Direct Manipulation

It is sometimes necessary to make changes directly to a component without using state/props to trigger a re-render of the entire subtree. When using React in the browser for example, you sometimes need to directly modify a DOM node, and the same is true for views in mobile apps. `setNativeProps` is the React Native equivalent to setting properties directly on a DOM node.

> **CAUTION**
>
> Use `setNativeProps` when frequent re-rendering creates a performance bottleneck!
>
> Direct manipulation will not be a tool that you reach for frequently. You will typically only be using it for creating continuous animations to avoid the overhead of rendering the component hierarchy and reconciling many views. `setNativeProps` is imperative and stores state in the native layer (DOM, UIView, etc.) and not within your React components, which makes your code more difficult to reason about.
>
> Before you use it, try to solve your problem with `setState` and `shouldComponentUpdate` .

setNativeProps with TouchableOpacity

TouchableOpacity uses `setNativeProps` internally to update the opacity of its child component:

```
const viewRef = useRef<View>();
const setOpacityTo = useCallback(value => {
  // Redacted: animation related code
  viewRef.current.setNativeProps({
    opacity: value,
  });
}, []);
```

This allows us to write the following code and know that the child will have its opacity updated in response to taps, without the child having any knowledge of that fact or requiring any changes to its implementation:

```
<TouchableOpacity onPress={handlePress}>
  <View>
    <Text>Press me!</Text>
  </View>
</TouchableOpacity>
```

Let's imagine that `setNativeProps` was not available. One way that we might implement it with that constraint is to store the opacity value in the state, then update that value whenever `onPress` is fired:

```
const [buttonOpacity, setButtonOpacity] = useState(1);
return (
  <TouchableOpacity
    onPressIn={() => setButtonOpacity(0.5)}
    onPressOut={() => setButtonOpacity(1)}>
    <View style={{opacity: buttonOpacity}}>
      <Text>Press me!</Text>
    </View>
  </TouchableOpacity>
);
```

This is computationally intensive compared to the original example - React needs to re-render the component hierarchy each time the opacity changes, even though other properties of the view and its children haven't changed. Usually this overhead isn't a concern but when performing continuous animations and responding to gestures, judiciously optimizing your components can improve your animations' fidelity.

If you look at the implementation of `setNativeProps` in NativeMethodsMixin you will notice that it is a wrapper around `RCTUIManager.updateView` - this is the exact same function call that results from re-rendering - see receiveComponent in ReactNativeBaseComponent.

Composite components and setNativeProps

Composite components are not backed by a native view, so you cannot call `setNativeProps` on them. Consider this example:

TypeScript JavaScript

setNativeProps with Composite Components **Expo**

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

const MyButton = (props: {label: string}) => (
  <View style={{marginTop: 50}}>
    <Text>{props.label}</Text>
  </View>
);

const App = () => (
  <TouchableOpacity>
    <MyButton label="Press me!" />
  </TouchableOpacity>
);

export default App;
```

Preview My Device iOS Android Web

If you run this you will immediately see this error: `Touchable child must either be native or forward setNativeProps to a native component`. This occurs because `MyButton` isn't directly backed by a native view whose opacity should be set. You can think about it like this: if you define a component with `createReactClass` you would not expect to be able to set a style prop on it and have that work - you would need to pass the style prop down to a child, unless you are wrapping a native component. Similarly, we are going to forward `setNativeProps` to a native-backed child component.

Forward setNativeProps to a child

Since the `setNativeProps` method exists on any ref to a `View` component, it is enough to forward a ref on your custom component to one of the `<View />` components that it renders. This means that a call to `setNativeProps` on the custom component will have the same effect as if you called `setNativeProps` on the wrapped `View` component itself.

TypeScript JavaScript

Forwarding setNativeProps Expo

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

const MyButton = React.forwardRef<View, {label: string}>((props, ref) =>
(
  <View {...props} ref={ref} style={{marginTop: 50}}>
    <Text>{props.label}</Text>
  </View>
));

const App = () => (
  <TouchableOpacity>
    <MyButton label="Press me!" />
  </TouchableOpacity>
);

export default App;
```

Preview My Device iOS Android Web

You can now use `MyButton` inside of `TouchableOpacity`!

You may have noticed that we passed all of the props down to the child view using `{...props}`. The reason for this is that `TouchableOpacity` is actually a composite component, and so in addition to depending on `setNativeProps` on its child, it also requires that the child perform touch handling. To do this, it passes on various props that call back to the `TouchableOpacity` component. `TouchableHighlight`, in contrast, is backed by a native view and only requires that we implement `setNativeProps`.

setNativeProps to edit TextInput value

Another very common use case of `setNativeProps` is to edit the value of the TextInput. The `controlled` prop of TextInput can sometimes drop characters when the `bufferDelay` is low and the user types very quickly. Some developers prefer to skip this prop entirely and instead use `setNativeProps` to directly manipulate the TextInput value when necessary. For example, the following code demonstrates editing the input when you tap a button:

TypeScript JavaScript

Clear text Expo

```
import React from 'react';
import {useCallback, useRef} from 'react';
import {
  StyleSheet,
  TextInput,
  Text,
  TouchableOpacity,
  View,
} from 'react-native';

const App = () => {
  const inputRef = useRef<TextInput>(null);
  const editText = useCallback(() => {
    inputRef.current?.setNativeProps({text: 'Edited Text'});
  }, []);

  return (
    <View style={styles.container}>
      <TextInput ref={inputRef} style={styles.input} />
      <TouchableOpacity onPress={editText}>
        <Text>Edit text</Text>
      </TouchableOpacity>
    </View>
  );
};
```

Preview My Device iOS Android Web

You can use the `clear` method to clear the `TextInput` which clears the current input text using the same approach.

Avoiding conflicts with the render function

If you update a property that is also managed by the render function, you might end up with some unpredictable and confusing bugs because anytime the component re-renders and that property changes, whatever value was previously set from `setNativeProps` will be completely ignored and overridden.

setNativeProps & shouldComponentUpdate

By intelligently applying `shouldComponentUpdate` you can avoid the unnecessary overhead involved in reconciling unchanged component subtrees, to the point where it may be performant enough to use `setState` instead of `setNativeProps`.

Other native methods

The methods described here are available on most of the default components provided by React Native. Note, however, that they are *not* available on composite components that aren't directly backed by a native view. This will generally include most components that you define in your own app.

measure(callback)

Determines the location on screen, width, and height in the viewport of the given view and returns the values via an async callback. If successful, the callback will be called with the following arguments:

- x
- y
- width
- height
- pageX
- pageY

Note that these measurements are not available until after the rendering has been completed in native. If you need the measurements as soon as possible and you don't need `pageX` and `pageY`, consider using the `onLayout` property instead.

Also the width and height returned by `measure()` are the width and height of the component in the viewport. If you need the actual size of the component, consider using the `onLayout` property instead.

measureInWindow(callback)

Determines the location of the given view in the window and returns the values via an async callback. If the React root view is embedded in another native view, this will give you the absolute coordinates. If successful, the callback will be called with the following arguments:

- x
- y
- width
- height

measureLayout(relativeToNativeComponentRef, onSuccess, onFail)

Like `measure()`, but measures the view relative to an ancestor, specified with `relativeToNativeComponentRef` reference. This means that the returned coordinates are relative to the origin `x`, `y` of the ancestor view.

> **NOTE**
>
> This method can also be called with a `relativeToNativeNode` handler (instead of reference), but this variant is deprecated.

TypeScript JavaScript

focus()

Requests focus for the given input or view. The exact behavior triggered will depend on the platform and type of view.

blur()

Removes focus from an input or view. This is the opposite of `focus()`.

Is this page useful?

Edit this page

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