Chat App

# 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

## 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.

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.

### 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.

## 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.
  + - 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), 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 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.
  + - UserTable methods:
      * get\_number\_of\_users()
      * check\_username\_exists()
      * validate\_password()
      * check\_form\_group\_valid()
      * get\_user\_record\_by\_username()
      * check\_user\_in\_group()
      * get\_number\_user\_groups()
      * get\_user\_groups()
      * get\_pending\_invite\_requests()
      * update\_existing\_user\_field()
      * delete\_user()
    - GroupTable Methods:
      * check\_group\_exists()
      * get\_number\_of\_groups()
      * get\_last\_group\_id()
      * create\_group()
      * get\_group\_record\_by\_group\_id()
      * get\_number\_of\_users()
      * get\_number\_of\_online\_users()
      * get\_group\_users()
      * get\_group\_user\_details()
      * get\_group\_datetime\_created()
      * get\_all\_group\_messages()
      * get\_latest\_group\_message()
      * update\_group\_name()
      * delete\_group()
    - UserGroupTable methods:
      * create\_user\_group()
      * delete\_user\_group()
    - MessageTable methods
      * check\_message\_exists()
      * get\_number\_of\_messages()
      * create\_message()
      * get\_message\_record\_by\_message\_id()
      * update\_message\_content()
* 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. message

* A black screen with white text

  AI-generated content may be incorrect.I am using MySQL for the database server.
* The ‘user’ table represents the user, and includes attributes username, display\_name, form\_group, datetime\_goined, 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.

## ‘Model’ Application Architecture