Purpose

This document lays out all the ways Dribble Game Mobile, an NBA trivia game built in Flutter, uses animations. This is to illustrate in-production usage patterns and to highlight pain points with animations in the Flutter framework.

This document tries to be as un-opinionated as possible and does not present solutions, only a catalog of real-world use cases.

About Dribble Game

Dribble Game is a ~2 year old NBA trivia app that currently supports 1.5k daily active users. It was built and maintained by one full time experienced (including 3 YoE in Flutter specifically) full stack developer and one part time product manager/data scientist.
The app has 5 form-like (as opposed to Flame based rendered worlds) trivia game modes with a lot of animations used for a wide range of user interaction and feedback.
Dribble Game is unfortunately not open source, but I've shared relevant code snippets for each animation use case.

Animation Use Cases

State Transitions

A UI element appears (`AnimatedWidget.animate(autoplay: true)`), is replaced (`AnimatedSwitcher`), or changes its properties (`AnimatedContainer`, `AnimatedAlign`, `AnimatedOpacity`, etc) in response to a direct or indirect user interaction.
These are both the most common and the most complicated animations used in Dribble Game. Animations are all implicit and managed by the relevant built-in implicitly animated widget with the one exception of `FlippyBoi` (I'm a serious developer writing serious code, I swear), a custom widget that wraps a controller based imperative card flip animation and implicitly triggers it on transition change. See details in examples.

Relevant Widget frequency:
AnimatedSwitcher (35), AnimatedContainer (14), AnimatedOpacity (12), AnimateWidgetExtension.animate() (10), AnimatedFractionallySizedBox (3), FlippyBoi (3)

Examples:

A. Three different footer elements (nothing, a live score panel, and a "next game" button) slide up and back out from the bottom of the screen depending on game state (animating in, actively playing, finished)

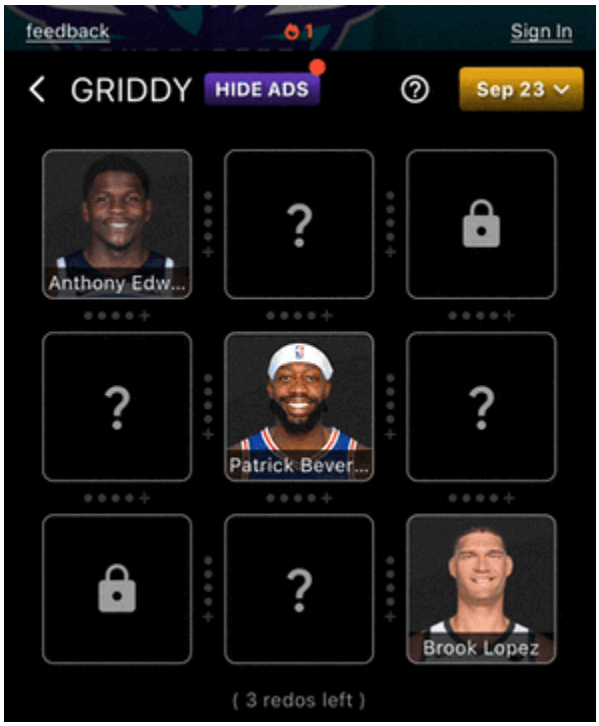
```
AnimatedSwitcher(
  duration: kDuration,
```

```
  transitionBuilder: (child, animation) {
    return SizeTransition(
      axisAlignment: -1,
      sizeFactor: Tween<double>(begin: 0, end: 1).animate(animation),
      child: child,
    );
  },
  // `animateCards` is true while the new round animation is playing
  // and `isPlaying` is true until a game finishes (win or loss).
  child: swipeball.animateCards
      ? const SizedBox()
      : swipeball.isPlaying
          // A panel showing the users guesses and score.
          ? const OutcomeRow()
          : Padding(
              padding: EdgeInsets.only(
                top: context.gutterSmall,
              ),
              // `Next Game` and `Share` buttons.
              child: const _UnderCardFooter(),
            ),
);
```

B. Displayed text is temporarily replaced with the message “copied to clipboard” when the user taps on it or the copy button next to it

```
AnimatedSwitcher(
  duration: kDuration,
  // `showMessage` is set to true in `onTap` and then reset to false one
  // second later.
  child: showMessage
      ? _CopiedMessage(text: effectiveText)
      : Builder(
          builder: (context) {
            final Widget child = Text(
              effectiveText,
              textAlign: TextAlign.left,
            );
            if (widget.mainAxisSize == MainAxisSize.max) {
              return SingleChildScrollView(
                scrollDirection: Axis.horizontal,
                child: child,
              );
            }
            return child;
          },
        ),
);
```

C. A card containing a `Player?` data class flips whenever its contained `Player?` attribute changes (the user made a guess, started a new round with a preset players, reset the board, remade their guess, etc)



```
late final Widget square;
// A LOT was snipped for brevity here, but this is the general idea.
if (isSelected) {
  square = PlayerInputSquare();
} else if (player != null) {
  square = DecoratedSquare(
    key: ValueKey(player),
    child: _PlayerSquareContent(player: player),
  );
} else {
  final bool isUnlocked = connectedNeighbors != 0;
  square = DecoratedSquare(
    glassy: false,
    onTap: isUnlocked ? onTap : null,
    margin: EdgeInsets.all(context.gutterSmall * 1.5),
    child: _EmptySquareContent(isUnlocked: isUnlocked),
  );
}
return AnimatedSwitcher(
  duration: kDurationSlow,
  transitionBuilder: flipTransitionBuilder,
  child: square,
);


Widget flipTransitionBuilder<T>(
  Widget child,
  Animation<double> animation, [
  // The key of the widget that is “underneath” in the switch stack.
  ValueKey<T>? underKey,
]) {
  final Animation<double> rotateAnim =
      Tween<double>(begin: math.pi, end: 0).animate(animation);
  return ListenableBuilder(
    listenable: rotateAnim,
    child: child,
    builder: (context, child) {
      final bool isUnder = underKey != child!.key;
      final double tilt = ((animation.value - 0.5).abs() - 0.5) *
          0.003 *
          (isUnder ? -1.0 : 1.0);
      final double value =
          isUnder ? math.min(rotateAnim.value, math.pi / 2) :
rotateAnim.value;
      // Children may not have opaque backgrounds so we need to make
      // the inactive widget transparent so it doesn’t show through.
      return Opacity(
```

```
        opacity: animation.value > .5 ? 1 : 0,
        child: Transform(
          transform: Matrix4.rotationY(value)..setEntry(3, 1, tilt),
          child: child,
          alignment: Alignment.center,
        ),
      );
    },
  );
}
```

D. A series of dots “light up” or turn off as a score value changes (user made a guess, user reset the board, user started a new round, etc) between 0 and 5+ (refer to vertical and horizontal dots in Example C’s gif above).

```
class _DotLineState extends State<DotLine> {
  late final ValueNotifier<int> counter = ValueNotifier(widget.value)
    ..addListener(() {
      setState(() {});
    });

  @override
  void didUpdateWidget(covariant DotLine oldWidget) {
    if (oldWidget.value == widget.value) {
      return;
    }
    final int effectiveValue =
        widget.value.clamp(0, widget.maxValue + 1).toInt();
    // `animateTo` is an extension on `ValueNotifier<int>` that
increments it
    // towards a new value in discrete steps over time.
    // So when the external value changes, we change the internal value
step by
    // step and the corresponding dots light up/turn off.
    // So there are two layers of declarative animation here: the value
notifier
    // animating between values and the individual dots animating
between the
    // on/off state.
    unawaited(counter.animateTo(kDurationFast,
effectiveValue).then((value) {
      if (!mounted) {
        return;
      }
      setState(() {});
    }));
    super.didUpdateWidget(oldWidget);
  }
```

```
  @override
  Widget build(BuildContext context) {
    final Color inactiveColor = Color.lerp(
      Theme.of(context).colorScheme.outline,
      Theme.of(context).colorScheme.surface,
      .8,
    )!;
    final Color defaultActiveColor =
        Theme.of(context).colorScheme.primaryContainer;
    return AnimatedSwitcher(
      duration: kDurationFast,
      child: widget.isVisible
          ? Flex(
              direction: widget.axis,
              mainAxisSize: MainAxisSize.min,
              children: [
                ...List<Widget>.generate(
                  widget.maxValue,
                  (i) {
                    final Color effectiveColor =
                        i >= counter.value ? inactiveColor :
defaultActiveColor;
                    return Dot(
                      color: effectiveColor,
                      isLit: i < counter.value,
                    );
                  },
                ),
                if (widget.showOverflow)
                  _PlusSign(
                    size: context.gutterSmall,
                    isActive: counter.value > widget.maxValue,
                    activeColor: defaultActiveColor,
                    inactiveColor: inactiveColor,
                  ),
              ].separated(const GutterTiny()),
            )
          : Container(),
    );
  }

...

// This is the dot which uses animated container to animate on/off.
return AnimatedContainer(
  duration: kDurationFast,
```

```
  decoration: BoxDecoration(
    color: color,
    shape: BoxShape.circle,
    boxShadow: [
      if (isLit)
        BoxShadow(
          color: kGlowShadowColor,
          blurRadius: context.gutterTiny,
        ),
    ],
  ),
  height: 1.5 * context.gutterTiny,
  width: 1.5 * context.gutterTiny,
);
```

E. As a new round starts, a deck of cards is “shuffled” onto the screen by sliding in from the top (with slight random variations in approach angle). A series of other elements animate in when the shuffle finishes so the currently animating state is held as a bool in the relevant view model. The gif crunches the frames to the point where it’s hard to see what is happening-you can see this for yourself by downloading the production app (dribblegame.com) and playing a round of Swipeball.


```dart
class Swiper extends StatelessWidget {
  const Swiper({super.key});

  @override
  Widget build(BuildContext context) {
    final SwipeballBloc swipeConfig = SwipeballBloc.notifier.value;
    final SwipeballGameBloc swipeball = SwipeballGameBloc.watch(context);
    final Random rand = KarmicRandom();
    // Gets kind of silly once you go past 10.
    final int stackCount = min(swipeball.playersRemaining, 10);
    // We need to store whether or not to animate in the controller so that:
    //   A. Other animations can trigger once this one is done
    //   B. The animation doesn't play if we leave the page and come back
    //      to an in-progress game. It only plays on a new game.
    if (!swipeball.animateCards) {
      return _RealSwiper(
        canWiggle: swipeball.canWiggle,
        isEnabled: swipeball.isPlaying || swipeConfig.enableSwipeNextRound,
      );
```

```
    }
    // Show an animating dummy-stack as we animate in, once the animation is
    // done we can replace the dummy stack with the actual user-interactable
    // swiper element.
    return Stack(
      children: List.generate(
        stackCount,
        (index) {
          final bool isTop = (index == stackCount - 1);
          // Slide in from randomized approach angles.
          final double rotation = .008 + .01 * (rand.nextDouble() - .5);
          final Widget child = isTop
              ? const _RealSwiper(isEnabled: false, canWiggle: false)
              : Padding(
                  padding: EdgeInsets.symmetric(horizontal: context.margin),
                  child: const CardBack(),
                );
          return child
              .animate()
              .slideY(
                curve: Curves.easeOut,
                // An arbitrary begin point that is far enough out to not be
                // visible at first.
                begin: -1.5,
                end: 0,
                duration: kDurationFast,
                // Multiply the delay by the index to make the cards chain
                // in after each other.
                delay: kDurationFast * index,
              )
              // After the cards have animated in, undo their rotation to
              // create a neat effect where the cards look like they're
              // being "tied" and lined up.
              .rotate(
                // Disorienting delay math that works but I don't remember
                // how.
                delay: (kDurationFast * (stackCount + 1)) +
                    (kDurationFaster * index),
                curve: Curves.easeOut,
                // Alternate the direction of the approach angle for
                // a nice effect.
                begin: (index.isEven ? -1 : 1) * rotation,
                end: 0,
                duration: kDurationFast,
              )
              .then()
              .callback(
```

```
            callback: (_) {
              if (index == 0) {
                // Notify the controller that we're done so that other UI
                // elements keying on animation state can animate in.
                swipeball.onAnimationComplete();
              }
            },
          );
        },
      ),
    );
  }
}
```

Feedback Animations

These animations are imperatively triggered by a user action and are intended to give the user feedback about their action but don't necessarily represent a direct state change in the animated widget.
I almost exclusively use the Flutter Animate package for these animations as it provides a really easy way to imperatively trigger an animation without cluttering the widget tree with excess nesting (it provides animations via extension methods).

Examples:

A. Throughout the app's 5 games I provide user feedback on correct/incorrect guess. If a guess is correct, the relevant widget expands outward and snaps back. If a guess is incorrect, the relevant widget shakes. In this case, the widget(s) that expand/shake on guess are the HUD around the game cards. The responding widget is often not the widget the user directly interacted with and the widget that animates on correct may not

even be the same widget that animates on wrong.


```dart
class AnimateOnOutcome extends StatefulWidget {
  const AnimateOnOutcome({
    super.key,
    required this.outcomeStream,
    required this.child,
  });

  final Stream<bool> outcomeStream;
  final Widget child;

  @override
  State<AnimateOnOutcome> createState() => _AnimateOnOutcomeState();
}

class _AnimateOnOutcomeState extends State<AnimateOnOutcome>
    with TickerProviderStateMixin {
  late final AnimationController wrongAnimation =
      AnimationController(vsync: this, duration: kDuration);
```

```dart
late final AnimationController correctAnimation =
    AnimationController(vsync: this, duration: kDuration * 1.5);

void onOutcome(bool correct) {
  if (correct) {
    correctAnimation.forward(from: 0);
  } else {
    wrongAnimation.forward(from: 0);
  }
}

late final StreamSubscription<bool> subscription;

@override
void initState() {
  // Needs to be in init to ensure it actually gets initialized.
  subscription = widget.outcomeStream.listen(onOutcome);
  super.initState();
}

@override
void dispose() {
  wrongAnimation.dispose();
  correctAnimation.dispose();
  unawaited(subscription.cancel());
  super.dispose();
}

@override
Widget build(BuildContext context) {
  return widget.child
      .animate(
        controller: correctAnimation,
        autoPlay: false,
      )
      .scaleXY(
        curve: Curves.easeInOut,
        duration: kDurationFast * 1.5,
        begin: 1,
        end: 1.1,
      )
      .then()
      .scaleXY(
        curve: Curves.easeInOut,
        duration: kDurationFast * 1.5,
        begin: 1,
        end: 1 / 1.1,
```

```
        )
        .animate(
          controller: wrongAnimation,
          autoPlay: false,
        )
        .shakeX(
          amount: context.gutterTiny,
        );
  }
}

.
.
.
// Here's where we use the above widget.
final SwipeballGameBloc swipeball = SwipeballGameBloc.watch(context);
// This is the relevant part, but we also need to animate this in/out
// depending on game state. Note how animations compound on each other
// where any single animated effect doesn't add that much complexity,
// but the sum total of effects quickly degrades readability.
final Widget textChild = AnimateOnOutcome(
  outcomeStream: swipeball.outcomeStream.map((o) => o.wasCorrect),
  child: SwipeballPromptView(prompt: swipeball.prompt),
);
return AnimatedSwitcher(
  duration: kDuration,
  transitionBuilder: (child, animation) {
    return SizeTransition(
      axisAlignment: 1,
      sizeFactor: Tween<double>(begin: 0, end: 1).animate(animation),
      child: child,
    );
  },
  child: swipeball.animateCards
      ? const SizedBox()
      : Container(
          color: Theme.of(context).colorScheme.background,
          padding: EdgeInsets.symmetric(horizontal: context.margin),
          child: Center(child: textChild),
        ),
);
```

Gesture Driven Effects

These are effects (translation, opacity, rotation, etc) that lerp between states driven by a user gesture (eg a drag). This is most often used by tying the translation to a scroll controller's position.

These are INCREDIBLY fun as they're very easy to do once you're familiar with the approach and add a very large amount to user-perceived app quality and fun.

Examples:

A. As the user drags up on my paywall sheet:
   a. The upper right back transparent background button rotates into a "go back up" elevated button

```
final DraggableScrollableController controller =
    Provider.of<DraggableScrollableController>(context);
final double progress = controller.progress;
final double opacity = (1.5 * (progress - .2)).clamp(0, 1);
final bool willClose = progress < .2;
return SafeArea(
  bottom: false,
  child: Container(
    child: LoggingIconButton(
      name: willClose ? 'close' : 'close_sheet',
      onTap: () {
        if (willClose) {
          DribbleBloc.instance.onModalChanged(null);
          return;
        }
        unawaited(resetSheet());
      },
      child: Transform.rotate(
        angle: -pi / 2 * progress,
        child: Transform.translate(
          offset: const Offset(4.25, 0),
          child: const Icon(Icons.arrow_back_ios),
        ),
      ),
    ),
    margin: EdgeInsets.only(left: context.gutter),
    decoration: BoxDecoration(
      color: Theme.of(context).colorScheme.surface.withOpacity(
            opacity,
          ),
      shape: BoxShape.circle,
      boxShadow: [
        BoxShadow(
          color: Theme.of(context).colorScheme.shadow.withOpacity(opacity),
          blurRadius: context.gutter,
        ),
      ],
    ),
  ),
```

```
);
```

b. The background image and text fades to grey

```
final double progress =
    Provider.of<DraggableScrollableController>(context).progress;
return Opacity(
  opacity: 1 - progress,
  child: Stack(
    children: [
      FractionalTranslation(
        translation: Offset(0, -progress / 2),
        child: FractionallySizedBox(
          heightFactor: .5,
          child: Transform.scale(
            scale: 1.2 - (.2 * progress),
            child:Image.asset(
                    'assets/images/dribble_background.png',
                    fit: BoxFit.cover,
                  ),
          ),
        ),
      ),
      Positioned.fill(
        child: FractionallySizedBox(
          alignment: Alignment.topCenter,
          heightFactor: 1 - .2 * progress,
          widthFactor: 1,
          child: DecoratedBox(
            decoration: BoxDecoration(
              gradient: RadialGradient(
                colors: [
                  Colors.transparent,
                  Colors.black.withOpacity(.75),
                ],
                radius: 1.25,
                center: const Alignment(0, -1.1),
              ),
            ),
          ),
        ),
      ),
      Positioned.fill(
        top: 0,
        child: _BackgroundCta(
          ctaLabel: ctaLabel,
          ctaIcon: ctaIcon,
        ),
```

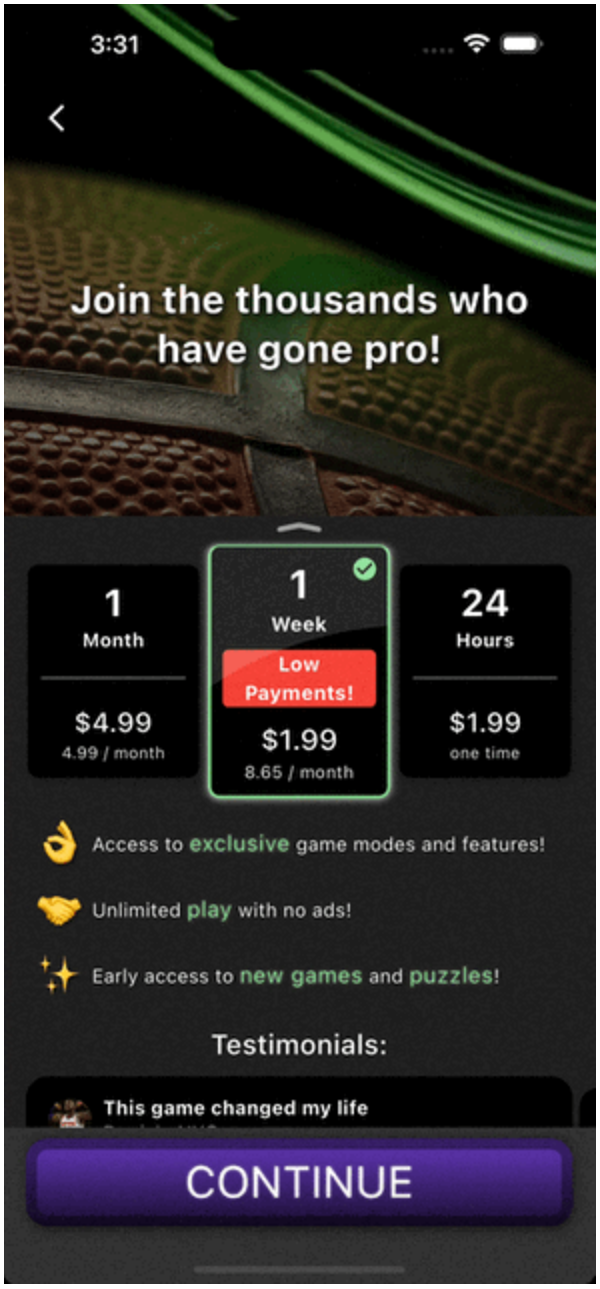
```
      ),
    ],
  ),
);
```

c. The “go up” chevron shaped drag handle/button un-shears into a flat drag handle

```
const Size handleSize = Size(32, 4);
final DraggableScrollableController controller =
    Provider.of<DraggableScrollableController>(context);
final double progress = controller.progress;
final double skew = .2 * (1 - progress);
return Container(
  alignment: Alignment.center,
  color: Theme.of(context).colorScheme.surface,
  child: LoggingInkWell(
    name: 'sheet_drag_handle',
    onTap: () async {
      await controller.animateTo(
        1,
        duration: kDuration,
        curve: Curves.easeInOut,
      );
    },
    child: Row(
      mainAxisSize: MainAxisSize.min,
      children: List.generate(
        2,
        (index) {
          final bool isLeft = index == 0;
          return Transform.flip(
            flipX: !isLeft,
            child: Transform.translate(
              offset: Offset(0, progress * context.gutterTiny / 3),
              child: Transform(
                transform: Matrix4.skewY(-skew),
                child: Container(
                  margin: EdgeInsets.only(
                    top: context.gutterSmall,
                    bottom: context.gutterTiny,
                  ),
                  height: handleSize.height,
                  width: handleSize.width / 2,
                  decoration: BoxDecoration(
                    borderRadius: BorderRadius.horizontal(
                      left: Radius.circular(handleSize.height / 2),
                    ),
                    color: Theme.of(context).colorScheme.outline,
```

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

d. The background text reappears at the bottom below the “CONTINUE” button
e. A grey panel slides in from the top to cover the safe area.



Page Transitions

Here I just use all the basic navigator idioms, won't bother showing example gifs or code here but it's stuff like the iOS horizontal page transition with the gesture-driven horizontal drag to go back and full screen modals that animate in/out from the bottom.

Pain Points/Whining

This section is pretty off-the-cuff and should all be taken as conversation starters and with a grain of salt.

A. There are A LOT of different animation systems with different pros and cons. It feels like with some improvements, they could be unified a little more. The most egregious spot is in applying effects-Flutter Animate has an awesome library of effects and `Tween` is an absolute pain to use directly, but it's not straightforward (or is it? I guess I haven't tried but it's not immediately obvious) to use the flutter animate out-of-the-box effects with the implicitly animated built ins or with the built in transformers and thus gesture driven animations.
B. The animation that needs to be triggered is often not the same as the widget the user interacted with that will trigger the animation you need a shared controller accessible to both widgets. If your state management package tries to be context dependent this is especially inconvenient as animation controllers require context to create. I solve this in several different ways (provider-held animation specific controller, bool streams held by my context-unaware controller…) but there isn't a single obvious "clean" way.
C. I'm constantly piping `kDuration`, `kDurationFast` etc into my widgets, would be nice to have that held by an inherited theme and implicitly defaulted to by the built ins.
D. I'm specifying curve in an entirely ad-hoc way and in most places I'm not even thinking to supply a non-default curve probably to the detriment of quality.
E. Animation chaining is really awkward, sometimes staggering using delays, sometimes holding animation bools in controllers and then referencing the bools in switchers, sometimes using `.then` in Flutter Animate. Another place where there are a lot of different approaches with different tradeoffs and it feels hard to pick between them.
F. Any single animation doesn't add that much complexity/hurt code readability that much, but as you layer animated effects on top of each other (especially mix and matching the different types as outlined above) your code gets really hard to read fast.
G. If you have a switch or a basic true/false controlling your animated switcher it's really easy, but if you have a more complicated case you can end up with a late local variable and/or an embedded method or nested ternaries, all of which feel sloppy. Not sure a good solution-maybe an optional builder argument for switcher that replaces child?
H. No `delay` on animated switcher making it hard to chain switch animations
I. It's hard to create a widget that animates the first time it appears, but then doesn't re-animate when re-entering the widget tree. See State Transitions Example E (shuffling cards in at the start of each round). I want the animation to play when the new round widget tree enters the sub tree, but I can't just simply use flutter animate's autoplay feature because then it'll re-animate if the user navigates away from the page and back so I have to store a bool somewhere in state just to check if the auto-play animation should be enabled or not.