mosaicly

Code Standards and Workflow Documentation

Product name:	Mosaicly
Team name:	Mosaicly Team
Revision number:	4
Revision date:	June 3rd, 2025

About

This code standards and workflow is a living document that will document the organization of the project, the workflow, and the best practices for the different programming languages and tools used. Layout inspired by Axolo.

Definition of Done

The current definitions for acceptable user stories and tasks are as follows:

User Stories

- The Product Owner (Lemon) accepts the functionality of all of the user story's tasks as satisfactory
- All the tasks' code has been merged onto Github via pull requests
- The merged code runs without runtime or compiler errors

Tasks

- The Product Owner (Lemon) accepts the functionality of the tasks as satisfactory
- The code runs without runtime or compiler errors
- The code has been merge onto Github via a pull request
- The code follows the coding style and guidelines as outlined in this document
- The code has been tested and stress tested manually as much as possible

Project Structure

High-level overview of the important aspects of the project structure and some notes. More information can be found in this SvelteKit documentation.

├ **src/**	- Contains most of the project files
│ ├ **lib/**	- Utilities and components folder
│ │ ├ **@const/**	
│ │ │ └ *dynamic.env.ts*	- Stores dynamic environment variables
│ │ └ **comp/**	- Components folder
│ │ │ ├ **canvas/**	
│ │ │ │ ├ **enums/**	- Canvas enums and defining constants
│ │ │ │ ├ **objects/**	- Canvas component objects and functionality
│ │ │ │ └ **utils/**	- Canvas functions and back end
│ │ │ ├ **layout/**	- Mobile layout
│ │ │ ├ **profile/**	
│ │ │ ├ **ui/**	
│ │ │ │ └ **icons/**	- heroicon SVG Svelte components
│ │ │ ├ *index.ts*	- Unused
│ │ │ └ *state.svelte.ts*	- High level states
│ ├ **routes/**	- The web application's file-based route definitions
│ │ ├ Example: **profile/**	
│ │ │ ├ *+server.ts*	- API endpoints that are used by the front end
│ │ │ ├ *+page.server.ts*	- Server-side/middleware code (with loads, actions, etc.)
│ │ │ └ *+page.svelte*	- Page definition (will likely use components from lib folder)
│ │ ├ *+layout.server.ts*	- Server-side functionality and auth before loading pages
│ │ ├ *+layout.svelte*	- Defines a layout that is common between pages
│ │ └ *+page.svelte*	- Home page
│ ├ *app.d.ts*	- Type and interface declaration
│ ├ *app.html*	- Page template that will be hydrated
│ └ *hooks.server.js*	- Handles server requests based on certain events (Hooks)
├ **static/**	

```
│ └ stylesheets/          - Defines global and reusable CSS styles
├ tests/                  - Folder to be used for unit testing
├ package.json
├ svelte.config.js
├ tsconfig.json
└ vite.config.js
```

Project Setup Guidelines

- Use `yarn`, ***NOT*** npm, pnpm, or other package managers
- Make sure your root directory has a `.env` file with the necessary API keys. This can be found on the secrets channel in the Discord server.

Version Control (Git and Github) Workflow

Git and Github Setup

- All developers except Lemon must do a pull request before merging their branch

Github Workflow

- Developers should make a branch to work on their task (with a clear branch name, such as the developer's name or the feature name)
- Developers should regularly fetch from the main branch, then merge their code and resolve merge conflicts, if any
- Developer should pull and merge changes from the main branch and any other necessary branches before pushing their code to the database
- Developers should thoroughly test their code before making a Pull Request or reviewing the code with Lemon

Error Handling

Svelte / SvelteKit

- Sveltekit middleware and API endpoints should use the error(status code) function
- Svelte should render UI if functionality errors occur (such as data not saving)
- Fallback or HTTP errors such as missing pages (404) should redirect to error pages

Naming Conventions

The following naming conventions or casings are stated, followed by the use cases for such conventions.

General Source Files and Folders

- lowercase: route (folder) names
- PascalCasing: class names and interface names
- + prefix: route file names

Svelte/SvelteKit

- PascalCasing: Svelte components
- camelCasing: functions, variables, everything else (including database fields)

HTML

- kebab-casing: id, class

CSS / SCSS

- kebab-casing: id, class, variables

TypeScript

- PascalCasing: Types, Interfaces, Classes
- camelCasing: functions, variables

Supabase / PostgreSQL

- snake_casing: database tables, attributes

Coding Standards and Best Practices

Formatting

- ALL strings will use double quotes (" "), not (' ')
- Typing runes should be done as `varName = $[rune]<type>();`

Svelte/SvelteKit

- NEVER use getElementById() or querySelector()
    - You can use bind:this instead (which allows the bi-directional reading of data between a parent and child component
- Make a component if there is repeated code
- Use $derived rune on $props (do this in most cases, but sometimes there are cases where you don't want your props to react)
    - Not needed on +page.svelte typically

HTML

- Avoid overloading elements with classes for specifying styles, sometimes it is better to use an ID for "one-off" cases

CSS / SCSS

Icons: heroicons and lucide icons

- Generally, try to follow Apple's HIC guidelines
- Standard unit = px
- Standard body text size = 16px font size, 22px line height
- Minimum touch area = 28px
- Standard Icon size = 24px

- Use the Figma as a guideline for design

- ALL components MUST include `@use "$static/stylesheets/guideline" as *;` so that there is a standard style
    - Also, only use colors from the guideline file, do not make your own color values
- Never use !important unless in extreme circumstances where overrides need to be forced
- Use flex box as much as possible
- ONLY use standard units (px) for absolute unit measurements
    - Using rem: May be done when needing spacing between components that is based on font size (like spacing between text)
    - Using %: In all reactive sizings
    - Using vw and vh: only used to adjust positional offsets that need to react to viewport size.
- Avoid overloading elements with classes for specifying styles, classes are best for "variants" of components
    - ID's are best for one-off styling
- Wrap calculations with `calc()`
- Mobile needs both the `:focus` and `:active` pseudo-classes
- Use only font-weight 400 (normal) and 700 (bold) unless otherwise specified

Specific Styling Cases

- Circle buttons
    - width: auto
    - aspect-ratio: 1/1
    - Border-radius: 100px

TypeScript

- Try not to use `any` when typing
- ***<u>Never</u>*** ignore TS compilation and syntax errors

Supabase/PostgreSQL

- Enforce RLS on all tables using (SELECT auth.uid()) and/or the authenticated role
- Normalize tables up to 3NF
- Use the public schema for now
- Put extensions in the extensions schema
- Avoid having NULL attributes (for empty strings, it is best to simply have them be empty)