**Light and Dark Theme**

**Introduction**

For relatively simple applications, it’s not usually too complicated to achieve a light theme and a dark theme for the UI. In these scenarios, the application doesn’t have too many screens or individual types of elements used to construct them. It is not too hard to assign some colors to different styles that are applied conditionally by flipping a variable at project-level between light and dark theme.

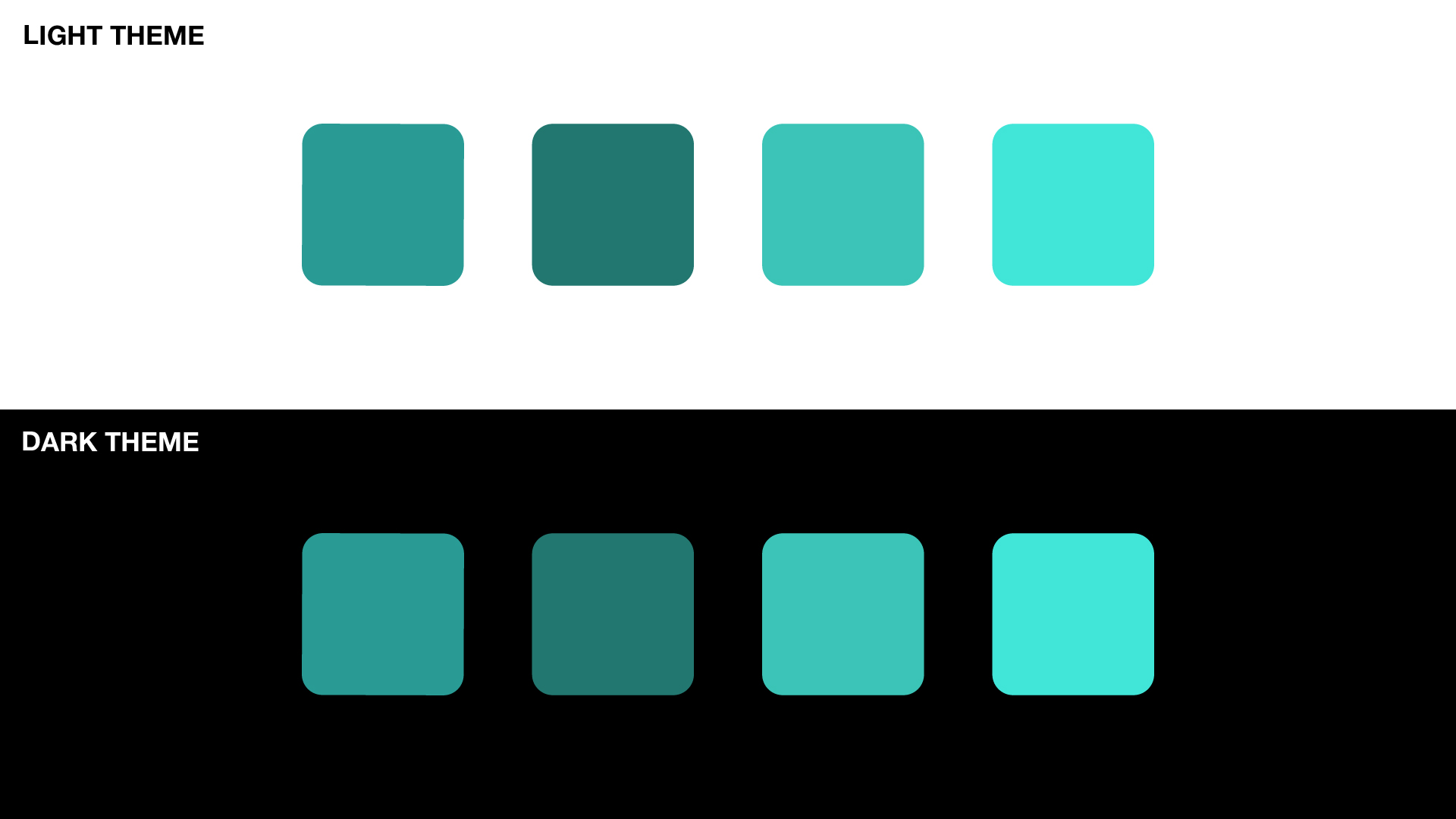
However, as an application’s codebase extends to larger sizes and it becomes necessary to standardize these practices across multiple projects, achieving light and dark themes and the fluid transition between them requires a greater degree of discipline and care. First, one can come up with a naming convention for the different colors used across both themes, then assigning colors to variables with those names at runtime in a dynamic theming structure [link to general theme system documentation module]. Then, by referring to those theme variables in the stylesheets of the components that build up the codebase, this centralized system can be distributed out efficiently and fluidly to them. In the event that these colors are changed, that change will be reliably seen throughout it.

**Choosing Colors**

Choosing colors for a user interface is a complicated enough process before even considering that they will have to work well in both a light and dark theme. One can generally compartmentalize this process into the evaluation of potential primary colors, and then how the greyscale of the palette is distributed out to theme variables in order to generate contrast neutrals for the light and dark theme.

**Primary Colors**

When coming up with the accent colors for an application, if the goal is for them to be versatile enough to work in both the light and dark themes, one will have to test them against both a white and black background.

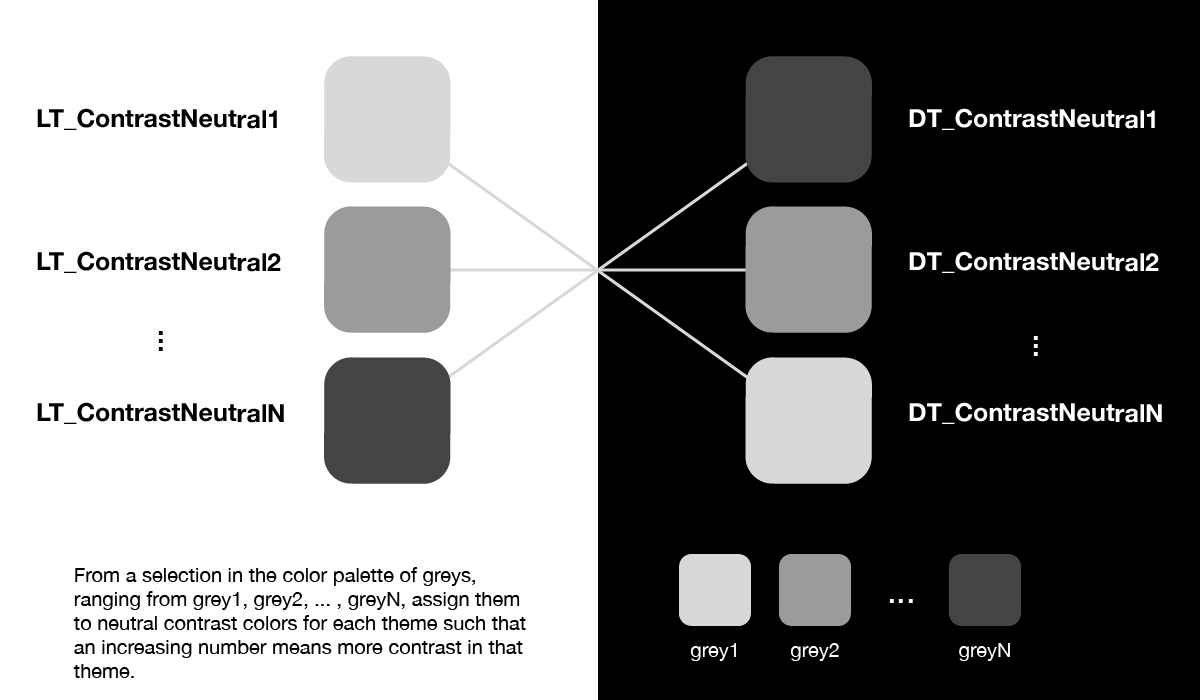


While this selection rationale applies to the primary colors used in the application, it also applies to supporting secondary theme colors and status colors (e.g. success, warning, failure/error).

**Assigning Contrast Neutrals**

In a user interface, if one wants to create contrast between elements, they can do so by assigning different background colors, ranging at the extremes with black and white, and then any number of greys that are chosen. This greyscale that’s put in an application’s color palette presents choices when assigning **contrast neutrals** throughout the user interface’s component parts.

From a choice of greys in a theme’s color palette then, the goal is to assign a range of contrast neutrals for both an application’s light and dark theme. Then they can be applied to component styles throughout the codebase. By numbering these contrast neutrals up to the number of greys available in the palette, one can facilitate a consistent developer experience when assigning them in component styles. Under this criteria, one would expect that the contrast neutrals in a light them would go from lighter to darker grey. When switching over to a dark theme, the greys just need to be inverted so that contrast neutrals go from darker to lighter grey.



**Managing Base Component Styles**

Consistently naming and defining the different colors needed to achieve a light and dark theme is a great start. However, at this point, there is some work remaining towards implementing them across a codebase. Integrated into a theme system [link to theming documentation module], these colors are now available through the withTheme() higher-order function as props. By tuning the base styles of these components to target these theme variables, they can be properly applied at render-time during component calibration. Additionally, if those theme variables are ever assigned to new values, all of the base styles that are looking to their value for instructions will change long with them. More information on base styling and component calibration can be found here [link to base styles documentation] and here [link to component calibration documentation].

What this means in practice when developing base styles is that portions that deal with layout colors will need to have a separate style rule for both light and dark theme. If building a button for example, one would create separate style rules for its overall cradle for both themes, so that under a light theme, its background color would be black, and under a dark theme, its background color would be white. Omitting much more code that would be typically involved in the base styles of a button component, its style generation function might look something like:

import {StyleSheet} from ‘aphrodite’;

const styleGen = (props) => {

const {theme} = props;

return StyleSheet.create({

buttonCradle: {

padding: theme.spacing.two,

display: ‘flex’,

flexDirection: ‘row’,

justifyContent: ‘center’,

alignItems: ‘center’,

},

buttonCradleLightTheme: {

backgroundColor: theme.palette.black,

},

buttonCradleDarkTheme: {

backgroundColor: theme.palette.white,

},

});

}

export default styleGen;

**Assigning Styles Correctly at Render-Time Based on Theme Status**

As is developed further in the component calibration documentation module [link to component calibration module], a component can be tuned to the snapshot presented to it by props and state by running through a calibrateComponent() function. This function takes in props and state, and base styles, conditionally applying those styles in order to present components properly. If the types of conventions introduced in the preceding section of the documentation module are followed, the current status of the application’s theme can be read from props, and the component can be colored according to this value. Continuing with the simplified button example from above, this component calibration function might look something like:

import {css} from ‘aphrodite’;

import {LIGHT\_THEME, DARK\_THEME} from

‘../../constants/themeConstants;

const calibrateComponent = (props, state, styles, themeInfo) => {

const {theme} = themeInfo;

const buttonCradleThemeStyle =

(theme === DARK\_THEME) ?

styles.buttonCradleDarkTheme : styles.buttonCradleLighTheme;

const buttonCradleStylechain = css(

styles.buttonCradle,

buttonCradleThemeStyle

);

return buttonCradleStylechain;

};

export default calibrateComponent;

**Other Resources to Consider**

When building an architecture flexible enough to handle light and dark themes across it, the main concern will of course be assigning color appropriately. However, there are other resources to keep in mind when building towards robust theming systems.

One example of such a resource are box shadows. When saving shadow presets, they will also need colors assigned to them. In this case, the most flexible option would be to assign ranges of box-shadow for light and dark theme, referring to appropriate color variables depending on the context. In the context of defining a styleguide object, these box shadows might look like:

const styleGuide = {

palette: {...},

spacing: {...},

fonts: {...},

typography: {...},

styles: {

...,

boxShadows: {

lightTheme: {

one: ‘0 1px 3px rgba(0,0,0,0.33)’,

...,

},

darkTheme: {

one: ‘0 1px 4px rgba(255,255,255,0.33)’,

...,

},

},

...,

},

};