Assignment 2 Report

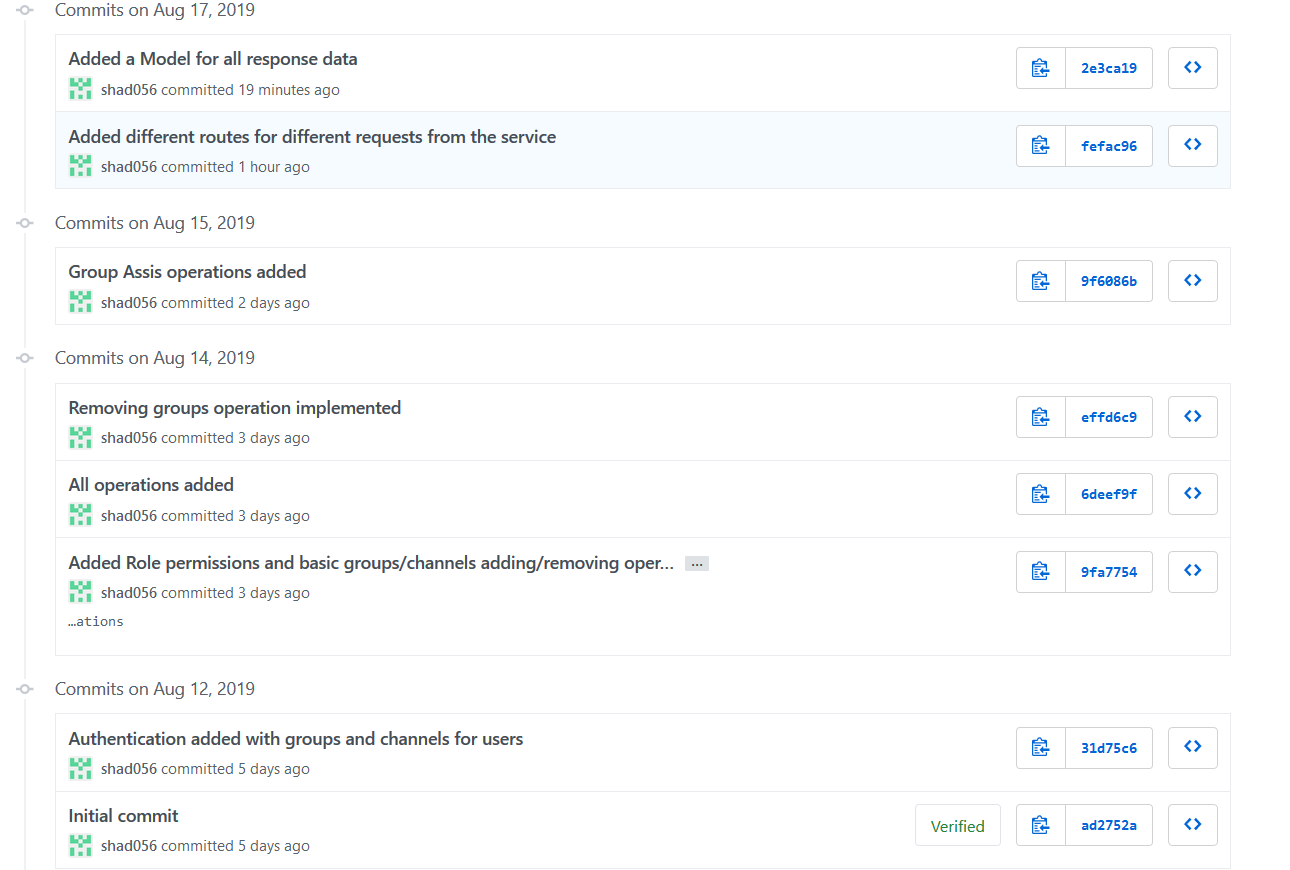
# Introduction

The following report aims to describe the necessary measures taken to produce a dashboard for a chat application. The chat application will allow users to join various chat groups and channels to communicate with others. The first project created does not contain the functionality of chat operations, however it 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 dashboard 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-1.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**

Referring to Fig 1, the first commit made on the 12th Aug 2019 was an initial commit to set up the organization of Git repository to further clone it into a local repository. Through local repository it had been easier to add, commit and push the changes to server repository. The second commit made on the 12th Aug 2019 (Fig 1), contains the source code for authenticating users based on their username only, this was possible by creating a JSON file named ‘users.json’ file which stored a set of user data. This user data contains usernames with their email, roles, groups and any custom channels that they had been added to by an admin. The JSON file is checked against the entered user input on the client, upon matching the user input with the username on the server through the file reading operation for the users.json file, the user is granted access to an account page. Furthermore, this commit also contains the data for displaying the groups and channels that the user is associated with and this data is further used on the client side to navigate to a channels chat page or viewing the chat history for that channel associated within a group. Channels are encapsulated within certain groups, that is each group contains different channels, a JSON file called ‘groups.json’ was created that reads the groups and their channels. A user is added to a group which displays the channels the user can access according to their group, however if an admin wants, they can add channels from different groups to the user to access as well.

The line of commits made on the 14th of August 2019 are described as follow:

* **Added Role permissions and basic groups/channels adding/removing operations:** This commit contains different types of admins that will control the application. These admins were first added to the users as roles for testing purposes. Later on, the super admin was chosen as the role that was held responsible for managing all role permissions and assigning different users with a group admin or group assis role. A service file named ‘metaService’ was added that corresponded to adding different routes from the client side that initiate a get/post request that will match with the same server-side routes. Server-side routes were further added that matched the client-side requests and then performed the business logic to further send response data to be rendered in the presentation view. In this commit, role permissions were analysed and checks were performed through out the account page to display different operations available for different roles. Based on the type of role a user was assigned, the basic operations such as adding a group/channel or deleting a group/channel were further implemented. These operations used the routes in the service to send the requested data that the group admin/super admin selects from the client side i.e. a group/channel. This request data is sent through the API route from the client and is received/matched by the server to further add or remove a group/channel from the available JSON files. Upon a successful addition/deletion by the server, the response data is sent back to the client, this response data indicates whether the operation was successful or not. Furthermore, once the user is authenticated, their session details are stored in a local storage, a log out functionality was implemented which logs the user out by removing the user session data from local storage, which redirects the user back to the login page and restricts them from the account page.
* **All operations added:** This commit was a further improvement on the previous commit and further fixes and modifies the operations previously produced, in order to perform more efficiently. Furthermore, it adds the functionality of adding/removing users through the same procedure as described for the previous commit. Role permissions assignment functionality was also added for users.
* **Removing groups operation implemented:** In most cases, it was essential to check that if a group has been removed, then its channels also must be removed. This logic was not implemented before and further had been caused due to a human error. Therefore, the logic of removing groups was re-implemented in order to remove their associated channels or the channels that the users were added to which belonged to the same group.

Group Assis operations were added on the 15th Aug 2019 (Fig 1). This role allows users to be added in a group or removed from a group, this role also allows to create channels in a group. These functionalities were implemented and allowed to add/remove group data for each user in the users.json file. The ability to create channels for a group allowed to store channels for a particular group in the channels.json file. The channels.json was added that is used to fetch channels associated for each group, such file was used to add channels into a group, remove channels in a group or fetch channels from each group to allow users to join a chat room or display chat history.

The commits made on the 17th of August 2019 are described below:

* **Added different routes for different requests from the service:** There were a distribution of various routes that were added for each functionality within different angular components. However, in order to manage it and make the code structure more reproducible and robust, a service file was created (metaService) that stores the front-end request routes. Furthermore, these routes were arranged more efficiently in this commit to separate different routes for each particular functionality committed and additional server-side routes were added to match the front-end requests to separate functionalities from each other for better encapsulation and robustness of code.
* **Added a model for all response data:** Upon the server performing business logic successfully, it sends some response data that is used by the client to display the user whether their requested operation was successfully completed by the server or not. This response data required sending variables each with different types, in order to manage this data on the client side, a class was created as a model. This model is named as ‘dataModel’ which contains the variables with their different types to be matched with the response data, and this allows to manage the response data more efficiently to be rendered by Angular.

# Data Structures

## Client-Side structures

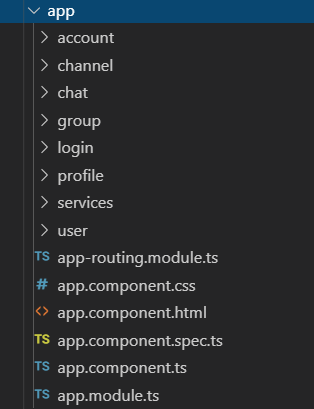
There are various data structures in the client side that range from variables contained within each component to display or store data received from the server side, in order to display it on the browser. The components receive the user data from the server mostly in the form of arrays, to represent 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. 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 users.json or the channels.json file. The structure of users.json file consists of an array of objects, each object contains the data for a particular user that includes a username, email, roles, groups and any custom channels that the user has been added to from another admin. Similarly, the channels.json file 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. 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.



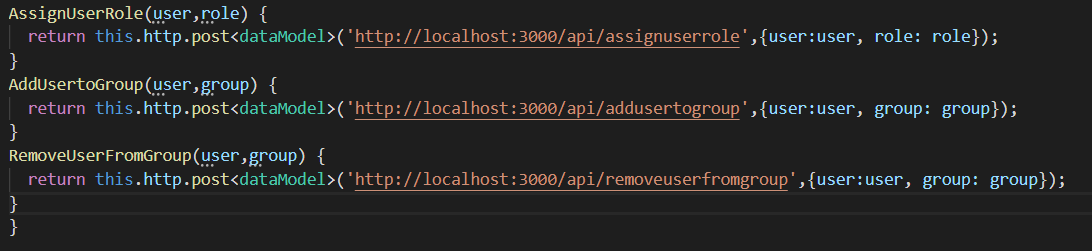
**Fig 2 List of components used in Angular**

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 in 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.

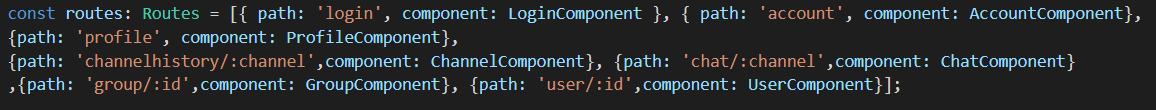






**Fig 3 Functions defined by the service**

* **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.

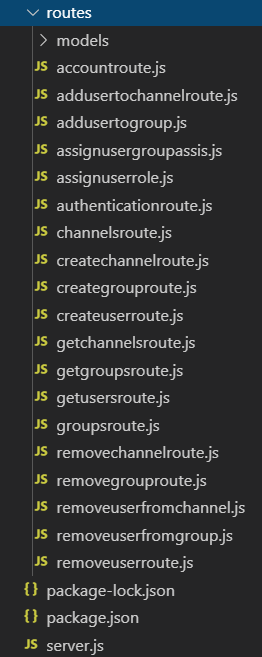
The second part of the routing which relates to the Restful API’s are defined in the services file, each route is specified with a HTTP GET/POST request. The following table further explains the routes associated with each function within the service file (for overviewing the route please refer to Fig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

|  |  |  |
| --- | --- | --- |
| **Function** | **Route** | **Route purpose** |
| Authenticate(uname) | http://localhost:3000/api/auth | This route takes the user input (uname) as a parameter and passes it to the server to heck uname against the usernames in the users.json file. This is used to authenticate users |
| LoadGroups(uname) | '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 |
| FindChannel(group) | 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 |
| AddGroup(group) | 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 |
| AllGroups() | http://localhost:3000/api/getgroups | This route fetches all the groups from the server through the JSON file |
| AddChannel(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 |
| AddUser(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 |
| 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 |
| 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 |
| AddUsertoChannel(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 |
| RemoveGroup(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 |
| RemoveChannel(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 |
| RemoveUserfromChannel(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 |
| RemoveUser(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 |
| AssignUserGroupAssis(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 |
| AssignUserRole(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 |
| AddUsertoGroup(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 |
| RemoveUserFromGroup(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 |

**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 **readFile** and **writeFile**. These functions are imported from a fs package and are used for reading and writing data to the JSON file.
* **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). 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 | 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 JSON files were affected depending on which route is called, if an attribute is added through the add paths, the JSON file gets pushed with more data. However, if the route for removing data is called the JSON files 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.