Họ và tên: Nguyễn Thị Ngọc Huyền

MSSV: 0850070020

Lớp: 08\_TTMT

TÌM HIỂU VỀ ANIMATIONS TRONG FLUTTER

1. **AnimatedWidget**

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.

**- Common animated widgets**

Chúng thường được đặt tên FooTransition, đâu Foolà tên của phiên bản không hoạt hình của tiện ích con đó. Không nên nhầm lẫn các lớp con của lớp này với các lớp con của ImplicitlyAnimatedWidget (xem ở trên), thường được đặt tên là AnimatedFoo. Các vật dụng hoạt hình thường được sử dụng bao gồm:

ListenableBuilder, sử dụng mẫu trình tạo hữu ích cho các trường hợp sử dụng Listenable phức tạp .

* AnimatedBuilder, sử dụng mẫu trình tạo hữu ích cho các trường hợp sử dụng hoạt hình phức tạp .
* AlignTransition, là phiên bản động của Align.
* DecoredBoxTransition, là phiên bản hoạt hình của DecoredBox.
* DefaultTextStyleTransition, là phiên bản động của DefaultTextStyle.
* PositionedTransition, là phiên bản hoạt hình của Positioned.
* RelativePositionedTransition, là phiên bản hoạt hình của Positioned.
* RotationTransition, tạo hiệu ứng xoay của tiện ích con.
* ScaleTransition, tạo hoạt ảnh theo tỷ lệ của tiện ích con.
* SizeTransition, tạo hoạt ảnh cho kích thước của chính nó.
* SlideTransition, làm sinh động vị trí của một tiện ích con so với vị trí bình thường của nó.
* FadeTransition, là phiên bản động của Opacity.
* AnimatedModalBarrier, là phiên bản hoạt hình của ModalBarrier.

**Inheritance**

Object > DiagnosticableTree > Widget > StatefulWidget > AnimatedWidget

1. **AnimatedBuilder**

- AnimatedBuilder hữu ích cho các vật dụng phức tạp hơn muốn bao gồm hoạt ảnh như một phần của chức năng xây dựng lớn hơn. Để sử dụng AnimatedBuilder , hãy xây dựng tiện ích con và chuyển cho nó một chức năng trình tạo.

- Performance optimizations

Nếu hàm xây dựng chứa một cây con không phụ thuộc vào hoạt ảnh được chuyển đến hàm tạo, thì việc xây dựng cây con đó một lần sẽ hiệu quả hơn thay vì xây dựng lại nó trên mỗi dấu tích hoạt hình.

Nếu một cây con dựng sẵn được truyền dưới dạng tham số con, thì AnimatedBuilder sẽ chuyển nó trở lại chức năng của trình tạo để nó có thể được tích hợp vào bản dựng.

Việc sử dụng công cụ con dựng sẵn này là hoàn toàn tùy chọn nhưng có thể cải thiện hiệu suất đáng kể trong một số trường hợp và do đó là một phương pháp hay.

Inheritance

Object > DiagnosticableTree > Widget > StatefulWidget > AnimatedWidget > ListenableBuilder > AnimatedBuilder

1. **AnimationController**

- AnimationController là một đối tượng đặc biệt tạo ra một giá trị mới bất cứ khi nào phần cứng đã sẵn sàng cho một khung hình mới. Theo mặc định, Một tuyến tính tạo ra các con số từ 0,0 đến 1,0 trong một khoảng thời gian nhất định. Ví dụ: mã này tạo ra một đối tượng, nhưng không bắt đầu chạy.

- AnimationController bắt nguồn từ , vì vậy nó có thể được sử dụng bất cứ nơi nào cần một đối tượng. Tuy nhiên, có các phương pháp bổ sung để kiểm soát hoạt ảnh. Ví dụ: bạn bắt đầu một hình ảnh động với phương pháp. Việc tạo ra các con số là gắn liền với việc làm mới màn hình, vì vậy thông thường 60 số được tạo cho mỗi thứ hai. Sau khi mỗi số được tạo, mỗi đối tượng gọi các đối tượng kèm theo.

- Khi tạo một , bạn truyền cho nó một đối số. Sự hiện diện của ngăn chặn hình ảnh động ngoài màn hình tiêu thụ tài nguyên không cần thiết. Bạn có thể sử dụng đối tượng trạng thái của mình làm vsync bằng cách thêm vào định nghĩa lớp. Bạn có thể xem ví dụ về điều này trong animate1 trên GitHub.

- 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 ngược lại hoặc dừng hoạt hình.

+ Đặt hoạt hình thành một giá trị cụ thể.

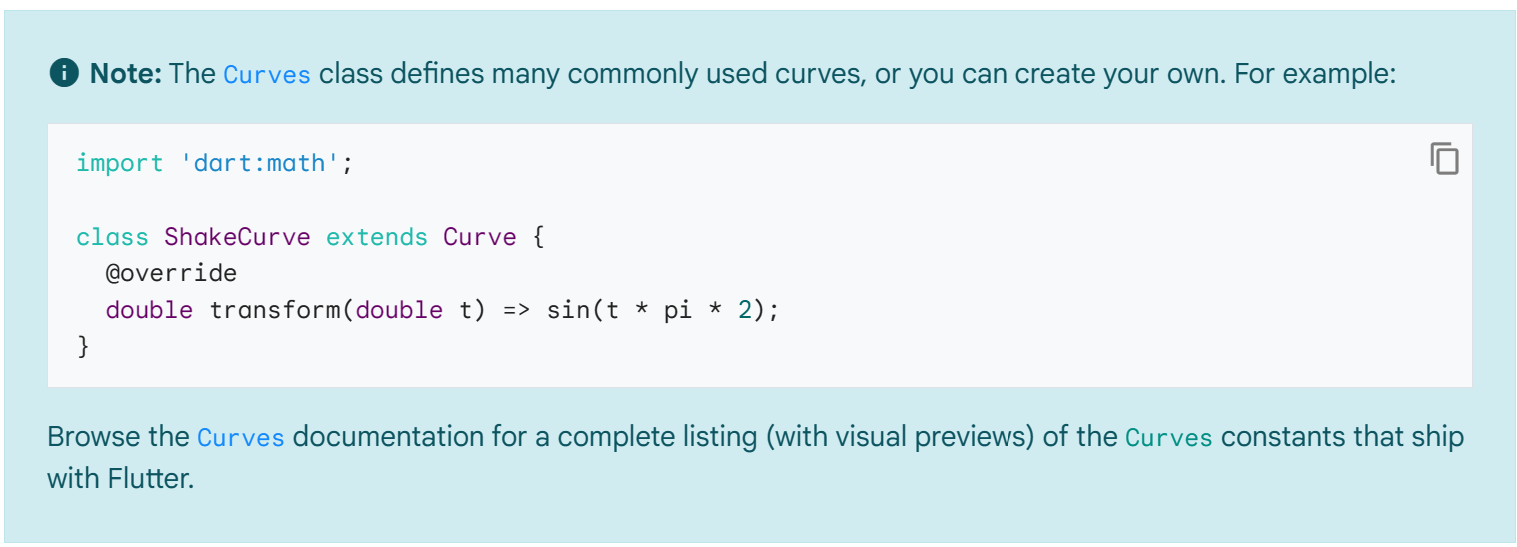
+ Xác định giá trị upperBound và lowerBound của hoạt hình.

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

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

1. **CuvedAnimation**

- A CurvedAnimation xác định tiến trình của hoạt hình như một đường cong phi tuyến tính.



- CurvedAnimation rất hữu ích khi bạn muốn áp dụng Đường cong phi tuyến tính cho Một đối tượng hoạt hình, đặc biệt nếu bạn muốn các đường cong khác nhau khi Hoạt hình đang tiến lên so với khi nó đi lùi.

- Tùy thuộc vào đường cong nhất định, đầu ra của CurvedAnimation có thể có phạm vi rộng hơn đầu vào của nó. Ví dụ, các đường cong đàn hồi như Curves.elasticIn sẽ vượt quá đáng kể hoặc thiếu mặc định phạm vi từ 0,0 đến 1,0.

- Nếu bạn muốn áp dụng Curve cho Tween, hãy cân nhắc sử dụng CurveTween.

1. **Hero**



* **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.

* **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. Chúng tôi 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,

),

),

Chúng tôi 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,

),

),

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.

Hero(

tag: "DemoTag",

child: Icon(

Icons.add,

size: 150.0,

),

placeholderBuilder: (context, widget) {

return Container(

height: 150.0,

width: 150.0,

child: CircularProgressIndicator(),

);

},

),

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 chúng tôi sẽ dùng biểu tượng tên lửa trong lúc di chuyển từ màn hình này sang màn hình khác thay cho icon "+"

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,

);

}

* **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

1. **Opacity**



* **Ẩn Widget (full opacity)**

Giả sử bạn có 1 list Widget được sắp xếp theo Column như sau:

class SomeWidget extends StatelessWidget {

@override

Widget build(BuildContext context) {

final widgets = [

MyWidget(Colors.green),

MyWidget(Colors.blue),

MyWidget(Colors.yellow),

];

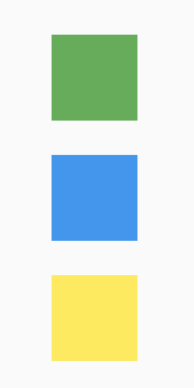
return Column(

children: widgets,

);

}

}



Thật dễ dàng để loại bỏ màu xanh lam chỉ bằng cách rebuild mà không dùng nó (comment code chỗ đó lại như sau):

final widgets = [

MyWidget(Colors.green),

// MyWidget(Colors.blue),

MyWidget(Colors.yellow),

];

Nhưng điều đó đã khiến Widget màu vàng di chuyển lên để chiếm vị trí của Widget màu xanh lam. Nếu bạn muốn bố cục giữ nguyên và chỉ làm cho Widget màu xanh lam biến mất, bạn có thể bọc nó trong một **Opacity** widget.

final widgets = [

MyWidget(Colors.green),

Opacity(

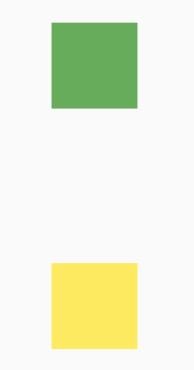
opacity: 0.0,

child: MyWidget(Colors.blue),

),

MyWidget(Colors.yellow),

];



Opacity widget rất hữu ích nếu bạn muốn làm cho toàn bộ Widget trong suốt. Tuy nhiên có lúc bạn chỉ cần tạo một màu trong suốt duy nhất mà thôi. [Đây](https://www.youtube.com/watch?v=_jKL5IY3OOM) là một ví dụ về AppBar trong suốt:

Opacity bằng 0.0 nghĩa là nó hoàn toàn trong suốt. Nếu bạn muốn làm cho nó hoàn toàn transparent (nghĩa là không còn trong suốt (transparency)), bạn sẽ đặt opacity thành 1.0. Bất kỳ thứ gì trong khoảng từ 0.0 đến 1.0 đều làm cho Widget trở nên trong suốt dần.

Chạy đoạn code trên ta được kết quả sau. Widget màu xanh lam vẫn ở đó, nhưng bạn không thể nhìn thấy nó.

