COMPSYS302 Project B
Final Report

Marcus Wong
mwon724

fort secure chat

How system meets client specifications

Client specifications for this project were developed on an ongoing basis. Core features included the ability for a user to login to the system, and interact with users through messaging, profiles and file transmission. Through continued consultation with our client and with our classmate peers, a list of requirements ordered by priority were created. The system implemented currently exceeds the minimum specifications to a large extent, with many more usability features implemented such as Two Factor Authentication, cross-browser compatibility, and messaging in Markdown format. For more detailed information, please refer to the appendix Figure 2.

Top-level view of system

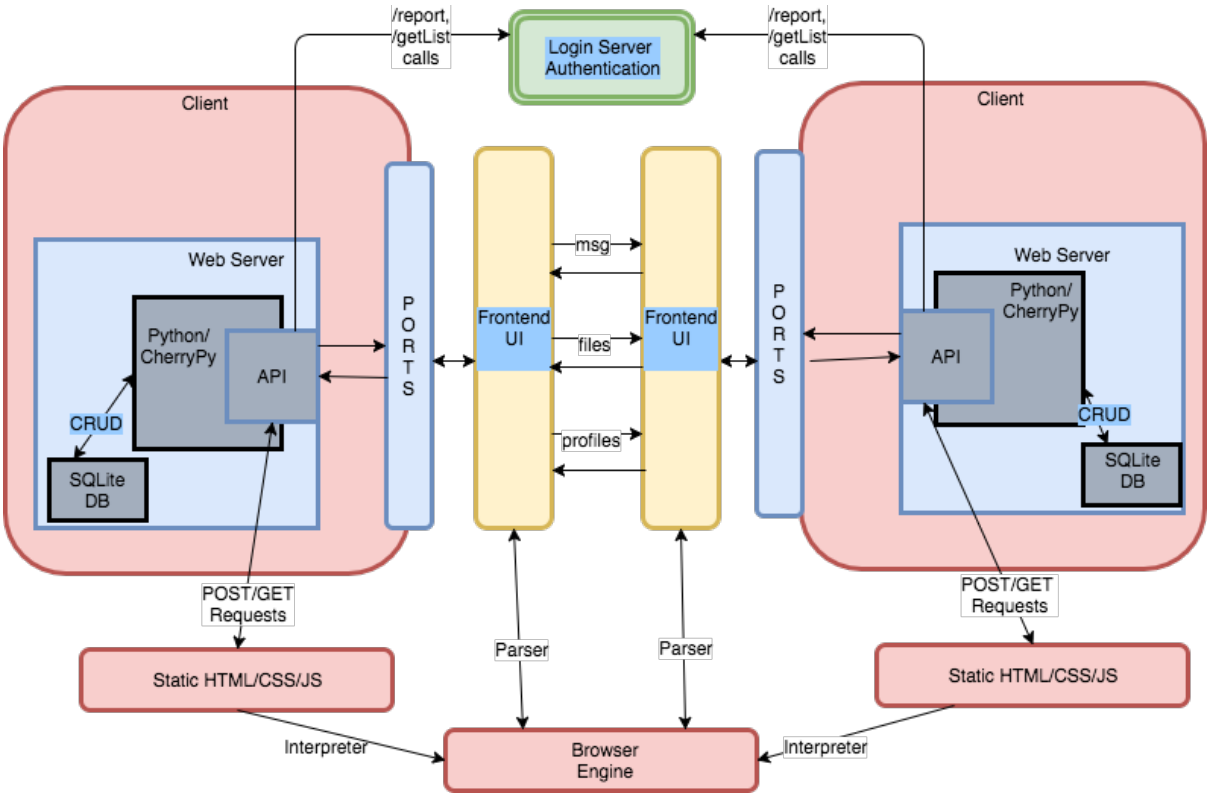

Figure 1: A high-level view of the system

The system works by facilitating peer to peer communication between two clients. In order to achieve this, a Python web framework, CherryPy, is used to provide an API to abstract away boilerplate code that handles HTTP requests and responses. Together with SQLite, the system creates a backend which stores, retrieves, selects and updates data locally (CRUD operations). On the frontend, static webpages are created using HTML and styled using CSS. These are languages that can be 'translated' easily with modern browsers to provide a user interface for interacting with the backend. JS is used for minor front end features, as well as sending AJAX requests which directly call our own APIs (which may in turn call other client's APIs). Some authentication-related calls are also made to the login server to retrieve a list of online users, as well as report the presence of our client on the network.

At the user-interface level, different clients interact with each other by sending and retrieving messages, files and profiles. Front-end actions are translated to requests to call the target user's API.

Significant Issues in Development

A significant issue faced during development was the integration between the front-end interface and back-end Python implementation. Although initially trivial with just HTML/CSS, an unforeseen issue was the sheer amount of JQuery/Javascript required in order to properly provide a 'nice' user interface that rivalled modern messaging clients. For example, jQuery was used to provide the dynamic refreshing of messages, AJAX POST and GET requests, as well as minor stylistic features like automatic message scroll-down. With the project requiring many different languages and tools, JQuery added to an already complicated stack. To overcome this issue, simply more time was spent developing the project. Use of public forums like Stack Overflow was used extensively, as JQuery is a fairly well-known language.

Another issue involved the nature of my personal development environment. Due to access restrictions, clients on the login server would only be able to facilitate two-way communication with clients at the same 'location' (lab computers, university wireless, or external IP). This led to difficulty in testing inter-app communication spontaneously, especially as a student who lived in university accommodation (on location '1' at night instead of '2'). To circumvent this issue, deliberate trips were made to connect to external IP routers capable of port forwarding, so that testing and development could be conducted at night after lab closure.

Additional Functionality Improvements

The developed system includes a modern, cross-browser compatible user interface. This interface is dynamically refreshed so that new file and message transmissions appear in the browser periodically, without needing a full page refresh.

The system provides accurate login status by implementing both a force logoff mechanism upon application exit/server crash, as well as use of threading to periodically communicate with the login server while logged in. This means that other clients will never 'mistakenly' send data to an already offline user. Precautionary measures such as blacklisting and rate limiting were also both implemented in order to filter out potentially undesirable clients, and to prevent denial of service attacks.

The client also (mostly) supports simultaneous users, by capitalising on CherryPy's excellent session management. For more detail on this, see Figure 2 in the Appendix.

The system is also made more usable with additional front-end features. One feature is the ability to search for specific profile pages to view, as well as being able to search sent and received messages. Message search is case insensitive and does not require the search query to match exactly, which fits the typical use case of message searching.

Lastly, Two Factor Authentication was implemented as an additional security feature of the system. This system utilised the Google Authenticator app available on smartphones to provide an extra layer of security, by requiring the client to enter a six-digit time-based password in addition to his/her credentials. Upon an initial user login, a QR code is displayed on the client for the user to scan in Google Authenticator, allowing Google Authenticator to share a common secret with the algorithm present in my system's backend. This algorithm is modelled off the RFC6238 standard for TOTP (Time-based One-Time Passwords).

Discussion: Suitability of P2P Methods

To relate back to the specific use case of our client, our system was designed to solve the issue of privacy concerns in a business sharing important business secrets. It was imperative that the system designed should aim to decentralise data storage and reduce the dependency (and thus, risk) on one node in the network – so some sort of peer to peer network would have to be implemented.

For our specific use case, the combination of a login server hybrid peer-to-peer network was a suitable option. With the current implementation, data is transferred directly from one node to another, with local data storage used in the form of a SQLite database. This means that the server does not have access to the actual information transferred, so an attack on the login server would not equate to (immediate) mass-leakage of data.

A server structure purely for authentication was still maintained, which has its merits. A login server is an easy scalable solution for registering new users. In a business environment, this may be important especially if short-term contract staff (e.g. consultants) require access to the network. Additionally, a login server system is more friendly towards development time, as authentication in a purely peer-to-peer network required advanced knowledge of network security – a skill that would need to be developed.

Discussion: Suitability of Protocol

In a peer to peer environment, the process of developing a protocol is mandatory to ensure working communication. With a wide number of ways to implement a high-level idea, the protocol ensures certain arguments are expected, and clients can react to pre-defined responses. Communication between clients in the implemented system is done purely in a peer-to-peer fashion, so explicit description was required for any methods related to receiving messages, profiles and files.

In short, the protocol was a vital part of the development process – without it, there simply would be no inter-app communication.

Discussion: Suitability of Protocol Development Process

The protocol was developed through ongoing discussion between the client and the development team (consisting of the COMPSYS 302 class and our client, Andrew). Groups were formed to develop their own protocols, with collective discussion occurring at the end. Objectively, this process was effective in deciding the 'general direction' of how the system would work, or what features would be declared as mandatory in the system.

However, it was still heavily varied in terms of precise implementation (such as specific arguments and the values for these arguments). A major factor of this was due to the lack of experience many developers had at the time, which may have impacted people's understanding of how a particular high-level task were to be tackled.

With this being an unavoidable issue given the circumstance, the overall protocol development process was suitable for our needs.

Discussion: Suitability of Suggested Tools

For our system backend, Python was used as our language of choice. Given the developer's previous experience in object oriented languages and data structures, this was a suitable choice. As a popular, high level language, plenty of documentation on Python was available online to learn the language quickly. Python is also the language for accessing the CherryPy API, which we use as the web framework to connect our frontend HTML/CSS/JS to our backend functionality.

CherryPy was helpful in developing a system of our nature. A large amount of the system involved sending and receiving API requests in JSON format, in which CherryPy provided in- built decorators for. CRUD operations, which are prevalent throughout all parts of the system, were made fairly easy due to CherryPy running under Python.

However, it is worth noting that CherryPy occupies a competitive area in the stack – there are many Python-based web frameworks such as Django which are also popular. Comparing the two frameworks, Django offers a much more prolific list of features, with easy-to-use frameworks for templating, authentication and forms – things CherryPy may not fully provide out of the box. Overall, I would suggest that both Django and CherryPy are suitable for the project at hand – with CherryPy edging out Django slightly for the educational benefit a more bare-bones framework provides.

The choice of using of HTML, CSS and JS for the front-end web interface was fairly trivial. These three languages are very commonly used in the majority of modern websites, with widespread support across a range of browsers. Although initially stated as 'optional', JS became a very important part of the implemented system as the styling of the user interface became more complex.

SQLite was used as our database engine. Compared to other database management tools, SQLite provides a light, self-contained and open-source system which fit perfectly with our current use case. The majority of sent and received data in the system was stored and co-ordinated through the use of an SQLite database, made easy through a native Python module (sqlite3). However, it is worth noting that SQLite's lack of concurrency may mean that more traditional relational database systems like MySQL may be suitable if the system exponentially grows in users.

Git was used for version control in this project. Despite being a solo software development project, I still found git surprisingly useful in its ability to managing and review code changes by line, as well as stashing for experimental features.

In addition to the suggested tools, a REST API client (Insomnia) was used heavily during development. I felt that Insomnia (or a similar REST client) should have been a suggested tool for all developers in the project, as the application proved invaluable in solving API request issues and providing JSON validation.

Future Development

With data security being the primary priority of our client, future development should focus on inter-app encryption and hashing to increase the security of data transmission between nodes. This includes the implementation of public key encryption, which would add an extra layer of security to the (relatively) weak login server system. With public key encryption, the client would adopt and share a public key with all other clients, allowing for authentication through the combination of a user's private key with another client's public key (and vice versa) to facilitate communication.

Another feature that would be at high priority is offline messaging. A major downside to a peer-to-peer network is the requirement for both the sender and receivers to be online in order for communication to occur. Given the client's requirements, offline messaging is likely to be a required feature.

Appendix

Important note: Detailed documentation on **all** exposed and non-exposed functions in my implementation can be found in the document, "fort-secure-chat-documentation-v2". This is found in the 'documentation' folder of my git commit.

3

Figure 2: Complete list of implemented features

Grade	Task/Feature Description	Done?
C	Application runs following README instructions on Linux	**Yes**
	User can log in	**Yes**
	User can see who is currently online	**Yes**
	User can see and edit their profile page, and see profiles for other users	**Yes**
	User can send, receive, and view messages+files with someone online	**Yes**
B/B-	Automatically refreshing page (or refreshing content) and/or notifications	**Yes**
	Good use of database(s)	**Yes**
	Use of encryption/hashing/data security within the application	
	Embedded media player for audio/video	**Yes**
	Displays confirmation of message receipt	
	Show user status (at a minimum, including online, away, offline)	**Yes**
	(Local) blacklisting of disruptive clients / unwanted users	**Yes**
	Logs out from the server upon application exit	**Yes**
B+/B	Communication of text formatting of messages	**Yes**[1]
	Rate Limiting	**Yes**
	Use of effective inter-app encryption/hashing/data security including handshake	
	Use of threading for communicating with login server regularly	**Yes**
	Search for profile pages and messages	**Yes**
	(Good) Page Templating	
	Modular and Pythonic code, including commenting and documentation	**Yes**[2]
	Group conversations	
A	Fallback P2P Networking if login server goes down	
	Multiple sessions (users) supported simultaneously	**Yes**[3]
	Unit Testing	
	Nice user interface (preferably compatible cross-browser)	**Yes**[4]
	Offline Messaging (without central server)	
	2FA (Two Factor Authentication)	**Yes**
	Events co-ordination system	
	Fails graciously when interacting with substandard clients	**Yes**[5]

Footnotes:

[1] Markdown messages can be sent and received formatted correctly. However, the frontend expects the user to format their message in markdown format manually and select to send the message as Markdown.

[2] Evidence of commenting and modular/pythonic code can be found throughout all the .py files in the project. Documentation for my APIs is found in the root folder, called 'fort-secure-chat-documentation-v2'. Code follows PEP8 standard for Python, managed by a linter (flake8) installed on the Atom IDE.

[3] Most operations function properly (utilising sessions), with the exception of blacklisting – blacklisted users are shared between all clients. In future development, a refactoring of the blacklist table to create a one to many relationship to the user_string table will be required.

[4] Tested on latest versions of Google Chrome, Firefox and Safari on Ubuntu.

[5] Evidence can be found in login.py with try-except clauses throughout the code.

Note: There is an issue in Ubuntu lab computers where database access is sometimes restricted with "OperationalError: Database is locked". To fix this, open up SQLite Manager, delete an entry, and close SQLite Manager. I recommend deleting values out of user_credentials as there is minimal impact on the running of this program (credentials will be re-created once two-factor authentication is successful).

Figure 3: Initial user interface with tutorial prompt

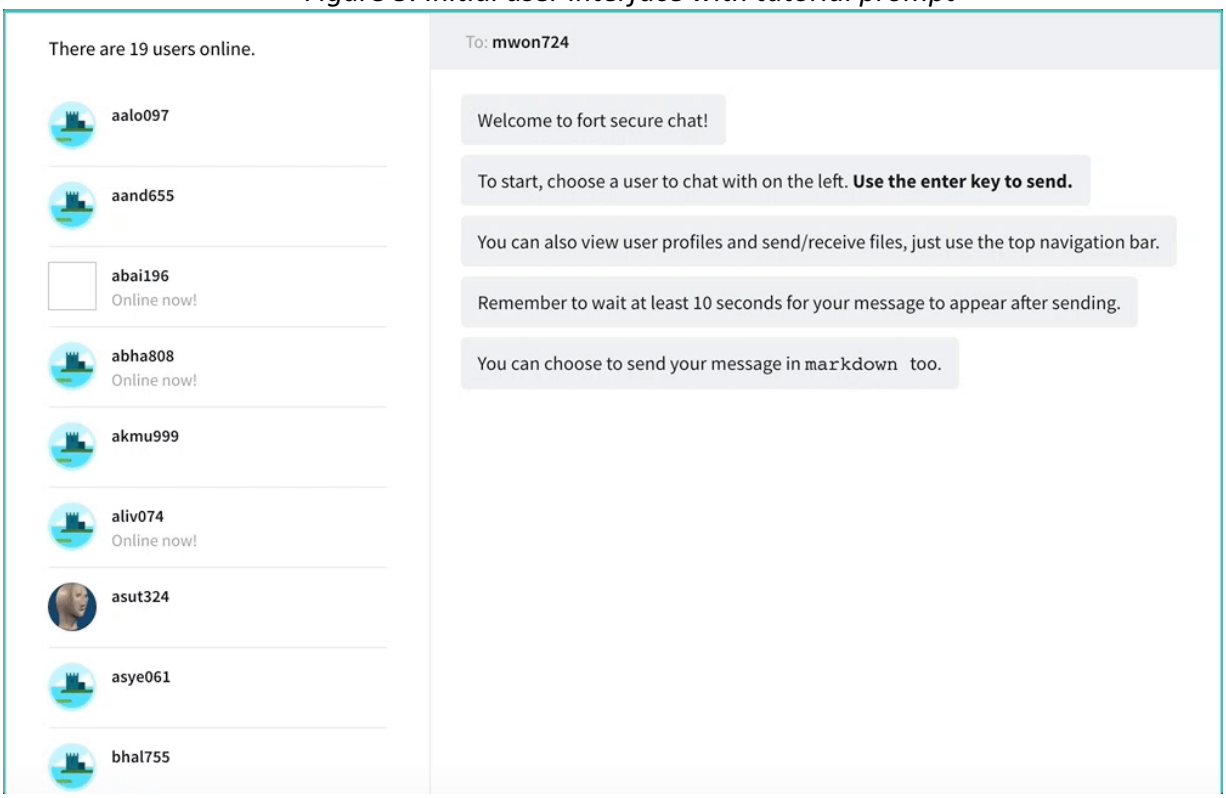

Figure 4: Personal profile page, with profile search function

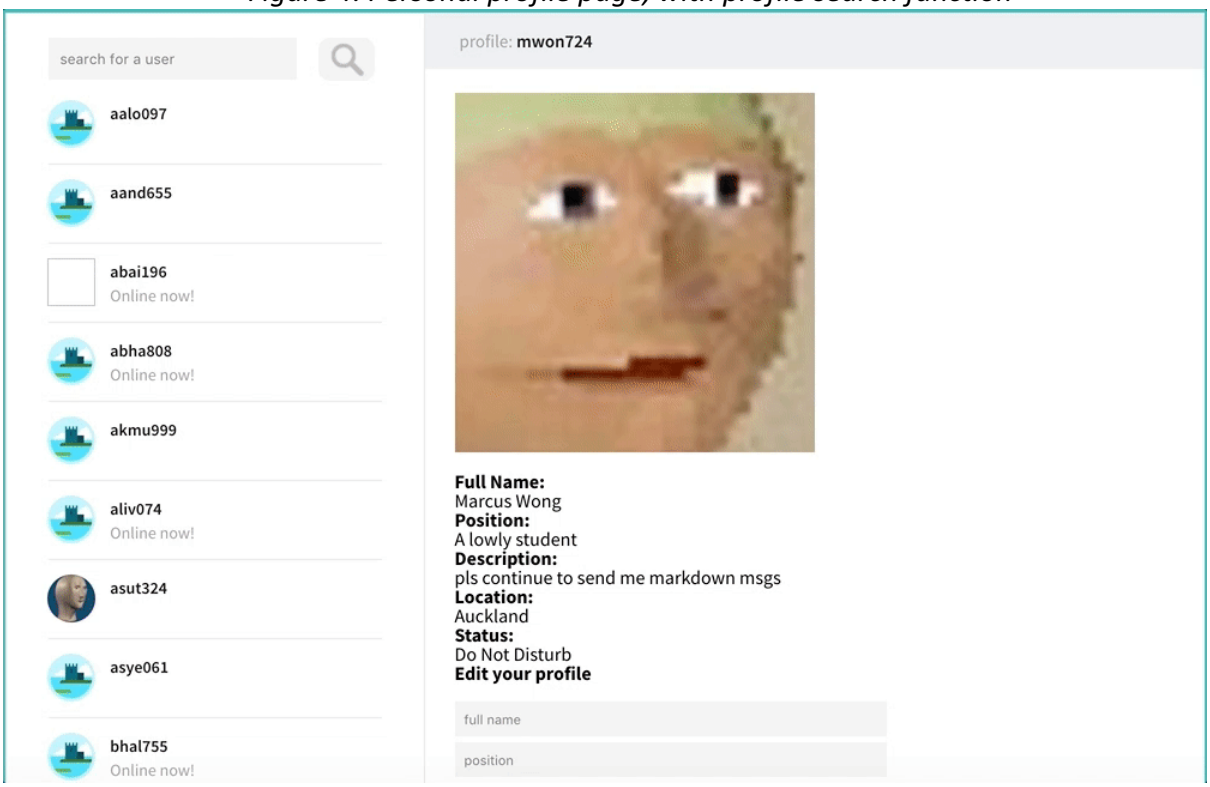

Figure 5: Message logs section, with search

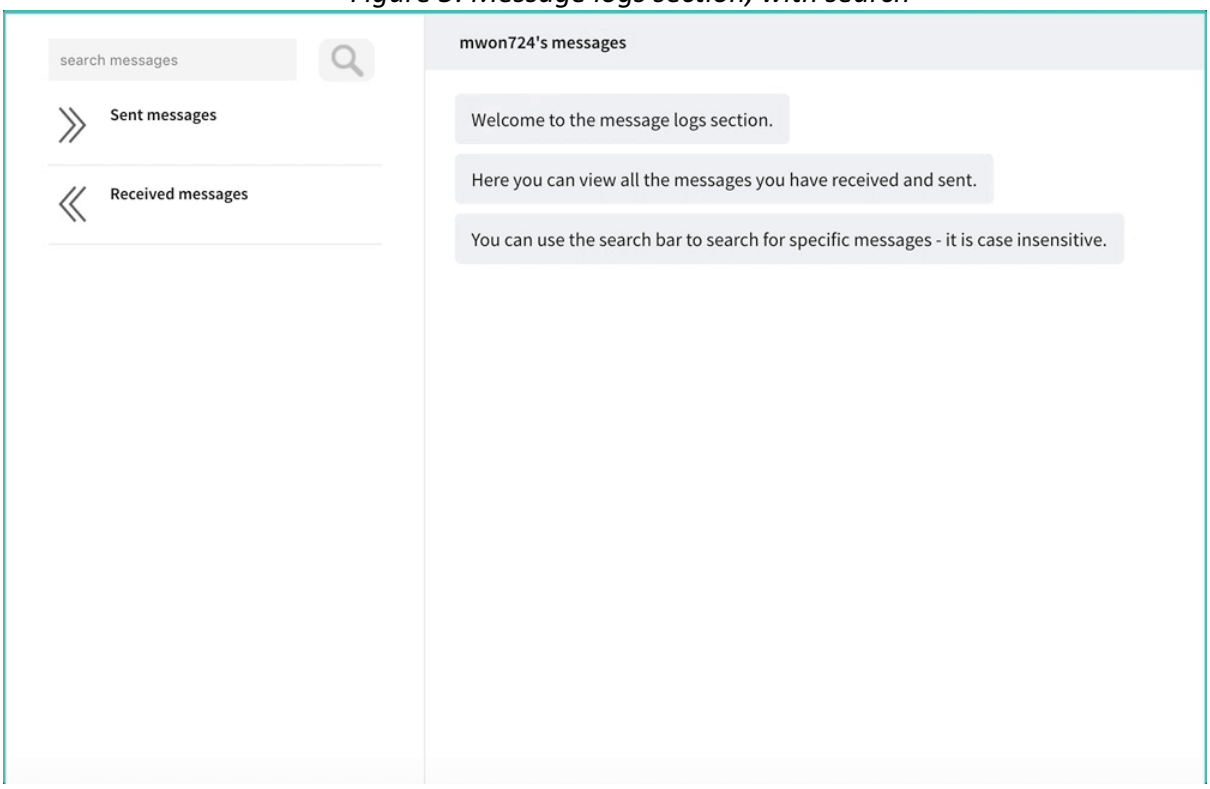

Figure 6: Files, with embedded player/downloads

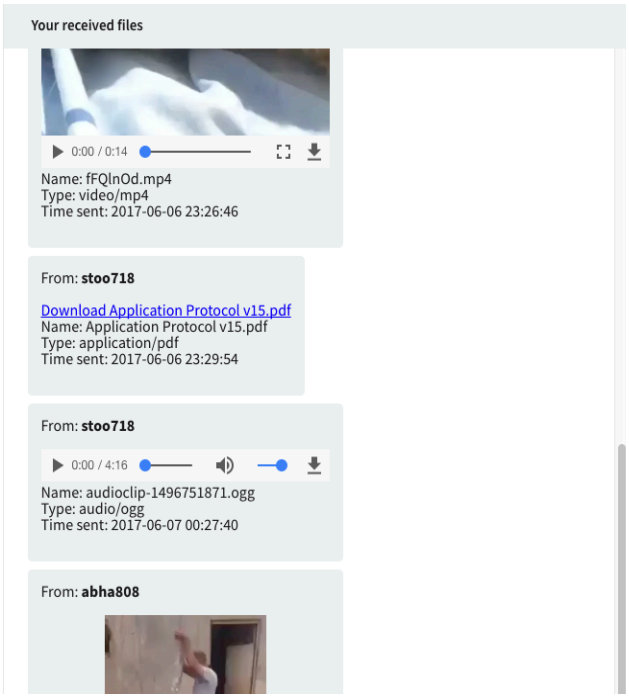

Figure 7: Two factor authentication on Initial login (QR code provided)

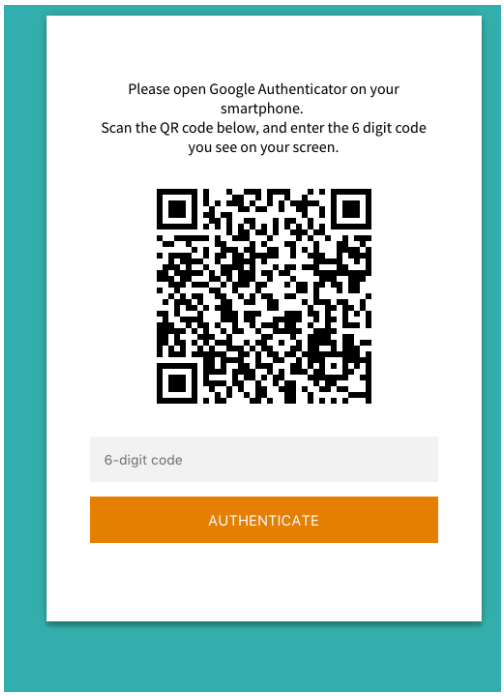

Figure 8: Markdown Text Support