Eliza Marija Kraule emk6 elizaMkraule
Jonah Yi jwy4 JonahYi
Shivam Dalmia sd95 shivam-dalmia

Description of Design:

The following is a breakdown of the design through the flow of the program:

- The main() function starts the application when the DOM content is loaded.
- In main the Controller is instantiated and upon construction the Model, View and Validator are instantiated and set up. All event listeners are set up for the Controller and the View is ready for the user to login by using the custom web component LoginPage.
- Upon a user inputting a username in the View, a custom event “LoginEvent” with the username is dispatched to and handled in the Controller which in turn delegates the username from the event to the Model to authenticate the user in the Database. Model then receives the authorization token, therefore successfully logs in the user meaning the Controller can call an update on the view so the user can view the contents of the messaging application. The View gets updated to display the custom web component WorkspacePage displaying the main workspace area where workspaces, channels and posts can be interacted with along with the username of the user and an option to logout.
- Now the User can perform multiple actions such as creating a workspace, opening an existing workspace, creating a channel, opening a channel and reading and replying to posts within a channel. All of these actions send out custom events like “createChannelEvent”, “logoutEvent” etc to the Controller which in turn interacts with the model and depending on the event creates or retrieves resources from the database. Upon successful or unsuccessful interactions with the database through the model,the controller updates the view accordingly like reflecting opening a channel or reflecting any errors like trying to create a channel with a duplicate name.
- Upon interaction with the database in the Model, any responses received from the database get validated with the Validator to ensure consistency of data in the messaging application.

Design Principles

1. Single Responsibility Principle: All our modules and their corresponding classes, structs, and functions focus on one part of the application and do it completely, regardless of the operations of the other modules. For instance, the “post” module only deals with the display and functionality of a singular post within a channel. It is within a channel and amongst other posts, but does all the operations for this post itself. This lets other modules rely on post operations fully when using its functionality. Our module names are a great indicator to show which tasks a module/class does. Logging is handled in login, validation handles validating, workspaces deal with workspaces and so on.

2. Open-Closed Principle: Our architecture uses fixed exports and imports in order to uphold the open-closed principle. Though many modules import other modules, the modules trust that the other modules will do the intended operations to the fullest extent. Oftentimes an export class is defined and its required attributes are given, then the imported module is free to construct and do its operations as it sees fit without oversight from the module that imported it. Thus, whenever an imported module is extended with design changes or extensions, such as our superscript/subscript extension in “post”, modules that import this extended module do not need to be modified, thus upholding the open-closed principle.

3. Dependency Inversion Principle: The dependency inversion principle (i.e. decoupling) was maintained in our architecture by organizing our modules into a Model-View Adapter. Separating the modules into Model, View, and Controller, our architecture made sure that modules communicated in specific ways to avoid unnecessary dependencies. Our View modules including “view” and related modules “login”, “posts”, “channel”, and “workspace” send user input to the Controller “main” as events. The main module will then receive these events through event listeners and call functions of View and Model accordingly based on current data and the event given. Model modules such as “model” will then update the data structure depending on the event. Model also sends out events back to Controller when changes in the data occur such as when another user posted on the same channel, and this event can cause Controller to call methods in View. Overall, the modules are depending on each other using abstract methods such as event calls and dependency is controlled by the Controller which acts as a bridge between the View and Model. Thus, dependency and the dependency inversion principle can be preserved.

Accessibility Principles

In our code we implemented two of the POUR accessibility principles - Perceivable and Operable. To meet the perceivability principle we focused on ARIA labels and the contrast of colors. The ARIA labels improve the accessibility of the website for screen reader users making it easier for them to interact with the content. Paying attention to color contrast ensures that everybody is able to perceive the website clearly and correctly regardless of their visual capabilities. To meet the principle of operability we ensured that the website is operable with a keyboard allowing users who cannot use a mouse to navigate and use the website effectively.

Extension

For our application extension we added two additional formatting options: superscript and subscript. This extension adds two additional buttons for our text and reply boxes which are “superscript” and “subscript”. These function similarly to “bold”, “italic”, and “link” as a user may type the markdown corresponding to each formatting option (^^ for subscript and __ for superscript), select some text and then click the button to add the formatting symbols around the select, or just click the button to insert formatting symbols into the text box. This extension

does not interfere with the previously established formatting options, and although it will lead to messages that are not fully decipherable by a client without the extension, the application itself should work correctly.

Concurrency management

Concurrency was managed in our system primarily using promises. Promises are used throughout several different modules in order to keep the algorithm running without having to stop for a single operation. Additionally, operations like .then() and .catch() were used alongside promises in order to serialize specific asynchronous operations and guarantee that errors are handled for. We also made sure to always return promises so we can use them instead of leaving them lingering in operation. We also used await to wait for operations that use promises.

We dealt with the double submit problem entirely in view. When “view” is about to upload a post using the send button, we make sure to check a parameter to first see if another post is currently being uploaded. This parameter turned to false before dispatching an event that uploads a post, and turned to true when the operation for redisplaying posts or displaying an error message popped up. These indicated that the upload completed and failed accordingly based on our code, which let “view” change the parameter’s value accordingly without having to do the uploading to data itself.

Architectural Diagram of our system.