**Secure Chat System with Registration, Login, and  
Encrypted Communication**

Contents

[Overview of the Application 2](#_Toc179749515)

[Libraries included 2](#_Toc179749516)

[Client-Side Implementation (client.py) Main Function 3](#_Toc179749517)

[Creating the Socket 3](#_Toc179749518)

[User Registration and Login Options 3](#_Toc179749519)

[Handling Registration 4](#_Toc179749520)

[Handling Login 4](#_Toc179749521)

[Diffie-Hellman Key Exchange 5](#_Toc179749522)

[AES Encryption 7](#_Toc179749523)

[AES decryption 8](#_Toc179749524)

[Chat ending 9](#_Toc179749525)

[Server-Side Implementation (server.py) 9](#_Toc179749526)

[Main Function 9](#_Toc179749527)

[Client handling 9](#_Toc179749528)

[Registration Process 9](#_Toc179749529)

[Password hashing 10](#_Toc179749530)

[Storing User Information 11](#_Toc179749531)

[Sending Confirmation 11](#_Toc179749532)

[Login Process 11](#_Toc179749533)

[Diffie-Hellman Key Exchange 12](#_Toc179749534)

[AES encryption 12](#_Toc179749535)

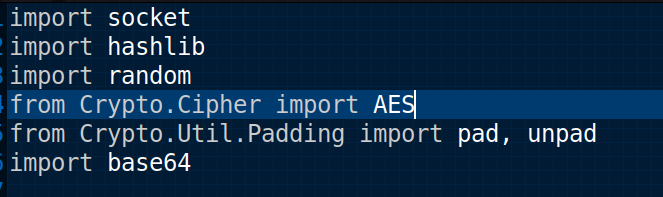
[AES decryption 12](#_Toc179749536)

[Chat ending 12](#_Toc179749537)

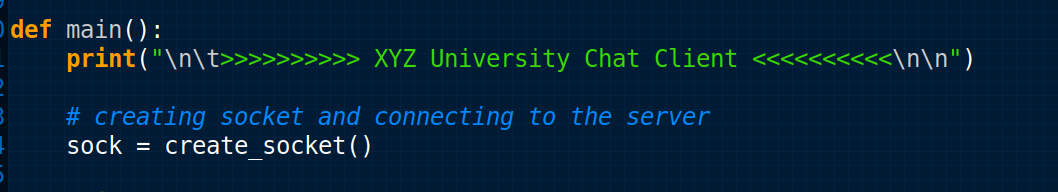
# Overview of the Application

The application consists of a server (server.py) and a client (client.py) that communicate over TCP sockets. The server handles user registration and login, while facilitating encrypted communication between clients. Key cryptographic techniques employed include password hashing and the Diffie-Hellman key exchange for secure message transmission using AES encryption.

# Libraries included

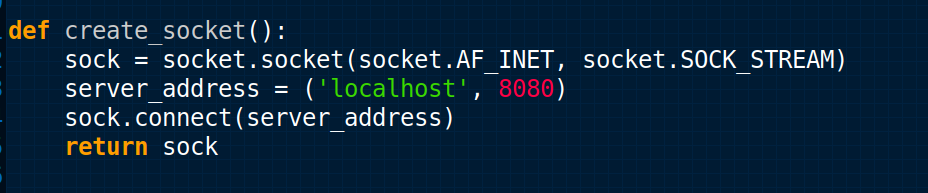
* **socket:** Handles network communication between client and server over TCP/IP using functions like send(), recv(), and accept().
* **hashlib:** Provides secure hashing using SHA-256. Used to hash passwords with salt and derive AES keys from Diffie-Hellman shared keys.
* **base64:** Encodes binary data (like encrypted messages) into text for transmission, and decodes it back to binary before decryption.
* **random:** Generates random numbers for private keys in Diffie-Hellman key exchange and random salts for password hashing.
* **os:** Manages process handling on the server, allowing multiple clients to be served concurrently by forking new processes.
* **Crypto.Cipher.AES:** Handles AES encryption and decryption in CBC mode for secure communication and encryption of sensitive data.
* **Crypto.Util.Padding**: Adds and removes padding to data for AES encryption since AES requires data to be in blocks of 16 bytes.
* 

# Client-Side Implementation (client.py) Main Function



* The main() function starts by printing a print statement for the client side.
* It calls create\_socket(), which establishes a connection to the server running on localhost at port 8080.

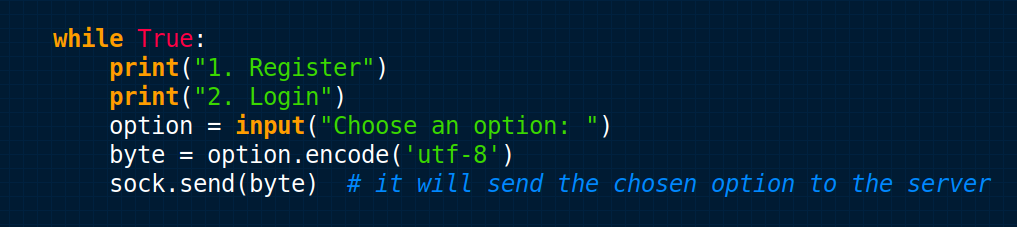
## Creating the Socket



* **socket Module:** This is part of Python’s standard library that provides access to the BSD socket interface, which is a way to communicate over the network.
* **socket.AF\_INET:** This specifies that we are using the IPv4 address family. It means that the socket will communicate using IPv4 addresses.
* **socket.SOCK\_STREAM**: This indicates that we are creating a TCP socket. TCP (Transmission Control Protocol) is a connection-oriented protocol that ensures reliable communication between the client and server. This is essential for chat applications where message delivery must be guaranteed.
* The **sock** variable now holds an instance of a TCP socket that can be used to communicate over the network.
* **server\_address = ('localhost', 8080)** This line defines the address of the server to which the client will connect.
* **8080** is the port number.

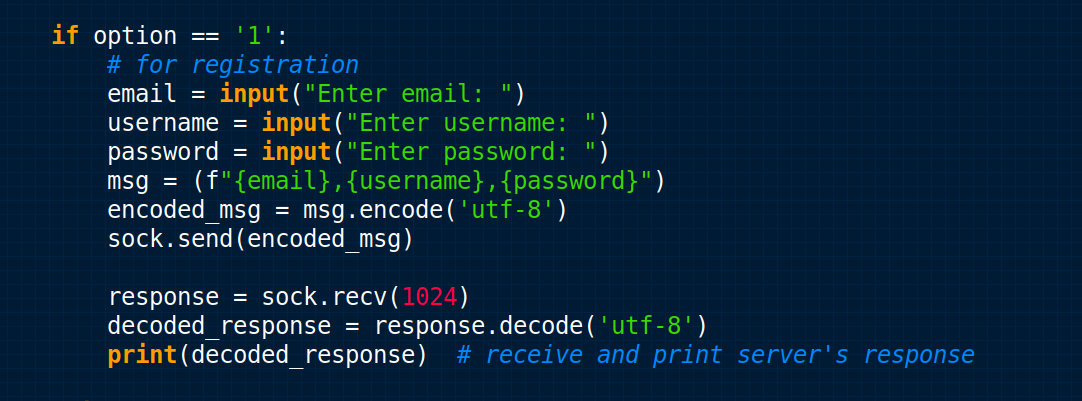
## 

## User Registration and Login Options



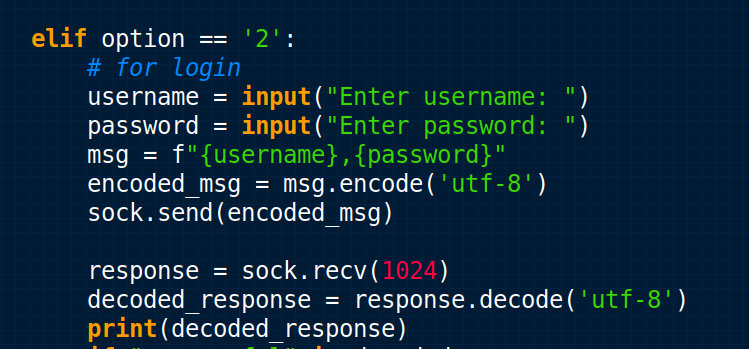
* The client presents options for user registration and login, sending the user's choice to the server.
* The user’s choice is encoded in UTF-8 before sending to the server. This is important because socket communication requires data to be transmitted in byte format. Encoding ensures the data is correctly formatted for transmission, handling characters beyond the standard ASCII set, and preventing potential issues with character representation.

## Handling Registration



* If the user chooses to register, the client collects email, username, and password inputs.
* These details are concatenated into a string and sent to the server.
* The client then waits for a response, which will be sent from the server, which is printed to the console.

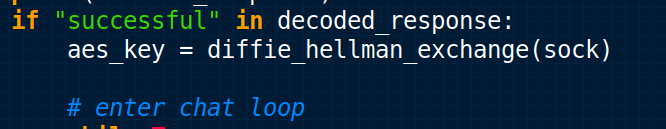
## Handling Login

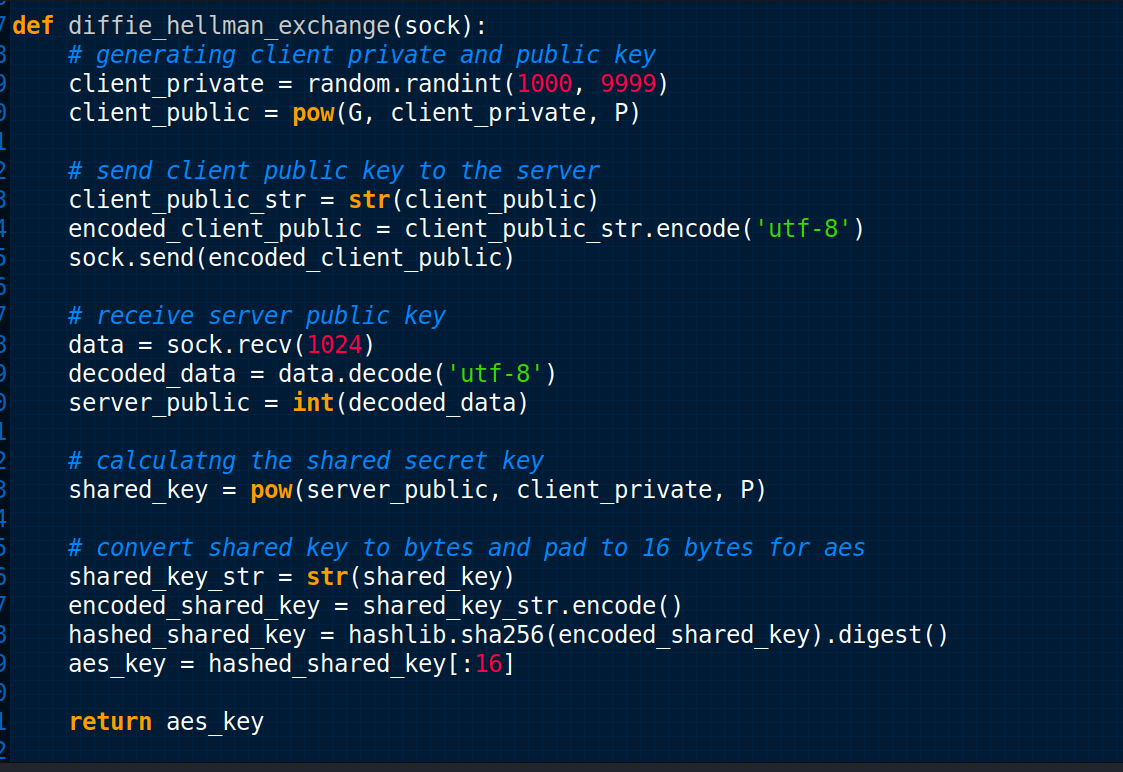


* For login, the client collects the username and password, sends them to the server, and prints the server’s response.
* If the login is successful, the client proceeds to the secure chat phase.

## Diffie-Hellman Key Exchange

* As soon as the client login a key is exchanged between the client and server for the aes encryption and decryption process.

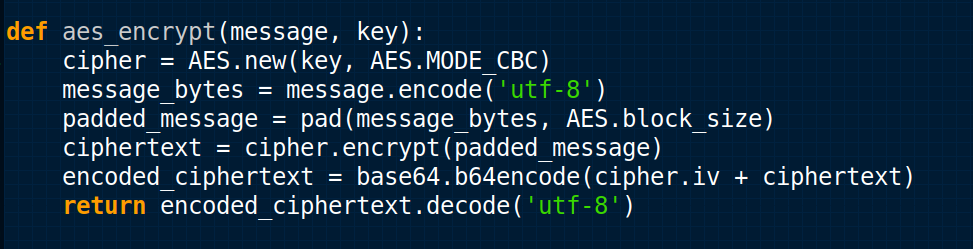




* In this function, the Diffie-Hellman key exchange is implemented to securely generate a shared secret key between the client and server.
* First, the client generates a private key, which is a random integer between 1000 and 9999, ensuring that each session will have a unique private key. The public key is then computed using the formula: client\_public = G^client\_private % P.
* Here, G is the base and P is the prime number defined earlier in the code. These values are common for both the client and server.
* The client's public key is then converted to a string (client\_public\_str) and encoded in UTF-8 format (encoded\_client\_public). The encoding process ensures that the public key can be safely transmitted over the socket in byte format, as sockets deal with raw bytes, not strings or other data types. This step allows the public key to be sent to the server.
* Next, the client receives the server's public key via the socket. The data is received in byte format, so it is decoded using UTF-8 into a string. After decoding, the string representation of the server's public key is converted back into an integer (server\_public).
* Now, using the received server public key and the client’s private key, the client calculates the shared secret key with the formula given above.
* To use this key for AES encryption, it needs to be converted into a byte format. First, the shared key is converted to a string (shared\_key\_str), then encoded into bytes (encoded\_shared\_key). After that, the shared key is hashed using the SHA-256 algorithm. Hashing ensures that the key is securely derived and can be truncated to the necessary 16-byte length required for AES encryption. The result is stored in the variable aes\_key, which will be used for secure message encryption and decryption.
* The function finally returns the 16-byte aes\_key, which will be used throughout the communication session to encrypt and decrypt messages securely.

## AES Encryption

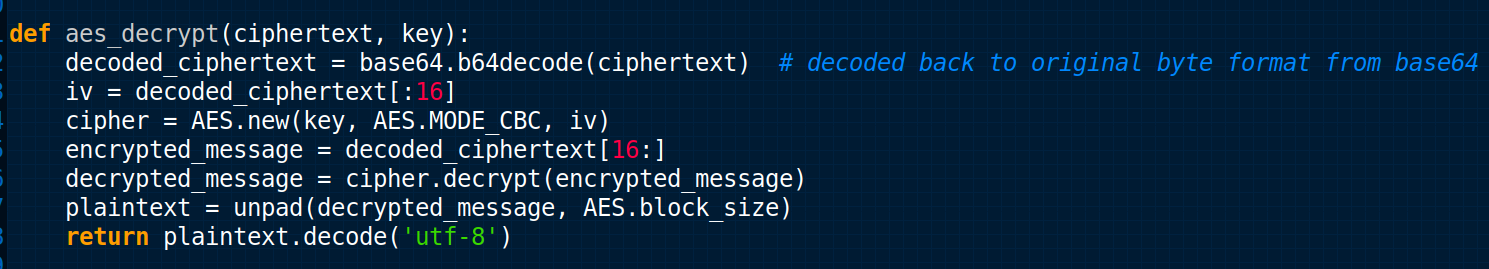
* In client.py, the user types a message. Before sending it, the message is encrypted using AES-CBC mode.



* **cipher = AES.new(key, AES.MODE\_CBC)**This line creates a new AES cipher object using the provided key. The AES.MODE\_CBC specifies that CBC mode is used, which means each block of plaintext is XORed with the previous ciphertext block before being encrypted.
* **message\_bytes = message.encode('utf-8')**The input message, which is a string, is converted to its byte representation using UTF-8 encoding. This is necessary because encryption algorithms work on bytes rather than text. UTF-8 ensures compatibility across different systems and encodes characters in a standardized format**.**
* **padded\_message = pad(message\_bytes, AES.block\_size)**AES encryption works with data blocks of fixed size (typically 16 bytes). If the message's length is not a multiple of the block size, it must be padded to the appropriate length. The pad function ensures that the message\_bytes is padded to match the AES block size (16 bytes in this case).
* **ciphertext = cipher.encrypt(padded\_message)**The padded message is then encrypted using the AES cipher created earlier. This step produces the ciphertext, which is the encrypted version of the message in byte format.
* **encoded\_ciphertext = base64.b64encode(cipher.iv + ciphertext)**CBC mode requires an initialization vector (IV), which is randomly generated when creating the cipher object. The IV is used to start the encryption process, but it must be shared along with the ciphertext to allow proper decryption. Here, the IV is concatenated with the ciphertext before encoding them together**.**The combined IV and ciphertext are then encoded using Base64, a method for converting binary data into a string of ASCII characters. This makes the encrypted data easier to transmit or store, as Base64 is a widely accepted encoding format for byte data.
* **return encoded\_ciphertext.decode('utf-8')**  
  Finally, the Base64-encoded ciphertext (including the IV) is decoded back into a UTF-8 string. This is necessary because the final encrypted message will be sent as a string, whether via network communication or stored in files.

## AES decryption

* in client.py when the message is received from the server it is decrypted using the following code snippet.



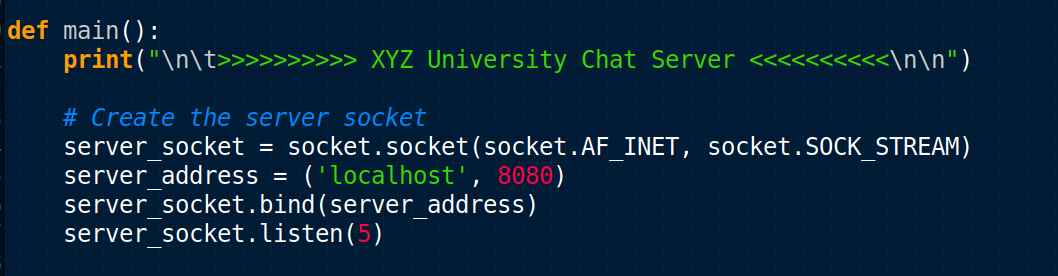
* **decoded\_ciphertext = base64.b64decode(ciphertext)**  
  The received ciphertext is first Base64 decoded back to its original byte format. Since the AES encryption process encoded the ciphertext (and IV) into Base64 for easier transmission or storage, this step reverses that process to retrieve the encrypted byte data.
* **iv = decoded\_ciphertext[:16]**  
  AES encryption in CBC mode requires an initialization vector (IV), which was prepended to the ciphertext during encryption. Here, the first 16 bytes of the decoded ciphertext are extracted as the IV. This is necessary for the decryption process since CBC mode uses the IV to reverse the XOR operation done during encryption.
* **cipher = AES.new(key, AES.MODE\_CBC, iv)**  
  A new AES cipher object is created using the same key that was used for encryption, in CBC mode. The extracted IV is provided as an argument to this cipher, ensuring that the decryption process aligns correctly with how the original encryption was performed.
* **encrypted\_message = decoded\_ciphertext[16:]**  
  After extracting the IV, the remaining part of the decoded ciphertext (i.e., the bytes after the first 16) is the actual encrypted message. This portion will be decrypted to retrieve the original padded message.
* **decrypted\_message = cipher.decrypt(encrypted\_message)**The extracted encrypted message is decrypted using the AES cipher. This operation reverses the encryption, but at this stage, the message is still padded because padding was added to make the original plaintext match the AES block size.
* **plaintext = unpad(decrypted\_message, AES.block\_size)**Since the original message was padded before encryption, this step removes the padding added during that process. The unpad function strips away the extra padding bytes that were appended to make the message fit into AES blocks, restoring the original plaintext.
* **return plaintext.decode('utf-8')**  
  The unpadded plaintext is still in byte format, so it's decoded back into a UTF-8 string (the format of the original message). This final step ensures that the original message is returned as a readable string, just as it was before encryption.

## Chat ending

If the client types “bye” The chat will be ended for the client and on the server “the client has exited the chat” will be displayed.

# Server-Side Implementation (server.py)

## Main Function



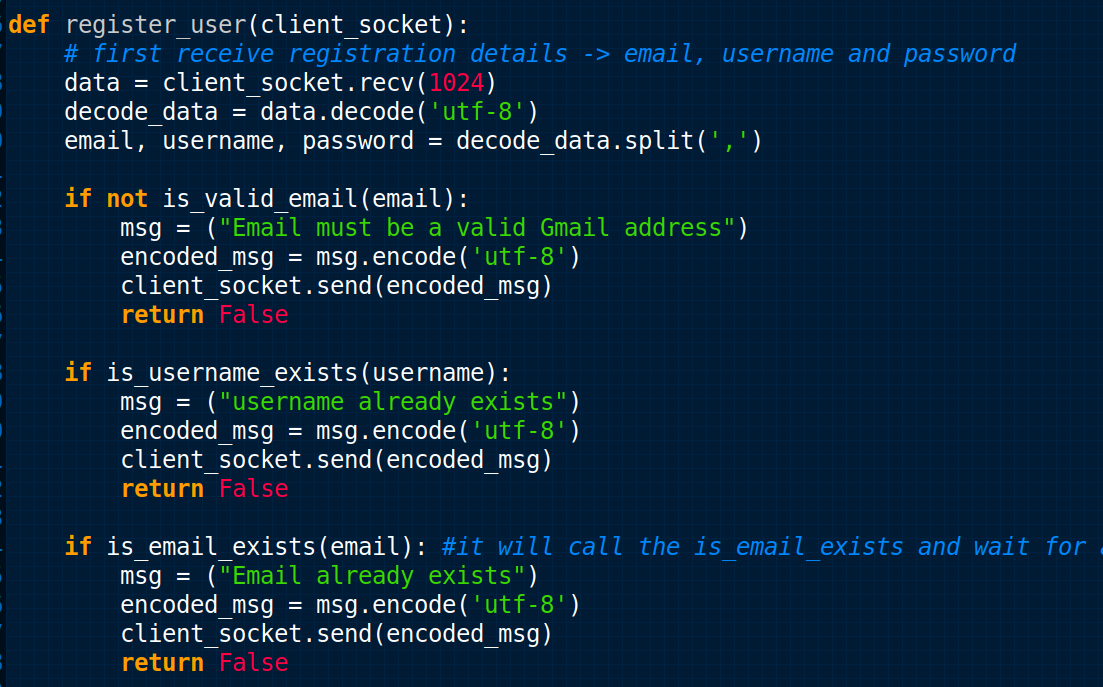
* The main() function initializes the server application, creates a TCP socket, binds it to localhost on port 8080, and prepares to listen for incoming connections.
* How the socket is created and how it works, it is already explained in the client-side implementation (client.py).

## Client handling

The client-handling process in the server involves managing both registration and login functionalities.

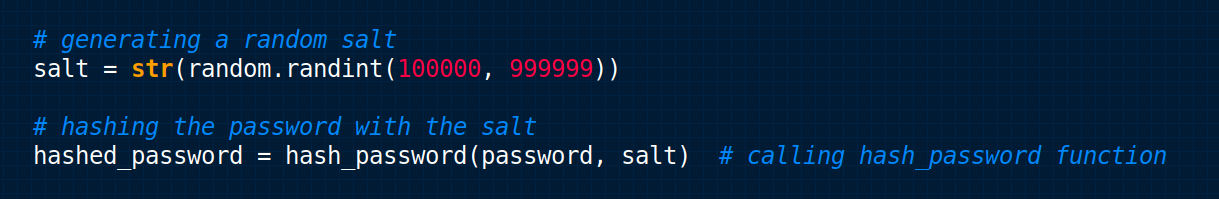
## Registration Process

When the client selects option '1' (for registration), the server handles this by calling the register\_user() function, which is responsible for processing and storing the user's information.

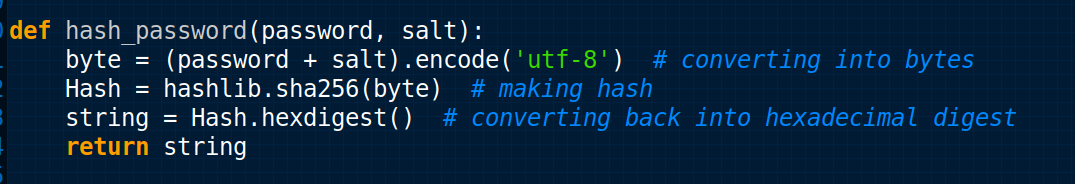


* Inside register\_user(client\_socket), the server receives the registration details (email, username, and password) as a single string, formatted like this: "email,username,password".
* This splits the data into three parts:
* email: The user's email address.
* username: The chosen username.
* password: The plain-text password entered by the user.
* It checks for email validation that the email must end with @gmail.com.
* It checks for the username's existence in cred.txt, if the username already exists it sends a corresponding message to the client.
* It checks the existence of email like username.

## Password hashing

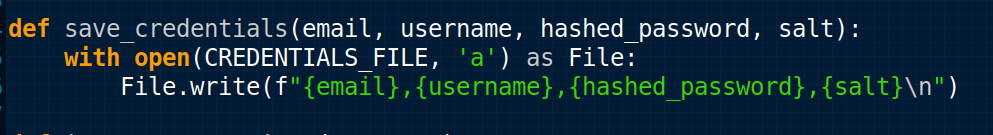


* First, a random salt is generated, which ensures that a different hash will be made even for the same password.
* Then the password along with the random salt is passed to the hashed\_password function.
* Let’s check the hash\_password() function.



* The password + salt is first encoded to bytes using UTF-8 encoding.
* It is then hashed using SHA-256 (hashlib.sha256()), and the resulting hash is converted into a hexadecimal string (hexdigest()), making it suitable for storage in text format.
* In the last step, the resulting string is returned.

## Storing User Information



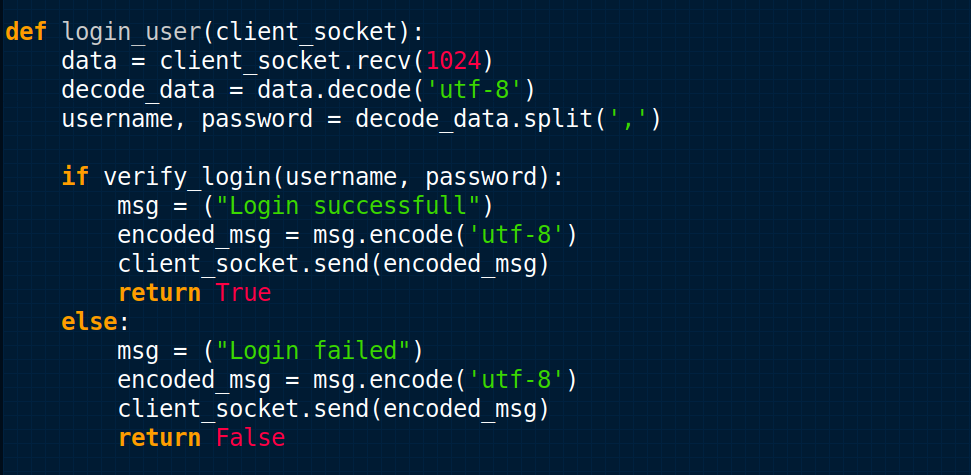
* This appends the user's details to the file, creating a permanent record that can later be used for login verification.
* The data is stored in the format email,username,hashed\_password,salt\n.

## Sending Confirmation

After successfully storing the registration information, the server sends a confirmation message back to the client “registration successful”.

## Login Process

When the client selects option '2' (for login), the server calls the login\_user() function to authenticate the user.



* The server receives the login credentials (username and password) from the client as a single string formatted like "username,password".
* The data is split into two variables username and password.
* First it checks for the username in the cred.txt. If the username is not found in the cred.txt it sends the error message to the client “invalid credentials”.
* if the username is found then take the corresponding salt for password hashing.
* Before verifying the password, the server hashes the plain-text password received from the client using the same SHA-256 hashing algorithm.
* After hashing the password the server compares the hashed password with the cred.txt password, if the password matches it sends the message “login successful” and if not match it sends the error message “login failed”.

## Diffie-Hellman Key Exchange

After successful login, a key is exchanged between the server and client through the Diffie Hellman key exchange protocol. This process is the same as explained in the client side (client.py).

After the key exchange, the chat communication starts between the server and the client. The server waits for the client's message and then sends the response.

## AES encryption

The server message is first encrypted using the AES encryption method before sending it to the client. This whole function is explained in detail in the client-side implementation.

## AES decryption

The server receives the message from the client in encrypted form and it decrypts the message using the AES decryption mechanism by using the mutual shared key. This function is also explained in detail in client-side implementation (client.py).

## Chat ending

An infinite loop for the chat is implemented in the application until the server types “bye” and the chat will be exited from the communication.