Architecture

> **CAUTION**
>
> We are in the process of merging the code base back into a mono-repo so this article will soon be out of date. Please get in touch if you would like to discuss this change.

UDS takes a system-of-systems approach to managing design systems. The UDS architecture is designed to enable decoupling, reuse, contribution, and continuous development. This guide provides an overview of that architecture.

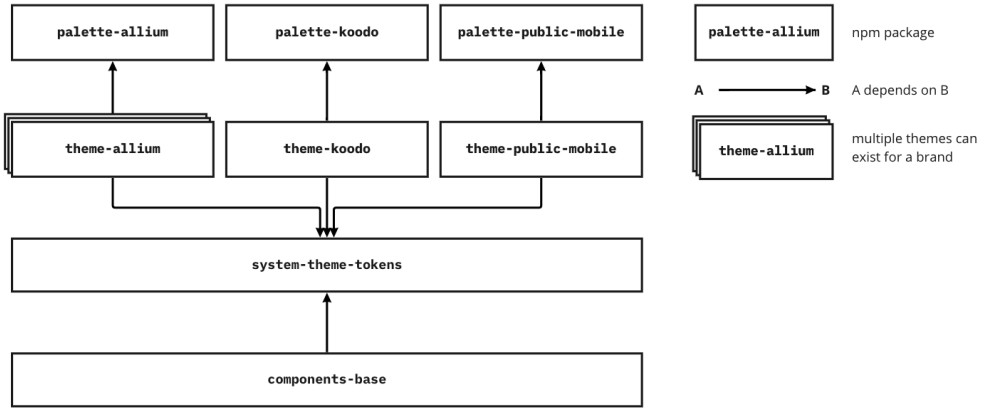
Public architecture

At the core of the UDS architecture stand 3 consumable layers: palettes, themes, and base components. Each of those three layers are distributed as an independently semantically versioned npm package in the `@telus-uds` scope.

Although there is no technical constraint on this, UDS was designed with the idea of having exactly one palette per brand. The palette distills the visual identity of a brand as a versioned collection of design tokens.

Conversely, there is exactly one base components package which is used by all brands. This is possible because base components are themable: they accept design tokens which control their look and feel. A complete collection of design tokens to style the base components is a UDS theme. Effectively a theme is a mapping from a brand palette onto the base components. Brands can have as many or as few themes as they choose.

Themes must provide a complete set of design tokens for the base components. All tokens must be present, there may be no additional tokens, and all tokens must be of the correct type (you can't pass a `color` value into a slot that's expecting a `size` token). This schema of design tokens is moderated by package called `system-theme-tokens` which decouples the themes from the base components. There is no direct dependency between themes and base components, instead both implement the design token schema defined in `system-theme-tokens`. In this way UDS can operate in a more decoupled manner, and teams are free to create themes independently of the core team.



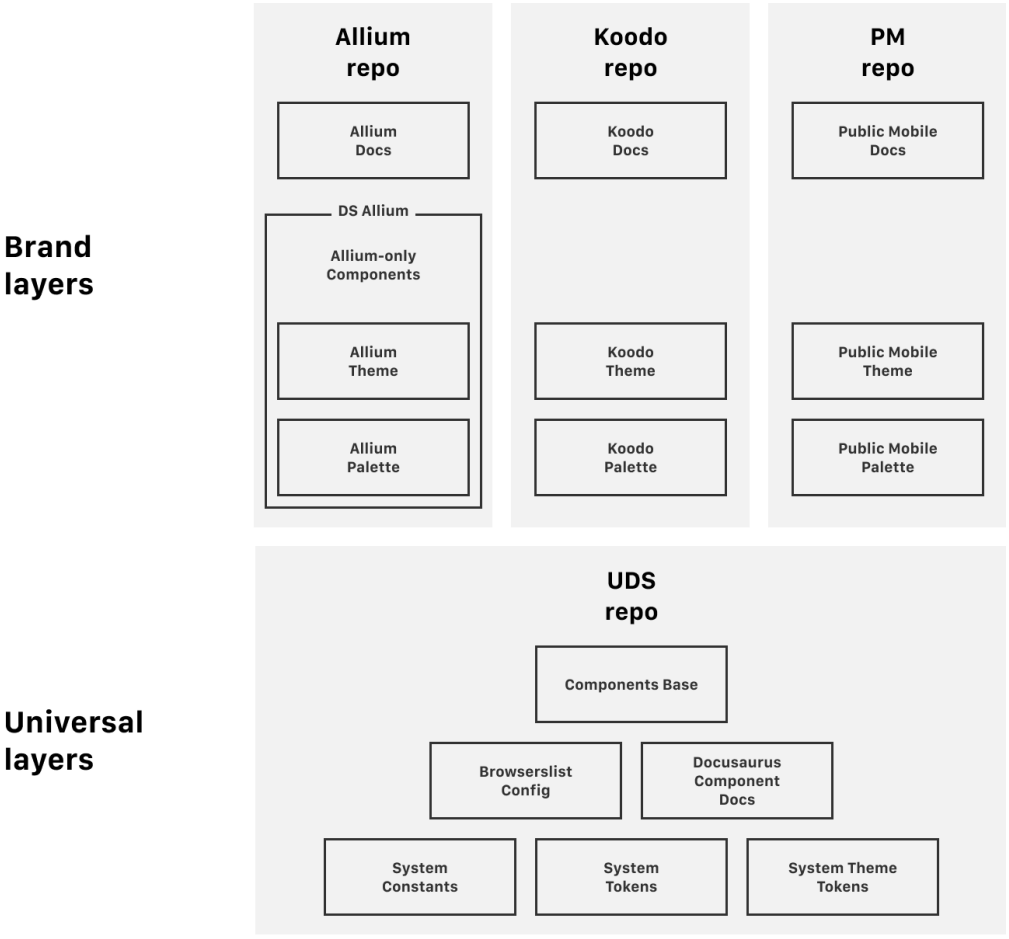
Core npm dependencies in UDS: note that themes and base components have no direct codependency.

It's worth noting that although we anticipate teams using palettes as a standalone resource, it only makes sense to use a UDS theme in conjunction with the base components.

Code organisation

A core principle of UDS is to provide a decoupled, open system. One way in which we achieve this is by distributing ownership of code and constituents of the system-of-systems outside of the core team.

In particular the branded parts of UDS are owned and operated by the respective brands. This enables brands to own, iterate, and publish branded resources and design systems with autonomy. In this model the core UDS team is responsible for the base components, supporting tooling, and DX across the system. Below we will cover in more detail the tooling and versioning model which enable this decoupling.



Distribution of packages within the UDS ecosystem. Note that `ds-allium` *is the distribution package for the TELUS brand and it exposes all web and multi-platform components with the TELUS theme applied.*

Supporting tooling

The public design systems which make up UDS are underpinned by a host of tooling packages which facilitate consistency and a great DX across UDS. Here we give a basic accounting of these and their purpose, more details can be found in the READMEs of the individual packages.

`@telus-uds/browserslist-config`

This package makes the browserslist configuration that we use to build the base components and UDS documentation websites available for all UDS consumers so that browser support is consistent across the ecosystem.

`@telus-uds/docusaurus-*`

We use Docusauraus to build documentation websites for UDS design systems. We maintain docusaurus plugin packages to support the reuse of base component documentation across all design systems which utilise base components.

`@telus-uds/system-constants`

This package houses system-wide constant values (*e.g.* viewport dimensions) for consistency across all UDS user interfaces. it's widely used by other packages within UDS and can be used when building new component libraries or other UDS extensions.

`@telus-uds/system-theme-tokens`

This package contains the design token schema for the base components. It is used by the base components to validate that a theme implements a matching version of this schema. It is used in the theme build process (see below) to validate that a theme contains valid design token values per the schema. It functions as the contract between themes and base components.

`@telus-uds/system-tokens`

This package contains all of the design token tooling which supports palettes and themes within UDS. It implements three primary functions for each of palettes and themes

1. Validate the inputs to a palette or theme
2. Build a palette or theme: transform those inputs into platform-appropriate outputs
3. Automate semantic versioning of palette and theme packages.

Details of how these functions work can be found in the package README.

Change management and version control

This topic is covered in more depth in the versioning guide, here we just lay out at a high level how version control enables parts of the system to change independently over time, and how brands are empowered to update and release their design systems with autonomy according to their own priorities. The

motivation here is to enable iteration on all aspects of the UDS system-of-systems, and give all users confidence in accepting upstream changes.

Palette changes

Palettes are one of the most fundamental aspects in UDS—all branded experiences are built on top of palettes. In order to let brands evolve, it's critical that we enable palettes to be iterated on, but at the same time we need to give confidence to consumers that changes won't break their applications. As palettes are fundamentally data structures we rely on (and automate) an extremely strict semantic versioning approach. This is covered in more detail in the versioning docs but essentially changing the value or presence of any key in the data structure counts as a breaking change and hence a major version. This enables designers to evolve palettes over time, and allows consumers to pull in non-major changes with no risk of breaking an application. We use a combination of beachball and custom code to automate the generation of semantic version bumps when publishing palettes. Both the code supporting this and more detailed technical documentation can be found in the `system-tokens` package.

Theme changes

As with palettes, themes are used as a foundational aspect in brand experiences. Again it is important to enable these to evolve whilst giving consumers confidence that these changes won't break their applications.

Themes are always tied to a specific version range of the base components via a matching dependency on the `system-theme-tokens` schema version. Themes implement a specific version of the schema, validated by that version of `system-theme-tokens`. Thus there are two ways that themes may change:

1. Without a change in the base component library (same schema version)
2. In response to updates to the base component library (new schema version)

Independent theme changes

In this case theme owners can change any of the design tokens in their theme to adjust the visual treatment of the base components. As with palettes we provide automated semantic versioning of themes via the `system-tokens` package.

Again this is covered in more detail in the versioning guide. Note that these types of changes require no input or coordination with either the UDS core team, or other brands.

Base components and theme schema changes

If a new base component has been created, or new design token options are added to existing components, changes will need to be made to the theme schema. This will be coordinated by the UDS core team. Once that new theme schema is in place, a new version of `system-theme-tokens` will be published, and theme owners can choose to either integrate those new design tokens into their theme and so upgrade to the latest version of the base components, or stick with their current version and defer upgrading to a later date. In this way brands and theme authors are free to operate their design systems with autonomy, and don't need to coordinate on the timing of changes to the base components.

Component library changes

Changes to the base components which change the design tokens that can be passed in will always result in a change to the `system-theme-tokens` package. This is the scenario described immediately above. For other changes to the base components (*e.g.* bug fixes, new functionality with no visual implications) we can create new versions of the base components without altering the `system-theme-tokens` schema, and they will automatically work with all existing themes.

For all component libraries—including the base components—we advocate for a pragmatic semantic versioning strategy. It's not practical to provide automated tooling to generate these semantic version bumps, instead this is left to developer discretion.

Edit this page