*3813ICT – Software Frameworks*

*We Are Tree*

Student: Zak Barker

Student#: S5085150

Subject: 3813ICT Software Frameworks

Email: Zak.Barker@Griffithuni.edu.au

* 1. ***Git Repository***

My Git repository is very simple, I’ve cloned my angular project directly into the repository in the same format it was created in. I have stored my Angular folder separate from my Server side files and the two are linked together with help from cors. My Server side holds my server.js file along with all of my routes.

I’ve been using my Github as a basic storage location. I did have a few minor problems, when I first created my assignment it was located inside my 3813ICT coursework folder along with all of my weekly workshops. I needed to copy the file and create a dedicated Git Repository for it so I lost all of the commits I did before that.

My approach to version control was simple, I committed after every few routes I created. Furthermore, I did need to create a test branch when I decided to test out changes to my data structure and did not want to ruin my entire assignment. This test was successful so I merged the branch back into my master. Furthermore, during implementation of assignment two I made another branch in order to test sockets without damaging the original project before merging the two.

* 1. ***Data Structure***
     1. ***Server Side***

My server side data structure is stored in a Mongo Database file. There are four collections within this database: Users, Groups, Channels and Messages

**Users**

* Username – String
* Password - String
* Email – String
* Role – String (‘Super’, ‘Admin’, ‘Assis’, ‘User’)
* Img - String

**Groups**

* Name – String
* Admin – String (User)
* Assis – Array (Array of Users)
* Users – Array (Array of Users)

**Channels**

* Name – String
* Group – String (Group)
* Access – Array (Array of Users)

**Messages**

* User – String (User)
* Group – String (Group)
* Channel – String (Channel
* Messages – Array (Array of Strings)

***2. Client Side***

Data is fetched from the server side and then it is stored on the client side. Once received, the client runs a number of functions on the data to prepare it before storing it in variables. The data is stored in the following arrays:

* **Groups** – This variable holds all of the groups and channels which are relevant to the user. This is the groups which user is apart of.
* **Users** – This variables holds all of the users except for the current User
* **User** – This variable holds all information for current user
* **Username**– Holds the current user’s username
* **UserRole** – Holds the current user’s role
* **GroupData**– Holds data for the current group which user is in
* **Channel** – Holds data for the current Channel User is in
* **Messages** – Holds data for all messages in database
* **Room** **Messages** – Holds data for messages in current room

This is how data is stored once it is retrieved from the server, additionally, there are variables which represent the user’s current state including:

* **IsInRoom** – If User is currently in a room.
* **IsInChannel** – If user is currently in a channel.
* **IsRoomAdmin** – If User is Admin of current room.
* **IsRoomAssis** – If User is Assis of current room.
* **CurrentGroup** – Name of Current Group user is in.
* **CurrentChannel** – Name of Current Channel user is in.

The rest of the variables on the angular side are for inputs made by the user along with errors to be output to the user.

* 1. ***Angular Architecture***
     1. ***Components***

This project could have been divided into many components based on the required functionality, however, only two components were utilized. These components are Login and Dash.

**Login Component** simply takes a username and password as inputs and grants access to the website upon verification.

**Dash Component** contains all of the functionality for the chat system. This includes all of the post requests to the server and all of the data needed to run the system. Since the whole chat system is run from a single component, it is quite heavy.

* + 1. ***Services***

**Socket Service** is the only service which has been used and this service takes care of all socket functionality within the project.

* + 1. ***Models***

There is a single view which is manipulated by the functionality of the html script in conjunction with the model stored in the typescript file. The data being stored in the typescript file is being displayed throughout the view and aspects of the view are altered based on the data being stored in the model.

* **Fetch -** Fetching Data from the server side. These are generally used upon initialisation of the component.
* **Manipulate Users -** Used to manipulate the Users within the database including creation of new users, deletion of users and altering the role of a current user.
* **Manipulate Groups** – Used to manipulate groups within the database. Includes creation and deletion of groups along with granting and revoking access rights to groups.
* **Manipulate Channels** - As above, used to manipulate channels. This includes creating and deleting channels along with granting and revoking access rights to channels.
  1. ***Node Server Architecture***
     1. ***Modules***

My Node server has a total of 18 modules, each corresponding to a given route. The server is queried by the client side via http post requests and forms on the angular template are used to specify data. This data is sent through to the Node server which manipulates the database. The data is stored as a JSON file which is read by the server, altered if necessary and rewritten to the data file. Data is also extracted by the server and returned to the client side for display. Modules are as follows:

* **checkUser -** This module is used to check the user credentials in order to authenticate their login. The route checks a username against the database and if the username exists, the user is granted access.
* **fetchUser** - This module takes a username parameter and returns the user object which is stored in the database.
* **fetchRole** - This module takes a username parameter and returns that users role.
* **fetchUser** - This Module returns all data about users.
* **fetchChannel** - This module returns all data about groups.
* **newChannel** - This route takes a group name and a channel name and creates a new channel within the specified group. It also checks to see if the channel name already exists within the group and returns the updated data about groups.
* **newGroup** - Module takes a group name as input along with an optional input of group admin. The route creates a new group object with empty arrays for Assis, Users and Channels, if an admin is specified, the group is created with that admin receiving privilege over it. Returns the updated list of groups.
* **newUser** - Module take username, email and role input and creates a new user object based on the specifications. Returns an updated list of users.
* **newAdmin** - Module takes a user as input and adds that user as an admin to a group – assuming that user has admin privileges. Returns updated list of groups.
* **newAssis** - Module takes a user as input and adds that user as an Assis to the specified channel. If user is not an Assis, they are promoted to Assis as well as being added. Returns updated list of groups and updated user list.
* **destroyGroup** - Module takes a group name as input and removes that group from the database. Returns updated list of groups.
* **destroyUser** - Module takes a username as input and removes that user from the database. Returns updates list of users.
* **destroyChannel** - Module takes a channel name and a group name as input and removes that channel from the database. Returns updated list of groups.
* **promoteUser** - Module takes a username and role. The User is granted that role. Updated user data is returned.
* **inviteChannel** - Module takes a username, group and channel as input. Module loops through groups to find the group which contains the specified channel. User is added to this channel. Updated groups list is returned.
* **inviteGroup** - Module takes a username and a group name as input. If user is not already a member of this group (admin, assis or user), user is added to group list. Updated group list is returned.
* **revokeChannel** - Module takes a username, group and channel. Module finds the group which corresponds to channel and removes user from that channel. Updated group list is returned.
* **revokeGroup** - Module takes a username and a group. Module removes user from group and all channels within that group.
* **fetchMessages** – Module fetches all messages in the database Takes no parameters and returns all messages.
* **uploadFile** – Module uploads file to server and returns path to specified route
  + 1. ***Functions***

No specific functions have been user on the server side.

* + 1. ***Files***

There are a total of 20 files within the Routes folder. These are described above.

Besides this, there is a server.js file which holds the express app along with its dependencies and houses all of the routes. There is a listen.js file which specifies the port which the route runs through. There is also a package file and a node\_module folder which holds all of the dependencies and npm packages for the application. There are no global variables besides those which enable npm packages such as cors and body parser. There is a socket.js file which holds all of the socket functionality.

* 1. ***Rest API & Division of Labour***

The server acts as an API and interfaces with the angular client. The client sends http post requests to the server at port 3000 via http methods. The API interfaces with the JSON file which holds all of the data necessary for the system. Access to the database is via routes. The responsibility of the server side is essentially to take queries from the client side in the form of data. The routes hold functionality which processes the data based on requirements by the client and whole data objects are returned to the client for display and use. The data acts as a model in the standard model, view controller paradigm and the client and server share the role of controller. The server retrieves data from the data file and either returns the data directly to the client or it alters the data and writes it back to the file before returning it.

The Client takes input from the view. This input is then packaged appropriately to send to the server for processing. The server processes this data and returns an object – generally being the list of users, groups or in some cases, both. The client holds a trim function for groups and users. The trim users function takes a list of all users and removes the current user from that list. In this way, the current user is not displayed in forms. The current users data is retrieved separately and stored for use. The trim groups function takes a list of all groups from the server. It then removes the groups and channels within those groups which the user is not apart of. In this way, the data which is held by the user is only data which is relevant. Eg. A normal user has all groups which they do not have access to trimmed from the list of groups. As data is processed by the client side in this manner, the responsibilities of controller are shared between client and server.

Aside from this, the view which displays all of the data simultaneously act as a view, moderate security and input validation. As users are only able to input predetermined values into most fields, the inputs are very safe. Aside from this, only correct data will be revealed inside of those fields so it will not be possible for users to be granted access to incorrect groups or granted access privileges which were not intended.

Finally, Sockets have been utilised separately to broadcast data between clients which have been connected to it. This provides chat functionality in real-time between clients. There is a route which returns a list of all messages. This list of messages is then trimmed depending on which channel the user has joined. When a user leaves a channel, the **roomMessages** array is cleared and on joining a new channel, the array is populated once again by referring to the **messages** array which contains all messages. It is trimmed for the new channel. When a user sends a chat message, the message is sent via sockets to the server side and appended to the database. It is then emitted to the client. If the client receives the emitted message, that message is appended to both the **roomMessage** as well as the **messages** array. This allows it to be displayed and updates the **messages** array so that new messages may be referred back to upon entry to the same channel.

* 1. ***Client-Side Display Alterations***

The html client is running a mass of angular templating. The most prominent of this templating correlates with the role of the current user. This will alter the display of the entire page. The user is only able to access various components within the view if they have the correct privilege. This data is stored in the server and retrieved upon user login. When users roles are changed, the state is changed in the view and different components are displayed for each user.

The variables retrieved from the database on the server side are sent to the client side. This replaces the current data in the client and this information is reflected in the view. For example, when a new user is added to a group, the information is sent to the server, stored in the database and the new state of the database is sent back to the client. This is then reflected in the list of users. Furthermore, if that user logs in, they will be able to see that they have access to a new group and they will be able to access that group. Another more prominent example is one in which a user is promoted to a super user. When a normal user logs into the chat, they have simply a list of the groups which they have access to. A super user on the other hand, has access to a whole dashboard full of commands. If the user is promoted to a super user, their view will change to reflect this. They will now have access to the entire dashboard.

Finally, there is a variable called **IsInRoom** and **IsRoomAdmin**. These variables are simple Boolean expressions. The chat component will only show if the **IsInRoom** variable is set to true. This represents a state change on the client side. If the user has the **IsInRoom** variable along with the **IsRoomAdmin** variables set to true, they will be able to view the chat component along with the accompanying commands which comes with Admin privilege. The same principles apply when entering a channel.