

**IMPLEMENTATION DETAILS**

**GROUP MEMBERS:**

**1.Fahad (054-72148)**

**2.Faizan (054-72198)**

**3.M.anas (054-72259)**

**APP.PY CODE**

# app.py

from flask import Flask, render\_template, request, jsonify

from flask\_socketio import SocketIO, emit, join\_room

from kms\_core import KMS

import json

import os

app = Flask(\_\_name\_\_, template\_folder="templates", static\_folder="static")

socketio = SocketIO(app)

kms = KMS()

MESSAGE\_FILE = "messages.json"

# Shared registration file with kms\_api.py

REGISTERED\_FILE = os.path.join(os.path.dirname(\_\_file\_\_), "registered\_users.json")

# Load or initialize registered users

if os.path.exists(REGISTERED\_FILE):

    with open(REGISTERED\_FILE, "r") as f:

        registered\_users = set(json.load(f))

else:

    registered\_users = set()

def save\_registered\_users():

    with open(REGISTERED\_FILE, "w") as f:

        json.dump(list(registered\_users), f)

def load\_messages():

    if os.path.exists(MESSAGE\_FILE):

        with open(MESSAGE\_FILE, "r") as f:

            return json.load(f)

    return []

def save\_message(sender, recipient, encrypted, algorithm):

    messages = load\_messages()

    messages.append({

        "sender": sender,

        "recipient": recipient,

        "encrypted": encrypted,

        "algorithm": algorithm

    })

    with open(MESSAGE\_FILE, "w") as f:

        json.dump(messages, f)

user\_sessions = dict()

@app.route("/")

def index():

    return render\_template("chat.html")

@socketio.on("register")

def handle\_register(data):

    user = data["user"]

    if user not in registered\_users:

        kms.generate\_user\_keys(user)

        registered\_users.add(user)

        save\_registered\_users()

    emit("registered", {"message": f"User '{user}' registered."})

@socketio.on("join")

def on\_join(data):

    username = data["username"]

    if username not in registered\_users:

        emit("error", {"message": f"User '{username}' is not registered."})

        return

    user\_sessions[request.sid] = username

    join\_room(username)

    emit("receive\_message", {

        "sender": "System",

        "recipient": username,

        "decrypted": f"{username} has joined."

    }, room=username)

@socketio.on("send\_message")

def handle\_message(data):

    sid = request.sid

    sender = user\_sessions.get(sid)

    recipient = data["recipient"]

    message = data["message"]

    algorithm = data.get("algorithm", "RSA")

    if sender not in registered\_users or recipient not in registered\_users:

        emit("error", {"message": "Sender or recipient is not registered."})

        return

    encrypted = kms.encrypt\_for\_user(recipient, message, algorithm=algorithm)

    try:

        decrypted = kms.decrypt\_for\_user(recipient, encrypted, algorithm=algorithm)

    except:

        decrypted = "[Only recipient can decrypt]"

    save\_message(sender, recipient, encrypted, algorithm)

    emit("receive\_message", {

        "sender": sender,

        "recipient": recipient,

        "encrypted": encrypted,

        "decrypted": decrypted,

        "algorithm": algorithm

    }, room=recipient)

@socketio.on("fetch\_messages")

def handle\_fetch\_messages(data):

    username = data["username"]

    if username not in registered\_users:

        emit("error", {"message": f"User '{username}' is not registered."})

        return

    user\_messages = []

    for msg in load\_messages():

        if msg["recipient"] == username:

            algorithm = msg.get("algorithm", "RSA")

            try:

                decrypted = kms.decrypt\_for\_user(username, msg["encrypted"], algorithm=algorithm)

            except:

                decrypted = "[Unable to decrypt]"

            user\_messages.append({

                "sender": msg["sender"],

                "decrypted": decrypted,

                "algorithm": algorithm

            })

    emit("inbox", user\_messages)

if \_\_name\_\_ == "\_\_main\_\_":

    socketio.run(app, debug=True)

**KMS\_CORE.PY CODE**

# kms\_core.py

import os, json

from cryptography.hazmat.primitives.asymmetric import rsa, padding

from cryptography.hazmat.primitives import hashes, serialization

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.fernet import Fernet

import base64

import secrets

class KMS:

    def \_\_init\_\_(self, key\_dir='keys'):

        os.makedirs(key\_dir, exist\_ok=True)

        self.key\_dir = key\_dir

    def generate\_user\_keys(self, user):

        priv\_path = f"{self.key\_dir}/{user}\_private.pem"

        pub\_path = f"{self.key\_dir}/{user}\_public.pem"

        if not os.path.exists(priv\_path):

            private\_key = rsa.generate\_private\_key(public\_exponent=65537, key\_size=2048)

            public\_key = private\_key.public\_key()

            with open(priv\_path, 'wb') as f:

                f.write(private\_key.private\_bytes(

                    encoding=serialization.Encoding.PEM,

                    format=serialization.PrivateFormat.PKCS8,

                    encryption\_algorithm=serialization.NoEncryption()

                ))

            with open(pub\_path, 'wb') as f:

                f.write(public\_key.public\_bytes(

                    encoding=serialization.Encoding.PEM,

                    format=serialization.PublicFormat.SubjectPublicKeyInfo

                ))

        self.get\_or\_generate\_aes\_key(user)

        self.get\_or\_generate\_fernet\_key(user)

    def load\_private\_key(self, user):

        with open(f"{self.key\_dir}/{user}\_private.pem", 'rb') as f:

            return serialization.load\_pem\_private\_key(f.read(), password=None)

    def load\_public\_key(self, user):

        with open(f"{self.key\_dir}/{user}\_public.pem", 'rb') as f:

            return serialization.load\_pem\_public\_key(f.read())

    def get\_or\_generate\_aes\_key(self, user):

        path = f"{self.key\_dir}/{user}\_aes.key"

        if os.path.exists(path):

            with open(path, "rb") as f:

                return f.read()

        key = secrets.token\_bytes(32)

        with open(path, "wb") as f:

            f.write(key)

        return key

    def get\_or\_generate\_fernet\_key(self, user):

        path = f"{self.key\_dir}/{user}\_fernet.key"

        if os.path.exists(path):

            with open(path, "rb") as f:

                return f.read()

        key = Fernet.generate\_key()

        with open(path, "wb") as f:

            f.write(key)

        return key

    def encrypt\_for\_user(self, recipient, message, algorithm="RSA"):

        if algorithm == "RSA":

            pub = self.load\_public\_key(recipient)

            return pub.encrypt(

                message.encode(),

                padding.OAEP(

                    mgf=padding.MGF1(algorithm=hashes.SHA256()),

                    algorithm=hashes.SHA256(),

                    label=None

                )

            ).hex()

        elif algorithm == "AES":

            key = self.get\_or\_generate\_aes\_key(recipient)

            iv = secrets.token\_bytes(16)

            cipher = Cipher(algorithms.AES(key), modes.CFB(iv))

            encryptor = cipher.encryptor()

            encrypted = encryptor.update(message.encode()) + encryptor.finalize()

            return (iv + encrypted).hex()

        elif algorithm == "Fernet":

            key = self.get\_or\_generate\_fernet\_key(recipient)

            f = Fernet(key)

            return f.encrypt(message.encode()).decode()

        else:

            raise ValueError(f"Unsupported algorithm: {algorithm}")

    def decrypt\_for\_user(self, user, ciphertext, algorithm="RSA"):

        if algorithm == "RSA":

            priv = self.load\_private\_key(user)

            return priv.decrypt(

                bytes.fromhex(ciphertext),

                padding.OAEP(

                    mgf=padding.MGF1(algorithm=hashes.SHA256()),

                    algorithm=hashes.SHA256(),

                    label=None

                )

            ).decode()

        elif algorithm == "AES":

            key = self.get\_or\_generate\_aes\_key(user)

            ciphertext\_bytes = bytes.fromhex(ciphertext)

            iv = ciphertext\_bytes[:16]

            actual\_ciphertext = ciphertext\_bytes[16:]

            cipher = Cipher(algorithms.AES(key), modes.CFB(iv))

            decryptor = cipher.decryptor()

            return (decryptor.update(actual\_ciphertext) + decryptor.finalize()).decode()

        elif algorithm == "Fernet":

            key = self.get\_or\_generate\_fernet\_key(user)

            f = Fernet(key)

            return f.decrypt(ciphertext.encode()).decode()

        else:

            raise ValueError(f"Unsupported algorithm: {algorithm}")

**KMS\_API.PY CODE**

# kms\_api.py

from flask import Flask, request, jsonify

from kms\_core import KMS

import os

import json

app = Flask(\_\_name\_\_)

kms = KMS()

MESSAGE\_FILE = "messages.json"

REGISTERED\_FILE = "registered\_users.json"

# Load or initialize registered users

if os.path.exists(REGISTERED\_FILE):

    with open(REGISTERED\_FILE, "r") as f:

        registered\_users = set(json.load(f))

else:

    registered\_users = set()

def save\_registered\_users():

    with open(REGISTERED\_FILE, "w") as f:

        json.dump(list(registered\_users), f)

@app.route("/")

def home():

    return jsonify({"message": "KMS API is running!"})

def load\_messages():

    if os.path.exists(MESSAGE\_FILE):

        with open(MESSAGE\_FILE, "r") as f:

            return json.load(f)

    return []

def save\_message(sender, recipient, encrypted, algorithm):

    messages = load\_messages()

    messages.append({

        "sender": sender,

        "recipient": recipient,

        "encrypted": encrypted,

        "algorithm": algorithm

    })

    with open(MESSAGE\_FILE, "w") as f:

        json.dump(messages, f)

@app.route("/register", methods=["POST"])

def register():

    data = request.get\_json()

    user = data.get("user")

    if not user:

        return jsonify({"error": "Missing 'user' field"}), 400

    kms.generate\_user\_keys(user)

    registered\_users.add(user)

    save\_registered\_users()

    return jsonify({"message": f"User '{user}' registered."})

@app.route("/encrypt", methods=["POST"])

def encrypt():

    data = request.get\_json()

    sender = data.get("sender")

    recipient = data.get("recipient")

    message = data.get("message")

    algorithm = data.get("algorithm", "RSA")

    if not all([sender, recipient, message]):

        return jsonify({"error": "Missing fields in request"}), 400

    if sender not in registered\_users or recipient not in registered\_users:

        return jsonify({"error": "Sender or recipient is not registered."}), 403

    encrypted = kms.encrypt\_for\_user(recipient, message, algorithm=algorithm)

    save\_message(sender, recipient, encrypted, algorithm)

    return jsonify({"encrypted": encrypted})

@app.route("/decrypt", methods=["POST"])

def decrypt():

    data = request.get\_json()

    user = data.get("user")

    encrypted = data.get("encrypted")

    algorithm = data.get("algorithm", "RSA")

    if not all([user, encrypted]):

        return jsonify({"error": "Missing fields in request"}), 400

    if user not in registered\_users:

        return jsonify({"error": "User is not registered."}), 403

    try:

        decrypted = kms.decrypt\_for\_user(user, encrypted, algorithm=algorithm)

        return jsonify({"decrypted": decrypted})

    except Exception as e:

        return jsonify({"error": str(e)}), 400

@app.route("/inbox/<user>", methods=["GET"])

def inbox(user):

    if user not in registered\_users:

        return jsonify({"error": "User is not registered."}), 403

    messages = load\_messages()

    user\_messages = []

    for msg in messages:

        if msg["recipient"] == user:

            algorithm = msg.get("algorithm", "RSA")

            try:

                decrypted = kms.decrypt\_for\_user(user, msg["encrypted"], algorithm=algorithm)

            except:

                decrypted = "[Unable to decrypt]"

            user\_messages.append({

                "sender": msg["sender"],

                "decrypted": decrypted,

                "algorithm": algorithm

            })

    return jsonify(user\_messages)

if \_\_name\_\_ == "\_\_main\_\_":

    app.run(debug=True)

**CHAT.HTML CODE**

<!DOCTYPE html>

<html>

<head>

  <title>Encrypted Chat</title>

  <script src="https://cdn.socket.io/4.5.4/socket.io.min.js"></script>

  <style>

    body { font-family: Arial; background: #f2f2f2; padding: 20px; }

    input, select, button { margin: 5px 0; padding: 10px; width: 100%; max-width: 400px; }

    #messages { margin-top: 20px; padding: 10px; border: 1px solid #ccc; background: #fff; height: 300px; overflow-y: scroll; }

    h2, h3 { color: #333; }

    #status { font-weight: bold; margin-bottom: 10px; }

  </style>

</head>

<body>

  <h2>Encrypted Chat App</h2>

  <div id="status">You are not logged in</div>

  <label for="username">Username:</label>

  <input type="text" id="username" placeholder="Enter your name">

  <button onclick="register()">Register</button>

  <button onclick="join()">Join</button>

  <label for="recipient">Recipient:</label>

  <input type="text" id="recipient" placeholder="Enter recipient name">

  <label for="message">Message:</label>

  <input type="text" id="message" placeholder="Enter your message">

  <label for="algorithm">Encryption Algorithm:</label>

  <select id="algorithm">

    <option value="RSA">RSA</option>

    <option value="AES">AES (Symmetric)</option>

    <option value="Fernet">Fernet (Symmetric)</option>

  </select>

  <button onclick="sendMessage()">Send Message</button>

  <button onclick="fetchInbox()">Fetch Inbox</button>

  <h3>Inbox</h3>

  <div id="messages"></div>

  <script>

    const socket = io();

    let user = "";

    function updateStatusDisplay() {

      const status = document.getElementById("status");

      if (user) {

        status.innerText = `You are logged in as: ${user}`;

      } else {

        status.innerText = "You are not logged in";

      }

    }

    function register() {

      user = document.getElementById("username").value;

      if (!user) return;

      socket.emit("register", { user });

      join(); // Automatically join after register

    }

    function join() {

      user = document.getElementById("username").value;

      if (!user) return;

      socket.emit("join", { username: user });

      updateStatusDisplay();

    }

    function sendMessage() {

      const recipient = document.getElementById("recipient").value;

      const message = document.getElementById("message").value;

      const algorithm = document.getElementById("algorithm").value;

      socket.emit("send\_message", {

        sender: user,

        recipient,

        message,

        algorithm

      });

    }

    function fetchInbox() {

      console.log("Fetching inbox for:", user);

      socket.emit("fetch\_messages", { username: user });

    }

    socket.on("receive\_message", (data) => {

      const box = document.getElementById("messages");

      if (data.sender === "System") {

        box.innerHTML += `<p><em>${data.decrypted}</em></p>`;

      } else {

        box.innerHTML += `<p><strong>${data.sender}</strong> (Encrypted with ${data.algorithm}):<br><code>${data.decrypted}</code></p>`;

      }

    });

    socket.on("inbox", (messages) => {

      const box = document.getElementById("messages");

      box.innerHTML = "";

      for (const msg of messages) {

        if (msg.sender === "System") {

          continue; // Skip system messages from inbox display

        }

        box.innerHTML += `<p><strong>${msg.sender}</strong> (Encrypted with ${msg.algorithm}):<br><code>${msg.decrypted}</code></p>`;

      }

    });

    socket.on("error", (data) => {

      alert("❌ " + data.message);

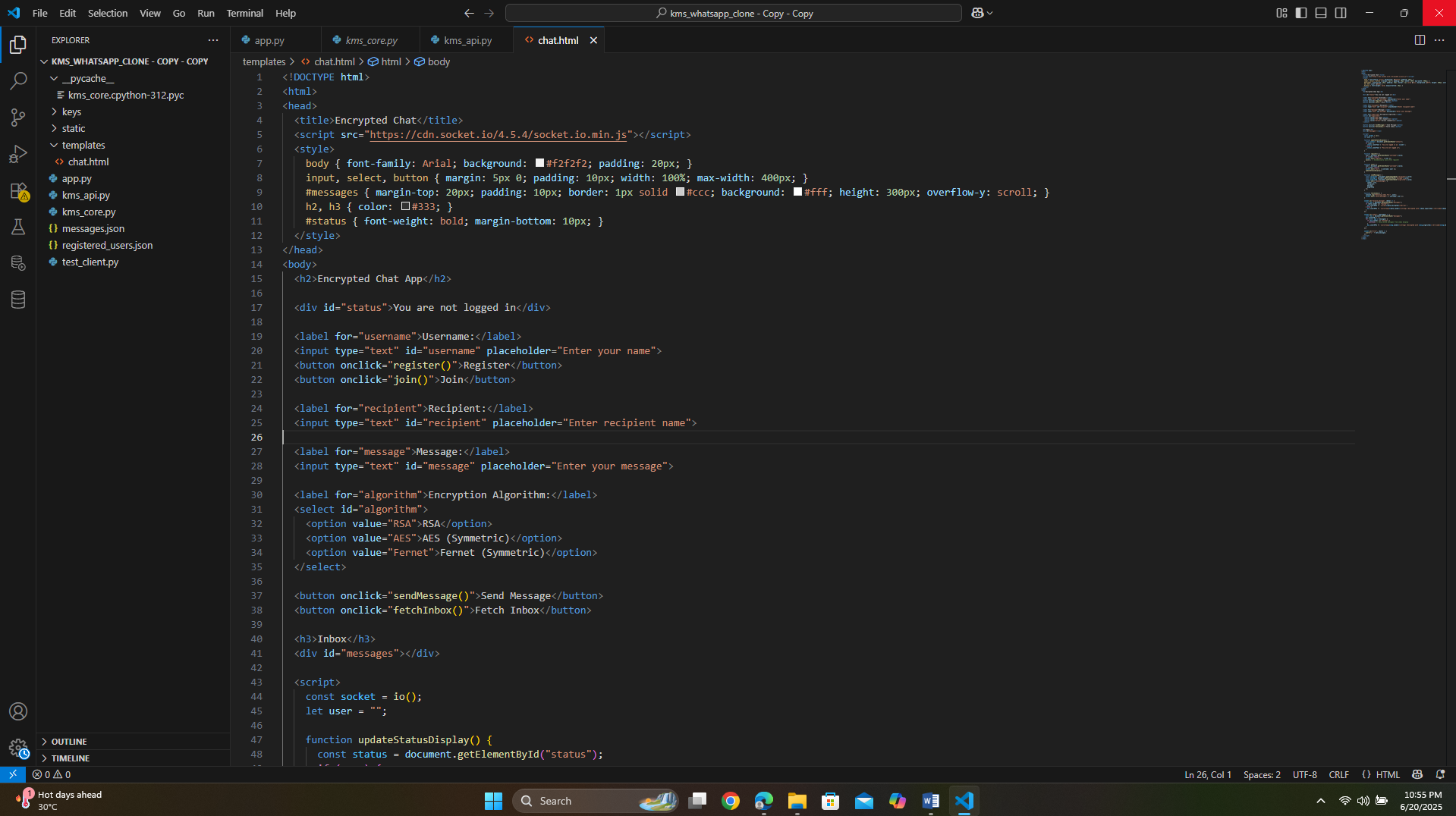
    });

  </script>

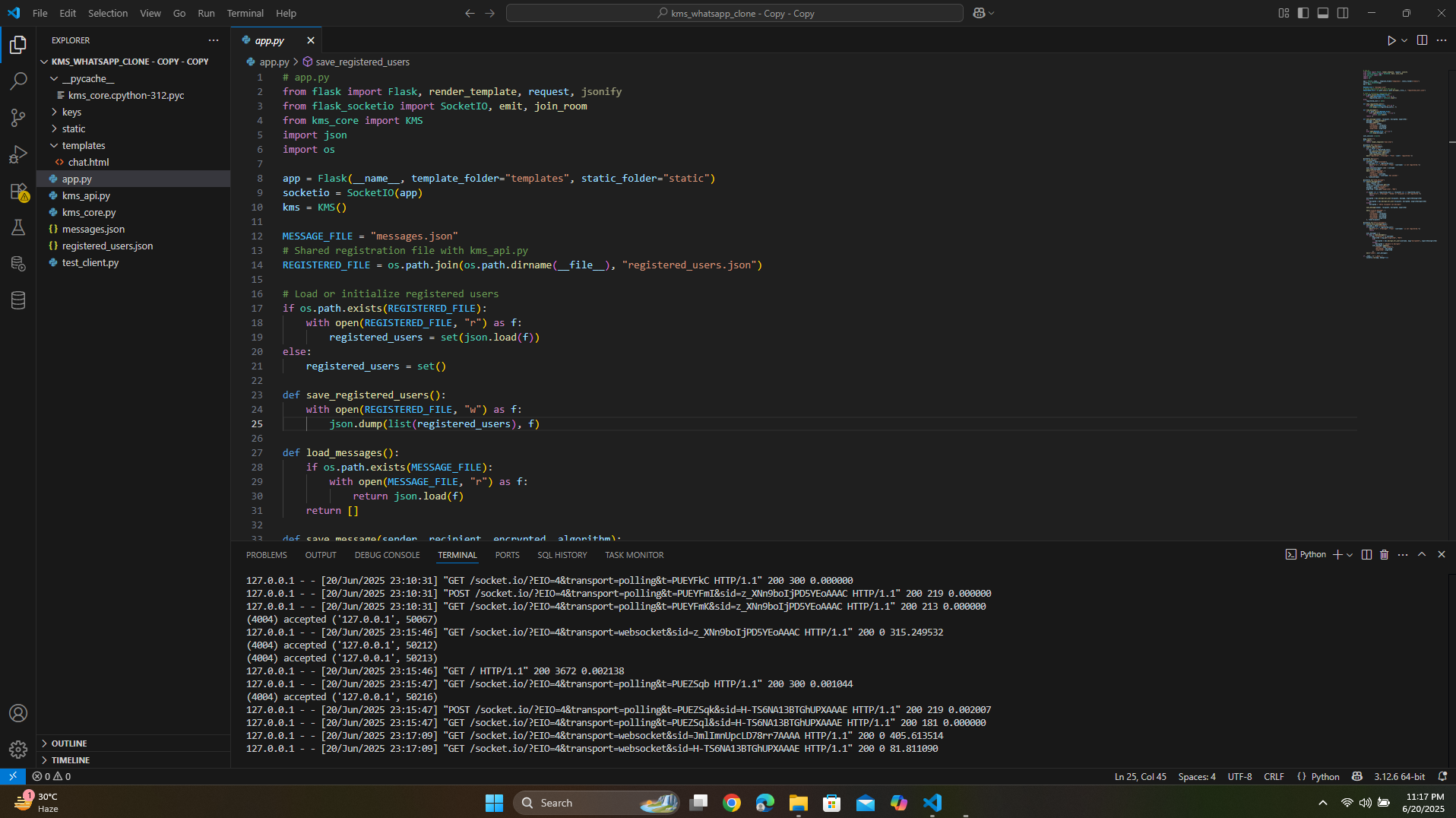
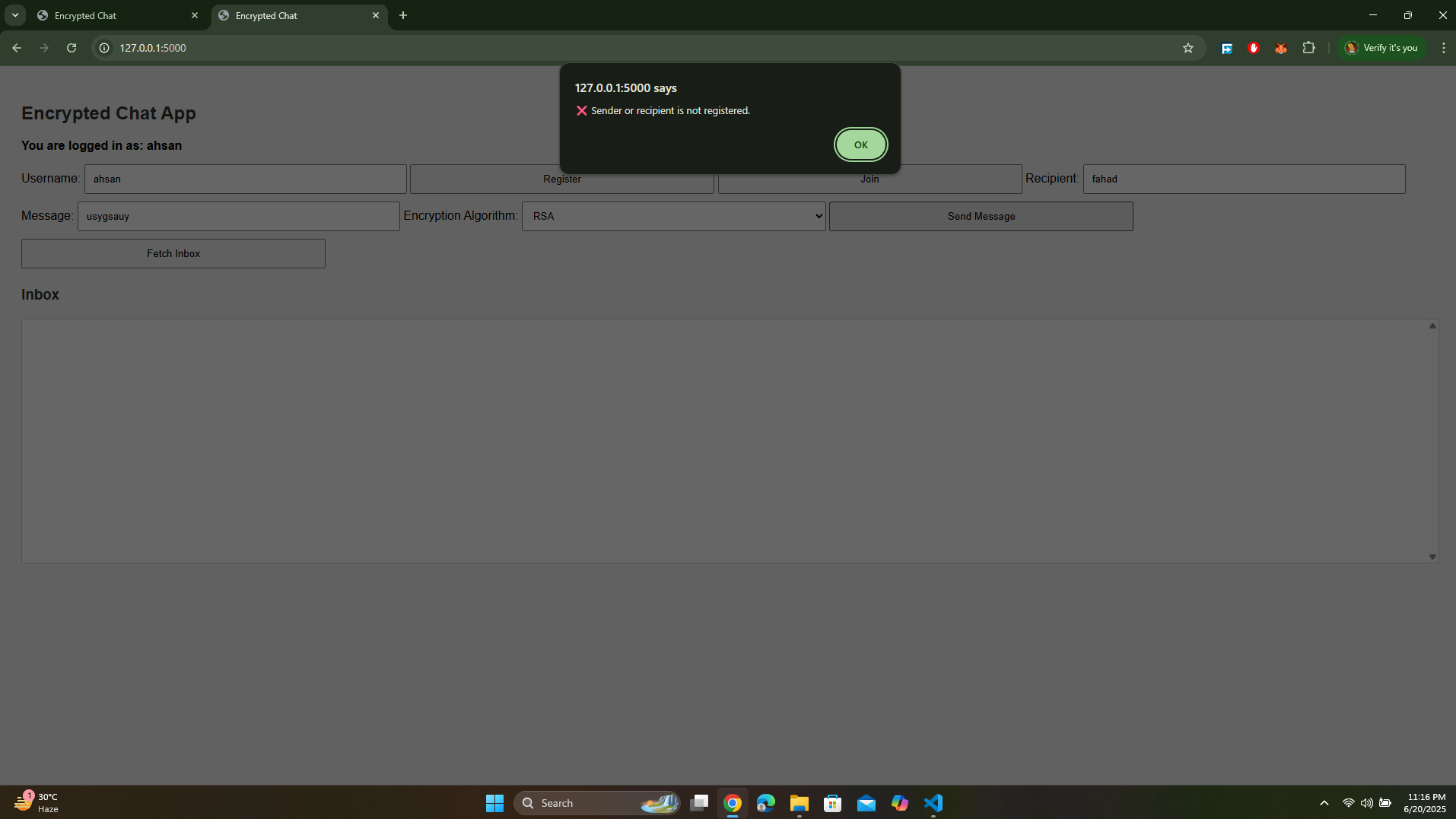
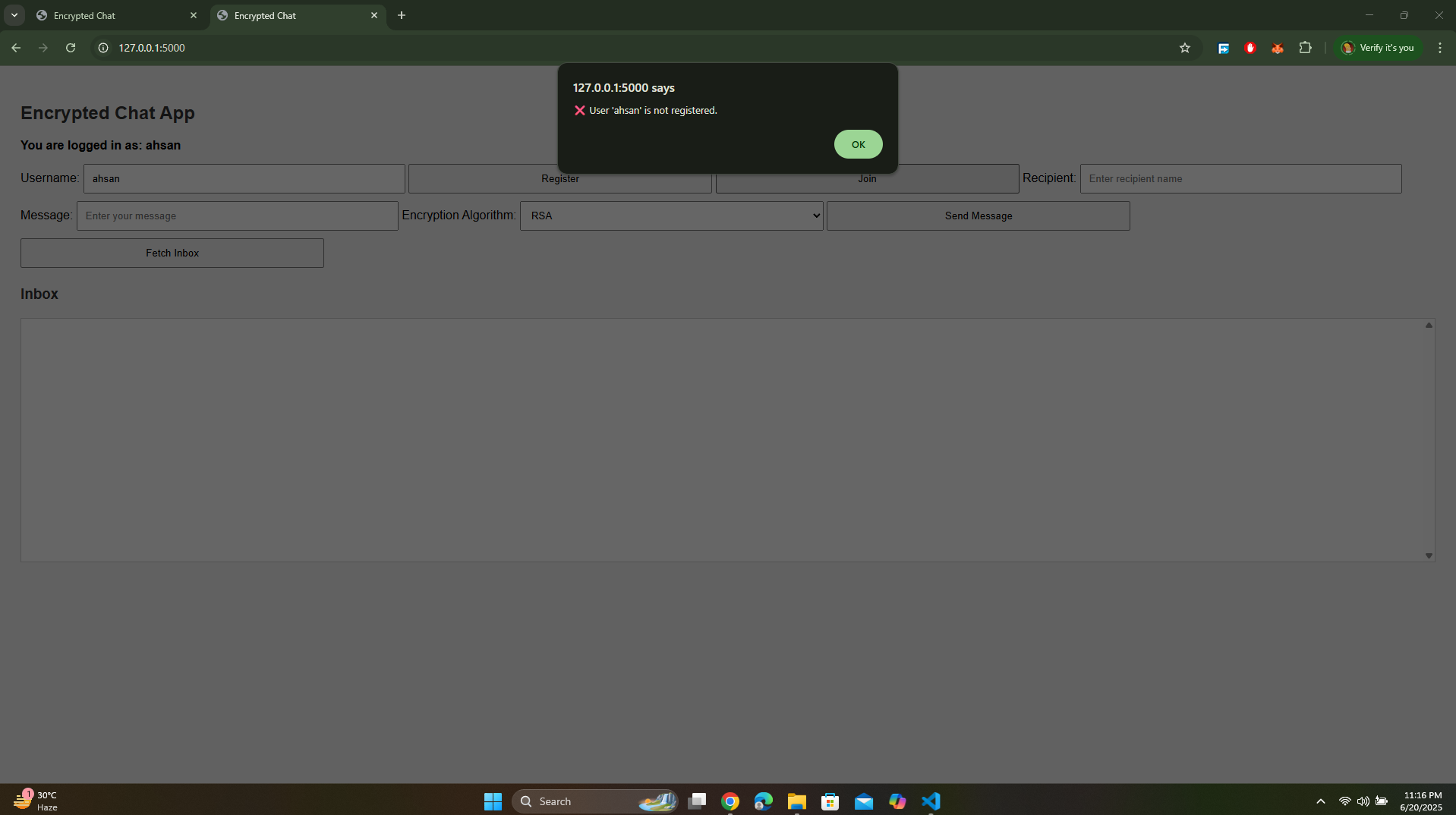
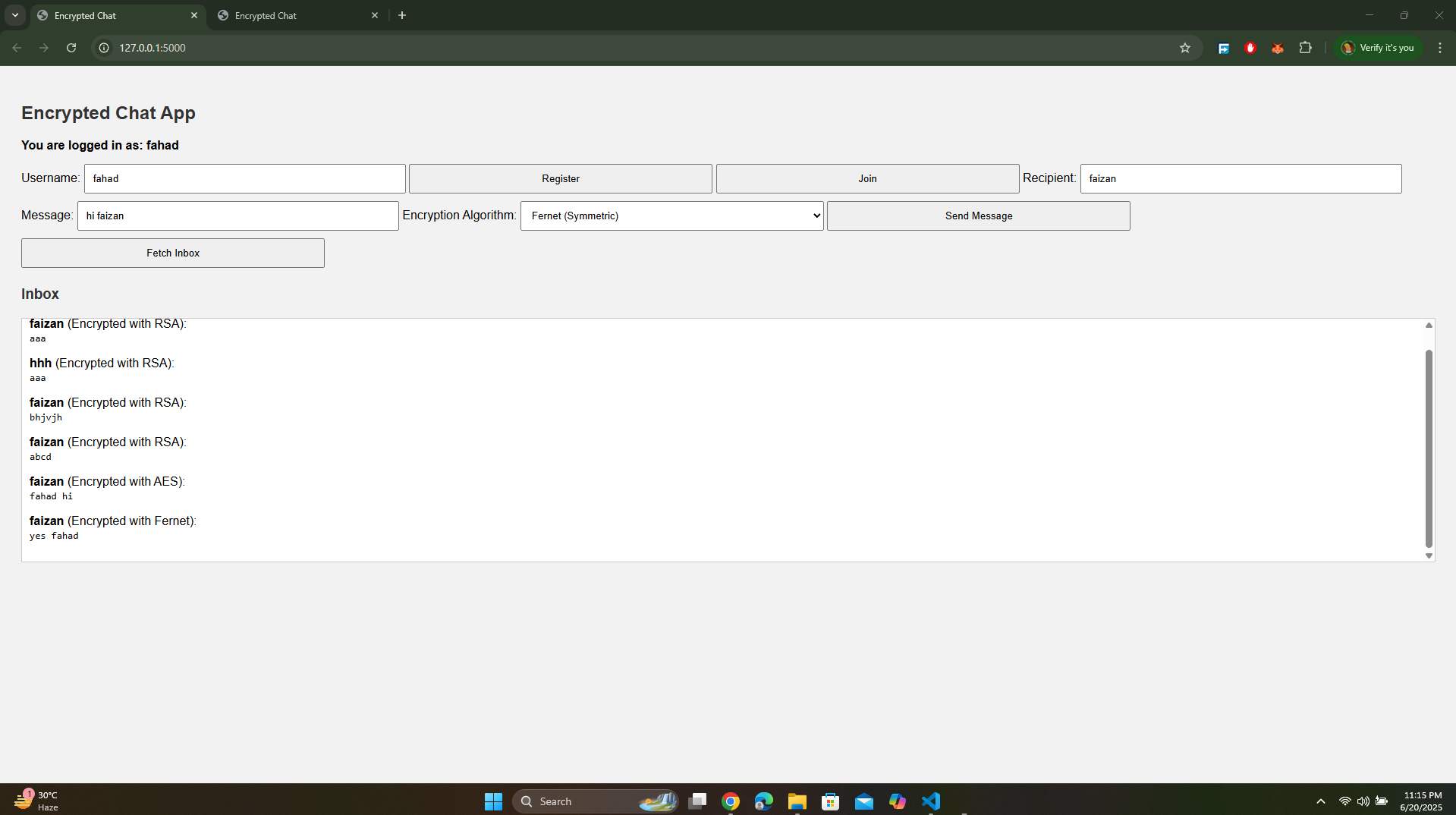
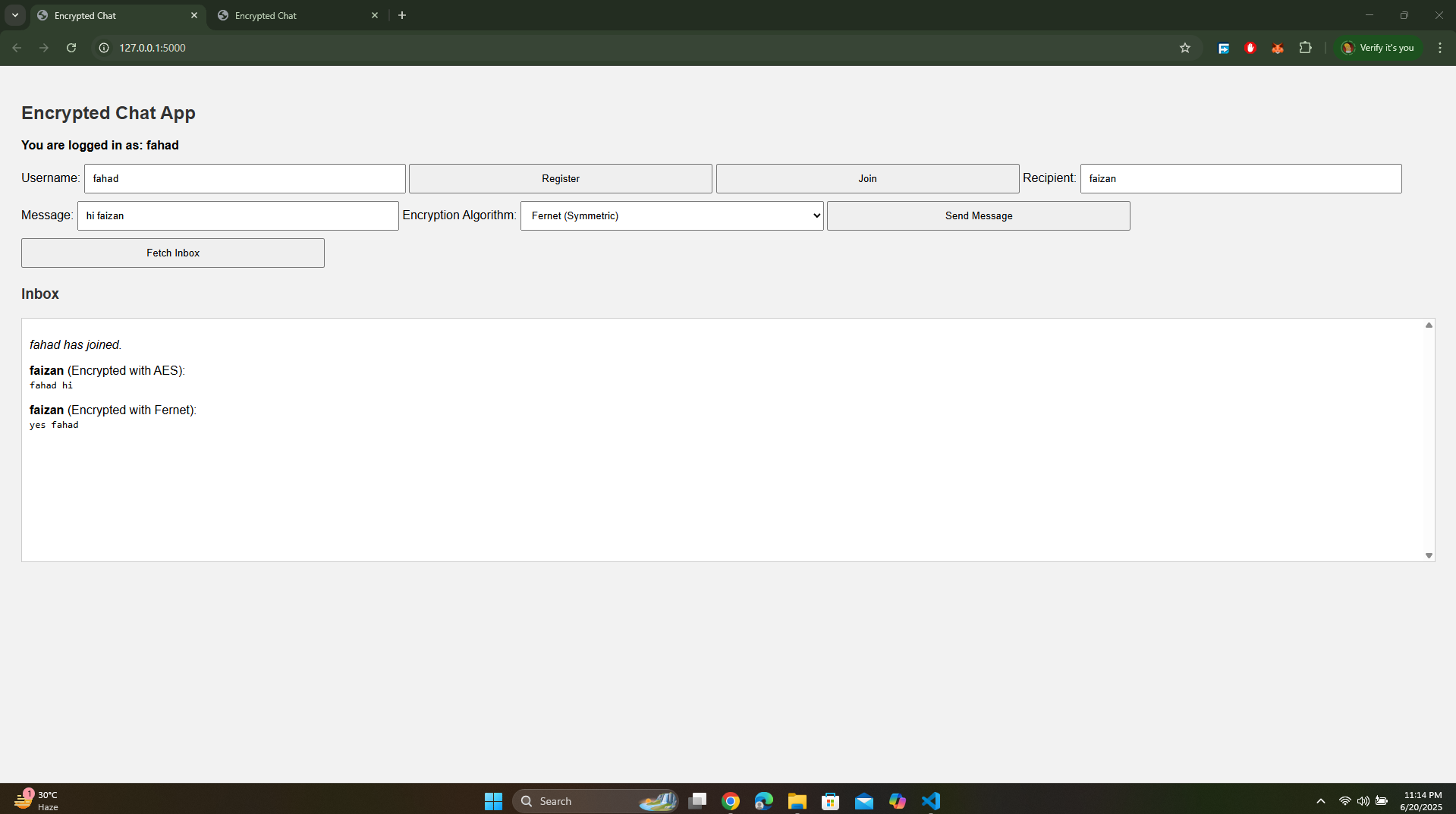
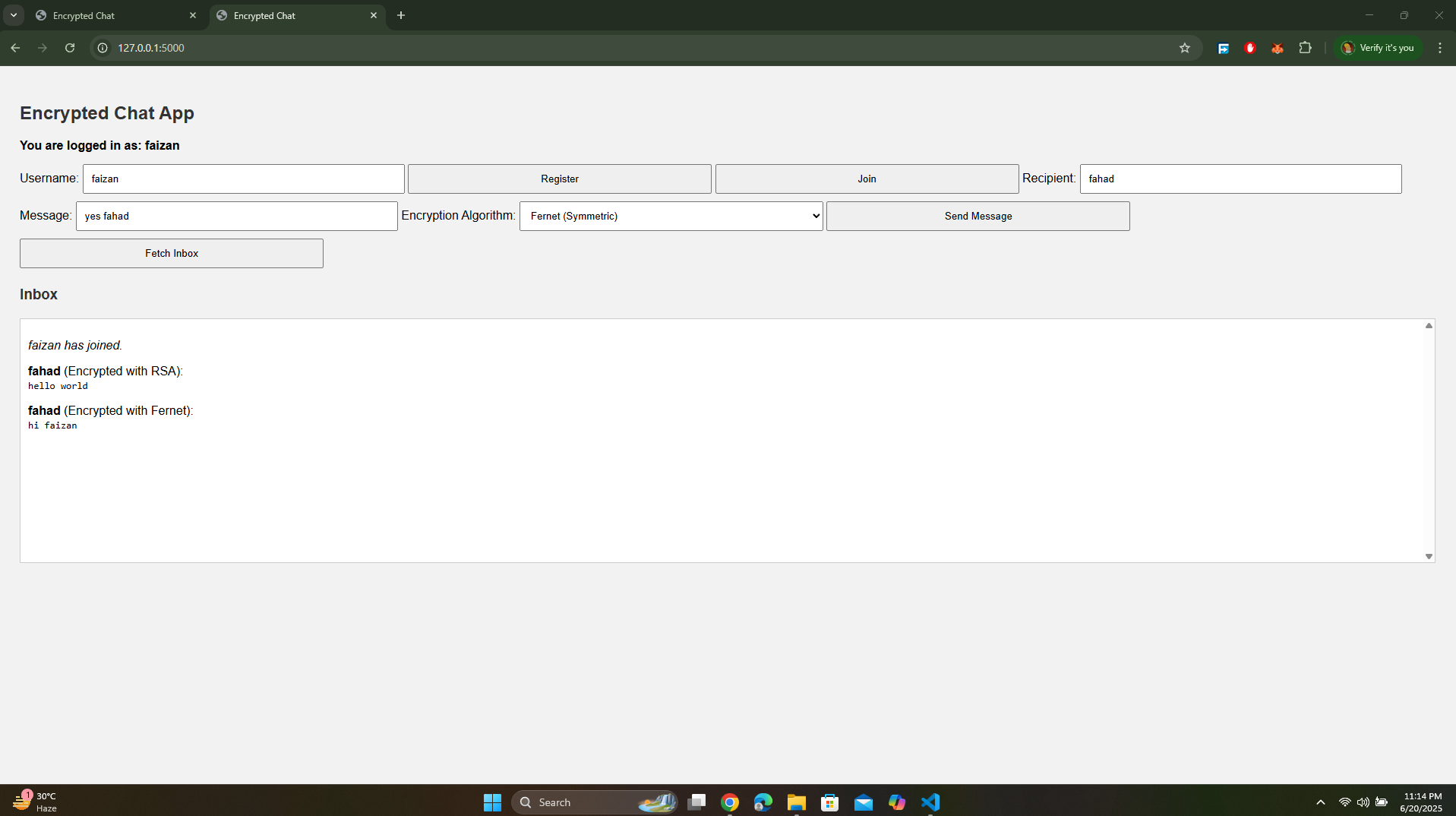
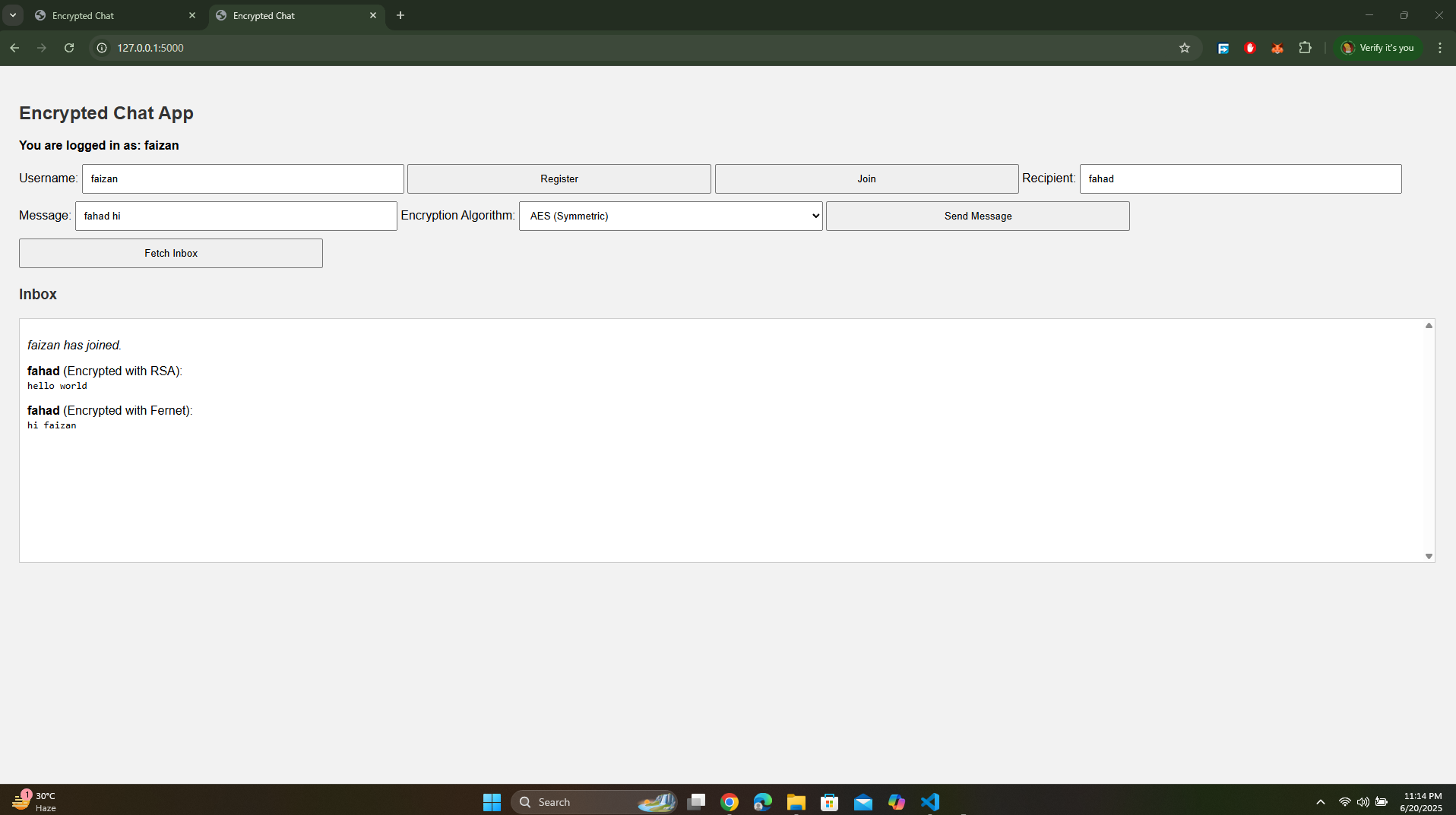
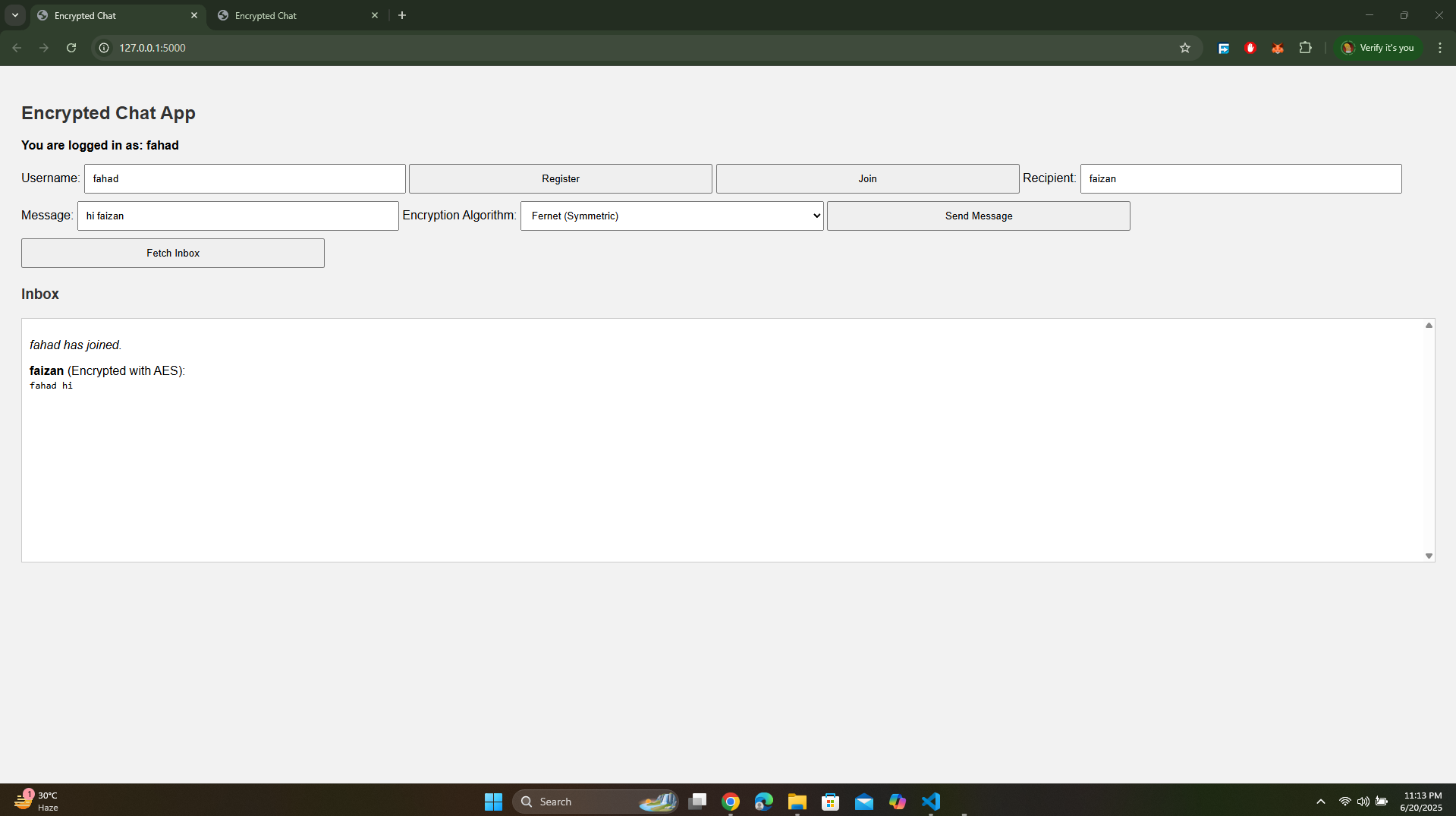
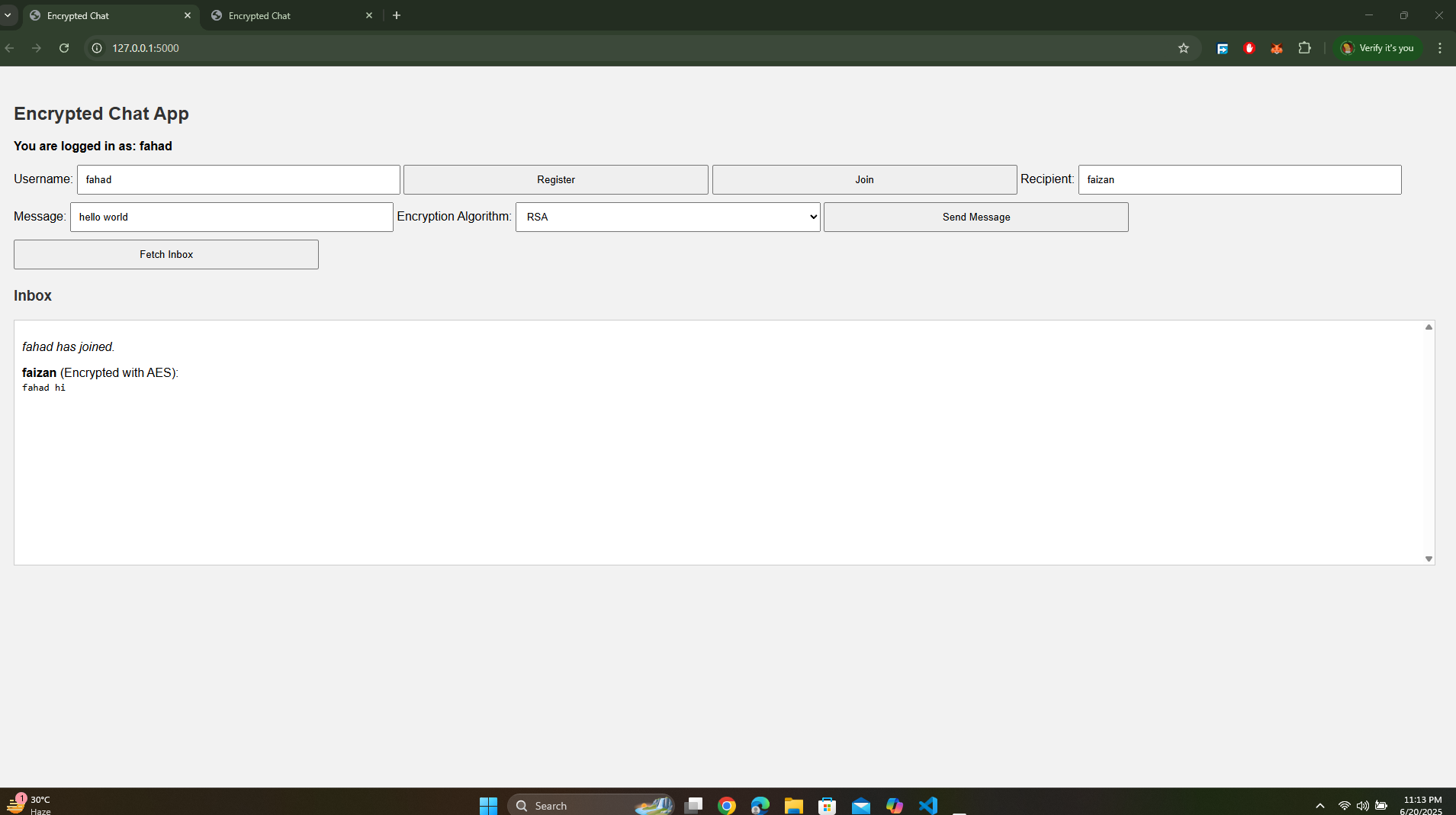
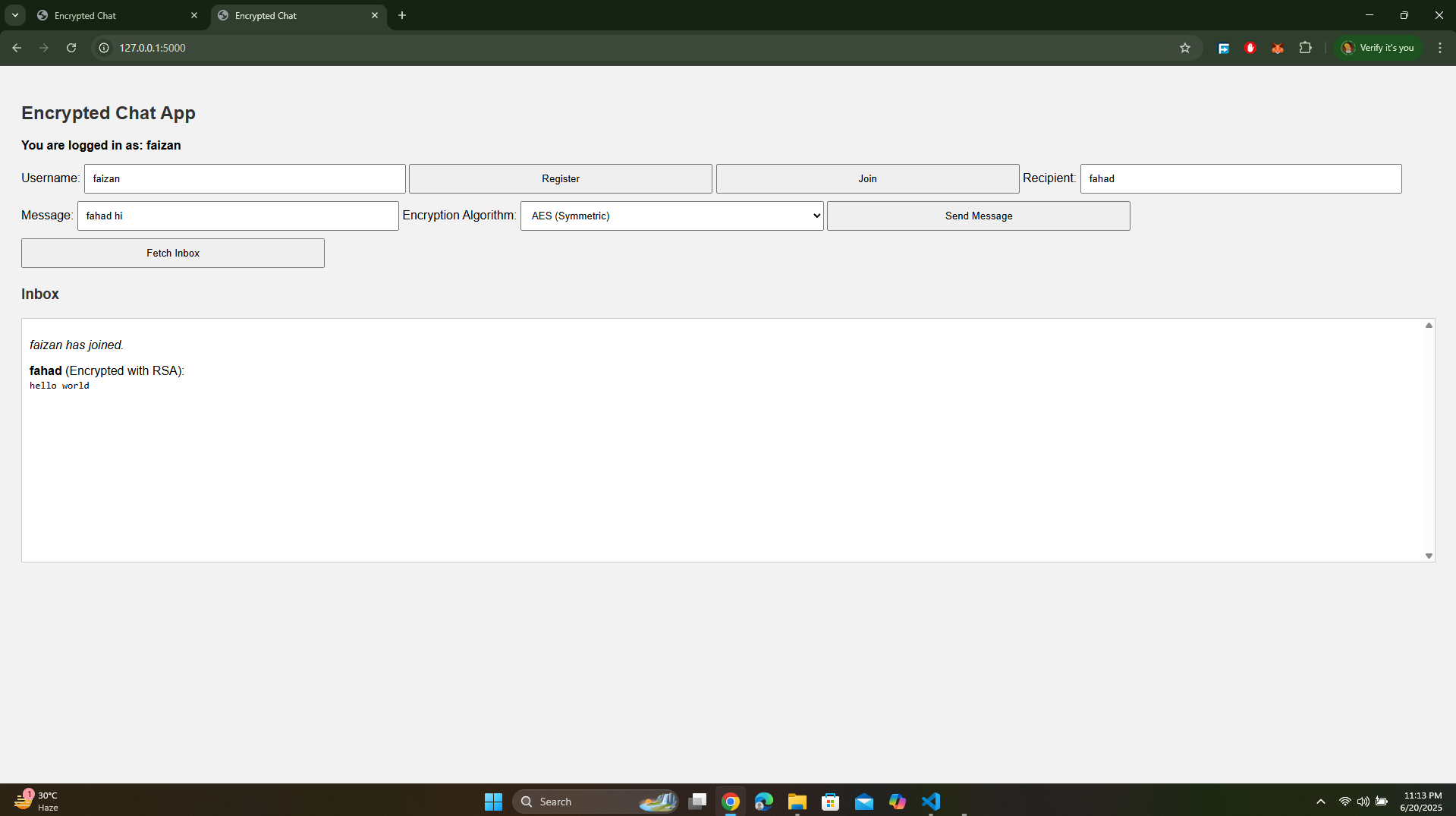
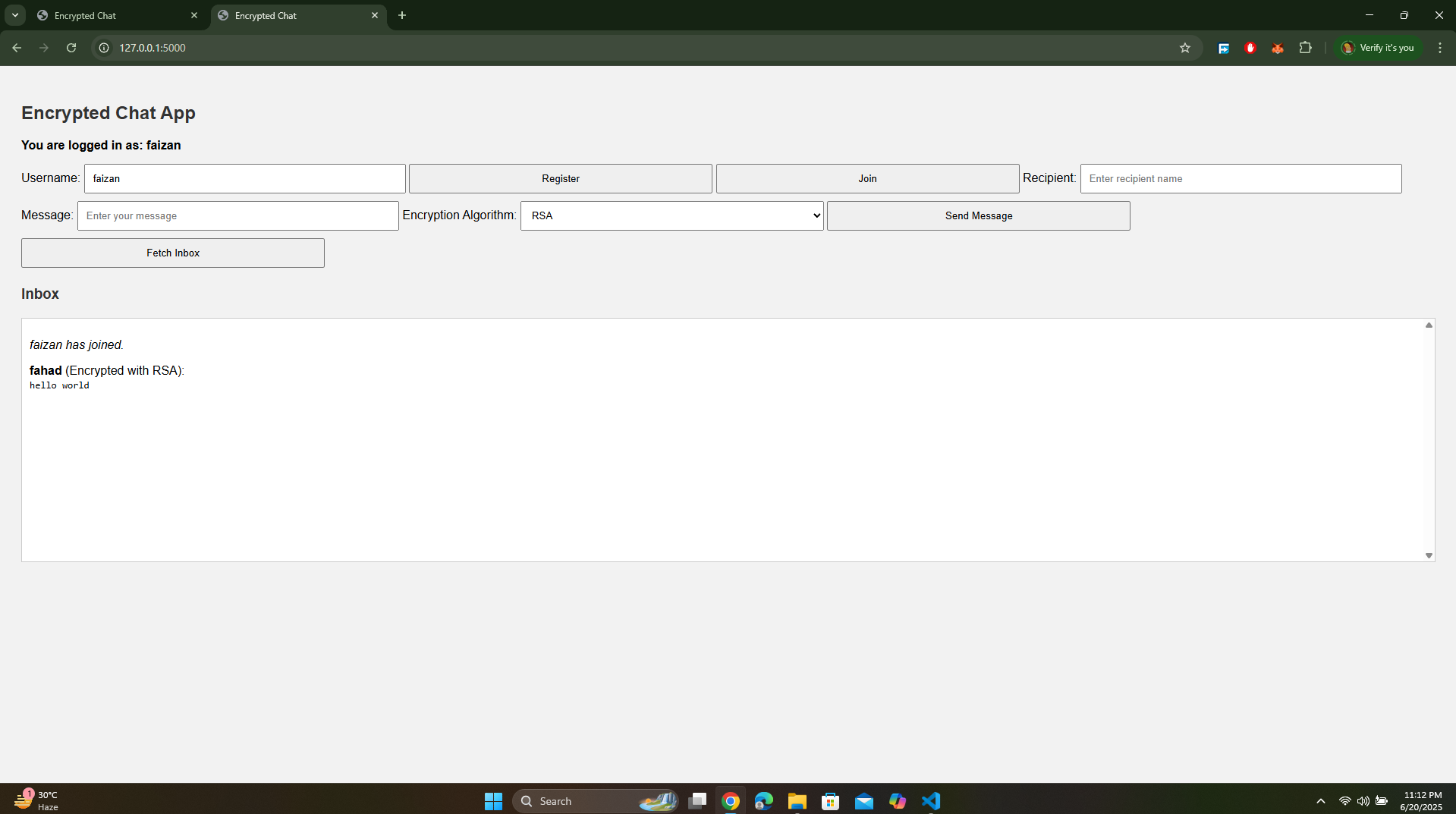
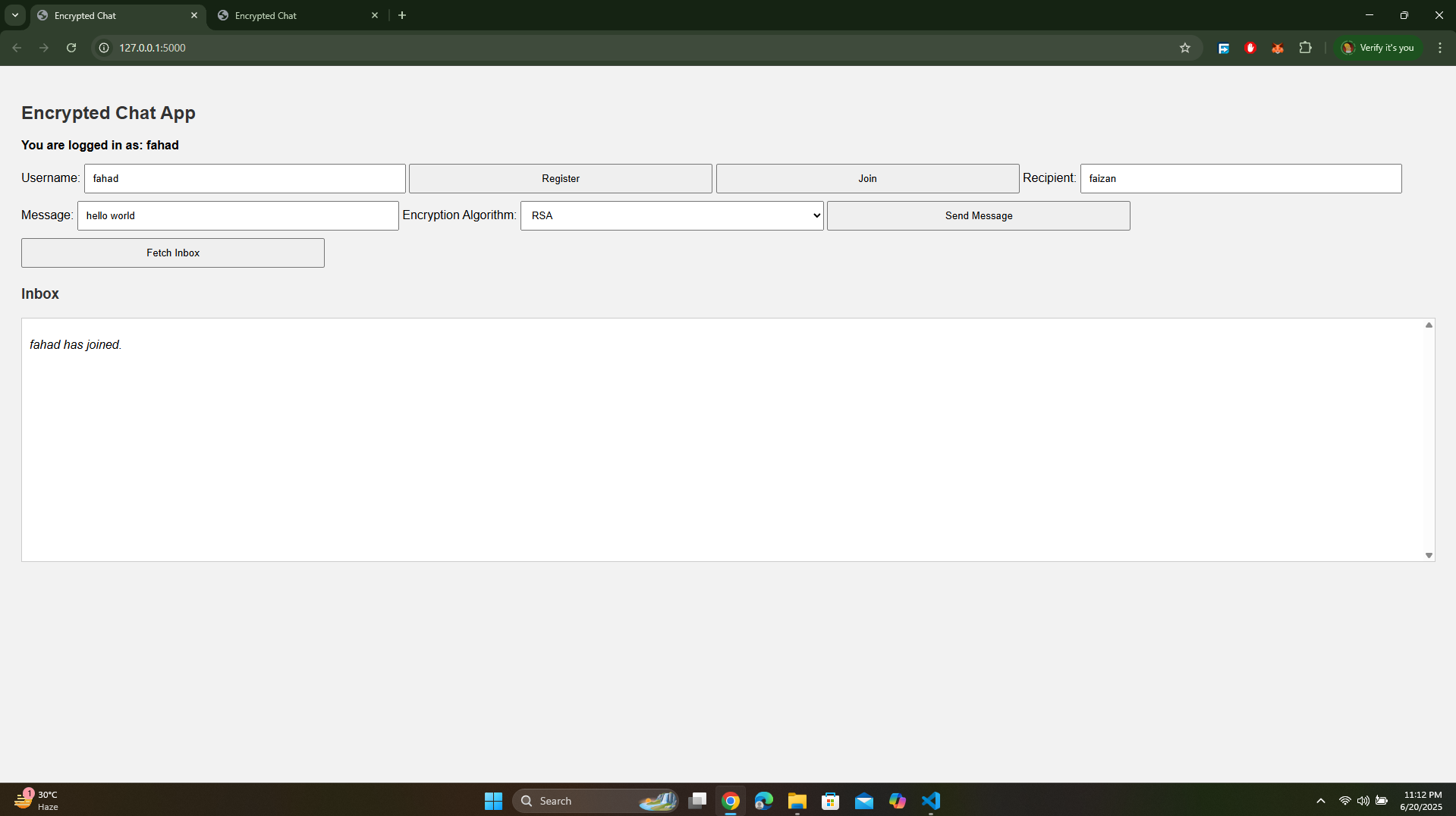
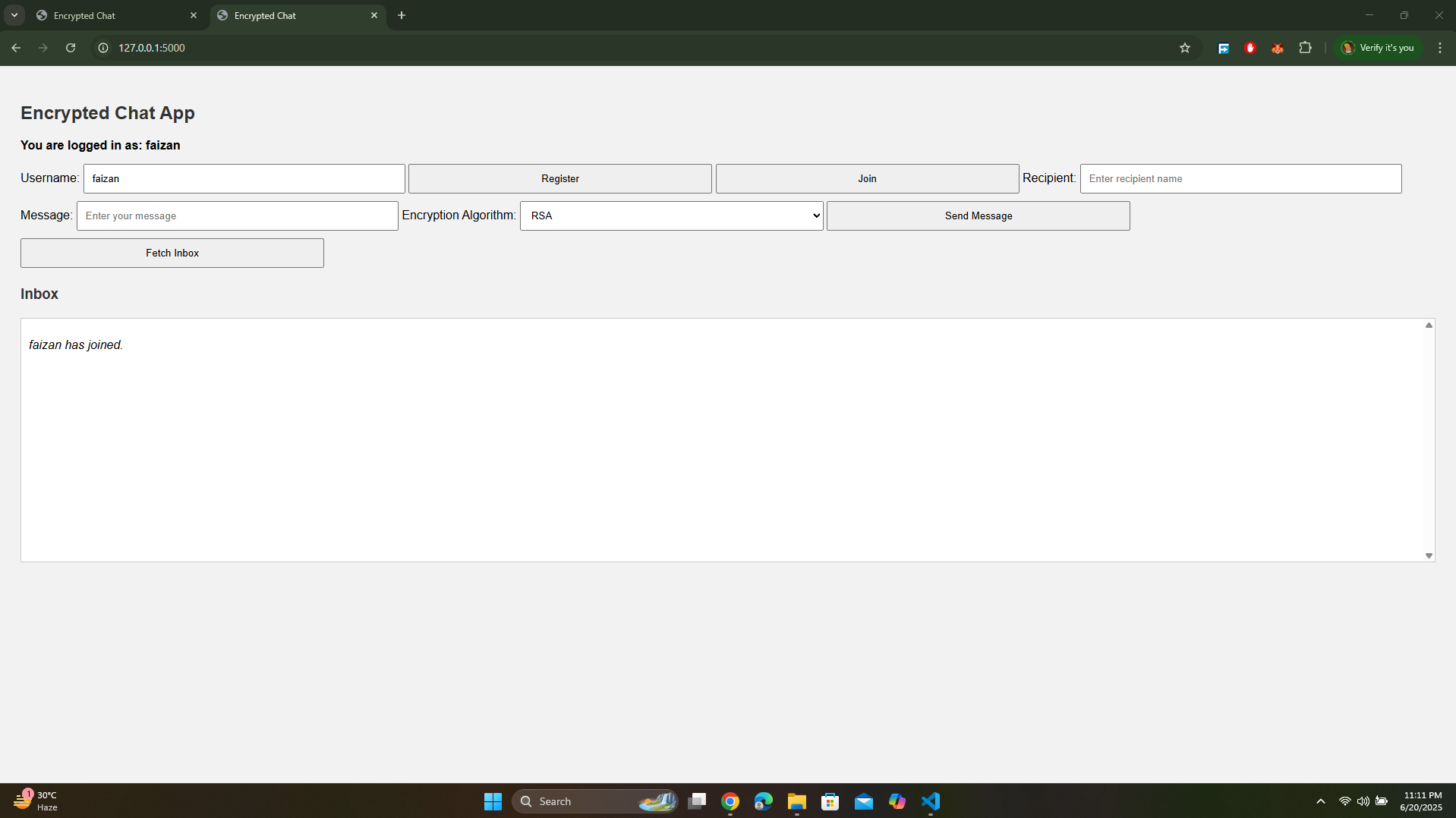
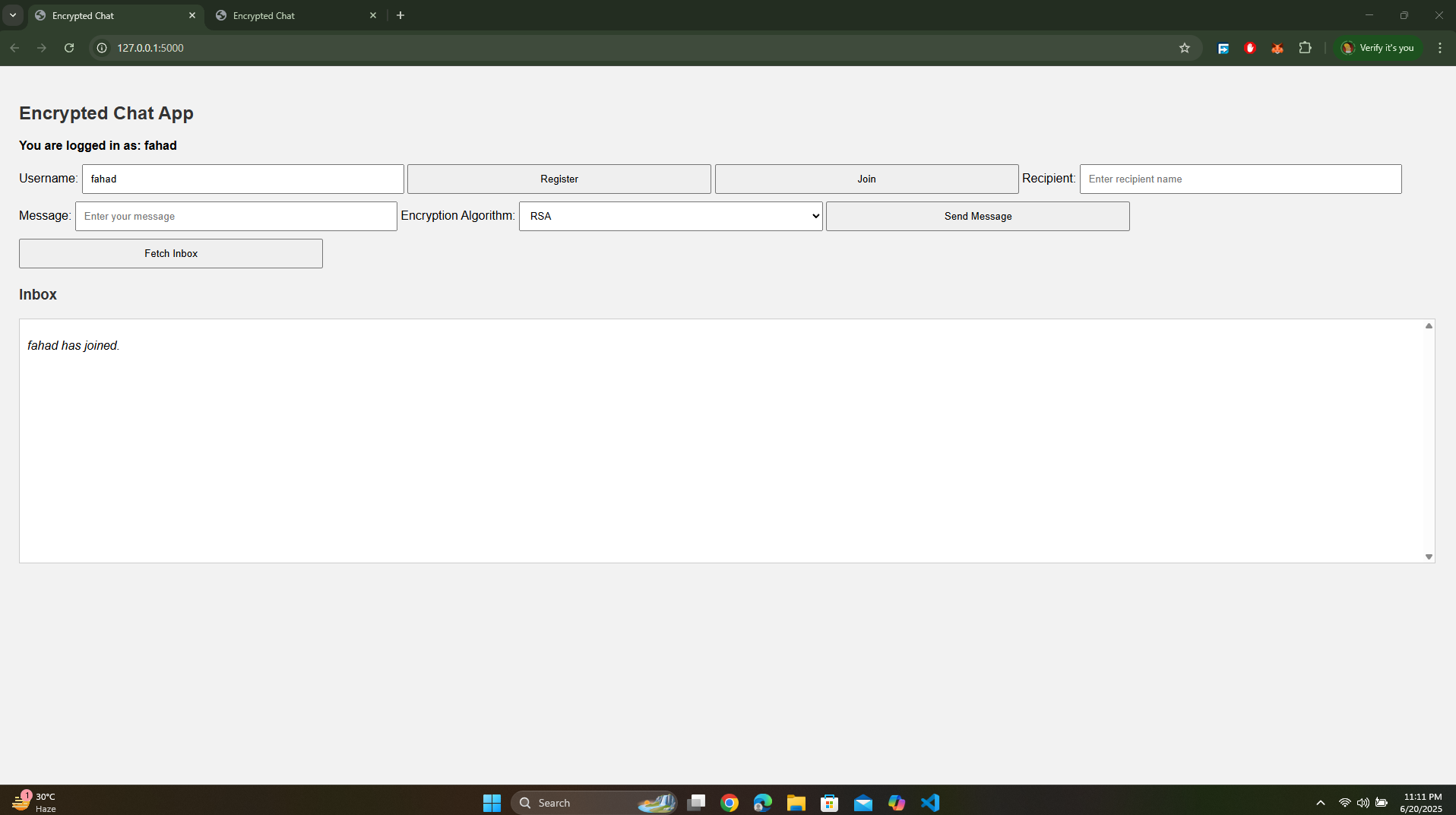
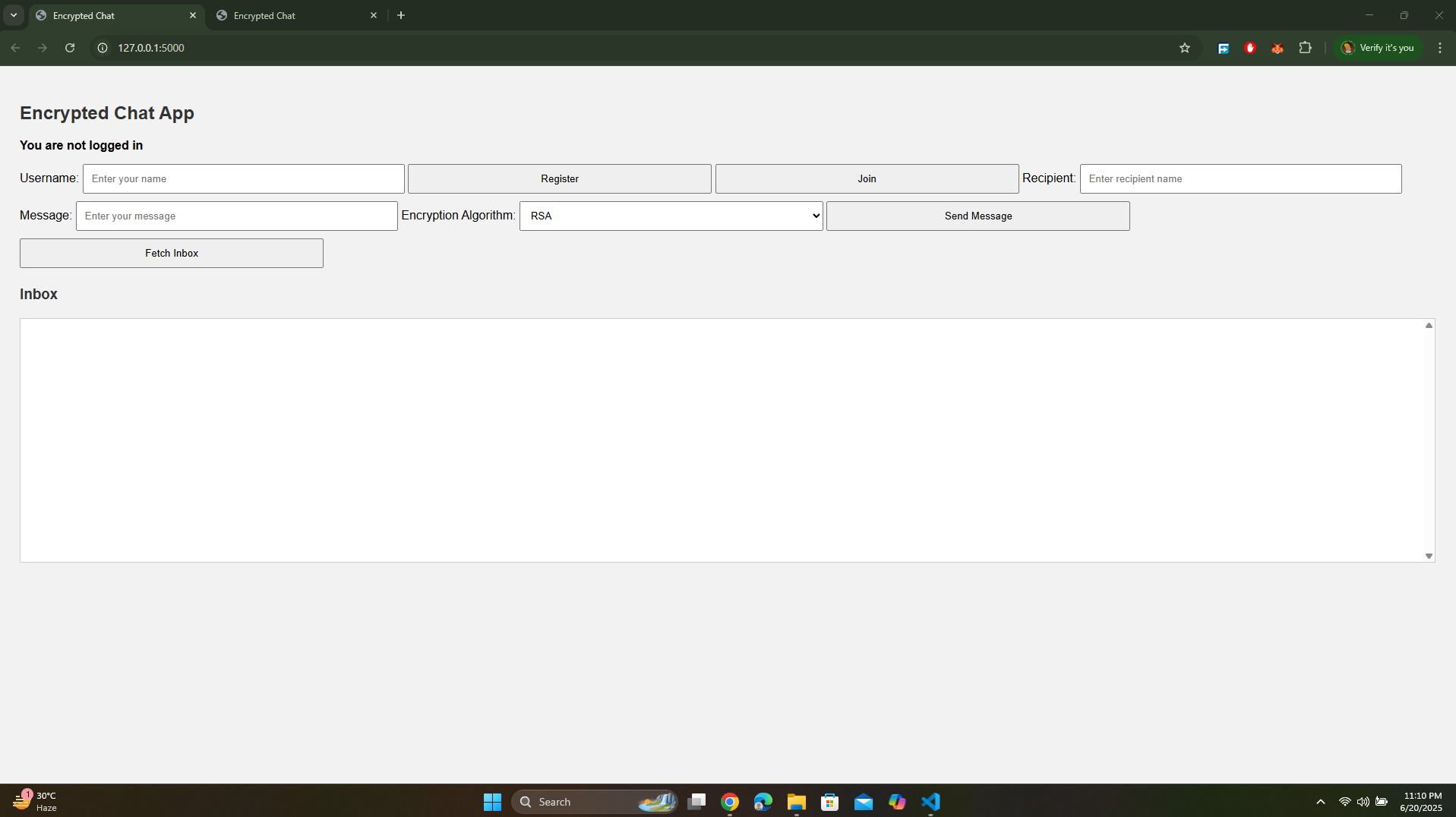
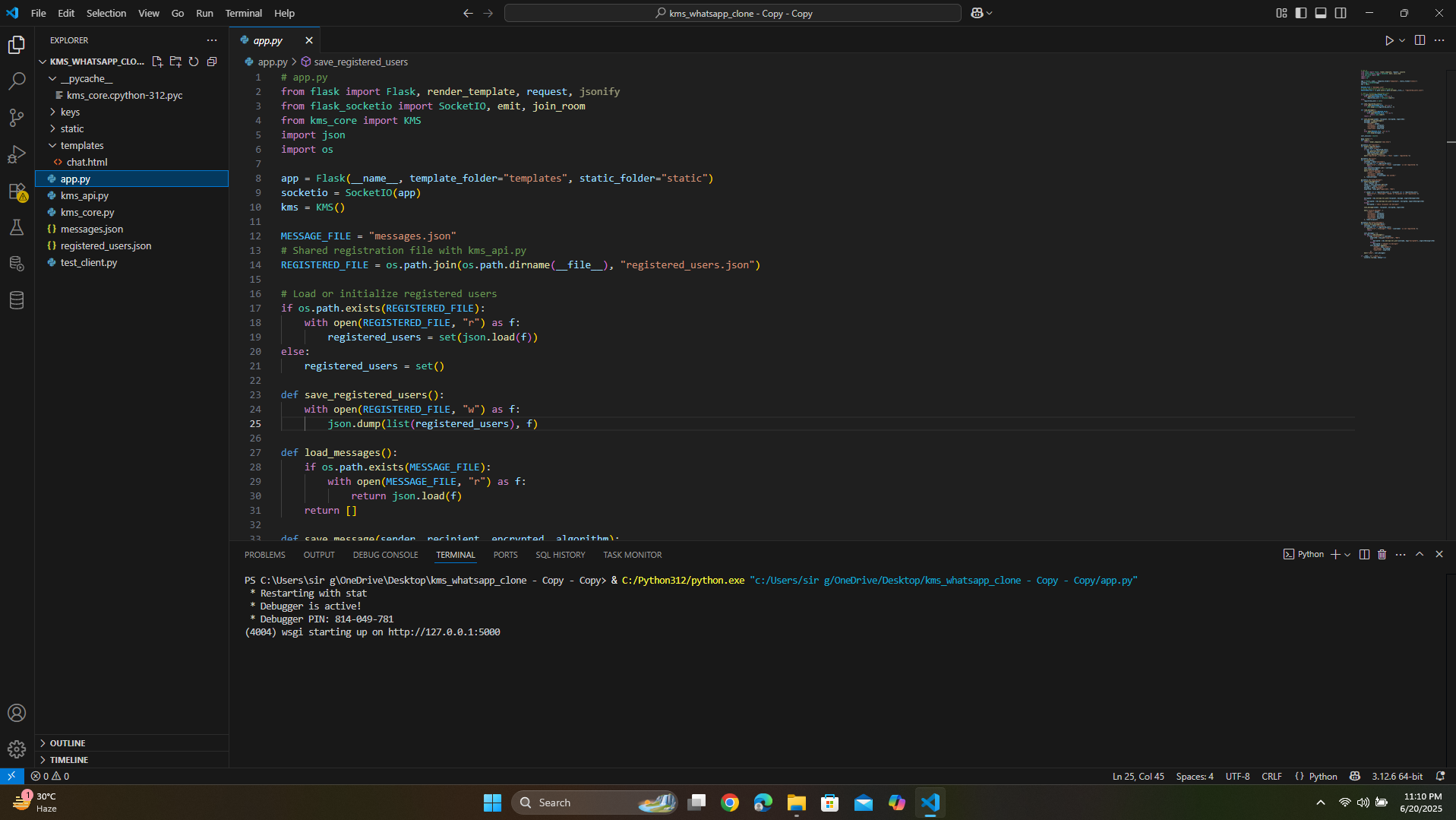
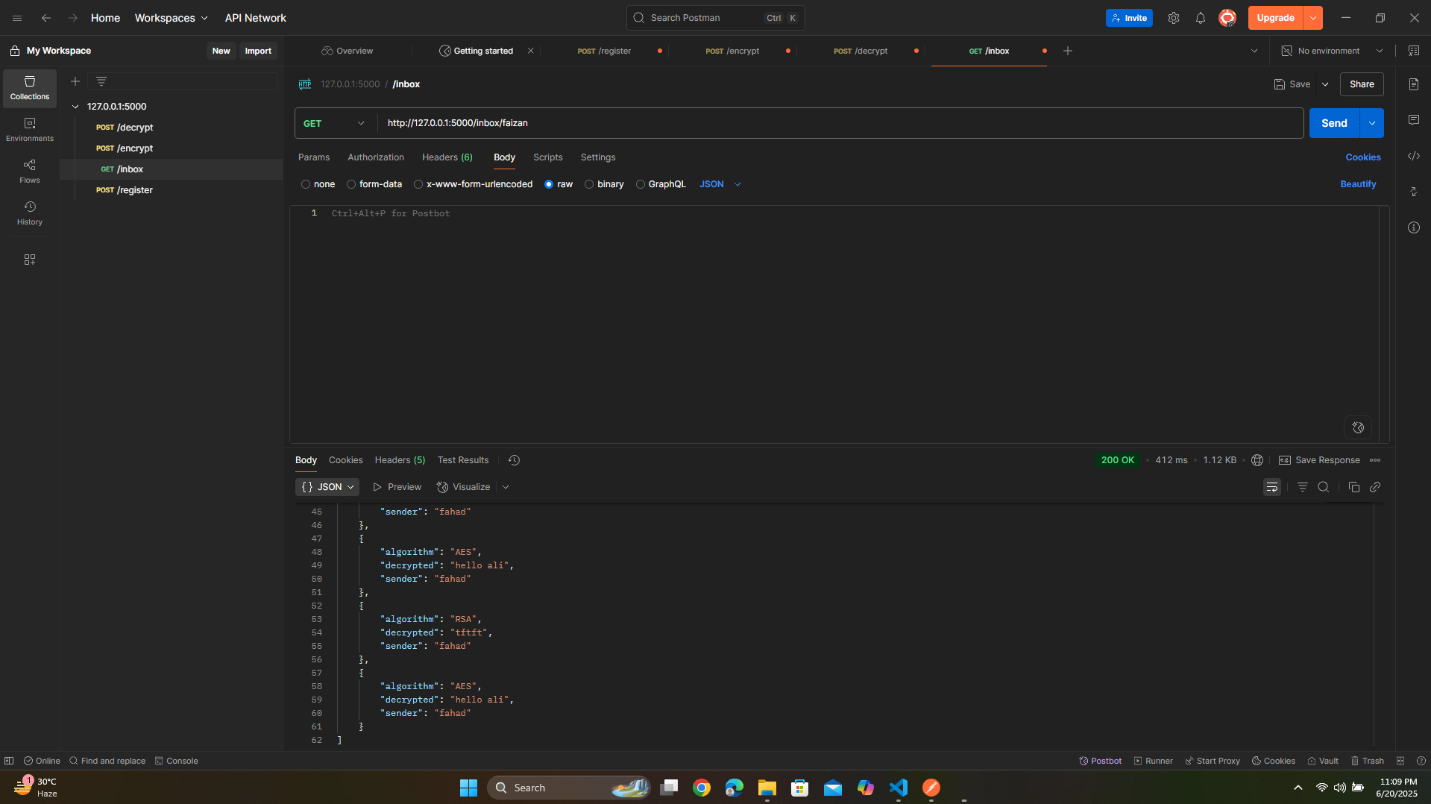
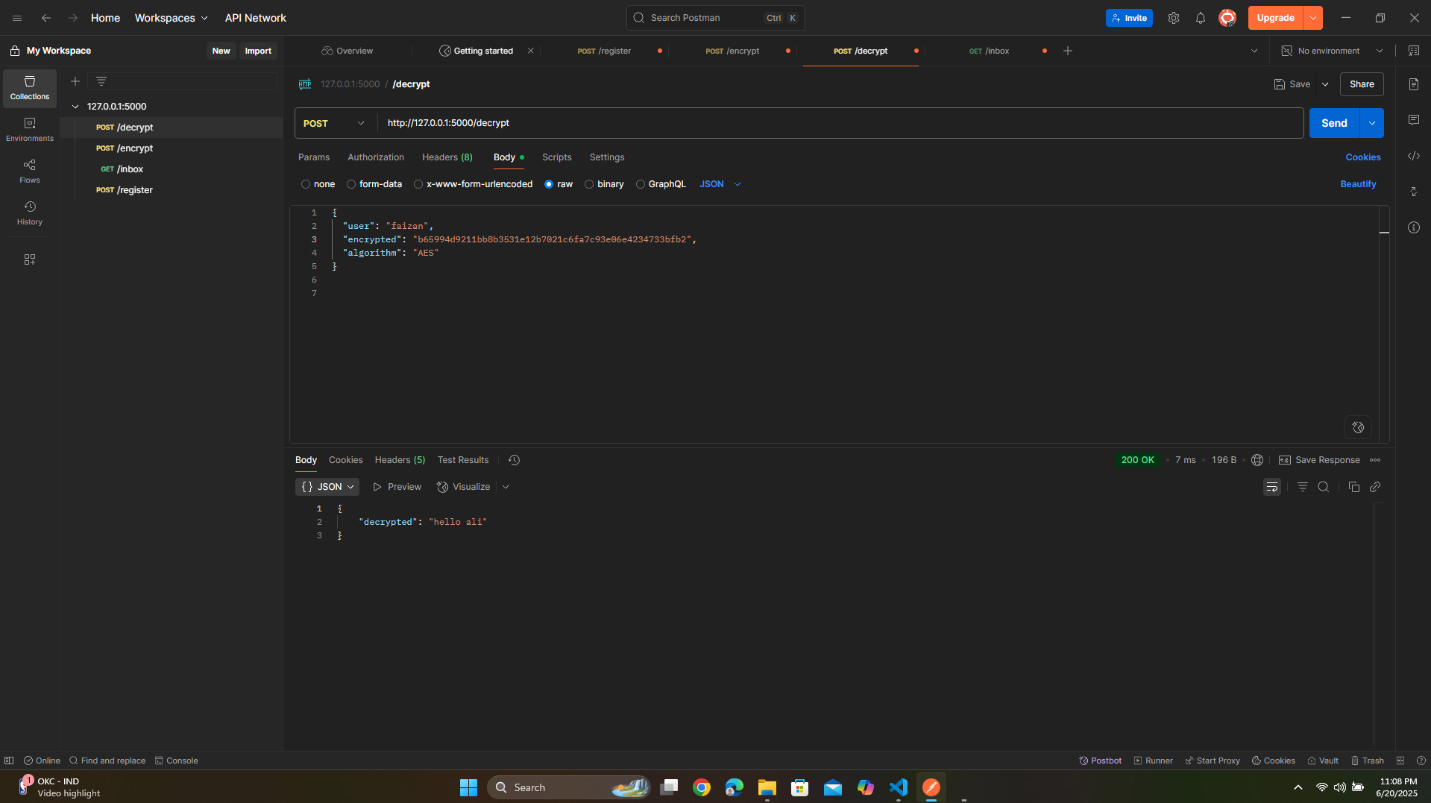
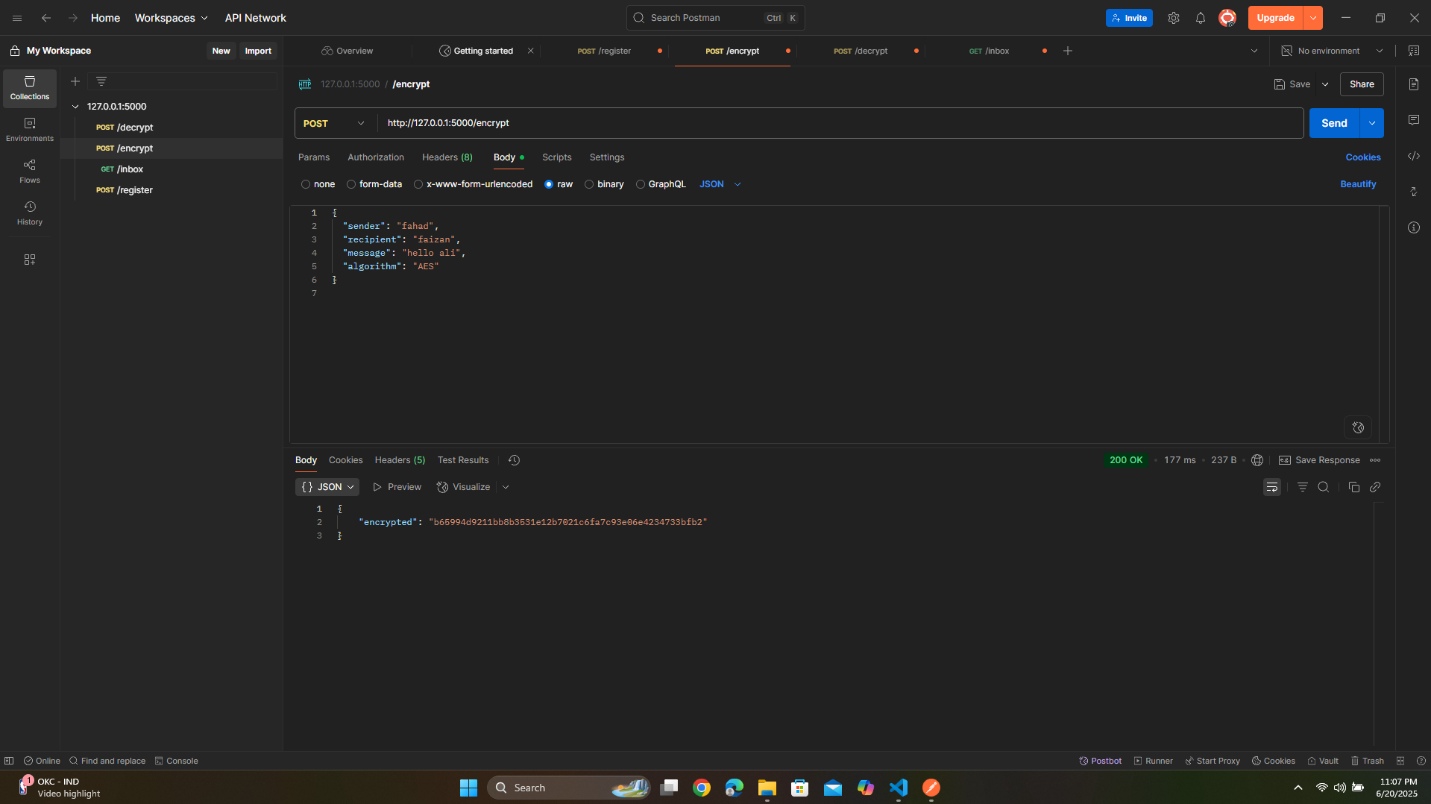
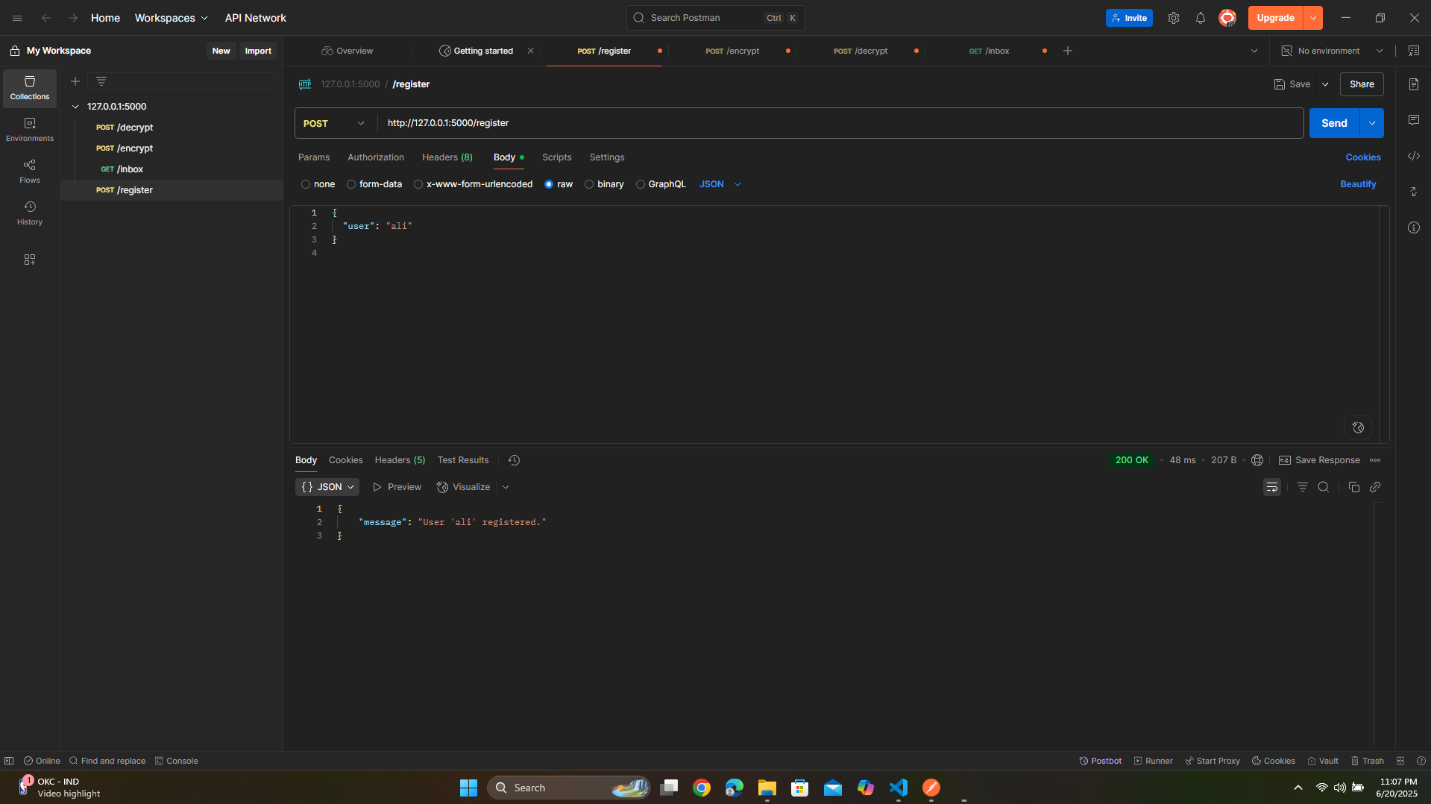
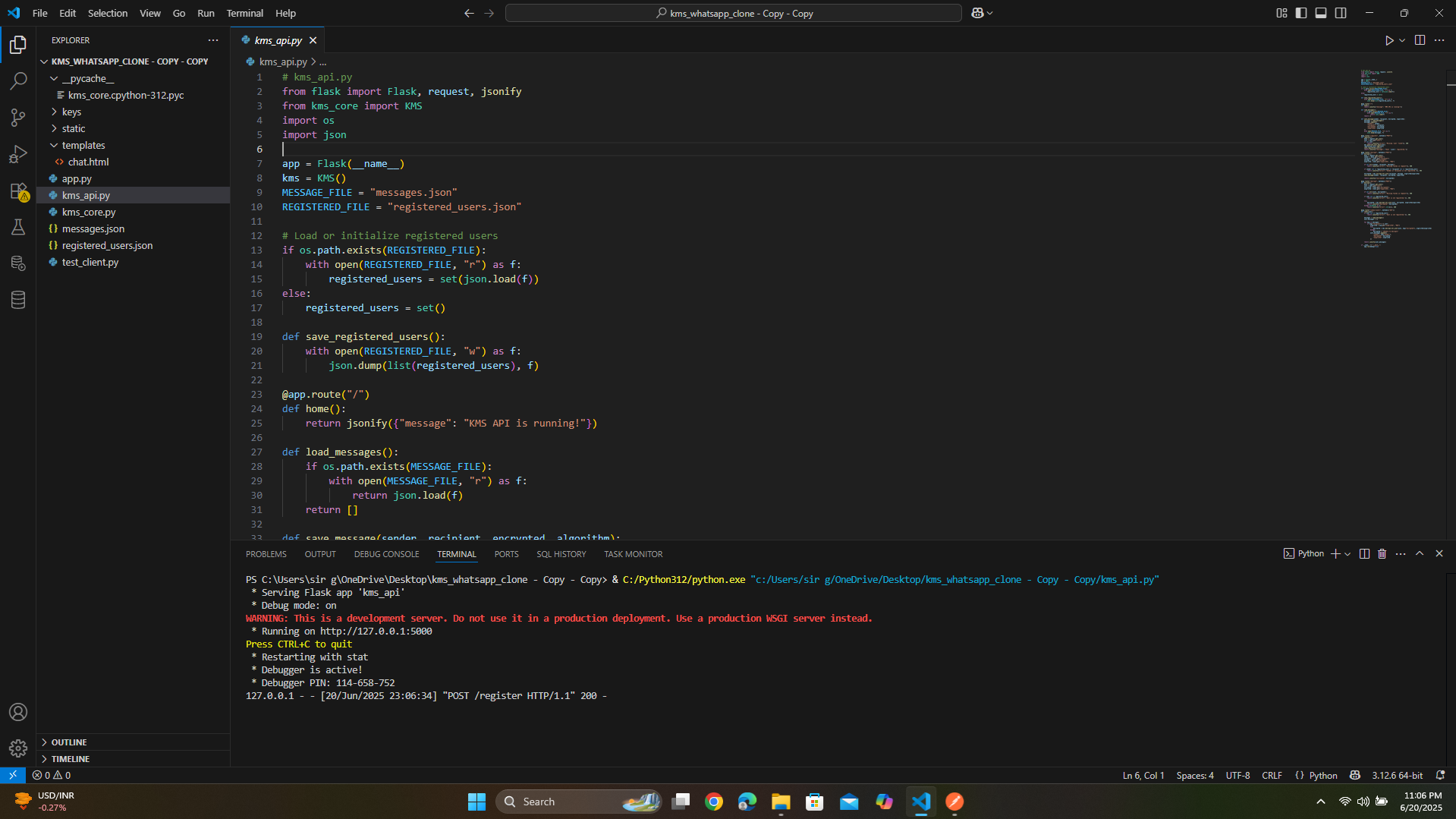
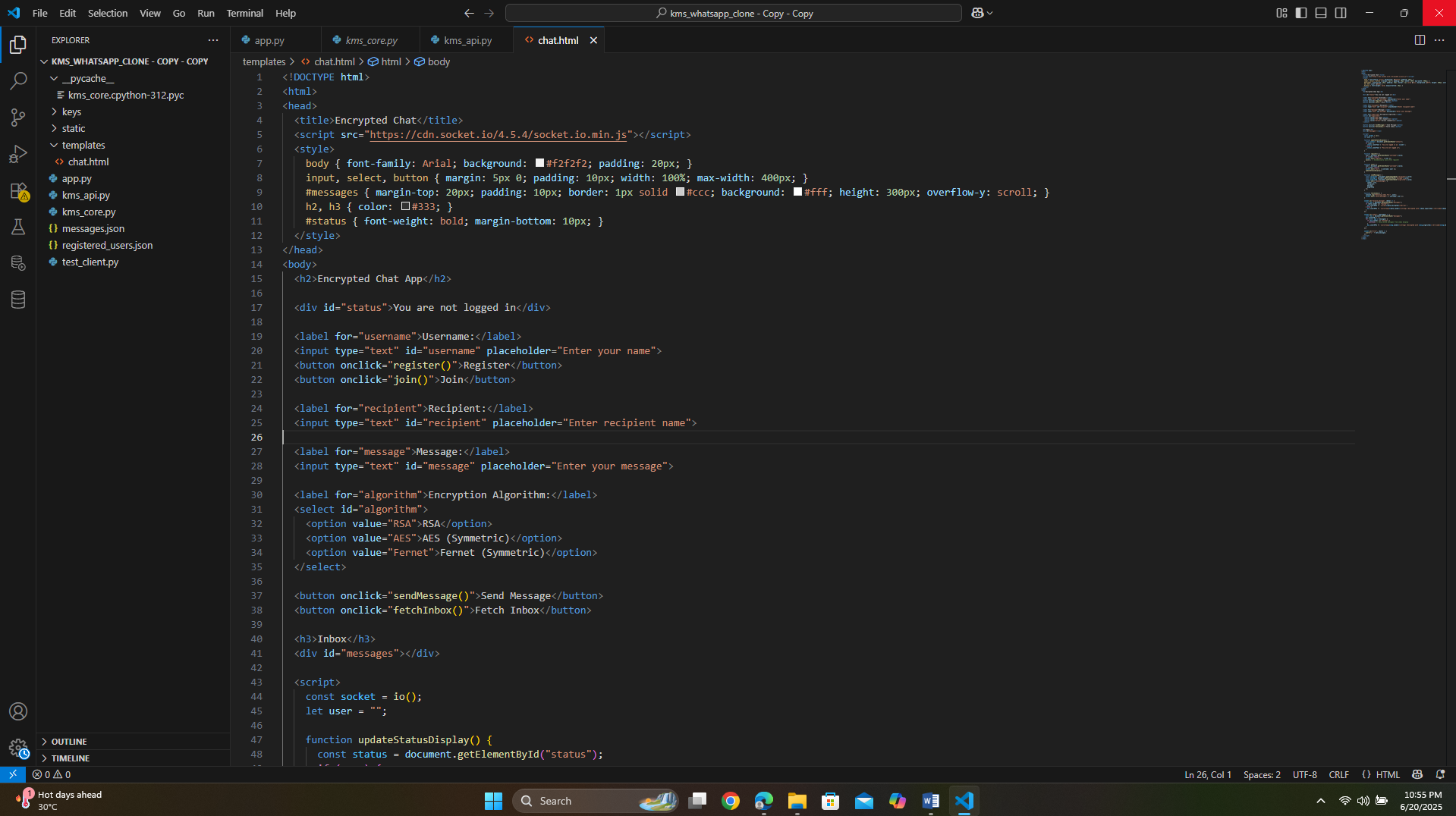
</body>

</html>

**SS OF FILE STRUCTURE**

****

**SS OF OUTPUTS AND RESULTS**

****