Implicit animations

Contents

- What are implicit animations?
- Example: Fade-in text effect
    - Fade-in (starter code)
    - Animate opacity with AnimatedOpacity widget
        - 1. Pick a widget property to animate
        - 2. Initialize a state variable for the animated property
        - 3. Set up a trigger for the animation, and choose an end value
        - 4. Set the duration of the animation
    - Fade-in (complete)
    - Putting it all together
- Example: Shape-shifting effect
    - Shape-shifting (starter code)
    - Animate color, borderRadius, and margin with AnimatedContainer
        - 1. Add an implicit animation
        - 2. Set starting values for animated properties
        - 3. Set up a trigger for the animation
        - 4. Set duration
    - Shape-shifting (complete)
    - Using animation curves
    - Putting it all together
- What's next?

Welcome to the implicit animations codelab, where you learn how to use Flutter widgets that make it easy to create animations for a specific set of properties.

To get the most out of this codelab, you should have basic knowledge about:

- How to make a Flutter app.
- How to use stateful widgets.

This codelab covers the following material:

- Using `AnimatedOpacity` to create a fade-in effect.
- Using `AnimatedContainer` to animate transitions in size, color, and margin.
- Overview of implicit animations and techniques for using them.

Estimated time to complete this codelab: 15-30 minutes.

What are implicit animations?

With Flutter's animation library, you can add motion and create visual effects for the widgets in your UI. One widget set in the library manages animations for you. These widgets are collectively referred to as *implicit animations*, or *implicitly animated widgets*, deriving their name from the ImplicitlyAnimatedWidgetclass that they implement. With implicit animations, you can animate a widget property by setting a target value; whenever that target value changes, the widget animates the property from the old value to the new one. In this way, implicit animations trade control for convenience—they manage animation effects so that you don't have to.

Example: Fade-in text effect

The following example shows how to add a fade-in effect to existing UI using an implicitly animated widget called AnimatedOpacity. **The example begins with no animation code**—it consists of a Material App home screen containing:

- A photograph of an owl.
- One **Show details** button that does nothing when clicked.
- Description text of the owl in the photograph.

Fade-in (starter code)

Click the **Run** button to run the example:

Important: This page uses an embedded version of DartPad to display examples and exercises. If you see empty boxes instead of DartPads, go to the DartPad troubleshooting page.

Animate opacity with AnimatedOpacity widget

This section contains a list of steps you can use to add an implicit animation to the fade-in starter code. After the steps, you can also run the fade-in complete code with the the changes already made. The steps outline how to use the `AnimatedOpacity` widget to add the following animation feature:

- The owl's description text remains hidden until the user clicks the **Show details** button.
- When the user clicks the **Show details** button, the owl's description text fades in.

1. Pick a widget property to animate

To create a fade-in effect, you can animate the `opacity` property using the `AnimatedOpacity` widget. Change the `Container` widget to an `AnimatedOpacity` widget:

{opacity1 → opacity2}/lib/main.dart

```
@@ -21,7 +21,7 @@
               style: TextStyle(color: Colors.blueAccent),
             ),
             onPressed: () => null),
-        Container(
+        AnimatedOpacity(
           child: Column(
26  26        children: <Widget>[
27  27          Text('Type: Owl'),
```

You can reference the line numbers in the example code to help track where to make these changes.

2. Initialize a state variable for the animated property

To hide the text before the user clicks **Show details**, set the starting value for `opacity` to zero:

🗎{opacity2 → opacity3}/lib/main.dart

```
          @@ -11,6 +11,8 @@
11  11     }
12  12     class _FadeInDemoState extends State<FadeInDemo> {
    13  +   double opacity = 0.0;
    14  +
13  15      @override
14  16      Widget build(BuildContext context) {
15  17        return Column(children: <Widget>[
          @@ -22,6 +24,8 @@
22  24           ),
23  25           onPressed: () => null),
24  26         AnimatedOpacity(
    27  +        duration: Duration(seconds: 3),
    28  +        opacity: opacity,
25  29          child: Column(
26  30           children: <Widget>[
27  31            Text('Type: Owl'),
```

3. Set up a trigger for the animation, and choose an end value

Configure the animation to trigger when the user clicks the **Show details** button. To do this, change `opacity` state using the `onPressed()` handler for `MaterialButton`. To make the `FadeInDemo` widget become fully visible when the user clicks the **Show details** button, use the `onPressed()` handler to set `opacity` to 1:

🗎{opacity4 → opacity5}/lib/main.dart

```
          @@ -18,11 +18,14 @@
18  18        return Column(children: <Widget>[
19  19         Image.network(owl_url),
20  20         MaterialButton(
21      -      child: Text(
22      -       'Show Details',
23      -       style: TextStyle(color: Colors.blueAccent),
24      -      ),
25      -      onPressed: () => null),
    21  +     child: Text(
    22  +      'Show Details',
    23  +      style: TextStyle(color: Colors.blueAccent),
    24  +     ),
    25  +     onPressed: () => setState(() {
    26  +      opacity = 1;
```

```diff
+        }),
+      ),
       AnimatedOpacity(
         duration: Duration(seconds: 2),
         opacity: opacity,
```

Notice that you only need to set the start and end values of `opacity`. The `AnimatedOpacity` widget manages everything in between.

4. Set the duration of the animation

In addition to an `opacity` parameter, `AnimatedOpacity` requires a [duration](#) to use for its animation. For this example, you can start with 2 seconds:

{opacity3 → opacity4}/lib/main.dart

```diff
@@ -24,7 +24,7 @@
           ),
           onPressed: () => null),
       AnimatedOpacity(
-        duration: Duration(seconds: 3),
+        duration: Duration(seconds: 2),
         opacity: opacity,
         child: Column(
           children: <Widget>[
```

Fade-in (complete)

Here's the example with the completed changes you've made—run this example and click the **Show details**button to trigger the animation.

Putting it all together

The [Fade-in text effect](#) example demonstrates the following features of `AnimatedOpacity`:

- `AnimatedOpacity` listens for state changes in its `opacity` property.
- Whenever `opacity` changes, `AnimatedOpacity` automatically animates the widget's transition to the new value for `opacity`.
- `AnimatedOpacity` requires a `duration` parameter to define the time it takes to animate the transition between an old `opacity` value and a new one.

Note that Implicit animations can only animate properties of a parent `StatefulWidget`, so this example begins with the `FadeInDemo` widget that extends `StatefulWidget`.

Notice also that `AnimatedOpacity` animates a single property: `opacity`. Some implicitly animated widgets can animate many properties, as the following example illustrates.

Example: Shape-shifting effect

The following example shows how to use the AnimatedContainer widget to animate multiple properties (`margin`, `borderRadius`, and `color`) with different types (`double` and `Color`). **The example begins with no animation code**—it starts with a Material App home screen that contains:

- A `Container` with `borderRadius`, `margin`, and `color` properties that are different each time you run the example.
- A **Change** button that does nothing when clicked.

Shape-shifting (starter code)

Click the **Run** button to run the example:

Animate color, borderRadius, and margin with AnimatedContainer

This section contains a list of steps you can use to add an implicit animation to the shape-shifting starter code. After the steps, you can also run the shape-shifting complete example with the changes already made.

In the shape-shifting starter code, each property in the `Container` widget (`color`, `borderRadius`, and `margin`) is assigned a value by an associated function (`randomColor()`, `randomBorderRadius()`, and `randomMargin()` respectively). By using an `AnimatedContainer` widget, you can refactor this code to do the following:

- Generate new values for `color`, `borderRadius`, and `margin` whenever the user clicks the **Change**button.
- Animate the transition to the new values for `color`, `borderRadius`, and `margin` whenever they are set.

1. Add an implicit animation

Change the `Container` widget to an `AnimatedContainer` widget:

{container1 → container2}/lib/main.dart

```
@@ -44,7 +44,7 @@
             SizedBox(
               width: 128,
               height: 128,
-              child: Container(
+              child: AnimatedContainer(
                 margin: EdgeInsets.all(margin),
                 decoration: BoxDecoration(
                   color: color,
```

You can reference the line numbers in the example code to help track where to make these changes in [shape-shifting starter code]

2. Set starting values for animated properties

`AnimatedContainer` automatically animates between old and new values of its properties when they change. Create a `change()` method that defines the behavior triggered when the user clicks the **Change**button. The `change()` method can use `setState()` to set new values for the `color`, `borderRadius`, and `margin` state variables:

{container2 → container3}/lib/main.dart

```
          @@ -35,6 +35,14 @@
 35   35          margin = randomMargin();
 36   36        }
      37  +     void change() {
      38  +       setState(() {
      39  +         color = randomColor();
      40  +         borderRadius = randomBorderRadius();
      41  +         margin = randomMargin();
      42  +       });
      43  +     }
      44  +
 37   45        @override
 38   46        Widget build(BuildContext context) {
 39   47          return Scaffold(
```

3. Set up a trigger for the animation

To set the animation to trigger whenever the user presses the **Change** button, invoke the `change()` method in the `onPressed()` handler:

{container3 → container4}/lib/main.dart

```
          @@ -66,7 +66,7 @@
 66   66                      'change',
 67   67                      style: TextStyle(color: Colors.white),
 68   68                    ),
 69       -                 onPressed: () => null,
      69  +                 onPressed: () => change(),
 70   70                  ),
 71   71                ],
 72   72              ),
```

4. Set duration

Finally, set the `duration` of the animation that powers the transition between the old and new values:

{container4 → container5}/lib/main.dart

```
          @@ -6,6 +6,8 @@
  6    6    import 'package:flutter/material.dart';
       7  + const _duration = Duration(milliseconds: 400);
       8  +
```

```diff
 double randomBorderRadius() {
   return Random().nextDouble() * 64;
 }
@@ -58,6 +60,7 @@
                   color: color,
                   borderRadius: BorderRadius.circular(borderRadius),
                 ),
+                duration: _duration,
               ),
             ),
             MaterialButton(
```

Shape-shifting (complete)

Here's the example with the completed changes you've made—run the code and click the **Change** button to trigger the animation. Notice that each time you click the **Change** button, the shape animates to its new values for `margin`, `borderRadius`, and `color`.

Using animation curves

The preceding examples show how implicit animations allow you to animate changes in values for specific widget properties, and how the `duration` parameter allows you to set the amount of time an animation takes to complete. Implicit animations also allow you to control changes to **the rate** of an animation within the `duration`. The parameter you use to define this change in rate is [curve](#).

The preceding examples do not specify a `curve`, so the implicit animations apply a [linear animation curve](#)by default. Add a `curve` parameter to the [shape-shifting complete](#) and watch how the animation changes when you pass the [easeInOutBack](#) constant for `curve`:

{container5 → container6}/lib/main.dart

```diff
@@ -61,6 +61,7 @@
                   borderRadius: BorderRadius.circular(borderRadius),
                 ),
                 duration: _duration,
+                curve: Curves.easeInOutBack,
               ),
             ),
             MaterialButton(
```

Now that you have passed `easeInOutBack` as the value for `curve` to `AnimatedContainer`, notice that the rates of change for `margin`, `borderRadius`, and `color` follow the curve defined by the `easeInOutBack`curve:

The `easeInOutBack` constant is only one of many that you can pass for the `curve` parameter. Explore the list of curve constants to discover more ways to use `curve` to modify the look and feel of your animations.

Putting it all together

The shape-shifting complete example animates transitions between values for `margin`, `borderRadius`, and `color` properties. Note that `AnimatedContainer` animates changes to any of its properties, including those you didn't use such as `padding`, `transform`, and even `child` and `alignment`! The shape-shifting completeexample builds upon fade-in complete by showing additional capabilities of implicit animations:

- Some implicit animations (for example, `AnimatedOpacity`) only animate a single property, while others (like `AnimatedContainer`) can animate many properties.
- Implicit animations automatically animate between the old and new values of properties when they change using the provided `curve` and `duration`.
- If you do not specify a `curve`, implicit animations default to a linear curve.