Documentation PubHubs VotingWidgetPlugin

Manager:
Evgeniia Egorova
Engineers:

Ard van der Putten	Arvid Bonten
Chris Musteata	Emir Bosnak
Jules Porteners	Mees van Gaalen
Prisca Roddeman	Robert Blaauwendraad

June 7, 2024

This document is outdated due to refacturing, implementing
and moving code of these plugins as part of the core PubHubs code.

So see this as a reference to the original project.

Contents

1 Introduction

2 General

This report will handle the functionality and general overview of work methods, code quality, bugs, methodology and testing of the PubHubs Plugin: Voting Widget, designed by our team. As a short introduction, this plugin will consist of 2 parts: polls and schedulers. Polls are either, you can vote on every option if you agree or not, or, you can vote on one of the available options. Schedulers[1] are a tool where the person sending the scheduler can add dates to a 'poll' and everyone can respond with either a yes/no/maybe. This plugin is integrated into the PubHubs client front end using their plugin system. We have created all the front end code, the data structure of the poll/scheduler and added the necessary function calls to the Matrix protocol. The plugins are part of the message event structure of the Matrix event framework.

2.1 Bugs

Almost all bugs encountered during development have been fixed. There is one significant bug left which we didn't have time to fix in the last sprint (probably introduced during sprint 3). The bug concerns the scheduler creation, letting the user input multiple exact dates as options. This bug does not cause any functionality issues but it is something that should be avoided. The same behaviour for polls have not been tested but it is possible to enter options with the exact same strings.

2.2 Pitfalls

Biggest pitfall we encountered, which caused one of our main ideas to be unviable was the impossibility of sending server-side events. We had come up with a feature to set a due date for the schedulers so that they can be closed automatically when the due date has come. However, since with this scheme there are no users sending events to the Matrix event timeline the close event must have been sent by the server. We found out that this was impossible with the way the Matrix protocol was set up by the PubHubs team.

Another pitfall was the implementation of the notification system. The notification system is still in development by the PubHubs team so we couldn't create a detailed notification structure for closing the scheduler and picking a date. However, we coded in necessary event triggers so that the when the notification system is implemented the PubHubs team can easily integrate it to our plugin.

2.3 Changes we would like in PubHubs

Overall the code and the structure of the events system is pretty solid. The plugin system was easy to work with to add the necessary components. However, even for implementing a simple poll message, we had to modify a lot of the code not contained to the plugin system. This caused conflicts with the updates the PubHubs team introduced during development. The plugin system can be isolated further, making it easier for new teams to introduce their own plugins without messing up other parts of the codebase. Otherwise, implementing even a basic plugin would need a lot of communication with the main PubHubs team.

2.4 Feedback on PubHubs Code

Generally, the PubHubs code was clear and commented well where necessary. There were often implementation examples present somewhere in the code, and Vue components are split up nicely.

[1]This is similar to 'datumprikker'

For some components, customizability might be something to take a look at. They may have background colors that cannot be changed from the parent, or hardcoded sizes that make it unusable or ugly in different places. Some examples include the Avatar component, which was mainly used in the chat screen but it had a fixed size that made it difficult to work with in the poll vote percentage bar and the view votes screens. The popover component has a fixed background color that makes it looks strange when used as the settings popup of a scheduler. The icon component gave every icon a positive stroke width which made filled icons look fat.

These issues were initially a bit annoying to deal with, but as our familiarity with the codebase and Vue in general grew it was not that big of a deal.

2.5 TODOs

There are two main TODOs left for the PubHubs team which we were not really able to get done. The first TODO is about the PubHubs API functions. To get information about the Matrix event timeline in a specific chat room, we are currently creating the endpoint and creating an API object ourselves in the plugin code. However, this could be changed with the PubHubs' own API interfacing functions. The second significant TODO is about the ViTest setup. It is possible to move the test environment setup code to a separate file to manage the mocked timeline more efficiently. Apart from these significant TODOs, there are some work to be done regarding refactoring the front end components to minimize code duplication. This was touched upon on section 2.4 already.

3 Design

At the start of the project, the client sent us some basic designs that were already made by the PubHubs team. We wanted to spend some time developing new designs, so that the poll and scheduler could be as user friendly and visually appealing as possible.

In general, we created the desktop version designs first, and afterwards converted them to the corresponding mobile designs. We wanted the designs to be as similar as possible when comparing mobile to desktop, as this would create a more intuitive experience when using the product. Since the mobile screen is a lot smaller, some designs we created for desktop would not fit nicely. Therefore, a few designs were changed to fill the entire screen for mobile (instead of a smaller pop-up window for example).

All design choices were reviewed by team members (working on non-design tasks) and the client before implementation.

A design which was created but left unused due to implementation difficulties is the due date for the scheduler (found in the ”Create Menu Due Date” Figma entry). It is an addition to the creation screen, which allows a user to select a due date for the scheduler. If this due date passes, the scheduler automatically closes. The design was left in the Figma page in case there will be a use for it in the future.

All of the designs can be found in the Figma page.

4 Data structures

This part will focus on a technical documentation of how the data structures made for the voting widget are used and structured. This part thus also discusses how the state of a voting widget is tracked and updated.

The structure in listing 1 is the message content used in the custom message matrix event for the initial voting widget. In the component, these fields are used to fill out a VotingWidget object as defined in `hub-client/src/plugins/PluginVotingWidget/voting-widget.ts`

Listing 1: Voting Widget Message Event Content

```
export interface M_VotingWidgetMessageEventContent
extends M_BaseMessageEventContent {
        msgtype: 'pubhubs.voting_widget.widget';
        title: string;
        description?: string;
        location?: string;
        options: Array<PollOption | SchedulerOption>;
        type: VotingWidgetType;
        voting_type?: PollVotingType;
}
```

When a user votes on a voting widget, they send a custom non-message matrix event with content that is filled out in the format specified in listing 2. Note that `event_id` in `m.relates_to` is the event id of the initial voting widget, `optionId` is the id of the option the user has voted for, and `vote` is the choice the user has made for that option (values of which can be `"yes"` for Polls, and `"yes", "maybe", "no"` for Schedulers).

Listing 2: Vote Event Content

```
export interface M_VotingWidgetVote {
        //this is not a room message!
        msgtype: 'pubhubs.voting_widget.vote';
        'm.relates_to': {
                event_id: string;
                rel_type: string;
        };
        optionId: number;
        vote: string;
}
```

This vote made by the user is stored in the data structure specified in listing 3 on the client side when a user receives it. The voting widget updates it's internal state by watching the timeline for changes, and, if an event is in response to it, processing it appropriately. When a voting widget is initially loaded, it will use an API call to get all the events related to it at once so it does not have to receive them on the timeline. The data structure used is an array with elements for each option of the voting widget, and then for each of these options an array of objects for each of the possible choices that option has — `"yes", "redacted"` for Polls, and `"yes", "maybe", "no", "redacted"` for Schedulers — that stores the ids of users that voted on that option for that choice, and also the time at which they voted.

Listing 3: Votes State Array

```
interface votesForOption {
    optionId: number;
    votes: Array<{
        choice?: string;
        userIds: Array<string>;
        userTime: Array<[string, string]>;
    }>;
}

const votesByOption = ref(new Array<votesForOption>());
```

When the creator of a voting widget edits that voting widget, a custom non-message matrix event is sent with the structure defined in listing 4 as it's content. Note that this is more or less the same structure as the initial widget (listing 1), but with an `m.relates_to` field added in so we know what voting widget it is modifying. When the timeline updates and the watch of the original voting widget is triggered, it will update it's state by changing it's fields to the new ones in the edit event (if it was sent by the voting widget's creator). Previous votes are brought over to options that have the same title for Polls, and options with the exact same date for Schedulers.

Listing 4: Voting Widget Edit Event Content

```
export interface M_VotingWidgetEditEventContent {
        msgtype: 'pubhubs.voting_widget.edit';
        'm.relates_to': {
                event_id: string;
                rel_type: string;
        };
        title: string;
        description?: string;
        location?: string;
        options: Array<PollOption | SchedulerOption>;
        type: VotingWidgetType;
        voting_type?: PollVotingType;
}
```

When the creator of a voting widget (specifically, a Scheduler) decides to open or close it, a non-message custom matrix event is sent with content defined as in listings 5 and 6 respectively. The `m.mentions` field is used for the notification system the PubHubs team is going to implement, and contains the user ids of all users that have voted on the corresponding Scheduler.

Listing 5: Voting Widget Close Event Content

```
export interface M_VotingWidgetClose {
        msgtype: 'pubhubs.voting_widget.close';
        'm.relates_to': {
                event_id: string;
                rel_type: string;
        };
        'm.mentions': {
                user_ids: string[];
        };
}
```

Listing 6: Voting Widget Reopen Event Content

```
export interface M_VotingWidgetOpen {
        msgtype: 'pubhubs.voting_widget.open';
        'm.relates_to': {
                event_id: string;
                rel_type: string;
        };
        'm.mentions': {
                user_ids: string[];
        };
}
```

When the creator of a poll has closed said poll they can pick an option, which sends a non-message custom matrix event with content as specified by listing 7. the `optionId` field is the id of the option the creator has chosen as the picked option.

Listing 7: Voting Widget Pick Option Event Content

```
export interface M_VotingWidgetPickOption {
        msgtype: 'pubhubs.voting_widget.pick_option';
        'm.relates_to': {
                event_id: string;
                rel_type: string;
        };
        optionId: number;
}
```

4.1 Design choices

These are several choices and improvements that are important when the code will be modified and used in the future.

- Automatic scrolling to the bottom when a new message is received is disabled by ignoring message types related to the functionality of polls and schedulers. This is done such that if new kinds of messages are added for another plugin they should not affected.
- The `votes` field in `votesForOption` (listing 3) still has the field `userIds`, even though `userTime` contains tuples of both user ids and the time of the vote. `userIds` should be removed, and every use of it should be replaced by `userTime`. We did not have time to change this, since it was relatively low priority as it did not impact functionality.
- There is one message type that contains the information of any type of edit (`M_VotingWidgetEditEventContent`, listing 4), instead of one for every different kind of edit (title, description, options). The reason for this is that the edits are created through the same menu, so they are sent at the same moment. Separating them would spam too many messages.
- Aggregation of votes and other replies is both done by local aggregation and by matrix API calls. When switching to a room the replies to a poll/scheduler are gathered by an API call, instead of looping over the entire timeline. When the user is inside a room the aggregation is done locally by watching changes in the timeline, to avoid spamming API calls and straining the system.
- For ranking the dates of a scheduler the football scoring system was used because the 'maybe' option represents the half way point of the 'yes' and 'no' options.

5 Rendering

In this section, we discuss the rendering of the plugin and its code structure, since they are closely intertwined. Specific choices made during development are referenced in the design choice list, where further explanations can be found.

5.1 Code structure

To implement the voting widget feature, we introduced a new message type and corresponding plugin 1. Below is a detailed breakdown of the code structure and components involved in this implementation.

5.1.1 Message Type

A new message type, referred to as a 'voting_widget' 2 , was created. This type of message required a unique rendering approach, which necessitated the development of a new plugin 1. Additionally, modifications were needed in the message input area to enable the selection and sending of this new message type (e.g., text, file, poll, or scheduler).

5.1.2 Plugin (plugin.ts)

The plugin initialization starts with a 'plugin.ts' file. This file defines essential properties such as:

- Plugin name
- Associated message type
- Component to use for rendering (in this case, 'PluginVotingWidget.vue')

5.1.3 Voting Widget Class (voting-widget.ts)

To manage both poll and scheduler functionalities, we created a base class 3. This class includes common fields like:

- Title
- Description
- Maximum number of options

5.1.4 Poll Class (poll.ts)

The poll feature extends the base voting widget class 3. Key components include:

- An enumerator 4 for the voting type.
- An interface 5 for 'PollOption', allowing us to create an array of options ('PollOption[]').
- Specific helper functions for the poll logic.

5.1.5 Scheduler Class (scheduler.ts)

Similar to the poll, the scheduler also extends the voting widget class. It includes additional properties:

- Location field.
- An interface for 'SchedulerOption' 5, structured differently from 'PollOption' to handle arrays of dates and statuses, supported by an enumerator 6.
- Specific helper functions for the scheduler logic.

5.1.6 Voting Widget Component (PluginVotingWidget.vue)

This file serves as the core component for rendering. It handles the common elements shared by polls and schedulers, such as titles and descriptions. Functions used by both widgets are defined here to avoid code duplication. Vue's component structure is employed to differentiate how poll and scheduler options are rendered, utilizing distinct components and subcomponents for each.

5.1.7 Option Lists (PollOptionList.vue & SchedulerOptionList.vue)

To manage and render poll/scheduler options, we use separate components. These components iterate over their respective options using a for loop, and include functions applicable to the entire list. This separation helps maintain clean and organized front-end code.

5.1.8 Option (PollOption.vue & SchedulerOption.vue)

These files are primarily templates filled with their respective details. They contain minimal logic and are focused on front-end rendering, leveraging Vue's for loops to maintain clean code and enhance readability. By structuring the code in this manner, we achieve a modular, maintainable, and scalable implementation for the voting widget feature, ensuring that shared functionality is centralized and specific details are encapsulated within dedicated components.

5.1.9 Message Input (MessageInput.vue)

The 'MessageInput.vue' file, originally created by the PubHubs team, was modified to enable the sending of polls and schedulers. We added buttons and functions to accommodate the new message types, similar to the existing functionalities. Significant refactoring was required to support editing voting widgets, leading to the implementation of a store 7.

5.1.10 Message Input Store (messageInput.ts)

A store was created to manage the state of the message input menus. This store includes functions for closing all menus and opening the basic text area input. It also holds 'pollObject' and 'schedulerObject' instances, allowing the message input to display and edit data for polls and schedulers.

5.1.11 Poll Message Input (PollMessageInput.vue)

A new message input screen was developed for sending polls, featuring fields specific to poll creation. It includes functions to update the poll object in the main message input file and to dynamically manage the input, such as adding a new option field when all current fields are filled.

5.1.12 Scheduler Message Input (SchedulerMessageInput.vue)

The scheduler message input operates similarly to the poll message input but includes additional features such as a location field and a date picker 8. The options have different states based on whether they are empty, being filled, or already filled.

5.2 Design choices

1. The PubHubs team is creating a plugin system so server hosts can add or remove functionalities according to their needs. This project was also meant to test whether it would be possible to implement all of the functionality within this plugin system. We did find we needed to modify *some* of the code outside of the system so we could not keep it completely contained.

2. We decided to make a single plugin containing the functionality for both polls and schedulers. That is because the two share a lot of functions and their HTML structure is also very similar. Where they did differ we branched with case distinctions, and if necessary, different Vue components.

3. We created a class for voting widgets to ensure a consistent data structure, and to similarly to point 2 avoid duplicate functions. We also made classes for polls and schedulers specifically, for consistency in accessing these classes.

4. We are using 2 different poll voting styles, stored in an enum. When the type is checkbox, multiple answers can be selected simultaneously. For the radio type, only one option can be voted on, and voting on something else cancels the previous vote.

5. We use an interface for PollOption because we do not require a separate class for new functions, but we want to create an array containing multiple instances of them.

6. When creating a scheduler, the options in the list have 3 different styles, depending on whether a date/time has been fully chosen (filled), one is being chosen in the screen on the right (filling) or it has not been clicked yet to create a new option (empty). We store these three states in an enum.
7. Vue can only share data between parent and child components. When editing a voting widget, the data from the widget needs to be shared with the message input box. To do this, we needed to use a store
8. We installed the vue-datepicker package, as it is well maintained and nicely customizable for use in the creation of schedulers. We figured creating one ourselves would have been out of scope, since a lot needs to be taken into consideration when creating a calendar viewer.

6 Testing

We have used two different testing techniques to test two different types of tests. Namely, unit tests and integration tests. The integration tests were asked for by the course (Software Engineering), while the unit tests (as long as we did not spend too much time on them, as this is a frontend application) was asked for by the iHub team.

The unit tests were created in ViTest, this testing tool is able to mock certain features of the components in an application s.t. it can run the code without connecting to the server.

In order to ensure that our code coverage is as high as possible and our features worked correctly, we also opted to implement integration testing, as previously mentioned. These tests were initially designed manually and later converted to run automatically. For automation we decided to use Playwright[2]. This tool is designed for end-to-end/integration testing and is similar to Selenium or Cypress.

6.1 ViTest

The ViTest testcases were structured in the following way:

1. Mock the data a component needs.
2. Initialize the component by wrapping it.
3. Call the function to be tested
4. Check if the result is equal to the expected output.

We have used branch coverage where possible. However, for certain cases, like the CollectNewVotesInTimeline and the API version of this function were not able to be tested with our current knowledge of the code. This is due to the problem with mocking the read-only variable room.currentRoom. These functions which are left take the current rooms' latest events and then update the widget accordingly. As the functionality of the components of these functions has been tested, we expect that if tests for these are still possible, these would not raise any new mistakes in the code.

6.1.1 Explanation mounted components

DISCLAIMER: This is a technical explanation on how to mount components in ViTest, it is not required to read this section to understand the voting widget.

[2]Available at https://playwright.dev/

To mock the component pluginVotingWidget, we had to figure out through numerous bad error messages what has gone wrong with the test cases. This is due to poor documentation from ViTest and poor cooperation with Vue components and matrix protocols. The way we eventually got this working combines a mock of the access token, the translation key, mounting the component of the file and setting the required fields both during the mount and after.

The next example uses the code from the unit test for the function removeRedactedVotes. This function removed all the redacted votes from the field 'Redacted' in the 'VotesByOption' interface. To test this function we first do the general steps shown above, we mock the access token for every test, so this is moved to the function beforeEach as seen in listing 8.

Listing 8: Before Each Code

```
beforeEach(() =>{
    setActivePinia(createPinia());
    //mock access token before each test
    const pubhubs = usePubHubs();
    const accessTokenMock = vi.fn();
    pubhubs.Auth.getAccessToken = accessTokenMock;
});
```

Then, we create the 'votesByOption' field s.t. when we put this in the component, it has all the props necessary so that there are no errors when mounting. After this, it should be possible to mount the component and call the function to be tested with the prefix:"wrapper.vm.", which can be done as in the following listing (9).

Listing 9: PluginVotingWidget Wrapper

```
let wrapper = mount(PluginVotingWidget, {props: data}
```

Where we put all the data required by the mount/function to be tested in the props field. [3]

Now, we can check the output of this function or the side-effects of the function to see if it is working correctly.

6.2 PlayWright

DISCLAIMER: The iHub team can ignore this section.

At first we followed the outline presented in the course, regarding the requirements of testing. This means that we started integration testing by first designing systematic test scenarios that can be easily replicable. They are still available on the Confluence page designated for testing, which can be accessed here. Each test scenario is based on a user story, as per the product backlog. If needed, multiple test cases were added for each scenario, in order to cover as much of the functionality as possible. For example, in the case of voting, we treat the cases of single votes and multiple votes independently.

Each test scenario was then converted into its own Playwright counterpart. In places where a user can make arbitrary choices (e.g. pick which option to vote), the test framework picks a random option out of the ones available. This randomness is set in place in order to ensure robustness of each feature and catch any unexpected behaviour.

Note: The PubHubs client requires login by QR code on first access. As Playwright uses an incognito browser by default, each run would require logging in again. We bypass this by recording the browser local storage and launching it with these settings every time. This way, each test

[3]see test/plugin/pluginVotingWidget.test for more info

runs in an already logged in browser. To set this up, the code can be run with the parameter `update_token` set to `True`. This gives the user 30 seconds to log into the client and have his session storage saved.