Styling

React Native for Web relies on authoring styles in JavaScript and produces optimized CSS.

Style declarations are authored in JavaScript and applied to elements using the **`style`** prop. React Native for Web includes a very small CSS reset that only removes unwanted User Agent styles beyond the reach of React components. All other styles are scoped to components and implemented as "utility" CSS that deduplicates styles and provides reliable rendering.

```
const style = { flex: 1, opacity: 0 };
const Component = () => <View style={style} />;
```

Style performance is improved when styles are defined outside of components using the **`StyleSheet`** API. This creates opaque references to the styles that cannot be directly accessed unless first passed to **`StyleSheet.flatten`**.

```
const styles = StyleSheet.create({ root: { flex: 1, opacity: 0 } });
const Component = () => <View style={styles.root} />;
```

All the React Native components accept a **`style`** property. The value can be a registered object, a plain object, or an array of objects. The array syntax will merge styles from left-to-right as normal JavaScript objects, and can be used to conditionally apply styles:

```
<View style={[ styles.element, isActive && styles.active ]} />
```

To let other components customize the style of a component's children you can expose a prop so styles can be explicitly passed into the component.

```
function List(props) {
  return (
    <View style={props.style}>
      {elements.map((element) =>
```

```
          <View style={[ styles.element, props.elementStyle ]} />
        )}
      </View>
    );
  }

  function App() {
    return (
      <List
        elementStyle={styles.listElement}
        style={styles.list}
      />
    );
  }
```

You have greater control over how styles are composed when compared to using class names. For example, you may choose to accept a limited subset of style props in the component's API, and control the order and conditions of their merging.

Styles API

React Native for Web supports all *long-form* CSS properties. There is no direct support for `@`-rules, selectors, pseudo-selectors, and pseudo-elements, equivalents of which are demonstrated in the styling patterns section below.

Short-form properties

The supported short-form CSS properties accept only a *single* value.

> **borderColor**: ?string
> Accepts only a single value that is applied to all sides.

> **borderRadius**: ?(number | string)
> Accepts only a single value that is applied to all sides.

> **borderStyle**: ?string

> Accepts only a single value that is applied to all sides.

> **borderWidth**: ?(number | string)
> Accepts only a single value that is applied to all sides.

> **flex**: ?number
> Accepts only positive integers, **`0`**, or **`-1`**.
> The value of **`-1`** is non-standard and equivalent to setting **`flowGrow:0`** and **`flexShrink:1`**.

> **inset**: ?(string | number)
> Equivalent to inset.

> **insetBlock**: ?(string | number)
> Equivalent to inset-block.

> **insetInline**: ?(string | number)
> Equivalent to inset-inline.

> **margin**: ?(number | string)
> Accepts only a single value that is applied to all sides.

> **marginBlock**: ?(number | string)
> Equivalent to margin-block.

> **marginInline**: ?(number | string)
> Equivalent to margin-inline. Accepts only a single value.

> **overflow**: ?("auto" | "hidden" | "visible")
> Accepts only a single value that is applied to both axes.

> **overscrollBehavior**: ?("auto" | "contain" | "none")
> Accepts only a single value that is applied to both axes.

> **padding**: ?(number | string)
> Accepts only a single value that is applied to all sides.

> **paddingBlock**: ?(number | string)
> Equivalent to padding-block.

> **paddingInline**: ?(number | string)

> Equivalent to padding-inline. Accepts only a single value.

Non-standard properties

React Native for Web includes compatibility with the following non-standard React Native properties and values.

> **animationKeyframes**: ?Object
> A web-only CSS extension for defining keyframes. The value is an object representing a CSS keyframes definition. For example: `{ '0%': { opacity: 1 } }`.

> **pointerEvents**: ?("all" | "none" | "box-only" | "box-none")
> Equivalent to CSS pointer-events with 2 additional values. A value of `"box-none"` preserves pointer events on the element's children; `"box-only"` disables pointer events on the element's children.

> **writingDirection**: ?("auto" | "ltr" | "rtl")
> Equivalent to direction.

Text style inheritance

Web developers will be used to setting "global" font styles that are applied to the entire document, taking advantage of inherited CSS properties.

```
html {
  font-family: Arial, sans-serif;
  font-size: 16px;
  color: #333;
}
```

However, this approach is problematic for component-based systems, as the rendering of a component may be affected by text styles unexpectedly inherited from its ancestors. React Native for Web adopts the same inheritance restrictions found in React Native: all text nodes **must** be contained by a `Text` component and cannot be rendered directly within a `View`; and text style inheritance is only available within `Text` subtrees.

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

The consequence of this is that default text styles cannot be set on **`View`** for an entire subtree. Although this may seem limiting, it ensures that different design systems can co-exist on the same page and text styles are always encapsulated. The recommended way to use consistent text styles across your application is to create a custom text component (e.g., **`AppText`**) that implements those styles and forms the basis of further app-specific text customization.

```
<View>
  <AppHeaderText>App header text</AppHeaderText>
  <AppText>App default text</AppText>
</View>
```

This still allows for text style inheritance within the **`Text`** subtree.

```
const bold = { fontWeight: 'bold' }
const red = { color: 'red' }

<Text style={bold}>
  I am bold
  <Text style={red}>and red</Text>
</Text>
```

This approach means that components are designed with 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.

Styling patterns

The styling system in React Native is a way of defining the styles your application requires; it does not concern itself with *where* or *when* those styles are applied to elements. As a result, there is no dedicted Media Query or pseudo-class API built into the styling system. Instead, the state of the application should be derived from the equivalent JavaScript APIs that have the benefit of not being limited to modifying only styles.

Responsive layouts

Media Queries may not be most appropriate for component-based designs, as adapting to the dimensions of a container is often preferred. This can be done with the **`onLayout`** prop found on all the core components. If you do choose to use Media Queries, using them in JavaScript via the **`matchMedia`** DOM API has the benefit of allowing you to swap out entire components, not just styles.

Interaction states

Interactions such as hover, focus, and press should be implemented using events (e.g., **`onFocus`**). Components like **`Pressable`** expose interaction state in a ready-to-use form.

Debugging

React Dev Tools supports inspecting and editing of React Native styles. It's recommended that you rely more on React Dev Tools and live/hot-reloading rather than inspecting and editing the DOM directly.

How it works

Style resolution is deterministic and slightly different from CSS. In the following HTML/CSS example, the **`.margin`** selector is defined last in the CSS and takes precedence over the previous rules, resulting in a margin of **`0,0,0,0`**.

```
<style>
  .marginTop { margin-top: 10px; }
  .marginBottom { margin-bottom: 20px; }
  .margin { margin: 0; }
</style>
```

```
<div class="marginTop marginBottom margin"></div>
```

But in React Native for Web the most *precise* style property takes precedence, resulting in margins of **10,0,20,0**.

```
const style = [
  { marginTop: 10 },
  { marginBottom: 20 },
  { margin: 0 }
];

const Box = () => <View style={style} />
```

React Native for Web transforms styles objects into CSS and inline styles. Any styles defined using **StyleSheet.create** will ultimately be rendered using CSS class names. Each rule is broken down into declarations, properties are expanded to their long-form, and the resulting key-value pairs are mapped to unique "atomic CSS" class names.

Input:

```
const Box = () => <View style={styles.box} />

const styles = StyleSheet.create({
  box: {
    margin: 0
  }
});
```

Output:

```
<style>
  .r-156q2ks { margin-top: 0px; }
  .r-61z16t { margin-right: 0px; }
  .r-p1pxzi { margin-bottom: 0px; }
  .r-11wrixw { margin-left: 0px; }
</style>
```

```
<div class="r-156q2ks r-61z16t r-p1pxzi r-11wrixw"></div>
```

This ensures that CSS order doesn't impact rendering and CSS rules are efficiently deduplicated. Rather than the total CSS growing in proportion to the number of *rules*, it grows in proportion to the number of *unique declarations*. As a result, the DOM style sheet is only written to when new unique declarations are defined and it is usually small enough to be pre-rendered and inlined.

Class names are deterministic, which means that the resulting CSS and HTML is consistent across builds – important for large apps using code-splitting and deploying incremental updates.

At runtime registered styles are resolved to DOM style props and memoized. Any dynamic styles that contain declarations previously registered as static styles can also be converted to CSS class names. Otherwise, they render as inline styles.

Updated July 20, 2023 Edit