Text

A React component for displaying text.

`Text` supports nesting, styling, and touch handling.

In the following example, the nested title and body text will inherit the `fontFamily` from `styles.baseText`, but the title provides its own additional styles. The title and body will stack on top of each other on account of the literal newlines:

Text Functional Component Example

Expo

```
import React, {useState} from 'react';
import {Text, StyleSheet} from 'react-native';

const TextInANest = () => {
  const [titleText, setTitleText] = useState("Bird's
Nest");
  const bodyText = 'This is not really a bird nest.';

  const onPressTitle = () => {
    setTitleText("Bird's Nest [pressed]");
  };

  return (
    <Text style={styles.baseText}>
      <Text style={styles.titleText} onPress=
{onPressTitle}>
        {titleText}
        {'\n'}
        {'\n'}
      </Text>
      <Text numberOfLines={5}>{bodyText}</Text>
    </Text>
  );
};

const styles = StyleSheet.create({
```

Preview My Device iOS Android Web

Nested text

Both Android and iOS allow you to display formatted text by annotating ranges of a string with specific formatting like bold or colored text (`NSAttributedString` on iOS, `SpannableString` on Android). In practice, this is very tedious. For React Native, we decided to use web paradigm for this where you can nest text to achieve the same effect.

Nested Text Example

Expo

```
import React from 'react';
import {Text, StyleSheet} from 'react-native';

const BoldAndBeautiful = () => {
  return (
    <Text style={styles.baseText}>
      I am bold
      <Text style={styles.innerText}> and red</Text>
    </Text>
  );
};

const styles = StyleSheet.create({
  baseText: {
    fontWeight: 'bold',
  },
  innerText: {
    color: 'red',
  },
});

export default BoldAndBeautiful;
```

Preview My Device iOS Android Web

Behind the scenes, React Native converts this to a flat `NSAttributedString` or `SpannableString` that contains the following information:

```
"I am bold and red"
0-9: bold
9-17: bold, red
```

Containers

The `<Text>` element is unique relative to layout: everything inside is no longer using the Flexbox layout but using text layout. This means that elements inside of a `<Text>` are no longer rectangles, but wrap when they see the end of the line.

```
<Text>
  <Text>First part and </Text>
  <Text>second part</Text>
</Text>
// Text container: the text will be inline if the space allowed it
// |First part and second part|

// otherwise, the text will flow as if it was one
// |First part |
// |and second |
// |part       |

<View>
  <Text>First part and </Text>
  <Text>second part</Text>
</View>
// View container: each text is its own block
// |First part and|
// |second part   |

// otherwise, the text will flow in its own block
// |First part |
// |and        |
// |second part|
```

Limited Style Inheritance

On the web, the usual way to set a font family and size for the entire document is to take advantage of inherited CSS properties like so:

```
html {
  font-family: 'lucida grande', tahoma, verdana, arial, sans-serif;
  font-size: 11px;
  color: #141823;
}
```

All elements in the document will inherit this font unless they or one of their parents specifies a new rule.

In React Native, we are more strict about it: **you must wrap all the text nodes inside of a `<Text>` component**. You cannot have a text node directly under a `<View>`.

```
// BAD: will raise exception, can't have a text node as child of a <View>
<View>
  Some text
</View>

// GOOD
<View>
  <Text>
    Some text
  </Text>
</View>
```

You also lose the ability to set up a default font for an entire subtree. Meanwhile, `fontFamily` only accepts a single font name, which is different from `font-family` in CSS. The recommended way to use consistent fonts and sizes across your application is to create a component `MyAppText` that includes them and use this component across your app. You can also use this component to make more specific components like `MyAppHeaderText` for other kinds of text.

```
<View>
  <MyAppText>
    Text styled with the default font for the entire application
  </MyAppText>
  <MyAppHeaderText>Text styled as a header</MyAppHeaderText>
</View>
```

Assuming that `MyAppText` is a component that only renders out its children into a `Text` component with styling, then `MyAppHeaderText` can be defined as follows:

```
class MyAppHeaderText extends Component {
  render() {
    return (
```

```
        <MyAppText>
          <Text style={{fontSize: 20}}>{this.props.children}</Text>
        </MyAppText>
      );
    }
  }
```

Composing `MyAppText` in this way ensures that we get the styles from a top-level component, but leaves us the ability to add / override them in specific use cases.

React Native still has the concept of style inheritance, but limited to text subtrees. In this case, the second part will be both bold and red.

```
<Text style={{fontWeight: 'bold'}}>
  I am bold
  <Text style={{color: 'red'}}>and red</Text>
</Text>
```

We believe that this more constrained way to style text will yield better apps:

- (Developer) React components are designed with strong isolation in mind: You should be able to drop a component anywhere in your application, trusting that as long as the props are the same, it will look and behave the same way. Text properties that could inherit from outside of the props would break this isolation.

- (Implementor) The implementation of React Native is also simplified. We do not need to have a `fontFamily` field on every single element, and we do not need to potentially traverse the tree up to the root every time we display a text node. The style inheritance is only encoded inside of the native Text component and doesn't leak to other components or the system itself.

Reference

Props

accessibilityHint

An accessibility hint helps users understand what will happen when they perform an action on the accessibility element when that result is not clear from the accessibility label.

TYPE
string

accessibilityLanguage iOS

A value indicating which language should be used by the screen reader when the user interacts with the element. It should follow the BCP 47 specification.

See the iOS `accessibilityLanguage` doc for more information.

TYPE
string

accessibilityLabel

Overrides the text that's read by the screen reader when the user interacts with the element. By default, the label is constructed by traversing all the children and accumulating all the `Text` nodes separated by space.

TYPE
string

accessibilityRole

Tells the screen reader to treat the currently focused on element as having a specific role.

On iOS, these roles map to corresponding Accessibility Traits. Image button has the same functionality as if the trait was set to both 'image' and 'button'. See the Accessibility guide for more information.

On Android, these roles have similar functionality on TalkBack as adding Accessibility Traits does on Voiceover in iOS

TYPE
AccessibilityRole

`accessibilityState`

Tells the screen reader to treat the currently focused on element as being in a specific state.

You can provide one state, no state, or multiple states. The states must be passed in through an object. Ex: `{selected: true, disabled: true}`.

TYPE
AccessibilityState

`accessibilityActions`

Accessibility actions allow an assistive technology to programmatically invoke the actions of a component. The `accessibilityActions` property should contain a list of action objects. Each action object should contain the field name and label.

See the Accessibility guide for more information.

TYPE	REQUIRED
array	No

`onAccessibilityAction`

Invoked when the user performs the accessibility actions. The only argument to this function is an event containing the name of the action to perform.

See the Accessibility guide for more information.

TYPE	REQUIRED
function	No

`accessible`

When set to `true`, indicates that the view is an accessibility element.

See the Accessibility guide for more information.

TYPE	DEFAULT
boolean	`true`

`adjustsFontSizeToFit`

Specifies whether fonts should be scaled down automatically to fit given style constraints.

TYPE	DEFAULT
boolean	`false`

`allowFontScaling`

Specifies whether fonts should scale to respect Text Size accessibility settings.

TYPE	DEFAULT
boolean	`true`

`android_hyphenationFrequency` Android

Sets the frequency of automatic hyphenation to use when determining word breaks on Android API Level 23+.

TYPE	DEFAULT
enum(`'none'`, `'normal'`, `'full'`)	`'none'`

`aria-busy`

Indicates an element is being modified and that assistive technologies may want to wait until the changes are complete before informing the user about the update.

TYPE	DEFAULT
boolean	false

`aria-checked`

Indicates the state of a checkable element. This field can either take a boolean or the "mixed" string to represent mixed checkboxes.

TYPE	DEFAULT
boolean, 'mixed'	false

`aria-disabled`

Indicates that the element is perceivable but disabled, so it is not editable or otherwise operable.

TYPE	DEFAULT
boolean	false

`aria-expanded`

Indicates whether an expandable element is currently expanded or collapsed.

TYPE	DEFAULT
boolean	false

`aria-label`

Defines a string value that labels an interactive element.

TYPE
string

`aria-selected`

Indicates whether a selectable element is currently selected or not.

TYPE
boolean

`dataDetectorType` Android

Determines the types of data converted to clickable URLs in the text element. By default, no data types are detected.

You can provide only one type.

TYPE	DEFAULT
enum('phoneNumber', 'link', 'email', 'none', 'all')	'none'

disabled Android

Specifies the disabled state of the text view for testing purposes.

TYPE	DEFAULT
bool	false

dynamicTypeRamp iOS

The Dynamic Type ramp to apply to this element on iOS.

TYPE	DEFAULT
enum('caption2', 'caption1', 'footnote', 'subheadline', 'callout', 'body', 'headline', 'title3', 'title2', 'title1', 'largeTitle')	'body'

ellipsizeMode

When `numberOfLines` is set, this prop defines how the text will be truncated. `numberOfLines` must be set in conjunction with this prop.

This can be one of the following values:

- `head` - The line is displayed so that the end fits in the container and the missing text at the beginning of the line is indicated by an ellipsis glyph. e.g., "...wxyz"
- `middle` - The line is displayed so that the beginning and end fit in the container and the missing text in the middle is indicated by an ellipsis glyph. "ab...yz"

- `tail` - The line is displayed so that the beginning fits in the container and the missing text at the end of the line is indicated by an ellipsis glyph. e.g., "abcd..."
- `clip` - Lines are not drawn past the edge of the text container.

> On Android, when `numberOfLines` is set to a value higher than `1`, only `tail` value will work correctly.

TYPE	DEFAULT
enum(`'head'`, `'middle'`, `'tail'`, `'clip'`)	`tail`

`id`

Used to locate this view from native code. Has precedence over `nativeID` prop.

TYPE
string

`maxFontSizeMultiplier`

Specifies the largest possible scale a font can reach when `allowFontScaling` is enabled. Possible values:

- `null/undefined`: inherit from the parent node or the global default (0)
- `0`: no max, ignore parent/global default
- `>= 1`: sets the `maxFontSizeMultiplier` of this node to this value

TYPE	DEFAULT
number	`undefined`

`minimumFontScale` iOS

Specifies the smallest possible scale a font can reach when `adjustsFontSizeToFit` is enabled. (values 0.01-1.0).

TYPE
number

`nativeID`

Used to locate this view from native code.

TYPE
string

`numberOfLines`

Used to truncate the text with an ellipsis after computing the text layout, including line wrapping, such that the total number of lines does not exceed this number. Setting this property to `0` will result in unsetting this value, which means that no lines restriction will be applied.

This prop is commonly used with `ellipsizeMode`.

TYPE	DEFAULT
number	`0`

`onLayout`

Invoked on mount and on layout changes.

TYPE
({nativeEvent: LayoutEvent}) => void

onLongPress

This function is called on long press.

TYPE
({nativeEvent: PressEvent}) => void

onMoveShouldSetResponder

Does this view want to "claim" touch responsiveness? This is called for every touch move on the `View` when it is not the responder.

TYPE
({nativeEvent: PressEvent}) => boolean

onPress

Function called on user press, triggered after `onPressOut`.

TYPE
({nativeEvent: PressEvent}) => void

onPressIn

Called immediately when a touch is engaged, before `onPressOut` and `onPress`.

TYPE
({nativeEvent: PressEvent}) => void

onPressOut

Called when a touch is released.

TYPE
({nativeEvent: PressEvent}) => void

onResponderGrant

The View is now responding to touch events. This is the time to highlight and show the user what is happening.

On Android, return true from this callback to prevent any other native components from becoming responder until this responder terminates.

TYPE
({nativeEvent: PressEvent}) => void | boolean

onResponderMove

The user is moving their finger.

TYPE
({nativeEvent: PressEvent}) => void

onResponderRelease

Fired at the end of the touch.

TYPE
`({nativeEvent: PressEvent}) => void`

`onResponderTerminate`

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 (e.g., happens with control center/ notification center on iOS)

TYPE
`({nativeEvent: PressEvent}) => void`

`onResponderTerminationRequest`

Some other `View` wants to become a responder and is asking this `View` to release its responder. Returning `true` allows its release.

TYPE
`({nativeEvent: PressEvent}) => boolean`

`onStartShouldSetResponderCapture`

If a parent `View` wants to prevent a child `View` from becoming a responder on a touch start, it should have this handler which returns `true`.

TYPE
`({nativeEvent: PressEvent}) => boolean`

`onTextLayout`

Invoked on Text layout change.

TYPE
(`TextLayoutEvent`) => mixed

`pressRetentionOffset`

When the scroll view is disabled, this defines how far your touch may move off of the button, before deactivating the button. Once deactivated, try moving it back and you'll see that the button is once again reactivated! Move it back and forth several times while the scroll view is disabled. Ensure you pass in a constant to reduce memory allocations.

TYPE
Rect, number

`role`

`role` communicates the purpose of a component to the user of an assistive technology. Has precedence over the `accessibilityRole` prop.

TYPE
Role

`selectable`

Lets the user select text, to use the native copy and paste functionality.

TYPE	DEFAULT
boolean	`false`

`selectionColor` Android

The highlight color of the text.

TYPE
color

`style`

TYPE
Text Style, View Style Props

`suppressHighlighting` iOS

When `true`, no visual change is made when text is pressed down. By default, a gray oval highlights the text on press down.

TYPE	DEFAULT
boolean	`false`

`testID`

Used to locate this view in end-to-end tests.

TYPE
string

`textBreakStrategy` Android

Set text break strategy on Android API Level 23+, possible values are `simple`, `highQuality`, `balanced`.

TYPE	DEFAULT
enum(`'simple'`, `'highQuality'`, `'balanced'`)	`highQuality`

`userSelect`

It allows the user to select text and to use the native copy and paste functionality. Has precedence over the `selectable` prop.

TYPE	DEFAULT
enum(`'auto'`, `'text'`, `'none'`, `'contain'`, `'all'`)	`none`

`lineBreakStrategyIOS` iOS

Set line break strategy on iOS 14+. Possible values are `none`, `standard`, `hangul-word` and `push-out`.

TYPE	DEFAULT
enum(`'none'`, `'standard'`, `'hangul-word'`, `'push-out'`)	`'none'`

Type Definitions

TextLayout

`TextLayout` object is a part of `TextLayoutEvent` callback and contains the measurement data for `Text` line.

Example

```
{
    capHeight: 10.496,
    ascender: 14.624,
    descender: 4,
    width: 28.224,
    height: 18.624,
    xHeight: 6.048,
    x: 0,
    y: 0
}
```

Properties

NAME	TYPE	OPTIONAL	DESCRIPTION
ascender	number	No	The line ascender height after the text layout changes.
capHeight	number	No	Height of capital letter above the baseline.
descender	number	No	The line descender height after the text layout changes.
height	number	No	Height of the line after the text layout changes.
width	number	No	Width of the line after the text layout changes.
x	number	No	Line X coordinate inside the Text component.
xHeight	number	No	Distance between the baseline and median of the line (corpus size).
y	number	No	Line Y coordinate inside the Text component.

TextLayoutEvent

`TextLayoutEvent` object is returned in the callback as a result of a component layout change. It contains a key called `lines` with a value which is an array containing `TextLayout` object corresponded to every rendered text line.

Example

```
{
  lines: [
    TextLayout,
    TextLayout,
    // ...
  ];
  target: 1127;
}
```

Properties

NAME	TYPE	OPTIONAL	DESCRIPTION
lines	array of TextLayouts	No	Provides the TextLayout data for every rendered line.
target	number	No	The node id of the element.

Is this page useful? 

Edit this page

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