Họ và Tên: Nguyễn Trần Trường Vũ

Mã Số: 0850080113

Lớp: 08\_ĐHCNPM

**Bài lí thuyết buổi 7**

1. **AnimatedWidget**

\_ AnimatedWidget là lớp thường được sử dụng với các đối tượng Animation, đó là các đối tượng có kiểu Listenable, nhưng nó cũng có thể được sử dụng với bất kỳ đối tượng Listenable nào, bao gồm ChangeNotifier và ValueNotifier.

\_ AnimatedWidget đặc biệt hữu ích cho các widget khác Stateless (không có trạng thái). Để sử dụng AnimatedWidget, bạn cần kế thừa nó và triển khai hàm build.

\_ Như đã đề cập, AnimatedWidget chủ yếu được sử dụng trong ngữ cảnh của Flutter, và nó giúp tạo các widget có khả năng thay đổi dựa trên các giá trị hoạt động (animation) hoặc các sự kiện phát ra từ các đối tượng Listenable khác.

\_ Khi bạn kế thừa từ AnimatedWidget và triển khai hàm build, Flutter sẽ tự động cập nhật các giá trị của đối tượng Listenable (như Animation) và gọi hàm build mỗi khi giá trị của Listenable thay đổi. Điều này giúp bạn xây dựng các widget động và hoạt động linh hoạt dễ dàng hơn mà không cần quản lý trạng thái một cách rườm rà.

\_ Có một số animated widgets được tích hợp sẵn trong framework của Flutter. Thông thường, chúng có tên dạng FooTransition, trong đó Foo là tên của phiên bản không có animation của widget đó. Các lớp con của class này không nên bị nhầm lẫn với các lớp con của ImplicitlyAnimatedWidget (xem phần trên), các lớp này thường có tên dạng AnimatedFoo. Dưới đây là một số animated widgets phổ biến mà bạn có thể sử dụng:

1. SlideTransition: Hiệu ứng chuyển động theo hướng từ vị trí ban đầu đến vị trí cuối của một widget.
2. ScaleTransition: Hiệu ứng thu phóng của widget từ kích thước ban đầu đến kích thước cuối.
3. RotationTransition: Hiệu ứng xoay một widget từ góc ban đầu đến góc cuối.
4. FadeTransition: Hiệu ứng làm mờ dần một widget từ không hiển thị (độ mờ 0) đến hiển thị hoàn toàn (độ mờ 1).
5. SizeTransition: Hiệu ứng thay đổi kích thước của widget từ kích thước ban đầu đến kích thước cuối.
6. DecoratedBoxTransition: Hiệu ứng thay đổi các thuộc tính của hộp được trang trí (DecoratedBox) như màu sắc, viền, v.v.
7. PositionedTransition: Hiệu ứng di chuyển và thay đổi kích thước của một widget theo một loạt các vị trí được xác định trước.
8. AlignTransition: Hiệu ứng chuyển động và thay đổi kích thước của một widget theo một loạt các vị trí được xác định bằng Align.
9. RelativePositionedTransition: Hiệu ứng di chuyển và thay đổi kích thước của một widget theo một loạt các vị trí tương đối (theo đơn vị phần trăm) đối với container cha.
10. ScaleTransition: Hiệu ứng scale một widget từ kích thước ban đầu đến kích thước cuối.

Ví dụ:

Đoạn mã ở đường dẫn này định nghĩa một widget có tên Spinner, được sử dụng để quay một hình vuông màu xanh lá liên tục. Widget này được xây dựng với sử dụng AnimatedWidget.

https://api.flutter.dev/flutter/widgets/AnimatedWidget-class.html



\_ Tóm lại, AnimatedWidget là một công cụ mạnh mẽ trong Flutter giúp bạn xây dựng các widget có khả năng thay đổi theo thời gian mà không cần phải viết nhiều mã phức tạp để quản lý trạng thái. Nó cho phép bạn tập trung vào việc triển khai hàm build để cung cấp giao diện người dùng mà không phải lo lắng về việc cập nhật dữ liệu hoặc trạng thái của widget

1. **AnimatedBuilder**

\_ AnimatedBuilder widget là một widget rất hữu ích khi xây dựng animation. Nó hiệu quả hơn việc gọi setState() mỗi khi có sự thay đổi giá trị của animation.

\_ AnimatedBuilder trong Flutter là một widget mạnh mẽ cho phép bạn tạo các hoạt ảnh tùy chỉnh bằng cách tách logic hoạt ảnh khỏi đại diện hình ảnh của widget. Nó rất hữu ích khi bạn cần kiểm soát hơn về cách xây dựng và hiển thị hoạt ảnh.

\_ Lớp AnimatedBuilder trong Flutter là một widget kế thừa từ StatefulWidget. Nó cho phép bạn tạo các hoạt ảnh tùy chỉnh bằng cách tách logic hoạt ảnh khỏi đại diện hình ảnh của widget. Mục đích chính của AnimatedBuilder là tối ưu hóa quá trình xây dựng lại của widget bằng cách chỉ xây dựng lại các phần thay đổi khi animation value thay đổi, giúp giảm tải và tăng hiệu suất cho ứng dụng.

\_ Lớp AnimatedBuilder chấp nhận một đối tượng Animation và một hàm gọi lại (builder callback) làm tham số của nó. Hàm gọi lại này sẽ được gọi mỗi khi giá trị hoạt ảnh thay đổi và nó trả về cây widget đại diện cho hình ảnh của widget được hoạt hóa. Lợi ích chính của việc sử dụng AnimatedBuilder là nó tối ưu hóa quá trình xây dựng lại bằng cách xây dựng lại chỉ cây widget được trả về bởi builder, thay vì xây dựng lại toàn bộ cây widget cha.

Cú pháp cơ bản của việc sử dụng AnimatedBuilder như sau:

AnimatedBuilder(

animation: animation, // Một đối tượng Animation (ví dụ: AnimationController, Tween, v.v.)

builder: (BuildContext context, Widget? child) {

// Trả về cây widget đại diện cho hình ảnh của widget được hoạt hóa.

// Cây widget này sẽ được xây dựng lại mỗi khi giá trị hoạt ảnh thay đổi.

// Bạn có thể sử dụng giá trị hoạt ảnh hiện tại để tùy chỉnh đại diện hình ảnh.

return YourCustomWidget(...); }, )

Dưới đây là 1 số cách sử dụng AnimatedBuilder:

1. AnimatedOpacity: Sử dụng AnimatedBuilder để tạo hiệu ứng mờ dần (opacity) của một widget.
2. AnimatedPositioned: Sử dụng AnimatedBuilder để tạo hiệu ứng di chuyển của một widget (thay đổi vị trí).
3. AnimatedContainer: Sử dụng AnimatedBuilder để tạo hiệu ứng thay đổi kích thước và thuộc tính của một Container.
4. AnimatedCrossFade: Sử dụng AnimatedBuilder để tạo hiệu ứng chuyển đổi giữa hai widget (phủ nhận và hiển thị).
5. AnimatedDefaultTextStyle: Sử dụng AnimatedBuilder để tạo hiệu ứng thay đổi phong cách văn bản của một widget.
6. AnimatedPadding: Sử dụng AnimatedBuilder để tạo hiệu ứng thay đổi lề của một widget.
7. AnimatedAlign: Sử dụng AnimatedBuilder để tạo hiệu ứng thay đổi vị trí cân chỉnh của một widget.
8. AnimatedPhysicalModel: Sử dụng AnimatedBuilder để tạo hiệu ứng thay đổi các thuộc tính vật lý của một widget.
9. **AnimatedController**

\_ Lớp AnimationController trong Flutter là một lớp được sử dụng để quản lý và điều khiển một hoạt ảnh (animation). Nó là một lớp con của lớp Animation<double>.

\_ AnimationController được sử dụng để tạo và kiểm soát giá trị hoạt ảnh (animation value), mà là một con số từ 0.0 đến 1.0 (hoặc từ -1.0 đến 1.0 tùy thuộc vào cài đặt). Giá trị hoạt ảnh có thể biến đổi theo thời gian và được sử dụng để điều khiển hiệu ứng hoạt ảnh của các widget trong Flutter.

final AnimationController controller = AnimationController(

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

\_ Bạn có thể kiểm soát giá trị hoạt ảnh bằng cách sử dụng các phương thức của AnimationController như:

1. forward(): Bắt đầu hoạt ảnh và tăng giá trị hoạt ảnh từ 0.0 đến 1.0.
2. reverse(): Bắt đầu hoạt ảnh ngược và giảm giá trị hoạt ảnh từ 1.0 về 0.0.
3. repeat(): Lặp lại hoạt ảnh liên tục, đảo ngược giá trị hoạt ảnh khi đạt đến 1.0 hoặc 0.0.
4. stop(): Dừng hoạt ảnh và giữ giá trị hoạt ảnh tại giá trị hiện tại.

final AnimationController controller = AnimationController(

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

\_ AnimationController cũng cho phép bạn thiết lập thời gian hoạt ảnh (duration), phạm vi giá trị hoạt ảnh (lowerBound và upperBound), và các thuộc tính khác để tùy chỉnh cách hoạt ảnh được điều khiển.

\_ Một cách phổ biến để sử dụng AnimationController là cùng với các widget như AnimatedBuilder, Tween, và các widget animated khác trong Flutter để tạo các hiệu ứng hoạt ảnh phong phú.

\_ Trong hàm dispose, chúng ta giải phóng tài nguyên của AnimationController khi widget bị hủy.

\_ Trong hàm build, chúng ta sử dụng AnimatedBuilder để tạo hiệu ứng hoạt ảnh cho widget của chúng ta. Trong builder callback, chúng ta truyền AnimationController \_controller làm đối số animation và sử dụng giá trị animation hiện tại \_controller.value để tạo hiệu ứng cho widget được tạo bởi YourAnimatedWidget (ví dụ: màu sắc, kích thước, v.v.).

1. **Curved Animation**

\_ Trong Flutter, "Curved Animation" là một loại hoạt ảnh được sử dụng để điều chỉnh tốc độ của hoạt ảnh dựa trên một đường cong (curve) nhất định. Điều này giúp tạo ra các hiệu ứng hoạt ảnh mượt mà và tự nhiên hơn.

\_ Khi sử dụng Curved Animation, bạn cần cung cấp một đối tượng Animation Controller (hoặc bất kỳ đối tượng Animation nào) và một curve (đường cong) cụ thể để áp dụng cho hoạt ảnh. Flutter cung cấp sẵn một số loại curve phổ biến như linear, ease-in, ease-out, ease-in-out, và nhiều curve tùy chỉnh khác.

\_ Để sử dụng Curved Animation, bạn cần sử dụng lớp CurvedAnimation trong Flutter. Cú pháp cơ bản của CurvedAnimation như sau:

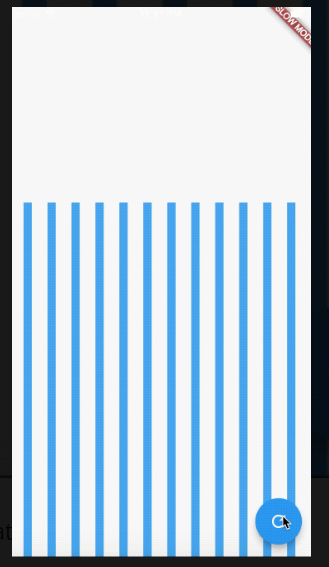
CurvedAnimation(

parent: animation, // Đối tượng Animation (ví dụ: AnimationController, Tween, v.v.)

curve: Curves.easeInOut, // Loại curve bạn muốn áp dụng (ví dụ: Curves.easeInOut, Curves.linear, v.v.)

)

Sau khi có CurvedAnimation, bạn có thể sử dụng nó để tùy chỉnh tốc độ của hoạt ảnh bằng cách sử dụng giá trị giữa 0.0 và 1.0 mà nó cung cấp. Giá trị 0.0 tương ứng với khởi đầu hoạt ảnh và giá trị 1.0 tương ứng với kết thúc hoạt ảnh theo đường curve đã xác định.



Khởi tạo hay định nghĩa một curvedAnimation như sau:

final CurvedAnimation curve =

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

Bạn có thể tự tạo một Curve riêng cho mình dựa vào lớp [Curves](https://docs.flutter.io/flutter/animation/Curves-class.html)

Ví dụ

class ShakeCurve extends Curve {

@override

double transform(double t) {

return math.sin(t \* math.PI \* 2);

}

}

CurvedAnimation và AnimationController đều là kiểu Animation<double>, vì thế chúng ta có thể sử dụng thay thế cho nhau. CurvedAnimation sẽ chứa đối tượng nó đang thay đổi, bạn không cần một lớp con AnimationController để thực hiện một curve.

1. **Hero**

\_ Trong Flutter, "Hero" là một widget được sử dụng để tạo hiệu ứng chuyển đổi mượt mà giữa hai màn hình hoặc hai vị trí khác nhau trong cùng một ứng dụng. Hero widget thường được sử dụng để tạo các hiệu ứng chuyển đổi hấp dẫn và sinh động khi người dùng điều hướng giữa các trang hoặc khi các phần tử di chuyển giữa các vị trí khác nhau trong một màn hình.

\_ Để sử dụng Hero, bạn cần xác định hai widget có cùng thuộc tính "tag" (nhãn) cho Hero. Một widget sẽ xuất hiện trên một màn hình và widget còn lại sẽ xuất hiện trên màn hình khác hoặc ở vị trí khác trên cùng một màn hình. Khi người dùng thực hiện hành động để chuyển đổi giữa các màn hình hoặc di chuyển các phần tử, Hero sẽ tạo ra hiệu ứng chuyển đổi mượt mà và tự động thay đổi kích thước, vị trí và màu sắc để tạo ra hiệu ứng chuyển đổi hấp dẫn.

\_ Để sử dụng Hero, bạn cần làm như sau:

1. Xác định hai widget với cùng một thuộc tính "tag" cho Hero.
2. Đặt một widget Hero trong màn hình nguồn và một widget Hero trong màn hình đích hoặc vị trí khác trên cùng một màn hình.
3. Khi thực hiện hành động chuyển đổi, Hero sẽ tạo ra hiệu ứng chuyển đổi tự động và mượt mà giữa hai widget có cùng tag.

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(

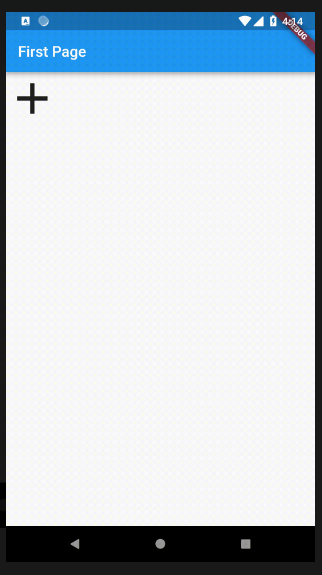
Icons.add,

size: 150.0,

),

),

Và đây là kết quả chúng ta thu được



1. **Opacity**

\_ Là Một widget làm cho con của nó bị mờ một phần.

\_ Lớp này vẽ con của nó vào một bộ đệm trung gian và sau đó kết hợp con vào cảnh một cách mờ một phần.

\_ Với các giá trị độ mờ khác 0.0 và 1.0, lớp này tương đối tốn kém vì nó yêu cầu vẽ con vào bộ đệm trung gian. Với giá trị 0.0, con không được vẽ vào bất kỳ bộ đệm nào. Với giá trị 1.0, con được vẽ ngay lập tức mà không cần bộ đệm trung gian.

\_ Sự hiện diện của bộ đệm trung gian, mặc định có nền trong suốt, có thể làm cho một số widget con hoạt động khác nhau. Ví dụ, một widget con BackdropFilter sẽ chỉ có thể áp dụng bộ lọc của nó vào nội dung giữa widget này và widget backdrop và có thể yêu cầu điều chỉnh thuộc tính BackdropFilter.blendMode để tạo ra kết quả mong muốn.

\_Ví dụ sau hiển thị một đoạn văn bản khi trường \_visible là true, và ẩn nó khi trường \_visible là false:

Opacity(

opacity: \_visible ? 1.0 : 0.0,

child: **const** Text("Bây giờ bạn thấy được tôi, nhưng giờ thì không!"),

)

\_ Điều này hiệu quả hơn so với việc thêm và xóa tiện ích con khỏi cây theo yêu cầu.

\_ Cân nhắc hiệu suất cho hoạt ảnh độ mờ (opacity animation):

\_ Khi thực hiện hoạt ảnh độ mờ (opacity animation) trực tiếp trên một widget Opacity, điều này làm cho widget (và có thể cả cây con của nó) phải được xây dựng lại mỗi khung hình, điều này không hiệu quả.

\_ Thay vào đó, hãy xem xét sử dụng một trong những widget thay thế này:

\_ Có hai widget thay thế mà bạn nên xem xét sử dụng để tăng hiệu suất cho hoạt ảnh độ mờ (opacity animation):

1. AnimatedOpacity: Widget này sử dụng một hoạt ảnh (animation) bên trong để thực hiện hoạt ảnh độ mờ một cách hiệu quả. Nó giúp giảm thiểu số lần xây dựng lại của cây và tăng hiệu suất của ứng dụng. Bạn chỉ cần cung cấp animation và AnimatedOpacity sẽ tự động xử lý việc cập nhật độ mờ của widget một cách thông minh.
2. FadeTransition: Widget này sử dụng animation mà bạn cung cấp để thực hiện hoạt ảnh độ mờ một cách hiệu quả. Nó cho phép bạn tự do tùy chỉnh animation và cách widget phản hồi khi độ mờ thay đổi. Tương tự như AnimatedOpacity, FadeTransition giúp tối ưu hoá hiệu suất của ứng dụng bằng cách tránh xây dựng lại cây quá nhiều lần.

\_ Ví dụ, Container(color: Color.fromRGBO(255, 0, 0, 0.5)) sẽ nhanh hơn rất nhiều so với Opacity(opacity: 0.5, child: Container(color: Colors.red)).

\_ Việc sử dụng Opacity widget khi chỉ có một Image hoặc Color đơn lẻ và muốn áp dụng độ mờ có thể là không cần thiết và tốn kém về hiệu suất. Khi chúng ta sử dụng Opacity, widget và tất cả các con của nó phải xây dựng lại để áp dụng độ mờ, điều này có thể làm giảm hiệu suất của ứng dụng.

\_ Thay vào đó, nếu bạn chỉ cần một hình ảnh hoặc màu duy nhất với độ mờ, hãy sử dụng trực tiếp hình ảnh hoặc màu đó với thuộc tính opacity trực tiếp (sử dụng Color.fromRGBO cho màu) để tăng hiệu suất và giảm tải cho ứng dụng.

\_ Ví dụ sau vẽ một Hình ảnh có độ mờ 0,5 mà không sử dụng Độ mờ Opacity:

Image.network(

'https://raw.githubusercontent.com/flutter/assets-for-api-docs/master/packages/diagrams/assets/blend\_mode\_destination.jpeg',

color: **const** Color.fromRGBO(255, 255, 255, 0.5),

colorBlendMode: BlendMode.modulate

)

\_ Việc vẽ trực tiếp một Hình ảnh (Image) hoặc Màu sử dụng độ mờ (opacity) sẽ nhanh hơn so với việc sử dụng Opacity widget ở phía trên bởi vì Opacity có thể áp dụng độ mờ lên một nhóm các widget và do đó, nó sẽ sử dụng một bộ đệm (offscreen buffer) tốn kém về hiệu suất.

\_ Việc vẽ nội dung vào bộ đệm ngoài màn hình có thể gây ra các chuyển đổi mục tiêu vẽ (render target switches), và việc chuyển đổi này đặc biệt chậm trên các GPU cũ. Điều này làm tăng tải cho ứng dụng và làm giảm hiệu suất.