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

# Proposal: News Aggregation App, An All-in-One News Source

## 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**: Small (90 Hours)

## MENTOR

Abhinav Kumar (@abhinavkrin)

## ABSTRACT

The goal of this project is to create a Rocket.Chat (RC) app capable of delivering news to users from various sources, including their preferred news aggregators, RSS feeds, or news source providers offering API calls. The application will also provide news based on user preferences, news summarization, links to the original websites for full articles, personalized recommendations, 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, many platform users may want to have daily news readily accessible on the platform without navigating elsewhere. They might request the Rocket.Chat workspace admins to facilitate this, but the question remains: how can they achieve it? Currently, there isn't an RC app available in the marketplace that can fulfill this. Hence, why not create one? The aim is to design an RC app capable of sourcing news from diverse providers, categorizing and summarizing them, and offering users a convenient means to access and read news.

This project will help people looking for such integrations and Rocket.Chat as a whole. Its benefits would be:

- Convenience: Users can easily access their favorite news content while enjoying leisure moments with their teams.
- Increased Engagement: With news readily available, users will spend more time on the platform, leading to increased engagement and interaction within the community.
- Perspective Exchange: By offering this feature, admins can promote discussions among team members on various topics, encouraging critical thinking abilities, and welcoming different perspectives.
- Competitive Advantage: Providing news aggregation within the Rocket.Chat platform through this app will set Rocket.Chat apart from other communication platforms. This can attract new users and organizations seeking a platform that promotes the exchange of ideas among teams.

Overall, I believe that with this addition, we will not only enhance the user experience but also reinforce the platform's position as a hub for collaboration, knowledge sharing, and critical discussions. Accordingly, it will help Rocket.Chat grow even further in the market.

## GOALS

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

1. Developing a news aggregation RC app capable of sourcing news from diverse channels.
2. Offering users, the option to choose their news preferences based on categories ensuring a personalized experience.
3. Ensuring the app is properly configurable across three modes: globally configurable by admins, configuration specific to channels, and personalized configuration available within DMs.
4. Categorize the news from various sources into specific categories defined in the app.
5. Integrate LLM and web scrappers to provide summarization.
6. Providing related news articles within the thread on user request to let users easily understand the context of the news.
7. Documenting the work thoroughly and providing user-friendly documentation to use available features without hassle.

## DELIVERABLES

**By the end of GSoC, I plan to deliver a News Aggregation app with the capability to fetch news from different sources, summarize news, categorize news, and search for relevant articles.**

1. An RC news aggregation app capable of sourcing news either periodically or on-demand from various sources, including APIs if provided by the news source, RSS feeds, and other news aggregators if admins have premium or free subscriptions, all while ensuring compliance with their terms of use. **[NEW]**
2. Implementing categorization into specific set of categories defined in app by creating a manual mapping between source provided category and our created categories. **[NEW]**
3. Offering admins, the flexibility to choose from supported news sources, while also enabling a global configuration setting for all types of news that can be fetched based on categories, nationality, locality, etc. In addition, providing them with an option to allow or disallow the flexibility mentioned in points 4 and 5. **[NEW]**

4. Allowing channel creators to choose a specific category and source of news for their channel. For example, if a channel is related to sports, creators can have the flexibility to choose from globally allowed categories and providers for that channel. **[NEW]**
5. Providing the same flexibility as mentioned in point 4 to users who want news in their personal DMs with the news aggregation bot. **[NEW]**
6. Feature for news summarization covering all cases where the news API returns a small description, a full description, or no description at all, either by directly showing a small description or integration with LLMs, or using web scrapers with LLMs to provide summarization **[NEW]**
7. Feature to retrieve news articles relevant to news article that users wish to understand within the thread upon request, utilizing the Google Custom Search JSON API.

## 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 support for sentiment analysis to display a label tag based on the news. **[NEW]**
2. Providing categorization and personalization based on sentiments. **[NEW]**
3. Adding the option to categorize news into our defined categories from sources that don’t explicitly categorize news, using a custom ML model or LLMs. **[NEW]**
4. Adding support for chatting with LLMs directly through the app to discuss the news further, understand technical terms, context, etc., especially if the news is hard to comprehend. **[NEW]**

## IMPLEMENTATION DETAILS

**1. Collecting News from Various Sources –** The primary challenge is collecting news from diverse sources, each with varying methods of retrieval. Some news sources provide APIs to fetch news directly, while others rely on RSS feeds that need parsing. Additionally, some news sources or aggregators might require an API key for access. The problem also extends to finding the iterable data from the `response` being returned. While some news sources directly contain the necessary data in `response.data`, others may store it in `response.data.items`, `response.data.articles`, or various other formats. Furthermore, the issue also lies in the format of the JSON object for each iterated article, which overall makes the process challenging.

Therefore, to maintain consistency in processing and keep the code manageable, it is necessary to use the concept of abstraction. Also, to address the handling of variations in JSON structure and fetching methods, I propose using a simple idea: implementing a global class called `NewsProcessor`. Within this class, we can have a function like "processNews" which accepts multiple arguments such as the URL, a function named "findIterable" to find out the array that needs to be iterated from the response, and another function called "handleJsonStructure" to manage the JSON structure for each article. Both functions would be implemented by specific news source classes like ` BBCNewsHandler`, ` TechCrunchHandler` etc. to parse the data accordingly and make it usable by the global `NewsProcessor`. The following are some images to demonstrate the same:

```
interface Article {
    link: string;
    title: string;
    description: string;
    pubDate: string;
}
      You, 3 days ago • basic news app complete
You, 3 days ago | 1 author (You)
interface IterableNews {
    iterableData: Array<any>;
}
```

Fig. 1: An interface `Article` to maintain a specific format while presenting news.

```
class NewsProcessor {
    static async processNews(
        http: IHttp,
        url: string,
        context: SlashCommandContext,
        modify: IModify,
        dataHandler: (data: any) => NewsData,
        iterableHandler: (data: any) => IterableNews
    ) {
        const response = await http.get(url);
        const datas = response.data;
        const iterableData = iterableHandler(datas).iterableData;      You, 4 days ago

        if (Array.isArray(iterableData)) {
            const newsMessages: string[] = [];
            iterableData.forEach((data) => {
                const newsData = dataHandler(data);
                const message = `${newsData.title}\n ${newsData.description} \n
                ${newsData.link} \n (${newsData.pubDate})`;
                newsMessages.push(message);
            });

            await this.sendMessage(context, modify, newsMessages.join("\n"));
        } else {
            console.error("Unexpected data structure:", datas);
        }
    }
```

Fig. 2: A global class `NewsProcessor`, which requires specific variables to process and send news accordingly.

```
class BBCNewsHandler {
    static handleJSONStructure(data: any): NewsData {
        const link = data.link;
        const title = data.title;
        const description = data.description;
        const pubDate = data.pubDate;

        return { link, title, description, pubDate };
    }

    static findIterable(data: any): Array<any> {
        return data.items;
    }
}

You, 6 minutes ago | 1 author (You)
class TechCrunchHandler {
    static handleJSONStructure(data: any): NewsData {
        const link = data.link;
        const titleHTML = data.title.rendered;
        const descriptionHTML = data.excerpt.rendered;
        let description = "";
        let title = "";
        try {
            description = this.parseHTMLDescription(descriptionHTML);
            title = this.parseHTMLDescription(titleHTML);
        } catch (err) {
            console.log(err);
            title = titleHTML;
            description = descriptionHTML;
```

Fig. 3: Source-specific classes `BBCNews` and `TechCrunch` implementing the required functions.

```
class TechCrunchHandler {
    static findIterable(data: any): Array<any> {
        return data;
    }

    static parseHTMLDescription(html: string): string {
        const plainText = html
            .replace(/<[^>]*>/g, "")
            .replace(/&#\d+;/g, (match) =>
                String.fromCharCode(Number(match.slice(2, -1)))
            );
        const processedText = plainText.replace(/\[&[a-zA-Z]+;\]/g, "");
        return processedText;
    }
}

You, 7 minutes ago | 1 author (You)
class NewsApiAggHandler {
    static handleJSONStructure(data: any): NewsData {
        const link = data.url;
        const title = data.title;
        const description = data.description;
        const pubDate = data.publishedAt;

        return { link, title, description, pubDate };
    }

    static findIterable(data: any): Array<any> {
        return data.articles;
    }
}
```

Fig. 4: i) Parsing HTML into text as required in `TechCrunch`,
ii) `NewsApi` class implementing the necessary functions.

```
public async getNews(
        switch (subcommand) {
            case "bbcnews":
                console.log("Fetching BBC News");
                await NewsProcessor.processNews(
                    http,
                    // An external service running at 3003 to parse rss feed
                    "http://localhost:3003/bbcrss",
                    context,
                    modify,
                    BBCNewsHandler.handleJSONStructure,
                    BBCNewsHandler.findIterable
                );      You, 8 hours ago • basic news app complete

                break;
            case "techcrunch":
                console.log("Fetching TechCrunch News");
                await NewsProcessor.processNews(
                    http,
                    "https://techcrunch.com/wp-json/wp/v2/posts",
                    context,
                    modify,
                    TechCrunchHandler.handleJSONStructure.bind(
                        TechCrunchHandler
                    ),
                    TechCrunchHandler.findIterable.bind(TechCrunchHandler)
                );
                break;
```

Fig. 5: Processing and sending news messages to user on demand

The code snippets clearly show how I've built a basic prototype to fetch news from different sources. There is a class `NewsProcessor` that mainly works on fetching the news articles, processing them, and sending the news articles to the user on demand, irrespective of the news source. Then there are source-specific classes that implement the required functions to parse the JSON data accordingly by providing the correct iterable array and providing correct data that complies with the interface `Article `. Now, when news from any specific source is expected on demand, the `processNews` function is called, passing the URL and expected implementation of the function required to parse the incoming response. There can be various ways of abstracting it; I have shown just one way. Another way could be to have a parent class, and the other classes will implement the required function within it rather than passing it. A demo code snippet is shown in Fig. 6.

```
abstract class NewsHandler {
    abstract handleNews(data: any): Article;
    abstract handleNewsData(data: any): IterableNews;
}

You, 12 minutes ago | 1 author (You)
class BBCNewsHandler extends NewsHandler {
    handleNews(data: any): Article {
        const link = data.link;
        const title = data.title;
        const description = data.description;
        const pubDate = data.pubDate;

        return { link, title, description, pubDate };
    }       You, 3 days ago • basic news app complete

    handleNewsData(data: any): IterableNews {
        return { iterableData: data.items };
    }
}
```

Fig. 6: Specific news handler extending the base class

```
    if (sources.includes(subcommand)) {
        switch (subcommand) {
            case "bbcnews":
                console.log("Fetching BBC News");
                await this.processNews(
                    http,
                    "http://localhost:3003/bbcrss",
                    context,
                    modify,
                    new BBCNewsHandler()
                );
                break;
```

Fig. 7: Using specific source specific classes to fetch news accordingly

In the actual implementation, the exact structure will be decided after a discussion with the mentor. After understanding the general structure, the following section discusses details regarding the handling of each type of retrieval mechanism.

**Simple APIs that don't require authentications –** The source providers like `Techcrunch` that have endpoints like https://techcrunch.com/wp-json/wp/v2/posts which provide JSON data directly without any need for authentication. In such cases, the implementation can be simple as we need to call that api using `http.get(url)`, parse the data and show it to user. The following image demonstrates a simple implementation for fetching news from such API.

```
async fetchTechCrunch(
    modify: IModify
) {
    const response = await http.get(
        "https://techcrunch.com/wp-json/wp/v2/posts"
    );
    const datas = response.data;

    const messages: string[] = [];

    for (const data of datas) {
        const link = data.link;
        const title = data.title.rendered;
        const content = data.content.rendered;
        const message = `*${title}*\n ${content}\n ${link}\n`;
        messages.push(message);
    }

    const allMessages = messages.join("\n");

    this.sendMessage(context, modify, allMessages);
}
```

Fig. 8: Fetching news from APIs that don't require authentication

**Parsing RSS Feeds –** RSS stands for "Really Simple Syndication," and many news source providers use this content distribution method. In this method, we receive XML data that must first be parsed into JSON or a relevant format to extract data and send it to the user as message. There are many libraries like rss-parser, xml2js, etc., that can parse it. However, given the limited capability of Rocket.Chat apps with npm packages, it's hard to predict which packages will work or not. This process is experimental, and we may need to try multiple packages to make it work. I attempted to use the `rss-parser` library, but it didn't work. Hence, there are other ways to achieve this. One could be to simply experiment more and find a library that works, or to write a function to create our own RSS parser. In the actual implementation, the exact approach can be decided. However, considering the limited support for npm packages in rc-apps, and the potential need for other sections, such as scraping content from news, which might require npm libraries like `cheerio`, I propose creating our own external service to implement all such tasks that necessarily require npm packages not supported by rc-apps and exposing endpoints that our rc-app can hit. In the rc-app settings panel, we could provide an option to enable specific features, requiring the deployment of the external server, while other features will continue to work even without deploying it.

I will try to avoid dependency on an external service as much as possible and instead include it in the rc-apps itself. However, when necessary, this approach can be used.

Having said that, the following are the implementation details of RSS parsers running as an external service.

i. Import necessary modules `http` and `rss-parser` and instantiate a new parser object.

ii. Assign the URL of the RSS feed to the variable `rssFeedUrl`.

iii. Using `http.createServer() ` method to create an HTTP server to listen to incoming GET requests on `/bbcrss`.

iv. Parse the RSS feed using `parser.parseURL() ` when request is received.

v. Respond back with JSON data to the client.

```
const http = require('http');
const Parser = require('rss-parser');
const parser = new Parser();

const rssFeedUrl = 'https://feeds.bbci.co.uk/news/rss.xml';

const server = http.createServer(async (req, res) => {
  if (req.method === 'GET' && req.url === '/bbcrss') {
    try {

      const feed = await parser.parseURL(rssFeedUrl);
      const responseData = {
        title: feed.title,
        description: feed.description,
        items: feed.items.map(item => ({
          title: item.title,
          link: item.link,
          description: item.contentSnippet,
          pubDate: item.pubDate
        }))
      };

      res.writeHead(200, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify(responseData));
    } catch (error) {
      res.writeHead(500, { 'Content-Type': 'application/json' });
      console.log("Failed to fetch RSS feed");
      res.end(JSON.stringify({ error: 'Error fetching or parsing the RSS feed' }));
```

Fig. 9: Using rss-parser library on external service to parse and return JSON response

Once, this external service has been made, we can easily call our service to get the news which requires RSS parsers as shown in Fig. 7

**Using APIs that require authentication –** There are source providers or news aggregators that require an API key to access the service. In such cases, I propose adding a text box in the RC-apps settings where the admin can enable such a news service by providing their API key. To implement these settings, we can define our settings as ISetting[] type, which is an array of objects with the required properties as shown:

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

export const settings: ISetting[] = [
    {
        id: "news-api-api-key",
        i18nLabel: "NewsAPI Api key",
        i18nDescription:
            "Provide your NewsApi API key to access news content from this news aggregator",
        type: SettingType.STRING,
        required: true,
        public: false,
        packageValue: "",
    },
];
```

Fig. 10: Creating a setting to add a text box for registering an API key

Once this array is created, we must register it while extending configuration using the `configuration.settings.provideSetting(setting)` method, as shown.

```
export class NewsAggregationApp extends App {
    constructor(info: IAppInfo, logger: ILogger, accessors: IAppAccessors) {
        super(info, logger, accessors);
    }

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

Fig. 11: Registering all the required setting in extendConfiguration

Since the settings have been registered, users can now provide required setting in settings panel of rc-app as shown:

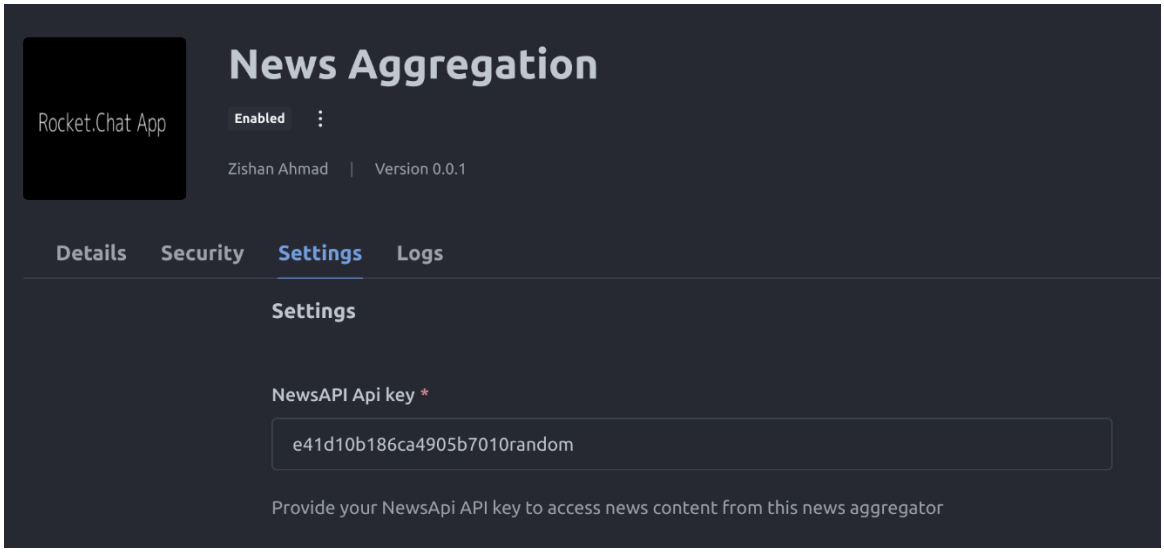


Fig. 12: User providing their api-key

Now that everything is established, the required settings can be fetched from the settings, and a regular API call can be made using our news processor shown earlier. The following code snippet demonstrates the same:

```
case "newsapi":
    const api_key = await this.app
        .getAccessors()
        .environmentReader.getSettings()
        .getValueById("news-api-api-key");

        await NewsProcessor.processNews(
            http,
            `https://newsapi.org/v2/top-headlines?country=in&apiKey=${api_key}`,
            context,
            modify,
            NewsApiAggHandler.handleJSONStructure,
            NewsApiAggHandler.findIterable
        );
    };      You, 1 second ago • Uncommitted changes

    break;
```

Fig. 13: Fetching news from `NewsAPI` after reading API key from settings

The following images demonstrates that the app is now capable of fetching news from various sources:

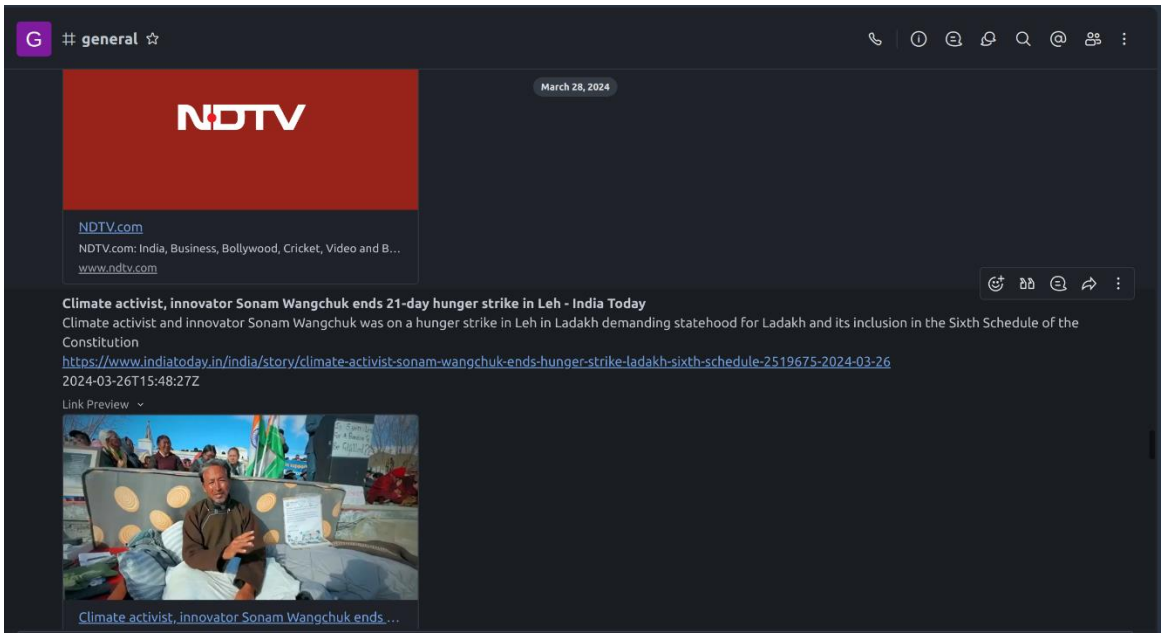


Fig. 14: Fetching news from NewsAPI in general channel

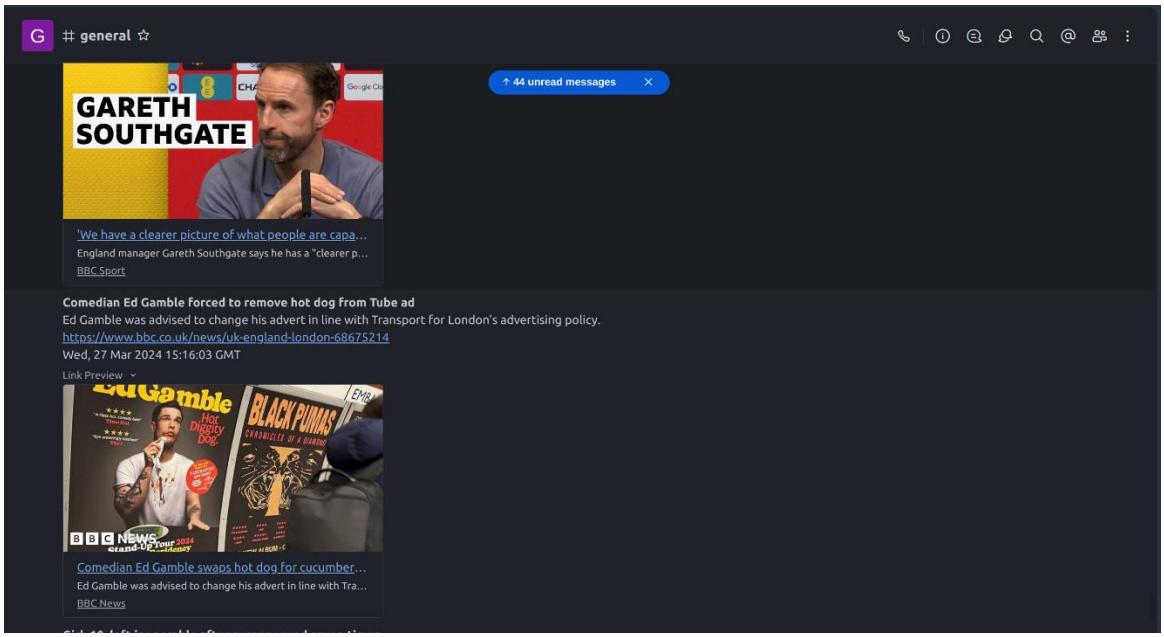


Fig. 15: Fetching news from BBC News from RSS Feed in general channel

Note: Currently, the implementation is set up in such a way that to fetch news from a source, one has to explicitly mention the news source in the slash command as a parameter like `/get-news techcrunch`. However, in the actual implementation, I will be utilizing preview blocks to offer options with available news sources to select from. The preview blocks can be implemented using the following code snippet, and then adding it into the message using `messageBuilder.setBlocks()`.

```
public createPreviewBlock(
    param: PreviewBlockParam
): PreviewBlockBase | PreviewBlockWithThumb {
    const { title, description, footer, thumb } = param;
    const previewBlock: PreviewBlockBase | PreviewBlockWithThumb = {
        type: LayoutBlockType.PREVIEW,
        title: this.createTextObjects(title),
        description: this.createTextObjects(description),
        footer,
        thumb,
    };

    return previewBlock;
}
```

Fig. 16: Creating preview block to list all the source providers when news is asked on demand

**2. Displaying news in the chat –** It is important that news displayed in the channel should have a consistent format irrespective of the news source to provide a consistent experience. The plan is to have the following structure:

- Title
- Small Description
- Summarization (On Demand)
- News Link
- Credit to Author
- Credit to Source
- Published Date
- Relevant articles in thread (On Demand)

The preview of the images will be taken care of automatically by the preview link feature of Rocket.Chat. In case of a link preview failure, a fallback image will be provided to maintain consistency in the implementation.

```
    private async sendMessage(
        context: SlashCommandContext,
        modify: IModify,
        message: string
    ): Promise<void> {
        const room = context.getRoom();
        const messageStructure = modify.getCreator().startMessage();
        messageStructure.setRoom(room);
        messageStructure.setText(message);
        await modify.getCreator().finish(messageStructure);
    }
```

Fig. 17: Implementation of sendMessage function.

To generate a message for the news, a variable `article` can be created as:

const article = `${title} \n ${description} \n ${link} \n ${author} \n ${sourceName} \n ${pubDate}`;

Once the variable is created, the following flow can be used to send a message:

i. Define a `sendMessage` function which accepts required parameters.

ii. In the function implementation, first fetch the room in which the news is requested.

iii. Create `messageStructure` using `modify.getCreator().startMessage();`

iv. Set the parameter message and determined room that to `messageStructure`.

v. Finishing it usng `modify.getCreator().finish(messageStructure)` will send a persistent message in the room.

In cases where a user demands news instead of default periodic news, rather than polluting the channel, the message can be sent as a notification visible only to the requester. Still, there will be an option to have persistent messages as well, when user wants to preserve them.

To implement messages in notification mode, the following approach can be taken:

i. Define a `notifyMessage ` function accepting required parameters.

ii. In the function implementation, first fetch the room in which the news is requested.

iii. Get a notifier using read.getNotifier() and make a messageBuilder using `notifier.getMessageBuilder()`;

iv. Set the parameter message that we received to `messageBuilder` and the determined room into `messageBuilder`.

v. Finishing it using `notifier.notifyUser(args) ` will send a notification in the room.

```
private async notifyMessage(
    context: SlashCommandContext,
    read: IRead,
    sender: IUser,
    message: string
): Promise<void> {
    const room = context.getRoom();
    const notifier = read.getNotifier();
    const messageBuilder = notifier.getMessageBuilder();
    messageBuilder.setText(message);
    messageBuilder.setRoom(room);
    return notifier.notifyUser(sender, messageBuilder.getMessage());
```

Fig. 18: Implementation of notifyMessage function.

### 3. Personalization options for admins, channel creators, and users –

Having the flexibility to choose the news providers, set category preferences, country preferences, etc., is necessary to provide a personalized feel to the user. Hence, I propose personalizing these options at three levels: 1) Workspace level, 2) Room Level, 3) User Level.

**Workspace level –** This is like the global preference setting for the entire workspace. The options to choose from different news source providers, news category preferences, country preferences and other preferences that are permitted in the workspace will be accessible from the settings panel of the RC-app. Based on these settings, the app will look into them before providing any news in the channel. This will help the admins in controlling the type of news that can be fetched in the rooms/direct messages. The implementation details are discussed as follows:

The required settings, such as the option to choose from available news sources and selecting the country for which the news has to be fetched, are defined in the settings array of type ISetting. The following code snippet defines MultiSelect options to choose from various options. For example, in the current implementation, three news source providers, namely 'TechCrunch', 'BBC News', and 'NewsAPI Aggregator', are allowed. Admins can choose which of the supported news providers to keep by simply selecting those providers. Similarly, in NewsAPI, there is an option to pass the country to fetch news country-wise, so all the possible countries for which the news can be fetched will be selected from the RC-app settings panel.

```
export const settings: ISetting[] = [
    {
        id: "source-select",
        i18nLabel: "Select from supported news providers",
        i18nDescription:
            "Help admins allowing / disallowing a certain news providers",
        type: SettingType.MULTI_SELECT,
        values: [
            { key: "bbcnews", i18nLabel: "BBC-News" },
            { key: "techcrunch", i18nLabel: "Techcrunch" },
            { key: "newsapi", i18nLabel: "News API Aggregator" },
        ],
        required: true,
        public: true,
        packageValue: "",
    },

    {
        id: "country-select",
        i18nLabel: "Select news country wise",
        i18nDescription: "Help admins to choose country specific news",
        type: SettingType.MULTI_SELECT,
        values: [
            { key: "in", i18nLabel: "India" },
            { key: "us", i18nLabel: "United States" },
```

Fig. 19: Code snippet for defining the required settings in RC-app settings panel

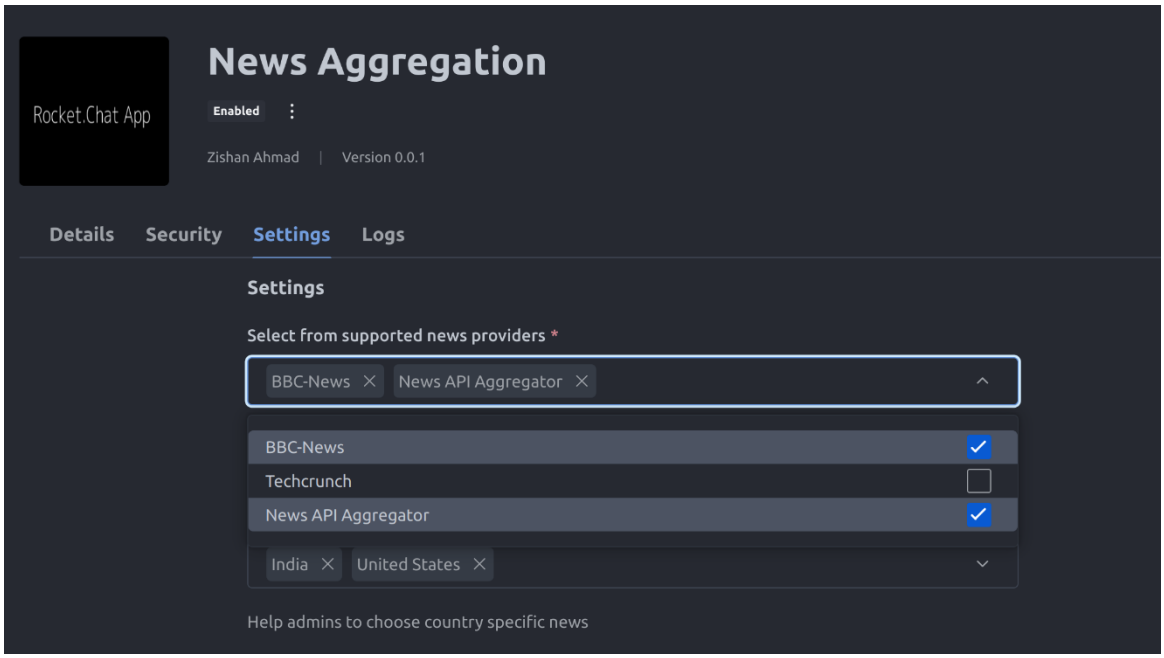


Fig. 20: Admin disabling the TechCrunch source provider in the workspace

Now, once the settings have been set, before fetching any news, the application will first retrieve the settings. It will then check if the source provider is included in the settings for which news has been requested. If it is included, only then will the news be fetched. Additionally, in the case of NewsAPI, only news from the countries defined in the app settings will be fetched upon request. This is just for demonstration purposes, and more ways of handling other properties are required. The following code snippets show the implementation.

```
        const globalSource = await this.app
            .getAccessors()
            .environmentReader.getSettings()
            .getValueById("source-select");

        const globalCountries = await this.app
            .getAccessors()
            .environmentReader.getSettings()
            .getValueById("country-select");

        this.getNews(
            http,
            context,
            modify,
            subcommand,
            globalSource,
            globalCountries
        );
    }
```

Fig. 21: Global settings being fetched from app settings


Fig. 22: Checks if source is allowed in the settings, then only fetch the news.


Fig. 23: Fetching news of all countries permitted by admin in the workspace

As shown in Fig. 21, Before fetching the news, the app will first look at the ISetting to find the key "country-select" to search for all the countries for which news is allowed in the global workspace setting. Once this has been determined, in the current implementation, it fetches from all the allowed countries in a loop, as NewsAPI doesn't provide a way to concatenate all the country names in one request. This could potentially overload the server. Hence, in APIs like this, the plan is to fetch global news, and if requested by the user, fetch country-wise news upon user request only. For APIs that allow country concatenation, the current approach can be used.

So, admins have the flexibility to enable/disable source providers, countries for which news can be fetched, and other preferences.

But consider a scenario where someone creates a room for sports discussion, and the room owner wants to enable only news from the sports category and specifically from BBC News as the source. How will they do this? Or imagine a situation where a user in that workspace requests news from the news aggregation bot in a direct message (DM), specifying preferences for only tech news from TechCrunch. How can they achieve this? I have addressed this through room/user-level personalization settings.

**Room level / User level –** I am covering the implementation details for both in the same section, as we know that even DMs are treated as rooms in Rocket Chat, where their RoomId is just a combination of the user ID and bot ID, making it unique to the user. Hence, we can utilize this to manage DM preferences. A small image illustrating this is attached below:

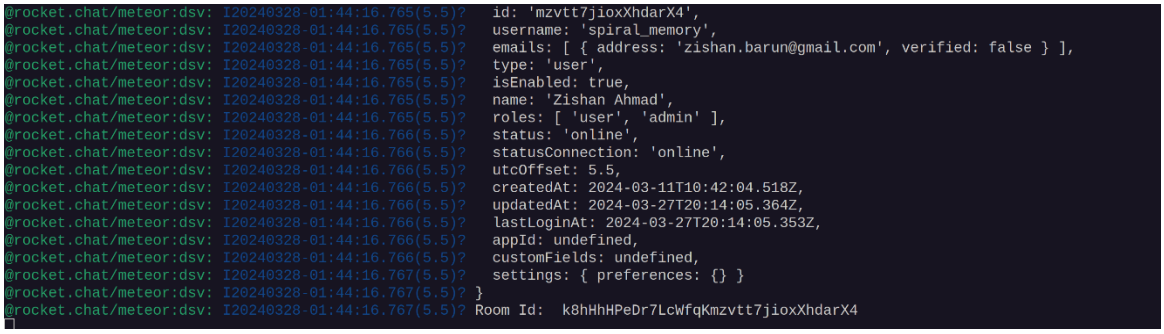


Fig. 24: DMs RoomId containing a portion of the userId.

To have personalization in a room or in DMs, Users simply need to execute a command or use an action button (currently implemented via slash command), which will open a UI-Kit modal. This modal will display the options enabled globally. Now, the room owner or the user requesting news in DMs will choose from those options to set up their room/personal preferences.

To implement this functionality, I propose using UI-Kit modal for choosing the room / DMs news preferences and once the Save button is clicked, the preferences will be saved into persistence storage, from which it will be fetched to provide periodic / demand-based news.

The following image shows how the UI-Kit modal will look while user choosing their preference:

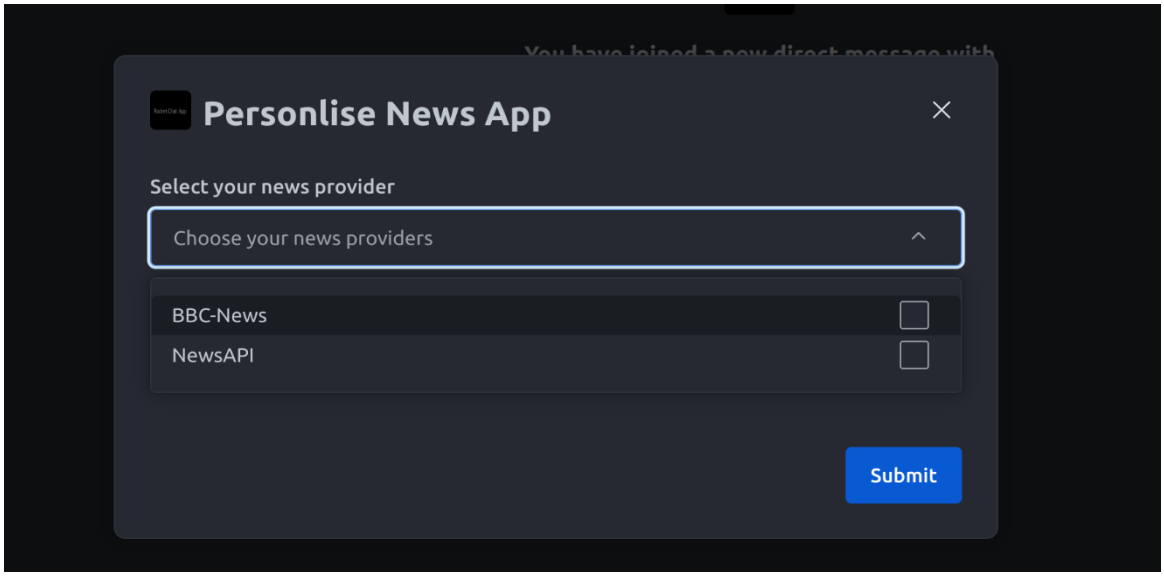


Fig. 25: User selecting their room/personal preferences

As shown in Fig. 25, the UI-Kit modal will allow the user to set their own source provider as room / personal preferences. Notice that `TechCrunch` is not available for selection. Remember? It was disabled globally by the admin.

Now, once the Submit button is clicked, the preferences will be saved in persistence storage with roomId as the key. However, there is a problem here: we can't extract roomId where the Submit button is clicked in the `executeViewSubmitHandler`. Even if we try, the roomId will be undefined. Then the question arises: how will we be able to access roomId and save preferences corresponding to a room?

**Q. How to extract roomId where the submit button has triggered?**

Indeed! Every problem has a solution, including this one. To pinpoint the roomId when submit button is triggered, following steps can be taken:

i. When the slash command is executed, extract the roomId from `SlashCommandContext`, which can be done.

ii. Store that roomId with the key as userId in persistence storage as we have access to both here.

iii. Now, when `executeViewSubmitHandler` is triggered, find out the userId, which can be done easily using `const {user, view} = context.getInteractionData() `.

iv. With the help of the userId, look for the roomId it is associated with.

v. Now, we have the roomId that we needed.

vi. Clear this storage to avoid any conflicts with other functionalities.

The following code snippets shows the implementation of the same:

```
You, 12 hours ago | 1 author (You)
interface IRoomInteractionStorage {
    storeInteractionRoomId(roomId: string): Promise<void>;
    getInteractionRoomId(): Promise<string>;
    clearInteractionRoomId(): Promise<void>;
}

You, 12 hours ago | 1 author (You)
export class RoomInteractionStorage implements IRoomInteractionStorage {
    private userId: string;
    constructor(      You, 12 hours ago • basic news app complete
        private readonly persistence: IPersistence,
        private readonly persistenceRead: IPersistenceRead,
        userId: string
    ) {
        this.userId = userId;
    }

    public async storeInteractionRoomId(roomId: string): Promise<void> {
        const association = new RocketChatAssociationRecord(
            RocketChatAssociationModel.USER,
            `${this.userId}#RoomId`
        );
        await this.persistence.updateByAssociation(
            association,
            { roomId: roomId },
            true
        );
    }
```

Fig. 26: Implementation of RoomInteractionStorage class and storeInteractionRoomId function

```
    async executor(
        context: SlashCommandContext,
        read: IRead,
        modify: IModify,
        http: IHttp,
        persis: IPersistence
    ): Promise<void> {
        const [subcommand] = context.getArguments();
        console.log("Personalise req received");

        const room = context.getRoom();
        const user = context.getSender();

        const persistenceRead = read.getPersistenceReader();
        const roomInteraction = new RoomInteractionStorage(      You, 12 hours ago • bas
            persis,
            persistenceRead,
            user.id
        );
        await roomInteraction.storeInteractionRoomId(room?.id);
```

Fig. 27: Saving the roomId in persistence storage when slash command is executed

```
You, 4 seconds ago | 1 author (You)
export class ExecuteViewSubmitHandler {
    constructor(
        private readonly app: NewsAggregationApp,
        private readonly read: IRead,
        private readonly http: IHttp,
        private readonly modify: IModify,
        private readonly persistence: IPersistence
    ) {}

    public async run(context: UIKitViewSubmitInteractionContext) {
        const { user, view } = context.getInteractionData();
        console.log(view.state);
        const persistenceRead = this.read.getPersistenceReader();
        const roomInteractionStorage = new RoomInteractionStorage(
            this.persistence,
            persistenceRead,
            user.id
        );
        const roomId = await roomInteractionStorage.getInteractionRoomId();
```

Fig. 28: Fetching the roomId where the submit button triggered

Now that the roomId has been identified, follow these steps to save the preferences associated with it in the persistent storage:

i. Extract the view using const {, view} = context.getInteractionData().

ii. Look for view.state.

iii. Save that state in the preference storage.

The implementation for preference storage and the method for saving the preferences in it are shown as follows

```
You, 12 hours ago | 1 author (You)
export class PreferenceStorage implements IRoomInteractionStorage {
    private roomId: string;
    constructor(
        private readonly persistence: IPersistence,
        private readonly persistenceRead: IPersistenceRead,
        roomId: string
    ) {
        this.roomId = roomId;
    }

    public async storePreference(preferences: any): Promise<void> {
        const association = new RocketChatAssociationRecord(
            RocketChatAssociationModel.ROOM,
            `${this.roomId}#Preferences`
        );
        await this.persistence.updateByAssociation(
            association,
            { preferences: preferences },
            true
        );
    }
```

Fig. 29: Implementation of storePreference function in PreferenceStorage class

```
public async getPreference(): Promise<any> {
    try {
        const association = new RocketChatAssociationRecord(
            RocketChatAssociationModel.ROOM,
            `${this.roomId}#Preferences`
        );
        const result = await this.persistenceRead.readByAssociation(
            association
        );
        return result;
    } catch (err) {
        console.log(err);
        return null;
    }
}
```

Fig. 30: Implementation of getPreference function in PreferenceStorage class

```
        const preferenceStorage = new PreferenceStorage(
            this.persistence,
            persistenceRead,
            roomId
        );
        await preferenceStorage.storePreference(view.state);
        await roomInteractionStorage.clearInteractionRoomId();
        return {
            success: true,
        };
    }
}
```

Fig. 31: Saving the room / user level personalization preferences

After storing the preferences, when a request is made from a room or DM, preferences will be read, and bot will respond accordingly. Some images are shown for demonstration:

```
async executor(
    http: IHttp,
    persis: IPersistence
): Promise<void> {
    const [subcommand] = context.getArguments();
    if (!subcommand) {
        throw new Error("Error!");
    }

    const room = context.getRoom();
    const persistenceRead = read.getPersistenceReader();
    const preferenceStorage = new PreferenceStorage(
        persis,
        persistenceRead,
        room.id
    );

    const preferences = await preferenceStorage.getPreference();      You, 12 hours ago • basic
    console.log(preferences);
    if (!(preferences.length === 0)) {
        const sourcePreferece =
            preferences[0].preferences.sourceSelectId.sourceAction;
        const countryPreference =
            preferences[0].preferences.countrySelectId.countryAction;

        this.getNews(
            http,
```

Fig. 32: Fetching preferences and accordingly responding back to the user

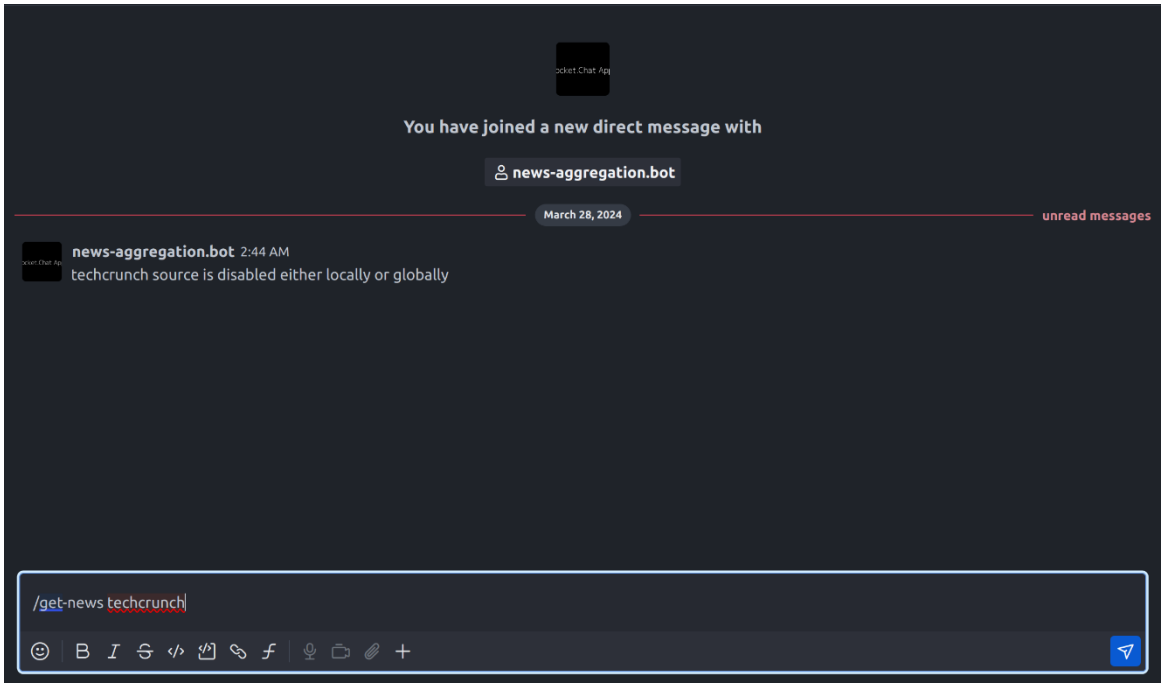


Fig. 33: News aggregation bot responds with a message indicating that TechCrunch is disabled either globally or in the channel, instead of displaying the news.

Hence, now, in the case of periodic news, the preferences will be read, and the news will be returned accordingly. In the case of on-demand news, the bot will not allow fetching news that are globally disabled or disabled in the channel. However, in the case of DMs, if explicitly requested by the user, other news can also be fetched without any issues.

**4. News Summarization in all cases –** One unique feature of this news aggregation app is its ability to summarize news in all possible cases. Some source providers offer a small summary as a description in the JSON response directly, while others provide a detailed description in the response, and some don’t provide any description at all. Now, in cases where the source doesn’t provide any description, we have two subcases: summaries are already available on their website, and even on the website, we don’t have a summary available. Hence, in this section, I will describe ways to handle each of those cases.

**Description provided is brief in the response –** In this scenario, they've already made our lives easy. We can use the concise description provided and include it in our message upon request for a summary. If a user desires summaries to be automatically added to all news, they'll have the freedom to select their preferred method.

**A complete description is provided in the response. –** In this case, we can directly use our LLMs directly or using LangChain and integrating different open-source models with a suitable prompt to summarize the news and return a summary. This summary will then be attached to the message when requested or automatically included depending on the preference.

**No description is provided in response –** In this case, there can be two subcases, as discussed. So, let's dive into each of them in detail.

> **Subcase 1**: Summary Available on the Website - In this scenario, we can use libraries like 'cheerio' to extract the news summary. We simply need to identify the div where the summary is provided, extract all the relevant information from that div, refine the text, and then present it as the summary.
>
> **Subcase 2**: No Summary Available on the Website - In such instances, we need to identify the div containing the full article, extract the entire article, make a request to the LLM with a well-crafted prompt, and then use the AI response to generate a summary.

I have tried all such cases, and here are the relevant code snippets and demo images for them, along with the required explanations:

In cases where a small description is provided, it is self-explanatory to simply use that description and attach it to the message either while sending or upon request later by using `messageExtender`. Now, in cases where a large description is provided, we use that description, create a substring as LLMs have word limits, call the LLM, and attach the response to the message as demonstrated:

```
async fetchTechCrunch(
    http: IHttp,
    context: SlashCommandContext,
    modify: IModify
) {
    const response = await http.get(
        "https://techcrunch.com/wp-json/wp/v2/posts"
    );
    const datas = response.data;
    datas.map(async (data) => {
        const link = data.link;
        const title = data.title.rendered;
        let content = data.content.rendered;
        content = content.substring(100, 300);
        const res = await http.post(getApiUrl(), getPayload(content));
        const summarisedContent =
            res?.data?.candidates?.[0]?.content?.parts?.[0]?.text ??
            `Unable to Summarise Content`;
        const message = `*${title}*\n
        *Summarised Content ->*  ${summarisedContent}\n ${link}`;
        this.sendMessage(context, modify, message);
```

Fig. 34: Adding summarized news content from long description

```
const getApiUrl = () =>
    `https://generativelanguage.googleapis.com/v1beta/models/gemini-pro:generateContent
    ?key=AIzaSyA9Qve3DrandomAS63iWV_Pmerandom`;
const getPayload = (content) => {
    const data = {
        contents: [
            {
                parts: [
                    {
                        text: `Summarise the following news article content: ${content}`,
                    },
                ],
            },
        ],
    };

    const headers = {
        "Content-Type": "application/json",
    };

    return {
        data,
        headers,
    };
};
```

Fig. 35: Relevant function required to make call to LLMs

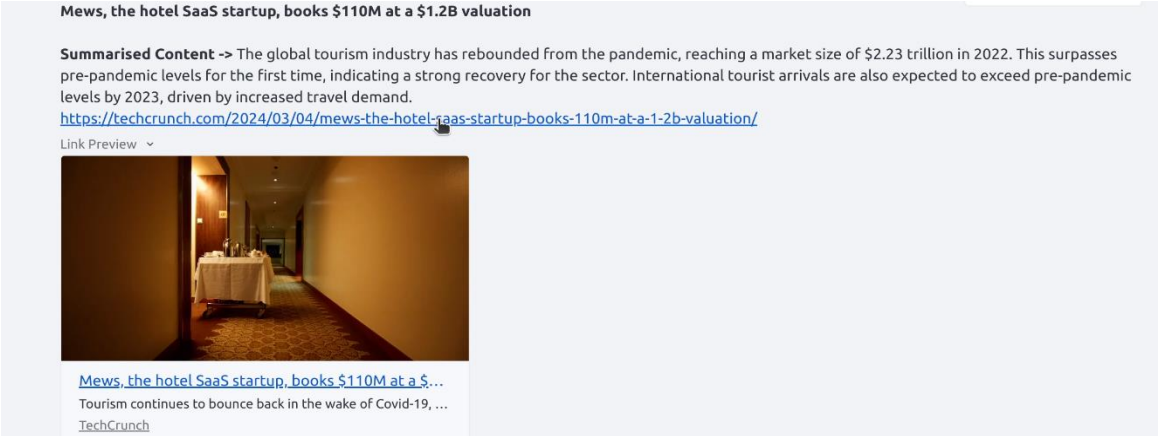


Fig. 36: User getting summarized content attached with the news

Now, let's discuss the case where a no description is provided, in that the flow will be as follows:

i. Use `cheerio` library directly in RC-app to scrap the news or use as external service (I have made it as external service for demonstration)

ii. When summarization is requested, extract the news URL from the message and call this external service by providing URL as param.

iii. External service will fetch the content from that link and use `cheerio` library to extract the text from the relevant `div`.

iv. If the scraped text is already a summary return that as response, Otherwise, the scraped text will be feeded to LLMs to generate summary and then provide that summary as response.

v. Use that response in rc app to append summary to the message using `messageExender`

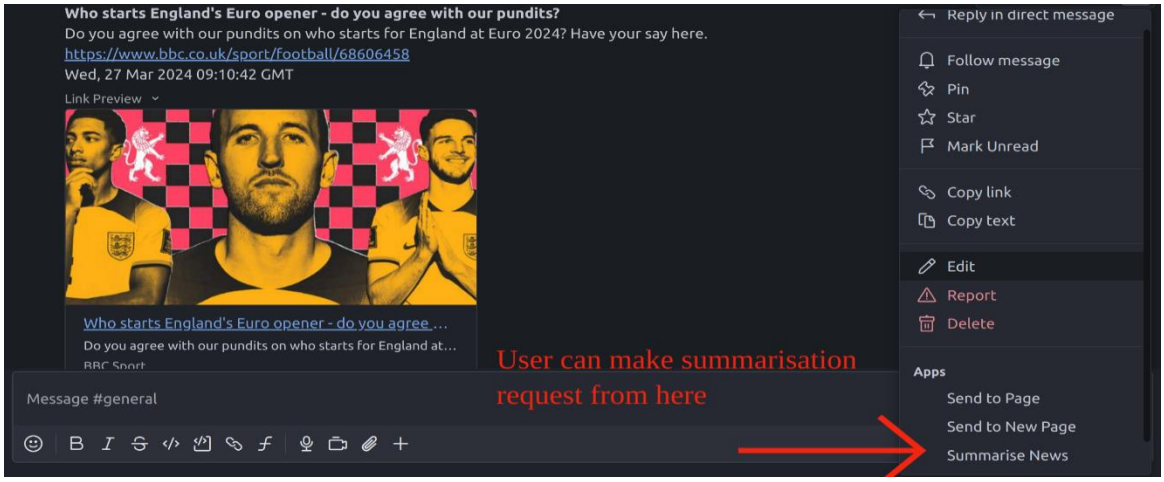


Fig. 37: User making a summarization request

```
export class ExecuteActionButtonHandler {
    public async handleActions(): Promise<IUIKitResponse> {
                const pattern: RegExp = /(https?|http):\/\/\S+/;
                const match: RegExpMatchArray | null | undefined =
                    messageText?.match(pattern);

                if (match) {
                    const link: string = match[0];

                    const response = await this.http.post(
                        "http://localhost:3002/scrape",
                        {
                            headers: { "Content-type": "application/json" },
                            content: link,
                        }
                    );

                    console.log(response.content);

                    const messageExtender = await this.getMessageExtender(
                        messageId,
                        sender,
                        this.modify
                    );
                    messageExtender.addCustomField("key", 1);
                    const messageAttachment: IMessageAttachment = {
                        text: "News Summary: " + response.content,
                    };
                    messageExtender.addAttachment(messageAttachment);
                    await this.modify.getExtender().finish(messageExtender);
                }
```

Fig. 38: Extracting news URL from message and making request to external service

```
You, 20 hours ago | 1 author (You)
const http = require("http");
const axios = require("axios");
const cheerio = require("cheerio");

async function scrapeWebsite(url) {
    try {
        const response = await axios.get(url);
        const html = response.data;
        const $ = cheerio.load(html);
        const olElement = $("ol.gs-u-m0.gs-u-p0.lx-stream__feed.qa-stream");
        const textItems = olElement
            .find("li")
            .map((index, element) => $(element).text())
            .get();

        const allText = textItems.join("\n");

        return allText;
    } catch (error) {
        console.error("Error fetching data:", error);
        throw error;
    }
}
```

Fig. 39: Scraping news content from BBC News

```
async function generateResponse(scrapedText) {
    if (scrapedText.trim() === "") {
        return "No news description found";
    }
    const apiKey = "random30T90hTPATeJdt_random";
    const url =
        "https://generativelanguage.googleapis.com/v1beta/models/gemini-pro:generateContent?key=" +
        apiKey;


    const requestData = {
        contents: [
            {
                parts: [
                    {
                        text: `${prompt}`      You, 1 second ago • Uncommitted changes
                    },
                ],
            },
        ],
    };

    try {
        const response = await axios.post(url, requestData, {
            headers: {
                "Content-Type": "application/json",
            },
        });
        return response.data.candidates?.[0]?.content?.parts?.[0]?.text;
    } catch (error) {
        console.error("There was a problem with the request:", error);
```

Fig. 40: Making request to LLM with our scraped text

```
const prompt =  `Generate a concise summary of the news article provided, adhering to the following guidelines:

Remove Introductory Text: Omit any text preceding the main content of the news article,
including phrases like "Posted at," "Copy this link," or similar indicators.

Use Text: Use text following the "Posted at" segment, as it will contain the relevant parts of the news.

Eliminate Social Media Text: Exclude any text related to sharing the article on social media platforms
such as Facebook, Twitter, etc. Ensure the summary does not contain references specific to social media.

Concise and Clear: Provide a succinct summary that captures the key points, main events,
and significant details of the news article. The summary should be easy to understand
and free from unnecessary elaboration.

Generate the summary with these guidelines under 300 words making sure you only
eturn summary with no heading or extra information. Scraped news article: ${scrapedText},
If scraped news article is empty, return the response "No news description found instead of hallucinating"`
```

Fig. 41: An example of well-crafted prompt


Fig. 42: User getting news summary as response

**5. Delivering periodic news to the user –** Earlier, in the section, I discussed getting the news on demand. However, in the news aggregation app, I'm also proposing to provide users with periodic news updates at a preferred schedule set by them. To achieve this, the plan is to offer a UI-Kit modal to set the preferred time when users may want to receive periodic updates and how frequently they want them. Once these preferences are saved, we can utilize the scheduler provided in the RC apps to schedule fetching the news. The overview of the implementation details with dummy code is as follows:

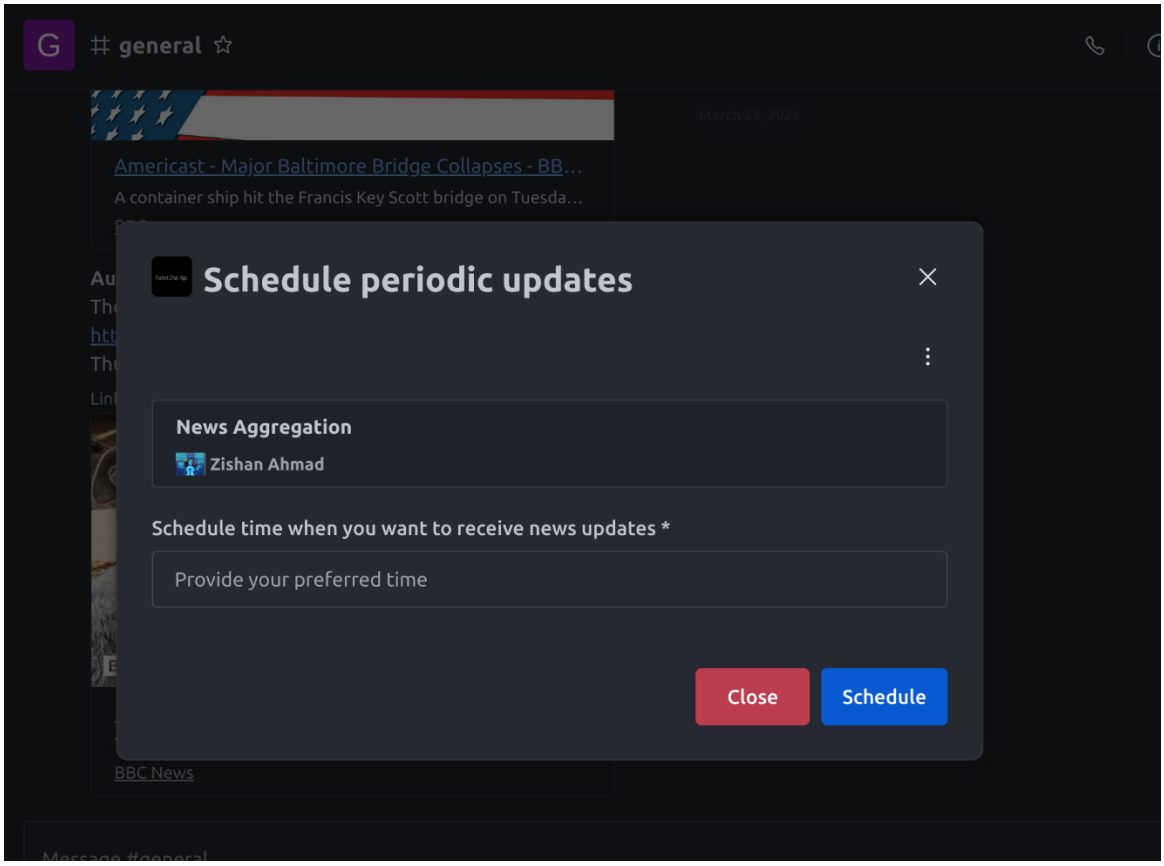


Fig. 43: User scheduling his preferred time to get periodic updates

```
    protected async extendConfiguration(
        configuration: IConfigurationExtend,
        environmentRead: IEnvironmentRead
    ): Promise<void> {
        configuration.scheduler.registerProcessors([
            {
                id: "periodicNews",
                processor: async (news) =>
                    provideNews.provide(news)
            },
        ]);      You, 24 hours ago • basic news app complete
    }
```

Fig. 44: User scheduling his preferred time to get periodic updates

```
async function scheduleNews(modify: IModify, News: any) {
    const task = {
        id: "periodicNews",
        interval: "86400 seconds",
        data: { news: News },
    };
    await modify.getScheduler().scheduleRecurring(task);
}
```

Fig. 45: Scheduling news when periodic updates are set

Note: The code provided for scheduling is merely just a dummy example to showcase how it might work. I haven't tested the scheduling yet.

**6. Delivering relevant news articles within the thread –** There are instances when we come across news articles without context, and we may wish to explore related articles to gain a better understanding. Therefore, I propose using the Google Custom Search API to retrieve relevant articles related to a news topic and publish them for users within the thread of that news.

The flow for the implementation is as follows:

i. The user requests relevant news articles by clicking the option from the Message Action button.

ii. Extract the title of the news from the message and attach a keyword like 'Show some relevant articles for ${title}'.

iii. Fetch the CX and API key of the Google Search API from the RC-app settings.

iv. Make a call to this API to fetch related articles.

v. Send the title and URL of these articles into the thread of the news for which relevant articles were requested.

The following are some images and code snippets showing the demonstration of the same:

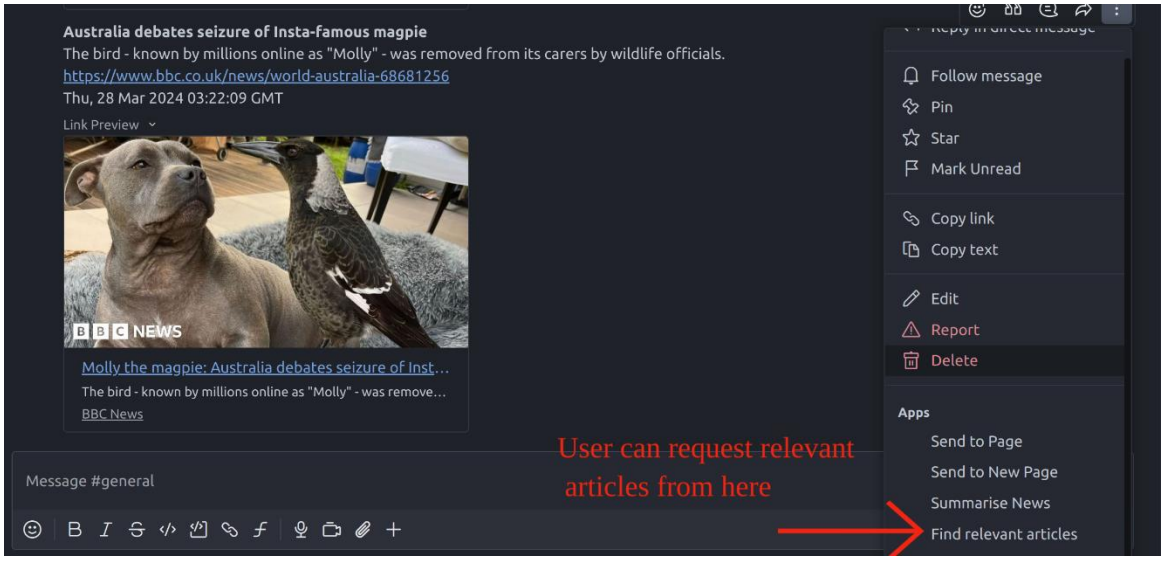


Fig. 46: User requesting for relevant articles

```
export class ExecuteActionButtonHandler {
    public async handleActions(): Promise<IUIKitResponse> {
                await this.modify.getExtender().finish(messageExtender);
            }

        case "relatedarticles":
            console.log(messageId);
            const g_api_key = await this.app
                .getAccessors()
                .environmentReader.getSettings()
                .getValueById("google-search-api-key");

            const cx = await this.app
                .getAccessors()      You, 22 hours ago • added relevant article option
                .environmentReader.getSettings()
                .getValueById("google-search-engine-id");

            const regex = /\*(.*?)\*/;
            const matchedText = messageText?.match(regex);
            const queryText = matchedText ? matchedText[1] : null;


            const response = await this.http.get(
                `https://www.googleapis.com/customsearch/v1?key=${g_api_key}&cx=${cx}&q=${encod
                `News related to : ${queryText}`
                )}`,
                {
                    headers: { "Content-type": "application/json" },
                }
            );

            const data = response.data;
```

Fig. 47: When requested, the message is extracted, keys are fetched, and a request is made to Google Search API

```
export class ExecuteActionButtonHandler {
    public async handleActions(): Promise<IUIKitResponse> {


            const response = await this.http.get(
                `https://www.googleapis.com/customsearch/v1?key=${g_api_key}&cx=${cx}&q=${encod
                `News related to : ${queryText}`
                )}`,
                {
                    headers: { "Content-type": "application/json" },
                }
            );

            const data = response.data;
            if (data.items) {
                data.items.forEach((item: any) => {
                    this.sendMessage(
                        room,
                        this.modify,
                        messageId,
                        item.title,
                        item.link
                    );
                });
            } else {
                console.log("No results found");
            }
        }

        return this.context.getInteractionResponder().successResponse();
    }
```

Fig. 48: Sending all relevant articles in threads

```
private async sendMessage(      You, 22 hours ago • added relevant article option
    room: IRoom,
    modify: IModify,
    messageId: string,
    title: string,
    url: string
) {
    const message = `*${title}*\n ${url}`;

    const messageStructure = modify.getCreator().startMessage();
    messageStructure.setRoom(room);
    messageStructure.setThreadId(messageId);
    messageStructure.setText(message);
    await modify.getCreator().finish(messageStructure);
}
}
```

Fig. 49: Function to sendMessage into the thread

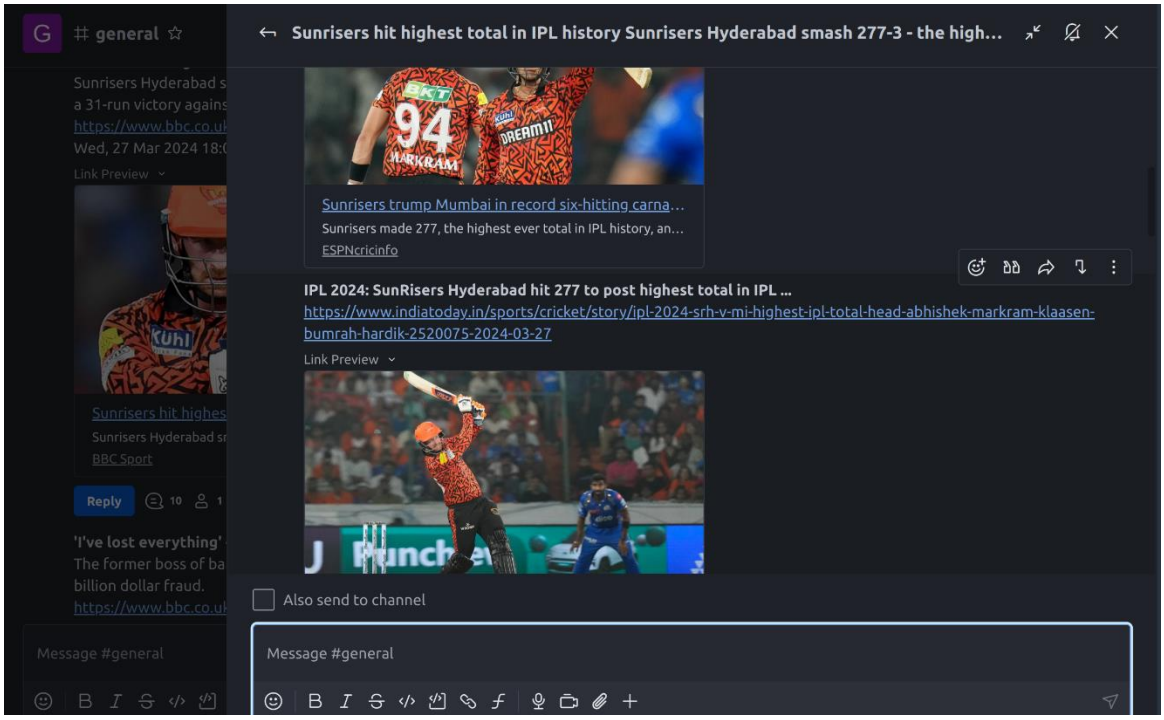


Fig. 50: User receiving related articles in thread

**7. Documentation –** As News Aggregation offers many features, such as personalization at 3 different levels, choice of multiple source providers, summarization, and showing related news in the thread, it becomes difficult for admins to experiment and explore to understand everything. Therefore, I will keep the following things in mind:

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

ii. I will address any frequently asked questions that may arise while configuring the “News Aggregation” app

iii. I will ensure that the documentation is regularly updated to reflect any changes or additions made to the “News aggregation” app, ensuring users have access to the latest information and features.

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

## DETAILED WORK PLAN

**My proposed timeline for the task will be as follows:**

**Before May 01**

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

### Community Bonding Period Begins!

**May 01- May 26**

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

## Coding officially begins!

**Week 1 [May 27 – June 03]**

- Setting up the development environment.
- Identifying various news sources and determining the method to extract content from their websites.
- Implementing an abstract function to retrieve news via APIs.
- Extending this function to fetch and process news from diverse sources that provide APIs, knowing that each may have a different JSON response structure.

**Week 2 [June 04 – June 10]**

- Work on integrating RSS parsers to extract news from sources that offer RSS feeds.
- Ensure that news obtained from APIs or processed through RSS parsers is accurately rendered in the app as a daily digest or periodically using scheduling mechanisms.

**Week 3 [June 11 – June 17]**

- Determine the set of categories that best suit all the selected news providers for the app.
- Work on creating a mapping between the categories offered by the news source providers and mapping them to the categories defined in the app.
- Start working on a feature to provide personalization based on the source and categories.
- Test all the changes made up to now.
- Work on bug fixes to ensure the code is bug-free.

**Week 4 [June 18 – June 24]**

- Work on finding the best web scraper by trying out multiple scrapers.
- Use this web scraper on news sources that don’t provide APIs or RSS feeds.
- Properly render those scrapped content in the news app in news feed format.
- Extend the mapping for categories mechanism to this as well.

## Midterm Evaluation

**June 25- June 29**

- Work on drafting a report for midterm evaluations.
- Submit the midterm report.
- Start working on the next feature, i.e., web scrappers for the news source which doesn't provide APIs and have RSS feeds.

**Week 5 [June 30 – July 05]**

- Work on providing on-demand news using slash commands and preview blocks with action buttons.
- Provide users with the option for summarizing the news content if a small description is provided by the news source.
- If a large description is provided by the news source, work on integrating LLMs to facilitate summarization.

## Coding continues!

**Week 6 [July 06 – July 12]**

- If no description is provided through APIs or RSS feeds, use web scrapers to scrape contents.
- Utilize our LLMs to summarize the content.
- Start working on providing an option for news association.

**Week 7 [July 13 – July 19]**

- Extract keywords from the news.
- Search for those keywords using the Google Custom Search JSON API.
- Use the returned results to display related links in the thread of that news.
- Work on some experimental features.
- Try handling hallucinations of LLMs by checking confidence scores, if possible, and add an option to regenerate the summary.
- Attempt to integrate multiple LLMs and allow the user to choose between them.

## Final Evaluation

**July 20-July 27**

- 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 mentor.

# RELATED WORK

I have been contributing to various repositories of Rocket.Chat since January, including Apps.Notion, which is a Rocket.Chat app. Thus, I have familiarized myself with the RC app's structure, its code flow, as well as its functionalities. While contributing to Apps.Notion and attending weekly RC app development workshops, I have already made strides towards learning about Slash commands, UI-Kit modals, Preview Blocks, Message Action Buttons, RC App Settings, integrating LLMs, etc., into RC apps. I have also worked on integrating the UI-Kit into EC, giving me a good understanding of how it works.

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

1. **[MERGED]** Apps.Notion #48 - **[ENHC]** Overflow menu to go to subpage modal.
2. **[MERGED]** Apps.Notion #53 - **[FIX]** Title input visibility issue.
3. **[MERGED]** Apps.Notion #51 - **[ENHC]** Close window after connection success/failure.
4. **[MERGED]** EmbeddedChat #481 : Added support for UI-Kit Modal rendering and added support for multiple params commands.
5. **[OPEN]** Apps.Notion #65 - **[ENHC]** Add content to page through Apps.Notion.
6. **[OPEN]** Apps.Notion #70 - **[ENHC]** Update DB Records through Apps.Notion

In addition to these related PRs, I also attempted to develop prototypes for all required features in News Aggregation app so that during the GSoC period, I can develop a fully functional News Aggregation RC App with minimal effort and without spending excessive time understanding the problem.

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, Abhinav Kumar, one of the maintainers for Embedded Chat project and now my mentor for this project, noticed me helping people with build issues. [Abhinav, if you're the one reading this, I would like to thank you for being so welcoming and supportive throughout my open-source journey]. 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, Apps.Notion, Embedded Chat, 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/Apps.Notion

- Issues (Total: 5)

Rocket.Chat/EmbeddedChat

- Issues (Total: 24)

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]** Apps.Notion #48 - **[ENHC]** Overflow menu to go to subpage modal.
3. **[MERGED]** Apps.Notion #53 - **[FIX]** Title input visibility issue.
4. **[MERGED]** Apps.Notion #51 - **[ENHC]** Close window after connection success/failure.
5. **[MERGED]** EmbeddedChat #401 - **[FIX]** TOTP Modal and Toast Message Display.
6. **[MERGED]** EmbeddedChat #403 - **[ENHC]** Close modal on overlay/ESC press
7. **[MERGED]** EmbeddedChat #406 - **[FIX]** EC build issue in Windows.
8. **[MERGED]** EmbeddedChat #413 - **[FIX]** Multiple design issues.

9. **[MERGED]** EmbeddedChat #419 - **[CHORE]** Error/Logout for nonexistent channels.
10. **[MERGED]** EmbeddedChat #421 - **[FIX]** App behavior in read-only channels.
11. **[MERGED]** EmbeddedChat #427 - **[CHORE]** Fixed channel fetch issue upon refresh.
12. **[MERGED]** EmbeddedChat #431 - **[FIX]** Message fetch issue in private channels.
13. **[MERGED]** EmbeddedChat #442 - **[ENHC]** Thread menu to access all thread msgs.
14. **[MERGED]** EmbeddedChat #456 - **[FIX]** UI issues in video recorder.
15. **[MERGED]** EmbeddedChat #460 - **[FIX]** Inconsistent back button across sections.
16. **[MERGED]** EmbeddedChat #462 - **[CHORE]** Fixed logout issue.
17. **[MERGED]** EmbeddedChat #472 - **[CHORE]** Fixed linting issue.
18. **[MERGED]** EmbeddedChat #480 - **[ENHC]** Added user-mention menu.
19. **[MERGED]** EmbeddedChat #481 - **[ENHC]** **[EPIC]** UI-Kit Modal Support.
20. **[MERGED]** EmbeddedChat #495 - **[FIX]** Added link preview.
21. **[MERGED]** EmbeddedChat #503 - **[FIX]** Infinite rendering issue
22. **[MERGED]** EmbeddedChat #506 - **[CHORE]** Fixed errors, bugs, and new msg btn.
23. **[MERGED]** EmbeddedChat #510 - **[CHORE]** Fixed message send issue.
24. **[MERGED]** EmbeddedChat #516 - **[FIX]** **[ENHC]** Fallback icon, download/delete option.
25. **[MERGED]** EmbeddedChat #525 - **[FIX]** Fixed sidebar UI inconsistencies and bugs.
26. **[MERGED]** EmbeddedChat#530 - **[FIX]** Static Message Limit.

**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 contributed to some other organizations to learn more about different projects and diverse communities. PRs for those are listed below.

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

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

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

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

## SUITABILITY

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

I am the right person to work on this project as I have prior experience working on the Apps.Notion repository thoroughly and have contributed over 500+ lines of code, demonstrating my in-depth understanding of various event listeners, webhooks, UI-Kit, Slash Commands, etc., which will be required in building our News Aggregation RC app. It's different features, such as support for personalization and choice selection of news providers, categories, etc., heavily rely on UI-Kit Modal which I learnt in depth while contributing to Apps.Notion.

I have also attended the RC-app development workshop every week, which helped me learn to write RC apps from scratch, add settings in RC-apps, and integrate LLMs. These skills will be useful in building features such as making the app globally configurable by admins through settings and integrating LLMs for news summarization. Hence, I have a good understanding of all required functionalities, making me a good fit for this project.

In addition, I have also added support for the UI-Kit modal rendering in Embedded Chat giving me an understanding on how it works internally. Hence, I understand the architecture, flow of the code and required functionalities.

Furthermore, I have made several prototypes, providing proof of concept that I can achieve what I am claiming here.

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

I have been a huge fan of keeping the code bug-free, hence I always love to review fellow contributors' work, helping them out in fixing those bugs, explaining the flow, and why they might be going in the wrong direction and appreciating their work. I also love having my work reviewed. Following the same principle, I try to fix any issue or bug introduced by any PR as quickly as possible, again showing my love for the project as well as my mindset of keeping the code as bug-free as possible. In addition, I am good at communicating with people and have also 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 25-30 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.