Creating component libraries

UDS was designed to be an extensible tooling platform for building and supporting an ecosystem of component libraries. This document gives guidance on creating your own component library with UDS.

First, we recommend you consult with your brand team and/or the UDS team before starting. Outcome teams are very much encouraged to build the component libraries that they need, but it's useful for brand teams to be able to understand what work is planned, and to help reduce duplication of effort.

Component library taxonomy

At the heart of UDS is a component library called UDS Base. This is a set of themable, atomic, React Native components. It ships as the `@telus-uds/components-base` npm package. We also have a component library for TELUS brand which encapsulates the UDS Base components, and extends them with some TELUS-specific web components. This ships as the `@telus-uds/ds-allium` npm package. We anticipate that we will soon have matching `@telus-uds/ds-koodo` and `@telus-uds/ds-public-mobile` packages to match.

The above libraries may all be considered global. However it is also very likely more specific component libraries will be needed to match a particular customer experience or context. In UDS we call these "domain component libraries" because they satisfy a use case specific to a domain (*e.g.* a checkout experience, a rich media player). Thus outcome team component libraries will always be domain libraries.

UDS does not set any strict rules about domain component library naming. We recommend avoiding either the `ds` prefix or the `base` name to avoid confusion. Choose a name for your component library which is concise but clearly indicates the domain(s) that your library applies to. This will help other teams discover and hopefully even contribute to your component library.

Using UDS to build components

UDS contains two main extension points for building domain-specific components. Note that these can be used equally to build component libraries or to build components directly within your application.

Building with brand palettes

We have a dedicated page for building with UDS palettes. Palettes are available in platform builds and can be used in building out components, here's an example using a brand palette for web

```
import palette from '@telus-uds/palette-
allium/build/web/palette.js'

const MyParagraphComponent = styled.p`
  padding: ${palette.size.size4};
  color: ${palette.color.purpleTelus};
`
```

Building with UDS Base components

UDS Base components are low-level, unopinionated, atomic components designed to be composed into higher-level components. If you are building in the context of a specific brand, you will likely find that branded versions of the UDS Base components are available from your brand's design system.

We strongly encourage creating your domain components out of the building blocks of UDS Base, it will reduce the amount of effort you need to invest in building components, and ensure you benefit from the work on accessibility, themability, and cross-platform support baked into the UDS Base components.

UDS Base components can be composed to build either React or React Native components, depending on which platforms you are targeting.

Tracking component quality

For UDS Base components we have a component checklist which we use to ensure component quality. We recommend using a similar approach for your component library. You can use the UDS checklist as a starting point and customise to your use case.

Setting up your component library

Using Storybook

We recommend you use Storybook for developing your component library. Storybook is an excellent tool for creating and testing components, and makes it easy to ensure the visibility of your component library within UDS. You can use Storybook for local development and testing of your components.

Storybook has great getting started guides:

- Storybook React for web components,
- Storybook React Native for cross-platform components.

Once your component library is built we encourage using GitHub pages and Storybook deployer to publish your component library to a public url. This enables other teams to see which components you have built and how they work. We recommend setting this up as a CI step in your GitHub repo so that your documentation is always up to date.

Linting and formatting

Linters and formatters scan code and flag or fix common problems. Linters catch common mistakes and hazards, while formatters style code consistently to be easier to read. The following are recommended:

- @telus/telus-standard applies common TELUS linting rules.
- @telus/prettier-config applies common TELUS code formatting conventions.

Accessibility linters can help catch accessibility problems early:

- On Web, TELUS Standard already includes eslint-plugin-jsx-a11y with common TELUS options applied. Where a library contains components that are re-used internally and which should follow the same guidelines as the HTML element it renders, it can be helpful to add a `components` list to settings, like `"jsx-a11y": { "components": { "CustomButton": "button" } }`.
- For cross-browser components, eslint-plugin-react-native-a11y should be added, to quickly catch common app accessibility problems such as

`Pressable` elements lacking a role or label. One change to its default options is recommended: at UDS we disable the requirement to always use `accessibilityHint` alongside `accessibilityLabel`, by setting `"react-native-a11y/has-accessibility-hint": 0`. This is because `accessibilityHint` is an iOS-only feature, not supported at all on Web and simply appended to `accessibilityLabel` on Android, so requiring its use can result in incomplete or duplicated labelling on some platforms.

Testing

Unit tests

For testing individual component features and behaviours, we recommend using Jest with the appropriate Testing Library offering:

- React Testing Library for web-only components,
- React Native Testing Library for cross-platform components.

These libraries have excellent documentation. In particular you can read more about the philosophy of Testing Library.

Accessibility tests

There are tools that can check the output of a component render for common accessibility problems. Note that these are **not an alternative** to real accessibility testing using real assistive technology, but they can save time by catching common mistakes early.

- On web, we recommend Jest Axe
- In iOS or Android context, React Native Accessibility Engine is worth considering; however, note that it is a much newer project than Jest Axe and covers far fewer accessibility rules and scenarios.
- For cross-platform components also intended to run on Web, we recommend including web-only tests set up to run in a web context using Jest Axe

End-to-end (e2e) tests

If your components manage a complex flow, you may wish to investigate automated end-to-end (e2e) testing. These tests use a tool that simulates real

step-by-step user actions on a real UI, like:

- On web, for in-browser testing of React components, consider Cypress
- Cross-platform, for in-app testing of React Native components, consider Detox or Appium. Detox offers more debugging data than Appium, but requires its own modified app builds which may be slow and harder to integrate into workflows.

Visual testing

Unexpected changes to visual appearance can be caught using a visual regression testing tool such as Chromatic, which looks for changes in the appearance of rendered Storybook stories pixel-by-pixel.

Jest Snapshot testing can also be used to catch unexpected changes in the rendered stylesheet styles of both Web and React Native components.

Organising components

We recommend that you avoid a complex component taxonomy when starting out with your component library. The simplest solution to get started is a flat structure where all components are listed at the root. If you find that your component library is growing rapidly and it's getting harder to find components, you may wish to explore the atomic design metaphor for organising your components. Alternatively a simple grouping by function can be effective.

Documentation

Storybook supports rich component documentation using both automated generation from JSDoc comments and MDX. We find that the easiest to maintain option is to put as much of your documentation as possible into JSDoc comments.

If you find that Storybook is too limiting for your component library documentation, it may be useful to explore Docusaurus which is a more fully-featured documentation tool. However Docusaurus is *only* a documentation tool and it will not replace Storybook as a component development environment, so will require more effort and upkeep than using Storybook alone.

Publishing your component library to npm

We recommend distributing your component library under the `@telus-uds` scope in npm. This makes it easier for other teams to discover your component library. You can set up any publishing pipeline you choose for your component library. If you're not sure how to set this up or don't have a strong opinion, you can simply follow the UDS Base setup:

1. Use beachball to enforce change files on PRs for your repo.
2. Set up a CI job which uses beachball to manage versioning and publishing of packages to npm.

For the above you will need to have an npm token in your GitHub actions environment, you can contact the UDS team to get this. Ensure you have permission from your brand to distribute your components via npm.

Getting help

If you have any questions about setting up a component library, you can always reach out. Posting any questions in the community channel enables anyone in the community, including the UDS team, to lend a hand.

Edit this page