7

Adding Animation to an App

WHAT YOU WILL LEARN IN THIS CHAPTER

- How to use `AnimatedContainer` to gradually change values over time
- How to use `AnimatedCrossFade` to cross-fade between two children widgets
- How to use `AnimatedOpacity` to show or hide widget visibility by animated fading over time
- How to use the `AnimationController` to create custom animations
- How to use the `AnimationController` to control staggered animations

In this chapter, you'll learn how to add animation to an app to convey action, which can improve the user experience (UX) if appropriately used. Too many animations without conveying the appropriate action can make the UX worse. Flutter has two types of animation: physics-based and Tween. This chapter will focus on Tween animations.

Physics-based animation is used to mimic real-world behavior. For example, when an object is dropped and hits the ground, it will bounce and continue to move forward, but with each bounce, it continues to slow down with smaller rebounds and eventually stop. As the object gets closer to the ground with each bounce, the velocity increases, but the height of the bounce decreases.

Tween is short for "in-between," meaning that the animation has beginning and ending points, a timeline, and a curve that specifies the timing and speed of the transition. The beauty is that the framework automatically calculates the transition from the beginning to end point.

Beginning Flutter®: A Hands On Guide To App Development, First Edition.
Marco L. Napoli.

USING ANIMATEDCONTAINER

Let's start with a simple animation by using the `AnimatedContainer` widget. This is a `Container` widget that gradually changes values over a period of time. The `AnimatedContainer` constructor has arguments called `duration`, `curve`, `color`, `height`, `width`, `child`, `decoration`, `transform`, and many others.

TRY IT OUT Creating the AnimatedContainer App

In this project, you'll animate the width of a `Container` widget by tapping it. For example, you could use this type of animation to animate a horizontal bar chart.

1. Create a new Flutter project and name it `ch7_animations`. You can follow the instructions in Chapter 4, "Creating a Starter Project Template." For this project, you need to create only the `pages` and `widgets` folders.
2. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder, select New ➪ Dart File, enter **`animated_container.dart`**, and click the OK button to save.
3. Import the `material.dart` library, add a new line, and then start typing **`st`**. The autocompletion help opens. Select the `stful` abbreviation and give it a name of `AnimatedContainerWidget`.
4. After the class `_AnimatedContainerWidgetState extends State<AnimatedContainerWidget>` and before `@override`, add the variables `_height` and `_width` and the `_increaseWidth()` method.

   The `_height` and `_width` variables are of type `double`.

   ```
   double _height = 100.0;
   ```

   The `_increaseWidth()` method calls the `setState()` method to notify the framework that the `_width` value has changed and schedules a build for the state of this object to redraw the subtree. If you do not call `setState()`, the `_width` value still changes, but the `AnimatedContainer` widget will not redraw with the new value.

   ```
   class _AnimatedContainerWidgetState extends State<AnimatedContainerWidget> {
     double _height = 100.0;
     double _width = 100.0;

     _increaseWidth() {
       setState(() {
         _width = _width >= 320.0 ? 100.0 : _width += 50.0;
       });
     }

     @override
     Widget build(BuildContext context) {
   ```

   When the page loads, the `_height` and `_width` variables are initiated with values of 100.0 pixels. When the `FlatButton` is tapped, the `onPressed` property calls the `_increaseWidth()` method.

Here you'll use 320.0 pixels as the maximum allowed width, but this could have been the device width instead. With each tap event, you increase the current width by 50.0 pixels starting from 100.0 pixels. As the width increases, once it goes above 320.0 pixels, you reset the size to 100.0 pixels. To calculate `AnimatedContainer`'s new `_width`, use the ternary operator. If the `_width` is greater than or equal to 320.0 pixels, then set `_width` to the original 100.0 pixels. This will animate `AnimatedContainer` back to the original size. Otherwise, take the current `_width` value and add 50.0 pixels. Note that the plus and equal signs (`+=`) are used to take the current value and add to it.

Notice that the height and width values are private variables, `_height` and `_width`, since you are leading them with the underscore symbol.

The `FlatButton` child is the label that shows the message "Tap to Grow Width." I am using `\n` to continue the text on the next line and using the `$` sign to pass the `_width` value.

Once the `FlatButton` is pressed, you add a call to the method `_increaseWidth()` in the `onPressed` property. Ignore the code editor's red squiggly lines since you have not created the variables and method yet.

The `duration` argument takes a `Duration()` object to specify the type of time to use such as `microseconds`, `milliseconds`, `seconds`, `minutes`, `hours`, and `days`.

```
duration: Duration(milliseconds: 500),
```

The `curve` argument takes a `Curves` class and uses `Curves.elasticOut`. Some of the types of `Curves` available are `bounceIn`, `bounceInOut`, `bounceOut`, `easeIn`, `easeInOut`, `easeOut`, `elasticIn`, `elasticInOut`, and `elasticOut`.

```
curve: Curves.elasticOut,
```

Here's the full `_AnimatedContainerWidget()` widget class:

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

class AnimatedContainerWidget extends StatefulWidget {
  const AnimatedContainerWidget({
    Key key,
  }) : super(key: key);

  @override
  _AnimatedContainerWidgetState createState() => _AnimatedContainerWidgetState();
}

class _AnimatedContainerWidgetState extends State<AnimatedContainerWidget> {
  double _height = 100.0;
  double _width = 100.0;

  void _increaseWidth() {
    setState(() {
      _width = _width >= 320.0 ? 100.0 : _width += 50.0;
    });
  }
```

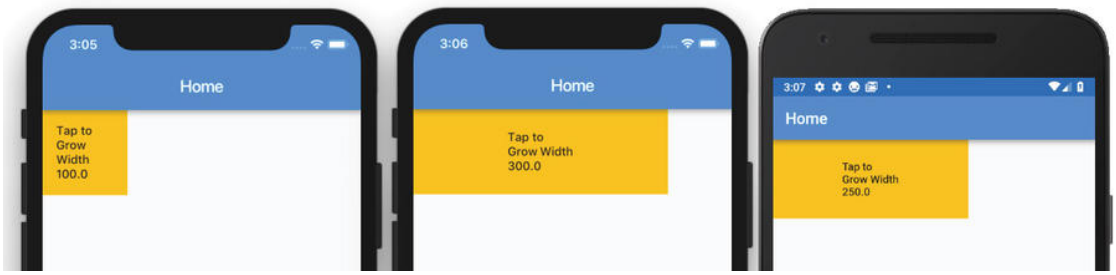
```
  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        AnimatedContainer(
          duration: Duration(milliseconds: 500),
          curve: Curves.elasticOut,
          color: Colors.amber,
          height: _height,
          width: _width,
          child: FlatButton(
            child: Text('Tap to\nGrow Width\n$_width'),
            onPressed: () {
              _increaseWidth();
            },
          ),
        ),
      ],
    );
  }
}
```

5. Open the `home.dart` file and import the `animated_container.dart` file to the top of the page.

```
import 'package:flutter/material.dart';
import 'package:ch7_animations/widgets/animated_container.dart';
```

6. Add to the body a `SafeArea` widget with `Column` as a `child`. In the `Column` children, add the call to the widget class `AnimatedContainerWidget()`.

```
body: SafeArea(
  child: Column(
    children: <Widget>[
      AnimatedContainerWidget(),
    ],
  ),
),
```



How It Works

The `AnimatedContainer` constructor takes a `duration` argument, and you use the `Duration` class to specify 500 milliseconds. The 500 milliseconds equals half a second. The `curve` argument gives the animation a spring effect by using `Curves.elasticOut`. The `onPressed` argument calls the `_increaseWidth()` method to change the `_width` variable dynamically. The `setState()` method notifies the Flutter framework that the internal state of the object changed and causes the framework to schedule a build for this `State` object. The `AnimatedContainer` widget automatically animates between the old `_width` value and the new `_width` value.

USING ANIMATEDCROSSFADE

The `AnimatedCrossFade` widget provides a great cross-fade between two children widgets. The `AnimatedCrossFade` constructor takes `duration`, `firstChild`, `secondChild`, `crossFadeState`, `sizeCurve`, and many other arguments.

TRY IT OUT Adding the AnimatedCrossFade Widget

This example creates a cross-fade between colors and size by tapping the widget. The widget will cross-fade the changing size and color from yellow to green.

1. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder, select New ➪ Dart File, enter **`animated_cross_fade.dart`**, and click the OK button to save.
2. Import the `material.dart` library, add a new line, and start typing **`st`**. The autocompletion help opens. Select the `stful` abbreviation and give it a name of `AnimatedCrossFadeWidget`.
3. After the class `_AnimatedCrossFadeWidgetState extends State<AnimatedCrossFadeWidget>` and before `@override`, add a variable for `_crossFadeStateShowFirst` and the `_crossFade()` method.

   The `_crossFadeStateShowFirst` variable is of type Boolean (`bool`).

   ```
   bool _crossFadeStateShowFirst = true;
   ```

   The `_crossFade()` method calls `setState()` to notify the framework that the `_crossFadeStateShowFirst` value has changed and schedules a build for the state of this object to redraw the subtree. If you do not call `setState()`, the `_width` value still changes, but the `AnimatedCrossFade` widget will not redraw with the new value.

   ```
   class _AnimatedCrossFadeWidgetState extends State<AnimatedCrossFadeWidget> {
     bool _crossFadeStateShowFirst = true;
   ```

```
  void _crossFade() {
    setState(() {
      _crossFadeStateShowFirst = _crossFadeStateShowFirst ? false : true;
    });
  }

  @override
  Widget build(BuildContext context) {
```

When the page is loaded, the `_crossFadeStateShowFirst` variable is initiated with a value of `true`. When `FlatButton` is tapped, the `onPressed` property calls the `_crossFade()` method.

To track which child (`firstChild`, `secondChild`) the `AnimatedCrossFade` widget should show and animate, use the ternary operator. If the `_crossFadeStateShowFirst` is `true`, then set the `_crossFadeStateShowFirst` value to `false`. Otherwise, set it to `true` since the current value is `false`.

4. To keep the UI clean, add each animation widget in a separate `Row`. Embed the `AnimatedCrossFade` widget in a `Row` with a `Stack` as a child. The reason to use a `Stack` is to add a `FlatButton` over the `AnimatedCrossFade` widget to give it a label and an `onPressed` event. You could also use a `GestureDetector` widget.

```
@override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        Stack(
          alignment: Alignment.center,
          children: <Widget>[
            AnimatedCrossFade(
              duration: Duration(milliseconds: 500),
              sizeCurve: Curves.bounceOut,
              crossFadeState: _crossFadeStateShowFirst ? CrossFadeState.showFirst :
CrossFadeState.showSecond,
              firstChild: Container(
                color: Colors.amber,
                height: 100.0,
                width: 100.0,
              ),
              secondChild: Container(
                color: Colors.lime,
                height: 200.0,
                width: 200.0,
              ),
            ),
            Positioned.fill(
              child: FlatButton(
                child: Text('Tap to\nFade Color & Size'),
                onPressed: () {
                  _crossFade();
                },
              ),
            ),
          ],
```

```
          ),
        ],
      );
    }
```

5. Use a duration of 500 milliseconds like in the previous animation.

```
duration: Duration(milliseconds: 500),
```

6. For `sizeCurve`, use `Curves.bounceOut` to see how the `Curves` class affects animations.

```
sizeCurve: Curves.bounceOut,
```

7. To decide which `Container` widget to show when the animation is completed, set the `crossFadeState` argument value by using the ternary operator to check whether the `_crossFadeStateShowFirst` value is true; then show `CrossFadeState.showFirst`; otherwise, show `CrossFadeState.showSecond`.

```
crossFadeState: _crossFadeStateShowFirst ? CrossFadeState.showFirst :
CrossFadeState.showSecond,
```

8. The animation will cross-fade between colors and sizes; for `firstChild` and `secondChild`, use a `Container`. The `firstChild` `Container` has a `Colors.amber` color property and `height` and `width` values of 100.0 pixels. The `secondChild` `Container` has a `Colors.lime` color property and `height` and `width` values of 200.0 pixels.

```
firstChild: Container(
  color: Colors.amber,
  height: 100.0,
  width: 100.0,
),
secondChild: Container(
  color: Colors.lime,
  height: 200.0,
  width: 200.0,
),
```

9. Add the second `Stack` child and call the `Positioned.fill` constructor with a child of `FlatButton`. Using `Positioned.fill` allows the `FlatButton` widget to resize itself to the maximum size of the `Stack` widget. For the `onPressed` property, add a call to the `_crossFade()` method.

```
Positioned.fill(
  child: FlatButton(
    child: Text('Tap to\nFade Color & Size'),
    onPressed: () {
      _crossFade();
    },
  ),
),
```

Here's the full `_AnimatedCrossFadeWidget()` widget class:

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

class AnimatedCrossFadeWidget extends StatefulWidget {
  const AnimatedCrossFadeWidget({
```

```
    Key key,
  }) : super(key: key);

  @override
  _AnimatedCrossFadeWidgetState createState() => _AnimatedCrossFadeWidgetState();
}

class _AnimatedCrossFadeWidgetState extends State<AnimatedCrossFadeWidget> {
  bool _crossFadeStateShowFirst = true;

  void _crossFade() {
    setState(() {
      _crossFadeStateShowFirst = _crossFadeStateShowFirst ? false : true;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        Stack(
          alignment: Alignment.center,
          children: <Widget>[
            AnimatedCrossFade(
              duration: Duration(milliseconds: 500),
              sizeCurve: Curves.bounceOut,
              crossFadeState: _crossFadeStateShowFirst ? CrossFadeState.showFirst :
CrossFadeState.showSecond,
              firstChild: Container(
                color: Colors.amber,
                height: 100.0,
                width: 100.0,
              ),
              secondChild: Container(
                color: Colors.lime,
                height: 200.0,
                width: 200.0,
              ),
            ),
            Positioned.fill(
              child: FlatButton(
                child: Text('Tap to\nFade Color & Size'),
                onPressed: () {
                  _crossFade();
                },
              ),
            ),
          ],
        ),
      ],
    );
  }
}
```

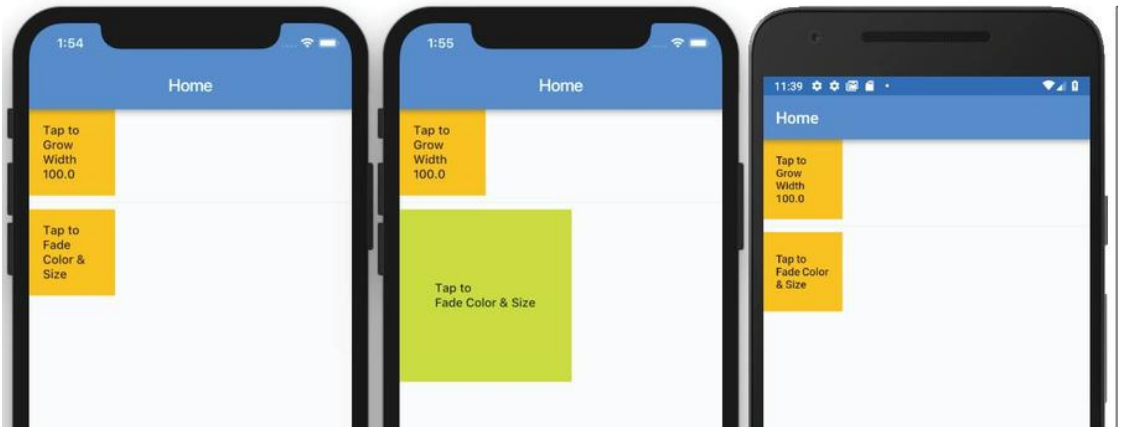
10. Continue editing the `home.dart` file and import the `animated_cross_fade.dart` file to the top of the page.

```
import 'package:flutter/material.dart';
import 'package:ch7_animations/widgets/animated_container.dart';
import 'package:ch7_animations/widgets/animated_cross_fade.dart';
```

11. Just after the `AnimatedContainerWidget()` widget class, add a `Divider()` widget. On the next line, add a call to the widget class `AnimatedCrossFadeWidget()`.

```
body: SafeArea(
  child: Column(
    children: <Widget>[
      AnimatedContainerWidget(),
      Divider(),
      AnimatedCrossFadeWidget(),
    ],
  ),
),
```

Notice how clean the code looks and how it improves the readability by separating each major section of the widget tree. More importantly, only the widget class running is being rebuilt, and the other animation widget classes are not, giving you great performance.



How It Works

The `AnimatedCrossFade` constructor takes a `duration` argument, and you use the `Duration` class to specify 500 milliseconds. The `sizeCurve` argument gives the animation between the two children's size a spring effect by using `Curves.bounceOut`. The `crossFadeState` argument sets the `child` widget to be shown once the animation is completed. By using the `_crossFadeStateShowFirst` variable, the correct `crossFadeState` `child` is displayed. The `firstChild` and `secondChild` arguments hold the two widgets to animate.

USING ANIMATEDOPACITY

If you need to hide or partially hide a widget, `AnimatedOpacity` is a great way to animate fading over time. The `AnimatedOpacity` constructor the takes `duration`, `opacity`, `curve`, and `child` arguments. For this example, you do not use a curve; since you want a smooth fade-out and fade-in, it's not necessary.

TRY IT OUT Adding the AnimatedOpacity Widget

The example uses `opacity` to partially fade out a `Container` widget. The widget will animate the `opacity` value from fully visible to almost faded out.

1. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder, select New ➪ Dart File, enter **`animated_opacity.dart`**, and click the OK button to save.
2. Import the `material.dart` library, add a new line, and then start typing **`st`**. The autocompletion help opens. Select the `stful` abbreviation and give it a name of `AnimatedOpacityWidget`.
3. After the class `_AnimatedOpacityWidgetState extends State<AnimatedOpacityWidget>` and before `@override`, add the variable for `_opacity` and the `_animatedOpacity()` method.

   The `_opacity` variable is of type `double`.

   ```
   double _opacity = 1.0;
   ```

   The `_animatedOpacity()` method calls `setState()` to notify the framework that the `_opacity` value has changed and schedules a build for the state of this object to redraw the subtree. If you do not call `setState()`, the `_opacity` value still changes, but the `AnimatedOpacity` widget does not redraw with the new value.

   ```
   class _AnimatedOpacityWidgetState extends State<AnimatedOpacityWidget> {
     double _opacity = 1.0;

     void _animatedOpacity() {
       setState(() {
         _opacity = _opacity == 1.0 ? 0.3 : 1.0;
       });
     }

     @override
     Widget build(BuildContext context) {
   ```

   When the page is loaded, the `_opacity` variable is initiated with a value of 1.0, which is fully visible. When the `FlatButton` widget is tapped, the `onPressed` property calls the `_animatedOpacity()` method.

   To calculate the widget opacity, use the ternary operator. If the `_opacity` value is `1.0`, then set `_opacity` to `0.3`. Otherwise, set it to `1.0` since the current value is `0.3`.

4. To keep the UI clean, add each animation widget in a separate `Row`. Embed the `AnimatedOpacity` widget in a `Row`.

```
@override
Widget build(BuildContext context) {
  return Row(
    children: <Widget>[
      AnimatedOpacity(
        duration: Duration(milliseconds: 500),
        opacity: _opacity,
        child: Container(
          color: Colors.amber,
          height: 100.0,
          width: 100.0,
          child: FlatButton(
            child: Text('Tap to Fade'),
            onPressed: () {
              _animatedOpacity();
            },
          ),
        ),
      ),
    ],
  );
}
```

5. Use a duration of 500 milliseconds like in the previous animations.

```
duration: Duration(milliseconds: 500),
```

6. The animation animates the `opacity` value of the `AnimatedOpacity` child widget, which is a `Container` in this case. Depending on the value of `opacity`, the `Container` widget fades out or fades in. An `opacity` value of 1.0 is fully visible, and an `opacity` value of 0.0 is invisible. You are going to animate the `_opacity` variable from 1.0 to 0.3, and vice versa.

```
opacity: _opacity,
```

7. Add a `Container` widget as a child of the `AnimatedOpacity` widget. Add a `FlatButton` widget as a child of the `Container` widget with `onPressed` calling the `_animatedOpacity()` method.

```
child: Container(
  color: Colors.amber,
  height: 100.0,
  width: 100.0,
  child: FlatButton(
    child: Text('Tap to Fade'),
    onPressed: () {
      _animatedOpacity();
    },
  ),
),
```

Here's the full `_buildAnimatedOpacity()` widget class:

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

class AnimatedOpacityWidget extends StatefulWidget {
  const AnimatedOpacityWidget({
    Key key,
  }) : super(key: key);

  @override
  _AnimatedOpacityWidgetState createState() => _AnimatedOpacityWidgetState();
}

class _AnimatedOpacityWidgetState extends State<AnimatedOpacityWidget> {
  double _opacity = 1.0;

  void _animatedOpacity() {
    setState(() {
      _opacity = _opacity == 1.0 ? 0.3 : 1.0;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        AnimatedOpacity(
          duration: Duration(milliseconds: 500),
          opacity: _opacity,
          child: Container(
            color: Colors.amber,
            height: 100.0,
            width: 100.0,
            child: FlatButton(
              child: Text('Tap to Fade'),
              onPressed: () {
                _animatedOpacity();
              },
            ),
          ),
        ),
      ],
    );
  }
}
```

8. Continue editing the `home.dart` file and import the `animated_opacity.dart` file to the top of the page.

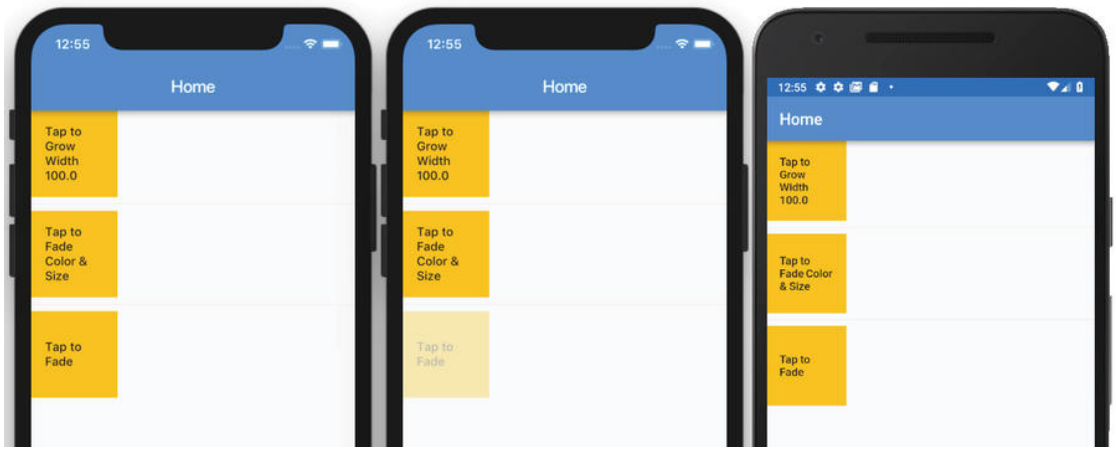
```
import 'package:flutter/material.dart';
import 'package:ch7_animations/widgets/animated_container.dart';
```

```
import 'package:ch7_animations/widgets/animated_cross_fade.dart';
import 'package:ch7_animations/widgets/animated_opacity.dart';
```

9. Just after the `AnimatedCrossFadeWidget()` widget class, add a `Divider()` widget. On the next line, add the call to the widget class `AnimatedOpacityWidget()`.

```
body: SafeArea(
  child: Column(
    children: <Widget>[
      AnimatedContainerWidget(),
      Divider(),
      AnimatedCrossFadeWidget(),
      Divider(),
      AnimatedOpacityWidget(),
    ],
  ),
),
```

Again, notice how clean the code looks and how it improves the readability by separating each major section of the widget tree.



How It Works

The `AnimatedOpacity` widget takes a `duration` parameter, and you use the `Duration` class to specify 500 milliseconds. The `opacity` parameter is a value from 0.0 to 1.0. The `opacity` value of 1.0 is fully visible, and as the value changes toward zero, it starts to fade away. Once it reaches zero, it's invisible.

USING ANIMATIONCONTROLLER

The `AnimationController` class gives you increased flexibility in animation. The animation can be played forward or reverse, and you can stop it. The `fling` animation uses a physics simulation like a spring.

The `AnimationController` class produces linear values for a giving duration, and it tries to display a new frame at around 60 frames per second. The `AnimationController` class needs a `TickerProvider` class by passing the `vsync` argument in the constructor. The `vsync` prevents off-screen animations from consuming unnecessary resources. If the animation needs only one `AnimationController`, use `SingleTickerProviderStateMixin`. If the animation needs multiple `AnimationControllers`, use `TickerProviderStateMixin`. The `Ticker` class is driven by the `ScheduleBinding.scheduleFrameCallback` reporting once per animation frame. It is trying to sync the animation to be as smooth as possible.

The `AnimationController` default object ranges from 0.0 to 1.0, but if you need a different range, you can use the `Animation` class (using `Tween`) to accept a different type of data. The `Animation` class is initiated by setting the `Tween` class (in-betweening) `begin` and `end` property values. For example, you have a balloon that floats from the bottom to the top of the screen, and you would set the `Tween` class `begin` value of 400.0, the bottom of the screen and the `end` value of 0.0, the top of the screen. Then you can chain the Tween `animate` method, which returns an `Animation` class. Simply put, it animates the Tween based on the animation, such as a `CurvedAnimation` class.

The `AnimationController` class at first can seem complex to use because of the different classes needed. The following are the basic steps that you take to create a custom animation (shown in Figure 7.1) or, eventually in the example, multiple animations running at the same time.

1. Add `AnimationController`.
2. Add `Animation`.
3. Initiate `AnimationController` with `Duration` (milliseconds, seconds, and so on).
4. Initiate `Animation` with `Tween` with `begin` and `end` values and chain the `animate` method with a `CurvedAnimation` (for this example).
5. Use the `AnimatedBuilder` with `Animation` using a `Container` with a balloon to start `Animation` by calling the `AnimationController.forward()` and `.reverse()` to run the animation backward. The `AnimatedBuilder` widget is used to create a widget that performs a reusable animation.

As you can see, once you break down the steps, it becomes more manageable and less complicated.

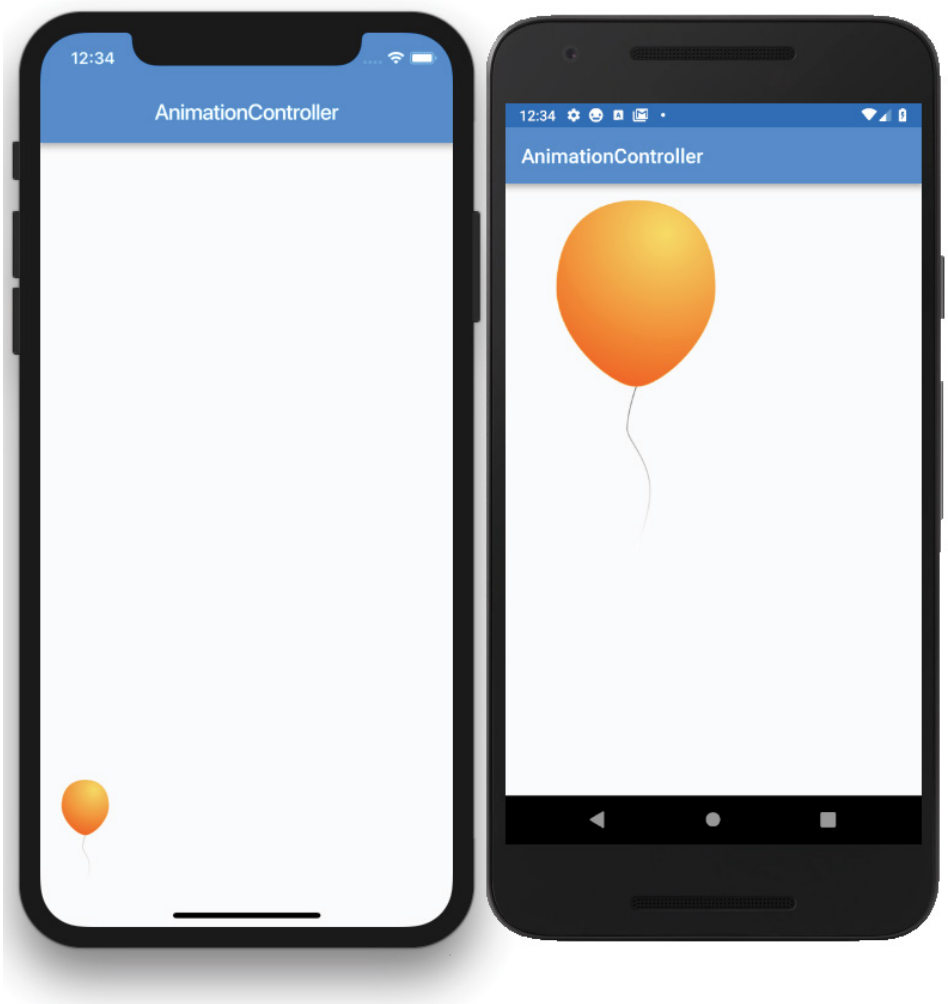


FIGURE 7.1: What you're building with AnimationController

TRY IT OUT Creating the AnimationController App

In this project, you'll animate a balloon that starts small at the bottom of the screen, and as it inflates, it floats toward the top, giving some nice spring animations. By using a `GestureDetector` tap on the balloon, the animation reverses, showing the balloon deflating and floating downward to the bottom of the screen. Every time the balloon is tapped, the animation starts again.

1. Create a new Flutter project and name it `ch7_animation_controller`. You can follow the instructions in Chapter 4. For this project, you need to create only the `pages`, `widgets`, and `assets/images` folders.

2. Open the `pubspec.yaml` file, and in `assets` add the `images` folder.

```
# To add assets to your application, add an assets section, like this:
assets:
  - assets/images/
```

Add the folder `assets` and subfolder `images` at the project's root and then copy the `Beginning-GoogleFlutter-Balloon.png` file to the `images` folder.

3. Click the Save button, and depending on the editor you are using, this automatically runs `flutter packages get`. Once finished, it shows a message of `Process finished with exit code 0`. It does not automatically run the command for you, though. Open the Terminal window (located at the bottom of your editor) and type **`flutter packages get`**.

4. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder, select New ➪ Dart File, enter **`animated_balloon.dart`**, and click the OK button to save.

5. Import the `material.dart` library, add a new line, and then start typing **`st`**. The autocompletion help opens. Select the `stful` abbreviation and give it a name of `AnimatedBalloonWidget`.

6. It is appropriate to create the `AnimationController` and `Animation` variables before you can reference them in your code. Declare `TickerProviderStateMixin` to the `_Animated-BalloonWidgetState` class by adding `with TickerProviderStateMixin`. The `Animation-Controller vsync` argument will use it. The `vsync` is referenced by `this`, meaning this reference of the `_AnimatedBalloonWidgetState` class.

```
class _AnimatedBalloonWidgetState extends State<AnimatedBalloonWidget> with
TickerProviderStateMixin {
```

7. After the class `_AnimatedBalloonWidgetState extends State<AnimatedBalloonWidget>` and before `@override`, create two `AnimationController`s to handle the duration of the balloon floating upward and the inflation of the balloon inflating. Create two `Animation`s to handle the actual movement range and inflation size. Override the `initState()` and `dispose()` methods to initiate `AnimationController` and dispose of them when the page closes.

An important note about using two `AnimationController`s is that you could have used one `AnimationController` and made use of the `Interval() curve` to stagger `Animation`. A staggered animation uses `Interval()` to `begin` and `end` animations sequentially or to overlap one another. In the "Using Staggered Animations" section of this chapter, you'll create a staggered animation app.

```
class _AnimatedBalloonWidgetState extends State<AnimatedBalloonWidget> with
TickerProviderStateMixin {
  AnimationController _controllerFloatUp;
  AnimationController _controllerGrowSize;
  Animation<double> _animationFloatUp;
  Animation<double> _animationGrowSize;

  @override
  void initState() {
    super.initState();
```

```
    _controllerFloatUp = AnimationController(duration: Duration( seconds: 4),
vsync: this);
    _controllerGrowSize = AnimationController(duration: Duration(seconds: 2),
vsync: this);
  }

  @override
  void dispose() {

    _controllerFloatUp.dispose();
    _controllerGrowSize.dispose();
    super.dispose();
  }
```

Note that in the `initState()` method you could call the `Animation` class after the `AnimationController` to start the animation as the page loads, but you are going to place the `Animation` class in `Widget build(BuildContext context)` instead. The reason for this is to use the `MediaQuery` class to obtain the screen size to place and size the balloon accordingly. The `initState()` method does not contain the `Widget context` object that `MediaQuery` requires. However, as `Widget build(BuildContext context)` is called, the animation starts on page load. If the user rotates the device, `Widget build(BuildContext context)` is called again, and the balloon will resize itself accordingly and run the animation if the balloon is located at the bottom of the screen.

8. After `Widget build(BuildContext context)`, create the balloon height, width, and page bottom position by using `MediaQuery.of(context).size`. I arrived at the following formulas for calculating the dimensions by testing different options to get the best look depending on different devices' screen size and orientation:

```
Widget build(BuildContext context) {
  double _balloonHeight = MediaQuery.of(context).size.height / 2;
  double _balloonWidth = MediaQuery.of(context).size.height / 3;
  double _balloonBottomLocation = MediaQuery.of(context).size.height -
_balloonHeight;
```

Create the `Animation` class by declaring a `Tween` with a `CurvedAnimation`. For the `_animationFloatUp` value, the `begin` property is the `_balloonBottomLocation` variable with the `end` property at 0.0. This means the floating-upward animation starts at the bottom of the screen and moves all the way to the top.

For the `_animationGrowSize` value, the `begin` property is 50.0 pixels. Set the `end` value to the `_balloonWidth` variable. This means you can start the balloon width at 50.0 pixels and increase it to the maximum `_balloonWidth` value calculated by `MediaQuery`. Use the `elasticInOut` curve to give it a beautiful spring animation that looks like a quick inflation of air.

9. Call both `_controllerFloatUp.forward()` and `_controllerGrowSize.forward()` to start the animation.

```
  _animationFloatUp = Tween(begin: _balloonBottomLocation, end: 0.0).
animate(CurvedAnimation(parent: _controllerFloatUp, curve: Curves.fastOutSlowIn));
  _animationGrowSize = Tween(begin: 50.0, end: _balloonWidth).
animate(CurvedAnimation(parent: _controllerGrowSize, curve: Curves.elasticInOut));
```

```
    _controllerFloatUp.forward();
    _controllerGrowSize.forward();
  }
```

10. Create `AnimatedBuilder` and let's take a look at a high-level structural breakdown for the `AnimatedBuilder` arguments that show how `AnimatedBuilder` `child` (`GestureDetector` with `Image`) is passed to the `builder`, which is the widget to receive the animation.

```
return AnimatedBuilder(
  animation: _animationFloatUp,
  builder: (context, child) {
    return Container(
      child: child,
    );
  },
  child: GestureDetector(/* Image*/) */
);
```

The `AnimatedBuilder` constructor passes the `animation` argument `_animationFloatUp`. For the `builder` argument, return a `Container` widget with the `child` property of `child`. I know that sounds weird for the `child` property, but this is the way the builder smartly redraws the `child` control being animated. The `child` widget being passed is declared next, and it will contain a `GestureDetector` widget and a `child` widget of `Image` (balloon).

The `AnimatedBuilder` constructor has the `animation`, `builder`, and `child` arguments.

Add to the `GestureDetector()` widget the `onTap` property. For the `GestureDetector()` widget, the `onTap` property checks for `_controllerFloatUp.isCompleted` (meaning the animation is done), and if yes, it starts the animation in reverse. This will deflate the balloon and start floating it down to the bottom of the page. The `else` portion handles the balloon already at the bottom of the screen and starts the animation forward by floating upward and inflating the balloon back to normal size.

11. Add to the `GestureDetector` `child` property a call to the `Image.asset()` constructor that loads the `BeginningGoogleFlutter-Balloon.png` file and sets `height` to `_balloonHeight` and `width` to `_balloonWidth`.

```
return AnimatedBuilder(
  animation: _animationFloatUp,
  builder: (context, child) {
    return Container(
      child: child,
      margin: EdgeInsets.only(
        top: _animationFloatUp.value,
      ),
```

```
        width: _animationGrowSize.value,
      );
    },
    child: GestureDetector(
      onTap: () {
        if (_controllerFloatUp.isCompleted) {
          _controllerFloatUp.reverse();
          _controllerGrowSize.reverse();
        }
        else {
          _controllerFloatUp.forward();
          _controllerGrowSize.forward();
        }
      },
      child: Image.asset(
          'assets/images/BeginningGoogleFlutter-Balloon.png',
          height: _balloonHeight,
          width: _balloonWidth),
    ),
  );
```

12. Open the `home.dart` file and import the `animated_balloon.dart` file to the top of the page.

```
import 'package:flutter/material.dart';
import 'package:ch7_animation_controller/widgets/animated_balloon.dart';
```

13. Add to the body a `SafeArea` with `SingleChildScrollView` as a `child`.
14. Add `Padding` as a child of the `SingleChildScrollView`.
15. Add a `Column` as a `child` of `Padding`.
16. In the `Column` children, add the call to the widget class `AnimatedBalloonWidget()`. Note I am using the `NeverScrollableScrollPhysics()` to stop the `SingleChildScrollView` to scroll content.

```
body: SafeArea(
  child: SingleChildScrollView(
    physics: NeverScrollableScrollPhysics(),
    child: Padding(
      padding: EdgeInsets.all(16.0),
      child: Column(
        children: <Widget>[
          AnimatedBalloonWidget(),
        ],
      ),
    ),
  ),
),
```

How It Works

Declaring `TickerProviderStateMixin` to the `AnimatedBalloonWidget` widget class allowed you to set the `AnimationController vsync` argument. You added `AnimationController` and declared the `_controllerFloatUp` variable to animate the floating upward and downward action. You declared the `AnimationController _controllerGrowSize` variable to animate the inflating and deflating actions. You declared the `_animationFloatUp` variable to hold the value from the `Tween` animation to show the balloon floating either upward or downward by setting the top margin of the `Container` widget. You declared the `_animationGrowSize` variable to hold the value from the `Tween` animation to show the balloon either inflating or deflating by setting the `width` value of the `Container` widget.

The `AnimatedBuilder` constructor takes the `animation`, `builder`, and `child` arguments. Next, you passed the `_animationFloatUp` animation to the `AnimatedBuilder` constructor. The `AnimatedBuilder builder` argument returns a `Container` widget with the child as an `Image` widget wrapped in a `GestureDetector` widget.

In the preceding example, I showed how you can use multiple `AnimationControllers` to run at the same time with different `Duration` values. In the next section, you'll use one `AnimationController` for a staggered animation.

Using Staggered Animations

A *staggered animation* triggers visual changes in sequential order. The animation changes can occur one after the other; they can have gaps without animations and overlap each other. One `AnimationController` class controls multiple `Animation` objects that specify the animation in a timeline (`Interval`). Now you'll walk through an example of using one `AnimationController` class and the `Interval() curve` property to start different animations at different times. As noted in the preceding section, a staggered animation uses `Interval()` to begin and end animations sequentially or to overlap one another.

TRY IT OUT Creating the Staggered Animations App

In this project, you'll re-create the balloon animation to duplicate the previous example but use only one `AnimationController` to take advantage of staggered animations. By using `Interval()`, you mark each animation's `begin` and `end` time to stagger the animations.

Like the previous project, you'll animate a balloon that starts small at the bottom of the screen, and as it inflates, it floats toward the top, giving some nice spring animations. By using a `GestureDetector` tap on the balloon, the animation reverses, showing the balloon deflating and floating downward to the bottom of the screen. Every time the balloon is tapped, the animation starts again.

1. Create a new Flutter project and name it `ch7_ac_staggered_animations`. You can follow the instructions in Chapter 4. For this project, you need to create only the `pages`, `widgets`, and `assets/images` folders.

2. Open the `pubspec.yaml` file, and under `assets`, add the `images` folder.

```
# To add assets to your application, add an assets section, like this:
assets:
  - assets/images/
```

   Add the folder `assets` and subfolder `images` at the project's root and then copy the `Beginning-GoogleFlutter-Balloon.png` file to the `images` folder.

3. Create a new Dart file under the `widgets` folder. Right-click the `widgets` folder, select New ➪ Dart File, enter **`animated_balloon.dart`**, and click the OK button to save. Import the `material.dart` library, add a new line, and then start typing **`st`**. The autocompletion help opens. Select the `stful` abbreviation and give it a name of `AnimatedBalloonWidget`.

4. It is appropriate to create the `AnimationController` and `Animation` variables before you can reference them in the code. Declare `SingleTickerProviderStateMixin` to the `_HomeState` class by adding `with SingleTickerProviderStateMixin`. The `AnimationController` constructor takes the `vsync` argument. The `vsync` argument is referenced by `this`, meaning this reference of the `_HomeState` class.

```
class _AnimatedBalloonWidgetState extends State<AnimatedBalloonWidget> with
SingleTickerProviderStateMixin {
```

5. Create one `AnimationController` only to handle the animation duration. Create two `Animations` to handle the actual movement range and growing size. Override the `initState()` and `dispose()` methods to initiate the `AnimationController` and dispose of them when the page closes.

```
class _HomeState extends State<Home> with SingleTickerProviderStateMixin {
  AnimationController _controller;
  Animation<double> _animationFloatUp;
  Animation<double> _animationGrowSize;

  @override
  void initState() {
    super.initState();
```

```
    _controller = AnimationController(duration: Duration(seconds: 4), vsync: this);
  }

  @override
  void dispose() {

    _controller.dispose();
    super.dispose();
  }
```

Note as stated in the previous section, in the `initState()` method you could call the `Animation` only after the `AnimationController` only to start the animation as page loads, but you are going to place it in the `_animatedBalloon()` method instead. The reason is to use the `MediaQuery` class to obtain the screen size to place and size the balloon accordingly. The `initState()` method does not contain the `Widget context` object that `MediaQuery` requires. However, as the `_animate-Balloon()` method is called, the animation starts on page load. If the user rotates the device, the `_animateBalloon()` method is called again, and the balloon resizes itself accordingly and runs the animation if the balloon is located at the bottom of the screen.

6. After `Widget build(BuildContext context)`, create the balloon height, width, and page bottom position by using `MediaQuery.of(context).size`. I arrived at the following formulas for calculating the dimensions by testing different options to get the best aesthetic look depending on different devices' screen size and orientation.

```
Widget build(BuildContext context) {
  double _balloonHeight = MediaQuery.of(context).size.height / 2;
  double _balloonWidth = MediaQuery.of(context).size.height / 3;
  double _balloonBottomLocation = MediaQuery.of(context).size.height -
_balloonHeight;
```

Creating an animation Tween is almost the same as the previous example except the `CurvedAnimation parent` is the `_controller`. The `curve` uses `Interval()` to mark each animation's begin and end time. Zero means `begin`, and 1.0 animation means `end`.

The `_controller duration` value is four seconds, `_animationGrowSize Interval begin` is 0.0, and `end` is 0.5. This means the balloon starts inflating as soon as the animation starts but finishes at 0.5, meaning two seconds. You can think of `Interval begin` and `end` as a percentage of the total duration of the animation.

```
void _animationFloatUp = Tween(begin: _balloonBottomLocation, end: 0.0).animate(
  CurvedAnimation(
    parent: _controller,
    curve: Interval(0.0, 1.0, curve: Curves.fastOutSlowIn),
  ),
);
```

```
void _animationGrowSize = Tween(begin: 50.0, end: _balloonWidth).animate(
  CurvedAnimation(
    parent: _controller,
    curve: Interval(0.0, 0.5, curve: Curves.elasticInOut),
  ),
);
```

7. Create the `AnimatedBuilder` only, and let's look at the high-level structural breakdown for the `AnimatedBuilder` constructor that shows how the `AnimatedBuilder` `child` (`GestureDetector` with `Image`) argument is passed to the `builder`, which is the widget to receive the `animation` argument.

```
return AnimatedBuilder(
  animation: _animationFloatUp,
  builder: (context, child) {
    return Container(
      child: child,
    );
  },
  child: GestureDetector(/* Image*/) */
);
```

Using the `AnimatedBuilder` is almost the same as the previous example with the difference that you use only one `AnimationController` only to start animation in forward or reverse with the _controller variable.

The `AnimatedBuilder` constructor takes the `animation`, `builder`, and `child` arguments.

Add to the `GestureDetector()` widget the `onTap` property. For `GestureDetector()`, the `onTap` property callback checks for `_controllerFloatUp.isCompleted` (the animation is done), and if yes, it starts the animation in reverse. This will deflate the balloon and have it start floating down to the bottom of the page. The `else` portion handles the balloon already at the bottom of the screen; it starts the animation by floating the balloon upward and inflating it back to normal size. You'll notice that since you have only one `AnimationController`, you use the `_controller` variable to reverse or forward the animation.

8. Add to the `GestureDetector` `child` a call to the `Image.asset()` constructor that loads the `BeginningGoogleFlutter-Balloon.png` file and sets the `height` value to `_balloonHeight` and the `width` value to `_balloonWidth`.

```
return AnimatedBuilder(
  animation: _animationFloatUp,
  builder: (context, child) {
    return Container(
      child: child,
      margin: EdgeInsets.only(
        top: _animationFloatUp.value,
      ),
```

```
        width: _animationGrowSize.value,
      );
    },
    child: GestureDetector(
      onTap: () {
        if (_controller.isCompleted) {
          _controller.reverse();
        } else {
          _controller.forward();
        }
      },
      child: Image.asset('assets/images/BeginningGoogleFlutter-Balloon.png',
          height: _balloonHeight, width: _balloonWidth),
    ),
  );
```

9. Open the `home.dart` file and import the `animated_balloon.dart` file to the top of the page.:

```
import 'package:flutter/material.dart';
import 'package:ch7_ac_staggered_animations/widgets/animated_balloon.dart';
```

10. Add to the body a `SafeArea` widget with `SingleChildScrollView` as a child.

11. Add `Padding` as a child of `SingleChildScrollView`. Add a `Column` widget as a child of `Padding`.

12. In the `Column` children, add the call to the method `_animateBalloon()`. Note that I am using `NeverScrollableScrollPhysics()` to stop the `SingleChildScrollView` to scroll content.

```
body: SafeArea(
  child: SingleChildScrollView(
    physics: NeverScrollableScrollPhysics(),
    child: Padding(
      padding: EdgeInsets.all(16.0),
      child: Column(
        children: <Widget>[
          AnimatedBalloonWidget(),
        ],
      ),
    ),
  ),
),
```

How It Works

Declaring `SingleTickerProviderStateMixin` to the `AnimatedBalloonWidget` widget class allows you to set the `AnimationController` `vsync` argument. The `SingleTickerProviderStateMixin` allows only one `AnimationController`. You added `AnimationController` and declared the `_controller` variable to animate both the floating upward or downward and the inflation or deflation of the balloon.

You declared the `_animationFloatUp` variable to hold the value from the Tween animation to show the balloon either floating upward or downward by setting the top margin of the `Container` widget. You also declared the `_animationGrowSize` variable to hold the value from the Tween animation to show the balloon either inflating or deflating by setting the `width` value of the `Container` widget.

The AnimatedBuilder constructor takes animation, builder, and child arguments. Next, you passed the _animationFloatUp animation to the AnimatedBuilder constructor. The AnimatedBuilder builder argument returns a Container widget with the child as an Image wrapped in a GestureDetector widget.

SUMMARY

In this chapter, you learned how to add animations to your app to improve the UX. You implemented AnimatedContainer to animate the width of a Container widget with a beautiful spring effect by using Curves.elasticOut. You added the AnimatedCrossFade widget to cross-fade between two children widgets. The color is animated from amber to green while at the same time the widget increased or decreased in width and height. To fade a widget in, out, or partially, you added the AnimatedOpacity widget. The AnimatedOpacity widget uses the opacity property passed over a period of time (Duration) to fade the widget. The AnimationController class allows the creation of custom animations.

You learned to use multiple AnimationControllers with different durations. You used two Animation classes to control the floating upward or downward and the inflation and deflation of the balloon at the same time. The animation is created by using Tween with begin and end values. You also used different CurvedAnimation class for a nonlinear effect like Curves.fastOutSlowIn to float upward or downward and Curves.elasticInOut to inflate or deflate the balloon. Lastly, you used one AnimationController class with multiple Animation classes to create staggered animations, which give a similar effect as the previous example.

In the next chapter, you'll learn the many ways of using navigation such as Navigator, Hero Animation, BottomNavigationBar, BottomAppBar, TabBar, TabBarView, and Drawer.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
`AnimatedContainer`	This gradually changes values over time.
`AnimatedCrossFade`	This cross-fades between two child widgets.
`AnimatedOpacity`	This shows or hides widget visibility by animating fading over time.
`AnimatedBuilder`	This is used to create a widget that performs a reusable animation.
`AnimationController`	This creates custom animations by using `TickerProviderStateMixin`, `SingleTickerProviderStateMixin`, `AnimationController`, `Animation`, `Tween`, and `CurvedAnimation`.