Documentation for 2811ICT Web Programming

Zaymon Foulds-Cook s5017391
Griffith University School of Information Communication Technology

`http://github.com/ZaymonFC/WebProgramming-Assignment`

I. Git Usage

A. Repository Organization

The git structure of the git repository is simple, the top level directories of the angular project are at the top level of the repository along with the server directory and the documentation directory. The reduction in nesting resulting from not having the Angular app in it's own directory is ideal.

B. Git Practices

Throughout the development process git has been used heavily. The first section of the assignment consists of over 33 commits. A commit has been added to the repository either after each logical increment in development or when fixes are completed for bugs or unfinished features. Each commit has been used as a wrapper for a collection of related changes and the version history of the project reflects the progression and development of the system. Good descriptive commit messages have always been used in order to summarize the changes contained within. In regards to workflow a process did not have to be standardized due to the size of the developer team (1 member).

II. Data Structures

Various types have been used to represent different entities in the system. The types are:

- **User** | Data type for a user entity.
  - id: string
  - username: string
  - rank: string
  - email: string
- **Group** | Data type for a group entity. Used on the front-end for the group detail view. Contains nested collections of users and channel summaries.
  - name: string
  - id: string
  - description: string
  - users: User: List<User>
  - channels: List<ChannelSummary>
- **Group Summary** | Lighter weight type for group entity that does not require nesting of associated user objects.
  - name: string
  - id: string
  - description: string
  - users: list<string> (user ids)
  - channels: list<string> (channel ids)
- **Channel** | Type for the channel entity, used in the channel detail view. Contains nested users and a reference to the group it belongs to.
  - id: string
  - name: string
  - users: list<User>
  - groupId: string
- **Channel Summary** | Lighter weight type for channel entity that does not require nesting of associated user objects.
  - id: string
  - groupId: string (id of group)
  - users: list<string> (user ids)

III. Separation of Responsibilities - REST API and Client

A. API Responsibilities

The responsibilities of the rest API are the concerns of data-access and persistence. When entities are created, modified or deleted on the client these changes must persist to be useful. For the client to be useful it must be able to present the user with data (hence the name presentation layer). To persist changes or fetch data the client must interact with the API. These interactions are managed by sending requests to controlled endpoints (routes) where the behavior of the server is unique to the action each route is responsible for. In this project the API exposes routes to fetch, update, create and delete all entities listed in section 2. The API makes use of a functional data layer to perform queries, search, select, delete and manage foreign keys on entities stored in separate

JSON files. The functional data layer also manages keys and makes use of UUIDV4 for primary keys. The data layer can enforce unique fields.

B. Client Responsibilities

The responsibilities of the angular client are to present data to the user and to allow users to interact with the application in a real time and visual way. In this assignment the client is responsible for presenting the data as well as allowing the users to manage entities such as groups, channels and other users. The client in this assignment also has a granular permission system implemented as simple action guards to hide sets of functionality from users of different ranks.

IV. Routes

- **GET /login/:username** | A simple route which returns a user object if the user is found in the data otherwise returning null..
  Parameters:
  - UserID: string

  Outputs:
  - User Object
- **GET /user** | Route to return the top 100 users (for user lists).
  Parameters:
  - None

  Outputs:
  - list<User>
- **GET /user/:id** | Route to find and return a specific user.
  Parameters:
  - UserID: string

  Outputs:
  - User Object
- **POST /user** | Route to create a new User entity.
  Parameters:
  - username: string
  - email: string
  - rank: string

  Outputs:
  - User Object
- **PATCH /user/:id** | Route to update the contents of some user in the database.
  Parameters:
  - UserId: string
  - Object of changes expressed as key value pairs (key attribute name, value is the new value).

  Outputs:
  - VOID (respond Status OK)
- **DELETE /user/:id** | Route to delete a user and clear all reference to that user in the group and channel collections.
  Parameters:
  - UserId: string

  Outputs:
  - VOID (respond Status OK)
- **GET /otherUsers/:id** | Route to fetch users that are not in a certain group (used for finding lists of users to add to groups).
  Parameters:
  - GroupId: string

  Outputs:
  - list<User>
- **GET /group** | Route to get the top 100 groups (used for listing groups).
  Parameters:
  - None

  Outputs:
  - list<GroupSummary>
- **GET /group/:id** | Route to find a specific Group, then queries the database and nests all User's associated with the group as nested objects. Also nests ChannelSummary's associated with the group.
  Parameters:
  - GroupId: string

  Outputs:
  - list<Group>
- **POST /group** | Route to create a Group entity and store it in the database. Sanitizes input and creates a group with a scheme. Either returns 'non-unique' or the newly created Group object.
  Parameters:
  - name: string
  - description: string

  Outputs:
  - Group: Group
- **DELETE /group/:id** | Route to delete a group from the database. (Also deletes all associated channels).
  Parameters:
  - groupId: string

  Outputs:

– Status OK

- **POST /channel** | Route to create a new Channel. Sanitizes input information and then saves a new channel object to the database (includes reference to parent group). Updates group table to include a reference to the Channel. Returns newly created Channel.
  Parameters:
  – name: string
  – groupId: string
  Outputs:
  – channel: Channel
- **DELETE /channel/:id** | Route to delete a channel and the parent groups reference to it.
  Parameters:
  – channelId: string
  Outputs:
  – Status OK
- **PATCH /group/:id/user** | Route to add or remove users from group.
  Parameters:
  – methodName: string (add or remove user)
  – groupId: string
  – userId: string
  Outputs:
  – Status OK
- **GET /channel/:id** | Route to get a channel with nested users.
  Parameters:
  – channelId: string
  Outputs:
  – channel: Channel
- **PATCH /channel** | Route to add and remove users from channels
  Parameters:
  – method: string
  – userId: string
  – channelId: string
  Outputs:
  – Status OK

V. Angular

A. Routes

- **‘’**, redirects to component HomeComponent
- **‘chat’**, redirects to component ChatComponent
- **‘404’**, redirects to component NotFoundComponent
- **‘dashboard’**, redirects to component DashboardComponent
- **‘group/:id’**, redirects to component GroupComponent with a group ID as input
- **‘channel/:id’**, redirects to component ChannelComponent with a channel ID as input
- **‘**’**, redirectTo component ’404’ (not found)

B. Components

Various components were created throughout development, some components form hierarchies of parent and child components. The app component is the top level component in the application. The app component in this application contains a simple template which include a router outlet and the menu bar. When the application navigates to other routes the router outlet will be populated with different components.

- **Menu** | Menu component for logging out and navigation.
- **Home** | The home component contains the log in form for logging into the application.
- **Chat** | A simple component containing a basic sockets chat.
- **Dashboard** | Component for managing and viewing users and groups.
  Nested Components
  - **dashboard-group** | Child list component which handles entering and deleting groups.
  - **dashboard-user** | Child list component which handles deleting, listing, promoting and demoting users.
- **Group** | Detail view for a group that contains functions and functionality to create and remove channels, add and remove users from the group.
- **Channel** | Detail view for a channel that contains functions and functionality to add and remove users from the group.
- **NotFound** | Component to display if the route is not found.
- **Counter** | Simple component that displays a badge with the value of the number passed into the parameter.

C. Services

- **User Service**
  Responsibilities
  - Manages user for current session
  - Creates users
  - Gets users
  - Finds user
  - Deletes users
  - Promotes and demotes users to different ranks
- **Channel Service**
  Responsibilities
  - Get channels
  - Remove Channels
  - Add user to channel
  - Remove user from channel
- **Group Service**
  Responsibilities
  - Get groups
  - Delete Groups
  - Find Groups
  - Add user to group
  - Remove user from group
- **Permission Service**
  Responsibilities
  - Defines granular permissions
  - Creates permission hierarchies based on role
  - Can check whether the current user has a certain permission in their associated permission hierarchy

D. Modules

The main module used during this assignment is the routing module, which defines the virtual routes for the application.