3813ICT - Assignment

Below is all of the information describing the technicalities of as well as the development process of the 3813ICT Software Frameworks Assignment.

Documentation - Phase 2

Link to repo

Git Layout

The development for this project still continued to use only the single branch for development. Using more than one as just a single developer is unneccessary overhead that just serves to clog up an already pretty clunky interface, so the decision was made to just develop on main.

The file structure for this project is broken up into 3 parts; `Angular`, `REST API`, and the `MongoDB` database. These are represented in the Git repository under the files `Assignment-Angular`, `server/RestAPI`, and `server/MongoDB` respectively. These were kept seperate so they could each be run as their own service, with all of the relevant files clumped together in a way that is intuitive and makes sense.

As the project was extremely rushed and late due to unforseen hurdles in life, there are very few commits representing the progress for phase 2. This is because it was all done in such a short time period, that the need for saving the state of working features would have been awkward. During the rushed development, multiple features were being worked on at a time so any one push would always have something in a unfinished state.

GitHub was chosen as the choice for version control as I have a lot of experience both in and out of uni with it and it's intuitive design makes me comfortable with my choice.

Rest API

Authentication Routes

Authentication routes, compared to the admin routes, generally do not require high levels of permissions and tend to only require the confirmation that you are who you say you are.

```
"/login",
"/me",
"/display/:field",
```

```
"/id",
"/pfp/:userId"
```

Route	`/login`
Method	`POST`
Parameters	`{username: string, password: string}`
Returns	Valid User : `{statusCode:200, data: {token: <UUID>}}` Invalid User : `{statusCode:401)`
Purpose	This route is used to confirm the identity of a user when they log into the web application.

Route	`/me`
Method	`GET`
Parameters	None. Uses auth token
Returns	`id, username, email, and role` of logged in user
Purpose	This route serves to populate the current user data that is required by some of the webpages, and converts a UUID access token into an object.

Route	`/display/groups`
Method	`GET`
Parameters	None. Uses auth token
Returns	Array of group objects containing only the id, name, channels, and group assistants.
Purpose	This route is called when generating the main chat view for the screen, and pulls all of the groups that the user is present in and can see on the dashboard.

Route	`/display/channels`
Method	`GET`
Parameters	None. Uses auth token
Returns	Array of channel objects containing only the id, name, and message history.
Purpose	This route is used to get all of the channels that the currently authenticated user can read and chat in.

Route	`/id`

Method	`GET`
Parameters	`user UUID passed via query param`
Returns	Returns the id, username, and profile picture of the requested user.
Purpose	This route is used to identify other users, so that clients chatting can generate the profile pictures and the user-friendly usernames of eachother.

Route	`/pfp/:userId`
Method	`POST`
Parameters	`userId via URL path`
Returns	Returns status code 200 if the operation was successful
Purpose	A post request is sent to this endpoint by a user to change the image on their profile..

Admin Routes

Admin routes require the highest levels of permissions, and are concerned with manupulating in great detail the full range of data stored in the system. The return results of each of these function vary depending on the level of authentication a user has, and may not always return a result of unauthorised.

```
"/users*",
"/groups*",
"/channels*"
```

Users

Route	`/users/:userId`
Method	`GET`
Parameters	`userId via URL path`
Returns	Returns status code 200 if the operation was successful, as well as an object containing the requested user's data.
Purpose	Used to allow admins to edit specific information of each user on the system.

Route	`/users/`
Method	`GET`
Parameters	None

Returns	Returns status code 200, as well as the list of all users that the authenticated user has admin permissions over.
Purpose	Used to show all of the users in the system on the admin settings screen.

Route	`/users`
Method	`POST`
Parameters	`username, email, password`
Returns	Returns status code 201 if the operation was successful, as well as the Id for the newly created user. Will fail if duplicate username exists.
Purpose	Route used to create a new user with the given parameters.

Route	`/users/:userId`
Method	`PATCH`

Parameters	`add: {<field to be added to>:<data to add to field>},`
	`remove: {<field to be removed from>:<data to be removed from field>},`
	`update: {<field to update>:<data to replace>}`

Returns	Returns status code 200 if the operation was successful
Purpose	This is used in order to be able to change every detail about a user in the system.

Route	`/users/:userId`
Method	`DELETE`
Parameters	`userId via URL path`
Returns	Returns status code 200 if the operation was successful, as well as a confirmation message.
Purpose	An admin makes a post request to this endpoint to permanently remove a user from the system.

Groups

Route	`/groups/`
Method	`GET`
Parameters	None

Returns	Returns status code 200, as well a list of every group that the authenticated user has high enough permissions to manage.
Purpose	Provides the data for the group list on the admin settings page.

Route	`/groups`
Method	`POST`
Parameters	`name`
Returns	Returns status code 201 if the operation was successful, as well as the Id for the newly created group.
Purpose	Used to create a new group in the database with the provided name.

Route	`/groups/:groupId`
Method	`PATCH`

Parameters	`add: {<field to be added to>:<data to add to field>},`
	`remove: {<field to be removed from>:<data to be removed from field>},`
	`update: {<field to update>:<data to replace>}`

Returns	Returns status code 200 if the operation was successful
Purpose	This is used in order to make fine-tuned changes to a group.

Route	`/groups/:groupId`
Method	`DELETE`
Parameters	`groupId via URL path`
Returns	Returns status code 200 if the operation was successful, as well as a confirmation message.
Purpose	An admin makes a post request to this endpoint to permanently remove a group from the system.

Channels

Route	`/channels/`
Method	`GET`
Parameters	None
Returns	Returns status code 200, and a list of all channels that the user has permission to modify.

Purpose	Used to show all of the users in the system on the admin settings screen.

Route	`/channels`
Method	`POST`
Parameters	`name`
Returns	Returns status code 201 if the operation was successful, as well as the Id for the newly created channel.
Purpose	Route used to create a new channel with the provided name.

Route	`/channels/:channelId`
Method	`PATCH`

Parameters	`add: {<field to be added to>:<data to add to field>},`
	`remove: {<field to be removed from>:<data to be removed from field>},`
	`update: {<field to update>:<data to replace>}`

Returns	Returns status code 200 if the operation was successful
Purpose	This is used in order to make fine-tuned changes to a channel object.

Route	`/channels/:channelId`
Method	`DELETE`
Parameters	`channelId via URL path`
Returns	Returns status code 200 if the operation was successful, as well as a confirmation message.
Purpose	An admin makes a post request to this endpoint to permanently remove a channel from the system.

Authorization

The authorization system implemented in the code uses a custom made token system, which holds the identity of a user and represents them in a way that is designed to be more tamper proof than simply storing the user object. Each time a user logs in, if there is already a valid, non expired token in their name, that is grabbed. Otherwise, a new token is generated with a 1 hour expiry which represents the users' identity and permission. The tokens are self validating and self renewing.

Data Structures

The data structures of the core objects of the project remained very similar to their design for phase 1, though with changes being made along the way that suited the natural progression of the design process. The sections in the phase 1 design document have been adjusted accordingly to the new standard formats. The most noticeable improvements is the changing of the names from - for example - `groupId` to `id`, which greatly improves readability and clarity when it comes to handling the data.

Four new helper data structures were created that simplified some of the more tedious tasks of the development. These are:

- `socketUtil.js`: This object was created in order to assist with preserving states of the server socket, while still allowing access to the main socket connection in the files that need it. It consists of an initial server connection function, a function that returns the currently active connection, and a final function that closes the connection. This, like the mongoUtil function, is node.js's solution to a Singleton design pattern.
- `error.js`: This file is as close to JavaScript can really come to a struct. It uses practically named object keys that represent both error codes and the associated message. This saves development work and makes more a much more consistent and user friendly interface.
- `authFunc.js`: This file contains all of the functions that handle user authentication. They were extracted and put into a shared folder because almost every endpoint needed to use them, so it was incredibly convenient.
- `mongoUtil.js`: Much like the socketUtil file, this serves as a Singleton manager for the mongo database connection, ensuring that a single connection is maintained for the duration of the programs runtime, which removes any possible data merging conflicts as well as clears up the mongo network and is more efficient.

Server & Client task distribution

The intended design principle of the second phase of this project was a strong focus on trying to stick to a MVC design pattern. Failing to do this during phase 1 of the assignment caused many headaches when attempting to remove the old and overly specific code that was embedded in the view of the web page and replace it with a new one.

The new system tries to have as much of the heavy lifting as possible handled by the REST API, including user permission filtering as well as micro and macro authentication. This frees up the model controlling the web page to be able to make clear and straightforward requests to endpoint on the API, that wont require a colossal rewrite if the inner working of some of the mechanisms change. The nature of angular means that the view is tied very closely to the model, while also being distinct enough that they dont step on eachothers toes. The Angular html of the project calls functions defined in the typescript component that keeps the front end uncluttered and only full of the information that is needed.

Angular Architechure

There has been very little change in the projects architecture since the shift from phase 1. This was because in terms of building blocks each piece was already where I wanted it, the largest time sinks in the transition was the extraction of all of the legacy code and function from inside of the angular components.

There are 3 created components for this assignment, those being `Settings`, `Groups`, and `Login`. The function and purpose of each of these components has already been described in detail in phase 1 of the report down below.

Server/Client Data Flow

In all of the cases except one, when a change is made to backend data and is reflect on the front end, this is due to the request being made from the front end immediately being sent upon its creation, and the relevant database state after these changes being returned. This means that the synchronicity between the client and the server is straight forward and as expected.

The one exception to this case is the messaging system. There are specific messages - i.e. ones from the system that indicate when a user is joining/ has left a channel - that are only updated on this local client. These are lost when the message history is refreshed again. This decision was made with performance and user friendliness in mind - the constant updating of people/leaving joining a channel creates unneccesarry clutter in the chat log, as well as unneccessary server traffic that could potentially slow down real attempts at communication.

When a message is receieved from the socket system, is it added to the currently opened channels history of messages. This does not require a request to the REST API backend, which saved processing time and slots as there could potentially be a lot of messages coming through at once. The message is is properly synced with the server once a user leaves/joins a new channel. This triggers a HTTP request which gets the current message history of that specific channel.

Documentation - Phase 1

Below is the documentation for phase 1 of the assignment. It also contains artefacts of information that may not be implemented yet, and may be inaccurate. This is because this documentation is written as a plan for the future development to be done in phase 2, and represents the end goal rather than the current state.

Terminology in this report

- `The app/application` - This referse specifically to the angular part of the assignment project, and does not include the node server that hosts the back end.

- `The server` - This refers to the backend server that is not used to handle UI, and containes routes. (Implemented only in phase 2)
- `Group` - This is a group as per the assignment specification, in which users can be assigned and view channels within. Can also have a Group Assis.
- `Channel` - A channel is where users can type their messages and view the chat history, and these channels are assigned to groups, and only specific users have access to any given channel.
- `User` - A registered user within the data of the project. Can read/write messages, and have permissions which may grant the ability to modify the contents of the database.

Git

The GitHub repository for the assignment is hosted here.

Layout

All of the work that has been completed for Phase 1 is hosted on the Main branch, where development will continue. Due to the straightforward nature of the work, the implementation of multiple branches into my workflow would only create unneccessary complications throughout the development process. The benefits of a multi-branch system are clear when a user is working in a group, and must confirm the viability of the changes that are being attempted, but as I am working alone this is an overkill solution and thus wont be implemented.

Commit Frequency

A commit would be made to the repository when the development of a block functionality was added, and the ability for a rollback was desirable. This saved time with constant commits to the main repository, and also helps to keep the commit history clean. A commit would also be made at the end of each session where development took place, in order to insure that no progress was lost and that I was working on the most up to date version of my application.

For phase 1 of the assignment, there are 4 major commits.

Commit Name	Description	Link
Initial commit	This commit made no functional changes, and simply served as the creation of a new GitHub repository	Link
Initial commit and basic UI design	This commit contained major changes to the layout of the git repository, and contained all of the angular code initialisation, as well as a basic layout for the UI of the future project. The project contained no functionality at this stage.	Link
Login functionality + dynamic content	The changes in this commit brought life to the featurless UI added in the previous commit, and was where the implementation of dynamically setting the page content based on variables in the code was made. At this state of development, information was taken directly from local JSON files rather than as serialised strings stored in code or from a Database. With this, a user was able to log in, see all of the channels and groups that they had been assigned to, and view the chat history of that channel/group combination.	Link

Commit Name	Description	Link
README/TODO + Settings functionality	This commit added the settings page to the repository, which is the component in the application that handles the creation/modification/deletion of users, groups, and channels. This page only shows users the relevant forms and abilities that their permissions allow, and the button to access this page is not visible if they aren't permitted to do anything. The commit also initialised this README file with the TODO at the bottom of the page.	Link

Data Structures

This project has a requirement that uses data extracted from a database (eventually, in phase 2) to dynamically change how a user can interact with the site. There are 4 different types of data structures used in this project, which collectively hold all of the information required to achieve the functionality defined in the assignment specifications.

Users

The users data structure is an array of user objects, which contains information about all of the registered users and the permissions they have. A user object is of a standard format, which an example of is given below:

```
{
    "id": <UUID>,
    "username": <String>,
    "email": <String>,
    "password": <String>,
    "role": Array<Number 0-2>
}
```

Values

- `id`: Unique identifier number that represents the user.
- `username`: User defined name of their account that is visible to other users, also unique.
- `email`: The email assigned to this user. Not visible to other users without permissions.
- `password`: The password of this user. Used when logging in to confirm the identity that the client is claiming.
- `role`: A number from 0-2 representing the permissions that the user has in this application. 0 meaning none, 1 meaning Group Admin, and 2 meaning Super Admin. Identification of group assis is defined in the group data object.

Groups

The groups data structure is an array of group objects, containing all of the linked channels to a group, as well as other information.

```
{
    "_id": <UUID>,
    "name": <String>,
    "channels": Array<UUID>,
    "users": Array<UUID>,
    "groupAssis": Array<UUID>,
}
```

Values

- `id`: Unique identifier number for this group.
- `name`: The user-friendly name of this group that is displayed to end users.
- `channels`: An array of UUIDs of channels that are assigned to this group.
- `users`: An array of UUIDs of users that have been added to this group.
- `groupAssis`: An array of UUIDs of users that have been granted permission to act as Group Assistant of this group, being able to add new channels and assign/unassign users to channels.

Channels

The channels array contains channel objects, containing information about the chat history of a given channel, as well as the users who are permitted to access that channel.

```
{
    "_id": <UUID>,
    "name": <String>,
    "userAccess": Array<UUID>,
    "messageHistory": Array<Message>
}
```

Values

- `_id`: The unique identifier of this channel.
- `name`: The user-friendly name of this channel that will be shown in the UI.
- `userAccess`: The list of users that have read/write access to this channel.
- `messageHistory`: An array of Message objects that contains information about every message that has been sent to this channel.

Message

A message object contains identification information for a given message, as well as the content and the user that posted that message.

```
{
    "_id": <UUID>,
    "author": <UUID>,
    "content": <String>,
```

```
    "time": <Int>
}
```

Values

- `_id`: The unique identifier of this message.
- `author`: The unique identifier of the author of this message.
- `content`: A string representing the content that was sent with this message.
- `time`: A 64-bit integer that represents the number of milliseconds since the epoch that the message was sent.

Angular

Services

When the Node.js server interaction is implemented, there will be a service created that will handle the interaction between the UI and the requests to the server. This will create a disconnect from the view and the controller, and greatly improve the simplicity of this implementation. Due to the requirement stating that data be hosted in local storage for phase 1, the implementation of a Node.js server was deemed unneccesary.

Components

There are only 4 components used in the creation of this angular project. Each page is represented by a component, so there is one for groups, settings, login, and the home page. Each component has been styled using the css and js provided by the Bootstrap library.

Home

This is the default route of the program, reach by directing to `/`. This hosts the router module and is a container for the rest of the pages. The code for this page handles redirects to `/login` when a user is not logged in, and to `/groups` when they have successfully authenticated. This component also extracts the json data from the local files when the page is first loaded. This will be changed in phase 2 of the assignment and replaced with functionality in the other components.

Login

This page handles the login for the user, and can be access at the `/login` route. It consists of 2 text input fields, for username and password, as well as a button to submit login. This page also contains a hidden dialog box that displays an error to the user if the credentials that were used for login are invalid. On a successful login, this page will redirect to `/groups`, where the user can now see the assigned groups and channels, as well as chat history.

Settings

This page has complex logic behind the scenes to control what screen elements are displayed based on the permissions of the user. It can be accessed at the `/settings` route. The page utilises modals for controlling

how components are edited, and the entering of the data that is being changed. Groups, channels, and users can be edited, created, and deleted here if the viewer has the required permissions.

The modals used for editing the components of the data are displayed as popups that cover the screen when the relevant element on the page is selected. The contents of these modals vary depending on the function and the permissions of the user. The page also implements an accordion display, with an accordion item for each group that displayed the channels and members assigned to each one.

Create user modal
This modal contains 3 text boxes for the input of information regarding a user. This includes the `username`, `email`, and the `password`. The modal also has 2 buttons for control of the menu, `close` and `save changes`, which will close the modal without making any change, and create a new user modal respectively.

This modal is only visible for users with the `group_admin` permission or above.

Edit user modal
This modal displays a checkbox list that represents all of the groups available, which can be used to assign a user to new groups or remove them from ones that they are already in. If the user is a `super_admin`, there is also a checkbox menu that will allow a user to be reassigned to `super_admin`, `group_admin`, or `no_permissions`. A user with the `super_admin` role can also use this modal to delete a user.

This modal is only visible for users with the `group_admin` permission or above.

Create channel modal
This modal contains one text box that allows an admin to enter a channel name, and save the creation of a new channel. The button to activate this is contained under the list of channels under a group accordion item, and it's creation will assign this channel to this group.

This modal is only visible for users with the `group_assis` permission or above.

Edit channel modal
This modal contains only a single text box, allowing an admin to edit the name of this channel. If the admin is a `group_admin` or above, they are also presented with a button that will delete this channel, and remove it's message history.

This modal is only visible for users with the `group_assis` permission or above.

Edit user in group modal
This modal is accessed by selecting a user under a group accordion item. This will allow an admin to select the channels that a user in this group can see and type in via the selection of a list of checkboxes. If a user is a `group_admin` or higher, this modal will also contain a checkbox that the admin can use to make a user a `group_assis` of this group.

This modal is only visible for users with the `group_assis` permission or above.

Create group modal
This modal allows an admin to create a new group, which can then have channels and users assigned to it. This modal contains a single text box for inputting the name of the new group that is being created, and a save button to confirm the creation.

This modal is only visible for users with the `group_admin` permission or above.

Group settings accordion
This is an accordion UI item that is a list of labels representing each group, that can be expanded. Expanding one of these items will present the channels and the users in this group. A group is only visible is an admin is at least `group_assis` of that group. If a user is `group_admin` or higher, they can see and access all groups. If a user is `super_admin` or above, they will also see a `Delete Group` button.

This UI element is only visible for users with the `group_assis` permission or above.

Groups

This page is where the user will mainly interact, being able to send and view messages, as well as the groups/channels that they have been assigned to in the admin menu. It can be accessed via the `/groups` route. There is a navigation bar up the top, which contains a list of all of the groups that a user can access. When selecting any of these groups, a list of the left side of the screen will update with all of the channels in this group that the user has been given permission to read/write in. In the bottom right of the screen - and taking up most of the screen real estate - is the chat window. This shows the chat history, as well as the authors of each chat message. At the bottom of this window is an input box with a button next to it that will allow a user to type and send a message in phase 2. In the top right is a logout button, and if the user has any kind of admin priviliges, a settings button will also be displayed where they can see their authenticated options.

Node

As of phase 1 of the assignment, there is no interaction with the node js server and it has not yet been implemented. I had a look through all of the assignment specifications and this was not listed as a requirement, so I took priority in working on the UI and dynamic elements of the Angular site of it.

As per the dividing of responsibilities of each part, the UI will only act as a vehicle for the user, and all of the authentication and data handling will be done by the back end server. This will be accomplished through API routes using a REST interface, where the UI can make requests using an authenticated users information to ensure that the data being changed and requested is allowed. This will add a layer of security and simplicity when it comes to keeping the two seperate.

The UI will send requests to an `/auth` endpoint of the node server, with user login information inside of the body of that request. The server can then send a response back to the front end either allowing or denying the login request, and providing the UI with a token to use as proof of verification for other operations.

Progress Tracking

TODO (29/08/2022)

- ~~Add ability to delete channels/groups/users~~ (Done: 07/09/2022)
- ~~Add ability to make new channels~~ (Done: 07/09/2022)
- ~~Add navbar to all pages of application~~ (Done: 30/08/2022)

- Documentation
- Create node js server

DONE (29/08/2022)

- Chat history
- Correct group/channel display for UI
- Log out button
- Complete log-in functionality
- Hidden settings button if insufficient perms
- Admin screen with permission checks.
    - Super Admin can:
        - Change roles of user (super_admin,group_admin)
        - && everything below
    - Group Admin can:
        - Assign any user to any group
        - Assign any user in a group to any channel
        - Create new channel
        - Create new Group
        - Create new User
        - Create new Group
        - Add users to channel/group
        - Can make a user group assis
        - && everything below
    - Group Assis can:
        - Add/remove users from a channel