Zishan Ahmad
zishan.barun@gmail.com
@zishan.ahmad:open.rocket.chat / Zishan Ahmad

Proposal: Embedded Chat 2024, A Focus on UI Enhancement, Security and Configuration

Google Summer of Code 2024

GENERAL DETAILS

Name: Zishan Ahmad

University: Indian Institute of Information Technology, Kottayam

Primary Email: zishan.barun@gmail.com

Secondary Email: zishanahmad20bcs78@iiitkottayam.ac.in

Rocket.Chat ID: @zishan.ahmad:open.rocket.chat

GitHub: Spiral-Memory

LinkedIn: Zishan Ahmad

Website: https://spiral-memory.netlify.app/

Country: India

Time Zone: India (GMT+5:30)

Size of the Project: Medium (175 Hours)

MENTOR

Sidharth Mohanty (@sidmohanty11)

ABSTRACT

The goal of this project is to enhance the overall UI by making components modular and providing pre-built themes. In addition, the aim is to provide security via cookie-based authentication by using RC-app as a bridge between Embedded Chat and the Rocket.Chat REST API and improve its configuration capabilities by allowing the instance to be configurable through RC-app, and much more.

BENEFITS TO COMMUNITY

Rocket.Chat, founded in 2016, has served 15 million users across 150 countries. It is one of the largest chat platforms that is open source. One of its projects, which can be integrated into any website as an in-chat app, is "Embedded Chat" which originated from an idea presented by the project mentor to address the iframe issue and provide easy embedding within any website. Its first commit was on 2 June 2022, and it has since helped several Rocket.Chat users in their businesses.

Embedded Chat provides all the required features, but in this modern era, where over 42% of users seek live chat and over 69% are inclined to use it for self-service, such an application enhances a website's appeal, presenting the brand as modern, attentive, and customer centric. There is a need for a cohesive user interface that can match the look and feel of the website it is integrated into, which it currently lacks. In addition, security improvements are needed, along with making this app easily customizable through the Rocket.Chat workspace.

The plan is to separate the styling of each component from its logic, making it headless in nature. Once it is in this shape, the styling of each component can be controlled externally, along with providing some pre-configured themes for easy setup. For security, we can use HTTP-only cookie-based authentication by having an Embedded Chat RC-app act as a bridge between the communication of EC client and Rocket.Chat REST APIs, making it safe from any XSS attacks. Along with these measures, the plan is to make EC as configurable as possible directly from the Rocket.Chat workspace.

These changes will help Embedded Chat to grow exponentially, as users will be able to customize everything with ease and feel safe with the security measures implemented.

This will make EC the best choice for in-app chat, overall improving customizability, security, and provide an incredibly valuable experience for both admins and their users.

GOALS

Throughout the course of GSoC my primary focus would be on:

1. Enhancing the UI components and styles to create a more appealing and user-friendly experience.
2. Designing themes, bringing them to life by coding, and providing multiple pre-configured templates to users.
3. Improving the security of the application by securing it against XSS attacks.
4. Making the Embedded Chat as configurable as possible through Rocket.Chat workspace and in-app UI editor.
5. Improving upon my previous work on the UI-Kit addition.
6. Documenting the work thoroughly and providing user-friendly documentation to use available features without hassle.

DELIVERABLES

By the end of GSoC, I plan to deliver an improved version of the Embedded Chat with different themes, improved security, and customizability directly or through RC-workspace.

1. Improving security by implementing cookie-based authentication using RC-app of Embedded Chat by making it act as a bridge between the EC client and the Rocket.Chat REST API. **[NEW]**
2. Offering admins, the flexibility to easily adjust all Embedded chat settings within the Rocket.Chat Workspace. **[NEW]**
3. Providing an editor pane and drag-and-drop dynamic layout editor for the admin to configure the UI of the Embedded Chat according to their requirements without writing any code. **[NEW]**
4. Improving the UI-Kit integration further by adding contextual bar support in the Embedded Chat and ensuring that everything functions based on user actions on both modal and contextual bar. For example: Performing actions upon submission, when a new option is selected in RC-app which updates the modal in the Rocket.Chat, the same should occur in Embedded Chat as well. **[ENHANCEMENT]**

5. Implementing a headless architecture for the Embedded chat by decoupling the styling and logic of components. This will be achieved by separating the CSS styling modules or inline Emotion CSS into an Emotion styling module, such as "component.style.js," to make styling more uniform and decoupled throughout the app. Additionally, other functional logics can be separated as hooks or utilities to further decouple them. **[NEW]**
6. Using this headless architecture as a foundation to develop different pre-configured themes to make Embedded Chat more appealing and user-friendly, for easy setup. **[NEW]**

EXTRAS

The following are the features that I plan to build during the GSoC period only if I complete my deliverables before the scheduled plan; otherwise, I will try to build these afterward.

1. Adding i18n files to provide support for different languages. **[NEW]**
2. Adding ability to extract theming colors from the page it is Embedded into and add those colors to the embedded chat to match the look and feel of the website **[NEW]**
3. Flexibility for admins to inject their custom CSS through the Rocket.Chat workspace into different components based on their needs. **[NEW]**
4. Adding a feature to save messages as drafts, allowing users to send it later at their convenience. In case of an interrupted internet connection, the message will automatically be sent once the connection is restored. **[NEW]**
5. Adding a feature for automatic deployment whenever someone raises a pull request allowing the maintainers to test the changes directly without the need to set it up locally, in addition to conducting code reviews. **[NEW]**

IMPLEMENTATION DETAILS

1. Shifting to headless architecture – Currently, the styles are not uniformly applied throughout the app. Some styles are applied using CSS modules, some are applied as inline CSS using the `style` prop, and others are applied through inline Emotion styling. The plan is to separate the styles into a `component.style.js` file while still utilizing Emotion styling, but not in an inline manner.

Knowing that Emotion styling has less priority than inline styles, we can define our base CSS as Emotion styles in a separate Emotion style file. Then, we can pass our styles as props through themes to override this. In addition, when using the component, apart from the theme, we can utilize `useComponentOverrides` to apply the custom styles which is already written. Apart from styling, the plan is to separate component logics as hooks and utils to keep files manageable, modular, and flexible.

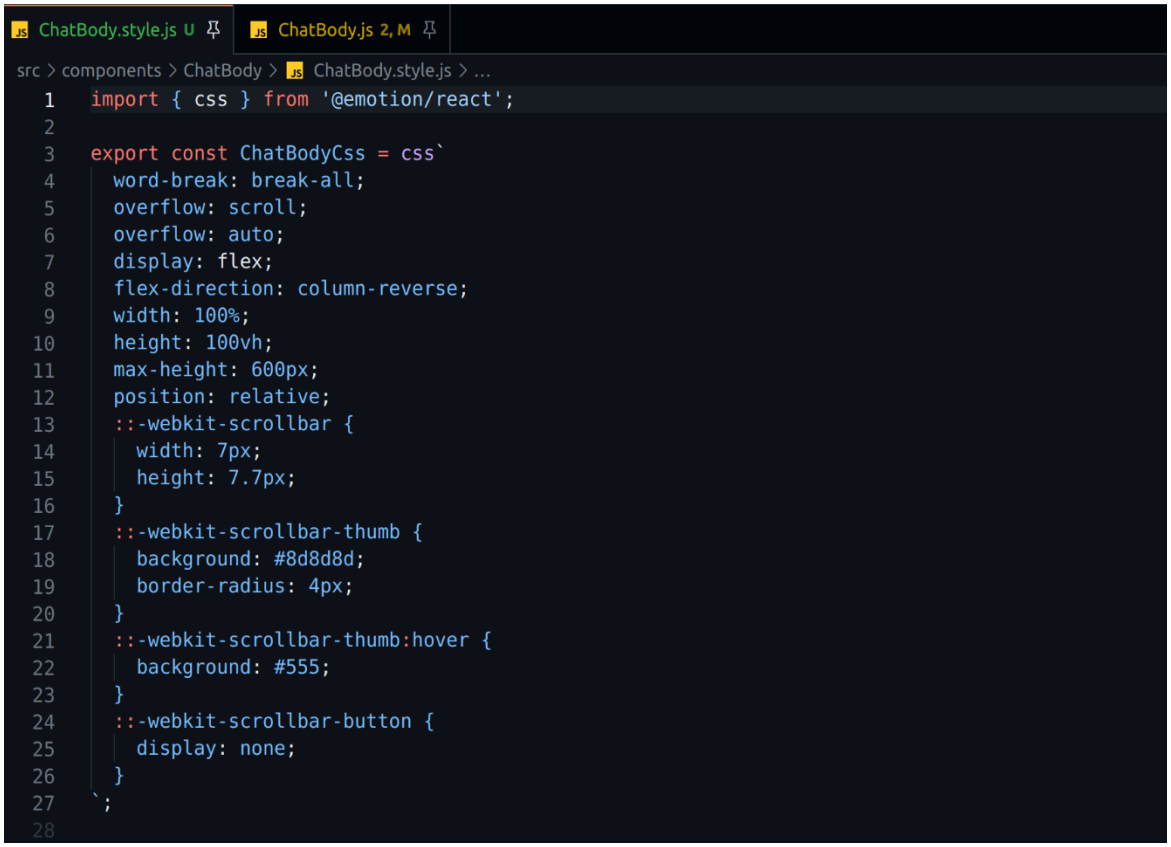

Fig. 1: Separating the stylings into `component.style.js`

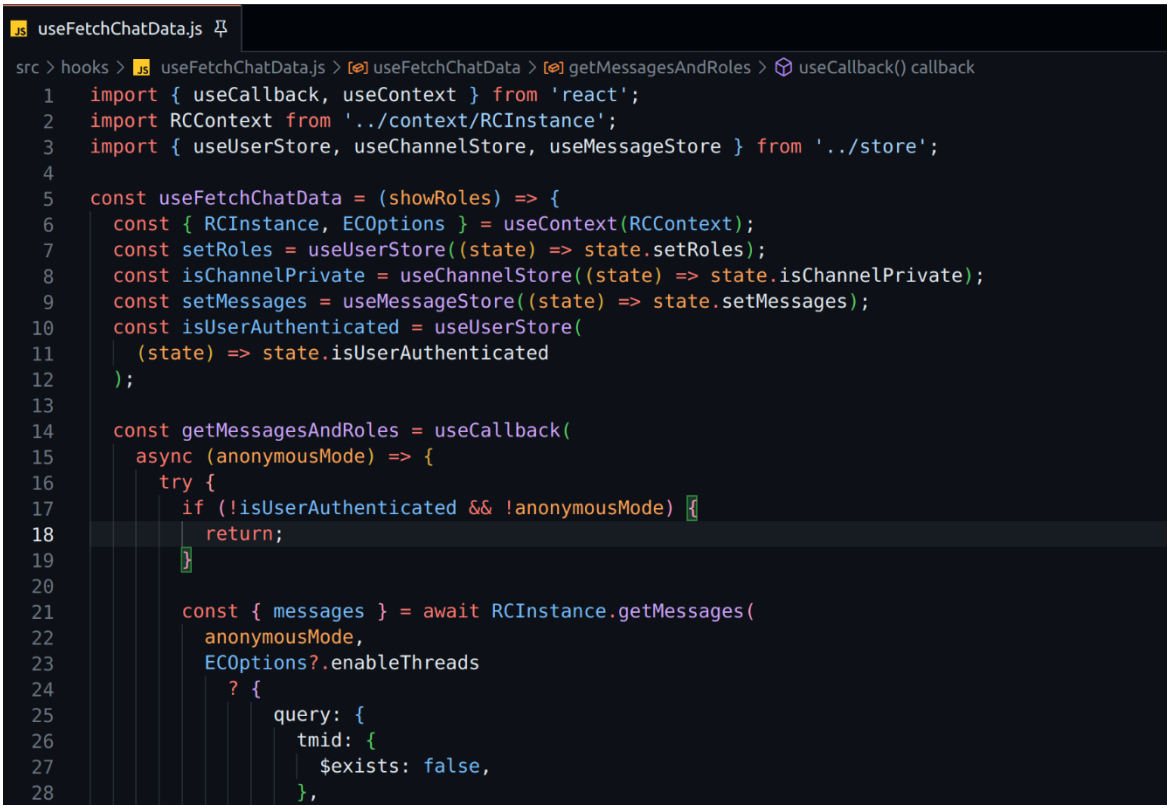

Fig. 2: An example of extracting logic as hook for messaging as a part of my existing PR

2. **Preconfigured themes with flexibility to customize –** Embedded Chat does provide all the required features but lacks in its styling. So, to overcome this, the plan is to use the headless architecture mentioned above and a theming foundation to create multiple beautiful and appealing themes. The option to choose from different themes will also be provided. When the user is an admin, they can select from multiple themes.

   To implement designs, followings steps can be taken:

   i. Stylings must be separated as mentioned above.

   ii. The existing theming foundation can be used to implement such designs. In the code, there is already a theming system set up where the styles can be passed.

   iii. `useComponentOverrides` attaches those passed styles to override the CSS and to pass general stylings, `useTheme` can be used.

   iv. The plan is to use the existing setup with styling separation to have different designs.

```
const CustomTheme = {
  breakpoints: {
    xs: 0,
    sm: 600,
    md: 900,
    lg: 1200,
    xl: 1536,
  },
  components: {
    ChatBody: {
      styleOverrides: {
        border: 'none',
      },
    },
    ChatInput: {
      styleOverrides: {
        fontWeight: 400,
        color: 'gray',
        borderRadius: '10px'      You, 7 seconds ago • Uncommitted changes
      },
    },
    Message: {
      classNames: 'myCustomClass',
    },
  },
```

Fig. 3: An example of defining custom theme

I have created some simple designs on Figma. Although I'm not very skilled in design, these designs are primarily for demonstration purposes. The actual designs can be finalized after discussion with the mentor. Below are some images of a simple Figma designs of the chat interface:

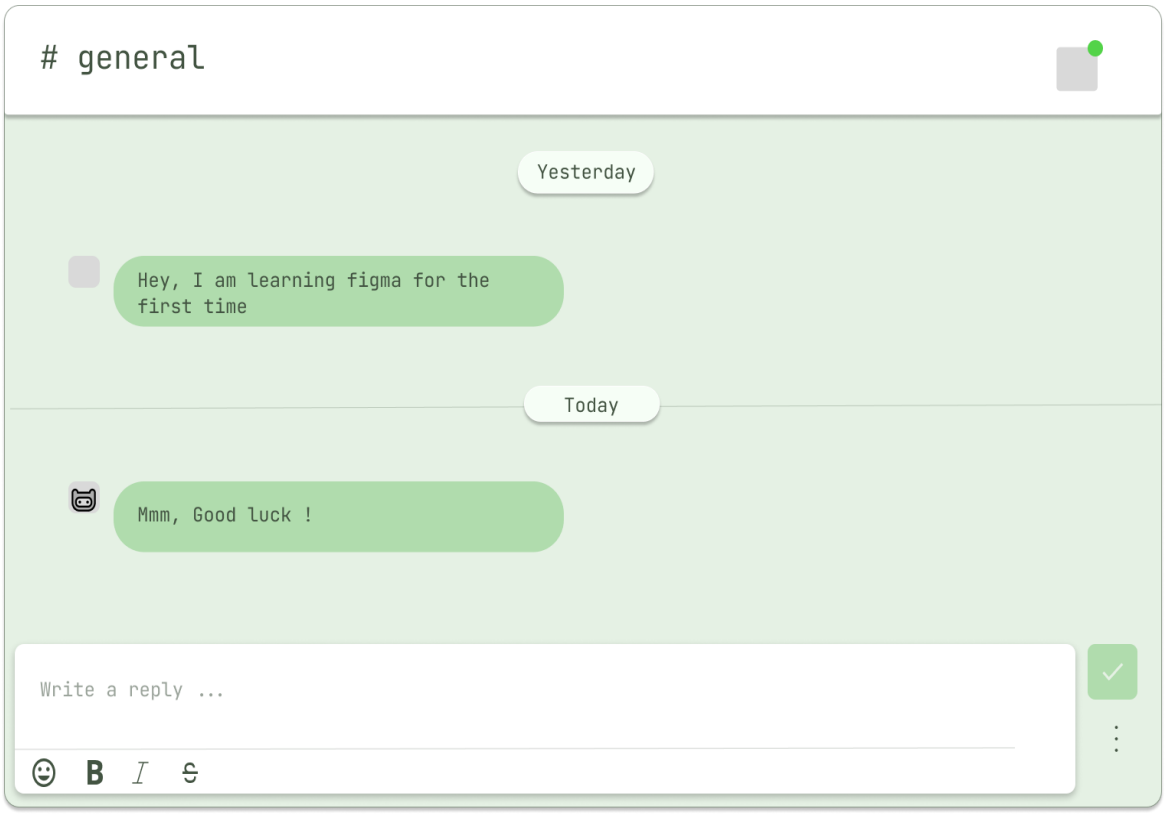

Fig. 4: The Beauty of Minimalist Discourse 😂

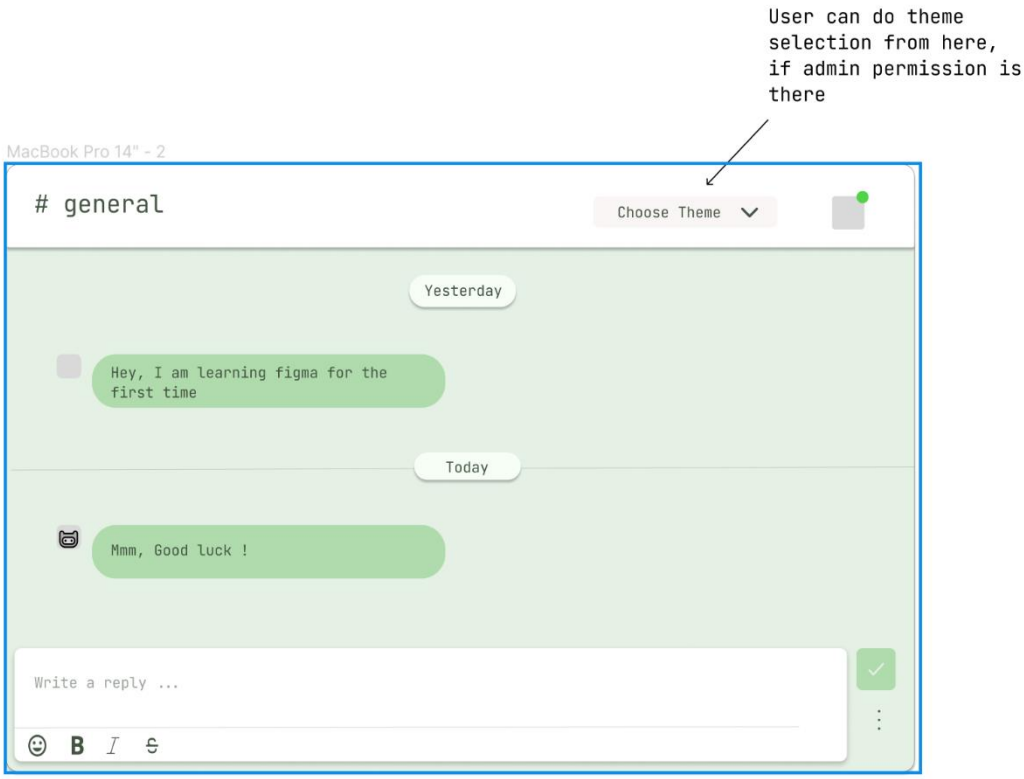

Fig. 5: Admin selecting theme from `Choose Theme dropdown`

3. **Editor Pane and Dynamic Layout Editor –** Currently, in embedded chat, the only way to customize the UI to match the needs is to make changes to the code, which might not be suitable for all admins, as it requires a sound technical understanding and can be cumbersome. Therefore, I'm proposing to add an editor pane and layout editor directly into the embedded chat itself where one can customize the UI according to their needs without writing a single bit of code.

   Addition 1: Editor Pane – The plan is to make components customizable through an editor pane for as many components as possible. The admin will enter an editor mode, select the component they want to customize, and then an editor pane will open where they can change different properties of that element directly through the editor. This will make embedded chat the best choice for in-app chat, providing such flexibility to customize. A basic prototyped version of the same can be seen in the commit: "added font related dynamic feat."

   Currently, all the properties that are editable are states and when admin changes the properties from the editor pane, those states get updated to reflect the changes in UI, but this won't be persistent over reloads. So, the approach will be to have all the updated states saved in a file on Embedded Chat, and these states will be loaded

from the settings first. Some of the code snippets, showing the current implementation is as follows:

```
const Editor = ({ onClose, setFontSize, color, setColor, setFontWeight }) => {
  const handleFontSizeChange = (event) => {
    const value = event.target.value.trim();
    if (value === '') {      You, 5 days ago • added font related dynamic feat
      setFontSize('26px');
    } else {
      setFontSize(`${value}px`);
    }
  };

  const handleFontWeightChange = (event) => {
    setFontWeight(event.target.value);
  };

  const [isSketchPicker, setIsSketchPicker] = useState(false);

  const pickerStyle = {
    default: {
      picker: {
        position: 'absolute',
        top: '220px',
        left: '80px',
      },
    },
  };
```

Fig. 6: Functions handling states of different properties through editor

```
        <Box
          className="FontColor"
          style={{
            display: 'flex',
            alignItems: 'center',
            justifyContent: 'space-between',
          }}
        >
          <span style={{ fontSize: '16px' }}>Font Color</span>
          <span style={{ fontSize: '16px' }}> Choose Color --{'>'} </span>
          <ActionButton
            square
            ghost
            onClick={() => setIsSketchPicker((prev) => !prev)}
          >
            <Icon name="brush" size="4rem" />
          </ActionButton>

          {isSketchPicker && (
            <SketchPicker
              styles={pickerStyle}
              color={color}
              onChange={(updatedColor) => setColor(updatedColor.hex)}
            />
          )}
        </Box>
      </Box>
    </Sidebar>
```

Fig. 7: Rendering of color picker in the editor pane

For demonstration purposes, I have shown how to enable/disable editor mode through props. However, in the actual implementation, this will not be the case. Instead, there will be a button that only admins (or those with permission to edit the UI) can use to toggle the edit mode.

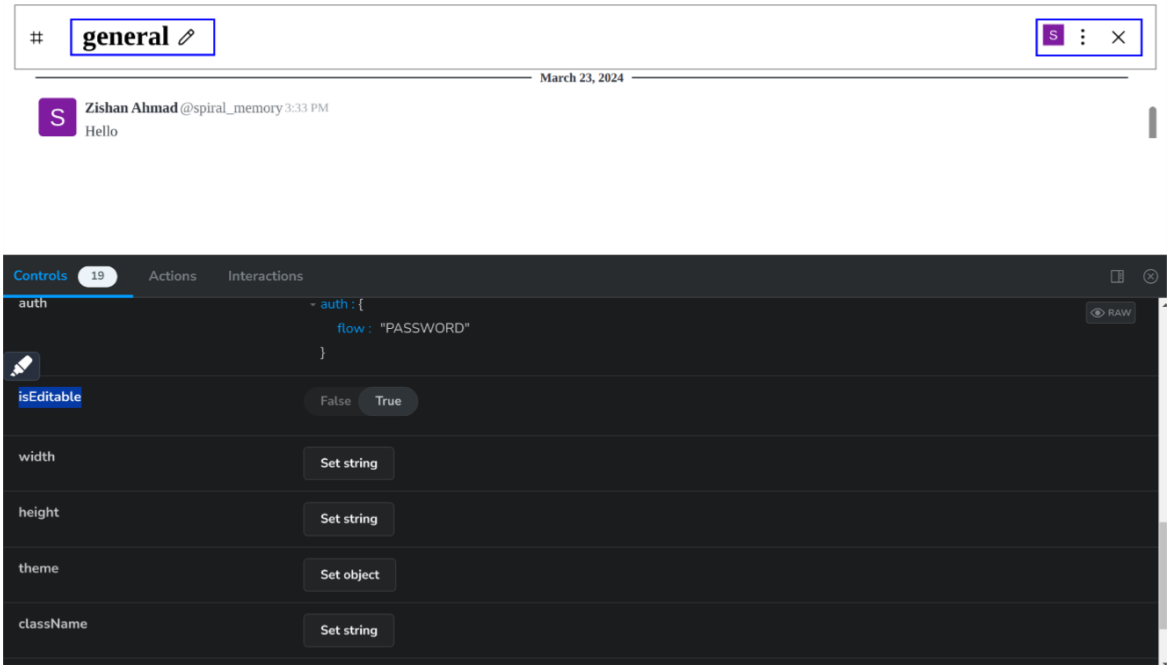

Fig. 8: Admin customizing the header chat header title styles.

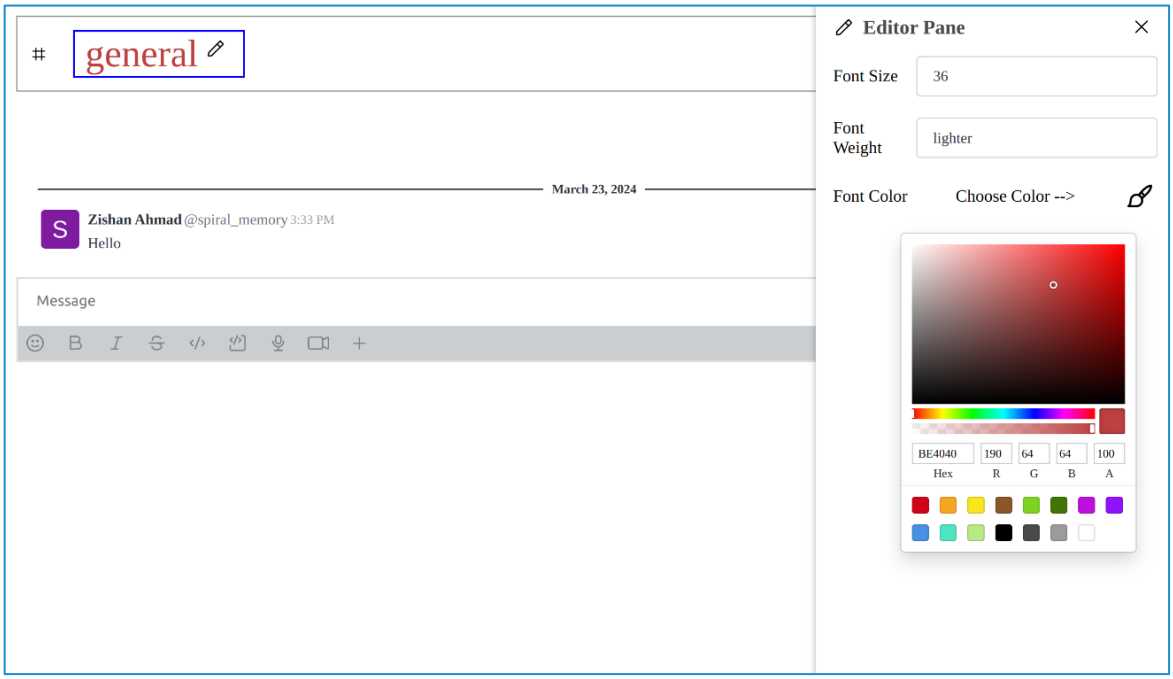

Fig. 9: Admin customizing the header chat header title styles.

Addition 2: Layout Editor – Ever imagined the ability to drag and drop the components to adjust their positions and then save the changes without compromising any functionality, all with zero code? This can be achieved with some great libraries such as [dnd kit.]

To implement a basic demo for such a thing, I have taken the followings steps:

i. Added a Drag and Drop (DnD) context listener to the box element where I wanted to make things draggable and droppable.

ii. Utilized the Sortable Context provided by DnD Kit and passed the items that I wanted to make sortable.

iii. Mapped through all the items and passed them through a custom React functional component that wraps the items with required styles and attaches attributes and listeners as required, to make them sortable as mentioned in the documentation.

iv. Implemented a function called `handleDragEnd` to let the DnD Kit know how to behave when a component is dragged onto another component.

v. Currently, I am saving the changes to localStorage to make them persistent over reloads. In the actual implementation, the plan is to save to a settings file as mentioned above.

Some of the code snippets, showing the current implementation is as follows:

```
import { CSS } from '@dnd-kit/utilities';

import { Box } from '../Box';
import { Menu } from '../Menu';
import { ActionButton } from '../ActionButton';
import { Icon } from '../Icon';

const STORAGE_KEY = 'sortableBoxItems';

export const SortableItem = ({ id, children }) => {
  const { attributes, listeners, setNodeRef, transform, transition } =
    useSortable({
      id,
    });

  const style = {
    transition,
    transform: CSS.Transform.toString(transform),
    cursor: 'grab',
  };
  return (
    <div ref={setNodeRef} style={style} {...attributes} {...listeners}>
      {children}
    </div>
  );      You, 4 days ago • Implemented sortable menu
};
```

Fig. 10: React functional component to attach required styles and listeners.

```
export const SortableBox = ({
  isEditable,
  avatarUrl,
  menuOptions,
  isClosable,
  setClosableState,
}) => {
  const initialItems = ['avatar', 'menu', 'closable'];
  const [items, setItems] = useState(initialItems);

  useEffect(() => {
    const storedItems = localStorage.getItem(STORAGE_KEY);
    if (storedItems) {
      setItems(JSON.parse(storedItems));
    }
  }, []);      You, 1 second ago • Uncommitted changes

  const handleDragEnd = (event) => {
    const { active, over } = event;

    if (active.id !== over.id) {
      const oldIndex = items.indexOf(active.id);
      const newIndex = items.indexOf(over.id);
      const newItems = arrayMove(items, oldIndex, newIndex);
      setItems(newItems);
      localStorage.setItem(STORAGE_KEY, JSON.stringify(newItems)); // Save items to loca
    }
  };
```

Fig. 11: Implementation of `handleDragEnd` and temporary storage mechanism

```
  return (
    <DndContext collisionDetection={closestCorners} onDragEnd={handleDragEnd}>
      <SortableContext items={items} strategy={horizontalListSortingStrategy}>
        <Box
          css={css`
            display: flex;
            align-items: center;
            gap: 0.125em;
            padding: 0 5px;
            border: ${isEditable ? '2px solid blue' : 'none'};
          `}
        >
          {items.map((itemId) => (
            <SortableItem key={itemId} id={itemId}>
              {itemId === 'avatar' && avatarUrl && (
                <img width="20px" height="20px" src={avatarUrl} alt="avatar" />
              )}
              {itemId === 'menu' && <Menu options={menuOptions} />}
              {itemId === 'closable' && isClosable && (
                <ActionButton
                  onClick={() => {
                    setClosableState((prev) => !prev);
                  }}
                  ghost
                  display="inline"
                  square
                  size="medium"
                >
```

Fig. 12: Attached Context listener and rendering of the component

A basic implementation of the same can be seen in the commit: [“implemented sortable menu”]

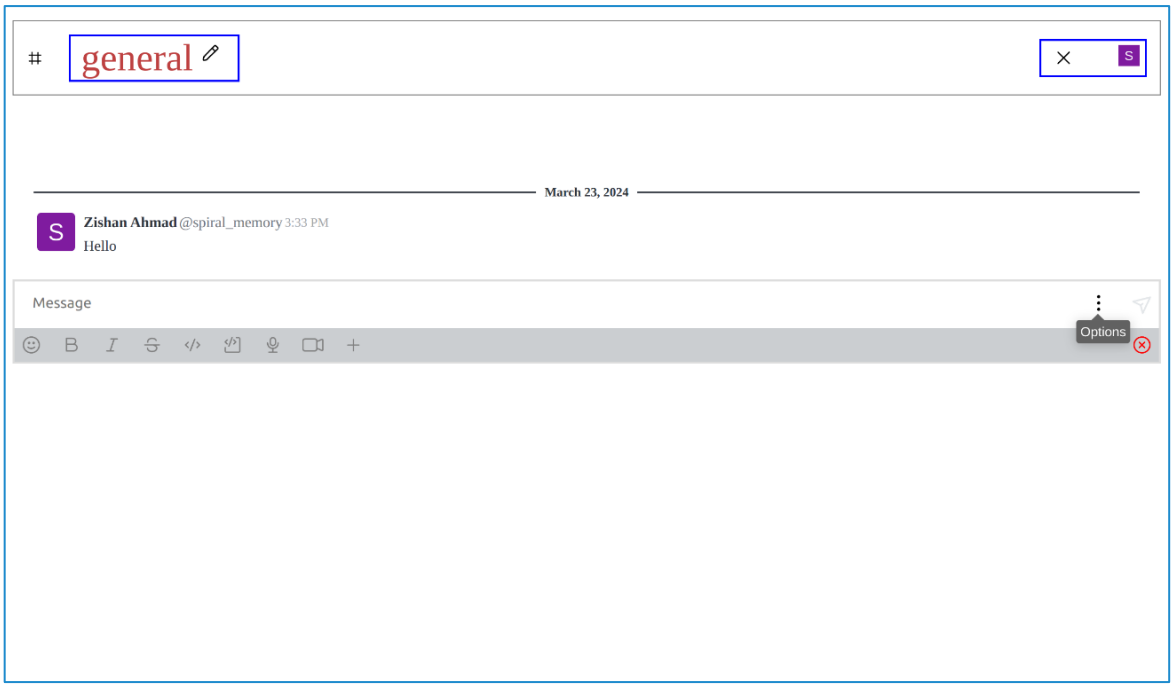

Fig. 13: Admin customizing the layout of component through drag and drop UI

4. **Improving the UI-Kit integration further –** As a part of my contribution, I have added the UI-Kit modal support in [PR #481] which enables the UI when a modal due to a RC-app triggers. To further improve the RC-app support inside Embedded Chat, the following improvements along with the solution as described as follows:

   Improvement 1: Adding contextual bar support - There are various RC-apps that open a contextual bar when triggered, but there is no support for that in the Embedded Chat. I am planning to use the sidebar component for that.

```
class TestParser extends UiKitParserContextualBar<unknown> {
        plain_text = (element: any, context: any, index: any): any => ({
                component: 'text',
                props: {
                        key: index,
                        children: element.text,
                        emoji: element.emoji,
                        block: context === BlockContext.BLOCK,
                },
        });
```

Fig. 14: Code inspiration can be taken from [actual implementation] of contextual bar

```
function ContextualBar({ view, errors, onSubmit, onClose, onCancel }) {
  useEffect(() => {
    }

    if (errors && Object.keys(errors).length) {
      const element = ref.current.querySelector(focusableElementsStringInvalid);
      element && element.focus();
    } else {
      const element = ref.current.querySelector(focusableElementsString);
      element && element.focus();
    }
  }, [errors]);
  // save focus to restore after close
  const previousFocus = useMemo(() => document.activeElement, []);
  // restore the focus after the component unmount
  useEffect(
    () => () => previousFocus && previousFocus.focus(),
    [previousFocus]
  );
  return (
    <Sidebar onClose={onClose} title={view.title} iconName={view.icon}>
      <form method="post" action="#" onSubmit={onSubmit}>
        <UiKitComponent render={uiKitContextualBar} blocks={view.blocks} />
      </form>
    </Sidebar>
  );
}

export default ContextualBar;
```

Fig. 15: Dummy code to highlight the implementation of contextual bar

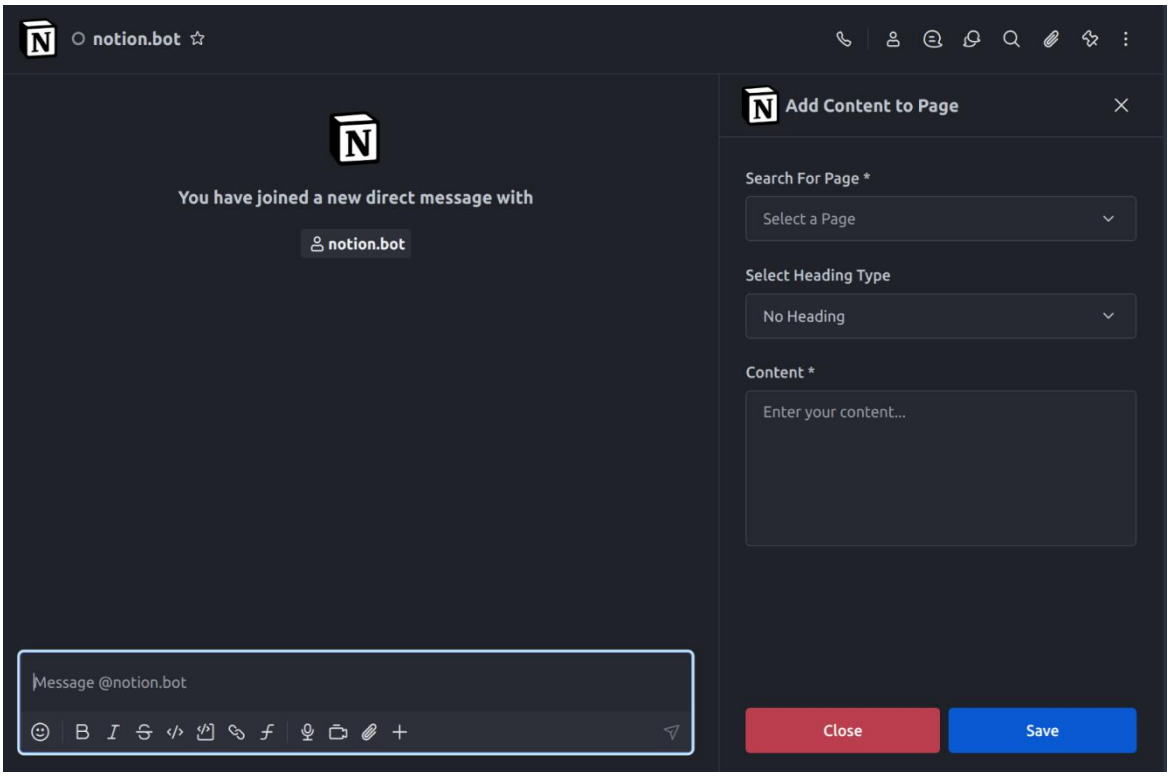

Fig. 16: Contextual bar in Rocket.Chat

Improvement 2: Action processing – As of now, any action taken in UI-Kit does not get propagated back to the RC server for processing. The plan is to make this addition by correctly capturing events like form submissions, button clicks, dropdown changes, etc., and communicating the same to the RC server for the next step.

To implement action processing, the following steps can be taken:

i. As soon as any component state changes, such as when the static-select option changes or when a button is clicked, capture the required changes.

ii. Use the API written in uiKit.js, i.e., /api/apps/ui.interaction/${appId}, to send the captured changes.

iii. When a response is returned, call a callback function to reflect the changes in the UI for the Modal/Contextual bar in Embedded Chat.

The following figure illustrates the implementation of one such handler.

```
async triggerBlockAction({
  try {
    const { userId, authToken } = (await this.auth.getCurrentUser()) || {};

    const triggerId = Math.random().toString(32).slice(2, 16);

    const payload = rest.payload || rest;

    const response = await fetch(
      `${this.host}/api/apps/ui.interaction/${appId}`,
      {
        headers: {
          "Content-Type": "application/json",
          "X-Auth-Token": authToken,
          "X-User-Id": userId,
        },
        method: "POST",
        body: JSON.stringify({
          type: "blockAction",
          actionId,
          payload,
          container,
          mid,
          rid,
          triggerId,
          viewId,
        }),
      }
    );
```

Fig. 17: Sending request to Rocket.Chat Api
to handle UI interactions

5. **Providing flexibility to customize Embedded Chat props through the Rocket.Chat workspace –** Currently, to customize different components, the user can pass the props once while setting up the Embedded Chat. Now, if the user has to change some settings like disable showing avatars or changing the room Embedded Chat is connected to, the only way is to modify those props in the code and save it to make changes, which can be cumbersome at times. So, I am proposing to add this flexibility into the RC-app. Admins can customize the props/settings of their EC integration directly by changing these settings in the RC-app of Embedded Chat. The flow for this process is as follows:

    i. The Embedded Chat will be listening to an endpoint called 'get-setting/update-config'.

    ii. Admins will change the Embedded Chat props through the settings panel of the Embedded Chat RC-app and save them.

    iii. The RC-app will send a request to this endpoint as soon as the settings are saved and `onSettingUpated` event triggers.

    iv. Embedded Chat will receive those required props, and the changes will be applied accordingly.

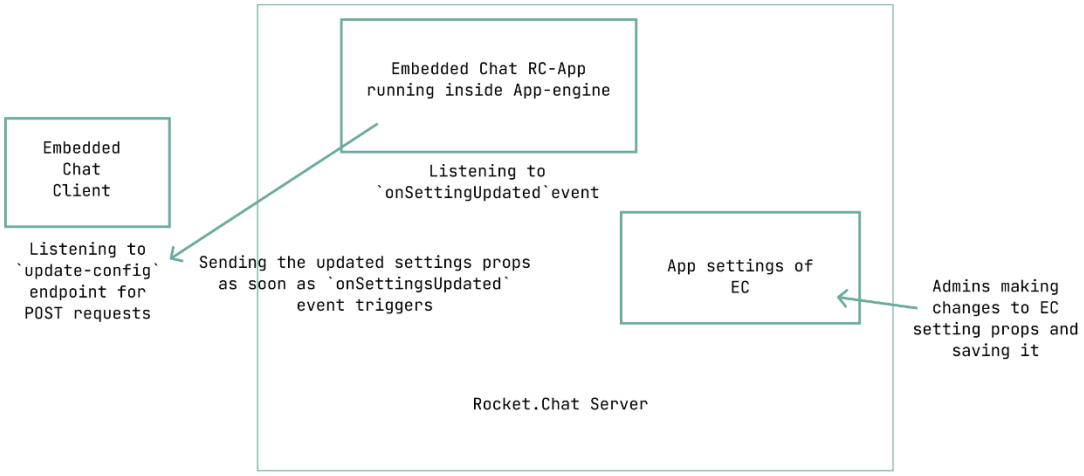

Fig. 18: Workflow: Updating EC Props through Rocket.Chat workspace

```
import {
    ISetting,
    SettingType,
} from "@rocket.chat/apps-engine/definition/settings";

export const settings: ISetting[] = [
    {
        id: "room-id",
        i18nLabel: "Room ID",
        i18nDescription: "The Room id of EC Instance",
        type: SettingType.STRING,
        required: true,      You, 6 days ago • added ec config in settings
        public: true,
        packageValue: "",
    },

    {
        id: "anonymous-id",
        i18nLabel: "isAnonymous?",
        i18nDescription: "Whether message can be read in anonymous mode",
        type: SettingType.SELECT,
        values: [
            { key: "true", i18nLabel: "True" },
            { key: "false", i18nLabel: "False" },
        ],
        required: true,
        public: true,
        packageValue: "",
    },
];
```

Fig. 19: Defining the required setting props as ISetting array

```
import {
    ISetting,
} from "@rocket.chat/apps-engine/definition/settings";

import {
    UIKitViewSubmitInteractionContext,
} from "@rocket.chat/apps-engine/definition/uikit";
import { ExecuteViewSubmitHandler } from "./handlers/ExecuteViewSubmitHandler";


import { settings } from "./settings/settings";
export class EmbeddedChatApp extends App {
    constructor(info: IAppInfo, logger: ILogger, accessors: IAppAccessors) {
        super(info, logger, accessors);
    }

    protected async extendConfiguration(
        configuration: IConfigurationExtend,
        environmentRead: IEnvironmentRead
    ): Promise<void> {
        await Promise.all([
            ...settings.map((setting) =>
                configuration.settings.provideSetting(setting)
            ),
        ]);
    }

    // Continued ...
```

Fig. 20: Providing ISetting array to extend setting configuration

```
public async onSettingUpdated(
    configurationModify: IConfigurationModify,
    read: IRead,
    http: IHttp
): Promise<void> {
    const room_id = await this.getAccessors()
        .environmentReader.getSettings()
        .getValueById("room-id");

    const isAnonymous = await this.getAccessors()
        .environmentReader.getSettings()
        .getValueById("anonymous-id");

        try {
            const url = "http://localhost:3002/updateconfig";

            const payload = {
                room_id,
                isAnonymous,
            };
            const response = await http.post(url, {
                headers: { "Content-type": "application/json" },
                content: JSON.stringify(payload),
            });

            console.log(response);
        } catch (err) {
            throw new Error(err);
        }
}
```

Fig. 21: Extracting settings props and & sending payload
to client at `updateconfig` endpoint

Currently, on `localhost:3002`, a dummy application is running and listening to the `updateconfig` endpoint just to test whether we can receive the updated setting prop in the response as soon as the settings have been saved. Fig 31 and 32 in the “Related Work” section demonstrate this process. Furthermore, the video demonstration can be accessed via link.

In the actual implementation, instead of a dummy application, this endpoint will be implemented as a POST endpoint on Embedded Chat.

6. Improving security through encryption or cookie-based approaches

- The current approach to handling authentication is as follows: the user makes a login request to the Rocket.Chat REST login endpoint, which returns an authToken and userId, both of which are then saved in localStorage. The problem with this approach is that if the site into which the Embedded Chat is integrated is vulnerable to XSS (Cross-site scripting) attacks, this token can be accessed, allowing the attacker to gain access to the user’s Rocket.Chat account. This problem can be solved in either of two ways, and I am planning to implement both and allow admins to choose one of these approaches.

Approach 1: Using cookie-based authentication – In this approach, we will have an RC app running on the apps-engine installed on the Rocket.Chat server. The app will act as a bridge between the Embedded Chat client and the Rocket.Chat REST APIs and authToken and userId will be explicitly set by the server as http-only cookie making it inaccessible through JavaScript. The flow will be as follows:

i. Embedded Chat will initiate a login request to the RC-app login endpoint, where the RC-app is listening for login requests.

ii. The RC-app will extract the user credentials and then make an HTTP request to the login endpoint of the Rocket.Chat REST API.

iii. The REST API will generate an authToken and a userId, and then send them back to the RC-app.

iv. The RC-app will extract those details, create a payload, set the authToken and userId as cookies, and then send this as a response to the client.

v. Embedded Chat will automatically send these tokens on subsequent requests as cookie headers.

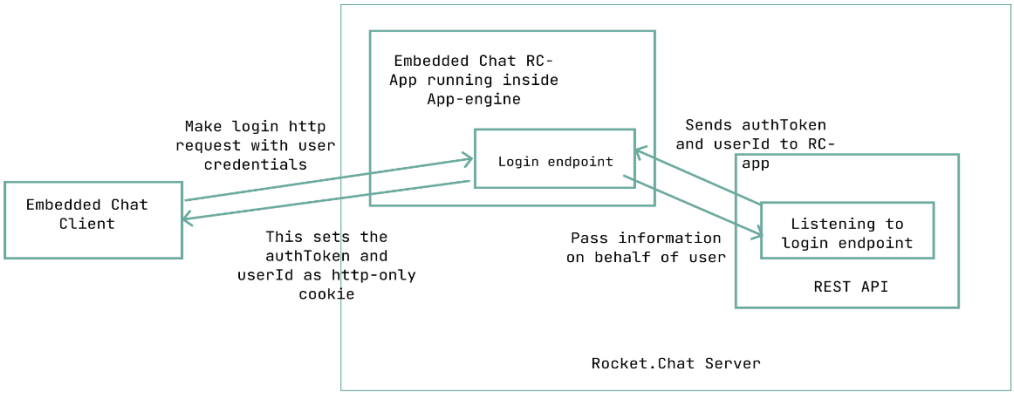

Fig. 22: Workflow: Cookie based authentication, RC-app acting as bridge

Now, once the login is completed, a similar procedure will be executed every time to fetch/post the data. For example, if Embedded Chat has to fetch messages in a channel, the flow will be as follows:

i. Embedded Chat will make a request to fetch the message endpoint on the RC-app. While making this request, the browser will automatically add the authToken and userId with the request.

ii. The RC-app, listening to these requests, will extract authToken and userId from cookies and use them to send a request to the Rocket.Chat REST API endpoint.

iii. The Rocket.Chat Server will return the messages to the RC-app, which will then return the messages to the Embedded Chat Client.

iv. Embedded Chat will render those messages in the UI.

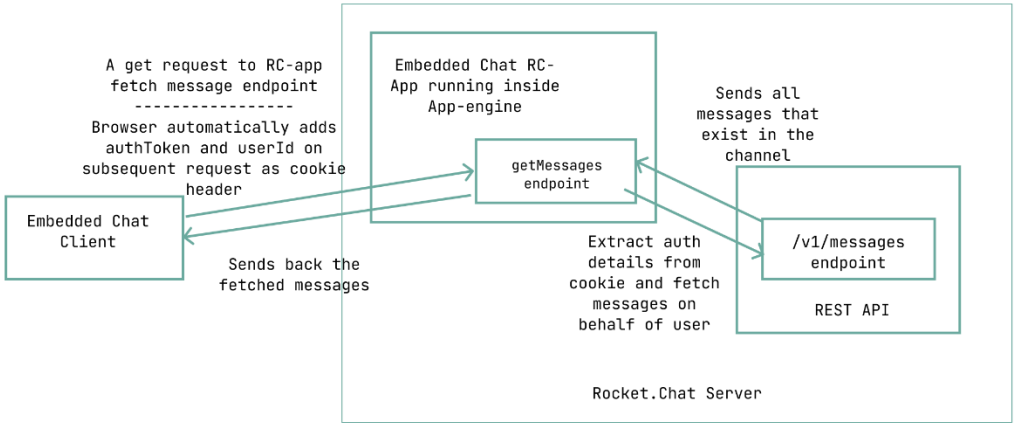

Fig. 23: Workflow: Fetching messages where RC-app is acting as bridge

To demonstrate, I have used `getMessagesEndpoint` as an example. However, in the actual implementation, we can rather have a single proxy that accepts the endpoint as a param and then forwards the requests accordingly to the Rocket.Chat REST Endpoint.

```
export class LoginEndpoint extends ApiEndpoint {
    public path = "login";

    public async post(
        request: IApiRequest,
        http: IHttp,
    ): Promise<IApiResponse> {
        const content = request.content;
        this.app.getLogger().debug(content);

        const url = "http://localhost:3000/api/v1/login";

        const response = await http.post(url, {
            headers: { "Content-type": "application/json" },
            content: content,
        });
```

Fig. 24: Listening to login endpoint, extracting credentials, and sending to RC REST API

```
export class LoginEndpoint extends ApiEndpoint {
    public async post(
        const { userId, authToken } = response.data.data;
        const accessToken = { userId, authToken };
        this.app.getLogger().debug(accessToken);

        const cookieOptions = {
            httpOnly: true,
            maxAge: 24 * 60 * 60 * 1000,
        };
        const responseHeaders = {
            "Access-Control-Allow-Origin": "*",
            "Access-Control-Allow-Methods": "POST, OPTIONS",
            "Access-Control-Allow-Headers": "Content-Type",
            "Set-Cookie": [
                `userId=${encodeURIComponent(userId)}; Max-Age=${
                    cookieOptions.maxAge
                }; Path=/; HttpOnly; SameSite=None`,
                `authToken=${encodeURIComponent(authToken)}; Max-Age=${
                    cookieOptions.maxAge
                }; Path=/; HttpOnly; SameSite=None`,
            ],
        };

        const responseContent = JSON.stringify("Request Received");

        return {
            status: 200,
            content: responseContent,
            headers: responseHeaders,
        };
    }
```

Fig. 25: Setting the authToken and userId as cookie header

```
export class GetMessageEndpoint extends ApiEndpoint {
    public async get(
        endpoint: IApiEndpointInfo,
        read: IRead,
        modify: IModify,
        http: IHttp,
        persis: IPersistence
    ): Promise<IApiResponse> {
        const cookieHeader = request.headers.cookie;

        const cookies: { [key: string]: string } = {};

        const cookiePairs = cookieHeader.split(";");
        for (const pair of cookiePairs) {
            const [key, value] = pair.trim().split("=");
            cookies[key] = decodeURIComponent(value);
        }

        const { userId, authToken } = cookies;

        this.app.getLogger().debug(userId);
        this.app.getLogger().debug(authToken);

        const url =
            "http://localhost:3000/api/v1/channels.messages?roomId=GENERAL";
        const response = await http.get(url, {
            headers: {
                "Content-type": "application/json",
                "X-Auth-Token": authToken,
                "X-User-Id": userId,
            },
        });
```

Fig. 26: Listening to get-messages endpoint, extracting tokens from cookie and sending it to REST API

```
export class GetMessageEndpoint extends ApiEndpoint {
    public async get(
            const [key, value] = pair.trim().split("=");
            cookies[key] = decodeURIComponent(value);
        }

        const { userId, authToken } = cookies;

        this.app.getLogger().debug(userId);
        this.app.getLogger().debug(authToken);

        const url =
            "http://localhost:3000/api/v1/channels.messages?roomId=GENERAL";
        const response = await http.get(url, {
            headers: {
                "Content-type": "application/json",
                "X-Auth-Token": authToken,
                "X-User-Id": userId,
            },
        });

        this.app.getLogger().debug(response.data?.messages);

        return {
            status: 200,
            content:response.data?.messages
        };
    }
```

Fig. 27: Sending messages back to Embedded Chat client

```
import { LoginEndpoint } from "./endpoints/LoginEndpoint";
import { GetMessageEndpoint } from "./endpoints/GetMessagesEndpoint";
export class EmbeddedChatApp extends App {
    constructor(info: IAppInfo, logger: ILogger, accessors: IAppAccessors) {
        super(info, logger, accessors);
    }

    protected async extendConfiguration(
        configuration: IConfigurationExtend,
        environmentRead: IEnvironmentRead
    ): Promise<void> {
        await Promise.all([
            configuration.api.provideApi({
                visibility: ApiVisibility.PUBLIC,
                security: ApiSecurity.UNSECURE,
                endpoints: [new LoginEndpoint(this)],
            }),

            configuration.api.provideApi({
                visibility: ApiVisibility.PUBLIC,
                security: ApiSecurity.UNSECURE,
                endpoints: [new GetMessageEndpoint(this)],
            }),

            configuration.slashCommands.provideSlashCommand(
                new updateECConfig()
            ),
        ]);
    }
```

Fig. 28: Providing `LoginEndpoint` and `GetMessageEndpoint` to Api configuration

Further images regarding the demonstration have been added in the Related Work section. Additionally, the video demonstration is available through the link.

Approach 2: Without proxy: This will be a simpler yet effective approach. The flow for this approach is as follows:

i. Embedded Chat (EC) makes a request to the Rocket.chat login endpoint.

ii. EC receives an authToken and userId.

iii. EC then makes a request to an endpoint called `setCookieToken` on RC-app. It converts the authToken to an encrypted version (for better security), called EC-token, and sets it as an HTTP-only cookie for the EC Client.

iv. Upon refresh, EC uses this cookie header to make a request to RC-app at an endpoint called `getRCToken`. RC-app decrypts it and provides the authToken and userId.

v. Now, with this authToken, we can resume our login session by passing {resume: authToken}.

Now that communication has been established and EC has the credentials set in the code, they will be used on subsequent requests to fetch any other information.

7. **Documentation –** As the Embedded chat already has many features, and with the addition of security, UI, and configurability aspects this year, it becomes difficult for admins to experiment and explore to understand everything. Therefore, I will keep the following things in mind:

   i. I will thoroughly maintain the work and write documentation, along with necessary diagrams and videos, so that admins/users can set up Embedded Chat and understand all required features without any hassle.

   ii. I will address any frequently asked questions that may arise while configuring the Embedded Chat app.

   iii. I will ensure that the documentation is regularly updated to reflect any changes or additions made to the Embedded Chat system, ensuring users have access to the latest information and best practices.

☺ **Fun-fact:** Over half of work time is taken up searching for information.

DETAILED WORK PLAN

My proposed timeline for the task will be as follows:

Before May 01

- I will continue working on the existing/new issues and my open PR, which might require modifications.
- I will keep interacting with the Rocket.Chat community to help each other if needed.
- I will research more about the implementation of the deliverables and will take notes so that I do not face any difficulty during the actual implementation.
- I will try making prototypes after research to have some hands-on experience.

Community Bonding Period Begins!

May 01- May 26

- With the help of my mentors, I will familiarize myself with the community and will try to know the culture and ethics of the organization.
- With the help of my mentors, I will plan a timetable and weekly calls/meetings, to submit the weekly report and ask for additional resources.
- Take inspiration from existing code and will discuss with my mentor regarding how implementation can be done in the most efficient ways.
- Discussion regarding the edge-cases we might encounter during implementation with my mentor to tackle it later.
- I will participate in discussions with the community to know the future plans of Rocket.Chat organization.

Coding officially begins!

Week 1 [May 27 – June 02]

- Setting up the development environment.
- Remove all the CSS files and inline-emotion styling, and instead convert them into separate `component.style.js` files for emotion styling.
- Extract the logic as hooks or utils as per the requirements.

- Ensure that everything is working as expected after implementing these changes.

Week 2 [June 03 – June 09]

- Work on developing multiple themes.
- Make any necessary changes to ensure that all components adapt to the themes.

Week 3 [June 10 – June 16]

- Work on UI refinements.
- Test all scenarios to confirm that all edge cases are handled correctly.
- Work on the responsiveness of the Embedded Chat to provide the best user experience.

Week 4 [June 17 – June 23]

- Improvement in the existing UI-Kit Modal.
- Work on adding contextual bar support.
- Test that in all cases, the Modal and Contextual bar are rendering correctly.

Week 5 [June 24 – June 30]

- Work on capturing events in the UI-Kit modal and contextual bar.
- Work on sending those events to the Rocket.Chat server for further processing.

Week 6 [July 01 – July 07]

- Test all the changes made up to now.
- Work on bug fixes to ensure the code is bug-free.

Midterm Evaluation

July 08- July 12

- Work on drafting a report for midterm evaluations.
- Submit the midterm report.
- Start working on the next feature, i.e., configuring component properties through in app UI Editor.

Coding continues!

Week 7 [July 13 – July 19]

- Work on identifying the components along with their properties that can be customized through UI without having any problems.
- Work on creating editor panes for all such components
- Ensure that these styles are applied properly.
- Develop a storage mechanism on the Embedded Chat so that these styles remain persistent over reloads.

Week 8 [July 20 – July 26]

- Work on developing drag and drop UI to customize layouts.
- Ensuring that the configuration remains persistent over reloads.

Week 9 [July 27 – August 2]

- Work on implementing the feature where the admin can pass props through the RC-app settings.
- Create an endpoint listening to this request on Embedded Chat.
- Send the updated props as soon as RC-app settings are updated.
- Update the props of Embedded Chat along with implementing some persistence mechanism.

Week 10 [August 3 – August 10]

- Work on cookie-based authentication using RC-app as a bridge.
- Set up a login endpoint on RC-app that will set authToken and userId as a cookie header, and appropriately handle it on the Embedded Chat client.
- Set up single proxy and accept the endpoints as params on RC-app to fetch messages, fetch members, fetch channel info, etc.

Week 11 [August 11 – August 18]

- Work on the second approach for authentication as mentioned in the “Implementation details”

- Handle the encryption/decryption (if required) on the Embedded Chat RC-app.
- Allow admins to choose between both security measures.
- Address any bug fixes, pending tasks, etc.

Final Evaluation

August 19- August 26

- Summarize the work done during this GSoC period.
- Write documentation.
- Utilize rest time as space for any delays or unforeseen events.

Post GSoC Goals

- Continue to contribute to open source.
- With respect to this project, work on any new/existing issue.
- Implement the functionality mentioned in Extras section after approval from the maintainers/mentor.

RELATED WORK

I have been contributing to Embedded Chat since January and have familiarized myself with the structure and flow of the code, as well as its functionalities. I have already made strides towards enhancing the UI and always try my best to fix any UI bug as soon as I encounter them, ensuring consistent styling throughout the app and providing the best user experience. In addition, I have already worked on integrating the UI-Kit in Embedded Chat, giving me a good understanding of how it works, with plans to further extend it during GSoC. Here is a list of some related PRs I have submitted:

1. **[MERGED]** EmbeddedChat #495 : Introduced a new feature to highlight link previews, enhancing user experience by allowing them to view key details without having to open the link.
2. **[MERGED]** EmbeddedChat #481 : Implemented all the required components for UI-Kit Modal rendering without Fuselage dependencies and added support for multiple params commands.

3. **[MERGED]** EmbeddedChat #460 : There were some inconsistencies in the styling between the thread and the (starred or pinned message) section. I tried to fix those to make the styling consistent across the app.
4. **[MERGED]** EmbeddedChat #456 : There was a new feature of video recording introduced by a fellow contributor, but it had several design issues. As soon as I noticed them, I tried fixing them in this PR, showing my care for UI consistencies.
5. **[MERGED]** EmbeddedChat #413 : This work focused on fixing multiple UI bugs, such as the emoji picker not being in the correct place, inconsistent stylings, and adding new UI features.

In addition to making UI adjustments, I also attempted to develop prototypes for other features so that during the GSoC period, I can integrate them into a real project with minimal effort and without spending excessive time understanding the problem. Attached are some screenshots of the Proof of Concept (POC) I created.

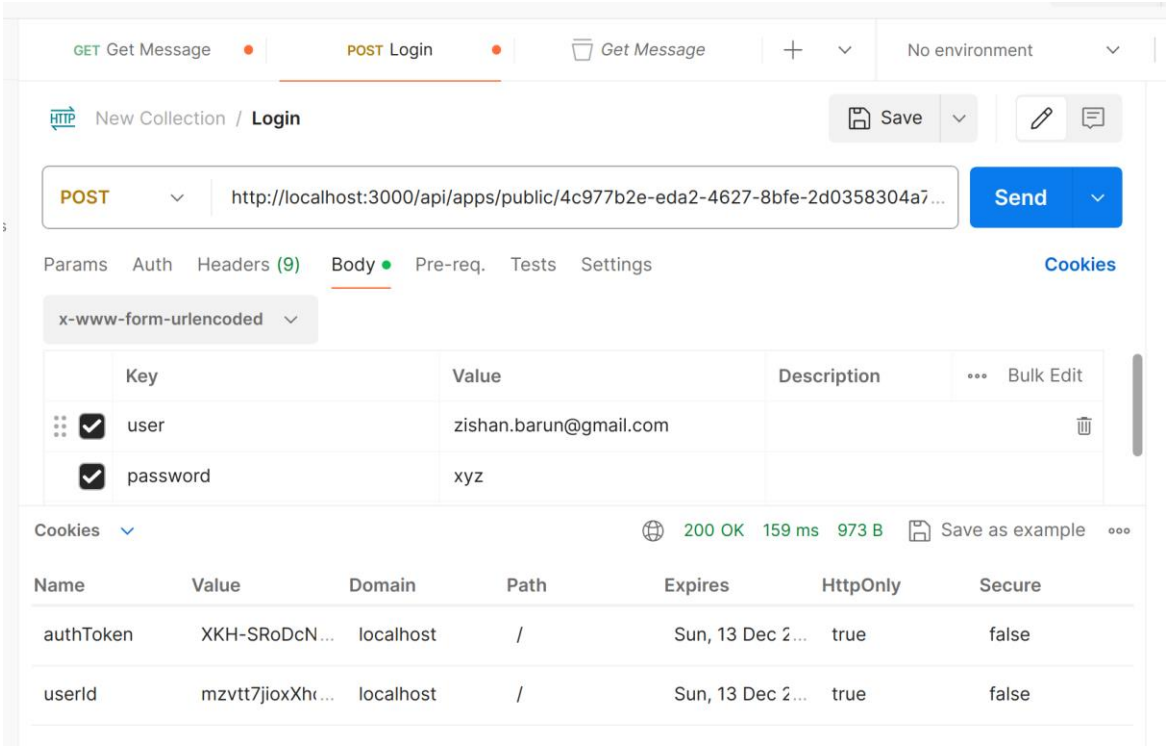

Fig. 29: Cookie-based authentication using RC-app as a bridge between the Client and REST API

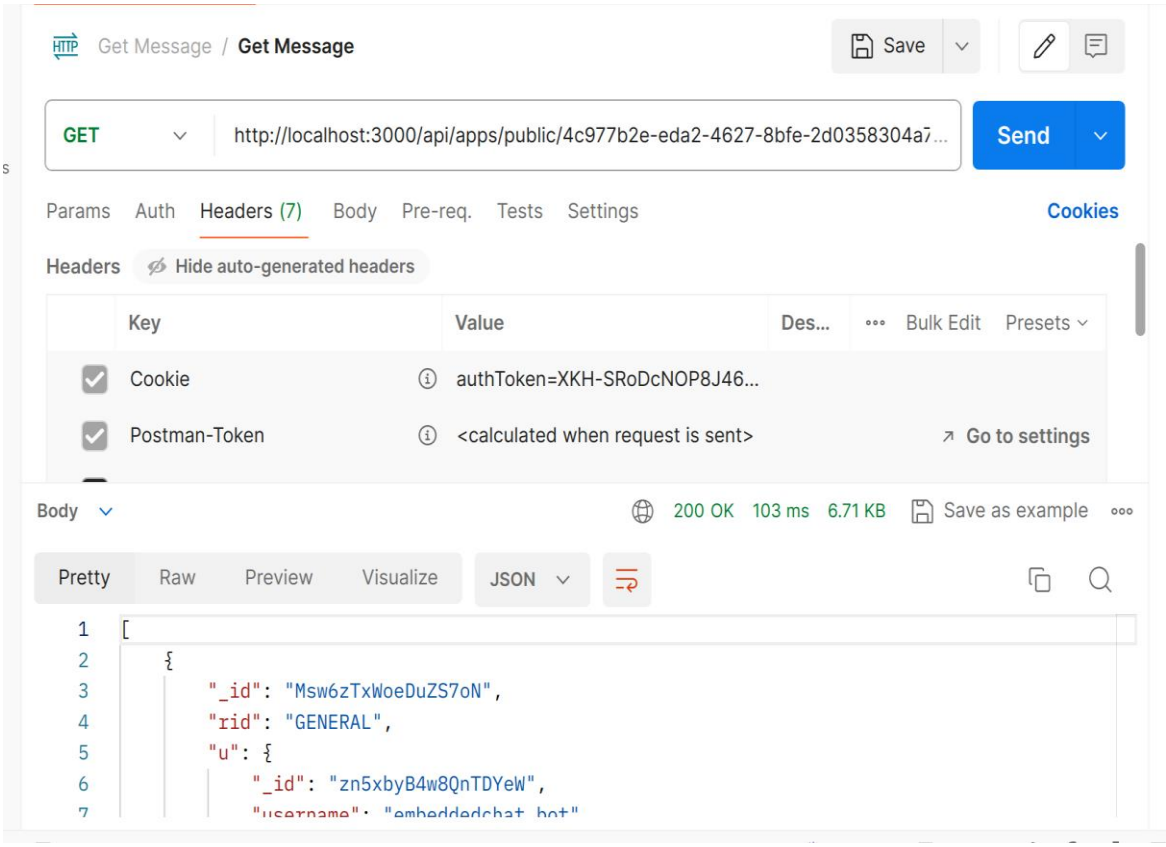

Fig. 30: Fetching messages via RC-app's endpoint, with the browser including the cookie.

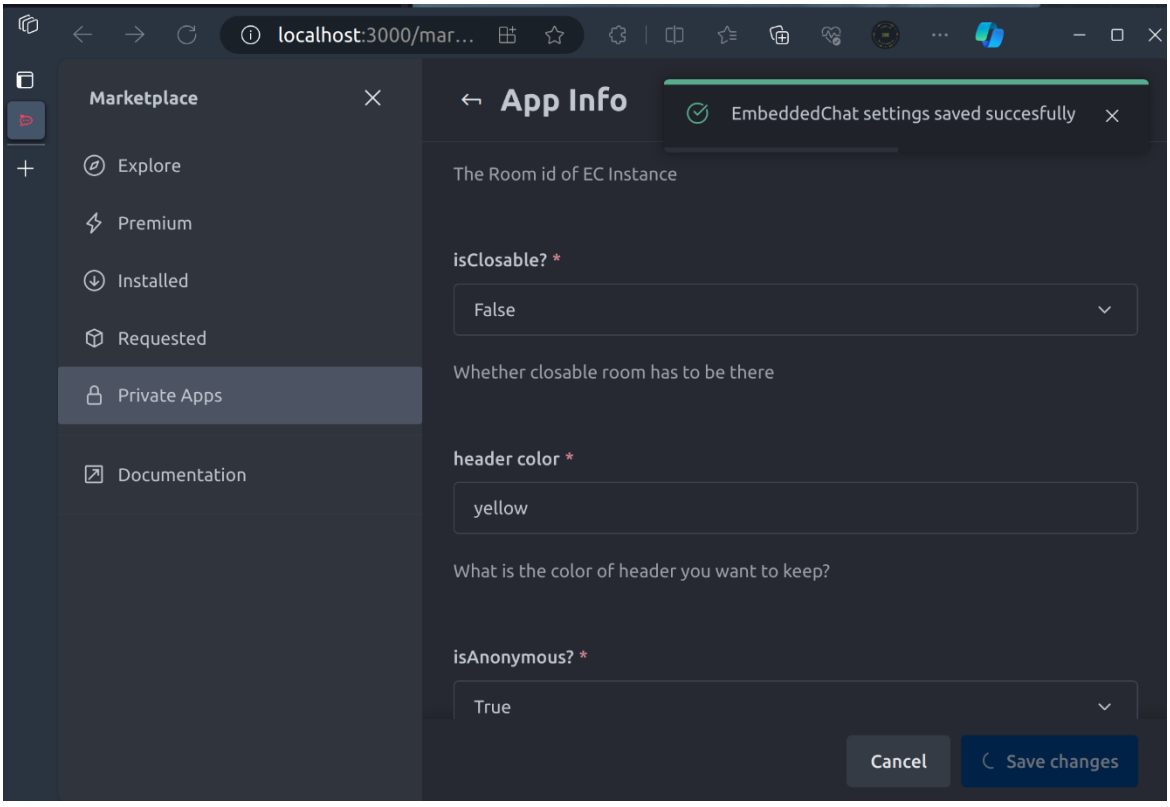

Fig. 31: User configuring the Embedded Chat instance settings directly through the Rocket.Chat.

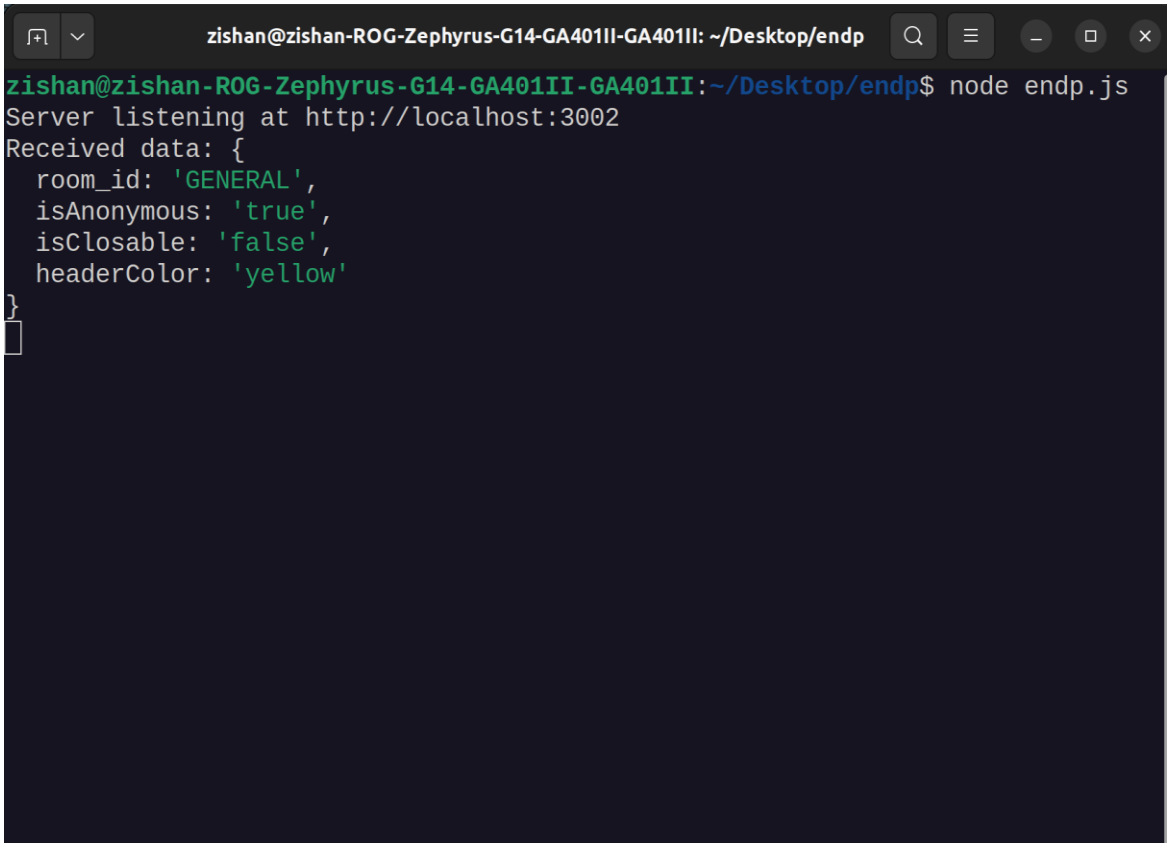

Fig. 32: Client receives updated EC settings immediately after admin saves settings in RC-app

The detailed explanation of all of it is explained in the Implementation Details section along with code and video link.

I also have several designs on Figma to demonstrate how the UI might look after enhancement and the flow in which I will achieve the proposed idea. All the wireframes can be found here.

You can find any recent video demonstrations of the prototypes I upload here, and the corresponding code can be found on my forked GitHub repository of Embedded Chat.

RELEVANT EXPERIENCES

I am a newcomer to the world of open source as a contributor, but from a user's standpoint, I have been using open-source software for a long time. From watching videos on VLC Media Player to conducting my lab experiments with Hadoop and Spark on Linux, and even writing code on VSCode, I have been actively engaged with open-source software for quite some time.

To give back to the community and play my part, I have decided to contribute to open-source projects as well. The journey began in December when I found out about Rocket.Chat while randomly looking for an open-source software to contribute to which matches my skills and tech stack. Despite a challenging build, the project sailed smoothly. I saw many people facing similar issues; I used to help everyone build the Rocket.Chat and Embedded Chat app on their local system on the community channel. One day, one of the project maintainers of Embedded Chat, Abhinav, noticed me helping people with build issues. At that time, he said to me that many people have been facing issues while building this project on Windows for a long time. "Can you please check that?" he asked. It was an awesome experience to get an issue assigned directly by the maintainer. That day, I started looking for the bug from evening until 6 A.M. the next morning. I debugged it and found that the issue was with the difference in how paths are handled in Windows and Linux. I fixed the Rollup configuration and raised a PR. It was a great moment for me to have my first PR merged in the Embedded Chat repository, and I felt so proud that my code would now be usable to so many users.

Since that day until now, I have been a consistent and active contributor to Rocket.Chat across multiple repositories, including Rocket.Chat, Embedded Chat, Apps.Notion, and Fuselage. I have managed to become a top contributor with the highest number of merged PRs on the GSoC 2024 leaderboard. I have addressed various issues within these projects, and a detailed list of all my issues and PRs is provided below:

ISSUES:

Rocket.Chat/Rocket.Chat

- Issues (Total: 3)

Rocket.Chat/EmbeddedChat

- Issues (Total: 24)

Rocket.Chat/Apps.Notion

- Issues (Total: 5)

Rocket.Chat/fuselage

- Issues (Total: 2)

PULL REQUESTS:

Total: 30 Merged: 26 Under Review: 4

Merged PRs

1. **[MERGED]** Rocket.Chat #31507 - **[FIX]** Can't remove the channel's join password.
2. **[MERGED]** EmbeddedChat #401 - **[FIX]** TOTP Modal and Toast Message Display.
3. **[MERGED]** EmbeddedChat #403 - **[ENHC]** Close modal on overlay/ESC press
4. **[MERGED]** EmbeddedChat #406 - **[FIX]** EC build issue in Windows.
5. **[MERGED]** EmbeddedChat #413 - **[FIX]** Multiple design issues.
6. **[MERGED]** EmbeddedChat #419 - **[CHORE]** Error/Logout for nonexistent channels.
7. **[MERGED]** EmbeddedChat #421 - **[FIX]** App behavior in read-only channels.
8. **[MERGED]** EmbeddedChat #427 - **[CHORE]** Fixed channel fetch issue upon refresh.
9. **[MERGED]** EmbeddedChat #431 - **[FIX]** Message fetch issue in private channels.
10. **[MERGED]** EmbeddedChat #442 - **[ENHC]** Thread menu to access all thread msgs.
11. **[MERGED]** EmbeddedChat #456 - **[FIX]** UI issues in video recorder.
12. **[MERGED]** EmbeddedChat #460 - **[FIX]** Inconsistent back button across sections.
13. **[MERGED]** EmbeddedChat #462 - **[CHORE]** Fixed logout issue.
14. **[MERGED]** EmbeddedChat #472 - **[CHORE]** Fixed linting issue.
15. **[MERGED]** EmbeddedChat #480 - **[ENHC]** Added user-mention menu.
16. **[MERGED]** EmbeddedChat #481 - **[ENHC]** **[EPIC]** UI-Kit Modal Support.
17. **[MERGED]** EmbeddedChat #495 - **[FIX]** Added link preview.
18. **[MERGED]** EmbeddedChat #503 - **[FIX]** Infinite rendering issue
19. **[MERGED]** EmbeddedChat #506 - **[CHORE]** Fixed errors, bugs, and new msg btn.
20. **[MERGED]** EmbeddedChat #510 - **[CHORE]** Fixed message send issue.
21. **[MERGED]** EmbeddedChat #516 - **[FIX]** **[ENHC]** Fallback icon, download/delete option.
22. **[MERGED]** EmbeddedChat #525 - **[FIX]** Fixed sidebar UI inconsistencies and bugs.
23. **[MERGED]** EmbeddedChat#530 - **[FIX]** Static Message Limit.

24. **[MERGED]** Apps.Notion #48 - **[ENHC]** Overflow menu to go to subpage modal.
25. **[MERGED]** Apps.Notion #53 - **[FIX]** Title input visibility issue.
26. **[MERGED]** Apps.Notion #51 - **[ENHC]** Close window after connection success/failure.

Open PRs

1. **[OPEN]** Rocket.Chat #31748 - **[FIX]** Quoted message links
2. **[OPEN]** Apps.Notion #62 - **[CHORE]** Terminology fixes
3. **[OPEN]** Apps.Notion #65 - **[ENHC]** Add content to page through Apps.Notion.
4. **[OPEN]** Apps.Notion #70 - **[ENHC]** Update DB Records through Apps.Notion

Along with contributing to the Rocket.Chat organization, I have also contributed to some other organizations to learn more about different projects and diverse communities. PRs for those are listed below.

1. **[MERGED]** care_fe #7408 - **[FIX]** Validation for experience text input.
2. **[OPEN]** care_fe #7404 - **[FIX]** Unnecessary mic permission warning.
3. **[OPEN]** ultimate_alarm_clock #515 - **[FIX]** Overflow pixel issue.

Prior to contributing to Rocket.Chat, I also interned at IIT Hyderabad on a 5G testbed project, aiming to enhance its architecture by implementing per-procedure network functions. This effort resulted in a 17.5% reduction in control plane traffic, as indicated in the research paper.

In addition, I also participated in the NeST Digital Hackathon, where me with my team developed an MT to MX converter using microservices architecture. We won the first prize in that hackathon.

To know more about me, you can find information about my projects, interests, and activities from the website.

SUITABILITY

Why are you the right person to work on this project?

I am the right person to work on this project as I have prior experience working on the Embedded Chat repository thoroughly and have contributed more than 2.5K lines of code. I have made various UI modifications to make the design consistent, which need to be further extended in this GSoC project. I have added support for the Ui-Kit modal rendering, which needs enhancement within this timeline. I understand the flow and code structure, making it easy for me to make the required modifications and additions.

I have also attended the RC-app development workshop every week, which helped me learn to write RC apps from scratch. Moreover, I have worked on the Notion RC app and implemented features with 500+ lines of code, demonstrating my in-depth understanding of various event listeners, webhooks, UI-Kit, etc., which will be required in building the cookie-based authentication through RC-app acting as bridge between the client and REST endpoint. Other features like configuring the Embedded Chat through RC workspace also heavily rely on Ui-Kit handling and settings UI, hence making me a good fit for this project. Furthermore, I have made several prototypes, providing proof of concept that I can achieve what I am claiming here.

I am well-versed in ReactJS, NodeJS, version control systems, RC-apps, Rocket.Chat REST APIs, etc., and have highlighted my skills by being a top contributor in the GSoC leaderboard. It demonstrates my hard work, dedication, and love for the Rocket.Chat community. I am also a quick learner, so even if I am unaware of something, I know I can learn it and work upon it.

I have been a huge fan of keeping the code bug-free, hence I always love to review fellow contributors' work, helping them out in fixing those bugs, explaining the flow, and why they might be going in the wrong direction and appreciating their work. I also love having my work reviewed. Following the same principle, I try to fix any issue or bug introduced by any PR as quickly as possible, again showing my love for the project as well as my mindset of keeping the code as bug-free as possible. In addition, I am quite good at communicating with people and have made several awesome friends while working on the project.

Moreover, I love the Rocket.Chat community. I have learned a lot from my peers and maintainers/mentors during this period and would like to do more of this in the future.

TIME AVAILABILITY

How much time do you expect to dedicate to this project?
As I am in my final year and my end-semester examinations will be over by April 22nd, I will not have any major academic commitments during the GSoC time. I would be able to dedicate 40-45 hours per week.

Please list jobs, summer classes, and/or vacations that you will need to work around.

I have not enrolled in any summer classes or internships. I do not have any plans to go on vacation during the GSoC period. I will be available for calls from 11 am IST to 9 pm IST on both weekdays and weekends. Starting in August, I might have to join the company that I received a PPO from, but I plan to complete my work before then. Even if it is not completed, I will try my best to manage both.