COMP1531 Major Project

✨ 🎴 UNSW Treats 🎴 ✨

Contents

[[*TOC*]]

0. Aims:

1. Demonstrate effective use of software development tools to build full-stack end-user applications.
2. Demonstrate effective use of static testing, dynamic testing, and user testing to validate and verify software systems.
3. Understand key characteristics of a functioning team in terms of understanding professional expectations, maintaining healthy relationships, and managing conflict.
4. Demonstrate an ability to analyse complex software systems in terms of their data model, state model, and more.
5. Understand the software engineering life cycle in the context of modern and iterative software development practices in order to elicit requirements, design systems thoughtfully, and implement software correctly.
6. Demonstrate an understanding of how to use version control, continuous integration, and deployment tooling to sustainably integrate code from multiple parties.

1. Overview

UNSW needs a change in business model. Revenue has been going down, despite the absolutely perfect MyExperience feedback.

When doing some research, UNSW found the video game industry industry, particularly mobile games like Candy Crush earn over $500 million each year.

UNSW has tasked me (Hayden), and my army of COMP1531 students with investigating the possibility of recreating this game, for UNSW profit.

Only one thing stands in the way...

Microsoft recently bought Candy Crush, and they also own UNSW's only communication platform, **Microsoft Teams**!

If we want to make a Candy Crush competitor, we're going to have to remake Teams first - or those Microsoft spies will shut us down before we even begin development!

The 22T2 cohort of COMP1531 students will build the **backend Javascript server** for a new communication platform, **UNSW Treats** (or just **Treats** for short). I plan to task future COMP6080 students to build the frontend for Treats, something you won't have to worry about.

UNSW Treats is the questionably-named communication tool that allows you to share, communicate, and collaborate virtually *without* intervention from Microsoft spies.

I have already specified a **common interface** for the frontend and backend to operate on. This allows both courses to go off and do their own development and testing under the assumption that both parties will comply with the common interface. This is the interface **you are required to use**.

The specific capabilities that need to be built for this project are described in the interface at the bottom. This is clearly a lot of features, but not all of them are to be implemented at once.

Good luck, and please don't tell anyone at Microsoft about this. 😊

(For legal reasons, this is a joke).

2. Iteration 0: Getting Started

Complete!

3. Iteration 1: Basic Functionality and Tests

Complete!

4. Iteration 2: Building a Web Server

Complete!

5. Iteration 3: Completing the Lifecycle

Iteration 3 builds off all of the work you've completed in iteration 2.

If you haven't completed the implementation of iteration 2, you must complete it as part of this iteration. The automarking for iteration 3 will test on a fully completed interface.

5.1. Task

In this iteration, you are expected to:

1. Make adjustments to your existing code and tests as per any feedback given by your tutor for iteration 2. In particular, you should take time to ensure that your code is well-styled and complies with good software writing practices and software and test design principles discussed in lectures.
2. Implement and test the HTTP Express server according to the entire interface provided in the specification, including features that were added in iteration 3.
    - Part of this section will be automarked.
    - It is required that your data is persistent, just like in iteration 2.
    - `eslint` is assessed identically to iteration 2.
    - Good coverage for all .ts files that aren't tests will be assessed: see section 5.4 for details.
    - You can structure your test files however you choose, as long as they are appended with `.test.ts`. You may place them inside a `/tests` folder, if you wish. For this iteration, we will only be testing your HTTP layer of tests.

- In iteration 2 and 3, we provide a frontend that can be powered by your backend: see section 6.8 for details. Note that the frontend will not work correctly with an incomplete backend. As part of this iteration, it is required that your backend code can correctly power the frontend.
  - You can, if you wish, make changes to the frontend code, but it is not required for this course.
- You must comply with instructions laid out in `5.3`
- Ensure that you correctly manage sessions (tokens) and passwords in terms of authentication and authorisation, as per requirements laid out in section 6.9.

3. Continue demonstrating effective project management and git usage.
   - You will be heavily marked on your thoughtful approach to project management and effective use of git. The degree to which your team works effectively will also be assessed.
   - As for iteration 1 and 2, all task tracking and management will need to be done via the GitLab Taskboard or other tutor-approved tracking mechanism.
   - As for iteration 1 and 2, regular group meetings must be documented with meeting minutes which should be stored at a timestamped location in your repo (e.g. uploading a word doc/pdf or writing in the GitLab repo wiki after each meeting).
   - As for iteration 1 and 2, you must be able to demonstrate evidence of regular standups.
   - You are required to regularly and thoughtfully make merge requests for the smallest reasonable units, and merge them into `master`.
4. Document the planning of new features.
   - You are required to scope out 2-3 problems to solve for future iterations of Treats. You aren't required to build/code them, but you are required to go through SDLC steps of requirements analysis, conceptual modelling, and design.
   - Full detail of this can be found in `5.6`.
5. Deploy your backend to the cloud.
   - You are required to deploy your backend to a cloud provider so that your backend can be accessed from anywhere in the world. **Add the URL to your deployed backend** inside `deployed-url.md`.
   - Full detail of this can be found in `5.7`

5.2. Running the server

To run the server, you can run the following command from the root directory of your project:

```
npm start
```

This will start the server on the port in the `src/server.ts` file, using `ts-node`.

If you get an OSError stating that the address is already in use, you can change the port number in `config.json` to any number from 1024 to 49151. Is it likely that another student may be using your original port number.

Please note: For routes such as `standup/start` and `message/sendlater`, you are not required to account for situations where the server process crashes or restarts while waiting. If the server ever restarts while these active "sessions" are ongoing, you can assume they are no longer happening after restart.

5.3. Implementing and testing features

Continue working on this project by making distinct "features". Each feature should add some meaningful functionality to the project, but still be as small as possible. You should aim to size features as the smallest amount of functionality that adds value without making the project more unstable. For each feature you should:

1. Create a new branch.
2. Write tests for that feature and commit them to the branch. These will fail as you have not yet implemented the feature.
3. Implement that feature.
4. Make any changes to the tests such that they pass with the given implementation. You should not have to do a lot here. If you find that you are, you're not spending enough time on your tests.
5. Create a merge request for the branch.
6. Get someone in your team who **did not** work on the feature to review the merge request. When reviewing, **not only should you ensure the new feature has tests that pass, but you should also check that the coverage percentage has not been significantly reduced.**
7. Fix any issues identified in the review.
8. Merge the merge request into master.

For this project, a feature is typically sized somewhere between a single function, and a whole file of functions (e.g. `auth.ts`). It is up to you and your team to decide what each feature is.

There is no requirement that each feature be implemented by only one person. In fact, we encourage you to work together closely on features, especially to help those who may still be coming to grips with Typescript.

Please pay careful attention to the following:

- We want to see **evidence that you wrote your tests before writing the implementation**. As noted above, the commits containing your initial tests should appear *before* your implementation for every feature branch. If we don't see this evidence, we will assume you did not write your tests first and your mark will be reduced.
- You should have black-box tests for all tests required (i.e. testing each function/endpoint). However, you are also welcome to write whitebox unit tests in this iteration if you see that as important.
- Merging in merge requests with failing pipelines is **very bad practice**. Not only does this interfere with your teams ability to work on different features at the same time, and thus slow down development, it is something you will be penalised for in marking.
- Similarly, merging in branches with untested features is also **very bad practice**. We will assume, and you should too, that any code without tests does not work.

- Pushing directly to `master` is not possible for this repo. The only way to get code into `master` is via a merge request. If you discover you have a bug in `master` that got through testing, create a bugfix branch and merge that in via a merge request.

5.4. Test coverage

To get the coverage of your tests locally, you will need to have two terminals open. Run these commands from the root directory of your project.

In the first terminal, run

```
npm run ts-node-coverage
```

In the second terminal, run jest as usual

```
npm run test
```

Back in the first terminal, stop the server with Ctrl+C or Command+C. There should now be a `/coverage` directory available. Open the `index.html` file in your web browser to see its output.

5.5. Dryrun

There is no new dryrun for iteration 3, however you can still run the dryrun for iteration 2. It consists of 4 tests, one each for your implementation of `clear/v1`, `auth/register/v2`, `channels/create/v2`, and `channels/list/v2`. These only check whether your server wrapper functions accept requests correctly, the format of your return types and simple expected behaviour, so do not rely on these as an indicator for the correctness of your implementation or tests.

To run the dryrun, you should be in the root directory of your project (e.g. `/project-backend`) and use the command:

```
1531 dryrun 2
```

To view the dryrun tests, you can run the following command on the CSE machines:

```
cat ~cs1531/bin/iter2_test.py
```

5.6. Planning for the next problems to solve

Software development is an iterative process - we're never truly finished. As we complete the development and testing of one feature, we're often then trying to understand the requirements and needs of our users to design the next set of features in our product.

For iteration 3 you are going to produce a short report in `planning.pdf` and place it in the repository. The contents of this report will be a simplified approach to understanding user problems, developing requirements, and doing some early designs.

N.B. If you don't know how to produce a PDF, you can easily make one in Google docs and then export to PDF.

We have opted not to provide you with a sample structure - because we're not interested in any rigid structure. Structure it however you best see fit, as we will be marking content.

[Requirements] Elicitation

Find 2-3 people to interview as target users. Target users are people who currently use a tool like Treats, or intend to. Record their name and email address.

Develop a series of questions (at least 4) to ask these target users to understand what *problems* they might have with teamwork-driven communication tools that are currently unsolved by Treats. Give these questions to your target users and record their answers.

Once you have done this, think about how you would solve the target users' problem(s) and write down a brief description of a proposed solution.

[Requirements] Analysis & Specification - Use Cases

Once you've elicited this information, it's time to consolidate it.

Take the responses from the elicitation step and express these requirements as **user stories** (at least 3). Document these user stories. For each user story, add user acceptance criteria as notes so that you have a clear definition of when a story has been completed.

Once the user stories have been documented, generate at least ONE use case that attempts to describe a solution that satifies some of or all the elicited requirements. You can generate a visual diagram or a more written-recipe style, as per lectures.

[Requirements] Validation

With your completed use case work, reach out to the 2-3 people you interviewed originally and inquire as to the extent to which these use cases would adequately describe the problem they're trying to solve. Ask them for a comment on this, and record their comments in the PDF.

[Design] Interface Design

Now that we've established our *problem* (described as requirements), it's time to think about our *solution* in terms of what capabilities would be necessary. You will specify these capabilities as HTTP endpoints, similar to what is described in `6.2`. There is no minimum or maximum of what is needed - it will depend on what problem you're solving.

[Design] Conceptual Modelling - State Diagrams

Now that you have a sense of the problem to solve, and what capabilities you will need to provide to solve it, add at least ONE state diagram to your PDF to show how the state of the application would change based on user actions. The aim of this diagram is to help a developer understand the different states of the application.

5.7. Deployment

You and your team are to host your backend on a cloud provider. Once your backend has been deployed to the cloud, you will be able to point the frontend to use the new URL of where the backend is deployed and use your backend from anywhere in the world. In summary:

- You get your server (that you wrote) deployed to the internet at a public URL
- You still run your frontend locally (which can connect to that server)

Depending on how you and your team have structured your project, your current method of using data may have to be rethought. Deploying to cloud and developing locally require two different mindsets and you and your team may find that you held some assumptions that are valid when developing locally but do not hold when being hosted on the cloud.

We have written a guide on how to deploy to a free cloud provider AlwaysData. Click here to view the guide.

Note that if you choose to use a different cloud provider, your tutor will not be able to assist you.

You must add the URL to your deployed backend inside `deploy-url.md`.

5.8. Marking Criteria

Section	Weighting	Criteria
Automarking (Testing & Implementation)	55%	• Correct implementation of specified functions • Correctly written tests based on the specification requirements • Code coverage (99% coverage gives full marks for the coverage component) • Correctly linted code (worth 4% of this iteration)
Code Quality	10%	• Demonstrated an understanding of good test **coverage** • Demonstrated an understanding of the importance of **clarity** in communicating the purpose of tests and code • Demonstrated an understanding of thoughtful test **design** • Appropriate use of Javascript data structures (arrays, objects, etc.) • Appropriate style as described in section 8.4 • Appropriate application of good software design practices • Implementation of persistent state
Feature demonstrations	10%	• Backend works with the supplied frontend. • Successful implementation of user/profile/uploadphoto and auth/passwordreset

		• Successful deployment to a cloud provider
Git & Project Management	10%	• Meaningful and informative git commit names being used • At least 12 merge requests into master made • A generally equal contribution between team members • Clear evidence of reflection on group's performance and state of the team • Effective use of course-provided MS Teams for communication, demonstrating an ability to competently manage teamwork online • Use of issue board on GitLab or other approved tracking mechanism to manage tasks • Effective use of agile methods such as standups • Minutes/notes taken from group meetings (and stored in a logical place in the repo)
Requirements & Design for future work	15%	• Requirements elicited from potential users, recorded as user stories • User journey justified and expressed as use case(s) • Interface proposed as a potential solution to provide capabilities • State diagram(s) drawn to demonstrate how application responds to actions
(Bonus Marks) Extra Features	10%	• Up to 10% extra marks can be gained through additions of "extra feature(s)". • Marks will be awarded based on 1) Originality, 2) Technical or creative achievement, 3) Lack of bugs associated with it, 4) Size/scale of the addition. • Your tutor is not required to provide any assistance with this, as it's intended for more advanced students once they complete all other criteria to a high standard. • A brief explanation of your additions must be written in a file `extra.md` that is added to your repo. • Section 5.10 provides some examples of extra features you may want to implement, if you need any suggestions. • To give a rough indication of how much time should be spent on extra features, a group scoring in the top 10% should spend ~40 hours collectively on this section

The formula used for automarking in this iteration is:

`Mark = 95*(t * i * min(c + 1, 100)^3) + 5*e` (Mark equals `t` multiplied by `i` multiplied by the minimum of `c + 1` and 100 to the power of three)

Where:

- `t` is the mark you receive for your tests running against your code (100% = your implementation passes all of your tests).
- `i` is the mark you receive for our course tests (hidden) running against your code (100% = your implementation passes all of our tests).
- `c` is the score achieved by running coverage on your entire codebase. Note that 99% coverage is enough to give you full marks for this part.
- `e` is the score between 0-1 achieved by running eslint against your code and the provided configuration.

5.9. Submission

This iteration due date described in section 7. Note there will be no demonstration for iteration 3.

5.10. Extra Features

Your tutor is not required to provide any assistance with this section, as it's intended for more advanced students once they complete all other criteria to a high standard.

A brief explanation of your additions must be written in a file `extra.md` that you need to add to your repo.

Here are some suggestions for extra features.

1. Frontend - **Hangman on Frontend**
    - After a game of Hangman has been started, any user in the channel can type "/guess X" where X is an individual letter. If that letter is contained in the word or phrase they're trying to guess, the app should indicate where it occurs. If it does not occur, more of the hangman is drawn.
    - There is a lot of flexibility in how you achieve this. It can be done only by modifying the backend and relying on messages to communicate the state of the game (e.g. after making a guess, the "Hangman" posts a message with a drawing of the hangman in ASCII/emoji art). Alternatively, you can modify the frontend, if you want to experiment with fancier graphics.
    - The app should use words and phrases from an external source, not just a small handful hardcoded into the app. One suitable source is `/usr/share/dict/words` available on Unix-based systems
    - Note that this part of the specification is deliberately open-ended. You're free to make your own creative choices in exactly how the game should work, as long as the end result is something that could be fairly described as Hangman.
2. Frontend - **Dark Mode** - Modify the frontend code so that on the flip of a switch in the navbar, the website can change to "dark mode" with a colour scheme of your choosing.
3. Frontend - **LaTEX / Markdown Support** - Modify the frontend code so that messages in channels and DMs can be rendered in LaTEX and/or Markdown.
4. **Databases** - Implementing persistence using a form of database via `typeorm`.

5. **New Features** - Implement one or more of the features you have elicited in your Requirements & Design.

5.11. Peer Assessment

Reference 8.5.

6. Interface specifications

These interface specifications come from Hayden & COMP6080, who are building the frontend to the requirements set out below.

6.1. Input/Output types

6.1.1. Iteration 0+ Input/Output Types

Variable name	Type
named exactly **email**	string
has suffix **id**	integer
contains substring **password**	string
named exactly **message**	string
named exactly **start**	integer
contains substring **name**	string
has prefix **is**	boolean

6.1.2. Iteration 1+ Input/Output Types

Variable name	Type
named exactly **email**	string
has suffix **id**	integer
named exactly **length**	integer
named exactly **start**	integer
contains substring **password**	string
named exactly **message**	string
contains substring **name**	string
has prefix **is**	boolean
has prefix **time**	integer (unix timestamp in seconds), [check this out]
(outputs only) named exactly **messages**	Array of objects, where each object contains types { messageId, uId, message, timeSent }

(outputs only) named exactly **channels**	Array of objects, where each object contains types { channelId, name }
has suffix **Str**	string
(outputs only) named exactly **user**	Object containing uId, email, nameFirst, nameLast, handleStr
(outputs only) name ends in **members**	Array of objects, where each object contains types of **user**
(outputs only) named exactly **users**	Array of objects, where each object contains types of **user**

6.1.3. Iteration 2+ Input/Output

Variable name	Type
named exactly **token**	string
(outputs only) named exactly **dms**	Array of objects, where each object contains types { dmId, name }
named exactly **uIds**	Array of user ids

6.1.4. Iteration 3+ Input/Output

Variable name	Type
contains substring **code**	string
has suffix **Id**	integer
has prefix **num**	integer
has suffix **Rate**	float between 0 and 1 inclusive

(outputs only) named exactly **userStats**	Object of shape {
	channelsJoined: [{numChannelsJoined, timeStamp}],
	dmsJoined: [{numDmsJoined, timeStamp}],
	messagesSent: [{numMessagesSent, timeStamp}],
	involvementRate
	}

(outputs only) named exactly **workspaceStats**	Object of shape {
	channelsExist: [{numChannelsExist, timeStamp}],
	dmsExist: [{numDmsExist, timeStamp}],
	messagesExist: [{numMessagesExist, timeStamp}],
	utilizationRate
	}

has suffix **End**	integer
has suffix **Start**	integer

has suffix **Url**	string
(outputs only) name ends in **reacts**	Array of objects, where each object contains types { reactId, uIds, isThisUserReacted } where: • reactId is the id of a react • uIds is an array of user id's of people who've reacted for that react • isThisUserReacted is whether or not the authorised user (user making the request) currently has one of the reacts to this message

(outputs only) named exactly **notifications**	Array of objects, where each object contains types { channelId, dmId, notificationMessage } where
	• channelId is the id of the channel that the event happened in, and is `-1` if it is being sent to a DM
	• dmId is the DM that the event happened in, and is `-1` if it is being sent to a channel
	• notificationMessage is a string of the following format for each trigger action:
	  ◦ tagged: "{User's handle} tagged you in {channel/DM name}: {first 20 characters of the message}"
	  ◦ reacted message: "{User's handle} reacted to your message in {channel/DM name}"
	  ◦ added to a channel/DM: "{User's handle} added you to {channel/DM name}"

(V3) (outputs only) named exactly **user**	Object containing uId, email, nameFirst, nameLast, handleStr, profileImgUrl
(V3) (outputs only) named exactly **messages**	Array of objects, where each object contains types { messageId, uId, message, timeSent, reacts, isPinned }

6.2. Interface

6.2.3. Iteration 2 Interface

Changelog:

- Error returns should be converted to the respective Exception (see table below and section 6.8.2)
- All routes that require a `token` should raise a `403 Error` object when the `token` passed in is invalid.
- Instead of passing `token` as a query or body parameter, you should pass it through a HTTP header (see section 6.9):
  - You should remove `token` from query and body parameters for all routes.
  - You also need to increment the version of each route that previously accepted `token` as a query or body parameter, e.g. v2 --> v3.
- New error case for `channel/leave/v2`, added in table below.
- Added functionality for `message/edit/v2` in regards to standups, in table below.

Name & Description	HTTP	Data Types	Exceptions

	Method		
`auth/login/v3` Given a registered user's email and password, returns their `authUserId` value.	**POST**	**Body Parameters:** `( email, password )` **Return type if no error:** `{ token, authUserId }`	**400 Error** when any of: • email entered does not belong to a user • password is not correct
`auth/register/v3` Given a user's first and last name, email address, and password, create a new account for them and return a new `authUserId`. A handle is generated that is the concatenation of their casted-to-lowercase alphanumeric (a-z0-9) first name and last name (i.e. make lowercase then remove non-alphanumeric characters). If the concatenation is longer than 20 characters, it is cut off at 20 characters. Once you've concatenated it, if the handle is once again taken, append the concatenated names with the smallest number (starting from 0) that forms a new handle that isn't already taken. The addition of this final number may result in the handle exceeding the 20 character limit (the handle 'abcdefghijklmnopqrst0' is allowed if the handle 'abcdefghijklmnopqrst' is already taken).	**POST**	**Body Parameters:** `( email, password, nameFirst, nameLast )` **Return type if no error:** `{ token, authUserId }`	**400 Error** when any of: • email entered is not a valid email (more in section 6.4) • email address is already being used by another user • length of password is less than 6 characters • length of nameFirst is not between 1 and 50 characters inclusive • length of nameLast is not between 1 and 50 characters inclusive
`channels/create/v3` Creates a new channel with the given name that is either a public or private channel. The user who created it automatically joins the channel.	**POST**	**Body Parameters:** `( name, isPublic )` **Return type if no error:** `{ channelId }`	**400 Error** when: • length of name is less than 1 or more than 20 characters

`channels/list/v3`
 	
Provide an array of all channels (and their associated details) that the authorised user is part of.
	GET	**Query Parameters:**
		`( )`
		
		Return type if no error:
		`{ channels }`
			N/A

`channels/listall/v3`
 	
Provide an array of all channels, including private channels, (and their associated details)
	GET	**Query Parameters:**
		`( )`
		
		Return type if no error:
		`{ channels }`
			N/A

`channel/details/v3`
 	
Given a channel with ID channelId that the authorised user is a member of, provide basic details about the channel.
	GET	**Query Parameters:**
		`( channelId )`
		
		Return type if no error:
		`{ name, isPublic, ownerMembers, allMembers }`
			400 Error when:
			• channelId does not refer to a valid channel
			
			403 Error when:
			• channelId is valid and the authorised user is not a member of the channel

`channel/join/v3`
 	
Given a channelId of a channel that the authorised user can join, adds them to that channel.
	POST	**Body Parameters:**
		`( channelId )`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• channelId does not refer to a valid channel
			• the authorised user is already a member of the channel
			
			403 Error when:
			• channelId refers to a channel that is private and the authorised user is not already a channel member and is not a global owner

`channel/invite/v3` Invites a user with ID uId to join a channel with ID channelId. Once invited, the user is added to the channel	**POST**	**Body Parameters:** `( channelId, uId )`	**400 Error** when any of: • channelId does not refer to a valid channel

immediately. In both public and private channels, all members are able to invite users.		**Return type if no error:** `{}`	• uId does not refer to a valid user • uId refers to a user who is already a member of the channel **403 Error** when: • channelId is valid and the authorised user is not a member of the channel

`channel/messages/v3`
 	
Given a channel with ID channelId that the authorised user is a member of, return up to 50 messages between index "start" and "start + 50". Message with index 0 is the most recent message in the channel. This function returns a new index "end" which is the value of "start + 50", or, if this function has returned the least recent messages in the channel, returns -1 in "end" to indicate there are no more messages to load after this return.
	GET	**Query Parameters:**
		`( channelId, start )`
		
		Return type if no error:
		`{ messages, start, end }`
			400 Error when any of:
			
			• channelId does not refer to a valid channel
			• start is greater than the total number of messages in the channel
			
			403 Error when any of:
			
			• channelId is valid and the authorised user is not a member of the channel

`user/profile/v3`
 	
For a valid user, returns information about their userId, email, first name, last name, and handle
	GET	**Query Parameters:**
		`( uId )`
		
		Return type if no error:
		`{ user }`
			400 Error when:
			
			• uId does not refer to a valid user

`clear/v1`
 	
Resets the internal data of the application to its initial state
	DELETE	**Parameters:**
		`()`
		
		Return type if no error:
		`{}`
			N/A

`auth/logout/v2` Given an active token, invalidates the token to log the user out.	**POST**	**Body Parameters:** `{ }` **Return type if**	N/A

		no error: `{}`	

`channel/leave/v2`
 	
Given a channel with ID channelId that the authorised user is a member of, remove them as a member of the channel. Their messages should remain in the channel. If the only channel owner leaves, the channel will remain.
	POST	**Body Parameters:**
		`{ channelId }`
		
		Return type if no error:
		`{}`
			400 Error when:
			• channelId does not refer to a valid channel
			• the authorised user is the starter of an active standup in the channel
			
			403 Error when any of:
			• channelId is valid and the authorised user is not a member of the channel

`channel/addowner/v2`
 	
Make user with user id uId an owner of the channel.
	POST	**Body Parameters:**
		`{ channelId, uId }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• channelId does not refer to a valid channel
			• uId does not refer to a valid user
			• uId refers to a user who is not a member of the channel
			• uId refers to a user who is already an owner of the channel
			
			403 Error when:
			• channelId is valid and the authorised user does not have owner permissions in the channel

`channel/removeowner/v2`
 	
Remove user with user id uId as an owner of the channel.
	POST	**Body Parameters:**
		`{ channelId, uId }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• channelId does not refer to a valid channel
			• uId does not refer to a valid user
			• uId refers to a user who is not an owner of the channel
			• uId refers to a user who is currently the only owner of the channel

			403 Error when any of: • channelId is valid and the authorised user does not have owner permissions in the channel

`message/send/v2`
 	
Send a message from the authorised user to the channel specified by channelId. Note: Each message should have its own unique ID, i.e. no messages should share an ID with another message, even if that other message is in a different channel.
	POST	**Body Parameters:**
		`{ channelId, message }`
		
		Return type if no error:
		`{ messageId }`
			400 Error when any of:
			• channelId does not refer to a valid channel
			• length of message is less than 1 or over 1000 characters
			
			403 Error when any of:
			• channelId is valid and the authorised user is not a member of the channel

`message/edit/v2`
 	
Given a message, update its text with new text. If the new message is an empty string, the message is deleted. **NEW IN ITERATION 3**: If a shared/standup message is edited, the entire contents will be edited as if it was a normal message.
	PUT	**Body Parameters:**
		`{ messageId, message }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• length of message is over 1000 characters
			• messageId does not refer to a valid message within a channel/DM that the authorised user has joined
			
			403 Error when any of:
			• If the authorised user does not have owner permissions, and the message was not sent by them

`message/remove/v2`
 	
Given a messageId for a message, this message is removed from the channel/DM
	DELETE	**Query Parameters:**
		`( messageId )`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• messageId does not refer to a valid message within a channel/DM that the authorised user has joined
			
			403 Error when any of:

			• If the authorised user does not have owner permissions, and the message was not sent by them

`dm/create/v2`
 	
`uIds` contains the user(s) that this DM is directed to, and will not include the creator. The creator is the owner of the DM. `name` should be automatically generated based on the users that are in this DM. The name should be an alphabetically-sorted, comma-and-space-separated concatenation of user handles, e.g. 'ahandle1, bhandle2, chandle3'.
	POST	**Body Parameters:**
		`{ uIds }`
		
		Return type if no error:
		`{ dmId }`
			400 Error when any of:
			• any uId in uIds does not refer to a valid user
			• there are duplicate 'uId's in uIds

`dm/list/v2`
 	
Returns the array of DMs that the user is a member of.
	GET	**Query Parameters:**
		`( )`
		
		Return type if no error:
		`{ dms }`
			N/A

`dm/remove/v2`
 	
Remove an existing DM, so all members are no longer in the DM. This can only be done by the original creator of the DM.
	DELETE	**Query Parameters:**
		`( dmId )`
		
		Return type if no error:
		`{}`
			400 Error when:
			• dmId does not refer to a valid DM
			
			403 Error when any of:
			• dmId is valid and the authorised user is not the original DM creator
			• dmId is valid and the authorised user is no longer in the DM

`dm/details/v2` Given a DM with ID dmId that the authorised user is a member of, provide basic details about the DM.	**GET**	**Query Parameters:** `( dmId )` **Return type if no error:**	**400 Error** when: • dmId does not refer to a valid DM **403 Error** when:

		`{ name, members }`	• dmId is valid and the authorised user is not a member of the DM

`dm/leave/v2`
 	
Given a DM ID, the user is removed as a member of this DM. The creator is allowed to leave and the DM will still exist if this happens. This does not update the name of the DM.
	POST	**Body Parameters:**
		`{ dmId }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• dmId does not refer to a valid DM
			
			403 Error when any of:
			• dmId is valid and the authorised user is not a member of the DM

`dm/messages/v2`
 	
Given a DM with ID dmId that the authorised user is a member of, return up to 50 messages between index "start" and "start + 50". Message with index 0 is the most recent message in the DM. This function returns a new index "end" which is the value of "start + 50", or, if this function has returned the least recent messages in the DM, returns -1 in "end" to indicate there are no more messages to load after this return.
	GET	**Query Parameters:**
		`( dmId, start )`
		
		Return type if no error:
		`{ messages, start, end }`
			400 Error when any of:
			• dmId does not refer to a valid DM
			• start is greater than the total number of messages in the channel
			
			403 Error when any of:
			• dmId is valid and the authorised user is not a member of the DM

`message/senddm/v2`
 	
Send a message from authorisedUser to the DM specified by dmId. Note: Each message should have it's own unique ID, i.e. no messages should share an ID with another message, even if that other message is in a different channel or DM.
	POST	**Body Parameters:**
		`{ dmId, message }`
		
		Return type if no error:
		`{ messageId }`
			400 Error when any of:
			• dmId does not refer to a valid DM
			• length of message is less than 1 or over 1000 characters
			
			403 Error when any of:
			• dmId is valid and the authorised user is not a member of the DM

`users/all/v2` Returns an array of all users and their associated details.	**GET**	**Query Parameters:** `( )` **Return type if**	N/A

		no error: `{ users }`	

`user/profile/setname/v2`
 	
Update the authorised user's first and last name
	PUT	**Body Parameters:**
		`{ nameFirst, nameLast }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• length of nameFirst is not between 1 and 50 characters inclusive
			• length of nameLast is not between 1 and 50 characters inclusive

`user/profile/setemail/v2`
 	
Update the authorised user's email address
	PUT	**Body Parameters:**
		`{ email }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• email entered is not a valid email (more in section 6.4)
			• email address is already being used by another user

`user/profile/sethandle/v2`
 	
Update the authorised user's handle (i.e. display name)
	PUT	**Body Parameters:**
		`{ handleStr }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• length of handleStr is not between 3 and 20 characters inclusive
			• handleStr contains characters that are not alphanumeric
			• the handle is already used by another user

6.2.4. Iteration 3 Interface

All return values should be an object, with keys identically matching the names in the table below, along with their respective values.

Name & Description	HTTP Method	Data Types	Exceptions

`notifications/get/v1`
 	
Return the user's most recent 20 notifications, ordered from most recent to least recent.
	GET	**Query Parameters:**
		`( )`
		
		Return type if no error:
		`{ notifications }`
			N/A

`search/v1`
 	
Given a query string, return a collection of messages in all of the channels/DMs that the user has joined that contain the query (case-insensitive). There is no expected order for these messages.
	GET	**Query Parameters:**
		`( queryStr )`
		
		Return type if no error:
		`{ messages }`
			400 Error when:
			• length of queryStr is less than 1 or over 1000 characters

`message/share/v1`
 	
`ogMessageId` is the ID of the original message. `channelId` is the channel that the message is being shared to, and is `-1` if it is being sent to a DM. `dmId` is the DM that the message is being shared to, and is `-1` if it is being sent to a channel. `message` is the optional message in addition to the shared message, and will be an empty string `''` if no message is given.
 	
A new message containing the contents of both the original message and the optional message should be sent to the channel/DM identified by the channelId/dmId. The format of the new message does not matter as long as both the original and optional message exist as a substring within the new message. Once sent, this new message has no link to the original message, so if the original message is edited/deleted, no change will occur for the new message.
	POST	**Body Parameters:**
		`{ ogMessageId, message, channelId, dmId }`
		
		Return type if no error:
		`{ sharedMessageId }`
			400 Error when any of:
			• both channelId and dmId are invalid
			• neither channelId nor dmId are -1
			• ogMessageId does not refer to a valid message within a channel/DM that the authorised user has joined
			• length of message is more than 1000 characters
			
			403 Error when:
			• the pair of channelId and dmId are valid (i.e. one is -1, the other is valid) and the authorised user has not joined the channel or DM they are trying to share the message to

`message/react/v1`
 	
Given a message within a channel or DM the authorised user is part of, add a "react" to that particular message.
	POST	**Body Parameters:**
		`{ messageId, reactId }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• messageId is not a valid message within a channel or DM that the authorised user has joined
			• reactId is not a valid react ID - currently, the

			only valid react ID the frontend has is 1 • the message already contains a react with ID reactId from the authorised user

`message/unreact/v1`
 	
Given a message within a channel or DM the authorised user is part of, remove a "react" to that particular message.
	POST	**Body Parameters:**
		`{ messageId, reactId }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• messageId is not a valid message within a channel or DM that the authorised user has joined
			• reactId is not a valid react ID
			• the message does not contain a react with ID reactId from the authorised user

`message/pin/v1`
 	
Given a message within a channel or DM, mark it as "pinned".
	POST	**Body Parameters:**
		`{ messageId }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• messageId is not a valid message within a channel or DM that the authorised user has joined
			• the message is already pinned
			
			403 Error when:
			• messageId refers to a valid message in a joined channel/DM and the authorised user does not have owner permissions in the channel/DM

`message/unpin/v1`
 	
Given a message within a channel or DM, remove its mark as "pinned".
	POST	**Body Parameters:**
		`{ messageId }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• messageId is not a valid message within a channel or DM that the authorised user has joined

			• the message is not already pinned **403 Error** when: • messageId refers to a valid message in a joined channel/DM and the authorised user does not have owner permissions in the channel/DM

`message/sendlater/v1`
 	
Send a message from the authorised user to the channel specified by channelId automatically at a specified time in the future. The returned messageId will only be considered valid for other actions (editing/deleting/reacting/etc) once it has been sent (i.e. after timeSent).
	POST	**Body Parameters:**
		`{ channelId, message, timeSent }`
		
		Return type if no error:
		`{ messageId }`
			400 Error when any of:
			
			• channelId does not refer to a valid channel
			• length of message is less than 1 or over 1000 characters
			• timeSent is a time in the past
			
			403 Error when:
			
			• channelId is valid and the authorised user is not a member of the channel they are trying to post to

`message/sendlaterdm/v1`
 	
Send a message from the authorised user to the DM specified by dmId automatically at a specified time in the future. The returned messageId will only be considered valid for other actions (editing/deleting/reacting/etc) once it has been sent (i.e. after timeSent). If the DM is removed before the message has sent, the message will not be sent.
	POST	**Body Parameters:**
		`{ dmId, message, timeSent }`
		
		Return type if no error:
		`{ messageId }`
			400 Error when any of:
			
			• dmId does not refer to a valid DM
			• length of message is less than 1 or over 1000 characters
			• timeSent is a time in the past
			
			403 Error when:
			
			• dmId is valid and the authorised user is not a member of the DM they are trying to post to

`standup/start/v1` For a given channel, start a standup	**POST**	**Body Parameters:** `{ channelId, length }`	**400 Error** when any of:

period lasting `length` seconds. During this standup period, if someone calls `standup/send` with a message, it will be buffered during the `length`-second window. Then, at the end of the standup, all buffered messages are packaged into one message, and this packaged message is sent to the channel from the user who started the standup: see section 6.13. for more details. If no standup messages are sent during the standup, no message should be sent at the end.		**Return type if no error:** `{ timeFinish }`	• channelId does not refer to a valid channel • length is a negative integer • an active standup is currently running in the channel **403 Error** when: • channelId is valid and the authorised user is not a member of the channel

`standup/active/v1`
 	
For a given channel, return whether a standup is active in it, and what time the standup finishes. If no standup is active, then timeFinish should be `null`.
	GET	**Query Parameters:**
		`( channelId )`
		
		Return type if no error:
		`{ isActive, timeFinish }`
			400 Error when:
			
			• channelId does not refer to a valid channel
			
			403 Error when:
			
			• channelId is valid and the authorised user is not a member of the channel

`standup/send/v1`
 	
For a given channel, if a standup is currently active in the channel, send a message to get buffered in the standup queue. Note: @ tags should not be parsed as proper tags (i.e. no notification should be triggered on send, or when the standup finishes)
	POST	**Body Parameters:**
		`{ channelId, message }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			
			• channelId does not refer to a valid channel
			• length of message is over 1000 characters
			• an active standup is not currently running in the channel
			
			403 Error when:
			
			• channelId is valid and the authorised user is not a member of the channel

`auth/passwordreset/request/v1`
 	
Given an email address, if the email address belongs to a registered user, send them an email containing a secret password reset code. This
	POST	**Body Parameters:**
		`{ email }`
		
		Return type if no error:
		`{}`
			N/A

code, when supplied to `auth/passwordreset/reset`, shows that the user trying to reset the password is the who got sent this email. No error should be raised when given an invalid email, as that would pose a security/privacy concern. When a user requests a password reset, they should be logged out of all current sessions.			

`auth/passwordreset/reset/v1`
 	
Given a reset code for a user, set that user's new password to the password provided. Once a reset code has been used, it is then invalidated.
	POST	**Body Parameters:**
		`{ resetCode, newPassword }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• resetCode is not a valid reset code
			• password entered is less than 6 characters long

`user/profile/uploadphoto/v1`
 	
Given a URL of an image on the internet, crop the image within bounds (xStart, yStart) and (xEnd, yEnd). Position (0,0) is the top left. Please note: the URL needs to be a non-https URL (it should just have "http://" in the URL). We will only test with non-https URLs.
	POST	**Body Parameters:**
		`{ imgUrl, xStart, yStart, xEnd, yEnd }`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• imgUrl returns an HTTP status other than 200, or any other errors occur when attempting to retrieve the image
			• any of xStart, yStart, xEnd, yEnd are not within the dimensions of the image at the URL
			• xEnd is less than or equal to xStart or yEnd is less than or equal to yStart
			• image uploaded is not a JPG

`user/stats/v1`
 	
Fetch the required statistics about this user's use of UNSW Treats.
	GET	**Query Parameters:**
		`( )`
		
		Return type if no error:
		`{ userStats }`
			N/A

`users/stats/v1` Fetch the required statistics about the workspace's use of UNSW Treats.	**GET**	**Query Parameters:** `( )`	N/A

		Return type if no error: `{ workspaceStats }`	

`admin/user/remove/v1`
 	
Given a user by their uId, remove them from the Treats. This means they should be removed from all channels/DMs, and will not be included in the array of users returned by `users/all`. Treats owners can remove other Treats owners (including the original first owner). Once users are removed, the contents of the messages they sent will be replaced by 'Removed user'. Their profile must still be retrievable with `user/profile`, however nameFirst should be 'Removed' and nameLast should be 'user'. The user's email and handle should be reusable.
	DELETE	**Query Parameters:**
		`( uId )`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• uId does not refer to a valid user
			• uId refers to a user who is the only global owner
			
			403 Error when:
			• the authorised user is not a global owner

`admin/userpermission/change/v1`
 	
Given a user by their user ID, set their permissions to new permissions described by permissionId.
	POST	**Body Parameters:**
		`( uId, permissionId )`
		
		Return type if no error:
		`{}`
			400 Error when any of:
			• uId does not refer to a valid user
			• uId refers to a user who is the only global owner and they are being demoted to a user
			• permissionId is invalid
			• the user already has the permissions level of permissionId
			
			403 Error when:
			• the authorised user is not a global owner

6.3. Valid email format

To check an email is valid, you may use the following package and function.

```
import validator from 'validator';

validator.isEmail('foo@bar.com');
```

6.4. Testing

A common question asked throughout the project is usually "How can I test this?" or "Can I test this?" In any situation, most things can be tested thoroughly. However, some things can only be tested sparsely, and in some other rare occasions, some things can't be tested at all. A challenge of this project is for you to use your discretion to figure out what to test, and how much to test. Often, you can use functions you've already written to test new functions in a black-box manner.

6.5. Pagination

The behaviour in which channelMessages returns data is called **pagination**. It's a commonly used method when it comes to getting theoretially unbounded amounts of data from a server to display on a page in chunks. Most of the timelines you know and love - Facebook, Instagram, LinkedIn - do this.

For example, in iteration 1, if we imagine a user with `authUserId` 12345 is trying to read messages from channel with ID 6, and this channel has 124 messages in it, 3 calls from the client to the server would be made. These calls, and their corresponding return values would be:

- `channelMessages(12345, 6, 0) => { [messages], 0, 50 }`
- `channelMessages(12345, 6, 50) => { [messages], 50, 100 }`
- `channelMessages(12345, 6, 100) => { [messages], 100, -1 }`

Pagination should also apply to messages in DMs.

6.6. Permissions

- Members in a channel/DM have one of two channel/DM permissions
    1. Owners of the channel/DM
    2. Members of the channel/DM
- Treats users have two global permissions
    1. Owners (permission id 1), who can also modify other owners' permissions
    2. Members (permission id 2), who do not have any special permissions
- All Treats users are global members by default, except for the very first user who signs up, who is a global owner

A user's primary permissions are their global permissions. Then the channel/DM permissions are layered on top. For example:

- An owner of Treats has channel owner permissions in every channel they've joined. Despite obtaining owner permissions upon joining a channel, they do not become channel owners unless a channel owner explicitly adds them as one. Hence, if they are removed as a global owner, they will no longer have those channel owner permissions.
- Treats owners do not have owner permissions in DMs. The only users with owner permissions in DMs are the original creators of each DM.

- A member of Treats is a member in channels that they are part of but are not owners of
- A member of Treats is an owner in channels they are owners of

6.7. User Sessions

Iteration 2 introduces the concept of **sessions**. With sessions, when a user logs in or registers, they receive a "token" (think of it like a ticket to a concert). These tokens are stored on the web browser, and nearly every time that user wants to make a request to the server, they will pass this "token" as part of this request. In this way, the server is able to take this token, look at it (like checking a ticket), and figure out who the user is.

A token (to represent a session) for iteration 2 can be as simple a randomly generated number (converted to a string as per the interface specifications) and stored as one of many possible sessions against a specific user.

In this structure, this also means it's possible to "log out" a particular user's session without logging out other sessions. I.e. One user can log in on two different browser tabs, click logout on tab 1, but still functionally use the website on tab 2.

Don't worry about creating a secure method of session storage in iteration 2 - that is for iteration 3.

6.8. Working with the frontend

There is a SINGLE repository available for all students at https://gitlab.cse.unsw.edu.au/COMP1531/22T2/project-frontend. You can clone this frontend locally. If you'd like to modify the frontend repo (i.e. teach yourself some frontend), please FORK the repository.

If you run the frontend at the same time as your express server is running on the backend, then you can power the frontend via your backend.

Please note: The frontend may have very slight inconsistencies with expected behaviour outlined in the specification. Our automarkers will be running against your compliance to the specification. The frontend is there for further testing and demonstration.

6.8.1. Example implementation

A working example of the frontend can be used at http://treats-unsw.herokuapp.com/. This is not a gospel implementation that dictates the required behaviour for all possible occurrences. Our implementation will make reasonable assumptions just as yours will, and they might be different, and that's fine.

The data is reset occasionally, but you can use this link to play around and get a feel for how the application should behave.

6.8.2. Error raising

Either a `400 (Bad Request)` or `403 (Forbidden)` is thrown when something goes wrong. A `400` error refers to issues with user input, whereas a `403` error refers to issues with authorisation. All of these cases are listed in the **Interface** table. If input implies that both errors should be thrown, throw a `403` error.

One exception is that, even though it's not listed in the table, for all routes except `auth/register`, `auth/login`, `auth/passwordreset/request` and `auth/passwordreset/reset`, a `403` error is thrown when the token passed in is invalid.

For errors to be appropriately raised on the frontend, they must be thrown as follows:

```
if (true) { // condition here
    throw new HTTPError(403, "description")
}
```

The quality of the descriptions will not be assessed, but you must modify your errors to this format.

There has also been a middleware handler added to your `server.ts` file to take care of errors encountered. The `middleware-http-errors`[https://www.npmjs.com/package/middleware-http-errors] package is custom-made for COMP1531 students, used as follows:

```
app.use(errorHandler());
```

6.9. Secure Sessions & Passwords

Passwords must be stored in an **encrypted** form.

You must **hash** tokens in iteration 3, and pass them through a custom `token` HTTP Header (rather than passing them plainly as `GET/DELETE` parameters).

In this model, you will replace `token` query and body parameters with a `token` HTTP header when dealing with requests only. You shouldn't remove `token` parameters from backend functions, as they must perform the validity checks.

You can access HTTP headers like so:

```
const token = req.header('token');
```

A sample flow logging a user in might be as follows (other flows exist too):

1. Client makes a valid `auth/login` call
2. Server generates `token` and `hashOf(token+secret)`
3. Server returns the hash as the `token` value in the response's body.

A sample flow creating a channel might be as follows:

1. Client makes a valid `channel/create` call
2. Server gets `token` hash in th request's HTTP header
3. Server passes `token` hash to the relevant backend function to compare to the stored `token`, determining if it's a valid session.

Why hash tokens? If we hash tokens (combined with a global secret) before storing them, and an attacker gets access to our backend of active sessions (i.e. our list of valid tokens), they won't be able to determine the client-side token (as they don't know the hash function or secret added to the token).

Why pass tokens as a HTTP header? Any query parameters (those used by `GET/DELETE` functions) can be read in plaintext by an eavesdropper spying on your HTTP requests. Hence, by passing an authentication token as a query parameter, we're allowing an attacker to intercept our request, steal our token and impersonate other users! On the other hand, HTTP headers are encrypted (as long as you use HTTPS protocol), meaning an eavesdropper won't be able to read token values.

While this safely protects sessions from server-side attacks (accessing our persistent data) and man-in-the-middle attacks (intercepting our HTTP requests), it doesn't protect against client-side attacks (stealing a token on the client-side, after the HTTP header has been decoded and received by the user).

You do not need to worry about mitigating client-side attacks, but you can read more about industry-standard session management here.

6.10. Notifications and tagging users

6.10.1 Notifications

If an action triggering a notification has been 'undone' (e.g. a message has been unreacted, or a tagged message has been edited/removed), the original notification should not be affected and will remain.

A user should not be notified of any reactions to their messages if they are no longer in the channel/DM that the message was sent in.

6.10.2 Tagging

A user is tagged when a message contains the @ symbol, followed immediately by the user's handle. The end of the handle is signified by the end of the message, or a non-alphanumeric character. The message '`hi@handle`' contains a valid tag. '`@handle1@handle2 hello!`' contains two valid tags.

Some additional requirements are:

- If the handle is invalid, or the user is not a member of the channel or DM, no one is tagged.
- A user should be able to tag themselves.
- A message can contain multiple tags.
- If the same valid tag appears multiple times in one message, the user is only notified once.

Tagging should also occur when messages are edited to contain tags and when the message/share optional message contains tags.

There is no requirement to have tags notify users inside a standup or when the buffered standup messages are sent.

6.11. Analytics

COMP6080 students have implemented analytics pages for users and for the Treats workspace on the frontend, and now these pages need data. Your task is to add backend functionality that keeps track of these metrics:

For users:

- The number of channels the user is a part of

- The number of DMs the user is a part of
- The number of messages the user has sent
- The user's involvement, as defined by this pseudocode: `sum(numChannelsJoined, numDmsJoined, numMsgsSent)/sum(numChannels, numDms, numMsgs)`. If the denominator is 0, involvement should be 0. If the involvement is greater than 1, it should be capped at 1.

For the Treats workspace:

- The number of channels that exist currently
- The number of DMs that exist currently
- The number of messages that exist currently
- The workspace's utilization, which is a ratio of the number of users who have joined at least one channel/DM to the current total number of users, as defined by this pseudocode: `numUsersWhoHaveJoinedAtLeastOneChannelOrDm / numUsers`

As UNSW is very interested in its users' engagement, the analytics must be **time-series data**. This means every change to the above metrics (excluding `involvementRate` and `utilizationRate`) must be timestamped, rather than just the most recent change. For users, the first data point should be 0 for all metrics at the time that their account was created. Similarly, for the workspace, the first data point should be 0 for all metrics at the time that the first user registers. The first element in each array should be the first metric. The latest metric should be the last element in the array.

For users, the number of channels and DMs that they are a part of can increase and decrease over time, however the number of messages sent will only increase (the removal of messages does not affect it).

For the workspace, `numMsgs` is the number of messages that exist at the current time, and should decrease when messages are removed, or when `dm/remove` is called. Messages which have not been sent yet with `message/sendlater` or `message/sendlaterdm` are not included, and `standup/send` messages only count when the final packaged standup message from `standup/start` has been sent. `numChannels` will never decrease as there is no way to remove channels, and `numDms` will only decrease when `dm/remove` is called.

In addition to keeping track of these metrics, you are required to implement two new endpoints, `user/stats` and `users/stats`.

6.12. Reacts

The only React ID currently associated with the frontend is React ID 1, which is a thumbs up. You are welcome to add more (this will require some frontend work).

6.13. Standups

Once a standup is finished, all of the messages sent to `standup/send` are packaged together in *one single message* posted by *the user who started the standup*. This packaged message is sent as a message to the channel where the standup was started, timestamped at the moment the standup finished.

The structure of the packaged message is like this:

```
[messageSender1Handle]: [message1]
[messageSender2Handle]: [message2]
```

```
[messageSender3Handle]: [message3]
[messageSender4Handle]: [message4]
```

For example:

```
jake: I ate a catfish
hayden: I went to kmart
emily: I ate a toaster
nick: my catfish ate a kmart toaster
```

Standups can be started on the frontend by typing "/standup X" (where X is the number of seconds that the standup lasts for) into the message input and clicking send. E.g., to start a 90-second standup, type "/standup 90" and press send.

You will not be tested on any behaviour involving the user who started a standup being removed from the channel or Treats while the standup is ongoing, therefore you can decide this behaviour yourself.

6.14. profileImgUrl & image uploads

For outputs with data pertaining to a user, a `profileImgUrl` must be present. When images are uploaded for a user profile, after processing them you should store them on the server such that your server now locally has a copy of the cropped image of the original file linked. Then, the `profileImgUrl` should be a URL to the server, such as http://localhost:5001/imgurl/adfnajnerkn23k4234.jpg (a unique url you generate).

For any given user, if they have yet to upload an image, there should be a site-wide default image used.

Note: This is most likely the most challenging part of the project, so don't get lost in this. We would strongly recommend most teams complete this capability *last*.

7. Due Dates and Weightings

Iteration	Due date	Demonstration to tutor(s)	Assessment weighting of project (%)
0	10pm Friday 10th June (**week 2**)	No demonstration	5%
1	10pm Friday 24th June (**week 4**)	In YOUR **week 5** laboratory	30%
2	10pm Friday 15th July (**week 7**)	In YOUR **week 8** laboratory	35%
3	10pm Friday 5th August (**week 10**)	No demonstration	30%

7.1. Submission & Late Penalties

There is no late penalty, as we do not accept late submissions. You will be assessed on the most recent version of your work at the due date and time.

To submit your work, open up a CSE terminal and run:

```
$ 1531 submit [iteration] [groupname]
```

For example:

```
$ 1531 submit iteration1 W11A_EGGS
```

Only one person per group needs to run this command. This will submit a copy of your latest git commit to our systems for automarking. Your tutor will also request you pull up this copy when marking you in the demonstration.

If the deadline is approaching and you have features that are either untested or failing their tests, **DO NOT MERGE IN THOSE MERGE REQUESTS**. In some cases, your tutor will look at unmerged branches and may allocate some reduced marks for incomplete functionality, but `master` should only contain working code.

Minor isolated fixes after the due date are allowed but carry a penalty if the mark after re-running the autotests is greater than your initial mark. This penalty equates to up to 30% of the automark for that iteration. Note that if the re-run automark after penalty is lower than your initial mark, we will keep your initial mark, meaning your automark cannot decrease after a re-run. E.g. if your initial mark is 50% and on re-run you get 70%, your mark may still be capped at 50%. If you want to have your automarking re-run, branch off the submission commit, make the necessary changes, and push them to another branch and share the name of the branch with your tutor.

7.2. Demonstration

For the demonstrations in weeks 5 and 8, all team members **must** attend this lab session, or they will not receive a mark. If you are unable to attend the session, you must apply for special consideration.

Demonstrations take place during lab time and consist of a 15 minute Q&A in front of your tutor and maybe some other students in your tutorial. For online classes, webcams are required to be on during this Q&A (your phone is a good alternative if your laptop/desktop doesn't have a webcam).

8. Individual Contribution

While we do award a tentative mark to your group as a whole, your actual mark for each iteration is given to you individually. Your individual mark is determined by your tutor, with your group mark as a reference point. Your tutor will look at the following items each iteration to determine your mark:

- Project check-in
- Code contribution
- Tutorial contributions
- Peer assessment

In general, all team members will receive the same mark (a sum of the marks for each iteration), **but if you as an individual fail to meet these criteria, your final project mark may be scaled down**, most likely quite significantly.

8.1. Project check-in

During your lab class, in weeks without project demonstrations, you and your team will conduct a short stand-up in the presence of your tutor. Each member of the team will briefly state what they have done in the past week, what they intend to do over the next week, and what issues they faced or are currently facing. This is so your tutor, who is acting as a representative of the client, is kept informed of your progress. They will make note of your presence and may ask you to elaborate on the work you've done.

Project check-ins are also excellent opportunities for your tutor to provide you with both technical and non-technical guidance.

Your attendance and participation at project check-ins will contribute to your individual mark component for the project.

These are easy marks. They are marks assumed that you will receive automatically, and are yours to lose if you neglect them.

8.2. Tutorial contributions

From weeks 2 onward, your individual project mark may be reduced if you do not satisfy the following:

- Attend all tutorials
- Participate in tutorials by asking questions and offering answers
- [online only] Have your web cam on for the duration of the tutorial and lab

We're comfortable with you missing or disengaging with 1 tutorial per term, but for anything more than that please email your tutor. If you cannot meet one of the above criteria, you will likely be directed to special consideration.

These are easy marks. They are marks assumed that you will receive automatically, and are yours to lose if you neglect them.

8.3. Code contribution

All team members must contribute code to the project to a generally similar degree. Tutors will assess the degree to which you have contributed by looking at your **git history** and analysing lines of code, number of commits, timing of commits, etc. If you contribute significantly less code than your team members, your work will be closely examined to determine what scaling needs to be applied.

Note that, **contributing code is not a substitute for not contributing more documentation**.

8.4. Documentation contribution

All team members must contribute documentation to the project to a generally similar degree.

In terms of code documentation, your functions such as `authRegister`, `channelInvite`, `messageSend`, etc. are also required to contain comments of the following format:

```
<Brief description of what the function does>

Arguments:
```

```
    <name> (<data type>)    - <description>
    <name> (<data type>)    - <description>
    ...

Return Value:
    Returns <return value> on <condition>
    Returns <return value> on <condition>
```

In each iteration you will be assessed on ensuring that every relevant function in the specification is appropriately documented.

In terms of other documentation (such as reports and other notes in later iterations), we expect that group members will contribute equally.

Note that, **contributing more documentation is not a substitute for not contributing code**.

8.5. Peer Assessment

At the end of each iteration, there will be a peer assessment survey where you will rate and leave comments about each team member's contribution to the project up until that point.

Your other team members will **not** be able to see how you rated them or what comments you left in either peer assessment. If your team members give you a less than satisfactory rating, your contribution will be scrutinised and you may find your final mark scaled down.

Iteration	Link	Opens	Closes
1	Click here	10pm Friday 24th June	9am Monday 27th June
2	Click here	10pm Friday 15th July	9am Monday 18th July
3	Click here	10pm Friday 5th August	9am Monday 8th August

8.6. Managing Issues

When a group member does not contribute equally, we are aware it can implicitly have an impact on your own mark by pulling the group mark down (e.g. through not finishing a critical feature), etc.

The first step of any disagreement or issue is always to talk to your team member(s) on the chats in MS teams. Make sure you've been:

1. Been clear about the issue you feel exists
2. Been clear about what you feel needs to happen and in what time frame to feel the issue is resolved
3. Gotten clarity that your team member(s) want to make the change.

If you don't feel that the issue is being resolved quickly, you should escalate the issue by talking to your tutor with your group in a project check-in, or alternatively by emailing your tutor privately outlining your issue.

It's imperative that issues are raised to your tutor ASAP, as we are limited in the mark adjustments we can do when issues are raised too late (e.g. we're limited with what we can do if you email your tutor with iteration 2 issues after iteration 2 is due).

9. Automarking & Leaderboard

9.1. Automarking

Each iteration consists of an automarking component. The particular formula used to calculate this mark is specific to the iteration (and detailed above).

When running your code or tests as part of the automarking, we place a 2.5 minute timer on the running of your group's tests. This is more than enough time to complete everything unless you're doing something very wrong or silly with your code. As long as your tests take under 2.5 minutes to run on the pipeline, you don't have to worry about it potentially taking longer when we run automarking.

9.2. Leaderboard

In the days preceding iterations 1, 2, and 3's due date, we will be running your code against the actual automarkers (the same ones that determine your final mark) and publishing the results of every group on a leaderboard. The leaderboard will be available here once released.

The leaderboard will be run on Monday, Wednesday, and Friday lunchtime during the week that the iteration is due.

Your position and mark on the leaderboard will be referenced against an alias for your group (for privacy). This alias will be emailed to your group in week 3. You are welcome to share your alias with others if you choose! (Up to you.)

The leaderboard gives you a chance to sanity check your automark (without knowing the details of what you did right and wrong), and is just a bit of fun.

If the leaderboard isn't updating for you, try hard-refreshing your browser (Ctrl+R or Command+R), clearing your cache, or opening it in a private window. Also note the HTTP (not HTTPS) in the URL, as the site is only accessible via HTTP.

10. Plagiarism

The work you and your group submit must be your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such submissions.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct.

Do not provide or show your project work to any other person, except for your group and the teaching staff of COMP1531. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted you may be penalized, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

Note, you will not be penalized if your work has the potential to be taken without your consent or knowledge.