Google Summer of Code - 2022
Project Proposal: Rocket.Chat

About Me:

I am **P Aswini Kumar**, a Computer Science and Engineering undergraduate student at the **Indian Institute of Technology (BHU)** pursuing a Bachelor of Technology in my second year. I was introduced to the world of programming and software development in my first year. Since then, I have been very enthusiastic about deep diving into various fields of Computer Science. I love working on web apps and the UI/UX part, making web apps more enjoyable and interactive. I have spent many of my nights hacking on such projects.
I have worked on several projects, and working on them gave me practical experience and knowledge in frontend development frameworks like Vue.js, React.js, and backend frameworks like Django, Django Rest Framework, Express, Nodejs.

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

Being a technology enthusiast even before I entered college, I always enjoyed the time I spent debugging my code and PC. During my first year, I was amazed at how genuinely vast the world of Computer Science is, and there is so much for me to learn, which I genuinely enjoy. My first interaction with open source was Linux and NPM; it was a whole new realization. I was using a project someone else developed, and it's helping me scale up my projects so quickly. The idea of collaborating with so many people working together to build amazing things attracted me to open source.
So I started contributing to open source in this year's Hacktoberfest. I contributed to some organizations. It is delightful to work with so many people way more experienced than me, getting reviews from them, improving my skills and knowledge base, interacting with them, which wouldn't have been possible without open source.
I have similar aspirations from Google Summer of Code. I want to work more in an open-source community where we have several people from different parts of the world working as a team and building projects collaboratively. And what better than **Google Summer of Code**, which will also allow me to hone my skills and acquire new ones under quality mentoring organizations and best mentors' guidance.

Why do I wish to apply in Rocket.Chat in particular ?

I came across Rocket.Chat while browsing through the projects in the GSOC 2021 Archive and found the projects of Rocket.Chat very fascinating. It was also based on the framework in which I was familiar which drew me to take a look at the codebase and start contributing. So far I have loved the community of Rocket.Chat. Everyone is friendly and helpful. Whenever someone faces an issue, the whole community comes together offering their help actively. They give credit to every single contribution at the release of a new patch which tells us the contributors are valued and keep us encouraged. This is why I wish to contribute to Rocket.Chat and try to give back to the community from whom I learnt so much. So Rocket.Chat is the only organization I am applying for in this GSOC.

Working on such a large project, Rocket.Chat, taught me a lot about Teamwork, Problem Solving, the Ability to find and fix bugs, Review code of other teammates, etc. I also learnt how to provide efficient and high-quality code for global projects used by thousands of people across the globe.

Project Title:

Weekly Video Meeting and Archive App - BigBlueButton + Rocket.Chat

A Rocket.Chat app that allows users in a channel to join an existing weekly meeting if it is in process, as well as search for recorded meeting videos and view them.

Abstract:

BigBlueButton is a global teaching platform which is one of the most popular open source virtual classroom and conferencing tools available today. The popularity of this platform calls for the need of an integration in the Rocket.Chat to let the users avail the features of this platform from the Rocket.Chat directly. This project will let the users conduct a weekly meeting and maintain a video archive of its recordings. In this project, I plan to incorporate all the features that would be required and improve upon those to make it one of the most popular Rocket.Chat Apps and make them user friendly and accessible to the users to augment the user experience and further the user-engagement achieved through the app.

Benefits to the community:

Simpler communication is the heart and soul of every communication platform. In addition to messaging, some communities might also require setting up meetings on a daily basis to discuss progress, updates and other important things. It becomes clumsy and not efficient to create meetings and send the link in the channels whenever there is a scheduled meeting. The manual approach also requires a room owner or organizer responsible for scheduling and opening the room while the automated approach doesn't. This asks for a less manual approach making the process more automated and simpler to use by integrating a feature which can create meetings and let the users join the meetings directly from the chat.
The BigBlueButton integration is vastly used by several communities. It is also natively integrated within several LMS Platforms as a synchronous learning tool and also available as a plugin for others. Using the BigBlueButton API, we can integrate BigBlueButton in Rocket.Chat as a Rocket.Chat App. This project is based on solving this issue. My work would benefit the community in that it would enable teams to collaborate more efficiently and with greater flexibility.
This integration is not only an essential feature for any team collaboration platform but also has complete businesses surrounding them. Some open source communication platforms such as Zulip and Mattermost also have a BigBlueButton integration whereas some other famous communication platforms like Slack don't have a BigBlueButton integration so the users or communities who seek such a feature could be drawn to Rocket.Chat widening the Rocket.Chat community and its users. This Weekly Video Meeting and Archive App might become the go to

solution for the communities, organizations, teams and individuals looking for an easier way to organize their meetings and make use of the several features that comes along with it.

Goals/Deliverables:

During the GSoC period I would be focussed on:

1. Defining different app settings asking for necessary information.
2. Adding a feature in the app which notifies the members in the channel every week before the scheduled meeting time reminding about the meeting using the scheduler api.
3. The app should start the room and make the link of the room available.
4. The app should authorize channel members to join.
5. Adding a slash command to get the link of the room which lets you join the meeting.
6. Meeting has ended, so when the recording is ready, it must be archived and uploaded through the video archive handler.
7. The app notifies the users in the channel after the recording has been uploaded to the archiving system.
8. Adding a help slash command which gives the user a brief overview on how to use the app.
9. Provide the documentation and the installation steps in the app guides section of the Rocket.Chat user documentation.
10. Final testing of the features and the improvements made.
11. Publishing the Rocket.Chat Weekly Video Meeting and Archive App - BigBlueButton in the Rocket.Chat Marketplace.

Implementation Details:

App Settings:

In a Rocket.Chat server, the some roles have access to the **administration** where they get several options from handling rooms and users to managing permissions and creating new roles and adding integrations.

Administration -> Apps

These roles also have the ability to manage the apps which include installing the required apps from the vast selections in the Rocket.Chat Marketplace and disabling/uninstalling them if they are not in use.

After an app is installed, the installed app can be seen in the “Installed” tab and clicking on it will display more information about the app like the contact information in the **Contacts** section and some more details about the app in the **Details** section. These details are imported from the app description file, named **app.json** which contains basic information about the app. There are many fields allowed in the app description file which are explained in detail in the app-schema.json.

```
{
   "id": "0dc8312a-f8aa-44cb-bbef-a9e9cf88f243",
   "version": "0.0.1",
   "requiredApiVersion": "^1.4.0",
   "iconFile": "icon.png",
   "author": {
       "name": "Aswini",
       "homepage": "github.com/aswinidev",
       "support": "github.com/aswinidev"
   },
   "name": "bbb.meet",
   "nameSlug": "bbbmeet",
   "classFile": "BbbMeetApp.ts",
   "description": "A weekly meeting and video archiving system"
}
```

Now, depending upon the app it may require some additional information called app settings. App Settings are configuration parameters which are needed for the app to work.

For this Weekly Video Meeting and Archive App, I plan to define the following settings:

- **Meeting Day:**
  This will be the day of the week at which the meeting will be conducted. It may be required to post the reminder in the groups to notify the members about the meeting. This setting will be a Select type of setting which will provide a dropdown menu to select from. The selections will include all the days of the week and their index will be used later to determine which day was selected.

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

export enum Settings {
   MeetingDay = 'Meeting_Day',
}

export const AppSettings : Array<ISetting> = [
   {
       id: Settings.MeetingDay,
       type: SettingType.SELECT,
       packageValue: '',
       required: true,
       public: true,
       i18nLabel: 'Meeting Day',
       i18nDescription: 'The Day when the weekly meetings are
supposed to be conducted.',
       values: [
           {key: 'sunday', i18nLabel: 'Sunday'},
           {key: 'monday', i18nLabel: 'Monday'},
           {key: 'tuesday', i18nLabel: 'Tuesday'},
           {key: 'wednesday', i18nLabel: 'Wednesday'},
           {key: 'thursday', i18nLabel: 'Thursday'},
           {key: 'friday', i18nLabel: 'Friday'},
           {key: 'saturday', i18nLabel: 'Saturday'}
       ]
   },
]
```

- **Meeting Time:**

  This app setting will be the time at which the weekly meetings are supposed to occur. This setting will be a String type of input. The admins will be required to provide the time of the weekly meetings in 24 hours format (For example 5:03 PM would be written as 17:03). The meeting time 24 hours format string can be further split into two parts taking the separator character as ':' one of which will represent hours and other will represent minutes.

  (As in the 1703 example, it would be split into "17" and "03" and a condition would be set which would check

    1. If the Minutes substring is between 00 and 60
    2. If the Hours substring is between 00 and 24

  If it fails to pass any of the above conditions then it would throw an error).

```
  {
       id: Settings.MeetingTime,
       type: SettingType.STRING,
       packageValue: '',
       required: true,
       public: true,
       i18nLabel: 'Meeting Time',
       i18nDescription: 'The Time when the weekly meetings are
supposed to be conducted.',
   }
```

Adding the above expression in the AppSettings const to define another app setting.

Challenges to sort out during the project:

Thinking about global teams, we would also need the time zone along with the time of the meeting.
Maybe add another app setting asking to select a time zone from the dropdown list.

```
   {
       id: Settings.MeetingTimeZone,
       type: SettingType.SELECT,
       packageValue: '',
       required: true,
       public: true,
       i18nLabel: 'Time Zone of the admin',
       i18nDescription: 'The Time Zone with respect to which the
meeting time was set',
       values: [
           // list of all time zones.
       ]
   },
```

Or we could use date and time as the same setting along with the timezone.

- ## Meeting Channel:

  This app setting will require the name of the channel in which the **SlashCommands** will run and the reminder/notifications will be sent. In other words it will be the channel in which the app will function. After a slash command is implemented we can check if the room in which the command is used is the room which is mentioned in the app settings. If it is not that specific room then i will throw an error and may log into the app logs.
  We can extend this app setting to take in **more than one meeting channel** separated by commas and then use the split() method to obtain an array of channels.

```
   {
       id: Settings.MeetingChannel,
       type: SettingType.STRING,
       packageValue: '',
       required: true,
       public: true,
       i18nLabel: 'Meeting Channel',
       i18nDescription: 'The Channel in which notifications are to
be sent and commands to be usable',
   }
```

Adding the above expression in the AppSettings const to define another app setting.

- ## BbbSettings (BigBlueButtonSettings):
  To maintain the simplicity of the code and make it more understandable we can create another settings file which will contain settings related to the BigBlueButton server.
  - ### ➔ BigBlueButton Server URL:
    While installing a BigBlueButton Server you are asked about specifying a hostname (For example bbb.rocket.com) and an email address (for Let's Encrypt) (For example bbbapp@rocket.com). This URL is important to specify because to make an API call to your BigBlueButton server, your application makes HTTPS requests to the BigBlueButton server API endpoint (usually the server's hostname followed by /bigbluebutton/api).
    For example the API calls for a create call would be like:
    http://yourserver.com/bigbluebutton/api/create?[parameters]&checksum=[checksum]
    Similarly for any call like join, isMeetingRunning, end etc. Let's say it to be **"method"** where method can be any of the above mentioned calls. The API calls would be like:
    http://yourserver.com/bigbluebutton/api/**method**?[parameters]&checksum=[checksum]
    (The highlighted "method" can be replaced)
    More details about the parameters of the different call types is well documented here**.**

    This app setting will require that URL so that the Rocket.Chat App can make API calls.

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

export enum Settings {
   BbbServerURL = 'BigBlueButton_Server_URL',
}

export const BbbSettings : Array<ISetting> = [
   {
       id: Settings.BbbServerURL,
       type: SettingType.STRING,
       packageValue: '',
       required: true,
       public: true,
       i18nLabel: 'BigBlueButton Server URL',
       i18nDescription: 'The Server URL of where the meetings
are supposed to be conducted',
   },
]
```

➔ BigBlueButton Shared Secret:

When you first install BigBlueButton on a server, the packaging scripts create a random 32 character sharedSecret. This sharedSecret can also be changed at any time using the command **bbb-conf --setsecret**. If you have the sharedSecret then the BigBlueButton API security model enables 3rd-party applications to make API calls but not allow other people (end users) to make API calls which is why it is important to make sure that this code is only be accessible to the server-side components of your application making it not visible to the end users.

To display this sharedSecret and retrieve your BigBlueButton API parameters (API endpoint and shared secret). You have to run:
`bbb-conf --secret`
A sample return by this command is given here**.**
This sharedSecret is used in calculating the checksum parameter in the API calls. This checksum parameter is actually a SHA-1 sum. The details of this are explained in the later sections.

```
  {
       id: Settings.BbbSharedSecret,
       type: SettingType.STRING,
       packageValue: '',
       required: true,
       public: true,
       i18nLabel: 'BigBlueButton sharedSecret',
       i18nDescription: 'The BigBlueButton API parameter which
could be obtained by running bbb-conf --secret',
   }
```

Adding the above expression in the AppSettings const to define another app setting.

➔ BigBlueButton moderatorPW and attendeePW:

There are two types of users in BigBlueButton:

(a) **Viewer:** They have the ability to chat, send/receive audio and video, respond to polls, display an emoji (such as their raised hand), and participate in breakout sessions. But they don’t possess any ability to affect other members in the room.

(b) **Moderator:** There can be multiple moderators in a session. Along with all the abilities that the viewer possesses, they also have the ability to mute/unmute other viewers, lock down viewers (such as restrict them from using private chat), make any user (including themselves) the current *presenter*, and start breakout rooms.

Presenters are made by a moderator of the room and there can be only one presenter at a time. The presenter is an additional layer of privileges that allows any user to upload slides, use the whiteboard to annotate any slide, enable/disable multi-user whiteboard, start a poll, Share a YouTube, vimeo, Peertube, or Canvas Studio video and share their screen.

The create API call can have several parameters as mentioned in the documentation here. Out of these parameters some are required and some are optional. While calling the **create** API call, we have to specify these parameters in the query string. Two of the parameters are the **moderatorPW** and **attendeePW** which are the password that can be later provided while calling the **join** API call as its password parameter to indicate the user will join as a viewer or as a moderator. If no **attendeePW** is provided, then the **create** call will return a randomly generated **attendeePW** password for the meeting. Similarly for the **moderatorPW**, a randomly generated **attendeePW** password for the meeting will be returned if no password is provided during the **create** call.

So to specify the roles of the person while they join the meeting, it is required to make an app setting which takes in the **attendeePW** and **moderatorPW**.

```
  {
      id: Settings.BbbModeratorPW,
      type: SettingType.STRING,
      packageValue: '',
      required: true,
      public: true,
      i18nLabel: 'BigBlueButton moderatorPW',
      i18nDescription: 'The password to let the person join
as a moderator of the meeting',
   },
   {
      id: Settings.BbbAttendeePW,
      type: SettingType.STRING,
      packageValue: '',
      required: true,
      public: true,
      i18nLabel: 'BigBlueButton attendeePW',
      i18nDescription: 'The password to let the person join as an attendee of the meeting',
   },
```

Challenges to sort out during the project:

Thinking about the user experience, attendeePW and moderatorPW shouldn't be set manually by the user this way. **It should be transparent and handled by the app.** The api passwords are only used to define roles and generally they aren't visible for the users. As the initial stage of the project in the rough code snippets of this proposal I have used the moderatorPW and attendeePW provided in the app settings to make api calls. I plan to overcome this challenge by introducing some other way to validate a user to join as a moderator.
It is mentioned in the BigBlueButton api documentation that if we don't provide the moderator and attendee passwords then it will randomly generate a password during the api call which is later returned in the XML response. We can use this randomly generated password directly. Then we can introduce a separate user's password, which can be used by the integration for some kind of validation such as allowing a user to join as moderator. Because usually the api passwords and user passwords are separated.

➔ BigBlueButton Meeting Name:

While making the create API call we need to specify a name for the meeting through the **name** parameter. This app setting refers to that name and will take an entry in form of a string.

```
{
        id: Settings.BbbMeetingName,
        type: SettingType.STRING,
        packageValue: '',
        required: true,
        public: true,
        i18nLabel: 'BigBlueButton Meeting Name',
        i18nDescription: 'The name of the meeting',
    },
```

➔ BigBlueButton Meeting ID:

This app setting value will be used in specifying the **meetingID** parameter while making API calls. It is required to be specified because it is a **required** parameter. Meeting IDs should only contain upper/lower ASCII letters, numbers, dashes, or underscores.

We can extend it further later by **removing this app setting** and generating a **GUID** value as this all but guarantees that different meetings will not have the same meetingID.

```
{
       id: Settings.BbbMeetingId,
       type: SettingType.STRING,
       packageValue: '',
       required: true,
       public: true,
       i18nLabel: 'BigBlueButton Meeting Id',
       i18nDescription: 'The Id of the meeting',
   },
```

After adding all of these settings, both of the settings files will look like the following:

bbbsettings file:

- The source code implementation could be found here.

appsettings file:

- The source code implementation could be found here.

Now to add these settings which can be configured by the administrator, we have to use **provideSetting** in the **extendConfiguration** function**.**

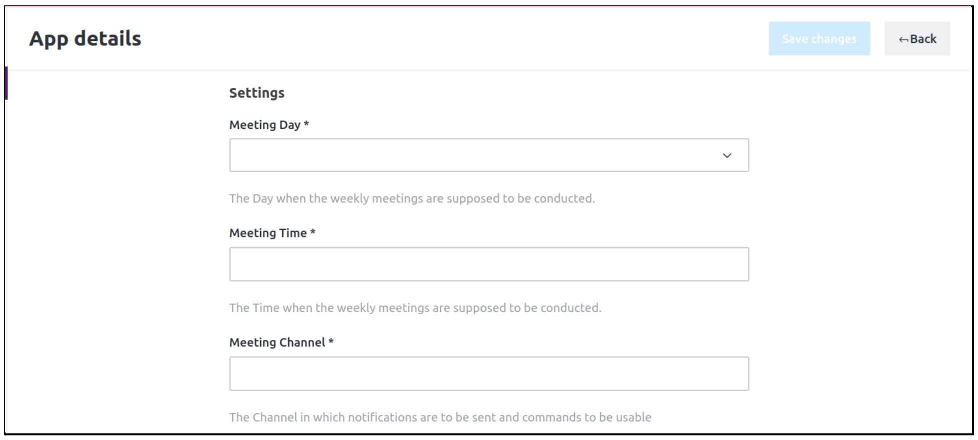
Adding them individually,

```
// Individual app settings
       configuration.settings.provideSetting({
           id: 'meeting-day',
           type: SettingType.STRING,
           packageValue: '',
           required: true,
           public: true,
           i18nLabel: 'Meeting Day',
           i18nDescription: 'The Day when the weekly meetings are supposed
to be conducted.',
       })
```

But doing this will not make the code much readable and make the BbbMeetApp.ts file unnecessarily cluttered which can be resolved by making another settings.ts file as I have made above. Then we import all of these settings in the BbbMeetApp.ts file and map them to add the app settings one by one.

```
import {
    IAppAccessors,
    IConfigurationExtend,
    IEnvironmentRead,
    ILogger,
} from '@rocket.chat/apps-engine/definition/accessors';
import { App } from '@rocket.chat/apps-engine/definition/App';
import { IAppInfo } from '@rocket.chat/apps-engine/definition/metadata';
import { ISetting } from '@rocket.chat/apps-engine/definition/settings';
import { AppSettings } from './settings/appsettings';
import { BbbSettings } from './settings/bbbsettings';

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

    protected async extendConfiguration(configuration:
IConfigurationExtend): Promise<void> {
        await Promise.all(AppSettings.map((setting) =>
configuration.settings.provideSetting(setting)));
        await Promise.all(BbbSettings.map((setting) =>
configuration.settings.provideSetting(setting)));
    }
}
```

Now the app settings will look like:



Settings

Meeting Day *

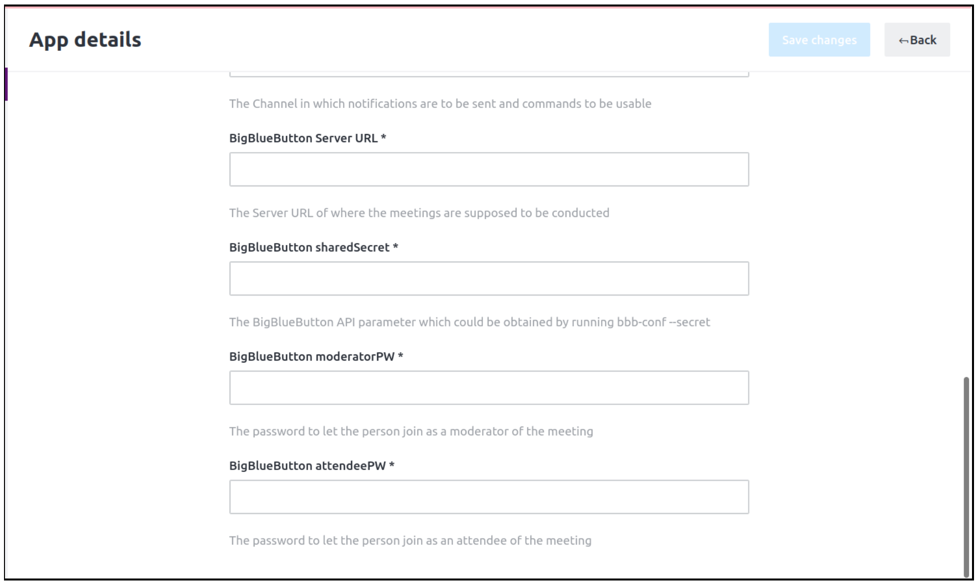
The Day when the weekly meetings are supposed to be conducted.

Meeting Time *

The Time when the weekly meetings are supposed to be conducted.

Meeting Channel *

The Channel in which notifications are to be sent and commands to be usable


App details

Save changes

←Back

The Channel in which notifications are to be sent and commands to be usable

BigBlueButton Server URL *

The Server URL of where the meetings are supposed to be conducted

BigBlueButton sharedSecret *

The BigBlueButton API parameter which could be obtained by running bbb-conf --secret

BigBlueButton moderatorPW *

The password to let the person join as a moderator of the meeting

BigBlueButton attendeePW *

The password to let the person join as an attendee of the meeting

Further Challenges to work on during the project:

As the initial stages of the project, I have used an approach which limits a Rocket.Chat instance to have a single weekly meeting configured. I plan to incorporate a more flexible approach in which teams can have a weekly meeting configured per channel. To implement this we can configure meeting day, meeting time and channel in the slash command to set up the weekly meeting.

A simple welcome message when the app is installed/enabled:

I plan to incorporate a simple welcome message which will be triggered when the app is enabled and let the members of the channel know that this app has been enabled.
To implement this we can use the **onInstall** Method which is called when the App is installed or it's being uninstalled from Rocket.Chat and it is called only once. Therefore it is not called when the App is updated or when the app is getting disabled manually.

```
public async onInstall(context: IAppInstallationContext, read: IRead,
http: IHttp, persistence: IPersistence, modify: IModify): Promise<void> {
       //logging in the app logs
       this.getLogger().log('Hello Everyone! I am a Rocket.Chat App which
will take care about your weekly meetings');

       //sending in the general channel
       const creator: IModifyCreator = modify.getCreator();
       const room = await read.getRoomReader().getByName('general');
       if (room === undefined) {
           this.getLogger().log(`room general doesn't exist`)
```

```
        } else {
            const sender: IUser = (await read.getUserReader().getAppUser())
as IUser;
            const messageTemplate: IMessage = {
                text: 'Hello Everyone! I am a Rocket.Chat App which will
take care about your weekly meetings',
                sender,
                room
            }
            const messageBuilder: IMessageBuilder =
creator.startMessage(messageTemplate)

            await creator.finish(messageBuilder)

        }
    }
```

The above code snippet is the code to send to the “general” channel because general is a default channel that is present in a Rocket.Chat server and because on installing the app settings might not have been filled so we can’t know about the meeting channel just yet. We can also extend this to send a message whenever the app is enabled letting people know that the app is enabled now. For that we have to use another method called **onEnable** which gets executed whenever the app is enabled.

Talking about the above snippet, the first part is to log in the app logs which can be viewed by the administrator by clicking on the view logs option in the app information page. To log into the app logs we have to use **this.getLogger()** and do normal style logging. It is such that due to the limitations in the NodeJS’s vm package for which Rocket.Chat had to create a custom logger class.

In the second part, to send a message in the app engine we need a creator object, namely an instance of **IModifyCreator** and we can get this creator object by using the **IModify.getCreator()** method. Then we can start the message using a resource builder/message builder for which we can use the **IModifyCreator.startMessage** method. And this method requires a message template which is an object representation of a message which at the very least should contain some text, a sender and a room (where the message is going to be sent). I found the room by the **getByName** method which lets you search a room by its name.

Meeting Notification/Reminder:

I think user experience also plays a major role in planning the features of this app which is why I plan to incorporate a feature in this app which notifies/reminds you about the meeting at the scheduled day and time. This will be a notifier/reminder which notifies the members of the “meeting channel” about the meeting. This reminder message will consist of a text and a join button along with it clicking on which will let them join the meeting.

This reminder feature can be achieved by the use of the scheduler API and a recurring job can be created which sets out a reminder in the “meeting channel” (From the app settings) every week.

To use the Scheduler API, the first thing to do is to register the processors or the job functions which are configured in the **extendConfiguration** method. Then the jobs are registered using the **configuration.scheduler.registerProcessors()** method which registers the job processor. Each job processor must have an **id** which is how each job is identified and they must be different. These **id**s are required while triggering the registered processor as a job. A job processor must also have a processor, that is, the functions to be run as jobs. These jobs are finally triggered using **modify.getScheduler().scheduleRecurring(task)** (For Recurring schedule which makes the job run in an interval else scheduleOnce(task) could be used which makes the job run only once (when)). The **getScheduler()** method gets the accessor for creating scheduled jobs. After the jobs are triggered and (in this case) “Recurring”, the functions that we defined as processors will run after the specified intervals. These processor functions are an async function and can receive arguments (say jobContext) which are passed during scheduling.

Now, to cancel a certain job we have to **cancelJob()** which cancels/stops a running job if you pass the job id as a parameter. We can also cancel all the running jobs from the app with the help of **cancelAllJobs()** method.

Before triggering the job we have to register the processor in the extendConfiguration method as mentioned above.

```
       //Register processors
       await configuration.scheduler.registerProcessors([
           {
               id: jobId.Reminder,
               processor: async (jobContext: IJobContext, read: IRead,
modify: IModify, http: IHttp, persis: IPersistence): Promise<void> => {
                   const block = modify.getCreator().getBlockBuilder()

                   this.getLogger().log('Reached Processor')
                   //Creating the block template
                   block.addSectionBlock({
                       text: {
                           type: TextObjectType.PLAINTEXT,
                           text: 'The scheduled weekly meeting is about to
start! Join by clicking on the "Join" button below'
                       }
                   })
                   block.addActionsBlock({
                       elements: [
                           block.newButtonElement({
                               actionId: "joinbutton",
                               text: {
                                   type: TextObjectType.PLAINTEXT,
                                   text: 'Join'
                               },
                           })
                       ]
                   })
                   //Find the meeting channel from the App settings
                   const setting =
read.getEnvironmentReader().getSettings()
                   const roomname = await
setting.getValueById('Meeting_Channel')
                   const room = await
read.getRoomReader().getByName(roomname);
                   const sender: IUser = (await
read.getUserReader().getAppUser()) as IUser

                   if (room === undefined){
                       console.log(`Room ${roomname} doesn't exist`)
                   } else {
                       await modify.getCreator().finish(

modify.getCreator().startMessage().setSender(sender).addBlocks(block.getBl
ocks()).setRoom(room)
                       )
                   }


               }
           }
       ]);
```

In the processor function I wrote the function that will be implemented when the required interval passes. As I mentioned above, I plan to send a text message in the “Meeting Channel” from the App settings which will act as a reminder and this reminder message will also contain a button named “Join” click on which will let the people join the meeting. In order to send this message I used the getBlockBuilder() method which gets a new instance of a BlockBuilder. Then using this method we create a block template by adding section blocks using the addSectionBlock() method and to display the button we use the addActionsBlock() method and assign a specific **action id** to this button so that the interactions can be recorded and handle the click event later on. After this we try to find out the desired channel to send this message which could be obtained by **read.getEnvironmentReader().getSettings()** which gets an instance of the App's settings reader and then we can get the values set for any of the setting by using the **getvalueById()** method which takes in the the id of the setting which we want to access. These Ids are the Ids that we specified earlier while we defined individual App settings. After we get all this information we can simply create a message with **modify.getCreator().startMessage()** and add other information such as sender, room and the blocks using other methods.

How do we trigger this registered job processor?

The scheduler will start after the required settings are provided in the App settings and the settings are updated. For this we have to use the onSettingUpdated method which is called whenever a setting which belongs to this App has been updated by an external system and not this App itself and the newly updated settings are passed. When the settings are updated this method will be called so we need to implement the triggering of the registered processor in this function only.

```
public async onSettingUpdated(setting: ISetting, configurationModify:
IConfigurationModify, read: IRead, http: IHttp): Promise<void> {
        // cancel all the existing jobs
        await configurationModify.scheduler.cancelJob(jobId.Reminder)

        // Find the meeting channel from the App settings
        const set = read.getEnvironmentReader().getSettings()
        const room = await set.getValueById('Meeting_Channel')

        // Calculating the Cron Expression
        const time = await set.getValueById('Meeting_Time')
        var [hrs,mins] = time.split(":")
        if(hrs.length !== mins.length){
            this.getLogger().log('Invalid Meeting Time Setting was found');
            return undefined
        }

        hrs = parseInt(hrs as string, 10)
        mins = parseInt(mins as string, 10)

        if(time.length !== 5 || hrs.isNaN || mins.isNaN || hrs>23 ||
mins>60 || hrs<0 || mins<0){
            this.getLogger().log('Invalid Meeting Time Setting was found');
            return undefined
        }

        const day = await set.getValueById('Meeting_Day')
        const dayind = ['sunday', 'monday', 'tuesday', 'wednesday',
'thursday', 'friday', 'saturday'].indexOf(day)

        if(dayind === -1){
            this.getLogger().log('No Meeting Day setting was found. Make
sure you have selected a day before proceeding.');
            return
        }

        if(room !== undefined){
            await configurationModify.scheduler.scheduleRecurring({
                id: jobId.Reminder,
                interval: `${mins} ${hrs} * * ${dayind}`,
                skipImmediate: true,
                data: {}
            })
            this.getLogger().log(`${mins} ${hrs} * * ${dayind}`)
        }
    }
```

When the App settings are updated we have to create a new job process and cancel the previous reminder notification process because this time the day/time of the meeting might have been changed. So, first we cancel all the existing processes. This can be done in two ways. One is to use **cancelJob()** which cancels a single job whose id would have been passed and the other is too use **cancelAllJobs()** which stops all the currently running jobs but using **cancelAllJobs()** would not be a wise choice because if in the future more features would be required to be incorporated in this app then it could cause problems.

Then we try to fetch the app settings and also check if they are valid. For the “Meeting Time” setting we check if the administrator has provided the time string as it was requested and also check if it is a valid 24 hours format or not (mentioned in the meeting time section in detail).

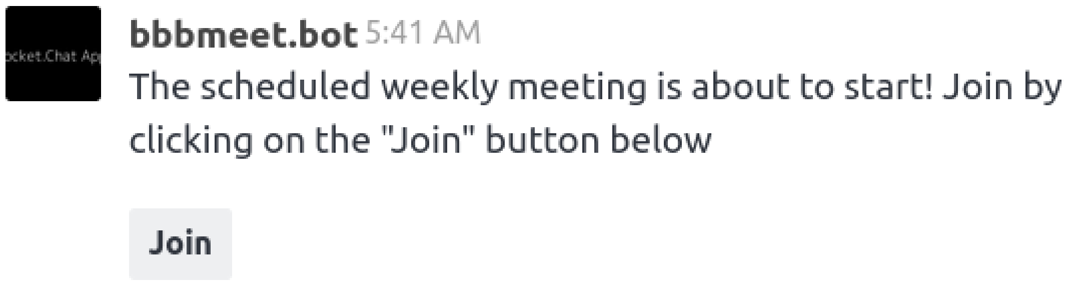
Then we also check the day setting and if it is one of the selections we have provided. Lastly check if it is a valid room. Failing to meet any of the conditions will result in an error.

How to send the reminder message on that particular day and time?

The issue that arises here is to figure out how to start the scheduling from that specific day of the week. If we register a processor and start the scheduler to work in an interval of 7 days then it will start the instant when the app is activated and say the app was installed on monday then it will notify on the next monday and not on the day the meet is supposed to occur which might be different from the day the app was installed.
To solve this issue so that we could continue to notify the users weekly without worrying about the day the notifications were activated is to use the cron expression. This lets us schedule the triggering day and time of the job.

After all the checkings, we can try to find out the **cron expression** which would be required to specify on which day and at what particular time the reminder message will be sent. The **"*"** stands for "any value" which means it can be any value and the days are determined by the value 0-6 where the **0** stands for **sunday**, **1** stands for **monday** and so on until **6** for **saturday**. **7** also stands for **sunday** but it is not standard and may not work with every cron. The time (minutes and hours) are written in 24 hours format. The cron expression calculated in the above code snippet is `` `${mins} ${hrs} * * ${day}` `` where **"mins"** stands for the minutes and **"hrs"** stands for the hours and lastly day for the day of the week.

The reminder message would look something like this:


bbbmeet.bot 5:41 AM

The scheduled weekly meeting is about to start! Join by clicking on the "Join" button below

Challenges to sort out during the project:

Currently in my prototype the reminder message is sent exactly at the time the meeting is scheduled. We could work on it further to **notify the users a few minutes before the weekly meeting**.

Slash Commands:

A Slashcommand is a way to call the app installed in Rocket.Chat and let the app interact within a room in Rocket.Chat. An App can have multiple slash commands. A SlashCommand is an instance of some class type that implements the **ISlashCommand** interface.

To implement the ISlashCommand interface, we need to define the required properties of the interface which are **command**, **i18nDescription**, **i18nParamsExample**, **providesPreview** and an **executor** method.

- The **command** property is the actual command which the user has to implement in order to use this slash command.
- The **i18nDescription** property is a brief description about the slash command.
- The **i18nParamsExample** property is an example of the parameters or an i18n string.
- The **providesPreview** lets the users know that this command does provide preview results if the boolean value is set to *true*. If it is set to true then it is required to define two other methods which are called before the **executor** method.
- The **executor** method is the method which gets called when the user implements the slash command.

It is recommended to create a separate directory to store all the slash commands related files.

We need some slash commands in order to let the users interact with this App. I have planned to incorporate the following slash commands.

- ## Join Slash Command:

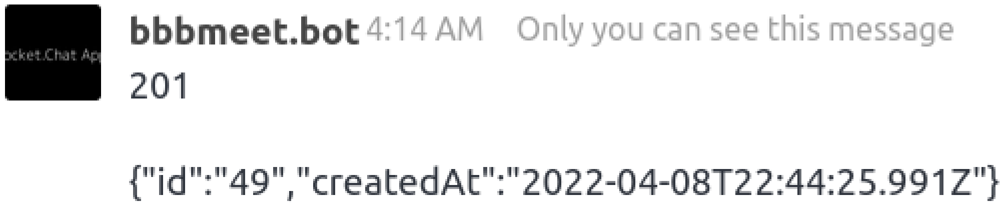
  This command will let the users join the weekly meetings directly from the Rocket.Chat channel by simply writing a command in the chat. As discussed in the settings section, BigBlueButton has two types of users: a **Moderator** and an **Attendee.**
  Before thinking about the join, I tried to make some simple **POST** requests to reqres.in and learn more about making post requests in the Apps Engine.

```
import { IRead, IModify, IHttp, IPersistence, IModifyCreator,
IMessageBuilder, IHttpRequest  } from
"@rocket.chat/apps-engine/definition/accessors";
import { IMessage } from
"@rocket.chat/apps-engine/definition/messages";
import { IRoom } from "@rocket.chat/apps-engine/definition/rooms";
import {ISlashCommand, SlashCommandContext} from
"@rocket.chat/apps-engine/definition/slashcommands";
import { BlockBuilder, TextObjectType } from
"@rocket.chat/apps-engine/definition/uikit";
import { IUser } from "@rocket.chat/apps-engine/definition/users";
import { sha1 } from "../SHA1/sha1";

export class JoinCommand implements ISlashCommand {
   public command = "joinmeet";
   public i18nDescription = "Lets you join weekly meetings";
   public providesPreview = false;
   public i18nParamsExample = "";
   private sharedSecret : string;

   public async executor(
       context: SlashCommandContext,
       read: IRead,
       modify: IModify,
       http: IHttp,
       persis: IPersistence
   ): Promise<void> {

       let options: IHttpRequest = {
           data : JSON.stringify({
               name: "John Doe",
               Job: "Content Writer"
           })
       }
       const sample = await
http.post('https://reqres.in/api/users',options)

       const sender : IUser = (await
read.getUserReader().getAppUser()) as IUser
       const room : IRoom = context.getRoom()

       const blockBuilder: BlockBuilder =
modify.getCreator().getBlockBuilder()
       blockBuilder.addSectionBlock({
           text: {
               type: TextObjectType.PLAINTEXT,
               text: `${sample.statusCode}`
           }
       })
       blockBuilder.addSectionBlock({
           text: {
               type: TextObjectType.PLAINTEXT,
               text: `${JSON.stringify(sample.data)}`
           }
       })

       await modify.getNotifier().notifyUser(context.getSender(), {
           sender,
           room,
           blocks: blockBuilder.getBlocks()
       })
   }
}
```

We also have to register this command in the **extendConfiguration()** method (lets us to extend the configurations of our app) of the main BbbMeetApp.ts file using:

```
await configuration.slashCommands.provideSlashCommand(new
JoinCommand())
```

This works flawlessly and we can see the results in the chat room.

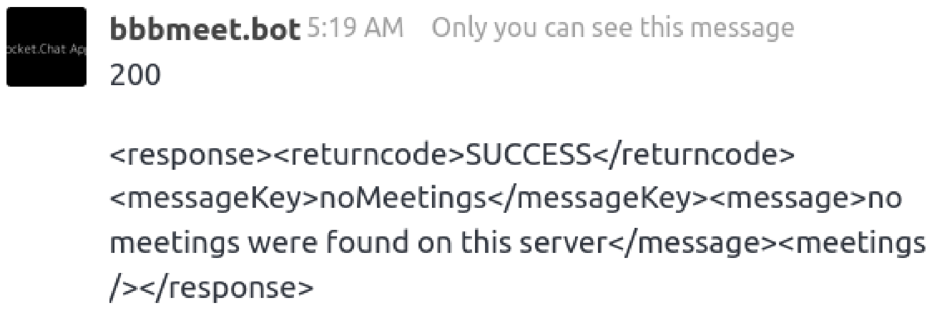


bbbmeet.bot 4:14 AM Only you can see this message
201

{"id":"49","createdAt":"2022-04-08T22:44:25.991Z"}

Then I tried to make some calls to the test server at [API Mate]() and get the response data.

```
public async executor(
        context: SlashCommandContext,
        read: IRead,
        modify: IModify,
        http: IHttp,
        persis: IPersistence
    ): Promise<void> {
        //make the call
        const response = await
http.get('https://test-install.blindsidenetworks.com/bigbluebutton/a
pi/getMeetings?checksum=d23fef405937517be465ffccae12d5c1103a5e00')

        const sender : IUser = (await
read.getUserReader().getAppUser()) as IUser
        const room : IRoom = context.getRoom()

        const xml = response.content
        const blockBuilder: BlockBuilder =
modify.getCreator().getBlockBuilder()
        blockBuilder.addSectionBlock({
            text: {
                type: TextObjectType.PLAINTEXT,
                text: `${response.statusCode}`
            }
        })

        blockBuilder.addSectionBlock({
            text: {
                type: TextObjectType.PLAINTEXT,
                text: `${xml}`
            }
        })

        await modify.getNotifier().notifyUser(context.getSender(), {
            sender,
            room,
            blocks: blockBuilder.getBlocks()
        })
    }
```


bbbmeet.bot 5:19 AM Only you can see this message

```
200

<response><returncode>SUCCESS</returncode>
<messageKey>noMeetings</messageKey><message>no
meetings were found on this server</message><meetings
/></response>
```

The BigBlueButton gives responses in the form of an xml file instead of json which makes it a little less accessible and we have to handle the xml file to find out the responses.

Now let’s work on the “joinmeet” command. First create a join command that lets us join the meeting directly as a moderator.

```
import { IRead, IModify, IHttp, IPersistence, IModifyCreator,
IMessageBuilder, IHttpRequest  } from
"@rocket.chat/apps-engine/definition/accessors";
import { IMessage } from
"@rocket.chat/apps-engine/definition/messages";
import { IRoom } from "@rocket.chat/apps-engine/definition/rooms";
import {ISlashCommand, SlashCommandContext} from
"@rocket.chat/apps-engine/definition/slashcommands";
import { BlockBuilder, TextObjectType } from
"@rocket.chat/apps-engine/definition/uikit";
import { IUser } from "@rocket.chat/apps-engine/definition/users";
import { sha1 } from "../SHA1/sha1";

export class JoinCommand implements ISlashCommand {
   public command = "joinmeet";
   public i18nDescription = "Lets you join weekly meetings";
   public providesPreview = false;
   public i18nParamsExample = "";
   private sharedSecret : string;

   public async executor(
       context: SlashCommandContext,
       read: IRead,
       modify: IModify,
       http: IHttp,
       persis: IPersistence
   ): Promise<void> {

       // Collect all the required settings
       const set = read.getEnvironmentReader().getSettings()
       const bbbserver = await
set.getValueById('BigBlueButton_Server_URL')
       this.sharedSecret = await
set.getValueById('BigBlueButton_sharedSecret')
       const moderatorPW = await
set.getValueById('BigBlueButton_moderatorPW')
       const attendeePW = await
set.getValueById('BigBlueButton_attendeePW')
       const meetingId = await
set.getValueById('BigBlueButton_Meeting_Id')
```

```
        const meetingName = await
set.getValueById('BigBlueButton_Meeting_Name')

        // Create the query string
        const query =
`name=${meetingName}&meetingID=${meetingId}&attendeePW=${attendeePW}
&moderatorPW=${moderatorPW}&record=true`
        const sha1string = "create" + query + `${this.sharedSecret}`
        // Calculate sha1 value
        const sha = sha1(sha1string)
        //Generate the final url
        const url = bbbserver + "/bigbluebutton/api/create?" + query
+ `&checksum=${sha}`

        //make the create call
        const response = await http.get(url)

        if(response.statusCode === 200){
            //Create the join query string
            const username = context.getSender().name
            const joinquery =
`fullName=${username}&meetingID=${meetingId}&password=${rolepw}&redi
rect=true`
            const joinsha1string = "join" + joinquery +
`${this.sharedSecret}`
            const joinsha1 = sha1(joinsha1string)
            const joinurl = bbbserver + "/bigbluebutton/api/join?" +
joinquery + `&checksum=${joinsha1}`

            const sender : IUser = (await
read.getUserReader().getAppUser()) as IUser
            const room : IRoom = context.getRoom()

            // The Message is sent with a join url
            const blockBuilder: BlockBuilder =
modify.getCreator().getBlockBuilder()
            blockBuilder.addSectionBlock({
                text: {
                    type: TextObjectType.MARKDOWN,
                    text: `Click on the join button below to join the
meeting:\n`
                }
            })
            blockBuilder.addActionsBlock({
                elements: [
                    blockBuilder.newButtonElement({
                        text: {
                            type: TextObjectType.PLAINTEXT,
                            text: 'Join'
                        },
                        url: joinurl
                    })
                ]
            })

            await
modify.getNotifier().notifyUser(context.getSender(), {
                sender,
                room,
                blocks: blockBuilder.getBlocks()
            })
        }
    }
}
```

In the above code snippet, we first fetch all the app settings which would be required in creating the final API call URL by **read.getEnvironmentReader().getSettings()**. I also defined a **private** string variable named **sharedSecret** because this is the code which lets us make api calls from 3rd party applications and if a person attains it then he/she will be able to make api calls himself and make it a security issue. But by making it **private** we prevent the end users from attaining this secret code.

I made a **create** call before the **join** call because **CREATE** is used to instantiate the room whereas **JOIN** is the URL call that redirects the user to the virtual room. We have to create the meeting before joining since the meeting needs to be instantiated first. And according to the BigBlueButton API docs, the create call is idempotent which means we can call it **multiple times** with the same parameters without side effects. There will not be any duplicate meetings created. Because of this we can always use the **create** call before the **join** call every time which will make sure the call always exists before joining it. The first **create** call actually creates the meeting and subsequent calls to create simply return **SUCCESS**.

For the **create** call, I created a query string which contained all the required parameters according to the create call in the BigBlueButton API docs. It requires a meeting name, meeting id, attendee password and moderator password which are recommended, and record parameter. Setting 'record=true' instructs the BigBlueButton server to record the media and events in the session for later playback.

Then I created a SHA-1 string and calculated the SHA-1 sum of that string which is later used to pass on as a **checksum** parameter which is a necessary parameter for a valid create api call. To find out the SHA-1 sum I borrowed the algorithm from here.

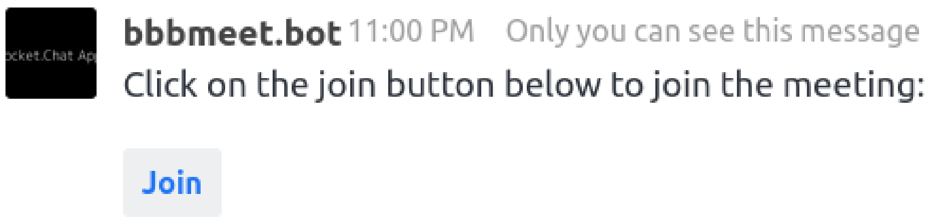
Then I finally made the create call using:

```
const response = await http.get(url)
```

Where http is an instance of the IHttp interface and it returns a IHttpResponse interface. Then the statusCode was passed through a conditional statement which checks if the response is positive or not. If it is positive, then another query string was created which contained all the required parameters according to the **join** call of the BigBlueButton API docs. It requires a fullName which is the name of the user, meetingID and a password or role to determine if the person is an attendee or a moderator. The display name of the user would be the same as the Rocket.Chat name which could be fetched using **context.getSender().name**.

Finally the URL was generated after calculating the **SHA-1 sum** and appending it. This URL would let the users join the BigBlueButton meeting. Then I created a join button and attached the join url i generated to this button. Now, when the user clicks on this join button, he is redirected to the meeting and lets him/her join the meeting.

The message would look something like this:



Click on the join button below to join the meeting:

The functioning could be seen through this demo video: Link

But the issue in the above code snippet is that it only lets the user join as a moderator. We have to include some feature which would let the users join as a moderator as well as an attendee.
To join as a moderator we have to pass the password of the moderator as the **password** parameter in the query string of the API call. Similarly, to join as an attendee we have to pass the password of the attendee as the **password** parameter in the query string of the API call. We have already defined App settings for both of them (i.e.,**moderatorPW** and **attendeePW**).

To implement this, we have to read the arguments followed by the slash command. The user has to give arguments along with the slash commands and then we have to read these arguments.
The template for the the “joinmeet” command is as follows:

`/joinmeet {moderator_password}`

- ➔ **To join as a Moderator:**
  To join as a Moderator, the user has to write **`/joinmeet {moderator_password}`** where **`moderator_password`** is the moderator password which is checked if it matches the password of the role specified according to the fetched settings from the app settings. If it doesn’t then it throws an error and sends a message which is only visible to the user.

- ➔ **To join as an Attendee:**
  To join as an Attendee, the user just has to write **`/joinmeet`** which directly lets the user join the meeting and doesn't ask for any password.

The rough code snippet implementing the above feature is as follows:

```
        let rolepw = attendeePW

        const [password]: Array<string> = context.getArguments()

        if(password !== undefined && password !== moderatorPW){
            const sender : IUser = (await
read.getUserReader().getAppUser()) as IUser
            const room : IRoom = context.getRoom()

            const blockBuilder: BlockBuilder =
modify.getCreator().getBlockBuilder()
            blockBuilder.addSectionBlock({
                text: {
                    type: TextObjectType.PLAINTEXT,
                    text: "Wrong Password!"
                }
            })

            await
modify.getNotifier().notifyUser(context.getSender(), {
                sender,
                room,
                blocks: blockBuilder.getBlocks()
            })
            return;
        } else if (password !== undefined){
            rolepw = moderatorPW
        }
```

Adding the above code will let the command read the arguments and give the response accordingly. If there is an argument then this conditional statement will check if it is a valid moderator password, else it will send out an error saying that it is a wrong password. If the argument is present and correct then we change the value of the **rolepw** to **moderatorPW** and then this password is passed in the join query string letting the user join as a moderator.

Further, we also need to check if the channel in which the command was implemented is the channel which is mentioned in the app settings. If not, then it will throw an error. We can do that with the help of following code snippet:

```
const set = read.getEnvironmentReader().getSettings()
        const meetroomname = await
set.getValueById('Meeting_Channel')
        const meetroom = await
read.getRoomReader().getByName(meetroomname);
        const commandroom = context.getRoom()
        if(meetroom === undefined){
            console.log(`Room ${meetroomname} doesn't exist`)
        } else if(meetroom.id !== commandroom.id) {
            const sender : IUser = (await
read.getUserReader().getAppUser()) as IUser
            const room : IRoom = context.getRoom()

            const blockBuilder: BlockBuilder =
modify.getCreator().getBlockBuilder()
            blockBuilder.addSectionBlock({
                text: {
                    type: TextObjectType.PLAINTEXT,
                    text: "This Command is not allowed in this
channel"
                }
            })

            await
modify.getNotifier().notifyUser(context.getSender(), {
                sender,
                room,
                blocks: blockBuilder.getBlocks()
            })
            return;
        }
```

In the above code, first we fetched all the settings from the App settings using the **`read.getEnvironmentReader().getSettings().`** Then we passed it through a check which checks if the setting mentioned was the name of an actual room. If it is not an actual room then the result of **`await read.getRoomReader().getByName(meetroomname);`** will be undefined and we throw an error saying it is not a valid channel and another check to find out if the current room in which the command was sent is sake as the channel mentioned in the App settings. If it is not then we send a text saying "This command is not allowed in this channel"

- ## List Recording Slash Command:

  It will be a slash command which will query for the list of all the recent recordings. We can find out which videos have been uploaded to the archiving system and list them for the user to view.

  We can also list out the recordings related to the meetingID specified in the app settings present in the BigBlueButton as well. To list all of the recordings, we have to use the **getRecordings** API call which takes in a meetingID parameter in the query string. It retrieves the recordings that are available for playback for a given meetingID. After we get all of the recordings that are available, we can send a message in the meeting channel notifying the person who ran this command about the list of recordings along with the recording id/playback url. The response from the above API call will be a XML doc which we have to handle separately before extracting the required data from it. More information about handling XML docs is written in sections below.

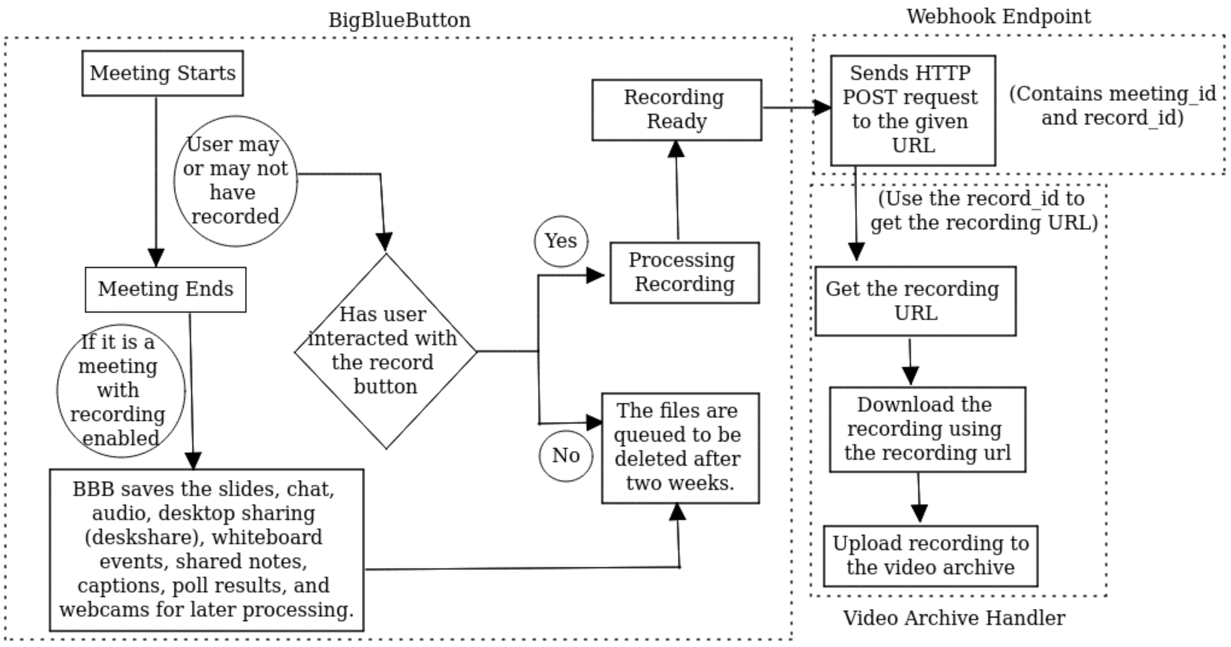
- **Help Command:**
  This command will be a help command which when implemented will send a message to the person who implemented the command which will contain all the information about how the app works and which app commands to use and how. This will be completed at the end of the project specifying all the features our app possesses.
  An initial rough implementation of this command is as follows:

```
import { IRead, IModify, IHttp, IPersistence, IModifyCreator,
IMessageBuilder  } from
"@rocket.chat/apps-engine/definition/accessors";
import { IMessage } from
"@rocket.chat/apps-engine/definition/messages";
import { IRoom } from "@rocket.chat/apps-engine/definition/rooms";
import {ISlashCommand, SlashCommandContext} from
"@rocket.chat/apps-engine/definition/slashcommands";
import { BlockBuilder, TextObjectType } from
"@rocket.chat/apps-engine/definition/uikit";
import { IUser } from "@rocket.chat/apps-engine/definition/users";

export class HelpCommand implements ISlashCommand {
   public command = "helpmeet";
   public i18nDescription = "Tells you how the app / app commands
work";
   public providesPreview = false;
   public i18nParamsExample = "";

   public async executor(
       context: SlashCommandContext,
       read: IRead,
       modify: IModify,
       http: IHttp,
       persis: IPersistence
   ): Promise<void> {
       const sender : IUser = (await
read.getUserReader().getAppUser()) as IUser
       const room : IRoom = context.getRoom()

       // A help command which will be completed at the end of the
project
```

```
        const blockBuilder: BlockBuilder =
modify.getCreator().getBlockBuilder()
        blockBuilder.addSectionBlock({
            text: {
                type: TextObjectType.PLAINTEXT,
                text: 'You have implemented the help command'
            }
        })

        await modify.getNotifier().notifyUser(context.getSender(), {
            sender,
            room,
            blocks: blockBuilder.getBlocks()
        })
    }
}
```

More information will be added later before the App is published in the Marketplace.

Video Archiving System:


BigBlueButton
Meeting Starts
User may or may not have recorded
Meeting Ends
If it is a meeting with recording enabled
BBB saves the slides, chat, audio, desktop sharing (deskshare), whiteboard events, shared notes, captions, poll results, and webcams for later processing.
Has user interacted with the record button
Yes
No
Processing Recording
Recording Ready
The files are queued to be deleted after two weeks.
Webhook Endpoint
Sends HTTP POST request to the given URL
(Contains meeting_id and record_id)
(Use the record_id to get the recording URL)
Get the recording URL
Download the recording using the recording url
Upload recording to the video archive
Video Archive Handler

If the recording is enabled for the meeting (record=true parameter is appended in the create api call), then after the meeting ends BigBlueButton checks whether any moderator has pressed on

the record button or not. If he/she didn't click on the record button throughout the meeting then no recording is saved, else the record is processed.
BigBlueButton has a feature where we can ask the BigBlueButton server to make a callback to our application when the recording for a meeting is ready for viewing. To specify the callback to BigBlueButton, we would have to pass a URL using the meta-parameter **meta_bbb-recording-ready-url** on the **create** command. After the recording is ready, it checks whether any **meta_bbb-recording-ready-url** is set, if it is set then it makes a HTTP POST request to this url.

If we want the callback url to be "https://open.rocket.chat/api/apps/public/appcode/recordingstatus", then we have to append a parameter in the create API call query string as "&meta_bbb-recording-ready-url=https%3A%2F%2Fopen.rocket.chat%2Fapi%2Fapps%2Fpublic%2Fappcode%2Frecordingstatus".

When the URL receives the POST request and the payload it will trigger the class that extended this webhook endpoint. The webhook endpoints are defined in Rocket.Chat app by extending the ApiEndpoint class.
The implementation of a simple webhook by Rocket.Chat App Engine is as follows:

```
import { HttpStatusCode, IHttp, IMessageBuilder, IModify, IPersistence,
IRead } from "@rocket.chat/apps-engine/definition/accessors";
import { ApiEndpoint, IApiEndpointInfo, IApiRequest, IApiResponse } from
"@rocket.chat/apps-engine/definition/api";
import { IMessage } from "@rocket.chat/apps-engine/definition/messages";
import { IUser } from "@rocket.chat/apps-engine/definition/users";

export class WebhookEndpoint extends ApiEndpoint{
   public path = 'recordingstatus';

   public async post(
       request: IApiRequest,
       endpoint: IApiEndpointInfo,
       read: IRead,
       modify: IModify,
       http: IHttp,
       persis: IPersistence,
   ): Promise<IApiResponse> {
       const set = read.getEnvironmentReader().getSettings()
       const room = await set.getValueById('Meeting_Channel')
       const sender = (await read.getUserReader().getAppUser()) as IUser
```

```
        const creator  = modify.getCreator()
        const messageTemplate: IMessage = {
            text: 'Recording is ready',
            sender,
            room
        }
        const messageBuilder: IMessageBuilder =
creator.startMessage(messageTemplate)
        await creator.finish(messageBuilder)

        return {
            status: HttpStatusCode.OK
        }
    }
}
```

First we specify the path (here “recordingstatus”) which specifies the last part of the api URL. The post method is one of the methods which are called whenever the publicly accessible url for this App is called. In the above code snippet, whenever the payload is received we are sending a message in the meeting channel notifying everyone in it that the recording is ready and returns a response to the source that sent the payload. The **request** parameter contains the contents of the payload and the headers.

The next step would consist of registering this webhook endpoint in the **extendConfiguration** method.

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

The BigBlueButton POST request body would be in “application/x-www-form-urlencoded” format and the payload will consist of two JSON keys one of which is the meeting id (meeting_id) of the meeting the recording is of and the other is the record id (record_id) which corresponds to the id returned in the getRecordings api.

We got the record_id of the recording from the previous step which we can further use to find the link to the recording of the presentation. We can use the **getRecordings** api call and get the recording available for playback with this particular record id by appending the **recordID** parameter in the API call. When the **getRecordings** api call is used, it returns each recording with two different formats: one is podcast which consists of only the audio and other is the presentation which consists of both audio and video. It gives us the URL that redirects us directly to the playback of the recording.

We need an archiving system to store the recordings. The BigBlueButton does store the recordings for later playback but it is not exactly an archiving system. Now, the url to the presentation is available. Using this mp4 URL we can download the playback video and then later upload it to the video archive (in the case of avideo, upload the video to the channel). The credentials required by the app to upload the video will be specified as an app setting. where the admin could provide the link of the archiving site may be avideo or something else and some credentials related to it which could be then fetched and then used to upload the recorded meeting.

The important thing that should be kept in mind while designing the video archiving system is that **avideo** is **not a hard dependency**, so the design of the app or the **architecture** of the code should make sure that plugging in other archive systems (for example youtube) is easy enough and could be done by making minimal to no changes in the core code.

But we can't support all the archiving systems in a single app so we have to design it in a way that supporting new archiving systems will be easy. The plan is to make it simple enough that supporting a new archiving system would be a simple class that implements an interface like **class YouTubeUploader implements VideoArchiveHandler** (in TypeScript) and then use **VideoArchiveHandler** as the handler type internally. This would require no need to change core code and if someone would like to support another archiving system, they can make an addition of a single file or a class.

The join button of the reminder messages:

We can attach the generated join url to this button. For this we can make a separate function which would generate a join url. In the reminder processor, we can also add a **create** API call which will create the meeting so that a meeting has already been instantiated before the user clicks on the join button and the join url comes into play. OR we could just include it in the function which I discussed earlier. This function can be later reused in the joinmeet slash command as well.
The display names in the meeting will be their **Rocket.Chat names**.

Handling XML responses:

The BigBlueButton server returns an XML response to all API calls, not a JSON which makes it complicated to interact with the response data. We have to handle these XML responses. To do this, we can convert the XML responses to JSON so that we can interact with the response data with greater ease. To convert the XML to JSON we can use several libraries available but we have to make sure that there are no add-on/native dependencies in it. Or else we can also use one of the libraries that handles the BBB API. One of the libraries suggested by my mentors is bigbluebutton-js but we can explore more about this and try out different libraries and after taking advice from my mentors we can finalize our plan of action on how to handle the XML responses.

Future Work:

- **Attendance:**
  Sometimes it might be required to record the attendance in the meeting to keep a track of people who are joining the meetings on a regular basis. So we can incorporate a feature through which we could generate an attendance list of the last meeting. To incorporate this feature we can use the **getMeetingInfo** API call which takes in the meetingID as a parameter and returns all of a meeting's information, including the list of attendees as well as start and end times. We can make a slash command (say "attendance") which will trigger this API call and generate an attendance list.

Contact info and timezone(s):

Mail: aswini123p@gmail.com , p.aswini.kumar.cse20@itbhu.ac.in
GitHub: https://github.com/aswinidev
LinkedIn: Aswini
Rocket.Chat Username: aswini.p
Timezone: IST (GMT+5.5)
Institute : Indian Institute of Technology. Varanasi (BHU)

Time commitment:

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

My end semester examinations are in the last week of April. Therefore, I would be able to dedicate around 35-40 hours per week to this project.

I would also inform about any sudden engagements (if any) and can easily cover up by working more in preceding and succeeding weeks

Other summer obligations:

What jobs, summer classes, and other obligations might you need to work around?

I don't have any prior commitments and don't have any other academic occupations throughout the GSoC period.

Communication channels:

How often, and through which channel(s), do you plan on communicating with your mentor?

I plan to schedule a time regularly every week to give status updates to my mentors after discussing with my mentors. The primary mode of communication between me and my mentors would be through 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 messaging from 11am IST to 1am IST both weekdays and weekends.

Why do I think I'm well suited for this project?

I have been contributing to Rocket.Chat for the past several months and have a good understanding of the architecture and the codebase. I love the Rocket.Chat community which is most welcoming and helpful whenever you need some help. Rocket.Chat Apps is a feature which always fascinated me, where we can create our own custom apps and use them or even publish them in the Apps Marketplace to reach the vast Rocket.Chat users and community.

Hence, I am really excited to be working on a Rocket.Chat App by adding new features and tweaking the user experience making it simplistic and accessible.

I have been active in the community as well for the past several months. In the past month I have discussed several interesting ideas with my mentors and thought about how to improve the app features by being in regular contact with them through the Rocket.Chat channels and provided them with the status update every few days and sought guidance from my mentors whenever I had a question or a doubt. I plan to be in regular contact with my mentors in the GSOC period as well and provide them with regular updates regarding my project to create an app that could become one of the most popular apps.

I have also experimented with the App Engine while discussing the ideas related to this project and I have made myself familiar with the functions, interfaces and features Rocket.Chat App Engine provides. I went through the Apps-Engine API reference / RC App Engine Typescript Definitions which contains schema and definitions of **all the interfaces and methods** that would be required while building apps. Interfaces are a typescript feature that can enforce a class that meets a particular contract. The definitions of the interfaces lets us know about the properties that could be provided and which of them are required properties and which of them are optional. It is like a dictionary for all of the App Engine.

I am a quick learner and I am willing to dedicate my time for the successful completion of this project. I like working on things which are different from the conventional stuff and Rocket.Chat App-Engine provides me with exactly what I seek. A framework that allows you to create Rocket.Chat Apps and provides the APIs for Rocket.Chat Apps to interact with the host system.

Related Work Done:

I have made myself familiar with the Rocket.Chat App Engine and have started to work on a prototype in which I have managed to build a part of the deliverables:

- Added several app settings asking for several necessary information. The source code implementation could be found here.
- Added a slash command to get the link of the room which lets you join the meeting. The app starts the room and makes the link of the room available. The source code implementation could be found here.
- Added a help slash command which provides a brief overview on how to use the Rocket.Chat app. The source code implementation could be found here.
- Added a reminder feature which notifies the members in the channel about the meeting every week. The source code implementation could be found here.
- Added a sample webhook endpoint to receive payload. The source code implementation could be found here.
- **Extra:** Added a feature which would let users with specific roles to stop the weekly reminder notifications. The source code implementation could be found here.
- **Extra:** Added a feature which would send out a message in the channel on being installed. That the app is installed. The source code implementation could be found here.

The link to my repository containing the prototype for the Weekly Video Meeting and Archive App is here.

Relevant Experiences:

I have been consistently contributing to Rocket.Chat since December 2021. My contributions include - fixing bugs, adding features, improving the documentation and raising issues.

The Links to all my Pull Requests that I made over the last couple of months can be seen below:

- Merged PRs Link (Total - 7)
- Open PRs Link (Total - 9)
- Issues Link (Total - 10)

I have been a top contributor according to the GSOC Contribution Leaderboard which is a leaderboard for the contributors to track their position based on the PRs, commits, and issues they've completed across the repositories of interest across your Github organization.
The Link to the GSOC Contribution Leaderboard is here**.**

Apart from this I have been a part of various development teams during my journey as a software developer for over 1.5 years. I have primarily worked as a Web Developer but have also interacted with several other developers from various fields.

Club Of Programers (COPS):

I am a member of Club of Programmers (COPS), IIT(BHU). COPS is a group of students in our college where students share their knowledge about different technologies in order to learn from each other. It also organizes various hackathons and coding events to enrich programming culture. I joined the club in my first year and learned a lot from my seniors by working and contributing to various projects of the club. It is through the seniors of this club that I came to know about open source contributions as well as about Google Summer of Code. I have worked under several projects taken by the Club which helped me gain experience in some frameworks, including several projects for aiding the student community and fun side projects and competitive projects for hackathons. All projects can be found at the Github Organization COPS IIT (BHU).

Projects:

- **SDG Site (Link)**
  - ➔ Along with a team of 20+ developers developed a performance site using NuxtJS and further improved it using the Lighthouse.
  - ➔ Optimized it further by Caching, Layout shift Optimizations, Promise Optimizations and async code Optimizations.

- **ShopHaul (Link)**

- under Prof Dr. Amrita Chaturvedi, Assistant Professor CSE (IIT BHU)

➔ Developed a Django based E-Commerce web application for the local sellers to increase their business.

➔ Designed and implemented features such as user authentication, managing the products, see the revenue earned and checking competition.

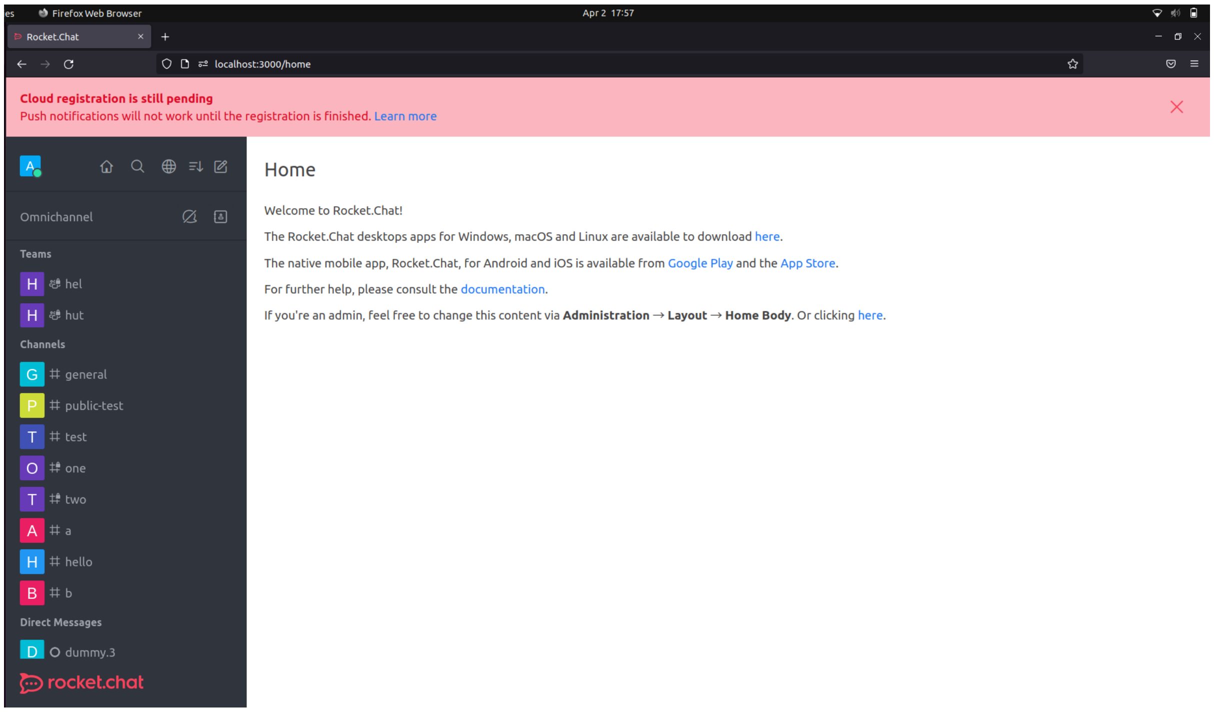
- **Orientation Bot (Link)**
    ➔ Worked on a telegram bot made using telegrafJS (Modern Telegram Bot API framework for Node.js) for the orientation of the freshers.

- **Nakama Bot (Link)**
    ➔ My very first project which made me interested in development.

    ➔ A Python Telegram chatbot which can do various things like interacting with you, giving you memes, talking about the weather and helping you to surf through the different sections of youtube.

Essential Prerequisites:

- **A running Rocket.Chat server in your system:**
  The Screenshot attached below is a running Rocket.Chat 4.7.0-develop server running flawlessly in my Ubuntu 21.10 system.



Cloud registration is still pending
Push notifications will not work until the registration is finished. Learn more

Home

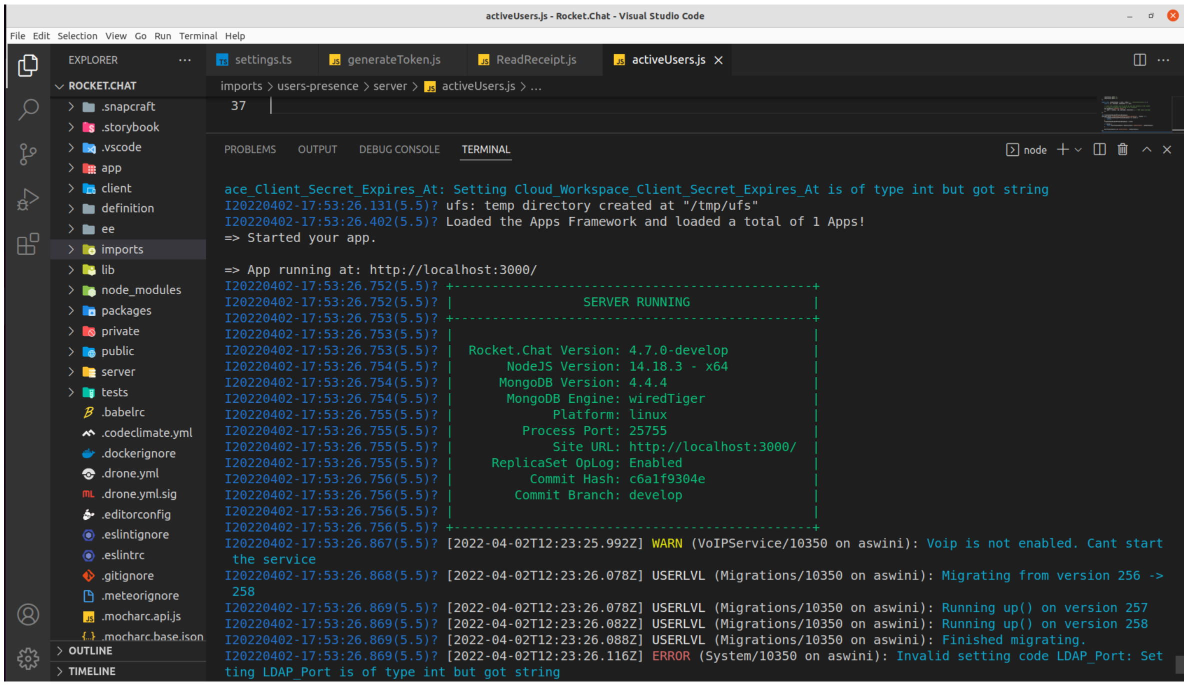
Welcome to Rocket.Chat!

The Rocket.Chat desktops apps for Windows, macOS and Linux are available to download here.

The native mobile app, Rocket.Chat, for Android and iOS is available from Google Play and the App Store.

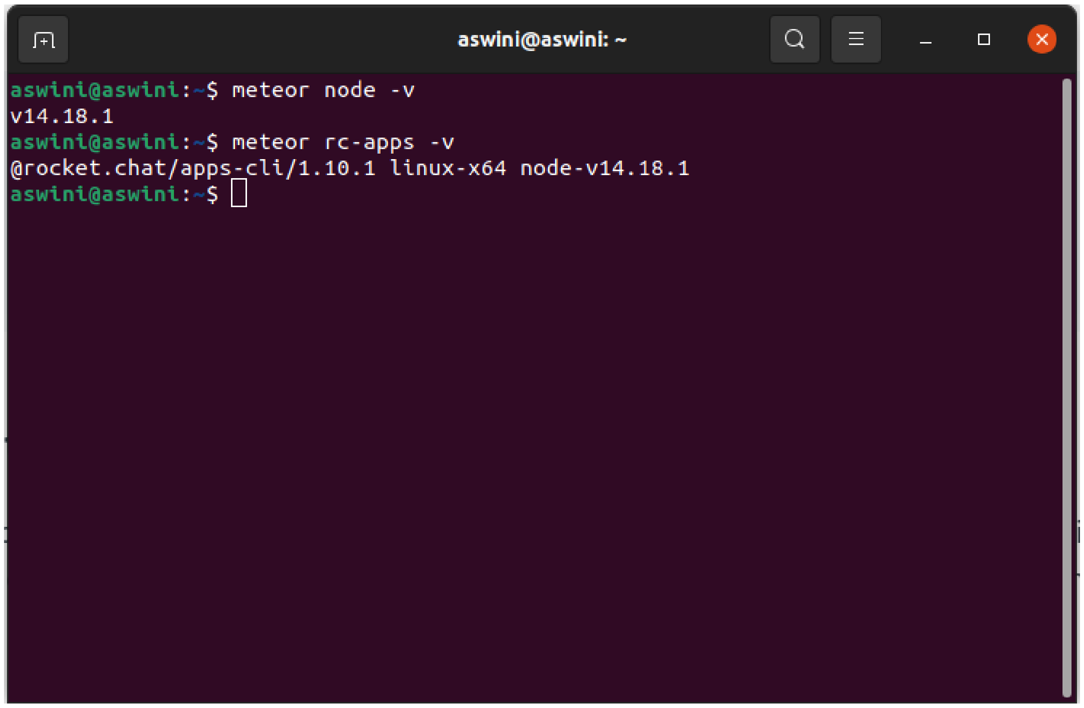
For further help, please consult the documentation.

If you're an admin, feel free to change this content via **Administration** → **Layout** → **Home Body**. Or clicking here.



```
ace_Client_Secret_Expires_At: Setting Cloud_Workspace_Client_Secret_Expires_At is of type int but got string
I20220402-17:53:26.131(5.5)? ufs: temp directory created at "/tmp/ufs"
I20220402-17:53:26.402(5.5)? Loaded the Apps Framework and loaded a total of 1 Apps!
=> Started your app.

=> App running at: http://localhost:3000/
I20220402-17:53:26.752(5.5)? +----------------------------------------------+
I20220402-17:53:26.752(5.5)? |                SERVER RUNNING                |
I20220402-17:53:26.753(5.5)? +----------------------------------------------+
I20220402-17:53:26.753(5.5)? |                                              |
I20220402-17:53:26.753(5.5)? |  Rocket.Chat Version: 4.7.0-develop          |
I20220402-17:53:26.754(5.5)? |       NodeJS Version: 14.18.3 - x64          |
I20220402-17:53:26.754(5.5)? |      MongoDB Version: 4.4.4                  |
I20220402-17:53:26.754(5.5)? |       MongoDB Engine: wiredTiger             |
I20220402-17:53:26.755(5.5)? |             Platform: linux                  |
I20220402-17:53:26.755(5.5)? |         Process Port: 25755                  |
I20220402-17:53:26.755(5.5)? |             Site URL: http://localhost:3000/ |
I20220402-17:53:26.755(5.5)? |     ReplicaSet OpLog: Enabled                |
I20220402-17:53:26.756(5.5)? |          Commit Hash: c6a1f9304e             |
I20220402-17:53:26.756(5.5)? |        Commit Branch: develop                |
I20220402-17:53:26.756(5.5)? |                                              |
I20220402-17:53:26.756(5.5)? +----------------------------------------------+
I20220402-17:53:26.867(5.5)? [2022-04-02T12:23:25.992Z] WARN (VoIPService/10350 on aswini): Voip is not enabled. Cant start
 the service
I20220402-17:53:26.868(5.5)? [2022-04-02T12:23:26.078Z] USERLVL (Migrations/10350 on aswini): Migrating from version 256 ->
 258
I20220402-17:53:26.869(5.5)? [2022-04-02T12:23:26.078Z] USERLVL (Migrations/10350 on aswini): Running up() on version 257
I20220402-17:53:26.869(5.5)? [2022-04-02T12:23:26.082Z] USERLVL (Migrations/10350 on aswini): Running up() on version 258
I20220402-17:53:26.869(5.5)? [2022-04-02T12:23:26.088Z] USERLVL (Migrations/10350 on aswini): Finished migrating.
I20220402-17:53:26.869(5.5)? [2022-04-02T12:23:26.116Z] ERROR (System/10350 on aswini): Invalid setting code LDAP_Port: Set
ting LDAP_Port is of type int but got string
```

- **Rocket.Chat App Engine installed and working:**
  It was mentioned in the developer docs of Rocket.Chat that we should not use a system which already has node js installed and therefore start with a clean system. That's why I installed the app engine cli using meteor so it won't cause any errors in running the Rocket.Chat development environment. But it still works flawlessly.



```
aswini@aswini:~$ meteor node -v
v14.18.1
aswini@aswini:~$ meteor rc-apps -v
@rocket.chat/apps-cli/1.10.1 linux-x64 node-v14.18.1
aswini@aswini:~$ 
```

- **PC specifications:**
  16 GB RAM
  Octa Core Ryzen 9
  1 TB SSD

Detailed Work Plan:

Application Review Period (Upto May 20):

During the application review period, I plan to gain a greater familiarity with the Rocket.Chat Apps-Engine along with the modules provided by it by further experimenting with the interfaces and methods. I also intend to discuss more about the project ideas and discuss the implementation of the features and how to improve upon them resulting in better user experience and make it user friendly. I will also try to make more contributions to Rocket.Chat and help the community.

Community Bonding Period (May 20 - June 12):

In the community bonding period, I plan to interact with the community more. I would try to get to know my mentors, read documentation and take a look at the codes of other Rocket.Chat apps and get to know more about the features and limitations of the App Engine. In this period, I also plan to set up communications with my mentors and decide upon when and how often I will give them updates regarding my work. I will focus on developing a roadmap and finalizing a plan of action with my mentors for the upcoming coding period.

[Coding Period Starts]

Week 1-2 (June 13 - June 19 & June 20 - June 26):

In this period, I plan to add different app settings that might be useful and work on the notifier that reminds the members in the channel about the meeting using the scheduler API. I plan to register the processor and then trigger the reminder job when the settings will be updated which could be done using the onSettingUpdated method which gets called once the app settings are updated. The interval will be set using the cron expression. A detailed explanation about the implementation is described in the implementation section of this document and the prototype could be found in the related work done section of this document.

Week 3 (June 27 - July 3):

This week, I plan to implement the slash command which lets the users join the meeting. It will also include integrating BigBlueButton with the Rocket.Chat app using the BigBlueButton API. When the join command will be called, the Rocket.Chat app will fetch necessary information from the app settings and then create the url to make the API call and send out a join button clicking on which will redirect the user to this url. Before creating the join url the app will create the meeting to make sure that the room is instantiated before joining the meeting. According to the BigBlueButton API documentation, creating the same meeting multiple times will not throw any errors or cause any duplicacy of the meetings making it convenient to create the meeting every time before joining.

Week 4 (July 4 - July 10): Video Archive Research

This week I will spend my time trying to learn more about the video archiving system and plan the design such that if a user wouldn’t want to use avideo specifically as the video archiving system will not have to go through too much trouble rather he/she can just make a simple change without changing the code code much to incorporate any other archiving system. I will discuss with my mentors and design the videoArchiveHandler interface.

Week 5 (July 11 - July 17): Video Archive Implementation.

This week I plan to start working on the implementation of the Video Archiving system taking into account my research from week 4.

Week 6 (July 18 - July 24):

This week is just the week before the first evaluations which is why I plan to keep this week as a buffer week and work on the feedback received from the mentors in evaluation and improvise on that. I also plan to complete any incomplete work from the above weeks due to any delay in the timeline before the first evaluations.

Week 7 (July 25 - July 31): Video Archive Implementation (continued)

This week I plan to continue with the implementation of the archiving system from week 5 and also create a slash command which will let the users list all/some of the recordings. This period I also plan to thoroughly test all of the features that have been implemented until this week.

Week 8 (August 1 - August 7):

This week I will focus on implementing the recording ready notifier which will be a webhook endpoint which upon receiving the payload will notify the users in the meeting channel that the recording is done processing and it is ready for playback.

Week 9 (August 8 - August 14): List Recordings.

This week I plan to work on the implementation of the slash command which will let the users list all/some of the recordings.

Week 10 (August 15 - August 21):

I plan to use Week 10 as a buffer week for any unforeseen circumstances like any disruption in the timeline or appearance of any persistent bug at any point. I also plan to thoroughly test all of the features that have been implemented thus far in order to eliminate any potential errors. In addition to this, I would also try to implement any recommendations or improvements I receive from my mentors and the Rocket.Chat community during Week 10.

Week 11 (August 21 - August 27):

This week, I plan to complete the help command and create documentation, tutorials, and quick-start guidelines for anyone new to the Weekly Video Meeting App during Week 11. In addition, I would make a video presentation of how to use the Weekly Video Meeting App's many features. By the end of this week, I'll have a project summary and a presentation for the Rocket.Chat community on the work I accomplished during my GSoC time and hopefully publish a new Rocket.Chat App in the Rocket.Chat marketplace.