Animated

The `Animated` library is designed to make animations fluid, powerful, and painless to build and maintain. `Animated` focuses on declarative relationships between inputs and outputs, configurable transforms in between, and `start` / `stop` methods to control time-based animation execution.

The core workflow for creating an animation is to create an `Animated.Value`, hook it up to one or more style attributes of an animated component, and then drive updates via animations using `Animated.timing()`.

> Don't modify the animated value directly. You can use the `useRef` Hook to return a mutable ref object. This ref object's `current` property is initialized as the given argument and persists throughout the component lifecycle.

Example

The following example contains a `View` which will fade in and fade out based on the animated value `fadeAnim`

Animated Expo

```
import React, {useRef} from 'react';
import {
  Animated,
  Text,
  View,
  StyleSheet,
  Button,
  SafeAreaView,
} from 'react-native';

const App = () => {
  // fadeAnim will be used as the value for opacity. Initial Value: 0
  const fadeAnim = useRef(new Animated.Value(0)).current;

  const fadeIn = () => {
    // Will change fadeAnim value to 1 in 5 seconds
    Animated.timing(fadeAnim, {
      toValue: 1,
      duration: 5000,
      useNativeDriver: true,
    }).start();
  };

  const fadeOut = () => {
    // Will change fadeAnim value to 0 in 3 seconds
    Animated.timing(fadeAnim, {
```

Preview My Device iOS Android Web

Refer to the Animations guide to see additional examples of `Animated` in action.

Overview

There are two value types you can use with `Animated`:

- `Animated.Value()` for single values
- `Animated.ValueXY()` for vectors

`Animated.Value` can bind to style properties or other props, and can be interpolated as well. A single `Animated.Value` can drive any number of properties.

Configuring animations

`Animated` provides three types of animation types. Each animation type provides a particular animation curve that controls how your values animate from their initial value to the final value:

- `Animated.decay()` starts with an initial velocity and gradually slows to a stop.
- `Animated.spring()` provides a basic spring physics model.
- `Animated.timing()` animates a value over time using easing functions.

In most cases, you will be using `timing()`. By default, it uses a symmetric easeInOut curve that conveys the gradual acceleration of an object to full speed and concludes by gradually decelerating to a stop.

Working with animations

Animations are started by calling `start()` on your animation. `start()` takes a completion callback that will be called when the animation is done. If the animation finished running normally, the completion callback will be invoked with `{finished: true}`. If the animation is done because `stop()` was called on it before it could finish (e.g. because it was interrupted by a gesture or another animation), then it will receive `{finished: false}`.

```
Animated.timing({}).start(({finished}) => {
  /* completion callback */
});
```

Using the native driver

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.

You can use the native driver by specifying `useNativeDriver: true` in your animation configuration. See the Animations guide to learn more.

Animatable components

Only animatable components can be animated. These unique components do the magic of binding the animated values to the properties, and do targeted native updates to avoid the cost of the React render and reconciliation process on every frame. They also handle cleanup on unmount so they are safe by default.

- `createAnimatedComponent()` can be used to make a component animatable.

`Animated` exports the following animatable components using the above wrapper:

- `Animated.Image`
- `Animated.ScrollView`
- `Animated.Text`
- `Animated.View`
- `Animated.FlatList`
- `Animated.SectionList`

Composing animations

Animations can also be combined in complex ways using composition functions:

- `Animated.delay()` starts an animation after a given delay.
- `Animated.parallel()` starts a number of animations at the same time.
- `Animated.sequence()` starts the animations in order, waiting for each to complete before starting the next.
- `Animated.stagger()` starts animations in order and in parallel, but with successive delays.

Animations can also be chained together by setting the `toValue` of one animation to be another `Animated.Value`. See Tracking dynamic values in the Animations guide.

By default, if one animation is stopped or interrupted, then all other animations in the group are also stopped.

Combining animated values

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

- `Animated.add()`
- `Animated.subtract()`
- `Animated.divide()`
- `Animated.modulo()`
- `Animated.multiply()`

Interpolation

The `interpolate()` function allows input ranges to map to different output ranges. By default, it will extrapolate the curve beyond the ranges given, but you can also have it clamp the output value. It uses linear interpolation by default but also supports easing functions.

- `interpolate()`

Read more about interpolation in the Animation guide.

Handling gestures and other events

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.

- `Animated.event()`

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
          }
        }
      }]
    )}
```

Reference

Methods

When the given value is a ValueXY instead of a Value, each config option may be a vector of the form `{x: ..., y: ...}` instead of a scalar.

`decay()`

```
static decay(value, config): CompositeAnimation;
```

Animates a value from an initial velocity to zero based on a decay coefficient.

Config is an object that may have the following options:

- `velocity`: Initial velocity. Required.
- `deceleration`: Rate of decay. Default 0.997.
- `isInteraction`: Whether or not this animation creates an "interaction handle" on the `InteractionManager`. Default true.
- `useNativeDriver`: Uses the native driver when true. Required.

`timing()`

```
static timing(value, config): CompositeAnimation;
```

Animates a value along a timed easing curve. The `Easing` module has tons of predefined curves, or you can use your own function.

Config is an object that may have the following options:

- `duration`: Length of animation (milliseconds). Default 500.
- `easing`: Easing function to define curve. Default is `Easing.inOut(Easing.ease)`.
- `delay`: Start the animation after delay (milliseconds). Default 0.
- `isInteraction`: Whether or not this animation creates an "interaction handle" on the `InteractionManager`. Default true.
- `useNativeDriver`: Uses the native driver when true. Required.

`spring()`

```
static spring(value, config): CompositeAnimation;
```

Animates a value according to an analytical spring model based on damped harmonic oscillation. Tracks velocity state to create fluid motions as the `toValue` updates, and can be chained together.

Config is an object that may have the following options.

Note that you can only define one of bounciness/speed, tension/friction, or stiffness/damping/mass, but not more than one:

The friction/tension or bounciness/speed options match the spring model in `Facebook Pop`, Rebound, and Origami.

- `friction`: Controls "bounciness"/overshoot. Default 7.
- `tension`: Controls speed. Default 40.
- `speed`: Controls speed of the animation. Default 12.
- `bounciness`: Controls bounciness. Default 8.

Specifying stiffness/damping/mass as parameters makes `Animated.spring` use an analytical spring model based on the motion equations of a damped harmonic oscillator. This behavior is slightly more precise and faithful to the physics behind spring dynamics, and closely mimics the implementation in iOS's CASpringAnimation.

- `stiffness`: The spring stiffness coefficient. Default 100.
- `damping`: Defines how the spring's motion should be damped due to the forces of friction. Default 10.
- `mass`: The mass of the object attached to the end of the spring. Default 1.

Other configuration options are as follows:

- `velocity`: The initial velocity of the object attached to the spring. Default 0 (object is at rest).
- `overshootClamping`: Boolean indicating whether the spring should be clamped and not bounce. Default false.
- `restDisplacementThreshold`: The threshold of displacement from rest below which the spring should be considered at rest. Default 0.001.
- `restSpeedThreshold`: The speed at which the spring should be considered at rest in pixels per second. Default 0.001.
- `delay`: Start the animation after delay (milliseconds). Default 0.
- `isInteraction`: Whether or not this animation creates an "interaction handle" on the `InteractionManager`. Default true.
- `useNativeDriver`: Uses the native driver when true. Required.

add()

```
static add(a: Animated, b: Animated): AnimatedAddition;
```

Creates a new Animated value composed from two Animated values added together.

subtract()

```
static subtract(a: Animated, b: Animated): AnimatedSubtraction;
```

Creates a new Animated value composed by subtracting the second Animated value from the first Animated value.

divide()

```
static divide(a: Animated, b: Animated): AnimatedDivision;
```

Creates a new Animated value composed by dividing the first Animated value by the second Animated value.

multiply()

```
static multiply(a: Animated, b: Animated): AnimatedMultiplication;
```

Creates a new Animated value composed from two Animated values multiplied together.

modulo()

```
static modulo(a: Animated, modulus: number): AnimatedModulo;
```

Creates a new Animated value that is the (non-negative) modulo of the provided Animated value

diffClamp()

```
static diffClamp(a: Animated, min: number, max: number): AnimatedDiffClamp;
```

Create a new Animated value that is limited between 2 values. It uses the difference between the last value so even if the value is far from the bounds it will start changing when the value starts getting closer again. (`value = clamp(value + diff, min, max)`).

This is useful with scroll events, for example, to show the navbar when scrolling up and to hide it when scrolling down.

delay()

```
static delay(time: number): CompositeAnimation;
```

Starts an animation after the given delay.

sequence()

```
static sequence(animations: CompositeAnimation[]): CompositeAnimation;
```

Starts an array of animations in order, waiting for each to complete before starting the next. If the current running animation is stopped, no following animations will be started.

parallel()

```
static parallel(
  animations: CompositeAnimation[],
  config?: ParallelConfig
): CompositeAnimation;
```

Starts an array of animations all at the same time. By default, if one of the animations is stopped, they will all be stopped. You can override this with the `stopTogether` flag.

`stagger()`

```
static stagger(
  time: number,
  animations: CompositeAnimation[]
): CompositeAnimation;
```

Array of animations may run in parallel (overlap), but are started in sequence with successive delays. Nice for doing trailing effects.

`loop()`

```
static loop(
  animation: CompositeAnimation[],
  config?: LoopAnimationConfig
): CompositeAnimation;
```

Loops a given animation continuously, so that each time it reaches the end, it resets and begins again from the start. Will loop without blocking the JS thread if the child animation is set to `useNativeDriver: true`. In addition, loops can prevent `VirtualizedList`-based components from rendering more rows while the animation is running. You can pass `isInteraction: false` in the child animation config to fix this.

Config is an object that may have the following options:

- `iterations`: Number of times the animation should loop. Default `-1` (infinite).

`event()`

```
static event(
  argMapping: Mapping[],
  config?: EventConfig
): (...args: any[]) => void;
```

Takes an array of mappings and extracts values from each arg accordingly, then calls `setValue` on the mapped outputs. e.g.

```
onScroll={Animated.event(
  [{nativeEvent: {contentOffset: {x: this._scrollX}}}],
  {listener: (event: ScrollEvent) => console.log(event)}, // Optional async listener
)}
 ...
onPanResponderMove: Animated.event(
  [
    null, // raw event arg ignored
    {dx: this._panX},
  ], // gestureState arg
  {
    listener: (
      event: GestureResponderEvent,
      gestureState: PanResponderGestureState
    ) => console.log(event, gestureState),
  } // Optional async listener
);
```

Config is an object that may have the following options:

- `listener`: Optional async listener.
- `useNativeDriver`: Uses the native driver when true. Required.

`forkEvent()`

```
static forkEvent(event: AnimatedEvent, listener: Function): AnimatedEvent;
```

Advanced imperative API for snooping on animated events that are passed in through props. It permits to add a new javascript listener to an existing `AnimatedEvent`. If

`animatedEvent` is a javascript listener, it will merge the 2 listeners into a single one, and if `animatedEvent` is null/undefined, it will assign the javascript listener directly. Use values directly where possible.

`unforkEvent()`

```
static unforkEvent(event: AnimatedEvent, listener: Function);
```

`start()`

```
static start(callback?: (result: {finished: boolean}) => void);
```

Animations are started by calling start() on your animation. start() takes a completion callback that will be called when the animation is done or when the animation is done because stop() was called on it before it could finish.

Parameters:

NAME	TYPE	REQUIRED	DESCRIPTION
callback	`(result: {finished: boolean}) => void`	No	Function that will be called after the animation finished running normally or when the animation is done because stop() was called on it before it could finish

Start example with callback:

```
Animated.timing({}).start(({finished}) => {
  /* completion callback */
});
```

`stop()`

```
static stop();
```

Stops any running animation.

`reset()`

```
static reset();
```

Stops any running animation and resets the value to its original.

Properties

`Value`

Standard value class for driving animations. Typically initialized with `new Animated.Value(0);`

You can read more about `Animated.Value` API on the separate page.

`ValueXY`

2D value class for driving 2D animations, such as pan gestures.

You can read more about `Animated.ValueXY` API on the separate page.

`Interpolation`

Exported to use the Interpolation type in flow.

`Node`

Exported for ease of type checking. All animated values derive from this class.

`createAnimatedComponent`

Make any React component Animatable. Used to create `Animated.View`, etc.

`attachNativeEvent`

Imperative API to attach an animated value to an event on a view. Prefer using `Animated.event` with `useNativeDriver: true` if possible.

Is this page useful? 

Edit this page

*Last updated on **Aug 17, 2023***