SoftwareFrameworksAssignment

Text/Video System Documentation

Introduction

This document outlines the design, structure, architecture of a real-time text/video chat application built using the MEAN stack. It also includes the purpose and use cases of components such as login and permission features depending on user authority, project's version control, data structures utilised to organise backend data and detailed API reference. The system allows users to be allocated into groups where there exists some channels where they will be able to communicate within. There is a three-tiered permission system: User, Group Admin and Super Admin.

Git Repository and Version Control

Repository Structure

The Git repository is structured into one main directory named "text-chat'. This directory includes all the frontend code (Entire Angular Application) but it also contains another directory named "server" which contains the Node.js and Express.js server application to keep the frontend and backend codebases separate and organized:

- `/text-chat`: Contains the entire Angular application including the backend server.
- `/text-chat/server`: Contains the Node.js and Express.js server application. The `node_modules` directories for both are included in the `.gitignore` file to prevent them from being committed to the repository. Additionally, there is .env configuration file that includes the MONGODB_URI which is included in the `.gitignore` file to prevent the MongoDB username and password from being leaked.

Branching Strategy

The project followed a seperated environment inspired branching strategy. A `main` branch holds the stable and working project, while the `develop` branch holds new changes and features to be integrated into `main` once fully functional. For instance, when developing the `groups`, `channels` and `messages` component, the `groups` component was developed first and then merged into main once fully functional and once the `channels` component was synchronised and functional with the `groups` component it was then merged into main and so forth (as each component was dependent on another).

Commit Frequency

Commits were made frequently, generally each commit holds a small modularised piece of work or feature such as developing the API endpoint for logging in a user. Each commit message succinctly describes the work completed.

Data Structures

The application's data is modeled and defined on the client side using TypeScript interfaces, where the server side follows these models when passing or accepting data to ensure type safety and consistency. The primary data models are as follows:

User

The `User` model represents a registered user in the system. It contains their credentials, permissions, and relationships to groups.

```
export interface User {
    id: number;
    username: string;
    email: string;
    profilePicture: string | null;
    roles: string[]; // e.g., ["User", "GroupAdmin"]
    password: string; // This is hashed in a real-world scenario but not
in this project
    groups: Group[];
    adminGroups: Group[];
    requests: Request[]; // Holds the requests from users that desire to
enter a group
    allowedChannels: Channel[];
    reports: Report[]; // Holds the reports made from other Group Admins
made about a User within a group
}

// A smaller version for displaying users within a group
export interface GroupUser {
    username: string;
    profilePicture: string | null;
    role: string; // The user's role specifically within that group
}

// A secure version for client-side storage, omitting the password
// An object of this model is stored in the browser's local storage to
check if a user is logged in
export interface SessionStorageUser {
    id: number;
    username: string;
    email: string;
    profilePicture: string | null;
    roles: string[];
    groups: Group[];
    adminGroups: Group[];
    requests: Request[];
    allowedChannels: Channel[];
    reports: Report[];
}
```

The `Group` model represents an established group in the system. It contains the users that are allowed in the group, the channels and the name of the group.

```
export interface Group {
    id: number;
    name: string;
    users: GroupUser[];
    channels: Channel[];
}
```

The `Channel` model represents a specific chat room within a group.

```
export interface Channel {
    groupId: number;
    id: number;
    name: string;
    messages: Message[];
    allowedUsers: string[];
}
```

The `Message` model represents a single chat message within a channel (It stores the group Id and channel Id it is within for easy access to its group or channel data).

```
export interface Message {
    username: string;
    profilePicture: string | null;
    message: string | null;
    image: string | null;
    groupId: number;
    channelId: number;
}
```

The `Request` model represents a user with the role 'User' that has requested to join a group.

```
export interface Request {
    username: string;
    profilePicture: string | null;
    groupId: number;
    group: string;
}
```

The `Report` model represents a report filed by a Group Admin on User within a specific group.

```
export interface Report {
    usernameReported: string;
    userProfilePicture: string | null;
    adminReport: string;
    adminProfilePicture: string | null;
    groupId: number;
}
```

The `State` model represents the current group and channel that the user is accessing at the moment and what is being displayed on their screen. An object of this model is then stored in the browser's local storage.

```
export interface State {
    currentGroup: Group | null;
    currentChannel: Channel | null;
}
```

Angular Architecture

Components

The frontend is constructed with a component-based architecture where each component is responsible for a specific view which fufills user requirements for the project.

- `Home.component`: Displays a home page for users who have not logged in to gain information about the web application.
- `Groups.component`: Displays all the groups that logged-in user have access to and also displays all the groups that are within the database for the user to request access to if they do not have access to it currently. Additionally, administrative lists for requests to groups and reports for users are shown for users with the respective roles and permissions.
- `Channel.component`: Displays all the channels that the user has access to within the user specified group. For admin users, it also displays the list of users within group and provides them the functionality to add/report/ban/delete users to the given group and to add/remove users from specified channels within the group.
- `Message.component`: Displays a text chat for the user given their current group and channel that they have specified. This text chat allows them to send text messages and images and loads the previous messages of the given channel and group. It also includes a video chat button which allows user to enter a video chat with other users within the same group and channel. This will allow the user to call another user who is active in the video chat.
- `Login.component`: Displays a form that allows users to enter their user credentials which include their email, username and password to access the web application.
- `Register.component`: Displays a form that only allows users with the role 'SuperAdmin' to create a new user given an email, username and password.
- `Account.component`: Displays the user's current credentials such as email, username, profile picture and the groups that they are within. It also provides them the functionality to logout, delete

their account and create new accounts (redirect to the register component) if the user is a 'SuperAdmin'.
- `App.component`: Displays the NavBar at the top of all component pages with a logo. It contains links to the home page, groups page and account page.

Services

Services are used to handle shared logic, data manipulation and communication with the backend Node.js and Express.js server (APIs).

- `AuthenticationService`: Manages user login, creation, logout, account deletion, updating and fetching a user's information. It is also responsible for storing and retrieving user credentials from the browsers local storage.
- `GroupService`: Manages all group, channel and message related operations. It is responsible for fetching, creating, deleting, managing user memberships within groups/channels/messages and stores and retrieves the current state (current group and current channel that the user is attempting access) of the web application from the browser's local storage.
- `SocketService`: Manages the socket connection for a user, allowing them to send/receive messages and join a video chat room.
- `VideoChatService`: Manages the local video and audio stream of the user and the remote video and audio stream of another user within a video chat utilising PeerJS.
- `NotificationService`: Provides a method to display notifications of success and error messages to the user across the web application.

Node.js Server Architecture

The server is built utilising Node.js and Express.js frameworks and connected to a MongoDB database. All the application data is stored within a single MongoDB document to simplify data management. Initially, when the server is run with `node server.js`, it checks if the MongoDB `'db'` database is empty. If it is then the `seed.js` file is run and the base data.json document is loaded into the MongoDB `'db'` database and the `'db'` collection.
Seed Base Data .json Document:

```
{
  "groups": [
    {
      "id": 0,
      "name": "Initial Group",
      "users": [
        {
          "username": "super",
          "profilePicture": null,
          "role": "SuperAdmin"
        }
      ],
      "channels": [
        {
          "groupId": 0,
```

```
          "id": 0,
          "name": "Initial Channel",
          "messages": [
            {
              "username": "super",
              "profilePicture": null,
              "message": "First message",
              "image": null,
              "groupId": 0,
              "channelId": 0
            }
          ],
          "allowedUsers": []
        }
      ]
    },
    {
      "id": 1,
      "name": "noadmin",
      "users": [
        {
          "username": "super",
          "profilePicture": null,
          "role": "SuperAdmin"
        }
      ],
      "channels": [
        {
          "groupId": 1,
          "id": 0,
          "name": "Initial Channel",
          "messages": [
            {
              "username": "super",
              "profilePicture": null,
              "message": "First message",
              "image": null,
              "groupId": 1,
              "channelId": 0
            }
          ],
          "allowedUsers": []
        }
      ]
    }
  ],
  "users": [
    {
      "id": 0,
      "username": "super",
      "email": "super@email.com",
      "profilePicture": null,
      "roles": [
        "SuperAdmin"
```

```
        ],
        "password": "123",
        "groups": [
          {
            "id": 0,
            "name": "Initial Group",
            "users": [
              {
                "username": "super",
                "profilePicture": null,
                "role": "SuperAdmin"
              }
            ],
            "channels": [
              {
                "groupId": 0,
                "id": 0,
                "name": "Initial Channel",
                "messages": [
                  {
                    "username": "super",
                    "profilePicture": null,
                    "message": "First message",
                    "image": null,
                    "groupId": 0,
                    "channelId": 0
                  }
                ],
                "allowedUsers": []
              }
            ]
          },
          {
            "id": 1,
            "name": "noadmin",
            "users": [
              {
                "username": "super",
                "profilePicture": null,
                "role": "SuperAdmin"
              }
            ],
            "channels": [
              {
                "groupId": 1,
                "id": 0,
                "name": "Initial Channel",
                "messages": [
                  {
                    "username": "super",
                    "profilePicture": null,
                    "message": "First message",
                    "image": null,
                    "groupId": 1,
```

```
                  "channelId": 0
                }
              ],
              "allowedUsers": []
            }
          ]
        }
      ],
      "adminGroups": [],
      "requests": [
        {
          "username": "dummy",
          "profilePicture": null,
          "groupId": 0,
          "group": "Initial Group"
        }
      ],
      "allowedChannels": [],
      "reports": []
    },
    {
      "id": 1,
      "username": "dummy",
      "email": "dummy@email.com",
      "profilePicture": null,
      "roles": [
        "User"
      ],
      "password": "123",
      "groups": [],
      "adminGroups": [],
      "requests": [],
      "allowedChannels": [],
      "reports": []
    }
  ]
}
```

Architecture Overview

- `Server.js`: The main entry point of the server. It sets up the Express application and configures middleware such as cors and Express.json to injest `.json` files and more importantly defines the API routes and functionalities.
- `Database.js`: A module that that defines and contains the logic to connecting to the MongoDB database. It exports a function that the route handlers use to get a database instance.
- `Socket.js`: A module that defines the socket rooms that users can enter or leave. It handles emitting messages, joining channels and video chat rooms. It exports a function that the server can use with the specified port and input and output defined by `sockets.io`.
- `Seed.js`: A module that inserts the seed data into the MongoDB database if it is currently empty.
- `/api` directory: Contains individual Javascript files for each API route handler for logging in, creating a new user, deleting a user, adding a user to a group, creating a group, getting user information,

making a request to enter a group, reporting another user and more. E.g. `login.js`, `createUser.js`.
- `/integrationTest` directory: Contains a `test.js` file that contains all the integration tests that test the routes and server logic.

Server API Routes (REST API)

The following table possesses the information of the REST API endpoints that the Angular frontend utilises to communicate with the Node.js server.

Authentication and Users API Endpoints:

Method	Route	Description	Request Body/Params	Success Response
POST	/api/register	Creates a new user account if the username/email is not taken.	{ username, email, password }	200 OK with { user: SessionStorageUser, valid: true }
POST	/api/login	Authenticates a user.	{ username, password }	200 OK with { user, valid: true }
GET	/api/getAllUsers	Retrieves a list of all users, without their passwords.	None	200 OK with { users: SessionStorageUser[], valid: true }
GET	/api/getUserInfo	Retrieves full, updated information for a single user.	Query Param: { username }	200 OK with { user, valid: true }
PATCH	/api/updateUser/profilePicture	Updates a user's profile picture throughout the database.	{ currentUsername, profilePicture }	200 OK with { valid: true }
PATCH	/api/updateUser/username	Updates a user's username.	{ currentUsername, username }	200 OK with { valid: true }
PATCH	/api/updateUser/email	Updates a user's email.	{ currentUsername, email }	200 OK with { valid: true }
PATCH	/api/updateUser/password	Updates a user's password.	{ currentUsername, password }	200 OK with { valid: true }

Method	Route	Description	Request Body/Params	Success Response
PATCH	/api/updateUser/groups	Updates a user's groups.	{ currentUsername, groups }	200 OK with { valid: true }
PATCH	/api/updateUser/adminGroups	Updates a user's admin groups.	{ currentUsername, adminGroups }	200 OK with { valid: true }
DELETE	/api/deleteAccount	Deletes a user account and associated data.	Query Param: { id }	200 OK with { valid: true }

Groups API Endpoints:

Method	Route	Description	Request Body/Params	Success Response
GET	/api/get/group	Retrieves information for a single group.	Query Param: { id }	200 OK with { group, valid: true }
GET	/api/get/allGroups	Retrieves all groups.	None	200 OK with { groups, valid: true }
POST	/api/createGroup	Creates a new group, adding creator as GroupAdmin and all SuperAdmins.	{ name, username, role }	200 OK with { group, valid: true }
DELETE	/api/removeGroup	Deletes a group and all references to it.	Query Param: { id }	200 OK with { valid: true }
POST	/api/requestGroup	Adds a join request to relevant admins' request lists.	Request object	200 OK with { valid: true }

Method	Route	Description	Request Body/Params	Success Response
DELETE	/api/rejectRequest	Rejects/cancels a group join request.	Query Param: { request } (username, groupId, group)	200 OK with { valid: true }
POST	/api/acceptRequest	Accepts a group join request.	Request object	200 OK with { valid: true }
POST	/api/addNewUser	Accepts or manually adds a user to a group, removing any requests.	{ groupId, user }	200 OK with { valid: true }
DELETE	/api/removeUserFromGroup	Removes a user from a group and cleans up data.	Query Param: { user, groupId }	200 OK with { valid: true }
POST	/api/promoteUser	Promotes a user to a new role, incl. "SuperAdmin" logic.	{ username, groupId, role }	200 OK with { valid: true }

Channels API Endpoints:

Method	Route	Description	Request Body/Params	Success Response
POST	/api/createChannel	Creates a new channel and syncs the updated group to members.	{ channelName, groupId }	200 OK with { valid: true }
DELETE	/api/removeChannel	Deletes a channel and syncs the updated group to members.	Query Param: { request } (groupId, channelId)	200 OK with { valid: true }
POST	/api/banUserFromChannel	Removes a user's access to a specific channel.	{ groupId, id, name, username }	200 OK with { valid: true }

Method	Route	Description	Request Body/Params	Success Response
POST	/api/addUserToChannel	Grants a user access to a specific channel.	{ groupId, channelId, name, username, channelObject }	200 OK with { valid: true }

Reports API Endpoints:

Method	Route	Description	Request Body/Params	Success Response
POST	/api/reportUser	Adds a report to the reports list of all SuperAdmins.	Report object	200 OK with { valid: true }
POST	/api/acceptReport	Bans a user from a group and cleans up related data.	Report object	200 OK with { valid: true }

Messages API Endpoints:

Method	Route	Description	Request Body/Params	Success Response
POST	/api/sendMessage	Adds a message to the messages array of a specified channel within a specified group	Message object	200 OK with { valid: true }

Client-Server Interaction

Note: Not all interaction flows will be documented as there are too many and only the main interaction flows will be described

User Login Attempt

1. Client (Admin's Browser): A user is within the login form page and enters their email/username and password and selects the `Login` button.
2. Angular Component: The (click) event triggers the login() method in login.component.ts. This method gets the `LoginAttempt` object which contains the user's email/username and password from the login request.
3. Angular Service: The component calls the AuthenticationService.login(user) method. The service packages the `LoginAttempt`object containing the user's credentials into a JSON payload.

4. HTTP Request: The service sends a HTTP POST request to the /api/login endpoint on the Node.js server.
5. Node.js Server: The route handler for /api/login receives the request.
    - It finds the users array and checks if a user exists with the same email/username and password.
    - If the user exists, then it sends back a 200 OK response with { user: SessionStorageUser, valid: true, error: "" }. Where the user variable contains all the user's information other than the password.
6. Client Response: The Angular AuthenticationService receives the success response. The local storage `Credentials` is set with the user's details and the login component then calls upon the router to navigate to the account component. However, if the response is a failure, then the login component initialises the error message from the received response and the notification service is called to notify the user for the error.

New User Registration

1. Client (Angular): A "SuperAdmin" user clicks the "Create New User" button on the account page and fills out the registration form which includes the new user's username, email and password in the register component and clicks the "Create New User" button.
2. Service: The AuthenticationService packages the username, email, and password into a JSON payload.
3. HTTP Request: The service sends a POST request to the /api/register endpoint on the Node.js server.
4. Server (Node.js): The route handler for /api/register receives the request.
5. It first queries the database to ensure the submitted username or email doesn't already exist.
    - If they are unique, it generates a new user ID, creates the new user object with the default role of "User", and pushes it into the main users array in the database.
    - HTTP Response: The server returns a 200 OK response with the new user object but without the password and a valid: true flag.
6. Client Update: The AuthenticationService receives the success response, and the Notification Service will display a success message indicating that the new user has been created and the "SuperAdmin" user will return back to the account compononet page.

Adding a User to a Group

1. Client (Admin's Browser): An admin views the pending requests in the GroupsComponent. They click the "Review button and then the "Accept" button next within the request popup.
2. Angular Component: The (click) event triggers the acceptRequest() method in groups.component.ts. This method gets the user object from the request.

3. Angular Service: The component calls the groupService.addNewUser(groupId, user) method. The service packages the groupId and user object into a JSON payload.
4. HTTP Request: The service sends a HTTP POST request to the /api/acceptRequest endpoint on the Node.js server.
5. Node.js Server: The route handler for /api/acceptRequest receives the request.
    - It finds the target group and checks that the user isn't already a member.
    - It performs several database updates in parallel:
        - Adds the user to the main group's users array.
        - Adds the group to the user's groups array with `'User'` role.
        - Removes the pending request from all admins' requests arrays.
        - It then propagates the updated group to all other members' groups and adminGroups arrays to ensure data consistency.
    - Finally, it sends back a 200 OK response with { valid: true, error: "" }.
6. Client Response: The Angular GroupService receives the success response. The GroupsComponent then calls its reloadComponent() method, which refreshes the view to reflect the changes where the request disappears from the list

OR

1. Client (Admin's Browser): An admin selects the "Add User" button within the channels component within a selected group. They then select the user they want to add to the group within the new popup and click the "Add" button.
2. The same process is performed like when accepting a request.
3. However, Client Response: The Angular GroupService receives the success response. The Channels component then calls its reloadComponent() method, which refreshes the view to reflect the newly added user to the given group.

Deleting a Group

1. Client (Angular): An administrator clicks the "Remove" button next to a group they manage in the GroupsComponent.
2. Service: The GroupService's removeGroup method is called.
3. HTTP Request: The service sends a DELETE request to the /api/removeGroup endpoint with the id of the group as a query parameter.
4. Server (Node.js): The route handler for /api/removeGroup receives the request.
5. It executes a bulkWrite operation to perform a comprehensive and atomic cleanup across the entire database document.

- It begins by removing the group from the groups list and removes all references to that given group from every user's groups and adminGroups, requests and reports array,
- HTTP Response: The server returns a 200 OK response with { valid: true } upon successful deletion.

6. Client Update: The GroupsComponent reloads, and the deleted group is completely removed from all lists in the UI and the notification service is called display a success message that the group was successfully deleted.

Requesting to Join a Group & Admin Approval

1. Client (Angular): A user in the GroupsComponent clicks the "Request Group" button next to a group they are not a member of within the all Groups list of groups.
2. Service: The GroupService's requestGroup method is called, which packages the user's details and the target groupId into a `Request` object.
3. HTTP Request: The service sends a POST request to /api/requestGroup. The server then adds this request to the requests array of all relevant administrators.
4. Server (Node.js): The /api/requestGroup handler receives the `Request` object.
   - It verifies the user is not already in the group.
   - It adds the `Request` object to all users' requests array with the `'SuperAdmin'` role or if they are a `'GroupAdmin'` of the given group.
5. Admin Action: An administrator sees the pending request in their GroupsComponent and clicks the "Accept" button inside the review modal. This triggers the addNewUser method in the GroupService.
6. Server (Node.js): The /api/addNewUser handler receives the groupId and user object.
   - It verifies the user is not already in the group.
   - It adds the user to the group's main users list and adds the group to the user's personal groups list.
   - Crucially, it removes the now-obsolete request from all admins' requests arrays.
   - It then propagates the updated group object (which now includes the new member) to all other users.
7. Client Update: The GroupsComponent reloads. The pending request disappears from the admin's view, and the user will now see the group in their "My Groups" list upon their next refresh and will be allowed to enter the group as a `'User'`.

Promoting a User's Role

1. Client (Angular): A Super Admin, from a modal in the ChannelComponent, selects a user, chooses the "GroupAdmin" or "SuperAdmin" role from a dropdown, and clicks the "Promote" button.

2. Service: The GroupService's promoteUser method is called. It packages the username, target groupId, and the new role ("GroupAdmin" or "SuperAdmin") into a JSON payload.
3. HTTP Request: The service sends a POST request to the /api/promoteUser endpoint.
4. Server (Node.js): The route handler enters its standard role promotion logic.
5. It performs three database updates in parallel:
    - Adds the new role to the user's main roles array.
    - Updates the user's role to "GroupAdmin" or "SuperAdmin" specifically within the target group's users array.
    - Adds the entire group object to the user's adminGroups array, granting them administrative permissions.
    - Then it updates all the groups within all user's groups and adminGroups arrays to ensure data consistency.
    - HTTP Response: The server returns a 200 OK response with { valid: true }.
6. Client Update: The ChannelComponent reloads. In the list of group members, the promoted user's role badge is updated to "GroupAdmin" or "SuperAdmin" if selected.

Entering a Video Chat

1. Client (Angular): A user, selects a group and then selects a channel to enter messages component. Then they select the "Enter Video Chat" button and it opens up a modal that displays their current camera display and a list of users within the same channel that are currently connected to "call". They select the "call" button for a specified connected user.
2. Service:
    - The `VideoChatService` handles the "call" request and gets the incoming remote video and audio stream from the `join-video-chat` socket room given the remotePeerId.
    - Then it accepts the incoming remote video and audio using PeerJS and then sets up a subscriber for its observers to reflect the changes.
    - Lastly it exposes the video and audio observers to the `Messages` component.
3. Client Update: The Message Component displays the live remote video and audio stream changes from the exposed observers from the `VideoChatService`.

Testing and Coverage

Frontend (Angular)

Testing Frameworks

- Karma and Jasmine for unit tests
- Angular TestBed for component/service testing

- HTTP mock calls were made utilising `HTTPTestingController`

What was Tested

Services:

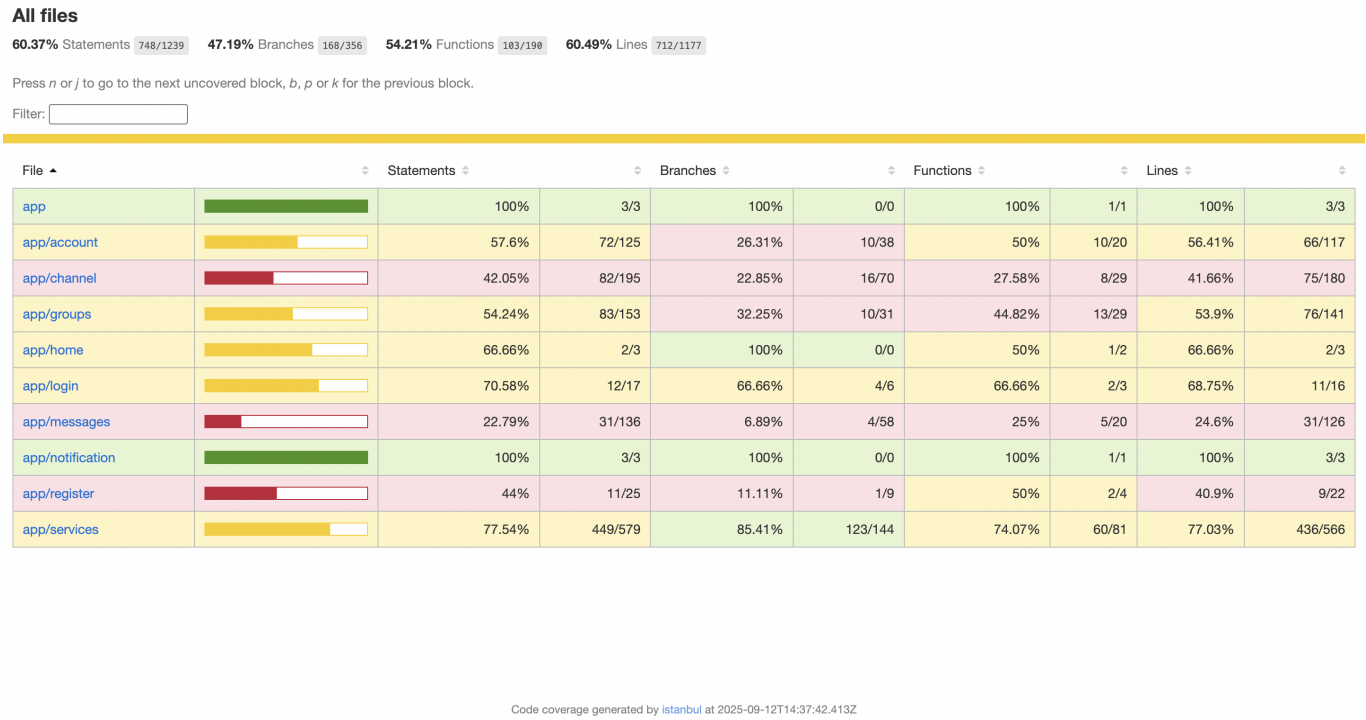
- `AuthenticationService`
- `GroupService`
- `VideoChatService`

Components:

- `Messages`
- `Login`
- `Groups`
- `Channel`
- `Account`

Coverage

Using the command: `ng test --code-coverage`



All files

60.37% Statements `748/1239` **47.19%** Branches `168/356` **54.21%** Functions `103/190` **60.49%** Lines `712/1177`

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
app	100%	3/3	100%	0/0	100%	1/1	100%	3/3
app/account	57.6%	72/125	26.31%	10/38	50%	10/20	56.41%	66/117
app/channel	42.05%	82/195	22.85%	16/70	27.58%	8/29	41.66%	75/180
app/groups	54.24%	83/153	32.25%	10/31	44.82%	13/29	53.9%	76/141
app/home	66.66%	2/3	100%	0/0	50%	1/2	66.66%	2/3
app/login	70.58%	12/17	66.66%	4/6	66.66%	2/3	68.75%	11/16
app/messages	22.79%	31/136	6.89%	4/58	25%	5/20	24.6%	31/126
app/notification	100%	3/3	100%	0/0	100%	1/1	100%	3/3
app/register	44%	11/25	11.11%	1/9	50%	2/4	40.9%	9/22
app/services	77.54%	449/579	85.41%	123/144	74.07%	60/81	77.03%	436/566

Code coverage generated by istanbul at 2025-09-12T14:37:42.413Z

Backend (Node.js/Express)

Testing Frameworks

- Cypress

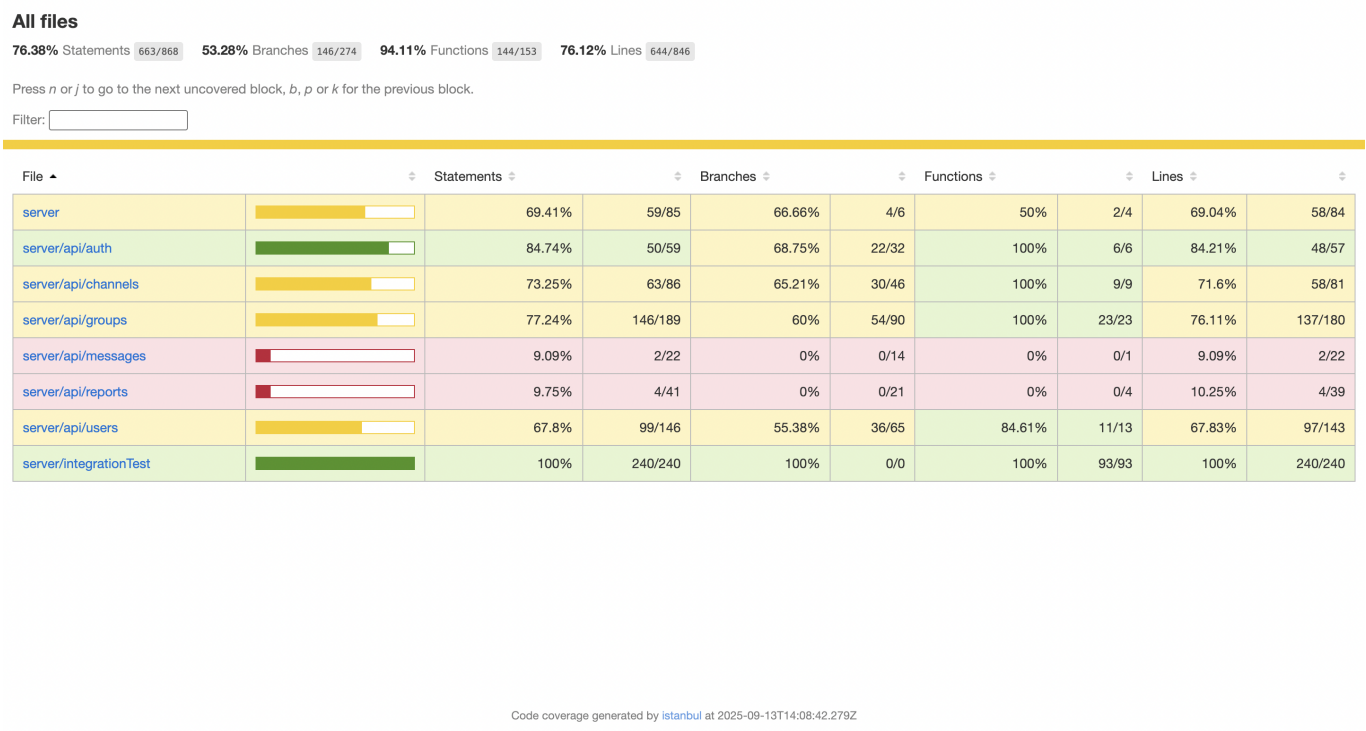
What was Tested

- User registration and login, getting user information, updating user information and deleting a user's account
- Group creation/deletion, getting all the groups in the database, getting a groups information and adding/removing a user from a group

- Request creation/rejection/acceptance
- Channel creation/deletion, banning a user from a channel and adding a user to a channel

Coverage

Using the command: `npm run coverage`



All files

76.38% Statements 663/868 **53.28%** Branches 146/274 **94.11%** Functions 144/153 **76.12%** Lines 644/846

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
server	69.41%	59/85	66.66%	4/6	50%	2/4	69.04%	58/84
server/api/auth	84.74%	50/59	68.75%	22/32	100%	6/6	84.21%	48/57
server/api/channels	73.25%	63/86	65.21%	30/46	100%	9/9	71.6%	58/81
server/api/groups	77.24%	146/189	60%	54/90	100%	23/23	76.11%	137/180
server/api/messages	9.09%	2/22	0%	0/14	0%	0/1	9.09%	2/22
server/api/reports	9.75%	4/41	0%	0/21	0%	0/4	10.25%	4/39
server/api/users	67.8%	99/146	55.38%	36/65	84.61%	11/13	67.83%	97/143
server/integrationTest	100%	240/240	100%	0/0	100%	93/93	100%	240/240

Code coverage generated by istanbul at 2025-09-13T14:08:42.279Z