Animations tutorial

Contents

What you'll learn

- How to use the fundamental classes from the animation library to add animation to a widget.
- When to use `AnimatedWidget` vs. `AnimatedBuilder`.

This tutorial shows you how to build explicit animations in Flutter. After introducing some of the essential concepts, classes, and methods in the animation library, it walks you through 5 animation examples. The examples build on each other, introducing you t different aspects of the animation library.

The Flutter SDK also provides transition animations, such as FadeTransition, SizeTransition, and SlideTransition. These simple animations are triggered by setting a beginning and ending point. They are simpler to implement than explicit animations, which a described here.

Essential animation concepts and classes

What's the point?

- Animation, a core class in Flutter's animation library, interpolates the values used to guide an animation.
- An `Animation` object knows the current state of an animation (for example, whether it's started, stopped, or moving forward or in reverse), but doesn't know anything about what appears onscreen.
- An AnimationController manages the `Animation`.
- A CurvedAnimation defines progression as a non-linear curve.
- A Tween interpolates between the range of data as used by the object being animated. For example, a `Tween` might define an interpolation from red to blue, or from 0 to 255.
- Use Listeners and StatusListeners to monitor animation state changes.

The animation system in Flutter is based on typed Animation objects. Widgets can either incorporate these animations in their bu functions directly by reading their current value and listening to their state changes or they can use the animations as the basis o more elaborate animations that they pass along to other widgets.

Animation<double>

In Flutter, an Animation object knows nothing about what is onscreen. An `Animation` is an abstract class that understands its cur value and its state (completed or dismissed). One of the more commonly used animation types is `Animation<double>`.

An `Animation` object sequentially generates interpolated numbers between two values over a certain duration. The output of an `Animation` object might be linear, a curve, a step function, or any other mapping you can devise. Depending on how the `Animatio` object is controlled, it could run in reverse, or even switch directions in the middle.

Animations can also interpolate types other than double, such as `Animation<Color>` or `Animation<Size>`.

An `Animation` object has state. Its current value is always available in the `.value` member.

An `Animation` object knows nothing about rendering or `build()` functions.

CurvedAnimation

A CurvedAnimation defines the animation's progress as a non-linear curve.

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

Note: The Curves class defines many commonly used curves, or you can create your own. For example:

```
import 'dart:math';

class ShakeCurve extends Curve {
  @override
  double transform(double t) => sin(t * pi * 2);
}
```

Browse the Curves documentation for a complete listing (with visual previews) of the `Curves` constants that ship with Flutter.

`CurvedAnimation` and `AnimationController` (described in the next section) are both of type `Animation<double>`, so you can pas them interchangeably. The CurvedAnimation wraps the object it's modifying—you don't subclass `AnimationController` to implen a curve.

AnimationController

AnimationController is a special `Animation` object that generates a new value whenever the hardware is ready for a new frame. B default, an `AnimationController` linearly produces the numbers from 0.0 to 1.0 during a given duration. For example, this code creates an Animation object, but does not start it running:

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

`AnimationController` derives from `Animation<double>`, so it can be used wherever an `Animation` object is needed. However, th `AnimationController` has additional methods to control the animation. For example, you start an animation with the `.forward()` method. The generation of numbers is tied to the screen refresh, so typically 60 numbers are generated per second. After each number is generated, each Animation object calls the attached Listener objects. To create a custom display list for each child, se RepaintBoundary.

When creating an `AnimationController`, you pass it a `vsync` argument. The presence of `vsync` prevents offscreen animations fr consuming unnecessary resources. You can use your stateful object as the vsync by adding `SingleTickerProviderStateMixin` the class definition. You can see an example of this in animate1 on GitHub.

Note: In some cases, a position might exceed the `AnimationController`'s 0.0-1.0 range. For example, the `fling()` functio allows you to provide velocity, force, and position (via the Force object). The position can be anything and so can be outside o the 0.0 to 1.0 range.

A `CurvedAnimation` can also exceed the 0.0 to 1.0 range, even if the `AnimationController` doesn't. Depending on the curve selected, the output of the `CurvedAnimation` can have a wider range than the input. For example, elastic curves such as Curves.elasticIn will significantly overshoot or undershoot the default range.

Tween

By default, the `AnimationController` object ranges from 0.0 to 1.0. If you need a different range or a different data type, you can a Tween to configure an animation to interpolate to a different range or data type. For example, the following `Tween` goes from -2(to 0.0:

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

A `Tween` is a stateless object that takes only `begin` and `end`. The sole job of a `Tween` is to define a mapping from an input range to output range. The input range is commonly 0.0 to 1.0, but that's not a requirement.

A `Tween` inherits from `Animatable<T>`, not from `Animation<T>`. An Animatable, like Animation, doesn't have to output double. For example, `ColorTween` specifies a progression between two colors.

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

A `Tween` object does not store any state. Instead, it provides the `evaluate(Animation<double> animation)` method that applies mapping function to the current value of the animation. The current value of the `Animation` object can be found in the `.value` method. The evaluate function also performs some housekeeping, such as ensuring that begin and end are returned when the animation values are 0.0 and 1.0, respectively.

Tween.animate

To use a `Tween` object, call `animate()` on the `Tween`, passing in the controller object. For example, the following code generates th integer values from 0 to 255 over the course of 500 ms.

```
AnimationController controller = AnimationController(
    duration: const Duration(milliseconds: 500), vsync: this);
Animation<int> alpha = IntTween(begin: 0, end: 255).animate(controller);
```

Note: The `animate()` method returns an Animation, not an Animatable.

The following example shows a controller, a curve, and a `Tween`:

```
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);
```

Animation notifications

An Animation object can have `Listener`s and `StatusListener`s, defined with `addListener()` and `addStatusListener()`. A `Listener` is called whenever the value of the animation changes. The most common behavior of a `Listener` is to call `setState(` cause a rebuild. A `StatusListener` is called when an animation begins, ends, moves forward, or moves reverse, as defined by `AnimationStatus`. The next section has an example of the `addListener()` method, and Monitoring the progress of the animation shows an example of `addStatusListener()`.

Animation examples

This section walks you through 5 animation examples. Each section provides a link to the source code for that example.

Rendering animations

What's the point?

- How to add basic animation to a widget using `addListener()` and `setState()`.
- Every time the Animation generates a new number, the `addListener()` function calls `setState()`.
- How to define an `AnimatedController` with the required `vsync` parameter.
- Understanding the "`..`" syntax in "`..addListener`", also known as Dart's *cascade notation*.
- To make a class private, start its name with an underscore (`_`).

So far you've learned how to generate a sequence of numbers over time. Nothing has been rendered to the screen. To render with `Animation` object, store the `Animation` object as a member of your widget, then use its value to decide how to draw.

Consider the following app that draws the Flutter logo without animation:

```
import 'package:flutter/material.dart';

void main() => runApp(LogoApp());

class LogoApp extends StatefulWidget {
  _LogoAppState createState() => _LogoAppState();
}

class _LogoAppState extends State<LogoApp> {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        margin: EdgeInsets.symmetric(vertical: 10),
        height: 300,
        width: 300,
        child: FlutterLogo(),
      ),
    );
  }
}
```

App source: animate0

The following shows the same code modified to animate the logo to grow from nothing to full size. When defining an `AnimationController`, you must pass in a `vsync` object. The `vsync` parameter is described in the AnimationController section.

The changes from the non-animated example are highlighted:

{animate0 → animate1}/lib/main.dart

```
@@ -1,3 +1,4 @@
```

```
+ import 'package:flutter/animation.dart';
  import 'package:flutter/material.dart';
  void main() => runApp(LogoApp());
@@ -6,16 +7,39 @@
    _LogoAppState createState() => _LogoAppState();
  }
- class _LogoAppState extends State<LogoApp> {
+ class _LogoAppState extends State<LogoApp> with SingleTickerProviderStateMixin {
+   Animation<double> animation;
+   AnimationController controller;
+
+   @override
+   void initState() {
+     super.initState();
+     controller =
+         AnimationController(duration: const Duration(seconds: 2), vsync: this);
+     animation = Tween<double>(begin: 0, end: 300).animate(controller)
+       ..addListener(() {
+         setState(() {
+           // The state that has changed here is the animation object's value.
+         });
+       });
+     controller.forward();
+   }
+
    @override
    Widget build(BuildContext context) {
      return Center(
        child: Container(
          margin: EdgeInsets.symmetric(vertical: 10),
-         height: 300,
-         width: 300,
+         height: animation.value,
+         width: animation.value,
          child: FlutterLogo(),
        ),
      );
    }
+
+   @override
+   void dispose() {
+     controller.dispose();
+     super.dispose();
+   }
  }
```

App source: animate1

The `addListener()` function calls `setState()`, so every time the `Animation` generates a new number, the current frame is marke dirty, which forces `build()` to be called again. In `build()`, the container changes size because its height and width now use `animation.value` instead of a hardcoded value. Dispose of the controller when the `State` object is discarded to prevent memory leaks.

With these few changes, you've created your first animation in Flutter!

Dart language tricks: You might not be familiar with Dart's cascade notation—the two dots in `..addListener()`. This syntax means that the `addListener()` method is called with the return value from `animate()`. Consider the following example:

```
animation = Tween<double>(begin: 0, end: 300).animate(controller)
  ..addListener(() {
    // ···
  });
```

This code is equivalent to:

```
animation = Tween<double>(begin: 0, end: 300).animate(controller);
animation.addListener(() {
    // ···
  });
```

You can learn more about cascade notation in the Dart Language Tour.

Simplifying with AnimatedWidget

What's the point?

- How to use the AnimatedWidget helper class (instead of `addListener()` and `setState()`) to create a widget that animates.

- Use `AnimatedWidget` to create a widget that performs a reusable animation. To separate the transition from the widget, use an AnimatedBuilder.
- Examples of `AnimatedWidget`s in the Flutter API: AnimatedBuilder, AnimatedModalBarrier, DecoratedBoxTransition, FadeTransition, PositionedTransition, RelativePositionedTransition, RotationTransition, ScaleTransition, SizeTransition, SlideTransition.

The `AnimatedWidget` base class allows you to separate out the core widget code from the animation code. `AnimatedWidget` does need to maintain a `State` object to hold the animation. Add the following `AnimatedLogo` class:

lib/main.dart (AnimatedLogo)

```dart
class AnimatedLogo extends AnimatedWidget {
  AnimatedLogo({Key key, Animation<double> animation})
      : super(key: key, listenable: animation);

  Widget build(BuildContext context) {
    final animation = listenable as Animation<double>;
    return Center(
      child: Container(
        margin: EdgeInsets.symmetric(vertical: 10),
        height: animation.value,
        width: animation.value,
        child: FlutterLogo(),
      ),
    );
  }
```

`AnimatedLogo` uses the current value of the `animation` when drawing itself.

The `LogoApp` still manages the `AnimationController` and the `Tween`, and it passes the `Animation` object to `AnimatedLogo`:

{animate1 → animate2}/lib/main.dart

```
@@ -10,2 +27,2 @@
 10  27  class _LogoAppState extends State<LogoApp> with SingleTickerProviderStateMixin {
 11  28    Animation<double> animation;
@@ -13,32 +30,18 @@
 13  30    @override
 14  31    void initState() {
 15  32      super.initState();
 16  33      controller =
 17  34          AnimationController(duration: const Duration(seconds: 2), vsync: this);
 18      -    animation = Tween<double>(begin: 0, end: 300).animate(controller)
     35  +    animation = Tween<double>(begin: 0, end: 300).animate(controller);
 19      -      ..addListener(() {
 20      -        setState(() {
 21      -          // The state that has changed here is the animation object's value.
 22      -        });
 23      -      });
 24  36      controller.forward();
 25  37    }
 26  38    @override
 27      -  Widget build(BuildContext context) {
     39  +  Widget build(BuildContext context) => AnimatedLogo(animation: animation);
 28      -    return Center(
 29      -      child: Container(
 30      -        margin: EdgeInsets.symmetric(vertical: 10),
 31      -        height: animation.value,
 32      -        width: animation.value,
 33      -        child: FlutterLogo(),
 34      -      ),
 35      -    );
 36      -  }
 37  40    @override
 38  41    void dispose() {
 39  42      controller.dispose();
 40  43      super.dispose();
 41  44    }
```

App source: animate2

Monitoring the progress of the animation

What's the point?

- Use `addStatusListener()` for notifications of changes to the animation's state, such as starting, stopping, or reversing direction.
- Run an animation in an infinite loop by reversing direction when the animation has either completed or returned to its starting state.

It's often helpful to know when an animation changes state, such as finishing, moving forward, or reversing. You can get notificati for this with `addStatusListener()`. The following code modifies the previous example so that it listens for a state change and pr an update. The highlighted line shows the change:

```
class _LogoAppState extends State<LogoApp> with SingleTickerProviderStateMixin {
  Animation<double> animation;
  AnimationController controller;

  @override
  void initState() {
    super.initState();
    controller =
        AnimationController(duration: const Duration(seconds: 2), vsync: this);
    animation = Tween<double>(begin: 0, end: 300).animate(controller)
      ..addStatusListener((state) => print('$state'));
    controller.forward();
  }
  // ...
}
```

Running this code produces this output:

```
AnimationStatus.forward
AnimationStatus.completed
```

Next, use `addStatusListener()` to reverse the animation at the beginning or the end. This creates a "breathing" effect:

{animate2 → animate3}/lib/main.dart

```
@@ -32,7 +32,15 @@
32 32    void initState() {
33 33      super.initState();
34 34      controller =
35 35          AnimationController(duration: const Duration(seconds: 2), vsync: this);
36    -     animation = Tween<double>(begin: 0, end: 300).animate(controller);
   36 +     animation = Tween<double>(begin: 0, end: 300).animate(controller)
   37 +       ..addStatusListener((status) {
   38 +         if (status == AnimationStatus.completed) {
   39 +           controller.reverse();
   40 +         } else if (status == AnimationStatus.dismissed) {
   41 +           controller.forward();
   42 +         }
   43 +       })
   44 +       ..addStatusListener((state) => print('$state'));
37 45      controller.forward();
38 46    }
```

App source: animate3

Refactoring with AnimatedBuilder

What's the point?

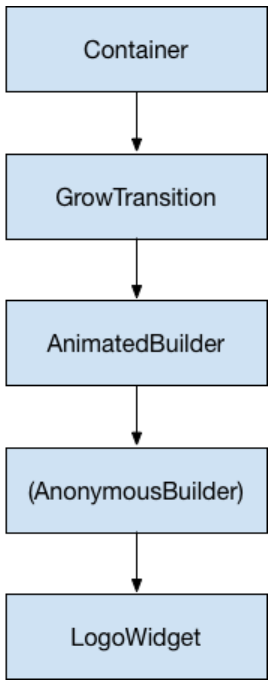
- An AnimatedBuilder understands how to render the transition.
- An `AnimatedBuilder` doesn't know how to render the widget, nor does it manage the Animation object.
- Use `AnimatedBuilder` to describe an animation as part of a build method for another widget. If you simply want to define a widget with a reusable animation, use AnimatedWidget.
- Examples of `AnimatedBuilders` in the Flutter API: `BottomSheet`, `ExpansionTile`, `PopupMenu`, `ProgressIndicator`, `RefreshIndicator`, `Scaffold`, `SnackBar`, `TabBar`, `TextField`.

One problem with the code in the animate3 example, is that changing the animation required changing the widget that renders th logo. A better solution is to separate responsibilities into different classes:

- Render the logo
- Define the Animation object
- Render the transition

You can accomplish this separation with the help of the `AnimatedBuilder` class. An `AnimatedBuilder` is a separate class in the render tree. Like `AnimatedWidget`, `AnimatedBuilder` automatically listens to notifications from the Animation object, and marks t widget tree dirty as necessary, so you don't need to call `addListener()`.

The widget tree for the animate4 example looks like this:



Starting from the bottom of the widget tree, the code for rendering the logo is straightforward:

```
class LogoWidget extends StatelessWidget {
  // Leave out the height and width so it fills the animating parent
  Widget build(BuildContext context) => Container(
        margin: EdgeInsets.symmetric(vertical: 10),
        child: FlutterLogo(),
      );
}
```

The middle three blocks in the diagram are all created in the `build()` method in `GrowTransition`, shown below. The `GrowTransit` widget itself is stateless and holds the set of final variables necessary to define the transition animation. The build() function cre and returns the `AnimatedBuilder`, which takes the (`Anonymous` builder) method and the LogoWidget object as parameters. The w of rendering the transition actually happens in the (`Anonymous` builder) method, which creates a `Container` of the appropriate size force the `LogoWidget` to shrink to fit.

One tricky point in the code below is that the child looks like it's specified twice. What's happening is that the outer reference of c is passed to `AnimatedBuilder`, which passes it to the anonymous closure, which then uses that object as its child. The net result that the `AnimatedBuilder` is inserted in between the two widgets in the render tree.

```
class GrowTransition extends StatelessWidget {
  GrowTransition({this.child, this.animation});

  final Widget child;
  final Animation<double> animation;

  Widget build(BuildContext context) => Center(
        child: AnimatedBuilder(
            animation: animation,
            builder: (context, child) => Container(
                  height: animation.value,
                  width: animation.value,
                  child: child,
                ),
            child: child),
      );
}
```

Finally, the code to initialize the animation looks very similar to the animate2 example. The `initState()` method creates an `AnimationController` and a `Tween`, then binds them with `animate()`. The magic happens in the `build()` method, which returns a `GrowTransition` object with a `LogoWidget` as a child, and an animation object to drive the transition. These are the three element listed in the bullet points above.

{animate2 → animate4}/lib/main.dart

```
@@ -27,22 +36,25 @@
 class _LogoAppState extends State<LogoApp> with SingleTickerProviderStateMixin {
   Animation<double> animation;
   AnimationController controller;
   @override
   void initState() {
     super.initState();
     controller =
         AnimationController(duration: const Duration(seconds: 2), vsync: this);
     animation = Tween<double>(begin: 0, end: 300).animate(controller);
     controller.forward();
   }
   @override
-  Widget build(BuildContext context) => AnimatedLogo(animation: animation);
+  Widget build(BuildContext context) => GrowTransition(
+        child: LogoWidget(),
+        animation: animation,
+      );
   @override
```

```
41  53    void dispose() {
42  54      controller.dispose();
43  55      super.dispose();
44  56    }
45  57  }
```

App source: animate4

Simultaneous animations

What's the point?

- The Curves class defines an array of commonly used curves that you can use with a CurvedAnimation.

In this section, you'll build on the example from monitoring the progress of the animation (animate3), which used `AnimatedWidge` animate in and out continuously. Consider the case where you want to animate in and out while the opacity animates from transparent to opaque.

Note: This example shows how to use multiple tweens on the same animation controller, where each tween manages a different effect in the animation. It is for illustrative purposes only. If you were tweening opacity and size in production code, you'd probably use FadeTransition and SizeTransition instead.

Each tween manages an aspect of the animation. For example:

```
controller =
    AnimationController(duration: const Duration(seconds: 2), vsync: this);
sizeAnimation = Tween<double>(begin: 0, end: 300).animate(controller);
opacityAnimation = Tween<double>(begin: 0.1, end: 1).animate(controller);
```

You can get the size with `sizeAnimation.value` and the opacity with `opacityAnimation.value`, but the constructor for `AnimatedWidget` only takes a single `Animation` object. To solve this problem, the example creates its own `Tween` objects and explicitly calculates the values.

Change `AnimatedLogo` to encapsulate its own `Tween` objects, and its `build()` method calls `Tween.evaluate()` on the parent's animation object to calculate the required size and opacity values. The following code shows the changes with highlights:

```dart
class AnimatedLogo extends AnimatedWidget {
  // Make the Tweens static because they don't change.
  static final _opacityTween = Tween<double>(begin: 0.1, end: 1);
  static final _sizeTween = Tween<double>(begin: 0, end: 300);

  AnimatedLogo({Key key, Animation<double> animation})
      : super(key: key, listenable: animation);

  Widget build(BuildContext context) {
    final animation = listenable as Animation<double>;
    return Center(
      child: Opacity(
        opacity: _opacityTween.evaluate(animation),
        child: Container(
          margin: EdgeInsets.symmetric(vertical: 10),
          height: _sizeTween.evaluate(animation),
          width: _sizeTween.evaluate(animation),
          child: FlutterLogo(),
        ),
      ),
    );
  }
}

class LogoApp extends StatefulWidget {
  _LogoAppState createState() => _LogoAppState();
}

class _LogoAppState extends State<LogoApp> with SingleTickerProviderStateMixin {
  Animation<double> animation;
  AnimationController controller;

  @override
  void initState() {
    super.initState();
    controller =
        AnimationController(duration: const Duration(seconds: 2), vsync: this);
    animation = CurvedAnimation(parent: controller, curve: Curves.easeIn)
      ..addStatusListener((status) {
        if (status == AnimationStatus.completed) {
          controller.reverse();
        } else if (status == AnimationStatus.dismissed) {
          controller.forward();
        }
      });
    controller.forward();
  }

  @override
  Widget build(BuildContext context) => AnimatedLogo(animation: animation);

  @override
  void dispose() {
    controller.dispose();
    super.dispose();
  }
}
```

App source: animate5

Next steps

This tutorial gives you a foundation for creating animations in Flutter using `Tweens`, but there are many other classes to explore. Y might investigate the specialized `Tween` classes, animations specific to Material Design, `ReverseAnimation`, shared element transitions (also known as Hero animations), physics simulations and `fling()` methods. See the animations landing page for the latest available documents and examples.