Animations in Flutter - easy guide - tutorial

June 20, 2018 in flutter

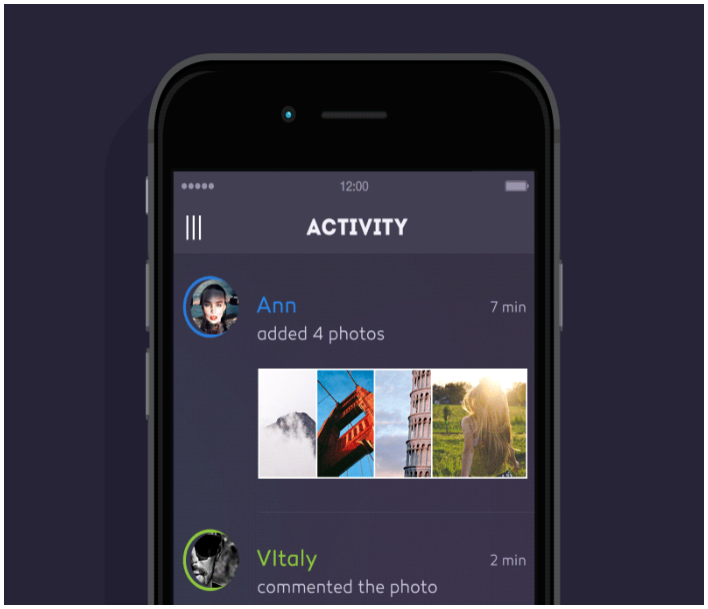
Animations in Flutter are powerful and very simple to use. Through a concrete example, you will learn everything you need to know how to build your own animations.

Difficulty: *Intermediate*

Today we cannot imagine any Mobile App without animations. When you move from one page to another, tap a Button (or InkWell)... there is an animation. Animations are everywhere.

Flutter made **animations** very easy to implement.

In very simple words, this article tackles this topic, earlier reserved to specialists and, in order to make this paper attractive, I took as a challenge the fact of implementing in Flutter, step by step, the following *Guillotine Menu* effect, posted by Vitaly Rubtsov on Dribble.



original

The first part of this article explains the theory and the main concepts. The second part is dedicated to the implementation of the animation, shown in the video here above.

The 3 pillars of an Animation

In order to have an **Animation**, the following 3 elements need to be present:

- a **Ticker**
- an **Animation**
- an **AnimationController**

Here follows an early introduction to these elements. More explanation will come later on.

The Ticker

In simple words, a *Ticker* is a class which *sends a signal* at *almost* regular interval (around 60 times per second). Think of your watch which *ticks* at each second.

At each *tick*, the *Ticker* invokes *callback* method(s) with the duration since the first tick, after it was started.

> ***IMPORTANT***
> *All tickers, even if started at different times, will* ***always be synchronized****. This is very useful to synchronize animations*

The Animation

An *Animation* is nothing else but a *value* (of a specific type) that can change over the lifetime of the animation. The way the *value* changes over the time of the animation can be linear (like 1, 2, 3, 4, 5...) or much more complex (see Curves, later).

The AnimationController

An *AnimationController* is a class that *controls* (start, stop, repeat...) an animation (or several animations). In other words, it makes the *Animation value* vary from a lowerBound to an upperBound in a certain duration, using a velocity (= rate of change of *value* per second).

The AnimationController class

This class gives the control over an animation. In order to be more precise, I should rather say "*over a scene*" since, as we will see a bit later, several distinct animations could be controlled by a same controller…

So, with this AnimationController class, we can:

- play a scene *forward*, *reverse*
- stop a scene
- set a scene to a certain *value*
- define the boundary values (*lowerBound*, *upperBound*) of a scene

The following pseudo-code shows the different initialization parameters of this class:

```
AnimationController controller = new AnimationController(
        value:          // the current value of the animation, usually 0.0 (= default)
        lowerBound:     // the lowest value of the animation, usually 0.0 (= default)
        upperBound:     // the highest value of the animation, usually 1.0 (= default)
        duration:       // the total duration of the whole animation (scene)
        vsync:          // the ticker provider
        debugLabel:     // a label to be used to identify the controller
                        // during debug session
);
```

Most of the time, *value*, *lowerBound*, *upperBound* and *debugLabel* are not mentioned when initializing an **AnimationController**.

How to bind the AnimationController to a Ticker?

In order to work, an *AnimationController* needs to be bound to a **Ticker**.

Usually, you will generate a *Ticker*, linked to an instance of a *Stateful Widget*.

```
class _MyStateWidget extends State<MyStateWidget>
                with SingleTickerProviderStateMixin {
        AnimationController _controller;

        @override
        void initState(){
```

```
 7            super.initState();
 8            _controller = new AnimationController(
 9                  duration: const Duration(milliseconds: 1000),
10                  vsync: this,
11            );
12          }
13
14          @override
15          void dispose(){
16            _controller.dispose();
17            super.dispose();
18          }
19
20          ...
21  }
```

- line 2

  you tell *Flutter* that you want to have a **new** single *Ticker*, linked to this instance of the *MyStateWidget*

- lines 8-10

  initialization of the *controller*. The total duration of a *scene* is set to 1000 milliseconds and bound to the *Ticker* (vsync: this).

  Implicit parameters are: lowerBound = 0.0 and upperBound = 1.0

- line 16

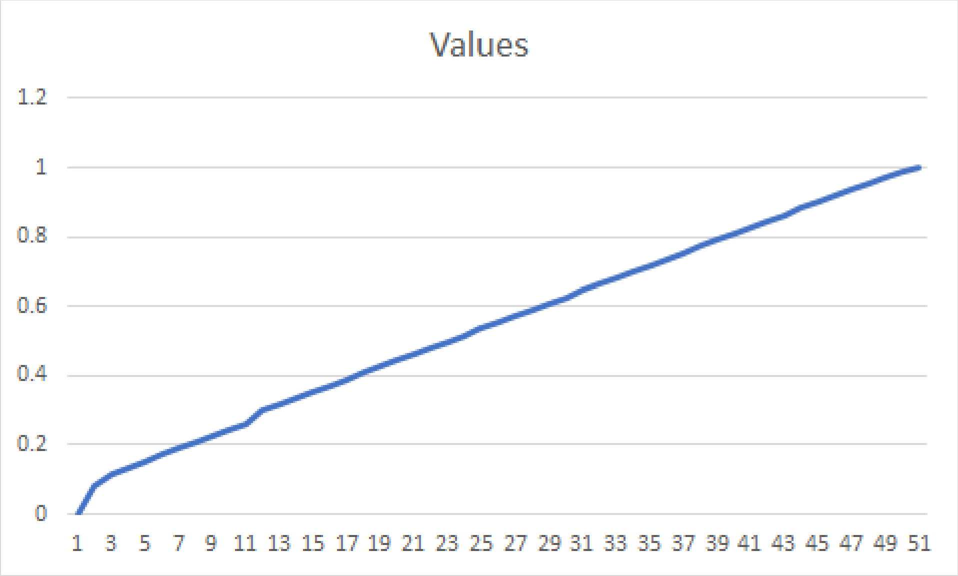
  VERY IMPORTANT, you need to release the *controller* when the instance of *MyStateWidget* is discarded.

> ***TickerProviderStateMixin or SingleTickerProviderStateMixin?***
> *If you have several AnimationController instances and you want to have distinct Tickers, replace SingleTickerProviderStateMixin by TickerProviderStateMixin.*

OK, I have the controller bound to a Ticker but how does it help?

Thanks to the *ticker*, which *ticks* around 60 times per second, the *AnimationController* **linearly** produces values from *lowerBound* to *upperBound* during a given duration.

An example of values, which are produced within these 1000 milliseconds:



We see that values vary from 0.0 (lowerBound) to 1.0 (upperBound) within 1000 milliseconds. 51 *different* values were generated.

Let's extend the code to see how to use this.

```
class _MyStateWidget extends State<MyStateWidget>
                with SingleTickerProviderStateMixin {
        AnimationController _controller;

        @override
        void initState(){
          super.initState();
          _controller = new AnimationController(
                duration: const Duration(milliseconds: 1000),
                vsync: this,
          );
          _controller.addListener((){
                  setState((){});
14            });
15            _controller.forward();
16          }
17
18          @override
19          void dispose(){
20            _controller.dispose();
21            super.dispose();
22          }
23
24          @override
25          Widget build(BuildContext context){
26                  final int percent = (_controller.value * 100.0).round();
27                  return new Scaffold(
28                          body: new Container(
29                                  child: new Center(
30                                          child: new Text('$percent%'),
31                                  ),
32                          ),
33                  );
34          }
35  }
```

- line 12

  this line tells the controller that each time its value changes, we need to rebuild the Widget (via the *setState()*)

- line 15

  as soon as the Widget initialization is complete, we tell the controller to start counting (*forward()* -> from the *lowerBound* to the *upperBound*)

- line 26

  we retrieve the value of the controller (*_controller.value*) and, as in this example this value ranges 0.0 to 1.0 (0% to 100%), we get the integer expression of this percentage, to be displayed at the center of the page.

The notion of Animation

As we just saw, the *controller* returns a series of **decimal values** which vary from each other in a **linear** way.

Sometimes, we would like to:

- use other **types** of *values*, such as **Offset**, **int**…
- use of range of values different than 0.0 to 1.0
- consider other **variation** types, other than linear to give some effect

Use of other value types

In order to be able to use other value **types**, the **Animation** class uses *templates*.

In other words, you may define:

```
Animation<int> integerVariation;
Animation<double> decimalVariation;
Animation<Offset> offsetVariation;
```

Use of different value range

Sometime, we would like to have a variation between 2 values, different than 0.0 and 1.0.

In order to define such range, we will use the **Tween** class.

To illustrate this, let's consider the case where you would like to have an angle that varies from 0 to π/2 rad.

```
Animation<double> angleAnimation = new Tween(begin: 0.0, end: pi/2);
```

Variation types

As already said, the default way of varying values from *lowerBound* to *upperBound* is **linear**.

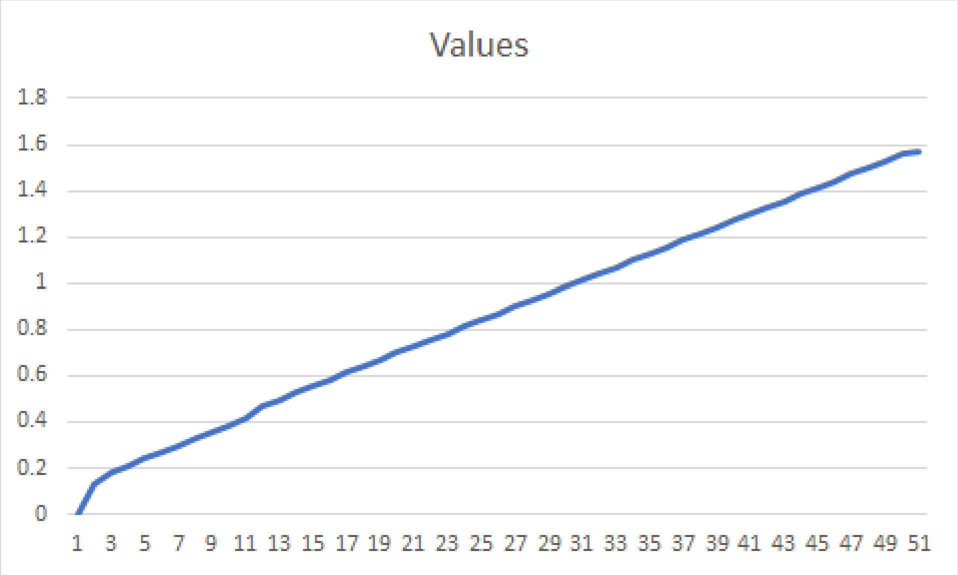
As already said, the default way of varying values from *lowerBound* to *upperBound* is linear, which is the way the *controller* works.

If you want to have the angle that linearly varies from 0 to π/2 radians, bind the *Animation* to the *AnimationController*:

```
Animation<double> angleAnimation = new Tween(begin: 0.0, end: pi/2).animate(_controller);
```

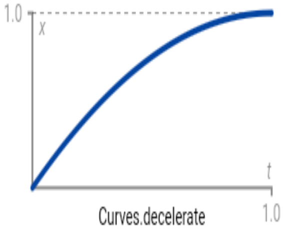
When you will start the animation (via *_controller.forward()*), the *angleAnimation.value* will use the *_controller.value* to interpollate the range [0.0; π/2].

The following graph shows such linear variation (π/2 = 1.57)

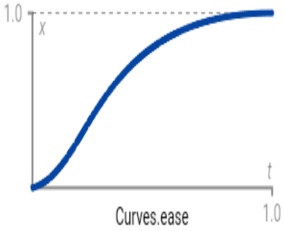


Using Flutter pre-defined Curved variations

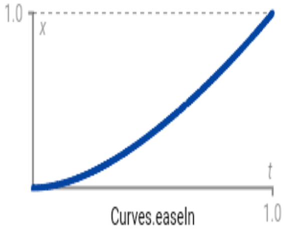
Flutter offers a set of pre-defined Curved variations. The list is shown here below:



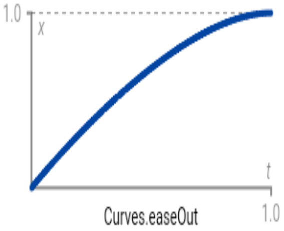
decelerate



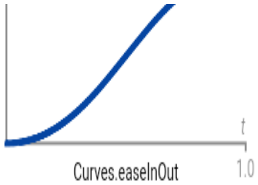
ease



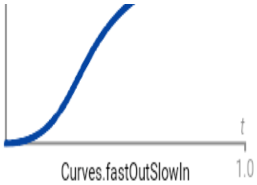
easeIn



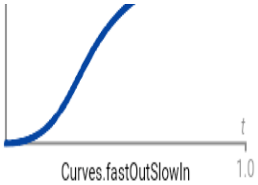
easeOut



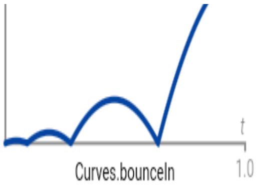
easeInOut



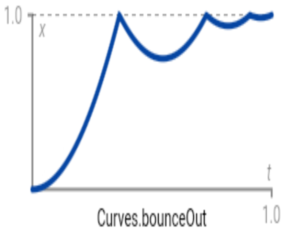
fastOutSlowIn



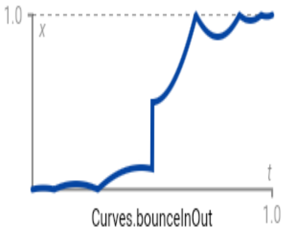
fastOutSlowIn



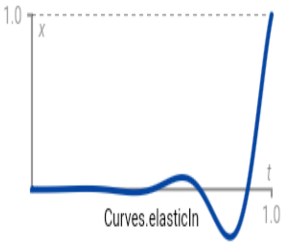
bounceIn



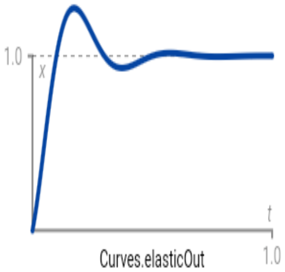
bounceOut



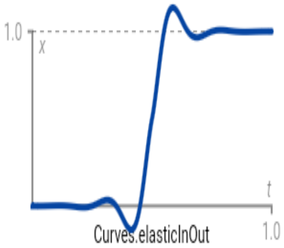
bounceInOut



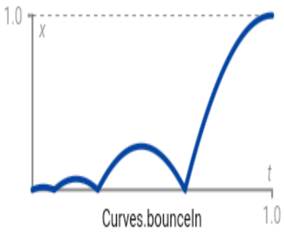
elasticIn



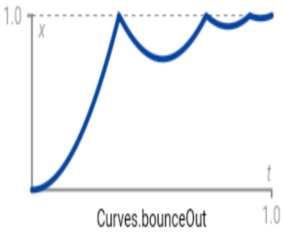
elasticOut



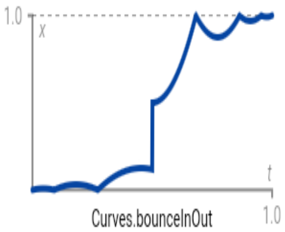
elasticInOut



bounceIn



bounceOut



bounceInOut

To use these variations:

```
Animation<double> angleAnimation = new Tween(begin: 0.0, end: pi/2).animate(
        new CurvedAnimation(
                parent: _controller,
                curve:  Curves.ease,
                reverseCurve: Curves.easeOut
        ));
```

This creates a variation of value [0; π/2], which varies using the

- **Curves.ease** when the animation goes 0.0 -> π/2 (= forward)
- **Curves.easeOut** when the animation goes π/2 -> 0.0 (= reverse)

Controlling the animation

The **AnimationController** is the class that allows you to take the *control* over the animation, through an API. (Here is the most commonly used API):

- *_controller.forward({ double from })*

  asks the *controller* to start varying the values from *lowerBound* -> *upperBound*

  The optional argument *from* may be used to force the *controller* to start "*counting*" from another value than the *lowerBound*

- *_controller.reverse({ double from })*

  asks the *controller* to start varying the values from *upperBound* -> *lowerBound*

  The optional argument *from* may be used to force the *controller* to start "*counting*" from another value than the *upperBound*

- *_controller.stop({ bool canceled: true })*

  stops running the animation

- *_controller.reset()*

  resets the animation to *lowerBound*

- *_controller.animateTo(double target, { Duration duration, Curve curve: Curves.linear })*

  drives the animation from its current value to the *target*

- *_controller.repeat({ double min, double max, Duration period })*

  starts running the animation in the forward direction, and restarts the animation when it completes.

  If defined, *min* and *max* limit the number of times the repeat occurs.

Let’s be safe…

Since an animation could be stopped unexpectedly (e.g. the screen is dismissed), when using one of these APIs, it is safer to add the “*.orCancel*“:

```
__controller.forward().orCancel;
```

Thanks to this little *trick*, no exception will be thrown if the *Ticker* is cancelled *before* the *_controller* is disposed.

The notion of Scene

This word “*scene*” does not exist in the official documentation but personally, I find it closer to the reality. Let me explain.

As I said, one **AnimationController** manages an *Animation*. However, we may understand the word “*Animation*” as a series of *sub-animations* which need to be played in *sequence* or with *overlap*. The definition on how we chain the *sub-animations* together, it is what I call a “*scene*”.

Consider the following case where a whole duration of an animation would be 10 seconds and that we would like:

- the first 2 seconds, a ball moves from the left side to the middle of the screen
- then, the same ball takes 3 seconds to move from the center to the top-center of the screen
- finally, the ball takes 5 seconds to fade out.

As you most probably already imagine, we have to consider 3 distinct animations:

```
///
/// Definition of the _controller with a whole duration of 10 seconds
///
AnimationController _controller = new AnimationController(
        duration: const Duration(seconds: 10),
        vsync: this
);
```

```
///
/// First animation that moves the ball from the left to the center
///
Animation<Offset> moveLeftToCenter = new Tween(
        begin: new Offset(0.0, screenHeight /2),
        end: new Offset(screenWidth /2, screenHeight /2)
).animate(_controller);

///
/// Second animation that moves the ball from the center to the top
///
Animation<Offset> moveCenterToTop = new Tween(
        begin: new Offset(screenWidth /2, screenHeight /2),
        end: new Offset(screenWidth /2, 0.0)
).animate(_controller);

///
/// Third animation that will be used to change the opacity of the ball to make it
disappear
///
Animation<double> disappear = new Tween(
        begin: 1.0,
        end: 0.0
).animate(_controller);
```

Now the question, how do we chain (or orchestrate) the sub-animations?

The notion of Interval

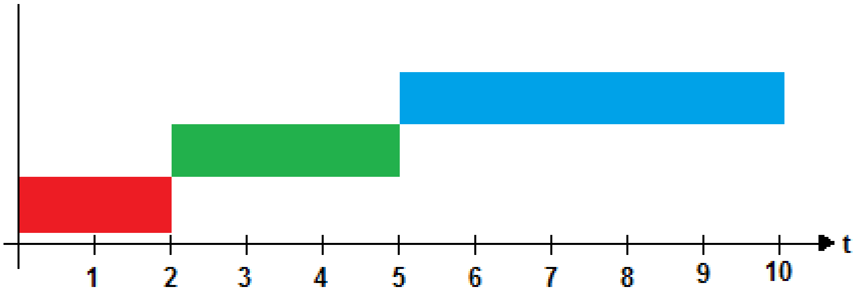
The answer is given by the use of the **Interval** class. But what is an *interval*?

> *In contradiction with the first thing that could pop up our mind, an interval does **NOT** relate to a time interval but to a **range of values**.*

If you consider the *_controller*, you have to remember that it makes a value vary from a **lowerBound** to an **upperBound**.

Usually, these 2 values are respectively kept to **lowerBound = 0.0** and **upperBound = 1.0** and this makes things much easier to consider since [0.0 -> 1.0] is nothing else but a variation from 0% to 100%. So, if the total duration of a *scene* is 10 seconds, it is most than probable that after 5 seconds, the corresponding *_controller.value* will be very close to 0.5 (= 50%).

If we put the 3 distinct animations on a timeline, we obtain:



If we now consider the intervals of values, for each of the 3 animations, we get:

- moveLeftToCenter

  duration: 2 seconds, begins at 0 second, ends at 2 seconds => range = [0;2] => percentages: from 0% to 20% of the whole scene => [0.0;0.20]

- moveCenterToTop

  duration: 3 seconds, begins at 2 seconds, ends at 5 seconds => range = [2;5] => percentages: from 20% to 50% of the whole scene => [0.20; 0.50]

- disappear

  duration: 5 seconds, begins at 5 seconds, ends at 10 seconds => range = [5;10] => percentages: from 50% to 100% of the whole scene => [0.50;1.0]

Now that we have these percentages, we may update the definition of each individual animation as follows:

```
///
/// Definition of the _controller with a whole duration of 10 seconds
///
AnimationController _controller = new AnimationController(
    duration: const Duration(seconds: 10),
        duration: const Duration(seconds: 10),
        vsync: this
);

///
/// First animation that moves the ball from the left to the center
///
Animation<Offset> moveLeftToCenter = new Tween(
        begin: new Offset(0.0, screenHeight /2),
        end: new Offset(screenWidth /2, screenHeight /2)
        ).animate(
            new CurvedAnimation(
                parent: _controller,
                curve:  new Interval(
                    0.0,
                    0.20,
                    curve: Curves.linear,
                ),
            ),
        );

///
/// Second animation that moves the ball from the center to the top
///
Animation<Offset> moveCenterToTop = new Tween(
        begin: new Offset(screenWidth /2, screenHeight /2),
        end: new Offset(screenWidth /2, 0.0)
        ).animate(
            new CurvedAnimation(
                parent: _controller,
                curve:  new Interval(
                    0.20,
                    0.50,
                    curve: Curves.linear,
                ),
            ),
        );

///
/// Third animation that will be used to change the opacity of the ball to make it
disappear
///
Animation<double> disappear = new Tween(begin: 1.0, end: 0.0)
```

```
    .animate(
        new CurvedAnimation(
            parent: _controller,
            curve:  new Interval(
                0.50,
                1.0,
                curve: Curves.linear,
            ),
        ),
    );
```

That’s all you need to set-up to define a *scene* (or a series of animations). Of course, nothing prevents you from overlapping the sub-animations…

Responding to the Animation State

Sometimes, it is convenient to know the status of an animation (or scene).

An animation may have 4 distinct statuses:

- *dismissed*: the animation is stopped at the beginning (or has not started yet)
- *forward*: the animation is running from beginning to the end
- *reverse*: the animation is running backwards, from end to beginning
- *completed*: the animation is stopped at the end

To get this status, we need to listen to the animation status changes, the following way:

```
myAnimation.addStatusListener((AnimationStatus status){
    switch(status){
        case AnimationStatus.dismissed:
            ...
            break;

        case AnimationStatus.forward:
            ...
            break;

        case AnimationStatus.reverse:
            ...
            break;

        case AnimationStatus.completed:
            ...
            break;
    }
});
```

A typical use if this status is a *toggle*. For example, once an animation completes, we want to reverse it. To achieve this:

```
myAnimation.addStatusListener((AnimationStatus status){
    switch(status){
        ///
        /// When the animation is at the beginning, we force the animation to play
        ///
        case AnimationStatus.dismissed:
            _controller.forward();
            break;

        ///
        /// When the animation is at the end, we force the animation to reverse
        ///
        case AnimationStatus.completed:
            _controller.reverse();
            break;
    }
});
```

Enough theory, let’s practice now !

Now that the theory has been introduced, it is time to practice…

As I mentioned at the beginning of this article, I am now going to put this notion of animation in practice by implementing an animation, called “*guillotine*”.

Analysis of the animations and initial skeleton

To have this *guillotine* effect, we initially need to consider:

- the page content itself
- a menu bar that rotates when we hit the *menu* (or *hamburger*) icon
- when rotating *in*, the menu overlaps the page content and fills in the whole viewport
- once the menu is fully visible and we hit the *menu* icon again, the menu rotates *out* in order to get back to its original position and dimensions

From these observations, we can immediately derive that we are not using a normal *Scaffold* with an *AppBar* (since the latter is fixed).

We will rather use a *Stack* of 2 layers:

- the page content (lower layer)
- the menu (upper layer)

Let’s first build this skeleton:

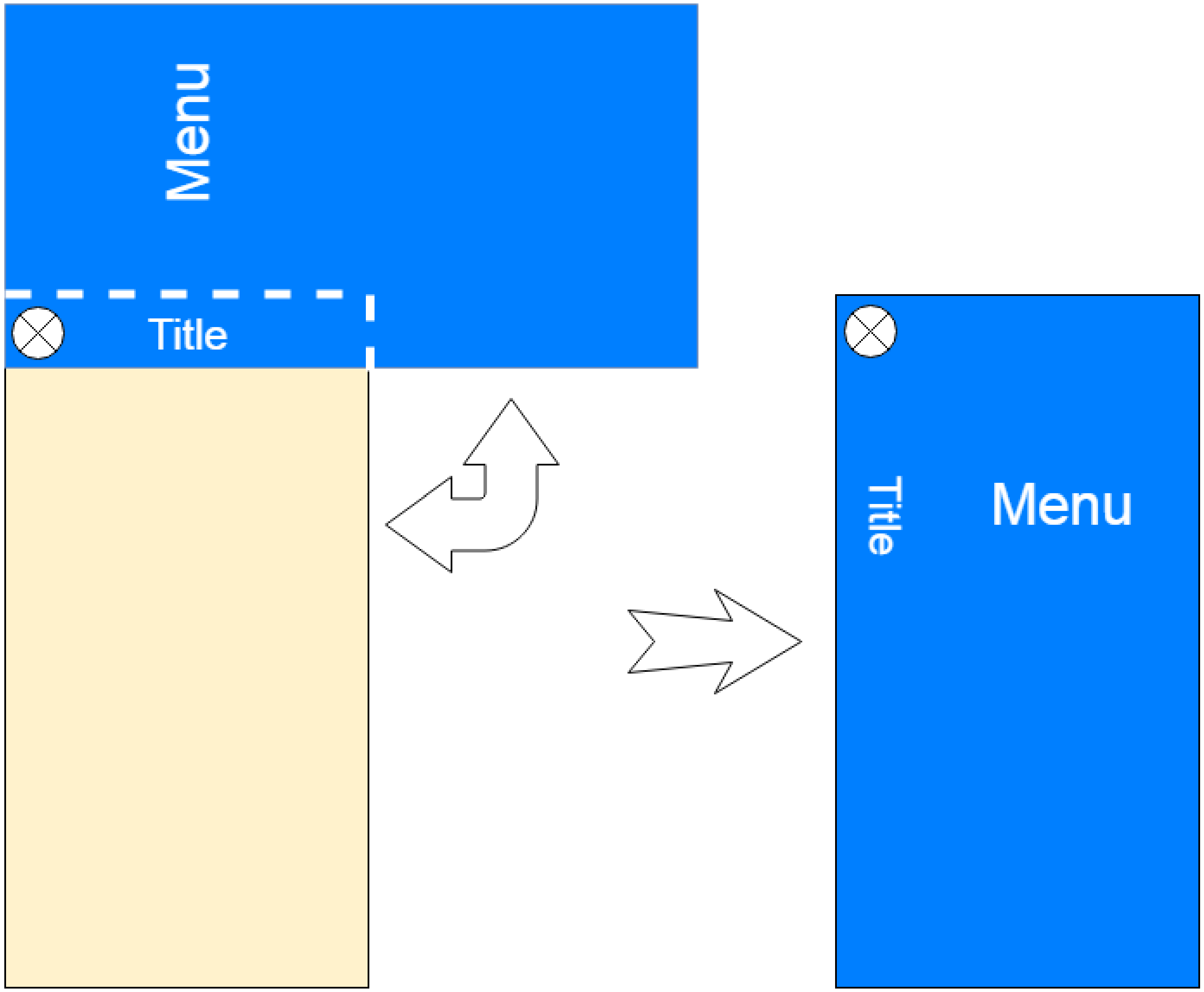
```
class MyPage extends StatefulWidget {
    @override
    _MyPageState createState() => new _MyPageState();
}

class _MyPageState extends State<MyPage>{
  @override
  Widget build(BuildContext context){
      return SafeArea(
        top: false,
        bottom: false,
        child: new Container(
          child: new Stack(
            alignment: Alignment.topLeft,
            children: <Widget>[
              new Page(),
              new GuillotineMenu(),
            ],
          ),
        ),
      );
  }
}

class Page extends StatelessWidget {
    @override
    Widget build(BuildContext context){
        return new Container(
            padding: const EdgeInsets.only(top: 90.0),
            color: Color(0xff222222),
        );
    }
}

class GuillotineMenu extends StatefulWidget {
    @override
    _GuillotineMenuState createState() => new _GuillotineMenuState();
}

class _GuillotineMenuState extends State<GuillotineMenu> {

    @overrride
    Widget build(BuildContext context){
        return new Container(
            color: Color(0xff333333),
        );
    }
}
```

The outcome of this code gives a black screen, only revealing the *GuillotineMenu*, covering the whole viewport.

Analysis of the menu itself

If you have a closer look to the video, you can see that when the menu is fully open, it entirely covers the viewport. When it is open, only something like an *AppBar* is visible.

Nothing prevents us from seeing the things differently... and what if the *GuillotineMenu* would initially be rotated and when we hit the *menu* button, we rotate it of π/2, as shown in the following picture?



We can then rewrite the *_GuillotineMenuState* class as follows: *(no explanation is given on the way to build the layout, since this is not the objective of this article)*

```
class _GuillotineMenuState extends State<GuillotineMenu> {
   double rotationAngle = 0.0;

    @override
    Widget build(BuildContext context){
        MediaQueryData mediaQueryData = MediaQuery.of(context);
                 double screenWidth = mediaQueryData.size.width;
                 double screenHeight = mediaQueryData.size.height;

                 return new Material(
                         color: Colors.transparent,
                         child: new Transform.rotate(
                                 angle: rotationAngle,
              origin: new Offset(24.0, 56.0),
              alignment: Alignment.topLeft,
              child: Container(
                  width: screenWidth,
                  height: screenHeight,
                  color: Color(0xFF333333),
                  child: new Stack(
                      children: <Widget>[
                          _buildMenuTitle(),
                          _buildMenuIcon(),
                          _buildMenuContent(),
                      ],
                  ),
              ),
          ),
      );
}

  ///
  /// Menu Title
  ///
  Widget _buildMenuTitle(){
      return new Positioned(
          top: 32.0,
          left: 40.0,
          width: screenWidth,
          height: 24.0,
          child: new Transform.rotate(
              alignment: Alignment.topLeft,
              origin: Offset.zero,
              angle: pi / 2.0,
              child: new Center(
              child: new Container(
                  width: double.infinity,
                  height: double.infinity,
                  child: new Opacity(
                  opacity: 1.0,
                  child: new Text('ACTIVITY',
                      textAlign: TextAlign.center,
                      style: new TextStyle(
                          color: Colors.white,
                          fontSize: 20.0,
                          fontWeight:
```

```
                                                        fontWeight: FontWeight.bold,
                                                        letterSpacing: 2.0,
                                                )),
                                        ),
                                ),
                        )),
                );
        }

        ///
        /// Menu Icon
        ///
        Widget _buildMenuIcon(){
                return new Positioned(
                        top: 32.0,
                        left: 4.0,
                        child: new IconButton(
                                icon: const Icon(
                                        Icons.menu,
                                        color: Colors.white,
                                ),
                                onPressed: (){},
                        ),
                );
        }

        ///
        /// Menu content
        ///
        Widget _buildMenuContent(){
                final List<Map> _menus = <Map>[
                        {
                        "icon": Icons.person,
                        "title": "profile",
                        "color": Colors.white,
                        },
                        {
                        "icon": Icons.view_agenda,
                        "title": "feed",
                        "color": Colors.white,
                        },
                        {
```

```
            "icon": Icons.swap_calls,
            "title": "activity",
            "color": Colors.cyan,
            },
            {
            "icon": Icons.settings,
            "title": "settings",
            "color": Colors.white,
            },
        ];

        return new Padding(
            padding: const EdgeInsets.only(left: 64.0, top: 96.0),
            child: new Container(
                width: double.infinity,
                height: double.infinity,
                child: new Column(
                    mainAxisAlignment:
MainAxisAlignment.start,
                    children: _menus.map((menuItem) {
                        return new ListTile(
                            leading: new Icon(
                            menuItem["icon"],
                            color:
menuItem["color"],
                            ),
                            title: new Text(
                            menuItem["title"],
                            style: new TextStyle(
                                color:
menuItem["color"],
                                fontSize:
24.0),
                            ),
                        );
                    }).toList(),
                ),
            ),
        );
    }
}
```

- Lines 12-15

  these lines define the rotation of the *Guillotine Menu*, around a *rotation center* (the position of the *menu icon*)

Now the outcome of this code gives an unrotated menu screen (since *rotationAngle* = 0.0), that shows the title vertically displayed.

Let's animate the menu

If you update the value of *rotationAngle* (between -π/2 and 0), you will see the menu, rotated by the corresponding angle.

Let's put some animation…

As explained earlier, we need

- a *SingleTickerProviderStateMixin*, since we have only 1 *scene*
- an *AnimationController*
- an *Animation* to have a angle variation

The code then becomes:

```
class _GuillotineMenuState extends State<GuillotineMenu>
        with SingleTickerProviderStateMixin {

    AnimationController animationControllerMenu;
    Animation<double> animationMenu;

        ///
        /// Menu Icon, onPress() handling
        ///
    _handleMenuOpenClose(){
        animationControllerMenu.forward();
    }

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

        ///
        /// Initialization of the animation controller
        ///
        animationControllerMenu = new AnimationController(
                        duration: const Duration(milliseconds: 1000),
                        vsync: this
                )..addListener((){
            setState((){});
        });

        ///
        /// Initialization of the menu appearance animation
        ///
        _rotationAnimation = new Tween(
                        begin: -pi/2.0,
                        end: 0.0
                ).animate(animationControllerMenu);
    }

    @override
    void dispose(){
        animationControllerMenu.dispose();
        super.dispose();
    }

    @override
    Widget build(BuildContext context){
        MediaQueryData mediaQueryData = MediaQuery.of(context);
                double screenWidth = mediaQueryData.size.width;
                double screenHeight = mediaQueryData.size.height;

                return new Material(
                        color: Colors.transparent,
                        child: new Transform.rotate(
                                angle: animationMenu.value,
                                origin: new Offset(24.0, 56.0),
                                alignment: Alignment.topLeft,
                                child: Container(
                                        width: screenWidth,
                                        height: screenHeight,
                                        color: Color(0xFF333333),
                                        child: new Stack(
                                                children: <Widget>[
                                                        _buildMenuTitle(),
                                                        _buildMenuIcon(),
                                                        _buildMenuContent(),
                                                ],
                                        ),
                                ),
                        ),
                );
    }

        ...
        ///
        /// Menu Icon
        ///
        Widget _buildMenuIcon(){
                return new Positioned(
                        top: 32.0,
                        left: 4.0,
                        child: new IconButton(
                                icon: const Icon(
                                        Icons.menu,
                                        color: Colors.white,
                                ),
                                onPressed: _handleMenuOpenClose,
                        ),
                );
        }
        ...
}
```

OK, when we press the *menu* button, the menu opens but does not close when we press the button again. Here comes the role of the *AnimationStatus*.

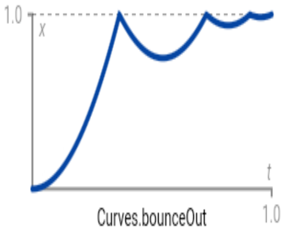
Let’s add a listener and based on the *AnimationStatus*, decide whether to run the animation forward or reverse.

```
///
/// Menu animation status
/// Menu animation status
///
enum _GuillotineAnimationStatus { closed, open, animating }

class _GuillotineMenuState extends State<GuillotineMenu>
        with SingleTickerProviderStateMixin {
    AnimationController animationControllerMenu;
    Animation<double> animationMenu;
    _GuillotineAnimationStatus menuAnimationStatus =
_GuillotineAnimationStatus.closed;

    _handleMenuOpenClose(){
        if (menuAnimationStatus == _GuillotineAnimationStatus.closed){
            animationControllerMenu.forward().orCancel;
        } else if (menuAnimationStatus == _GuillotineAnimationStatus.open) {
            animationControllerMenu.reverse().orCancel;
        }
    }

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

        ///
        /// Initialization of the animation controller
        ///
        animationControllerMenu = new AnimationController(
                         duration: const Duration(milliseconds: 1000),
                         vsync: this
                ))..addListener((){
            setState((){});
        })..addStatusListener((AnimationStatus status) {
            if (status == AnimationStatus.completed) {
                ///
                /// When the animation is at the end, the menu is open
                ///
              menuAnimationStatus = _GuillotineAnimationStatus.open;
            } else if (status == AnimationStatus.dismissed) {
                ///
                /// When the animation is at the beginning, the menu is closed
                ///
              menuAnimationStatus = _GuillotineAnimationStatus.closed;
            } else {
                ///
                /// Otherwise the animation is running
                ///
              menuAnimationStatus = _GuillotineAnimationStatus.animating;
            }
          });

        ...
    }
...
}
```

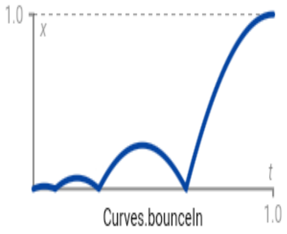
The menu is now opening or closing as expected but the video shows us a opening/closing movement which is not linear but looks like a bouncing effect. Let’s add this effect.

For this I will choose the following 2 effects:

- *bounceOut* when the menu is opening
- *bounceIn* when the menu is closing



bounceOut



bounceIn

```
class _GuillotineMenuState extends State<GuillotineMenu>
        with SingleTickerProviderStateMixin {
...
    @override
    void initState(){
        ...
        ///
        /// Initialization of the menu appearance animation
        ///
        animationMenu = new Tween(
                begin: -pi / 2.0,
                end: 0.0
        ).animate(new CurvedAnimation(
                parent: animationControllerMenu,
                curve: Curves.bounceOut,
                reverseCurve: Curves.bounceIn,
        ));
    }
...
}
```

There is still something that misses in this implementation... the fact that the title disappears when opening the menu and gets back when closing it. This is a face in/out effect, to be processed as an animation as well. Let’s add it.

```
class _GuillotineMenuState extends State<GuillotineMenu>
    with SingleTickerProviderStateMixin {
  AnimationController animationControllerMenu;
  Animation<double> animationMenu;
  Animation<double> animationTitleFadeInOut;
  _GuillotineAnimationStatus menuAnimationStatus;

...
  @override
  void initState(){
        ...
    ///
    /// Initialization of the menu title fade out/in animation
    ///
    animationTitleFadeInOut = new Tween(
                begin: 1.0,
                end: 0.0
        ).animate(new CurvedAnimation(
        parent: animationControllerMenu,
        curve: new Interval(
                0.0,
                0.5,
                curve: Curves.ease,
        ),
    ));
  }
...
  ///
  /// Menu Title
  ///
  Widget _buildMenuTitle(){
    return new Positioned(
          top: 32.0,
          left: 40.0,
          width: screenWidth,
          height: 24.0,
          child: new Transform.rotate(
                alignment: Alignment.topLeft,
                origin: Offset.zero,
                angle: pi / 2.0,
                child: new Center(
                  child: new Container(
                        width: double.infinity,
                        height: double.infinity,
                          child: new Opacity(
                            opacity: animationTitleFadeInOut.value,
                                child: new Text('ACTIVITY',
                                        textAlign: TextAlign.center,
                                        style: new TextStyle(
                                                color: Colors.white,
                                                fontSize: 20.0,
                                                fontWeight: FontWeight.bold,
                                                letterSpacing: 2.0,
                                        )),
                                ),
                        ),
                )),
        );
  }
...
}
```

Result

Here is the result I obtain, which is very close to the original, isn't it?

result

Conclusions

As you saw, it is very simple to build animations, even complex ones.

I hope that this quite long article succeeded in demystifying the notion of **Animations** in Flutter.

Stay tuned for next articles and happy coding.

TAGGED IN

flutter animation

didierboelens Comment Policy

All comments are welcome. Non-relevant comments will be removed.

1 Comment **didierboelens** 1 **Login**

♡ **Recommend** 1 **Share** **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

swapnil kumar • 13 days ago

I have followed the steps but the problem is how to get touch events on the main page below i am unable to get the events

• Reply • Share ›

ALSO ON **DIDIERBOELENS**

Widget - State - Context - InheritedWidget

1 comment • 3 months ago

Avatar Enokas Sakone — Merci, tres bien expliqué.

Flutter - Bienvenue

2 comments • 3 months ago

Avatar Enokas Sakone — Merci :)

Flutter - Range Slider

4 comments • a month ago

Avatar Enokas Sakone — cool

Animations en Flutter - guide facile - tutoriel

2 comments • 2 months ago

Avatar Enokas Sakone — Merci

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Disqus' Privacy PolicyPrivacy PolicyPrivacy**