COMP1531 Major Project

Teamwork makes the [UNSW] Streams work

Contents

1. Aims:

- To provide students with hands on experience testing, developing, and maintaining a backend server in Python.
- To develop students' problem solving skills in relation to the software development lifecycle.
- Learn to work effectively as part of a team by managing your project, planning, and allocation of responsibilities among the members of your team.
- Gain experience in collaborating through the use of a source control and other associated modern team-based tools.
- Apply appropriate design practices and methodologies in the development of their solution
- Develop an appreciation for product design and an intuition of how a typical customer will use a product.

2. Overview

Please watch our introduction video here: * Part 1: About the project * Part 2: Getting started

To manage the transition from trimesters to hexamesters in 2021, UNSW has established a new focus on building an in-house digital collaboration and communication tool for groups and teams to support the high intensity learning environment.

Rather than re-invent the wheel, UNSW has decided that it finds the functionality of **Microsoft Teams** to be nearly exactly what it needs. For this reason, UNSW has contracted out Penguin Pty Ltd (a small software business run by Hayden) to build the new product. In UNSW's attempt to try and add a lighter note to the generally fatigued and cynical student body, they have named their UNSW-based product **UNSW Streams** (or just **Streams** for short). **UNSW Streams** is the communication tool that allows you to share, communication, and collaborate to (attempt to) make streams a reality.

Penguin Pty Ltd has sub-contracted two software firms:

- BlueBottle Pty Ltd (two software developers, Andrea and Andrew, who will build the initial web-based GUI)
- YourTeam Pty Ltd (a team of talented misfits completing COMP1531 in 21T3), who will build the backend Python server

In summary, UNSW contracts Penguin Pty Ltd, who sub contracts:

- BlueBottle (Andrea and Andrew) for frontend work
- YourTeam (you and others) for backend work

Penguin Pty Ltd met with Andrea and Andrew (the frontend development team) 2 weeks ago to brief them on this project. While you are still trying to get up to speed on the requirements of this project, Andrea and Andrew understand the requirements of the project very well.

Because of this they have already specified a **common interface** for the frontend and backend to operate on. This allows both parties 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.

3. Iteration 1: Basic Functionality and Tests

Now complete

4. Iteration 2: Building a Web Server

4.1. Task

NOTE: In merging the instructions for this iteration into your repo, you may get a failed pipeline. This is most likely because your code is not pylint compliant. If this is the case, that is the *first* thing you should address for this iteration. It is important you have a *stable* master branch before proceeding to add additional features.

In this iteration, more features were added to the specification, and the focus has been changed to HTTP endpoints. Many of the theory surrounding iteration 2 will be covered in week 4-5 lectures. Note that there will still be some features of the frontend that will not work because the routes will not appear until iteration 3.

In this iteration, you are expected to:

1. Make adjustments to your existing code as per any feedback given by your tutor for iteration 1.
2. Implement and test the HTTP Flask server according to the entire interface provided in the specification.
    - Part of this section may be automarked.
    - Your implementation should build upon your work in iteration 1, and ideally your HTTP layer is just a wrapper for underlying functions you've written

that handle the logic. Your implementation will rely on topics taught in week 4 (HTTP servers and testing) as well as week 5 (authentication and authorisation).

- Your implementation will need to implement persistence of data (see section 4.6).
- Ensure that you correctly manage sessions and tokens in terms of authentication and authorisation, as per requirements laid out in section 6.10.
- You can structure your tests however you choose, as long as they are appended with `_test.py`. For this iteration and iteration 3 we will only be testing your HTTP layer of tests. You may still wish to use your iteration 1 tests and simply wrap up them - that is a design choice up to you. An example of an HTTP test can be found in section 4.4.
- You do not have to rewrite all of your pytests as HTTP tests - the latter can test the system at a higher level. For example, to test a success case for `message/send` via HTTP routes you will need to call `auth/register` and `channels/create`; this means you do not need the success case for those two functions seperately. Your HTTP tests will need to cover all success/error conditions for each endpoint, however.

3. Comply with some additional expectations
    - Pylint has been added to your continuous integration file, meaning that code that isn't pylint compliant will now fail the pipeline. The provided `.pylintrc` file is *very* lenient, so there is no reason you should have to disable any additional checks.
    - Additionally, CI pipelines will measure *branch* coverage for all `.py` files that aren't tests. The coverage percentage for master is visible in a badge at the top of this repo and changes in coverage will appear in Merge Requests. Do note that coverage of `server.py` is not measured.
4. Continue demonstrating effective project management and effective git usage
    - You will be heavily marked for your use of thoughtful project management and use of git effectively. The degree to which your team works effectively will also be assessed.
    - As for iteration 1, all task tracking and management will need to be done via the GitLab Issue Board.
    - As for iteration 1, 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, 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`.

A frontend has been built by Andrea and Andrew that you can use in this iteration, and use your backend to power it (note: an incomplete backend will mean the frontend cannot work). You can, if you wish, make changes to the frontend code, but it is not required. The source code for the frontend is only provided for your own fun or curiosity.

As part of this iteration it is required that your backend code can correctly power the frontend. You should conduct acceptance tests (run your backend, run the frontend and check that it works) prior to submission.

4.2. Running the server

To run the server you should always use the command from the root directory of your project:

```
python3 -m src.server
```

This will start the server on the port in the src/config.py file.

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

If you get any errors relating to `flask_cors`, ensure you have installed all the necessary Python libraries for this course (the list of libraries was updated for this iteration). You can do this with:

```
pip3 install $(curl https://www.cse.unsw.edu.au/~cs1531/21T3/requirements.txt)
```

4.3. Implementing and testing features

You should first approach this project by considering its 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.py`). 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 Python.

Please pay careful attention to the following:

Your tests, keep in mind 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. * As is the case with any system or functionality, there will be some things that you can test extensively, some things that you can test sparsely/fleetingly, and some things that you can't meaningfully test at all. You should aim to test as extensively as you can, and make judgements as to what things fall into what categories.

4.4 Testing the interface

In this iteration, **the layer of abstraction has changed to the HTTP level**, meaning that you are only required to write integration tests that check the HTTP endpoints, rather than the style of tests you write in iteration 1 where the behaviour of the python functions themselves was tested.

You will need to check as appropriate for each success/error condition: * The return value of the endpoint; * The behaviour (side effects) of the endpoint; and * The status code of the response.

An example of how you would now test the echo interface is:

```
import pytest
import requests
import json
from src import config

def test_echo():
    '''
    A simple test to check echo
    '''
    resp = requests.get(config.url + 'echo', params={'data': 'hello'})
    assert json.loads(resp.text) == {'data': 'hello'}
```

4.5. Recommended approach

Our recommendation with this iteration is that you start out trying to implement the new functions similarly to how you did in iteration 1.

1. Write HTTP unit tests. These will fail as you have not yet implemented the feature.
    - Hint: It would be a good idea to consider good test design and the usage of helper functions for your HTTP tests. Is there a way so that you do not have to completely rewrite your tests from iteration 1?
2. Implement the feature and write the Flask route/endpoint for that feature too.
3. Run the tests and continue following 4.3. as necessary.

4.6. Storing data

→ save data to a file/database?

You are required to store data persistently in this iteration.

Modify your backend such that it is able to persist and reload its data store if the process is stopped and started again. The persistence should happen at regular intervals so that in the event of unexpected program termination (e.g. sudden power outage) a minimal amount of data is lost. You may implement this using whatever method of serialisation you prefer (e.g. pickle, JSON).

4.7. Versioning

You might notice that some routes are suffixed with `v1` and `v2`, and that all the new routes are `v1` yet all the old routes are `v2`. Why is this? When you make changes to specifications, it's usually good practice to give the new function/capability/route a different unique name. This way, if people are using older versions of the specification they can't accidentally call the updated function/route with the wrong data input.

Hint: Yes, your `v2` routes can use the `X_Y_v1` functions you had in iteration 1, regardless of whether you rename the functions or not. The layer of abstraction in iteration 2 has changed from the function interface to the HTTP interface, and therefore your 'functions' from iteration 1 are essentailly now just implementation details, and therefore are completely modifiable by you.

4.8. Dryrun

We have provided a very simple dryrun for iteration 2 consisting 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
```

4.9. Marking Criteria

Section	Weighting	Criteria

Automarking (Testing & Implementation)	50%	• 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
Code Quality	30%	• Demonstrated an understanding of good test **coverage** • Demonstrated an understanding of the importance of **clarity** on the communication test and code purposes • Demonstrated an understanding of thoughtful test **design** • Appropriate use of Python data structures (lists, dictionaries, etc.) • Appropriate style as described in section 8.4 • Appropriate application of good software design and Pythonic patterns • Implementation of persistent state
Git & Project Management	20%	• 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, with initiative to improve in future iterations • Effective use of course-provided MS Teams for communicating, demonstrating an ability to communicate and manage effectivelly digitally • Use of issue board on Gitlab to track and 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)

For this and for all future milestones, you should consider the other expectations as outlined in section 8 below.

The formula used for automarking in this iteration is:

```
Automark = 95*(t * i * min(c + 1, 100)^3) + 5*p
```

(Mark equals `t` multiplied by `i` multiplied by the maximum of `c + 1` and 100 to the power of three). This formula produces a value between 0 and 1.

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

4.10. Submission

This iteration due date and demonstration week is described in section 7. You will demonstrate this submission in line with the information provided in section 7.

5. Iteration 3: Coming Soon

Coming Soon

6. Interface specifications

These interface specifications come from Andrea and Andrew, who are building the frontend to the requirements set out below.

6.1. Input/Output types

Variable name	Type
named exactly **email**	string
has suffix **id**	integer
named exactly **length**	integer
contains substring **password**	string
named exactly **message**	string
contains substring **name**	string
has prefix **is_**	boolean
has prefix **time_**	integer (unix timestamp), [check this out](https://www.tutorialspoint.com/How-to-convert-Python-date-to-Unix-timestamp)

(outputs only) named exactly **messages**	List of dictionaries, where each dictionary contains types { message_id, u_id, message, time_created }
(outputs only) named exactly **channels**	List of dictionaries, where each dictionary contains types { channel_id, name }
has suffix **_str**	string
(outputs only) name ends in **members**	List of dictionaries, where each dictionary contains types of**user**
(outputs only) named exactly **user**	Dictionary containing u_id, email, name_first, name_last, handle_str
(outputs only) named exactly **users**	List of dictionaries, where each dictionary contains types of**user**
named exactly **token**	string
(outputs only) named exactly **dms**	List of dictionaries, where each dictionary contains types { dm_id, name }
named exactly **u_ids**	List of user ids

6.2. Interface

Name & Description	HTTP Method	Data Types	Exceptions

`auth/login/v2`
 	
Given a registered user's email and password, returns their `token` value.
	POST	**Parameters:**
		`{ email, password }`
		
		Return Type:
		`{ token, auth_user_id }`
			InputError when any of:
			• email entered does not belong to a user
			• password is not correct

`auth/register/v2`
 	
Given a user's first and last name, email address, and password, create a new account for them and return a new `token`.
 	
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	**Parameters:**
		`{ email, password, name_first, name_last }`
		
		Return Type:
		`{ token, auth_user_id }`
			InputError 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 name_first is not between 1 and 50 characters inclusive
			• length of name_last is not between 1 and 50 characters inclusive

`auth/logout/v1`
 	
Given an active token, invalidates the token to log the user out.
	POST	**Parameters:**
		`{ token }`
		
		Return Type:
		`{}`
			N/A

`channels/create/v2`
 	
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	**Parameters:**
		`{ token, name, is_public }`
		
		Return Type:
		`{ channel_id }`
			InputError when:
			• length of name is less than 1 or more than 20 characters

`channels/list/v2`
 	
Provide a list of all channels (and their associated details) that the authorised user is part of.
	GET	**Parameters:**
		`{ token }`
		
		Return Type:
		`{ channels }`
			N/A

`channels/listall/v2`
 	
Provide a list of all channels, including private channels, (and their associated details)
	GET	**Parameters:**
		`{ token }`
		
		Return Type:
		`{ channels }`
			N/A

`channel/details/v2`
 	
Given a channel with ID channel_id that the authorised user is a member of, provide basic details about the channel.
	GET	**Parameters:**
		`{ token, channel_id }`
		
		Return Type:
		`{ name, is_public, owner_members, all_members }`
			InputError when:
			• channel_id does not refer to a valid channel
			
			AccessError when:
			• channel_id is valid and the authorised user is not a member of the channel

`channel/join/v2`
 	
Given a channel_id of a channel that the authorised user can join, adds them to that channel.
	POST	**Parameters:**
		`{ token, channel_id }`
		
		Return Type:
		`{}`
			InputError when any of:
			• channel_id does not refer to a valid channel
			• the authorised user is already a member of the channel
			
			AccessError when:
			• channel_id 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/v2`
 	
Invites a user with ID u_id to join a channel with ID channel_id. Once invited, the user is added to the channel immediately. In both public and private channels, all members are able to invite users.
	POST	**Parameters:**
		`{ token, channel_id, u_id }`
		
		Return Type:
		`{}`
			InputError when any of:
			• channel_id does not refer to a valid channel
			• u_id does not refer to a valid user
			• u_id refers to a user who is already a member of the channel
			
			AccessError when:
			• channel_id is valid and the authorised user is not a member of the channel

`channel/messages/v2`
 	
Given a channel with ID channel_id 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	**Parameters:**
		`{ token, channel_id, start }`
		
		Return Type:
		`{ messages, start, end }`
			InputError when any of:
			• channel_id does not refer to a valid channel
			• start is greater than the total number of messages in the channel
			
			AccessError when:
			• channel_id is valid and the authorised user is not a member of the channel

`channel/leave/v1`
 	
Given a channel with ID channel_id 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	**Parameters:**
		`{ token, channel_id }`
		
		Return Type:
		`{}`
			InputError when:
			• channel_id does not refer to a valid channel
			
			AccessError when:
			• channel_id is valid and the authorised user is not a member of the channel

`channel/addowner/v1`

Make user with user id u_id an owner of the channel.

↳ need to be a member to become an owner

needs auth function and channel/create

POST

Parameters:
`{ token, channel_id, u_id }`

Return Type:
`{}`

InputError when any of:

- channel_id does not refer to a valid channel
- u_id does not refer to a valid user
- u_id refers to a user who is not a member of the channel
- u_id refers to a user who is already an owner of the channel

AccessError when:

- channel_id is valid and the authorised user does not have owner permissions in the channel

↳ token is not an owner

`channel/removeowner/v1`

Remove user with user id u_id as an owner of the channel.

↳ needs to be an owner to be removed

needs auth function and channel/create

POST

Parameters:
`{ token, channel_id, u_id }`

Return Type:
`{}`

InputError when any of:

- channel_id does not refer to a valid channel
- u_id does not refer to a valid user
- u_id refers to a user who is not an owner of the channel
- u_id refers to a user who is currently the only owner of the channel

AccessError when:

- channel_id is valid and the authorised user does not have owner permissions in the channel

↳ token is not an owner

`message/send/v1`

Send a message from the authorised user to the channel specified by channel_id. 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.

↳ need to keep track of the number of messages across all channels

needs auth function and channel/create

POST

Parameters:
`{ token, channel_id, message }`

Return Type:
`{ message_id }`

InputError when:

- channel_id does not refer to a valid channel
- length of message is less than 1 or over 1000 characters

AccessError when:

- channel_id is valid and the authorised user is not a member of the channel

↳ token is not a member

`message/edit/v1`

Given a message, update its text with new text. If the new message is an empty string, the message is deleted.

needs auth, channels/create, message/send and message/remove

PUT

Parameters:
`{ token, message_id, message }`

Return Type:
`{}`

InputError when any of:

- length of message is over 1000 characters ↳ deleted if < 1
- message_id does not refer to a valid message within a channel/DM that the authorised user has joined

AccessError when message_id refers to a valid message in a joined channel/DM and none of the following are true: (can't edit someone else's message)

- the message was sent by the authorised user making this request
- the authorised user has owner permissions in the channel/DM

↳ only owners have edit perm

`message/remove/v1`

Given a message_id for a message, this message is removed from the channel/DM

needs auth, channels/create, message/send

DELETE

Parameters:
`{ token, message_id }`

Return Type:
`{}`

InputError when:

- message_id does not refer to a valid message within a channel/DM that the authorised user has joined

AccessError when message_id refers to a valid message in a joined channel/DM and none of the following are true: (can't edit someone else's message)

- the message was sent by the authorised user making this request
- the authorised user has owner permissions in the channel/DM

↳ only owners have edit perm

`dm/create/v1`
 	
`u_ids` 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 list of user handles, e.g. 'ahandle1, bhandle2, chandle3'.
	POST	**Parameters:**
		`{ token, u_ids }`
		
		Return Type:
		`{ dm_id }`
			InputError when:
			• any u_id in u_ids does not refer to a valid user

`dm/list/v1`
 	
Returns the list of DMs that the user is a member of.
	GET	**Parameters:**
		`{ token }`
		
		Return Type:
		`{ dms }`
			N/A

`dm/remove/v1`
 	
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	**Parameters:**
		`{ token, dm_id }`
		
		Return Type:
		`{}`
			InputError when:
			• dm_id does not refer to a valid DM
			
			AccessError when:
			• dm_id is valid and the authorised user is not the original DM creator

`dm/details/v1`
 	
Given a DM with ID dm_id that the authorised user is a member of, provide basic details about the DM.
	GET	**Parameters:**
		`{ token, dm_id }`
		
		Return Type:
		`{ name, members }`
			InputError when:
			• dm_id does not refer to a valid DM
			
			AccessError when:
			• dm_id is valid and the authorised user is not a member of the DM

`dm/leave/v1`
 	
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	**Parameters:**
		`{ token, dm_id }`
		
		Return Type:
		`{}`
			InputError when:
			• dm_id does not refer to a valid DM
			
			AccessError when:
			• dm_id is valid and the authorised user is not a member of the DM

`dm/messages/v1`
 	
Given a DM with ID dm_id 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	**Parameters:**
		`{ token, dm_id, start }`
		
		Return Type:
		`{ messages, start, end }`
			InputError when any of:
			• dm_id does not refer to a valid DM
			• start is greater than the total number of messages in the channel
			
			AccessError when:
			• dm_id is valid and the authorised user is not a member of the DM

`message/senddm/v1`
 	
Send a message from authorised_user to the DM specified by dm_id. 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	**Parameters:**
		`{ token, dm_id, message }`
		
		Return Type:
		`{ message_id }`
			InputError when any of:
			• dm_id does not refer to a valid DM
			• length of message is less than 1 or over 1000 characters
			
			AccessError when:
			• dm_id is valid and the authorised user is not a member of the DM

`users/all/v1`
 	
Returns a list of all users and their associated details.
	GET	**Parameters:**
		`{ token }`
		
		Return Type:
		`{ users }`
			N/A

`user/profile/v1`
 	
For a valid user, returns information about their user_id, email, first name, last name, and handle
	GET	**Parameters:**
		`{ token, u_id }`
		
		Return Type:
		`{ user }`
			InputError when:
			
			• u_id does not refer to a valid user

`user/profile/setname/v1`
 	
Update the authorised user's first and last name
	PUT	**Parameters:**
		`{ token, name_first, name_last }`
		
		Return Type:
		`{}`
			InputError when any of:
			
			• length of name_first is not between 1 and 50 characters inclusive
			• length of name_last is not between 1 and 50 characters inclusive

`user/profile/setemail/v1`
 	
Update the authorised user's email address
	PUT	**Parameters:**
		`{ token, email }`
		
		Return Type:
		`{}`
			InputError 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/v1`
 	
Update the authorised user's handle (i.e. display name)
	PUT	**Parameters:**
		`{ token, handle_str }`
		
		Return Type:
		`{}`
			InputError when any of:
			
			• length of handle_str is not between 3 and 20 characters inclusive
			• handle_str contains characters that are not alphanumeric
			• the handle is already used by another user

`admin/user/remove/v1`
 	
Given a user by their u_id, remove them from the Streams. This means they should be removed from all channels/DMs, and will not be included in the list of users returned by users/all. Streams owners can remove other Streams 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 name_first should be 'Removed' and name_last should be 'user'. The user's email and handle should be reusable.
	DELETE	**Parameters:**
		`{ token, u_id }`
		
		Return Type:
		`{}`
			InputError when any of:
			
			• u_id does not refer to a valid user
			• u_id refers to a user who is the only global owner
			
			AccessError 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 permission_id.
	POST	**Parameters:**
		`{ token, u_id, permission_id }`
		
		Return Type:
		`{}`
			InputError when any of:
			
			• u_id does not refer to a valid user
			• u_id refers to a user who is the only global owner and they are being demoted to a user
			• permission_id is invalid
			
			AccessError when:
			
			• the authorised user is not a global owner

`clear/v1` Resets the internal data of the application to its initial state	DELETE	**Parameters:** `{}` **Return Type:** `{}`	N/A

6.3. Errors for all functions

Either an `InputError` or `AccessError` is thrown when something goes wrong. All of these cases are listed in the **Interface** table. If input implies that both errors should be thrown, throw an `AccessError`.

One exception is that, even though it's not listed in the table, for all functions except `auth/register`, `auth/login`, `auth/passwordreset/request` (iteration 3) and `auth/passwordreset/reset` (iteration 3), an `AccessError` is thrown when the token passed in is invalid.

6.4. Valid email format

A valid email should match the following regular expression:

```
'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}$'
```

The Python `re` (regular expression) module allows you to determine whether a string matches a regular expression. You do not need to understand regular expressions to effectively utilise the `re` module to check if the email is correct. Check this link if you need more help.

6.5. 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.

6.6. Pagination

The behaviour in which channel_messages 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 `token` "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: * `channel_messages("12345", 6, 0) => { [messages], 0, 50 }` * `channel_messages("12345", 6, 50) => { [messages], 50, 100 }` * `channel_messages("12345", 6, 100) => { [messages], 100, -1 }`

Pagination should also apply to messages in DMs.

6.7. Permissions

- Members in a channel/DM have one of two channel/DM permissions 1) Owner of the channel/DM 2) Members of the channel/DM
- Streams 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 Streams users are members by default, except for the very first user who signs up, who is an owner

A user's primary permissions are their global permissions. Then the channel/DM permissions are layered on top. For example: * An owner of Streams 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 adds them as one, meaning if they are removed as a global owner, they will no longer have those channel owner permissions. * Streams 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 Streams is a member in channels they are not owners of * A member of Streams is an owner in channels they are owners of

6.8. Token

Many of these functions (nearly all of them) need to be called from the perspective of a user who is logged in already. When calling these "authorised" functions, we need to know: 1) Which user is calling it 2) That the person who claims they are that user, is actually that user

We could solve this trivially by storing the user ID of the logged in user on the frontend, and every time the frontend (from Andrea and Andrew) calls your background, they just sent a user ID. This solves our first problem (1), but doesn't solve our second problem! Because someone could just "hack" the frontend and change their user id and then log themselves in as someone else.

To solve this when a user logs in or registers the backend should return a "token" (an authorisation hash) that the frontend will store and pass into most of your functions in future. When these "authorised" functions are called, those tokens returned from register/login will be passed into those functions, and from there you can check if a token or token is valid, and determine the user ID.

Passwords must be stored in an encrypted form, and tokens must use JWTs (or similar).

6.9. Working with the frontend

There is a SINGLE repository available for all students at https://gitlab.cse.unsw.edu.au/COMP1531/21T3/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 flask 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.

6.9.1. Example implementation

A working example of the frontend can be used at http://streams-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.

6.9.2. Error raising for the frontend

For errors to be appropriately raised on the frontend, they must be raised by the following:

```
if True: # condition here
    raise InputError(description='Description of problem')
```

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

The types in error.py have been modified appropriately for you.

6.10. 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 determine whether it's really you or not.

This notion of a session is explored in the authentication (Hashing) & authorisation (JWT), and is an expectation that it is implemented in iteration 2 and beyond.

For iteration 2 and beyond, we also expect you to handle multiple concurrent 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.

7. Due Dates and Weightings

Iteration	Due date	Demonstration to tutor(s)	Assessment weighting of project (%)
1	10am Monday 4th October (**week 4**)	In YOUR **week 4** laboratory	25%
2	10am Monday 25th October (**week 7**)	In YOUR **week 7** laboratory	40%
3	10am Monday 15th November (**week 10**)	In YOUR **week 10** laboratory	35%

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 listed.

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

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

For example:

```
$ 1531 submit iteration1 W11A_EAGLE
```

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.

7.2. Demonstration

For the demonstrations in weeks 4, 7, and 10, all team members **must** attend this lab session, or they will not receive a mark.

When you demonstrate this iteration in your lab time, it will 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. Other Expectations

While it is up to you as a team to decide how work is distributed between you, for the purpose of assessment there are certain key criteria all members must follow.

- Code contribution
- Documentation contribution
- Usage of git/GitLab
- Attendance
- Peer assessment
- Academic conduct

The details of each of these is below.

While, in general, all team members will receive the same mark (a sum of the marks for each iteration), **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.

8.2. Code Style and Documentation

You are required to ensure that your code: * Follows Pythonic principles discussed in lectures and tutorials * Follows stylistic convenctions discussed in lectures and tutorials * (For iterations 2 & 3) your code should achieve a `10.00/10 pylint` score

Examples of things to focus on include: * Correct casing of variable, function and class names * Meaningful variable and function names * Readability of code and use of whitespace * Modularisation and use of helper functions where needed

Your functions such as `auth/register`, `channel/invite`, `message/send`, etc. are also required to contain docstrings of the following format:

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

Arguments:
    <name> (<data type>)    - <description>
    <name> (<data type>)    - <description>
    ...

Exceptions:
    InputError  - Occurs when ...
    AccessError - Occurs when ...

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/endpoint in the specification is appropriately documented.

8.3. Individual Marks

Whilst 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 your code contribution, documentation contribution (for iteration 3), peer assessment results, any other issues raised by group members throughout term, and your attendance at project check-ins and demonstrations.

8.3.1. 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.

8.3.2. Documentation contribution

All team members must contribute documentation to the project to a generally similar degree. Tutors will assess the degree to which you have contributed by looking at your **git history** but also **asking questions** (essentially interviewing you) during your demonstration.

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

8.3.3. 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.

8.3.4. Attendance

It is generally assumed that all team members will be present at the demonstrations and at weekly check-ins. If you're absent for more than 80% of the weekly check-ins or any of the demonstrations, your mark may be scaled down.

If, due to exceptional circumstances, you are unable to attend your lab for a demonstration, please apply for special consideration.

9. 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 minutes timer on the running of your groups tests and implementation. This is more than enough time to complete everything unless you're doing something very wrong or silly with your code.

Once you receive your results for each iteration, your mark may be lower than expected. If you find this is due to many marks being lost due to one or two trivial, systematic bugs, you are welcome to branch off the submission commit, make some changes, and push to another branch. If you share this branch with your tutor and ask them to rerun it, they can rerun it for you. Your new automarking mark will be your new mark with a 20% penalty (of the total automark mark) for the change.

Prior to the iterations being due, 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. Your position and mark on the leaderboard will be referenced against an alias for your group (for privacy). By now you should know your alias.

The automarker for the leaderboard will be run at 10am on Tuesday 19th, Thursday 21st, and Saturday 23rd of October. This means the leaderboard will only be updated **3 times** prior to your submission.

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.

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.

Function progression:

(a)
1. auth/login/v2
2. auth/register/v2
3. auth/logout/v2

(b)
4. channels/create/v2
5. channels/list/v2
6. channels/listall/v2

(c)
7. channel/details/v2
8. channel/join/v2
9. channel/invite/v2
10. channel/messages/v2
11. channel/leave/v1
12. channel/addowner/v1
13. channel/removeowner/v1

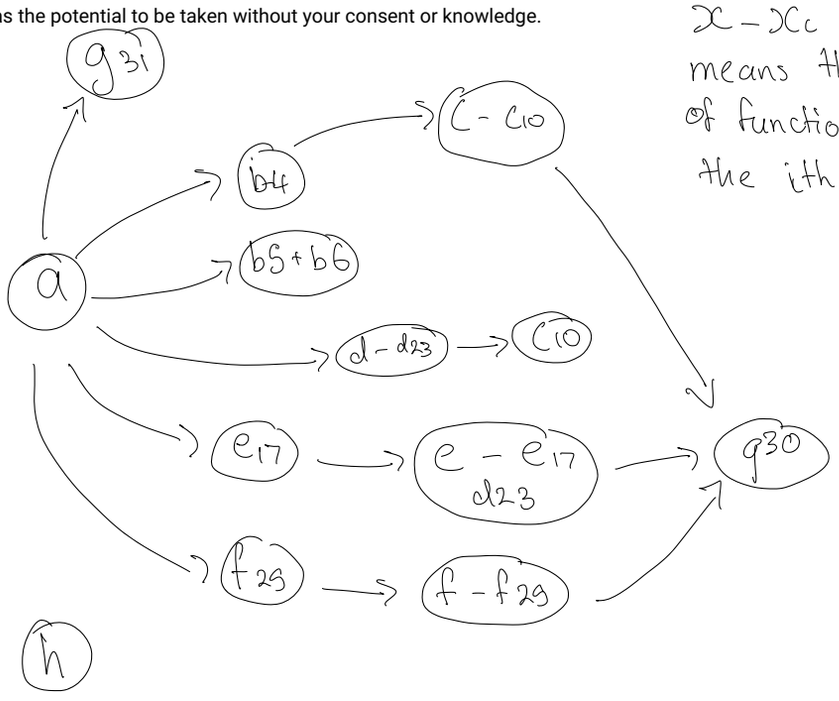
(d)
14. message/send/v1
15. message/edit/v1
16. message/remove/v1
28. message/senddm/v1

(e)
17. dm/create/v1
18. dm/list/v1
19. dm/remove/v1
20. dm/details/v1
21. dm/leave/v1
22. dm/messages/v1

(f)
24. users/all/v1
25. user/profile/v1
26. user/profile/setname/v1
27. user/profile/setname/v1
28. user/profile/setemail/v1
29. user/profile/sethandle/v1

(g)
30. admin/user/remove/v1
31. admin/userpermission/change/v1

(h)
32. clear/v1



x - xc means that group of functions excluding the ith function