Assignment 2 Report

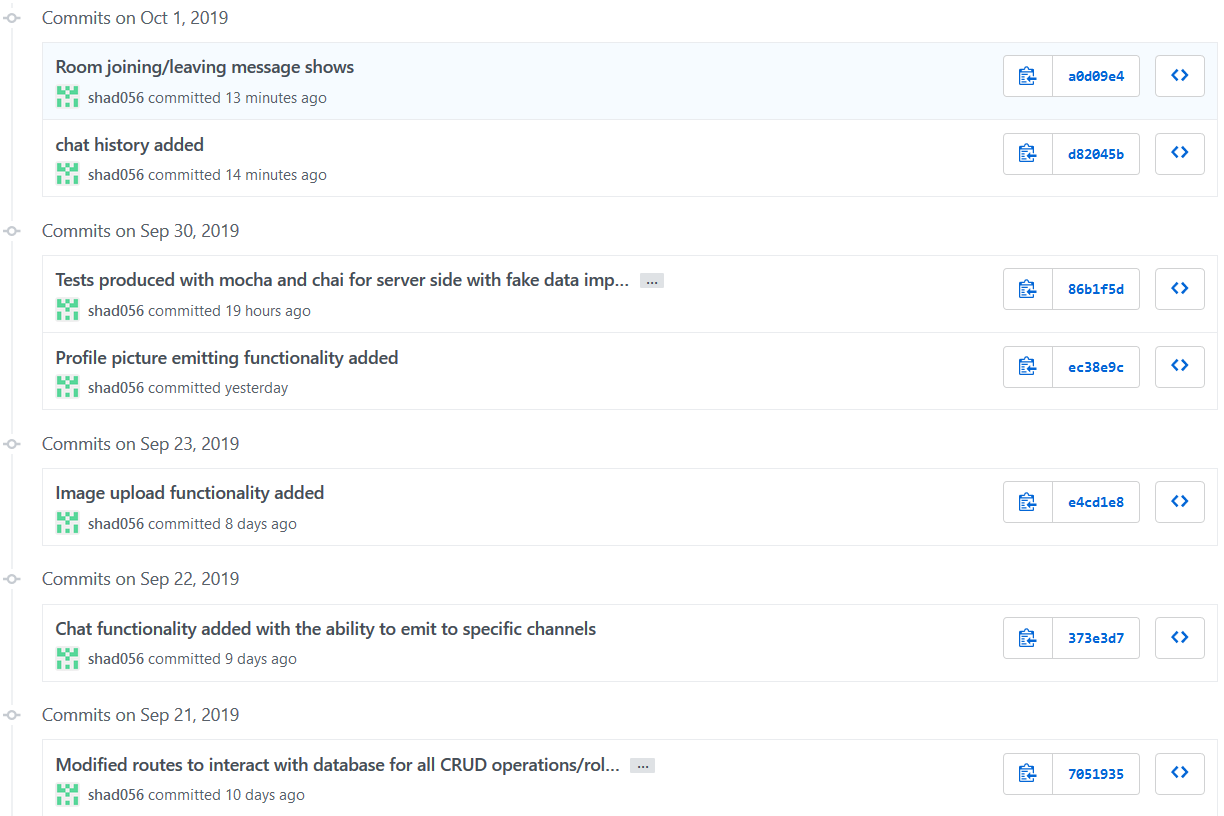
# Introduction

The following report aims to describe the necessary measures taken to develop a chat application using sockets. The chat application will allow users to join various chat groups and channels to communicate with others. The project created contains the functionality of chat operations, along with a dashboard that contains a broad set of permissions associated for different kind of roles to control the application. These roles include admin roles that control the application by adding/removing users, channels and groups. This report will identify the software architecture of the chat application implemented using the MEAN stack technology and will demonstrate the client/server interaction routes, angular/node architecture and routes created to implement the application, along with the use of a version control system.

# Version Control

To clone the repository clone through the following link: <https://github.com/shad056/Assignment-2.git>

A version control tool was used through out the development of this application, this tool is an online web hosting service known as ‘GitHub’ that uses Git to version manage the source code. All the changes in the source code were pushed on GitHub. However, in most cases when a major functionality/s were added these were then committed and pushed to GitHub. Fig 1 displays the total commits for the application.



**Fig 1. Commits on GitHub**

Through local repository it had been easier to add, commit and push the changes to server repository. The first commit made on the 21st Sep 2019 (Fig 1): **Modified routes to interact with database for all CRUD operations/roles**: contains the source code for converting the dashboard from previous project to depend on a database, in this case a NoSQL database known as ‘Mongo DB’ is being used. The main functionality implemented was converting the data dependency from using a file system to manage the operations of CRUD (Create, Read, Update and Delete) to being depend on Mongo DB. This allowed to store all data inside Mongo DB and further transfer and communicate data during the rendering from the server side to implement client side admin operations on the dashboard. The database is used as a local service and contains 3 documents that are: channels (to store group data with their specified channels), users (to store all user data with their roles, groups, channels, email and authentication data) and channelhistory (to store all the chat history within the specified channels). The functions that are dependent on the stored data normally includes: adding/removing groups, users, channels and specifying certain roles to users. These roles have different operations associated with them. The RESTFUL API is mainly responsible for executing a HTTP based request and the server is responsible for matching the request and further rendering the requested operations by fetching the data from the database and sending a valid response, which is rendered on the client side.

The other line of commits made are described as follow:

* **Chat functionality added with the ability to emit to specific channels:** This commit contains the functionality to allow sockets to communicate in real time with the server. Through this it was possible to render server side operations to allow emitting of real time messages within the specified channels that a user chooses to join. Once a channel is joined, the sockets allow to emit messages only to this channel, through which if another user joins this channel, they are able to view the messages within the specified channel.
* **Image upload functionality added:** This commit contains the functionality to add/upload images to the server side, inside a folder to store images. The user selects an image on the client side using a file explorer system and further stores the image. This image has a specified path that is stored on the database. The image stored is normally represented as a profile picture of the user that has uploaded this file and this image is called using RESTFUL API to call a GET request to the server to fetch all the user data from the database**.** Once the image path is known, this path allows to fetch the particular image with the name being contained in the image path. This allows to show the image on the client side.
* **Profile emitting functionality added:** This commit contains the functionality to emit profile picture image names along with the chat messages using sockets in specified channels. The reason for emitting image name, is solely due to the fact that the images are stored on the server side. This emitting process allows to fetch the specified image using its name and further display this image represented as a profile picture for a particular user in the chat room. Sockets play a major role for emitting the real time communication messages within the specified channels, which allow different users to overview the messages and the images/profile pictures.
* **Test produced with mocha and chai for server side with fake data implementation:** Server side routes were further tested using node packages known as mocha and chai. These tests were integrated as integration test cases, where each route had a test associated with it. Furthermore, as these routes contain functionality which are dependent on the data being fetched from the database, therefore, there was fake data implemented before each test started and this data was rendered during tests to produce the expected result. Once the expected result is achieved, the test passes and after the test finishes, the fake data created was removed from the database as it fulfilled its purpose for being rendered for testing with the specified route.
* **Room joining/leaving message shows:** As the sockets contain the functionality for emitting specified messages, it was also essential to implement a functionality to notify each channel when a user joins or leaves that channel. To further elaborate, this functionality was similar to emitting messages in the channel. Once the user joins a channel, a message is emitted with the username and notifying that the user has joined the channel. A similar process is followed when the user leaves the channel.
* **Chat history added:** This commit contained the functionality to add the messages being emitted in the chat rooms to the database for the specified channels that are acting as chat rooms or are chosen as chat rooms by the users. This functionality consisted of sending a POST request each time a chat message is emitted by a user in the channel, this post request contained the header data that included the channel name, user name, date/time and message. This data is rendered on the server side and this allows the server to query the database to find the channel name (that is received in the body request) and once this channel name is found, store the other request data inside this channel, indicating the channel history.

# Data Structures

## Client-Side structures

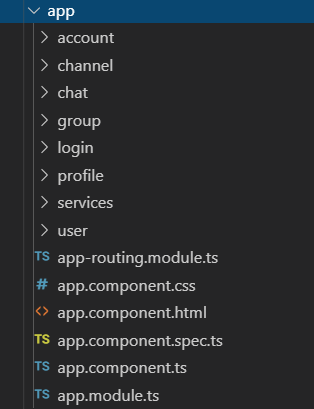
There are various data structures in the client side that range from variables contained within each component to display or store data received from the server side, in order to display it on the browser. The components receive the user data from the server mostly in the form of arrays, to represent or fetch data from the server, the client side contains variables such as groups, channels, roles and users that are arrays used to store data. The variables that store the data are further rendered on the html files and are iterated through ngFor directives to iterate through these arrays. To display certain role operations according to each role, the use of ngIf directives were implemented that checks against certain role permissions e.g. if the authenticated user is a group admin, only then display the option to remove a group. Data binding attributes were used in the html file to collect the entered user input on the browser and further store in a variable e.g. for user input login, the variable ‘selectuser’ is used to store the entered value and send this value with the route to the server side. Similarly select bars are used to render the groups and channels, when a group/channel is selected by the user, data binding stores data in ‘selectgroup’ and ‘selectchannel’ variables respectively. Event handling is also used for buttons on the client side that are binded to a click event linked to a function stored in the ts file. For adding/removing users, groups and channels, all the buttons are linked to their respective functions in the .ts file to perform certain functionalities. Mostly the functions which are linked to buttons, fetch data from the variables that contain data from data binding and the specified service is called to send this data with the specified route contained within the service to allow the server to manage business logic.

## Server-side Structures

In server side, mostly the logic is handled by different routes created for each individual request sent from the client side. The server side depends on a database to fetch/receive data and further render and send it as a response to the client side. A folder named ‘routes’ is created which stores all the routes necessary to match the get/post requests coming from the client side. The body parser module is used to extract the request data from the request routes and further render it on the server side. The server contains similar variables to the client side which include users, groups, channels and roles to store data received from the client side. This data is then used to fetch, add or remove data from the database that contains 3 different documents: channels, channelhistory and users. The structure of users document consists of an array of objects, each object contains the data for a particular user that includes a username, password, email, roles, groups and any channels that the user has been added to from another admin. Similarly, the channels document structure also consists of an array of objects that contains data from each group, it contains the name of the group along with the channels associated for that group. The channelhistory document contains objects that store the history containing user name, messages and date/time within each channel that are received as a result of chat messages being emitted in a particular channel. For each functionality implemented by the client side, there is a different route on the server side for each functionality/request received by the client-side.

# Angular Architecture

* **Components:** Angular applications consist of different components which are linked together to perform a set of functionalities. In this application there is a master component known as ‘app component’, this component stores and renders every component created to display different views and rendering logic. Each component is registered in the app.module file. There are various components used in this application, refer to Fig 2 for the list of the components.



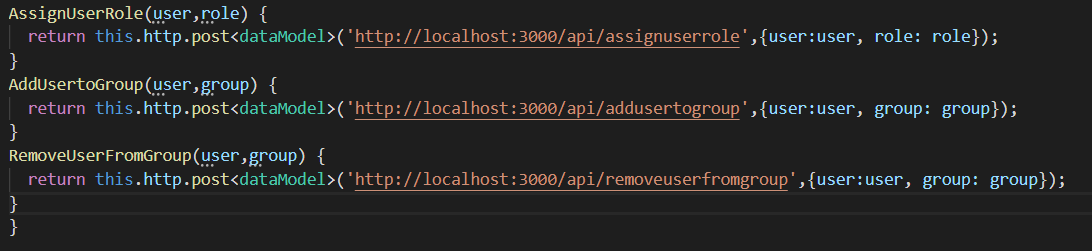
**Fig 2 List of components used in Angular**

Normally in order to navigate from one component to another the use of routing is required which is achieved through built in angular routing. The **app.routing** file in Fig 2 allowed to link a path to its component, on the browser a path can be called and the specified component will be loaded inside the master component (**app.component**). Analysing Fig 2, the **login component** was used to render and display the login form to authenticate a user. The data was gathered from the user input in the login.component.html file and further bind to the logic in the **login.component.ts** which calls a route in a service file to render the logic on the server side. The login component was mainly used to authenticate a user, once the user is authenticated, they are navigated to the **account component** through routing. The account component is responsible for displaying all the functionalities the authenticated user is able to perform according to their set role. These functionalities vary from adding/removing groups/channels or adding/removing users or adding users to various groups/channels. Depending upon what functionality the admin wishes to perform, the particular component is rendered. In this application, if an admin wishes to perform anything related to adding/removing a group/channel, then the certain functionality is navigated to a **group component**, however if the admin performs an operation that involves a user, then the functionality is navigated to a **user component.** There are various id’s set in the html of each button that is linked with a functionality, this id is further fetched in the .ts file and is passed as a parameter to the route that is navigating to the group/user component. It is then the responsibility of the user/group component to fetch the id from the parameter and perform check for the certain functionality to be implemented. E.g. if an admin wants to add a group, and decides to click on the add a group button, then the id associated with the button i.e. ‘addgroup’ will be fetched and passed as a parameter to the route that navigates to the group component. The group component will fetch the id and check if the id is equal to addgroup, upon the id matching, it will show a text box with a button which will be used to create or add a new group. The **chat component** is navigated by the account component when the user selects a group and a channel on the account page and clicks on Goto Chat button, this navigates to the chat component and displays a chat room with a text box to send a message in that particular selected channel. The id of the channel is fetched by the chat component, id is passed as a parameter from the account component. Similarly, the **channel component** performs the exact same functionality as the chat component, the only difference is that channel component is used to display the chat history for that channel and no text box or a button is visible. The services folder in Fig 2 contains the service file used by the components.

* **Services:** A service is used to initiate a client-side request and fetch the response data received from the server. In this case there was only one main service file defined which is known as ‘metaService’, it contained all the list of functions necessary to initiate the get/post request and call the routes to achieve a response back. This service is called by every component to fetch groups/channels/users data and store it in the variables defined in the ts file. Such data is then rendered on the browser to allow the admin to perform certain operations. Upon initiating a functionality, the service is called and a particular function is called which sends a route to the server according to the functionality e.g. If the admin selects a user and clicks on remove a user button then the service function RemoveUser(user) is called from the service which sends a request to the server to render the business logic and remove the selected user from the JSON file. Refer to Fig 3 for all the functions.



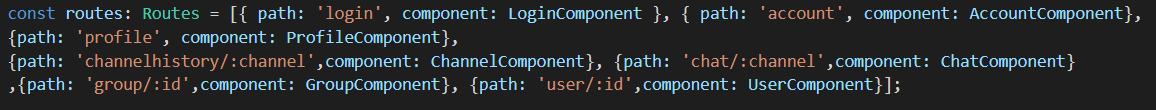




**Fig 3 Functions defined by the service**

The other service being used is known as ‘Socket service’, this service is responsible for containing routes to further allow a user to join/leave a chat room and emit messages by passing user data (user name and message) to one of the specified routes through this service. Mainly the server side file ‘socket.js’ is responsible for emitting the information to a particular channel, however the socket service is reponsible for not only passing this information to the server file but also responsible for subscribing to the subscription for which the service emits the data to be received and rendered.

* **Models:** A model is a normal class that is used to manage the response data more efficiently by allowing the response data to be rendered on the client-side through the variables defined in the class/model. On the server, mostly the variable ‘valid’ is sent which is a boolean value displaying whether the request has been succeeded or not. Therefore, only one model is used through out the application as there are a few routes on the server which are responsible for sending response data that contains the username and an array of that user’s specfied roles, groups and channels. Therefore due to the less or similar amount of response data for each route, the only model file defined is known as ‘dataModel’ (under the services folder). This Model contains user, role, channel and group variables are arrays as the response data coming from the server is in the form of an array (due to data being pushed) and 2 more variables that are message (message received by the response data to be rendered on the client side) and valid (boolean value indicating true from the response (successful request) or false (unsucessful request).
* **Routes:** All the client side routes are distributed in to 2 parts: first part is component routing and the second part are the routes for the Restful API to match with the server routes. The component routing is only used to match a url path with a component so the user is able to navigate to different pages within the application. These routes are defined in the app.routing file. Refer to Fig 4 for these routes.



**Fig 4 Component Routing**

Each path defined in Fig 4 is navigated to its component, on the logic a this.navigateByUrl() function is called in which the path is passed, once that path is initiated the specified component is rendered on the browser.

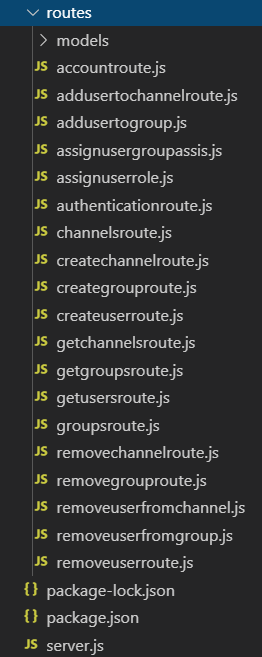
The second part of the routing which relates to the Restful API’s are defined in the services file, each route is specified with a HTTP GET/POST request. The following table further explains the routes associated with each function within the service file (for overviewing the route please refer to Fig 3):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Function** | **Input** | **Route** | **Route purpose** | **Expected Output** |
| Authenticate(uname,password) | Username and password | http://localhost:3000/api/auth | This route takes the user input (uname and password) as a parameter and passes it to the server to check uname/password against the values stored in the database users document. This is used to authenticate users | Valid User: true/false |
| LoadGroups(uname) | Username | 'http://localhost:3000/api/groups' | This route takes the user input (uname) as a paramter and passes it to the server, in order to fetch all the groups that this user has been assigned to | Groups:[] |
| FindChannel(group) | Group name | http://localhost:3000/api/channels | This route takes the group name (group) as a paramter and passes it to the server, in order to fetch all the channels that are within this group | Channels:[] |
| AddGroup(group) | Group Name | http://localhost:3000/api/creategroup | This route takes the user input (group) as a paramter and passes it to the server, in order to create this new group in the JSON file | Group Created/Error |
| AllGroups() |  | http://localhost:3000/api/getgroups | This route fetches all the groups from the server through the JSON file | Groups:[] |
| AddChannel(group, channel) | Group, Channel | http://localhost:3000/api/createchannel | This route takes the user inputs (group, channel) as a paramter and passes it to the server, the parameter group allows to choose the group and the parameter channel is the name of the channel that the user wants to be created. The server further takes these parameters and creates a channel within the specified group in the JSON file | Channel Added/Error |
| AddUser(user,email) | User, email | http://localhost:3000/api/createuser | This route takes the user input (user (username to be created), email(email of the user to be created)) as paramters and passes it to the server. The server creates a new user | User added/Error |
| AllChannels() |  | http://localhost:3000/api/getchannels | This route is passed to the server and is used to fetch all the channels available in all groups in the JSON file | Channels:[] |
| AllUsers() |  | http://localhost:3000/api/getusers | This route is passed to the server and is used to fetch all the users available in the JSON file | Users:[] |
| AddUsertoChannel(user,channel) | User, channel | http://localhost:3000/api/addusertochannel | This route takes the user input (user (the user), channel (the channel name)) as paramters and passes it to the server, in order to add the user in the channel | User added to channel/error |
| RemoveGroup(group) | Group | http://localhost:3000/api/removegroup | This route takes the user input (group (group name)) as a paramter and passes it to the server, in order to remove that particular group from the JSON file | Group Removed/Error |
| RemoveChannel(channel) | Channel | http://localhost:3000/api/removechannel | This route takes the user input (channel (channel name)) as a paramter and passes it to the server, in order to remove the specified channel from the JSON file | Channel Removed/Error |
| RemoveUserfromChannel(user,channel) | User,channel | http://localhost:3000/api/removeuserfromchannel | This route takes the user inputs (user (user), channel (name of channel)) as paramters and passes it to the server, in order to remove user from the specified channel | User is removed from channel/Error |
| RemoveUser(user) | User | http://localhost:3000/api/removeuser | This route takes the user input (user) as a paramter and passes it to the server, in order to remove the specified user data from the JSON file | User is removed/Error |
| AssignUserGroupAssis(user) | User | http://localhost:3000/api/assignusergroupassis | This route takes the user input (user) as a paramter and passes it to the server, in order to assign the specified user the role of a Group Assis | User is assigned the role of group assis/Error |
| AssignUserRole(user,role) | User, Role | http://localhost:3000/api/assignuserrole | This route takes the user inputs (user, role) as paramters and passes it to the server, in order to assign the specified user the input role | User is assigned a role/Error |
| AddUsertoGroup(user,group) | User, Group | http://localhost:3000/api/addusertogroup | This route takes the user input (user, group) as paramters and passes it to the server, in order to assign the specified user into the input group | User is added to group/Error |
| RemoveUserFromGroup(user,group) | User, Group | http://localhost:3000/api/removeuserfromgroup | This route takes the user input (user, group) as paramters and passes it to the server, in order to remove the specified user from the input group | User is removed from a group/Error |

**Fig 5 Routes defined for the client side**

# Node Architecture

* **Modules:** Modules are essentially encapsulates all related code into a single file. The main file that is being used to initialize the server and import all the necessary packages/modules is the server.js file. The server.js file imports all the defined modules which has necessary routes to match the specified client side requests. The defined modules are inside this folder called ‘routes’, these routes have the modules which render the business logic according to the specified requested routes. The file architecture for node is displayed in Fig 6.



**Fig 6 Routes for requests**

* **Functions:** Functions are mostly defined inside the routes, the purpose of such functions is to render the business logic and match the client side requests through the express JS middle ware. There are various functions according to each file. The main functions used are inside routes to match the http based request occurring from the client side and further render business logic.
* **Files:** There are various files within the node architecture each with their own purpose. The **server.js** file is used to initialize the server on local host port 3000, it also sets up routing for express JS and imports packages such as body-parser (for extracting request body parameters), fs (an API that contains all asynchronous and synchronous functions for file I/O), express, path (for path parsing and resolution for file paths), mongodb dependency modules to make a connection to the mongo db local database. Other files are under routes folder that match the specified HTTP GET/POST requests from the client side and perform operations by using functions from the fs module and express module to send back a response.
* **Global variables:** There were a few global variables used that were used for extracting request body data, these variables were mainly named as username, group, channel and role. These variables were globally defined in the server.js file and allowed to be used in the route files.

# Client-Server Interaction

This was the most crucial aspect of this project and in order to implement this, it was important to efficiently map out routing in both client and server. The first step to initiate the interaction was to define routes in the client side, which were defined in a service file (refer to Fig 5 for a brief description of these routes). The second step was to define the same routes in the server file to be matched (separate files were defined under the routes table). Upon the user initiating an action on the client, an event action is called e.g. button press. Upon such action, the event is handled by an event handler (function bind to the button), this function allows to fetch the data available in the request body and further call the service with the specified function which contains the route for the function to be implemented. Each event handler is linked to a different service function depending on the functionality required. Once the function is called a RESTFUL API is initiated which is passed to the server, upon the express routes matching the particular route passed, the server fetches the client request body data. This data is then rendered on the server and depending on the required functionality, it is written on a JSON file or other data related to it is read from a JSON file. This data is then fetched and sent as a response along with another variable known as valid which is set to true due to a successful operation and can also be set to false upon the failure of such operation. This response is received on the client due to the client subscribing the service function that sent the request, in order to fetch back a response, once a response is received with the valid variable being equal to true, the client fetches the data associated with the response. This data is rendered on the view to the user. If in case the client receives the valid variable as false, then the client renders an error on the view. A list of all the routes, parameters, return values and purpose on the server side are displayed in Fig 7.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Files** | **Routes** | **Parameters** | **Return Values** | **Purpose** |
| Addusertochannelroute.js | app.post('/api/addusertochannel' | username | Valid: true/false | This route is used to add a user to a specified channel |
| Addusertogroup.js | app.post('/api/addusertogroup' | User, group | Valid: true/false | This route is used to add user to a specified group |
| Assignusergroupassis.js | app.post('/api/assignusergroupassis' | user | Valid: true/false | This route is used to assign a user the role of a group assis |
| Assignuserrole.js | app.post('/api/assignuserrole' | User, role | Valid: true/false | This route is used to assign the user a specified role |
| Authenticationroute.js | app.post('/api/auth' | User,password | Username, roles, groups, channels, Valid: true/false | This route is used to check against all the values in the users.json to match the specified user in order to authenticate the user and send user related data |
| Channelsroute.js | app.post('/api/channels' | group | Channels, Valid: true/false | This route is used to fetch all the channels associated with the specified group |
| Createchannelroute.js | app.post('/api/createchannel' | Channel, group | Valid: true/false | This route is used to create a new channel in a group |
| Creategrouproute.js | app.post('/api/creategroup' | Group | Valid: true/false | This route is used to create a new group |
| Createuserroute.js | app.post('/api/createuser' | User | Valid: true/false | This route is used to create a new user |
| Getchannelsroute.js | app.get('/api/getchannels' |  | Channels, Valid: true/false | This route is used to fetch all the available channels |
| Getgroupsroute.js | app.get('/api/getgroups' |  | Groups, Valid: true/false | This route is used to fetch all the available groups |
| Getusersroute.js | app.get('/api/getusers' |  | Users, Valid: true/false | This route is used to fetch all the available users |
| Removechannelroute.js | app.post('/api/removechannel' | Channel | Valid: true/false | This route is used to remove a particular channel from a group |
| Removegrouproute.js | app.post('/api/removegroup' | Group | Valid: true/false | This route is used to remove a group and all the channels associated with it |
| Removeuserfromchannel.js | app.post('/api/removeuserfromchannel' | User, Channel | Valid: true/false | This route is used to remove a channel from a user |
| Removeuserfromgroup.js | app.post('/api/removeuserfromgroup' | User, Group | Valid: true/false | This route is used to remove a user from a group |
| Removeuserroute.js | app.post('/api/removeuserroute' | User | Valid: true/false | This route is used to remove a user |

**Fig 7 List of all node routes**

The database documents were affected depending on which route is called, if an attribute is added through the add paths, the document gets pushed with more data. However, if the route for removing data is called the documents are affected by removing the required data. Once the response of valid true/false is sent to the client. This response indicates how the view will be displayed. Depending on the response values of the server, the client is affected by it. If the server provides values by using its global variables, they are changed for each route and therefore store different data for different requested routes. The client receives this data and displays it on the view. The angular component will have its variables being assigned the response data, in order to render this data on the view. This means that this data can be used on the front end and further managed to be displayed to the user.

## Chat functionality

The chat functionality was implemented using **sockets.io** that allow real time communication on the server. In order to implement a full fledge chat system, a socket.js file was produced on the server side that includes all the functions to allow the emitting of messages passed from the client side. Starting with the client side a socket service was used that consists of different functions to enable communication with the server side file. The following table explains what the purpose of each function within the socket service is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Client Function** | **Input** | **Purpose** | **Output** |
| joinRoom() | Username, channel name, image path | Once a user joins a channel, this function is called in the ngOnInit() function of the component. The purpose of this function is to emit the input data to the socket to indicate that the user has joined the channel (the channel is determined from the passed in channel name in the input data). Through this function, the server matches the join event and allows the user to join a channel | None |
| newUserJoined() |  | When the server side emits the data in the join event to allow the user to join a channel, the input data is passed on/emitted and the client side function newUserJoined further receives this data and subscribes to it. The message within the received data is emitted to a channel indicating that a user has joined a channel. The emitted data also contains the user name of the user which emitted the data with their image path that is rendered as a profile picture on the client side | User name, image path and message |
| leaveRoom() | Username, channel name, image path | Once a user leaves a channel, this function is called in the ngOnDestroy() function of the component. The purpose of this function is to emit the input data to the socket to indicate that the user has left the channel (the channel is determined from the passed in channel name in the input data). Through this function, the server matches the leave event and allows the user to leave a channel |  |
| userLeftRoom() |  | When the server side emits the data in the leave event to allow the user to leave a channel, the input data is passed on/emitted and the client side function UserLeftRoom further receives this data and subscribes to it. The message within the received data is emitted to a channel indicating that a user has left a channel. The emitted data also contains the user name of the user which emitted the data with their image path that is rendered as a profile picture on the client side | User name, image path and message |
| sendMessage() | Username, channel name, image path, message | Once a user sends a message in a channel, this function is called in the client side. The purpose of this function is to emit the input data to the socket to indicate that the user has sent a message in the channel (the channel is determined from the passed in channel name in the input data). Through this function, the server matches the message event and allows the message to be further emitted to the sent channel name |  |
| newMessageReceived() |  | When the server side emits the data in the message event to allow the channel to display the message sent by a user, the input data is passed on/emitted and the client side function newMessageReceived further receives this data and subscribes to it. The message within the received data is emitted to a channel indicating that a user has sent a message. The emitted data also contains the user name of the user which emitted the data with their image path that is rendered as a profile picture on the client side | User name, image path and message |
| RecordHistory() | Username, message, date Time, Channel | This function is called after every function subscribing to an event being emitted in the chat functionality. Once the data is received in the channel, this data is further sent to the server side using this function in a post request. The server receives this data and stores it in the database for the particular channel. This data is represented as the chat history occurred in that channel |  |

The server side file socket.js has the following functions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Server Function** | **Input** | **Purpose** | **Output** |
| socket.on('join', function(data) | Username, channel name, image path | When the joinRoom() function is called on the client side, it matches with the join event on the server side, which allows the user to join a particular channel |  |
| io.in(data.channel).emit('join' |  | This function further emits the join event in the particular channel and sends back the input data received in the join event along with a message indicating that a user has joined a channel | Username, message, image path |
| socket.on('leave', function(data) | Username, channel name, image path | When the leaveRoom() function is called on the client side, it matches with the leave event on the server side, which allows the user to leave a particular channel |  |
| io.in(data.channel).emit('leave' |  | This function further emits the leave event in the particular channel and sends back the input data received in the leave event along with a message indicating that a user has left a channel | Username, message, image path |
| socket.on('add-message',function(data) | Username, channel name, image path | When the sendMessage() function is called on the client side, it matches with the add-message event on the server side, which allows a channel to receive a message that is sent by a user |  |
| io.in(data.channel).emit('new message' |  | This function further emits the add-message event in the particular channel and sends back the input data received in the event, this indicates the received message which was sent by the user in the channel | Username, message, image path |

## Testing

Testing had been conducted on the server side using mocha and chai node frameworks. Each server side route was tested according to their functionality. In order to validate each operation, fake data was implemented/created on the database before each test conducts and after the test is completed, this data is removed from the database. The following table explains the purpose of each test and its expected results with the data implemented before/after test:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Route** | **Tests** | **Before Test/s** | **After Test/s** | **Expected Result** |
| Adding Image Route | 1. Should upload image and return true in response 2. Should upload image and return no errors | Create a new user | Delete the created user | 1. Image uploaded 2. Image uploaded and no errors observed |
| Adding user to a channel | 1. Should add user to a channel 2. Should add user to a channel and not return any unexpected parameters | Create a new user | Remove the created user | 1. The user was added to a channel 2. The user was added to a channel and no unexpected parameters were returned |
| Adding user to a group route | 1. Should add user to a group 2. Should add user to a group and not return any unexpected parameters | Create a new user | Remove the created user | 1. The user was added to a group 2. The user was added to a group and no unexpected parameters were returned |
| Assigning user a group assis role | 1. Should assign user a group assis role 2. Should assign user a group assis role with no errors | Create a new user | Remove the created user | 1. The user is assigned a group assis role 2. No errors were returned |
| Adding user a role | 1. Should assign user a role 2. Should assign user a role with no errors | Create a new user | Remove the created user | 1. The user is assigned a role 2. The user is assigned a role and no errors returned |
| Authentication rote | 1. Should authenticate a user 2. Should not authenticate a user | Create a new user | Remove the created user | 1. The user was authenticated 2. The user was not authenticated |
| Create a channel route | 1. Should create a channel in a group 2. Should show an error for an existing channel in the group | Create a new group | Remove the created group | 1. Creates a channel in a group 2. Shows error |
| Create a group route | 1. Should create a group 2. Should show an error for an existing group |  | Remove the created group | 1. Creates a group 2. Shows error |
| Creating a user route | 1. Should create a user 2. Should show an error for an existing user |  | Remove the created user | 1. Creates a user 2. Shows error |
| Getting channels route | 1. Should get all channels 2. Should get all channels with no errors |  |  | 1. Gets all channels 2. Shows no errors |
| Getting groups route | 1. Should get all groups 2. Should get all groups with no errors |  |  | 1. Gets all groups 2. Shows no errors |
| Getting users route | 1. Should get all users 2. Should get all users with no errors |  |  | 1. Gets all users 2. Shows no errors |
| Getting history route | 1. Should get history for a channel 2. Should get history with no errors | Create a channel | Remove the created channel | 1. Gets history 2. Shows no errors |
| Get a user data route | 1. Should get user data 2. Should not get user data | Create a user | Remove a user | 1. Gets user data 2. Does not gets user data |
| Recording history route | 1. Should record history of a channel with a user 2. Should not record history | Create a channel and a user | Remove channel and a user | 1. Records history of a channel 2. Does not records history of a channel |
| Removing a channel route | 1. Should remove a channel 2. Should remove a channel with no errors | Create a channel | Remove a channel | 1. Removes a channel 2. Shows no errors |
| Removing a group route | 1. Should remove a group 2. Should remove a group with no errors | Create a group | Remove a group | 1. Removes a group 2. Shows no errors |
| Remove user from channel | 1. Should remove user from a channel 2. Should not remove user from a channel | Create a user | Remove a user | 1. Removes a user from a channel 2. Does not removes a user from a channel |
| Remove user from group | 1. Should remove user from a group 2. Should not remove user from a group | Create a user | Remove a user | 1. Removes a user from a group 2. Does not removes a user from a group |
| Remove user | 1. Should remove a user 2. Should not remove a user | Create a user | Remove a user | 1. Removes a user 2. Does not removes a user |

All the above tests were executed as integration tests using the command ‘npm run-script test’. The describe function is used to describe the tests with a description for each tests and it is used to perform the functionality for an individual test to compute a final result. The before function is called before a test is conducted, this function creates data on the database which the tests can use to render and achieve a final expected result. The after function is called after the tests are completed and a result is achieved, the after function removes the created data by the before function from the database. This is done to not disturb the process of the entire application. All the tests passed successfully.