Themes

Overview

Themes use design tokens to codify design decisions. Note that designers aren't providing hardcoded values: instead the design decisions are references to the design options available in the brand palette (or the system theme, see Appearances below for details). This coupling of a theme to a palette guarantees brand consistency.

Themes are an abstracted layer that sits between the brand palettes and the components, making it possible to customise the look and feel of components freely in a variety of different contexts. i.e. different brands or different expressions of a brand. So the same component can be reused for TELUS e-commerce, TELUS internal dashboard applications, Koodo, Public Mobile or any other brand in the future. Please note that currently only UDS Base components use themes, but in principle any new component can leverage them in the future.

Our notion of them-ability defined here is much more powerful than that typically offered by generic design systems. Rather than only allowing for a relatively small number of customisation points (e.g. primary colour, size and font styles) a UDS theme can define any component attribute as well as the component variants. So one theme might define a primary and secondary button while another defines default, large, danger and info buttons. UDS themes do not limit UX designers to the opinionated constraints predefined by a 3rd party, nor do they need to make compromises with other brands. Each brand theme can independently define whatever it needs, without having to change the code of the components themselves.

It really is the best of both worlds. Designers have the freedom to be creative, while developers focus on the functionality, reusability and performance of the components.

Anatomy of a theme

A theme as a detailed design specification document that uses a strict format to define the following:

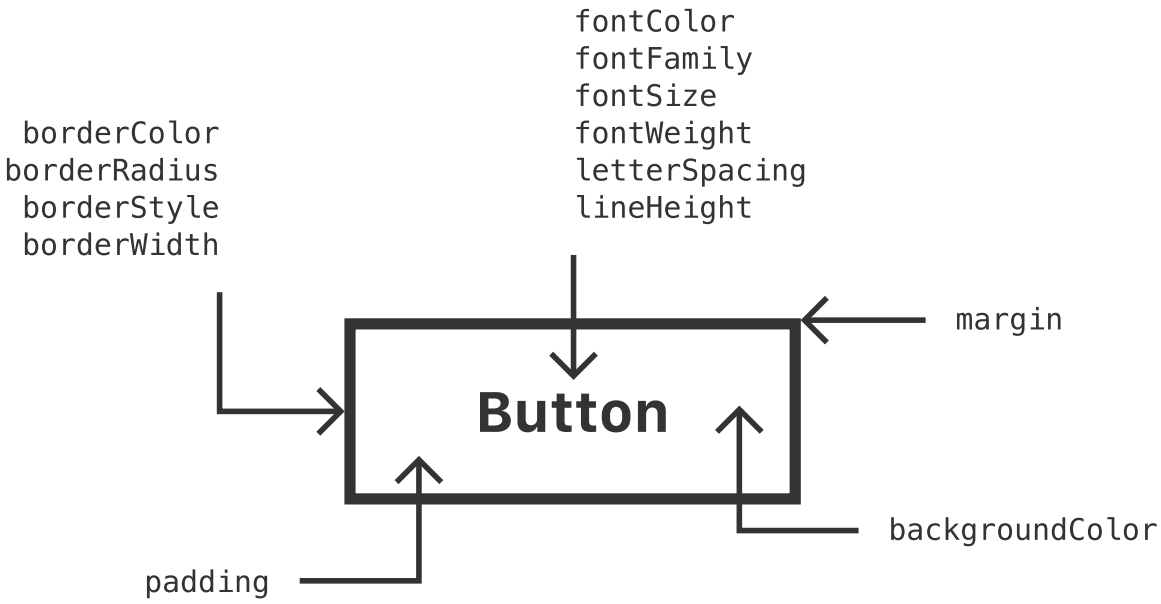
Appearances

Appearances define the **component states** (e.g. hover, focus, pressed, inactive) and the **component variants** (e.g. primary, secondary, warning).

- **All used appearances must be defined.** The theme editor is responsible for defining all the appearances that are used for a component via the `appearances` property. System-wide appearances can be imported from system-tokens. Editors should only define the appearances that they use in rules (for example, if a component supports an `inactive` appearance, but the particular theme doesn't make use of it, it shouldn't be imported into the theme).
- **Declare all component variants.** All component variants should have a `type` of `variant`.
- **Strict use of the system-theme-tokens.** Appearances should only be added to system-theme-tokens if the value of that appearance is controlled by the system (i.e., UDS Base code provides the value). For example, `iconPosition` is a valid system appearance for `Link` because it is provided by a UDS Base prop; `size:small` is not a valid system appearance for Typography because it is not set by the UDS Base code.

Tokens

Tokens are the codified design decisions which map values from the brand palette to the specific properties of the component in it’s default state.



- **Tokens must be references.** Whenever defining a theme, all of the design tokens you use must be references to either the respective brand palette or the system object; for example:

```
color: '{palette.color.green}',
textUnderline: system.textLine.none,
```

- **All tokens must be defined.** In the root `tokens` object for a component in a theme, all tokens available in the schema must be defined and resolve to values other than `undefined` (see ADR design tokens always required). Tokens defined in `rules` must be a subset of those defined in the root `tokens` property – each rule can define as many or as few of the schema tokens as it likes.
- **No logic in theme files.** Theme files are written as static JSON files containing references to tokens. During the theme build, this will be converted to the appropriate platform format, allowing themes to be built for any platform.

Rules

Rules are used to define any exceptions from the default state. This is where we specify when alternative tokens are to be used. Rules take the form of simple `if` statements, making them easy for humans to read and understand (e.g. `if focus = true` then set the `outerBorderColor` to `greenAccessible`). Note: every property defined in `rules`, first needs to have its default value set in `tokens`.

Theme schema

The theme schema is the contract that defines which properties of a UDS Base component can be themed e.g. `colour`, `fontSize` or `fontWeight`. This list of properties is then defined with palette values in the tokens section of the theme file (as described above).

Changes to a single theme only affect that theme, but changes to the theme schema affect all the themes that use it.

Common reasons you might need to alter the contract between UDS Base components and themes include:

- Adding a new UDS Base component
- Adding, editing, or removing a design token consumed by a UDS Base component
- Adding, editing, or removing the appearances supported by a UDS Base component

If you're adding a new UDS Base component, be sure to use the `npm run create-component <ComponentName>` command, as this will bootstrap all the required file changes for a new component.

Subthemes (extending / overriding themes)

It's possible to take an existing theme and add variants, add rules or modify default tokens by creating a **subtheme**. A subtheme is just like a theme but without the requirement to define every component and every token - instead, it is expected to be merged over another, complete theme.

For example, a "dark mode" theme could be created which swaps dark and light colour tokens while leaving everything else untouched, or a "compact" theme could be created which adds smaller variants and reduces default spacing and sizing.

1. Create a theme package for the subtheme just like existing theme packages, but with a `build` script that calls `system-tokens-build-subtheme`, instead of `system-tokens-build-theme` which requires its input to define every token for every component.

2. Use the `mergeThemes` utility from `@telus-uds/system-constants` to merge it over the relevant main theme. This can be done either:

- At run time: recommended for apps that need to switch the subtheme on and off so the file sizes of the theme and subthemes are as small as possible. Import the themes and subthemes on the frontend, then pass the merged theme returned from `mergeThemes(baseTheme, subTheme)` to the `ThemeProvider`.
- At build time: recommended for themes that are standalone and not switched. Call `mergeThemes` on the original theme `json` files and pass the merged `json` it returns to `system-tokens-build-theme`.

Note that small, one-off theme overrides can be applied to individual instances of components using the `tokens` prop. Subthemes are best for cases where every instance of a component or variant in an app should change in a common way.

Theme options

Sometimes you may need to provide some additional parameters to the `ThemeProvider` to specify how the theme is supposed to be handled. In order to do that, one can use `themeOptions` prop to specify the options. Currently, the following theme options are available:

- forceAbsoluteFontSize: a flag indicating whether text-related components have to use absolute font sizing (in pixels) instead of relative font sizing (in `rem` units).

Multi-brand applications

Historically, we'd need to write application logic to map one brands components to another. This can be tricky when those brands were never designed to be compatible. In the scenario where it was not possible to map one-to-one, more application code would need to be written to fill any gaps. Leading to lots of localised application specific logic which was difficult to share.

Theme schemas solves this problem. You now create themes dedicated to specific contexts, aligning component variants is simple. All those design

decisions are codified, versioned, then packaged up and distributed for use by any team in any application.

The UDS theming system makes this possible, **so long as the two themes are interoperable**. By interoperable we mean that they can have different `Tokens` and different `Rules`, but they need to have the same `Appearances` (i.e. component variants and states).

Edit this page