M3ssag1n8 Design Doc - p2group61

Names, NetIDs, and GitHub IDs:
Kush Mangla (NetID: kvm3; Github: kushmangla1963)
Nicholas Alamir (NetID: na56; Github: nicholasalamir)
Jialiang Yang (NetID: jy105; Github: KvnY07)

Description Of The System Using The Architectural Diagram

The M3ssag1n8 system is a modular, web-based messaging application built with a clear separation of concerns. It integrates various components to handle user interactions, manage API requests, validate data, and ensure smooth initialization of the application. For our system, we used the MVA pattern. Below is a description of all our components and their roles:

1. **Core Components** (Internal Logic):
   - **Model.ts**: Placed directly under the src directory, this handles the internal data structure and central application logic, ensuring consistent data flow within the system.
   - **Fetch.ts**: Placed directly under the src directory, this provides utility functions to make API requests using typedFetch, abstracting repetitive code.
2. **InitializeApp.ts** (App Setup):
   - Placed directly under the src directory, this bootstraps the application by connecting all major components, including views, adapters, and core components.
   - It integrates the user-facing interfaces (Views) with the backend logic (Adapters) and sets up the initial state.
3. **Adapters** (API Interaction):
   - Placed under the src directory, this acts as an intermediary between **Model** and **Views** and contains the following files:
     - **LoginAdapter.ts**: Manage user authentication workflows.
     - **WorkspaceAdapter.ts**: Facilitates workspace-related API calls, validating JSON data against schemas.
     - **ChannelAdapter.ts**: Handles API requests and responses for managing channels, including schema validation.
     - **PostAdapter.ts**: Manages posts and validates them against schemas.
   - Adapters depend on **Fetch** for executing API requests, **Validators** for ensuring schema validation, and **Model** to access OwlDB elements like tokens required for API calls.
4. **Views** (User-Facing Components):
   - Placed under the src directory, this defines the user interface and interaction layers and contains the following:
     - **LoginView.ts**: Provide UI for user authentication.
     - **WorkspaceView.ts, ChannelView.ts, PostView.ts, StarView.ts**: Display and interact corresponding entities (workspaces, channels, posts, and stars).
     - **Errorview.ts**: Displays error messages and ensures user feedback for issues.
   - Views utilize **Adapters** to fetch and manipulate data but do not directly interact with **Model**, adhering to the MVA pattern.
5. **LogoutHandler.ts** (Logout Functionality):
   - Placed directly under the src directory, this handles all logout functionality, making sure that the UI is refreshed back to the login page and that the token is deleted from the backend.

6. **Schemas** (Predefined Schema JSON files):
   - Placed outside the src, this includes JSON schemas that define the structure and validation rules for various entities:
     - **WorkspacesSchema.ts, ChannelsSchema.ts, PostsSchema.ts**: Ensure data consistency across workspaces, channels, and posts.
   - These schemas are used by the adapters for validation using **Validators.ts**.
7. **Validators.ts** (Handles Validation):
   - Placed directly under the src directory, this implements JSON validation, depending on schemas to ensure all inputs coming from OwlDB conform to the required structure before they are processed by the system.
8. **Static Components**:
   - **index.html**: Provides the base HTML structure for the application interface.
   - **styles.css**: Defines the visual appearance and layout of the application.

Dependencies:

- **InitializeApp.ts** sets up the entire application, ensuring proper connections between **Adapters**, **Views**, and **Model**, including logout functionality.
- **Adapters** heavily depend on **Fetch.ts** for API interactions, **Validators.ts** for input validation, and **Model.ts** for access to data.
- **Views** rely on **Adapters** for API communication but maintain independence from **Model**, ensuring clean separation of concerns.
- **Schemas** serve as a foundation for both **Validators** and **Adapters**, ensuring a consistent data structure throughout the system.

Design Principles

The Single Responsibility Principle (SRP) is prominently reflected in our project's folder organization under the src directory, where functionality is basically divided into **Adapters**, **Model**, and **Views**, adhering to the **MVA** pattern. Each component serves a specific purpose, ensuring that no component has overlapping responsibilities. For instance, **Model** defines data structures and manages the logic directly associated with the core data, such as workspaces, channels, and user interactions. The **Views** folder is responsible for handling UI rendering tasks, such as displaying workspaces, channels, and posts, ensuring a clear boundary between data and its presentation. Meanwhile, the **Adapters** folder acts as an intermediary, facilitating data flow between **Model** and Views. By ensuring the separation of concerns, this modular organization allows each folder and its corresponding components to maintain a singular focus. This not only simplifies the codebase, making it easier to navigate and maintain, but also makes future extensions and modifications straightforward without inadvertently introducing bugs or complexity. Adherence to SRP in this way enhances the project's maintainability and scalability.

The Liskov Substitution Principle (LSP) is effectively incorporated into our project through both the use of TypeScript type guards (e.g., isWorkspace, isChannel and isPost) and our class-based design for views, such as class PostView extends HTMLElement. These type guards ensure that components interact with consistent and substitutable interfaces, enabling objects like JsonResponse to be safely passed wherever they are expected without breaking functionality. Similarly, by extending HTMLElement, classes like PostView adhere to the behaviors and expectations of their base class. This ensures that any instance of PostView can replace an HTMLElement in the application, promoting compatibility and flexibility. The adherence to LSP guarantees that derived or extended components can integrate smoothly into the system, maintaining robust interface handling while creating a scalable and reliable architecture.

The Open/Closed Principle (OCP) is clearly demonstrated in our project through the use of reusable TypeScript files like **Validators.ts** and the extensible design of components such as **PostView.ts** and **ChannelView.ts**. These files or components have been designed to be open for extension while remaining closed for modification, meaning new functionalities can be added without altering the existing code. For example, we can seamlessly introduce new validators or extend UI components to accommodate additional features without modifying the application's core logic. This approach minimizes the risk of introducing bugs when implementing changes and ensures that the existing functionality remains intact. Adhering to OCP makes our project inherently scalable, allowing for efficient growth and adaptation to new requirements while maintaining a robust and reliable codebase. This design principle promotes flexibility and ensures long-term maintainability and stability.

Accessibility Principles

We adhered to the WCAG principles to ensure our system is fully accessible:

Perceivable:

Our system ensures that all content is perceivable without exceptions. This was achieved by carefully selecting font families, font sizes, and color schemes that meet appropriate contrast ratio requirements (minimum AA level of 4.5:1 and AAA level of 7:1 where possible). To support colorblind users, we avoided using green and red together and ensured that color is never the sole means of conveying information. All interface elements are visually intuitive, with hover actions for buttons that distinguish actionable from non-actionable elements. Additionally, our system supports dynamic resizing, ensuring all elements adjust proportionally when users zoom in or out, maintaining visibility at any magnification level up to 200% without loss of content or functionality.

Operable:

All system elements are navigable using either a mouse/trackpad or a keyboard, supporting the WCAG guideline for keyboard accessibility. Buttons and text inputs can be interacted with via mouse clicks or keyboard navigation (e.g., tab and enter/return keys), and clear focus styles are applied to focusable elements. We ensured no reordering of focusable elements and used semantic HTML for consistent and logical navigation. Interactive elements can be activated using "keydown" events (e.g., return keys), ensuring accessibility for users with mobility impairments or those relying on assistive technologies.

Understandable:

Our system is designed to be intuitive for all users. Upon login, users see a welcome message with clear instructions for their first action. Errors are communicated through an actionable error dialog on the right side of the screen, which clearly explains the issue and recovery steps without technical jargon. Users can dismiss errors via an ‘x’ icon or reattempt the action, which smoothly resolves the error. Input fields include clear labels or placeholder text, such as *“Please enter a username”* for login and *“Please type your message here…”* for sending a post. To support users requiring screen readers, we avoided vague elements like divs and spans, instead using semantic HTML to improve clarity and maintain proper structure.

Robust:

Our system was rigorously tested on multiple browsers (Chrome, Safari, and Firefox) and operating systems (iOS and Windows). We validated our HTML to ensure compliance with accessibility standards, maximizing compatibility across platforms and browsers. To further enhance accessibility, we implemented ARIA roles and properties where necessary to provide additional context for assistive technologies (especially for visual features used in the post view like reaction icons, reply icon, star icon, etc.).

Application Extension - Star

We added the functionality for users to **star** posts in any channel, marking important messages for later reference. All posts have an empty star by default, and a user can simply click on the star to star it, changing it to a filled gold star to align with what users are familiar with through their use of other systems that have a similar feature. A conveniently located button in the post view allows users to access all their starred posts in a read-only *starred view* mode. The read-only mode ensures that the extension does not interfere with other required functionalities. While the status of whether a post is starred is visible to all users, each user can privately view their starred posts in a personalized, pop-out view. This *starred view* operates independently and does not affect real-time updates in the channel. Users can toggle back to the regular post view at any time, keeping their starred posts just one click away. A user can also “unstar” a post if they change their mind or accidentally clicked on the star icon. If a post is unstarred, it is immediately removed from the *starred view*. This makes our extension robust as it gives the user the ability to recover from potential errors. Additionally, if a reply is starred but its parent post is not, the reply appears in the starred posts view with a note saying, *"This post is a reply,"* for clarity. Overall, our extension is unique and robust, and significantly enhances user functionality.

Concurrency Management

In our messaging app system, we implemented several strategies to effectively manage concurrency and ensure a seamless user experience. One key feature is live updates relying on the **SSE**, which automatically refresh posts in real-time for all users with the same channel open, ensuring everyone sees the latest changes without manually refreshing. The post messages sent in the same channel will be organized in the order of the time they are created at, making sure that the post view is displayed consistently when there are multiple users sending messages. Additionally, we included **refresh** buttons for workspaces and channels, allowing users to manually check for newly created channels or workspaces that other users may have added to the shared database. To ensure the robustness of our concurrency handling, we conducted extensive testing with multiple users interacting with the same database simultaneously, identifying and resolving potential edge cases.

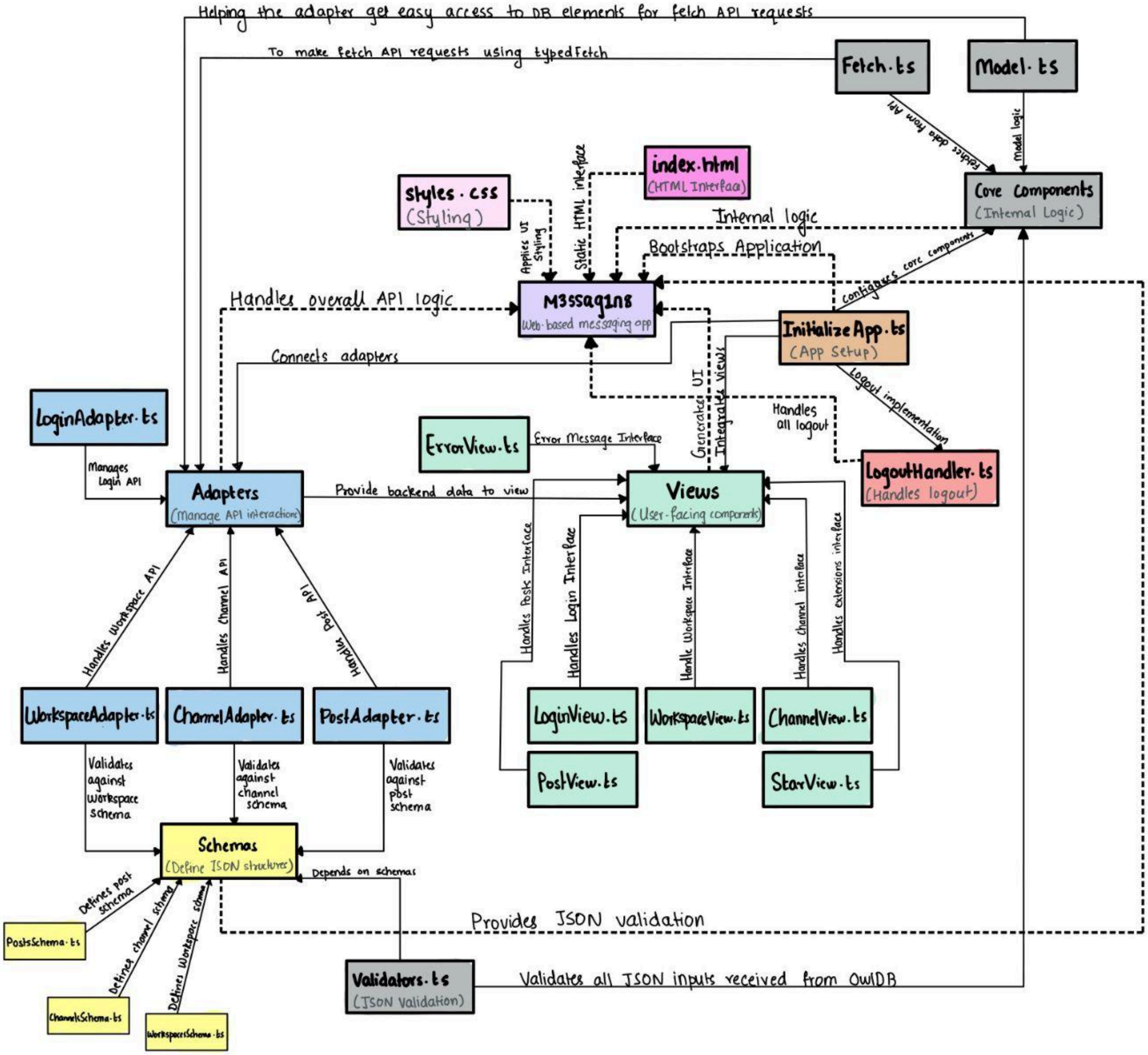
Furthermore, we leveraged OwlDB's *nooverwrite* mode to prevent users from creating workspaces or channels with duplicate names. This approach prevents race conditions, ensures consistency across the frontend and backend, and maintains a conflict-free, synchronized state for all users, even during simultaneous interactions, thereby significantly enhancing the concurrency and reliability of the system.

Our system also includes robust **error handling** through an intuitive UI element of *error dialog*, which plays a critical role in supporting concurrency, especially during concurrent operations like the deletion of workspaces and channels. For instance, if one user deletes a channel while another user still has that channel open, the system ensures that any attempt by the latter user to interact with the deleted channel triggers a real-time error message. This message informs the user that the action cannot be completed and advises them to refresh the system. By providing immediate and clear feedback, this mechanism keeps all users informed about the current state of the system, ensuring consistency across the platform. This approach not only prevents conflicting operations but also helps maintain a synchronized and transparent user experience, which is essential for effective concurrency management.

To handle potential write conflicts, we implemented a **last-write-wins (LWW)** policy, which uses metadata such as timestamps and user IDs to identify and apply the most recent changes. If conflicts arise, users receive clear, jargon-free error messages indicating that their action failed and needs to be retried. This transparent conflict resolution mechanism ensures updates remain consistent across users while avoiding unnecessary confusion.

Additionally, we extensively used **async/await** to manage concurrency and streamline asynchronous operations. Network requests for fetching or updating data were handled with async functions, ensuring a responsive UI while waiting for server responses. By using await, we avoided callback-related issues and ensured readable, maintainable code, especially for sequential tasks like fetching metadata before processing user actions. This design optimized the system's performance under high concurrency and enhanced the overall user experience by enabling smooth, non-blocking interactions.

Architectural Diagram



Helping the adapter get easy access to DB elements for fetch API requests
To make fetch API requests using typedFetch
Fetch.ts
Model.ts
Fetches data from API
Model logic
Core Components
(Internal Logic)
index.html
(HTML Interface)
styles.css
(Styling)
Applies UI Styling
Static HTML interface
Internal logic
Bootstraps Application
configures core components
Handles overall API logic
M3ssag1n8
Web-based messaging app
InitializeApp.ts
(App Setup)
Connects adapters
Generates UI
Integrates views
Logout implementation
Handles all logout
LogoutHandler.ts
(Handles logout)
LoginAdapter.ts
Manages Login API
Adapters
(Manage API interactions)
Provide backend data to view
ErrorView.ts
Error Message Interface
Views
(User-facing components)
Handles Workspace API
Handles channel API
Handles Post API
Handles Posts Interface
Handles Login Interface
Handle Workspace Interface
Handles channel interface
Handles extensions interface
WorkspaceAdapter.ts
ChannelAdapter.ts
PostAdapter.ts
LoginView.ts
WorkspaceView.ts
ChannelView.ts
PostView.ts
StarView.ts
Validates against workspace schema
Validates against channel schema
Validates against post schema
Schemas
(Define JSON structures)
Depends on schemas
Provides JSON validation
Defines post schema
Defines channel schema
Defines Workspace schema
PostsSchema.ts
ChannelsSchema.ts
WorkspacesSchema.ts
Validators.ts
(JSON Validation)
Validates all JSON inputs received from OWlDB