

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

```
1. import tkinter as tk
2. from tkinter import *
3. from tkinter import ttk
4. from tkinter import scrolledtext
5. import sqlite3
6. from sqlite3 import Error
7. import datetime
8.
9. '''
10. colours
11. Purple = #8575ef
12. ChatBg = #ececce
13. Buttons = #2288e6
14. Black = #000000
15.
16. '''
17. cHeader = '#8575ef'
18. cButton = '#2288e6'
19. cWhite = '#ffffff'
20. cGrey = '#ececce'
21. cBlack = '#000000'
22. cButtonActive = '#1c75c7'
23.
24. class tkinterApp(tk.Tk):
25.     # self is the tkWindow and so has all the relevant functions
26.
27.     # __init__ function for class tkinterApp
28.     def __init__(self, *args, **kwargs):
29.         # colors
30.         p = '#8575ef'
31.
32.         # __init__ function for class Tk
33.         tk.Tk.__init__(self, *args, **kwargs)
34.         self.title('Orans App [!87]')
35.         self.resizable(False, False) # means cant resize window
36.         self.geometry('800x500')
37.
38.         # creating a container
39.         container = tk.Frame(self)
40.         container.pack(side="top", fill="both", expand=True)
41.
42.         container.grid_rowconfigure(0, weight=1)
43.         container.grid_columnconfigure(0, weight=1)
44.
45.         # initializing frames to an empty dict
46.         self.frames = {}
47.
48.         # iterating through a tuple consisting
49.         # of the different page layouts
50.         for F in (Login, CreateAccount, HomePage):
51.
52.             frame = F(container, self)
53.             # initializing frame of that object from
54.             # startpage, page1, page2 respectively with
55.             # for loop
56.
57.             self.frames[F] = frame
58.
59.             frame.grid(row=0, column=0, sticky="nsew")
60.
61.         self.show_frame(Login)
62.
63.         # to display the current frame passed as
64.         # parameter
65.         def show_frame(self, cont):
```

```
66.         frame = self.frames[cont]
67.         frame.tkraise()
68.
69. # lables should NOT existing in the __init__ function!
70.
71. class Login(tk.Frame, tk.Tk):
72.
73.     def __init__(self, parent, controller):
74.         tk.Frame.__init__(self, parent)
75.         # this line is needed but i honestly dont rly get how it helps with
self.controller.show_frame(HomePage)
76.         self.controller = controller
77.         # creating frames
78.         titleFrame = tk.Frame(self, width=800, height=100, bg=cWhite)
79.         titleFrame.grid(row=0, column=0, sticky='n')
80.         titleFrame.pack_propagate(False) # prevents frame resizing
81.
82.         usernameFrame = tk.Frame(self, width=800, height=100, bg=cGrey)
83.         usernameFrame.grid(row=1, column=0, sticky='n')
84.         usernameFrame.pack_propagate(False)
85.
86.         passwordFrame = tk.Frame(self, width=800, height=100, bg='blue')
87.         passwordFrame.grid(row=2, column=0, sticky='n')
88.         passwordFrame.pack_propagate(False)
89.
90.         ButtonFrame = tk.Frame(self, width=800, height=100, bg='red')
91.         ButtonFrame.grid(row=3, column=0, sticky='n')
92.         ButtonFrame.pack_propagate(False)
93.
94.         # creating lables
95.         loginTextLable = tk.Label(
96.             titleFrame, text='Login', foreground=cHeader, font=('Arial', 24))
97.
98.         usernameLable = tk.Label(usernameFrame, text='Username', bg='blue')
99.         username = tk.StringVar() # where the text entered is held
100.        usernameEntry = ttk.Entry(usernameFrame, textvariable=username)
101.
102.        passwordLable = ttk.Label(passwordFrame, text='Password')
103.        password = tk.StringVar() # where the text entered is held
104.        passwordEntry = ttk.Entry(
105.            passwordFrame, textvariable=password, show='*')
106.
107.        showPasswordButton = tk.Button(passwordFrame, fg=cWhite, bg=cButton,
activebackground=cButtonActive, activeforeground=cWhite,
108.            text='Show Password', command=lambda:
self.toggle_password(passwordEntry, showPasswordButton))
109.
110.        loginButton = tk.Button(ButtonFrame, fg=cWhite, bg=cButton,
activebackground=cButtonActive,
111.            activeforeground=cWhite, text='Login', command=lambda:
self.validateLogin(username, password))
112.
113.        createAccountButton = tk.Button(ButtonFrame, fg=cWhite, bg=cButton,
activebackground=cButtonActive,
114.            activeforeground=cWhite, text='Create Account',
command=lambda: controller.show_frame(CreateAccount))
115.
116.        # placing lables
117.        # .pack(side=TOP, anchor=NW, padx=10, pady=10)
118.        loginTextLable.pack(pady=20)
119.        usernameLable.pack(pady=10)
120.        usernameEntry.pack()
121.
122.        passwordLable.pack(pady=10)
123.        passwordEntry.pack()
124.
125.        showPasswordButton.pack(pady=10)
126.
127.        loginButton.pack(pady=10)
128.        createAccountButton.pack()
```

```
129.
130.     def validateLogin(self, username, password):
131.         print(username.get())
132.         print(password.get())
133.         self.controller.show_frame(HomePage)
134.
135.     def toggle_password(self, passwordEntry, showPasswordButton):
136.         if passwordEntry.cget('show') == '':
137.             passwordEntry.config(show='*')
138.             showPasswordButton.config(text='Show Password')
139.         else:
140.             passwordEntry.config(show='')
141.             showPasswordButton.config(text='Hide Password')
142.
143. class CreateAccount(Login):
144.
145.     def __init__(self, parent, controller):
146.         tk.Frame.__init__(self, parent)
147.
148.         # creating lables
149.         usernameLabel = ttk.Label(self, text='Username')
150.         username = tk.StringVar() # where the text entered is held
151.         usernameEntry = ttk.Entry(self, textvariable=username)
152.
153.         screennameLabel = ttk.Label(self, text='Screen name')
154.         screenname = tk.StringVar() # where the text entered is held
155.         screennameEntry = ttk.Entry(self, textvariable=screenname)
156.
157.         passwordLabel = ttk.Label(self, text='Password')
158.         password = tk.StringVar() # where the text entered is held
159.         passwordEntry = ttk.Entry(self, textvariable=password, show='*')
160.
161.         password2Label = ttk.Label(self, text='Enter Password Again')
162.         password2 = tk.StringVar() # where the text entered is held
163.         password2Entry = ttk.Entry(self, textvariable=password2, show='*')
164.
165.         showPasswordButton = ttk.Button(self, text='Show Password', command=lambda:
self.toggle_password(
166.             passwordEntry, showPasswordButton))
167.
168.         createAccount = ttk.Button(self, text='Create Account', command=lambda:
self.validateAccountCreation(
169.             username, screenname, password, password2, error))
170.
171.         error = ttk.Label(self, text='Error')
172.         # placing lables
173.         usernameLabel.grid(row=0, column=0)
174.         usernameEntry.grid(row=0, column=1)
175.
176.         screennameLabel.grid(row=1, column=0)
177.         screennameEntry.grid(row=1, column=1)
178.
179.         passwordLabel.grid(row=2, column=0)
180.         passwordEntry.grid(row=2, column=1)
181.
182.         password2Label.grid(row=3, column=0)
183.         password2Entry.grid(row=3, column=1)
184.
185.         showPasswordButton.grid(row=2, column=3)
186.         createAccount.grid(row=4, column=0)
187.
188.     def validateAccountCreation(self, username, screenname, password, password2, error):
189.         print(username.get())
190.         print(password.get())
191.         print(screenname.get())
192.         if password.get() != password2.get():
193.             error.grid(row=5, column=0)
194.             error.config(text='Passwords must match')
195.         # check for username clash
196.         # hash password
```

```
197.         self.addUserRec(username, screenname, password)
198.
199.     def addUserRec(self, username, screenname, password):
200.         try:
201.             conn = sqlite3.connect("userTbl.db")
202.             cursor = conn.cursor()
203.             cursor.execute("""INSERT INTO userTbl (Username, Screenname, Password) VALUES
204.             (?, ?, ?)""",
205.                             (username, screenname, password))
206.             conn.commit()
207.             print("record added " + username)
208.         except Error as e:
209.             print(e)
210.
211.         # map(lambda x: x*2, [a, b, c])
212.
213.     class Message(tk.Frame):
214.         def __init__(self, text, parent):
215.             tk.Frame.__init__(self, parent)
216.             self.text = text
217.             message = ttk.Label(parent, text=text)
218.             message.grid(row=1, column=1)
219.             print(message.winfo_height())
220.             # message will have to go in some sort of queue
221.             print(text)
222.
223.     class HomePage(tk.Frame, tk.Tk):
224.         def __init__(self, parent, controller):
225.             tk.Frame.__init__(self, parent)
226.             self.controller = controller
227.
228.         def displayMessage(self, message):
229.             currentTime = datetime.datetime.now()
230.             date = currentTime.strftime("%x")
231.             time = currentTime.strftime('%X')
232.             messageBox.config(state=tk.NORMAL)
233.             messageBox.insert(tk.END, 'Username: ' +
234.                                   message + ' [' + time + ']' + '\n')
235.             messageBox.config(state=tk.DISABLED)
236.             inputMessage.delete(0, END)
237.
238.         def sendMessage(self):
239.             displayMessage(self, message.get())
240.
241.         self.grid_rowconfigure(0, weight=1)
242.         # self.grid_columnconfigure(0, weight=1)
243.         # self.grid_columnconfigure(1, weight=1)
244.
245.         usersFrame = tk.Frame(self, width=150, height=450, bg=cWhite)
246.         usersFrame.grid(row=0, column=0, sticky='nsew')
247.         usersFrame.pack_propagate(False) # prevents frame resizing
248.         settingFrame = tk.Frame(self, width=150, height=50, bg=cGrey)
249.         settingFrame.grid(row=0, column=1, sticky='se')
250.         settingFrame.pack_propagate(False)
251.
252.         messageFrame = tk.Frame(self, width=650, height=450, bg=cGrey)
253.         messageFrame.grid(row=0, column=1, sticky='ne')
254.         messageFrame.pack_propagate(False)
255.
256.         inputFrame = tk.Frame(self, width=650, height=50, bg='blue')
257.         inputFrame.grid(row=1, column=1, sticky='ew')
258.         inputFrame.pack_propagate(False)
259.
260.         chatsLabel = tk.Label(usersFrame, text='Chats',
261.                               fg=cHeader, bg=cWhite, font=('Arial', 24))
262.         chatsLabel.pack(pady=10)
263.         message = tk.StringVar()
264.         inputMessage = tk.Entry(inputFrame, textvariable=message, width=70)
265.         inputMessage.pack(padx=10)
266.         inputMessage.bind('<Return>', sendMessage)
```

```
266.
267.     messagebox = scrolledtext.ScrolledText(messageFrame, width=650)
268.     messagebox.config(state=tk.DISABLED)
269.     messagebox.pack(side=BOTTOM)
270.
271.     activeChatName = 'Active Chat Name'
272.     recipientNameLabel = Label(
273.         messageFrame, text=activeChatName, fg=cHeader, bg=cWhite, font=('Arial', 16),
height=1)
274.     recipientNameLabel.pack(side=TOP, anchor=NW, padx=10, pady=10)
275.
276. if __name__ == '__main__':
277.     app = tkinterApp()
278.     app.mainloop()
279.
280.
```

Tkinter-Designer Output

```
1. # This file was generated by the Tkinter Designer by Parth Jadhav
2. # https://github.com/ParthJadhav/Tkinter-Designer
3.
4. from pathlib import Path
5.
6. # from tkinter import *
7. # Explicit imports to satisfy Flake8
8. from tkinter import Tk, Canvas, Entry, Text, Button, PhotoImage
9.
10. OUTPUT_PATH = Path(__file__).parent
11. # ASSETS_PATH = OUTPUT_PATH / \
12. # Path(r"C:\Users\orank\OneDrive\Desktop\Computer Science\Seperate
GUI\CREATEACCOUNT\build\assets\frame1")
13.
14. ASSETS_PATH = OUTPUT_PATH / \
15. Path(r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level
NEA\build\assets\frame0")
16.
17. def relative_to_assets(path: str) -> Path:
18.     return ASSETS_PATH / Path(path)
19.
20. window = Tk()
21.
22. window.geometry("745x504")
23. window.configure(bg="#0A0C10")
24.
25. canvas = Canvas(
26.     window,
27.     bg="#0A0C10",
28.     height=504,
29.     width=745,
30.     bd=0,
31.     highlightthickness=0,
32.     relief="ridge"
33. )
34.
35. canvas.place(x=0, y=0)
36.
37. # UI
38.
39. create_account_entry = PhotoImage(
40.     file=relative_to_assets("create_account_entry.png"))
41.
42. # USERNAME ENTRY
43. username_entry_image = canvas.create_image(
44.     205.5,
45.     149.5,
46.     image=create_account_entry
47. )
48. username_entry_feild = Entry(
49.     bd=0,
```

```
50.     bg="#617998",
51.     fg="#000716",
52.     highlightthickness=0
53. )
54. username_entry_feild.place(
55.     x=87.5,
56.     y=133.0,
57.     width=236.0,
58.     height=33.0
59. )
60.
61. canvas.create_text(
62.     85.0,
63.     110.0,
64.     anchor="nw",
65.     text="USERNAME",
66.     fill="#E3E7ED",
67.     font=("MontserratRoman Regular", 16 * -1)
68. )
69.
70. # SCREEN NAME ENTRY
71. screen_name_entry_image = canvas.create_image(
72.     205.5,
73.     217.5,
74.     image=create_account_entry
75. )
76. screen_name_entry_feild = Entry(
77.     bd=0,
78.     bg="#617998",
79.     fg="#000716",
80.     highlightthickness=0
81. )
82. screen_name_entry_feild.place(
83.     x=87.5,
84.     y=201.0,
85.     width=236.0,
86.     height=33.0
87. )
88.
89. canvas.create_text(
90.     85.0,
91.     178.0,
92.     anchor="nw",
93.     text="SCREEN NAME",
94.     fill="#E3E7ED",
95.     font=("MontserratRoman Regular", 16 * -1)
96. )
97.
98. # PASSWORD ENTRY
99. password_entry_bg = PhotoImage(
100.     file=relative_to_assets("create_account_password_entry.png"))
101.
102. password_entry_image = canvas.create_image(
103.     515.5,
104.     149.5,
105.     image=password_entry_bg
106. )
107. password_entry_feild = Entry(
108.     bd=0,
109.     bg="#617998",
110.     fg="#000716",
111.     highlightthickness=0,
112.     show='*'
113. )
114. password_entry_feild.place(
115.     x=422.5,
116.     y=133.0,
117.     width=186.0,
118.     height=33.0
119. )
```

```
120.
121. canvas.create_text(
122.     421.0,
123.     110.0,
124.     anchor="nw",
125.     text="PASSWORD",
126.     fill="#E3E7ED",
127.     font=("MontserratRoman Regular", 16 * -1)
128. )
129.
130. # CONFIRM PASSWORD ENTRY
131. confirm_password_image = canvas.create_image(
132.     540.5,
133.     217.5,
134.     image=create_account_entry
135. )
136. comfirm_password_feild = Entry(
137.     bd=0,
138.     bg="#617998",
139.     fg="#000716",
140.     highlightthickness=0,
141.     show='*'
142. )
143. comfirm_password_feild.place(
144.     x=422.5,
145.     y=201.0,
146.     width=236.0,
147.     height=33.0
148. )
149.
150. canvas.create_text(
151.     421.0,
152.     178.0,
153.     anchor="nw",
154.     text="CONFIRM PASSWORD",
155.     fill="#E3E7ED",
156.     font=("MontserratRoman Regular", 16 * -1)
157. )
158.
159. # CREATE ACCOUNT HEADER
160. canvas.create_text(
161.     227.0,
162.     35.0,
163.     anchor="nw",
164.     text="Create Account",
165.     fill="#E3E7ED",
166.     font=("MontserratRoman Bold", 40 * -1)
167. )
168.
169. # ERROR MESSAGE
170. canvas.create_text(
171.     240.0,
172.     422.0,
173.     anchor="nw",
174.     text="PASSWORDS MUST MATCH",
175.     fill="#FF4747",
176.     font=("MontserratRoman Bold", 20 * -1)
177. )
178.
179. submit_button_image = PhotoImage(
180.     file=relative_to_assets("create_account_submit.png"))
181.
182. submit_button = Button(
183.     image=submit_button_image,
184.     borderwidth=0,
185.     highlightthickness=0,
186.     command=lambda: print("[BUTTON CLICKED] submit_button"),
187.     relief="flat",
188.     activebackground='#0A0C10'
189. )
```



```
190. submit_button.place(
191.     x=293.0,
192.     y=266.0,
193.     width=149.0,
194.     height=43.0
195. )
196.
197. login_option_image = PhotoImage(
198.     file=relative_to_assets("create_account_login_option.png"))
199. login_option_button = Button(
200.     image=login_option_image,
201.     borderwidth=0,
202.     highlightthickness=0,
203.     command=lambda: print("[BUTTON CLICKED] login_option_button"),
204.     relief="flat",
205.     activebackground='#0A0C10'
206. )
207. login_option_button.place(
208.     x=227.0,
209.     y=342.0,
210.     width=286.0,
211.     height=48.0
212. )
213.
214. # create_account_eye_closed
215. button_image_3 = PhotoImage(
216.     file=relative_to_assets("create_account_eye_closed.png"))
217. button_3 = Button(
218.     image=button_image_3,
219.     borderwidth=0,
220.     highlightthickness=0,
221.     command=lambda: print("button_3 clicked"),
222.     relief="flat",
223.     activebackground='#0A0C10'
224. )
225. button_3.place(
226.     x=633.0,
227.     y=130.0,
228.     width=39.0,
229.     height=40.0
230. )
231. window.resizable(False, False)
232. window.mainloop()
233.
234.
```

Client Server Prototype Code

Client

```
1. import socket
2. import threading
3. global_host = '192.168.0.52'
4. local_host = '127.0.0.1'
5. host = '127.0.0.1'
6. port = 80 # tcp port
7.
8. def listen_for_messages_from_server(client):
9.     while 1:
10.         message = client.recv(2048).decode('utf-8')
11.         # message2 = client.recv(2048)
12.         if message != '':
13.             print(message)
14.             # username = message.split('~')[0]
15.             # content = message.split('~')[1]
16.             # print(f'[{username}] {content}')
17.             # print(f'UTF=8: {message2}')
18.             # client.close()
19.         else:
20.             print('Message recieved from client is empty')
```

```
21.
22. def send_message_to_server(client):
23.     while 1:
24.         message = input('->')
25.         if message != '':
26.             client.sendall(message.encode())
27.         else:
28.             print('Empty message')
29.             exit(0)
30.
31. def communicate_to_server(client):
32.     username = input('Enter username: ')
33.     if username != '':
34.         client.sendall(username.encode())
35.     else:
36.         print('Username cannpt be empty')
37.         exit(0)
38.
39.     threading.Thread(target=listen_for_messages_from_server,
40.                      args=(client,)).start()
41.     send_message_to_server(client)
42.
43. def main():
44.     client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
45.
46.     # conect to server
47.     try:
48.         client.connect((host, port))
49.         print(f'Sucsessful connection to {host}')
50.     except:
51.         print(f'Unable to connect to server {host} {port}')
52.
53.     communicate_to_server(client)
54.
55. if __name__ == '__main__':
56.     main()
57.
58.
```

Server

```
1. import socket
2. import threading
3.
4. host = '127.0.0.1'
5. local_host = '127.0.0.1' # local host. Allows it to host on itself
6. global_host = '192.168.0.52'
7. port = 80 # 0-65535 0 lets OS choose which port you are using #5050 tcp port
8. listiner_limit = 6 # maximum allowed connections
9. active_clients = [] # list of all connected users
10.
11. def listen_for_messages(client, username): # listen for upcoming messages
12.     while 1:
13.         message = client.recv(2048).decode('utf-8')
14.         if message != '':
15.             final_msg = username + '~' + message
16.             send_messages_to_all(final_msg)
17.         else:
18.             print(f'Message send from client {username} is empty')
19.
20. def send_message_to_client(client, message):
21.     client.sendall(message.encode()) # utf8 is encode
22.     print(message.encode())
23. # func to send any new message to ALL clients currently connected
24.
25. def send_messages_to_all(message):
26.     for user in active_clients:
27.         print(user)
```

```
28.         send_message_to_client(user[1], message)
29.
30. # func to handle client
31. # client socket object returned in the while loop with server.accept()
32. def client_handler(client):
33.     # server will listen for client message containing the username
34.     while 1:
35.         # 2048 is max message size. Decode into utf-8
36.         username = client.recv(2048).decode('utf-8')
37.         if username != '':
38.             active_clients.append((username, client))
39.             break
40.
41.     else:
42.         print('Client username empty')
43.
44.     threading.Thread(target=listen_for_messages,
45.                      args=(client, username,)).start()
46.
47. def main():
48.     # socket class object
49.     # AF_INET is IPv4
50.     # sock_stream is using TCP packets for communication. SOCK_DGRAM is UDP
51.     # creating server itself
52.     server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
53.     print('Program is running')
54.
55.     try:
56.         server.bind((host, port))
57.         print('Server running')
58.     except:
59.         print(
60.             f'Unable to bind to host {host} and port {server.getsockname()[1]}')
61.
62.     # setting server limit - how many possible connections
63.     server.listen(listener_limit)
64.
65.     # keep listening for client connections
66.     while 1:
67.         client, address = server.accept()
68.         print(f'Successfully connected to client {address[0]} {address[1]}")
69.
70.         # creating a new thread which will perform client_handler func at the same time as
the main loop
71.         threading.Thread(target=client_handler, args=(client,)).start()
72.
73. if __name__ == '__main__':
74.     main()
75.
76.
```

Technical Solution

Server and Client-Side SQL Approach Code

```
1. import sqlite3
2. from sqlite3 import Error
3. from random import randint
4.
5.
6. """
7. status:
8. req - requested
9. acc - accepted
10. blk - blocked
11. """
12.
13. class Database():
14.     def __init__(self, db_path: str, ) -> None:
```

```
15.         self.db_path = db_path
16.         self.conn = self.create_connection(db_path)
17.
18.     def create_connection(self, db_file: str):
19.         """ create a database connection to a SQLite database """
20.         conn = None
21.         try:
22.             conn = sqlite3.connect(db_file, check_same_thread=False)
23.             print(
24.                 f"\nConnected to Database\nSQLite3 Version: {sqlite3.version}\nDatabase
path: {db_file}\n")
25.         except Error as e:
26.             print(e)
27.
28.         return conn
29.
30.     def create_table(self, sql_create_x_table):
31.         """Creates a table using value of sql_create_x_table as the sql code"""
32.         try:
33.             c = self.conn.cursor()
34.             c.execute(sql_create_x_table)
35.         except Error as e:
36.             print(e)
37.
38.     def close_connection(self):
39.         self.conn.commit()
40.         self.conn.close()
41.
42.     def execute_update(self, sql: str, values: tuple):
43.         c = self.conn.cursor()
44.         try:
45.             c.execute(sql, values)
46.             self.conn.commit()
47.         except sqlite3.Error as e:
48.             print(f"UPDATE SQL ERROR: {e}\nfor {sql = }\n{values = }")
49.             self.conn.rollback()
50.         finally:
51.             c.close()
52.
53.     def execute_insert(self, sql: str, values: tuple):
54.         c = self.conn.cursor()
55.         try:
56.             c.execute(sql, values)
57.             self.conn.commit()
58.         except sqlite3.Error as e:
59.             print(f"INSERT SQL ERROR: {e}\nfor {sql = }\n{values = }")
60.             self.conn.rollback()
61.         finally:
62.             c.close()
63.
64.     # -----SERVER DATABASE FUNCTIONS-----
65.
66.     def server_tables(self):
67.         sql_create_user_table = """
68.         CREATE TABLE IF NOT EXISTS users(
69.         user_id text NOT NULL PRIMARY KEY,
70.         screen_name text NOT NULL,
71.         hashed_password text NOT NULL,
72.         salt blob NOT NULL,
73.         public_key text NOT NULL,
74.         UNIQUE(user_id)
75.         );"""
76.         if self.conn is not None:
77.             self.create_table(sql_create_user_table)
78.
79.     def add_user(self, user_id: str, screen_name: str, hashed_password: str, salt: bytes,
public_key: str) -> bool:
80.         """Adds a new user to the database"""
81.         success = False
82.         values = (user_id, screen_name, hashed_password, salt, public_key)
```

```
83.         sql = """INSERT INTO users(user_id, screen_name, hashed_password, salt, public_key)
VALUES(?,?,?,?,?)"""
84.         c = self.conn.cursor()
85.         try:
86.             c.execute(sql, values)
87.             self.conn.commit()
88.             print(f'\n[DB] New user {values} created\n')
89.             success = True
90.         except sqlite3.Error as e:
91.             print(e)
92.             self.conn.rollback()
93.         return success
94.
95.     def get_user_details(self, user_id):
96.         """Gets user_id, screen_name and public_key"""
97.         c = self.conn.cursor()
98.         c.execute(
99.             "SELECT user_id, screen_name, public_key from users WHERE user_id=?",
100.        (user_id,))
101.        return c.fetchall()[0]
102.
103.    def get_screen_name(self, user_id: str):
104.        """Gets the screen_name of the specified user_id"""
105.        c = self.conn.cursor()
106.        c.execute(
107.            "SELECT screen_name FROM users WHERE user_id=?", (user_id,))
108.        return c.fetchall()[0][0]
109.
110.    def get_public_key(self, user_id: str):
111.        """Gets the public key of the specified user_id"""
112.        c = self.conn.cursor()
113.        c.execute("SELECT public_key from users WHERE user_id=?", (user_id, ))
114.        return c.fetchall()[0][0]
115.
116.    def get_password(self, user_id: str):
117.        """Gets hashed password of the specified user_id"""
118.        c = self.conn.cursor()
119.        c.execute(
120.            "SELECT hashed_password, salt FROM users WHERE user_id=?", (user_id,))
121.        return c.fetchall()
122.
123.    def check_user_id_exists(self, user_id: str) -> bool:
124.        """Returns True if user exists. False if they do not"""
125.        c = self.conn.cursor()
126.        c.execute("""SELECT COUNT(*) FROM users WHERE user_id = ? ;""", (user_id,))
127.        return c.fetchall()[0][0] != 0
128.
129.    def update_screen_name_server(self, user_id: str, new_screen_name: str):
130.        values = (new_screen_name, user_id)
131.        sql = """
132.        UPDATE users
133.        SET screen_name = ?
134.        WHERE user_id = ?;
135.        """
136.        self.execute_update(sql, values)
137.
138.    def delete_account_server(self, user_id: str):
139.        success = False
140.        posfix = randint(10000000, 99999999)
141.        new_user_id = f"deleted account({posfix})"
142.        values = (new_user_id, new_user_id, user_id)
143.        sql = """
144.        UPDATE users
145.        SET user_id = ?, screen_name = ?
146.        WHERE user_id = ?;
147.        """
148.        c = self.conn.cursor()
149.        try:
150.            c.execute(sql, values)
151.            self.conn.commit()
```

```
151.         success = True
152.     except sqlite3.Error as e:
153.         print(e)
154.         self.conn.rollback()
155.         success = False
156.     finally:
157.         c.close()
158.         return success, new_user_id
159.
160. # -----CLIENT DATABASE FUNCTIONS-----
161.
162. def client_tables(self):
163.     sql_create_messages_table = """
164.     CREATE TABLE IF NOT EXISTS messages (
165.     message_id INTEGER PRIMARY KEY,
166.     friend_id text NOT NULL,
167.     encrypted_Epk blob NOT NULL,
168.     message_text blob NOT NULL,
169.     date text NOT NULL,
170.     time text NOT NULL,
171.     from_me integer NOT NULL,
172.     is_image integer NOT NULL,
173.     FOREIGN KEY (friend_id) REFERENCES friendships(friend_id)
174.     );"""
175.
176.     sql_create_friendships_table = """
177.     CREATE TABLE IF NOT EXISTS friendships (
178.     friend_id text NOT NULL UNIQUE,
179.     friend_screen_name text NOT NULL,
180.     public_key text NOT NULL,
181.     status text NOT NULL,
182.     specifier_id text NOT NULL,
183.     PRIMARY KEY (friend_id)
184.     );"""
185.
186.     if self.conn is not None:
187.         self.create_table(sql_create_messages_table)
188.         self.create_table(sql_create_friendships_table)
189.
190. def new_blocked_friend(self, user_id: str, friend_user_id: str):
191.     """Sets status to blk where user_id has blocked friend_user_id"""
192.     values = (user_id, friend_user_id)
193.     sql = """
194.     UPDATE friendships
195.     SET status = 'blk', specifier_id = ?
196.     WHERE friend_id = ?
197.     """
198.     self.execute_update(sql, values)
199.
200. def unblocked_friend(self, user_id: str, friend_user_id: str):
201.     """Sets status to acc where user_id has unblocked friend_user_id"""
202.     values = (user_id, friend_user_id)
203.     sql = """
204.     UPDATE friendships
205.     SET status = 'acc', specifier_id = ?
206.     WHERE friend_id = ?
207.     """
208.     self.execute_update(sql, values)
209.
210. def get_friend_list(self):
211.     c = self.conn.cursor()
212.     c.execute("""
213.     SELECT friend_id, public_key, friend_screen_name, status, specifier_id
214.     FROM friendships
215.     WHERE status = 'acc' or status = 'blk';
216.     """)
217.     return c.fetchall()
218.
219. def get_all_acc_friends_user_ids(self):
220.     c = self.conn.cursor()
```

```
221.         c.execute("""
222.             SELECT friend_id
223.             FROM friendships
224.             WHERE status = 'acc';
225.         """)
226.         return c.fetchall()
227.
228.     def get_friend_request_list(self, user_id: str):
229.         c = self.conn.cursor()
230.         c.execute("""
231.             SELECT friend_id, public_key
232.             FROM friendships
233.             WHERE status = 'req' and specifier_id != ?;
234.             """, (user_id,))
235.         return c.fetchall()
236.
237.     def get_pending_friends_list(self, user_id: str):
238.         print(f"GETTING PENDING FRIEND LIST")
239.         c = self.conn.cursor()
240.         c.execute("""
241.             SELECT friend_id
242.             FROM friendships
243.             WHERE status = 'req' and specifier_id = ?;
244.             """, (user_id,))
245.         return c.fetchall()
246.
247.     def add_new_friend_request(self, friend_id: str, friend_screen_name: str, public_key:
str, specifier_id: str):
248.         print(f"Adding new friend {friend_id = }")
249.         data = (friend_id, friend_screen_name, public_key, 'req', specifier_id)
250.         sql = """INSERT INTO friendships (friend_id, friend_screen_name, public_key,
status, specifier_id) VALUES (?, ?, ?, ?, ?)"""
251.         self.execute_insert(sql, data)
252.
253.     def check_if_user_is_already_friends(self, friend_user_id: str):
254.         c = self.conn.cursor()
255.         c.execute(
256.             "SELECT COUNT(*) FROM friendships WHERE friend_id = ?", (friend_user_id,))
257.         return c.fetchall()[0][0]
258.
259.     def accept_friend_request(self, friend_id: str, specifier_id: str):
260.         values = (specifier_id, friend_id)
261.         sql = """
262.             UPDATE friendships
263.             SET status = 'acc', specifier_id = ?
264.             WHERE friend_id = ?
265.             """
266.         self.execute_update(sql, values)
267.
268.     def reject_friend_request(self, friend_id: str):
269.         c = self.conn.cursor()
270.         try:
271.             c.execute("DELETE FROM friendships WHERE friend_id = ?", (friend_id,))
272.             self.conn.commit()
273.         except sqlite3.Error as e:
274.             print(f"DELETION SQL ERROR: {e}")
275.         finally:
276.             c.close()
277.
278.     def get_message_list(self, friend_id: str):
279.         c = self.conn.cursor()
280.         c.execute("""
281.             SELECT encrypted_Epk, message_text, date, time, from_me, is_image
282.             FROM messages
283.             WHERE friend_id = ?
284.             ORDER BY message_id ASC;
285.             """, (friend_id, ))
286.         return c.fetchall()
287.
```

```
288.     def store_message(self, friend_id: str, encrypted_Epk: bytes, encrypted_message: bytes,
date: str, time: str, from_me: int, is_image: int):
289.         values = (friend_id, encrypted_Epk,
290.                   encrypted_message, date, time, from_me, is_image)
291.         sql = """INSERT INTO messages(friend_id, encrypted_Epk, message_text, date, time,
from_me, is_image) VALUES(?,?,?,?,?,?,?)"""
292.         self.execute_insert(sql, values)
293.
294.     def update_friend_screen_name(self, friend_id: str, friend_screen_name: str):
295.         values = (friend_screen_name, friend_id)
296.         sql = """
297.         UPDATE friendships
298.         SET friend_screen_name = ?
299.         WHERE friend_id = ?;
300.         """
301.         self.execute_update(sql, values)
302.
303.     def friend_deleted_account(self, friend_id: str, account_deletion_name: str):
304.         values = (account_deletion_name, account_deletion_name, friend_id)
305.         sql = """
306.         UPDATE friendships
307.         SET friend_screen_name = ?, friend_id = ?
308.         WHERE friend_id = ?;
309.         """
310.         self.execute_update(sql, values)
311.
312.
```

Server-Side SQL Approach Code

```
1. import sqlite3
2. from sqlite3 import Error
3. from pathlib import Path
4.
5. database = r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level
NEA\build\appDatabase.db"
6.
7. def create_connection(db_file):
8.     """ create a database connection to a SQLite database """
9.     conn = None
10.    try:
11.        conn = sqlite3.connect(db_file)
12.        print(f"Sqlite3 Version: {sqlite3.version}")
13.    except Error as e:
14.        print(e)
15.
16.    return conn
17.
18. def get_db(conn):
19.     c = conn.cursor()
20.     c.execute("PRAGMA database_list;")
21.     return c.fetchone()[2]
22.
23. def create_table(conn, sql_create_x_table):
24.     try:
25.         c = conn.cursor()
26.         c.execute(sql_create_x_table)
27.     except Error as e:
28.         print(e)
29.
30. def add_user(conn, values):
31.     """Adds a user to the database"""
32.     sql = """INSERT INTO users(user_id, screen_name, password) VALUES(?,?,?)"""
33.     c = conn.cursor()
34.     c.execute(sql, values)
35.     conn.commit()
36.     print(f'New User {values} successfully created')
37.     return c.lastrowid
```



```
38.
39. def check_login(conn, values):
40.     """Verifys the users login information"""
41.     password_match = False
42.     c = conn.cursor()
43.     c.execute("SELECT password FROM users WHERE user_id=?", (values[0],))
44.     try:
45.         if c.fetchall()[0][0] == values[1]:
46.             password_match = True
47.     except:
48.         password_match = False
49.     return password_match
50.
51. def get_screen_name(conn, values):
52.     """Finds screen_name from user_id"""
53.     c = conn.cursor()
54.     c.execute("SELECT screen_name FROM users WHERE user_id=?", (values,))
55.     return c.fetchall()[0]
56.
57. def get_user_id(conn, values):
58.     """Finds user_id from screen_name"""
59.     c = conn.cursor()
60.     c.execute("SELECT user_id FROM users WHERE screen_name=?", (values,))
61.     return c.fetchall()[0][0]
62.
63. def add_friend(conn, values):
64.     """Sends a friend request to another user"""
65.     sql = """INSERT INTO friendships(
66.         user_one, user_two, status, specifier_id
67.     )
68.     VALUES(?,?,?,?)"""
69.     c = conn.cursor()
70.     c.execute(sql, values)
71.     conn.commit()
72.     print("New Friendship added")
73.     return c.lastrowid
74.
75. def update_friend_status(conn, values):
76.     """Updates an existing friendships status"""
77.     pass
78.
79. # def get_friend_list(conn, values):
80. #     """Retrieves a list of users friends"""
81. #     c = conn.cursor()
82. #     c.execute("""SELECT
83. #         CASE
84. #             WHEN participant_1 = ? THEN participant_2
85. #             WHEN participant_2 = ? THEN participant_1
86. #         END cgroup
87. #     FROM conversations
88. #     WHERE participant_1 = ? OR participant_2 = ?""", (values))
89. #     return c.fetchall()
90.
91. def get_friend_list(conn, values):
92.     c = conn.cursor()
93.     c.execute("""
94.         SELECT
95.             c.conversation_id,
96.             CASE
97.                 WHEN c.participant_1 = ? THEN u2.user_id
98.                 WHEN c.participant_2 = ? THEN u1.user_id
99.             END AS other_user_id,
100.            CASE
101.                WHEN c.participant_1 = ? THEN u2.screen_name
102.                WHEN c.participant_2 = ? THEN u1.screen_name
103.            END AS other_screen_name
104.        FROM
105.            conversations c
106.        JOIN
107.            users u1 ON c.participant_1 = u1.user_id
```

```
108.         JOIN
109.         users u2 ON c.participant_2 = u2.user_id
110.     WHERE
111.         ? IN (c.participant_1, c.participant_2);
112.     """ , (values))
113.     return c.fetchall()
114.
115. def new_message(conn, values):
116.     sql = """INSERT INTO messages(conversation_id, message_text, date, time, sender_id)
117.     VALUES(?, ?, ?, ?, ?)"""
117.     c = conn.cursor()
118.     c.execute(sql, values)
119.     conn.commit()
120.     return c.lastrowid
121.
122. def add_conversation(conn, values):
123.     sql = """INSERT INTO conversations(participant_1, participant_2) VALUES(?, ?)"""
124.     c = conn.cursor()
125.     c.execute(sql, values)
126.     conn.commit()
127.     return c.lastrowid
128.
129. def unique_username(conn, username):
130.     c = conn.cursor()
131.     c.execute("""SELECT COUNT(*) FROM users WHERE user_id = ? ;""", (username,))
132.     return c.fetchall()[0][0] == 0
133.
134. def get_users(conn):
135.     c = conn.cursor()
136.     c.execute("""select * from users""")
137.     return c.fetchall()
138.
139. def get_recipient_id(conn, values):
140.     c = conn.cursor()
141.     c.execute("""SELECT
142.         CASE
143.             WHEN participant_1 = ? THEN participant_2
144.             WHEN participant_2 = ? THEN participant_1
145.             END AS other_user_id
146.         FROM conversations
147.         WHERE conversation_id = ?""", (values))
148.     return c.fetchall()
149.
150. def get_message_history(conn, values):
151.     # get message history but replace all user name with the relevant screen name
152.     c = conn.cursor()
153.     c.execute(
154.         """SELECT m.message_id, m.conversation_id, m.message_text, m.date, m.time,
155.         m.sender_id, users.screen_name
156.         FROM messages m
157.         JOIN users ON m.sender_id = users.user_id
158.         WHERE m.conversation_id = ?
159.         ORDER BY m.message_id ASC""", (values,))
159.     return c.fetchall()
160.
161. def sql(conn):
162.     sql_create_user_table = """
163.     CREATE TABLE IF NOT EXISTS users (
164.         user_id text NOT NULL PRIMARY KEY,
165.         screen_name text NOT NULL,
166.         password text NOT NULL,
167.         UNIQUE(user_id)
168.     );"""
169.
170.     sql_create_conversations_table = """
171.     CREATE TABLE IF NOT EXISTS conversations (
172.         conversation_id integer PRIMARY KEY,
173.         participant_1 text NOT Null,
174.         participant_2 text NOT Null,
175.         FOREIGN KEY (participant_1) REFERENCES users(user_id),
```

```
176.         FOREIGN KEY (participant_2) REFERENCES users(user_id),
177.         CHECK (participant_1 != participant_2)
178.     );"""
179.
180.     sql_create_messages_table = """
181.     CREATE TABLE IF NOT EXISTS messages (
182.         message_id integer PRIMARY KEY,
183.         conversation_id INT,
184.         message_text text NOT Null,
185.         date text NOT NULL,
186.         time text NOT NULL,
187.         sender_id text NOT NULL,
188.         FOREIGN KEY (conversation_id) REFERENCES conversations(conversation_id)
189.     );"""
190.
191.     sql_create_friendships_table = """
192.     CREATE TABLE IF NOT EXISTS friendships (
193.         user_one text NOT NULL,
194.         user_two text NOT NULL,
195.         status text NOT NULL,
196.         specifier_id text NOT NULL,
197.         PRIMARY KEY (user_one, user_two),
198.         FOREIGN KEY (user_one) REFERENCES users(user_id),
199.         FOREIGN KEY (user_two) REFERENCES users(user_id),
200.         FOREIGN KEY (specifier_id) REFERENCES users(user_id),
201.         CHECK (user_one != user_two),
202.         CHECK (specifier_id = user_one OR specifier_id = user_two)
203.     );"""
204.
205.     if conn is not None:
206.         create_table(conn, sql_create_user_table)
207.         create_table(conn, sql_create_conversations_table)
208.         create_table(conn, sql_create_messages_table)
209.         create_table(conn, sql_create_friendships_table)
210.     else:
211.         print('Error! Cannot create db connection')
212.
213.     conn = create_connection(database)
214.
215.     sql(conn)
216.
217.
```

AES Code Version 1

```
1. S_BOX = [
2.     [0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
3.         0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76],
4.     [0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
5.         0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0],
6.     [0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
7.         0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15],
8.     [0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
9.         0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75],
10.    [0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
11.        0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84],
12.    [0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
13.        0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf],
14.    [0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
15.        0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8],
16.    [0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
17.        0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2],
18.    [0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
19.        0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73],
```

```
20.     [0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
21.         0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb],
22.     [0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
23.         0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79],
24.     [0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
25.         0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08],
26.     [0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
27.         0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a],
28.     [0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
29.         0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e],
30.     [0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
31.         0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf],
32.     [0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
33.         0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16],
34. ]
35.
36. CONSTANT_COLUMN = [[0x1, 0, 0, 0],
37.                     [0x2, 0, 0, 0],
38.                     [0x4, 0, 0, 0],
39.                     [0x8, 0, 0, 0],
40.                     [0x10, 0, 0, 0],
41.                     [0x20, 0, 0, 0],
42.                     [0x40, 0, 0, 0],
43.                     [0x80, 0, 0, 0],
44.                     [0x1b, 0, 0, 0],
45.                     [0x36, 0, 0, 0]]
46.
47. key = 'key123qwertyuioI'
48. key_schedule = []
49.
50. def init_variables():
51.     for i in range(11):
52.         temp = [[0 for col in range(4)] for row in range(4)]
53.         key_schedule.append(temp)
54.
55. # takes in 128 bit key as a string and returns a string of binary
56. def key_to_bin(key):
57.     binary = ''
58.     # {:0>8} means fill with 0 up to a max of 8 characters and align these zeros on the
left
59.     for character in key:
60.         binary += ('{:0>8}'.format(format(ord(character), 'b'))))
61.     return binary
62.
63. def rot_word(final_column):
64.     last = [final_column[-1]]
65.     rot_column = last + final_column[:-1]
66.     return (rot_column)
67.
68. def sub_word(rot_column):
69.     sub_word = []
70.     for byte in rot_column:
71.         # print(byte)
72.         byte = str(byte)
73.         row_nibble = int(byte[:4], 2)
74.         column_nibble = int(byte[4:], 2)
75.         sub_value = str(bin(int(S_BOX[row_nibble][column_nibble])))
76.         sub_value = '{:0>8}'.format(sub_value.replace('0b', ''))
77.         sub_word.append(sub_value)
78.     return sub_word
79.     # CHECK
80.
81. def round_constant(sub_column, current_round):
82.     round_column = []
83.     for i in range(4):
84.         ccb = '{:0>8}'.format(bin(CONSTANT_COLUMN[current_round][i]))
85.         sc = sub_column[i]
86.         # print(CONSTANT_COLUMN[current_round][i], int(sc, 2))
87.         xor = CONSTANT_COLUMN[current_round][i] ^ int(sc, 2)
88.         xor = '{:0>8}'.format(format(xor, 'b'))
```

```
89.         # print(ccb, sc, xor)
90.         round_column.append(xor)
91.     return round_column
92.
93. def xor(init_column, transformed_column):
94.     column = []
95.     for i in range(4):
96.         xor = int(init_column[i], 2) ^ int(transformed_column[i], 2)
97.         xor = '{:0>8}'.format(format(xor, 'b'))
98.         column.append(xor)
99.     return column
100.
101. def key_expansion(key):
102.     binary_key = key_to_bin(key)
103.     # adding binary_key to key schedule
104.     for i in range(4):
105.         for j in range(4):
106.             key_schedule[0][i][j] = binary_key[0:8]
107.             binary_key = binary_key[8:]
108.     # print(key_schedule)
109.     for current_round in range(10):
110.         # take last column, rotword, subword, rcon it then add keys
111.         final_column = key_schedule[current_round][3]
112.         transformed_column = (round_constant(
113.             sub_word(rot_word(final_column)), 1))
114.         # print(transformed_column)
115.         for i in range(4):
116.             transformed_column = xor(
117.                 key_schedule[current_round][i], transformed_column)
118.             key_schedule[current_round+1][i] = transformed_column
119.
120. def main():
121.     init_variables()
122.     key_expansion(key)
123.     print(key_schedule)
124.
125. main()
```

AES Code Version 2

```
1. # GLOBAL VARIABLES
2. key_schedule = []
3.
4. encoding = 'utf-8'
5.
6. S_BOX = (
7.     0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7,
0xAB, 0x76,
8.     0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4,
0x72, 0xC0,
9.     0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8,
0x31, 0x15,
10.    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27,
0xB2, 0x75,
11.    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3,
0x2F, 0x84,
12.    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C,
0x58, 0xCF,
13.    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C,
0x9F, 0xA8,
14.    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF,
0xF3, 0xD2,
15.    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D,
0x19, 0x73,
16.    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E,
0x0B, 0xDB,
17.    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95,
0xE4, 0x79,
```

```
18.     0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A,
0xAE, 0x08,
19.     0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD,
0x8B, 0x8A,
20.     0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1,
0x1D, 0x9E,
21.     0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55,
0x28, 0xDF,
22.     0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54,
0xBB, 0x16,
23. )
24.
25. INVERSE_S_BOX = (
26.     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
27. )
28.
29. CONSTANT_COLUMN = (0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36)
30.
31. MATRIX = [[0x2, 0x3, 0x1, 0x1], [0x1, 0x2, 0x3, 0x1],
32.            [0x1, 0x1, 0x2, 0x3], [0x3, 0x1, 0x1, 0x2]]
33.
34. INV_MATRIX = [[0x0e, 0x0b, 0x0d, 0x09], [0x09, 0x0e, 0x0b, 0x0d],
35.                [0x0d, 0x09, 0x0e, 0x0b], [0x0b, 0x0d, 0x09, 0x0e]]
36.
37. # KEY EXPANSION
38.
39. def bytes_to_matrix(key):
40.     # converts 16 bytes in format b'text' into a 4x4 matrix
41.     return [list(key[j:j+4]) for j in range(0, 16, 4)]
42.
43. def rot_word(column):
44.     # print(column)
45.     # puts item in front of list at back and shifts all forward 1
46.     return column[1:] + [column[0]]
47.
48. def sub_word(column, encrypt):
49.     # uses both nibbles of 1 bit as coordinates for s box substitution
50.     if encrypt == True:
51.         for i in range(4):
52.             column[i] = S_BOX[column[i]]
53.     else:
54.         for i in range(4):
55.             column[i] = INVERSE_S_BOX[column[i]]
56.     return column
57.
58. def round_constant(column, round):
59.     # xor column with column from constant column dependant on current round
60.     column[0] ^= CONSTANT_COLUMN[round]
61.     return column
62.
63. def xor(a, b):
64.     # xors 2 same length lists together
65.     for i in range(len(a)):
66.         a[i] ^= b[i]
67.     return a
68.
69. # INPUT: string
70. # OUTPUT: NONE
```

```
71.
72. def key_expansion(key):
73.     # key = key.encode(encoding)
74.     key = bytes_to_matrix(key)
75.     # Adding first round key
76.     key_schedule.append(key)
77.     # key expansion 10 rounds for 128 bit key
78.     for current_round in range(10):
79.         round_key = []
80.         # taking last column and applying set of operations to it
81.         final_column = key_schedule[current_round][3]
82.         transformed_column = round_constant(
83.             sub_word(rot_word(final_column), True), current_round)
84.         # using transformed_column to create next round key by xoring with previous round
keys
85.         for i in range(4):
86.             transformed_column = xor(
87.                 transformed_column, key_schedule[current_round][i])
88.             # copies value in list rather than list itself
89.             xor_column = transformed_column[:]
90.             round_key.append(xor_column)
91.         key_schedule.append(round_key)
92.         print(f"key_schedule: {key_schedule} \n\n")
93.
94. # AES ENCRYPTION
95. # Most functions here can be used for both encrypt and decrypt
96. # Hence the 'encrypt' bool that is passed into most functions
97.
98. def padding(data):
99.     # pads data to become a multiple of 128 bits
100.    bytes_remaining = (16-len(data) % 16)
101.    characters = (chr(bytes_remaining).encode(encoding)) * bytes_remaining
102.    data += characters
103.    return data
104.
105. def remove_padding(data):
106.     # x = final character in string
107.     # remove x ammount of characters from end of string
108.     data = sum(sum(data, []), []) # moving from 3D list to 1D list
109.     return data[:-data[-1]]
110.
111. def sub_bytes(block, encrypt):
112.     for i in range(len(block)):
113.         block[i] = sub_word(block[i], encrypt)
114.     return block
115.
116. def shift_rows(block):
117.     for i in range(4):
118.         block[i] = block[i][i:] + block[i][:i]
119.     return block
120.
121. def d_shift_rows(block):
122.     for i in range(4):
123.         block[i] = block[i][-i:] + block[i][:-i]
124.     return block
125.
126. def rotate(block):
127.     # swaps rows and columns
128.     block = list(zip(*block))
129.     return [list(block[i]) for i in range(len(block))]
130.
131. def gf_mult(a, b):
132.     # multiplication in gaussian feild 2^8
133.     if b == 1:
134.         # if b is one the result is a as a*1 = 1
135.         result = a
136.     else:
137.         result = 0
138.         for i in range(8):
139.             # loop through each bit in b, if it is one add to result
```

```
140.         if (b & 1):
141.             # if lsb = 1 xor with a
142.             result ^= a
143.             a_msb = a & 0x80
144.             a <<= 1 # shift a left to ensure it is multiplied by the correct power
145.             if a_msb:
146.                 # multiply here by irriducible polynomial
147.                 a ^= 0x11b
148.             b >>= 1 # move to next bit in b
149.     return result
150.
151. def mix_columns(block, encrypt):
152.     mixed_columns = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
153.     if encrypt == True:
154.         for i in range(4):
155.             for j in range(4):
156.                 for k in range(4):
157.                     mixed_columns[i][j] ^= gf_mult(block[i][k], MATRIX[j][k])
158.     else:
159.         for i in range(4):
160.             for j in range(4):
161.                 for k in range(4):
162.                     mixed_columns[i][j] ^= gf_mult(
163.                         block[i][k], INV_MATRIX[j][k])
164.     return mixed_columns
165.
166. def xor_key(block, round):
167.     # xors column with specific round key
168.     return [xor(block[i], key_schedule[round][i]) for i in range(4)]
169.
170. # INPUT: String
171. # OUTPUT: 3D list
172. def encrypt(plain_text, key):
173.     cipher_text = []
174.     key_expansion(key)
175.     # formatting input
176.     plain_text = plain_text.encode(encoding)
177.     padded_plain_text = padding(plain_text)
178.     # Encryption Algorithm - encrypts in blocks of 16 bytes
179.     for i in range(0, len(padded_plain_text), 16):
180.         block = padded_plain_text[i:i+16]
181.         block = bytes_to_matrix(block)
182.         # Start of actual encryption
183.         # XOR with IV before this step.
184.
185.         block = xor_key(block, 0)
186.         for round in range(1, 10):
187.             block = sub_bytes(block, True)
188.             block = rotate(shift_rows(rotate(block)))
189.             block = mix_columns(block, True)
190.             block = xor_key(block, round)
191.             block = sub_bytes(block, True)
192.             block = rotate(shift_rows(rotate(block)))
193.             block = xor_key(block, 10)
194.             # IV = block
195.             cipher_text.append(block)
196.     return cipher_text
197.
198. # INPUT: 3D list
199. # OUTPUT: String
200. def decrypt(cipher_text, key):
201.     plain_text = []
202.     key_expansion(key)
203.     # Decryption Algorithm - decrypts in blocks of 16 bytes
204.     for block in cipher_text:
205.         block = xor_key(block, 10)
206.         block = rotate(d_shift_rows(rotate(block)))
207.         block = sub_bytes(block, False)
208.         for round in range(1, 10):
209.             block = xor_key(block, 10-round)
```



```
210.         block = mix_columns(block, False)
211.         block = rotate(d_shift_rows(rotate(block)))
212.         block = sub_bytes(block, False)
213.         block = xor_key(block, 0)
214.         plain_text.append(block)
215.     plain_text = remove_padding(plain_text)
216.     return (''.join(map(chr, plain_text)))
```

AES Code Version 3

```
1. """
2. An OOP approach to AES in python.
3.
4. If you are using OOP you can inherit from aes.Encrypt and aes.Decrypt
5.
6. Otherwise create instances of each class and use .encrypt() and .decrypt()
7. """
8.
9. # CONSTANTS
10. S_BOX = (
11.     0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7,
12.     0xAB, 0x76,
13.     0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4,
14.     0x72, 0xC0,
15.     0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8,
16.     0x31, 0x15,
17.     0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27,
18.     0xB2, 0x75,
19.     0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3,
20.     0x2F, 0x84,
21.     0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C,
22.     0x58, 0xCF,
23.     0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C,
24.     0x9F, 0xA8,
25.     0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF,
26.     0xF3, 0xD2,
27.     0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D,
28.     0x19, 0x73,
29.     0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E,
30.     0x0B, 0xDB,
31.     0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95,
32.     0xE4, 0x79,
33.     0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A,
34.     0xAE, 0x08,
35.     0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD,
36.     0x8B, 0x8A,
37.     0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1,
38.     0x1D, 0x9E,
39.     0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55,
40.     0x28, 0xDF,
41.     0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54,
42.     0xBB, 0x16,
43. )
44.
45. INVERSE_S_BOX = (
46.     82, 9, 106, 213, 48, 54, 165, 56, 191, 64, 163, 158, 129, 243, 215, 251, 124, 227, 57,
47.     130, 155, 47, 255, 135, 52, 142, 67, 68, 196, 222, 233, 203, 84, 123, 148, 50, 166, 194, 35, 61,
48.     238, 76, 149, 11, 66, 250, 195, 78, 8, 46, 161, 102, 40, 217, 36, 178, 118, 91, 162, 73, 109,
49.     139, 209, 37, 114, 248, 246, 100, 134, 104, 152, 22, 212, 164, 92, 204, 93, 101, 182, 146, 108,
50.     112, 72, 80, 253, 237, 185, 218, 94, 21, 70, 87, 167, 141, 157, 132, 144, 216, 171, 0, 140, 188,
51.     211, 10, 247, 228, 88, 5, 184, 179, 69, 6, 208, 44, 30, 143, 202, 63, 15, 2, 193, 175, 189, 3,
52.     1, 19, 138, 107, 58, 145, 17, 65, 79, 103, 220, 234, 151, 242, 207, 206, 240, 180, 230, 115,
53.     150, 172, 116, 34, 231, 173, 53, 133, 226, 249, 55, 232, 28, 117, 223, 110, 71, 241, 26, 113,
54.     29, 41, 197, 137, 111, 183, 98, 14, 170, 24, 190, 27, 252, 86, 62, 75, 198, 210, 121, 32, 154,
55.     219, 192, 254, 120, 205, 90, 244, 31, 221, 168, 51, 136, 7, 199, 49, 177, 18, 16, 89, 39, 128,
56.     236, 95, 96, 81, 127, 169, 25, 181, 74, 13, 45, 229, 122, 159, 147, 201, 156, 239, 160, 224, 59,
57.     77, 174, 42, 245, 176, 200, 235, 187, 60, 131, 83, 153, 97, 23, 43, 4, 126, 186, 119, 214, 38,
58.     225, 105, 20, 99, 85, 33, 12, 125
59. )
60.
61. """
```

```
33. CONSTANT_COLUMN = (0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36)
34.
35. MATRIX = [[0x2, 0x3, 0x1, 0x1], [0x1, 0x2, 0x3, 0x1],
36.            [0x1, 0x1, 0x2, 0x3], [0x3, 0x1, 0x1, 0x2]]
37.
38. INV_MATRIX = [[0x0e, 0x0b, 0x0d, 0x09], [0x09, 0x0e, 0x0b, 0x0d],
39.               [0x0d, 0x09, 0x0e, 0x0b], [0x0b, 0x0d, 0x09, 0x0e]]
40.
41. ENCODING = 'utf-8'
42.
43. class keyExpansion():
44.
45.     def key_expansion(self, key: bytes):
46.         key_schedule = []
47.         # key = key.encode(encoding)
48.         key = self.bytes_to_matrix(key)
49.         # Adding first round key
50.         key_schedule.append(key)
51.         # key expansion 10 rounds for 128 bit key
52.         for current_round in range(10):
53.             round_key = []
54.             # taking last column and applying set of operations to it
55.             final_column = key_schedule[current_round][3]
56.             transformed_column = self.round_constant(
57.                 self.sub_word(self.rot_word(final_column), True), current_round)
58.             # using transformed_column to create next round key by xoring with previous
round keys
59.             for i in range(4):
60.                 transformed_column = self.xor(
61.                     transformed_column, key_schedule[current_round][i])
62.                 # copies value in list rather than list itself
63.                 xor_column = transformed_column[:]
64.                 round_key.append(xor_column)
65.                 key_schedule.append(round_key)
66.         return key_schedule
67.
68.     def bytes_to_matrix(self, key: bytes) -> list:
69.         """converts 16 bytes into a 4x4 matrix"""
70.         return [list(key[j:j+4]) for j in range(0, 16, 4)]
71.
72.     def rot_word(self, column):
73.         """Shifts all items in list forward one with the front most item moving to the
back"""
74.         return column[1:] + [column[0]]
75.
76.     def sub_word(self, column, encrypt):
77.         # uses both nibbles of 1 bit as coordinates for s box substitution
78.         if encrypt == True:
79.             for i in range(4):
80.                 column[i] = S_BOX[column[i]]
81.         else:
82.             for i in range(4):
83.                 column[i] = INVERSE_S_BOX[column[i]]
84.         return column
85.
86.     def round_constant(self, column, round):
87.         # xor column with column from constant column dependant on current round
88.         column[0] ^= CONSTANT_COLUMN[round]
89.         return column
90.
91.     def xor(self, a, b):
92.         # xors 2 same length lists together
93.         for i in range(len(a)):
94.             a[i] ^= b[i]
95.         return a
96.
97. class SharedFunctions(keyExpansion):
98.     def sub_bytes(self, block, encrypt: bool):
99.         """
100.         Substitues all bytes in block for relevant bytes in S_BOX
```

```
101.     Parameters:
102.     - encrypt (bool): If encrypting set True // If decrypting set False
103.     """
104.     for i in range(len(block)):
105.         block[i] = self.sub_word(block[i], encrypt)
106.     return block
107.
108. def rotate(self, block):
109.     """swaps rows and columns"""
110.     block = list(zip(*block))
111.     return [list(block[i]) for i in range(len(block))]
112.
113. def gf_mult(self, a: int, b: int) -> int:
114.     """multiplication in gaussian feild 2^8"""
115.     if b == 1:
116.         result = a
117.     else:
118.         result = 0
119.         for i in range(8):
120.             if (b & 1): # if lsb = 1 xor with a
121.                 result ^= a
122.             sgt255 = a & 0x80
123.             a <<= 1 # shift a left
124.             if sgt255: # modulo
125.                 a ^= 0x11b
126.             b >>= 1 # shift b down
127.         return result
128.
129. def mix_columns(self, block, encrypt: bool):
130.     """
131.     Uses matrix multiplicatin to mix
132.     Parameters:
133.     - encrypt (bool): If encrypting set True // If decrypting set False
134.     """
135.     mixed_columns = [[0, 0, 0, 0], [
136.         0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
137.     if encrypt == True:
138.         for i in range(4):
139.             for j in range(4):
140.                 for k in range(4):
141.                     mixed_columns[i][j] ^= self.gf_mult(
142.                         block[i][k], MATRIX[j][k])
143.     else:
144.         for i in range(4):
145.             for j in range(4):
146.                 for k in range(4):
147.                     mixed_columns[i][j] ^= self.gf_mult(
148.                         block[i][k], INV_MATRIX[j][k])
149.     return mixed_columns
150.
151. def xor_key(self, block, round: int, key_schedule: list):
152.     # xors column with specific round key
153.     return [self.xor(block[i], key_schedule[round][i]) for i in range(4)]
154.
155. class Encrypt(SharedFunctions):
156.
157.     def encrypt(self, plain_text: bytes, key: bytes) -> bytes:
158.         cipher_text = []
159.         key_schedule = self.key_expansion(key)
160.         # formatting data
161.         # plain_text = plain_text.encode(ENCODING)
162.         padded_plain_text = self.padding(plain_text)
163.         # Encrpytion Algorithm - encrypts in blocks of 16 bytes
164.         for i in range(0, len(padded_plain_text), 16):
165.             block = padded_plain_text[i:i+16]
166.             block = self.bytes_to_matrix(block)
167.             # Start of actual encryption
168.             # XOR with IV before this step.
169.
170.             block = self.xor_key(block, 0, key_schedule)
```

```
171.         for round in range(1, 10):
172.             block = self.sub_bytes(block, True)
173.             block = self.rotate(
174.                 self.encrypt_shift_rows(self.rotate(block)))
175.             block = self.mix_columns(block, True)
176.             block = self.xor_key(block, round, key_schedule)
177.             block = self.sub_bytes(block, True)
178.             block = self.rotate(self.encrypt_shift_rows(self.rotate(block)))
179.             block = self.xor_key(block, 10, key_schedule)
180.             # IV = block
181.             cipher_text.append(block)
182.
183.     # Convert the flattened list to a continuous byte stream
184.     # flattened_cipher_text = [
185.     #     byte for sublist in cipher_text for byte in sublist]
186.
187.     # Decode the bytes to UTF-8
188.     # utf8_result = bytes(flattened_cipher_text)
189.     ct = sum(sum(cipher_text, []), [])
190.     return bytes(ct)
191.
192. def padding(self, data: bytes) -> bytes:
193.     """padds data to become a multiple of 128 bits"""
194.     bytes_remaining = (16-len(data)) % 16
195.     characters = (chr(bytes_remaining).encode(ENCODING)) * bytes_remaining
196.     data += characters
197.     return data
198.
199. def encrypt_shift_rows(self, block):
200.     for i in range(4):
201.         block[i] = block[i][i:] + block[i][:i]
202.     return block
203.
204. class Decrypt(SharedFunctions):
205.
206.     def decrypt(self, cipher_text: bytes, key: bytes) -> bytes:
207.         plain_text = []
208.         key_schedule = self.key_expansion(key)
209.         # Decryption Algorithm - decrypts in blocks of 16 bytes
210.
211.         for i in range(0, len(cipher_text), 16):
212.             block = cipher_text[i:i+16]
213.             block = self.bytes_to_matrix(block)
214.
215.             # STARTING THE ACTUAL DECRYPTION NOW BABY
216.             block = self.xor_key(block, 10, key_schedule)
217.             block = self.rotate(self.decrypt_shift_rows(self.rotate(block)))
218.             block = self.sub_bytes(block, False)
219.             for round in range(1, 10):
220.                 block = self.xor_key(block, 10-round, key_schedule)
221.                 block = self.mix_columns(block, False)
222.                 block = self.rotate(
223.                     self.decrypt_shift_rows(self.rotate(block)))
224.                 block = self.sub_bytes(block, False)
225.             block = self.xor_key(block, 0, key_schedule)
226.             plain_text.append(block)
227.
228.             # flattened_plain_text = [
229.             #     byte for sublist in plain_text for byte in sublist]
230.             pt = self.remove_padding(plain_text)
231.
232.             # ptt = sum(sum(pt, []), [])
233.
234.             return bytes(pt)
235.             # plain_text = self.remove_padding(plain_text)
236.             # return (''.join(map(chr, plain_text))).encode()
237.
238.     def remove_padding(self, data):
239.         # x = final character in string
240.         # remove x ammount of characters from end of string
```

```
241.         data = sum(sum(data, []), []) # moving from 3D list to 1D list
242.         return data[:-data[-1]]
243.
244.     def decrypt_shift_rows(slef, block):
245.         for i in range(4):
246.             block[i] = block[i][-i:] + block[i][:i]
247.         return block
248.
249.
```

Server Version 1

```
1. import socket
2. import threading
3. import json
4. import appDatabase as sql
5. import atexit
6.
7. HEADER = 64 # make bigger if new message is needed
8. PORT = 65432 # TCP/UDP packets
9.
10. # 127.0.0.1
11. SERVER = socket.gethostbyname(socket.gethostname())
12. ADDR = (SERVER, PORT)
13. FORMAT = 'utf-8'
14. database = r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level
NEA\build\appDatabase.db"
15. conn = sql.create_connection(database)
16.
17. active_clients = []
18. logged_in_clients = []
19.
20. class ClientDisconnectException(Exception):
21.     pass
22.
23. class USER():
24.     def __init__(self, client, addr):
25.         self.user_id = ''
26.         self.client = client
27.         self.addr = addr
28.         self.public_key = None
29.         self.symmetric_key = None
30.         self.screen_name = ''
31.         self.friends = None
32.         active_clients.append(self)
33.
34.     def send_data(self, data):
35.         data = json.dumps(data)
36.         message = data.encode(FORMAT)
37.         self.client.send(self.packet_header(message))
38.         self.client.send(message)
39.
40.     def packet_header(self, data):
41.         data_length = len(data)
42.         send_length = str(data_length).encode(FORMAT)
43.         send_length += b' ' * (HEADER - len(send_length))
44.         return send_length
45.
46.     def recive_data(self):
47.         data_length = self.client.recv(HEADER).decode(FORMAT)
48.         if data_length != 0:
49.             data_length = int(data_length)
50.             json_data = self.client.recv(data_length).decode(FORMAT)
51.             data = json.loads(json_data)
52.             if data['type'] == 'DISCONNECT':
53.                 handel_disconnect(self)
54.                 raise ClientDisconnectException('Client Disconnected')
55.
```

```
56.         else:
57.             return data
58.
59. def handel_disconnect(user):
60.     active_clients.remove(user)
61.     user.client.close()
62.     print(
63.         f'[DISCONNECTED] {user.addr[0], user.addr[1], user.user_id} disconnected')
64.
65. def ppkp(new_user):
66.     """Public Private Key Protocol"""
67.     pass
68.
69. def verify_login(user_id, password):
70.     # print("[FUNCTION RUNNING] verify_login")
71.     """
72.     1) Hash password
73.     2) Check password against stored hash with SQL
74.     3) If a match send login success back to user
75.     3a) Send Screen name friends list etc back to user
76.     4) If not send login failure back to user
77.     """
78.     if sql.check_login(conn, (user_id, password)):
79.         return True
80.     else:
81.         return False
82.
83. def handel_login(new_user):
84.     valid_password = False
85.     print(f'[HANDLING] Login for {new_user.addr[0]}')
86.     login_data = new_user.recv_data()
87.     user_id = login_data['user_id']
88.     password = login_data['password']
89.     if verify_login(user_id, password):
90.         new_user.user_id = user_id
91.         # print(f'[USER LOGGED IN]{new_user.user_id}, {new_user.screen_name}')
92.         valid_password = True
93.         # send true
94.     else:
95.         valid_password = False
96.     send_validation = {
97.         'valid_password': valid_password
98.     }
99.     new_user.send_data(send_validation)
100.    if valid_password:
101.        send_ui_data(new_user)
102.    return valid_password
103.
104. def handel_create_account(new_user):
105.     unique = False
106.     print('[HANDLING] Create account')
107.     username = new_user.recv_data()['username']
108.     # sql checking
109.     if sql.unique_username(conn, username):
110.         unique = True
111.     new_user.send_data({'unique': unique})
112.     if unique:
113.         data = new_user.recv_data()
114.         screen_name = data['screen_name']
115.         password = data['password']
116.         new_user.user_id = username
117.         new_user.screen_name = screen_name
118.         new_user.password = password
119.         sql.add_user(
120.             conn, (new_user.user_id, new_user.screen_name, new_user.password))
121.     return unique
122.
123. # -----
124.
125. def send_ui_data(user):
```

```
126.     user.screen_name = sql.get_screen_name(conn, (user.user_id))[0]
127.     user.friends = sql.get_friend_list(conn, (user.user_id, user.user_id,
128.                                             user.user_id, user.user_id, user.user_id))
129.     ui_data = {
130.         'friend_list': user.friends,
131.         'screen_name': user.screen_name
132.     }
133.     user.send_data(ui_data)
134.
135. def send_message_to_client(user, conn, data):
136.     recipient = sql.get_recipient_id(
137.         conn, (user.user_id, user.user_id, data['chat_id']))
138.     for c in active_clients:
139.         if c.user_id == recipient[0][0]:
140.             c.send_data(data)
141.             print(f'[SENT MESSAGE] From: {user.user_id} to {recipient[0][0]}')
142.
143. def handel_initial_contact(new_user):
144.     """
145.     1) Client connect so server
146.     2) Client sends public key to establish a secure tunnle
147.     3) Client sends symmetrical encryption key
148.     4) Waits for message dictating login or account creation
149.     5) handel accoringly by calling different functions
150.     """
151.     print(f"[NEW CONNECTION] {new_user.addr[0], new_user.addr[1]} connected.")
152.     print(f"[TOTAL CONNECTIONS] {len(active_clients)}\n")
153.     login = False
154.     while login != True:
155.         determiner = new_user.recv_data()
156.         if determiner['choice'] == 'login':
157.             if handel_login(new_user):
158.                 login = True
159.         elif determiner['choice'] == 'create account':
160.             if handel_create_account(new_user):
161.                 login = True
162.         else:
163.             print('Error')
164.
165.     return login
166.
167. def handel_logged_in_client(user): # runs for each new client
168.     handel_client_conn = sql.create_connection(database)
169.     # client contains info about connected client
170.     # addr contains just IP and source port eg the port the client is sending from
171.     print(
172.         f"[USER LOGIN] {user.addr[0], user.addr[1], user.user_id} logged in.\n")
173.     # active_clients.append(user)
174.     connected = True
175.     # screen_name = sql.get_screen_name(handel_client_conn, user.user_id)
176.     # screen_name_data = {
177.     #     'screen_name': screen_name
178.     # }
179.     # user.send_data(screen_name_data)
180.     try:
181.         while connected:
182.             data = user.recv_data()
183.
184.             if data['type'] == 'message history request':
185.                 # print(data['conversation_id'])
186.                 message_history = sql.get_message_history(
187.                     handel_client_conn, (data['conversation_id']))
188.                 user.send_data({'type': 'message history',
189.                                'message history': message_history})
190.                 print(
191.                     f"[DATA SEND] {user.user_id} message history for chat
192.                     {data['conversation_id']}")
193.                 elif data['type'] == 'update screen name':
194.                     print('Updating Screen name')
195.             else:
```

```
195.         print(
196.             f"[RECIEVED MESSAGE] addr[0]: {user.addr[0]}, addr[1]: {user.addr[1]},
{user.user_id} in {data['chat_id']}: {data['message']}"
197.             sql.new_message(
198.                 handel_client_conn, (data['chat_id'], data['message'], data['date'],
data['time'], user.user_id))
199.             send_message_to_client(user, handel_client_conn, data)
200.     except ClientDisconnectException:
201.         print('[CLIENT DISCONNECT] Logged in TRUE')
202.         connected = False
203.     finally:
204.         return handel_client_conn
205.
206. def start():
207.     """Starts the server and calls functions to handel client logic"""
208.
209.     server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
210.     try:
211.         server.bind(ADDR)
212.     except:
213.         print(f"Unable to bind to server {SERVER} and port {PORT}")
214.
215.     server.listen()
216.     print(f"[LISTENING] Server is listening on {SERVER}")
217.
218.     while True:
219.         client, addr = server.accept() # waits till new connection
220.         new_user = USER(client, addr)
221.         try:
222.             login = handel_inital_contact(new_user)
223.             if login:
224.                 logged_in_clients.append(new_user)
225.                 thread = threading.Thread(
226.                     target=handel_logged_in_client, args=(new_user, ))
227.                 thread.start()
228.                 print(
229.                     f"[ACTIVE LOGGED IN CONNECTIONS] {threading.active_count() - 1} \n")
230.             except ClientDisconnectException:
231.                 print('[CLIENT DISCONNECT] Logged in FALSE')
232.
233. print("[STARTING] server is starting...")
234. start()
```

NetworkingProtocols.py

```
1. """
2. Provides a base class with generic send/recieve functions but with in built data
serialization
3. """
4. # Cryptography imports
5. import rsa
6. import class_based_aes as aes
7.
8. # Data serialization imports
9. import json
10. import pickle
11. import base64
12.
13. HEADER = 2048 # used for send message protocol
14. FORMAT = 'utf-8'
15. # PORT = 65432 # TCP/UDP packets
16. # SERVER = "192.168.0.30"
17. # ADDR = (SERVER, PORT)
18.
19. class ClientDisconnectException(Exception):
20.     pass
21.
22. class BaseClass(aes.Encrypt, aes.Decrypt):
```



```
23.     def __init__(self, cs: str, client, addr):
24.         """
25.         Parameters:
26.         - type (str): Should be either 'SERVER' or 'CLIENT'.
27.         - client: (socket)
28.         - addr: (tuple)
29.         """
30.         self.type = cs
31.         self.client = client
32.         self.addr = addr
33.
34.     def validate_signature(self, serialized_data, deserialized_signature) -> bool:
35.         """Validates a signature for argument passed into serialized_data """
36.         signature = deserialized_signature['signature']
37.         public_key = deserialized_signature['public_key']
38.         valid = False
39.         try:
40.             rsa.verify(serialized_data, signature, public_key)
41.             valid = True
42.         except:
43.             valid = False
44.         finally:
45.             return valid
46.
47.     def generate_signature(self, serialized_data: bytes, private_key, public_key) ->
dict[str, bytes]:
48.         """
49.         Creates a signature for parameter passed into serialized_data.
50.
51.         Returns signature along with public key to a dict with keys: signature, public_key
52.         """
53.         a = {'signature': rsa.sign(
54.             serialized_data, private_key, 'SHA-1'), 'public_key': public_key}
55.         return a
56.
57.     def add_packet_header(self, data: bytes) -> bytes:
58.         """Returns bytes of exactly HEADER length with the first few bytes containing the
number of bytes argument data is"""
59.         data_length = len(data)
60.         send_length = str(data_length).encode(FORMAT)
61.         send_length += b' ' * (HEADER - len(send_length))
62.         return send_length
63.
64.     def send_with_header(self, data: bytes):
65.         """Sends a fixed sized packet containing the number of bytes in the next packet"""
66.         self.client.send(self.add_packet_header(data))
67.         self.client.send(data)
68.
69.     def receive_data_with_header(self) -> bytes:
70.         """Receives the header and handles relevant logic for receiving relevant data
returning the json data"""
71.         data_length = self.client.recv(HEADER).decode(FORMAT)
72.         if data_length != 0:
73.             data_length = int(data_length)
74.             json_data = self.client.recv(data_length)
75.             return json_data
76.
77.     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):
78.         """
79.         Sends data to self.client. Serializes and encrypts it before sending
80.
81.         Parameters:
82.         - data (dict): data you want to send
83.         - Epk (bytes): One time key used for symmetrical encryption
84.         - private_key: SENDERS private key
85.         - public_key: SENDERS public key
86.         - recipient_public_key
87.         - recipient: server or client
```

```
88.         - *recipient_user_id: only needed if data is a message from client to client
89.         """
90.
91.         serialized_data = self.serialize_dict(data)
92.         encrypted_data = self.encrypt(serialized_data, Epk)
93.
94.         encrypted_Epk = rsa.encrypt(Epk, recipient_public_key)
95.
96.         lump_data = {'encrypted_data': encrypted_data,
97.                     'encrypted_Epk': encrypted_Epk,
98.                     'recipient': recipient}
99.
100.        if data['type'] == 'message':
101.            lump_data['type'] = 'message'
102.
103.        if len(recipient_user_id) != 0:
104.            lump_data['recipient_user_id'] = recipient_user_id[0]
105.
106.        serialized_lump_data = self.serialize_dict(lump_data)
107.
108.        signature = self.generate_signature(
109.            serialized_lump_data, private_key, public_key)
110.
111.        serialized_signature = self.serialize_dict(signature)
112.
113.        self.send_with_header(serialized_lump_data)
114.        self.send_with_header(serialized_signature)
115.
116.        if return_message:
117.            return serialized_lump_data
118.
119.        def recieve_encrypted_data(self, private_key: rsa.PrivateKey, return_public_key=False,
120.        return_Epk=False):
121.            """Recieves encrypted data from self.client returning data + others depending on
122.            arguments"""
123.
124.            serialized_lump_data = self.receive_data_with_header()
125.            serialized_signature = self.receive_data_with_header()
126.
127.            if serialized_lump_data != 0 and serialized_signature != 0:
128.                lump_data = self.deserialize_dict(serialized_lump_data)
129.                signature = self.deserialize_dict(serialized_signature)
130.
131.                # If data should NOT be forwarded
132.                if (self.type == 'SERVER' and lump_data['recipient'] == 'server') or (self.type
133.                == 'CLIENT' and lump_data['recipient'] == 'client'):
134.                    if self.validate_signature(serialized_lump_data, signature):
135.                        encrypted_Epk = lump_data['encrypted_Epk']
136.                        Epk = rsa.decrypt(encrypted_Epk, private_key)
137.
138.                        serialized_decrypted_data = self.decrypt(
139.                            lump_data['encrypted_data'], Epk)
140.                        data = self.deserialize_dict(serialized_decrypted_data)
141.
142.                        if self.type == 'SERVER' and data['type'] == 'DISCONNECT':
143.                            raise ClientDisconnectException('Client Disconnected')
144.                        else:
145.                            print("[+] Signature fail / message was empty")
146.
147.                        if return_public_key:
148.                            # data is for either
149.                            return True, data, signature['public_key']
150.                        elif return_Epk:
151.                            return True, data, Epk # data is for either
152.                        else:
153.                            return True, data # data is for either
154.                    else:
155.                        # data is for server to forward to another client
156.                        return False, serialized_lump_data, serialized_signature
157.
158.        def forward_data(self, serialized_data, serialized_signature):
```

```
155.         """Sends data without signature or encryption"""
156.         self.send_with_header(serialized_data)
157.         self.send_with_header(serialized_signature)
158.
159.     def send_data(self, data: dict, private_key: rsa.PrivateKey, public_key:
160.         rsa.PublicKey):
161.         """
162.             Sends data and signature to self.client serialising it before sending
163.
164.             Parameters:
165.             - data: data you want to send
166.             - private_key: SENDERS private key
167.             - public_key: SENDERS public_key
168.         """
169.         serialized_data = self.serialize_dict(data)
170.
171.         signature = self.generate_signature(
172.             serialized_data, private_key, public_key)
173.
174.         serialized_signature = self.serialize_dict(signature)
175.
176.         self.send_with_header(serialized_data)
177.         self.send_with_header(serialized_signature)
178.
179.     def receive_data(self) -> dict:
180.         """Receives data from self.client deserializing it before returning"""
181.         json_data = self.receive_data_with_header()
182.         json_signature = self.receive_data_with_header()
183.
184.         signature = self.deserialize_dict(json_signature)
185.
186.         if json_data != 0 and json_signature != 0 and self.validate_signature(json_data,
187.             signature):
188.             data = self.deserialize_dict(json_data)
189.             if self.type == 'SERVER' and data['type'] == 'DISCONNECT':
190.                 raise ClientDisconnectException('Client Disconnected')
191.             else:
192.                 return data
193.         else:
194.             print("[+] Signature Invalid/Recieved Empty Mesage")
195.
196.     def serialize_dict(self, data: dict) -> bytes:
197.         """Serializes a dictionary and encodes it in a JSON format then encodes it """
198.         obj_mapping = [] # contains the keys in the dict where the value was an object
199.         bytes_mapping = [] # contains the keys in the dict where the value was bytes
200.         for key, value in data.items():
201.             if self.is_user_defined_class(value):
202.                 data[key] = self.serialize_object(value)
203.                 obj_mapping.append(key)
204.             elif isinstance(value, bytes):
205.                 data[key] = self.serialize_bytes(value)
206.                 bytes_mapping.append(key)
207.
208.         # all dicts need 'type' in for sending and recieving purposes in client and server
209.         code
210.         if 'type' not in data:
211.             data['type'] = 'None'
212.
213.         data['obj_mapping'] = obj_mapping
214.         data['bytes_mapping'] = bytes_mapping
215.         data = json.dumps(data)
216.         data = data.encode(FORMAT)
217.         return data
218.
219.     def deserialize_dict(self, data: bytes) -> dict:
220.         """Turns serialised bytes into a dictionary also deserializing any objects or bytes
221.         too"""
222.         data = data.decode(FORMAT)
223.         data = json.loads(data)
```

```
221.         for key, value in data.items():
222.             if key in data['obj_mapping']:
223.                 data[key] = self.deserialize_object(value)
224.             elif key in data['bytes_mapping']:
225.                 data[key] = self.deserialize_bytes(value)
226.         return data
227.
228.     def serialize_bytes(self, data: bytes) -> str:
229.         """Returns bytes encoded to base64"""
230.         b64 = base64.b64encode(data)
231.         return b64.decode(FORMAT)
232.
233.     def deserialize_bytes(self, data: str) -> bytes:
234.         """Returns a string encoded in base64 into bytes"""
235.         b64 = data.encode(FORMAT)
236.         return base64.b64decode(b64)
237.
238.     def serialize_object(self, obj: object) -> str:
239.         """Returns a serialised string of the object using pickle"""
240.         pickled_object = pickle.dumps(obj)
241.         string_of_object = self.serialize_bytes(pickled_object)
242.         return string_of_object
243.
244.     def deserialize_object(self, serialized_object: str) -> object:
245.         """Returns an python object of the serialised_object using pickle"""
246.         pickled_object = self.deserialize_bytes(serialized_object)
247.         unpickled_object = pickle.loads(pickled_object)
248.         return unpickled_object
249.
250.     def is_user_defined_class(self, obj: object) -> bool:
251.         """
252.         Checks if object is an in-built python object or not
253.         """
254.         # 'hacky way' of checking taken from:
255.         # https://stackoverflow.com/questions/14612865/how-to-check-if-object-is-instance-of-new-style-user-defined-class
256.         if hasattr(obj, '__class__'):
257.             return (hasattr(obj, '__dict__') or hasattr(obj, '__slots__'))
258.
```

Server.py

```
1. # socket imports
2. from networkingProtocols import BaseClass
3. from networkingProtocols import ClientDisconnectException
4. import socket
5.
6. # cryptography imports
7. import rsa
8. import secrets
9.
10. # password hashing imports
11. import bcrypt
12. import hmac
13. import hashlib
14. import keyring
15.
16. import threading
17. import serverDatabase
18.
19. HEADER = 2048 # make bigger if needed
20. PORT = 65432 # TCP/UDP packets
21.
22. SERVER = socket.gethostbyname(socket.gethostname())
23. ADDR = (SERVER, PORT)
```

```
24. FORMAT = 'utf-8'
25.
26. active_clients = []
27. logged_in_clients = []
28. server_public_key, server_private_key = rsa.newkeys(512)
29.
30. database = r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level
NEA\OrganisedServerCode\serverDB.db"
31. sql = serverDatabase.Database(database)
32. sql.server_tables()
33.
34. class UserHandler(BaseClass):
35.     def __init__(self, type: str, client, addr):
36.         super().__init__(type, client, addr)
37.
38.         active_clients.append(self)
39.         self.client_public_key = None
40.         self.client_user_id = None
41.         self.can_recieve_msg = False
42.
43.         print(f"\n[NEW CONNECTION] {self.addr[0], self.addr[1]} connected.")
44.         print(f"[TOTAL CONNECTIONS] {len(active_clients)}\n")
45.
46.     def get_name(self):
47.         """Returns ip, port and client userID"""
48.         return f"{self.addr[0], self.addr[1], self.client_user_id}"
49.
50.     def handel_disconnect(self):
51.         """Removes self from list of active clients and closes connection"""
52.         active_clients.remove(self)
53.         self.client.close()
54.         print(
55.             f'[CLIENT DISCONNECTED] {self.addr[0], self.addr[1], self.client_user_id}')
56.         print(f"[TOTAL CONNECTIONS] {len(active_clients)}\n")
57.
58.     # -----SENDING AND RECIEVING DATA-----
59.
60.     def send_data_to_client(self, data: dict):
61.         """Sends data to client autofilling public and private key arguments"""
62.         self.send_data(data,
63.                         server_private_key, server_public_key)
64.
65.     def send_encrypted_data_to_client(self, data: dict):
66.         """Sends encrypted data to client autofilling nessesary arguments"""
67.         Epk = secrets.token_bytes(16) # Epk is unique to each new message
68.
69.         self.send_encrypted_data(
70.             data, Epk, server_private_key, server_public_key, self.client_public_key,
71.             'client')
72.
73.     def recieve_encrypted_data_from_client(self):
74.         """Returns the recieved decrypted data from client autofilling Private Key
75.         argument"""
76.         return self.recieve_encrypted_data(server_private_key)[1]
77.
78.     # -----HANDEL INITIAL CONTACT FUNCTIONS-----
79.
80.     def handel(self):
81.         """Runs the initial functions to handel the first contact between a client and a
82.         server"""
83.         print(f"[HANDELING LOGGED OUT CLIENT {self.get_name()}]")
84.         self.handel_initial_contact()
85.         self.handle_logged_out_client()
86.
87.     def handel_initial_contact(self):
88.         self.swap_public_keys()
89.
90.     def swap_public_keys(self):
91.         """Swaps public keys with the client setting self.client_public_key in the
92.         process"""
```

```
89.         print(
90.             f"[PUBLIC KEY SWAP] Swapping public keys with {self.get_name()}")
91.         self.client_public_key = self.receive_data()['public_key']
92.
93.         self.send_data_to_client(
94.             {'recipient': 'client', 'public_key': server_public_key})
95.
96.         print(f"[PUBLIC KEY SWAP FINISHED] with {self.get_name()}\n")
97.
98.         # -----HANDEL LOGGED OUT CLIENT FUNCTIONS-----
99.
100.        def handle_logged_out_client(self):
101.            """Handels logged out client by waiting to recieve a determiner specifying what the
102.            user is trying to do and calling functions appropriately"""
103.            login = False
104.
105.            while login != True:
106.                determiner = self.receive_data()
107.                if determiner['type'] == 'login request':
108.                    if self.handel_login():
109.                        login = True
110.                    elif determiner['type'] == 'create account request':
111.                        if self.handel_create_account():
112.                            login = True
113.            self.handle_logged_in_client()
114.
115.        def handel_login(self) -> bool:
116.            """Handels clients login attempt"""
117.            login_details = self.recieve_encrypted_data_from_client()
118.            user_exists = sql.check_user_id_exists(
119.                login_details['user_id'])
120.
121.            if not user_exists: # if user does not exist
122.                valid_password = False
123.            elif self.is_user_already_online(login_details['user_id']):
124.                valid_password = False
125.            else:
126.                valid_password = self.validate_login_info(
127.                    login_details['user_id'], login_details['password'])
128.
129.            if valid_password:
130.                self.client_user_id = login_details['user_id']
131.
132.            self.send_encrypted_data_to_client(
133.                {'recipient': 'CLIENT', 'valid_password': valid_password})
134.            return valid_password
135.
136.        def handel_create_account(self):
137.            """Handels clients create account attempt"""
138.            account_created = False
139.
140.            details = self.recieve_encrypted_data_from_client()
141.
142.            user_id_already_used = sql.check_user_id_exists(details['user_id'])
143.
144.            self.send_encrypted_data_to_client(
145.                {'recipient': 'CLIENT', 'user_id_already_used': user_id_already_used})
146.
147.            if not user_id_already_used: # if user id not taken
148.                # Creating account
149.                try:
150.                    user_account_details = self.recieve_encrypted_data_from_client()
151.                    account_created = self.store_user_login_details(
152.                        user_account_details['user_id'], user_account_details['screen_name'],
153.                        user_account_details['password'])
154.                except Exception as e:
155.                    print(e)
156.                finally:
157.                    self.send_encrypted_data_to_client(
158.                        {'recipient': 'CLIENT', 'account created': account_created})
```

```
157.         self.client_user_id = user_account_details['user_id']
158.     print(
159.         f'[ACCOUNT CREATED FOR] for {self.get_name()}'
160.     )
161.     return account_created
162.
163. def is_user_already_online(self, user_id):
164.     for user in active_clients:
165.         if user.client_user_id == user_id:
166.             return True
167.     return False
168.
169. def store_user_login_details(self, user_id, screen_name, password):
170.     """Adds users id, screen name, hashed password, password salt and public key to the
171.     servers database"""
172.     salt = bcrypt.gensalt() # salt generated outside function as it needs to be stored
173.     peppered_hash = self.hash_password(password, salt)
174.     serialized_public_key = self.serialize_object(
175.         self.client_public_key) # sql cant store python objects
176.
177.     return sql.add_user(user_id, screen_name, peppered_hash,
178.                         salt, seralized_public_key)
179.
180. def hash_password(self, password: bytes, salt: bytes) -> str:
181.     """Hashes and salts peppers password as per OWASP guidelines 2024"""
182.     password = password.encode()
183.     hash = bcrypt.hashpw(password, salt)
184.     temp = hmac.new(self.get_pepper(), hash, hashlib.sha256)
185.     peppered_hash = temp.hexdigest()
186.     return peppered_hash
187.
188. def validate_login_info(self, user_id: str, password: str):
189.     """Checks recieved password against stored hash and returns relevant bool"""
190.     print(f'[VALIDATING LOGIN FOR {self.get_name()}]')
191.     valid_password = False
192.     data = sql.get_password(user_id)
193.     stored_password = data[0][0]
194.     salt = data[0][1]
195.     if self.hash_password(password, salt) == stored_password:
196.         valid_password = True
197.     return valid_password
198.
199. def get_pepper(self):
200.     """Gets stored pepper from Windows Key Store"""
201.     return keyring.get_password("a_level_nea", "oran").encode()
202.
203. # -----HANDEL LOGGED IN CLIENT FUNCTIONS-----
204.
205. def handle_logged_in_client(self):
206.     print(f'[HANDLING LOGGED IN USER {self.get_name()}]')
207.     # second swap required to get the 'real' keys rather than the temp keys
208.     self.swap_public_keys()
209.
210.     screen_name = sql.get_screen_name(self.client_user_id)
211.     self.send_data_to_client(
212.         {'recipient': 'CLIENT', 'screen_name': screen_name})
213.
214.     self.recieve_data_from_logged_in_user()
215.
216. def recieve_data_from_logged_in_user(self):
217.     """Recieved encrypted data from logged in user and handels it accordingly"""
218.
219.     # recieved_data = (for_server: bool, seralized_lump_data: dict,
220.     # seralized_signature: dict)
221.     connected = True
222.     while connected:
223.         recieved_data = self.recieve_encrypted_data(server_private_key)
224.         if recieved_data[0]: # data is for the server
225.             print(f'[RECEIVED DATA FROM {self.get_name()} FOR SERVER]')
226.             data = recieved_data[1]
227.             print(f"{data = }\n\n")
```

```
225.
226.         if data['type'] == 'check_if_friend_code_exists':
227.             self.send_encrypted_data_to_client(
228.                 {'exist': sql.check_user_id_exists(data['friend_code'])})
229.
230.         elif data['type'] == 'get_recipient_public_key':
231.             serialized_public_key = sql.get_public_key(
232.                 data['recipient_user_id'])
233.             self.send_encrypted_data_to_client(
234.                 {'recipient_public_key': serialized_public_key})
235.
236.         elif data['type'] == 'get_friend_detials':
237.             self.send_encrypted_data_to_client(
238.                 {
239.                     'screen_name': sql.get_screen_name(data['friend_user_id']),
240.                     'public_key': sql.get_public_key(data['friend_user_id'])
241.                 })
242.         elif data['type'] == 'request_all_user_data':
243.             user_details = sql.get_user_details(self.client_user_id)
244.             self.send_encrypted_data_to_client(
245.                 {
246.                     'user_details': user_details
247.                 }
248.             )
249.         elif data['type'] == 'deleting_account':
250.             account_deleted, account_deletion_name = sql.delete_account_server(
251.                 self.client_user_id)
252.             self.send_encrypted_data_to_client(
253.                 {
254.                     'account_deleted': account_deleted,
255.                     'account_deletion_name': account_deletion_name
256.                 }
257.             )
258.
259.         elif data['type'] == 'change_screen_name':
260.             new_screen_name = data['new_screen_name']
261.             print('UPDATING SCREEN NAME')
262.             sql.update_screen_name_server(
263.                 self.client_user_id, new_screen_name)
264.
265.         elif data['type'] == 'can_recieve_msg_value':
266.             self.can_recieve_msg = data['can_recieve_msg']
267.             if self.can_recieve_msg:
268.                 self.recieved_message_queue()
269.
270.         else: # lump data is for the client!
271.             print(f"[RECIEVED DATA FROM {self.get_name()} TO FORWARD]")
272.             self.forward_data_to_client(recieved_data)
273.
274.     def forward_data_to_client(self, recieved_data):
275.         """Attempts to forward data to intended recipient. If it can't adds to the message
276.         queue instead"""
277.         recipient_found = False
278.         serialized_lump_data = recieved_data[1]
279.         serialized_signature = recieved_data[2]
280.         recipient_user_id = self.deserialize_dict(serialized_lump_data)[
281.             'recipient_user_id']
282.
283.         for client in active_clients:
284.             if client.client_user_id == recipient_user_id and client.can_recieve_msg ==
285.             True:
286.                 from_message_queue = self.get_name() == client.get_name()
287.                 print(
288.                     f"[FORWARDING DATA FROM {self.get_name()} TO {client.get_name()}] FROM
289.                     MESSAGE QUEUE {from_message_queue}")
290.                 client.forward_data(serialized_lump_data, serialized_signature)
291.                 recipient_found = True
292.                 break
293.
294.         if not recipient_found:
```



```
292.         # adding recipient_user_id as it means deserialization is not needed for each
item when searching queue
293.         message_queue.enqueue(recipient_user_id, recieved_data)
294.
295.     def recieved_message_queue(self):
296.         """Sends messages waiting in message_queue to relevant client"""
297.         for message in message_queue.get().copy():
298.             if message[0] == self.client_user_id:
299.                 message_queue.dequeue(self.client_user_id, message[1])
300.                 self.forward_data_to_client(message[1])
301.
302. class MessageQueue():
303.     def __init__(self):
304.         self.__items = []
305.
306.     def isEmpty(self):
307.         return len(self.__items) == 0
308.
309.     def enqueue(self, client_recieving_user_id: str, message: list):
310.         values = (client_recieving_user_id, message)
311.         self.__items.append(values)
312.
313.     def dequeue(self, client_recieving_user_id: str, message: list):
314.         if not self.isEmpty():
315.             values = (client_recieving_user_id, message)
316.             try:
317.                 self.__items.remove(values)
318.             except Exception as e:
319.                 print(f"[deQueue ERROR] {e}")
320.
321.     def get(self):
322.         return self.__items
323.
324. def worker(client, addr):
325.     # allows disconnect excpetion to be handeled within each new users thread rather than
having to leave the thread
326.     new_user = UserHandler('SERVER', client, addr)
327.     try:
328.         new_user.handel()
329.     except ClientDisconnectException:
330.         new_user.handel_disconnect()
331.
332. # -----DRIVER CODE-----
333.
334. def main():
335.     # create new socket object
336.     server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
337.     try:
338.         # bind socket to port and IP
339.         server.bind(ADDR)
340.     except:
341.         print(f"Unable to bind to server {SERVER} and port {PORT}")
342.     server.listen()
343.     print(f"[LISTENING] Server is listening on {SERVER} {PORT}")
344.     while True:
345.         client, addr = server.accept() # waits till new connection
346.         thread = threading.Thread(target=worker, args=(client, addr))
347.         thread.start()
348.
349. message_queue = MessageQueue()
350. main()
351.
352.
```

Client.py

```
1. # Cryptography imports
2. import secrets
```

```
3. import rsa
4.
5. # socket imports
6. from networkingProtocols import BaseClass
7. import socket
8.
9. # file handling and database imports
10. from tkinter import filedialog
11. import serverDatabase
12. from PIL import Image
13. import os
14.
15. # hashing imports
16. from hashlib import md5 # used in secret keeping NOT for password hashing
17.
18. import threading
19.
20. HEADER = 2048 # used for send message protocol
21. PORT = 65432 # TCP/UDP packets
22. FORMAT = 'utf-8'
23. SERVER = "192.168.0.30"
24. ADDR = (SERVER, PORT)
25. CLIENT = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
26.
27. class Client(BaseClass):
28.     def __init__(self, cs='CLIENT', client=CLIENT, addr=ADDR):
29.         super().__init__(cs, client, addr)
30.         self.client = client
31.         self.sql = None
32.
33.         self.server_public_key = None
34.         self.public_key = None
35.         self.private_key = None
36.         self.master_key = None
37.         self.password_hash = None # used for locally storing sensitive data
38.
39.         self.user_path = None
40.         self.image_path = None
41.         self.image_name_and_format = None
42.         self.user_id = ''
43.         self.screen_name = ''
44.
45.         self.friend_list = None
46.         self.friend_request_list = None
47.         self.pending_friend_list = None
48.         self.current_message_history = []
49.
50.         self.connected = False
51.         self.logged_in = False
52.
53.         self.can_recieve_msg = False
54.         self.listen_thread = None
55.         self.stop_event = threading.Event()
56.
57.         self.ChatPage = None
58.         self.AddFriendPage = None
59.
60.         print('[CLIENT INSTANCE CREATED]')
61.
62.     def connect(self) -> bool:
63.         """
64.         Attempts to connect to the server specified in the ADDR variable.
65.
66.         Returns:
67.             - True if connection successful
68.             - False if not
69.         """
70.
71.         print("[ATTEMPTING TO CONNECT TO SERVER]")
72.         try:
```

```
73.         self.client.connect(ADDR)
74.         self.connected = True
75.     except Exception as e:
76.         print(f"[ERROR] Client was unable to connect to server: {ADDR}")
77.         self.connected = False
78.     print(f"[CONNECTED] {self.connected}")
79.     return self.connected
80.
81.     def establish_initital_contact(self):
82.         """establishes initial contact with the server by generating public and private
keys and swapping them"""
83.         print(f"[establish_initital_contact STARTED]")
84.         self.generate_init_keys()
85.         self.swap_public_keys()
86.         print(f"[establish_initital_contact FINISHED]")
87.
88.     def send_disconnect_message(self):
89.         print("[SENDING DISCONNECT MESSAGE]")
90.         if self.logged_in:
91.             self.send_encrypted_data_to_server({'type': 'DISCONNECT'})
92.         else:
93.             self.send_data_to_server({'type': 'DISCONNECT'})
94.         print("[DISCONNECTED FROM SERVER]")
95.
96.     def close_client(self):
97.         print('[CLOSING CLIENT]')
98.         self.client.close()
99.
100.    def connect_to_database(self):
101.        """Creates and or connects to database in self.user_path"""
102.        db_path = os.path.join(self.user_path, 'user_data.db')
103.        self.sql = serverDatabase.Database(db_path)
104.
105.    def close_db_connection(self):
106.        if self.logged_in:
107.            print('[CLOSING DATABASE]')
108.            self.sql.close_connection()
109.
110.    def send_data_to_server(self, data):
111.        """Sends data to server autofilling public and private key parameter"""
112.        self.send_data(data,
113.                        self.private_key, self.public_key)
114.
115.    def send_encrypted_data_to_server(self, data: dict):
116.        """Creates Epk and sends encrypted data to server"""
117.        print(f"[SENDING ENCRYPTED DATA TO SERVER] {data}")
118.        Epk = secrets.token_bytes(16)
119.
120.        self.send_encrypted_data(
121.            data, Epk, self.private_key, self.public_key, self.server_public_key, 'server')
122.
123.    def recieve_encrypted_data_from_server(self):
124.        """returns decrypted recieved data from server autofilling private key parameter"""
125.        return self.recieve_encrypted_data(self.private_key)[1]
126.
127.    def send_encrypted_data_to_recipient(self, data, recipient_user_id,
return_config_data=False):
128.        """Creates Epk, gets recipients private key and sends data to server to forward to
recipient"""
129.        print(f"[SENDING ENCRYPTED DATA TO {recipient_user_id}] {data}")
130.        Epk = secrets.token_bytes(16)
131.
132.        data['sender'] = self.user_id
133.        data['public_key'] = self.public_key
134.
135.        # getting recipient public key from server
136.        self.send_encrypted_data_to_server(
137.            {'type': 'get_recipient_public_key',
138.             'recipient_user_id': recipient_user_id}
139.        )
```

```
140.         serialised_recipinet_public_key = self.recieve_encrypted_data_from_server()[
141.             'recipient_public_key']
142.         recipinet_public_key = self.deserialize_object(
143.             serialised_recipinet_public_key)
144.
145.         encrypted_message = self.send_encrypted_data(
146.             data, Epk, self.private_key, self.public_key, recipinet_public_key, 'client',
return_config_data, recipient_user_id)
147.
148.         if return_config_data:
149.             return Epk
150.
151.     # -----CREATE/LOGIN/DELETE ACCOUNT -----
152.
153.     def login(self, user_id: str, password: str):
154.         print("[ATTEMPTING TO LOGIN]")
155.         login = False
156.
157.         self.send_data_to_server({'type': 'login request'})
158.         self.send_encrypted_data_to_server(
159.             {'user_id': user_id, 'password': password})
160.
161.         if self.recieve_encrypted_data_from_server()['valid_password']:
162.             login = True
163.             self.user_id = user_id
164.             self.password_hash = md5(password.encode()).digest()
165.             self.user_path = os.path.join('App./Users./', user_id)
166.             self.user_images_path = os.path.join(self.user_path, 'images')
167.             self.connect_to_database()
168.             self.get_keys_from_file()
169.             self.swap_public_keys()
170.             print(f"[LOGIN {login}]")
171.             return login
172.
173.     def create_account(self, user_id: str, password: str, screen_name: str):
174.         print(f"[HANDELING create_account STARTED]")
175.         print(f"[VALIDATING ACCOUNT CREATION DETAILS]")
176.
177.         valid_user_id = False
178.         account_created = False
179.
180.         self.send_data_to_server({'type': 'create account request'})
181.         self.send_encrypted_data_to_server(
182.             {'user_id': user_id, 'recipient': 'SERVER'})
183.
184.         if not self.recieve_encrypted_data_from_server()['user_id_already_used']:
185.             valid_user_id = True
186.             # user ID is unique therefore try to create account then send details to server
187.             self.password_hash = md5(password.encode()).digest()
188.
189.             self.user_path = os.path.join('App./Users./', user_id)
190.             self.user_images_path = os.path.join(self.user_path, 'images')
191.             try: # if new user
192.                 os.makedirs(self.user_images_path)
193.                 self.write_keys_to_file()
194.                 self.connect_to_database()
195.                 self.sql.client_tables()
196.
197.                 self.send_encrypted_data_to_server(
198.                     {'user_id': user_id, 'password': password, 'screen_name': screen_name})
199.
200.                 if self.recieve_encrypted_data_from_server()['account created']:
201.                     account_created = True
202.             except: # if user already exists locally - should never occur
203.                 print(f"[ERROR] User: {user_id} already exists")
204.                 valid_user_id = False
205.
206.         print(f"[HANDELING create_account FINISHED]")
207.         if valid_user_id and account_created:
208.             self.user_id = user_id
```

```
209.         self.swap_public_keys()
210.         return valid_user_id, account_created
211.
212.     def delete_account(self):
213.         """
214.         Sends account deletion notification to server and all friends
215.
216.         Returns if it was successful or not
217.         """
218.         account_deleted = False
219.         self.send_encrypted_data_to_server({'type': 'deleting_account'})
220.         details = self.receive_encrypted_data_from_server()
221.         if details['account_deleted']:
222.
223.             account_deleted = True
224.             friend_user_ids = self.sql.get_all_acc_friends_user_ids()
225.             flat_friend_user_id = [
226.                 id for tuple in friend_user_ids for id in tuple]
227.             for friend_id in flat_friend_user_id:
228.                 self.send_encrypted_data_to_recipient(
229.                     {'type': 'sync_account_deletion', 'account_deletion_name':
230.                     details['account_deletion_name']}, friend_id)
231.             return account_deleted
232.
233.     def delete_directory(self):
234.         """Deletes everything in the users account directory and then the directory
235.         itself"""
236.         print("[DELETING ACCOUNT]")
237.         for filename in os.listdir(self.user_path):
238.             if os.path.isfile(os.path.join(self.user_path, filename)):
239.                 os.remove(os.path.join(self.user_path, filename))
240.         for filename in os.listdir(self.user_images_path):
241.             if os.path.isfile(os.path.join(self.user_images_path, filename)):
242.                 os.remove(os.path.join(self.user_images_path, filename))
243.         os.rmdir(self.user_images_path)
244.         os.rmdir(self.user_path)
245.
246.         # -----PUBLIC AND PRIVATE KEYS-----
247.
248.     def generate_init_keys(self):
249.         """Generates the clients public, private and master keys"""
250.         self.public_key, self.private_key = rsa.newkeys(512)
251.         self.master_key = secrets.token_bytes(16)
252.
253.     def write_keys_to_file(self):
254.         """Writes clients public, private and master keys to local file in
255.         self.user_path"""
256.
257.         # Encrypting and serializing keys
258.         serialized_private_key = self.serialize_object(
259.             self.private_key).encode()
260.         key_data = {
261.             'public_key': self.public_key,
262.             'private_key': self.encrypt(serialized_private_key, self.password_hash),
263.             'master_key': self.encrypt(self.master_key, self.password_hash)
264.         }
265.         serialized_key_data = self.serialize_dict(key_data)
266.
267.         with open(os.path.join(self.user_path, 'keys.txt'), 'wb') as keys:
268.             keys.write(serialized_key_data)
269.
270.     def get_keys_from_file(self):
271.         """Retrieves clients public, private and master keys from local file in
272.         self.user_path"""
273.
274.         # Reading data from file
275.         with open(os.path.join(self.user_path, 'keys.txt'), 'rb') as keys:
276.             serialized_key_data = keys.read()
277.
278.         # Decrypting and deserializing keys
```

```
275.         key_data = self.deserialize_dict(serialized_key_data)
276.         self.public_key = key_data['public_key']
277.         self.master_key = self.decrypt(
278.             key_data['master_key'], self.password_hash)
279.         serialized_private_key = self.decrypt(
280.             key_data['private_key'], self.password_hash).decode()
281.         self.private_key = self.deserialize_object(serialized_private_key)
282.
283.     def swap_public_keys(self):
284.         """sends public key to server and recieves servers public key"""
285.         self.send_data_to_server(
286.             {'recipient': 'server', 'public_key': self.public_key})
287.         self.server_public_key = self.receive_data()['public_key']
288.
289.     # -----GETTING FRIENDS LISTS-----
290.
291.     def get_friend_list(self):
292.         """Sets self.friend_list to equal users friend list"""
293.         self.friend_list = self.sql.get_friend_list()
294.
295.     def get_friend_request_list(self):
296.         """Sets self.friend_request_list = list of incoming friend requests"""
297.         self.friend_request_list = self.sql.get_friend_request_list(
298.             self.user_id)
299.
300.     def get_pending_friends_list(self):
301.         """Sets self.pending_friend_list = list of outgoing friend requests"""
302.         self.pending_friend_list = self.sql.get_pending_friends_list(
303.             self.user_id)
304.         print(self.pending_friend_list)
305.
306.     # -----LISTEN FOR MESSAGES-----
307.
308.     def handel_logged_in_client(self):
309.         self.logged_in = True
310.         self.screen_name = self.receive_data()['screen_name']
311.
312.     def listen(self):
313.         """Send message to server saying it can listen and starts the listening thread"""
314.         self.can_recieve_msg = True
315.         self.send_encrypted_data_to_server(
316.             {'type': 'can_recieve_msg_value', 'can_recieve_msg': self.can_recieve_msg})
317.
318.         self.stop_event.clear()
319.         self.listen_thread = threading.Thread(
320.             target=self.recieving_data_from_client)
321.         self.listen_thread.start()
322.
323.     def stop_listen(self):
324.         """Send message to server saying it can't listen and stops the listening thread"""
325.         try:
326.             self.can_recieve_msg = False
327.             self.send_encrypted_data_to_server(
328.                 {'type': 'can_recieve_msg_value', 'can_recieve_msg': self.can_recieve_msg})
329.
330.             self.stop_event.set()
331.             self.listen_thread.join()
332.             print('STOP Listen')
333.         except Exception as e:
334.             print(e)
335.             pass
336.
337.     def recieving_data_from_client(self):
338.         while not self.stop_event.is_set():
339.             try:
340.                 self.client.settimeout(2)
341.                 # 2 will also be the delay time when stopping the thread
342.                 total_data = self.recieve_encrypted_data(
343.                     self.private_key, False, True)
344.                 recieved_data = total_data[1]
```

```
345.         Epk = total_data[2]
346.         if recieved_data['type'] == 'friend_request':
347.             seralized_public_key = self.serialize_object(
348.                 recieved_data['public_key'])
349.             self.sql.add_new_friend_request(
350.                 recieved_data['sender'], recieved_data['screen_name'],
351.                 seralized_public_key, recieved_data['sender'])
352.             self.get_friend_request_list()
353.         elif recieved_data['type'] == 'accepted_friend_request':
354.             self.sql.accept_friend_request(
355.                 recieved_data['sender'], recieved_data['sender'])
356.             self.get_friend_list()
357.             self.ChatPage.update_friend_list()
358.         elif recieved_data['type'] == 'rejected_friend_request':
359.             self.sql.reject_friend_request(recieved_data['sender'])
360.             self.get_friend_list()
361.         elif recieved_data['type'] == 'blocked':
362.             self.sql.new_blocked_friend(
363.                 recieved_data['sender'], recieved_data['sender'])
364.             self.get_friend_list()
365.             self.ChatPage.update_friend_list()
366.         elif recieved_data['type'] == 'unblocked':
367.             self.sql.unblocked_friend(
368.                 recieved_data['sender'], recieved_data['sender'])
369.             self.get_friend_list()
370.             self.ChatPage.update_friend_list()
371.         elif recieved_data['type'] == 'sync_new_screen_name':
372.             new_friends_screen_name = recieved_data['new_screen_name']
373.             friend_id = recieved_data['sender']
374.             self.sql.update_friend_screen_name(
375.                 friend_id, new_friends_screen_name)
376.         elif recieved_data['type'] == 'sync_account_deletion':
377.             account_deletion_name = recieved_data['account_deletion_name']
378.             friend_id = recieved_data['sender']
379.             self.sql.friend_deleted_account(
380.                 friend_id, account_deletion_name)
381.         elif recieved_data['type'] == 'message':
382.             self.handel_recieved_message(recieved_data, Epk)
383.     except socket.timeout:
384.         continue
385.
386.     # -----SENDING/RECIEVING MESSAGES-----
387.
388.     def handel_send_message(self, data: dict):
389.         """Sends message to server and then stores it"""
390.         print("[SENDING MESSAGE]")
391.         self.stop_listen()
392.         Epk = self.send_encrypted_data_to_recipient(
393.             data, data['recipient'], True)
394.         message = data['message']
395.         is_image = 0
396.         if data['is_image']:
397.             message = self.store_sent_image_to_files(self.image_path)
398.             is_image = 1
399.
400.         self.store_sent_message(
401.             data, Epk, self.encrypt(message.encode(), Epk), is_image)
402.         self.listen()
403.
404.     def handel_recieved_message(self, data: dict, Epk: bytes):
405.         """Handels recieved message accordingly if it is text or an image"""
406.         message = data['message']
407.
408.         if data['is_image']:
409.             image_data = data['message']
410.             image_name_and_format = data['image_name_and_format']
411.             message = self.store_image_to_files(
412.                 image_name_and_format, image_data)
413.             self.image_path = message
```



```
414.
415.     message = message.encode()
416.     encrypted_message = self.encrypt(message, Epk)
417.     self.store_recieved_message(data, Epk, encrypted_message)
418.
419.     # message_details = (x, x, x ,x from_me, is_image)
420.     message_details = (
421.         Epk, encrypted_message, data['date'], data['time'], 0, data['is_image'])
422.     message = self.decrypt_message(message_details, False)
423.
424.     # if chat related recieved message is open display it
425.     if data['sender'] == self.ChatPage.active_chat_user_details.get().split(' ')[0]:
426.         formatted_message = self.ChatPage.format_stored_message_for_display(
427.             message)
428.         self.ChatPage.display_message(formatted_message)
429.
430.     def decrypt_message(self, message_details, Epk_encrypted=True):
431.         """Returns tuple (decrypted_message, date, time, from_me, is_image)"""
432.         Epk = message_details[0]
433.         encrypted_message = message_details[1]
434.         date = message_details[2]
435.         time = message_details[3]
436.         from_me = message_details[4]
437.         is_image = message_details[5]
438.
439.         if Epk_encrypted:
440.             Epk = self.decrypt(Epk, self.master_key)
441.
442.         decrypted_message = self.decrypt(encrypted_message, Epk).decode()
443.         return (decrypted_message, date, time, from_me, is_image)
444.
445.     def store_sent_message(self, data: dict, Epk: bytes, encrypted_message: bytes,
446. is_image: int):
447.         """Stores sent message encrypting the Epk with self.master_key"""
448.         encrypted_Epk = self.encrypt(Epk, self.master_key)
449.         self.sql.store_message(
450.             data['recipient'], encrypted_Epk, encrypted_message, data['date'],
451. data['time'], 1, is_image)
452.
453.     def store_recieved_message(self, data: dict, Epk: bytes, encrypted_message: bytes):
454.         """Stores recieved message encrypting the Epk with self.master_key"""
455.         encrypted_Epk = self.encrypt(Epk, self.master_key)
456.         self.sql.store_message(
457.             data['sender'], encrypted_Epk, encrypted_message, data['date'], data['time'],
458. 0, data['is_image'])
459.
460.     def get_message_history(self, friend_id: str):
461.         return self.sql.get_message_list(friend_id)
462.
463.     def decrypt_message_history(self, friend_id: str):
464.         self.current_message_history = []
465.         encrypted_message_history = self.get_message_history(friend_id)
466.
467.         for message_details in encrypted_message_history:
468.             values = self.decrypt_message(message_details)
469.             self.current_message_history.append(values[:])
470.
471.     # -----Add Friend Page fuctions-----
472.
473.     def block_friend(self, friend_user_id):
474.         self.sql.new_blocked_friend(self.user_id, friend_user_id)
475.         self.send_encrypted_data_to_recipient(
476.             {'type': 'blocked'}, friend_user_id)
477.
478.     def unblock_friend(self, friend_user_id):
479.         self.sql.unblocked_friend(self.user_id, friend_user_id)
480.         self.send_encrypted_data_to_recipient(
481.             {'type': 'unblocked'}, friend_user_id)
482.
483.     def accept_friend_request(self, friend_id: str, specifier_id: str):
```



```
481.         self.sql.accept_friend_request(friend_id, specifier_id)
482.         self.send_encrypted_data_to_recipient(
483.             {'type': 'accepted_friend_request',
484.              'friend_id': self.user_id
485.             }, friend_id)
486.
487.     def reject_friend_request(self, friend_id: str):
488.         self.sql.reject_friend_request(friend_id)
489.         self.send_encrypted_data_to_recipient(
490.             {'type': 'rejected_friend_request',
491.              'friend_id': self.user_id
492.             }, friend_id)
493.
494.     def send_friend_request(self, friend_user_id: str):
495.         self.send_encrypted_data_to_recipient(
496.             {'type': 'friend_request',
497.              'screen_name': self.screen_name
498.             }, friend_user_id)
499.         self.store_friend_request(friend_user_id)
500.         self.AddFriendPage.update_pending_friends_list(friend_user_id)
501.
502.     def store_friend_request(self, friend_user_id: str):
503.         """Gets friend details from server for an accepted friend request and stores
504.         them"""
505.         self.send_encrypted_data_to_server(
506.             {'type': 'get_friend_details', 'friend_user_id': friend_user_id})
507.         details = self.receive_encrypted_data_from_server()
508.         self.sql.add_new_friend_request(
509.             friend_user_id, details['screen_name'], details['public_key'], self.user_id)
510.
511.     def check_if_user_is_already_friends(self, friend_user_id: str):
512.         already_friends = True
513.         friend_count = self.sql.check_if_user_is_already_friends(
514.             friend_user_id)
515.         if friend_count == 0:
516.             already_friends = False
517.         return already_friends
518.
519.     def check_if_friend_code_exists(self, friend_user_id: str) -> bool:
520.         self.send_encrypted_data_to_server(
521.             {'type': 'check_if_friend_code_exists', 'friend_code': friend_user_id})
522.         return self.receive_encrypted_data_from_server()['exist']
523.
524.     # -----SENDING IMAGES-----
525.
526.     def get_image_data(self, image_path: str):
527.         """Returns binary data of image at image_path"""
528.         image_file = open(image_path, 'rb')
529.         return image_file.read()
530.
531.     def store_sent_image_to_files(self, image_path: str):
532.         """
533.         Gets sent image sent image path and then stores it
534.
535.         Returns stored image path
536.         """
537.         image_name_and_format = image_path.split('/')[-1]
538.         image_file = open(image_path, 'rb')
539.         image_data = image_file.read()
540.         return self.store_image_to_files(image_name_and_format, image_data)
541.
542.     def store_image_to_files(self, image_name_and_format, image_data):
543.         """Stores image to self.user_images_path"""
544.         new_image_path = self.find_sutable_image_path(image_name_and_format)
545.         new_image_file = open(new_image_path, 'wb')
546.         new_image_file.write(image_data)
547.         return new_image_path
548.
549.
```

```
550.     def find_sutable_image_path(self, image_name_and_format: str):
551.         """
552.         Finds suitable image name by adding (i) if there are duplicates
553.
554.         Returns full image path
555.         """
556.         counter = 0
557.         image_path = os.path.join(self.user_images_path, image_name_and_format)
558.         path_name, extention = os.path.splitext(image_path)
559.
560.         base_path = f"{path_name}({counter}){extention}"
561.         while os.path.isfile(base_path):
562.             counter += 1
563.             base_path = f"{path_name}({counter}){extention}"
564.         return base_path
565.
566.     def compress_image(self, image_path: str):
567.         """
568.         Compresses an image to:
569.             - max 100px,100px
570.             - LANCZOS resampling
571.             - save quality: 65%
572.
573.         Returns new image path
574.         """
575.         old_path, extension = os.path.splitext(image_path)
576.         file_name = old_path.split('/')[-1]
577.         split_path = image_path.split('/')[-1]
578.         new_path =
579.         f"{os.path.join(split_path)}/{file_name}(downsized_for_encryption){extension}"
580.         image = Image.open(image_path)
581.         image.thumbnail((100, 100), Image.Resampling.LANCZOS)
582.         image.save(new_path, optimize=True, quality=65)
583.         return new_path
584.
585.     def client_get_image_path(self):
586.         """
587.         Displays popup asking user to select an image file
588.
589.         Returns selected image path
590.         """
591.
592.         image_path = filedialog.askopenfilename(initialdir="/downloads",
593.                                                  title="Select Image",
594.                                                  filetypes=(("jpeg files", "*.jpeg"), ("png
595. files", "*.png")))
596.         if not image_path: # if user cancels on file explorer
597.             return False
598.
599.         self.image_path = self.compress_image(image_path)
600.         self.image_name_and_format = self.image_path.split('/')[-1]
601.
602.         return self.image_path
603.
604. # -----ACCOUNT MODIFICATION-----
605.
606.     def change_screen_name(self, new_screen_name):
607.         """Notifys the server of screen name change"""
608.         self.screen_name = new_screen_name
609.         self.send_encrypted_data_to_server(
610.             {'type': 'change_screen_name', 'new_screen_name': self.screen_name})
611.
612.         friend_user_ids = self.sql.get_all_acc_friends_user_ids()
613.
614.         # flatten list
615.         flat_friend_user_id = [id for tuple in friend_user_ids for id in tuple]
616.         for friend_id in flat_friend_user_id:
617.             self.send_encrypted_data_to_recipient(
```

```
617.         {'type': 'sync_new_screen_name', 'new_screen_name': self.screen_name},
friend_id)
618.
619.     def request_all_user_data(self):
620.         """Returns all data server has on user"""
621.         self.send_encrypted_data_to_server({'type': 'request_all_user_data'})
622.         user_details = self.receive_encrypted_data_from_server()[
623.             'user_details']
624.         return f"user_id:
{user_details[0]}\nscreen_name{user_details[1]}\nserialized_public_key: {user_details[2]}"
625.
626.     def get_output_path(self):
627.         """Returns chosen path to save user data"""
628.         output_path = filedialog.askdirectory(
629.             title="Select place to save data")
630.         return output_path
631.
632.
```

GUI OOP Approach Version 1

```
1. from pathlib import Path
2. from random import randint
3.
4. # from tkinter import *
5. # Explicit imports to satisfy Flake8
6. # from tkinter import *
7. import tkinter as tk
8. from tkinter.ttk import *
9. from tkinter import ttk
10. from tkinter import messagebox
11. from datetime import datetime
12. # import ttkbootstrap as tb
13. # from ttkbootstrap.constants import *
14. import sqlite3
15. from sqlite3 import Error
16. from PIL import Image, ImageTk
17. import socket
18. import threading
19. import json
20. import ast
21.
22. OUTPUT_PATH = Path(__file__).parent
23. ASSETS_PATH = OUTPUT_PATH / \
24.     Path(r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level
NEA\build\assets\frame0")
25.
26. def relative_to_assets(path: str) -> Path:
27.     return ASSETS_PATH / Path(path)
28.
29. HEADER = 64 # used for send message protocol
30. PORT = 65432 # TCP/UDP packets
31. FORMAT = 'utf-8'
32. SERVER = "192.168.0.30"
33. ADDR = (SERVER, PORT)
34.
35. class Client:
36.     def __init__(self):
37.         self.user_id = None
38.         self.public_key = None
39.         self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
40.         self.connection_failed = False
41.         self.ChatPage = None
42.         self.screen_name = ''
43.         self.friends = None
44.
45.     def set_chat_page(self, ChatPage):
46.         self.ChatPage = ChatPage
```

```
47.
48. def start(self):
49.     self.connect()
50.     # self.send_first_contact()
51.
52.     """LISTEN THREAD TO BE CALLED ACTIVE ONCE LOGIN OCCURS!
53.     DEACTIVATE WHEN SERVER NEEDS TO SEND OTHER DATA"""
54.
55. def start_listen_thread(self):
56.     thread_listen = threading.Thread(target=self.listen)
57.     thread_listen.start()
58.
59. def connect(self):
60.     try:
61.         self.client.connect(ADDR)
62.     except Exception as e:
63.         print(f"[ERROR] Client was unable to connect to server: {ADDR}")
64.         self.connection_failed = True
65.
66. def validate_login_info(self, username, password):
67.     print('[FUNCTION RUN] validate_login_info')
68.     user_data = {
69.         'type': 'null',
70.         'user_id': username,
71.         'password': password
72.     }
73.     self.send_data({'type': 'null', 'choice': 'login'})
74.     self.send_data(user_data)
75.     data = self.recv_data()
76.     return data['valid_password']
77.     # recieve true or false
78.
79. def recv_data(self):
80.     print('[WAITING] to recieve data')
81.     data_length = self.client.recv(HEADER).decode(FORMAT)
82.     if data_length != 0:
83.         data_length = int(data_length)
84.         json_data = self.client.recv(data_length).decode(FORMAT)
85.         data = json.loads(json_data)
86.         return data
87.
88. def send_data(self, data):
89.     print('[DEBUG] SENDING DATA')
90.     data = json.dumps(data)
91.     data = data.encode(FORMAT)
92.     self.client.send(self.packet_header(data))
93.     self.client.send(data)
94.
95. def send(self, msg):
96.     message = msg.encode(FORMAT)
97.     self.client.send(self.packet_header(message))
98.     self.client.send(message)
99.
100. def packet_header(self, msg):
101.     msg_length = len(msg)
102.     send_length = str(msg_length).encode(FORMAT)
103.     print(f'[DEBUG] PROTOCOL {msg_length} -- {send_length}')
104.     send_length += b' ' * (HEADER - len(send_length))
105.     print(f'[SEND LENGTH] {send_length}')
106.     return send_length
107.
108. def listen(self):
109.     print('[CLIENT LISTENING]')
110.     while True:
111.         data = self.recv_data()
112.         if data['type'] != 'message':
113.             print('RECIEVED FRIEND LIST')
114.             self.ChatPage.current_friend_message_history = data['message history']
115.             self.ChatPage.display_friend_messages()
116.         else:
```

```
117.         self.ChatPage.message_recieved(data)
118.
119.     def get_user_input(self):
120.         while True:
121.             message = input('Enter message: ')
122.             self.send(message)
123.
124. class AppUI(tk.Tk):
125.
126.     def __init__(self, *args, **kwargs):
127.
128.         tk.Tk.__init__(self, *args, **kwargs)
129.         self.geometry("745x504")
130.         container = tk.Frame(self)
131.         self.screen_name = ""
132.         self.client = Client()
133.
134.         container.pack(side="top", fill="both", expand=True)
135.
136.         container.grid_rowconfigure(0, weight=1)
137.         container.grid_columnconfigure(0, weight=1)
138.
139.         self.frames = {}
140.
141.         for F in (LoginPage, ChatPage, PageTwo, CreateAccountPage, SettingsPage):
142.
143.             frame = F(container, self, self.client)
144.
145.             self.frames[F] = frame
146.
147.             frame.grid(row=0, column=0, sticky="nsew")
148.         self.connect_to_server()
149.         self.show_frame(LoginPage)
150.         self.ChatPage = self.frames[ChatPage]
151.
152.         self.client.set_chat_page(self.ChatPage)
153.
154.     def show_frame(self, new_frame):
155.         frame = self.frames[new_frame]
156.         frame.on_show()
157.         frame.tkraise()
158.
159.     def set_screen_name(self, data):
160.         self.screen_name = data
161.
162.     def get_screen_name(self):
163.         return self.screen_name
164.
165.     def connect_to_server(self):
166.         self.client.start()
167.         if self.client.connection_failed:
168.             print('AppUI reporting - connection error')
169.
170.     def on_close(self):
171.         print("[CLOSING APP]")
172.         try:
173.             self.client.send_data({'type': 'DISCONNECT'})
174.         except:
175.             pass
176.         self.destroy()
177.
178. class LoginPage(tk.Frame):
179.     def __init__(self, parent, window, client):
180.         self.window = window
181.         self.client = client
182.         tk.Frame.__init__(self, parent)
183.         self.create_and_place()
184.
185.     def on_show(self):
186.         if self.client.connection_failed:
```

```
187.         self.canvas.itemconfig(
188.             self.connection_error_message, state='normal'
189.         )
190.         for w in self.winfo_children():
191.             # disable all widgets on login frame
192.             # prevents user logging in when server is down
193.             w.configure(state="disabled")
194.
195.     def login(self):
196.         username_value = self.username_entry.get()
197.         password_value = self.password_entry.get()
198.         self.canvas.itemconfig(self.error_message, state='hidden')
199.         if username_value == '' or password_value == '':
200.             self.canvas.itemconfig(
201.                 self.error_message, text='Please fill in required fields')
202.             self.canvas.moveto(self.error_message, 257.0, 384.0)
203.             self.canvas.itemconfig(self.error_message, state='normal')
204.         else:
205.             valid_password = self.client.validate_login_info(
206.                 username_value, password_value)
207.             if valid_password:
208.                 print("Password Valid")
209.                 print("[LOGGED IN]")
210.                 json_data = self.client.recv_data()
211.                 self.client.screen_name = json_data['screen_name']
212.                 # self.window.set_screen_name(self.client.screen_name)
213.                 self.client.friends = json_data['friend_list']
214.                 self.client.start_listen_thread()
215.                 self.window.show_frame(ChatPage)
216.             else:
217.                 print("Invalid Password")
218.                 self.canvas.itemconfig(
219.                     self.error_message, text='Incorrect username or password')
220.                 self.canvas.moveto(self.error_message, 230.0, 384.0)
221.                 self.canvas.itemconfig(self.error_message, state='normal')
222.
223.     def show_hide_password(self):
224.         print('[BUTTON CLICKED] toggle_password_button')
225.         if self.password_entry.cget('show') == '':
226.             self.password_entry.config(show='*')
227.             self.toggle_password_button.config(image=self.eye_closed_image)
228.         else:
229.             self.password_entry.config(show='')
230.             self.toggle_password_button.config(image=self.eye_open_image)
231.         try:
232.             if self.confirm_password_feild.cget('show') == '':
233.                 self.confirm_password_feild.config(show='*')
234.                 self.toggle_password_button.config(image=self.eye_closed_image)
235.             else:
236.                 self.confirm_password_feild.config(show='')
237.                 self.confirm_password_feild.config(image=self.eye_open_image)
238.         except:
239.             pass
240.
241.     def create_and_place(self):
242.         self.canvas = tk.Canvas(
243.             self,
244.             bg="#0A0C10",
245.             height=504,
246.             width=745,
247.             bd=0,
248.             highlightthickness=0,
249.             relief="ridge"
250.         )
251.         self.canvas.place(x=0, y=0)
252.
253.         login_header_text = self.canvas.create_text(
254.             316.0,
255.             30.0,
256.             anchor="nw",
```

```
257.         text="Login",
258.         fill="#E3E7ED",
259.         font=("MontserratRoman Bold", 40 * -1)
260.     )
261.
262.     self.login_entry_image = tk.PhotoImage(
263.         file=relative_to_assets("login_entry.png"))
264.
265.     # USERNAME ENTRY
266.     username_entry_text = self.canvas.create_text(
267.         252.0,
268.         118.0,
269.         anchor="nw",
270.         text="USERNAME",
271.         fill="#E3E7ED",
272.         font=("MontserratRoman Regular", 16 * -1)
273.     )
274.     username_entry_bg = self.canvas.create_image(
275.         372.0,
276.         156.5,
277.         image=self.login_entry_image
278.     )
279.     self.username_entry = tk.Entry(
280.         self,
281.         bd=0,
282.         bg="#617998",
283.         fg="#000716",
284.         highlightthickness=0
285.     )
286.     self.username_entry.place(
287.         x=255.5,
288.         y=140.0,
289.         width=233.0,
290.         height=33.0
291.     )
292.
293.     # PASSWORD ENTRY
294.     password_entry_text = self.canvas.create_text(
295.         252.0,
296.         223.0,
297.         anchor="nw",
298.         text="PASSWORD",
299.         fill="#E3E7ED",
300.         font=("MontserratRoman Regular", 16 * -1)
301.     )
302.     password_entry_bg = self.canvas.create_image(
303.         372.0,
304.         261.5,
305.         image=self.login_entry_image
306.     )
307.     self.password_entry = tk.Entry(
308.         self,
309.         bd=0,
310.         bg="#617998",
311.         fg="#000716",
312.         highlightthickness=0,
313.         show='*'
314.     )
315.     self.password_entry.place(
316.         x=255.5,
317.         y=245.0,
318.         width=233.0,
319.         height=33.0
320.     )
321.     self.password_entry.bind(
322.         '<Return>', lambda e: self.login())
323.
324.     # CREATE ACCOUNT BUTTON
325.     self.create_account_button_image = tk.PhotoImage(
326.         file=relative_to_assets("login_create_account_button.png"))
```

```
327.
328.     create_account_button = tk.Button(
329.         self,
330.         image=self.create_account_button_image,
331.         borderwidth=0,
332.         highlightthickness=0,
333.         command=lambda: self.window.show_frame(CreateAccountPage),
334.         relief="flat",
335.         activebackground='#0A0C10'
336.     )
337.     create_account_button.place(
338.         x=271.0,
339.         y=415.0,
340.         width=203.0,
341.         height=25.0
342.     )
343.
344.     # LOGIN BUTTON
345.     # self is required because as soon as the function finishes the variable is
destroyed
346.     self.login_button_image = tk.PhotoImage(
347.         file=relative_to_assets("login_login_button.png"))
348.
349.     self.login_button = tk.Button(
350.         self,
351.         image=self.login_button_image,
352.         borderwidth=0,
353.         highlightthickness=0,
354.         command=lambda: self.login(),
355.         relief="flat",
356.         activebackground='#0A0C10'
357.     )
358.     self.login_button.place(
359.         x=311.0,
360.         y=328.0,
361.         width=123.0,
362.         height=39.0
363.     )
364.
365.     # TOGGLE PASSWORD VISABILITY BUTTON
366.     self.eye_open_image = tk.PhotoImage(
367.         file=relative_to_assets("login_eye_open.png"))
368.     self.eye_closed_image = tk.PhotoImage(
369.         file=relative_to_assets("login_eye_closed.png"))
370.
371.     self.toggle_password_button = tk.Button(
372.         self,
373.         image=self.eye_closed_image,
374.         borderwidth=0,
375.         highlightthickness=0,
376.         command=lambda: self.show_hide_password(),
377.         relief="flat",
378.         background='#0A0C10',
379.         activebackground='#0A0C10'
380.     )
381.     self.toggle_password_button.place(
382.         x=519.0,
383.         y=237.0,
384.         width=51.0,
385.         height=51.0
386.     )
387.
388.     # ERROR MESSAGE
389.     self.error_message = self.canvas.create_text(
390.         257.0,
391.         384.0,
392.         anchor="nw",
393.         text='',
394.         fill="#FF4747",
395.         font=("MontserratRoman Bold", 20 * -1),
```



```
396.         state='hidden'
397.     )
398.
399.     # CONNECTION ERROR MESSAGE
400.     self.connection_error_message = self.canvas.create_text(
401.         225.0,
402.         80.0,
403.         anchor="nw",
404.         text='Client unable to connect to server',
405.         fill="#FF4747",
406.         font=("MontserratRoman Bold", 20 * -1),
407.         state='hidden'
408.     )
409.
410. class CreateAccountPage(tk.Frame):
411.     def __init__(self, parent, window, client):
412.         self.window = window
413.         self.client = client
414.         tk.Frame.__init__(self, parent)
415.         self.create_and_place()
416.
417.     def on_show(self):
418.         pass
419.
420.     def check_unique_username(self, username_value):
421.         """
422.         Send username value to server
423.         Server checks if it is unique
424.         Return false
425.         """
426.         self.client.send_data({'type': 'null', 'username': username_value})
427.         return self.client.recv_data()['unique']
428.         # return True
429.
430.     def create_new_account(self):
431.         """Creates new account OR shows error message if feilds not filled in
correctly"""
432.         print("[BUTTON CLICKED] submit_button")
433.         username_value = self.username_entry_feild.get()
434.         screen_name_value = self.screen_name_entry_feild.get()
435.         password_value = self.password_entry.get()
436.         password_confirm_value = self.comfirm_password_feild.get()
437.         self.canvas.itemconfig(self.error_message, state='hidden')
438.         if username_value == '' or screen_name_value == '' or password_value == '' or
password_confirm_value == '':
439.             self.canvas.itemconfig(
440.                 self.error_message, text='Please fill in required fields')
441.             self.canvas.moveto(self.error_message, 250.0, 422.0)
442.             self.canvas.itemconfig(self.error_message, state='normal')
443.         elif password_value != password_confirm_value:
444.             self.canvas.itemconfig(
445.                 self.error_message, text='Your entered passwords do not match')
446.             self.canvas.moveto(self.error_message, 207.0, 422.0)
447.             self.canvas.itemconfig(self.error_message, state='normal')
448.         else:
449.             self.client.send_data({'type': 'null', 'choice': 'create account'})
450.             if not self.check_unique_username(username_value):
451.                 self.canvas.itemconfig(
452.                     self.error_message, text='Username taken. Please try a different
username')
453.                 self.canvas.moveto(self.error_message, 157.0, 422.0)
454.                 self.canvas.itemconfig(self.error_message, state='normal')
455.             else:
456.                 print(['ACCOUNT CREATED'])
457.                 self.client.send_data({
458.                     'type': 'null',
459.                     'screen_name': screen_name_value,
460.                     'password': password_value
461.                 })
462.                 self.client.screen_name = screen_name_value
```

```
463.         self.window.show_frame(ChatPage)
464.
465.     def create_and_place(self):
466.         self.canvas = tk.Canvas(
467.             self,
468.             bg="#0A0C10",
469.             height=504,
470.             width=745,
471.             bd=0,
472.             highlightthickness=0,
473.             relief="ridge"
474.         )
475.         self.canvas.place(x=0, y=0)
476.
477.         self.create_account_entry_image = tk.PhotoImage(
478.             file=relative_to_assets("create_account_entry.png"))
479.
480.         # USERNAME ENTRY
481.         username_entry_bg = self.canvas.create_image(
482.             205.5,
483.             149.5,
484.             image=self.create_account_entry_image
485.         )
486.         self.username_entry_feild = tk.Entry(
487.             self,
488.             bd=0,
489.             bg="#617998",
490.             fg="#000716",
491.             highlightthickness=0
492.         )
493.         self.username_entry_feild.place(
494.             x=87.5,
495.             y=133.0,
496.             width=236.0,
497.             height=33.0
498.         )
499.         self.canvas.create_text(
500.             85.0,
501.             110.0,
502.             anchor="nw",
503.             text="USERNAME",
504.             fill="#E3E7ED",
505.             font=("MontserratRoman Regular", 16 * -1)
506.         )
507.
508.         # SCREEN NAME ENTRY
509.         screen_name_entry_bg = self.canvas.create_image(
510.             205.5,
511.             217.5,
512.             image=self.create_account_entry_image
513.         )
514.         self.screen_name_entry_feild = tk.Entry(
515.             self,
516.             bd=0,
517.             bg="#617998",
518.             fg="#000716",
519.             highlightthickness=0
520.         )
521.         self.screen_name_entry_feild.place(
522.             x=87.5,
523.             y=201.0,
524.             width=236.0,
525.             height=33.0
526.         )
527.         self.canvas.create_text(
528.             85.0,
529.             178.0,
530.             anchor="nw",
531.             text="SCREEN NAME",
532.             fill="#E3E7ED",
```

```
533.         font=("MontserratRoman Regular", 16 * -1)
534.     )
535.
536.     # PASSWORD ENTRY
537.     self.password_entry_image = tk.PhotoImage(
538.         file=relative_to_assets("create_account_password_entry.png"))
539.
540.     password_entry_bg = self.canvas.create_image(
541.         515.5,
542.         149.5,
543.         image=self.password_entry_image
544.     )
545.     self.password_entry = tk.Entry(
546.         self,
547.         bd=0,
548.         bg="#617998",
549.         fg="#000716",
550.         highlightthickness=0,
551.         show='*'
552.     )
553.     self.password_entry.place(
554.         x=422.5,
555.         y=133.0,
556.         width=186.0,
557.         height=33.0
558.     )
559.
560.     self.canvas.create_text(
561.         421.0,
562.         110.0,
563.         anchor="nw",
564.         text="PASSWORD",
565.         fill="#E3E7ED",
566.         font=("MontserratRoman Regular", 16 * -1)
567.     )
568.
569.     # CONFIRM PASSWORD ENTRY
570.     confirm_password_bg = self.canvas.create_image(
571.         540.5,
572.         217.5,
573.         image=self.create_account_entry_image
574.     )
575.     self.confirm_password_feild = tk.Entry(
576.         self,
577.         bd=0,
578.         bg="#617998",
579.         fg="#000716",
580.         highlightthickness=0,
581.         show='*'
582.     )
583.     self.confirm_password_feild.place(
584.         x=422.5,
585.         y=201.0,
586.         width=236.0,
587.         height=33.0
588.     )
589.
590.     self.canvas.create_text(
591.         421.0,
592.         178.0,
593.         anchor="nw",
594.         text="CONFIRM PASSWORD",
595.         fill="#E3E7ED",
596.         font=("MontserratRoman Regular", 16 * -1)
597.     )
598.
599.     # CREATE ACCOUNT HEADER
600.     self.canvas.create_text(
601.         227.0,
602.         35.0,
```

```
603.         anchor="nw",
604.         text="Create Account",
605.         fill="#E3E7ED",
606.         font=("MontserratRoman Bold", 40 * -1)
607.     )
608.
609.     # ERROR MESSAGE
610.     self.error_message = self.canvas.create_text(
611.         240.0,
612.         422.0,
613.         anchor="nw",
614.         text="",
615.         fill="#FF4747",
616.         font=("MontserratRoman Bold", 20 * -1),
617.         state='hidden'
618.     )
619.
620.     # SUBMIT BUTTON
621.     self.submit_button_image = tk.PhotoImage(
622.         file=relative_to_assets("create_account_submit.png"))
623.
624.     submit_button = tk.Button(
625.         self,
626.         image=self.submit_button_image,
627.         borderwidth=0,
628.         highlightthickness=0,
629.         command=lambda: self.create_new_account(),
630.         relief="flat",
631.         activebackground='#0A0C10'
632.     )
633.     submit_button.place(
634.         x=293.0,
635.         y=266.0,
636.         width=149.0,
637.         height=43.0
638.     )
639.
640.     # LOGIN OPTION BUTTON
641.     self.login_option_image = tk.PhotoImage(
642.         file=relative_to_assets("create_account_login_option.png"))
643.
644.     login_option_button = tk.Button(
645.         self,
646.         image=self.login_option_image,
647.         borderwidth=0,
648.         highlightthickness=0,
649.         command=lambda: self.window.show_frame(LoginPage),
650.         relief="flat",
651.         activebackground='#0A0C10'
652.     )
653.     login_option_button.place(
654.         x=227.0,
655.         y=342.0,
656.         width=286.0,
657.         height=48.0
658.     )
659.
660.     # TOGGLE PASSWORD VISABILITY BUTTON
661.     self.eye_open_image = tk.PhotoImage(
662.         file=relative_to_assets("login_eye_open.png"))
663.     self.eye_closed_image = tk.PhotoImage(
664.         file=relative_to_assets("login_eye_closed.png"))
665.     self.toggle_password_button = tk.Button(
666.         self,
667.         image=self.eye_closed_image,
668.         borderwidth=0,
669.         highlightthickness=0,
670.         command=lambda: LoginPage.show_hide_password(self),
671.         relief="flat",
672.         background='#0A0C10',
```

```
673.         activebackground='#0A0C10'
674.     )
675.     self.toggle_password_button.place(
676.         x=633.0,
677.         y=125.0,
678.         width=51.0,
679.         height=51.0
680.     )
681.
682. class SettingsPage(tk.Frame):
683.     def __init__(self, parent, window, client):
684.         self.window = window
685.         self.client = client
686.         tk.Frame.__init__(self, parent)
687.         self.parent = parent
688.
689.     def on_show(self):
690.         self.create_and_place()
691.         self.canvas.itemconfig(
692.             self.current_screen_name_text, text=self.client.screen_name)
693.
694.     def show_chats_page(self):
695.         # self.canvas.delete(self.header_rectangle)
696.         self.window.show_frame(ChatPage)
697.
698.     def change_screen_name(self):
699.         self.canvas.itemconfig(self.error_message, state='hidden')
700.         if self.new_screen_name_entry.get() == self.client.screen_name:
701.             # if screen name is same as old screen name display error message
702.             self.canvas.itemconfig(self.error_message, state='normal',
703.                                     text='Your new screen can not be the same as your
current screen name')
704.         elif self.new_screen_name_entry.get() == '':
705.             self.canvas.itemconfig(self.error_message, state='normal',
706.                                     text='Your new screen name can not be empty')
707.         else:
708.             self.canvas.itemconfig(
709.                 self.current_screen_name_text, text=self.new_screen_name_entry.get())
710.             self.client.screen_name = self.new_screen_name_entry.get()
711.             self.client.send_data(
712.                 {'type': 'update screen name', 'new screen name':
self.client.screen_name})
713.
714.     def popup_frame(self):
715.         top = tk.Toplevel(self.window)
716.         top.protocol("WM_DELETE_WINDOW", self.popup_frame_error)
717.         self.window.protocol("WM_DELETE_WINDOW", self.popup_frame_error)
718.         top.geometry("479x375")
719.         top.resizable(False, False)
720.         top.title("Confirm account deletion")
721.
722.         popup_canvas = tk.Canvas(
723.             top,
724.             bg="#0A0C10",
725.             height=375,
726.             width=479,
727.             bd=0,
728.             highlightthickness=0,
729.             relief="ridge"
730.         )
731.
732.         popup_canvas.place(x=0, y=0)
733.         warning_header_text = popup_canvas.create_text(
734.             58.0,
735.             14.0,
736.             anchor="nw",
737.             text="Are you sure you want to \n  delete your account?",
738.             fill="#FF4747",
739.             font=("MontserratRoman Bold", 32 * -1)
740.         )
```

```
741.
742.     self.yes_button_image = tk.PhotoImage(
743.         file=relative_to_assets("confirmation_popup_yes.png"))
744.     yes_button = tk.Button(
745.         top,
746.         image=self.yes_button_image,
747.         borderwidth=0,
748.         highlightthickness=0,
749.         command=lambda: self.yes_button_clicked(top),
750.         relief="flat",
751.         activebackground='#2A3441'
752.     )
753.     yes_button.place(
754.         x=57.0,
755.         y=164.0,
756.         width=139.0,
757.         height=48.0
758.     )
759.
760.     self.no_button_image = tk.PhotoImage(
761.         file=relative_to_assets("confirmation_popup_no.png"))
762.     no_button = tk.Button(
763.         top,
764.         image=self.no_button_image,
765.         borderwidth=0,
766.         highlightthickness=0,
767.         command=lambda: self.no_button_clicked(top),
768.         relief="flat",
769.         activebackground='#2A3441'
770.     )
771.     no_button.place(
772.         x=279.0,
773.         y=163.0,
774.         width=139.0,
775.         height=50.0
776.     )
777.
778.     def popup_frame_error(self):
779.         tk.messagebox.showerror('Error', 'Please select either yes or no')
780.
781.     def no_button_clicked(self, top):
782.         top.destroy()
783.         self.window.protocol("WM_DELETE_WINDOW", self.window.on_close)
784.
785.     def yes_button_clicked(self, top):
786.         print('[DELETING ACCOUNT]')
787.         self.window.protocol("WM_DELETE_WINDOW", self.window.on_close)
788.
789.     def create_and_place(self):
790.         self.canvas = tk.Canvas(
791.             self,
792.             bg="#0A0C10",
793.             height=504,
794.             width=745,
795.             bd=0,
796.             highlightthickness=0,
797.             relief="ridge"
798.         )
799.
800.         self.canvas.place(x=0, y=0)
801.         self.header_rectangle = self.canvas.create_rectangle(
802.             0.0,
803.             0.0,
804.             745.0,
805.             43.0,
806.             fill="#12161C",
807.             outline=""
808.         )
809.         settings_header_text = self.canvas.create_text(
810.             16.0,
```

```
811.         8.0,
812.         anchor="nw",
813.         text="Settings",
814.         fill="#E3E7ED",
815.         font=("MontserratRoman Bold", 24 * -1)
816.     )
817.
818.     # CURRENT SCREEN NAME DISPLAY
819.     current_screen_name_header_text = self.canvas.create_text(
820.         16.0,
821.         61.0,
822.         anchor="nw",
823.         text="Current screen name",
824.         fill="#E3E7ED",
825.         font=("MontserratRoman Bold", 24 * -1)
826.     )
827.
828.     self.option_addcurrent_screen_name_image = tk.PhotoImage(
829.         file=relative_to_assets("settings_current_screen_name.png"))
830.     current_screen_name_bg = self.canvas.create_image(
831.         176.5,
832.         112.0,
833.         image=self.option_addcurrent_screen_name_image
834.     )
835.
836.     self.current_screen_name_text = self.canvas.create_text(
837.         37.0,
838.         99.0,
839.         anchor="nw",
840.         text="",
841.         fill="#E3E7ED",
842.         font=("MontserratRoman Bold", 24 * -1)
843.     )
844.
845.     # CHANGE SCREEN NAME DISPLAY
846.     change_screen_name_text = self.canvas.create_text(
847.         16.0,
848.         160.0,
849.         anchor="nw",
850.         text="Change screen name",
851.         fill="#E3E7ED",
852.         font=("MontserratRoman Bold", 24 * -1)
853.     )
854.
855.     self.new_screen_name_entry_image = tk.PhotoImage(
856.         file=relative_to_assets("settings_new_screen_name_entry.png"))
857.     new_screen_name_entry_bg = self.canvas.create_image(
858.         176.5,
859.         213.0,
860.         image=self.new_screen_name_entry_image
861.     )
862.     self.new_screen_name_entry = tk.Entry(
863.         self,
864.         bd=0,
865.         bg="#617998",
866.         fg="#E3E7ED",
867.         highlightthickness=0,
868.         font=("MontserratRoman")
869.     )
870.     self.new_screen_name_entry.place(
871.         x=37.0,
872.         y=192.0,
873.         width=279.0,
874.         height=40.0
875.     )
876.
877.     self.new_screen_name_entry.bind(
878.         '<Return>', lambda e: self.change_screen_name())
879.
880.     # CONFIRM SCREEN NAME BUTTON DISPLAY
```

```
881. self.confirm_screen_name_button_image = tk.PhotoImage(  
882.     file=relative_to_assets("settings_confirm_screen_name_button.png"))  
883. confirm_screen_name_button = tk.Button(  
884.     self,  
885.     image=self.confirm_screen_name_button_image,  
886.     borderwidth=0,  
887.     highlightthickness=0,  
888.     command=lambda: self.change_screen_name(),  
889.     relief="flat",  
890.     activebackground='#0A0C10'  
891. )  
892. confirm_screen_name_button.place(  
893.     x=359.0,  
894.     y=193.0,  
895.     width=151.0,  
896.     height=42.0  
897. )  
898.  
899. # DELETE ACCOUNT BUTTON DISPLAY  
900. self.delete_account_button_image = tk.PhotoImage(  
901.     file=relative_to_assets("settings_delete_account_button.png"))  
902. delete_account_button = tk.Button(  
903.     self,  
904.     image=self.delete_account_button_image,  
905.     borderwidth=0,  
906.     highlightthickness=0,  
907.     command=self.popup_frame,  
908.     relief="flat",  
909.     activebackground='#0A0C10'  
910. )  
911. delete_account_button.place(  
912.     x=16.0,  
913.     y=436.0,  
914.     width=229.0,  
915.     height=42.0  
916. )  
917.  
918. # REQUEST DATA BUTTON DISPLAY  
919. self.request_data_button_image = tk.PhotoImage(  
920.     file=relative_to_assets("settings_request_data_button.png"))  
921. request_data_button = tk.Button(  
922.     self,  
923.     image=self.request_data_button_image,  
924.     borderwidth=0,  
925.     highlightthickness=0,  
926.     command=lambda: print("button_2 clicked"),  
927.     relief="flat",  
928.     activebackground='#0A0C10'  
929. )  
930. request_data_button.place(  
931.     x=443.0,  
932.     y=436.0,  
933.     width=302.0,  
934.     height=42.0  
935. )  
936.  
937. # BACK TO CHATS BUTTON DISPLAY  
938. self.back_to_chats_button_image = tk.PhotoImage(  
939.     file=relative_to_assets("settings_back_to_chats_button.png"))  
940. back_to_chats_button = tk.Button(  
941.     self,  
942.     image=self.back_to_chats_button_image,  
943.     borderwidth=0,  
944.     highlightthickness=0,  
945.     command=lambda: self.show_chats_page(),  
946.     relief="flat",  
947.     activebackground='#0A0C10'  
948. )  
949. back_to_chats_button.place(  
950.     x=640.0,
```



```
951.         y=61.0,
952.         width=95.0,
953.         height=87.0
954.     )
955.
956.     self.error_message = self.canvas.create_text(
957.         16.0,
958.         250.0,
959.         anchor="nw",
960.         text="",
961.         fill="#FF4747",
962.         font=("MontserratRoman Bold", 20 * -1),
963.         state='hidden'
964.     )
965.
966. class ChatPage(tk.Frame):
967.     def __init__(self, parent, window, client):
968.         self.window = window
969.         self.client = client
970.         tk.Frame.__init__(self, parent)
971.         self.parent = parent
972.         # self.list_of_friends = ['Craig', 'Dave', 'Calum', 'Gilbert']
973.         # self.create_and_place()
974.         # self.add_temp_text()
975.         self.current_friend_message_history = None
976.         # self.last_selected = None
977.
978.     def on_show(self):
979.         self.create_and_place()
980.         self.add_temp_text()
981.         # self.client.start_listen_thread()
982.         print("[FRAME SHOWN] ChatsPage")
983.         self.message_entry_box.config(state='disabled')
984.         self.canvas.itemconfig(self.user_profile_text,
985.                                text=self.client.screen_name)
986.         if len(self.client.friends) != 0:
987.             self.show_friends()
988.             self.canvas.itemconfig(
989.                 self.recipient, text=self.client.friends[0][2])
990.             self.message_entry_box.config(state='normal')
991.             self.last_selected = tk.StringVar(
992.                 self.friend_list_area, str(list(self.client.friends[0])))
993.             self.v.set(str(list(self.client.friends[0])))
994.             # self.friend_chat_btn_press(self.v.get())
995.             self.client.send_data(
996.                 {'type': 'message history request', 'conversation_id':
self.get_radio_button_value()[0]})
997.             # print(f"[CHATS PAGE SHOWN] self.v set to:{self.client.friends[0]}")
998.             # print(f"[VALUE] self.v = {repr(self.v.get())}")
999.
1000.     def show_settings_page(self):
1001.         self.window.show_frame(SettingsPage)
1002.
1003.     def display_friend_messages(self):
1004.         for message in self.current_friend_message_history:
1005.             formatted_message = self.format_message(
1006.                 message[6], message[3], message[4], message[2])
1007.             self.display_message(formatted_message)
1008.
1009.     def friend_chat_btn_press(self, friend):
1010.         current_button = self.v.get()
1011.         self.message_display_box.config(state='normal')
1012.         self.message_display_box.delete(1.0, 'end')
1013.         if current_button != self.last_selected.get():
1014.             self.canvas.itemconfig(self.recipient, text=friend)
1015.             # Get chat specific messages from server
1016.             # delete all in current window
1017.             # paste in messages!
1018.             self.client.send_data(
```

```
1019.         {'type': 'message history request', 'conversation_id':
self.get_radio_button_value()[0]})
1020.         # message_history = self.client.recv_data()
1021.         # print(self.current_friend_message_history)
1022.         # self.message_display_box.config(state='normal')
1023.         # self.message_display_box.delete(1.0, 'end')
1024.         # print("CURRENT RADIO BUTTON")
1025.         # print(f"[VALUE] self.v = {repr(self.v.get())}")
1026.         self.last_selected.set(current_button)
1027.
1028.     def add_new_friend_to_UI(self, friend):
1029.         # print(f"[FRIEND RAIDOBTN] {friend}, {repr(friend)}")
1030.         btn = tk.Radiobutton(self.friend_list_area,
1031.                               text=friend[2],
1032.                               variable=self.v,
1033.                               value=str(friend),
1034.                               indicatoron=0,
1035.                               bg='#2A3441',
1036.                               height=2,
1037.                               width=23,
1038.                               relief='flat',
1039.                               fg='#E3E7ED',
1040.                               selectcolor='#12161C',
1041.                               command=lambda: self.friend_chat_btn_press(
1042.                                   friend[2]),
1043.                               borderwidth=0,
1044.                               font=("MontserratRoman", 9, 'normal'),
1045.                               )
1046.
1047.         self.friend_list_area.window_create('end', window=btn)
1048.         self.friend_list_area.insert('end', '\n')
1049.
1050.     def open_add_friend_popup(self):
1051.         pass
1052.
1053.     def add_temp_text(self):
1054.         self.message_entry_box.config(fg='#2a3441')
1055.         self.message_entry_box.insert(0, 'Type your message...')
1056.
1057.     def remove_temp_text(self):
1058.         self.message_entry_box.config(fg='#E3E7ED')
1059.         self.message_entry_box.delete(0, 'end')
1060.
1061.     def get_radio_button_value(self):
1062.         values = self.v.get()
1063.         # split_values = values.split(' ')
1064.         # print(f"[DEBUG] Radio button value = {values}")
1065.         return ast.literal_eval(values)
1066.
1067.     def message_recieved(self, data):
1068.         print(f"[MESSAGE RECIEVED!] {data}")
1069.         if data['screen_name'] == self.get_radio_button_value()[2]:
1070.             # data = json.dumps(data)
1071.             self.display_message(data)
1072.
1073.     def format_message(self, screen_name, date, time, message):
1074.         # sending screenname is nessesarary as while it is redundant
1075.         # due to the database on the serve it keeps a consistant format
1076.         user_data = {
1077.             'type': 'message',
1078.             'screen_name': screen_name,
1079.             'date': date,
1080.             'time': time,
1081.             'message': message,
1082.             'chat_id': self.get_radio_button_value()[0]
1083.         }
1084.         return user_data
1085.
1086.     def send_message(self, message):
1087.         print(f"[SEND MESSAGE FUNCTION CALLED] {message}")
```

```
1088.     screen_name, date, time = self.get_timestamp()
1089.     # format message to send to server in json format
1090.     # format message to display
1091.     formatted_message = self.format_message(
1092.         screen_name, date, time, message)
1093.     self.client.send_data(formatted_message)
1094.     self.display_message(formatted_message)
1095.     # self.store_message(timestamp, message)
1096.     # ALL we need to know is what conversation the user has click on.
1097.
1098.     def get_timestamp(self):
1099.         # screen_name = find_screen_name(conn, username)
1100.         screen_name = self.client.screen_name
1101.         now = datetime.now()
1102.         date = now.strftime("%d/%m/%Y")
1103.         time = now.strftime("%H:%M:%S")
1104.         return screen_name, date, time
1105.
1106.     def display_message(self, formatted_message):
1107.         # formatted_message = json.loads(formatted_message)
1108.         dt = f"{{formatted_message['date']}} {{formatted_message['time']}}"
1109.         screen_name = f"{{formatted_message['screen_name']}}"
1110.         timestamp = f"\n<{{screen_name}} {{dt}}>\n"
1111.         message = f"{{formatted_message['message']}}\n"
1112.         self.message_display_box.tag_add(
1113.             'timestamp_formatting', '1.0', '1.end')
1114.         self.message_display_box.tag_config('timestamp_formatting', font=(
1115.             'MontserratRoman', 10, 'italic'))
1116.         self.message_display_box.config(state='normal')
1117.         self.message_display_box.insert(
1118.             'end', timestamp, 'timestamp_formatting')
1119.         self.message_display_box.insert('end', message)
1120.         self.message_display_box.see('end')
1121.         self.message_display_box.config(state='disabled')
1122.         self.message_entry_box.delete(0, 'end')
1123.
1124.     def create_and_place(self):
1125.
1126.         self.canvas = tk.Canvas(
1127.             self,
1128.             bg="#0A0C10",
1129.             height=504,
1130.             width=745,
1131.             bd=0,
1132.             highlightthickness=0,
1133.             relief="ridge"
1134.         )
1135.
1136.         self.canvas.place(x=0, y=0)
1137.         self.side_bar_rectangle = self.canvas.create_rectangle(
1138.             0.0,
1139.             0.0,
1140.             190.0,
1141.             504.0,
1142.             fill="#2A3441",
1143.             outline="",
1144.             tags='rectangle')
1145.
1146.         self.canvas.create_text(
1147.             9.0,
1148.             14.0,
1149.             anchor="nw",
1150.             text="Chats",
1151.             fill="#E3E7ED",
1152.             font=("MontserratRoman Bold", 24 * -1)
1153.         )
1154.
1155.         # self.message_display_image = tk.PhotoImage(
1156.         #     file=relative_to_assets("message_display.png"))
1157.         # message_display_bg = canvas.create_image(
```

```
1158.         # 465.5,
1159.         # 251.5,
1160.         # image=self.message_display_image
1161.         # )
1162.     self.message_display_box = tk.Text(
1163.         self,
1164.         bd=0,
1165.         bg="#12161C",
1166.         fg="#E3E7ED",
1167.         highlightthickness=0,
1168.         font=("MontserratRoman", 14, 'normal'),
1169.         state='disabled'
1170.     )
1171.     self.message_display_box.place(
1172.         x=199.0,
1173.         y=54.0,
1174.         width=533.0,
1175.         height=393.0
1176.     )
1177.
1178.     self.message_entry_box_image = tk.PhotoImage(
1179.         file=relative_to_assets("entry_2.png"))
1180.     message_entry_box_bg = self.canvas.create_image(
1181.         485.0,
1182.         477.0,
1183.         image=self.message_entry_box_image
1184.     )
1185.
1186.     self.message_entry_box = tk.Entry(
1187.         self,
1188.         bd=0,
1189.         bg="#617998",
1190.         fg="#E3E7ED",
1191.         highlightthickness=0,
1192.         font=("MontserratRoman")
1193.     )
1194.     self.message_entry_box.place(
1195.         x=292.0,
1196.         y=463.0,
1197.         width=386.0,
1198.         height=28.0
1199.     )
1200.
1201.     self.message_entry_box.bind(
1202.         '<FocusIn>', lambda e: self.remove_temp_text())
1203.     self.message_entry_box.bind(
1204.         '<FocusOut>', lambda e: self.add_temp_text())
1205.     self.message_entry_box.bind(
1206.         '<Return>', lambda e: self.send_message(self.message_entry_box.get()))
1207.
1208.     # -----
1209.
1210.     self.recipient = self.canvas.create_text(
1211.         199.0,
1212.         14.0,
1213.         anchor="nw",
1214.         text="Add a friend to start chatting",
1215.         fill="#E3E7ED",
1216.         font=("MontserratRoman Bold", 24 * -1)
1217.     )
1218.
1219.     self.user_profile_text = self.canvas.create_text(
1220.         10.0,
1221.         466.0,
1222.         anchor="nw",
1223.         text='',
1224.         fill="#E3E7ED",
1225.         font=("MontserratRoman Bold", 24 * -1)
1226.     )
1227.
```

```
1228.         self.settings_image = tk.PhotoImage(  
1229.             file=relative_to_assets("settings_button.png"))  
1230.  
1231.         settings_button = tk.Button(  
1232.             self,  
1233.             image=self.settings_image,  
1234.             borderwidth=0,  
1235.             highlightthickness=0,  
1236.             command=lambda: self.show_settings_page(),  
1237.             relief="flat",  
1238.             bg='#2A3441',  
1239.             activebackground='#2A3441'  
1240.         )  
1241.         settings_button.place(  
1242.             x=151.0,  
1243.             y=463.0,  
1244.             width=30.0,  
1245.             height=30.0  
1246.         )  
1247.         self.add_friend_image = tk.PhotoImage(  
1248.             file=relative_to_assets("add_friend_button.png"))  
1249.         add_friend_button = tk.Button(  
1250.             self,  
1251.             image=self.add_friend_image,  
1252.             borderwidth=0,  
1253.             highlightthickness=0,  
1254.             command=lambda: self.open_add_friend_popup(),  
1255.             relief="flat",  
1256.             bg='#2A3441',  
1257.             activebackground='#2A3441'  
1258.         )  
1259.         add_friend_button.place(  
1260.             x=151.0,  
1261.             y=15.0,  
1262.             width=30.0,  
1263.             height=30.0  
1264.         )  
1265.  
1266.         self.send_message_image = tk.PhotoImage(  
1267.             file=relative_to_assets("send_message_button.png"))  
1268.         send_message_button = tk.Button(  
1269.             self,  
1270.             image=self.send_message_image,  
1271.             borderwidth=0,  
1272.             highlightthickness=0,  
1273.             command=lambda: self.send_message(self.message_entry_box.get()),  
1274.             relief="flat",  
1275.             bg='#0A0C10',  
1276.             activebackground='#0A0C10'  
1277.         )  
1278.         send_message_button.place(  
1279.             x=702.0,  
1280.             y=463.0,  
1281.             width=30.0,  
1282.             height=30.0  
1283.         )  
1284.  
1285.         self.attachment_button_image = tk.PhotoImage(  
1286.             file=relative_to_assets("attachment_button.png"))  
1287.         attachment_button = tk.Button(  
1288.             self,  
1289.             image=self.attachment_button_image,  
1290.             borderwidth=0,  
1291.             highlightthickness=0,  
1292.             command=lambda: print("[BUTTON CLICKED] attachment_button"),  
1293.             relief="flat",  
1294.             bg='#0A0C10',  
1295.             activebackground='#0A0C10'  
1296.         )  
1297.         attachment_button.place(  

```

```
1298.         x=199.0,
1299.         y=463.0,
1300.         width=30.0,
1301.         height=30.0
1302.     )
1303.
1304.     self.emoji_keyboard_image = tk.PhotoImage(
1305.         file=relative_to_assets("emoji_button.png"))
1306.
1307.     emoji_keyboard_button = tk.Button(
1308.         self,
1309.         image=self.emoji_keyboard_image,
1310.         borderwidth=0,
1311.         highlightthickness=0,
1312.         command=lambda: print("[BUTTON CLICKED] emoji_keyboard_button"),
1313.         relief="flat",
1314.         bg='#0A0C10',
1315.         activebackground='#0A0C10'
1316.     )
1317.     emoji_keyboard_button.place(
1318.         x=238.0,
1319.         y=463.0,
1320.         width=30.0,
1321.         height=30.0
1322.     )
1323.
1324. # FRIENDS LIST CODE SHTUFFSS
1325.
1326.     self.friend_list_area = tk.Text(
1327.         self,
1328.         bd=0,
1329.         bg="#12161C",
1330.         fg="#E3E7ED",
1331.         highlightthickness=0,
1332.         cursor='arrow'
1333.     )
1334.     self.friend_list_area.place(
1335.         x=0.0,
1336.         y=93.0,
1337.         width=190,
1338.         height=356
1339.     )
1340.     sb = tk.Scrollbar(self.friend_list_area,
1341.                       command=self.friend_list_area.yview, width=20)
1342.     sb.pack(side='right', fill=tk.Y)
1343.     self.friend_list_area.configure(yscrollcommand=sb.set)
1344.     self.v = tk.StringVar(self.friend_list_area)
1345.
1346.     def show_friends(self):
1347.         for friend in self.client.friends:
1348.             print(f"[SHOW FRIEND DEBUG] {friend}")
1349.             self.add_new_friend_to_UI(friend)
1350.
1351. class PageTwo(tk.Frame):
1352.
1353.     def __init__(self, parent, controller, client):
1354.         tk.Frame.__init__(self, parent)
1355.         label = tk.Label(self, text="Page Two!!!")
1356.         label.pack(pady=10, padx=10)
1357.
1358.         button1 = tk.Button(self, text="Back to Home",
1359.                             command=lambda: controller.show_frame(ChatPage))
1360.         button1.pack()
1361.
1362.         button2 = tk.Button(self, text="Page One",
1363.                             command=lambda: controller.show_frame(LoginPage))
1364.         button2.pack()
1365.
1366. app = AppUI()
1367. app.resizable(False, False)
```

```
1368. app.protocol("WM_DELETE_WINDOW", app.on_close)
1369. app.mainloop()
1370.
1371.
```

GUI OOP Approach Version 2

```
1. # GUI Related imports
2. import tkinter as tk
3. from tkinter import filedialog
4. import tkinter.messagebox as tk1
5. from PIL import Image, ImageTk
6. from datetime import datetime
7. from emojiKeyboardPackage import emojiKeyboard
8. import re
9.
10. # File manipulation imports
11. from pathlib import Path
12. import os
13.
14. # Networking Imports
15. import neat_secure_client as secure_client
16.
17.
18. # Asset management
19. OUTPUT_PATH = Path(__file__).parent
20. ASSETS_PATH = OUTPUT_PATH / \
21.     Path(r"C:\Users\orank\OneDrive\Desktop\Computer Science\A-level
NEA\build\assets\frame0")
22.
23.
24. def relative_to_assets(path: str) -> Path:
25.     return ASSETS_PATH / Path(path)
26.
27.
28. class UIController(tk.Tk):
29.
30.     def __init__(self, *args, **kwargs):
31.
32.         tk.Tk.__init__(self, *args, **kwargs)
33.
34.         # Creating main app
35.         self.geometry("745x504")
36.         app_frame = tk.Frame(self)
37.         app_frame.pack(side="top", fill="both", expand=True)
38.         app_frame.grid_rowconfigure(0, weight=1)
39.         app_frame.grid_columnconfigure(0, weight=1)
40.
41.         self.client = secure_client.Client()
42.
43.         self.screen_name = None
44.
45.         # Creating the different pages/frames
46.         self.frames = {}
47.
48.         for F in (LoginPage, CreateAccountPage, ChatPage, AddFriendPage, SettingsPage):
49.
50.             frame = F(self, app_frame, self.client)
51.
52.             self.frames[F] = frame
53.
54.             frame.grid(row=0, column=0, sticky="nsew")
55.
56.         # Connect to server and display first frame
57.         if self.client.connect():
58.             self.client.establish_initial_contact()
59.             self.show_frame(LoginPage)
60.
```

```
61.         self.ChatPage = self.frames[ChatPage]
62.         self.AddFriendPage = self.frames[AddFriendPage]
63.
64.         self.client.ChatPage = self.ChatPage
65.         self.client.AddFriendPage = self.AddFriendPage
66.
67.     def show_frame(self, new_frame: object):
68.         """Raises the frame in the parameter: new_frame"""
69.
70.         frame = self.frames[new_frame]
71.         # on_show acts like the __init__ class but for when a frame is shown instead
72.         frame.on_show()
73.         frame.tkraise()
74.
75.     def on_close(self, account_deletion=False):
76.         """Function called when the user closes the entire app"""
77.         try:
78.             self.client.stop_listen()
79.             self.client.send_disconnect_message()
80.             print('[-] CLOSING APP...')
81.         except Exception as e:
82.             print(e)
83.             self.destroy()
84.             self.client.close_db_connection()
85.             if account_deletion:
86.                 self.client.delete_directory()
87.             self.client.close_client()
88.
89.
90. class LoginPage(tk.Frame):
91.     def __init__(self, parent, window, client: secure_client.Client):
92.         self.controller = parent
93.         self.client = client
94.         tk.Frame.__init__(self, window)
95.
96.     def on_show(self):
97.         """Function called when frame is shown"""
98.         self.create_and_place()
99.         self.check_client_connection()
100.
101.     def create_and_place(self):
102.         """Creates and places all tkinter objects onto the frame"""
103.
104.         self.canvas = tk.Canvas(
105.             self,
106.             bg="#0A0C10",
107.             height=504,
108.             width=745,
109.             bd=0,
110.             highlightthickness=0,
111.             relief="ridge"
112.         )
113.         self.canvas.place(x=0, y=0)
114.
115.         login_header_text = self.canvas.create_text(
116.             316.0,
117.             30.0,
118.             anchor="nw",
119.             text="Login",
120.             fill="#E3E7ED",
121.             font=("MontserratRoman Bold", 40 * -1)
122.         )
123.
124.         self.login_entry_image = tk.PhotoImage(
125.             file=relative_to_assets("login_entry.png"))
126.
127.         # USERNAME ENTRY
128.         username_entry_text = self.canvas.create_text(
129.             252.0,
130.             118.0,
```



```
131.         anchor="nw",
132.         text="USERNAME",
133.         fill="#E3E7ED",
134.         font=("MontserratRoman Regular", 16 * -1)
135.     )
136.     username_entry_bg = self.canvas.create_image(
137.         372.0,
138.         156.5,
139.         image=self.login_entry_image
140.     )
141.     self.username_entry = tk.Entry(
142.         self,
143.         bd=0,
144.         bg="#617998",
145.         fg="#000716",
146.         highlightthickness=0
147.     )
148.     self.username_entry.place(
149.         x=255.5,
150.         y=140.0,
151.         width=233.0,
152.         height=33.0
153.     )
154.
155.     # PASSWORD ENTRY
156.     password_entry_text = self.canvas.create_text(
157.         252.0,
158.         223.0,
159.         anchor="nw",
160.         text="PASSWORD",
161.         fill="#E3E7ED",
162.         font=("MontserratRoman Regular", 16 * -1)
163.     )
164.     password_entry_bg = self.canvas.create_image(
165.         372.0,
166.         261.5,
167.         image=self.login_entry_image
168.     )
169.     self.password_entry = tk.Entry(
170.         self,
171.         bd=0,
172.         bg="#617998",
173.         fg="#000716",
174.         highlightthickness=0,
175.         show='*'
176.     )
177.     self.password_entry.place(
178.         x=255.5,
179.         y=245.0,
180.         width=233.0,
181.         height=33.0
182.     )
183.     self.password_entry.bind(
184.         '<Return>', lambda e: self.login())
185.
186.     # CREATE ACCOUNT BUTTON
187.     self.create_account_button_image = tk.PhotoImage(
188.         file=relative_to_assets("login_create_account_button.png"))
189.
190.     create_account_button = tk.Button(
191.         self,
192.         image=self.create_account_button_image,
193.         borderwidth=0,
194.         highlightthickness=0,
195.         command=lambda: self.controller.show_frame(CreateAccountPage),
196.         relief="flat",
197.         activebackground='#0A0C10'
198.     )
199.     create_account_button.place(
200.         x=271.0,
```

```
201.         y=415.0,
202.         width=203.0,
203.         height=25.0
204.     )
205.
206.     # LOGIN BUTTON
207.     # self is required because as soon as the function finishes the variable is
destroyed
208.     self.login_button_image = tk.PhotoImage(
209.         file=relative_to_assets("login_login_button.png"))
210.
211.     self.login_button = tk.Button(
212.         self,
213.         image=self.login_button_image,
214.         borderwidth=0,
215.         highlightthickness=0,
216.         command=lambda: self.login(),
217.         relief="flat",
218.         activebackground='#0A0C10'
219.     )
220.     self.login_button.place(
221.         x=311.0,
222.         y=328.0,
223.         width=123.0,
224.         height=39.0
225.     )
226.
227.     # TOGGLE PASSWORD VISABILITY BUTTON
228.     self.eye_open_image = tk.PhotoImage(
229.         file=relative_to_assets("login_eye_open.png"))
230.     self.eye_closed_image = tk.PhotoImage(
231.         file=relative_to_assets("login_eye_closed.png"))
232.
233.     self.toggle_password_button = tk.Button(
234.         self,
235.         image=self.eye_closed_image,
236.         borderwidth=0,
237.         highlightthickness=0,
238.         command=lambda: self.show_hide_password(),
239.         relief="flat",
240.         background='#0A0C10',
241.         activebackground='#0A0C10'
242.     )
243.     self.toggle_password_button.place(
244.         x=519.0,
245.         y=237.0,
246.         width=51.0,
247.         height=51.0
248.     )
249.
250.     # ERROR MESSAGE
251.     self.error_message = self.canvas.create_text(
252.         257.0,
253.         384.0,
254.         anchor="nw",
255.         text='',
256.         fill="#FF4747",
257.         font=("MontserratRoman Bold", 20 * -1),
258.         state='hidden'
259.     )
260.
261.     # CONNECTION ERROR MESSAGE
262.     self.connection_error_message = self.canvas.create_text(
263.         225.0,
264.         80.0,
265.         anchor="nw",
266.         text='Client unable to connect to server',
267.         fill="#FF4747",
268.         font=("MontserratRoman Bold", 20 * -1),
269.         state='hidden'
```

```
270.         )
271.
272.     def check_client_connection(self):
273.         """
274.         Checks the connection status of the client.
275.
276.         If client is NOT connected it disables all UI responsiveness
277.         """
278.         if self.client.connected == False:
279.             self.canvas.itemconfig(
280.                 self.connection_error_message, state='normal'
281.             )
282.             for w in self.wininfo_children():
283.                 # disable all widgets on login frame
284.                 # prevents user logging in when server is down
285.                 w.configure(state="disabled")
286.
287.     def login(self):
288.         username_value = self.username_entry.get()
289.         password_value = self.password_entry.get()
290.         self.canvas.itemconfig(self.error_message, state='hidden')
291.         if username_value == '' or password_value == '':
292.             self.canvas.itemconfig(
293.                 self.error_message, text='Please fill in required fields')
294.             self.canvas.moveto(self.error_message, 257.0, 384.0)
295.             self.canvas.itemconfig(self.error_message, state='normal')
296.         else:
297.             valid_password = self.client.login(username_value, password_value)
298.             if valid_password:
299.                 self.password_entry.unbind('<Return>')
300.                 self.client.handle_logged_in_client()
301.                 self.controller.show_frame(ChatPage)
302.             else:
303.                 print("Invalid Password")
304.                 self.canvas.itemconfig(
305.                     self.error_message, text='Incorrect username or password')
306.                 self.canvas.moveto(self.error_message, 230.0, 384.0)
307.                 self.canvas.itemconfig(self.error_message, state='normal')
308.
309.     def show_hide_password(self):
310.         print('{BUTTON CLICKED} toggle_password_button')
311.         if self.password_entry.cget('show') == '':
312.             self.password_entry.config(show='*')
313.             self.toggle_password_button.config(image=self.eye_closed_image)
314.         else:
315.             self.password_entry.config(show='')
316.             self.toggle_password_button.config(image=self.eye_open_image)
317.
318.         # Try block as function used by CreateAccountPage too
319.         try:
320.             if self.confirm_password_feild.cget('show') == '':
321.                 self.confirm_password_feild.config(show='*')
322.                 self.toggle_password_button.config(image=self.eye_closed_image)
323.             else:
324.                 self.confirm_password_feild.config(show='')
325.                 self.confirm_password_feild.config(image=self.eye_open_image)
326.         except:
327.             pass
328.
329.
330. class CreateAccountPage(tk.Frame):
331.     def __init__(self, parent, window, client: secure_client.Client):
332.         self.controller = parent
333.         self.client = client
334.         tk.Frame.__init__(self, window)
335.
336.     def on_show(self):
337.         # self.client.get_screen_name()
338.         self.create_and_place()
339.
```

```
340. def create_and_place(self):
341.     self.canvas = tk.Canvas(
342.         self,
343.         bg="#0A0C10",
344.         height=504,
345.         width=745,
346.         bd=0,
347.         highlightthickness=0,
348.         relief="ridge"
349.     )
350.     self.canvas.place(x=0, y=0)
351.
352.     self.create_account_entry_image = tk.PhotoImage(
353.         file=relative_to_assets("create_account_entry.png"))
354.
355.     # USERNAME ENTRY
356.     username_entry_bg = self.canvas.create_image(
357.         205.5,
358.         149.5,
359.         image=self.create_account_entry_image
360.     )
361.     self.username_entry_feild = tk.Entry(
362.         self,
363.         bd=0,
364.         bg="#617998",
365.         fg="#000716",
366.         highlightthickness=0
367.     )
368.     self.username_entry_feild.place(
369.         x=87.5,
370.         y=133.0,
371.         width=236.0,
372.         height=33.0
373.     )
374.     self.canvas.create_text(
375.         85.0,
376.         110.0,
377.         anchor="nw",
378.         text="USERNAME",
379.         fill="#E3E7ED",
380.         font=("MontserratRoman Regular", 16 * -1)
381.     )
382.
383.     # SCREEN NAME ENTRY
384.     screen_name_entry_bg = self.canvas.create_image(
385.         205.5,
386.         217.5,
387.         image=self.create_account_entry_image
388.     )
389.     self.screen_name_entry_feild = tk.Entry(
390.         self,
391.         bd=0,
392.         bg="#617998",
393.         fg="#000716",
394.         highlightthickness=0
395.     )
396.     self.screen_name_entry_feild.place(
397.         x=87.5,
398.         y=201.0,
399.         width=236.0,
400.         height=33.0
401.     )
402.     self.canvas.create_text(
403.         85.0,
404.         178.0,
405.         anchor="nw",
406.         text="SCREEN NAME",
407.         fill="#E3E7ED",
408.         font=("MontserratRoman Regular", 16 * -1)
409.     )
```

```
410.
411. # PASSWORD ENTRY
412. self.password_entry_image = tk.PhotoImage(
413.     file=relative_to_assets("create_account_password_entry.png"))
414.
415. password_entry_bg = self.canvas.create_image(
416.     515.5,
417.     149.5,
418.     image=self.password_entry_image
419. )
420. self.password_entry = tk.Entry(
421.     self,
422.     bd=0,
423.     bg="#617998",
424.     fg="#000716",
425.     highlightthickness=0,
426.     show='*'
427. )
428. self.password_entry.place(
429.     x=422.5,
430.     y=133.0,
431.     width=186.0,
432.     height=33.0
433. )
434.
435. self.canvas.create_text(
436.     421.0,
437.     110.0,
438.     anchor="nw",
439.     text="PASSWORD",
440.     fill="#E3E7ED",
441.     font=("MontserratRoman Regular", 16 * -1)
442. )
443.
444. # CONFIRM PASSWORD ENTRY
445. confirm_password_bg = self.canvas.create_image(
446.     540.5,
447.     217.5,
448.     image=self.create_account_entry_image
449. )
450. self.comfirm_password_feild = tk.Entry(
451.     self,
452.     bd=0,
453.     bg="#617998",
454.     fg="#000716",
455.     highlightthickness=0,
456.     show='*'
457. )
458. self.comfirm_password_feild.place(
459.     x=422.5,
460.     y=201.0,
461.     width=236.0,
462.     height=33.0
463. )
464.
465. self.canvas.create_text(
466.     421.0,
467.     178.0,
468.     anchor="nw",
469.     text="CONFIRM PASSWORD",
470.     fill="#E3E7ED",
471.     font=("MontserratRoman Regular", 16 * -1)
472. )
473.
474. # CREATE ACCOUNT HEADER
475. self.canvas.create_text(
476.     227.0,
477.     35.0,
478.     anchor="nw",
479.     text="Create Account",
```

```
480.         fill="#E3E7ED",
481.         font=("MontserratRoman Bold", 40 * -1)
482.     )
483.
484.     # ERROR MESSAGE
485.     self.error_message = self.canvas.create_text(
486.         240.0,
487.         422.0,
488.         anchor="nw",
489.         text="",
490.         fill="#FF4747",
491.         font=("MontserratRoman Bold", 20 * -1),
492.         state='hidden'
493.     )
494.
495.     # SUBMIT BUTTON
496.     self.submit_button_image = tk.PhotoImage(
497.         file=relative_to_assets("create_account_submit.png"))
498.
499.     submit_button = tk.Button(
500.         self,
501.         image=self.submit_button_image,
502.         borderwidth=0,
503.         highlightthickness=0,
504.         command=lambda: self.create_new_account(),
505.         relief="flat",
506.         activebackground='#0A0C10'
507.     )
508.     submit_button.place(
509.         x=293.0,
510.         y=266.0,
511.         width=149.0,
512.         height=43.0
513.     )
514.
515.     # LOGIN OPTION BUTTON
516.     self.login_option_image = tk.PhotoImage(
517.         file=relative_to_assets("create_account_login_option.png"))
518.
519.     login_option_button = tk.Button(
520.         self,
521.         image=self.login_option_image,
522.         borderwidth=0,
523.         highlightthickness=0,
524.         command=lambda: self.controller.show_frame(LoginPage),
525.         relief="flat",
526.         activebackground='#0A0C10'
527.     )
528.     login_option_button.place(
529.         x=227.0,
530.         y=342.0,
531.         width=286.0,
532.         height=48.0
533.     )
534.
535.     # TOGGLE PASSWORD VISABILITY BUTTON
536.     self.eye_open_image = tk.PhotoImage(
537.         file=relative_to_assets("login_eye_open.png"))
538.     self.eye_closed_image = tk.PhotoImage(
539.         file=relative_to_assets("login_eye_closed.png"))
540.     self.toggle_password_button = tk.Button(
541.         self,
542.         image=self.eye_closed_image,
543.         borderwidth=0,
544.         highlightthickness=0,
545.         command=lambda: LoginPage.show_hide_password(self),
546.         relief="flat",
547.         background='#0A0C10',
548.         activebackground='#0A0C10'
549.     )
```

```
550.         self.toggle_password_button.place(
551.             x=633.0,
552.             y=125.0,
553.             width=51.0,
554.             height=51.0
555.         )
556.
557.     def secure_password(self, password: str):
558.         valid = False
559.         special_char = re.compile('[@_!$%^&*()<>?/\|]{~:}#')
560.         general_pattern = re.compile(
561.             '^(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9]).{8,64}$')
562.
563.         if special_char.search(password) == None and re.match(general_pattern, password):
564.             valid = True
565.         return valid
566.
567.     def create_new_account(self):
568.         """Creates new account OR shows error message if feilds not filled in
correctly"""
569.         print("{BUTTON CLICKED} submit_button")
570.         self.canvas.itemconfig(self.error_message, state='hidden')
571.
572.         # getting relevant values
573.         username_value = self.username_entry_feild.get()
574.         screen_name_value = self.screen_name_entry_feild.get()
575.         password_value = self.password_entry.get()
576.         password_confirm_value = self.comfirm_password_feild.get()
577.
578.         if username_value == '' or screen_name_value == '' or password_value == '' or
password_confirm_value == '':
579.             self.canvas.itemconfig(
580.                 self.error_message, text='Please fill in required fields')
581.             self.canvas.moveto(self.error_message, 250.0, 422.0)
582.             self.canvas.itemconfig(self.error_message, state='normal')
583.             elif password_value != password_confirm_value:
584.                 self.canvas.itemconfig(
585.                     self.error_message, text='Your entered passwords do not match')
586.                 self.canvas.moveto(self.error_message, 207.0, 422.0)
587.                 self.canvas.itemconfig(self.error_message, state='normal')
588.             elif not self.secure_password(password_value):
589.                 self.canvas.itemconfig(
590.                     self.error_message, text='Password is not secure enough')
591.                 self.canvas.moveto(self.error_message, 240.0, 422.0)
592.                 self.canvas.itemconfig(self.error_message, state='normal')
593.
594.             else: # actually send data to client
595.                 unique_userID, account_created = self.client.create_account(
596.                     username_value, password_value, screen_name_value)
597.                 if unique_userID and account_created:
598.                     self.client.handel_logged_in_client()
599.                     self.controller.show_frame(ChatPage)
600.                     print('{ACCOUNT CREATED}')
601.                 elif unique_userID != True:
602.                     self.canvas.itemconfig(
603.                         self.error_message, text='Username taken. Please try a different
username'
604.                     )
605.                     self.canvas.moveto(self.error_message, 157.0, 422.0)
606.                     self.canvas.itemconfig(self.error_message, state='normal')
607.                 else:
608.                     self.canvas.itemconfig(
609.                         self.error_message, text='Something went wrong when creating your
account, please try again later'
610.                     )
611.                     self.canvas.moveto(self.error_message, 157.0, 422.0)
612.                     self.canvas.itemconfig(self.error_message, state='normal')
613.
614.
615. class ChatPage(tk.Frame):
```

```
616. def __init__(self, parent, window, client: secure_client.Client):
617.     self.controller = parent
618.     self.client = client
619.     tk.Frame.__init__(self, window)
620.     self.current_friend_message_history = None
621.     self.active_chat_user_details = None
622.     self.last_selected = None
623.     self.list_of_buttons = []
624.     self.friend_screen_name = None
625.     self.send_image_data = None
626.     self.images = []
627.
628. def on_show(self):
629.     self.client.get_friend_list()
630.     self.create_and_place()
631.     self.add_temp_text()
632.     self.show_friends()
633.     self.client.listen()
634.     self.disable_message_buttons()
635.
636. def add_temp_text(self):
637.     self.add_temp_text_to_message_entry()
638.     if len(self.client.friend_list) == 0:
639.         self.canvas.itemconfig(
640.             self.recipient, text='Add a friend to start chatting')
641.     else:
642.         self.canvas.itemconfig(
643.             self.recipient, text='Click on a friend to start chatting')
644.
645. def create_and_place(self):
646.
647.     self.canvas = tk.Canvas(
648.         self,
649.         bg="#0A0C10",
650.         height=504,
651.         width=745,
652.         bd=0,
653.         highlightthickness=0,
654.         relief="ridge"
655.     )
656.
657.     self.canvas.place(x=0, y=0)
658.     self.side_bar_rectangle = self.canvas.create_rectangle(
659.         0.0,
660.         0.0,
661.         190.0,
662.         504.0,
663.         fill="#2A3441",
664.         outline="",
665.         tags='rectangle')
666.
667.     self.canvas.create_text(
668.         9.0,
669.         14.0,
670.         anchor="nw",
671.         text="Chats",
672.         fill="#E3E7ED",
673.         font=("MontserratRoman Bold", 24 * -1)
674.     )
675.
676.     self.message_display_box = tk.Text(
677.         self,
678.         bd=0,
679.         bg="#12161C",
680.         fg="#E3E7ED",
681.         highlightthickness=0,
682.         font=("MontserratRoman", 14, 'normal'),
683.         state='disabled'
684.     )
685.     self.message_display_box.place(
```



```
686.         x=199.0,
687.         y=54.0,
688.         width=533.0,
689.         height=393.0
690.     )
691.
692.     self.message_entry_box_image = tk.PhotoImage(
693.         file=relative_to_assets("entry_2.png"))
694.     message_entry_box_bg = self.canvas.create_image(
695.         485.0,
696.         477.0,
697.         image=self.message_entry_box_image
698.     )
699.
700.     self.message_entry_box = tk.Entry(
701.         self,
702.         bd=0,
703.         bg="#617998",
704.         fg="#E3E7ED",
705.         highlightthickness=0,
706.         font=("MontserratRoman")
707.     )
708.     self.message_entry_box.place(
709.         x=292.0,
710.         y=463.0,
711.         width=386.0,
712.         height=28.0
713.     )
714.
715.     self.message_entry_box.bind(
716.         '<FocusIn>', lambda e: self.remove_temp_text_from_message_entry())
717.     self.message_entry_box.bind(
718.         '<FocusOut>', lambda e: self.add_temp_text_to_message_entry())
719.
720.     self.message_entry_box.bind(
721.         '<Return>', lambda e: self.send_message(self.message_entry_box.get()))
722.
723.     # -----
724.
725.     self.recipient = self.canvas.create_text(
726.         199.0,
727.         14.0,
728.         anchor="nw",
729.         text="",
730.         fill="#E3E7ED",
731.         font=("MontserratRoman Bold", 24 * -1)
732.     )
733.
734.     self.screen_name_text = self.canvas.create_text(
735.         10.0,
736.         466.0,
737.         anchor="nw",
738.         text=self.client.screen_name,
739.         fill="#E3E7ED",
740.         font=("MontserratRoman Bold", 24 * -1)
741.     )
742.
743.     self.settings_image = tk.PhotoImage(
744.         file=relative_to_assets("settings_button.png"))
745.
746.     settings_button = tk.Button(
747.         self,
748.         image=self.settings_image,
749.         borderwidth=0,
750.         highlightthickness=0,
751.         command=lambda: self.controller.show_frame(SettingsPage),
752.         relief="flat",
753.         bg='#2A3441',
754.         activebackground='#2A3441'
755.     )
```

```
756.         settings_button.place(  
757.             x=151.0,  
758.             y=463.0,  
759.             width=30.0,  
760.             height=30.0  
761.         )  
762.         self.add_friend_image = tk.PhotoImage(  
763.             file=relative_to_assets("add_friend_button.png"))  
764.         add_friend_button = tk.Button(  
765.             self,  
766.             image=self.add_friend_image,  
767.             borderwidth=0,  
768.             highlightthickness=0,  
769.             command=lambda: self.controller.show_frame(AddFriendPage),  
770.             relief="flat",  
771.             bg='#2A3441',  
772.             activebackground='#2A3441'  
773.         )  
774.         add_friend_button.place(  
775.             x=151.0,  
776.             y=15.0,  
777.             width=30.0,  
778.             height=30.0  
779.         )  
780.  
781.         self.send_message_image = tk.PhotoImage(  
782.             file=relative_to_assets("send_message_button.png"))  
783.         self.send_message_button = tk.Button(  
784.             self,  
785.             image=self.send_message_image,  
786.             borderwidth=0,  
787.             highlightthickness=0,  
788.             command=lambda: self.send_message(  
789.                 self.message_entry_box.get()),  
790.             relief="flat",  
791.             bg='#0A0C10',  
792.             activebackground='#0A0C10'  
793.         )  
794.         self.send_message_button.place(  
795.             x=702.0,  
796.             y=463.0,  
797.             width=30.0,  
798.             height=30.0  
799.         )  
800.  
801.         self.attachment_button_image = tk.PhotoImage(  
802.             file=relative_to_assets("attachment_button.png"))  
803.         self.attachment_button = tk.Button(  
804.             self,  
805.             image=self.attachment_button_image,  
806.             borderwidth=0,  
807.             highlightthickness=0,  
808.             command=self.handel_getting_image,  
809.             relief="flat",  
810.             bg='#0A0C10',  
811.             activebackground='#0A0C10'  
812.         )  
813.         self.attachment_button.place(  
814.             x=199.0,  
815.             y=463.0,  
816.             width=30.0,  
817.             height=30.0  
818.         )  
819.  
820.         self.emoji_keyboard_image = tk.PhotoImage(  
821.             file=relative_to_assets("emoji_button.png"))  
822.  
823.         self.emoji_keyboard_button = tk.Button(  
824.             self,  
825.             image=self.emoji_keyboard_image,
```

```
826.         borderwidth=0,
827.         highlightthickness=0,
828.         command=lambda: self.open_emoji_keyboard(),
829.         relief="flat",
830.         bg='#0A0C10',
831.         activebackground='#0A0C10'
832.     )
833.     self.emoji_keyboard_button.place(
834.         x=238.0,
835.         y=463.0,
836.         width=30.0,
837.         height=30.0
838.     )
839.
840.
841. # FRIENDS LIST CODE
842.
843.     self.friend_list_area = tk.Text(
844.         self,
845.         bd=0,
846.         bg="#12161C",
847.         fg="#E3E7ED",
848.         highlightthickness=0,
849.         cursor='arrow',
850.         state='disabled'
851.     )
852.     self.friend_list_area.place(
853.         x=0.0,
854.         y=93.0,
855.         width=190,
856.         height=356
857.     )
858.     sb = tk.Scrollbar(self.friend_list_area,
859.                       command=self.friend_list_area.yview, width=20)
860.     sb.pack(side='right', fill=tk.Y)
861.     self.friend_list_area.configure(yscrollcommand=sb.set)
862.     self.active_chat_user_details = tk.StringVar(self.friend_list_area)
863.
864.     def disable_message_buttons(self):
865.         self.message_entry_box.config(state='disabled')
866.         self.send_message_button.config(state='disabled')
867.         self.emoji_keyboard_button.config(state='disabled')
868.         self.attachment_button.config(state='disabled')
869.
870.
871. # -----EMOJI KEYBOARD LOGIC-----
872.
873.     def open_emoji_keyboard(self):
874.         print('Opening keyboard')
875.         # self.newWindow = tk.Toplevel(self.controller)
876.         emoji_keyboard = emojiKeyboard.Keyboard(self)
877.
878.     def insert_emoji(self, emojis: str):
879.         self.message_entry_box.focus_set()
880.         self.remove_temp_text_from_message_entry()
881.         self.message_entry_box.insert('end', emojis)
882.
883. # -----SENDING IMAGE LOGIC-----
884.
885.     def handel_getting_image(self):
886.         self.client.client_get_image_path()
887.         if self.client.image_path:
888.             self.send_image_data = self.client.get_image_data(
889.                 self.client.image_path)
890.             image_name = self.client.image_path.split('/')[-1]
891.             self.remove_temp_text_from_message_entry()
892.             self.add_specific_temp_text_to_msg_entry(image_name)
893.
894.             self.message_entry_box.config(state='disabled')
895.
```

```
896.
897. # -----FRIEND LIST LOGIC-----
898.
899.     def add_new_friend_to_UI(self, friend_screen_name: str, friend_details: str, status:
str, specifier_id: str):
900.
901.         state = 'normal'
902.         colour = '#2A3441'
903.         active = '#12161C'
904.         if status == 'blk' and specifier_id == self.client.user_id: # you have blocked
them
905.             colour = '#FF4747'
906.             active = '#612a2a'
907.             elif (status == 'blk' and specifier_id != self.client.user_id) or ('deleted
account' in friend_screen_name): # they have blocked you
908.                 state = 'disabled'
909.
910.         # when clicked self.active_chat_user_details is set to value
911.         # text is the text shown on the button value itself
912.         btn = tk.Radiobutton(self.friend_list_area,
913.                               text=friend_screen_name,
914.                               variable=self.active_chat_user_details,
915.                               value=friend_details,
916.                               indicatoron=0,
917.                               bg=colour,
918.                               height=2,
919.                               width=23,
920.                               relief='flat',
921.                               fg='#E3E7ED',
922.                               selectcolor=active,
923.                               command=lambda: self.friend_chat_btn_press(btn),
924.                               borderwidth=0,
925.                               font=("MontserratRoman", 9, 'normal'),
926.                               state=state
927.                             )
928.         self.list_of_buttons.append(btn)
929.         self.friend_list_area.window_create('end', window=btn)
930.         self.friend_list_area.insert('end', '\n')
931.
932.     def friend_chat_btn_press(self, btn: tk.Radiobutton):
933.         if self.active_chat_user_details.get() != self.last_selected:
934.             # getting details
935.             friend_details = self.active_chat_user_details.get().split(' ')
936.             friend_user_id = friend_details[0]
937.             friend_public_key = friend_details[1]
938.             self.friend_screen_name = btn.cget("text")
939.
940.             # updating UI
941.             self.canvas.itemconfig(
942.                 self.recipient, text=self.friend_screen_name)
943.             self.last_selected = self.active_chat_user_details.get().split(' ')
944.
945.             self.message_display_box.config(state='normal')
946.             self.message_display_box.delete(1.0, 'end')
947.
948.             # insert message history here
949.             self.client.decrypt_message_history(
950.                 self.active_chat_user_details.get().split(' ')[0])
951.             self.show_message_history()
952.
953.             if btn.cget('bg') == '#FF4747':
954.                 self.message_entry_box.config(state='disabled')
955.                 self.send_message_button.config(state='disabled')
956.                 self.emoji_keyboard_button.config(state='disabled')
957.                 self.attachment_button.config(state='disabled')
958.             else:
959.                 self.message_entry_box.config(state='normal')
960.                 self.send_message_button.config(state='normal')
961.                 self.emoji_keyboard_button.config(state='normal')
962.                 self.attachment_button.config(state='normal')
```

```
963.         self.message_entry_box.focus_set()
964.         self.message_entry_box.delete(0, 'end')
965.
966.     def show_friends(self):
967.         for friend in self.client.friend_list:
968.             friend_screen_name = friend[2]
969.             friend_details = friend[0:2]
970.             status = friend[3]
971.             specifier_id = friend[4]
972.             self.add_new_friend_to_UI(
973.                 friend_screen_name, friend_details, status, specifier_id)
974.
975.     def update_friend_list(self):
976.         for btn in self.list_of_buttons:
977.             btn.destroy()
978.         self.show_friends()
979.
980.
981. # -----SEND MESSAGE LOGIC-----
982.
983.
984.     def show_message_history(self):
985.         for message_data in self.client.current_message_history:
986.             formatted_message = self.format_stored_message_for_display(
987.                 message_data)
988.             self.client.image_path = formatted_message['message']
989.             self.display_message(
990.                 self.format_stored_message_for_display(message_data))
991.
992.     def handel_sending_message_or_image(self, message: str):
993.         pass
994.
995.     def send_message(self, message: str):
996.         """Executes procedure to display message and send message to the server"""
997.         if len(message) != 0:
998.             is_image = 0
999.             msg = message
1000.             image_name_and_format = ''
1001.
1002.             if self.message_entry_box.cget('state') == 'disabled':
1003.                 print('[ - ] SENDING IMAGE')
1004.                 is_image = 1
1005.                 msg = self.send_image_data
1006.                 image_name_and_format = self.client.image_name_and_format
1007.
1008.             date, time = self.get_timestamp()
1009.             formatted_message = self.format_message(
1010.                 self.client.screen_name, date, time, msg, is_image, image_name_and_format)
1011.             self.client.handel_send_message(formatted_message)
1012.             self.display_message(formatted_message)
1013.             if is_image:
1014.                 os.remove(self.client.image_path)
1015.
1016.     def get_timestamp(self):
1017.         """Gets current data and time"""
1018.         now = datetime.now()
1019.         date = now.strftime("%d/%m/%Y")
1020.         time = now.strftime("%H:%M:%S")
1021.         return date, time
1022.
1023.     def format_stored_message_for_display(self, message_data):
1024.         decrypted_message = message_data[0]
1025.         date = message_data[1]
1026.         time = message_data[2]
1027.         from_me = message_data[3]
1028.         is_image = message_data[4]
1029.
1030.         if from_me:
1031.             screen_name = self.client.screen_name
1032.         else:
```

```
1033.         screen_name = self.friend_screen_name
1034.
1035.         return self.format_message(screen_name, date, time, decrypted_message, is_image)
1036.
1037.     def format_message(self, screen_name, date, time, message, is_image,
image_name_and_format=''):
1038.         """Turns message into dict"""
1039.         user_data = {
1040.             'type': 'message',
1041.             'recipient': self.active_chat_user_details.get().split(' ')[0],
1042.             'config': 'message',
1043.             'sender_user_id': self.client.user_id,
1044.             'sender_screen_name': screen_name,
1045.             'date': date,
1046.             'time': time,
1047.             'message': message,
1048.             'is_image': is_image,
1049.             'image_name_and_format': image_name_and_format
1050.         }
1051.         return user_data
1052.
1053.     def display_message(self, formatted_message: dict):
1054.         """Displays message with appropriate formatting onto screen"""
1055.
1056.         self.message_display_box.tag_add(
1057.             'timestamp_formatting', '1.0', '1.end')
1058.         self.message_display_box.tag_config('timestamp_formatting', font=(
1059.             'MontserratRoman', 10, 'italic'))
1060.
1061.         # formatting text
1062.         if formatted_message['is_image'] == False:
1063.             self.display_message_text(formatted_message)
1064.         else:
1065.             self.display_message_image(formatted_message)
1066.             self.message_entry_box.delete(0, 'end')
1067.
1068.     def display_message_image(self, formatted_message: dict):
1069.         dt = f"{formatted_message['date']} {formatted_message['time']}"
1070.         screen_name = f"{formatted_message['sender_screen_name']}"
1071.         timestamp = f"\n<{screen_name} {dt}>\n"
1072.         image_path = self.client.image_path
1073.
1074.         # global image
1075.         unprocessed_image = Image.open(image_path)
1076.
1077.         image = ImageTk.PhotoImage(unprocessed_image)
1078.         self.images.append(image)
1079.
1080.         self.message_display_box.config(state='normal')
1081.         self.message_display_box.insert(
1082.             'end', timestamp, 'timestamp_formatting')
1083.         self.message_display_box.image_create(tk.END, image=image)
1084.         self.message_display_box.insert('end', '\n')
1085.         self.message_display_box.see('end')
1086.         self.message_display_box.config(state='disabled')
1087.         self.message_entry_box.config(state='normal')
1088.
1089.     def display_message_text(self, formatted_message: dict):
1090.         dt = f"{formatted_message['date']} {formatted_message['time']}"
1091.         screen_name = f"{formatted_message['sender_screen_name']}"
1092.         timestamp = f"\n<{screen_name} {dt}>\n"
1093.         message = f"{formatted_message['message']}\n"
1094.
1095.         # displaying message
1096.         self.message_display_box.config(state='normal')
1097.         self.message_display_box.insert(
1098.             'end', timestamp, 'timestamp_formatting')
1099.         self.message_display_box.insert('end', message)
1100.         self.message_display_box.see('end')
1101.         self.message_display_box.config(state='disabled')
```

```
1102.
1103. def add_temp_text_to_message_entry(self):
1104.     if len(self.message_entry_box.get()) == 0:
1105.         self.message_entry_box.delete(0, 'end')
1106.         self.message_entry_box.config(fg='#2a3441')
1107.         self.message_entry_box.insert(0, 'Type your message...')
1108.
1109. def remove_temp_text_from_message_entry(self):
1110.     self.message_entry_box.config(fg='#E3E7ED')
1111.     if self.message_entry_box.get() == 'Type your message...':
1112.         print('REMOVING TEMP TEXT')
1113.         self.message_entry_box.config(fg='#E3E7ED')
1114.         self.message_entry_box.delete(0, 'end')
1115.
1116. def add_specific_temp_text_to_msg_entry(self, text: str):
1117.     self.message_entry_box.config(fg='#2a3441')
1118.     self.message_entry_box.insert(0, text)
1119.
1120.
1121. class AddFriendPage(tk.Frame):
1122.     def __init__(self, parent, window, client: secure_client.Client):
1123.         self.controller = parent
1124.         self.client = client
1125.         tk.Frame.__init__(self, window)
1126.
1127.         self.active_friend_button = None
1128.         self.active_request_button = None
1129.
1130.     def on_show(self):
1131.         """Function called when frame is shown"""
1132.         self.client.stop_listen()
1133.         self.client.get_friend_list()
1134.         self.client.get_friend_request_list()
1135.         self.client.get_pending_friends_list()
1136.         self.create_and_place()
1137.         self.add_text()
1138.         self.add_radio_buttons()
1139.
1140.     def create_and_place(self):
1141.         """Creates and places all tkinter objects onto the frame"""
1142.         self.canvas = tk.Canvas(
1143.             self,
1144.             bg="#0A0C10",
1145.             height=504,
1146.             width=745,
1147.             bd=0,
1148.             highlightthickness=0,
1149.             relief="ridge"
1150.         )
1151.
1152.         self.canvas.place(x=0, y=0)
1153.         self.canvas.create_rectangle(
1154.             0.0,
1155.             0.0,
1156.             745.0,
1157.             32.0,
1158.             fill="#12161C",
1159.             outline="")
1160.
1161.         page_header = self.canvas.create_text(
1162.             8.0,
1163.             6.0,
1164.             anchor="nw",
1165.             text="Friends",
1166.             fill="FFFFFF",
1167.             font=("MontserratRoman Bold", 16 * -1)
1168.         )
1169.
1170.         self.back_to_chats_button_image = tk.PhotoImage(
1171.             file=relative_to_assets("back_to_chats_button.png"))
```

```
1172. back_to_chats_button = tk.Button(  
1173.     self,  
1174.     image=self.back_to_chats_button_image,  
1175.     borderwidth=0,  
1176.     highlightthickness=0,  
1177.     command=lambda: self.controller.show_frame(ChatPage),  
1178.     relief="flat"  
1179. )  
1180. back_to_chats_button.place(  
1181.     x=640.0,  
1182.     y=61.0,  
1183.     width=95.0,  
1184.     height=87.0  
1185. )  
1186.  
1187. friends_friend_code_entry_text = self.canvas.create_text(  
1188.     8.0,  
1189.     121.0,  
1190.     anchor="nw",  
1191.     text="Enter your friend's code to invite them to chat",  
1192.     fill="#FFFFFF",  
1193.     font=("MontserratRoman Bold", 16 * -1)  
1194. )  
1195.  
1196. personal_friend_code_entry_text = self.canvas.create_text(  
1197.     9.0,  
1198.     38.0,  
1199.     anchor="nw",  
1200.     text="Your Friend Code",  
1201.     fill="#FFFFFF",  
1202.     font=("MontserratRoman Bold", 16 * -1)  
1203. )  
1204.  
1205. self.friend_code_error_message = self.canvas.create_text(  
1206.     8.0,  
1207.     208.0,  
1208.     anchor="nw",  
1209.     text="",  
1210.     fill="#FF4747",  
1211.     font=("MontserratRoman Bold", 13 * -1),  
1212.     state='hidden'  
1213. )  
1214.  
1215. self.personal_friend_code_entry_image = tk.PhotoImage(  
1216.     file=relative_to_assets("add_friend_personal_friend_code_entry.png"))  
1217. personal_friend_code_entry_bg = self.canvas.create_image(  
1218.     169.5,  
1219.     84.0,  
1220.     image=self.personal_friend_code_entry_image  
1221. )  
1222.  
1223. self.friends_friend_code_entry_image = tk.PhotoImage(  
1224.     file=relative_to_assets("add_friend_friends_friend_code_entry.png"))  
1225. entry_bg_2 = self.canvas.create_image(  
1226.     169.5,  
1227.     171.0,  
1228.     image=self.friends_friend_code_entry_image  
1229. )  
1230. self.friends_friend_code_entry = tk.Entry(  
1231.     self,  
1232.     bd=0,  
1233.     bg="#617998",  
1234.     fg="#E3E7ED",  
1235.     highlightthickness=0  
1236. )  
1237. self.friends_friend_code_entry.place(  
1238.     x=30.0,  
1239.     y=150.0,  
1240.     width=279.0,  
1241.     height=40.0
```



```
1242.     )
1243.
1244.     self.copy_button_image = tk.PhotoImage(
1245.         file=relative_to_assets("add_friend_copy_button.png"))
1246.     copy_button = tk.Button(
1247.         self,
1248.         image=self.copy_button_image,
1249.         borderwidth=0,
1250.         highlightthickness=0,
1251.         command=lambda: self.copy_user_id_to_clipboard(),
1252.         relief="flat"
1253.     )
1254.     copy_button.place(
1255.         x=352.0,
1256.         y=63.0,
1257.         width=105.0,
1258.         height=42.0
1259.     )
1260.
1261.     self.add_friend_submit_button_image = tk.PhotoImage(
1262.         file=relative_to_assets("add_friend_submit_button.png"))
1263.     add_friend_submit_button = tk.Button(
1264.         self,
1265.         image=self.add_friend_submit_button_image,
1266.         borderwidth=0,
1267.         highlightthickness=0,
1268.         command=lambda: self.send_friend_request(),
1269.         relief="flat"
1270.     )
1271.     add_friend_submit_button.place(
1272.         x=352.0,
1273.         y=150.0,
1274.         width=105.0,
1275.         height=42.0
1276.     )
1277.
1278.     self.personal_friend_code = self.canvas.create_text(
1279.         19.0,
1280.         75.0,
1281.         anchor="nw",
1282.         text="",
1283.         fill="#FFFFFF",
1284.         font=("MontserratRoman Bold", 16 * -1)
1285.     )
1286.
1287.     # -----incoming friend request area-----
1288.
1289.     friend_request_box_header_text = self.canvas.create_text(
1290.         31.0,
1291.         264.0,
1292.         anchor="nw",
1293.         text=" Friend Requests",
1294.         fill="#FFFFFF",
1295.         font=("MontserratRoman Bold", 16 * -1)
1296.     )
1297.
1298.     self.friend_request_area = tk.Text(
1299.         self,
1300.         bd=0,
1301.         bg="#2A3441",
1302.         fg="#2A3441",
1303.         highlightthickness=0,
1304.         cursor='arrow',
1305.         state='disabled'
1306.     )
1307.
1308.     self.friend_request_area.place(
1309.         x=9.0,
1310.         y=294.0,
1311.         width=190,
```

```
1312.         height=201
1313.     )
1314.
1315.     sb = tk.Scrollbar(self.friend_request_area,
1316.                       command=self.friend_request_area.yview, width=20)
1317.     sb.pack(side='right', fill=tk.Y)
1318.     self.friend_request_area.configure(yscrollcommand=sb.set)
1319.     self.incoming_request_value = tk.StringVar(self.friend_request_area)
1320.
1321.     # accept button
1322.     self.accept_button_image = tk.PhotoImage(
1323.         file=relative_to_assets("add_friend_accept_button.png"))
1324.     accept_button = tk.Button(
1325.         self,
1326.         image=self.accept_button_image,
1327.         borderwidth=0,
1328.         highlightthickness=0,
1329.         command=lambda: self.accept_friend_request(),
1330.         relief="flat"
1331.     )
1332.     accept_button.place(
1333.         x=213.0,
1334.         y=294.0,
1335.         width=119.23919677734375,
1336.         height=42.0
1337.     )
1338.
1339.     # reject button
1340.
1341.     self.reject_button_image = tk.PhotoImage(
1342.         file=relative_to_assets("add_friend_reject_button.png"))
1343.     reject_button = tk.Button(
1344.         self,
1345.         image=self.reject_button_image,
1346.         borderwidth=0,
1347.         highlightthickness=0,
1348.         command=lambda: self.reject_friend_request(),
1349.         relief="flat"
1350.     )
1351.     reject_button.place(
1352.         x=213.0,
1353.         y=453.0,
1354.         width=119.23919677734375,
1355.         height=42.0
1356.     )
1357.
1358.     # -----current friends area-----
1359.
1360.     current_friends_box_header_text = self.canvas.create_text(
1361.         468.0,
1362.         264.0,
1363.         anchor="nw",
1364.         text="Friends",
1365.         fill="#FFFFFF",
1366.         font=("MontserratRoman Bold", 16 * -1)
1367.     )
1368.
1369.     self.friend_list_area = tk.Text(
1370.         self,
1371.         bd=0,
1372.         bg="#2A3441",
1373.         fg="#2A3441",
1374.         highlightthickness=0,
1375.         cursor='arrow',
1376.         state='disabled'
1377.     )
1378.     self.friend_list_area.place(
1379.         x=405,
1380.         y=294.0,
1381.         width=190,
```

```
1382.         height=201
1383.     )
1384.
1385.     sb = tk.Scrollbar(self.friend_list_area,
1386.                       command=self.friend_list_area.yview, width=20)
1387.     sb.pack(side='right', fill=tk.Y)
1388.     self.friend_list_area.configure(yscrollcommand=sb.set)
1389.     self.friend_list_value = tk.StringVar(self.friend_list_area)
1390.
1391.     # block button
1392.     self.add_friend_block_button_image = tk.PhotoImage(
1393.         file=relative_to_assets("add_friend_block_button.png"))
1394.     add_friend_block_button = tk.Button(
1395.         self,
1396.         image=self.add_friend_block_button_image,
1397.         borderwidth=0,
1398.         highlightthickness=0,
1399.         command=lambda: self.block_friend(),
1400.         relief="flat"
1401.     )
1402.     add_friend_block_button.place(
1403.         x=609.0,
1404.         y=294.0,
1405.         width=119.23919677734375,
1406.         height=42.0
1407.     )
1408.
1409.     # unblock button
1410.
1411.     self.add_friend_unblock_button_image = tk.PhotoImage(
1412.         file=relative_to_assets("add_friend_unblock_button.png"))
1413.     add_friend_unblock_button = tk.Button(
1414.         self,
1415.         image=self.add_friend_unblock_button_image,
1416.         borderwidth=0,
1417.         highlightthickness=0,
1418.         command=lambda: self.unblock_friend(),
1419.         relief="flat"
1420.     )
1421.     add_friend_unblock_button.place(
1422.         x=609.0,
1423.         y=450.0,
1424.         width=119.23919677734375,
1425.         height=42.0
1426.     )
1427.
1428.     # outgoing friend requests
1429.
1430.     pending_friend_request_box_header = self.canvas.create_text(
1431.         505.0,
1432.         38.0,
1433.         anchor="nw",
1434.         text="Pending",
1435.         fill="#FFFFFF",
1436.         font=("MontserratRoman Bold", 16 * -1)
1437.     )
1438.
1439.     self.pending_friends_area = tk.Text(
1440.         self,
1441.         bd=0,
1442.         bg="#2A3441",
1443.         fg="#2A3441",
1444.         highlightthickness=0,
1445.         cursor='arrow',
1446.         state='disabled'
1447.     )
1448.     self.pending_friends_area.place(
1449.         x=474,
1450.         y=63.0,
1451.         width=135,
```

```
1452.         height=145
1453.     )
1454.
1455.     pending_friend_scrollbar = tk.Scrollbar(self.pending_friends_area,
1456.                                             command=self.pending_friends_area.yview,
1457. width=20)
1458.     pending_friend_scrollbar.pack(side='right', fill=tk.Y)
1459.     self.pending_friends_area.configure(
1460.         yscrollcommand=pending_friend_scrollbar.set)
1461.
1462. def send_friend_request(self):
1463.     friend_code = self.friends_friend_code_entry.get()
1464.     self.canvas.itemconfig(self.friend_code_error_message, state='hidden')
1465.     self.canvas.itemconfig(self.friend_code_error_message, fill='#FF4747')
1466.
1467.     if friend_code == '' or friend_code == self.client.user_id:
1468.         self.canvas.itemconfig(
1469.             self.friend_code_error_message, text='Please enter a valid friend code')
1470.         self.canvas.itemconfig(
1471.             self.friend_code_error_message, state='normal')
1472.     elif self.client.check_if_user_is_already_friends(friend_code):
1473.         self.canvas.itemconfig(
1474.             self.friend_code_error_message, text='You are already friends with this
1475. person')
1476.         self.canvas.itemconfig(
1477.             self.friend_code_error_message, state='normal')
1478.     elif not self.client.check_if_friend_code_exists(friend_code):
1479.         self.canvas.itemconfig(self.friend_code_error_message,
1480.                               text='Friend code does not exist. Please enter a valid
1481. friend code')
1482.         self.canvas.itemconfig(
1483.             self.friend_code_error_message, state='normal')
1484.     else:
1485.         self.canvas.itemconfig(self.friend_code_error_message,
1486.                               text='Friend Request Sent')
1487.         self.canvas.itemconfig(
1488.             self.friend_code_error_message, fill='#FFFFFF')
1489.         self.canvas.itemconfig(
1490.             self.friend_code_error_message, state='normal')
1491.         self.client.send_friend_request(friend_code)
1492.
1493. def add_text(self):
1494.     self.canvas.itemconfig(self.personal_friend_code,
1495.                             text=self.client.user_id)
1496.
1497. def copy_user_id_to_clipboard(self):
1498.     t = tk.Tk()
1499.     t.withdraw()
1500.     t.clipboard_clear()
1501.     t.clipboard_append(self.client.user_id)
1502.     t.update() # now it stays on the clipboard after the window is closed
1503.     t.destroy()
1504.
1505. def add_radio_buttons(self):
1506.     self.create_friend_list()
1507.     self.create_friend_requests_list()
1508.     self.create_pending_friends_list()
1509.
1510. def create_friend_list(self):
1511.     for friend in self.client.friend_list:
1512.         friend_screen_name = friend[2]
1513.         friend_details = friend[0:2]
1514.         status = friend[3]
1515.         specifier_id = friend[4]
1516.         self.add_friend_radiobutton_to_ui(
1517.             friend_screen_name, friend_details, status, specifier_id)
1518.
1519. def add_friend_radiobutton_to_ui(self, txt, val, status, specifier_id):
1520.     state = 'normal'
1521.     colour = '#2A3441'
```

```
1519.         active = '#12161C'
1520.         if status == 'blk' and specifier_id == self.client.user_id: # you have blocked
1521.             then
1522.                 colour = '#FF4747'
1523.                 active = '#612a2a'
1524.             elif (status == 'blk' and specifier_id != self.client.user_id) or ('deleted
1525. account' in txt): # they have blocked you
1526.                 state = 'disabled'
1527.
1528.         btn = tk.Radiobutton(self.friend_list_area,
1529.                               text=txt,
1530.                               variable=self.friend_list_value,
1531.                               value=val,
1532.                               indicatoron=0,
1533.                               bg=colour,
1534.                               height=2,
1535.                               width=23,
1536.                               relief='flat',
1537.                               fg='#E3E7ED',
1538.                               selectcolor=active,
1539.                               command=lambda: self.set_active_button_value(btn),
1540.                               borderwidth=0,
1541.                               font=("MontserratRoman", 9, 'normal'),
1542.                               state=state
1543.                               )
1544.
1545.         self.friend_list_area.window_create('end', window=btn)
1546.         self.friend_list_area.insert('end', '\n')
1547.
1548.     def create_friend_requests_list(self):
1549.         for request in self.client.friend_request_list:
1550.             print(request)
1551.             friend_user_id = request[0]
1552.             friend_details = request[0:1]
1553.             self.add_friend_request_radiobutton_to_ui(
1554.                 friend_user_id, friend_details)
1555.
1556.     def update_friend_request_list(self, friend_user_id, friend_public_key):
1557.         friend_details = (friend_user_id, friend_public_key)
1558.         self.add_friend_request_radiobutton_to_ui(
1559.             friend_user_id, friend_details)
1560.
1561.     def add_friend_request_radiobutton_to_ui(self, txt, val):
1562.
1563.         btn = tk.Radiobutton(self.friend_request_area,
1564.                               text=txt,
1565.                               variable=self.incoming_request_value,
1566.                               value=val,
1567.                               indicatoron=0,
1568.                               bg='#2A3441',
1569.                               height=2,
1570.                               width=23,
1571.                               relief='flat',
1572.                               fg='#E3E7ED',
1573.                               selectcolor='#12161C',
1574.                               command=lambda: self.set_active_friend_request_button_value(
1575.                                   btn),
1576.                               borderwidth=0,
1577.                               font=("MontserratRoman", 9, 'normal')
1578.                               )
1579.
1580.         self.friend_request_area.window_create('end', window=btn)
1581.         self.friend_request_area.insert('end', '\n')
1582.
1583.     def set_active_friend_request_button_value(self, btn):
1584.         self.active_request_button = btn
1585.
1586.     def set_active_button_value(self, btn):
1587.         self.active_friend_button = btn
```

```
1587. def block_friend(self):
1588.     if self.active_friend_button.cget('bg') != '#FF4747':
1589.         self.active_friend_button.config(bg='#FF4747')
1590.         self.active_friend_button.config(selectcolor='#1c1212')
1591.         self.client.block_friend(
1592.             self.friend_list_value.get().split(' ')[0])
1593.     else:
1594.         print('User already blocked')
1595.
1596. def unblock_friend(self):
1597.     if self.active_friend_button.cget('bg') != '#2A3441':
1598.         self.active_friend_button.config(bg='#2A3441')
1599.         self.active_friend_button.config(selectcolor='#12161C')
1600.         self.client.unblock_friend(
1601.             self.friend_list_value.get().split(' ')[0])
1602.     else:
1603.         print('User already unblocked')
1604.
1605. def accept_friend_request(self):
1606.     """
1607.     - update personal database to be accepted
1608.     - send message to client to update their own database to fit
1609.     - delete radiobutton
1610.     """
1611.     friend_id = self.incoming_request_value.get().split(' ')[0]
1612.     self.client.accept_friend_request(friend_id, self.client.user_id)
1613.     self.active_request_button.destroy()
1614.
1615. def reject_friend_request(self):
1616.     """
1617.     - delete friend from personal database
1618.     - send message to client to delete their own db to fit
1619.     - delete radiobutton
1620.     """
1621.     friend_id = self.incoming_request_value.get().split(' ')[0]
1622.     self.client.reject_friend_request(friend_id)
1623.     self.active_request_button.destroy()
1624.
1625. def create_pending_friends_list(self):
1626.     for pending_request in self.client.pending_friend_list:
1627.         friend_user_id = pending_request
1628.         self.add_pending_friends_radiobutton_to_ui(friend_user_id)
1629.
1630. def update_pending_friends_list(self, friend_user_id):
1631.     self.add_pending_friends_radiobutton_to_ui(friend_user_id)
1632.
1633. def add_pending_friends_radiobutton_to_ui(self, friend_user_id):
1634.     btn = tk.Radiobutton(self.pending_friends_area,
1635.                           text=friend_user_id,
1636.                           indicatoron=0,
1637.                           bg='#2A3441',
1638.                           height=2,
1639.                           width=15,
1640.                           relief='flat',
1641.                           fg='#E3E7ED',
1642.                           selectcolor='#12161C',
1643.                           borderwidth=0,
1644.                           font=("MontserratRoman", 9, 'normal'),
1645.                           state='disabled'
1646.                           )
1647.
1648.     self.pending_friends_area.window_create('end', window=btn)
1649.     self.pending_friends_area.insert('end', '\n')
1650.
1651.
1652. class SettingsPage(tk.Frame):
1653.     def __init__(self, parent, window, client: secure_client.Client):
1654.         self.controller = parent
1655.         self.client = client
1656.         tk.Frame.__init__(self, window)
```

```
1657.
1658. def on_show(self):
1659.     """Function called when frame is shown"""
1660.     self.client.stop_listen()
1661.     self.create_and_place()
1662.     self.add_text()
1663.
1664. def create_and_place(self):
1665.
1666.     self.canvas = tk.Canvas(
1667.         self,
1668.         bg="#0A0C10",
1669.         height=504,
1670.         width=745,
1671.         bd=0,
1672.         highlightthickness=0,
1673.         relief="ridge"
1674.     )
1675.
1676.     self.canvas.place(x=0, y=0)
1677.     self.header_rectangle = self.canvas.create_rectangle(
1678.         0.0,
1679.         0.0,
1680.         745.0,
1681.         43.0,
1682.         fill="#12161C",
1683.         outline="")
1684.
1685.     settings_header_text = self.canvas.create_text(
1686.         16.0,
1687.         8.0,
1688.         anchor="nw",
1689.         text="Settings",
1690.         fill="#E3E7ED",
1691.         font=("MontserratRoman Bold", 24 * -1)
1692.     )
1693.
1694.     # CURRENT SCREEN NAME DISPLAY
1695.     current_screen_name_header_text = self.canvas.create_text(
1696.         16.0,
1697.         61.0,
1698.         anchor="nw",
1699.         text="Current screen name",
1700.         fill="#E3E7ED",
1701.         font=("MontserratRoman Bold", 24 * -1)
1702.     )
1703.
1704.     self.option_addcurrent_screen_name_image = tk.PhotoImage(
1705.         file=relative_to_assets("settings_current_screen_name.png"))
1706.     current_screen_name_bg = self.canvas.create_image(
1707.         176.5,
1708.         112.0,
1709.         image=self.option_addcurrent_screen_name_image
1710.     )
1711.     self.current_screen_name_text = self.canvas.create_text(
1712.         37.0,
1713.         99.0,
1714.         anchor="nw",
1715.         text="",
1716.         fill="#E3E7ED",
1717.         font=("MontserratRoman Bold", 24 * -1)
1718.     )
1719.
1720.     # CHANGE SCREEN NAME DISPLAY
1721.     change_screen_name_text = self.canvas.create_text(
1722.         16.0,
1723.         160.0,
1724.         anchor="nw",
1725.         text="Change screen name",
1726.         fill="#E3E7ED",
```

```
1727.         font=("MontserratRoman Bold", 24 * -1)
1728.     )
1729.
1730.     self.new_screen_name_entry_image = tk.PhotoImage(
1731.         file=relative_to_assets("settings_new_screen_name_entry.png"))
1732.     new_screen_name_entry_bg = self.canvas.create_image(
1733.         176.5,
1734.         213.0,
1735.         image=self.new_screen_name_entry_image
1736.     )
1737.
1738.     self.new_screen_name_entry = tk.Entry(
1739.         self,
1740.         bd=0,
1741.         bg="#617998",
1742.         fg="#E3E7ED",
1743.         highlightthickness=0,
1744.         font=("MontserratRoman")
1745.     )
1746.
1747.     self.new_screen_name_entry.place(
1748.         x=37.0,
1749.         y=192.0,
1750.         width=279.0,
1751.         height=40.0
1752.     )
1753.
1754.     self.new_screen_name_entry.bind(
1755.         '<Return>', lambda e: self.change_screen_name())
1756.
1757.     # CONFIRM SCREEN NAME BUTTON DISPLAY
1758.
1759.     self.confirm_screen_name_button_image = tk.PhotoImage(
1760.         file=relative_to_assets("settings_confirm_screen_name_button.png"))
1761.     confirm_screen_name_button = tk.Button(
1762.         self,
1763.         image=self.confirm_screen_name_button_image,
1764.         borderwidth=0,
1765.         highlightthickness=0,
1766.         command=lambda: self.change_screen_name(),
1767.         relief="flat",
1768.         activebackground='#0A0C10'
1769.     )
1770.     confirm_screen_name_button.place(
1771.         x=359.0,
1772.         y=193.0,
1773.         width=151.0,
1774.         height=42.0
1775.     )
1776.
1777.     # DELETE ACCOUNT BUTTON DISPLAY
1778.
1779.     self.delete_account_button_image = tk.PhotoImage(
1780.         file=relative_to_assets("settings_delete_account_button.png"))
1781.     delete_account_button = tk.Button(
1782.         self,
1783.         image=self.delete_account_button_image,
1784.         borderwidth=0,
1785.         highlightthickness=0,
1786.         command=self.popup_frame,
1787.         relief="flat",
1788.         activebackground='#0A0C10'
1789.     )
1790.     delete_account_button.place(
1791.         x=16.0,
1792.         y=436.0,
1793.         width=229.0,
1794.         height=42.0
1795.     )
1796.
```



```
1797.         # REQUEST DATA BUTTON DISPLAY
1798.
1799.         self.request_data_button_image = tk.PhotoImage(
1800.             file=relative_to_assets("settings_request_data_button.png"))
1801.         request_data_button = tk.Button(
1802.             self,
1803.             image=self.request_data_button_image,
1804.             borderwidth=0,
1805.             highlightthickness=0,
1806.             command=lambda: self.request_all_user_data(),
1807.             relief="flat",
1808.             activebackground='#0A0C10'
1809.         )
1810.         request_data_button.place(
1811.             x=443.0,
1812.             y=436.0,
1813.             width=302.0,
1814.             height=42.0
1815.         )
1816.
1817.         # BACK TO CHATS BUTTON DISPLAY
1818.
1819.         self.back_to_chats_button_image = tk.PhotoImage(
1820.             file=relative_to_assets("settings_back_to_chats_button.png"))
1821.         back_to_chats_button = tk.Button(
1822.             self,
1823.             image=self.back_to_chats_button_image,
1824.             borderwidth=0,
1825.             highlightthickness=0,
1826.             command=lambda: self.controller.show_frame(ChatPage),
1827.             relief="flat",
1828.             activebackground='#0A0C10'
1829.         )
1830.         back_to_chats_button.place(
1831.             x=640.0,
1832.             y=61.0,
1833.             width=95.0,
1834.             height=87.0
1835.         )
1836.         self.error_message = self.canvas.create_text(
1837.             16.0,
1838.             250.0,
1839.             anchor="nw",
1840.             text="",
1841.             fill="#FF4747",
1842.             font=("MontserratRoman Bold", 20 * -1),
1843.             state='hidden'
1844.         )
1845.
1846.         def add_text(self):
1847.             self.canvas.itemconfig(
1848.                 self.current_screen_name_text, text=self.client.screen_name)
1849.
1850.         def change_screen_name(self):
1851.             self.canvas.itemconfig(self.error_message, state='hidden')
1852.             if self.new_screen_name_entry.get() == self.client.screen_name:
1853.                 # if screen name is same as old screen name display error message
1854.                 self.canvas.itemconfig(self.error_message, state='normal',
1855.                                         text='Your new screen can not be the same as your
current screen name')
1856.             elif self.new_screen_name_entry.get() == '':
1857.                 self.canvas.itemconfig(self.error_message, state='normal',
1858.                                         text='Your new screen name can not be empty')
1859.             elif "deleted account" in self.new_screen_name_entry.get():
1860.                 self.canvas.itemconfig(self.error_message, state='normal',
1861.                                         text="This can't be your screen name")
1862.             else:
1863.                 self.canvas.itemconfig(
1864.                     self.current_screen_name_text, text=self.new_screen_name_entry.get())
1865.                 self.client.change_screen_name(self.new_screen_name_entry.get())
```

```
1866.
1867. def request_all_user_data(self):
1868.     path = self.client.get_output_path()
1869.     if not path:
1870.         # tk.messagebox.showerror(
1871.         #     title="Invalid Path!", message="Enter a valid output path.")
1872.         return
1873.     output_path = Path(f"{path}/user_data_dump.txt").expanduser().resolve()
1874.     if output_path.exists():
1875.         response = tk1.askyesno(
1876.             "Continue?",
1877.             f"Directory {path} already contains user_data_dump.txt\n"
1878.             "Do you want to continue and overwrite?")
1879.         if not response: # they said no
1880.             return
1881.     print('[Requesting user data]')
1882.     user_data = self.client.request_all_user_data()
1883.     try:
1884.         output_file = open(output_path, 'w')
1885.         output_file.write(user_data)
1886.         tk.messagebox.showinfo(
1887.             "Success!", f"User data successfully saved at {output_path}")
1888.     except:
1889.         tk.messagebox.showerror(
1890.             title="Error", message=f"Something went wrong when writing to
1891. {output_path}")
1892. # POPUP FRAME FUNCTIONS
1893.
1894. def popup_frame(self):
1895.     top = tk.Toplevel(self.controller)
1896.     top.protocol("WM_DELETE_WINDOW", self.popup_frame_error)
1897.     self.controller.protocol("WM_DELETE_WINDOW", self.popup_frame_error)
1898.     top.geometry("479x375")
1899.     top.resizable(False, False)
1900.     top.title("Confirm account deletion")
1901.     self.disable_all_buttons()
1902.
1903.     popup_canvas = tk.Canvas(
1904.         top,
1905.         bg="#0A0C10",
1906.         height=375,
1907.         width=479,
1908.         bd=0,
1909.         highlightthickness=0,
1910.         relief="ridge"
1911.     )
1912.
1913.     popup_canvas.place(x=0, y=0)
1914.     warning_header_text = popup_canvas.create_text(
1915.         58.0,
1916.         14.0,
1917.         anchor="nw",
1918.         text="Are you sure you want to \n delete your account?",
1919.         fill="#FF4747",
1920.         font=("MontserratRoman Bold", 32 * -1)
1921.     )
1922.
1923.     self.yes_button_image = tk.PhotoImage(
1924.         file=relative_to_assets("confirmation_popup_yes.png"))
1925.     yes_button = tk.Button(
1926.         top,
1927.         image=self.yes_button_image,
1928.         borderwidth=0,
1929.         highlightthickness=0,
1930.         command=lambda: self.yes_button_clicked(top),
1931.         relief="flat",
1932.         activebackground='#2A3441'
1933.     )
1934.     yes_button.place(
```

```
1935.         x=57.0,
1936.         y=164.0,
1937.         width=139.0,
1938.         height=48.0
1939.     )
1940.
1941.     self.no_button_image = tk.PhotoImage(
1942.         file=relative_to_assets("confirmation_popup_no.png"))
1943.     no_button = tk.Button(
1944.         top,
1945.         image=self.no_button_image,
1946.         borderwidth=0,
1947.         highlightthickness=0,
1948.         command=lambda: self.no_button_clicked(top),
1949.         relief="flat",
1950.         activebackground='#2A3441'
1951.     )
1952.     no_button.place(
1953.         x=279.0,
1954.         y=163.0,
1955.         width=139.0,
1956.         height=50.0
1957.     )
1958.
1959.     def popup_frame_error(self):
1960.         tk.messagebox.showerror('Error', 'Please select either yes or no')
1961.
1962.     def no_button_clicked(self, top):
1963.         top.destroy()
1964.         self.controller.protocol("WM_DELETE_WINDOW", self.controller.on_close)
1965.         self.enable_all_buttons()
1966.
1967.     def yes_button_clicked(self, top):
1968.         print('[DELETING ACCOUNT]')
1969.         if self.client.delete_account():
1970.             self.controller.on_close(True)
1971.         else:
1972.             tk.messagebox.showerror(
1973.                 title="Account Deletion Error", message="Something went wrong when
1974. deleting your account please try again later")
1975.             self.controller.protocol(
1976.                 "WM_DELETE_WINDOW", self.controller.on_close)
1977.             self.enable_all_buttons()
1978.
1979.     def disable_all_buttons(self):
1980.         for widget in self.wininfo_children():
1981.             if isinstance(widget, tk.Button):
1982.                 try:
1983.                     widget.config(state='disabled')
1984.                 except Exception as e:
1985.                     print(e)
1986.
1987.     def enable_all_buttons(self):
1988.         for widget in self.wininfo_children():
1989.             if isinstance(widget, tk.Button):
1990.                 widget.config(state='normal')
1991.
1992. # Driver Code
1993. app = UIController()
1994. app.resizable(False, False)
1995. app.protocol("WM_DELETE_WINDOW", app.on_close)
1996. app.mainloop()
1997.
```