**Chat Support (CRM)**

**Summary**

* This CRM lets users to chat 1 on1 with agent regarding issues they are facing in SOFTWARE,HR,ER,IT etc.
* Users can chat with agents regarding **Software**,**Human Resource**,**Information Technology**,**ER**.
* Login is on the basis of **EMS** don’t have register just login if not registered then users will be registered.
* If no agents are available to chat then users will be asked to fill email,mobile,and query. And will be contacted by agent later.
* One agent can receive up to 10 chats at a time , and both agents have option to exit chat and also exit with email (chat will be sent to given email and also to ems)

**Requirement**

* React (18)
* Node JS
* Mongo DB
* Express JS
* NPM
* NGINX (Web Server)
* Socket IO

**API DETAILS**

* Production API s.
* EMS API s,
* Demo API s

**CRM (FLOW)**

**Login -**

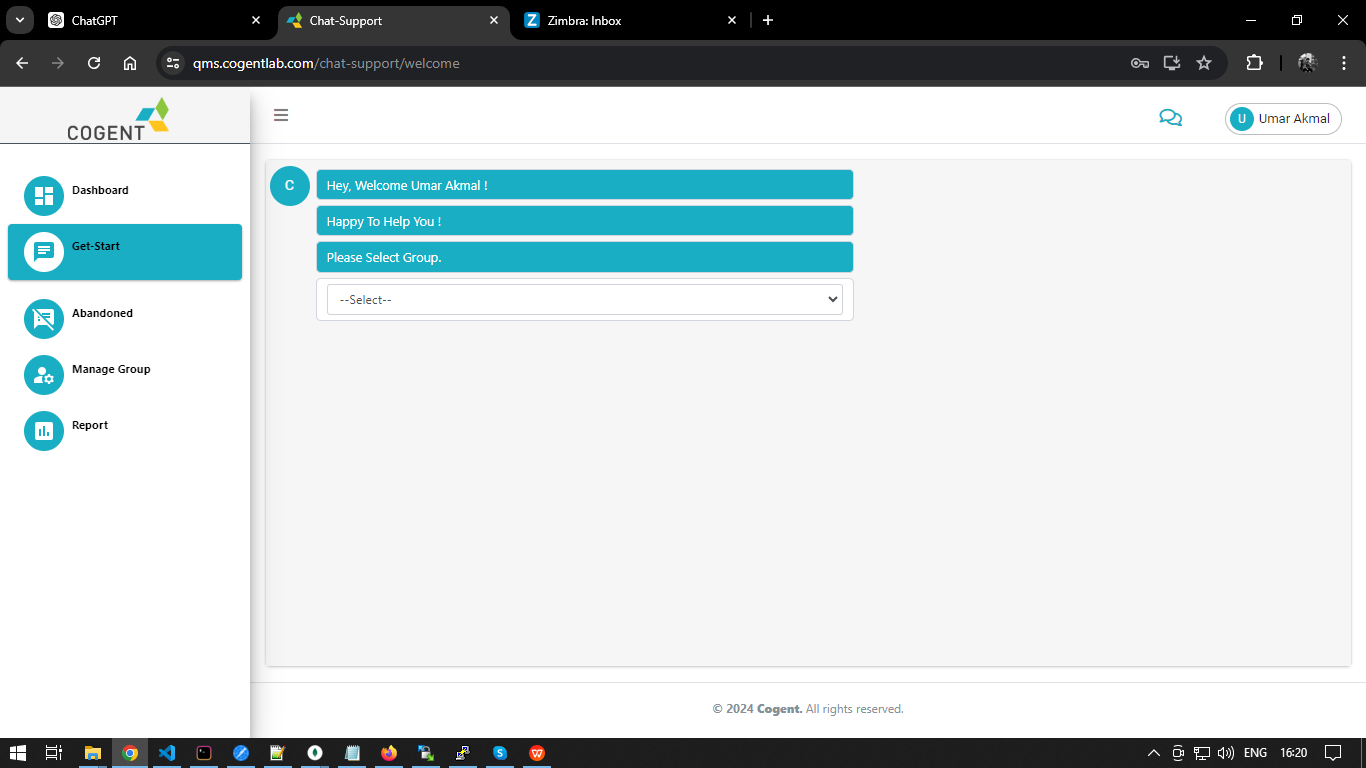
* Enter your **EMS Employee ID** and **Password** to get a successful login .
* You will get redirected to the **Get-Start** page if you are customer , and **Dashboard** page if agent or admin.
* Based on the type of Users(Customer,Agent,admin) can access different pages such as customer will have only get-start page access and agent will have get-start,abandon,report,dashboard,and all the chats that has been assigned where as admin will have all the above mentioned page with manage group.

**Customer’s can access pages -**

* Customer’s can access only Get-Start page.

#### Get-Start -

* Here users can chat with agents regarding concerns they are facing for example(Software, Human Resource,Information Technology,ER).
* Select a group .
* Will go into chat with available agent.
* If agent is not available then customer will be asked to give mobile,email and query.
* If customer logins then subscription token is generated from this page and push notification has been used which uses these tokens.



* Validations

1. Group selection is mandatory to chat.
2. If no agents are available then abandon window will show which is mandatory.

* API s

1- /api/chat/get-active-chat

This API Checks if customer has an active chat or not.

2- /subscribe

This API registers the subscription for push notification.

3- /api/chat/get-all-groups

This API gets all the groups.

4- /api/create-abandon

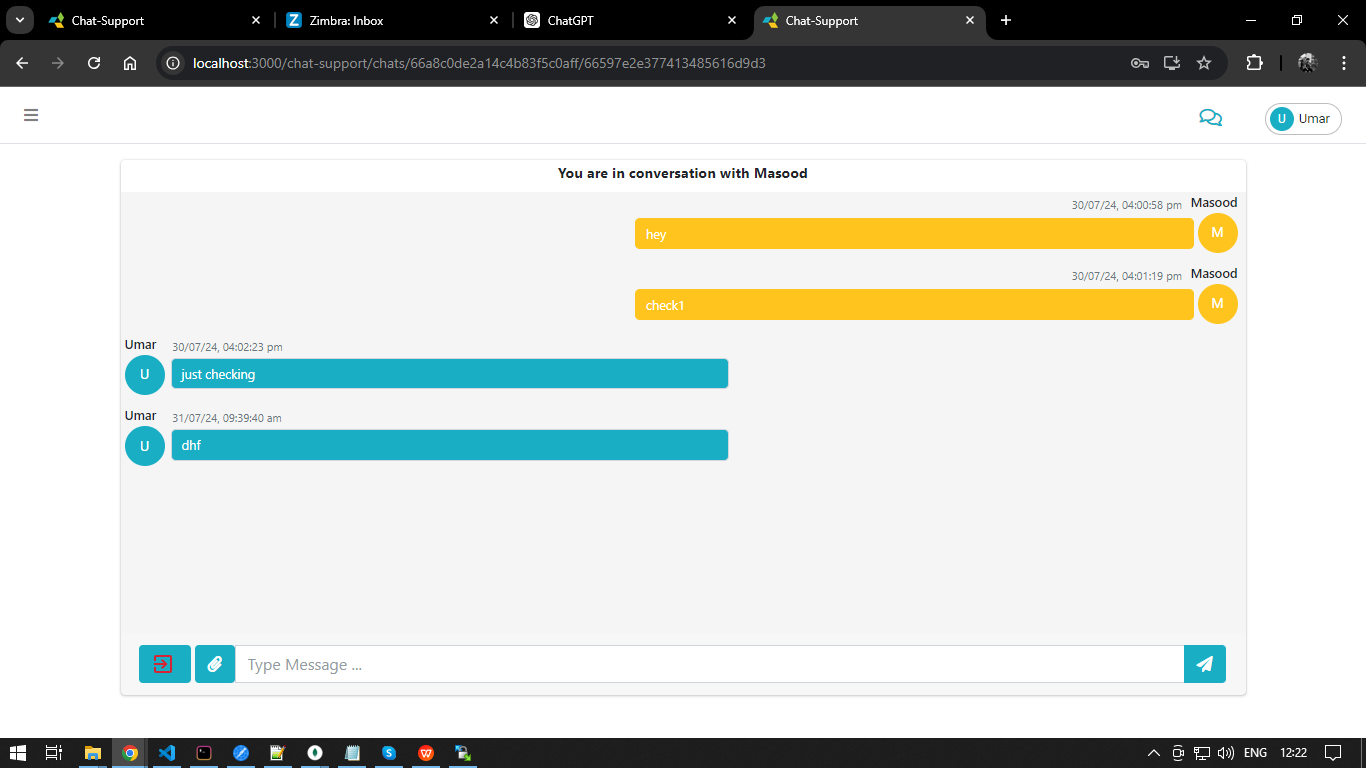
This API submits customers email,contact,query when agents are unavailable.

1. /api/chat

This API adds customers with agents.

Chat with agent -

* Customers can chat with the agents here.
* Here after chat has been completed customers have two options either exit chat or exit and send, where chat will me mailed to the given email.
* Customer can also share image,PDF , email and messages.



* Validations

1. If exit and send then email id is mandatory.

* API s

1- /api/message/${ChatID}/${userId}

This API gets total of 10 messages at a time between two users.

2- /api/get-reciever

This API gets a details of receivers for sender.

3- /api/chat/get-chat-status

This API checks status of chat if it is active or has been closed.

1. /api/message

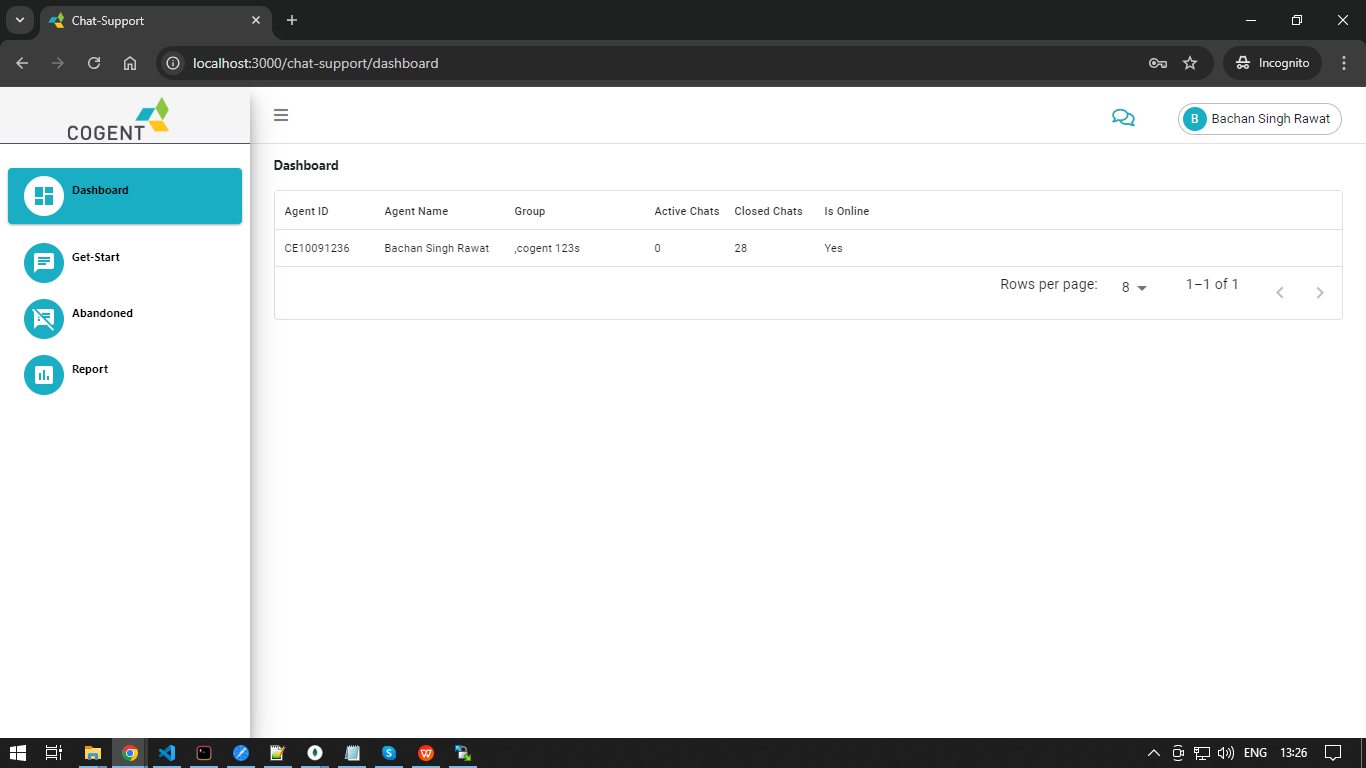
This API is used for sending messages.

1. /api/chat/update-chat

This API updates when chat is over and either customer or agent is closing.

**Agent’s can access pages -**

Agent’s can access Report, Dashboard, Get-Start, Abandoned,Chats.

* 

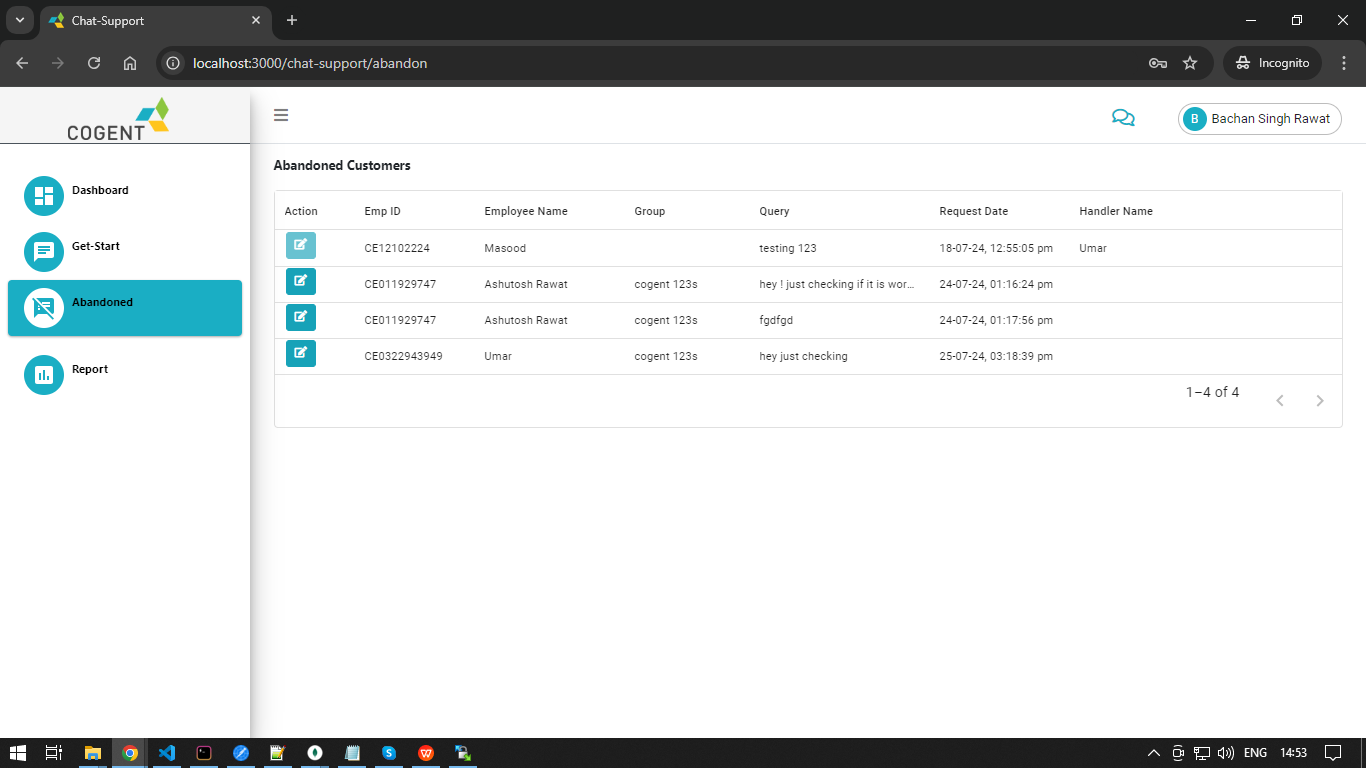
Chats That Are Assigned -

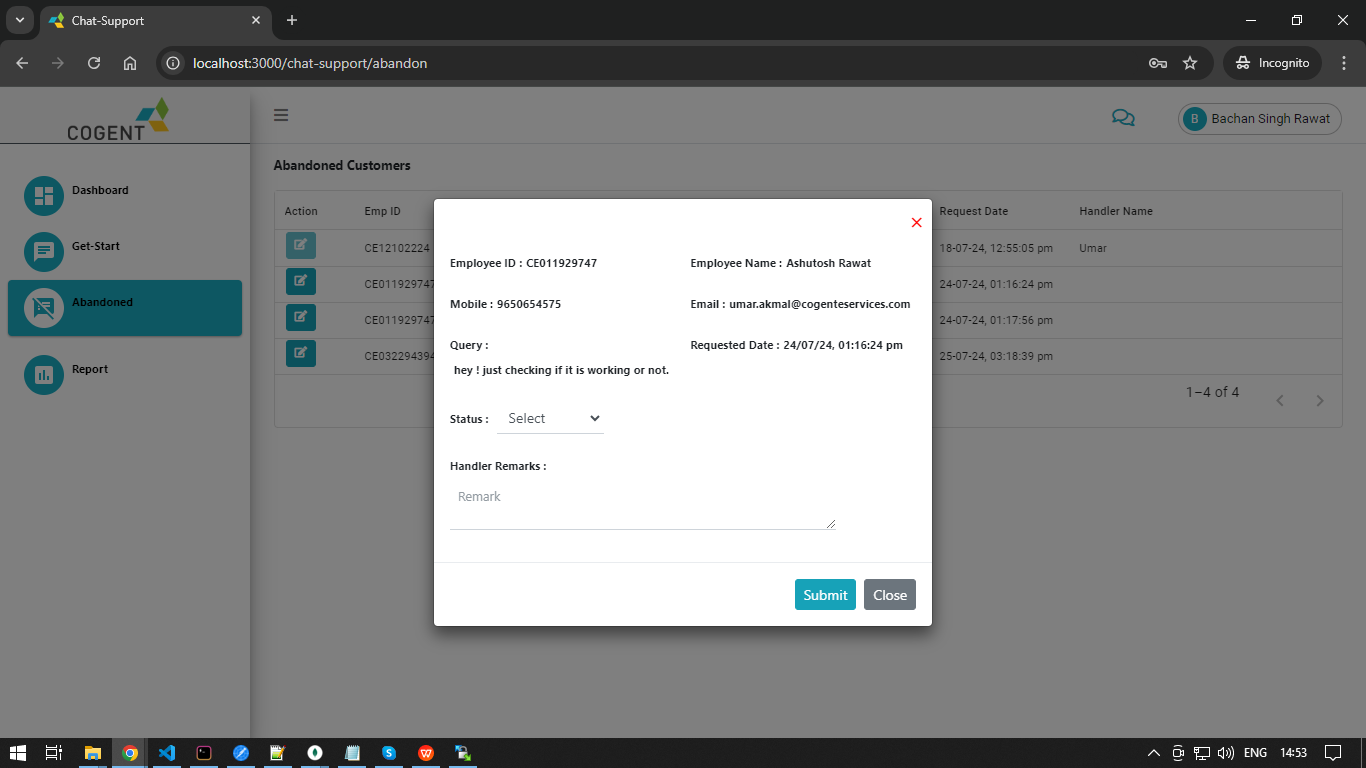
- Similar to customers as mentioned above.

- One agent will have maximum of 10 chats at a time.

Abandoned -

* When agents are unavailable then customers are asked to fill abandon form where contact,email and query of that customer is registered and will reflect to concerning groups here.
* Agents on click of edit button can assign themselves.
* Agent can later contact through email or contact no. and update the status of abandon requests along with remarks.
* When status is closed then it will only reflect in report, here only pending or in progress requests will reflect.
* If abandon request is already is assigned to any agent then it will be disabled for all the other agents





* Validations -

1. Validation on status.

* API s

1. /api/get-abandon

This API gets all the abandon Requests according to the group agents are mapped.

2- /api/assign-hander

This API assigns the abandon requests to the handler.

3- /api/get-abandon-by-id

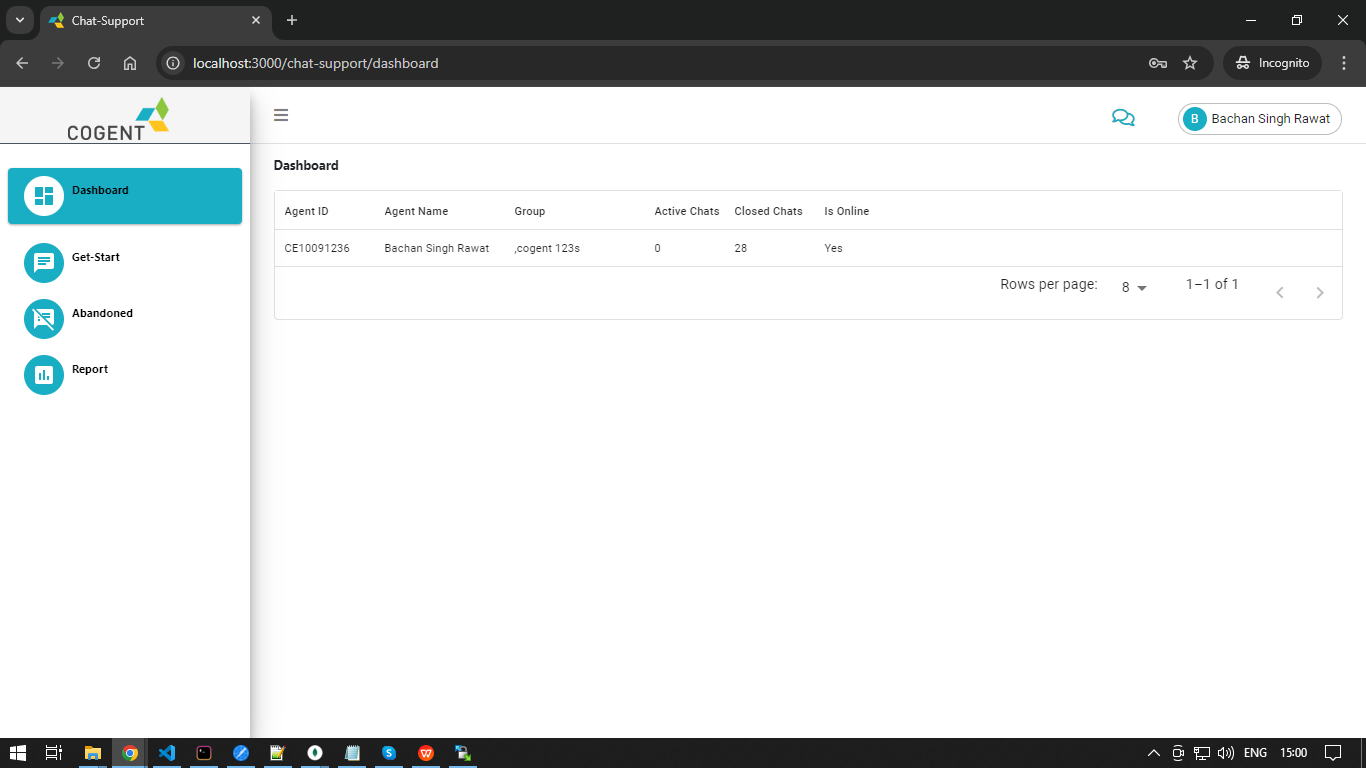
This API gets specific abandon request according to the id.

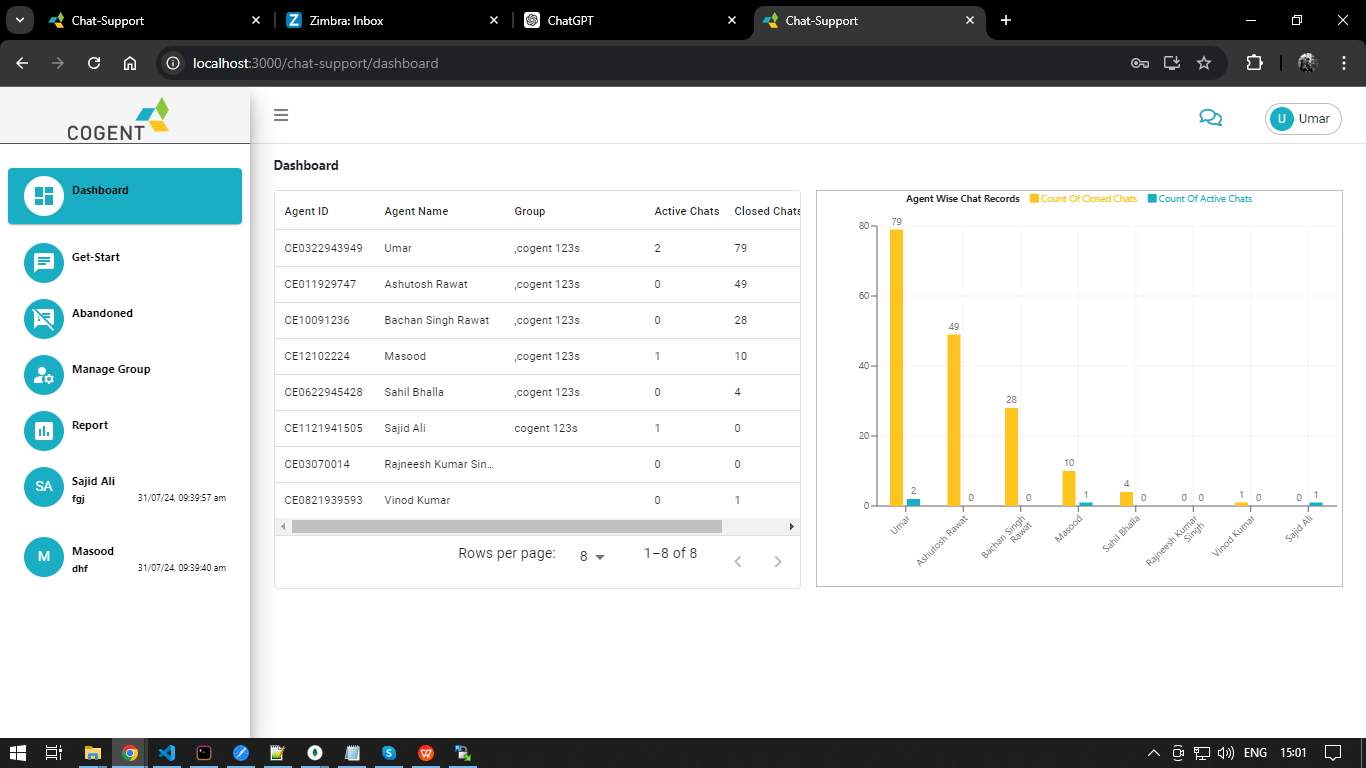
4- api/handler-update

This API updates the abandon request according to the agent input.

Dashboard -

* If users are agents then only that particular agents details of active and closed,chats will be reflected here.
* If user is admin then all the agents details of active and closed chat will be reflected here along with a graph but graph will have only show top 12 agents whose total no. Of chats are greater.
* For agent and admin push notification token is registered on this page.





* API s -

1- http://localhost:3000/sw.js

This API activates service workers for push notifications. `````

2- /api/get-report-dashboard

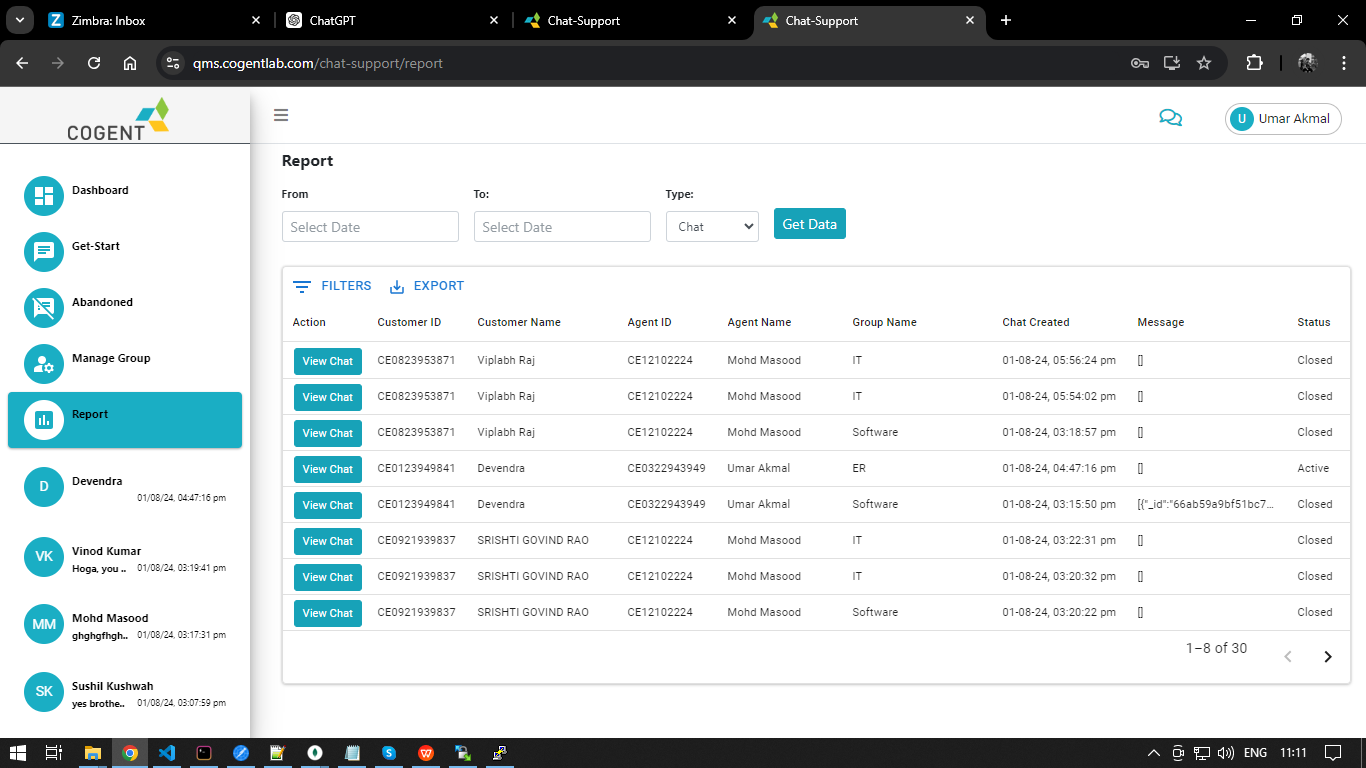
This API gets data for either particular agent or if admin then all the agents of closed chats and active chats

3- /api/get-report-dashboard-graph

This API gets data for dashboard graph.

Report -

* This page shows the report of the Chats and Abandon requests according to date range.
* By default it shows chat report of the current month.
* If admin then it will show all the chats else it will show only login user chat report.



* API s -

1. /api/get-report

This API gets data of chats according to date range if admin then all the chats else only their chats to them.

1. /api/get-report-current-month

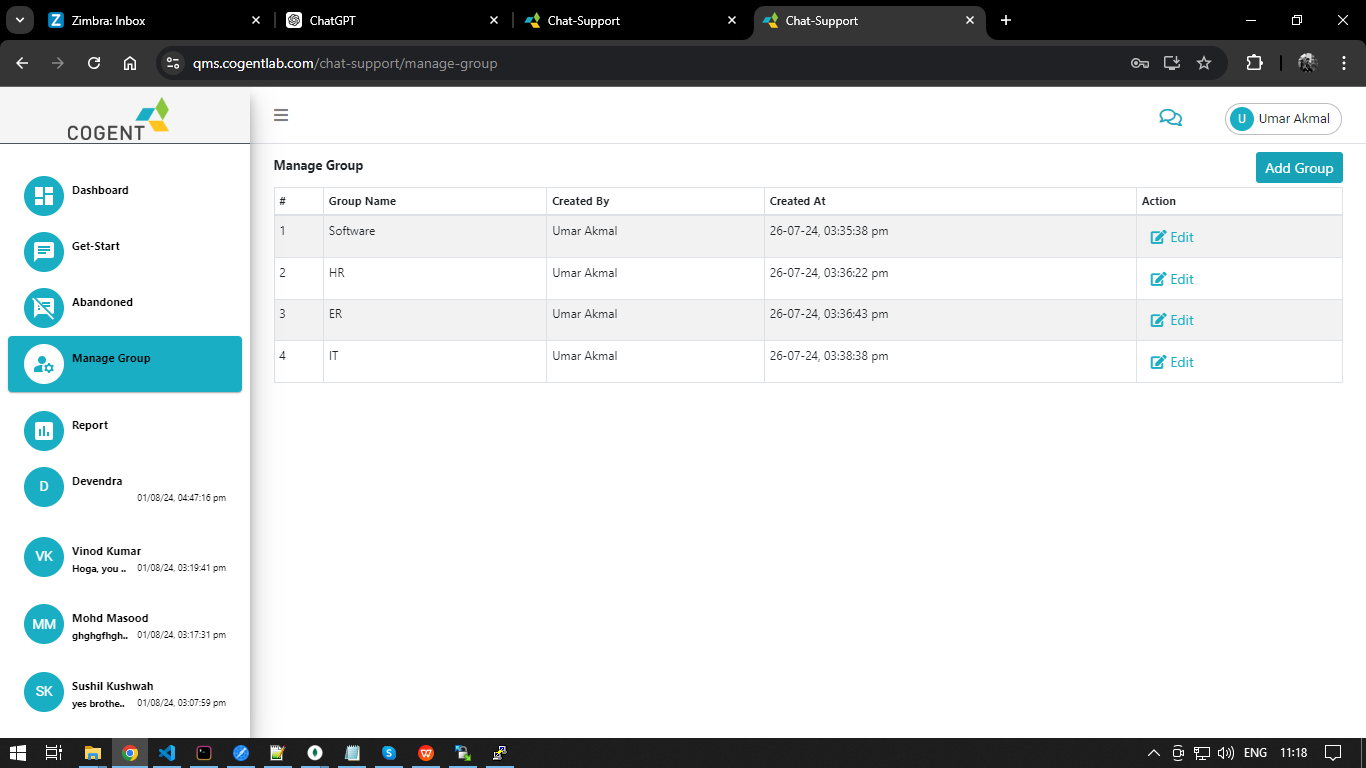
This API gets current month chat when user goes to report.

1. /api/get-report-abandon

This API gets abandon report according to the groups that are assigned to the agents

**Manage Group -**

* This page manages the group where admin can add/edit group and also add/edit users in the group.
* If agent is removed from group then it checks if agent is in any other group if not then agent isAgent status becomes false.



* Validations -

1. When adding user to group min. no. of users should be 2.
2. When exiting users from the group then select at least one.

* **API s -**

1. /api/chat/get-all-manage

This API gets all the groups

1. /api/alladduser

This API gets all users who are not in the particular group for updation of users.

1. /api/alluserleave

This API gets all the users who are currently in the group.

1. /api/alluser

This API gets all the users to add in a new group .

1. /api/chat/get-chat-by-id

This API gets group details by its id for edit.

1. /api/chat/groupAdd

This API adds users to the group.

1. /api/chat/group

This API creates a new group.

1. /api/chat/groupRemove

This API removes users from groups.

Crons -

-There is a cronjob on upload2 folder from where every file except for the latest gets deleted everyday.

Notification - When user is online then socket notification is used but when user is offline then push notification has been used.

**Backend**

Path

bcryptjs

Cors

multer

express

sharp

fluent-ffmpeg

Web-push

express-jwt

**Backend Dependencies**

dotenv

mongoose

Node-cron

jsonwebtoken

cookie-parser

nodemon

Node-mailer

Socket.io

axios

@ffmpeg-installer/ffmpeg

body-parser

helmet

#### Configure Environment Variables

PORT = **5006**

DATABASE='**mongodb://localhost:27017/chatDB**'

JWT\_SECRET="**secret**"

SECRET="**secret**"

CLIENT\_ID=

EMAIL = "**ems@cogenteservices.in**"

PASSWORD = "**cogentems@123**"

MAIL\_HOST = "**mail.cogenteservices.in**"

MAIL\_PORT = "**25**"

DEMO\_API = **https://demo.cogentlab.com/erpm/Services**

BASE\_URL\_FILE = **http://localhost:5006/uploads**

publicVapidKey = '**BIfLg85nlDeDB9sZT2JsHONEFm7sjNR-aXiVQp1r- XjlDrgyU7zOW8VjwcCsMVct1In\_CZnJezxrAKKWKF-kFzQ**'

privateVapidKey = '**JPp5SnMzYnh6Z2BwjpUHKMlJa7PWguv4rJrKiAeLVh4**'

EmailVapid = **umar.akmal@cogenteservices.com**

* **Models**
* **userSchema**

const userSchema = mongoose.Schema(

**Remove whitespaces**

    {

        userid: { type: String,trim: true, required: true},

        username: {type: String,trim: true},

        count: {type: Number,default: 0,},

        isOnline: {type: Boolean, default: true,},

        isAgent: {type: Boolean,default: false,},

        isAdmin: {type: Boolean,default: false,},

    },

    {timestamps: true}

);

* **chatSchema**

const chatSchema = mongoose.Schema(

    {

Reference to user model

        chatName: {type: String,},

        isGroup: { type: Boolean, default: false,},

Reference to message model

        users: [{type: mongoose.Schema.Types.ObjectId,ref: 'User',},],

        latestMessage: {type: mongoose.Schema.Types.ObjectId,ref: 'Message',},

        groupAdmin: { type: mongoose.Schema.Types.ObjectId,ref: 'User',},

        isActive: {type: Boolean,default: true,},

        groupName: {type: String,trim: true},

    },

    {timestamps: true,}

);

* **messageSchema**

const messageSchema = mongoose.Schema(

    {

        sender: { type: mongoose.Schema.Types.ObjectId, ref: "User",},

        message: {type: String,trim: true,},

        reply: {type: String,trim: true,},

        replyId: {type: String, trim: true,},

        attachment: {type: String,trim: true,},

        isRead: {type: Boolean,default: false,},

        chatId: {type: mongoose.Schema.Types.ObjectId,ref: "Chat",},

    },

    { timestamps: true,}

);

* **abandonSchema**

const abandonSchema = mongoose.Schema(

    {

        userid: {type: String,trim: true,required: true,},

        username: {type: String,trim: true},

        mobile: {type: String,trim: true,},

        groupId: {type: String,},

        groupName: {type: String, trim: true,},

        email: {type: String, trim: true,},

        subject: {type: String,trim: true,},

        query: {type: String, trim: true,},

        handlerID: {type: String,trim: true,},

        handlerName: { type: String,trim: true,},

        handlerRemark: {type: String, trim: true,},

        status: {type: String, default: "Pending"},

    },

    { timestamps: true,}

);

* **NotificationSchema**

const notificationSchema = new mongoose.Schema({

    recieverId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },

    chatId: { type: mongoose.Schema.Types.ObjectId, ref: 'Chat' },

    messageId: { type: mongoose.Schema.Types.ObjectId, ref: 'Message' },

    isRead: { type: Boolean, default: false },

},

{timestamps: true,});

* **Controllers**
* **Users**

1. **searchUsers**

The searchUsers controller searches for users by username or userid using a case- insensitive regex query. It excludes the user with the provided \_id. If a search query is present, it filters results accordingly.

1. **getUserById**

The Getuserbyid controller will get specific user id from params and then search for the specific user in database.

1. **Login**

This controller handles user login by verifying credentials with an external API (Get\_Login.php) and managing user authentication in the database.

1. **Logout**

This controller will delete the token .

1. **getAllUsers**

Getallusers controller will get unique userId from body and then it will search for the id in the database then it will return all users.

1. **getAddUsers**

getAddUsers controller will reterive all users except users who are invovle in chat.

1. **getUsersLeave**

The getUsersLeave is an Express.js controller that retrieves users from a specific chat and optionally filters out a given user.

1. **getRecieversData**

This controller retrieves receiverId from body and find receiver’s details in user model .

* **Chat**

1. **accessChats**

This controller handles one-on-one chat access in a messaging system. It either retrieves an existing chat between two users or creates a new chat if one doesn’t exist.

1. **fetchAllChats**

This controller retrieves all chats (both private and group chats) for a logged-in user, sorted by the most recently updated ones.

1. **creatGroup**

The createGroup controller adds rootUserId to users, creates a group chat, sets rootUserId as groupAdmin, updates users' isAgent field to true, and returns the group chat with populated data. Errors are handled if any occur.

1. **renameGroup**

RenameGroup controller will get chatId and chatName from body and update the chatName using findbyIdAndUpdate method.

1. **updateChat**

The updateChat controller retrieves chatId and email from the request body, updates the chat by adding a new user and possibly a new chat name, and if the chat exists, updates the flag for the specified users. If an email is provided, it sends an email notification before returning the updated chat.

1. **addToGroup**

addToGroup controller retreives userId ,chatId and chatName from body.Find the existing chat and it Creates the update object and then  Update the chat with the new user and possibly a new chat name and also  Update users to set isAgent to true .

1. **removeFromGroup**

Removefromgroup controller removes specific user from group and it get specific userId from body.

1. **GetAllGroup**

getAllgroup controller will return all groups present in database.

1. **getActiveChat**

The getActiveChat controller retrieves userId from the request body and checks if there is a chat where the users field includes that userId, while ensuring isGroup is false and isActive is true. If a matching chat is found, it returns the chat; otherwise, it handles errors according

1. **GetAllGroupForManage**

getAllgroupForManage controller try to find all groups and if it is successful then return all existing groups in database.

1. **getChatStatus**

This getChatStatus controller willretrieves chatId from body and then find it inside the chat

Model if the chatId is present it will return chat and if it is not , it returns a message chat not found.

1. **HandlerUpdateEmail**

handleUpdateEmail controller will retrieves id,status,handleRemark,email,subject and name from body and check if status is close then send an email which will have email,subject and emailContent.

* **Message**

1. **sendMessage**

This sendMessage function is an asynchronous Express.js route handler for sending messages in a chat application. It includes file uploads, validation, and optional processing for images and audio.

1. **GetMessages**

This function retrieves messages from a specific chat with pagination and updates notifications.

1. **GetMessagesForReport**

The getMessagesForReport controller retrieves paginated chat messages for a given chat ID.

1. **IsReadUpdate**

This function updates the read status of messages in a chat when the receiver reads them.

* **Notification**

1. **getNotifications**

This controller retrieves unread notifications for a specific user and formats them before sending the response.

1. **deleteNotification**

This controller deletes all the notifications of a specific user.

* **Abandon**

1. **createAbandonData**

This controller takes data from body and save it in abandon model.

1. **getAbandonData**

This controller, getAbandonData, is an Express.js controller that retrieves data about abandoned chats in group conversations for a given user.

1. **assignHandler**

This controller, assignHandler, is an Express.js controller that updates an abandoned chat entry by assigning a handler to it.

1. **getAbandonByID**

This controller getAbandonByID reterives id from body , search’s for the id in the abandon model then returns all data related to that id.

1. **HandlerUpdate**

This controller handlerUpdate is used to update existing document in the abandon model

* **Reports**

1. **getReport**

This controller getReport is an asynchronous Express.js route handler that retrieves chat reports based on a provided date range and an agent ID. It queries the database for chat records, fetches related messages, and formats the response before sending it back.

1. **getReportCurrentMonth**

The getReportAbandon controller fetches records from the Abandon model within a given date range. It extracts date1 and date2, converts them into Date objects, and queries records where createdAt falls between them. If successful, it returns the data; otherwise, it handles errors.

1. **getReportAbandon**

The getReportAbandon controller retrieves date1 and date2 from the request body, converts them into Date objects, and queries the Abandon collection for records created within the specified date range. If successful, it returns the filtered data; otherwise, it handles and returns any errors.

1. **getReportDashboard**

This controller, getReportDashboard generates a dashboard report for agents, analyzing their chat activities. It is used to fetch and process chat-related data.

1. **getReportDashboardGraph**

This controller, getReportDashboard generates a dashboard graph for agents, analyzing their chat activities. It is used to fetch and process chat-related data.

* **Conclusion**

This documentation provides a complete guide to setting up, running, and managing chat-support app using the **MERN** stack**.**