Interactions

Interactions and gestures for the web.

React Native for Web extends the interaction modes available in React Native to account for mouse, touch, and keyboard use. The Responder System is available for more advanced gesture recognition.

PointerEvent Props API

Pointer interactions are available on supporting elements. These events are React DOM synthetic mouse events. The `click` events may be dispatched by the browser following user interactions with a pointer (mouse or touch) as well as a keyboard.

In cases where a native `click` is not dispatched following a valid keyboard interaction (due to the native semantics of the host element), it will be emulated by React Native for Web. This helps to improve the accessibility of elements for all forms of interaction hardware.

> **onClick**: ?(event: PointerEvent) => void
> Called when the element is clicked.

> **onClickCapture**: ?(event: PointerEvent) => void
> Called when the element is clicked. (Capture phase.)

> **onContextMenu**: ?(event: PointerEvent) => void
> Called when a native context menu is displayed. This may be in response to mouse, touch, or mouse+keyboard interaction.

> **onPointer***: ?(event: PointerEvent) => void
> Support for the PointerEvent API.

FocusEvent Props API

Focus interactions are available on supporting elements. The focus events are React DOM synthetic focus events. These events are also fired in response to focus being programmatically moved.

> **onBlur**: ?(event: FocusEvent) => void
> Called when the element loses focus.

> **onFocus**: ?(event: FocusEvent) => void
> Called when the element receives focus.

KeyboardEvent Props API

Keyboard interactions are available on supporting elements. The keyboard events are React DOM synthetic keyboard events.

> **onKeyDown**: ?(event: KeyboardEvent) => void
> Called when a key is pressed down.

> **onKeyDownCapture**: ?(event: KeyboardEvent) => void
> Called when a key is pressed down. (Capture phase.)

> **onKeyUp**: ?(event: KeyboardEvent) => void
> Called when a key is released.

> **onKeyUpCapture**: ?(event: KeyboardEvent) => void
> Called when a key is released. (Capture phase.)

ResponderEvent Props API

“Responder” interactions are available on supporting elements. The Responder System allows views and gesture recognizers to opt-in to negotiating over a single, global “interaction lock”. For a view to become the “responder” means that pointer interactions are exclusive to that view and none other. A view can negotiate to become the “responder” without requiring knowledge of other views. A more specialized API for working with multi-pointer gestures is available by using the PanResponder module.

A view can become the "responder" after the following native events: `scroll`, `touchstart`, `touchmove`, `mousedown`, `mousemove`. If nothing is already the "responder", the event propagates to (capture) and from (bubble) the event target until a view returns `true` for `on*SetResponder(Capture)`. If a view is *currently* the responder, the negotiation event propagates to (capture) and from (bubble) the lowest common ancestor of the event target and the current responder. Then negotiation happens between the current responder and the view that wants to become the responder.

NOTE: For historical reasons (originating from React Native), mouse interactions are represented as a single `touch`.

Negotiation props

A view can become the responder by using the negotiation callbacks. During the capture phase the deepest node is called last. During the bubble phase the deepest node is called first. The capture phase should be used when a view wants to prevent a descendant from becoming the responder. The first view to return `true` from any of the `on*ShouldSetResponderCapture` / `on*ShouldSetResponder` callbacks will either become the responder or enter into negotiation with the existing responder.

N.B. If `stopPropagation` is called on the event for any of the negotiation callbacks, it only stops further negotiation within the Responder System. It will not stop the propagation of the native event (which has already bubbled to the `document` by this time.)

> **onStartShouldSetResponder**: ?(event: ResponderEvent) => boolean
> On `pointerdown`, should this view attempt to become the responder? If the view is not the responder, this callback may be called for every pointer start on the view.

> **onStartShouldSetResponderCapture**: ?(event: ResponderEvent) => boolean
> On `pointerdown`, should this view attempt to become the responder during the capture phase? If the view is not the responder, this callback may be called for every pointer start on the view.

> **onMoveShouldSetResponder**: ?(event: ResponderEvent) => boolean
> On `pointermove` for an active pointer, should this view attempt to become the responder? If the view is not the responder, this callback may be called for every pointer move on the view.

> **onMoveShouldSetResponderCapture**: ?(event: ResponderEvent) => boolean
> On `pointermove` for an active pointer, should this view attempt to become the responder during the capture phase? If the view is not the responder, this callback may be called for every pointer move on the view.

> **onScrollShouldSetResponder**: ?(event: ResponderEvent) => boolean
> On `scroll`, should this view attempt to become the responder? If the view is not the responder, this callback may be called for every scroll of the view.

> **onScrollShouldSetResponderCapture**: ?(event: ResponderEvent) => boolean
> On `scroll`, should this view attempt to become the responder during the capture phase? If the view is not the responder, this callback may be called for every scroll of the view.

> **onResponderTerminationRequest**: ?(event: ResponderEvent) => boolean
> The view is the responder, but another view now wants to become the responder. Should this view release the responder? Returning `true` allows the responder to be released.

Transfer props

If a view returns `true` for a negotiation callback then it will either become the responder (if none exists) or be involved in the responder transfer. The following callbacks are called only for the views involved in the responder transfer (i.e., no bubbling.)

> **onResponderGrant**: ?(event: ResponderEvent) => void
> The view is granted the responder and is now responding to pointer events. The lifecycle callbacks will be called for this view. This is the point at which you should provide visual feedback for users that the interaction has begun.

> **onResponderReject**: ?(event: ResponderEvent) => void
> The view was not granted the responder. It was rejected because another view is already the responder and will not release it.

> **onResponderTerminate**: ?(event: ResponderEvent) => void
> The responder has been taken from this view. It may have been taken by another view after a call to `onResponderTerminationRequest`, or it might have been taken by the browser without asking (e.g., window blur, document scroll, context menu open). This is the point at which you should provide visual feedback for users that the interaction has been cancelled.

Lifecycle props

If a view is the responder, the following callbacks will be called only for this view (i.e., no bubbling.) These callbacks are *always* bookended by `onResponderGrant` (before) and either `onResponderRelease` or `onResponderTerminate` (after).

> **onResponderStart**: ?(event: ResponderEvent) => void
> A pointer down event occured on the screen. The responder is notified of all start events, even if the pointer target is not this view (i.e., additional pointers are being used). Therefore, this callback may be called multiple times while the view is the responder.

> **onResponderMove**: ?(event: ResponderEvent) => void
> A pointer move event occured on the screen. The responder is notified of all move events, even if the pointer target is not this view (i.e., additional pointers are being used). Therefore, this callback may be called multiple times while the view is the responder.

> **onResponderEnd**: ?(event: ResponderEvent) => void
> A pointer up event occured on the screen. The responder is notified of all end events, even if the pointer target is not this view (i.e., additional pointers are being used). Therefore, this callback may be called multiple times while the view is the responder.

> **onResponderRelease**: ?(event: ResponderEvent) => void
> As soon as there are no more pointers that *started* inside descendants of the responder, this callback is called on the responder and the interaction lock is released. This is the point at which you should provide visual feedback for users that the interaction is over.

ResponderEvent

Every callback is called with a `ResponderEvent` event. Data dervied from the native events, e.g., the native `target` and pointer coordinates, can be used to determine the return value of the negotiation callbacks, etc.

> **currentTarget**: EventTarget
> The DOM element acting as the responder view.

> **defaultPrevented**: boolean

> **eventPhase**: number

> **isDefaultPrevented**: () => boolean

> **isPropagationStopped**: () => boolean

> **isTrusted**: boolean

> **nativeEvent**: TouchEvent

> **target**: EventTarget

> **timeStamp**: number

> **touchHistory**: TouchHistory
> An object containing information about the history of touches.

TouchHistory

> **indexOfSingleActiveTouch**: number

> **mostRecentTimeStamp**: number

> **numberActiveTouches**: number

> **touchBank**: TouchBank

TouchBank

> **currentPageX**: number

> **currentPageY**: number

> **currentTimeStamp**: number

> **previousPageX**: number

> **previousPageY**: number

> **previousTimeStamp**: number

> **startPageX**: number

> **startPageY**: number

startTimeStamp: number

touchActive: number

Updated July 20, 2023 Edit