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

Proposal: VSCode extension for Rocket.Chat, Enabling Code-Focused Interaction

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

Debdut Chakraborty (@debdutdeb)

ABSTRACT

The goal of this project is to create a VSCode extension that enables communication with members of Rocket.Chat workspace directly from the VSCode editor. This extension will provide developers with functionality to share code directly from editor, engage in discussions on the code, enable code collaboration, save meeting transcripts with exact code segments, 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. With that said, in organizations where multiple developers collaborate on shared code using Rocket.Chat channels, users often face the cumbersome task of manually copying and pasting code snippets from their VSCode editor to Rocket.Chat for discussion. In addition, locating these snippets along with discussions later within the codebase is a challenge. During code reviews conducted within the channel, reviewers may suggest changes, but there is presently no way to show and suggest these modifications directly in the code editor, nor is there a way to keep a record of the discussions around specific code segments. So how can all these problems be solved? Why not introduce a VSCode extension to bridge the gap between Rocket.Chat and VSCode by integrating code discussions within their workflow?

This project will assist developers looking to integrate discussions over Rocket.Chat into their workflow and Rocket.Chat as a whole. Its benefits would include:

- Convenience: Developers can easily discuss their code segments with team members directly from the editor. They can also engage in general conversations in DMs or channels without leaving VS Code.
- Collaboration: This extension will promote easy collaboration among team members by enabling them to document all conversations with the precise code segments.
- Code Review: Reviewers can suggest changes and provide feedback on the channel. Developers can see those modifications in the editor itself, reducing time and effort, leading to more efficient code reviews and iterations.
- Competitive Advantage: Providing such an extension that integrates discussions happening over Rocket.Chat into code editor will set Rocket.Chat apart from other communication platforms. This can attract new organizations seeking a platform that reduces the time in review cycle and makes developers' lives easier.

GOALS

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

1. Developing a VSCode extension that facilitates easy collaboration and discussions within the team directly from the VSCode editor.
2. Implementing login via Rocket.Chat OAuth. If time permits, work on other login functionality to make the extension flexible and allow users to choose from various login options.
3. Implementing the display of channels/teams/contacts within the VSCode editor.
4. Enable discussion based on precise code segments with persistence directly from code editor using VSCode Comment API.
5. Working on a mechanism to ensure that code discussions remain intact at the right point even after code modifications by storing the commit hash in metadata. This will enable comparison of the latest code with its previous versions and placement of comments in the appropriate position.
6. Having the ability to share code links and open them as such.
7. Enabling code editing directly by reviewer within the Rocket.Chat channel and reflecting those suggested changes into the code editor.
8. Enabling general chat functionality along with required features like pinning messages, starred messages, etc., directly from VSCode.

DELIVERABLES

By the end of GSoC, I plan to deliver a VSCode extension for Rocket.Chat that enable developers to share code, discuss, and accept reviewed suggestions directly inside VSCode for easy collaboration.

1. A VSCode extension for Rocket.Chat that can enable all developer-centric functionalities related to communication directly within VSCode. **[NEW]**
2. Implementing the login functionality to access the workspace via OAuth with Rocket.Chat if enabled by the workspace admin. If time permits, optionally work on developing different login methods such as entering email and password through the `Webview` or command panel, or using OAuth with alternate providers. **[NEW]**

3. Implementing a `Tree View` to display the teams, channels, and contacts accessible to the user in the workspace, allowing developers to choose whom they want to communicate with. **[NEW]**
4. Adding functionality to share code directly from the editor and have a discussion on the code segment using the Comment API, properly including thread title, participants, etc. **[NEW]**
5. Make those discussions persistent in the editor by saving metadata of that code segment with thread ID in a JSON file to showcase the discussion whenever required. **[NEW]**
6. Improving upon my work on point 5 by making it resistant to code changes, by saving discussions based on commit snapshots and adjusting the position of comments by comparing them with the current changes. **[NEW]**
7. Adding functionality to discuss the same code segment in multiple channels simultaneously. **[NEW]**
8. Implementing feature to share code links directly from VSCode. **[NEW]**
9. Enable code editing from the channel itself and reflect those suggested changes directly into the editor. **[NEW]**
10. Implement other general chat functionality using `WebView` so that users can discuss other topics as well without leaving their coding workflow. **[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 a labeling mechanism to help developers understand the type of discussion at a glance, whether it's a code review, general discussion, bug fix, etc. **[NEW]**
2. Extending the feature mentioned in point 9 to enhance Rocket.Chat's role as a bridge between two communications happening with users from VSCode. One user can make changes, which will be sent via Rocket.Chat, and the other can then view the changes and vice versa, along with saving the session transcript as a discussion/thread in the channel. **[NEW]**

3. Adding a feature to notify developers of updates in a particular channel/room/thread only if the developer has subscribed to the discussion in that channel. **[NEW]**
4. Adding a feature to notify reviewers as soon as a PR is raised by a developer, enabling instant code reviews with the help of the VSCode extension and integration of the RC app. **[NEW]**

IMPLEMENTATION DETAILS

1. Enabling Code Discussion from editor – This feature is the most important aspect and is the primary goal of this project, which is to enable discussions on code segments directly from the editor. The entire proposal mainly focuses on addressing various scenarios, preserving discussions, and making the discussion resistant to code-change, along with other features to be discussed later. Since this is our primary focus, I will first discuss this section. However, to implement this, we will require login functionality and a display of channels/contacts for selection. To understand this, I will abstract those functionalities for now. Later, each part, including the implementation of login, implementing WebSocket communication, display of channel/contacts using Tree View is explained in detail. With that said, Let's start with the implementation details for this feature:

To enable this feature, we can use the VSCode comments provided by the Extension API. To utilize this comment API, we first need to define some commands in `contributes` within `package.json` that can be triggered to start discussions, reply to existing discussions, refresh current messages, etc. Once the command is registered, we need to add "comments/commentThread/title" and "comments/commentThread/context" in contributes to enable the comment API in the extension. The addition to `contributes` which can enable this is shown as follows:

Fig. 1: Registering required commands to contributes, which will be triggered when needed and can act as buttons.

Fig. 2: Adding support for the comment API functionality involves adding the mentioned items to `contributes` in `package.json`.

Now, once the setup is complete in `package.json`, we must register these commands to the context and provide a function that will trigger once these commands are executed. Registering these commands can be done as shown:

```
  context.subscriptions.push(
    vscode.commands.registerCommand(
      "vsCodeRc.startDiscussion",
      (reply: vscode.CommentReply) => {
        rcComment.startDiscussion(reply);
      }
    )
  );

  context.subscriptions.push(
    vscode.commands.registerCommand(      You, 4 weeks ago • a functional vscode rc exten
      "vsCodeRc.reply",
      (reply: vscode.CommentReply) => {
        rcComment.replyNote(reply);
      }
    )
  );

  context.subscriptions.push(
    vscode.commands.registerCommand(
      "vsCodeRc.refreshMsg",
      (thread: vscode.CommentThread) => {
        rcComment.refreshMsg(thread);
      }
    )
  );
}
```

Fig. 3: Registering these commands and providing required function to execute.

The initial setup is now complete, now to enable comment support in UI, we need to have `commentController` and then it also has to be registered in the context. The following images shown the implementation of the same:

```
You, 4 weeks ago | 1 author (You)
export class RCComment {
  public commentController: vscode.CommentController;

  constructor() {
    this.commentController = vscode.comments.createCommentController(
      "send-code",
      "Code Sharing RC"
    );
```

Fig. 4: Initializing `commentController`

```
const rcComment = new RCComment();
context.subscriptions.push(rcComment.commentController);
```

Fig. 5: Pushing this `commentController` to context

Now, we have to enable commenting range provider, with the help of that, users will be able to select the code and can start the commenting process. The following shows the implementation of commenting range provider:

```
this.commentController.commentingRangeProvider = {
  provideCommentingRanges: (document: vscode.TextDocument) => {
    const lineCount = document.lineCount;
    return [new vscode.Range(0, 0, lineCount - 1, 0)];
  },
};
```

Fig. 6: Spanning entire document to enable comments on every section of code

Finally, we need to implement the logic to initiate discussions, reply to discussions, refresh messages, etc. In the current implementation, I've enabled messages to be sent in the "GENERAL" channel, and for each code segment, there will be a thread where the discussion will take place. This allows all code segments to be discussed separately. In the actual implementation, users will have the flexibility to choose team channels or discussion channels before sending messages, enabling appropriate partitioning and management mechanisms in a team situation.

The code snippets and demonstration images show the implementation of the required feature as follows:

```
class NoteComment implements vscode.Comment {
  savedBody: string | vscode.MarkdownString;
  id: number;

  constructor(
    public body: string | vscode.MarkdownString,
    public mode: vscode.CommentMode,
    public author: vscode.CommentAuthorInformation,
    public parent?: vscode.CommentThread,
    public contextValue?: string
  ) {
    this.id = ++commentId;
    this.savedBody = this.body;
  }
}
```

Fig. 7: A class that provides a uniform way to create new comments and append them to a thread.

```
public async startDiscussion(reply: vscode.CommentReply) {
  const thread = reply.thread;

  const sentResponse = await apiClient.handleSendMessage(
    AuthData.getAuthToken(),
    AuthData.getUserId(),
    selectedText
      ? "```\n" + selectedText.replace(/`/g, "\\`") + "\n```"
      : "No code selected"
  );

  const threadId = sentResponse.message._id;
  const range: any = thread.range;
  const rangeString = `${range.start.line}-${range.start.character}-${range.end.line}-${range
  mappings.set(rangeString, threadId);
  const res = await apiClient.handleSendMessage(
    AuthData.getAuthToken(),
    AuthData.getUserId(),
    reply.text,
    threadId
  );
```

```
  const newComment = new NoteComment(
    reply.text,
    vscode.CommentMode.Preview,
    {
      name: `@${res.message.u?.username}`,
      iconPath: vscode.Uri.parse(
        `https://open.rocket.chat/avatar/${res.message.u?.username}`
      ),
    },
    thread,
    thread.comments.length ? "canDelete" : undefined
  );

  thread.comments = [...thread.comments, newComment];
}
```

Fig. 8: Implementation of `startDiscussion` function

The flow for the implementation is as follows:

i. As soon as the discussion starts, the function first sends the selected code segment to the channel. It then receives a response from the Rocket.Chat REST API, from which it extracts the message _id so that other messages can be sent into the thread.

ii. It creates a mapping between the threadId and line numbers that the VSCode comment API relies on.

iii. Next, it creates a new Comment using the class shown before.

iv. Finally, it appends the new comment into the VSCode comment interface.

```
public async replyNote(reply: vscode.CommentReply) {
  const thread = reply.thread;
  const range: any = thread.range;
  const rangeString = `${range.start.line}-${range.start.character}-${range.end.line}-${range.end
  const threadId = mappings.get(rangeString);

  const res = await apiClient.handleSendMessage(
    AuthData.getAuthToken(),
    AuthData.getUserId(),
    reply.text,
    threadId
  );

  const newComment = new NoteComment(
    reply.text,
    vscode.CommentMode.Preview,
    {
      name: `@${res.message.u?.username}`,
      iconPath: vscode.Uri.parse(
        `https://open.rocket.chat/avatar/${res.message.u?.username}`
      ),
    },
    thread,
    thread.comments.length ? "canDelete" : undefined
  );

  thread.comments = [...thread.comments, newComment];
}
```

Fig. 9: Implementation of `replyNote` function

The `replyNote` function implementation is like the implementation of `createDiscussion`; it's just that now, instead of creating a mapping, it looks for the mapping and sends a message into the respective thread, then create a NoteComment and append that comment into the VSCode comment UI.

```
public async refreshMsg(thread: vscode.CommentThread) {
  const range: any = thread.range;
  const rangeString = `${range.start.line}-${range.start.character}-${range.end.line}-${range.end
  const threadId = mappings.get(rangeString);
  const res = await apiClient.getThreadMessage(
    AuthData.getAuthToken(),
    AuthData.getUserId(),
    threadId
  );

  thread.comments = [];
  res.messages?.forEach((message: any) => {
    const newComment = new NoteComment(
      message.msg,
      vscode.CommentMode.Preview,
      {
        name: `@${message.u?.username}`,
        iconPath: vscode.Uri.parse(
          `https://open.rocket.chat/avatar/${message.u?.username}`
        ),
      },
      thread,
      thread.comments.length ? "canDelete" : undefined
    );

    thread.comments = [...thread.comments, newComment];
  });
}
```

Fig 10: Fetch new messages when 'Refresh` icon is clicked

Note: In the current implementation, users must fetch the messages to see if there are any new messages in the thread. In the actual implementation, the plan is to fetch messages in real-time for the open thread. Currently, I have established Websocket communication, but the problem is that the VSCode comment API requires a trigger within the comment interface to activate a function. I'm still researching it; there might be ways to implement it. If possible, I plan to do so. Otherwise, we may stick with the current approach, as even in the GitHub Pull Request extension, one has to refresh to fetch new comments.

The following is an image demonstrating the current implementation in action:

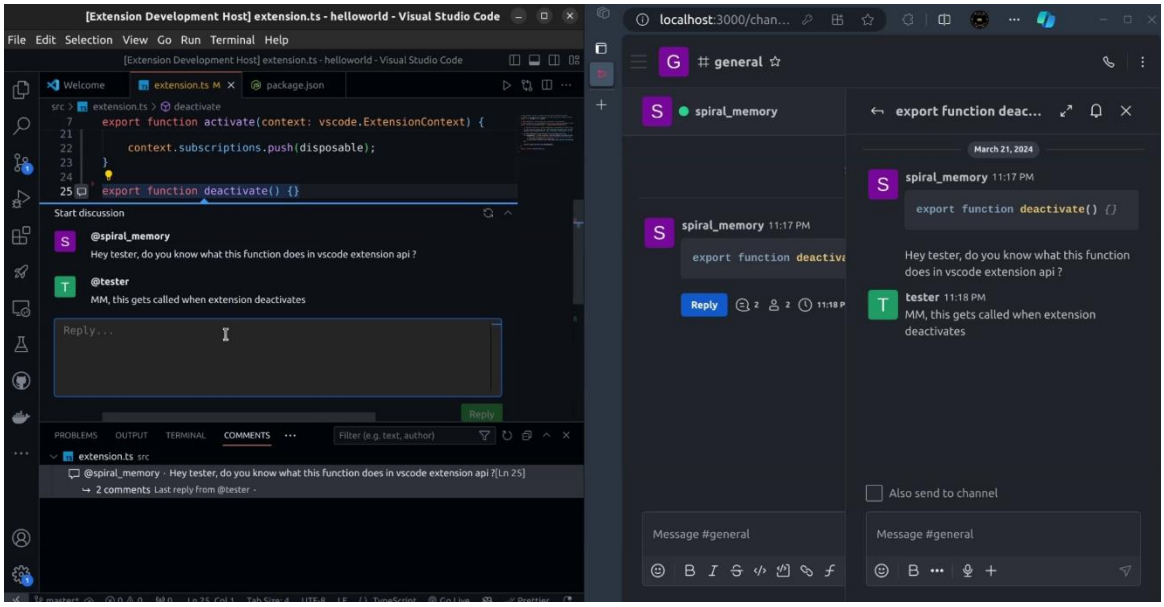

Fig. 11: A working demo of the current implementation

Now, there are various issues that need to be addressed. Next, I'll discuss those issues and potential solutions that I plan to implement to resolve them.

Q. How can we make the discussion persistent?

A common problem with this VSCode comment API is that the discussion will be lost as soon as the extension is stopped. It will obviously persist in the Rocket.Chat channel, but the discussion will be lost in the code editor, which was the main purpose. Hence, to solve this issue, I propose that we automatically create a folder named `discussions`, and then within that, we can have a JSON file which stores the metadata of the code, such as line range, character range, and the associated threadID. Then, our extension will fetch the data from that JSON file and accordingly place the comments/discussion in the desired location where the discussion happened earlier. This inspiration is taken from another extension named "Code Tours" as well as after a discussion with the mentor.

This potential approach can solve the issue and will make the discussion persistent across sessions and among all the users working on the codebase. An example image showing such metadata stored in a JSON file is displayed below:

Fig. 12: A dummy JSON file storing metadata of code segments

Q. How can we make the discussion resistant to code changes?

Another problem is that currently, we have stored the metadata of the code in such a way that to identify a particular code segment, we are using line numbers. However, when there are changes to the code by developers, the code segment on which the discussion happened will displace from its location. Since we are using line numbers to pinpoint the code segment, the extension will display the discussion at the wrong location. This is a serious problem. To make it resistant to code changes, we can integrate a git module into the extension. While saving the metadata, we can also store the commit hash on which the discussion happened. Now, when the code has changed and other developers are viewing it, we can compare the current file with the code in that commit hash, find out the differences between them, and automatically adjust the location of comments accordingly in three cases:

1. If lines are added above the code on which it was originally discussed, add the number of lines changed above the code to find the correct location of the discussion.
2. If new code is added below the code segment on which it is discussed, no changes are required.
3. If lines are changed within the discussion range, we don't have to do anything, but at least we can inform the user that in this commit, this code segment was changed. So, to understand the context, go to that commit.

This way, this problem can be handled, and the extension will work properly even if the code has changed drastically.

The following images demonstrates the problem as well as the solution:

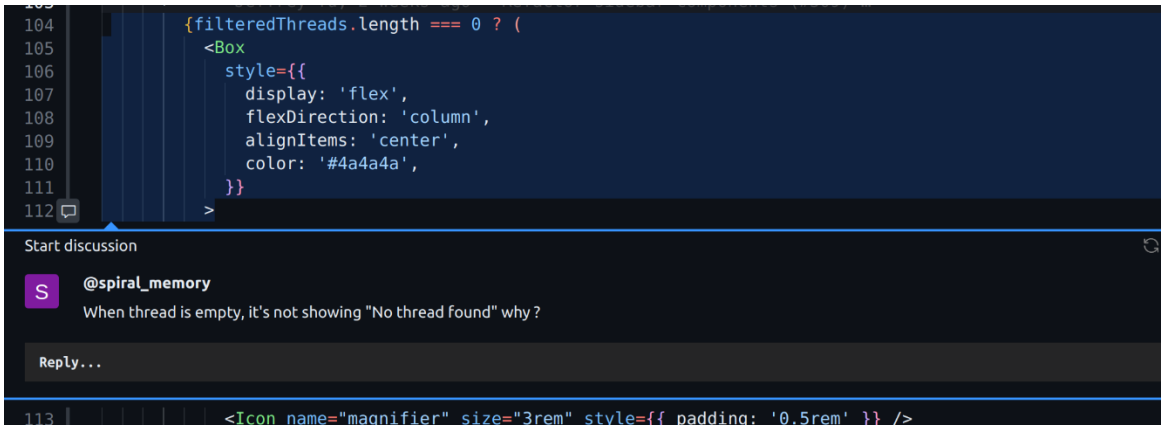

Fig. 13: Discussion occurred on code lines 104 to 112.

Fig. 14: Due to a function implemented at the top,
the location of the discussion comment has been changed.

To fix this issue, simply add 6 lines at the starting point to display the discussion at the correct location, which can be achieved by comparing the current code with the commit snapshot.

Q. How can someone discuss exact code lines on different channels?

So, currently, we can discuss different code segments on different channels as per our choice. However, consider a scenario where a developer needs to send a code segment to a team of testers for testing and wants to discuss the same segment with one of their developer friends. How can they achieve this? The plan is to implement a tab system in the comment API so that users can navigate between different discussions, even for the same code segments. Accordingly, the structure in the JSON can be adjusted by storing an array of threadIDs instead of a single threadID.


```
{
    "title": "Discussions",
    "threads": [
      {
        "threadIDs": ["86jh87jgf65hjhhi77rcx665", "9df8n7djf6gh3j4kj89d", "4n8d7kfj3gh67fj2k91"],
        "metadata": {
          "filename": "src/AllThreads/Allthreads.js",
          "startLine": 436,
          "endLine": 532,
          "startChar": 1,
          "endChar": 48
        },
        "threadTitle": "Could you please review this code?"
      },
      {
        "threadIDs": ["123abc456def789ghi"],
        "metadata": {
          "filename": "src/AnotherThread/AnotherThread.js",
          "startLine": 123,
          "endLine": 234,
          "startChar": 1,
          "endChar": 56
        },
        "threadTitle": "Discussion about feature implementation"
      }
    ]
  }
```


Fig. 15: Instead of a single thread ID, we now have an array of threadIDs in which discussion happened on a particular code segment.

2. Implementing login methods – We know that, Rocket.Chat can be either cloud-hosted or self-hosted by the organization. Hence, there is no one central server, and the organization can host it on their server. So, before we think of logging in, we should first consider the functionality to connect the user to the required workspace. As soon as the extension is installed, we need to prompt the user to enter the workspace URL with which the developer wants to connect. To implement this, we can either set it up using the `showInputBox` provided by the VSCode extension API, or we can integrate it while setting up the webview view later on. An example implementation using `showInputBox` is shown below:

```
context.subscriptions.push(vscode.commands.registerCommand('rocket-chat.setupWorkspace', () => {
    vscode.window.showInputBox({
        prompt: 'Enter your workspace URL',
        placeHolder: 'e.g.http://open.rocket.chat/...'
    }).then((input) => {
        if (input) {
            setupWorkspace(input);
        }
    });
}));
```

Fig. 16: Code snippet showing the implementation for input box.

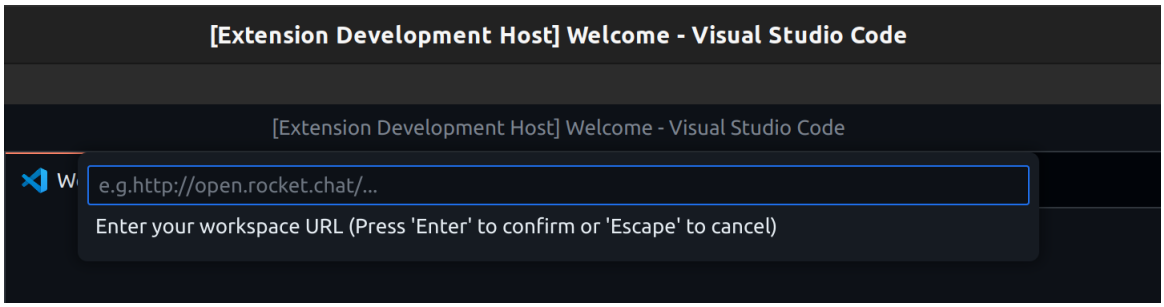

Fig. 17: Prompting user to provide workspace URL to connect to

Now, once the user is successfully connected to the workspace, they must log in to their account. Rocket.Chat provides many ways of logging in, and the organization may have its preferences. Some organizations may prefer their developers to log in through normal email and password, while others might allow users to log in via a desired OAuth service.

So, the plan is to support multiple login options so that users can log in via their preferred method, with our primary focus on implementing OAuth login through Rocket.Chat. If time permits, we will include other login methods such as logging in through "Email/Username and Password" and enabling OAuth with different providers.

Let's discuss the implementation details for “Login via Rocket.Chat OAuth” and “Login Via Credentials” methods in detail:

Method 1: Login via Rocket.Chat OAuth – This is the preferred method for the user to sign in to the app. The user clicks on the login button, which will open the webpage asking for their credentials if they are not logged in. Once successful, the OAuth provider will ask to grant access to the extension. Once done, the extension will exchange the authorization code for an access token, which can be further used by our extension. I haven’t tested this yet, but I believe Rocket.Chat does provide an option to set up an external app. Hence, I believe it will be a good addition as it streamlines the login process. Most extensions have it implemented for their work.
The flow of the login will be as follows:

i. The Rocket.Chat admins configure Rocket.Chat to register this extension and provide a redirect URI.

ii. Admins will receive client ID, client secret, auth URL, access token URL, etc.

iii. Implement this login method into the VSCode extension, and once configured, it can initiate login and will listen to the callback URL.

iv. Once authentication is successful, start the session as usual.

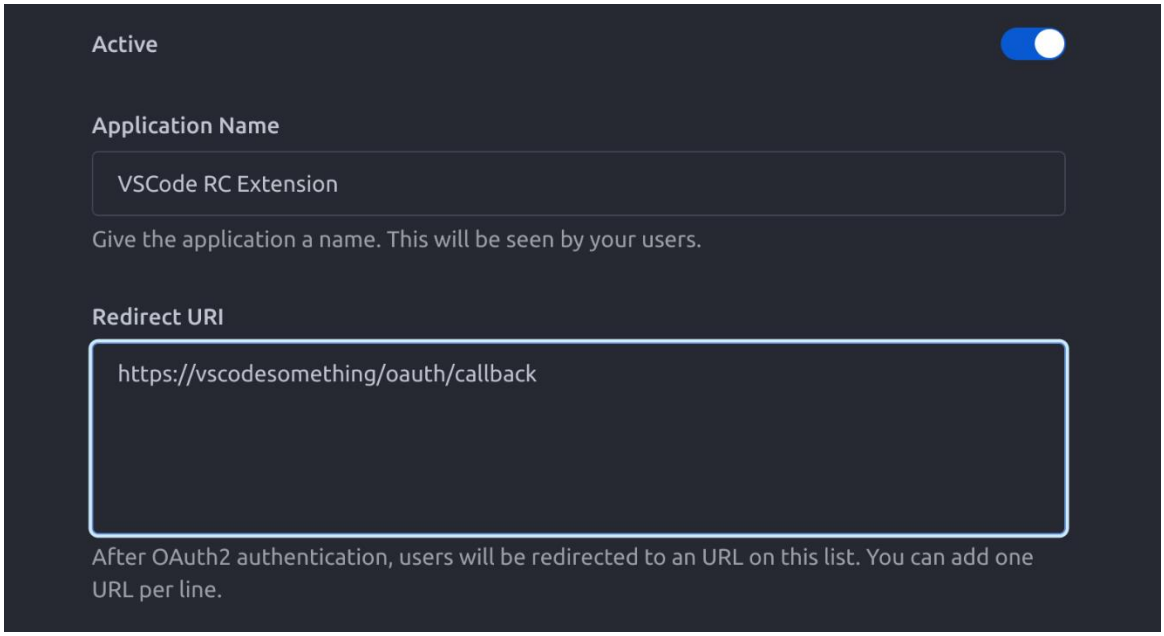

Fig. 18: Admin registering VSCode RC extension for 3rd party login

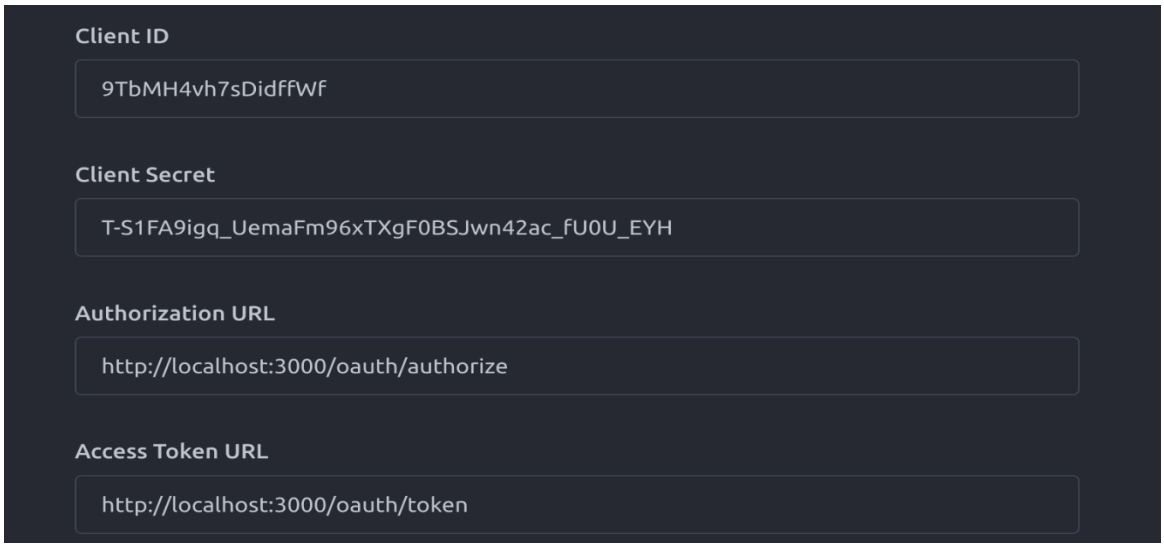

Fig. 19: Admin getting Client ID and Client Secret and other relevant details that will be helpful in setting up the SSO login

Now, we have access to these details; the admin can share this information to enable developers to set up their SSO login facility.

Method 2: Login via Username and Password – This is the simplest login method that Rocket.Chat provides. The user just inputs the username/email along with password. To perform this login, we can use the REST API endpoint `/api/v1/login`, which will return us an authToken and userId. These will be used further in making subsequent requests and setting up real-time communication with the Rocket.Chat server. In the VSCode extension, this login functionality can either be implemented into the webview or by simply using the `showInputBox`. The implementation details for each are as follows:

Approach 1: Using webview view

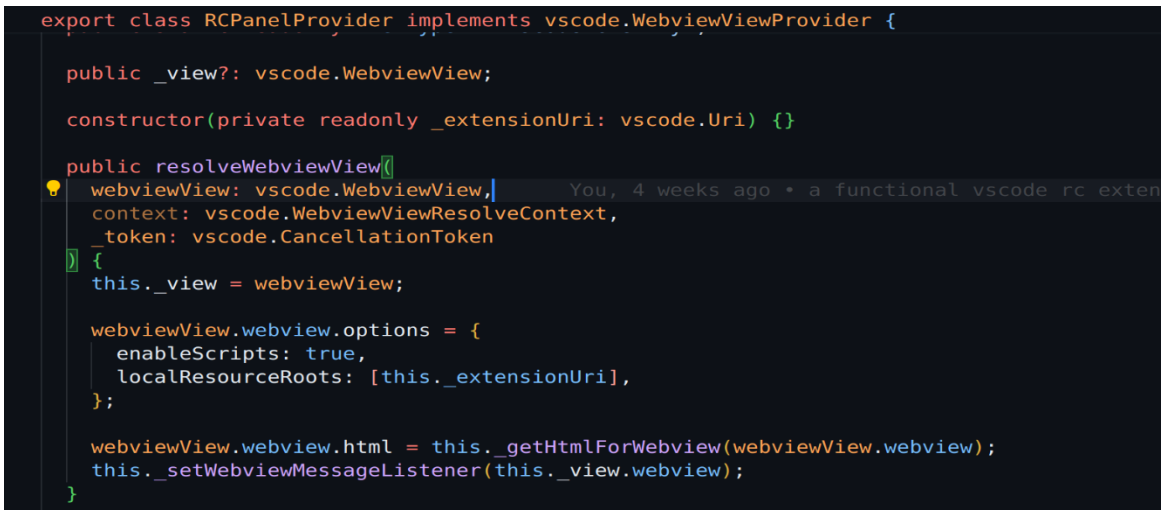

Fig. 20: Setting up webview in the sidebar (Webview View)

```
  private _getHtmlForWebview(webview: vscode.Webview) {
    const webviewUri = getUri(webview, this._extensionUri, [
      "out",
      "webview.js",
    ]);
    const cssStyles = getUri(webview, this._extensionUri, [
      "media",
      "index.css",
    ]);
    const nonce = getNonce();
```

Fig. 21: Setting up required CSS and script files.

```
import {
  provideVSCodeDesignSystem,
  vsCodeButton,
  Button,
  vsCodeTextField,
  TextField,
} from "@vscode/webview-ui-toolkit";

provideVSCodeDesignSystem().register(vsCodeButton(), vsCodeTextField());
```

Fig. 22: Setting up VSCode UI Toolkit to make webview elements adapt to different themes in the VSCode

Now, once the webview has been set up according to the documentation and UI toolkit guide, we can design our simple user interface for login using the components provided by the UI toolkit. Here is a simple implementation of a login page using the suggested components:

```
return /*html*/ `<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>RC VSCode Extension</title>
    <link href="${cssStyles}" rel="stylesheet">
</head>

<body>
    <div id = "login-page">
      <h1>Welcome to Rocket.Chat VSCode Extension</h1>
      <h2>Login</h2>
      <form id="login-form">
          <vscode-text-field id="username" size="50" placeholder="example@example.com">Email or username
              *</vscode-text-field>
          <vscode-text-field id="password" size="50" type="password">Password *</vscode-text-field>
          <vscode-button id="login-btn">Login</vscode-button>
      </form>
    </div>
```

Fig. 23: Implementation of a basic login UI

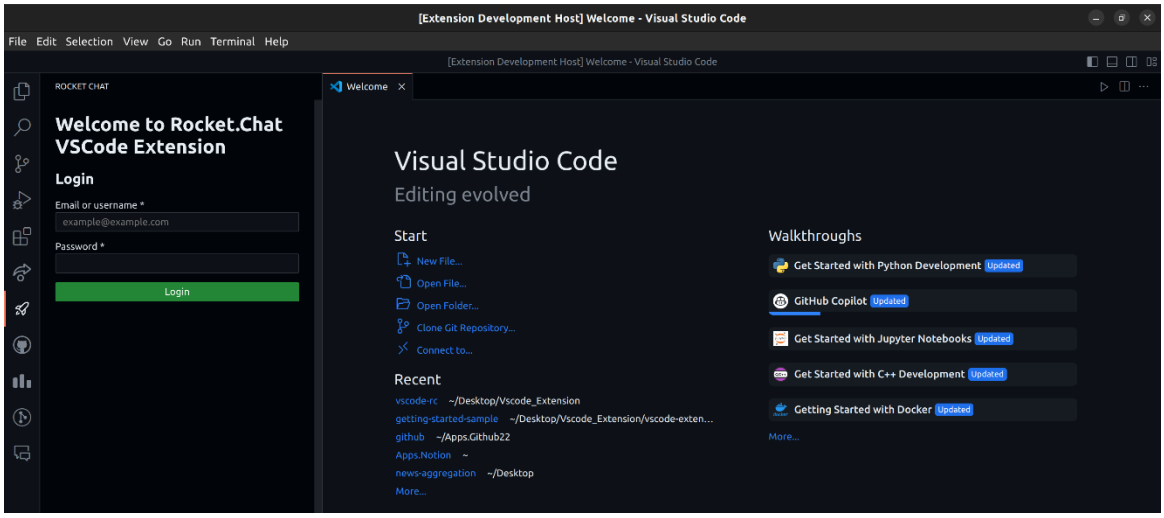

Fig. 24: Login form in Webview View allowing user to Sign in

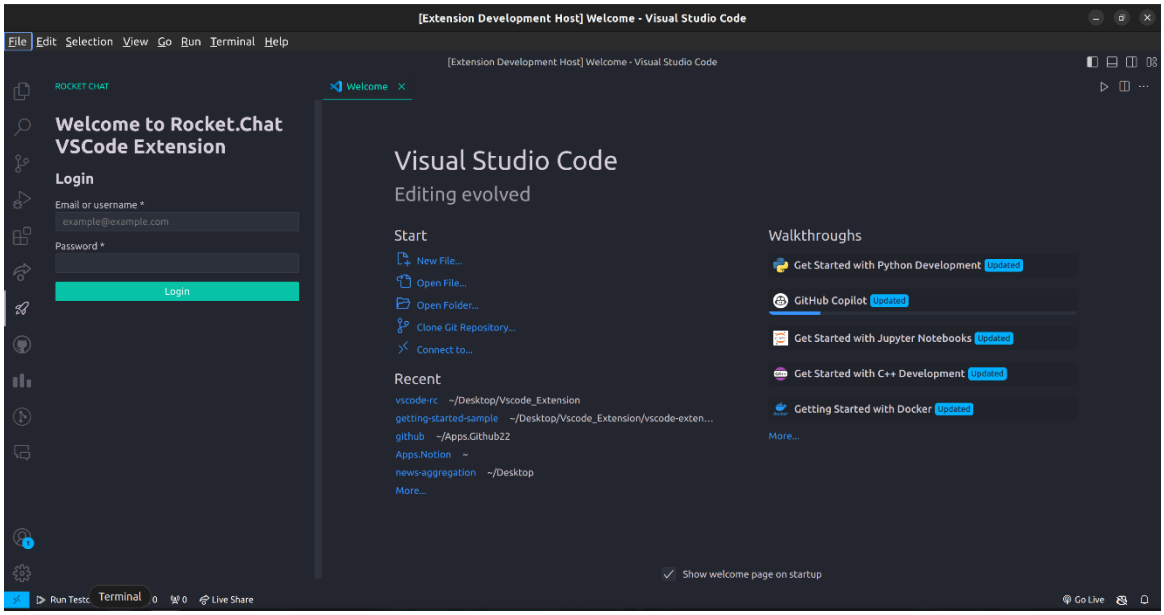

Fig. 25: Login page adapting to VSCode themes due to the use of UI-Toolkit

Once the UI for login is ready, the functional logic must be written, so the flow of it will be as follows:

i. As soon as the webview window is loaded, register an event listener on the login button.

ii. When the user inputs their credentials and clicks on the login button, the handleLogin() function is triggered. This function utilizes the message passing mechanism provided by VSCode to pass the credentials to the extension from webview.

iii. When the webview message listener receives this request, it calls the required function to handle the login by making requests to the Rocket.Chat REST API endpoint.

iv. Upon successful login, display a success information message and save the authToken and userId, which will be used for making other calls.

The code snippets for the implementation are as follows:

```
window.addEventListener("load", main);

function main() {
  const loginBtn = document.getElementById("login-btn") as Button;
  loginBtn?.addEventListener("click", handleLogin);
```

Fig. 26: Registering an event listener on login button

```
async function handleLogin() {
  let username = (document.getElementById("username") as TextField)?.value;
  let password = (document.getElementById("password") as TextField)?.value;
  const credentials = {
    user: username,
    password: password,
  };

  vscode.postMessage({
    method: "login",
    data: credentials,
  });
}
```

Fig. 27: Extracting the user credentials and using message passing to pass information from webview to extension

```
private _setWebviewMessageListener(webview: vscode.Webview) {
  webview.onDidReceiveMessage(async (message: any) => {
    if (message.method) {
      const method = message.method;
      const requestData = message.data;

      switch (method) {      You, 4 weeks ago • a functional vscode rc extension
        case "login":
          const res = await apiClient.handleLogin(requestData);
          if (res.status === "success") {
            vscode.window.showInformationMessage("Login Successful !");
            AuthData.setAuthToken(res.data.authToken);
            AuthData.setUserId(res.data.userId);
```

Fig. 28: Calling the handleLogin function of apiClient which makes requests to REST API endpoints

Fig. 29: Implementation of `handleLogin` function which is making calls to Rocket.Chat login endpoint

Note: Webviews are only to provide the entire chat functionality if the user wants inside VSCode; the actual features that are useful for developers are mainly focused on code discussions. This discussion for setting up the webview view for login functionality makes a base to further work on making the entire chat functionality with webview within the VSCode interface that will be explained further in the section.

There are also other simple ways to implement login like simply using showInputBox and then calling the handleLogin function of apiClient, shown as follows:

Approach 2: Using VSCode input box: This is a simple and straightforward approach. The user can execute the login command, and they will be prompted with a textbox asking for a username and then a password. Once both are entered, we will use our existing `apiClient`, as discussed earlier, to make a login request and save it into `AuthData`, as shown earlier. Here is a implementation of same with demo images:

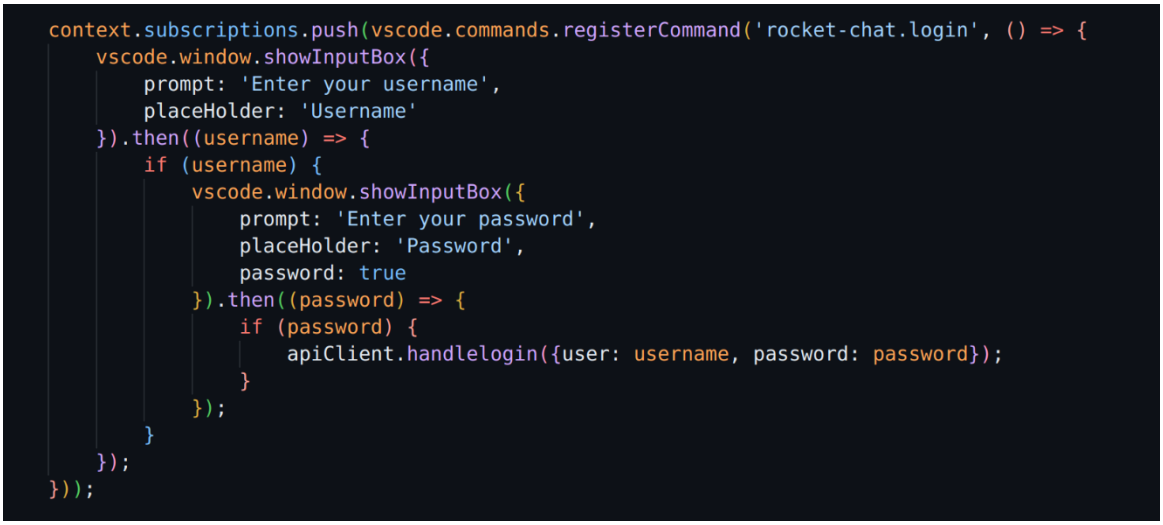

Fig. 30: Implementation of login functionality using input boxes

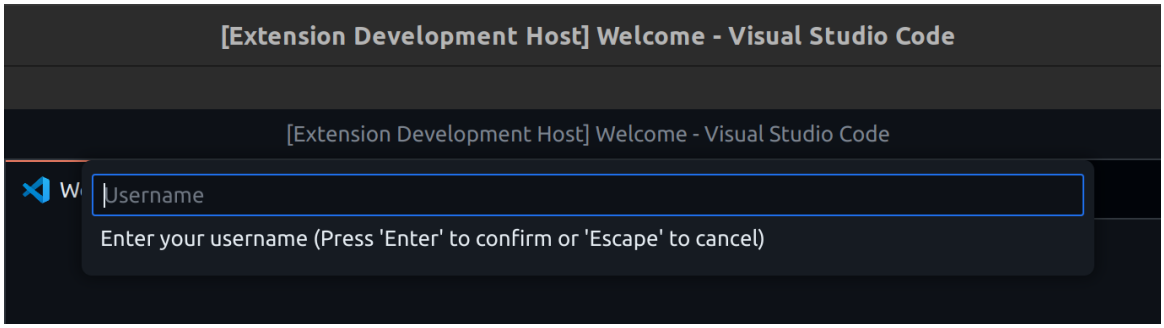

Fig. 31: Prompt input box asking for username

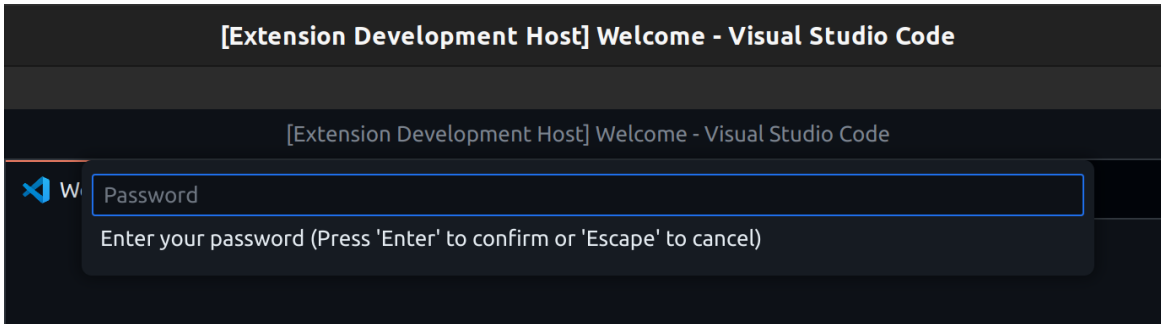

Fig. 32: Prompt input box asking for password

3. Implementing Channel / Contact List using Tree View – In the code discussion feature implementation I showed earlier, the channel was hardcoded to "GENERAL," to send messages to, but that should not be the case in the actual implementation. The plan is to provide a tree view where the extension user can see the list of all channels/teams/discussions, and the user will have the flexibility to choose from those channels for both general chat functionality as well as for code discussions in the editor.

To implement the following, the following steps must be followed:

i. Register a view `channelList` in the sidebar.

ii. Fetch all the channel lists of the workspace using the REST API endpoint `/api/v1/channels.list`

iii. Implement a class with `getTreeItem` and `getChildren` methods for the `vscode.TreeDataProvider interface`.

iv. Using the implemented class, utilize `vscode.window.createTreeView` to register the tree into the required view specified in `package.json`.

```
  "views": {
    "vscode-rc-poc": [
      {
        "type": "webview",
        "id": "vsCodeRc.entry",
        "name": "Rocket Chat"
      },
      {
        "id": "channelList",
        "name": "Channels"
      }      You, 4 weeks ago • a functional vscode rc extension
    ]
```

Fig. 33: Creating another view to show `Channel Lists`

```
class TreeItemNode extends vscode.TreeItem {
    constructor(
        public readonly label: string,
        public readonly collapsibleState: vscode.TreeItemCollapsibleState,
        public readonly command?: vscode.Command
    ) {
        super(label, collapsibleState);
    }
}

export class TreeDataProvider implements vscode.TreeDataProvider<TreeItemNode> {
    private _onDidChangeTreeData: vscode.EventEmitter<TreeItemNode | undefined | null | void> = new
    readonly onDidChangeTreeData: vscode.Event<TreeItemNode | undefined | null | void> = this._onDi

    getTreeItem(element: TreeItemNode): vscode.TreeItem {
        return element;
    }
```

```
  getChildren(element?: TreeItemNode): Thenable<TreeItemNode[]> {
      if (!element) {
          return Promise.resolve([
              new TreeItemNode("General", vscode.TreeItemCollapsibleState.Collapsed),
              new TreeItemNode("GSoC 2024", vscode.TreeItemCollapsibleState.Collapsed)
          ]);
      } else {
          switch (element.label) {
              case 'GSoC 2024':
                  return Promise.resolve([
                      new TreeItemNode("#idea: VSCode extension for Rocket.Chat", vscode.TreeItemCollapsib
                      new TreeItemNode("#idea: embedded chat 2024", vscode.TreeItemCollapsibleState.None),
                      new TreeItemNode("#idea: api documenation generator", vscode.TreeItemCollapsibleStat
                      new TreeItemNode("#idea: AI assistant to help with understanding Rocket.Chat core co
                      new TreeItemNode("#idea: AI in-channel gif image generator", vscode.TreeItemCollapsi

                  ]);
              default:
                  return Promise.resolve([]);
          }
      }
  }
```

Fig. 34: Implementation of Tree View with hardcoded channel names.

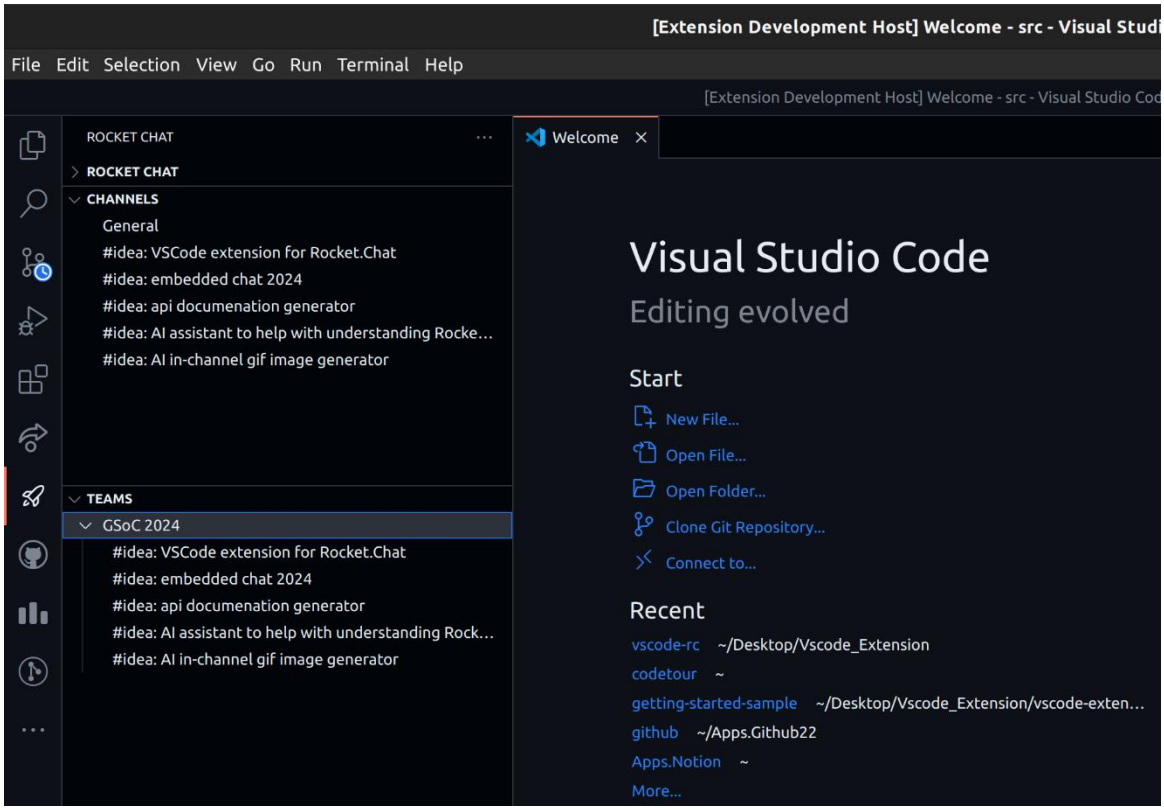

Fig. 35: A sample image showing Channel / Team lists present in the workspace.

Currently, the channels and teams have been hardcoded. In the actual implementation, the channels/teams present in the workspace can be fetched using a REST API endpoint and populated in Tree View.

4. Enable sharing of code line links – This feature is useful for developers working on the same codebase. If one developer wants to inquire about specific lines, instead of copying and pasting the code, they can simply select the lines, generate a line link, and share it via proposed VSCode extension. The other developer can then click on the link. The extension listens for the URI, and when detected, automatically navigates the user to the line of code. This will enable better discussion, overall making the collaboration natural and efficient.

To achieve this, I propose using the GitHub permalink if Git is initialized in the project. This approach would have two purposes: first, enabling individuals who are not using the extension to open the link in a browser and directly view the required code lines from GitHub; and second, for users using the extension, the shared GitHub link can be parsed to open the file with highlighted lines in the code editor. The GitHub permalink follows this format:
`https://github.com/${org}/${project}/blob/${commit_hash}/${file_location}#L${line}`

So, with the help of Git related information about the project, file locations, and line numbers, we can easily create a GitHub permalink to share with another user.
In case Git is not initialized in the project, we can simply have a link in the format `file:///path/to/file.txt#123`, and accordingly, the extension can parse this to navigate the user to the desired location. A dummy code snippet showing that might work is shown:

```
const uri = vscode.Uri.parse(uriString);
const filePath = uri.path;
const lineNumber = parseInt(uri.fragment);

vscode.workspace.openTextDocument(filePath).then((doc) => {
    vscode.window.showTextDocument(doc).then((editor) => {
        const line = doc.lineAt(lineNumber);
        editor.revealRange(line.range, vscode.TextEditorRevealType.InCenter);
    });
```

Fig. 36: Dummy code snippet parsing the URI
and navigating user to desired location

Note: I haven't implemented this feature as of now; hence, the above code is just a dummy demonstration of how it might work.

5. Enable Reviewers to edit code within channels, with suggestions visible in VSCode for acceptance – There are situations where a PR hasn't been raised yet; only a developer is asking for code review in a team channel before raising it. With this extension, we can send code for review, and other users can discuss it. But what if a reviewer needs to suggest changes in the code? How can this be achieved? To solve this issue, I propose that reviewers be allowed to make changes to the code directly from the channel and suggest those changes to the user requesting a review. The extension can then check the type of request; if it's a suggestion, the extension can parse that response and display it as a suggestion block with a button to accept those changes. Once accepted, the existing code will be replaced by the suggested changes.

To implement this feature, the flow will be as follows:

i. Reviewer made changes to the code that were asked for review and sent it as a message.

ii. When a message is received, the extension checks if it's a code markdown. If yes, the extension compares it with the existing sent code.

iii. Then it appropriately highlights the changed part in the code editor using TextEditor.setDecorations(decorationType, rangesOrOptions).

iv. In VSCode comments, add a button Accept Suggestion.

v. When this button is clicked, using the class vscode.WorkspaceEdit and its replace function, replace the text with the suggested changes.

The code snippet below shows possible implementation of the approach:

```
"colors": [
    {
        "id": "rc_poc.oldCode",
        "description": "Background decoration color for oldCode",
        "defaults": {
            "dark": "#ff000086",
            "light": "#ff000086",
            "highContrast": "#ff000086"
        }
    },

    {
        "id": "rc_poc.newCode",
        "description": "Background decoration color for suggested code",
        "defaults": {
            "dark": "#4cb84886",
            "light": "#4cb84886",
            "highContrast": "#4cb84886"
        }
    }
]
```

Fig. 37: Setting highlight colors for old and new code

```
const oldCodeDecorationType = vscode.window.createTextEditorDecorationType({
    backgroundColor: { id: 'rc_poc.oldCode' }
});

const newCodeDecorationType = vscode.window.createTextEditorDecorationType({
    backgroundColor: { id: 'rc_poc.newCode' }
});
```

Fig. 38: Creating text decoration for old and new code

```
    }
    activeEditor.setDecorations(oldCodeDecorationType, oldCode);
    activeEditor.setDecorations(newCodeDecorationType, newCode);
}
```

Fig. 39: Setting highlight colors

Fig. 40: Suggested code change highlighted in the editor

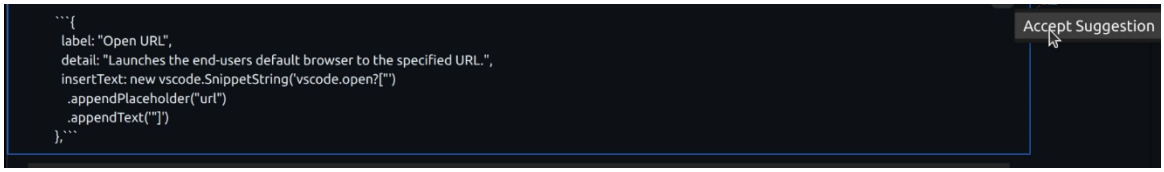

Fig. 41: User having an option to Accept Suggestion

6. Implementing General Chat functionality into Webviews – As the code discussion functionality has already been enabled, there might be cases where developers just want to have some general discussions with the team either on the channel or DMs. One way is to keep the Rocket.Chat app tiled side by side with VSCode. However, when our aim is to allow developers to perform all communication tasks from the VSCode interface itself, why not develop a webview where we can use VSCode UI-Toolkit elements to create our own interface inside the VSCode editor? Hence, a general implementation is demonstrated as follows:

Setting up the required APIs – First, all the required APIs that are responsible for sending messages, getting messages, etc., have to be set up. I have used a class called `RocketChatApi`, in which the required functions such as `getMessage`, `sendMessage`, etc., are defined. These functions call the necessary REST API endpoints.

```
class RocketChatApi {
  private host: string;

  constructor(host: string) {
    this.host = host;
  }
```

Fig. 42: RocketChatAPI class in which required functions are implemented

```
public async getMessage(
  authToken: string | null,
  userID: string | null
): Promise<any> {
  if (authToken && userID) {
    try {
      const res = await fetch(      You, 4 weeks ago • a functional vscode rc ext
        `${this.host}/api/v1/channels.messages?roomId=GENERAL`,
        {
          method: "GET",
          headers: {
            "Content-Type": "application/json",
            "X-Auth-Token": authToken,
            "X-User-Id": userID,
          },
        }
      );
      return await res.json();
    } catch (error) {
      console.error("Error:", error);
    }
```

Fig. 43: Function `getMessage` to fetch the messages on initial load

```
class RocketChatApi {
  public async handleSendMessage(
    authToken: string | null,
    userID: string | null,
    message: any,
    tmid?: any
  ): Promise<any> {
    if (authToken && userID) {
      try {
        const requestBody: any = {
          message: {
            rid: "GENERAL",
            msg: message,
          },
        };
        if (tmid) {
          requestBody.message.tmid = tmid;
        }

        const res = await fetch(`${this.host}/api/v1/chat.sendMessage`, {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            "X-Auth-Token": authToken,
            "X-User-Id": userID,
          },
          body: JSON.stringify(requestBody),
        });

        return await res.json();
      } catch (error) {
        console.error("Send error:", error);
      }
```

Fig. 44: Function `handleSendMessage` to send the messages in the channel

Setting up the Realtime API (WebSocket) – To enable real-time communication, we must set up WebSocket communication between the VSCode extension and Rocket.Chat. To do that, I'm utilizing the class `RocketChatRealtime`, in which the required function is defined. This function accepts a callback function. As soon as a message is sent to the channel from anywhere, it will trigger the callback function, which sends the required message to the Webview through message passing. Once the message is received on the Webview, there is logic to render the same. The implementation of setting up WebSocket communication is shown below.

```
class RocketChatRealtime {      You, 4 weeks ago • a functional vscode rc extension
  constructor(host: string) {
    this.client = new Rocketchat({
      protocol: "ddp",
      host: host,
      useSsl: !/http:\/\//.test(host),
      reopen: 20000,
    });
  }
```

Fig. 45: Defining a class for Realtime Websocket Communication

```
  public async listenMessage(
    token: string | null,
    rid: string,
    callback: any
  ): Promise<void> {
    if (token) {
      try {
        await this.client.connect({});
        await this.client.resume({ token });
        await this.client.subscribeRoom(rid);
        this.client.onMessage((message) => {
          callback(message);
        });
      } catch (err) {
        console.log(err);
      }
    }
  }
}

export { RocketChatRealtime };
```

Fig. 46: Setting up the WebSocket communication which triggers a callback function when a new message is received

UI and message rendering on Webview: As soon as the login completes, the chat interface will appear where the user can chat with their teams. The basic implementation of the chat interface and the logic to render messages can be done as follows:

```
    <div id = "chat-container">      You, 4 weeks ago • a functional vscode rc extension
        <div id="message-view-box">
        </div>
        <div class="message-input-container">
            <vscode-text-field id="msg-input" size="50" placeholder="Type your message ..."
                ></vscode-text-field>
            <vscode-button id="send-btn">Send</vscode-button>
        </div>
    </div>
    <script type="module" nonce="${nonce}" src="${webviewUri}"></script>
```

Fig. 47: Basic UI Implementation for Chat interface

```
  const sendBtn = document.getElementById("send-btn") as Button;
  sendBtn?.addEventListener("click", () => {
    handleSendMessage();
  });

  window.addEventListener("message", (event) => {
    const vscodeMessage = event.data;
    if (vscodeMessage.messages) {
      renderMessage(vscodeMessage.messages);
    }

    if (vscodeMessage.realTimeMsg) {
      renderMsgRealtime(vscodeMessage.realTimeMsg);
    }
  });
}
```

Fig. 48: Setting up the event listener to send messages and render incoming messages.

```
async function handleSendMessage() {
  const msgInputBox = document.getElementById("msg-input") as TextField;
  const msgValue = msgInputBox.value;

  vscode.postMessage({
    method: "sendMessage",
    data: msgValue,
  });

  msgInputBox.value = "";
}
```

Fig. 49: When `Send` button is clicked, informing to extension via message passing to call required API to send message

```
async function renderMessage(messages: any) {
  document.getElementById("login-page")!.style.display = "none";
  document.getElementById("chat-container")!.style.display = "flex";
  const messageViewBox = document.getElementById("message-view-box");
  messageViewBox!.innerHTML = "";
  messages.forEach((message: any) => {
    const newMsgTemplate = `
      <div class="message-body">
        <div class="username"><b>@${message.u?.username}</b></div>
        <div class="message-content">${message.msg}</div>
      </div>`;

    if (messageViewBox) {
      messageViewBox.innerHTML += newMsgTemplate;
    }
  });
}      You, 4 weeks ago • a functional vscode rc extension
```

Fig. 50: Logic to render message for the first time after login

```
async function renderMsgRealtime(message:any) {
  const messageViewBox = document.getElementById("message-view-box");
  const newMsgTemplate = `
  <div class="message-body">
    <div class="username"><b>@${message.u?.username}</b></div>
    <div class="message-content">${message.msg}</div>
  </div>`;
  if (messageViewBox) {
    messageViewBox.innerHTML += newMsgTemplate;
  }
}
```

Fig. 51: Logic to render messages in Realtime when new messages are received

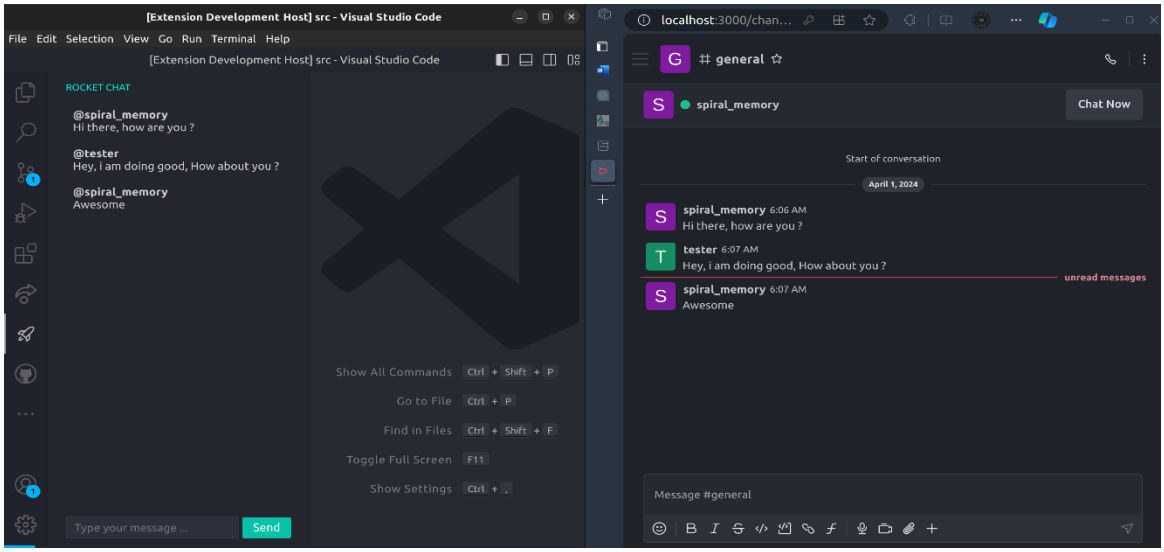

Fig. 52: A sample image showing duplex communication between VSCode client and Rocket.Chat Server

7. Documentation – As the proposed VSCode extension for Rocket.Chat offers many features, such as support for various login methods and chat functionality through Webview and the VSCode comment API, as well as reflecting code changes made in channels to VSCode etc., it becomes difficult for developers/users to experiment and explore to understand everything. Therefore, I will keep the following things in mind:

i. I will thoroughly maintain the work and create walkthroughs by using `walkthrough: [{"id": "rocketchat-walkthrough", ... }]` array in `contributes` in `package.json`, along with necessary images, so that developers can set up this extension and understand all required features without any hassle.

ii. I will address any frequently asked questions that may arise while configuring the VSCode extension for Rocket.Chat.

iii. I will ensure that the documentation is regularly updated to reflect any changes or additions made to this extension, ensuring users have access to the latest information.

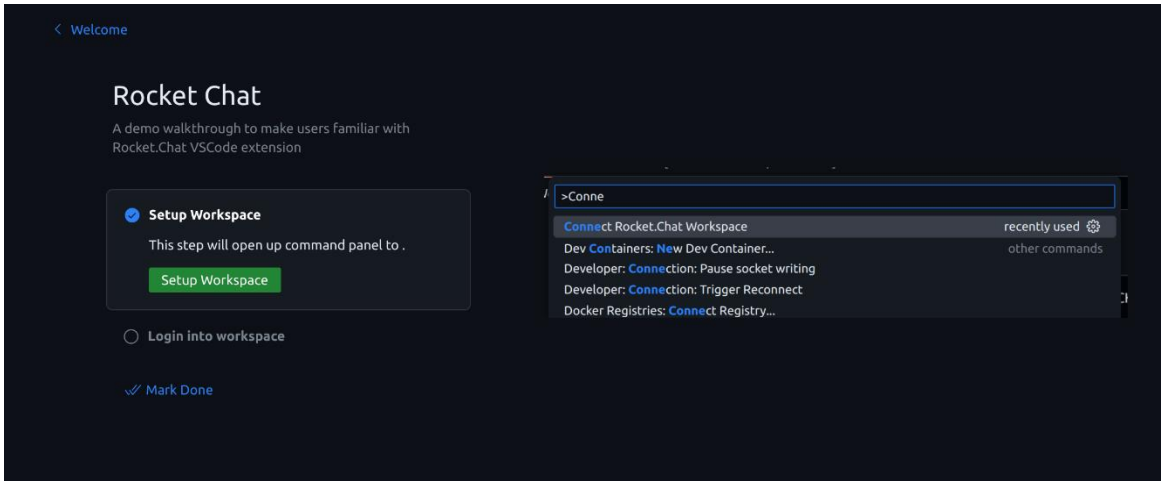

Fig. 53: A walkthrough for users to connect with their Rocket.Chat workspace

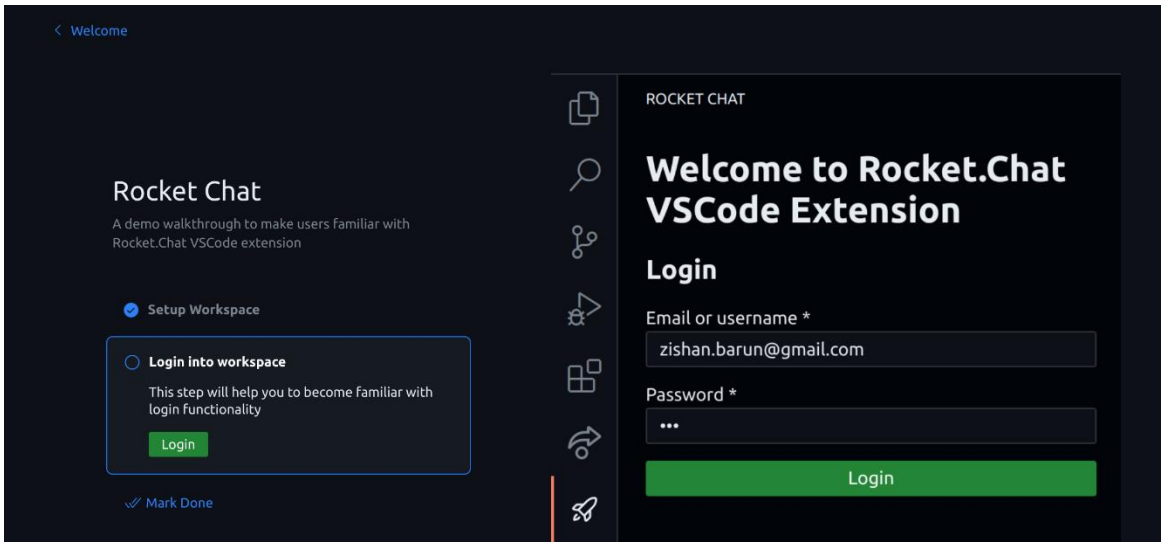

Fig. 54: Additional steps assisting users in logging into the workspace.

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 other open-source VSCode extensions and discuss with 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.
- Implementing login functionality through Rocket.Chat OAuth methods if they are enabled in the workspace.

- Start working on the login functionality through the command panel by prompting for the workspace URL and user credentials. (If required)
- Work on implementing OAuth logins from alternate providers. (If required)

Week 2 [June 03 – June 09]

- Work on setting up the `Webview view` and implementing the login functionality here as well. (If required)
- Implement `Tree View` to showcase the channels, contact list, etc., in the sidebar of VSCode.

Week 3 [June 10 – June 16]

- Start to work on having feature to discuss regarding code directly from editor using VSCode comment API.
- Implement duplex communication between Rocket.Chat and VSCode extension through REST APIs and WebSocket.
- Adding various features to this comment API such as the ability to edit messages, delete messages, pin messages, etc.

Week 4 [June 17 – June 23]

- Work on making these comments persistent by storing the metadata of selected code into a JSON file, including line numbers, thread ID, etc.
- Begin exploring ways to integrate VCS into the extension.

Week 5 [June 24 – June 30]

- Work on making these comments resistant to code changes.
- Use the commit hash as an identifier along with line numbers, and later compare the current changes with that snapshot to adjust the line numbers, accordingly, showing the comment at the correct place.

Week 6 [July 01 – July 07]

- Enhance the VSCode comments by adding appropriate thread titles and listing participants who have discussed the code.
- Implement a partitioning mechanism to enable users to communicate on same code lines with multiple channels/DMs by providing tab-like options in VSCode comments.

Midterm Evaluation

July 08- July 12

- Work on drafting a report for midterm evaluations.
- Submit the midterm report.
- Start working on extending the code discussion feature further.

Coding continues!

Week 7 [July 13 – July 19]

- Start working on the feature to share line links / blame links from VSCode editor directly.
- Listening to editor selection and generate required link for the code lines and files.
- Adding ability to share this link to Rocket.Chat channel.
- Set up a URI listener and parse it accordingly so that the extension will navigate other users to the exact code file and lines.

Week 8 [July 20 – July 26]

- Start working on making the code directly editable from the Rocket.Chat channel/DMs.
- Send the suggested code back to VSCode with markdown so that the extension can recognize it as a suggestion.
- If required, to enable a better code editing interface in Rocket.Chat, create an RC app that can allow editing in the UI-Kit modal and configure the RC app to send those changes to the VSCode extension.

Week 9 [July 27 – August 2]

- Create an endpoint in the extension that will listen to such requests.
- Start working on the ability to show suggestions and highlight the code lines where the reviewer has suggested changes.

Week 10 [August 3 – August 10]

- Allowing users to accept the changes to reflect them in the VSCode editor.
- Allow users to further communicate about those suggested changes.

- Address any bug fixes, pending tasks, etc.

Week 11 [August 11 – August 18]

- Implement usual chat functionality in the `Webview view` (sidebar) by using REST APIs and Realtime APIs (WebSocket) to allow developers to use the complete functionality of Rocket.Chat without switching from VSCode to a browser.
- Work on connecting the Tree View Channel/Contact List event with the `Webview` so that if a user chooses a channel/contact, they can be redirected to the `Webview` for chat.
- Test all scenarios to confirm that all edge cases are handled correctly

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 various repositories of Rocket.Chat since January, including Rocket.Chat, Apps.Notion, and EmbeddedChat. Thus, I have familiarized myself with the Rocket.Chat REST API endpoints, RC app structure, its code flow, as well as its functionalities. While contributing to the Apps.Notion and EmbeddedChat repositories, I have already made strides towards learning about different Rocket.Chat REST API endpoints, setting up web sockets to enable real-time communication, etc. In addition, I have a good understanding of UI-Kit, Preview Blocks, RC App Settings, etc. I have also worked on integrating the UI-Kit into EmbeddedChat, giving me a solid understanding of how it works, which can be helpful in implementing some of the features of this proposed extension.

In addition to these related PRs, I also attempted to develop prototypes for major features in the VSCode extension for Rocket.Chat, showcasing my ability to develop extensions. This effort is aimed at enabling me to develop a fully functional extension with minimal effort during the GSoC period, without spending excessive time understanding the problem.

Here is a list of some related PRs I have submitted:

1. **[MERGED]** EmbeddedChat #401 - **[FIX]** TOTP Modal and Toast Message Display.
2. **[MERGED]** EmbeddedChat #431 - **[FIX]** Message fetch issue in private channels.
3. **[MERGED]** EmbeddedChat #442 - **[ENHC]** Thread menu to access all thread msgs.
4. **[MERGED]** EmbeddedChat #480 - **[ENHC]** Added user-mention menu.
5. **[MERGED]** EmbeddedChat #481 - **[ENHC]** **[EPIC]** UI-Kit Modal Support.
6. **[MERGED]** EmbeddedChat#530 - **[FIX]** Static Message Limit.
7. **[MERGED]** Apps.Notion #48 - **[ENHC]** Overflow menu to go to subpage modal.
8. **[MERGED]** Apps.Notion #53 - **[FIX]** Title input visibility issue.
9. **[MERGED]** Rocket.Chat #31507 - **[FIX]** Can't remove the channel's join password.

The code for my prototype is available on my GitHub repository, and a video demonstration for each of the developed POCs is available here.

In addition, a detailed explanation of all the features that I am proposing, along with their implementation details, is provided in the Implementation Details section.

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 with the tech stack, including Node.JS and React.JS, through my personal projects and contributions to open-source projects, demonstrating a strong understanding of JavaScript/TypeScript. This is evident from my contributions to the EmbeddedChat repository, where I have contributed more than 2.5K lines of code. I have made various UI modifications to ensure design consistency, and I have also made changes to several pieces of code related to understanding and handling Rocket.Chat REST API endpoints, giving me a deep understanding of how things work in Rocket.Chat. This knowledge will be very helpful in this project as well because it will involve tasks such as setting up login functionality, fetching channels and users, and enabling chat functionality by setting up Realtime API using the Rocket.Chat SDK.

In addition, I have regularly attended the RC-app development workshop, which has helped me learn to develop RC apps from scratch. Moreover, I have worked on the Notion RC app and implemented features with over 500 lines of code, demonstrating my good understanding of various event listeners, webhooks, and the UI-Kit. I have added support for UI-Kit modal rendering in EmbeddedChat, which has provided me with a solid understanding of its functioning. This knowledge will help me develop a compatible RC app to extend the feature of code editing from the Rocket.Chat channel, by providing a feature-rich code editor directly within Rocket.Chat.

Furthermore, I have created enough prototypes to provide proof of concept to support my claims and showcase my ability in developing VSCode extensions with the required features. I have a good understanding of TypeScript, VSCode extension development, and various Extension APIs it offers, as well as Rocket.Chat app development, making me a suitable candidate for this project.

I have also 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.