Fonts

Understanding the font model and React Native limitations

React Native broadly adheres to the CSS model for fonts, and so we use this as the basis of our font modelling for UDS. Font properties are named to correspond to CSS properties (*e.g.* `fontWeight` is equivalent to `font-weight`). In general you can specify design tokens corresponding to all CSS font properties.

However, React Native does not handle font families in the same way as CSS does for web. On the web it is normal to specify a font via a `font-family`, and then apply a `font-weight` and `font-style` from within that family. You could then create equivalent `font-face` declarations which "group" all of the font weights and styles of that font face into a font family. Unfortunately, React Native does not have particularly good support for font weights within a font family. In order to support font weights effectively on React Native we need to define a new font family for every font face/font weight/font style combination. So whereas in CSS you might write:

```
.my-font-class {
  font-family: "MyFont";
  font-weight: 400;
  font-style: normal;
}
.my-font-class.bold {
  font-weight: 700;
}
.my-font-class.italic {
  font-style: italic;
}
```

The equivalent in a React Native stylesheet would be:

```
const styles = StyleSheet.create({
  myFont: {
    fontFamily: 'MyFont400normal'
```

```
  },
  bold: {
    fontFamily: 'MyFont700normal'
  },
  italic: {
    fontFamily: 'MyFont400italic'
  }
})
```

In UDS Base components this mapping is handled for you, but if you are building your own React or React Native components you need to be aware of how fonts are defined in your application. The below section documents how to load fonts into your application depending on your target platforms and tooling.

Using system fonts in React Native (for ios and android) is possible, but requires non-trivial consideration of font faces and weights. As such **system fonts are currently not supported in UDS**.

Font loading options

Broadly speaking there are 3 variables to consider:

1. **Defining fonts in CSS or JS** - If you are building cross-platform, you will need to import your fonts into your javascript code rather than define them with CSS.
2. **Included in bundle or served from CDN** - Again if you are targeting cross-platform, you will need to bundle your font assets. If you are targeting web-only then you can choose to either include fonts in your bundle, or reference resources on a CDN.
3. **Split or whole families** - For the reasons above, UDS Base components expect font faces to be defined split into individual declarations, *e.g.* `MyFont700normal`. If you're using UDS Base components in your application, or a library which depends on UDS Base, you'll need to make sure these font faces are defined. You can also define fonts as families, where you have a single font name (*e.g.* MyFont) for multiple weights and styles. This is a more typical approach for web projects.

Note that in each case you will be using either a web build or a React Native build of the brand palette. Each build offers several options for importing fonts.

React Native Build

With Expo

In expo applications you can use `expo-fonts`:

```
/* in app.js */
import { useFonts } from 'expo-font'
import AppLoading from 'expo-app-loading' // recommended to
handle loading state
import * as fonts from '@telus-uds/<my-palette-
name>/build/rn/fonts'

export default () => {
  ...
  const [fontsLoaded, fontsError] = useFonts(fonts)
  return fontsLoaded ? <App /> : <AppLoading />
}
```

In a web application (e.g. storybook)

```
import '@telus-uds/<my-palette-name>/build/rn/fonts/fonts-
local.css'
```

Note: this assumes you have an appropriate loader set up to handle css imports (this is true for create-react-app by default).

With either of these setups, fonts are only available in the React Native format for font families (*e.g.* `MyFont700normal`)

Web build

The web build of the brand palette offers 4 different built css files for defining fonts. Include the appropriate file(s) into your build tooling the same way you would any other third party css file.

1. `@telus-uds/<my-palette-name>/build/web/fonts/fonts-local.css` - defines split font faces with local font file imports. Great for use with UDS Base if you want to bundle your fonts into your app.
2. `@telus-uds/<my-palette-name>/build/web/fonts/fonts-cdn.css` - Exactly the same definitions as fonts-local, but references resources on a

CDN.

3. `@telus-uds/<my-palette-name>/build/web/fonts/fonts-local-family.css` - Includes the same fonts as above, but defined in web-friendly font families, also uses local font file imports.
4. `@telus-uds/<my-palette-name>/build/web/fonts/fonts-cdn-family.css` - The font family declarations as above, but referencing CDN resources.

Note you may well wish to include both the split and the family declarations in your app (for example you're using UDS Base components and you have some of your own web components which use the font weights within a font family model). This is fine and won't increase your bundle size or the number of network requests (for local or CDN respectively) because both sets of declarations point at exactly the same resources. However you **should not mix CDN and local CSS**. If you do you'll end up with the fonts in your bundle, and being requested from the CDN.

Palette font assets

As with all design token options, the fonts are defined in the palette. Fonts have the following hierarchy in the palette:

```
{
  // ...

  font: {
    values: {
      Arial: {
        value: {
          100: 'font/arial-100.otf',
          200: 'font/arial-200.otf',
          300: 'font/arial-300.otf'
        }
      },
      TimesNewRoman: {
        value: {
          200: 'font/arial-200.otf',
          500: 'font/arial-500.otf'
        }
      }
    }
```

```
  },

  // ...
}
```

When the palette is built during the release process, the font file assets are given a unique name and uploaded to the CDN. The CDN urls for the assets are defined in the `fonts-cdn*.css` files described in the previous section.

ℹ️ Every release generates unique file names for these CDN assets allowing all versions to co-exist on the CDN.

When referencing fonts in themes, the theme build process un-nests the font weights and family names, creating *virtual* palette tokens for `fontName` and `fontWeight`. These tokens are *virtual* because they are not found the published palette files, but are added virtually to the palette during the theme build. All component tokens and rules referencing fonts must define a valid `fontName` and `fontWeight` token pair. This is validated during theme builds. Given the above palette source, themes can assume the palette has the following tokens:

```
{
  // ...

  fontName: {
    Arial: 'Arial',
    TimesNewRoman: 'TimesNewRoman'
  },
  fontWeight: {
    100: '100',
    200: '200',
    300: '300',
    500: '500'
  },

  // ...
}
```

The theme build process will validate that components define `fontName` and `fontWeight` pairs and that these combinations exist in teh original palette.

Edit this page