1
Frontend

1.1 Why make a frontend?

There's a certain challenge to making your work accessible. Building and deploying a web application necessitates a different skill set than that required by the implementation of algorithms. There are layers added and layers complexified in shifting from a command line based program to a UI. A common practice of reading/writing from local files translates to managing some sort of data store, be it a proper database or some sort of browser storage like cookies, local storage, IndexedDB, or whatnot. We made a portal to accompany our APIs in order to have our work accessible to whoever may wish to use it regardless of their background in programming.

1.1.1 Overview of our frontend

Our webapp is a Single Page Application (SPA) built using VueJS. Vue is one of big players in terms of frontend frameworks along with Facebook's ReactJS and Google's Angular, and is reknowned for it's simplicity, scalability, and speed. We used the Vue CLI tool for development and building, although there is alternatively a library for the view layer via jsdeliver. As such, we have largely kept to the structure of the scaffolding, keeping top level directories for assets, components, composables, and views. Composables were added later with the adoption of Vue's new composition API—in RFC (request for comments) at the time of use—which gives a

fundamentally different way to interact with the Vue instance, giving a hook-oriented/function-based approach as opposed to the previous options API (an approach based off of certain properties in an object). Outside of Vue, we have several dependencies for the odd piece here and there, the largest part of the workflow would be tailwindcss: a utility-based CSS framework. Tailwind ultimately gives a wide array of low-level utility classes that make for more concise CSS, and more flexible in-template stylings.

Our project is a SPA, although it has many views. Traditional websites may have several files that are served per URL, the rise of web applications led to a need for routers that mount and unmount components based on `window.location`. The Vue-Router takes an array of objects that specify path and may specify other aspects, including the component to mount, name, redirect location, and props. We strictly specified path, name, and component for our routes, and we left the router in hash mode to save on some server config for associated with history mode; we have routes for the home page, the thesaurus, the models, writing, documentation, and presentations. As per the likeness in characteristics, we will use "routes" and "pages" interchangably.

The home page simply serves as a hub to link to all pages as well as the repository on GitHub, as the main navbar does not span all routes. The thesaurus page gives an input box to query the thesaurus and senselevel collections, and displays synonyms and antonyms by part of speech as well as sense associations by word-sense including synonyms shared between the sense words and the queried word. The model page offers interaction with each of our models; the model is selected through a dropdown menu, and the associated controls are displayed (a textbox, various inputs such as sliders and toggles for targetting the endpoint's parameters, and a submit button); each model has a unique results section. The writing section is very simply an online version of this paper rendered to HTML and injected. The documentation page is a custom renderer for our Postman collection. Postman is the HTTP client we used for development to organize requests and inject variables for things like staging; it also allows for exporting your collection as JSON, which we are using to inform our documentation page. The presentation route holds our HTML presentation rendering which allows for ease of access and navigation.

1.1.2 Configuration

There's not too much configuration to be done here as per the fairly minimal frontend and the niceties that Node has to offer. There are a few config files to edit to get tailwind integrated and to add our customizations to it, and there is some tinkering with webpack to add a loader to import `.txt` files as strings. We set `NODE_ENV` within our start scripts to be able to target different URI's for our APIs.

1.2 Creating our Component Library

Various frameworks take different approaches to organizing the HTML, CSS, and Javascript that make up each component. Some frameworks, like Angular, tend to keep separate files for each piece of the component, whereas frameworks like Vue opt for single file components (SFC). This is simply an organizational difference, and each approach has pros and cons. We need various components to build our frontend, from basic UI components like text inputs and sliders to page components that are associated with each route. As per the options API, each component may take a set of props, which is specifies data to be passed to them from the parent; this follows Vue's one-way data flow principle—should data have to be passed to a parent, we do so by emitting events and attaching a listener to the parent.

1.2.1 Different Kinds of Components

There are multiple ways to write components within Vue. There are different ways to target the template; one of which is to write all of the markup in HTML with directives and bindings like `v-if`, `v-for`, `v-bind`, `v-on`, and other way is through the render function. The render function is what is used underneath the hood; Vue compiles the markup templates to a render function that creates a VNode (virtual node), as Vue is based on the virtual DOM. The render function taking some parameter to specify tag or component options, an object that translates to element attributes, and a list of children. There is also a distinction between functional and full-featured components; functional components do not manage their own state, watch state, nor have lifecycle

hooks, and thus do not need the same reactivity that full-featured components require. Below is an example of the same UI represented as a render function and as a template for the sake of comparison:

```
render(h, {props, listeners}) {
  const hasResults = (obj) => {
    for(let key of Object.keys(obj)){
      if(obj[key].length > 0) return true;
    }
    return false
  }
  let entry = props.entry;
  return h("div", { staticClass: "my-12" }, [
    h("div", {
        staticClass: "thesaurus--results-box",
      }, [
      Object.keys(entry).map(key => {
        return hasResults(entry[key]) ? h("div", {}, [
          h("h2", {
            staticClass: "thesaurus--category",
            domProps: { "innerHTML": key },
          }),
          h("div", {
            staticClass: "thesaurus--pos__wrapper"
          }, Object.keys(entry[key]).map(pos => {
            if(entry[key][pos].length > 0) {
              return h("div", { staticClass: "mx-4 md:mx-8"}, [
                h("h3", { staticClass: "thesaurus--pos" }, pos),
                h("ul", { "class": "thesaurus--entry__wrapper" },
                  entry[key][pos].map(word => {
                    return h("li", {
                      attrs: { tabindex: 0 },
                      staticClass: "thesaurus--entry",
                      domProps: { "innerHTML": word },
                      on: {
                        keydown: (e) => {
                          if(e.key === 'Enter' || e.key === ' ') {
                            const emit_event = listeners.event_from_child;
                            emit_event(word);
                          }
                        },
                        click: () => {
                          const emit_event = listeners.event_from_child;
                          emit_event(word);
                        }
                      }
                    })
                  })
                )
              ])
            }
          }))
        ]) : null
      })
      ]
    )
  ])
}
```

```
<template functional>
```

```
<div class="thesaurus--results-box">
  <div
    v-for="(key, i) in Object.keys(props.entry)"
    :key="[key, i].join('-')"
  >
    <div v-if="Object.entries(props.entry[key]).map(([k, v]) => v.length > 0).reduce((a,
        c) => (a || c))">
      <h2 class="thesaurus--category">
        {{ key }}
      </h2>
      <div class="thesaurus--pos__wrapper">
        <div
          v-for="(pos, j) in Object.keys(props.entry[key])"
          :key="[pos, props.entry.key, j].join('-')"
        >
          <div
            v-if="props.entry[key][pos].length > 0"
            class="mx-4 md:mx-8"
          >
            <h3 class="thesaurus--pos">
              {{ pos }}
            </h3>
            <ul class="thesaurus--entry__wrapper">
              <li
                class="thesaurus--entry"
                v-for="(word, k) in props.entry[key][pos]"
                :key="[pos, props.entry.key, word, k].join('-')"
                v-on:keydown.enter="() => listeners.event_from_child(word)"
                v-on:keydown.space="() => listeners.event_from_child(word)"
                v-on:click="() => listeners.event_from_child(word)"
                tabindex="0"
              >
                {{word}}
              </li>
            </ul>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
</template>
```

Ultimately the render function provides greater flexibility with all of Javascript at your disposal rather than relying on the tools provided by Vue through directives. Writing raw render functions is a process somewhere between writing JSX (which render functions support as an alternative through a Babel plugin), and writing vanilla Javascript to manipulate the DOM.

1.2.2 Our Components

We have all basic UI components within a directory with the following components: `InfoHover`, `Loading`, `ProgressBar` `Slider`, `TextInput`, `Toggle`, `WordList`. `InfoHover` provides an info icon with a child conditionally rendered on hover or on click via a `slot`; we have a prop for `size`

that controls the width and height of the icon-button, as `size` can only take certain values, we add a validator to ensure that valid values are used. `Loading` is a simple loading spinner that is displayed while a request is being processed, we do this with partial border coloring and infinite-duration CSS animations—this is a component that makes a clear case to be a functional component, as it doesn't even need it's own Javascript to function. `ProgressBar` gives a rectangle colored proportionally to the `value` prop and with a color mapped to by the `color` prop; `color` also has a validator attached. `Slider` wraps the default `type=‘‘range’’` input, taking care of data binding, styling, and providing a slot for a label; there are props for `min`, `max`, `step`, and `value`. `TextInput` is similar to `Slider` in that it wraps the default `type=‘‘range | X’’` input, handles data binding and styling, and has props for `value` and `error` which is a string that affects style and adds a message to the DOM. `Toggle` is an wrapper for the default `type=‘‘checkbox’’` input that overhauls the styling in order to target the feel of a switch rather than a checkbox; this only takes a prop for `value`. `WordList` eases the process of rendering each word from an array, simply looping over the items in it's `value` prop and displaying them in order. These are all basic UI building blocks that are used throughout the application, and largely serve to bind date or display pieces of information passed to them.

There are a few other components across the application that don't quite fit within the basic UI group nor the documentation group. These include `ScoreBars`, `GroupScores`, `NetScores`, `Navbar`, `SenseLevelResult`, `ThesaurusResult`, and `ThesaurusEntry`. `ScoreBars` displays a group of labelled `ProgressBar`s or a label signifying that each value was zero; there are props for `name`, `size`, and `value`. `GroupScores` renders a series of `ScoreBars` for the output of each topic in the LDA model, and only takes a `value` prop. `NetScores` operates similarly to groupscores, but ignores trying to render some properties that do not exist. `Navbar` only displays a group of `router-link`s. `SenseLevelResult` handles the senselevel rendering; it must make a request to our API for the current word and parse the response, finding common synonyms, and organizing by sense. `ThesaurusResult` takes the response from the thesaurus endpoint and renders the result by synonym/antonym and part of speech; this was a place we wrote our own render

function. `ThesaurusEntry` is a layout component that gives a text input for the current word and renders the `SenseLevelResult` and `ThesaurusResult` components upon submission.

1.2.3 Creating Our Documentation Renderer

A large part of using components is to break down monolithic markup into readable/maintainable chunks under the assumption that fifty small files with a narrow focus are easier to grasp than a handful of huge files responsible for a wide range of behavior. The other is to cut down on the copy-pasting of similar markup that will be used multiple times. Both usages appear within our documentation renderer.

We have a layout component, `DocRenderer`, that receives an object representing the JSON from the documentation and sends appropriate data to its children based off of the current selected folder. `DocRenderer` provides a navigation bar with a link to `/home`, buttons for each folder of requests, and buttons for each request within the selected folder. We make use of gsap's `ScrollToPlugin` to scroll to the correct page offset once a user selects a request. We pass the selected folder to our `Folder` component, which displays a header with a name and description, and mounts a series of `DocRequest` components for each request within the folder; this uses markdown-it as a markdown rendering engine for the description. `DocRequest` is a component responsible for rendering a single request's information; it displays name, HTTP method, url, description, request headers, request body, and response, the latter three being abstracted to components. `Headers` renders a table of accepted headers for the current request; `Body` and `Response` render the request body and response as preformatted JSON—prettified by re-parsing and stringifying—with the additional status code section for the `Response` to render. Put together with example responses and targetted requests through Postman, we arrive at an intuitive documentation portal for our API in place of a 3.3M JSON document.

1.3 Consuming the composition API

Analagous to the level of abstraction achievable through separate components comes that of the Vue Composition API. This serves not necessarily to replace the options API, but to compliment

it, and to give greater access to the underlying APIs of Vue. It's usage within this project is limited to that of `useModel`, as it has only come out over the course of this project. While it is limited in its use across our frontend, `useModel` manages to be used by each of our four model components. `useModel` serves as a testiment to the use-case of the composition API as we are able to abstract almost all logic from the model components. Rather than binding reactive data via entries within the `data` property, we are given `ref` and `reactive` which can be used largely interchangeably although `ref` is moreso for Javascript primitives and `reactive` is for objects (that is `ref` calls `reactive` when given an object and `reactive` cannot accept primitives), and values are exposed to the template by returning from the `setup` function.

As such, each model component has various reactive state corresponding to the inputs—a `ref` for the text input, the probability of changing a word, whether to ignore stopwords, etc—and state returned by the model targeted within `useModel`. Each model hook exported by `useModel` calls the `useModel` function, passing the appropriate endpoint; this is a higher order function wrapping `responseState`—a `reactive` object with properties relavent to rendering an HTTP response—and a function called `postData` that wraps `window.fetch` to deal with the repeated logic across requests for injecting endpoints and binding data to `responseState`. This allows for a large reduction in repeated logic across the four model components, in that each component only needs to call `postData` with the request body.

1.4 Making things pretty

Much of the frontend is complex purely for the reason of aesthetics. Out of the box, without any CSS, we get a functional, responsive website; it will be black text left justified on a white background, and it will look as though the text content of a website was copied and pasted into a Word document. This is to say, it will work, although it may be significantly harder and less enjoyable to use. User interfaces and user experience (UI/UX) play a large role in web development with UX informing how a site should work and UI informing how it should work.

We have already seen a decent amount of this with our base UI components, that often served to override the default HTML inputs. Among the simpler examples would be our `TextInput` component; here we specify that the text should align left and that the element should take the full width of the parent on the wrapper element, and we pass styles to the `input` element. We apply the classes `[w-full px-2 font-normal rounded border-2 border-primary-50 text-primary-10 appearance-none]` which are generated by tailwind and translates to:

```
{
  width: 100%;
  padding-left: 0.5rem;
  padding-right: 0.5rem;
  font-weight: 400;
  border-radius: 0.25rem;
  border: 2px solid var(--primary-50);
  color: var(--primary-10);
  appearance: none;
}
```

We also add some variants to the `input` element. We add focus styles, placeholder styles, and dark mode variants. The associated error message element features orange text, a heavier fontweight, a smaller text size, and a different text alignment to set it off from the styles of the `input` and to signify that something has gone wrong.

Often, the main pieces of markup are the layout containers and the content elements. The layout container is responsible for the positioning of its children in a general sense: we may have a `display: flex; flex-direction: row` container that uses `justify-content: between; align-items: baseline` which places its children evenly across its full extent of the x-axis and are set on the y-axis such that their baselines are even; a layout container with `display: grid; grid-gap: 1rem` would display its children in a single column with 1rem of space between each item. Content elements tend to set properties like the typography, margin, etc that only affect the element itself.

CSS is a large, tricky, beautiful language. Despite the power behind it, there's little to discuss without getting far into specifics regarding my design choices. Rather than get into why some places have 18pt font and others have 14pt and other similar decisions, I will point to w3 schools

to learn more on CSS, and leave a line of Javascript that you can put in your browser's developer console to disable the CSS stylesheets for the page you are on (only until you reload):

```
for (const s of document.styleSheets) s.disabled = true
```

1.5 Making things accessible

Where beautification may not be the most important aspect to discuss, accessibility absolutely warrants a nod. Accessibility is something often overlooked in websites, especially among junior developers; as a junior developer myself, I am not an expert, but there are several easy steps you can take to improve the accessibility of your site.

In creating something navigatable by keyboard, the user will be traversing between elements largely through arrow keys and the tab key. When an element has focus, there is some sort of ring around it, for example `outline: 5px auto -webkit-focus-ring-color` this stands out to signify which element has focus. This focus ring is not the prettiest in terms of UI, despite being functional, and so as to avoid the focus ring rising when things like buttons are clicked (or other elements that receive focus without dismissing on click), it is unfortunately common to see many elements with `selector:focus { outline: none }` which leaves keyboard users unable to properly determine what element has focus. In order to achieve a pretty UI while considering accessibility, we create specific focus styles for relevant focusable elements; this can come in the form of tailwind's `shadow-outline` that applies a solid blue box-shadow around the element (as box-shadow respects border-radius, but outline does not), an underline (although this is rarely sufficient), or changes in background/text/border color.

There are several other key aspects to accessibility; one of which is the use of screen readers. This is something harder to target if you aren't specifically trying to, as it often has no impact for those that are seeing, and many people either do not have a screen reader or are not proficient in using it. That being said, there are several easy pieces that can be addressed to improve the experience of using your site via screen reader: using semantic HTML where applicable rather than creating and styling generic elements (`div`s and `span`s) gives more insight as to what

the element is; using appropriate headings when applicable helps with navigation; and proper labels for inputs gives insight into the functionality of controls. The latter piece, with labels, is something noteworthy as it is still common to see inputs use placeholder text in place of labels; this may sound reasonable, but screen readers need a label to read as a prompt once there is focus. There is a practice, which is commendable, of providing text nodes that are only accessible via screen reader (essentially creating elements that are present but invisible), although providing a screen reader only label may not even be enough! Chrome's auto translation has skipped over translating attributes (including placeholder), which leaves an untranslated label; even beyond this there are accessibility issues at stake with recall such that a user may lose track of the label for an input if it is a placeholder that disappears once it recieves input; there are issues with color contrast ratios of placeholder text, and so on.

Speaking of contrast ratios, this is one of the most glaring problems for visually impaired users. There is a formula from the Web Content Accessibility Guidelines (WCAG) that determines contrast score and ratio, although there are certain aspects of contrast like hue difference that are largely ignored, as well as CSS properties that contribute to contrast like shadows. Regardless of its flaws, it serves as a solid heuristic for accessible typography. Some browsers are able to provide contrast scores from within the developer tools, which helps a developer to determine if their site is meeting minimum contrast ratios.

In terms of browser support for assisting in targetting accessibility issues, there are certain developer features browsers may have to audit a webpage's accessibility features. Chrome does this via lighthouse, which again has politics attached in that it is run by Google who targets certain things that make it easier to serve ads, etc. Disclaimer aside, it is a powerful, helpful tool that can automate some (not all) of accessibility checking, checking things including:

```
- [aria-*] attributes match their roles
- [aria-*] attributes have valid values
- [aria-*] attributes are valid and not misspelled
- Buttons have an accessible name
- The page contains a heading, skip link, or landmark region
- Background and foreground colors have a sufficient contrast ratio
- Document has a <title> element
- [id] attributes on the page are unique
- <html> element has a [lang] attribute
- <html> element has a valid value for its [lang] attribute
```

```
- Form elements have associated labels
- Links have a discernible name
- Lists contain only <li> elements and script supporting elements (<script> and - <
    template>).
- List items (<li>) are contained within <ul> or <ol> parent elements
- [user-scalable="no"] is not used in the <meta name="viewport"> element and the - [
    maximum-scale] attribute is not less than 5.
- No element has a [tabindex] value greater than 0
```

Our frontend is not perfect on every page even by the lighthouse audits—it reports that the documentation page is missing a title page, and there is insufficient contrast on the request method badge, leading to a score of 91/100. It is a per-page audit, and gives clear feedback on what must be done to address the concerns, which is a wonderful nicety in terms of developer experience, and helps to keep people aware of serious concerns with their websites.