Documentation for Software Frameworks Assignment Phase 1 (3813ICT)

Git

Git Layout

GitHub was used as the Version Control System (VCS) for this project, the link to the repository can be found below.

https://github.com/kevinpho1997/SoftwareFrameworksAssignment

At the root of the Git repository is the node modules for the Angular app and its node modules. In the Angular app is the src folder containing the Angular components, HTML and CSS. The Angular app folder also contains the node.js server that handles HTTP requests.

Version Control Methodology

Throughout development, I tried my best to only commit working code to ensure that the app compiles if a revert to a previous version. This was particularly with main, as I would work on main directly when making small changes but I would use branches for more complex tasks. The branches allow testing of new functionality without affecting main, which would then be merged into main when they are completed. This allows main to be always be capable of building when pulled from and is best practice when working with a team.

Testing

Server-side testing

First the node server must be started with in the /SoftFrameAssignment/server directory `nodemon server.js`

In the same directory run this command `npm run-script test`

Angular-side testing

To start server in /SoftFrameAssignment: `ng serve`

e2e testing

`npx cypress run` or `npx cypress open`

Component testing

`ng test`

Data Structures

The main data structures for this Angular project are:

- Users
- Groups
- Channels
- Permissions

Users is an array containing 0 or more user objects. Each user has the following attributes:

- `username`: string
- `email`: string
- `id`: number
- `role`: string

A User can have the roles: `default`, `super`, `gAdmin`, `gAssist`. These role respectively give the permissions of a default, super admin, group admin and group assistant.

Groups is an array containing 0 or more group objects. Each group has the following attributes:

- `id`: number
- `groupName`: string

Channels is an array containing 0 or more group objects. Each channel has the following attributes:

- `id`: number
- `channelName`: string

Users, Groups, Channels, etc are stored in a MondoDB database, where its collections are CRUD'ed through queries.

Permissions is an array of permission objects. Each permission object contains a `groupId` (a reference to the `id` of a group object) and an array of members that will have the permission level. Each member will have a `userId` (a reference to an id of a User) and an array of `channelIds` (a reference to an id of a Channel), which are the channels the user is subscribed to.

```
{groupId: number, members: [
    {userId: number, channelIds: [number]
]}
```

REST API

For the following routes, the parameters and return values are as stated in previously in the "data structures" section, unless explicitly stated

Auth route

Route overview	**This route checks if a user exists the database. If the user exists, it will call a login function to log the user in; returning an error if they do not.**
Route path	/auth
Method	POST

Route overview	**This route checks if a user exists the database. If the user exists, it will call a login function to log the user in; returning an error if they do not.**
Parameter/s	`username`
Return value/s	if (auth): `{valid: true, user: {username, email, id, role}}` if !(auth): `{valid: false, errors:{}}`
Description	This route takes a `username` in the http request. If there is a match in the database then the route returns `valid: true` and the user information. If there is no match, valid: false is returned.

Set permissions route

Route overview	**This route gives app permissions to a user depending on the value of user.role**
Route path	/setPermissions
Method	POST
Parameter/s	`user: {username, email, id, role}`
Return value/s	***See below code block for proper formatting
Description	This route takes a `user` and returns the `permissions` object where default users have no permissions. Relevant permissions are returned according to the user `roles`.

```
{

    rolePermissions: {

        // super admin

        canCreateUser: boolean,

        canRemoveUser: boolean,

        canUpgradeUserSuper: boolean,

        // group admin

        canCreateGroup: boolean,

        canCreateGroupChannel: boolean,

        canCreateChannelUser: boolean,

        canInviteCreateChannelUser: boolean,

        canUpdateChannel: boolean,
```

```
        canUpgradeUserGAssist: boolean,
        // group assist
        canAddRemoveChannelUser: boolean,
        canCreateGroupChannel: boolean
    }
}
```

Create User

Route overview	**Creates a new user if the username doesn't exist**
Route path	/user/create
Method	POST
Parameter/s	`{user: {username, email, id: undefined, role}}`
Return value/s	No user match: `users` user match: `error`
Description	The route reads the database, and if there is no `user` match then it will add the `user` to the `userInfo` collection and assign a `userId`. It writes `users` to the `users` collection for login and returns the new area of `users`.

Update User

Route overview	**Updates a User**
Route path	/user/:id
Method	POST
Parameter/s	Request parameters: `id` Request body: `{user: {username, email, id, role}}`
Return value/s	`users`
Description	A `userID` and a `user` object is received and `userInfo` collection is read. It matches the `userID` and `id` of the `user` object to a document in the database and inserts the `user` object values into the document. The `users` object also replaces the username in the `users` collection.

Delete User

Route overview	**Deletes a user**
Route path	/user/delete
Method	POST
Parameter/s	Request parameters: `userId`
Return value/s	`users`
Description	This route matches the received `id` with the `userID` stored in the mongo database . It then matches the `username` to a user in the `userInfo` collection and deletes the document. It then finds the user in the `user` collection that matches the one previously deleted, and deletes that document as well.

Get Users

Route overview	**Gets all users**
Route path	/users
Method	POST
Parameter/s	N/A
Return value/s	`users`
Description	onInit() of a page that requires a list of users, this function is called, which queries the `userInfo` collection and returns the data as an array.

Get User Channels, Groups and permissions

Route overview	**Gets Channels, Groups and permissions that belongs to the user**
Route path	/users/channels
Method	POST
Parameter/s	`{user: {username, email, id, role}}`
Return value/s	`userAccess`
Description	The route receives a user object and reads channels, groups and permissions from the collections and initialises `userAccess`. The route matches channels, groups and permissions with a user that has a `userid` matching the input parameter and pushes this data to `userAccess` and responds with this data.

Create Groups

Route overview	**Creates a new group**
Route path	/group/create
Method	POST
Parameter/s	`name: string`
Return value/s	`groupID: number`
Description	The route reads the `groups` collection, and if there is no `group` matched, then it will add the `group`'s name to the `groups` collection and assign a `groupId`. It returns `groupId`.

Delete Group

Route overview	**Deletes a group**
Route path	/group/delete
Method	POST
Parameter/s	`groupId: number`
Return value/s	N/A
Description	Uses `groupId` and deletes the matching document in the groups collection and permissions collection with the matching `groupId`.

Add User to Group

Route overview	**Adds user to group**
Route path	/group/addUser
Method	POST
Parameter/s	`userId: number[], groupID: number`
Return value/s	N/A
Description	Receives a `userId` and `groupId` and searches the `group` collection. If the `groupId` exists and the `userId` does not, then it adds the userId to the array.

Remove User from Group

Route overview	**Removes User from group**
Route path	/group/removeUser
Method	POST
Parameter/s	`userId: number, groupID: number`
Return value/s	N/A
Description	Receives a `userId` and `groupId` and searches the `group` collection`with that groupId. If the userId exists in that document, then it extract and splice the id from the`permissions`array and updates the document field with the updated`userId`s.

Create Channel

Route overview	**Creates a new channel**
Route path	/channel/create
Method	POST
Parameter/s	`{ name: string, groupId }`
Return value/s	`groupID: number`
Description	The route reads the `groups` collection and `channels` collection, and if there is a matching `group` and no matching channel name then it will create the `channel` with an id of the result of `countDocuments() channels` collection + 1. It also extracts the `channel` field from the group collection appends the new group name and inserts it back into the collection.

Delete Channel

Route overview	**Deletes a channel**
Route path	/channel/delete
Method	POST
Parameter/s	`channelId: number`
Return value/s	N/A

Route overview	**Deletes a channel**
Description	Uses `channelId` and reads the `channels` in the `groups`, `channels` and `permissions` collections with matching `groupIds`. It extracts the fields and rewrites them with updated values and rewrites them into the 3 collections.

Add User to Channel

Route overview	**Adds user to channel**
Route path	/channel/addUser
Method	POST
Parameter/s	`userId: number, groupID: number, channelID: number`
Return value/s	N/A
Description	Receives a `userId`, `groupId` and `channelID` and finds the document that matches `groupId` in the permissions collections. If the userId does not exist, then it adds the object containing `userId channelId[]` to the`permissions` collection.

Remove User from Channel

Route overview	**Removes user from channel**
Route path	/channel/removeUser
Method	POST
Parameter/s	`userId: number, groupID: number, channelID: number`
Return value/s	N/A
Description	Receives a `userId`, `groupId` and `channelID` and finds the document in the permissions collection that matches `userId` and `channelId`. It then extracts the `channelIds` field, removes the `userId` and inserts it back into the document.

Angular Architecture

Components

Login

A login form greets the user upon visiting the site prompting them to login. It is a form requiring a username and password inputs, which sends a HTTP POST request to /api/auth. If the login is valid (`valid = true`),

then it sets the `username` and `password` into `localStorage`. It also simutaneously calls the permissions service's `updateUser()` that will update components that is subscribed to `loggedInUser`.

Chat - Core

The core functionality of the program. It consists of a navigation bar and a `routerLink` to each component.

Chat - Site permissions

Action	Overview	Permission	Has permission
Create user	Can create users with Group Admin role.	canCreateUser	Super Admin
Remove User	Can remove all users	canRemoveUser	Super Admin
Upgrade permission to super	Can upgrade another user to Super admin and lower permissions	canUpgradeUserSuper	Super Admin
Create group	Can create groups	canCreateGroup	Super Admin, Group Admin
Create group channel	Can create channels (subgroups) within groups	canCreateGroupChannel	Super Admin, Group Admin
Invite + create channel user	Can create and or invite users to a channel	canInviteCreateChannelUser	Super Admin, Group Admin
Update channel	Can remove groups, channels, and users from channels	canUpdateChannel	Super Admin, Group Admin
Upgrade permission to Assist	Can allow a user to become a Group Assis of the group	canUpgradeUserGAssist	Super Admin, Group Admin

Action	Overview	Permission	Has permission
Add/remove users from group channels	Can add or remove users in the group from channels within the group	canAddRemoveGroupChannels	Super Admin, Group Admin, Group Assist
Create channel within group	Can create channels within the group	canCreateGroupChannel	Super Admin, Group Admin, Group Assist

Chat - Action Menu WIP

Chat - Get Chat History WIP

Groups WIP

Channels

The channels component is where users would chat to each other. `ngOnInit()` Initialises the socket `initIoConnection()` Creates the connection to the socket. Subscribes to the getMessage() observable and push and changes to the messages array, which contains all the messages emitted on this socket. `chat` Is called by the Send button to send a message. The function calles the `send()` function to send the message to the server. Resets the message content for the next message.

User Administration (UA)

UA - CRUD menu

The menu allowing users to perform create, read, update and delete on the main data structures. This menu and certain actions will only be available (and visible) to users who have the appropriate permissions (as show in the permission table above). These permissions are assigned `onInit()` to users by using `setUserRolePermissions` route to return the appropriate permissions based on the current user's role.

Create (register) user

Pressing the "register" button will bring a form that will allow a user to input `username` and `email`, which will be posted to the server to create a user.

Edit user WIP

Delete user

For super admins, pressing the "delete" button will delete the user from the database. The button and route will not be available for other users.

Guards

Auth (Login)

This guard is used to stop unauthorised users from accessing routes in which they do not have permissions for. It will give users an alert to say why they cannot access this route and redirect them to login if they need to authenticate themselves.

Services

Auth (Login)

`loginEvent()` Takes a `uname` and `pword` and posts it to the server to handle logins `logoutEvent()` Logs a user out of their account and removes their data from `localStorage`

User Administration (CRUD)

`registerUser()` Takes a `User` object and posts the data to the server for the creation of a new user. `getAllUsers()`s Returns all registered users on the website. `deleteUser()` Takes a `userId` and posts it to the server to handle deletion of a user.

Socket Service

`initSocket()` Is called in onInit() when the socket service is needed in the component, where it initialises a socket `send()` Sends the message to the socket with the emit() method `getMessage()` Observes changes to the socket when emitting a `message`, if a message is observed, then it completes the next() function