**General Codebase Guide**

This document serves as a top level overview of this component library/catalyst codebase so far. It seeks to explain how it’s organized overall and digs into some of the design choices made with its different parts.

**Base Style Generation**

The TL;DR on how styles are being generated in components is that I am using a software library called Aphrodite, which takes styles defined with TypeScript and parses them into CSS to be used in the components. Defining styles using TypeScript ends up being a more flexible solution, because those styles can be generated dynamically according to props, state, and other context. You can find more information on this process in the Base Style Generation documentation which is included in the InformationArchitecture directory of this codebase.

**Component Calibration**

The style generation explained above results in a variety of CSS classes which can be applied to different markup sections of a component in order to obtain the proper styling at render-time. However, in many cases, these styles need to be chained together according to a context decided by state, props, and other information (e.g. a button that receives a ‘rounded’ prop for its geometry will need to apply the base button cradle style along with another one that will create rounded corners). The component calibration method is built for certain complex components like this. According to information passed in, it determines conditional styles, and then chains them together. The method then returns those chains of CSS classes which are assigned to different parts of the component markup. You can find more information on this function in the Component Calibration documentation which is included in the InformationArchitecture directory of this codebase.

**UIContext**

At a very high level of detail, this React codebase makes use of the Context API in order to centralize information regarding the styling of the application and its current state. By wrapping the component hierarchy with this context provider, other components further down the tree can utilize the useContext() React Hook in order to access this centralized information. This centralizes key styling information in one place, allowing for easy codebase-wide modifications at a future time. The UIContext directory in the InformationArchitecture directory of this codebase contains documentation that explains the construction and operation of this data structure. It also contains documentation on the static styleguide which goes into the UIContext structure, and documentation explaining how the codebase can be extended in order to accommodate a dark theme and light theme.

**Components**

This codebase is very much a reflection of my individual efforts to learn how to build relatively competent versions of common UI components in a larger-scale effort to improve my front end development skills. I’m super open to using any pre-built components where any skill gaps exist, but I’ve been working on developing my own versions in order to learn how to build them and to integrate them comprehensively with how I’ve been building my larger styling systems. Below, I provide a summary of the types of components in the codebase so far:

**AvatarImage**

General purpose component that is meant to display either a profile image, or as a fallback, a person’s initials.

**Backdrop**

This component is included at the top of the component hierarchy and provides a backdrop that sits on top of content and can provide a color overlay at a certain opacity. It is controlled through the UI\_Context structure and is meant to be activated later on when the codebase includes components like Modals when doing things like creating/modifying/deleting resources.

**Button**

General purpose button component that can accommodate a number of different contexts. A constant work in progress as I discover new ways to make button components even more sophisticated. One future improvement I’d like to add is to introduce a loadingComponent prop which would allow the developer using the component to provide a component that is to display when clicking the button triggers a loading state (AKA confirming that you’d like to submit payment on a purchase).

**ClickableListItem**

I built this component as a shortcut to generating list items that have potentially an avatar image, some explanatory text, and are clickable to trigger some action regarding the list item.

**Dropdown**

General purpose dropdown component that is still very much a work in progress. I recently refactored this to TypeScript from its initial form as a JavaScript component. I haven’t had too much of a chance to test it out, but I have many cases in mind where it will be useful later on.

One improvement I have in mind for this component as well is to be able to provide a dropdown component as a prop. In some cases (e.g. choosing the filter or sort buttons on the EmbeddableResourceBrowser), I’d like a small panel to drop down with additional options on it, rather than a list of options to choose from. At the moment, the dropdown component only takes a list of options.

**EmbeddableResourceBrowser**

This is meant as a kind of first crack at creating a Google Drive style UX. The component takes a resource (as defined in the Types for the codebase), and is then able to navigate through the resulting directory structure allowing a user to access the resources they need.

I manage the internal state of the component using the useReducer() hook, where I maintain an internal history structure for the component that allows a user to navigate forwards and backwards.

So far, the component allows the user to switch between a grid and a list view. Future improvements include enabling filtering and sorting of the resource contents of the browser, along with a Breadcrumbs component along the top that gives the user an impression of their current position in the resource tree.

**Fallbacks**

Logical grouping folder for any fallback components that are used throughout the application. One example that I have in there currently is the RouteNotFoundFallback component, which is displayed when a user tries to access a route that is not registered with the codebase.

**LayoutUtilities**

Logical grouping folder for any utility components that have to do with overall layout of other components. So far, I have a FillUnderNavBarCradle component which wraps around any content and makes sure that it takes up as much space as is available underneath the navigation bar. I also have a spacer component which allows a developer to create some arbitrary amount of horizontal or vertical distance between other elements.

**Lists**

Logical grouping folder for different generic list components. So far, I have a ClickableItemList component in there that I reuse in different scenarios.

**Loaders**

Logical grouping folder for different loader components. I have a bunch of works in progress in there so far. The two loaders that do work very well are the HorizontalBubblesLoader and the CircleLoader component. These can be included as placeholder content in other components while they are hanging in a loading state.

**NavBar**

Component for the application’s navbar. In the future, as an authentication/authorization layer is built on the system, I’d like to include a login button when not logged in in the top right, and then a dropdown component that has the user’s avatar image in the top right when they’re logged in.

**Routing Module**

This is a component that I use in order to encapsulate all the routing needs of the application into once place.

**Tag**

General purpose component used to display some piece of information or related pieces of information together in a visually distinguished manner. The tag component can be static, displaying some text content as is, or it can be toggleable as well (e.g. when used as part of a series of options that someone can choose from). I use these components so far in the EmbeddableResourceBrowser in order to demonstrate what types of data a resource represents (e.g. documents, images, videos, audio, etc…)

**UI\_InfoProvider**

As explained further in the UIContext documentation, this is the place in the component library that I keep this component. It is used to load in a static styleguide representing the design language of the application, and to keep track of dynamic information such as whether or not the backdrop is currently engaged.

**VisualUtilities**

Logical grouping folder for small components meant to act as visual utilities within larger ones. So far, this directory includes:

* A variety of icon presets, where I’ve hardcoded certain choices from the React-Icons library to represent different entities or actions within the system (e.g. choosing a specific set of chevrons and arrows to use consistently in the codebase).
* An ImageWrapper component.
* An indicator component meant to be used in the construction of things like checkboxes, or to indicate things like unread notifications.
* A LineSegment component that can be used to generate vertical and horizontal lines of different sizes and colors. These are most often used as dividers between content in different components in the application.
* A LinkWrapper component that ensures that certain cybersecurity measures are included in the link markup.

**WrappedApp**

This is a utility component where I abstract the top level work of wrapping the main <App/> component in various infrastructure components. At the moment, the app component is wrapped by the router and the UIContext provider. In the future, I envision that it will also be wrapped by an AuthContext component and the ApolloClient as the front end is connected to a GraphQL API.

**Pages**

This is meant to be a part of the codebase for React components that take up an entire page and that possibly act as the “smart” top-level components where data operations occur. At the moment, the application is just using static JSON data, but I’m trying to create this separation early in order to smooth the transition as the application goes full-stack.

**Ecosystems**

While the codebase is not at this scale yet, as it grows, there will be families of components that can be linked around certain entities in the system (e.g. all the components involved with creating, updating, or deleting a resource). As the codebase reaches this complexity level, I envision this folder as a place to logically group those types of components together.

**StyleGuide**

This is the directory in which I build the static styleguide for the application, along with sub-files that contain parts of the styleguide (e.g. the distance table and color palette). More information on this styleguide can be found in the UIContext part of the documentation in the InformationArchitecture directory of the codebase.

**Types**

This is a general place where I’ve been keeping codebase-wide relevance types. As the system goes full stack, I will probably autogenerate these types from my GraphQL code using some of Apollo’s codegen tools. Writing them out manually is fine for now though.

**Helpers**

This part of the codebase is meant to house utility functions that are used in multiple places. So far, the helpers do things like mapping resource types to resource type icons, merging an entity’s communications and socials media information into one data structure, and finding a resource by ID recursively in a resource tree. This directory is meant to grow as more helpers are added in the future.

**Constants**

In an effort to make the code as modular and maintainable as possible, I have endeavored to generate string constants for any phrase or word that is used multiple times in the codebase. So far, I have constants for color schemes, geometry constants, different online networks, resource types, URL routes, along with general theming and user interface constants. This too is a directory that is meant to grow as more words and phrases are deemed to be refactorable into string constants.