Gesture Responder System

The gesture responder system manages the lifecycle of gestures in your app. A touch can go through several phases as the app determines what the user's intention is. For example, the app needs to determine if the touch is scrolling, sliding on a widget, or tapping. This can even change during the duration of a touch. There can also be multiple simultaneous touches.

The touch responder system is needed to allow components to negotiate these touch interactions without any additional knowledge about their parent or child components.

Best Practices

To make your app feel great, every action should have the following attributes:

- Feedback/highlighting- show the user what is handling their touch, and what will happen when they release the gesture
- Cancel-ability- when making an action, the user should be able to abort it mid-touch by dragging their finger away

These features make users more comfortable while using an app, because it allows people to experiment and interact without fear of making mistakes.

TouchableHighlight and Touchable*

The responder system can be complicated to use. So we have provided an abstract `Touchable` implementation for things that should be "tappable". This uses the responder system and allows you to configure tap interactions declaratively. Use `TouchableHighlight` anywhere where you would use a button or link on web.

Responder Lifecycle

A view can become the touch responder by implementing the correct negotiation methods. There are two methods to ask the view if it wants to become responder:

- `View.props.onStartShouldSetResponder: evt => true,` - Does this view want to become responder on the start of a touch?
- `View.props.onMoveShouldSetResponder: evt => true,` - Called for every touch move on the View when it is not the responder: does this view want to "claim" touch responsiveness?

If the View returns true and attempts to become the responder, one of the following will happen:

- `View.props.onResponderGrant: evt => {}` - The View is now responding for touch events. This is the time to highlight and show the user what is happening
- `View.props.onResponderReject: evt => {}` - Something else is the responder right now and will not release it

If the view is responding, the following handlers can be called:

- `View.props.onResponderMove: evt => {}` - The user is moving their finger
- `View.props.onResponderRelease: evt => {}` - Fired at the end of the touch, ie "touchUp"
- `View.props.onResponderTerminationRequest: evt => true` - Something else wants to become responder. Should this view release the responder? Returning true allows release
- `View.props.onResponderTerminate: evt => {}` - The responder has been taken from the View. Might be taken by other views after a call to `onResponderTerminationRequest`, or might be taken by the OS without asking (happens with control center/ notification center on iOS)

`evt` is a synthetic touch event with the following form:

- `nativeEvent`
  - `changedTouches` - Array of all touch events that have changed since the last event
  - `identifier` - The ID of the touch

- `locationX` - The X position of the touch, relative to the element
- `locationY` - The Y position of the touch, relative to the element
- `pageX` - The X position of the touch, relative to the root element
- `pageY` - The Y position of the touch, relative to the root element
- `target` - The node id of the element receiving the touch event
- `timestamp` - A time identifier for the touch, useful for velocity calculation
- `touches` - Array of all current touches on the screen

Capture ShouldSet Handlers

`onStartShouldSetResponder` and `onMoveShouldSetResponder` are called with a bubbling pattern, where the deepest node is called first. That means that the deepest component will become responder when multiple Views return true for `*ShouldSetResponder` handlers. This is desirable in most cases, because it makes sure all controls and buttons are usable.

However, sometimes a parent will want to make sure that it becomes responder. This can be handled by using the capture phase. Before the responder system bubbles up from the deepest component, it will do a capture phase, firing `on*ShouldSetResponderCapture`. So if a parent View wants to prevent the child from becoming responder on a touch start, it should have a `onStartShouldSetResponderCapture` handler which returns true.

- `View.props.onStartShouldSetResponderCapture: evt => true,`
- `View.props.onMoveShouldSetResponderCapture: evt => true,`

PanResponder

For higher-level gesture interpretation, check out PanResponder.

Is this page useful? 

Edit this page

*Last updated on **Jun 21, 2023***