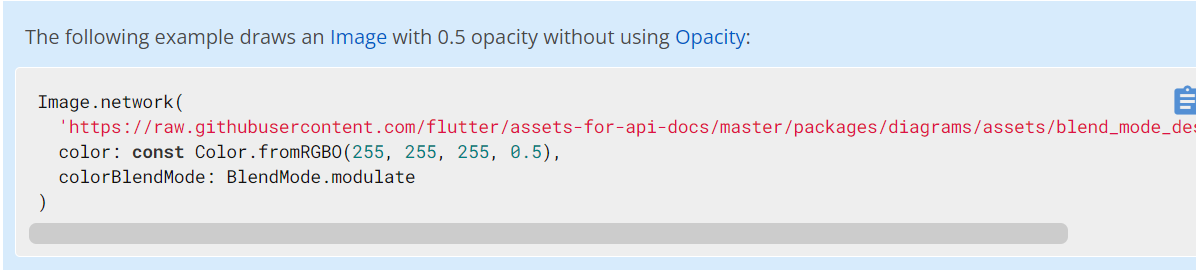
Giá trị trong suốt của opacity animation có thể thay đổi và tái kiến trúc mỗi frame khiến thuật toán chạy không hiệu quả, cho nên ta có 2 lệnh sau:

* **AnimatedOpacity**: sẽ thảy đổi giá trị opacity bên trong nó, giúp thuật toán hiệu quả hơn
* **FadeTransition:** cung cấp phương pháp khác tốt hơn để tha đổi giá trị opacity

**Transparent image:**

Phương pháp này khiến cho hình ảnh hoặc màu sắc của đối tượng trở nên trong suốt hoặc mờ đi tùy vào giá trị opacity

**Ví dụ:**



Khi ta làm trong suốt ảnh trực tiếp thì thuật toán sẽ nhanh hơn bởi vì thông thường opacity sẽ tác động lên các nhóm widget trước rồi sau đó mới là hình ảnh đồng thời như thế sẽ giảm “lượng” vùng đệm được sử dụng.

**6.1. Cấu trúc:**

Opacity({Key? key, required double opacity, bool alwaysIncludeSemantics = false, W idget? child})

**6.2. Các câu lệnh phổ biến:**

* alwaysIncludeSemantics → bool
* child → Widget?
* hashCode → int
* key → Key?
* opacity → double
* runtimeType → Type

**6.3. Một số cách sử dụng phổ biển:**

* createElement() → SingleChildRenderObjectElement
* createRenderObject(BuildContext context) → RenderOpacity
* debugDescribeChildren() → List<DiagnosticsNode>
* debugFillProperties(DiagnosticPropertiesBuilder properties) → void
* didUnmountRenderObject(covariant RenderObject renderObject) → void
* noSuchMethod(Invocation invocation) → dynamic
* toDiagnosticsNode({String? name, DiagnosticsTreeStyle? style}) → DiagnosticsNode
* toStringShallow({String joiner = ', ', DiagnosticLevel minLevel = DiagnosticLevel.debug}) → String
* toStringShort() → String
* updateRenderObject(BuildContext context, covariant RenderOpacity renderObject) → void
* toStringDeep({String prefixLineOne = '', String? prefixOtherLines, DiagnosticLevel minLevel = DiagnosticLevel.debug}) → String