Chat App

Contents

[2 Analysis 2](#_Toc191551332)

[2.1 Problem Area 2](#_Toc191551333)

[2.2 Prospective users 2](#_Toc191551334)

[2.3 Research of existing systems 3](#_Toc191551335)

[2.3.1 WhatsApp 3](#_Toc191551336)

[2.3.2 Telegram 3](#_Toc191551337)

[2.4 Proposed Features of My Chat App 3](#_Toc191551338)

[2.5 Prospective Client Interviews 3](#_Toc191551339)

[2.5.1 Interview 1: Christopher Patrick Bacon 3](#_Toc191551340)

[2.5.2 Interview 2: Gilmore T. Azell 4](#_Toc191551341)

[2.6 Final Objective List 5](#_Toc191551342)

[3 Design 5](#_Toc191551343)

[3.1 Technologies Used in Development 5](#_Toc191551344)

[3.2 User Interface Mock-ups 6](#_Toc191551345)

[3.2.1 Sign In Interface 6](#_Toc191551346)

[3.2.2 Sign Up Interface 6](#_Toc191551347)

[3.2.3 Index Page (will show list of group chats and currently selected chat window) 7](#_Toc191551348)

[3.2.4 Group Options Panel 8](#_Toc191551349)

[3.2.5 About Page 8](#_Toc191551350)

[3.2.6 Invites Panel (Received Invites) 9](#_Toc191551351)

[3.2.7 Invites Panel (Outgoing Invites) 9](#_Toc191551352)

[3.2.8 Create Group Page 9](#_Toc191551353)

[3.3 Proposed solution 10](#_Toc191551354)

[3.3.1 Client-side 10](#_Toc191551355)

[3.3.2 Server-side 10](#_Toc191551356)

[3.3.3 Database Management 11](#_Toc191551357)

[3.4 Application Structure 11](#_Toc191551358)

[3.4.1 File Structure 12](#_Toc191551359)

[3.4.2 Detailed overview 12](#_Toc191551360)

[3.4.3 Model (/database\_operations) 14](#_Toc191551361)

[3.5 Database Design 15](#_Toc191551362)

[3.6 MODELS.PY 17](#_Toc191551363)

[3.6.1 Class: UserTable 17](#_Toc191551364)

[3.6.2 Class: GroupTable 24](#_Toc191551365)

[3.6.3 Class: UserGroupTable 30](#_Toc191551366)

[3.6.4 Class: MessageTable 31](#_Toc191551367)

[3.6.5 Class: InviteRequestTable 34](#_Toc191551368)

[3.7 Routes.py 39](#_Toc191551369)

[3.7.1 List of APIs 40](#_Toc191551370)

[3.7.2 List of Webpage Routes 47](#_Toc191551371)

# Analysis

## Problem Area

At my school, the only way for Sixth Formers to communicate with each other on the school’s network is via email. However, there are a number of issues with this. For example, when email chains become very long, they can be difficult to navigate. When you have a lot of unread emails, spread across multiple email threads, such cluttered inboxes can be hard to manage. Emails are also generally expected to be quite formal, and there is demand among the Sixth Form for a more casual mode of online conversation.

There is strong demand within my school for a instant-messaging chat system for use by the Sixth Form that can be used to on the school’s network to fill this gap. A system such as this would be easier to use, and would have other benefits over email, such as being more aesthetically pleasing, could have presence indicators (to show when a user is online/offline), the ability to create dedicated groups and would be easier to view past messages compared to email.

The aim of my project is to create an intuitive and visually appealing chat system for Sixth Formers for this purpose, enabling users to chat in real time, through chat groups, with a secure authentication system.

## Prospective users

The chat system is intended for the use of Sixth Formers in my school. There are around 20 classes (of around 10 people each) in Lower Sixth and Upper Sixth each, and the chat system will allow people to sign up with their name and form group – this will make it easier for other users to find each other on the system and chat with each other.

## Research of existing systems

### WhatsApp

WhatsApp is an instant messaging mobile, desktop and web app that also supports VoIP calling. It was created by WhatsApp Inc. in California and was first released on the App Store for iPhone in August 2009 yet was later acquired by Facebook (now Meta Platforms) in 2014. As of June 2024, WhatsApp has almost 3 billion unique users worldwide.

**High-level feature overview** (*at time of writing, not exhaustive*)**:**

* Users can sign up with a cellular mobile telephone number on the mobile app and are then able to set a username and profile picture. On mobile, users can set up biometric authentication to secure the app on devices that have the necessary hardware.
* The desktop and web apps mirror your phone’s chat and call functionality, although WhatsApp must first be set up on the mobile app.
* The app allows users to send text messages, voice messages, images, videos, documents, locations, emojis and stickers within chats, while users are also able to react to messages with emojis. Messages can be edited and deleted.
* Group chats can be created with up to 1024 members. Chat members with administrator permissions can manage the chat, control who can join, moderate the group and kick people from the chat. A name, description and image can be set for the group.
* The app also supports voice and video calling between individuals and group calls with up to 32 participants.
* All messages and calls are end-to-end encrypted to ensure user privacy and security.
* WhatsApp also provides several privacy settings giving users the ability to restrict sees their online status and last seen, limit who can view their picture and status, disable read receipts, send disappearing messages and more.
* The WhatsApp status lets users share photo, text, video and GIF updates that disappear after 24 hours and reply to others’ status updates through private messages.
* Users can back up settings, chats and media to cloud storage providers Google Drive or iCloud (if they already use them).

### Telegram

Telegraph is another popular instant messaging platform, which launched for iOS in August 2013 and Android in October that year. The company behind it was founded by Nikolai and Pavel Durov. Company data indicates that 950 million use the software across the world.

**High-level feature overview** (*at time of writing, not exhaustive*)**:**

* Registration requires a smartphone or one of a few NFTs issued in December 2022.
* Similar messaging features to WhatsApp, alongside ‘silent messages’ where users can send messages that do not trigger a notification.
* A single group chat can support up to 200,000 members, with similar features to WhatsApp. Group analytics provide statistics for group administrators to monitor activity and engagement.
* Voice and video call support similar to WhatsApp. A single voice chat can accommodate thousands of listeners, while a group video call can support up to a thousand video viewers.
* Telegram channels (which can private, with restricted access, or public, allowing anyone to join) support sending messages to an unlimited number of subscribers.
* Web, mobile and desktop apps.
* Telegram stands out for its privacy and security features including end-to-end encryption, secret chats (messages that can self-destruct and cannot be forwarded), self-destructing media, two-step verification, username-based communication and anonymous admins.
* Personalisation including custom themes, notifications and chat folders.
* Developers can create custom chatbots.
* Cloud backup to Telegram’s cloud, except for secret chats.

## Proposed Features of My Chat App

I intend to a make a relatively simple chat app that will allow users to sign up with a unique username, display name (this will not have to be unique), their form group and a password (this will need to meet some password strength criteria). Users will need to login to authenticate themselves and will be able to logout. For convenience, session cookies will be used to ‘remember’ users and authenticate them if they have logged in recently. Users will be able to able to view sign in, sign up and personal profile pages (to edit their own details and delete their account).

Group chats will be the core of messaging functionality. Users will be able to create, join and leave group chats. To ensure privacy, to join a group chat, users must first receive an invite request from an existing member of the group chat and can then accept or reject the request. There will be an invites panel, so users can keep track of outgoing and received invitation history and status. Within the group chat, users will be able to send text messages of up to 2000 characters at a time.

To keep things fair, there will be no administrator/elevated permissions for different users functionality for each group as one of the key principles behind this application is all students should have equal permissions, although users must first be invited to a chat for privacy. This will also allow students to run chats themselves. Groups will also have group information pages, detailing group name, members, other admin information and will provide an option for users to leave the group.

## Prospective Client Interviews

### Interview 1: Christopher Patrick Bacon

Christopher is a member of my school’s Sixth Form who is very frustrated by the lack of a chat system for use within school among our year group – he is very keen to see an application to meet his needs.

1. What features would you like to see on this chat application?

*The app would obviously need the basics, such as a user authentication system to sign up, sign in and sign out. Fast real-time messaging and the ability to create group chats are also important, and it would be great to be able to only invite specific people you want to be in your groups. There would also need to be a way of accepting these invitations. A visually appealing user interface is also important, of course.*

1. Do you prefer apps with a clean and minimalistic design, or those with more features and customization options?

*I personally prefer those with a more clean and minimalistic design. I personally, and I think other people who would use the app would agree with me on this, think that we’d rather only really use this in school, and that since there are feature-heavy apps people already use to communicate outside of school, people would be fine with something simpler to use in school. Plus, I think a cleaner and more minimalistic app would also be more visually appealing.*

1. How important is security and privacy to you when using a chat app?

*Security in terms of user accounts is very important – people don’t want their details leaked! No wants to use an unsafe app. In terms of privacy, the ability to create private chats and private group chats is number one. More broadly, the more privacy features, the better.*

1. Would you be interested in voice and/or video calling features?

*I wouldn’t be very interested in such features myself, most people I know prefer text-based communication to face-to-face interaction.*

1. Any other points from the user’s point of view?

*Some features, like a centralised group panel to see all invitations in one place would be nice. It would also be nice to see when people are online and offline – that way you know whether or not to expect a quick reply.*

### Interview 2: Gilmore T. Azell

1. What features would you like to see on this chat application?

*In the ideal chat app, I’d be able to create an account then login and logout, with the ability to create, join and leave different group chats (which can handle different numbers of users). Those are just the bare minimum – obviously other special features would be nice.*

1. Do you prefer apps with a clean and minimalistic design, or those with more features and customization options?

*The more features and customisation options the better – but I think the UI itself should be minimalistic. This would fit with modern apps, which are generally more minimalistic in general, and don’t overload you with loads of features all at once.*

1. How important is security and privacy to you when using a chat app?

*Security is important, I’d be content with just a basic login/logout system. I would also want peace of mind, knowing my account details are stored safely. Group chats should also be private.*

1. Would you be interested in voice and/or video calling features?

*Because I’d only really be using this app in school, I wouldn’t be that interested in voice and or calling. Most people prefer to text anyways.*

1. Any other points from the user’s point of view?

*An intuitive, user-friendly interface is a must. Customizable themes and a vast library of stickers and emojis would add a fun touch. Smart notifications, advanced search functionality, and integration with other apps would make communication even better.*

These interviews have shed light on what features users would appreciate and will help with creating a list of objectives for the software to meet. Because of time constraints, I may not be able to include every feature mentioned in the interviews.

## Final Objective List

1. Users should be able to sign up with details including a unique username, a display name, their class name and a password.
   1. The password should meet password strict strength criteria for security purposes
   2. Class names should be validated so they are in my school’s class name format, i.e. begin with “L6” or “U6” and end with 2 or 3 uppercase characters.
2. Users should be able to login and logout with their username and password.
3. Users should be able to create group chats and invite other users to join that group chat.
4. Users should be able to leave group chats.
5. Users should be able to easily navigate between group chats, with group chats with more recent activity being easier to access.
6. Users should be able to easily navigate between different parts of the app.
7. Users should be able to send text messages within the group chat.
8. Users should be able to view all the text messages within the group chat they are part of and who sent them.
9. Users should have and group invites panel, to be able to see received and outgoing invitation history and status (pending, accepted or rejected).
   1. Users should be able to accept or reject group chat invitations using this panel.
10. There should be persistent storage of user and chat data, in a database.
    1. Passwords should be stored as hashes, for security.
11. There should be an ‘About’ page which shows live counts of the number of users, groups, group invitations and messages and information about the application.

# Design

## Technologies Used in Development

As my primary code editor, I plan on using VSC, as it is a lightweight, modern editor with syntax highlighting. There are many extensions in VSC, that allow it be more compatible with different languages and frameworks – this will make my development process easier.

In order to interact with my database, I plan on using MySQL Workbench – according to the website, the software offers “MySQL Workbench delivers visual tools for creating, executing, and optimizing SQL queries. The SQL Editor provides colour syntax highlighting, auto-complete, reuse of SQL snippets, and execution history of SQL.”

For version control, and in order to backup my codebase, I plan on using a GitHub repository. I will use GitHub Desktop for committing my changes to the repository – the GUI it provides is simpler to use than the command-line tool.

To view and test my application, I will use Google Chrome – this a modern web browser used by the majority of Internet users.

## User Interface Mock-ups

In order to style my application, and create a user-friendly interface, I will be using CSS and HTML (along with Jinja templating to reduce code repetition). I will post screenshots of my GUI.

To make my application, I will create a navbar to allow my user to navigate between different sections of my application. I have used a ‘dark theme’ aesthetic to go easy on users’ eyes, and I have gone for a modern and minimalistic theme.

### Sign In InterfaceA screenshot of a computer AI-generated content may be incorrect.

### Sign Up Interface

A screenshot of a computer

AI-generated content may be incorrect.

### Index Page (will show list of group chats and currently selected chat window)A screenshot of a computer AI-generated content may be incorrect.

### Group Options PanelA screenshot of a computer AI-generated content may be incorrect.

### About Page

A screenshot of a chat app

AI-generated content may be incorrect.

### Invites Panel (Received Invites)A screenshot of a computer AI-generated content may be incorrect.

### A screenshot of a computer AI-generated content may be incorrect.Invites Panel (Outgoing Invites)

### Create Group Page

A screenshot of a computer

AI-generated content may be incorrect.

## Proposed solution

### Client-side

Given the ubiquity of modern smartphones among teenagers, I first considered a mobile app. However, not everyone is able to download the certificate necessary to connect to the school’s internet, meaning many are dependent on mobile data for an Internet connection within school. However, there are several areas within school with very weak signal (there is a courtyard with high walls around them, in which download speeds are extremely slow) meaning a mobile app would be constrained by this issue.

The other device that every pupil in my school has access to is a school-issue laptop. I decided against a desktop application – the laptops have limited storage space, and people are not used to downloading and installing applications outside of traditional app stores such as the Microsoft Store – the admin required to get your app on such stores is also cumbersome.

These issues make a **web-based application** (optimised for usage on desktop computers and laptops) the ideal solution – the frontend will be completed using HTML and CSS, which I am familiar with.

There are many features of my application that require data to be sent from clients and the server in real-time, such as the messaging and invites to groups. If a user had to reload the page in order to check for new messages, this would obviously not be ideal. Therefore, I will use **JavaScript** (and its Fetch API) to interact with API endpoints on the server to ‘fetch’ this data and update it on the user’s webpage in real time, providing a much better user experience.

### Server-side

The chat app will use a server rather than letting users message each other directly, as this will allow me to centralise the storage of all data and make it easier to implement group chat functionality.

Given my experience with it, I decided to choose **Python** as the server-side programming language. Although there are other popular web frameworks for Python, the ones most widely used and with the most documentation (important for a beginner such as me) are Flask and Django. While I have some experience with Django, although its ‘batteries-included’ approach to web development means that development is fast and easily scalable, ‘opting out’ of some of those built-in features such as the database object-relational mapper (ORM) can unnecessarily complicate your project. In essence, the fact that Django comes with a lot of ‘built-in’ features means that using all of these features together streamlines the development timeline. However, some of the features abstract much of the complexity away, meaning using some features such as the ORM would not allow you to implement the complex algorithms (such as cross-table parameterised SQL in group A) necessary for scoring high marks. Trying to use Django without the ORM requires altering writing the rest of the project in an unorthodox and what I believe is an overcomplicated format that removes the benefits of fast development that Django’s ‘batteries-included’ approach normally provides.

Instead, I chose **Flask**. Flask is a lightweight and flexible micro web framework for Python. While it lacks built-in features such as a database abstraction layer (such as Django’s ORM), Flask supports ‘extensions’ that add features as if they were implemented in Flask itself. The beauty of this approach is that the developer can choose where to use external libraries and where they would prefer to implement the logic themselves. This will allow me to write the more easily write the complex algorithms at ‘Group A’ level to access higher marks.

Alongside this, I will be using the Jinja templating language – this allows you to create HTML pages that extend preexisting HTML pages, reducing the amount of code repetition significantly.

### Database Management

For persistent storage of user and chat data, I will use a relational database management system as this provides several advantages over a file management system including a locking system for concurrency and consistency control, much faster read/write and search times and built-in security.

Given my prior experience with SQL and that many of the example Group A algorithms mention SQL, a SQL relational database management system (RDBMS) is the obvious solution to handle my database needs. In the context of a chat system for a Sixth Form of around 400 pupils, most SQL RDBMS should be able to support a load of this level. Another constraint was my lack of a budget – this meant I was restricted to open-source relational database management systems.

The first option I considered was SQLite, a simple and lightweight RDBMS. A major advantage was its small footprint – SQLite takes up very little disk space and memory, potentially enabling me to choose cheaper hardware for the server later on. It also requires little setup time as it is serverless. On the other hand, one limitation with SQLite is its scalability, and may be less suitable for applications with high concurrency and many users reading and writing from the database at the same time. Another issue is that SQLite lacks a built-in user authentication system, although you can set file permissions to the .db file in which the database is stored.

Instead, I chose **MySQL**, a RDBMS that is very fast and scalable. In addition, MySQL supports user management for direct access to the database, making the server itself more secure, while it does not take very long to setup (although still longer than it takes to set up SQLite).

I also considered PostgreSQL, another fast and scalable RDBMS. However, I felt that it had poorer memory performance than MySQL, although this difference would largely be unnoticeable at the level of a small project such as mine. I decided to use MySQL as I had more experience with it.

Python is not directly compatible with any RDBMSs, so I will need a library to interface the Flask application server with the database server. For this, I will be using **SQLAlchemy** – I will not be using its ORM, but I will be using its engine to interface with the MySQL database server and execute and commit SQL queries.

## Application Structure

My application will use the **Model-View-Controller** (**MVC**) **Architecture**. This architecture divides the application into 3 sections: the controller, the view and the model. *See diagram – files and subfolders involved in each section are listed.*

* **Controller**: Contains the core logic and handles incoming requests and returns the response to the browser. This is part of the backend of the application.
* **View**: The view contains the UI for the application, with the HTML templates and static code involved in making this happen. The browser is then responsible for rendering the output. This is part of the frontend of the application.
* **Model**: The model is essentially an abstraction of the database. The controller will be able to interact with the database by using the model (which will handle all SQL) as an intermediary. This is part of the backend of the application.
* **Database**: Stores data – this is part of the backend of the application.

### File Structure

The image shows the file structure of my application – the following explains the structure I chose. Note that the \_\_pycache\_\_ subfolders are there to help Python run and is automatically created when you run the program – they just help the program start slightly faster.

* The **database\_operations** subfolder contains the ‘model’ part of the application.
* A screenshot of a computer

  AI-generated content may be incorrect.The **static** subfolder contains subfolders called **js**, **css** and **img**. The ‘js’ folder contains the JavaScript for each of the HTML templates, while the ‘css’ subfolder contains the css styling code for those templates. Separating the JavaScript and CSS into separate files will allow me to decompose the separate parts of the ‘view’ templates into more readable chunks.
* The **templates** subfolder contains the HTML templates as part of the ‘view’ part of the application.
* The files ‘auth.py’, ‘forms.py’, ‘routes.py’, ‘config.py’, ‘main.py’ and ‘\_\_init\_\_.py’ are part of the ‘controller’ part of the application and handle core logic.
* The ‘Written Documents’ folder contains this Word document and is not relevant to the code.
* The README.md is there for the GitHub repository which I have been using to store the code and is not relevant to the code either.

### Detailed overview

#### Controller

* A diagram on a piece of paper

  AI-generated content may be incorrect.The controller part of the code is largely handled by ‘routes.py’, which contains functions to handle all the HTTPS requests from the browser for different parts of the app. When a request is received, a specific template is returned from the /templates subfolder. All of the templates have css stored in the /css folder which improves the styling of the application.
  + ‘routes.py’ also handles API POST requests from parts of the application which require the real-time updating of dynamic content on the website. There are many parts of the application where it is fine if the user has to reload the page to view updated data, but in most cases the user will need to see the latest data, at all times – it would be inconvenient to have the reload the page every second to see the latest chat messages in the middle of a conversation.
  + Therefore, the parts of the application which require this have templates which include JavaScript (stored in the /js subfolder) which contain code to send POST requests to the application from the client-side (and handled in ‘routes.py’ on the server-side) and render the data on the webpage (this is why the arrow between ‘routes.py’ and ‘/templates and /static’ is double-headed).
* The file ‘auth.py’ contains code to allow my application to interface with the Flask-Login extension, which handles session cookies as part of the user authentication system. The file contains a class ‘User’ which is wrapper for the static class ‘UserTable’ (this represents the User table in the database – more on this as part of the model section) to allow Flask-Login to interpret the users represented by ‘UserTable’.
* The file ‘forms.py’ contains four form classes, which inherit from FlaskForm, part of the flask\_wtf library which I am using to simplify the frontend of implementation of forms within my application. Each form class contains attributes which represent the form fields, which flask\_wtf will render in templates. Form classes:
  + SignInForm: Form for allowing users to sign in – will include username, password, remember\_me and submit fields.
  + SignUpForm: Form for allowing users to sign up – will include username, display\_name, form\_group (i.e. the class the user is in), password, password2 (validation field to ensure user has typed the password correctly) and submit fields.
  + BubbleListField: This is a special field/UI component I am implementing as part of the BubbleForm. It will involve a box and a text input component, with an ‘add user’ button next to it. When a user types a username in the text input component and clicks on the ‘add user’ button, the username will appear as a bubble within the box. The user will be able to click on the bubble to remove it. Once submit is clicked on the bubble form
  + BubbleForm: a form which will allow users to add users to a list of users which will be invited to a new group – will include group\_name and bubble\_list fields. A separate component will be added to allow users to submit the group within the HTML itself – this will allow me to more easily customise the appearance of the button.
* The file ‘\_\_init\_\_.py’ is needed to initialise the Flask application. The flask module, the LoginManager is imported from the flask\_login extension and the routes.py and auth.py files are also imported to initialise the application. ‘sign\_in’ is specified as being the custom route name for the login\_view – this is a parameter required for the flask\_login to function correctly.
* The file ‘config.py’ contains the application’s ‘SECRET\_KEY’ – a string known only known by the server administrator for security purposes.
* The file ‘main.py’ only imports the ‘app’ Flask object created in ‘\_\_init\_\_.py’ – this is boilerplate needed for the Flask application to function correctly.

### Model (/database\_operations)

* The file ‘database\_connection.py’ imports a library called ‘sqlalchemy’ which is used to connect to the database server (I will be hosting this locally on my computer). It creates an engine (this is an object) with a connection\_string containing parameters such as the database name, username, password, port, RDBMS language and database driver (I am using ‘pymysql’). The engine object can then be used to execute parameterised SQL queries.
  + - A diagram of a software company

      AI-generated content may be incorrect.To simplify this process, the file includes a function ‘query\_db()’ which takes parameters including the parameter dictionary (all SQL queries will be parameterised to protect against SQL injection – this is the purpose of ‘parameter dictionary’), the actual raw SQL query and whether any rows are to be returned. This abstracts away the process of executing queries on the engine in different ways depending on different parameters, reducing the amount of code that needs to be repeated. The image below shows the query\_db() algorithm.
* The file ‘create\_tables.py’ imports the function ‘query\_db()’ and includes several functions to create each of the different tables (user, group, user\_group, message and invite\_request – more on database design later) in the database. The function ‘setup\_database()’ is used to create run each of the table creation functions together and turn on foreign key checks. This is the only file which is not run as part of the server. This file would need to be run separately on its own, before starting up the server.
* The file ‘models.py’ contains the bulk of the logic as part of the ‘model’ part of the application. The file contains five static classes (UserTable, GroupTable, UserGroupTable, InviteRequestTable), each of which represents its corresponding table in the database. Each class contains methods to perform CRUD operations on the table – UserGroupTable (link table to represent many-to-many relationship between the user and group tables) is slightly different as its methods are instead largely within UserTable and GroupTable as this simplifies the code when calling this methods elsewhere. For example, GroupTable.get\_number\_of\_users() makes more sense when reading the code than UserGroupTable.get\_number\_of\_users(). The classes will be unassociated.
* There is a file \_\_init.py\_\_ within the ‘/database\_operations’ subfolder – this does not contain any code. This is a part of python – ‘\_\_init\_\_.py’ files within directories are used to initialise Python packages. Without an \_\_init\_\_.py file, Python will not recognise the directory as a package, and you wouldn’t be able to import its submodules.

## Database Design

* I chose to use five tables:

1. user
2. group
3. user\_group
4. invite\_request
5. messageA black screen with white text

   AI-generated content may be incorrect.

* The ‘user’ table represents the user, and includes attributes username, display\_name, form\_group, datetime\_joined, password\_hash (passwords will be stored as hashes), is\_authenticated, is\_active and is\_anonymous (this field will not be used, but is needed for my application to be compatible with Flask-Login).
* The ‘group’ table represents chat groups, and includes attributes group\_id, group\_name and datetime\_created.
  + There is a many-to-many relationship between ‘user’ and ‘group’ as one user can be in many groups and one group can contain many groups. In order to both represent this many-to-many relationship and to ensure my database is in 3NF, I am using a link table called ‘user\_group’ – it uses a composite PK of ‘username’ and ‘group\_id’, both of which are also foreign keys.
* The table ‘message’ represents messages, and includes message\_id, message\_content (limited to 2000 characters), message\_date\_time, sender\_username and group\_id.
  + There is no receiver\_username field as messages will be tied to specific chat groups than specific users. In order to simplify my database, all chats will be represented as chat groups, even if there are only 2 users. This means that the only receiver of a chat message will be the group itself (group\_id is used as a foreign key). There is, however, a sender\_username (a foreign key from username of ‘user’ table), so users know who sent the message.
  + Relationships from ‘user’ to ‘message’ and ‘group’ to ‘message’ will be one-to-many.
* The table ‘invite\_request’ represents invites that users send to ‘invite’ people to a specific group chat. It has attributes request\_id, receiver\_username, sender\_username, group\_id, status and request\_date\_time.
  + The fields receiver\_username and sender\_username are foreign keys from the username field in the ‘user’ table. Although messages are only received by groups, it would be useful for receivers of invitations to groupchats to know who sent them, so a receiver\_username field has been added. There is an addition foreign key for group\_id – all invitations will be specific to one group.

* + The relationships from ‘invite\_request’ to ‘user’ and relationships from ‘invite\_request’ to ‘group’ will be one-to-many – one user can send/receive many invitations, but each invitation must be tied to one sender/receiver. Likewise, one group can have many invitations to it, but each invitation should be tied to one group.
  + The status field is a ‘VARCHAR(50)’ rather than ‘INT’ or ‘BOOLEAN’ – this is because it will be taking three values: ‘pending’, ‘accepted’ or ‘rejected’. I could have used ‘INT’ to minimise storage space taken up, and handled conversion within the application logic, but I decided that would have added needless complexity when I could instead ensure, within the application, the ‘status’ field was only set to one of those three values.
* I have largely decided to use ‘INT AUTO\_INCREMENT’ for my primary keys. This means the database server will automatically generate a new integer primary key when a new row is created, one higher than the integer previously used for the previous row created (regardless of whether rows are deleted) – this simplifies application logic.

## MODELS.PY

This is a comprehensive list of all the classes, methods and attributes I will be using to represent my database in the ‘model’ part of my application – this will all be part of the file models.py. SQL queries will be run using the query\_db function. Parameters within the SQL queries that begin with a colon, will be arguments taken from the function – this matches the syntax required by the SQLAlchemy engine.

### Class: UserTable

#### Attributes:

* INVALID\_FIELD\_VALUES = [None, ’’]

#### Methods:

* get\_number\_of\_users() -> int
  + **Docstring:**
  + """\_summary\_  
    Returns number of users in user table  
      
    Returns:  
     int: number of users in user table  
    """
  + SELECT COUNT(username) FROM user WHERE username = :username;
* [check\_username\_exists(username: str) -> bool](http://_vscodecontentref_/0)
  + **Docstring:**
  + """\_summary\_  
    Checks if username exists in user table  
      
    Args:  
     username (str): username to check  
      
    Returns:  
     bool: True if username exists, False otherwise  
    """
* [validate\_password(raw\_password: str) -> list](http://_vscodecontentref_/1)
  + **Docstring:**
  + """\_summary\_  
    Validates password based on criteria: Length must be at least 12 chars, must contain at least one numerical   
    digit, must contain at least one capital letter, must contain at least one lowercase letter, must contain at   
    least one special character in !@#$%^&\*()-\_+=[]{}|;:,.<>?/~`.  
      
    Args:  
     raw\_password (str): password to validate  
      
    Returns:  
     list: index 0: True if password is valid, False otherwise  
     index 1: list of reasons why password is invalid (if password is invalid)  
    """
* [check\_form\_group\_valid(form\_group: str) -> bool](http://_vscodecontentref_/2)
  + **Docstring:**
  + """\_summary\_  
    Checks if form\_group is valid  
      
    Args:  
     form\_group (str): form\_group to check  
      
    Returns:  
     bool: True if form\_group is valid, False otherwise  
    """
  + REGEX: “^(L6|U6)[A-Z]{2,3}$”
* [create\_user(username: str, display\_name: str, form\_group: str, raw\_password: str) -> None](http://_vscodecontentref_/3)
  + **Docstring:**
  + """\_summary\_  
    Creates a user in the user table - calling context must specify username, display\_name, form\_group, and raw\_password  
      
    Args:  
     username (str): username (PK) of user  
     display\_name (str): display name of user  
     form\_group (str): form\_group of user, must meet criteria specified by UserTable.check\_form\_group\_valid  
     raw\_password (str): raw password of user, must meet criteria specified by UserTable.validate\_password  
      
    Raises:  
     Exception: username, display\_name, form\_group, or raw\_password is empty  
     Exception: form\_group is invalid  
     Exception: password is invalid  
     Exception: username already exists  
      
    Returns:  
     None  
    """
  + INSERT INTO database1.user (username, display\_name, form\_group, password\_hash)VALUES (:username, :display\_name, :form\_group, :password\_hash);
* [get\_user\_record\_by\_username(username: str) -> dict](http://_vscodecontentref_/4)
  + **Docstring:**
  + """\_summary\_  
    Returns dictionary of user data for a given username  
      
    Args:  
     username (str): username (PK) of record in User table to be accessed  
      
    Raises:  
     Exception: username is empty  
     Exception: username not in table  
      
    Returns:  
     dict: dictionary of user data (username, display\_name, form\_group, datetime\_joined, password\_hash, is\_authenticated, is\_active, is\_anonymous)  
    """
  + SELECT \* FROM database1.user WHERE username = :username;
* [check\_user\_in\_group(username: str, group\_id: int) -> bool](http://_vscodecontentref_/5)
  + **Docstring:**
  + """\_summary\_  
    Checks if user with username is in group with group\_id  
      
    Args:  
     username (str): username (PK) of user  
     group\_id (int): group\_id (PK) of group  
      
    Returns:  
     bool: True if user is in group, False otherwise  
    """
  + SELECT COUNT(\*)

FROM database1.user\_group, database1.user, database1.group

WHERE user.username = :username

AND group.group\_id = :group\_id

AND user\_group.username = user.username

AND user\_group.group\_id = group.group\_id;

* [get\_number\_user\_groups(username: str) -> int](http://_vscodecontentref_/6)
  + **Docstring:**
  + """\_summary\_  
    Returns number of groups user is in  
      
    Args:  
     username (str): username (PK) of user to get number of groups for  
      
    Raises:  
     Exception: username is empty  
     Exception: username not in table  
      
    Returns:  
     int: number of groups user is in  
    """
  + SELECT COUNT(group\_id)

FROM database1.user\_group

WHERE username = :username;

* [get\_user\_groups(username: str) -> dict](http://_vscodecontentref_/7)
  + **Docstring:**
  + """\_summary\_  
    Returns dict of group\_id, group\_name, any\_messages, last\_message\_user\_display\_name, last\_message, last\_message\_datetime for all groups user is in  
      
    Args:  
     username (str): username (PK) of user to get groups for  
      
    Raises:  
     Exception: username is empty  
     Exception: username not in table  
      
    Returns:  
     dict: group\_id, group\_name, any\_messages, last\_message\_user\_display\_name, last\_message, last\_message\_datetime for all groups user is in  
     \nif any\_messages is False, last\_message\_user\_display\_name, last\_message, last\_message\_datetime will be None  
    """

A screenshot of a computer program

AI-generated content may be incorrect.

* [get\_pending\_invite\_requests(username: str) -> list](http://_vscodecontentref_/8)
  + **Docstring:**
  + """\_summary\_  
    Returns list of tuples of pending invite request records for a given username  
      
    Args:  
     username (str): username (PK) of user to get pending invite requests for  
      
    Raises:  
     Exception: username is empty  
     Exception: username not in table  
      
    Returns:  
     list: list of all pending invite request records for specified username"""
  + SELECT

invite\_request.request\_id,

invite\_request.receiver\_username,

invite\_request.sender\_username,

invite\_request.group\_id,

invite\_request.status,

invite\_request.request\_date\_time

FROM

database1.invite\_request, database1.user

WHERE

receiver\_username = :username

AND status = 'pending'

* + - AND invite\_request.receiver\_username = user.username

AND user.username = :username;

* [update\_existing\_user\_field(username: str, field: str, new\_value: str) -> None](http://_vscodecontentref_/9)
  + **Docstring:**
  + """field can be 'username', 'display\_name', 'form\_group', 'password\_hash', 'is\_authenticated', 'is\_active'  
    if field to update is 'password\_hash', new\_value can be raw password  
    cannot update 'datetime\_joined' or 'is\_anonymous' fields  
    Note: even if new\_value is a boolean, it must be passed as a string for the sql query to work"""  
      
    """\_summary\_  
    Updates a field in the user table for a given username  
      
    Args:  
     username (str): username (PK) of user to update  
     field (str): field to update, must be one of: [username, display\_name, form\_group, password\_hash, is\_authenticated, is\_active]  
     new\_value (str): new value to update field with  
      
    Raises:   
     Exception: field is invalid  
     Exception: new\_value is empty  
     Exception: form\_group is invalid  
     Exception: username already exists  
     Exception: display\_name has not changed  
      
    Returns:  
     None  
    """
  + UPDATE database1.user

SET {field} = :new\_value

WHERE username = :username;

* + Here, field does not need to be parameterised as it is validated in the first if statement
  + Sqlalchemy framework also does not support parameterising field names in queries – will use f-strings in Python instead and validate beforehand.
* [delete\_user(username: str) -> None](http://_vscodecontentref_/10)
  + **Docstring:**
  + """\_summary\_  
    Deletes a user from the user table  
      
    Args:  
     username (str): username (PK) of user to delete  
      
    Raises:  
     Exception: username is empty  
     Exception: user does not exist  
      
    Returns:  
     None  
    """
  + DELETE FROM database1.user

WHERE username = :username;

### Class: GroupTable

#### Attributes:

* INVALID\_FIELD\_VALUES = [None, ’’]

#### Methods:

* [check\_group\_exists(group\_id: int) -> bool](http://_vscodecontentref_/11)
  + **Docstring:**
  + """\_summary\_  
    Checks if group with group\_id exists  
      
    Args:  
     group\_id (int): primary key field of group table  
      
    Returns:  
     bool: True if group exists, False otherwise  
    """
  + SELECT COUNT(\*) FROM database1.group WHERE group\_id = :group\_id;
  + Will return boolean of whether the result of that SQL query is greater than 0.
* [get\_number\_of\_groups() -> int](http://_vscodecontentref_/12)
  + **Docstring:**
  + """Returns number of groups in group table"""
  + SELECT COUNT(group\_id) FROM database1.group;
* [get\_last\_group\_id()](http://_vscodecontentref_/13)
  + **Docstring:**
  + """\_summary\_  
    Returns the last group\_id that was created  
      
    Returns:  
     int: last group\_id that was created  
    """
  + SELECT MAX(group\_id) FROM database1.group;
* [create\_group(group\_name: str) -> None](http://_vscodecontentref_/14)
  + **Docstring:**
  + """\_summary\_  
    Creates a group in the group table - calling context must specify group\_name  
      
    Args:  
     group\_name (str): name of group to create  
      
    Raises:  
     Exception: group\_name is empty  
      
    Returns:  
     None  
    """
  + INSERT INTO database1.group (group\_name)

VALUES (:group\_name);

* [get\_group\_record\_by\_group\_id(group\_id: int) -> dict](http://_vscodecontentref_/15)
  + **Docstring:**
  + """\_summary\_  
    returns dictionary of group data (group\_id, group\_name, datetime\_created) for a given group\_id  
      
    Args:  
     group\_id (int): primary key field of group table  
      
    Raises:  
     Exception: group\_id cannot be empty  
     Exception: group with group\_id does not exist  
      
    Returns:  
     dict: dictionary of group\_id, group\_name, datetime\_created  
    """
  + SELECT \* FROM database1.group WHERE group\_id = :group\_id;
* [get\_number\_of\_users(group\_id: int) -> int](http://_vscodecontentref_/16)
  + **Docstring:**
  + """Returns number of users in a given group"""
  + SELECT COUNT(username)

FROM database1.user\_group, database1.group

WHERE user\_group.group\_id = :group\_id

AND group.group\_id = user\_group.group\_id;

* + Ensuring as much of the database CRUD logic is contained within SQL queries rather than the application itself manipulating the output of those SQL queries will be much more efficient and fast to run – this is why there are separate SQL queries even for things such as getting the number of online users and getting the number of users.
* [get\_number\_of\_online\_users(group\_id: int) -> int](http://_vscodecontentref_/17)
  + **Docstring:**
  + """Returns number of online users in a given group"""
* [get\_group\_users(group\_id: int) -> list](http://_vscodecontentref_/18)
  + **Docstring:**
  + """\_summary\_  
    returns list of usernames in a given group with group\_id  
      
    Args:  
     group\_id (int): primary key field of group table  
      
    Raises:  
     Exception: group\_id cannot be empty  
     Exception: group with group\_id does not exist  
      
    Returns:  
     list: list of all usernames of users in specified group with group\_id   
    """
  + SELECT COUNT(user\_group.username)

FROM database1.user\_group, database1.user, database1.group

WHERE group.group\_id = :group\_id

AND user\_group.group\_id = group.group\_id

AND user.username = user\_group.username

AND user.is\_authenticated = 1;

* [get\_group\_user\_details(group\_id: int) -> dict](http://_vscodecontentref_/19)
  + **Docstring:**
  + """\_summary\_  
    Returns dictionary of username, user display name, form group, is\_authenticated for every user in the group  
      
    Args:  
     group\_id (int): primary key field of group table  
      
    Raises:  
     Exception: group\_id cannot be empty  
     Exception: group with group\_id does not exist  
      
    Returns:  
     dict: dictionary of username, user display name, form group, is\_authenticated for every user in the group  
    """
  + SELECT

u.username,

u.display\_name,

u.form\_group,

u.is\_authenticated

FROM

database1.user\_group ug

JOIN

database1.user u

ON ug.username = u.username

WHERE ug.group\_id = :group\_id;

* [get\_group\_datetime\_created(group\_id: int) -> str](http://_vscodecontentref_/20)
  + **Docstring:**
  + """\_summary\_  
    Returns the datetime\_created for a given group\_id  
      
    Args:  
     group\_id (int): primary key field of group table  
      
    Raises:  
     Exception: group\_id cannot be empty  
     Exception: group with group\_id does not exist  
      
    Returns:  
     datetime: datetime\_created of the group  
    """
  + SELECT datetime\_created FROM database1.group WHERE group\_id = :group\_id;
  + The output will be formatted using .strftime('%d/%m/%y %H:%M:%S')
* [get\_all\_group\_messages(group\_id: int) -> list](http://_vscodecontentref_/21)
  + **Docstring:**
  + """\_summary\_   
    returns message records (as tuples) in a given group with group\_id ordered by message\_date\_time (oldest messages first)  
    tuple format: (messaage\_id, message\_content, message\_date\_time, sender\_username, sender\_display\_name)  
      
    Args:  
     group\_id (int): primary key field of group table  
      
    Raises:  
     Exception: group\_id cannot be empty  
     Exception: group with group\_id does not exist  
      
    Returns:  
     list: list of all message records in specified group with group\_id, along with sender display name  
    """
  + SELECT

m.message\_id,

m.message\_content,

m.message\_date\_time,

m.sender\_username,

u.display\_name

FROM

database1.message AS m

JOIN

database1.user AS u

ON m.sender\_username = u.username

WHERE

m.group\_id = :group\_id

ORDER BY

m.message\_date\_time ASC;

* + Here, message\_date\_time is in ascending order as I need the oldest messages first. When the messages are rendered on the page, I will append the oldest messages first, so they appear on top, with the most recent messages at the bottom – this matches the style of modern chat applications.
* [get\_latest\_group\_message(group\_id: int) -> tuple](http://_vscodecontentref_/22)
  + **Docstring:**
  + """\_summary\_  
    returns latest message record in a given group with group\_id  
    tuple format: (message\_id, message\_content, message\_date\_time, sender\_username, sender\_display\_name)  
      
    Args:  
     group\_id (int): primary key field of group table  
      
    Raises:  
     Exception: group\_id cannot be empty  
     Exception: group with group\_id does not exist  
      
    Returns:  
     tuple: latest message record in specified group with group\_id, along with sender display name  
    """
  + SELECT

m.message\_id,

* + - m.message\_content,
    - m.message\_date\_time,
    - m.sender\_username,
    - u.display\_name

FROM

* + - database1.message AS m

JOIN

* + - database1.user AS u
    - ON m.sender\_username = u.username

WHERE

m.group\_id = :group\_id

ORDER BY

* + - m.message\_date\_time DESC

LIMIT 1;

* + Here, message\_date\_time is ordered in descending order as I need the most recent message only.
* [update\_group\_name(group\_id: int, new\_group\_name: str) -> None](http://_vscodecontentref_/23)
  + **Docstring:**
  + """\_summary\_  
    Updates group\_name field in group table for a given group\_id  
      
    Args:  
     group\_id (int): primary key field of group table  
     new\_group\_name (str): new group name to update group with  
      
    Raises:  
     Exception: group\_id or new\_group\_name is empty  
     Exception: group name has not changed  
     Exception: group with group\_id does not exist  
      
    Returns:  
     None   
    """
  + UPDATE database1.group

SET group\_name = :new\_group\_name

WHERE group\_id = :group\_id;

* [delete\_group(group\_id: int) -> None](http://_vscodecontentref_/24)
  + **Docstring:**
  + """\_summary\_  
    Deletes a group from the group table with a given group\_id  
      
    Args:  
     group\_id (int): primary key field of group table  
      
    Raises:  
     Exception: group\_id is empty  
     Exception: group with group\_id does not exist  
      
    Returns:  
     None   
    """
  + DELETE FROM database1.group

WHERE group\_id = :group\_id;

### Class: UserGroupTable

#### Attributes:

* INVALID\_FIELD\_VALUES = [None, ’’]

#### Methods:

* [create\_user\_group(username: str, group\_id: int) -> None](http://_vscodecontentref_/25)
  + **Docstring:**
  + """\_summary\_  
    Creates a user\_group record in the user\_group table - calling context must specify username and group\_id  
      
    Args:  
     username (str): username (PK) of user table  
     group\_id (int): group\_id (PK) of group table  
      
    Raises:  
     Exception: username or group\_id is empty  
     Exception: user with username does not exist  
     Exception: group with group\_id does not exist  
     Exception: user with username is already in group with group\_id  
      
    Returns:  
     None  
    """
  + INSERT INTO database1.user\_group (username, group\_id)

VALUES (:username, :group\_id);

* [delete\_user\_group(username, group\_id)](http://_vscodecontentref_/26)
  + **Docstring:**
  + """\_summary\_  
    Deletes a user\_group record in the user\_group table - calling context must specify username and group\_id  
      
    Args:  
     username (str): username (PK) of user table  
     group\_id (int): group\_id (PK) of group table  
      
    Raises:  
     Exception: username or group\_id is empty  
     Exception: user with username is not in group with group\_id (no such record to delete)  
      
    Returns:  
     None  
    """
  + DELETE FROM database1.user\_group

WHERE username = :username AND group\_id = :group\_id;

### Class: MessageTable

#### Attributes:

* INVALID\_FIELD\_VALUES = [None, ’’]
* MAX\_MESSAGE\_LENGTH = 2000

#### Methods:

* [check\_message\_exists(message\_id: int) -> bool](http://_vscodecontentref_/27)
  + **Docstring:**
  + """\_summary\_  
    Checks if message with message\_id exists  
      
    Args:  
     message\_id (int): primary key field of message table  
      
    Returns:  
     bool: True if message exists, False otherwise  
    """
  + SELECT COUNT(\*) FROM database1.message

WHERE message\_id = :message\_id;

* [get\_number\_of\_messages() -> int](http://_vscodecontentref_/28)
  + **Docstring:**
  + """\_summary\_  
    Returns number of messages in message table  
    Returns:  
     int: number of messages in message table  
    """
  + SELECT COUNT(message\_id) FROM database1.message;
* [create\_message(message\_content: str, sender\_username: str, group\_id: int) -> None](http://_vscodecontentref_/29)
  + **Docstring:**
  + """\_summary\_  
    Creates a message in the message table - calling context must specify message\_content, sender\_username, and group\_id  
      
    Args:  
     message\_content (str): Message text (must be <= MessageTable.MAX\_MESSAGE\_LENGTH (2000) characters)  
     sender\_username (str): username (PK) of message sender   
     group\_id (int): group\_id (PK) of group message is sent in  
      
    Raises:  
     Exception: message\_content length is > MessageTable.MAX\_MESSAGE\_LENGTH (2000) characters  
     Exception: message\_content, sender\_username, or group\_id is empty  
     Exception: user with sender\_username does not exist  
     Exception: group with group\_id does not exist  
     Exception: user with sender\_username is not in group with group\_id  
      
    Returns:  
     None  
    """
  + INSERT INTO database1.message (message\_content, sender\_username, group\_id)

VALUES (:message\_content, :sender\_username, :group\_id);

* [get\_message\_record\_by\_message\_id(message\_id: int) -> dict](http://_vscodecontentref_/30)
  + **Docstring:**
  + """\_summary\_  
    returns dictionary of message data for a given message\_id  
      
    Args:  
     message\_id (int): primary key field of message table  
      
    Raises:  
     Exception: message\_id cannot be empty  
     Exception: a message with message\_id does not exist  
      
    Returns:  
     dictionary\_output (dict): of message\_id, message\_content, message\_date\_time, sender\_username, group\_id  
    """
  + SELECT \* FROM database1.message

WHERE message\_id = :message\_id;

* [update\_message\_content(message\_id: int, new\_message\_content)](http://_vscodecontentref_/31)
  + **Docstring:**
  + """\_summary\_  
    Updates message content for a given message\_id  
      
    Args:  
     message\_id (int): primary key field of message table  
     new\_message\_content (str): new content to update message with  
      
    Raises:  
     Exception: message\_id or new\_message\_content is empty  
     Exception: message with message\_id does not exist  
     Exception: new\_message\_content length is > MessageTable.MAX\_MESSAGE\_LENGTH (2000) characters  
      
    Returns:  
     None  
    """
  + UPDATE database1.message

SET message\_content = :new\_message\_content

WHERE message\_id = :message\_id;

* [delete\_message(message\_id)](http://_vscodecontentref_/32)
  + **Docstring:**
  + """\_summary\_  
    Deletes a message from the message table with a given message\_id  
      
    Args:  
     message\_id (int): primary key field of message table  
      
    Raises:  
     Exception: message\_id is empty  
     Exception: message with message\_id does not exist  
      
    Returns:  
     None  
    """
  + DELETE FROM database1.message

WHERE message\_id = :message\_id;

### Class: InviteRequestTable

#### Attributes:

* INVALID\_FIELD\_VALUES = [None, ’’]

#### Methods:

* get\_number\_of\_invite\_requests() -> int
  + **Docstring:**
  + """\_summary\_  
    Returns number of invite requests in invite\_request table  
      
    Returns:  
     int: number of invite requests in invite\_request table  
    """
  + SELECT COUNT(request\_id) FROM database1.invite\_request;
* get\_number\_received\_invite\_requests(receiver\_username: str) -> int
  + **Docstring:**
  + """\_summary\_  
    Get number of invite requests a user has received  
    Args:  
     receiver\_username (str): the user who is receiving the invite requests  
      
    Raises:  
     Exception: receiver\_username is empty  
     Exception: receiver\_username not in User Table (does not exist)  
      
    Returns:  
     int: number of invite requests specified user has received  
    """
  + SELECT COUNT(\*) FROM database1.invite\_request WHERE receiver\_username = :receiver\_username;
* check\_invite\_request\_id\_not\_in\_use(request\_id: int) -> bool
  + **Docstring:**
  + """\_summary\_  
    Checks if invite\_request\_id is not in use  
      
    Args:  
     request\_id (int): primary key field of invite\_request table  
      
    Returns:  
     bool: True if invite\_request\_id is not in use, False otherwise  
    """
  + SELECT COUNT(\*) FROM database1.invite\_request WHERE request\_id = :request\_id;
* create\_invite\_request(receiver\_username: str, sender\_username: str, group\_id: int, status: str) -> None
  + **Docstring:**
  + """\_summary\_  
    Creates an invite\_request record in the invite\_request table - calling context must specify receiver\_username, sender\_username, group\_id, and status  
      
    Args:  
     receiver\_username (str): username of user to receive invite  
     sender\_username (str): username of user who sent invite  
     group\_id (int): group\_id of group invite is for  
     status (str): status of invite request (pending, accepted, or rejected)  
      
    Raises:  
     Exception: receiver\_username, sender\_username, group\_id, or status is empty  
     Exception: receiver\_username, sender\_username, or group\_id does not exist  
     Exception: status is not one of 'pending', 'accepted', or 'rejected'  
      
    Returns:  
     None  
    """
  + INSERT INTO database1.invite\_request (receiver\_username,

sender\_username, group\_id, status)

VALUES (:receiver\_username, :sender\_username, :group\_id, :status);

* get\_invite\_request\_record\_by\_request\_id(request\_id: int) -> dict
  + **Docstring:**
  + """\_summary\_  
    Returns dictionary of invite\_request data for a given request\_id  
      
    Args:  
     request\_id (int): primary key field of invite\_request table  
      
    Raises:  
     Exception: request\_id cannot be empty  
     Exception: request with request\_id does not exist  
      
    Returns:  
     dict: dictionary of request\_id, receiver\_username, sender\_username, group\_id, status  
    """
  + SELECT \* FROM database1.invite\_request WHERE request\_id = :request\_id;
* get\_received\_invite\_requests(receiver\_username: str) -> list
  + **Docstring:**
  + """Gets list of received invite requests (as dictionaries) for a specific user.  
      
    Args:  
     receiver\_username (str): The user who is receiving these invite requests  
      
    Raises:  
     Exception: receiver\_username is empty  
     Exception: receiver username does not exist (not in user table)  
      
    Returns:  
     list: list of dictionaries {sender\_username, group\_name, status, request\_date\_time}  
    """
  + SELECT sender\_username, group\_name, status, request\_date\_time, request\_id

FROM database1.invite\_request, database1.group

WHERE database1.invite\_request.receiver\_username = :receiver\_username

AND database1.group.group\_id = database1.invite\_request.group\_id

ORDER BY

CASE

WHEN status = 'pending' THEN 1

ELSE 2

END,

request\_date\_time DESC;

* get\_sent\_invite\_requests(sender\_username: str) -> list
  + **Docstring:**
  + """Gets list of sent invite requests (as dictionaries) for a specific user.  
      
    Args:  
     sender\_username (str): User who has sent these invite requests.  
      
    Raises:  
     Exception: sender\_username is empty  
     Exception: user with sender\_username does not exist (not in user table)  
      
    Returns:  
     list: list of dictionaries of invite requests (receiver\_username, group\_name, status, request\_date\_time)  
    """
  + SELECT receiver\_username, group\_name, status, request\_date\_time, request\_id

FROM database1.invite\_request, database1.group

WHERE database1.invite\_request.sender\_username = :sender\_username

AND database1.group.group\_id = database1.invite\_request.group\_id

ORDER BY

CASE

WHEN status = 'pending' THEN 1

ELSE 2

END,

request\_date\_time DESC;

* update\_invite\_request\_status(request\_id: int, new\_status: str) -> None
  + **Docstring:**
  + """\_summary\_  
    Updates status field in invite\_request table for a given request\_id  
      
    Args:  
     request\_id (int): primary key field of invite\_request table  
     new\_status (str): new status to update request with  
      
    Raises:  
     Exception: request\_id or new\_status is empty  
     Exception: status is not one of 'pending', 'accepted', or 'rejected'  
     Exception: status has not changed  
     Exception: request with request\_id does not exist  
      
    Returns:  
     None  
    """
  + UPDATE database1.invite\_request

SET status = :new\_status

WHERE request\_id = :request\_id;

* delete\_invite\_request(request\_id: int) -> None
  + **Docstring:**
  + """\_summary\_  
    Deletes an invite\_request record in the invite\_request table - calling context must specify request\_id  
      
    Args:  
     request\_id (int): primary key field of invite\_request table  
      
    Raises:  
     Exception: request\_id cannot be empty  
     Exception: request with request\_id does not exist  
      
    Returns:  
     None  
    """
  + DELETE FROM database1.invite\_request

WHERE request\_id = :request\_id;

## Routes.py

This file will have two roles:

1. Handling the user’s HTTPS requests to different parts of the application and returning templates as a response. This will involve importing the different table classes from models.py to access data from the database, manipulate this data and use this to provide functionality to the user. This data will then be displayed within the rendered template as a part of the HTTPS response.
2. APIs. There will be several APIs in many application – in cases where real-time updating of data is required, normal HTTPS requests won’t suffice as otherwise, the user would need to reload the page to check for new chat messages – not great for a real-time messaging app. Instead, templates which require real-time data updating functionality will use JavaScript (specifically the JavaScript FETCH API) to send requests to APIs handled by the server. The server will take the requests (some of which will include parameters), ensure the user is authenticated, use the imported table classes from models.py to access and then manipulate the data, before returning said data to the webpage. The JavaScript will then be able to update the webpage in real-time, creating a much better user experience. The APIs will be POST requests, rather than GET requests for a few reasons.
   1. Because GET parameters are passed via the URL, these parameters would be stored in server logs and browser history – not ideal if the parameters are confidential user data. GET requests make it easier to alter the request parameters being sent to the server, as it will appear directly in the browser’s URL address bar.
   2. This can also lead to long, visually unappealing parameters directly in the address bar – while, at least from the user’s perspective, POST requests would make it feel like the application was working its magic behind the scenes.
   3. However, it is still easy to fake POST requests, and using POST requests nevertheless cannot be trusted outright – I will still be using strict validation criteria.
   4. Flask\_login’s session cookie functionality will also be very helpful here, as it allows me to easily check the username (PK in the user table) of the request sender and if it matches the username stored in the secure session cookie.

### List of APIs

#### [leave\_group()](http://_vscodecontentref_/32)

* + **Docstring:**

"""\_summary\_  
Allows the current authenticated user to leave a group.

If the user is not authenticated, a 403 error is raised - useris redirected to a 403 page - there is a separate route for this.

If the group ID extracted from the request parameters is invalid, an error message is flashed.

If the user is not a member of the group, a 403 error is raised.

Will use UserTable.check\_user\_in\_group() to check if the user is in the group.

If the user successfully leaves the group, a success message is flashed.

UserTable.delete\_user\_group() is used to remove the user from the group.

This API is slightly different from the others in that it is not called by the JavaScript, but is instead called when the user clicks the 'Leave Group' button on a group options panel.

Returns:

A redirect response to the index page.  
"""

#### [number\_online\_users()](http://_vscodecontentref_/32)

* + **Docstring:**

"""\_summary\_  
Get the number of online users in a specific group.

This function checks if the current user is authenticated. If not, it returns a response indicating that the user is unauthenticated. This uses flask\_login's session cookie functionality.

If the user is authenticated, it retrieves the group ID from the request JSON parameters, fetches the number of online usersusing GroupTable.get\_number\_of\_online\_users(),

and returns the count in a JSON response.

Returns:

Response: A JSON response containing either the number of online users or an error message with the appropriate HTTP status code.  
"""

#### [number\_total\_users()](http://_vscodecontentref_/32)

* + **Docstring:**

"""\_summary\_  
Get the total number of users in a group.

This function checks if the current user is authenticated. This uses flask\_login's session cookie functionality.

If not, it returns a response indicating that the user is unauthenticated, with a 401 code.

If the user is authenticated, it retrieves the group ID from the request JSON parameters, fetches the total number of users in the specified group, and returns this number in the response. This uses GroupTable.get\_number\_of\_users().

Returns:

Response: A JSON response containing either an error message with a 401 status code if the user is unauthenticated, or the total number of users in the group with a 200 status code.  
"""

#### [group\_list()](http://_vscodecontentref_/32)

* + **Docstring:**

*"""\_summary\_*  
Retrieve the list of groups for the authenticated user.

If the current user is not authenticated (this uses flask\_login's session cookie functionality), returns a 401 response with an 'UNAUTHENTICATED USER' message.

Otherwise, retrieves the list of groups associated with the current user's username from the UserTable and returns it in a 200 response.

This uses UserTable.get\_user\_groups() to retrieve the list of groups for the current user (this uses that same flask\_login session cookie functionality).

Returns:

Response: A Flask response object containing the list of user groups in JSON format with a 200 status code, or an 'UNAUTHENTICATED USER' message with a 401 status code if the user is not authenticated.  
*"""*

#### [group\_user\_details()](http://_vscodecontentref_/32)

* + **Docstring:**

"""\_summary\_  
Retrieve details of users in a specified group.

This function checks if the current user is authenticated. If not, it returns a 401 response indicating an unauthenticated user.

This uses flask\_login's session cookie functionality.

If the user is authenticated, it retrieves the group ID from the request JSON parameters, fetches the user details for that group from the GroupTable using GroupTable.get\_user\_group\_details(), and returns the details as a JSON response.

Returns:

Response: A JSON response containing either an error message with a 401 status code if the user is unauthenticated, or a list of user details with a 200 status code if the user is authenticated.

"""

#### [all\_group\_messages()](http://_vscodecontentref_/32)

* + **Docstring:**
  + """\_summary\_

Retrieve all messages for a specific group.

This function checks if the current user is authenticated. If not, it returns a 401 response.

This will prevent unauthorised access of data and uses flask\_login's session cookie functionality.

If authenticated, it retrieves all messages for the specified group ID from the request JSON parameters.

The messages are then formatted into a response list with additional user information.

Returns:

Response: A JSON response containing a list of messages with the following fields:

- message\_id (int): The ID of the message.

- message\_content (str): The content of the message.

- message\_date\_time (str): The date and time the message was sent, formatted as '%d/%m/%y %H:%M:%S' using strftime.

- sender\_username (str): The username of the sender.

- sender\_display\_name (str): The display name of the sender.

- usertype (str): Indicates if the sender is the current user ('current-user') or another user ('other-user').

Raises:

401 Error: If user is not authenticated. This will prevent unauthorised access of data. This will redirect the user to a 401 error page.  
"""

#### [latest\_group\_message()](http://_vscodecontentref_/32)

* + **Docstring:**
  + """\_summary\_  
    Retrieve the latest message from a group chat.

This function checks if the current user is authenticated. If not, it returns a 401 response. This will redirect the user to a 401 error page.

If authenticated, it retrieves the latest message from the specified group and formats the response.

Returns:

Response: A JSON response containing the latest message details or an error message if the user is unauthenticated.

Response JSON Structure:

{

'message\_id': int,

'message\_content': str,

'message\_date\_time': str (formatted as '%d/%m/%y %H:%M:%S'),

'sender\_username': str,

'sender\_display\_name': str,

'usertype': str ('current-user' or 'other-user')  
"""

#### [send\_message()](http://_vscodecontentref_/32)

* + **Docstring:**

"""\_summary\_  
Handles the sending of a message by an authenticated user.

Returns:

Response: A JSON response indicating the result of the message sending operation.

- If the message is successfully sent, returns a 200 status with 'MESSAGE SENT'.

Raises:

401 Error: If user is not authenticated. This will prevent unauthorised access of data.  
"""

#### [all\_received\_group\_invites()](http://_vscodecontentref_/32)

* + **Docstring:**

"""\_summary\_  
Retrieve all received group invites for the authenticated user.

This function checks if the current user is authenticated. If not, it returns a 401 UNAUTHENTICATED USER response.

If the user is authenticated, it fetches all received group invites from the InviteRequestTable.get\_received\_invite\_requests() for the current user.

The invites are then formatted into a list of dictionaries containing the sender's username, group name, request date and time, status, and request ID. The formatted list is returned as a JSON response with a 200 status code.

Returns:

Response: A JSON response containing a list of dictionaries with the following keys:

- sender\_username (str): The username of the sender.

- group\_name (str): The name of the group.

- request\_date\_time (str): The date and time of the request in the format '%d/%m/%y %H:%M:%S'.

- status (str): The status of the invite.

- request\_id (int): The ID of the request.

Raises:

401 Error: If user is not authenticated. This will prevent unauthorised access of data.  
"""

#### [all\_sent\_group\_invites()](http://_vscodecontentref_/32)

* + **Docstring:**
  + """

Retrieve all group invites sent by the current authenticated user.

This function checks if the current user is authenticated. If not, it returns a 401 response indicating an unauthenticated user.

If the user is authenticated, it fetches all sent group invite requests from the InviteRequestTable.get\_sent\_invite\_requests() for the current user's username.

The invite requests are then formatted into a list of dictionaries containing the receiver's username, group name, request date and time, status, and request ID. The formatted list is returned as a JSON response with a 200 status code.

Returns:

Response: A Flask response object containing a JSON list of sent group invites and a status code.

Raises:

401 error: If the user is not authenticated.  
"""

#### [accept\_invite()](http://_vscodecontentref_/32)

* + **Docstring:**
  + """

Accepts an invite request for the current authenticated user.

Returns:

Response: A JSON response indicating the result of the operation.

- 200 if the invite request is successfully accepted.

Raises:

401 Error: If user is not authenticated.  
"""

#### [reject\_invite()](http://_vscodecontentref_/32)

* + **Docstring:**
  + """

Rejects an invite request.

This function checks if the current user is authenticated. If not, it returns a 401 response.

If the user is authenticated, it retrieves the request ID from the JSON payload of the request, updates the invite request status to "rejected" in the InviteRequestTable, and returns a 200 response.

Returns:

Response: A Flask response object with a JSON message and an appropriate HTTP status code.

Raises:

401 Error: If user is not authenticated. This will prevent unauthorised access to user data.  
"""

#### [cancel\_outgoing\_invite()](http://_vscodecontentref_/32)

* + **Docstring:**
  + *"""*

*Cancels an outgoing invite request.*

*This function checks if the current user is authenticated. If not, it returns a 401 response indicating an unauthenticated user.*

*If the user is authenticated, it retrieves the request ID from the JSON payload of the request, deletes the invite request from the InviteRequestTable using InviteRequest.delete\_invite\_request(), and returns a 200 response indicating that the invite request has been cancelled.*

*Returns:*

*Response: A Flask response object with a JSON message and an appropriate HTTP status code - 200.*

*Raises:*

*401 Error: If user is not authenticated. This will prevent unauthorised access to user data.  
"""*

### List of Webpage Routes

* [index()](http://_vscodecontentref_/32)
  + **Docstring:**

"""

Render the index page for the chat application.

If the current user is authenticated, it checks if the user is part of any groups.

If the user is not in any groups, it sets a placeholder message prompting the user to create a group or check group invites. If the user is in groups, it sets a placeholder message to select a group chat.

Note that the index page itself, will extend (template rendering is handled using the Jinja template engine) from group\_list.html, which extends from base.html.

This means that the template will render base.html, which will then render group\_list.html within that.

Index.html will then be rendered within group.html - this significantly reduces code repetition across templates.

Returns:

rendered template: Rendered template for the index page with the appropriate placeholder message if the user is authenticated.

Response: Alternatively, redirect to the sign-in page if the user is not authenticated.

"""

* [chat\_window()](http://_vscodecontentref_/32)
  + **Docstring:**
  + *"""*

*Renders the chat window for a given group.*

*This function is annotated with @login\_required, which ensures that the user is authenticated before accessing the chat window.*

*Users will be redirected to the sign-in page if they attempt to visit this page without being authenticated by Flask.*

*Args:*

*group\_id (int or str): The ID of the group.*

*Returns:*

*Response: The rendered template for the chat window.*

*Raises:*

*404: If the group does not exist.*

*403: If the current user is not a member of the group.  
"""*

* [chat\_options()](http://_vscodecontentref_/32)
  + **Docstring:**
  + *"""*

*Renders the chat options page for a given group.*

*404 and 403 pages will have their own routes.*

*Args:*

*group\_id (int or str): The ID of the group.*

*Returns:*

*Response: The rendered template for the chat options page.*

*Raises:*

*404: If the group does not exist.*

*403: If the current user is not a member of the group.  
"""*

* [group\_invites()](http://_vscodecontentref_/32)
  + **Docstring:**
  + *"""*

*Render the group invites page.*

*It then renders the 'invites.html' template with the appropriate title.*

*The majority of the code for this page will be within the APIs which get sent and received invite requests for a user. JavaScript on invites.html will be able to use the responses from those APIs to render the data on this page in a user-friendly format.*

*Returns:*

*rendered template: The rendered HTML content for the group invites page.  
"""*

* [create\_group()](http://_vscodecontentref_/32)
  + **Docstring:**
  + *"""*

*Handle the creation of a new chat group.*

*This function performs the following steps:*

*1. Checks if the current user is authenticated. If not, redirects to the sign-in page.*

*2. If authenticated, initializes the BubbleForm.*

*3. Validates the form submission.*

*4. Checks if the usernames in the bubble list are valid.*

*5. Checks if the group name is valid (alphanumeric characters or spaces only).*

*6. Creates a new group and adds the current user to the group.*

*7. Sends invite requests to the users in the bubble list.*

*Returns:*

*- Redirects to the sign-in page if the user is not authenticated.*

*- Renders the 'create\_group.html' template with the form if there are validation errors.*

*- Redirects to the chat window of the newly created group if successful.  
"""*

* [sign\_in()](http://_vscodecontentref_/32)
  + **Docstring:**
  + *"""*

*Handle the sign-in process for users.*

*If the current user is already authenticated, they are redirected to the index page.*

*Otherwise, a sign-in form is presented to the user. Upon form submission, the user's credentials are validated. If the credentials are correct, the user is logged in and redirected to the next page or the index page. If the credentials are incorrect, an error message is flashed and the user is redirected back to the sign-in page.*

*Returns:*

*Response: A redirect response to the appropriate page based on the authentication status and form submission outcome.  
"""*

* [sign\_out()](http://_vscodecontentref_/32)
  + **Docstring:**
  + *"""*

*Signs out the current user by updating their authentication status in the database*

*and logging them out of the session.*

*Steps:*

*1. Updates the 'is\_authenticated' field of the current user's record in the UserTable to '0' – i.e. unauthenticated.*

*2. Logs out the current user using the logout\_user() function.*

*3. Redirects the user to the sign-in page.*

*Returns:*

*A redirect response to the sign-in page.  
"""*