COMP318 Group 50, Project 2 Design Document

Group Members

- **Name**: Jason Han, **NetID**: jh146, **GitHub Username**: jasonhan3
- **Name**: Manning Unger, **NetID**: amu1, **GitHub Username**: tobahhh1
- **Name**: Jingwu Wang, **NetID**: jw133, **GitHub Username**: Jingwu-01

Design Principles

Single Responsibility Principle

Our user interface is built using web components, allowing us to encapsulate specific functionalities within each component. Our workspace list and dialog handles all functionalities related to selecting, adding, and removing workspaces. Our channel list and dialog also handles all functions related to selecting, adding, and removing channels. For our posts, we have created different web components encapsulating specific functionality. One component, our Post Display component, is solely responsible for inserting or modifying posts based on the adapter's instruction. Posts themselves are responsible for displaying the post content. Each reaction button is its own web component that handles a user reacting to that post.

Within our adapter, we have four separate adapters for logging in/logging out, workspace, channel, and post events. Each adapter is solely responsible for events related to that specific data type.

We additionally have another module called a 'state-manager', which stores information about the state of the user interface (such as the currently opened workspace, channel, and posts displayed). This module is responsible for storing this data, requesting data from the model, and updating the view when the state of the application changes; it operates by discretion of the adapter (and is not triggered by events in the view nor model).

Within our model, we have separate packages for making requests to a server following the OwlDB API specification for each of workspaces, channels, and posts.

Interface Segregation Principle

Our view has the sole role of displaying information to the user. It does not decide what gets displayed (this is the job of the adapter). In general, our view operates by taking in updates. Each update has an "operation" field that tells the view what operation to perform as well as the data to display. Thus, when displaying workspaces, channels, or posts, our view looks at the operation (either an insertion, removal, or modification) and performs the specified operation on

the workspace, channel, or post. Our view also displays that the user's inputs are 'in progress' by means of a loading spinner.

Our adapter serves the role of deciding what should be displayed on the view and what request should be made to the model. The adapter does not care how the view displays the information, nor how the model makes the request. For all workspace, channel, or post events that the adapter receives from the view, the adapter makes a request to the model based on the event and calls a function on the view with the specified update (either to insert, remove, or replace the data). Additionally, for posts, the adapter receives events from the model, ensures that the post's parent is received and decides the position to upsert the post. Whenever the adapter receives an event from the view, it stores its ID, and notifies the view when that event is no longer pending (either it is done, or the event failed due to an error).

Our state manager stores the state of the application, as we decided that the adapter does not need to store information related to the user interface (the adapter's job is to listen for events from the view or model and adapt the information, not store it).

Our model has the sole role of making requests to a database server. It does not care who makes the request, nor does it care how the information from the request is used.

Dependency Inversion Principle

We follow the model-view-adapter pattern, where our model and view are completely decoupled from each other (they do not import each other). The model does not store any information about the view, and the view only cares about the information it receives from the adapter, and does not depend at all on the model's implementation. The adapter itself does not directly import our view and model. Instead, the controller constructs the adapter by injecting in view and model objects that meet a view and model interface, respectively. Our adapter hence depends on interfaces of the view and model, not on specific implementations.

In addition, within our adapter, we have four adapters, each of which are decoupled from each other. Our adapters themselves depend on an interface for our state manager and not a specific implementation.

Accessibility

Perceivable: All non-text content has a text alternative. For example, the owl image has an alternative attribute that describes the image. The iconify buttons all have arial labels that describe the functionality of each button and have role labels specifying they are buttons. Also, we avoided using divs and instead implemented semantic tags for all html files. Last but not least, we choose colors carefully so that the contrast ratio of the text and background is at least 4:1. In this way, visually impaired users could navigate through the website and find it friendly to use.

Operable: The functionality of the website is fully accessible via keyboard navigation. Specifically, users can use the "Tab" key to traverse all clickable sections of the site. We manipulated ":focus-visible" CSS classes to highlight focused elements with red squares. After

focusing on the desired element, users can simulate a mouse click by pressing the "Enter" key. This action triggers exactly the same response as a conventional mouse click, which ensures an accessible browsing experience for mobility impaired users.

Extension

Our extension is the ability to star posts, specific to individual users in each channel in each workspace. For each post, at its top right, there is an additional ‘Star’ button that you see. It behaves similarly to a reaction (each user can either star or unstar a post). Only when the user has selected a channel can they view their starred posts in this channel by navigating to the user icon on the top right and navigating to the ‘My Starred Posts’ button. Once that button is activated, a dialog will pop up displaying the user’s starred posts in that channel. The user can see the reaction counts (and react to) their starred posts as well as unstar them, which will also update the corresponding post in the channel as well. We extended our existing framework for post concurrency to build this extension.

Concurrency

We manage concurrency in two main ways: disabling parts of the user interface to prevent users from creating race conditions in the application, and managing out-of-order events received by the model.

User Interface Concurrency

When the Adapter begins working on an event sent from the View, it immediately tells the View which parts of the state (posts, workspaces, channels, or the logged-in user) it will change (or “are loading”). Components in the View can listen for when the state is set to loading. If a component relies on a piece of state that is currently loading, or if interacting with a component would lead to the submission of another event that might cause a race condition, it will disable itself. We define a race condition here as an action that the user performs that affects their previous actions. For example, suppose the View dispatches an event to log out the user. The Adapter will then tell the view that the user is loading. If the user tries to make another post while the log out request is processing, then different things will happen based on when the requests were made. So, the post editor disables itself while the user is loading: the user cannot submit a post if they are in the process of logging out. When the Adapter is done processing that event, it tells the view that all of the state is done loading, and disabled components will un-disable themselves. In this way, the user cannot make simultaneous requests (the user cannot select another workspace until the initial request for selecting a workspace is completed, for example), making it impossible for the user to cause race conditions in our application.

Server Subscription Concurrency

When we receive post update events from the server, we have no guarantee about the order that the events were received with respect to the time that the post was created. This

leads to two issues: (1) we receive a post before we receive its parent, and (2) we receive a post that is created before already received posts that were created later. Our adapter deals with this issue by creating a tree of posts in the adapter, where each post has an ordered list of other posts that are replies to it.

First, whenever it receives a post, it checks if its parent post has been received. If the parent post has been received, then the adapter inserts the post into the parent's children as a reply. Otherwise, we choose not to display this post, because its parent post has not yet been received. As a result, we store this post and keep track of the name of its parent post. The alternative is that we could display a filler post as its parent, but we decided not to implement this approach because if we receive a post with a malformed parent name (such as a parent name that never exists), we would never want to display that post if its parent post is never received. Once its parent post has been received, we then can add this post as well as all of this post's replies, which depend on this post having been received.

Then, once we know that the parent post has been received, the adapter decides where to insert that child post in the parent's children array by performing a linear scan over the array and deciding where that post should be inserted based on its creation timestamp.

Finally, our adapter notifies the view with four pieces of information: (1) the content of the post to be inserted, (2) the name of the post's parent, (3) the index at which the post should be inserted, and (4) whether the post should be inserted or modified.

Architectural Diagram Description

We follow a model, view, adapter design pattern. Our adapter is decoupled into four adapters, one for handling login/logout, workspaces, channels, and posts. We additionally have a state manager that our adapter uses to store information about the user interface.

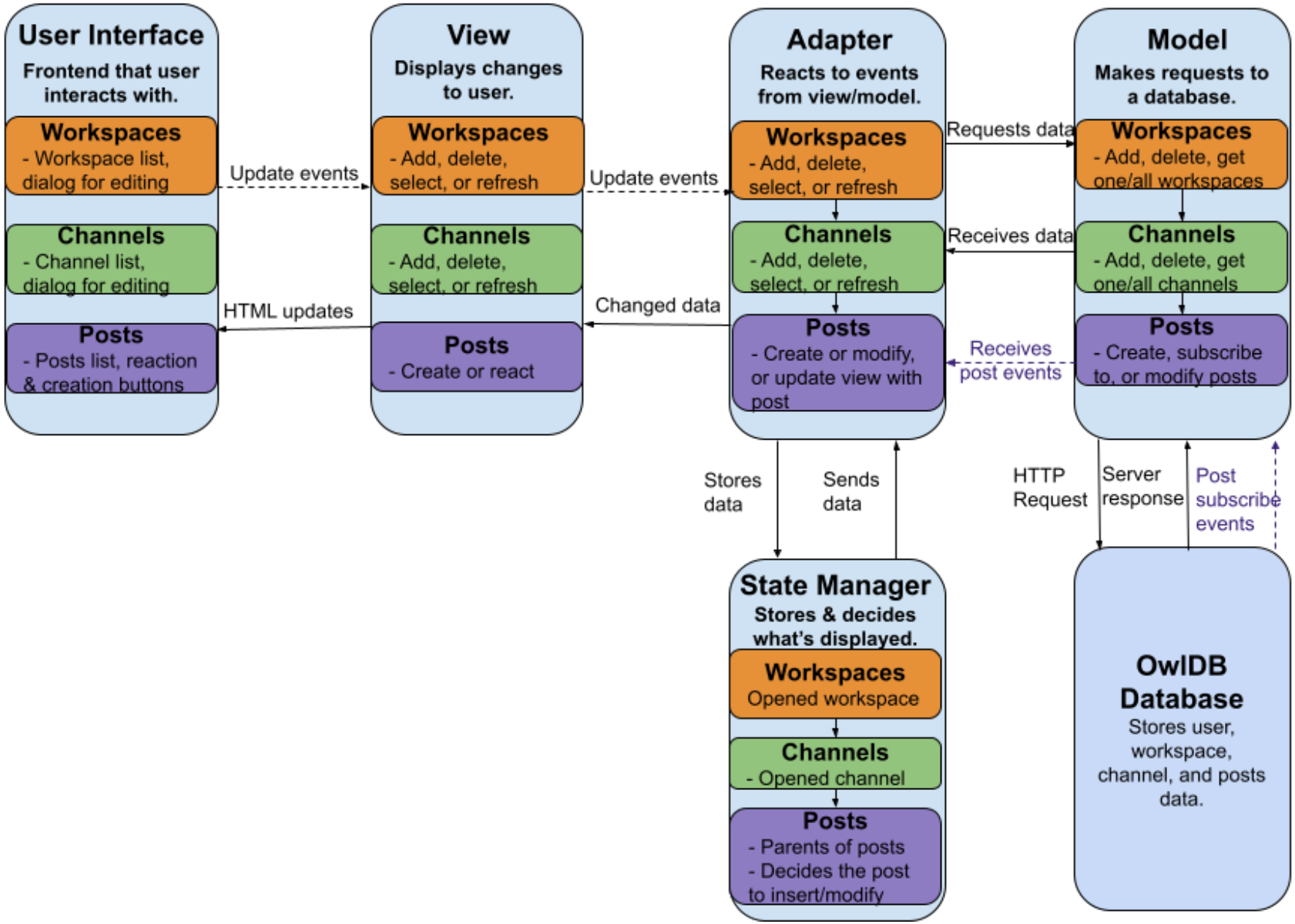
The user interface is built in a modular way using web components. Web components receive updates from the view by acting as listeners.

The view operates by maintaining lists of each type of listener. Specifically, the view has posts, channels, workspaces, login, and loading listeners, each of which receives updates for the respective data. When functions are called on the view to display information, it delegates to all the web components that listen for that update.

The adapter listens for events from both the view and the model and updates both accordingly. Our adapter is split into four decoupled adapters: one for handling authentication, one for workspace updates, one for channel updates, and one for post updates. Each adapter listens for events from the view, makes a request for information from the model based on that event, and updates the view based on the response.

The state manager stores the state of the user interface by the direction of the adapter. The adapter requests the state manager to store the state of the user interface, which makes requests to the model as needed to update the state for each of workspaces, channels, posts, and login/logout requests.

The model is modularized into different packages which handle HTTP requests to the database for workspace, channel, posts, and login/logout updates.


User Interface
Frontend that user interacts with.
Workspaces
- Workspace list, dialog for editing
Channels
- Channel list, dialog for editing
Posts
- Posts list, reaction & creation buttons
Update events
HTML updates
View
Displays changes to user.
Workspaces
- Add, delete, select, or refresh
Channels
- Add, delete, select, or refresh
Posts
- Create or react
Update events
Changed data
Adapter
Reacts to events from view/model.
Workspaces
- Add, delete, select, or refresh
Channels
- Add, delete, select, or refresh
Posts
- Create or modify, or update view with post
Requests data
Receives data
Receives post events
Model
Makes requests to a database.
Workspaces
- Add, delete, get one/all workspaces
Channels
- Add, delete, get one/all channels
Posts
- Create, subscribe to, or modify posts
Stores data
Sends data
HTTP Request
Server response
Post subscribe events
State Manager
Stores & decides what's displayed.
Workspaces
Opened workspace
Channels
- Opened channel
Posts
- Parents of posts
- Decides the post to insert/modify
OwlDB Database
Stores user, workspace, channel, and posts data.