Technical Design

Tech Stack

- Have to use Electron since it is based on Chrome and the live theme preview only works on chrome
- React for the UI
  - easier to work with
  - could explore non framework solution in the future
  - The idea is to use React as a view library, not as the entire application
- Typescript (duh)
- Vite vs Webpack
  - as fast and cool as Vite is, Webpack gives better DX (in terms of configurations) and feels like less “hacking” has to be done to get everything working
  - Can use MonacoPlugin for Webpack
  - Use Electron Forge
- MobX for state management
  - different experience, has good performance
  - observable pattern
  - Helps decouple model/business logic from React
- Use react-virtuoso for the virtual list/file explorer: **petyosi / react-virtuoso**
- Styling
  - SASS modules
    - Sass gives CSS superpowers
    - using modules makes using classnames a lot easier
    - for the file tree with dynamic margin I can just use the `style` attribute

- Inspired by MVC design pattern
- `child_process` vs `execa` module for Theme Preview implementation
  - `execa` allows for better shebang support and window
  - `execa` also force kills the process if it remains/killing it the first time does not work

Notification types

- Error
- Info
- Success
- warning

OS 2.0 Themes ONLY

- would love to expand to Hydrogen headless storefronts
- for custom themes (like this one), should give users the options to provide their own command that they want to use to start the process

In an ideal world, we would have the user link their Shopify Partner account and any test store(s) they want to develop themes on. Unfortunately, they only way to authenticate accounts is when a CLI command is triggered (and the web browser is opened to sign the user in)

- Will probably just use theme access passwords
- this will require devs to have this password saved with them

Directory Structure

- `src`

    - `main`
        - `preload`
        - `services`
        - `models`
        - `utils`
    - `renderer`
        - `components`
        - `models`
        - `context`
        - `hooks`
        - `ui`
    - `common`

Theme/File Paths

- As of right now, all theme/folder paths will be absolute as they are gotten from the file dialog (gives only absolute paths)
- the recent document feature also use these paths so they too will also be absolute
- This means that for the time being, paths do not need to be normalized and should be used as is

Menu Bar

- Helium
    - About Helium
    - Settings
    - Hide Helium
    - Show All

    - Quit
- File
    - New File
    - New Window
    - Open Theme
    - Save
    - Save As
    - Auto Save?
    - Close Editor
    - Close Window
- Edit
    - Undo
    - Redo
    - Cut
    - Copy
    - Paste
    - Select All
- View
    - Toggle Full Screen
    - Actual Size
    - Zoom In
    - Zoom Out
- Window
    - Tile Window to Left of Screen
    - Tile Window to Right of Screen
    - move

    - Minimize
    - Close
    - Bring All to Front
- Theme
    - Push Theme
    - Pull Theme
    - Publish Theme
    - Start/Stop Theme Preview
    - Connect/Disconnect Shop
    - Open Shop
- Help
    - Toggle DevTools

`helium` API

All API calls act in the context of the window that it is being called from

- `app` → all API's related to the actual desktop application
    - `minimizeWindow()`
    - `getWindowState`
        - `isMinimized`
        - `isMaximized`
        - `isFocused`
    - `maximizeWindow()`
    - `closeWindow()`
    - `openFolderDialog() -> Promise<string>`
    - `onReady((initalState) => {})`

```
setWindowTitle(title: string)
// doesn't just use the document.title???
```

- `fs` → all APIs related to working with the file system
  - `readFile()`
  - `writeFile()`
  - `deleteFile()`
  - `createDireactory()`
  - `readDirectory()`
  - `pathExists()`
  - `onDirectoryChange()`
  - `rename(oldPath, newPath)`
  - `attachDirectoryWatcher(directory)`
  - `removeDirectoryWatcher(directory)`
- `shopify`
  - `openLocalTheme(path)`
  - `openRemoteTheme(gitUrl)`
    - `openLocalTheme()` and `openRemoteTheme()` might be combined into one method `openTheme({ pathOrUrl })`, and will handle each appropriately
    - Opening a local theme (for the first time) won't (and shouldn't) be linked with a store
    - The dev can link a store to the theme (or multiple other stores) during their development session
    - What is stored should be the **last store** that was linked with the theme
    - When the theme is opened again, it should be automatically linked to the last used store

        - Instead of doing this, it probably will be best if the functionality of pulling a remote theme and opening it should be separate
    - `getThemes()`
    - `startThemePreview()`
    - `stopThemePreview()`
    - `pullTheme(id)`
    - `pushTheme(id)`
    - `getStores()`
    - `connectedStoreUrl`
    - `connectStore({ storeUrl, password })`
        - It would be nice to be able to use/switch between multiple stores
    - `connectedStorePrefix`
    - `onStoreChange()`
    - `onPreviewStateChange()`
    - `onThemeInfoChange`
    - `previewState`
        -
- `settings`
    - `TODO`
- `utils`
    - `getOS`
    - `isDev`

State Model

- use a single global store for all file and directory info
    - this way, if info about a file changes, the change happens in one place and is reflected everywhere

- Frontend Models (MobX store)
  - `Workspace` → Overall/root store that encompasses all stores and some observables
    - `getTitle()`
    - `tabs: Tab[]`
    - `storeUrl` will be null if no store is linked
    - `Theme` → each Workspace has a `Theme` instance that is used to manage all things related to the current theme such as
      - `themeName`
      - `themeId` (might not be present for themes linked to a store - local themes won't have them)
      - basic theme config info
      - `themePath`
      - `themeFiles` (this will be the representation of the file tree)
      - The `Theme` object is populated/created when a new theme folder is chosen

```
interface ThemeInfo {
    name: string;
    heliumId: string;
    shopifyId?: string;
    path: string;
    version: string;
    author: string;
    files: ThemeFSNode[]
}

// the config/settings_schema.json should be watched for c
// in order to update the Theme Info Side Panel

interface ThemeFileSystemEntry | ThemeFSEntry | ThemeFSNo
```

```
    path: string;
    basename: string
    isFile: boolean;
    language?: LanguageMode.(...)
}

// create a new theme model like this:
Theme.fromThemeInfo(...);

const { themeInfo: loadedThemeInfo, files } = await heliu

// workspace.loadFromInitialState(state);
workspace.setTheme(new Theme(loadedThemeInfo));

// can also update theme like so:
workspace.theme.updateFromThemeInfo(themeInfo)

helium.shopify.onThemeInfoChange((updatedThemeInfo) => {
    workspace.theme.updateFromThemeInfo(updatedThemeInfo)
});
```

- `EditorStore` → store for handling all things related to the editors
- `Layout` → helps to try and manage all the panels and general workspace layout
- `FileSystemStore` → store to save all file systems items (files and directories) in a single place
  - think about this
  - in reality, this should be a simple Map
  - this might still work in the end since all `actions` should be batched together, so if it looks like this when getting a subtree, it should (hopefully) be fine.
  -

```
const openTree = action(() => {
    ...
    const subTree = await readDir(path);
    workspace.fs.saveEntries(subtree); //updateEntries()?
    ...
})
```

- Backend Models (Electron)
  - `HeliumApplication`
    - `dataStore` THIS IS NOT A PRIORITY RIGHT NOW
      - Schema

```
interface StoreInfo { // StoreInfo
    heliumId: HeliumId // id specificaly for use in Hel
    themeAccessPassword: string;
    url: string;
}

type HeliumId = string; // "helium-{uuid}"

interface HeliumDataStore {
    // see above for ThemeInfo
    themes: ThemeInfo[];
    stores: StoreInfo[];
    lastOpenedTheme: HeliumId; // think about this
}
```

    - `windows`
    - `focusedWindows`
  - `HeliumWindow`
    - `currentTheme`

- `currentLinkedStore`
  - the idea here is that different windows can be linked to different stores if they want to
  - Themes should be linked to stores if possible so that when a theme is chosen to be opened, that theme last used store is automatically retrieved
  - REMINDER: THEMES CAN BE DEVELOPED WITHOUT A STORE
  - THEMES CAN BE DEVELOPED/TESTED ACROSS MULTIPLE STORES
- `ShopifyCliClient`
  - `initTheme(url)`
    - A theme can be created without s store being linked
    - A theme can also be edited without a store being linked
    - However, a store must be linked to the theme for the preview and other features to work
    - In theory, you should be able to test/develop a theme across multiple stores
    - The only thing that should be stored in the `dataStore` is the last store that the theme was edited on
  - `getThemes()`
  - `pullTheme()`
  - `pushTheme()`
  - `startPreview()`
  - `endPreview()`
  - `isPreviewOn()`
  - Ipc events can be handled/scoped to each window individually instead of relying on the `BrowserWindow` of the event
  -

```
// preload

async function initPreload() {
    const currentWindowId = await ipcRenderer.invoke('g

    contextBridge.exposeInMainWorld('helium', {
        desktop: getDesktopApi(currentWindowId),
        ...
    })

}

const scopedEvent = (id, event, func) => () => ipcMain

const getMe = scopedEvent(currentWindowId, 'close-windo
    ...
});
```

Components

- `ApplicationContainer` (might not be a container, might just be a `div` with `display: flex`)
  - `TitleBar`
  - `HeliumWorkspace`
    - `SideBar`
      - `SideBarItem`
    - `SidePane`
      - `FileTreeView`
      - `StoreView`
      - `ThemeInfoView`

        - `EditorPane` (multiple of this allowed through an `EditorGroupPane`
            - `TabBar`
            - `CodeEditor`
            - `ImagePreview`
            - `MarkdownPreview`
        - `ThemePreviewPane`
            - `NavBar`
            - `Preview`
        - `StatusBar`
        - Should the `EditorPane` and the `ThemePreviewPane` be grouped together???

Preview State

- Starting: `OFF` → `STARTING` → `RUNNING`
- Stoping: `RUNNING` → `STOPPING` → `OFF`
- Preview starts
    - `STARTING`
    - `spawn`
        - if `previewUrl` reachable → `RUNNING`
            - If `previewUrl` is unreachable → `ERROR` → `STOPPING` → `OFF`
    - `error`
        - `ERROR` → `STOPPING` → `OFF`
- Stoping preview
    - Clean Preview process (make sure the `previewUrl` is no longer available)
    - `STOPPING` → `OFF`
    - if error, clean up preview process, `STOPPING` → `OFF`

Editor and Tab Implementation

- Using Monaco for the editor (VSCode-like experience)
- Work with Monaco models
- Also need to track whether a file has been created
  - Based on manipulating the `currentTab` , `tabs` , and `currentFile` structures
    - `currentTab` could be an index of the `tabs` array
    - `currentFile` should be the path (or consider `HeliumId`) of the file (to be referenced by the `workspace.fs.getEntry()` function)
    - changes in the `currentTab` should automatically change the `currenFile`
      - opening a new file should create a new tab an select it, which in turn should create and set the new model
- `<CodeEditor/>` props
  - `model`: Monaco model that represents a file
- Whenever a file is opened for the first time:
  - read the contents of the file
  - a new tab should be created
  - the model of that file should be created (`TODO:` do the models need to be stored???)
  - create a new tab
  - set the new model as the `currentFile` and the new tab as the `currentTab`
  - render the `model` in the `<CodeEditor/>` component
- When a tab is selected
  - If the selected tab is the same as the current tab, just ignore it
  - get the file that the tab represents
  - Get the model of the selected file from Monaco (or internal store)
  - set the new model as the `currentFile` and the new tab as the `currentTab`

    - render the `model` in the `<CodeEditor/>` component
- When a tab is deleted
    - If it is the only tab open
        - set the `currentTab` and `currenFile` to null (don't delete the model)
        - render the `<Placeholder/>` instead of the <CodeEditor/>
    - If the tab is the `currentTab`
        - set the `currentTab` to an adjacent tab (left) and change to the associated `currentFile` as well
        - remove the tab from the `tabs` array
    - If the tab is not selected
        - simply remove the tab from the `tabs` array
- When a file is deleted
    - If the file is the `currentFile`
        - delete the file from the fs
            - this should trigger file tree change based on watched dir
        - delete the tab (see steps above)
    - If the file is not the `currentFile`
        - get the associated model and dispose of it
        - if the file was open as a tab, delete the tab (see steps above)
- When a file is renamed (when a file is moved??)
- When a file is created
    - main issue right now is that it might not show up in the file tree.
    - should I manually add the file to the tree???
    - in terms of behaviour, we would want to have the file open and if the file was created in an unopen directory, the directory should be open

        - should we wait for the folder watchers to tell us that the file has been added????
    - so many edge cases

Supported File Types

- `.liquid`
- `.md`
- `.yml`, `.yaml` (general config like files)
- `.json`, `.json5`
- `.js`, `cjs`, `.mjs` (JavaScript and other JS supersets)
- `.ts`
- `.jsx, .tsx`
- `.css` (including CSS like languages/template languages)
- `.html` (html like languages)
- `toml`
- `.gitignore` type files
- images (using the `<ImageViewer/>`
- Pretty much all filetypes related to web dev and assets and `.liquid`
- If filetype not detected use a plain text

FileTree/ File Explorer

Approach

- The plan is to do everything “lazily” - don’t call on/use resources until needed (dynamically)
    - It is also important to know that the tree is more for presentation/visual purposes, so it does not require that much data and should be kept as light as possibly

- when the user picks a theme/folder to open, the steps are just to:
  - read the directory and get a list
  - validate if the directory is actually a Shopify theme (check theme structure)
  - determine which items are directories or files and filter out junk (like `.DS_Store` and vim swapfiles)
  - detect file type
    - handle special file names (eg: .eslintrc)
    - test if it is a text file and return file type if is
    - if not, test if it is an image and return file type (image) if it is
    - else test if it is a binary file and if it is, say that it is
  - Should be sorted by alphabetical order (can use `.localeCompare()`)
  - return the list to the frontend
- On the frontend, that initial list will be rendered as the initial state (with each item being a separate file tree item)
- On file tree item click:
  - if a file was clicked:
    - If the file is a binary file, show error message box
    - If an editor/model is already open for that file, redirect there to that tab/editor
    - The file will be read (TODO: need to make sure it is fast and efficient, especially for large files)
    - Necessary info about the file (like encoding and language) are detected
    - New editor/model/tab is created and rendered
    - Doing it this way makes sure that only files that are used are read
  - If a directory item clicked

- If there is a cached “subtree” for this directory, render it
- do similar steps for the theme/folder opening (read the directory and analyze the children)
- cache this subtree (very important for future use)
- render the “subtree”
- Another way to do this would be to try and render it as a list with different “depths” (margin) (think of a “file tree” as a flat list with depth)
    - this way, the “tree” structure is a simple array and removing and adding subtrees when a directory item is toggled is simply removing the “subtree” array from the tree array
    - This in the long run should take up less memory and time as there is less objects used (including no nesting), and the array would be linear (would removing the items be fast???)
    - Could help out with virtualized rendering....
    - need to confirm is it would be more efficient than a tree structure
- This makes sure that only used directories are read

New File/Folder

- show New File/Folder dialog and get the path
- create the file and wait for it to be done
- read the parent dir and take note of any expanded children (expand them as well)
    - problem with this is that it will be a pain to handle is the user nests the file in a bunch of sub directories
- when that is done, if it is a file, select open

File Explorer Data Structure

- Basically the FileExplorer is uses an array structure internally and is rendered using it
- This is because of
  - virtual list rendering (easily done with array/list of item)
  - easier (and probably faster) to manage/work on than an internal tree structure
- When a directory is open, it is inserted **flat** into the tree
-

```
// directory structure

/*
assets
sections
    --footer
    --header
*/

// as an array

["assets", "sections", "sections/footer", "sections/header"]
```

- The way we do indenting is with the `depth` property
- this makes it **look** like a tree u
- supports 3 operations
  - `colapse(directory)`
    - from the index of that directory, delete the entire subtree from the array
      - from that index, iterate from the index and count all subtree items
      - after that, splice that subtree from the array

-

```
function collaps(dir) {
    const dirIndex = findIndex(dir);

    let subTreeLength = 0; // or 1?
    for (let i = dirIndex; i < this.data.length; i+
        if (this.data[i].path.startsWith(dir.path))
            // the good thing about this method is
            // handles subtrees of subtrees (expande
            // since they all have the inital dir as
            // all the child subtree item's paths st
            subTreeLength += 1;
        }
    }

    this.data = this.removeSubtree(dirIndex + 1, sub
}
```

- `expand(directory)`
- `refresh()`

Tree vs Array?

- Tree Data Structure
  - Pros
    - Handling subtrees is a breeze (much easier than array) for `collapse()` and `expand()`
      - adding a subtree → after getting the selected directory node in the tree, set the subtree like this: `node.items = subTree;`
      - removing a subtree → after getting the selected directory node in the tree, remove the subtree like this `node.items = null`

        - another approach would be to simply toggle a boolean property like `node.isExpanded = false` , and we decide whether to render the subtrees based on this value
            - this is good as it theoretically makes both operations `O(1)` , but requires having the tree structure continually grow in size in memory, even when certain subtrees are not being used/rendered
    - expansion states can be represented as part of the data (if it is expanded, the child nodes are there, if not, it is null)
    - makes using a subtree cache a lot easier since we don't have to think about child expansion states (which in turn, makes collapsing and expanding faster)
    - Makes file system updates (potentially) a lot easier
        - if a directory is updated, all we need to do is get that node and update its subtree (this would be even better if getting a node is `O(1)`)
- Cons
    - can take up lot of memory (especially if I keep a map of nodes to paths to keep subtree retrieval `O(1)`
    - tree would only be the internal data structure and I would probably still need to flatten tree into an array in order to get the benefits of virtual rendering (not cheap, most likely exponential time complexity)
    - Searching for nodes (without the a node map) could potentially be expensive (`O(N)` with `N` being the number of entries in the tree or exponential time complexity)

- Array Data Structure
    - Pros
        - Closest to the final structure that is used directly in rendering

- all algorithms are going to have a time complexity of O(N), with N being the number of files and directories
- handling subtrees can be pretty straight forward for `collapse()` and `expand()` (not as easy as the tree tho)
    - adding a subtree → splice the subtree array into the array after the selected directory
    - removing a subtree → find the number of entries in that sub tree and delete that number of entries from the array staring from the selected directory (added bonus of bing `O(N - i)`, where `N` is the number of entries and `i` is the index of the selected directory item in the array structure

- Cons
    - Using a sub tree cache becomes **SIGNIFICANTLY** harder
        - how do you handle when a child directory in the sub tree is expanded? the only way to handle that is to “rebuild” that sub tree 🤮. That brings MORE issues: there could be multiple parent directories to a single, so which cache items do I update???
        - Overall, it involves A LOT of complexity that is much easily prone to error and inconsistencies
    - Expansion states have to be managed manually in a separate `Set` data structure, and it will need to be kept up to date when directories are removed
    - Directory updates are just “okay” in complexity, but can easily be prone to error

```
/*

assets_cool
assets
-- test.ts
```

```
*/

// looking for assets/test.ts
// should test if they have the same dirname

// test if the given path starts with the node path
// check if they have the same dirname
```

Notes

- The cache used in the file tree will need to be some kind of `Map()` , with the keys being the directory path, and the value being the “subtree”/content array
- In order to dynamically remove subtrees from the internal data structure, tree operation functions will be needed to perform needed actions and produce a new tree with updated state (maybe contained in a hook)
  - An internal tree data structure might not be needed
    - using the cache, I can just loop over the “subtree” and when a hit some open directory (can be stored in an array/`Set` or map), I then just render that subtree at a deeper depth (increased left padding)
- When reading the directory, it would be useful to be able to ignore all hidden files. Unfortunately, it is not as easy as looking for a dot as some files/folders are hidden in other ways. This will probably be ignored for now
- To make things more efficient, I can move this stuff to a different thread using `fork()`
- An advantage of using the open folder method is that I can choose to simply watch those directories and update the cache/view as needed.
- Only load supported file???

- When creating a file with inexistent parent directories, take note of all parent directories and add them to the open array
- The folder holding a file **does not** have to be open (i.e.: a file can be open with their parent folders not being opened in the file tree)

Theme

- theme will be based on Textmate grammar JSON file
- Theme will be loaded on the main thread as an object (`HeliumEditorTheme`)
  - when the `HeliumApplication` object is loading and initializing all services
- Themes are application wide (apply to all windows)
- The background color of each window will be set in the `HeliumWindow` class (meaning that the `HeliumWindow` will need access to the theme object)
- When the renderer state is initialized, go through the theme object and define CSS variables on the `:root` so that other parts of the editor can use it
- set the theme in the `shikiHighliter`

Tree vs Array?