M3SSAG1N8 Design Document (Group 37)

Nat Hill (wnh1, nat-hill), Zoe Katz (zrk1, ZoeKatz14),
Cameron Liu (hl101, Cami101)

1 Design Diagram Explanation

When we run the program (with `npm run start`), we centrally control it with `main.ts`, which handles the creation of the models, initialization of event listeners to call the data from the database via the models, and sending the data that corresponds to that via events. In addition, since our program is single page application, the initial user page and layout is managed in `index.html`. Then, the prerequisite styles are managed in `styles.css`. Any relevant listeners for updates or user inputs to UI are initialized in `view.ts` and then modified and managed in `main.ts`.

The bulk of the program is interacted with through our typescript components. There are three central display components that the user interacts with.

- `post-display.ts`: Displays the posts within the currently selected channel and workspace, alongside components to react and reply to each post if needed.
- `message-box.ts`: Provides the user with an input text box to allow them to input messages to the channel. This component is re-used when replying to a post in the channel.
- `workspace-btn.ts`: Allows the user to select, create, refresh, and delete workspaces in the current database.

In addition, we have the following models that

- `readModel.ts`: Performs any *read*-only operations on the data in our database. (Including managing server side event subscriptions)
- `writeModel.ts`: Performs any *write*-only operations on the data in our database.

Each of these models, as well as main, have reusable functions in `helpers.ts` that centralizes certain commonly used functions. Finally, we have tests in both `cypress` and `jest` that allow us to confirm that our modifications and program work as expected.

2 Design Principles

2.1 Single-Responsibility Principle

Within our program, we are attempting to follow the single-responsibility principle by separating out abstractions into specific and focused components that allow us to easily modify the code and control the program in focused areas.

For our modules, we split our model up into three parts: One model that performs reads (ReadModel), one model that performs writes (WriteModel). *Note that subscriptions are managed by the read model, as we considered them a type of read operation.* These abstractions allow us to easily call

each model when needed, but being careful not to mix read and writes where not absolutely necessary. In addition, it follows the "Naming" convention of this principle, as the names are simple and straightforward, describing to the user exactly what they should do.

In addition, the typescript components used in our code follow a similar structure. As seen in the diagram, each one has a singular and distinct purpose that they do in their entirety. `workspace-btn`, manages workspaces, in both selection, as well as creation, deletion, and refreshing. Everything related to workspaces can be be modified singularly within that button.

2.2 Open-Closed Principle

For the Open-Closed principle, we attempt to allow our program to be module and extendable without comprehensive modification. In this context, this takes the form of a modular design that keeps components separate, but with some shared functionality.

For example, each of our models needed to make typed fetches to the database with the same function, so we created a set of common re-used helper functions that had repeatable behavior within our 'helpers.ts' file. This allowed us to easily modify the helper function in multiple places if needed, without repeating code. In addition, by allowing multiple models, and components, it becomes easily for another developer to create their own components and models that can easily slot in place of the previous ones if needed, thereby extending the program.

One other example, is that when we need to reply to a post and display the input box below the post, we simply have to display a message-box component under the post in the channel, where you could easily use any interchangeable message-box module in its place.

2.3 Interface Segregation Principle

Throughout the development of our program, we considered how to maintain our program's minimal amount of data and dependencies. So, in order to do this, we avoided forcing clients implementing unnecessary behaviors in two key ways. First of all, in our helper function library, `helpers.ts`, we initialize the JSON validation code. Now, instead of re-initializing and compiling the schemas whenever we check a post, we can do it once, and not implement extra validation code in the multiple files in which it is used. In addition, we use separate schemas for posts, channels, and workspaces. Since we have types that correspond to each, we can only check for the minimal amount of data that each needs (for example, channels and workspaces don't have document data associated with them, where posts do!).

In addition, when you consider the event types we use throughout the program, we ultimately only ever implement the minimal amount of information and fields necessary to each event. This means that the recipient will never need to receive or parse unnecessary information while using the data from a given event, and the interface for the event uses the least possible info needed to achieve its goal.

3 Managing Concurrency

In order to manage concurrent users and posts, we focused our program around `owldb`, as it is a highly concurrent database that is essentially designed to fit the needs of this product.

One core aspect is that we centrally manage posts by subscribing to the posts collection corresponding to the currently selected workspace and channel. What this allows us to do, is that our program reacts and displays any new posts, reactions, or replies to posts in the channel as they are sent to the program in the server side event stream, and processed by our program.

So, whenever a message is posted or a reaction is created, we simple make a `fetch` request to the database, which then update the database, notifying our subscription feed, and updates each currently subscribed user, which could potentially be any number of people. It almost goes without saying, but we also make a request for all of the posts in a given channel and workspace when it is selected for the first time.

Instead of refreshing the entire channel every time a post is received, we use a queue of posts that stores posts that do not yet have their parent on the screen, and allows any incoming posts to be displayed on the page without reloading.

Because all of our server-side operations, such as creation and deletion of work-spaces and channels, and all operations on posts, are centralized around our database, this lets us refresh posts and channels when the user chooses to, and keep a core set of data that is maintained concurrently and reflected live within our program's frontend.

4 Accessibility Principles

4.1 Perceivable

1.1: Our program is successful in terms of Text alternatives in that we...

- The only non-text content are icons for buttons or emojis
- All of our emojis and non-text icons have alternate ARIA labels that explain what they are clearly.

1.4: Our program is successful in terms of Distinguish-ability in that we...

- We use high contrast colors across our application to make clear posts, the navigation bar, and everything else from the background.
- By using primarily black on white, and black on very light grey, the contrast in our website is very high, with a ratio of 19.26 : 1 on text, and similar levels on emojis and other components on the website
- Text and components use `rem` instead of `px`, keeping relative page views accessibility when resizing the page over 200% zoom.
- All visual presentation of text has black on light gray background, creating a very high level of contrast and readability.
- We use a simple sans-serif font that is readable on all web platforms.
- All items wrap around and/or are scrollable such that nothing is cut off from the page or requires unreasonable navigation to find content, such as our scrollable channel box.

4.2 Understandable

3.2: Our program is predictable in that we

- Our components are modular and repeatable in a way that ensure every component has distinct labeling and behavior that does not change across components of the same type.
- Unique post IDs are identified with human-readable labels and contents on posts.

- Every mechanism for navigation (the post display, and workspace selection button) is in the same relative order and position on the page each time they are created and accessed.

3.3: Our program is successful in terms of Input Assistance in that we...

- Label all user inputs with default texts showing you what they should do
- Any input errors with invalid names or submissions are caught and displayed to the user on a modal popup.
- The issue with any input errors is clearly and distinctly displayed to the user, showing them how to fix the error in a human readable format.

5 Extension

Our extension is **schedule sending posts**. We thought that this would be helpful, as our group uses a feature like this on email all the time, and certain messaging apps support this as well. Sometimes you want to send a post, but it's too late, or the recipient would be more likely to see it at a different time. This is where schedule sending a post comes in handy! What happens within the program is that you select a date and time for the message to be send within the message box GUI. In addition, we validate that the time is not in the past. Then, once this date and time is selected, we add the scheduled post a queue that is maintained by the subscription model. This queue checks if there are any posts that should be created every time the SSE stream updates, and POSTs the corresponding post to the database when the schedule time is met, making it available to other users at the selected time.

When the post is displayed, it has the timestamp corresponding to when the post was scheduled to be sent, not when it was initially sent. As such, it is organized on the screen with that in mind. However, one limitation that should be noted, is that as the scheduled message queue is maintained by the typescript application, and thus the application must remain open or the scheduled message will not be sent. We considered putting this information in the `extensions` attribute of a post in the database, but that wouldn't work well for implementations of the messaging app that do not check the extensions, so we thought this would be more extensible across multiple platforms.

6 Design Diagram

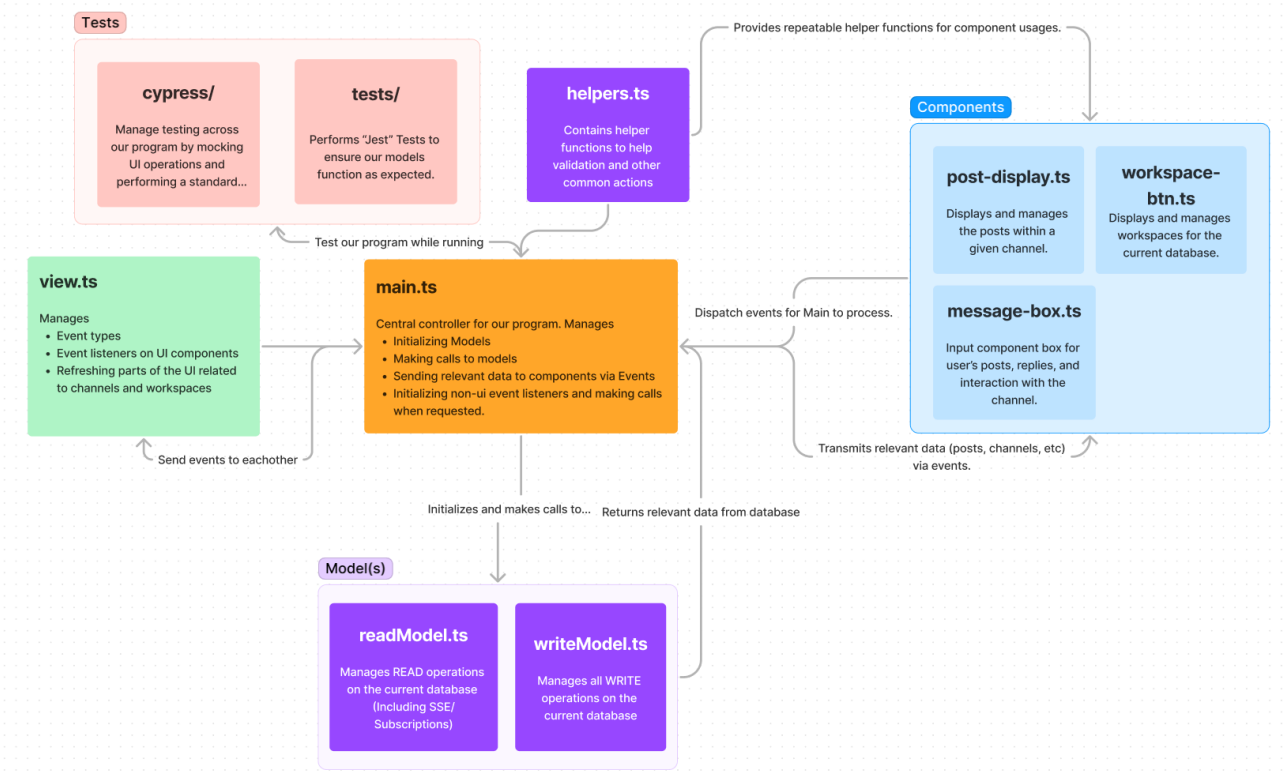

Figure 1: Design flow for our M3ssag1n8 application. Starting from `main.ts` with model initialization, we delegate to the `components` for the core of the application. *See Section 1 for further explanation.*