

Animations

Animations are very important to create a great user experience. Stationary objects must overcome inertia as they start moving. Objects in motion have momentum and rarely come to a stop immediately. Animations allow you to convey physically believable motion in your interface.

React Native provides two complementary animation systems: Animated for granular and interactive control of specific values, and LayoutAnimation for animated global layout transactions.

Animated API

The Animated API is designed to concisely express a wide variety of interesting animation and interaction patterns in a very performant way. Animated focuses on declarative relationships between inputs and outputs, with configurable transforms in between, and `start / stop` methods to control time-based animation execution.

Animated exports six animatable component types: `View`, `Text`, `Image`, `ScrollView`, `FlatList` and `SectionList`, but you can also create your own using `Animated.createAnimatedComponent()`.

For example, a container view that fades in when it is mounted may look like this:

TypeScript

JavaScript

Example

```
import React, {useRef, useEffect} from 'react';
import {Animated, Text, View} from 'react-native';
import type {PropsWithChildren} from 'react';
import type {ViewStyle} from 'react-native';

type FadeInViewProps = PropsWithChildren<{style:
ViewStyle}>;

const FadeInView: React.FC<FadeInViewProps> = props => {
  const fadeAnim = useRef(new Animated.Value(0)).current;
  // Initial value for opacity: 0

  useEffect(() => {
    Animated.timing(fadeAnim, {
      toValue: 1,
      duration: 10000,
      useNativeDriver: true,
    }).start();
  }, [fadeAnim]);

  return (
    <Animated.View // Special animatable View
      style={{
        ...props.style,
        opacity: fadeAnim, // Bind opacity to animated
value
  --
```

Preview



My Device

iOS

Android

Web

Let's break down what's happening here. In the `FadeInView` constructor, a new `Animated.Value` called `fadeAnim` is initialized as part of `state`. The `opacity` property on the `View` is mapped to this animated value. Behind the scenes, the numeric value is extracted and used to set `opacity`.

When the component mounts, the `opacity` is set to 0. Then, an easing animation is started on the `fadeAnim` animated value, which will update all of its dependent mappings (in this case, only the `opacity`) on each frame as the value animates to the final value of 1.

This is done in an optimized way that is faster than calling `setState` and re-rendering. Because the entire configuration is declarative, we will be able to implement further optimizations that serialize the configuration and runs the animation on a high-priority thread.

Configuring animations

Animations are heavily configurable. Custom and predefined easing functions, delays, durations, decay factors, spring constants, and more can all be tweaked depending on the type of animation.

`Animated` provides several animation types, the most commonly used one being `Animated.timing()`. It supports animating a value over time using one of various predefined easing functions, or you can use your own. Easing functions are typically used in animation to convey gradual acceleration and deceleration of objects.

By default, `timing` will use an `easing.out` curve that conveys gradual acceleration to full speed and concludes by gradually decelerating to a stop. You can specify a different easing function by passing an `easing` parameter. Custom duration or even a delay before the animation starts is also supported.

For example, if we want to create a 2-second long animation of an object that slightly backs up before moving to its final position:

```
Animated.timing(this.state.xPosition, {  
  toValue: 100,  
  easing: Easing.back(),  
  duration: 2000,  
  useNativeDriver: true,  
}).start();
```

Take a look at the [Configuring animations](#) section of the `Animated` API reference to learn more about all the config parameters supported by the built-in animations.

Composing animations

Animations can be combined and played in sequence or in parallel. Sequential animations can play immediately after the previous animation has finished, or they can start after a specified delay. The `Animated` API provides several methods, such as `sequence()` and `delay()`, each of which take an array of animations to execute and automatically calls `start()` / `stop()` as needed.

For example, the following animation coasts to a stop, then it springs back while twirling in parallel:

```
Animated.sequence([
  // decay, then spring to start and twirl
  Animated.decay(position, {
    // coast to a stop
    velocity: {x: gestureState.vx, y: gestureState.vy}, // velocity from gesture release
    deceleration: 0.997,
    useNativeDriver: true,
  }),
  Animated.parallel([
    // after decay, in parallel:
    Animated.spring(position, {
      toValue: {x: 0, y: 0}, // return to start
      useNativeDriver: true,
    }),
    Animated.timing(twirl, {
      // and twirl
      toValue: 360,
      useNativeDriver: true,
    }),
  ]),
]).start(); // start the sequence group
```

If one animation is stopped or interrupted, then all other animations in the group are also stopped. `Animated.parallel` has a `stopTogether` option that can be set to `false` to disable this.

You can find a full list of composition methods in the [Composing animations](#) section of the `Animated` API reference.

Combining animated values

You can combine two animated values via addition, multiplication, division, or modulo to make a new animated value.

There are some cases where an animated value needs to invert another animated value for calculation. An example is inverting a scale ($2x \rightarrow 0.5x$):

```
const a = new Animated.Value(1);
const b = Animated.divide(1, a);

Animated.spring(a, {
  toValue: 2,
  useNativeDriver: true,
}).start();
```

Interpolation

Each property can be run through an interpolation first. An interpolation maps input ranges to output ranges, typically using a linear interpolation but also supports easing functions. By default, it will extrapolate the curve beyond the ranges given, but you can also have it clamp the output value.

A basic mapping to convert a 0-1 range to a 0-100 range would be:

```
value.interpolate({
  inputRange: [0, 1],
  outputRange: [0, 100],
});
```

For example, you may want to think about your `Animated.Value` as going from 0 to 1, but animate the position from 150px to 0px and the opacity from 0 to 1. This can be done by modifying `style` from the example above like so:

```
style={{
  opacity: this.state.fadeAnim, // Binds directly
  transform: [{
    translateY: this.state.fadeAnim.interpolate({
      inputRange: [0, 1],
      outputRange: [150, 0] // 0 : 150, 0.5 : 75, 1 : 0
    }),
  ]},
}
```

`interpolate()` supports multiple range segments as well, which is handy for defining dead zones and other handy tricks. For example, to get a negation relationship at -300 that goes to 0 at -100, then back up to 1 at 0, and then back down to zero at 100 followed by a dead-zone that remains at 0 for everything beyond that, you could do:

```
value.interpolate({
  inputRange: [-300, -100, 0, 100, 101],
  outputRange: [300, 0, 1, 0, 0],
});
```

Which would map like so:

Input	Output
-400	450
-300	300
-200	150
-100	0
-50	0.5
0	1
50	0.5
100	0
101	0
200	0

`interpolate()` also supports mapping to strings, allowing you to animate colors as well as values with units. For example, if you wanted to animate a rotation you could do:

```
value.interpolate({
  inputRange: [0, 360],
  outputRange: ['0deg', '360deg'],
});
```

`interpolate()` also supports arbitrary easing functions, many of which are already implemented in the `Easing` module. `interpolate()` also has configurable behavior for extrapolating the `outputRange`. You can set the extrapolation by setting the `extrapolate`, `extrapolateLeft`, or `extrapolateRight` options. The default value is `extend` but you can use `clamp` to prevent the output value from exceeding `outputRange`.

Tracking dynamic values

Animated values can also track other values by setting the `toValue` of an animation to another animated value instead of a plain number. For example, a "Chat Heads" animation like the one used by Messenger on Android could be implemented with a `spring()` pinned on another animated value, or with `timing()` and a `duration` of 0 for rigid tracking. They can also be composed with interpolations:

```
Animated.spring(follower, {toValue: leader}).start();
Animated.timing(opacity, {
  toValue: pan.x.interpolate({
    inputRange: [0, 300],
    outputRange: [1, 0],
    useNativeDriver: true,
  }),
}).start();
```

The `leader` and `follower` animated values would be implemented using `Animated.ValueXY()`. `ValueXY` is a handy way to deal with 2D interactions, such as panning or dragging. It is a basic wrapper that contains two `Animated.Value` instances and some helper functions that call through to them, making `valueXY` a drop-in replacement for `Value` in many cases. It allows us to track both `x` and `y` values in the example above.

Tracking gestures

Gestures, like panning or scrolling, and other events can map directly to animated values using `Animated.event()`. This is done with a structured map syntax so that values can be extracted from complex event objects. The first level is an array to allow mapping across multiple args, and that array contains nested objects.

For example, when working with horizontal scrolling gestures, you would do the following in order to map `event.nativeEvent.contentOffset.x` to `scrollX` (an `Animated.Value`):

```
onScroll={Animated.event(
  // scrollX = e.nativeEvent.contentOffset.x
  [{nativeEvent: {
    contentOffset: {
```

```

        x: scrollX
      }
    }
  }
}
})

```

The following example implements a horizontal scrolling carousel where the scroll position indicators are animated using the `Animated.event` used in the `ScrollView`

ScrollView with Animated Event Example

Animated

^ Expo

```

import React, {useRef} from 'react';
import {
  SafeAreaView,
  ScrollView,
  Text,
  StyleSheet,
  View,
  ImageBackground,
  Animated,
  useWindowDimensions,
} from 'react-native';

const images = new Array(6).fill(
  'https://images.unsplash.com/photo-1556740749-887f6717d7e4',
);

const App = () => {
  const scrollX = useRef(new Animated.Value(0)).current;

  const {width: windowWidth} = useWindowDimensions();

  return (
    <SafeAreaView style={styles.container}>
      <View style={styles.scrollContainer}>
        <ScrollView
          horizontal={true}

```

Preview

My Device

iOS

Android

When using `PanResponder`, you could use the following code to extract the `x` and `y` positions from `gestureState.dx` and `gestureState.dy`. We use a `null` in the first position of the array, as we are only interested in the second argument passed to the `PanResponder` handler, which is the `gestureState`.


```
onPanResponderMove={Animated.event(  
  [null, // ignore the native event  
  // extract dx and dy from gestureState  
  // like 'pan.x = gestureState.dx, pan.y = gestureState.dy'  
  {dx: pan.x, dy: pan.y}  
)]})
```

PanResponder with Animated Event Example



Responding to the current animation value

You may notice that there is no clear way to read the current value while animating. This is because the value may only be known in the native runtime due to optimizations. If you need to run JavaScript in response to the current value, there are two approaches:

- `spring.stopAnimation(callback)` will stop the animation and invoke `callback` with the final value. This is useful when making gesture transitions.
- `spring.addListener(callback)` will invoke `callback` asynchronously while the animation is running, providing a recent value. This is useful for triggering state changes, for example snapping a bobble to a new option as the user drags it closer, because these larger state changes are less sensitive to a few frames of lag compared to continuous gestures like panning which need to run at 60 fps.

`Animated` is designed to be fully serializable so that animations can be run in a high performance way, independent of the normal JavaScript event loop. This does influence the API, so keep that in mind when it seems a little trickier to do something compared to a fully synchronous system. Check out `Animated.Value.addListener` as a way to work around some of these limitations, but use it sparingly since it might have performance implications in the future.

Using the native driver

The `Animated` API is designed to be serializable. By using the native driver, we send everything about the animation to native before starting the animation, allowing native code to perform the animation on the UI thread without having to go through the bridge on every frame. Once the animation has started, the JS thread can be blocked without affecting the animation.

Using the native driver for normal animations can be accomplished by setting `useNativeDriver: true` in animation config when starting it. Animations without a `useNativeDriver` property will default to `false` for legacy reasons, but emit a warning (and typechecking error in TypeScript).

```
Animated.timing(this.state.animatedValue, {
  toValue: 1,
  duration: 500,
  useNativeDriver: true, // <-- Set this to true
}).start();
```

Animated values are only compatible with one driver so if you use native driver when starting an animation on a value, make sure every animation on that value also uses the native driver.

The native driver also works with `Animated.event`. This is especially useful for animations that follow the scroll position as without the native driver, the animation will always run a frame behind the gesture due to the async nature of React Native.

```
<Animated.ScrollView // <-- Use the Animated ScrollView wrapper
  scrollEventThrottle={1} // <-- Use 1 here to make sure no events are ever missed
  onScroll={Animated.event(
    [
      {
        nativeEvent: {
          contentOffset: {y: this.state.animatedValue},
        },
      },
    ],
    {useNativeDriver: true}, // <-- Set this to true
  )}>
  {content}
</Animated.ScrollView>
```

You can see the native driver in action by running the [RNTester](#) app, then loading the Native Animated Example. You can also take a look at the [source code](#) to learn how these examples were produced.

Caveats

Not everything you can do with `Animated` is currently supported by the native driver. The main limitation is that you can only animate non-layout properties: things like `transform` and `opacity` will work, but `Flexbox` and position properties will not. When using `Animated.event`, it will only work with direct events and not bubbling events. This means it does not work with `PanResponder` but does work with things like `ScrollView#onScroll`.

When an animation is running, it can prevent `VirtualizedList` components from rendering more rows. If you need to run a long or looping animation while the user is

scrolling through a list, you can use `isInteraction: false` in your animation's config to prevent this issue.

Bear in mind

While using transform styles such as `rotateY`, `rotateX`, and others ensure the transform style `perspective` is in place. At this time some animations may not render on Android without it. Example below.

```
<Animated.View
  style={{
    transform: [
      {scale: this.state.scale},
      {rotateY: this.state.rotateY},
      {perspective: 1000}, // without this line this Animation will not render on Android
    ],
  }}
/>
```

while working fine on iOS

Additional examples

The RNTester app has various examples of `Animated` in use:

- [AnimatedGratuitousApp](#)
- [NativeAnimationsExample](#)

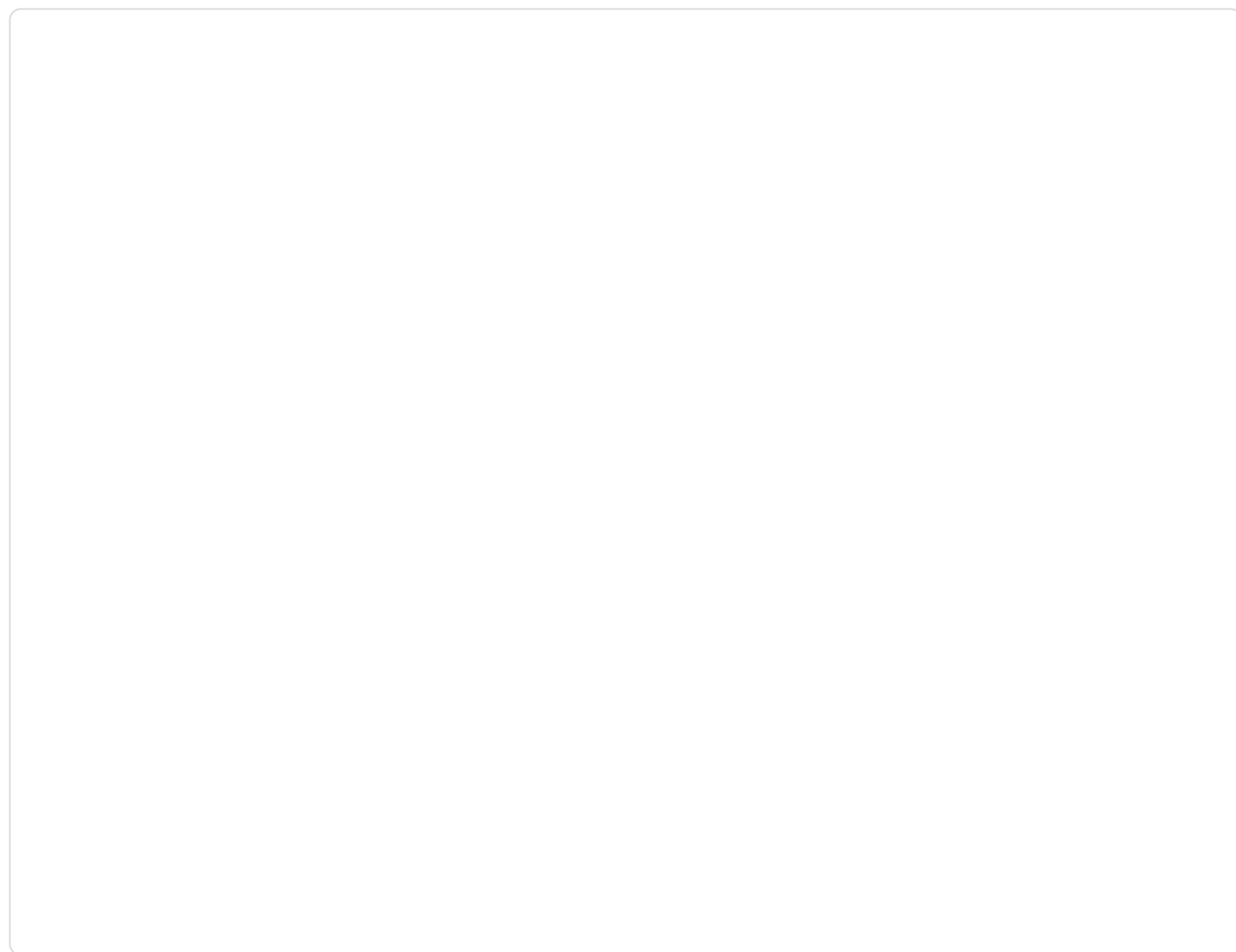
LayoutAnimation API

`LayoutAnimation` allows you to globally configure create and update animations that will be used for all views in the next render/layout cycle. This is useful for doing Flexbox layout updates without bothering to measure or calculate specific properties in order to animate them directly, and is especially useful when layout changes may affect ancestors, for example a "see more" expansion that also increases the size of the parent and pushes down the row below which would otherwise require explicit coordination between the components in order to animate them all in sync.

Note that although `LayoutAnimation` is very powerful and can be quite useful, it provides much less control than `Animated` and other animation libraries, so you may need to use another approach if you can't get `LayoutAnimation` to do what you want.

Note that in order to get this to work on **Android** you need to set the following flags via `UIManager` :

```
UIManager.setLayoutAnimationEnabledExperimental(true);
```



This example uses a preset value, you can customize the animations as you need, see [LayoutAnimation.js](#) for more information.

Additional notes

requestAnimationFrame

`requestAnimationFrame` is a polyfill from the browser that you might be familiar with. It accepts a function as its only argument and calls that function before the next repaint. It is an essential building block for animations that underlies all of the JavaScript-based animation APIs. In general, you shouldn't need to call this yourself - the animation APIs will manage frame updates for you.


setNativeProps

As mentioned in the [Direct Manipulation](#) section, `setNativeProps` allows us to modify properties of native-backed components (components that are actually backed by native views, unlike composite components) directly, without having to `setState` and re-render the component hierarchy.

We could use this in the Rebound example to update the scale - this might be helpful if the component that we are updating is deeply nested and hasn't been optimized with `shouldComponentUpdate`.

If you find your animations with dropping frames (performing below 60 frames per second), look into using `setNativeProps` or `shouldComponentUpdate` to optimize them. Or you could run the animations on the UI thread rather than the JavaScript thread [with the `useNativeDriver` option](#). You may also want to defer any computationally intensive work until after animations are complete, using the [InteractionManager](#). You can monitor the frame rate by using the In-App Dev Menu "FPS Monitor" tool.

Is this page useful?  

 Edit this page

Last updated on **Jun 21, 2023**