ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

**Topic 8:**

**ANIMATION LIBRARIES**

**Nhóm:**

18127014 – Huỳnh Nhật Nam

18127092 – Phạm Vũ Duy

18127223 – Nguyễn Phúc Thịnh

**Lớp:** 18KTPM

**Môn : Phát triển ứng dụng cho TBDĐ nâng cao**

Thành phố Hồ Chí Minh năm 2022

# GIỚI THIỆU

\* Lưu ý: Thư viện animation cơ bản được tích hợp sẵn trong flutter tương đối ở level thấp. Nếu muốn tạo một animation phức tạp thì nên coi thử một số animation package cung cấp interface ở mức cao hơn trên pub.dev

## Tại sao nên dùng animation

* Animation nếu được thiết kế tốt có thể khiến cho UI của ứng dụng trở nên mượt mà và tự nhiên hơn. Điều đó khiến cho trải nghiệm của người dùng có thể tốt hơn làm tăng sức cạnh tranh của ứng dụng trên thị trường.

## Làm thế nào để chọn animation

* Flutter hỗ trợ rất nhiều widget và package để cung cấp cách tạo và sử dụng animation.
* Về cơ bản có 2 loại animation chính: drawing-based animations và code-based animations:
  + Drawing-based animation: Các loại animation trông như 1 hình vẽ. Ví dụ như hình một con cánh cụt chyển động. Để tạo các animation loại này ta có thể dùng một số thư viện như Flare hoặc Lottie nếu animation chứa các đồ họa vector hoặc ảnh.
  + Code-based animation: Các animation liên quan đến code và tập trung vào các widget của flutter như Text, Column, Row,.. Các loại code-based animation có tác dụng hỗ trợ cho các widget thực hiện animation như đổi màu, đổi kích cỡ,.. Code-based animations cũng chia ra làm 2 loại. explicit animation và implicit animation:
    - Implicit animation: đặt một giá trị mới cho widget (ví dụ height của container hiện tại là 100 và đặt giá trị mới là 200) và flutter sẽ lo việc animate sự thay đổi đó trên UI (ví dụ: AnimatedPositioned widget)
    - Explicit animation: chỉ bắt đầu khi yêu cầu controller bắt đầu quá trình animate. Vì thế nên explicit animation cần có AnimationController. Ví dụ:

var \_controller = AnimationController(

duration: Duration(seconds: 1),

vsync: this,

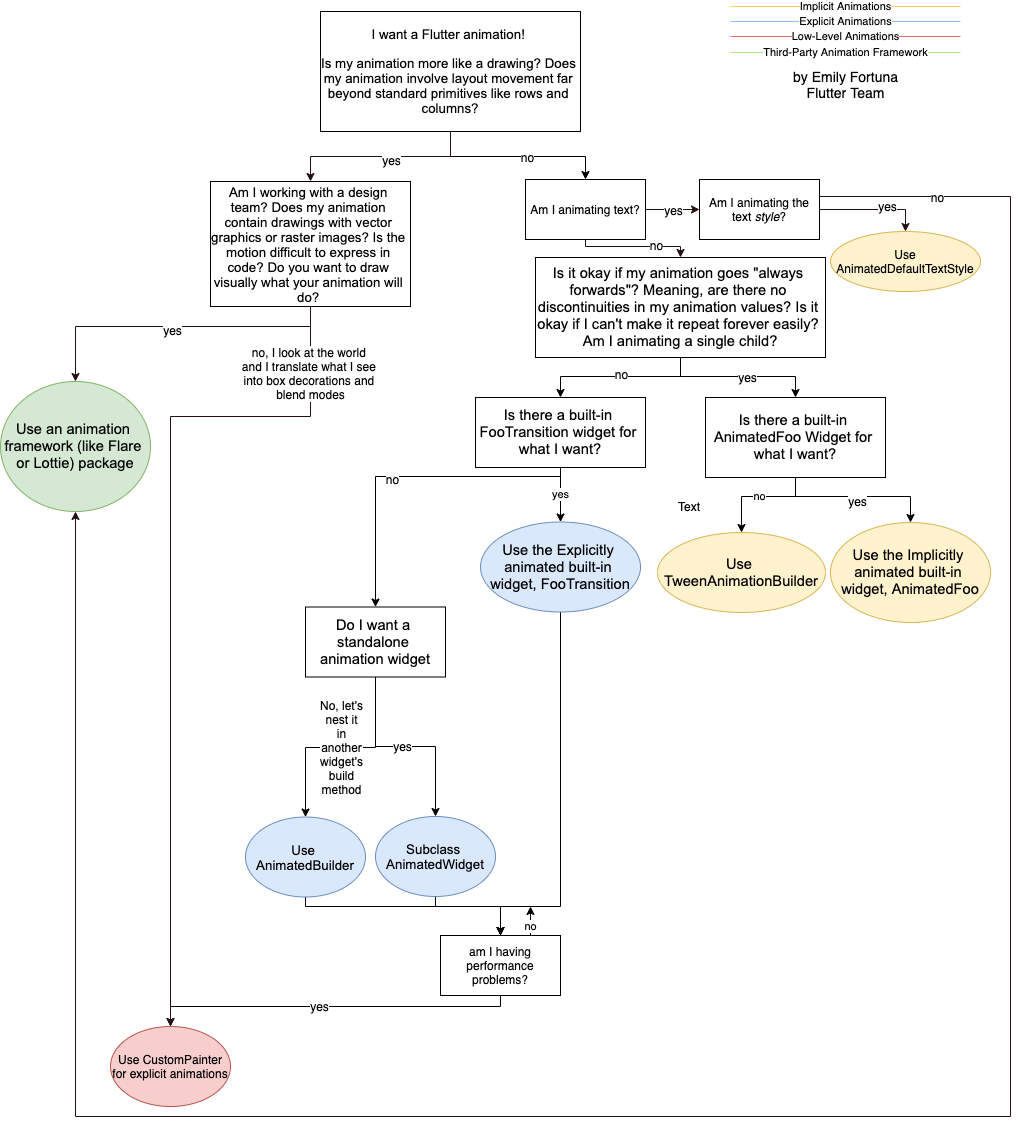
)

…

\_controller.forward()

Lưu ý rằng AnimationController cần phải được quản lý trong State. Khởi tạo ở initState() và dispose ở dispose() của State

* Quá trình chọn lựa công cụ phù hợp với yêu cầu animation được thể hiện qua hình dưới:



# TỔNG QUAN

Giới thiệu tổng quan về một số lớp căn bản của thư viện animation có sẵn của flutter.

## Lớp Animation:

* Hệ thống animation trong flutter hoạt động dựa trên các object kiểu Animation. Animation tượng trưng cho một giá trị của kiểu nhất định. Đa số widget thực hiện animation đều sẽ nhận được một đối tượng Animation làm tham số và dựa vào giá trị đó để thực hiện animation.
* Đối với các đối tương Animation. Ta có thể gọi hàm addListener để lắng nghe sự thay đối giá trị của đối tượng. Thường một đối tượng State khi lắng nghe một đối tượng Animation sẽ gọi setState() cho chính nó nếu giá trị đối tượng Animation có sự thay đổi.
* Có 2 loại widget hỗ trợ cho việc rebuild khi giá trị trên thay đổi là AnimatedWidget và AnimatedBuilder:
  + AnimatedWidget: thường hữu dụng cho một stateless animated widget riêng lẻ. Cài đặt bằng cách subclass AnimatedWidget và override hàm build
  + AnimatedBuilder: thường hữu dụng cho một widget phức tạp mà ta chỉ muốn thêm animation là 1 phần của hàm build.
* Animation cũng cung cấp AnimationStatus và ta có thể sử dụng addStatusListener() để lắng nghe nó. Animation có các status là dismissed, forward, reverse, completed. Ví dụ Animation có giá trị tăng từ 0.0 đến 1.0. Các animation bắt đầu từ dismissed là 0.0, forward là đang chạy từ 0.0 đến 1.0, reverse là chạy từ 1.0 đến 0.0, completed khi giá trị đạt 1.0

## AnimationController:

* AnimationController là lớp điều khiển các animation.

## Tweens:

* Tweens là từ ngắn gọn cho In-betweening. Tween<T> là lớp giúp ta animate với khoảng giá trị không chỉ đơn giản như là 0.0 tới 1.0 mà được nội suy từ giá trị begin đến giá trị end. Flutter hỗ trợ một số loại Tween theo type cụ thể như ColorTween và ReactTween. Ta có thể tự tạo một subclass của Tween và override hàm lerp.
* Tween căn bản chỉ định nghĩa cách nội suy một giá trị trong khoảng begin và end. Nếu muốn lấy một giá trị cụ thể cho một frame của animation, ta cần phải có thêm một đối tượng Animation. Ta có thể dùng evaluate() để lấy giá trị từ đối tượng Animation hoặc dùng animate() để tạo ra một đối tượng Animation mới.

## Kiến trúc:

### Scheduler:

* Mỗi khi một frame cần được thể hiện trên màn hình. Engine của Flutter sẽ trigger một “begin frame” callback cho tất cả các listener sử dụng hàm scheduleFrameCallback(). Tất cả các callback này đều được cung cấp một Duration() .Nhờ đó tất cả các animation hoạt động nhờ callback này sẽ thể hiện trên màn hình một cách đồng bộ.

### Tickers:

* Ticker được sử dụng trong cơ chế của scheduleFrameCallback() để gọi callback mỗi tick.
* Bởi vì với mỗi tick, Ticker cung cấp callback với khoảng thời gian trôi qua của ticker kể từ tick đầu tiên chứ không phải từ thời gian nó bắt đầu cho nên tất cả các ticker đều đồng bộ,

### Simulations:

* Simulation abstract class map một giá trị thời gian (khoảng thời gian trôi qua) với một giá trị double và có định nghĩa về sự hoàn thành của nó.
* Có nhiều loại concrete Simulation (xem [physics library](https://api.flutter.dev/flutter/physics/physics-library.html)) cho các loại hiệu ứng ví dụ như BouncingScrollSimulation để mô phỏng hiệu ứng scroll của iOS và ClampingScrollSimulation để mô phỏng hiệu ứng scroll của Android

### Animatables:

* Animatable abstract class map một giá trị double với một giá trị thuộc kiểu cụ thể.
* Các lớp Animatable là stateless và immutable.

#### Tween:

* Tween là subclass chính của Animatable. Abstract class Tween<T> map một giá trị double được định danh trong khoảng từ 0.0 đến 1.0 với một giá trị thuộc kiểu xác định (ví dụ Color). Xem phần Tween ở trên

#### Composing animatables:

* Ta có thể tạo một animatable mà sẽ sử dụng mapping của đối tượng cha sau đó đến mapping của đối tượng con bằng cách truyền một đối tượng cha Animatable<double> cho hàm chain() của đối tượng con

### Curves:

* abstract class Curve map các giá trị định danh double trong khoảng 0.0 đến 1.0 với các giá trị định danh double trong cùng khoảng. ([Chi tiết](https://api.flutter.dev/flutter/animation/Curves-class.html))

### Animations:

* Abstract class Animation cung cấp: giá trị thuộc một kiểu cụ thể, khái niệm phương hướng của animation và trạng thái animation, interface để gán callback mỗi khi giá trị hoặc status thay dổi.
* Lớp Animation<double> là một lớp đặc biệt bởi vì nó có thể được sử dụng cho các giá trị định danh double trong khoảng 0.0 – 1.0, các giá trị này là input cho Curve và Tween cũng như một số subclass khác của Animation
* Một số subclass của Animation là stateless, chỉ chuyển tiếp các listener tới parent của chúng. Một số thì lại stateful.

#### Composable animations:

* Đa số subclass của Animation nhận đôi tượng cha là Animation<double> và hoạt động dựa trên đối tượng đó. Ví dụ như Curved Animation, ReverseAnimation, ProxyAnimation, TrainHopping Animation.

#### Animation controllers:

* AnimationController là một stateful Animation<double> sử dụng Ticker. Nó có thể được khởi động hoặc dừng. Với mỗi tick, AnimationController lấy thời gian trôi qua kể từ khi nó được khởi động và truyền thời gian đó cho Simulation để lấy giá trị. Nếu Simulation báo lại rằng ở thời điểm đó nó đã kết thúc thì controller sẽ tự động dừng
* Một animation controller có thể được cung cấp một giá trị cận trên và cận dưới để animate giữa chúng, và duration của animation

#### Gắn animatables vào animations:

* Truyền một Animation<double> (đối tượng cha mới) vào hàm animate() của Animatables sẽ tạo một subclass mới của Animation hoạt động như là Animatable nhưng hoạt động theo đối tượng cha.

# HƯỚNG DẪN VỀ EXPLICIT ANIMATIONS

* Hướng dẫn tự xây dụng custom explicit animation (hoặc có thể sử dụng các lớp built-in explicit animation như FadeTransition, SizeTransition,… nếu các lớp đó có thể hỗ trợ cho nhu cầu của bạn)

## Các khái niệm và lớp quan trọng:

### Animation<T> (Ta thường sử dụng Animation<double>)

* Một đối tượng thuộc lớp Animation chỉ biết giá trị và trạng thái của chính nó mà không biết những gì đang diễn ra trên màn hình. Nó không biết gì về rendering hay hàm build() của widget.
* Một đối tượng thuộc lớp Animation tạo ra một giá trị kiểu T trong khoảng cho trước (cũng thuộc kiểu T) ở một thời diểm nhất định một cách tuần tự. Các giá trị output đó có thể được tạo ra theo tuyến tính, theo đường cong, theo một hàm step hoặc theo bất kì một cách map giá trị nào đó.
* Animation có thể chạy theo chiều ngược lại hoặc đổi chiều ngay trong khi đang chạy tùy vào cách nó hoạt động
* Một đối tượng lớp Animation có state và giá trị hiện tại của nó có thể được lấy qua .value

### CurvedAnimation:

* CurvedAnimation định ra một quá trình khởi tạo giá trị một cách không tuyến tính của animation.

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

* Các lớp [Curve](https://api.flutter.dev/flutter/animation/Curves-class.html) thường có sẵn hoặc bạn có thể tự định nghĩa bằng các subclass lớp Curve: Ví dụ:

class ShakeCurve extends Curve {

@override

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

}

### AnimationController:

* AnimationController là một đối tượng Animation đặc biệt mà sẽ tạo ra một giá trị mới mỗi frame. Mặc định nó sẽ tạo ra giá trị tuyến tính từ 0.0 đến 1.0 trong một khoảng thời gian nhất định. Ví dụ:

controller = AnimationController(duration: const Duration(seconds: 2), vsync: this);

* AnimationController kế thừa từ Animation<double> nhưng có thêm các phương thức để kiểm soát animation.
* Tác dụng của vsync là để ngăn các animation ngoài màn hình ngốn tài nguyên một cách không cần thiết. Thường thì ta sẽ dùng mixin SingleTickerProviderStateMixin cho các lớp mà extends State<T> sau đó truyền this cho argument vsync.

### Tween:

* Mặc định AnimationController có khoảng giá trị từ 0.0 tới 1.0. Cho nên nếu ta muốn một khoảng giá trị khác hoặc một kiểu giá trị khác, ta có thể dùng Tween<T>. Ví dụ Tween đi từ -200 đến 0:

tween = Tween<double>(begin: -200, end: 0);

* Tween là một đối tượng stateless chỉ lấy 2 giá trị begin và end. Nhiệm vụ của nó chỉ là định nghĩa một cách mapping từ một khoảng input đến một khoảng output khác.
* Tween không kế thừa từ Animation<T> mà là Animatable<T>. Tween có thể cho ra các output với kiểu dữ liệu khác với double, ví dụ như:

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

* Một đối tượng Tween là stateless cho nên nó không chứa bất kì state gì như Animation. Thay vào đó, Tween cung cấp phương thức [evaluate(animation)](https://api.flutter.dev/flutter/animation/Animatable/evaluate.html) để map giá trị hiện tại của một animation thành giá trị theo Tween và giá trị đó được lưu trữ trong phương thức .value của animation.
* Ngoài phương phức evalute(), ta thường dùng animate() của Tween và truyền vào một đối tượng controller. Ví dụ đoạn code sau sẽ tạo một đối tượng Animation<int> tạo ra giá trị integer từ 0 tới 255 trong khoảng thời gian 500ms:
  + AnimationController controller = AnimationController( duration: const Duration(milliseconds: 500), vsync: this);
  + Animation<int> alpha = IntTween(begin: 0, end: 255).animate(controller);

### Animation notifications:

* Như đã giới thiệu ở phần Overview, ta có thể sử dụng 2 phương thức là addListener() và addStatusListener() của một đối tương Animation:
* Thường ta sẽ gọi setState() mỗi khi Listener để cập nhật giá trị của animation và rebuild lại widget (bởi vì về cơ bản animation chỉ tạo ra giá trị với từng frame, ta phải rebuild lại)
* StatusListener sẽ được gọi mỗi khi animation có các trạng thái bắt đầu, kết thúc, đang chạy hoặc đang chạy ngược lại (AnimationStatus).

## Examples:

### Rendering animations:

* Ta đã biết tạo ra một chuỗi giá trị theo khoảng nhất định qua phần trên. Tuy nhiên để áp dụng lên màn hình. Ta cần lưu một đối tượng Animation là một biến của Widget và sử dụng giá trị tạo ra của animation để quyết định nên build widget như thế nào từ giá trị đó.
* Đầu tiên tạo một StatefulWidget và thêm vào State một mixin SingleTickerProviderMixin
* Tạo một animation controller để điều khiển animation. Sau đó ta sẽ tạo một đối tượng Animation bằng Tween.animate() với tham số đầu vào là animation controller. Sau đó ta sẽ thêm listener bằng phương thức addListener và gọi setState() để rebuild lại widget
* Tất cả những thao tác trên nên được để trong hàm initState(). Và cuối cùng gọi dispose() của animation controller trong hàm dispose() của State.
* Mẫu code trong State:

class AnimatedImage extends StatefulWidget {  
 const AnimatedImage({Key? key,  
 }) : super(key: key);  
  
 @override  
 State<StatefulWidget> createState() => AnimatedImageState();  
}  
  
class AnimatedImageState extends State<AnimatedImage> with SingleTickerProviderStateMixin {  
  
 late AnimationController \_controller;  
 late Animation \_animation;  
  
 @override  
 void initState() {  
 \_controller = AnimationController(duration: const Duration(seconds: 1,), vsync: this);  
 \_animation = Tween<double>(begin: 20, end: 300).animate(\_controller)  
 ..addListener(() {  
 setState(() {  
  
 });  
 });  
  
 \_controller.forward();  
  
 super.initState();  
 }  
  
 @override  
 void dispose() {  
 \_controller.dispose();  
 super.dispose();  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return Container(  
 width: \_animation.value,  
 height: \_animation.value,  
 child: const FlutterLogo(),  
 );  
 }  
  
}  
  
class RenderAnimationPage extends StatelessWidget {  
 const RenderAnimationPage({Key? key}) : super(key: key);  
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text("Rendering Animation"),  
 ),  
 body: SafeArea(  
 child: Center(  
 child: AnimatedImage(),  
 ),  
 ),  
 );  
 }  
}

### Đơn giản hóa với AnimatedWidget:

* AnimatedWidget giúp ta tách code chính của widget và code animation ra. AnimatedWidget không cần ta phải tạo một lớp State cho widget ta cần animate để giữ đối tượng Animation như trên mà để widget cha lo liệu việc tạo đối tượng Animation và chỉ cần truyền vào constructor của AnimatedWidget. Ta cũng không cần gọi addListener. Tuy nhiên trong lớp extends AnimatedWidget ta cần cast listenable (1 member có sẵn của lớp AnimatedWidget) thành đối tượng Animation<T> cần thiết để lấy giá trị .value
* Như ví dụ trên thì AnimatedImageState sẽ trở thành AnimatedImage extends AnimatedWidget
* class AnimatedImage extends AnimatedWidget {  
   const AnimatedImage({Key? key, required Animation<double> animation})  
   : super(key: key, listenable: animation);  
    
   @override  
   Widget build(BuildContext context) {  
   final animation = listenable as Animation<double>;  
    
   return Container(  
   width: animation.value,  
   height: animation.value,  
   child: const FlutterLogo(),  
   );  
   }  
  }
* và các member như \_controller, \_animation, sẽ được đặt ở cha của AnimatedImage (widget mà sử dụng AnimatedImage trong build)
* class AnimatedWidgetPage extends StatefulWidget {  
   const AnimatedWidgetPage({Key? key}) : super(key: key);  
    
   @override  
   State<StatefulWidget> createState() => AnimatedWidgetPageState();  
  }  
    
  class AnimatedWidgetPageState extends State<AnimatedWidgetPage> with SingleTickerProviderStateMixin {  
    
   late AnimationController \_controller;  
   late Animation<double> \_animation;  
    
   @override  
   void initState() {  
   \_controller = AnimationController(duration: const Duration(seconds: 1,), vsync: this);  
   \_animation = Tween<double>(begin: 20, end: 300).animate(\_controller);  
    
   \_controller.forward();  
    
   super.initState();  
   }  
    
   @override  
   void dispose() {  
   \_controller.dispose();  
   super.dispose();  
   }  
    
   @override  
   Widget build(BuildContext context) {  
   return Scaffold(  
   appBar: AppBar(  
   title: Text("AnimatedWidget"),  
   ),  
   body: SafeArea(  
   child: Center(  
   child: AnimatedImage(animation: \_animation,),  
   ),  
   ),  
   );  
   }  
  }

### Theo dõi status của animation:

* Để theo dõi status của animation, sử dụng phương thức addStatusListener() của lớp Animation. Xem [AnimationStatus](https://api.flutter.dev/flutter/animation/AnimationStatus.html) để biết các trạng thái có thể có của animation
* Ta sẽ thêm code dưới vào initState()
* \_animation.addStatusListener((status) {  
   if (status == AnimationStatus.dismissed) {  
   \_controller.forward();  
   } else if (status == AnimationStatus.completed) {  
   \_controller.reverse();  
   }  
  });
* Lúc này, animation sẽ tự động chạy mỗi khi status là dismissed hoặc completed

### Sử dụng AnimatedBuilder:

* AnimatedBuilder chỉ biết cách render sự chuyển đổi trong animation chứ không biết cách render widget hay quản lý đối tượng Animation.
* AnimatedBuilder và AnimatedWidget được sử dụng khi ta không muốn rebuild lại toàn bộ widget mà chỉ là 1 phần cần animation trong đó. Điểm khác biệt là AnimatedWidget tách một phần widget cần animation ra làm 1 widget riêng lẻ còn AnimatedBuilder có thể được sử dụng trong chính phương thức build() của widget
* Như ví dụ ở trên, ta có thể thêm vào lớp:
* class GrowTransition extends StatelessWidget {  
   const GrowTransition({required this.child, required this.animation, Key? key})  
   : super(key: key);  
    
   final Widget child;  
   final Animation<double> animation;  
    
   @override  
   Widget build(BuildContext context) {  
   return Center(  
   child: AnimatedBuilder(  
   animation: animation,  
   builder: (context, child) {  
   return SizedBox(  
   height: animation.value,  
   width: animation.value,  
   child: child,  
   );  
   },  
   child: child,  
   ),  
   );  
   }  
  }
* Và thay vì
* @override  
  Widget build(BuildContext context) {  
   return Scaffold(  
   appBar: AppBar(  
   title: Text("AnimatedWidget"),  
   ),  
   body: SafeArea(  
   child: Center(  
   child: AnimatedImage(animation: \_animation,),  
   ),  
   ),  
   );  
  }

Ta có thể thay bằng

* @override  
  Widget build(BuildContext context) {  
   return Scaffold(  
   appBar: AppBar(  
   title: Text("Animated Builder"),  
   ),  
   body: SafeArea(  
   child: Center(  
   child: GrowTransition(animation: \_animation, child: const FlutterLogo(),),  
   ),  
   ),  
   );  
  }
* Ở trong hàm build của GrowTransition, ta có thể thêm nhiều widget khác, chỉ cần có AnimatedBuilder thì trong quá trình animation, chỉ phần builder (hay child) của AnimatedBuilder được rebuild

### Áp dụng đồng thời nhiều animations:

* Ở các ví dụ trên, ta chỉ áp dụng một animation là thay đổi kích cỡ của hình ảnh. Sau đây, ta sẽ áp dụng thêm một loại animation về độ mờ:
* class AnimatedImage extends AnimatedWidget {  
    
   static final *\_opacityTween* = Tween<double>(begin: 0.1, end: 1);  
   static final *\_sizeTween* = Tween<double>(begin: 0, end: 300);  
    
   const AnimatedImage2({Key? key, required Animation<double> animation})  
   : super(key: key, listenable: animation);  
    
   @override  
   Widget build(BuildContext context) {  
   final animation = listenable as Animation<double>;  
    
   return Opacity(  
   opacity: *\_opacityTween*.evaluate(animation),  
   child: Container(  
   width: *\_sizeTween*.evaluate(animation),  
   height: *\_sizeTween*.evaluate(animation),  
   child: const FlutterLogo(),  
   ),  
   );  
   }  
  }
* Về cơ bản, các Tween chỉ map giá trị của Animation ban đầu thành một giá trị khác. Cho nên ta có thể áp dụng nhiều loại Tween khác nhau cho cùng một đối tượng Animation và vì thế khiến cho nhiều animation chạy đồng thời cùng 1 lúc.

# Implicit animations

* Đây là thư viện hỗ trợ animation ngầm hiểu (tức là mặc định có sẵn trong flutter cơ bản: flutter/animation.dart)
* Gọi chung là animation ngầm hoặc widget animation ngầm. Tên gọi được lấy từ class ImplicitlyAnimatedWidget
* Gồm các animation ứng với các class:
  + TweenAnimationBuilder thay đổi một Tween từ 1 giá trị ban đầu đến 1 giá trị cố định:
    - Thay đổi được các giá trị animated property như color, rect, double…
    - Giá trị tween thay đổi thì tạo ra animation thay đổi property của đối tượng
  + AnimatedAlign - ứng với Align class.
    - Thay đổi alignment của đối tượng có animation (di chuyển đối tượng, fade in – fade out….)
  + AnimatedContainer - ứng với  Container.
    - Thay đổi các property của container bằng hiệu ứng.
  + AnimatedDefaultTextStyle - ứng với DefaultTextStyle.
    - Thay đổi style của Text.
  + AnimatedScale - ứng với  Transform.scale.
    - Thay đổi kích thước của đối tượng trong thời gian cho trước
  + AnimatedRotation - ứng với  Transform.rotate.
    - Xoay đối tượng trong thời gian cho trước
  + AnimatedSlide ứng với vị trí của widget relative so với vị trí bình thường
    - Di chuyển đối tượng theo dạng trượt
  + AnimatedOpacity - ứng với  Opacity.
    - Thay đổi độ trong suốt của đối tượng
  + AnimatedPadding - ứng với  Padding.
    - Thay đổi padding của Padding
  + AnimatedPhysicalModel - ứng với  PhysicalModel.
    - Thay đổi các property: borderRadius, Elevation, Color, Curves…
  + AnimatedPositioned - ứng với  Positioned.
    - Thay đổi vị trí của đối tượng
  + AnimatedPositionedDirectional - ứng với  PositionedDirectional.
    - Tương tự positioned nhưng thực hiện với class PositionedDirectional
    - Chỉ dùng được cho child của stack
  + AnimatedTheme - ứng với  Theme class.
    - Thay đổi property của theme như màu, …
  + AnimatedCrossFade
    - Dùng để làm mờ dần 2 child nhất định và tạo animation cho kích thước của child
  + AnimatedSize
    - Dùng trong thay đổi kích thước trong thời gian cho trước
  + AnimatedSwitcher
    - Chuyển từ widget này sang widget khác.

# Hero Animation

* Hero Animation là một widget giúp chúng ta tạo các hiệu ứng khi chuyển màn.
* 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.
* Thường có 2 loại hero animation hay được sử dụng:
  + Standard hero animations:
    - Làm 1 hero (hình ảnh, icon..) bay từ 1route tới 1 route khác (có thể hiểu đơn giản là từ một trang này tới 1 trang khác).
    - Thường hero nằm ở vị trí khac 1nhau và khác kích thước
  + Radial hero aniimations:
    - Tương tự như standard nhưng trong quá trình “bay” thì shape của hero sẽ thay đổi từ tròn sang chữ nhật
* Cấu trúc cơ bản:
  + Có 2 Hero widget:
    - Widget ở route ban đầu, route gốc (tạm gọi là source route)
    - Widget ở route sau khi diễn ra animation, route đích ( des route)
* Cấu trúc code:

1. Khai báo source hero widget, các tính chất của widget, khai báo một “identifying tag”. Widget này được khai báo và hiển thị ở source route
2. Khai báo ending hero Widget. Nó phải được khai báo tag giống như source hero widget (thường là đối tượng đại diện cho dữ liệu cơ bản). Để kết quả đẹp nhất thì nên khai báo widget trees tương tự nhau
3. Tạo des route.
4. Trigger animation bằng cách dùng navigator stack. Navigator push và pop sẽ trigger hero animation với từng cặp hero widget matching tags với nhau ở source và des.

* Cơ chế hoạt động:
  + Graphical user interface, application

    Description automatically generatedTrước khi animation diễn ra:
  + Trong lúc diễn ra animation:

Graphical user interface, application, Word

Description automatically generated

* + Sau khi animation kết thúc

Graphical user interface, application, Word

Description automatically generated

* Class cần thiết:
  + Hero: widget sẽ “bay” từ source tới des. Phải chung tag
  + Inkwell: quản lí onTap() để push và pop.
  + Navigator: dùng chuyển giữa các route
  + Route: Quản lí screen or page.
* Tạo một standard hero animation:
  + Bước 1: tạo widget trong đó có dùng Hero widget
  + Bước 2: tạo trang source route, trang này có sử dụng widget vừa tạo ở trên
  + Bước 3: tạo trang des route, có sử dụng widget vừa tạo ở trên, lưu ý tag phải trùng với widget của source route
  + Bước 4: Dùng navigator push và pop để chuyển qua lại giữa 2 route, hero animation sẽ tự match với nhau
* Tạo một radial hero animation:
  + Bước 1: Tạo radial expansion class để quy định cách flip hero trong lúc diễn ra animation.
  + Bước 2: Tạo hero animation được bọc trong radial expansion vừa tạo.
  + Bước 3: Tạo trang source route, trang này có sử dụng widget vừa tạo ở trên
  + Bước 4: Tạo trang des route, có sử dụng widget vừa tạo ở trên, lưu ý tag phải trùng với widget của source route
  + Bước 5: Dùng navigator push và pop để chuyển qua lại giữa 2 route, hero animation sẽ tự match với nhau

# Staggered Animation

* Là một chuỗi các animation được thực hiện theo một trình tự. Chúng có thể xảy ra một cách liên tục có hoặc không có độ trễ kèm theo
* Cấu trúc cơ bản của Staggered Animation:
  + Tất cả các animation được điều khiên bằng một AnimationController
  + Bất kể animation kéo dài bao lâu trong thực tế, độ dài của nó khi khai báo ở controller phải là giá trị trong khoảng [0.0, 1.0]
  + Thời gian xảy ra của một animation được khai báo trong controller theo một biến dạng **Interval**
  + Mỗi khi tới thời gian diễn ra của một Interval, một biến Twen sẽ được tạo ra. Biến Twen này được sử dụng để tạo thành một Animation để AnimationController quản lý
* Ví dụ:

borderRadius = BorderRadiusTween(

begin: BorderRadius.circular(4.0),

end: BorderRadius.circular(75.0),

).animate(

CurvedAnimation(

parent: controller,

curve: Interval(

0.375, 0.500,

curve: Curves.ease,

),

),

),

* “begin”: Trạng thái ban đầu của Animation
* “end”: Trạng thái kết thúc của Animation
* “parent”: Controller điều khiển hoạt động của các chuỗi Animation
* “curve”: Thời điểm bắt đầu và kết thúc của Animation biểu diễn qua biến loại **Interval**

# REFERENCES

[Flutter Animations](https://docs.flutter.dev/development/ui/animations)