Version: 2.6.0 – 2.12.0

Upgrading to the new API introduced in Gesture Handler 2

Make sure to migrate off the `RNGestureHandlerEnabledRootView` (Android only)

Gesture Handler 1 required you to override `createRootView` to return an instance of `RNGestureHandlerEnabledRootView`. This class has been the cause of many hard to debug and fix crashed and was deprecated in version 2.0, and subsequently removed in version 2.4. If you are still using it, check out migrating off RNGHEnabledRootView guide.

Upgrading to the new API

The most important change brought by the Gesture Handler 2 is the new Gesture API, along with the `GestureDetector` component. It makes declaring gesture easier, as it handles much of the work under the hood and reduces the amount of necessary boilerplate code. Instead of a separate component for every type of gesture, the `GestureDetector` component is used to attach gestures to the underlying view based on the configuration object passed to it. The configuration objects are created using the `Gesture` object, here is a simple example:

```
const tapGesture = Gesture.Tap().onStart(() => {
  console.log('Tap!');
});
...
return (
  <GestureDetector gesture={tapGesture}>
    <View />
  </GestureDetector>
);
```

As you can see, there are no `onGestureEvent` and `onHandlerStateChange` callbacks, instead the state machine is handled under the hood and relevant callbacks are called for specific transitions or events:

- `onBegin` - called when the gesture transitions to the `BEGAN` state, which in most cases is when the gesture starts processing the touch stream - when the finger first touches the view
- `onStart` - called when the activation criteria for the gesture are met and it transitions from `BEGAN` to `ACTIVE` state
- `onUpdate` - replaces `onGestureEvent`, called every time the gesture sends a new event while it's in the `ACTIVE` state
- `onChange` - if defined, called just after `onUpdate`, the events passed to it are the same as the ones passed to `onUpdate` but they also contain `change` values which hold the change in value they represent since the last event (i.e. in case of the `Pan` gesture, the event will also contain `changeX` and `changeY` properties)
- `onEnd` - called when the gesture transitions from the `ACTIVE` state to either of `END`, `FAILED` or `CANCELLED` - you can tell whether the gesture finished due to user interaction or because of other reason (like getting cancelled by the system, or failure criteria) using the second value passed to the `onEnd` callback alongside the event
- `onFinalize` called when the gesture transitions into either of `END`, `FAILED` or `CANCELLED` state, if the gesture was `ACTIVE`, `onEnd` will be called first (similarily to `onEnd` you can determine the reason for finishing using the second argument)

The difference between `onEnd` and `onFinalize` is that the `onEnd` will be called only if the gesture was `ACTIVE`, while `onFinalize` will be called if the gesture has `BEGAN`. This means that you can use `onEnd` to clean up after `onStart`, and `onFinalize` to clean up after `onBegin` (or both `onBegin` and `onStart`).

Configuring the gestures

The new gesture objects are configured in the builder-like pattern. Instead of properties, each gesture provides methods that allow for its customization. In most cases the names of the methods are the same as the relevant props, ot at least very similar. For example:

```
return (
  <TapGestureHandler
    numberOfTaps={2}
    maxDurationMs={500}
    maxDelayMs={500}
    maxDist={10}
    onHandlerStateChange={({ nativeEvent }) => {
      if (nativeEvent.state === State.ACTIVE) {
        console.log('Tap!');
      }
```

```
    }}>
    <View />
  </TapGestureHandler>
);
```

Would have the same effect as:

```
const tapGesture = Gesture.Tap()
  .numberOfTaps(2)
  .maxDuration(500)
  .maxDelay(500)
  .maxDistance(10)
  .onStart(() => {
    console.log('Tap!');
  });

return (
  <GestureDetector gesture={tapGesture}>
    <View />
  </GestureDetector>
);
```

You can check the modifiers available to specific gestures in the API Reference under Gestures.

Using multiple gestures on the same view

Using the gesture handler components, if you wanted to have multiple gestures on one view, you would have to stack them on top of each other and, in case you wanted to use animations, add an `Animated.View` after each handler, resulting in a deep component tree, for example:

```
return (
  <TapGestureHandler ... >
    <Animated.View>
      <PanGestureHandler ... >
        <Animated.View>
          <PinchGestureHandler ... >
            <YourView />
          </PinchGestureHandler>
        </Animated.View>
      </PanGestureHandler>
    </Animated.View>
  </TapGestureHandler>
);
```

With the `GestureDetector` you can use the Gesture Composition API to stack the gestures onto one view:

```
const tapGesture = Gesture.Tap();
const panGesture = Gesture.Pan();
const pinchGesture = Gesture.Pinch();

return (
  <GestureDetector gesture={Gesture.Race(tapGesture, panGesture, pinchGesture)}>
    <YourView />
  </GestureDetector>
);
```

Similarily, you can use `Gesture.Simultaneous` to replace stacked gesture handlers that should be able to recognize gestures simultaneously, and `Gesture.Exclusive` to replace stacked gesture handlers that require failure of others.

Replacing `waitFor` and `simultaneousHandlers`

If you want to make relations between the gestures attached to the same view, you should use the Gesture Composition API described above. However, if you want to make a relation between gestures attached to different views, or between gesture and an old gesture handler, you should use `simultaneousWithExternalGesture` instead of `simultaneousHandlers`, and `requireExternalGestureToFail` instead of `waitFor`. In case you need a ref object to pass to an old gesture handler, you can set it to the gesture using `.withRef(refObject)` modifier.