FINAL PROJECT REPORT

IN NETWORK

Submitted to: Dr. Ahmad Elhajj

Submitted by:

Sajed Hamdan- 202202006

Abdallah Makke- 202002471

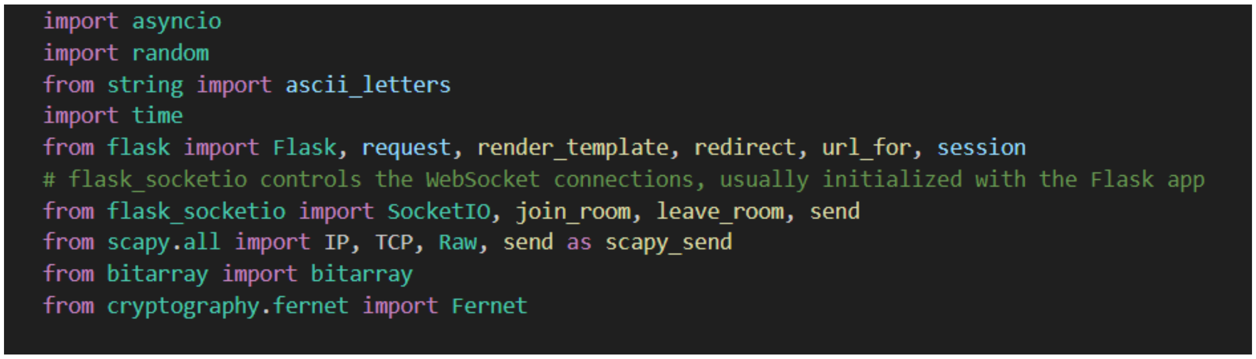
Marwa Zeineddine- 202201705

Suzan Skafi- 201901379

Code Explanation

The code is divided into several sections, each responsible for different aspects of the application's functionality.

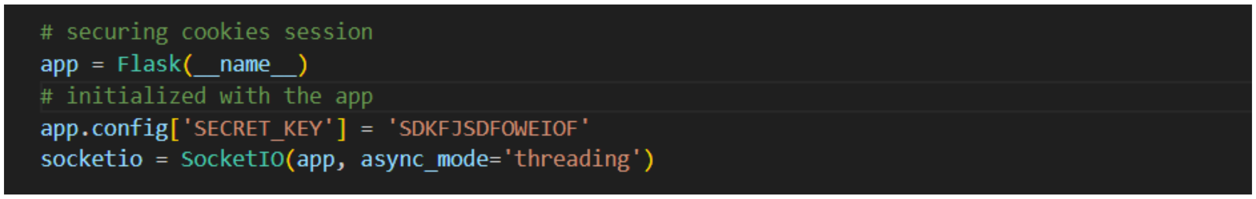
1. Imports and Initial Setup



```python
import asyncio
import random
from string import ascii_letters
import time
from flask import Flask, request, render_template, redirect, url_for, session
# flask_socketio controls the WebSocket connections, usually initialized with the Flask app
from flask_socketio import SocketIO, join_room, leave_room, send
from scapy.all import IP, TCP, Raw, send as scapy_send
from bitarray import bitarray
from cryptography.fernet import Fernet
```

This section imports all the necessary libraries. asyncio is used to run things asynchronously. random and string help generate unique room codes. time provides time-related functions. flask and flask_socketio are used for web server and real-time communication functionalities. The scapy library deals with network packet manipulation. The bitarray is used for efficient bit array operations. In addition, the cryptography library provides encryption and decryption functionalities to ensure messages are transmitted safely between users.

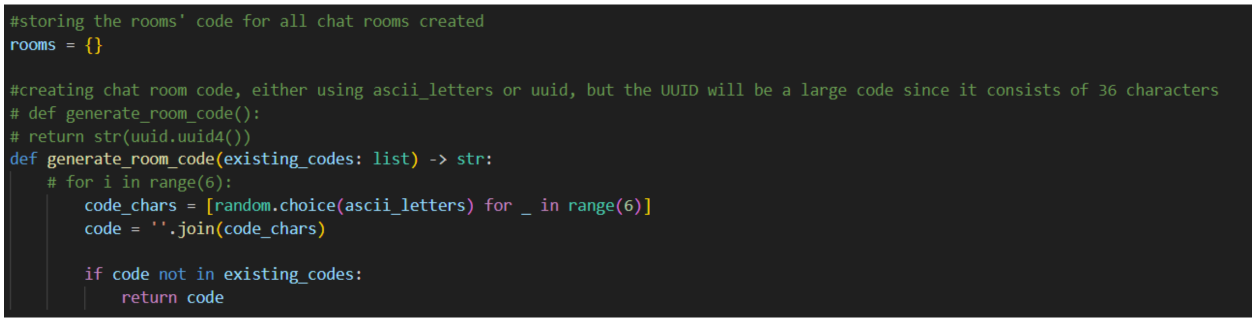
2. Flask Application and SocketIO Setup



```python
# securing cookies session
app = Flask(__name__)
# initialized with the app
app.config['SECRET_KEY'] = 'SDKFJSDFOWEIOF'
socketio = SocketIO(app, async_mode='threading')
```

This sets up the base for the initialization of a Flask app and then assigns a secret key to the session. It also defines the usage of threading by Flask-SocketIO for asynchronous operations.

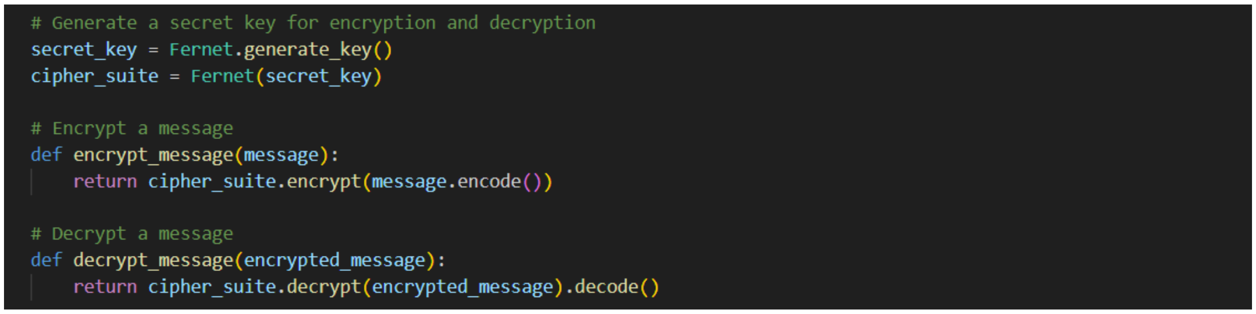
3. Room Management



```
#storing the rooms' code for all chat rooms created
rooms = {}

#creating chat room code, either using ascii_letters or uuid, but the UUID will be a large code since it consists of 36 characters
# def generate_room_code():
# return str(uuid.uuid4())
def generate_room_code(existing_codes: list) -> str:
    # for i in range(6):
        code_chars = [random.choice(ascii_letters) for _ in range(6)]
        code = ''.join(code_chars)

        if code not in existing_codes:
            return code
```

- rooms: A dictionary to store chat rooms and their details.
- generate_room_code: Generates a unique 6-character room code using random ASCII letters, ensuring no duplicates.

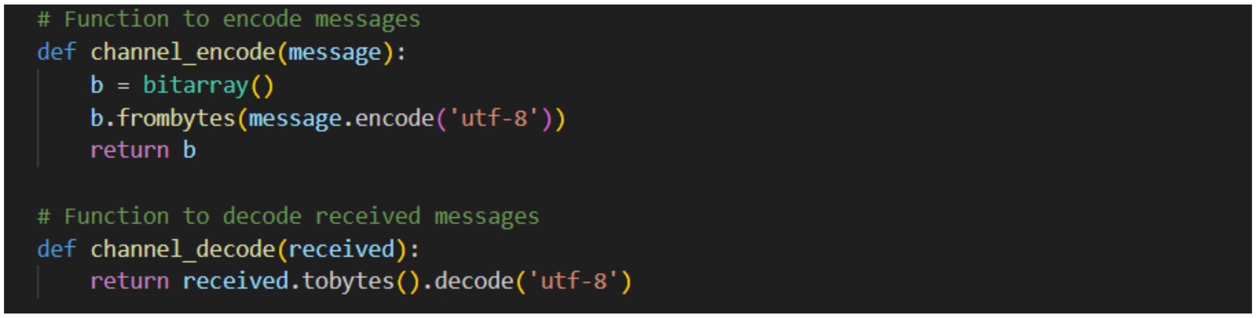
4. Encryption and Decryption



```
# Generate a secret key for encryption and decryption
secret_key = Fernet.generate_key()
cipher_suite = Fernet(secret_key)

# Encrypt a message
def encrypt_message(message):
    return cipher_suite.encrypt(message.encode())

# Decrypt a message
def decrypt_message(encrypted_message):
    return cipher_suite.decrypt(encrypted_message).decode()
```

In this section, secret keys for encryption are generated with the Fernet module from the cryptography library. Functions for encryption and decryption of messages with this key are defined to ensure secured communication between users.

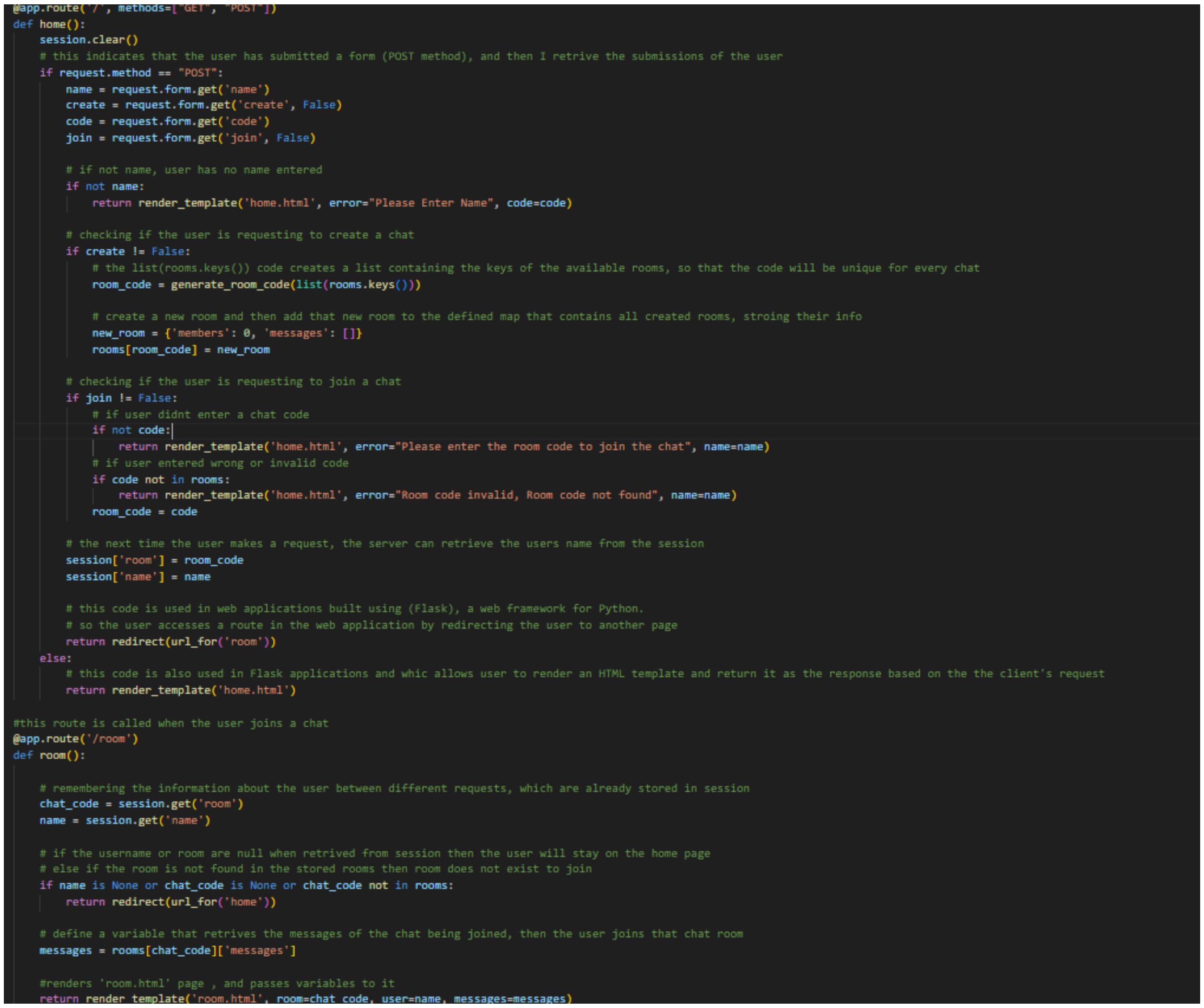
5. Message Encoding and Decoding



```
# Function to encode messages
def channel_encode(message):
    b = bitarray()
    b.frombytes(message.encode('utf-8'))
    return b

# Function to decode received messages
def channel_decode(received):
    return received.tobytes().decode('utf-8')
```

The messages exchanged by the users are encoded as bit arrays before transmission and decoded upon reception. The encoding of messages and decoding, with the facilitation of the bitarray library, increases the efficiency of network transmission, hence faster and more reliable communication.

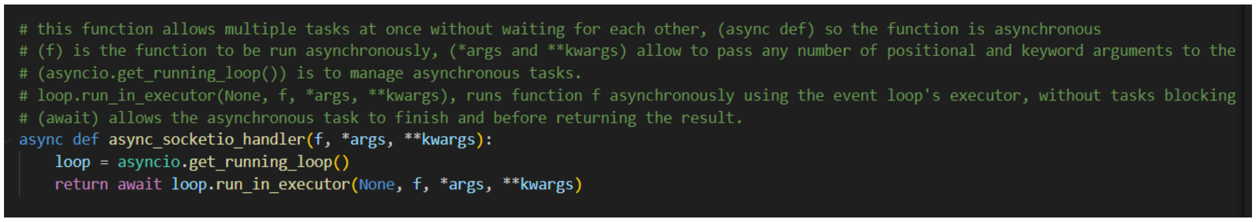
6. HTTP Routes



```python
@app.route('/', methods=["GET", "POST"])
def home():
    session.clear()
    # this indicates that the user has submitted a form (POST method), and then I retrive the submissions of the user
    if request.method == "POST":
        name = request.form.get('name')
        create = request.form.get('create', False)
        code = request.form.get('code')
        join = request.form.get('join', False)

        # if not name, user has no name entered
        if not name:
            return render_template('home.html', error="Please Enter Name", code=code)

        # checking if the user is requesting to create a chat
        if create != False:
            # the list(rooms.keys()) code creates a list containing the keys of the available rooms, so that the code will be unique for every chat
            room_code = generate_room_code(list(rooms.keys()))

            # create a new room and then add that new room to the defined map that contains all created rooms, stroing their info
            new_room = {'members': 0, 'messages': []}
            rooms[room_code] = new_room

        # checking if the user is requesting to join a chat
        if join != False:
            # if user didnt enter a chat code
            if not code:
                return render_template('home.html', error="Please enter the room code to join the chat", name=name)
            # if user entered wrong or invalid code
            if code not in rooms:
                return render_template('home.html', error="Room code invalid, Room code not found", name=name)
            room_code = code

        # the next time the user makes a request, the server can retrieve the users name from the session
        session['room'] = room_code
        session['name'] = name

        # this code is used in web applications built using (Flask), a web framework for Python.
        # so the user accesses a route in the web application by redirecting the user to another page
        return redirect(url_for('room'))
    else:
        # this code is also used in Flask applications and whic allows user to render an HTML template and return it as the response based on the the client's request
        return render_template('home.html')

#this route is called when the user joins a chat
@app.route('/room')
def room():

    # remembering the information about the user between different requests, which are already stored in session
    chat_code = session.get('room')
    name = session.get('name')

    # if the username or room are null when retrived from session then the user will stay on the home page
    # else if the room is not found in the stored rooms then room does not exist to join
    if name is None or chat_code is None or chat_code not in rooms:
        return redirect(url_for('home'))

    # define a variable that retrives the messages of the chat being joined, then the user joins that chat room
    messages = rooms[chat_code]['messages']

    #renders 'room.html' page , and passes variables to it
    return render_template('room.html', room=chat_code, user=name, messages=messages)
```

This section defines HTTP routes to handle user requests. The / route handles a request to the home page, where users can create or join chat rooms. It collects user inputs: name, create/join option, and room code if applicable, upon the form submission. Based on the user's action, this redirects either to the chat room page (/room) or displays an error. The /room route renders the chat room page, listing messages and enabling real-time chat.

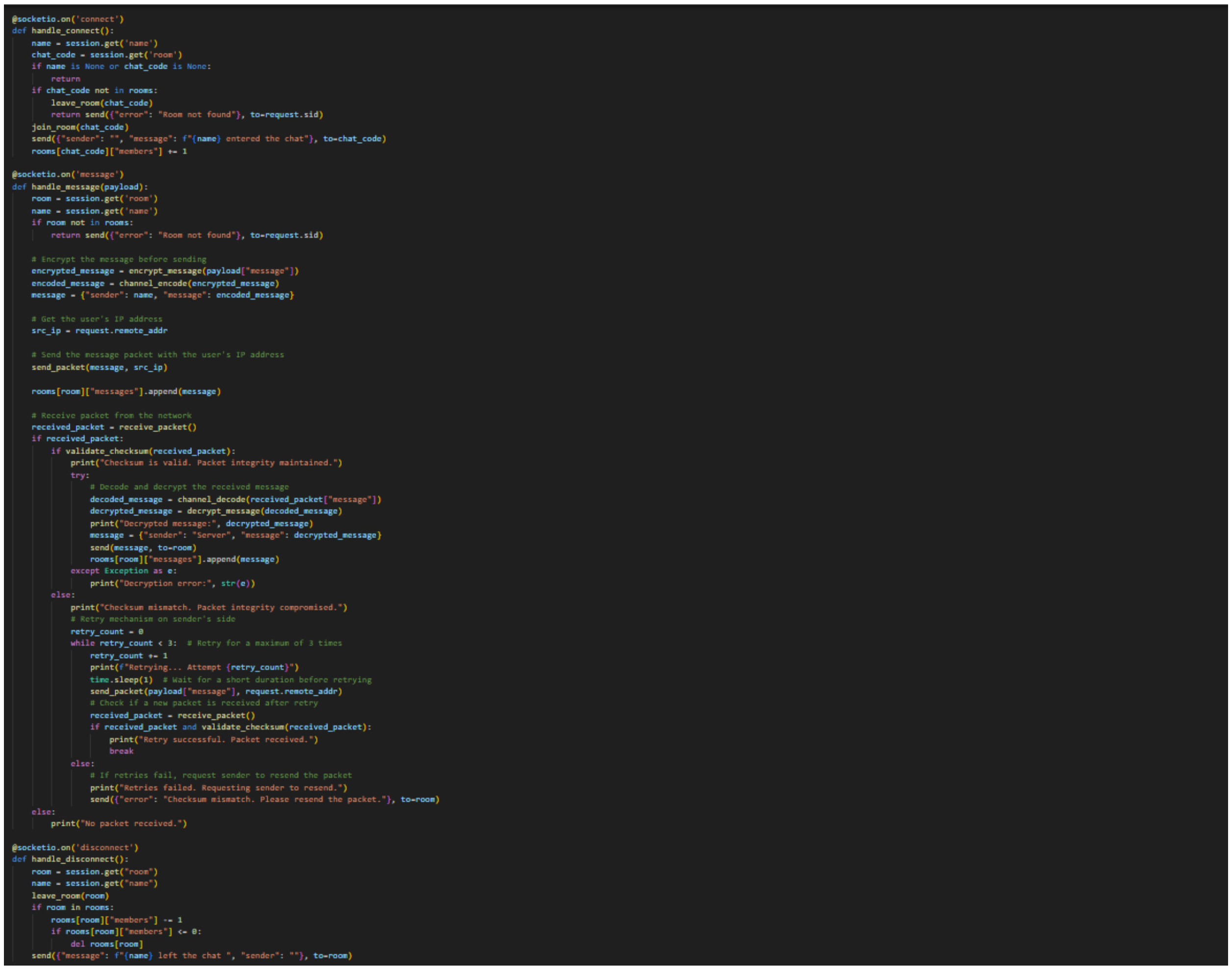
7. Asynchronous Task Handling



```python
# this function allows multiple tasks at once without waiting for each other, (async def) so the function is asynchronous
# (f) is the function to be run asynchronously, (*args and **kwargs) allow to pass any number of positional and keyword arguments to the
# (asyncio.get_running_loop()) is to manage asynchronous tasks.
# loop.run_in_executor(None, f, *args, **kwargs), runs function f asynchronously using the event loop's executor, without tasks blocking
# (await) allows the asynchronous task to finish and before returning the result.
async def async_socketio_handler(f, *args, **kwargs):
    loop = asyncio.get_running_loop()
    return await loop.run_in_executor(None, f, *args, **kwargs)
```

In this section, the asynchronous function called async_socketio_handler has been defined to run tasks concurrently without blocking the main program. In this function, the asyncio library is used to run tasks asynchronously, which makes the execution of events over WebSockets flow smoothly for 'connect', 'send', 'message', and 'receive' operations.

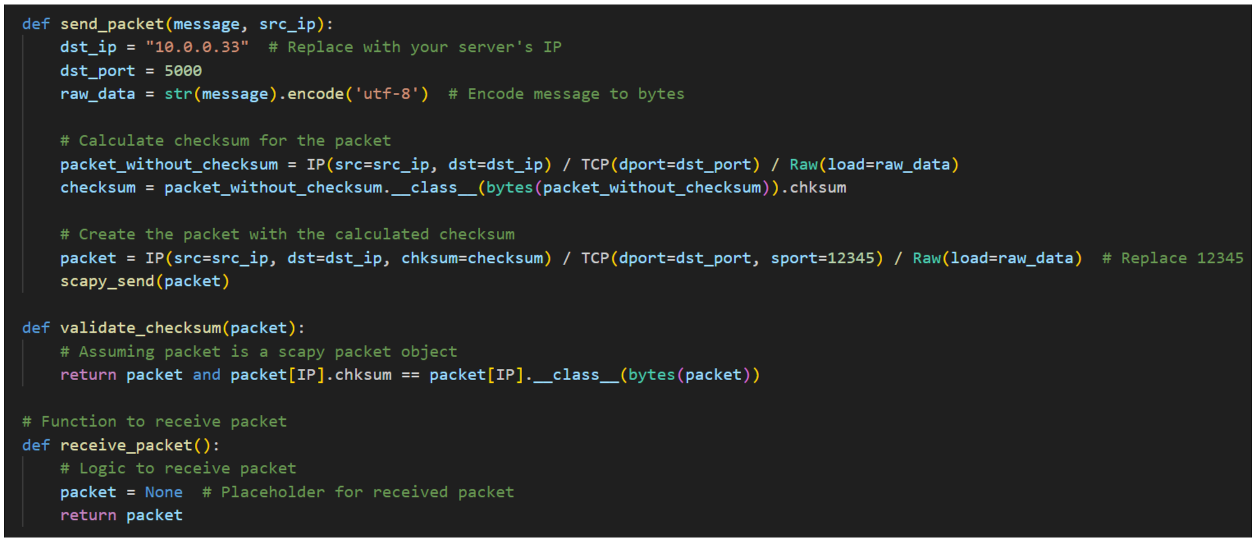
8. WebSocket Event Handlers



```python
@socketio.on('connect')
def handle_connect():
    name = session.get('name')
    chat_code = session.get('room')
    if name is None or chat_code is None:
        return
    if chat_code not in rooms:
        leave_room(chat_code)
        return send({"error": "Room not found"}, to=request.sid)
    join_room(chat_code)
    send({"sender": "", "message": f"{name} entered the chat"}, to=chat_code)
    rooms[chat_code]["members"] += 1

@socketio.on('message')
def handle_message(payload):
    room = session.get('room')
    name = session.get('name')
    if room not in rooms:
        return send({"error": "Room not found"}, to=request.sid)

    # Encrypt the message before sending
    encrypted_message = encrypt_message(payload["message"])
    encoded_message = channel_encode(encrypted_message)
    message = {"sender": name, "message": encoded_message}

    # Get the user's IP address
    src_ip = request.remote_addr

    # Send the message packet with the user's IP address
    send_packet(message, src_ip)

    rooms[room]["messages"].append(message)

    # Receive packet from the network
    received_packet = receive_packet()
    if received_packet:
        if validate_checksum(received_packet):
            print("Checksum is valid. Packet integrity maintained.")
            try:
                # Decode and decrypt the received message
                decoded_message = channel_decode(received_packet["message"])
                decrypted_message = decrypt_message(decoded_message)
                print("Decrypted message:", decrypted_message)
                message = {"sender": "Server", "message": decrypted_message}
                send(message, to=room)
                rooms[room]["messages"].append(message)
            except Exception as e:
                print("Decryption error:", str(e))
        else:
            print("Checksum mismatch. Packet integrity compromised.")
            # Retry mechanism on sender's side
            retry_count = 0
            while retry_count < 3:  # Retry for a maximum of 3 times
                retry_count += 1
                print(f"Retrying... Attempt {retry_count}")
                time.sleep(1)  # Wait for a short duration before retrying
                send_packet(payload["message"], request.remote_addr)
                # Check if a new packet is received after retry
                received_packet = receive_packet()
                if received_packet and validate_checksum(received_packet):
                    print("Retry successful. Packet received.")
                    break
            else:
                # If retries fail, request sender to resend the packet
                print("Retries failed. Requesting sender to resend.")
                send({"error": "Checksum mismatch. Please resend the packet."}, to=room)
    else:
        print("No packet received.")

@socketio.on('disconnect')
def handle_disconnect():
    room = session.get("room")
    name = session.get("name")
    leave_room(room)
    if room in rooms:
        rooms[room]["members"] -= 1
        if rooms[room]["members"] <= 0:
            del rooms[room]
    send({"message": f"{name} left the chat ", "sender": ""}, to=room)
```

WebSocket event handlers handle such things as the connection to WebSocket, sending and receiving messages, and disconnections from the server. @socketio.on('connect'), @socketio.on('message'), and @socketio.on('disconnect') define functions to handle these events. These functions would ensure message encryption, encoding, and sending, validate message integrity using checksums, and implement a retry mechanism for failed transmissions.

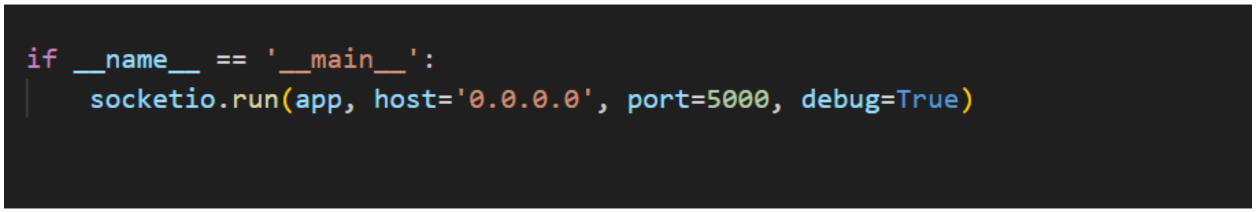
9. Network Packet Handling



```python
def send_packet(message, src_ip):
    dst_ip = "10.0.0.33"  # Replace with your server's IP
    dst_port = 5000
    raw_data = str(message).encode('utf-8')  # Encode message to bytes

    # Calculate checksum for the packet
    packet_without_checksum = IP(src=src_ip, dst=dst_ip) / TCP(dport=dst_port) / Raw(load=raw_data)
    checksum = packet_without_checksum.__class__(bytes(packet_without_checksum)).chksum

    # Create the packet with the calculated checksum
    packet = IP(src=src_ip, dst=dst_ip, chksum=checksum) / TCP(dport=dst_port, sport=12345) / Raw(load=raw_data)  # Replace 12345
    scapy_send(packet)

def validate_checksum(packet):
    # Assuming packet is a scapy packet object
    return packet and packet[IP].chksum == packet[IP].__class__(bytes(packet))

# Function to receive packet
def receive_packet():
    # Logic to receive packet
    packet = None  # Placeholder for received packet
    return packet
```

This section defines functions for sending and receiving network packets using Scapy. The send_packet function constructs and sends network packets containing encrypted messages, ensuring secure communication. The validate_checksum function checks the checksum of the received packets for ensuring message integrity. Apart from this, the receive_packet function is a placeholder to receive the packets that can be defined by the user based on the requirements.

10. Running the Application

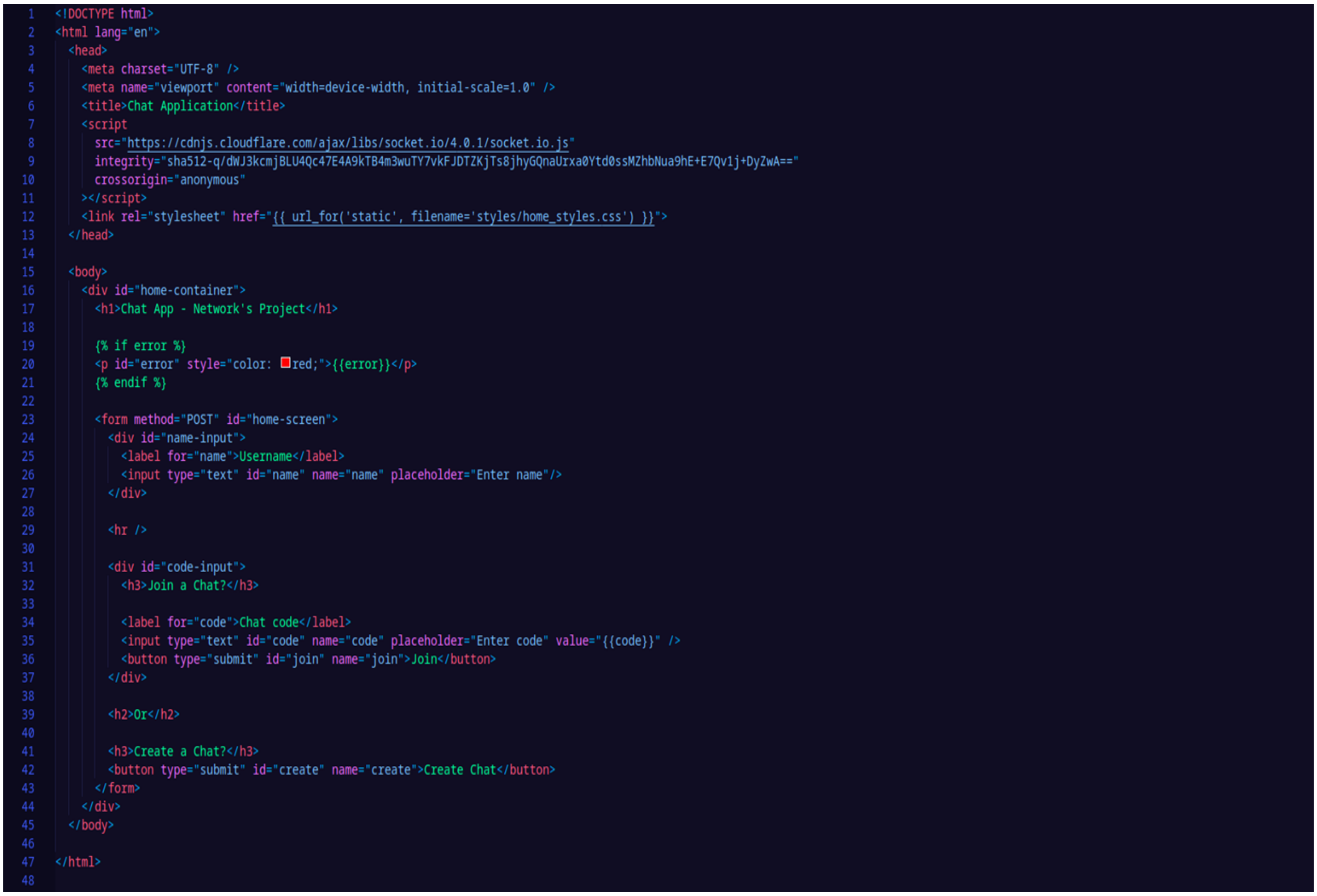


```python
if __name__ == '__main__':
    socketio.run(app, host='0.0.0.0', port=5000, debug=True)
```

Finally, the application is run using the socketio.run() method, which starts up the underlying Flask application with support for SocketIO. It listens on all network interfaces on port 5000, so users can access this chat application through their web browser and start communicating in a secure, real-time environment.

Html Code:

home.html



```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Chat Application</title>
    <script
      src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.0.1/socket.io.js"
      integrity="sha512-q/dWJ3kcmjBLU4Qc47E4A9kTB4m3wuTY7vkFJDTZKjTs8jhyGQnaUrxa0Ytd0ssMZhbNua9hE+E7Qv1j+DyZwA=="
      crossorigin="anonymous"
    ></script>
    <link rel="stylesheet" href="{{ url_for('static', filename='styles/home_styles.css') }}">
  </head>

  <body>
    <div id="home-container">
      <h1>Chat App - Network's Project</h1>

      {% if error %}
      <p id="error" style="color: 🟥red;">{{error}}</p>
      {% endif %}

      <form method="POST" id="home-screen">
        <div id="name-input">
          <label for="name">Username</label>
          <input type="text" id="name" name="name" placeholder="Enter name"/>
        </div>

        <hr />

        <div id="code-input">
          <h3>Join a Chat?</h3>

          <label for="code">Chat code</label>
          <input type="text" id="code" name="code" placeholder="Enter code" value="{{code}}" />
          <button type="submit" id="join" name="join">Join</button>
        </div>

        <h2>Or</h2>

        <h3>Create a Chat?</h3>
        <button type="submit" id="create" name="create">Create Chat</button>
      </form>
    </div>
  </body>

</html>
```

(<div id="home-container">): This <div> element serves as the main container for the content of the webpage.

({% if error %}): This block is used for conditional templating in Flask or similar frameworks. It checks if an "error" variable exists.

(<p id="error" style="color: red;">{{error}}</p>): If the "error" variable exists, this <p> element will display the error message in red. The content of the error message is based on the checking done in the 'main.py'.

(<form method="POST" id="home-screen">):

The form element is used to collect user input. It has a label and input tag for the username ,

(<label for="name">Username</label> and <input type="text" id="name" name="name" placeholder="Enter name"/>).

Another label and input tag for the Chat Code,

(<label for="code">Chat code</label> and <input type="text" id="code" name="code" placeholder="Enter code" value="{{code}}" />).

And for the chat code, there is a join button that when,

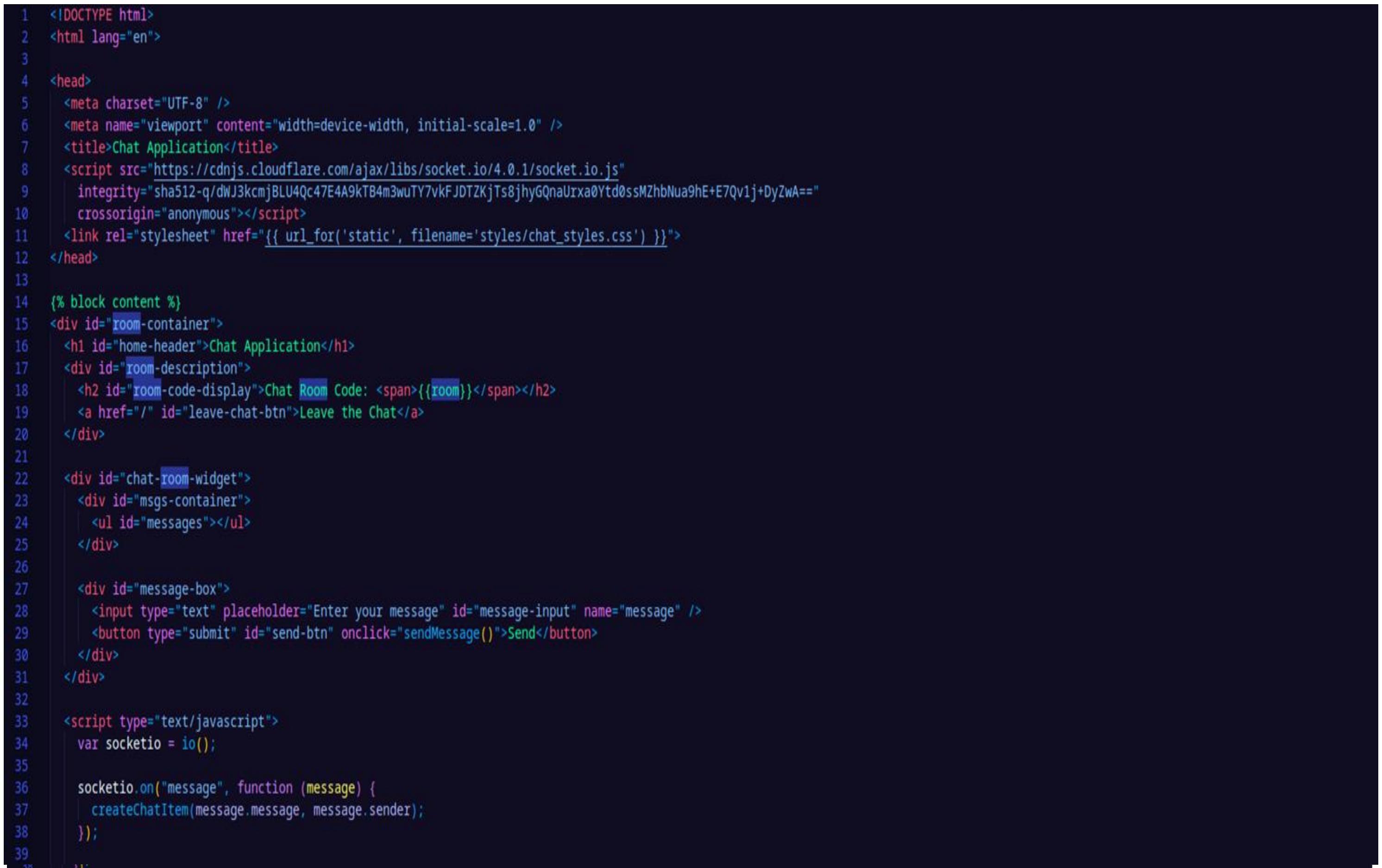
(<button type="submit" id="join" name="join">Join</button>).

When clicking this button it submits the form data to join the specified chat.

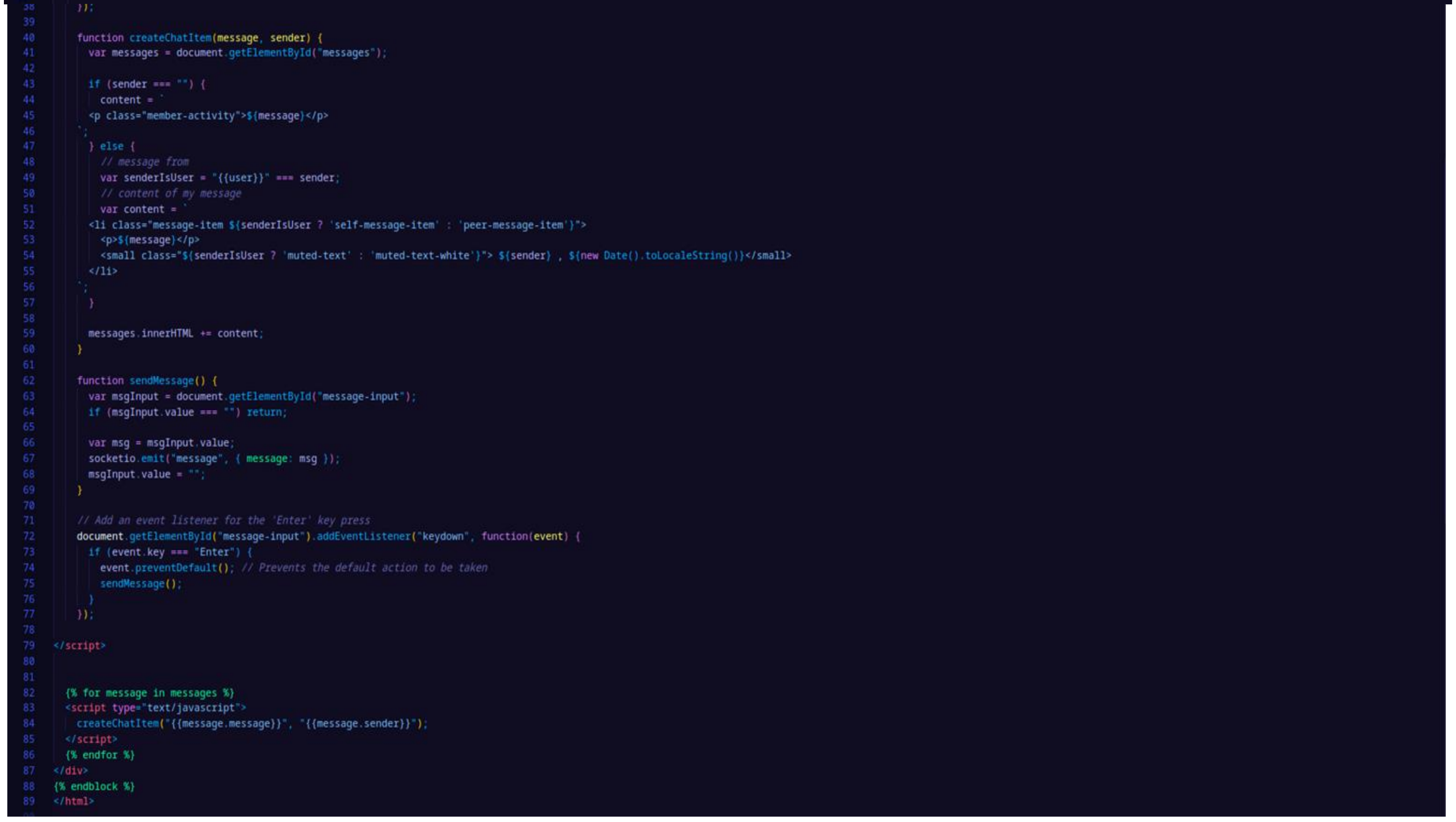
If the user decides to create a chat, it submits the form data to create a new chat,

(<button type="submit" id="create" name="create">Create Chat</button>).

Room html



```html
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Chat Application</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.0.1/socket.io.js"
    integrity="sha512-q/dWJ3kcmjBLU4Qc47E4A9kTB4m3wuTY7vkFJDTZKjTs8jhyGQnaUrxa0Ytd0ssMZhbNua9hE+E7Qv1j+DyZwA=="
    crossorigin="anonymous"></script>
  <link rel="stylesheet" href="{{ url_for('static', filename='styles/chat_styles.css') }}">
</head>

{% block content %}
<div id="room-container">
  <h1 id="home-header">Chat Application</h1>
  <div id="room-description">
    <h2 id="room-code-display">Chat Room Code: <span>{{room}}</span></h2>
    <a href="/" id="leave-chat-btn">Leave the Chat</a>
  </div>

  <div id="chat-room-widget">
    <div id="msgs-container">
      <ul id="messages"></ul>
    </div>

    <div id="message-box">
      <input type="text" placeholder="Enter your message" id="message-input" name="message" />
      <button type="submit" id="send-btn" onclick="sendMessage()">Send</button>
    </div>
  </div>

  <script type="text/javascript">
    var socketio = io();

    socketio.on("message", function (message) {
      createChatItem(message.message, message.sender);
    });

```



```html
  });

  function createChatItem(message, sender) {
    var messages = document.getElementById("messages");

    if (sender === "") {
      content = `
    <p class="member-activity">${message}</p>
  `;
    } else {
      // message from
      var senderIsUser = "{{user}}" === sender;
      // content of my message
      var content = `
    <li class="message-item ${senderIsUser ? 'self-message-item' : 'peer-message-item'}">
      <p>${message}</p>
      <small class="${senderIsUser ? 'muted-text' : 'muted-text-white'}"> ${sender} , ${new Date().toLocaleString()}</small>
    </li>
  `;
    }

    messages.innerHTML += content;
  }

  function sendMessage() {
    var msgInput = document.getElementById("message-input");
    if (msgInput.value === "") return;

    var msg = msgInput.value;
    socketio.emit("message", { message: msg });
    msgInput.value = "";
  }

  // Add an event listener for the 'Enter' key press
  document.getElementById("message-input").addEventListener("keydown", function(event) {
    if (event.key === "Enter") {
      event.preventDefault(); // Prevents the default action to be taken
      sendMessage();
    }
  });

</script>


  {% for message in messages %}
  <script type="text/javascript">
    createChatItem("{{message.message}}", "{{message.sender}}");
  </script>
  {% endfor %}
</div>
{% endblock %}
</html>
```

(<div id="room-container">):

This element serves as the main container for the chat room interface.

(<div id=”room-description">):

Contains information about the current chat room, such as the room code,

(<h2 id="room-code-display">Chat Room Code: <span>{{room}}</span></h2>**).**

And has a Leave Chat Link,

(<a href="/" id="leave-chat-btn">Leave the Chat</a>**).**

(<div id="chat-room-widget">):

Contains the components for sending and displaying chat messages.

The Messages Container (<div id="msgs-container">**), contains a** container for displaying chat messages, these messages are listed inside a <ul> element.

Message Box (<div id="message-box">**), a**llowing users to input messages, through an input element **(**<input type="text" placeholder="Enter your message" id="message-input" name="message" />**),** and a send Button, **(**<button type="submit" id="send-btn" onclick="sendMessage()">Send</button>**)**, that triggers a sendFunction() when triggered(onclick=”sendMessage()”).

JavaScript Code:

var socketio = io(): Initializes a Socket.IO connection. This creates a WebSocket connection to the server, (Javascript Library) allowing real-time bidirectional communication.

socketio.on("message", function (message) { }): sets up an event listener for the "message" event. When the server emits a "message" event, this function is triggered. Inside this function there is a createChatItem(message.message, message.sender) that calls the createChatItem function with the received message and sender information. This function is responsible for displaying the received message in the chat interface.

createChatItem(message, sender) {}: Defines the createChatItem function, which generates HTML elements to display chat messages.

It first retrieves the <ul> element with the ID "messages" where chat messages are displayed.

Depending on whether the message is from the current user or another user, it creates HTML content accordingly. If the sender is empty (indicating a system message), it displays the message differently.

The generated HTML content is then appended to the "messages" container.

sendMessage() {}: Handles sending messages to the server and retrieves the message input from the DOM.

If the message is not empty, it emits a "message" event to the server using the Socket.IO connection, along with the message content. After sending the message, it clears the input field.

document.getElementById("message-input").addEventListener("keydown", function(event) {}): Listens for the "keydown" event on the message input field.

When the "Enter" key is pressed, it prevents the default action (submitting the form) and calls the sendMessage function, allowing users to send messages by pressing "Enter".

({% for message in messages %} block):

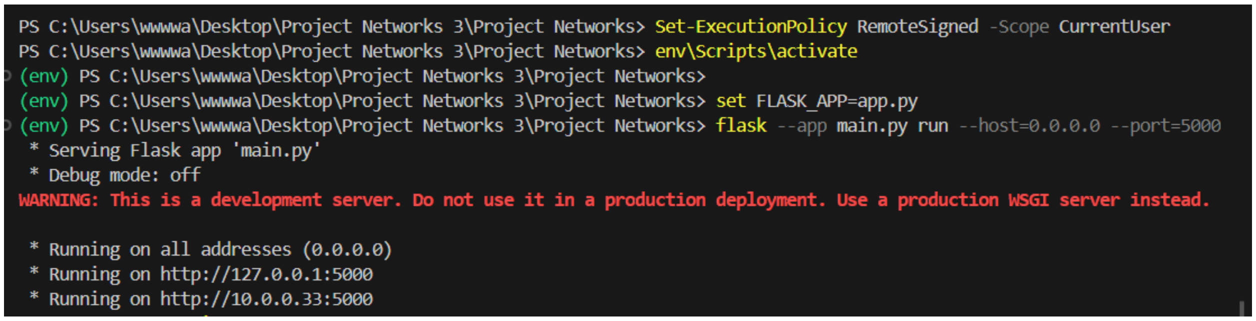
Utilizes Flask's templating system to render chat messages dynamically and iterates over a list of messages and invokes the createChatItem function to display each message in the chat interface.

(<script type="text/javascript">):

Initializes Socket.IO for real-time communication and defines functions for sending and receiving messages.

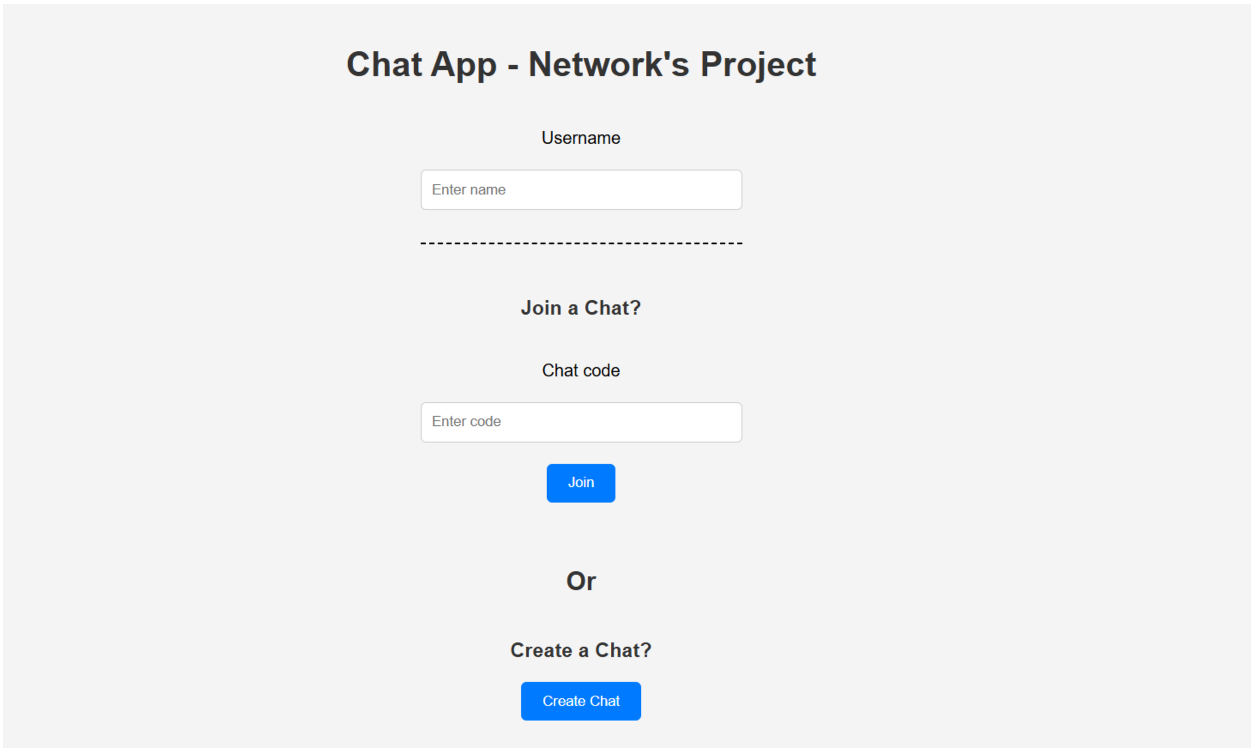
Adding an event listener to send messages when the "Enter" key is pressed.

Results:

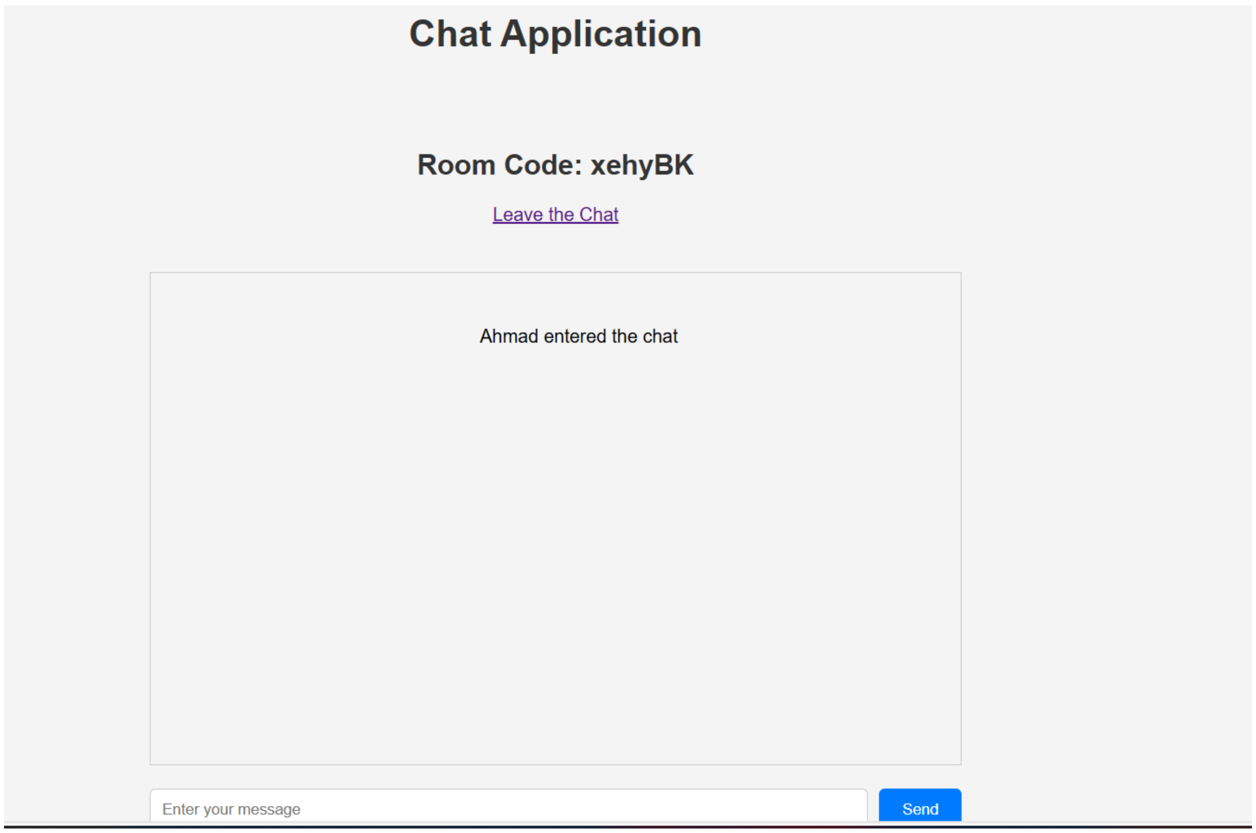


```
PS C:\Users\wwwwa\Desktop\Project Networks 3\Project Networks> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
PS C:\Users\wwwwa\Desktop\Project Networks 3\Project Networks> env\Scripts\activate
(env) PS C:\Users\wwwwa\Desktop\Project Networks 3\Project Networks>
(env) PS C:\Users\wwwwa\Desktop\Project Networks 3\Project Networks> set FLASK_APP=app.py
(env) PS C:\Users\wwwwa\Desktop\Project Networks 3\Project Networks> flask --app main.py run --host=0.0.0.0 --port=5000
 * Serving Flask app 'main.py'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.

 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://10.0.0.33:5000
```

After activating the environment and setting the flask. We reached two IP addresses for the server. The first is the loopback address and the other that we will use to now connect devices on same wifi.



In the above figure, we see the homepage where the user either creates a room or joins a one.

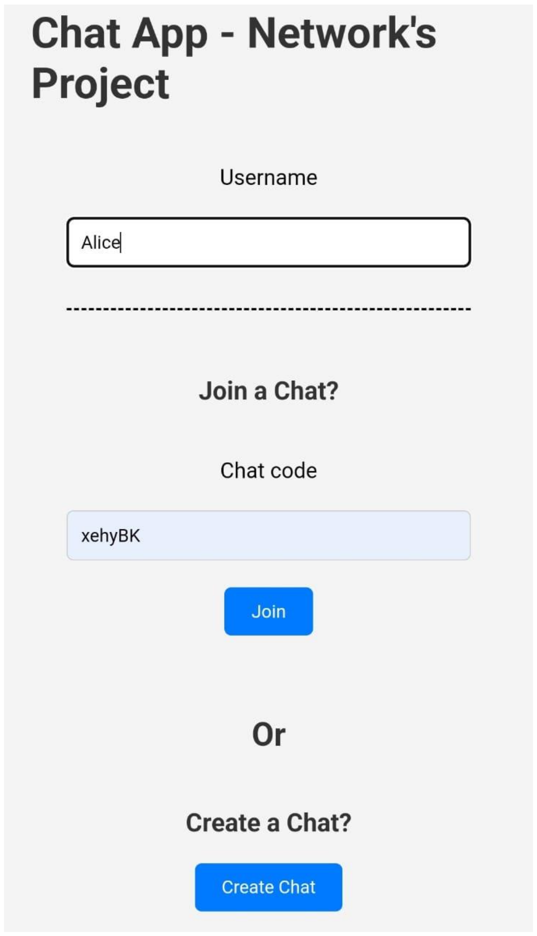


Ahmad entered the chat

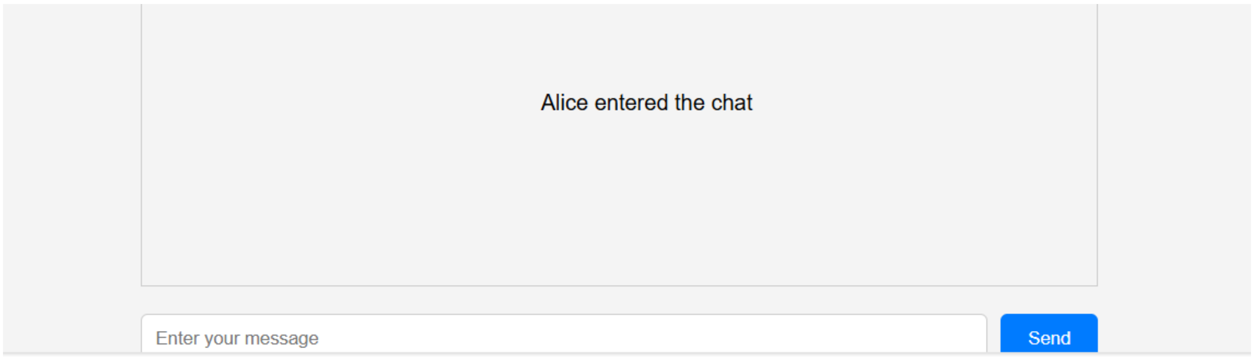
A user named Ahmad created a room and had a code of xehyBK. Ahmad is using a laptop.

Alice joined using her phone on the same wifi of Ahmad with url http://10.0.0.33:5000

Or by simply writing 10.0.0.33

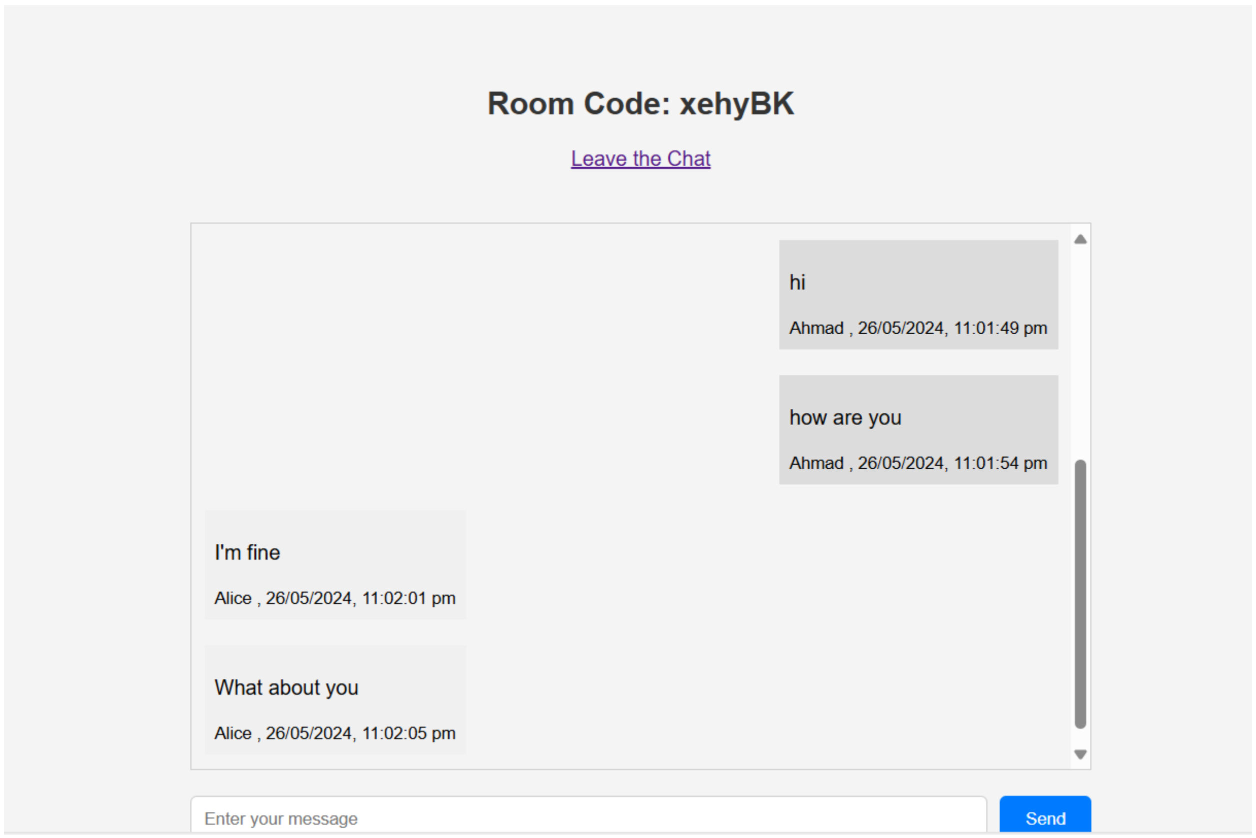


She wrote her name, inserted the code and pressed join.



Alice entered the chat

Ahmad saw that she entered the chat and sent hi.



Room Code: xehyBK

Leave the Chat

hi

Ahmad , 26/05/2024, 11:01:49 pm

how are you

Ahmad , 26/05/2024, 11:01:54 pm

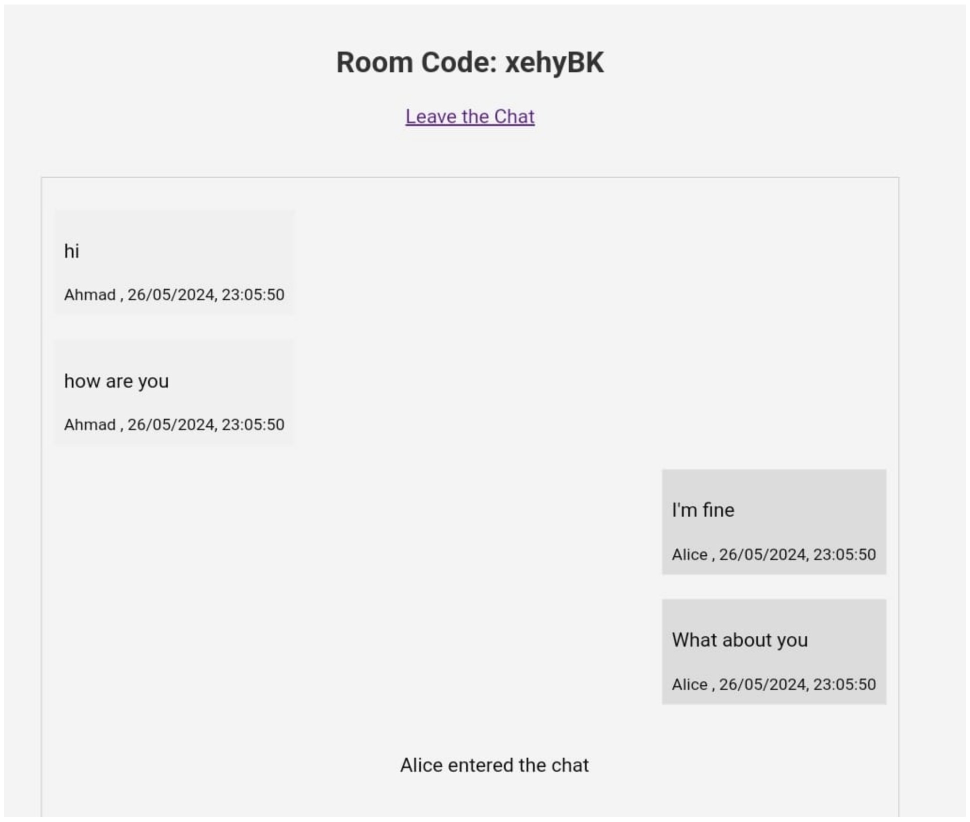
I'm fine

Alice , 26/05/2024, 11:02:01 pm

What about you

Alice , 26/05/2024, 11:02:05 pm

That's how they started their conversation. However, due to internet connection problem Alice left. She returned and everything was saved.



Room Code: xehyBK

Leave the Chat

hi

Ahmad , 26/05/2024, 23:05:50

how are you

Ahmad , 26/05/2024, 23:05:50

I'm fine

Alice , 26/05/2024, 23:05:50

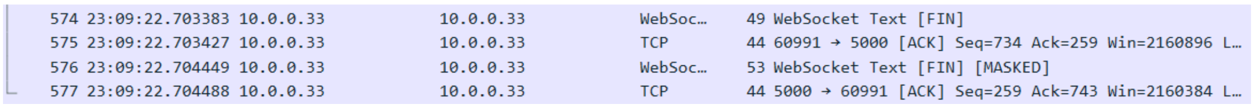
What about you

Alice , 26/05/2024, 23:05:50

Alice entered the chat

This is a short example of what can the application do.

At the same time, everything was being shown on wireshark, after using the filter ip.addr == 10.0.0.33



```
574 23:09:22.703383 10.0.0.33    10.0.0.33    WebSoc…   49 WebSocket Text [FIN]
575 23:09:22.703427 10.0.0.33    10.0.0.33    TCP       44 60991 → 5000 [ACK] Seq=734 Ack=259 Win=2160896 L…
576 23:09:22.704449 10.0.0.33    10.0.0.33    WebSoc…   53 WebSocket Text [FIN] [MASKED]
577 23:09:22.704488 10.0.0.33    10.0.0.33    TCP       44 5000 → 60991 [ACK] Seq=259 Ack=743 Win=2160384 L…
```

This is an example of one message being sent with an ack and ack being received that the message was received. As we see, the source and destination IP are the same because it is a loopback even if they are receiving via Wifi (same Wifi here). As mentioned above, we’ve used websocket for sending and receiving messages and the TCP is for ACK.