**Styleguide – React**

**Introduction**

Building user interfaces is a matter of applying a number of foundational design principles together over and over again in different ways in order to achieve a certain design. Throughout a codebase, these principles, such as colors, distance (aka margin and padding), fonts, typography, and shadows are expressed as styling code, adding up to the entire application as it is rendered out.

One can establish a centralized styleguide residing in one place in the codebase in order to define these aspects, and commit them to code. If the application’s components then derive their styles from that structure (e.g. referring to one of its color presets instead of defining the color locally), it is possible to achieve a consistent look-and-feel, and reduces the complexity involved in refactoring the entire codebase when one wants to make systemic changes to that look and feel.

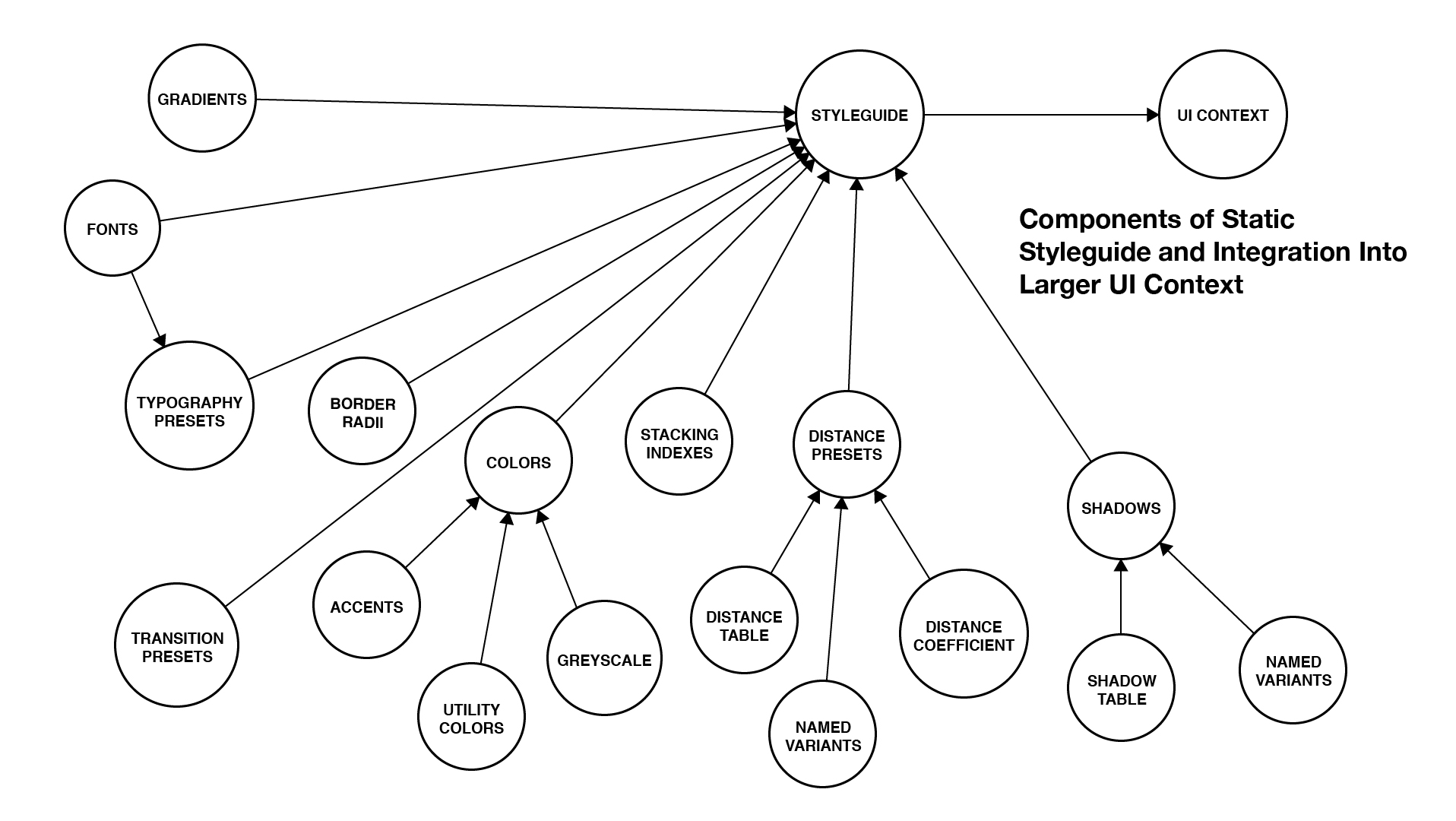
The range of aspects that can be defined and captured in such a styleguide will surely include many more than are included in this document. However, some key aspects which will be of concern to any developer looking to create consistent user interfaces include:

* A distance system which can set the stage for consistently and flexibly applying margin and padding throughout the codebase.
* A color palette that captures the interface’s main styling needs and also accounts for things like establishing a greyscale and error, warning, and success statuses.
* Gradients applied to different interface elements.
* Font presets which are then referred to in the theme’s typography presets.
* The definition of typography presets needed to present text in its appropriate context throughout the interface.
* Stacking presets which set the stage for different components to be presented on top of eachother throughout the interfaces (e.g. backdrop overlays, modals, toasts, notification cards, side drawers, etc…).
* Border radius presets defined in order to achieve different roundness on shapes.
* Sizing presets for things like icons in different contexts (e.g. size of icons used for buttons in header bars or bottom tab navigators).
* Transition presets for different aspects of the user interface (e.g. background color transitions).

Once this styleguide has been defined and committed to a data structure (a nested object in a Typescript context, it can be loaded into a dynamic data structure which is responsible for maintaining and providing an up to date user interface context to other components when they’re going into their render passes. As component’s go to style themselves, their styling code can have this information passed to it dynamically at render time, allowing the user interface to stay dynamically linked to this styleguide at all times.

In addition to the static information provided in the styleguide, this UI Context structure is capable of changing the sets of current primary, secondary, tertiary, quaternary, and quintenary colors, and is capable of assigning whites, blacks, and greys in a dynamic fashion so as to be able to facilitate a light and dark mode in the interface. Beyond the management of colors in the application, this dynamic context structure can also track the visibility status of elements such as backdrops and alerts, along with the APIs needed to control their status from other components.

A diagram can be found below which outlines the different parts of the styleguide and then denotes how it is included into the user interface context structure at run time. More information on this dynamic structure can be found in the UIContext documentation module [Include hyperlink to UIContext documentation module].



**Distance System**

**TL;DR**

Create a system of distance presets which can be applied to contexts like margin and padding. Each preset is labeled by number (e.g. ‘one’, ‘two’, ‘three’, etc…), where increasing multiples of a constant distance coefficient gives you a linear progression. Tweaking the distance coefficient up and down by small amounts will correspondingly increase and decrease the proportionate spacing that is set between elements in the user interface.

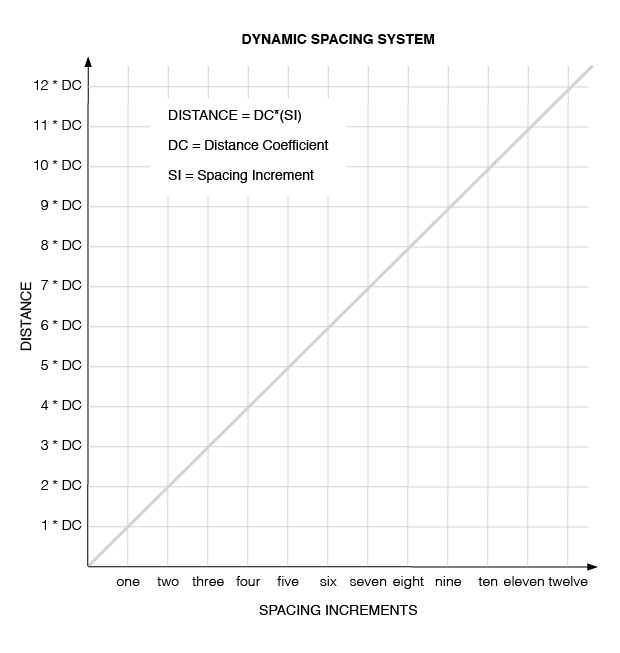
**Explanation**

When starting out with user interface design, it is common to hardcode margin and padding values directly into component styles on a case-by-case basis. While this strategy works fine as one is building relatively small interfaces it starts to compound technical debt as the overall size and complexity begins to increase to intermediate levels and above.

Faced with this rising complexity, the next thought might be to abstract a number of spacing presets into a central theme file, running along some semantic scale with names such as ‘extra small’ all the way up to ‘extra large’. If one wants to improve consistency, they might ramp up the spacing amounts along these presets by the same increment at every step of the way. However, these named presets present an issue concerning running out of easy and consistent way to keep coming up with names they are added to the style guide. This semantic naming convention also faces multilingual barriers when working with developers who speak another primary language than the one the codebase is written in.

What next then? Well, by assigning each steady increment with a number instead of a semantic name, one can keep this consistent progression of spacing but have an indefinite and extensible methodology for naming those presets. And by tuning the coefficient at which each increment is multiplied by, they can continuously, rather than discretely control the relative margins and paddings used across the entire user interface. Additionally, if the zeroth increment is to start at some value other than 0px, this scale can start shifted to some positive or negative amount.

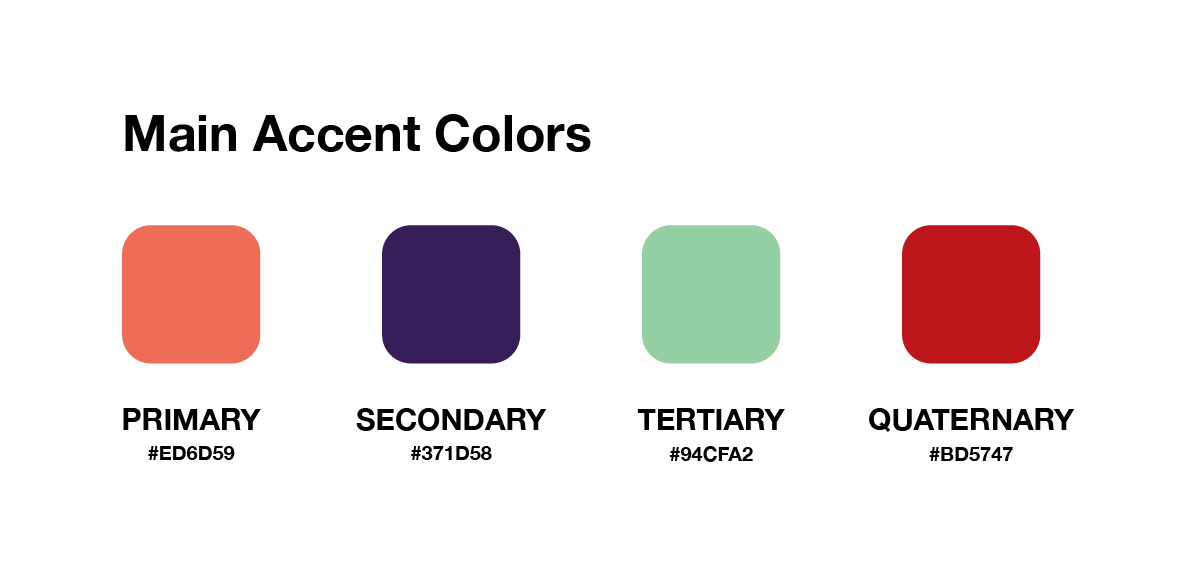
Together, the mapping of spacing increment to amount of spacing actually meted out forms a linear equation of the form y = mx + b, where each increment (**x**) is mapped (**y**) to some multiple of the theme’s spacing coefficient (**m**), plus or minus some shift from zero, observed at the zeroth increment (**b**).



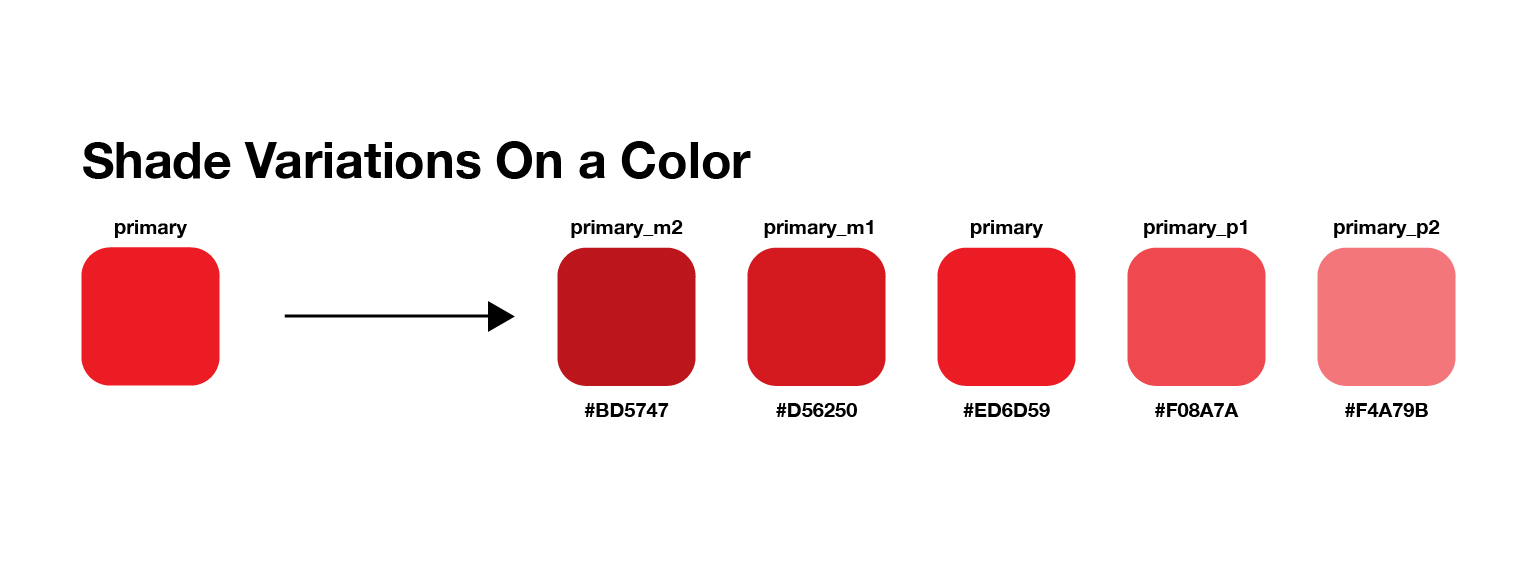
**Color Palette**

Effective user interfaces require a wide-ranging use of color in order to be effective. In order to create contrast and context between elements, a good palette will certainly need to include black, white, and a range of different greys. In order to leave space for adding more or taking away greys in the future, it is also wise to name them in a numbered rather than semantic fashion (e.g. grey1, grey2, … greyN). Then, in order to indicate success, warning, and error states, one could choose a shade of green, yellow, and red. Keeping in mind users who experience varieties of color blindness however, those green and red status colors will also need some alternative which creates contrast for those users (e.g. green and purple for those who can’t make out red very well).

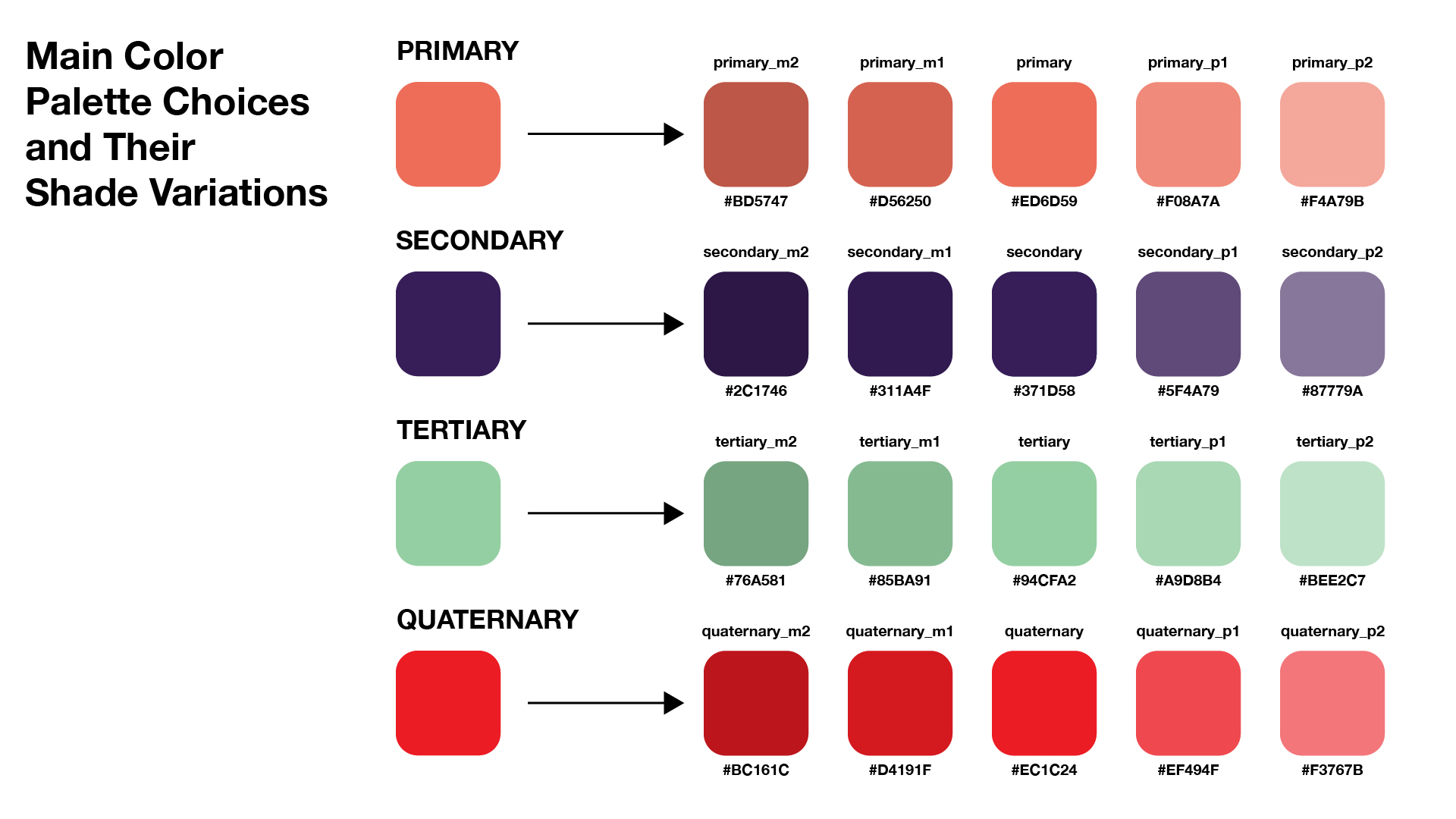
Through the course of designing a user interface, it is common to come up with a number of key colors which are applied to things like text, buttons, and banners. These accent colors set the tone for the interface and can be used to reinforce things like branding strategies. This styleguide is designed to allow for the choice of up to five such colors.



Now, for each of those colors, it is useful to have a variety of shades of them available to us as well. They can provide a subtle variety of color throughout the interface while still staying congruent to the (up to five) main choices made. These gradations of shade on a color can be auto-generated with resources like [this](https://www.tutorialrepublic.com/html-reference/html-color-picker.php) or one can estimate it themselves. The figure below demonstrates how these variations are labeled in the styleguide for a color. The m1, m2 suffixes are meant to denote a reduction in brightness and the p1, p2 suffixes are meant to signify the opposite.



Once these shades are all generated for the theme colors chosen, one ends up with up to 25 main colors to use throughout the interface.



In order for user interface components to remain maximally flexible over time, their styling rules then make reference to whatever the current theme’s primary, secondary, tertiary, and quaternary colors are. As different colors get chosen from the palette in order to fulfill this role, all components that followed this convention should smoothly change over to them in a smooth and consistent way.

Another area of concern when coming up with color palettes is looking ahead to the possibility that one might want to be able to provide a light theme along with a dark theme to the users of an application. With black and white in the color palette, along with a range of greys, this is highly achievable with disciplined naming conventions and a flexible styling strategy for components. One can read more about this in the dark/light theme documentation module [link to dark/light theme module].

**Fonts**

Foundational to the effectiveness of a user interface is the fonts with which text is presented to users. It’s also very important that these decisions on typeface be carried out in a consistent way across the UI. Therefore, one can define a number of primary and supporting fonts that will be used across the interface in this static styleguide. Then, as components need to style text, the typography presets that are applied (next section) can refer to these fonts.

**Typography**

Having the fonts that will be used throughout an application is a good first step, but it’s also important to think about the different contexts in which text will be presented (e.g. big splashing headlines vs. small footnotes and captions), and then reflect these choices in preset combinations of typeface, font weight, line height, kerning, color, and so on. In order to make these presets flexible as well, it is important to assign fonts programmatically, according to the fonts that were entered into the ‘fonts’ part of the styleguide. That way, if one switches out the fonts they choose in a styleguide, it will be applied across corresponding typography presets automatically. Examples of these types of typography presets can be found in design systems such as [Material Design](https://material.io/design/typography/the-type-system.html#type-scale) and [Ant Design](https://ant.design/docs/spec/font).

**Shadows**

In order to give the appearance that an element in the UI is floating above the “surface” of a background, one can assign shadows to them. This spot in the styleguide is a place to define such presets, which can be applied to elements like buttons and cards and panels as needed. One suggestion is to define a scale of shadow presets (‘one’, ‘two’, ‘three’, etc…) which appear to raise the element they’re applied to further and further off the background surface.

**Stacking**

When putting together components, markup that is nested within parent elements will be displayed on top of it. Generally, this is enough to properly display the elements of an interface and provide the user experience needed in that context. However, there are certain contexts (e.g. modals, notifications, side-drawers, toasts) where content that is taken outside of the default layout flow needs to be displayed on top of what is already there. In this case, it is necessary to assign higher z-index values to the styles of those components in order to achieve this desired stacking effect.

However hard-coding those z-index values directly into the styles of these components can become un-maintainable very quickly as one comes up with increasing levels of relative stacking between UI elements and they lose track of all of the places in the codebase that they’ve assigned them.

One solution is to assign these stacking presets in a centralized theme, assigning them with intuitive names relating to their context and maintaining their relative order, even as new stacking presets are introduced (e.g. making sure that the z-index of a backdrop is always underneath that of a modal).

**Border Radius Presets**

A common variant that can be applied to many UI components (e.g. text inputs, modals, cards) is to change the curvature of their border corners from orthogonal (90 degrees) at one end to completely rounded at the other. By defining what some of the border radiuses should be one can ensure that when curvature is applied to shapes in the user interface, it is done in a consistent, and easily refactorable manner.

**Sizing Presets**

This is more of a general category for aspects about the user interface that need to be sized consistently according to certain criteria or contexts. For example, when deciding on the sizes of icon buttons, there are certain contexts in the user interface, such as header bars, where you might like to enforce a uniform size.

**Transition Presets**

This is a place in the styleguide where one can define common transitions that can take place in different parts of the user interface throughout its lifecycle. For example, when hovering the cursor over different elements such as buttons or items in a list, one might want the background color of that element to transition to some other one over a certain period of time and with a certain Bezier curve. By defining that css transition rule in a central styleguide, it can be consistently applied over elements where that kind of background color transition is desired.

**Gradient Presets**

There are different places in the interface where one might like to assign color gradients (e.g. the background of buttons or the backgrounds of entire pages). The styleguide includes a set of these so that developers can create them as wanted, and easily refer to them when needed in their styling code.