

## Introduction

### About me

I am **Abhinav Kumar**, an Information Technology undergraduate at Heritage Institute of Technology, Kolkata. Since school, I have always been interested in computers and software. I was introduced to application development in high school. Since then, I have kept learning more about web technology and application development. I have done a few personal projects from which my favorites are “tournify.in” and “esportsweb.in”. They are an esports management platform and have over 150,000 registered users combined. I love working on javascript projects, but not limited to, those that solve problems and make our lives easy.

Website Link: <https://avitechlab.com>

Linkedin: <https://linkedin.com/in/abhinavkrin>

### Why do I wish to participate in the Google Summer of Code?

Although I have developed my personal projects and felt a sense of achievement but looking at the open-source projects, I feel a lot can be improved in my style. I want to make myself more aware of the open source standards and the way of writing code like a poem. I have always dreamed of being a part of a community and an open-source community is just perfect for me.

I have always wanted to be part of such communities and meet people with whom I can learn and share my ideas. And I have seen people who have been accepted into GSoC become active members of the community and keep contributing to it. Therefore, GSoC is the best opportunity to join open-source communities and contribute and learn from them.

### Why do I want to apply to RocketChat in particular?

I came across RocketChat while I was browsing the accepted organizations in previous years of GSoC. Having the same tech stack, in which I was experienced, pushed me to learn more about RocketChat. I have always wanted to contribute to projects widely used by folks around the globe. Finding that RocketChat is one of the most used open-source communication platforms, I decided to contribute to RocketChat. After I joined RocketChat's open community website, I loved how everyone was so welcoming and supportive.

I spent hours learning about RocketChat projects and contributed to RC4Community and EmbeddedChat. Everyone applauded for successful PR merges, helped me when I was stuck, and gave me valuable feedback which helped me to grow.

Working on such a large project in RocketChat, I learned about teamwork, problem-solving, debugging, and effectively communicating with fellow community members. I am excited to contribute to RocketChat and be a valuable member of the community. Therefore, I have decided to apply only for RocketChat for GSoC 2023.

# Project Description

## Title: EmbeddedChat 2023

Integrate real-time chat into web and mobile applications that adapt to the application's theme with the power of RocketChat.

## Abstract

The EmbeddedChat project enables developers to embed real-time chat into their website or application. It utilizes RocketChat's chat engine through its REST and real-time APIs to support powerful chat features like reactions, online presence, typing status, threads, and much more. Currently, the EmbeddedProject is very basic and unoptimized for production. I plan to make it production ready so that thousands of developers and communities could start integrating their web and mobile applications with RocketChat.

This could be achieved by:-

- Separating various parts of the project into their module/package - API, react, react native, auth, etc.
- Reducing the bundle size. Currently, it is very large.
- Replacing HTTP calls with real-time DDP communication wherever necessary.
- Creating libraries for react-native, react, etc.
- Giving more flexibility in authentication methods.
- Introducing theming for client libraries so that EmbeddedChat could adapt to the look and feel of the application it is embedded into.
- And much more which I will discuss in the goals and deliverables.

## Benefits to the community

Enabling chat in applications be it an online game or an online club or an online food ordering application brings a nice user experience. But making a fully-fledged chat feature could be a challenging and time-consuming task. It would require significant time and energy for the developers to build it and include it in the application. However, this time and energy could be saved if we use EmbeddedChat to implement the chat feature in the application. Since EmbeddedChat uses RocketChat, developers would be assured of its robustness, security, and scalability. They could focus more on building the main application and leave the chat feature to be handled by EmbeddedChat.

An increase in the popularity of EmbeddedChat would indirectly contribute to the popularity of the RocketChat project.

# Goals and Deliverables

## 1. Improve authentication - support all OAuth services

Currently, only Google authentication and password authentication are supported in EmbeddedChat. It should support all OAuth services as it has been supported in RocketChat. Also, the login flow is currently handled by EmbeddedChat. This would require the user to login in two times - one for the application level and another for the EmbeddedChat. This contradicts our goal to include EmbeddedChat as a part of the application. There should be one single login flow for authenticating both the application and the EmbeddedChat. So, I will add an additional option in EmbeddedChat for authentication by providing the `access_token` from the OAuth services.

## 2. Move to a mono repo - API, react, react-native, HTML-embed, server

One of the goals of this project would be to separate parts of EmbeddedChat into their own repository. They all together will form a single mono repo. The initial plan is to separate auth, API/SDK, react components, etc into their own repo. Separating them would make the task of creating new libraries for other frameworks like - ReactNative, Angular, Vue, etc.

## 3. HTML embed feature

We will create js bundles which would be used to embed the EmbeddedChat by directly pasting the HTML code into the webpage. This would make the EmbeddedChat framework agnostic where not much configuration is required or the user is not very tech-savvy.

## 4. Theming

We will introduce powerful theming into the client libraries of EmbeddedChat so that it adapts to the look and feel of the application/website.

## 5. Improving API

I will work on improving the current API implementation of the EmbeddedChat which includes but is not limited to:-

- Moving to real-time DDP connection wherever possible.
- Separating authentication library to include more and more authentication methods and strategies supported by RocketChat.

## 6. Slash commands

Slash commands are one of the many powerful features of RocketChat, but EmbeddedChat is yet to support them. I will work on including them in EmbeddedChat by leveraging the slash commands API.

## 7. Reducing bundle size

To make EmbeddedChat production ready, we need to keep the bundle size of the EmbeddedChat as small as possible. I will analyze EmbeddedChat to find out the issues that are adding to the huge bundle size of EmbeddedChat. Then, I will start working on reducing the bundle size.

## 8. Moderation

Currently, only the owner of a message could delete it in EmbeddedChat. But, we need some kind of moderation to keep the chat box environment healthy and prevent spam and abuses. I will leverage the permissions feature of RocketChat to give moderation privileges, like deleting messages and muting and removing users, to the concerned members of the chat room.

## 9. Docs and tutorials

I will work on creating docs and tutorials for EmbeddedChat so that users and contributors find it easy to get started with EmbeddedChat.

## 10. Replacing playground with Storybook

Currently, the only way to try EmbeddedChat while developing is the playground create-react-app. I and my fellow contributors have found it to be slow. In addition to that, there seems to be some issue with the playground app as I found that it has grown to 10Gib and keeps on growing. For the client libraries like react, vue, angular, etc we could StoryBook. With Storybook, we could build, document, and test our UI components easily and fastly.

## 11. Comment Mode

This is an experimental idea. The goal is to use EmbeddedChat in comment mode. That is, it will work as the commenting system of the application.

# Implementation Details

## Improving authentication

RocketChat supports several authentication methods/strategies - password-based, popular OAuths, and custom OAuths. And we need to leverage this feature of RocketChat to support a variety of authentication options in EmbeddedChat as well. Currently, EmbeddedChat supports only two methods - password-based and Google OAuth.

Based on the use-case of EmbeddedChat we could support these types of authentication flow:-

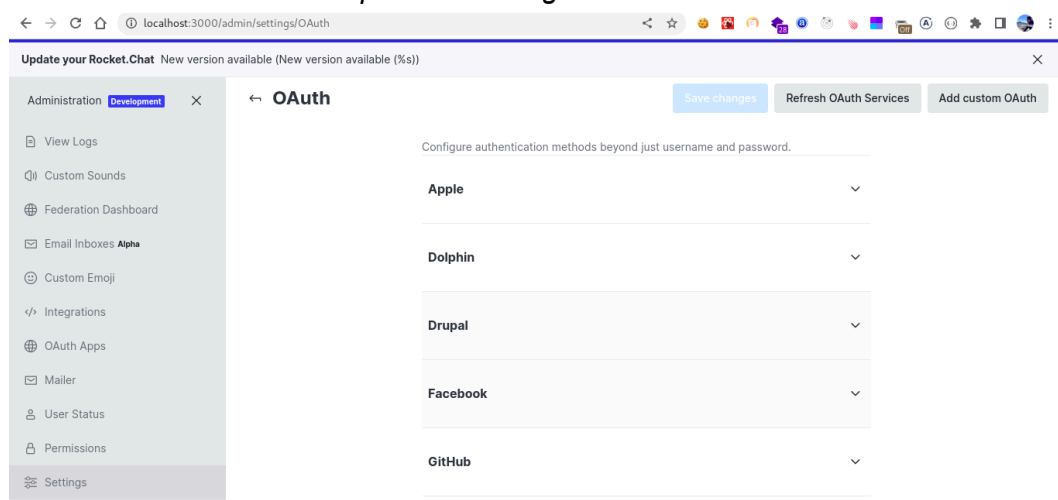
1. The first flow in which the developer will just pass the **accessToken** and the **serviceName** to which the access\_token is related. EmbeddedChat will use them to call POST **/api/v1/login** to get the rc\_token and user details. Let's call this flow **TOKEN FLOW**.
2. The second flow in which EmbeddedChat will display the login options it gets on fetching **api/v1/settings.oauth** and handle the authentication according to the option selected by the user. Let's call this **MANAGED FLOW**.

## TOKEN FLOW

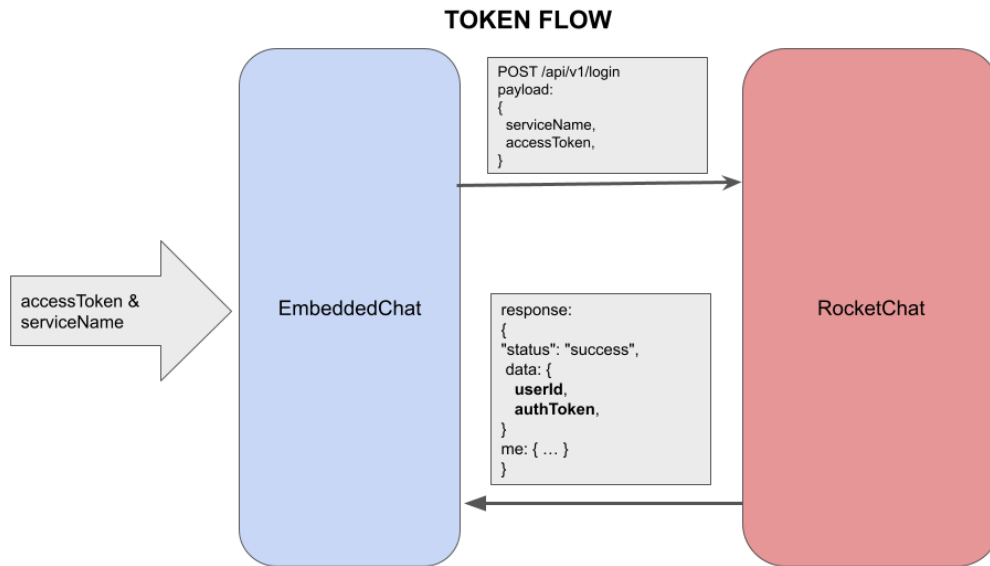
In this flow, the developers will directly pass the **accessToken** and **serviceName** to the EmbeddedChat as props. EmbeddedChat will use these data to call **/api/v1/login** endpoint to authenticate the user and retrieve the **authToken** and **userId**.

Here, **serviceName** is the name of the OAuth service which we have configured in the RocketChat setting.

*RocketChat Home > Workspace -> Settings -> OAuth*



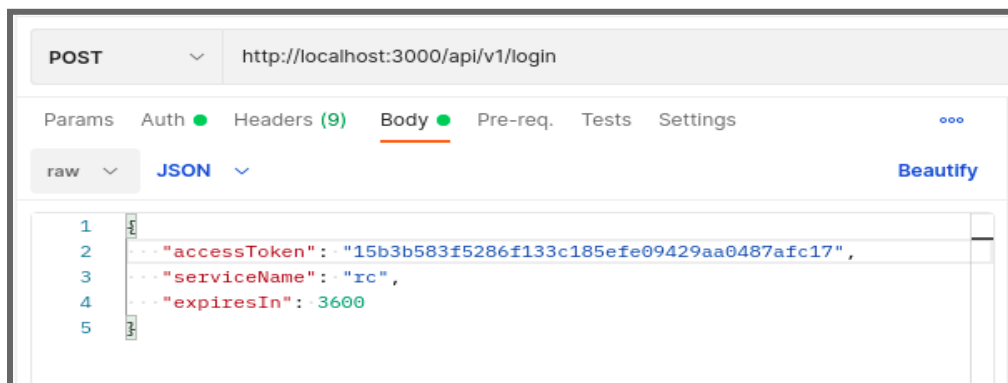
Here, **service names** are apple, dolphin, drupal, etc. These OAuth services must be configured and enabled before using them.



First, the developer will pass the authentication config to EmbeddedChat via props. This should look somewhat like this:

```
<EmbeddedChat auth={{
  flow: "TOKEN",
  credentials: {
    "accessToken": "cbd5b16f76217089b99a337726d56cb0abb4d520",
    "serviceName": "my-custom-oauth",
    "expiresIn": 3600
  }
}}>
```

Then EmbeddedChat will use this data to make a POST request to /api/v1/login with the following payload



On success, RocketChat will send `authToken` along with the user details. The successful response will have the following payload.



We will be using the **userId** and **authToken** for calling authenticated RocketChat APIs.

Benefits of Token flow:

The token flow will be helpful when we would want EmbeddedChat to work as a part of the application because we would not want our application's users to log in again into chat while they are already logged in to the application.

For example, I have an app called Todo List and then I can configure the same Google OAuth configuration for both my app and the EmbeddedChat. In that way when a user signs in to my app using Google OAuth, I can reuse the `accessToken` to sign into the EmbeddedChat using TOKEN flow. Hence, the user won't have to log in again and have a seamless experience.

## MANAGED FLOW

As the name suggests, this authentication flow would be fully managed by EmbeddedChat. It will fetch all the OAuth authentication services enabled on RocketChat and display them to the user. Once the user selects the desired option, he would be redirected to the OAuth service's authorization page and handle the authorization response accordingly to authenticate the user in EmbeddedChat.



```
<EmbeddedChat auth={{ flow: "MANAGED" }}>
```



Step 1 will be to fetch active OAuth services from RocketChat by making a GET request to `/api/v1/settings.oauth`. The response payload will look something like this:-

```
{
  "services": [
    {
      "_id": "HvBRozuyzioBaJZ2a",
      "name": "google",
      "clientId": "-----",
      "buttonLabelText": "",
      "buttonColor": "",
      "buttonLabelColor": "",
      "custom": false
    },
    {
      "_id": "CGj9okJBDpyFpvnBd",
      "service": "mycustomoauth",
      "accessTokenParam": "access_token",
      "authorizePath": "/oauth/authorize",
      "avatarField": "avatar",
      "buttonColor": "#1d74f5",
      "buttonLabelColor": "#FFFFFF",
      "buttonLabelText": "",
      "channelsAdmin": "rocket.cat",
      "channelsMap": "{\n\t\"rocket-admin\": \"admin\", \n\t\"tech-support\": \"support\"\n}",
      "clientId": "",
      "custom": true,
      "emailField": "email",
      "groupsClaim": "groups",
      "identityPath": "/oauth/userinfo",
      "identityTokenSentVia": "default",
      "keyField": "email",
      "loginStyle": "popup",
      "mapChannels": false,
      "mapChannels": false
    }
  ]
}
```

We can see that in the response payload, we have the **clientId**, **service**, **serverUrl**, and **authorizePath**. Now, these services will be displayed to the users and they will have to choose the service they want to log in with. Suppose from the above example, the user selects **mycustomoauth** to log in with. Then, we will create the authorization URL which will be opened in a pop-up.

The authorize url will be of the format

**`{serverUrl}+{authorizePath}?client_id={clientId}&redirectUri={rocketchatUrl}/_oauth/{service}&response_type=code&response_mode=form_post&state={state}`**

- **serverUrl** - This is the base URL of the OAuth service server.
- **authorizePath** - the path of the page on the OAuth service where users will get a consent screen, and will be asked to authorize the EmbeddedChat to get the authorization code.
- **clientId** - the client id of the OAuth service.
- **redirectUri** - the URL where the user will be redirected with the **authorization code** on the successful authorization. For RocketChat it is always `<rocketchat_server_base_url>/_oauth/<serviceName>`. One can also find the redirectUri for the service in the *RocketChat Home > Workspace -> Settings -> OAuth -> serviceName* page. When setting the OAuth Provider, this **redirectUri** must be included in the Callback URLs list in the OAuth provider's setting on their server.

- **state** - The primary reason for using the state parameter is to mitigate CSRF attacks by using a unique and non-guessable value associated with each authentication request about to be initiated. That value allows us to prevent the attack by confirming that the value coming from the response matches the one we sent. The state parameter can also be used to encode any other information in it.

For the current example service, **mycustomoauth**, it will be

```
http://localhost:3000/oauth/authorize
?client_id=P7XsQzmkNG7KBg2Zs
&redirect_uri=http://localhost:3000/_oauth/mycustomoauth
&scope=openid
&response_type=code
&state=aar8q6rsowq
```

The user will then be taken to the consent page in a popup. After authorization, the user will be redirected within the popup to

[http://localhost:3000/\\_oauth/mycustomoauth?code=<authorization\\_code>&state=<state>](http://localhost:3000/_oauth/mycustomoauth?code=<authorization_code>&state=<state>).

If we dig into the HTML of the [http://localhost:3000/\\_oauth/mycustomoauth](http://localhost:3000/_oauth/mycustomoauth), we will find that it contains **credentialToken**, and **credentialSecret**.

```
view-source:localhost:3000/_oauth/mycustomoauth?code=a0e4c61c32b987k
line wrap ☒
1 <!DOCTYPE html>
2 <html>
3 <body>
4   <p id="completedText" style="display:none;">
5     Login completed. <a href="#" id="loginCompleted">
6       Click here</a> to close this window.
7   </p>
8
9   <div id="config" style="display:none;">
10    {"setCredentialToken":true,"credentialToken":"aar8q6rsowq","credentialSecret":
11    "QNIIVLda93EkkwMQ9dxK_VXRiz-
12    TiqSggnuuHAWYxX1t","storagePrefix":"Meteor.oauth.credentialSecret-
13    ","isCordova":false}</div>
14   <script type="text/javascript"
15     src="/packages/oauth/end_of_popup_response.js"></script>
16 </body>
17 </html>
```

And again if we look at line 10 in the above/previous image, there is a script tag with file “end\_of\_popup\_response.js” whose codes are:-

```
1 // NOTE: This file is added to the client as asset and hence ecmascript package has no effect here.
2 (function() {
3
4     var config = JSON.parse(document.getElementById("config").innerHTML);
5
6     if (config.setCredentialToken) {
7         var credentialToken = config.credentialToken;
8         var credentialSecret = config.credentialSecret;
9
10        if (config.isCordova) {
11            var credentialString = JSON.stringify({
12                credentialToken: credentialToken,
13                credentialSecret: credentialSecret
14            });
15
16            window.location.hash = credentialString;
17        }
18
19        if (window.opener && window.opener.Package &&
20            window.opener.Package.oauth) {
21            window.opener.Package.oauth.OAuth.handleCredentialSecret(
22                credentialToken, credentialSecret);
23        } else {
24            try {
25                localStorage[config.storagePrefix + credentialToken] = credentialSecret;
26            } catch (err) {
27                // We can't do much else, but at least close the popup instead
28                // of having it hang around on a blank page.
29            }
30        }
31    }
32
33    if (! config.isCordova) {
34        document.getElementById("completedText").style.display = "block";
35        document.getElementById("loginCompleted").onclick = function() { window.close() };
36        window.close();
37    }
38 })();
39
```

In line 25, we could see that the **credentialToken** and **credentialSecret** are stored in the local storage. We will read these data after the popup is closed by the user.

Now that we have **credentialToken** and **credentialSecret**, we could use them to make a POST request to **/api/v1/login** to authenticate the user. The request payload would look something like this:-

```
{
  "oauth": {
    "credentialToken": "aar8q6rsowq",
    "credentialSecret": "QNIVLda93EkkwMQ9dxK_VXRIz-TiqSggnuuHAWYxX1t"
  }
}
```

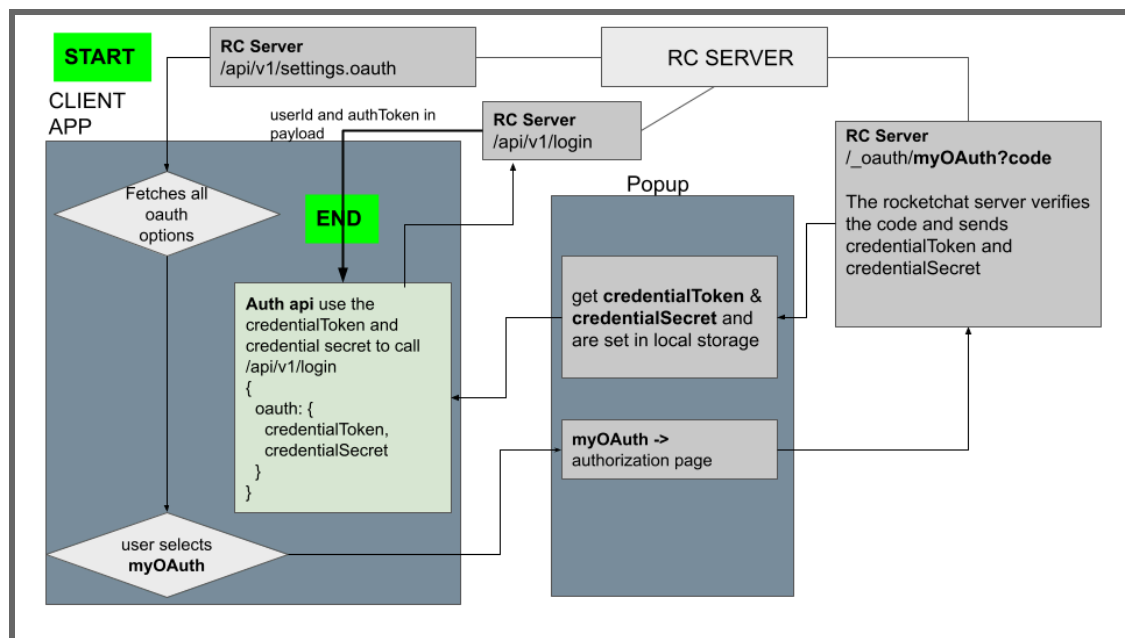
## Response payload on success

```
{
  "status": "success",
  "data": {
    "userId": "AGuDq7LmM2ehHNoeD",
    "authToken": "mArcvx0ZYCRm9Vwj1YaX0l10y5Gc1qw-kl03nGKS2Lm",
    "me": {
      "_id": "AGuDq7LmM2ehHNoeD",
      "services": { ... },
      "emails": [
        {
          "address": "xyz@gmail.com",
          "verified": true
        }
      ]
    },
    "status": "away",
    "active": true,
    "_updatedAt": "2023-03-31T02:58:04.549Z",
    "roles": [
      "user",
      "admin"
    ],
    "name": "Abhinav Kumar",
    "statusConnection": "away",
    "username": "abhinav",
    "utcOffset": 5.5,
    "banners": { ... },
    "oauth": {
      "authorizedClients": [
```

Now, that we have **authToken** and **userId**, we will use them to call authenticated endpoints. Hence, the user is now authenticated.

**Note:** This flow supports both the redirection method and popup method

To summarise, the **MANAGED FLOW**:-



## Benefits of Managed Flow

Managed flow is beneficial when:-

- The application in which EmbeddedChat is being used does not have an authentication system.
- The web admin is not very tech-savvy.
- It is easy to use.

## Moving to a mono-repo system

As the project EmbeddedChat grows we need to separate different parts of the project into their independent modules. It is easier to refactor code across all packages and ensures code reusability. I am planning to make these sub-projects for EmbeddedChat.

- **embeddedchat/api** - Currently we have our api defined in `/src/lib/api.js`. I will separate it into its project. This will help share api across different frameworks and maintain the code base as it grows.
- **embeddedchat/auth** - Currently the authentication options are very basic - password and Google auth. Moreover, the codes written for auth are mainly for React. I will create a separate project inside embeddedchat which will have framework-agnostic functions for authentications. This will help share authentication methods across different frameworks.
- **embeddedchat/react** - Ready to use react components for EmbeddedChat. Currently, EmbeddedChat is already on react. But, this project would be almost a rewrite of the EmbeddedChat react project with various improvements discussed later in this proposal.
- **embeddedchat/react-native** - Ready to use react native components for EmbeddedChat.
- **embeddedchat/shared** - This project will contain utility and helper functions that would be shared across projects.
- **embeddedchat/htmlembed** - This project will contain codes for building the js bundle using the EmbeddedChat React project. EmbeddedChat could be integrated into a webpage by direct copy-pasting the HTML code (which contains the generated bundle loaded with script tag) into the HTML of the webpage.

Any other frameworks that would be added in the future would be added in  
`@embeddedchat/<framework-name>`

## How will I implement mono-repo for EmbeddedChat?

I will be using tools like Lerna and Yarn Workspace to set up the EmbeddedChat mono-repo.

**Lerna** - Lerna is a tool used to manage mono repo. The repositories are structured into sub-repositories.

**Yarn workspace** - Yarn workspaces are used to optimize dependency management. When we use yarn workspaces, all project dependencies are installed in one go. Tools like Lerna make use of Yarn workspaces' low-level primitives

I will add the following additional lines to the package.json of the EmbeddedChat root folder.

```
{
  "name": "EmbeddedChat",
  "version": "1.0.0",
  "private": true,
  "workspaces": ["packages/*"]
}
```

The **workspaces** option is used to specify which subfolder contains the various packages in our mono repo. Each folder within **packages** will be considered a separate project.

Next, I will add **Lerna** to our EmbeddedChat root folder.

```
yarn add lerna -D -W
yarn lerna init
```

This initializes a **lerna.json** configuration file. In the lerna.json file, we will set our npmClient as **yarn**.

```
{
  "packages": ["packages/*"],
  "npmClient": "yarn",
  "version": "0.0.0",
  "useWorkspaces": true
}
```

Now our mono repo is set up. I will create folders - api, react, shared, auth, and react-native. Each folder will act as a subproject.

## HTML Embed

One of the goals of this proposal is to have a capability, where people can just copy and paste an HTML code into a webpage and the EmbeddedChat will be loaded with the minimum configuration required.

### Creating the HTML Embed bundle js

We will use **vitejs** for creating the HTML embed project.

1. Create a project for HTML embedding in the *htmlembed* directory. The config.json of the project would have the following dependencies

```

"dependencies": {
  "embeddedchat": "^0.0.2",
  "react": "^18.2.0",
  "react-dom": "^18.2.0"
},
"devDependencies": {
  "@types/react": "^18.0.28",
  "@types/react-dom": "^18.0.11",
  "@vitejs/plugin-react": "^3.1.0",
  "live-server": "^1.2.2",
}

```

2. The `htmlembed/vite.config.js` would have the following configuration setting.

```

import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import cssInjectedByJsPlugin from 'vite-plugin-css-injected-by-js'
import path from 'path';

export default defineConfig({
  plugins: [react(), cssInjectedByJsPlugin()],
  build: {
    minify: true,
    cssCodeSplit: false,
    lib: {
      entry: path.resolve(__dirname, 'src/EmbeddedChatEmbed.jsx'),
      name: 'EmbeddedChatEmbed',
      formats: ['umd'],
      fileName: () => 'embeddedchat.js',
    }
  },
})

```

3. Now the main library which will render the EmbeddedChat would be located in `/src/EmbeddedChatEmbed.jsx`

```

import React from 'react'
import ReactDOM from 'react-dom/client'
import { RCTComponent } from 'embeddedchat';

const EmbeddedChatEmbed = {
  start(config){
    ReactDOM.createRoot(
      document
        .currentScript.parentNode.querySelector('.embeddedchat')
    )
    .render(
      <React.StrictMode>
        <RCTComponent {...config} />
      </React.StrictMode>,
    )
  }
}
export default EmbeddedChatEmbed

```

The **start(config)** function will render the embeddedchat into its parent node's child with the '.embeddedchat' class. In doing so, it will also apply the config parameter as the props for the embeddedchat component.

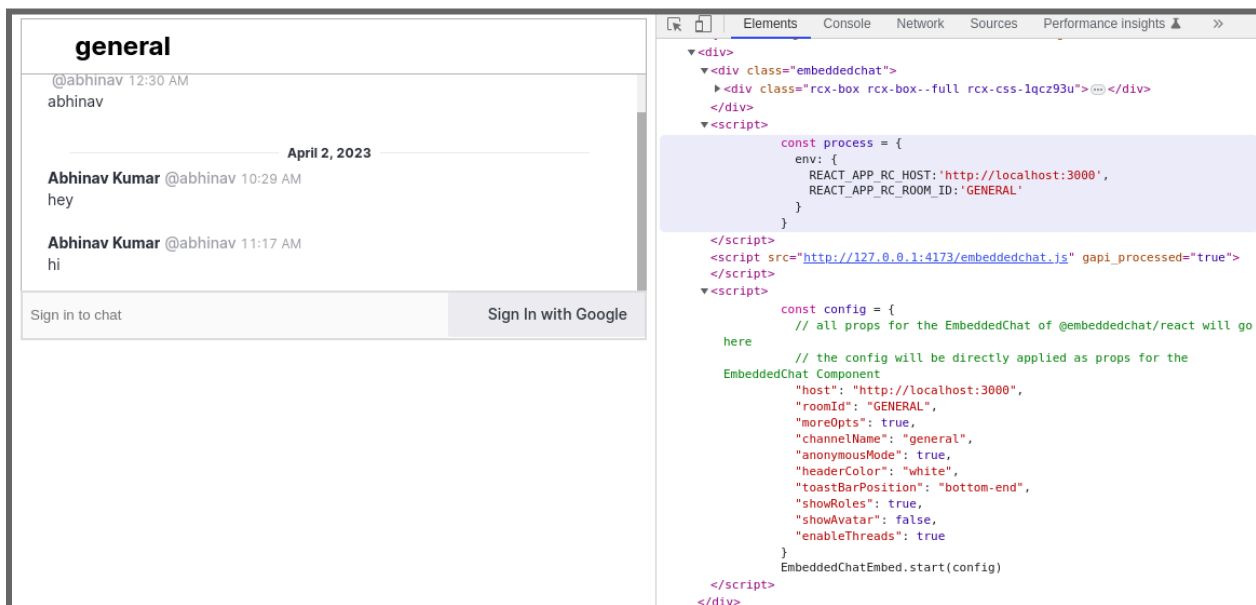
4. To embed it, we will have to paste the following code.

```
<div>
  <script src="path-to-embeddedchat-bundle.js"></script>
  <script>
    const config = {
      // all props for the EmbeddedChat of @embeddedchat/react will go here
      // the config will be directly applied as props for the EmbeddedChat Component
      "host": "http://localhost:3000",
      "roomId": "GENERAL",
      "moreOpts": true,
      "channelName": "general",
      "anonymousMode": true,
      "headerColor": "white",
      "toastBarPosition": "bottom-end",
      "showRoles": true,
      "showAvatar": false,
      "enableThreads": true
    }
    EmbeddedChatEmbed.start(config)
  </script>
</div>
```

## Proof of concept

I have created a proof of concept branch in my fork. It could be accessed at

<https://github.com/abhinavkrin/EmbeddedChat/tree/html-embed-poc/htmlembed>



**Note:** Currently, I have used vitejs as our bundler. If during implementation, we find a better bundler and configuration I will switch to it. But, overall implementation and concept would remain the same.



## Theming

A theme represents the overall look and feel of the organization's brand. Any feature integrated into the app should adapt the look and feel of the application. So adding theming to EmbeddedChat would make it feel like it is a part of the application. This means developers could highly customize the embedded chat to look the way that developer wants.

Currently, we are using Fuselage for the UI of EmbeddedChat. Fuselage is a large library and we need fewer and simpler components to make EmbeddedChat work. Further, it would be tougher to achieve theming with Fuselage. One of our goals is to shift to our custom UI components system that is lightweight, simpler, and has just enough components that are required for EmbeddedChat. This would not only help us decrease the bundle size but also improve the overall performance.

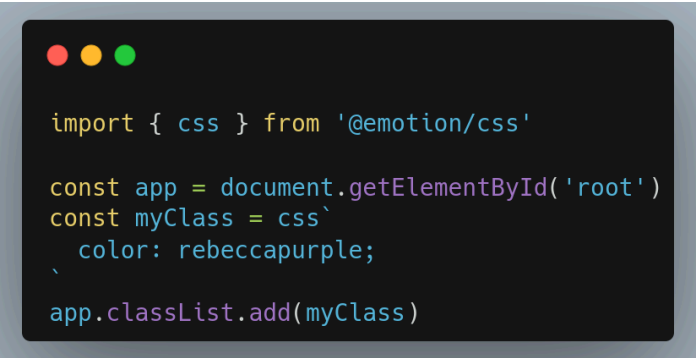
### How will I add styles and theming?

I will be using **Emotion** which is a library designed for writing CSS styles with JavaScript. It provides powerful and predictable style composition in addition to a great developer experience with features such as source maps, labels, and testing utilities. Both string and object styles are supported.

#### Framework agnostic styling

The **@emotion/css** package is framework agnostic and the simplest way to use Emotion. It requires no additional setup, babel plugin, or other config changes. It has support for auto vendor prefixing, nested selectors, and media queries. We simply prefer to use the `css` function to generate class names and `cx` to compose them.

1. Creating class name using the **css()** method

A code editor window with a dark background and light-colored text. It shows the following code:

```
import { css } from '@emotion/css'

const app = document.getElementById('root')
const myClass = css`
  color: rebeccapurple;
`
app.classList.add(myClass)
```

2. Using the **cx()** method to compose class names. Though it is not required in all cases, it is useful for adding class names in the correct order and conditionally as well.

```
import { cx, css } from '@emotion/css'

const cls1 = css`
  font-size: 20px;
  background: green;
`

const cls2 = css`
  font-size: 20px;
  background: blue;
`

<div className={cx(cls1, cls2)} />
```

## React/ReactNative

Styles to a component can be applied in these ways:

1. Using the **css** prop in which styles will be passed as string/object. We will need to import **css** from **@emotion/react**.

```
import { css } from '@emotion/react'

const color = 'white'

render(
  <div
    css={css`
      padding: 32px;
      background-color: hotpink;
      font-size: 24px;
      border-radius: 4px;
      &:hover {
        color: ${color};
      }
    `}
  >
    Hover to change color.
  </div>
)
```

Hover to change color.

(Edit code to see changes)

2. Using **styled.div** style of adding styles by importing **@emotion/styled**.

```
import styled from '@emotion/styled'

const Button = styled.button`
  padding: 32px;
  background-color: hotpink;
  font-size: 24px;
  border-radius: 4px;
  color: black;
  font-weight: bold;
  &:hover {
    color: white;
  }
`

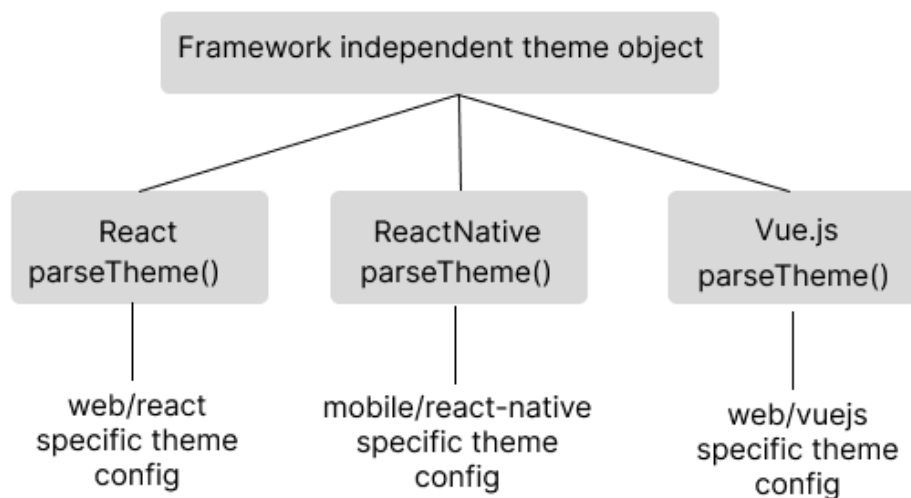
render(<Button>This my button component.
</Button>)
```



(Edit code to see changes)

How do we achieve theming in a framework-agnostic way?

1. For each component across different frameworks, we will pass our language-independent **theme object** as a global parameter.
2. The **global theme object** would be parsed with framework-specific `parseTheme()` to convert them into a format that is easily usable in that framework.
3. The **parsed theme object** would then be propagated and used in the framework-specific method. I will explain more about the theme object later in this proposal.



Theming using Emotion for React/ReactNative

Theming feature of emotion is included in the **@emotion/react** package. I will add the ThemeProvider to the top level of EmbeddedChat and provide the theme object to it.

Example:-

```
import { ThemeProvider } from '@emotion/react'

const defaultTheme = {
  colors: {
    primary: 'blue',
    secondary: 'hotpink',
    success: 'green',
    error: 'red',
    warning: 'yellow',
  }
}

export default function EmbeddedChat({theme, ...otherProps}) {
  return (
    <ThemeProvider theme={theme || defaultTheme} {...otherProps}>
      <div
        css={theme => ({
          color: theme.colors.primary
        })}>
        ...
      </div>
    </ThemeProvider>
  )
}
```

Each component will then be able to access the theme object using the **withTheme** wrapper or the **useTheme** hook.

1. **withTheme**: The component should be wrapped with **withTheme** which is a high-order function. It will pass the theme to the wrapped component through the **theme** prop.
2. **useTheme**: Using the **useTheme** hook, the theme component can directly access the theme object.

## Theme Object Schema

Our theme object would be **framework agnostic**. The example theme schema given below would be used in all frameworks like react, react-native, vue, etc.

```
type Theme = {
  cssBaseline: React.CSSProperties;
  palette: {
    mode: "light" | "dark";
    primary: ColorOptions,
    secondary: ColorOptions;
    . . . other colors
  },
  typography: Typography,
  shadows: Shadows,
  zIndex: {
    [componentName: string]: number;
  },
}
```

```

components: {
  [ComponentName: string]: {
    defaultProps?: Object;
    styleOverrides?: React.CSSProperties;
    classNames?: string;
  }
},
breakpoints: { ...}
}

```

For complete theme schema please check out this gist:

<https://gist.github.com/abhinavkrin/455ddaa9a9ed7f91419d3e1cf4e9a488>

I have also created an example theme object which could be access in this gist.

**Example theme object:-** <https://gist.github.com/abhinavkrin/475e9712d8f4092397d6ed8f387cdd94>

How many built-in theme objects would be there?

There would be one default theme object that would have the look and feel of the fuselage components. It will be applied if no theme is provided.

I will also create some ready-to-use built-in theme objects.

Application and usage of the theme object for customization of the components. How do we modify the look and feel at the component level?

We have an option for components in the theme object. This could be used for defining custom styles for each component using **styleOverrides** which will be applied directly to the **style** of the component. We can also pass custom classes for a component using **classNames** options.

```

...
"components": {
  "ChatInput": {
    "styleOverrides": {
      "fontWeight": 400,
      "color": "gray",
      "border": "1px solid black"
    },
    "classNames": "myCustomClassForChatInput"
  },
  "Message": {
    "classNames": "myCustomClass"
  }
}
...

```

How will I prevent style clashes that may be inherited from the website from a top level?

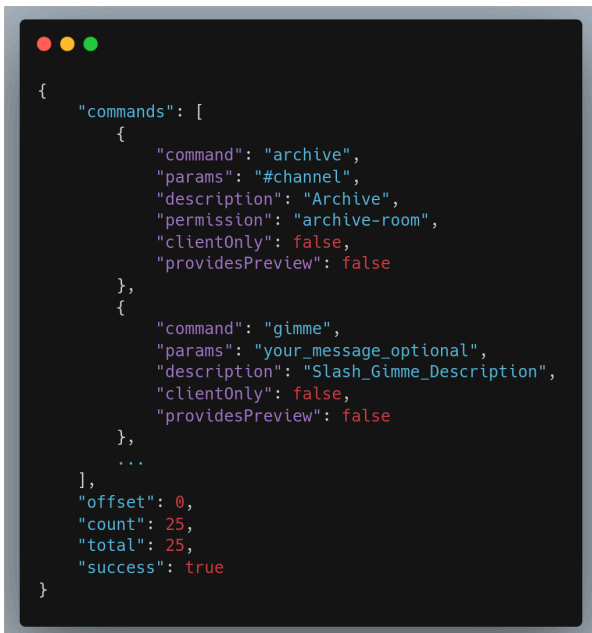
I will be using **cssBaseline** which will preserve useful defaults and normalizes styles for a wide range of elements. This will be done at the top level of the EmbeddedChat component. I will create a default Value for cssBaseline which will be used if the user does not provide a value explicitly.

## Slash Commands

Slash commands are powerful features of RocketChat. It starts with a / and executes a specific task. Currently, EmbeddedChat does not support RocketChat's slash commands. Our goal is to include it in EmbeddedChat as well.

Getting List of available commands

To get a list of available slash commands, I will use the **/api/v1/commands.list** endpoint. The payload of this endpoint would look something like this:-

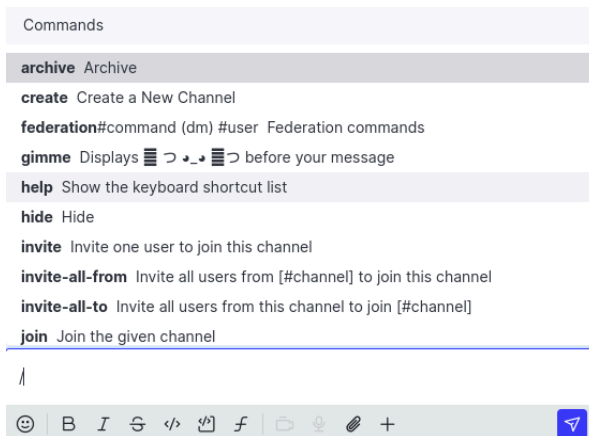


```
{
  "commands": [
    {
      "command": "archive",
      "params": "#channel",
      "description": "Archive",
      "permission": "archive-room",
      "clientOnly": false,
      "providesPreview": false
    },
    {
      "command": "gimme",
      "params": "your_message_optional",
      "description": "Slash_Gimme_Description",
      "clientOnly": false,
      "providesPreview": false
    },
    ...
  ],
  "offset": 0,
  "count": 25,
  "total": 25,
  "success": true
}
```

After the commands are fetched, they will be stored in the state.

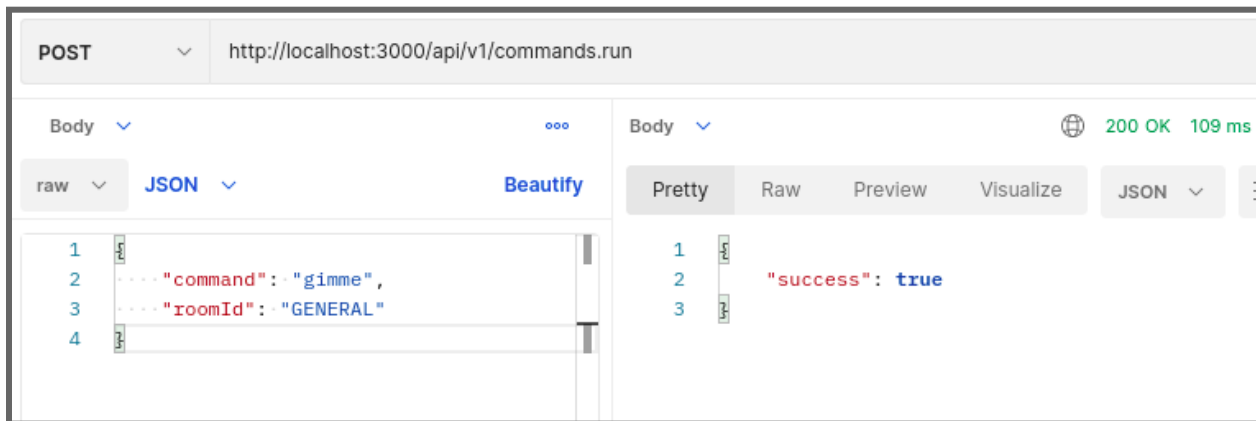
Displaying the commands list

The chat input would check if the cursor position is placed after a ' / ' character. If yes, then a list of commands will appear right above the chat input as it is done in RocketChat. Only those commands will be displayed for which the current user has appropriate permission.

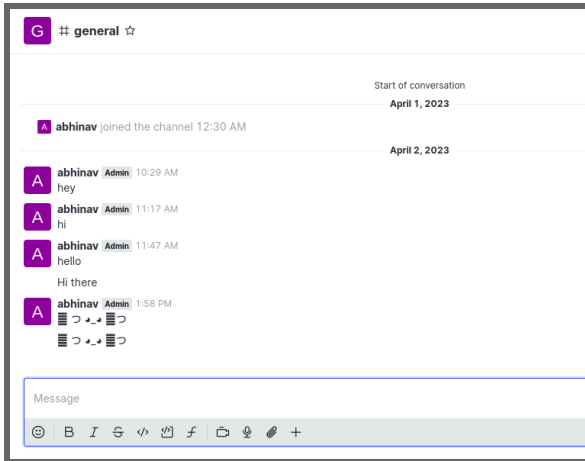


## Executing the slash commands

Now to execute a command, a POST request will be made to **/api/v1/commands.run** with the payload having roomId, command name, params, and tmid (message id of the thread). Refer to <https://developer.rocket.chat/reference/api/rest-api/endpoints/core-endpoints/commands-endpoints/execute-a-slash-command>



After the request is successful, the command is executed in the room.



How will I handle UI that might open up by slash command execution?

UI changes due to slash commands can occur due to two events -

1. Real-time messages received from the app.
2. Result of any UI interaction with the app's block.

To receive UI updates from a RocketChat app, I will first subscribe to [stream-notify-user](#). Any event with **<currentUserId>/message** having [UIKit building blocks](#) will be meant to display UI changes. Each block will contain **appId** and **blockId**. Now, if there is a block with type= "actions". It is meant to be interactive. To perform the interaction, I will call **api/apps/ui.interaction/<appId>**. The response may contain type = modal.open or modal.update. The former means a modal has to be opened with the blocks provided in the payload. The latter means the opened modal has to be replaced with the new block.

For example, on running /remind command for Remind Bot, we receive the payload



```

{
  "msg": "changed",
  "collection": "stream-notify-user",
  "id": "id",
  "fields": {
    "eventName": "eXqQEdNFypqoy6g6G/message",
    "args": [
      {
        "groupable": false,
        "_id": "9B3SvwJTq9P7dtv3i",
        "rid": "GENERAL",
        // ...
        "blocks": [
          {
            "type": "section",
            "text": {
              "type": "mrkdwn",
              "text": "Hello Abhinav, I'm Reminder Bot 🌟..."
            },
            "blockId": "b75ab3d0-d243-11ed-acd1-c3f80de475fa",
            "appId": "7a8cd36b-5f7b-4177-bd7f-bfc9be908bf8"
          },
          {
            "type": "actions",
            "elements": [
              {
                "type": "button",
                "text": {
                  "type": "plain_text",
                  "text": "Create Reminder",
                  "emoji": false
                },
                "actionId": "create-reminder",
                "style": "primary"
              },
              // ...
            ],
            "blockId": "b75ab3d1-d243-11ed-acd1-c3f80de475fa",
            "appId": "7a8cd36b-5f7b-4177-bd7f-bfc9be908bf8"
          },
          // ...
        ]
      }
    ]
  }
}

```

For pull payload: <https://gist.github.com/abhinavkrin/a2c1406c06771c34246526db9054d51c>

The “blocks” contains UIKit building blocks. This block data will be used to display the UI.

## How will I handle user interaction with action buttons?

I will make a POST request to **api/apps/ui.interaction/<appId>** endpoint with the following payload:

```

{
  "actionId": "create-reminder"
}
▼ container: {type: "message", id: "nf7R6ffin2vE5MwaX"}
  id: "nf7R6ffin2vE5MwaX"
  type: "message"
  mid: "nf7R6ffin2vE5MwaX"
▼ payload: {blockId: "5e332a20-d2ad-11ed-be54-4734c244d43a", value: ""}
  blockId: "5e332a20-d2ad-11ed-be54-4734c244d43a"
  value: ""
  rid: "GENERAL"
  triggerId: "dgqdgj5mfio"
  type: "blockAction"

```

The response payload may contain instructions to **open/update** a modal along with **UI blocks** to be displayed.

```
{success: true, type: "modal.open", triggerId: "dgqdgj5mfio",...}
  appId: "7a8cd36b-5f7b-4177-bd7f-bfc9be908bf8"
  success: true
  triggerId: "dgqdgj5mfio"
  type: "modal.open"
  view: {appId: "7a8cd36b-5f7b-4177-bd7f-bfc9be908bf8", type: "modal",...}
    appId: "7a8cd36b-5f7b-4177-bd7f-bfc9be908bf8"
    blocks: [{type: "input", blockId: "reminder",...}, {type: "input", blockId: "
    close: {type: "button", text: {type: "plain_text", text: "Cancel", emoji: f
      id: "reminderCreateModal-026125b6-0aed-4904-a559-a39f59f6aa40"
      showIcon: true
    submit: {type: "button", text: {type: "plain_text", text: "Create", emoji:
    title: {type: "plain_text", text: "Create Reminder", emoji: false}
    type: "modal"
```

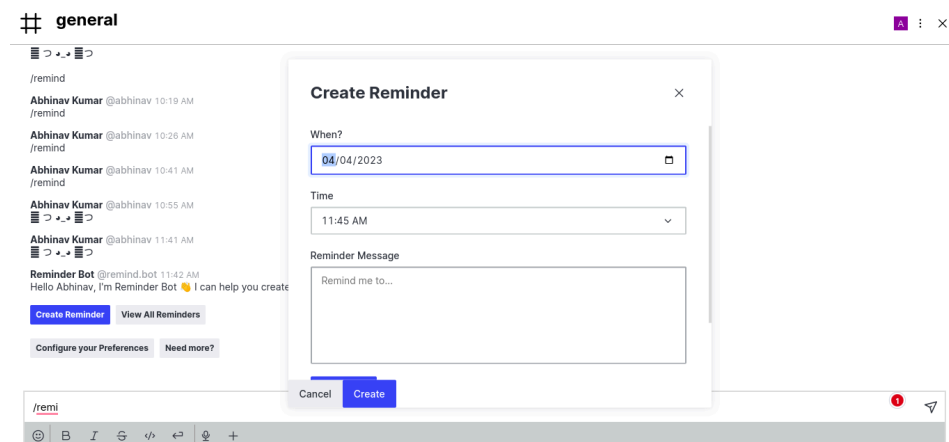
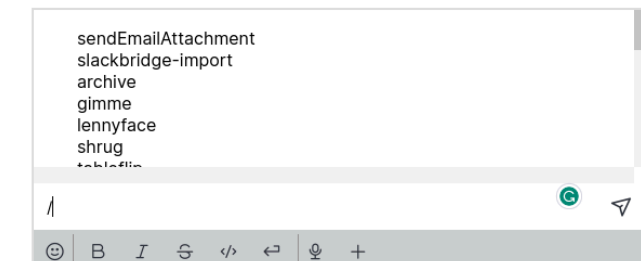
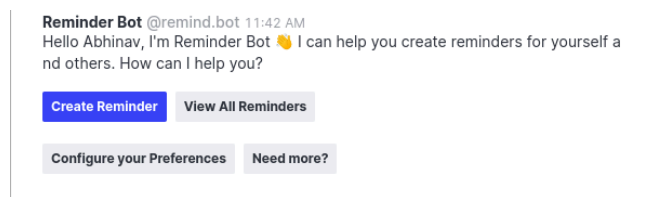
## Proof of concept

I have also created a simple implementation as proof of concept to show how I will handle the UI trigger by slash commands and UI interaction by users.

video link: [https://drive.google.com/file/d/1s2QIkHiCzaEqhvdCUS223yJZuxBSLJ1J/view?usp=share\\_link](https://drive.google.com/file/d/1s2QIkHiCzaEqhvdCUS223yJZuxBSLJ1J/view?usp=share_link)

branch link in my fork: <https://github.com/abhinavkrin/EmbeddedChat/tree/slash-command-poc>

## Screenshots:



## Adding Moderation

Chat moderation is required to review and regulate user-generated messages and content posted on a platform to ensure it's not inappropriate or otherwise harmful to the brand's reputation or its users. But currently in embedded chat-

1. The owner of a message can only delete it.
2. Users can report a message if it violates community standards.

Proposed changes to the current moderation system in EmbeddedChat

1. Users with delete-message permission can also delete messages. That user will act as a moderator.
2. Users with remove-user permission can remove a user.

Mechanism of determining user permission

1. Get all the permissions from **/api/v1/permissions.listAll**. The payload has the permissions along with the associated roles.
2. Get user's roles from the user details retrieved from **/api/v1/me**

Payload from **/api/v1/permissions.listAll**

```
{
  "update": [ {
    "_id": "delete-message",
    "_updatedAt": "2023-03-31T19:00:15.570Z",
    "roles": [
      "admin",
      "owner",
      "moderator"
    ]
  },
  ... other roles
],
... other groups of permissions
}
```

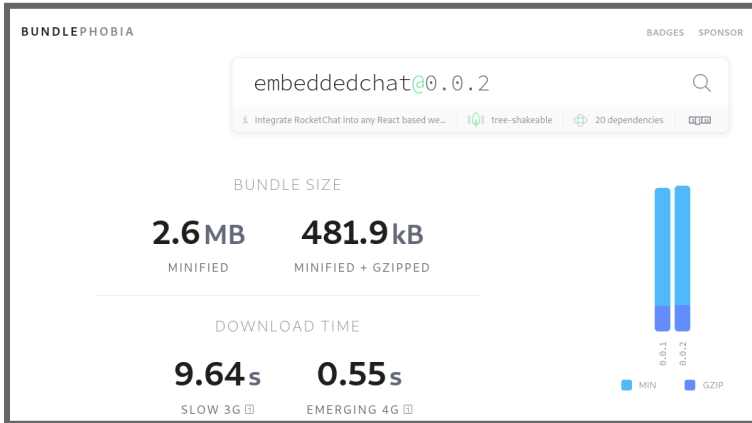
Payload from **/api/v1/me**

```
{
  "_id": "eXqQEdNFypqoy6g6G",
  "emails": [...],
  "status": "online",
  "active": true,
  "_updatedAt": "2023-04-02T08:29:21.496Z",
  "roles": [
    "user",
    "admin"
  ],
  ...
}
```

3. After that, we can determine the user's permissions.  
user -> role  
permission -> role  
**=> user -> role -> permission**
4. Now that we have the current user's all permission, we can determine if the current user has permission to delete messages or remove participants. According, I will show UI to the user for the actions.

## Reducing Bundle Size

Currently, our bundle size is quite large and this can seriously affect the performance of the EmbeddedChat. One of my goals is to reduce and keep the bundle size of EmbeddedChat as small as possible.



The steps I am going to take to reduce the bundle sizes are:-

1. The most important step that will decrease our bundle size is **moving to our own component system** for different frameworks. We will **shift from the fuselage** as it is large and overkill for our use case. We only need a few simpler components for EmbeddedChat. Further, we will be able to add **theming and more customization freedom** while moving to our component system.

```
Rollup File Analysis
-----
bundle size:      3.862 MB
original size:    3.864 MB
code reduction:   0.04 %
module count:     276

file:              /node_modules/@rocket.chat/fuselage/dist/fuselage.development.js
bundle space:      46.34 %
rendered size:     1.79 MB
original size:     1.81 MB
code reduction:    1.13 %
dependents:        1
- /node_modules/@rocket.chat/fuselage/index.js

file:              /node_modules/@rocket.chat/fuselage/dist/fuselage.production.js
bundle space:      12.83 %
rendered size:     495.659 KB
original size:     495.57 KB
code reduction:    0 %
dependents:        1
- /node_modules/@rocket.chat/fuselage/index.js

file:              /node_modules/emoji-toolkit/lib/js/joypixels.js
bundle space:      12.52 %
rendered size:     483.518 KB
original size:     483.461 KB
code reduction:    0 %
dependents:        3
- /src/components/Markup/elements/Emoji.js
- /src/components/Markdown/Markdown.js
- /src/lib/emoji.js
```

We can see that bundle analysis by [rollup-plugin-analyzer](#) shows that (46 + 12) % of total bundle space is taken by the fuselage.

2. We can also reduce the bundle sizes by using **dynamic imports** and **code splitting**. There could be many components, like those for threads and emoji pickers which need not be

included in the main bundle. They could be dynamically imported on demand. This will also help in reducing bundle size.

3. Implementing our utility functions for simple tasks and avoiding using third-party packages for simple utilities.
4. Install bundle size analyzer like [rollup-plugin-analyzer](#) and configure it in the build config. This will help us always keep track of what package is taking how much bundle space. The output of the analyzer has been shown in point 1.

```
EmbeddedChat > 2 rollup.config.js > ...
6 import json from '@rollup/plugin-json';
7 import bundleSize from 'rollup-plugin-bundle-size';
8 import { terser } from 'rollup-plugin-terser';
9 import analyze from 'rollup-plugin-analyzer'
10
11 const packageJson = require('./package.json');
12 const env = process.env.NODE_ENV;
13
14 export default [
15   {
16     input: 'src/index.js',
17     output: [
18       {
19         format: 'es',
20         file: 'dist/index.js',
21       },
22       {
23         format: 'cjs',
24         file: 'dist/index.cjs',
25       },
26     ],
27     plugins: [
28       resolve({ browser: true }),
29       commonjs({ include: ['node_modules/**'] }),
30       babel({
31         exclude: 'node_modules/**',
32         presets: ['@babel/env', '@babel/preset-react'],
33         babelHelpers: 'bundled',
34       }),
35       postcss(),
36       json(),
37       external(),
38       bundleSize(),
39       analyze(),
40     ],
41   },
42 ];
```

5. **Reduce very old browser support.** To support older browsers build tools need to add additional code to replace functionality that's not natively supported. This increases JavaScript file size. Reducing the set of browsers supported in your build configuration can reduce bundle size by 5-15%. Libraries like rollup-plugin-babel can make this easy to do. Example usage of babel plugin for rollup.

```
import babel from 'rollup-plugin-babel';
...

export default {
  ...
  plugins: [
    ...
    babel({
      exclude: 'node_modules/**',
      presets: [
        '@babel/env',
        {
          modules: 'false',
          targets: {
            browsers: '> 1%, IE 11, not op_mini all, not dead',
            node: 8
          },
          useBuiltIns: 'usage'
        }
      ]
    })
  ],
  ...
};
```

## Improving API

My targets for improving api:-

1. moving api to its own project so that it could be shared across projects for different frameworks. This will also help in reducing repetitive code in our repository and also it will act as a single source of truth. So if any problem occurs with the api in any framework, we will know where to look. And also it will be fixed in all the frameworks.
2. Replacing REST API calls with real-time API method calls wherever possible.
3. Adding auth API project.
4. Adding automatic retries up to a few times if API calls fail.
5. Adding debouncing where necessary.
6. Making api project framework agnostic.

### Using real-time API method calls.

When I started contributing to EmbeddedChat, all messages were loaded with API polling (using REST API). One of the goals told by my mentor was to shift to a real-time message subscription and other features to real-time instead of polling.

I created PR to use a real-time message subscription to get new message updates. The PR was merged and it was our first step to move to real-time api instead of API polling. The details about the PR could be found here <https://github.com/RocketChat/EmbeddedChat/pull/169>

Further, I added the feature to get the real-time typing status of users in EmbeddedChat. The PR could be found here <https://github.com/RocketChat/EmbeddedChat/pull/177>. After my PR was merged, the support for the old method of typing message subscriptions was removed in the latest version. I created a PR to support the current subscription model for getting typing status of users. Its PR could be found here <https://github.com/RocketChat/EmbeddedChat/pull/199>

There are still a lot of functions that could also use real-time API as well as REST API. Some of them are:-

1. [Send Message](#)
2. [Update Message](#)
3. [Star Message](#)
4. [Set Reaction](#)
5. [Pin Message](#)
6. [Delete Message](#)

I will add real-time alternatives for the functions wherever possible.

## How to call real-time API methods?

We are using `"@rocket.chat/sdk": "^1.0.0-alpha.42"` for interacting with RocketChat api. To call a real-time method:

```
async sendTypingStatus(username, typing) {  
  try {  
    const res = await this.rcClient.methodCall(  
      'stream-notify-room',  
      `${this.rid}/user-activity`,  
      username,  
      typing ? ['user-typing'] : []  
    );  
  } catch (err) {  
    console.error(err.message);  
  }  
}
```

Here, rcClient is an instance of RocketChat js sdk. res object will contain the ID of the method call which can be used to identify the result of the method call.

## How will I leverage both real-time api and rest api to achieve a better user experience?

1. I would be to using real-time api only for events that need to update the UI over time, like user interactions (typing, uploading...), user presence, etc.
2. Also realtime API would be used to subscribed to receive events for new messages, message updates, delete message updates, room events, user activity status.
3. Some of the events that need to be subscribed are - **stream-room-messages** to receive new message updates, **stream-notify-room** to received updates like user-activity (user typing status), and message deletion.
4. While initial load I will use REST api to poll the latest messages and after that new messages would be added in real-time by subscribing to **stream-room-messages**.
5. Further I will take inspiration from the react-native app to choose between real-time and REST api. However, user experience and app performance will be a top priority.

## How would I implement listening for real-time events?

I have decided to use the **addEventListener(event, callback)** way of calling functions when a certain event occurs. I have already added **addMessageListener**, **addMessageDeleteListener**, **addTypingStatusListener**, etc to the current api implementation. One of the PRs could be found here: <https://github.com/RocketChat/EmbeddedChat/pull/169>

The code of **addMessageListener** looks something like this:

```
async addMessageListener(callback) {
  const idx = this.onMessageCallbacks.findIndex((c) => c === callback);
  if (idx !== -1) {
    this.onMessageCallbacks[idx] = callback;
  } else {
    this.onMessageCallbacks.push(callback);
  }
}

async removeMessageListener(callback) {
  this.onMessageCallbacks = this.onMessageCallbacks.filter(
    (c) => c !== callback
  );
}
```

Now the added **callbacks** will be called every time a new message is received from the subscription.

```
// this code would implemented in our API class
async connect() {
  try {
    ... more code
    await this.rcClient.subscribe('stream-room-messages', this.rid);
    await this.rcClient.onMessage((data) => {
      this.onMessageCallbacks.map((callback) => callback(data));
    });
    ... more code
  } catch (err) {
    await this.close();
  }
}
```

To add an event listener, the developer could **add a callback** in the following manner:

```
RCInstance.connect().then(() => {
  RCInstance.addMessageListener(upsertMessage);
  RCInstance.addMessageDeleteListener(removeMessage);
});
```

This method of adding event listeners is easy and simple and needs zero configuration. It will be framework agnostic and will be used in our components for React, React Native, vuejs, etc.



## Docs and tutorials

Accurate documentation are essential for maintaining a codebase because it allows developers to quickly understand what the code does and how to work with it. It will also help new contributors to understand our project and encourage them to start and keep contributing.

### Documentation

I will use a storybook for docs as well as for live testing and development of components. The document for a component would be added as soon as I finish coding it. I will discuss the storybook in the next section.

### Tutorials

I will create a wiki that will include various tutorials for EmbeddedChat usage. Some topics that I will be taking first are:-

1. Adding EmbeddedChat in a React project.
2. Using different auth flows in EmbeddedChat.
3. Using EmbeddedChat HTML embed version in WordPress.
4. Using custom themes in EmbeddedChat
5. Different variants are possible in EmbeddedChat
6. Adding EmbeddedChat in a react-native project

## Storybook implementation

Adding [storybook](#) to our project will significantly improve the developer experience in UI development. We could even add documentation for each component. Currently storybook supports diverse web frameworks, including React, Vue, Angular, Web Components, Svelte, and over a dozen others. We will have the storybook initialized in each framework project. By running **`npm storybook init`** we can initialize storybook in an existing project. Storybook will look into our framework-level project's dependencies during its installation process and provide the best configuration available. The command above will make the following changes to your local environment:

1. Install the required dependencies.
2. Set up the necessary scripts to run and build Storybook.
3. Add the default Storybook configuration.
4. Add some boilerplate stories to get you started.
5. Set up telemetry to help us improve the Storybook.

In storybook, a **story** captures the rendered state of a UI component. It's a function that returns a component's state given a set of arguments. We can create a story with `ComponentName.stories.jsx` format filename.

An example story would look like this -

```
import React from 'react';
import { RCTComponent } from '..';

export default {
  title: 'UI/RCTComponent',
  component: RCTComponent,
  parameters: {
    componentSource: {
      component: 'RCTComponent',
    },
  },
};

const Template = (args) => <RCTComponent {...args} />;
export const Simple = Template.bind({});
Simple.args = {
  host: 'http://localhost:3000',
  roomId: 'GENERAL',
  moreOpts: true,
  channelName: 'general',
  anonymousMode: true,
  headerColor: 'white',
  toastBarPosition: 'bottom-end',
  showRoles: true,
  showAvatar: false,
};
```

**Documentation** for a component could be written in mdx (markdown + jsx) format in Component.stories.mdx file name format. Example doc looks like this -

```
import { Meta, Story, Canvas } from '@storybook/addon-docs';
import { RCTComponent } from '../index.js';

# EmbeddedChat

<Meta title="RCTComponent" component={RCTComponent} />

An easy-to-use full-stack component (ReactJS + backend behaviors) embedding Rocket.Chat into
your web app.

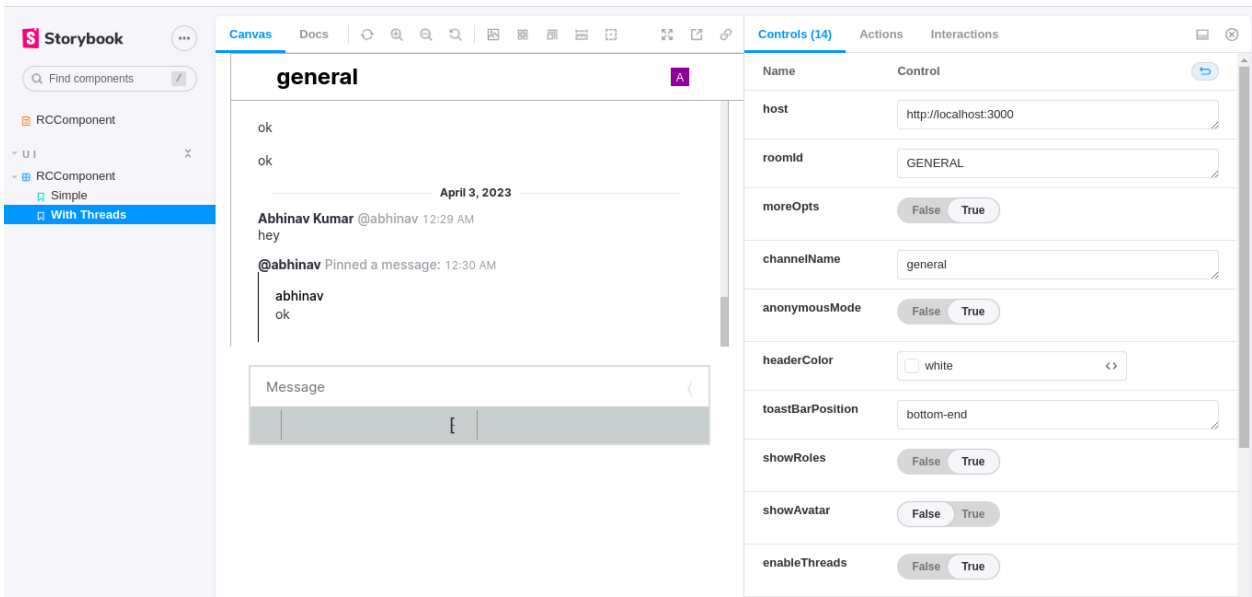
_EmbeddedChat is a full-stack React component node module of the RocketChat application that is
fully configurable, extensible, and flexible for use. It is tightly bound with the RocketChat
server using Rocket.Chat nodejs SDK and its UI using RocketChat's Fuselage Design System._

![embeddedchatwall] (https://user-images.githubusercontent.com/73601258/178119162-ecabb9b7-e3ae-
4c70-8ab2-f6c02856f4c6.png)

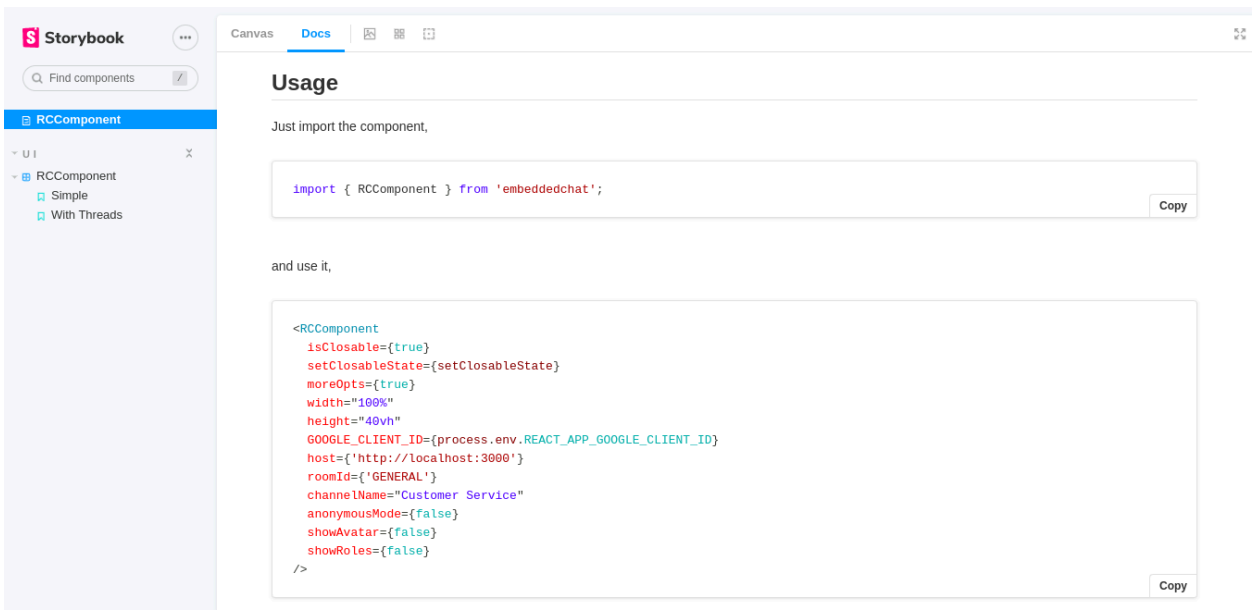
<div align="center" width="100%"> ... so on
```

# Proof of concept for storybook

## Story preview for EmbeddedChat



## Story preview for docs



I have also created a branch in my fork that has storybook implementation in the current EmbeddedChat.  
<https://github.com/abhinavkrin/EmbeddedChat/tree/storybook-poc>

## Comment Mode

This would be an experimental feature. I plan to introduce a ready-to-embed comment system just like [Disqus](#). The application logic of EmbeddedChat will remain the same. It would be just a UI layout change that will make EmbeddedChat appear as a commenting system.

**Message -> Main Comment** - Main messages would be displayed as comments

**Thread Message -> Replies** - Thread messages would be displayed as comment replies.



I will export a different Component for the component mode. Example code for React.

```
import { EmbeddedComments } from '@embeddedchat/react';

const props = {
  // ...
};

ReactDOM
  .createRoot(document.getElementById('root'))
  .render(<EmbeddedComponents {...props}/>);
```

## Time commitment FAQs

How much time will I be able to commit to this project (per week or day)?

I will be able to spend at least 5 hours on weekdays which may be extended. On weekends, I will spend about 6-8 hours daily. Therefore I would be able to spend about 40 hours per week. Hence, In 13 weeks I would be able to spend ~500 hours. This gives me enough time to complete the project and take a few days off due to unforeseen conditions. The approximate time for large projects is 350 hours.

## Communications Channels

How does someone reach me?

website	<a href="http://avitechlab.com">http://avitechlab.com</a>
open.rocket.chat Username	abhinav.kumar30
Linkedin	<a href="https://linkedin.com/in/abhinavkrin">https://linkedin.com/in/abhinavkrin</a>
Github	<a href="https://github.com/abhinavkrin">https://github.com/abhinavkrin</a>
Timezone	IST (GMT+5.5)

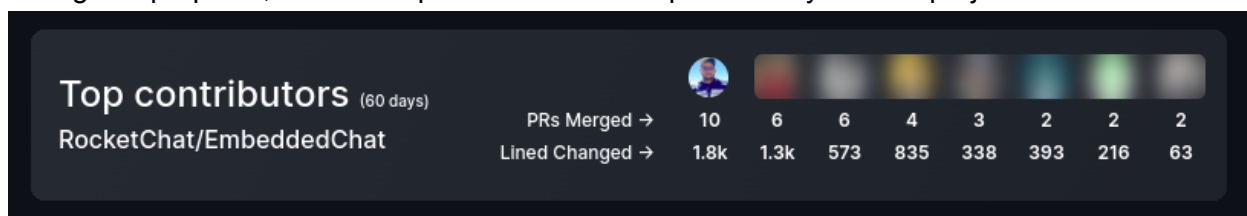
How often, and through which channel(s), do I plan on communicating with my mentors?

I plan to schedule a virtual meeting regularly every week to give status updates to my mentors after discussing our availability and agenda for the meeting. The primary mode of communication between me and my mentors would be through open.rocket.chat itself. Other modes of communication could also be used taking into account the availability of everyone. I would be available on call or by messaging from 11 am IST to 1 am IST on both weekdays and weekends.

## Why do I find myself suited for this project?

I have been contributing to RocketChat for the past few months. I love the community and have made new friends out here. This community has helped me a lot and I want to contribute my part to it.

I have contributed significantly to the EmbeddedChat project in the past few months and at the time of writing this proposal, I am the top contributor in the past 60 days to this project.



I am among the **top contributors** in the **RocketChat GSoC 2023 leaderboard**. The GSoC leaderboard could be accessed at <https://gsoc.rocket.chat>.

As a result, I have acquired a thorough understanding of the EmbeddedChat project. I have also spent hours hacking and debugging RocketChat to find solutions for EmbeddedChat. Like I found the solution to support all OAuths in EmbeddedChat without any backend. I had to go through RocketChat's ReactNative and core project to find out how things are implemented in RocketChat.

I have personally become fond of the EmbeddedChat project and want to keep working on it and improving it.

## Related work done and other contributions

I have contributed to EmbeddedChat and have become familiar with its codebase and goals. I have also completed a few parts of the goals included in the original idea list.

### MERGED PR to EmbeddedChat

Click here for the latest list

<https://github.com/RocketChat/EmbeddedChat/pulls?q=is%3Amerged+is%3Apr+author%3Aabhinavkrin>

1. [NEW] Using DDP to get messages: <https://github.com/RocketChat/EmbeddedChat/pull/169>
2. [ADD] Added real-time typing status for users in the channel  
<https://github.com/RocketChat/EmbeddedChat/pull/177>
3. [Improve] Pending message status feedback to user  
<https://github.com/RocketChat/EmbeddedChat/pull/188>
4. [ADD] Thread Chat support  
<https://github.com/RocketChat/EmbeddedChat/pull/183>
5. [IMPROVE] shift+enter adds a newline in chat input  
<https://github.com/RocketChat/EmbeddedChat/pull/156>
6. [FIX] Room not changing on changing roomId prop  
<https://github.com/RocketChat/EmbeddedChat/pull/161>
7. [BUGFIX] mentions and roles causing the initial app load to crash when unauthed  
<https://github.com/RocketChat/EmbeddedChat/pull/174>
8. [IMPROVE] refactor-message-components  
<https://github.com/RocketChat/EmbeddedChat/pull/196>
9. [HOTFIX] fixes for previous merges  
<https://github.com/RocketChat/EmbeddedChat/pull/197>
10. [IMPROVE] typing status support for rc 6.0+  
<https://github.com/RocketChat/EmbeddedChat/pull/199>
11. [FIX] Fixes minor UI issues  
<https://github.com/RocketChat/EmbeddedChat/pull/204>
12. [FIX] replace Math.random() with crypto api.  
<https://github.com/RocketChat/EmbeddedChat/pull/206>

## OPEN PRs to EmbeddedChat

Click here for the latest list

<https://github.com/RocketChat/EmbeddedChat/pulls?q=is%3Aopen+author%3Aabhinavkrin>

1. [ADD] Previews for URLs

<https://github.com/RocketChat/EmbeddedChat/pull/201>

## Other Contributions to RocketChat project

Besides EmbeddedChat, I am also contributing to [RC4Community](#) and [GSoC-Community-Hub](#)

All my PRs could be found here: [click here](#)

I have also created several issues that were fixed and some will be fixed. All my issues could be found here: [click here](#).

## Relevant Experiences

I have been consistently contributing to RocketChat. My contributions include fixing bugs, adding features, raising issues, and helping other community members.

The links to all my PRs and issues that I have created are:-

- [Merged PRs](#) (16 PRs including 7 chores/minor fixes)
- [Open PRs](#) (2 open PRs)
- [Issues](#) (13 issues)

I am among the **top contributors** in the **RocketChat GSoC 2023 leaderboard**. The GSoC leaderboard could be accessed at <https://gsoc.rocket.chat>.

## Projects that I have worked on

**Tournify.in** — *A simple points table maker for esports.*

Link: <https://tournify.in>

**Hitknotice.netlify.app** — *An app to show my college's notices and send notifications for new ones.*

Link: <https://hitknotice.netlify.app>

**esportsweb.in** — *A complete eSports tournament management platform.*

Link: <https://esportsweb.in>

To know more about my works and projects, feel free to visit my website at <https://avitechlab.com>.

## Future Work and Goals

I plan to add Vuejs components and more pre-built themes. I would love to keep contributing to RocketChat projects and be an active member of the community. I am also working on GSoC Community Hub along with other contributors and will try to make it live before the next GSoC season. I will learn more about the RocketChat core and contribute my part in improving it

## Work Plan:

Application review period: April 4 - May 4

During this time I will spend my time learning more about the RocketChat core. I am also working on the [GSoC Community Hub](#) project. So I will spend more time on contributing to it. I will keep myself active in the community helping my fellow contributors and also learning from them. I will also spend time discussing the project ideas and discuss the implementation of the features and how to improve upon them resulting in a better user experience and making it user-friendly.

Community bonding period

During this period I will interact more with the community. I will try to know my mentors, read the documentation of RocketChat, and make myself familiar with how things work with other RocketChat clients - mobile and desktop. I will set up communication channels with my mentors and decide how often I would connect with them to share my updates and get their feedback.

Week 1: May 29 - June 4

- Restructuring the project and shifting to mono-repo. Set up sub-projects for embeddedchat/api, embeddedchat/react, embeddedchat/auth, and embeddedchat/shared.
- Start working on the authentication project and try to complete both flows - **Token** and **Managed**.

Week 2: June 5 - June 11

- Complete implementation of the @embeddedchat/api project.
- Bootstrap the @embeddedchat/react project, and set up the storybook.
- Now, the development of both the api and the react project will go hand in hand

Week 3 - 4: June 12 - June 18 & June 19 - June 25

These two weeks will also be spent working on:-

- React project
- Creating the default theme and working on the theming object.
- Shifting api functions to use real-time method calls.

I will finish the api, react components, and theming tasks by the end of this period.



Week 5: June 26 - July 2

- Add slash commands support.
- I will test this feature with various commands available in RocketChat.

I might encounter a few issues which I will discuss with my mentor and fix them. Slash commands would be ready by the end of this week.

Week 6: July 3 - July 9

- Add the moderation feature.
- Add functions in the api project that would be used for fetching the user roles and all the permissions available on the RocketChat server.
- Create a utility function in the shared project that will help in determining if the current user has certain permission or not. For example - **hasPermission('delete-message', userRoles, all-permissions)**, **hasPermission('delete-message', role)**, and other variations.
- build the UI that will show moderation options based on the user role.

Week 7: July 10 - July 16 (July 14 midterm evaluation, keep buffer)

- keep this week as a buffer week as it includes the first evaluations.
- I will work on the feedback received from the mentors in the evaluation and work on that.
- Complete any incomplete work from the above weeks due to any delay in the timeline before the first evaluations.

Week 8: July 17 - July 23

- I will work on the HTML embed task this week.
- Test the embedded feature in different environments - WordPress, ghost, etc

Week 9, 10, 11: July 24 - July 30, July 31 - August 6 & August 7 - August 13

- Work on react-native components of embedded chat.
- I will set up the storybook for it.
- I will share my updates with my mentors regularly and work on the feedback provided.

Week 12: August 14 - August 20

- I will be working on the comment mode feature in React project.
- I will share my updates with my mentor and work on the feedback. I will also spend time in adding the docs that are yet to be created.
- I will add tutorials, maybe in a wiki, on the topics that I have mentioned in the implementation details.

Week 13: August 21 - August 28 (Final product and evaluation submission period, keep buffer)

- I will keep this week as a buffer to work on the feedback received from my mentors' evaluations.
- I also plan to complete any incomplete work from the above weeks due to any delay in the timeline before the second evaluation.