**Generating Base Styles – React**

In building out the visual presentation of a component, one ends up defining different base styles around it’s different parts, how they look, and how they’re laid out in relation to eachother. Based on the props that the component receives and its internal state, those base styles are combined together in order to properly match the component’s presentation to the current context.

Using [aphrodite](https://www.npmjs.com/package/aphrodite)’s StyleSheet module, one can define these base styles for a component as a Javascript object. Then, by using the StyleSheet.create() method along with the css() method, the javascript is translated into equivalent CSS classes which are applied to DOM nodes of the JSX markup in its render pass.

Defined statically, this works fine as a styling solution for many user interface components. One can block out the major sections of the component and apply custom rules in order to achieve the look and feel they’re going for. However, by creating a **styleGen()** function, one can abstract that process, allowing them to pass in parameters such as props and state in order to conditionally define and apply styles in a manner that offers much more control and flexibility. For example, one might read a prop provided to a <Button /> component called color and apply it to the backgroundColor property of the button’s cradle. In the event that no color prop is provided, that property might be set to some fallback color, such as the current theme’s primary color.

The preparation of styling for a component (especially ones that reach higher levels of complexity) will typically be complemented by a component calibration helper function [insert link to component calibration function documentation]. This helper computes which styles are applied based on props and state, and combines those styles together in a specific order in order to ensure flexibility and reusability (e.g. including custom styles last in the combination chains so that those styles override all others).

**Caveats**

When generating CSS styles using the [aphrodite](https://www.npmjs.com/package/aphrodite) helper library’s StyleSheet.create() and css()to translate Javascript styles to CSS classes, it appears that the css() method does not accept styles that were generated during more than one StyleSheet.create() invocation. So, it is important to note that generating style objects on the fly in the calibrateComponent() method is not a possibility under this setup. Any styles that are to be combined together using the css() method will need to be computed during the one call to StyleSheet.create() in the base style generator function. So for example, any custom styles passed from props will need to be assigned into style objects during this style generator function, and then combined as appropriate during component calibration.

**Pseudocode Walkthrough**

These style generator functions are typically stored in a separate file alongside the main component file that they support as part of a larger component directory structure. The pseudo code that one might follow would look like:

import {StyleSheet} from ‘aphrodite’;

// -> Import other constants and helper functions here.

const styleGen = (props, state) => {

// -> Here, an application’s theme is imported through a

// component’s props. More information on how this is

// accomplished, and theming in general, can be found

// in the theming documentation [link to theming

// documentation]

const {

theme, color, customCradleStyles,

/\* other props you need \*/

} = props;

const {/\* stuff you need from state \*/} = state;

// -> Here you might compute certain styles as results of

// complex helper functions, or run simple ternary

// operators, applying a certain prop to a given style

// property if supplied or falling back to a default value.

const computedBoxShadow =

helperToDetermineBoxShadow(props, state);

const computedBackgroundColor =

(color) ? color : theme.palette.white;

return StyleSheet.create({

style1: {

boxShadow: computedBoxShadow,

},

style2: {

backgroundColor: computedBackgroundColor,

}

customCradleStyles: {...customCradleStyles},

// -> Add in whatever other base styles you need.

});

};

export default styleGen;