**UIContext – React**

**Introduction**

One of the most complex aspects of user interface development is keeping the visual look and feel of the application consistent across all of its different parts and user flows. These areas of concern are considerable, including aspects such as margin and padding, color palette, typography, border radii, shadows, and fonts, which are applied across the different components of the interface.

While it is certainly possible to define all of these aspects and beyond on a case-by-case basis as one builds out an interface, this can create quite a lot of technical debt when attempting to diagnose the highly likely inconsistencies that will arise from this, or if the developers on a project decide that they’d like to refactor aspects of this theming in a uniform way across the user interface.

One of React’s greatest strengths is the ability that it provides to heavily modularize and encapsulate reusable components with everything they need to function reliably and flexibly. This allows developers to move quickly when developing large, complex user interfaces, since they can stay focused on the bigger picture of the overall UX while not getting mired in the minutia of implementing lower level primitives such as buttons, dropdown menus, and all manner of information inputs. However, when it comes to theming an overall application, this is one area where it could be argued that **centralization is warranted**.

If one desires a consistent user interface, then a wise approach is to define the rules that cover look and feel in a central location. Then, by providing this base style guide into a dynamic memory structure at run time, components can refer to it for instructions around how to style themselves. In the event that something changes, such as changing the primary color assigned to a theme, all style instructions in the wider component library that are assigning colors based on this value will change accordingly on the next render pass, causing a consistent and distributed change across the codebase.

**The Static Styleguide**

As mentioned in the introduction to this documentation module, one can establish a centralized styleguide in order to define aspects such as color palette, margin and padding, along with typography/fonts. If the components in the application then refer to that styleguide in order to define their styles, 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 spacing 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 error, warning, and success statuses, disabled states, and the capacity to change fluidly from dark to light themes.
* 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 set the geometry of different components (e.g. orthogonal, rounded, or round).
* 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).

You can find more information on each part of this structure in the Styleguide documentation. [ Provide link to styleguide documentation ]

**Implementing the UI Context Provider**

React’s unidirectional data flow is great, letting you pass in information to components when you’re trying to build them. However, when dealing with a centralized structure like a theme, it becomes unwieldly to pass down this information all the way through a component hierarchy into the ones you’re trying to style. For this situation, React created the [Context API](https://reactjs.org/docs/context.html). It allows one to create centralized information structures just like the user interface context system being described here, which sit at the top of component trees. Then through a variety of mechanisms, component source code can access this information without it having to be passed all the way down from the top level.

Equipped with a static styleguide, the idea with the UI context provider is to make a copy of it available to components as they need it, along with other runtime decisions such as setting primary, secondary, tertiary, and quaternary colors, or switching between dark and light themes. By using this API and setting up a custom <ThemeProvider /> component, we can wrap it around the entire application and make use of it throughout the interface.

**Putting The Styleguide into State Along with Other Dynamic Information**

Beginning the overall process of creating this UI context provider, the first step is to use the Context API in order to create an instance of the system created by it. Following this, one creates a class which will use this context in order to render the theme provider component. At this point, one can define the state of that component, where statuses around being in dark or light theme can be defined, colors can be assigned from the palette to be primary, secondary, tertiary, and quaternary, and the rest of the styleguide can be spread in. Up to this point, the code for such as setup will look something along the lines of:

import \* as React from ‘react’;

import {styleGuide} from ‘../../styleGuide/styleGuide’;

import { generateTypographyTable } from ‘../../styleguide/typography’;

enum themes {

LIGHT = “light”,

DARK = “dark”,

}

export type UIContext = {

navBarVisible: boolean;

backdropVisible: boolean;

toggleTheme(): void;

toggleBackDrop(): void;

themeInfo: ThemeInfo;

}

export type ThemeInfo = {

theme: themes;

palette: any;

fonts: any;

typography: any;

distance: any;

stackingIndexes: any;

borderRadii: any;

shadows: any;

gradients: any;

transitions: any;

styles: any;

}

const initContextValue: UIContext = {

navBarVisible: true,

backdropVisible: false,

toggleTheme: () => {},

toggleBackdrop: () => {},

themeInfo: {

theme: themes.LIGHT,

palette: {},

fonts: {},

typography: {},

distance: {},

stackingIndexes: {},

borderRadii: {},

shadows: {},

gradients: {},

transitions: {},

styles: {},

}

}

export const UIContext = React.createContext(initContextValue);

export default class UI\_InfoProvider extends React.Component {

static instance: any

state = {

navBarVisible: true,

backdropVisible: false,

themeInfo: {

theme: themes.LIGHT,

palette: {...styleGuide.palette },

fonts: {...styleGuide.fonts },

typography: generageTypographyTable(styleGuide.fonts,

styleguide.palette),

distance: {...styleGuide.distance },

stackingIndexes: {...styleGuide.stackingIndexes },

borderRadii: { ...styleGuide.borderRadii },

shadows: { ...styleGuide.shadows },

gradients: { ...styleGuide.gradients },

transitions: { ...styleGuide.transitions },

styles: {...styleGuide.styles },

}

};

}

**Defining getInstance()**

In order for this theme system to be as dynamic and flexible as it can be, it is important that, at certain points in the user interface’s lifecycle, components can get a reference to the theme provider’s instance, in order to assert control over actions such as changing the color scheme or moving from light to dark theme.

This is done, firstly, by defining a static property on the class called ‘instance’, and initialize it to null. Then, in the componentDidMount() lifecycle method, one can throw an error if this instance property is not null, and if not, assign it’s value to ‘this’, the reference to the instantiation of the component class. Finally, one can define a static method on the class called getInstance(), which throws an error if this property has not been defined yet, and otherwise returns that instance. After these additions, the theme provider source file looks like.

/\* Imports, enums, types and init context definition, context instantiation \*/

export default class UI\_InfoProvider extends React.Component {

/\* Initial types and state definition \*/

static instance = null;

componentDidMount() {

if (ThemeProvider.instance !== null) {

throw new Error(

“Only one ThemeProvider instance can

be mounted at a time.”

);

}

ThemeProvider.instance = this;

};

static getInstance() {

if (!ThemeProvider.instance) {

throw new Error(“ThemeProvider has not been mounted yet.”);

}

return ThemeProvider.instance;

}

}

**Defining render()**

The render function of this UI context provider returns a Provider, which is a static property made available by the Context API. The current theme is passed in from state, and it then wraps around the entire application’s component tree, making the theme available to all components that need it in order to properly render. With this render function defined, the theme context component code looks like:

/\* Imports, enums, types and init context definition, context instantiation \*/

export default class UI\_InfoProvider extends React.Component {

/\* Initial state definition \*/

/\* instance property, componentDidMount, and getInstance \*/

render() {

const {children} = this.props;

const uiInfo = { ...this.state };

return (

<UIContext.Provider value={{

...uiInfo,

toggleTheme: this.toggleTheme,

toggleBackdrop: this.toggleBackdrop

}}>

{children}

</UIContext.Provider>

);

};

}

**Using the UI\_InfoProvider in Other Components**

With the UIContext wrapping the application, components further down in the hierarchy are able to programmatically ask for this data structure in order to style themselves and be aware of contextual UI aspects such as the backdrop being set to visible. Through the use of the useContext() react hook, components can access this data structure as they build themselves like so (using the <Button /> component as an example):

import React, { useContext } from ‘react’;

import { UIContext } from ‘../UI\_InfoProvider/UI\_InfoProvider;

const Button: React.FC<IButtonProps> = (props) => {

const uiInfo = useContext(UIContext);

// - Component source code continues from here.

// - Component is able to use the uiInfo data structure for

// its own purposes.

}