NEA Appendix

Table of Contents

Analysis 2
GUI Prototype Code 2
Tkinter-Designer Output 6
Client Server Prototype Code 9
Technical Solution 11
Server and Client-Side SQL Approach Code 11
Server-Side SQL Approach Code 16
AES Code Version 1 19
AES Code Version 2 21
AES Code Version 3 25
Server Version 1 29
NetworkingProtocols.py 32
Server.py 36
Client.py 41
GUI OOP Approach Version 1 51
GUI OOP Approach Version 2 71

Analysis

GUI Prototype Code

```python
import tkinter as tk
from tkinter import *
from tkinter import ttk
from tkinter import scrolledtext
import sqlite3
from sqlite3 import Error
import datetime

'''
colours
Purple = #8575ef
ChatBg = #ececec
Buttons = #2288e6
Black = #000000

'''
cHeader = '#8575ef'
cButton = '#2288e6'
cWhite = '#ffffff'
cGrey = '#ececec'
cBlack = '#000000'
cButtonActive = '#1c75c7'

class tkinterApp(tk.Tk):
  # self is the tkWindow and so has all the relevant functions

    # __init__ function for class tkinterApp
    def __init__(self, *args, **kwargs):
        # colors
        p = '#8575ef'

        # __init__ function for class Tk
        tk.Tk.__init__(self, *args, **kwargs)
        self.title('Orans App [¦87')
        self.resizable(False, False)  # means cant resize window
        self.geometry('800x500')

        # creating a container
        container = tk.Frame(self)
        container.pack(side="top", fill="both", expand=True)

        container.grid_rowconfigure(0, weight=1)
        container.grid_columnconfigure(0, weight=1)

        # initializing frames to an empty dict
        self.frames = {}

        # iterating through a tuple consisting
        # of the different page layouts
        for F in (Login, CreateAccount, HomePage):

            frame = F(container, self)
            # initializing frame of that object from
            # startpage, page1, page2 respectively with
            # for loop

            self.frames[F] = frame

            frame.grid(row=0, column=0, sticky="nsew")

        self.show_frame(Login)

    # to display the current frame passed as
    # parameter
    def show_frame(self, cont):
        frame = self.frames[cont]
        frame.tkraise()

# lables should NOT existing in the __init__ function!

class Login(tk.Frame, tk.Tk):

    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)
        # this line is needed but i honestly dont rly get how it helps with self.controller.show_frame(HomePage)
        self.controller = controller
        # creating frames
        titleFrame = tk.Frame(self, width=800, height=100, bg=cWhite)
        titleFrame.grid(row=0, column=0, sticky='n')
        titleFrame.pack_propagate(False)  # prevents frame resizing

        usernameFrame = tk.Frame(self, width=800, height=100, bg=cGrey)
        usernameFrame.grid(row=1, column=0, sticky='n')
        usernameFrame.pack_propagate(False)

        passwordFrame = tk.Frame(self, width=800, height=100, bg='blue')
        passwordFrame.grid(row=2, column=0, sticky='n')
        passwordFrame.pack_propagate(False)

        ButtonFrame = tk.Frame(self, width=800, height=100, bg='red')
        ButtonFrame.grid(row=3, column=0, sticky='n')
        ButtonFrame.pack_propagate(False)

        # creating lables
        loginTextLable = tk.Label(
            titleFrame, text='Login', foreground=cHeader, font=('Arial', 24))

        usernameLable = tk.Label(usernameFrame, text='Username', bg='blue')
        username = tk.StringVar()  # where the text entered is held
        usernameEntry = ttk.Entry(usernameFrame, textvariable=username)

        passwordLable = ttk.Label(passwordFrame, text='Password')
        password = tk.StringVar()  # where the text entered is held
        passwordEntry = ttk.Entry(
            passwordFrame, textvariable=password, show='*')

        showPasswordButton = tk.Button(passwordFrame, fg=cWhite, bg=cButton, activebackground=cButtonActive, activeforeground=cWhite,
                                       text='Show Password', command=lambda: self.toggle_password(passwordEntry, showPasswordButton))

        loginButton = tk.Button(ButtonFrame, fg=cWhite, bg=cButton, activebackground=cButtonActive,
                                activeforeground=cWhite, text='Login', command=lambda: self.validateLogin(username, password))

        createAccountButton = tk.Button(ButtonFrame, fg=cWhite, bg=cButton, activebackground=cButtonActive,
                                        activeforeground=cWhite, text='Create Account', command=lambda: controller.show_frame(CreateAccount))

        # placing labels
        # .pack(side=TOP, anchor=NW, padx=10, pady=10)
        loginTextLable.pack(pady=20)
        usernameLable.pack(pady=10)
        usernameEntry.pack()

        passwordLable.pack(pady=10)
        passwordEntry.pack()

        showPasswordButton.pack(pady=10)

        loginButton.pack(pady=10)
        createAccountButton.pack()

    def validateLogin(self, username, password):
        print(username.get())
        print(password.get())
        self.controller.show_frame(HomePage)

    def toggle_password(self, passwordEntry, showPasswordButton):
        if passwordEntry.cget('show') == '':
            passwordEntry.config(show='*')
            showPasswordButton.config(text='Show Password')
        else:
            passwordEntry.config(show='')
            showPasswordButton.config(text='Hide Password')

class CreateAccount(Login):

    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)

        # creating lables
        usernameLable = ttk.Label(self, text='Username')
        username = tk.StringVar()  # where the text entered is held
        usernameEntry = ttk.Entry(self, textvariable=username)

        screennameLable = ttk.Label(self, text='Screen name')
        screenname = tk.StringVar()  # where the text entered is held
        screennameEntry = ttk.Entry(self, textvariable=screenname)

        passwordLable = ttk.Label(self, text='Password')
        password = tk.StringVar()  # where the text entered is held
        passwordEntry = ttk.Entry(self, textvariable=password, show='*')

        password2Lable = ttk.Label(self, text='Enter Password Again')
        password2 = tk.StringVar()  # where the text entered is held
        password2Entry = ttk.Entry(self, textvariable=password2, show='*')

        showPasswordButton = ttk.Button(self, text='Show Password', command=lambda: self.toggle_password(
            passwordEntry, showPasswordButton))

        createAccount = ttk.Button(self, text='Create Account', command=lambda: self.validateAccountCreation(
            username, screenname, password, password2, error))

        error = ttk.Label(self, text='Error')
        # placing labels
        usernameLable.grid(row=0, column=0)
        usernameEntry.grid(row=0, column=1)

        screennameLable.grid(row=1, column=0)
        screennameEntry.grid(row=1, column=1)

        passwordLable.grid(row=2, column=0)
        passwordEntry.grid(row=2, column=1)

        password2Lable.grid(row=3, column=0)
        password2Entry.grid(row=3, column=1)

        showPasswordButton.grid(row=2, column=3)
        createAccount.grid(row=4, column=0)

    def validateAccountCreation(self, username, screenname, password, password2, error):
        print(username.get())
        print(password.get())
        print(screenname.get())
        if password.get() != password2.get():
            error.grid(row=5, column=0)
            error.config(text='Passwords must match')
        # check for username clash
        # hash password
        self.addUserRec(username, screenname, password)

    def addUserRec(self, username, screenname, password):
        try:
            conn = sqlite3.connect("userTbl.db")
            cursor = conn.cursor()
            cursor.execute("""INSERT INTO userTbl (Username, Screenname, Password)  VALUES
(?,?,?)""",
                               (username, screenname, password))
            conn.commit()
            print("record added " + username)
        except Error as e:
            print(e)

        # map(lambda x: x*2, [a, b, c])

class Message(tk.Frame):
    def __init__(self, text, parent):
        tk.Frame.__init__(self, parent)
        self.text = text
        message = ttk.Label(parent, text=text)
        message.grid(row=1, column=1)
        print(message.winfo_height())
        # message will have to go in some sort of queueueue
        print(text)

class HomePage(tk.Frame, tk.Tk):
    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)
        self.controller = controller

        def displayMessage(self, message):
            currentTime = datetime.datetime.now()
            date = currentTime.strftime("%x")
            time = currentTime.strftime('%X')
            messageBox.config(state=tk.NORMAL)
            messageBox.insert(tk.END, 'Username: ' +
                                  message + ' [' + time + ']' + '\n')
            messageBox.config(state=tk.DISABLED)
            inputMessage.delete(0, END)

        def sendMessage(self):
            displayMessage(self, message.get())

        self.grid_rowconfigure(0, weight=1)
        # self.grid_columnconfigure(0,weight=1)
        # self.grid_columnconfigure(1,weight=1)

        usersFrame = tk.Frame(self, width=150, height=450, bg=cWhite)
        usersFrame.grid(row=0, column=0, sticky='nsew')
        usersFrame.pack_propagate(False)  # prevents frame resizing
        settingFrame = tk.Frame(self, width=150, height=50, bg=cGrey)
        settingFrame.grid(row=0, column=1, sticky='se')
        settingFrame.pack_propagate(False)

        messageFrame = tk.Frame(self, width=650, height=450, bg=cGrey)
        messageFrame.grid(row=0, column=1, sticky='ne')
        messageFrame.pack_propagate(False)

        inputFrame = tk.Frame(self, width=650, height=50, bg='blue')
        inputFrame.grid(row=1, column=1, sticky='ew')
        inputFrame.pack_propagate(False)

        chatsLabel = tk.Label(usersFrame, text='Chats',
                                  fg=cHeader, bg=cWhite, font=('Arial', 24))
        chatsLabel.pack(pady=10)
        message = tk.StringVar()
        inputMessage = tk.Entry(inputFrame, textvariable=message, width=70)
        inputMessage.pack(padx=10)
        inputMessage.bind('<Return>', sendMessage)

        messageBox = scrolledtext.ScrolledText(messageFrame, width=650)
        messageBox.config(state=tk.DISABLED)
        messageBox.pack(side=BOTTOM)

        activeChatName = 'Active Chat Name'
        recipientNameLabel = Label(
            messageFrame, text=activeChatName, fg=cHeader, bg=cWhite, font=('Arial', 16), height=1)
        recipientNameLabel.pack(side=TOP, anchor=NW, padx=10, pady=10)

if __name__ == '__main__':
    app = tkinterApp()
    app.mainloop()


```

Tkinter-Designer Output

```
# This file was generated by the Tkinter Designer by Parth Jadhav
# https://github.com/ParthJadhav/Tkinter-Designer

from pathlib import Path

# from tkinter import *
# Explicit imports to satisfy Flake8
from tkinter import Tk, Canvas, Entry, Text, Button, PhotoImage

OUTPUT_PATH = Path(__file__).parent
# ASSETS_PATH = OUTPUT_PATH / \
#    Path(r"C:\Users\orank\OneDrive\Desktop\Computer Science\Seperate GUIS\CREATEACCOUNT\build\assets\frame1")

ASSETS_PATH = OUTPUT_PATH / \
    Path(r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level NEA\build\assets\frame0")

def relative_to_assets(path: str) -> Path:
    return ASSETS_PATH / Path(path)

window = Tk()

window.geometry("745x504")
window.configure(bg="#0A0C10")

canvas = Canvas(
    window,
    bg="#0A0C10",
    height=504,
    width=745,
    bd=0,
    highlightthickness=0,
    relief="ridge"
)

canvas.place(x=0, y=0)

# UI

create_account_entry = PhotoImage(
    file=relative_to_assets("create_account_entry.png"))

# USERNAME ENTRY
username_entry_image = canvas.create_image(
    205.5,
    149.5,
    image=create_account_entry
)
username_entry_feild = Entry(
    bd=0,
    bg="#617998",
    fg="#000716",
    highlightthickness=0
)
username_entry_feild.place(
    x=87.5,
    y=133.0,
    width=236.0,
    height=33.0
)

canvas.create_text(
    85.0,
    110.0,
    anchor="nw",
    text="USERNAME",
    fill="#E3E7ED",
    font=("MontserratRoman Regular", 16 * -1)
)

# SCREEN NAME ENTRY
screen_name_entry_image = canvas.create_image(
    205.5,
    217.5,
    image=create_account_entry
)
screen_name_entry_feild = Entry(
    bd=0,
    bg="#617998",
    fg="#000716",
    highlightthickness=0
)
screen_name_entry_feild.place(
    x=87.5,
    y=201.0,
    width=236.0,
    height=33.0
)

canvas.create_text(
    85.0,
    178.0,
    anchor="nw",
    text="SCREEN NAME",
    fill="#E3E7ED",
    font=("MontserratRoman Regular", 16 * -1)
)

# PASSWORD ENTRY
password_entry_bg = PhotoImage(
    file=relative_to_assets("create_account_password_entry.png"))

password_entry_image = canvas.create_image(
    515.5,
    149.5,
    image=password_entry_bg
)
password_entry_feild = Entry(
    bd=0,
    bg="#617998",
    fg="#000716",
    highlightthickness=0,
    show='*'
)
password_entry_feild.place(
    x=422.5,
    y=133.0,
    width=186.0,
    height=33.0
)
```

```

canvas.create_text(
    421.0,
    110.0,
    anchor="nw",
    text="PASSWORD",
    fill="#E3E7ED",
    font=("MontserratRoman Regular", 16 * -1)
)

# CONFIRM PASSWORD ENTRY
confirm_password_image = canvas.create_image(
    540.5,
    217.5,
    image=create_account_entry
)
comfirm_password_feild = Entry(
    bd=0,
    bg="#617998",
    fg="#000716",
    highlightthickness=0,
    show='*'
)
comfirm_password_feild.place(
    x=422.5,
    y=201.0,
    width=236.0,
    height=33.0
)

canvas.create_text(
    421.0,
    178.0,
    anchor="nw",
    text="CONFIRM PASSWORD",
    fill="#E3E7ED",
    font=("MontserratRoman Regular", 16 * -1)
)

# CREATE ACCOUNT HEADER
canvas.create_text(
    227.0,
    35.0,
    anchor="nw",
    text="Create Account",
    fill="#E3E7ED",
    font=("MontserratRoman Bold", 40 * -1)
)

# ERROR MESSAGE
canvas.create_text(
    240.0,
    422.0,
    anchor="nw",
    text="PASSWORDS MUST MATCH",
    fill="#FF4747",
    font=("MontserratRoman Bold", 20 * -1)
)

submit_button_image = PhotoImage(
    file=relative_to_assets("create_account_submit.png"))

submit_button = Button(
    image=submit_button_image,
    borderwidth=0,
    highlightthickness=0,
    command=lambda: print("[BUTTON CLICKED] submit_button"),
    relief="flat",
    activebackground='#0A0C10'
)
```

```python
submit_button.place(
    x=293.0,
    y=266.0,
    width=149.0,
    height=43.0
)

login_option_image = PhotoImage(
    file=relative_to_assets("create_account_login_option.png"))
login_option_button = Button(
    image=login_option_image,
    borderwidth=0,
    highlightthickness=0,
    command=lambda: print("[BUTTON CLICKED] login_option_button"),
    relief="flat",
    activebackground='#0A0C10'
)
login_option_button.place(
    x=227.0,
    y=342.0,
    width=286.0,
    height=48.0
)

# create_account_eye_closed
button_image_3 = PhotoImage(
    file=relative_to_assets("create_account_eye_closed.png"))
button_3 = Button(
    image=button_image_3,
    borderwidth=0,
    highlightthickness=0,
    command=lambda: print("button_3 clicked"),
    relief="flat",
    activebackground='#0A0C10'
)
button_3.place(
    x=633.0,
    y=130.0,
    width=39.0,
    height=40.0
)
window.resizable(False, False)
window.mainloop()


```

Client Server Prototype Code

Client

```python
import socket
import threading
global_host = '192.168.0.52'
local_host = '127.0.0.1'
host = '127.0.0.1'
port = 80  # tcp port

def listen_for_messages_from_server(client):
    while 1:
        message = client.recv(2048).decode('utf-8')
        # message2 = client.recv(2048)
        if message != '':
            print(message)
            # username = message.split('~')[0]
            # content = message.split('~')[1]
            # print(f'[{username}] {content}')
            # print(f'UTF=8: {message2}')
            # client.close()
        else:
            print('Message recieved from client is empty')
```

```python

def send_message_to_server(client):
    while 1:
        message = input('->')
        if message != '':
            client.sendall(message.encode())
        else:
            print('Empty message')
            exit(0)

def communicate_to_server(client):
    username = input('Enter username: ')
    if username != '':
        client.sendall(username.encode())
    else:
        print('Username cannpt be empty')
        exit(0)

    threading.Thread(target=listen_for_messages_from_server,
                     args=(client,)).start()
    send_message_to_server(client)

def main():
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # conect to server
    try:
        client.connect((host, port))
        print(f'Sucsessful connection to {host}')
    except:
        print(f'Unable to connect to server {host} {port}')

    communicate_to_server(client)

if __name__ == '__main__':
    main()


```

Server

```python
import socket
import threading

host = '127.0.0.1'
local_host = '127.0.0.1'  # local host. Allows it to host on itself
global_host = '192.168.0.52'
port = 80  # 0-65535 0 lets OS choose which port you are using #5050 tcp port
listiner_limit = 6  # maximum allowed connections
active_clients = []  # list of all connected users

def listen_for_messages(client, username):  # listen for upcoming messages
    while 1:
        message = client.recv(2048).decode('utf-8')
        if message != '':
            final_msg = username + '~' + message
            send_messages_to_all(final_msg)
        else:
            print(f'Message send from client {username} is empty')

def send_message_to_client(client, message):
    client.sendall(message.encode())  # utf8 is encode
    print(message.encode())
# func to send any new message to ALL clients currently connected

def send_messages_to_all(message):
    for user in active_clients:
        print(user)
        send_message_to_client(user[1], message)

# func to handle client
# client socket object retyurned in the while loop with server.accept()
def client_handler(client):
    # server will listen for clinet message containg the username
    while 1:
        # 2048 is max message size. Decode into utf-8
        username = client.recv(2048).decode('utf-8')
        if username != '':
            active_clients.append((username, client))
            break

        else:
            print('Client username empty')

    threading.Thread(target=listen_for_messages,
                     args=(client, username,)).start()

def main():
    # socket class object
    # AF_INET is IPv4
    # sock_stream is using TCP packets for communication. SOCK_DGRAM is UDP
    # creating server itself
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print('Program is running')

    try:
        server.bind((host, port))
        print('Server running')
    except:
        print(
            f'Unable to bind to host {host} and port {server.getsockname()[1]}')

    # setting server limit - how many possible connections
    server.listen(listiner_limit)

    # keep listing for client connections
    while 1:
        client, address = server.accept()
        print(f"Successfully connected to client {address[0]} {address[1]}")

        # creating a new thread which will preform client_handler func at the same time as
the main loop
        threading.Thread(target=client_handler, args=(client,)).start()

if __name__ == '__main__':
    main()


```

Technical Solution

Server and Client-Side SQL Approach Code

```
import sqlite3
from sqlite3 import Error
from random import randint


"""
status:
req - requested
acc - accepted
blk - blocked
"""

class Database():
    def __init__(self, db_path: str, ) -> None:
```

```python
        self.db_path = db_path
        self.conn = self.create_connection(db_path)

    def create_connection(self, db_file: str):
        """ create a database connection to a SQLite database """
        conn = None
        try:
            conn = sqlite3.connect(db_file, check_same_thread=False)
            print(
                f"\nConnected to Database\nSqlite3 Version: {sqlite3.version}\nDatabase path: {db_file}\n")
        except Error as e:
            print(e)

        return conn

    def create_table(self, sql_create_x_table):
        """Creates a table using value of sql_create_x_table as the sql code"""
        try:
            c = self.conn.cursor()
            c.execute(sql_create_x_table)
        except Error as e:
            print(e)

    def close_connection(self):
        self.conn.commit()
        self.conn.close()

    def execute_update(self, sql: str, values: tuple):
        c = self.conn.cursor()
        try:
            c.execute(sql, values)
            self.conn.commit()
        except sqlite3.Error as e:
            print(f"UPDATE SQL ERROR: {e}\nfor {sql = }\n{values = }")
            self.conn.rollback()
        finally:
            c.close()

    def execute_insert(self, sql: str, values: tuple):
        c = self.conn.cursor()
        try:
            c.execute(sql, values)
            self.conn.commit()
        except sqlite3.Error as e:
            print(f"INSERT SQL ERROR: {e}\nfor {sql = }\n{values = }")
            self.conn.rollback()
        finally:
            c.close()

    # ----------SERVER DATABASE FUNCTIONS----------

    def server_tables(self):
        sql_create_user_table = """
        CREATE TABLE IF NOT EXISTS users(
        user_id text NOT NULL PRIMARY KEY,
        screen_name text NOT NULL,
        hashed_password text NOT NULL,
        salt blob NOT NULL,
        public_key text NOT NULL,
        UNIQUE(user_id)
        );"""
        if self.conn is not None:
            self.create_table(sql_create_user_table)

    def add_user(self, user_id: str, screen_name: str, hashed_password: str, salt: bytes, public_key: str) -> bool:
        """Adds a new user to the database"""
        success = False
        values = (user_id, screen_name, hashed_password, salt, public_key)
        sql = """INSERT INTO users(user_id, screen_name, hashed_password, salt, public_key) VALUES(?,?,?,?,?)"""
        c = self.conn.cursor()
        try:
            c.execute(sql, values)
            self.conn.commit()
            print(f'\n[DB] New user {values} created\n')
            success = True
        except sqlite3.Error as e:
            print(e)
            self.conn.rollback()
        return success

    def get_user_details(self, user_id):
        """Gets user_id, screen_name and public_key"""
        c = self.conn.cursor()
        c.execute(
            "SELECT user_id, screen_name, public_key from users WHERE user_id=?", (user_id,))
        return c.fetchall()[0]

    def get_screen_name(self, user_id: str):
        """Gets the screen_name of the specified user_id"""
        c = self.conn.cursor()
        c.execute(
            "SELECT screen_name FROM users WHERE user_id=?", (user_id,))
        return c.fetchall()[0][0]

    def get_public_key(self, user_id: str):
        """Gets the public key of the specified user_id"""
        c = self.conn.cursor()
        c.execute("SELECT public_key from users WHERE user_id=?", (user_id, ))
        return c.fetchall()[0][0]

    def get_password(self, user_id: str):
        """Gets hashed password of the specified user_id"""
        c = self.conn.cursor()
        c.execute(
            "SELECT hashed_password, salt FROM users WHERE user_id=?", (user_id,))
        return c.fetchall()

    def check_user_id_exists(self, user_id: str) -> bool:
        """Returns True if user exists. False if they do not"""
        c = self.conn.cursor()
        c.execute("""SELECT COUNT(*) FROM users WHERE user_id = ? ;""", (user_id,))
        return c.fetchall()[0][0] != 0

    def update_screen_name_server(self, user_id: str, new_screen_name: str):
        values = (new_screen_name, user_id)
        sql = """
        UPDATE users
        SET screen_name = ?
        WHERE user_id = ?;
        """
        self.execute_update(sql, values)

    def delete_account_server(self, user_id: str):
        success = False
        posfix = randint(10000000, 99999999)
        new_user_id = f"deleted account({posfix})"
        values = (new_user_id, new_user_id, user_id)
        sql = """
        UPDATE users
        SET user_id = ?, screen_name = ?
        WHERE user_id = ?;
        """
        c = self.conn.cursor()
        try:
            c.execute(sql, values)
            self.conn.commit()
            success = True
        except sqlite3.Error as e:
            print(e)
            self.conn.rollback()
            success = False
        finally:
            c.close()
            return success, new_user_id

    # ----------CLIENT DATABASE FUNCTIONS----------

    def client_tables(self):
        sql_create_messages_table = """
        CREATE TABLE IF NOT EXISTS messages (
        message_id INTEGER PRIMARY KEY,
        friend_id text NOT NULL,
        encrypted_Epk blob NOT NULL,
        message_text blob NOT NULL,
        date text NOT NULL,
        time text NOT NULL,
        from_me integer NOT NULL,
        is_image integer NOT NULL,
        FOREIGN KEY (friend_id) REFERENCES friendships(friend_id)
        );"""

        sql_create_friendships_table = """
        CREATE TABLE IF NOT EXISTS friendships (
        friend_id text NOT NULL UNIQUE,
        friend_screen_name text NOT NULL,
        public_key text NOT NULL,
        status text NOT NULL,
        specifier_id text NOT NULL,
        PRIMARY KEY (friend_id)
        );"""

        if self.conn is not None:
            self.create_table(sql_create_messages_table)
            self.create_table(sql_create_friendships_table)

    def new_blocked_friend(self, user_id: str, friend_user_id: str):
        """Sets status to blk where user_id has blocked friend_user_id"""
        values = (user_id, friend_user_id)
        sql = """
        UPDATE friendships
        SET status = 'blk', specifier_id = ?
        WHERE friend_id = ?
        """
        self.execute_update(sql, values)

    def unblocked_friend(self, user_id: str, friend_user_id: str):
        """Sets status to acc where user_id has unblocked friend_user_id"""
        values = (user_id, friend_user_id)
        sql = """
        UPDATE friendships
        SET status = 'acc', specifier_id = ?
        WHERE friend_id = ?
        """
        self.execute_update(sql, values)

    def get_friend_list(self):
        c = self.conn.cursor()
        c.execute("""
            SELECT friend_id, public_key, friend_screen_name, status, specifier_id
            FROM friendships
            WHERE status = 'acc' or status = 'blk';
            """)
        return c.fetchall()

    def get_all_acc_friends_user_ids(self):
        c = self.conn.cursor()
        c.execute("""
            SELECT friend_id
            FROM friendships
            WHERE status = 'acc';
            """)
        return c.fetchall()

    def get_friend_request_list(self, user_id: str):
        c = self.conn.cursor()
        c.execute("""
            SELECT friend_id, public_key
            FROM friendships
            WHERE status = 'req' and specifier_id != ?;
            """, (user_id,))
        return c.fetchall()

    def get_pending_friends_list(self, user_id: str):
        print(f"GETTING PENDING FRIEND LIST")
        c = self.conn.cursor()
        c.execute("""
            SELECT friend_id
            FROM friendships
            WHERE status = 'req' and specifier_id = ?;
            """, (user_id,))
        return c.fetchall()

    def add_new_friend_request(self, friend_id: str, friend_screen_name: str, public_key: str, specifier_id: str):
        print(f"Adding new friend {friend_id = }")
        data = (friend_id, friend_screen_name, public_key, 'req', specifier_id)
        sql = """INSERT INTO friendships (friend_id, friend_screen_name, public_key, status, specifier_id) VALUES (?, ?, ?, ?, ?)"""
        self.execute_insert(sql, data)

    def check_if_user_is_already_friends(self, friend_user_id: str):
        c = self.conn.cursor()
        c.execute(
            "SELECT COUNT(*) FROM friendships WHERE friend_id = ?", (friend_user_id,))
        return c.fetchall()[0][0]

    def accept_friend_request(self, friend_id: str, specifier_id: str):
        values = (specifier_id, friend_id)
        sql = """
        UPDATE friendships
        SET status = 'acc', specifier_id = ?
        WHERE friend_id = ?
        """
        self.execute_update(sql, values)

    def reject_friend_request(self, friend_id):
        c = self.conn.cursor()
        try:
            c.execute("DELETE FROM friendships WHERE friend_id = ?", (friend_id,))
            self.conn.commit()
        except sqlite3.Error as e:
            print(f"DELETION SQL ERROR: {e}")
        finally:
            c.close()

    def get_message_list(self, friend_id: str):
        c = self.conn.cursor()
        c.execute("""
        SELECT encrypted_Epk, message_text, date, time, from_me, is_image
        FROM messages
        WHERE friend_id = ?
        ORDER BY message_id ASC;
        """, (friend_id, ))
        return c.fetchall()

    def store_message(self, friend_id: str, encrypted_Epk: bytes, encrypted_message: bytes, date: str, time: str, from_me: int, is_image: int):
        values = (friend_id, encrypted_Epk,
                  encrypted_message, date, time, from_me, is_image)
        sql = """INSERT INTO messages(friend_id, encrypted_Epk, message_text, date, time, from_me, is_image) VALUES(?,?,?,?,?,?,?)"""
        self.execute_insert(sql, values)

    def update_friend_screen_name(self, friend_id: str, friend_screen_name: str):
        values = (friend_screen_name, friend_id)
        sql = """
        UPDATE friendships
        SET friend_screen_name = ?
        WHERE friend_id = ?;
        """
        self.execute_update(sql, values)

    def friend_deleted_account(self, friend_id: str, account_deletion_name: str):
        values = (account_deletion_name, account_deletion_name, friend_id)
        sql = """
        UPDATE friendships
        SET friend_screen_name = ?, friend_id = ?
        WHERE friend_id = ?;
        """
        self.execute_update(sql, values)


```

Server-Side SQL Approach Code

```
import sqlite3
from sqlite3 import Error
from pathlib import Path

database = r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level NEA\build\appDatabase.db"

def create_connection(db_file):
    """ create a database connection to a SQLite database """
    conn = None
    try:
        conn = sqlite3.connect(db_file)
        print(f"Sqlite3 Version: {sqlite3.version}")
    except Error as e:
        print(e)

    return conn

def get_db(conn):
    c = conn.cursor()
    c.execute("PRAGMA database_list;")
    return c.fetchone()[2]

def create_table(conn, sql_create_x_table):
    try:
        c = conn.cursor()
        c.execute(sql_create_x_table)
    except Error as e:
        print(e)

def add_user(conn, values):
    """Adds a user to the database"""
    sql = """INSERT INTO users(user_id, screen_name, password) VALUES(?,?,?)"""
    c = conn.cursor()
    c.execute(sql, values)
    conn.commit()
    print(f'New User {values} sucsessfully created')
    return c.lastrowid
```

```

def check_login(conn, values):
    """Verifys the users login information"""
    password_match = False
    c = conn.cursor()
    c.execute("SELECT password FROM users WHERE user_id=?", (values[0],))
    try:
        if c.fetchall()[0][0] == values[1]:
            password_match = True
    except:
        password_match = False
    return password_match

def get_screen_name(conn, values):
    """Finds screen_name from user_id"""
    c = conn.cursor()
    c.execute("SELECT screen_name FROM users WHERE user_id=?", (values,))
    return c.fetchall()[0]

def get_user_id(conn, values):
    """Finds user_id from screen_name"""
    c = conn.cursor()
    c.execute("SELECT user_id FROM users WHERE screen_name=?", (values,))
    return c.fetchall()[0][0]

def add_friend(conn, values):
    """Sends a friend request to another user"""
    sql = """INSERT INTO friendships(
                user_one, user_two, status, specifier_id
                )
                VALUES(?,?,?,?)"""
    c = conn.cursor()
    c.execute(sql, values)
    conn.commit()
    print("New Friendship added")
    return c.lastrowid

def update_friend_status(conn, values):
    """Updates an existing friendships status"""
    pass

# def get_friend_list(conn, values):
#     """Retrieves a list of users friends"""
#     c = conn.cursor()
#     c.execute("""SELECT
#                 CASE
#                 WHEN participant_1 = ? THEN participant_2
#                 WHEN participant_2 = ? THEN participant_1
#                 END cgroup
#                 FROM conversations
#                 WHERE participant_1 = ? OR participant_2 = ?""", (values))
#     return c.fetchall()

def get_friend_list(conn, values):
    c = conn.cursor()
    c.execute("""
            SELECT
                c.conversation_id,
                CASE
                    WHEN c.participant_1 = ? THEN u2.user_id
                    WHEN c.participant_2 = ? THEN u1.user_id
                END AS other_user_id,
                CASE
                    WHEN c.participant_1 = ? THEN u2.screen_name
                    WHEN c.participant_2 = ? THEN u1.screen_name
                END AS other_screen_name
            FROM
                conversations c
            JOIN
                users u1 ON c.participant_1 = u1.user_id
            JOIN
                users u2 ON c.participant_2 = u2.user_id
            WHERE
                ? IN (c.participant_1, c.participant_2);
              """, (values))
    return c.fetchall()

def new_message(conn, values):
    sql = """INSERT INTO messages(conversation_id, message_text, date, time, sender_id) VALUES(?,?,?,?,?)"""
    c = conn.cursor()
    c.execute(sql, values)
    conn.commit()
    return c.lastrowid

def add_conversation(conn, values):
    sql = """INSERT INTO conversations(participant_1, participant_2) VALUES(?,?)"""
    c = conn.cursor()
    c.execute(sql, values)
    conn.commit()
    return c.lastrowid

def unique_username(conn, username):
    c = conn.cursor()
    c.execute("""SELECT COUNT(*) FROM users WHERE user_id = ? ;""", (username,))
    return c.fetchall()[0][0] == 0

def get_users(conn):
    c = conn.cursor()
    c.execute("""select * from users""")
    return c.fetchall()

def get_recipient_id(conn, values):
    c = conn.cursor()
    c.execute("""SELECT
                    CASE
                        WHEN participant_1 = ? THEN participant_2
                        WHEN participant_2 = ? THEN participant_1
                    END AS other_user_id
                 FROM conversations
              WHERE conversation_id = ?""", (values))
    return c.fetchall()

def get_message_history(conn, values):
    # get message histroy but replace all user name with the relevant screen name
    c = conn.cursor()
    c.execute(
        """SELECT m.message_id, m.conversation_id, m.message_text, m.date, m.time, m.sender_id, users.screen_name
            FROM messages m
            JOIN users ON m.sender_id = users.user_id
            WHERE m.conversation_id = ?
            ORDER BY m.message_id ASC""", (values,))
    return c.fetchall()

def sql(conn):
    sql_create_user_table = """
    CREATE TABLE IF NOT EXISTS users (
        user_id text NOT NULL PRIMARY KEY,
        screen_name text NOT NULL,
        password text NOT NULL,
        UNIQUE(user_id)
    );"""

    sql_create_conversations_table = """
    CREATE TABLE IF NOT EXISTS conversations (
        conversation_id integer PRIMARY KEY,
        participant_1 text NOT Null,
        participant_2 text NOT Null,
        FOREIGN KEY (participant_1) REFERENCES users(user_id),
        FOREIGN KEY (participant_2) REFERENCES users(user_id),
        CHECK (participant_1 != participant_2)
    );"""

    sql_create_messages_table = """
    CREATE TABLE IF NOT EXISTS messages (
        message_id integer PRIMARY KEY,
        conversation_id INT,
        message_text text NOT Null,
        date text NOT NULL,
        time text NOT NULL,
        sender_id text NOT NULL,
        FOREIGN KEY (conversation_id) REFERENCES conversations(conversation_id)
    );"""

    sql_create_friendships_table = """
    CREATE TABLE IF NOT EXISTS friendships (
        user_one text NOT NULL,
        user_two text NOT NULL,
        status text NOT NULL,
        specifier_id text NOT NULL,
        PRIMARY KEY (user_one, user_two),
        FOREIGN KEY (user_one) REFERENCES users(user_id),
        FOREIGN KEY (user_two) REFERENCES users(user_id),
        FOREIGN KEY (specifier_id) REFERENCES users(user_id),
        CHECK (user_one != user_two),
        CHECK (specifier_id = user_one OR specifier_id = user_two)
    );"""

    if conn is not None:
        create_table(conn, sql_create_user_table)
        create_table(conn, sql_create_conversations_table)
        create_table(conn, sql_create_messages_table)
        create_table(conn, sql_create_friendships_table)
    else:
        print('Error! Cannot create db connection')

conn = create_connection(database)

sql(conn)

```

AES Code Version 1

```
S_BOX = [
    [0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
        0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76],
    [0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
        0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0],
    [0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
        0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15],
    [0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
        0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75],
    [0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
        0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84],
    [0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
        0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf],
    [0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
        0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8],
    [0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
        0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2],
    [0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
        0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73],
```

```python
    [0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
        0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb],
    [0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
        0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79],
    [0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
        0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08],
    [0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
        0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a],
    [0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
        0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e],
    [0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
        0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf],
    [0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
        0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16],
]

CONSTANT_COLUMN = [[0x1, 0, 0, 0],
                   [0x2, 0, 0, 0],
                   [0x4, 0, 0, 0],
                   [0x8, 0, 0, 0],
                   [0x10, 0, 0, 0],
                   [0x20, 0, 0, 0],
                   [0x40, 0, 0, 0],
                   [0x80, 0, 0, 0],
                   [0x1b, 0, 0, 0],
                   [0x36, 0, 0, 0]]

key = 'key123qwertyuioI'
key_schedule = []

def init_variables():
    for i in range(11):
        temp = [[0 for col in range(4)] for row in range(4)]
        key_schedule.append(temp)

# takes in 128 bit key as a string and returns a string of binary
def key_to_bin(key):
    binary = ''
    # {:0>8} means fill with 0 up to a max of 8 characters and align these zeros on the left
    for character in key:
        binary += ('{:0>8}'.format(format(ord(character), 'b')))
    return binary

def rot_word(final_column):
    last = [final_column[-1]]
    rot_column = last + final_column[:-1]
    return (rot_column)

def sub_word(rot_column):
    sub_word = []
    for byte in rot_column:
        # print(byte)
        byte = str(byte)
        row_nibble = int(byte[:4], 2)
        column_nibble = int(byte[4:], 2)
        sub_value = str(bin(int(S_BOX[row_nibble][column_nibble])))
        sub_value = '{:0>8}'.format(sub_value.replace('0b', ''))
        sub_word.append(sub_value)
    return sub_word
    # CHECK

def round_constant(sub_column, current_round):
    round_column = []
    for i in range(4):
        ccb = '{:0>8}'.format(bin(CONSTANT_COLUMN[current_round][i]))
        sc = sub_column[i]
        # print(CONSTANT_COLUMN[current_round][i], int(sc, 2))
        xor = CONSTANT_COLUMN[current_round][i] ^ int(sc, 2)
        xor = '{:0>8}'.format(format(xor, 'b'))
        # print(ccb, sc, xor)
        round_column.append(xor)
    return round_column

def xor(init_column, transformed_column):
    column = []
    for i in range(4):
        xor = int(init_column[i], 2) ^ int(transformed_column[i], 2)
        xor = '{:0>8}'.format(format(xor, 'b'))
        column.append(xor)
    return column

def key_expansion(key):
    binary_key = key_to_bin(key)
    # adding binary_key to key schedule
    for i in range(4):
        for j in range(4):
            key_schedule[0][i][j] = binary_key[0:8]
            binary_key = binary_key[8:]
    # print(key_schedule)
    for current_round in range(10):
        # take last column, rotword, subword, rcon it then add keys
        final_column = key_schedule[current_round][3]
        transformed_column = (round_constant(
            sub_word(rot_word(final_column)), 1))
        # print(transformed_column)
        for i in range(4):
            transformed_column = xor(
                key_schedule[current_round][i], transformed_column)
            key_schedule[current_round+1][i] = transformed_column

def main():
    init_variables()
    key_expansion(key)
    print(key_schedule)

main()
```

AES Code Version 2

```
# GLOBAL VARIABLES
key_schedule = []

encoding = 'utf-8'

S_BOX = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16,
)

INVERSE_S_BOX = (
    82, 9, 106, 213, 48, 54, 165, 56, 191, 64, 163, 158, 129, 243, 215, 251, 124, 227, 57, 130, 155, 47, 255, 135, 52, 142, 67, 68, 196, 222, 233, 203, 84, 123, 148, 50, 166, 194, 35, 61, 238, 76, 149, 11, 66, 250, 195, 78, 8, 46, 161, 102, 40, 217, 36, 178, 118, 91, 162, 73, 109, 139, 209, 37, 114, 248, 246, 100, 134, 104, 152, 22, 212, 164, 92, 204, 93, 101, 182, 146, 108, 112, 72, 80, 253, 237, 185, 218, 94, 21, 70, 87, 167, 141, 157, 132, 144, 216, 171, 0, 140, 188, 211, 10, 247, 228, 88, 5, 184, 179, 69, 6, 208, 44, 30, 143, 202, 63, 15, 2, 193, 175, 189, 3, 1, 19, 138, 107, 58, 145, 17, 65, 79, 103, 220, 234, 151, 242, 207, 206, 240, 180, 230, 115, 150, 172, 116, 34, 231, 173, 53, 133, 226, 249, 55, 232, 28, 117, 223, 110, 71, 241, 26, 113, 29, 41, 197, 137, 111, 183, 98, 14, 170, 24, 190, 27, 252, 86, 62, 75, 198, 210, 121, 32, 154, 219, 192, 254, 120, 205, 90, 244, 31, 221, 168, 51, 136, 7, 199, 49, 177, 18, 16, 89, 39, 128, 236, 95, 96, 81, 127, 169, 25, 181, 74, 13, 45, 229, 122, 159, 147, 201, 156, 239, 160, 224, 59, 77, 174, 42, 245, 176, 200, 235, 187, 60, 131, 83, 153, 97, 23, 43, 4, 126, 186, 119, 214, 38, 225, 105, 20, 99, 85, 33, 12, 125
)

CONSTANT_COLUMN = (0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36)

MATRIX = [[0x2, 0x3, 0x1, 0x1], [0x1, 0x2, 0x3, 0x1],
          [0x1, 0x1, 0x2, 0x3], [0x3, 0x1, 0x1, 0x2]]

INV_MATRIX = [[0x0e, 0x0b, 0x0d, 0x09], [0x09, 0x0e, 0x0b, 0x0d],
              [0x0d, 0x09, 0x0e, 0x0b], [0x0b, 0x0d, 0x09, 0x0e]]

# KEY EXPANSION

def bytes_to_matrix(key):
    # converts 16 bytes in format b'text' into a 4x4 matrix
    return [list(key[j:j+4]) for j in range(0, 16, 4)]

def rot_word(column):
    # print(column)
    # puts item in front of list at back and shifts all forward 1
    return column[1:] + [column[0]]

def sub_word(column, encrypt):
    # uses both nibbles of 1 bit as coordinates for s box subsitution
    if encrypt == True:
        for i in range(4):
            column[i] = S_BOX[column[i]]
    else:
        for i in range(4):
            column[i] = INVERSE_S_BOX[column[i]]
    return column

def round_constant(column, round):
    # xor column with column from constant column dependant on current round
    column[0] ^= CONSTANT_COLUMN[round]
    return column

def xor(a, b):
    # xors 2 same length lists together
    for i in range(len(a)):
        a[i] ^= b[i]
    return a

# INPUT: string
# OUTPUT: NONE
```

```python

def key_expansion(key):
    # key = key.encode(encoding)
    key = bytes_to_matrix(key)
    # Adding first round key
    key_schedule.append(key)
    # key expansion 10 rounds for 128 bit key
    for current_round in range(10):
        round_key = []
        # taking last column and applying set of opperations to it
        final_column = key_schedule[current_round][3]
        transformed_column = round_constant(
            sub_word(rot_word(final_column), True), current_round)
        # using transformed_column to create next round key by xoring with previous round keys
        for i in range(4):
            transformed_column = xor(
                transformed_column, key_schedule[current_round][i])
            # copies value in list rather than list itself
            xor_column = transformed_column[:]
            round_key.append(xor_column)
        key_schedule.append(round_key)
    print(f"key_schedule: {key_schedule} \n\n")

# AES ENCRYPTION
# Most functions here can be used for both encrypt and decrypt
# Hence the 'encrypt' bool that is passed into most functions

def padding(data):
    # padds data to become a multiple of 128 bits
    bytes_remaining = (16-len(data) % 16)
    characters = (chr(bytes_remaining).encode(encoding)) * bytes_remaining
    data += characters
    return data

def remove_padding(data):
    # x = final character in string
    # remove x ammount of characters from end of string
    data = sum(sum(data, []), [])  # moving from 3D list to 1D list
    return data[:-data[-1]]

def sub_bytes(block, encrypt):
    for i in range(len(block)):
        block[i] = sub_word(block[i], encrypt)
    return block

def shift_rows(block):
    for i in range(4):
        block[i] = block[i][i:] + block[i][:i]
    return block

def d_shift_rows(block):
    for i in range(4):
        block[i] = block[i][-i:] + block[i][:-i]
    return block

def rotate(block):
    # swaps rows and columns
    block = list(zip(*block))
    return [list(block[i]) for i in range(len(block))]

def gf_mult(a, b):
    # multiplication in gaussian feild 2^8
    if b == 1:
        # if b i one the result is a as a*1 = 1
        result = a
    else:
        result = 0
        for i in range(8):
            # loop through each bit in b, if it is one add to result
            if (b & 1):
                # if lsb = 1 xor with a
                result ^= a
            a_msb = a & 0x80
            a <<= 1  # shift a left to ensure it is multiplied by the correct power
            if a_msb:
                # multiply here by irriducible polynomial
                a ^= 0x11b
            b >>= 1  # move to next bit in b
    return result

def mix_columns(block, encrypt):
    mixed_columns = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
    if encrypt == True:
        for i in range(4):
            for j in range(4):
                for k in range(4):
                    mixed_columns[i][j] ^= gf_mult(block[i][k], MATRIX[j][k])
    else:
        for i in range(4):
            for j in range(4):
                for k in range(4):
                    mixed_columns[i][j] ^= gf_mult(
                        block[i][k], INV_MATRIX[j][k])
    return mixed_columns

def xor_key(block, round):
    # xors column with specific round key
    return [xor(block[i], key_schedule[round][i]) for i in range(4)]

# INPUT: String
# OUTPUT: 3D list
def encrypt(plain_text, key):
    cipher_text = []
    key_expansion(key)
    # formatting input
    plain_text = plain_text.encode(encoding)
    padded_plain_text = padding(plain_text)
    # Encrpytion Algorithm - encrypts in blocks of 16 bytes
    for i in range(0, len(padded_plain_text), 16):
        block = padded_plain_text[i:i+16]
        block = bytes_to_matrix(block)
        # Start of actual encryption
        # XOR with IV before this step.

        block = xor_key(block, 0)
        for round in range(1, 10):
            block = sub_bytes(block, True)
            block = rotate(shift_rows(rotate(block)))
            block = mix_columns(block, True)
            block = xor_key(block, round)
        block = sub_bytes(block, True)
        block = rotate(shift_rows(rotate(block)))
        block = xor_key(block, 10)
        # IV = block
        cipher_text.append(block)
    return cipher_text

# INPUT: 3D list
# OUTPUT: String
def decrypt(cipher_text, key):
    plain_text = []
    key_expansion(key)
    # Decryption Algorithm - decrypts in blocks of 16 bytes
    for block in cipher_text:
        block = xor_key(block, 10)
        block = rotate(d_shift_rows(rotate(block)))
        block = sub_bytes(block, False)
        for round in range(1, 10):
            block = xor_key(block, 10-round)
```

```python
            block = mix_columns(block, False)
            block = rotate(d_shift_rows(rotate(block)))
            block = sub_bytes(block, False)
        block = xor_key(block, 0)
        plain_text.append(block)
    plain_text = remove_padding(plain_text)
    return (''.join(map(chr, plain_text)))
```

AES Code Version 3

```python
"""
An OOP approach to AES in python.

If you are using OOP you can inherit from aes.Encrypt and aes.Decrypt

Otherwise create instances of each class and use .encrypt() and .decrypt()
"""

# CONSTANTS
S_BOX = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16,
)

INVERSE_S_BOX = (
    82, 9, 106, 213, 48, 54, 165, 56, 191, 64, 163, 158, 129, 243, 215, 251, 124, 227, 57, 130, 155, 47, 255, 135, 52, 142, 67, 68, 196, 222, 233, 203, 84, 123, 148, 50, 166, 194, 35, 61, 238, 76, 149, 11, 66, 250, 195, 78, 8, 46, 161, 102, 40, 217, 36, 178, 118, 91, 162, 73, 109, 139, 209, 37, 114, 248, 246, 100, 134, 104, 152, 22, 212, 164, 92, 204, 93, 101, 182, 146, 108, 112, 72, 80, 253, 237, 185, 218, 94, 21, 70, 87, 167, 141, 157, 132, 144, 216, 171, 0, 140, 188, 211, 10, 247, 228, 88, 5, 184, 179, 69, 6, 208, 44, 30, 143, 202, 63, 15, 2, 193, 175, 189, 3, 1, 19, 138, 107, 58, 145, 17, 65, 79, 103, 220, 234, 151, 242, 207, 206, 240, 180, 230, 115, 150, 172, 116, 34, 231, 173, 53, 133, 226, 249, 55, 232, 28, 117, 223, 110, 71, 241, 26, 113, 29, 41, 197, 137, 111, 183, 98, 14, 170, 24, 190, 27, 252, 86, 62, 75, 198, 210, 121, 32, 154, 219, 192, 254, 120, 205, 90, 244, 31, 221, 168, 51, 136, 7, 199, 49, 177, 18, 16, 89, 39, 128, 236, 95, 96, 81, 127, 169, 25, 181, 74, 13, 45, 229, 122, 159, 147, 201, 156, 239, 160, 224, 59, 77, 174, 42, 245, 176, 200, 235, 187, 60, 131, 83, 153, 97, 23, 43, 4, 126, 186, 119, 214, 38, 225, 105, 20, 99, 85, 33, 12, 125
)

```

```
CONSTANT_COLUMN = (0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36)

MATRIX = [[0x2, 0x3, 0x1, 0x1], [0x1, 0x2, 0x3, 0x1],
          [0x1, 0x1, 0x2, 0x3], [0x3, 0x1, 0x1, 0x2]]

INV_MATRIX = [[0x0e, 0x0b, 0x0d, 0x09], [0x09, 0x0e, 0x0b, 0x0d],
              [0x0d, 0x09, 0x0e, 0x0b], [0x0b, 0x0d, 0x09, 0x0e]]

ENCODING = 'utf-8'

class keyExpansion():

    def key_expansion(self, key: bytes):
        key_schedule = []
        # key = key.encode(encoding)
        key = self.bytes_to_matrix(key)
        # Adding first round key
        key_schedule.append(key)
        # key expansion 10 rounds for 128 bit key
        for current_round in range(10):
            round_key = []
            # taking last column and applying set of opperations to it
            final_column = key_schedule[current_round][3]
            transformed_column = self.round_constant(
                self.sub_word(self.rot_word(final_column), True), current_round)
            # using transformed_column to create next round key by xoring with previous 
round keys
            for i in range(4):
                transformed_column = self.xor(
                    transformed_column, key_schedule[current_round][i])
                # copies value in list rather than list itself
                xor_column = transformed_column[:]
                round_key.append(xor_column)
            key_schedule.append(round_key)
        return key_schedule

    def bytes_to_matrix(self, key: bytes) -> list:
        """converts 16 bytes into a 4x4 matrix"""
        return [list(key[j:j+4]) for j in range(0, 16, 4)]

    def rot_word(self, column):
        """Shifts all items in list forward one with the front most item moving to the 
back"""
        return column[1:] + [column[0]]

    def sub_word(self, column, encrypt):
        # uses both nibbles of 1 bit as coordinates for s box subsitution
        if encrypt == True:
            for i in range(4):
                column[i] = S_BOX[column[i]]
        else:
            for i in range(4):
                column[i] = INVERSE_S_BOX[column[i]]
        return column

    def round_constant(self, column, round):
        # xor column with column from constant column dependant on current round
        column[0] ^= CONSTANT_COLUMN[round]
        return column

    def xor(self, a, b):
        # xors 2 same length lists together
        for i in range(len(a)):
            a[i] ^= b[i]
        return a

class SharedFunctions(keyExpansion):
    def sub_bytes(self, block, encrypt: bool):
        """
        Substitues all bytes in block for relevant bytes in S_BOX
        Parameters:
        - encrypt (bool): If encrypting set True // If decrypting set False
        """
        for i in range(len(block)):
            block[i] = self.sub_word(block[i], encrypt)
        return block

    def rotate(self, block):
        """swaps rows and columns"""
        block = list(zip(*block))
        return [list(block[i]) for i in range(len(block))]

    def gf_mult(self, a: int, b: int) -> int:
        """multiplication in gaussian feild 2^8"""
        if b == 1:
            result = a
        else:
            result = 0
            for i in range(8):
                if (b & 1):  # if lsb = 1 xor with a
                    result ^= a
                sgt255 = a & 0x80
                a <<= 1  # shift a left
                if sgt255:  # modulo
                    a ^= 0x11b
                b >>= 1  # shift b down
        return result

    def mix_columns(self, block, encrypt: bool):
        """
        Uses matrix multiplicatin to mix
        Parameters:
        - encrypt (bool): If encrypting set True // If decrypting set False
        """
        mixed_columns = [[0, 0, 0, 0], [
            0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
        if encrypt == True:
            for i in range(4):
                for j in range(4):
                    for k in range(4):
                        mixed_columns[i][j] ^= self.gf_mult(
                            block[i][k], MATRIX[j][k])
        else:
            for i in range(4):
                for j in range(4):
                    for k in range(4):
                        mixed_columns[i][j] ^= self.gf_mult(
                            block[i][k], INV_MATRIX[j][k])
        return mixed_columns

    def xor_key(self, block, round: int, key_schedule: list):
        # xors column with specific round key
        return [self.xor(block[i], key_schedule[round][i]) for i in range(4)]

class Encrypt(SharedFunctions):

    def encrypt(self, plain_text: bytes, key: bytes) -> bytes:
        cipher_text = []
        key_schedule = self.key_expansion(key)
        # formatting data
        # plain_text = plain_text.encode(ENCODING)
        padded_plain_text = self.padding(plain_text)
        # Encrpytion Algorithm - encrypts in blocks of 16 bytes
        for i in range(0, len(padded_plain_text), 16):
            block = padded_plain_text[i:i+16]
            block = self.bytes_to_matrix(block)
            # Start of actual encryption
            # XOR with IV before this step.

            block = self.xor_key(block, 0, key_schedule)
            for round in range(1, 10):
                block = self.sub_bytes(block, True)
                block = self.rotate(
                    self.encrypt_shift_rows(self.rotate(block)))
                block = self.mix_columns(block, True)
                block = self.xor_key(block, round, key_schedule)
            block = self.sub_bytes(block, True)
            block = self.rotate(self.encrypt_shift_rows(self.rotate(block)))
            block = self.xor_key(block, 10, key_schedule)
            # IV = block
            cipher_text.append(block)

        # Convert the flattened list to a continuous byte stream
            # flattened_cipher_text = [
            #     byte for sublist in cipher_text for byte in sublist]

            # Decode the bytes to UTF-8
            # utf8_result = bytes(flattened_cipher_text)
            ct = sum(sum(cipher_text, []), [])
        return bytes(ct)

    def padding(self, data: bytes) -> bytes:
        """padds data to become a multiple of 128 bits"""
        bytes_remaining = (16-len(data) % 16)
        characters = (chr(bytes_remaining).encode(ENCODING)) * bytes_remaining
        data += characters
        return data

    def encrypt_shift_rows(self, block):
        for i in range(4):
            block[i] = block[i][i:] + block[i][:i]
        return block

class Decrypt(SharedFunctions):

    def decrypt(self, cipher_text: bytes, key: bytes) -> bytes:
        plain_text = []
        key_schedule = self.key_expansion(key)
        # Decryption Algorithm - decrypts in blocks of 16 bytes

        for i in range(0, len(cipher_text), 16):
            block = cipher_text[i:i+16]
            block = self.bytes_to_matrix(block)

            # STARTING THE ACTUAL DECRYPTION NOW BABY
            block = self.xor_key(block, 10, key_schedule)
            block = self.rotate(self.decrypt_shift_rows(self.rotate(block)))
            block = self.sub_bytes(block, False)
            for round in range(1, 10):
                block = self.xor_key(block, 10-round, key_schedule)
                block = self.mix_columns(block, False)
                block = self.rotate(
                    self.decrypt_shift_rows(self.rotate(block)))
                block = self.sub_bytes(block, False)
            block = self.xor_key(block, 0, key_schedule)
            plain_text.append(block)

        # flattened_plain_text = [
        #    byte for sublist in plain_text for byte in sublist]
        pt = self.remove_padding(plain_text)

        # ptt = sum(sum(pt, []), [])

        return bytes(pt)
        # plain_text = self.remove_padding(plain_text)
        # return (''.join(map(chr, plain_text))).encode()

    def remove_padding(self, data):
        # x = final character in string
        # remove x ammount of characters from end of string
        data = sum(sum(data, []), [])  # moving from 3D list to 1D list
        return data[:-data[-1]]

    def decrypt_shift_rows(slef, block):
        for i in range(4):
            block[i] = block[i][-i:] + block[i][:-i]
        return block


```

Server Version 1

```
import socket
import threading
import json
import appDatabase as sql
import atexit

HEADER = 64  # make bigger if new message is needed
PORT = 65432  # TCP/UDP packets

# 127.0.0.1
SERVER = socket.gethostbyname(socket.gethostname())
ADDR = (SERVER, PORT)
FORMAT = 'utf-8'
database = r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level
NEA\build\appDatabase.db"
conn = sql.create_connection(database)

active_clients = []
logged_in_clients = []

class ClientDisconnectException(Exception):
    pass

class USER():
    def __init__(self, client, addr):
        self.user_id = ''
        self.client = client
        self.addr = addr
        self.public_key = None
        self.symmetric_key = None
        self.screen_name = ''
        self.friends = None
        active_clients.append(self)

    def send_data(self, data):
        data = json.dumps(data)
        message = data.encode(FORMAT)
        self.client.send(self.packet_header(message))
        self.client.send(message)

    def packet_header(self, data):
        data_length = len(data)
        send_length = str(data_length).encode(FORMAT)
        send_length += b' ' * (HEADER - len(send_length))
        return send_length

    def recive_data(self):
        data_length = self.client.recv(HEADER).decode(FORMAT)
        if data_length != 0:
            data_length = int(data_length)
            json_data = self.client.recv(data_length).decode(FORMAT)
            data = json.loads(json_data)
        if data['type'] == 'DISCONNECT':
            handel_disconnect(self)
            raise ClientDisconnectException('Client Disconnected')

        else:
            return data

def handel_disconnect(user):
    active_clients.remove(user)
    user.client.close()
    print(
        f'[DISCONNECTED] {user.addr[0], user.addr[1], user.user_id} disconnected')

def ppkp(new_user):
    """Public Private Key Protocol"""
    pass

def verify_login(user_id, password):
    # print("[FUNCTION RUNNING] verify_login")
    """
    1) Hash password
    2) Check password against stored hash with SQL
    3) If a match send login sucsess back to user
    3a) Send Screen name friends list etc back to user
    4) If not send login falirour back to user
    """
    if sql.check_login(conn, (user_id, password)):
        return True
    else:
        return False

def handel_login(new_user):
    valid_password = False
    print(f'[HANDLING] Login for {new_user.addr[0]}')
    login_data = new_user.recive_data()
    user_id = login_data['user_id']
    password = login_data['password']
    if verify_login(user_id, password):
        new_user.user_id = user_id
        # print(f"[USER LOGGED IN]{new_user.user_id}, {new_user.screen_name}")
        valid_password = True
        # send true
    else:
        valid_password = False
    send_validation = {
        'valid_password': valid_password
    }
    new_user.send_data(send_validation)
    if valid_password:
        send_ui_data(new_user)
    return valid_password

def handel_create_account(new_user):
    unique = False
    print('[HANDLING] Create account')
    username = new_user.recive_data()['username']
    # sql checking
    if sql.unique_username(conn, username):
        unique = True
    new_user.send_data({'unique': unique})
    if unique:
        data = new_user.recive_data()
        screen_name = data['screen_name']
        password = data['password']
        new_user.user_id = username
        new_user.screen_name = screen_name
        new_user.password = password
        sql.add_user(
            conn, (new_user.user_id, new_user.screen_name, new_user.password))
    return unique

# --------

def send_ui_data(user):
```

```python
    user.screen_name = sql.get_screen_name(conn, (user.user_id))[0]
    user.friends = sql.get_friend_list(conn, (user.user_id, user.user_id,
                                              user.user_id, user.user_id, user.user_id))
    ui_data = {
        'friend_list': user.friends,
        'screen_name': user.screen_name
    }
    user.send_data(ui_data)

def send_message_to_client(user, conn, data):
    recipient = sql.get_recipient_id(
        conn, (user.user_id, user.user_id, data['chat_id']))
    for c in active_clients:
        if c.user_id == recipient[0][0]:
            c.send_data(data)
            print(f'[SENT MESSAGE] From: {user.user_id} to {recipient[0][0]}')

def handel_inital_contact(new_user):
    """
    1) Client connect so server
    2) Client sends public key to establish a secure tunnle
    3) Client sends symmetrical encryption key
    4) Waits for message dictating login or account creation
    5) handel accoringly by calling different functions
    """
    print(f"[NEW CONNECTION] {new_user.addr[0], new_user.addr[1]} connected.")
    print(f"[TOTAL CONNECTIONS] {len(active_clients)}\n")
    login = False
    while login != True:
        determiner = new_user.recive_data()
        if determiner['choice'] == 'login':
            if handel_login(new_user):
                login = True
        elif determiner['choice'] == 'create account':
            if handel_create_account(new_user):
                login = True
        else:
            print('Error')

    return login

def handel_logged_in_client(user):  # runs for each new client
    handel_client_conn = sql.create_connection(database)
    # client contains info about connected client
    # addr contains just IP and source port eg the port the client is sending from
    print(
        f"[USER LOGIN] {user.addr[0], user.addr[1], user.user_id} logged in.\n")
    # active_clients.append(user)
    connected = True
    # screen_name = sql.get_screen_name(handel_client_conn, user.user_id)
    # screen_name_data = {
    #     'screen_name': screen_name
    # }
    # user.send_data(screen_name_data)
    try:
        while connected:
            data = user.recive_data()

            if data['type'] == 'message history request':
                # print(data['conversation_id'])
                message_history = sql.get_message_history(
                    handel_client_conn, (data['conversation_id']))
                user.send_data({'type': 'message history',
                               'message history': message_history})
                print(
                    f"[DATA SEND] {user.user_id} message history for chat 
{data['conversation_id']}")
            elif data['type'] == 'update screen name':
                print('Updating Screen name')
            else:
                print(
                    f"[RECIEVED MESSAGE] addr[0]: {user.addr[0]}, addr[1]: {user.addr[1]}, {user.user_id} in {data['chat_id']}: {data['message']}")
                sql.new_message(
                    handel_client_conn, (data['chat_id'], data['message'], data['date'], data['time'], user.user_id))
                send_message_to_client(user, handel_client_conn, data)
    except ClientDisconnectException:
        print('[CLIENT DISCONNECT] Logged in TRUE')
        connected = False
    finally:
        return handel_client_conn

def start():
    """Starts the server and calls functions to handel client logic"""

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        server.bind(ADDR)
    except:
        print(f"Unable to bind to server {SERVER} and port {PORT}")

    server.listen()
    print(f"[LISTENING] Server is listening on {SERVER}")

    while True:
        client, addr = server.accept()  # waits till new connection
        new_user = USER(client, addr)
        try:
            login = handel_inital_contact(new_user)
            if login:
                logged_in_clients.append(new_user)
                thread = threading.Thread(
                    target=handel_logged_in_client, args=(new_user, ))
                thread.start()
                print(
                    f"[ACTIVE LOGGED IN CONNECTIONS] {threading.active_count() - 1} \n")
        except ClientDisconnectException:
            print('[CLIENT DISCONNECT] Logged in FALSE')

print("[STARTING] server is starting...")
start()
```

NetworkingProtocols.py

```python
"""
Provides a base class with generic send/recieve functions but with in built data serialization
"""
# Cryptography imports
import rsa
import class_based_aes as aes

# Data serialization imports
import json
import pickle
import base64

HEADER = 2048  # used for send message protocol
FORMAT = 'utf-8'
# PORT = 65432  # TCP/UDP packets
# SERVER = "192.168.0.30"
# ADDR = (SERVER, PORT)

class ClientDisconnectException(Exception):
    pass

class BaseClass(aes.Encrypt, aes.Decrypt):
    def __init__(self, cs: str, client, addr):
        """
        Parameters:
        - type (str): Should be either 'SERVER' or 'CLIENT'.
        - client: (socket)
        - addr: (tuple)
        """
        self.type = cs
        self.client = client
        self.addr = addr

    def validate_signature(self, serialized_data, deseralized_signature) -> bool:
        """Validates a signature for argument passed into serialized_data """
        signature = deseralized_signature['signature']
        public_key = deseralized_signature['public_key']
        valid = False
        try:
            rsa.verify(seralized_data, signature, public_key)
            valid = True
        except:
            valid = False
        finally:
            return valid

    def generate_signature(self, serialized_data: bytes, private_key, public_key) ->
dict[str, bytes]:
        """
        Creates a signature for parameter passed into seralized_data.

        Returns signature along with public key to a dict with keys: signature, public_key
        """
        a = {'signature': rsa.sign(
            seralized_data, private_key, 'SHA-1'), 'public_key': public_key}
        return a

    def add_packet_header(self, data: bytes) -> bytes:
        """Returns bytes of exactly HEADER length with the first few bytes containing the
number of bytes argument data is"""
        data_length = len(data)
        send_length = str(data_length).encode(FORMAT)
        send_length += b' ' * (HEADER - len(send_length))
        return send_length

    def send_with_header(self, data: bytes):
        """Sends a fixed sized packet containing the number of bytes in the next packet"""
        self.client.send(self.add_packet_header(data))
        self.client.send(data)

    def receive_data_with_header(self) -> bytes:
        """Receives the header and handels relevant logic for receiving relevant data
returning the json data"""
        data_length = self.client.recv(HEADER).decode(FORMAT)
        if data_length != 0:
            data_length = int(data_length)
            json_data = self.client.recv(data_length)
            return json_data

    def send_encrypted_data(self, data: dict, Epk: bytes, private_key: rsa.PrivateKey,
public_key: rsa.PublicKey, recipient_public_key: rsa.PublicKey, recipient, return_message=False,
*recipient_user_id):
        """
        Sends data to self.client. Seralizes and encrypts it before sending

        Parameters:
        - data (dict): data you want to send
        - Epk (bytes): One time key used for symmetrical encryption
        - private_key: SENDERS private key
        - public_key: SENDERS public key
        - recipient_public_key
        - recipient: server or client
        - *recipient_user_id: only needed if data is a message from client to client
        """

        seralized_data = self.serialize_dict(data)
        encrypted_data = self.encrypt(seralized_data, Epk)

        encrypted_Epk = rsa.encrypt(Epk, recipient_public_key)

        lump_data = {'encrypted_data': encrypted_data,
                     'encrypted_Epk': encrypted_Epk,
                     'recipient': recipient}

        if data['type'] == 'message':
            lump_data['type'] = 'message'

        if len(recipient_user_id) != 0:
            lump_data['recipient_user_id'] = recipient_user_id[0]

        seralized_lump_data = self.serialize_dict(lump_data)

        signature = self.generate_signature(
            seralized_lump_data, private_key, public_key)

        seralized_signature = self.serialize_dict(signature)

        self.send_with_header(seralized_lump_data)
        self.send_with_header(seralized_signature)

        if return_message:
            return seralized_lump_data

    def recieve_encrypted_data(self, private_key: rsa.PrivateKey, return_public_key=False, return_Epk=False):
        """Recieves encrypted data from self.client returning data + others depending on arguments"""
        seralized_lump_data = self.receive_data_with_header()
        seralized_signature = self.receive_data_with_header()

        if seralized_lump_data != 0 and seralized_signature != 0:
            lump_data = self.deserialize_dict(seralized_lump_data)
            signature = self.deserialize_dict(seralized_signature)

            # If data should NOT be forwarded
            if (self.type == 'SERVER' and lump_data['recipient'] == 'server') or (self.type == 'CLIENT' and lump_data['recipient'] == 'client'):
                if self.validate_signature(seralized_lump_data, signature):
                    encrypted_Epk = lump_data['encrypted_Epk']
                    Epk = rsa.decrypt(encrypted_Epk, private_key)

                    seralized_decrypted_data = self.decrypt(
                        lump_data['encrypted_data'], Epk)
                    data = self.deserialize_dict(seralized_decrypted_data)

                    if self.type == 'SERVER' and data['type'] == 'DISCONNECT':
                        raise ClientDisconnectException('Client Disconnected')
                else:
                    print("[+] Signature fail / message was empty")

                if return_public_key:
                    # data is for either
                    return True, data, signature['public_key']
                elif return_Epk:
                    return True, data, Epk  # data is for either
                else:
                    return True, data  # data is for either
            else:
                # data is for server to forward to another client
                return False, seralized_lump_data, seralized_signature

    def forward_data(self, seralized_data, seralized_signature):
```

```python
        """Sends data without signature or encryption"""
        self.send_with_header(seralized_data)
        self.send_with_header(seralized_signature)

    def send_data(self, data: dict, private_key: rsa.PrivateKey, public_key: rsa.PublicKey):
        """
        Sends data and signature to self.client serialising  it before sending

        Parameters:
        - data: data you want to send
        - private_key: SENDERS private key
        - public_key: SENDERS public_key
        """

        seralized_data = self.serialize_dict(data)

        signature = self.generate_signature(
            seralized_data, private_key, public_key)

        seralized_signature = self.serialize_dict(signature)

        self.send_with_header(seralized_data)
        self.send_with_header(seralized_signature)

    def receive_data(self) -> dict:
        """Receives data from self.client deserializing it before returning"""
        json_data = self.receive_data_with_header()
        json_signature = self.receive_data_with_header()

        signature = self.deserialize_dict(json_signature)

        if json_data != 0 and json_signature != 0 and self.validate_signature(json_data, signature):
            data = self.deserialize_dict(json_data)
            if self.type == 'SERVER' and data['type'] == 'DISCONNECT':
                raise ClientDisconnectException('Client Disconnected')
            else:
                return data
        else:
            print("[+] Signature Invalid/Recieved Empty Mesage")

    def serialize_dict(self, data: dict) -> bytes:
        """Serializes a dictionary and encodes it in a JSON format then encodes it """
        obj_mapping = []  # contains the keys in the dict where the value was an object
        bytes_mapping = []  # contains the keys in the dict where the value was bytes
        for key, value in data.items():
            if self.is_user_defined_class(value):
                data[key] = self.serialize_object(value)
                obj_mapping.append(key)
            elif isinstance(value, bytes):
                data[key] = self.serialize_bytes(value)
                bytes_mapping.append(key)

        # all dicts need 'type' in for sending and recieving purposes in client and server code
        if 'type' not in data:
            data['type'] = 'None'

        data['obj_mapping'] = obj_mapping
        data['bytes_mapping'] = bytes_mapping
        data = json.dumps(data)
        data = data.encode(FORMAT)
        return data

    def deserialize_dict(self, data: bytes) -> dict:
        """Turns serialised bytes into a dictionary also deserializing any objects or bytes too"""

        data = data.decode(FORMAT)
        data = json.loads(data)
        for key, value in data.items():
            if key in data['obj_mapping']:
                data[key] = self.deserialize_object(value)
            elif key in data['bytes_mapping']:
                data[key] = self.deserialize_bytes(value)
        return data

    def serialize_bytes(self, data: bytes) -> str:
        """Returns bytes encoded to base64"""
        b64 = base64.b64encode(data)
        return b64.decode(FORMAT)

    def deserialize_bytes(self, data: str) -> bytes:
        """Returns a string encoded in base64 into bytes"""
        b64 = data.encode(FORMAT)
        return base64.b64decode(b64)

    def serialize_object(self, obj: object) -> str:
        """Returns a serialised string of the object using pickle"""
        pickled_object = pickle.dumps(obj)
        string_of_object = self.serialize_bytes(pickled_object)
        return string_of_object

    def deserialize_object(self, serialized_object: str) -> object:
        """Returns an python object of the serialised_object using pickle"""
        pickled_object = self.deserialize_bytes(serialized_object)
        unpickled_object = pickle.loads(pickled_object)
        return unpickled_object

    def is_user_defined_class(self, obj: object) -> bool:
        """
        Checks if object is an in-built python object or not
        """
        # 'hacky way' of checking taken from: https://stackoverflow.com/questions/14612865/how-to-check-if-object-is-instance-of-new-style-user-defined-class
        if hasattr(obj, '__class__'):
            return (hasattr(obj, '__dict__') or hasattr(obj, '__slots__'))


```

Server.py

```python
# socket imports
from netrworkingProtocols import BaseClass
from netrworkingProtocols import ClientDisconnectException
import socket

# cryptography imports
import rsa
import secrets

# password hashing imports
import bcrypt
import hmac
import hashlib
import keyring

import threading
import serverDatabase

HEADER = 2048  # make bigger if needed
PORT = 65432  # TCP/UDP packets

SERVER = socket.gethostbyname(socket.gethostname())
ADDR = (SERVER, PORT)
```

```python
FORMAT = 'utf-8'

active_clients = []
logged_in_clients = []
server_public_key, server_private_key = rsa.newkeys(512)

database = r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level NEA\OrganisedServerCode\serverDB.db"
sql = serverDatabase.Database(database)
sql.server_tables()

class UserHandler(BaseClass):
    def __init__(self, type: str, client, addr):
        super().__init__(type, client, addr)

        active_clients.append(self)
        self.client_public_key = None
        self.client_user_id = None
        self.can_recieve_msg = False

        print(f"\n[NEW CONNECTION] {self.addr[0], self.addr[1]} connected.")
        print(f"[TOTAL CONNECTIONS] {len(active_clients)}\n")

    def get_name(self):
        """Returns ip, port and client userID"""
        return f"{self.addr[0], self.addr[1], self.client_user_id}"

    def handel_disconnect(self):
        """Removes self from list of active clients and closes connection"""
        active_clients.remove(self)
        self.client.close()
        print(
            f'[CLIENT DISCONNECTED] {self.addr[0], self.addr[1], self.client_user_id}')
        print(f"[TOTAL CONNECTIONS] {len(active_clients)}\n")

    # ------SENDING AND RECIEVING DATA------

    def send_data_to_client(self, data: dict):
        """Sends data to client autofilling public and private key arguments"""
        self.send_data(data,
                       server_private_key, server_public_key)

    def send_encrypted_data_to_client(self, data: dict):
        """Sends encrypted data to client autofilling nessesary arguments"""
        Epk = secrets.token_bytes(16)  # Epk is unique to each new message

        self.send_encrypted_data(
            data, Epk, server_private_key, server_public_key, self.client_public_key, 'client')

    def recieve_encrypted_data_from_client(self):
        """Returns the recieved decrypted data from client autofilling Private Key argument"""
        return self.recieve_encrypted_data(server_private_key)[1]

    # ------HANDEL INITAL CONTACT FUNCTIONS------

    def handel(self):
        """Runs the inital functions to handel the first contact between a client and a server"""
        print(f"[HANDELING LOGGED OUT CLIENT {self.get_name()}]")
        self.handel_inital_contact()
        self.handle_logged_out_client()

    def handel_inital_contact(self):
        self.swap_public_keys()

    def swap_public_keys(self):
        """Swaps public keys with the client setting self.client_public_key in the process"""
        print(
            f"[PUBLIC KEY SWAP] Swapping public keys with {self.get_name()}")
        self.client_public_key = self.receive_data()['public_key']

        self.send_data_to_client(
            {'recipient': 'client', 'public_key': server_public_key})

        print(f"[PUBLIC KEY SWAP FINISHED] with {self.get_name()}\n")

    # ------HANDEL LOGGED OUT CLIENT FUNCTIONS------

    def handle_logged_out_client(self):
        """Handels logged out client by waiting to recieve a determiner specifying what the
user is trying to do and calling functions appropriately"""
        login = False

        while login != True:
            determiner = self.receive_data()
            if determiner['type'] == 'login request':
                if self.handel_login():
                    login = True
            elif determiner['type'] == 'create account request':
                if self.handel_create_account():
                    login = True
        self.handle_logged_in_client()

    def handel_login(self) -> bool:
        """Handels clients login attempt"""
        login_details = self.recieve_encrypted_data_from_client()
        user_exists = sql.check_user_id_exists(
            login_details['user_id'])

        if not user_exists:  # if user does not exist
            valid_password = False
        elif self.is_user_already_online(login_details['user_id']):
            valid_password = False
        else:
            valid_password = self.validate_login_info(
                login_details['user_id'], login_details['password'])

        if valid_password:
            self.client_user_id = login_details['user_id']

        self.send_encrypted_data_to_client(
            {'recipient': 'CLIENT', 'valid_password': valid_password})
        return valid_password

    def handel_create_account(self):
        """Handels clients create account attempt"""
        account_created = False

        details = self.recieve_encrypted_data_from_client()

        user_id_already_used = sql.check_user_id_exists(details['user_id'])

        self.send_encrypted_data_to_client(
            {'recipient': 'CLIENT', 'user_id_already_used': user_id_already_used})

        if not user_id_already_used:  # if user id not taken
            # Creating account
            try:
                user_account_details = self.recieve_encrypted_data_from_client()
                account_created = self.store_user_login_details(
                    user_account_details['user_id'], user_account_details['screen_name'],
user_account_details['password'])
            except Exception as e:
                print(e)
            finally:
                self.send_encrypted_data_to_client(
                    {'recipient': 'CLIENT', 'account created': account_created})
                self.client_user_id = user_account_details['user_id']
            print(
                f'[ACCOUNT CREATED FOR] for {self.get_name()}')
        return account_created

    def is_user_already_online(self, user_id):
        for user in active_clients:
            if user.client_user_id == user_id:
                return True
        return False

    def store_user_login_details(self, user_id, screen_name, password):
        """Adds users id, screen name, hashed password, password salt and public key to the servers database"""
        salt = bcrypt.gensalt()  # salt generated outside function as it needs to be stored
        peppered_hash = self.hash_password(password, salt)
        seralized_public_key = self.serialize_object(
            self.client_public_key)  # sql cant store python objects

        return sql.add_user(user_id, screen_name, peppered_hash,
                            salt, seralized_public_key)

    def hash_password(self, password: bytes, salt: bytes) -> str:
        """Hashes and salts peppers password as per OWASP guidelines 2024"""
        password = password.encode()
        hash = bcrypt.hashpw(password, salt)
        temp = hmac.new(self.get_pepper(), hash, hashlib.sha256)
        peppered_hash = temp.hexdigest()
        return peppered_hash

    def validate_login_info(self, user_id: str, password: str):
        """Checks recieved password against stored hash and returns relevant bool"""
        print(f"[VALIDATING LOGIN FOR {self.get_name()}]")
        valid_password = False
        data = sql.get_password(user_id)
        stored_password = data[0][0]
        salt = data[0][1]
        if self.hash_password(password, salt) == stored_password:
            valid_password = True
        return valid_password

    def get_pepper(self):
        """Gets stored pepper from Windows Key Store"""
        return keyring.get_password("a_level_nea", "oran").encode()

    # ------HANDEL LOGGED IN CLIENT FUNCTIONS----------

    def handle_logged_in_client(self):
        print(f"[HANDELING LOGGED IN USER {self.get_name()}]")
        # second swap required to get the 'real' keys rather than the temp keys
        self.swap_public_keys()

        screen_name = sql.get_screen_name(self.client_user_id)
        self.send_data_to_client(
            {'recipient': 'CLIENT', 'screen_name': screen_name})

        self.recieve_data_from_logged_in_user()

    def recieve_data_from_logged_in_user(self):
        """Recieved encrypted data from logged in user and handels it accordingly"""

        # recieved_data = (for_server: bool, seralized_lump_data: dict, seralized_signature: dict)
        connected = True
        while connected:
            recieved_data = self.recieve_encrypted_data(server_private_key)
            if recieved_data[0]:  # data is for the server
                print(f"[RECIEVED DATA FROM {self.get_name()} FOR SERVER]")
                data = recieved_data[1]
                print(f"{data = }\n\n")
```

```python

                if data['type'] == 'check_if_friend_code_exists':
                    self.send_encrypted_data_to_client(
                        {'exist': sql.check_user_id_exists(data['friend_code'])})

                elif data['type'] == 'get_recipient_public_key':
                    serialized_public_key = sql.get_public_key(
                        data['recipient_user_id'])
                    self.send_encrypted_data_to_client(
                        {'recipient_public_key': serialized_public_key})

                elif data['type'] == 'get_friend_detials':
                    self.send_encrypted_data_to_client(
                        {
                            'screen_name': sql.get_screen_name(data['friend_user_id']),
                            'public_key': sql.get_public_key(data['friend_user_id'])
                        })
                elif data['type'] == 'request_all_user_data':
                    user_details = sql.get_user_details(self.client_user_id)
                    self.send_encrypted_data_to_client(
                        {
                            'user_details': user_details
                        }
                    )
                elif data['type'] == 'deleting_account':
                    account_deleted, account_deletion_name = sql.delete_account_server(
                        self.client_user_id)
                    self.send_encrypted_data_to_client(
                        {
                            'account_deleted': account_deleted,
                            'account_deletion_name': account_deletion_name
                        }
                    )

                elif data['type'] == 'change_screen_name':
                    new_screen_name = data['new_screen_name']
                    print('UPDATING SCREEN NAME')
                    sql.update_screen_name_server(
                        self.client_user_id, new_screen_name)

                elif data['type'] == 'can_recieve_msg_value':
                    self.can_recieve_msg = data['can_recieve_msg']
                    if self.can_recieve_msg:
                        self.recieved_message_queue()

            else:  # lump data is for the client!
                print(f"[RECIEVED DATA FROM {self.get_name()} TO FORWARD]")
                self.forward_data_to_client(recieved_data)

    def forward_data_to_client(self, recieved_data):
        """Attempts to forward data to intended recipient. If it can’t adds to the message queue instead"""
        recipient_found = False
        seralized_lump_data = recieved_data[1]
        seralized_signature = recieved_data[2]
        recipient_user_id = self.deserialize_dict(seralized_lump_data)[
            'recipient_user_id']

        for client in active_clients:
            if client.client_user_id == recipient_user_id and client.can_recieve_msg == True:
                from_message_queue = self.get_name() == client.get_name()
                print(
                    f"[FORWARDING DATA FROM {self.get_name()} TO {client.get_name()}] FROM MESSAGE QUEUE {from_message_queue}")
                client.forward_data(seralized_lump_data, seralized_signature)
                recipient_found = True
                break

        if not recipient_found:
            # adding recipient_user_id as it means deserialization is not needed for each item when searching queue
            message_queue.enQueue(recipient_user_id, recieved_data)

    def recieved_message_queue(self):
        """Sends messages waiting in message_queue to relevant client"""
        for message in message_queue.get().copy():
            if message[0] == self.client_user_id:
                message_queue.deQeueu(self.client_user_id, message[1])
                self.forward_data_to_client(message[1])

class MessageQueue():
    def __init__(self):
        self.__items = []

    def isEmpty(self):
        return len(self.__items) == 0

    def enQueue(self, client_recieving_user_id: str, message: list):
        values = (client_recieving_user_id, message)
        self.__items.append(values)

    def deQeueu(self, client_recieving_user_id: str, message: list):
        if not self.isEmpty():
            values = (client_recieving_user_id, message)
            try:
                self.__items.remove(values)
            except Exception as e:
                print(f"[deQueue ERROR] {e}")

    def get(self):
        return self.__items

def worker(client, addr):
    # allows disconnect excpetion to be handeled within each new users thread rather than having to leave the thread
    new_user = UserHandler('SERVER', client, addr)
    try:
        new_user.handel()
    except ClientDisconnectException:
        new_user.handel_disconnect()

# ------DRIVER CODE------

def main():
    # create new socket object
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        # bind socket to port and IP
        server.bind(ADDR)
    except:
        print(f"Unable to bind to server {SERVER} and port {PORT}")
    server.listen()
    print(f"[LISTENING] Server is listening on {SERVER} {PORT}")
    while True:
        client, addr = server.accept()  # waits till new connection
        thread = threading.Thread(target=worker, args=(client, addr))
        thread.start()

message_queue = MessageQueue()
main()


```

Client.py

```
# Cryptography imports
import secrets
import rsa

# socket imports
from netrworkingProtocols import BaseClass
import socket

# file handling and database imports
from tkinter import filedialog
import serverDatabase
from PIL import Image
import os

# hashing imports
from hashlib import md5  # used in secret keeping NOT for password hashing

import threading

HEADER = 2048  # used for send message protocol
PORT = 65432  # TCP/UDP packets
FORMAT = 'utf-8'
SERVER = "192.168.0.30"
ADDR = (SERVER, PORT)
CLIENT = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

class Client(BaseClass):
    def __init__(self, cs='CLIENT', client=CLIENT, addr=ADDR):
        super().__init__(cs, client, addr)
        self.client = client
        self.sql = None

        self.server_public_key = None
        self.public_key = None
        self.private_key = None
        self.master_key = None
        self.password_hash = None  # used for locally storing sensative data

        self.user_path = None
        self.image_path = None
        self.image_name_and_format = None
        self.user_id = ''
        self.screen_name = ''

        self.friend_list = None
        self.friend_request_list = None
        self.pending_friend_list = None
        self.current_message_history = []

        self.connected = False
        self.logged_in = False

        self.can_recieve_msg = False
        self.listen_thread = None
        self.stop_event = threading.Event()

        self.ChatPage = None
        self.AddFriendPage = None

        print('[CLIENT INSTANCE CREATED]')

    def connect(self) -> bool:
        """
        Attemps to connect to the server specified in the ADDR variable.

        Returns:
            - True if connection sucsessful
            - False if not
        """

        print("[ATTEMPTING TO CONNECT TO SERVER]")
        try:
```

```python
            self.client.connect(ADDR)
            self.connected = True
        except Exception as e:
            print(f"[ERROR] Client was unable to connect to server: {ADDR}")
            self.connected = False
        print(f"[CONNECTED] {self.connected}")
        return self.connected

    def establish_inital_contact(self):
        """establishes initial contact with the server by generating public and private keys and swapping them"""
        print(f"[establish_inital_contact STARTED]")
        self.generate_init_keys()
        self.swap_public_keys()
        print(f"[establish_inital_contact FINISHED]")

    def send_disconnect_message(self):
        print("[SENDING DISCONNECT MESSAGE]")
        if self.logged_in:
            self.send_encrypted_data_to_server({'type': 'DISCONNECT'})
        else:
            self.send_data_to_server({'type': 'DISCONNECT'})
        print("[DISCONNECTED FROM SERVER]")

    def close_client(self):
        print('[CLOSING CLIENT]')
        self.client.close()

    def connect_to_database(self):
        """Creates and or connects to database in self.user_path"""
        db_path = os.path.join(self.user_path, 'user_data.db')
        self.sql = serverDatabase.Database(db_path)

    def close_db_connection(self):
        if self.logged_in:
            print('[CLOSING DATABASE]')
            self.sql.close_connection()

    def send_data_to_server(self, data):
        """Sends data to server autofilling public and private key parameter"""
        self.send_data(data,
                       self.private_key, self.public_key)

    def send_encrypted_data_to_server(self, data: dict):
        """Creates Epk and sends encrypted data to server"""
        print(f"[SENDING ENCRYPTED DATA TO SERVER] {data}")
        Epk = secrets.token_bytes(16)

        self.send_encrypted_data(
            data, Epk, self.private_key, self.public_key, self.server_public_key, 'server')

    def recieve_encrypted_data_from_server(self):
        """returns decrypted recieved data from server autofilling private key parameter"""
        return self.recieve_encrypted_data(self.private_key)[1]

    def send_encrypted_data_to_recipient(self, data, recipient_user_id, return_confg_data=False):
        """Creates Epk, gets recipients private key and sends data to server to forward to recipient"""
        print(f"[SENDING ENCRYTED DATA TO {recipient_user_id}] {data}")
        Epk = secrets.token_bytes(16)

        data['sender'] = self.user_id
        data['public_key'] = self.public_key

        # getting recipient public key from server
        self.send_encrypted_data_to_server(
            {'type': 'get_recipient_public_key',
             'recipient_user_id': recipient_user_id}
        )
        seralised_recipinet_public_key = self.recieve_encrypted_data_from_server()[
            'recipient_public_key']
        recipinet_public_key = self.deserialize_object(
            seralised_recipinet_public_key)

        encrypted_message = self.send_encrypted_data(
            data, Epk, self.private_key, self.public_key, recipinet_public_key, 'client', 
return_confg_data, recipient_user_id)

        if return_confg_data:
            return Epk

    # --------CREATE/LOGIN/DELETE ACCOUNT ----------

    def login(self, user_id: str, password: str):
        print("[ATTEMPING TO LOGIN]")
        login = False

        self.send_data_to_server({'type': 'login request'})
        self.send_encrypted_data_to_server(
            {'user_id': user_id, 'password': password})

        if self.recieve_encrypted_data_from_server()['valid_password']:
            login = True
            self.user_id = user_id
            self.password_hash = md5(password.encode()).digest()
            self.user_path = os.path.join('App./Users./', user_id)
            self.user_images_path = os.path.join(self.user_path, 'images')
            self.connect_to_database()
            self.get_keys_from_file()
            self.swap_public_keys()
        print(f"[LOGIN {login}]")
        return login

    def create_account(self, user_id: str, password: str, screen_name: str):
        print(f"[HANDELING create_account STARTED]")
        print(f"[VALIDATING ACCOUNT CREATION DETAILS]")

        valid_user_id = False
        account_created = False

        self.send_data_to_server({'type': 'create account request'})
        self.send_encrypted_data_to_server(
            {'user_id': user_id, 'recipient': 'SERVER'})

        if not self.recieve_encrypted_data_from_server()['user_id_already_used']:
            valid_user_id = True
            # user ID is unique therefore try to create account then send details to server
            self.password_hash = md5(password.encode()).digest()

            self.user_path = os.path.join('App./Users./', user_id)
            self.user_images_path = os.path.join(self.user_path, 'images')
            try:  # if new user
                os.makedirs(self.user_images_path)
                self.write_keys_to_file()
                self.connect_to_database()
                self.sql.client_tables()

                self.send_encrypted_data_to_server(
                    {'user_id': user_id, 'password': password, 'screen_name': screen_name})

                if self.recieve_encrypted_data_from_server()['account created']:
                    account_created = True
            except:  # if user already exists locally - should never occur
                print(f"[ERROR] User: {user_id} already exists")
                valid_user_id = False

        print(f"[HANDELING create_account FINISHED]")
        if valid_user_id and account_created:
            self.user_id = user_id
```

```python
            self.swap_public_keys()
        return valid_user_id, account_created

    def delete_account(self):
        """
        Sends account deletion notification to server and all friends

        Returns if it was sucsessful or not
        """
        account_deleted = False
        self.send_encrypted_data_to_server({'type': 'deleting_account'})
        details = self.recieve_encrypted_data_from_server()
        if details['account_deleted']:

            account_deleted = True
            friend_user_ids = self.sql.get_all_acc_friends_user_ids()
            flat_friend_user_id = [
                id for tuple in friend_user_ids for id in tuple]
            for friend_id in flat_friend_user_id:
                self.send_encrypted_data_to_recipient(
                    {'type': 'sync_account_deletion', 'account_deletion_name': details['account_deletion_name']}, friend_id)
        return account_deleted

    def delete_directory(self):
        """Deletes everything in the users account directory and then the directory itself"""
        print("[DELETING ACCOUNT]")
        for filename in os.listdir(self.user_path):
            if os.path.isfile(os.path.join(self.user_path, filename)):
                os.remove(os.path.join(self.user_path, filename))
        for filename in os.listdir(self.user_images_path):
            if os.path.isfile(os.path.join(self.user_images_path, filename)):
                os.remove(os.path.join(self.user_images_path, filename))
        os.rmdir(self.user_images_path)
        os.rmdir(self.user_path)

    # -------PUBLIC AND PRIVATE KEYS-------

    def generate_init_keys(self):
        """Generates the clients public, private and master keys"""
        self.public_key, self.private_key = rsa.newkeys(512)
        self.master_key = secrets.token_bytes(16)

    def write_keys_to_file(self):
        """Writes clients public, private and master keys to local file in self.user_path"""

        # Encrypting and seralizing keys
        seralized_private_key = self.serialize_object(
            self.private_key).encode()
        key_data = {
            'public_key': self.public_key,
            'private_key': self.encrypt(seralized_private_key, self.password_hash),
            'master_key': self.encrypt(self.master_key, self.password_hash)
        }
        seralized_key_data = self.serialize_dict(key_data)

        with open(os.path.join(self.user_path, 'keys.txt'), 'wb') as keys:
            keys.write(seralized_key_data)

    def get_keys_from_file(self):
        """Retrieves clients public, private and master keys from local file in self.user_path"""

        # Reading data from file
        with open(os.path.join(self.user_path, 'keys.txt'), 'rb') as keys:
            serialized_key_data = keys.read()

        # Decrypting and deseralizing keys
        key_data = self.deserialize_dict(seralized_key_data)
        self.public_key = key_data['public_key']
        self.master_key = self.decrypt(
            key_data['master_key'], self.password_hash)
        seralized_private_key = self.decrypt(
            key_data['private_key'], self.password_hash).decode()
        self.private_key = self.deserialize_object(seralized_private_key)

    def swap_public_keys(self):
        """sends public key to server and recieves servers public key"""
        self.send_data_to_server(
            {'recipient': 'server', 'public_key': self.public_key})
        self.server_public_key = self.receive_data()['public_key']

    # -------GETTING FRIENDS LISTS-------

    def get_friend_list(self):
        """Sets self.friend_list to equal users friend list"""
        self.friend_list = self.sql.get_friend_list()

    def get_friend_request_list(self):
        """Sets self.friend_request_list = list of incoming friend requests"""
        self.friend_request_list = self.sql.get_friend_request_list(
            self.user_id)

    def get_pending_friends_list(self):
        """Sets self.pending_friend_list = list of outgoing friend requests"""
        self.pending_friend_list = self.sql.get_pending_friends_list(
            self.user_id)
        print(self.pending_friend_list)

    # -------LISTEN FOR MESSAGES-------

    def handel_logged_in_client(self):
        self.logged_in = True
        self.screen_name = self.receive_data()['screen_name']

    def listen(self):
        """Send message to server saying it can listen and starts the listening thread"""
        self.can_recieve_msg = True
        self.send_encrypted_data_to_server(
            {'type': 'can_recieve_msg_value', 'can_recieve_msg': self.can_recieve_msg})

        self.stop_event.clear()
        self.listen_thread = threading.Thread(
            target=self.recieving_data_from_client)
        self.listen_thread.start()

    def stop_listen(self):
        """Send message to server saying it can't listen and stops the listening thread"""
        try:
            self.can_recieve_msg = False
            self.send_encrypted_data_to_server(
                {'type': 'can_recieve_msg_value', 'can_recieve_msg': self.can_recieve_msg})

            self.stop_event.set()
            self.listen_thread.join()
            print('STOP Listen')
        except Exception as e:
            print(e)
            pass

    def recieving_data_from_client(self):
        while not self.stop_event.is_set():
            try:
                self.client.settimeout(2)
                # 2 will also be the delay time when stopping the thread
                total_data = self.recieve_encrypted_data(
                    self.private_key, False, True)
                recieved_data = total_data[1]
                Epk = total_data[2]
                if recieved_data['type'] == 'friend_request':
                    seralized_public_key = self.serialize_object(
                        recieved_data['public_key'])
                    self.sql.add_new_friend_request(
                        recieved_data['sender'], recieved_data['screen_name'], seralized_public_key, recieved_data['sender'])
                    self.get_friend_request_list()
                elif recieved_data['type'] == 'accepted_friend_request':
                    self.sql.accept_friend_request(
                        recieved_data['sender'], recieved_data['sender'])
                    self.get_friend_list()
                    self.ChatPage.update_friend_list()
                elif recieved_data['type'] == 'rejected_friend_request':
                    self.sql.reject_friend_request(recieved_data['sender'])
                    self.get_friend_list()
                elif recieved_data['type'] == 'blocked':
                    self.sql.new_blocked_friend(
                        recieved_data['sender'], recieved_data['sender'])
                    self.get_friend_list()
                    self.ChatPage.update_friend_list()
                elif recieved_data['type'] == 'unblocked':
                    self.sql.unblocked_friend(
                        recieved_data['sender'], recieved_data['sender'])
                    self.get_friend_list()
                    self.ChatPage.update_friend_list()
                elif recieved_data['type'] == 'sync_new_screen_name':
                    new_friends_screen_name = recieved_data['new_screen_name']
                    friend_id = recieved_data['sender']
                    self.sql.update_friend_screen_name(
                        friend_id, new_friends_screen_name)
                elif recieved_data['type'] == 'sync_account_deletion':
                    account_deletion_name = recieved_data['account_deletion_name']
                    friend_id = recieved_data['sender']
                    self.sql.friend_deleted_account(
                        friend_id, account_deletion_name)
                elif recieved_data['type'] == 'message':
                    self.handel_recieved_message(recieved_data, Epk)

            except socket.timeout:
                continue

    # -------SENDING/RECIEVING MESSAGES-------

    def handel_send_message(self, data: dict):
        """Sends message to server and then stores it"""
        print("[SENDING MESSAGE]")
        self.stop_listen()
        Epk = self.send_encrypted_data_to_recipient(
            data, data['recipient'], True)
        message = data['message']
        is_image = 0
        if data['is_image']:
            message = self.store_sent_image_to_files(self.image_path)
            is_image = 1

        self.store_sent_message(
            data, Epk, self.encrypt(message.encode(), Epk), is_image)
        self.listen()

    def handel_recieved_message(self, data: dict, Epk: bytes):
        """Handels recieved message accordingly if it is text or an image"""
        message = data['message']

        if data['is_image']:
            image_data = data['message']
            image_name_and_format = data['image_name_and_format']
            message = self.store_image_to_files(
                image_name_and_format, image_data)
            self.image_path = message
```

```python

        message = message.encode()
        encrypted_message = self.encrypt(message, Epk)
        self.store_recieved_message(data, Epk, encrypted_message)

        # message_details = (x, x, x ,x from_me, is_image)
        message_details = (
            Epk, encrypted_message, data['date'], data['time'], 0, data['is_image'])
        message = self.decrypt_message(message_details, False)

        # if chat related recieved message is open display it
        if data['sender'] == self.ChatPage.active_chat_user_details.get().split(' ')[0]:
            formatted_message = self.ChatPage.format_stored_message_for_display(
                message)
            self.ChatPage.display_message(formatted_message)

    def decrypt_message(self, message_details, Epk_encrypted=True):
        """Returns tuple (decrypted_message, date, time, from_me, is_image)"""
        Epk = message_details[0]
        encrypted_message = message_details[1]
        date = message_details[2]
        time = message_details[3]
        from_me = message_details[4]
        is_image = message_details[5]

        if Epk_encrypted:
            Epk = self.decrypt(Epk, self.master_key)

        decrypted_message = self.decrypt(encrypted_message, Epk).decode()
        return (decrypted_message, date, time, from_me, is_image)

    def store_sent_message(self, data: dict, Epk: bytes, encrypted_message: bytes, is_image: int):
        """Stores sent message encrypting the Epk with self.master_key"""
        encrypted_Epk = self.encrypt(Epk, self.master_key)
        self.sql.store_message(
            data['recipient'], encrypted_Epk, encrypted_message, data['date'], data['time'], 1, is_image)

    def store_recieved_message(self, data: dict, Epk: bytes, encrypted_message: bytes):
        """Stores recieved message encrypting the Epk with self.master_key"""
        encrypted_Epk = self.encrypt(Epk, self.master_key)
        self.sql.store_message(
            data['sender'], encrypted_Epk, encrypted_message, data['date'], data['time'], 0, data['is_image'])

    def get_message_history(self, friend_id: str):
        return self.sql.get_message_list(friend_id)

    def decrypt_message_history(self, friend_id: str):
        self.current_message_history = []
        encrypted_message_history = self.get_message_history(friend_id)

        for message_details in encrypted_message_history:
            values = self.decrypt_message(message_details)
            self.current_message_history.append(values[:])

    # -------Add Friend Page fuctions-------

    def block_friend(self, friend_user_id):
        self.sql.new_blocked_friend(self.user_id, friend_user_id)
        self.send_encrypted_data_to_recipient(
            {'type': 'blocked'}, friend_user_id)

    def unblock_friend(self, friend_user_id):
        self.sql.unblocked_friend(self.user_id, friend_user_id)
        self.send_encrypted_data_to_recipient(
            {'type': 'unblocked'}, friend_user_id)

    def accept_friend_request(self, friend_id: str, specifier_id: str):
        self.sql.accept_friend_request(friend_id, specifier_id)
        self.send_encrypted_data_to_recipient(
            {'type': 'accepted_friend_request',
             'friend_id': self.user_id
             }, friend_id)

    def reject_friend_request(self, friend_id: str):
        self.sql.reject_friend_request(friend_id)
        self.send_encrypted_data_to_recipient(
            {'type': 'rejected_friend_request',
             'friend_id': self.user_id
             }, friend_id)

    def send_friend_request(self, friend_user_id: str):
        self.send_encrypted_data_to_recipient(
            {'type': 'friend_request',
             'screen_name': self.screen_name
             }, friend_user_id)
        self.store_friend_request(friend_user_id)
        self.AddFriendPage.update_pending_friends_list(friend_user_id)

    def store_friend_request(self, friend_user_id: str):
        """Gets friend details from server for an accepted friend request and stores them"""
        self.send_encrypted_data_to_server(
            {'type': 'get_friend_detials', 'friend_user_id': friend_user_id})
        details = self.recieve_encrypted_data_from_server()
        self.sql.add_new_friend_request(
            friend_user_id, details['screen_name'], details['public_key'], self.user_id)

    def check_if_user_is_already_friends(self, friend_user_id: str):
        already_friends = True
        friend_count = self.sql.check_if_user_is_already_friends(
            friend_user_id)
        if friend_count == 0:
            already_friends = False
        return already_friends

    def check_if_friend_code_exists(self, freind_user_id: str) -> bool:
        self.send_encrypted_data_to_server(
            {'type': 'check_if_friend_code_exists', 'friend_code': freind_user_id})
        return self.recieve_encrypted_data_from_server()['exist']

    # -------SENDING IMAGES-------

    def get_image_data(self, image_path: str):
        """Returns binary data of image at image_path"""
        image_file = open(image_path, 'rb')
        return image_file.read()

    def store_sent_image_to_files(self, image_path: str):
        """
        Gets sent image sent image path and then stores it

        Returns stored image path
        """
        image_name_and_format = image_path.split('/')[-1]

        image_file = open(image_path, 'rb')

        image_data = image_file.read()
        return self.store_image_to_files(image_name_and_format, image_data)

    def store_image_to_files(self, image_name_and_format, image_data):
        """Stores image to self.user_images_path"""
        new_image_path = self.find_sutable_image_path(image_name_and_format)
        new_image_file = open(new_image_path, 'wb')
        new_image_file.write(image_data)
        return new_image_path

    def find_sutable_image_path(self, image_name_and_format: str):
        """
        Finds sutable image name by adding (i) if there are duplicates

        Returns full image path
        """
        counter = 0
        image_path = os.path.join(self.user_images_path, image_name_and_format)
        path_name, extention = os.path.splitext(image_path)

        base_path = f"{path_name}({counter}){extention}"
        while os.path.isfile(base_path):
            counter += 1
            base_path = f"{path_name}({counter}){extention}"
        return base_path

    def compress_image(self, image_path: str):
        """
        Compresses an image to:
            - max 100px,100px
            - LANCZOS resampling
            - save quality: 65%

        Returns new image path
           """
        old_path, extension = os.path.splitext(image_path)
        file_name = old_path.split('/')[-1]
        split_path = image_path.split('/')[:-1]
        new_path = 
f"{'/'.join(split_path)}/{file_name}(downsized_for_encryption){extension}"

        image = Image.open(image_path)
        image.thumbnail((100, 100), Image.Resampling.LANCZOS)
        image.save(new_path, optimize=True, quality=65)
        return new_path

    def client_get_image_path(self):
        """
        Displays popup asking user to select an image file

        Returns selected image path
        """

        image_path = filedialog.askopenfilename(initialdir="/downloads",
                                                title="Select Image",
                                                filetypes=(("jpeg files", "*.jpeg"), ("png 
files", "*.png")))
        if not image_path:  # if user cancels on file explorer
            return False

        self.image_path = self.compress_image(image_path)
        self.image_name_and_format = self.image_path.split('/')[-1]

        return self.image_path

    # -------ACCOUNT MODIFICATION-------

    def change_screen_name(self, new_screen_name):
        """Notifys the server of screen name change"""
        self.screen_name = new_screen_name
        self.send_encrypted_data_to_server(
            {'type': 'change_screen_name', 'new_screen_name': self.screen_name})

        friend_user_ids = self.sql.get_all_acc_friends_user_ids()

        # flatten list
        flat_friend_user_id = [id for tuple in friend_user_ids for id in tuple]
        for friend_id in flat_friend_user_id:
            self.send_encrypted_data_to_recipient(
                {'type': 'sync_new_screen_name', 'new_screen_name': self.screen_name}, friend_id)

    def request_all_user_data(self):
        """Returns all data server has on user"""
        self.send_encrypted_data_to_server({'type': 'request_all_user_data'})
        user_details = self.recieve_encrypted_data_from_server()[
            'user_details']
        return f"user_id: {user_details[0]}\nscreen_name{user_details[1]}\nseralized_public_key: {user_details[2]}"

    def get_output_path(self):
        """Returns chosen path to save user data"""
        output_path = filedialog.askdirectory(
            title="Select place to save data")
        return output_path


```

GUI OOP Approach Version 1

```
from pathlib import Path
from random import randint

# from tkinter import *
# Explicit imports to satisfy Flake8
# from tkinter import *
import tkinter as tk
from tkinter.ttk import *
from tkinter import ttk
from tkinter import messagebox
from datetime import datetime
# import ttkbootstrap as tb
# from ttkbootstrap.constants import *
import sqlite3
from sqlite3 import Error
from PIL import Image, ImageTk
import socket
import threading
import json
import ast

OUTPUT_PATH = Path(__file__).parent
ASSETS_PATH = OUTPUT_PATH / \
    Path(r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level NEA\build\assets\frame0")

def relative_to_assets(path: str) -> Path:
    return ASSETS_PATH / Path(path)

HEADER = 64  # used for send message protocol
PORT = 65432  # TCP/UDP packets
FORMAT = 'utf-8'
SERVER = "192.168.0.30"
ADDR = (SERVER, PORT)

class Client:
    def __init__(self):
        self.user_id = None
        self.public_key = None
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connection_failed = False
        self.ChatPage = None
        self.screen_name = ''
        self.friends = None

    def set_chat_page(self, ChatPage):
        self.ChatPage = ChatPage

    def start(self):
        self.connect()
        # self.send_first_contact()

        """LISTEN THREAD TO BE CALLED ACTIVE ONCE LOGIN OCCURS!
        DEACTIVATE WHEN SERVER NEEDS TO SEND OTHER DATA"""

    def start_listen_thread(self):
        thread_listen = threading.Thread(target=self.listen)
        thread_listen.start()

    def connect(self):
        try:
            self.client.connect(ADDR)
        except Exception as e:
            print(f"[ERROR] Client was unable to connect to server: {ADDR}")
            self.connection_failed = True

    def validate_login_info(self, username, password):
        print('[FUNCTION RUN] validate_login_info')
        user_data = {
            'type': 'null',
            'user_id': username,
            'password': password
        }
        self.send_data({'type': 'null', 'choice': 'login'})
        self.send_data(user_data)
        data = self.recive_data()
        return data['valid_password']
        # recieve true or false

    def recive_data(self):
        print('[WAITING] to recieve data')
        data_length = self.client.recv(HEADER).decode(FORMAT)
        if data_length != 0:
            data_length = int(data_length)
            json_data = self.client.recv(data_length).decode(FORMAT)
            data = json.loads(json_data)
            return data

    def send_data(self, data):
        print('[DEBUG] SENDING DATA')
        data = json.dumps(data)
        data = data.encode(FORMAT)
        self.client.send(self.packet_header(data))
        self.client.send(data)

    def send(self, msg):
        message = msg.encode(FORMAT)
        self.client.send(self.packet_header(message))
        self.client.send(message)

    def packet_header(self, msg):
        msg_length = len(msg)
        send_length = str(msg_length).encode(FORMAT)
        print(f'[DEBUG] PROTOCOL {msg_length} -- {send_length}')
        send_length += b' ' * (HEADER - len(send_length))
        print(f'[SEND LENGTH] {send_length}')
        return send_length

    def listen(self):
        print('[CLIENT LISTENING]')
        while True:
            data = self.recive_data()
            if data['type'] != 'message':
                print('RECIEVED FRIEND LIST')
                self.ChatPage.current_friend_message_history = data['message history']
                self.ChatPage.display_friend_messages()
            else:
```

```python
                self.ChatPage.message_recieved(data)

    def get_user_input(self):
        while True:
            message = input('Enter message: ')
            self.send(message)

class AppUI(tk.Tk):

    def __init__(self, *args, **kwargs):

        tk.Tk.__init__(self, *args, **kwargs)
        self.geometry("745x504")
        container = tk.Frame(self)
        self.screen_name = ""
        self.client = Client()

        container.pack(side="top", fill="both", expand=True)

        container.grid_rowconfigure(0, weight=1)
        container.grid_columnconfigure(0, weight=1)

        self.frames = {}

        for F in (LoginPage, ChatPage, PageTwo, CreateAccountPage, SettingsPage):

            frame = F(container, self, self.client)

            self.frames[F] = frame

            frame.grid(row=0, column=0, sticky="nsew")
        self.connect_to_server()
        self.show_frame(LoginPage)
        self.ChatPage = self.frames[ChatPage]

        self.client.set_chat_page(self.ChatPage)

    def show_frame(self, new_frame):
        frame = self.frames[new_frame]
        frame.on_show()
        frame.tkraise()

    def set_screen_name(self, data):
        self.screen_name = data

    def get_screen_name(self):
        return self.screen_name

    def connect_to_server(self):
        self.client.start()
        if self.client.connection_failed:
            print('AppUI reporting - connection error')

    def on_close(self):
        print("[CLOSING APP]")
        try:
            self.client.send_data({'type': 'DISCONNECT'})
        except:
            pass
        self.destroy()

class LoginPage(tk.Frame):
    def __init__(self, parent, window, client):
        self.window = window
        self.client = client
        tk.Frame.__init__(self, parent)
        self.create_and_place()

    def on_show(self):
        if self.client.connection_failed:
            self.canvas.itemconfig(
                self.connection_error_message, state='normal'
            )
            for w in self.winfo_children():
                # disable all widgets on login frame
                # prevents user loging in when server is down
                w.configure(state="disabled")

    def login(self):
        username_value = self.username_entry.get()
        password_value = self.password_entry.get()
        self.canvas.itemconfig(self.error_message, state='hidden')
        if username_value == '' or password_value == '':
            self.canvas.itemconfig(
                self.error_message, text='Please fill in required fields')
            self.canvas.moveto(self.error_message, 257.0, 384.0)
            self.canvas.itemconfig(self.error_message, state='normal')
        else:
            valid_password = self.client.validate_login_info(
                username_value, password_value)
            if valid_password:
                print("Password Valid")
                print("[LOGGED IN]")
                json_data = self.client.recive_data()
                self.client.screen_name = json_data['screen_name']
                # self.window.set_screen_name(self.client.screen_name)
                self.client.friends = json_data['friend_list']
                self.client.start_listen_thread()
                self.window.show_frame(ChatPage)
            else:
                print("Invalid Password")
                self.canvas.itemconfig(
                    self.error_message, text='Incorrect username or password')
                self.canvas.moveto(self.error_message, 230.0, 384.0)
                self.canvas.itemconfig(self.error_message, state='normal')

    def show_hide_password(self):
        print('[BUTTON CLICKED] toggle_password_button')
        if self.password_entry.cget('show') == '':
            self.password_entry.config(show='*')
            self.toggle_password_button.config(image=self.eye_closed_image)
        else:
            self.password_entry.config(show='')
            self.toggle_password_button.config(image=self.eye_open_image)
        try:
            if self.comfirm_password_feild.cget('show') == '':
                self.comfirm_password_feild.config(show='*')
                self.toggle_password_button.config(image=self.eye_closed_image)
            else:
                self.comfirm_password_feild.config(show='')
                self.comfirm_password_feild.config(image=self.eye_open_image)
        except:
            pass

    def create_and_place(self):
        self.canvas = tk.Canvas(
            self,
            bg="#0A0C10",
            height=504,
            width=745,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )
        self.canvas.place(x=0, y=0)

        login_header_text = self.canvas.create_text(
            316.0,
            30.0,
            anchor="nw",
            text="Login",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 40 * -1)
        )

        self.login_entry_image = tk.PhotoImage(
            file=relative_to_assets("login_entry.png"))

        # USERNAME ENTRY
        username_entry_text = self.canvas.create_text(
            252.0,
            118.0,
            anchor="nw",
            text="USERNAME",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )
        username_entry_bg = self.canvas.create_image(
            372.0,
            156.5,
            image=self.login_entry_image
        )
        self.username_entry = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0
        )
        self.username_entry.place(
            x=255.5,
            y=140.0,
            width=233.0,
            height=33.0
        )

        # PASSWORD ENTRY
        password_entry_text = self.canvas.create_text(
            252.0,
            223.0,
            anchor="nw",
            text="PASSWORD",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )
        password_entry_bg = self.canvas.create_image(
            372.0,
            261.5,
            image=self.login_entry_image
        )
        self.password_entry = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0,
            show='*'
        )
        self.password_entry.place(
            x=255.5,
            y=245.0,
            width=233.0,
            height=33.0
        )
        self.password_entry.bind(
            '<Return>', lambda e: self.login())

        # CREATE ACCOUNT BUTTON
        self.create_account_button_image = tk.PhotoImage(
            file=relative_to_assets("login_create_account_button.png"))

        create_account_button = tk.Button(
            self,
            image=self.create_account_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.window.show_frame(CreateAccountPage),
            relief="flat",
            activebackground='#0A0C10'
        )
        create_account_button.place(
            x=271.0,
            y=415.0,
            width=203.0,
            height=25.0
        )

        # LOGIN BUTTON
        # self is required because as soon as the function finishes the variable is destroyed
        self.login_button_image = tk.PhotoImage(
            file=relative_to_assets("login_login_button.png"))

        self.login_button = tk.Button(
            self,
            image=self.login_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.login(),
            relief="flat",
            activebackground='#0A0C10'
        )
        self.login_button.place(
            x=311.0,
            y=328.0,
            width=123.0,
            height=39.0
        )

        # TOGGLE PASSWORD VISABILITY BUTTON
        self.eye_open_image = tk.PhotoImage(
            file=relative_to_assets("login_eye_open.png"))
        self.eye_closed_image = tk.PhotoImage(
            file=relative_to_assets("login_eye_closed.png"))

        self.toggle_password_button = tk.Button(
            self,
            image=self.eye_closed_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.show_hide_password(),
            relief="flat",
            background='#0A0C10',
            activebackground='#0A0C10'
        )
        self.toggle_password_button.place(
            x=519.0,
            y=237.0,
            width=51.0,
            height=51.0
        )

        # ERROR MESSAGE
        self.error_message = self.canvas.create_text(
            257.0,
            384.0,
            anchor="nw",
            text='',
            fill="#FF4747",
            font=("MontserratRoman Bold", 20 * -1),
            state='hidden'
        )

        # CONNECTION ERROR MESSAGE
        self.connection_error_message = self.canvas.create_text(
            225.0,
            80.0,
            anchor="nw",
            text='Client unable to connect to server',
            fill="#FF4747",
            font=("MontserratRoman Bold", 20 * -1),
            state='hidden'
        )

class CreateAccountPage(tk.Frame):
    def __init__(self, parent, window, client):
        self.window = window
        self.client = client
        tk.Frame.__init__(self, parent)
        self.create_and_place()

    def on_show(self):
        pass

    def check_unique_username(self, username_value):
        """
        Send username value to server
        Server checks if it is unique
        Return false
        """
        self.client.send_data({'type': 'null', 'username': username_value})
        return self.client.recive_data()['unique']
        # return True

    def create_new_account(self):
        """Creates new account OR shows error messasge if feilds not filled in correctly"""
        print("[BUTTON CLICKED] submit_button")
        username_value = self.username_entry_feild.get()
        screen_name_value = self.screen_name_entry_feild.get()
        password_value = self.password_entry.get()
        password_confirm_value = self.comfirm_password_feild.get()
        self.canvas.itemconfig(self.error_message, state='hidden')
        if username_value == '' or screen_name_value == '' or password_value == '' or password_confirm_value == '':
            self.canvas.itemconfig(
                self.error_message, text='Please fill in required fields')
            self.canvas.moveto(self.error_message, 250.0, 422.0)
            self.canvas.itemconfig(self.error_message, state='normal')
        elif password_value != password_confirm_value:
            self.canvas.itemconfig(
                self.error_message, text='Your entered passwords do not match')
            self.canvas.moveto(self.error_message, 207.0, 422.0)
            self.canvas.itemconfig(self.error_message, state='normal')
        else:
            self.client.send_data({'type': 'null', 'choice': 'create account'})
            if not self.check_unique_username(username_value):
                self.canvas.itemconfig(
                    self.error_message, text='Username taken. Please try a different username')
                self.canvas.moveto(self.error_message, 157.0, 422.0)
                self.canvas.itemconfig(self.error_message, state='normal')
            else:
                print('[ACCOUNT CREATED]')
                self.client.send_data({
                    'type': 'null',
                    'screen_name': screen_name_value,
                    'password': password_value
                })
                self.client.screen_name = screen_name_value
                self.window.show_frame(ChatPage)

    def create_and_place(self):
        self.canvas = tk.Canvas(
            self,
            bg="#0A0C10",
            height=504,
            width=745,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )
        self.canvas.place(x=0, y=0)

        self.create_account_entry_image = tk.PhotoImage(
            file=relative_to_assets("create_account_entry.png"))

        # USERNAME ENTRY
        username_entry_bg = self.canvas.create_image(
            205.5,
            149.5,
            image=self.create_account_entry_image
        )
        self.username_entry_feild = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0
        )
        self.username_entry_feild.place(
            x=87.5,
            y=133.0,
            width=236.0,
            height=33.0
        )
        self.canvas.create_text(
            85.0,
            110.0,
            anchor="nw",
            text="USERNAME",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )

        # SCREEN NAME ENTRY
        screen_name_entry_bg = self.canvas.create_image(
            205.5,
            217.5,
            image=self.create_account_entry_image
        )
        self.screen_name_entry_feild = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0
        )
        self.screen_name_entry_feild.place(
            x=87.5,
            y=201.0,
            width=236.0,
            height=33.0
        )
        self.canvas.create_text(
            85.0,
            178.0,
            anchor="nw",
            text="SCREEN NAME",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )

        # PASSWORD ENTRY
        self.password_entry_image = tk.PhotoImage(
            file=relative_to_assets("create_account_password_entry.png"))

        password_entry_bg = self.canvas.create_image(
            515.5,
            149.5,
            image=self.password_entry_image
        )
        self.password_entry = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0,
            show='*'
        )
        self.password_entry.place(
            x=422.5,
            y=133.0,
            width=186.0,
            height=33.0
        )

        self.canvas.create_text(
            421.0,
            110.0,
            anchor="nw",
            text="PASSWORD",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )

        # CONFIRM PASSWORD ENTRY
        confirm_password_bg = self.canvas.create_image(
            540.5,
            217.5,
            image=self.create_account_entry_image
        )
        self.comfirm_password_feild = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0,
            show='*'
        )
        self.comfirm_password_feild.place(
            x=422.5,
            y=201.0,
            width=236.0,
            height=33.0
        )

        self.canvas.create_text(
            421.0,
            178.0,
            anchor="nw",
            text="CONFIRM PASSWORD",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )

        # CREATE ACCOUNT HEADER
        self.canvas.create_text(
            227.0,
            35.0,
            anchor="nw",
            text="Create Account",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 40 * -1)
        )

        # ERROR MESSAGE
        self.error_message = self.canvas.create_text(
            240.0,
            422.0,
            anchor="nw",
            text="",
            fill="#FF4747",
            font=("MontserratRoman Bold", 20 * -1),
            state='hidden'
        )

        # SUBMIT BUTTON
        self.submit_button_image = tk.PhotoImage(
            file=relative_to_assets("create_account_submit.png"))

        submit_button = tk.Button(
            self,
            image=self.submit_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.create_new_account(),
            relief="flat",
            activebackground='#0A0C10'
        )
        submit_button.place(
            x=293.0,
            y=266.0,
            width=149.0,
            height=43.0
        )

        # LOGIN OPTION BUTTON
        self.login_option_image = tk.PhotoImage(
            file=relative_to_assets("create_account_login_option.png"))

        login_option_button = tk.Button(
            self,
            image=self.login_option_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.window.show_frame(LoginPage),
            relief="flat",
            activebackground='#0A0C10'
        )
        login_option_button.place(
            x=227.0,
            y=342.0,
            width=286.0,
            height=48.0
        )

        # TOGGLE PASSWORD VISABILITY BUTTON
        self.eye_open_image = tk.PhotoImage(
            file=relative_to_assets("login_eye_open.png"))
        self.eye_closed_image = tk.PhotoImage(
            file=relative_to_assets("login_eye_closed.png"))
        self.toggle_password_button = tk.Button(
            self,
            image=self.eye_closed_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: LoginPage.show_hide_password(self),
            relief="flat",
            background='#0A0C10',
```

```python
            activebackground='#0A0C10'
        )
        self.toggle_password_button.place(
            x=633.0,
            y=125.0,
            width=51.0,
            height=51.0
        )

class SettingsPage(tk.Frame):
    def __init__(self, parent, window, client):
        self.window = window
        self.client = client
        tk.Frame.__init__(self, parent)
        self.parent = parent

    def on_show(self):
        self.create_and_place()
        self.canvas.itemconfig(
            self.current_screen_name_text, text=self.client.screen_name)

    def show_chats_page(self):
        # self.canvas.delete(self.header_rectangle)
        self.window.show_frame(ChatPage)

    def change_screen_name(self):
        self.canvas.itemconfig(self.error_message, state='hidden')
        if self.new_screen_name_entry.get() == self.client.screen_name:
            # if screen name is same as old screen name display error message
            self.canvas.itemconfig(self.error_message, state='normal',
                                   text='Your new screen can not be the same as your current screen name')
        elif self.new_screen_name_entry.get() == '':
            self.canvas.itemconfig(self.error_message, state='normal',
                                   text='Your new screen name can not be empty')
        else:
            self.canvas.itemconfig(
                self.current_screen_name_text, text=self.new_screen_name_entry.get())
            self.client.screen_name = self.new_screen_name_entry.get()
            self.client.send_data(
                {'type': 'update screen name', 'new screen name': self.client.screen_name})

    def popup_frame(self):
        top = tk.Toplevel(self.window)
        top.protocol("WM_DELETE_WINDOW", self.popup_frame_error)
        self.window.protocol("WM_DELETE_WINDOW", self.popup_frame_error)
        top.geometry("479x375")
        top.resizable(False, False)
        top.title("Confirm account deletion")

        popup_canvas = tk.Canvas(
            top,
            bg="#0A0C10",
            height=375,
            width=479,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )

        popup_canvas.place(x=0, y=0)
        warning_header_text = popup_canvas.create_text(
            58.0,
            14.0,
            anchor="nw",
            text="Are you sure you want to \n   delete your account?",
            fill="#FF4747",
            font=("MontserratRoman Bold", 32 * -1)
        )

        self.yes_button_image = tk.PhotoImage(
            file=relative_to_assets("confirmation_popup_yes.png"))
        yes_button = tk.Button(
            top,
            image=self.yes_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.yes_button_clicked(top),
            relief="flat",
            activebackground='#2A3441'
        )
        yes_button.place(
            x=57.0,
            y=164.0,
            width=139.0,
            height=48.0
        )

        self.no_button_image = tk.PhotoImage(
            file=relative_to_assets("confirmation_popup_no.png"))
        no_button = tk.Button(
            top,
            image=self.no_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.no_button_clicked(top),
            relief="flat",
            activebackground='#2A3441'
        )
        no_button.place(
            x=279.0,
            y=163.0,
            width=139.0,
            height=50.0
        )

    def popup_frame_error(self):
        tk.messagebox.showerror('Error', 'Please select either yes or no')

    def no_button_clicked(self, top):
        top.destroy()
        self.window.protocol("WM_DELETE_WINDOW", self.window.on_close)

    def yes_button_clicked(self, top):
        print('[DELETING ACCOUNT]')
        self.window.protocol("WM_DELETE_WINDOW", self.window.on_close)

    def create_and_place(self):
        self.canvas = tk.Canvas(
            self,
            bg="#0A0C10",
            height=504,
            width=745,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )

        self.canvas.place(x=0, y=0)
        self.header_rectangle = self.canvas.create_rectangle(
            0.0,
            0.0,
            745.0,
            43.0,
            fill="#12161C",
            outline="")

        settings_header_text = self.canvas.create_text(
            16.0,
            8.0,
            anchor="nw",
            text="Settings",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        # CURRENT SCREEN NAME DISPLAY
        current_screen_name_header_text = self.canvas.create_text(
            16.0,
            61.0,
            anchor="nw",
            text="Current screen name",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        self.option_addcurrent_screen_name_image = tk.PhotoImage(
            file=relative_to_assets("settings_current_screen_name.png"))
        current_screen_name_bg = self.canvas.create_image(
            176.5,
            112.0,
            image=self.option_addcurrent_screen_name_image
        )

        self.current_screen_name_text = self.canvas.create_text(
            37.0,
            99.0,
            anchor="nw",
            text="",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        # CHANGE SCREEN NAME DISPLAY
        change_screen_name_text = self.canvas.create_text(
            16.0,
            160.0,
            anchor="nw",
            text="Change screen name",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        self.new_screen_name_entry_image = tk.PhotoImage(
            file=relative_to_assets("settings_new_screen_name_entry.png"))
        new_screen_name_entry_bg = self.canvas.create_image(
            176.5,
            213.0,
            image=self.new_screen_name_entry_image
        )
        self.new_screen_name_entry = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#E3E7ED",
            highlightthickness=0,
            font=("MontserratRoman")
        )
        self.new_screen_name_entry.place(
            x=37.0,
            y=192.0,
            width=279.0,
            height=40.0
        )

        self.new_screen_name_entry.bind(
            '<Return>', lambda e: self.change_screen_name())

        # CONFIRM SCREEN NAME BUTTON DISPLAY
        self.confirm_screen_name_button_image = tk.PhotoImage(
            file=relative_to_assets("settings_confirm_screen_name_button.png"))
        confirm_screen_name_button = tk.Button(
            self,
            image=self.confirm_screen_name_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.change_screen_name(),
            relief="flat",
            activebackground='#0A0C10'
        )
        confirm_screen_name_button.place(
            x=359.0,
            y=193.0,
            width=151.0,
            height=42.0
        )

        # DELETE ACCOUNT BUTTON DISPLAY
        self.delete_account_button_image = tk.PhotoImage(
            file=relative_to_assets("settings_delete_account_button.png"))
        delete_account_button = tk.Button(
            self,
            image=self.delete_account_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=self.popup_frame,
            relief="flat",
            activebackground='#0A0C10'
        )
        delete_account_button.place(
            x=16.0,
            y=436.0,
            width=229.0,
            height=42.0
        )

        # REQUEST DATA BUTTON DISPLAY
        self.request_data_button_image = tk.PhotoImage(
            file=relative_to_assets("settings_request_data_button.png"))
        request_data_button = tk.Button(
            self,
            image=self.request_data_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: print("button_2 clicked"),
            relief="flat",
            activebackground='#0A0C10'
        )
        request_data_button.place(
            x=443.0,
            y=436.0,
            width=302.0,
            height=42.0
        )

        # BACK TO CHATS BUTTON DISPLAY
        self.back_to_chats_button_image = tk.PhotoImage(
            file=relative_to_assets("settings_back_to_chats_button.png"))
        back_to_chats_button = tk.Button(
            self,
            image=self.back_to_chats_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.show_chats_page(),
            relief="flat",
            activebackground='#0A0C10'
        )
        back_to_chats_button.place(
            x=640.0,
```

```python
            y=61.0,
            width=95.0,
            height=87.0
        )

        self.error_message = self.canvas.create_text(
            16.0,
            250.0,
            anchor="nw",
            text="",
            fill="#FF4747",
            font=("MontserratRoman Bold", 20 * -1),
            state='hidden'
        )

class ChatPage(tk.Frame):
    def __init__(self, parent, window, client):
        self.window = window
        self.client = client
        tk.Frame.__init__(self, parent)
        self.parent = parent
        # self.list_of_friends = ['Craig', 'Dave', 'Calum', 'Gilbert']
        # self.create_and_place()
        # self.add_temp_text()
        self.current_friend_message_history = None
        # self.last_selected = None

    def on_show(self):
        self.create_and_place()
        self.add_temp_text()
        # self.client.start_listen_thread()
        print("[FRAME SHOWN] ChatsPage")
        self.message_entry_box.config(state='disabled')
        self.canvas.itemconfig(self.user_profile_text,
                               text=self.client.screen_name)
        if len(self.client.friends) != 0:
            self.show_friends()
            self.canvas.itemconfig(
                self.recipient, text=self.client.friends[0][2])
            self.message_entry_box.config(state='normal')
            self.last_selected = tk.StringVar(
                self.friend_list_area, str(list(self.client.friends[0])))
        self.v.set(str(list(self.client.friends[0])))
        # self.friend_chat_btn_press(self.v.get())
        self.client.send_data(
            {'type': 'message history request', 'conversation_id':
self.get_radio_button_value()[0]})
        # print(f"[CHATS PAGE SHOWN] self.v set to:{self.client.friends[0]}")
        # print(f"[VALUE] self.v = {repr(self.v.get())}")

    def show_settings_page(self):
        self.window.show_frame(SettingsPage)

    def display_friend_messages(self):
        for message in self.current_friend_message_history:
            formatted_message = self.format_message(
                message[6], message[3], message[4], message[2])
            self.display_message(formatted_message)

    def friend_chat_btn_press(self, friend):
        current_button = self.v.get()
        self.message_display_box.config(state='normal')
        self.message_display_box.delete(1.0, 'end')
        if current_button != self.last_selected.get():
            self.canvas.itemconfig(self.recipient, text=friend)
            # Get chat specific messages from server
            # delete all in current window
            # paste in messages!
            self.client.send_data(
                {'type': 'message history request', 'conversation_id': self.get_radio_button_value()[0]})
            # message_history = self.client.recive_data()
            # print(self.current_friend_message_history)
            # self.message_display_box.config(state='normal')
            # self.message_display_box.delete(1.0, 'end')
        # print("CURRENT RADIO BUTTON")
        # print(f"[VALUE] self.v = {repr(self.v.get())}")
        self.last_selected.set(current_button)

    def add_new_friend_to_UI(self, friend):
        # print(f'[FRIEND RAIDOBTN] {friend}, {repr(friend)}')
        btn = tk.Radiobutton(self.friend_list_area,
                             text=friend[2],
                             variable=self.v,
                             value=str(friend),
                             indicatoron=0,
                             bg='#2A3441',
                             height=2,
                             width=23,
                             relief='flat',
                             fg='#E3E7ED',
                             selectcolor='#12161C',
                             command=lambda: self.friend_chat_btn_press(
                                 friend[2]),
                             borderwidth=0,
                             font=("MontserratRoman", 9, 'normal'),
                             )

        self.friend_list_area.window_create('end', window=btn)
        self.friend_list_area.insert('end', '\n')

    def open_add_friend_popup(self):
        pass

    def add_temp_text(self):
        self.message_entry_box.config(fg='#2a3441')
        self.message_entry_box.insert(0, 'Type your message...')

    def remove_temp_text(self):
        self.message_entry_box.config(fg='#E3E7ED')
        self.message_entry_box.delete(0, 'end')

    def get_radio_button_value(self):
        values = self.v.get()
        # split_values = values.split(' ')
        # print(f"[DEBUG] Radio button value = {values}")
        return ast.literal_eval(values)

    def message_recieved(self, data):
        print(f"[MESSAGE RECIEVED!] {data}")
        if data['screen_name'] == self.get_radio_button_value()[2]:
            # data = json.dumps(data)
            self.display_message(data)

    def format_message(self, screen_name, date, time, message):
        # sending screename is nessesasary as while it is redundant
        # due to the database on the serve it keeps a consistant format
        user_data = {
            'type': 'message',
            'screen_name': screen_name,
            'date': date,
            'time': time,
            'message': message,
            'chat_id': self.get_radio_button_value()[0]
        }
        return user_data

    def send_message(self, message):
        print(f'[SEND MESSAGE FUNCTION CALLED] {message}')
        screen_name, date, time = self.get_timestamp()
        # format message to send to server in json format
        # format message to display
        formatted_message = self.format_message(
            screen_name, date, time, message)
        self.client.send_data(formatted_message)
        self.display_message(formatted_message)
        # self.store_message(timestamp, message)
        # ALL we need to know is what conversation the user has click on.

    def get_timestamp(self):
        # screen_name = find_screen_name(conn, username)
        screen_name = self.client.screen_name
        now = datetime.now()
        date = now.strftime("%d/%m/%Y")
        time = now.strftime("%H:%M:%S")
        return screen_name, date, time

    def display_message(self, formatted_message):
        # formatted_message = json.loads(formatted_message)
        dt = f"{formatted_message['date']} {formatted_message['time']}"
        screen_name = f"{formatted_message['screen_name']}"
        timestamp = f"\n<{screen_name} {dt}>\n"
        message = f"{formatted_message['message']}\n"
        self.message_display_box.tag_add(
            'timestamp_formatting', '1.0', '1.end')
        self.message_display_box.tag_config('timestamp_formatting', font=(
            'MontserratRoman', 10, 'italic'))
        self.message_display_box.config(state='normal')
        self.message_display_box.insert(
            'end', timestamp, 'timestamp_formatting')
        self.message_display_box.insert('end', message)
        self.message_display_box.see('end')
        self.message_display_box.config(state='disabled')
        self.message_entry_box.delete(0, 'end')

    def create_and_place(self):

        self.canvas = tk.Canvas(
            self,
            bg="#0A0C10",
            height=504,
            width=745,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )

        self.canvas.place(x=0, y=0)
        self.side_bar_rectangle = self.canvas.create_rectangle(
            0.0,
            0.0,
            190.0,
            504.0,
            fill="#2A3441",
            outline="",
            tags='rectangle')

        self.canvas.create_text(
            9.0,
            14.0,
            anchor="nw",
            text="Chats",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        # self.message_display_image = tk.PhotoImage(
        #     file=relative_to_assets("message_display.png"))
        # message_display_bg = canvas.create_image(
        #     465.5,
        #     251.5,
        #     image=self.message_display_image
        # )
        self.message_display_box = tk.Text(
            self,
            bd=0,
            bg="#12161C",
            fg="#E3E7ED",
            highlightthickness=0,
            font=("MontserratRoman", 14, 'normal'),
            state='disabled'
        )
        self.message_display_box.place(
            x=199.0,
            y=54.0,
            width=533.0,
            height=393.0
        )

        self.message_entry_box_image = tk.PhotoImage(
            file=relative_to_assets("entry_2.png"))
        message_entry_box_bg = self.canvas.create_image(
            485.0,
            477.0,
            image=self.message_entry_box_image
        )

        self.message_entry_box = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#E3E7ED",
            highlightthickness=0,
            font=("MontserratRoman")
        )
        self.message_entry_box.place(
            x=292.0,
            y=463.0,
            width=386.0,
            height=28.0
        )

        self.message_entry_box.bind(
            '<FocusIn>', lambda e: self.remove_temp_text())
        self.message_entry_box.bind(
            '<FocusOut>', lambda e: self.add_temp_text())
        self.message_entry_box.bind(
            '<Return>', lambda e: self.send_message(self.message_entry_box.get()))

        # -------------------------

        self.recipient = self.canvas.create_text(
            199.0,
            14.0,
            anchor="nw",
            text="Add a friend to start chatting",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        self.user_profile_text = self.canvas.create_text(
            10.0,
            466.0,
            anchor="nw",
            text='',
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        self.settings_image = tk.PhotoImage(
            file=relative_to_assets("settings_button.png"))

        settings_button = tk.Button(
            self,
            image=self.settings_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.show_settings_page(),
            relief="flat",
            bg='#2A3441',
            activebackground='#2A3441'
        )
        settings_button.place(
            x=151.0,
            y=463.0,
            width=30.0,
            height=30.0
        )
        self.add_friend_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_button.png"))
        add_friend_button = tk.Button(
            self,
            image=self.add_friend_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.open_add_friend_popup(),
            relief="flat",
            bg='#2A3441',
            activebackground='#2A3441'
        )
        add_friend_button.place(
            x=151.0,
            y=15.0,
            width=30.0,
            height=30.0
        )

        self.send_message_image = tk.PhotoImage(
            file=relative_to_assets("send_message_button.png"))
        send_message_button = tk.Button(
            self,
            image=self.send_message_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.send_message(self.message_entry_box.get()),
            relief="flat",
            bg='#0A0C10',
            activebackground='#0A0C10'
        )
        send_message_button.place(
            x=702.0,
            y=463.0,
            width=30.0,
            height=30.0
        )

        self.attachment_button_image = tk.PhotoImage(
            file=relative_to_assets("attachment_button.png"))
        attachment_button = tk.Button(
            self,
            image=self.attachment_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: print("[BUTTON CLICKED] attachment_button"),
            relief="flat",
            bg='#0A0C10',
            activebackground='#0A0C10'
        )
        attachment_button.place(
            x=199.0,
            y=463.0,
            width=30.0,
            height=30.0
        )

        self.emoji_keyboard_image = tk.PhotoImage(
            file=relative_to_assets("emoji_button.png"))

        emoji_keyboard_button = tk.Button(
            self,
            image=self.emoji_keyboard_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: print("[BUTTON CLICKED] emoji_keyboard_button"),
            relief="flat",
            bg='#0A0C10',
            activebackground='#0A0C10'
        )
        emoji_keyboard_button.place(
            x=238.0,
            y=463.0,
            width=30.0,
            height=30.0
        )

# FRIENDS LIST CODE SHTUFFSS

        self.friend_list_area = tk.Text(
            self,
            bd=0,
            bg="#12161C",
            fg="#E3E7ED",
            highlightthickness=0,
            cursor='arrow'
        )
        self.friend_list_area.place(
            x=0.0,
            y=93.0,
            width=190,
            height=356
        )
        sb = tk.Scrollbar(self.friend_list_area,
                          command=self.friend_list_area.yview, width=20)
        sb.pack(side='right', fill=tk.Y)
        self.friend_list_area.configure(yscrollcommand=sb.set)
        self.v = tk.StringVar(self.friend_list_area)

    def show_friends(self):
        for friend in self.client.friends:
            print(f"[SHOW FRIEND DEBUG] {friend}")
            self.add_new_friend_to_UI(friend)

class PageTwo(tk.Frame):

    def __init__(self, parent, controller, client):
        tk.Frame.__init__(self, parent)
        label = tk.Label(self, text="Page Two!!!")
        label.pack(pady=10, padx=10)

        button1 = tk.Button(self, text="Back to Home",
                            command=lambda: controller.show_frame(ChatPage))
        button1.pack()

        button2 = tk.Button(self, text="Page One",
                            command=lambda: controller.show_frame(LoginPage))
        button2.pack()

app = AppUI()
app.resizable(False, False)
app.protocol("WM_DELETE_WINDOW", app.on_close)
app.mainloop()


```

GUI OOP Approach Version 2

```
# GUI Related imports
import tkinter as tk
from tkinter import filedialog
import tkinter.messagebox as tk1
from PIL import Image, ImageTk
from datetime import datetime
from emojiKeyboardPackage import emojiKeyboard
import re

# File manipulation imports
from pathlib import Path
import os

# Networking Imports
import neat_secure_client as secure_client


# Asset management
OUTPUT_PATH = Path(__file__).parent
ASSETS_PATH = OUTPUT_PATH / \
    Path(r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level NEA\build\assets\frame0")


def relative_to_assets(path: str) -> Path:
    return ASSETS_PATH / Path(path)


class UIController(tk.Tk):

    def __init__(self, *args, **kwargs):

        tk.Tk.__init__(self, *args, **kwargs)

        # Creating main app
        self.geometry("745x504")
        app_frame = tk.Frame(self)
        app_frame.pack(side="top", fill="both", expand=True)
        app_frame.grid_rowconfigure(0, weight=1)
        app_frame.grid_columnconfigure(0, weight=1)

        self.client = secure_client.Client()

        self.screen_name = None

        # Creating the different pages/frames
        self.frames = {}

        for F in (LoginPage, CreateAccountPage, ChatPage, AddFriendPage, SettingsPage):

            frame = F(self, app_frame, self.client)

            self.frames[F] = frame

            frame.grid(row=0, column=0, sticky="nsew")

        # Connect to server and display first frame
        if self.client.connect():
            self.client.establish_inital_contact()
        self.show_frame(LoginPage)

        self.ChatPage = self.frames[ChatPage]
        self.AddFriendPage = self.frames[AddFriendPage]

        self.client.ChatPage = self.ChatPage
        self.client.AddFriendPage = self.AddFriendPage

    def show_frame(self, new_frame: object):
        """Raises the frame in the paremeter: new_frame"""

        frame = self.frames[new_frame]
        # on_show acts like the __init__ class but for when a frame is shown instead
        frame.on_show()
        frame.tkraise()

    def on_close(self, account_deletion=False):
        """Function called when the user closes the entire app"""
        try:
            self.client.stop_listen()
            self.client.send_disconnect_message()
            print('[-] CLOSING APP...')
        except Exception as e:
            print(e)
        self.destroy()
        self.client.close_db_connection()
        if account_deletion:
            self.client.delete_directory()
        self.client.close_client()


class LoginPage(tk.Frame):
    def __init__(self, parent, window, client: secure_client.Client):
        self.controller = parent
        self.client = client
        tk.Frame.__init__(self, window)

    def on_show(self):
        """Function called when frame is shown"""
        self.create_and_place()
        self.check_client_connection()

    def create_and_place(self):
        """Creates and places all tkinter objects onto the frame"""

        self.canvas = tk.Canvas(
            self,
            bg="#0A0C10",
            height=504,
            width=745,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )
        self.canvas.place(x=0, y=0)

        login_header_text = self.canvas.create_text(
            316.0,
            30.0,
            anchor="nw",
            text="Login",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 40 * -1)
        )

        self.login_entry_image = tk.PhotoImage(
            file=relative_to_assets("login_entry.png"))

        # USERNAME ENTRY
        username_entry_text = self.canvas.create_text(
            252.0,
            118.0,
            anchor="nw",
            text="USERNAME",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )
        username_entry_bg = self.canvas.create_image(
            372.0,
            156.5,
            image=self.login_entry_image
        )
        self.username_entry = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0
        )
        self.username_entry.place(
            x=255.5,
            y=140.0,
            width=233.0,
            height=33.0
        )

        # PASSWORD ENTRY
        password_entry_text = self.canvas.create_text(
            252.0,
            223.0,
            anchor="nw",
            text="PASSWORD",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )
        password_entry_bg = self.canvas.create_image(
            372.0,
            261.5,
            image=self.login_entry_image
        )
        self.password_entry = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0,
            show='*'
        )
        self.password_entry.place(
            x=255.5,
            y=245.0,
            width=233.0,
            height=33.0
        )
        self.password_entry.bind(
            '<Return>', lambda e: self.login())

        # CREATE ACCOUNT BUTTON
        self.create_account_button_image = tk.PhotoImage(
            file=relative_to_assets("login_create_account_button.png"))

        create_account_button = tk.Button(
            self,
            image=self.create_account_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.controller.show_frame(CreateAccountPage),
            relief="flat",
            activebackground='#0A0C10'
        )
        create_account_button.place(
            x=271.0,
            y=415.0,
            width=203.0,
            height=25.0
        )

        # LOGIN BUTTON
        # self is required because as soon as the function finishes the variable is destroyed
        self.login_button_image = tk.PhotoImage(
            file=relative_to_assets("login_login_button.png"))

        self.login_button = tk.Button(
            self,
            image=self.login_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.login(),
            relief="flat",
            activebackground='#0A0C10'
        )
        self.login_button.place(
            x=311.0,
            y=328.0,
            width=123.0,
            height=39.0
        )

        # TOGGLE PASSWORD VISABILITY BUTTON
        self.eye_open_image = tk.PhotoImage(
            file=relative_to_assets("login_eye_open.png"))
        self.eye_closed_image = tk.PhotoImage(
            file=relative_to_assets("login_eye_closed.png"))

        self.toggle_password_button = tk.Button(
            self,
            image=self.eye_closed_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.show_hide_password(),
            relief="flat",
            background='#0A0C10',
            activebackground='#0A0C10'
        )
        self.toggle_password_button.place(
            x=519.0,
            y=237.0,
            width=51.0,
            height=51.0
        )

        # ERROR MESSAGE
        self.error_message = self.canvas.create_text(
            257.0,
            384.0,
            anchor="nw",
            text='',
            fill="#FF4747",
            font=("MontserratRoman Bold", 20 * -1),
            state='hidden'
        )

        # CONNECTION ERROR MESSAGE
        self.connection_error_message = self.canvas.create_text(
            225.0,
            80.0,
            anchor="nw",
            text='Client unable to connect to server',
            fill="#FF4747",
            font=("MontserratRoman Bold", 20 * -1),
            state='hidden'
        )

    def check_client_connection(self):
        """
        Checks the connection status of the client.

        If client is NOT connected it disables all UI reponsiveness
        """
        if self.client.connected == False:
            self.canvas.itemconfig(
                self.connection_error_message, state='normal'
            )
            for w in self.winfo_children():
                # disable all widgets on login frame
                # prevents user loging in when server is down
                w.configure(state="disabled")

    def login(self):
        username_value = self.username_entry.get()
        password_value = self.password_entry.get()
        self.canvas.itemconfig(self.error_message, state='hidden')
        if username_value == '' or password_value == '':
            self.canvas.itemconfig(
                self.error_message, text='Please fill in required fields')
            self.canvas.moveto(self.error_message, 257.0, 384.0)
            self.canvas.itemconfig(self.error_message, state='normal')
        else:
            valid_password = self.client.login(username_value, password_value)
            if valid_password:
                self.password_entry.unbind('<Return>')
                self.client.handel_logged_in_client()
                self.controller.show_frame(ChatPage)
            else:
                print("Invalid Password")
                self.canvas.itemconfig(
                    self.error_message, text='Incorrect username or password')
                self.canvas.moveto(self.error_message, 230.0, 384.0)
                self.canvas.itemconfig(self.error_message, state='normal')

    def show_hide_password(self):
        print('{BUTTON CLICKED} toggle_password_button')
        if self.password_entry.cget('show') == '':
            self.password_entry.config(show='*')
            self.toggle_password_button.config(image=self.eye_closed_image)
        else:
            self.password_entry.config(show='')
            self.toggle_password_button.config(image=self.eye_open_image)

        # Try block as function used by CreateAccountPage too
        try:
            if self.comfirm_password_feild.cget('show') == '':
                self.comfirm_password_feild.config(show='*')
                self.toggle_password_button.config(image=self.eye_closed_image)
            else:
                self.comfirm_password_feild.config(show='')
                self.comfirm_password_feild.config(image=self.eye_open_image)
        except:
            pass


class CreateAccountPage(tk.Frame):
    def __init__(self, parent, window, client: secure_client.Client):
        self.controller = parent
        self.client = client
        tk.Frame.__init__(self, window)

    def on_show(self):
        # self.client.get_screen_name()
        self.create_and_place()

    def create_and_place(self):
        self.canvas = tk.Canvas(
            self,
            bg="#0A0C10",
            height=504,
            width=745,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )
        self.canvas.place(x=0, y=0)

        self.create_account_entry_image = tk.PhotoImage(
            file=relative_to_assets("create_account_entry.png"))

        # USERNAME ENTRY
        username_entry_bg = self.canvas.create_image(
            205.5,
            149.5,
            image=self.create_account_entry_image
        )
        self.username_entry_feild = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0
        )
        self.username_entry_feild.place(
            x=87.5,
            y=133.0,
            width=236.0,
            height=33.0
        )
        self.canvas.create_text(
            85.0,
            110.0,
            anchor="nw",
            text="USERNAME",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )

        # SCREEN NAME ENTRY
        screen_name_entry_bg = self.canvas.create_image(
            205.5,
            217.5,
            image=self.create_account_entry_image
        )
        self.screen_name_entry_feild = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0
        )
        self.screen_name_entry_feild.place(
            x=87.5,
            y=201.0,
            width=236.0,
            height=33.0
        )
        self.canvas.create_text(
            85.0,
            178.0,
            anchor="nw",
            text="SCREEN NAME",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )

        # PASSWORD ENTRY
        self.password_entry_image = tk.PhotoImage(
            file=relative_to_assets("create_account_password_entry.png"))

        password_entry_bg = self.canvas.create_image(
            515.5,
            149.5,
            image=self.password_entry_image
        )
        self.password_entry = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0,
            show='*'
        )
        self.password_entry.place(
            x=422.5,
            y=133.0,
            width=186.0,
            height=33.0
        )

        self.canvas.create_text(
            421.0,
            110.0,
            anchor="nw",
            text="PASSWORD",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )

        # CONFIRM PASSWORD ENTRY
        confirm_password_bg = self.canvas.create_image(
            540.5,
            217.5,
            image=self.create_account_entry_image
        )
        self.comfirm_password_feild = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#000716",
            highlightthickness=0,
            show='*'
        )
        self.comfirm_password_feild.place(
            x=422.5,
            y=201.0,
            width=236.0,
            height=33.0
        )

        self.canvas.create_text(
            421.0,
            178.0,
            anchor="nw",
            text="CONFIRM PASSWORD",
            fill="#E3E7ED",
            font=("MontserratRoman Regular", 16 * -1)
        )

        # CREATE ACCOUNT HEADER
        self.canvas.create_text(
            227.0,
            35.0,
            anchor="nw",
            text="Create Account",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 40 * -1)
        )

        # ERROR MESSAGE
        self.error_message = self.canvas.create_text(
            240.0,
            422.0,
            anchor="nw",
            text="",
            fill="#FF4747",
            font=("MontserratRoman Bold", 20 * -1),
            state='hidden'
        )

        # SUBMIT BUTTON
        self.submit_button_image = tk.PhotoImage(
            file=relative_to_assets("create_account_submit.png"))

        submit_button = tk.Button(
            self,
            image=self.submit_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.create_new_account(),
            relief="flat",
            activebackground='#0A0C10'
        )
        submit_button.place(
            x=293.0,
            y=266.0,
            width=149.0,
            height=43.0
        )

        # LOGIN OPTION BUTTON
        self.login_option_image = tk.PhotoImage(
            file=relative_to_assets("create_account_login_option.png"))

        login_option_button = tk.Button(
            self,
            image=self.login_option_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.controller.show_frame(LoginPage),
            relief="flat",
            activebackground='#0A0C10'
        )
        login_option_button.place(
            x=227.0,
            y=342.0,
            width=286.0,
            height=48.0
        )

        # TOGGLE PASSWORD VISABILITY BUTTON
        self.eye_open_image = tk.PhotoImage(
            file=relative_to_assets("login_eye_open.png"))
        self.eye_closed_image = tk.PhotoImage(
            file=relative_to_assets("login_eye_closed.png"))
        self.toggle_password_button = tk.Button(
            self,
            image=self.eye_closed_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: LoginPage.show_hide_password(self),
            relief="flat",
            background='#0A0C10',
            activebackground='#0A0C10'
        )
```

```python
        self.toggle_password_button.place(
            x=633.0,
            y=125.0,
            width=51.0,
            height=51.0
        )

    def secure_password(self, password: str):
        valid = False
        special_char = re.compile('[@_!$%^&*()<>?/\|}{~:]#')
        general_patern = re.compile(
            '^(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9]).{8,64}$')

        if special_char.search(password) == None and re.match(general_patern, password):
            valid = True
        return valid

    def create_new_account(self):
        """Creates new account OR shows error messasge if feilds not filled in correctly"""
        print("{BUTTON CLICKED} submit_button")
        self.canvas.itemconfig(self.error_message, state='hidden')

        # getting relevant values
        username_value = self.username_entry_feild.get()
        screen_name_value = self.screen_name_entry_feild.get()
        password_value = self.password_entry.get()
        password_confirm_value = self.comfirm_password_feild.get()

        if username_value == '' or screen_name_value == '' or password_value == '' or password_confirm_value == '':
            self.canvas.itemconfig(
                self.error_message, text='Please fill in required fields')
            self.canvas.moveto(self.error_message, 250.0, 422.0)
            self.canvas.itemconfig(self.error_message, state='normal')
        elif password_value != password_confirm_value:
            self.canvas.itemconfig(
                self.error_message, text='Your entered passwords do not match')
            self.canvas.moveto(self.error_message, 207.0, 422.0)
            self.canvas.itemconfig(self.error_message, state='normal')
        elif not self.secure_password(password_value):
            self.canvas.itemconfig(
                self.error_message, text='Password is not secure enough')
            self.canvas.moveto(self.error_message, 240.0, 422.0)
            self.canvas.itemconfig(self.error_message, state='normal')

        else:  # actually send data to client
            unique_userID, account_created = self.client.create_account(
                username_value, password_value, screen_name_value)
            if unique_userID and account_created:
                self.client.handel_logged_in_client()
                self.controller.show_frame(ChatPage)
                print('{ACCOUNT CREATED}')
            elif unique_userID != True:
                self.canvas.itemconfig(
                    self.error_message, text='Username taken. Please try a different username'
                )
                self.canvas.moveto(self.error_message, 157.0, 422.0)
                self.canvas.itemconfig(self.error_message, state='normal')
            else:
                self.canvas.itemconfig(
                    self.error_message, text='Something went wrong when creating your account, please try again later'
                )
                self.canvas.moveto(self.error_message, 157.0, 422.0)
                self.canvas.itemconfig(self.error_message, state='normal')


class ChatPage(tk.Frame):
    def __init__(self, parent, window, client: secure_client.Client):
        self.controller = parent
        self.client = client
        tk.Frame.__init__(self, window)
        self.current_friend_message_history = None
        self.active_chat_user_details = None
        self.last_selected = None
        self.list_of_buttons = []
        self.friend_screen_name = None
        self.send_image_data = None
        self.images = []

    def on_show(self):
        self.client.get_friend_list()
        self.create_and_place()
        self.add_temp_text()
        self.show_friends()
        self.client.listen()
        self.disable_message_buttons()

    def add_temp_text(self):
        self.add_temp_text_to_message_entry()
        if len(self.client.friend_list) == 0:
            self.canvas.itemconfig(
                self.recipient, text='Add a friend to start chatting')
        else:
            self.canvas.itemconfig(
                self.recipient, text='Click on a friend to start chatting')

    def create_and_place(self):

        self.canvas = tk.Canvas(
            self,
            bg="#0A0C10",
            height=504,
            width=745,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )

        self.canvas.place(x=0, y=0)
        self.side_bar_rectangle = self.canvas.create_rectangle(
            0.0,
            0.0,
            190.0,
            504.0,
            fill="#2A3441",
            outline="",
            tags='rectangle')

        self.canvas.create_text(
            9.0,
            14.0,
            anchor="nw",
            text="Chats",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        self.message_display_box = tk.Text(
            self,
            bd=0,
            bg="#12161C",
            fg="#E3E7ED",
            highlightthickness=0,
            font=("MontserratRoman", 14, 'normal'),
            state='disabled'
        )
        self.message_display_box.place(
            x=199.0,
            y=54.0,
            width=533.0,
            height=393.0
        )

        self.message_entry_box_image = tk.PhotoImage(
            file=relative_to_assets("entry_2.png"))
        message_entry_box_bg = self.canvas.create_image(
            485.0,
            477.0,
            image=self.message_entry_box_image
        )

        self.message_entry_box = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#E3E7ED",
            highlightthickness=0,
            font=("MontserratRoman")
        )
        self.message_entry_box.place(
            x=292.0,
            y=463.0,
            width=386.0,
            height=28.0
        )

        self.message_entry_box.bind(
            '<FocusIn>', lambda e: self.remove_temp_text_from_message_entry())
        self.message_entry_box.bind(
            '<FocusOut>', lambda e: self.add_temp_text_to_message_entry())

        self.message_entry_box.bind(
            '<Return>', lambda e: self.send_message(self.message_entry_box.get()))

        # -------------------------

        self.recipient = self.canvas.create_text(
            199.0,
            14.0,
            anchor="nw",
            text="",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        self.screen_name_text = self.canvas.create_text(
            10.0,
            466.0,
            anchor="nw",
            text=self.client.screen_name,
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        self.settings_image = tk.PhotoImage(
            file=relative_to_assets("settings_button.png"))

        settings_button = tk.Button(
            self,
            image=self.settings_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.controller.show_frame(SettingsPage),
            relief="flat",
            bg='#2A3441',
            activebackground='#2A3441'
        )
```

```
        settings_button.place(
            x=151.0,
            y=463.0,
            width=30.0,
            height=30.0
        )
        self.add_friend_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_button.png"))
        add_friend_button = tk.Button(
            self,
            image=self.add_friend_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.controller.show_frame(AddFriendPage),
            relief="flat",
            bg='#2A3441',
            activebackground='#2A3441'
        )
        add_friend_button.place(
            x=151.0,
            y=15.0,
            width=30.0,
            height=30.0
        )

        self.send_message_image = tk.PhotoImage(
            file=relative_to_assets("send_message_button.png"))
        self.send_message_button = tk.Button(
            self,
            image=self.send_message_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.send_message(
                self.message_entry_box.get()),
            relief="flat",
            bg='#0A0C10',
            activebackground='#0A0C10'
        )
        self.send_message_button.place(
            x=702.0,
            y=463.0,
            width=30.0,
            height=30.0
        )

        self.attachment_button_image = tk.PhotoImage(
            file=relative_to_assets("attachment_button.png"))
        self.attachment_button = tk.Button(
            self,
            image=self.attachment_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=self.handel_getting_image,
            relief="flat",
            bg='#0A0C10',
            activebackground='#0A0C10'
        )
        self.attachment_button.place(
            x=199.0,
            y=463.0,
            width=30.0,
            height=30.0
        )

        self.emoji_keyboard_image = tk.PhotoImage(
            file=relative_to_assets("emoji_button.png"))

        self.emoji_keyboard_button = tk.Button(
            self,
            image=self.emoji_keyboard_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.open_emoji_keyboard(),
            relief="flat",
            bg='#0A0C10',
            activebackground='#0A0C10'
        )
        self.emoji_keyboard_button.place(
            x=238.0,
            y=463.0,
            width=30.0,
            height=30.0
        )


# FRIENDS LIST CODE

        self.friend_list_area = tk.Text(
            self,
            bd=0,
            bg="#12161C",
            fg="#E3E7ED",
            highlightthickness=0,
            cursor='arrow',
            state='disabled'
        )
        self.friend_list_area.place(
            x=0.0,
            y=93.0,
            width=190,
            height=356
        )
        sb = tk.Scrollbar(self.friend_list_area,
                          command=self.friend_list_area.yview, width=20)
        sb.pack(side='right', fill=tk.Y)
        self.friend_list_area.configure(yscrollcommand=sb.set)
        self.active_chat_user_details = tk.StringVar(self.friend_list_area)

    def disable_message_buttons(self):
        self.message_entry_box.config(state='disabled')
        self.send_message_button.config(state='disabled')
        self.emoji_keyboard_button.config(state='disabled')
        self.attachment_button.config(state='disabled')


# ----------EMOJI KEYBOARD LOGIC----------

    def open_emoji_keyboard(self):
        print('Opening keyboard')
        # self.newWindow = tk.Toplevel(self.controller)
        emoji_keyboard = emojiKeyboard.Keyboard(self)

    def insert_emoji(self, emojis: str):
        self.message_entry_box.focus_set()
        self.remove_temp_text_from_message_entry()
        self.message_entry_box.insert('end', emojis)

# ----------SENDING IMAGE LOGIC----------

    def handel_getting_image(self):
        self.client.client_get_image_path()
        if self.client.image_path:
            self.send_image_data = self.client.get_image_data(
                self.client.image_path)
            image_name = self.client.image_path.split('/')[-1]
            self.remove_temp_text_from_message_entry()
            self.add_specific_temp_text_to_msg_entery(image_name)

            self.message_entry_box.config(state='disabled')

```

```python

# ----------FRIEND LIST LOGIC---------------

    def add_new_friend_to_UI(self, friend_screen_name: str, friend_details: str, status: str, specifier_id: str):

        state = 'normal'
        colour = '#2A3441'
        active = '#12161C'
        if status == 'blk' and specifier_id == self.client.user_id:  # you have blocked them
            colour = '#FF4747'
            active = '#612a2a'
        elif (status == 'blk' and specifier_id != self.client.user_id) or ('deleted account' in friend_screen_name):  # they have blocked you
            state = 'disabled'

        # when clicked self.active_chat_user_details is set to value
        # text is the text shown on the button value itself
        btn = tk.Radiobutton(self.friend_list_area,
                             text=friend_screen_name,
                             variable=self.active_chat_user_details,
                             value=friend_details,
                             indicatoron=0,
                             bg=colour,
                             height=2,
                             width=23,
                             relief='flat',
                             fg='#E3E7ED',
                             selectcolor=active,
                             command=lambda: self.friend_chat_btn_press(btn),
                             borderwidth=0,
                             font=("MontserratRoman", 9, 'normal'),
                             state=state
                             )
        self.list_of_buttons.append(btn)
        self.friend_list_area.window_create('end', window=btn)
        self.friend_list_area.insert('end', '\n')

    def friend_chat_btn_press(self, btn: tk.Radiobutton):
        if self.active_chat_user_details.get() != self.last_selected:
            # getting details
            friend_details = self.active_chat_user_details.get().split(' ')
            friend_user_id = friend_details[0]
            friend_public_key = friend_details[1]
            self.friend_screen_name = btn.cget("text")

            # updating UI
            self.canvas.itemconfig(
                self.recipient, text=self.friend_screen_name)
            self.last_selected = self.active_chat_user_details.get().split(' ')

            self.message_display_box.config(state='normal')
            self.message_display_box.delete(1.0, 'end')

            # insert message history here
            self.client.decrypt_message_history(
                self.active_chat_user_details.get().split(' ')[0])
            self.show_message_history()

            if btn.cget('bg') == '#FF4747':
                self.message_entry_box.config(state='disabled')
                self.send_message_button.config(state='disabled')
                self.emoji_keyboard_button.config(state='disabled')
                self.attachment_button.config(state='disabled')
            else:
                self.message_entry_box.config(state='normal')
                self.send_message_button.config(state='normal')
                self.emoji_keyboard_button.config(state='normal')
                self.attachment_button.config(state='normal')
                self.message_entry_box.focus_set()
                self.message_entry_box.delete(0, 'end')

    def show_friends(self):
        for friend in self.client.friend_list:
            friend_screen_name = friend[2]
            friend_details = friend[0:2]
            status = friend[3]
            specifier_id = friend[4]
            self.add_new_friend_to_UI(
                friend_screen_name, friend_details, status, specifier_id)

    def update_friend_list(self):
        for btn in self.list_of_buttons:
            btn.destroy()
        self.show_friends()


# ---------SEND MESSAGE LOGIC----------------------


    def show_message_history(self):
        for message_data in self.client.current_message_history:
            formatted_message = self.format_stored_message_for_display(
                message_data)
            self.client.image_path = formatted_message['message']
            self.display_message(
                self.format_stored_message_for_display(message_data))

    def handel_sending_message_or_image(self, message: str):
        pass

    def send_message(self, message: str):
        """Executes procedure to display message and send message to the server"""
        if len(message) != 0:
            is_image = 0
            msg = message
            image_name_and_format = ''

            if self.message_entry_box.cget('state') == 'disabled':
                print('[-] SENDING IMAGE')
                is_image = 1
                msg = self.send_image_data
                image_name_and_format = self.client.image_name_and_format

            date, time = self.get_timestamp()
            formatted_message = self.format_message(
                self.client.screen_name, date, time, msg, is_image, image_name_and_format)
            self.client.handel_send_message(formatted_message)
            self.display_message(formatted_message)
            if is_image:
                os.remove(self.client.image_path)

    def get_timestamp(self):
        """Gets current data and time"""
        now = datetime.now()
        date = now.strftime("%d/%m/%Y")
        time = now.strftime("%H:%M:%S")
        return date, time

    def format_stored_message_for_display(self, message_data):
        decrypted_message = message_data[0]
        date = message_data[1]
        time = message_data[2]
        from_me = message_data[3]
        is_image = message_data[4]

        if from_me:
            screen_name = self.client.screen_name
        else:
```

```python
            screen_name = self.friend_screen_name

        return self.format_message(screen_name, date, time, decrypted_message, is_image)

    def format_message(self, screen_name, date, time, message, is_image, image_name_and_format=''):
        """Turns message into dict"""
        user_data = {
            'type': 'message',
            'recipient': self.active_chat_user_details.get().split(' ')[0],
            'config': 'message',
            'sender_user_id': self.client.user_id,
            'sender_screen_name': screen_name,
            'date': date,
            'time': time,
            'message': message,
            'is_image': is_image,
            'image_name_and_format': image_name_and_format
        }
        return user_data

    def display_message(self, formatted_message: dict):
        """Displays message with appropriate formatting onto screen"""

        self.message_display_box.tag_add(
            'timestamp_formatting', '1.0', '1.end')
        self.message_display_box.tag_config('timestamp_formatting', font=(
            'MontserratRoman', 10, 'italic'))

        # formatting text
        if formatted_message['is_image'] == False:
            self.display_message_text(formatted_message)
        else:
            self.display_message_image(formatted_message)
        self.message_entry_box.delete(0, 'end')

    def display_message_image(self, formatted_message: dict):
        dt = f"{formatted_message['date']} {formatted_message['time']}"
        screen_name = f"{formatted_message['sender_screen_name']}"
        timestamp = f"\n<{screen_name} {dt}>\n"
        image_path = self.client.image_path

        # global image
        unprocessed_image = Image.open(image_path)

        image = ImageTk.PhotoImage(unprocessed_image)
        self.images.append(image)

        self.message_display_box.config(state='normal')
        self.message_display_box.insert(
            'end', timestamp, 'timestamp_formatting')
        self.message_display_box.image_create(tk.END, image=image)
        self.message_display_box.insert('end', '\n')
        self.message_display_box.see('end')
        self.message_display_box.config(state='disabled')
        self.message_entry_box.config(state='normal')

    def display_message_text(self, formatted_message: dict):
        dt = f"{formatted_message['date']} {formatted_message['time']}"
        screen_name = f"{formatted_message['sender_screen_name']}"
        timestamp = f"\n<{screen_name} {dt}>\n"
        message = f"{formatted_message['message']}\n"

        # displaying message
        self.message_display_box.config(state='normal')
        self.message_display_box.insert(
            'end', timestamp, 'timestamp_formatting')
        self.message_display_box.insert('end', message)
        self.message_display_box.see('end')
        self.message_display_box.config(state='disabled')

    def add_temp_text_to_message_entry(self):
        if len(self.message_entry_box.get()) == 0:
            self.message_entry_box.delete(0, 'end')
            self.message_entry_box.config(fg='#2a3441')
            self.message_entry_box.insert(0, 'Type your message...')

    def remove_temp_text_from_message_entry(self):
        self.message_entry_box.config(fg='#E3E7ED')
        if self.message_entry_box.get() == 'Type your message...':
            print('REMOVING TEMP TEXT')
            self.message_entry_box.config(fg='#E3E7ED')
            self.message_entry_box.delete(0, 'end')

    def add_specific_temp_text_to_msg_entery(self, text: str):
        self.message_entry_box.config(fg='#2a3441')
        self.message_entry_box.insert(0, text)


class AddFriendPage(tk.Frame):
    def __init__(self, parent, window, client: secure_client.Client):
        self.controller = parent
        self.client = client
        tk.Frame.__init__(self, window)

        self.active_friend_button = None
        self.active_request_button = None

    def on_show(self):
        """Function called when frame is shown"""
        self.client.stop_listen()
        self.client.get_friend_list()
        self.client.get_friend_request_list()
        self.client.get_pending_friends_list()
        self.create_and_place()
        self.add_text()
        self.add_radio_buttons()

    def create_and_place(self):
        """Creates and places all tkinter objects onto the frame"""
        self.canvas = tk.Canvas(
            self,
            bg="#0A0C10",
            height=504,
            width=745,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )

        self.canvas.place(x=0, y=0)
        self.canvas.create_rectangle(
            0.0,
            0.0,
            745.0,
            32.0,
            fill="#12161C",
            outline="")

        page_header = self.canvas.create_text(
            8.0,
            6.0,
            anchor="nw",
            text="Friends",
            fill="#FFFFFF",
            font=("MontserratRoman Bold", 16 * -1)
        )

        self.back_to_chats_button_image = tk.PhotoImage(
            file=relative_to_assets("back_to_chats_button.png"))
        back_to_chats_button = tk.Button(
            self,
            image=self.back_to_chats_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.controller.show_frame(ChatPage),
            relief="flat"
        )
        back_to_chats_button.place(
            x=640.0,
            y=61.0,
            width=95.0,
            height=87.0
        )

        friends_friend_code_entry_text = self.canvas.create_text(
            8.0,
            121.0,
            anchor="nw",
            text="Enter your friend’s code to invite them to chat",
            fill="#FFFFFF",
            font=("MontserratRoman Bold", 16 * -1)
        )

        personal_friend_code_entry_text = self.canvas.create_text(
            9.0,
            38.0,
            anchor="nw",
            text="Your Friend Code",
            fill="#FFFFFF",
            font=("MontserratRoman Bold", 16 * -1)
        )

        self.friend_code_error_message = self.canvas.create_text(
            8.0,
            208.0,
            anchor="nw",
            text="",
            fill="#FF4747",
            font=("MontserratRoman Bold", 13 * -1),
            state='hidden'
        )

        self.personal_friend_code_entry_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_personal_friend_code_entry.png"))
        personal_friend_code_entry_bg = self.canvas.create_image(
            169.5,
            84.0,
            image=self.personal_friend_code_entry_image
        )

        self.friends_friend_code_entry_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_friends_friend_code_entry.png"))
        entry_bg_2 = self.canvas.create_image(
            169.5,
            171.0,
            image=self.friends_friend_code_entry_image
        )
        self.friends_friend_code_entry = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#E3E7ED",
            highlightthickness=0
        )
        self.friends_friend_code_entry.place(
            x=30.0,
            y=150.0,
            width=279.0,
            height=40.0
        )

        self.copy_button_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_copy_button.png"))
        copy_button = tk.Button(
            self,
            image=self.copy_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.copy_user_id_to_clipboard(),
            relief="flat"
        )
        copy_button.place(
            x=352.0,
            y=63.0,
            width=105.0,
            height=42.0
        )

        self.add_friend_submit_button_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_submit_button.png"))
        add_friend_submit_button = tk.Button(
            self,
            image=self.add_friend_submit_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.send_friend_request(),
            relief="flat"
        )
        add_friend_submit_button.place(
            x=352.0,
            y=150.0,
            width=105.0,
            height=42.0
        )

        self.personal_friend_code = self.canvas.create_text(
            19.0,
            75.0,
            anchor="nw",
            text="",
            fill="#FFFFFF",
            font=("MontserratRoman Bold", 16 * -1)
        )

        # ----------incoming friend request area---------

        friend_request_box_header_text = self.canvas.create_text(
            31.0,
            264.0,
            anchor="nw",
            text=" Friend Requests",
            fill="#FFFFFF",
            font=("MontserratRoman Bold", 16 * -1)
        )

        self.friend_request_area = tk.Text(
            self,
            bd=0,
            bg="#2A3441",
            fg="#2A3441",
            highlightthickness=0,
            cursor='arrow',
            state='disabled'
        )

        self.friend_request_area.place(
            x=9.0,
            y=294.0,
            width=190,
            height=201
        )

        sb = tk.Scrollbar(self.friend_request_area,
                          command=self.friend_request_area.yview, width=20)
        sb.pack(side='right', fill=tk.Y)
        self.friend_request_area.configure(yscrollcommand=sb.set)
        self.incoming_request_value = tk.StringVar(self.friend_request_area)

        # accept button
        self.accept_button_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_accept_button.png"))
        accept_button = tk.Button(
            self,
            image=self.accept_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.accept_friend_request(),
            relief="flat"
        )
        accept_button.place(
            x=213.0,
            y=294.0,
            width=119.23919677734375,
            height=42.0
        )

        # reject button

        self.reject_button_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_reject_button.png"))
        reject_button = tk.Button(
            self,
            image=self.reject_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.reject_friend_request(),
            relief="flat"
        )
        reject_button.place(
            x=213.0,
            y=453.0,
            width=119.23919677734375,
            height=42.0
        )

        # -----------current friends area-----------

        current_friends_box_header_text = self.canvas.create_text(
            468.0,
            264.0,
            anchor="nw",
            text="Friends",
            fill="#FFFFFF",
            font=("MontserratRoman Bold", 16 * -1)
        )

        self.friend_list_area = tk.Text(
            self,
            bd=0,
            bg="#2A3441",
            fg="#2A3441",
            highlightthickness=0,
            cursor='arrow',
            state='disabled'
        )
        self.friend_list_area.place(
            x=405,
            y=294.0,
            width=190,
            height=201
        )

        sb = tk.Scrollbar(self.friend_list_area,
                          command=self.friend_list_area.yview, width=20)
        sb.pack(side='right', fill=tk.Y)
        self.friend_list_area.configure(yscrollcommand=sb.set)
        self.friend_list_value = tk.StringVar(self.friend_list_area)

        # block button
        self.add_friend_block_button_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_block_button.png"))
        add_friend_block_button = tk.Button(
            self,
            image=self.add_friend_block_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.block_friend(),
            relief="flat"
        )
        add_friend_block_button.place(
            x=609.0,
            y=294.0,
            width=119.23919677734375,
            height=42.0
        )

        # unblock button

        self.add_friend_unblock_button_image = tk.PhotoImage(
            file=relative_to_assets("add_friend_unblock_button.png"))
        add_friend_unblock_button = tk.Button(
            self,
            image=self.add_friend_unblock_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.unblock_friend(),
            relief="flat"
        )
        add_friend_unblock_button.place(
            x=609.0,
            y=450.0,
            width=119.23919677734375,
            height=42.0
        )

        # outgoing friend requests

        pending_friend_request_box_header = self.canvas.create_text(
            505.0,
            38.0,
            anchor="nw",
            text="Pending",
            fill="#FFFFFF",
            font=("MontserratRoman Bold", 16 * -1)
        )

        self.pending_friends_area = tk.Text(
            self,
            bd=0,
            bg="#2A3441",
            fg="#2A3441",
            highlightthickness=0,
            cursor='arrow',
            state='disabled'
        )
        self.pending_friends_area.place(
            x=474,
            y=63.0,
            width=135,
            height=145
        )

        pending_friend_scrollbar = tk.Scrollbar(self.pending_friends_area,
                                                command=self.pending_friends_area.yview, width=20)
        pending_friend_scrollbar.pack(side='right', fill=tk.Y)
        self.pending_friends_area.configure(
            yscrollcommand=pending_friend_scrollbar.set)

    def send_friend_request(self):
        friend_code = self.friends_friend_code_entry.get()
        self.canvas.itemconfig(self.friend_code_error_message, state='hidden')
        self.canvas.itemconfig(self.friend_code_error_message, fill='#FF4747')

        if friend_code == '' or friend_code == self.client.user_id:
            self.canvas.itemconfig(
                self.friend_code_error_message, text='Please enter a valid friend code')
            self.canvas.itemconfig(
                self.friend_code_error_message, state='normal')
        elif self.client.check_if_user_is_already_friends(friend_code):
            self.canvas.itemconfig(
                self.friend_code_error_message, text='You are already friends with this person')
            self.canvas.itemconfig(
                self.friend_code_error_message, state='normal')
        elif not self.client.check_if_friend_code_exists(friend_code):
            self.canvas.itemconfig(self.friend_code_error_message,
                                   text='Friend code does not exist. Please enter a valid friend code')
            self.canvas.itemconfig(
                self.friend_code_error_message, state='normal')
        else:
            self.canvas.itemconfig(self.friend_code_error_message,
                                   text='Friend Request Sent')
            self.canvas.itemconfig(
                self.friend_code_error_message, fill='#FFFFFF')
            self.canvas.itemconfig(
                self.friend_code_error_message, state='normal')
            self.client.send_friend_request(friend_code)

    def add_text(self):
        self.canvas.itemconfig(self.personal_friend_code,
                               text=self.client.user_id)

    def copy_user_id_to_clipboard(self):
        t = tk.Tk()
        t.withdraw()
        t.clipboard_clear()
        t.clipboard_append(self.client.user_id)
        t.update()  # now it stays on the clipboard after the window is closed
        t.destroy()

    def add_radio_buttons(self):
        self.create_friend_list()
        self.create_friend_requests_list()
        self.create_pending_friends_list()

    def create_friend_list(self):
        for friend in self.client.friend_list:
            friend_screen_name = friend[2]
            friend_details = friend[0:2]
            status = friend[3]
            specifier_id = friend[4]
            self.add_friend_radiobutton_to_ui(
                friend_screen_name, friend_details, status, specifier_id)

    def add_friend_radiobutton_to_ui(self, txt, val, status, specifier_id):
        state = 'normal'
        colour = '#2A3441'
```

```python
        active = '#12161C'
        if status == 'blk' and specifier_id == self.client.user_id:  # you have blocked them
            colour = '#FF4747'
            active = '#612a2a'
        elif (status == 'blk' and specifier_id != self.client.user_id) or ('deleted account' in txt):  # they have blocked you
            state = 'disabled'

        btn = tk.Radiobutton(self.friend_list_area,
                             text=txt,
                             variable=self.friend_list_value,
                             value=val,
                             indicatoron=0,
                             bg=colour,
                             height=2,
                             width=23,
                             relief='flat',
                             fg='#E3E7ED',
                             selectcolor=active,
                             command=lambda: self.set_active_button_value(btn),
                             borderwidth=0,
                             font=("MontserratRoman", 9, 'normal'),
                             state=state
                             )

        self.friend_list_area.window_create('end', window=btn)
        self.friend_list_area.insert('end', '\n')

    def create_friend_requests_list(self):
        for request in self.client.friend_request_list:
            print(request)
            friend_user_id = request[0]
            friend_details = request[0:1]
            self.add_friend_request_radiobutton_to_ui(
                friend_user_id, friend_details)

    def update_friend_request_list(self, friend_user_id, friend_public_key):
        friend_details = (friend_user_id, friend_public_key)
        self.add_friend_request_radiobutton_to_ui(
            friend_user_id, friend_details)

    def add_friend_request_radiobutton_to_ui(self, txt, val):

        btn = tk.Radiobutton(self.friend_request_area,
                             text=txt,
                             variable=self.incoming_request_value,
                             value=val,
                             indicatoron=0,
                             bg='#2A3441',
                             height=2,
                             width=23,
                             relief='flat',
                             fg='#E3E7ED',
                             selectcolor='#12161C',
                             command=lambda: self.set_active_friend_request_button_value(
                                 btn),
                             borderwidth=0,
                             font=("MontserratRoman", 9, 'normal')
                             )

        self.friend_request_area.window_create('end', window=btn)
        self.friend_request_area.insert('end', '\n')

    def set_active_friend_request_button_value(self, btn):
        self.active_request_button = btn

    def set_active_button_value(self, btn):
        self.active_friend_button = btn

    def block_friend(self):
        if self.active_friend_button.cget('bg') != '#FF4747':
            self.active_friend_button.config(bg='#FF4747')
            self.active_friend_button.config(selectcolor='#1c1212')
            self.client.block_friend(
                self.friend_list_value.get().split(' ')[0])
        else:
            print('User already blocked')

    def unblock_friend(self):
        if self.active_friend_button.cget('bg') != '#2A3441':
            self.active_friend_button.config(bg='#2A3441')
            self.active_friend_button.config(selectcolor='#12161C')
            self.client.unblock_friend(
                self.friend_list_value.get().split(' ')[0])
        else:
            print('User already unblocked')

    def accept_friend_request(self):
        """
        - update personal database to be accepted
        - send message to client to update their own database to fit
        - delete radiobutton
        """
        friend_id = self.incoming_request_value.get().split(' ')[0]
        self.client.accept_friend_request(friend_id, self.client.user_id)
        self.active_request_button.destroy()

    def reject_friend_request(self):
        """
        - delete friend from personal database
        - semd message to client to delete their own db to fit
        - delete radiobutton
        """
        friend_id = self.incoming_request_value.get().split(' ')[0]
        self.client.reject_friend_request(friend_id)
        self.active_request_button.destroy()

    def create_pending_friends_list(self):
        for pending_request in self.client.pending_friend_list:
            friend_user_id = pending_request
            self.add_pending_friends_radiobutton_to_ui(friend_user_id)

    def update_pending_friends_list(self, friend_user_id):
        self.add_pending_friends_radiobutton_to_ui(friend_user_id)

    def add_pending_friends_radiobutton_to_ui(self, friend_user_id):
        btn = tk.Radiobutton(self.pending_friends_area,
                             text=friend_user_id,
                             indicatoron=0,
                             bg='#2A3441',
                             height=2,
                             width=15,
                             relief='flat',
                             fg='#E3E7ED',
                             selectcolor='#12161C',
                             borderwidth=0,
                             font=("MontserratRoman", 9, 'normal'),
                             state='disabled'
                             )

        self.pending_friends_area.window_create('end', window=btn)
        self.pending_friends_area.insert('end', '\n')


class SettingsPage(tk.Frame):
    def __init__(self, parent, window, client: secure_client.Client):
        self.controller = parent
        self.client = client
        tk.Frame.__init__(self, window)

    def on_show(self):
        """Function called when frame is shown"""
        self.client.stop_listen()
        self.create_and_place()
        self.add_text()

    def create_and_place(self):

        self.canvas = tk.Canvas(
            self,
            bg="#0A0C10",
            height=504,
            width=745,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )

        self.canvas.place(x=0, y=0)
        self.header_rectangle = self.canvas.create_rectangle(
            0.0,
            0.0,
            745.0,
            43.0,
            fill="#12161C",
            outline="")

        settings_header_text = self.canvas.create_text(
            16.0,
            8.0,
            anchor="nw",
            text="Settings",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        # CURRENT SCREEN NAME DISPLAY
        current_screen_name_header_text = self.canvas.create_text(
            16.0,
            61.0,
            anchor="nw",
            text="Current screen name",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        self.option_addcurrent_screen_name_image = tk.PhotoImage(
            file=relative_to_assets("settings_current_screen_name.png"))
        current_screen_name_bg = self.canvas.create_image(
            176.5,
            112.0,
            image=self.option_addcurrent_screen_name_image
        )
        self.current_screen_name_text = self.canvas.create_text(
            37.0,
            99.0,
            anchor="nw",
            text="",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        # CHANGE SCREEN NAME DISPLAY
        change_screen_name_text = self.canvas.create_text(
            16.0,
            160.0,
            anchor="nw",
            text="Change screen name",
            fill="#E3E7ED",
            font=("MontserratRoman Bold", 24 * -1)
        )

        self.new_screen_name_entry_image = tk.PhotoImage(
            file=relative_to_assets("settings_new_screen_name_entry.png"))
        new_screen_name_entry_bg = self.canvas.create_image(
            176.5,
            213.0,
            image=self.new_screen_name_entry_image
        )

        self.new_screen_name_entry = tk.Entry(
            self,
            bd=0,
            bg="#617998",
            fg="#E3E7ED",
            highlightthickness=0,
            font=("MontserratRoman")
        )

        self.new_screen_name_entry.place(
            x=37.0,
            y=192.0,
            width=279.0,
            height=40.0
        )

        self.new_screen_name_entry.bind(
            '<Return>', lambda e: self.change_screen_name())

        # CONFIRM SCREEN NAME BUTTON DISPLAY

        self.confirm_screen_name_button_image = tk.PhotoImage(
            file=relative_to_assets("settings_confirm_screen_name_button.png"))
        confirm_screen_name_button = tk.Button(
            self,
            image=self.confirm_screen_name_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.change_screen_name(),
            relief="flat",
            activebackground='#0A0C10'
        )
        confirm_screen_name_button.place(
            x=359.0,
            y=193.0,
            width=151.0,
            height=42.0
        )

        # DELETE ACCOUNT BUTTON DISPLAY

        self.delete_account_button_image = tk.PhotoImage(
            file=relative_to_assets("settings_delete_account_button.png"))
        delete_account_button = tk.Button(
            self,
            image=self.delete_account_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=self.popup_frame,
            relief="flat",
            activebackground='#0A0C10'
        )
        delete_account_button.place(
            x=16.0,
            y=436.0,
            width=229.0,
            height=42.0
        )

        # REQUEST DATA BUTTON DISPLAY

        self.request_data_button_image = tk.PhotoImage(
            file=relative_to_assets("settings_request_data_button.png"))
        request_data_button = tk.Button(
            self,
            image=self.request_data_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.request_all_user_data(),
            relief="flat",
            activebackground='#0A0C10'
        )
        request_data_button.place(
            x=443.0,
            y=436.0,
            width=302.0,
            height=42.0
        )

        # BACK TO CHATS BUTTON DISPLAY

        self.back_to_chats_button_image = tk.PhotoImage(
            file=relative_to_assets("settings_back_to_chats_button.png"))
        back_to_chats_button = tk.Button(
            self,
            image=self.back_to_chats_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.controller.show_frame(ChatPage),
            relief="flat",
            activebackground='#0A0C10'
        )
        back_to_chats_button.place(
            x=640.0,
            y=61.0,
            width=95.0,
            height=87.0
        )
        self.error_message = self.canvas.create_text(
            16.0,
            250.0,
            anchor="nw",
            text="",
            fill="#FF4747",
            font=("MontserratRoman Bold", 20 * -1),
            state='hidden'
        )

    def add_text(self):
        self.canvas.itemconfig(
            self.current_screen_name_text, text=self.client.screen_name)

    def change_screen_name(self):
        self.canvas.itemconfig(self.error_message, state='hidden')
        if self.new_screen_name_entry.get() == self.client.screen_name:
            # if screen name is same as old screen name display error message
            self.canvas.itemconfig(self.error_message, state='normal',
                                   text='Your new screen can not be the same as your
current screen name')
        elif self.new_screen_name_entry.get() == '':
            self.canvas.itemconfig(self.error_message, state='normal',
                                   text='Your new screen name can not be empty')
        elif "deleted account" in self.new_screen_name_entry.get():
            self.canvas.itemconfig(self.error_message, state='normal',
                                   text="This can't be your screen name")
        else:
            self.canvas.itemconfig(
                self.current_screen_name_text, text=self.new_screen_name_entry.get())
            self.client.change_screen_name(self.new_screen_name_entry.get())

    def request_all_user_data(self):
        path = self.client.get_output_path()
        if not path:
            # tk.messagebox.showerror(
            #     title="Invalid Path!", message="Enter a valid output path.")
            return
        output_path = Path(f"{path}/user_data_dump.txt").expanduser().resolve()
        if output_path.exists():
            response = tk1.askyesno(
                "Continue?",
                f"Directory {path} already contains user_data_dump.txt\n"
                "Do you want to continue and overwrite?")
            if not response:  # they said no
                return
        print('[Requesting user data]')
        user_data = self.client.request_all_user_data()
        try:
            output_file = open(output_path, 'w')
            output_file.write(user_data)
            tk.messagebox.showinfo(
                "Success!", f"User data successfully saved at {output_path}")
        except:
            tk.messagebox.showerror(
                title="Error", message=f"Something went wrong when writing to
{output_path}")

    # POPUP FRAME FUNCTIONS

    def popup_frame(self):
        top = tk.Toplevel(self.controller)
        top.protocol("WM_DELETE_WINDOW", self.popup_frame_error)
        self.controller.protocol("WM_DELETE_WINDOW", self.popup_frame_error)
        top.geometry("479x375")
        top.resizable(False, False)
        top.title("Confirm account deletion")
        self.disable_all_buttons()

        popup_canvas = tk.Canvas(
            top,
            bg="#0A0C10",
            height=375,
            width=479,
            bd=0,
            highlightthickness=0,
            relief="ridge"
        )

        popup_canvas.place(x=0, y=0)
        warning_header_text = popup_canvas.create_text(
            58.0,
            14.0,
            anchor="nw",
            text="Are you sure you want to \n   delete your account?",
            fill="#FF4747",
            font=("MontserratRoman Bold", 32 * -1)
        )

        self.yes_button_image = tk.PhotoImage(
            file=relative_to_assets("confirmation_popup_yes.png"))
        yes_button = tk.Button(
            top,
            image=self.yes_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.yes_button_clicked(top),
            relief="flat",
            activebackground='#2A3441'
        )
        yes_button.place(
            x=57.0,
            y=164.0,
            width=139.0,
            height=48.0
        )

        self.no_button_image = tk.PhotoImage(
            file=relative_to_assets("confirmation_popup_no.png"))
        no_button = tk.Button(
            top,
            image=self.no_button_image,
            borderwidth=0,
            highlightthickness=0,
            command=lambda: self.no_button_clicked(top),
            relief="flat",
            activebackground='#2A3441'
        )
        no_button.place(
            x=279.0,
            y=163.0,
            width=139.0,
            height=50.0
        )

    def popup_frame_error(self):
        tk.messagebox.showerror('Error', 'Please select either yes or no')

    def no_button_clicked(self, top):
        top.destroy()
        self.controller.protocol("WM_DELETE_WINDOW", self.controller.on_close)
        self.enable_all_buttons()

    def yes_button_clicked(self, top):
        print('[DELETING ACCOUNT]')
        if self.client.delete_account():
            self.controller.on_close(True)
        else:
            tk.messagebox.showerror(
                title="Account Deletion Error", message="Something went wrong when deleting your account please try again later")
            self.controller.protocol(
                "WM_DELETE_WINDOW", self.controller.on_close)
            self.enable_all_buttons()

    def disable_all_buttons(self):
        for widget in self.winfo_children():
            if isinstance(widget, tk.Button):
                try:
                    widget.config(state='disabled')
                except Exception as e:
                    print(e)

    def enable_all_buttons(self):
        for widget in self.winfo_children():
            if isinstance(widget, tk.Button):
                widget.config(state='normal')


# Driver Code
app = UIController()
app.resizable(False, False)
app.protocol("WM_DELETE_WINDOW", app.on_close)
app.mainloop()

```